Name: Viswa Nihar Nukala
UB ID: 50414392

① Maximum Contingous Subsequence Sum Problem.

To find the subarray with maximum sum we can divide the array into 2 different arrays, but when we divide, we can get the solution as one of the following 3 scenarios.

→ Whole subarray is in the left half of the array
i.e, A[i,j] such that low < i < j < mid

→ Whole subarray is in the right half of the array
i.e, A[i,j] such that mid < i < j < high

→ Subarray is crossing the mid and is in both halves of the array i.e, A[i,j], low < i < mid < j < high

For either the first or second scenario, we can use divide and conquer strategy directly by treating them as sub problems.

For the 3rd scenario, we cannot apply the direct approach but treat them as 2 seperate problems and solve them seperately.

After solving the above 2, we can take solution as the sum which is highest of the 2.

The algorithm for finding the 3rd scenario is :-

```
1. find_max_crossing_subarray (A, low, mid, high)
2.     left_sum = -∞
3.     sum = 0
4.     for i = mid downto low                    ┐
5.         sum = sum + A[i]                      │ Find the highest sum in
6.         if sum > left_sum                     │ the left from some 'i'
7.             left_sum = sum                    │ to mid
8.             max_left = i                      ┘
9.     right_sum = -∞
10.    sum = 0
11.    for j = mid+1 to high                     ┐
12.        sum = sum + A[j]                      │ Find the highest sum in
13.        if sum > right_sum                    │ the right from mid+1
14.            right_sum = sum                   │ to some 'j'
15.            max_right = j                     ┘
16.    return (max_left, max_right, left_sum + right_sum)
```

Total number of iterations = $\frac{n}{2} + \frac{n}{2}$ [$\frac{n}{2}$ each for left and right]

$= n$

The algorithm for solving the 1st two scenarios is:-

```
1. find_max_subarray(A, low, high)
2.     if high == low
3.         return (low, high, A[low])
4.     else
5.         mid = (high + low)/2
6.         (left_low, left_high, left_sum)
7.                 = find_max_subarray(A, low, mid)
8.         (right_low, right_high, right_sum)
9.                 = find_max_subarray(A, mid+1, high)
10.        (cross_low, cross_high, cross_sum)
11.                = find_max_crossing_subarray(A, low, mid,
                                                high)
12.        if left_sum > right_sum and left_sum > cross_sum
13.            return (left_low, left_high, left_sum)
14.    else if right_sum > left_sum and right_sum > cross_sum
15.            return (right_low, right_high, right_sum)
16.        else
17.            return (cross_low, cross_high, cross_sum)
```

In the above algorithm we recursively find the highest sum in the left and right subarrays and then compare it with the crossing sum.

Finally we return the indices and sum of the highest of the 3.

Time complexity analysis:-

$$T(n) = \theta(1) + 2T(n/2) + \theta(n) + \theta(1)$$

$$\Rightarrow T(n) = 2T(n/2) + \theta(n)$$

For base case,

$T(n) = \theta(1)$ when there is only 1 element

Combining:-

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2) + \theta(n) & \text{if } n>1 \end{cases}$$

From Master's theorem

$\Rightarrow T(n) = O(n \log n)$ where $a=2$ $b=2$ $k=1$

(3) To find :-

Given is a $n \times n$ matrix, where n is a power of 2 and satisfies monotone property.

Describe an algorithm of $O(n \log n)$ for finding the maximum entry for each row of M.

Proof :-

This can be solved using Divide and Conquer. Following steps can be used to solve the problem :-

→ Divide the array horizontally with upper part from 0 to $\frac{i}{2} - 1$ and the lower part from from $\frac{i}{2} + 1$ to $n-1$.

→ Find the max element in the $\frac{i}{2}$ row. Let's assume that the maximum element was found in j^{th} column.

→ In the upper part for each row the maximum element will be in $k \leq j$ column.

→ In a similar way, for each row in the bottom part, the maximum element will be in $k \geq j$.

→ We pass the upper array from 0 to j and for the lower array, from j to $n-1$.

```
1.  Find_max_monotone (A, i_low, i_high, j_low, j_high, result)
2.      if (i_low == i_high)
3.          result [i_low] = index (max(A[i_low]))
4.          return result.
5.      else if (i_low < i_high)
6.          i_mid = (i_low + i_high)/2
7.          max_index = index (max (A[i_mid]))
8.          result [i_mid] = max_index
9.          if (max_index == j_low)
10.             result [i_low : i_mid] = j_low
11.             Find_max_monotone (A, i_mid+1, i_high,
                        max_index, j_high, result)
12.         else if (max_index == j_high)
13.             result [i_mid+1 : i_high] = j_high
14.             Find_max_monotone (A, i_low, i_mid-1,
                        j_low, max_index, result)
15.         Find_max_monotone (A, i_low, i_mid-1,
                        j_low, max_index, result)
16.         Find_max_monotone (A, i_mid+1, i_high,
                        max_index, j_high, result)
```

The initial call to the function is

Find_max_monotone (A, 0, n-1, 0, n-1, result)

The lines 9 to 11 assign the max of the top sub matrix to max_index due to the monotone property.

In a similar way, line 12 to 14 assign the max of all rows in the lower maxtrix to max_index due to monotone property.

Then in line 15 & 16 we call recursively the same function by reducing the size of the matrix to be send based on where the max_index is.

The time taken to run this algorithm is the worst case is $O(n \log n)$ when all the maximum elements are towards the end of the matrix.

It is nlogn because in the first call it iterates n times to find the max and subsequently in the next call only 1 iteration would be done but the function gets called for logn times.

Hence $T(n) = O(n \log n)$ is the time complexity of the above algorithm.

④ Aim:- To find k closest elements to the median.

We can find the k closest elements in the following 4 steps:-

→ Find median, $\frac{n}{2}$ biggest element of an array using linear time selection which takes linear time

→ Find another hashmap with keys of the array as the absolute difference between the elements of the array obtained above and the median, and value as the element value of the array

→ Apply linear selection algorithm to find the k^{th} largest element in this hashmap using the keys

→ After the 3rd step, the first k elements of the hashmap would be the k elements closest to the median.

All the four steps in the above algorithm takes linear amount of time. Hence, the algorithm takes $O(n)$

```
1.  Partition (A, p, r)
2.      x ← A[r]
3.      i ← p-1
4.      for j ← p-1 to r-1 do
5.          if A[j] ≤ then
6.              i ← i+1
7.              Swap A[i] and A[j]
8.          end if
9.      end for
10.     Swap A[i+1] and A[r]
11.     return i+1
```

If A is hashmap, do the operations based on keys

p → lower index
r → upper index
i → i^{th} biggest element to be found

```
1.  Select (A, p, r, i)
2.      if p=r return A[p]
3.      if len(A)< 5
4.          Sort A
5.          return A[i]
6.      divide A into subsets S[i] of 5 elements
7.      for i=1 to n/5 do
8.          qth[i] = select (s[i], 0, 5, 3, B)
9.      q = select ([qth[i]], 0, n/5, n/10)
10.     k = q - p + 1
```

If A is hashmap, do the operations based on keys

qth → q^{th} biggest element of each subarray

```
11.     if i=k, return A[q]
12.     if i<k, return select(A, p, q-1, i)
13.     if i>k, return Select(A, q+1, r, i-k)
```

```
1.  find_k_neighbours(A, k):
2.      median = select(A, 0, n, n/2)
3.      new_A (int, int>
4.      for i ← 0 to n-1 do
5.          new_A.push(abs(median - A[i]), A[i])
6.      n_neighbour = Select(new_A, 0, n, k)
7.      return new_A[0:k]
```

↳ return 1st k elements

Hence $T(n) = O(n)$ is the time complexity since the function iterates 'n' times for 4 times.

(6) Given :- An array of n distinct and unsorted real numbers and weights such that

$$\sum w_i = 1.$$

Solution :-

We can find the weighted mean of the above with a small modification to the linear time selection algorithm.

First we compute the median using the select algorithm and calculate two subsets such that :-

→ one set contains all the elements less than median

→ one set which contains all the elements greater to the median.

[The question assumes that there is only one weighted median]

The next step is to calculate the sum of the weights for each subset. Let the sum of the weights to left be w_low and to right be w_right.

If $w\_low + w < \frac{1}{2}$ and $w\_right + w \leq \frac{1}{2}$ then return the median.

If $w\_low + w > \frac{1}{2}$, then recursively apply the function on the left subset.

If w_left + w ≥ 1/2 then recursively apply the function on the right subset.

PS:- w is the balance weight from the previous recursive call and initial value is 0.

Algorithm is as follows:-

```
1.  Find_weighted_median(A, w)
2.      n ← length(A)
3.      m ← Select(A, 0, n, n/2)
4.      left_subset ← [ ]
5.      right_subset ← [ ]
6.      w_left ← 0
7.      if n = 1
8.          then return A[1]
9.      for i ← 1 to n
10.         do if A[i] < m
11.             then w_left ← w_left + w;
12.                 Append A[i] to left_subset
                    else Append A[i] to right_subset
13.     if w + w_left > 1/2
14.         then Find_weighted_median(left_subset, w)
15.         else Find_weighted_median(right_subset,
16.                                   w_left)
```

The initial call to this algorithm would be

Find_weighted_median(A, 0).

The algorithm terminates when the elements in A is 0.

Splitting A into left_subset and right_subset takes $\theta(n)$ time. Each recursive call reduces the size of the array by half.

$\therefore T(n) = T(n/2) + \theta(n)$

From Master's theorem

$T(n) = \theta(n)$

Reference link :-

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/assignments/ps2sol.pdf