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

Homework #2

February 9, 2021

Name: Letao Zhang *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. *Algorithm.* Firstly we sort array B using quick-sort. Instead of using the median as pivot, we use magic-pivot m/3 and 2m/3 divide those array into 3 parts. B1 = B[1...(m/3)-1], B2 = B[(m/3)+1...(2m/3)-1], B3 = A[(2m/ 3)+1...m]. The logic is simple, after sorting array, we start from the leftmost element and keep track of index of smaller (or equal to) elements as m/3, furthermore 2m/3. While traversing, if we find a smaller element, we swap current element with B[i]. Otherwise we ignore current element.

Proof of Correctness. By the correctness of quick sort, we can assume that B will be sorted correctly. The proof is by induction on the value of i, and the statement is that after iteration i of the for-loop,

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

rank of numbers is computed correctly.
For the base case, we have that when i = m/3, the algorithm output rank of number is the smallest in the array, this is certainly correct.
For the induction step, let us assume the statement is true for i = m/3 and we prove it for i = m/3 + 1. By induction hypothesis, rank of numbers B[m/3 : 2m/3] is computed correctly; now if B[m/3 + 1] = B[m/3], we give the same rank to B as well which will be correct as the rank of number.
This proves the induction step and thus the entire induction hypothesis.

Runtime Analysis. As from the above algorithm the array is partitioned in such three parts. Thus, it is possible that in worst case, array is partitioned in three parts such that one part is smaller than m/3, one part is greater and equal to m/3 and smaller and equal to 2m/3, and other part is greater than 2m/3, makes its time complexity will be O(log N). Whereas there is a loop in the above algorithm. So, complexity is O(n). Therefore, the total runtime is $O(n \log n)$.

(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. *Algorithm.* Firstly 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]. Let A[1:q] denote the given array and denote the order statistic by r. The black-box subroutine on A returns the q element. While traversing, if we find a smaller element, we swap current element with A[q]. Otherwise we ignore current element.

Proof of Correctness. By the correctness of quick sort, we can assume that A will be sorted correctly. We now find those numbers correctly. The proof is by induction, and the statement is that after iteration i of the for-loop, rank of numbers A[1 : q] is computed correctly.
For the base case, we have that when i = 1, the algorithm output rank of number is sorted, this is certainly correct.
For the induction step, let us assume the statement is true for i = q and we prove it for i = q + 1. By induction hypothesis, rank of players A[1 : q] is computed correctly; now if A[q + 1] = A[q], we give the same rank to A[q + 1] as well which will be correct as the rank of numbers. On the other hand, if A[q + 1] ¿ A[q], then since A is sorted, we know that A[q + 1] is the smallest number larger than A[q]. This proves the induction step and thus the entire induction hypothesis. By applying the hypothesis, we get that elements in A computed correctly.

Runtime Analysis. This final approach takes O(n) time to find the elements.

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.

[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 $O(n)$.

(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. The probability would be 1/3. Because we assume that n itself is a multiple of 3 and is at least 3, A[i] is divisible by 3. So for i is a correct answer, there will be always one correct answer within 3 numbers, makes the prob be 1/3.

(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 would be 2/3. In the case, we could repeatedly sample indices, makes the probability of choosing numbers which can be divided by 3 the same. Therefore, remaining are those numbers which can't be divided by 3, makes the prob be 1-1/3 = 2/3.

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. In the worst case, the Find-Index-1() function compares all the elements of array A. Therefore, the expected worst case time complexity of linear search would be $O(n)$.

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. In the worst case, the Find-Index-2() function still need to compare all the elements of array A. Therefore, the worst case time complexity of linear search would be $O(n)$, and the expected worst-case running time would be $O(n)$.

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. *Algorithm.* Given an input array A[1 : n] with values in 1, 2, . . . , M. 1) Create an array C[1 : M] and initialize it to be all 0. 2) For i = 1 to n: increase C[A[i]] by one. 3) 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.

Proof of Correctness. We first observe that after Line (2) of the algorithm, C[j] is equal to the number of times number j appears in the array A. We now consider Line (3): for any iteration j of the for-loop in this line, define pj as the value of pointer p after this line. The proof is by induction over index j. After iteration j of the for-loop, all the numbers in the original array A that were ¡= j are now placed in a sorted order in the new array.
Base case: For j = 1, the while-loop places C[1] many copies of number of 1 in the array A[1 : p1]. 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 : p1], proving the induction step.
Induction step: Suppose this is true for all integers up to j and we prove it for j + 1. By induction hypothesis, we already placed all copies in the array A. In iteration j + 1, the while-loop places C[j + 1] many copies of number j + 1 in the array A, which makes the array A contains all elements in the array A that are $\leq j + 1$ in a sorted order (since every element in A 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 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, this means that the new array is now a sorted version of A, proving the correctness.

Runtime Analysis. This would be O(n) as we only run a for-loop once.

(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. *Algorithm.* The maximum value we can obtain by picking a subset of the first i items, i.e., items 1, . . . , i, when we have a knapsack of size r. Given an input array A[1 : n] with values in 1, 2, . . . , m. 1) Create an array C[1 : m] and initialize it to be all 0. 2) For i = 1 to n: increase C[A[i]] by one. 3) 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.

Proof of Correctness. At this point, we have simply written some recursive formula for A[i] but in no way it is clear that whether this formula does what it is supposed to do or not. We thus need to also prove that this is indeed a correct formula for computing A[i]. Let us consider the base case of this function i = 0 . In this case , we have A[i] = 0 which is also the value we can achieve by using the first 0 item or when the knapsack has no size (i.e., r = 0). So the base case of this function matches the specification.
We now consider the larger values of i and r. Suppose firstly $i > r$, A[i] equals A[i-1]. This is correct because we cannot fit item i in a knapsack of size j and thus the best value we can achieve is by picking the best combination of items from the first i -1 items, which is captured by A(i -1). Thus, whenever i ¿ r, A[i] equals A[i - 1] precisely captures the specification we had.

Runtime Analysis. Since we only compute each recursive call once, the total runtime is then $O(n)$.

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. *Algorithm.* The first and most important step (for the n-th time) is to write a recursive formula/algorithm for the problem, specify it in plain English, and write a solution for it (including the proof of correctness). Firstly make a 1-D array where A[i] represents the minimum number of coins. Then, we let A[i]= INT MAX if it is not possible to make i using the given coins. Furthermore, we have 3 ways to make i element: a) if ($i \geq 5$) A[i] = min(A[i], A[i-5] + 1). b) if($i \geq 9$) A[i] = min(A[i], A[i-9] + 1), c) if($i \geq 13$) A[i] = min(A[i], A[i-13] + 1).

Proof of Correctness. We consider each case of the recursive formula separately: If i = 1: there is only one subsequence of A[1 : 1] which is A[1] and is increasing on its own, so it is the correct choice. We now consider larger values of i.
Firstly, for base case, there are no coins required to make 0. A[0] = 0. For int i=1 to n, A[i]= INT MAX. After we pick A[i], the previous entries in the subsequence should all be smaller than A[i]. Moreover, if we decide to pick some $j < i$ where $A[j] < A[i]$ in the subsequence also, we should pick the longest subsequence that ends in A[j] so that we can also maximize the length of the longest subsequence that ends in A[i]. As

we are interested in taking the longest sequence, we are simply taking a maximum of all available choices. This concludes the proof of correctness of the formula.

Runtime Analysis. This dynamic programming algorithm has worst-case runtime of $O(n)$ as we only run a loop from 1 to n though entire array A.

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.

Solution. *Algorithm.* Given an input array A[1 : n] with values in 1, 2, . . . , m. Create an array C[1 : m] and initialize 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.

Proof of Correctness. We first observe that after Line (2) of the algorithm, for every $1 \leq j \leq m$, C[j] is equal to the number of times number j appears in the array A (this is a very basic observation and we do not need to prove it really). We now consider Line (3): for any iteration j of the for-loop in this line, define pj 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 ¡= j are now placed in a sorted order in the new array A.
Base case: For j = 1, the while-loop places C[1] many copies of number of 1 in the array A[1 : p1]. 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 : p1], proving the induction step.
Induction step: Suppose this is true for all integers up to j and we prove it for j + 1. By induction hypothesis, we already placed all copies of 1, . . . , j in the array A. In iteration j + 1, the while-loop places C[j + 1] many copies of number j + 1 in the array A – this makes the array A contains all elements in the array A that are $\leq j+1$ in a sorted order (since every element in A 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.

Runtime Analysis. The first line of the algorithm takes O(m) time to initialize the array C. The second line takes O(n) time to iterate over all elements. The third and fourth lines together take O(n + m) time since each iteration of the for-loop increases j by 1 and j can only increase m time, and each iteration of the while-loop increases p and p can only increase n times in total. Hence, the runtime is $O(n+m)$.

(+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 $n \geq 1$:

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

Solution. For the base case n = 1, we know that

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$$

which is correct.

Then, for n = k,

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^k = \begin{bmatrix}F_{k+1} & F_k\\ F_k & F_{k-1}\end{bmatrix}$$

For the induction step, n = k + 1,

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^(k+1) = \begin{bmatrix}F_{k+2} & F_k+1\\ F_k+1 & F_k\end{bmatrix}$$

Thus,

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^(k+1) = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^k \qquad * \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$$

According to n = k, we consider that

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^(k+1) = \begin{bmatrix}F_{k+1} & F_k\\ F_k & F_{k-1}\end{bmatrix} \qquad * \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$$

Therefore, solving it we get

$$\begin{bmatrix}F_{k+1}+F_k & F_{k+1}+0\\ F_k+F_{k+1} & F_k+0\end{bmatrix} = \begin{bmatrix}F_{k+2} & F_k+1\\ F_k+1 & F_k\end{bmatrix} \qquad = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^(k+1)$$

Hence, it is proved by induction.

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