1. Given a string S and access to a function makeSuffixArray(S) which creates and returns the suffix array of S in linear time,
determine the number of times that a string R occurs as a substring in S as efficiently as possible.

A. Using each entry in the suffix array, we can check each index's prefix (before the $ in the array')
to see if their first chars in S match to R. If there are, add matching char to a string T,
then increment and check again until you find T = R (add 1 to total if so and clear the string T). If there is any difference, or you reach the $, break and move to next entry.
If chars are matching and you reach $, check indexes immediatly following the $. If there is no potential for R to appear on the right side of the $, move to next index in suffix array
If the string we've been building is equal to R, then add 1 to total, if else do nothing. Return total when you reach the end of suffix array

B.
```
function numOccurencesA(S, R):
    total = 0
    startIndex = 0
    Q = makeSuffixArray(S)
    for i in range 0 to len(Q)-1:
        T = ""
        if Q[i][0] == R[startIndex]:
            T+=Q[i][j]
            if startIndex > len(R):
                startIndex++
            for j in range 1 to len(Q[i])-1:
                if Q[i][j] == R[startIndex]:
                    T+=Q[i][j]
                    if startIndex > len(R):
                        startIndex++
                    else
                        break
                else
                    break
        if T == R:
            total += 1
    return total
```

C.
T(n) = O(n*m)

3.
A. We can do BFS starting on s until we reach t. Once T is reached, we backtrack using the parents array and determining the distance from S,
and choose whatever is the least distance to s as our shortest path. Adj is the adjacency matrix of G.

B.
```
function countShortestO=Paths(G, s, t):
    shortest = []
    distance = []
    parents = []
    for int i = 0, to i < size(G):
        distance[i] = negative infinity
```

```
        parents[i] = i
    distance[s] = 0
    queue Q
    Q.push(s)
    while(!Q.empty):
        int u = Q.front
        Q.pop()
        for(int i = 0 to i < Adj[u].size):
            int v = Adj[u][i]
            if(distance[v] == negative infinity):
                distance[v] = distance[u] + 1
                parents[v] = u
                Q.push(v)

            if v == t:
                shortest.append(v)
                while we havent reached s:
                    parent = parents[v]
                    if distance[p] > distance[v]
                    shortest.append(p)
                    v = parent
    return size(shortest)
```

C. T(n) = O(V+E)

4.

A. To solve, we can use a 2D array A, where A[i][j] is the amount of ways to produce an amount 'j' using the first 'i' coins in the set of denominations. The base case(s) are A[0][j] = 0 for all of j, and A[i][0] = 0 for all of i, there is only one way to produce no money, which is to use 0 coins. For every i and j greater than 1, we set A[i][j] equal to A[i-1][j] + A[i][j-C[i-1]]. A[i-1][j] is the number of ways to produce amount 'j' using the 'ith' first coins in the set.

A[i][j-C[i-1]] is the amount of ways to get an amount of j-C[i-1] using the i coin denominatons from C. We only use the second term if j >= C[i-1]

The number of ways to produce an amount of money m from C is A[size(C)][m]

B.

```
function numWays(m,C):
    A = [m+1][C+1] //size of A
    for i in range size(C)+1:
        A[i][0] = 0
    for i in range 1 to size(C)+1:
        for j in range 1 to m+1:
            A[i][j] = A[i-1][j]
            if j >= C[i-1]:
                A[i][j] += A[i][j-C[i-1]]

    return A[size(C)[m]]
```

C. T(n) = O((C)(m))

5.

A. Let A be a 2D boolean array. A[i][j] represents whether or not we can form a subset of the first i elem ents of the mulltiset S that adds up to the value j.

Base cases would be A[0][0] = true, since we can always make an empty sum from an empty multiset, and the other is A[0][j] = false, since we cant have a non-zero sum

from an empty multiset. Once we build the array, we only need to return A[n][m] where n is the size of S, and m is the sum to determine if there is a subset R that adds up to m.

B.

```
function subsetSum(S,m)
n = S.size
A = [m+1][n+1] // array size
for all A[i][j]:
    A[i][j] = False
for all A[i][0]:
    A[i][0] = True
for i in range 1 to n+1:
    for j in range 1 to m+1:
        if j < S[i-1]:
            A[i][j] = A[i-1][j]
        else:
            A[i][j] = A[i-1][j] or A[i-1][j-S[i-1]]
return A[n][m]
```

C. T(n) = O(n*m)