Name: Ebram Raafat Thabet

ID: 900214496

Analysis and complexity:

1- The complexity of this function is O (m), where m is the length of the string.

```
vector <string> From_line_to_desired(string s){

vector <string> res;

string insertion = "";
s += ' ';

// add to the vector if the character equal to a space
for(int i = 0; i < s.length();i++){
    if (s[i] != ' '){
        insertion+=s[i];
    }
    else{
            res.push_back(insertion);
            insertion = "";

    }
}

return res;
}
```

2- The first while loop of getting the data is of an O(n*m) complexity, where n is the number of lines in the data file, and m is the length of the string (as we call the "From_line_to_desired() function").

```
while(getline(in,x)){

// skipping the first line "date value"

if(x[0] == 'd'){continue;}

// passing the line to the rectifing function and adding it to the vector
vector <string> vec = From_line_to_desired(x);

// create a node to hold the data from the rectifying function
Hnode N;
N.date = vec[0]; N.value = stod(vec[1]);
vector_of_tokens.push_back(vec);
vector_of_nodes_max.push_back(N);
vector_of_nodes_min.push_back(N);
vector_of_nodes_for_max_sub.push_back(N);
}
```

3- Then the for loop of getting the differences if O(n), where n is the is the size of the vector of nodes.

```
for (int j = 0; j< vector_of_nodes_for_max_sub.size()-1;j++){
double diff_dash = vector_of_nodes_for_max_sub[j+1].value- vector_of_nodes_for_max_sub[j].value;
differences.push_back(diff_dash);
}
```

4- maxSubsequentSum():

```
// This is a function to implement the maximum subsequent algorithm and return the indices of the
double maxSubsequenceSum(vector<double>& differences, int& start, int& end) {
    double maxSum = 0;
    double currentSum = 0;
    start = 0;

    for (int i = 0; i < differences.size(); ++i) {
        currentSum += differences[i];
    // The current sum is negative this means it needs to be reset and the second element should be considered now
        if (currentSum < 0) {
            currentSum = 0;
            start = i + 1;
        }
    // if the current sum is greater than the maximum sum just assign current sum to maxSum
        if (currentSum > maxSum) {
            maxSum = currentSum;
            end = i;
        }
    }

    return maxSum;
}
```

The complexity is O(n) (one for loop), where n is the size of the differences vector.

5- heapify_max():

```
// Heapify max function
void heapify_max(vector<Hnode>& arr, int n, int i) {

// declaring the parent, left, and right childs
    int greatest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

// update if left child is greater

    if (left < n && arr[left].value > arr[greatest].value) {
        greatest = left;
    }
// update if right child is greater

    if (right < n && arr[right].value > arr[greatest].value) {
        greatest = right;
    }

    if (greatest != i) {
        swap(arr[i], arr[greatest]);
        heapify_max(arr, n, greatest);
    }
}
```

The swap function has a complexity of O(1).

The total complexity of the function is O(height of the tree) = O(log (n)). Where n is the number of nodes.

6- Build_max_heap()

```
void build_max_heap(vector<Hnode>& arr) {
    int s = arr.size();
    for (int i = s / 2 - 1; i >= 0; i--) {
        heapify_max(arr, s, i);
    }
}
```

This function is of complexity O (n log (n)) as it linearly loops to heapify and the heapify_max function is of a complexity of log (n). So, the total complexity can be computed as follows:

O (n/2-1) = O (n) (the for loop) * O (log (n)) (the one of heapify_max() function). This gives total complexity of O (n log(n)).

7- heapify_min()

```
void heapify_min(vector<Hnode>& arr, int n, int i) {
    int smallest = i;
    int left_c = 2 * i + 1;
    int right = 2 * i + 2;

// update if left child is smaller
    if (left_c < n && arr[left_c].value < arr[smallest].value) {
        smallest = left_c;
    }
// update if right child is smaller
    if (right < n && arr[right].value < arr[smallest].value) {
        smallest = right;
    }

    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        heapify_min(arr, n, smallest);
    }
}
```

The swap again is O (1).

The case is the same as heapify_max, the function is O(tree height) or O (log (n)), where n is the number of nodes.

8- build_min_heap()

```
// This function builds a minimum heap usign heapify min algorithm
void build_min_heap(vector<Hnode>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify_min(arr, n, i);
    }
}
```

This function is of complexity O (n log (n)) as it linearly loops to heapify and the heapify_min() function is of a complexity of O(log (n)). So, the total complexity can be computed as follows:

O (n/2-1) = O (n) (the for loop) * O (log (n)) (the one of heapify_min() function). This gives total complexity of O (n log(n)).

9- remove_by_k():

```
Hnode remove_by_k(vector<Hnode> &VecOfN , int k){
    Hnode res;
    // k = 1 for min heap
    if (k == 1 ){
        res = VecOfN[0];
        VecOfN.erase(VecOfN.begin());
        build_min_heap(VecOfN);
    }
    // k = 2 for max heap
    else if (k == 2){
        res = res = VecOfN[0];
        VecOfN.erase(VecOfN.begin());
        build_max_heap(VecOfN);
    }
    return res;
}
```

This function either calls the build_max_heap() or build_min_heap(). Both have complexity of

O(n log (n)). The complexity of the erase function is O (n) where n is the size of the vector because after removing the whole vector has to be shifted one position. vector.begin() has a complexity of O(1). So, the total complexity of the function is O(n log(n)) + O(n) + O(1) = O (n log (n)).

10- This for loop has the remove_by_k() function and the complexity can be computed as follows:

```
for (int i = 0; i < 10; i++){
Hnode z = remove_by_k(vector_of_nodes_min,1);
cout << i+1 << ". Value: "<< z.value << "  Date: " << z.date<<endl;}
```

Since the range of the for loop is constant every time this for loop has a complexity of O(10) = O (1).

The remove_by_k() has a complexity of O (n log (n)). Therefore, the total complexity is O (1) + O (n log(n)) = O (n log(n)).

Aggregating all the aforementioned complexities, we reach

While loop: O(n * m) +

The for loop of the differences: O(n) +

The maximum subsequent sum function : O(n) +

build_max_heap function: O(n log (n)) +

build_min_heap function: O(n log (n)) +

for loop of remove_by_k function: O (n log(n))

= O(n log (n)).

Screenshot of the output:

```
The period of maximum increasing is between 1/4/1999 and 1/5/1999 and the value of this change is 0.7723


Max values:
1. Value: 1.5994  Date: 4/22/2008
2. Value: 1.5946  Date: 4/16/2008
3. Value: 1.5936  Date: 7/11/2008
4. Value: 1.5924  Date: 7/21/2008
5. Value: 1.5917  Date: 7/15/2008
6. Value: 1.5906  Date: 4/21/2008
7. Value: 1.5904  Date: 7/14/2008
8. Value: 1.5897  Date: 4/17/2008
9. Value: 1.5889  Date: 4/23/2008
10. Value: 1.5879  Date: 7/2/2008

Min values:
1. Value: 0.8271  Date: 10/25/2000
2. Value: 0.8295  Date: 10/26/2000
3. Value: 0.8355  Date: 10/24/2000
4. Value: 0.8357  Date: 10/23/2000
5. Value: 0.8369  Date: 7/5/2001
6. Value: 0.8383  Date: 11/24/2000
7. Value: 0.8389  Date: 10/27/2000
8. Value: 0.8389  Date: 10/18/2000
9. Value: 0.8405  Date: 10/30/2000
10. Value: 0.8414  Date: 10/20/2000
```