Homework 3 - training problems

Pratyay Dutta (Codeforces ID : pdutt005)

February 17, 2024

Contents

1 Challenge problems

1.1 Motorcade : Submission ID - 246420168

To solve this problem:
State : s[i] = total time for the first i cars to cross the bridge.
Recurrence : To compute s[i], we need to enumerate all possible second last splitting point j such that the first j-1 cars will cross the bridge using the best time they can achieve (which is s[j-1]) and the jth to the ith car will go as a batch. The total weight of the cars crossing in each case should always be lower than the weight limit W. The time needed to cross the bridge by a batch of cars is the maximum time that can be taken by a car in that group i.e L/minimum speed(the batch of selected cars). s[i] will be the split which gives us the minimum time for first j cars to cross. The recurrence can be written as : s[i] = min(s[j-1] + maximum time taken by a car in the group to cross) where $0 < j < i$ and sum(weights(car_i to car_j)$<= W$.
Base case : s[0] = 0 since total time for first 0 cars to cross is 0.
Answer : s[n] = minimum total time for first n cars i.e all the cars to cross the bridge. **Code explanation** : We first initialise the dp array with all 0 values. We loop for all i = 1 to n+1. We consider each car starting from the first car and check if adding that car satisfies the weight limit. If it does, then we continue to find the car with the minimum speed from among the cars selected and find the maximum time to cross the bridge $time_to_cross$. There can be two cases:

1. We dont consider the jth car. In that case, s[i] remains unchanged.
2. We consider the jth car and s[i] = time for first j-1 cars to cross + $time_to_cross$ for the cars from j to i.

s[i] = min(1,2) since we want to minimize the time. The answer is s[n] which is in hours since the length L is in km and the speeds s_i are in km/h. To convert this to minutes, we multiply the value by 60. That is our answer.
Time complexity : The time complexity of this algorithm can be analyzed based on its nested loops and the operations performed within those loops: Outer Loop (over i): This loop iterates over all trucks, with each iteration considering an ending point for a segment of trucks crossing the bridge. The outer loop runs n times, where n is the number of trucks.
Inner Loop (over j): For each i, the inner loop iterates from i back to 1, effectively considering all possible starting points for a segment ending with truck i. In the worst case, this loop runs i times for each i, leading to a sum of iterations equal to 1+2+...+n, which is n(n+1)/2 (a quadratic number of iterations).
Operations within the Inner Loop: The main operations within the inner loop are basic arithmetic operations (addition, comparison, division) and assignments. These operations are considered constant time, O(1).
Given the above, the time complexity of the algorithm is determined by the nested loops, leading to a total of O(n(n+1)/2) iterations of constant-time work. This simplifies to overall time complexity $O(n^2)$

1.2 Smart and fun : Submission ID - 246645016

The dp solution for this problem is not a vanilla dp matrix problems. The dp data structure is a dictionary here.
State : The state in this DP solution is represented by a dictionary dp, where each key-value pair represents a potential combination of total smartness (TS) and total funness (TF) that can be achieved by selecting a certain subset of cows. Specifically, the key in the dictionary represents the total smartness (TS) of a group of cows, and the value represents the maximum total funness (TF) that can be achieved for that TS.

- Key (TS): The total smartness accumulated by selecting a certain subset of cows.
- Value (TF): The maximum total funness associated with the accumulated total smartness (TS).

Recurrence : The recurrence relation in this DP solution determines how the state transitions from one to another as each cow is considered for inclusion in the group. When considering whether to include a particular cow with smartness s and funness f, the solution updates the DP table based on the following logic:

1. Including the cow: This creates a new state (ts + s, tf + f). If ts + s already exists as a key in the DP table, the solution compares and updates the associated value with max(dp[ts + s], tf + f), ensuring that the value represents the maximum possible total funness for the new total smartness.
2. Excluding the cow: The existing state (ts, tf) remains unchanged in the DP table.

Base case : dp0:0 meaning that a state where the total smartness is 0, the total funniness is 0 meaning that no cows have been considered.
Answer:After considering all cows, the solution searches for the maximum sum of TS and TF among all valid states (where both TS and TF are non-negative) in the DP table. This is done by iterating over all key-value pairs in the DP table, summing each pair (ts + tf), and identifying the maximum sum where both components are non-negative.
Time complexity :The function iterates over each cow, of which there are n. For each cow, the algorithm iterates over the items in the ‘dp‘ dictionary. The size of ‘dp‘ can grow in each iteration. In the worst case, if each cow contributes a unique combination of smartness and funness, the size of ‘dp‘ can approach the number of subsets of cows, which grows exponentially. However, realistically, since ‘dp‘ is updated based on the sum of smartness and funness, its growth is constrained by the sum of these values rather than the number of cows directly. Let’s denote the average number of items iterated in ‘dp‘ as m.
Within the nested loop, the algorithm performs a constant-time check and potentially updates the ‘dp‘ dictionary. The update operation itself is $O(1)$, but it happens within the context of the nested iteration.
Given these steps, the time complexity of the algorithm can be approximated as $O(n \cdot m)$, where: - n is the number of cows, and - m is the average number of items in the ‘dp‘ dictionary during each iteration. The critical factor affecting m is how the smartness and funness values are distributed among the cows. In the best-case scenario, where the smartness and funness values are highly repetitive, m would be relatively small because many cows would fall into existing categories in the ‘dp‘ dictionary. In the worst-case scenario, where each cow adds a new, non-overlapping combination of smartness and funness, m could grow significantly, but still bounded by the total possible sums of smartness and funness rather than the number of cows directly.

1.3 Select courses : Submission ID : 246745562

We implemented a brute force approach to solve this using dp. My idea was to treat this similar to a 0/1 knapsack problem where we can pick once from a group. The groups here are the courses without prerequisites and the corresponding courses whose prerequisite is the main course. For each group, we have to find all the combinations which can exist.
I got partial marks (0.3) since i passed 3 test cases and the time limit was exceeded for the rest 7.
For every group, we create a tree. The root is the main course and the children are the courses whose prereq is the parent. After, we make a tree for each group, we find all the possible combinations we can get from that group. For the given example, the groups are 1 and 4. The possible combination of courses from group 1 : [[1],[1,2],[1,2,3],[1,2,5],[1,2,3,5]]. Now we have to choose either of this combinations from this group when we consider this group. However, while making the combinations, we make the filter that the length of the combination cannot exceed m. This is now a 0/1 knapsack problem after getting the combinations. **State** : s[i][j] = maximum credit that we can get using j courses considering the first i groups.
Recurrence : There can be two cases:

1. We dont pick any course from group i. s[i][j] = s[i-1][j]
2. We pick a course from group i. Therefore, for all combinations k possible in group i, we have to find max(s[i-1][j-k.length] + k.credits)

To fill out the dp matrix we need : s[i][j] = max(1,2).
Base case : s[i][0] = s[0][j] = 0. Considering 0 courses or the first 0 groups will have maximum benefit = 0
Answer : s[n][m], where n is the number of groups and m is the number of courses allowed to take.
Time complexity : Lets say for each course not having a prerequisite, there are k courses which require it as a prerequisite. Let n be the number of courses without prerequisites. Let the number of combinations for each group be O(f(n)). To make the items list, we need O(nf(n)) computations. To populate the dp matrix, we loop though each group and for each budget range from 1 to W. The third for loop iterates for every possibility in each group which is 4. Therefore the time complexity to fill up the dp matrix is $O(n.W.f(n)) = O(nWf(n))$. f(n) is dependent on the input data and it can go upto n!. Therefore, our total time complexity of our algorithm id O(nWf(n)).

1.4 Online CS program : Submission ID - 246722176

This problm is similar to the first written problem. I treated it as a variation of the 0/1 knapsack problem. We consider groups. The number of groups are the number of main courses there are. In each group, a main course is there and the discussion courses of that course are there. We have to choose the courses from the groups in such a way that it maximises the course value. For each group there can be multiple ways we can consider a group :

1. We pick only the main course from the group.
2. We pick the main course and one other course (if available)
3. We pick the main course and two other courses (if available)

We have to find all possible combinations in which the group can be considered. For example in a group of courses where the main course is 1 and the discussion courses are 2 and 3. When we are considering this group, we can consider the following combination of courses : [1],[1,2],[1,3],[1,2,3]. Therefore, we create a list *items* which contains all possible combinations in which all the groups can be considered. For this, we first create a list $main_courses$ which records the discussion courses of each main subject and the (value,price) pair for all possible combinations of a group. We create a *items* list to track the (price,value) pair of each combination in each group i.e items[i] = list of all (price,value) pairs of group i. Now our problem have been simplified to a vanilla 0/1 knapsack problem where we have to find out the maximum value we can get given a set of groups *items* and only one element can be picked from that group(i.e the group can be considered only once). Therefore, we formulate a dp approach to solve this knapsack problem.
State : s[i][j] = maximum value that we can get at budget j considering the first i groups.
Recurrence : There can be two cases:

1. We dont pick any item from group i. s[i][j] = s[i-1][j]
2. We pick an item from group i. Therefore, for all combinations k possible in group i, we have to find max(s[i-1][j-k.price] + k.value)

To fill out the dp matrix we need : s[i][j] = max(1,2).
Base case : s[i][0] = s[0][j] = 0: considering 0 budget or the first 0 groups will have maximum benefit = 0
Answer : s[n][W], where n is the number of groups and W is the budget.
We have implemented a iterative solution of this dp formulation where we initialise the dp matrix to all zeros. We loop over the number of groups i and the budget j and find the max of s[i-1][j] and s[i-1][j-$combination_k.price$] + $combination_k.value$. After populating the dp matrix, we output the value at s[n][W].
Time complexity : Considering worst case where for every main course, there are 2 discussion courses. Let n be the number of main courses. The number of combinations for each group will be 4. To make the items list, we need O(4n) computations. To populate the dp matrix, we loop though each group and for each budget range from 1 to W. The third for loop iterates for every possibility in each group which is 4. Therefore the time complexity to fill up the dp matrix is $O(n.W.4) = O(nW)$. The time complexity to return the value is O(1). Therefore, the total time complexity of our algorithm id O(nW).

2 Everything on sale

2.1 In optimal solution, we never buy two from a same group

The objective of our solution is to maximise the total benefit $p_i - t_i$ from purchases. We formulate the data in such a way that some items are grouped together and the items which are standalone are also considered a group of 1 item and hence that individual item can have the discount. The total benefit, we can reap from a group is the maximum of the benefit from each item in the group given that they satisfy the cost ceiling. For all the other items in the group, the contribution to the benefit is 0.
Let us consider an optimal solution S = $a_1, a_2, ...a_n$ containing two elements a_m, a_n from the same group g_k. Since only one of the items can have the discount, the contribution to benefit is $max(a_m - t_m, a_n - t_n)$.
Let there be another solution S' = S-a_n where a_n is the item to which we cannot put discount.
Therefore benefit(S) = benefit(S') since a_n is sold at p_i hence benefit(a_n = 0).
Also, $size(S') < size(S)$ and S' is within the weight limit. Yihan prefers a solution with fewest items if the benefit is the same for both solutions. So we see that S' is an optimal solution and S is not. Hence, our original assumption was wrong.
S cannot be an optimal solution. This proves that an optimal solution cannot contain more than 1 element from each group.

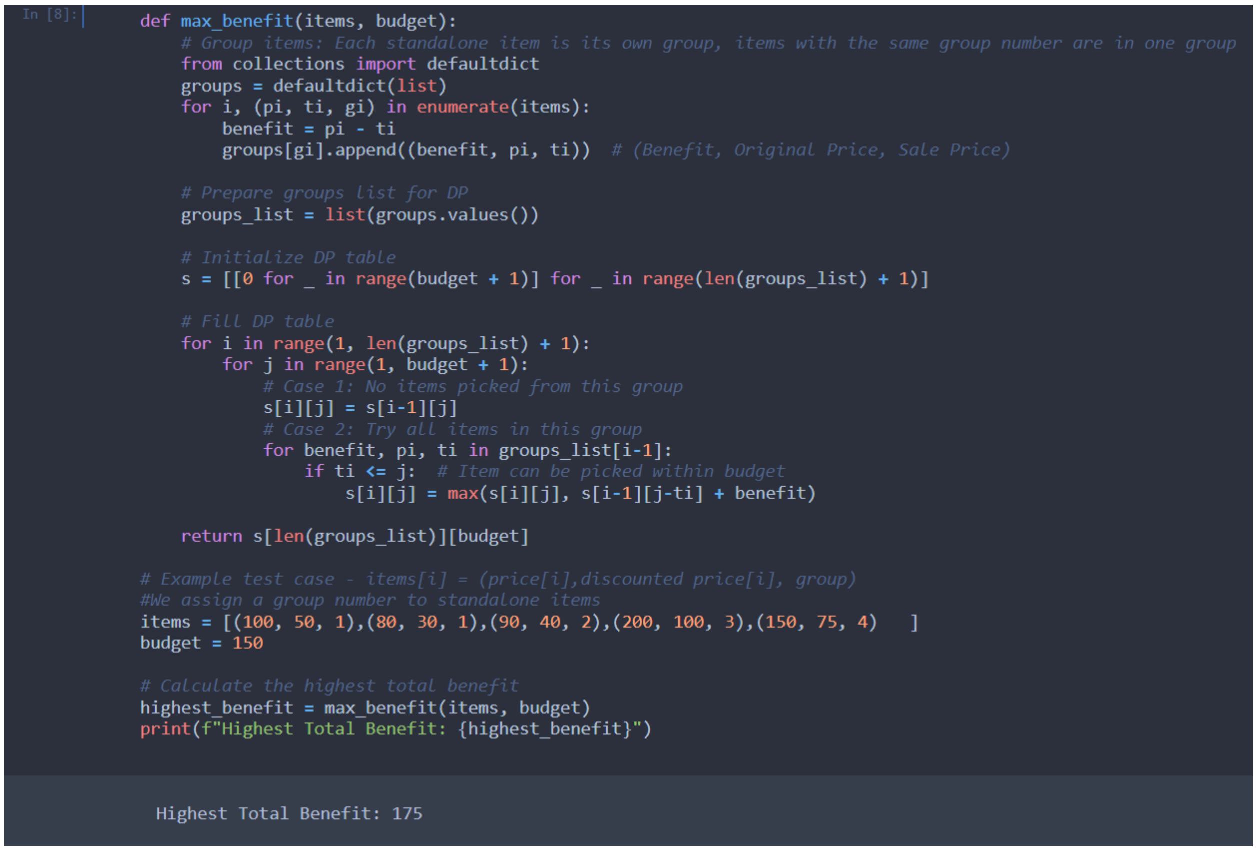
2.2 Describe algorithm

Setup : We consider groups. For any item which is standalone, we consider that item to be a separate group of one element. If no item can be picked from a group at given budget, we go on to the next group without adding anything to the maximum benefit. If any item can be picked from a group, we check for each element. We have to choose only one element from a group. So we check for each item k in the group and select it and update the dp matrix. Since, we will select only one item from each group,we will consider that item at the discounted price.
State : s[i][j] = The maximum benefit we can get considering budget j and the first i groups.
Recurrence : There are two possible cases :

1. s[i-1][j] : If no items can be picked from group i
2. For all items in group i, max(s[i-1][j-i_k] + $p_k - i_k$) : If any item k can be picked from group i, we will update our matrix with the max value obtained after considering each item.

s[i][j] = max(1,2)
Base case : s[0][j] = s[i][0] = 0
Max benefit from first 0 groups = 0 and max benefit with budget 0 = 0
Answer = s[n][W] where n is the number of groups and W is the budget.
Example input : items = [(100, 50, 1),(80, 30, 1), (90, 40, 2), (200, 100, 3), (150, 75, 4)]
budget = 150
The maximum benefit from this input will be 175 which we will get by picking item 2 from group 1(benefit = 50), item 3 from group 2(benefit = 50) and item 5 from group 3(benefit = 75). **Python implementation**

and output:



```
In [8]:
        def max_benefit(items, budget):
            # Group items: Each standalone item is its own group, items with the same group number are in one group
            from collections import defaultdict
            groups = defaultdict(list)
            for i, (pi, ti, gi) in enumerate(items):
                benefit = pi - ti
                groups[gi].append((benefit, pi, ti))  # (Benefit, Original Price, Sale Price)

            # Prepare groups list for DP
            groups_list = list(groups.values())

            # Initialize DP table
            s = [[0 for _ in range(budget + 1)] for _ in range(len(groups_list) + 1)]

            # Fill DP table
            for i in range(1, len(groups_list) + 1):
                for j in range(1, budget + 1):
                    # Case 1: No items picked from this group
                    s[i][j] = s[i-1][j]
                    # Case 2: Try all items in this group
                    for benefit, pi, ti in groups_list[i-1]:
                        if ti <= j:  # Item can be picked within budget
                            s[i][j] = max(s[i][j], s[i-1][j-ti] + benefit)

            return s[len(groups_list)][budget]

        # Example test case - items[i] = (price[i],discounted price[i], group)
        #We assign a group number to standalone items
        items = [(100, 50, 1),(80, 30, 1),(90, 40, 2),(200, 100, 3),(150, 75, 4)   ]
        budget = 150

        # Calculate the highest total benefit
        highest_benefit = max_benefit(items, budget)
        print(f"Highest Total Benefit: {highest_benefit}")

         Highest Total Benefit: 175
```

Given the dimensions of the DP table and the iteration over items, the time complexity of filling the DP table can be approximated as O(gWk). Since gk=n, this simplifies to O(nW).

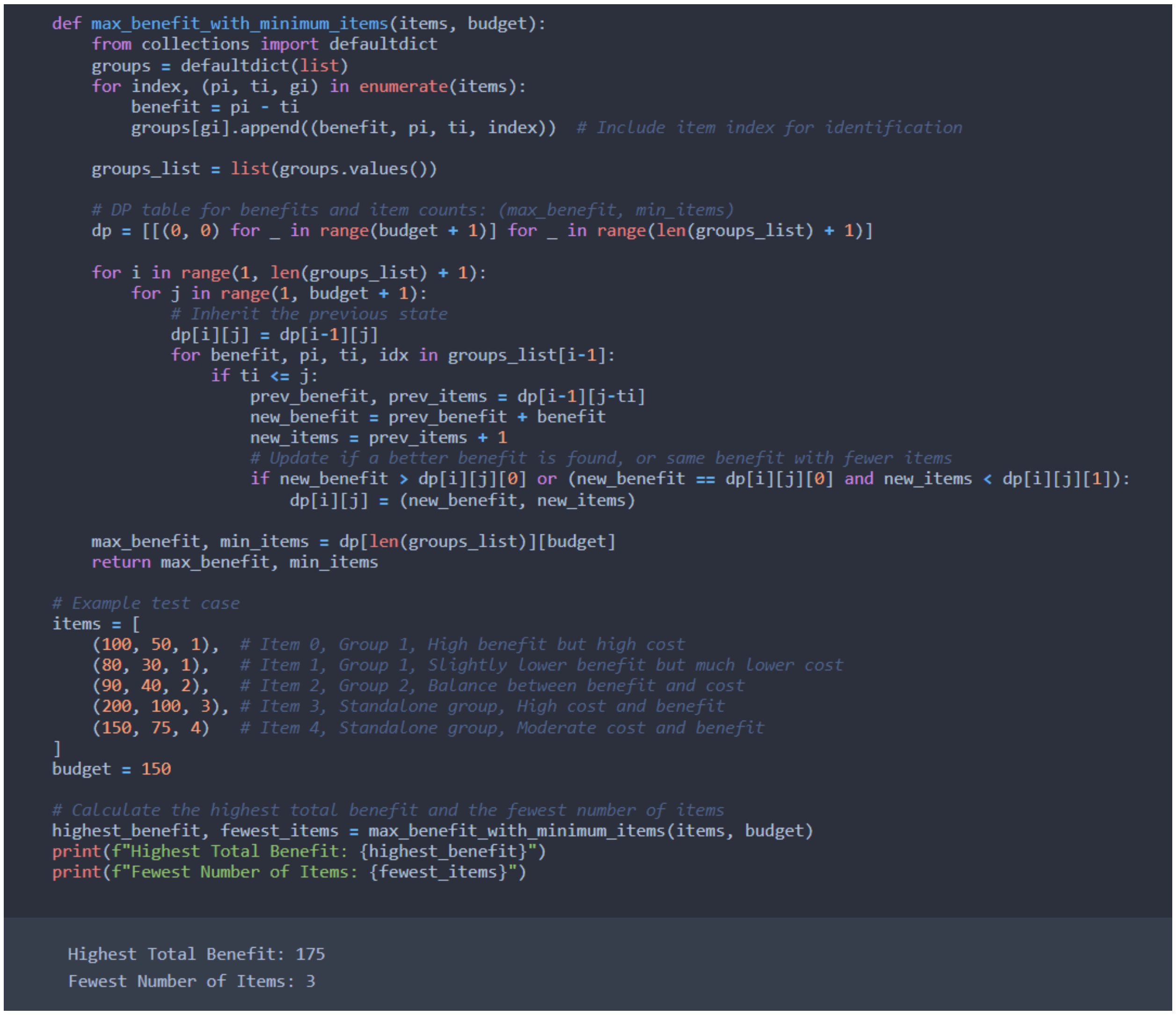
2.3 Fewest number of items

To find the fewest number of items to maximise the benefit, we have to keep track of the minimum number of items needed to achieve this benefit. We need to store one more value in each cell of our dp matrix. Instead of just storing the maximum benefit we can get with first i groups and budget j, we store the maximum benefit and the minimum items in this cell as a tuple.
State : dp[i][j] = (max benefit, min items).
Recurrence:We consider both the benefit and the number of items. If a new maximum benefit is found, we update the benefit and reset the item count. If the same maximum benefit is found, check if the new solution uses fewer items and update accordingly. For each group and each budget j, there can be two cases:

1. s[i-1][j] : Not picking any item from the current group.
2. Picking an item from the current group. Update the DP table only if:
   - The benefit is increased
   - The benefit is the same as the maximum found so far, but with fewer items.

Not picking any item from the current group. This involves comparing (s[i-1][j-i_k] + $p_k - i_k$, min items + 1) with the current state and updating if it leads to an improved solution based on the criteria above.

Answer = s[n][W] where n is the number of groups and W is the budget. This returns a tuple which gives us the maximum benefit considering all groups and budget W and the minimum number of items picked to reach the maximum benefit.
With the given example, the minimum number of items to get to the maximum value = 175 is 3 (items 2,3,5) **Python implementation and output**:



```python
def max_benefit_with_minimum_items(items, budget):
    from collections import defaultdict
    groups = defaultdict(list)
    for index, (pi, ti, gi) in enumerate(items):
        benefit = pi - ti
        groups[gi].append((benefit, pi, ti, index))  # Include item index for identification

    groups_list = list(groups.values())

    # DP table for benefits and item counts: (max_benefit, min_items)
    dp = [[(0, 0) for _ in range(budget + 1)] for _ in range(len(groups_list) + 1)]

    for i in range(1, len(groups_list) + 1):
        for j in range(1, budget + 1):
            # Inherit the previous state
            dp[i][j] = dp[i-1][j]
            for benefit, pi, ti, idx in groups_list[i-1]:
                if ti <= j:
                    prev_benefit, prev_items = dp[i-1][j-ti]
                    new_benefit = prev_benefit + benefit
                    new_items = prev_items + 1
                    # Update if a better benefit is found, or same benefit with fewer items
                    if new_benefit > dp[i][j][0] or (new_benefit == dp[i][j][0] and new_items < dp[i][j][1]):
                        dp[i][j] = (new_benefit, new_items)

    max_benefit, min_items = dp[len(groups_list)][budget]
    return max_benefit, min_items

# Example test case
items = [
    (100, 50, 1),  # Item 0, Group 1, High benefit but high cost
    (80, 30, 1),   # Item 1, Group 1, Slightly lower benefit but much lower cost
    (90, 40, 2),   # Item 2, Group 2, Balance between benefit and cost
    (200, 100, 3), # Item 3, Standalone group, High cost and benefit
    (150, 75, 4)   # Item 4, Standalone group, Moderate cost and benefit
]
budget = 150

# Calculate the highest total benefit and the fewest number of items
highest_benefit, fewest_items = max_benefit_with_minimum_items(items, budget)
print(f"Highest Total Benefit: {highest_benefit}")
print(f"Fewest Number of Items: {fewest_items}")
```

```
Highest Total Benefit: 175
Fewest Number of Items: 3
```

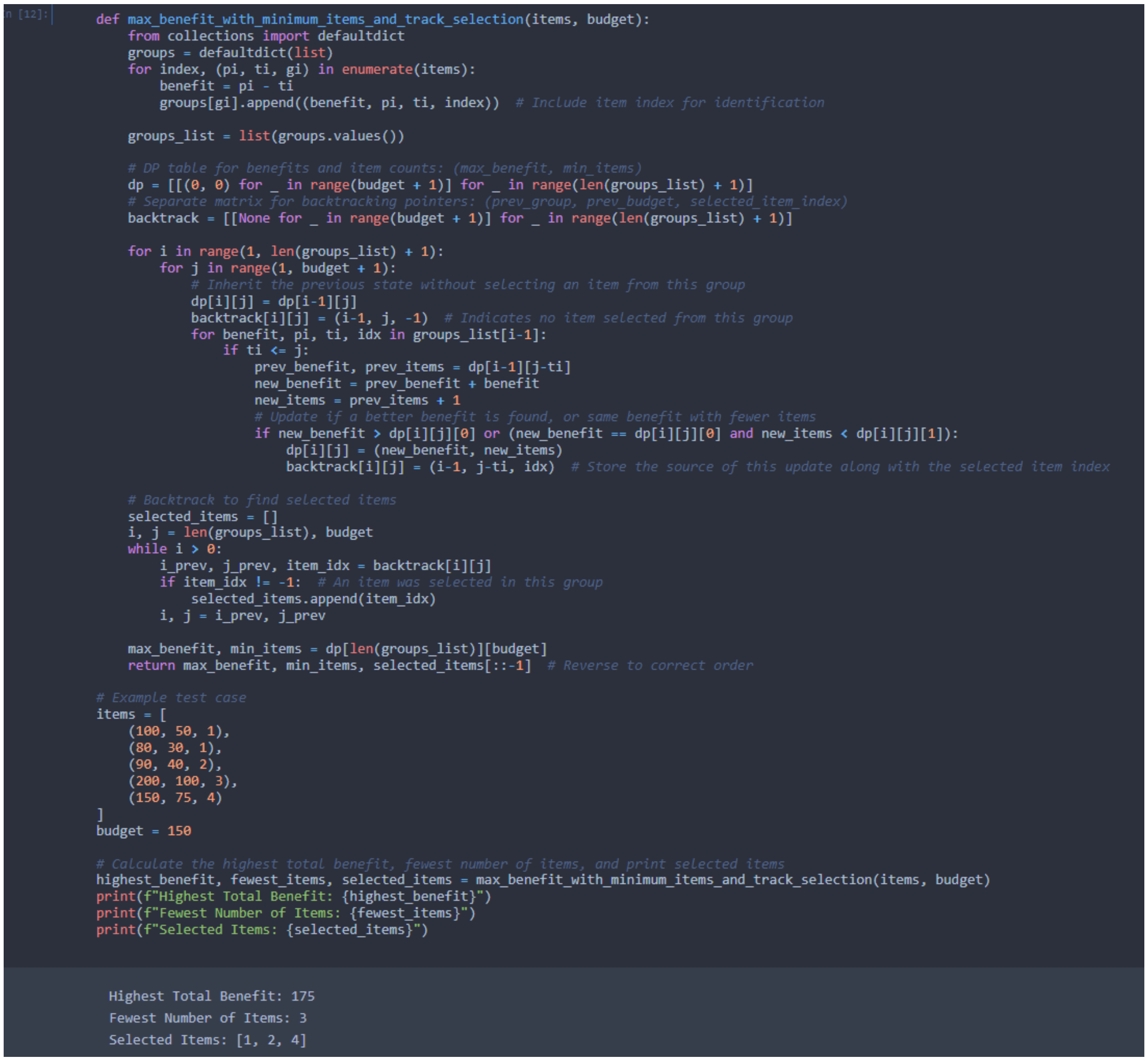
2.4 List of items to buy

To print the list of items to buy, we have to employ a backtracking algorithm on our dp matrix. We introduce a 'backtrack' matrix of tuples containing 3 elements i.e the previous number of groups, the previous budget and the previous item index from which it is getting updated, specifically the previous state (i-1, j-ti) and the item index (idx) that led to the current state.

- If no item is selected from a group, then we dont update the index of the backtrack matrix cell and just move to the next group.

- If any item is selected, we store the source of this update i.e the previous group i-1, the new smaller budget j-i_k and the index of the element idx.

After we have populated the backtrack matrix, we need to start from the bottom right cell (We have to start from the cell where we end and we go back and see which cells we covered before we reach this cell)and traverse till the point where no groups are selected.
As we encounter a cell where the index of an element is present(this means at the given budget and first i groups, an item from this group is selected). We record that index of the selected item by iterating through the items in the listin a list 'selected items'. We output the reversed selected items list since we are appending the indexes to the list in the reverse order we selected them in. The following python code demonstrates this implementation :



```
n [12]:
def max_benefit_with_minimum_items_and_track_selection(items, budget):
    from collections import defaultdict
    groups = defaultdict(list)
    for index, (pi, ti, gi) in enumerate(items):
        benefit = pi - ti
        groups[gi].append((benefit, pi, ti, index))  # Include item index for identification

    groups_list = list(groups.values())

    # DP table for benefits and item counts: (max_benefit, min_items)
    dp = [[(0, 0) for _ in range(budget + 1)] for _ in range(len(groups_list) + 1)]
    # Separate matrix for backtracking pointers: (prev_group, prev_budget, selected_item_index)
    backtrack = [[None for _ in range(budget + 1)] for _ in range(len(groups_list) + 1)]

    for i in range(1, len(groups_list) + 1):
        for j in range(1, budget + 1):
            # Inherit the previous state without selecting an item from this group
            dp[i][j] = dp[i-1][j]
            backtrack[i][j] = (i-1, j, -1)  # Indicates no item selected from this group
            for benefit, pi, ti, idx in groups_list[i-1]:
                if ti <= j:
                    prev_benefit, prev_items = dp[i-1][j-ti]
                    new_benefit = prev_benefit + benefit
                    new_items = prev_items + 1
                    # Update if a better benefit is found, or same benefit with fewer items
                    if new_benefit > dp[i][j][0] or (new_benefit == dp[i][j][0] and new_items < dp[i][j][1]):
                        dp[i][j] = (new_benefit, new_items)
                        backtrack[i][j] = (i-1, j-ti, idx)  # Store the source of this update along with the selected item index

    # Backtrack to find selected items
    selected_items = []
    i, j = len(groups_list), budget
    while i > 0:
        i_prev, j_prev, item_idx = backtrack[i][j]
        if item_idx != -1:  # An item was selected in this group
            selected_items.append(item_idx)
        i, j = i_prev, j_prev

    max_benefit, min_items = dp[len(groups_list)][budget]
    return max_benefit, min_items, selected_items[::-1]  # Reverse to correct order

# Example test case
items = [
    (100, 50, 1),
    (80, 30, 1),
    (90, 40, 2),
    (200, 100, 3),
    (150, 75, 4)
]
budget = 150

# Calculate the highest total benefit, fewest number of items, and print selected items
highest_benefit, fewest_items, selected_items = max_benefit_with_minimum_items_and_track_selection(items, budget)
print(f"Highest Total Benefit: {highest_benefit}")
print(f"Fewest Number of Items: {fewest_items}")
print(f"Selected Items: {selected_items}")
```

```
Highest Total Benefit: 175
Fewest Number of Items: 3
Selected Items: [1, 2, 4]
```

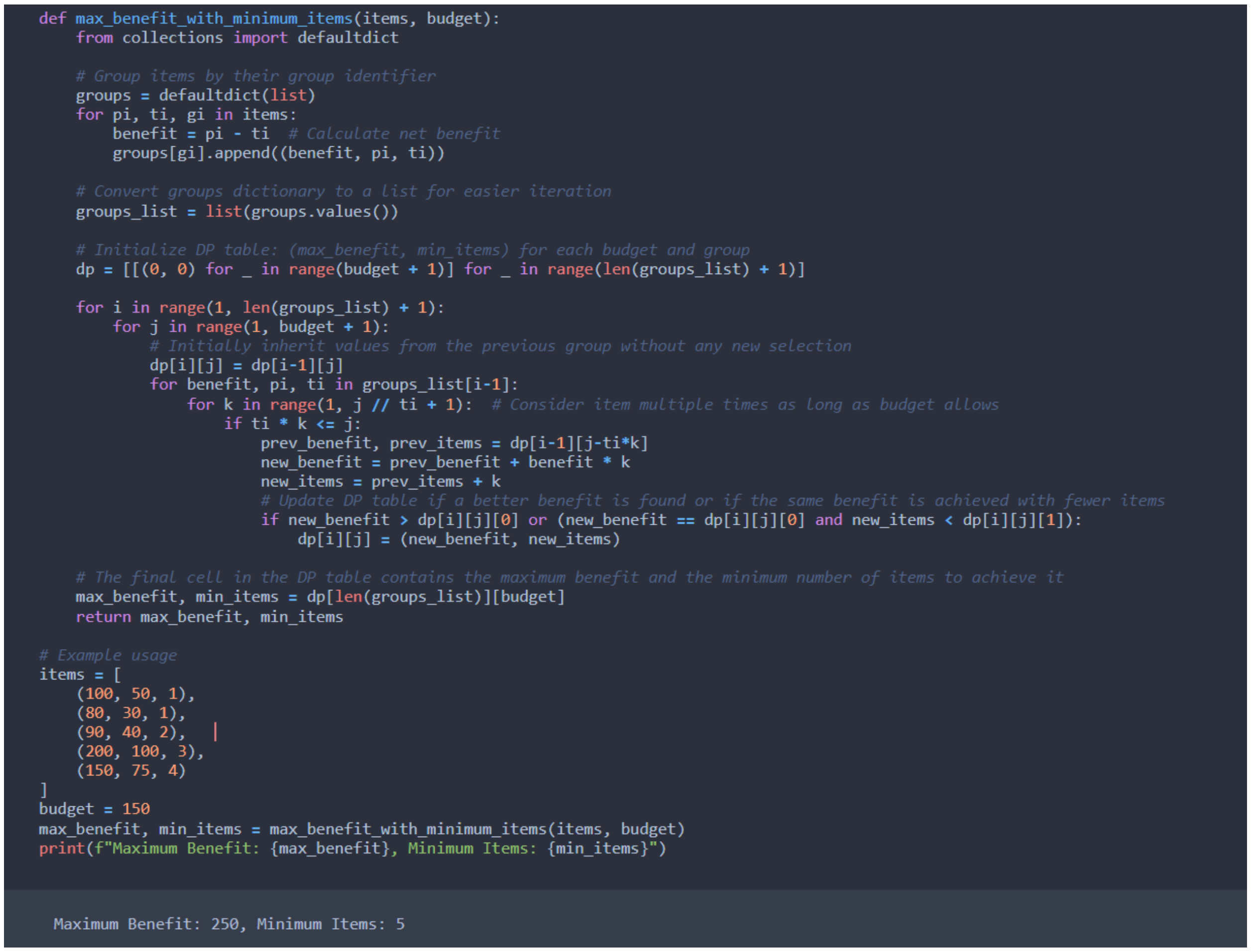
With the same given input, this code gives the output we expected, Highest benefit = 175. Minimum number of items = 3 and the items are 2,3 and 5.

2.5 Unbounded number of items

When we can pick an unbounded number of items from each group, we have to modify the algorithm in such a way that after picking an item from a group, yihan can keep picking the same item as many times she wants. For every value of budget from 1 to W, we check for each group and each item in each group if they can be picked or not and store it in the dp matrix. We keep picking this item till it exhausts the budget and check update the matrix every time we check for another time we consider that element. The state, base cases and the answer remains the same in this case. The dp matrix structure is the same but the way its being filled is different. The recurrence for our solution looks like this: **Recurrence** : There can be two cases:

1. The group is not chosen : s[i-1][j]
2. The group is chosen. Now we have to check for each element and for the number of times each element can be considered in the given budget and update the matrix : max(s[i-1][j-$i_k n$] + $p_k n - i_k n$) where $p_k n$ and $i_k n$ is the actual cost and the discounted cost for picking an item k in the group i, n times where n ranges from 1 to budget//i_k

s[i][j] = max(1,2)
Answer : The maximum benefit by picking all the groups and at given budget W i.e. s[len(groups)][W].
To fill up the dp matrix, we first loop through every budget value from 1 to W i.e W times. For each budget, we check all groups and all items in each group i.e g*k times where g is the number of groups and k is the mean number of people in each group. But g*k = n (n = number of items). For all items, we check the number of times it can be considered in budget j (j//p_k). For worst case, the price is 1 and it is considered j times. Therefore, while checking each budget value j, for each item, we will be checking j times. Therefore, the number of checks for each item = $\sum_{j=1}^{W} j = (W^2 + W)/2 = O(W^2)$. Therefore, the total time complexity of our algorithm for all items = $O(nW^2)$
The following python code is the implementation of this algorithm. We see that for the same given input, we get a higher maximum value, now that unbounded condition is there.



```
def max_benefit_with_minimum_items(items, budget):
    from collections import defaultdict

    # Group items by their group identifier
    groups = defaultdict(list)
    for pi, ti, gi in items:
        benefit = pi - ti  # Calculate net benefit
        groups[gi].append((benefit, pi, ti))

    # Convert groups dictionary to a list for easier iteration
    groups_list = list(groups.values())

    # Initialize DP table: (max_benefit, min_items) for each budget and group
    dp = [[(0, 0) for _ in range(budget + 1)] for _ in range(len(groups_list) + 1)]

    for i in range(1, len(groups_list) + 1):
        for j in range(1, budget + 1):
            # Initially inherit values from the previous group without any new selection
            dp[i][j] = dp[i-1][j]
            for benefit, pi, ti in groups_list[i-1]:
                for k in range(1, j // ti + 1):  # Consider item multiple times as long as budget allows
                    if ti * k <= j:
                        prev_benefit, prev_items = dp[i-1][j-ti*k]
                        new_benefit = prev_benefit + benefit * k
                        new_items = prev_items + k
                        # Update DP table if a better benefit is found or if the same benefit is achieved with fewer items
                        if new_benefit > dp[i][j][0] or (new_benefit == dp[i][j][0] and new_items < dp[i][j][1]):
                            dp[i][j] = (new_benefit, new_items)

    # The final cell in the DP table contains the maximum benefit and the minimum number of items to achieve it
    max_benefit, min_items = dp[len(groups_list)][budget]
    return max_benefit, min_items

# Example usage
items = [
    (100, 50, 1),
    (80, 30, 1),
    (90, 40, 2),
    (200, 100, 3),
    (150, 75, 4)
]
budget = 150
max_benefit, min_items = max_benefit_with_minimum_items(items, budget)
print(f"Maximum Benefit: {max_benefit}, Minimum Items: {min_items}")
```

```
Maximum Benefit: 250, Minimum Items: 5
```

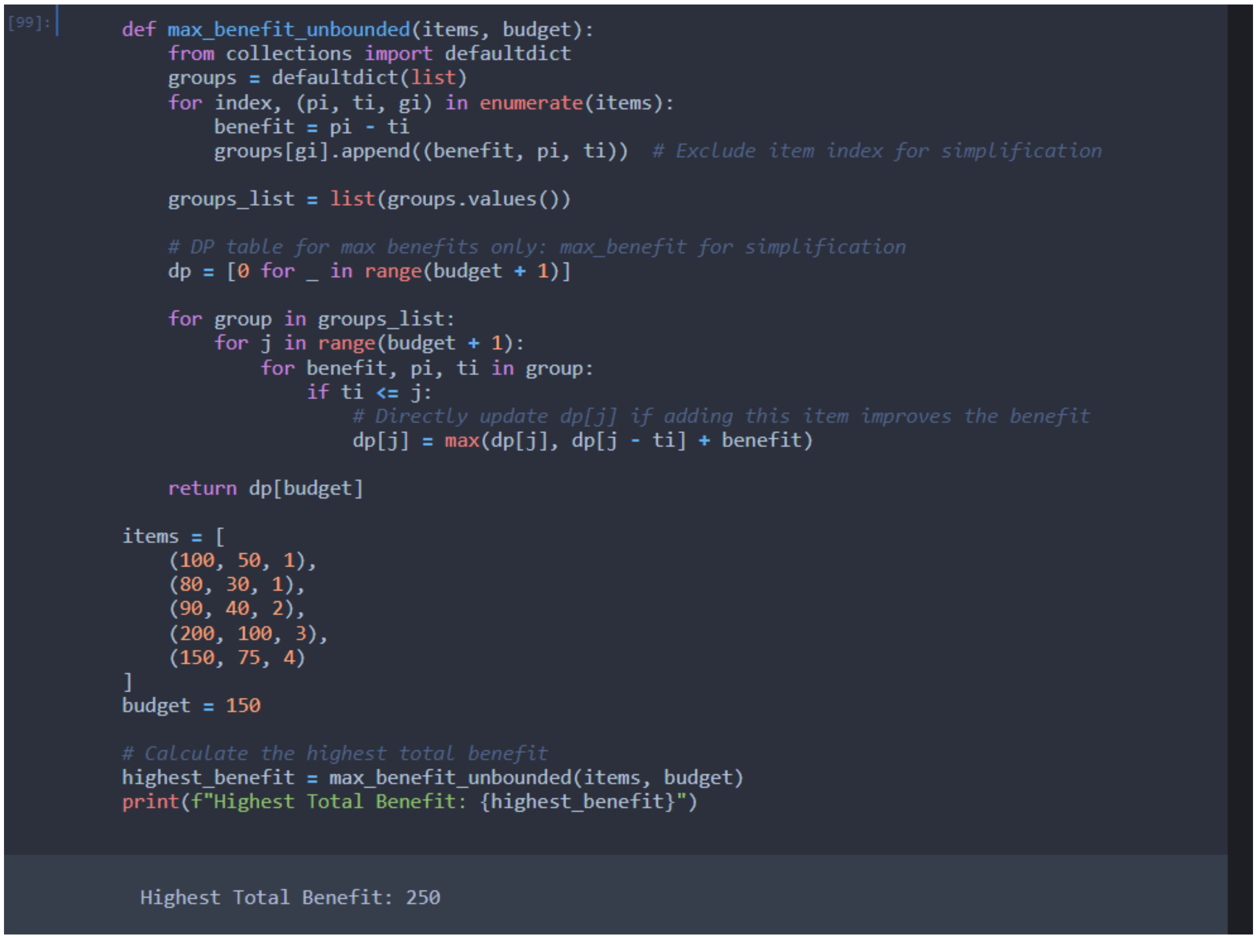
We get the maximum benefit = 250 by picking item 1 5 times.

2.6 Optimize to O(nW) time

To change the complexity of the code to O(nW), we treat the problem similarly to the "unbounded knapsack problem" without explicitly iterating over each possible quantity of an item. Instead, we can dynamically update the DP table for each budget level by considering the addition of an item to all possible previous states directly.
Direct State Updates: For each item in each group, the DP table is updated for every budget level directly, considering the addition of that item. This effectively incorporates the possibility of selecting multiple quantities of the same item without explicitly iterating over quantities.
Unlike the previous approach, we don't need a separate loop for multiples of an item since we're effectively considering all multiples by updating dp[j] in a single pass. **State** : s[i] = maximum value we can get at budget i
Recurrence : For every group we consider, we check from every budget value from W to 1 the maximum value we can get by considering an item as many times as possible. There can be two cases:

1. Dont consider any item from group i: s[j]
2. Consider any item k in group i, n times where n=j//p_k : s[j-$t_k n$] + benefit

s[j] = max(1,2)
Base : s[0] = 0
Answer : s[budget] **Time complexity** : For each group, we check W times, each item in the group(let k be the number of items in each group on average). Therefore, for each group, we check k*W times which is O(kW). Let there be g groups. Therefore, total time complexity of our problem = O(gkW). But g*k=n (n=total number of items).
Time complexity = O(nW). The below python implementation runs in O(nW) time and gives the same output for the given example.



```
[99]:
def max_benefit_unbounded(items, budget):
    from collections import defaultdict
    groups = defaultdict(list)
    for index, (pi, ti, gi) in enumerate(items):
        benefit = pi - ti
        groups[gi].append((benefit, pi, ti))  # Exclude item index for simplification

    groups_list = list(groups.values())

    # DP table for max benefits only: max_benefit for simplification
    dp = [0 for _ in range(budget + 1)]

    for group in groups_list:
        for j in range(budget + 1):
            for benefit, pi, ti in group:
                if ti <= j:
                    # Directly update dp[j] if adding this item improves the benefit
                    dp[j] = max(dp[j], dp[j - ti] + benefit)

    return dp[budget]

items = [
    (100, 50, 1),
    (80, 30, 1),
    (90, 40, 2),
    (200, 100, 3),
    (150, 75, 4)
]
budget = 150

# Calculate the highest total benefit
highest_benefit = max_benefit_unbounded(items, budget)
print(f"Highest Total Benefit: {highest_benefit}")
```

```
Highest Total Benefit: 250
```

3 Morse Code

3.1 Morse code for UCR

The morse code for UCR is '..- -.-..-.'
Let the morse code representation of UCR be X = '..- -.-..-.'. Any other interpretations of X are:

- 'IQUE' = 9 strokes
- 'EACR' = 9 strokes
- 'IQF' = 8 strokes

IQF has the fewest strokes i.e 8.

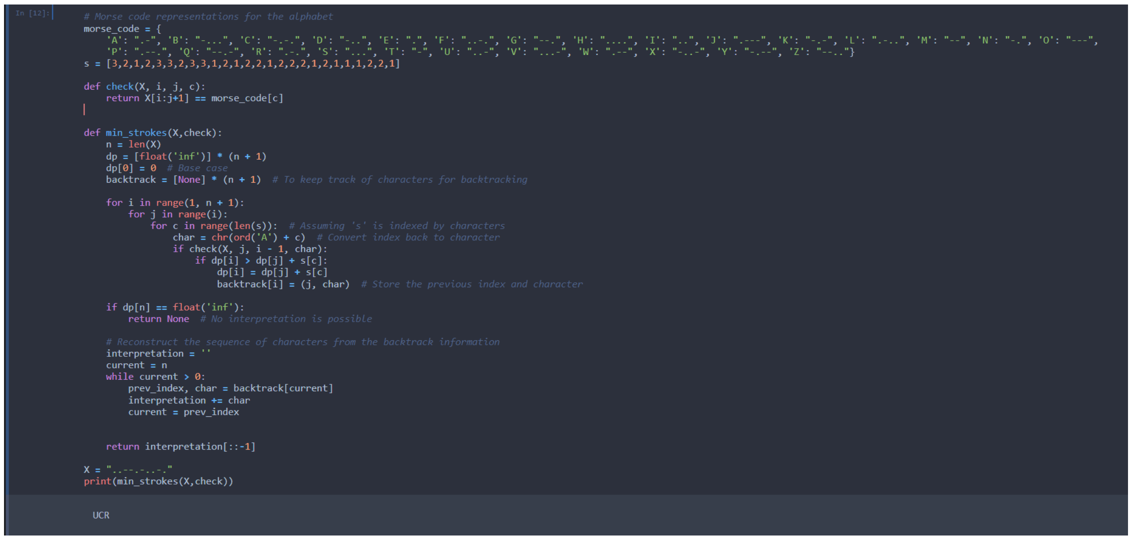
3.2 Find interpretation with fewest strokes

To solve this problem, we will first find an algorithm which finds the minimum number of strokes needed to decode a morse code. We will solve this using dynamic programming. Once we have populated the dp matrix, we will find the characters chosen to reach this solution by backtracking through the dp matrix.
To find the minimum number of strokes needed to decode a morse code, we construct our dp problem in the following way:
State : s[i] = The fewest strokes we can get considering the first i elements of X i.e X[:i]
Recurrence: While considering the first i characters of X, we will consider each substring starting from X[0:i] to X[i-1:i] i.e for every $0 <= j < i$, we will check for each substring X[j+1:i] if there exists a corresponding character to the selected substring according to the given function check(X,j,i,c). There can be two cases:

1. s[i] : No character is possible from the substring X[j+1:i]. Minimum value remains same.
2. s[j] + strokes(c) : If there is a character that can be formed from the substring X[j+1:i], then we check for the next substring X[0:j] and add the number of strokes to form a character from the substring X[j+1:i]. s[j] represents the minimum number of strokes to compute a character of the substring X[0:j].

s[i] = min(1,2). Since we have to get the minimum number of strokes, we have to take the min of the two possible cases.
Base case : s[0] = 0 since no character can be formed from the substring X[:0] which is empty. We populate the dp matrix according to the given rules. For every i = 1 to n-1 and for every corresponding j from 0 to i, we check for all characters(c), if any of them can be constructed from the substring X[j:i-1]. We update the s[i] value as the min of s[i] and s[j]+strokes(c). The minimum number of strokes required to interpret a given Morse code is s[n] where n is the length of the morse code.
After populating the dp matrix, we have to find out the characters which have been used to find the interpretation of the given Morse code X. for that, we have to backtrack the dp matrix from the last element s[n] to the beginning.
We create an array 'backtrack' of the same length as X. While populating the dp matrix, if we come across a substring X[j+1:i]such that a character can be constructed from it and if it updates the dp matrix to a new minimum value, that means we consider the character.
Therefore, to record that value, we record the previous index j and the character c in our backtrack array. Hence, at backtrack[i], it will store the previous index of X i.e 'j' till which(X[:j]) we have to find the validity of a character and it stores the character c, which is interpreted at minimum cost from X[j+1:i].
The backtrack array will have None values in all cells excepts the ones which have values stored in them. If an interpretation can be reach from a given morse code X, then the cell backtrack[n] will contain the character which has been formed last and the index of the previous cell where the second last character is stored and so on. Thus, to output the required interpretation, we print the characters from backtrack where each corresponding cell from backtrack[n] points to, in a reverse order.

Python Implementation : We demonstrate the correctness of our code with the given X = '..- -.-..-.' and we find that the interpretation with the minimum number of strokes is 'UCR'.



```
In [12]:
# Morse code representations for the alphabet
morse_code = {
    'A': ".-", 'B': "-...", 'C': "-.-.", 'D': "-..", 'E': ".", 'F': "..-.", 'G': "--.", 'H': "....", 'I': "..", 'J': ".---", 'K': "-.-", 'L': ".-..", 'M': "--", 'N': "-.", 'O': "---",
    'P': ".--.", 'Q': "--.-", 'R': ".-.", 'S': "...", 'T': "-", 'U': "..-", 'V': "...-", 'W': ".--", 'X': "-..-", 'Y': "-.--", 'Z': "--.."}
s = [3,2,1,2,3,3,2,3,3,1,2,1,2,2,1,2,2,2,1,2,1,1,1,2,2,1]

def check(X, i, j, c):
    return X[i:j+1] == morse_code[c]


def min_strokes(X,check):
    n = len(X)
    dp = [float('inf')] * (n + 1)
    dp[0] = 0  # Base case
    backtrack = [None] * (n + 1)  # To keep track of characters for backtracking

    for i in range(1, n + 1):
        for j in range(i):
            for c in range(len(s)):  # Assuming 's' is indexed by characters
                char = chr(ord('A') + c)  # Convert index back to character
                if check(X, j, i - 1, char):
                    if dp[i] > dp[j] + s[c]:
                        dp[i] = dp[j] + s[c]
                        backtrack[i] = (j, char)  # Store the previous index and character

    if dp[n] == float('inf'):
        return None  # No interpretation is possible

    # Reconstruct the sequence of characters from the backtrack information
    interpretation = ''
    current = n
    while current > 0:
        prev_index, char = backtrack[current]
        interpretation += char
        current = prev_index


    return interpretation[::-1]

X = "..--.-..-."
print(min_strokes(X,check))
```

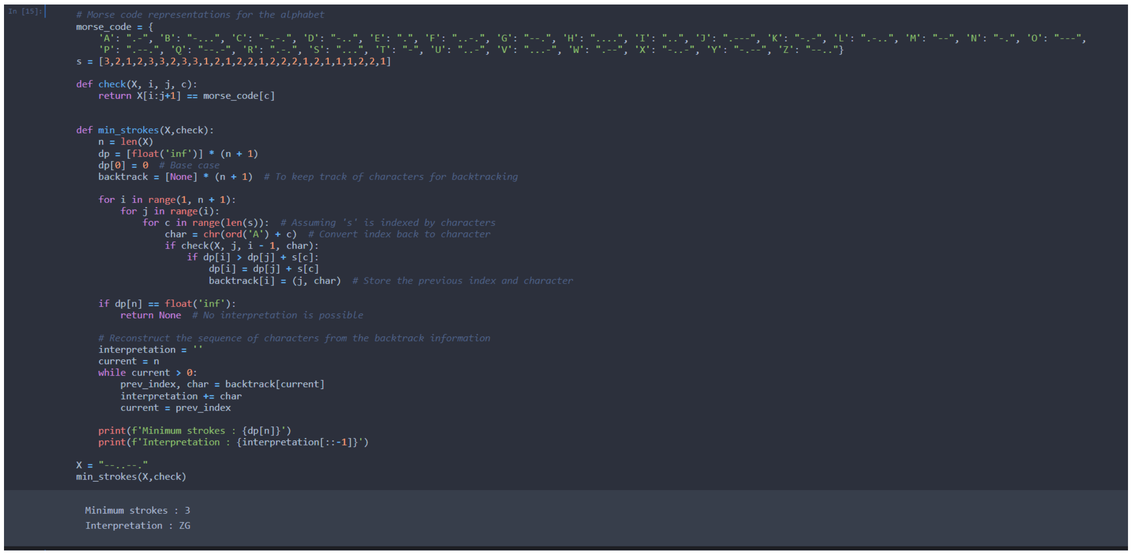
```
UCR
```

3.3 Time complexity

In our algorithm, the time complexity to check for one character for every substring X[j+1:i] is $O(n^2)$ since we are checking k times where k = 1 (X[0:1]) + 2(X[0:2],X[1:2]) + 3 + ... + n
Therefore $k = n(n+1)/2 = (n^2+n)/2$. To check for all characters, we check $O(mn^2)$ times where m = 26. Therefore the time complexity of our algorithm is $O(n^2)$.

3.4 Compute given morse code

Using our algorithm, the smallest number of strokes we can get by interpreting string X = '- -..- -.' is 3. The resulting interpretation is 'ZG' where Z(1) = '- -. .' and G(2) ='- - .' . The number of strokes is 1+2 =3. This can be shown by the output from our python code as follows:



```
In [15]:
# Morse code representations for the alphabet
morse_code = {
    'A': ".-", 'B': "-...", 'C': "-.-.", 'D': "-..", 'E': ".", 'F': "..-.", 'G': "--.", 'H': "....", 'I': "..", 'J': ".---", 'K': "-.-", 'L': ".-..", 'M': "--", 'N': "-.", 'O': "---",
    'P': ".--.", 'Q': "--.-", 'R': ".-.", 'S': "...", 'T': "-", 'U': "..-", 'V': "...-", 'W': ".--", 'X': "-..-", 'Y': "-.--", 'Z': "--.."}
s = [3,2,1,2,3,3,2,3,3,1,2,1,2,2,1,2,2,2,1,2,1,1,1,2,2,1]

def check(X, i, j, c):
    return X[i:j+1] == morse_code[c]


def min_strokes(X,check):
    n = len(X)
    dp = [float('inf')] * (n + 1)
    dp[0] = 0  # Base case
    backtrack = [None] * (n + 1)  # To keep track of characters for backtracking

    for i in range(1, n + 1):
        for j in range(i):
            for c in range(len(s)):  # Assuming 's' is indexed by characters
                char = chr(ord('A') + c)  # Convert index back to character
                if check(X, j, i - 1, char):
                    if dp[i] > dp[j] + s[c]:
                        dp[i] = dp[j] + s[c]
                        backtrack[i] = (j, char)  # Store the previous index and character

    if dp[n] == float('inf'):
        return None  # No interpretation is possible

    # Reconstruct the sequence of characters from the backtrack information
    interpretation = ''
    current = n
    while current > 0:
        prev_index, char = backtrack[current]
        interpretation += char
        current = prev_index

    print(f'Minimum strokes : {dp[n]}')
    print(f'Interpretation : {interpretation[::-1]}')

X = "--..--."
min_strokes(X,check)
```

```
Minimum strokes : 3
Interpretation : ZG
```