Solving Double Counting Problems with Dynamic Programming Algorithm

Wenrong Jiang*
School of Computer and Information, Shanghai Polytechnic University, Shanghai, China
*Corresponding author email: wrjiang@sspu.edu.cn

Abstract. In view of the repetitive calculation problems that often appear in algorithmic questions, this paper analyzes the common thought methods and dynamic programming for solving such problems. Through several types of classic questions, such as the backpack problem, the longest common subsequence, etc., the analysis methods and scope of application of dynamic programming are summarized.

Keywords: Dynamic programming; Overlapping sub-problems; Knapsack problem; Fibonacci sequence.

1. Introduction
Dynamic Programming is an algorithmic idea for solving a class of optimization problems. If a problem can be broken down into several sub-problems and these sub-problems are repeated, the problem is called Overlapping Subproblems[1]. Dynamic programming avoids a lot of double counting by directly recording the results of overlapping sub-problems so that the same sub-questions are used the next time. Therefore, a problem must have overlapping sub-problems in order to use dynamic programming to solve.

2. What is Dynamic Planning?
Dynamic Programming is an algorithmic idea for solving a class of optimization problems.[2] In simple terms, dynamic programming decomposes a complex problem into several sub-problems, and obtains the optimal solution of the original problem by synthesizing the optimal solution of the sub-problem. The application scenarios are very wide, and the classic questions have Fibonacci numbers, knapsack problems[3], and the largest continuous subcolumns and other series of problems.

Next, the author begins with the simplest Fibonacci sequence to analyze how dynamic programming reduces the problem of repetitive calculations to reduce time complexity.

2.1. Testing the Fibonacci sequence
The Fibonacci sequence is a classic algorithm problem, defined as F0 =1, F 1 =1, Fn=Fn-1 +Fn-2 (n>=2). The simplest algorithm is as follows:

```c
int F(int n)
{
    if(n == 0 || n == 1) return 1;  /* If it is F 0 or F 1, the return value is 1
else return F(n-1) + F(n-2);   /* Otherwise recursive*/
}
```

This recursive method involves a large number of repeated calculations. Take Figure 1 as an example. When n==5, you can get F(5)=F(4)+F(3). When F(4) is calculated, there will be F(4)=F(3)+F(2). In
Figure 1, the gray frames are all duplicated. It is inferred from this that when the value of \( n \) is large, the number of repeated calculations will be unimaginable. The time complexity will be as high as \( O(2^n) \).

![Fibonacci sequence recursion diagram.](image1.png)

![Fibonacci sequence memorized search graph.](image2.png)

To avoid double counting, you can open a one-dimensional array \( dp \) to hold the calculated results, where \( dp[n] \) records the result of \( F(n) \) and uses \( dp[n]=-1 \) to indicate \( F(n) \) current. Has not been calculated yet. code show as below:

```c
int F(int n)
{
    int dp[50] = {0}; /* Initialize dp[50] to 0*/
    if(n == 0 || n == 1) return 1; /* If n=0 or n=1, return the result 1*/
    else{
        dp[0] = 1; /* When n>0 and 1, assign value to dp[0]*/
        dp[1] = 1; /* assigns value to dp[1]*/
        for(int i = 2; i <= n; i++) /* When i<=n, calculate and store the value of dp[i]*/
        {
            dp[i] = dp[i-1] + dp[i-2]; /* Calculate and store the value of dp[i]*/
        }
        return dp[n]; /*Returns dp[n]*/
    }
}
```

It has been calculated that kind of put the contents of records down, then when the next encounter the same problem, you can return the results directly, without double counting, so can save more than half of invalid calculations. As shown in Figure 2, this approach reduces the time complexity \( O(2^n) \) directly to the \( O(n) \) linear level, saving significant time.

### 2.2. The Advantages of Dynamic Programming

The above example can lead to the concept that if a problem can be broken down into several sub-problems and these sub-problems are repeated, the problem is called Overlapping Subproblems.\(^4\) Dynamic programming avoids a lot of double counting by directly recording the results of overlapping sub-problems so that the same sub-questions are used the next time. Therefore, a problem must have overlapping sub-problems in order to use dynamic programming to solve.

From the perspective of time complexity, dynamic programming can directly reduce the exponential level of time complexity \( O(2^n) \) to \( O(n) \) linear level, which greatly increases the speed of program running.\(^5\)

### 3. Classic Backpack Problem

#### 3.1. 01 Backpack

Problem Description: Given \( n \) items and a backpack. The weight of item \( i \) seems to be \( w_i \), its value is \( v_i \), and the capacity of the backpack is \( c \). Asked how to choose the items that are loaded into the backpack so that the total value of the items in the backpack is the largest?
3.1.1. Backtracking. The backtracking method is the violent enumeration method. Each item has two options: no choice. Therefore, for n items, there are $2^n$ options to choose from, and then choose the one with the highest value. The time complexity is $O(2^n)$. Code show as below:

```c
int f(int i, int cw, int cv){ /*i: The first few items cw: current weight cv: current value */
    if(cw == w || i == n){
        if(cv > MAX) MAX = cv; /* Update MAX*/
        return MAX;
    }
    if(cw + weight[i] <= w){
        f(i+1, cw + weight[i], cv+value[i]); /*Select this item*/
    }
    if(i+1, cw, cv);   /*Do not choose this item*/
    return MAX;
}
```

3.1.2. Dynamic Planning. If you use the backtracking method described above, the speed of the operation will increase exponentially as the item grows larger. To solve such problems we should use dynamic programming. As mentioned above, it is assumed that there are 5 items with weights of 20g, 20g, 40g, 60g and 30g respectively. The value of the items is $300, $400, $800, $900, $600, and the maximum capacity of the backpack is 90g. The solution to dynamic programming is as follows. First draw a grid and look at it line by line. Assume that the current line selects only the line and the previous item in the line. For example, line A can only select item A, and line C can select A, B, C three items, and then calculate the maximum value of each small backpack capacity can be loaded.

![backpack grid](image)

|     | 20g | 30g | 40g | 50g | 60g | 70g | 80g | 90g |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| A   | $300A$ | $300A$ | $300A$ | $300A$ | $300A$ | $300A$ | $300A$ | $300A$ |
| B   | $400B$ | $400B$ | $700A$ | $700A$ | $700A$ | $700A$ | $700A$ | $700A$ |
| C   | $400B$ | $400B$ | $800C$ | $800C$ | $1200$ | $1200$ | $1500$ | $1500$ |
| D   | $400B$ | $400B$ | $800C$ | $800C$ | $1200$ | $1200$ | $1500$ | $1500$ |
| E   | $400B$ | $600E$ | $800C$ | $1000$ | $1200$ | $1200$ | $1400$ | $1800$ |

So the value is the maximum value of the article A as shown above, rows A, because only A selectable. In line B, since B can be selected, B and A have the same weight, and B has a higher value. Therefore, when only the weight of B can be loaded, we choose B. When the capacity of the backpack is large enough, two items are selected. In line C, when the capacity of the backpack can be loaded 40g, there are two options. One is not to choose C. The current maximum value is $700 in cell[B][40g]$; the second is to select C, then the current maximum value is $800$, it is easy to judge that the second option is more valuable. Look at the last grid cell[E][90g]. If you don't choose E, the current maximum value is $1500$ in cell[D][90g]. If you choose E, because E weighs 30g, the current backpack is left 60g, while the maximum value of the backpack is 1200, plus the value of E, the total value is $1800$, obviously, the choice of E is more cost-effective.

From the above process, a judgment condition can be obtained, cell[i][j] = max(cell[i-1][j], cell[i-1][j+weight[i]]+value[i]) (weight[i] is the current item weight, value[i] is the current item value). The code looks like this:

```c
#include<algorithm>
using namespace std;
const int MAX = 100;

int main(){
    int n, W;  /*n: number of items W: capacity of the backpack */
    int w[MAX], v[MAX]; /*w[MAX]: the weight of each item v[MAX]: the value of each item*/
    int dp[MAX][MAX]; /*dp[MAX][MAX]: store the maximum value of each cell*/
    scanf("%d %d", &n, &W);
    /*Enter the number of items and backpack capacity*/
    for(int i = 1; i <= n; i++)
        scanf("%d", &w[i]);
    /*Initialize the weight of each item*/
    for(int i = 1; i <= n; i++)
        scanf("%d", &v[i]);
    /*Enter the weight of each item*/
    for(int i = 1; i <= n; i++)
        for(int j = 0; j <= W; j++)
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]]+v[i]);
    /*Calculate the maximum value of each cell*/
    int res = dp[n][W];
    printf("The maximum value is %d\n", res);
    return 0;
}
```
```c
for(int i = 1; i <= n; i++) {
    scanf("%d", &v[i]);
}
for(int j = 0; j <= W; j++) {
    dp[0][j] = 0;
}
for(int i = 1; i <= n; i++) {
    for(int j = w[i]; j <= W; j++) {
        dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]);
    }
}
printf("%d", dp[n][W]);
```

3.2. Full Backpack

The full backpack problem is very similar to the 01 backpack problem. The only difference is that the 01 backpack has only one item for each item, and the full backpack can take countless pieces for each item. Draw the following grid diagram using the above conditions:

**Table 2.** Full backpack grid.

|     | 20g | 30g | 40g | 50g | 60g | 70g | 80g | 90g |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| A   | $300A$ | $300A$ | $600A$ | $600A$ | $900A$ | $900A$ | $1200$ | $1200$ |
| B   | $400B$ | $400B$ | $800B$ | $800B$ | $1200$ | $1200$ | $1600$ | $1600$ |
| C   | $400B$ | $400B$ | $800C$ | $800C$ | $1200$ | $1200$ | $1600$ | $1600$ |
| D   | $400B$ | $400B$ | $800C$ | $800C$ | $1200$ | $1200$ | $1600$ | $1600$ |
| E   | $400B$ | $600E$ | $800C$ | $1000$ | $1200$ | $1400$ | $1600$ | $1800$ |

As shown above, in a cell[B][60g] as an example, this table is filled, there are two strategies, selected from the group B, and B is not selected. Hold time B, the cell[B][60g] = cell[A][60g], and the backpack 01 is the same; If you choose B, then the backpack 01 and the treatment is different, because each backpack 01 The item is only the same. If you do not select the item, it means that you must transfer to cell[A][60g-20g], and the full backpack can be placed in any item because of the item, so the state is transferred to cell[B][60g-20g].

Therefore, the state of the complete knapsack problem equation is shifted to cell[i][j] = max(cell[i-1][j],cell[i][j]+v[i]) For the current item weight,v[i] is the current item value).

4. Other Usage Scenarios for Dynamic Programming

Dynamic programming In addition to the Fibonacci sequence, there are many classic application scenarios, such as the number of towers, the maximum continuous subsequence, the longest increasing subsequence, and the longest common subsequence. The following table is a time complexity analysis of the violent and dynamic programming methods:

**Table 3.** Comparison table of time complexity between violent method and dynamic programming method.

| Algorithm                     | Abbreviation | Violence | Dynamic programming |
|-------------------------------|--------------|----------|---------------------|
| maximum continuous subsequence| MCS          | O(n²)    | O(n)                |
| longest increasing subsequence| LIS          | O(2ⁿ)    | O(2ⁿ)               |
| longest common subsequence    | LCS          | O(2ᵐⁿ⁺⁻) | O(nᵐ)               |
4.1. Maximum Continuous Subsequence Sum

The maximum continuous subsequence sum problem is described as: Given a sequence of numbers \(A_1, A_2, \ldots, A_n\), find \(i, j\) (\(1 \leq i \leq j \leq n\)), so that \(A_i + \ldots + A_j\) max, output this maximum sum. The idea of dynamic programming is to calculate a continuous sequence ending with \(A[i]\). The state transition equation is: \(dp[i] = \max(A[i], dp[i-1] + A[i])\) (\(dp[i]\) The maximum sum of consecutive sequences ending in \(A[i]\). The key parts of the code are as follows:

The above is the key code part of the violence law:

It can be seen from the comparison that the violent method uses a double loop, and the dynamic programming method can solve the single loop, which greatly improves the efficiency of the calculation.

4.2. Longest Increasing Subsequence

The longest non-decreasing subsequence problem is: in a sequence of numbers, find the longest subsequence (which can be discontinuous) so that the subsequence does not fall (non-decreasing). Let \(dp[i]\) denote the longest non-declining subsequence ending with \(A[i]\), and if there is an element \(A[j]\) before \(A[i]\) such that \(A[j] \leq A[i]\), then \(A[i]\) follows the longest non-declining subsequence ending with \(A[j]\). The state transition equation is: \(dp[i] = \max(1, dp[j]+1)\) (\(dp[i]\) is the largest non-declining subsequence ending with \(A[i]\)). The following are the key code segments:

The above is the key code part of the violence law:

It can be seen from the comparison that the violent method uses a double loop, and the dynamic programming method can solve the single loop, which greatly improves the efficiency of the calculation.

4.3. Longest Common Subsequence

The longest common subsequence problem is described as: Given two strings (or sequence of numbers) \(A\) and \(B\), find a string such that the string is the longest common part of \(A\) and \(B\) (the subsequences can be discontinuous). Suppose the string \(A\) is: sadstory and the string \(B\) is: adminsorry. Then the longest common subsequence is "adsory". The dynamic planning grid diagram is asTable 4. As shown in cell[2][3] above, both headers are d, so cell[2][3]=cell[1][2]+1, when the headers are not equal, such as cell[4][3], behavior i, listed as d, so their longest common subsequence cannot be extended, cell[i][j] inherits dp[i-1][j] and dp[i][j-1] The larger value in. When the state transition equation is: \(A[i]==B[j]\), \(dp[i][j]=dp[i-1][j-1]+1\), \(A[i]!=B[j]\), \(dp[i][j]=\max(dp[i-1][j], dp[i][j-1])\). The key code segments are as above.

5. Summary

Through the above classical model of dynamic programming, the regularity of dynamic programming can be summarized. The feature of dynamic programming is "one model and three features".
Table 4. The dynamic planning grid table

| s | a | d | s | t | o | r | y |
|---|---|---|---|---|---|---|---|
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| d | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| m | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| i | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| n | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| s | 1 | 1 | 2 | 3 | 3 | 3 | 3 |
| o | 1 | 1 | 2 | 3 | 3 | 4 | 4 |
| r | 1 | 1 | 2 | 3 | 4 | 5 | 5 |
| r | 1 | 1 | 2 | 3 | 4 | 5 | 5 |
| y | 1 | 1 | 2 | 3 | 3 | 4 | 5 |

"One model" refers to a model that dynamically plans the problem that is best suited to solve. Dynamic programming is generally used to solve optimal problems. In the process of solving problems, it is necessary to go through multiple decision stages. Each phase corresponds to a set of states. Then look for a set of decision sequences through which the optimal values of the final desired solution can be generated.

The "three characteristics" are the optimal substructure, no aftereffect and repeat subproblems. The optimal substructure means that the optimal solution of the problem contains the optimal solution of the subproblem. No post-effect has two meanings. The first meaning is that when deriving the state of the later stage, it only cares about the state value of the previous stage, and does not care how this state is derived step by step; the second meaning is that Once the stage status is determined, it is not affected by the decision of the later stage. A repeating subproblem means that when different decision sequences arrive at a certain stage, a repeating state may occur.

Problems that meet the above characteristics can generally be solved by the idea of dynamic programming.

Acknowledgments
This work is supported by the Key Disciplines of Computer Science and Technology of Shanghai Polytechnic University under Grant No. XXKZD1604.

References
[1] Wang Zheng. Dynamic programming theory [EB/OL].[2018-12-31]. Https://time.geekbang.org/column/article/75702.
[2] Wenrong Jiang. Research on Existential Problems Based onDrawer Principle[J]. 2019 International Conference on Mathematics, Big Data Analysis and Simulation and Modeling (MBDASM2019), Changsha, China, August 30-31, 2019. 4-7.
[3] Wenrong Jiang. Analysis and Discussion of Radar Construction Problems with Greedy Algorithm[J]. 3rd International Conference on Computer, Communication and Computational Sciences (IC4S2018), Bangkok, Thailand, Springer. October 20, 2018. 303-312.
[4] Wenrong Jiang, Jian Chen, Wei Tao. The Development of Automotive Interior Sales Website[J]. Communications in Computer and Information Science (Volumes 268), 2011(12):342-348. EI:20122015019844
[5] Wenrong Jiang, Shijie He. The research of small and medium-sized wedding photography enterprise’s product types and consumer participation behavior[C]. Proceedings of the 2016 International Conference on Social Science, Humanities and Modern Education(SSHME 2016), Guangzhou, China. Atlantis Press. May 30, 2016, 103-107.WOS:000390400800020.