Dynamic Programming Quals Problems

Sijian Tan

1 Trick-or-treating (HW4)

Your nephew is going trick-or-treating to n houses along a street where each house has a limited number of candies. For any house i, if the residents see a trick-or-treater picking candies from any neighbouring house in the radius of two houses, they will not open the door for that person. For example, consider a row of houses A,B,C,D,E,F. If you take candies from house D, you cannot take candies from houses B,C and E,F.

Now given an array T , where $T(i)$ represents the number of candies you can get from house i, please design a dynamic programming algorithm (DP) to help your nephew determine the maximum number of treats he can get.

1. Define your subproblem, and write the recurrence relation including the base case.
2. Write the bottom-up pseudocode.
3. Analyze time and space complexity.

1.1 Strategy

The key of this problem is that at each house, we either visit the house or move to the next one depending which way we can get max number of treats. If we choose to visit, then we need get rid of the neighbours of this house. If not, we could just use the max number from last house.

1.2 Recurrence Relation

Subproblem: Define $OPT(i)$ as the max total number of candies we can get from house 0 to i.

Recurrence relation could be expressed as:

$$OPT(i) = \max\{OPT(i-1), OPT(i-2), OPT(i-3) + T(i)\} \quad (1)$$

For the first three houses, we have several base cases:

$$OPT(0) = T(0) \quad (2)$$

$$OPT(1) = \max\{T(0), T(1)\} \quad (3)$$

$$OPT(2) = \max\{T(0), T(1), T(2)\} \quad (4)$$

1.3 Bottom-up

Algorithm 1 Best Candies (Bottom-up)

```
function BestCandies(T)
    n ← length(T)

    Create an array OPT of size n
    OPT[0] ← T[0]
    OPT[1] ← max(T[0], T[1])
    OPT[2] ← max(T[0], T[1], T[2])

    for i ← 3 to n − 1 do
        OPT[i] ← max(OPT[i − 1], OPT[i − 2], T[i] + OPT[i − 3])
    end for
    return OPT[n − 1]
end function
```

1.4 Top-down

Algorithm 2 Best Candies (Top-down)

```
function BestCandies(T)
    n ← length(T)
    OPT = [−float('inf')] ∗ n

    function compute_opt(T, i, OPT)
        if i < 0 then
            return 0
        end if
        if OPT[i]! = −float('inf') then
            return OPT[i]
        end if
        if i == 0 then
            OPT[i] = T[0]
        else if i == 1 then
            OPT[i] = max{T[0], T[1]}
        else if i == 2 then
            OPT[i] = max{T[0], T[1], T[2]}
        else
            OPT[i] = max{OPT[i − 1], OPT[i − 2], T[i] + OPT[i − 3]}
        end if
        return OPT[i]
    end function

    return compute_opt[T, n − 1, OPT]
end function
```

1.5 Backtracing

Algorithm 3 Best Candies Backtracing

```
1: function BACKTRACING(n, T, OPT)
2:     i = n − 1
3:
4:     while i ≥ 0 do
5:
6:         if i = 0 then
7:             print 0
8:             return
9:         end if
10:
11:        if i = 1 then
12:            if max(OPT[0], OPT[1]) = OPT[0] then
13:                print 0
14:            else
15:                print 1
16:            end if
17:            return
18:        end if
19:
20:        if i = 2 then
21:            if max(OPT[0], OPT[1], OPT[2]) = OPT[0] then
22:                print 0
23:            else if max(OPT[0], OPT[1], OPT[2]) = OPT[1] then
24:                print 1
25:            else
26:                print 2
27:            end if
28:            return
29:        end if
30:
31:        if T[i] + OPT[i − 3] = max then
32:            print i
33:            i = i − 3
34:        else if OPT[i − 2] = max then
35:            i = i − 2
36:        else
37:            i = i − 1
38:        end if
39:
40:    end while
41:
42: end function
```

1.6 Complexity Analysis

For the time complexity, we need to fill the OPT array of size n once, and each $OPT[i]$ computation is done in constant time, so it is $O(n)$.

For the space complexity, we use 1-D array OPT of size n to store the results at each step, so it is also $O(n)$.

2 Water Supply Schedule (Spring 2024)

Suppose you are choosing a water supplier for your house. For each of the next n weeks, you will need s_i tons of water, which have to be supplied by a water supplier. Each week's water supply can be provided by only one of two water companies, A or B.

- Company A charges a fixed rate r per ton (so it costs $r \times s_i$ to provide a week's water demand s_i)
- Company B makes contracts for a fixed cost c per week, no matter how many tons are ordered. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A schedule, for your house, is a choice of water supply company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The cost of the schedule is the total amount paid to companies A and B, according to the description above. Design a dynamic programming algorithm that takes a sequence of water values $s_1, s_2, \cdots, s_n$ and returns a schedule of minimum cost.

- Give the recurrence relations (do not forget the base cases).
- Give the pseudocode of the top-down implementation.
- Analyze time and space complexity.

2.1 Strategy

The key of this problem it that at each week, we either choose company A or B. For choosing A, it is easy, simply just add the cost to the last week cost, but choosing B is complex. The cases will be different with this week's position in a four-week cycle.

2.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the minimum cost we could reach at week i.

The **recurrence relation** will be:

$$OPT(i) = \min \begin{cases} OPT[i-1] + r \cdot s_i \\ OPT[i-1] + c \\ OPT[i-2] + 2c \\ OPT[i-3] + 3c \\ OPT[i-4] + 4c \end{cases} \tag{5}$$

And the base cases will be:

$$OPT[0] = 0 \tag{6}$$

$$OPT[1] = \min\{rs_1, c\} \tag{7}$$

$$OPT[2] = \min\{OPT[1] + rs_2, OPT[1] + c, 2c\} \tag{8}$$

$$OPT[3] = \min\{OPT[2] + rs_3, OPT[2] + c, OPT[1] + 2c, 3c\} \tag{9}$$

2.3 Complexity Analysis

Both time and space complexities are $O(n)$.

3 Climb Step (Fall 2023)

You are given an integer array `cost`, where `cost[i]` is the cost of the i-th step on a staircase. Once you pay the cost, you can either climb one step or two steps. Design a dynamic programming algorithm which finds (i) the minimum cost to reach the top floor and (ii) the sequence of steps to achieve that minimum.

1. **Example 1**: if input is $cost = [\underline{10}, \underline{15}, 20]$, then the output will be:
   - Minimum cost = 25
   - sequence of steps = [0, 1]
2. **Example 2**: if input is $cost = [\underline{1}, 100, \underline{1}, 1, \underline{1}, 100, \underline{1}, \underline{1}, 100, \underline{1}]$, then the output will be:
   - Minimum cost = 6
   - sequence of steps = [0, 2, 4, 6, 7, 9]

You are required to provide the recurrence relation and write a pseudocode.

3.1 Strategy

The key of this problem is that at each step i, we either finish this step by taking one step, costing $cost[i]$, or finish this step by $i-1$ step with two steps, costing $cost[i-1]$.

3.2 Recurrence Relation

Subproblem: Define $OPT(i)$ as the minimum cost to reach step i (not finish this step, just arrive this step).

Recurrence relation could be expressed as:

$$OPT(i) = \min\{OPT(i-1) + cost[i], OPT(i-2) + cost[i-1]\} \quad (10)$$

The basic case will be:

$$OPT[0] = 0 \quad (11)$$

3.3 Complexity Analysis

Both the time and space complexities are $O(n)$.

4 States and Electoral Votes (2D, Spring 2023)

This problem is to determine the set of states with the smallest total population that can provide the votes to win the election.

Formally, the problem is: We are given a list of states $\{1, \ldots, n\}$ where each state i has population p_i, and v_i, which is the number of electoral votes for state i. All electoral votes of a state go to a single candidate. The overall winning candidate is the one who receives at least V electoral votes, where

$$V = \left(\sum_i v_i\right)/2 + 1.$$

Our goal is to find a set of states S that minimizes the value of

$$\sum_{i \in S} p_i$$

subject to the constraint that

$$\sum_{i \in S} v_i \geq V.$$

Please design a dynamic programming algorithm for this problem. Define the subproblem, give the recurrence relations, and analyze the time and space complexity of your algorithm. Remember to include steps to output the optimal set of states (you do not need to output all the optimal solutions if there are more than one).

4.1 Strategy

In this problem, there are two constraints. The first one is that we want to minimize the number of population of total selected states, this could be done by the normal 1D recurrence relation. The other one is that we need to make sure the total votes pass V. For this kind of problem, **it is better to start from V and keep reducing the amount.** The recurrence relation is either we choose a state i or not.

4.2 Recurrence Relation

Subproblem: Define $OPT[i][j]$ to be the minimum population required to get j votes.

Recurrence relation could be expressed as:

$$OPT[i][j] = \min\{OPT[i-1][j], OPT[i-1][j-v_i] + p_i\} \tag{12}$$

For the base case, we initialize $OPT[i][0] = 0$ for all i because zero electoral votes require zero population, and all other $OPT[i][j] = \infty$, because initially they are unreachable. Notice that in this problem, we assume there is a solution just satisfy the votes requirement.

4.3 Top-down, Backtracing

Algorithm 4 States and Votes (Top-down)

```
function MIN_POPULATION_TO_WIN(s, p, v)
    n ← length(p)
    V = sum(v)//2 + 1
    Initialize a n × V memoization table (memo), with all elements as None
    function OPT(i,j)
        if j = 0 then
            return 0          ▷ Base case: if j = 0, no votes needed, population is 0
        end if
        if i = 0 then
            return ∞ ▷ If we have considered all states and still need votes, return ∞ (impossible)
        end if

        if memo[i][j] is not None then
            return memo[i][j]      ▷ If the value is already computed, just return it
        end if

        option1 = OPT(i − 1, j)                                  ▷ Do not include state i
        if j ≥ v[i] then
            option2 = OPT(i − 1, j − v[i]) + p[i]                ▷ Include state i
        else
            option2 = float('inf')
        end if

        memo[i][j] = min(option1, option2)
        return memo[i][j]
    end function

    min_population = OPT(n, V)

    S = []                                   ▷ Initialize a optimal set of states
    i, j = n, V
    while j > 0 and i > 0 do
        if memo[i][j] == memo[i − 1][j]  then
            i = i − 1
        else
            S.append(s[i])
            j = j − v[i]
            i = i − 1
        end if
    end while
    return min_population, S
end function
```

4.4 Bottom-up, Backtracing

Algorithm 5 States and Votes (Bottom-up)

```
function MIN_POPULATION_TO_WIN(s, p, v)
    n ← length(p)
    V = sum(v)//2 + 1
    Initialize the (n × V) DP table (OPT)

    for i ← 0 to n do
        for j ← 0 to n do
            if j == 0 then
                OPT[i][j] = 0
            else
                OPT[i][j] = ∞
            end if
        end for
    end for

    for i ← 0 to n do
        for j ← 0 to V do
            if  j ≥ v[i] then
                OPT[i][j] = min{OPT[i − 1][j], OPT[i − 1][j − v[i]] + p[i]}
            else
                OPT[i][j] = OPT[i − 1][j]
            end if
        end for
    end for

    min_population = OPT[n][V]

    S = []                                  ▷ Initialize a optimal set of states
    i, j = n, V

    while j > 0 and i > 0 do
        if OPT[i][j] == OPT[i − 1][j]  then
            i = i − 1
        else
            S.append(s[i])
            j = j − v[i]
            i = i − 1
        end if
    end while

    return min_population, S
end function
```

4.5 Complexity Analysis

Both the time complexity and space complexity are $O(nV)$.

5 Merge Two Sequences (2D, Spring 2023)

Let $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be two genomic sequences represented by strings of letters and $C[i, j]$ be the cost function defined on pairs of letters, one from X and one from Y.

Our task is to merge these two sequences to create a new genomic sequence Z and we need to find the cheapest merge of X and Y, while maintaining the order of letters from both X and Y. Therefore, for instance, if $X = \{a, b, a, c\}$ and $Y = \{d, a, e, b\}$, then $Z = \{a, d, a, b, a, e, c, b\}$ and $Z = \{a, d, a, e, b, b, a, c\}$ are valid merges, but $Z = \{b, a, e, c, d, a, b\}$ is not because e from the second sequence is used before d and a.

Total cost of the merge is the sum of the merging costs of the adjacent letters from different sequences. So if Z includes x_i and x_{i+1} as consecutive letters, there is no cost, but if Z has $x_s y_t$ as consecutive letters, then there is a cost $C[s, t]$ and it should be added to the total cost of the merge. *Note: You can assume that the cost function is symmetric.*

Please design a dynamic programming algorithm to find the minimum cost of merging X and Y. Please define the subproblem(s) and give the recurrence relations. Analyze the time and space complexity of your algorithm. Backtracking step and pseudocode are not required.

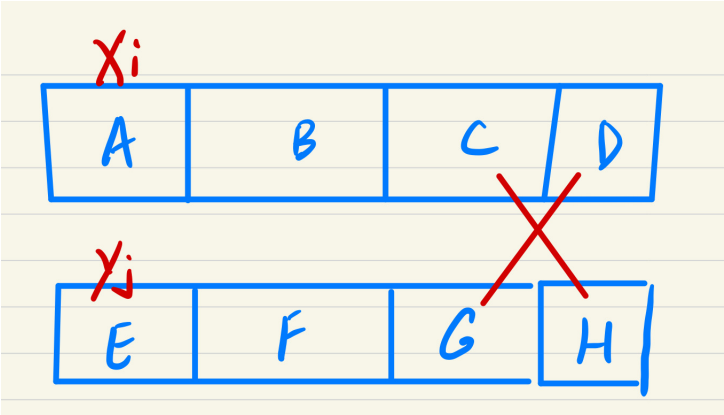

Figure 1: Merge Sequences

5.1 Strategy

The key of this problem is to **find the best place to cross the sequences,** because at the same sequence there will be no cost. To address this, we need a 2D recurrence relation.

5.2 Recurrence Relation

Subproblem: Define $OPT(i, j)$ as the minimum cost of merging the elements before (and include) x_i and y_j (both using 0-based).

Recurrence relation could be expressed as:

$$OPT(i,j) = \min \begin{cases} OPT(i-1,j) + C[i,j] & \text{Add } x_i \text{ from } y_j \\ OPT(i,j-1) + C[i,j] & \text{Add } y_j \text{ from } x_i \end{cases} \tag{13}$$

5.3 Complexity Analysis

Both time complexity and space complexity are $O(mn)$.

6 String Reading (Segmentation, Spring 2024)

Many languages (including Chinese and Japanese) are written without spaces between words. You are given text in such a language, and you are required to design an algorithm to infer likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like "meeteateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight", or "meet ate ight", or any of a huge number of possibilities).

To solve the problem, you are given a function `Quality()` that takes any string of letters and returns a number that indicates the quality of the word formed by the string. A high number indicates that the string resembles a word in the language (e.g. "meet"), whereas a low number means that the string does not resemble a word (e.g. "eeta"). The total quality of a segmentation is determined by adding up the qualities of each of its words.

Design a bottom-up dynamic programming algorithm that takes a string y and computes a segmentation of maximum total quality. You are required to provide the recurrence relation, pseudocode and running time analysis (you can treat the call to `Quality` as a single computational step, $O(1)$).

6.1 Strategy

In this problem, we need to examine all the combinations possible to find the optimal solution. This is usually done by segmentation method, which is introducing a new variable to loop through the existing index.

6.2 Recurrence Relation

Subproblem: Define $OPT[i]$ be the maximum quality of segmentation for the prefix of the string y of length i.

Recurrence relation could be expressed as:

$$OPT[i] = \max_{0 \le j \le i}(OPT[j] + Quality(y[j+1:i])) \tag{14}$$

Because an empty string has zero quality, so the base case will be:

$$OPT[0] = 0 \tag{15}$$

6.3 Bottom-up

Algorithm 6 String Reading (Bottom-up)

```
function SEG_MAX_QUALITY(y)
    n ← length(y)
    Initialize OPT as a (1 × (n + 1)) array
    OPT[0] = 0

    for i ← 1 to n do
        max_quality = float('inf')
        for j ← 0 to i do
            max_quality = max(max_quality, OPT[j] + Quality(y[j + 1 : i]))
        end for
    end for

    return OPT[n]
end function
```

6.4 Top-down

Algorithm 7 String Reading (Top-down)

```
function SEG_MAX_QUALITY(y)
    n ← length(y)
    Initialize OPT as a (1 × (n + 1)) array, assign all the values as −1
    OPT[0] = 0

    function COMPUTE_MAX_QUALITY(i)
        if OPT[i]! = −1 then
            return OPT[i]
        end if
        max_quality = float('-inf')

        for j ← 0 to n do
            max_quality = max(max_quality, OPT[j] + Quality(y[j + 1 : i]))
        end for

        OPT[i] = max_quality
        return OPT[i]
    end function

    return compute_max_quality(n)
end function
```

6.5 Complexity Analysis

The time complexity of this problem is $O(n^2)$ because there is a nested loop. The space complexity of this problem is $O(n)$, we only need a 1D array.

7 Bookshelf (Segmentation, Fall 2023)

You are given n books, $b_1, b_2, \ldots, b_n$ that need to be arranged into a bookshelf. The books are already sorted by their indices. Each book b_i has thickness t_i and height h_i. The books must be arranged in the given order of their indices, from the lowest level to the highest level of the bookshelf. The bookshelf has a total width of L, and the height of each level on the bookshelf can be adjusted.

The aim is to minimize the *total space usage* of the n books, defined as the sum of the heights of the highest book on each level, multiplied by the bookshelf width L. An illustration is shown below (the figure may not show an optimal solution):

Example: we have three books b_1, b_2, b_3. The thickness values are: $t_1 = 1$, $t_2 = 1$ and $t_3 = 1$. The heights of the books are: $h_1 = 1$, $h_2 = 2$, $h_3 = 3$. The width of bookshelf $L = 2$. The optimal solution is to put b_1 on level 1 and put b_2 and b_3 on level 2, which results in a total space usage of 8.

Please design a dynamic programming algorithm to find the minimum total space usage of the n books. Please define the subproblem(s) and give the recurrence relations. Analyze the time and space complexity of your algorithm. Backtracing step and pseudocode are not required.

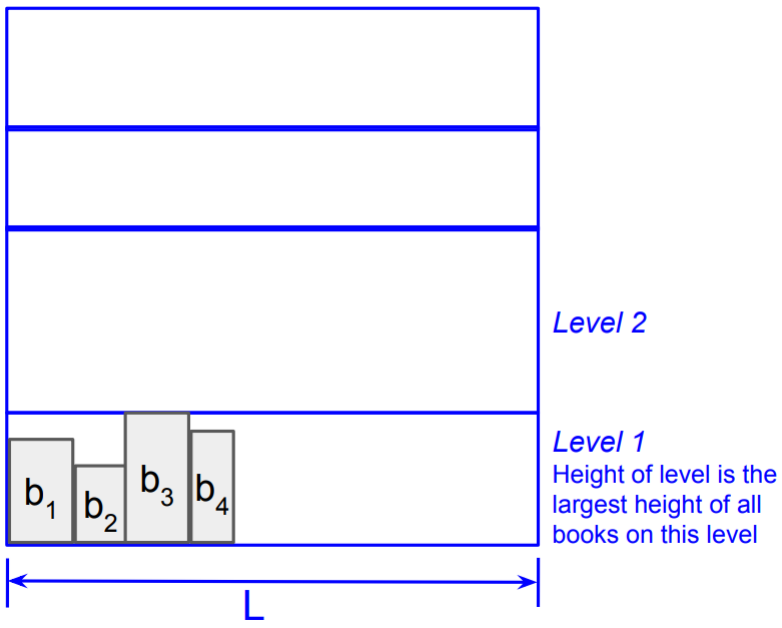

Figure 2: Bookshelf Problem

7.1 Strategy

This problem includes many situations. When we decide either we take next book into the same level, or put it to the next level with other previous books, we also need to consider the length constraint. However, all of these are overthinking. We need to use dynamic programming to simplify the problem, utilizing segmentation method.

7.2 Recurrence Relation

To define the recurrence relation, we need to consider how to place i^{th} book and the books before it. Specifically, we need to determine **where the last level starts**.

For each book, consider placing it on a new level or adding it to the current level. For a new level, we need to consider the total thickness of books up to i not exceeding L.

Now let's iterate each possible **starting point of the last level**, denoted by j (j ranges from 1 to i). The height of the last level will be the **maximum height among the books from j to i**.

Subproblem: Define $OPT[i]$ as the minimum space usage for the first i books.

Recurrence relation could be expressed as:

$$OPT[i] = \min_{1 \le j \le i}(OPT[j-1] + \max(h[j], h[j+1], \cdots, h[i]) \times L) \tag{16}$$

This equation works if $t[j] + t[j+1] + \cdots + t[i] \le L$. Here, the base case will be $OPT[0] = 0$.

7.3 Complexity Analysis

The time complexity will be $O(n^2)$, because there is a nested loop. The space complexity will be $O(n)$, because we only use a 1D array.

8 Longest Subsequence (Fall 2022)

You are given a list of distinct numbers $a_1, a_2, \ldots, a_n$. Please design a dynamic programming algorithm which finds the longest subsequence of numbers where the numbers are strictly increasing from smaller indices to larger indices. A subsequence means a subset of numbers from the original list where the relative positions of numbers should be maintained but the indices need not be consecutive. That is, in subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$, we have $i_1 < i_2 < \cdots < i_m$.

For example, given a list of numbers {82, 77, 65, 89, 83, 68, 88, 71, 91}, one optimal solution is {77, 83, 88, 91}.

Please describe the algorithm clearly (you may give the pseudocode), give the recurrence relations, and analyze the time and space complexity of your algorithm. Remember to include steps to output the optimal subsequence (you do not need to output all the optimal solutions if there are more than one.)

8.1 Strategy

In this problem, it is similar with finding the compatible combinations and excluding the incompatible ones. But in this case, we need to examine all the possible cases. Notice that in the recurrence relation we could actually state the required conditions, without actually implementing them into the loop.

8.2 Recurrence Relation

Subproblem: Define $OPT[i]$ be the length of the longest subsequence that ends with the element $a[i]$.

Recurrence relation could be expressed as:

$$OPT[i] = \max_{0 \le j \le i}(OPT[j] + 1, OPT[i-1]) \text{ if } a[j] < a[i] \quad (17)$$

Because each element is a subsequence of length 1 by itself, so $OPT[i] = 1$ initially for all i.

8.3 Bottom-up, Backtracing

Algorithm 8 Longest Subsequence (Bottom-up)

```
1:  function LONGEST_SUBSEQUENCE(a)
2:      n ← length(a)
3:      OPT = [1] * n
4:      prev = [−1] * n
5:
6:      for i ← 1 to n do
7:          for j ← 0 to n do
8:              if a[j] < a[i] and OPT[i] < OPT[j] + 1 then
9:                  OPT[i] = OPT[j] + 1
10:                 prev[i] = j
11:             end if
12:         end for
13:     end for
14:     max_length = max(OPT)
15:     index = OPT.index(max_length)
16:     lis = []
17:     while index! = −1 do
18:         lis.append(a[index])
19:         index = prev[index]
20:     end while
21:     return lis
22: end function
```

8.4 Complexity Analysis

The time complexity of this problem is $O(n^2)$, because there is a nested loop. The space complexity of this problem is $O(n)$, because we only use a 1D array.

9 Meal Delivery (Fall 2022)

As the new semester starts, George is making a plan for ordering meal deliveries from a restaurant for the entire semester. For each week, George has two choices: either skip the week or order a meal for the entire week. Since the restaurant shares the menu of each week, George can give a tasty score to the menu of each week depending on how much he likes the food. Suppose there are n weeks, the tasty scores are $x_1, x_2, \ldots, x_n$, where each score is an integer, and can be positive, 0, or negative.

Since the restaurant encourages ordering of consecutive weeks, there can be penalties for skipping weeks:

- If George skips one week, there is no penalty.
- If George skips two or three consecutive weeks, there will be a penalty of 20 points.
- Skipping for four weeks or more than four weeks is not allowed. This can be considered as a penalty of ∞ points.

The overall happiness score of the semester is the sum of all the tasty scores of the weeks George decides to order delivery, minus the corresponding penalty for the weeks that are skipped. For example, if there are 5 weeks, and the tasty scores are $\{10, -10, -5, 15, 6\}$, and the plan for the 5 weeks is {order, skip, skip, order, order}, the overall happiness score is $10 - 20 + 15 + 6 = 11$.

Please design a dynamic programming algorithm to find the weekly plan for George that maximizes the overall happiness score of the semester. Please describe the algorithm clearly (you may give the pseudocode), give the recurrence relations, and analyze the time and space complexity of your algorithm. Remember to include steps to output the optimal weekly plan.

9.1 Strategy

This problem is straight forward. We need the determine the week i is better if we order or skip, and also how many weeks to skip.

9.2 Recurrence relation

Subproblem: Define $OPT[i]$ as the maximum happiness score we can get until week i.

Recurrence relation could be expressed as:

$$OPT(i) = \max \begin{cases} OPT[i-1] \\ OPT[i-1] + x_i \\ OPT[i-2] - 20 \\ OPT[i-3] - 20 \end{cases} \tag{18}$$

The base cases include many situations:

$$OPT[0] = 0 \tag{19}$$

$$OPT[1] = \max\{OPT[0], OPT[0] + x_1\} \tag{20}$$

$$OPT[2] = \max\{OPT[1], OPT[1] + x_2, OPT[0] - 20\} \tag{21}$$

9.3 Pseudocode

Similar with the maximum candies case.

9.4 Complexity Analysis

The space complexity is $O(n)$, and the time complexity is $O(n)$

10 Cut Points (Spring 2022)

Let A be the set of all integers in the range of 1 to n. For each pair of numbers in A, denoted as (i, j), where $1 \leq i \leq j \leq n$, a cost function $C[i, j] > 0$ is defined and given. The task is to find an increasing sequence of cutpoints $i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, n-1\}$ to minimize the total cost $\sum_{t=0}^{k} C[i_t + 1, i_{t+1}]$, where $i_0 = 0$ and $i_{k+1} = n$.

In other words, you need to partition the sequence $\{1, 2, \ldots, n\}$ into $k+1$ segments, where the t^{th} segment ends with number i_t, and you want to find the best way of partitioning such that the total cost is minimized. Note that k is not fixed and it depends on the solution you find.

Explain a Dynamic Programming algorithm to find the minimum cost of the segmentation and provide the recurrence (including the base case(s)). Analyze the time and space complexity.

10.1 Strategy

This problem is a classical segmentation problem. It seems like multiple cut points will generate multiple segments, make the situation very complex. But actually the DP will take care of it, starting from the smallest segment and consider each segment combination. The recurrence relation is similar with that of *Quality* problem.

10.2 Recurrence Relation

Subproblem: We define $OPT[i]$ as the minimum cost we could reach for the integers in range of 1 to i.

Recurrence relation could be expressed as:

$$OPT[i] = \min_{1 \leq j \leq i}(OPT[j] + C[j, i]) \tag{22}$$

The base case will be:

$$OPT[1] = 0 \tag{23}$$

10.3 Complexity Analysis

The time complexity will be $O(n^2)$, because there is a nested loop. The space complexity will be $O(n)$ because we are using 1D array.

11 Difficult Assignment (Spring 2022)

You are taking a course this semester which has n weeks. Each week, there are two assignments released, an "easy" assignment and a "difficult" assignment. You can choose from one of them but you cannot choose both during the same week. At week i, accomplishing the easy assignment will earn you e_i points, and the difficult assignment will earn you d_i points. However, if you plan to do a difficult assignment in week i, you must do no assignment in week $i-1$, because you will need the time of week $i-1$ to study and prepare for the difficult assignment in week i. On the other hand, you can take an easy assignment in week i no matter what assignment you do in week $i-1$.

Given a sequence of e_i values for the points one can gain through the easy assignments, and a sequence of d_i values for the points one can gain through the difficult assignments, $1 \leq i \leq n$, you need to come up with a plan for each week, in order to maximize the total points you gain for the semester. Note that when you devote a week to work on an assignment you always get full credits, that is, getting partial credits is not a scenario to consider. Therefore, for each week i, there are three choices: 1) take an easy assignment and earn e_i points; 2) take a difficult assignment and earn d_i points; 3) take no assignment and earn 0 points.

Please design a dynamic programming algorithm that finds the optimal total points and the optimal weekly plan. Please describe the algorithm clearly (you may give the pseudocode), give the recurrence relations, and analyze the time and space complexity of your algorithm.

11.1 Strategy

Classical incompatible question.

11.2 Recurrence Relation

Subproblem: We define $OPT[i]$ as the maximum total points we could get until week i.

Recurrence relation could be expressed as:

$$OPT(i) = \max \begin{cases} OPT[i-1] + e_i \\ OPT[i-2] + d_i \end{cases} \tag{24}$$

The base case will be:

$$OPT[0] = 0 \tag{25}$$

$$OPT[1] = e_1 \tag{26}$$

11.3 Complexity Analysis

Both the time and space complexities are $O(n)$.

12 Courses and Hours (Fall 2020)

Suppose you are taking n courses, each with a project that has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you will spend an integer number of hours on each project.

To figure out how to best divide up your time, you have come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ to estimate the grade you will get for each project given that you spend h hours on that project. That is, if you spend $h \leq H$ hours on the project for course i, you will get a grade of $f_i(h)$. You may assume that the functions f_i are non-decreasing: if $h < h'$, then $f_i(h) \leq f_i(h')$.

So the problem is: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your total grade, as computed according to the f_i, is as large as possible. The running time of your algorithm should be polynomial in n and H. Please describe the algorithm clearly (you may give the pseudocode), give the recurrence relations, and analyze the time and space complexity of your algorithm.

12.1 Strategy

This problem has two constraints, the first one is to make sure the total hours smaller than H, and we want to maximize the points we could get. This is similar with the votes problem, we need a 2D DP.

12.2 Recurrence Relation

Subproblem: We define $OPT[i][h]$ as the maximum points up to course i, with remaining available hours as h.

Recurrence relation could be expressed as:

$$OPT[i][h] = \max_{0 \leq x \leq h}(OPT[i-1][h-x] + f_i(x)) \tag{27}$$

Here $0 \leq i \leq n$, and $0 \leq h \leq H$. The base case will be:

$$OPT[0][0] = 0, \;\; OPT[i][0] = 0, \;\; OPT[0][h] = 0 \tag{28}$$

12.3 Complexity Analysis

Notice that this recurrence relation includes three loops (i, h, x), therefore the time complexity will be $O(nH^2)$. And because we are using 2D array, so the space complexity will be $O(nH)$.

13 Red, Black Color (Fall 2019)

Consider the following game. You are given a sequence of n positive numbers $(a_1, a_2, \ldots, a_n)$. Initially, they are all colored black. At each move, you choose a black number a_k and color it and its immediate neighbors (if any) red (the immediate neighbors are the elements a_{k-1}, a_{k+1}). You get a_k points for this move. The game ends when all numbers are colored red. The goal is to get as many points as possible.

(a) Describe a greedy algorithm for this problem. Verify that it does not always maximize the number of points and give a tight approximation ratio (i.e., provide a family of instances where the greedy algorithm returns solutions that reach this bound, and an informal proof that, on any instance, the solution returned by the greedy algorithm will not exceed that bound).

(b) Describe and analyze an efficient dynamic programming algorithm for this problem that returns optimal solutions. (Linear time is possible.)

13.1 Strategy

The DP problem is just a simple compatible question. For the Greedy part, notice that the sequence is not sorted, so this problem is more like a Knapsack problem, we want to choose the ball with largest points first.

13.2 Greedy

The Greedy algorithm is shown below:

1. At each step, choose the largest available number a_k **that is still black**.
2. Color a_k and its immediate neighbors (if any) red.
3. Add a_k to the total points, repeat until all numbers are colored red.

13.3 Recurrence Relation

Subproblem: We define $OPT[k]$ as the maximum points we can get up to the ball k.

Recurrence relation could be expressed as:

$$OPT[k] = \max\{OPT[k-2] + a_k, OPT[k-1]\} \tag{29}$$

Here $1 \leq k \leq n$. The base case will be:

$$OPT[0] = 0 \tag{30}$$

$$OPT[1] = a_1 \tag{31}$$

13.4 Complexity Analysis

Both the time and space complexities are $O(n)$.

14 Minimize Length (Hardest, Fall 2020, Fall 2019, HW 4)

You are given a sorted set of points $P = (P_1, P_2, \ldots, P_n)$ on a line. Given a constant k, show how to select a subset of $k-1$ of these points, say (still in sorted order) $(P_{j_1}, \ldots, P_{j_{k-1}})$, so as to partition the segment from P_1 to P_n into k pieces that are as close to equal in length as possible. Specifically, writing $L = (P_n - P_1)/k$, we want to minimize the square error

$$(P_{j_1} - P_1 - L)^2 + \sum_{i=1}^{k-2} (P_{j_{i+1}} - P_{j_i} - L)^2 + (P_n - P_{j_{k-1}} - L)^2$$

Describe and analyze an algorithm for this problem that runs in $\Theta(kn^2)$ time.

14.1 Strategy, Complexity Analysis

The running time is a major hint as to the nature of the DP. The key observation is that the optimal way of selecting j points from points $1, \ldots, i$ (such that these j points partition the segment between these r points with minimum square error) with the jth point being i must consist of an optimal way of selecting $j-1$ points from points $1, \ldots, r$, where r can be any point in $j, \ldots, i$ and the $(j-1)$st point is r.

To build the DP matrix for this recurrence, we use one row for each sequence of points $1, \ldots, i$, for $i = 1, \ldots, n$ (hence n rows) and k columns. Cell (i, j) contains the list of $j-1$ points, delimiting j intervals (point i is the jth partition point), selected from points $(1, \ldots, i)$. The first column is trivial to fill since the point selected has to be the end point of the interval. The matrix will be lower triangular since we must have $i > j$. To fill a cell (i, j) we need to look at only the entries of the previous column: to select $j-1$ points from the best possible way of selecting $j-2$ points. Overall, then, we use $\Theta(n)$ time per cell and **thus $\Theta(n^2k)$ time overall, using $\Theta(nk)$ space.**

14.2 Recurrence Relation

Subproblem: We define $OPT[i][j]$ as the optimal cost for selecting $j-1$ points from the first i points to partition the segment from P_1 to P_i into j pieces.

Recurrence relation could be expressed as:

$$OPT(i,j) = \begin{cases} \min\limits_{j-1<r\leq i} [OPT(r,j-1) + (P_i - P_r - L)^2] & \text{if } j > 1 \\ (P_i - P_1 - L)^2 & \text{if } j = 1 \end{cases} \tag{32}$$

15 Constraint Subset (Final)

You are given an interval $[0, M]$ and you have n points on this interval $x_1, x_2, \ldots, x_n$ (see illustration below). Each point x_i is associated with a score s_i. You will be choosing a subset of these points, and earn the total score which is the sum of scores of the chosen points. There is a constraint, that the distance between any two adjacent chosen points is at least k, where $k < x_n - x_1$.

Please design a dynamic programming algorithm to choose the subset of points from $x_1, x_2, \ldots, x_n$ such that the total score you collect is maximized.

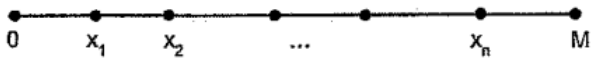

Figure 3: Constraint Subset

1. Define the subproblem and write the recurrence (include base cases) relations.
2. Provide the bottom-up implementation pseudocode.
3. What are the time and space requirements in terms of n of the bottom-up implementation?
4. Give pseudocode for the back-tracing step that returns the optimal subset of points.

15.1 Strategy

This problem is similar with the **Quality problem**. In the Quality problem, if we choose i^{th} point, we need to consider j^{th} point with $0 \leq j \leq i$. But in this problem, j is not explicitly stated, it is determined by a constraint. In this case, the constraint could not easily stated using math relation, **so we can just state it in words!** Notice that in Quality problem, we have to choose all the points (segmentation), but here we are allowed to not include some points, therefore the binary choices need to be implemented.

15.2 Recurrence Relation

Subproblem: We define $OPT[i]$ as the maximum score we can get by considering the first i points and including the i^{th} point.

Recurrence relation could be expressed as:

$$OPT[i] = \max\{OPT[i-1], s_i + OPT[j]\} \tag{33}$$

Here j is the **largest index such that** $x_j \leq x_i - k$. Therefore, the base case will be (assuming there is no point to consider):

$$OPT[0] = 0 \tag{34}$$

15.3 Bottom-up

```
function maxScore(n, k, x, s):
    # Initialize dp array
    dp = array of size n + 1, all elements set to 0

    # Populate the dp array
    for i from 1 to n:
        # Find the largest j such that x[j] <= x[i] - k
        j = i - 1
        while j >= 1 and x[j] > x[i] - k:
            j -= 1

        if j >= 1:
            dp[i] = max(dp[i-1], s[i] + dp[j])
        else:
            dp[i] = max(dp[i-1], s[i])

    # The answer will be in dp[n]
    return dp[n]
```

Figure 4: Bottom-up Algorithm

15.4 Backtracing

```
function findOptimalSubset(n, k, x, s, dp):
    subset = empty list
    i = n

    while i > 0:
        # If including point i gives a better score
        if dp[i] == dp[i-1]:
            i -= 1
        else:
            subset.add(x[i])
            # Find the largest j such that x[j] <= x[i] - k
            j = i - 1
            while j >= 1 and x[j] > x[i] - k:
                j -= 1
            i = j

    # Reverse the subset to get the correct order
    subset.reverse()

    return subset
```

Figure 5: Backtracing Algorithm

15.5 Complexity Analysis

The time complexity is $O(n^2)$ because we have a nested loop. The space complexity is $O(n)$ because we are using 1D array.