CS 6150: HW1 – Data structures, recurrences - Late Days Used: 3

Pranav Rajan, Collaborators: Adithya Badideymurali, Ryan Howell, Shubham Mazumder, Frost Office Hours, Yo Office Hours

Submission date: Monday, Sep 13, 2021 (11:59 PM)

This assignment has 6 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Basics	7	
Bubble sort basics	5	
Deletion in prefix trees	6	
Binary search and test pooling	10	
Recurrences, recurrences	16	
Dynamic arrays: is doubling important?	6	
Total:	50	

Note. *When asked to describe and analyze an algorithm, you need to first write the pseudocode, provide a running time analysis by going over all the steps (writing recurrences if necessary), and provide a reasoning for why the algorithm is correct.* Skipping or having incorrect reasoning will lead to a partial credit, even if the pseudocode itself is OK.

Question 1: Basics .... **[7]**

In (b)-(d) below, answer YES/NO, along with a line or so of reasoning. Simply answering YES/NO will fetch only half the points.

(a) [**1**] Sign up for the course on Piazza! - Yes

(b) [**2**] Let $f(n)$ be a function of integer parameter n, and suppose that $f(n) \in O(n \log n)$. Is it true that $f(n)$ is also $O(n^2)$?
Ordering of Dominating Terms: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 ....$
Since $n^2 > \log n$, in the list of ordering of dominating terms, it is correct to say that $f(n)$ is also On^2.

(c) [**2**] Suppose $f(n) = \Omega(n^{2.5})$. Is it true that $f(n) \in o(n^5)$?
Ordering of Dominating Terms: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 ....$
By the definition of Ω which represents a lower bound, our algorithm cannot have a better runtime than $n^{2.5}$ In the case of the upper bound, we are not limited to $o(n^5)$ since any class of functions with a power greater than 2.5 is a valid upper bound according to the order of dominating terms. Thus it is incorrect to say that $f(n) \in o(n^5)$.

(d) [**2**] Let $f(n) = n^{\log n}$. Is $f(n)$ in $o(2^{\sqrt{n}})$?
Using L'Hopital's rule we can prove that $f(n) = o(g(n))$ by showing $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.
$\to 0 < n^{\log n} < c \cdot 2^{\sqrt{n}}$
$\to \log(n^{\log n}) < c \cdot \log 2^{\sqrt{n}}$
$\to \log n(\log n) < \sqrt{n} \log(2)$
$\to \lim_{n \to \infty} \frac{\log n(\log n)}{\sqrt{n}} = 0$ by L'Hopital's Rule.
Since we have proven L'Hopital's Rule to hold true, we can say that $n^{\log n}$ is $o(2^{\sqrt{n}})$.

Question 2: Bubble sort basics .... **[5]**

Recall the bubble sort procedure we saw in Lecture 1 (see notes): while the input array $A[]$ is not sorted, go over the array from left to right, swapping i and $(i + 1)$ if they are out of order. As I mentioned in class, the running time of the algorithm depends on the input.

Given a parameter $1 < k < n$, give an input array $A[]$ for which the bubble sort procedure takes time $\Theta(nk)$. (Recall that to prove a $\Theta(\cdot)$ bound, you need to show upper and lower bounds.)

Using the Bubble Sort procedure from class as a starting point we can interpret k as the unsortedness of the array. There are two cases for bubble sort: 1) In the first case of bubble sort (worst case) we try to sort an array A[] where all the elements are in descending order to ascending order. In this case, k = N so the outer loop from the procedure will run N times and the inner for loop from the procedure will run N times producing an upper bound of $O(n^2)$. In case 2) (best case) the elements of array A[] are in ascending order and we loop through the outer while loop 1 time and the inner for loop N times. This produces a run time of $O(n)$.

Thus to get a run time of $\Theta(n \cdot k)$ we want to have an array A[] where the first k elements of the array are in descending order and the remaining N - k elements are already sorted. Thus using the procedure description from class the outer while loop will run k times to sort and the inner for loop will loop over all the elements producing a $\Theta(n \cdot k)$.

Question 3: Deletion in prefix trees .... **[6]**

In class, we saw how a prefix tree can be used to store a *set of strings* (which we called a dictionary) over some alphabet Σ. Specifically, we saw how to implement the `add` and `query` operations on such a data structure. (See the lecture notes for more details.) Now, consider implementing the `delete` operation. As suggested in the notes, one way to do this is to mimic the query, and for the node corresponding to the word, set the "IsWord" boolean to false. However, if we are adding and removing multiple strings, this can lead to many tree paths that were created, but don't correspond to any word currently in the dictionary. Show how to modify the data structure so that this can be avoided. More formally, if S is the set of words remaining after a sequence of add/delete operations, we would like to ensure space utilization that is the same (possibly up to a constant factor) of the space needed to store only the elements of S. If your modifications impact the running time of the `add`, `query`, and `delete` operations, explain how.

Based on the description of the problem we want to remove unnecessary nodes. Using the data from Professor Bhaskara's notes on prefix trees, our word set is the following: $a, an, ant, cat, cats$. Now using the tree from the notes, say we delete *cats* from the set of strings S. We mark the isWord boolean for the corresponding word to false and remove the s leaf node. Say the next word we want to remove from our set of strings is the word *cat*. We repeat the same process for cat that we did for cats. Once we remove the t node for *cat* we run into a special case. CA is not a valid word which we get from $isWord$. If the node for A does not have any children, say a word like $camera$, then we can delete the A node. We then can repeat this process until we hit the root node. This process handles the space constraint described in the problem because we can assume that we have a finite amount of space to store the set of strings. The *deletion* operation of removing a node is constant time but traversing up the prefix tree removing nodes will take $\log(h)$ where h represents the height of the prefix tree. The *add* and *query* operations do not get affected.

Question 4: Binary search and test pooling .... **[10]**

"Test pooling" is a trick that is used when testing for a disease is expensive or has limited availability. The idea is the following: suppose we have n people (numbered $1, 2, \ldots, n$ for convenience), instead of testing each one, samples from a subset S of the people are combined and tested, where a test runs in $O(1)$ time regardless of the size of S. If at least one of the people in S has the disease, the test comes out positive, and if none of the people in S has the disease, it comes out negative. (Let us ignore the test error for this problem.)

It turns out that if only a "few" people have the disease, this is much better than testing all n people.

(a) **[4]** Suppose we know that *exactly one* of the n people has the disease and our aim is to find out which one. Describe an algorithm that runs in time $O(\log n)$ for this problem. (For this part, pseudocode suffices, you don't need to analyze the runtime / correctness.) This problem can be interpreted as a variation of the binary search algorithm//

Algorithm 1 Test Pooling Algorithm: Binary Search

```
Require: $A[1, 2, 3, 4, 5, ...n]$ where each value is a unique id to the person for testing, lo - the
  starting index of the array, hi - the last index of the array
  if $hi - lo$ then
      return negative
  end if
  $mid \leftarrow (lo + hi)/2$
  if $mid == lo == hi$ then
      return positive
  end if
  if isPositive(A[lo, .... mid - 1]) then        ▷ we are assuming this array does not include mid
      TestPooling(A, lo, mid - 1)
  end if
  if isPositive(A[mid, ... hi]) then             ▷ we are assuming this array includes the mid
      TestPoolling(A, mid, hi)
  end if
```

(b) [**6**] Now suppose we know that *exactly two* of the n people have the disease and our aim is to identify the two infected people. Describe **and analyze** (both runtime and correctness) an algorithm that runs in time $O(\log n)$ for this problem.
To find the two values that are positive, we utilize storage in the form of an array where the algorithm will be run twice and each time the algorithm is run, the positive result will be added to the array.
Algorithm

Algorithm 2 Test Pooling Algorithm: Binary Search

```
Require: $A[1, 2, 3, 4, 5, ...n]$ where each value is a unique id to the person for testing, lo - the
  starting index of the array, hi - the last index of the array, $result[]$ - this represents storage for
  storing the final output of the two values
  if $hi - lo < 0$ then
      return
  end if
  $mid \leftarrow (lo + hi)/2$
  if $mid == lo == hi$ then
      add element to result
      return
  end if
  if isPositive(A[lo, .... mid - 1]) then        ▷ we are assuming this array does not include mid
      TestPooling(A, lo, mid - 1)
  end if
  if isPositive(A[mid, ... hi]) then             ▷ we are assuming this array includes the mid
      TestPooling(A, mid, hi)
  end if
```

Correctness
We can prove the correctness of $TestPooling$ as a recursive routine.

Let us define L of the subarray for the subset S. Then if $L \leq 1$ then we are a bound to get the right answer because of the statement where we say if $lo == mid == hi$ then we have found the right person because we have exactly one item in our list.
Next, we assume if the algorithm returns the right answer for $L \leq t$ where $5 \geq 1$. Since we include the midepoint for the right subarray, we only add a value to *return* if $lo == mid == hi$ which is a single value. Thus we use small subarrays and the inductive hypothesis holds because we keep subdividing the set S if our *isPositive* function returns true until a reach a list of a single element. Since we also define A as sorted at the beginning of our algorithm we are bound to find one of the positive values in either the left or right subarrays if *isPositive* functions returns true. Then if the algorithm returns the right answer for $L \leq t$ for some $t \geq 1$, then it also returns the right value for $t + 1$ using similar logic from the correctiveness proof from Professor Bhaskara's notes.
Runtime
Since we stated that Testing Pool is a variation of binary search which was proven in Professor Bhaskara's notes the running time is $O(\log n)$ because we are subdividing our sample into two until we reach the case with a single element. *TestingPool* needs to be called twice because a single run of the algorithm finds only a single positive person at a time. Dropping the leading constant terms we get a runtime of $O(\log n)$.

Question 5: Recurrences, recurrences .... **[16]**
Solve each of the recurrences below, and give the best $O(\cdot)$ bound you can for each of them. You can use any theorem you want (Master theorem, Akra-Bazzi, etc.), but please state the theorem in the form you use it. Also, when we write $n/2$, $n/3$, etc. we mean the floor (closest integer less than or equal to the number) of the corresponding quantity. **Please show how you obtained your answer.**

(a) **[4]** $T(n) = 3T(n/3) + n^2$. As the base case, suppose $T(n) = 1$ for $n < 3$.
This problem is solved using the Master Theorem.
Master Theorem:
$T(n) = aT(\frac{a}{b}) + O(n^d)$ for some constants $a > 0, b > 1, d >= 0$.
By the equation $a = 3, b = 3, d = 2$
$\rightarrow \log_b a = 1$
$d > log_b a \rightarrow$ runtime: $O(n^2)$

(b) **[6]** $T(n) = 2T(n/2) + T(n/3) + n$. As the base case, suppose $T(0) = T(1) = 1$.
This problem is solved using the Akra-Bazzi Method.
Akra-Bazzi Method:
$T(n) = \sum_{i=1}^{k} a_i T(b_i n) + f(n)$
$a_1 = 2, a_2 = 1, b_1 = \frac{1}{2}, b_2 = \frac{1}{3}, f(n) = n$
$\rightarrow a_1 b_1^p + 2b_2^p = 2 \cdot (\frac{1}{2})^p + 1 \cdot (\frac{1}{3})^p = 1$
Using wolframalpha to solve for p $p \approx 1.3646$
$T(n) = \Theta(n^{1.3646}(1 + \int_1^n \frac{f(u)}{u^{1.3646+1}} du))$
$\rightarrow \Theta(n^{1.3646}(1 + \int_1^n \frac{u}{u^{2.3646}} du))$
$\rightarrow \Theta(n^{1.3646})$

(c) **[6]** $T(n) = 2(T(\sqrt{n}))^2$. As the base case, suppose $T(1) = 4$.
This problem is solved using the Plug and Chug Method.
$T(n) = 2(T(\sqrt{n}))^2$

$\rightarrow T(n^{\frac{1}{4}}) = 2 \cdot T((n^{\frac{1}{4}}))^2$
$\rightarrow T(n) = 2^3 \cdot T(n^(\frac{1}{4})^4$
$\rightarrow T(n) = \prod_{i=0}^{k-1} \cdot T(n^{\frac{1}{2k}})^{2^k}$
$\rightarrow k = \log(log(n)) \rightarrow 2^k = \log(n)$
$\rightarrow T(n^{\frac{1}{2k}}) = T(n^{\frac{1}{2}^{\log(\log(n))}}) = T(2) = 32$
$\rightarrow 2^{\log(n)-1} \cdot T(2)^{\log(n)}$
$\rightarrow 2^{\log(n)-1} \cdot 32^{\log(n)}$
$\rightarrow \frac{2^{\log(n)}}{2} \cdot 2^{\log(n)^5}$
$\rightarrow \frac{n}{2} \cdot n^5 = \frac{n^6}{2} = O(n^6)$

Question 6: Dynamic arrays: is doubling important? . . . . **[6]**

Consider the 'add' procedure for dynamic arrays (DA) that we studied in class. Every time the add operation was called and the array was full, the procedure created a new array of twice the size and copied all the elements, and then added the new element.

Suppose we consider an alternate implementation, where the array size is always a multiple of some integer, say 32. Every time the add procedure is called and the array is full (and of size n), suppose we create a new array of size $n + 32$, copy all the elements and then add the new element.

For this new add procedure, analyze the asymptotic running time for N consecutive add operations.
We first define a default size $n_0 = 32$ which represents the initial size of our array. We next then write out the first few terms to analyze the behavior:
$\rightarrow$ 0 elements - the array is initialized to n_0
$\rightarrow$ 32 elements - the array is doubled to a a new size of 64 to make space for the 33rd insertion
$\rightarrow$ 96 elements - the array is tripled to a new size of 96 elements to make space for inserting 65 insertion
Next we can describe a formula for this behavior as the following:

$$T(n) = \begin{cases} n + 32 & \text{where n = 32k for some integer k, where we have to double the size of the array} \\ 1 & \text{for all cases that do not involve doubling the array} \end{cases} \tag{1}$$

Next to show ammortize anlysis we need to show the average cost across all additions. This is written as the following:
$\sum_{i=0}^{N-1} T(n) \rightarrow \sum_{i=0}^{N-1} T(n) = \sum_{i=0}^{N-1} i + 32 + \sum_{i=0}^{N-1} 1$
The summation adding 1 can be simplified to $N - k$.
Using Gauss's Rule, we can simplify the first summation to the following:
$\frac{32+32k}{2} \cdot k$.
We now substitute these two summations into the ammortized insert cost functions:
$\frac{1}{N} \cdot \sum_{i=0}^{N-1} T(n) = \frac{32+32k}{2} \cdot k + N - k$
$\rightarrow \frac{16k+16k^2+N-k}{N}$
$\rightarrow \frac{16k^2+47k}{32k} = O(k)$
Thus the ammortized runtime for the addition operation described in this problem is $O(n)$ as proven above.