CSc 30400 Introduction to Theory of Computer Science

5th Homework Set - Solutions

Due Date: 5/11

1 Easy questions

E 1. The table is shown below:

$n!$
2^n
$3n^3 + 5n + 4$
$2n^2 + 5n - 4$, $\frac{1}{5}n^2$
$2^{10}n$, n
$\log_2 n$, $\log_{10} n$

E 2. (a) $2^{(n+1)} = \Theta(2^n)$, because $2^{(n+1)} = 2 \cdot 2^n$ which is a constant factor apart from 2^n.

(b) $2^n = o(3^n)$, because $\lim_{n\to\infty} \frac{2^n}{3^n} = 0$.

(c) $\log_5 n^2 = \Theta(\log_2 n)$, because $\log_5 n^2 = 2 \cdot log_5 n = 2 \cdot \log_5 2 \cdot log_2 n$ which is a constant factor apart from $\log_2 n$

(d) $\log_2(n!) = o(n^2)$, because $\log_2(n!) = \log \Pi_{i=1}^n i = \Sigma_{i=1}^n \log i \leq n \log n$ and $\lim_{n\to\infty} \frac{n \log n}{n^2} = 0$.

E 3. (a) This algorithm performs n iterations and inside each iteration a constant number of assignments, so it performs $\mathrm{O}(n)$ assignments.

(b) This algorithm performs $\frac{n}{2}$ iterations and inside each iteration a constant number of assignments, so it performs $\mathrm{O}(n)$ assignments.

(c) This algorithm performs n iterations with a nested loop inside each iteration which is repeated n times in each iteration, so there are $O(n^2)$ repetitions. Each repetition contains a constant number of assignments so it performs $\mathrm{O}(n^2)$ assignments in total.

(d) This algorithm performs n iterations with a nested loop inside each iteration which is repeated i times in each iteration, so there are $1+2+3+\ldots+n = \frac{n(n+1)}{2} = O(n^2)$ repetitions. Each repetition contains a constant number of assignments so it performs $O(n^2)$ assignments in total.

(e) This algorithm performs x iterations. The index i is initially set to n and goes down to 1. In each iteration i is divided by 2 so x is the number of times we should halve n in order to get 1. In other words, $\frac{n}{2^x} = 1$, so $x = \log n$. Each iteration contains a constant number of assignments, so it performs $O(\log n)$ assignments.

E 4.

i *Does A in NP imply A in P?* That would mean that P=NP. It is widely believed that this is unlike.

ii *Does A in NP imply that A is NP-Complete?* No unless P=NP. In order for a problem to be NP-complete it should be in NP and also NP-hard. If P≠NP then it can be shown that there are problems in NP that are not NP-hard.

iii *Is it possible that A is in P and NP-Complete at the same time?* No, unless P=NP. If we find a polynomial time algorithm for an NP-complete problem A, that would mean that any problem in NP can be solved in polynomial time by reducing it to A in polynomial time and then solving A in polynomial time. It is widely believed that P is a proper subset of NP.

iv *Do A in P and $B \leq_P A$ imply B in P?* Yes! If we can transform any instance of B into an instance of A in polynomial time and then solve A in polynomial time, this is a way to solve B in polynomial time.

v *Do A in NP and $A \leq_P B$ imply B in NP?* No. The fact that A is reducible to B can only imply that B is at least as hard as A but it doesn't prove anything about the exact complexity of B. B might be even harder than an NP problem.

vi *Do A in NP $A \leq_P B$ imply that B is NP-Complete?* No. The fact that an NP problem reduces to B doesn't mean that all NP problems reduce to B. In order to show that a problem is NP-complete we should reduce an NP-complete problem to it.

vii *Do A is NP-Complete and $A \leq_P B$ imply that B is NP-Complete?* Showing that an NP-complete problem A is reduced to a problem B proves that B is NP-hard. In order to show that it is NP-complete, B should also be in NP.

viii *Do A in P and B is NP-Complete imply $A \leq_P B$?* Of course! By definition, an NP-complete problem is an NP problem that any problem in NP is reduced to it and since A is in P it is also in NP, thus A should be reducible to B.

ix *Do A is NP-Complete and B is NP-Complete imply $A \leq_P B$?* Yes. See previous answer.

x *Does $A \leq_P B$ imply $B \leq_P A$?* No. If both A $\leq_P$ B and B $\leq_P$ A hold, that means that A and B have the same complexity, which is very unlike given the fact that A, B are two arbitrary problems.

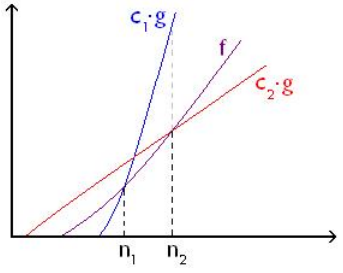
2 Hard questions

H 1. (a) $f = O(g)$ means that $\exists c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0\ f(n) \leq c \cdot g(n)$, which implies that $g(n) \geq \frac{1}{c} \cdot f(n)$. Thus $\exists c' = \frac{1}{c} > 0, \exists n_0 > 0$ such that $\forall n \geq n_0\ g(n) \geq c' \cdot f(n)$, which means that $g = \Omega(f)$.

In simple words, if there is a point after which f is upper-bounded by cg then after this same point g should be lower-bounded by $\frac{1}{c}f$.

(b) $f = O(g)$ means that $\exists c_1 > 0, \exists n_1 > 0$ such that $\forall n \geq n_1\ f(n) \leq c_1 \cdot g(n)$. $f = \Omega(g)$ means that $\exists c_2 > 0, \exists n_2 > 0$ such that $\forall n \geq n_2\ f(n) \geq c_2 \cdot g(n)$. Assume that $n_1 < n_2$ (the other case is similar). Then for $c_1 > 0, n_2 > 0$ and $\forall n \geq n_2\ c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$. Thus $f = \Theta(g)$.

In simple words, if there is a point n_1 after which f is upper-bounded by $c_1 g$ and a point n_2 after which f is lower-bounded by $c_2 g$ then one of n_1, n_2 (say n_2) is a common point after which f is upper-bounded by $c_1 g$ and lower bounded by $c_2 g$ (see figure).



H 2. If the number is expressed in binary then the input length is $l = \log n$. We would need at least $n = 2^l$ steps just to write the unary representation. But this is exponential in the size of the input. Thus there is no way that this transformation can be done in polynomial time.

H 3. (a) In the Longest Path problem we are given a graph of n vertices and a number k and we ask whether there is a path with no repeated vertices of size at least k. The size of the input is the size of the graph and the size of k. The size of the graph is $O(n^2)$ (if the graph is given as an adjacency matrix) and the size of k is $\log k$, since k is a number and it can be given in binary.

A "yes" instance of the Longest Path problem is k or more vertices that form a path in the graph with no repeated vertices. The Longest Path problem is in NP because we can verify in polynomial time in the size of the input a "yes" instance. Indeed, given k or more vertices, one can verify in polynomial time in the size of the input if they form a path and if no vertex in the path is repeated:

- For any two consecutive vertices i, j in the solution, check in the adjacency matrix A if $A[i, j]$ is 1 (in other words if there is an edge connecting i and j). This can be done in time $\mathrm{O}(k)$ and since k is bounded by n it is $\mathrm{O}(n)$, which is polynomial in the size of the input.
- We should also make sure that all the given vertices are different. To do this we need to sort the k vertices (which is done in time $\mathrm{O}(k \log k)$). Now that the vertices are sorted it is easy to check if any vertex is repeated (in linear time $\mathrm{O}(k)$): if there are repeated vertices there are going to be the one next to the other, so we just scan the sorted vertices once. Again, since k is bounded by n, all this procedure takes time $\mathrm{O}(n \log n)$ which is again polynomial in the size of the input.

(b) Given an arbitrary instance of Hamilton Path (a graph G) we transform it in polynomial time to an instance of Longest Path such that the two instances will always have the same answer. The produced instance of Longest Path is the following: The graph remains the same (G) and the number k that is created is n. Observe that a path of n vertices in G is exactly a Hamilton path so if the answer to the Hamilton path is "yes" then the answer to the longest path with parameter n is also "yes" and vice versa.

(c) In the previous two questions we showed that the Longest Path problem is in NP and NP-hard (by reducing the Hamilton Path problem which is an NP-hard problem to it). That shows that the Longest Path problem is NP-complete. There is no polynomial time algorithm for any NP-complete problem, unless P=NP (which is widely believed that it is not the case).