CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Practice Midterm Exam #1

Name: _____ *NetID:* _____

Instructions

1. Do not forget to write your name and NetID above.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points. You have 75 minutes to finish the exam. The exam is closed-book and closed notes.
3. **Note that problems appear on both odd and even numbered pages.** There is more than enough space to write down your solution for each problem below the problem itself. But if you ran out of space, you can also use the extra sheet at the end of the exam; if you do so, be clear about which problem you are solving.
4. Remember that you can leave a problem (or parts of it) entirely blank and receive 25% of the grade for that problem (or part). However, this should not discourage you from attempting a problem if you think you know how to approach it as you will receive partial credit more than 25% if you are on the right track. But keep in mind that if you simply do not know the answer, writing a very wrong answer may lead to 0% credit.

   The only **exception** to this rule is the extra credit problem: you do not get any credit for leaving the extra credit problem blank, and it is harder to get partial credit on that problem.
5. **You should always prove the correctness of your algorithm and analyze its runtime.** Also, as a general rule, avoid using complicated pseudo-code and instead explain your algorithm in English.
6. You may use any algorithm presented in the class as a building block for your solutions.

Suggestion: Leave the extra credit problem for last as it is harder than the rest and worths fewer points.

Problem. #	Points	Score
1	20	
2	25	
3	25	
4	30	
5	+10	
Total	100 + 10	

Problem 1.

(a) Determine the *strongest* asymptotic relation between the functions $f(n) = n^{\log\log n}$ and $g(n) = n^5$, i.e., whether $f(n) = o(g(n))$, $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, or $f(n) = \Theta(g(n))$. Prove the correctness of your choice. **(10 points)**

Solution. $n^{\log\log n} = \omega(n^5)$ or alternatively $n^5 = o(n^{\log\log n})$. To prove this, it suffices to show that $\lim_{n\to\infty} \frac{n^5}{n^{\log\log n}} = 0$.

We first have $n^5 = 2^{5\log n}$ and $n^{\log\log n} = 2^{\log n \cdot \log\log n}$, so we need to prove $\lim_{n\to\infty} \frac{2^{5\log n}}{2^{\log n\cdot\log\log n}} = 0$.

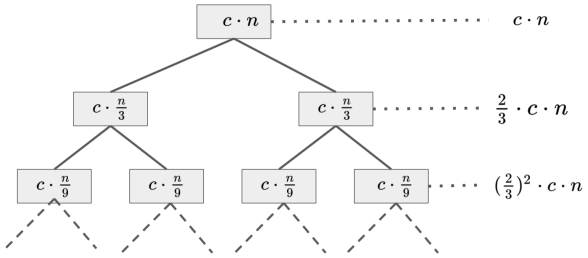
This follows from the fact that $\lim_{n\to\infty}(5\log n - \log n\cdot\log\log n) = \lim_{n\to\infty}\log n\cdot(5-\log\log n) = -\infty$, and thus $\lim_{n\to\infty} \frac{2^{5\log n}}{2^{\log n\cdot\log\log n}} = 0$, concluding the proof.

(b) Use the *recursion tree* method to solve the following recurrence $T(n)$ by finding the *tightest* function $f(n)$ such that $T(n) = O(f(n))$. **(10 points)**

$$T(n) \leq 2 \cdot T(n/3) + O(n)$$

Solution. The correct answer is $T(n) = O(n)$. We prove this as follows.

We first replace $O(n)$ with $c \cdot n$ and then draw the following recursion tree:



At level i of the tree, we have 2^i nodes each with a value of $c \cdot \frac{n}{3^i}$, hence the total value of level i is $(\frac{2}{3})^i \cdot c \cdot n$. The total number of levels in this tree is also $\lceil \log_3 n \rceil$. As such, the total value is:

$$\sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{2}{3})^i \cdot c \cdot n = c \cdot n \sum_{i=0}^{\lceil \log_3 n \rceil} (\frac{2}{3})^i \leq c \cdot n \sum_{i=0}^{\infty} (\frac{2}{3})^i = c \cdot n \cdot \frac{1}{1 - 2/3} = 3 \cdot c \cdot n.$$

(the second last equality is by using the formula for sum of geometric series)

Hence, $T(n) = O(n)$, finalizing the proof.

Problem 2. Consider the algorithm below for finding the smallest two numbers in an array A of size $n \geq 2$.

`FIND-SMALLEST-TWO`$(A[1:n])$:

1. If $n = 2$: return $(A[1], A[2])$.
2. Otherwise, let $(m_1, m_2) \leftarrow$ `FIND-SMALLEST-TWO`$(A[1 : n-1])$.
3. If $A[n]$ is smaller than at least one of m_1, m_2, return $A[n]$ and $\min\{m_1, m_2\}$; otherwise, return (m_1, m_2).

We analyze `FIND-SMALLEST-TWO` in this question.

(a) Use *induction* to prove the correctness of this algorithm. **(15 points)**

Solution.

Induction hypothesis: For any integer $n \geq 2$ and array A of size n, `FIND-SMALLEST-TWO`$(A[1 : n])$ returns the smallest two numbers in the array A, or in other words, returns the correct answer.

Induction base: For $n = 2$, the two smallest numbers in any array A of size 2 are $A[1], A[2]$ (as there is no other element!), thus `FIND-SMALLEST-TWO` returns the correct answer in the base case.

Induction step: Suppose the induction hypothesis is true for all $n \leq i$ for some integer $i \geq 2$ and we prove it for $n = i + 1$.

The algorithm first runs `FIND-SMALLEST-TWO`$(A[1 : n-1])$ which is on an array of size $i \geq 2$. Hence, we can apply the induction hypothesis and have that (m_1, m_2) are indeed the smallest two numbers in $A[1 : n-1] = A[1 : i]$.

We now prove that the choice taken by the algorithm next is the correct one:

(a) If $A[n]$ is smaller than at least one of m_1, m_2, then since m_1, m_2 are the smallest two numbers in $A[1 : n-1]$, $A[n]$ would be one of the smallest numbers in $A[1 : n]$; moreover, the other number clearly can only be $\min\{m_1, m_2\}$. Hence, the returned solution by the algorithm is correct here, proving the induction hypothesis in this case.

(b) If $A[n]$ is larger than both m_1, m_2, then since m_1, m_2 are the smallest two numbers in $A[1 : n-1]$, they will also be the smallest two numbers in $A[1 : n]$ and thus returning them, as done by the algorithm, is the correct choice. Hence the induction hypothesis holds in this case as well.

This concludes the proof of induction hypothesis.

The correctness of the algorithm now follows directly from the induction hypothesis.

(b) Write a recurrence for this algorithm and solve it to obtain a tight upper bound on the worst case runtime of this algorithm. You can use any method you like for solving this recurrence. **(10 points)**

Solution. The algorithm recursively calls itself on an array of length $n-1$ and then spends $O(1)$ additional time to output the solution. Hence, if we define $T(n)$ to be the worst case runtime of the algorithm on any array of length n, we have $T(n) \leq T(n-1) + O(1)$.

To solve this recurrence, we first change $O(1)$ with constant c and have $T(n) \leq T(n-1) + c$. We then solve this recurrence by substitution:

$$T(n) \leq T(n-1) + c \leq T(n-2) + c + c \leq T(n-3) + c + c + c \leq \cdots \leq \underbrace{c + c + \cdots + c}_{n} = c \cdot n.$$

Hence, $T(n) = O(n)$ and thus the runtime of this algorithm is $O(n)$.

Problem 3. You are given an *unsorted* array $A[1:n]$ of n real numbers with possible repetitions. Design an algorithm that in $O(n \log n)$ time finds the largest number of time any entry is repeated in the array A.

Example. For $A = [1, 7.5, 2, 5.5, 7.5, 7.5, 1, 5, 9.5, 1]$ the correct answer is 3 (1 and 7.5 are repeated 3 times).

(a) *Algorithm:* **(10 points)**

Solution. We first use merge sort to sort the array A. We then iterate over the elements of the array one by one and maintain a counter COUNTER and a variable MAX initialized to be one. In each iteration $i > 1$, if $A[i] \neq A[i-1]$, we reset COUNTER to one; otherwise if $A[i] = A[i-1]$, we increase COUNTER by one and let MAX $\leftarrow \max\{$COUNTER,MAX$\}$. At the end, we return MAX.

(b) *Proof of Correctness:* **(10 points)**

Solution. After sorting A, all copies of the same number would appear right after each other. Hence, to count how many times the number $A[i]$ appears in the array A, we only need to consider the entries in the immediate neighborhood of A as is done in the algorithm. More precisely, the value of COUNTER right before it gets reset since $A[i] \neq A[i-1]$ is equal to the number of times $A[i-1]$ is repeated in A. Since MAX is simply maintaining the maximum value of COUNTER throughout the algorithm, we obtain that the final answer is correct.

(c) *Runtime Analysis:* **(5 points)**

Solution. The algorithm involves running merge sort in $O(n \log n)$ time and then a single n iteration for-loop with each iteration taking $O(1)$ time, hence $O(n)$ in total for the for-loop. As such, the total runtime is $O(n \log n)$.

Problem 4. You are given a list of n items with positive values $v_1, \ldots, v_n$. The goal is to pick as many items as possible to maximize the total value of the items taken. However, you are not allowed to pick any two (or more) consecutive items. Design an $O(n)$ time dynamic programming algorithm to compute the maximum obtainable value under this constraint.

(a) *Specification of recursive formula for the problem (in plain English)*: **(7.5 points)**

Solution. For any integer $0 \leq i \leq n$, we define:

- $V(i)$: the maximum value we can achieve by picking a subset of items from $\{1, \ldots, i\}$ that satisfies the constraint of not having two (or more) consecutive items.

The solution to the problem can be obtained by returning $V(n)$.

(b) *Recursive solution for the formula and its proof of correctness:* **(10 points)**

Solution. The recursive formula for $V(i)$ is as follows:

$$V(i) = \begin{cases} 0 & \text{if } i = 0 \\ v_1 & \text{if } i = 1 \\ \max\{V(i-1), V(i-2) + v_i\} & \text{if } i \geq 2 \end{cases}$$

Proof of correctness: We consider each case separately:

- If $i = 0$, there is no item we can pick and thus $V(0) = 0$ is the correct value.
- If $i = 1$, since all items, and in particular item $A[1]$ has a positive value, we can and should pick $A[1]$ which makes $V(1) = v_1$.
- Otherwise, we have two options: (1) we do not pick item i – in this case, we will not gain any value from i but we can pick any item from $\{1, \ldots, i-1\}$ including the item $i-1$ because now that i is not part of the solution, picking $i-1$ has no problem; (2) we pick item i – in this case, we will gain the value of v_i but no longer can pick item $i-1$ as otherwise we will have two consecutive items in the solution which is not possible; thus, in this case, we should skip over item $i-1$ and pick the best solution from $\{1, \ldots, i-2\}$ captured by $V(i-2)$ (since $i \geq 2$, $i-2 \geq 0$ and thus $V(i-2)$ is well defined).

  Consequently, in option (1) we get the value of $V(i-1)$ (just from the items we pick from $\{1, \ldots, i-1\}$) and in option (2) we get the value of $V(i-2) + v_i$ (from the items we pick from $\{1, \ldots, i-2\}$ and the item i that we picked). Since our goal is to find the maximum value possible, taking the max of the two options, as done by the formula, ensures the correctness.

(c) *Algorithm (either memoization or bottom-up dynamic programming):* **(5 points)**

Solution. We design a memoization algorithm for the problem.

Let $T[0:n]$ be an $n+1$ dimensional array originally filled with 'undefined'. We write the following recursive function:

$\texttt{memV}(i)$:

(1) If $T[i] \neq$ 'undefined' return $T[i]$.
(2) If $i = 0$, let $T[i] = 0$;
(3) Else if $i = 1$, let $T[i] = v_i$;
(4) Else let $T[i] = \max\{\texttt{memV}(i-1), \texttt{memV}(i-2) + v_i\}$.
(5) Return $T[i]$.

The answer to the problem can then be computed from $\texttt{memV}(n)$.

(d) *Runtime Analysis:* **(7.5 points)**

Solution. There are $n + 1 = O(n)$ subproblems for the recursive formula $V(\cdot)$. Moreover, each subproblem takes $O(1)$ to compute, hence the total runtime of the algorithm is $O(n)$ as desired.

Problem 5. [**Extra credit**] You are given an *unsorted* array $A[1 : n]$ of n distinct numbers with the promise that each element is at most k positions away from its correct position in the sorted order. Design an algorithm that sort the array A in $O(n \log k)$ time. **(+10 points)**

Solution. A complete solution consists of three parts, algorithm, proof of correctness, and runtime analysis.

Algorithm: The algorithm is as follows: we use merge sort to sort $A[1 : 2k]$, then sort $A[k + 1 : 3k]$, $A[2k + 1 : 4k]$, and so on and so forth until we sort $A[n - 2k + 1 : n]$ (note that there are overlaps of size k elements between the consecutive arrays).

Proof of Correctness: Let $1 \leq i \leq \frac{n}{2k} - 1$ be an index that goes over the arrays $A[1 : 2k], A[k + 1 : 3k], \ldots$ that we sort. I.e., $i = 1$ refers to the array $A[1 : 2k]$, $i = 2$ refers to $A[k + 1 : 3k]$, and similarly for the rest.

We prove the correctness by induction over i: our induction hypothesis is that after we are done sorting the i-th array, all the numbers in arrays $1, \ldots, i - 1$ are in their correct (sorted) position in the array A.

The base case of the induction corresponds to sorting array $A[1 : 2k]$. Since we are promised that every element of A is at most k position away from its sorted order, the smallest k numbers in A all belong to $A[1 : 2k]$; hence, after sorting this array, the first k positions of A will contain these numbers proving the induction hypothesis in this case.

Now suppose the induction hypothesis is true for all integers up to i and we prove it for $i+1$. When sorting the $(i + 1)$-th array, by induction hypothesis for i, we have that all numbers in arrays 1 to $i - 1$ are in their correct position already. This means that only the elements in array i minus array $i - 1$ may not be in their correct position. These numbers however belong to the array $i + 1$. Moreover, since each number is at most k indices away from its correct position, after sorting the array $i + 1$, all numbers that are now in array i must be in their correct position (otherwise, they have been more than k indices away from their correct position before sorting). This proves the induction step and concludes the proof of induction hypothesis.

To finalize the proof, notice that by the induction hypothesis for the largest value of i, we obtain that all numbers except maybe for the ones in $A[n - k + 1 : n]$ are in their correct position. However, since we are also sorting $A[n - 2k + 1 : n]$ at the end and these numbers belong to this array (both now and also in their correct position), these numbers will also be placed in their correct position at the end of this step. This concludes the proof.

Runtime analysis: We are performing $O(n/k)$ different merge sorts, each on an array of size $O(k)$ which takes $O(k \log k)$ time per array and $O(n \log k)$ time in total.

Extra Workspace