CIS 575: Introduction to Algorithm Analysis
Exam 3 (Final) with suggested answers

May 15, 2013, 4:10-6:00pm

General Notes

- You can have two sheets (each side may be used) of notes produced by *you* (hand-written or printed), but no other material and no use of laptops or other computing devices.
- If you believe there is an error or ambiguity in any question, mention that in your answer, and *state your assumptions*.
- Please write your name on this page.

Good Luck!

NAME:

1. *Asymptotic Notation, 20p.* Assume that $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, that $f(n) = f_1(n) + f_2(n)$ for all $n \geq 0$, and that $g(n) \geq g_1(n)$ and $g(n) \geq g_2(n)$ for all $n \geq 0$. Prove, using the definition of big-O, that $f \in O(g)$.

Answer: By assumption there exists $n_1 \geq 0$ and $c_1 > 0$ such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq n_1$, and there exists $n_2 \geq 0$ and $c_2 > 0$ such that $f_2(n) \leq c_2 g_2(n)$ for $n \geq n_2$. Now define $c = c_1 + c_2 > 0$, and $n_0 = \max(n_1, n_2)$. For $n \geq n_0$ we now have

$$f(n) = f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_1 g(n) + c_2 g(n) = c\ g(n)$$

and thus $f(n) \leq c\ g(n)$ for $n \geq n_0$ which amounts to $f \in O(g)$.

2. *Dynamic Programming, 20p.* Given an array $A[1..n]$ of positive integers, we want to find $X \subseteq \{1..n\}$ such that

- two "neighbors" cannot both be in X: for all $i \in 1..n-1$, if $i \in X$ then $i+1 \notin X$.
- the sum $V(X) = \sum_{i \in X} A[i]$ is as big as possible.

For example, for the table

i	1	2	3	4	5	6
$A[i]$	5	3	2	8	9	4

we should pick $X = \{1, 4, 6\}$ which gives $V(X) = 17$.

We shall solve this problem using *dynamic programming*, introducing an array $B[0..n]$ such that $B[i]$ is the highest possible value of $V(X)$ when $X \subseteq \{0..i\}$.

In time $\Theta(n)$, it is possible to tabulate B and then print the elements of X, as done by the following code where you must *fill in the details*:

$B[0] \leftarrow \underline{0}$

$B[1] \leftarrow \underline{A[1]}$

for $i \leftarrow 2$ to n do

    $B[i] \leftarrow \max(\underline{A[i] + B[i-2]}, \underline{B[i-1]})$

$j \leftarrow n$

while $\underline{j \geq 1}$

    if $\underline{B[j] > B[j-1]}$

        print j

        $j \leftarrow \underline{j-2}$

    else

        $j \leftarrow \underline{j-1}$

Also show the values of B produced by your algorithm on the above example.

Answer: we construct the below table (and then print 6, 4, 1).

i	0	1	2	3	4	5	6
$A[i]$		5	3	2	8	9	4
$B[i]$	0	5	5	7	13	16	17

3. *Complexity Analysis, 20p.* Consider the following algorithm, where we assume that the global variable G is a directed graph with nodes $1..n$ that is represented as an adjacency matrix.

```
RANK(P[1..n])
  let C[1..n] be a new array
  if n = 1
    C[1] ← P[1]
  else
    A ← RANK(P[1 .. ⌊n/2⌋])
    B ← RANK(P[(⌊n/2⌋ + 1) .. n])
    C ← MERGE(A, B)
  return C

MERGE(A[1..k], B[1..m])
  let C[1..k + m] be a new array
  i, j ← k, m
  while i + j > 0
    if i = 0 or (i, j ≥ 1 and G has an edge from A[i] to B[j])
      C[i + j] ← B[j]
      j ← j − 1
    else
      C[i + j] ← A[i]
      i ← i − 1
  return C
```

Analyze the running time of RANK when called with an array $P[1..n]$. Argue for your answer which should be of the form $\Theta(f(n))$. *Hint:* You do not need to understand what the algorithm does, but you will need to first analyze the running time of MERGE.

Answer: By our assumption about G having a matrix representation, the body of the **while** loop in MERGE executes in constant time. Hence the running time of MERGE($A[1..k]$,$B[1..m]$) is in $\Theta(k+m)$. In the body of RANK, the call to MERGE is with arguments of size approximately $n/2$ and the cost of the call is thus in $\Theta(n)$. Therefore the running time $T(n)$ of RANK can be described by the recurrence

$$T(n) \in 2 \cdot T(\frac{n}{2}) + \Theta(n)$$

The Master Theorem, with $a = b = 2$ and $r = \log_b(a) = 1 = q$, now tells us that $T(n) \in \Theta(n \log(n))$.

4. *Greedy Algorithms, 20p.* We consider the scheduling of n jobs, each taking one unit of server time. Each job $j \in 1..n$ has a profit $p_j > 0$, and each time slot $t \in 1..n$ has a worth $w_t > 0$. We may assume that all p_j are different, and all w_t are different.

A schedule assigns a time slot $t(j)$ to each job j. The value of a schedule is given by

$$\sum_{j=1}^{n} p_j w_{t(j)}$$

Now consider the greedy strategy that constructs a schedule as follows: first assign the job with highest profit to the time slot with the highest worth; next assign the job with the second highest profit to the time slot with the second highest worth; etc.

> Example: for $n = 3$, let p_j and w_t be given by
>
> | j | 1 | 2 | 3 |
> |---|---|---|---|
> | p_j | 6 | 5 | 7 |
>
> | t | 1 | 2 | 3 |
> |---|---|---|---|
> | w_t | 2 | 4 | 3 |
>
> Then this strategy will assign time slot 2 to job 3 (for value $7 \cdot 4 = 28$), time slot 3 to job 1 (for value $6 \cdot 3 = 18$), and time slot 1 to job 2 (for value $5 \cdot 2 = 10$), giving a total value of 56.

Prove that this greedy strategy produces an optimal schedule, that is, a schedule with maximal value.

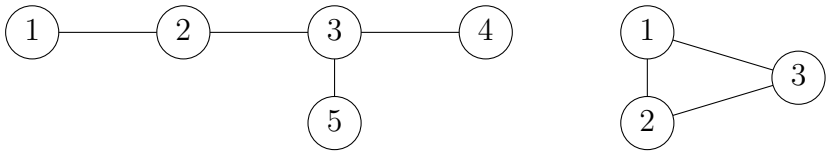
Hint: Consider a schedule S that cannot have been generated by the greedy strategy. That is, there exists j and k such that $p_j > p_k$ but yet t_k is worth more than t_j. Now argue that it is possible to improve on S.

Answer: Assume we have a schedule that cannot have been generated by our strategy. Then there exists j and k such that $p_j > p_k$ but yet, with $t(j) = u$ and $t(k) = v$, we have $w_u < w_v$. But now let us swap j and k so that j gets v and k gets u. This will improve our schedule, as can be seen from a general result that says that if $a > b > 0$ and $x > y > 0$ then $ax + by > ay + bx$. (For $(ax + by) - (ay + bx) = (a - b)(x - y) > 0$.)

Thus any schedule not following our strategy can be improved. Since an optimal schedule does exist, it must be the one produced by our strategy!

There is one more question on the back page!

5. *Depth-First Search, 20p.* Given an undirected connected graph $G = (V, E)$, a *2-coloring* is a mapping c from V to $\{1, 2\}$ such that if $(u, w) \in E$ then $c(u) \neq c(w)$. For example, consider the two graphs below:



The leftmost graph can be given a 2-coloring, with $c(1) = c(3) = 1$, and $c(2) = c(4) = c(5) = 2$. On the other hand, the rightmost graph cannot be 2-colored since we must demand $c(1) \neq c(2)$, $c(1) \neq c(3)$, $c(2) \neq c(3)$ which is impossible as c maps into $\{1, 2\}$.

Find an algorithm to decide if a given graph can be 2-colored. Your algorithm should be constructed by augmenting the generic algorithm for depth-first search given below. That is, you should instantiate the 5 abstract actions with concrete code, such that a call DFS(u) with u an arbitrary node gives a 2-coloring $c[1..n]$ if one such exists, and reports failure otherwise. You do not need to argue for correctness, or estimate running time.

Hint: to the root, assign an arbitrary color which will then force the coloring of all other nodes (or a conflict will arise). It will be an invariant that for all u except the root, $c[u]$ is defined immediately *before* DFS(u) is called.

```
DFS(u)
  d[u] ← current_time
  PreNode action: if u is the root then assign c[u] ← 1 (arbitrary)
  foreach (u, w) ∈ E
    if d[w] is undefined
      PreEdge action: c[w] ← 3 − c[u] (w gets the opposite color of u)
      DFS(w)
      PostEdge action: NONE
    else
      OtherEdge action: if c[u] = c[w] then report fail
  PostNode action: NONE
```