Justin Baraboo
Design and Analysis of Algorithms
Assignment 2

March 17, 2015

1 Problem 1

Solution For this problem we will use that:
For $i < j$:
If $B[i] < B[j]$ then for any $S[k], k \geq i$ $B[i]$ will outperform $B[j]$. However, if $B[i] > B[j]$ then for any $S[k], k \geq j$ $B[j]$ will outperform $B[j]$. However, in the second case, if there exists a $S[q], q < j$ such that the profit is better in the lower indexes than the higher indexes, we will choose the lower. So we will start at the beginning of the buy array and look for a smaller element. If no element exists, we only need to find the largest Sell price within the sell array.
If there is a smaller, we find the highest sell price within $m...i-1$ where i is the index of the smaller buy price and m is the index of the buy price we are using.
We keep track of buy price index (in the first case 1) and the highest sell price within the above range.
We repeat this from the smaller buy price element we found until we cannot find any other smaller buy price, by either running out of elements or finding the smallest buy price.
We then compute the profit for each of these pairs, and pick the indexes of the pairs that give the highest profit.

```
FindSmaller( B[r...s] )
{
   i = r;
   while( B[r] <= B[i] & i <= s )
   {
```

```
      i++
   }
   if( i > s) return r
   else return i
}

FindMaxProfit( B[1...n], S[1...n])
{
   index = 1
   buyOptions[]
   sellOptions[]
   buyOptions.add(index)
   while( index <=n )
   {
      smaller = FindSmaller( B[index...n])
      if( index == smaller) //we found smallest Buy Price
         sell = indexofMax( S[index] ... S[n])
         sellOptions.add(sell)
         break while
      else
         buyOptions.add(smaller)
         sell = indexofMax( S[index] ... S[smaller -1])
         sellOptions.add( sell )
         index = smaller
   }
   profit[]
   for( i <= sizeof(buyOptions[]) )
      {
         profit.add( sellOptions[i] - buyOptions[i] )
      }
   maxProfitIndex = index of Max (
       profit[1]...profit[sizeof(profit)]] )
   return buyOptions[maxProfitIndex],sellOptions[maxProfitIndex]
}
```

Analysis:
Our input size is n, the number of places me may buy and sell. Our basic operation is the comparison operator (used to test for smaller buy prices and find the max sell price for the chunk of the available sell prices).
We iterate through the entire buy price array, and thus make n comparisions

trying to find smaller places to buy.
We will then iterate through the entire sell price array piecewise after finding a smaller buy price, yet will still only make n seperate comparisions total.
We then make m comparisions to find the best profit where $1 \leq m \leq n$.
Thus we have a worse case complexity of $3n \in O(n)$.
Thus $T(n) \in O(n)$.

2 Problem 2

Solution THIS ONE NEEDS TO KEEP TRACK OF WHICH ONE IS HAS THE LARGEST FINISHING TIME AND TRY TO ADD THERE FIRST
OTHERWISE IT IS RIGHT. PROOF IS WRONG AS PSEUDOCODE IS WRONG. USE HIS INDUCTIVE PROOF.
Activities are assumed to be a structure that has access to their starting and finishing times. We first sort the activities by finish time. We then will add the first available activity to the lecture hall 1. For the next activity, we will add it to the first lecture hall if its starting time is after the lecture halls latest finish time or we will add it to the second lecture hall. Once the second lecture hall has an activity, we will check against its latest finish time if an activity is compatible as well. We will do this until we run out of activities.

```
mergesortFinish(Activities[r..s])
{
   left[], right[], result[]
   if r >= s
       return Activities[r].finish
   else
   {
      mid = Floor( (r+s) / 2 )
       left = Activities[ r... mid ]
       right = Activities[mid+1 ... s]
       left = mergesortFinish(left)
       right = mergesortFinish(right)
       if last(left) <= first(right)
          append right to left
          return left
```

```
        result = merge(left, right)
        return result
    }
}

merge(left[r..s],right[f...k])
{
    result[]
    while length(left) > 0 and length(right) > 0
        if first(left).finish <= first(right).finish
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) > 0
        append rest(left) to result
    if length(right) > 0
        append rest(right) to result
    return result
}

scheduleActivities( LH1[], LH2[], Activities[1...n])
{
    //sort the activities by Finishing Time
    Activities[1...n] = mergesortFinish( Activities[1...n] )
    i1,i2 = 1
    LH1finish = -1
    LH2finish = -1
    for( i = 1, i <= n i++)
    {
        if( Activities[i].start > LH1finish)
        {
            LH1[i1] = Activities[i]
            i1 = i1+1
            LH1finish = Activities[i1].finish
        }
        else if( Activities[i].start > LH2finish)
        {
            LH2[i2] = Activities[i]
```

```
        i2= i2+1
        LH2finish = Activities[i2].finish
      }
      //else ignore it and continue
   }
}
```

Proof. Let's assume that our algorithm does not give the optimal activity number, say n, but instead gives $m < n$. By the pigeon-hole principle we can find at least one activity in either lecture hall whose activity could instead be replaced by at least 2 more activities. So $\exists A_i A_j \in Opt$ st $A_i A_j$ conflict with some $A_k \in Alg$. Where Opt is the optimum set and Alg is our algorithms set.
We assume that $i, j > k$. If $i, j < k$, then those activities must be at conflict with an activity in our algorithm that is earlier on as well (else they would be in our set as we pick by earliest finishing times, and thus not conflict), so choose that activity to be A_k instead. If $i < k < j$, then they must conflict with an activity earlier on as well as this one, and thus replacing the earlier and this one wouldn't net a gain in activities. Since $i, j > k$ and A_i, A_j, conflicts with A_k. So we know that:
$A_i.finish > A_k.finish$ & $A_j.finish > A_k.finish$
$A_i.start < A_k.finish$ & $A_j.start < A_k.finish$
as the activities conflict with ours.
Since both activities start before $A_k.finish$ and end after it, $A_i.start > A_j.finish$ and $A_j.start > A_i.finish$. So they conflict with each other.

Therefore they cannot be in the same lecture hall. Thus a contradiction as we assumed them both to be in the optimal activities lecture hall set and thus not conflict with one another.
This can easily be constructed for any $i_1, i_2 ... i_k$ conflicts in an optimal set with an activity in our algorithm under the same line of reasoning as each of these will have a finish time after our activity but a starting time before and thus conflict with one another.

Analysis:
Our input size is the number of activities that takes place. Our basic operation is the comparison operator which manifests in the mergesort and in our for loop. Mergesorting the finishing times takes $O(n \lg(n))$ time.

Our algorithm will never take more than $2n$ comparisons to schedule activities.
Thus we have a $T(n) \in O(n \lg(n))$.

3 Problem 3

Solution For this problem, we will use:
Scalar Distance: absolute distance between a house and a school. Answer Array: the array that keeps track of how many houses a school services. Since the houses and schools are sorted from the least to greatest distance from the west end, if a house chooses some school, i, subsequent houses need to only look at schools from i to n. So for each house, we start at School i (where i is initially 1 for the first school). If i is m, we merely increment its value in an Answer array. Else, we will compute the scalar distance of School i and School $i+1$ with a house.
If School i distance> School$i+1$ distance, $i = i+1$ and we repeat. If School i distance $\leq$ School$i+1$ distance, we increment the Answer Array at i and continue to the next house until we are done.

```
FindBestSchool( School[1...m], House[1..n])
{
   AnswerArray[1...m] \\initialized to all 0s

   i = 1
   for( h = 1, h <= n, h++)
   {
      notDone = true
      while(notDone & i < m)
      {
         sd1 = | H[h] - S[i] |
         sd2 = | H[h] - S[i+1] |
         if( sd1 <= sd2)
         {
            AnswerArray[i] = AnswerArray[i] + 1
            notDone = false
         }
         else
         {
            i = i+1
```

```
        }
      }
      if( i == m )
      {
         AnswerArray[m] = AnswerArray[m] + 1
      }
   }
   bestSchool = indexofMax(
       AnswerAray[1],AnswerArray[2]...AnswerArray[m] )
   return bestSchool

}
```

Analysis:
Our input size is the number of houses, n and number of schools m with $m < n$. Our basic operation is the comparison operator in the if statement. To count the number of basic operations, we can visualize our algorithm as paths in an $m \times n$ matrix.
We start in the lower left corner of the matrix, and when we increment h, we go right 1 cell.
When we increment i, we go up one cell. This works because we never go backwards in either h or i variables
We stop when we reach the right side of the matrix (when h = n).
This fully describes any possible state of our algorithm and the maximum length of a path is any path from the lower left corner to the upper right corner, which has length: $m + n < n + n = 2n$.
Finding the index of the maximum value of the answer array merely takes m, also initializing the answer array also only takes m, so these do not affect the order.
Thus $T(n) \in O(n)$.

4 Problem 4

Solution For this problem, we will use that:
$S_1 < S_2 < ... < S_n$
$F_{k_1} < F_{k_2} < ... < F_{k_n}$
$S_1 < F_1, S_2 < F_2, ..., S_n < F_n$

We will look at the first Finish Time F_{k_1}. Then we will start at which index is corresponds to in the start time array (the k_1 index) and find the first element from there that is larger, if one exists. If we find an element that is larger at index j, and with index i corresponding to the finishes times starting time (that is S_{k_1}), then there are $j-i-1$ incompatibilities with that activity as two activities are incompatible if one begins before the other but the other one starts before it ends. If a larger doesn't exist, there are $n-i$ incompatibilities. We do this for every element. We start our comparisons at the index j where the previous finishing time was less than.

In this algorithm, K(k1...kn) is an array of the permutation of the indexes 1...n for the finishing times to be sorted.

```
FindIncompatibilities( S[1...n], F[1...n], K(k1....kn) )
{
   incompatibilities = 0
   i = K(1)
   for( j = 1, j <n, j++)
   {
      finish = F(K(j)) \\gives F_k_j
      while( finish < S(i) & i <= n)
      {
         i++
      }
      if( i > n)
      {
         incompatibilities = incompatibilities + (n - K(j) )
      }
      else
      {
         incompatibilities = incompatibilities + (i - K(j) -1 )
      }

   }
   return incompatibilities
}
```

Analysis: Our input size is the number of activities (and thus starting and finishing times) n. Our basic operation is the comparision in the if.

Again, we can visualize our algorithm as paths in an $n \times n$ matrix.
In our algorithm we actually start at k_1 row but in the worst case this is still the lower left corner.
Again, when we increase j, we go right one cell.
When we increase i we go up one cell. This works because we never go backwards in either i or j variables
The maximum path is any path from the lower left corner to the upper right corner which will have length $n + n$.
Thus $T(n) \in O(n)$.