CS325: Analysis of Algorithms, Fall 2016

Midterm Solutions

- *I don't know policy:* you may write "I don't know" *and nothing else* to answer a question and receive 25 percent of the total points for that problem whereas a *completely* wrong answer will receive zero.
- There are 6 problems in this exam.
- This formula may be useful:

$$1 + c + c^2 + \ldots + c^n = \sum_{i=1}^{n} c^i = \begin{cases} \Theta(1), & \text{if } c < 1 \\ \Theta(n), & \text{if } c = 1 \\ \Theta(c^n), & \text{if } c > 1 \end{cases}$$

Problem 1	
Problem 2	
Problem 3	
Problem 4	
Problem 5	
Problem 6	

Problem 1. **[5 pts]** Which of the following statements is true or false?

(a) If $f(n) = 123n - 21$ then $f(n) = \Theta(n)$. **T**

(b) If $f(n) = 34n^2 + n + 1$ then $f(n) = \Omega(n)$ **T**

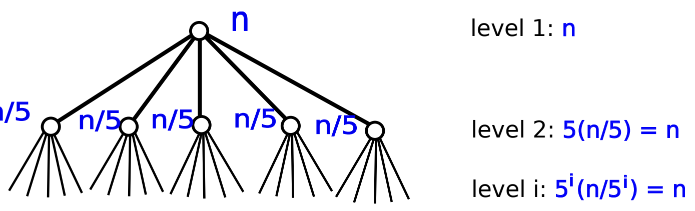
(c) If $f(n) = 34n^2 + n + 1$ then $f(n) = O(n \log n)$ **F**

(d) If $f(n) = 24n \log n + 12n$ then $f(n) = \Theta(n \log n)$. **T**

(e) If $f(n) = 2^{2+n}$ then $f(n) = \Theta(2^n)$. **T**

Problem 2. **[4pts]** The running time of an algorithm can be described by the equation $T(n) = 5T(n/5) + O(n)$. Use the recursion tree method to solve this recursion. Here is the recursion tree.

The blue numbers next to the nodes give the amount of non-recursive work at the nodes, from which we compute the amount of non-recursive work at each level.



The total running time can be computed by adding the work at different levels. Since the depth of the tree is $\log n$, the running time is:

$$O(\sum_{i=1}^{\log n} n) = O(n \log n).$$

Problem 3. A lazy programmer thinks that implementing the procedure Merge (of the merge sort) is too complicated. To simplify merge sort he eliminates the merge all together, and only recursively "sorts" the two halves of the array. Here is his pseudo code.

```
procedure LazySort($A[1 \cdots n]$)
    if $n > 1$ then
        $m \leftarrow \lfloor n/2 \rfloor$
        LazySort($A[1 \cdots m]$)
        LazySort($A[m+1 \cdots n]$)
```

Help the lazy programmer to understand his algorithm by answering the following questions.

(a) **[2 pts]** What does LazySort do?

It does not change anything in $A[1 \cdots n]$. It does nothing.

(b) **[1 pts]** What is the running time of LazySort?

If the running time is $T(n)$, we have:

$$T(n) = 2T(n/2) + O(1).$$

We can use the recursion tree method, or just expand to obtain:

$$T(n) = O(n).$$

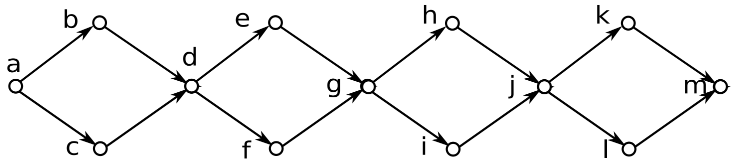
Problem 4. A topological sort of a directed acyclic graph $G = (V, E)$ is an ordering of its vertices such that for each directed edge $(u, v) \in E$ the vertex u appears before the vertex v.

(a) **[2 pts]** Give a topological sort of the following graph.

$$a, b, c, d, e, f, g, h, i, j, k, l, m.$$

(b) **[2 pts]** What is the number of different topological sorts of this graph?

Consider the topological sort algorithm that iteratively selects sources. It has one option in the beginning, the vertex a. It has two options in the next step, either b or c. In general, the algorithm gets to choose between two options four times: to choose from $\{b, c\}$, $\{e, f\}$, $\{h, i\}$, and $\{k, l\}$. Therefore, there are $2^4 = 16$ topological sorts.



Problem 5. Let $X_1[1 \cdots n], X_2[1 \cdots n], \ldots, X_k[1 \cdots n]$ be k *sorted* arrays. We would like to merge them into one sorted array, using the merge procedure that we saw in class. You can use MERGE($P[1 \cdots p], Q[1 \cdots q]$) as a black box. Provided *sorted* P and Q, MERGE merges them in $O(p+q)$ time, and it returns the merged list.

(a) **[2 pts]** What is the running time of the following simple algorithm to merge all $X_1[1 \cdots n], X_2[1 \cdots n], \ldots, X_k[1 \cdots n]$?

The call to MERGE takes $O(i \times n)$ time. Therefore, the total running time is:

$$O(n) + O(2n) + \ldots + O(kn) = n\,(O(1) + \ldots + O(k)) = O(nk^2).$$

```
1: procedure MergeArrays($X_1[1 \cdots n], X_2[1 \cdots n], \ldots, X_k[1 \cdots n]$)
2:     for $i \leftarrow 1$ to $k$ do
3:         $A[1 \cdots i \times n] \leftarrow$Merge($A[1 \cdots (i-1) \times n], X_i[1 \cdots n]$)
4:     return $A[1 \cdots k \times n]$
```

(b) **[4 pts]** Use divide and conquer to design a faster algorithm. (The running time of this faster algorithm must be $O(nk \log k)$.)

The idea is to split k arrays into two sets each composed of $k/2$ arrays (suppose k is even), recursively merge them, to obtain B and C, and use MERGE to merge B and C. Here is the pseudocode:

```
1: procedure DCMergeArrays($X_1[1 \cdots n], X_2[1 \cdots n], \ldots, X_k[1 \cdots n]$)
2:     if $k > 1$ then
3:         $m \leftarrow \lfloor k/2 \rfloor$
4:         $B[1 \cdots mn] \leftarrow$DCMergeArrays($X_1[1 \cdots n], X_2[1 \cdots n], \ldots, X_m[1 \cdots n]$)
5:         $C[1 \cdots (k-m)n] \leftarrow$DCMergeArrays($X_{m+1}[1 \cdots n], X_2[1 \cdots n], \ldots, X_k[1 \cdots n]$)
6:         return Merge($B[1 \cdots mn], C[1 \cdots (k-m)n]$)
```

(c) **[1 pts]** Analyze the running time of your algorithm.

Let $T(k)$ be the running time to merge k arrays, where each of them has n elements. We have the following recursion:

$$T(k) = 2T(k/2) + O(kn).$$

Again, we can use the recursion tree method to solve this recursion. Also, you can observe that n is independent of k, therefore, this is exactly similar to the recursion we saw for the running time of Merge sort, with an extra n at each level. So, we have:

$$T(k) = O(nk \log k).$$

Problem 6. A sequence of numbers $X[1 \cdots k]$ is exponentially-increasing if $X[i] > (2 \cdot X[i-1])$ for all $1 < i \leq k$. For example, $2, 5, 11, 30$ is exponentially increasing. In this problem we design an algorithm to find the largest *exponentially-increasing* subsequence of an input sequence $A[1 \cdots n]$. Here is an example of an exponentially-increasing subsequence.

$$\underbrace{1} \quad 10 \quad \underbrace{3} \quad 5 \quad \underbrace{11} \quad 7 \quad \underbrace{25} \quad \underbrace{53} \quad 12$$

(a) **[4 pts]** Let $S(i)$ be the length of the longest exponentially-increasing subsequence of $A[1 \cdots n]$ that ends with $A[i]$. Find a recursive relation to specify $S(i)$ in terms of $S(1), \ldots, S(i-1)$.

Let $s(i)$ be a subsequence of length $S(i)$ that ends with $A[i]$. There are two possibilities, (i) $S(i) = 1$, or (ii) the second last element in $s(i)$ is a $j < i$ such that $A[i] > 2A[j]$. We consider these possibilities to design our recursion.

$$S(i) = \max\left(1, \max_{j<i,\ A[i]>2A[j]} (S(j)+1)\right).$$

(b) **[3 pts]** Use your recursive relation to design a dynamic programming to compute the longest exponentially-increasing subsequence of $A[1 \cdots n]$. Give a psudo-code of your algorithm.

Let S be a table, we fill it iteratively similar to the maximum increasing subsequence problem. In the end, we return the maximum of all $S[i]$'s.

```
procedure MEIS($A[1 \cdots n]$)
    for $i \leftarrow 1$ to $n$ do
        $S[i] \leftarrow 1$
        for $j \leftarrow 1$ to $i-1$ do
            if $A[i] > 2A[j]$ and $S[i] < S[j]+1$ then
                $S[i] \leftarrow S[j]+1$
    return $\max_{0 \leq i \leq n} (S[i])$
```

(c) **[1 pts]** What is the running time of your dynamic programming?

The algorithm is composed of two nested loops with $O(1)$ work inside. The running time is

$$\sum_{i=1}^{n} O(i) = O(n^2).$$