Homework 2

Indukuri Vijay Anand Varma (VXI210000)
Karthik Raghunath Ananda Kumar (KXA200005)

Problem 1. Function Queries (16 points)

Let $f : \mathbb{R} \to \mathbb{R}$ be a *monotonically increasing* function. You are not told f directly, but you have blackbox access to f. Specifically, for any value x you can query $f(x)$.

Let n be the smallest positive integer for which $f(n) > 0$. You can assume the n is well defined, i.e. you are told there exists values for which f is positive.

Give an algorithm to compute n, where your algorithm queries the blackbox for f on only $O(\log n)$ values.

Intuition:

We have to find an element which is greater than zero in monotonically increasing function. This is nothing but finding first non-zero element in an increasing array. Using binary search, we can search for an element in sorted array in O (log n) time. Similarly, we can use binary search to find the first non-zero element in a monotonically increasing function.

Since we don't know the start element and end element for given function, we assume that function starts at Integer.MIN_VALUE and function ends at Integer.MAX_VALUE. We took maximum and minimum values of integer because it is given that n is smallest positive integer.

Following is the pseudo code for binary search in a sorted array. If f (x) is negative, we will increment our lower bound to mid+1. Similarly, if f(x) is positive, we decrement the higher bound to mid. And then we call the findFirstPositive function recursively after updating low and high values. Base case is when low and high are equal. After reaching base case we return low because this will be the index of first positive number.

Pseudo Code:

```
def findFirstPositive (f, low, high):
        if (low = high)
                return low
        mid = (low + high) / 2
        if (f(mid) <= 0):
                return findFirstPositive (f, mid+1, high)
        else:
                return findFirstPositive (f, low, mid)
findFirstPositive (f, Integer.MIN_VALUE, Integer.MAX_VALUE)
```

Executable Code:

We wrote the following code assuming that function values are represented in the form of a sorted array.

edit fork download copy

```
1.  f = [-4, -3, -2, -1, 0, 1, 2]
2.
3.  def find_n_gt_zero(input_list, low = None, high = None):
4.      if low == high:
5.          return low                  # given there is atleast one element > 0
6.      mid = int((low + high) / 2)
7.      if f[mid] <= 0:
8.          return find_n_gt_zero(input_list, low=mid + 1, high=high)
9.      else:
10.         return find_n_gt_zero(input_list, low=low, high=mid)
11.
12. smallest_n_gt_zero = find_n_gt_zero(f, low = 0, high = len(f))
13. print(smallest_n_gt_zero)
```

Success #stdin #stdout 0.03s 8960KB comments (0)

stdin copy

```
Standard input is empty
```

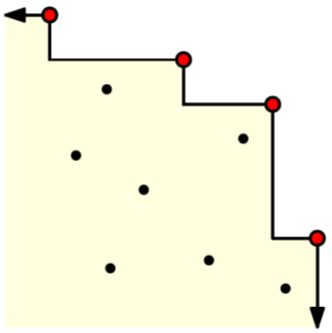
stdout copy

```
5
```

Time Complexity:

Above algorithm is nothing but a binary search in sorted array. We know that binary search takes O (log(n)) time complexity. Thus, above algorithm will run in O (log(n)) time complexity.

Problem 2. Cascading Waterfalls (27 Points)



A set P of 11 points, where $CW(P)$ has four points shown in red.

Given two points $p = (p.x, p.y)$ and $q = (q.x, q.y)$ in the plane, we say that q *dominates* p if $q.y > p.y$ and $q.x > p.x$. Given a set P of n points in the plane, the *cascading waterfall* of P, denoted $CW(P)$, is the subset of points from P which are *not* dominated by any other point in P, where the points in $CW(P)$ are listed in sorted order by x-coordinate value. Intuitively, the points in $CW(P)$ look like the boundary of a cascading waterfall. For an example, see the above figure.

Describe and analyze a *divide and conquer* algorithm to compute $CW(P)$ in $O(n \log n)$ time. Note there is more than one way to solve this problem, but your solution must use divide and conquer. (Note you can sort the points if you wish, but this does not count as using divide and conquer.) For simplicity assume any two points in P have distinct x and y coordinate values.

Intuition:

Given a random set of points, we first sort the points by x-coordinate values. Then we get the subset of points CW(P) which are dominated by other points by performing a divide and conquer approach where we recursively divide the points into two regions and merge them back depending on the condition that points in right half of our division do not dominate the point on the left half of our division and propagate the merged points back up the recursion tree.

Pseudo Code:

```
def merge_points(left_sub_points, right_sub_points):
        left_sub_points_len = len(left_sub_points)
        ptr_l = 0
        ptr_r = 0
        merged_points = []
        while ptr_l < left_sub_points_len:
                if left_sub_points[ptr_l][1] < right_sub_points[ptr_r][1]:
                        break
                merged_points.append(left_sub_points[ptr_l])
                ptr_l += 1
        merged_points.extend(right_sub_points)
        return merged_points
```

```
def divide_and_conquer(sorted_coords, low, high):
        if low == high:
                return [sorted_coords[low]]
        mid = int((low + high) / 2)
        left_sub_points = divide_and_conquer(sorted_coords, low, mid)
        right_sub_points = divide_and_conquer(sorted_coords, mid + 1, high)
        merged_points = merge_points(left_sub_points, right_sub_points)
        return merged_points
```

Executable Code:

edit fork download copy

```
1.  def merge_points(left_sub_points, right_sub_points):
2.      left_sub_points_len = len(left_sub_points)
3.      ptr_l = 0
4.      ptr_r = 0
5.      merged_points = []
6.      while ptr_l < left_sub_points_len:
7.          if left_sub_points[ptr_l][1] < right_sub_points[ptr_r][1]:
8.              break
9.          merged_points.append(left_sub_points[ptr_l])
10.         ptr_l += 1
11.     merged_points.extend(right_sub_points)
12.     return merged_points
13.
14. def divide_and_conquer(sorted_coords, low, high):
15.     if low == high:
16.         return [sorted_coords[low]]
17.     mid = int((low + high) / 2)
18.     left_sub_points = divide_and_conquer(sorted_coords, low, mid)
19.     right_sub_points = divide_and_conquer(sorted_coords, mid + 1, high)
20.     merged_points = merge_points(left_sub_points, right_sub_points)
21.     return merged_points
22.
23. coords = [(5,5), (14,1), (2,4), (12,5), (11,4), (9,7), (7,3)]
24. sorted_coords = sorted(coords, key = lambda k: k[0])
25. merged_points = divide_and_conquer(sorted_coords, 0, len(sorted_coords) - 1)
26. print("Cascading Points:", merged_points)
27.
```

Success #stdin #stdout 0.02s 8976KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
Cascading Points: [(9, 7), (12, 5), (14, 1)]
```

Time Complexity:

Above algorithm is similar to merge sort and we know that merge sort runs in O (n log(n)). Thus, time complexity of above algorithm is O (n log(n)).

Problem 3. Inversions (27 points)

An *inversion* in an array $A[1 \ldots n]$ is a pair of indices $\{i, j\}$ such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

Intuition:

We can intuitively say that if array is sorted in ascending order, then number of inversions will be 0. We can just modify the merge sort algorithm to get number of inversions. This can be done by initializing a counter and this counter is incremented when we encounter smaller value in right subpart.

We can get the inversion count by summing up inversion counts in left half, right half and inversion counts across both halves. We can get inversion counts from left and right halves using divide and conquer. Now we only need to get the inversion count across. This is done when a smaller element is found in the right half. When smaller element is found in right half then we increment the counter by number of larger elements found in the left half. Then we get the final count by adding all the inversions in left, inversions in right and inversions across.

Pseudo Code:

```
def merge_points(left_points, right_points, count_inv_l, count_inv_r):
    ptr_l = 0
    ptr_r = 0
    left_points_len = len(left_points)
    right_points_len = len(right_points)
    merged_points = []
    count_invs = 0
    while ptr_r < right_points_len:
        if ptr_l == left_points_len:
            break
        if left_points[ptr_l] < right_points[ptr_r]:
            merged_points.append(left_points[ptr_l])
            ptr_l += 1
        else:
            merged_points.append(right_points[ptr_r])
            count_invs += (left_points_len - ptr_l)
            ptr_r += 1
    if ptr_r != right_points_len:
        merged_points.extend(right_points[ptr_r:right_points_len])
    if ptr_l != left_points_len:
        merged_points.extend(left_points[ptr_l:left_points_len])
    total_count_inversions = count_invs + count_inv_l + count_inv_r
    return total_count_inversions, merged_points

def divide_and_conquer(points, low, high):
    if low == high:
        return 0, [points[low]]
    mid = int((low + high) / 2)
```

count_inversion_l, left_inversions = divide_and_conquer(points, low, mid)
count_inversion_r, right_inversions = divide_and_conquer(points, mid + 1, high)
count_total, total_inversions = merge_points(left_inversions, right_inversions, count_inversion_l, count_inversion_r)
return count_total, total_inversions

Executable Code:

We return inversion count and also sorted points from merge_points function to keep track of the merged points as well.

edit fork download copy

```
1.  def merge_points(left_points, right_points, count_inv_l, count_inv_r):
2.      ptr_l = 0
3.      ptr_r = 0
4.      left_points_len = len(left_points)
5.      right_points_len = len(right_points)
6.      merged_points = []
7.      count_invs = 0
8.      while ptr_r < right_points_len:
9.          if ptr_l == left_points_len:
10.             break
11.         if left_points[ptr_l] < right_points[ptr_r]:
12.             merged_points.append(left_points[ptr_l])
13.             ptr_l += 1
14.         else:
15.             merged_points.append(right_points[ptr_r])
16.             count_invs += (left_points_len - ptr_l)
17.             ptr_r += 1
18.     if ptr_r != right_points_len:
19.         merged_points.extend(right_points[ptr_r:right_points_len])
20.     if ptr_l != left_points_len:
21.         merged_points.extend(left_points[ptr_l:left_points_len])
22.     total_count_inversions = count_invs + count_inv_l + count_inv_r
23.     return total_count_inversions, merged_points
24.
25.
26. def divide_and_conquer(points, low, high):
27.     if low == high:
28.         return 0, [points[low]]
29.     mid = int((low + high) / 2)
30.     count_inversion_l, left_inversions = divide_and_conquer(points, low, mid)
31.     count_inversion_r, right_inversions = divide_and_conquer(points, mid + 1, high)
32.     count_total, total_inversions = merge_points(left_inversions, right_inversions, count_inversio
    n_l, count_inversion_r)
33.     return count_total, total_inversions
34.
35. points = [1,2,3,5,4,-1,-2]
36. low = 0
37. high = len(points) - 1
38. count_inversions, sorted_list = divide_and_conquer(points, low, high)
39. print(count_inversions)
```

Success #stdin #stdout 0.02s 9124KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
12
```

Time Complexity:

Above algorithm runs in the same way as merge sort and so running time complexity will be same as that of merge sort. Thus, time complexity will be O (n log(n))

Problem 4. Selection in Sorted Arrays (30 points)

As input you are given two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots m]$ of integers. For simplicity assume that all $n + m$ values in the arrays are distinct. Describe and analyze an algorithm to find the kth ranked value in the union of the two arrays (where the 1st ranked value is the smallest element). The running time of your algorithm should be $O(\log(n + m))$. Note: If your analysis happens to give an $O(\log(n) + \log(m))$ running time, observe that $\log(n) + \log(m) = O(\log(n + m))$.
[Hint: Recall Binary search and/or Quickselect.]

Intuition:

We have to find k^{th} indexed element in the union of two given arrays. We can do this by merging both the arrays initially and then finding the k^{th} element. But merging both the arrays will take O (n + m) time complexity. We have to solve this question in O (log (n + m)) time complexity.

Firstly, we take mid points of both the arrays (n/2 and m/2) and then if summation of these two mid indexes is less than k, then we can remove first half of the array which has lower mid value and update k to k – mid. Now we run the recursive main function after updating k and updating the array that has lower mid value.

If summation of both mid points is greater than k, then we remove higher half elements from Array that has higher mid value and run the recursive main function with same k and updating the array that has higher mid value.

Finally, whenever one of the arrays become null, we directly take the updated k^{th} element from the array that is not null.

Pseudo Code:

```
def kthLargest(arr1, arr2, k):
    if (arr1.length = 0):
        return arr2[k]
    else if (arr2.length = 0):
        return arr1[k]
    mid1 = arr1.length / 2
    mid2 = arr2.length / 2
    if (mid1 + mid2 < k):
        if (arr[mid1] > arr[mid2]):
            return kthLargest(arr1[], arr2[mid2+1:], k-mid2-1)
        else:
            return kthLargest(arr1[mid1+1:], arr2[], k-mid1-1)
    else:
        if (arr[mid1] > arr[mid2]):
            return kthLargest(arr1[:mid1], arr2[], k)
```

else

return kthLargest(arr1[], arr2[:mid2], k)

Executable Code:

edit fork download copy

```
def kth_rank_element(arr_list_1, arr_list_2, k):
    if len(arr_list_1) == 0:
        return arr_list_2[k]
    elif len(arr_list_2) == 0:
        return arr_list_1[k]
    mid_a1 = len(arr_list_1) // 2  # integer division
    mid_a2 = len(arr_list_2) // 2
    if mid_a1 + mid_a2 < k:
        if arr_list_1[mid_a1] > arr_list_2[mid_a2]:
            return kth_rank_element(arr_list_1, arr_list_2[mid_a2+1:], k - mid_a2 - 1)
        else:
            return kth_rank_element(arr_list_1[mid_a1+1:], arr_list_2, k - mid_a1 - 1)
    else:
        if arr_list_1[mid_a1] > arr_list_2[mid_a2]:
            return kth_rank_element(arr_list_1[:mid_a1], arr_list_2, k)
        else:
            return kth_rank_element(arr_list_1, arr_list_2[:mid_a2], k)

arr_list_1 = [1,2,3,4,8]
arr_list_2 = [5,6,7]
rank = 2
pivot = rank - 1
kth_rank = kth_rank_element(arr_list_1, arr_list_2, pivot)
print(kth_rank)
```

Success #stdin #stdout 0.02s 8980KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
2
```

Time Complexity:

Time complexity will be T(n/2) + T(m/2), since worst case scenario we will iterate through each half of both arrays. So, time complexity will be O (log(n) + log(m)) which is O (log (n + m))