Homework 4

Thursday, March 25, 2021 9:57 PM

Ian Chadwick
U90596416

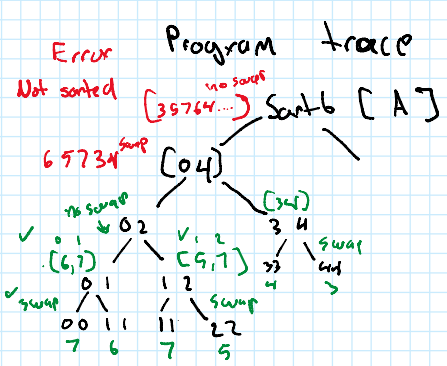
1. Does it sort? [10 pt]

For the following algorithm:

i. Does the algorithm correctly sort (in ascending order) all arrays A of size *n* containing only positive integers, for $n \geq 8$?

ii. If the answer to the previous question is "yes", provide a short explanation and a worst-case (asymptotic) running time. If "no" give a small counterexample (i.e. an input array and the corresponding output array that is not sorted properly).

```
sortB(Array A[i..j]) // call sortB(A[0..n-1]) to sort A[0..n-1]
   if (i == j)
      return
   mid = (i + j)/2
   sortB(A[i..mid])
   sortB(A[mid+1..j])
   if (A[i] < A[mid+1])
      swap(A[i], A[mid+1])
   return A;
```



i: 0 1 2 3 4 5 6 7 8

A [7,6,5,4,3,2,1,0]

mid = 0+8/2 = 4

This function does not sort the array. After tracing the first half of array A above I stopped the trace because it wasn't sorted and it would remain that way despite the other half

first half = A [3 5 7 6 4 ...]

2. Heap [10 pt]

Given eight numbers $\{n_1, n_2, \ldots, n_8\}$, show that you can construct a heap (either min-heap or max-heap) using eight comparisons between these numbers (e.g., comparing n_1 and n_2 would be one comparison).

{1 2 3 4 5 6 7 8}

Make 4 comparisons to make ordered pairs

2>1, 4>3, 6>5, 8>7 4 comparisons

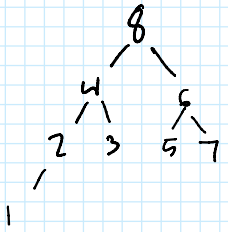
Make comparisons between greater of each ordered pair to make 2 trees

2<4 so, and 6<8 so 4+2 = 6 total comparisons

2 < 4 so, and 6 < 8 so 4 + 2 = 6 total comparisons

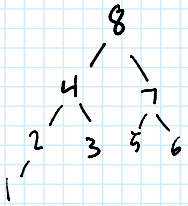
Then compare the roots of each tree

8 > 4 so, 6 + 1 = 7 comparisons



Then you can use the last comparison to arrange the right most tree

7 > 6 7 + 1 = 8 comparisons



3. Median [10 pt]
Describe an efficient algorithm for finding the median of all numbers between *two sorted arrays each of size n*. Your algorithm should be asymptotically faster than $O(n)$. Analyze the running time of your algorithm.

To make the runtime faster than O(n) you would have to eliminate some of the elements. The best way to do this is with a recursive algorithm.

1) The base case for this algorithm would be if each array was reduced to size 2 then the median would be the sum of the maximum of the 1st index value from the two arrays and the minimum of the 2nd index of the two arrays divided by two. Return that value as median

2) If that case is not true, then find the median index of each array and get the value of that number.

3) compare the two median values. If they are equal, then that value is the median of both arrays. Return that value

4) if the median from the first array is less than the median

value is the median of both arrays. Return that value

4) if the median from the first array is less than the median from the second array, get rid of all values less than median1 from array 1 and all values greater than median2 from array two and recurse on those arrays

5) If median 1 is greater than median2 then do the opposite of the previous step and get rid of all values greater than median 1 from array 1 and all values less than median 2 from array 2. Then recurse on both arrays again

This would run in O(logn) because each recursive call would eliminate half of each array, like binary search.

Pseudocode:

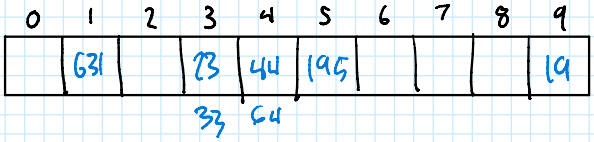
```
FindMedian (int A[], int B[], int start_a, int end_a, int start_b, int end_b)
if ((end_a - start_a == 1) && (end_b - start_b == 1))
    return (max(A[start_a], B[start_b]) + min(A[end_a], B[end_b]))/2

midA ← (start_a + end_a)/2
midB ← (start_b + end_b)/2
medianA ← A[midA]
medianB ← B[midB]
if (medianA == medianB)
    return medianA;
if (medianA < medianB)
{ start_a = midA
  end_b = midB
}
else // (medianA > medianB)
{ start_b = midB
  end_a = midA
}
return FindMedian(A, B, start_a, end_a, start_b, end_B);
```

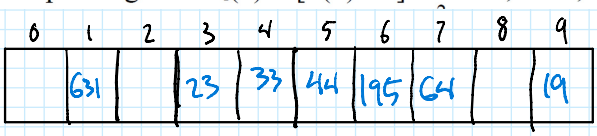
4. Hashing [20 pt]

Given a sequence of inputs 631, 23, 33, 19, 44, 195, 64 and the hash function h(x) = x mod 10, show the resulting hash table for each of the following cases of conflict resolution.

(a) Chaining



(b) Linear probing with $h_i(x) = [h(x) + i] \bmod 10$, i = 0, 1, 2, … , 9

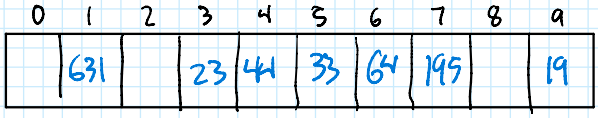


Values 631, 23, 33, 19, 44, 195, 64

Keys (1, 3, 3, 9, 4, 5, 4)

Probe 4 5 6 7

(c) Quadratic probing with $h_i(x) = [h(x) + i^2 + i] \bmod 10$, i = 0, 1, 2, …, 9

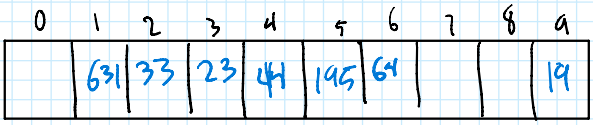


Values 631, 23, 33, 19, 44, 195, 64

Keys (1, 3, 3, 9, 4, 5, 4)

Probe 5 7

(d) Using a second hash function $h_2(x) = 7 - (x \bmod 7)$



Values 631, 23, 33, 19, 44, 195, 64

1st Hash Keys (1, 3, 3, 9, 4, 5, 4)

2nd Hash 2 6

Ian Chadwick
U90596416

Programming Question 5a)

I used a modified counting sort algorithm because we know that the input must be one of 26 lowercase letters.

1. I created a comparison string of the alphabet and an integer "count" array of size 26 initialized with 0s to hold the counts of each letter. O(1)
2. Then I looped through the input string once for each letter of the alphabet and stored the count in the count array this ends up being 26 * 0(n).
3. Then I looped through the count array and pick the smallest value to put into the output string, with alphabetical order being the tiebreaker. O(1)
    a. I used a while loop to keep adding letters until the count at that index is 0. O(1)
4. Once the loop checked the entire count array, exit the loop and return the string with the sorted letters in the correct order. O(1)

Time complexity: 26* O(n) + 4*O(1) = O(n)
Space complexity: n + 1 array size 26 + 1 string size 26 = O(n)

Since we must traverse the entire input string at least once because there is no other way to guess the input based on the given information, that automatically makes the best run time 1* O(n). Since we're looping through input string n 26 times that also makes the algorithm in O(n). The space used is a fair tradeoff for the speed with only an additional string of size n, a string of size 26, and an integer count array of size 26 added. This means that the space complexity is O(n).

In order to reduce the space, you could sort it in place, but that would come at the cost of increasing the runtime significantly because you would have to do a comparison sort, the best of which runs in O(nlogn).

Ian Chadwick
U90596416

Programming Question 5b)

Since the vector given is sorted, you can step through the vector number by number and compare each value at each index with the value at the next index. If they are equal, increase the count by one. Then, you can check to see if the new count is greater than the current maximum count. If it is, then replace the value with the value at that index.

If the value at the current index is not equal to the value at the next index, reset the count to 1 and step through again.

Once the maximum count is greater than the remaining number of values you can exit and return the current value as the mode because it's not possible for any other value to be the mode.

This ends up running in O(n) time. With the constant being some fractional value based on the count of the mode and how early it can exit the vector because the mode count had exceeded the count of the remaining values. The best case would be if the first half of the array was the mode so it would exit after the half. The worst case would be O(n) because it would have to traverse the entire vector if all the counts were 1.

The space complexity is only the size of the input vectors and a few constant values for the count and the value. If you don't include the input vector in the space it would be space complexity O(1) constant, but if you do include the vector it would be O(n).

Time Complexity = O(n)
Space Complexity = O(1) (or O(n) if the input vector is included)