CSC373, Fall 2016 - Assignment 1

Name: Jinnan Lu
Student Number: 997698807

Due:Friday, 30 September 2016 at 04:59

Problem 1

a) since i, j <= n and i, j >= 1 and i <= j, then

```
when i=1, the maximum number of possible j is n (from 1 to n),
when i=2, there are maximum of n-1 choices for j,
similarly, when i=n, j is just n, so only one possible j exists,
therefore, the total number of intervals equals n+(n-1)+...+1=n(n-1)/2
```

b)

```
#Assuming the question only asks to return the maximum sum of a subarray
def max_interval(A):
        i, j = 1, 1
        curr_sum = A[i]
        max_sum = A[i]
        # simply return the maximum element if A
        # contains only negative elements
        if max(A) < 0:
            return max(A), i, j
        for m from 2 to n:
            curr_sum = max(A[m], curr_sum+A[m])
            if curr_sum > max_sum:
                max_sum = curr_sum
        return max_sum, i, j
```

This algorithm loops through index 2 to n of the list, where each element is checked for only once at each iteration, so the runtime is bounded by O(n).

If A contains only negative elements, the function returns the maximum value and the index of this value as both i and j. The rest of the code checks the sum of the subarray that ends at m in each iteration, where the ending position j gets updated if the sum doesn't decrease, and starting position i gets updated at the last time the sum became zero or went negative.

c)

```
def maxSum_contain_m(A, l, m, r):

    # Calculate the sum of left subarray
    sum = 0
    left_sum = min(A)
    for i from m to l:
        sum = sum + A[i]
        if (sum > left_sum):
          left_sum = sum

    # Calculate the sum of right subarray
    sum = 0
    right_sum = min(A)
    for i from m+1 to r:
        sum = sum + A[i]
        if (sum > right_sum)
          right_sum = sum

    # Return the sum of elements on both left and right subarrays
    return left_sum + right_sum

def maxIntervalSum(A, l, r):

   # Handle base case where there's only one element
   if (l == r):
     return A[l]

   #Find middle point
   m = (l + r)/2

   # Return maximum of the left sum, crossing sum and right sum
   return max(maxIntervalSum(A, l, m),
              maxIntervalSum(A, m+1, r),
              maxSum_contain_m(A, l, m, r))
```

Problem 2

a) Consider a BST with n nodes, where the root node has the highest frequency, and every node only contains exactly one child to its left except for the leaf node, their frequencies are in decreasing order as level goes deeper. In order to achieve the optimality, the BST is degenerated to a linked list. Therefore, the height of the tree is exactly the height of root which is bounded by Ω(n).

In this tree, node with higher frequency is put closer to the root, if interpreting it as a linked list, then the first element of the list is the root with the highest frequency, and the rest of them are sorted in decreasing order with respect to their frequencies, therefore, this tree is frequency-based and it's optimal.

b) $C(i,i) = f[i] * height(e[i])$
$C(i,j) = C(i,i) + min_{i<k<j}(C(i,k-1) + C(k,j))$

c)

```
def findOptimalTree(i, j):
    #return node i as the root since there's only one element
    if (i == j) then return e[i]
    for k = i to j:
        # make recursive calls for left and right sub-trees of each root k
        optimal_k = C(i,i)+ C(i,k-1)+C(k,j)
    return arg_k(min(optimal_k))
```

d) OptimalCost(i, j) = OptimalCost(i, k-1) + OptimalCost(k, j), where k is determined by checking each node between i and j. This algorithm is exponential, because the worst case running time of recursively computing the cost would be $O(n^2)$

Problem 3

for strategy a, this algorithm is not optimal, proof by contradiction:

assume there two contestant A and B: where $A_s = 3h$, $A_{b+r} = 1h$, $B_s = 1h$, $B_{b+r} = 2h$, it's easy to tell that the overall projected time of A is greater than B, according to the algorithm, we will have minimum completion time if send out A first and as soon as A gets out of the pool, we send B, then the overall completion time is 3h + 1h + 2h = 6h, however, if we send out B first, then A, then the completion time is not 1h + 3h + 1h = 4h, since 4h ¡ 6h, then this algorithm fails to minimize the completion time.

for strategy b, this algorithm is not optimal neither, proof by contradiction:

similar as above, let $A_s = 1h$, $A_{b+r} = 1h$, $B_s = 2h$, $B_{b+r} = 2h$, by running the algorithm, we send out A first then B since A swims faster than B, and the completion time is 1h + 2h + 2h = 5h, however, if we send out B first then A, the completion time is now 2h + 1h + 1h = 4h, which is better than the order suggested by this algorithm.

for strategy c, this algorithm does minimize the completion time, proof by induction:

assume this algorithm is correct, then there exist A and B where sending A out first and B right after minimizes the completion time, according to the algorithm, the completion time is $A_s + B_s + B_{b+r}$ and this value has to be better than the completion time of " sending B out first and A right after " , for the later case, the completion time is $B_s + A_s + A_{b+r}$, thus the algorithm implies:

$$A_s + B_s + B_{b+r} \leq B_s + A_s + A_{b+r}$$

Note if A and B have the same same projected biking and running time, this equation holds

by simplifying this equation we have:

$$B_{b+r} < A_{b+r}$$

Therefore, in order for this algorithm to minimize the completion time, $B_{b+r} < A_{b+r}$ has to be true, so we always send out the person whose sum projected time for biking and running is biggest.

Problem 4

a) In order to have as few fire stations as possible, each fire station needs to cover as many houses as possible, so

```
def placeFireStations(H):
    set = [] # create an empty cover set and fire station list
    F = []
    for i in range(0, len(H)):
        # Ignore any house that's within the range of the last fire station
        if F != []:
            if H[i] <= F[-1] + 4:
                continue
        # Put house in cover set for further processing
        if H[i] not in set:
            set.append(H[i])
        # If the last house is more than 4 miles away
        # from the closest fire station, then place a fire station at here
        if (i == len(H) - 1) & (H[i] not in F):
            F.append(H[i])
            set[:] = []
            return F
        # Place any house within the range of fire station in cover set
        if H[i] + 4 >= H[i+1]:
            set.append(H[i+1])
        # Place the fire station that covers all houses in set
        if (H[i+1] - 4 > set[0]) | (H[i] + 4 < H[i+1]):
            F.append(H[i])
            set[:] = []
    return F
```

b) My algorithm runs in linear time which is in O(n), it checks each house exactly once at each iteration, a house is added to a cover set that's being expanded as large as possible, while the algorithm still keeps the maximum distance between any two houses in this set to be within 4 miles, once a house out of the cover set's range is found, a fire station is put at the house i in set, where h_i is the largest among the set, thus it covers all houses in it. This implies that each cover set has exactly one fire station, the more houses each set covers, the less number of sets will be created, which leads to less number of fire stations being placed.