CSE 101 Practice Midterm, Nov 4
Topics: Order, Recurrence Relations, Analyzing Programs,
Divide-and-Conquer, Back-tracking
Answer Key

Order: Let $f(n) = \sum_{i=1}^{i=n} i^2$. Show that $f(n) \in \Theta(n^3)$.

This kind of function comes up in an analysis of a loop like: For I =1 to n: For J=1 to I For K=1 to I do: (constant time)

There are a few ways of doing this one. The simplest way if you notice it is that the sum has a closed form expression given in the text. (I hadn't realized that.)

However, we don't need a closed form expression to show that it is $\Theta(n^3)$. We can go by the definition of Θ: There exist constants n_0, $c_0, c_1 > 0$ so that $c_0 n^3 \leq f(n) \leq c_1 n^3$ for all $n \geq n_0$.

To see the upper bound, $f(n) \in O(n)$, we simply bound each term by the largest possible value. Since $i^2 \leq n^2$ for $1 \leq i \leq n$, $\sum_{i=1}^{i=n} i^2 \leq \sum_{i=1}^{i=n} n^2 = n^3$. So we can pick $c_1 = 1$ in the definition of O.

If we try the same idea for the lower bound directly, we would lower bound each term by 1, giving a sum of n. But most of the terms are bigger, which we can use by bounding the sum by the sum of the bigger half of the terms: $\sum_{i=1}^{i=n} i^2 \geq \sum_{i=n/2+1}^{i=n} i^2 =\geq (n/2)(n/2)^2 = n^3/8$. So we can pick $c_0 = 1/8$ in the definition of Ω.

Another method is to prove the claim by induction. Logically, we need to pick c_0 first but we can wait until we start the proof before we back up and see which value to pick.

Pick $c_0 =?$ (eventually, we'll use ?=1/3).

We prove by induction that $f(n) \geq c_0 n^3$. First, the base case is that $f(1) = 1 \geq c_0 * 1$. (Note to self: $c_0 \leq 1$.) Since $c_0 \leq 1$, the base case holds.

Assume $f(n) \geq c_0 n^3$ Then $f(n+1) = f(n) + (n+1)^2 \geq c_0(n^3) + (n+1)^2$ (Note to self on scratch paper: we want $c_0 n^3 + (n^2 + 2n + 1) \geq c_0(n+1)^3 = c_0 n^3 + 3c_0 n^2 + 3c_0 n + c_0$, which is the same as $n^2 + 2n + 1 \geq 3c_0 n^2 + 3c_0 n + c_0$. We can see that if $c_0 \leq 1/3$, all the coefficients on the left are smaller than co-efficients on the right. At this point we go back and pick $c_0 = 1/3$.) $c_0 n^3 + (n+1)^2 = c_0 n^3 + n^2 + 2n + 1 > c_0 n^3 + 3(1/3)n^2 + 3(1/3)n + 1/3 = c_0(n^3 + 3n^2 + 3n + 1) = c_0(n+1)^3$. So the claim is also true for $n+1$.

By induction, $f(n) \geq 1/3n^3$ for all n.

Divide-and-Conquer Analysis Consider the following algorithm, to find the largest consecutive sum of elements in an integer array A, i,e $Max_{1 \leq I \leq n, 1 \leq J \leq n} \sum_{K=I}^{K=J} A[K]$. (We count an empty sum as 0, so MCS is never negative.)

$MCS(A[1...n]$:array of integers): integer;

1. If $n = 1$ return $Max(A[1], 0)$;

2. $m \leftarrow \lfloor n/2 \rfloor$.
3. $S_1 \leftarrow MCS(A[1..m])$;{* finds the largest consecutive sum totally within first half*}
4. $S_2 \leftarrow MCS(A[m+1..n])$;{* finds the largest consecutive sum totally within second half *}
5. $Biggest \leftarrow 0; Sum \leftarrow 0$
6. FOR $I = m$ TO 1 do:
7. begin;{*for loop*}
8. $Sum \leftarrow Sum + A[I]$
9. IF $Biggest < Sum$ THEN $Biggest \leftarrow Sum$.
10. end;{*for loop*}
11. $S3 \leftarrow Biggest$ {* S3 is the biggest sum that goes from A[I]...A[m] *}
12. $Biggest \leftarrow 0; Sum \leftarrow 0$;
13. FOR $I = m + 1$ TO n do:
14. begin;{*for loop*}
15. $Sum \leftarrow Sum + A[I]$
16. IF $Biggest < Sum$ THEN $Biggest \leftarrow Sum$.
17. end;{*for loop*}
18. $S3 \leftarrow S3 + Biggest$ {* Adds in the biggest sum that goes from A[m+1]...A[J], to get the biggest sum of a series that contains part of both halves*}
19. Return $Max(S_1, S_2, S_3)$

(For example, on the array $2, -6, 3, -2, 8, -4, 10, -2$, the algorithm first recursively computes 3 as the MCS of the left half, 14 as the MCS on the right half. Then it computes all sums in the first half starting from -2, -2, 1, -5, -3 and takes the largest, 1. Then it does the same on the right half, 8,4,14,12, and adds the largest, 14, to the 1, giving 15. Since this was larger than either side, the MCS is 15= 3-2+8-4+10.)

Give a recurrence relation for the time complexity of the algorithm, and use it to give a time analysis, up to order.

The main procedure calls itself recursively twice, in lines 2 and 3. Both times, it calls itself on arrays of size $n/2$. Then the rest of the algorithm is dominated by two linear time loops. Thus, $T(n) = 2T(n/2) + O(n)$. By Theorem B.6., with $a = 2, b = 2, k = 1$, since $a = 2 = 2^1 = b^k$, we can apply the steady-state case, and $T(n) \in O(n \log n)$.

Divide-and-conquer, variable sizes A binary tree data structure is used to store bank transactions. Each node x, in addition to pointers $left(x)$, $right(x)$ and $parent(x)$, has two real-valued fields $time(x)$ and $amount(x)$,

the time and amount of the deposit stored at x. You want to given a pointer to the root x and a time T, find the customer's balance at time T, i.e., $\sum_{\{y|time(y)\leq T\}}(amount(y))$. (If a node has no child or parent, the corresponding pointer fields return NIL. Following a pointer and arithmetical operations are constant time.)

Give an $O(n)$ algorithm to solve this problem, and prove your time analysis.

We recursively compute the sums in the two subtrees. Then we need to add $amount(x)$ if that transaction occured before time T.

This gives us: RecBalance(x:node, T: real)

1. IF $x = NIL$ return 0.
2. $LeftBalance \leftarrow RecBalance(left(x), T)$
3. $RightBalance \leftarrow RecBalance(right(x), T)$
4. $Balance \leftarrow LeftBalance + RightBalance$
5. $IF(time(x) \leq T$ THEN $Balance \leftarrow Balance + amount(x)$
6. Return Balance.

Let l be the number of nodes in the subtree rotted at $left(x)$, r at right x. $n = l + r + 1$ is the total number of nodes. Since the algorithm calls itself rcursively on the two subtrees, and then does a constant number of operations, $T(n) = T(l) + T(r) + O(1)$. As we saw in class, this leads to $T(n) \in O(n)$.

Back-tracking: Give a back-tracking algorithm for the following problem:

There are m types of machines $M_1...M_m$, and n jobs $J_1, ..J_n$. For each machine M_i, you are given a list $CanDo_i$ of jobs that machine can do, and a positive integer price $Price_i$. You want to purchase a set of machines so that each job can be done by at least one machine purchased, and wish to minimize the total prices of purchased machines. Your algorithm should compute the lowest possible price of such a set of machines.

(For example, if M_1 costs 3\$ and can do J_1, J_2, M_2 costs 4 \$ and can do J_1, J_3, J_4, and M_3 costs 2 \$ and can do J_2, J_3, J_4, then our best purchase is to buy M_1 and M_3 for a total of 5\$.)

If we buy M_1, we need to purchase an additional set of machines in $M_2, ..M_n$ to perform all the Jobs except $CanDo_1$. If we do not buy M_1, we need to peform all the jobs with future purchases in $M_2, ..M_n$.

This leads to the following back-tracking algorithm:

BTCost(Machines[1..n], CanDo[1..n], Jobs);

1. If —Jobs—=0 return 0
2. If there is a job not in $CanDo_i$ for any i, return inf

3. $buy \leftarrow BTCost(Machines[2,..n], CanDo[2,..n], Jobs - CanDo_1) + Price(1)$
4. $passup \leftarrow BTCost(Machines[2,..n], CanDo[2,..n], Jobs)$
5. Return $Min(buy, passup)$.

If all you want is to see an adequate solution for the exam, you can stop reading now.

We could make the above a little more efficient by putting in some other cases :

If $CanDo_1 \cap Jobs$ is empty, don't compute buy , just return $passup$. M_1 doesn't do any new job, so don't buy it.

If there is a job that can only be done by one machine, only compute the buy relative to that machine. We need to buy it eventually.

If $CanDo_1 \cap Jobs = J_i$, a single new job, there must be another machine that can do J_i, call it M_k. We will never buy M_1 unless we don't buy M_k, so in the case when we buy M_1, we can also delete M_k from the list of machines.

For a time analysis, which wasn't required, we can see that we delete at least one machine every time we make a recursive call, and we make at most 2 recursive calls. Therefore, we will have a binary tree of height at most n that represents a run of the algorithm, so the total time will be $O(2^n)$.

However, other analyses are possible, in terms on n, m. We never increase n or m. If we put in the additional cases, in the buy case, we either decrease n by 1 and m by at least 2, or decrease n by 2. In the $passup$ case, we decrease n by 1. Let $P = n + m/2$. If we measure progress in terms of P, we always decrese P by 2 in the first case, and 1 in the second case. So $T(P) \leq T(P-2) + T(P-1)$, which is the same as the fibonnacci sequence, which grows about $2^{.7P} = 2.7n + .35m$. This will be a better analysis if the number of jobs is substantially smaller than the number of machines, but worse if the number of jobs is larger. Both are valid upper bounds, so we can say $T(n, m) \in O(min(2^n, 2^{.7n+.35m}))$