

CS 473 (ug): Algorithms, Fall 2005

Mid-Term 1, Oct 4

Name: SOLUTIONS
Net ID:
Alias:

-
- Neatly print your full name, your NetID, and an alias of your choice in the boxes above. Exam grades may be listed on the course web site by alias. Please write the same alias on subsequent exams! For privacy reasons, your alias should not resemble your name or NetID. By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed. ***Do not*** give us your *Social Security number!*
 - Write your name in the upper right corner of every page. Do it now.
 - This is a closed book exam. No notes, books, calculators, etc. are allowed. The exam is designed to take 75 minutes. However, you may take up to two hours.
 - Read all the questions carefully. If something seems unclear, ask one of the course staff.
 - Do all the work in the space provided or on the back sheets if necessary. The space provided after the question should be sufficient for both solutions and rough work, but if some of your answers cannot fit into this space, you may use the back sheets. Please tell us where to look, though. For bonus points equivalent to an extra credit hw problem, write the word “rutabaga” just above the table appearing in problem 2.
 - You may state and use without proof any results proved in class or in the homeworks.
 - A table of logarithm values is provided to help with question 2. You may use these for any other problem, if necessary.
 - Writing “**I don’t know**” is worth 25 percent of any problem or part of a problem, **except for problem 1.**
-

#	1	2	3	4	5	Total
Score						
Grader						

Leave this page blank.

1. True/False questions

Directions: This section consists of true/false questions. If the statement is necessarily true, **write out the entire word “true”**. If the statement is not necessarily true, **write out the entire word “false”**. Your score will be twice the number correct, minus the number incorrect. Writing “I don’t know” **will not** score any points for this particular problem.

- (a) A motivation for using $O()$ analysis and ignoring constants is that if A is an algorithm, then the best case behavior for A as a function of input size, and the worst case as a function of input size, do not differ asymptotically by more than some constant.

FALSE. There can be a drastic difference between best case and worst case running times of an algorithm. For example, bubble sort requires worst case $O(n^2)$, but is done in $O(n)$ in the best case that the data was already sorted (assuming there is a simple check for swaps done). There are a variety of reasons we don’t obsess about constants, but this is not one of them. The main reason is that we’d like a theory that is robust across different computing platforms (which have different speeds for different types of instructions, and/or different clock speeds to begin with) and different programming languages (where a basic “step” means different things).

- (b) Let G be a weighted graph, and let e be an edge with maximum weight. Then e is in an MST T for G if and only if $\{e\}$ is a cut set - that is, the removal of e disconnects the graph.

FALSE. Don’t feel bad if you missed this one; most people did. All that is required is that e be a minimum weight edge in some cut. This can happen without e comprising the entire cut, if there all other edges in the cut are max weight. A simple example: Consider a triangle of vertices a, b, c , with edge weights $ab = 10$, $ac = 10$, and $bc = 1$. Then either ab or ac must be chosen, since this is the only way to reach a , each is a max weight edge, and neither alone forms a cut set.

- (c) Let G be a weighted graph with all edge weights distinct. Then there are two distinct MSTs T_1 and T_2 for G such that T_1 can be obtained from T_2 by removing two edges from $T_2 - T_1$ and replacing them with two edges in $T_1 - T_2$.

FALSE. If G has distinct edge weights, then there is a unique MST for G . Thus, you can stop reading right after the words “Then there are two distinct MSTs...”

- (d) Recall that when the BFS algorithm is run from some vertex s of a graph, “layered” sets $S_0 = \{s\}$, S_1 , etc., are created, where S_i contains all vertices that are distance i from s . The vertices of G can be colored with two colors so that no two adjacent vertices are the same color if and only if no edge joins any pair of vertices in some set S_i .

TRUE. We showed this in class, and it is in the text. If the property holds, a valid two-coloring is simply to color all vertices in even-indexed sets S_{2i} with orange, and all vertices in odd-indexed sets S_{2i+1} with blue. Conversely, if the graph is 2-colorable, then there can be no odd cycles, and this precludes edges between two elements of any S_i since from such an edge one can find a simple odd cycle by tracing the endpoints back towards the start node s until these paths first meet.

- (e) Consider a greedy algorithm for making change, given available coin values $C_k > C_{k-1}, \dots, > C_1 = 1$.

```
MAKE-CHANGE( $n$ )
if  $n > 0$ 
  let  $i = \max\{j : n \geq C_j\}$ 
  output a coin of value  $C_i$ 
  MAKE-CHANGE( $n - C_i$ )
```

Then algorithm MAKE-CHANGE always outputs the fewest number of coins.

FALSE. While the greedy coin-changing algorithm works for US coinage 25, 10, 5, 1, it does not work for all coin systems. For example, suppose we replace the nickel with a 15cent piece. Then 20 cents made greedily would require six coins (the 15cent piece and five pennies), but two dimes is optimal.

2. Solving Recurrences

If needed, you may refer to the following table, where the entry in row i , column j shows the value of $\log_i j$.

	2	3	4	5
2	1	1.5850	2	2.3219
3	0.6309	1	1.2619	1.4650
4	0.5	0.7925	1	1.1610
5	0.4307	0.6826	0.8614	1

Give tight *asymptotic* (“big-Oh”) bounds for the following (2 points each). For additional points, (2 for part (a), 1 each for parts (b) and (c)), solve the recurrences exactly.

(a) $T(n) = 2T(n/3) + 5n$, $T(1) = 1$

Quick $O()$ answer: $O(n)$, since the ratio between work done at the first level ($5n$) is $3/2$ the work done at the second level ($10n/3$) in the recursion tree, and this holds throughout, the sum of work for the geometric series is dominated by the first term.

Exact answer:

At first glance, we have a geometric series with first term $5n$, each successive term $2/3$ the previous, and the number of terms is $\log_3 n$. However, a closer look reveals that our attempt to save you grief by omitting floors or ceilings from the expression creates an issue: The boundary condition “ $T(1) = 1$ ” is only reached provided the input is a power of 3; otherwise we have an infinite regress and an undefined sum, since there is no base case. In this case, we just take the infinite sum, since we have convergence, so stopping condition doesn’t matter. If you summed only the first ceiling($\log_3 n$) terms, that is ok also. You will have to add the total work at the bottom, which is $2^{\log_3 n} \times 1 = 3^{\log_3 2^{\log_3 n}} = 3^{\log_3 n^{\log_3 2}} = n^{0.6309}$. This will work out to $15n - 15(n^{0.6309}) + n^{0.6309} = 15n - 14n^{0.6309}$.

(b) $T(n) = 3T(n/9) + \log_9 n$, $T(1) = 1$

Quick $O()$ answer: $O(\sqrt{n})$. At the root of recursion tree we have $\log_9 n$. The next level we have $3(\log_9 n/9) = 3(\log_9 n - 1)$. Noting that the second expression is more than twice the first, and only three less than three times the first, this tells us that the majority of work will happen at the bottom of the tree; the work is basically tripling at each level. To get a $O()$ estimate it will be sufficient to look only at the bottom level. The depth (assuming we ignore the issue about never actually hitting $T(1)$ without floors or ceilings to help) will be $\log_9 n$, since we divide by 9 at each level. Thus, at the bottom level, the number of steps is:

$$\begin{aligned} 3^{(\log_9 n)} \times T(1) &= 9^{1/2^{\log_9 n}} \times 1 \\ &= 9^{\log_9 n^{1/2}} \\ &= n^{1/2} \end{aligned}$$

Exact answer:

At level i (counting the top as level 0), we do $3^i \times \log_9 \frac{n}{9^i}$ units of work, except at the last level, where we do \sqrt{n} work.

Therefore, the total amount of work is:

$$\begin{aligned}
T(n) &= \sqrt{n} + \sum_{i=0}^{\log_9 n - 1} 3^i \times \log_9 \frac{n}{9^i} \\
&= \sqrt{n} + \sum_{i=0}^{\log_9 n - 1} 3^i \times (\log_9 n - \log_9 9^i) \\
&= \sqrt{n} + \sum_{i=0}^{\log_9 n - 1} 3^i \times (\log_9 n - i \log_9 9) \\
&= \sqrt{n} + \sum_{i=0}^{\log_9 n - 1} 3^i \times (\log_9 n - i) \\
&= \sqrt{n} + \log_9 n \sum_{i=0}^{\log_9 n - 1} 3^i - \sum_{i=0}^{\log_9 n - 1} i \times 3^i \\
&= \sqrt{n} + \log_9 n \frac{\sqrt{n} - 1}{2} - \left(\frac{(\log_9 n - 1)\sqrt{n}}{2} - \frac{3}{4} \left(\frac{\sqrt{n}}{3} - 1 \right) \right) \\
&= \sqrt{n} - \frac{\log_9 n}{2} + \frac{\sqrt{n}}{2} + \frac{\sqrt{n}}{4} - \frac{3}{4} \\
&= \frac{7}{4}\sqrt{n} - \frac{\log_9 n}{2} - \frac{3}{4}
\end{aligned}$$

(c) $T(n) = 5T(n/3) + n^2, \quad T(1) = 1$

Quick $O()$ answer: Comparing n^2 (first level) with $5(n/3)^2 = (5/9)n^2$, we see again a shrinking geometric series, so the first term dominates, and the answer is $O(n^2)$.

Exact answer: Again ignoring pesky stopping condition due to no ceilings or floors, we can sum the infinite series to get $n^2(1 + 5/9 + (5/9)^2 + \dots)$ which is $n^2 \frac{1}{1-(5/9)} = \frac{9}{4}n^2$. For the case where n is a power of 3, we simply sum the first $\log_3 n + 1$ terms of the series, to get:

$$\begin{aligned}
&\frac{9}{4}n^2 \left(1 - \frac{5^{1+\log_3 n}}{9} \right) \\
&= \frac{9}{4}n^2 \left(1 - \frac{5}{9} \left(\frac{n^{\log_3 5}}{n^2} \right) \right) \\
&= \frac{9}{4}n^2 - \frac{5}{4}n^{1.4650}
\end{aligned}$$

3. Professor I.M. Cheep recently bought a bargain-basement “ePod”, a product similar to the iPod, but without quite as many nice features. The ePod stores all music in a flat list of the form artist-name.song-name. Songs cannot be shuffled, or organized by song-name. Instead, the ePod lets users define a display-order for artists (they need not be alphabetical). All songs by a given artist will appear consecutively. The ePod scrolls down one song on its list when you whack it firmly on concrete or other hard surface. Another single button lets you reset to the top of the list. Thus, when Professor Cheep wants to listen to songs by a particular group X , he pushes the reset, and then whacks it enough times to scroll past all songs of all artists appearing before group X . Because the ePod has an expected lifetime of only 20,000 whacks, he wants to organize his artists so that on average the ePod does not get whacked any more than necessary.

He formalizes the problem this way:

For each of n artists numbered $i = 1$ through n , a value s_i gives the number of songs stored for artist i , and a fraction $0 \leq f_i < 1$ for each i gives the frequency with which the professor intends to scroll to listen to songs by artist i . For a particular ordering of artists, the number of “whacks” required to scroll to artist i ’s songs (on average) is exactly $f_i \cdot \sum_{j \text{ before } i} s_j$. The average number of whacks is defined to be the sum, over all artists, of the number of whacks per artist.

For example, suppose the ePod database contains the following data:

Artist	Number of Songs	Access Frequency
Damien Rice	15	1/4
Puddle of Mudd	8	1/3
Ray Charles	12	1/6
Bebel Gilberto	5	1/4

Then the average whacks needed to scroll to (the top of) a desired artist’s songs with the ordering (Damien, Puddle, Ray, Bebel) is $1/4 * 0 + 1/3 * 15 + 1/6 * 23 + 1/4 * 35$.

The average whacks for the reverse ordering (Gilberto, Charles, Mudd, Rice) would be $1/4 * 0 + 1/6 * 5 + 1/3 * 17 + 1/4 * 25$.

Describe an algorithm that, given a collection of artists $1, 2, \dots, n$, number of songs s_i for each, and frequency of access f_i for each, outputs an ordering of artists so that the average number of whacks is minimized.

Prove that your algorithm is correct. Briefly explain the running time. Half credit will be given for a solution for the special case that $f_i = 1/n$ for each of the n artists.

SOLUTION: For each artist, let $r_i = \frac{f_i}{s_i}$. Sort the artists in decreasing (or non-increasing) order of r_i ’s, and output the resulting ordering.

Proof: The proof follows a standard exchange argument. Suppose by way of contradiction our ordering were not optimal; consider the optimal playlist, which must be different from (and better than) ours. There must be some pair of artists a, b such that artist a appears before

artist b in the optimal playlist, but artist b is placed before artist a by our algorithm; we say that such a pair is *inverted*.

If there are multiple optimal playlist, consider the one with fewest inversions. Artists a and b may not be adjacent in the optimal schedule, but there must be an adjacent pair i, j of inverted artists between them. (For proof of this claim, see the textbook, or the solution to problem 2 in HW1.) Now consider inverted artists i and j such that artist i occurs immediately before artist j in the optimal playlist with fewest inversions. Because our algorithm placed artist j before artist i , we know that $\frac{f_j}{s_j} \geq \frac{f_i}{s_i}$. We show that swapping artists i and j decreases the number of inversions in the playlist without increasing the average number of whacks. Consider the cost O of the original optimal playlist, and M of the modified playlist that is identical to the optimal schedule, but with i and j swapped. Let S_k be the total number of songs before artist k in the optimal playlist.

$$\begin{aligned}
 O &= \sum_{k=1}^{i-1} S_k f_k + f_i S_i + f_j (S_i + s_i) + \sum_{k=j+1}^n S_k f_k \\
 M &= \sum_{k=1}^{i-1} S_k f_k + f_j S_i + f_i (S_i + s_j) + \sum_{k=j+1}^n S_k f_k \\
 O - M &= (f_i S_i + f_j (S_i + s_i)) - (f_j S_i + f_i (S_i + s_j)) \\
 &= f_j s_i - f_i s_j \\
 &\geq 0 \quad \left(\text{Because } \frac{f_j}{s_j} \geq \frac{f_i}{s_i} \right)
 \end{aligned}$$

$O - M$ is non-negative, implying that our modified playlist M has average number of whacks at most that of O , with one fewer inversion. But this contradicts the assumption that O was the optimal playlist with fewest inversions. Therefore, there cannot exist a playlist better than ours.

Note that in the unweighted case, since all the frequencies are equal, we just wish to find an ordering that minimize the sum of the number of songs before each artist. This problem is identical to the sample problem where beer mugs are thrown in the western-themed bar; instead of a riding time for each patron, we have a number of songs for each artist.

The problem is also very similar to problem 2 from HW1 (4.13 from the textbook). There we wished to minimize the weighted sum of *completion times*; here we want to minimize the weighted sum of songs before each artist. If the HW problem had been to minimize the weighted sum of beginning times, the two problems would have been identical. Even now, the difference between the problems is very small; we sort by the ratio of the weight to processing time/number of songs, and the proof of correctness is essentially the same.

4. In order to ensure a more fair tournament, the organizers of a three-on-three basketball tournament have decided to categorize all players as short, medium, and tall, and require that each team contain exactly one player from each group. Further, the sum of the three heights should not exceed the maximum sum-of-heights which is given as H .

A player s on the short list wonders what the “tallest” team is that contains him... that is, among teams of the form $\{s, m, t\}$, where m and t are on the medium and tall lists, what is the maximum $s + m + t$ possible such that $s + m + t \leq H$. (Here we use s, m, t to denote the players as well as their heights).

For example, if $M = \{7, 8, 11\}$, $T = \{11, 12, 15\}$, $H = 30$, then when $s = 6$, the team of largest total height that does not exceed 30 is 29: $\{6, 8, 15\}$, or $\{6, 11, 12\}$, whereas if $s = 4$, then a team of total height 30 is possible $\{4, 11, 15\}$.

Give an efficient algorithm that takes as input two lists M and T , and a value s , and determines $\max\{s + m + t : m \in M, t \in T, s + m + t \leq H\}$, and outputs $m \in M$ and $t \in T$ realizing that maximum. “Efficient” here means anything better than the trivially obtainable $O(|M| * |T|)$. That is, to receive credit, your algorithm must have running time $o(|M| * |T|)$.

SOLUTION:

```

Sort the smaller set (suppose this is  $T$ )
 $max \leftarrow -\infty$ 
 $best_m \leftarrow -\infty, best_t \leftarrow -\infty$ 
for every  $m \in M$ 
     $a \leftarrow H - (s + m)$ 
    Perform a binary search in  $T$  for the largest value  $t$  less than  $a$ 
    if  $max < s + m + t$ 
         $max \leftarrow s + m + t$ 
         $best_m \leftarrow m$ 
         $best_t \leftarrow t$ 
end for
output  $max, best_m, best_t$ 

```

The running time for this algorithm is $O(|T| \log |T|)$ to sort, followed by $|M|$ binary searches, each of which takes $O(\log |T|)$ time. Therefore, the total running time is $O((|M| + |T|) \log |T|)$, (assuming that T is the smaller set; if M were smaller, we would first sort it, and then run $|T|$ binary searches in M ; the running time would then be $O((|M| + |T|) \log |M|)$). Clearly, this is better than $O(|M| * |T|)$.

The proof of correctness follows immediately from the algorithm; for each element m of M , we find the largest element of T that can form a team with s and m . That is, for s and each m , we find the tallest acceptable team containing s and m . We then find the maximum over all choices of m ; this is the tallest allowable team that includes s .

Note that you do not have to do a binary search; another acceptable approach would be to sort both M and T , and then for the last element in M , find the largest element t in T that

can be used. This can be done with a linear scan. Then, for the next smaller element in M , find the largest element of T that can be used; this element must be $\geq t$, and so we do not need to search the entire array T . Carefully implemented, this can be done in linear time. (Try to work out how.) The running time for this approach is $O(|M| \log |M| + |T| \log |T|)$ to sort, followed by $O(|M| + |T|)$, which is $O(|M| \log |M| + |T| \log |T|)$ overall. This is slightly worse than the previous method, but still much better than $O(|M| * |T|)$ if $|M|$ and $|T|$ are of comparable size.

5. Rectilinear (Manhattan) Skyline Problem.

Let each of n buildings on a line be simple rectangles, where building i has base $[a_i, b_i]$, and height h_i . Thus, a building is just a rectangle of height h_i that has the interval $[a_i, b_i]$ as its base. Imagine we are seeing the buildings of a city in silhouette, so that there can be overlap among the bases.

The *skyline* of a set of n buildings is a sequence of at most $2n$ pairs (x_j, m_j) such that each x_j is some a_i or b_i , $x_j \leq x_{j+1}$, and m_j is the maximum height of all buildings which span (at least) the half-open interval $[x_j, x_{j+1})$. Thus, $m_j = \max\{h_i : a_i \leq x_j \leq x_{j+1} \leq b_i\}$.

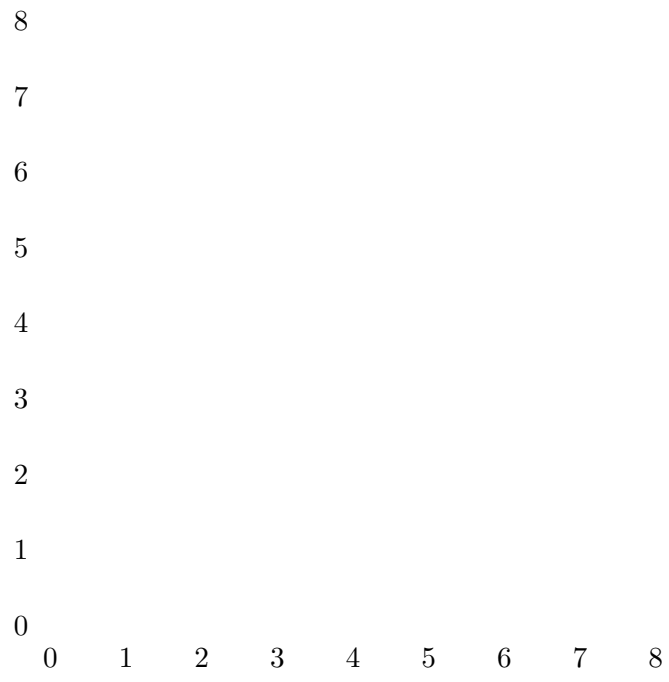
For example, if the buildings are:

Interval	Height
[1,3]	5
[2,4]	7
[3,6]	6
[4,5]	5

Then the skyline is the pairs (1,5), (2,7), (4,6) representing the following interval/max-height information:

Interval	Max-height
[1,2)	5
[2,4)	7
[4,6)	6

The figure below shows the buildings in the example above, and the resulting skyline in bold. (Note that the bold line should coincide with the thinner lines; it is slightly raised in the figure for clarity.)



- (a) Let S_1 be the skyline for the set of buildings A_1, \dots, A_n all defined on some interval, and let S_2 be the skyline for the set of buildings B_1, \dots, B_m defined on that same interval. Give a linear time algorithm for computing the skyline of the set of buildings $A_1, \dots, A_n, B_1, \dots, B_m$. Describe how your algorithm works briefly. You need not prove that it is correct.
- (b) Give an $O(n \log n)$ algorithm for computing the skyline of a set of n buildings. Hint: one way is to use part (a), though there are other ways that are fairly straightforward.

SOLUTION:

```

( $x_a, m_a$ )  $\leftarrow$  first pair from  $S_1$ 
Remove first pair from  $S_1$ 
( $x_b, m_b$ )  $\leftarrow$  first pair from  $S_2$ 
Remove first pair from  $S_2$ 
 $lastPosition \leftarrow x_a$ 
 $lastHeight \leftarrow 0$ 
while( $S_1$  and  $S_2$  are not empty)
     $newHeight \leftarrow \max\{m_a, m_b\}$ 
    if ( $newHeight \neq lastHeight$ )
        output ( $lastPosition, newHeight$ )
         $lastHeight \leftarrow newHeight$ 
    ( $next_a, temp$ )  $\leftarrow$  first pair from  $S_1$ 
    ( $next_b, temp$ )  $\leftarrow$  first pair from  $S_2$ 
    if ( $next_a < next_b$ )
        ( $x_a, m_a$ )  $\leftarrow$  first pair from  $S_1$ 
        Remove first pair from  $S_1$ 
         $lastPosition \leftarrow x_a$ 
    else
        ( $x_b, m_b$ )  $\leftarrow$  first pair from  $S_2$ 
        Remove first pair from  $S_2$ 
         $lastPosition \leftarrow x_b$ 
end while

```

This algorithm actually finds an *optimal skyline*. That is, it describes the skyline using as few pairs as possible; your solutions were not required to find an algorithm with this property, though.

The algorithm maintains the variable $lastPosition$; at any point in the execution of the algorithm, the skyline upto $lastPosition$ has been output, and m_a and m_b are (respectively) the heights of the skylines of sets A and B at $lastPosition$. The greater of these determines the height of the skyline of the combined set at $lastPosition$. The next position the skyline can change is either a point where the skline of A changed, or the skyline of B did. We find which changed first (say A did), and move to this point. This becomes the new $lastPosition$,

and we update the value m_a . This process is repeated in the while loop until the end of the interval is reached.

The running time is linear, because in each iteration of the while loop, we remove a pair from either S_1 or S_2 . Since $|S_1| \leq n$ and $|S_2| \leq m$, there are at most $n + m$ iterations of the loop, each taking constant time.

(b) Split the buildings into two halves arbitrarily. Find the skyline for each part recursively, and then combine the two skylines to find the skyline for the entire set in linear time, using the method of part (a). The base case is when there is only 1 building in the subproblem, which is trivial.

The recurrence for the running time of this algorithm is

$$T(n) = 2T(n/2) + O(n) = O(n \log n).$$