CSCE 411 Project

Sahil Palnitkar

1)

```
S[0] = 0
maxval = 0
for j = 1 to n // first pass
S[j] = max(S [j-1] + $a_j$, $a_j$)
for j = 0 to n // second pass
    if S[j] > maxval
        update maxval, remember subsequence
    return maxval and subsequence
```

This runs is 2 passes of size n. Therefore the total runetime is linear O(n).

2)

We can define 2 functions to help with the algorithm

p[i] = max of (max$_{j<i}${p[j] + alpha(m_i,m_j)xp_i) or p_i

alpha(m_i,m_j) = 0 if $m_i - m_j < k$

or 1 if $m_i - m_i \geq k$

where n are the number of locations and p[1..n] is the profit p(i) at each location i

```
expected total profit (n, p):
Profit[1..n] //max expected profit profit[i] at each location i
for i = 1 to n:
    profit[i] = 0
for i = 2 to n:
    for j = 1 to i-1:
        temp = profit[j] + alpha($m_i$, $m_j$) x p[i]
        if temp > profit[i]:
            temp = profit[i]
    if profit[i] < p[i]:
        profit[i] = p[i]
```

This algorithm has two loops which gives runtime of O(n^2)

3)

a)

```
valid_sentence(t,dict):
t[0] = true
for i from 1 to n:
    t[i] = false
    for j from i to i+1:
        $s_i$ = i – 1
        $s_j$ = j – 1
        substring = s[$s_j$ .. $s_i$ + 1]
        if dict(substring) and t[j-1]:
            t[i] = true
output t[n-1]
```

b)

```
output_list(t,dict,next_word):
t[0] = true
for i from 1 to n:
    t[i] = false
    for j from i to i+1:
        $s_i$ = i – 1
        $s_j$ = j – 1
        substring = s[$s_j$ .. $s_i$ + 1]
        if dict(substring) and t[j-1]:
            t[i] = true
            next_word[j-1] = i
prev = 0
words = []
for i in next_word:
    if i != 0:
        word = s[prev to i]
        words.pushback(word)
        prev = i

output words
```

5)
Build a 2d table T, where T[i,j] stores the longest palindrome in the string xi,…xj. If xi and xj are the same, they are part of the palindrome and T[i,j] is equal to T[i+1, j-1]. If they are different, they cannot be part of any palindrome.
T[i,j] will be T[i+1, j-1] if xi=xj otherwise max(T[i+1,j] , T[i,j-1])

If i = j, the answer is 1.

```
longest_palindrome(T)
for i = 1 to n:
    T[i,i] = 1
for i = n-1 to 1
    for j = i+1 to n
        T[i,j] according to the above function
```

output is T[1,n]

The runtime of this algorithm is O(n^2) as there are two for loops.

6)
$A[i,j] = \min_{i \leq k \leq j} \{A[i,k] + A[k,j] + d_{i,k} + d_{k,j}\}$

$d_{i,j} = (\mathrm{sqrt}((x_i - x_j)^2) + (y_i - y_j)^2))$ if $j-1 \geq 2$
    else 0

Based on these definitions
Assuming the vertices are ordered in a clockwise direction

```
min_cost_triangulation(p1,…pn):
A[i,j]
for i = 1 to n:
    A[i,i] = 0
for s = 1 to n-1:
    for i = 1 to n-s:
        j = i + s
        $A[i,j] = \min_{i \leq k \leq j} \{A[i,k] + A[k,j] + d_{i,k} + d_{k,j}\}$
return A[i,n]
```

This algorithm has 2 outer loops which take O(n^2) time and a loop which calculates each A[i,j] which takes O(j-i) time which is O(n).
The total runtime is O(n^3)

9)
a)
Suppose all codewords have length of at least 2, i.e. the tree has at least 2 levels. Let the weight of a node be the sum of the frequencies of all leaves that can be reached from that node. Assume the weights of the level 2 nodes are a, b, c, d. Without loss of generality, assume a and b are joined first, then C and D.
If a or b is greater than 2/5, then c and d are greater than 2/5. So a+b+c+d > 6/5 > 1 which is not possible. So either c > 2/5 (same for d > 2/5), then a+b > 2/5, so either a > 1/5 or b > 1/5 which means d > 1/5, we get a+b+c+d > 1 which is not possible. Therefore, there is guaranteed to be a codeword of length 1.

b)
Assume there is a codeword of length 1. There are 3 cases – The tree has 1 single level 1 node, 2 level 1 node, or 1 level 1 leaf node and 2 level 2 nodes with an unknown amount of leaf nodes below them. We will try to prove the contrapositive of the original statement, i.e. If there is a codeword of length 1, there must be a node with frequency greater than 1/3.
In case 1, our leaf has frequency 1 which is greater than 1/3. In case 2, if the tree has 2 nodes, one of them will have a frequency of at least 1/2. In case 3, The tree has 1 level 1 leaf (assume

weight a) and two level 2 nodes (assume weight b and c). We get, $b,c \leq a$. So $1 = a+b+c \leq 3a$, or $a \geq 1/3$.

10)
The longest codeword can be of length n-1. An encoding of n symbols with n-2 of them have probabilities 1/2, 1/4, …, $1/2^{n-2}$ and two of them have probability $1/2^{n-1}$ achieves this value. No codeword can ever be longer than length n-1. To prove this, consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least n + 1 leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves

11)
A matching is a collection of edges that do not share vertices. It is perfect if every vertex is covered by an edge from the matching.

```
check_matching():
for every vertex v, let seen[v] = false and matched[v] = false.
modified_dfs(s)
if matched[v] = false:
    return false    //no perfect matching
else:
    return true     //perfect matching

modified_dfs(v):
let seen[v] = true
for every neighbor u of v:
    if seen[u] == false:
        run modified_dfs(u)
    else:
        parent = u
if matched[v] == false:
    if matched[parent] == true:
        return false //no perfect matching
    matched[v] = true
    matched[parent] = true
```

Here, we keep matching vertices that have only one neighbor that has not been matched. We start with these single neighbor vertices, then we match vertices that have all but one neighbor matched.

The runtime uses slightly modified DFS and adds a constant number of steps. Therefore, the total runtime is O(n)

24)

Any shortest path from two vertices s to t must pass through v0. Thus, any such path is composed of a path from s to v0 and a path from v0 to t. We first use Dijkstra's algorithm to find the shortest path length from v0 to any other vertex, B[·]. We then reverse the graph and find the shortest path length to v0 from any vertex, A[·]. The shortest path from s to t that passes through v0 has length A[s] + B[t]. This takes O(|V | + |E|) log |V |) time. Note that we do not store all of the shortest path lengths directly, as that would require O(n^2) time.

31)
On an input <G,k> for CLIQUE we attach a tail of length k to every vertex of G. Now we run the oracle for KITE on this new graph with the same number k. Note that if the oracle returns YES then the original contains a k-Clique, and if it returns NO then the original graph does not contain a k-Clique. Therefore the problem is in NP as given the vertices that form a kite as a subgraph, we can check whether it is a valid kite or not.