CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

April 5, 2019

1. Course Information
2. Introduction
3. Divide-and-Conquer Algorithms
4. Greedy Algorithms
5. Dynamic Programming Algorithms
6. Graph Algorithms
7. Intractability and Undecidability

Table of Contents

1 Course Information

Course mechanics

- My Sections:
  - LEC 003, T Th 4:00–5:20, PHY 235
  - LEC 004, T Th 1:00–2:20, MC 2035
- My Scheduled Office Hours:
  - Tuesday, 11:00–12:00, DC 3522

Course mechanics

- Come to class! Not all the material will be on the slides or in the text.
- You will need an account in the student.cs environment
- The course website can be found at

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

  - Syllabus, calendar, policies, etc. can be found there.

Learn and Piazza

- Slides and assignments will be available on the course website.
- Grades and assignment solutions will be available on Learn.
- Discussion related to the course will take place on Piazza (piazza.com).
  - General course questions, announcements
  - Assignment-related questions
  - You will be getting an invitation via email to join Piazza in the first week of classes.
- Keep up with the information posted on the course website, Learn and Piazza.

Courtesy

- Please silence cell phones and other mobile devices before coming to class.
- Questions are encouraged, but please refrain from talking in class – it is distracting to your classmates who are trying to listen to the lectures and to your professor who is trying to think, talk and write at the same time.
- Carefully consider whether using your laptop, ipad, smartphone, etc., in class will help you learn the material and follow the lectures.
- Do not play games, tweet, watch youtube videos, update your facebook page or use a mobile device in any other way that will distract your classmates.

Course syllabus

- You are expected to be familiar with the contents of the course syllabus
- Available on the course home page
- If you haven't read it, read it as soon as possible

Plagiarism and academic offenses

- We take academic offences very seriously
- There is a good discussion of plagiarism online:
  - `https://uwaterloo.ca/academic-integrity/sites/ca.academic-integrity/files/uploads/files/revised_undergraduate_tip_sheet_2013_final.pdf`
- Read this and understand it
  - Ignorance is no excuse!
  - Questions should be brought to instructor
- Plagiarism applies to both text and code
- You are free (even encouraged) to exchange ideas, but no sharing code or text

Plagiarism (2)

- Common mistakes
  - Excess collaboration with other students
    - ★ Share ideas, but no design or code!
  - Using solutions from other sources (like for previous offerings of this course, maybe written by yourself)
- More information linked to from course syllabus

Grading scheme for CS 341

- Midterm (25%)
  - Tuesday, Feb. 26, 2019, 7:00–8:50 PM
- Assignments (30%)
  - There will be five assignments.
  - Work alone
  - See syllabus for reappraisal policies, academic integrity policy, and other details
- Final (45%)
- For medical conditions, you need to submit a Verification of Illness form.

Assignments

- All sections will have the same assignments, midterm and final exam.
- Assignments will be due at 6:00 PM on the due date.
- No late submissions will be accepted.
- You need to notify your instructor well before the due date of a severe, long-lasting or ongoing problem that prevents you from doing an assignment.

Assignment due dates

- Assignment 1: due Friday Jan. 25
- Assignment 2: due Friday Feb. 8
- Assignment 3: due Friday March 1
- Assignment 4: due Friday March 22
- Assignment 5: due Friday April 5

Textbook

- Introduction to Algorithms, Third Edition, by Cormen, Leiserson, Rivest and Stein, MIT Press, 2009.

- You are expected to know
  - all the material presented in class
  - relevant textbook sections, as listed on course website

Table of Contents

2 **Introduction**
- Algorithm Design and Analysis
- The 3SUM Problem
- Reductions
- Definitions and Terminology
- Order Notation
- Formulae
- Loop Analysis Techniques

Analysis of Algorithms

In this course, we study the **design** and **analysis** of algorithms. "Analysis" refers to mathematical techniques for establishing both the **correctness** and **efficiency** of algorithms.

Correctness: The level of detail required in a proof of correctness depends greatly on the type and/or difficulty of the algorithm.

Analysis of Algorithms (cont.)

Efficiency: Given an algorithm A, we want to know how efficient it is. This includes several possible criteria:

- What is the **asymptotic complexity** of algorithm A?
- What is the **exact number** of specified computations done by A?
- How does the **average-case** complexity of A compare to the **worst-case** complexity?
- Is A the most efficient algorithm to solve the given problem? (For example, can we find a **lower bound** on the complexity of **any** algorithm to solve the given problem?)
- Are there problems that cannot be solved efficiently? This topic is addressed in the theory of **NP-completeness**.
- Are there problems that cannot be solved by **any** algorithm? Such problems are termed **undecidable**.

Design of Algorithms

"Design" refers to **general strategies** for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms. Also, we want to avoid using **ad hoc** algorithms that are hard to analyze and understand.

Here are some examples of useful design strategies, many of which we will study:

reductions

divide-and conquer

greedy

dynamic programming

depth-first and breadth-first search

local search

exhaustive search (backtracking, branch-and-bound)

The "3SUM" Problem

Problem 2.1

3SUM

Instance: *an array A of n distinct integers, $A = [A[1], \ldots, A[n]]$.*

Question: *do there exist three distinct elements in A whose sum equals 0?*

The **3SUM** problem has an obvious algorithm to solve it.

Algorithm: *Trivial3SUM*$(A = [A[1], \ldots, A[n]])$

for $i \leftarrow 1$ **to** $n-2$

do
- **for** $j \leftarrow i+1$ **to** $n-1$
  - **do**
    - **for** $k \leftarrow j+1$ **to** n
      - **do**
        - **if** $A[i] + A[j] + A[k] = 0$
          - **then output** (i, j, k)

The complexity of *Trivial3SUM* is $O(n^3)$.

An Improvement

Instead of having three nested loops, suppose we have two nested loops (with indices i and j, say) and then we search for an $A[k]$ for which $A[i] + A[j] + A[k] = 0$.

If we sequentially try all possible k-values, then we basically have the previous algorithm.

What can we do to make the search more efficient?

What effect does this have on the complexity of the resulting algorithm?

An Improved Algorithm for the "3SUM" Problem

Algorithm: *Improved3SUM*$(A = [A[1], \ldots, A[n]])$

$(B, \pi) \leftarrow$ *sort*(A)

comment: π is the permutation such that $A[\pi(i)] = B[i]$ for all i

for $i \leftarrow 1$ **to** $n - 2$

do
- **for** $j \leftarrow i + 1$ **to** $n - 1$
  - **do**
    - perform a binary search for $B[k] = -B[i] - B[j]$ in the subarray $[B[j + 1], \ldots, B[n]]$
    - **if** the search is successful
      - **then output** $(\pi(i), \pi(j), \pi(k))$

Note that $\pi(i), \pi(j), \pi(k)$ are indices in in the original array A.

The complexity of *Improved3SUM* is $O(n \log n + n^2 \log n) = O(n^2 \log n)$.

The $n \log n$ term is the *sort*. The $n^2 \log n$ term accounts for n^2 binary searches.

A Further Improvement

The *sort* is an example of **pre-processing.**

It modifies the input to permit a more efficient algorithm to be used (binary search as opposed to linear search).

Note that a pre-processing step is only done once.

However, there is a better way to make use of the sorted array B.

Namely, for a given $B[i]$, we simultaneously scan from both ends of A looking for $B[j] + B[k] = -B[i]$.

We start with $j = i + 1$ and $k = n$.

At any stage of the algorithm, we either increment j or decrement k (or both, if $B[i] + B[j] + B[k] = 0$).

The resulting algorithm will have complexity $O(n \log n + n^2) = O(n^2)$.

A Quadratic Time Algorithm for the "3SUM" Problem

Algorithm: *Quadratic3SUM*$(A = [A[1], \ldots, A[n]])$

$(B, \pi) \leftarrow$ *sort*(A)

comment: π is the permutation such that $A[\pi(i)] = B[i]$ for all i

for $i \leftarrow 1$ **to** $n - 2$

do
- $j \leftarrow i + 1$
- $k \leftarrow n$
- **while** $j < k$
  - **do**
    - $S \leftarrow B[i] + B[j] + B[k]$
    - **if** $S < 0$ **then** $j \leftarrow j + 1$
    - **else if** $S > 0$ **then** $k \leftarrow k - 1$
    - **else**
      - **output** $(\pi(i), \pi(j), \pi(k))$
      - $j \leftarrow j + 1$
      - $k \leftarrow k - 1$

Example

Consider the sorted array $-11 \quad -10 \quad -7 \quad -3 \quad 2 \quad 4 \quad 8 \quad 10$.

The algorithm will not find any solutions when $i = 1$:

$\boxed{-11}$	$\boxed{-10}$	-7	-3	2	4	8	$\boxed{10}$	$S = -11$
$\boxed{-11}$	-10	$\boxed{-7}$	-3	2	4	8	$\boxed{10}$	$S = -8$
$\boxed{-11}$	-10	-7	$\boxed{-3}$	2	4	8	$\boxed{10}$	$S = -4$
$\boxed{-11}$	-10	-7	-3	$\boxed{2}$	4	8	$\boxed{10}$	$S = 1$
$\boxed{-11}$	-10	-7	-3	$\boxed{2}$	4	$\boxed{8}$	10	$S = -1$
$\boxed{-11}$	-10	-7	-3	2	$\boxed{4}$	$\boxed{8}$	10	$S = 1$

When $i = 2$, we have

-11	$\boxed{-10}$	$\boxed{-7}$	-3	2	4	8	$\boxed{10}$	$S = -7$
-11	$\boxed{-10}$	-7	$\boxed{-3}$	2	4	8	$\boxed{10}$	$S = -3$
-11	$\boxed{-10}$	-7	-3	$\boxed{2}$	4	8	$\boxed{10}$	$S = 2$
-11	$\boxed{-10}$	-7	-3	$\boxed{2}$	4	$\boxed{8}$	10	$S = 0$

Reductions

Suppose Π_1 and Π_2 are problems and we can use a hypothetical algorithm solving Π_2 as a subroutine to solve Π_1.

Then we have a **Turing reduction** (or more simply, a **reduction**) from Π_1 to Π_2; this is denoted $\Pi_1 \leq \Pi_2$.

The hypothetical algorithm solving Π_2 is called an **oracle**.

The reduction must treat the oracle as a **black box**.

There may be more than one call made to the oracle in the reduction.

Suppose

- $\Pi_1 \leq \Pi_2$ and
- we also have an algorithm $\mathbf{A_2}$ that solves Π_2.

If we plug $\mathbf{A_2}$ into the reduction, then we obtain an algorithm that solves Π_1.

Reductions potentially allow re-using code, which may be advantageous.

A Simple Reduction

Consider the algebraic identity

$$xy = \frac{(x+y)^2 - (x-y)^2}{4}.$$

This identity allows us to show that **Multiplication** $\leq$ **Squaring**.

Algorithm: *MultiplicationtoSquaring*(x, y)
external *ComputeSquare*
$s \leftarrow$ *ComputeSquare*$(x+y)$
$t \leftarrow$ *ComputeSquare*$(x-y)$
return $((s-t)/4)$

Note that the "division by 4" just consists of deleting the two low-order bits, which are guaranteed to be 00.

The "Target 3SUM" Problem

Problem 2.2

Target3SUM

Instance: *an array A of n distinct integers, $A = [A[1], \ldots, A[n]]$ and a target T.*

Question: *do there exist three distinct elements in A whose sum equals T?*

It is straightforward to modify any algorithm solving the **3SUM** problem so it solves the **Target3SUM** problem.

Another approach is to find a reduction **Target3SUM** $\leq$ **3SUM**. This would allow us to re-use code as opposed to modifying code.

Target3SUM $\leq$ 3SUM

Observe that $A[i] + A[j] + A[k] = T$ if and only if

$$(A[i] - T/3) + (A[j] - T/3) + (A[k] - T/3) = 0.$$

This suggests the following approach, which works if T is divisible by three.

Algorithm: *Target3SUMto3SUM*$(A = [A[1], \ldots, A[n]], T)$
comment: assume T is divisible by 3
external *3SUM-solver*
for $i \leftarrow 1$ **to** n
  do $B[i] \leftarrow A[i] - T/3$
return (*3SUM-solver*(B))

Modification: the transformation $B[i] \leftarrow 3A[i] - T$ works for any integer T.

Complexity analysis

Suppose we replace the oracle *3SUM-solver* by a "real" algorithm. What is the complexity of the resulting algorithm?

If we plug in *Trivial3SUM*, the complexity is $O(n + n^3) = O(n^3)$.

If we plug in *Improved3SUM*, the complexity is $O(n + n^2 \log n) = O(n^2 \log n)$.

If we plug in *Quadratic3SUM*, the complexity is $O(n + n^2) = O(n^2)$.

In each case, it turns out that the "n" term is subsumed by the second term.

Many-one Reductions

The reduction on the previous slide had a very special structure:

- We transformed an instance of the first problem to an instance of the second problem.
- We called the oracle once, on the transformed instance.

Reductions of the this form, in the context of decision problems, are called **many-one reductions** (also known as **polynomial transformations** or **Karp reductions**).

We will many examples of these in the section on **intractability**.

The "3array3SUM" Problem

Problem 2.3

3array3SUM

Instance: *three arrays of n distinct integers, A, B and C.*

Question: *do there exist array elements, one from each of A, B and C, whose sum equals 0?*

Algorithm: *3array3SUMto3SUM*(A, B, C)
external *3SUM-solver*
for $i \leftarrow 1$ **to** n
do $\begin{cases} D[i] \leftarrow 10A[i] + 1 \\ E[i] \leftarrow 10B[i] + 2 \\ F[i] \leftarrow 10C[i] - 3 \end{cases}$
let A' denote the concatenation of D, E and F
if *3SUM-solver*$(A') = (i, j, k)$
then return $(i, j - n, k - 2n)$

3array3SUM $\leq$ 3SUM (cont.)

To show that *3array3SUMto3SUM* is a reduction, we show that (i, j, k) is a solution to the instance A' of *3SUM* if and only if $(i, j - n, k - 2n)$ is a solution to the instance A, B, C of *3array3SUM*.

Assume first that (i', j', k') is a solution to *3array3SUM*. Then

$$A[i'] + B[j'] + C[k'] = 0.$$

Hence,

$$D[i'] + E[j'] + F[k'] = 10A[i'] + 1 + 10B[j'] + 2 + 10C[k'] - 3 = 0$$

and thus

$$A'[i'] + A'[j' + n] + A'[k' + 2n] = 0.$$

Conversely, suppose that $A'[i] + A'[j] + A'[k] = 0$. We claim that this sum consists of one element from each of D, E and F. This can be proven by considering the sum modulo 10 and observing that the only way to get a sum that is divisible by 10 is $1 + 2 - 3 \bmod 3$. The result follows by translating back to the original arrays A, B and C.

Problems

Problem: Given a problem instance I for a problem **P**, carry out a particular computational task.

Problem Instance: **Input** for the specified problem.

Problem Solution: **Output** (correct answer) for the specified problem.

Size of a problem instance: Size(I) is a positive integer which is a measure of the size of the instance I.

Algorithms and Programs

Algorithm: An algorithm is a step-by-step process (e.g., described in **pseudocode**) for carrying out a series of computations, given some appropriate input.

Algorithm solving a problem: An Algorithm A **solves** a problem **P** if, for every instance I of **P**, A finds a valid solution for the instance I in finite time.

Program: A program is an **implementation** of an algorithm using a specified computer language.

Running Time

Running Time of a Program: $T_{\mathbf{M}}(I)$ denotes the running time (in seconds) of a program $\mathbf{M}$ on a problem instance I.

Worst-case Running Time as a Function of Input Size: $T_{\mathbf{M}}(n)$ denotes the **maximum** running time of program $\mathbf{M}$ on instances of size n:

$$T_{\mathbf{M}}(n) = \max\{T_{\mathbf{M}}(I) : \text{Size}(I) = n\}.$$

Average-case Running Time as a Function of Input Size: $T^{avg}_{\mathbf{M}}(n)$ denotes the **average** running time of program $\mathbf{M}$ over all instances of size n:

$$T^{avg}_{\mathbf{M}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I:\text{Size}(I)=n\}} T_{\mathbf{M}}(I).$$

Complexity

Worst-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has **worst-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm A such that $T_{\mathbf{M}}(n) \in \Theta(f(n))$.

Average-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has **average-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm A such that $T^{avg}_{\mathbf{M}}(n) \in \Theta(f(n))$.

Running Time vs Complexity

Running time can only be determined by implementing a program and running it on a specific computer.

Running time is influenced by many factors, including the programming language, processor, operating system, etc.

Complexity (AKA **growth rate**) can be analyzed by high-level mathematical analysis. It is **independent** of the above-mentioned factors affecting running time.

Complexity is a less precise measure than running time since it is asymptotic and it incorporates unspecified constant factors and unspecified lower order terms.

However, if algorithm A has lower complexity than algorithm B, then a program implementing algorithm A will be faster than a program implementing algorithm B for sufficiently large inputs.

Order Notation

O-notation:

$f(n) \in O(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Here the complexity of f is **not higher** than the complexity of g.

Ω-notation:

$f(n) \in \Omega(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Here the complexity of f is **not lower** than the complexity of g.

Θ-notation:

$f(n) \in \Theta(g(n))$ if **there exist** constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Here f and g have the **same complexity**.

Order Notation (cont.)

o-notation:

$f(n) \in o(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Here f has **lower complexity** than g.

ω-notation:

$f(n) \in \omega(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Here f has **higher complexity** than g.

Exercises

1. Let $f(n) = n^2 - 7n - 30$. Prove from first principles that $f(n) \in O(n^2)$.
2. Let $f(n) = n^2 - 7n - 30$. Prove from first principles that $f(n) \in \Omega(n^2)$.
3. Suppose $f(n) = n^2 + n$. Prove from first principles that $f(n) \notin O(n)$.

Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty. \end{cases}$$

Exercises Using the Limit Method

1. Compare the growth rate of the functions $(\ln n)^2$ and $n^{1/2}$.

2. Use the limit method to compare the growth rate of the functions n^2 and $n^2 - 7n - 30$.

Additional Exercises

1. Compare the growth rate of the functions $(3 + (-1)^n)n$ and n.

2. Compare the growth rates of the functions $f(n) = n\,|\sin \pi n/2| + 1$ and $g(n) = \sqrt{n}$.

Relationships between Order Notations

$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

$f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

Algebra of Order Notations

"Maximum" rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Then:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$
$$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$$
$$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$$

"Summation" rules: Suppose I is a finite set. Then

$$O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O(f(i))$$
$$\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))$$
$$\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))$$

Some Common Growth Rates (in increasing order)

polynomial	$\Theta(1)$
	$\Theta(\log n)$
	$\Theta(\sqrt{n})$
	$\Theta(n)$
	$\Theta(n^2)$
	$\Theta(n^c)$
	$\Theta\left(e^{c(\log n)^{1/3}(\log\log n)^{2/3}}\right)$ (number field sieve)
	$\Theta\left(n^{\sqrt{n}\log_2 n}\right)$ (graph isomorphism)
exponential	$\Theta(1.1^n)$
	$\Theta(2^n)$
	$\Theta(e^n)$
	$\Theta(n!)$
	$\Theta(n^n)$

Sequences

Arithmetic sequence with $d > 0$:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

Geometric sequence:

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$

Sequences (cont.)

Arithmetic-geometric sequence:

$$\sum_{i=0}^{n-1}(a+di)r^i = \frac{a}{1-r} - \frac{(a+(n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}$$

provided that $r \neq 1$.

Harmonic sequence:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

More precisely, it is possible to prove that

$$\lim_{n\to\infty} (H_n - \ln n) = \gamma,$$

where $\gamma \approx 0.57721$ is **Euler's constant**.

Logarithm Formulae

1. $\log_b xy = \log_b x + \log_b y$
2. $\log_b x/y = \log_b x - \log_b y$
3. $\log_b 1/x = -\log_b x$
4. $\log_b x^y = y \log_b x$
5. $\log_b a = \frac{1}{\log_a b}$
6. $\log_b a = \frac{\log_c a}{\log_c b}$
7. $a^{\log_b c} = c^{\log_b a}$

Miscellaneous Formulae

$n! \in \Theta\left(n^{n+1/2}e^{-n}\right)$

$\log n! \in \Theta(n \log n)$

Another useful formula is

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6},$$

which implies that

$$\sum_{i=1}^{n} \frac{1}{i^2} \in \Theta(1).$$

A sum of powers of integers when $c \geq 1$:

$$\sum_{i=1}^{n} i^c \in \Theta(n^{c+1}).$$

Two General Strategies for Loop Analysis

Sometimes a O-bound is sufficient. However, we often want a precise Θ-bound. Two general strategies are as follows:

- Use Θ-bounds **throughout the analysis** and thereby obtain a Θ-bound for the complexity of the algorithm.
- Prove a O-bound and a **matching** Ω-bound **separately** to get a Θ-bound. Sometimes this technique is easier because arguments for O-bounds may use simpler upper bounds (and arguments for Ω-bounds may use simpler lower bounds) than arguments for Θ-bounds do.

Techniques for Loop Analysis

Identify **elementary operations** that require constant time (denoted $\Theta(1)$ time).

The complexity of a loop is expressed as the **sum** of the complexities of each iteration of the loop.

Analyze independent loops **separately**, and then **add** the results: use "maximum rules" and simplify whenever possible.

If loops are nested, start with the **innermost loop** and proceed outwards. In general, this kind of analysis requires evaluation of **nested summations**.

Elementary Operations in the Unit Cost Model

For now, we will work in the **unit cost model**, where we assume that arithmetic operations such as $+$, $-$, $\times$ and integer division take time $\Theta(1)$.

This is a reasonable assumption for integers of bounded size (e.g., integers that fit into one work of memory).

If we want to consider the complexity of arithmetic operation on integers of arbitrary size, we need to consider **bit complexity**, where we express the complexity as a function of the length of the integers (as measured in bits).

We will see some examples later, such as **multiprecision multiplication**.

Example of Loop Analysis

Algorithm: *LoopAnalysis1*$(n : integer)$

(1) $sum \leftarrow 0$

(2) **for** $i \leftarrow 1$ **to** n

do { **for** $j \leftarrow 1$ **to** i

do { $sum \leftarrow sum + (i - j)^2$

$sum \leftarrow \lfloor sum/i \rfloor$

(3) **return** (sum)

Θ-bound analysis

(1)	$\Theta(1)$
(2)	Complexity of inner **for** loop: $\Theta(i)$
	Complexity of outer **for** loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
(3)	$\Theta(1)$
total	$\Theta(1) + \Theta(n^2) + \Theta(1) = \Theta(n^2)$

Example of Loop Analysis (cont.)

Proving separate O- and Ω-bounds

We focus on the two nested **for** loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per iteration.

Upper bound:

$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds **match**, the complexity is $\Theta(n^2)$.

Another Example of Loop Analysis

Algorithm: *LoopAnalysis2*$(A : array; n : integer)$

$max \leftarrow 0$
for $i \leftarrow 1$ **to** n
do
  for $j \leftarrow i$ **to** n
  do
    $sum \leftarrow 0$
    for $k \leftarrow i$ **to** j
    do
      $sum \leftarrow sum + A[k]$
      if $sum > max$
      then $max \leftarrow sum$
return (max)

Another Example of Loop Analysis (cont.)

Θ-bound analysis The innermost loop (**for** k) has complexity $\Theta(j-i+1)$. The next loop (**for** j) has complexity

$$\begin{aligned}\sum_{j=i}^{n}\Theta(j-i+1) &= \Theta\left(\sum_{j=i}^{n}(j-i+1)\right)\\ &= \Theta\left(1+2+\cdots+(n-i+1)\right)\\ &= \Theta\left((n-i+1)(n-i+2)\right).\end{aligned}$$

The outer loop (**for** i) has complexity

$$\begin{aligned}\sum_{i=1}^{n}\Theta((n-i+1)(n-i+2)) &= \Theta\left(\sum_{i=1}^{n}(n-i+1)(n-i+2)\right)\\ &= \Theta\left(1\times 2+2\times 3+\cdots+n(n+1)\right)\\ &= \Theta\left(n^3/3+n^2+2n/3\right) \quad \text{from Maple}\\ &= \Theta(n^3).\end{aligned}$$

Another Example of Loop Analysis (cont.)

Proving an Ω-bound

Consider two loop structures:

L_1	L_2
$i = 1, \ldots, n/3$	$i = 1, \ldots, n$
$j = 1 + 2n/3, \ldots, n$	$j = i + 1 \ldots, n$
$k = 1 + n/3, \ldots, 1 + 2n/3$	$k = i \ldots, j$

It is easy to see that $L_1 \subset L_2$, where L_2 is loop structure of the algorithm.

This is because the algorithm examines all triples (i, k, j) with $1 \leq i \leq k \leq j \leq n$.

There are $(n/3)^3 = n^3/27$ iterations in L_1.

Therefore the number of iterations in L_2 is $\Omega(n^3)$.

Yet Another Example of Loop Analysis

Algorithm: *LoopAnalysis3*$(n : integer)$

$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n

do
- $j \leftarrow i$
- **while** $j \geq 1$
  - **do**
    - $sum \leftarrow sum + i/j$
    - $j \leftarrow \left\lfloor \frac{j}{2} \right\rfloor$

return (sum)

Table of Contents

3 **Divide-and-Conquer Algorithms**
- Recurrence Relations
- Master Theorem
- Divide-and-Conquer Design Strategy
- Mergesort
- Divide-and-Conquer Recurrence Relations
- Non-dominated Points
- Closest Pair
- Multiprecision Multiplication
- Matrix Multiplication
- Selection and Median

Recurrence Relations

Suppose $a_1, a_2, \ldots,$ is an infinite sequence of real numbers.

A recurrence relation is a formula that expresses a general term a_n in terms of one or more previous terms $a_1, \ldots, a_{n-1}$.

A recurrence relation will also specify one or more initial values starting at a_1.

Solving a recurrence relation means finding a formula for a_n that does not involve any previous terms $a_1, \ldots, a_{n-1}$.

There are many methods of solving recurrence relations. Two important methods are **guess-and-check** and the **recursion tree method**.

We will make extensive use of the recursion tree method. However, we first take a quick look at the guess-and-check method.

Guess-and-check Method

step 1 Tabulate some values $a_1, a_2, \ldots$ using the recurrence relation.

step 2 Guess that the solution a_n has a specific form, involving undetermined constants.

step 3 Use $a_1, a_2, \ldots$ to determine specific values for the unspecified constants.

step 4 Use induction to prove your guess for a_n is correct.

Example of the Guess-and-check Method

Suppose we have the recurrence $T(n) = T(n-1) + 6n - 5$, $T(0) = 4$.

We compute a few values: $T(1) = 5$, $T(2) = 12$, $T(3) = 25$, $T(4) = 44$.

If we are sufficiently perspicacious, we might guess that $T(n)$ is a **quadratic function**, e.g., $T(n) = an^2 + bn + c$.

Next, we use $T(0) = 4$, $T(1) = 5$, $T(2) = 12$ to compute a, b and c by solving three equations in three unknowns.

We get $a = 3$, $b = -2$, $c = 4$.

Now we can use induction to prove that $T(n) = 3n^2 - 2n + 4$ for all $n \geq 0$.

Another Example

Consider the recurrence

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n \\ T(1) &= 1 \\ T(2) &= 2 \end{aligned}$$

Suppose we tabulate some values of $T(n)$ and then guess that $T(n) \leq cn$ for all $n \geq 1$, for some constant c.

We can use empirical data to guess an appropriate value for c.

However, an alternative approach it to carry out the induction proof in order to determine a value of c that works.

Recursion Tree Method

The following recurrence relation arises in the analysis of *Mergesort*:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \text{ is a power of } 2 \\ d & \text{if } n = 1, \end{cases}$$

where c and d are constants.

We can solve this recurrence relation when n is a power of two, by constructing a **recursion tree**, as follows:

step 1 Start with a one-node tree, say N, having the value $T(n)$.

step 2 Grow two children of N. These children, say N_1 and N_2, have the value $T(n/2)$, and the value of N is replaced by cn.

step 3 Repeat this process recursively, terminating when a node receives the value $T(1) = d$.

step 4 Sum the values on each level of the tree, and then compute the sum of all these sums; the result is $T(n)$.

Solving the Mergesort Recurrence

Let $n = 2^j$.

level	# nodes	value at each node	value of the level
j	1	$c2^j$	$c2^j$
$j-1$	2	$c2^{j-1}$	$c2^j$
$j-2$	2^2	$c2^{j-2}$	$c2^j$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	2^{j-1}	$c2^1$	$c2^j$
0	2^j	d	$d2^j$

Summing the values at all levels of the recursion tree, we have that

$$T(n) = d2^j + cj2^j.$$

Since $n = 2^j$, we have $j = \log_2 n$ and

$$T(n) = dn + cn \log_2 n \in \Theta(n \log n).$$

Another Example

Recall the recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n$. We showed by induction that $T(n) \in O(n)$.

Here, we give an informal justification of this result (not a proof) using the recurrence tree method.

We ignore all "floors", and compute the sum of the all the levels of the tree.

Master Theorem

The **Master Theorem** provides a formula for the solution of many recurrence relations typically encountered in the analysis of algorithms.

The following is a simplified version of the **Master Theorem**:

Theorem 3.1

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y), \tag{1}$$

where n *is a power of* b. *Denote* $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x. \end{cases}$$

Proof of the Master Theorem (simplified version)

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = aT\left(\frac{n}{b}\right) + cn^y, \qquad T(1) = d.$$

Let $n = b^j$.

level	# nodes	value at each node	value of the level
j	1	cn^y	cn^y
$j-1$	a	$c(n/b)^y$	$ca(n/b)^y$
$j-2$	a^2	$c(n/b^2)^y$	$ca^2(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	a^{j-1}	$c(n/b^{j-1})^y$	$ca^{j-1}(n/b^{j-1})^y$
0	a^j	d	da^j

Computing $T(n)$

Summing the values at all levels of the recursion tree, we have that

$$T(n) = da^j + cn^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i.$$

Recall that $b^x = a$ and $n = b^j$. Hence $a^j = (b^x)^j = (b^j)^x = n^x$.

The formula for $T(n)$ is a **geometric sequence** with ratio $r = a/b^y = b^{x-y}$:

$$T(n) = dn^x + cn^y \sum_{i=0}^{j-1} r^i.$$

There are **three cases**, depending on whether $r > 1$, $r = 1$ or $r < 1$.

Complexity of $T(n)$

case	r	y, x	complexity of $T(n)$
heavy leaves	$r > 1$	$y < x$	$T(n) \in \Theta(n^x)$
balanced	$r = 1$	$y = x$	$T(n) \in \Theta(n^x \log n)$
heavy top	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

heavy leaves means that the value of the recursion tree is dominated by the values of the leaf nodes.

balanced means that the values of the levels of the recursion tree are constant (except for the last level).

heavy top means that the value of the recursion tree is dominated by the value of the root node.

Complexity of $T(n)$ (cont.)

Let

$$S = \sum_{i=0}^{j-1} r^i.$$

In case 1, we have $x > y$ so $r > 1$. $S \in \Theta(r^j)$, so $T(n) \in \Theta(n^x + n^y r^j)$. However,

$$r^j = (b^{x-y})^j = (b^j)^{x-y} = n^{x-y}.$$

Therefore

$$T(n) \in \Theta(n^x + n^y n^{x-y}) = \Theta(n^x).$$

In case 2, we have $x = y$ so $r = 1$. $S \in \Theta(j) = \Theta(\log n)$, so

$$T(n) \in \Theta(n^x + n^y \log n) = \Theta(n^x + n^x \log n) = \Theta(n^x \log n).$$

In case 3, we have $x < y$ so $r < 1$. $S \in \Theta(1)$, so

$$T(n) \in \Theta(n^x + n^y) = \Theta(n^y).$$

The complexity does not depend on the initial value d.

Some Examples of Applying the Formulas

1. $T(n) = 2T(n/2) + cn.$
2. $T(n) = 3T(n/2) + cn.$
3. $T(n) = 4T(n/2) + cn.$
4. $T(n) = 2T(n/2) + cn^{3/2}.$

Master Theorem (modified general version)

Theorem 3.2

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where n *is a power of* b. *Denote* $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } f(n) \in O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\ \Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \\ & \text{for some } \epsilon > 0. \end{cases}$$

Some Examples

1. $T(n) = 3T(n/4) + n \log n.$

2. $T(n) = 2T(n/2) + n \log n.$

Solving the Second Recurrence

We can solve the above $T(n) = 2T(n/2) + n \log n$ using the recursion tree method. Assume $T(1) = 1$. Let $n = 2^j$.

level	# nodes	value at each node	value of the level
j	1	$j2^j$	$j2^j$
$j-1$	2	$(j-1)2^{j-1}$	$(j-1)2^j$
$j-2$	2^2	$(j-2)2^{j-2}$	$(j-2)2^j$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	2^{j-1}	2^1	2^j
0	2^j	1	2^j

Summing the values at all levels of the recursion tree, we have that

$$T(n) = 2^j \left(1 + \sum_{i=1}^{j} i\right) = 2^j \left(1 + \frac{j(j+1)}{2}\right).$$

Since $n = 2^j$, we have $j = \log_2 n$ and $T(n) \in \Theta(n(\log n)^2)$.

The Divide-and-Conquer Design Strategy

divide: Given a problem instance I, construct one or more smaller problem instances, denoted $I_1, \ldots, I_a$ (these are called **subproblems**). Usually, we want the size of these subproblems to be small compared to the size of I, e.g., half the size.

conquer: For $1 \leq j \leq a$, solve instance I_j recursively, obtaining solutions $S_1, \ldots, S_a$.

combine: Given $S_1, \ldots, S_a$, use an appropriate **combining** function to find the solution S to the problem instance I, i.e., $S \leftarrow \mathit{Combine}(S_1, \ldots, S_a)$.

Example: Design of Mergesort

Here, a problem instance consists of an array A of n integers, which we want to sort in increasing order. The size of the problem instance is n.

divide: Split A into two subarrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A.

conquer: Run *Mergesort* on A_L and A_R.

combine: After A_L and A_R have been sorted, use a function *Merge* to merge A_L and A_R into a single sorted array. Recall that this can be done in time $\Theta(n)$ with a single pass through A_L and A_R. We simply keep track of the "current" element of A_L and A_R, always copying the smaller one into the sorted array.

Mergesort

Algorithm: *Mergesort*$(A : array; n : integer)$

if $n = 1$

then $S \leftarrow A$

else
$$\begin{cases} n_L \leftarrow \lceil \frac{n}{2} \rceil \\ n_R \leftarrow \lfloor \frac{n}{2} \rfloor \\ A_L \leftarrow [A[1], \ldots, A[n_L]] \\ A_R \leftarrow [A[n_L + 1], \ldots, A[n]] \\ S_L \leftarrow \textit{Mergesort}(A_L, n_L) \\ S_R \leftarrow \textit{Mergesort}(A_R, n_R) \\ S \leftarrow \textit{Merge}(S_L, n_L, S_R, n_R) \end{cases}$$

return (S, n)

Analysis of Mergesort

Let $T(n)$ denote the time to run *Mergesort* on an array of length n.

divide takes time $\Theta(n)$

conquer takes time $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$

combine takes time $\Theta(n)$

Recurrence relation:

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Sloppy and Exact Recurrence Relations

It is simpler to replace the $\Theta(n)$ term by cn, where c is an unspecified constant. The resulting recurrence relation is called the **exact recurrence**.

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

If we then remove the floors and ceilings, we obtain the so-called **sloppy recurrence**:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

The exact and sloppy recurrences are identical when n is a power of two.

Further, the sloppy recurrence makes sense only when n is a power of two.

Solution to the Recurrence

The **Master Theorem** provides the **exact** solution of the recurrence when $n = 2^j$ (it is in fact a **proof** for these values of n).

We can express this solution (for powers of 2) as a function of n, using Θ-notation.

It can be shown that the resulting function of n will in fact yield the **complexity** of the solution of the exact recurrence for **all values** of n.

This derivation of the complexity of $T(n)$ is not a proof, however. If a rigorous mathematical proof is required, then it is necessary to use **induction** along with the **exact recurrence**.

Non-dominated Points

Given two points $(x_1, y_1), (x_2, y_2)$ in the Euclidean plane, we say that (x_1, y_1) **dominates** (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$.

Problem 3.3

Non-dominated Points

Instance: *A set S of n points in the Euclidean plane, say $S = \{S[1], \ldots, S[n]\}$. For simplicity, we will assume that the x-co-ordinates of all these points are distinct, and the the y-co-ordinates of all these points are also distinct.*

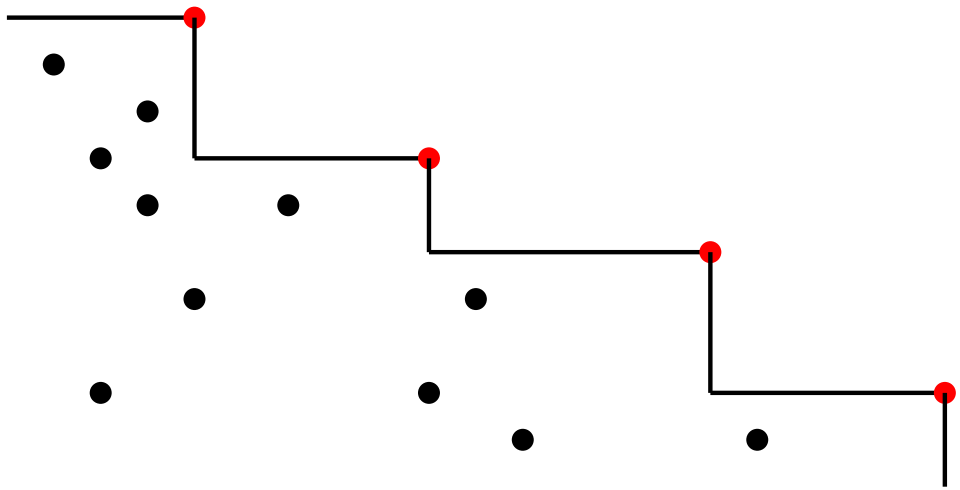
Question: *Find all the non-dominated points in S, i.e., all the points that are not dominated by any other point in S.*

Non-dominated Points has a trivial $\Theta(n^2)$ algorithm to solve it, based on comparing all pairs of points in S. Can we do better?

Staircases

Observe that the non-dominated points form a **staircase** and all the other points are "under" this staircase.

The treads of the staircase are determined by the y-co-ordinates of the non-dominated points. The risers of the staircase are determined by the x-co-ordinates of the non-dominated points. The staircase descends from left to right.



Problem Decomposition

Suppose we **pre-sort** the points in S with respect to their x-co-ordinates. This takes time $\Theta(n \log n)$.

Divide: Let the first $n/2$ points be denoted S_1 and let the last $n/2$ points be denoted S_2.

Conquer: Recursively solve the subproblems defined by the two instances S_1 and S_2.

Combine: Given the non-dominated points in S_1 and the non-dominated points in S_2, how do we find the non-dominated points in S?

Observe that **no point in S_1 dominates a point in S_2**.

Therefore we only need to eliminate the points in S_1 that are dominated by a point in S_2. It turns out that this can be done in time $O(n)$.

The Combine Step

We compute k to be the maximum i such that

$$\text{the } y\text{-co-ordinate of } Q_i \text{ is } > \text{ the } y\text{-co-ordinate of } R_1.$$

This is just a linear search. (We could actually do a binary search, but the overall complexity will not be affected.)

Then, Combine$(\mathsf{ND}(S_1), \mathsf{ND}(S_2)) = (Q_1, \ldots, Q_k, R_1, \ldots, R_m)$.

The x-co-ordinates of the points in Combine$(\mathsf{ND}(S_1), \mathsf{ND}(S_2))$ are in increasing order, so this can be regarded as a post-condition of the algorithm.

Non-dominated Points

Algorithm: *Non-dominated*$(S_1, \ldots, S_n)$

comment: these n points are in increasing order WRT their x-co-ordinates

if $n = 1$ **then return** $(S[1])$

else
$(Q[1], \ldots, Q[\ell]) \leftarrow$ *Non-dominated*$(S[1], \ldots, S[\lfloor n/2 \rfloor])$
$(R[1], \ldots, R[m]) \leftarrow$ *Non-dominated*$(S[\lfloor n/2 \rfloor + 1], \ldots, S[n])$
$i \leftarrow 1$
while $i \leq \ell$ **and** $Q[i].y > R[1].y$
  do $i \leftarrow i + 1$

return $(Q[1], \ldots, Q[i-1], R[1], \ldots, R[m])$

comment: these points are in increasing order WRT their x-co-ordinates

Closest Pair

Problem 3.4

Closest Pair

Instance: *a set Q of n distinct points in the Euclidean plane,*

$$Q = \{Q[1], \ldots, Q[n]\}.$$

Find: *Two distinct points $Q[i] = (x, y), Q[j] = (x', y')$ such that the Euclidean distance*

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.

Closest Pair: Problem Decomposition

Suppose we **pre-sort** the points in Q with respect to their x-coordinates.

Then we can easily find the vertical line that partitions the set of points Q into two sets of size $n/2$: this line has equation $x = Q[m].x$, where $m = n/2$.

Divide: We have two subproblems, consisting of the first $n/2$ points and the last $n/2$ points.

Conquer: Recursively solve the two subproblems.

Combine: Given that we have determined the shortest distance among the first $n/2$ points and the shortest distance among the last $n/2$ points, what additional work is required to determine the overall shortest distance?

Problem Decomposition (cont.)

We will construct the critical strip R of width 2δ consisting of all points whose x-coordinates are within δ of the vertical splitting line, which has equation $x = xmid$, where $xmid = Q[m].x$.

Suppose δ_L is the minimum distance in the left half, δ_R is the minimum distance in the right half. Let $\delta = \min\{\delta_L, \delta_R\}$.

If there is a pair of points having distance $< \delta$, they must be in the critical strip.

Perhaps all the points are in the critical strip, so it will not be efficient to check all pairs of points in the critical strip ($n/2 \times n/2 = n^2/4 \in \Theta(n^2)$).

Key idea: Sort the points in the critical strip WRT y-co-ordinates. This takes time $\Theta(n \log n)$.

It turns out that we only need to compute distances from each point to the next seven points This means that there are at most $7n$ pairs of points to check, which can be done in time $\Theta(n)$.

The Critical Strip

Lemma 3.5

Suppose the points in the critical strip are sorted WRT their y-co-ordinates. Suppose that $R[j]$ and $R[k]$ are two points in the critical strip, where $j < k$, and suppose the distance between $R[j]$ and $R[k]$ is less than δ. Then $k \leq j + 7$.

Proof.

Construct a rectangle $\mathcal{R}$ having width 2δ and height δ, in which the base is the line $y = R[j].y$. Consider $\mathcal{R}$ to be partitioned into eight squares of side $\delta/2$. There is at most one point inside each of these eight squares, one of which is $R[j]$. If $k \geq j + 8$, then $R[k]$ is above $\mathcal{R}$ and the distance between $R[j]$ and $R[k]$ is greater than δ. □

Closest Pair: Solution 1

Algorithm: *ClosestPair1*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else
$$\begin{cases} m \leftarrow \lfloor (\ell + r)/2 \rfloor \\ \delta_L \leftarrow \textit{ClosestPair1}(\ell, m) \\ \delta_R \leftarrow \textit{ClosestPair1}(m+1, r) \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q[m].x) \\ R \leftarrow \textit{SortY}(R) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$$

return (δ)

Selecting Candidates from the Vertical Strip

Algorithm: *SelectCandidates*$(\ell, r, \delta, xmid)$

$j \leftarrow 0$

for $i \leftarrow \ell$ **to** r

do { **if** $|Q[i].x - xmid| \leq \delta$

then { $j \leftarrow j + 1$

$R[j] \leftarrow Q[i]$

return (R)

Checking the Vertical Strip

Algorithm: *CheckStrip*(R, δ)

$t \leftarrow$ *size*(R)
$\delta' \leftarrow \delta$
for $j \leftarrow 1$ **to** $t - 1$
do
  for $k \leftarrow j + 1$ **to** $\min\{t, j + 7\}$
  do
    $x \leftarrow R[j].x$
    $x' \leftarrow R[k].x$
    $y \leftarrow R[j].y$
    $y' \leftarrow R[k].y$
    $\delta' \leftarrow \min\left\{\delta', \sqrt{(x' - x)^2 + (y' - y)^2}\right\}$
return (δ')

An Example to Illustrate the Recursive Calls

Suppose we have eight points, ordered by x-co-ordinates:

$$(2,3),(6,4),(8,9),(10,6),(11,7),(12,2),(14,1),(16,5)$$

We generate two recursive calls with indices $(1,4)$ and $(5,8)$.

$(1,4)$ generates recursive calls with indices $(1,2)$ and $(3,4)$.

$(1,2)$ generates $(1,1)$ and $(2,2)$. $\delta_L = \delta_R = \infty$. Then $\delta = \sqrt{17}$.

$(3,4)$ generates $(3,3)$ and $(4,4)$. $\delta_L = \delta_R = \infty$. Then $\delta = \sqrt{13}$.

$(1,4)$ receives $\delta_L = \sqrt{17}$ and $\delta_R = \sqrt{13}$. Then $\delta = \sqrt{13}$.

$(5,8)$ generates $(5,6)$ and $(7,8)$.

$(5,6)$ generates $(5,5)$ and $(6,6)$. $\delta_L = \delta_R = \infty$. Then $\delta = \sqrt{26}$.

$(7,8)$ generates $(7,7)$ and $(8,8)$. $\delta_L = \delta_R = \infty$. Then $\delta = \sqrt{40}$.

$(5,8)$ receives $\delta_L = \sqrt{26}$ and $\delta_R = \sqrt{20}$. Then $\delta = \sqrt{5}$.

$(1,2)$ receives $\delta_L = \sqrt{13}$ and $\delta_R = \sqrt{5}$. Finally, $\delta = \sqrt{2}$.

An Improvement

To improve the complexity, we eliminate the sorting of the points in critical strip WRT their y-co-ordinates.

The precondition for *ClosestPair2* is that the relevant points in Q, namely $Q[\ell], \ldots, Q[r]$. are sorted WRT their x-co-ordinates.

The postcondition for *ClosestPair2* is that $Q[\ell], \ldots, Q[r]$ are sorted WRT their y-co-ordinates.

This can be accomplished by merging two sublists $Q[\ell], \ldots, Q[m]$ and $Q[m+1], \ldots, Q[r]$ which are recursively sorted WRT their y-co-ordinates (this is identical to the merging step in *MergeSort*).

Closest Pair: Solution 2

Algorithm: *ClosestPair2*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else

$m \leftarrow \lfloor (\ell + r)/2 \rfloor$

$Xmid \leftarrow Q[m].x$

$\delta_L \leftarrow$ *ClosestPair2*(ℓ, m)

comment: $Q[\ell], \ldots, Q[m]$ is sorted WRT y-coordinates

$\delta_R \leftarrow$ *ClosestPair2*$(m + 1, r)$

comment: $Q[m + 1], \ldots, Q[r]$ is sorted WRT y-coordinates

$\delta \leftarrow \min\{\delta_L, \delta_R\}$

Merge(ℓ, m, r)

$R \leftarrow$ *SelectCandidates*$(\ell, r, \delta, Xmid)$

$\delta \leftarrow$ *CheckStrip*(R, δ)

return (δ)

Multiprecision Multiplication

Problem 3.6

Multiprecision Multiplication

Instance: *Two k-bit positive integers, X and Y, having binary representations*

$$X = [X[k-1], ..., X[0]]$$

and

$$Y = [Y[k-1], ..., Y[0]].$$

Question: *Compute the $2k$-bit positive integer $Z = XY$, where*

$$Z = (Z[2k-1], ..., Z[0]).$$

Here, we are interested in the **bit complexity** of algorithms that solve **Multiprecision Multiplication**, which means that the complexity is expressed as a function of k (the size of the problem instance is $2k$ bits).

A Divide-and-Conquer Approach

Assume k is even.

Let X_L be the integer formed by the $k/2$ high-order bits of X and let X_R be the integer formed by the $k/2$ low-order bits of X.

Similarly for Y.

Thus

$$X = 2^{k/2}X_L + X_R \quad \text{and} \quad Y = 2^{k/2}Y_L + Y_R.$$

Therefore, we have

$$XY = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R.$$

Multiplication by a power of 2 is just a left shift.

Not-So-Fast D&C Multiprecision Multiplication

Algorithm: *NotSoFastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else
$$\begin{cases} Z_1 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_R, k/2) \\ Z_4 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_L, k/2) \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 + Z_4, k/2) \end{cases}$$

return (Z)

What is the complexity of this algorithm?

An Improvement

Recall

$$XY = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R.$$

Karatsuba's algorithm reduces the number of subproblems from 4 to 3.

Define

$$\begin{aligned} Z_1 &= X_L Y_L \\ Z_2 &= X_R Y_R \\ Z_3 &= (X_L + X_R)(Y_L + Y_R). \end{aligned}$$

Then

$$X_L Y_R + X_R Y_L = Z_3 - Z_1 - Z_2.$$

Karatsuba Multiplication

Algorithm: *Karatsuba*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else
$$\begin{cases} X_T \leftarrow X_L + X_R \\ Y_T \leftarrow Y_L + Y_R \\ Z_1 \leftarrow \textit{Karatsuba}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{Karatsuba}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{Karatsuba}(X_T, Y_T, k/2), \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 - Z_1 - Z_2, k/2) \end{cases}$$

return (Z)

What is the complexity of this algorithm?

Matrix Multiplication

Problem 3.7

Matrix Multiplication

Instance: *Two n by n matrices, A and B.*

Question: *Compute the n by n matrix product $C = AB$.*

The naive algorithm for **Matrix Multiplication** has complexity $\Theta(n^3)$.

D&C Matrix Multiplication: Problem Decomposition

Let

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

If A, B are n by n matrices, then $a, b, ..., h, r, s, t, u$ are $\frac{n}{2}$ by $\frac{n}{2}$ matrices, where

$$r = ae + bg \qquad s = af + bh$$
$$t = ce + dg \qquad u = cf + dh$$

We require 8 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

What is the complexity of the resulting divide-and-conquer algorithm?

Strassen Matrix Multiplication

Define

$$P_1 = a(f-h) \qquad P_2 = (a+b)h$$
$$P_3 = (c+d)e \qquad P_4 = d(g-e)$$
$$P_5 = (a+d)(e+h) \qquad P_6 = (b-d)(g+h)$$
$$P_7 = (a-c)(e+f).$$

Then, compute

$$r = P_5 + P_4 - P_2 + P_6 \qquad s = P_1 + P_2$$
$$t = P_3 + P_4 \qquad u = P_5 + P_1 - P_3 - P_7.$$

We now require only 7 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

What is the complexity of the resulting divide-and-conquer algorithm?

Selection

Problem 3.8

Selection

Instance: *An array $A[1], \ldots, A[n]$ of distinct integer values, and an integer k, where $1 \leq k \leq n$.*

Find: *The kth smallest integer in the array A.*

The problem **Median** is the special case of **Selection** where $k = \lceil \frac{n}{2} \rceil$.

QuickSelect

Suppose we choose a **pivot** element y in the array A, and we *Restructure* A so that all elements less than y precede y in A, and all elements greater than y occur after y in A. (This is exactly what is done in *Quicksort*, and it takes **linear time**.)

Suppose that $A[posn] = y$ after restructuring. Let A_L be the subarray $A[1], \ldots, A[posn-1]$ and let A_R be the subarray (of size $n - posn$) $A[posn+1], \ldots, A[n]$.

Then the kth smallest element of A is

$$\begin{cases} y & \text{if } k = posn \\ \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\ \text{the } (k - posn)\text{th smallest element of } A_R & \text{if } k > posn. \end{cases}$$

We make (at most) one recursive call at each level of the recursion.

Average-case Analysis of QuickSelect

We say that a pivot is **good** if $posn$ is in the middle half of A, i.e., $n/4 \leq posn \leq 3n/4$.

The probability that a pivot is good is $1/2$.

On average, after **two iterations**, we will encounter a good pivot.

If a pivot is good, then $|A_L| \leq 3n/4$ and $|A_R| \leq 3n/4$.

With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.

Let's consider the average-case recurrence relation: $T(n) = T(3n/4) + \Theta(n)$.

Apply the **Master Theorem** with $a = 1$, $b = 4/3$ and $y = 1$. Here $x = \log_{4/3} 1 = 0 < 1 = y$ so we are in case 3.

This yields $T(n) \in \Theta(n)$ on average.

Achieving $O(n)$ Worst-Case Complexity: A Strategy for Choosing the Pivot

We choose the pivot to be a certain **median-of-medians**:

step 1 Given $n \geq 15$, write $n = 10r + 5 + \theta$, where $r \geq 1$ and $0 \leq \theta \leq 9$.

step 2 Divide A into $2r + 1$ disjoint subarrays of 5 elements. Denote these subarrays by $B_1, \ldots, B_{2r+1}$.

step 3 For $1 \leq i \leq 2r + 1$, find the median of B_i non-recursively, i.e., by brute force, and denote it by m_i.

step 4 Define M to be the array consisting of elements $m_1, \ldots, m_{2r+1}$.

step 5 Find the median y of the array M recursively.

step 6 Use the element y as the pivot for A.

Example

Suppose $|A| = 15$ and we divide A into three groups of size 5:

						median
1	10	5	8	21	$\rightarrow$	8
34	6	7	12	23	$\rightarrow$	12
2	4	30	11	25	$\rightarrow$	11

The median of the three medians is 11. Then we have

$$\begin{aligned} A_L &= 1, 10, 5, 8, 6, 7, 2, 4 \\ A_R &= 21, 34, 12, 23, 30, 25 \end{aligned}$$

Median-of-medians-QuickSelect

Algorithm: *MOM-QuickSelect*(k, n, A)

1. **if** $n \leq 14$ **then** sort A and **return** $(A[k])$
2. write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
3. construct $B_1, \ldots, B_{2r+1}$ (subarrays of A, each of size 5)
4. find medians $m_1, \ldots, m_{2r+1}$ non-recursively
5. $M \leftarrow [m_1, \ldots, m_{2r+1}]$
6. $y \leftarrow$ *MOM-QuickSelect*$(r + 1, 2r + 1, M)$
7. $(A_L, A_R, posn) \leftarrow$ *Restructure*(A, y)
8. **if** $k = posn$ **then return** (y)
9. **else if** $k < posn$ **then return** (*MOM-QuickSelect*$(k, posn - 1, A_L)$)
10. **else return** (*MOM-QuickSelect*$(k - posn, n - posn, A_R)$)

Recursive Calls in Mom-QuickSelect

We claim that the number of elements $> y$, or $< y$, is at most $7n/10$ (roughly).

Consider $n/5$ groups of five numbers, $B_1, \ldots, B_{n/5}$, and let m_i be the median of B_i, for $1 \leq i \leq n/5$.

Let y be the median of the m_i's.

There are $n/10$ j's such that $m_j < y$.

For each such j, there are three elements in B_j that are less than y (namely, m_j and two other elements of B_j).

So there are at least $3n/10$ elements that are less than y and hence there are at most $7n/10$ elements that are greater than y.

Similarly, there are at most $7n/10$ elements that are less than y.

Worst-case Analysis of MOM-QuickSelect

Therefore, the recursive call is to a subarray of size at most $7n/10$ (roughly).

More precisely, the worst-case complexity $T(n)$ of this algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{7n+12}{10} \right\rfloor\right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14. \end{cases}$$

How do we prove that $T(n)$ is $O(n)$?

Table of Contents

4 **Greedy Algorithms**
- Optimization Problems
- Design Strategy
- Interval Selection
- Interval Colouring
- Knapsack
- Coin Changing
- Stable Matching Problem

Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Problem Instance: **Input** for the specified problem.

Problem Constraints: **Requirements** that must be satisfied by any feasible solution.

Feasible Solution: For any problem instance I, *feasible*(I) is the set of all outputs (i.e., solutions) for the instance I that satisfy the given constraints.

Objective Function: A function f : *feasible*$(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of f as being a **profit** or a **cost** function.

Optimal Solution: A feasible solution $X \in$ *feasible*(I) such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).

The Greedy Method

partial solutions

Given a problem instance I, it should be possible to write a feasible solution X as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer n. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a **partial solution** if no constraints are violated.

Note: it may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the **choice set**

$$\mathit{choice}(X) = \{y : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$

The Greedy Method (cont.)

local evaluation criterion

A **local evaluation criterion** is a function g such that, for any partial solution $X = [x_1, \ldots, x_i]$ and any $y \in \mathit{choice}(X)$, $g(x_1, \ldots, x_i, y)$ measures the cost or profit of extending the partial solution X to include y.

extension

Given a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, choose $y \in \mathit{choice}(X)$ so that $g(y)$ is as small (or large) as possible. Update X to be the $(i+1)$-tuple $[x_1, \ldots, x_i, y]$.

greedy algorithm

Starting with the "empty" partial solution, repeatedly extend it until a feasible solution X is constructed. This feasible solution may or may not be optimal.

Features of the Greedy Method

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function g, followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function g).

Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

Interval Selection

Problem 4.1

Interval Selection

Instance: *A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of* **intervals**.

For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where s_i is the **start time** *of interval A_i and f_i is the* **finish time** *of A_i.*

Feasible solution: *A subset $\mathcal{B} \subseteq \mathcal{A}$ of* **pairwise disjoint intervals**.

Find: *A feasible solution of maximum size (i.e., one that maximizes $|\mathcal{B}|$).*

Possible Greedy Strategies for Interval Selection

1. Sort the intervals in increasing order of starting times. At any stage, choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is s_i).
2. Sort the intervals in increasing order of duration. At any stage, choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).
3. Sort the intervals in increasing order of finishing times. At any stage, choose the **earliest finishing** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is f_i).

Does one of these strategies yield a correct greedy algorithm?

A Greedy Algorithm for Interval Selection

Algorithm: *GreedyIntervalSelection*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$

$\mathcal{B} \leftarrow \{A_1\}$

$prev \leftarrow 1$

comment: $prev$ is the index of the last selected interval

for $i \leftarrow 2$ **to** n

do { **if** $s_i \geq f_{prev}$ **then** { $\mathcal{B} \leftarrow \mathcal{B} \cup \{A_i\}$; $prev \leftarrow i$

return $(\mathcal{B})$

Correctness Proof

We give an induction proof.

Let $\mathcal{B}$ be the greedy solution,

$$\mathcal{B} = (A_{i_1}, \ldots, A_{i_k}),$$

where $i_1 < \cdots < i_k$.

Let $\mathcal{O}$ be any optimal solution,

$$\mathcal{O} = (A_{j_1}, \ldots, A_{j_\ell}),$$

where $j_1 < \cdots < j_\ell$.

Observe that $\ell \geq k$ since $\mathcal{O}$ is optimal.

We want to prove that $\ell = k$.

Correctness Proof (cont.)

Lemma 4.2 (Greedy stays ahead)

$f_{i_m} \leq f_{j_m}$ *for* $m = 1, 2, \ldots$.

Proof.

Initial case $m = 1$. We have $f_{i_1} \leq f_{j_1}$ since the greedy algorithm begins by choosing $i_1 = 1$. (A_1 has the earliest finishing time.)
Induction assumption: $f_{i_{m-1}} \leq f_{j_{m-1}}$. Consider A_{i_m} and A_{j_m}. We have

$$s_{j_m} \geq f_{j_{m-1}} \geq f_{i_{m-1}}.$$

A_{i_m} has the earliest finishing time of any interval that starts after $f_{i_{m-1}}$ finishes. Therefore $f_{i_m} \leq f_{j_m}$. □

Correctness Proof (cont.)

Now we complete the proof.

From the Lemma, we have $f_{i_k} \leq f_{j_k}$.

Suppose that $\ell > k$.

$A_{j_{k+1}}$ starts after A_{j_k} finishes, and $f_{i_k} \leq f_{j_k}$.

So $A_{j_{k+1}}$ is feasible WRT the greedy solution, and therefore the greedy solution would not have terminated with A_{i_k}.

This contradiction shows that $\ell = k$.

A Slick Proof

Induction is a standard way to prove correctness of greedy algorithms; however, sometimes shorter "slick" proofs are possible.

Let $F = \{f_{i_1}, \ldots, f_{i_k}\}$ be the finishing times of the intervals in $\mathcal{B}$.

There is no interval in $\mathcal{O}$ that is "between" $f_{i_{m-1}}$ and f_{i_m} for any $m \geq 2$, since A_{i_m} would not be chosen by the greedy algorithm in this case.

As well, there is no interval in in $\mathcal{O}$ that finishes before f_{i_1}, or one that starts after f_{i_k}.

Therefore every interval in $\mathcal{O}$ contains a point in F (or has a point in F as a finishing time).

No two intervals in $\mathcal{O}$ contain the same point in F because the intervals are disjoint.

Hence, there is an injective mapping from $\mathcal{O}$ to F and therefore $|\mathcal{O}| \leq |F|$.

Then we have

$$\ell = |\mathcal{O}| \leq |F| = |\mathcal{B}| = k.$$

Interval Colouring

Problem 4.3

Interval Colouring

Instance: *A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of* **intervals**.

For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where s_i is the **start time** *of interval A_i and f_i is the* **finish time** *of A_i.*

Feasible solution: *A* **c-colouring** *is a mapping $col : \mathcal{A} \to \{1, \ldots, c\}$ that assigns each interval a* **colour** *such that two intervals receiving the same colour are always disjoint.*

Find: *A c-colouring of $\mathcal{A}$ with the minimum number of colours.*

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first $i < n$ intervals using d colours.

We will colour the $(i+1)$st interval with **any permissible colour**. If it cannot be coloured using any of the existing d colours, then we introduce a **new colour** and d is increased by 1.

Question: In **what order** should we consider the intervals?

Consider the following example:

$$\begin{array}{llll} A_1 = [0,3) & A_2 = [8,11) & A_3 = [14,20) & A_4 = [4,9) \\ A_5 = [16,20) & A_6 = [6,13) & A_7 = [10,15) & A_8 = [0,7) \\ A_9 = [12,20) & A_{10} = [0,5) & & \end{array}$$

A Greedy Algorithm for Interval Colouring

Algorithm: *GreedyIntervalColouring*$(\mathcal{A})$

sort the intervals so that $s_1 \leq \cdots \leq s_n$
$d \leftarrow 1$
$colour[1] \leftarrow 1$
$finish[1] \leftarrow f_1$
for $i \leftarrow 2$ **to** n

do
- $flag \leftarrow$ **false**
- $c \leftarrow 1$
- **while** $c \leq d$ **and** (**not** $flag$)
  - **do**
    - **if** $finish[c] \leq s_i$ **then**
      - $colour[i] \leftarrow c$
      - $finish[c] \leftarrow f_i$
      - $flag \leftarrow$ **true**
    - **else** $c \leftarrow c + 1$
- **if not** $flag$ **then**
  - $d \leftarrow d + 1$
  - $colour[i] \leftarrow d$
  - $finish[d] \leftarrow f_i$

return $(d, colour)$

Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a "slick" method—we give the "slick" proof:

Let D denote the number of colours used by the algorithm.

Suppose $A_i = [s_i, f_i)$ is the first interval to receive the last colour, D.

For every colour $c < D$, there is an interval $A_c = [s_c, f_c)$ such that $s_c \leq s_i < f_c$ (i.e., A_c overlaps A_i).

Therefore we have D intervals, all of which contain the point s_i.

These D intervals must all receive different colours, so there is no colouring with fewer than D colours.

Comments and Questions

Excluding the sort, the complexity of the algorithm is $O(nD)$, where D is the value of d returned by the algorithm.

We don't know the value of D ahead of time; all we know is that $1 \leq D \leq n$.

If it turns out that $D \in \Omega(n)$, then the best we can say is that the complexity is $O(n^2)$.

What **inefficiencies** exist in this algorithm?

What **data structure** would allow a more efficient algorithm to be designed?

What would be the complexity of an algorithm making use of an appropriate data structure?

Implementation Details

For each interval, suppose we searched the d existing colours to find if one of them is suitable. This is a linear search.

A modification is to use the colour of the interval having the earliest finishing time among the most recently chosen intervals of each colour.

We can use a priority queue to keep track of these finishing times.

Whenever we colour interval A_i with colour c, we *insert* (f_i, c) into the priority queue (here f_i is the "key").

When we want to want to colour the next interval A_i, we look at the minimum key f in the priority queue. If $f \leq s_i$, then we do a *deletemin* operation, yielding the pair (f, c) and we use colour c for interval A_i. If $f > s_i$, we introduce a new colour.

Note that each interval is inserted once and deleted once from the priority queue. Therefore, the complexity of this approach is $O(n \log D)$. Since $D \leq n$, it is $O(n \log n)$. (The initial sort is also $O(n \log n)$.)

Knapsack Problems

Problem 4.4

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An* n*-tuple* $X = [x_1, \ldots, x_n]$ *where* $\sum_{i=1}^{n} w_i x_i \leq M$.

In the **0-1 Knapsack** *problem (often denoted just as* **Knapsack***), we require that* $x_i \in \{0, 1\}$, $1 \leq i \leq n$.

In the **Rational Knapsack** *problem, we require that* $x_i \in \mathbb{Q}$ *and* $0 \leq x_i \leq 1$, $1 \leq i \leq n$.

Find: *A feasible solution* X *that maximizes* $\sum_{i=1}^{n} p_i x_i$.

Possible Greedy Strategies for Knapsack Problems

1. Consider the items in decreasing order of **profit** (i.e., the local evaluation criterion is p_i).
2. Consider the items in increasing order of **weight** (i.e., the local evaluation criterion is w_i).
3. Consider the items in decreasing order of **profit divided by weight** (i.e., the local evaluation criterion is p_i/w_i).

Does one of these strategies yield a correct greedy algorithm for the **Rational Knapsack** problem?

A Greedy Algorithm for Rational Knapsack

Algorithm: *GreedyRationalKnapsack*$(P, W : array; M : integer)$

sort the items so that $p_1/w_1 \geq \cdots \geq p_n/w_n$

$X \leftarrow [0, \ldots, 0]$

$i \leftarrow 1$

$CurW \leftarrow 0$

while $(CurW < M)$ **and** $(i \leq n)$

do
- **if** $CurW + w_i \leq M$
  - **then**
    - $x_i \leftarrow 1$
    - $CurW \leftarrow CurW + w_i$
    - $i \leftarrow i + 1$
  - **else**
    - $x_i \leftarrow (M - CurW)/w_i$
    - $CurW := M$

return (X)

Correctness Proof

For simplicity, assume that the profit / weight ratios are all distinct, so

$$\frac{p_1}{w_1} > \frac{p_2}{w_2} > \cdots > \frac{p_n}{w_n}.$$

Suppose the greedy solution is $X = (x_1, \ldots, x_n)$ and the optimal solution is $Y = (y_1, \ldots, y_n)$.

We will prove that $X = Y$, i.e., $x_j = y_j$ for $j = 1, \ldots, n$. Therefore there is a unique optimal solution and it is equal to the greedy solution.

Suppose $X \neq Y$.

Pick the smallest integer j such that $x_j \neq y_j$.

It is impossible that $x_j < y_j$, so we have $x_j > y_j$.

There exists an index $k > j$ such that $y_k > 0$ (otherwise Y is not optimal).

Correctness Proof (cont.)

Let $\delta = \min\{w_k y_k, w_j(x_j - y_j)\}$; note that $\delta > 0$.

Define

$$y'_j = y_j + \frac{\delta}{w_j} \quad \text{and} \quad y'_k = y_k - \frac{\delta}{w_k}.$$

Then let Y' be Y with y_j and y_k updated to y'_j and y'_k, respectively.

The idea is to show that

1. Y' is feasible, and
2. profit$(Y') >$ profit(Y).

This contradicts the optimality of Y and proves that $X = Y$.

Correctness Proof (cont.)

To show Y' is feasible, show that $y'_k \geq 0$, $y'_j \leq 1$ and weight$(Y') \leq M$.

First, we have

$$y'_k = y_k - \frac{\delta}{w_k} \geq y_k - \frac{w_k y_k}{w_k} = 0.$$

Second,

$$y'_j = y_j + \frac{\delta}{w_j} \leq y_j + \frac{w_j(x_j - y_j)}{w_j} = x_j \leq 1.$$

Third,

$$\text{weight}(Y') = \text{weight}(Y) + \frac{\delta}{w_j} w_j - \frac{\delta}{w_k} w_k = \text{weight}(Y) \leq M.$$

Finally, we compute

$$\text{profit}(Y') = \text{profit}(Y) + \frac{\delta p_j}{w_j} - \frac{\delta p_k}{w_k} = \text{profit}(Y) + \delta \left(\frac{p_j}{w_j} - \frac{p_k}{w_k} \right) > \text{profit}(Y),$$

since $\delta > 0$ and $p_j/w_j > p_k/w_k$.

Coin Changing

Problem 4.5

Coin Changing

Instance: *A list of* **coin denominations**, $d_1, d_2, \ldots, d_n$, *and a positive integer* T, *which is called the* **target sum**.

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$, *such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

In the **Coin Changing** problem, a_i denotes the number of coins of denomination d_i that are used, for $i = 1, \ldots, n$.

The total value of all the chosen coins must be exactly equal to T. We want to **minimize** the number of coins used, which is denoted by N.

A Greedy Algorithm for Coin Changing

Algorithm: *GreedyCoinChanging*$(D : array; T : integer)$

comment: $D = [d_1, \ldots, d_n]$

sort the coins so that $d_1 > \cdots > d_n$

$N \leftarrow 0$

for $i \leftarrow 1$ **to** n

do $\begin{cases} a_i \leftarrow \left\lfloor \frac{T}{d_i} \right\rfloor \\ T \leftarrow T - a_i d_i \\ N \leftarrow N + a_i \end{cases}$

if $T > 0$

then return $(fail)$

else return $([a_1, \ldots, a_n], N)$

Proof of Optimality for $D = [100, 25, 10, 5, 1]$

We will prove that the greedy algorithm always finds an optimal solution for coin denominations $D = [100, 25, 10, 5, 1]$.

We will make use of the following properties of any optimal solution:

(1) the number of pennies is at most 4 (replace five pennies by a nickel)

(2) the number of nickels is at most 1 (replace two nickels by a dime)

(3) the number of quarters is at most 3 (replace four quarters by a loonie), and

(4) the number of nickels + the number of dimes is at most 2 (replace three dimes by a quarter and a nickel; replace two dimes and a nickel by a quarter; the number of nickels is at most one).

The proof is by induction on T. As (trivial) base cases, we can take $T = 1, 2, 3, 4$.

Proof of Optimality (cont.)

Suppose $5 \leq T < 10$. First, assume there is no nickel in the optimal solution. Then the optimal solution consists only of pennies, so $T \leq 4$ (property (1)); contradiction. Therefore the optimal solution contains at least one nickel. Clearly the greedy solution contains at least one nickel. By induction, the greedy solution for $T - 5$ is optimal. Therefore the greedy solution for T is also optimal.

Suppose $10 \leq T < 25$. First, assume there is no dime in the optimal solution. Then the optimal solution contains only nickels and pennies, so $T \leq 5 + 4 = 9$ (property (2)); contradiction. Therefore the optimal solution contains at least one dime. Clearly the greedy solution contains at least one dime. By induction, the greedy solution for $T - 10$ is optimal. Therefore the greedy solution for T is also optimal.

$25 \leq T < 100$. Exercise.

$100 \leq T$. Exercise.

The Stable Matching Problem

Problem 4.6

Stable Matching

Instance: *Two sets of size n say $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_n]$. Each x_i has a* **preference ranking** *of the elements in Y, and each y_i has a preference ranking of the elements in X. pref$(x_i, j) = y_k$ if y_k is the j-th favourite element of Y of x_i; and pref$(y_i, j) = x_k$ if x_k is the j-th favourite element of X of y_i.*

Find: *A* **matching** *of the sets X and Y such that there does not exist a pair (x_i, y_j) which is not in the matching, but where x_i and y_j prefer each other to their existing matches. A matching with this this property is called a* **stable matching**.

Overview of the Gale-Shapley Algorithm

Elements of X propose to elements of Y.

If y_j accepts a proposal from x_i, then the pair $\{x_i, y_j\}$ is **matched**.

An unmatched y_j must accept a proposal from any x_i.

If $\{x_i, y_j\}$ is a matched pair, and y_j subsequently receives a proposal from x_k, where y_j prefers x_k to x_i, then y_j accepts and the pair $\{x_i, y_j\}$ is replaced by $\{x_k, y_j\}$.

If $\{x_i, y_j\}$ is a matched pair, and y_j subsequently receives a proposal from x_k, where y_j prefers x_i to x_k, then y_j rejects and nothing changes.

A matched y_j never becomes unmatched.

An x_i might make a number of proposals (up to n); the order of the proposals is determined by x_i's preference list.

Gale-Shapley Algorithm

Algorithm: *Gale-Shapley*$(X, Y, pref)$

$Match \leftarrow \emptyset$

while there exists an unmatched x_i

do
- let y_j be the next element in x_i's preference list
- **if** y_j is not matched
  - **then** $Match \leftarrow Match \cup \{x_i, y_j\}$
  - **else**
    - suppose $\{x_k, y_j\} \in Match$
    - **if** y_j prefers x_i to x_k
      - **then**
        - $Match \leftarrow Match \backslash \{x_k, y_j\} \cup \{x_i, y_j\}$
        - comment: x_k is now unmatched

return $(Match)$

Example

Suppose we have the following preference lists:

$$x_1 : y_2 > y_3 > y_1 \qquad y_1 : x_1 > x_2 > x_3$$
$$x_2 : y_1 > y_3 > y_2 \qquad y_2 : x_2 > x_3 > x_1$$
$$x_3 : y_1 > y_2 > y_3 \qquad y_3 : x_3 > x_2 > x_1$$

The *Gale-Shapley algorithm* could be executed as follows:

proposal	result	*Match*
x_1 proposes to y_2	y_2 accepts	$\{x_1, y_2\}$
x_2 proposes to y_1	y_1 accepts	$\{x_1, y_2\}, \{x_2, y_1\}$
x_3 proposes to y_1	y_1 rejects	
x_3 proposes to y_2	y_2 accepts	$\{x_3, y_2\}, \{x_2, y_1\}$
x_1 proposes to y_3	y_3 accepts	$\{x_3, y_2\}, \{x_2, y_1\}, \{x_1, y_3\}$

Another Example

Suppose we have the following preference lists:

$$x_1 : y_1 > y_2 > y_3 > y_4$$
$$x_2 : y_2 > y_3 > y_1 > y_4$$
$$x_3 : y_3 > y_1 > y_2 > y_4$$
$$x_4 : y_1 > y_2 > y_3 > y_4$$

$$y_1 : x_2 > x_3 > x_4 > x_1$$
$$y_2 : x_3 > x_4 > x_1 > x_2$$
$$y_3 : x_4 > x_1 > x_2 > x_3$$
$$y_4 : x_1 > x_2 > x_3 > x_4$$

Exercise: Show the execution of the *Gale-Shapley algorithm*.

Proof of Correctness

First we need to show that the algorithm always terminates, i.e., it is impossible that an unmatched x_i has proposed to every y_j.

Proof: Once an element of Y is matched, they are never unmatched. If x_i has proposed to every y_j, then every y_j is matched. But then every element of X is matched, which is a contradiction.

We now prove that the algorithm terminates with a stable matching. Suppose there is an instability: x_i is matched with y_j, x_k is matched with y_ℓ, x_i prefers y_ℓ to y_j and y_ℓ prefers x_i to x_k. Observe that x_i proposed to y_ℓ before proposing to y_j.

There three cases to consider:

(1) y_ℓ rejected x_i's proposal.

(2) y_ℓ accepted x_i's proposal, but later accepted another proposal.

(3) y_ℓ accepted x_i's proposal, and did not accept any subsequent proposal.

Proof of Correctness (cont.)

(1) y_ℓ rejected x_i's proposal. This could happen only if y_ℓ was already matched with someone they preferred to x_i. But y_ℓ ended up matched with someone they liked less than x_i. We conclude that y_ℓ did not reject a proposal by x_i.

(2) y_ℓ accepted x_i's proposal, but later accepted another proposal. This could happen only if y_ℓ later received a proposal from someone they preferred to x_i. But y_ℓ ended up matched with someone they liked less than x_i, so this also did not happen.

(3) y_ℓ accepted x_i's proposal, and did not accept any subsequent proposal. In this case, y_ℓ would have ended up matched to x_i, which did not happen.

Complexity

It is obvious that the number of iterations is at most n^2 since every x_i proposes at most once to every y_j.

It is possible to prove the stronger result that the maximum number of iterations is $n^2 - n + 1$.

The average number of iterations is $\Theta(n \log n)$ (but we will not prove this).

Is there an efficient way to identify an unmatched x_i at any point in the algorithm?

What data structure would be helpful in doing this?

What can we then say about the complexity of the algorithm?

Additional Comments

All executions of the *Gale-Shapley algorithm* result in the same matching.

This matching can be characterized as follows: For every x_i, define *best*$(x_i) = y_j$ if there exists at least one stable matching in which x_i is paired with y_j, and there is no stable matching in which x_i is paired with a y_k that is preferred to y_j.

That is, *best*(x_i) represents the "optimal outcome" for x_i given that the result is required to be a stable matching.

It can be shown that the matching resulting from the *Gale-Shapley algorithm* is

$$\mathcal{M} = \{\{x_i, \textit{best}(x_i)\} : 1 \leq i \leq n\}.$$

$\mathcal{M}$ can also be characterized as $\mathcal{M} = \{\{y_j, \textit{worst}(y_j)\} : 1 \leq j \leq n\}$. That is, every y_j receives their "worst possible" match.

The algorithm is completely biased in favour of the proposers!

Table of Contents

5 **Dynamic Programming Algorithms**
- Fibonacci Numbers
- Design Strategy
- 0-1 Knapsack
- Coin Changing
- Longest Common Subsequence
- Minimum Length Triangulation
- Memoization

Computing Fibonacci Numbers Inefficiently

Algorithm: *BadFib*(n)

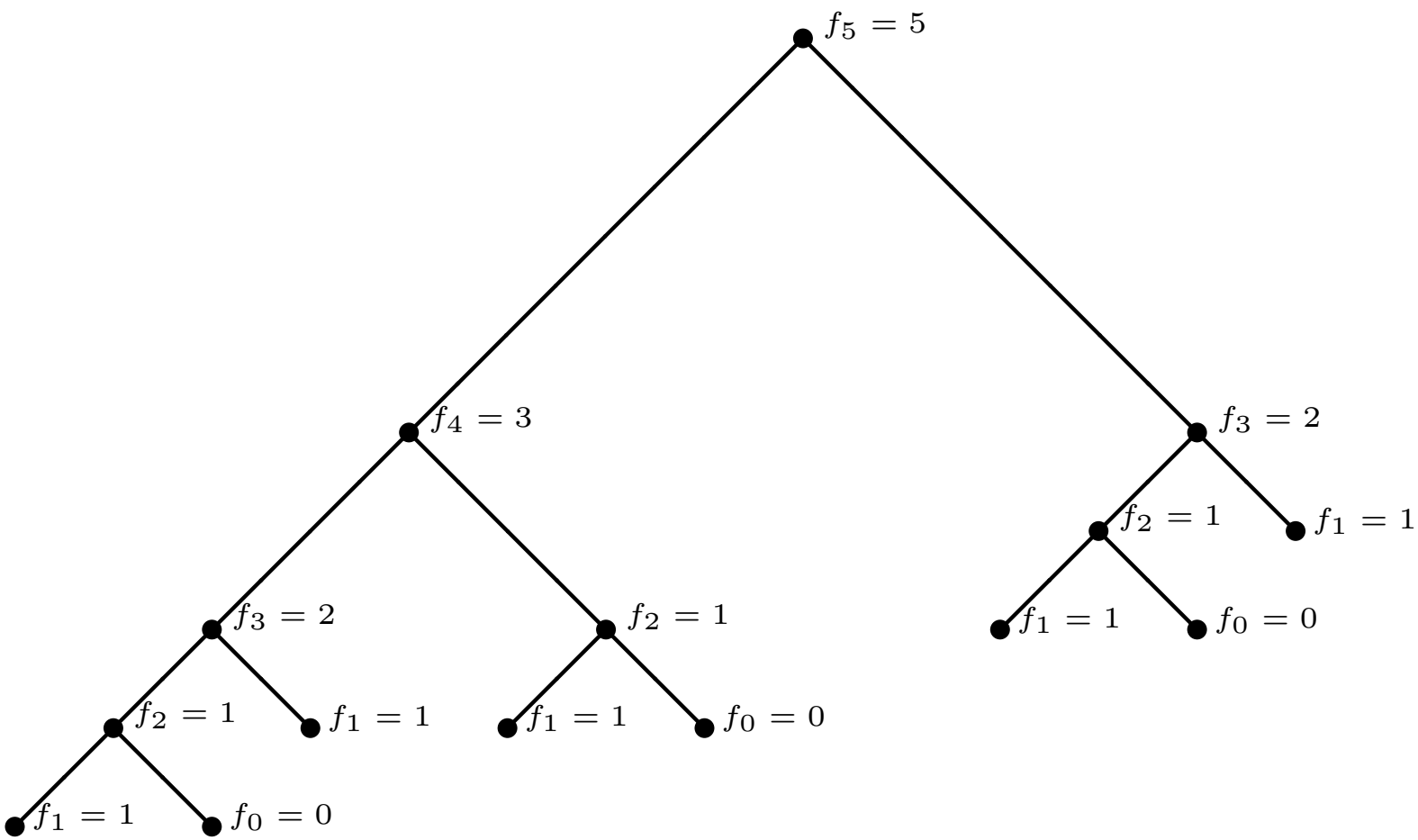
if $n = 0$ **then** $f \leftarrow 0$

else if $n = 1$ **then** $f \leftarrow 1$

else
$$\begin{cases} f_1 \leftarrow \textit{BadFib}(n-1) \\ f_2 \leftarrow \textit{BadFib}(n-2) \\ f \leftarrow f_1 + f_2 \end{cases}$$

return (f);

The Recursion Tree to Evaluate f_5:



Complexity of the Algorithm

The recurrence tree has f_n leaf nodes with the value 1 and f_{n-1} leaf nodes with the value 0. So there are a total of f_{n+1} leaf nodes.

The number of interior nodes is $f_{n+1} - 1$.

In the unit cost model, the complexity of computing f_n is $\Theta(f_{n+1})$.

How quickly does f_n grow? Let $\phi = (1 + \sqrt{5})/2$; then

$$f_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor.$$

Therefore $f_n \in \Theta(\phi^n)$ and hence we also have $f_{n+1} \in \Theta(\phi^n)$.

The value $\phi \approx 1.6$ is the **golden ratio**.

The time to compute f_n is exponential in n.

Computing Fibonacci Numbers More Efficiently

Algorithm: *BetterFib*(n)

$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do $f[i] \leftarrow f[i-1] + f[i-2]$
return $(f[n])$

Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure

Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain **subproblems** of I.

Define Subproblems

Define a set of subproblems $\mathcal{S}(I)$ of the instance I, the solution of which enables the optimal solution of I to be computed. I will be the last or largest instance in the set $\mathcal{S}(I)$.

Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation

Derive a **recurrence relation** on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. Compute these solutions using the recurrence relation in a **bottom-up** fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to I.

0-1 Knapsack

Problem 5.1

0-1 Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An* n*-tuple* $X = [x_1, \ldots, x_n]$, *where* $x_i \in \{0, 1\}$ *for* $1 \leq i \leq n$, *and* $\sum_{i=1}^{n} w_i x_i \leq M$.

Find: *A feasible solution* X *that maximizes* $\sum_{i=1}^{n} p_i x_i$.

Developing a Dynamic Programming Algorithm for 0-1 Knapsack

Optimal Structure:

- Suppose $X = [x_1, \ldots, x_n]$ is an optimal solution to an instance I.
- If $x_n = 0$, then $X' = [x_1, \ldots, x_{n-1}]$ is the optimal solution to the instance (subproblem) with profits $p_1, \ldots, p_{n-1}$, weights $w_1, \ldots, w_{n-1}$ and capacity M.
- If $x_n = 1$, then X' is the optimal solution to the instance (subproblem) with profits $p_1, \ldots, p_{n-1}$, weights $w_1, \ldots, w_{n-1}$ and capacity $M - w_n$.

Subproblems:

- If we apply the above analysis recursively, we consider subproblems consisting of the first i objects (having profits $p_1, \ldots, p_i$ and weights $w_1, \ldots, w_i$) and capacity m, for all $1 \leq i \leq n$ and all $0 \leq m \leq M$.
- Let $P[i, m]$ denote the optimal profit for this subproblem. Then $P[n, M]$ is the final answer we are looking for.

Developing a Dynamic Programming Algorithm (cont.)

Recurrence Relation:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$

Compute Optimal Solutions:

- We fill in the rows one at a time, beginning with row 1.
- We fill in each row of the table from left to right.
- For the last row, we only need to compute the last value $P[n, M]$.

A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M)$

for $m \leftarrow 0$ **to** M

do { **if** $m \geq w_1$
  then $P[1, m] \leftarrow p_1$
  else $P[1, m] \leftarrow 0$

for $i \leftarrow 2$ **to** n

do { **for** $m \leftarrow 0$ **to** M

do { **if** $m < w_i$
  then $P[i, m] \leftarrow P[i-1, m]$
  else $P[i, m] \leftarrow \max\{P[i-1, m-w_i] + p_i, P[i-1, m]\}$

return $(P[n, M])$;

Example

Suppose we have profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30.

The following table is computed:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	18

For example,

$$P[3, 16] = \max\{P[2, 16], P[2, 11] + 3\} = \max\{3, 3 + 3\} = 6.$$

Computing the Optimal Knapsack X

The optimal solution is computed by tracing back through the table.

For the previous example, consisting of profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30, the optimal solution is $[1, 1, 0, 1, 0, 1]$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	18

Computing the Optimal Knapsack X

Algorithm: *ComputeOptimalKnapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M, P)$

$m \leftarrow M$
$p \leftarrow P[n, M]$
for $i \leftarrow n$ **downto** 2
do
- **if** $p = P[i-1, m]$
  - **then** $x_i \leftarrow 0$
  - **else**
    - $x_i \leftarrow 1$
    - $p \leftarrow p - p_i$
    - $m \leftarrow m - w_i$

if $p = 0$
then $x_1 \leftarrow 0$
else $x_1 \leftarrow 1$
return (X);

Complexity of the Algorithm

Suppose we assume the unit cost model, so additions / subtractions take time $O(1)$.

The complexity to initialize and construct the table is $\Theta(nM)$.

Is this a polynomial-time algorithm, as a function of the size of the problem instance?

We have

$$\text{size}(I) = \log_2 M + \sum_{i=1}^{n} \log_2 w_i + \sum_{i=1}^{n} \log_2 p_i.$$

Note in particular that M is exponentially large compared to $\log_2 M$. So constructing the table is not a polynomial-time algorithm, even in the unit cost model.

What would the complexity of a recursive algorithm be?

Coin Changing

Problem 5.2

Coin Changing

Instance: *A list of* **coin denominations**, $1 = d_1, d_2, \ldots, d_n$, *and a positive integer* T, *which is called the* **target sum**.

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$, *such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

What subproblems should be considered?

What table of values should we fill in?

What is the complexity of the algorithm?

How do we compute the optimal set of coins (in addition to the number of coins)?

A Dynamic Programming Algorithm for Coin Changing

Algorithm: *Coin Changing*$(d_1, \ldots, d_n, T)$

comment: $d_1 = 1$

for $t \leftarrow 0$ **to** T

do $\begin{cases} N[1,t] \leftarrow t \\ A[1,t] \leftarrow t \end{cases}$

for $i \leftarrow 2$ **to** n

do $\begin{cases} \textbf{for } t \leftarrow 0 \textbf{ to } T \\ \quad \textbf{do} \begin{cases} N[i,t] \leftarrow N[i-1,t] \\ A[i,t] \leftarrow 0 \\ \textbf{for } j \leftarrow 1 \textbf{ to } \lfloor (t/d_i) \rfloor \\ \quad \textbf{do} \begin{cases} \textbf{if } j + N[i-1, t-jd_i] < N[i,t] \\ \quad \textbf{then} \begin{cases} N[i,t] \leftarrow j + N[i-1,t-jd_i] \\ A[i,t] \leftarrow j \end{cases} \end{cases} \end{cases} \end{cases}$

return $(N[n,T])$

Computing the Optimal Set of Coins

We trace back through the table to compute the optimal set of coins.

There are two possible approaches:

1. Recompute the relevant table entries $N[i, t]$ during the traceback
2. Store relevant extra information, while the table $N[i, t]$ is being constructed, in another table $A[i, t]$.

Suppose we follow the second approach.

The $A[i, t]$ values make it easy to determine number of coins of each denomination in the optimal solution $N[i, T]$.

This is kind of similar to 0-1 Knapsack.

Longest Common Subsequence

Problem 5.3

Longest Common Subsequence

Instance: *Two sequences* $X = (x_1, \ldots, x_m)$ *and* $Y = (y_1, \ldots, y_n)$ *over some finite alphabet* Γ.

Find: *A maximum length sequence* Z *that is a subsequence of both* X *and* Y.

$Z = (z_1, \ldots, z_\ell)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, Z is a subsequence of Y if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Computing the Length of the LCS of X and Y

Consider $X' = (x_1, \ldots, x_{m-1})$ and $Y' = (y_1, \ldots, y_{n-1})$.

1. If $x_m = y_n$, then $\text{LCS}(X, Y) = 1 + \text{LCS}(X', Y')$ (the LCS ends with $x_m = y_n$).
2. If $x_m \neq y_n$, then $\text{LCS}(X, Y) = \max\{\text{LCS}(X, Y'), \text{LCS}(X', Y)\}$.

We consider subproblems consisting of all possible **prefixes** of X and Y.

Let $c[i, j]$ denote the length of the LCS of $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$.

If $i = 0$ or $j = 0$, then we are considering the "empty prefix" of X or Y (respectively).

The optimal solution to the original problem instance is $c[m, n]$.

We have the following recurrence relation:

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j \\ 0 & \text{if } i = 0 \text{ or } j = 0. \end{cases}$$

Computing the Length of the LCS of X and Y

Algorithm: *LCS1*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  do { **for** $j \leftarrow 1$ **to** n
    do { **if** $x_i = y_j$
      then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
      else $c[i, j] \leftarrow \max\{c[i, j - 1], c[i - 1, j]\}$
return $(c[m, n])$;

Finding the LCS of X and Y

Algorithm: *LCS2*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m **do** $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ **to** n **do** $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ **to** m

do
- **for** $j \leftarrow 1$ **to** n
- **do**
  - **if** $x_i = y_j$
    - **then**
      - $c[i, j] \leftarrow c[i-1, j-1] + 1$
      - $\pi[i, j] \leftarrow \mathbf{UL}$
  - **else if** $c[i, j-1] > c[i-1, j]$
    - **then**
      - $c[i, j] \leftarrow c[i, j-1]$
      - $\pi[i, j] \leftarrow \mathbf{L}$
  - **else**
    - $c[i, j] \leftarrow c[i-1, j]$
    - $\pi[i, j] \leftarrow \mathbf{U}$

return (c, π);

Finding the LCS

Algorithm: *FindLCS*(c, π, v)

$seq \leftarrow ()$
$i \leftarrow m$
$j \leftarrow n$
while $\min\{i, j\} > 0$
do
- **if** $\pi[i, j] = \mathbf{UL}$
  - **then**
    - $seq \leftarrow x_i \parallel seq$
    - $i \leftarrow i - 1$
    - $j \leftarrow j - 1$
- **else if** $\pi[i, j] = \mathbf{L}$ **then** $j \leftarrow j - 1$
- **else** $i \leftarrow i - 1$

return (seq)

LCS Example

Suppose $X =$ gdvegta and $Y =$ gvcekst.

		X		g	d	v	e	g	t	a
Y		$i = 0$		1	2	3	4	5	6	7
	$j = 0$	0	0	0	0	0	0	0	0	0
g	1	0	↖ 1	← 1	← 1	← 1	↖ 1	← 1	← 1	
v	2	0	↑ 1	↑ 1	↖ 2	← 2	← 2	← 2	← 2	
c	3	0	↑ 1	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	
e	4	0	↑ 1	↑ 1	↑ 2	↖ 3	← 3	← 3	← 3	
k	5	0	↑ 1	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	
s	6	0	↑ 1	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	
t	7	0	↑ 1	↑ 1	↑ 2	↑ 3	↑ 3	↖ 4	← 4	

Minimum Length Triangulation

Problem 5.4

Minimum Length Triangulation v1

Instance: *n points $q_1, \ldots, q_n$ in the Euclidean plane that form a convex n-gon P.*

Find: *A triangulation of P such that the sum S_c of the lengths of the $n-3$ chords is minimized.*

Problem 5.5

Minimum Length Triangulation v2

Instance: *n points $q_1, \ldots, q_n$ in the Euclidean plane that form a convex n-gon P.*

Find: *A triangulation of P such that the sum S_p of the perimeters of the $n-2$ triangles is minimized.*

Let L denote the perimeter of P. Then we have that $S_p = L + 2S_c$.
Hence the two versions have the same optimal solutions.

Problem Decomposition

We consider version 2 of the problem.

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

For a given k, we have:

1. the triangle $q_1 q_k q_n$,
2. the polygon with vertices $q_1, \ldots, q_k$,
3. the polygon with vertices $q_k, \ldots, q_n$.

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).

Recurrence Relation

For $1 \leq i < j \leq n$, let $S[i, j]$ denote the optimal solution to the subproblem consisting of the polygon having vertices $q_i, \ldots, q_j$.

Let $\Delta(q_i, q_k, q_j)$ denote the perimeter of the triangle having vertices q_i, q_k, q_j.

The we have the recurrence relation

$$S[i, j] = \min\{\Delta(q_i, q_k, q_j) + S[i, k] + S[k, j] : i < k < j\}.$$

The base cases are given by

$$S[i, i + 1] = 0$$

for all i.

We compute all $S[i, j]$ with $j - i = c$, for $c = 2, 3, \ldots, n - 1$.

Memoization

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal.

Memoization is a recursive algorithm based on same recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.

Example: Computing the Fibonacci Numbers

Algorithm: *MemoFib*(n)

procedure *RecFib*(n)

if $n = 0$ **then** $f \leftarrow 0$

else if $n = 1$ **then** $f \leftarrow 1$

else if $M[n] \neq -1$ **then** $f \leftarrow M[n]$

else
$$\begin{cases} f_1 \leftarrow \textit{RecFib}(n-1) \\ f_2 \leftarrow \textit{RecFib}(n-2) \\ f \leftarrow f_1 + f_2 \\ M[n] \leftarrow f \end{cases}$$

return (f);

main

for $i \leftarrow 2$ **to** n

do $M[i] \leftarrow -1$

return $(\textit{RecFib}(n))$

Complexity

Memoization reduces the size of the recursion tree to $\Theta(n)$.

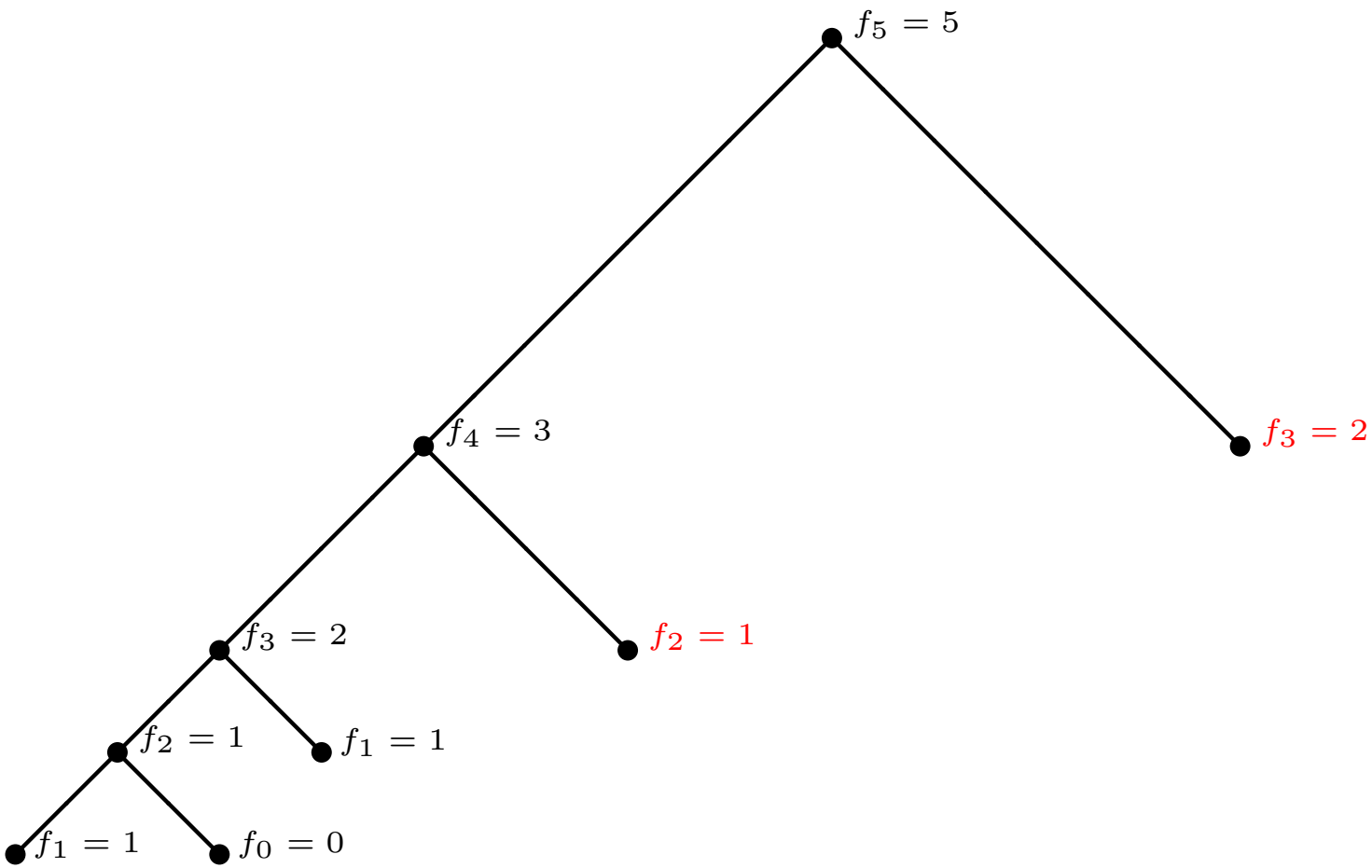


Table of Contents

6 **Graph Algorithms**
- Graphs
- Breadth-first Search
- Depth-first Search
- Topological Ordering
- Strongly Connected Components
- Minimum Spanning Trees
- Single Source Shortest Paths
- All-Pairs Shortest Paths

Graphs and Digraphs

A **graph** is a pair $G = (V, E)$. V is a set whose elements are called **vertices** and E is a set whose elements are called **edges**. Each edge joins two distinct vertices. An edge can be represented as a set of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as uv or vu.

We often denote the number of vertices by n and the number of edges by m. Clearly $m \leq \binom{n}{2}$.

A **directed graph** or **digraph** is also a pair $G = (V, E)$. The elements of E are called **directed edges** or **arcs** in a digraph. Each arc joins two vertices, and an arc can be represented as a ordered pair, e.g., (u, v). The arc (u, v) is directed from u (the **tail**) to v (the **head**), and we allow $u = v$.

If we denote the number of vertices by n and the number of arcs by m, then $m \leq n^2$.

Data Structures for Graphs: Adjacency Matrices

There are two main data structures to represent graphs: an **adjacency matrix** and a set of **adjacency lists**.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The **adjacency matrix** of G is an n by n matrix $A = (a_{u,v})$, which is indexed by V, such that

$$a_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

There are exactly $2m$ entries of A equal to 1.

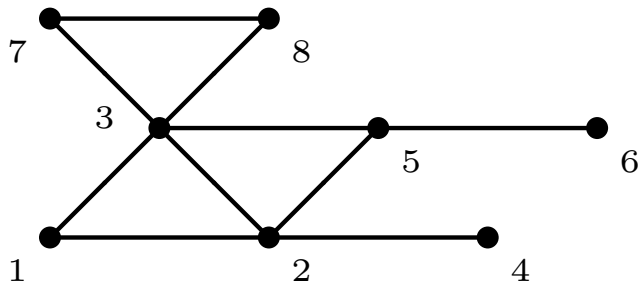
If G is a digraph, then

$$a_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For a digraph, there are exactly m entries of A equal to 1.

Example

$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{12, 13, 23, 24, 25, 35, 37, 38, 56, 78\}$.



The adjacency matrix is

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Data Structures for Graphs: Adjacency Lists

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An **adjacency list representation** of G consists of n linked lists.

For every $u \in V$, there is a linked list (called an **adjacency list**) which is named $Adj[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled v. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in $Adj[u]$ and a node u in $Adj[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.

Example

The adjacency lists for the previous graph are

$$
\begin{aligned}
Adj[1]: &\quad 2 \rightarrow 3 \\
Adj[2]: &\quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \\
Adj[3]: &\quad 1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 8 \\
Adj[4]: &\quad 2 \\
Adj[5]: &\quad 2 \rightarrow 3 \rightarrow 6 \\
Adj[6]: &\quad 5 \\
Adj[7]: &\quad 3 \rightarrow 8 \\
Adj[8]: &\quad 3 \rightarrow 7
\end{aligned}
$$

Breadth-first Search of an Undirected Graph

A **breadth-first search** of an undirected graph begins at a specified vertex s.

The search "spreads out" from s, proceeding in **layers**.

First, all the neighbours of s are **explored**.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A **queue** is used to keep track of the vertices to be explored.

Breadth-first Search

Algorithm: *BFS*(G, s)

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \textbf{gray}$

InitializeQueue(Q)

Enqueue(Q, s)

while $Q \neq \emptyset$

do $\begin{cases} u \leftarrow \textit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \textbf{white} \\ \quad \textbf{then} \begin{cases} colour[v] = \textbf{gray} \\ \pi[v] \leftarrow u \\ \textit{Enqueue}(Q, v) \end{cases} \end{cases} \\ colour[u] \leftarrow \textbf{black} \end{cases}$

Example

We run *breadth-first search* with $s = 1$ on the previous graph:

$colour[1] \leftarrow \textbf{grey}, Q = [1]$

$u \leftarrow 1, Q = []$

$v \leftarrow 2, colour[2] \leftarrow \textbf{grey}, \pi[2] \leftarrow 1, Q = [2]$

$v \leftarrow 3, colour[3] \leftarrow \textbf{grey}, \pi[3] \leftarrow 1, Q = [2, 3]$

$colour[1] \leftarrow \textbf{black}$

$u \leftarrow 2, Q = [3]$

$v \leftarrow 4, colour[4] \leftarrow \textbf{grey}, \pi[4] \leftarrow 2, Q = [3, 4]$

$v \leftarrow 5, colour[5] \leftarrow \textbf{grey}, \pi[5] \leftarrow 2, Q = [3, 4, 5]$

$colour[2] \leftarrow \textbf{black}$

$u \leftarrow 3, Q = [4, 5]$

$v \leftarrow 7, colour[7] \leftarrow \textbf{grey}, \pi[7] \leftarrow 3, Q = [4, 5, 7]$

$v \leftarrow 8, colour[8] \leftarrow \textbf{grey}, \pi[8] \leftarrow 3, Q = [4, 5, 7, 8]$

$colour[3] \leftarrow \textbf{black}$

Example (cont.)

$u \leftarrow 4, Q = [5, 7, 8]$
$colour[4] \leftarrow$ **black**

$u \leftarrow 5, Q = [7, 8]$
$v \leftarrow 6, colour[6] \leftarrow$ **grey**, $\pi[6] \leftarrow 5, Q = [7, 8, 6]$
$colour[5] \leftarrow$ **black**

$u \leftarrow 7, Q = [8, 6]$
$colour[7] \leftarrow$ **black**

$u \leftarrow 8, Q = [6]$
$colour[8] \leftarrow$ **black**

$u \leftarrow 6, Q = []$
$colour[6] \leftarrow$ **black**

The tree edges are $12, 13, 24, 25, 37, 38, 56$.

Properties of Breadth-first Search

A vertex is **white** if it is **undiscovered**.

A vertex is **gray** if it has been **discovered**, but we are still processing its adjacent vertices.

A vertex becomes **black** when all the adjacent vertices have been processed.

If G is **connected**, then every vertex eventually is coloured black and every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

When we explore an edge $\{u, v\}$ starting from u:

- if v is **white**, then uv is a **tree edge** and $\pi[v] = u$ is the **predecessor** of v in the **BFS tree**
- otherwise, uv is a **cross edge**.

The BFS tree consists of all the tree edges.

Shortest Paths via Breadth-first Search

Algorithm: *BFS*(G, s)

for each $v \in V(G)$ **do** $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \textbf{gray}$

$\boxed{dist[s] \leftarrow 0}$

InitializeQueue(Q)

Enqueue(Q, s)

while $Q \neq \emptyset$

do $\begin{cases} u \leftarrow \textit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \textbf{white} \quad \textbf{then} \begin{cases} colour[v] = \textbf{gray} \\ \pi[v] \leftarrow u \\ \textit{Enqueue}(Q, v) \\ \boxed{dist[v] \leftarrow dist[u] + 1} \end{cases} \end{cases} \\ colour[u] \leftarrow \textbf{black} \end{cases}$

Distances in Breadth-first Search

Lemma 6.1

If u is discovered before v, then $dist[u] \leq dist[v]$.

Proof.

By contradiction. Let v be the *first* vertex such that $dist[u] > dist[v]$ for some u that was discovered before v. Denote $d = dist[v]$; then $dist[u] \geq d + 1$. Let $\pi[v] = v_1$; then $dist[v_1] = d - 1$. Let $\pi[u] = u_1$; then $dist[u_1] \geq d$. Note that v_1 was discovered before u_1 since v is the first "out-of-order" vertex. So, in order of discovery, we have v_1, u_1, u, v. Vertex v was discovered while processing $Adj[v_1]$ and vertex u was discovered while processing $Adj[u_1]$. This means that v was discovered before u, a contradiction. □

Distances in Breadth-first Search (cont.)

Lemma 6.2

If $\{u, v\}$ *is* **any edge**, *then* $|dist[u] - dist[v]| \leq 1$.

Proof.

WLOG suppose u is discovered before v.

(1) v is white when we process $Adj[u]$. Then $dist[v] = dist[u] + 1$.

(2) v is grey when we process $Adj[u]$. Let $\pi[v] = v_1$; then v was discovered when $Adj[v_1]$ was being processed. So v_1 was discovered before u. By Lemma 6.1, $dist[v_1] \leq dist[u]$. Also, $dist[v] = dist[v_1] + 1$, so $dist[u] \geq dist[v] - 1$. Since u was discovered before v, we have $dist[u] \leq dist[v]$ by Lemma 6.1. Therefore, $dist[u] \leq dist[v] \leq dist[u] + 1$.

(3) v is black when we process $Adj[u]$. Then $Adj[v]$ has been completely processed and we would already have discovered u from v; contradiction.



Distances in Breadth-first Search (cont.)

Theorem 6.3

$dist[v]$ is the length of the **shortest path** *from s to v.*

Proof.

Let $\delta(v)$ denote the length of the shortest path from s to v. Consider the path $v \quad \pi[v] \quad \pi[\pi[v]] \quad \cdots \quad s$. This path has length $dist[v]$, so $\delta(v) \leq dist[v]$. To complete the proof, we show that $\delta(v) \geq dist[v]$; we will prove this by induction on $\delta(v)$.

Base case: $\delta(v) = 0$. Then $v = s$ and $dist[v] = 0 = \delta(v)$.

Induction assumption: Assume $\delta(v) \geq dist[v]$ if $\delta(v) \leq d - 1$. Now suppose $\delta(v) = d$. Let $s \quad v_1 \quad v_2 \quad \cdots \quad v_{d-1} \quad v_d = v$ be a shortest path (having length d). Then $\delta(v_{d-1}) = d - 1 = dist[v_{d-1}]$ by induction. We have that $dist[v] \leq dist[v_{d-1}] + 1$ (by Lemma 6.2). But $dist[v_{d-1}] = d - 1$, so $dist[v] \leq d = \delta(v)$ and we're done. □

Bipartite Graphs and Breadth-first Search

A graph is **bipartite** if the vertex set can be partitioned as $V = X \cup Y$, in such a way that all edges have one endpoint in X and one endpoint in Y.

A graph is bipartite if and only if it does not contain an **odd cycle**.

BFS can be used to test if a graph is bipartite:

- if we encounter an edge $\{u, v\}$ with $dist[u] = dist[v]$, then G is not bipartite, whereas
- if no such edge is found, then define $X = \{u : dist[u] \text{ is even}\}$ and $Y = \{u : dist[u] \text{ is odd}\}$; then X, Y forms a bipartition.

Bipartite Graphs

Theorem 6.4

A graph is bipartite if an only if it contains no cycle of odd length.

Proof.

($\Rightarrow$): Suppose G contains an odd cycle, $v_1, v_2, \ldots, v_{2k+1}, v_1$. WLOG colour v_1 red. Then we are forced to colour v_2 blue, v_3 red, $\ldots$, and v_{2k+1} red. Then the edge $v_{2k+1}v_1$ joins two red vertices, so G is not bipartite.

($\Leftarrow$): Suppose G is not bipartite. WLOG assume G is connected. Let s be any vertex. Define $X = \{v : dist[v] \text{ is even}\}$ and $Y = \{v : dist[v] \text{ is odd}\}$. Since G is not bipartite, there is an edge uv where $u, v \in X$ or $u, v \in Y$. $dist[u]$ and $dist[v]$ are both even or both odd, so $dist[u] = dist[v]$ since $|dist[u] - dist[v]| \leq 1$ for any edge uv. Denote $d = dist[u] = dist[v]$. $u, u_1 = \pi[u], \ldots, u_d = \pi[u_{d-1}] = s$ and $v, v_1 = \pi[v], \ldots, v_d = \pi[v_{d-1}] = s$ are paths of length d in G. Let $j = \min\{i : u_i = v_i\}$ ($j \leq d$ since $u_d = v_d = s$). Then $v_i, \ldots, v_1, v, u, u_1, \ldots, u_i = v_i$ is an odd cycle. □

Depth-first Search of a Directed Graph

A **depth-first search** uses a **stack** (or **recursion**) instead of a queue.

We define predecessors and colour vertices as in BFS.

It is also useful to specify a **discovery time** $d[v]$ and a **finishing time** $f[v]$ for every vertex v.

We increment a **time counter** every time a value $d[v]$ or $f[v]$ is assigned.

We eventually visit all the vertices, and the algorithm constructs a **depth-first forest**.

The complexity of **depth-first search** is $\Theta(n + m)$.

Depth-first Search

Algorithm: *DFS*(G)
for each $v \in V(G)$
do $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$
$time \leftarrow 0$
for each $v \in V(G)$
do $\begin{cases} \textbf{if } colour[v] = \textbf{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow$ **gray**

$time \leftarrow time + 1$

$d[v] \leftarrow time$

comment: $d[v]$ is the discovery time for vertex v

for each $w \in Adj[v]$

do { **if** $colour[w] =$ **white**
then { $\pi[w] \leftarrow v$
DFSvisit(w)

$colour[v] \leftarrow$ **black**

$time \leftarrow time + 1$

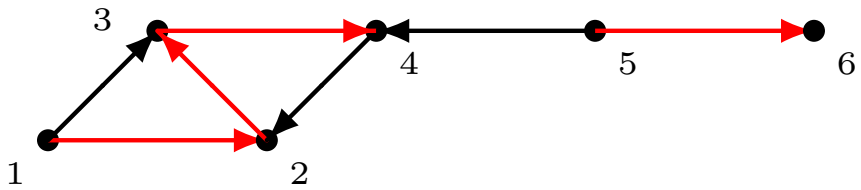
$f[v] \leftarrow time$

comment: $f[v]$ is the finishing time for vertex v

Example of Depth-first Search

Consider the directed graph on vertex set $\{1, 2, 3, 4, 5, 6\}$ with the following adjacency lists:

$$Adj[1] : 2 \to 3 \qquad Adj[2] : 3$$
$$Adj[3] : 4 \qquad Adj[4] : 2$$
$$Adj[5] : 4 \to 6 \qquad Adj[6] :$$



Initial call: *DFSvisit*(1), recursive calls: *DFSvisit*(2), *DFSvisit*(3), *DFSvisit*(4).

Initial call: *DFSvisit*(5), recursive call: *DFSvisit*(6).

The depth-first forest consists of two trees. One tree has arcs $12, 23, 34$ (initial call from *DFSvisit*(1)) and the other tree has arc 56 (initial call from *DFSvisit*(5)).

Classification of Edges in Depth-first Search

- uv is a **tree edge** if $u = \pi[v]$
- uv is a **forward edge** if it is not a tree edge, and v is a descendant of u in a tree in the depth-first forest
- uv is a **back edge** if u is a descendant of v in a tree in the depth-first forest
- any other edge is a **cross edge**.

Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex v when an edge uv is discovered, and the relation between the start and finishing times of u and v, for each possible type of edge uv.

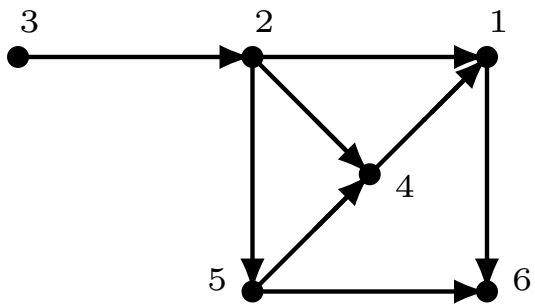
edge type	colour of v	discovery/finish times
tree	**white**	$d[u] < d[v] < f[v] < f[u]$
forward	**black**	$d[u] < d[v] < f[v] < f[u]$
back	**gray**	$d[v] < d[u] < f[u] < f[v]$
cross	**black**	$d[v] < f[v] < d[u] < f[u]$

Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never **overlap**. Two intervals are either **disjoint** or **nested**. This is sometimes called the parenthesis theorem.

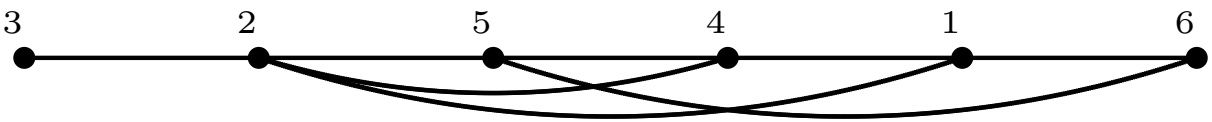
Topological Orderings and DAGs

A directed graph G is a **directed acyclic graph**, or **DAG**, if G contains no directed cycle.

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in V such that $u < v$ whenever $uv \in E$.



Here is a topological ordering—all edges are directed from left to right:



Some Interesting/useful Facts

Lemma 6.5

A DAG contains a vertex of indegree 0.

Proof.

Suppose we have a directed graph in which every vertex has positive indegree. Let v_1 be any vertex. For every $i \geq 1$, let $v_{i+1}v_i$ be an arc. In the sequence $v_1, v_2, v_3, \ldots$, consider the first repeated vertex, $v_i = v_j$ where $j > i$. Then $v_j, v_{j-1}, \ldots, v_i, v_j$ is a directed cycle. □

Some Interesting/useful Facts (cont.)

Theorem 6.6

A directed graph D has a topological sort if and only if it is a DAG.

Proof.

($\Rightarrow$): Suppose D has a directed cycle $v_1, v_2, \ldots, v_j, v_1$. Then $v_1 < v_2 < \cdots < v_j < v_1$, so a topological ordering does not exist.

($\Leftarrow$): Suppose D is a DAG. Then the algorithm below constructs a topological ordering. □

Algorithm: *TopOrdering*(D)

$D_1 \leftarrow D$

for $i \leftarrow 1$ **to** n

do { let v_i be a vertex in D_i having indegree 0
construct D_{i+1} from D_i by deleting v_i and all arcs $v_i v_j$

return $(v_1, v_2, \ldots, v_n)$

Developing an Algorithm based on DFS

Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

Proof.

($\Rightarrow$): Any back edge creates a directed cycle.
($\Leftarrow$): Suppose $C = v_1, v_1, \ldots, v_\ell$ is a directed cycle. WLOG suppose that v_1 is the vertex in C having the lowest discovery time. Consider the arc $v_\ell v_1$. We will prove that $v_\ell v_1$ is a back edge. First, since $d[v_\ell] > d[v_1]$, this arc must be a cross edge or a back edge (see slide # 203). Suppose $v_\ell v_1$ is a cross edge. Then v_1 is black and v_ℓ is grey when the arc $v_\ell v_1$ is processed. But v_1 is not coloured black until all vertices reachable from v_1 are black. This is a contradiction, and hence $v_\ell v_1$ is a back edge. □

Developing an Algorithm based on DFS (cont.)

Lemma 6.8

Suppose D is a DAG. Then $f[v] < f[u]$ for every arc uv.

Proof.

Look at the classification on slide # 203. In a DAG, there are no back edges. For any other type of arc uv, it holds that $f[v] < f[u]$. □

Therefore, if D is a DAG and we order the vertices in reverse order of finishing time, then we get a topological ordering.

Topological Ordering via Depth-first Search

Algorithm: *DFS*(G)

InitializeStack(S)

$DAG \leftarrow true$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \textbf{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

if DAG **then return** (S) **else return** (DAG)

Topological Ordering via Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow$ **gray**

$time \leftarrow time + 1$

$d[v] \leftarrow time$

comment: $d[v]$ is the discovery time for vertex v

for each $w \in Adj[v]$

do { **if** $colour[w] =$ **white**
  then { $\pi[w] \leftarrow v$
    DFSvisit(w)
  if $colour[w] =$ **gray** **then** $DAG \leftarrow false$

$colour[v] \leftarrow$ **black**

Push(S, v)

$time \leftarrow time + 1$

$f[v] \leftarrow time$

comment: $f[v]$ is the finishing time for vertex v

Example

We consider the graph from slide # 204.

It has the the following adjacency lists:

$$\begin{array}{ll} Adj[1]: & 6 \\ Adj[2]: & 1 \to 4 \to 5 \\ Adj[3]: & 2 \end{array} \qquad \begin{array}{ll} Adj[4]: & 1 \\ Adj[5]: & 4 \to 6 \\ Adj[6]: & \end{array}$$

The initial calls are *DFSvisit*(1), *DFSvisit*(2) and *DFSvisit*(3).

The discovery/finish times are as follows:

v	$d[v]$	$f[v]$
1	1	4
2	5	9
3	11	12

v	$d[v]$	$f[v]$
4	6	7
5	8	9
6	2	3

The topological ordering is $3, 2, 5, 4, 1, 6$ (reverse order of finishing time).

Connectivity and Strong Connectivity

An undirected graph is **connected** if there is a path between any two vertices.

How do we determine if a graph is connected using *DFS*?

In a directed graph, a vertex v is **reachable** from a vertex w if there is a directed path from v to w.

A directed graph G is **strongly connected** if any vertex is reachable from any other vertex.

Prove that a directed graph G is strongly connected if all vertices are reachable from s and s is reachable from all vertices, for an arbitrary (fixed) vertex s.

Testing a Directed Graph to see if it is Strongly Connected

1. Pick any vertex s in the directed graph G.
2. Run *DFSvisit*(s) on G.
3. If there exists a white vertex, then QUIT (G is not strongly connected).
4. Otherwise, reverse the direction of all edges in G to construct another digraph G_1.
5. Run *DFSvisit*(s) on G_1.
6. G is strongly connected if and only if there is no white vertex in G_1.

What is the complexity of this algorithm?

(Strongly) Connected Components

For two vertices x and y of G, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from x to y **and** from y to x.

The relation $\sim$ is an **equivalence relation**.

The **strongly connected components** of G are the equivalence classes of vertices defined by the relation $\sim$.

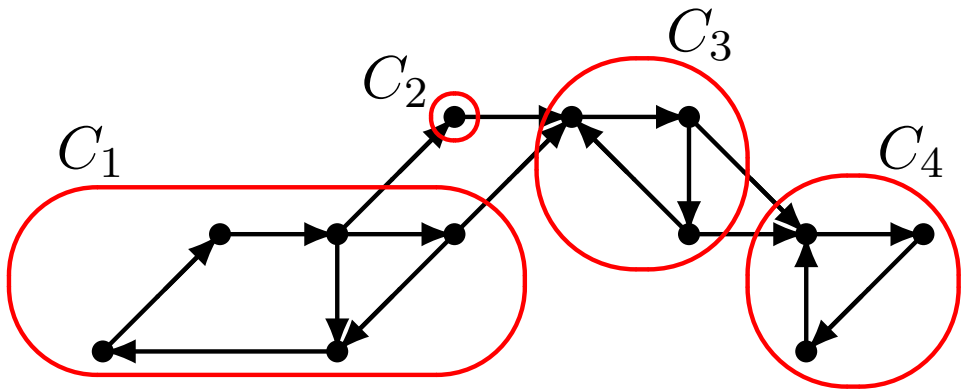
A strongly connected component of a digraph G is a maximal strongly connected subgraph of G.

For undirected graphs, the definition is similar, except we define $x \sim y$ if $x = y$; or if $x \neq y$ and there exists a path joining x and y.

Exercise: How do we determine the connected components of an undirected graph?

Strongly Connected Components of a Digraph G

The following directed graph has strongly connected components as indicated.



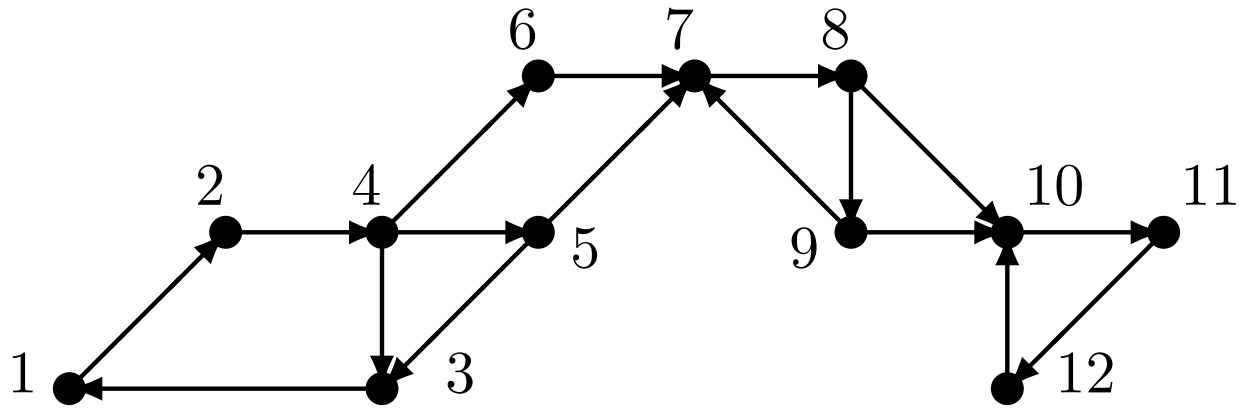
The **component graph** of a directed graph G is a directed graph whose vertices are the strongly connected components of G. There is an arc from C_i to C_j if and only if there is an arc in G from some vertex of C_i to some vertex of C_j.

Exercise: Prove that the component graph of G is a DAG.

Sharir's Algorithm to Find the Strongly Connected Components

1. Perform a depth-first search of G, recording the finishing times $f[v]$ for all vertices v.
2. Construct a directed graph H from G by **reversing** the direction of all edges in G.
3. Perform a depth-first search of H, considering the vertices in **decreasing** order of the values $f[v]$ computed in step 1.
4. The strongly connected components of G are the trees in the depth-first forest constructed in step 3.

Example



Here are the discovery and finish times for each vertex:

v	$d[v]$	$f[v]$
1	1	24
2	2	23
3	19	20
4	3	22
5	18	21
6	4	17

v	$d[v]$	$f[v]$
7	5	16
8	6	15
9	7	14
10	8	13
11	9	12
12	10	11

Depth-first Search of H

Assume that $f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}]$.

Algorithm: *DFS*(H)

for $j \leftarrow 1$ **to** n
  do $colour[v_{i_j}] \leftarrow$ **white**
$scc \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do **if** $colour[v_{i_j}] =$ **white**
    then $scc \leftarrow scc + 1$
      DFSvisit(H, v_{i_j}, scc)
return $(comp)$

comment: $comp[v]$ is the strongly connected component containing v

DFSvisit for H

Algorithm: *DFSvisit*(H, v, scc)

$colour[v] \leftarrow$ **gray**

$comp[v] \leftarrow scc$

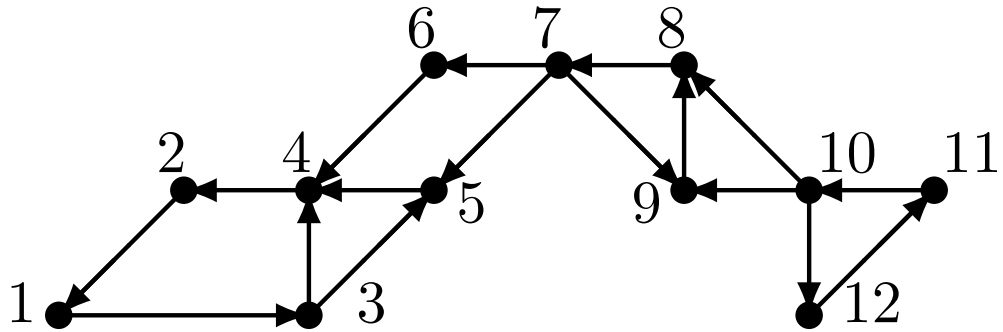
for each $w \in Adj[v]$

do **if** $colour[w] =$ **white**
then *DFSvisit*(H, w, scc)

$colour[v] \leftarrow$ **black**

Example

Here is the previous directed graph with edge directions reversed:



Recall that we process the vertices in the order

$$1, 2, 4, 5, 3, 6, 7, 8, 9, 10, 11, 12.$$

DFSvisit(1) explores the s.c.c having vertices $\{1, 2, 3, 4, 5\}$.

DFSvisit(6) explores the s.c.c having vertices $\{6\}$.

DFSvisit(7) explores the s.c.c having vertices $\{7, 8, 9\}$.

DFSvisit(10) explores the s.c.c having vertices $\{10, 11, 12\}$.

Properties of Strongly Connected Components of a Digraph G

For a strongly connected component C, define $f[C] = \max\{f[v] : v \in C\}$ and $d[C] = \min\{d[v] : v \in C\}$.

Lemma 6.9

If C_i, C_j are strongly connected components, and there is an arc from C_i to C_j in the component graph, then $f[C_i] > f[C_j]$.

Proof.

Suppose $d(C_i) < d(C_j)$. Let $u \in C_i$ be the first discovered vertex. All vertices in $C_i \cup C_j$ are reachable from u, so they are descendants of u in the DFS tree. Hence $f(v) < f(u)$ for all $v \in C_i \cup C_j$, $v \neq u$. Therefore $f(C_i) > f(C_j)$.

Suppose $d(C_i) > d(C_j)$. In this case, no vertices in C_i are reachable from C_j, so $f(C_j) < d(C_i) < f(C_i)$. □

Proof of Correctness of Sharir's Algorithm

First, note that G and H have the same strongly connected components

Let $u = v_{i_1}$ be the first vertex visited in step 3. Let C be the s.c.c. containing u and let C' be any other s.c.c.

$f(C) > f(C')$, so there is no edge from C' to C in G (by Lemma 6.9).

Therefore there is no edge from C to C' in H.

Hence no vertex in C' is reachable from u in H.

Therefore, *DFSvisit*(u) explores the vertices in C (and only those vertices); this forms one DFS tree in H.

Next, *DFSvisit*(v_{i_2}) explores the vertices in the s.c.c. containing v_{i_2}, etc.

Every time we make an initial call to *DFSvisit*, we are exploring a new s.c.c.

We increment scc, which is used to label the various s.c.c.

$comp[v]$ denotes the label of the s.c.c. containing v.

Minimum Spanning Trees

A **spanning tree** in a connected, undirected graph $G = (V, E)$ is a subgraph T that is a tree containing every vertex of V.

T is a spanning tree of G if and only if T is an acyclic subgraph of G that has $n - 1$ edges (where $n = |V|$).

Problem 6.10

Minimum Spanning Tree

Instance: *A connected, undirected graph $G = (V, E)$ and a* **weight function** $w : E \to \mathbb{R}$.

Find: *A spanning tree T of G such that*

$$\sum_{e \in T} w(e)$$

is minimized (this is called a **minimum spanning tree**, *or* **MST**).

Kruskal's Algorithm

As a preprocessing step, sort and relabel the edges so

$$w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m),$$

where $m = |E|$.

Algorithm: *Kruskal*(G, w)

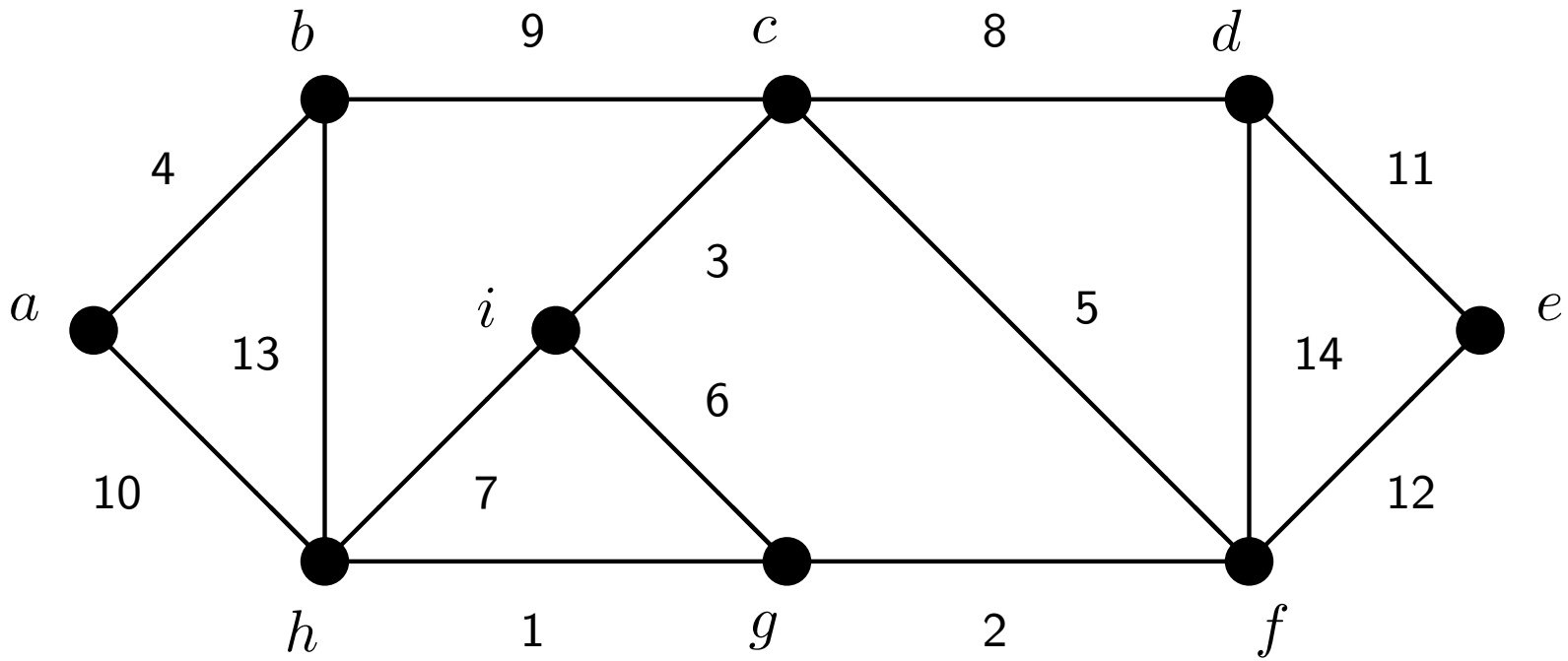
$A \leftarrow \emptyset$

for $j \leftarrow 1$ **to** m

do { **if** $A \cup \{e_j\}$ does not contain a cycle
then $A \leftarrow A \cup \{e_j\}$

return (A)

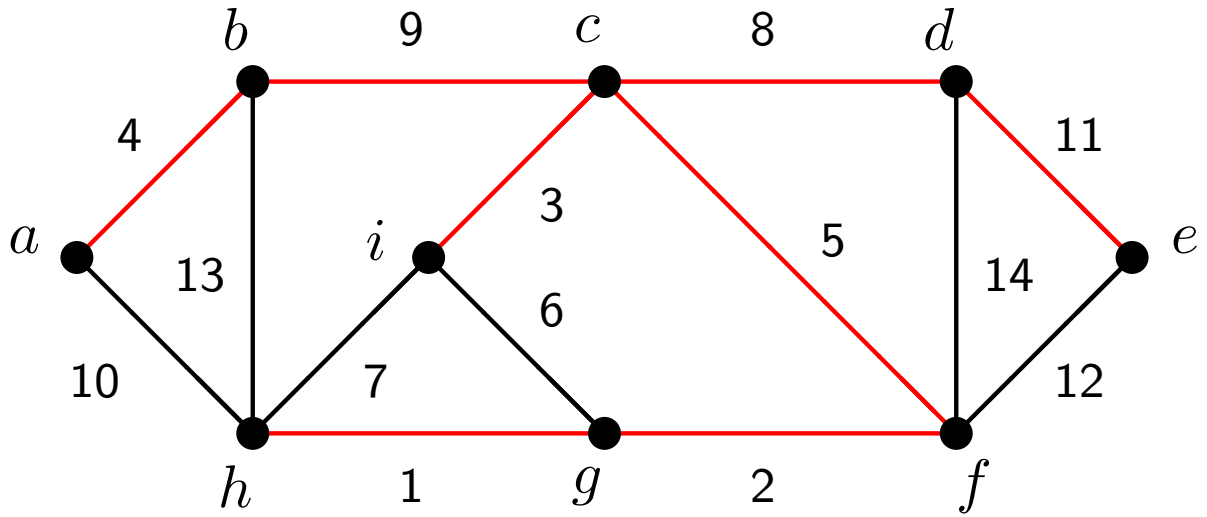
An Example of Kruskal's Algorithm



An Example of Kruskal's Algorithm (cont.)

edge	weight	add to tree?
gh	1	yes
fg	2	yes
ci	3	yes
ab	4	yes
cf	5	yes
gi	6	no
hi	7	no

edge	weight	add to tree?
cd	8	yes
bc	9	yes
ah	10	no
de	11	yes
ef	12	no
bh	13	no
df	14	no



Implementation Details for Kruskal's Algorithm

We use a **union-find** data structure to determine if an edge uv has vertices in two different trees.

Every tree T will contain a **leader vertex**.

To find the leader vertex from a vertex v, we use an auxiliary array L.

From v, follow a directed path $v \to L[v] \to L[L[v]] \cdots$ until we reach a vertex w with $L[w] = w$; then $w = \mathit{find}(v)$ is the leader vertex for the tree containing v.

Two vertices u and v are in the same tree if and only if $\mathit{find}(u) = \mathit{find}(v)$.

Initially, there are n one-vertex trees and $L[v] = v$ for all v.

When we use an edge uv to merge two trees, we perform the following *union* operation

1. $u' \leftarrow \mathit{find}(u)$
2. $v' \leftarrow \mathit{find}(v)$
3. $L[u'] \leftarrow v'$.

Implementation Details and Complexity

Suppose we also keep track of the depth of each tree. In step 3, we always take u' to be the leader of the tree having smaller depth.

If we merge two trees of depth d, we get a tree of depth $d+1$. If we merge a tree of depth d and one of depth $< d$, we have a tree of depth d.

Then *union* and *find* each run in $O(\log n)$ time (this is because a tree of depth d has at least 2^d vertices, a fact that can be proven by induction on d).

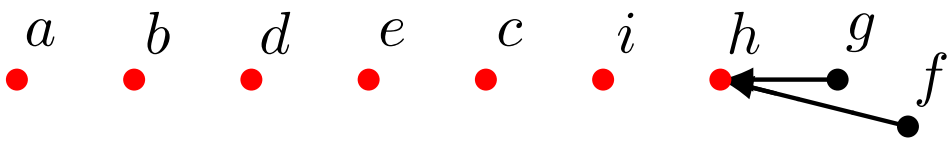
This leads to an algorithm for MST having complexity $O(m \log n)$ (the pre-sort has complexity $O(m \log m)$, and the iterative part of the algorithm has complexity $O(m \log n)$).

Union-find Data Structure (example)

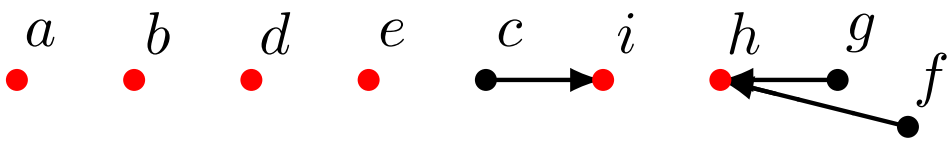
Considering edge gh, we have *find*$(g) = g$, *find*$(h) = h$. The two trees have depth 0. Suppose we direct $g \to h$; then the tree $\{g, h\}$ has depth 1.



Considering edge fg, we have *find*$(f) = f$, *find*$(g) = h$. The two trees have depth 0 and 1, resp. Thus we must direct $f \to h$ (see the previous slide). Then the resulting tree $\{f, g, h\}$ has depth 1.

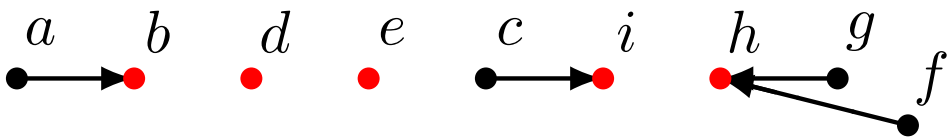


Considering edge ci, we have *find*$(c) = c$, *find*$(i) = i$. The two trees have depth 0. Suppose we direct $c \to i$; then the tree $\{c, i\}$ has depth 1.

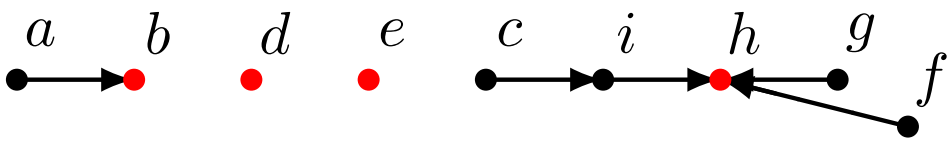


Union-find Data Structure (example, cont.)

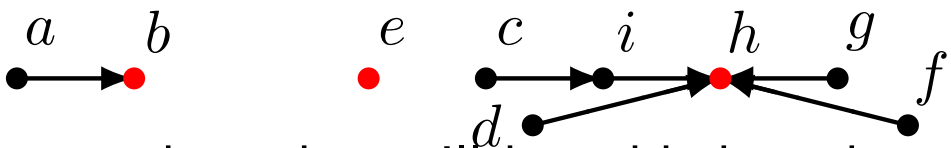
Considering edge ab, we have *find*$(a) = a$, *find*$(b) = b$. The two trees have depth 0. Say we direct $a \to b$; then the tree $\{a, b\}$ has depth 1.



Considering edge cf, we have *find*$(c) = i$, *find*$(f) = h$. The two trees have depth 1. Suppose we direct $i \to h$; then the tree $\{c, f, g, h, i\}$ has depth 2.



Edges gi and hi are not added to the MST because *find*$(g) =$ *find*(i) and *find*$(h) =$ *find*(i). Considering edge cd, we have *find*$(c) = h$, *find*$(d) = d$. The two trees have depth 0 and 2, resp. Thus we must direct $d \to h$. Then the resulting tree containing vertices $\{c, d, f, g, h, i\}$ has depth 2.



There are two more edges that will be added to the tree, namely bc and de. The reader can check the details.

Proof of Correctness

Let's assume that all edge weights are distinct. Let A be the spanning tree constructed by *Kruskal's algorithm* and let A' be an arbitrary MST.

Suppose the edges in A are named $f_1, f_2, \ldots, f_{n-1}$, where $w(f_1) < w(f_2) \cdots < w(f_{n-1})$. Suppose $A \neq A'$ and let f_j be the first edge in $A \setminus A'$.

$A' \cup \{f_j\}$ contains a unique cycle, say C. Let e' be the first (i.e, lowest weight) edge of C that is not in A (such an edge exists because $C \not\subseteq A$). Define $A'' = A' \cup \{f_j\} \setminus \{e'\}$. Then $w(A'') = w(A') + w(f_j) - w(e')$. Since A' is an MST, we must have $w(A'') \geq w(A')$. Therefore, $w(f_j) \geq w(e')$. The edge weights are all distinct, so $w(f_j) > w(e')$.

What happened when *Kruskal's algorithm* considered the edge e'? This occurred before it considered the edge f_j, because $w(f_j) > w(e')$. Since *Kruskal's algorithm* rejected the edge e', the edges $f_1, \ldots, f_{j-1}, e'$ must contain a cycle. However, A' contains all these edges and A' is a tree, so we have a contradiction. Therefore $A = A'$ and A is an MST.

Prim's Algorithm (idea)

We initially choose an arbitrary vertex u_0.

Define $V_A = \{u_0\}$ and $A = \{e\}$, where e is the **minimum weight** edge incident with u_0.

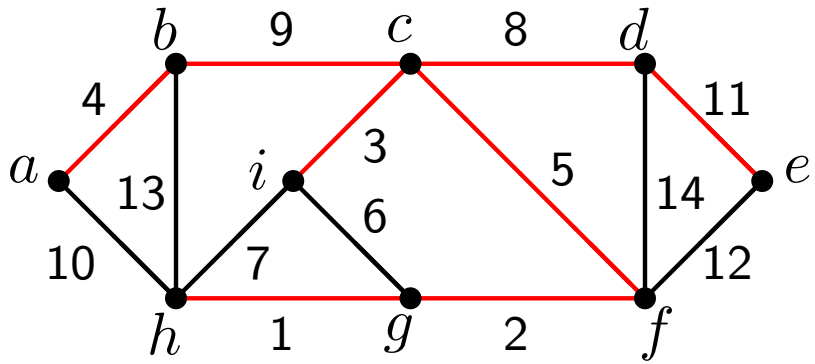
A is always a **single tree** and V_A is the set of vertices in A.

At each step, we select the minimum weight edge that joins a vertex $u \in V_A$ to a vertex $v \notin V_A$.

We add v to V_A and we add the edge uv to A.

Then repeat these operations until A is a spanning tree.

Example



Start at a; $V_A = \{a\}$.

Possible edges ab, ah; minimum edge ab; $V_A = \{a, b\}$.

Possible edges ah, bh, bc; minimum edge bc; $V_A = \{a, b, c\}$.

Possible edges ah, bh, hi, cd, cf, ci; minimum edge ci; $V_A = \{a, b, c, i\}$.

Possible edges ah, bh, hi, gi, cf, cd; minimum edge cf; $V_A = \{a, b, c, f, i\}$.

Possible edges $ah, bh, hi, gi, fg, cd, df, ef$; minimum edge fg; $V_A = \{a, b, c, f, g, i\}$.

Possible edges $ah, bh, hi, gh, cd, df, ef$; minimum edge gh; $V_A = \{a, b, c, f, g, h, i\}$.

Possible edges cd, df, ef; minimum edge cd; $V_A = \{a, b, c, d, f, g, h, i\}$.

Possible edges de, ef; minimum edge de; $V_A = \{a, b, c, d, e, f, g, h, i\}$.

Implementation

Assume $w(u, v) = \infty$ if $\{u, v\} \notin E$.

For a vertex $v \notin V_A$, define

$$\begin{aligned} N[v] &= u, \text{ where } \{u, v\} \text{ is a minimum weight edge such that } u \in V_A \\ W[v] &= w(N[v], v). \end{aligned}$$

The vertex to be added to V_A is the vertex $v \notin V_A$ with the minimum W-value.

The edge to be added to A is $v\,N(v)$.

Updating W-values: when we add the vertex v to V_A, vv' is a new "candidate edge".

So we update $W[v']$ using the formula

$$W[v'] = \min\{W[v'], w(v, v')\}$$

for every edge vv'.

Prim's Algorithm

Algorithm: *Prim*(G, w)

$A \leftarrow \emptyset$

$V_A \leftarrow \{u_0\}$, where u_0 is arbitrary

for all $v \in V \backslash \{u_0\}$

do $\begin{cases} W[v] \leftarrow w(u_0, v) \\ N[v] \leftarrow u_0 \end{cases}$

while $|A| < n - 1$

do $\begin{cases} \text{choose } v \in V \backslash V_A \text{ such that } W[v] \text{ is minimized} \\ V_A \leftarrow V_A \cup \{v\} \\ u \leftarrow N[v] \\ A \leftarrow A \cup \{uv\} \\ \textbf{for all } v' \in V \backslash V_A \\ \quad \textbf{do} \begin{cases} \textbf{if } w(v, v') < W[v'] \\ \quad \textbf{then} \begin{cases} W[v'] \leftarrow w(v, v') \\ N[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (A)

Data Structures and Complexity

Simple implementation:

- There are $n-1$ iterations of the "while" loop.
- Finding v takes time $O(n)$.
- Updating W-values takes time $O(n)$.
- The algorithm has complexity $O(n^2)$.

Priority queue implementation: Use a priority queue (implemented as a min-heap) to store the W-values.

Fibonacci heap implementation: Use a **Fibonacci heap** to store the W-values.

A General Algorithm: Definitions

Let $G = (V, E)$ be a graph. A **cut** is a partition of V into two non-empty (disjoint) sets, i.e., a pair $(S, V \backslash S)$, where $S \subseteq V$ and $1 \leq |S| \leq n - 1$.

Let $(S, V \backslash S)$ be a cut in a graph $G = (V, E)$. An edge $e \in E$ is a **crossing edge** with respect to the cut $(S, V \backslash S)$ if e has one endpoint in S and one endpoint in $V \backslash S$.

Let $A \subseteq E$. A cut $(S, V \backslash S)$ **respects** the set of edges A provided that no edge in A is a crossing edge.

A General Greedy Algorithm to Find an MST

Algorithm: *GreedyMST*(G, w)

$A \leftarrow \emptyset$

while $|A| < n - 1$

do
- let $(S, V \backslash S)$ be a cut that respects A
- let e be a minimum weight crossing edge
- $A \leftarrow A \cup \{e\}$

return (A)

Correctness Proof of the General Greedy Algorithm to Find an MST

We prove that the spanning tree A constructed by the general greedy algorithm is a MST, assuming all edge weights are distinct.

Let $e_1, \ldots, e_{n-1}$ be the edges in A in order that they are added to A. We prove by induction on j that $\{e_1, \ldots, e_j\}$ is contained in an MST. $j = 0$ is a trivial base case.

Induction assumption: suppose that $A_{j-1} = \{e_1, \ldots, e_{j-1}\} \subseteq T$, where T is an MST, and consider e_j. If $e_j \in T$, we're done, so assume $e_j \notin T$.

There is a cut $(S, V \setminus S)$ respecting A_{j-1} for which e_j is the minimum crossing edge. $T \cup \{e_j\}$ contains a unique cycle C. There is an edge $e' \neq e_j$ such that $e' \in C$ and e' is a crossing edge for the cut $(S, V \setminus S)$.

Let $T' = T \cup \{e_j\} \setminus \{e'\}$. $w(T') = w(T) + w(e_j) - w(e')$. T' is a spanning tree and T is an MST, so $w(T') \geq w(T)$; hence $w(e_j) \geq w(e')$. But e_j is the minimum weight crossing edge, so $w(e_j) < w(e')$; contradiction.

Single Source Shortest Paths

Problem 6.11

Single Source Shortest Paths

Instance: *A directed graph $G = (V, E)$, a non-negative* **weight function** $w : E \to \mathbb{R}^+ \cup \{0\}$, *and a* **source vertex** $u_0 \in V$.

Find: *For every vertex $v \in V$, a directed path P from u_0 to v such that*

$$w(P) = \sum_{e \in P} w(e)$$

is minimized.

The term **shortest path** really means **minimum weight path**.

We are asked to find n different shortest paths, one for each vertex $v \in V$.

Dijkstra's Algorithm (Main Ideas)

Dijkstra's algorithm requires that the graph have no edge weights < 0; it works for directed or undirected graphs.

S is a subset of vertices such that the shortest paths from u_0 to all vertices in S are known; initially, $S = \{u_0\}$.

For all vertices $v \in S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v, and all vertices on P_v are in the set S.

For all vertices $v \notin S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v in which all interior vertices are in S.

For $v \neq u_0$, $\pi[v]$ is the **predecessor** of v on the path P_v.

At each stage of the algorithm, we choose $v \in V \backslash S$ so that $D[v]$ is minimized, and then we add v to S (see the Lemma on the next slide).

Then the arrays D and π are updated appropriately.

Dijkstra's Algorithm (Main Ideas, cont.)

Lemma 6.12

Suppose v has the smallest D-value of any vertex not in S. Then $D[v]$ equals the weight of the shortest (u_0, v)-path.

Proof.

Suppose there is a (u_0, v)-path P' with weight less than $D[v]$. Let v' be the first vertex of P' not in S. Observe that $v' \neq v$. Decompose P' into two paths: a (u_0, v')- path P_1 and a (v', v)- path P_2. We have

$$\begin{aligned} w(P') &= w(P_1) + w(P_2) \\ &\geq D[v'] + w(P_2) \\ &\geq D[v] \quad (w(P_2) \geq 0 \text{ because all edge weights are } \geq 0). \end{aligned}$$

This is a contradiction because we assumed $w(P') < w(P)$. □

Updating D and π

Lemma 6.12 says we can add v to S.

To update a value $D[v']$ for $v' \notin S$, we consider the new "candidate" path consisting of the shortest (u_0, v)-path together with the edge vv'.

If this is shorter than the current best (u_0, v')-path, then update.

Updating is only required for vertices $v' \in Adj[v]$

Dijkstra's Algorithm

Algorithm: *Dijkstra*(G, w, u_0)

$S \leftarrow \{u_0\}$

$D[u_0] \leftarrow 0$

for all $v \in V \backslash \{u_0\}$

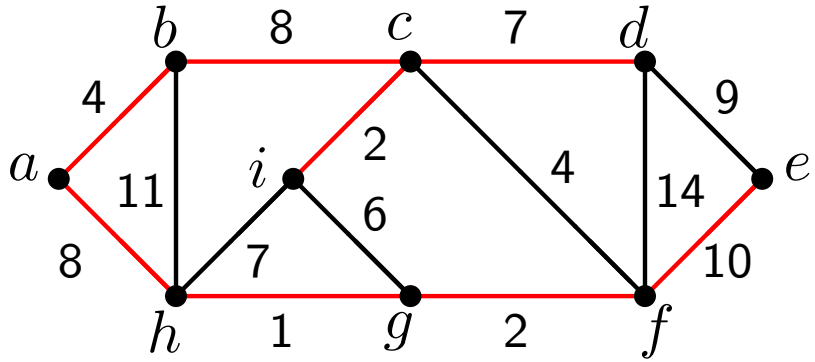
do $\begin{cases} D[v] \leftarrow w(u_0, v) \\ \pi[v] \leftarrow u_0 \end{cases}$

while $|S| < n$

do $\begin{cases} \text{choose } v \in V \backslash S \text{ such that } D[v] \text{ is minimized} \\ S \leftarrow S \cup \{v\} \\ \textbf{for all } v' \in V \backslash S \\ \quad \textbf{do} \begin{cases} \textbf{if } D[v] + w(v, v') < D[v'] \\ \quad \textbf{then} \begin{cases} D[v'] \leftarrow D[v] + w(v, v') \\ \pi[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (D, π)

Example of Dijkstra's Algorithm



	D-values							
S	b	c	d	e	f	g	h	i
$\{a\}$	4	∞	∞	∞	∞	∞	8	∞
$\{a, b\}$	4	12	∞	∞	∞	∞	8	∞
$\{a, b, h\}$	4	12	∞	∞	∞	9	8	15
$\{a, b, g, h\}$	4	12	∞	∞	11	9	8	15
$\{a, b, f, g, h\}$	4	12	25	21	11	9	8	15
$\{a, b, c, f, g, h\}$	4	12	19	21	11	9	8	14
$\{a, b, c, f, g, h, i\}$	4	12	19	21	11	9	8	14
$\{a, b, c, d, f, g, h, i\}$	4	12	19	21	11	9	8	14
$\{a, b, c, d, e, f, g, h, i\}$	4	12	19	21	11	9	8	14

Finding the Shortest Paths

Algorithm: *FindPath*(u_0, π, v)

$path \leftarrow v$

$u \leftarrow v$

while $u \neq u_0$

do $\begin{cases} u \leftarrow \pi[u] \\ path \leftarrow u \parallel path \end{cases}$

return $(path)$

Dijkstra and Prim

Dijkstra's algorithm (for shortest paths) is very similar to *Prim's algorithm* (for MST).

The only difference is the updating step (also known as relaxation)

Prim	Dijkstra
if $w(v, v') < W[v']$	**if** $D[v] + w(v, v') < D[v']$
then $W[v'] \leftarrow w(v, v')$	**then** $D[v'] \leftarrow D[v] + w(v, v')$

Shortest Paths and Negative Weight Cycles

Subsequent algorithms we will be studying will solve shortest path problems as long as there are no cycles having negative weight.

If there is a negative weight cycle, then there is no shortest path (why?).

There is still a shortest simple path, but there are apparently no known efficient algorithms to find the shortest simple paths in in graphs containing negative weight cycles.

If there are no negative weight cycles, then for any two vertices u and v, there is a shortest (u, v)-path that is simple (a shortest path could contain cycles of weight 0, but removal of all such cycles would yield a simple shortest path having the same weight).

Negative weight edges in an undirected graph are not allowed, as they would give rise to a negative weight cycle (consisting of two edges) in the associated directed graph.

Shortest Paths in a DAG

If G is a DAG, we perform a topological ordering of the vertices. Suppose the resulting ordering is $v_1, \ldots, v_n$. Then we find all the shortest paths in G with source v_1.

Algorithm: *DAG Shortest paths*(G, w, v_1)

for $j \leftarrow 1$ **to** n

do $\begin{cases} D[v_1] \leftarrow \infty \\ \pi[v_j] \leftarrow undefined \end{cases}$

$D[v_1] \leftarrow 0$

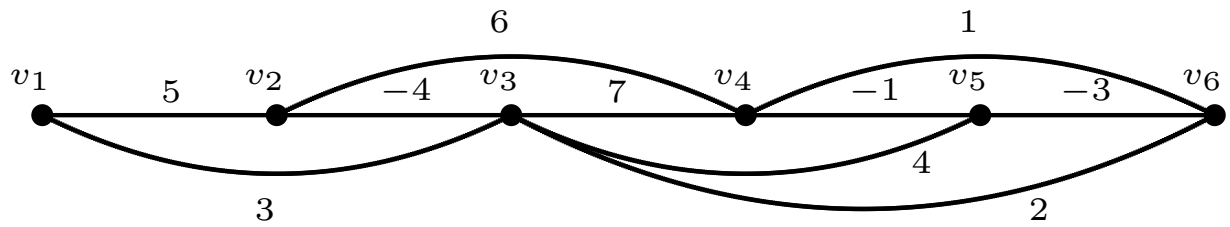
for $j \leftarrow 1$ **to** $n-1$

do $\begin{cases} \textbf{for all } v' \in Adj[v_j] \\ \quad \textbf{do} \begin{cases} \textbf{if } D[v_j] + w(v_j, v') < D[v'] \\ \quad \textbf{then} \begin{cases} D[v'] \leftarrow D[v_j] + w(v_j, v') \\ \pi[v'] \leftarrow v_j \end{cases} \end{cases} \end{cases}$

return (D, π)

Example

A directed graph, where all edges are directed from left to right:



j	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$	$D[v_5]$	$D[v_6]$
0	0	∞	∞	∞	∞	∞
1	0	5	3	∞	∞	∞
2	0	5	1	11	∞	∞
3	0	5	1	8	5	3
4	0	5	1	8	5	3
5	0	5	1	8	5	2

Bellman-Ford

The *Bellman-Ford algorithm* solves the single source shortest path problem in any directed graph without negative weight cycles.

The algorithm is very simple to describe:

Repeat $n-1$ times: *relax* every edge in the graph (where *relax* is the updating step in Dijkstra's algorithm).

Dijkstra's algorithm has complexity $O(m \log n)$ (using priority queues) whereas *Bellman-Ford* has complexity $O(mn)$.

However, *Dijkstra* requires that the graph contain no negative weight edges.

Bellman-Ford (cont.)

Algorithm: *Bellman-Ford*(G, w, u_0)

for all u

do $\begin{cases} D[u] \leftarrow \infty \\ \pi[u] \leftarrow undefined \end{cases}$

$D[u_0] \leftarrow 0$

for $i \leftarrow 1$ **to** $n - 1$

do $\begin{cases} \textbf{for all } (u, v) \in E \\ \quad \textbf{do} \begin{cases} \textbf{if } D[u] + w(u, v) < D[v] \\ \quad \textbf{then} \begin{cases} D[v] \leftarrow D[u] + w(u, v) \\ \pi[v] \leftarrow u \end{cases} \end{cases} \end{cases}$

return (D, π)

All-Pairs Shortest Paths

Problem 6.13

All-Pairs Shortest Paths

Instance: *A directed graph $G = (V, E)$, and a* **weight matrix** *W, where $W[i, j]$ denotes the weight of edge ij, for all $i, j \in V$, $i \neq j$.*

Find: *For all pairs of vertices $u, v \in V$, $u \neq v$, a directed path P from u to v such that*

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in G.

A Dynamic Programming Approach

Suppose we successively consider paths of length $1, 2, \ldots, n-1$. Let $L_m[i,j]$ denote the minimum-weight (i,j)-path having at most m edges. We want to compute L_{n-1}. We can use a dynamic programming approach to do this.

Initialization: $L_1 = W$.

Optimal structure: Let k be the predecessor of j on the minimum-weight (i,j)-path P having at most m edges. Then the portion of P from i to k, say P', is a minimum-weight (i,k)-path having at most $m-1$ edges. This is the *optimal structure* required in order to find a dynamic programming algorithm.

Updating: For $m \geq 2$,

$$L_m[i,j] = \min\{L_{m-1}[i,k] + L_1[k,j] : 1 \leq k \leq n\}.$$

(Note that $k = i, j$ does not cause any problems.)

Complexity: $O(n^4)$.

First Solution

Algorithm: *FairlySlowAllPairsShortestPath*(W)

$L_1 \leftarrow W$
for $m \leftarrow 2$ **to** $n-1$
do
  for $i \leftarrow 1$ **to** n
  do
    for $j \leftarrow 1$ **to** n
    do
      $\ell \leftarrow \infty$
      for $k \leftarrow 1$ **to** n
      do $\ell \leftarrow \min\{\ell, L_{m-1}[i,k] + W[k,j]\}$
      $L_m[i,j] \leftarrow \ell$
return (L_{n-1})

Second Solution: Successive Doubling

Algorithm: *FasterAllPairsShortestPath*(W)

$L_1 \leftarrow W$
$m \leftarrow 1$
while $m < n - 1$
do
  for $i \leftarrow 1$ **to** n
    do
      for $j \leftarrow 1$ **to** n
        do
          $\ell \leftarrow \infty$
          for $k \leftarrow 1$ **to** n
            do $\ell \leftarrow \min\{\ell, L_m[i,k] + L_m[k,j]\}$
          $L_{2m}[i,j] \leftarrow \ell$
  $m \leftarrow 2m$
return (L_m)

Third Solution: Floyd-Warshall

Algorithm: *FloydWarshall*(W)

$D_0 \leftarrow W$

for $m \leftarrow 1$ **to** n

do { **for** $i \leftarrow 1$ **to** n

do { **for** $j \leftarrow 1$ **to** n **do**

$D_m[i,j] \leftarrow \min\{D_{m-1}[i,j], D_{m-1}[i,m] + D_{m-1}[m,j]\}$

return (D_n)

Correctness of Floyd-Warshall

Let P be the shortest (i, j) path having all of its interior vertices in the set $\{1, \ldots, m\}$. We do not have to assume that P is a simple path.

Case 1: If P has all of its interior vertices in the set $\{1, \ldots, m-1\}$, then $w(P) = D_{m-1}[i, j]$.

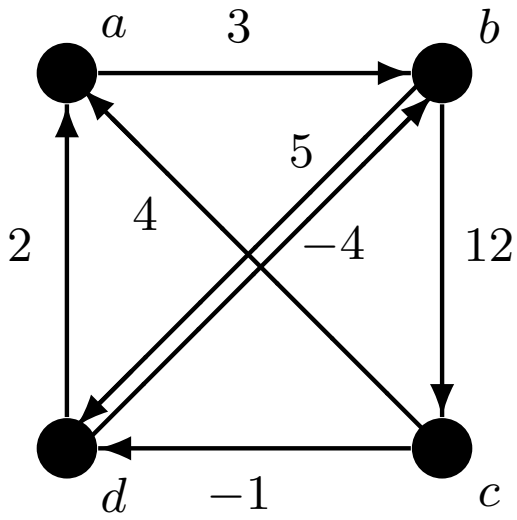
Case 2: If P contains m as an interior vertex, then we can decompose P into two disjoint paths: an (i, m) path P_1 and an (m, j) path P_2. It does not matter if P_1 and P_2 are simple.

Clearly P_1 is a shortest (i, m) path in which the interior vertices are in the set $\{1, \ldots, m-1\}$, and P_2 is a shortest (m, j) path in which the interior vertices are in the set $\{1, \ldots, m-1\}$,

Hence, $w(P_1) = D_{m-1}[i, m]$ and $w(P_2) = D_{m-1}[m, j]$

Then $w(P) = w(P_1) + w(P_2) = D_{m-1}[i, m] + D_{m-1}[m, j]$.

Example of Floyd-Warshall



Example of Floyd-Warshall (cont.)

$$D_0 = \begin{pmatrix} 0 & 3 & \infty & \infty \\ \infty & 0 & 12 & 5 \\ 4 & \infty & 0 & -1 \\ 2 & -4 & \infty & 0 \end{pmatrix} \qquad D_1 = \begin{pmatrix} 0 & 3 & \infty & \infty \\ \infty & 0 & 12 & 5 \\ 4 & \boxed{7} & 0 & -1 \\ 2 & -4 & \infty & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 3 & \boxed{15} & \boxed{8} \\ \infty & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & \boxed{8} & 0 \end{pmatrix} \qquad D_3 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ \boxed{16} & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ \boxed{7} & 0 & 12 & 5 \\ \boxed{1} & \boxed{\text{-5}} & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

Table of Contents

7 **Intractability and Undecidability**
- Decision Problems
- The Complexity Class P
- Decision, Optimal Value and Optimization Problems
- The Complexity Class NP
- Reductions
- NP-completeness and NP-complete Problems
- Undecidability

Decision Problems

Decision Problem: Given a problem instance I, answer a certain question "yes" or "no".

Problem Instance: Input for the specified problem.

Problem Solution: Correct answer ("yes" or "no") for the specified problem instance. I is a **yes-instance** if the correct answer for the instance I is "yes". I is a **no-instance** if the correct answer for the instance I is "no".

Size of a problem instance: $Size(I)$ is the number of bits required to specify (or encode) the instance I.

The Complexity Class P

Algorithm Solving a Decision Problem: An algorithm A is said to **solve** a decision problem Π provided that A finds the correct answer ("yes" or "no") for every instance I of Π in finite time.

Polynomial-time Algorithm: An algorithm A for a decision problem Π is said to be a **polynomial-time algorithm** provided that the complexity of A is $O(n^k)$, where k is a positive integer and $n = \mathit{Size}(I)$.

The Complexity Class P denotes the set of all decision problems that have polynomial-time algorithms solving them. We write $\Pi \in \mathbf{P}$ if the decision problem Π is in the complexity class **P**.

Cycles in Graphs

Problem 7.1

Cycle

Instance: *An undirected graph $G = (V, E)$.*

Question: *Does G contain a cycle?*

Problem 7.2

Hamiltonian Cycle

Instance: *An undirected graph $G = (V, E)$.*

Question: *Does G contain a hamiltonian cycle?*

A **hamiltonian cycle** is a cycle that passes through every vertex in V exactly once.

Knapsack Problems

Problem 7.3

0-1 Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$*?*

Problem 7.4

Rational Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in [0,1]^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$*?*

Polynomial-time Turing Reductions

Suppose Π_1 and Π_2 are problems (not necessarily decision problems). A (hypothetical) algorithm B to solve Π_2 is called an **oracle** for Π_2.

Suppose that A is an algorithm that solves Π_1, assuming the existence of an oracle B for Π_2. (B is used as a subroutine within the algorithm A.)

Then we say that A is a **Turing reduction** from Π_1 to Π_2, denoted $\Pi_1 \leq^T \Pi_2$.

A Turing reduction A is a **polynomial-time Turing reduction** if the running time of A is polynomial, under the assumption that the oracle B has **unit cost** running time.

If there is a polynomial-time Turing reduction from Π_1 to Π_2, we write $\Pi_1 \leq_P^T \Pi_2$.

Informally: Existence of a polynomial-time Turing reduction means that if we can solve Π_2 in polynomial time, then we can solve Π_1 in polynomial time.

Travelling Salesperson Problems

Problem 7.5

TSP-Optimization

Instance: *A graph G and edge weights $w : E \to \mathbb{Z}^+$.*

Find: *A hamiltonian cycle H in G such that $w(H) = \sum_{e \in H} w(e)$ is minimized.*

Problem 7.6

TSP-Optimal Value

Instance: *A graph G and edge weights $w : E \to \mathbb{Z}^+$.*

Find: *The minimum T such that there exists a hamiltonian cycle H in G with $w(H) = T$.*

Problem 7.7

TSP-Decision

Instance: *A graph G, edge weights $w : E \to \mathbb{Z}^+$, and a target T.*

Question: *Does there exist a hamiltonian cycle H in G with $w(H) \leq T$?*

TSP-Optimal Value $\leq_P^T$ TSP-Dec

Algorithm: *TSP-OptimalValue-Solver*(G, w)
external *TSP-Dec-Solver*
$hi \leftarrow \sum_{e \in E} w(e)$
$lo \leftarrow 0$
if not *TSP-Dec-Solver*(G, w, hi) **then return** (∞)
while $hi > lo$
do
$mid \leftarrow \lfloor \frac{hi+lo}{2} \rfloor$
if *TSP-Dec-Solver*(G, w, mid)
then $hi \leftarrow mid$
else $lo \leftarrow mid + 1$
return (hi)

This is a standard binary search technique.

TSP-Optimization $\leq_P^T$ TSP-Dec

Algorithm: *TSP-Optimization-Solver*$(G = (V, E), w)$

external *TSP-OptimalValue-Solver*, *TSP-Dec-Solver*

$T^* \leftarrow$ *TSP-OptimalValue-Solver*(G, w)

if $T^* = \infty$ **then return** ("no hamiltonian cycle exists")

$w_0 \leftarrow w$

$H \leftarrow \emptyset$

for all $e \in E$

do
- $w_0[e] \leftarrow \infty$
- **if not** *TSP-Dec-Solver*(G, w_0, T^*)
  - **then**
    - $w_0[e] \leftarrow w[e]$
    - $H \leftarrow H \cup \{e\}$

return (H)

Proof of Correctness

Clearly H contains a hamiltonian cycle of minimum weight T^* at the end of the algorithm (note that H just consists of the edges that are not deleted from G). We claim that H is precisely a hamiltonian cycle.

Suppose not; then $C \cup \{e\} \subseteq H$, where C is a hamiltonian cycle of weight T^* and $e \in G \setminus C$. Consider the iteration when e was added to H. Let G' denote the graph G at this point in time. G' contains a hamiltonian cycle of weight T^* but $G' \setminus \{e\}$ does not, so e is included in H. We are assuming that

$$C \cup \{e\} \subseteq H,$$

which implies

$$C \subseteq H \setminus \{e\}.$$

Since $H \subseteq G'$, we have

$$C \subseteq H \setminus \{e\} \subseteq G' \setminus \{e\}.$$

Therefore e would not have been added to H, which is a contradiction.

Certificates

Certificate: Informally, a certificate for a yes-instance I is some "extra information" C which makes it easy to **verify** that I is a yes-instance.

Certificate Verification Algorithm: Suppose that *Ver* is an algorithm that verifies certificates for yes-instances. Then $\mathit{Ver}(I, C)$ outputs "yes" if I is a yes-instance and C is a valid certificate for I. If $\mathit{Ver}(I, C)$ outputs "no", then either I is a no-instance, or I is a yes-instance and C is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm *Ver* is a polynomial-time certificate verification algorithm if the complexity of *Ver* is $O(n^k)$, where k is a positive integer and $n = \mathit{Size}(I)$.

The Complexity Class NP

Certificate Verification Algorithm: A certificate verification algorithm *Ver* is said to **solve** a decision problem Π provided that

- **for every** yes-instance I, **there exists** a certificate C such that *Ver*(I, C) outputs "yes".
- **for every** no-instance I and **for every** certificate C, *Ver*(I, C) outputs "no".

The Complexity Class NP denotes the set of all decision problems that have polynomial-time certificate verification algorithms solving them. We write $\Pi \in$ **NP** if the decision problem Π is in the complexity class **NP**.

Finding Certificates vs Verifying Certificates: It is **not required** to be able to **find** a certificate C for a yes-instance in polynomial time in order to say that a decision problem $\Pi \in$ **NP**.

Important Fact: **P** $\subseteq$ **NP**.

Certificate Verification Algorithm for Hamiltonian Cycle

A certificate consists of an n-tuple, $X = [x_1, \ldots, x_n]$, that might be a hamiltonian cycle for a given graph $G = (V, E)$ (where $n = |V|$).

Algorithm: *Hamiltonian Cycle Certificate Verification*(G, X)

$flag \leftarrow$ **true**
$Used \leftarrow \{x_1\}$
$j \leftarrow 2$
while $(j \leq n)$ **and** $flag$
do
- $flag \leftarrow (x_j \notin Used)$ **and** $(\{x_{j-1}, x_j\} \in E)$
- **if** $(j = n)$ **then** $flag \leftarrow flag$ **and** $(\{x_n, x_1\} \in E)$
- $Used \leftarrow Used \cup \{x_j\}$
- $j \leftarrow j + 1$

return $(flag)$

Polynomial Transformations

For a decision problem Π, let $\mathcal{I}(\Pi)$ denote the set of all instances of Π. Let $\mathcal{I}_{\textbf{yes}}(\Pi)$ and $\mathcal{I}_{\textbf{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of Π.

Suppose that Π_1 and Π_2 are decision problems. We say that there is a **polynomial transformation** from Π_1 to Π_2 (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \to \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

- $f(I)$ is computable in polynomial time (as a function of *size*(I), where $I \in \mathcal{I}(\Pi_1)$)
- if $I \in \mathcal{I}_{\textbf{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\textbf{yes}}(\Pi_2)$
- if $I \in \mathcal{I}_{\textbf{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\textbf{no}}(\Pi_2)$

Polynomial Transformations (cont.)

Polynomial transformations are also known as **Karp reductions** or **many-one reductions**.

A polynomial transformation can be thought of as a (simple) special case of a polynomial-time Turing reduction, i.e., if $\Pi_1 \leq_P \Pi_2$, then $\Pi_1 \leq_P^T \Pi_2$.

Given a polynomial transformation f from Π_1 to Π_2, the corresponding Turing reduction is as follows:

1. Given $I \in \mathcal{I}(\Pi_1)$, construct $f(I) \in \mathcal{I}(\Pi_2)$.
2. Given an oracle for Π_2, say A, run $A(f(I))$.

We transform the instance, and then make a single call to the oracle.

Very important point: We do not know whether I is a yes-instance or a no-instance of Π_1 when we transform it to an instance $f(I)$ of Π_2.

To prove the implication "if $I \in \mathcal{I}_{\mathbf{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\mathbf{no}}(\Pi_2)$", we usually prove the contrapositive statement "if $f(I) \in \mathcal{I}_{\mathbf{yes}}(\Pi_2)$, then $I \in \mathcal{I}_{\mathbf{yes}}(\Pi_1)$.

Two Graph Theory Decision Problems

Problem 7.8

Clique

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a clique of size $\geq k$? (A clique is a subset of vertices $W \subseteq V$ such that $uv \in E$ for all $u, v \in W$, $u \neq v$.)*

Problem 7.9

Vertex Cover

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a vertex cover of size $\leq k$? (A vertex cover is a subset of vertices $W \subseteq V$ such that $\{u, v\} \cap W \neq \emptyset$ for all edges $uv \in E$.)*

Clique $\leq_P$ Vertex-Cover

Suppose that $I = (G, k)$ is an instance of **Clique**, where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ and $1 \leq k \leq n$.

Construct an instance $f(I) = (H, \ell)$ of **Vertex Cover**, where $H = (V, F)$, $\ell = n - k$ and

$$v_i v_j \in F \Leftrightarrow v_i v_j \notin E.$$

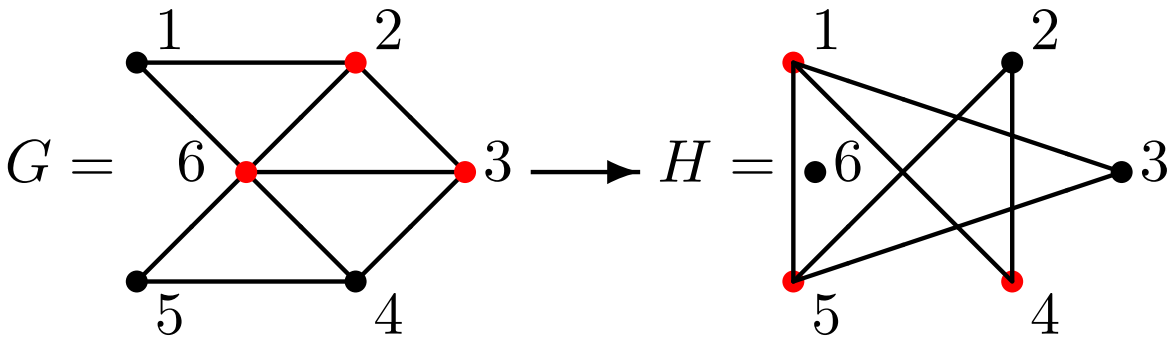
H is called the **complement** of G, because every edge of G is a non-edge of H and every non-edge of G is an edge of H.

We have $\mathit{Size}(I) = n^2 + \log_2 k \in \Theta(n^2)$ Computing H takes time $\Theta(n^2)$ and computing ℓ takes time $\Theta(\log n)$, so $f(I)$ can be computed in time $\Theta(\mathit{Size}(I))$, which is polynomial time.

Clique $\leq_P$ Vertex-Cover (cont.)

Suppose I is a yes-instance of **Clique**. Therefore there exists a set of k vertices W such that $uv \in E$ for all $u, v \in W$. Define $W' = V \setminus W$. Clearly $|W'| = n - k = \ell$. We claim that W' is a vertex cover of H. Suppose $uv \in F$ (so $uv \notin E$). If $\{u, v\} \cap W' \neq \emptyset$, we're done, so assume $u, v \notin W'$. Therefore $u, v \in W$. But $uv \notin E$, so W is not a clique. This is a contradiction and hence $f(I)$ is a yes-instance of **Vertex Cover**.

Suppose $f(I)$ is a yes-instance of **Vertex Cover**. Therefore there exists a set of $\ell = n - k$ vertices W' that is a vertex cover of H. Define $W = V \setminus W'$. Clearly $|W| = k$. We claim that W is a clique in G ....



Properties of Polynomial-time Transformations

Theorem 7.10

If Π_1 *and* Π_2 *are decision problems,* $\Pi_1 \leq_P \Pi_2$ *and* $\Pi_2 \in \mathbf{P}$*, then* $\Pi_1 \in \mathbf{P}$*.*

Proof.

Suppose A is a poly-time algorithm for Π_2, having complexity $O(m^\ell)$ on an instance of size m. Suppose f is a transformation from Π_1 to Π_2 having complexity $O(n^k)$ on an instance of size n. We solve Π_1 as follows:

1. Given $I \in \mathcal{I}(\Pi_1)$, construct $f(I) \in \mathcal{I}(\Pi_2)$.
2. Run $A(f(I))$.

It is clear that this yields the correct answer. We need to show that these two steps can be carried out in polynomial time as a function of $n = \mathit{Size}(I)$. Step (1) can be executed in time $O(n^k)$ and it yields an instance $f(I)$ having size $m \in O(n^k)$. Step (2) takes time $O(m^\ell)$. Since $m \in O(n^k)$, the time for step (2) is $O(n^{k\ell})$, as is the total time to execute both steps.

Properties of Polynomial-time Transformations (cont.)

Theorem 7.11

Suppose that Π_1, Π_2 *and* Π_3 *are decision problems. If* $\Pi_1 \leq_P \Pi_2$ *and* $\Pi_2 \leq_P \Pi_3$, *then* $\Pi_1 \leq_P \Pi_3$.

Proof.

We have a polynomial transformation f from Π_1 to Π_2, and another polynomial transformation g from Π_2 to Π_3. We define $h = f \circ g$, i.e., $h(I) = g(f(I))$ for all instances I of Π_1. (Exercise: fill in the details.) □

The Complexity Class NPC

The complexity class **NPC** denotes the set of all decision problems Π that satisfy the following two properties:

- $\Pi \in$ **NP**
- For all $\Pi' \in$ **NP**, $\Pi' \leq_P \Pi$.

NPC is an abbreviation for **NP-complete**.

Note that the definition does not imply that NP-complete problems exist!

The Complexity Class NPC (cont.)

Theorem 7.12

If $\mathbf{P} \cap \mathbf{NPC} \neq \emptyset$, *then* $\mathbf{P} = \mathbf{NP}$.

Proof.

We know that $\mathbf{P} \subseteq \mathbf{NP}$, so it suffices to show that $\mathbf{NP} \subseteq \mathbf{P}$. Suppose $\Pi \in \mathbf{P} \cap \mathbf{NPC}$ and let $\Pi' \in \mathbf{NP}$. We will show that $\Pi' \in \mathbf{P}$.

1. Since $\Pi' \in \mathbf{NP}$ and $\Pi \in \mathbf{NPC}$, it follows that $\Pi' \leq_P \Pi$ (definition of NP-completeness).
2. Since $\Pi' \leq_P \Pi$ and $\Pi \in \mathbf{P}$, it follows that $\Pi' \in \mathbf{P}$ (see Theorem 7.10 on slide # 279).

□

Satisfiability and the Cook-Levin Theorem

Problem 7.13

CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables $x_1, \ldots, x_n$, such that F is the* **conjunction** *(logical "and") of m* **clauses**, *where each clause is the* **disjunction** *(logical "or") of literals. (A* **literal** *is a boolean variable or its negation.)*

Question: *Is there a truth assignment such that F evaluates to* `true`*?*

Theorem 7.14 (Cook-Levin Theorem)

CNF-Satisfiability $\in$ **NPC**.

Proving Problems NP-complete

Now, given any NP-complete problem, say Π_1, other problems in **NP** can be proven to be NP-complete via polynomial transformations from Π_1, as stated in the following theorem:

Theorem 7.15

Suppose that the following conditions are satisfied:

- $\Pi_1 \in$ **NPC**,
- $\Pi_1 \leq_P \Pi_2$, *and*
- $\Pi_2 \in$ **NP**.

Then $\Pi_2 \in$ **NPC**.

More Satisfiability Problems

Problem 7.16

3-CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **three** *literals.*

Question: *Is there a truth assignment such that F evaluates to* `true`*?*

Problem 7.17

2-CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **two** *literals.*

Question: *Is there a truth assignment such that F evaluates to* `true`*?*

3-CNF-Satisfiability $\in$ **NPC**, while **2-CNF-Satisfiability** $\in$ **P**.

CNF-Satisfiability $\leq_P$ 3-CNF-Satisfiability

Suppose that $(X, \mathcal{C})$ is an instance of **CNF-SAT**, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{C} = \{C_1, \ldots, C_m\}$. For each C_j, do the following:

case 1 If $|C_j| = 1$, say $C_j = \{z\}$, construct four clauses

$$\{z, a, b\}, \{z, a, \overline{b}\}, \{z, \overline{a}, b\}, \{z, \overline{a}, \overline{b}\}.$$

case 2 If $|C_j| = 2$, say $C_j = \{z_1, z_2\}$, construct two clauses

$$\{z_1, z_2, c\}, \{z_1, z_2, \overline{c}\}.$$

case 3 If $|C_j| = 3$, then leave C_j unchanged.

case 4 If $|C_j| \geq 4$, say $C_j = \{z_1, z_2, \ldots, z_k\}$, then construct $k - 2$ new clauses

$$\{z_1, z_2, d_1\}, \{\overline{d_1}, z_3, d_2\}, \{\overline{d_2}, z_4, d_3\}, \ldots,$$
$$\{\overline{d_{k-4}}, z_{k-2}, d_{k-3}\}, \{\overline{d_{k-3}}, z_{k-1}, z_k\}.$$

Correctness of the Transformation

Suppose I is a yes-instance of **CNF-SAT**. We show that $f(I)$ is a yes-instance of **3-CNF-SAT**. Fix a truth assignment for X in which every clause contains a true literal. We consider each clause C_j of the instance I.

1. If $C_j = \{z\}$, then z must be true. The corresponding four clauses in $f(I)$ each contain z, so they are all satisfied.
2. If $C_j = \{z_1, z_2\}$, then at least one of the z_1 or z_2 is true. The corresponding two clauses in $f(I)$ each contain z_1, z_2, so they are both satisfied.
3. If $C_j = \{z_1, z_2, z_3\}$, then C_j occurs unchanged in $f(I)$.
4. Suppose $C_j = \{z_1, z_2, z_3, \ldots, z_k\}$ where $k > 3$ and suppose $z_t \in C_j$ is a true literal. Define $d_i =$ **true** for $1 \leq i \leq t-2$ and define $d_i =$ **false** for $t-1 \leq i \leq k$. It is straightforward to verify that the $k-2$ corresponding clauses in $f(I)$ each contain a true literal.

Correctness of the Transformation (cont.)

Conversely, suppose $f(I)$ is a yes-instance of **3-CNF-SAT**. We show that I is a yes-instance of **CNF-SAT**.

1. Four clauses in $f(I)$ having the form $\{z, a, b\}$, $\{z, a, \overline{b}\}$, $\{z, \overline{a}, \overline{b}\}$ $\{z, \overline{a}, \overline{b}\}$ are all satisfied if and only if $z =$ **true**. Then the corresponding clause $\{z\}$ in I is satisfied.
2. Two clauses in $f(I)$ having the form $\{z_1, z_2, c\}$, $\{z_1, z_2, \overline{c}\}$ are both satisfied if and only if at least one of $z_1, z_2 =$ **true**. Then the corresponding clause $\{z_1, z_2\}$ in I is satisfied.
3. If $C_j = \{z_1, z_2, z_3\}$ is a clause in $f(I)$, then C_j occurs unchanged in I.

Correctness of the Transformation (cont.)

4. Finally, consider the $k-2$ clauses in $f(I)$ arising from a clause $C_j = \{z_1, z_2, z_3, \ldots, z_k\}$ in I, where $k > 3$. We show that at least one of $z_1, z_2, \ldots, z_k = \textbf{true}$ if all $k-2$ of these clauses contain a true literal.
   Assume all of $z_1, z_2, \ldots, z_k = \textbf{false}$. In order for the first clause to contain a true literal, $d_1 = \textbf{true}$. Then, in order for the second clause to contain a true literal, $d_2 = \textbf{true}$. This pattern continues, and in order for the second last clause to contain a true literal, $d_{k-3} = \textbf{true}$. But then the last clause contains no true literal, which is a contradiction.
   We have shown that at least one of $z_1, z_2, \ldots, z_k = \textbf{true}$, which says that the clause $\{z_1, z_2, z_3, \ldots, z_k\}$ contains a true literal, as required.

3-CNF-Satisfiability $\leq_P$ Clique

Let I be the instance of **3-CNF-SAT** consisting of n variables, $x_1, \ldots, x_n$, and m clauses, $C_1, \ldots, C_m$. Let $C_i = \{z_1^i, z_2^i, z_3^i\}$, $1 \leq i \leq m$.

Define $f(I) = (G, k)$, where $G = (V, E)$ according to the following rules:

- $V = \{v_j^i : 1 \leq i \leq m, 1 \leq j \leq 3\}$,
- $v_j^i v_{j'}^{i'} \in E$ if and only if $i \neq i'$ and $z_j^i \neq \overline{z_{j'}^{i'}}$, and
- $k = m$.

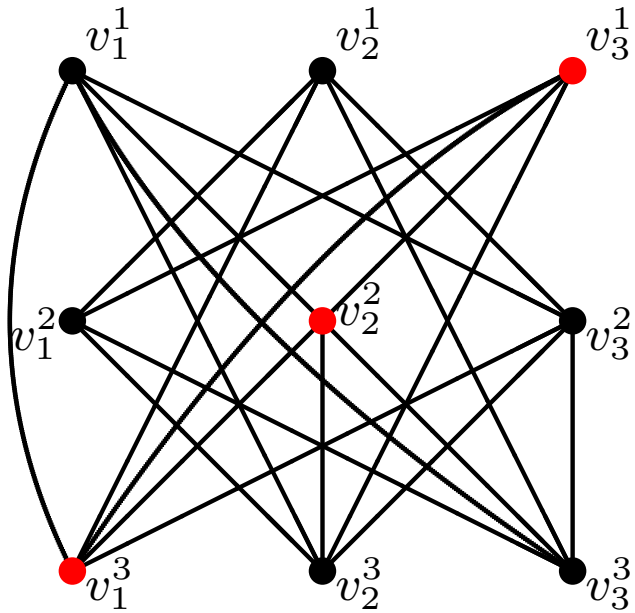
Non-edges of the constructed graph correspond to

1. "inconsistent" truth assignments of literals from two different clauses; or
2. any two literals in the same clause.

Example

$$I : \begin{cases} C_1 = \{x_1, \overline{x_2}, \overline{x_3}\} \\ C_2 = \{\overline{x_1}, x_2, x_3\} \\ C_3 = \{x_1, x_2, x_3\} \end{cases} \qquad x_1 = \textbf{true}, x_2 = \textbf{true}, x_3 = \textbf{false}$$

$f(I)$:



Subset Sum

Problem 7.18

Subset Sum

Instance: *A list of* **sizes** $S = [s_1, \ldots, s_n]$*; and a* **target sum**, T*. These are all positive integers.*

Question: *Does there exist a subset* $J \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in J} s_i = T$*?*

Vertex Cover $\leq_P$ Subset Sum

Suppose $I = (G, k)$, where $G = (V, E)$, $|V| = n$, $|E| = m$ and $1 \leq k \leq n$.

Suppose $V = \{v_1, \ldots, v_n\}$ and $E = \{e_0, \ldots, e_{m-1}\}$. For $1 \leq i \leq n$, $0 \leq j \leq m-1$, let $C = (c_{ij})$, where

$$c_{ij} = \begin{cases} 1 & \text{if } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise.} \end{cases}$$

Define $n + m$ sizes and a target sum W as follows:

$$\begin{aligned} a_i &= 10^m + \sum_{j=0}^{m-1} c_{ij} 10^j \quad (1 \leq i \leq n) \\ b_j &= 10^j \quad (0 \leq j \leq m-1) \\ W &= k \cdot 10^m + \sum_{j=0}^{m-1} 2 \cdot 10^j \end{aligned}$$

Then define $f(I) = (a_1, \ldots, a_n, b_0, \ldots, b_{m-1}, W)$.

Correctness of the Transformation

Suppose I is a yes-instance of **Vertex Cover**. There is a vertex cover $V' \subseteq V$ such that $|V'| = k$. For $i = 1, 2$, let E^i denote the edges having exactly i vertices in V'. Then $E = E^1 \cup E^2$ because V' is a vertex cover. Let

$$A' = \{a_i : v_i \in V'\} \quad \text{and} \quad B' = \{b_j : e_j \in E^1\}.$$

The sum of the sizes in A' is

$$k \cdot 10^m + \sum_{\{j:e_j \in E^1\}} 10^j + \sum_{\{j:e_j \in E^2\}} 2 \times 10^j.$$

The sum of the sizes in B' is

$$\sum_{\{j:e_j \in E^1\}} 10^j.$$

Therefore the sum of all the chosen sizes is

$$k \cdot 10^m + \sum_{\{j:e_j \in E\}} 2 \cdot 10^j = k \cdot 10^m + \sum_{j=1}^{m} 2 \cdot 10^j = W.$$

Correctness of the Transformation (cont.)

Conversely, suppose $f(I)$ is a yes-instance of **Subset Sum**. We show that I is a yes-instance of **Vertex Cover**. Let $A' \cup B'$ be the subset of chosen sizes. Define $V' = \{v_i : a_i \in A'\}$. We claim that V' is a vertex cover of size k. In order for the coefficient of 10^m to be equal to k, we must have $|V'| = k$ (there can't be any carries occurring). The coefficient of any other term 10^j $(0 \leq j \leq m-1)$ must be equal to 2. Suppose that $e_j = v_i v_{i'}$. There are two possible situations that can occur:

1. a_i and $a_{i'}$ are both in A'. Then V' contains both vertices incident with e_j.
2. exactly one of a_i or $a_{i'}$ is in A' and $b_j \in B'$. In this case, V' contains exactly one vertex incident with e_j.

In both cases, e_j is incident with at least one vertex in V'.

Subset Sum $\leq_P$ 0-1 Knapsack

Let I be an instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_n]$ and target sum T.

Define

$$p_i = s_i,\ 1 \leq i \leq n$$
$$w_i = s_i,\ 1 \leq i \leq n$$
$$M = T$$

Then define $f(I)$ to be the instance of **0-1 Knapsack** consisting of profits $[p_1, \ldots, p_n]$, weights $[w_1, \ldots, w_n]$, capacity M and target profit T.

Exercise: Prove the correctness of this transformation.

Hamiltonian Cycle $\leq_P$ TSP-Dec

Let I be an instance of **Hamiltonian Cycle** consisting of a graph $G = (V, E)$.

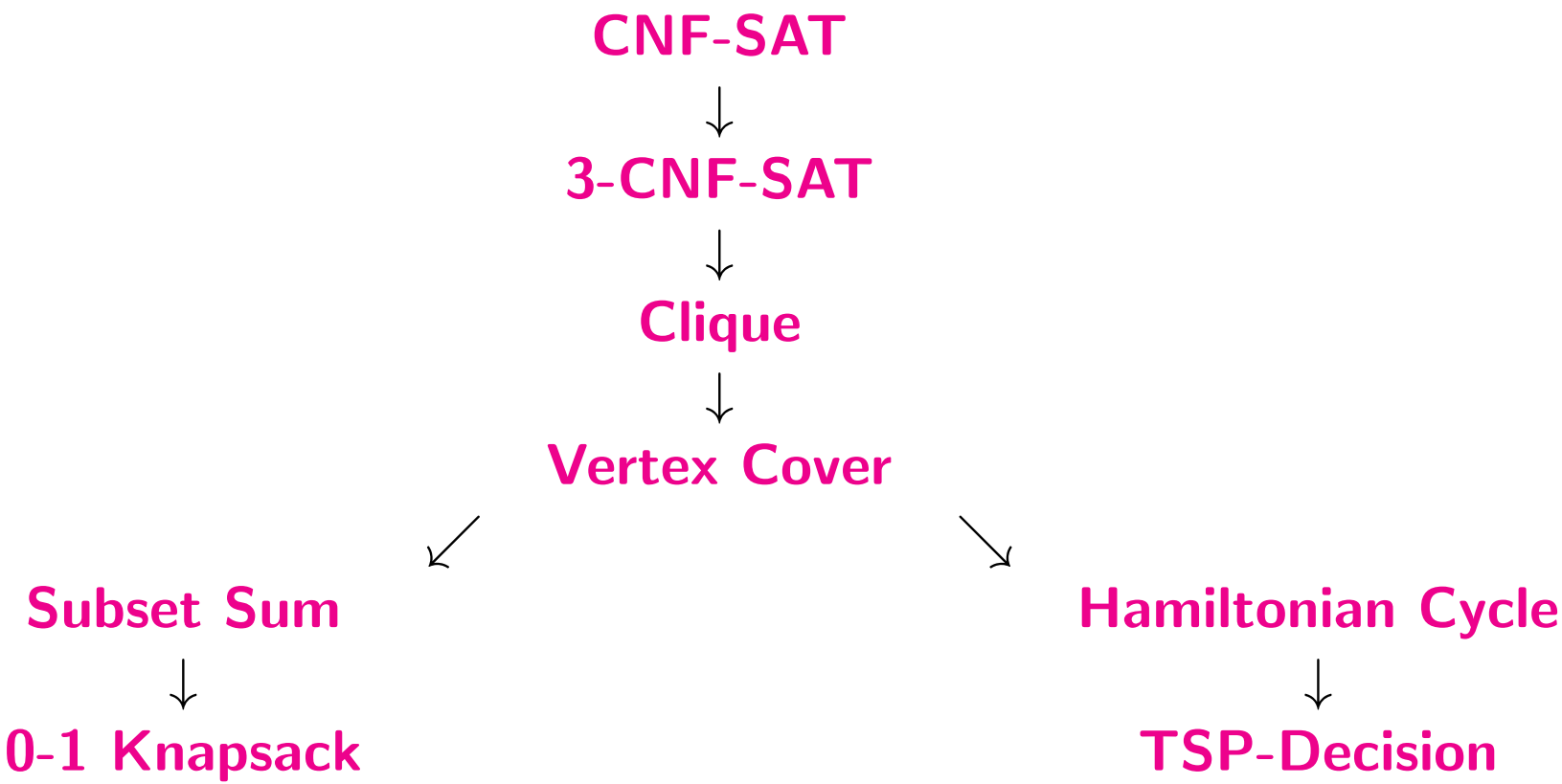
For the complete graph K_n, where $n = |V|$, define edge weights as follows:

$$w(uv) = \begin{cases} 1 & \text{if } uv \in E \\ 2 & \text{if } uv \notin E. \end{cases}$$

Then define $f(I)$ to be the instance of **TSP-Dec** consisting of the graph K_n, edge weights w and target $T = n$.

Exercise: Prove the correctness of this transformation.

Summary of Polynomial Transformations



In the above diagram, arrows denote polynomial transformations. The transformation **Vertex Cover** $\leq_P$ **Hamiltonian Cycle** is complicated and is described in a supplementary note.

NP-hard Problems

A problem Π is **NP-hard** if there exists a problem $\Pi' \in$ **NPC** such that $\Pi' \leq_P^T \Pi$.

Every NP-complete problem is automatically NP-hard, but there exist NP-hard problems that are not NP-complete.

Typical examples of NP-hard problems are optimization problems corresponding to NP-complete decision problems.

For example, **TSP-Decision** $\leq_P^T$ **TSP-Optimization** and **TSP-Decision** $\in$ **NPC**, so **TSP-Optimization** is NP-hard.

This is a "trivial" Turing reduction; the reduction in the reverse direction, which was given on slide # 269, is more complex.

Undecidable Problems

A decision problem Π is **undecidable** if there does not exist an algorithm that solves Π.

If Π is undecidable, then for every algorithm A, there exists at least one instance $I \in \mathcal{I}(\Pi)$ such that $A(I)$ does not find the correct answer ("yes" or "no") in finite time.

Problem 7.19

Halting

Instance: *A computer program A and input x for the program A.*

Question: *When program A is executed with input x, will it halt in finite time?*

Undecidability of the Halting Problem

Suppose that *Halt* is a program that solves the **Halting Problem**.
Consider the following algorithm *Strange*.

Algorithm: *Strange*(A)
external *Halt*
if not *Halt*(A, A)
then return (!)
else $\begin{cases} i \leftarrow 1 \\ \textbf{while } i \neq 0 \textbf{ do } i \leftarrow i + 1 \end{cases}$

What happens when we run *Strange*(*Strange*)?

Undecidability of the Halting Problem (cont.)

The statement "*Halt* solves the Halting problem" means that

$$\mathit{Halt}(A, x) = \begin{cases} \textbf{true} & \text{if } A(x) \text{ halts} \\ \textbf{false} & \text{if } A(x) \text{ doesn't halt.} \end{cases}$$

Note that A (the "algorithm") and I (the "input" to A) are both strings over some finite alphabet.

What happens when we run *Strange*(*Strange*)?

We have

$$\begin{aligned} \mathit{Strange}(\mathit{Strange}) \text{ halts} &\Leftrightarrow \mathit{Halt}(\mathit{Strange}, \mathit{Strange}) = \textbf{false} \\ &\Leftrightarrow \mathit{Strange}(\mathit{Strange}) \text{ does not halt.} \end{aligned}$$

The only conclusion we can make is that the program *Halt* does not exist!

Another Undecidable Problem

Here is another example of an undecidable problem. The problem **Halt-All** takes a program A as input and asks if A halts on all inputs x.

We describe a Turing reduction **Halting** $\leq^T$ **Halt-All**, which proves that **Halt-All** is undecidable.

Assume we have a program *HaltAllSolver*.

For a fixed program A and input x, let $B_x()$ be the program that executes $A(x)$ (so B_x has no input).

Here is the reduction:

1. Given A and x (an instance of **Halting**), construct the program B_x.
2. Run *HaltAllSolver*(B_x),

We have

$$\textit{HaltAllSolver}(B_x) = \texttt{true} \quad \Leftrightarrow \quad A(x) \text{ halts},$$

so we can solve the halting problem.

The Post Correspondence Problem

The following problem is also undecidable.

Problem 7.20

Post Correspondence

Instance: *two finite lists* $\alpha_1, \ldots, \alpha_N$ *and* $\beta_1, \ldots, \beta_N$ *of words over some alphabet* A *of size* ≥ 2.

Question: *Does there exist a finite list of indices, say* $i_1, \ldots, i_K$, *where* $i_j \in \{1, \ldots, N\}$ *for* $1 \leq j \leq K$, *such that*

$$\alpha_{i_1} \cdots \alpha_{i_K} = \beta_{i_1} \cdots \beta_{i_K},$$

where a "product" of words denotes their concatenation.