CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #2

February 9, 2021

Name: Ryan Coslove *Extension: No*

Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "use a hash table of size m with chaining on an array of length n to get expected worst-case runtime of $O(1 + \frac{n}{m})$ for searching each element".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question).
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

Problem 1. Suppose we have an array $A[1:n]$ of n *distinct* numbers. For any element $A[i]$, we define the **rank** of $A[i]$, denoted by $rank(A[i])$, as the number of elements in A that are strictly smaller than $A[i]$ plus one; so $rank(A[i])$ is also the correct position of $A[i]$ in the sorted order of A.

Suppose we have an algorithm **magic-pivot** that given any array $B[1:m]$ (for any $m > 0$), returns an element $B[i]$ such that $m/3 \leq rank(B[i]) \leq 2m/3$ and has worst-case runtime $O(n)$[1].

Example: if $B = [1, 7, 6, 2, 13, 3, 5, 11, 8]$, then **magic-pivot**$(B)$ will return one arbitrary number among $\{3, 5, 6, 7\}$ (since sorted order of B is $[1, 2, 3, 5, 6, 7, 8, 11, 13]$)

(a) Use **magic-pivot** as a black-box to obtain a deterministic quick-sort algorithm with worst-case running time of $O(n \log n)$. **(10 points)**

Solution.

The general worst case time complexity of quicksort is $O(n^2)$. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot. To get an $O(n \log n)$ time, we will use partition algorithm where the median of the array is picked as the pivot. Using magic-pivot, we will sort the array into thirds (three subarrays) and quick sort them by using the median as the partition pivot.

Proof of Correctness

[1]Such an algorithm indeed exists, but its description is rather complicated and not relevant to us in this problem.

We now prove the induction step: suppose quick sort can sort all arrays of length $n \leq k$ and we prove it also correctly sorts any array of size $n = k+1$. Let A be any array of size $n = k+1$. Quick sort first runs partition (with median as the pivot) to partition the array into $A[1 : q/3-1], A[q/3+1 : 2q/3-1]$ and $A[2q/3+1 : n]$, where A[q] is in its correct position and every element in $A[1 : q/3-1]$ is at most as large as A[q] while every element in $A[2q/3+1 : n]$ is at least as large as A[q]. The algorithm then recursively sorts $A[1 : q/3-1]$ and $A[2q/3 : n]$ – since size of both array is smaller than $k+1$, by induction hypothesis, both arrays are sorted correctly. By the guarantee of the partition algorithm and this sorting step, we have $A[1 : q/3-1]$ is sorted array of elements $\leq$A[q], and $A[2q/3+1 : n]$ is the sorted array of elements $\geq$ A[q]; hence A is also now sorted.This proves the induction step and the correctness of the algorithm.

Runtime Analysis

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

$T(n) = O(n \log n)$

(b) Use **magic-pivot** as a black-box to design an algorithm that given the array A and any integer $1 \leq r \leq n$, finds the element in A that has rank r in $O(n)$ time[2]. **(15 points)**

Hint: Suppose we run **partition** subroutine in quick sort with pivot p and it places it in position q. Then, if $r < q$, we only need to look for the answer in the subarray $A[1 : q]$ and if $r > q$, we need to look for it in the subarray $A[q+1 : n]$ (although, what is the new rank we should look for now?).

Solution.

Given that the array has been sorted with quick sort, we can compare r to q, the correct position of a sorted array. If $r \leq q/3-1$ but r$\geq A[1]$, then iterate through $A[1 : q/3-1]$ to find the element in A that has rank r. If $r \geq q/3+1$ but $r \leq 2q/3-1$ then iteratre through $A[q/3+1 : 2q/3-1]$. If $r \geq 2q/3-1$ and $r \leq n$ then iterate through $A[2q/3+1 : n]$. This will find the element in A that has rank r. Because we are iterating through a sorted array, it should take $O(n)$ time. $T(n) = O(n)$

Problem 2. Suppose we have an array $A[1 : n]$ which consists of numbers $\{1, \ldots, n\}$ written in some arbitrary order (this means that A is a *permutation* of the set $\{1, \ldots, n\}$). Our goal in this problem is to design a very fast randomized algorithm that can find an index i in this array such that $A[i] \mod 3 = 0$, i.e., $A[i]$ is divisible by 3. For simplicity, in the following, we assume that n itself is a multiple of 3 and is at least 3 (so a correct answer always exist). So for instance, if $n = 6$ and the array is $A = [2, 5, 4, 6, 3, 1]$, we want to output either of indices 4 or 5.

(a) Suppose we sample an index i from $\{1, \ldots, n\}$ uniformly at random. What is the probability that i is a correct answer, i.e., $A[i] \mod 3 = 0$? **(5 points)**

Solution.

Because n is always a multiple of 3, there are $\frac{n}{3}$ numbers divisible by 3. If we randomly select any 1 index there's exactly $\frac{1}{3}$ chance it's divisble by 3 per: $A[i] \mod 3 = 0$.

Example: n = 3. Let's say A = [2, 3, 1]. When we randomly select A[i], only 1 out of 3 are divisble by 3, giving us our answer of probability $\frac{1}{3}$. This is true for any value n that is a multiple of 3.

---

[2]Note that an algorithm with runtime $O(n \log n)$ follows immediately from part (a)—sort the array and return the element at position r. The goal however is to obtain an algorithm with runtime $O(n)$.

(b) Suppose we sample m indices from $\{1, \ldots, n\}$ uniformly at random and with repetition. What is the probability that none of these indices is a correct answer? **(5 points)**

Solution.

The probability of selecting an i that is divisible by 3 is still 1/3. That stays true with repetitions as well. Using binomial distribution to get probabilty of the event not occuring n times in a row is: $Pr(x) = \binom{n}{m} * p^m * q^{n-m}$.

P = proability, n = number of trials, m = number of trials for specific outcome in n trials, p = probability of success in a single trial, q = probability of failure in a single trial. p will equal 2/3 because there is a 2/3 chance of an index not being divisible by 3.

The probability of $Pr(x) = \frac{2}{3}^m$ or $O(x) = \frac{2}{3}^m$

Now, consider the following simple algorithm for this problem:

Find-Index-1$(A[1:n])$**:**

- Let $i = 1$. While $A[i] \mod 3 \neq 0$, sample $i \in \{1, \ldots, n\}$ uniformly at random. Output i.

The proof of correctness of this algorithm is straightforward and we skip it in this question.

(c) What is the **expected** worst-case running time of **Find-Index-1**$(A[1:n])$? Remember to prove your answer formally. **(7 points)**

Solution.

Imagine random variable x whose range is {1, 2, ... n}. Of course, this sample could run infinitely as we may never reach an $A[i] \mod 3 = 0$.

However let's find the expected worst-case running time of Find-Index-1(A[1:n]), as infinity is not the expected. The expected value of x is $E[x] = \sum_{i=1}^{\infty} Pr(x = i) * i$. If we say that it runs through all numbers in the array 1 times, then x will value 1. This would mean the runtime is O(x) or O(1). Plus, the run time from our answer in part b is included in the expectation. So the expected run time is $Pr(x = i) * i = O(\frac{2}{3}^m) * O(1) => Pr(x = i) * i => O(\frac{2}{3}^m)$

The problem with **Find-Index-1** is that in the worst-case (and not in expectation), it may actually never terminate! For this reason, let us consider a simple variation of this algorithm as follows.

Find-Index-2$(A[1:n])$**:**

- For $j = 1$ to n:
  - Sample $i \in \{1, \ldots, n\}$ uniformly at random and if $A[i] \mod 3 = 0$, output i and terminate; otherwise, continue.
- If the for-loop never terminated, go over the array A one element at a time to find an index i with $A[i] \mod 3 = 0$ and output it as the answer.

Again, we skip the proof of correctness of this algorithm.

(d) What is the **worst-case running time** of **Find-Index-2**($A[1 : n]$)? What about its **expected** worst-case running time? Remember to prove your answer formally.

(8 points)

Solution.

Now, we iterating through the array to see if $A[i] \mod 3 = 0$. This iteration will go at O(n) time, going up to the nth element of the array in worst-case.

Let us have 2 random variables x and y where $x \epsilon \{1, 2, ..., n\}$ is the iterations in the for-loop and $y \epsilon \{0, n\}$ for the time it takes in the second step. Run time is $O(x + y)$ or $E[x + y] = E[x] + E[y]$

We know $E[x] = O(1)$. $E[y] = Pr(y = 0) * 0 + Pr(y = n) * n$. Because $Pr(y = n) \leq \frac{2}{3}^m$, we can upper bound E[y] to $m\frac{2}{3}^m$

Now, our total run time should be: $O(m\frac{2}{3}^m)$

Problem 3. Given an array $A[1 : n]$ of a combination of n positive and negative integers, our goal is to find whether there is a sub-array $A[l : r]$ such that

$$\sum_{i=l}^{r} A[i] = 0.$$

Example. Given $A = [13, 1, 2, 3, -4, -7, 2, 3, 8, 9]$, the elements in $A[2 : 8]$ add up to zero. Thus, in this case, your algorithm should output *Yes*. On the other hand, if the input array is $A = [3, 2, 6, -7, -20, 2, 4]$, then no sub-array of A adds up to zero and thus your algorithm should output *No*.

Hint: Observe that if $\sum_{i=l}^{r} A[i] = 0$, then $\sum_{i=1}^{l-1} A[i] = \sum_{i=1}^{r} A[i]$; this may come handy!

(a) Suppose we are promised that every entry of the array belongs to the range $\{-5, -4, \ldots, 0, \ldots, 4, 5\}$. Design an algorithm for this problem with worst-case runtime of $O(n)$. **(15 points)**

Hint: Counting sort can also be used to efficiently sort arrays with negative entries whose absolute value is not too large; we just need to "shift" the values appropriately.

Solution. Because counting sort uses positive integers and every array we use will be within the range of {-5, ... , 5}, it is important to shift the numbers to be positive to do our sorting. Given the first index is -5, we will want to shift all the elements by +6. Now, the range is from {1, 2, ... , 10, 11}. Now we can implement counting sort.

- Shift by 6: $A[i] = A[i] + 6$
- Create an array $C[1 : M]$ and initailize it to be all 0.
- For $i = 1$ to n: increase $C[A[i]]$ by one.
- Let $p = 0$ and for $j = 1$ to M:
- While $C[j] > 0$: decrease $C[j]$ by one; let $A[p] = j$, and increase p by one.
- Reverse shift by 6: $A[i] = A[i] - 6$

Proof of correctness:

Now that we have array C, we update A. As A is being updated, we shift the elements back to their original integers. For every $1 \leq j \leq M, C[j]$ is equal to the number of times number j appears in the array A.

For any iteration j of the for-loop in this line, define p_j as the value of pointer p after this line. The proof is by induction over index j. Our induction hypothesis is that for every $1 \leq j \leq M$, after iteration j of the for-loop, all the numbers in the original array A that were $\leq j$ are now placed in asorted order in the new array $A[1 : p_j - 1]$.

- Base case: For $j = 1$, the while-loop places $C[1]$ many copies of number of 1 in the array $A[1 : p_1]$ (which may be zero copies). Since $C[1]$ was equal to the number of times number 1 appears in the old array A, this means that after iteration 1, we copied number of 1's copies of 1 in $A[1 : p_1]$, proving the induction step.
- Induction step: Suppose this is true for all integers up tojand we prove it for j+ 1. By induction hypothesis, we already placed all copies of {1, . . . , j}in the array $A[1 : p_j - 1]$. In iteration $j + 1$, the while-loop places $C[j + 1]$ many copies of number $j + 1$ in the array $A[p_j : p_j + 1 - 1]$ - this makes the array $A[1 : p_j + 1 - 1]$ contains all elements in the array A that are $\leq j + 1$ in a sorted order (since every element in $A[p_j : p_j + 1 - 1]$ is equal to $j + 1$ and is thus larger than all previous elements that were sorted by induction hypothesis for j). This proves induction step.

We can now apply our induction hypothesis to $j = M$ and have that the array $A[1 : p_M - 1]$ contains all elements in the original array A that are in the range {1, . . . , M} in a sorted order. Since all elements of A were in this range after we shifted them back, this means that the new array is now a sorted version of A, proving the correctness.

Runtime Analysis:

It takes $O(n)$ time to iterate over all elements and $M = O(n)$ in counting sort so the worst case run time is $O(n)$.

(b) Now suppose that there is no promise on the range of the entries of A. Design a randomized algorithm for this problem with expected worst-case runtime of $O(n)$. **(10 points)**

Solution. We will use a randomized hash function. We will have a hash family, H, that will include hash functions. We will make the hash family universal, to avoid collisions. This would be for every choice of integers $x \neq y$, $Pr_{h \epsilon H}(h(x) = h(y)) \leq \frac{2}{m})$). We iterate through array A[i], calculate elements from 0 to i (sum += A[i]). If the current sum has been seen before, then there is a zero-sum array. Use hash to store the sum values so we can see if a curren sum has been seen before.

$H = h_a(x) = ((a * x \mod p) \mod m) + 1 | 1 \leq a \leq m - 1$

E[number of collisions] $= \sum_{i \neq j} E[X_{i,j}] \leq \sum_{i \neq j} \frac{1}{n} = \binom{n}{2} * \frac{1}{n} < n^2 * \frac{1}{n} = n$

So when we are given an array A, we will run it through a Hash family and its hash functions, randomly, and the hash functions will tell us if the values will sum to 0. Because the expected number of collisions when running through array A with the hash functions is expected n times, mapping every sum to a single position, we can expect the worst case run time to be $O(n)$.

Problem 4. We want to purchase an item of price n and for that we have an unlimited (!) supply of three types of coins with values 5, 9, and 13, respectively. Our goal is to purchase this item using the *smallest* possible number of coins or outputting that this is simply not possible. Design a dynamic programming algorithm for this problem with worst-case runtime of $O(n)$. **(25 points)**

Example. A couple of examples for this problem:

- Given $n = 17$, the answer is "not possible" (try it!).
- Given $n = 18$, the answer is 2 coins: we pick 2 coins of value 9 (or 1 coin of value 5 and 1 of value 13).
- Given $n = 19$, the answer is 3 coins: we pick 1 coin of value 9 and 2 coins of value 5.
- Given $n = 20$, the answer is 4 coins: we pick 4 coins of value 5.
- Given $n = 21$, the answer is "not possible" (try it!).
- Given $n = 22$, the answer is 2 coins: we pick 1 coin of value 13 and 1 coin of value 9.
- Given $n = 23$, the answer is 3 coins: we pick 1 coin of value 13 and 2 coins of value 5.

Solution.

Let's make an array where d[i] is minimum number of coins needed to make i. $d[i] = INTMAX$ if not possible to make i.

3 options to make i:

- 1. if $(i \geq 5)$, then $d[i] = min(d[i], d[i-5] + 1)$.
- 2. if $(i \geq 9)$, then $d[i] = min(d[i], d[i-9] + 1)$.
- 3. if $(i \geq 13)$, then $d[i] = min(d[i], d[i-13] + 1)$.

We know this is correct because we take the value we are looking for, and recursively are decreasing the value n by the coin value we are using, then adding 1 to the type of coin we are using to keep track of how much of each coin we are using. Like in the example of n=23, we create d[i] and check that if $23 \geq 13$, we subtract 13 from 23 and add 1 to the value of number of 13-value coins we are using. Next it will recursively call to see that $10 \geq 5$ so we subtract 5 from 10 and the number of 5 coins increases by 1. That same step repeats and we finish with 1 coin of 13 and 2 coins of 5.

Time complexity is $O(n)$ because we only run a loop from 1 to n to fill the d array.

Challenge Yourself. Suppose we have two arrays $A[1:n]$ and $B[1:m]$ which are both in the sorted order and are consisting of distinct numbers. Design an algorithm that given an integer $1 \leq r \leq m+n$, find the element with rank r in the union of arrays A and B. Your algorithm should run in only $O(\log(n+m))$ time.

(+10 points)

Example. Suppose $A = [1, 5, 7, 9]$ and $B = [2, 4, 6, 12]$ and so $n = m = 4$. Then, the answer to $r = 3$ is 4 and the answer to $r = 7$ is 9 because the union of arrays A and B in the sorted order is $[1, 2, 4, 5, 6, 7, 9, 12]$.

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. **(+10 points)**

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

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

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