1. 1. Algorithm:

   P(i , j) represents the palindrome longest palindrome subsequence of the slice from i-th to j-th character of the string. Initialize all the P(i,i) as s[i].

   Iterate over the substring lengths from 2 to n. For each length, iterate over the starting index from 0 to n - length:

   If s[i]=s[j], then: P(i,j) = s[i]+ P(i+1,j-1)+s[j] .

   Else, P(i,j)= max{P(i,j-1), P(i+1,j)}. Randomly select one of they are of the same length.

   In the end of the iteration, the longest palindrome is stored in P(1,n).

   2. Correctness: As the iteration go from short slices to the longest, the palindrome grows to the largest scale. So we can get the longest palindrome.

   Topological order: P(i,j) needs the value of P(m,n) (|n-m|<|i-j|). |i-j| is increasing, so it is a topologycal order.

   Base case: P(i,i) is i-th charater exactly.

   Induction:

   If s[i] != s[j] ,the longest palindrome palindromes considering s[j] without s[i] is : L1 : P(i+1, j)

   The longest palindrome palindromes considering s[i] without s[j] is :L2: P(i, j -1). If there is a longer palindrome palindromes s' from s[i] to s[j],because s'[0] != s'[max_length] , WLS, assume that s' doesn't contain s[j], s' <=L2, contradict! Then the longest palindrome palindromes considering from s[i] to s[j] is max(L1, L2).

   If s[i]=s[j], the longest palindrome iss[i]+ P(i+1,j-1)+s[j].

   3. Time complexity: The iteration is done at most $\sum i*(n-i)$times, so time complexity is $O(n^2)$

2. Algorithm:
   1. Calculate the total sum of all the elements.
   2. If n is not divisible by 3 or sum is not divisible by 3, return false, as it won't be possible to partition the elements equally.
   3. f(i , j,m) denotes there are 2 subsets A,B satisfying that their sum are j,m. A∩B=∅. A,B are subsets of the first i numbers.
   4. Iterate over the elements from 1 to n:
      - For each element ai, iterate over the possible sums j from 0 to sum/3:
        - For each possible sum j, iterate over the possible sums k from 0 to sum/3:
          - If f(i,j,k) is not none, set$f(i, j+a_i, k)$ and $f(i, j, k+a_i)$ accordingly as well.
   5. If f(n,sum/3,sum/3) is not none, we get the division, else there's no such divison.

   Correctness:

   As the iteration goes through all possible ways of division, the required division can be found if exists.

f(i,0, 0) = none.

If f(i - 1, j - a[i] , m) exist (in other word != None) , put a[i] into the first subset can be f(i, j,m) obviusly.

If f(i - 1, j , m - a[i]) exist (in other word != None) , put a[i] into the second subset can be f(i, j ,m) obviusly.

If f(u,j,m) exists, for i>u, f(i,j,m) exists.

Time complexity:

According to the iteration, the time complexity is $O(sum^2 n)$.

3. Denote the vertices in the graph as V = {v_1, v_2, ..., vn}, where v_1 is the start point s. Define the DP state as follows: dp(i, U) represents the maximum profit minus cost achievable by starting at vertex v_i and visiting the subset of vertices U. Here, U ⊆ V \ {v_i}.

   Algorithm:

   1. Initialize the DP table with dp(i, U) = -∞ for all i and U.
   2. For each vertex v_i ∈ V, set dp(i, ∅) = p(v_i) - c(v_i, v_1), where c(v_i, v_1) represents the cost of the edge (v_i, v_1). This represents the profit minus cost of returning to the start vertex without visiting any other vertex. c(v_i, v_1)=+∞ if there's no edge.
   3. For each subset of vertices U ⊆ V \ {v_1}, start from small sets to large ones, regarding the number of elements:
      - For each vertex v_i ∈ U:
        - For each vertex v_j ∈ U \ {v_i}:
          - Update dp(j, U) as follows: dp(j, U) = max(dp(j, U), dp(i, U \ {v_j}) + p(v_j) - c(v_j, v_i))
   4. Compute the maximum profit minus cost as follows: max_profit_minus_cost = max(dp(j, V \ {v_1}) + p(v_1) - c(v_1, v_j)), for all v_j ∈ V \ {v_1}, c(v_1, v_j) != +∞ .
   5. Trace back the optimal path to retrieve the vertices included in U that yield the maximum profit minus cost.

   Correctness:

   1. Topological order:

      Because for the sets, we follow an increasing order of size. So when we subsets are always gone through before we need their results.

   2. Induction:

      c(v_i, v_1)=+∞ if there's no edge

      dp(v,{v})=p(v)-2c(s,v)

      dp(u,∅) = 0

      dp(i, U \ {v_j}) + p(v_j) - c(v_j, v_i) is a possible value of dp(j, U).

   Time complexity:

   Number of sets is 2^n, according to the iteration, the total time complexity is $O(n2^n)$

4. 1. 1. Algorithm:

f(i) is the maximum revenue considering first i elements, g(i) is the max continuous subsequence containing a_i.

Initialize g(1) = a_1, f(1)=max(0,g(1))

Iteration: for i from 2 to n:

1. $g(i) = max(g(i-1) + a_i, a_i)$
2. $f(i) = max(f(i-1), g(i))$

2. Correctness:

Assume that g(j) (j < i) are all correct, if g(i-1) <0, g(i)=a_i. If g(i - 1) > 0 , $g(i) = g(i-1) + a_i$

Assume that f(j) (j < i) are all correct, if g(i)<f(i-1) then a_i should not be included, f(i)=f(i-1). Else a_i should be included, so f(i)=g(i).

3. Time complexity

According to the iteration, the time complexity is O(n), because we need to calculate all f(i) and g(i).

2. 1. Algorithm:

f(i) is the maximum revenue considering first i elements, g(i) is the max valid subsequence containing a_i

Initialize g(i) = a_t for 1 to L, f(i)=max(0,g(i))

Iteration: for i from L+1 to n:

1. $g(i) = max(g(i-m) + a_i, a_i)$ for all m from L to R
2. $f(i) = max(f(i-1), g(i))$

2. Correctness:

Assume g(j) (j < i) are all correct, if there is a subsequence S containing a_i that is >g(i), then:

S - {a_i } > g(i) - a_i, g(i-1) is not correct. Contradict. So g(i) is the largest.

Assume that f(j) (j < i) is correct, if there's a larger S: When S doesn't contain ai, S<=f(i-1), contradict. When S contains a_i, S<=g(i-1), contradict. So f(i) is correct.

3. Time complexity

Calculating g(i) takes O(n) here, so according to iteration the time complexity is $O(n^2)$.

3. 1. Algorithm:

Define a priority Q, the elements can be considered maximum subsequence before i.

When i-L>0,we can update:

1. if g(i - R - 1) is Q.head(): popHead.
2. if Q is not empty and g(i - L) >= Q.back() :popBack
3. pushBack(g(i-L)).

So,g(m) > g(n) , i - R <= m < n <= i - L

We can get g(i) by: g(i) = max(a[i], Q.head() + a[i])

2. Correctness: We need to prove that Q.head() is the max subsequence before i.

   Because g(i - R - 1) is poped, g(i - L) is pushed, all valid subsequences are contained. So Head must be the max.

   Then with the same steps as 2, we can prove that f(i) and g(i) are correct.

3. Time complexity

   Now calculating g(i) takes only O(1), so it is O(n) in total.