

Course Introduction

August 28, 2003

1 Key Goals

The focus of this course is studying techniques for developing fast algorithms, particularly for **optimization problems**. For these problems there are typically a finite number of candidate solutions and the goal is to find an optimal solution (with **min cost** or **max gain**). These problems differ from sorting/searching type problems studied in CS 241 because the solution is not obvious. In fact, verifying whether a candidate solution is optimal often requires non-trivial work. Example optimization problems studied in CS 241 are the closest pair problem, the minimum spanning tree problem and the shortest path problem.

In CS 441T/539T, for each algorithm you design for an optimization problem you must:

- PROVE that it outputs a correct (i.e. valid and optimal) solution.
- Analyze the asymptotic time complexity.

You will learn techniques that enable you to design the fastest possible algorithm, prove it is correct, and, in some cases, prove that your algorithm is asymptotically optimal.

1.1 Some Areas Where Optimization Problems Frequently Occur

- Scheduling (e.g. CPU usage, manufacturing,)
- Graph Problems (e.g path planning, minimum spanning tree)
- Networking
- Resource Management (e.g. environmental control, balance costs and benefits)
- profit maximization
- data compression
- optimal paths for a plotter
- packing (e.g. trucks, warehouses)
- internet routing, and many more!

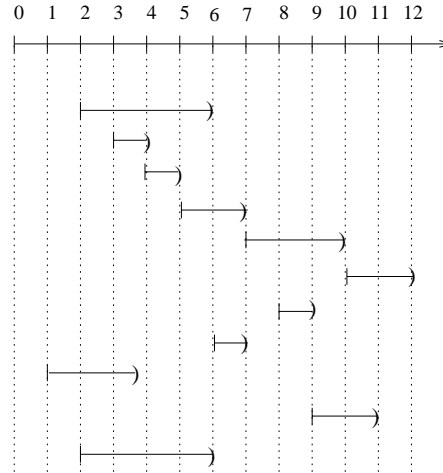
1.2 An Example Optimization Problem

Here's an example problem that we will consider as we begin the course. It is a "scheduling" problem in which the input is a set E_1, \dots, E_n of n events (or activities). There is a single resource that is needed by each event and in this simplified version of the problem, you must decide which events to schedule and which events to turn down. For $1 \leq i \leq n$, E_i is specified by a desired start time/arrival a_i and a length ℓ_i . Hence E_i is requesting to use the shared resource during the time interval $[a_i, a_i + \ell_i)$. The desired output from the scheduling algorithm is a set of events \mathcal{E} to schedule such that

1. no two jobs in \mathcal{E} run at the same time
2. $|\mathcal{E}|$ is the largest possible

Here's an example input:

	a	l	$f=a+l$
E_1	2	4	6
E_2	3	1	4
E_3	4	1	5
E_4	5	2	7
E_5	7	3	10
E_6	10	2	12
E_7	8	1	9
E_8	6	1	7
E_9	1	3	4
E_{10}	9	2	11
E_{11}	2	4	6



One (of many) interesting variations is as follows.

Input: For $1 \leq i \leq n$, $\langle a_i, l_i, d_i \rangle$ defining event E_i .

Output: For each i assign E_i a start time satisfying $a_i \leq s_i \leq a_i + d_i$ or $s_i = \infty$ (meaning that E_i is not scheduled) such that:

1. $\forall i$ such that $s_i \neq \infty$, $[s_i, s_i + l_i)$ are disjoint (i.e. scheduled jobs don't overlap).
2. The number of scheduled jobs (i.e. the number of $s_i \neq \infty$) is maximized.

How efficiently can these two problems be solved?

2 Model of Computation

The goal of algorithm design is to come up with the best possible algorithm. Practical constraints interfere with this goal, however. First, since algorithms are implemented on different hardware platforms, and using different programming languages, no clear winner may be possible. Especially, when parallel or distributed computers are included. Second, problems come in varying sizes (small to enormous). Different algorithms may be suitable at different sizes. We use the **asymptotic growth** rate of an algorithm as its performance metric. Typically, N will denote the problem size—number of input elements.

For a given value of input size, an algorithm will behave differently on different **instances** of input. The **best case** measures the performance of an algorithm when the problem instance is most favorable for the algorithm. The **worst case** occurs when the input is least favorable. The **average case** is the expected performance over all inputs.

2.1 Pros and Cons

- **Best Case** is mostly good for hype and advertising.
- **Average Case** frequently used, but subject to criticism because an algorithm designer rarely knows the distribution of inputs.

- **Worst Case** offers the most iron-clad guarantee: the algorithm will never exceed its predicted time. However, this bound can be overly pessimistic if the inputs are heavily skewed—the case of a few bad apples in the basket.

Worst-case is the metric of choice in computer science. It is always a good start in any case. When worst-case bounds are unpalatable (or show significant deviation from empirical behavior), people often resort to average case (when possible using an input distributions like what is seen “in practice”).

2.2 The Good, the Bad, and the Ugly

1. [**Good.**] An algorithm whose worst-case running time is bounded by a **polynomial** in the input size is **good**. Examples: $O(N)$, $O(N^{10})$, $O(\log N)$.
2. [**Bad.**] An algorithm whose worst-case running is **exponential** in input size is **bad**. Examples: $O(2^N)$, $O(N^N)$, $O(N!)$, $O(N^{\log N})$.
3. [**Ugly.**] Use class **ugly** for algorithms whose running time is **worse** than exponential. Examples: $O(2^{2^N})$, Ackermann’s function etc.

The cherished goal of algorithm design is to design good (polynomial time) algorithms.

2.3 Some Caveats

No mathematical model can reflect reality with perfect accuracy. Mathematical models are abstractions, and thus necessarily flawed. For instance, it is well known that the **unit-cost** arithmetic model of RAM (size of each real is treated as 1) can be abused by encoding horribly complicated expressions in large integers and solve intractable problems in polynomial time. But this violates the unwritten rule of good taste. A possible preventive measure is to use **log-cost** model (size of a real is the number of bits to encode it). However, it has been observed that when used as intended, the unit cost model reflects experimental observations more closely than the log cost model for data of moderate size, besides making the mathematical analysis a lot simpler. So, we will stick with the unit cost model.

An abusive use of asymptotic analysis can also lead to contrived and arcane solutions that may be superior by the measure of asymptotic complexity, but whose constant factors are so large or whose implementation will be so cumbersome that no improvement in technology would ever make them feasible.

Despite these shortcomings and occasional misuse, the asymptotic complexity occupies an unshakable position in our computer science consciousness, and, when used judiciously, has done more to guide us in improving technology in the design and analysis of algorithms than any other mathematical abstraction.

3 Polynomial vs. Exponential Time

Do all problems have polynomial time algorithms? Obviously not, since one can easily make up problems that demand an exponential size listing of the output. For instance, consider the problem of **enumerating** all permutations of $\{1, 2, \dots, N\}$. Every algorithm solving such a problem would necessarily spend an exponential time. We call such trivial problems **unnatural** and instead work with **natural** problems, which typically have a small (polynomial size) output, or even better, just a YES/NO answer.

The **holy grail** of Theory of Computing: Determine the complexity of all natural problems. That is, determine the best algorithm for each problem. Failing that, at least classify each problem in one of three categories: good, bad, or ugly.

Unfortunately, the **status** of a large number of very important optimization problems (and some other problems such as factoring) is **not known!** They are suspected to be so hard that no polynomial time algorithm can solve them, but no proof of this fact is known. A special complexity class has been designated for these problems: **NP (nondeterministic polynomial)**. The complexity class **P** is the class of problems which are solvable in polynomial time whereas informally **NP** is the class of problems in which a provided solution can be verified in polynomial time. The most famous theory of computing problem of the past 30 years: Does **P = NP**?

This course will introduce the theory of **NP-Completeness**. NP-Complete is a class of problems with the following surprising equivalence property: If any of these problems can be solved in polynomial time, then they all can. If any of them can be shown to require exponential time, then they all will require exponential time.

For NP-complete problems, we study approximation algorithms which are polynomial-time algorithms that generate a solution which is provably “close” to optimal.