CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Practice Midterm Exam #1 Solutions

Name: ______________________________ *NetID:* ______________

Instructions

1. Do not forget to write your name and NetID above, and to sign Rutgers honor pledge below.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points.
3. The exam should be done **individually** and you are not allowed to discuss these questions with anyone else. This includes asking any questions or clarifications regarding the exam from other students or posting them publicly on Piazza (**any inquiry should be posted privately on Piazza**). You may however consult all the materials used in this course (video lectures, notes, textbook, etc.) while writing your solution, but **no other resources are allowed**.
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 there is almost no 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 or homeworks as a building block for your solutions.

Rutgers honor pledge:

On my honor, I have neither received nor given any unauthorized assistance on this examination.

Signature:______________________________

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

Problem 1.

(a) Determine the *strongest* asymptotic relation between the functions

$$f(n) = \sqrt{\log n} \quad \text{and} \quad g(n) = \frac{n}{2^{(\log \log n)}},$$

i.e., whether $f(n) = o(g(n))$, $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, or $f(n) = \Theta(g(n))$. Remember to prove the correctness of your choice. **(15 points)**

Solution. The correct answer is $f(n) = o(g(n))$. We now prove this as follows.

Firstly, $\sqrt{\log n} \leq \log n$ so we can instead prove $\log n = o(g(n))$ by transitivity.

Secondly, $2^{\log \log n} = \log n$ (you can do a change of variable $m = \log n$ to see this easily).

So our goal is to prove that $\log n = o(\frac{n}{\log n})$, which require us to show that $\lim_{n\to+\infty} \frac{\log n}{\frac{n}{\log n}} = 0$. We have,

$$\lim_{n\to+\infty} \frac{\log n}{\frac{n}{\log n}} = \lim_{n\to+\infty} \frac{\log^2 n}{n} = 0,$$

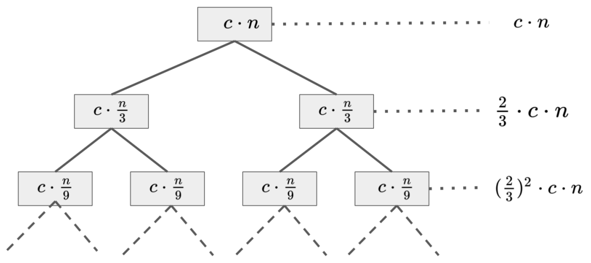
where the last equation is because $\log^2(n) = o(n)$ (generally, $\log^c(n) = o(n)$ for all constants $c > 0$).

(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:



The root is at level 1. At level i of the tree, we have 2^{i-1} nodes each with a value of $c \cdot \frac{n}{3^{i-1}}$, hence the total value of level i is $(\frac{2}{3})^{i-1} \cdot c \cdot n$. 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. Return the two smallest numbers among $m_1, m_2$, and $A[n]$.
```

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: Considering (m_1, m_2) are the two smallest numbers among $A[1:n-1]$, the smallest two numbers of the array $A[1:n]$ overall would definitely be a subset of $\{m_1, m_2, A[n]\}$ (if there is a smaller number in $A[1:n-1]$, that number should be chosen instead of m_1 or m_2). So, by returning the smaller two numbers among this set, as done by the algorithm, we also obtain the two smallest numbers in $A[1:n]$. 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.

Remember to *separately* write your algorithm (**10 points**), the proof of correctness (**10 points**), and runtime analysis (**5 points**).

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).

Solution. *Algorithm:* The algorithm is as follows:

1. Use merge sort to sort the array A.
2. Let COUNTER = 1 and MAX = 1.
3. For $i = 2$ to n:
   (a) if $A[i] \neq A[i-1]$, set COUNTER = 1;
   (b) otherwise let COUNTER = COUNTER + 1 and MAX = the larger of MAX and COUNTER.
4. Return MAX.

Proof of correctness: 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 due to the condition $A[i] \neq A[i-1]$ 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.

Runtime analysis: The algorithm involves running merge sort in $O(n \log n)$ time and then a single $n-1$ 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 set of n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$. You are also given *two* knapsacks with sizes W_1, W_2. The goal is to pick a subset of items with maximum value such that we can place each of the chosen items in one of the two knapsacks with the restriction that the total weight of items in each knapsack should be smaller than its size. Design an $O(n \cdot W_1 \cdot W_2)$ time dynamic programming algorithm that outputs the maximum value we can obtain by picking a subset of items that fits the two knapsacks.

Remember to *separately* write your specification of recursive formula in plain English (**7.5 points**), your recursive solution for the formula and its proof of correctness (**10 points**), the algorithm using either memoization or bottom-up dynamic programming (**7.5 points**), and runtime analysis (**5 points**).

Example: For items with weights $[1, 4, 2, 5, 1]$ and values $[2, 3, 1, 10, 1]$, and knapsacks of size 4 and 3, the correct output is 6: this is achieved by picking items 1 and 3 in knapsack two and picking item 2 in knapsack one – note that we cannot fit item 4 in either knapsack as its weight is larger than the size of each one.

Solution.

Specification: For every $0 \leq i \leq n$, $0 \leq j \leq W_1$, and $0 \leq k \leq W_2$, we define:

- $K(i, j, k)$: the maximum value we can obtain by picking a subset of items $\{1, \ldots, i\}$ that fits a knapsack of size j and another knapsack of size k.

The solution to our original problem is then $K(n, W_1, W_2)$.

Recursive formula: We design the following recursive formula:

$$K(i,j,k) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = k = 0 \\ K(i-1,j,k) & \text{if } w_i > j \text{ and } w_i > k \\ \max\{K(i-1,j,k), K(i-1,j-w_i,k) + v_i\} & \text{if } w_i \leq j \text{ but } w_i > k \\ \max\{K(i-1,j,k), K(i-1,j,k-w_i) + v_i\} & \text{if } w_i > j \text{ but } w_i \leq k \\ \max\{K(i-1,j,k), K(i-1,j-w_i,k) + v_i, K(i-1,j,k-w_i) + v_i\} & \text{otherwise} \end{cases}$$

We now prove the correctness of this algorithm:

- If $i = 0$ there is no item to pick and if both $j = k = 0$ then there is no way we can fit any item in either of the knapsacks; hence, $K(i, j, k) = 0$ here is the correct value.
- If both $w_i > j$ and $w_i > k$, item i will not fit either knapsacks; so the only thing we can do is to skip this item altogether and from the items $\{1, \ldots, i-1\}$, pick the *best* solution (corresponding to $K(i-1, j, k)$ by definition) so that we maximize the value for items $\{1, \ldots, i\}$ as well. Hence, $K(i, j, k) = K(i-1, j, k)$ in this case is the correct value.
- If $w_i \leq j$ but $w_i > k$, we only have two options: (1) we still ignore picking item i and hence collect the total value $K(i-1, j, k)$ (as described in the previous part), or (2) we pick item i in the first knapsack (thus collect value v_i), reduce the size of knapsack one to $j - w_i$ (as now item i with weight w_i is in the knapsack), and then pick the best solution from the remaining items (corresponding to $K(i-1, j-w_i, k)$); so with this option, we collect $K(i-1, j-w_i, k)+v_i$ value. As our goal is to maximize the value obtained, the correct choice in this case is $\max\{K(i-1, j, k), K(i-1, j-w_i, k) + v_i\}$ (note that we cannot fit item i in the second knapsack in this case).
- If $w_i > j$ but $w_i \leq k$, we can do exactly as in the previous case with the difference that we now need to pick item i in the knapsack two instead.
- If $w_i \leq j$ and $w_i \leq k$, we can either skip picking i and get $K(i-1, j, k)$ value, pick i in the first knapsack and (by the argument above) get $K(i-1, j-w_i, k) + v_i$, or pick i in the second knapsack and (by the argument above) get $K(i-1, j, k-w_i) + v_i$; thus returning the maximum of these three terms gives the correct answer.

Algorithm: We give a bottom-up dynamic programming algorithm for the problem (you could also do memoization and that is absolutely fine). We have to pick an evaluation order: since each $K(i,j,k)$ is only updated from $K(i',j',k')$ where $i' < i$, $j' \leq j$, and $k' \leq k$, we only need to make sure we iterate over all i,j,k in increasing order of their values. The algorithms is as follows:

```
(1) Let $T[0:n][0:W_1][0:W_2]$ be a 3D array of size $(n+1) \times (W_1+1) \times (W_2+1)$ initialized with 0's (this
    takes care of the base case automatically).
(2) For $i = 1$ to $n$:
      For $j = 1$ to $W_1$:
        For $k = 1$ to $W_2$:
          If $w_i > j$ and $w_i > k$, then $T[i][j][k] = T[i-1][j][k]$;
          Else if $w_i \leq j$ but $w_i > k$, then $T[i][j][k] = \max\{T[i-1][j][k], T[i-1][j-w_i][k] + v_i\}$;
          Else if $w_i > j$ but $w_i \leq k$, then $T[i][j][k] = \max\{T[i-1][j][k], T[i-1][j][k-w_i] + v_i\}$;
          Else $T[i][j][k] = \max\{T[i-1][j][k], T[i-1][j-w_i][k] + v_i, T[i-1][j][k-w_i] + v_i\}$.
(3) Return $T[n][W_1][W_2]$.
```

Runtime analysis: The first line of the algorithm above takes $O(n \cdot W_1 \cdot W_2)$ to initialize the array T. Moreover, we do a for-loop of length n, inside it another for-loop of length W_1, and inside that another for-loop of length W_2, where each iteration of the inner most loop takes $O(1)$ time; thus the total runtime is $O(n \cdot W_1 \cdot W_2)$.

(For a memoization algorithm, the runtime analysis spells out that there are $O(n \cdot W_1 \cdot W_2)$ subproblems $K(i,j,k)$ for all valid choices of i,j,k, and each one takes $O(1)$ time to compute, hence the total runtime again would be $O(n \cdot W_1 \cdot W_2)$.)

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. *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