Lecture #11: Wednesday, 10 February 2016
Topics: **Example midterm problems and solutions from a long time ago**[*]**— Spring 1998**

Problem 1. [15 points] Recurrences and Asymptotics

Give asymptotic 'Θ' solutions for the following recurrences; **argue** that your solution is correct. Assume $T(1) = 1$.

(a) $T_1(n) = 6T_1(\lfloor n/3 \rfloor) + n^2 \lg n$ [2 points]

Using the Master theorem we get

$$\begin{aligned} a &= 6 \\ b &= 3 \\ f(n) &= n^2 \lg n \\ n^{\log_b a} &= n^{\log_3 6} \end{aligned}$$

Since $\log_3 6 < 1.64$, *there exists* $\varepsilon > 0$ *(for example,* $\varepsilon = 0.1$*) such that* $f(n) = n^2 \lg n = \Omega(n^{\log_3 6+\varepsilon})$, *and so case 3 of the Master theorem seems to apply. But we must also check the regularity condition: can we find a* $c < 1$ *such that*

$$\begin{aligned} af\left(\frac{n}{b}\right) &= 6\left(\frac{n}{3}\right)^2 \lg \frac{n}{3} \\ &= \frac{6}{9}n^2(\lg n - \lg 3) \\ &= \frac{2}{3}n^2 \lg n - \frac{2}{3}n^2 \lg 3 \end{aligned}$$

is no greater than

$$cf(n) = cn^2 \lg n$$

for all sufficiently large n*? Yes. If* $c = 2/3$, *then the regularity condition is satisfied as long as* $n \geq 1$. *We conclude that*

$$T(n) = \Theta(n^2 \lg n).$$

Common error: Setting $\log_3 6 = 2$

Graded by Suresh.

[*]Note that textbook refs here are to the first edition.

(b) $T_2(n) = 3T_2(\lfloor n/3 \rfloor) + n\lg^3 n$ [2 points]

Assuming n is a power of 3, but without compromising the generality of our final result as we observed in class, we use the iteration method to get

$$\begin{aligned}
T_2(n) = & \ 3T_2(n/3) + n\lg^3 n \\
= & \ n\lg^3 n + 3[\, 3T_2(n/9) + n/3 \cdot \lg^3(n/3) \,] \\
= & \ n\lg^3 n + n\lg^3(n/3) + 9[\, 3T_2(n/27) + n/9 \cdot \lg^3(n/9) \,] \\
\cdots & \\
= & \ n\lg^3 n + n\lg^3(n/3) + n\lg^3(n/9) + \cdots + n\lg^3(n/(n/3)) + nT_2(1).
\end{aligned}$$

The last sum in the above expression contains $\Theta(\lg n)$ terms, each no greater than $n\lg^3 n$, and so $T_2(n) = O(n\lg^4 n)$.

On the other hand, the first half of the $\lg^3$ terms — or, to be precise, the first $m = \lfloor \frac{\log_3 n}{2} \rfloor$ terms — are each at least as big as $n\lg^3 \sqrt{n} = (n\lg^3 n)/8$ — or, to be precise again, at least as big as

$$n\lg^3 \frac{n}{3^{m-1}}.$$

So $T_2(n) = \Omega(n\lg^4 n)$. Together we get $T_2(n) = \Theta(n\lg^4 n)$.

We can also get this result directly using the version of the Master theorem given in class, and appearing as exercise 4.4-2 on page 72 of CLR.

It is also possible to solve this problem using the tree method. However, we should be careful to show both a lower and an upper asymptotic bound of $n\lg^4 n$. That is, arguing that the tree has depth at least $\log_3 n$, with each level costing at most $n\lg^3 n$ successfully proves that $T_2(n) = O(n\lg^4 n)$ — but we must also show that $T_2(n) = \Omega(n\lg^4 n)$ to conclude that $T_2(n) = \Theta(n\lg^4 n)$. This can be done by considering the cummulative cost of just half the tree levels, in a manner similar to our previous derivation using the iteration method.

Common error: *We cannot apply the basic form of the Master theorem (page 63 of CLR) since*

$$\begin{aligned}
a &= 3 \\
b &= 3 \\
f(n) &= n\lg^3 n \\
n^{\log_b a} &= n^{\log_3 3} = n
\end{aligned}$$

and, even though $n\lg^3 n = \Omega(n)$, it is not $\Omega(n^{1+\varepsilon})$ for any $\varepsilon > 0$. *Why? Because if $n\lg^3 n = \Omega(n^{1+\varepsilon})$, we could deduce that $\lg^3 n = \Omega(n^\varepsilon)$, which contradicts the formula on page 35 of CLR. So case 3 does not apply.*

Common error 2: *When we obtain the summation*

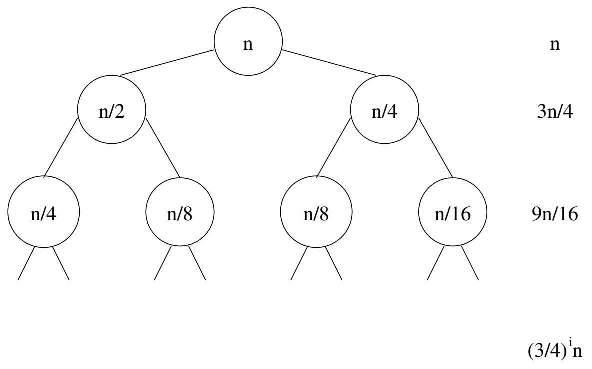
$$n\lg^3 n + n\lg^3(n/3) + n\lg^3(n/9) + \cdots + n\lg^3(n/(n/3))$$

in the recurrence, this **cannot** *be approximated as $O(\lg^3 n)$. A simple way to analyse this would be to observe that each term is $O(\lg^3 n)$ and there are $O(\lg n)$ such terms, yielding an upper bound of $O(\lg^4 n)$.*

Graded by Suresh.

(c) $T_3(n) = T_3(\lfloor n/2 \rfloor) + T_3(\lfloor n/4 \rfloor) + n$ [2 points]

Using the tree method we get the following tree



Note that the tree will be unbalanced and that the subtree corresponding to $T_3(\lfloor n/2 \rfloor)$ will be deeper (i.e. it will have more levels). At all levels after the shallow subtree — the one corresponding to $T_3(\lfloor n/4 \rfloor)$ — becomes exhausted, each level will cost no more than $\left(\frac{3}{4}\right)^i n$. So we have

$$T_3(n) \leq \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i n = O(n).$$

Now the top level costs n, and hence

$$T_3(n) \geq n = \Omega(n).$$

Combining the two, we concude that

$$T_3(n) = \Theta(n).$$

Common error: *"Since there are $O(\lg n)$ levels, and each level costs $O(n)$, the total cost is $O(n \lg n)$". The problem with this argument is that it is loose. There are $O(\lg n)$ levels, but the cost at each level decreases. In fact, the cost at level i is $n\left(\frac{3}{4}\right)^i$. Summing up, we obtain the above bound of $O(n)$.*

Graded by Suresh.

(d) For each functions$f(n)$ along the left side of the table below and each function $g(n)$ across the top, write O, Ω, or Θ in the appropriate box, depending on whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. If there is more than one relation between $f(n)$ and $g(n)$, write only the strongest one. The first line is a demo solution. [1 point per entry]

$f(n) \mid g(n)$	$\lg n$	$n \lg n$	4^n
n^2	Ω	Ω	O
$T_1(n)$			
$T_2(n)$			
$T_3(n)$			

The answers are:

$f(n) \mid g(n)$	$\lg n$	$n \lg n$	4^n
$T_1(n) = \Theta(n^2 \lg n)$	Ω	Ω	O
$T_2(n) = \Theta(n \lg^4 n)$	Ω	Ω	O
$T_3(n) = \Theta(n)$	Ω	O	O

Common error: *$n \lg^3 n$ is **not** $O(n \lg n)$.*

Graded by Suresh.

Problem 2. [18 (3 points per question)] **Potpourri**

For each of the statements below:

- circle whether is TRUE or FALSE
- **Explain** your answer in one or two sentences

(a) An adversary places a bet that he can select an input so that when you run RANDOMIZED QUICK-SORT once on his input, the algorithm will take $\Omega(n^2)$ steps. There is a positive (i.e, non-zero) probability that the adversary will win his bet.

TRUE FALSE

The answer is TRUE. *In fact, the adversary does not even have to think hard: any input s/he gives to randomized* QUICK-SORT *has a non-zero probability of forcing the algorithm through $\Omega(n^2)$ steps. The reason is that the running time of randomized* QUICK-SORT *depends on the internal, random selection of pivots; hence, it is not impossible (i.e. there is a positive probability) that the selection will be bad at every single recursive step, partitioning the array into subproblems of sizes $n-1$ and 1.*

Also, if the input consists of n repetitions of the same number, then a QUICK-SORT *which uses the* PARTITION *procedure shown in class will* definitely *take $\Omega(n^2)$ steps, whether it's deterministic or randomized.*

Graded by David.

(b) A *max*-based heap can be transformed into a *min*-based heap in linear time.

TRUE FALSE

The answer is TRUE. *In fact,* any *array (i.e. not necessarily a heap) can be turned into a* min-*based heap in linear time. The way to do this is to simply run* BUILD-HEAP *on the array, which takes linear time. Here, of course, we are using a* BUILD-HEAP *procedure based on a new* HEAPIFY *which, in turn, guarantees that every parent is no greater than its children.*

Although it is possible to build a min-*based heap by extracting the elements of the* max-*based heap in reverse sorted order, this algorithm takes $\Theta(n \lg n)$ steps.*

It is not possible to convert a max-*based heap into a* min-*based heap by reversing the elements in the heap array. As a counterexample, consider the heap stored as*

$$5, 1, 4$$

This heap appears on the right side in the figure below. Reversing the contents of the array produces

$$4, 1, 5$$

which is not a min-*based heap since the root, 4, is bigger that its left child, 1.*

Graded by David.

(c) A **preorder** tree walk on a (max-)heap will output the values stored in the heap in reverse sorted order.

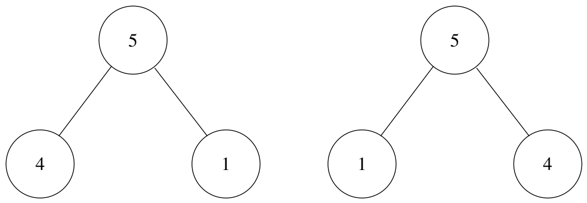
TRUE FALSE

The answer is FALSE. *Or, to be more precise, not necessarily. For example, the left heap in the figure below produces*

$$5, 4, 1$$

during a preorder walk, while the right heap produces

$$5, 1, 4.$$



One counterexample suffices to show the fallacy of the claim. Another, more general way to prove that no tree walk (whether inorder, preorder, or postorder) can output the nodes in reverse sorted order is to simply observe that a heap guarantees only *that a parent is no smaller that its children; there is no guarantee as to the relative ordering of the two children.*

Graded by David.

(d) Hashing with chaining is preferable to open addressing, if the load factor α of the hash table is likely to get greater than 1.

TRUE FALSE

The answer is TRUE. *If we have $\alpha > 1$ then the number of elements to be hashed is greater than the number of slots available, so we can't use open addressing. Even when α doesn't exceed 1 but it is close to 1, chaining is much better since clustering will make open addressing very inefficient.*

Students who argued the inefficiency of open addressing for α close to 1 but didn't say that open addressing fails for $\alpha > 1$ lost only 1 point.

Graded by David.

(e) If we know the ordered lists of keys produced by a **preorder** and an **inorder** traversal of a binary search tree T, then we have enough information to fully reconstruct T.

TRUE FALSE

The answer is TRUE. *In fact we don't even need the inorder output because we have all the information we need to reconstruct the tree from the preorder output. (Actually, we could get the inorder output by sorting the preorder output, if we really wanted to.) Now, since a preorder traversal outputs the root of the tree before either subtree, then to reconstruct the tree we only need to do the following:*

1. *Initialize the tree to be empty.*
2. *Insert the elements in the order given by the preorder output.*

We'll get the original tree because the root of the original tree (and every subtree recursively) is inserted into the new tree before any of its original subtrees. Thus the structure of the new tree is exactly the same as that of the original tree.

Another way to think about it is the following: imagine that every element we visit in a preorder traversal is deleted from the tree and inserted in a second (originally empty) tree. It will go exactly at the same (corresponding) position from which it was deleted.

Graded by David.

(f) During a Red/Black tree INSERT, $\Theta(\log n)$ tree links are modified in the worst-case.

TRUE FALSE

The answer is FALSE. *In the worst case may we need to move all the way up the tree from the new leaf to the root; however, the only operation that propagates this way is the* COLOR-FLIP *operation that does not modify tree links. Balance can always be restored at the end by one or two rotations, each of which modifies only three links.*

Graded by David.

Problem 3. [21 points] Non-comparative Sorts

Let $n = 2^k - 1$. An array $A[1,...,n]$ contains all k-bit strings *except one*. The only operation we can use to access the array A is "fetch the j-th bit of $A[i]$", which takes constant time. Show that we can determine the missing string in $O(n)$ time.

In the first phase of the algorithm, we look at the least significant bit of each number we are given (i.e. FETCH$(x,1)$ *for each number $A[i]$ — a total of n probes). Let S_b, $b \in \{0,1\}$, be the set of numbers with b as their least significant bit. If $|S_1| < n/2$ then there must be at least one number in the range $[0,n]$ which is not in the sequence of given numbers and which has a 1 as its least significant bit. Similarly, if $|S_0| < n/2$, then there must be at least one number in the range $[0,n]$ which is not in the sequence of given numbers and which has a 0 as its least significant bit. We now eliminate the numbers in the larger set since these numbers all differ from the number we are looking for in the least significant bit, and we record the value of the least significant bit.*

We now proceed to probe the second least significant bit of the remaining set of numbers (the size of this set is less than $n/2$), form S_0 and S_1 as above, record the second least significant bit and again eliminate the larger set.

We continue this process until we determine all the bits of the missing number. That is, in the j-th phase of this algorithm, we look at the j-th least significant bit of the remaining numbers (at most $\frac{n}{2^{j-1}}$ probes), and eliminate at least half of these numbers for the next phase. In short, we go through a total of k phases and deduce the missing string bit-by-bit; alternatively, we can go through $k-1$ phases and determine the missing string by flipping the most significant bit of the single member of $S_0 \cup S_1$ (i.e. the one and only string that survived all $k-1$ phases).

Since this algorithm eliminates at least half the sequence of numbers it looks at in each phase, it takes at most $\lg n$ phases to finish. In the j-th phase, the algorithm makes at most $\frac{n}{2^{j-1}}$ calls to the procedure FETCH. *Thus the total number of probes is at most:*

$$\sum_{j=0}^{\lg n} \frac{n}{2^j} = n \sum_{j=0}^{\lg n} \frac{1}{2^j} \leq 2n$$

Several other correct but (asymptotically) slower solutions are possible. They all end up looking at every bit of every string, thus taking $\Theta(kn) = \Theta(n \lg n)$ time to complete; recall that, as $n = 2^k - 1$, k is by no means a constant — instead, $k = \Theta(\lg n)$. Examples include:

- *Create an array $B[0,\ldots,n]$ of booleans, all initialized to* FALSE. *Go through all elements of A, extracting each $A[i]$ bit by bit and interpreting it as a binary number; then set $B[A[i]]$ to* TRUE.

Finally, go through all elements of InA, and find the one, say it's $B[j]$, which is still FALSE*; the missing string is the binary representation of j.*

- *Perform k passes over A. During pass j, extract the jth bit of all the elements of A, counting separately the zeros and the ones. If there are more (less) zeros than ones, conclude the pass by setting the jth bit of the missing string to one (zero).*
- *Set the missing string M to 0. Go through all elements of A, extracting each $A[i]$ bit by bit and interpreting it as a binary number; then XOR this number with M, storing the result back in M. When you are done, M contains the missing string.*
- *Use a non-comparative sort to order the elements of A, interpreted as binary numbers, in increasing order. Then look at the least significant bits of adjacent elements in the sorted array A. If you ever find two identical bits in adjacent elements $A[i]$ and $A[i+1]$, this means that the missing string is $A[i]+1$. To cover the cases where either the string $00\ldots00$ or the string $11\ldots11$ is missing, simply compare the first and last elements of A against $00\ldots01$ and $11\ldots10$, respectively.*

Common errors:

***1. If** $n = 2^k - 1$**, then** $k = \lg(n+1) \neq O(1)$. This is an error that nearly everyone made. It is essential to note here that k is not an independent parameter. It is defined in terms of n, and since we can solve for $k = \lg(n+1)$, we cannot claim that k is a constant. All algorithms that run in time $O(kn)$ run in time $O(n\lg n)$, which is not linear.*

***RADIX-SORT does not run in "linear" time** Given n input numbers that are d digits long, radix sort runs in time $O(dn+kd)$, where $k = O(2^d)$ is the range of each digit. In our case, $d = \lg n$, hence RADIX sort, runs in time $O(n\lg n)$.*

Graded by Suresh.