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

Midterm Exam #1 Solution

March 06, 2022

Problem 1.

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

$$f(n) = 2^{\sqrt{n}} \quad \text{and} \quad g(n) = n^4,$$

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)**

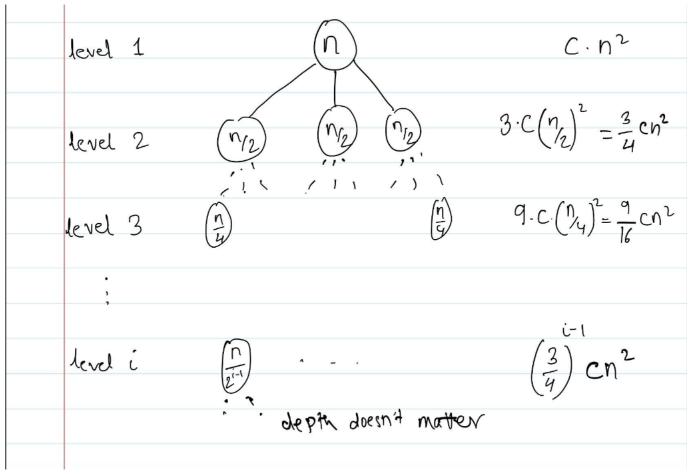
Solution. The correct answer is $f(n) = \omega(g(n))$ or alternatively $g(n) = o(f(n))$.

To prove this, we do a change of variable $m = \sqrt{n}$. This way, $2^{\sqrt{n}} = 2^m$ and $n^4 = m^8$. We know that $m^8 = o(2^m)$ (say, as in Homework 1), and thus $n^4 = o(2^{\sqrt{n}})$ also, which means $g(n) = o(f(n))$ which is also the same as $f(n) = \omega(g(n))$.

(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)**

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

Solution. We prove that $T(n) = O(n^2)$. The recursion tree is as follows because at every step we have 3 subproblems, each of size $n/2$, and a subproblem of size m has value $C \cdot m^2$ in the tree.



level 1
level 2
level 3
level i
n
$n/2$
$n/4$
$\frac{n}{2^{i-1}}$
$C \cdot n^2$
$3 \cdot C\left(\frac{n}{2}\right)^2 = \frac{3}{4} cn^2$
$9 \cdot C\left(\frac{n}{4}\right)^2 = \frac{9}{16} cn^2$
$\left(\frac{3}{4}\right)^{i-1} cn^2$
depth doesn't matter

Thus, the total value of the tree is

$$\sum_{i=1}^{\text{depth}} (3/4)^{i-1} \cdot C \cdot n^2 \leq C \cdot n^2 \cdot \sum_{i=0}^{\infty} (3/4)^i = C \cdot n^2 \cdot \frac{1}{1-(3/4)} = 4C \cdot n^2 = O(n^2).$$

Thus, $T(n) = O(n^2)$.

Problem 2. Consider the algorithm below for finding the sum of numbers in an array A of size $n \geq 1$.

```
TOTAL-SUM($A[1:n]$):
  1. If $n = 1$, return $A[1]$.
  2. Otherwise, return TOTAL-SUM($A[1:n-1]$) + $A[n]$.
```

We analyze `TOTAL-SUM` in this question.

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

Solution. Our induction hypothesis is that for any $n \geq 1$, `TOTAL-SUM`$(A[1:n])$ on any array A returns the sum of numbers in the array A correctly.

Base case. For $n = 1$, `TOTAL-SUM`$(A[1:1])$ simply returns $A[1]$ which is also the sum of numbers in $A[1:1]$ as there is only one number $A[1]$ in A.

Induction step. Suppose the induction hypothesis is true for all $n \leq k$ and we prove it for $n = k+1$. In `TOTAL-SUM`$(A[1:k+1])$, the algorithm returns `TOTAL-SUM`$(A[1:k]) + A[k+1]$ as $n = k+1$. By induction hypothesis, as length of the array $A[1:k]$ is less than $k+1$, we know that the algorithm correctly returns $\sum_{i=1}^{k} A[i]$. By adding this number with $A[k+1]$, we get $\sum_{i=1}^{k+1} A[i]$, which is the desired quantity. This proves the induction step.

The correctness of the algorithm 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. Let $T(n)$ denote the worst case runtime of the algorithm on inputs of length n. The algorithm involves calling itself on a subproblem of size $n-1$ and then spends $O(1)$ time to return the solution. Thus, $T(n) = T(n-1) + O(1)$. By replacing $O(1)$ with some unspecified constant $C > 0$, we have,

$$T(n) = T(n-1) + C = T(n-2) + C + C = T(n-3) + C + C + C = \ldots = n \cdot C = O(n).$$

This proves that the runtime of the algorithm is $O(n)$ as desired.

Problem 3. You are given an *unsorted* array $A[1:n]$ of n distinct positive integers. Design an algorithm that in $O(n)$ worst-case time finds the largest number in $\{1,\ldots,n+1\}$ that is missing from A. **(25 points)**

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

Example. For $n = 8$ and $A = [9, 8, 70, 30, 2, 7, 4, 5]$ the correct answer is 6 (because the largest number in $\{1,\ldots,9\}$ which is missing from A is 6).

Solution. A correct solution consists of three steps, the algorithm, proof of correctness, and runtime analysis.

Algorithm: We make an additional array $C[1:n+1]$, which we will construct in the following in a way that $C[x]$ is 1 if and only if there is some $1 \leq i \leq n$ such that $A[i] = x$. In words, $C[x]$ is 1 only if we "see" x in A and otherwise it is 0. Then, we iterate over C until we find the *largest* index x such that $C[x]$ is 0, and we output this x as the answer. More concretely, our algorithm is:

1. Initialize $C[1:n+1]$ as all zeros.
2. For $i = 1$ to n: if $A[i] \leq n+1$, set $C[A[i]] = 1$.
3. For $x = n+1$ down to 1: if $C[x] = 0$, output x and terminate.

Proof of correctness: First, note that for the purpose of this problem, the elements of A which are larger than $n+1$ have no effect on the output, and can be safely ignored, as our goal is anyway to find the largest missing element from $\{1,\ldots,n+1\}$. Moreover, note that since A consists of only n elements, there must be at least one "empty hole" in the integers $\{1,\ldots,n+1\}$, i.e., at least one of these $n+1$ numbers cannot belong to A.

For any integer $1 \leq x \leq n+1$ that appears in A (say at position i_x), the for-loop in step 2 hits i_x and updates $C[x]$ to its correct value. Since A contains only distinct integers, x does not occur in the array again, and hence $C[x]$ is not updated after this iteration. For any $1 \leq x \leq n+1$ that does *not* appear in A, the initial value of $C[x]$ (which is 0) is never updated, and is hence correct.

This implies that at the end of step 2, $C[x] = 1$ for any $x \in \{1,\ldots,n+1\}$ if and only if x belongs to A. Finally, as we iterate over x from $n+1$ down to 1 in step 3, we will find the largest x missing from A and output it correctly.

Runtime analysis: Step 1 initializes an array of size $n+1$, and hence runs in $O(n)$ time. Step 2 iterates over an array of size n, and in each iteration does $O(1)$ work, and hence runs in $O(n)$ time. Step 3 iterates over an array of size $n+1$, and in each iteration does $O(1)$ work, and hence runs in $O(n)$ time.

Hence the overall running time is $O(n)$.

Problem 4. We want to purchase an item of price M and for that we have a collection of n different coins in an array $C[1:n]$ where coin i has value $C[i]$ (we only have one copy of each coin). Our goal is to purchase this item using the *smallest* possible number of coins or outputting that the item cannot be purchased with these coins. Design a **dynamic programming** algorithm for this problem with worst-case runtime of $O(n \cdot M)$. **(25 points)**

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 (**7.5 points**), the algorithm using either memoization or bottom-up dynamic programming (**5 points**), and runtime analysis (**5 points**).

Example:

- Given $M = 15$, $n = 7$, and $C = [4, 9, 3, 2, 7, 5, 6]$, the correct answer is 2 by picking $C[2] = 9$ and $C[7] = 6$ which add up to 15.
- Given $M = 11$, $n = 4$, and $C = [4, 3, 5, 9]$, the correct answer is that 'the item cannot be purchased' as no combination of these coins adds up to a value of 11 (recall that we can only use each coin once).

Solution.

Specification: For any integers $0 \leq i \leq n$ and $0 \leq j \leq M$, we define the function:

- $F(i, j)$: the minimum number of coins from $C[1:i]$ that can be used to purchase an item of value j. The answer will be $+\infty$ if there is no way to exactly purchase an item of value j using any subset of these coins.

The final answer to the problem is then $F(n, M)$.

Solution: We design the following recursive formula for $F(i, j)$:

$$F(i,j) = \begin{cases} 0 & \text{if } j = 0 \\ +\infty & \text{if } i = 0 \text{ and } j > 0 \\ F(i-1, j) & \text{if } j < C[i] \\ \min\{F(i-1, j - C[i]) + 1, F(i-1, j)\} & \text{otherwise} \end{cases}.$$

Proof of Correctness: We prove correctness of each line of this recursive formula.

For the first line, whenever $j = 0$, we need no coins to purchase of an item with value 0 and thus the right answer to $F(i, j)$ is also 0.

For the second line, whenever $i = 0$, we have no coin left to purchase the item and since $j > 0$, we cannot buy it and should instead return $+\infty$ as in the specification. (Note that the case when $i = 0$ and $j = 0$ is already handled by the first line).

For the third line, if we have $j < C[i]$, any valid solution from $C[1:i]$ cannot use the coin $C[i]$ anyway. Thus, solution to $F(i, j)$ is the same as the one on $C[1:i-1]$ for an item of value j which is $F(i-1, j)$.

For the last line, we have two options regarding coin i: either we pick it in the solution and thus should now purchase the remaining $j - C[i]$ value with the coins in $C[1:i-1]$, which gives us value $F(i-1, j-C[i]) + 1$ (the $+1$ is to account for the fact that we already added $C[i]$ to the solution). The other option is to not pick $C[i]$ and solve the problem using coins $C[1:i-1]$ which has the solution $F(i-1, j)$. Returning the minimum of these two then returns the best possible option as desired.

As these cases cover all possibilities as input to F, we get the correctness of the formula.

Dynamic Programming Algorithm: We give a bottom up algorithm for this problem.

```
1. Create an array $D[0 : n][0 : M]$ with undefined entries.
2. For $i = 0$ to $n$, set $D[i][0] = 0$. Similarly, for $j = 1$ to $M$, set $D[0][j] = +\infty$.
3. For $i = 1$ to $n$:
   (a) For $j = 1$ to $M$:
       i. If $j < C[i]$, $D[i][j] = D[i-1][j]$. Otherwise, $D[i][j] = \min(D[i-1][j - C[i]] + 1, D[i-1][j])$.
4. Return $D[n][M]$ if it is not $+\infty$. Otherwise return the problem has no valid solution.
```

As we are simply implementing the function F via this algorithm, there is no need to prove its correctness.

Runtime Analysis: There are $O(nM)$ subproblems in F and each one takes $O(1)$ to compute. So the total runtime is $O(nM)$.

Alternatively, the first line of the algorithm takes $O(nM)$ time, the second line $O(n + M)$ time, and the third line takes another $O(nM)$ time. So, the total runtime is $O(nM)$.

Problem 5 (Extra credit). You are given an *unsorted* array $A[1:n]$ of n distinct numbers with the promise that $A[1] > A[2]$ and $A[n] > A[n-1]$. Design an algorithm that in $O(\log n)$ worst case runtime finds an index $i \in \{2, \ldots, n-1\}$ such that $A[i]$ is smaller than both $A[i-1]$ and $A[i+1]$. If there are multiple indices with this property, your algorithm can output one of them. **(+10 points)**

Example: Given $A = [3, 2, 7, 8, 1, 4, 5, 6]$, one right answer is to output $A[2] = 2$, and another one is $A[5] = 1$.

Solution. A correct solution consists of three steps, the algorithm, proof of correctness, and runtime analysis.

Algorithm: We implement an algorithm similar in spirit to binary search.

```
ALG($A[1:n]$):
  1. If $n = 3$, return $A[2]$.
  2. Let $m = \lceil n/2 \rceil$. If $A[m] < A[m+1]$ and $A[m] < A[m-1]$, return $m$.
  3. Else, if $A[m] > A[m-1]$ return ALG($A[1:m]$).
  4. Else, return ALG($A[m-1:n]$).
```

Proof of correctness: The first part of the proof of correctness is to show that in any array A in this problem, there is at least one index that is a correct solution to the problem.

Claim: In any array $A[1:n]$ with distinct elements and with $A[1] > A[2]$ and $A[n] > A[n-1]$ there is an index i such that $A[i] < A[i+1]$ and $A[i] < A[i-1]$.

Proof. Assume for contradiction that such an index does not exist. We know that $A[1] > A[2]$ so we should have $A[2] > A[3]$ otherwise we will find an index that we can return as answer. We can inductively show that $A[i] > A[i+1]$ otherwise we will get an index where the value is smaller than the neighboring indices. In particular, we can show that $A[n-1] > A[n]$ but this is contradiction because we know that $A[n] > A[n-1]$. Thus, the assumption is false and an index with such a property always exists. □

We now use this to prove the correctness of the algorithm by induction. On any array of length 3, the algorithm returns $A[2]$ which is the only correct index. We now prove the correctness of the algorithm on longer arrays.

By the above claim, initially there is at least one right index as answer to the problem on array $A[1:n]$. Now, if the algorithm returns the index $m = \lceil n/2 \rceil$, by the check in line 2, we know that it will be a right answer. Otherwise, the algorithm either recurse on $A[1:m]$ or $A[m-1:n]$. In the first case, by the check of the algorithm, we have that $A[m] > A[m-1]$ and since $A[1] > A[2]$ also, by the claim above, we know array A contains at least one right index. As such, by recursing on this part of the array and by induction hypothesis for this smaller sub-array, we get that the algorithm returns a right index. This is similarly true for the second case by symmetry because in that case $A[m-1] > A[m]$ and $A[n] > A[n-1]$ so we can apply the claim again.

This concludes the proof of correctness of the algorithm.

Runtime analysis: Let $T(n)$ denote the worst case runtime of the algorithm on any array of length n (with the given properties). We have:

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

because the algorithm checks if the middle index has the property in constant time and then recurses on one half of the array which also has the given properties. This implies that $T(n) = O(\log n)$.