Algorithms

CS 341 Lecture Notes (Fall 2019)

Eric Blais

University of Waterloo

Contents

LECTURE 1

Introduction

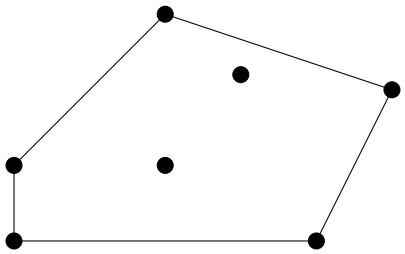
The purpose of CS 341 is to learn how to *design* and *analyze* efficient algorithms. This is one of the most important skills to master for computer scientists. It's also something that can be done simply with pen and paper, and does not require any advanced knowledge of any programming language. To get an idea of how this course will proceed, let's start with an example.[1]

1. First example: the convex hull problem

Our first problem is one of the central problems in computational geometry: the *convex hull* problem.

PROBLEM 1. *Given a set of n points in the plane, find their convex hull—the smallest convex set that contains all n points.*

For example, here is a set of 7 points and their convex hull:



Our goal will be to design an efficient algorithm for solving the convex hull problem. Before we do so, let's talk about the problem itself a little bit more.

First, the convex hull problem statement is rather abstract—after all, which software developer in the real world would ever directly encounter the convex hull problem? This is by design: our goal in this class is to introduce techniques that are useful throughout many different areas of computer science. As such, throughout the course we will only consider problems that are as simple as possible; after this course, you will see that underneath many complex real-world computational problems lie exactly the problems that we see in this class or slight variations of them. (For example, the convex hull problem itself is at the heart of multiple fundamental problems in image processing, optimization problems, mapping applications, etc.)

Second, there is one convention that we used in the problem statement and will use throughout the course: the variable n will be reserved exclusively for the most natural parameter of interest that represents the "size" of the input. Our goal will be to design

[1] Note that this example is provided only to illustrate and motivate the topics that we will see in the rest of the class. You will not be responsible for understanding the details of this example in the assignments or the exam—though you are more than welcome to explore it in more detail if you find it interesting!

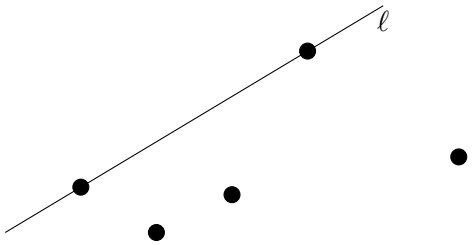
algorithms whose running time is as small as possible as a function of n. (We'll talk about this convention and related topics in the next lecture when we formalize the model of computation a bit more precisely.)

Ok, so now we're ready to tackle our challenge: Can you design an efficient algorithm that solves the convex hull problem?

1.1. A first algorithm. The first step in addressing all the algorithm challenges we will see throughout the course is to start by designing *some* algorithm that solves the problem, and not to worry too much about its efficiency. For the current challenge, we get a good hint about one possible algorithm by considering an alternative definition of the convex hull of a set of points.

DEFINITION 1.1. The *convex hull* of a set X of points is the polygon whose sides are the lines ℓ which

(1) pass through at least 2 of the points in the set X, and
(2) for which no points in X lie on one of the sides of ℓ.



This definition leads us to the following simple algorithm.

Algorithm 1: SIMPLECONVEXHULL(X)

for *every pair of points* $r, s \in X$ **do**
  $\ell \leftarrow$ a line passing through r and s;
  if *every point in* X *lies to one side of* ℓ **then**
    Add ℓ to the convex hull;

Note that this algorithm is presented in rather high-level pseudocode. In the rest of the course, we will usually specify the algorithm in a bit more detail to make sure that we have a clear and precise algorithm, and to make its time complexity explicitly clear. (For instance, how do we define the lines in the algorithm? And, most importantly, how do we test whether all the points in X lie on one side of the line?[2]) But we will never aim for a completely formal definition/implementation of the algorithm: our goal is always to be able to present an algorithm in a way that we can

- verify the *correctness* of the algorithm,
- determine its *running time*, and
- *understand* clearly how it works.

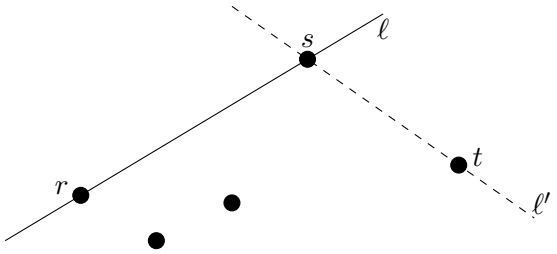
We will discuss all these points again in the next lecture. But for now, can we determine the efficiency of the SIMPLECONVEXHULL algorithm? Assuming that all the operations on pairs of points take constant time and that we use a straightforward implementation of the test in the **if** condition, we have a total runtime of $O(n^3)$.

[2]Hint: Cross products!

THEOREM 1.2. *The* SIMPLECONVEXHULL *algorithm solves the convex hull problem in time* $O(n^3)$.

Can we do better? Yes!

1.2. Jarvis march. If we spend a bit of time with the SIMPLECONVEXHULL algorithm and some examples, we eventually notice a natural improvement: if we have found a line ℓ that goes through r and s in X and is in the convex hull of X, then we can find the next line on the convex hull by "rotating" the line ℓ on the point s until it hits another point t in X.



The algorithm obtained by using this idea is known as *Jarvis' march*, or the *gift wrapping algorithm*.

Algorithm 2: JARVISMARCH(X)

$r \leftarrow$ left-most point in X;
$s \leftarrow$ next point in X obtained by rotating vertical line from r;
Add (r, s) to the convex hull;
while *s is not the left-most point of X* **do**
  $t \leftarrow$ next point in X obtained by rotating line (r, s);
  $(r, s) \leftarrow (s, t)$;
  Add (r, s) to the convex hull;

As before, more details are required to formalize this algorithm. As an exercise, you should verify that finding the left-most point in X *and* finding the point t at each iteration of the while loop can both be accomplished in $O(n)$ time since they are equivalent to the problem of finding the minimum element of a set. The Jarvis march algorithm therefore gives a significant improvement over our simple convex hull algorithm.

THEOREM 1.3. *The* JARVISMARCH *algorithm solves the convex hull problem in time* $O(n^2)$.

In fact, we can even say something stronger: if we let h denote the number of vertices on the convex hull of X, then the time complexity of the JARVISMARCH algorithm is $O(n \cdot h)$.

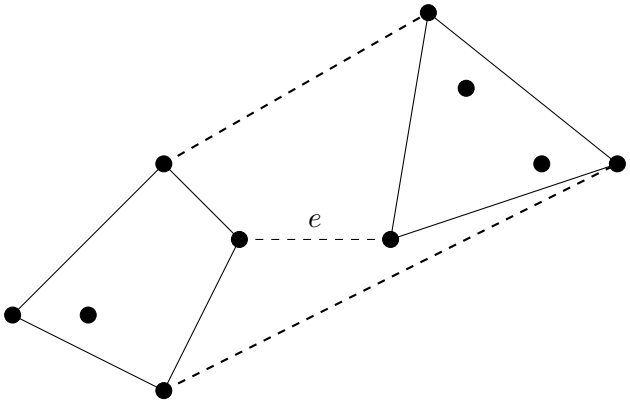
Can we do better? Once again, yes!

1.3. Divide and conquer. A very powerful algorithm design method that we will revisit in this course is *divide and conquer*: split the original problem up into smaller subproblems, solve the subproblems separately, and merge the solutions. This approach works very well for the convex hull problem: we split the set X into two halves X_0 and X_1, find the convex hulls of these subsets, and find the upper and lower *bridges* that are needed to connect the two convex hulls.

Algorithm 3: ConvexHullDC(X)

$X_0 \leftarrow$ left half of X;
$X_1 \leftarrow$ right half of X;
$H_0 \leftarrow$ ConvexHullDC(X_0);
$H_1 \leftarrow$ ConvexHullDC(X_1);
$(ub, lb) \leftarrow$ upper and lower bridges of H_0, H_1;
return merged convex hull of H_0 and H_1 using ub, lb;

As in our earlier example, more details are needed to complete the algorithm and are left as an exercise. The step of merging the convex hulls using the upper and lower bridges is straightforward, but finding the bridges is a little bit more subtle. This can be done by starting with the edge e connecting the right-most point in X_0 to the left-most point in X_1 and "walking" the edge up and down as far as possible.



Once we complete the algorithm, we see that finding the upper and lower bridges and merging the convex hull can be done in linear time. The time complexity of the ConvexHullDC algorithm therefore satisfies the recurrence relation

$$T(n) = 2\,T(\tfrac{n}{2}) + O(n).$$

This is the same recurrence relation as for merge sort, so we end up with a final time complexity of $O(n \log n)$.

Theorem 1.4. *The* ConvexHullDC *algorithm solves the convex hull problem in time* $O(n \log n)$.

Can we do better? Yes and no.

1.4. Other algorithms. Having gone from an initial algorithm that solves the convex hull problem in time $O(n^3)$ to one that has time complexity only $O(n \log n)$, it's natural to ask if we can improve that result even a little bit more, ideally to obtain an algorithm that runs in time $O(n)$. This is not possible in any setting where sorting takes time $\Omega(n \log n)$:

Theorem 1.5. *In any setting where sorting* n *numbers takes time* $\Omega(n \log n)$*, then every algorithm for solving the convex hull problem also has time complexity* $\Omega(n \log n)$.

Proof. The theorem can be established using the idea of *reduction*: we show that an efficient convex hull algorithm can also be used to sort numbers efficiently. The idea of this argument is that if wish to sort a list of n numbers $a_1, a_2, \ldots, a_n$, we can do this by

(1) Creating the set $X = \{(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)\}$,
(2) Finding the convex hull of X,
(3) and extracting the sorted list of numbers by following the convex hull of X.

Steps 1 and 3 can be shown to both take time $O(n)$, so if we have a convex hull algorithm with complexity $o(n \log n)$, we would also achieve the same runtime for sorting as well. □

This theorem shows that, in a sense, the ConvexHullDC algorithm is optimal. But in fact we can push our exploration of the problem even further. What if we assume that the points in X are not stored in an arbitrary order but instead are sorted according to their first coordinate? In that case, the convex hull problem can be solved more efficiently.

THEOREM 1.6. *The convex hull problem can be solved in time $O(n)$ when the set X of input points is sorted according to the first coordinate.*

The proof of this theorem is left as a challenge. A small hint is that you may want to consider an algorithm that finds the upper and lower parts of the convex hull separately. (If you want to read more on the algorithms that achieves this bound, you can also search for the *Graham scan algorithm* or *Andrew's monotone chain algorithm.*)

Another natural question is raised by comparing the bounds of the JARVISMARCH and CONVEXHULLDC algorithms: the refined analysis of the former shows that it runs in time $O(nh)$ where h is the number of points on the convex hull, whereas the latter runs in time $O(n \log n)$ regardless of the value of h. Thus, the algorithms are in some sense incomparable: for some inputs—those corresponding to sets with convex hull that hit $o(\log n)$ points—the JARVISMARCH algorithm is better; for other sets, the CONVEXHULLDC algorithm is much faster. Is there a better algorithm that does at least as well as both of these algorithms for all inputs? Yes!

THEOREM 1.7. *The convex hull problem can be solved in time $O(n \log h)$ when the set X of n input points has a convex hull of size h.*

The proof of this theorem is again left as a challenge. Interested readers can find more details about the solution to the challenge by searching for the *Kirkpatrick–Seidel ultimate convex hull algorithm* or *Chan's algorithm.*

2. Overview of the class

2.1. Goal. By the end of this class, you should be able to tackle a wide variety of computational problems just like we did for the Fibonacci sequence above so that you can *analyze* the time complexity of algorithms, *design* better algorithms in many cases, and *recognize* when some problems are intractable.

2.2. Topics. The topics we will cover to help you achieve this goal are grouped as follows:

(1) Analysis of algorithms
(2) Greedy algorithms
(3) Dynamic programming
(4) Graph search algorithms
(5) NP-completeness

2.3. Resources. All the course information and relevant links are found on the course web page

`www.student.cs.uwaterloo.ca/~cs341`

The lecture notes and tutorial notes will be posted throughout the term on that page. We will also use Piazza for discussions.

The main textbook for the class is Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms* (3rd edition). It is not required but is highly recommended. Another great alternative textbook that is also highly recommended is *Algorithms* by Dasgupta, Papadimitriou, and Vazirani. There are also other alternatives that are on reserve at the library; see the website for details.

2.4. Marking scheme. The marks breakdown for the class is as follows:

- 35%: Weekly assignments
- 25%: Midterm
- 40%: Final exam

All sections of the class have the same assignments, midterm, and final exam. Two of the assignments are programming assignments; the rest are written assignments.

2.5. How to succeed in CS 341. There are a few tricks that will help you succeed in CS 341 and master all the tools and techniques we introduce in the class.

- Bring pen and paper to class, and nothing else. Lecture notes will be provided so you should not worry about taking complete notes in class. But there will be many pauses in the lectures where you will be asked to think about a problem before the answer is revealed. The best way to do this will be in pairs, with pen and paper. Doing so actively (whether you do find the answer or not) will help you learn the material much more effectively than if you're just here to listen.
- Complete the homeworks *by yourself*. This will be the single most important factor determining your success in the course.
- Attend tutorial sessions. This is a new component for CS 341. The idea is to have extra practice at solving problems, on top of the assignments, in a setting where you can also get guidance and support from IAs. Take advantage of them!
- Ask questions! We are here to help. You are encouraged to discuss the material with other students, to ask questions on Piazza, to visit instructors or TAs during office hours, etc.

LECTURE 2

Analysis of algorithms

In this lecture, we go back to the beginning: we will define exactly what we mean by *algorithms* and how we measure their time complexity. We then introduce the important idea of *reductions* via the 2Sum and 3Sum problems.

1. Algorithms and the computational model

To make our exploration of the efficiency of algorithms more precise, we need to first establish some fundamental definitions. Let's start at the very beginning: what is an algorithm?

Definition 2.1 (Algorithm). An *algorithm* is the description of a process that is:

- effective, (we can carry out each step, or basic operation, of the process)
- unambiguous, (there is no room for interpretation of the steps)
- and finite (we can write it down with a finite number of lines)

which takes some *input* and halts and generates some *output* after a finite number of steps.

A formal description of an algorithm starts with a list of the elementary operations (or steps) that the algorithm can carry out. We will not do so explicitly very often in this course: almost everywhere, the set of elementary operations will be the same as the ones you have seen in CS 240 (reading or writing to a specific index in an array, adding/subtracting/multiplying numbers, comparing two numbers, applying logical operators, etc.). And we describe the algorithms informally using pseudocode. This will allow us to specify the algorithm precisely enough that we can analyze it, but without so much implementation detail that it obscures the main ideas of the algorithm.

Before examining the time complexity of algorithms, we will want to make sure that the algorithm actually does what we want it to. Formally, this means that we first define a *problem* by specifying the instances of the problem (which correspond to the inputs of an algorithm) and the valid solutions for each instance. Then we can formally define what we mean when we say that an algorithm "solves a problem".

Definition 2.2 (Solving a problem). An algorithm *solves* a problem P if for every instance I of the problem P, when the algorithm receives I as input and is run, it outputs a valid solution to P.

In this class, it's not enough to identify algorithms that solve a given problem; we also want to measure their *time complexity*. To do so, we must first define the model of computation that we consider. There are two natural options.

Line-cost model: Each execution of a line of an algorithm takes 1 time step.

Bit-cost model: Operations on a *single bit* take 1 time step.

Neither of these options is ideal. The line-cost model is perfectly reasonable in many cases, but not always. For example, consider the simple algorithm that follows. According

Algorithm 4: TOWER(n)

```
k ← 1;
for i = 1, ..., n do
    k ← 2^k;
return k;
```

to the line-cost model, this algorithm runs in time $O(n)$. But the value that it returns is

$$\textsc{Tower}(n) = \underbrace{2^{2^{\cdots^2}}}_{n},$$

a number that requires way (way, way) more than n bits even to write down, so the assumption that the line inside the for loop runs in 1 time step does not correspond in any way to reality.

The bit-cost model suffers from a different problem: it quickly gets too complicated. A line of code might consist of multiplying two positive integers a and b. What is the time complexity of this operation? For starters, $\log a$ and $\log b$ bits are required to encode the two integers in general. And we can as we will see in later lectures, the most efficient algorithm for multiplying integers is far from trivial and has time complexity

$$O((\log a)(\log b)^{0.59}).$$

In some settings, it is important to account for this complexity accurately; in many other cases, however, keeping track of this detailed complexity simply makes it harder to understand what is really going on.

The model that we will end up using for this class is in between the two: it is known as the *word random-access-memory (RAM)* model.

DEFINITION 2.3 (Word RAM model). The *Word RAM* model is the computational model in which for an algorithm run on an input of size n,

- the memory of the algorithm is broken up into *words* of length w (typically, $w = \lceil \log n \rceil$), and
- any elementary operation (read, write, add, multiply, AND, etc.) on any single word in memory takes 1 time step.

The Word RAM model balances the needs of our model to be realistic (so that algorithms that we show to be faster than alternatives really are in practice) while still being simple (so that we can actually measure the time complexity of algorithms within the model).

As a general rule of thumb, for most of this class the time complexity of an algorithm according to the word RAM model will correspond exactly to the line-cost model. There will be a few situations, however, where we will refer back to this definition when the notion of an "elementary operation" is not completely obvious.

2. Big-O notation

We are now in a position to measure the time complexity of an algorithm on a specific input. But what we really want to do is define a *global* measure of the time complexity of the algorithm. We do this using *worst-case time complexity*.

DEFINITION 2.4 (Worst-case time complexity). The *worst-case time complexity* of an algorithm A is the function $T_A : \mathbb{N} \to \mathbb{N}$ obtained by letting $T_A(n)$ be the maximum time complexity of A over any input of size n.

REMARK 2.5. Worst-case is not the only way we can measure the time complexity of an algorithm as a function of its input size. (One could take the *average* time complexity over some distribution on inputs of length n instead, for example). But this is a model that is particularly useful and the one we will focus on throughout this course.

We now come to one of the central ideas in the analysis of algorithms: what we care about, when we measure the time complexity of an algorithm, is not the *exact* expression for this time complexity, but rather its *asymptotic* rate of growth as the inputs get larger. This is best done using big-O notation.

DEFINITION 2.6 (Big-O notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 1}$ satisfy $f = O(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \leq c\, g(n)$.

REMARK 2.7. Here and throughout,

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ is the set of natural numbers,
- $\mathbb{R}^+$ be the set of positive real numbers, and
- $\mathbb{R}^{\geq 1}$ be the set of real numbers that have value at least 1.

Big-O notation is so useful in part because it lets us simplify even very complicated expressions and only worry about the "most significant" aspects of time complexity. For example, we have the following fact.

PROPOSITION 2.8. *The function* $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ *satisfies* $f = O(n^7)$.

PROOF. For every $n \geq 4$,

$$f(n) = 4n^7 + 100n^3 + \tfrac{1}{3}n^2 + \pi \leq 4n^7 + 100n^3 + n^2 + n \leq 106n^7$$

so $f = O(n^7)$ by the definition with $c = 106$ and $n_0 = 4$. □

One word of warning: since $n^7 \leq n^{100}$ for every $n \geq 1$, it is also correct to say that the function f defined in the proposition satisfies $f = O(n^{100})$. To describe asymptotics of a function more precisely, we need the big-Ω and the big-Θ notation. The big-Ω notation is used to give lower bounds on the asymptotic growth of a function.

DEFINITION 2.9 (Big-Ω notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Omega(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \geq c\, g(n)$.

The big-Θ notation is used to show that we have matching upper and lower bounds on the asymptotic growth of a function.

DEFINITION 2.10 (Big-Θ notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

We can use this notation to strengthen our last proposition.

PROPOSITION 2.11. *The function* $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ *satisfies* $f = \Theta(n^7)$.

PROOF. We have already seen that $f = O(n^7)$. We also have that for every $n \geq 1$, $f(n) \geq 4n^7$ so by definition $f = \Omega(n^7)$ (with $n_0 = 1$ and $c = 4$) and, therefore, $f = \Theta(n^7)$ as well. □

There are also situations where we want to argue that a function grows *asymptotically slower* than another reference function. We can state this formally with the little-o notation.

DEFINITION 2.12 (little-o notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = o(g)$ if for every $c \in \mathbb{R}^+$, there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) < c\, g(n)$.

REMARK 2.13. We also have that $f = o(g)$ whenever $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. This characterization can be easier to work with when the limit exists; feel free to do so.

Similarly, we can say that a function f grows asymptotically *faster* than another function g using the little-ω notation.

DEFINITION 2.14 (little-ω notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \omega(g)$ if and only if $g = o(f)$.

We will be using this notation extensively throughout the course, so it is critical that you are all very comfortable working with it. We will be covering some exercises related to this notation in this week's tutorial and in the first assignment, but you can also find much more information about it and many other exercises to work on in the textbook.

Part 1

Divide & Conquer

LECTURE 3

Solving recurrences

In this lecture, we examine two of the most powerful tools in an algorithm designer's kit: reduction and recursion.

1. The 2SUM problem

Let's examine a simple problem.

DEFINITION 3.1 (2SUM). Given an array $A \in \mathbb{Z}^n$ of n integers and an integer $m \in \mathbb{Z}$, return a pair of indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] = m$ if any such pair exists, and $\perp$ if no such pair exists.

The following algorithm solves the 2SUM problem.

Algorithm 5: SIMPLE2SUM(A, m)

```
for i = 1, ..., n do
    for j = i, ..., n do
        if A[i] + A[j] = m return (i, j);
return ⊥;
```

The claim that SIMPLE2SUM solves the 2SUM problem requires a proof, but that proof is very simple: if there exist k, ℓ for which $A[k]+A[\ell] = m$, then we also have that $A[\ell]+A[k] = m$ so in the iteration of the inner loop with $i = \min\{k, \ell\}$ and $j = \max\{k, \ell\}$, the **if** test will be satisfied and the algorithm returns a valid solution. And if no such k, ℓ pair exists, then the **if** test is never satisfied, so the algorithm always returns $\perp$.

The time complexity of the SIMPLE2SUM algorithm is also established with a simple argument, but we have to be a little bit careful if we want the argument to hold in the Word RAM model as specified in the last lecture. In this problem, what we really care about in terms of time complexity for this problem is the number of basic operations (addition, incrementing, and comparison) on integers. We do this in the Word RAM model by setting the width w of words to be large enough that each integer in a problem instance can be stored with w bits.

PROPOSITION 3.2. *The* SIMPLE2SUM *algorithm has time complexity* $\Theta(n^2)$ *in the Word RAM model when the word size* w *is large enough that each integer in the input array* A *fits in a single word.*

PROOF. Recall that we measure the time complexity of our algorithms in the worst-case setting. This worst-case is attained on inputs that return $\perp$, since for these inputs the two for loops run to completion without interruption. In this case, the number of additions

performed by the algorithm is

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}.$$

For every $n \geq 2$, we have $\frac{n^2}{4} \leq \frac{n(n+1)}{2} \leq n^2$ so the algorithm performs $\Theta(n^2)$ additions and the same bound also holds for the total number of increments and compare operations. □

Can we do better? Yes! Let's break down what the algorithm is doing in detail. For each index $i \in \{1, 2, \ldots, n\}$, the inner loop is trying to determine if there is an integer $A[j]$ in A for which $A[i] + A[j] = m$, or, to rephrase the same statement: if the integer $m - A[i]$ is in the array.

As it turns out, we know a very efficient method for determining whether an integer (such as $m - A[i]$) is in an array: binary search! This, of course, only works when the array is sorted, but that's something we can do in the algorithm as well. The resulting algorithm is as follows.

Algorithm 6: 2Sum(A, m)

$B \leftarrow$ Sort(A);
for $i = 1, \ldots, n$ **do**
  $j \leftarrow$ FindBinarySearch$(B, m - B[i])$;
  if $j \neq \perp$ then **return** $\big($Find$(A, B[i])$, Find$(A, B[j])\big)$;
return $\perp$;

What we have just done looks simple, perhaps even obvious, yet it is a great example of an extremely powerful algorithmic technique: reduction. Specifically, we *reduced* the 2SUM problem to the problem of finding a given integer in an array. And since we already know how to solve the latter problem, we can use the algorithm for that problem to solve 2SUM as well.

Reduction idea: Use known algorithms to solve new problems.

We'll see many other examples where the reduction technique proves very useful—and it will even return with a starring role in the last module of this class, when we tackle NP-completeness. But for now, let's continue our work and establish the correctness and time complexity of the 2Sum algorithm.

Proposition 3.3. *The* 2Sum *algorithm solves the 2SUM problem.*

Proof. Consider first the case where there exist indices k, ℓ for which $A[k] + A[\ell] = m$. Then there exist indices k', ℓ' for which $B[k'] + B[\ell'] = m$. We can rewrite the identity as $B[\ell'] = m - A[k']$, so for the iteration of the for loop where $i = k'$, the integer $m - B[i]$ is in B and the FindBinarySearch algorithm returns ℓ' or another index ℓ'' for which $B[\ell''] = B[\ell']$ and 2Sum returns indices i^*, j^* for which $A[i^*] + A[j^*] = B[k'] + B[\ell'] = m$.

Consider now the case where for every indices i, j we have $A[i] + A[j] \neq m$. Then it is also true that for every indices i, j we have $B[i] + B[j] \neq m$ or, equivalently, $B[j] \neq m - B[i]$. So for each value of i, the integer $m - B[i]$ is not in B, FindBinarySearch returns $\perp$, and so does 2Sum. □

PROPOSITION 3.4. *When* SORT *has time complexity* $O(n \log n)$*,* FINDBINARYSEARCH *has time complexity* $O(\log n)$*, and* FIND *has time complexity* $O(n)$*, the* 2SUM *algorithm has time complexity* $O(n \log n)$*.*

PROOF. The initial call to SORT has time complexity $\Theta(n \log n)$. The FINDBINARYSEARCH algorithm is called at most n times for total time complexity $n \cdot O(\log n) = O(n \log n)$, and the FIND algorithm is called at most twice, for an additional time complexity $O(n)$. □

Can we do even better? It takes $\Theta(n)$ time just to read the integers stored in the array A, so the best we can probably hope for is to remove the extra $\log n$ term in our runtime. But instead of trying to do this, let's see if we can use the Reduction idea to solve even more complex problems.

2. The 3SUM problem

DEFINITION 3.5 (3SUM). Given an array $A \in \mathbb{Z}^n$ of n integers and an integer $m \in \mathbb{Z}$, return three indices $i, j, k \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] + A[k] = m$ if any such triple exists, and $\perp$ if no such triple exists.

We can again define a simple algorithm to solve the 3SUM problem.

Algorithm 7: SIMPLE3SUM(A, m)

```
for i = 1, ..., n do
    for j = i, ..., n do
        for k = j, ..., n do
            if A[i] + A[j] + A[k] = m return (i, j, k);
return ⊥;
```

This algorithm is not so efficient. It has time complexity $\Theta(n^3)$, and we would like to do better. We can again use the Reduction technique to find a better algorithm. One option is to again try to reduce to the FIND problem. Or, since we just designed an efficient 2SUM algorithm, perhaps we can reduce 3SUM to 2SUM instead? Let's find out by again fixing an index i and seeing what the two inner loops are doing. They're trying to find out if there are indices j, k for which

$$A[i] + A[j] + A[k] = m \quad \iff \quad A[j] + A[k] = m - A[i].$$

This is an instance of the 2SUM problem!

Algorithm 8: 3SUM(A, m)

```
for i = 1, ..., n do
    (j, k) ← 2SUM(A, m − A[i]);
    if (j, k) ≠ ⊥ then return (i, j, k);
return ⊥;
```

Notice how simple the resulting algorithm is. As a bonus, the analysis of the time complexity of this algorithm is also very simple.

THEOREM 3.6. *The 3SUM problem can be solved by an algorithm with time complexity* $\Theta(n^2 \log n)$.

PROOF. When the 3SUM algorithm calls the algorithm 2SUM with time complexity $\Theta(n \log n)$ that we designed in the previous section, its total time complexity is $n \cdot \Theta(n \log n) = \Theta(n^2 \log n)$. □

Can you do even better? With some work, you may be able to design an algorithm that solves the 3SUM problem in time $\Theta(n^2)$. Determining whether we can do even better is one of the major open problems in algorithms research today: it was only very recently that researchers were able to show that 3SUM can be solved by an algorithm with time complexity $o(n^2)$, and whether or not there is an algorithm that solves 3SUM with time complexity $O(n^\gamma)$ for some $\gamma < 2$ remains unknown.

3. Recurrence trees

Recursion is a special type of reduction, where we reduce the original problem to the *same* problem, but on a smaller input. This is a powerful technique, as you have already seen in previous classes when considering the MERGESORT algorithm:

Algorithm 9: MERGESORT(A)

```
if n = 1 return;
MergeSort(A[1, ..., ⌊n/2⌋]);
MergeSort(A[⌊n/2⌋ + 1, ..., n]);
Merge(A[1, ..., ⌊n/2⌋], A[⌊n/2⌋ + 1, ..., n]);
```

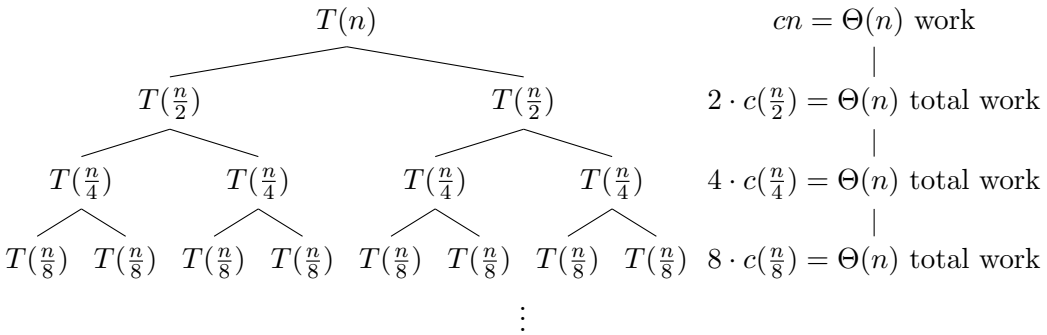
When MERGE is an algorithm that merges the two input arrays in time $\Theta(n)$, what is the time complexity of MERGESORT? If we let $T(n)$ denote the time complexity of the algorithm on arrays with n entries, we find that

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n).$$

To simplify the analysis, let's assume that n is a power of 2. Then we have

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n).$$

To determine the solution to this recursion, the most natural approach is to draw the recursion tree and write down how much time is spent on each level of the tree:



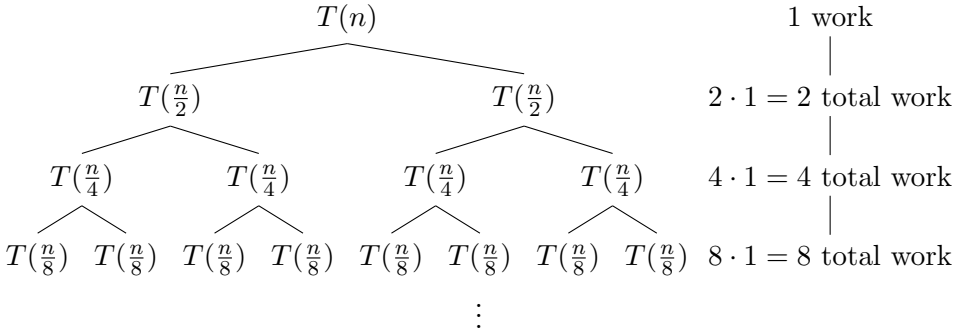
The recursion tree has depth $\log n$ and $\Theta(n)$ time is spent at each level of the tree on completing the merges. The total time complexity of the MergeSort algorithm is therefore

$$T(n) = \Theta(n \log n).$$[1]

Many other recursions can also be solved using the same recursion tree approach. For example, what if we (magically) allowed the Merge algorithm to run in a single unit of time? Then the recursion defining the time complexity of the MergeSort algorithm would be

$$T(n) = 2\,T(\tfrac{n}{2}) + 1$$

and the recursion tree would now look like this:



This tree again has depth $\log n$, so the total time complexity in this case is

$$T(n) = 1 + 2 + 4 + 8 + \cdots + \tfrac{n}{2} + n = 2n - 1 = \Theta(n).$$

One more variant: what if we proceed recursively but now divide the array in three instead of two pieces?

Algorithm 10: TriMergeSort(A)

```
if n = 1 return;
TriMergeSort(A[1, ..., ⌊n/3⌋]);
TriMergeSort(A[⌊n/3⌋ + 1, ..., ⌊2n/3⌋]);
TriMergeSort(A[⌊2n/3⌋ + 1, ..., n]);
Merge(A[1, ..., ⌊n/3⌋], A[⌊n/3⌋ + 1, ..., ⌊2n/3⌋], A[⌊2n/3⌋ + 1, ..., n]);
```

Let Merge again be an algorithm with time complexity $\Theta(n)$. When n is a power of 3, the time complexity of the TriMergeSort algorithm is

$$T(n) = 3\,T(\tfrac{n}{3}) + \Theta(n).$$

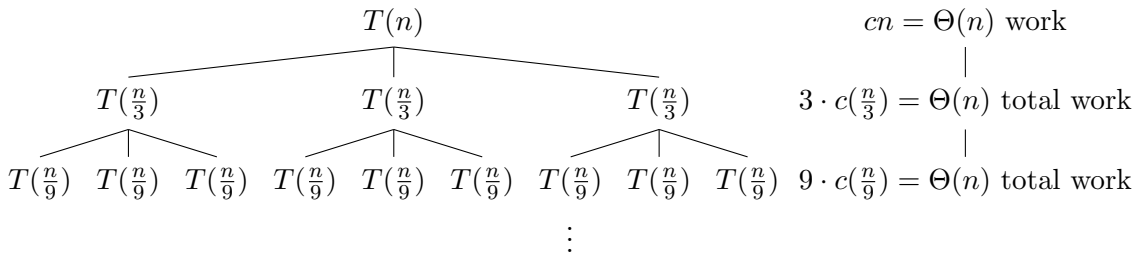
The recursion tree for this recurrence is:

The tree has depth $\log_3 n$, so the time complexity of TriMergeSort is

$$T(n) = \Theta(n \log_3 n) = \Theta(n \log n),$$

the same (asymptotically) as MergeSort!

[1] Here and throughout this course, logarithms without subscripts are over base 2.



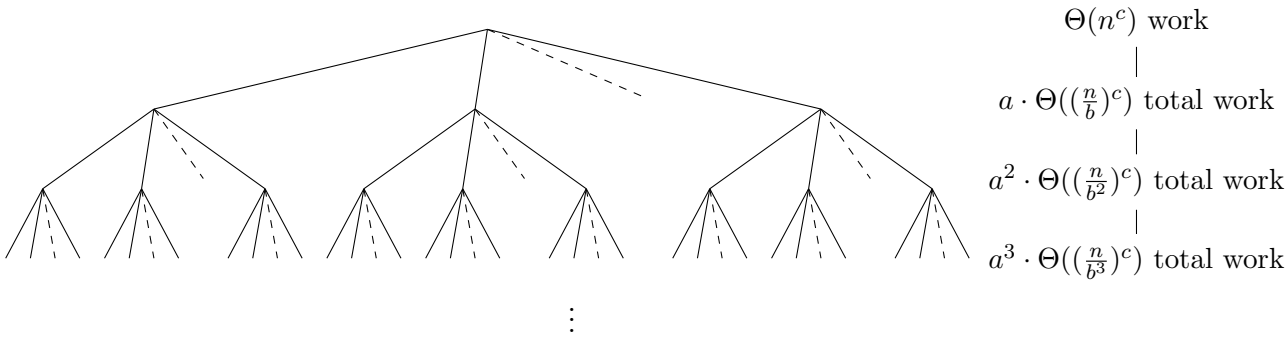
4. Master theorem

We could continue exploring various different examples using the recursion tree method. But let's be a bit more ambitious and try to consider a general scenario that captures many individual examples at once. Let T be defined by the recurrence

$$T(n) = aT(\tfrac{n}{b}) + \Theta(n^c)$$

for some constants $a > 0$, $b > 1$, and $c \geq 0$.

We can again use the recursion tree approach to solve for T in this case.



We can therefore express T as

$$T(n) = \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n}\right) \cdot \Theta(n^c).$$

To determine T, we must simply understand the geometric sequence with ratio $\frac{a}{b^c}$. There are three cases to consider.

Case 1. When $\frac{a}{b^c} = 1$, then each term in the sequence is 1 and so

$$T(n) = \log_b(n) \cdot \Theta(n^c) = \Theta(n^c \log n).$$

Case 2. When $\frac{a}{b^c} < 1$, then the geometric series $1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n} = \Theta(1)$ is a constant and so

$$T(n) = \Theta(n^c).$$

Case 3. When $\frac{a}{b^c} > 1$, then we have an increasing geometric series that is dominated by the last term, so that

$$T(n) = \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n}\right) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}).$$

This result establishes what is known as the *Master Theorem*.

THEOREM 3.7 (Master theorem). *If* $T(n) = aT(\frac{n}{b}) + \Theta(n^c)$ *for some constants* $a > 0$, $b > 1$, *and* $c \geq 0$, *then*

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } c > \log_b a \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } c < \log_b a. \end{cases}$$

What is most important to remember of this theorem is not the statement of the theorem itself but rather the method that we used to obtain it: with the recursion tree method, you can easily recover the Master Theorem itself and also solve recursions that are not of the form covered by the theorem.

5. Geometric series

The different cases of the Master Theorem were handled rather quickly in the section above. It's worth slowing down a bit to see exactly how the asymptotic results about geometric series are obtained. The starting point for those results is the fundamental identity for geometric series.[2]

FACT 1. *For any* $r \neq 1$ *and any* $n \geq 1$,

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}.$$

We can now use this lemma to give the big-Θ closed form expressions for geometric series when $r \neq 1$. Let's start with the case where $r < 1$.

THEOREM 3.8. *For any* $0 < r < 1$, *the function* $f : n \mapsto 1 + r + r^2 + \cdots + r^n$ *satisfies* $f = \Theta(1)$.

PROOF. From the geometric series identity,

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}$$

we observe that for every $n \geq 1$, $f(n) \leq \frac{1}{1-r}$ so, setting $n_0 = 1$ and $c = \frac{1}{1-r}$ we have that for all $n \geq n_0$, $f(n) \leq c \cdot 1$ and, therefore, $f = O(1)$.

Similarly, if we let n_0 be the smallest integer for which $r^{n_0} \leq \frac{1}{2}$ (which is the value $n_0 = \lceil \log_{1/r}(2) \rceil$), then for every $n \geq n_0$ we have $\frac{1-r^{n+1}}{1-r} \geq \frac{1}{2(1-r)}$ and, taking the constant $c = \frac{1}{2(1-r)}$, $f \geq c \cdot 1$ so $f = \Omega(1)$.

Since $f = O(1)$ and $f = \Omega(1)$, then also $f = \Theta(1)$. □

The analysis of geometric series when $r > 1$ is similar.

THEOREM 3.9. *For any* $r > 1$, *the function* $f : n \mapsto 1 + r + r^2 + \cdots + r^n$ *satisfies* $f = \Theta(r^n)$.

PROOF. The geometric series identity can also be expressed as

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}.$$

[2] Can you see how to prove this identity? *Hint:* Multiply the left-hand side of the identity by $1 - r$ and remove the terms that cancel out.

With $n_0 = 1$ and $c = \frac{r}{r-1}$, this identity implies that for all $n \geq n_0$, $f(n) \leq \frac{r^{n+1}}{r-1} = c \cdot r^n$ so $f = O(r^n)$.

And when we choose n_0 to be the smallest integer that satisfies $r^{n_0} \geq 2$ (or, equivalently, $n_0 = \lceil \log_r(2) \rceil$) and $c = \frac{r}{2(r-1)}$ then we have that for all $n \geq n_0$, $f(n) = \frac{r^{n+1}-1}{r-1} \geq \frac{r^{n+1}-\frac{1}{2}r^{n+1}}{r-1} = c \cdot r^n$ so $f = \Omega(r^n)$.

Since $f = O(r^n)$ and $f = \Omega(r^n)$, then also $f = \Theta(r^n)$. □

LECTURE 4

Fast multiplication

The MergeSort algorithm is a classic example of the power of the *divide and conquer* technique. This technique is a general approach for solving problems with a three-step approach:

Divide: the original problem into smaller subproblems,
Conquer: each smaller subproblem separately, and
Combine: the results back together.

With MergeSort, we divided the initial array in two, conquered the sorting problem on the two smaller arrays with recursive calls to MergeSort, and combined the results using Merge. In the next few lectures, we will see other problems where this approach is applicable.

Note. Before discussing divide-and-conquer algorithms in class, we reviewed the Master theorem. See the notes for Lecture 3 for all the details on this topic.

1. Counting inversions

Here are two sequences of numbers:

$$1\ 2\ 5\ 4\ 6\ 7\ 9$$
$$9\ 1\ 5\ 2\ 7\ 6\ 4$$

Which one is closer to sorted? It seems obvious that the first sequence is, but how can we measure the "unsortedness" of a sequence to make this notion precise? One way to do this is by counting *inversions*.

Definition 4.1 (Inversion). The pair of indices $(i, j) \in \{1, 2, \ldots, n\}$ with $i < j$ forms an *inversion* in the sequence $a_1, a_2, \ldots, a_n$ if $a_i > a_j$. The *number of inversions* of the sequence $a_1, \ldots, a_n$ is

$$K(a) = |\{i, j \in [n] : i < j \text{ and } a_i > a_j\}|,$$

the total number of pairs of indices that form an inversion in the sequence.

In the *counting inversions* problem, we are given a sequence $a_1, \ldots, a_n$ of n integers and our task is to determine the number $K(a)$ of inversions in the sequence.

The brute force algorithm for counting inversions checks all $\binom{n}{2}$ pairs of indices $i < j$ to see if they form an inversion. This algorithm has time complexity $\Theta(n^2)$. Can we do better? With divide-and-conquer algorithms, yes!

The first step in applying the divide-and-conquer algorithm is to figure out how we can divide the problem into smaller subproblems. In this case, there's a very natural option: let's simply divide the sequence $a_1, \ldots, a_n$ into the smaller subsequences $L = (a_1, \ldots, a_{\frac{n}{2}})$ and $R = (a_{\frac{n}{2}+1}, \ldots, a_n)$.

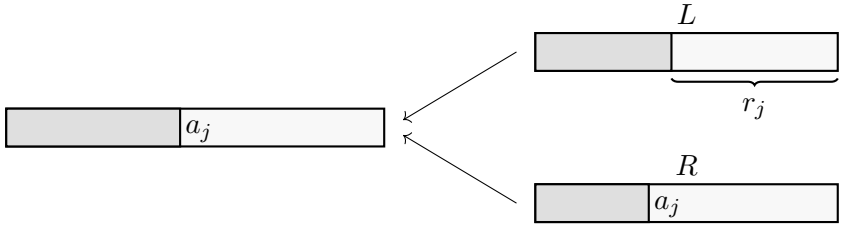
Can we conquer the smaller subproblems obtained with the division approach we identified above? Certainly! We can count the numbers $K(L)$ and $K(R)$ of inversions within L and within R with recursive calls to our algorithm.

So now we are left with the final task: to combine the solutions to the smaller subproblems into a solution for the original problem. And in this case it is not entirely trivial: the values $K(L)$ and $K(R)$ computed in the conquer step do not account for all of the inversions because there can also be inversions in a where $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$ that are not in either L or R. So we have that

$$K(a) = K(L) + K(R) + r$$

with r denoting the number of inversion pairs that include one element from L and one from R.

So how can we compute r? Well, for each element $a_j \in R$, we need to determine how many elements in L are smaller than a_j. Call this number r_j. Then $r = \sum_{j=n/2+1}^{n} r_j$. So now our task is to compute r_j efficiently, and at this point we might notice that there's a very close connection to the MergeSort algorithm: if we had sorted L and R and merge the sorted list, r_j is exactly the number of elements left in L at the moment when we moved a_j to the merged list.



This observation suggests that we can compute r efficiently with a simple modification to our standard MergeSort algorithm.

Algorithm 11: SortAndCountInversions(A)

```
if n = 1 return;
(L, r_L) ← SortAndCountInversions(A[1, ..., ⌊n/2⌋]);
(R, r_R) ← SortAndCountInversions(A[⌊n/2⌋ + 1, ..., n]);
r ← 0, S ← ∅;
while L, R are not empty do
    if the minimum element left in R is smaller than minimum left in L then
        Move the minimum element of R to the merged list S;
        r+ = |L|;
    else
        Move the minimum element of L to the merged list S;
return (S, r_L + r_R + r);
```

As an exercise, you should verify that the merge step of this algorithm can be turned into more precise pseudocode that runs in time $O(n)$. When this is done, we are left with an algorithm that has time complexity T satisfying the recurrence relation

$$T(n) = 2T(\tfrac{n}{2}) + O(n),$$

which as we already have seen with the analysis of MergeSort (or as we can verify again with the Master theorem) satisfies $T(n) = O(n \log n)$.

2. Integer multiplication

Let's consider one of the most fundamental arithmetic problems around: how to multiply two positive integers.

Definition 4.2 (Integer multiplication problem). Given two n-bit positive integers x and y, output their product xy.

Humans invented an algorithm for this problem thousands of years ago. It is now known as the grade school algorithm. With this algorithm, we multiply x by y_i and shift the result $n - i$ positions to the left. As an example, when we run this algorithm to solve 13×11, we obtain

```
    1101
  × 1011
  ------
    1101
   1101
  0000
+ 1101
  ------
  10001111      (= 143)
```

Notice that in the computations, we produces $O(n^2)$ intermediate bits, and this is indeed the time complexity of this algorithm. This begs the question: can we use the Divide & Conquer approach to obtain a more efficient algorithm?

To apply the Divide & Conquer approach, we need to first determine how we can break up the original multiplication problem into problems on smaller inputs. The natural way to do this is to split the n-bit integer x into two $n/2$-bit integers x_L (containing the $n/2$ most-significant bits) and x_R (containing the least-significant bits) and to do the same with y. Then

$$x = 2^{n/2} x_L + x_R,$$
$$y = 2^{n/2} y_L + y_R,$$

and

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

This is exactly what we were hoping to see! We have divided up the problem of multiplying two n-bit integers into four instances of the smaller problem of multiplying two $n/2$-bit integers. The resulting algorithm is as follows.

At first, it might be worrisome to see that we have also added extra multiplications by 2^n and $2^{n/2}$, but in fact these are just shifts (by n and $n/2$ bits, respectively), so these operations have time complexity $O(n)$ and the total time complexity of the Multiply algorithm is defined by the recursion

$$T(n) = 4\,T(n/2) + O(n).$$

Algorithm 12: MULTIPLY $(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ SPLIT (x);
$(y_L, y_R) \leftarrow$ SPLIT (y);
$P_{LL} \leftarrow$ MULTIPLY (x_L, y_L);
$P_{LR} \leftarrow$ MULTIPLY (x_L, y_R);
$P_{RL} \leftarrow$ MULTIPLY (x_R, y_L);
$P_{RR} \leftarrow$ MULTIPLY (x_R, y_R);
return $2^n P_{LL} + 2^{n/2}(P_{LR} + P_{RL}) + P_{RR}$;

We can apply the Master Theorem with parameters $a = 4$, $b = 2$, and $c = 1$ and the observation that $1 = c < \log_b a = 2$ to obtain

$$T(n) = O(n^2).$$

This is exactly the same as the grade school algorithm! So while the Divide & Conquer approach gives an elegant algorithm, it does not appear to have led to any efficiency improvement.

3. Fast integer multiplication

. . . And yet if that was the end of the story, we probably wouldn't be covering the integer multiplication problem at this point in the class! Recall that

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

There's a deceptively simple observation that can be traced back to Gauss which will be immensely useful to us: the middle term $x_L y_R + x_R y_L$ satisfies the identity

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

We can use this identity to express the term $x_L y_R + x_R y_L$ with a difference of three multiplications. This would be a step back for us (we only need 2 multiplications to compute the same term directly!) except that two of the multiplications, $x_L y_L$ and $x_R y_R$ are multiplications that we already need to do in the multiplication algorithm anyways! As a result, we can now implement a Divide & Conquer multiplication algorithm that performs only 3 multiplications instead of 4.

Algorithm 13: FASTMULTIPLY $(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ SPLIT (x);
$(y_L, y_R) \leftarrow$ SPLIT (y);
$P_{LL} \leftarrow$ FASTMULTIPLY (x_L, y_L);
$P_{RR} \leftarrow$ FASTMULTIPLY (x_R, y_R);
$P_{\text{sum}} \leftarrow$ FASTMULTIPLY $(x_L + x_R, y_L + y_R)$;
return $2^n P_{LL} + 2^{n/2}(P_{\text{sum}} - P_{LL} - P_{RR}) + P_{RR}$;

The time complexity of this algorithm now satisfies

$$T(n) = 3\,T(n/2) + O(n).$$

We can again apply the Master Theorem, this time with parameters $a = 3$, $b = 2$, and $c = 1$. Since $1 = c < \log_b a = \log_2 3 < 1.59$, we obtain

$$T(n) = O(n^{1.59}).$$

This result is a great illustration of the power of algorithmic thinking: despite countless mathematicians (and students of all ages) using multiplication algorithms over thousands of years, it was only in 1960 that Karatsuba showed with the above algorithm that it was possible to solve the integer multiplication in subquadratic time.

LECTURE 5

Closest pair of points

You have already seen how Divide & Conquer can be used to sort lists (with MergeSort), count inversions and do fast integer multiplication. Today, we explore how the same technique can also be used to solve the matrix multiplication problem and a fundamental problem in computational geometry.

Before jumping into our problems, however, we revisit recurrence relations one last time.

1. Recurrences by induction

We already saw two methods for solving recurrences: recurrence trees, and the Master Theorem. There is another very powerful technique for solving recurrences: proofs by induction. This technique, which is also known as the *substitution method*, consists of two steps:

(1) Guess an (ideally tight) upper bound on $T(n)$.
(2) Prove that your guess is correct using induction.

One advantage of this method is that it works even with more complicated functions of n inside the recursions. For example, take the recurrence

$$T(n) = 2T(\lfloor \tfrac{n}{2} \rfloor) + n$$

with the base case $T(1) = 1$. Previously, we simplified such recurrences by ignoring the floor operator, but that's not necessary with this method. Instead, we start by guessing that $T(n) \leq cn \log n$ for some constant c (that we will fix later) whenever $n \geq 2$. Then we can prove that this upper bound holds by induction.

Induction step: The induction hypothesis is that for every $2 \leq k < n$, $T(k) \leq ck \log k$. Fix $n \geq 4$. By the induction hypothesis,

$$\begin{aligned} T(n) &= 2T(\lfloor \tfrac{n}{2} \rfloor) + n \\ &\leq 2c\lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor + n \\ &\leq cn \log \tfrac{n}{2} + n \\ &= cn \log n - cn + n \leq cn \log n \end{aligned}$$

with the last inequality holding whenever $c \geq 1$.

Base cases: $T(2) = 2T(1) + 2 = 4 \leq c \cdot 2 \log 2$ as long as $c \geq 2$. And $T(3) = 2T(1) + 3 = 5 \leq c3 \log 3$ when $c \geq 2$ (and in fact when $c \geq \frac{5}{3 \log 3}$.

Put together, we have shown that $T(n) \leq 2n \log n$.

Warnings. There are two main warnings to remember about this approach.

Warning 1. Guesses that are wrong can sometimes look "almost right" in the induction step... but they are still wrong!

For example, if we had made the wrong guess that $T(n) \leq cn$ for some constant c in the recurrence above and ran the induction step argument, we would end up with the upper bound $T(n) \leq (c+1)n$. Even though $c+1$ is also a constant when c is constant, this does not mean that our guess was right (it's not!) because it actually causes the bound to grow as we apply the argument recursively.

Warning 1. Guesses that are technically correct but are not tight can look correct in the induction step.

Say that we had guessed $T(n) \leq cn^2 \log n$ in the recurrence above. In this case, the proof by induction does work out correctly and indeed the bound is true. But it is not tight, so it is worth checking that the guesses we made are as good as possible.

Changing variables. There is one more trick associated with the proofs by induction that is useful: change of variables (or *substitution*). Consider the recurrence relation

$$T(n) = 2T(\sqrt{n}) + \log n.$$

We could guess a bound for $T(n)$ and try to prove it by induction direction, but in this case it is easier to take a slightly different approach: define the variable $m = \log n$, so that $n = 2^m$. Then we can rewrite our recurrence relation as

$$T(2^m) = 2T(2^{m/2}) + \log n.$$

Define $S(m) = T(2^m)$. Then the above recurrence is equivalent to

$$S(m) = 2S(m/2) + m.$$

But we have already seen this recurrence! It satisfies $S(m) = O(m \log m)$. This means that $T(2^m) = O(m \log m)$ and, therefore, $T(n) = O(\log n \log \log n)$.

2. Fast matrix multiplication

A similar approach can also be used to obtain a fast matrix multiplication algorithm. Given two $n \times n$ matrices A and B, their product is the matrix $C = AB$ whose entries satisfy

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}.$$

The algorithm that uses this definition directly has time complexity $O(n^3)$. Again, we can try to use the Divide & Conquer method to obtain a more efficient algorithm. In this case, it is natural to divide each matrix in four:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

We then observe that the 4 submatrices of C can each be obtained by taking products of the submatrices of A and B:

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

The Divide & Conquer algorithm obtained using these identities makes 8 multiplications on $\frac{n}{2} \times \frac{n}{2}$ matrices and requires $O(n^2)$ extra work (since the matrices it adds together have n^2 entries) so its time complexity satisfies

$$T(n) = 8\,T(n/2) + O(n^2).$$

With the Master Theorem, we see that this time complexity is $T(n) = O(n^3)$, the same as the naïve matrix multiplication algorithm.

As was the case with the integer multiplication problem, however, it is possible to be more clever in how we choose the multiplications of the submatrices so that we compute the product C with only 7 (instead of 8) multiplications. Defining

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),
\end{aligned}$$

we have that

$$\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Using this approach, we obtain an algorithm with time complexity

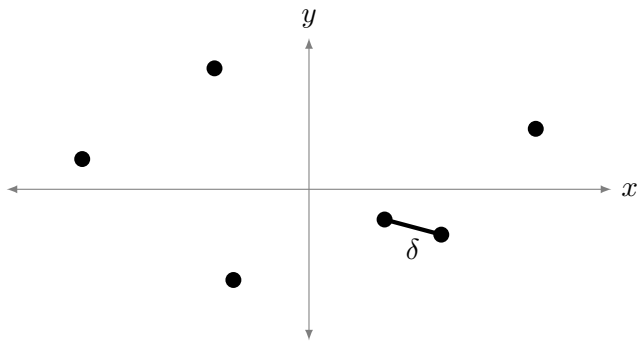
$$T(n) = 7\,T(\tfrac{n}{2}) + O(n^2),$$

which is $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$. The resulting algorithm is known as *Strassen's algorithm* and has been extremely influential in practice as well as in theory: there are numerous computational problems today that require multiplication of enormous matrices, and Strassen's algorithm is provides significant efficiency improvements over the naïve algorithms in those settings.

As a good algorithm designer, however, there is one question that you should be asking yourself at this point: can we do even better than Strassen's algorithm? Indeed we can. At the moment, the best known algorithm for matrix multiplication has time complexity $O(n^{2.37286\ldots})$. It is conjectured that matrix multiplication can be done in time $O(n^2)$ which, if true, would clearly be optimal since there are n^2 entries in an $n \times n$ matrix.

3. Closest pair of points on the line

Definition 5.1 (Closest pair). An instance of the *Closest Pair problem* is a set of n points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R} \times \mathbb{R}$ on the plane. A valid solution to such an instance is the distance $\delta = \min_{1 \le i < j \le n} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ between the closest pair of points in the set.

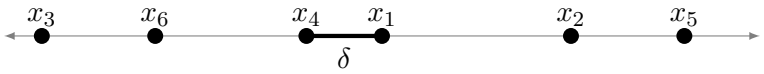


Algorithms for the closest pair problem are used as a building block for many other problems, in computational geometry and beyond. But the naïve brute-force algorithm for the problem, which computes the distance between every pair of points in the set, has time complexity $\Theta(n^2)$. Our goal today is to find a more efficient algorithm for the same problem.

REMARK 5.2. In many cases, what we want to compute is not quite the minimum distance between the closest pair of points, but rather to *identify* the pair of points that are closest to each other. By the end of the lecture, you should be able to easily modify the algorithms that we obtain to solve this variant of the problem as well.

It's not immediately obvious how we can do better than the brute-force algorithm. In situations like this, it is often extremely helpful to consider a simpler version of the problem first to build up some intuition for the problem. Let's do this by considering the one-dimensional version of the problem.

DEFINITION 5.3 (Closest pair on the line). Given a set of n points $x_1, \ldots, x_n \in \mathbb{R}$ on the line, return the distance $\delta = \min_{1 \le i < j \le n} |x_i - x_j|$ between the closest pair of points in the set.



Note that the points $x_1, \ldots, x_n$ are not necessarily sorted. The simplest brute-force algorithm for this problem involves checking the distance between each pair of points and returning the minimum distance. This algorithm has time complexity $\Theta(n^2)$. We can easily do better by sorting the points first and observing that the minimum distance must occur between *adjacent* points in the sorted list.

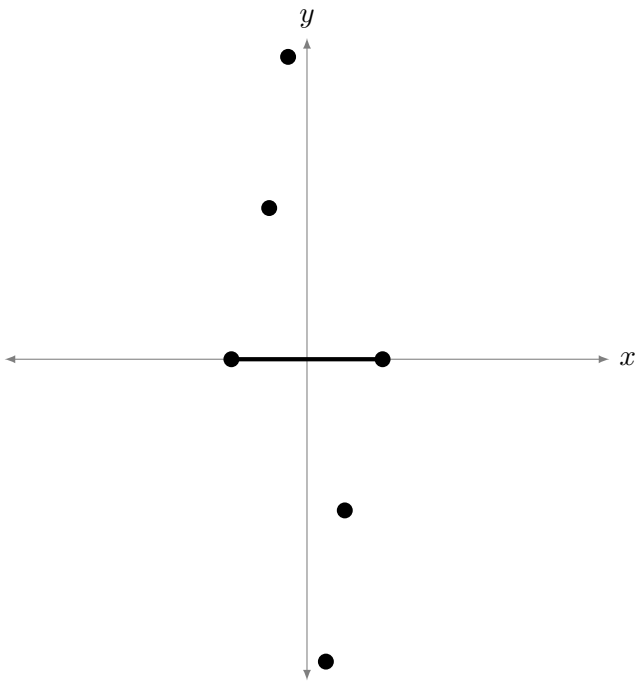
Algorithm 14: CLOSESTPOINTSLINE$(x_1, \ldots, x_n)$

```
Sort the points;
δ ← |x_1 − x_2|;
for i = 3, ..., n do
    δ ← min{δ, |x_i − x_{i−1}|};
return δ;
```

REMARK 5.4. It's a good exercise to try to design a divide and conquer algorithm for this problem. Can you find one that, given a list of sorted points, finds the minimum distance in time $O(n)$?

So our conclusion is that the closest points problem is simple when the points are on the line. We're now ready to return to the original version of the problem where the points are in the plane; we'll see if the ideas we used for the line setting can be extended to the plane as well.

One natural conjecture is that maybe we can solve the original problem with exactly the same idea as in the line setting: sort the points by x position, then check the distance of adjacent pairs of points in the sorted list. Unfortunately, that algorithm is not correct, as the following example shows.



Note that in the example, the closest pair of points is (x_1, y_1) and (x_n, y_n), so the position of the points in the list where they are sorted by x coordinate really does not give any indication of how close they are.

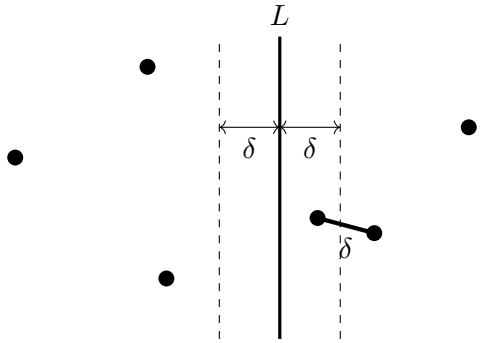
We can try to sort the points in other ways as well, but this line of attack quickly leads to multiple dead ends and/or algorithms that are very hard to analyze. A better idea is to still sort the points by x coordinate but then combine it with the divide & conquer technique. This approach leads to the following algorithm sketch:

(1) (Initially sort the list of points by x position.)
(2) *Divide* the sorted list of points into the two smaller sets $Q = \{(x_1, y_1), \ldots, (x_{\frac{n}{2}}, y_{\frac{n}{2}}\}$ and $R = \{(x_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}), \ldots, (x_n, y_n)\}$.
(3) *Conquer* the smaller instances by using recursive calls with inputs Q and R to find the smaller distances of pairs of points within each subset.
(4) *Combine* by returning the smallest value of (i) the minimum distance between pairs of points in Q, (ii) the minimum distance between pairs of points in R, and (iii) the minimum distance between a point in Q and a point in R.

This looks promising, but we have to be careful: we can't afford to compute the distance between every pair of points $q \in Q$ and $r \in R$ (otherwise we are back to a time complexity $\Theta(n^2)$) so we have to find a more clever way to compute the *combine* step.

Restricting the area of focus. There's a neat observation that will prove to be extremely useful: to implement the *combine* step, we don't necessarily have to compute the minimum distance between a point in Q and a point in R: if we let δ be the minimum distance between 2 points in Q or 2 points in R, we only have to worry about the case where there's a point in Q and a point in R that are at distance less than δ from each other.

Let L denotes a vertical line separating the points in Q from the points in R and S denotes the set of points in X that are at distance at most δ from L, then the above observation means that we only need to consider the points in S.



PROPOSITION 5.5. *Any pair of points $(x_i, y_i) \in Q$ and $(x_j, y_j) \in R$ at distance*

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq \delta$$

from each other must satisfy $(x_i, y_i) \in S$ and $(x_j, y_j) \in S$.

PROOF. If $(x_i, y_i) \notin S$, then the distance between the two points is

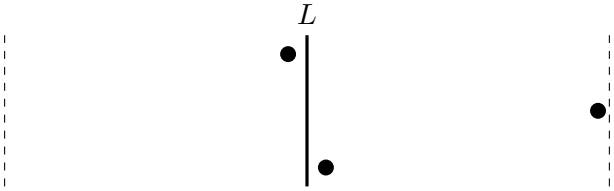
$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq \sqrt{(x_i - x_j)^2} = |x_i - x_j| > \delta$$

and the same lower bound applies if $(x_j, y_j) \notin S$ as well. □

Intuitively, this should be very helpful: we may expect that for many "well-behaved" inputs, only a few points are in S and we can compute the distance between every pair of these points to complete the *combine* step. Unfortunately, that approach does not work in general, because *all* n points might be contained in S. (This is the case, for instance, in the example we constructed at the beginning of the section.)

Nonetheless, if we look at this image carefully, we can see that we have made some progress in that the problem of finding the closest pair of points in S is starting to look a lot more like the one-dimensional version of the problem!

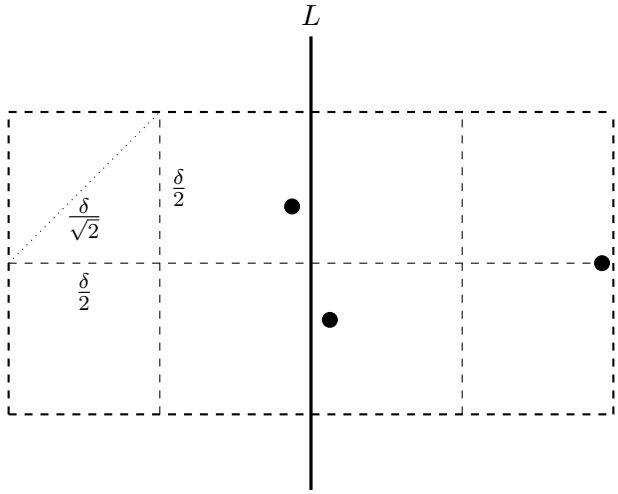
Key observation. If the problem of finding the closest pair of points in S is *exactly* like the one-dimensional version of the problem, we could then sort the points in S by their y value and then compute the distance between every adjacent pair of points. This will not quite work, because of examples like the following.



However, we will be able to show the next-best thing: that the closest pair of points in S will be *nearly* adjacent to each other when we sort by y value. This is because of the following key observation.

PROPOSITION 5.6. *For any point $(x^*, y^*) \in S$, there can be at most 8 points $(x', y') \in S$ where $y^* \leq y' \leq y^* + \delta$.*

PROOF. The space of all points that are at distance at most δ from L and with y coordinate bounded by $[y^*, y^* + \delta]$ can be partitioned into 8 squares with side length $\frac{\delta}{2}$. But there can be at most 1 point in S contained in each of those squares, because two points in the same square would have distance at most $\sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \frac{\sqrt{2}}{2}\delta < \delta$ from each other and would both be in Q or in R, contradicting the fact that δ is the minimum distance between any two points in Q or two points in R. □



The proposition guarantees that when we do the *combine* step, we only need to compute the distances of points that are at most 8 positions apart in the list that is sorted by y coordinate.

We therefore obtain the following divide & conquer algorithm for the Closest Pair problem, which we run after sorting the points $(x_1, y_1), \ldots, (x_n, y_n)$ so that $x_1 \leq x_2 \leq \cdots \leq x_n$.

What is the time complexity of this algorithm? Sorting the set S takes time $\Theta(n \log n)$ in the worst-case[1] and the rest of the work in the algorithm takes linear time, so that the time complexity of the algorithm on inputs that contain n points satisfies the recurrence

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n \log n)$$

... which you will solve as part of Assignment 2. (Your solution should show that the time complexity of this algorithm is much better than $\Theta(n^2)$.)

An optimization. We can improve the time complexity of the algorithm by avoiding the step of sorting S at every iteration of the algorithm. The main idea is that we can sort the initial set of points by y positions (as well as by x positions; we keep both lists). Then when we build S we can use the list sorted by y position to create it in sorted order directly.

With this optimization, the time complexity of the algorithm now satisfies the recurrence

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n)$$

[1] Recall that $|S|$ can be as large as n.

Algorithm 15: ClosestPair$((x_1, y_1), \ldots, (x_n, y_n))$

$\delta_Q \leftarrow \text{ClosestPair}\big((x_1, y_1), \ldots, (x_{\frac{n}{2}}, y_{\frac{n}{2}})\big)$;
$\delta_R \leftarrow \text{ClosestPair}\big((x_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}), \ldots, (x_n, y_n)\big)$;
$\delta \leftarrow \min\{\delta_Q, \delta_R\}$;
$S \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
    if $x_{\frac{n}{2}} - \delta \leq x_i \leq x_{\frac{n}{2}} + \delta$ **then**
        $S \leftarrow S \cup \{(x_i, y_i)\}$;
SortByY(S);
for $1 \leq i < j \leq |S|$, $j - i \leq 7$ **do**
    $\delta = \min\{\delta, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\}$;
return δ;

which, by the Master Theorem, is satisfied by $T(n) = \Theta(n \log n)$. The initial sorts by x and by y both have the same time complexity, which means that we obtain an algorithm that solves the Closest Pair problem in time $\Theta(n \log n)$. This is much better than the $\Theta(n^2)$ bound of the brute-force algorithm!

Part 2

Greedy algorithms

LECTURE 6

Scheduling problems

In the last two lectures, we explored how the divide and conquer technique is useful for designing algorithms for many different problems. Today, we start examining another technique: *greedy algorithms*.

1. Greedy algorithms

A *greedy* algorithm is one in which we:

(1) Break down a problem into a sequence of decisions that need to be made, then
(2) Make the decisions one at a time, each time choosing the option that is optimal at the moment (and not worrying about later decisions).

This is one of the simplest algorithm design techniques around, yet as we will see it yields efficient algorithm for many different problems. But the challenge with this technique in many instances is proving the algorithm's correctness—or, often, *determining* whether the algorithm is correct or not in the first place.

You have already seen one classic greedy algorithm in CS 240: Huffman codes are optimal prefix codes obtained by running the greedy algorithm to join trees with different frequencies together. This algorithm is both correct and efficient.

We also use a greedy algorithm in real life when we make change.

PROBLEM 2 (Making change). *Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $v \geq 1$, determine the minimum number of coins whose denominations sum to v. (I.e., a valid solution $a_1, \ldots, a_n \in \mathbb{N}$ is one that satisfies $\sum_{i=1}^{n} a_i d_i = v$ and minimizes $\sum_{i=1}^{n} a_i$.)*

For example, we have coins worth 200, 100, 25, 10, 5, and 1 cents in current circulation in Canada. How would you make change for 1.43$ using those coins? You would start by taking a coin with maximal denomination that is at most 143 (here the loonie worth 100 cents), subtract that amount from the total (leaving 43 cents in this case), and repeat.

Algorithm 16: GREEDYCHANGE($d_1 > d_2 > \cdots > d_n; v$)

for $i = 1, \ldots, n$ **do**
    $a_i \leftarrow \lfloor v \,/\, d_i \rfloor$;
    $v \leftarrow v \bmod d_i$;
return $(a_1, \ldots, a_n)$;

Does this algorithm always return a valid solution to the Making Change problem? It does for every value v when the coins have denominations 200, 100, 25, 10, 5, and 1 but the proof of correctness in this case is not trivial. In general, however, there are denominations and choices of v where the algorithm does not return the minimal number of coins. Take for example the case where the denominations are 8, 7, and 1 and the value to return is 14. The greedy algorithm will return the solution $(1, 0, 6)$, for a total of 7 coins, when the solution $(0, 2, 0)$ requires only two coins.

Today, we explore another fundamental problem that can be solved using greedy algorithms.

2. Interval scheduling

Many real-world scheduling problems can be formulated in terms of the abstract *interval scheduling problem*.

DEFINITION 6.1 (Interval scheduling problem). Given a set of n pairs of start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ where each pair (s_i, f_i) satisfies $s_i < f_i$, find a maximum subset $I \subseteq \{1, 2, \ldots, n\}$ such that no two intervals in I overlap. (I.e., for every $i \neq j \in I$, $s_i > f_j$ or $s_j > f_i$.)

It is quite natural to try to use the greedy method to solve this problem: to do so, we just need to decide how the algorithm chooses the "best" interval to add to I given the intervals that have already been added to this set. We have many possibilities on how to define the notion of "best" interval:

Earliest starting time: Pick the interval with the earliest starting time that does not overlap any of the intervals we already added to I.

Earliest finish time: Pick the interval with the earliest finish time that does not overlap any of the intervals we already added to I.

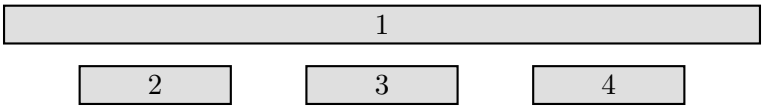
Shortest interval: Pick the interval with the shortest length $f_i - s_i$ among those that don't overlap any of the intervals we already added to I.

Minimum conflicts: Let J be the set of intervals that don't overlap with any interval in I. Pick the interval in J that overlaps with the fewest other number of other intervals in J.

All four options sound perfectly reasonable, but only one yields an algorithm that always outputs a valid solution to the interval scheduling problem.

PROPOSITION 6.2. *The greedy algorithm with earliest starting time does not always find a valid solution to the interval scheduling problem.*

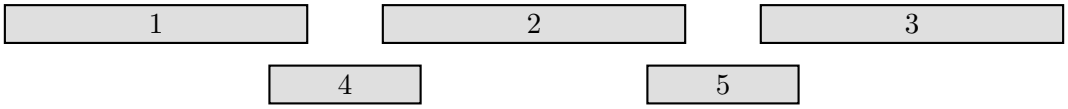
PROOF. Consider the set of intervals



The valid solution is $I = \{2, 3, 4\}$ but the greedy algorithm outputs $I = \{1\}$ instead. □

PROPOSITION 6.3. *The greedy algorithm with shortest interval does not always find a valid solution to the interval scheduling problem.*

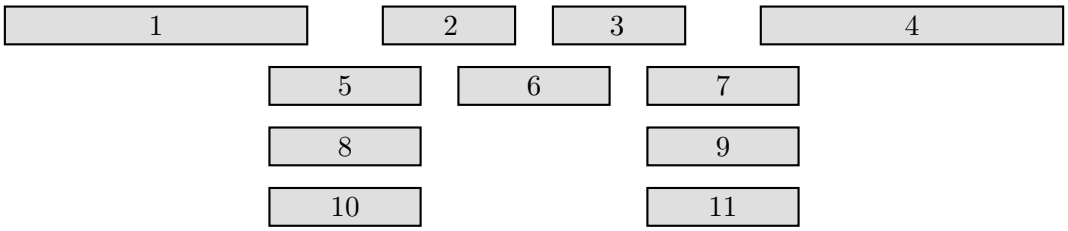
PROOF. Consider the set of intervals



The valid solution is $\{1, 2, 3\}$ but the greedy algorithm outputs $I = \{4, 5\}$ instead. □

PROPOSITION 6.4. *The greedy algorithm with minimal conflicts does not always find a valid solution to the interval scheduling problem.*

PROOF. Consider the set of intervals



The valid solution is $\{1, 2, 3, 4\}$ but the greedy algorithm outputs $I = \{1, 4, 6\}$ instead. □

We are now left with a single candidate algorithm: earliest finish time. We can implement this greedy algorithm in a simple way: sort the intervals by finish time, and when we go through the list we add an interval (s_i, f_i) iff its start time is larger than the last (and therefore most recently added) finish time of any interval in I.

Algorithm 17: GREEDYSCHEDULER((s_1, f_1),...,(s_n, f_n))

```
A ← indices {1, ..., n} sorted by f_i;
I ← A[1];
f* ← f_{A[1]};
for i = 2, ..., n do
    if s_{A[i]} > f* then
        I ← I ∪ A[i];
        f* ← f_{A[i]};
return I;
```

The time complexity of the GreedyIntervalScheduler is $\Theta(n \log n)$. It remains to show that it always returns a valid solution to the interval scheduling problem. We do so via an *always ahead* argument: we show that at every point in the execution of our greedy algorithm, we can complete the partial solution we have obtained so far into a valid solution to the original problem.

As a first step, let us show how we can always run the greedy algorithm to obtain the *first* interval in a maximum set I^* of non-overlapping intervals.

PROPOSITION 6.5. *Let $I^* = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, n\}$ be the indices of a maximum set of non-overlapping intervals in some instance of the Interval Scheduling problem sorted by finish times so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$. Let $j \in [n]$ be the index of the first interval selected by the GreedyIntervalScheduler. Then $I^\dagger = \{j, i_2, i_3, \ldots, i_k\}$ is also a maximum set of non-overlapping intervals.*

PROOF. The set $I^\dagger$ must have the same cardinality as I^* since $f_j \leq f_{i_1} < f_{i_\ell}$ for each $\ell \in \{2, 3, \ldots, k\}$ and so $j \notin \{i_2, \ldots, i_k\}$. We need to show that $I^\dagger$ is a set of non-overlapping intervals. Since I^* is a set of non-overlapping intervals, the only thing we need to show is that the interval j does not overlap with any of the intervals $i_2, \ldots, i_k$.

The fact that I^* contains non-overlapping intervals and that f_{i_1} is the smallest finish time implies that we must have $f_{i_1} < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$. And the definition of the GreedyIntervalScheduler guarantees that $f_j \leq f_{i_1}$, so we also have $f_j < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$ and the interval j does not overlap with any of the intervals $i_2, \ldots, i_k$, as we wanted to show. □

We can now use a proof by induction to establish the correctness of the GREEDYSCHEDULER algorithm.

THEOREM 6.6. *The* GREEDYSCHEDULER *solves the interval scheduler problem.*

PROOF. Let $I^* = \{i_1, i_2, \ldots, i_k\}$ be a maximum set of non-overlapping intervals, and let $I^\dagger = \{j_1, j_2, \ldots, j_m\}$ be the set of non-overlapping intervals returned by the algorithm. We again sort the indices so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$ and $f_{j_1} < \cdots < f_{j_m}$. Clearly, $m \leq k$; we want to show that in fact $m = k$.

Let us now use a proof by induction on c to show that for every c in the range $1 \leq c \leq m$, the set $I^{(c)} = \{j_1, j_2, \ldots, j_c, i_{c+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. The base case was established in Proposition 6.5.

For the induction step with $c \geq 2$, the induction hypothesis is that $I^{(c-1)}$ is a maximum set of non-overlapping intervals; we want to show the same is true of $I^{(c)}$. Note that $I^{(c)} = (I^{(c-1)} \setminus \{i_c\}) \cup \{j_c\}$. We must have $f_{j_c} \leq f_{i_c} < f_{i_{c+1}} < \cdots < f_{i_k}$ so $j_c \notin \{j_1, \ldots, j_{c-1}, i_{c+1}, \ldots, i_k\}$ and $|I^{(c)}| = |I^{(c-1)}|$. Furthermore, since $I^\dagger$ and $I^{(c-1)}$ are sets of non-overlapping intervals and since $f_{j_c} \leq f_{i_c} < s_{i_{c+1}} < \cdots < s_{i_k}$, the set $I^{(c)}$ is a maximum set of non-overlapping intervals.

The proof by induction we just completed shows that $I^{(m)} = \{j_1, j_2, \ldots, j_m, i_{m+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. But the greedy algorithm returns $I^\dagger = \{j_1, \ldots, j_m\}$ only if all other intervals with finish times greater than j_m intersect with some interval already in $I^\dagger$; this means that we must have $m = k$ and the algorithm returns a maximum set of non-overlapping intervals. □

3. Minimizing lateness

There are a number of other scheduling problems for which the greedy algorithm is effective. Here's one that may be relevant with respect to juggling coursework for multiple classes at the same time:

DEFINITION 6.7 (Minimizing lateness problem). Given a set of n tasks with processing times $p_1, \ldots, p_n$ and deadlines $d_1, \ldots, d_n$, find an ordering of the tasks that minimizes the maximum lateness of any task when they are performed one at a time in that order.

Formally, given an ordering π of $\{1, 2, \ldots, n\}$, the *lateness* of the kth task in this order is

$$L_k^{(\pi)} := \sum_{i:\pi(i)\leq\pi(k)} p_i - d_k$$

and π is a valid solution if $\max_{k\leq n} L_k^{(\pi)}$ is minimal among all permutations.

For example, an input to the problem may be tasks with the following processing times and deadlines:

	1	2	3	4	5	6	7
p_i	4	2	4	3	1	4	5
d_i	5	9	1	3	3	4	10

If we process the tasks in the order above, we obtain lateness values

	1	2	3	4	5	6	7
L_i	-1	-3	9	10	11	14	13

so that the maximum lateness of this ordering is 14.

Let's explore how we can solve this problem using greedy algorithms. Here the natural way to break down the problem into individual decisions is to pick which task to do first, then which one to do second, etc. There are a number of different criteria we could use to decide which task to schedule next.

Shortest processing time first: Sort the tasks in order of increasing processing times.

Earliest deadline: Sort the tasks in order of increasing deadlines.

Smallest slack: At each step, pick the task with the smallest "slack" value $d_j - p_j$.

Of those three options, only the earliest deadline criterion yields a greedy algorithm that solves the Minimal Lateness problem.[1]

Algorithm 18: GreedyLateness$(p_1, \ldots, p_n, d_1, \ldots, d_n)$

$\pi \leftarrow$ a permutation of $\{1, \ldots, n\}$ for which $d_{\pi(1)} \leq d_{\pi(2)} \leq \cdots \leq d_{\pi(n)}$;

return π;

The permutation π can be computed in time $\Theta(n \log n)$, and this is also the time complexity of the algorithm. The more challenging aspect of the analysis of this algorithm is its proof of correctness. We use an *exchange* argument: we will show that given any valid solution to the lateness problem, we can convert it to the output of the greedy algorithm without increasing its maximum lateness.[2]

We also use the fact that we can sort an out-of-order sequence by swapping pairs of consecutive elements.

Fact 2 (Bubble sort). *For π any ordering of $1, 2, \ldots, n$, if we repeatedly*

(1) *Find a pair of elements $i < j$ where j is right before i in π;*
(2) *Update π by swapping elements i and j;*

then after at most $\Theta(n^2)$ swaps, we end up with the sorted ordering $1\,2\,3\,\cdots\,n$.

[1] Exercise: Prove that statement!

[2] In class, we first saw the proof of correctness of the greedy algorithm in the special case where $n = 2$, then described how to obtain the general result from the special case. The following proof establishes the result directly; it is a good exercise to rewrite the proof of the special case for $n = 2$.

For example, starting with the ordering 1 5 3 4 2, the sorting algorithm described above yields the following sequence of orderings:

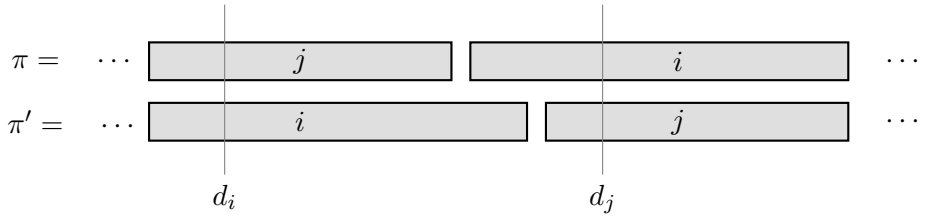
$$1\,\mathbf{5}\,\mathbf{3}\,4\,2$$
$$1\,3\,\mathbf{5}\,\mathbf{4}\,2$$
$$1\,3\,4\,\mathbf{5}\,\mathbf{2}$$
$$1\,3\,\mathbf{4}\,\mathbf{2}\,5$$
$$1\,\mathbf{3}\,\mathbf{2}\,4\,5$$
$$1\,2\,3\,4\,5$$

We are now ready to prove the correctness of the GreedyLateness algorithm.

Theorem 6.8. *The* GreedyLateness *algorithm solves the Maximal Lateness problem.*

Proof. For simplicity, let us reorder the tasks by increasing deadline so that the greedy algorithm performs them in order (i.e., 1, then 2, then 3, etc.) We want to prove that this order is a valid solution.

Consider any other ordering π. Then in that order there must be two consecutive tasks i, j with $d_i \leq d_j$ but j performed right before i. Swap those two tasks to obtain a new ordering π'.



Then every task except i and j have the same lateness in π and in π'. The lateness of i satisfies

$$L_i^{(\pi')} \leq L_i^{(\pi)}$$

since we do task i earlier in π'. And the lateness of j satisfies

$$L_j^{(\pi')} \leq L_i^{(\pi)}$$

because $d_i \leq d_j$. Therefore, the maximum lateness of π' is at most that of π. This means that we can continue swapping in this way until we obtain the ordering $1\,2\,\cdots\,n$ of the greedy algorithm, and at every step along the way we never increase the maximum lateness so that the algorithm's solution is valid. □

LECTURE 7

More greedy algorithms

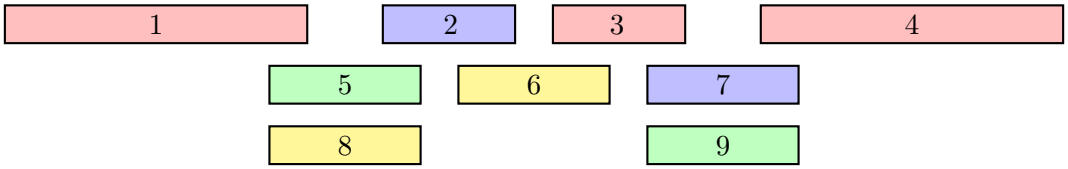
We saw in the last lecture how the greedy algorithm method can be used to solve the interval scheduling and minimizing lateness problems. In this lecture, we will examine a few other problems that can be solved by greedy algorithms and see how a common exchange method can be used to establish the correctness of these algorithms.

1. Interval colouring

There are some situations where we can prove the correctness of the greedy algorithm directly (or with a *structural proof*) instead of having to use either the always-ahead or the exchange argument. One such case is with the interval colouring problem.

DEFINITION 7.1 (Interval colouring problem). Given a set of n pairs of start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ where each pair (s_i, f_i) satisfies $s_i < f_i$, find a colouring of the intervals with as few colours as possible such that no two intervals that overlap share the same colour.

For example, here is a colouring of a set of 9 intervals that satisfies the condition that no two overlapping intervals share the same colour:



Note that in this case the colouring is not optimal because there is another colouring that uses only 3 colours.

There is a simple greedy algorithm that solves the interval colouring problem:

- Sort the intervals by start time.
- At each interval i, if there is a colour c that was already used previously and is not assigned to any interval overlapping i, assign that colour to i; otherwise, assign a new colour to i.

By construction, this algorithm guarantees that the overlapping intervals will always be assigned distinct colours. But how can we argue that it requires the minimum number of colours? This is where a direct argument can be used:

THEOREM 7.2. *Let d be the maximum number of intervals that cover any given point on the line. Then any legal colouring of the intervals requires at least d colours and that is exactly the number of colours used by the greedy algorithm.*

PROOF. Let p denote a point that is covered by d intervals. Since all d intervals that cover p overlap each other, any colouring of the intervals that assigns distince colours to overlapping intervals must use at least d colours.

And since at most d intervals cover any point, when we reach the start point of an interval, at most $d-1$ other intervals cover that point, so if d colours have already been used previously, at least 1 of them is available to colour the current interval. □

2. Fractional knapsack

The *fractional knapsack* problem aims to find the maximum value of items that we can fit in a backpack, when we can subdivide items into any fractional parts.

DEFINITION 7.3 (Fractional knapsack). Given a set of n items that have positive weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, as well as a maximum weight capacity W of the knapsack, find a set of amounts $x_1, \ldots, x_n$ of each item that you put in your backpack that maximizes the total value

$$\sum_{i=1}^{n} \frac{x_i}{w_i} v_i$$

among all possible sets of amounts where for each item i we have $0 \leq x_i \leq w_i$ and the total weight taken is $\sum_{i=1}^{n} x_i \leq W$.

For example, an instance may have three items with weights and values

	1	2	3
w_i	4	3	3
v_i	12	7	6

and a total knapsack capacity weight $W = 6$.

There is a natural greedy algorithm for this problem: sort the items by decreasing relative value v_i/w_i, then consider each element in turn and put as much of it in the knapsack as can fit.

Algorithm 19: GREEDYKNAPSACK$(w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

```
Order the items by decreasing value of v_i/w_i;
for i = 1, ..., n do
    x_i ← min{W, w_i};
    W ← W − x_i;
return x_1, ..., x_n;
```

THEOREM 7.4. *The* GREEDYKNAPSACK *algorithm solves the fractional knapsack problem.*

PROOF. Let $y_1, \ldots, y_n$ be a valid solution to the fractional knapsack problem, using the ordering defined by the algorithm where v_i/w_i is decreasing. And let $x_1, \ldots, x_n$ be the solution returned by the GREEDYKNAPSACK algorithm. We prove that $x_1, \ldots, x_n$ is a valid solution by induction on the number of indices $i \leq n$ for which $x_i \neq y_i$.

In the base case, when $x_i = y_i$ for each $i = 1, 2, \ldots, n$, then $x_1, \ldots, x_n$ is the same solution as $y_1, \ldots, y_n$ so it is a valid solution.

For the induction step, let $m = |\{i \leq n : x_i \neq y_i\}| \geq 1$ be the number of differences in the solution. By the induction hypothesis, if there is a valid solution $y'_1, \ldots, y'_n$ with

$|\{i \leq n : x_i \neq y'_i\}| < m$, then $x_1, \ldots, x_n$ is also a valid solution. Our goal is to now use an exchange argument to show that such a solution $y'_1, \ldots, y'_n$ exists.

Let $k \leq n$ be the smallest index where $x_k \neq y_k$. Then it must be that $x_k > y_k$ since GreedyKnapsack maximizes the value of x_k. And since $\sum x_i = \sum y_i$, there must be an index $\ell > k$ for which $y_\ell > x_\ell$. Let's exchange a δ amount of weight of element ℓ for element k to obtain the solution y' where

$$y'_k = y_k + \delta \qquad \text{and} \qquad y'_\ell = y_\ell - \delta.$$

Choose $\delta = \min\{x_k - y_k, y_\ell - x_\ell\}$ so that $y'_k = x_k$ or $y'_\ell = x_\ell$ and, therefore, $|\{i \leq n : x_i \neq y'_i\}| < m$. To complete the proof, we must show that $y'_1, \ldots, y'_n$ is a valid solution. The difference in the total value of the solutions y' and y satisfies

$$\delta(v_k/w_k) - \delta(v_\ell/w_\ell) = \delta(v_k/w_k - v_\ell/w_\ell).$$

The ordering of the elements guarantees that $v_k/w_k \geq v_\ell/w_\ell$ so that the total value of y' is at least as large as that of y and, therefore, y' is a valid solution. □

What if we now consider the variant of the problem where we are only allowed to choose $x_i \in \{0, w_i\}$? (I.e., where the items are indivisible.) Does the greedy algorithm above still work? You should convince yourself that it is no longer correct—and you should be able to see how the exchange argument we used above fails in this situation! We will revisit this version of the knapsack problem in the next section, when we see how the *dynamic programming* algorithm design technique can be used to solve it efficiently when W is not too large. (And we will see the problem again in the NP-completeness section, when we will see why we shouldn't expect to find an efficient algorithm for this problem in general.)

3. Bonus: Offline caching

Note: You are not responsible for the material in this section. It was not covered in class and is included here just for interest.

A *cache* in a computer system is a small but very fast block of memory that enables significant speedup on the data that we store on it. But how do we decide what data to keep in the cache? To define this problem, we introduce a bit of notation. The units of data in memory are called *pages*. A cache can contain up to k pages; we consider the setting where the total number of pages is much larger than k. A sequence of *page requests* is a list of page identifiers. A page request incurs a cost of 0 if the page is currently in the cache; otherwise it incurs a cost of 1 as the page needs to be brought into memory—this event is known as a *page fault*. If the cache already contains k pages at the moment of a page fault, one of the k current pages must be *evicted*—removed from the cache—to make room for the requested page.

In the (idealized) setting where we know ahead of time exactly the entire sequence of page requests, the problem of minimizing the cost of serving all the page requests is known as *offline caching*.

Definition 7.5 (Offline caching problem). Given a cache of size k and a sequence of k page requests, determine which page should be evicted (if any) at each page fault to minimize the cost of serving all the page requests in the sequence.

As an example, if we label the pages `a`, `b`, and `c` and consider the setting where the cache has size $k = 2$, the page request sequence

```
a b c b c b a c
```

can be served with cost 4 (i.e., with 4 page faults in total). We can see this by drawing the contents of the cache after each request (with element that is newly inserted in red).

Requests:	a	b	c	b	c	b	a	c
Cache:	a	a	c	c	c	c	c	c
	_	b	b	b	b	b	a	a

In this example, 4 page faults is optimal: there is no alternative way that we could have chosen to keep other elements in the cache that would result in fewer faults. But can we design an algorithm that will *always* guarantee that it serves the page requests with minimum cost? There are in fact many reasonable greedy algorithms for the problem that we can examine:

LRU: (Least-recently-used) Evict the page that has been accessed least recently.
FIFO: (First-in-first-out) Evict the page that was added to the cache least recently.
LFU: (Least-frequently-used) Evict the page that has been accessed least frequently.
LFD: (Longest-forward-distance) Evict the page that will be accessed the furthest in the future.

The first three of these options are popular because they are *online* strategies: they do not require us to know the entire sequence of page requests that will be coming in the future. But, as you can check by constructing counter-examples, none of those three can guarantee that they will always minimize the number of page faults. The LFD algorithm, however, *does* offer this guarantee and solve the offline caching problem.

THEOREM 7.6. *The greedy LFD algorithm always minimizes the number of page faults.*

PROOF SKETCH. We again want to use an exchange argument.

Let $C_1, \ldots, C_n$ be the contents of the cache of any sequence of page request services that minimizes the total number of page faults. Let $G_1, \ldots, G_n$ be the contents of the cache when we run the LFD algorithm. The notion of "closeness" between the two sequences we use is the minimum index j where $C_j \neq G_j$. In this proof, we want to show that given C and G, we can always construct a new optimal algorithm whose cache contents $C'_1, \ldots, C'_n$ will satisfy $C'_1 = G_1, \ldots, C'_j = G_j$.

Let $\mathtt{a} = C_j \setminus G_j$ be the element kept in the cache of the optimal algorithm but expelled from the greedy algorithm's cache at request j, and let $\mathtt{b} = G_j \setminus C_j$ be the element that the greedy algorithm kept in the cache instead. We build $C'$by copying the greedy algorithm until step j, then after that we continue expelling the same elements as the optimal algorithm C does for the requests $j+1, \ldots, k-1$ where k is the first request in which either

- C evicts a; or
- C brings b back into the cache by evicting some other page c.

The key observation is that the first step where we can't mimic the optimal algorithm must be of one of these two forms; in general it could also be possible that we get a page request for a before either of the above two types of requests occur, but by definition of our greedy algorithm, this can only happen after there was a page request for b (the second case above).

Now if at time k the optimal algorithm evicted a, we evict b from C'. And if instead at request k the optimal algorithm evicts c to bring b back into the cache, we evict c and bring back a into the cache C'. In both cases, we then obtain $C'_k = C_k$ and we can continue following the optimal algorithm to handle the requests $k+1, \ldots, n$. □

Note that the above argument is not a complete proof, for a slightly technical reason: the algorithm we constructed to have cache C' doesn't exactly follow the rules of cache algorithms we defined earlier. In the case where at time k, the optimal algorithm evicted `c` and brought `b` into the cache, we made C' also replace an element from its cache as well—but in fact for this request C' does not get a page fault (because it already has `b` in cache) so it does not get to update its cache content.

Exercise 7.1 (For interested students). Complete the proof of correctness of the LFD algorithm by either (i) modifying the definition of cache algorithms to allow updates even on non-page faults, or (ii) showing how we can turn C' into a "proper" cache algorithm that updates its cache only on page faults without changing the contents of $C'_1, \ldots, C'_j$.

Part 3

Dynamic programming

LECTURE 8

Weighted interval scheduling

We now turn our attention to another general technique for designing algorithms: *dynamic programming*. The main idea for this technique can be summarized as follows: We can solve a big problem by

(1) Breaking it up into smaller sub-problems;
(2) Solving the sub-problems from smallest to largest; and
(3) Storing solutions along the way to avoid repeating our work.

Note the similarity with the greedy method technique: there also we break up the problem into smaller sub-problems and solve them one by one in order. The main difference of the dynamic programming technique lies in *how* we solve these sub-problems: whereas with the greedy algorithm we usually resort to a simple local decision criterion, with dynamic programming we will be critically relying on using the *work we have done so far* to solve the current sub-problem. All of this discussion is best explained with illustrations. We begin with a simple example—computing Fibonacci numbers—and then will consider a range of other problems where dynamic programming is particularly useful over the next few lectures.

1. Fibonacci numbers

DEFINITION 8.1. The *Fibonacci sequence* of numbers is the famous sequence of integers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ defined by the rule

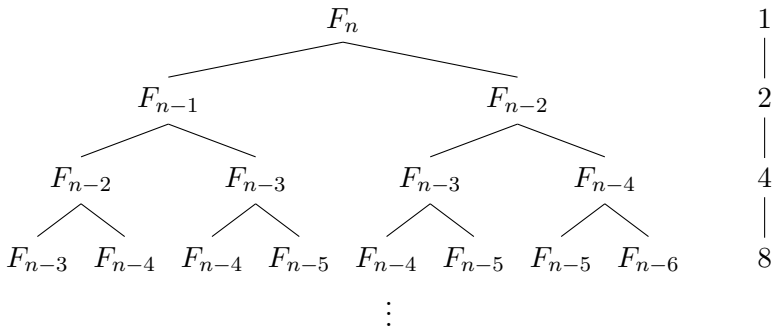
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

The definition can be turned directly into a simple algorithm:

Algorithm 20: FIB1(n)

if $n = 0$, **return** 0;
if $n = 1$, **return** 1;
return FIB1($n - 1$) + FIB1($n - 2$);

Is this a good algorithm for computing Fibonacci numbers? Not at all! It's correct, but horribly inefficient. Drawing the recurrence tree, we see that the amount of work performed by this algorithm is exponential in n.



In fact, the time complexity of this algorithm satisfies

$$T(n) \geq T(n-1) + T(n-2).$$

so $T(n) \geq F_n$ so the time complexity of the algorithm is bounded below by the Fibonacci numbers themselves. (If you're interested, you can prove directly that $F_n \geq 2^{n/2}$ for every $n > 6$ by induction; you can also read up on Binet's formula to obtain a closed-form expression for F_n.)

We can do better than FIB1 by examining the recurrence tree we already drew to spot a glaring inefficiency: the intermediate values $F_{n-1}, F_{n-2}, F_{n-3}, \ldots$ are each recomputed multiple times within this tree. Eliminating this inefficiency can be done easily with a simple application of the *dynamic programming* technique:

- **Subproblems:** Compute $F_1, F_2, F_3, \ldots, F_n$ in that order.

Algorithm 21: FIB2 (n)

```
if n = 0 return 0;
A[0] ← 0;
A[1] ← 1;
for i = 2, ..., n do
    A[i] = A[i − 1] + A[i − 2];
return A[n];
```

The algorithm we obtain now run in time that is *linear* in n. It's hard to overstate how much of an improvement this represents over the original algorithm: whereas FIB1 can't even compute reasonably small values like F_{50} even on the world's biggest supercomputers, FIB2 can easily compute much larger values—say, $F_{5000000}$—on any computer.

2. Text segmentation

Can the following sequence of letters be split up into (actual English) words?

`thecustomofdrinkingorangejuicewithbreakfastisnotverywidespread`

In this case, yes:

`the custom of drinking orange juice with breakfast is not very widespread`

And in general, we can ask the same question about any string of letters to obtain the *text segmentation* problem.

DEFINITION 8.2 (Text segmentation problem). Given an array A of n letters from the sets $\{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and access to a function IsWord that for any sequence w of letters returns True if w is a word in the English language and False otherwise, determine whether A can be split into a sequence of English words or not.

The example we saw above suggests that a greedy algorithm would provide a good solution to the text segmentation problem: just keep calling IsWord on the words $A[1]$, $A[1..2]$, $A[1..3]$,..., $A[1..k]$ until it returns True, make that the first word, then continue from $A[k+1]$ onwards. But if you try to prove that this algorithm works, you will find that no argument can establish its correctness... for good reason! Consider the input

`themexamples`

that can be separated into the two words `them examples`—if you run the simple greedy algorithm on this input, it will first identify the word `the`, then `me`, but it will then be stuck with the non-word `xamples` and incorrectly report that the string cannot be split into words.

Similarly, an alternative greedy algorithm that always try to extract the *longest* possible word at the beginning of the string will extract the word `theme` and be stuck with `xamples` on the input above, so this algorithm does not work either. In fact, I'm not aware of any greedy algorithm that correctly solves the text segmentation problem. Instead, to get a correct and efficient solution, we need to turn to dynamic programming.

The key, as always with the dynamic programming technique, is to identify the right subproblems. And here, there is one option that seems quite natural: determining if the initial part $A[1..k]$ of the array can be split into words.

- **Subproblems:** For $k = 1, 2, \ldots, n$, determine if $A[1..k]$ can be split into words.

Solving these subproblems in order, how can we determine if $A[1..k]$ can be split into words or not? Simple: It can be split if and only if there is some index $j < k$ for which $A[1..j]$ can be split into words and $A[j+1..k]$ is a word! This is enough to complete our dynamic programming algorithm.

Algorithm 22: Split($A[1..n]$)

```
S[0] ← True;
for k = 1, ..., n do
    S[k] ← False;
    for j = 0, ..., k − 1 do
        if S[j] and IsWord(A[j + 1..k]) then
            S[k] ← True;
return S[n];
```

If we assume that each call to IsWord has time complexity $O(1)$, then the total time complexity of the algorithm is $O(n^2)$.

Note that the algorithm can also be easily modified so that when a string can be converted into a sequence of words, the algorithm not only returns true but outputs a valid segmentation of the string. This is left as an exercise.

3. Longest increasing subsequence

An *increasing subsequence* in a given sequence of numbers is a subset of those numbers (not necessarily all next to each other in the original sequence) in increasing order. For instance, in the sequence

5 2 1 4 3 1 6 9 2

we can find the increasing subsequence 2 3 6 9 since those numbers are increasing and are in that order within the original sequence:

5 **2** 1 4 **3** 1 **6** **9** 2

DEFINITION 8.3 (Longest increasing subsequence problem). Given a sequence A of n positive integers, find the length of the longest increasing subsequence of A.

Once again, we can use the dynamic programming technique to design an efficient algorithm for the longest increasing subsequence problem. But we have to be careful in how we choose which subproblems to solve. The most natural idea is to find the longest subsequence within each prefix of the original sequence.

- **Candidate Subproblems:** For $k = 1, 2, \ldots, n$, find the length of the longest increasing subsequences in $A[1..k]$.

But how can we compute the value of $A[1..k]$ when we have solved the earlier subproblems? Unfortunately, just knowing the length of the longest common subsequence in $A[1..j]$ for every $j < k$ does not give us enough information to compute $A[1..k]$ because we don't know to which subsequence we can append the number $A[k]$ and make a longer increasing subsequence. So for this problem it turns out that we want to consider slightly different subproblems.

- **Subproblems:** For $k = 1, 2, \ldots, n$, find the length of the longest increasing subsequences in $A[1..k]$ *that includes* $A[k]$ itself.

The small change makes it much easier to solve the subproblems: find the value $j < k$ where (i) $A[1..j]$ has the longest increasing subsequence; and (ii) $A[j] < A[k]$. This solution yields the following algorithm.

Algorithm 23: LIS($A[1..n]$)

```
for k = 1, ..., n do
    S[k] ← 1;
    for j = 1, ..., k − 1 do
        if A[j] < A[k] then
            S[k] ← max{S[k], S[j] + 1};
return max{S[1], S[2], ..., S[n]};
```

Note that because of our choice of subproblems, the value to return at the end is not $S[n]$ (as this would correspond to the longest increasing subsequence of $A[1..n]$ that includes $A[n]$ itself) but rather the maximum value $S[k]$ over all $k = 1, 2, \ldots, n$.

It's helpful to visualize what the algorithm does on an example:

Input	5	2	1	4	3	1	6	9	2
$S[k]$	1	1	1	2	2	1	3	4	2
Coming from $j =$	0	0	0	2	2	0	4	7	3

EXERCISE 8.1. Show how to modify the LIS algorithm to return a longest increasing sequence instead of just its length. (*Hint.* Consider storing the information from the bottom line of the table in a separate array.)

The time complexity of the LIS algorithm is $O(n^2)$. Note that this is not best possible: it is also possible to solve the longest increasing subsequence in time $O(n \log n)$.

4. Longest common subsequence

A *common subsequence* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is a string $z_1, \ldots, z_k$ for which there are indices $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ where for each $\ell \leq k$, $z_\ell = x_{i_\ell} = y_{j_\ell}$. For example, in the pair of strings

$$x = \texttt{POLYNOMIAL}$$
$$y = \texttt{EXPONENTIAL},$$

the string $z = \texttt{PONIAL}$ is a subsequence of x and y. (As are the strings `POL`, `PNA`, etc.)

DEFINITION 8.4 (Longest common subsequence problem). An instance of the *longest common subsequence (LCS)* problem is a pair of strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. The valid solution to an instance is the length of the longest common subsequence of x and y.

The natural way to break down the LCS problem into smaller subproblems is to consider the longest common subsequence of prefixes of x and y. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i, j) = \text{length of LCS of } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When i or j is 0 (which corresponds to x or y being an empty string), then

$$M(0, j) = 0 \qquad \text{and} \qquad M(i, 0) = 0.$$

Given that we have computed $M(i-1, j)$, $M(i, j-1)$, and $M(i-1, j-1)$, can we now compute $M(i, j)$? Indeed we can!

$$M(i, j) = \max \begin{cases} M(i-1, j-1) + 1 & \text{if } x_i = y_j \\ M(i-1, j) \\ M(i, j-1) \end{cases}$$

This gives the following algorithm.

Algorithm 24: LCS$(x_1, \ldots, x_m, y_1, \ldots, y_n)$

```
for i = 1, ..., m do M[i, 0] = 0;
for j = 1, ..., n do M[0, j] = 0;

for i = 1, ..., m do
    for j = 1, ..., n do
        M[i, j] = max{M[i − 1, j], M[i, j − 1]};
        if x_i = y_j then M[i, j] = max{M[i, j], M[i − 1, j − 1] + 1};
return M[m, n];
```

We can picture the algorithm as filling out the table of values $M[i, j]$, row by row. With the instance $x = \texttt{ALGORITHM}$, $y = \texttt{ANALYSIS}$, the table looks as follows.

The time complexity of the algorithm is $\Theta(mn)$.

	∅	A	N	A	L	Y	S	I	S
∅	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1
L	0	1	1	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
O	0	1	1	1	2	2	2	2	2
R	0	1	1	1	2	2	2	2	2
I	0	1	1	1	2	2	2	3	3
T	0	1	1	.	.	.	.	.	.
H	0	.	.	.	.	.	.	.	.
M	0	.	.	.	.	.	.	.	.

The LCS algorithm determines the length of the longest common subsequence between the two strings given as input, but it does not identify the subsequence itself. What if we want to identify it? There are a number of different ways we can modify the algorithm to do so. Or, if we have already computed the matrix M of LCS lengths for all prefixes, we can identify the LCS itself by working backwards from $M(m, n)$.

Algorithm 25: PRINTLCS(x, y, M, i, j)

```
if i > 1 and M[i, j] = M[i − 1, j] then
    PrintLCS(x, y, M, i − 1, j);
else if j > 1 and M[i, j] = M[i, j − 1] then
    PrintLCS(x, y, M, i, j − 1);
else
    /* We must have matched x_i = y_j                                  */
    PrintLCS(x, y, M, i − 1, j − 1);
    Print x_i;
```

Calling PRINTLCS(x, y, M, m, n) will print the longest common subsequence of x and y. Interestingly, by calling it with any other $i \leq m$ and $j \leq n$ we can also print the longest common subsequence of any prefixes of x and y just as efficiently.

LECTURE 9

String problems

In this lecture, we continue our exploration of the dynamic programming technique by examining two more problems where the technique is useful.

Note. The problem of finding a longest common sequence in two strings that was covered at the beginning of the lecture is described in Lecture 8.

1. Edit distance

The length of the longest common subsequence can be interpreted as a measure of how similar two strings are. A more sophisticated measure of the similarity between different strings is known as *edit distance*.

DEFINITION 9.1 (Edit distance). The *edit distance* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is the minimum number of edit operations required to transform x into y, where the 3 possible edit operations are:

Adding: a letter to x,
Deleting: a letter from x, and
Replacing: a letter in x with another one.

For example, the edit distance between the strings `POLYNOMIAL` and `EXPONENTIAL` is 6, as this is the minimum number of edit operations required to go from one string to the other:

```
--POLYNOMIAL
EXPONEN-TIAL
```

In this example, the full list of operations that transformed `POLYNOMIAL` into `EXPONENTIAL` is as follows.

`POLYNOMIAL`	
`EPOLYNOMIAL`	(Add `E`)
`EXPOLYNOMIAL`	(Add `X`)
`EXPONYNOMIAL`	(Replace `L` with `N`)
`EXPONENOMIAL`	(Replace `Y` with `E`)
`EXPONEN-MIAL`	(Delete `O`)
`EXPONENTIAL`	(Replace `M` with `T`)

A basic computational problem is to find the edit distance between two strings.

DEFINITION 9.2 (Edit distance problem). An instance of the *edit distance problem* is two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$; the valid solution to an instance is the edit distance between x and y.

We can again use the dynamic programming method to solve this problem by considering the subproblems obtained by computing the edit distance between prefixes of x and y. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i,j) = \text{edit distance between } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When $i = 0$ or $j = 0$, the edit distance is easy to compute: it is exactly the length of the other string. So

$$M(i,0) = i \qquad \text{and} \qquad M(0,j) = j.$$

What about for the other entries? If $x_i = y_j$, then we can match those characters together and we get $M(i,j) = M(i-1, j-1)$. If not, we have three choices:

(1) Replace x_i with y_j. In this case, we get $M(i,j) = M(i-1,j-1) + 1$.
(2) Delete x_i. With this choice, $M(i,j) = M(i-1,j) + 1$.
(3) Add the character y_j to the string x right before x_i. This choice gives us $M(i,j) = M(i,j-1) + 1$.

To compute $M(i,j)$, we want to choose the option among the ones above that has minimum value. So this means that we get

$$M(i,j) = \min \begin{cases} M(i-1,j-1) & \text{if } x_i = y_j \\ M(i-1,j-1) + 1 & \text{if } x_i \neq y_j \\ M(i-1,j) + 1 & \\ M(i,j-1) + 1. & \end{cases}$$

As long as we compute the values of M in an order where $M(i-1,j-1)$, $M(i-1,j)$, and $M(i,j-1)$ have all been computed before we compute $M(i,j)$, computing this value takes $\Theta(1)$ time. We can again proceed row by row, obtaining an algorithm that looks very similar to the one we used to compute the length of the longest common subsequence.

Algorithm 26: EDITDISTANCE$(x_1, \ldots, x_m, y_1, \ldots, y_n)$

```
for i = 1, ..., m do M[i, 0] = i;
for j = 1, ..., n do M[0, j] = j;

for i = 1, ..., m do
    for j = 1, ..., n do
        if x_i = y_j then
            r = M[i − 1, j − 1];
        else
            r = M[i − 1, j − 1] + 1;
        M[i, j] = min{M[i − 1, j] + 1, M[i, j − 1] + 1, r};
return M[m, n];
```

The time complexity of this algorithm is again $\Theta(mn)$.

2. Weighted Interval Scheduling

Let's revisit the interval scheduling problem, with a slight twist.

DEFINITION 9.3 (Weighted interval scheduling). In the *weighted interval scheduling problem*, an instance is a set of n pairs of intervals that have start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ and positive *weights* $w_1, \ldots, w_n$. A valid solution is a subset $I \subseteq \{1, 2, \ldots, n\}$ of non-overlapping intervals that maximizes $\sum_{i \in I} w_i$.

Let's even simplify the problem slightly to only aim to find the *total weight* of a valid solution to the weighted interval scheduling problem.

Without weights, we saw that the greedy algorithm that sorts the intervals by finish time and considers them in that order solves the problem. It's not clear whether greedy algorithms can also solve the weighted version of the problem, but as it turns out it is still useful to sort the intervals by finish times and consider them in order. But instead of making greedy decisions, we can define the following subproblems.

- **Subproblems.** For $k = 0, 1, \ldots, n$, define $W[k]$ to be the maximum sum of weights of a set $I \subseteq \{1, ..., k\}$ of non-overlapping intervals taken from the k *intervals with earliest finish times.*

We can now design a dynamic programming algorithm that computes $W[1], W[2], \ldots, W[n]$ in that order and returns $W[n]$.

The values of the first subproblems are easy to determine: $W[0] = 0$ (since in this case there is no interval to select) and $W[1] = w_1$. Our task is now to figure out how to compute $W[k]$ given the values of $W[j]$ for each $1 \leq j < k$. Examining this problem carefully, we realize there are exactly two possibilities to consider:

Case 1: $k \notin \text{OPT}(k)$: The first possibility is that the kth interval is not in the maximum-weight subset $I \subseteq \{1, 2, \ldots, k\}$ of non-overlapping intervals. If that's the case, then

$$W[k] = W[k-1]$$

since the optimal set is the same whether or not we consider the kth interval.

Case 2: $k \in \text{OPT}(k)$: Otherwise, the kth interval is in the maximum-weight subset $I \subseteq \{1, 2, \ldots, k\}$ of non-overlapping intervals. Then, letting $j^* < k$ be the largest index such that $f_{j^*} < s_k$, we have that

$$W[k] = W[j^*] + w_k$$

since any set $I \subseteq \{1, \ldots, k\}$ of non-overlapping intervals that contains interval k cannot include any of the intervals $j^* + 1, \ldots, k - 1$.

We can compute the value of both candidates to determine which is correct. The resulting dynamic programming algorithm is as follows.

Algorithm 27: WEIGHTEDIS$((s_1, f_1), \ldots, (s_n, f_n), w_1, \ldots, w_n)$

```
(Sort intervals so that f_1 ≤ f_2 ≤ ··· ≤ f_n);
W[0] ← 0;
for k = 1, ..., n do
    j* ← max{0 ≤ i < k : f_i < s_k};
    W[k] = max{W[k − 1], W[j*] + w_k};
return W[n];
```

What is the running time of this algorithm? The naïve algorithm for identifying j^* in each iteration of the for loop is a linear search that takes time $O(k)$, for a total time

complexity $O(n^2)$; but since the intervals are sorted by finish time we can use binary search instead. The total time complexity of the resulting implementation of the algorithm is then $O(n \log n)$.

As a final step, we should see how to revise the algorithm to find the maximum-weight set of non-overlapping intervals itself, and not just its total weight. We leave this as an exercise for now.

EXERCISE 9.1. Modify the dynamic programming solution to solve the (original version of the) Weighted Interval Scheduling problem.

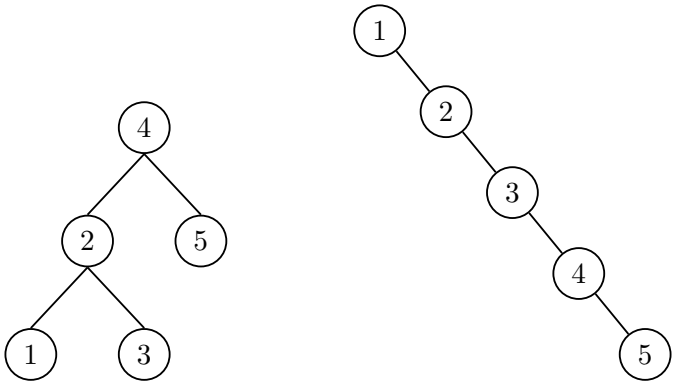
LECTURE 10

Coins and knapsacks

In this lecture, we complete our exploration of the dynamic programming technique by revisiting problems related to trees and knapsacks.

1. Optimal binary search trees

Here's a trick question: which of the following two binary search trees is best?



Without any additional information on how the binary search tree will be used, the first example is better: its depth is 3 (versus 5 for the second tree) which means that the worst-case cost of querying elements in that tree is better in that tree. But what if we know with which probability each element is queried by the application, and we find that element 1 will be queried more often than any other element, and by a large margin: in this case, the second tree will be more efficient.

We can formalize the problem of finding the optimum binary search tree for a set of items with known query probabilities as follows.

Definition 10.1 (Optimal binary search tree problem). Given items $1, 2, \ldots, n$ and probabilities $p_1, \ldots, p_n$, construct a binary search tree T that minimizes the search cost

$$\sum_{i=1}^{n} p_i \cdot \text{depth}_T(i)$$

where $\text{depth}_T(i)$ is the depth of (= number of queries required to reach) element i in the tree T.

(Note that when we build a binary search tree, we are *not* allowed to place the items in arbitrary order; *optimal Huffman code* problem is the variant where you are allowed to reorder the items in the tree.)

For example, consider the following instances of the optimal binary search tree problem with $n = 5$:

- $p_1 = p_2 = \cdots = p_5 = \frac{1}{5}$.
  In this case the left tree is optimal and has search cost
  $$3 \cdot \tfrac{1}{5} + 2 \cdot \tfrac{1}{5} + 3 \cdot \tfrac{1}{5} + 1 \cdot \tfrac{1}{5} + 2 \cdot \tfrac{1}{5} = \tfrac{11}{5}.$$
- $p_1 = 0.6$, $p_2 = \cdots = p_5 = 0.1$.
  In this case, the cost of the left tree is
  $$3 \cdot 0.6 + 2 \cdot 0.1 + 3 \cdot 0.1 + 1 \cdot 0.1 + 2 \cdot 0.1 = 2.6$$
  and the search cost of the right tree is
  $$1 \cdot 0.6 + 2 \cdot 0.1 + 3 \cdot 0.1 + 4 \cdot 0.1 + 5 \cdot 0.1 = 2$$
  but neither is optimal: there is another binary search tree with cost 1.8. (Exercise: can you find it?)

Dynamic programming can be used to obtain an efficient algorithm that solves the optimal binary search tree problem. As always with this method, the key step is in identifying the subproblems and figuring out in which order we solve them.

With this problem, it is natural to consider building trees "from the bottom up", so that the most promising subproblems involve solving the optimal binary search tree problem for items $i, i+1, \ldots, j-1, j$.

Subproblems: For each $1 \le i \le j \le n$, define $M[i, j]$ to be minimum search cost of binary search tree for items $i, i+1, \ldots, j$.

Order of subproblems: We solve the subproblems by increasing values of $j - i$ (i.e., for the smallest intervals $i, \ldots, j$ first, then for longer intervals).

Now, let us see how to solve each subproblem, given our solutions to previous subproblems. To find the optimum binary search tree for elements $i, \ldots, j$, we need to do three things:

(1) Identify the node $k \in \{i, \ldots, j\}$ that we choose for the root;
(2) Find the search cost of the optimal subtrees for items $i, \ldots, k-1$ and $k+1, \ldots, j$; and
(3) Compute the search cost of the subtree we construct in this way.

The first item has a simple solution: try all possible choices of k and choose the best one! The second item is also easily handled—our solutions to the previous problems will give us exactly the answers we need. So it remains to compute the search tree of our final subtree. This turns out to be remarkably simple as well: the total search cost C of tree for items $i, \ldots, j$ with left and right subtrees of costs C_L and C_R is

$$C = C_L + C_R + \sum_{\ell=i}^{j} p_\ell.$$

Putting all three items together, we obtain that the solution to each subproblem is given by the expression

$$M[i, j] = \min_{k=i}^{j} \{M[i, k-1] + M[k+1, j]\} + \sum_{\ell=i}^{j} p_\ell.$$

The full algorithm for the problem is as follows.

The time complexity of the algorithm is $O(n^3)$ since there are $O(n^2)$ subproblems and each subproblem is solved in time $O(n)$. It's also possible to design an improved algorithm

Algorithm 28: OPTIMUMBST$(p_1, \ldots, p_n)$

```
for i = 1, ..., n do M[i, i] ← p_i; M[i + 1, i] ← 0;
for d = 1, ..., n − 1 do
    for i = 1, ..., n − d do
        j ← i + d;
        b ← M[i + 1, j];
        for k = i + 1, ..., j do
            b ← min{b, M[i, k − 1] + M[k + 1, j]};
        M[i, j] ← b + Σ_{ℓ=i}^{j} p_ℓ;
return M[1, n];
```

with time complexity $O(n^2)$: interested readers can look to Donald Knuth's original article titled *Optimum Binary Search Trees* (Acta Informatica, 1971) for all the details.

2. Knapsack

We saw a variant of the knapsack problem where we were allowed to divide items and only include a fraction of them in our knapsack. In the standard version of the problem, we no longer have that power: we either include or exclude an item in the knapsack.

DEFINITION 10.2 (Knapsack). An instance of the *knapsack problem* is a set of n items that have positive integer weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, as well as a maximum weight capacity W of the knapsack. A valid solution to the problem is a subset $S \subseteq \{1, 2, \ldots, n\}$ of the items that you put in your backpack that satisfies $\sum_{i \in S} w_i \leq W$ and maximizes the total value $V = \sum_{i \in S} v_i$ among all sets that satisfy the weight condition.

To distinguish this problem explicitly from the fractional knapsack problem, it is also sometimes called the 0-1 *knapsack* problem.

We can solve the knapsack problem using the dynamic programming technique. Let's consider the natural way to do this, following the approach that we used in previous lectures. A natural way to break down the problem into smaller subproblems is to consider only the items $1, \ldots, k$ for each $k \in \{1, 2, \ldots, n\}$ (along with the trivial subproblem when $k = 0$).

Then, as in the other problems we consider, we have a simple observation that can let us solve the subproblem with the first k items when we already solved it with the first $k - 1$ items: either k is in the optimal subset $S_k \subseteq \{1, 2, \ldots, k\}$ of items we put in the knapsack, or it is not. If it is not, then the optimal value $V_k = V_{k-1}$. But if it is, we realize that there is a twist that we need to consider: we need to find the maximum subset $S'_{k-1} \subseteq \{1, 2, \ldots, k-1\}$ that fit in a knapsack with capacity $W - w_k$ (not W!) if we are to put these items into the knapsack along with item k.

Therefore, we need to consider subproblems where we consider the first k elements *and* where we fix the capacity of a knapsack to be w, for $k \in \{0, 1, 2, \ldots, n\}$ and for $w = \{0, 1, 2, \ldots, W\}$. We do so by defining

$$M(k, w) = \max_{S \subseteq \{1,2,\ldots,k\} : \sum_{i \in S} w_i \leq w} \sum_{i \in S} v_i.$$

Then for every $w \leq W$ and $k \leq n$,

$$M(0, w) = 0 \qquad \text{and} \qquad M(k, 0) = 0.$$

For $k \geq 1$, we then have two possibilities: either $w_k > w$, in which case the item k does not fit into the knapsack and $M(k, w) = M(k-1, w)$, or $w_k \leq w$ in which case $M(k, w)$ is the maximum of the optimal value $M(k-1, w)$ obtained by leaving out item k and the optimal value $v_k + M(k-1, w-w_k)$ obtained by including item k in the knapsack. So for each $k = \{1, 2, \ldots, n\}$ we have

$$M(k,w) = \begin{cases} M(k-1,w) & \text{if } w_k > w \\ \max\{M(k-1,w), v_k + M(k-1, w-w_k)\} & \text{if } w_k \leq w. \end{cases}$$

The resulting algorithm is as follows.

Algorithm 29: KNAPSACK$(w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

```
for w = 0, 1, 2, ..., W do M[0, w] ← 0;
for k = 0, 1, 2, ..., n do M[k, 0] ← 0;

for k = 1, ..., n do
    for w = 1, ..., W do
        if w_k ≤ w then
            M[k, w] ← max{M[k − 1, w], v_k + M[k − 1, w − w_k]};
        else
            M[k, w] ← M[k − 1, w];
return M[n, W];
```

THEOREM 10.3. *The* KNAPSACK *algorithm solves the knapsack problem.*

PROOF. Let $\mathrm{OPT}(k, w)$ denote the maximum value of a subset of items $1, \ldots, k$ that fit into a knapsack with capacity w. We show $M[k, w] = \mathrm{OPT}(k, w)$ for all $k = 0, 1, \ldots, n$ and $w = 0, 1, \ldots, W$ by induction on k and w. In the base cases, when $k = 0$ or $w = 0$, then $M[k, w] = 0 = \mathrm{OPT}(k, w)$.

For the induction step, the induction hypothesis lets us assume that $M[k-1, w'] = \mathrm{OPT}(k-1, w')$ for every $w' \leq w$. Consider now the value $\mathrm{OPT}(k, w)$and a corresponding set $S \subseteq \{1, 2, \ldots, k\}$ of items with total weight at most w and value $\mathrm{OPT}(k, w)$. There are two cases to consider.

(1) If $w_k > w$, then it must be that $k \notin S$ so that $\mathrm{OPT}(k, w) = \mathrm{OPT}(k-1, w)$ and by the induction hypothesis

$$M[k, w] = M[k-1, w] = \mathrm{OPT}(k-1, w) = \mathrm{OPT}(k, w).$$

(2) If $w_k \leq w$, then $\mathrm{OPT}(k, w) = \mathrm{OPT}(k-1, w)$ (if $k \notin S$) or $\mathrm{OPT}(k, w) = v_k + \mathrm{OPT}(k-1, w-w_k)$ (if $k \in S$), whichever is larger. So by the induction hypothesis

$$\begin{aligned} M[k, w] &= \max\{M[k-1, w], v_k + M[k-1, w-w_k]\} \\ &= \max\{\mathrm{OPT}(k-1, w), v_k + \mathrm{OPT}(k-1, w-w_k)\} = \mathrm{OPT}(k, w). \end{aligned}$$

□

THEOREM 10.4. *The time complexity of the* KNAPSACK *algorithm is* $\Theta(nW)$.

PROOF. The code inside the two nested for loops is run a total of nW times and has complexity $\Theta(1)$. The two initialization for loops have time complexity $\Theta(W)$ and $\Theta(n)$, respectively. So the total time complexity of the algorithm is $\Theta(nW + n + W) = \Theta(nW)$. □

If we want to find the optimal set of items to put in the knapsack, we can again use the backtracking approach to find which items to put in the knapsack based on the $M[k, w]$ values.

Part 4

Graph exploration

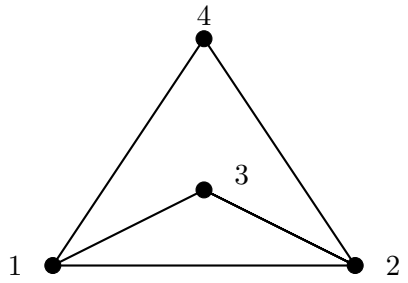
LECTURE 11

Breadth-first search

1. Graphs

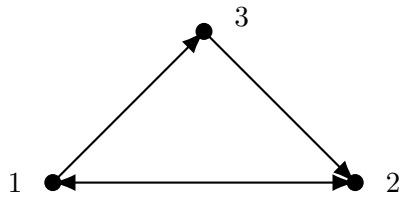
A *graph* $G = (V, E)$ is a set V of *vertices* (sometimes called *nodes*) and a set $E \subseteq V \times V$ of *edges* connecting pairs of vertices. Whenever we work on graphs, we will let $n = |V|$ denote the number of vertices in the graph and $m = |E|$ denote the number of edges of the graph.

An *unordered graph* is a graph whose edges are undirected.



The vertex set of this graph is $V = \{1, 2, 3, 4\}$ and the edge set of this graph is $E = \big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\big\}$.

An *ordered graph* has edges that are directed from one end vertex to the other.



The vertex set of this graph is $V = \{1, 2, 3\}$ and its edge set is $E = \big\{(1, 2), (1, 3), (2, 1), (3, 2)\big\}$.

1.1. Definitions. There are a number of basic definitions that we will use.

Adjacent: Two vertices u and v are *adjacent* (or *neighbours*) if they are connected by an edge.

Incidence: A vertex v is *incident* to an edge e if it is one of its endpoints.

Degree: The *degree* of a vertex v is the number of edges to which it is incident.

In/Out-degree: In directed graphs, the *in-degree* of a vertex v is the number of edges that end at v and the *out-degree* of v is the number of edges that start at v.

Path: A *path* in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for each $1 \leq i < k$, $(v_i, v_{i+1}) \in E$.

Simple path: A *simple path* is a path in which no vertices appear more than once except possibly at the endpoints.

Cycle: A *cycle* is a path of length at least 3 that starts and ends at the same vertex and contains no other vertex repetitions.

Connectedness: An undirected graph is *connected* if every two vertices in V is connected by a path.

Tree: A *tree* is a connected graph that does not contain any cycle.

Connected component: A *connected component* of a graph is a maximal connected subgraph.

1.2. Computing with graphs. There are two natural representations of graphs that are useful when we solve computational problems on them: the adjacency matrix representation, and the adjacency list representation.

The *adjacency matrix* representation of a graph $G = (V, E)$ is an $n \times n$ matrix $A \in \{0, 1\}^{n \times n}$ where

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

for every pair i, j of vertices. When the graph G is unordered, $A[i, j] = A[j, i]$ is always satisfied. The adjacency matrix representation of a graph always has space complexity $O(n^2)$ (even when the graph is *sparse*, i.e. when $m \ll n^2$). But with this representation we can answer the question "is i adjacent to j" in constant time for any pair of vertices.

The *adjacency list* representation of a graph $G = (V, E)$ is a set of n linked lists, one for each vertex in the graph, in which the list associated with v stores all the neighbours of v in G. The space complexity of this representation is $O(n + m)$ (which is much better than $O(n^2)$ when $m \ll n^2$)) in the Word RAM model where we set the length of the words to be $O(\log n)$ so that vertex labels can be stored in a single word.

With the adjacency list representation, the query "is i adjacent to j" requires traversing a linked list and therefore has time complexity $O(n)$. But identifying all the neighbours of a vertex v can be done in time $O(\deg(v))$—we will use this fact in our graph exploration algorithms.

2. Breadth-first search

There is a basic building block used in many different graph algorithms: the exploration problem. In this problem, from a starting vertex $s \in V$, we wish to visit all the vertices that are reachable from s in the graph G.

Definition 11.1. An instance of the Graph Exploration problem is a graph $G = (V, E)$ and a vertex $s \in V$. A valid solution to this instance is a list of all the vertices in the connected component of G that contains s.

There are two fundamental algorithms for solving the Graph Exploration problem: Breadth-first search (BFS), and Depth-first search (DFS).

The *breadth-first search* algorithm can be thought of as the cautious explorer: first, it finds all the vertices that can be reached from s by following only a single edge, then it finds all the vertices that can be reached by following 2 edges from s, then the ones reachable by following 3 edges from s, etc.

In the implementation of breadth-first search, each vertex in the graph is assigned one of three statuses: `undiscovered`, `discovered`, and `explored`. Initially, all vertices except the start vertex s are marked `undiscovered`, and s itself is marked `discovered`. Then at every step of the search, we choose one `discovered` vertex v and traverse the list of list of

its neighbours, marking all of the ones that are `undiscovered` as `discovered` and ignoring the other ones. Once we have finished visiting the list of neighbours of v, we change its status as `explored`. We then continue the process with another `discovered` vertex until no vertices with that status are left.

Breadth-first search, as described above, can be implemented efficiently with the adjacency list representation of vertices by using a queue to store the `discovered` vertices.

Algorithm 30: BFS$(G = (V, E), s)$

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
status[s] ← discovered;
Q ← {s};
L ← ∅;
while Q ≠ ∅ do
    v ← Dequeue(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            Enqueue(Q, w);
    Add(L, v);
    status[v] ← explored;
return L;
```

It's worth seeing the BFS algorithm in action by going over an example.

[See CLRS §22 for a complete example.]

Let's wait a little bit before proving the correctness of the BFS algorithm: it is easiest to do so when we consider the distance variant of the algorithm in the next section. For now, let's jump ahead to examine the time complexity of the algorithm.

Theorem 11.2. *The time complexity of the BFS algorithm is $O(n + m)$.*

Proof. By only adding vertices in the queue when they are `undiscovered` and by changing their status to `discovered` when we do so (and by never resetting the state of any vertex to `undiscovered` after the initialization), we guarantee that each vertex is added to the queue at most once, so that the total number of queue operations is $O(n)$ and since those operations take constant time, the time complexity of those operations is $O(n)$.

Furthermore, since the adjacency list of each vertex is visited at most once (when that vertex is dequeued) and a constant amount of time is spent for each element of those lists, the total amount of time spent in the inner for loop is $O(m)$.

The initialization also has time complexity $O(n)$, so the total time complexity of the BFS algorithm is $O(n) + O(n) + O(m) = O(n + m)$. □

3. Applications of BFS

Breadth-first search is a very simple algorithm, but it is amazingly useful: there are many fundamental problems on graphs that can be solved with only very minor modifications to the BFS algorithm. Let's examine a few.

3.1. Single-source shortest path. A common task when dealing with graphs is finding the shortest path between vertices. One of the common variants of this problem is the *single-source shortest path*, where we want to find the shortest path between a special source vertex s and all the other vertices in the graph.

DEFINITION 11.3 (Single-source shortest path (SSSP)). Given a graph $G = (V, E)$ and a vertex $s \in V$, find the shortest path between s and every vertex in $V \setminus \{s\}$.

(When s and v are in different conected components in G, by convention we say that the shortest path between s and v has length ∞.)

The BFS algorithm already essentially solves the SSSP problem, since the first time that we visit a vertex v, it is always by visiting the last edge of a shortest path to v. All that we need to do to complete the algorithm is add a few extra lines to the BFS algorithm.

Algorithm 31: BFS-SSSP($G = (V, E), s$)

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
    dist[v] ← ∞;
status[s] ← discovered;
dist[s] ← 0;
Q ← {s};
while Q ≠ ∅ do
    v ← Dequeue(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            dist[w] ← dist[v] + 1;
            Enqueue(Q, w);
    status[v] ← explored;
return dist;
```

We are now ready to prove the correctness of both the original BFS algorithm and the BFS-SSSP algorithm. The key statement in both proofs of correctness is the following assertion.

LEMMA 11.4. *Fix any graph $G = (V, E)$ with diameter D. For each $d = 0, 1, 2, \ldots, D+1$, there is a moment in the BFS-SSSP algorithm's execution on G at which point*

1. *All nodes at distance at most d from s have their distance correctly set;*
2. *All other nodes have their distance set to ∞; and*
3. *The set of nodes that are in the queue (and, equivalently, the set of nodes at state* `discovered`*) is the vertices at distance exactly d from s.*

PROOF. We prove the statement by induction on d. In the base case, when $d = 0$ the three conditions hold right before we enter the while loop.

For the induction step, assume that the statement is true for distance $d - 1$. Let's say that there are exactly k nodes at distance $d-1$ from s that are in the queue at that moment. Consider what happens after we run the while loop k more times. Then all those nodes

have been processed, and all the neighbours of those k nodes that had not been previously discovered are added to the queue and have their distance set to d. But these are exactly the nodes that are at distance exactly d from s—they all have distance at most d from s since they are reachable by following one more edge from a vertex at distance $d-1$ from s, and they are at distance at least d since otherwise, by the induction hypothesis, they would have been discovered and had their distance set earlier. The remaining nodes at distance at least $d+1$ from s still have their distance set to ∞, so the three conditions hold for d as well. □

The proof of correctness of the BFS-SSSP and BFS algorithms follow almost immediately from the Lemma.

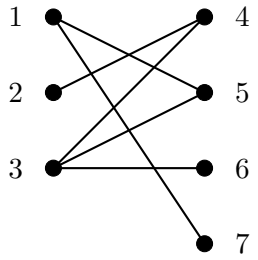
THEOREM 11.5. *The BFS-SSSP algorithm solves the Single-Source Shortest Path problem.*

PROOF. The lemma guarantees that when the algorithm exits the while loop, all nodes that are reachable from s have the correct distance set and all other ones still have distance ∞. □

THEOREM 11.6. *The BFS algorithm solves the Graph Exploration problem.*

PROOF. In the original BFS algorithm, instead of keeping track of the distance of each vertex to s, we added it to the list of vertices reachable from s that is output at the end of the algorithm. Therefore, the list of vertices output by BFS corresponds exactly to the set of vertices with finite distance reported by BFS-SSSP, which is just the set of vertices in the same connected component as s. □

3.2. Testing bipartiteness. A graph $G = (V, E)$ is *bipartite* if there is a partition of V into two parts V_L and V_R such that every edge in E connects a vertex in V_L to one in V_R. For example, the following graph is bipartite,



but if we add the edge $(2, 3)$ to the above graph, it is no longer bipartite.

DEFINITION 11.7 (Testing bipartiteness problem). Given a connected graph G, determine if it is bipartite or not.

The BFS algorithm can be modified easily to solve this problem. We start choosing any arbitrary vertex s and put it in the Left part V_L. (This is again an arbitrary choice that does not matter: we can always flip the labels of the parts V_L and V_R of a bipartite graph without changing the graph itself.) Then every time we visit a new vertex, it must be in the other part of the graph than the neighbour that led to this vertex. So whenever we discover a vertex, we also know which of the two parts V_L or V_R to assign it to, if the graph is indeed bipartite.

It remains to verify that the graph is actually bipartite: this is done simply by checking that whenever we encounter an edge between two vertices that have already been discovered, those two vertices are in different parts. This will be true for all edges if and only if the graph is bipartite. The resulting algorithm is as follows.

Algorithm 32: BFS-Bipartite($G = (V, E)$)

```
s ← any vertex in V;
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
status[s] ← discovered;
InLeftPart[s] ← True;
Q ← {s};
while Q ≠ ∅ do
    v ← Dequeue(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            InLeftPart[w] ← Not(InLeftPart[v]);
            Enqueue(Q, w);
        else if InLeftPart[v] == InLeftPart[w] then
            return False;
    status[v] ← explored;
return True;
```

The proof of correctness of the algorithm is left as an exercise. Another good exercise is to see how to generalize the algorithm so that it can test whether *any* graph (connected or not) is bipartite. (Hint: a graph is bipartite if and only if all its connected components are bipartite.)

3.3. Spanning tree. A *spanning tree* of a connected graph $G = (V, E)$ is a tree $T = (V, E')$ where $E' \subseteq E$ and T is also a connected graph. Spanning trees are useful for a number of reasons; for instance, they can be used to store a sparse representation of G (with only $n - 1$ edges) that still enables us to find a path in G between every two vertices $v, w \in V$.

Definition 11.8 (Spanning tree problem). Given a connected graph $G = (V, E)$, output a spanning tree of G.

We can represent a rooted tree with an array of n nodes, where the entry for node v is its parent in the tree and the entry for the root simply has $\emptyset$ as its parent. The following simple modification of the BFS algorithm computes a spanning tree of connected graphs using this representation.

Algorithm 33: BFS-SpanningTree($G = (V, E), s$)

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
status[s] ← discovered;
parent[s] ← ∅;
Q ← {s};
while Q ≠ ∅ do
    v ← DEQUEUE(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            parent[w] ← v;
            ENQUEUE(Q, w);
    status[v] ← explored;
return parent;
```

We again leave the proof of correctness of this algorithm as an exercise. We will also revisit spanning trees in later lectures.

LECTURE 12

Depth-first search

In the last lecture, we saw how the breadth-first search algorithm can solve the Graph Exploration problem (as well as a number of other similar problems, with appropriate slight modifications to the base BFS algorithm). Today, we examine a different algorithm for solving the same Graph Exploration problem: depth-first search.

1. Depth-first search

The breadth-first search algorithm can be described as a careful, thorough method for exploring the graph (which explores all the vertices at distance d form the source before moving on to explore any vertices at distance $d + 1$). The depth-first search algorithm, by contrast, corresponds to an impatient and impulsive method for exploring a graph: whenever it discovers a new vertex, the depth-first search seeks to explore at least one of its incident edges. This can still result in a provably correct algorithm for solving the Graph Exploration problem; it just explores the vertices in a different order than the BFS does.

We can implement the depth-first search algorithm by modifying the BFS algorithm to use a stack instead of a queue. It's also possible to implement depth-first search with a simple recursive algorithm. For this algorithm, we assume that visited is a global array of n Boolean values, and we put in placeholder functions PreVisit and PostVisit that we will define later.

Algorithm 34: Explore$(G = (V, E), v, p)$

```
visited[v] ← True;
PreVisit(v, p);
for each w ∈ Adj(G, v) do
    if not visited[w] then Explore(G, w, v);
PostVisit(v, p);
```

Algorithm 35: DFS$(G = (V, E), s)$

```
for each v ∈ V do
    visited[v] ← False;
Explore(G, s, ∅);
```

To solve the Graph Exploration problem, we simply need to implement the PreVisit or the PostVisit function to add v to a list of vertices in the same connected component as s and output that list at the end. The correctness of the resulting algorithm follows from the following argument.

THEOREM 12.1. *The DFS algorithm calls* EXPLORE *once on every vertex that is in the same connected component as s, and on no other vertex of V.*

PROOF. That EXPLORE is called at most once on every vertex in V follows from the fact that we only call it on a vertex when its status is visited$[v]$ = `False` and that the status is updated to `True` as soon as we call EXPLORE on that vertex.

The vertices that are not in the same connected component as s in G are never explored because the DFS exploration follows only the edges of the graph.

And to show that DFS explores *all* the vertices in the same connected component as s, assume on the contrary that it misses some vertex u that is in that connected component. Since s and u are in the same connected component, we can find a simple path $s, v_1, v_2, \ldots, v_k, u$ from s to u in G. Let $z = v_i$ be the last vertex along the path that is visited by the DFS algorithm, and let $w = v_{i+1}$ be the first vertex along the path that is not visited. (Such a pair of vertices must exist if s was visited and u was not). But then we obtain a contradiction, since the EXPLORE call on z examines all the neighbours of z and explores those vertices that have not yet been visited, so it would call EXPLORE on w. □

For the time complexity of the algorithm, we see that EXPLORE is called on at most n vertices, and the for loop is executed $O(m)$ times since each edge is visited at most twice. The total time complexity of the algorithm is therefore $O(n + m)$, the same as the BFS algorithm.

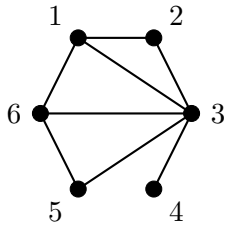
2. Spanning tree

Just as we can easily modify the BFS algorithm to generate a spanning tree of a connected graph, we can also do this with the DFS algorithm. In this case, we simply need to let $T = (V, E')$ be a global tree. Initially, the set of edges of the tree is taken to be $E' = \emptyset$, then we implement PREVISIT to add the edge between v and its parent p in E'.

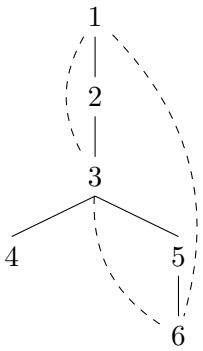
Algorithm 36: PREVISIT(v, p)

if $p \neq \emptyset$ **then** $E' \leftarrow E' \cup \{(v, p)\}$;

Let's look at an example. When we run the resulting DFS algorithm on the graph



we obtain the following tree (assuming the neighbours of a given vertex are visited in order).



The extra edges of the graph that are not included in the spanning tree are drawn with a dotted line in this example.

What is particularly useful about the spanning tree obtained by the DFS algorithm is that it captures some important structural information about the original connected graph G. Let us call the edges contained in the spanning tree defined by the DFS algorithm *tree edges*, and let the remaining edges of G be called *non-tree edges*. Furthermore, when a vertex v is in the subtree rooted at w in the spanning tree, we will say that v is a *descendant* of w and that w is an *ancestor* of v.

LEMMA 12.2. *Every non-tree edge in a connected graph G connects an ancestor to one of its descendants.*

PROOF. Consider any non-tree edge $(u, v) \in E$. Say that u was discovered first in the DFS of G. Then every node that is discovered while following a path from u to other unexplored vertices ends as a descendant of u; v must be one of those vertices because otherwise it would be explored directly from u (and (u, v) would be a tree edge) before we complete the exploration of u. □

We can get even more structural information from the DFS tree by not only building the tree itself but also keeping track of the times at which (or the order in which) each node is first discovered and when it is completely explored. We can do this easily with a `clock` counter that we initially set to 0 and use in the following implementations of PREVISIT and POSTVISIT.

Algorithm 37: PREVISIT(v, p)

$pre[v] \leftarrow$ `clock`;
`clock` $\leftarrow$ `clock` $+ 1$;

Algorithm 38: POSTVISIT(v, p)

$post[v] \leftarrow$ `clock`;
`clock` $\leftarrow$ `clock` $+ 1$;

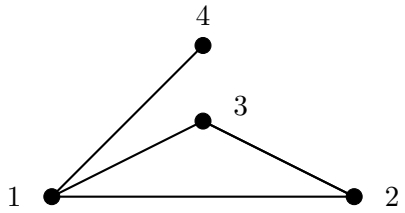
The theorem we established about the structure of non-tree edges can also be rephrased in terms of the intervals defined by these timings.

THEOREM 12.3. *For any two vertices $u, v \in V$ in a connected graph $G = (V, E)$, either the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are disjoint, or one is contained within the other one.*

We could continue to work on developing more structural results that we get with the DFS tree, but let's stop here and see how the DFS tree and timing information can be used to determine just *how* well connected a graph is.

3. Finding cut vertices

A graph is connected, as we already saw, if there is a path between any two vertices in the graph. Some graphs are connected, but "barely so", in the sense that if we remove one of the vertices of the graph, the resulting subgraph is no longer connected. For example, in the graph



if we remove vertex 1 then we are left with a graph that is no longer connected. Vertex 1, in this case, is known as a *cut vertex*.

DEFINITION 12.4. A vertex $v \in V$ is a *cut vertex* in a connected graph $G = (V, E)$ if the subgraph $G' = (V \setminus \{v\}, E')$ with $E' = \{(u, w) \in E : u, w \neq v\}$ is not connected. A graph that has no cut vertex is called *2-connected* (or sometimes *2-vertex-connected*).

These definitions immediately suggest the problem of determining if a graph is 2-connected or not.

DEFINITION 12.5 (Testing 2-connectedness). Given a connected graph $G = (V, E)$, determine if it is 2-connected or not. (Equivalently: determine if it has a cut vertex.)

We can use the DFS tree of a graph to solve this problem. As a first step, we can easily test if the root of that tree is a cut vertex.

LEMMA 12.6. *The root v of the DFS tree of a connected graph $G = (V, E)$ is a cut vertex in G if and only if it has at least 2 children.*

PROOF. If the root v has at most 1 child, then removing the node v leaves a tree (or an empty graph), so that G' is still connected.

If the root v has at least 2 children, then Lemma 12.2 guarantees that there can be no edge connecting a vertex u from the left subtree to a vertex w in the right subtree, so that any path between a vertex in one subtree to the other must pass through v. Therefore, v is a cut vertex. □

There is also a similar lemma for the other vertices of the DFS tree.

LEMMA 12.7. *The non-root vertex v of the DFS tree T of a connected graph $G = (V, E)$ is a cut vertex in G if and only if it has a child u such that the subtree of T rooted at u has no non-tree edge going from one of its vertices to an ancestor of T.*

PROOF. If the condition is satisfied, then removing v from G isolates the subtree rooted at u from the rest of the graph, so the resulting subgraph is not connected.

In the other direction, if v is a cut vertex of G then removing it must isolate some vertices from the rest of the graph, which means that one of its subtrees must be disconnected from the rest of the tree and, in particular, has no non-tree edge going to any ancestor of v. □

So the problem of determining if a graph has a cut vertex can be reduced to the problem of determining if a vertex satisfies either of the above conditions. The condition for the root of the tree is easily checked. And for the other nodes, we can determine if the condition is satisfied by using the $pre[v]$ values to keep track of the earliest ancestor connected by an edge from the current subtree. This is implemented with slight changes to the EXPLORE function.

Algorithm 39: EXPLORE'$(G = (V, E), v, p)$

```
if p = ∅ and |Adj(G, v)| ≥ 2 then
    Print v is a cut vertex;
visited[v] ← True;
PreVisit(v, p);
lowest[v] ← pre[v];
for each w ∈ Adj(G, v) do
    if not visited[w] then
        Explore'(G, w, v);
        lowest[v] ← min{lowest[v], lowest[w]};
        if p ≠ ∅ and lowest[w] ≥ pre[v] then
            Print v is a cut vertex;
    else
        lowest[v] ← min{lowest[v], pre[w]};
PostVisit(v, p);
```

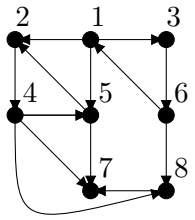
LECTURE 13

Depth-first search in directed graphs

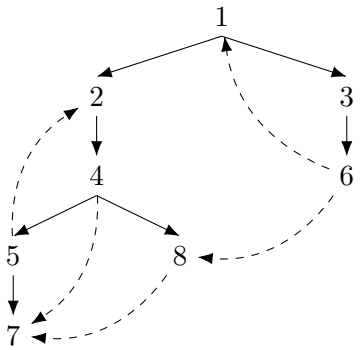
In the first lecture of this section of the course, we introduced directed (or *ordered*) graphs, but so far we have only considered algorithms on undirected graphs. In this lecture, we now turn our attention to directed graphs.

1. DFS tree in directed graphs

Recall that a directed graph is a graph where each edge is directed between a *source* vertex and a *destination* vertex. We can run the DFS algorithm as-is on directed graphs (following edges only in the direction from their source to their destination), and the algorithm again builds a DFS tree. For example, on the input graph



the DFS algorithm builds the following DFS tree



As the example shows, in the directed graph setting, we are no longer guaranteed to have non-tree edges connecting only ancestors and descendants. In this setting, the non-tree edges belong to one of three categories: *forward edges* that lead from an ancestor to a descendant, *backward edges* that lead from a descendant to one of its ancestors, and *cross edges* to lead from a vertex to another one that is neither its ancestor nor its descendant. In the DFS tree above, for example, the edge $(5, 7)$ is a forward edge, the edges $(6, 1)$ and $(5, 2)$ are backward edges, and the edges $(6, 8)$ and $(8, 7)$ are cross edges.

2. Directed acyclic graphs

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to \cdots \to v_k \to v_0$. (Note that unlike in undirected graphs, the path $v_0 \to v_1 \to v_0$ is a cycle.) A directed graph that contains no cycle is known as a *directed acyclic graph*, or *dag* for short: these graphs are particularly important, and appear in many different contexts. It's therefore important to be able to tell efficiently when a graph is a dag.

DEFINITION 13.1 (Testing DAGs). Given a directed graph $G = (V, E)$, determine if it is a directed acyclic graph.

We can solve the problem of testing dags using depth-first search.

THEOREM 13.2. *A directed graph $G = (V, E)$ contains a cycle if and only if its DFS tree contains a backward edge.*

PROOF. If (u, v) is a backward edge in the DFS tree for G, then the path from v to u in the tree along with the edge (u, v) forms a cycle.

If G contains a cycle, we can write it as $v_0 \to v_1 \to \cdots \to v_k \to v_0$ where v_0 is the first vertex in the set $\{v_0, v_1, \ldots, v_k\}$ to be discovered by the DFS. Then each vertex $v_1, \ldots, v_k$ is a descendant of v_0 in the DFS since the subtree rooted at v_0 includes all the vertices reachable from v_0 in G. So the edge (v_k, v_0) must be a backward edge. □

We then have a simple algorithm for checking if a directed graph is a DAG: run the DFS algorithm, and check if any of the non-tree edges encountered along the way is a backward edge. (Exercise: see how you can use the $post[v]$ values to determine if an edge (u, v) is a backward edge or not.)

3. Topological sorting of DAGs

DAGs can be used to represent chains of dependencies. For example, if I want to make a fancy cup of coffee, I need to grind some beans, boil water, pour the water over the beans, prewarm the cups, etc. Some of these tasks can be done in any order—whether I grind the beans first or boild the water first does not (really) matter—but other tasks can only be done after others have been completed—pouring the water over the beans before you boil it will not give the same result! A directed graph can represent the tasks (as nodes) and the dependencies between the tasks (as directed edges); when the graph is a DAG, it means that there are no circular dependencies.

If the DAG represents a set of tasks that need to be completed and the edges represent dependencies between the tasks, the problem of determining in what order we should complete all the tasks (so that we complete a task only when all the ones that it depends on have already been completed) corresponds to the problem of *linearizing*, or *ordering*, a DAG.

DEFINITION 13.3 (Linearizing DAGs problem). Given a directed acyclic graph $G = (V, E)$, find an order $v_1, \ldots, v_n$ of the vertices in V such that for every directed edge $(v_i, v_j) \in E$, we have $i < j$.

It's not immediately clear that every DAG can be linearized, much less that we can find a linearization easily. But, as it turns out, the DFS algorithm already does so!

THEOREM 13.4. *For any directed acyclic graph $G = (V, E)$, after we run the DFS algorithm, every edge $(u, v) \in E$ satisfies $post[u] > post[v]$.*

PROOF. PostVisit(u) will be called only when we have finished exploring all of the vertices that are reachable from u (including v itself). □

We can then sort the vertices of G by their post number to linearize G or, alternatively, build the sorted order as we perform the DFS itself with a slight modification of the PostVisit function.

There are two types of nodes in a DAG worth introducing here because they will appear again in later lectures:

DEFINITION 13.5 (Source and sink vertices). A *source vertex* in a dag $G = (V, E)$ is a vertex with in-degree 0. A *sink vertex* in G is a vertex with out-degree 0.

In a linearization of G, the first vertex must be a source vertex, and the last vertex must be a sink vertex.

4. Strong connectivity

A directed graph $G = (V, E)$ is *strongly connected* if for every pair of vertices $u, v \in V$, there is a path from u to v and a path from v to u in G. Can we efficiently determine whether a directed graph is strongly connected or not?

DEFINITION 13.6 (Testing strong connectivity). Given a directed graph $G = (V, E)$, determine whether it is strongly connected.

When we run the DFS algorithm on a directed graph G from the start vertex s, we find the set of vertices in V that are reachable from s in G—but it's not always the case that there is also always a path from those vertices to s in G.

However, it is easy to find the set of vertices $v \in V$ for which there is a path from v to s in G: run DFS on the directed graph G^R obtained by flipping the direction of every edge in G!

So by running the DFS algorithm on G and on its reversal G^R, we can test whether s is strongly connected to every vertex in the graph. The following lemma shows that this is enough to test if the graph G is strongly connected.

LEMMA 13.7. *Fix any vertex $s \in V$. The directed graph $G = (V, E)$ is strongly connected if and only if for every $v \in V$, there is a path from s to v and from v to s in G.*

PROOF. If G is strongly connected, the conclusion is true for every pair of vertices $u, v \in V$, so it is also true for every pair where $u = s$.

Conversely, if there is a path from every v to s and from s to every v, then for any pair $u, v \in V$, there is a path from u to v obtained by following the path from u to s and then the path from s to v, and there is a path from v to u by going from v to s and from s to u, so G is strongly connected. □

4.1. Finding strongly-connected components. In the setting of undirected graphs, we saw that every graph can be partitioned into disjoint connected components. And it is easy to find the set of connected components of a graph: run a depth-first or breadth-first search from any initial vertex $s \in V$ to find the first connected component, then repeat the same process on any vertex that is not covered by any of the connected components identified so far until no such vertices remain.

In the setting of directed graphs, there is a natural analogue of connected components: a *strongly-connected component* of a directed graph $G = (V, E)$ is a maximal set $S \subseteq V$ of

vertices where for every pair $u, v \in S$, there is a path from u to v and a path from v to u in G.

Can we modify the strong connectivity test above to find the strongly-connected components of a directed graph? This is not nearly as simple as in the undirected setting! In fact, there are quite a few clever algorithms that have been designed for this problem, using some of the ideas that we have seen in today's lecture. I highly recommend trying this yourselves before you look at our textbooks to see those solutions:

CHALLENGE 13.8. *Design an algorithm that identifies the strongly-connected components of a directed graph in time* $O(n + m)$.

LECTURE 14

Minimum spanning trees

Today, we examine weighted graphs and focus on a particularly fundamental problem: finding the minimum spanning tree of a graph.

1. Spanning trees

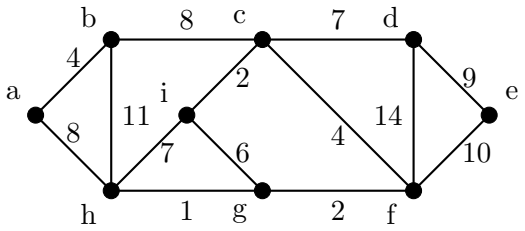
Recall that a *spanning tree* of a connected undirected graph $G = (V, E)$ is a tree $T = (V, E')$ whose edges $E' \subseteq E$ are a subset of the edges of G. We can think of a spanning tree as a "minimum skeleton representation" of a graph: using a spanning tree T of G, we can identify a path in G between any two vertices $u, v \in V$, and yet to store T we only need to store $n - 1$ edges (instead of as many as $\Omega(n^2)$ for G itself).

We already saw that finding a spanning tree of a connected graph can be done in time $O(n + m)$ by running either a depth-first search or a breadth-first search and storing the resulting DFS or BFS tree.

In general, a connected graph G can have multiple spanning trees. Depending on the application we have in mind, we may want to find the "best" spanning tree of a graph. This problem becomes particularly important when we consider *weighted* graphs.

Definition 14.1 (Weighted graphs). A *weighted graph* is a graph $G = (V, E)$ and a function $w : E \to \mathbb{R}$, where for each edge $e \in E$, the value $w(e)$ is the *weight* of e in G.

The following is an example of a weighted graph.



Weighted graphs as we have defined them are sometimes called *edge-weighted graphs*, to distinguish them from the graphs for which we assign weights to the vertices instead of the edges.

There are many problems that map nicely to weighted graphs. For example, with the graph obtained by representing airports with vertices and connecting two vertices with an edge if and only if there is a direct flight going between the two corresponding airports, we can use weights to represent the time of the flight, or the cost of a ticket on one of those flights, for example. The weight of the edges can then be used to solve problems such as finding the cheapest route between two airports, or the one with the shortest total flight time, or other similar problems.

The weights on edges give a very natural way to compare different spanning trees, which gives rise to the *mininum spanning tree* problem.

DEFINITION 14.2 (Minimum spanning tree). An instance of the *minimum spanning tree (MST)* problem is a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}$. A valid solution to this problem is a spanning tree $T = (V, E')$ of G with minimum total weight $\sum_{e \in E'} w(e)$.

The typical example that you can keep in mind when considering the MST problem is the road building problem: you have a number of cities that you want to connect with roads (or with high-speed train lines!). We can represent the pairs of cities that you can connect with a road using edges; the weight of each edge represents the cost of building that road. (Not all vertices need be connected by an edge, as you may not be able to build a road directly between two cities if, for example, they are separated by a formidable mountain chain.) Then the cheapest way to connect all the cities by roads corresponds to the minimum spanning tree of the resulting graph.

2. Kruskal's algorithm

There is a natural greedy strategy for constructing the minimum spanning tree of a graph G: consider the edges of G in order of increasing weight (i.e., from the lightest to the heaviest), adding an edge to our tree whenever its endpoints are two vertices that have not already been connected. This greedy algorithm is known as *Kruskal's algorithm*.

Algorithm 40: Kruskal$(G = (V, E), w)$

```
Sort the edges in E by increasing weight w;
E' ← ∅;
for each edge (u, v) ∈ E do
    if Component(E', u) ≠ Component(E', v) then
        E' ← E' ∪ {(u, v)};
return T = (V, E');
```

To fully specify Kruskal's algorithm, we must also define the Component function and ways to check if two components are identical. Let us leave this task for later. For now, let's treat that algorithm as a black box and show that Kruskal's algorithm returns a minimum spanning tree of the input graph G. You can see how Kruskal's algorithm finds a minimum spanning tree in the example graph at the beginning of this lecture in the course textbook (CLRS).

To show that Kruskal's algorithm correctly solves the MST problem on all graphs, we establish a powerful lemma that can also be used to prove the correctness of other MST algorithms as well.

DEFINITION 14.3. A *cut* in a graph $G = (V, E)$ is a partition of the vertices V into two sets $(S, V \setminus S)$. An edge is said to *cross* the cut if it connects a vertex in S to one in $V \setminus S$.

LEMMA 14.4 (Cut property). *Let X be a set of edges that are part of a minimum spanning tree of the connected graph $G = (V, E)$. Pick any set $S \subseteq V$ such that no edge in X crosses the cut $(S, V \setminus S)$ and let e be the lightest edge that crosses this cut. Then $X \cup \{e\}$ is part of a minimum spanning tree of G.*

PROOF. Let T be a MST of G that includes all the edges in X. If T also includes the edge e, then we are done. Otherwise, consider the graph obtained by adding the edge $e = (u, v)$ to T. Since T is a spanning tree of G, it already contains a path connecting

u to v. Thus, the new edge e must be part of a cycle in $T \cup \{e\}$. And since a cycle containing e includes vertices in both S and in $V \setminus S$ (namely, u and v), it must include another edge e' that also crosses the cut $(S, V \setminus S)$. If we remove e', we obtain the graph $T' = (T \cup \{e\}) \setminus \{e'\}$. This graph T' is connected because removing a vertex from a cycle never disconnects a graph, and it contains the same number of edges as T so it must also be a tree; it is therefore a spanning tree of G. Furthermore, the weight of T' satisfies

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

But $w(e) \leq w(e')$ since e is the lightest edge crossing the cut $(S, V \setminus S)$, so $\text{weight}(T') \leq \text{weight}(T)$ and T' is a minimum spanning tree of G that contains all the edges in $X \cup \{e\}$. □

The proof of correctness of Kruskal's algorithm follows easily from the cut property.

THEOREM 14.5. *Kruskal's algorithm solves the Minimum Spanning Tree problem.*

PROOF. We proceed by induction, showing at every step of Kruskal's algorithm, there is a minimum spanning tree of G that contains the set E' of edges that have been selected up to that point. The base case is the initial set $E' = \emptyset$, for which the assertion is trivially true.

For the induction step, the induction hypothesis states that the set $X = E'$ of edges already selected by the algorithm are part of a MST of G. The next edge e added by Kruskal's algorithm connects two previously unconnected components; call them T_1 and T_2. Let S be the set of vertices in T_1. Then X does not have any edges that cross the cut $(S, V \setminus S)$ and by the order in which we consider the edges e is the lightest edge that crosses the cut, so $X \cup \{e\}$ is also part of a MST of G, as we wanted to show. □

2.1. Implementing Kruskal's algorithm. How efficient is Kruskal's algorithm? Naïvely, we might think that it runs in time $\Theta(m \log m)$ since it sorts the edges by weight then considers each edge only once. But there is one non-trivial catch: to obtain this runtime, we need an efficient algorithm for checking if two vertices are in different connected components of the graph. We can do this with DFS and BFS algorithms, but this implementation is not very efficient; to obtain a $O(m \log m)$ time complexity for Kruskal's algorithm, we need to use a more sophisticated approach. This can be done with the *Union Find* data structure, a topic that is covered in CS 466.

3. Prim's algorithm

There is another very natural greedy approach to building the minimum spanning tree of a graph G, that follows almost immediately from the cut property. At each step, we have a tree connecting a subset of the vertices of the graph (initially it is a single vertex) and we grow the tree by finding the lightest edge that connects the tree to another previously-unconnected vertex.

We can again use the Cut Property to prove the correctness of this algorithm.

THEOREM 14.6. *Prim's algorithm solves the Minimum Spanning Tree problem.*

PROOF. By definition, at every iteration of the while loop the edges in E' do not cross the cut $(S, V \setminus S)$ so the correctness of the algorithm again follows from the Cut Property. □

Algorithm 41: Prim$(G = (V, E), w)$

```
Initialize S = {v} for any v ∈ V;
Initialize E' = ∅;
Sort the edges in E by weight w;
while S ≠ V do
    (u, v) ← FindMinCutEdge(E, S);
    E' ← E' ∪ {(u, v)};
    S ← S ∪ {u, v};
return T = (V, E');
```

3.1. Implementation. Once again, it is not immediately obvious how to implement Prim's algorithm most efficiently. This can be done with a priority queue. A natural implementation of this approach has time complexity $O(m \log m)$. It is also possible to use Fibonacci heaps to do even better, with a final time complexity of $O(m + n \log n)$. See the course textbook for the details.

LECTURE 15

Single-source shortest paths

We saw in the last lecture that greedy algorithms can be used to find minimum spanning trees of weighted graphs. Today we will see how greedy algorithm can also be used to solve the Single-source shortest path (SSSP) problem on weighted directed graphs.

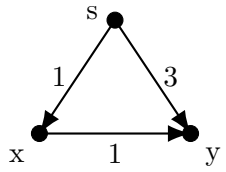
1. SSSP with Small Integer Weights

The Single-source shortest path problem is generalized in the natural way to weighted graphs.

DEFINITION 15.1. In the *Single-source shortest path (SSSP)* problem on weighted graphs, our input is a (directed or undirected) graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and a source vertex $s \in V$. The valid solution is the minimum total weight of a path in G from s to v for every vertex $v \in V$.

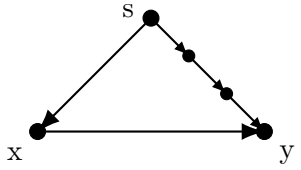
If we consider the special case of the problem where the weight of each edge is 1, then we are back to the original unweighted version of the SSSP problem and we can solve this problem in time $O(m + n)$ using breadth-first search.

What if we now consider a slightly more general setting where all the edges of the graph have weight 1,2, or 3?



Running BFS as-is on the graph G (while ignoring the edge weights) no longer solves the SSSP problem in this setting, as we see in the example above when the BFS starts at vertex s: since vertex y is reachable directly from s by following an edge of weight 3, this is how y will be discovered by the BFS and the solution built in this way will claim that y is at distance 3 from s, whereas there is a shorter path of length 2 between the two vertices in the graph.

We can, however, reduce the SSSP problem on graphs with edge weights 1, 2, or 3 to the SSSP problem on unweighted graphs: all we need to do is add extra (dummy) vertices to the graph to split up longer edges appropriately.



This guarantees that BFS now visits the vertices in order of increasing distance (as measured by the weights of the edges of the original graphs, not just the number of edges followed) and that when we output the distance from the source to each non-dummy nodes, we correctly solve the SSSP problem. The time complexity of the algorithm is $O(m' + n')$ when m' and n' are the number of edges and vertices of the unweighted graph we created with the dummy variables. This is still $O(m + n)$ when the weights are only 1, 2, or 3, but this complexity grows very quickly if we allow (much) larger weights. To handle that scenario more efficiently, we want to consider a slightly more sophisticated approach.

2. SSSP with Nonnegative Weights: Dijkstra's algorithm

Consider now the SSSP problem on graphs with arbitrary *non-negative* edge weights. We can solve this problem by modifying BFS instead of trying to use it directly: let's design an algorithm that visits the vertices in order of distance from the start vertex s. We can do this with a Priority Queue. The resulting algorithm is known as *Dijkstra's algorithm*.

Algorithm 42: Dijkstra$(G = (V, E), w, s)$

```
for all v ∈ V do
    dist[v] ← ∞;
dist[s] ← 0;
H ← MakePriorityQueue(V, dist);
while H ≠ ∅ do
    u ← DeleteMin(H);
    for all v ∈ AdjList(E, u) do
        if dist[u] + w(u, v) < dist[v] then
            dist[v] ← dist[u] + w(u, v);
            DecreaseKey(H, v, dist[v]);
return dist;
```

We say that Dijsktra's algorithm is a greedy algorithm because at each step it explores the vertex that is closest to the source among all of those that have not been visited yet. You can see an example of the execution of Dijkstra's algorithm on a weighted graph in the course textbook.

Informally, Dijsktra's algorithm is correct because we can never have a shortest path from s to a vertex v that goes through a vertex u which is further away from s than v. (Note that this will no longer be true when we allow negative edge weights!) We can formalize this argument with a proof by induction.

THEOREM 15.2. *Fix any directed weighted graph G with non-negative edge weights. At every iteration of the while loop in Dijkstra's algorithm, the distance $\texttt{dist}[v]$ from s to every vertex v that is not in H is correct.*

PROOF. We prove the theorem by induction on the number of vertices that are not in H.

In the base case, when only s is removed from H, the theorem is true since we initialize $\texttt{dist}[s] = 0$.

For the induction step, we assume that it is true for all the vertices that have been removed from H before the current iteration of the loop and we want to show that it is also true for the vertex u removed from the priority queue in the current iteration.

That is: we want to show that

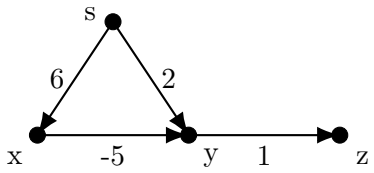
$$\texttt{dist}[u] = \min_{v \notin H}\{\texttt{dist}[v] + w(v,u)\}.$$

Let $v^* \notin H$ be a vertex where the minimum is attained. By the induction hypothesis, $\texttt{dist}[v^*]$ is the distance from s to v^*, so we do have that $\texttt{dist}[u] \leq \texttt{dist}[v^*] + w(v^*, u)$.

To show a matching lower bound on $\texttt{dist}[u]$, we want to show that every path P from s to u has length at least $\texttt{dist}[v^*] + w(v^*, u)$. Since P starts from a vertex (namely: s) that has previously been removed from H and ends at a vertex (u) that was not previously removed from H, there must be an edge (x, y) in P where $x \notin H$ and $y \in H$. Then the length of the path P must be at least $\texttt{dist}[x] + w(x, y)$ by the induction hypothesis and the fact that G has no edge with negative weight. The choice of u as the next vertex to remove from H implies that we must also have $\texttt{dist}[x] + w(x, y) \geq \texttt{dist}[v^*] + w(v^*, u)$. Therefore, $\texttt{dist}[u] \geq \texttt{dist}[v^*] + w(v^*, u)$. □

3. SSSP with Negative Edge Weights

3.1. Dijkstra's algorithm. If we consider graphs with negative edge weights, Dijkstra's algorithm no longer computes the correct weight of the shortest path from the source to every other vertex in the weighted graph. Consider for instance the following simple example.

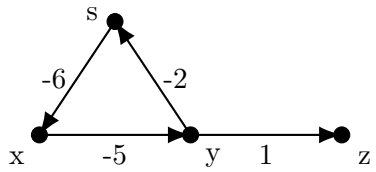


Dijkstra's algorithm will return the distances

$$\texttt{dist}[s] = 0 \qquad \texttt{dist}[x] = 6 \qquad \texttt{dist}[y] = 2 \qquad \texttt{dist}[z] = 3$$

whereas the distance from s to y is actually 1, from the length of the path $s \to x \to y$ and the distance from s to z is 2, not 3. We can modify Dijkstra's algorithm so that the distances to vertices are updated when we discover a shorter path to a vertex v obtained by following negative-weight edges, but we have to be careful when we do so to update the distances to all vertices whose shortest path goes through v as well. This means that we will no longer have a greedy algorithm. And so, instead of applying this ad hoc method for fixing the algorithm, it's worth taking a step back and seeing if another algorithm design technique can be used to solve the SSSP problem. Before we do so, however, there is another worrisome example that we need to consider.

3.2. Negative-weight cycles. What is the weight of the shortest path between s and z in the following graph?



The path $s \to x \to y \to z$ has total weight -10. But the path $s \to x \to y \to s \to x \to y \to z$ has total weight -23. And the path that follows the cycle $s \to x \to y \to s$ twice has total weight -36, etc. What we observe is that if a graph has a cycle with negative total weight, then the distance of vertices that can be reached via a path that goes through the cycle is unbounded (or can be set to $-\infty$).

For now, let's avoid the problems caused by negative weight cycles by only considering graphs that don't contain any.

3.3. Bellman–Ford algorithm. Let's now revisit the problem of designing an algorithm for SSSP with negative weights but no negative-weight cycles. Since greedy algorithms don't work for this problem, it's natural to try to design a Dynamic Programming solution for the problem. The key question when we do so is: what will be our simpler subproblems? Recall that we define

$$\texttt{dist}[v] = \text{length of the shortest path from } s \text{ to } v.$$

We obtain simpler subproblems if we don't consider *all* paths from s to v but only paths that use at most 1, 2, 3, ... edges. Specifically, for our subproblems we wish to compute

$$d_i[v] = \text{length of the shortest path from } s \text{ to } v \text{ that uses} \leq i \text{ edges}$$

for every $v \in V$ and for every $i = 1, 2, \ldots, n-1$. The base case is easy to compute:

$$d_1[v] = \begin{cases} w(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

And the values $d_i[v]$ are easy to compute once we have computed $d_{i-1}[u]$ for every $u \in V$:

$$d_i[v] = \min \begin{cases} d_{i-1}[v] \\ d_{i-1}[u] + w(u,v) \quad \text{for each } u \text{ such that } (u,v) \in E. \end{cases}$$

Putting everything together, we obtain the *Bellman–Ford algorithm*.

The time complexity of this algorithm is $O(n^3)$ when edge queries can be performed in constant time. We will see in the next lecture a slight reformulation of the same algorithm that runs in time $O(nm)$ when the graph is stored in the adjacency list model.

Algorithm 43: BellmanFord($G = (V, E), w, s$)

```
for all v ∈ V do
    if (s, v) ∈ E then
        d_1[v] ← w(s, v);
    else
        d_1[v] ← ∞;
for i = 2, 3, ..., n − 1 do
    for all v ∈ V do
        d_i[v] ← d_{i−1}[v];
        for all u ∈ V do
            if (u, v) ∈ E and d_{i−1}[u] + w(u, v) < d_i[v] then
                d_i[v] ← d_{i−1}[u] + w(u, v);
return d_{n−1};
```

LECTURE 16

All-pairs shortest paths

1. APSP Problem

We have seen some algorithms for solving the *Single-source shortest path* (SSSP) problem, where we want to compute the length of the shortest path in a graph $G = (V, E)$ between some special source vertex $s \in V$ and every other vertex in V. Today we explore a natural variant of the problem, where we want to compute the minimum distance between *all* pairs of vertices in V.

DEFINITION 16.1 (All-pairs shortest path (APSP)). In the *all-pairs shortest path (APSP)* problem, we are given a weighted (directed or undirected) graph $G = (V, E)$ and we must return the length of the shortest path from u to v for every $u, v \in V$.

As we did with the SSSP problem, we will consider only the special case of the problem where the input graph is guaranteed not to contain any negative-weight cycles.

There is a simple solution to the APSP problem: run the Bellman–Ford algorithm n times on the same graph with all possible vertices as the source vertex.

Algorithm 44: SIMPLEAPSP$(G = (V, E), w)$

```
for all u ∈ V do
    d[u, ∗] ← BELLMANFORD(G, w, u);
return d;
```

The correctness of this algorithm follows directly from the correctness of the Bellman–Ford algorithm. Recall that the Bellman–Ford algorithm is obtained via the dynamic programming technique by considering the shortest paths from s to v that go over only 1, 2, 3, $\ldots$, $n-1$ edges. We saw in the last lecture that we can store these distances in vectors $d_1, d_2, \ldots, d_{n-1}$. It turns out that we don't need to store all these distance vectors separately, and we can simplify the Bellman–Ford algorithm to use a single distance vector that we update as follows.

Algorithm 45: BellmanFord$(G = (V, E), w, s)$

```
for all v ∈ V do
    if (s, v) ∈ E then
        d[v] ← w(s, v);
    else
        d[v] ← ∞;
for i = 2, 3, ..., n − 1 do
    for all (u, v) ∈ E do
        if d[u] + w(u, v) < d[v] then
            d[v] ← d[u] + w(u, v);
return d;
```

See the textbook for more discussion and an analysis of this version of the algorithm. For today's lecture, the important point to note is that its time complexity is $\Theta(nm)$. This means that the SimpleAPSP algorithm we designed earlier has time complexity $\Theta(n^2m)$, which can be as large as $\Theta(n^4)$ when the graph is dense. Can we do better?

2. Modifying the Bellman–Ford algorithm

The first observation we might make of our SimpleAPSP algorithm is that it appears to be inefficient in that each call to the Bellman–Ford algorithm results in the computation of a lot of shortest paths that end up being discarded when we call it again with another source vertex. It would probably be much better to design a dynamic programming algorithm directly for the APSP problem so that we don't get such waste.

And it turns out that we don't need to look very far to find a good option for designing a dynamic programming algorithm for APSP: let's use the same subproblems! For every pair $u, v \in V$ of vertices, define

$$d_i(u, v) = \text{shortest path from } u \text{ to } v \text{ that goes through} \leq i \text{ edges.}$$

Then we can apply the same base case and recurrence relation as we did in the Bellman–Ford algorithm, but update distances for all pairs of vertices, resulting in the following algorithm.

Algorithm 46: BellmanFordAPSP$(G = (V, E), w)$

```
for all u, v ∈ V × V do
    if (u, v) ∈ E then
        d[u, v] ← w(u, v);
    else
        d[u, v] ← ∞;
for i = 2, 3, ..., n − 1 do
    for all u ∈ V do
        for all (x, v) ∈ E do
            if d[u, x] + w(x, v) < d[u, v] then
                d[u, v] ← d[u, x] + w(x, v);
return d;
```

This algorithm has time complexity $O(n^2m)\ldots$ exactly the same as the SIMPLEAPSP algorithm! Conclusion: To obtain a more efficient algorithm with the dynamic programming technique, we need to find other subproblems to solve.

3. Floyd–Warshall algorithm

Let's go back to the problem we're trying to solve. We want to compute

$$\text{dist}[u, v] = \text{minimum weight of any path from } u \text{ to } v.$$

To design the Bellman–Ford algorithm, we noted that the problem of computing the distance from u to v is much easier if we restrict the set of paths that we consider. This observation led us to consider only paths that go through a small number of edges. But that's not the only way to restrict our attention to some of the paths only. In particular, instead of limiting the number of edges that the paths can use, we can limit the *set of intermediate nodes* that those paths can traverse. Let us label the vertices in V as $v_1, v_2, \ldots, v_n$. Then for each $0 \leq k \leq n$, we can define

$$\begin{aligned}\text{dist}_k[u, v] =& \text{minimum weight of any path from } u \text{ to } v \text{ that}\\ &\text{only uses } v_1, \ldots, v_k \text{ as intermediate nodes.}\end{aligned}$$

The base case of these subproblems is the same as before: when $k = 0$, the only possible path from u to v is the one that traverses the edge (u, v), if it exists. So

$$\text{dist}_0[u, v] = \begin{cases} w(u, v) & \text{if } (u, v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Now to compute $\text{dist}_k[u, v]$, we observe that there are two possibilities: either the shortest path from u to v that only uses $v_1, \ldots, v_k$ as intermediate nodes goes through v_k, or it does not. So

$$\text{dist}_k[u, v] = \min \begin{cases} \text{dist}_{k-1}[u, v_k] + \text{dist}_{k-1}[v_k, v] \\ \text{dist}_{k-1}[u, v]. \end{cases}$$

We combine those observations to obtain the *Floyd–Warshall algorithm*. In the following algorithm, let $V = \{v_1, v_2, \ldots, v_n\}$.

Algorithm 47: FLOYDWARSHALL$(G = (V, E), w)$

```
for all u, v ∈ V × V do
    if (u, v) ∈ E then
        dist_0[u, v] ← w(u, v);
    else
        dist_0[u, v] ← ∞;
for k = 1, 2, 3, ..., n do
    for all u ∈ V do
        for all v ∈ V do
            dist_k[u, v] ← min{dist_{k−1}[u, v_k] + dist_{k−1}[v_k, v], dist_{k−1}[u, v]};
return dist_n;
```

The time complexity of this algorithm is $\Theta(n^3)$.

Part 5

NP-completeness and hard problems

LECTURE 17

Exhaustive search

1. Systematic search

Let's say you are given the task to solve a comptuational problem where none of the design techniques we have covered in the class work and—even worse—no-one knows how to design an efficient algorithm for the problem. What do you do then? There are a few options.

- Design a *heuristic* argument that runs quickly but has no provable guarantee of correctness.
- Design an *approximation algorithm* that does not solve the problem optimally but at least gives some guarantees about its solution.
- Design an *exponential-time* exact algorithm for the problem.

Today's lecture aims to explore the last approach: it's never ideal to have an algorithm with time complexity that grows exponentially with the size of the input, but in some cases that's the best option. (If the input data is reasonably bounded, and/or if you have massive amounts of computation available—and if there really aren't any more efficient algorithms out there for the task.) Our goal is to see some techniques that will enable us to design these exponential-time algorithms in a way that guarantees they will be correct, and hopefully keep them as efficient as possible.

2. Backtracking

Consider the following generalization of the 3Sum problem we saw earlier in the course.

Definition 17.1. In the *Subset sum* problem, we are given as input an array of n elements $1, 2, \ldots, n$ with positive weights $w_1, \ldots, w_n$, along with a target weight T. We must determine whether there is a subset $S \subseteq \{1, 2, \ldots, n\}$ of elements with total weight $\sum_{i \in S} w_i = T$ or not.

There is no known polynomial-time algorithm for this problem. In fact, as we will see in the next part of the course, the Subset Sum problem is NP-complete, which means that it is quite possible that no such algorithm exists. So what can we do? We can explore the

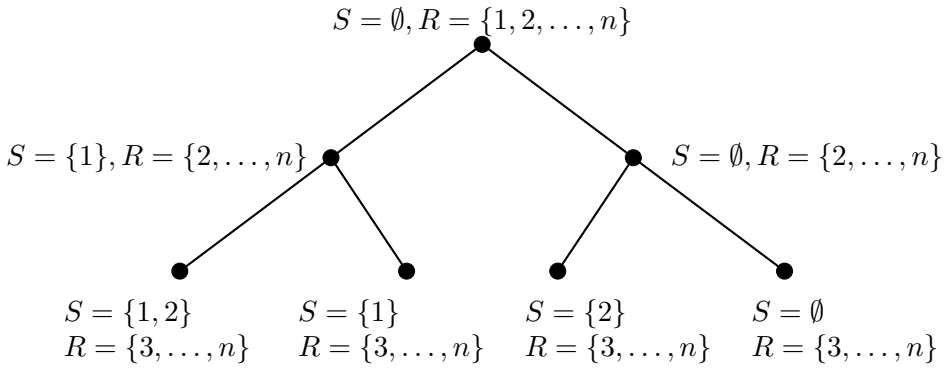
space of all possible sets S, one by one in some fixed order.

$$\begin{aligned} S &= \emptyset \\ S &= \{1\} \\ S &= \{2\} \\ S &= \{1,2\} \\ S &= \{3\} \\ &\vdots \\ S &= \{1,2,\ldots,n\} \end{aligned}$$

Running this algorithm on some examples, however, we might notice some obvious inefficiencies. For example, consider an input where $w_1 > T$. We discover then that with $S = \{1\}$, we already have $\sum_{i \in S} w_i = w_1 > T$, so there's really no point in checking $S = \{1,2\}$, $S = \{1,3\}$, $S = \{1,2,3\}$, etc.

With the *backtracking* technique, we can avoid checking all unnecessary cases while still making sure that we don't miss any potential solutions. The idea is that we can consider the set of all possible *configurations*—corresponding to full or partial solutions—in a tree structure.

For the Subset Sum problem, a configuration corresponds to a set $S \subseteq \{1,2,\ldots,k\}$ of elements that we have decided to add to our solution, and a set $R \subseteq \{k+1,\ldots,n\}$ of elements that we might still add to the set. Then the top of our tree of configurations looks like this:



Every node at level k has two children: one corresponding to the case where we add element $k+1$ to S, and one where we don't. (In both cases, we remove $k+1$ from R.) We never build the tree explicitly, but we design the algorithm to explore this tree of configurations with the following rules at a given node with current sets S and R:

- If $\sum_{i \in S} w_i = T$, SUCCESS! We have solved the problem and can return True;
- If $\sum_{i \in S} w_i > T$, DEAD END. We stop exploring this branch since all nodes underneath the current one will overshoot the target, so we *backtrack* to the parent node directly and continue exploring other paths from there.
- If $\sum_{i \in S \cup R} w_i < T$, DEAD END. We again backtrack since all nodes underneath the current one will undershoot the target.

We can implement this easily in an algorithm for the Subset Sum problem. Or, better yet, we can use the general backtracking algorithm and implement only a few helper functions instead.

Algorithm 48: BACKTRACKING

```
Initialize the set A of active configurations;
while A ≠ ∅ do
    C ← next configuration of A;
    if C is a solution return True;
    if C is not a dead end then
        EXPAND(C) and add the resulting configurations to A;
return False;
```

Note that there are two ways that we can implement the BACKTRACKING algorithm: by keeping the set of active configurations in $\mathcal{A}$ in a queue or in a stack. The two options correspond to designing an algorithm that explores the configuration tree using BFS or DFS—which is best depends on the width and the height of the configuration tree. For the Subset sum problem, the best option if we want to optimize the set complexity of the algorithm is DFS, as it will require keeping only $O(n)$ active configurations in memory at any time, versus up to $\Omega(2^n)$ active configurations if we use BFS instead.

2.1. Time complexity analysis. What is the time complexity of this algorithm? It is still $\Theta(2^n)$! This is a good exercise to try: can you come up with concrete examples where the backtracking algorithm explores $\Omega(2^n)$ nodes of the configuration tree before returning? In the *worst-case* time complexity setting, these hard examples show that we haven't gained anything over the simple exhaustive search, but in practice there are many situations where the gains obtained by the backtracking algorithm over naïve search are quite significant.

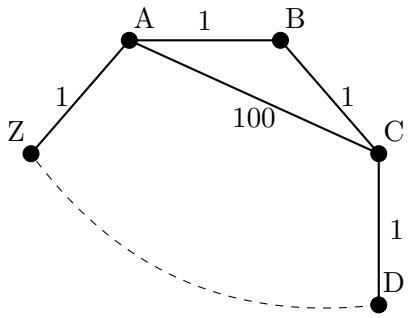
3. Branch-and-bound

There is another similar algorithm technique when we consider optimization problems instead of decision problems. Consider for example the famous Travelling Salesman problem.

DEFINITION 17.2. In the *Travelling Salesman problem (TSP)*, we are given a weighted undirected graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^{\geq 0}$. Our goal is to find a cycle C that goes through each vertex in V exactly once (called a TSP tour) with minimum weight $\sum_{e \in C} w(e)$.

The Travelling Salesman Problem is also NP-complete. We don't have any polynomial-time algorithm for the problem—and in fact the optimal tour that visits all $< 2M$ cities on Earth is still not known (see `http://www.math.uwaterloo.ca/tsp/world/index.html` for all the details and instructions on how to submit your improved tour when you beat the world record!).

We can again solve this problem by computing the lengths of all $|V|!$ possible tours of the graph, but again this approach feels wasteful. Consider the following graph:



If we consider the tour $A \to B \to \cdots \to Z$ first, we obtain a tour with total weight 26. This means that *any* tour which uses the edge $A \to C$ will not be optimal, since it will have weight strictly greater than 26. To avoid multiple unnecessary checks, we can use the *branch and bound* technique. At each step, we

- Consider a set A of possible configurations;
- *Branch* by splitting A into subsets $A_1, \ldots, A_t$ of possible configurations; and
- *Bound* the minimum value of any solution in configuration A_i; we discard A_i if this value is greater than the optimal solution discovered so far.

The general branch-and-bound algorithm looks very similar to the BACKTRACKING algorithm.

Algorithm 49: BRANCHANDBOUND

Initialize the set $\mathcal{A}$ of active configurations;
Initialize `best` $\leftarrow \infty$;
while $\mathcal{A} \neq \emptyset$ **do**
  $C \leftarrow$ next configuration of $\mathcal{A}$;
  if C *is a solution and* VALUE(C) $<$ `best` **then**
    `best` $\leftarrow$ VALUE(C);
  else if C *is not a dead end and* VALUELOWERBOUND(C) $<$ `best` **then**
    EXPAND(C) and add the resulting configurations to $\mathcal{A}$;
return `best`;

In this algorithm,

- VALUE(C) is the value of the current configuration when it represents a (complete) solution;
- a *dead end* is a configuration C that cannot lead to a valid solution; and
- VALUELOWERBOUND(C) is a lower bound on the value of any (complete) solution that can be obtained by expanding configuration C.

Note that the VALUELOWERBOUND of a configuration C does not have to be (and will usually not be) a tight approximation to the actual minimum value of any solution obtained by expanding C, but the more accurate this estimate can be, the fewer branches the algorithm has to explore.

For the travelling salesman problem, a configuration C consists of

- a set $I \subseteq E$ of edges that are *included* in the tour, and
- a set $X \subseteq E$ of edges that are *excluded* in the tour.

We maintain $I \cap X = \emptyset$ throughout the execution of the branch-and-bound algorithm. Initially, $I = X = \emptyset$. We *branch* by considering any edge $e \in E \setminus (I \cup X)$ and constructing the two configurations $C_1 = (I \cup \{e\}, X)$ and $C_2 = (I, X \cup \{e\})$.

A configuration $C = (I, X)$ forms a solution to the problem when I forms a TSP tour. This solution is the best so far when the corresponding tour has smaller weight than the best tour found so far; when this happens, we update the minimum weight and the best tour found so far to I.

Otherwise, we *bound* our search by first determining whether the configuration $C = (I, X)$ is a dead end. We do this by checking whether

- Every vertex has degree at most 2 in I;
- There is no cycle in I;
- The graph $G' = (V, E \setminus X)$ is 2-connected.

If any of the condition is not satisfied, then the current configuration cannot lead to a TSP tour so we discard it.

When the current configuration $C = (I, X)$ is not a complete TSP tour or a dead end, we then do one last test before we expand it to make sure it might lead to a new best tour: we check whether the total weight $\sum_{e \in I} w(e)$ of the edges already included in I is not larger than the weight of the best tour found so far. When this condition is satisfied, we branch.

There are a number of optimizations to this algorithm that we can implement: we can get a stronger lower bound on the cost of any tour obtained from some configuration, we can use clever branching strategies to explore more promising configurations first, etc. None of those enhancements lead to a polynomial-time algorithm, but they do yield algorithms that perform reasonably well on instances of TSP encountered in practice.

LECTURE 18

Polynomial-time algorithms

1. Efficient algorithms and tractability

What is an *efficient* algorithm? That depends on your situation. In various settings, a requirement for an algorithm to be efficient might be that the algorithm

- Has time complexity $O(n)$;
- Runs in $O(n)$ time *and* only needs to scan the input once (streaming algorithm);
- Can be run in $o(n)$ time using distributed algorithms (e.g., MapReduce);
- Runs in $o(n)$ time by sampling the input;
- Has time complexity $O(n \log n)$;
- Has time complexity $O(n^2)$;
- Has time complexity $O(n^3)$ *and* space complexity $O(\log n)$;
- etc., etc., etc.

For this class, we won't explore any of those definitions. Instead, we will take a much more lenient notion of efficiency: we'll say that an algorithm is (reasonably) efficient if it is a *polynomial-time algorithm*.

Definition 18.1. An algorithm A is a *polynomial-time algorithm* if there exists a constant k such that on any input of size n, the algorithm A runs in time $O(n^k)$.

Remark 18.2. For this definition, we measure the size of the input in terms of the number of bits required to encode the input (as opposed to, say, the number of entries in a matrix).

And then we get to the main question that we will study in this part of the course:

> Which computational problems can be solved by polynomial-time algorithms?

We will call such problems *tractable*. Most of the problems we have covered in this class—and indeed most of the problems you see in all CS courses combined!—are tractable. The only exceptions in this class are the problems of the last lecture and the problems that can be solved with *pseudo-polynomial-time algorithms* we saw in the dynamic programming section, but it turns out that there are a *ton* of seemingly reasonable problems that, to the best of our knowledge today, are intractable. In this section, our task is to learn how to recognize these problems.

1.1. Why polynomial time? But before we continue, there's one rather puzzling question that we should examine right away: *why* polynomial-time algorithms? After all, we listed many other perfectly reasonable possible definitions of efficient algorithms, and you may even argue that one (or any!) of them would be much better at capturing the idea of efficiency or tractability, so why didn't we pick one of them instead? I can think of three reasons.

(1) **Robustness.** We believe that the class of problems that can be solved by polynomial-time algorithms is *independent* of the way we define algorithms. This belief is known as the *strong Church-Turing thesis*, and this is something that would *not* be true if we defined tractability with any of the other candidates we listed earlier. This means that if you show that a problem is intractable and are faced with the criticism that your algorithm model is not the right one, you can reply that it doesn't matter! Any reasonable modification to the model won't affect the tractability or intractability of the problem. (See CS 360/365 for all the details.)
(2) **Strength of the conclusion.** The main point of this section is that we will want to be able to argue that some problems are *not* tractable. It turns out that, in this case, it is good to have a very lenient notion of tractability, because by showing that a problem has no polynomial-time algorithm, we will also imply at the same time that the problem can't be solved by an efficient algorithm under any of the other alternative definitions of efficiency above either! (And many others as well.) So instead of ruling out algorithms in one setting after another in a long and tedious process, we do so all at once.
(3) **Applicability.** All of the discussion above is very nice, but wouldn't be of interest if most problems could be solved by polynomial-time algorithms. But as we're about to see, that's far from the case. In fact, I would be extremely surprised if most of you don't end up running up against intractable problems multiple times throughout your careers. So knowing how to show that a problem is intractable is not just something that's of purely intellectual interest; it is something that will be useful in practice.

2. The class P

We saw in the last lecture two types of computational problems:

Decision problems: : problems where the possible outputs are `True` or `False` (or, equivalently, 1 or 0; `Yes` or `No`; etc.); and

Optimization problems: : problems where we are trying to find the value of the *best* solution.

There is also a third large class of problems that we can call

Search problems: : problems where the goal is to *find* a valid (or the best) solution itself.

We will now focus exclusively on *decision problems*. This makes everything we do from now on much easier, and it's still useful even if the problem you care about is an optimization or a search problem because we can often obtain analogous decision problems very easily. For example, we can define:

MULT: : Given two n-bit positive integers x and y and a coordinate $i \in \{1, 2, \ldots, 2n\}$, determine if the ith bit of the product xy is 1.

INTSCHED: : Given a set S of n intervals $(s_1, f_1), \ldots, (s_n, f_n)$ and a positive integer k, determine if there is a set of k non-overlapping intervals in S.

LCS: : Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ and a positive integer k, determine if the longest common subsequence of x and y has length at least k.

The class of tractable decision problems is of such fundamental importance that we give it a name.

DEFINITION 18.3. The class **P** is the set of all decision problems that can be solved by polynomial-time algorithms.

We have already seen that the problems 3SUM, INTSCHED, LCS, CONNECTIVITY,... are all tractable by designing polynomial-time algorithm for them. In mathematical notation, we write this as

$$\begin{aligned} \text{3SUM} &\in \mathbf{P} \\ \text{INTSCHED} &\in \mathbf{P} \\ \text{LCS} &\in \mathbf{P} \\ \text{CONNECTIVITY} &\in \mathbf{P}. \end{aligned}$$

3. Reductions

How can we show that a new problem is in **P**? We can design a polynomial-time algorithm that solves the problem. Or we can use the idea of *reduction*.

DEFINITION 18.4. The decision problem A is *polynomial-time reducible* to the decision problem B, written

$$A \leq_{\mathbf{P}} B,$$

if there is a polynomial-time algorithm F that, on any input I_A to the problem A, produces an input I_B to the problem B that has the same answer.

In other words, A is polynomial-time reducible to B if there is a polynomial-transformation of the inputs of A to those of B that maps all `Yes` inputs of A to the `Yes` inputs of B and all the `No` inputs of A to the `No` inputs of B. You can see a picture of this view of polynomial-time reductions in the class textbook.

You have already seen one example of a polynomial-time reduction in a tutorial earlier this term.

DEFINITION 18.5. In the LIS (*longest increasing subsequence*), you are given a sequence $x_1, \ldots, x_n$ of n positive integers in the range $\{1, 2, \ldots, \ell\}$ and a positive integer k, and you must determine if the longest increasing subsequence of x has length at least k.

THEOREM 18.6. LIS $\leq_{\mathbf{P}}$ LCS.

PROOF. Given an input to the LIS problem, let F be the algorithm that sorts x to obtain the sequence y and returns the sequences x and y and the integer k. The algorithm F runs in polynomial-time and its result corresponds to an input of the LCS problem which has a longest common subsequence of length at least k if and only if the original sequence x had a longest increasing subsequence of length at least k. □

Now here is an observation that will prove to be extremely powerful: we can show polynomial-time reductions between A and B *even when we don't know of any polynomial-time algorithm for either problem*! For example, take the following two problems that we have not yet considered.

- CLIQUE: Given a graph $G = (V, E)$ and a positive integer k, determine if G contains a clique of size k;
- INDEPSET: Given a graph $G = (V, E)$ and a positive integer k, determine if G contains an independent set of size k.

We have not yet seen any polynomial-time algorithms for either of these problems, but we can still show polynomial-time reductions between the two problems.

THEOREM 18.7. CLIQUE $\leq_{\mathbf{P}}$ INDEPSET *and* CLIQUE $\leq_{\mathbf{P}}$ INDEPSET.

PROOF. To show that CLIQUE $\leq_{\mathbf{P}}$ INDEPSET, let F be the algorithm that takes in a graph $G = (V, E)$ and generates the complement graph $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{(u, v) \notin E\}$. The construction takes polynomial time, and the resulting graph $\overline{G}$ has an independent set of size k if and only if the original graph has a clique of size k.

To show that INDEPSET $\leq_{\mathbf{P}}$ CLIQUE, we also use the same algorithm F, and we observe that every independent set of G becomes a clique in $\overline{G}$. □

What does this mean? The definition of polynomial-time reductions and the result CLIQUE $\leq_{\mathbf{P}}$ INDEPSET means that if someone designs a polynomial-time algorithm for INDEPSET, then there is also a polynomial-time algorithm for CLIQUE. This also means that if someone shows that there is *no* polynomial-time algorithm for CLIQUE, then there can be no polynomial-time algorithm that solves INDEPSET either. More generally, our interest in polynomial-time reductions stems from the following main lemma.

LEMMA 18.8. *If A and B are two decision problems that satisfy $A \leq_{\mathbf{P}} B$, then*

- *If $B \in \mathbf{P}$ then also $A \in \mathbf{P}$; and*
- *If $A \notin \mathbf{P}$ then also $B \notin \mathbf{P}$.*

For most of the hard problems that we care about, we do not know how to show that they are not in **P**. Starting in the next lecture, we will use the theory of **NP**-completeness to do the next-best thing: show that if *one* of those hard problems is tractable, then they *all* are.

LECTURE 19

Polynomial-time reductions

Recall that **P** is the set of all decision problems that can be solved by polynomial-time algorithms, and that a *polynomial-time reduction* from the decision problem A to B, written $A \leq_{\mathbf{P}} B$, exists when there is a polynomial-time algorithm F that transforms inputs I_A to A into inputs I_B to B that have the same answer. In today's lecture, we continue our exploration of polynomial-time reductions and of how they might be used to show that some algorithms are (probably) not in **P**.

1. More polynomial-time reductions

Let's consider the following problems on graphs:

Clique: Given the graph G and a positive integer k, determine if G has a clique of size at least k.

IndepSet: Given the graph G and a positive integer k, determine if G has an independent set of size at least k.

VertexCover: Given the graph G and a positive integer k, determine if G has a vertex cover (=a set S of vertices that "cover" all the edges $(u, v) \in E$ in the sense that at least one of u or v is in S) of size at most k.

NonEmpty: Given the graph G, determine if it has at least one edge.

HamCycle: Given the graph G on at least 3 vertices, determine if it has a *Hamiltonian cycle*—a cycle that visits each vertex in G exactly once.

HamPath: Given the graph G on at least 2 vertices, determine if it has a *Hamiltonian path*—a path that visits each vertex in G exactly once.

We can prove a number of polynomial-time reductions between these problems. In the first example, we see that the transformation in a reduction is sometimes very simple.

LEMMA 19.1. NONEMPTY $\leq_{\mathbf{P}}$ CLIQUE.

PROOF. Consider the polynomial-time algorithm F that transforms the input graph G into the pair $(G, 2)$ that includes the same graph and the positive integer 2.

If G is nonempty, it contains at least one edge $(u, v) \in E$. Then the set $\{u, v\}$ is a clique of size 2 in the graph, so every `Yes` instance of NONEMPTY is mapped to a `Yes` instance of CLIQUE.

To prove that every `No` instance of NONEMPTY is mapped to a `No` instance of CLIQUE, we establish the contrapositive statement: that every `Yes` instance of CLIQUE is obtained from a `Yes` instance of NONEMPTY. This is straightforward: if G contains a clique $\{u, v\}$ of size 2, then $(u, v) \in E$ must be an edge so G is nonempty. □

We can also obtain non-trivial reductions without modifying the graph itself.

LEMMA 19.2. INDEPSET $\leq_{\mathbf{P}}$ VERTEXCOVER.

PROOF. Consider the polynomial-time algorithm F that transforms the input (G, k) to IndepSet into the input $(G, n-k)$ to VertexCover.

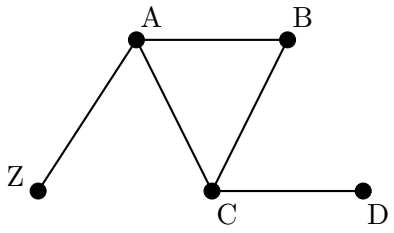
If G contains an independent set $I \subseteq V$ of size k, then the set $S = V \setminus I$ is a vertex cover of G of size $n-k$. (It is a vertex cover since no edge can have both endpoints in I since it is an independent set.)

And if G contains a vertex cover $S \subseteq V$ of size n, then the set $I = V \setminus S$ is a set of size $n-(n-k) = k$ that must be an independent set, since each edge in the graph G has at least one endpoint in S. □

We saw in the last lecture that some polynomial-time reductions, like Clique $\leq_{\mathbf{P}}$ IndepSet, are obtained by modifying the input graph itself. Here's another similar example.

LEMMA 19.3. HamPath $\leq_{\mathbf{P}}$ HamCycle.

Before we prove the lemma, it's worth pausing to see that the two problems are not equivalent: there are graphs that have a Hamiltonian path but no Hamiltonian cycle. This is one example of such a graph.



PROOF OF LEMMA 19.3. Let F be the algorithm that takes the input $G = (V, E)$ and creates the graph $G' = (V', E')$ where the set of vertices of the new graph

$$V' = V \cup \{s\}$$

is obtained by adding a new vertex s and the set of edges

$$E' = E \cup \{(s, v) : v \in V\}$$

is the original set of edges along with an edge between every vertex in V and the new vertex s. This transformation is easily computed in polynomial-time, and to complete the reduction we need to show that G has a Hamiltonian path if and only if G' has a Hamiltonian cycle.

- If G has a Hamiltonian path P with endpoints $a, b \in V$, then the cycle obtained by adding the edges (s, a) and (b, s) to the beginning and the end of the path P, respectively, forms a Hamiltonian cycle in G'.
- If G' has a Hamiltonian cycle C, that cycle contains two edges that we can call (s, a) and (s, b) that are incident to s. Removing those two edges leaves us with a Hamiltonian path in G.

□

All of these examples provide reductions between different problems on graphs, but this does not have to be the case: we can have reductions between very different types of problems as well. Consider for example the SetCover problem.

SetCover: : Given a collection $\mathcal{S}$ of subsets of $\{1, 2, \ldots, m\}$ and a positive integer k, determine if there are k sets $S_1, \ldots, S_k \in \mathcal{S}$ such that $S_1 \cup \cdots \cup S_k = \{1, 2, \ldots, m\}$.

LEMMA 19.4. VERTEXCOVER $\leq_{\mathbf{P}}$ SETCOVER.

PROOF. Let F be the algorithm that transforms a graph $G = (V, E)$ into a collection of subsets over $\{1, 2, \ldots, m\}$ with $m = |E|$. The transformation is done by labeling each edge in E with the numbers $1, \ldots, m$. Then for each vertex $v \in V$ we create the set

$$S_v = \{i \leq m : v \text{ is incident to edge } i \text{ in } G\}.$$

Let $\mathcal{S} = \{S_v\}_{v \in V}$. The result of this transformation is $(\mathcal{S}, k)$. The transformation can be completed in polynomial time. To verify that it is a reduction from VERTEXCOVER to SETCOVER, we need to verify that G has a vertex cover of size k if and only if $\mathcal{S}$ has a set cover of size at most k.

- If G has a vertex cover C of size k, then the family of sets S_v for each $v \in C$ satisfy $\bigcup_{v \in C} S_v = \{1, 2, \ldots, m\}$ since each edge is adjacent to at least one of the vertices in C.
- If $\mathcal{S}$ has a set cover $S_{v_1}, \ldots, S_{v_k}$ of size k, then the set $C = \{v_1, \ldots, v_k\}$ is a vertex cover for G since the edge i is adjacent to the vertex v_j corresponding to the set S_{v_j} that contained i.

□

2. Facts about polynomial-time reductions

Having now seen quite a few polynomial-time reductions, we can take a step back and try to identify some of the important basic properties of these reductions. The first one is that polynomial-time reductions are *transitive* operations on decision problems.

THEOREM 19.5. *If A, B, and C are decision problems that satisfy $A \leq_{\mathbf{P}} B$ and $B \leq_{\mathbf{P}} C$, then $A \leq_{\mathbf{P}} C$.*

PROOF. Let F_1 be the polynomial-time algorithm that transforms inputs for A into inputs for B, and let F_2 be the polynomial-time algorithm that transforms inputs for B into inputs for C. Let F be the algorithm that runs F_1 (on the input to A) and then F_2 (on the result of F_1). This algorithm also runs in polynomial time, and transforms inputs for A into inputs for C that satisfy the conditions of polynomial-time reductions. □

This lets us obtain new reductions by combining the ones we have already established.

LEMMA 19.6. CLIQUE $\leq_{\mathbf{P}}$ SETCOVER.

PROOF. This result follows immediately from the transitivity of polynomial-time reductions and the fact that CLIQUE $\leq_{\mathbf{P}}$ INDEPSET (we proved this last lecture), that INDEPSET $\leq_{\mathbf{P}}$ VERTEXCOVER, and that VERTEXCOVER $\leq_{\mathbf{P}}$ SETCOVER. □

Another important property of polynomial-time reductions is that it is *not* symmetric.

THEOREM 19.7. *There are some decision problems A and B such that $A \leq_{\mathbf{P}} B$ but $B \not\leq_{\mathbf{P}} A$.*

PROOF IDEA. We saw that NONEMPTY $\leq_{\mathbf{P}}$ CLIQUE but we have strong reasons to believe (as we will explore in the next few lectures) that there is no polynomial-time reduction from CLIQUE to NONEMPTY.[1] □

[1] You should give it a try!

This is a point worth emphasizing: the direction that you do a polynomial-time reduction really matters! And the way we will be doing these reductions will from now on usually be in the same pattern: we will reduce a known hard problem to a new problem, so that we show that the new problem is hard as well.

THEOREM 19.8. *If* HARD *is a decision problem that satisfies* HARD $\notin \mathbf{P}$ *and* B *is a decision problem that satisfies* HARD $\leq_{\mathbf{P}} B$*, then* $B \notin \mathbf{P}$*.*

PROOF. Assume for contradiction that $B \in \mathbf{P}$. Then there is a polynomial-time algorithm F that transforms inputs for HARD into inputs for B (which preserves the `Yes` and `No` answers) and there is a polynomial-time algorithm $\mathcal{A}_B$ that solves B, so we obtain a polynomial-time algorithm that solves HARD by running F and then $\mathcal{A}_B$ and outputing the result; this contradicts the fact that HARD $\notin \mathbf{P}$. □

REMARK 19.9. But note that if HARD $\notin \mathbf{P}$ and we show $A \leq_{\mathbf{P}}$ HARD, this says nothing about whether A is easy or hard!

3. The class NP

Now that we have a good handle on polynomial-time reductions, we are ready to put them to good use and prove that lots of basic problems are intractable... except that to do so we first need to begin with at least *one* problem that we know is hard.

3.1. Verifiers. But before we identify our hard problem, it's better to pause for a second and examine the problems CLIQUE, SUBSET SUM, VERTEX COVER, etc. that we would very much like to solve efficiently but can't currently solve with any polynomial-time algorithm. Do these problems have anything in common?

As it turns out, all these problems share one very interesting property: while they are all hard to solve (we believe), they are all easy to *verify*: if I claim that a graph has a clique of size k, I can convince you that this is true by identifying the k vertices that I claim form a clique. You can then check that those vertices form a clique in polynomial time with a very simple algorithm. Similarly, if I claim there is a subset of the elements that sum up to exactly the target value, I can identify the subset and you can verify that the sum is indeed the target in polynomial time. And you can verify that we can do this for every decision problem we have covered in the last two lectures.

Let's introduce some definitions to make this discussion more precise.

DEFINITION 19.10. An algorithm A *verifies* (or *is a verifier for*) the decision problem P if

- It takes in 2 inputs, x and y, and outputs `Yes` or `No`;
- For every input x to problem P, x is a `Yes` input for P if and only if there exists a *certificate* y such that $A(x, y)$ outputs `Yes`.

So in words, a verifier is an algorithm that receives some additional "help" input y that is meant to help the algorithm verify that an input x is indeed a `Yes` input—but the algorithm must never be fooled into outputing `Yes` by *any* possible certificate when x is a `No` input.

3.2. Efficient verifiers. The idea of an algorithm being an *efficient* verifier for a problem can now be formalized in the same way that we associated efficient algorithms with polynomial-time algorithms.

DEFINITION 19.11. Algorithm A is a *polynomial-time verifier* for the decision problem P if it is a polynomial-time algorithm that verifies P *and* for any `Yes` input x, there is a certificate y for x of length polynomial in the length of x.

REMARK 19.12. Why the second condition (after the *and*) in the definition? Remember that a polynomial-time algorithm is an algorithm whose runtime is polynomial in the size of its *input*—which for a verifier is the size of x and y; we need the condition on the length of y to guarantee that the runtime of the verifier will also be polynomial in the length of x by itself.

We now are ready to define another fundamental class of problems: **NP**.

DEFINITION 19.13. The class **NP** is the set of all decision problems that have polynomial-time verifiers.

The famous **P** vs. **NP** problem can thus be restated as follows.

> *Can every problem that has a polynomial-time verifier also be solved with a polynomial-time algorithm?*

Answer this question—and provide a valid proof that justifies your answer!—and you will have solved one of the most significant open problems in all of computer science and mathematics today. But this is bad news for us: it means that until the **P** vs. **NP** problem has been resolved, we will not be able to show that CLIQUE $\notin$ **P**, SUBSETSUM $\notin$ **P**, etc.—since all these problems are in **NP** and it is possible that **P** = **NP**, then it's possible that all these problems are in **P** as well. But we will be able to show that if **P** $\neq$ **NP**, then CLIQUE $\notin$ **P** (and similarly for many other problems as well).

LECTURE 20

NP-completeness

In this lecture, we introduce the notion of **NP**-complete problems and discuss why we believe that these problems cannot be solved efficiently.

1. NP-Completeness

In the last lecture, we saw that some decision problems that appear to be very hard to solve—namely, Clique and SubsetSum—can be efficiently *verified*: these problems are in the class **NP**. Ideally, we would like to justify the fact that we can't find algorithms that solve these problems in polynomial time with a proof that these problems are not in **P**. However, since the problem of determining whether $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$ is (notoriously) open, we can't hope to show that *any* problem in **NP** is not in **P** at the moment. Instead, we can do the next best thing by showing that Clique, SubsetSum, and other similar problems are the "hardest" problems in **NP** in a precise way using the notion of **NP**-*completeness*.

DEFINITION 20.1. The decision problem X is **NP**-*complete* if

- $X \in \mathbf{NP}$, and
- For every problem $A \in \mathbf{NP}$, $A \leq_{\mathbf{P}} X$.

Proving that a decision problem X is **NP**-complete has some significant immediate implications.

THEOREM 20.2. *Let X be any* **NP**-*complete problem. Then:*

- *If $X \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$ and so* every *problem in* **NP** *can be solved with a polynomial-time algorithm; and*
- *If $X \notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$ and* no **NP**-*complete problem can be solved with a polynomial-time algorithm.*

As a result, proving that a problem is **NP**-complete is a very strong indication that a problem cannot be solved efficiently: the only way that the problem can be solved with a polynomial-time algorithm is if every other **NP**-complete problem can also be solved that efficiently—and computer scientists have spent countless hours trying to find efficient algorithms for many of those problems, without any success so far!

2. 3SAT is NP-complete

When we are working with a specific decision problem X, proving that every $A \in \mathbf{NP}$ satisfies $A \leq_{\mathbf{P}} X$ is typically a rather arduous task. Thankfully, we don't have to do all this work every time; because of the transitivity of polynomial-time reductions, all we have to do is to show that *one* **NP**-complete problem A satisfies $A \leq_{\mathbf{P}} X$.

THEOREM 20.3. *Let A be an* **NP**-*complete problem and let $X \in \mathbf{NP}$ be a decision problem such that $A \leq_{\mathbf{P}} X$. Then X is* **NP**-*complete.*

This is the approach we will take to prove that the problems we have been considering since the beginning of last week are **NP**-complete. But to do this, we must first identify *one* (easy-to-work-with) **NP**-complete problem. Cook (1971) and Levin (1973) were the first to do so, by showing explicitly that every problem in **NP** has a polynomial-time reduction to *satisfiability* problems. The full proof of this theorem is one of the gems of CS 360/365. For reductions, the variant on the satisfiability problems that is most useful is known as 3SAT.

DEFINITION 20.4. In the 3SAT *problem*, we are given a Boolean CNF formula (an OR of ANDs) on n variables in which each clause has at most 3 literals; we must determine whether there is an assignment of `True` or `False` values to the n variables that makes the formula true (i.e., that *satisfies* the formula).

For example, the formula

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5)$$

is an input to the 3SAT problem for which the answer is `True`. (Set x_1 and x_4 to be `True`, assign any value to the remaining three variables.)

COOK–LEVIN THEOREM. 3SAT *is* **NP**-*complete.*

(As an exercise: can you prove that 3SAT $\in$ **NP**?)

3. More NP-complete problems

We can now use the **NP**-completeness of 3SAT to prove that CLIQUE is **NP**-complete as well.

THEOREM 20.5. CLIQUE *is* **NP**-*complete.*

PROOF. We already showed that CLIQUE $\in$ **NP**.

We want to show that 3SAT $\leq_{\mathbf{P}}$ CLIQUE. Let φ be a CNF formula with c clauses that each contain at most 3 literals. Define G to be the graph G where we create one vertex for each literal in each clause (for a total of at most $3c$ vertices) and we connect two vertices with an edge if

- the vertices represent literals in *different* clauses; and
- the literals represented by the vertices do not contradict each other (e.g., they do not represent x_i and $\neg x_i$, respectively).

This transformation is easily completed in polynomial time. And we have the following two observations.

(1) If G contains a clique $C \subseteq V$ of size c, the clique must contain one vertex that represents a literal in each of the c different clauses (since two vertices from the same clause are never connected by an edge), and none of those literals contradict each other. So we can assign values to the variables that make all those literals `True` to satisfy all the clauses in φ.
(2) If φ is satisfiable, take any satisfying assignment. This makes at least one literal in each clause evaluate to `True`. Consider the set S of vertices obtained by choosing any one `True` literal from each clause. This set S is a clique of size c since all its vertices correspond to literals in different clauses that don't contradict each other.

Therefore, the transformation we defined gives a polynomial-time reduction from 3SAT to CLIQUE. □

Using the polynomial-time reductions we established last week, we immediately obtain the **NP**-completeness of a number of other problems as well.

THEOREM 20.6. *The decision problems* INDEPSET*,* VERTEXCOVER*, and* SETCOVER *are all* **NP***-complete.*

PROOF. All three problems are in **NP**.

INDEPSET: The certificate is a set $S \subseteq V$ of k vertices. The verifier checks that no two vertices in S are connected by an edge.

VERTEXCOVER: The certificate is a set $S \subseteq V$ of k vertices. The verifier checks that each edge in the graph has at least one endpoint in S.

SETCOVER: The certificate is a set $I \subseteq \{1, 2, \ldots, n\}$ of k indices. The verifier checks that $\bigcup_{i \in I} S_i$ covers each element in the ground set $\{1, 2, \ldots, m\}$.

We already saw in Lecture 19 that

$$\text{CLIQUE} \leq_{\mathbf{P}} \text{INDEPSET} \leq_{\mathbf{P}} \text{VERTEXCOVER} \leq_{\mathbf{P}} \text{SETCOVER}.$$

and we just saw that CLIQUE is **NP**-complete, so the **NP**-completeness of the other three problems follows from Theorem 20.3. □

LECTURE 21

More NP-completeness

We finished the last lecture by showing that from Cook–Levin's theorem that 3SAT is **NP**complete, we can show that CLIQUE is also **NP**-complete. This result (along with the simple proofs that the following languates are in **NP**) implies that all of the decision problems

INDEPSET,
VERTEXCOVER, and
SETCOVER

are also **NP**-complete. Today, we continue our exploration of **NP**-completeness by adding more problems to this list.

1. HAMPATH is NP-complete

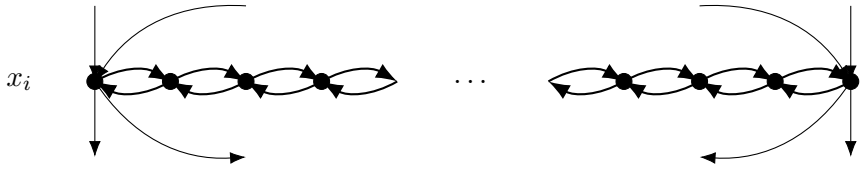
Let's show that HAMPATH is **NP**-complete. The easiest way to do so is by first showing that the analogous problem on directed graph is **NP**-complete. Define DIRHAMPATH to be the problem where we are given a directed graph $G = (V, E)$ and must determine whether it contains a Hamiltonian path (= a path in G visiting each vertex of V exactly once).

THEOREM 21.1. DIRHAMPATH *is* **NP***-complete.*

PROOF IDEA. DIRHAMPATH $\in$ **NP** since a verifier that demands the Hamiltonian path P through G as the certificate easily checks in time polynomial in the size of the graph whether P does visit each vertex in V exactly once and whether it is a valid path in G.

To show that DIRHAMPATH is **NP**-complete, we now want to show that 3SAT $\leq_{\mathbf{P}}$ DIRHAMPATH.

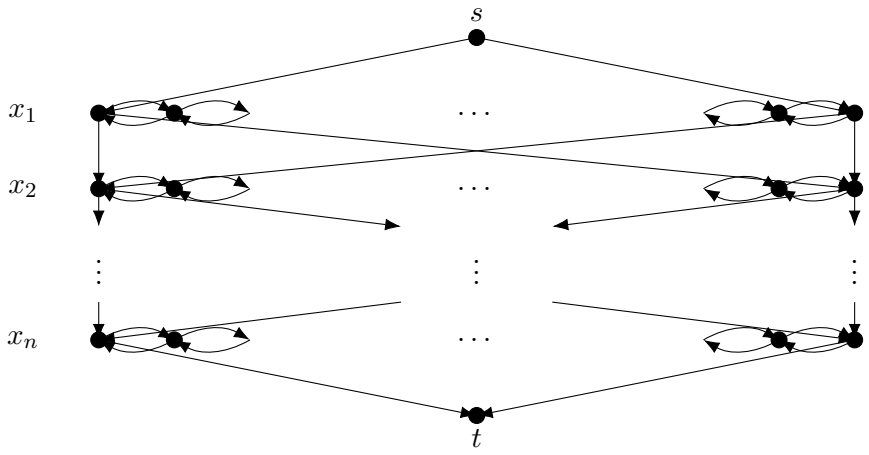
The main idea is that we will construct a *gadget* which is a graph that will correspond to the idea of assigning `True` or `False` values to each of the variables $x_1, \ldots, x_n$ in the original formula φ. The gadget is composed of a path of vertices for each variable x_i:



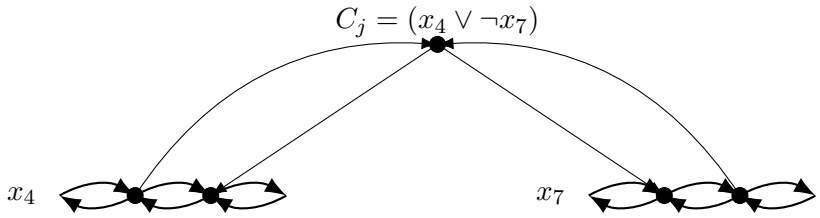
We have two choices in a Hamiltonian path: we either visit the vertices in that row left-to-right (which we will associate with setting of $x_i \leftarrow$ `True`) or right-to-left (which will correspond to $x_i \leftarrow$ `False`). The edges between each row mean that we can choose the assignment of `True` or `False` to each variable separately.

For each $i < n$, there are edges connecting the left- and right-most vertices of the path to x_i to the left- and right-most vertices of the path for x_{i+1}. We then complete the base gadget by adding a source vertex s with edges to the left- and right-most vertices of the

path for x_1 and a target vertex t with edges going from the left- and right-most vertices of the path for x_n to t.



Now the gadget by itself has a Hamiltonian path from s to t. We want to add one more vertex for each clause in φ and connect it to the gadget in a way that we can visit the vertex corresponding to a clause only when our tour through the main gadget corresponds to an assignment to the variables that satisfies the clause. We can do this by constructing one vertex per clause and attaching it to the paths of the variables in the clause with the direction of the connection determined by whether that variable is negated or not in the clause, as in the following example:



Then the last subtle point is that we need to make sure that we attach those constructions with at least one spare vertex between any connections. When we complete the construction, we see that every satisfiable formula φ results in a graph G that has a Hamiltonian path: simply traverse x_i in the direction corresponding to the value assigned to it, and visit each clause vertex during the traversal of the row corresponding to one of its satisfying literals.

In the other direction, we want to argue that if G has a Hamiltonian path, the original formula φ is satisfiable. The key idea to doing this is to show that the only way that the vertex corresponding to a clause can be visited is as a detour through the path through the row corresponding to one of its satisfied literals: that's because path must start at s and end at t; if (x, c) edge is followed and (c, y) edge is not, then any path that goes through b must come from s_2 and reaches a dead-end. Therefore, we can read the direction through each row to find a satisfying assignment for φ. □

We are now ready to show that HamPath (on undirected graphs) is **NP**-complete.

Theorem 21.2. HamPath *is* **NP**-*complete.*

PROOF. HAMPATH $\in$ **NP** by essentially the same argument that we used for DIRHAMPATH.

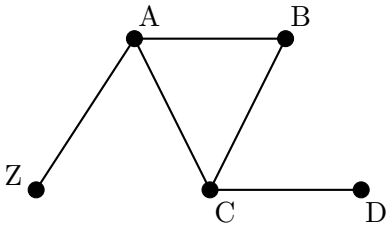
Let us now show that DIRHAMPATH $\leq_{\mathbf{P}}$ HAMPATH. Given the directed graph $G = (V, E)$, let us build a graph $G' = (V', E')$ where for each vertex $v \in V$, we now create three vertices in V': v itself, v_{in}, and v_{out}. We will build E' by adding edges (v_{in}, v) and (v, v_{out}) for each $v \in V$, and for each $(u, v) \in E$ by adding the edge (u_{out}, v_{in}) in E'.

With this construction, if there is a Hamiltonian path $v^{(0)}, v^{(1)}, \ldots, v^{(n)}$ in G, then the path $v_{in}^{(0)}, v^{(0)}, v_{out}^{(0)}, v_{in}^{(1)}, \ldots, v_{out}^{(n)}$ is a Hamiltonian path in G'.

And if we have a Hamiltonian path in G', then we must have a Hamiltonian path in G as well... though this is not completely obvious! (Exercise: why is this claim true?) □

2. HAMCYCLE is NP-complete

Recall that a Hamiltonian cycle is a cycle that visits each vertex in the graph exactly once. It's worth pausing to note that the problem of determining if a graph has a Hamiltonian cycle is not equivalent to the problem of determining if it has a Hamiltonian path: there are graphs that have a Hamiltonian path but no Hamiltonian cycle. This is one example of such a graph.



LEMMA 21.3. HAMCYCLE *is* **NP***-complete.*

PROOF. HAMCYCLE is in **NP**: let the certificate be a claimed Hamiltonian cycle. The verifier can check in polynomial-time that a certificate indeed corresponds to a cycle that visits all the vertices in the graph exactly once.

We now show that HAMPATH $\leq_{\mathbf{P}}$ HAMCYCLE. Let F be the algorithm that takes the input $G = (V, E)$ and creates the graph $G' = (V', E')$ where the set of vertices of the new graph

$$V' = V \cup \{s\}$$

is obtained by adding a new vertex s and the set of edges

$$E' = E \cup \{(s, v) : v \in V\}$$

is the original set of edges along with an edge between every vertex in V and the new vertex s. This transformation is easily computed in polynomial-time, and to complete the reduction we need to show that G has a Hamiltonian path if and only if G' has a Hamiltonian cycle.

- If G has a Hamiltonian path P with endpoints $a, b \in V$, then the cycle obtained by adding the edges (s, a) and (b, s) to the beginning and the end of the path P, respectively, forms a Hamiltonian cycle in G'.
- If G' has a Hamiltonian cycle C, that cycle contains two edges that we can call (s, a) and (s, b) that are incident to s. Removing those two edges leaves us with a Hamiltonian path in G.

□

3. SubsetSum is NP-complete

We can also use a reduction from 3Sat to show that SubsetSum is **NP**-hard. The key insight for this reduction is that we will want to use *really* big numbers in the reduction.

Theorem 21.4. SubsetSum *is* **NP***-complete.*

Proof. We already saw in the last lecture that SubsetSum $\in$ **NP**. We complete the proof by showing that 3Sat $\leq_{\mathbf{P}}$ SubsetSum.

Given an instance φ of the 3Sat problem on n variables and with m clauses, we want to construct a set of numbers that we will turn into an instance of the SubsetSum problem. Let's start by building $2n$ numbers: one for each possible literal $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$. Each number will have m digits: digit j of x_i will be 1 if x_i is part of the jth clause and 0 otherwise, and likewise for all the negations. So we end up with numbers that look like this:

	C_1	C_2	C_3	$\cdots$	C_m
$x_1 =$	0	0	1	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1
$\neg x_2 =$	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0
$T =$	≥ 1	≥ 1	≥ 1	$\cdots$	≥ 1

If we have a satisfying assignment to φ, then we can choose a subset of those numbers such that when we take the sum of this subset of numbers, each digit is ≥ 1—specifically, either 1, 2, or 3. We can add another $2m$ numbers to act as "slack" numbers that we can add to other numbers to make each digit *exactly* 4 when each constraint is satisfied. Our set of numbers and our target number now look like this:

	C_1	C_2	C_3	$\cdots$	C_m
$x_1 =$	0	0	1	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1
$\neg x_2 =$	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0
$s_{1,1} =$	1	0	0	$\cdots$	0
$s_{1,2} =$	2	0	0	$\cdots$	0
$s_{2,1} =$	0	1	0	$\cdots$	0
$s_{2,2} =$	0	2	0	$\cdots$	0
$s_{3,1} =$	0	0	1	$\cdots$	0
$s_{3,2} =$	0	0	2	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$s_{m,1} =$	0	0	0	$\cdots$	1
$s_{m,2} =$	0	0	0	$\cdots$	2
$T =$	4	4	4	$\cdots$	4

There's one remaining issue that we need to handle: at the moment, nothing prevents us to choose the numbers corresponding to both x_i and $\neg x_i$ in our set of numbers that we choose to try to reach the target number. How can we disallow this? We can do it by adding extra digits to the numbers and to the target where the target can be reached only when we choose exactly one of x_i or $\neg x_i$ for each $i = 1, 2, \ldots, n$. The final construction looks like this:

	C_1	C_2	C_3	$\cdots$	C_m	x_1	x_2	$\cdots$	x_n
$x_1 =$	0	0	1	$\cdots$	0	1	0	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0	1	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1	0	1	$\cdots$	0
$\neg x_2 =$	1	1	0	$\cdots$	0	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0	0	0	$\cdots$	1
$s_{1,1} =$	1	0	0	$\cdots$	0	0	0	$\cdots$	0
$s_{1,2} =$	2	0	0	$\cdots$	0	0	0	$\cdots$	0
$s_{2,1} =$	0	1	0	$\cdots$	0	0	0	$\cdots$	0
$s_{2,2} =$	0	2	0	$\cdots$	0	0	0	$\cdots$	0
$s_{3,1} =$	0	0	1	$\cdots$	0	0	0	$\cdots$	0
$s_{3,2} =$	0	0	2	$\cdots$	0	0	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$s_{m,1} =$	0	0	0	$\cdots$	1	0	0	$\cdots$	0
$s_{m,2} =$	0	0	0	$\cdots$	2	0	0	$\cdots$	0
$T =$	4	4	4	$\cdots$	4	1	1	$\cdots$	1

This construction guarantees that there is a subset of the integers that sums up to T if and only if there is a satisfying assignment to φ. □

4. NP-complete problems are everywhere

There are a ton of **NP**-complete problems that have appeared in all areas of computer science. Some examples include the following.

Graph Colourability.: Vizing's theorem states that for every graph G with maximum degree Δ, either Δ or $\Delta + 1$ colours are necessary and sufficient to colour the edges of the graph so that no two edges that share a common vertex endpoint have the same colour. Given a graph G, can we determine if Δ or $\Delta + 1$ colours are required for that graph? That's an **NP**-complete problem!

Super Mario.: Many video games, including Super Mario and Tetris, involve decision problems (e.g., can you go from s to t in the map) that are **NP**-complete.

0-1 Linear programming.: A really powerful algorithmic tool that we do not cover in this class is *linear programming*. The usefulness of this technique stems from the fact that there are many natural problems that can be stated in the form

$$\begin{aligned} &\text{maximize} \sum_i x_i \text{subject to} && x_i + x_j \leq a_{i,j} \forall i, j \\ &0 \leq x_i \leq 1 && \forall i \end{aligned}$$

and that such problems can be solved in polynomial time. But if we consider the very slight modification of linear programs where we require the variables to take

only the values 0 or 1 (instead of any real value between them), then the (associated decision) problem becomes **NP**-complete.

Number theory.: Here's a simple basic number theoretic question: given some positive integers a and b, does there exist x such that $x^2 \equiv a \pmod{b}$? This problem is **NP**-complete.

We could go on and on—with problems in databases, computational geometry, networking, scheduling jobs on servers, comparing regular expressions, etc. In fact, there is a whole book devoted to collecting many of the fundamental **NP**-complete problems: Garey and Johnson's *Computers and Intractability: A Guide to the theory of* **NP***-completeness*. If you ever want to see if a problem X that you are considering is **NP**-complete, that's the right place to look first for a related problem A that is **NP**-complete and that would let you obtain an easy proof that $A \leq_{\mathbf{P}} X$ to show that X is **NP**-complete as well.

5. Closing thoughts

There is lots more we could say about the class **NP**. Here are perhaps the four most important questions in the FAQ about **NP** that we have not yet covered.

5.1. Do NP-complete and NP-hard mean the same thing? You will often see a problem labelled as "**NP**-hard" instead of **NP**-complete. This does *not* mean the same thing: it's a slightly weaker statement:

DEFINITION 21.5. The decision problem X is **NP**-*hard* if every problem $A \in \mathbf{NP}$ satisfies $A \leq_{\mathbf{P}} X$.

(In other words, for **NP**-hardness we do not need to show that X is in **NP**; for **NP**-completeness we *do* need to show that.) There are some problems that are **NP**-hard but not **NP**-complete, but almost all of the "hard" problems that you will encounter (and all of the ones we have covered so far in the course) are in **NP**, so for this class we will *always* aim to show that a problem is **NP**-complete, not just **NP**-hard.

5.2. Can we use other types of reductions to show that a problem is hard? Yes! There's a more general type of polynomial-time reduction between decision problems known as *Cook reductions*, which are also often called *Turing reductions*.

DEFINITION 21.6. There is a *Cook reduction* from A to B if we can design a polynomial-time algorithm F that, using a polynomial-time algorithm for B as a black-box, solves A in polynomial time.

This means that if we can show a Cook reduction from some problem HARD that we know has no polynomial-time algorithm to the problem X, then it means that there is no polynomial-time algorithm for X either.

Note that if $A \leq_{\mathbf{P}} B$ then there is a Cook reduction from A to B, but the converse is not known to be true. Let's discuss this a bit more in the next question.

5.3. Are there problems that we can't verify efficiently? Yes! Or, at least, there are problems that we *don't know* how to verify in polynomial time at the moment. Consider for example the problem we obtain by negating the CLIQUE problem.

DEFINITION 21.7. coCLIQUE: Given a graph $G = (V, E)$ and a positive integer k, is the largest clique of G of size $< k$?

Here it is easy to give a certificate that would help convince a verifier that the answer is `No`—but we have to provide certificates when the answer is `Yes` instead! Is there any way to define a certificate that would help convince a verifier that a graph does *not* contain any clique of size k? It's certainly not enough to give a particular subgraph of size k and show that this subgraph is not a clique—the clique could be elsewhere in the graph! In fact, we currently don't know of *any* way to efficiently verify the coClique problem.

The class of problems that have polynomial-time verifiers for `No` inputs instead of `Yes` inputs is known as **coNP**. It is believed that **NP** $\neq$ **coNP**... except that once again we can't hope to prove this statement before we first settle the famous **P** vs. **NP** problem.

5.4. Are there problems that are even harder than the ones in NP?. Yes, lots! There are even *undecidable* problems—problems that cannot be solved by *any* algorithm, no matter how efficient or inefficient. We'll talk about this in a bit more detail in one of the last two lectures of the class.

LECTURE 22

Approximation algorithms

1. Solving hard optimization problems

We saw that for many natural optimization problem—the Travelling Salesman Problem (TSP), Vertex Cover, Clique, etc.—even the (simpler) decision versions of the problems are **NP**-complete, and so we believe that there are no polynomial-time algorithms that solve those problems exactly. And as we have already mentioned a few weeks ago, if we really need to solve those problems, then we are faced with an important choice. We can:

- Use a *heuristic* algorithm that might work in practice but does not have any provable guarantees;
- Solve the problem exactly, using an exhaustive search algorithm; or
- Solve the problem *approximately* in polynomial time.

Today, we will briefly touch upon this last option, to see how for many optimization problems, there are very simple algorithms that might not always return the optimal solution but (provably!) always return a solution that is "not much worse" than the optimal solution.

2. Metric TSP

In the options we listed above, we assume that you *really* want to solve the hard problem itself. But in practice, your first task after you establish that a problem you are studying is **NP**-complete is to determine if there's another problem you can solve—or if it's possible to solve just a *special case* of your problem.

For example, TSP is used to model the minimum distance that a travelling salesman has to drive between cities to make its sales tour... but in practice we don't need to force the salesperson to visit each city *exactly* once if driving through one city on the way to another one is shorter than driving around it. So we can remove the condition, and from our original graph we can construct a new complete graph where the weight between the vertices u and v is the *shortest path* distance between u and v in the original graph. These distances will not be arbitrary, as they will now satisfy the *triangle inequality*. The problem of solving TSP on such graphs is known as the *metric TSP* problem.

Definition 22.1. In the MetricTSP problem, we are given a complete weighted graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}^{>0}$ that satisfies

$$w(u, v) \leq w(u, x) + w(x, v)$$

for every $u, v, x \in V$; our goal is to find the length ℓ_{TSP} of the shortest tour of G that visits each vertex in V.

MetricTSP is **NP**-complete, but there is a simple algorithm that is guaranteed to return a tour of G with length at most twice that of the optimal tour. Such an algorithm is known as a *2-approximation algorithm*.

DEFINITION 22.2. For any $k \geq 1$ and any minimization problem P, a *k-approximation algorithm* is an algorithm A where for every instance x of P with optimal value OPT, the algorithm $A(x)$ outputs a solution with value at most $k \cdot \text{OPT}$.

The 2-approximation algorithm for METRICTSP is as follows.

Algorithm 50: APPROXMETRICTSP$(G = (V, E), w)$

$T \leftarrow$ a MST of G;
$L \leftarrow$ list of vertices visited in an in-order traversal of T;
tour $\leftarrow$ tour through G obtained by shortcutting paths in L that visit previously-seen vertices;
return *tour*;

The algorithm simply gets a minimum spanning tree T of the graph G (using Kruskal or Prim's algorithms, for example), then visits the edges of this tree to obtain a tour through the graph that visits each vertex at least once (but possibly multiple times). The final solution returned is the tour of the graph obtained by replacing paths that revisit some of the vertices along the way with the shortest paths to the next unvisited vertex. This algorithm takes polynomial time.

THEOREM 22.3. APPROXMETRICTSP *is a 2-approximation algorithm for* METRICTSP.

PROOF. Let ℓ_{OPT} be the length of the optimal TSP tour through G and let ℓ_{MST} be the total length of the minimum spanning tree of G. Then

$$\ell_{MST} \leq \ell_{TSP}$$

because if we delete any edge from the TSP tour of G, we obtain a spanning tree of G. Also, by using the triangle inequality and observing that the in-order traversal of T visits each edge of the tree twice, we see that the length ℓ of the tour returned by the algorithm satisfies

$$\ell \leq 2\ell_{MST}.$$

Therefore, $\ell \leq 2\ell_{MST} \leq 2\ell_{OPT}$. □

3. Vertex Cover

Here's a nice and simple greedy algorithm for the VERTEXCOVER problem.

Algorithm 51: APPROXVC$(G = (V, E))$

$C \leftarrow \emptyset$;
for *each* $(u, v) \in E$ **do**
  if $u \notin C$ *and* $v \notin C$ **then**
    $C \leftarrow C \cup \{u, v\}$;
return C;

The APPROXVC algorithm runs in polynomial time, but it's not immediately clear that it returns a good approximation of the minimum vertex cover of G. As it turns out, it does quite well: the cover C returned by the algorithm can be at most twice as large as the optimal cover C_{OPT} of G.

THEOREM 22.4. APPROXVC *is a 2-approximation algorithm to the* VERTEXCOVER *problem.*

PROOF. Fix any input graph G. Let $M \subseteq E$ be the set of edges (u, v) that caused the algorithm to add u and v to the cover C. Then M forms a *matching* in G: no two edges in M share a common end vertex. We have that

$$|C| = 2|M|,$$

since both endpoints of every edge in M is added to the cover C. But we also have that

$$|M| \leq |C_{OPT}|$$

since any vertex cover of G must include at least one of the endvertices of each edge in M. Therefore, $|C| = 2|M| \leq 2|C_{OPT}|$. □

It's very tempting to try to improve on the APPROXVC algorithm by including only one of the two vertices of an edge that was not previously covered by C. (Perhaps taking the vertex that has the largest degree?) You should certainly try to do so! But you should also be warned that many people have tried without success so far: if you can obtain a k-approximation algorithm for VERTEXCOVER for any constant $k < 2$, this will be a significant breakthrough in algorithms research.

4. TSP

Emboldened by our success so far, we may want to conjecture that *every* **NP**-complete optimization problem has a polynomial-time 2-approximation algorithm, or some other reasonable approximation algorithm. This would be fantastic! Unfortunately, it's not true (Unless, of course, $\mathbf{P} = \mathbf{NP}$).

THEOREM 22.5. *If* $\mathbf{P} \neq \mathbf{NP}$*, then for any constant* $k \geq 1$*, there is no polynomial-time* k*-approximation algorithm for TSP.*

PROOF. We prove the contrapositive statement: that if there exists some constant k for which there is a polynomial-time algorithm A that is a k-approximation algorithm for TSP, then $\mathbf{P} = \mathbf{NP}$. We obtain this conclusion by showing that such an algorithm A can be used to design a polynomial-time algorithm for the **NP**-complete problem HAMCYCLE.

Algorithm 52: HAMCYCLESOLVER($G = (V, E)$)

```
Construct the complete graph G' = (V, (V choose 2));
for each (u, v) ∈ (V choose 2) do
    if (u, v) ∈ E then
        w(u, v) ← 1;
    else
        w(u, v) ← kn;
T ← A(G');
if length(T) ≤ kn then
    return Yes;
else
    return No;
```

If G contains a Hamiltonian cycle, then there is a tour of G' that uses only edges of weight 1 and so has total length n. This means that $A(G')$ returns a tour of length at most kn, and the algorithm HamCycleSolver correctly returns `Yes`.

If G contains no Hamiltonian cycle, then any tour of G' must use at least one edge of weight kn, so its total length must be at least $(n-1)+kn > kn$. But then any tour returned by A has length greater than kn and HamCycleSolver correctly returns `No`. □

5. Concluding remarks

These three examples provide just a tiny sample of the richness of approximation algorithms: there are problems in this area that can be computed with simple greedy algorithms; others that require complex arguments; some problems can be approximated to *arbitrary* accuracy in polynomial time; some can't be approximated to constant factors but can be approximated to $\log n$ factors; still others have no reasonable approximation algorithm whatsoever. And the idea of designing algorithms that approximately solve a given problem does not just help with overcoming **NP**-completeness—the same idea has given rise to *sublinear-time* algorithms (that only need to examine a tiny fraction of the input) and to *sublinear-space* (or *streaming*) algorithms—that only need to examine the data in a linear stream.

Part 6

Bonus lectures

LECTURE 23

On hard problems

1. Impossible problems

We spent quite a few lectures in the **NP**-completeness part of the course showing how there are many natural problems that we do not know how to solve in polynomial-time (and that many people believe we simply can't solve in polynomial time). But all of the problems that we saw can easily be solved if we allow our algorithms to run in time that is *exponential* in the size of their inputs.

There are problems that are even harder than that. In fact, there are problems that cannot be solved by any algorithm whatsoever—even if we were to let the algorithm run for some ridiculous amount of time (like $2^{2^{2^{2^n}}}$ steps on inputs of length n, or any other function you can come up with). Such problems are called *undecidable*. You have already seen such problems in earlier classes. One classic undecidable problem is the *halting problem*.

Definition 23.1 (Halting problem). Given as input the binary code for an algorithm, determine whether the code halts after a finite number of steps on every input.

Theorem 23.2 (Turing 1936). *There is no algorithm that can solve the* Halting *problem.*

This theorem is obtained with a beautiful application of the *diagonalization* proof technique. But if you're like me, even knowing that this theorem is true, it still doesn't seem possible. After all, how hard can it really be to determine if natural algorithms (as opposed to rather contrived examples that you might cook up just to prove this result) halt or not? Again, however, we really don't need to look far before we see just how hard the problem really is. Take the following simple procedure that takes in a positive integer x as input.

Algorithm 53: Collatz(x)

```
while x > 1 do
    if x is even then
        x ← x/2;
    else
        x ← 3x + 1;
```

Does this algorithm halt after a finite number of steps on every input x? We don't know! Lothar Collatz introduced the problem in 1937 and conjectured that the procedure does always halt, but nobody has been able to make any significant progress on the question apart from trying it experimentally on (many, many) specific integers. My favourite quote about the problem is from Paul Erdős, who stated that, quite simply, "mathematics may not be ready for such problems."

We can even consider an easier variant of the HALTING problem that only takes in algorithms which don't even have any inputs (or the variant where we only care to know if the algorithm halts on a given input). Even with this simplification, the situation is grim: there is no algorithm that can solve this variant of the HALTING problem either. And once again there are natural examples of algorithms for which we don't even know whether they halt or not. For instance, you may have heard of Goldbach's conjecture from 1742: can every even number be represented as the sum of two primes? That's a conjecture that we could resolve if we could determine whether the following algorithm halts.

Algorithm 54: GOLDBACH

```
n ← 4;
counter-example ← False;
while !counter-example do
    n ← n + 2;
    counter-example ← True;
    for x = 3, 5, 7, ..., n − 1 do
        if IsPrime(x) and IsPrime(n − x) then
            counter-example ← False;
return "n is a counter-example to the Goldbach conjecture!"
```

(There is a polynomial-time algorithm that solves ISPRIME, though for the task of settling Goldbach's conjecture it would suffice to implement the function with a simple naïve test that checks whether any number in the range $2, 3, \ldots, x-1$ divides x.)

If you are currently considering a problem that you suspect is impossible to solve with an algorithm, you can prove that this is the case using the notion of reduction we have worked on extensively over the last few weeks. In this setting, the only difference is that your reduction does not need to run in polynomial time.

THEOREM 23.3. *If X is a decision problem for which the reduction* HALTING $\leq X$ *holds, then there is no algorithm that can solve X.*

2. Very difficult problems

Even when we restrict our attention to problems that *can* be solved with algorithms, we can still identify some very difficult problems—problems that are even more difficult than the **NP**-complete problems we have seen already.

One particularly notable difficult problem is *TQBF*, which stands for "Totally Quantified Boolean Formula".

DEFINITION 23.4 (TQBF problem). Given a totally quantified Boolean formula

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \phi(x_1, \ldots, x_n),$$

determine if the formula is true or not.

This problem captures a wide variety of natural 2-player games, as we can see by noticing that trying to figure out a winning strategy for such a game consists of trying to answer the question "Is there a first move M_1 that I can make such that no matter what move M_2 my opponent makes, there exists a response M_3 where for any possible next move M_4 of the opponent I have a further move M_5... for which I end up in a winning position?".

Note that unlike SAT, TQBF appears to be both hard to solve *and* hard to verify—after all, what certificate could you devise that would convince a polynomial-time verifier that a given formula is true? Whether or not TQBF can be solved in polynomial-time, however, remains open: it is in fact equivalent to the problem of determining whether all problems that can be solved by algorithms that use a polynomial amount of memory can also be solved by algorithms that use a polynomial amount of time. (In complexity theory terms: does $\mathbf{P} = \mathbf{PSPACE}$?)

There are also problems that are *known* to require more than polynomial time. One such problem is another variant of the HALTING problem.

DEFINITION 23.5 (HALTINkSTEPS problem). Given an algorithm A, an input x to A, and a positive integer k, does A halt on input x after at most k steps?

When we fix A and x and then consider $k \to \infty$, then the input to the HALTINkSTEPS problem has size $O(\log k)$. And this problem can be solved in time (roughly) $O(k)$ by simulating the algorithm A for k steps. A modification of the proof of Turing's theorem shows that this is also essentially best possible: all algorithms that solve the HALTINkSTEPS problem have time complexity $\Omega(k)$, which is exponential in the input size.

Finally, for a problem that can't even be solved in exponential time, consider the task of determining if two regular expressions are equivalent to each other: if we let these regular expressions have negations as well as the usual union and star operators, then this problem can be solved but has time complexity that is not bounded by $O(2^{2^{2^{\cdots^{2^n}}}})$ when this tower has *any* constant type.

These problems show that there is a wide variety of different difficulties for computational problems. And, once again, if you suspect that the problem X you are considering is at least as hard as any of them, a polynomial-time reduction from one of these problems to X is all you need to prove that fact.

3. Rather difficult problems

Let's now turn back our attention to the more "reasonable" problems in **NP**. Let's say that we're considering some problem like CLIQUE. We have seen that we can't hope to show that there is no polynomial-time algorithm that can solve CLIQUE before we resolve the infamous **P** vs. **NP** problem. But what if we *really* want an unconditional lower bound on the amount of time required to solve the problem? This shouldn't be too unreasonable to ask for—if we don't expect even $O(n^{1000})$-time algorithms to be able to solve CLIQUE, perhaps we could at least hope for a lower bound of, say $\Omega(n^4)$, to say that it's at least "not very easy"!

Alas, even that is too much to ask for. The best lower bound we have for CLIQUE is the following.

THEOREM 23.6. *Any algorithm that solves* CLIQUE *must have time complexity* $\Omega(n+m)$ *on input graphs with* n *vertices and* m *edges.*

PROOF IDEA. Any deterministic algorithm that solves CLIQUE must at least read all of its input before producing the answer! □

Amazingly, this lower bound is essentially the only unconditional lower bound we have on the time complexity of *any* explicit problem in **NP**, as long as we consider all possible algorithms—the only stronger lower bounds that we have apply to specific models of computation.

4. Slightly difficult problems

The fact that we don't have any non-trivial unconditional lower bounds for CLIQUE is mitigated by the fact that we can at least show that the problem is **NP**-complete and thereby at least obtain some explanation/justification for the fact that we can't come up with polynomial-time algorithm for the problem. But what if we are considering a problem that *is* in **P**? If you're looking at a well-studied problem, like 3SUM, then the fact that many researchers have looked at the problem before you without finding better algorithms than the one you have is good information. In fact, for this problem, it has even been conjectured that (essentially) the best possible algorithm has already been found.

DEFINITION 23.7. In the 3SUM problem, we are given a set $S \subseteq \mathbb{Z}$ of integers and we must determine whether there are three integers $a, b, c \in S$ such that $a + b + c = 0$.

CONJECTURE 1 (3SUM conjecture). *For every $\epsilon > 0$, any algorithm that solves* 3SUM *has time complexity $\Omega(n^{2-\epsilon})$.*

But what if you are considering a new problem that, as far as you know, nobody has studied before? Take the following problem, for instance.

DEFINITION 23.8. In the COLINEAR problem, we are given a set of n points in the plane, and we must determine if there is a line in the plane that passes through at least 3 of the points.

How efficiently can you solve this problem? There is a simple $O(n^3)$-time algorithm: enumerate all sets of 3 points and for each such set determine if they are in a line or not. Can you do better? With some work, we can obtain an $O(n^2 \log n)$ time algorithm and you may even find an algorithm with time complexity $O(n^2)$. But then no matter how hard you try, you can't do any better. Can we explain why we get stuck at this point? Indeed we can, using the idea of reductions!

THEOREM 23.9. *If the* 3SUM *conjecture is true, then there is no algorithm that solves* COLINEAR *and has time complexity $O(n^c)$ for any constant $c < 2$.*

PROOF. The idea is that we can construct a reduction from 3SUM to COLINEAR that takes only $O(n)$ time—this means that if we obtain an algorithm for COLINEAR that runs in time $O(n^c)$ for some $1 \leq c < 2$, we can apply the reduction and invoke this algorithm to solve 3SUM in time $O(n + n^c) = O(n^c)$.

What is the reduction? For each integer $a \in S$ in the 3SUM problem, add the point (a, a^3) to the instance of COLINEAR. Then three points (a, a^3), (b, b^3), and (c, c^3) are colinear if and only if

$$\begin{aligned} \frac{b^3 - a^3}{b - a} &= \frac{c^3 - b^3}{c - b} \\ \Leftrightarrow a^2 + b^2 + ab &= b^2 + c^2 + bc \\ \Leftrightarrow a^2 + ab + ac &= c^2 + bc + ac \\ \Leftrightarrow (a - c)(a + b + c) &= 0 \end{aligned}$$

But this can only hold when $a + b + c = 0$ since all the integers in S are disjoint and so $a - c \neq 0$. □

LECTURE 24

Three fun problems

In the CS 341 class, we covered a number of algorithm design and analysis techniques that are very useful for solving many different problems. But, perhaps just as important as the techniques themselves is the *idea* that the best way to solve many practical problems is to first formulate them as simple and clean abstract problems that we can then tackle effectively. In this last lecture, we cover three very different problems that are outside the scope of CS 341—and that are very different from all the problems we have seen so far—to see how we can do this.

1. Finding the median

We saw in a tutorial that there is a linear-time algorithm for finding the median in a set of n integers using the divide and conquer technique. This is great if all your data is stored locally—but what if the data is so large that it needs to be stored on 2 different servers? We can certainly run the same algorithm—perhaps on a third server that accesses the data on the 2 servers with queries made through the internal communication network. But then we will quickly find out that the resulting algorithm is extremely slow, and that the bottleneck for the algorithm is the amount of data that needs to be sent across the communication network: this algorithm requires a *linear* amount of communication, i.e., we essentially need to send the entire contents of at least one of the servers through the network, and this will take an enormous amount of time!

Can we do better? We have already done the first step in designing a more efficient algorithm: we have identified the bottleneck and most significant factor in the running time of any algorithm that solves the problem. In this case, it is the amount of communication required between the machines. (By comparison to the amount of time it takes to send any data across the network, local computation on any machine is essentially free.) Our next step is to see how we can define the problem we are trying to solve as precisely (and as simply!) as we can. We can do so in the *communication complexity* model of computation.

Definition 24.1. In the communication complexity version of the Median problem, two players named Alice and Bob hold sets of integers. Specifically, Alice has a list L_a of n integers in the range $\{1, 2, \ldots, n\}$ and Bob has a list L_b of n integers in the same range. They want to identify a median m of $L_a \cup L_b$, and the goal is to do so by designing a *protocol* (=algorithms run by Alice and Bob that can send messages to each other) that requires as few bits of communication as possible.

Note that here we *don't* worry about the time complexity of the algorithms, only the amount of communication exchanged between Alice and Bob. As we argued above, this is a realistic model of the cost of running algorithms on separate machines since communication will be orders of magnitude slower than any local computation.

There is a simple protocol that requires n bits of communication: Alice sends all her data to Bob, who then runs any median-finding algorithm on the joint data. Can you do better?

THEOREM 24.2. *There is an algorithm that solves the communication complexity version of the* MEDIAN *problem and requires only* $O(\log^2 n)$ *bits of communication between Alice and Bob.*

PROOF. The idea is to apply divide & conquer (or binary search) techniques. We can design a protocol where in each round, Alice and Bob know that the median lies in the range $[i, j]$ and, after the round, they know that the median lies in either $[i, \frac{i+j}{2}]$ or in $[\frac{i+j}{2}, j]$. Initially, Alice and Bob know that the median must lie in the range $[1, n]$ so after at most $O(\log n)$ rounds, they will have identified the median.

So let us now solve the single-round problem. Alice and Bob start the round both knowing the values of i and j. They can both compute $k = \lfloor \frac{i+j}{2} \rfloor$ (without using any communication!). Then Alice can count how many of her values are smaller than k and send that number to Bob, using $O(\log n)$ bits of communication. Bob can add that number to the number of his elements that are smaller than k: if the total is exactly n, then k is the median! If the total is less than n, then the median must be in the range $[k, j]$. And if the total is more than n, then the median must be in the range $[i, k]$. □

The protocol described above will run much, much faster than any MEDIAN algorithm that uses a linear amount of communication between the machines. You might be curious, however, if it is the *best* we can do for the problem. It's not: there is another protocol that only requires $O(\log n)$ bits of communication.

CHALLENGE 24.3. *Design a communication protocol for the* MEDIAN *problem that requires only* $O(\log n)$ *bits of communication between Alice and Bob.*

2. Heavy hitters

Here's a completely different problem: you're now in charge of all internet servers. You have a server (or a router) that handles requests, but because of possible attacks on your servers you want to notice whenever there is some source that is making too many requests. Here, we can define "too many" as, say, $n/100$ of all n queries in a day. The challenge here is that the requests are coming in fast, and you certainly don't have time or space to store much of the sources' information locally. And unlike all the other problems we have seen so far, we have no control over the order in which we access the data: we must process the requests in the order that they are given to us. This problem is known as the *heavy hitters* problem, and it can be formalized in the *streaming algorithms* framework.

DEFINITION 24.4. In the HEAVYHITTERS problem, we are given some value k and we observe a sequence of positive integers $x_1, x_2, \ldots, x_n \in \{1, 2, \ldots, n\}$ in that order. We want to output a set S of at most k integers such that any integer that appears more than n/k times in the sequence will be in S. The goal is to do this while using as little memory as possible.

This problem is certainly an idealized version of the original problem. First, we simplified the problem itself by considering a sequence of integers instead of more complex objects (like source IP addresses). But if we solve this problem we'll see that it's very easy to replace "integers" with any other types of values in the future. Second, we don't say

anything about time complexity—though in this case if we have a simple router we *do* want to make sure we have an efficient algorithm. In this case, the motivation for the simplification is the observation that reducing the memory requirements of the algorithm appears to be the most challenging aspect of the problem, so it's best just to focus on that first. If we succeed in designing an algorithm that requires little memory, we can then focus on fast computation time afterwards. And third, you might notice that our requirement for a solution to the problem is weaker than what we might want in practice. (For instance, it might be important not to have any *false positives*; we might want *all* the entries in S to be present at least n/k times in the sequence.) This is again a simplification meant to help us obtain at least some progress on the original question: once we have solved this version of the problem, we can certainly revisit our solution to try to make it even stronger!

Let's again begin addressing the problem with a trivial solution: we can certainly solve the problem with n cells (so $O(n \log n) bits$) of memory: we store all the integers in the sequence locally, and then we run any algorithm that we like to find any heavy hitters in the sequence. But of course our goal is to do much better.

THEOREM 24.5. *There is an algorithm that solves the* HEAVYHITTERS *problem and requires only $O(k \log n)$ bits of memory.*

PROOF. There can be at most k different heavy hitters in any sequence. The idea of the algorithm is to keep track of (at most) k candidate heavy hitters at all times along with a counter for each of these candidates. Then we run a simple greedy algorithm: whenever we see the next element x_i of the sequence we first check if it is one of the candidate heavy hitters. If so, we increment its counter. If it's not a candidate but we have space to store one more candidate, we add it to our list. Otherwise, we decrement the counter of each candidate currently in memory and expel any candidate that reached the count 0. The resulting algorithm is known as the *Misra–Gries algorithm*.

Algorithm 55: MISRAGRIES(k, n)

```
A ← ∅;
for i = 1, 2, ..., n do
    if x_i ∈ A then
        count[x_i] ← count[x_i] + 1;
    else if |A| < k then
        A ← A ∪ {x_i};
        count[x_i] ← 1;
    else
        for each j ∈ A do
            count[j] ← count[j] − 1;
            if count[j] = 0 then
                A ← A \ {j};
return A;
```

The key observation in the proof of correctness of this algorithm is that if j is a heavy hitter (that occurs more than n/k times in the stream) then every time we decrement the count for j in the algorithm, we are also decrementing the count for $k-1$ other variables as well. And since every element in the sequence causes at most 1 counter increment and

we have at least as many increment as decrement operations, it means that at most n/k of the elements in the sequence can cause a decrement operation. Therefore, every heavy hitter element will be in A when the algorithm terminates. □

CHALLENGE 24.6. *Improve the algorithm so that the set A returned by the algorithm satisfies two conditions:*

(1) *Every heavy hitter that appears more than n/k times in the sequence is present in A, and*
(2) *Every value returned in A is a "moderately heavy hitter" that appears at least $n/2k$ times in the sequence.*

3. Maximal independent set

One last problem in yet another completely different setting. We now have a very large set of sensors distributed over some environment. Two sensors *overlap* if their measurements are not independent of each other (e.g., if they are close to each other). We want to find a maximal set of sensors that do not overlap with each other. We don't have a centralized server connecting all the sensors, so the algorithm we design for this task has to be a *distributed algorithm*.

Note that for this problem the difference between *maximal* and *maximum* is quite important: we want a set S of non-overlapping sensors such that every other sensor overlaps with one of the sensors in S (so that we can't add to our current set without causing some overlaps), but S does not have to be the largest possible set of non-overlapping sensors.

The problem can be phrased as a graph problem.

DEFINITION 24.7. In the MAXINDEPSET problem, we have a graph $G = (V, E)$ on $|V| = n$ vertices that has maximum degree d. We run a (distributed) algorithm on each node such that each node can perform local computation and communicate with its neighbours. At the end of the computation, each node determines whether it is included in the final set $I \subseteq V$ or not. The final set I must be a maximal independent set in G, and our goal is to satisfy this condition while minimizing the (parallel) time complexity of the algorithm.

There is a very simple greedy algorithm for finding a maximal independent set in a graph: choose the vertex with the smallest label, add it to I, remove all its neighbours from the graph, and repeat. This algorithm is very efficient in the standard model of computation, but it does not parallelize: how could a node figure out that it is the one with the smallest label left in the graph without communicating with many other nodes in the graph?

Now, a small detour: we might get stuck on this question for a while and eventually throw our hands up in the air in frustration and decide that in the end, it's just much easier to go back to the centralized model and have a computer be in charge of knowing where all the sensors are and which ones overlap, so that it can compute the maximum non-overlapping set (or, abstractly: compute a maximal independent set of the overlapping graph) directly. Good idea! But given that our sensor networks will be quite large, we really would like our algorithm to *parallelize* effeciently, so that if we use an expensive multi-core computer as our centralized server, we do use all the cores effectively in our computation... but if that's what we want, we realize that we're right back to where we started: the simple maximal independent set algorithm will *not* parallelize efficiently, and the tasks of designing a distributed or a parallelizable algorithm for MAXINDEPSET are in fact closely related to each other.

THEOREM 24.8. *There is a randomized algorithm that solves the* MAXINDEPSET *in expected time* $O(d \log n)$.

PROOF IDEA. The idea of the algorithm is to divide up the computation into a series of rounds. In each round, all the nodes that are still active generate a random number between 0 and 1. The nodes share their generated number with all of their neighbours. If a vertex v has the minimal number among all its neighbours, it adds itself to I and all its neighbours decide that they won't be in I; all these nodes then become inactive. And we repeat this process until all the nodes become inactive. That's it! The standard pseudo-code for the algorithm would look like this:

Algorithm 56: LUBY($G = (V, E)$)

```
I ← ∅;
while V ≠ ∅ do
    for each v ∈ V do
        r[v] ← random number in [0, 1];
        if r[v] < r[w] for each w ∈ N(v) then
            I ← I ∪ {v};
            V ← V \ ({v} ∪ N(v));
return I;
```

As a challenge: can you see how to write pseudo-code for the distributed version of the algorithm (that you would run on each node)? And how would you carefully describe the parallel version of the algorithm?

The correctness of the algorithm follows from the fact that you can never have two vertices connected by an edge add themselves to I—at most one of the two can have the minimum number $r[v]$ among all its neighbours, and when one does, the other node becomes inactive. The more challenging aspect of the analysis of this algorithm is the time complexity analysis. This is left as another exercise. □

CHALLENGE 24.9. *Prove that each round of Luby's algorithm eliminates a constant fraction of the edges in the (active) graph in expectation. (Extra challenge: prove that this is true with high probability as well.)*