ESO207: Theoretical Assignment 1 - Part 2

Debaditya Bhattacharya

February 12, 2021

Name: Debaditya Bhattacharya **Roll No:** 190254
Email id: debbh@iitk.ac.in **Hackerrank Id:** debbh922

Problem 5

a) Let $f(n) = \min(n^2, 10^{12})$, $g(n) = 1$ and $M = 10^{12} + 1$. Then,

$f(n) \leq Mg(n), \forall n > 1, n \in \mathbb{N}$
$\implies f(n) = \mathcal{O}(g(n)) = \mathcal{O}(1)$

Hence Proved

b) Let $f(n) = n^2 + n \log n$. Now,

$\log(n) < n, \forall n \in \mathbb{N}$
$\implies n \log n < n^2, \forall n \in \mathbb{N}$
$\therefore f(n) = n^2 + n \log n < 2n^2 \implies f(n) = \mathcal{O}(n^2)$

Hence Proved

c) Proof by contraction.

Let us assume that $f(n) = n^3 + 3n^2 + 8 = \mathcal{O}(g(n)) = \mathcal{O}(n^2)$, where $g(n) = n^2$ then,

$\exists M, n_0 \in \mathbb{N}$ such that $f(n) \leq Mn^2, \forall n \geq n_0, n \in \mathbb{N}$
Consider $k = Mn$.
$\therefore f(Mn) = M^3n^3 + 3M^2n^2 + 8 \geq M^2n^2 = g(Mn), \forall n > 1, n \in \mathbb{N}$ which is a clear contraction to our assumed statement.

Hence, $f(n) = n^3 + 3n^2 + 8 \neq \mathcal{O}(n^2)$

d) Proof by contraction.

Let us assume that $f(n) = 4^n = \mathcal{O}(2^n)$Then,

$\exists M, n_0 \in \mathbb{N}$ such that $f(n) \leq M2^n, \forall n \geq n_0, n \in \mathbb{N}$
$2^{2n} \leq M2^n$
$2^n \leq M$
Let $n_1 > n_0 + \log M$ then,
$2^{n_1 - n_0} > M$
$for n_1 > n_0$ contraction.
$\therefore$ our assumption is wrong.
Hence, $4^n \neq \mathcal{O}(2^n)$

e) $n! = n \cdot (n-1) \cdot (n-1) \cdot \dots \cdot 2 \cdot 1 \leq n \cdot n \cdot \dots \cdot n \cdot n = n^n, \forall n \in \mathbb{N}$
$\therefore \log(n!) \leq log(n^n) = n \log n, \forall n \in \mathbb{N}$ $\therefore \log(n!) = \mathcal{O}(n \log n)$

Hence Proved

Problem 6

Description:

The algorithm is based off the divide and conquer paradigm. To understand my approach better consider the case where the array B is flipped($B'[k] = B[n-1-k]$). Then the algorithm reduces to finding the index at which $A[k] = B'[k]$. This can be done with a divide and conquer algorithm which will take $T = \mathcal{O}(\log n)$ time. As the arrays are sorted we use the condition (1) $A[k] > B[n-1-k]$. If (1) is true, set $R = k-1$ else set $L = K+1$. The while loop is terminated when $L = R$ and returns a -1 (failure) or when $A[k] = B[n-1-k]$ and returns the value of k.

Algorithm:

```
function find_symmetric(A, B, n){
    L <- 0;
    R <- n-1;
    found <- false;
    i <- -1;
    while (!found){
        m <- (L+R)/2;
        if (A[mid] = B[n-1-mid]){           \\Found
            i <- mid;
            found <- true;
        }
        else {
            if (L == R){                    \\Not found
                i <- -1;
                found <- true;
            }
            if (A[mid]>B[n-1-mid]){         \\Condition
                R <- mid-1;
            }
            else {
                L <- mid+1;
            }
        }
    }
    return i;
}
```

Listing 1: Find symmetric

Proof of correctness:

The two arrays A and B are sorted (assume increasing without loss of generality)

Assertion $P(i)$: If there is a common element, it exists in the set $\{A[k] : L \leq k \leq R\}$
Base case (i = 0): $L = 0$, $R = n-1$. If there exists a common element it belongs to A and B.

Define $B'[k] = B[n-1-k]$. As B is sorted in increasing order, B' is sorted in decreasing order.

Induction step: Assume $P(i)$ to be true. Then consider the middle element $m = (L+R)/2$. If $A[m] < B'[m]$ then $A[k] < B'[k]$ for all $k < m$ as A is increasing and B' is decreasing. Then let $R = m-1$ and divide again. If $A[m] > B'[m]$ then $A[k] > B'[k]$ for all $k > m$ as A is increasing and B' is decreasing. Then let $L = m+1$ and divide again. If $A[m] = B'[m]$ we have found our point. If $L = R$ and $A[m] \neq B'[m]$ then there exists no such point. Return -1.

Problem 7

Description:

Initially lines A and B are parallel but oriented in some direction. Find slope θ and apply coordinate transformation ($T = \mathcal{O}(n)$) to orient new axes such that lines are parallel to new x axis. Then for each line $y' = constant$. Sort the points in time $T = \mathcal{O}(n \log n)$ according to their x coordinate. Traverse along the two lines together and keep comparing to get points a_{min} and b_{min} ($T = \mathcal{O}(n)$) apply inverse coordinate transformation to bring back into original form.

Algorithm:

```
function return_closest(A, B){
    theta <- Calculate slope from 2 points of A such that A is parallel to new x axis.
    R <- Rotation matrix from theta
    C <- Ra for a in A                                  \\T = O(n)
    D <- Rb for b in B                                  \\T = O(n)
    A_s <- Sort C according to x values (y is constant)  \\T = O(nlogn)
    B_s <- Sort D according to x values (y is constant)  \\T = O(nlogn)
    i <- 0;
    j <- 0;
    min_distance = MAX_int;
    a=0;
    b=0;
    while (i!=n-1 && j!=n-1){                           \\Hasn't reached edges
        if(distance(A_s[i],B_s[j])<min_distance){       \\T = O(2n) = O(n)
            min_distance = distance(A_s[i],B_s[j]);
            a = i;
            b = j
        }
        if(A_s[i]<B_s[j] && i!=n-1){
            i++;
        }elseif(j!=n-1){
            j++;
        }
    }
    R' <- Inverse rotation matrix.      \\Inverse rotation matrix
    Return RA_s[a], RB_s[b];            \\Rotate found points back into desired coordinates
}
```

Listing 2: Closest points

Proof of correctness:

We first apply a rotation to bring the points A and B to be parallel to the new axis. This allows for easier manipulation as one of the coordinates is made constant. The slope can be found in $\mathcal{O}(1)$ time. The matrix R can be calculated in $\mathcal{O}(1)$ time. Multiplication of the matrix R with a point p takes $\mathcal{O}(1)$ time. Hence RA (R applied to all elements in A) takes time $\mathcal{O}(n)$

We consider $C = RA$ and $D = RB$. We then proceed to sort C and D according to their x values. This process takes $\mathcal{O}(n \log n)$ time by using the merge sort algorithm. $A' = sort(C)$ and $B' = sort(D)$.

Reduced problem: Find the pair of elements in two arrays A' and B' such that $(a'_x - b'_x)^2 + (a'_y - b'_y)^2$ is minimized. Equivalent to minimizing $|a'_x - b'_x|$, $a'_x \in A', b'_x \in B'$

For $i, j = 0$ start comparing x component of elements of A' and B', as stated above and store the min distance and corresponding elements in $min\_distance$ and a, b, while iterating. If $A'[i] < B'[j]$ then $i++$, else $j++$. The loop will terminate in $T = \mathcal{O}(2n) = \mathcal{O}(n)$ as in each iteration i or j increases and we visit each element in A' and B' only once.

Let R' be the inverse rotation. Return $R'a$, $R'b$.

Problem 8

Description:

Preprocessing:
Sort the array A into A' in $T = \mathcal{O}(n \log n)$. Traverse along the array and count the different number of each element. Store these unique values and counts in two arrays *key* and *count*. This will take $T = O(n)$ time as the A' is sorted, and space $S = O(n)$. The array *key* will already be sorted because A' was sorted.

For problem 1. we calculate the cumulative sum *cumsum* of *counts*.

For problem 2. calculate the range maxima (like range minima) data structure of the array *count* as we discussed in class and store it as *table*. Instead of storing the maximum values in this table explicitly, store index i corresponding to element in *counts* (can be retrieved in $T = \mathcal{O}(1)$ Therefore overall preprocessing time complexity is $T = \mathcal{O}(n \log n)$ and space complexity is $S = \mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$

Example of Keys and counts:

A	1.5	1.5	-2	0	2	0	0	3.2	0	3	2.4	-1	1	1	1.7	1.5	1.2	-3	-2.1	-5
A'	-5.0	-3.0	-2.1	-2.0	-1.0	0.0	0.0	0.0	0.0	1.0	1.0	1.2	1.5	1.5	1.5	1.7	2.0	2.4	3.0	3.2

keys	-5.0	-3.0	-2.1	-2.0	-1.0	0.0	1.0	1.2	1.5	1.7	2.0	2.4	3.0	3.2
counts	1	1	1	1	1	4	2	1	3	1	1	1	1	1

Processing:
(1) Given $[a, b]$ We can find a', b' in *keys* using binary search in time $T = \mathcal{O}(logn)$, (a', b' are indexes of the elements in the *keys*). We then evaluate the total number of elements as $cumsum[b'] - cumsum[a' - 1]$. Handle for edge case if $a' = 0$ return $cumsum[b']$

(2) Given $[a, b]$ we can find a', b' in *keys* by using binary search in time $T = \mathcal{O}(logn)$ (a', b' are indexes of the elements in the *keys*). We then use the range maxima approach to get the *max_index* in between a' and b'. Return $keys[max\_index]$

Algorithm:

```
function preprocess(A, n){
    A' <- sort(A)                                           \\merge sort T = O(nlogn)

    current_key = A'[0]

    keys <- array same size as A
    count <- array same size a A initialized as zeros.
    k <- 0;                                                 \\for counting keys
    for (int i = 0; i <n; i++){                             \\generating keys and counts T = O(n)
        current_key;
        if (current_key!=keys[k]){
            k++;
            keys[k] = current_key;
        }
        count[k]++;
    }
    keys <- keys, but now trimmed to size k;                \\Trim keys T = O(n)
    counts <- counts, but now trimmed to size k;            \\Trim counts T = O(n)

    cumsum <- array of size(K) initialized to zero.
    cumsum[0] <- counts[0];
    for (int i = 1; i < k; i++){                            \\Generate cumulative sum of counts
        cumsum[k] = cumsum[k-1] + counts[0];                \\ T = O(n)
    }

    table <- generate range maxima table as in class, but with index of keys instead of value.  \\T = O(nlogn), S = O(nlogn)

    return keys, counts, cumsum, table;
}

function total_elements(a, b, keys, cumsum) {
    a' <- find index a' in keys such that keys[i] >= a for i >= a'; \\Binary search
    b' <- find index b' in keys such that keys[i] <= b for i <= b'; \\Binary search
    if (a' == 0)
        return cumsum[b'];
    else
```

```
            return cumsum[b'] - cumsum[a'-1]
    }
    function max_occour(keys, counts, keys', counts', table){
        a' <- find index a' in keys such that keys[i] >= a for i >= a'; \\Binary search
        b' <- find index b' in keys such that keys[i] <= b for i <= b'; \\Binary search

        range_max_idx <- fetch range maximum from table using table and counts. \\T = O(1)

        return keys[range_max_idx];
    }
```

Listing 3: Problem 8