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

Homework #2

October 3, 2019

Name: Himesh Buch *Extension: Yes*

Problem 1. Your goal in this problem is to analyze the *runtime* of the following (imaginary) recursive algorithms for some (even more imaginary) problem:

(A) Algorithm A divides an instance of size n into 3 subproblems of size $n/2$ each, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(B) Algorithm B divides an instance of size n into 2 subproblems, one with size $n/2$ and one with size $n/3$, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(C) Algorithm C divides an instance of size n into 4 subproblems of size $n/5$ each, recursively solves each one, and then takes $O(n^2)$ time to combine the solutions and output the answer.

(D) Algorithm D divides an instance of size n in to 2 subproblems of size $n-1$ each, recursively solves each one, and then takes $O(1)$ time to combine the solutions and output the answer.

For each algorithm, write a recurrence that captures its runtime and *use the recursion tree method* to solve this recurrence and find the *tightest asymptotic* upper bound on runtime of the algorithm. **(25 points)**

Solution. The solution to problem one goes here.

(A) Recurrence for algorithm A: We can say that T(n) = 3T(n/2) + O(n)
We first replace O(n) with c · n
Each leaf will be divided by two, hence each of them will have $T(n/2^2)$ and so on upto $T(n/2^i)$ for some i.
Total time at some level i = c · $(n/2^i)$
Number of nodes at some level i = 3^i
Total = $(3/2)^i \cdot c \cdot n$
Depth d = $\log_2 n$
We know from the question that cost at each level is O(n)
Hence,

$$\text{Total cost} = \sum_{i=0}^{\log_2 n} (3/2)^i \cdot c \cdot n$$
$$T(n) \leq c \cdot n \cdot \log_2 n$$
$$= O(n \log n)$$

Upper bound = $O(n \log n)$

(B) Recurrence for algorithm B: We can say that T(n) = T(n/2) + T(n/3) + O(n)
We first replace O(n) with c · n
Each node's one leaf will be divide by the factor of (n/2), and the other wil be by (n/3). So it will go from $T(n/2^2)$ upto $T(n/2^i)$ and $T((n/2)/3)$ upto $T(n/3^i)$ for some i

Base case will be T(1)

$$\begin{aligned} T(n/2^i) &= T(1) \\ (n/2^i) &= 1 \\ n &= 2^i \\ i &= \log n \end{aligned}$$

It will result in same if we take $T(n/3^i)$, rather than $T(n/2^i)$
Depth d = $\log n$
We know from the question that cost at each level is O(n)
Upper bound = $O(n \log n)$

(C) Recurrence for algorithm C: We can say that, T(n) = 4T(n/5) + $O(n^2)$
We first replace $O(n^2)$ with $c \cdot n^2$
Each leave will be divided by a factor of (n/5) and it will go from T(n/5) upto $T(n/5^i)$ for some i
Total time at some level i = c · $(n/5^i)$
Number of nodes at some level i = 4^i
Depth d = $\log_5 n$
We know from the question that cost at each level is $O(n^2)$
Hence,

$$\begin{aligned} \text{Total cost} &= \sum_{i=0}^{\log_5 n} (4/25)^i \cdot c \cdot n^2 \\ &= \sum_{i=0}^{\infty} (1/1 - (4/25))^i \cdot c \cdot n^2 \\ &= (25/21) \cdot c \cdot n^2 \end{aligned}$$

Upper bound = $O(n^2)$

(D) Recurrence for algorithm D: We can say that, T(n) = 2T(n-1) + O(1)
We first replace O(1) with c
Each leave will start from T(n-1) upto T(n-i) for some i
Base case is T(1)

$$\begin{aligned} T(n-i) &= T(1) \\ (n-i) &= 1 \\ i &= i-1 \end{aligned}$$

Depest node = T(n-1)
We know from the question that cost at each level is O(1)
Upper bound = $O(1) \cdot O(n-1)$ = O(n-1) or O(n)

Problem 2. You are given a permutation of $\{1, \ldots, n\}$ in an array $A[1:n]$. Design a *randomized* algorithm that finds an index of some *even* number in this array in only $O(1)$ *expected runtime*. You can assume that creating a random number from 1 to n can be done in $O(1)$ time. Note that the array A is already in the memory and your algorithm does not need to 'read' this array. **(25 points)**

Example: Suppose the input is $[3, 2, 4, 5, 1]$; then the algorithm can return either index 2 or index 3.

Solution.

Objective: To design an algorithm that can find an even number from a given array in O(1) time

Algorithm:

- Create an array of n elements
- if there is only one element in the array, check if it is divisible by 2, and return the result
- if there are more than one elements, randomly pick an element from the array and see if it's divisible by 2, in order to determine if it's even
  - if it is divisible by 2, then return that element, else pick another element and see if it is even.

Proof of Correctness:

- if there is only one element, it is pretty straight forward that we will check if it is divisible by 2, and return the result. The time complexity for that is O(1)
- We basically randomly pick an element from the array and check if it is divisible by 2. By doing that we determine if the element is even or not. The run time of this step will be O(1), since we are just checking if it is divisible by 2. We are also assuming that we find the even element the first time
- Now, we already know that creating an array of size n takes O(1) time

  Hence, we can say that

$$\begin{aligned}\text{Total time} &= O(1) + O(1)\\ &= O(1)\end{aligned}$$

- The only concerning thing here is, we are chosing the element at random, so we might not be able to find the even element the first time, hence we repeat the process of picking and comparing n times. Time complexity of that is,

$$\begin{aligned}\text{Total time} &= n * O(1) + O(1)\\ &= O(1)\end{aligned}$$

  Hence, proved.

Problem 3. You are given an array $A[1:n]$ of n positive *real* numbers (not necessarily distinct). We say that A is satisfiable if after sorting the array A, $A[i] \geq i$ for all $1 \leq i \leq n$. Design an $O(n)$ time algorithm that determines whether or not A is satisfiable. **(25 points)**

Example: $[1.5, 4.1, 0.5, 5.3]$ is *not* satisfiable: after sorting we get $[0.5, 1.5, 4.1, 5.3]$ and $A[1] = 0.5 < 1$. But array $[1.5, 4.1, 2.4, 5.3]$ is satisfiable: after sorting we get $[1.5, 2.4, 4.1, 5.3]$ and for $i \in \{1, 2, 3, 4\}$, $A[i] \geq i$.

Solution.

Objective: To design an algorithm that can satisfy the condition A[i] $\geq$ i for all $1 \leq$ i $\leq$ n for a given sorted array A in O(n) time

Algorithm:

- we basically check, for some index i, if there are at least i elements less than index i, the array is not satisfiable, otherwise it is satisfiable.
- We also keep a counter variable that keeps count of i

Proof of Correctness:

- We iterate over each element and check how many elements are less than the index
- The counter variable will keep the count, and we can compare in the end to see if the array is satisfiable
- For example, in the first iteration we will check how many elements are less than 1. If there is at least 1 element less than one, the array will not be satisfiable
- We do the same thing with next iterations
- Once we know how many elements are less than their index, we will compare them
- For example, for some array A,

  Elements less than 1 = Elements less than 0
  Elements less than 2 = Elements less than 0 + Elements less than 1
  Elements less than 3 = Elements less than 0 + Elements less than 1 + Elements less than 2
  Elements less than 4 = Elements less than 0 + Elements less than 1 + Elements less than 2 + Elements less than 3

  and so on ...
- With this logic, there is only one for loop, or we iterate over the items only once, and hence, it will take linear time or time of O(n)

Problem 4. Alice and Bob are delivering for the Pizza store that has n orders today. From past experience, they both know that if Alice delivers the i-th order, the tip would be $A[i]$ dollars, and if Bob delivers, the tip would be $B[i]$ dollars. Alice and Bob are close friends and their goal is to maximize the total tip they can get and then distribute it evenly between themselves. The problem is that Alice can only handle a of the orders and Bob can only handle b orders. They do not know how to do this so they ask for your help.

Design a dynamic programming algorithm that given the arrays $A[1:n], B[1:n]$, and integers $1 \leq a, b \leq n$, finds the largest amount of tips that Alice and Bob can collect together subject to their constraints. The runtime of your algorithm should be $O(n \cdot a \cdot b)$ in the worst case. **(25 points)**

Example: Suppose $A = [2, 4, 5, 1]$, $B = [1, 7, 2, 5]$, $a = 2$ and $b = 1$. Then the answer is 14 by Alice delivering $2, 3$ and Bob delivering 4 (so $4 + 5 + 5 = 14$), or Alice delivering $1, 3$ and Bob delivering 2 ($2 + 5 + 7 = 14$).

Solution.

Objective: To design an algorithm that can help Bob and Alice get maximum amount of tips in $O(n \cdot a \cdot b)$ time

Algorithm:

- We have to take the bottom up approach rather than start looking from top, so we will start by looking at who delivers the last pizza.
- Keeping the above point in mind, there could be three possibilities,

  - Bob delivers the last pizza
  - Alice delivers the last pizza
  - No one delivers the last pizza
- Looking at these possibilities we can decide who delivers what amount of pizzas
- We will arbitarily decide who delivers what amount of pizzas, and reduce the number of pizzas once Bob or Alice (or noone) delivers it
- The important thing here is, say for example, out of n pizzas, Bob decides to deliver the last pizza, we will only have to consider the other n-1 choices, since the last pizza is delivered.
- We will also see if Bob or Alice delivering the last pizza help them get the most amount of tips
- We are also given that Bob and Alice delivers $\leq$ n amount of pizzas where n is the length of the array.
- Hence, for Alice, she can deliver n, or (n-1), or (n-2), ..., 1, and same with Bob.
- Hence, we can represent it as a 3x3 matrix, that will help us determing the permutations

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ \vdots & \vdots & \vdots \\ n-1 & a-1 & b-1 \\ n & a & b \end{bmatrix}$$

- The above matrix will help us determinig the time complexity of the algorithm as well
- Not only that, once we know who delivers what amount of pizzas, we will compare those indexes to get the most amount of tips. For example, assuming Bob delivers n^{th} pizza (last pizza), and we have arbitarily assigned a = 3 and b = 3, now we will be left with a = 3 and b = 2, with a total of n-1 pizzas left. Of course we will check if Bob delivering the last pizza help them get the most tips

Recursive formula:

- **Specification:** For some orders between 1$\leq$a, b$\leq$n, for both Bob and Alice, P(A[i], B[i]), is the max amount of tip they can get, where A[i] and B[i], is the tip.
- **Recursive solution:** Taking the bottom up approach we can say that, P(A[i], B[i]) will be zero, if niether Bob nor Alice delivers the last order. For the other two possibilities (either Bob or Alice delivering last order), P(A[i], B[i]) will be the sum of P(A[i], B[i]) and P(A[i-n], B[i-n]), meaning, we don't consider the order for both Bob and Alice once we know who delivers it.

Proof of Correctness:

- Here we have a total of n choices, for the number of pizzas. In order to determine how many pizzas Bob and Alice will deliver, we will have to repeat the process n times
- As mentioned in the algorithm, if Bob decides to deliver the last pizza, we will only consider the other n-1 pizzas that are not delivered
- For Bob, he can deliver 1, 2, ..., b amount of pizzas. This will take b $\cdot$ O(n), which is O(b).
- Same thing goes for Alice, it will take O(a) time

- Now, the total time,

$$\begin{aligned}\text{Total time} &= nO(a \cdot b) \\ &= O(n \cdot a \cdot b)\end{aligned}$$

- We notice here that, chosing a and b happens simultaneously, hence the total time for that will be $O(a \cdot b)$, and since we are repeating the process n times, total time will be $O(n \cdot a \cdot b)$
- Here this will be the worst case because we are comparing each time if delivering that pizza can help them achieve bigger tip. (we are comparing all array indices basically)

Memoization: We can speed up the computation of this recursive algorithm significantly by simply storing the computed answers in a table and make sure we do not recompute them again and again by consulting this table. Specifically, we can do as follows. We first see arrays A[1:n] or B[1:n]. We then run the following algorithm:

- Arbitarily chose who delivers th n^{th} pizza
- Now consider only A[n-1] and B[n-1] array
- For all other entries in array, check *if A[i] > B[i]*, where i is some index and if it is true, then *a = a-1, and b = b*
- else *if B[i] > A[i]*, then *b = b-1, and a = a*
- Recursivley call P(A[i], B[i]) + P(A[i-n], B[i-n]), to add up the tip, and return once it has reached the end
- Hence, we recursivly added the tips and made sure to decrease the number of pizzas that were supposed to be deliverd (a and/or b)

Challenge Yourself. A standard dynamic programming algorithm for Problem 4 requires $O(n \cdot a \cdot b)$ *space* in addition to the same runtime. The goal of this question is to improve this space bound and further extend the algorithm. **(0 points)**

(a) Design an algorithm for this problem that requires only $O(n + a \cdot b)$ space instead and still has the worst-case runtime of $O(n \cdot a \cdot b)$ (this part is rather standard and is not that challenging).

Solution. In above algorithm, we go to each index of the array and check who delivers pizza. Meaning, we check for n = 0, . . . , n, along with a = 0, . . . , a, and b = 0, . . . , b. Rather than doing this for each n, we do it once only for n = n. The entire algorithm stays the same, but unlike before now we have a 2x2 matrix with fixed n,

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \\ \vdots & \vdots \\ a-1 & b-1 \\ a & b \end{bmatrix}$$

- As mentioned in the algorithm, if Bob decides to deliver the last pizza, we will only consider the other n-1 pizzas that are not delivered
- For Bob, he can deliver 1, 2, . . . , b amount of pizzas. This will take b · O(n), which is O(b).

- Same thing goes for Alice, it will take O(a) time
- Now, the total time,

$$\text{Total time} = O(n + a \cdot b)$$

- We notice here that, chosing a and b happens simultaneously, hence the total time for that will be $O(a \cdot b)$, and since we are not repeating the process n times, but just taking the fixed n, hence, total time will be $O(n + a \cdot b)$
- As mentioned earlier, the only difference here is that we have a fixed n

---

Fun with Algorithms. Recall that Fibonacci numbers form a sequence F_n where $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. The standard algorithm for finding the n-th Fibonacci number takes $O(n)$ time. The goal of this question is to design a significantly faster algorithm for this problem. **(0 points)**

(a) Prove by induction that for all $n \geq 1$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}.$$

Solution.

Base case: For n = 1, we can say,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

By comparing the matrix we get,
$F_2 = 1$, $F_1 = 1$, and $F_0 = 0$, which follows the fibonachi series, $F_n = F_{n-1} + F_{n-2}$

Inductive case: For inductive case, we can say that if,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

is true then,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

Therefore,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

By substituting value,

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

By the rules of matrix multiplication,

$$\begin{bmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

We can say from fibonachi sequence that,
$F_{n+2} = Fn + 1 + F_n$, and
$F_{n+1} = Fn + F_{n-1}$
Substituting those values, we get,

$$\begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

Therefore, LHS = RHS
Hence, proved

(b) Use the first part to design an algorithm that finds F_n in $O(\log n)$ time.

Solution.

The problem with above algorithm is that it calls the same values over and over, and thus, the it takes huge amount of time in computing the fibonachi sequence (exponential time!)
In order to modify the algorithm, we do,
We rewrit the main equation in matrix form,

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

Now, we can say the same thing for the next pair,

$$\begin{bmatrix} F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

So, in general,

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

What we did here is similar to what is given in the question but faster
In order to compute F_n, we raise the matrix to n^{th} power, hence we get the value of T(n) = T(n/2) + O(n), and by Recursion tree method,

$$\begin{aligned} \text{Total cost} &= \sum_{i=0}^{\log_2 n} (1/2)^i \cdot c \cdot n \\ T(n) &\leq c \cdot \log_2 n \\ &= O(\log n) \end{aligned}$$

And thus, total time taken is reduced by $O(\log n)$
We achieved this by reducing the artihmetic operation by (n-1)
