Midterm Examination

CS 4104 (Fall 2009)

Assigned: October 12, 2009.
Due: at the beginning of class on October 21, 2009.

Name: ____________________

PID: ____________________

Instructions

1. For every algorithm you describe, prove its correctness and analyse its running time. I am looking for clear descriptions of algorithms and for the most efficient algorithms and analysis that you can come up with. I am not specifying the desired running time for each algorithm. I will give partial credit to inefficient algorithms, as long as they are correct.
2. You may consult the textbook, your notes, or the course web site to solve the problems in the examination. You **may not** work on the exam with anyone else, ask anyone questions, or consult other textbooks or sites on the Web for answers. **Do not use** concepts from Chapters 6 and later in the textbook.
3. You must prepare your solutions digitally and submit a hard-copy.
4. I prefer that you use LaTeX to prepare your solutions. However, I will not penalise you if you use a different system. To use LaTeX, you may find it convenient to download the LaTeX source file for this document from the link on the course web site. At the end of each problem are three commented lines that look like this:

   ```
   % \solution{
   %
   % }
   ```

   You can uncomment these lines and type in your solution within the curly braces.
5. Do not forget to staple the hard copy you hand in.

Good luck!

Problem 1 (25 points) Solve the following recurrence relation. Note that the recurrence involves two variables n and k.

$$T(n,k) \leq \begin{cases} 2T(n,k-1) + ckn & \text{if } k > 1, \\ cn & \text{if } k = 1. \end{cases} \tag{1}$$

You can assume that c is a positive constant. *Hint:* Unrolling the recurrence may yield a sum that seems hard to analyse. At this point, make a reasonable guess about this sum, and prove the guess by induction (as discussed in class). You may have to play around with your guess till the proof by induction works out correctly.

Solution: Let us unroll the recurrence. There are at most k levels to the recurrence, since k decreases by 1 at each level and $k = 1$ in the base case. At level i, there are 2^i sub-problems parameterised by n and $k - i$, with each sub-problems contributing at most $c(k-i)n$ to $T(n,k$, for a total contribution of $c(k-i)2^i n$ work at the level. Therefore

$$\begin{aligned} T(n,k) &\leq \sum_{i=1}^{k} c(k-i)2^i n \\ &\leq cn \sum_{i=1}^{k} (k-i)2^i \\ &\leq c(2^{k+1} - k - 2)n \\ &= O(2^k n) \end{aligned}$$

This sum is somewhat non-trivial to bound, but is important to note that the tight bound is $O(2^k n)$ and not $O(k2^k n)$, which has an extra factor of k.

We will use induction to prove that $T(n,k) \leq c(2^{k+1} - 2k - 1)n$. The proof is valid for every value of n. In other words, we behave as if n acts like a "constant" in the proof.

Base case: $k = 1$. From the recurrence, we have $T(n,1) \leq cn$. Substituting $k = 1$ into our guess, we have $T(n,1) \leq c(2^2 - 1 - 2)n = cn$. Since $cn \leq cn$ (with the left hand side coming from the recurrence and the right hand side coming from the guess), the base case is true.

Inductive hypothesis: Let us assume that $T(n,k-1) \leq c(2^k - (k-1) - 2)n = c(2^k - k - 1)n$.

Inductive step: We will prove that $T(n,k) \leq c(2^{k+1} - k - 2)n$, using the following series of inequalities:

$$\begin{aligned} T(n,k) &\leq 2T(n,k-1) + ckn, && \text{from the recurrence relation} \\ &\leq 2\left(c(2^k - k - 1)n\right) + ckn, && \text{from the inductive hypothesis} \\ &= c(2^{k+1} - 2k - 2 + k)n \\ &= c(2^{k+1} - k - 2)n, && \text{the bound we needed to prove.} \end{aligned}$$

Remarks: I deduced 5 points if you proved a bound of $O(k2^k n)$. I heavily penalised students who forgot the proof by induction. There were some basic flaws in some proofs by induction:

1. The right inductive hypothesis is that $T(n,k-1) \leq c(2^k - 2(k-1) - 1)n$. It is incorrect to change the right-hand side of the inductive hypothesis to include terms you get from the recurrence or to add extra terms that do not appear in the bound you want to prove.
2. It is correct to act as if n behaves like a constant throughout the proof. If you base your inductive hypothesis on $T(n-1,k-1)$, then your inductive step must prove the bound for $T(n,k-1)$ and on $T(n,k)$.

3. Some students used equalities throughout their proofs. Doing so is incorrect because the recurrence is given to us as an inequality.

Problem 2 (15 points) Consider the problem of minimising lateness that we discussed in class. We are given n jobs. For each job $i, 1 \leq i \leq n$, we are given a time $t(i)$ and a deadline $d(i)$. Let us assume that all the deadlines are distinct. We want to schedule all jobs on one resource. Our goal is to assign a starting time $s(i)$ to each job such that each job is delayed as little as possible. A job i is *delayed* if $f(i) > d(i)$; the *lateness of the job* is $\max(0, f(i) - d(i))$.

Define

1. the *lateness of a schedule* as $\max_i \big(\max\big(0, f(i) - d(i)\big)\big)$ and
2. the *delay of a schedule* as $\sum_{i=1}^{n} \big(\max\big(0, f(i) - d(i)\big)\big)$.

Note that although the words "lateness" and "delay" are synonyms, for the purpose of this problem we are defining them to mean different quantities: the lateness of a schedule is the *maximum* of the latenesses of the individual jobs, while the delay of a schedule is the *sum* of the latenesses of the individual jobs.

Consider the algorithm that we discussed in class for computing a schedule with the smallest lateness: we sorted all the jobs in increasing order of deadline and scheduled them in this order. In this problem, we will show in two different ways that this algorithm does not compute the schedule with the smallest delay:

(i) (5 points) Provide a counter-example to show that this algorithm will not compute the schedule with the smallest delay. You will not need more than two jobs in your counter-example.

(ii) (10 points) We proved that the earliest-deadline-first algorithm correctly solves the problem of minimising lateness. If we were to use the *same proof* to try to demonstrate that the algorithm correctly solves the problem of minimising delay, where does the proof break down?

Solution:

(i) Consider two jobs 1 and 2 with $t(1) = 1$, $d(1) = 4$, $t(2) = 10$, and $d(2) = 3$. The earliest-deadline-first algorithm with schedule job 2 before job 1, incurring a delay of $(10 - 3) + (11 - 4) = 14$. However, scheduling job 1 before 2 incurs a smaller delay of $0 + 11 - 3 = 8$.

(ii) The proof breaks down when we try to show that removing an inversion from a schedule cannot increase the delay. If the two (consecutive) jobs that have the inversion are i and j, we defined the notation $l(i)$ as the lateness of job i in the schedule with the inversion and $l'(i)$ as the lateness in the schedule without the inversion. While it is true that $l'(i)$ and $l'(j)$ are both less than the maximum of $l(i)$ and $l(j)$, we cannot guarantee that $l'(i) + l'(j) \leq l(i) + l(j)$. Since the delay is defined in terms of the sum of the individual latenesses, it is possible that removing an inversion actually increases the total delay.

Remarks: Most students did well on this problem. Some students did not explicitly specify the proof that their example was a counter-example.

Problem 3 (30 points) We say that one two-dimensional point $p = (p_x, p_y)$ *looks down on* another two-dimensional point $q = (q_x, q_y)$ if $p_x \geq q_x$ and $p_y \geq q_y$. For example, the point $(2, 10)$ looks down upon the point $(-5, 8)$ but not upon the point $(-5, 12)$. In a set P of n two-dimensional points, a point p is said to be *majestic* if no point in P looks down upon p. Devise an efficient algorithm to compute all majestic points in P. Note that the number of majestic points can vary anywhere from 1 (e.g., if all points are on the line $x = y$, then the point with the largest y-coordinate is the only majestic point) to $|P|$ (e.g., if all points are on the line $x + y = 1$, then all points are majestic). Your algorithm must automatically determine the correct subset of P that make up the majestic points. Therefore, your proof of correctness must show that

1. all points returned by your algorithm are majestic and

2. all majestic points in P are computed by your algorithm.

Solution:

Algorithm 1: This approach uses a sort and a linear scan of the sorted points. Sort the points in *decreasing* order of x-coordinate and let $p_i, 1 \leq i \leq n$ be the ith point in this order. Note that p_1 is majestic since no other point has a larger x-coordinate. For $1 \leq i \leq n$, define $y_i = \max_{j=1}^{i} p_{i_y}$[1], i.e., y_i is the largest y-coordinate of the first i points. If $p_{(i+1)_y} < y_i$,[2] then p_{i+1} cannot be majestic, since there is some point with a larger x-coordinate that also has a larger y-coordinate. On the other hand, if $p_{(i+1)_y} > y_i$, then p_{i+1} must be majestic: no point with a larger x-coordinate than p_{i+1} has a larger y-coordinate than p_{i+1}. These observations imply a simple linear scan over the points: output p_1 as majestic, set $y_1 = p_{1_y}$ and for every other $i, 1 < i \leq n$, if $p_{i_y} > y_{i-1}$, output p_i as majestic and update $y_i = p_{i_y}$. This algorithm takes $O(n \log n)$ time ($O(n \log n)$ time for the initial sort and $O(n)$ time for the scan).

Algorithm 2: This approach uses divide and conquer. Let $M(P)$ denote the set of points computed by the algorithm presented below for the set P.[3] If P contains just one point, simply return P. Otherwise, sort the points in increasing order of x-coordinate. Divide the points into two sets L and R, with L containing the $\lfloor n/2 \rfloor$ points with the smallest x-coordinates and R containing the remaining points. Recursively compute $M(L)$ and $M(R)$, *while ensuring that the points in $M(L)$ and $M(R)$ are sorted in decreasing order of y-coordinate.*[4] In the conquer step, we will merge $M(L)$ and $M(R)$ to yield $M(P)$. Let p be the first point in $M(R)$. Using a linear scan, identify the set $M'(L)$ of points in $M(L)$ whose y-coordinates are larger than p_y. Concatenate $M'(L)$ and $M(R)$ and return these points as $M(P)$.

The running time of this algorithm is $O(n \log n)$ since we solve a problem of size n by recursively solving two problems of size at most $\lceil n/2 \rceil$ and we spend $O(n)$ time in the conquer step.

The algorithm's correctness follows from the following three statements, which we will prove by induction on the size of P:

(i) All points in $M(P)$ are majestic.

(ii) Every majestic point is in $M(P)$.

(iii) $M(P)$ is sorted in decreasing order of y-coordinate.

The base case is when $|P| = 1$: all these statements are true because the only point in P is majestic and $M(P)$ contains this point.

For the inductive hypothesis, assume that all three statements hold true for the set $M(Q)$ computed by the algorithm for all sets of points Q, where $|Q| < |P|$.

In the inductive step, we need to prove that given $M(L)$ and $M(R)$ (for each of whom the three properties hold by the inductive hypothesis), the conquer step computes $M(P)$ correctly. Let us prove each of the properties.

(i) *All points in $M(P)$ are majestic.* The algorithm constructs $M(P)$ from a subset of points in $M(L)$ and all points in $M(R)$.

Let us consider $M(R)$ first. Each point in $M(R)$ has a larger x-coordinate than *every* point in L. Therefore, no point in L can look down on any point in $M(R)$. Furthermore, no point in $M(R)$ can look down on another point in $M(R)$, since $M(R)$ is the set of majestic points for

[1]Double subscripts are evil; don't use them. Try to rewrite this proof without using double subscripts.

[2]Double subscripts with operations in them are even more evil.

[3]Note that we are not defining $M(P)$ to be the set of majestic points in P, since we have not yet specified the algorithm, let alone proven that the algorithm computes the set of majestic points correctly.

[4]If this condition is satisfied, you can prove that the points in $M(L)$ and in $M(R)$ are sorted in increasing order of x-coordinate.

R. Therefore, the algorithm does not include any non-majestic points when it includes $M(R)$ in $M(P)$.

Now let us consider the subset $M'(L)$ of points from $M(L)$ that the algorithm includes in $M(P)$. All these points have y-coordinate larger than that of p, the first point in $M(R)$. Since p is the first point in $M(R)$ and since $M(R)$ is sorted in decreasing order of y-coordinate (by the inductive hypothesis), p is the point with largest y-coordinate in R. Therefore, every point in $M'(L)$ has a larger y-coordinate than every point in R, proving that $M'(L)$ contains only majestic points.

(ii) *Every majestic point is in $M(P)$.* Let q be a majestic point for the set of points P; q is an element either of L or of R. Suppose $q \in R$. Then q must be majestic in R as well. Therefore, by the inductive hypothesis, $q \in M(R)$ and thus $q \in M(P)$ by construction, since the algorithm includes all points in $M(R)$ in $M(P)$. Suppose $q \in L$; q must be majestic in L as well. Therefore, by the inductive hypothesis, $q \in M(L)$. The point p identified by the algorithm has a larger x-coordinate than q. Since q is majestic in P, q must have a larger y-coordinate than p. Therefore, the algorithm will include q in $M'(L)$ and therefore in $M(P)$.

(iii) *$M(P)$ is sorted in decreasing order of y-coordinate.* Note that we must prove this property since we used this property to prove the first property. The algorithm constructs $M(P)$ by concatenating $M'(L)$ with $M(R)$. Since $M'(L)$ is a sub-sequence of $M(L)$ ($M'(L)$ consists of all points in $M(L)$ with y-coordinate larger than p_y) and $M(L)$ is sorted in decreasing order of y-coordinate (by the inductive hypothesis), so is $M'(L)$. Furthermore, $M(R)$ is also sorted in decreasing order of y-coordinate (by the inductive hypothesis). Every point in $M'(L)$ has larger y-coordinate than every point in $M(R)$ by construction. Therefore, the concatenation of $M'(L)$ and $M(R)$ is sorted in decreasing order of y-coordinate.

Remarks: The primary mistake was to provide an algorithm with $O(n^2)$ running time by essentially comparing each point to every other point. Since this solution is very easy to obtain, I deducted 20 points. Other students had very short or sketchy proofs, which boiled down to "My algorithm is correct because every point it computes is majestic." I deducted anywhere between 5 and 10 points for such proofs.

Problem 4 (30 points) The curriculum in the Department of Computer Science at the University of Draconia consists of n courses. Each course is mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. In such a case, a student cannot take course w in the same or an earlier semester than she takes course v. A student can take any number of courses in a single semester. Design an algorithm that computes the minimum number of semesters necessary to complete the curriculum. You may assume that G does not contain cycles, i.e., it is a directed acyclic graph. Chapter 3.6 of the textbook discusses directed acyclic graphs. The graph can be split into multiple components. For example, an extreme case is when there are no pre-requisites at all, in which case each course is in a separate component. Of course, one semester suffices in this trivial case.

If you are concerned about how G is represented, assume that for each course, you have a list of courses for which it is a pre-requisite (the adjacency list representation). If you need, you can assume that the "list" for each course is stored in an array. if you want to use another representation, please describe it. *Do not use an adjacency matrix representation.*

Solution: There are numerous solutions to this problem. We will present a greedy strategy.

Consider any path P in G. The courses in P must be scheduled in different semesters, since each course (but the last) in P is a pre-requisite for the next course in P. Therefore, we require a number of semesters at least equal to the length of P. Applying the argument to the longest path Q in G, the smallest number of semesters **must be at least as large as** the length of Q. Let k be the length of Q. If we could develop an algorithm that schedules all courses in k semesters, it would be optimal.

A greedy strategy suggests itself. The algorithm operates in rounds. In round $i, i > 1$, the algorithm finds all courses that have no pre-requisites, schedules them in semester i, and deletes them from G. The algorithm terminates when G is empty.

Let us prove the optimality of the algorithm. Note that there must be at least one course without a pre-requisite in a DAG, as proven in (3.19) on page 102 of your textbook. Thus the algorithm removes at least one course in each round. The remaining graph is a DAG, since deleting a node from a DAG cannot introduce cycles. Therefore, the algorithm terminates in at most n rounds.

We will now prove that the algorithm actually terminates in k rounds, thereby establishing its optimality. Note that it is not enough to prove that the courses in the longest path Q are scheduled in k semesters. We have to prove, in addition, that for every path in G, the courses in that path are scheduled within k semesters. To do so, we define the depth d_v of a course $v \in G$ to be the number of courses in the longest path in G that terminates at v; we include v in this count. Clearly, the largest depth of a course is k. It suffices to prove that the greedy algorithm schedules every course v in semester d_v.

We can prove this fact by induction. The base case is $d_v = 1$. These are precisely the courses in G that have no pre-requisites. Indeed, the algorithm schedules these courses in semester 1. Now, for the inductive hypothesis, assume that the algorithm schedules all courses w with $d_w \leq l$ in semester d_w. We will now prove the statement for semester $l + 1$, i.e., for courses v with $d_v = l + 1$. Consider the graph G_l that remains at the end of round l. Let v be a course with depth $l+1$ in G. What is the depth of v in G_l? If this depth is 1, we are done with the proof, since the algorithm will schedule v in this round (which is $l+1$). Suppose the depth of v in G_l is larger than 1. Then v must have a pre-requisite u in G_l. What is d_u, the depth of u in G? By the inductive hypothesis, the algorithm has already scheduled all courses w with $d_w \leq l$ in semester d_w. Therefore, d_u must be larger than l, implying that the depth of v is $d_v > d_u + 1 > l + 1$, which contradicts the fact that $d_v = l + 1$. Note that we used the fact that the depth of a course must be at least 1 more than the depth of any pre-requisite for the course. Consequently, v has no pre-requisite in G_l, meaning that v will be scheduled in semester $d_v = l + 1$, as desired. This completes the proof.

As for the running time, we can modify the algorithm for topologically sorting a DAG on pages 103 and 104 of your textbook to achieve a running time of $O(m + n)$, where m is the number of pre-requisite pairs in G. Briefly, for every course v, we maintain a counter p_v set to the number of pre-requisites of v in G. In the first round, we find all courses with $p_v = 0$ using a linear scan through G and schedule them. In round i, when we schedule a course u, we

(i) decrement by 1 the p_v values for all courses v for which u is a pre-requisite and

(ii) add such a node v to the list of nodes to be scheduled in the next round if p_v reaches 0.

Since G is provided to us in adjacency list format, we can perform this operation in time proportional to the number of courses for which u is a pre-requisite. We process any edge (u, v) only when we schedule u. Therefore the total work done by the algorithm is $O(m + n)$.

Remarks: Some students proved the optimality of the algorithm by using the argument that the greedy algorithm always stays ahead of any other algorithm. Suppose the greedy algorithm uses l semesters to schedule all courses and the optimal algorithm uses $m < l$ semesters. Let $G_i, 1 \leq i \leq l$ be the set of the courses the greedy algorithm schedules in semester i and let $O_i, 1 \leq i \leq m$ be the set of courses that the optimal algorithm schedules in semester i. Then for all $1 \leq j \leq m$,

$$\bigcup_{i=1}^{j} G_i \supseteq \bigcup_{i=1}^{j} O_i.$$

These students proved the statement by induction, and then showed that $m = l$. This proof is valid and elegant, but it gives you less insight into the problem since it does not relate l to any structural quantity in G, namely the length of the longest path in G.

It is not enough just to say that since "the greedy algorithm schedules each courses as soon as it can be scheduled, it must output the smallest number of semesters." How do we know that an algorithm that does adopts a different strategy will not use fewer semesters? I deduced points for solutions without proof of correctness.

Another technical inaccuracy in many proofs was claiming that the smallest number of semesters in which the courses can be scheduled **equals** the length k of the longest path Q in G. This statement is not immediately obvious. The only thing that is clearly true is that the smallest number of semesters **must be at least as large as** k. It is always possible that more semesters may be needed. The algorithm is a constructive proof that all courses can indeed be scheduled in k semesters. I deduced some points for students who made this mistake.

Many students based their proof on notions such as "depth" of a node without defining the term. Since I could not guess what you mean, I deducted points, depending on how difficult it was to understand your solution.

Other students decided to use depth-first search, often by incrementing and decrementing counters keeping track of the length of the longest path. However, can you prove that you can indeed use DFS to compute the longest path in a DAG? I deducted 10 points for DFS-based answers.