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

Homework #2 Solution

February 23, 2022

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 index i such that $rank(B[i]) = m/4$ and has worst-case runtime of $O(n)$[1].

Example: if $B = [1, 7, 6, 3, 13, 4, 5, 11]$, then **magic-pivot**$(B)$ will return index 4 as rank of $B[4] = 3$ is 2 which is $m/4$ in this case (rank of $B[4] = 3$ is 2 since there is only one element smaller than 3 in B).

(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. A complete solution has three steps, algorithm, proof of correctness, and runtime analysis.

Algorithm: Recall that in quick sort, we pick the pivot p as any arbitrary index of the array A. In our modification, the only change is that we pick p as the output of **magic-pivot**; formally:

modified-quick-sort$(A[1:n])$:

(a) If $n = 0$ or $n = 1$, return A.
(b) Let $p =$ **magic-pivot**(A).
(c) Run **partition**(A, p) and let q be the index of the correct position of pivot.
(d) Run **modified-quick-sort**$(A[1:q-1])$ and **modified-quick-sort**$(A[q+1:n])$.

Proof of correctness: By the correctness of **magic-pivot**, we will always be able to find an index p. Since original quick-sort works with any arbitrary choice of index p as pivot, the **modified-quick-sort** algorithm works correctly as well. (In fact, the entire point of using **magic-pivot** is to speedup quick-sort with minimal connection to its proof of correctness.)

Runtime analysis: Let $T(n)$ be the function for the worst-case runtime of the algorithm on inputs of length n. We claim that

$$T(n) \leq T(n/4) + T(3n/4) + O(n)$$

Let us replace $O(n)$ with $C \cdot n$ for some integer constant $C > 0$. The recursion tree for this recurrence is as follows:

- The root has a value of $C \cdot n$, and two child-nodes, one on a subproblem of size $n/4$, and another on a subproblem of size $3n/4$.
- The child-node of root for subproblem $n/4$ has value $C \cdot n/4$ and the subproblem $3n/4$ has value $C \cdot 3n/4$. Thus, the total value of the next level is also $C \cdot n = C \cdot n/4 + C \cdot 3n/4$.
- Continuing this way, each level has a total value of $C \cdot n$. Moreover, the total number of levels in the tree is $\log_{4/3} n$ (on the branch corresponding to 3/4-subproblems) which is $O(\log n)$.

Combining the above implies that the total value of the tree is $T(n) = O(n \log n)$ (since C is a constant).

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

(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. A complete solution has three steps, algorithm, proof of correctness, and runtime analysis.

Algorithm: The algorithm is as follows:

find-rank$(A[1 : n], r)$:

(a) If $n = 1$, return $A[1]$.
(b) Let $p =$ **magic-pivot**(A).
(c) Run **partition**(A, p) and let q be the index of the correct position of pivot.
(d) If $q = r$, return $A[q]$.
(e) Else, if $q > r$, return **find-rank**$(A[1 : q - 1], r)$; otherwise, return **find-rank**$(A[q + 1 : n], r - q)$ (note the change in the value of second argument).

Proof of Correctness: Proof is by induction: our hypothesis is that **find-rank**(A, r) outputs the correct answer for any choice of n and $1 \leq r \leq n$.

The base case is true when $n = 1$, since in this case $r = 1$ and the element of rank 1 is $A[1]$.

For the induction step, suppose this is true for all choices of $n \leq i + 1$ and we prove it for $n = i + 1$. By the correctness of **magic-pivot** and **partition**, we know that q is the correct position of $A[q]$ in the sorted array after the partitioning step; in other words, rank of $A[q]$ is q.

So if $q = r$, outputting $A[q] = A[r]$ is the correct answer.

If $q > r$, this means that the element with rank r belongs to the sub-array $A[1 : q - 1]$ as these are the elements smaller than $A[q]$ and since $r < q$, $A[r] < A[q]$ also by definition of rank. Thus, by induction hypothesis, **find-rank**$(A[1 : q - 1], r)$ finds the element of rank r in $A[1 : q - 1]$ which is also the element of rank r in A, making the answer correct.

Finally, if $q < r$, the element of rank r belongs to $A[q + 1 : n]$. Note however since we are removing q elements with value smaller than $A[r]$ from consideration, when looking at $A[q + 1 : n]$, the element of rank r in A will have rank $q - r$ in $A[q + 1 : n]$. By induction hypothesis, **find-rank**$(A[q + 1 : n], q - r)$ will find this element, finalizing the proof.

Runtime analysis: Define $T(n)$ as the worst-case runtime of **find-rank** on any array of length n (and for any choice of r). We have

$$T(n) = \max\{T(n/4), T(3n/4)\} + O(n)$$

because we only recurse on one subproblem and $q = n/4$ (unlike in part (a) which we recursed on both subproblems). Moreover, since $T(n)$ is a monotone function of n (runtime of algorithm on a larger input can only become larger), we have $T(n) \leq T(3n/4) + O(n)$. This means (by replacing $O(n)$ with $C \cdot n$ for some constant $C > 0$),

$$T(n) \leq T(3n/4) + C \cdot n \leq T(9n/16) + C \cdot (n + 3n/4) \leq C \cdot n \cdot \sum_{i=0}^{+\infty} (3/4)^i = O(n),$$

as the sum of a geometric series with ratio less than 1 converges to $O(1)$. As such, the runtime of **find-rank** is $O(n)$ as desired.

[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 $A[i] \mod 8 \in \{1,2\}$, i.e., the reminder of dividing $A[i]$ by 8 is either 1 or 2. For simplicity, in the following, we assume that n itself is a multiple of 8 and is at least 8 (so a correct answer always exist).

For instance, if $n = 8$ and the array is $A = [8,7,2,5,4,6,3,1]$, we want to output either of indices 3 or 8.

(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., $A[i] \mod 8 \in \{1,2\}$? **(5 points)**

Solution. The correct answer is $1/4$.

There are exactly $2n/8 = n/4$ numbers in $\{1,\ldots,n\}$ such that $A[i] \mod 8 \in \{1,2\}$ (as n itself is a multiple of 8 there is no corner case). Since we are picking i uniformly at random, the probability that i is any of these numbers is exactly $(n/4)/n = 1/4$.

(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 correct answer is $(3/4)^m$.

By part (a), the probability that each index is *not* correct is $1 - 1/4 = 3/4$. Since we are sampling each index *independently* (as it is with repetition), the probability that no index is correct among m trials is $(3/4)^m$.

Now, consider the following simple algorithm for this problem:

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

- Let $i = 1$. While $A[i] \mod 8 \notin \{1,2\}$, 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 worse-case **expected** running time of **Find-Index-1**$(A[1:n])$? Remember to prove your answer formally. **(7 points)**

Solution. Define a random variable $X \in [1:+\infty]$ where $X = j$ if the number of times we run the while-loop is j (it is a random variable depending on the randomness of the algorithm). Each run of the algorithm takes $O(X)$ time (but this is a random variable and so we need to turn it into a formula); thus the expected worst-case runtime of the algorithm is $O(\mathbf{E}[X])$. So, we only need to compute $\mathbf{E}[X]$.

We have,

$$\begin{aligned}
\Pr(X = j) &= \Pr(\text{first } j-1 \text{ trials fail and } j\text{-th trial succeeds}) && \text{(by the definition of while-loop)} \\
&= \Pr(\text{first } j-1 \text{ trials fail}) \cdot \Pr(j\text{-th trial succeeds}) \\
& && \text{(by independence of trials in different iterations)} \\
&= (3/4)^{j-1} \cdot (1/4) && \text{(by part (b) and part (a), respectively)} \\
&< (3/4)^j.
\end{aligned}$$

As such, by the definition of expectation,

$$\mathbf{E}[X] = \sum_{j=1}^{\infty} \Pr(X = j) \cdot j \leq \sum_{j=1}^{\infty} (3/4)^j \cdot j = 12,$$

as the series converges to 12. So $O(\mathbf{E}[X]) = O(1)$, meaning that the worst-case expected runtime of the algorithm 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 modification to 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 $A[i] \mod 8 \in \{1, 2\}$, 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 $A[i] \mod 8 \in \{1, 2\}$ 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 worst-case **expected** running time? Remember to prove your answer formally.

(8 points)

Solution. The worst-case runtime of the new algorithm happens when we finish the for-loop without success and then do a linear search over the array; both of these takes $\Theta(n)$ time so the worst-case runtime is $\Theta(n)$.

For the worst-case expected runtime, let us define two variables. We use $X \in \{1, \ldots, n, n+1\}$ to denote the number of iterations of the first for-loop where $X = n + 1$ means that the for-loop failed. So, when $X \leq n$, the runtime of the algorithm is $O(X)$ and when $X = n + 1$, the runtime of the algorithm is $O(n)$ (for the first for-loop) plus another $O(n)$ (for the second for-loop); either way, the runtime of the algorithm is $O(X)$. We thus need to compute expected value of X to get the expected worst-case runtime of the algorithm.

$$\mathbf{E}[X] = \sum_{j=1}^{n+1} \Pr(X = j) \cdot j \leq \sum_{j=1}^{\infty} (3/4)^j \cdot j = 12,$$

where the calculations is exactly as in part (a). Thus, in this case also, the worst-case expected runtime of the algorithm is still $O(1)$.

Problem 3. Given an array $A[1:n]$ of n positive integers (possibly with repetitions), your goal is to find whether there is a sub-array $A[l:r]$ such that

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

Example. Given $A = [13, 1, 2, 3, 4, 7, 2, 3, 8, 9]$ for $n = 10$, the answer is *Yes* since elements in $A[2 : 5]$ add up to $n = 10$. On the other hand, if the input array is $A = [3, 2, 6, 8, 20, 2, 4]$ for $n = 7$, the answer is *No* since no sub-array of A adds up to $n = 7$.

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

(a) Design an algorithm for this problem with worst-case runtime of $O(n)$. **(15 points)**

Solution. As some of you pointed out on Piazza and office hours, it turns out that this problem has a simpler solution than what we intended that works for both part (a) and (b). So, in the following, we present that solution instead (but will also include a solution for part (b) using randomization in case someone is interested).

A complete solution has three steps, algorithm, proof of correctness, and runtime analysis.

Algorithm: The algorithm works by considering two pointers i and j forming a "sliding window" with the intuition that i will be "searching" for the index l and j will be "searching" for r so that $\sum_{k=l}^{r} A[k] = n$. We now formalize the algorithm:

(a) Let $SUM = A[1]$, $i = 1$, and $j = 1$.

(b) While $i \leq n$ AND $j \leq n$:

   i. If $SUM = n$, return *Yes* and terminate.

   ii. If $SUM < n$, update $j \leftarrow j + 1$ and $SUM \leftarrow SUM + A[j]$

   iii. If $SUM > n$, update $SUM \leftarrow SUM - A[i]$ and $i \leftarrow i + 1$. If $i > j$, update $j \leftarrow i$.

(c) Return *No*.

Proof of Correctness: For any value of i and j during the algorithm, we define $SUM(i, j)$ as the value of variable SUM when the indices i and j have that specific value (for instance, $SUM(1, 4)$ means value of SUM when $i = 1$ and $j = 4$ during the algorithm). We first claim that

$$SUM(i, j) = \sum_{k=i}^{j} A[k],$$

namely, $SUM(i, j)$ calculates the sum of numbers in A that are between indices i and j. This is true by induction because $SUM(1, 1) = A[1]$ as set at the beginning of the algorithm and whenever we increase i, we subtract the previous value from SUM and whenever we increase j, we add the new value to SUM.

Now, we would like to prove that the algorithm returns *Yes* if and only if there are some $l \leq r$ such that $\sum_{k=l}^{r} A[k] = n$.

Firstly, if the algorithm returns Yes, it means that it has found indices i and j such that $SUM(i, j) = n$, which by the equation above means that $\sum_{k=i}^{j} A[k] = n$. Thus, there is some choice of $l = i$ and $r = j$ that solves the problem in this case and our answer was correct.

We now need to prove that if there are some indices $l \leq r$ such that $\sum_{k=l}^{r} A[k] = n$, the algorithm also will output Yes at some point. To do this, we have the following:

- As long as $i \leq l$, index j cannot become larger than r. This is because for j to go larger than r, we should have $SUM(i, r) < n$ according to the algorithm. But, we have,

$$SUM(i, r) = \sum_{k=i}^{r} A[k] \geq \sum_{k=l}^{r} A[k] = n,$$

  where the inequality is because all the elements in the array are positive and since $i \leq l$. This means that $SUM(i, r) \geq n$ whenever $i \leq l$ and thus j will never increase beyond r in this case.

- As long as $l \leq j \leq r$, index i cannot become larger than l. This is because for i to go larger than l, we should have $SUM(l, j) > n$ according to the algorithm. But, we have,

$$SUM(l, j) = \sum_{k=l}^{j} A[k] \leq \sum_{k=l}^{r} A[k] = n,$$

where in the inequality is because all the elements in the array are positive and since $l \leq j \leq r$. This means that $SUM(l, j) \leq n$ whenever $l \leq j \leq r$ and thus i will never increase beyond l in this case.

Combining the above parts with the fact that in every iteration of while-loop, either i increases or j increases, we have that either the algorithm already returns Yes earlier, or at some point we get $i = l$ and $j = r$. But in that point, we have $SUM(i, j) = SUM(l, r) = n$, thus the algorithm will output Yes then.

This concludes the proof of correctness of the algorithm.

Runtime Analysis: Every iteration of the while-loop increases either i or j and neither can increase more than n times. Thus, we only have $O(n)$ iterations. Moreover, each iteration takes $O(1)$ time, so the total runtime is $O(n)$.

Remark: At no point in this solution, we really used the fact that $\sum_{k=l}^{r} A[k]$ has to be n, as opposed to anything different, say n^2. Thus, the same algorithm also works as is by just changing this criteria slightly inside it for part (b) also. Note that you can always use a deterministic algorithm in place of a randomized one (but not the other way around).

(b) Now suppose our goal is to instead find whether there is a sub-array $A[l, r]$ such that

$$\sum_{i=l}^{r} A[i] = n^2.$$

Design a <u>randomized</u> algorithm for this case with worst-case <u>expected</u> runtime of $O(n)$. **(10 points)**

Solution. A complete solution has three steps, algorithm, proof of correctness, and runtime analysis.

Algorithm: We start by constructing a prefix sum array B as follows.

(a) $B[0] = 0$, $B[1] = A[1]$.
(b) For $i = 2$ to n, $B[i] = B[i-1] + A[i]$

This way $B[i] = \sum_{j=1}^{i} A[j]$. Thus, for any $i \leq j$, we have $B[j] - B[i-1] = \sum_{k=i}^{j} A[k]$ (where we assume $B[0] = 0$). We now design our algorithm for this part.

(a) Create the prefix sum array B as above.
(b) Pick a near-universal hash family and construct a hash table T of size $m = n$ using this hash function and the chaining method for handling collisions.
(c) For $j = 1$ to n,
    i. If T.search($B[j] - n^2$) is true, return *Yes*.
    ii. Else, insert $B[j]$ to the hash table T.
(d) Return *No*.

Proof of correctness: Suppose first that there are indices l and r such that $\sum_{k=l}^{r} A[k] = n^2$. In that case, when $j = r$ in our algorithm, we have already inserted $B[l-1]$ to the hash table and so when searching for $B[j] - n^2$, we are searching for

$$B[j] - n^2 = (\sum_{k=1}^{j} A[k]) - n^2 = (\sum_{k=1}^{l-1} A[k] + \sum_{k=l}^{j} A[k]) - n^2 = \sum_{k=1}^{l-1} A[k] = B[l-1],$$

which is a number already inside the hash table. Thus, the algorithm finds this number also and outputs Yes which is the correct answer.

Now suppose there are no indices l and r such that $\sum_{k=l}^{r} A[k] = n^2$. If the algorithm outputs Yes in this case, it means that it has found two indices $i \leq j$ such that $B[j] - B[i] = n^2$. But that means $\sum_{k=i+1}^{j} A[k] = n^2$ as argued earlier which is a contradiction (if we take $l = i+1$ and $r = j$, we get the contradiction). So, in this case, the algorithm always outputs No.

Runtime Analysis: Creating the prefix sum array and the hash table all take deterministically $O(n)$ time. Each search also in *expectation* takes $O(1 + n/m) = O(1)$ time as we are using a randomized near-universal hash functions on a table of size $m = n$ and we insert at most n elements in the hash table. By linearity of expectation, the total expected runtime of the for-loop is also $O(n)$. Thus, the worst-case expected runtime of the algorithm is $O(n)$.

Problem 4. You are given an array of characters $s[1:n]$ such that each $s[i]$ is a small case letter from the English alphabet. You are also provided with a black-box algorithm *dict* that given two indices $i, j \in [n]$, $dict(i,j)$ returns whether the sub-array $s[i:j]$ in array s forms a valid word in English or not in $O(1)$ time. (Note that this algorithm is provided to you and you do not need to implement it in any way).

Design and analyze a dynamic programming algorithm to find whether the given array s can be partitioned into a sequence of valid words in English. The runtime of your algorithm should be $O(n^2)$.

Example: Input Array: $s = [maytheforcebewithyou]$.

Assuming the algorithm *dict* returns that $may, the, force, be, with$ and you are valid words (this means that for instance, for may we have $dict(1,3) =$ True), this array can be partitioned into a sequence of valid words.

Solution.

Specification: For an integer i, $1 \leq i \leq n$, define:

- $dp(i)$ to be 1 if the string $s[1:i]$ can be partitioned into a sequence of valid words, and 0 otherwise.

The final answer to the problem will be $dp(n)$.

Solution: We can calculate $dp(i)$ recursively as follows:

$$dp(i) = \begin{cases} 1 & \text{if } dict(1,i) \text{ is True} \\ 1 & \text{if } dp(j) = 1 \text{ and } dict(j+1,i) \text{ is True for some } j < i \\ 0 & \text{otherwise} \end{cases}$$

We will now prove the correctness of this solution.

For the base case, we check directly if the first letter of the string is valid and set $dp(1)$ accordingly. We now consider the other two cases.

For one direction, if $s[1:i]$ is a valid sequence of words, either $dict(1,i)$ returns true, or there is some $j \leq i$ such that $s[1:j]$ can be partitioned into a valid sequence and $dict(j+1,i)$ returns True. In the first case, the first line of $dp(i)$ returns True, and in the second case, we have both $dp(j)$=True and $dict(j+1,i)$=True, so then the second line returns True.

For the other direction, we want to show that if $dp(i)$=True, then $s[1:i]$ can be partitioned into a valid sequence. If the algorithm sets $dp(i) = 1$, then either $dict(1,i)$ returns True, or it finds an $j < i$ such that $dp(j)$=True and $dict(j+1,i)$ returns True. Thus, $s[1:j]$ is a sequence of words and this combined with the word $s[j+1:i]$ forms a valid partition for $s[1:i]$ also.

Dynamic programming algorithm: We will do a bottom-up dynamic programming for this problem.

1. Create array $D[1:n]$ with 0 as its entries.
2. Let $D[1] = 1$ if $dict(1,1)$ returns True, $D[1] = 0$ otherwise.
3. For $j = 2$ to n,
   (a) $D[j] = 1$ if $dict(1,j)$ returns True.
   (b) Check if there exists $k < j$ such that $D[k] = 1$ and $dict(k+1,j)$ returns True by going over each $k < j$ in a for-loop. If so, set $D[j] = 1$.
4. Return s is a valid string if $D[n] = 1$, and not valid otherwise.

Runtime Analysis: To find the value at each index in the array D we take $O(n)$ time and the size of the array is n. So the runtime is $O(n^2)$.