CS 5006 Final

Northeastern University— Seattle

Summer 2018

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

Name: _____

1. Consider a recursive divide and conquer algorithm that satisfies the following recurrence on its running time: $T(n) = 4T(n/3) + n$, with $T(1) = 1$. In the following you may assume that n is a power of 3.

   (a) How many subproblems are there at depth k in the recursion tree? (The number of subproblems at depth 0, namely at the root of the tree, is 1.) (4)

   Solution:
   4^k

   (b) What is the size of each subproblem at depth k of the recursion tree? (The size of the subproblem at depth 0, namely at the root of the tree, is n). (4)

   Solution:
   $\frac{n}{3^k}$

   (c) What is the running time $T(n)$ of this algorithm? (4)

   Solution:
   Running time is bounded by [amount of work done at leaves] + [sum over levels of work done on each level]
   Number of levels: $\lg n - 1$
   ~~Number of subproblems at depth k: 4^k~~
   Size of subproblem at depth k: $n/3^k$
   Recurrence:

$$\begin{aligned}\sum_{k=0}^{\lg n-1} 4^k \cdot \frac{n}{3^k} &= n \sum_{k=0}^{\lg n-1} \frac{4^k}{3^k} \\ &= n \sum_{k=0}^{\lg n-1} (\frac{4}{3})^k \\ &= n \frac{1-(4/3)^{\lg n}}{1-(4/3)} \\ &\dots\end{aligned}$$

2. Solve the following recurrences. Show your work.

(a) (4)

$$T(n) = \begin{cases} 3T(n/3) + n^2 & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Solution: $O(n^2)$

(b) (4)

$$T(n) = \begin{cases} 4T(n/3) + n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Solution:
$4^{\log_3 n} = n^{\log_3 4}$

(c) (4)

$$T(n) = \begin{cases} T(n/3) + 1 & \text{if } n > 1 \\ 0 & \text{if } n = 0 \end{cases}$$

Solution: $O(\log n)$

3. Answer Yes or No to the following questions. Show your work for partial credit.

   (a) A divide and conquer algorithm for $n \times n$ matrix multiplication that solves the problem using 24 matrix multiplication subproblems of size $n/3$ and $O(n^2)$ additional work for recombining would be more efficient than the usual $\Theta(n^3)$matrix multiplication algorithm. (5)

Solution:

$$T(n) = 24T(n/3) + O(n^2)$$

* Looking for the recurrence, solution, and comparison to the given $\Theta(n^3)$

   (b) A divide and conquer algorithm for the selection problem that reduced the problem on lists of size n to one selection problem of size $3n/20$ and another selection problem of size $9n/10$ plus a linear amount of extra work would yield a linear time algorithm. (5)

Solution:

$$T(n) = T(3n/20) + T(9n/10) + O(n)$$

* Looking for the recurrence, solution, and comparison to the given $O(n)$

4. In World War I, the army was testing recruits for syphilis, which was rare, but required a time-consuming though accurate blood test. They realized that they could pool the blood from several recruits at once and save time by eliminating large groups of recruits who didn't have syphilis.

   But, today, you are given a sequence of n bits $x_1, ..., x_n$, where each $x_i \in \{0, 1\}$. I want you to design an algorithm that outputs either

   - Any i such that $x_i = 1$
   - 0 if the input is all 0s.

   The only operation you are allowed to use to access the inputs is a function GROUP-TEST where:

$$\text{GROUP-TEST}(i, j) = \begin{cases} 1 & \text{if some bit in } x_i, \ldots, x_j \text{ has value } 1 \\ 0 & \text{if all bits in } x_i, \ldots, x_j \text{ have value } 0 \end{cases}$$

   (a) Design a divide and conquer algorithm to solve the problem that uses only $O(\log n)$ calls to GROUP-TEST in the worst case. Your algorithm should never access the x_i directly. (10)

Solution: CODE IS NOT QUITE RIGHT

```
FINDTHEBIT(X)
1   // X is a sequence of n bits: x_1, ..., x_n
2   i = 1, j = n
3   while i < j
4       mid←⌊i + j/2⌋
5       if GROUP-TEST(i, mid) == 1
6           j = mid
7       else
8           i = mid + 1
9
10      Return Group-Test(i, j)
11
```

LOOK FOR: split the input, choose one half to check.

Log n calls to GROUP-TEST.

(b) Show that your algorithm above produces the correct result. (10)

Solution:

Our goal is to find a single index i that is a 1, if it exists.

Given an interval i, j, we partition it into two (almost) equal size intervals. If there is a 1 bit in the first interval, we'll continue to recursively search the first interval until we find the 1 bit.

If there is not a 1 bit in the first half interval, we check the second half interval. If GROUP-TEST returns 0 on that, then there are no 1s in the input and we can return 0. Otherwise, we recurse on that second-half interval.

(c) Briefly justify your bound on the number of calls. (5)

Solution: The length of the interval decreases by a factor 2 each time. So, we are only going to call GROUP-TEST at most $O(\lg n)$ many times.

5. You are given a directed acyclic graph $G = (V, E)$ and a node $t \in V$. Design a linear time algorithm to compute for each vertex $v \in V$ the number of different paths from v to t in G. Analyze its running time in terms of $n = |V|$ and $m = |E|$. (25)

   Be sure to address correctness and runtime.

   HINT: Start with a topological sort.

Solution: Compute a topological sorting of the vertices and let's rename them to 1, 2, . . . , n such that for every edge (i, j) we have i < j. Now, let OP T(i) be the number of paths from vertex i to t. Observe that if i > t then OP T(i) = 0 because every path from i only goes to vertices with higher index but t < i.

Now, let's compute OP T(i). Obviously OP T(t) = 1 because if we go out of t we cannot get back to t.

Now, suppose we have computed OP T(j) for all $i < j \leq t$. We compute OP T(i). Let's try to characterize OP T(i). The first edge of OP T(i) out of vertex i goes to a neighbor of i say j. At this moment we don't know j but later we will brute force for all possibilities of j.

Observe that by IH there are OP T(j) many paths from j to t. Any of paths from j to t will give us a path from i to t once we also include the edge (i, j). Therefore in this case we get

$$OPT(i) = OPT(j).$$

Taking all possibilities of j we get

$$OPT(i) = \sum_{j:(i,j)\in E} OPT(j)$$

In the above sum we have counted all paths because every path out of i starts with an edge (i, j) and we have not double counted any path because if the starting edge of two paths are different, they are different paths. So, we get the recurrence

$$OPT(i) = \begin{cases} 0 & \text{if } i > t \\ 1 & \text{if } i = t \\ \sum_{j:(i,j)\in E} OPT(j) & \text{otherwise} \end{cases}$$

Running Time: The algorithm takes deg(i) steps to compute M[i] for each vertex i < t.

Find a topological sorting of G and rename vertices such that i < j for all (i, j) ∈ E.

```
1  for i = t + 1 → n
2        M[i] = 0
3  M[t] = 1
4  for i = t − 1 → 1
5        M[i] = 0
6        for each edge (i, j) ∈E
7              M[i] = M[i] + M[j]
```

For $i \geq t$, we compute $M[i]$ in $O(1)$ steps. Therefore the algorithm runs in O(P i deg(i)) = O(n + m) in the worst case.

6. The two processor interval scheduling problem takes as input a sequence of request intervals $(s_1, f_1), ..., (s_n, f_n)$ (10) just like the unweighted interval scheduling problem except that it produces two disjoint subsets $A_1, A_2 \subset [n]$ such that all requests in A_1 are compatible with each other and all requests in A_2 are compatible with each other and $|A_1 \cup A_2|$ is as large as possible.

   (A_1 might contain requests that are incompatible with requests in A_2.) Does the following greedy algorithm produce optimal results? If yes argue why it does; if no produce a counterexample.

   DoThing($\{(s_1, f_1), \ldots, (s_n, f_n)\}$)

   1 Sort requests by increasing finish time
   2 $A_1 = \emptyset$
   3 $A_2 = \emptyset$
   4 **while** there is any request (s_i, f_i) compatible with either A_1 or A_2:
   5 Add the first unused request, if any, compatible with A_1 to A_1.
   6 Add the first unused request, if any, compatible with A_2 to A_2.

   Solution: Sort requests by increasing finish time A1 = ∅ A2 = ∅ while there is any request (si , fi) compatible with either A1 or A2 do Add the first unused request, if any, compatible with A1 to A1. Add the first unused request, if any, compatible with A2 to A2. end while Answer. Greedy is not optimal in this case. Consider the following example: (1, 2),(3, 4),(1, 5) These jobs are sorted by increasing finish time. The optimum solutions sends (1, 2),(3, 4) the first machine and (1, 5) to the second machine so it schedules all jobs. The above Greedy algorithm first allocates (1, 2) to A1, then it allocates (3, 4) to A2. Then it cannot allocate (1, 5) to either of the machines.

7. Given sorted array of n distinct integers, arranged in increasing order $A[1, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give an algorithm that runs in time $O(\log n)$ for this problem. (15)

Solution:

HINT: Consider the algorithm that compares $A\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor$, and uses that comparison to recurse on either the first half or the second half of the array. Prove that if $A[\lfloor n/2 \rfloor] > \lfloor n/2 \rfloor$, such an i cannot be in last n–$\lfloor n/2 \rfloor$ coordinates, and if $A[\lfloor n/2 \rfloor] < \lfloor n/2 \rfloor$, then such an i cannot be in the first $\lfloor n/2 \rfloor$ coordinates.

CHECKIT(A)

```
1   k=1, j=n
2   while j-k>1
3       mid = ⌊j + k/2⌋
4       j
5       if A[]
6           Output
7       else
8           end
9           if A[k] = k then
10          Output k;
11          else if A[j] = j then Output j ;
12          else Output "No such index"
```

Question	Points	Score
1	12	
2	12	
3	10	
4	25	
5	25	
6	10	
7	15	
Total:	109	

Scratch Paper

Scratch Paper