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

Homework #2

February 9, 2021

Name: Yinfeng Cong *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. As we learn in the lecture of the week of quick sort. The recurrence of it is $T(n) \leq max\ (T(q-1) + T(n-q) + O(n))$
To make $T(n) = O(n \log(n))$, we need to make q = $\frac{n}{2}$
To make sure we get nlogn for the running time, we need to apply magic-pivot to help us with it.
Algorithm:
For array A[1:n]
we use magic-pivot to get rank(A[i]) that $m/3 \leq rank(A[i]) \leq 2m/3$, we plug in rank(A[i]) as the number q in the quick sort recurrence

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

we find that the runtime of the quicksort would be nlogn
Proof of correctness:
We know the correctness of magic-pivot and quick sort, what we want to prove is that for any number q that , $m/3 \leq q \leq 2m/3$ the runtime of the function would be nlogn
basecase: q = n/3
$T(n) \leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$, and the runtime of it is $T(n) \leq cn\log_3(n) = O(nlogn)$.
it obeys the answer
As we learn in the lecture, the runtime of the quick sort is also nlogn for the time that q = n/2
And for q = 2n/3, the equation is the same as the equation of q = n/3, so its runtime is also nlogn
As we learned in the lecture, when q = n/2, the T(q-1) would be seen as T(n/2) but not T(n/2 -1)
The reason of this is that n/2 is much larger than 1, and 1 is just a constant so we do not need to show this 1 out
So the recurrence would become $T(n) \leq max\ (T(q) + T(n-q) + O(n))$
We suppose that the function works for q = k, and we want to prove that it also works for q = k+1
When we plug in q = k+1, we get $T(n) \leq\ (T(k+1) + T(n-k-1) + O(n))$
because $m/3 \leq k+1 \leq 2m/3$, so the number of levels of the recurrence tree is always logn, and the sum of each level is always Cn
As a result, the sum of the runtime is $T(n) =\ \log(n) \cdot C \cdot n = O(n\log(n))$

(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. Solution to part (b) goes here.
For this question, we know that the rank of the element we want is r
Algorithm:
The way to solve this problem is run magic-pivot first to find a proper pivot for the partition.
Then, we use the pivot we find q as the pivot of the partition to see if the rank of q is r, if it is r, we just see that the element of rank r is q
If it is not the rank we want, we just see if r is smaller than the rank of q or bigger than it.
If rank of q is bigger than r, we use magic pivot and partition again in [1:q] until we find the element has rank r
If rank of q is smaller than r, we use magic pivot and partition again in [q+1 : n], and we find the element in the new array that has the rank r-q
Both the situation runs recursively until we get the answer r, and the runtime of it is $C \cdot O(n)$ (we need to run partition for several times, and each time take O(n) for runtime), total runtime = O(n)
Proof of correctness:
basecase:
when we need to fnd the element of rank r, the proof has just been shown in the algorithm above
induction:
We have proved partition in the lecture, and we use magic pivot as a blackbox, so we do not need to prove for the correctness of them at all
we know that it is correct for rank = r, and we want to prove it correct for rank = r+1
But we have known the correctness of magic-pivot and partition, and the process of the algorithms are just the same
So we can also get then rank of r+1 using the same method
As a result, the method is correct

[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 $O(n)$.

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. Solution to part (a) goes here.
For the reason that n is the multiple of 3 and whenever we have 3 elements in the array, we have one answer that mod3 = 0
So the probability of it is 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. Solution to part (b) goes here.
For the reason that we can pick the sample with repetition, and we have known that the probability of picking one sample is 1/3
So the probability of picking a incorrect sample is 2/3
so for picking m samples from the array, the probability of it is $\left(\frac{2}{3}\right)^m$

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. Solution to part (c) goes here.
The expected worst-case running time is O(1)
For the reason that the runtime of this algorithm is O(1+l(x))
l(x) is the time that $A[i] \mod 3 \neq 0$, and number of times that we sample $i \in \{1,\ldots,n\}$
but for the expected worst-case, we only need to test A[1] for once, and we find that A[1] mod 3 = 0, then we can just output 1
As a result, the expected worst-case of runtime is O(1)

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. Solution to part (d) goes here.
The worst-case runtime of it is O(n), and the expected worst-case running time is O(1)
For the worst-case runtime part
we run the for loop for n times until j = n, but we still do not find one i that makes A[i] mod 3 = 0, so we spend O(n) time here and we move to the next part
Then, we go over the array A to find an index i makes A[i] mod3 = 0, the runtime here again is also O(n)
Total runtime for worst case:
O(n) + O(n) = O(n)
For the expected worst-case part
It is the same as the problem c
which means that for the first random sample i, we find that A[i] mod 3 =0, so the runtime would be O(1)

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. Solution to part (a) goes here.
Algorithm:
We create a new array B of size n, and we make the number of B[i] = A[i] + B[i-1], if i =1, B[i] = A[i], the runtime for this should be O(n)
so B[i] contains a number that is the sum of all the numbers from A[1] to A[i]
After creating the array, we use counting sort, and we create a new array C that contains m numbers (the values of numbers in B are from 1 to m)

For i= 1 to n: increase C[B[i]] by one, the runtime here should be O(n) as well
Then search through the array C from C[1] to C[m], if there is a number j that makes $C[j] \geq 2$, output yes, if there is not such a number j, then output no. The runtime here should be O(m)
Runtime:
The runtime of this algorithm is O(n) + O(n) + O(m) = O(n)
Proof of correctness:
we know the correctness of counting sort and search which are mentioned in the lecture.
So we just need to prove for the part of create array B
Basecase:
for n = 1, we just have one element in B, so we just search if B[1]=0. If B[1] = 0, output yes. Otherwise, output no
Induction:
we know it is correct for n = k, and we want to prove it for n=k+1
We find that B[k+1] = B[k]+ A[k+1], so the induction is proved

---

(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. Solution to part (b) goes here.
Algorithm:
Same as the part(a), we create a new array of size n, and we make the number of B[i] = A[i] + B[i-1], if i =1, B[i] = A[i], the runtime for this should be O(n)
so B[i] contains a number that is the sum of all the numbers from A[1] to A[i]
After creating this array, we create a array T that contains m numbers (the values of numbers in B are from 1 to m)
We pick a hash function h(x) = x, which map all the numbers in B to numbers 1,...m which are indices of our hash table T
When there is a collision during the function, break. Then, output yes
If there is no collision during the function until the end of the function, output no
Runtime:
The runtime of it is again O(n)
Proof of correctness:
we have known the correctness of hash table, and we have proved the correctness of creating array B in part(a)
So the algorithm is proved

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. Solution to problem 4 goes here.
Specification:
For every $1 \leq i \leq n$, we define:
SNC(i): We get a number n, and n is the number of price we need to pay by using coins with values 5,9,13, and we create an array table [ ] to represent the minimum number of coins required
Solution:
SNC(i) =
not possible (when $n < 5$)
(when $n \geq 5$): condition 1: not possible
condition 2: min (table [i], table [i - 5] +1, table [i - 9] + 1, table [i - 13] +1)
We use memoization to avoid the recomputation of the same subproblems, that is why we build the array table [ ] with undefined in bottom-up manner, and we build an array coins[ ] to store the value of the coins in the array
we define a recursive function SNC (i):
for (int a = 1;$i <= i$; a++)
for (int j = 0; $j < 3$; j++)
if ($coins\,[j] \leq a$), int m = table[a - coins[j]]
if (the value not return not possible and $m + 1 < table[i]$), table[i] = m + 1
return table [i]
Runtime analysis:
We have two for loops in the function, so the runtime of the function is O(3n) = O(n)
Proof of correctness:
basecase:
when n ¡5, there is no coin can pay the money, so we output not possible
Induction:
we used memoization and induction here, so the proof of correctness just follows the recursive formula

---

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 $O(\log n)$ 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 $n \geq 1$:

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

Solution. Basease:
n =1, $F_0 = 0, F_1 = 1, F_2 = 1$
We know that it is true.

Induction:
We assume that we know the correctness of n = k, and we want to prove the correctness of n = k+1
For n = k, we have matrix:

$$\begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}.$$

For n = k+1, we have matrix:

$$\begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}.$$

we know that

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

beacause we know that $F_k + F_{k+1} = F_{k+2}$
The induction is proved

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