Problem Set 2

● Graded

Student

Alina Afghan

Total Points

24 / 24 pts

Question 1

Selection Bounds **6** / 6 pts

✓ **– 0 pts** Correct

– 3 pts Incorrect part (a).

– 1.5 pts part(a): Complexity of comparison based sorting of a list containing n^(3/4) largest elements is incorrect.

– 2 pts part (a): Complexity of sorting the sorted chunks into a single sorted list using merge sort is nlogn

– 1.5 pts part(a): Sorting of the elements in the GREATER set is missing.

– 1 pt part(a): In order to obtain a list of n^(3/4) largest elemnents, the k^th element needs to be identified where k=n-n^(3/4).

– 3 pts Incorrect part (b).

– 1.5 pts part(b): Proof of "minimum n-1 comparisons are required" is missing.

– 1.5 pts part(b): Proof of "minimum n-1 comparisons are required" is incorrect.

Question 2

Union of Median

Resolved 6 / 6 pts

✓ **+ 2 pts** Non-exact upper and lower bounds as logarithmic functions i.e. $\theta(log(n))$
or
Non-exact explanation of recursive solution that includes halving arrays

+ 0.5 pts Upper bound derived from $O(log_2(2n))$ explicitly

+ 0.5 pts Lower bound derived from $\Omega(log_2(2n))$ explicitly

✓ **+ 1 pt** Correct argument for location of median w.r.t arrays

✓ **+ 1 pt** Recursively run on $A' = A[\frac{n}{2}+1]...A[n]$ and $B' = B[1]...B[\frac{n}{2}]$ or equivalent, w.r.t to medians

+ 1 pt Complexity of creating new problem i.e. one comparison of medians for dividing arrays in half
$T(n) = 1 + log(n) = log(2n)$

− 6 pts Missing/Plagiarized/Completely incorrect solution

+ 2 pts Point adjustment

Regrade Request **Submitted on: Mar 23**

Have given argument for lower bound Ω that essentially states the EXACT given lower bound, mentioned log(2n) and splitting in half which implies log base 2.

Adjusted.

Reviewed on: Apr 11

Question 3

Space Efficient Arrays

6 / 6 pts

− 0.5 pts (a) Sequence or overall cost is missing.

− 1 pt (a) Argument has inaccuracies.

− 2 pts (a) Incorrect.

− 1 pt (b) Inaccuracy in argument for banking method or equivalent.

− 1.5 pts (b) The cost calculated is not correct.

− 2 pts (b) Incorrect.

− 1 pt (c) Argument has inaccuracies.

− 2 pts (c) Incorrect.

✓ **− 0 pts** Correct

Question 4

Searching for Sum One

6 / 6 pts

✓ **– 0 pts** Correct

– 1 pt (a) Incorrect algorithm.

– 1 pt (b) i. Incorrect proof.

– 1 pt (b) ii. Incorrect construction of A and / or B.

– 1 pt (b) iii. Incorrect proof structure.

– 1 pt (b) iii. Details of reduction are incorrect or missing.

– 1 pt (b) iii. Incorrect conclusion / contradiction.

Question assigned to the following page: 1

ps2

Alina Afghan 24491

February 2024

1

(a) We can use deterministic Quick Select to traverse set S.

Set S has n arbitrary but distinct numbers. In order to identify the largest $n^{3/4}$ elements in set S through a deterministic algorithm, we can modify the deterministic version of Quick Select from the notes. Instead of using it to find the k^{th} smallest element, we can use it to find the k^th largest element instead, and take $n - n^{3/4}$.

The recursion relation will be the same, deterministically choose median as pivot in the same way, we have already seen that at least 3/10 of the array will be ¿ p and that means recursing on the bigger sub-list of size 7/10 will be the same as well.

We also have to find the ordered list of those $n^{3/4}$ elements. to do this we can in n comparisons find the ones greater than our k^{th} largest element. then sort the list using a sorting algorithm in $nlog(n)$ time, but our n in this case is $n^{3/4}$, so we have $n^{3/4}logn^{3/4}$.

lets reduce this down.

$$= n^{3/4}logn^{3/4}$$
$$= n^{3/4}3/4log(n)$$
$$= n^{1/4} \times n^{3/4} + n^{3/4}3/4log(n)$$
$$= n^{1/4} + 3/4log(n)$$

3/4 is a constant we can discard, and $n^1/4$ is gonna be the dominant term in this so we can do this in O(n).

(b) Assume for the sake of contradiction that we can have a deterministic comparison based algorithm that correctly finds median on n inputs in lesser than n-1 time, e.g n-2 time. If this were possible then that means we have one element that has not been compared and the value of the median may change based on that value.

We can prove this by representing this algorithm in the form of a graph where every element is a vertex and we draw an edge in between any two vertices

where we carry out a comparison, so with n-2 comparisons we obtain a graph of n vertices and n-2 edges. Since in order to be fully connected a graph needs n-1 edges, our graph is going to be separated into two components. The median will be in one of those two components, and there will exist some vertex in the other component that is not connected to the median vertex, which mean that we don't know whether that element is greater or lesser than the median. This means that either the number of elements greater than the median or the ones lesser than the median may change, which would also change the value of the median itself.

Question assigned to the following page: 2

2

In order to prove the upper bound for this problem, we can define the following algorithm. Lets call the median of A M(A) and the median of B M(B).

If M(A) < M(B), then we know that the elements in A < M(A) will lie on the left side of the union set A and B, i.e from the 1st to the $\frac{n}{2}th$ indices, and the median cannot lie within those, so it can be discarded. The same is true for elements in B that are > M(B), so we can then recurse on the reduced A and B arrays.

In the case of M(B) > M(A), we can discard the right side of A and the left side of B using the same logic.

Keep recursively applying this algorithm until we find the median. Since we divide 2n by 2 until only one element is left on either side, this results in $log_2(2n)$ recursions. This is similar to a binary search algorithm and since both A and B are of size n, the time complexity is $O(log(n))$.We can define the recurrence relation as follows, taking 1 as the constant for the comparison of medians.

$T(n) = T(\frac{n}{2}) + 1$

To prove a matching lower bound, we can say that since there are n elements in both arrays, there are 2n possibilities for the median of the union of A and B. In the worst case, the median of the union may lie at the end of either array, which means we need to recurse until only one element is left in each array. Since there are 2n elements in A U B, this would mean splitting A and B log(2n) times.

Each time we can only reduce the array by half because if we reduced it by more than half, then one sub-list would be greater and the other sub-list would be smaller. If we take the adversarial approach to this problem, then the adversary would be able to choose the greater sub-list every time in order to increase the complexity, so the best possible case we can take is for both sub-lists to be of equal length.

Question assigned to the following page: 3

3

(a) In the worst case, we would have a case where after every operation, we need to resize the array. In order to generate this case, lets say that half of our operations are simple pushes, so we perform n/2 push operations. And lets say that at the $(\frac{n}{2}+1)^{th}$ operation, our stack becomes full. For our remaining n/2 operations, we say half of them are pushes and half of them are pops, so we alternate between 2 pushes then 2 pops for n/4 operations each. Each push and pop operation will cost $\frac{n}{2}$. We can do the cost analysis for this as follows:-

$$\text{cost} = \frac{n}{4} + \frac{n}{2} + \frac{n}{4} + \left[\frac{n}{4}\left(\frac{n}{4}\right) + \frac{n}{4}\left(\frac{n}{4}\right)\right]$$

$$\text{cost} = \frac{3n}{4} + \frac{n}{4} + \left[\frac{n}{4}\left(\frac{n}{4}\right) + \frac{n}{4}\left(\frac{n}{4}\right)\right]$$

$$= \frac{4n}{4} + \frac{2n^2}{16}$$

$$= n + \frac{n^2}{8}$$

$$= 1 + \frac{n}{8}$$

In the last step we have conducted aggregate analysis and divided by n, to obtain an O(n) cost for this strategy.

(b) Lets use the accounting method to solve this problem. We can say that out of 3 units, we use one unit to perform the operation and the other two are stored in the bank to use for resizing operations.

When we have just performed a resize operation, we will either have just performed a push or a pop. In the case of a push, we had L elements and resized into a stack of 2L, so now if we take the new size to be L we have L/2 elements. In the case of a pop, we cut the array size from L to L/2 when k = L/4 so now our stack is half full, and there are L/2 elements. So we know that in either operation that causes a resize, we have L/2 elements in the stack.

Now lets assume our current bank balance is 0, since the credits will be used up by the resize. Before arriving at the next resize operation, we need to reach L elements, so from L/2 we need L/2 more, and putting two credits in the bank for each op results in $\frac{L}{2} \times 2$ equals L credits in the bank.

The resizing operation is going to cost L as well, so our bank balance will use the provided credits and not go into negative.

Question assigned to the following page: 3

In terms of pops, whenever a list of L/4 elements is being resized, we know it went back down from L/2, so L/4 pops will have occurred, and once again saving 2 credits per operation means $\frac{L}{4} \times 2$ equals $\frac{L}{2}$ credits stored in the bank.

The resizing operation for pops will use $\frac{L}{4}$ credits since that is the amount of elements we must move. This results in $\frac{L}{2} - \frac{L}{4} = \frac{L}{4}$ remaining credits, meaning our bank balance stays positive.

(c) In this strategy, we are performing the following summation for n operations for some k.

$$n = 1 + 2 + 3 + ... + k = \frac{k(k+1)}{2}$$

This gives us the number of operations in terms of k. In each operation, the length being added to resize the operation increases by 1, so at each level L we are moving k-L elements. Since we know the cost of resizing is the number of elements moved we can say that:

$$[(k)(1) + [(k-1)(2)] + [(k-2)(3)] + ... + [(1)(k)]]$$

$$\sum_{i=0}^{k} [(i)(k+i-1)]$$

This sum resolves to a polynomial in terms of k^3. In order to conduct aggregate analysis, we divide by k^2, and obtain k. Now when we replace this in our original summation $\frac{k(k+1)}{2}$, it resolves to $\sqrt{n}$. So our amortized cost per operation for n pushes will be $\sqrt{n}$.

4

(a) We can define an algorithm to solve the sum query problem using only $O(n)$ calls to the test operation as follows.

Begin at the 1st index of array A, so smallest value and last index of B, so largest since both are sorted, and then call test. If the response is true then store q. If the sum of A and B is $< q$ then increment the index of A. If the sum of A and B is $> q$, then decrement the index of B. In the worst case, we end up traversing all of A and B, resulting in an order of $n + n$ magnitude, which is just $O(n)$.

(b)(i) Lets consider the case that no such k exists. In this case, we would have to compare every element in C with q in order to ensure the absence of k. Since there are n elements in C, this results in n comparisons. Lets assume for the sake of contradiction that it would be possible to find k in less than n time. Lets say it can be found in $n - 1$ time. We can show this using a graph of n vertices for n elements in C, and add one more vertex for q. Draw edges between each vertex and q wherever we are conducting a comparison. At most we will have n-1 edges, which means one node will remain disconnected. This node may end up being equal to q, which means our algorithm becomes non deterministic.

(ii) We can define the construction as follows:

n_1	n_2	n_3	n_4	n_5	n_6	...	n_i

Table 1: Array C

indices	1	2	...	$n-1$	n
A	$-n_1 - (1 \times S)$	$-n_2 - (2 \times S)$	...	$2n_{n-1} + ((n-1) \times S)$	$2n_n + (n \times S)$
B	$-n_n - (1 \times n)$	$-n_{n-1} - ((n-1) \times S)$	...	$2n_2 + (2 \times S)$	$2n_1 + (1 \times S)$

Table 2: Construction of arrays A and B

Since we are not allowed to compare any elements, but we are restricted to keep certain elements in both arrays greater than the max, we take a value S, and say that S is the sum of every element in C, in order to fulfill the given condition.

For all indices $i < \frac{n}{2}$, assign the elements of A the number stored at the matching index of C but flip the sign to negative, and subtract the index times S.

For all indices $i > \frac{n}{2}$, assign the elements of A the number stored at the matching index of C times 2, and add the index times S. We do the exact same for array B, but the indices go from n to 0 instead of 0 to n.

Question assigned to the following page: 4

(iii) Lets say that Sum-Query can be done through using less than n calls, lets call the algorithm that does this algorithm A. Since we have shown that, without using any comparisons, an unsorted array C can be used to construct arrays A and B, pairs of elements of whom sum to form the elements in C, we can work backwards and directly take the array C as the sum of A[i] and B[j], and call the test operation to determine whether or not this equals q. Thus we can use algorithm A to find whether or not there exists an index k in an unsorted array C such that C[k] = q in less than n comparisons. (since the test operation is a comparison). However, in part i we have proved that we require n comparisons in the worst case to find q in C, which means that algorithm A cannot be correct. Therefore, Sum-Query cannot be solved using less than n calls to test.

collaborated with Maaz Karim 24503