

1 Introduction

Dynamic programming (DP) is a method for solving complex problems by breaking down the original problem into relatively simpler sub-problems. It is not a specific algorithm but a method to solve specific problems, it appears in a variety of data structures, and the types of questions related to it are more complicated. Common types of DP includes:

Linear DP, Knapsack DP, Interval DP, DAG DP and Tree DP.

Without specified mentioned, assume `arr` is declared as a signed integer array. The following code is added to the beginning of all C++ programs.

```
#include <bits/stdc++.h>
using namespace std;
```

In the whole topic of DP, we will use 1-based indexing for convenience.

2 Knapsack DP

Definition

The **Knapsack Problem** is a classic dynamic programming (DP) problem.

You are given a set of items, each with a weight and a value, and you need to maximize the total value without exceeding a given weight capacity of the knapsack.

0/1 Knapsack

There are N items and a knapsack with capacity M .

The weight of the i^{th} item is w_i , the value is v_i .

Find out the largest total cost of the items that the knapsack can afford.

1. Observation

- Let v be the current value, c be the remaining capacity:

- Before putting the i^{th} item: (v, c)
- After putting the i^{th} item: $(v + v_i, c - w_i)$

2. Define state and target

- Let $dp[i][j]$ be the max value of putting the first i items in a bag of capacity j .
- The target of the question is $dp[N][M]$.

3. Transitional equation

- Non-optimized: $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$
- Rolling Array Optimized: $dp[j] = \max(dp[j], dp[j-w_i] + v_i)$

4. Initialize DP array

- $dp[0][k] = 0$ for any k

5. Confirm the traversal order

- Non-optimized: i from 1 to N , j in any order
- Rolling Array Optimized: j from M to 0

6. Time and space complexity

- Time Complexity: $O(NM)$
- Space Complexity:
 - Non-optimized: $O(NM)$
 - Optimized: $O(M)$

C++ Code Implementation

```
int N, W, w, v;
int dp[1100];
int main() {
    cin >> W >> N;
    for (int i = 1; i <= N; i++) {
        cin >> w >> v;
        for (int j = W; j >= 0; j--)
        {
            if (j-w >= 0) dp[j] = max(dp[j], dp[j-w] + v);
        }
    }
    cout << dp[W];
    return 0;
}
```

Unbounded Knapsack (UKP)

UKP is a variation of 0/1 Knapsack Problem which allows unlimited repetition of items.

1. Naïve Solution

$$f(i, j) = \max \begin{cases} f(i-1, j) \\ f(i-1, j - v[i] * 1) + w[i] * 1 \\ \dots \\ f(i-1, j - v[i] * k) + w[i] * k \quad k * v[i] \leq j \end{cases}$$

2. Observation

- Note that the enumerate of j is from 0 to M
- We can get all info of $j - k * v[i]$ before j
- $j - v[i]$ is already updated using the info of $j - 2 * v[i]$
- We just need to compare j and $j - v[i]$

3. Define state and target

- Let $dp[i][j]$ be the max value of putting the first i items in a bag of capacity j .
- The target of the question is $dp[N][M]$.

4. Transitional equation

- Non-optimized: $dp[i][j] = \max(dp[i-1][j], dp[i][j - w_i] + v_i)$
- Rolling Array Optimized: $dp[j] = \max(dp[j], dp[j - w_i] + v_i)$

5. Initialize DP array

- $dp[0][k] = 0$ for any k

6. Confirm the traversal order
 - Non-optimized: i from 1 to N, j from 0 to M
 - Rolling Array Optimized: j from 0 to M
7. Time and space complexity
 - Time Complexity: $O(NM)$
 - Space Complexity:
 - Non-optimized: $O(NM)$
 - Optimized: $O(M)$

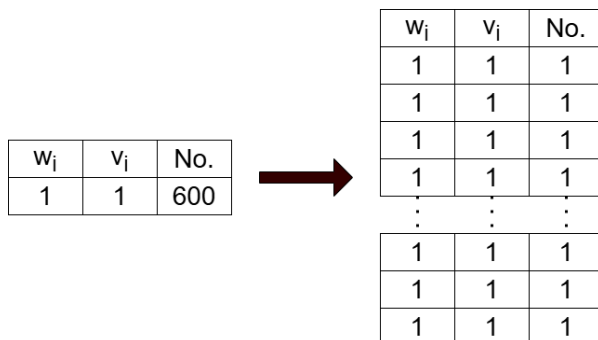
C++ Code Implementation

```
int N, W, w, v;
long long dp[10000000];
int main() {
    cin >> W >> N;
    for (int i = 1; i <= N; i++) {
        cin >> w >> v;
        for (int j = 0; j <= W; j++)
        {
            if (j-w >= 0) dp[j] = max(dp[j-w] + v, dp[j]);
        }
    }
    cout << dp[W];
    return 0;
}
```

Bounded Knapsack (BKP)

BKP is a variation of 0/1 Knapsack Problem which there are k_i of each type of item.


1. Naïve Solution



2. Binary Grouping Optimization

- **Any number can be represented by sum of 2^k s:** e.g. $11 = 2^0 + 2^1 + 2^3$
- Think about the binary representation of a number will easily understand this statement.
- As $600 = 1 + 2 + 4 + \dots + 128 + 256 + 89$, we can actually split the items like this:

w_i	v_i	No.
1	1	600



w_i	v_i	No.
1	1	1
2	2	1
4	4	1
8	8	1
...
128	128	1
256	256	1
89	89	1

- All cases can be represented by the numbers split:
 1. Putting 0 to 511 of the items can be represented by combinations of putting items weighted 1, 2, 4, ..., 256.
 2. Putting 512 to 600 of the items can be represented by putting item weighted 89 + putting 423 to 511 of the items (which is same as case 1)
- By just doing 0/1 Knapsack on the binary grouping table, we can obtain the optimal solution among all the possible ways of putting 600 same type of items.

3. Define state and target

- Let $dp[i][j]$ be the max value of putting the first i items in a bag of capacity j .
- The target of the question is $dp[N][M]$.

4. Transitional equation

- Non-optimized: $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$
- Rolling Array Optimized: $dp[j] = \max(dp[j], dp[j-w_i] + v_i)$

5. Initialize DP array

- $dp[0][k] = 0$ for any k

6. Confirm the traversal order

- Non-optimized: i from 1 to N , j from 0 to M
- Rolling Array Optimized: j from M to 0

7. Time and space complexity

- Time Complexity:
 - Non-optimized: $O(W \sum_{i=1}^N k_i)$
 - Optimized: $O(W \sum_{i=1}^N \log k_i)$.
- Space Complexity:
 - Non-optimized: $O(NM)$
 - Optimized: $O(M)$

C++ Code Implementation

```
int N, M, ans, cnt=1;
int dp[1000005], w[1000005], v[1000005];
int main()
{
    int vi, wi, ki;
    cin >> N >> M;
    for (int i = 1; i <= N; i++)
    {
        cin >> vi >> wi >> ki;
        for (int j = 1; j <= ki; j <= 1)
        {
            v[++cnt] = j * vi, w[cnt] = j * wi;
            ki -= j;
        }
        If (ki) v[++cnt] = vi * ki, w[cnt] = wi * ki;
    }
    for (int i = 1; i <= cnt; i++)
    for (int j = M; j >= w[i]; j--) dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    cout << dp[M];
    return 0;
}
```

3 Interval DP

Definition

Interval DP is a dynamic programming technique used to solve problems.

The solution involves optimizing or counting over **subintervals** of a given sequence (e.g., arrays, strings). It breaks down the problem into smaller subproblems defined over intervals and combines their solutions.

In simple words, **Interval DP** is often used in problems requiring you to merge (or split) subintervals while each operation worth a cost.

Stone Merging Problem

There are N stones arranged in a chain. The i^{th} stone is marked a score a_i on it.

In each operation, you merge two adjacent piles of stones into one, and the score gained is equal to the sum of the numbers on the stones in the merged pile. The goal is to maximize the total score.

1. Observation

- To get a result of 3 stones merged together, the last step must be merging two piles.
- For example, to get the pile $a_{1...3}$, we must merge a_1 to $a_{2...3}$ or merge a_3 to $a_{1...2}$.
Trivially, it's impossible to merge a_2 to some piles to get $a_{1...3}$.
- Therefore, to get the pile $a_{1...3}$, we can either have score:

1. Max score got from the pile $a_{2...3} + \sum_{i=1}^3 a_i$

2. Max score got from the pile $a_{1...2} + \sum_{i=1}^3 a_i$

3. Maximum score we can get from pile $a_{1...3} = \max$ (Case 1, Case 2)

- It's not hard to derive that, to get the pile $a_{i...j}$, we can have score:

1. Max score of $a_{i...k} + a_{k+1...j} + \sum_{t=i}^j a_t$ where $k \in [i, j - 1]$.

- Max score of $a_{i...j} = \max (k = i, k = i + 1, \dots, k = j - 1)$.

2. Define state and target

- Let $dp[i][j]$ be the max score got from the pile $a_{i...j}$.
- The target of the question is $dp[1][N]$.

3. Transitional equation

- $dp[i][j] = \max_{k \in [i, j-1]} dp[i][k] + dp[k+1][j] + \sum_{t=i}^j a_t$

4. Initialize DP array

- $dp[k][k] = 0$ for any k

5. Confirm the traversal order

- len from 1 to N , i from 1 to $N + 1 - \text{len}$, k from i to $j - 1$

6. Time and space complexity

- Time Complexity: $O(N^3)$
- Space Complexity: $O(N^2)$

C++ Code Implementation

```
int N, m;

int dp[310][310], stone[310], psum[310];

int main(){
    cin >> N;

    for (int i = 1; i <= N; i++)
    {
        cin >> stone[i];
        psum[i] = psum[i - 1] + stone[i];
        for (int j = 1; j <= N; j++) dp[i][j] = 1e9;
        dp[i][i] = 0;
    }

    for (int len = 1; len <= N; len++)
    {
        for (int i = 1; i + len - 1 <= N; i++)
        {
            int j = i + len - 1;
            for (int k = i; k <= j - 1; k++)
            {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + psum[j] - psum[i - 1]);
            }
        }
    }

    cout << dp[1][N];

    return 0;
}
```


Longest Palindromic Subsequence (LPS)

Given a sequence a with length n , find the length of the longest palindromic subsequence of a .

If a subsequence is **palindromic**, it's equal to itself after reverse.

1. Observation

- Same as the LCS Problem, but do it in both end of the same sequence.

2. Define state and target

- Let $dp[i][j]$ be the longest palindromic subsequence of $a[i : j]$
- The target of the question is $dp[1][n]$.

3. Transitional equation

- For $a[i] \neq a[j]$, $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$
- For $a[i] = a[j]$, $dp[i][j] = \max(dp[i+1][j], dp[i][j-1], dp[i+1][j-1] + 2)$

4. Initialize DP array

- $dp[k][k] = 1$ for any k

5. Confirm the traversal order

- len from 1 to n , i from 1 to n

6. Time and space complexity

- Time Complexity: $O(n^2)$
- Space Complexity: $O(n^2)$

C++ Code Implementation

```
string s; int n, dp[5010][5010]; char num[5010];
int main() {
    cin >> s;
    n = int(s.size());
    for (int len = 1; len <= n; len++) {
        for (int i = 1; i + len - 1 <= n; i++) {
            int j = i + len - 1;
            dp[i][j] = max(dp[i][j-1], dp[i+1][j]);
            if (s[i-1] == s[j-1]) dp[i][j] = max(dp[i][j], dp[i+1][j-1] + 2);
            if (i == j) dp[i][j] = 1;
        }
    }
    cout << dp[1][n];
    return 0;
}
```

Post Office Problem

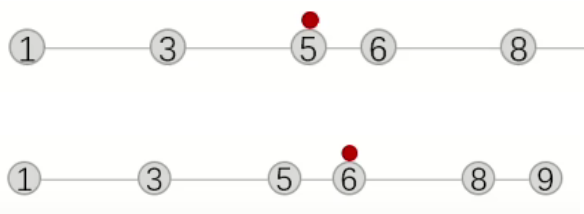
There are n villages on some position on a number line.

m post offices can be built in some villages.

Find the least sum of distances between the villages and the corresponding nearest post office of each village.

1. Observation

- Consider building only one post office in some villages:
 - Let $w[i][j]$ be the minimum total distance of building a post office in the i^{th} village to the j^{th} village.
 - It can be easily observed and proven that the spot that we should build the post office is the **median** of those villages.



No. of villages is odd, build in median.

No. of villages is even, build in any of the two villages at the middle. (5 or 6)

- All $w[i][j]$ can be calculated by recursion:
 - $w[i][j] = w[i][j-1] + (x[j] - x[\frac{i+j}{2}])$

2. Define state and target

- Let's define $dp[i][j]$ be minimum total distance of building j post offices in the first i^{th} village.
- The answer of the question is $dp[n][m]$.

3. Transitional equation

- $dp[i][j] = \min_{k \in [0, i-1]} dp[k][j-1] + w[k+1][i]$

4. Initialize DP array

- $dp[1][k] = dp[k][0] = w[k][k] = 0$ for any k

5. Confirm the traversal order

- i from 1 to n , j from 1 to m , k from 0 to m

6. Time and space complexity

- Time Complexity: $O(n^2m)$
- Space Complexity: $O(nm)$

C++ Code Implementation

```

int n,m;
int x[3001];
int dp[3001][301];
int w[3001][3001];
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> x[i];
    for (int i = 1; i <= n; i++)
    {
        for (int j = i + 1; j <= n; j++)
        {
            w[i][j] = w[i][j-1] + x[j] - x[(i+j)/2];
        }
    }
    memset(dp,0x3f,sizeof(dp));
    dp[0][0] = 0;
    for (int j = 1; j <= m; j++)
    {
        for (int i = j; i <= n; i++)
        {
            for (int k = 0; k < i; k++)
            {
                if (dp[i][j] > dp[k][j-1] + w[k+1][i])
                {
                    dp[i][j] = dp[k][j-1] + w[k+1][i];
                }
            }
        }
    }
    cout << dp[n][m] << endl;
    return 0;
}

```