CSE222: Assignment 1

Ananya Lohani (2019018) | Mihir Chaturvedi (2019061)

Problem 1: Q31, page 61

Part (a)

Algorithm

```
FindIndex(A, low, high):
    if low > high:
        return NONE
    mid ← (low + high)/2
    if A[mid] < mid:
        return FindIndex(A, mid + 1, high)
    else if A[mid] > mid:
        return FindIndex(A, low, mid - 1)
    else
        return mid
    endif
```

Time Complexity

The recurrence relation for the given algorithm is as follows:

$$T(n) = T(n/2) + c$$

Which gives us the asymptotic time complexity as $O(logn)$.

Correctness

We shall provide an inductive proof for the algorithm's correctness.

Base Case

<u>An array with 0 elements.</u>

If $n = 0$, $low = 1$ and $high = 0$. Since $low > high$, we return NONE which is correct.

Inductive Hypothesis

FindIndex works for an array of n elements.

Inductive Step

Proving that *FindIndex* works for an array of $n+1$ elements.

Because the array is sorted and all the elements are distinct, if $A[mid] < mid$, then $A[i] < i \ \forall \ i < mid$. Thus, we only need to search in the range $(mid+1, \ high)$. Since the range $(mid+1, \ high)$ is necessarily smaller than the range $(1, \ n+1)$, and we've assumed that *FindIndex* works for all arrays having less than $n+1$ elements, it works for the range $(mid+1, \ high)$.

If $A[mid] > mid$, we only need to search in the range $(low, \ mid-1)$ by a similar logic. If $A[mid] = mid$, we are done.

Part (b)

Algorithm

```
FindIndex(A):
    if A[1] = 1:
        return 1
    else:
        return NONE
    endif
```

Time Complexity

Constant time: $O(1)$

Correctness

Since the array is sorted in increasing order and all elements are distinct and we know that $A[1] > 0$, the only way to find an index i with $A[i] = i$ is if the very first element is 1 itself. If $A[1] > 1$, then $A[i] > i$ for all $i > 1$ and hence there is no such index for which $A[i] = i$.

Problem 2: Q12, page 50

Part (a)

The correctness can be proved by induction on the length of the array. Let *P(n)* be the hypothesis that the algorithm returns a sorted array. The case of $n = 1$ is trivial. The base case is $n = 2$. `Cruel` will divide it into 2 arrays with one element each and then call `Unusual` on the original array. `Unusual` will check if $A[1] > A[2]$ and swap the elements if it's true. Otherwise, we already have a sorted array. Hence this is correct. Now suppose this is true for all arrays of length 1, 2, 3... n-1.

Consider *P(n)*. The algorithm first splits the array into two (roughly equal) halves X and Y. The following cases are possible:

1. Largest element of $X \leq$ Smallest element of Y
2. Largest element of $X >$ Smallest element of Y

By the induction hypothesis, Cruel sorts X and Y correctly. Hence, the first case is returned correctly, and our final array is sorted. To prove the second case, we prove the correctness of the `Unusual` subroutine. Three cases arise:

Case 1: $X[i] > Y[j]$ for some $i \leq n/4$ *and* $j \leq n/4$

Case 2: $X[i] > Y[j]$ for some $n/4 < i \leq n/2$ *and* $j \leq n/4$

Case 3: $X[i] > Y[j]$ for some $n/4 < i \leq n/2$ *and* $n/4 < j \leq n/2$

We prove all the above cases by induction.

Base Case

`Unusual` merges an array of length 2 correctly.

Inductive Hypothesis

`Unusual` merges an array of length *i* correctly for $2 \leq i < n$.

Inductive Step

`Unusual` merges an array of length *n* correctly.

Case 1

The swap for-loop swaps the second half of X and first half of Y such that now $Y[j]$ is swapped with $X[k]$ for some $n/4 < k \leq n/2$, i.e. $Y[j]$ is in the second half of X.

Then `Unusual` is called on the first, second and middle halves of the array A.
By our induction hypothesis, `Unusual(`$A[1...n/2]$`)` returns a correctly merged array with the (earlier) $Y[j]$ in its correct position. `Unusual(`$A[n/2+1...n]$`)` also returns a correctly merged array. And finally, `Unusual(`$A[n/4+1...3n/4]$`)` orders the elements in the middle half correctly, hence giving us a correctly sorted array.

Case 2
In this case, the swap for-loop swaps the second half of X and first half of Y, thereby positioning all $X[i] > Y[j] \;\; n/4 < i \leq n/2 \; and \; j \leq n/4$, in correct halves.
By our induction hypothesis, `Unusual(`$A[1...n/2]$`)` and `Unusual(`$A[n/2+1...n]$`)` will sort the first and second halves respectively, separately. Finally, the middle half will be correctly sorted by the call to `Unusual(`$A[n/4+1...3n/4]$`)`.

Case 3
As with the previous cases, the swap for-loop will bring the $X[i]$ in question to the latter half of A.
By our induction hypothesis, the first half will be sorted with a call to `Unusual` on applied on the first half. The second half will be sorted, including the aforementioned $X[i]$, by the call to `Unusual(`$A[n/2+1...n]$`)`. The remaining sort-and-merge will be handled by the call to `Unusual` on the middle half.

Part (d)

`Unusual` is called with an array of size n. It then:

- does constant time operations in a loop of size $n/4$;
- recursively calls itself three times, each with size $n/2$;

The final recurrence relation is: $T(n) = 3T(n/2) + cn/4$
Which gives us the asymptotic time complexity of $O(n^{log_2 3})$ using Master Theorem.

Part (e)

`Cruel` is called with an array of size n. It then:

- recursively calls itself two times, each with size $n/2$;
- Calls `Unusual` with size n;

The final recurrence relation is: $T(n) = 2T(n/2) + O(n^{log_2 3})$

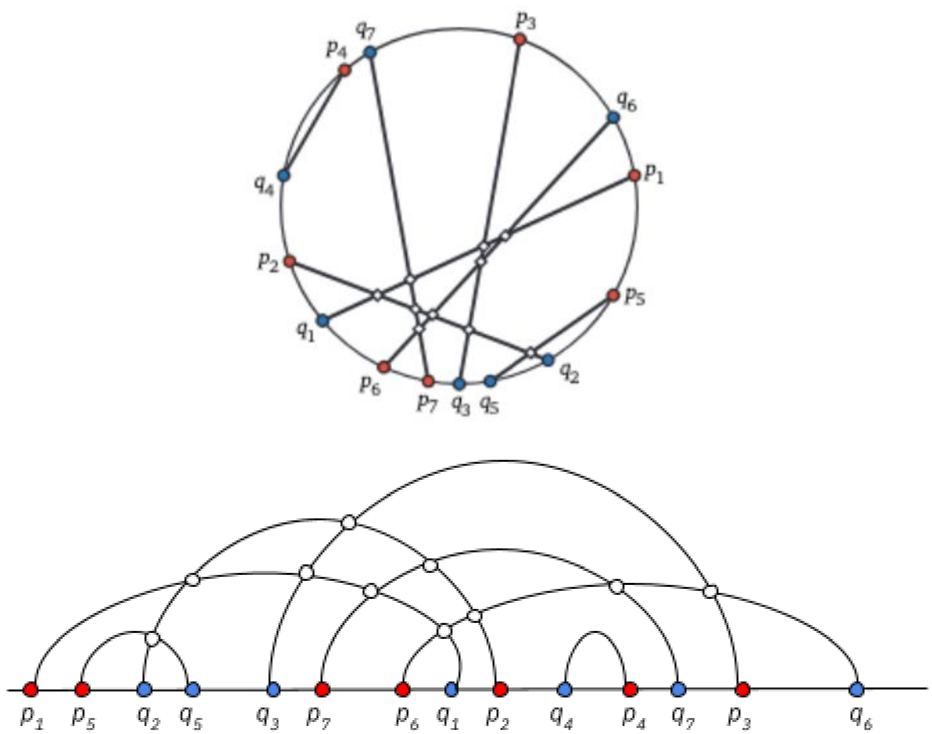
Which gives us the asymptotic time complexity of $O(n^{\log_2 3})$ using Master Theorem.

Problem 3: Q14, page 51

Part (b)

We have directly formulated an algorithm with a runtime of O(nlogn) in Part (c)

Part (c)



Let the number of points on the circle be $2n$. We pick any two consecutive points and break the circle from between them. Now we consider the straight line that we get after fully opening and spreading the circle, preserving the relative positions of the points. Since the relative positions are the same, the number of intersections between the chords on the circle are the same as the intersections after spreading them out on the line. Let us label all points on the line as 1 to n in consecutive order (e.g. $p_1 \rightarrow 1$, $p_5 \rightarrow 2$, $q_2 \rightarrow 3$ and so on).

We generate an array A containing n pairs of the form (x, y) (such that x < y) of the 2 end points of a chord. In the example we have taken above, we get
$A = [(1, 8), (2, 4), (3, 9), (5, 13), (6, 12), (7, 14), (10, 11)]$.
We sort A by the x and y coordinates to get B and C respectively.
We have devised the following algorithm that counts the number of intersections between the (x, y) pairs in A.

Algorithm

```
CountIntersections(A):
      B ← sortByX(A)
      C ← sortByY(A)
      count ← 0
      i ← 1
      while i <= size(B):
            j ← BinarySearchLessThan(B, B[i].y, "x")
            if j > i:
                  count ← count + (j - i)
                  k ← BinarySearchLessThan(C, B[i].y, "y")
                  h ← BinarySearch(B, C[k])   // regular BinarySearch
                  if h > i:
                        count ← count - (h - i)
                  endif
            endif
      endwhile
      return count

BinarySearchLessThan(A, target, coord):
      low ← 1
      high ← size(A)
      index ← -1
      while low <= high:
            if coord = "x":
                  value ← A[mid].x
            else:
                  value ← A[mid].y
            endif
            mid ← (low + high) / 2
            if value > target:
                  high ← mid - 1
            else if value < target:
                  index ← mid
                  low ← mid + 1
            else:
                  high ← mid - 1
            endif
      endwhile
      return index
```

Time Complexity

We first sort A according to x and y coordinates to get B and C respectively. Sorting is done in $O(nlogn)$, time. The `CountInversions` function loops our initial array `B` of size n, and in each iteration, calls `BinarySearchLessThan` twice and `BinarySearch` once, each on an array of size n. Since binary search has a time complexity of $O(logn)$, the final relation is:

$$T(n) = O(nlogn) + O(n) \times O(logn)$$
$$\Rightarrow T(n) = O(nlogn)$$

Correctness

Claim 1

Consider an iteration of the while loop of CountIntersections. Then,

Case 1: If there's no such j for which $B[j].x < B[i].y$, there are no intersections between the pairs $B[i]$ and $B[j]$.

Case 2: If $B[j].x < B[i].y$ for some j, then every point in range $B[1..j]$ intersects with $B[i]$, except points of range $B[1...k]$ if there exists $0 < k \leq j$ such that $B[k].y < B[i].y$.

Proof

Case 1: If $B[j].y > B[j].x > B[i].y > B[i].x$, there is no overlapping between pairs $B[i]$ and $B[j]$.

Case 2:

- If $B[j].x < B[i].y$ and $B[j].y > B[i].y$, and we know from the sorted order of B that $B[i].x < B[j].x$, we observe that the single point $B[j].x$ will be positioned between the points of $B[i]$, and their connecting chords will thus create a single intersection.
- If $B[j].x < B[i].y$ and $B[j].y < B[i].y$, we observe that both the points of $B[j]$ will be positioned between the points of $B[i]$, and their connecting chords will not overlap and no intersection will be created.

Claim 2

Consider the value of the variable i at the start of any iteration of the while loop. Then the variable count already accounts for the intersections of $B[1,\ 2, \ldots i-1]$ with all other pairs in B.

Proof

The correctness of `BinarySearchLessThan` follows from the correctness of Binary Search. We will prove the correctness of `CountIntersections` by induction on the number of iterations m of the while loop.

Let $P(m)$: At the start of the m^{th} iteration of the while loop, count accounts for the number of intersections of $B[1,\ 2, ...m-1]$ with all the other pairs in B.

Base Case

When $m = 1$, $count = 0$; Hence, $P(1)$ is true.

Inductive Hypothesis

Suppose $P(m-1)$ is true for some $m \geq 2$.

Inductive Step

Proving that $P(m)$ is true.

Case 1

If there's no such j for which $B[j].x < B[i].y$, `BinarySearchLessThan` returns -1 in j, which means $j < i$. In this case, count is not incremented because there are no intersections between $B[i]$ and the rest of the pairs. This proves $P(m)$.

Case 2

- `BinarySearchLessThan` returns the largest j for which $B[j].x < B[i].y$. Clearly, $B[k].x < B[j].x \Rightarrow B[k].x < B[i].y$, for $k < j$ also. Since count already accounts for all the possible intersections of pairs in range $B[1,\ 2,\ \ldots,\ i-1]$, we must count only the *new* intersection and it to the count variable, to prevent double-counting. Thus, we simply add $(j-i)$ to count.
- The above count *over*counts the possible intersections, since there might be cases where no intersection is produced, as proved in Claim 1's proof of case 2. These include pairs for which $B[j].y < B[i].$. We call `BinarySearchLessThan` on C (array sorted by y) to find the largest index k for which $C[k].y < B[i].y$. Suppose the index of $C[k]$ in B is h. To prevent double counting again, we subtract $(h-i)$ from count. This gives us the intersections of pair $B[i]$ with all other pairs, hence proving $P(m)$.