Henry Daniels-Koch
October 28 2016

Algorithms: Lab 5 Part 2

5). To find the optimal location of the main pipeline in linear time, we find the median y value of all the oil pipelines and lay our pipe down at this y value. This algorithm works because if the pipeline were to be shifted up 1 y value from the median, all of the wells below would be 1 unit farther away, all of the wells above would be 1 unit closer, AND the well at the median (assuming an odd number of wells) is also 1 unit farther away. Thus, moving the pipe 1 unit up causes the total distance between all the wells and the pipe to be at least 1 unit farther away. The same argument holds for moving the pipe down 1 unit. This algorithm runs in $O(n)$ since finding the median takes linear time using selection. The pseudocode is as follows:

```
y=pipe(A,n)
    median = select(A,n/2)
    return median
```

6). To find a local minimum in $O(lgn)$ time, we first check if the middle index of the array is a local minimum. If so, we return this value. If not, we keep going. We then check if the element at the middle index is <= the element at the index immediately to the right of the middle index. If so, a local minimum must be on the left half of the array since the elements initially decrease. We then recurse on the left half. If the element at the middle index is >= the element at the index immediately to the right of the middle index, a local minimum must of on the right half of the array since the elements at the end of the array initially decrease from the right. We then recurse on the right half. Since we recursively split the array in half, this algorithm runs in $O(lgn)$ time. The pseudocode is as follows:

```
findLocalMin(A,left,right)
    mid = (right-left)/2 + left
    if A[mid] is a local min
        return A[mid/2]
    if A[mid] < A[mid+1]
        findLocalMin(A,left,mid)
    else
        findLocalMin(A,mid,right)
```

7a). To combine two skylines of size n_1 and n_2, we iterate through every element in both lists. We store the current x coordinates and heights of both lists throughout the algorithm. We first compare the next x values of both lists and take the smaller x value. We then compare the height at this smaller x value to the height of the other skyline. If this height is bigger, we add a tuple of the smaller x value and its height. If this height is smaller, we add a tuple of the smaller x value and the height of the other skyline. We then update the x coordinates accordingly and store the current height of smaller x value. Since we check each value of both lists, this algorithm runs in $O(n_1 + n_2)$. The pseudocode is as follows:

```
combineSkies(list1,n1,list2,n2)
    height1=list1[0].height //current height of skyline 1
    height2=list2[0].height //current height of skyline 2
    while(i < n1 and j < n2)
        if list1[i].x < list2[j].x //x value is smaller
            if list1[i].height > height2 //height of smaller x value is greater than height of skyline2
```

skyline.set($list1[i].x, list1[i].height$)
height1 = list1[i].height
else //height of skyline2 is bigger (or equal)
skyline.set($list1[i].x, height2$)
height1 = list1[i].height
i++
else if $list2[j].x < list1[i].x$ //x value is smaller
if $list2[j].height > height1$ //height of smaller x value is greater than height of skyline1
skyline.set($list2[j].x, list2[j].height$)
height2 = list2[j].height
else //height of skyline1 is bigger (or equal)
skyline.set($list2[j].x, height1$)
height2 = list2[j].height
j++
else //x values are the same
if $list1[i].height > list2[j].height$
skyline.set(list1[i].x,list1[i].height)
else
skyline.set(list2[j].x,list2[j].height)
height1 = list1[i].height
height2 = list2[j].height

7b). To find the skyline of n buildings, we basically divide down the number of buildings in the skyline until we have 2 buildings and conquer them by merging the skylines of the two buildings using the algorithm in part a. We implement this by first checking if our list of buildings has only 2 buildings in it. If so, we merge the skylines with the algorithm above. If not, we recursively call the function on the bottom and top parts of the list of buildings. This algorithm runs in $O(nlgn)$ since our recursion takes lgn time and our merge algorithm in part a takes linear time. The pseudocode is as follows:

```
findSky(buildList,left,right)
    mid = (left-right) / 2 + left
    if (left - right) == 2
        mergeBuildings(buildList[left], 1, buildList[right], 1)
    findSky(buildList,0,mid)
    findSky(buildList,mid,right)
```

8a). The inversions of the array [2,3,8,6,1] are (2,1), (3,1), (6,1), (6,1), and (8,6).

b). An array that is sorted backwards (or sorted in decreasing order) has the most inversions. It has $n * n/2 = n^2/2$ inversions. We compute this from adding the geometric sum of inversions from 1 to n-1 (i.e. $1 + 2 + 3 + 4 + ...n - 1$).

c). To count the number of inversions in an array in $O(n^2)$ time, we iterate from the first to the second to last element. For each of these elements, we check if it is greater than the elements to right of it in the list. If so, we increment the number of inversions. Since for each element, we iterate through (n-i-1) elements, this algorithm runs in $O(n^2)$ time. The pseudocode is as follows:

```
findInv(A,n)
    numInv = 0
    for i = 0 to n-2
```

```
for j=i+1 to n-1
    if A[i] > A[j]
        numInv++
```

d). To determine the number of inversions in an array in $O(nlgn)$ time, we implement an algorithm similar to mergeSort. Our base case occurs when the array is of size 1 in which we return. Next we recursively call mergeInv on both halves of the array. After these functions are called, we need to merge the two halves, inside of which we check for inversions. In this merge function, we check whether the elements in the right half are inversions pairs with any of the elements on the left. Once an element is compared, the smaller element is sent up to a sorted array. The elements in both halves are already sorted, which enables us to only iterate through them once, allowing an $O(n)$ runtime. Thus, since the merge function recursively breaks the array down into halves in $O(lgn)$ time, the total runtime is $O(nlgn)$. The pseudocode for mergeInv is below. I do not implement the pseudocode for merge since it is so involved, but the basic idea was described above.

```
mergeInv(A,left,right)
    if (right-left) == 1
        return      mid=(right-left)/2 + left
    mergeInv(A,left,mid)
    mergeInv(A,mid,right)
    merge(A,left,mid,right)
```