Algorithmic Combinatorics (Discrete Mathematics II) WS 2018 / 2019
Tibor Szabó

Notes about the first week.

These are some rough informal notes about the material of the first week – to augment your own notes.

Algorithms, efficiency.

Algorithm: (informally) finite set of precise instruction for performing a calculation or solving problem.

Input, Output, (Example: Kruskal's Algorithm. Input: weighted graph (in form of an n-by-n matrix), Output: list of edges (pairs of vertices) forming a spanning tree and having minimum weight)

Definiteness: each step of the algorithm must be defined precisely

Correctness: correct output should be produced for every input

Finiteness: produce output in finitely many steps

How to describe an algorithm? different amount of detail: computer program; pseudocode; words/sentences; precision!!! (at every imaginable scenario the algorithm should uniquely define the next step. In a wordy description we might skip some of the precision (if it is indeed insignificant with respect to correctness and our running time analysis); for example occasionally we might not need to (and hence don't want to) define precisely which "arbitrary" element of a set of eligible elements the algorithm chooses. (To precisify this generically, one could always say that there is some predefined order of the eligible elements and as "arbitrary", the algorithm is supposed to choose the smallest eligible element according to this predefined order. However, in case we indeed choose to ignore what "arbitrary" means, we must live with the "devil" giving us his choice as "arbitrary", which will be the "worst" possible element for us.))

Analysis: How fast is the algorithm? How much space (memory) does it require? time complexity; space complexity;

Running time: number of bit operations/elementary steps/... How fast is an algorithm A? Running it on input I takes $T(A, I)$ steps.

Given an integer n, how long does it take that algorithm A surely finishes on any input of size n? No matter which input of size n it is given! Take the maximum of $T(A, I)$ over all inputs I of size n, denote this with $T_A(n)$. This is the worst-case complexity of A.

Average-case complexity: Given a probability distribution on all inputs of size n (Say the uniform distribution: choose an input uniformly at random among all inputs of size n) take the expected value of $T(A, I)$ and denote it with $\bar{T}_A(n)$. We will not talk about average-case complexity in this course.

What is considered efficient? Roughly: an algorithm whose running time is bounded by a polynomial of the input size. Why is this a "reasonable" definition? Consider the following

table:

Input size	Number of bit operations				
n	$1000000n$	$1000n \log n$	$10n^2$	2^n	$n!$
10	0.01 sec	0.00003 sec	0.000001 sec	0.000001 sec	0.003 sec
100	0.1 sec	0.0007 sec	0.0001 sec	40000000000000 years	$> 10^{100}$ years
1000000	17 min	20 sec	2h 50 min	$> 10^{100}$ years	

Estimates.

- Stirling formula: $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$
  For every $n \in \mathbb{N}$: $\left(\frac{n}{e}\right)^n < n! < n \left(\frac{n}{e}\right)^n$
- Binomial coefficients: $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$
- Almost 1: $e^{-\frac{x}{1-x}} \leq 1 - x \leq e^{-x}$

Our **Asymptotic notation:** Let $f, g : \mathbb{N} \to R$ be two functions.

- $f(n) = O(g(n))$ if there exist constants C and K such that $|f(n)| \leq C|g(n)|$ for every $n \geq K$
- $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $f(n) = o(g(n))$ (or sometimes we write $f(n) \ll g(n)$) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$
- $f(n) \approx g(n)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

Sorting.

Given n distinct numbers $a_1, a_2, \ldots, a_n$, how long does it take to sort them completely. That is find the $\pi \in S_n$, such that $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$. How should we measure running time? We count number of comparisons: "Is $a_i < a_j$?"

If A is a sorting algorithm and $\pi \in S_n$, then let $T(A, \pi)$ be the number of comparisons A does until it outputs the sorted sequence $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$ when starting with the input sequence $a_1, a_2, \ldots, a_n$.

Let $T_A(n) = \max_{\pi \in S_n} T(A, \pi)$ denote the worst case running time of algorithm A.

Algorithm `InsertionSort`

Running time: $T_{IS}(n) = \binom{n}{2}$.

Algorithm `BinaryInsertionSort`

Running time: $T_{BIS}(n) \leq (1 + o(1)n \log_2 n$

Algorithm `MergeSort`$(a_1, a_2, \ldots, a_n)$

Step 0. If $n = 1$, then return list as ordered

Step 1. Divide sequence into (almost) equally long parts $(a_1, \ldots, a_{\lfloor n/2 \rfloor})$ and $(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$

Step 2. Apply `MergeSort` on both of them and create ordered lists $a_{\pi_1(1)} < \cdots < a_{\pi_1(\lfloor n/2 \rfloor)}$ and $a_{\pi_2(\lfloor n/2 \rfloor + 1)} < \cdots < a_{\pi_2(n)}$, where π_1 is a permutation of $\{1, \ldots, \lfloor n/2 \rfloor\}$ and π_2 is a permutation of $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$.

Step 3. Merge the two sorted lists by iteratively comparing the two smallest elements, removing the smaller one from its list and placing it to the top of the final list. When one of the list gets empty, put the other list unchanged on the top of the final list.

Real Time exercise Use MergeSort to sort 3,1,5, 2, 4. How many comparisons did you make?

Let $M(n) = T_{\texttt{MergeSort}}(n)$ the worst case running time of `MergeSort`. For the simpler analysis, first we assume that $n = 2^k$. Then

$$M(n) \leq 2M(n/2) + n - 1.$$

So

$$\begin{aligned}
M(2^k) &\leq 2M(2^{k-1}) + 2^k - 1 \leq 2\left(2M(2^{k-2}) + 2^{k-1} - 1\right) + 2^k - 1 \\
&\leq 2^2 M(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
&\leq \cdots \\
&\leq 2^i M(2^{k-i}) + i2^k - (2^i - 1) \leq 2^i \left(2M(2^{k-i-1}) + 2^{k-i} - 1\right) + i \cdot 2^k - (2^i - 1) \\
&= 2^{i+1} M(2^{k-(i+1)}) + (i+1)2^k - (2^{i+1} - 1) \\
&\leq \cdots \\
&\leq 2^k M(1) + k2^k - (2^k - 1) = n \log_2 n - n + 1,
\end{aligned}$$

since $M(1) = 0$ (we do not need to make single comparison if there is only one number on the list).

How fast is `MergeSort` compared to other sorting algorithms? Is there a faster one? One which maybe sorts just with $O(n \log \log n)$ comparisons? This question involves the complexity of the sorting problem itself. What is the fastest possible algorithm that will sort n numbers? How many comparison queries must it make in the worst case?

These are the type of questions we must immediately ask when we see an algorithm for some problem. Is there a faster one? To show that a particular algorithm is the fastest one to solve some problem is hopeless most of problems. Because this would involve giving a lower bound on the running time of *all* algorithms for the problem. To do that, we would need to consider *all* possible algorithms for the problem (How? How???) and for *each of them* come up with an example of a input instance such that the alogrithm runs long on that input instance.

Complexity of a problem: taking the "best algorithm" what is its worst case complexity? Formally, we look at the minimum of $T_A(n)$ taken over all algorithms A solving the problem. In general it is very difficult to give lower bounds on this.

Sorting is an exception: we will be able to give a lower bound which matches the running time of our algorithms. As you will see, we are able to do this because our algorithms are so fast that a simple information theoretically motivated lower bound proof provides the right lower bound.

Let $T_{\texttt{Sort}}(n) = \min_A T_A(n)$ be the worst case running time of the "fastest" sorting algorithm.

By the analysis of `MergeSort` we have an upper bound: $T_{\texttt{Sort}}(n) \leq n \log_2 n$. This is one of the rare examples of a problem when we can give a tight lower bound. This is mostly due to the fact, that there is a very efficient algorithm (`MergeSort`) which is almost as good as the limit of any algorithm established by some "simple" reason: the information theoretic lower bound. This practically means the following: 1 bit of information (a 0 or a 1) cannot be sent with less than 1 bit. After $T_{\texttt{Sort}}(n)$ questions (each of which has two possible answers YES or NO) we obtain $T_{\texttt{Sort}}(n)$ bits of information. If after asking $T_{\texttt{Sort}}(n)$ questions we gained enough information which of the $n!$ permutations is the correct one. Well, how much information is that? To describe an element of a $n!$-element set we need to use at least $\lceil \log_2(n!) \rceil$ bits. So $T_{\texttt{Sort}}(n)$ should be at least as large as $\log_2(n!)$.

Let's see first whether there is an algorithm which could have gotten the correct order of *any* sequence a_1, a_2, a_3, a_4, a_5 with less than 7 comparisons (what we needed for $3, 1, 5, 2, 4$ with `MergeSort`. Consider the lower bound proof via a game. One player, called Al, asks a comparison question, the other player, Carole (i.e., Oracle) answer with Yes or No. There are $5! = 120$ permutations. Each question of Al eliminates some of them from consideration. After Al's first question, by symmetry, there are exactly 60 eligible permutations remaining. In fact after each ofthe questions of Al, the set of remaining eligible permutations is partitioned into two classes, based on whether the answer to Al's question would be YES or NO, respectively. One of these sets is obviously at least half of the size of the original set. Then Carole can always answer the question with the answer for which the set of remaining eligible permutations is larger, and hence it is at least half of what was eligible before. Hence after just six questions, the set of still eligible permutations is at least $\lceil\lceil\lceil\lceil\lceil\lceil 120/2\rceil /2\rceil /2\rceil /2\rceil /2\rceil /2\rceil = 2$, so Al cannot possible know which one of the two were asked.

To argue formally, let us fix an arbitrary sorting algorithm `Al` with worst case complexity $T_{\texttt{Sort}}(n) =: k$. We prove that if $2^k < n!$, then there exist a permutation$\pi_{\texttt{Al}} \in S_n$, such that `Al` gives an incorrect output when run with input $\pi_{\texttt{Al}}$, leading to a contradiction with the correctness of `Al`. An algorithm, like `Al`, consists of a description of the next comparison query in any imaginable scenario (history) of the algorithm. Each answer to a question of `Al` reduces the set of permutations that are still possible as a solution. We define a partition of $S_n = \cup R_\alpha$ of all permutations into 2^k subsets R_α, indexed by vectors $\alpha \in \{YES, NO\}^k$ with the property that for any permutation $\pi \in R_\alpha$ if we run `Al` with input π, then for every $j = 1, 2 \ldots, k$ we get answer α_j to the jth question of `Al`. The set R_α contains all

permutations that are consistent with all the answers $(\alpha_1, \ldots, \alpha_k)$ for the first k questions of `Al`. The sets R_α are pairwise disjoint for different αs. To make this fully precise we artificially make sure that `Al` does ask k comparisons for every input: even if `Al` would be able to output the right permutation after just $i < k$ queries, it will ask $k - i$ further questions, say repeating its last questions again and again.

Since we assumed $2^k < n!$, after $k = T_{\texttt{Sort}}(n)$ steps there is a partition of S_n into less than $n!$ parts, hence by the Pigeonhole Principle, there is a YES/NO-answer-sequence β of length k, such that part S_β contains more than one permutations. Let $\pi_1, \pi_2 \in S_\beta$, $\pi_1 \neq \pi_2$. The algorithm `Al` when run on input π_1 will receive an answer sequence β and output a permutation π^*. If $\pi_1 \neq \pi^*$, then we get an immediate contradiction with the correctness of `Al`: it produced a wrong answer. Suppose now that $\pi^* = \pi_1$. Run now `Al` on input π_2. Since π_2 is also in S_β, the answer sequence will again be β and hence the output again will be π^*. But now $\pi^* = \pi_1 \neq \pi_2$, so the `Al` was incorrect in this case as well, a contradiction.

So $2^k \geq n!$ and hence by Stirling's formula

$$T_{\texttt{Sort}}(n) = k \geq \log_2(n!) = n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 n + \log_2 \sqrt{2\pi} + o(1) = (1 + o(1)) n \log_2 n.$$

Hence the `MergeSort` algorithm has asymptotically optimal worst case running time.