Report for assignment 3

Shashwat Gupta (14IE10028)

January 22, 2016

Problem-I

1 Random Skyline Buildings Generation

I have taken time as the seed for the random variable generation and i initialise it to zero on every execution so that i get a unique set of inputs every time. I allow the user to input the limits and number of buildings following which i check if there are any inconsistencies in the input, in which case the code is immediately terminated. The random filling is done next according to the limits as can be seen in the code provided below. The building points are displayed to the user and then the critical points obtained as output are displayed.

The following is the Code of int main()

```
int main()
{
    srand(time(NULL));
    int size,i,x1,x2,hh;
    printf("Enter number of buildings : ");
    scanf("%d",&size);
    printf("Enter left limit : ");
    scanf("%d",&x1);
    printf("Enter right limit : ");
    scanf("%d",&x2);
    printf("Enter height limit : ");
    scanf("%d",&hh);
    if(x2<x1||size<=0||hh<=0)    {
        printf("Incorrect inputs, program terminated\n");
        exit(0);
    }
    triple *building = (triple *)malloc(sizeof(triple)*size);
    for(i=0;i<size;i++)    {
        building[i].l=(rand()%(x2-x1))+x1;
        building[i].r=(rand()%(x2-building[i].l))+building[i].l+1;
        building[i].h=(rand()%hh)+1;
    }
    prin_buildings(building,size);
    point* a=mergeSort_skyline(building,0,size-1);
    prin_points(a,size*2);
}
```

Listing 1: main

2 Printing Functions

The following code is the function that displays the tuple points for the randomly created buildings to populate the skyline.

```
void prin_buildings(triple *building, int size)
{
    printf("printing building triplets: \n");
    int i;
    printf("left\tright\theight\n");
    for (i = 0; i < size; ++i)
        printf("%d\t%d\t%d\n",building[i].l,building[i].r,building[
    i].h);
}
```

Listing 2: print buildings

The following code is the function that displays the tuple points for the critical points obtained as output that marks the silhoutte of the skyline.

```
void prin_points(point *skyline, int n)
{
    printf("printing critical point(s): \n");
    int i;
    for (i = 0; i < n; i++)
    {
        if(i!=0&&skyline[i].l==0&&skyline[i-1].l>0)
            break;
        printf("(%d, %d)\n",skyline[i].l,skyline[i].h);
    }
}
```

Listing 3: print points

3 Divide and Conquer

The problem at hand is solved using the Divide and Conquer algorithm. My approach is very similar to the Merge Sort routine. There is one Recursively implemented function whose task is to take the given building tuples and divide it into two parts which would represent the two independent silhouettes to be merged into a single final silhouette. Each part is called recursively and hence subdivided into two more parts which are also called recursively. The recursion is broken with the base case, when there is just one building and in this case the two critical points of that building are returned. These successive critical points are collectively merged to form the silhouettes. There is another function implemented just to merge two given silhouettes. The recursive function makes use of the merge function to join two silhouettes and form the combined silhouette. Thus, the recursive function divides and with the help of the merge function it conquers the final output.

```
point* mergeSort_skyline(triple *building, int l, int r)
{
    int m;
    if (l == r)
    {
        point *sky = (point *)malloc(sizeof(point)*2);
        point p1;
        p1.l = building[l].l; p1.h = building[l].h;
        point p2;
        p2.l = building[l].r; p2.h = 0;
        sky[0] = p1;
        sky[1] = p2;
        return sky;
    }
    m = (l + r)/2;
    point* leftsky = mergeSort_skyline(building, l, m);
    point* rightsky = mergeSort_skyline(building, m+1, r);
    return merge(leftsky, (m-l+1)*2, rightsky, (r-m)*2);
}
```

Listing 4: recursive function

The merging routine is the function that accepts two silhouettes and merges them and returns the final silhouette with all the relevant critical points. It also removes all the redundant points from the silhouette. The logic for merging is very much related to merge sort algorithm. We treat the two silhouettes like two arrays that need to be merged and sorted. We compare the first two points of each and take the one with the lesser x-coordinate. Now we check if the current height of the point chosen from the chosen silhouette is greater than the latest noted height of the other silhouette point, and whichever height is greater is stored. We then move to the next point of the chosen silhouette and compare again. At the same time the height variable is updated of the current silhouette. This process repeats and the rest is the same as merge sort. Thus the steps to perform are:

· Make two height trackers and set them to be zero initially.

- Take the first two elements of each silhouettes.
- Choose the one with the lesser x.
- Choose max height between chosen point and latest height of other silhouette
- move to next point and update height of current silhouette
- keep repeating the process till one of the silhouettes exhausts.
- add the remaining points of the remaining silhouettes while comparing each elements height with the last seen height of the exhausted silhouette.
- return the merged silhouette

```
point* merge(point *L, int n1, point *R, int n2)
{
    int i, j, k, h1=0, h2=0;;
    point *final = (point *)malloc(sizeof(point)*(n1+n2));
    i = 0;
    j = 0;
    k = 0;
    while (i < n1 && j < n2)
    {
        if (L[i].l < R[j].l)
        {
            if(k!=0&&MAX(L[i].h,h2)==final[k-1].h)
            {
                h1=L[i].h;
                i++;
                continue;
            }
            final[k].h = MAX(L[i].h,h2);
            final[k].l = L[i].l;
            h1=L[i].h;
            i++;
        }
        else if (L[i].l > R[j].l)
        {
            if(k!=0&&MAX(R[j].h,h1)==final[k-1].h)
            {
                h2=R[j].h;
                j++;
                continue;
            }
            final[k].h = MAX(R[j].h,h1);
            final[k].l = R[j].l;
            h2=R[j].h;
            j++;
        }
        else
        {
            if(k!=0&&MAX(R[j].h,L[i].h)==final[k-1].h)
            {
                h1=L[i].h;
                h2=R[j].h;
```

```
                i++;
                j++;
                continue;
            }
            final[k].l=L[i].l;
            final[k].h=MAX(L[i].h,R[j].h);
            h1=L[i].h;
            h2=R[j].h;
            i++;
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        final[k].h = MAX(L[i].h,h2);
        final[k].l = L[i].l;
        i++;
        k++;
    }
    while (j < n2)
    {
        final[k].h = MAX(R[j].h,h1);
        final[k].l = R[j].l;
        j++;
        k++;
    }
    return final;
}
```

Listing 5: merging function

4 Time Complexity Calculation

Time complexity of above recursive implementation is same as Merge Sort.

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

Solving this equation by recursive method yields T(n) $\in O(nLogn)$.

Problem-II

5 Random Skyline Buildings Generation

I have taken time as the seed for the random variable generation and I initialise it to zero on every execution so that i get a unique set of inputs every time. I allow the user to input the limits and number of buildings following which i check if there are any inconsistencies in the input, in which case the code is immediately terminated. The random filling is done next according to the limits as can be seen in the code provided below. The building points are displayed to the user and then the critical points obtained as output are displayed.

The following is the Code of int main()

```
int main()
{
    srand(time(NULL));
    int size,i,x1,x2,hh;
    printf("Enter number of buildings : ");
    scanf("%d",&size);
    printf("Enter left limit : ");
    scanf("%d",&x1);
    printf("Enter right limit : ");
    scanf("%d",&x2);
    printf("Enter height limit : ");
    scanf("%d",&hh);
    if(x2<x1||size<=0||hh<=0)
    {
        printf("Incorrect inputs, program terminated\n");
        exit(0)
    }
    quad *building = (quad *)malloc(sizeof(quad)*size);
    for(i=0;i<size;i++)
    {
        building[i].l=(rand()%(x2-x1))+x1;
        building[i].r=(rand()%(x2-building[i].l))+building[i].l+1;
        building[i].lh=(rand()%hh)+1;
        building[i].rh=(rand()%hh)+1;
    }
    prin_buildings(building,size);
    point* a=mergeSort_skyline(building,0,size-1);
    prin_points(a,size*2);
}
```

Listing 6: main

6 Printing Functions

The following code is the function that displays the quadruple points for the randomly created buildings to populate the skyline.

```
void prin_buildings(quad *building, int size)
{
    printf("printing building quadruples: \n");
    int i;
    printf("left\th-l\tright\th-r\n");
    for (i = 0; i < size; ++i)
        printf("%d\t%d\t%d\t%d\n",building[i].l,building[i].lh,
    building[i].r,building[i].rh);
}
```

Listing 7: print buildings

The following code is the function that displays the tuple points for the critical points obtained as output that marks the silhoutte of the skyline.

```
void prin_points(point *skyline, int n)
{
    printf("printing critical point(s): \n");
    int i;
    for (i = 0; i < n; i++)
    {
        if(i!=0&&skyline[i].l==0&&skyline[i-1].l>0)
            break;
        printf("(%d, %d)\n",skyline[i].l,skyline[i].h);
    }
}
```

Listing 8: print points

7 Divide and Conquer

The problem at hand is solved using the Divide and Conquer algorithm. My approach is very similar to the Merge Sort routine. There is one Recursively implemented function whose task is to take the given building tuples and divide it into two parts which would represent the two independent silhouettes to be merged into a single final silhouette. Each part is called recursively and hence subdivided into two more parts which are also called recursively. The recursion is broken with the base case, when there is just one building and in this case the three critical points of that building are returned. These successive critical points are collectively merged to form the silhouettes. There is another function implemented just to merge two given silhouettes. The recursive function makes use of the merge function to join two silhouettes and form the combined silhouette. Thus, the recursive function divides and with the help of the merge function it conquers the final output.

```
point* mergeSort_skyline(quad *building, int l, int r)
{
    int m;
    if (l == r)
    {
        point *sky = (point *)malloc(sizeof(point)*3);
        point p1;
        p1.l = building[l].l; p1.h = building[l].lh;
        point p2;
        p2.l = building[l].r; p2.h = building[l].rh;
        point p3;
        p3.l = building[l].r; p3.h = 0;
        sky[0] = p1;
        sky[1] = p2;
        sky[2] = p3;
        return sky;
    }
    m = (l + r)/2;
    point* leftsky = mergeSort_skyline(building, l, m);
    point* rightsky = mergeSort_skyline(building, m+1, r);
    return merge(leftsky, (m-l+1)*3, rightsky, (r-m)*3);
}
```

Listing 9: recursive function

The merging routine is the function that accepts two silhouettes and merges them and returns the final silhouette with all the relevant critical points. It also removes all the redundant points from the silhouette. The logic for merging is very much related to merge sort algorithm. We treat the two silhouettes like two arrays that need to be merged and sorted. We compare the first two points of each and take the one with the lesser x-coordinate. Now we check if the current height of the point chosen from the chosen silhouette is greater than the latest noted height of the other silhouette point, and whichever height is greater is stored. We use the slope function to get the slopes of two consecutive points

```
float slope(int x1,int x2,int y1,int y2)
{
```

```
    return((y2-y1)/(x2-x1));
}
```

Listing 10: recursive function

We then move to the next point of the chosen silhouette and compare again. At the same time the height variable is updated of the current silhouette. This process repeats and the rest is the same as merge sort. Thus the steps to perform are:

- Make two height trackers and set them to be zero initially.
- Take the first two elements of each silhouettes.
- Choose the one with the lesser x.
- Choose max height between chosen point and latest height of other silhouette
- Find slopes of the two points and compare them two find intersections
- move to next point and update height of current silhouette
- keep repeating the process till one of the silhouettes exhausts.
- add the remaining points of the remaining silhouettes while comparing each elements height with the last seen height of the exhausted silhouette.
- return the merged silhouette

```
point* merge(point *L, int n1, point *R, int n2)
{
    int i, j, k, h1=0, h2=0;;
    point *final = (point *)malloc(sizeof(point)*(n1+n2+2));
    i = 0;
    j = 0;
    k = 0;
    while (i < n1 && j < n2)
    {
        if (L[i].l < R[j].l)
        {
            if(k!=0&&MAX(L[i].h,h2)==final[k-1].h)
            {
                h1=L[i].h;
                i++;
                continue;
            }
            final[k].h = MAX(L[i].h,h2);
            final[k].l = L[i].l;
            h1=L[i].h;
            i++;
        }
        else if (L[i].l > R[j].l)
        {
            if(k!=0&&MAX(R[j].h,h1)==final[k-1].h)
            {
                h2=R[j].h;
```

```
                j++;
                continue;
            }
            final[k].h = MAX(R[j].h,h1);
            final[k].l = R[j].l;
            h2=R[j].h;
            j++;
        }
        else
        {
            if(k!=0&&MAX(R[j].h,L[i].h)==final[k-1].h)
            {
                h1=L[i].h;
                h2=R[j].h;
                i++;
                j++;
                continue;
            }
            final[k].l=L[i].l;
            final[k].h=MAX(L[i].h,R[j].h);
            h1=L[i].h;
            h2=R[j].h;
            i++;
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        final[k].h = MAX(L[i].h,h2);
        final[k].l = L[i].l;
        i++;
        k++;
    }
    while (j < n2)
    {
        final[k].h = MAX(R[j].h,h1);
        final[k].l = R[j].l;
        j++;
        k++;
    }
    return final;
}
```

Listing 11: merging function

8 Time Complexity Calculation

Time complexity of above recursive implementation is same as Merge Sort.

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

Solving this equation by recursive method yields T(n) $\in O(nLogn)$.