CS3230: Design and Analysis of Algorithms
Semester 2, 2020-21, School of Computing, NUS

Midterm Test

Date: 6th March, 2021, Time Allowed: 2 hours

Instructions

- This paper consists of FIVE questions and comprises of TWELVE (12) printed pages, including this page.
- Answer ALL the questions.
- Write ALL your answers in this examination book.
- This is an OPEN BOOK examination.
- Write your Student Number (that starts with "A") on the top of every page.

Student Number:

Tutorial Group Number:

Question	Maximum	Score
Q1	5	
Q2	15	
Q3	25	
Q4	15	
Q5	20	
Total	**80**	

Question 1 [5 marks]: Rank the following functions in increasing order of growth; that is, the function $f(n)$ is before function $g(n)$ in your list, only if $f(n) = O(g(n))$.

- $n^{n+4} + n!$
- $n^{11\sqrt{n}}$
- $2^{4n \log n}$
- $4^{n^{1.1}}$
- $n^{11+\frac{\log n}{n}}$

To simplify notations, we write $f(n) < g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \approx g(n)$ to mean $f(n) = \Theta(g(n))$. E.g., the four functions n^2, n, $2021n^2 + n$ and n^3 could be written in increasing order of growth as follows: $n < n^2 \approx (2021n^2 + n) < n^3$.

Note, all the logarithms are of base 2. No need to write down the proofs for this problem.

Solution: $n^{11+\frac{\log n}{n}} < n^{11\sqrt{n}} < n^{n+4} + n! < 2^{4n \log n} < 4^{n^{1.1}}$

Question 2 [15 marks]: Solve the following recurrence relations.

(a) $T(n) = T(n/10) + \log n$.

(b) $T(n) = 27T(n/3) + n^3/\log^2 n$.

(c) $T(n) = 3T(n/5) + \log^2 n$.

(d) $T(n) = 2T(n/3) + n \log n$.

(e) $T(n) = T(n-2) + \log n$.

Among the above recurrences, for each recurrence solvable by master theorem, please indicate which case it belongs to (either case 1, 2, or 3) with proper reasoning and state the (time) complexity. For the recurrences that are not solvable by master theorem, solve them using any of the methods taught in the class. (Express your answers using $\Theta()$ notation.)

Solution:

(a) Case 2 of Master theorem. So $T(n) = \Theta(\log^2 n)$.

(b)

$$\begin{aligned}
T(n) &= 27T(n/3) + \frac{n^3}{\log^2 n} \\
\Rightarrow \frac{T(n)}{n^3} &= \frac{T(n/3)}{(n/3)^3} + \frac{1}{\log^2 n} \\
\frac{T(n/3)}{(n/3)^3} &= \frac{T(n/9)}{(n/9)^3} + \frac{1}{\log^2(n/3)} \\
\frac{T(n/9)}{(n/9)^3} &= \frac{T(n/27)}{(n/27)^3} + \frac{1}{\log^2(n/9)} \\
&\vdots \\
\frac{T(3)}{2^3} &= \frac{T(1)}{1^3} + \frac{1}{\log^2 3}
\end{aligned}$$

So, using the Telescoping method we get the following:
$T(n) = \frac{1}{(\log_2 3)^2}[\frac{n^3}{(\log_3 n)^2} + \frac{n^3}{(\log_3 n-1)^2} + \frac{n^3}{(\log_3 n-2)^2} + \cdots + \frac{n^3}{1^2}] = \frac{n^3}{(\log_2 3)^2} \sum_{x=1}^{\log_3 n} \frac{1}{x^2}$.
Observe,

$$\begin{aligned}
\sum_{x=1}^{\log_3 n} \frac{1}{x^2} &\leq 1 + \sum_{x=2}^{\log_3 n} \frac{1}{(x-1)x} \\
&= 1 + \sum_{x=1}^{\log_3 n - 1} \frac{1}{x(x+1)} \\
&= 1 + \sum_{x=1}^{\log_3 n - 1} (\frac{1}{x} - \frac{1}{x+1}) \\
&= 1 + (1 - \frac{1}{\log_3 n}) \qquad \text{using telescoping series} \\
&\leq 2
\end{aligned}$$

Furthermore, $\sum_{x=1}^{\log_3 n} \frac{1}{x^2} \geq 1$.
Therefore, $T(n) = \Theta(n^3)$.

(c) Note, $\log^2 n = O(n^{\log_5 3-\epsilon})$ for some $\epsilon > 0$. So this is Case 1 of Master theorem. Thus $T(n) = \Theta(n^{\log_5 3})$.

(d) Note, $n \log n = \Omega(n^{\log_3 2+\epsilon})$ for some $\epsilon > 0$ and $\frac{2}{3} n \log(n/3) \leq \frac{2}{3} n \log n$. So this is Case 3 of Master theorem. Thus $T(n) = \Theta(n \log n)$.

(e) We can use the recursion tree method to get that $T(n) = \Theta(n \log n)$.

Question 3 [7+8+10=25 marks]: Consider a *circular array*, where the first and the last cells are neighbors. More specifically, in a circular array $A[1, \cdots, n]$ of size n, $A[1]$ and $A[n]$ are neighbors of each other. An element in the array is called a *peak* if it is greater than or equal to its neighbors. Now we would like to find a peak in a circular array. (Just to clarify, the array we consider in this question is one dimensional and unsorted.)

(a) Suppose we know the values of $A[1], A[i], A[i+1]$ and $A[n]$ for some $1 < i < n$. Consider the maximum value among these four cells. Then depending on the four possible maximum, in which of the following two circular sub-arrays $A[1, \cdots, i]$ and $A[i+1, \cdots, n]$, are you guaranteed to find a peak? (Note, you have to give answers for the four possible cases. Provide your answer with proper explanation.)

Solution: We consider four different cases depending on which cell among $A[0]$, $A[k]$, $A[k+1]$ and $A[n-1]$ obtains the maximum value.

- $A[0] > A[k], A[0] > A[k+1], A[0] > A[n-1]$: A must have a peak in the range 0 to k. Thus a peak of A will be in the circular sub-array $A[0, ..., k]$.
- $A[k] > A[0], A[k] > A[k+1], A[k] > A[n-1]$: A must have a peak in the range 0 to k. Thus a peak of A will be in the circular sub-array $A[0, ..., k]$.
- $A[k+1] > A[0], A[k+1] > A[k], A[k+1] > A[n-1]$: A must have a peak in the range $k+1$ to $n-1$. Thus a peak of A will be in the circular sub-array $A[k+1, ..., n-1]$.
- $A[n-1] > A[0], A[n-1] > A[k], A[n-1] > A[k+1]$: A must have a peak in the range $k+1$ to $n-1$. Thus a peak of A will be in the circular sub-array $A[k+1, ..., n-1]$.

(b) Design a divide-and-conquer algorithm that finds a peak in a circular array of size n in time $O(\log n)$. (Provide the running time analysis of your algorithm.)

Solution: Our algorithm for finding peak in circular arrays is a divide-and-conquer approach which is very similar to the algorithm taught in class for finding peak in 1d-arrays. Using the result of previous part and setting $k = \lfloor n/2 \rfloor$, by looking at value of $A[0], A[k], A[k+1]$ and $A[n-1]$, we can find a circular sub-array of A (of size half of the size of A) that contains a peak of A. Moreover, the

stop rule for our recursion is the following: if the size of the circular array is less than 4, return the maximum of $A[0], A[1]$ and $A[2]$ (which is trivially a peak).
Let $T(n)$ denote the running time of our algorithm over a circular array of size n. Then, we have the following recursion relation for T:

$$
\begin{aligned}
T(n) &= 4c + T(n/2) \quad \%c \text{ is a constant} \\
T(n) &= 4c + 4c + T(n/4) \\
&\vdots \\
T(n) &= 4c + 4c + \cdots + 4c + T(n/2^i) \quad \% \text{ after } i \text{ iterations}
\end{aligned}
$$

Thus $T(n) = i \cdot 4c + T(n/2^i)$ where $2^i \leq n/3$. Setting $i = \log(n/3)$ and $T(3) = 4$, we have $T(n) = O(\log n)$.

(c) Prove a lower bound of $\Omega(\log n)$ on the running time of any comparison-based algorithm that finds a peak in a circular array of size n. (Assume, each comparison among two numbers x, y has two possible outcomes: Either $x \leq y$ or $x > y$.)

Solution: Proving the $\Omega(\lg n)$ lower bound for finding a peak in the array A requires two steps: (1) showing that there are n possible locations for the peak, and (2) constructing a binary decision tree with n leaves, and using it to prove the lower bound.

Following the hint, consider an array A that has only one peak. The location of this peak could be any index i between 0 and $n-1$, inclusive. We do not know which index i corresponds to a peak, but we do know that there is exactly one such index i for which $A[i-1] \leq A[i] \geq A[i+1]$. Therefore, there are n possible locations for the peak.

Construct a binary decision tree with n leaves that correspond to the possible locations for the peak in the array A, i.e. there is exactly one leaf for each possible location i between 0 and $n-1$, inclusive. At every internal node of the decison tree we are making a comparison, and narrowing down the set of possible locations of the peak in A. Note that the path from the root of the decision tree down to a leaf corresponds to the execution of a peak finding algorithm. Because the decision tree is binary, its height is $\Omega(\lg n)$. Therefore, *any* algorithm for peak finding over A takes $\Omega(\lg n)$-time. This proves the lower bound. □

Question 4 [15 marks]: Suppose you are given an array A of n pairs of positive integers (p_i, q_i), where $p_i < n$ and $q_i < n^3$ for all $i \in \{1, 2, \cdots, n\}$. Let us define *r-value* of a pair (p, q) as the real number $\sqrt{p} + q\sqrt{r}$. Unfortunately, you **cannot** (exactly) compute the r-value of an arbitrary pair (due to various reasons). Now, design an $O(n)$ time algorithm that sorts (in non-decreasing order) the pairs in A by their r-values for $r = n$.

Solution: Since $\sqrt{a_i} < \sqrt{n}$ for all $i \in \{1, \ldots, n\}$, then $\sqrt{a_i} + b_i\sqrt{n} < \sqrt{a_j} + b_j\sqrt{n}$ whenever $b_i < b_j$, independent of a_i and a_j. Thus, we can sort the pairs by their n-values by first sorting by a_i's using radix sort, and then sorting by b_i's, also with radix sort. Thus algorithm is runs in worst-case $O(n)$ time, since the largest number u is less than n^3, and is correct because the b_i's are more significant than the a_i's and counting sort is stable.

Question 5 [20 marks]: Consider the insertion sort algorithm (perhaps, one of the first sorting algorithms we learned). The algorithm proceeds through the array and puts each element into place one at a time by comparing it to all the preceding items in the array. The details are not important for this question. However, for your reference, let us provide the pseudocode of the insertion sort algorithm. (Note, the pseudocode considers 0 as the starting index of an array. So the input array is $A[0, \cdots, n-1]$. However, for this question, it is not important.)

```
InsertionSort(Array A, integer n)
    for i=1 to (n-1) do
        item = A[i];
        int slot = i;
        while (slot > 0) and (A[slot] > item) do
             A[slot] = A[slot-1];
             slot = slot-1;
        A[slot] = item;
}
```

Figure 1: Pseudocode of the insertion sort algorithm

The key property that you need to consider is that the running time of InsertionSort depends on the number of *inversions* in the permutation being sorted. Given a sequence of distinct integers $\{b_1, b_2, \cdots, b_n\}$, an *inversion* is a pair (b_i, b_j) such that $i < j$ but $b_i > b_j$. E.g., the sequence $\{2, 3, 8, 5, 4\}$ contains only three inversions $(8, 5), (8, 4)$ and $(5, 4)$.

Without loss of generality, assume that the input to the InsertionSort is an (arbitrary) permutation of $\{1, 2, \cdots, n\}$. The following result relates the running time of InsertionSort to the number of inversions in the input sequence: If a permutation S of $\{1, 2, \cdots, n\}$ contains k inversions, then InsertionSort(S, n) runs in time $\Theta(n + k)$. (No need to prove this result. You can assume it.)

Now, analyze the average-case performance of InsertionSort. More specifically, let S be a permutation of $\{1, 2, \cdots, n\}$ chosen uniformly at random from the set of all permutations of $\{1, 2, \cdots, n\}$. Show that the expected running time of InsertionSort on S in $\Theta(n^2)$.

Solution: Let $S = \{a_1, a_2, \ldots, a_n\}$ be the random input sequence. Fix two elements of the sequence i and j. Without loss of generality, assume $i < j$. Since the input sequence is a random permutation, $\Pr[a_i > a_j] = 1/2$. That is, elements i and j create an inversion with probability $1/2$.

For every pair (i, j) where $i < j$, let $X_{i,j}$ be the indicator random variable that equals 1 if $a_i > a_j$ and 0 otherwise. We know that $\Pr[X_{i,j}] = 1/2$, and hence $\mathrm{E}[X_{i,j}] = 1/2$.

Let X be the total number of inversions in the random input sequence. Notice that $X = \sum_{i<j} X_{i,j}$. Thus:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i,j} X_{i,j}\right] \\
&= \sum_{i<j} \mathrm{E}[X_{i,j}] \\
&= \sum_{i<j} \frac{1}{2} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{2} \\
&= \frac{n(n-1)}{4}
\end{aligned}$$

Thus, the expected number of inversions is $\Theta(n^2)$, and hence the expected running time of InsertionSort is $\Theta(n + n^2) = \Omega(n^2)$.