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

Midterm Exam #1 Solutions

October 11, 2019

Problem 1.

(a) Determine the *strongest* asymptotic relation between the functions $f(n) = 2^{\sqrt{\log n}}$ and $g(n) = n$, i.e., whether $f(n) = o(g(n))$, $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, or $f(n) = \Theta(g(n))$. Prove the correctness of your choice. **(10 points)**

Solution. We prove $2^{\sqrt{\log n}} = o(n)$ by showing that $\lim_{n\to\infty} \frac{2^{\sqrt{\log n}}}{2^{\log n}} = 0$.

This is true because $n = 2^{\log n}$ and $\lim_{n\to\infty} \sqrt{\log n} - \log n = -\infty$ which implies $\lim_{n\to\infty} \frac{2^{\sqrt{\log n}}}{2^{\log n}} = 0$.

(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)**

$$T(n) \leq 5 \cdot T(n/5) + O(n)$$

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

We skip drawing the recursion tree and simply explain how the tree looks: at the root we have the node with value $c \cdot n$, which has 5 child-nodes, each with the value $c \cdot n/5$, and so on: at each level i (assuming root is at level 0), we have 5^i nodes, each with value $c \cdot n/5^i$. So the total value at level i is $5^i \cdot c \cdot n/5^i = c \cdot n$. Moreover, we have at most $\lceil \log_5 (n) \rceil$ levels in this tree before $n/5^i$ becomes 1 and hence we reach a leaf-node. So the total value of all the nodes in the tree is $c \cdot n \cdot \lceil \log_5 (n) \rceil = O(n \log n)$.

Problem 2. Consider the following algorithm for finding the maximum number in an array A of length n:

`MAX-ALG`(A):

1. If $n = 1$, return $A[1]$.
2. Otherwise, let $m_1 \leftarrow$ `MAX-ALG`$(A[1 : \lfloor n/2 \rfloor])$ and $m_2 \leftarrow$ `MAX-ALG`$(A[\lfloor n/2 \rfloor + 1 : n])$.
3. Return the maximum between m_1 and m_2.

We analyze `MAX-ALG` in this question.

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

Solution.

Induction hypothesis: For any integer $n \geq 1$ and array A of size n, `MAX-ALG`$(A[1 : n])$ returns the maximum of the array A, or in other words, returns the correct answer.

Induction base: For $n = 1$, there is only one element in A and thus returning $A[1]$ as done by the algorithm is the correct answer in the base case.

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

The algorithm first runs `MAX-ALG`$(A[1 : \lfloor n/2 \rfloor])$ and $m_2 \leftarrow$ `MAX-ALG`$(A[\lfloor n/2 \rfloor + 1 : n])$: since size of each of these arrays is strictly smaller than n (and hence are at most equal to i), by the induction hypothesis, m_1 would be equal to the maximum of $A[1 : \lfloor n/2 \rfloor]$ and m_2 would be equal to the maximum of $A[\lfloor n/2 \rfloor + 1 : n]$.

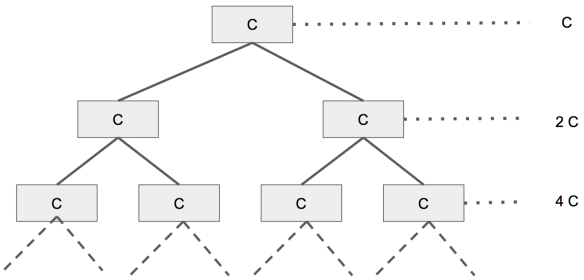
Since maximum of A either belongs to $A[1 : \lfloor n/2 \rfloor]$ (in which case it would be m_1), or belongs to $A[\lfloor n/2 \rfloor + 1 : n]$ (in which case it would be m_2), returning maximum of m_1 and m_2, as done by the algorithm, ensures that the returned answer is maximum of $A[1 : n]$, proving the induction step. This concludes the proof of the 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 two arrays of length at most $\lceil n/2 \rceil$ 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 $T(n) \leq 2 \cdot T(n/2) + O(1)$ (recall that we ignore floors and ceiling when writing the recurrences).

To solve this recurrence, we first change $O(1)$ with constant c and have $T(n) \leq 2 \cdot T(n/2) + c$. We then solve this recurrence by the following recursion tree:



In particular, at level i of this tree, we have 2^i nodes, each with the value c, making the total value of the level $c \cdot 2^i$. The total number of levels in the tree is also at most $\lceil \log(n) \rceil$. So the total value of the tree is

$$T(n) \leq \sum_{i=1}^{\log n} c \cdot 2^i = c \cdot 2^{\log n + 1} = O(n).$$

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 positive integers. Design an $O(n)$ time algorithm that finds the *smallest positive integer* that does *not* appear in this array.

Example. For the input array $A = [5, 2, 3, 1, 7]$, the correct answer is 4.

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

Solution. We use an algorithm in the spirit of the counting sort:

(1) Create an array B of size $n + 1$ initialized to 0.

(2) For $i = 1$ to n: if $A[i] \leq n + 1$, increase $B[A[i]]$ by one.

(3) Iterate over B and find the smallest index j where $B[j] = 0$; return j as the answer.

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

Solution. The smallest missing positive integer is at most $n + 1$ since there are at most n distinct integers in A. Moreover, after the second step, for any integer $1 \leq j \leq n+1$ that belongs to A, $B[j] > 0$. Hence, the least integer for which $B[j] = 0$ is the smallest positive integer which is missing from A as all the integers $1, \ldots, (j-1)$ must be present in A (otherwise, some entry in $B[1], \ldots, B[j-1]$ must have remained 0). As the algorithm outputs this integer j, we obtain the correctness of the algorithm.

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

Solution. The first line of the algorithm takes $O(n)$ time as we are initializing $n + 1$ numbers to 0. The second line involves a for-loop over the n elements of A and hence takes $O(n)$ time. The last line requires going over the $n + 1$ entries of B which again can be done in $O(n)$ time. Hence, the total runtime is $O(n)$.

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.

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.

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

Solution. 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 $K(n, W_1, W_2)$.

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

Solution. 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.

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

Solution. 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 $T[n][W_1][W_2]$.

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

Solution. 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 spell 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)
Consider the following algorithm for finding the minimum of a list of n *distinct* numbers:

`MIN-RAND-ALG`(L):

1. While size of L is larger than one:
   (a) Pick one of the entries of the list, denoted by p, uniformly at random.
   (b) Iterate over the list and delete each element which is *larger* than p.
2. Return the only element in L.

Prove that **(a)** this algorithm correctly finds the minimum element and that **(b)** its expected worst case runtime is $O(n)$. You may assume that picking a random entry from the list and deleting each single element of the list can be done in $O(1)$ time. **(+10 points)**

Solution.

(a) The algorithm *upon termination* can only output the minimum element: this is because the numbers in the list are distinct and we may only remove an item from the list if it is larger than at least one other number (namely p); this means that the minimum element is *never* removed from the list; thus upon termination, namely when there is only one element in the list left, the remaining element can only be the minimum element. This concludes the proof of correctness.

(b) Let us define $T(n)$ as the *expected* worst case running time of this algorithm on a list of length n: recall that expected worst case runtime means that we pick the worst case list but then analyze the *expected* runtime of the algorithm on this list, not the worst combination of random bits.

We argue that $T(n)$ follows the following recurrence:

$$T(n) \leq \frac{1}{2} \cdot T(n/2) + \frac{1}{2} \cdot T(n) + O(n).$$

This is because, when we have a list of length n, we pick an element p that is smaller than at least half the elements in the list with probability half; conditioned on this event, the runtime would be at most $T(n/2)$ (to solve the problem on a list of size *at most* $n/2$ but potentially much smaller also). Otherwise, which happens with probability half also, the runtime would be at most $T(n)$ (even in the worst case scenario when we pick the maximum of the list, we will never increase the size of the list for sure!). Finally, at each iteration, we are spending $O(n)$ time to iterate over the list. This proves the correctness of the recurrence.

Finally, we can simplify this recurrence by taking the $T(n)$ term in the right hand side to the left, replace $O(n)$ with $c \cdot n$, and have $\frac{1}{2}T(n) \leq \frac{1}{2}T(n/2) + c \cdot n$; by multiplying both sides with 2, we also get, $T(n) \leq T(n/2) + 2c \cdot n$. This recurrence can be solved by substitution (or by recursion trees) to get:

$$T(n) \leq T(n/2) + 2c \cdot n \leq T(n/4) + c \cdot n + 2c \cdot n \leq T(n/8) + \frac{1}{2} \cdot c \cdot n + c \cdot n + 2c \cdot n = c \cdot n \cdot \sum_{i=1}^{\log n} \frac{1}{2^i} = O(n),$$

as this final series obtained is converging to a constant. This shows the expected worst case runtime is $O(n)$.
