Discrete Mathematics II WS 2012 / 2013
Tibor Szabó

Some notes about the first week.

These are some rough informal notes about the material of the first week – to augment your own notes. This is only part, the rest (Arthur-Merlin, P/NP, Travelling Salesman approximation) can be found in the book of Lovász-Pelikán-Vesztergombi)

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 or our running time analysis), for example we might not want to define precisely which "arbitrary" element the algorithm chooses. (This could mean for example that there is a predefined order on the elements and the algorithm chooses the smallest eligible element. However, in case we choose to ignore what "arbitrary" means for us, we must live with the "devil" giving us his choice, 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. 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 $AT_A(n)$. We will not talk about this 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	n	$n \log n$	n^2	2^n	$n!$
10	0.00000001 sec	0.00000003 sec	0.0000001 sec	0.000001 sec	0.003 sec
100	0.0000001 sec	0.0000007 sec	0.00001 sec	40000000000000 years	$> 10^{100}$ years
1000000	0.001 sec	0.02 sec	17 min	$> 10^{100}$ years	

Estimates.

- Stirling formula: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- 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 $g(n) = O(f(n))$
- $f(n) = \Theta(g(n))$ (or sometimes we write $f(n) \sim g(n)$) 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 outputting $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$ when starting with the sequence $a_1, a_2, \ldots, a_n$.

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

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

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 repeatedly comparing the two smallest elements, removing the smaller one from its list and placing it to the top of the final list.

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 type of questions we must immediately ask when we see an algorithm. Is there a faster one? This question is very difficult in most of the cases. As the story of Arthur and Merlin indicated we do not expect an algorithm which decides whether a graph is Hamiltonian or not in polynomial time. To prove this we would need to consider all possible algorithms for deciding Hamiltonicity of a graph and for each of them come up with an example of a graph (or rather a graph sequence on an increasing number of vertices) such that the algorithm runs on the graph longer than any polynomial time.

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 all algorithm solving the problem. In general it is very difficult to give lower bounds.

Let $T(n) = \min_A T_A(n)$ the worst case running time of the "fastest" sorting algorithm.

By the analysis of `MergeSort` we have an upper bound: $T(n) \leq n\log_2 n + n - 1$. 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 $T(n)$ questions (each of which has two possible answers YES or NO) we obtain $T(n)$ bits of information. If after asking $T(n)$ 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 $T(n)$ should be at least as large as $\log_2(n!)$.

To argue formally, let us fix an arbitrary sorting algorithm A with worst case complexity $T(n)$. We prove that if $2^{T(n)} < n!$, then there exist a permutation$\pi_A \in S_n$, such that A gives an incorrect output when run with input π_A. Such an algorithm consists of a description of the next comparison query in any imaginable scenario (history) of the algorithm. Each answer to a question of S_v reduces the set of permutations that are still possible as a solution. For each $i = 0, 1, 2, \ldots, T(n)$ we inductively define a partition of S_n of all permutations into 2^i subsets S_v, indexed by vectors $v \in \{YES, NO\}^i$ with the property that for any permutation $\pi \in S_v$ if we run A with input π, then for every $j = 1, 2 \ldots, i$ we get answer v_j to the jth question of A. The set S_v contains all permutations that are still possible after receiving answers $(v_1, \ldots, v_i)$ for the first i questions of A.

For the empty vector $v = \emptyset$ we define the 1-element partition $S_\emptyset = S_n$. Assume that $i \geq 0$ and we have defined the appropriate partition for i. For a given vector $v \in \{YES, NO\}^i$ consider the set S_v of permutations that satisfy the first i questions that were asked. Given these answers, the $(i+1)$th question of A is specified, say it is: "Is $a_p < a_q$?" (Note that this question could depend on the answers to the first i queries.) Then the answer to the $(i+1)$th query of Algorithm partitions S_v according to whether $a_p < a_q$ or $a_q > a_p$. If $a_p < a_q$ for some permutation $\pi \in S_v$, then π is put into set $S_{v_1,\ldots,v_i,YES}$, otherwise it is put into $S_{v_1,\ldots,v_i,NO}$.

Since we assumed $2^{T(n)} < n!$, after $T(n)$ steps there is a partition of S_n into less than $n!$ parts, hence by the Pigeonhole Principle, there is a part S_v which contains more than one distinct permutations. The algorithm A, when receiving the answers $(v_1, \ldots, v_{T(n)})$ in this order, will output a permutation π in at most $T(n)$ steps. Then $\pi \in S_v$, but there exists another permutation $\pi_A \in S_v$, $\pi_A \neq \pi$, such that upon input π_A, the algorithm A would ask the same questions, would receive the same answers and hence would give the same output π, which in this case would be wrong.

So $2^{T(n)} \geq n!$ and hence by Stirling's formula

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

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

Running time of Kruskal's Algorithm. Recall Kruskal's Algorithm from Discrete Math I. Given a connected graph G with edge weights $w : E(G) \to \mathbb{R}$. The algorithm does the following.

- order edges according to weight: $w(e_1) \leq \cdots \leq w(e_m)$
- Maintain a spanning forest H. Start with $V(H) = V(G)$ and $E(H) = \emptyset$. For each vertex v, maintain an index c_v denoting the index of the component v is in. Start with $c(\{v_i\}) = i$

- Iteratively, following the above order, check whether the next edge uw with smallest weight creates a cycle. For this check whether $c_u = c_w$. If YES, do nothing and iterate. If NO, then update $H := H + uw$ and for every $z \in V$ with $c_z = c_w$, update $c_z := c_u$. Then iterate.

What is the running time for a graph with n vertices and m edges? The sorting at the beginning takes $O(m \log m)$ comparisons. In each iteration we check whether they are in the same component of H, that is, whether $c_u = c_w$, which takes just one step, then we either do nothing more or perform the updating of the index of the components of each vertex, which takes at most $O(n)$ steps. There are m iterations, so the cumulative running time is $O(m \log m) + O(nm)$. (With appropriate data structure the second step only needs $m \log m$ steps)