Question-1

Task is find a subset of shops such that profit is maximised and no two shops have distance less than 'k' between them.

Arr[i] is defined as maxprofit of subsets of shops considering only first i shops and definitely containing ith shop in it.

```
def maxprofit(d,p,k):
    #dynamic programming approach
    arr=[0]*len(p)
    arr[0]=p[0]
    for i in range(1,len(p)):
        arr[i]=p[i]
        for j in range(0,i):
            if(d[i]-d[j]>=k):
                arr[i]=max(arr[i],arr[j]+p[i])
            else:
                break
    return max(arr)
```

$$arr[i] = \max(p[i], arr[j] + p[i])$$
$$\text{where } j \in [0, i-1]$$

↑ Optimal substructure.

Time complexity of the Algorithm = $O(n^2)$, $n = len(d)$
As we have two for loops running in the code.

- we created substructures in such a way that all possible cases are covered. So the algorithm is bound to be correct

Question 1b

Greedy / Fat orcale approach may not work in all the cases, as it inhibits some of the shops from taking part in final optimal solution.

Example:

```
d = [4, 5, 6]  # distances
profits = [10, 12, 11]
K = 2
```

with greedy approach we get maxprofit = 12,
where as optimal solution has maxprofit = 21

Question-2

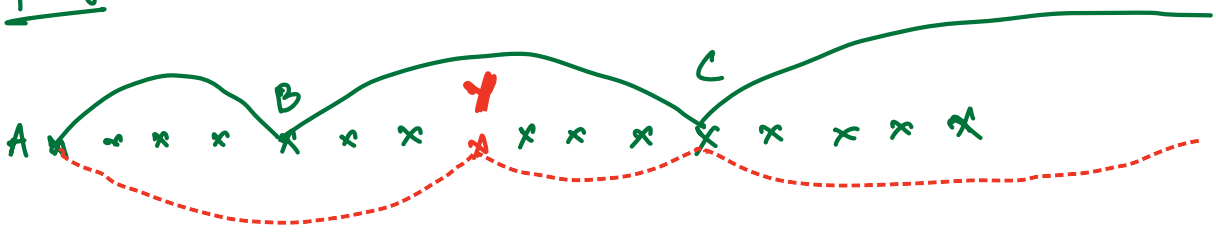
Task is to find the minimum no. of stops required to reach the final destination, such that max distance that can be travelled without stopping is m.

```python
def min_stops(d,m):
    # greedy approach
    stops=0
    prev_stop=0
    i=0
    # check if there is a gap greater than m, as it is not possible to cover such a gap
    for i in range(len(d)):
        if(d[i]-d[i-1])>m:
            return -1
    i=0
    while(i<len(d)):
        if(d[i]-d[prev_stop]<=m):
            # if the distance between the current stop and the previous stop is less than m, then we can continue
            i+=1
        else:
            # if the distance between the current stop and the previous stop is greater than m,
            # then we need to stop at the destination before the current stop
            stops+=1
            prev_stop=i-1
    return stops+1 # +1 because we need to stop at the last stop
```

Approach: Starting from first position, try not to stop till the distance covered from starting position is just less than m.

Complexity of the solution is O(n), n is no. of destinations, As we consider each destination exactly once in the while loop.

Visual proof:



Let us assume that ABC is part of the optimal solution.

Let Y be the destination, where ship cannot travel anymore without stopping from 'A'

So, if C is reachable from B, it is also reachable from Y.

So, Instead of stopping at B, if we choose to stop at Y. It would not effect our min. no. of stops.

This proves the correctness of greedy approach.

Question-3

Given that C_i is the time required to serve i^{th} customer.

We need to minimise T, where

$$T = \sum_{i=1}^{n} (\text{waiting time of each customer})$$

Proposed Algorithm : Arrange customers in the ascending order of their serving time, to obtain min (T).

Let us prove this by contradiction.

Assume $C_1 < C_2 < C_3 < \cdots < C_n$

then by the above greedy approach

$$T = nC_1 + (n-1)C_2 + (n-2)C_3 + \cdots + C_n$$

So our T will be of form

$$x_1C_1 + x_2C_2 + x_3C_3 + \cdots + x_nC_n$$

where $x_1, x_2, \cdots, x_n$ are choosed from $\{1, 2, \cdots, n\}$ without replacement.

Let us assume in our optimal solution, there exists x_k, x_l such that $x_k < x_l$ where

$C_k < C_l$

$x_k c_k + x_l c_l$ is the sum contributed by these two terms.

but $x_k c_k + x_l c_l$ is always greater than $x_l c_k + x_k c_l$

$$x_k c_k + x_l c_l > x_l c_k + x_k c_l$$

As $$(x_l - x_k) c_l > (x_l - x_k) c_k$$

So in our final solution $x_k > x_l$ where $c_k < c_l$

⇒ Customer with lowest wait time should have higher coefficient.

⇒ customer with lowest wait time should be served first

Pseudo code (# C is array with index starting from 1)

```
def min_T (C):
    c = sort (C)  #ascending order
    sum = 0
    for i = n → 1
        sum += i x C[i-(n-1)]  # Add wait-time contributed due to each customer
    return (sum)
```

Time complexity of the above code is $O(n \log n)$ as we sort C in the process of finding min (T).

Question-4

Task is to find the Longest palindromic subsequence in a given string.

Let $T(i,j)$ be the longest palindromic subsequence by considering only $A[i:j]$ (# including $A[i]$ & $A[j]$)

assume all arrays start with index 1.

So our required result is $T(1,n)$

$\Rightarrow T(i,j)$ is valid only when $i \leq j$

$$T(i,j) = \begin{cases} 2 + T(i+1, j-1) & \text{if } (A[i] == A[j]) \\ \max(T(i,j-1), T(i+1,j)) & \text{if } (A[i] != A[j]) \end{cases}$$

Edge cases

$$T(i,i) = 1$$

$$T(i, i+1) = \begin{cases} 2 & \text{if } A[i] == A[j] \\ 0 & \text{if } A[i] != A[j] \end{cases}$$

* we created substructures in such a way that all possible cases are covered. So the algorithm is bound to be correct.

Algorithm (Pseudo Code)

```
def max_len_palindrome(A):
    T = zeros(len(A), len(A))  # matrix of size (n,n)
    for i = n → 1:
        for j = i → n:
            if (i == j):  # edge cases
                T[i][j] = 1
            elif (j == i+1):
                if A[i] == A[j]:
                    T[i][j] = 2
                else:
                    T[i][j] = 0
            else:  # sub structure.
                if A[i] == A[j]:
                    T[i][j] = 2 + T[i+1][j-1]
                else:
                    T[i][j] = max(T[i+1][j], T[i][j-1])
    return T[1][n]
```

→ Complexity of the algorithm is $O(n^2)$ where n is the length of string A, as we have two 'for' loops in the algorithm and each subproblem is dealt exactly once.

Question-5

Task is to find if the given amount A is possible with given denominations.

Define T(s) = True if the amount s is possible with given denominations, T(s): False, otherwise.

Algorithm

```
def is_it_possible (A, D):
    T = bool (A+1, False)   # Define boolean Array of size (A+1) with false.
    T[0] = True
    for i = 1 -> A:         # Let the index of array start with 1.
        temp = False
        for j in D:
            if j > i:
                continue
            else:
                temp = temp (or) T[i-j]
        T[i] = temp
    return T[A]
```

for every number from 1 to A we traverse through the all denominations once. So Complexity of the algorithm is $O(nA)$. $(n = |D|)$

we created substructures in such a way that all possible cases are covered. So the algorithm is bound to be correct.

Question - 6

Task is to find if the given 'A' is possible to acheive with given denominations D using atmost K coins.

Definitions

T(s) = True if the amount s is possible with given denominations, T(s): False, otherwise.

N(s) = min. no. of coins required to acheive s with given denominations.

N(s): ∞ if T(s): False.

Complexity

for every number from 1 to A we traverse through the all denominations once. So Complexity of the algorithm is $O(nA)$. $(n = |D|)$

we created substructures in such a way that all possible cases are covered. So the algorithm is bound to be correct.

```
def is_it_possible(A, D, K):
    T(S) = bool(A+1, False) # initialize T, N
    N(S) = [∞]*(A+1)

    N(0) = 0
    T(0) = True
    for i = 1 → A: # Find T(S), N(S) for all values
        temp = False
        temp_num = ∞
        for j in D:  # traverse through all denominations
            if j > i:
                continue
            else:
                if T[i-j] == True:
                    temp = True
                    temp_num = min(temp_num, N[i-j]+1)
        T[i] = temp
        N[i] = temp_num
    return (T[A] and (N[A] <= K)) if min number of
```

coins to achieve is greater than K, then it is not possible to achieve the given sum with only K coins