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

# Homework #2 Solutions

February 25, 2021

**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.** 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 index of output of **magic-pivot**; formally:

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

(a) If $n = 0$ or $n = 1$, return $A$.
(b) Let $b =$ **magic-pivot**$(A)$. Iterate over the array $A$ and find the index $p$ where $A[p] = b$.
(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

$$T(n) \leq \max_{\frac{n}{3} \leq q \leq \frac{2n}{3}} \left(T(q-1) + T(n-q)\right) + O(n);$$

this is because by **magic-pivot** will return an element whose correct position in the array, which is the index $q$, will be between $n/3$ and $2n/3$; the rest is the same as quick-sort.

For any choice of $\alpha \in [\frac{1}{3}, \frac{2}{3}]$, consider the function $S_\alpha(n) = S_\alpha(\alpha \cdot n) + S_\alpha((1-\alpha) \cdot n) + O(n)$. By the definition of $T(n)$, we have $T(n) = O(\max_\alpha S_\alpha(n))$.

If we write the recursion tree for $S_\alpha$, at every level the work done by the algorithm will be $C \cdot n$ (for some constant $C > 0$), and there will be at most $O(\max\left\{\log_{1/\alpha}(n), \log_{1/(1-\alpha)}(n)\right\})$ levels in the tree (this is similar to several other recursion trees we have written and so we omit it here.) This means that $S_\alpha(n) = O(n \cdot \max\left\{\log_{1/\alpha}(n), \log_{1/(1-\alpha)}(n)\right\})$. Finally, since $\alpha \in [\frac{1}{3}, \frac{2}{3}]$, $\max\left\{\log_{1/\alpha}(n), \log_{1/(1-\alpha)}(n)\right\} \leq \log_{(3/2)}(n) = \Theta(\log n)$ over all choices of $\alpha$.

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

Finally, this means that the runtime of the algorithm is $T(n) = O(n \log n)$, as desired.

*Note:* The above runtime analysis was a very formal way of proving the upper bound on $T(n)$ without making any assumptions. If, in your homework, you have simply stated that the "worst-case" of the recursion is when the split is most unbalanced, i.e., $T(n) \leq T(n/3) + T(2n/3) + O(n)$, which implies $T(n) = O(n \log n)$, you will receive the full grade. The goal of showing the proof in full generality was to also prove the unbalanced split case is indeed the worst case.

---

(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 $b =$ **magic-pivot**$(A)$. Iterate over the array $A$ and find the index $p$ where $A[p] = b$.

(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

$$T(n) \leq \max_{\frac{n}{3} \leq q \leq \frac{2n}{3}} T(q) + O(n);$$

[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 $O(n)$.

this is by exactly the same argument as in part (a). Given that $T(n)$ is a monotone function of $n$ (runtime of algorithm on a larger input can only become larger), we have $T(n) \leq T(\frac{2n}{3}) + O(n)$. This means (by replacing $O(n)$ with $C \cdot n$ for some constant $C > 0$),

$$T(n) \leq T(2n/3) + C \cdot n \leq T(4n/9) + C \cdot (n + 2n/3) \leq C \cdot n \cdot \sum_{i=0}^{+\infty} (2/3)^i = O(n),$$

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.

---

**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.** There are exactly $n/3$ numbers in $\{1, \ldots, n\}$ that are multiples of 3 (as $n$ itself is a multiple of 3 there is no corner case). Since we are picking $i$ uniformly at random, the probability that $i$ is any of these numbers is exactly $(n/3)/n = 1/3$. So the answer 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.** By part $(a)$, the probability that each index is *not* correct is $1 - 1/3 = 2/3$. Since we are sampling each index *independently* (as it is with repetition), the probability that no index is correct among $m$ trials is $(2/3)^m$.

---

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.** 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,

$$\begin{aligned}
\Pr(X = j) &= \Pr(\text{first } j-1 \text{ trials fail and } j\text{-th trial succeeds}) && \text{(by the definition of while-loop)} \\
&= \Pr(\text{first } j-1 \text{ trials fail}) \cdot \Pr(j\text{-th trial succeeds}) \\
& && \text{(by independence of trials in different iterations)} \\
&= (2/3)^{j-1} \cdot (1/3) && \text{(by part (b) and part (a), respectively)} \\
&< (2/3)^j.
\end{aligned}$$

As such, by the definition of expectation,

$$\mathbf{E}[X] = \sum_{j=1}^{\infty} \Pr(X = j) \cdot j \leq \sum_{j=1}^{\infty} (2/3)^{j-1} \cdot j = 9,$$

as the series converges to 9. So $O(\mathbf{E}[X]) = O(1)$, meaning that the expected worst-case 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 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.** 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 expected worst-case 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.

$$\mathbf{E}[X] = \sum_{j=1}^{n+1} \Pr(X = j) \cdot j \leq \sum_{j=1}^{\infty} (2/3)^{j-1} \cdot j = 9,$$

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

---

**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.** 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 $B[i] = \sum_{j=1}^{i} A[j]$. We know each element in the array is at most 5 and at least $-5$, so we have that every $-5n \leq B[i] \leq 5n$.

We now design our algorithm for this part.

(a) Create the prefix sum array $B$ as above.
(b) Initialize array $C$ of size $10n + 1$ to be zero.
(c) For $i = 0$ to $n$, $C[B[i] + 5n + 1] = C[B[i] + 5n + 1] + 1$.
(d) For $i = 1$ to $10n + 1$, if $C[i] > 1$, return *Yes*.
(e) Return *No*.

*Proof of Correctness:* The fact that for each $i$, $B[i] = \sum_{j=1}^{i} A[j]$, can be proven using induction on $i$ (it is actually so simple that you do not need to provide a proof for it). The base case is when $i = 1$, $B[1] = A[1]$ by construction of $B$. We assume for some $k$ $B[k] = \sum_{j=1}^{k} A[j]$. For $k + 1$, we know $B[k+1] = B[k] + A[k+1] = \sum_{j=1}^{k+1} A[j]$ by the induction hypothesis.

Now note that, by the hint, we have $\sum_{i=l}^{r} A[i] = 0$ if and only if $\sum_{i=1}^{l-1} A[i] = \sum_{i=1}^{r} A[i]$ or in other words, $B[l-1] = B[r]$. Thus, the algorithm only needs to check if there are two indices $1 \leq i < j \leq n$, where $B[i] = B[j]$. We prove the second part of the algorithm does that.

In the algorithm $C[i]$ will contain the number of elements in array $B$ which have value $i - 5n - 1$. This follows from proof of correctness of counting sort. If the frequency of any element is greater than 1, this means that two different indices in $B$ have the same value and thus the answer should be *Yes*; otherwise the answer is *No*: this is exactly what is done by the algorithm, proving the correctness.

*Runtime Analysis:* Creating the prefix sum array takes $O(n)$ time; creating $C$ and running the search takes two for-loop each with $O(n)$ iteration, again taking $O(n)$ time. So the runtime 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.** A complete solution has three steps, algorithm, proof of correctness, and runtime analysis.

*Algorithm:* We again create the prefix sum array $B$ as follows and again search if there are two indices $i$ and $j$ where $B[i] = B[j]$. The only difference is that since we do not have a bound on the range of the elements in $A$, we use hashing to find if there are duplicates in $B$ without (implicitly) sorting $B$.

(a) Create the prefix sum array $B$ as before.

(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 $i = 1$ to $n$,

  i. If $T$.search($B[i]$) is true, return *Yes*.

  ii. Else, insert $B[i]$ to the hash table $T$.

(d) Return *No*.

*Proof of correctness:* Suppose first that array $B$ has a duplicate and $k$ is the first index where there exist $j < k$ such that $B[j] = B[k]$. In this case, before inserting $B[k]$, the value $B[j]$ already exists in $T$ (as we have inserted all previous entries of $B$ to $T$), and thus we find the duplicate and return *Yes* correctly.

On the other hand, if array $B$ has no duplicate, we will never find any $B[i]$ inside the table before inserting it and after the for-loop, we return *No* correctly.

*Runtime Analysis:* Creating the prefix sum array takes $O(n)$ time 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 expected worst-case runtime of the algorithm is $O(n)$.

---

**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.** We will apply the two steps for solving a dynamic programming problem: *Specification* and *Solution*. Only then, we turn our recursive formula into an algorithm (using memoization) and analyze the runtime.

*Specification:*

- For any integers $1 \leq i$, define:
  $K(i)$: the minimum number of coins required to have a total value of $i$; if it is not possible to purchase the item using any combination of coins, we *define* $K(i) = +\infty$.

To return the answer, we simply need to return $K(n)$.

*Solution:*

$$K(i) = \begin{cases} +\infty & \text{if } i < 0 \\ 0 & \text{if } i = 0 \\ 1 + \min\{K(i-5), K(i-9), K(i-13)\} & \text{otherwise} \end{cases}$$

We now prove the correctness of this solution.

By definition, there is no way for us to purchase an item of negative value and so $K(i) = +\infty$ for $i < 0$ is correct (just by definition).

Let us consider thee other base case of $i = 0$. In this case, by the specification, $K(0) = 0$. This is obviously correct, because if an item costs 0 (i.e., is free), we would need no coins to purchase it.

Now let us consider larger values of $i$. Because we know that we can only use the coins $5, 9$, and 13, we know that every combination of coins that can purchase an item of value $i > 0$ must contain at least one of those three coins. If we pick coin 5 to begin, then we end up using one coin and have to purchase the remaining amount which is $i-5$; thus, in this case, the number of coins will be $1+K(i-5)$ by the definition of $K(i-5)$. Similarly, if we decide to use coin 9 or 13, then we have to pay $1+K(i-9)$ and $1+K(i-13)$, respectively. As our goal is to use the minimum number of coins, taking the minimum of these three possible options, gives us the correct answer.

*Dynamic Programming Algorithm (Memoization):*

We will store a an array $D[1:n]$ initialized with 'undefined' everywhere.

MemCoin($i$):

1. if $i < 0$: return $+\infty$;
2. if $i = 0$: return 0
3. if $D[i] \neq$ 'undefined': return $D[i]$
4. Otherwise, let $D[i] = 1 + \min\{\text{MemCoin}(i-5), \text{MemCoin}(i-9), \text{MemCoin}(i-13)\}$
5. return $D[i]$

This concludes the algorithm. The correctness follows from the correctness of recursive formula.

*Runtime:* Our memoization algorithm runs in $O(n)$ time, as there are $n$ subproblems and each subproblem, ignoring the time it takes to do the inner recursions, takes $O(1)$ time.

---