CIT 596 Online **Summer 2020**

Module 3–4

Name: Solutions

Questions requiring written answers.

1. Design a divide-and-conquer algorithm for the problem of finding the subinterval of maximum sum that you encountered in module 1. You should analyze the running time of your algorithm and argue that it is correct. [**Hint:** If you divide the input array into two halves, the best subinterval can be contained in the left half, contained in the right half, or straddle the two halves. Try and find an $O(n)$ algorithm for finding the best subinterval that straddles the two halves, and use this in your recurrence relation for analyzing the running time.]

   Solution:

   Algorithm: If the array has 0 elements, then return the empty interval. If the array has 1 element, return that singleton interval if it is greater than 0 (return empty interval otherwise). Otherwise, divide the array in (roughly) half and consider 3 intervals: (i) the maximum sum subinterval of the left half (computed by recursively using this algorithm on the left half), (ii) the maximum sum subinterval on the right half (computed recursively), and (iii) the maximum subinterval that straddles the two halves. In order to compute (iii), start at the last element of the left half, and iteratively add the next element on the left while keeping track of the (suffix) subinterval with maximum sum; similarly, start with the first element on the right half and iteratively add the next element on the right while keeping track of the (prefix) subinterval with maximum sum. Then combine the maximum-sum suffix subinterval on the left with the maximum-sum prefix subinterval on the right half to get (iii). Return the subinterval whose sum is maximum out of (i), (ii), and (iii).

   Correctness
   Base Case: $n = 0$ and $n = 1$. Algorithm clearly returns the maximum-sum subinterval since in both cases, it considers all possible subintervals.
   Induction Hypothesis: Let k be an integer ≥ 2. For all arrays of size j such that $2 \leq j \leq k$, assume that the algorithm correctly returns the subinterval with maximum sum.
   Induction Step: Consider an array with size $k + 1$. As per the hint, the best subinterval must either be contained in the left half, right half, or must straddle the two halves. By the strong IH, the first two cases (i) and (ii) are both correctly computed. For (iii), note that subintervals straddling the two halves must consist of a suffix of the first half combined with a prefix of the second; our algorithm considers only the best prefix and the best suffix, so it correctly computes (iii). Thus, the algorithm correctly computes (i), (ii), and (iii), and then correctly returns the best subinterval of those three.

   Running Time: Our base cases clearly run in $O(1)$ time. For the general case, note that we use two recursive calls on arrays with half the size of the input, and additionally consider all prefixes and suffixes of the two halves, respectively. Thus, we get the recurrence:

   $$T(n) = \begin{cases} O(1) & n = 0, n = 1 \\ 2T(n/2) + O(n) & n \geq 2 \end{cases}$$

   By the Master Theorem, the entire algorithm is $O(n \log n)$.

2. Let A be a sorted array of n integers. Given a value d, we want to know if there are two indices i, j such that $A[j] - A[i] = d$. Design an efficient algorithm to solve this problem and analyze the running time of the algorithm.

   Solution:

Algorithm: Keep two indices lo and hi both starting at the first index. Repeat the following: if $A[hi] - A[lo] = d$, then stop and return lo and hi. If $A[hi] - A[lo] > d$, increment lo by 1. Otherwise, increment hi by 1. If it's no longer possible to increment an index (e.g. it becomes out of bounds), then return that there is no i and j such that $A[j] - A[i] = d$.

Correctness: If there don't exist two indices i and j such that $A[j] - A[i] = d$, then the algorithm will clearly be correct because it only returns the two indices if they actually exist (and the algorithm verifies that the difference is indeed d. If there do exist such indices, we will prove that the algorithm is sure to find them.

While the upper index is at some particular value hi, the lower index goes through a range of values $[a, b]$. (Note that a could be equal to b.) For each $i \in [a, b]$ the algorithm explicitly computes $A[hi]-A[i]$ and would have found if some pair of indices here gave a difference of d. We just need to argue that we didn't miss possible solutions where the lower index is less than a or greater than b. Since a was the first lower index considered when the upper index reached hi, consider when the lower index was incremented to a. Suppose it happened when the upper index was $hi' < hi$. Then we know $A[hi'] - A[a - 1] > d$, and hence $A[hi] - A[i] > d$ for all $i \leq a - 1$. Finally, when we increment the upper index from hi to $hi + 1$, the lower index is at b, and we know that $A[hi] - A[b] < d$. Thus $A[hi] - A[i] < d$ for any value of $i > b$. Thus we have proven that for each upper index hi, we consider some interval of lower indices, and can rule out any lower indices that do not lie in this interval. Thus our algorithm will correctly find a suitable index pair if one exists.

Running Time: In the worst case, both lo and hi traverse (almost) the entire array, incrementing one at a time. The algorithm runs in $O(n)$ time.

3. Prove that if a and n are not relatively prime, then we cannot find integer d such that $ak = 1 mod n$. (Recall that two numbers are not relatively prime exactly when they have a common factor $d > 1$.)

   Solution:

   Assume towards a contradiction that a has a multiplicative inverse modulo n. That is, $ak \equiv 1 \mod n$ for some integer k. Then $ak = 1 + nj$ for some integer j. Since a and n are not relatively prime, they have a common factor $d > 1$. Then we can write

   $$dx_1k = 1 + dx_2j$$

   for some integers x_1 and x_2, and thus

   $$d(x_1k - x_2j) = 1$$

   However since $d > 1$, $x_1k - x_2j < 1$ (otherwise their product would be greater than 1). Thus, $0 < x_1k - x_2j < 1$. This is impossible since $x_1k - x_2j$ must be an integer (all variables within are integers); contradiction.

4. Suppose you roll a fair die 10 times (all rolls are mutually independent). What is the probability that you roll a 2 in exactly 3 of the 10 rolls?

   Solution:

   There are $\binom{10}{3}$ ways of choosing 3 rolls out of the 10 rolls. Once we have chosen some set of 3 rolls, the probability that each of them yields a 2 is $(\frac{1}{6})^3$. The probability that none of the other 7 rolls yields a 2 is $(\frac{5}{6})^7$. Putting this all together, the probability of the described event happening is:

   $$\binom{10}{3}\left(\frac{1}{6}\right)^3\left(\frac{5}{6}\right)^7$$

5. You have a coin that has a probability p of coming up Heads. (Note that this may not be a fair coin, and p may be different from 1/2.) Suppose you toss this coin repeatedly until you observe 1 Head. What is the expected number of times you have to toss it?

Solution:

The probability that we succeed after 1 toss is p. The probability we get our first Head on the i^{th} toss is $(1-p)^{i-1}p$. This is because we must get tails on the first $i-1$ tosses and heads on the i^{th} and this is the probability of that happening. Recall that for a random variable X, the expectation $E[X]$ is defined as $\sum_v v\,\Pr[[]X = v]$ where v ranges over all possible values X can take. In our case, X is the number of coin tosses until the first Head, and X ranges from 1 to ∞. Thus

$$E[X] = \sum_{i=1}^{\infty}(1-p)^{i-1}pi$$

$$E[X] = p.1 + (1-p)p.2 + (1-p)^2p.3 + .. \tag{1}$$

Multiply by (1-p) on both the sides.

$$(1-p)E[X] = p(1-p) + 2p(1-p)^2 + 3p.(1-p)^3 + ... \tag{2}$$

Subtracting (2) from (1) we have:

$$p.E[X] = p + p(1-p) + p(1-p)^2 + p(1-p)^3 + ... \tag{3}$$

$$E[X] = 1 + (1-p) + (1-p)^2 + (1-p)^3 + ... \tag{4}$$

But, this sum of infinite geometric series with common ratio (1-p). Hence,

$$E[X] = 1/p \tag{5}$$

Alternative solution:
By formula for expectation for non-negative integers we know that:

$$E[X] = \sum_{i=1}^{\infty} Pr[X \geq i]$$

For our case:

$$E[X] = \sum_{i=1}^{\infty} Pr[\text{fail to get heads in i-1 flips}]$$

$$= \sum_{i=1}^{\infty}(1-p)^{i-1}$$

Changing the lower limit from i = 1 to i = 0

$$= \sum_{i=0}^{\infty}(1-p)^{i}$$

Using the formula of summation of infinite geometric series:

$$E[X] = \frac{1}{1-(1-p)} = \frac{1}{p}$$

6. **Coupon Collector Problem:** In many countries toys or baseball cards are packed in with certain kinds of candy you buy. As a concrete example of this, suppose you get a baseball card each time you buy a pack of bubblegum. Say there are n different cards and each time you buy bubblegum, you get

a random one of these cards. What is the expected number of bubblegums you have to buy before you have the complete set?

Solution:

Let X denote the random variable that returns the number of bubblegum packs that you must open before having all baseball cards. Let X_i be the random variable that returns the number of packs you must open before obtaining your ith distinct baseball card, after you have already obtained $i-1$ distinct baseball cards. Clearly,

$$X = \sum_{i=1}^{n} X_i$$

Note that each X_i is a random variable of the type we analyzed in the previous question, with parameter $p = 1 - \frac{i-1}{n}$ and so

$$\mathbf{E}[X_i] = \frac{1}{1 - \frac{i-1}{n}} = \frac{n}{n-i+1}$$

Then

$$\begin{aligned}
\mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathbf{E}[X_i] && \text{(linearity)} \\
&= \sum_{i=1}^{n} \frac{n}{n-i+1} \\
&= n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right) \\
&= n\sum_{i=1}^{n} \frac{1}{i}
\end{aligned}$$

If you recall, the sum in the last expression is a Harmonic series, which sums to close to $\ln n$, the natural log of n. Thus, the expected number of bubblegums we need to buy is $O(n \log n)$.

7. You are given an integer $r \in [1..n]$ and a sequence $\sigma = s_1, s_2, \ldots, s_n$ of n distinct elements in which elements are presented one at a time. When element s_i is presented, you can no longer access any of $s_1, \ldots, s_{i-1}$ unless your algorithm has stored them. You are asked to output the r^{th} smallest element in σ. Design an algorithm that can accomplish this using $O(r)$ space and $O(n)$ time.

   Solution:

   Algorithm: Create an array of size $2r$ and fill it up with the first $2r$ elements in σ. Select the rank-r element of the array, and do this in-place so that the array is partitioned by this rank-r element when this is done. Now replace the $r+1$ to $(2r)$th (inclusive) elements in the array with the next r elements in σ. Now repeat the previous procedure, finding the rank-r element in-place and replacing the $r+1$ to $(2r)$th elements of the array, until all elements have been added to the array. Output the rank-r element of the final array.

   Correctness: Note that every element (in particular, the rth smallest element in σ) is added to the array of size $2r$ at some point in the algorithm. Moreover, all elements with rank $\leq r$ (rank relative to all elements in σ) will never be removed from the array; this is because at every iteration of the procedure, the array is partitioned by the rank-r element (rank within the array), so any elements with rank $\leq r$ relative to all elements in σ will certainly also have rank $\leq r$ in the array (since the array's values are a subset of σ). Then at the end of the algorithm, the array's first r indices will contain the r smallest elements in σ, and so the algorithm correctly returns the rth smallest element in σ.

Running Time: Creating the array of size $2r$ takes $O(r)$ time, and it takes $O(r)$ time to select the rth smallest element. We observe that the procedure of replacing the highest r indices in the array and then selecting the rank-r element is repeated until we have considered all elements. This results in $O(n/r)$ iterations. Since each iteration takes $O(r)$ time, the total runtime is $O\left(\frac{n}{r} \cdot r\right) = O(n)$ time.

Space Analysis: The algorithm uses $2r = O(r)$ space to store the array. Each use of the linear-time selection algorithm may require $O(r)$ auxiliary space, but this can be re-used in each separate call of the selection algorithm. Thus, the algorithm uses a total of $O(r)$ space.