ESO207: Theoretical Assignment 1 - Part 1

Debaditya Bhattacharya

February 12, 2021

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

Problem 2

Description:

Let $r = k \mod (q-p)$ (To rotate properly when $k > q-p$). If $r = 0$ no modification needed, return *head*. If $r \neq 0$ Iterate through the list to get a_p, a_q and a_{q-r} ($T = \mathcal{O}(1)$). We will now work with the sub-list $a_p \ldots a_q$ Connect a_p and a_q to form circular loop. a_{q-r+1} takes first position in the sub array. a_{q-r} takes last position. Adjust all linking paths appropriately.

Algorithm:

```
function rotate_sublist(head, p, q, k)
{
    r <- k%(p-q);

    if (r == 0);                    \\ No rotation needed
        return head;

    ptr <- head;
    for(i = 1; i <= q; i++)         \\ O(log(q)) time to fetch required pointers
    {
        if (p == i)
            ptr_p <- ptr;           \\Pointer to p element
        if (q == i)
            ptr_q <- ptr;           \\Pointer to q element
        if (q-r == i)
            ptr_q-r <- ptr;         \\Pointer to q-r element
        if (i != q)
            ptr <- ptr.right;
    }
    if(ptr_p.left == NULL)          \\p was head
        ptr_p-1 <- NULL;
    else
        ptr_p-1 <- ptr_p.left;


    if(ptr.q.right == NULL)         \\q was tail
        ptr_q+1 <- NULL;
    else
        ptr_q+1 <- ptr_q.right;

    ptr_p.left <- ptr_q;            \\Update pointers
    ptr_q.right <- ptr_p;
    ptr_q-r+1.left <- ptr_p-1;
    ptr_q-r.right <- ptr_q+1;
    if(ptr_p-1 != NULL)
        ptr_p-1.right <- ptr_q-r+1;
    if(ptr_q+1 != NULL)
        ptr_q+1.left <- ptr_q-r;

    return head;
}
```

Listing 1: Rotate sublist

Problem 3

Description:

In merged and sorted array C, the elements $C[i]$ such that $i < n$ will contain elements from A or B. If $C[i]$ contains the first r elements of A then it must contain the first $n-1-r$ elements of B. To find r we impose the condition (1) $A[r] > B[n-1-r]$ and condition (2) $A[r-1] < B[n-1-r+1]$. If (1) is false ignore the left side of L and divide. If (1) is true, and (2) is false ignore the right hand side. If Both (1) and (2) is median is $\frac{(max([A[r-1], B[n-1-r]])+min([A[r], B[n-r]]))}{2}$.

index	0	1	2	3	4
A	12*	17*	23*	34*	65
B	40*	53	59	61	66
A	12*	17*	23*	34*	65
B'	66	61	59	53	40*
index	4	3	2	1	0

To understand better consider the example above, where we have flipped B to B' for easier understanding. We need to find compare 23 and 59. $23 < 59$ therefore we ignore the left side of 23. We then move to the next middle element 34. As $34 < 53$ we ignore to left of 34. Once at 65, (1) and (2) are met. therefore median is $\frac{max(34,40)+min(53,65)}{2} = (40+53)/2 = 46.5$.

Algorithm:

```
    function find_median(A, B, n){
        if (A[0] > B[n-1])                              \\Trivial case
            return (B[n-1] + A[0])/2.0;
        if (A[n-1] < B[0])                              \\Trivial case
            return (A[n-1] + B[0])/2.0;
        L <- 0;                                         \\Boundary control
        R <- n-1;
        idx = -1;
        while (idx == 0){
            m = (L+R)/2;                                \\Middle element
            if (A[m] > B[n-1-m]){                       \\check Condition (1)
                if (A[m-1] < B[n-1-m+1]){               \\check Condition (2)
                    idx = m;                            \\Both Condition (1) and (2) true
                    break;
                }
                R = m-1;                                \\Condition (1) but not (2)
            }
            else{
                L = m+1;                                \\Neither (1) nor (2)
            }
        }
        median <- (max([A[idx-1], B[n-1-idx]]) + min([A[idx], B[n-1-idx+1]]))/2;
        return median;
    }
```

Listing 2: Find median

Proof of correctness:

Trivial cases are when first $n-1$ elements of the merged array are either completely A or B. Mode is then just the (last element of first array + first element of last array)/2.

Assertion: P(i): There is a turning point(k such that condition (1) $A[k] > B[n-1-k]$ and condition (2) $A[k-1] < B[n-k]$) for $L \leq k \leq R$ hold. (Interpret as as point k such that in A, for all points before k, is before the median, and for B all points before $n-1-k$ are is before the median.)

Base Case: $L = 0, R = n-1$. As $A[0] \not> B[n-1]$ and $A[n-1] \not< B[0]$, we will exist k such that $P(1)$ is true, as we have excluded the trivial cases.

Induction case: Let $P(i)$ be true. Then consider the case $P(i+1)$, take $m = (L+R)/2$. If (1) does not hold for $A[m]$ then (1) does not hold for all $k < m$ because the array A is increasing. Hence we can set $L = m+1$. If condition (1) holds for m, and condition (2) does not hold, then we can say that (2) does not hold for all $k > m$ because the array A is increasing. We can then set $R = m-1$. If both (1) and (2) hold. We have found the turning point k.

Median: At the turning point the $C[n-1], C[n]$ can be found as we know that $C[n-2], C[n-1], C[n], C[n+1]$ terms are $A[k-1], B[n-1-k], A[k], B[n-k]$ in unsorted order. We can then find the median of this 4 element array in O(1) time. this is the median of the larger array.

Problem 4

Description:

First we reduce create an auxiliary array X using the relation $X[i] = \frac{a^2 - cA[i]}{b}$. We note that as $A[i]$ was sorted in increasing order, $X[i]$ will be in decreasing. We then start comparing the elements of $X[i]$ and $A[j]$(from the back), increasing $i$if $X[i]$ is smaller than $A[j]$ and decreasing j if $A[j]$ is smaller. In this way we traverse and compare the all the elements in $T = \mathcal{O}(2n)$, in the worst case. If $X[i] = A[j]$, return j, else at the end return -1.

Algorithm:

```
function checkXY(A, n, a, b, c){
    X <- empty array of floats array of size A.size;

    for (i = 0; i<n; i++){                          \\T(n) = O(n)
        X[i] = (a*a - c*A[i])/b;                    \\Reduce to comparable form.
    }

    i <- 0;
    j <- n-1;
    while ((i<n) && (j>=0)){
        if(X[i]==A[j]){
            if (!i = n-1-j){                        \\Since distinct x,y
                return 1;                           \\Found element
            }
        }
        if(X[i]<A[j]){
            j--;
        } else {
            i++;
        }
    }

    return -1                                       \\Not found, does not exist.
}
```

Listing 3: checkXY

Proof of correctness:

For the relation $a^2 = bx + cy$ to hold for distinct $x, y \in A$. It is sufficient to show that if $y \in A$ then x must be of the form $x = \frac{a^2 - cy}{b}$.

We may find possible values of $X[i] = \frac{a^2 - cA[i]}{b}$ through an iterative manner in $T = \mathcal{O}(n)$. As A is sorted in increasing order (Assume without loss of generality), X will be sorted in decreasing order. We proceed to find comparing X and A to find the any common elements.

As the arrays are sorted in opposite order, we start comparing the values of X from the start and A from the end. If $X[i] < A[j]$, we shift then shift our focus to $A[j-1]$, and vice versa. If $X[i] = A[j]$ and $i \neq n-1-j$ (Since x, y need to be distinct). At each iteration, we will shift one of i or j. Hence we will visit each element at most once. The worst case time taken will be $T = \mathcal{O}(2n) = \mathcal{O}(n)$.

Hence the entire algorithm wil run in $T = \mathcal{O}(n)$, and will return 1 if such $x, y \in A$ and will return 0 if it is not possible.

Problem 1

Description:

We assume that we have n computers at our disposal. Computers are labeled $0, 1, 2 \ldots n$. Computer i compares $A[i-1], A[i], A[i+1]$. If A is a local maximum return i, else do not return anything. The driver CPU will now receive 2 values of i from two computers. These are the required data points. This can be done in $T = \mathcal{O}(1) = \mathcal{O}(\log n)$

Algorithm:

```
    function check_maxima(A,n){
        i <- SELF_LABEL                   \\Predefined macro available to the computer.
        if (i == 0){
            if (A[i]>A[i+1])
                return i;                \\First element is maxima, return 0
            else
                return;
        }
        if (i == n-1){
            if (A[i]>A[i-1])             \\Last element is maxima, return n-1
                return i;
            else
                return;
        }
        if (A[i]>A[i-1] && A[i]>A[i+1]) \\Element is maximum, return i
            return i;
        else
            return;
    }
```

Listing 4: Find two maximas.
