CS 341 Algorithms. Spring 2021. University of Waterloo

Lecture 1: Course Introduction

We will first go through some logistics of the course, and then have an overview of the course.

Then, we start with defining time complexity, and then work out a simple and interesting problem.

Course Information

The course page address is https://cs.uwaterloo.ca/~lapchi/cs341/.

The course information/syllabus is posted under the notes section.

We have a piazza page for discussions and Q&A; see the course page.

We will provide handwritten course notes and they are usually posted a day before the lecture

We will have online live lectures on every Tuesday and Thursday 11am - 12:30pm EST.

The lectures will be recorded and posted for you to view afterwards.

We will also post the slides (with annotations) on the course page.

We use three reference books and refer to them using the following shorthands.

- [DPV] Algorithms, by Dasgupta, Papadimitriou and Vazirani.
- [KT] Algorithm design, by Kleinberg and Tardos.
- [CLRS] Introduction to algorithms, by Cormen, Leiserson, Rivest, and Stein.

Course Overview

The main focus of the course is on the design and analysis of efficient algorithms, and these are fundamental building blocks in the development of Computer Science.

Towards the end of the course, however, we will realize that we do not have efficient algorithms for many interesting and important problems, and we will introduce the theory of NP-completeness to explain this phenomenon formally.

To develop an efficient algorithm that is useful in practice, there are a few steps.

First, we need to understand the structures and mathematical properties of the problem (e.g. a nice recurrence relation, optimal solution is unique, etc).

Then, we use these observations to design algorithms and prove that the algorithms are correct and analyze the time complexity.

Finally, we may use some good data structures to speed up the operations, and

also implement it carefully to optimize the performance

This course is theoretically oriented. We will focus on the first two steps and spend most of the time in mathematical proofs.

For data structures, the standard ones that you have learnt (e.g. queue, stack, heap, balanced search trees) will be enough for the purpose of this course, although keep in mind that some of the fastest algorithms heavily rely on the use of sophisticated data structures.

There will be some programming problems in the assignments to practice your implementation skills.

Syllabus

We will learn basic techniques to design and analyze algorithms, through the study of various problem in different topics.

The tentative schedule include:

- divide and conquer and solving recurrence (3-4 lectures)
- simple graph algorithms using BFS and DFS (3-4 lectures)
- greedy algorithms (3 lectures)
- dynamic programming (4 lectures)
- bipartite matching (3 lectures)
- NP-completeness and reductions (4 lectures)

The concept of reduction is important in designing algorithms as well as showing hardness and intractability.

Two Classical Problems

To give you a more concrete idea about the course, let's consider the following two problems.

We are given an undirected graph with n vertices and m edges, where each edge has a non-negative cost.

The traveling salesman problem asks us to find a minimum cost tour to visit every vertex of the graph at least once (visit all cities).

The Chinese postman problem asks us to find a minimum cost tour to visit every edge of the graph at least once (visit all streets).

A naive algorithm to solve the traveling salesman problem is to enumerate all permutations and return the minimum cost one. This takes $O(n!)$ time which is way too slow (e.g. can't solve more than 15 vertices).

Using dynamic programming, we can solve it in $O(2^n)$ time, and this is essentially the best known algorithm that we know of.

We will prove that this problem is "NP-complete", and probably efficient algorithms for this problem do not exist.

Surprisingly, the Chinese postman problem, which looks very similar, can be solved in $O(n^4)$ time, using techniques from graph matching.

Learning Outcome

- Know well-known algorithms well.
- Have the skills to design new algorithms for simple problems.
- Can prove correctness and analyze the time complexity of an algorithm.
- Use reductions to solve problems and to prove hardness.

Time Complexity

How do we define the time complexity of an algorithm?

Roughly speaking, we count the number of operations that the algorithm requires.

One may count exactly how many operations, say $100n^2$ comparisons to sort n numbers.

The precise constant is probably machine-dependent (depending on the primitive operations that are supported) and may be also difficult to work out.

So, the standard practice is to use asymptotic time complexity to analyze algorithms.

Asymptotic Time Complexity

Given two functions $f(n)$, $g(n)$, we say

- (upper bound, big-O) $g(n) = O(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c$ for some constant c (independent of n).
- (lower bound, big-Ω) $g(n) = \Omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \geq c$ for some constant c.
- (same order, big-Θ) $g(n) = \Theta(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some constant c.
- (loose upper bound, small-o) $g(n) = o(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
- (loose lower bound, small-ω) $g(n) = \omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

$n \to \infty$ $f(n)$

Some examples: $100n^2 = \Theta(n^2)$, $2n^3 + \frac{n^3}{\log n} = \Theta(n^3)$, $n\log n = O(n^2)$, $\frac{n^3}{\log\log n} = o(n^3)$, $2^n = o(n^n)$.

Questions: How to compare $2^{\sqrt{n}}$ vs $n^{\log n}$, $\sqrt{n}$ vs $2^{\sqrt{\log n}}$, $2^{n^{1.1}}$ vs $n!$?

(We won't see these tricky running time bounds in this course, but they appear in research.)

Side remark: Some people think that it is more appropriate to write say $n^2 \in O(n^3)$, thinking of $O(n^3)$ as a set of functions.

Some people write say $n^2 \leq O(n^3)$ and $n^2 \geq \Omega(n)$ to highlight their relations.

Both are acceptable in this course.

Worst Case Complexity

We say an algorithm has time complexity $O(f(n))$ if it requires at most $O(f(n))$ primitive operations for all inputs of size n (e.g. n bits, n numbers, n vertices, etc).

By adopting the asymptotic time complexity, we are ignoring the leading constant and lower order terms.

For example, we will say that an algorithm with running time $2^{2^{100}} n^2$ is faster than another algorithm with running time n^3, although for all practical purposes the n^3 algorithm is faster.

First, this rarely happens (but it does happen in research papers), and even if it happens usually the (quadratic) algorithm can be improved to have much smaller coefficient.

More importantly, the $2^{2^{100}} n^2$ time algorithm does run faster than the n^3 time algorithm when n is large enough, and it is inherently more efficient.

Also, when our computers become faster, the problem size that we can deal with will become larger, and this asymptotic analysis would be more reasonable.

So, even though this asymptotic analysis may not be a very accurate measure of the algorithm's practical performance, it usually is a good measure and it makes sense theoretically.

"Good" Algorithms

For most optimization problems, such as the traveling salesman problem, there is a straightforward exponential time algorithm.

For those problems we are most interested in designing a polynomial time algorithm, with running time $O(n^c)$ for some constant c independent of n.

Again, an $10^{10^{10}} n^{100}$-time algorithm would not run faster than a 2^n-time algorithm in our lifetime, but still it is a significant achievement because it tells us that this problem has some fundamental properties that allow us to solve it faster than brute-force enumeration, and this distinguishes the problem from other problems that brute-force enumeration is essentially the best algorithm.

We will come back to polynomial-time computations towards the end of this course.

Computation models

When we say that we have an $O(n^2)$-time algorithm, we need to be more precise about what are the primitive operations that we assume.

In this course, we usually assume the word-RAM model, in which we assume that we can access an arbitrary position of an array in constant time, and also that each word operation (such as addition, read/write) can be done in constant time.

For example, in a graph with n vertices, we need to use $\log_2 n$ bits to identify a vertex, but we usually just assume that these $\log_2 n$ bits can be fit into one word and consider an operation such as comparing the labels of two vertices can be done in constant time.

This model is usually good enough in practice, because the problem size can usually be fit in the main memory, and so the word size is large enough and each memory access can be done in more or less the same time.

For example, we can do binary search in a word-RAM model in $O(\log n)$ word operations, while in a Turing machine (with a one-dimensional tape) we cannot do binary search.

A more practical scenario is when we analyze numerical algorithms such as computing the determinant. As you may know, by doing Gaussian elimination in an arbitrary manner, the intermediate numbers could blow up exponentially, and so it is not reasonable anymore to assume that each arithmetic operation can be done in constant time.

For those problems, we usually consider the bit-complexity, i.e. how many bit operations involved in the algorithm.

So, please pay some attention to these assumptions when we analyze the time complexity of an algorithm, especially for numerical problems.

That said, the word-complexity and bit-complexity usually don't make a big difference

in this course (e.g. at most a $\log(n)$ factor) and so we just use the word-RAM model.

3SUM

Let's do a warmup problem to see the process of designing and analyzing algorithms.
This kind of ad-hoc problems are often asked in job interviews at tech companies.

The 3-SUM problem is simple to state: we are given $n+1$ numbers $a_1, a_2, \ldots, a_n$ and c and we would like to determine if there are i, j, k such that $a_i + a_j + a_k = c$.

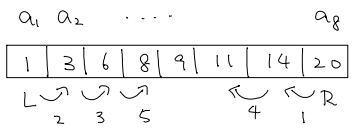
Algorithm 1: Enumerate all triples and check whether its sum is c. Time: $O(n^3)$.

Algorithm 2: Observe that $a_i + a_j + a_k = c$ can be rewritten as $c - a_i - a_j = a_k$
We can enumerate all pairs $a_i + a_j$ and check whether $c - a_i - a_j$ is equal to some a_k.

To do this checking efficiently, we can first sort the n numbers so that $a_1 \le a_2 \le \ldots \le a_n$.
Then checking whether $c - a_i - a_j$ is equal to some a_k can be done by a binary search, in $O(\log n)$ time.
So, the total complexity is $O(\underset{\text{sorting}}{n\log n} + \underset{\text{binary search for each pair}}{n^2 \log n}) = O(n^2 \log n)$.

Algorithm 3: This time we write the condition to check as $a_i + a_j = c - a_k$.
Suppose again that we sort the n numbers so that $a_1 \le a_2 \le \ldots \le a_n$.
The idea is that given this sorted array and the number $b := c - a_k$, we can check whether there are i, j such that $a_i + a_j = b$.
In other words, the 2-SUM problem can be solved in $O(n)$ time given the sorted array.
If this is true, then we can obtain an $O(n^2)$-time algorithm for 3-SUM, by trying $b := c - a_k$ for each $1 \le k \le n$.
That is, we reduce the 3-SUM problem to n instances of the 2-SUM problem.

2-SUM: It remains to show that 2-SUM can be solved in $O(n)$ time given a sorted array.
To get an idea, let's see an example:



Algorithm

We keep a left index L and a right index R.

Initially, we set $L = 1$ and $R = n$.

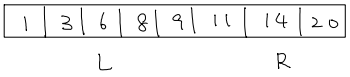
While $L \le R$

Check whether $a_L + a_R = c$. If so, we are done.

Otherwise, if $a_L + a_R > c$, then decrease R by 1;

else if $a_L + a_R < c$, then increase L by 1.

The idea is that if $a_L + a_R > 19$



then we know that $a_L + a_{R'} > 19$ for all $R' \ge R$, and so we don't need to check those pairs.

This is, of course, not a proof.

Proof of Correctness If there are no i, j such that $a_i + a_j = b$, then we won't find them.

Suppose $a_i + a_j = b$ for some $i \le j$.

Since the algorithm runs until $L = R$, there is an iteration such that $L = i$ or $R = j$.

Without loss, suppose that $L = i$ happens first, and $R > j$; the other case is symmetric.

As long as $R > j$, we have $a_i + a_R > a_i + a_j = b$, and so we will decrease R until $R = j$, and so the algorithm will find this pair. □

Time Complexity: $O(n)$, because $R - L$ decreases by one in each iteration, so the algorithm will stop within $n-1$ iterations. □

Remark: Suppose this is an exam problem. You will get at least half the marks if you give algorithm 2. You will get at least half the marks if you state algorithm 3 correctly but did not provide the correctness proof. You will get full marks if you do both. Note that you don't need to write precise code for the algorithm. It is okay if you describe the algorithm as in the above example.

Literature (optional) It has been a long standing open problem whether $O(n^2)$ is optimal for 3SUM. Many people have worked on this problem without success, and they started to conjecture that such algorithms do not exist, and use 3-SUM as a "hard" problem to prove that other problems do not have faster algorithms. The logic is: if a problem A can be solved faster than $O(n^2)$ time, then 3-SUM can be solved faster than $O(n^2)$ time, and so it is impossible.

We will talk much more about reductions when we study NP-completeness.

Recently, there is an $O(n^2/(\log n/\log\log n)^{2/3}) = o(n^2)$-time algorithm discovered, so

the strong form of the conjecture is not true, but it remains open say whether an $O(n^{1.999})$-time algorithm for 3-SUM is possible.
