Problem Set 2

Hiba Mallick - 24015

CSE 317 Design and Analysis of Algorithms
Spring 2024

1 Selection Bounds

(a) To find the largest $n^3/4$ numbers in S in sorted order, we will use a modified deterministic Quick Select. We will begin by choosing a pivot point (Lecture Notes, Feb 1, page 4, steps 1-4) and then we will recurse on that p, comparing the index p to $[n - n^{3/4}]$.
If $p < [n - n^{3/4}]$ then $p = p + 1$
and if p > $[n - n^{3/4}]$ then p = p - 1
else (where p = $[n - n^{3/4}]$, we have found the $n^{3/4}$th largest number and section of the list including p and the elements to its right (list G) are the largest $n^3/4$ numbers in S.
Now, we can sort the list which is of length $n^3/4$. Sorting for length n can be done in $nlogn$ so this can be done in $n^{3/4}logn^{3/4}$.
We know that steps 1 and 2 (choosing and recursing on a pivot) can be done in $\Theta(n)$ comparisons, and step 3 (sorting list of length $n^{3/4}$ can be done in $\Theta(n^{3/4}logn^{3/4})$ comparisons. Thus, this can be done in $\Theta(n)$

(b) To show that finding the median of n distinct elements must use at least n-1 comparisons, let's consider the best case when using a deterministic sorting algorithm, which would be when we partition our problem in such a way that our partitioning produces two sub-problems that are roughly the same size.
For instance, a binary tree where the height of the left and right sub-trees of any node differ by not more than 1 (balanced tree). This can be possible if, in a sorting algorithm like deterministic QuickSort, our initial pivot when using median of medians is such that the size of the LESS and GREATER lists differ by not more than 1.
In that case, we would compare each other element (n-1) of the list of length n to check if it was LESS or GREATER than the chosen pivot.
If the initial pivot is the true median (best case), it would only take n-1 comparisons. If the initial pivot is not the true median, the position of the pivot would shift by 1 to the side of the longer sublist. Thus proving that finding the median requires *at least* n-1 comparisons in the best case.

2 Union of Median

To find the median of two sorted arrays A and B of size n, we will use the comparison-based lower bound method studied in class.
We know that the lower bound of finding the median is $\Omega(\log n)$ since at each point only 1 element is discarded. When there are 2n elements (n from A and B each, it takes $log_2(2n) = \log_2 n$ comparisons in the worst case to find the median. To find the mean. assume M1 is the median of A and M2 is the median of B and compare them. If M1<M2, apply the algorithm on only the right half of A and left half of B. If M1>M2, apply the algorithm on only the right half of B and left half of A. Else (M1 = M2), we have found the median of the union.

The recurrence relation for this is $T(n) = T(n/2) + O(1)$, where $T(n)$ is the time to find the median of two arrays of size n. The time complexity of this recurrence relations comes out to be $O(\log n)$which is equal to the lower bound.
Thus we can see that the upper and lower bound match and there is no gap between the lower and upper bound for this problem..

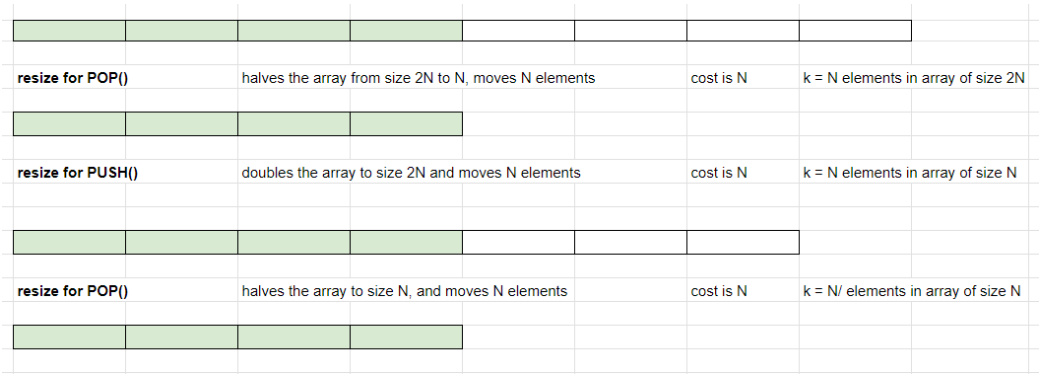

Figure 1:

3 Space Efficient Arrays

(a) For simplicity, we will be defining the number of elements present in the list (k) and size of the list (L) in terms of N
We will first define a cost model for this method of halving the array of size 2N with N elements whenever we perform a pop and doubling the size of the array whenever it is full (performing a push).
In the case that we need to perform a roughly equal number of pushes and pops alternatively, we can calculate the total cost of all operations.
As seen in Figure 1, we know that when resizing (halving) for a pop, N elements will be moved, which costs N.
Similarly, from the question and Figure 1, we know that each push operation (doubling) will cost us the number of elements moved which also costs N.

For N operations, where each operation requires N elements moved, the total cost of all operations will be N^2
The amortized cost will be $N^2/N = \Theta(n)$
This is called 'thrashing', and it is an inefficient strategy since halving the array at each pop causes N time at each operation, leading to N^2 time overall.

(b) If we change our method of resizing for a pop to halving the size of the array only when there are N/4 elements in a list of length N (k = L/4), we can define our cost model by:
Halving an array when L/4 elements remain costs L/4
Doubling an array when full costs L.
This method ensures that the array is always between a quarter full or com-

pletely full, so we don't have to cut down the array as frequently as in part a.
Since the halving will move L/4 of the elements, the cost of a resize for pop is L/4, and the cost of doubling for a push will be L. Thus, every time the array is resized and made smaller, we have already paid for the work done for the expensive, less frequent operations by the cheaper, more frequent operations (whose extra cost is stored in a bank)
For any sequence of n operations, the total cost for resizing becomes 1+2+4+8+...2^i < n. where 2^i will be the largest power of 2 less than n, so the cost $< 2n-1$. or $2n$
The insertions and removal operations cost n each so adding that, we get the total cost to be $< 3n$ The amortized cost is thus $< 3n/n = 3$

(c) In this new cost model, we increase the size of the array by n at the nth resize. Thus, the size of the array and the number of elements moved (cost) at each resize is:
k = 1 initially
2 after 1st resize
4 after 2nd resize
7 after 3rd resize
11 after 4th resize and so on. The cost increases by n at the nth resize.
This is a recurrence relation in which k can be written as a function of n (where k is the number of elements we have moved and n is the number of resizes performed):
$f(n) = n + f(n-1)$
$f(n) = n + (n-1) + f(n-2)$
$f(n) = n + (n-1) + (n-2) + ... + 1 + f(0)$

since the length of the array at 0 resizes is 1
$f(n) = n + (n-1) + (n-2) + ... + 1 + 1$
$f(n) = 1 + (1 + 2 + 3 + ... + n)$
$f(n) = 1 + (n(n+1))/2$
$f(n) = (n^2 + n + 2)/2$

This shows us that after the nth resize, the length of the array after n resizes or the work done for n resizes will be:
$(n^2 + n + 2)/2$ or n^2
The amortized cost per n operations is $\Theta(n)$

4 Searching For Sum One

(a) For the Sum-Query problem, we will refer to A's indices as i and B's indices as j. To check if a given real number between 0 and M, called q defined as A[i] + B[j], we will initialize a nested for loop.
The first loop initializes i = 0 where i < N
The second loop initializes j = n where j > 0
At each iteration, perform A[i] + B[j] and call test(i,j) to give the result of comparing q with A[i] and B[j]. test(i,j) returns three possible answers: or A[i] + B[j] is Greater than q, A[i] + B[j] is Less than q, or A[i] + B[j]is Equal to q.
If A[i] + B[j] is Greater than q, we will decrement the value of j to n-1
If A[i] + B[j] is lesser than q, we will increment the value of i to 2.
If A[i] + B[j] is Equal to q, we have found our answer.
This helps us solve Sum-Query in O(n) calls to test(i,j).

(b) (i)
To check if a given real number q is present in our unsorted list C of size N, we will have to compare each element in the list with q, a.k.a linear search. Since linear search requires that we linearly check each of the n elements in the list against the given q, there will be n comparisons in the worst case, which is the case where k = n (k is the last element in the list).

(ii)
From the first condition, we will begin filling A's values with the same values from C at the respective index and fill B with 0s, so that the diagonal indexes sum up to the C[i]s. Then in each array, have a difference of M between each index as in the worst case subtract M from M (in B) or add M to 0 (in A) to ensure that the other conditions are fulfilled. According to the 3rd condition, the sum should be greater than M if i + j > n+ 1, since M is not explicitly defined, we intuitively know that the sum of all elements of C is greater than M, so we use SUM to fulfill this condition.

Thus A must be constructed as
A[i] = C[i] + i.*SUM (where SUM = the sum of all elements of C and is the range of the index of the arrays.)
A[1] = C[1] + S),

A[2] = (C[2] + 2S)
A[3] = C[3] + 3S)
A[4] = (C[n] + n*SUM)

B must be constructed as:
B[i] = 0 − (n + 1 − i)*SUM
B[1] = (−nS)
B[2] =(−(n − 1)S)
B[3] = (−(n − 2)S)
B[4] = (−S).

(iii)

To infer a lower bound of n calls to test(.,.) for the Sum-Query problem from parts b(i) and b(ii) we will think of the last part as a mapping of b(i) to the Sum-Query problem. Since at least n comparisons are required to solve b(i), b(ii) also needs n comparisons. Since we are now able to construct the two arrays where (A[i] and B[n + 1 − i]) = C[i] with no comparisons, we know that the lower bound for finding q here is also going to be O(n).