Problem 1: Road Trip: Suppose you are going on a road trip with friends. Unfortunately, your headlights are broken, so you can only drive in the daytime. Therefore, on any given day you can drive no more than d miles. You have a map with n different hotels and the distances from your start point to each hotel x1< x2< ... < xn. Your final destination is the last hotel.

a)

- Description

  In order to minimize the number of days, the farthest hotel within the distance 'd' from the starting point(start point, or previous hotel) has to be visited for each day.

- Pseudo Code

```
I.   Build heap for the hotels
II.  Find the farthest hotel within distance 'd' from the starting point in the heap.
III. Set the starting point as the hotel found in step I.
IV.  Iterate step II ~ III until the last hotel is found in Step II.
```

b)

Running time for searching the hotel from the heap is $O(lgn)$, and building heap is $O(n)$.

- Best Case: $O(n)$

(The destination is distance 'd' away from the start point, $O(n) + O(lgn) = O(n)$)

- Average Case: $O(nlgn)$

(running time: $cnlgn, 0 < c \leq 1 \Rightarrow O(nlgn), O(n) + O(nlgn) = O(nlgn)$)

- Worst Case: $O(nlgn)$

(Each hotel is distance 'd' away from each other, $O(n) + O(nlgn) = O(nlgn)$)

Problem 2:

- Explanation

  This approach selects the latest starting time of the activities every time. After selecting the latest starting, this approach selects the next latest starting time of activity, and so on. Since this approach is keep selecting the best case in the aspect of last activity to start, this approach is a greedy algorithm.

- Proof

  Let a_m be an activity in S_k (non-empty subset) with the latest starting time.

  Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the latest start time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of Sk. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $s_m \leq s_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m

Problem 3:

- Explanation

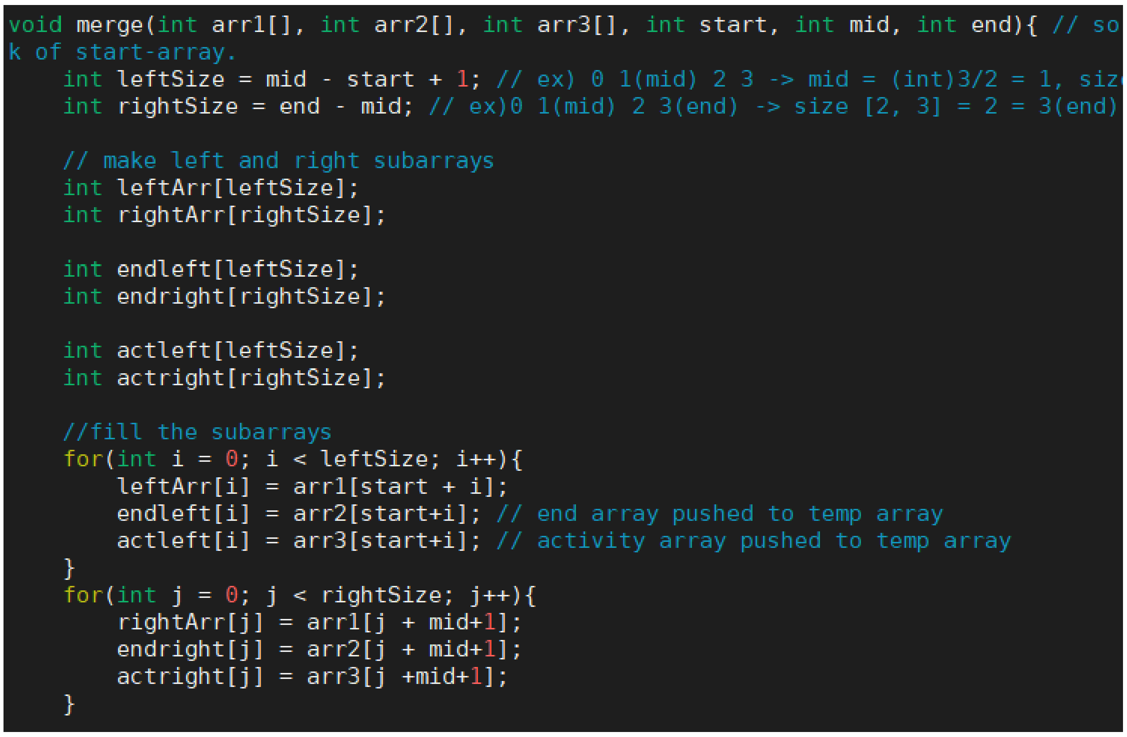
  The program will first sort the activity timeline in descending order. By using greedy algorithm, the program will select the biggest start time, and then pop that activity from the array. The program will still iterate this process until there is no activity which fits in the time frame. Through these processes, the program finds maximum number of activities and the list from the activity array.

- Pseudo Code

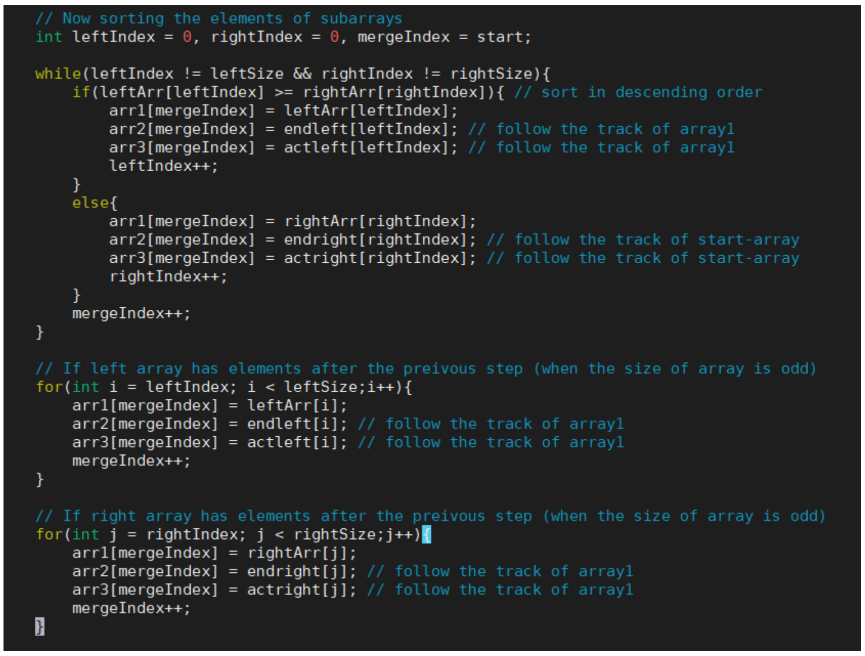
```
I.   Sort the activity start time with descending order (Merge Sort)
II.  Put the first element of activity to the bag
III. M = 2 ~n(number of activities), I = 1
IV.  If start[M] is greater than or equal end[I], then put Mth element of activity to the bag,
     I <- M
V.   Go to step IV until M equals to n(amount of activities)
```

- Sorting

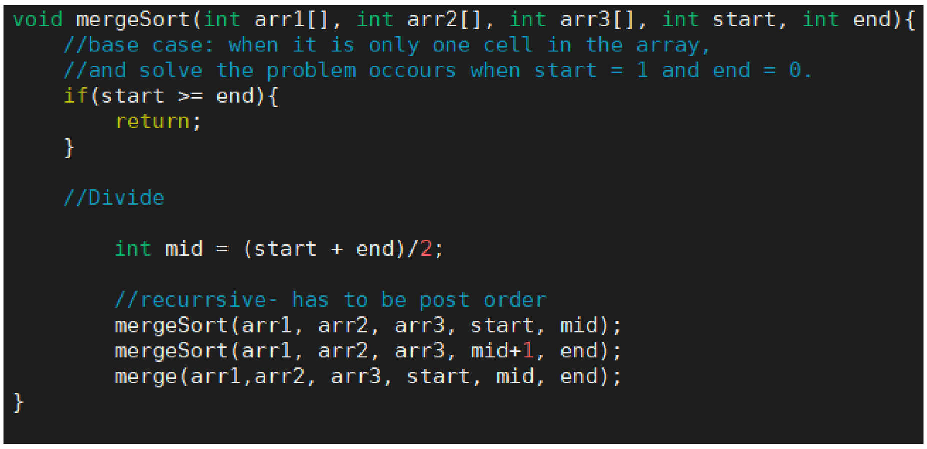
  I used merge sort to sort the starting time of activity in descending order. Theoretical running time of merge sort is $O(n \lg n)$.



```
void merge(int arr1[], int arr2[], int arr3[], int start, int mid, int end){ // so
k of start-array.
    int leftSize = mid - start + 1; // ex) 0 1(mid) 2 3 -> mid = (int)3/2 = 1, siz
    int rightSize = end - mid; // ex)0 1(mid) 2 3(end) -> size [2, 3] = 2 = 3(end)

    // make left and right subarrays
    int leftArr[leftSize];
    int rightArr[rightSize];

    int endleft[leftSize];
    int endright[rightSize];

    int actleft[leftSize];
    int actright[rightSize];

    //fill the subarrays
    for(int i = 0; i < leftSize; i++){
        leftArr[i] = arr1[start + i];
        endleft[i] = arr2[start+i]; // end array pushed to temp array
        actleft[i] = arr3[start+i]; // activity array pushed to temp array
    }
    for(int j = 0; j < rightSize; j++){
        rightArr[j] = arr1[j + mid+1];
        endright[j] = arr2[j + mid+1];
        actright[j] = arr3[j +mid+1];
    }
```



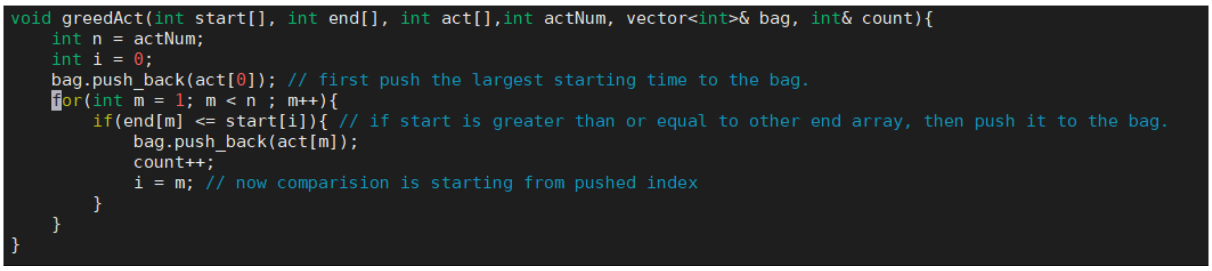
```
    // Now sorting the elements of subarrays
    int leftIndex = 0, rightIndex = 0, mergeIndex = start;

    while(leftIndex != leftSize && rightIndex != rightSize){
        if(leftArr[leftIndex] >= rightArr[rightIndex]){ // sort in descending order
            arr1[mergeIndex] = leftArr[leftIndex];
            arr2[mergeIndex] = endleft[leftIndex]; // follow the track of array1
            arr3[mergeIndex] = actleft[leftIndex]; // follow the track of array1
            leftIndex++;
        }
        else{
            arr1[mergeIndex] = rightArr[rightIndex];
            arr2[mergeIndex] = endright[rightIndex]; // follow the track of start-array
            arr3[mergeIndex] = actright[rightIndex]; // follow the track of start-array
            rightIndex++;
        }
        mergeIndex++;
    }

    // If left array has elements after the preivous step (when the size of array is odd)
    for(int i = leftIndex; i < leftSize;i++){
        arr1[mergeIndex] = leftArr[i];
        arr2[mergeIndex] = endleft[i]; // follow the track of array1
        arr3[mergeIndex] = actleft[i]; // follow the track of array1
        mergeIndex++;
    }

    // If right array has elements after the preivous step (when the size of array is odd)
    for(int j = rightIndex; j < rightSize;j++){
        arr1[mergeIndex] = rightArr[j];
        arr2[mergeIndex] = endright[j]; // follow the track of array1
        arr3[mergeIndex] = actright[j]; // follow the track of array1
        mergeIndex++;
    }
```



```
void mergeSort(int arr1[], int arr2[], int arr3[], int start, int end){
    //base case: when it is only one cell in the array,
    //and solve the problem occours when start = 1 and end = 0.
    if(start >= end){
        return;
    }

    //Divide

        int mid = (start + end)/2;

        //recurrsive- has to be post order
        mergeSort(arr1, arr2, arr3, start, mid);
        mergeSort(arr1, arr2, arr3, mid+1, end);
        merge(arr1,arr2, arr3, start, mid, end);
}
```

- Greedy algorithm: Last activity to Start

  Last-to-start algorithm iterates until the variable m reaches n-1 so that theoretical running time of the algorithm is $O(n)$.



```
void greedAct(int start[], int end[], int act[],int actNum, vector<int>& bag, int& count){
    int n = actNum;
    int i = 0;
    bag.push_back(act[0]); // first push the largest starting time to the bag.
    for(int m = 1; m < n ; m++){
        if(end[m] <= start[i]){ // if start is greater than or equal to other end array, then push it to the bag.
            bag.push_back(act[m]);
            count++;
            i = m; // now comparision is starting from pushed index
        }
    }
}
```

- Entire running time

  The entire running time of the algorithm is $\mathrm{O(nlgn) + O(n)}$ which is equivalent to $O(nlgn)$. Therefore, the theoretical entire running time of Activity Selection Last-to-Start is $O(nlgn)$.