Homework 4

Tags	Homework
Date	

Problem 1 — Exercise 15 on page 196

ALGORITHM

```
let I be the set of all intervals
let schedule be most efficient set of intervals
sort I from earliest finish time first to latest

for each interval i in I
  if i is unmarked and finishes the earliest
    let marked = set of all elements that intersect with i
    for all other intervals j
      if interval j overlaps with i
        mark j
    select element from marked that finishes the latest and add to schedule

 return schedule
```

PROOF

- for each interval, this algorithm finds the greatest overlapping interval that extends from between the start of interval i to end of interval j
- each other interval j that overlaps with i would be marked to skip in the main loop because we have already visited them and chosen the interval with the latest finish time
- by way of contradiction, assume that we don’t find the efficient set of intervals
    - that means that at some point there is an interval j that isn’t marked or isn’t the interval with the latest end time
    - we’ve showed that each interval j will be marked if it overlaps with i
    - from those marked intervals, we choose the latest end time and the next iteration of the loop starts with that interval to find the next largest interval because we sort I
    - therefore we have to select the greatest maximizing interval times

- that means we are left with the most efficient set of intervals

RUNTIME

- if the set of intervals aren't already sorted, sorting takes at worst $O(nlog(n))$
- we 2 for loops that iterates through each interval in I, but is nested so we have $O(n^2)$
- each if operation and marking is $O(1)$ because they are all adds / comparison operators
- therefore we have $O(n^2 + nlog(n)) \Rightarrow O(n^2)$

Problem 2 — Exercise 16 on page 19

SETUP

We are given that each suspicious transaction x_i has a time interval of $[t_t - e_i, t_t + e_i]$.

ALGORITHM

```
sort timings such that x_1  ≤ x_2 ≤ ... ≤ x_n.
for all transactions i in x
  if there exists unmatched intervals that contain x_i
    let marked = overlapped intervals that contain x_i
    for all transactions j
      add j to marked
    match x_i with t_j that ends the earliest
  else
    return false

return true
```

PROOF

- By way of contradiction, assume the list should have a perfect matching but there exists an unmatched interval that contains x_i.
- If that unmatched interval matches with t_j but is supposed to match with t_k, we know that $t_k + e_k < t_j + e_j$.
- If t_k is already mapped to another x call it $x_q$that is not x_i, the algorithm would then have instead matched t_k to x_i and because $t_k + e_k < t_j + e_j$ and x_q to t_j
- therefore by contradiction, if a list contains a perfect matching this algorithm will find that

RUNTIME

- The algorithm loops through each of the transactions twice
  - first to iterate through each transactions for the first time
  - then to find with time t_i that matches with that transaction
  - is nested ⇒ n * n = n^2
- Hence worst case, it will take $O(n^2)$

Problem 3 — Exercise 4 on Page 247

ALGORITHM

```
define a convolution with
  a = vector of all charges for q_0 to q_n
  b = vector (1/n^2, 1/(n-1)^2, ... 1, 0, -1, -1/(n-1)^2, -1/n^2)
where each element in the convolution is satisfied by
  return C * [summation from i < j (q_i*q_j / (j-i)^2) - summation from j < i (q_i*q_j / (i-j)^2)]
```

PROOF

- The convolution is a method to iterate through each of the given charges without the use of nested for loops and calculate the polynomial, instead we calculate each product as wanted by the problem as a function created by the convolution
- by definition of a convolution, this gives us the resulting set of numbers that correlate to the forces F_j after each is multiplied by C.

RUNTIME

- We have to iterate for each charge from q_0 to q_n hence we have O(n) operations
- And for each of those operations, we calculate the resulting force in a divide and conquer way using Fourier method of convolutions, hence for each operation n we have a log(n) to calculate the force.
- Therefore, our runtime is $O(nlog(n)$

Problem 4 — Exercise 7 on page 248

SETUP

Let us define a local minimum if the probed value of a node x_v is less than all other nodes x_w that are connected to it by an edge.

ALGORITHM

```
recursive function that takes in a graph and level of nodes starting at root level
  middle_nodes = set of all level nodes not on border
  middle_probes = set of all probed values of level nodes not on the border
  for all nodes in middle_nodes
    add probed node to middle_probes

  if local minimum exists in middle_probes
    return corresponding node from middle_nodes
  else
    select minimum valued node min from middle_nodes
    recurse and pass in n x n/2 graph for the left and right side to min
```

PROOF

We only need to find A local minimum not a specific local minimum. Henceforth, this algorithm finds that in the following manner

- a simple traversal is done where we probe up to O(n) by probing the middle nodes in each level of the grid
- assume by way of contradiction that we cannot find find a local minimum within the middle nodes in between the border
    - that means on the outermost rows and columns on the grid, there contains at least one node not in on the border that is adjacent to a node on the border
    - that means a global minimum cannot exist on the border, implying that there must exist at least one local minimum within the middle nodes
- therefore, we must iterate through each of the middle nodes to find a local minimum
- if one isn’t found on the level but there exists a minimum valued node, there must exist a node to the left or ride of that node such that its probed value is lesser. henceforth, we recurse and eventually find a local minimum

RUNTIME

- setting the temporary arrays require O(1) time

- adding for each node of the level takes a worst case of of O(n) probes but it is less than that because we do not probe the border nodes and we stop continuation of probing once either a local minimum is found or continue probing but only on half the graph
- therefore, at worst case we have a total amount of O(n) probes

Problem 5 — Suppose you are given an array of sorted integers that has been circularly shifted **K** positions to the right. For example taking (1 3 4 5 7) and circularly shifting it 2 positions to the right you get (5 7 1 3 4). Design an efficient algorithm a number X in the array (linear time is trivial)

SETUP

This problem was confusing to understand because of the last line, so after clarifying with the TA I'm assuming the problem is asking to find K.

ALGORITHM

```
let arr = array of sorted integers circularly shifted K positions to the right
function takes in arr and a temp arr
let n = size of arr

let arr[m] be the middle element
let arr[a] be the first element

if size of temp arr = 1:
  let pos = position of reminaing element in original arr
  return pos + 1

if arr[a] > arr[m]
  recurse for temp arr = [arr[m], arr[-1]] where -1 is the last element
else
  recurse for temp arr = [arr[a], arr[m]]
```

PROOF

- By way of contradiction, assume that the result found is a position that is not equal to K
- that would mean as we recurse and half the array using binary search, we would run into a situation where the wrong half is recursed over

- this cannot happen because we are given that the list is sorted — henceforth, if the middle element is less than the first element, that means the shifted values end before the middle element, or if the middle element is greater or equal than than the first element, then we are still over the shifted element or its shifted into the right side of the list
- therefore, the wrong half could never be recursed over, and we would keep continuing until we are left with a list with two elements, and the position of whichever element is not in sorted order + 1 is the number of circularly shifted elements.

RUNTIME

- The algorithm uses a modified version of binary search with extra level comparisons to determine whether to search through the left or right side of the split arrays.
- We know that binary search has runtime O(log n) and the rest of the recursive function is built of if statements with O(1) time
- Hence, we have worst case runtime $O(log(n))$

Problem 6 — Consider **d** sorted array of integers each containing n1, n2, .. nd numbers. The numbers ni's are arbitrary. The total number of all elements is n (sum of all ni's). Design an O(n log d) algorithm that merges all arrays into one sorted list.

ALGORITHM

```
function merge(arr1, arr2)
  let n1 = size of arr1
  let n2 = size of arr2
  let arr = size of n1 + n2
  let k = index to insert in arr
  while i < n1 and j < n2
      if arr1[1] < arr2[j]
        set arr[k] to arr[i]
        increment i
      else
        set arr[k] to arr[j]
        increment j
      increment k

  if there are remaining elements in arr1 or arr2, add them individually to arr
    at the end

  return arr
```

```
function mergelists(arr, mid): // initially passes d and 0
  find middle index of all lists in arr and set to mid
  let left = recurse for all lists with indexes less than and equal to middle index
  let right = recurse for all lists with indexes greater than middle index

  return merge(left,right)
```

PROOF

- This proof divides each of the lists in d into a single list, and then takes two lists at a time, merge sorts them, and repeats the process for the rest of the pairs
- using recursion, we split the left side and the ride side of the main list d to get the individual lists
- by way of contradiction assume that when merging these lists, it doesn't sort them correctly
    - that would mean that that some point a value in one array is greater than a value in another array, but is placed before it
    - that cannot occur because we increment through both arrrays and only add when one array's value is less than the value in the other array using a separate index counter k
    - that would mean that our merges sort each array regardless of their associated sizes
- after each merge, the number of arrays in arr decreases henceforth we achieve efficiency in maximizing the number of merges that can be made, and we are left with a sorted array that contains all the elements from each list in d

RUNTIME

- we start by merging arrays in groups of 2, leaving d/2 arrays. then in groups of two again, leaving d/4 arrays and so forth. similar to binary search, we half the number of arrays being merged for each operation
- that means that for all the elements there are n * log(k) lists for each merge of each list
- henceforth, the runtime is $O(nlog(d))$