Find More Coins

G23

闵思韬 万园 陈思雅

Date: 2016-04-17

Chapter 1: Introduction

Problem description,

Eva loves to collect coins from all over the universe, including some other planets like Mars. One day she visited a universal shopping mall which could accept all kinds of coins as payments. However, there was a special requirement of the payment: for each bill, she must pay the exact amount. Since she has as many as 10^4 coins with her, she definitely needs your help. You are supposed to tell her, for any given amount of money, whether or not she can find some coins to pay for it.

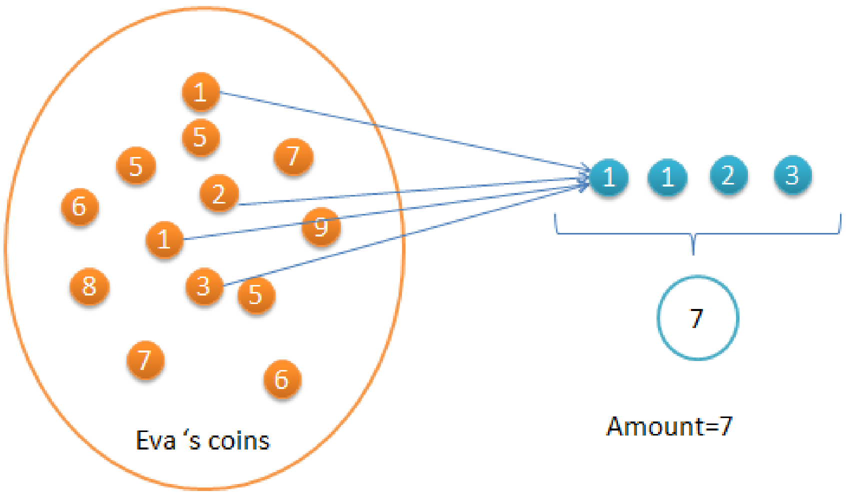
- **Input Specification:**

Each input file contains one test case. For each case, the first line contains 2 positive numbers: N (the total number of coins) and M(the amount of money Eva has to pay). The second line contains N face values of the coins, which are all positive numbers. All the numbers in a line are separated by a space.

- **Output Specification:**

For each test case, print in one line the face values V_1 V_2 ... V_k such that $V_1+ V_2 + ... +V_k = M$. All the numbers must be separated by a space, and there must be no extra space at the end of the line. If such a solution is not unique, output the smallest sequence. If there is no solution, output "No Solution" instead. Note: sequence {A[1], A[2], ...} is said to be "smaller" than sequence {B[1],B[2], ...} if there exists k1 such that A[i]=B[i] for all i < k, and A[k] < B[k].

We draw a picture to show what this problem means.



This problem can be easy to classified as 0-1 Knapsack problem. The knapsack problem is a problem in combinatorial optimization, which can be descripted as follow: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. **In the problem of our project, the amount of money equals to knapsack size and the coins eva has equal to the items which would be added to the knapsack and coins value equals to items value. And if the best collection of items found in the knapsack problem equal to the amount of money, it means we can find the coins to pay the money, otherwise we should output "No Solution".**

Purpose of Report

In our report, we try to f**ind some solutions (algorithm)** to the 0-1 knapsack problem and **compare their time complexity and space complexity** so that we can have **a deeper understanding of algorithm** we learned and we can **understand the advantages and shortcomings of these algorithms** which can make us to find a more appropriate algorithm when we meet some problems in our future works and researches.

Possible Solution and Algorithm Background

Several algorithms are available to solve this problem, based on dynamic programming approach, approach

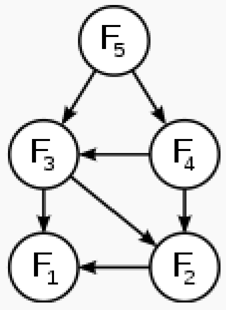
1. Dynamic programming

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler **sub-problems** in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively; Bellman called this the "Principle of Optimality". Likewise, in computer science, a problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have optimal substructure.

If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems.

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and **overlapping sub-problems.**

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion. **Overlapping sub-problems** means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems. For example, consider the recursive formulation for generating the Fibonacci series: Fi=Fi−1+Fi−2, with base case F1 = F2 = 1. Then F43=F42+F41, andF42=F41+F40. NowF41 is being solved in the recursive sub-trees of both F43 as well as F42. Even though the total number of sub-problems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once.



The sub problem graph for the Fibonacci sequence

Method of DP:

- **Top-down approach:** This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it direct, otherwise we solve the sub-problem and add its solution to the table.
- **Bottom-up approach:** Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F41 and F40, we can directly calculate the value of F42.

In our project, we use DP algorithm to solve the problem.

2. Backtracking

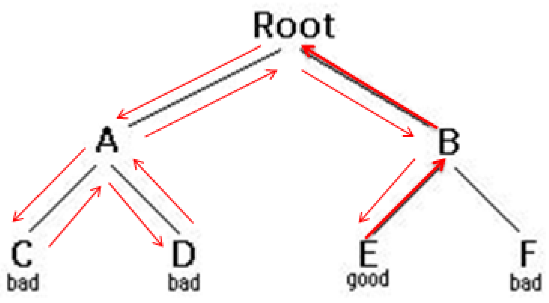
Backtracking is a form of recursion.

The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

It can be described as, conceptually, you start at the root of a tree; the tree probably has some good leaves and some bad leaves, though it may be that the leaves are all good or all bad. You want to get to a good leaf. At each node, beginning with the root, you choose one of its children to move to, and you keep this up until you get to a leaf.

Suppose you get to a bad leaf. You can backtrack to continue the search for a good leaf by revoking your most recent choice, and trying out the next option in that set of options. If you run out of options, revoke the choice that got you here, and try another choice at that node. If you end up at the root with no options left, there are no good leaves to be found.

This needs an example.



In this example we drew a picture of a tree. The tree is an abstract model of the possible sequences of choices we could make. And the step of finding a good answer is below:

1. Starting at Root, your options are A and B. You choose A.
2. At A, your options are C and D. You choose C.
3. C is bad. Go back to A.
4. At A, you have already tried C, and it failed. Try D.
5. D is bad. Go back to A.
6. At A, you have no options left to try. Go back to Root.
7. At Root, you have already tried A. Try B.
8. At B, your options are E and F. Try E.
9. E is good. Congratulations!

In our project, we can use backtracking to solve the problem, we can regard each coin as a choice, and draw a tree of coins. Then we use dsp to traverse the tree to find the good choice (correct answer) of the problem. When we find the answer, can we recursively from leaf return to the root, and the answer is one path of the coins tree.

Chapter 2: Data Structure / Algorithm Specification

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures. Or, if you are to introduce a new data structure and its related operations, do it in this chapter.

1. Dynamic programming

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures.
Or, if you are to introduce a new data structure and its related operations, do it in this chapter.

The core algorithm in this project is Dynamic Programming. Dynamic programming solves problem by combining the solutions to subproblems. It applies when the subproblems

share subprobems, and it solves each subprobles just once and then save its answer in a table, thereby avoiding the work of recomputing the answer.
Train of thought:
1. The problem is consisted of two question: Can Eva pay M with N coins and no change? If she can, what is the smallest sequence of the coins Eva can pay? Now we are trying to answer the questions.

The first question is a typical “Knapsack problem”, so we can think about this question by the regular way of solving “Knapsack problem”, that is how much is the maximum but not more than M we can pay using N coins? And if the amount is equivalent to M, we find the answer of the firt question. The recurrence formula is

$$F(n,m)=\max\{F(n-1,m),F(n-1,m-v(n))+v(n)\}$$

F(n,m): the maximum but not more than M one can pay with N coins
v(n): the value of the nth coin
The formula means that to know F(n,m), we can compare the amount using the nth coin and the amount without using the nth coin. The larger one is the answer.
2. According to the way of thinking, we found a simpler way without comparing, that is, we don’t need to calculate F(n,m) and judge that whether F(N,M) is equivalent to M, and we use Can(N,M) to show whether we can acquire the requirement of m with n coins directly. If Can(n-1,m-v(n)) is true, than with the nth coin, Can(n,m) is true as well. If (n-1) coins cannot acquire (m-v(n)), the nth coin become useless because we can not replace a coin by the nth coin to acquire m, so whether Can(n,m) is true depends on whether Can(n-1,m) is true.

```
if Can(n-1,m-v(n)) is true then
    Can(n,m) is true
else
    Can(n, m) ← Can(n-1,m)
End if
```

3. The algorithm always answered the first question, now we try to find the smallest sequence and record it so that we can output it. Actually, we have already known that whether the nth will be used when we use n coins to acquire m, so we just need to add a flag to the nth coin after judgement, so that we can know whether we should output the nth coin according to the flag when we output the answers.

```
if Can(n-1,m-v(n)) is true then
    Can(n,m) ← true and flag(n,m) ← true
End if
```

To get the smallest sequence, we should sort the coins first. Knowing that the algorithm tends to replace the (n-1)th coin by the nth coin if it is reasonable, so the sorting should be descending, than the smaller coin will be prior to be used. Finally we can get the smallest sequence after computing.
4. So far, we totally solved the problem with the way of recurrence，however, the process of recurrence contains a number of repeater calls(jutiyidian). Now we are going to try to

optimize the algorithm using Dynamic program
ming.(chushihua de maodun)

If there is a table that records the consequence of every call, then we just need check the table to find the value we need. So we construct a two-dimensional array named can[i][j] to record whether i coins can acquire lm. We can determine the value of can[i][j] according to the value we determined previously, and after a nested loop, we will fill the table. The last value of the table which is can[N][M] can tell us the answer. In addition, record the information of flag[n][m] in is[i][j].

```
for i from 1 to n
    for j from coins[i] to m
        if can[i - 1][j - coins[i]] is true then
            can[i][j] ← true;
            is[i][j] ← true;
        else
        can[i][j] ← can[i - 1][j];
        End if
    End for
End for
```

5. We can output answers according to is[i][j]. First, check is[N][M], this is reasonable, for we want the smallest sequence and the nth coin is the smallest coin. If it is true, we can output it and then check is[N-1][M-v(N)], because now we have N-1 coins and we want to pay M, so what we need to do is as the same as the last time: check whether the smallest coin should be used. If is[N][M] is faulse, we can turn to is[N-1][M] to check whether the (n-1)th coin will be used.

```
while n > 0)
    if is[n][m] is true then
        Print coins[n]
        m ← m - coins[n];
        n ← n - 1
    else
        n ← n - 1
    End if
End while
```

To explain the algorithm clearly, an example may be useful. Assume that Eva needs to pay 8 with 4 coins:1,2,8,5. As said, we sort the coins in descending order first, so now we have 4 coins in order: 8,5,2,1.
can[i][0] has been initialized with 1, because 0 can be acquired with any number of coins, and this is the fundament of the following works. When i = 1, j = coins[1] = 8, because can[i-1][j-coins[i]] = can[0][0] = 1, so can[1][8] = 1, and the 1th coin will be used when we need to acquire 8. After computing, two arrays are both filled with true or false.

```
e.g.    INPUT: 4 8
                   1 2 8 5
        After sorted: 8 5 2 1
        can[i][j]: 1 0 0 0 0 0 0 0 1
                   1 0 0 0 0 1 0 0 1
                   1 0 1 0 0 1 0 1 1
                   1 1 1 1 0 1 1 1 1
        is[i][j]:  0 0 0 0 0 0 0 0 0
                   0 0 0 0 0 1 0 0 0
                   0 0 1 0 0 0 0 1 0
                   0 1 0 1 0 0 1 0 1
```

Now we can find which coins should be output according to the values of is[i][j]. is[4][8] is true, so the 4th coin which is 1 should be output, then we turn to i[3][7] and it is also true, so 2 should be output. Finally we can find all the coins should be output.

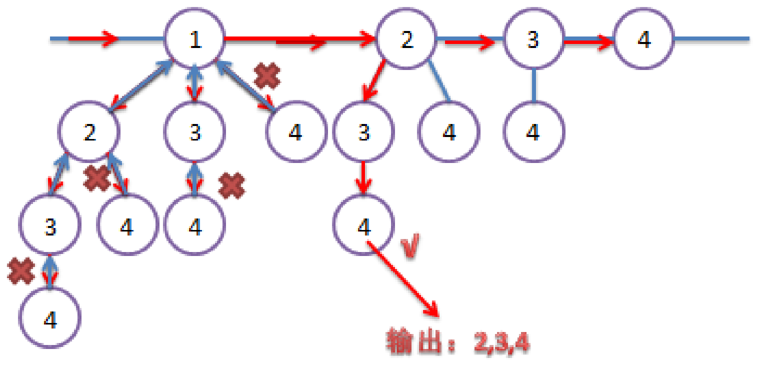
2. Backtracking

We can use backtracking to solve this problem. Backtracking is a recursive method to traverse all possible sequence of the coins, and if we find the sum of some coins equals to the amount of the money, we return the each coins level by level. Finally we can find the correct answer. The step is as below:

```
if Can(n-1,m-v(n)) is true then
    Can(n,m) is true
else
    Can(n, m) ← Can(n-1,m)
End if
```

1. sort all coins and make them placed from small to big.
2. Do the recursion to find the correct answer, when we find wrong answer we return the former level of the recursion and find another path to try and do these again and again until we get the true answer.
3. When we find the right coin, we store it in a array and return former level of recursion.
4. When return to the root all coins whose sum is m are all stored in the array, then we can output the right answer.

Use coins{1,2,3,4}, m=8 as an example:We show the step of the backtracking in figure, as below:



The right arrow is the step of recursion, blue arrow shows that when we get wrong answer we backtrack to the former level of recursion.

Chapter 3: Testing Results

We upload the program to the pta and pat, and the program is passing the test on the PAT, so our program is right.

	Iuput	purpose	output	result
1	8 9 5 9 8 7 2 3 4 1	Sample have solution	1 3 5	*corrected*
2	4 8 7 2 4 3	Sample No solution	No Solution	*corrected*
3	4 100 25 25 25 25	All coins with the same value	25 25 25 25	*corrected*
4	5 2 10 9 20 3 2	The answer coin is in the end	2	*corrected*
5	1 100 20	One coin	No Solution	*corrected*
6	5 15 1 2 3 4 5	All coins are used	1 2 3 4 5	*corrected*
7	100 100 1 1 1 1 1 1 ……1	M=100 critial condition	1 1 1 1 1 1 …1	*corrected*
8	pass	Large amount of data	pass	*corrected*

Chapter 4: Analysis and Comments

Theoretical time and space complexity:

1. Dynamic programming

Because we use a m*n matrix to store the flag, and we traverse the matrix when we do the algorithm, so the worst time complexity is **O(n*m),** space complexity is also **O(n*m).**

```
for i from 1 to n
    for j from coins[i] to m
        if can[i - 1][j - coins[i]] is true then
            can[i][j] ← true;
            is[i][j] ← true;
        else
        can[i][j] ← can[i - 1][j];
        End if
    End for
End for
```

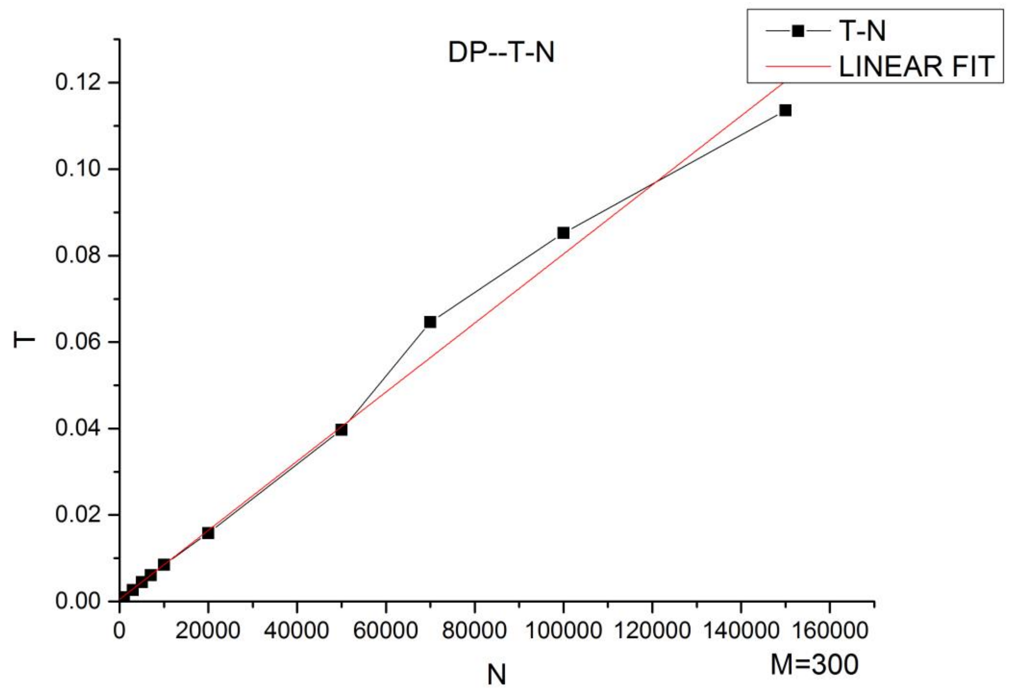
	1	2	3	4	5	……	M
1	0	1	...	...	...	...	...
2	...	...	...	...	...	...	...
3	...	...	...	...	...	...	...
4	...	...	...	...	...	...	...
....	...	...	...	...	...	...	...
n	...	...	...	...	...	...	...

2. Backtracking

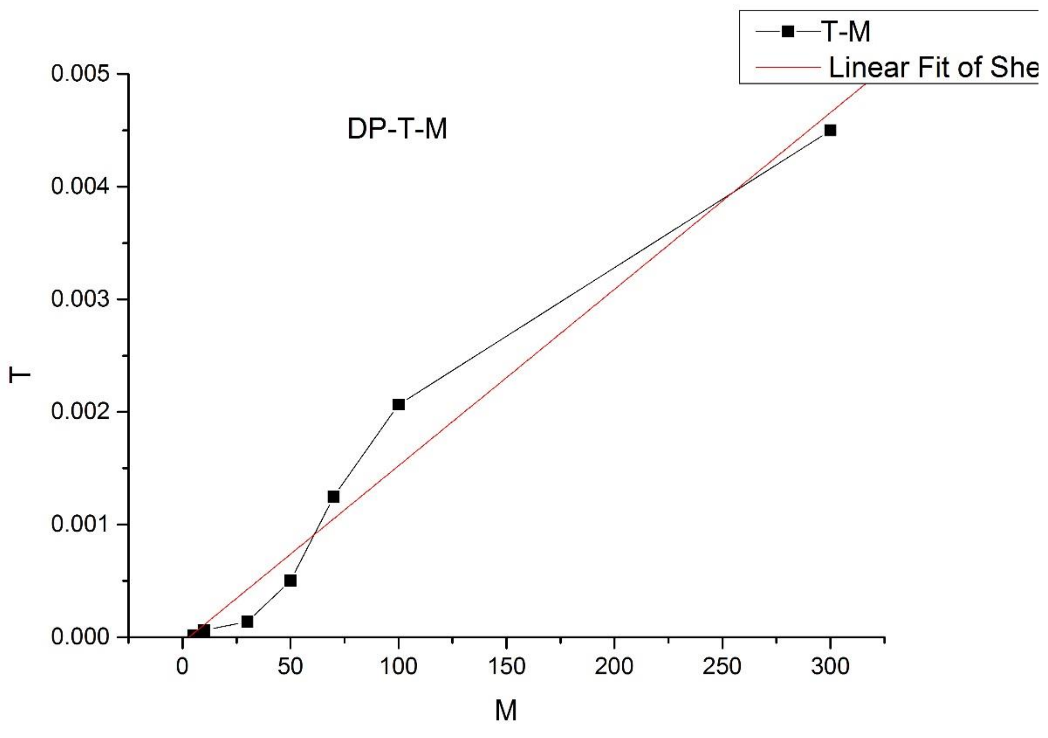
The recursion express is $T(n)=T(n-1)+T(n-2)+T(n-2)+T(n-3).....+T(1)$, so
$T(n-1)=T(n-2)+T(n-2)+T(n-3)+T(n-4)+ .....+T(1)$, and
$T(n-2)=T(n-3)+T(n-4)+T(n-5)+T(n-6)+ .....+T(1)$ …..
After substituting the later formulas into the first formula, we get $T(n)=(2^n)*T(1)$.
So the **worst time complexity of backtracking algorithm** is **$O(2^n)$.** **Space complexity is O(n)**

Testing

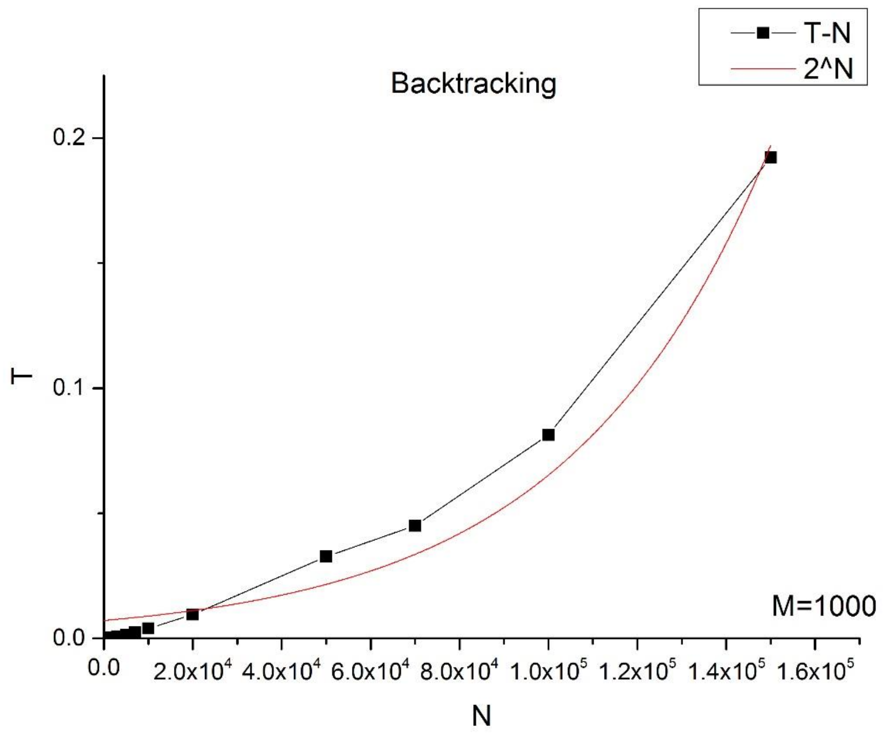
Here is our testing result: you can see that all time complexity is same with our analysis.



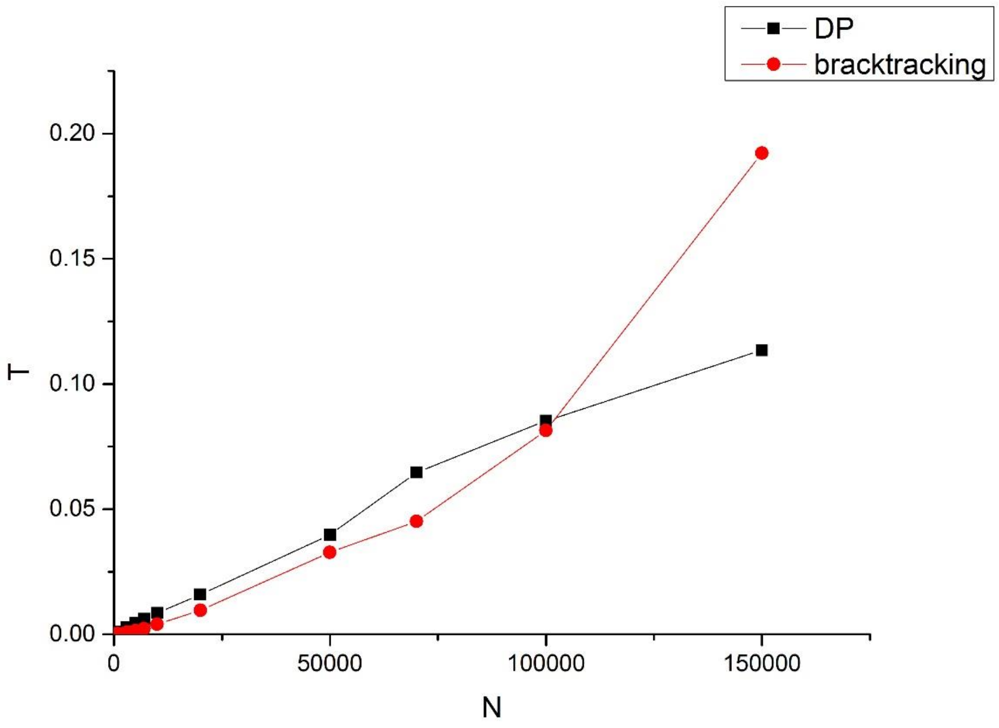
Dynamic programing T-n



Dynamic programming T-m



Recursion solution T-N



Compare DP and recursion solution

From the charts of time used, we can see that the time which the dynamic programming used is obviously less than backtracking, and as the amount we tested increases, the gap is even lager. And the results of two tests are both in consistency with our expectation about their time complexities. The datas have a little fluctuations because we used random numbers to test, and some numbers may be easier or more difficult to find the coins.

Appendix: Source Code (if required)

Dynamic Programming:

```
#include<stdio.h>
#include<stdlib.h>
/*
compare: 降序排列所需函数
*/
int compare(const void *a, const void *b)
{
    return *(int*)b - *(int*)a;
}
/*数组较大，放在主函数栈外以免溢出*/
/*
coins:存放给出的硬币
ans:存放答案
*/
int coins[10001], ans[10001];
/*
can[i][lm]:记录前i个硬币能否凑成lm元
is[i][lm]:记录第i个硬币有没有被用于凑成lm元
*/
bool can[10001][101], is[10001][101];
int main(){
    /*can和is中用到的i和lm*/
    int i, lm;
    /*Eva所带硬币的数量n,目标金额m*/
    int n, m;
    /*输出答案中用于for循环的变量k*/
    int k;
    /*写入所给硬币数量、目标金额和各枚硬币的金额*/
    scanf("%d %d", &n, &m);
    for (i = 1; i <= n; i++){
        scanf("%d", &coins[i]);
    }
    /*为了得到最小字典序的答案，对给出的硬币序列进行降序排列*/
    qsort(coins + 1, n, sizeof(int), compare);
    /*数组初始化为0*/
    can[10001][101] = { 0 };
    is[10001][101] = { 0 };
    /*目标金额为0时，前i个硬币总能满足*/
    for (i = 0; i <= n; i++){
        can[i][0] = 1;
    }
    /*
    判断前i个硬币能否凑出lm元：
    如果前i-1个硬币可以凑出lm-第i枚硬币的金额，那么前i个硬币可以凑出lm元，而且
第i个硬币被征用了；
    如果不能，则前i个硬币能否凑出lm元取决于前i-1个硬币能否直接凑出m元。
    每次判断的结果都存放在can和is数组中
    */
    for (i = 1; i <= n; i++){
        for (lm = coins[i]; lm <= m; lm++){
            if (can[i - 1][lm - coins[i]]){
                can[i][lm] = 1;
                is[i][lm] = 1;
            }
            else{
                can[i][lm] = can[i - 1][lm];
            }
        }
    }
    /*
    如果给出n个硬币可以凑出m元，输出答案。
    */
    if (can[n][m]){
        int total = 0;
        while (n > 0){
            if (is[n][m]){
                ans[total] = coins[n];
                m = m - coins[n];
                n --;
                total++;
            }
            else{
                n--;
            }
        }
        for (k = 0; k < total - 1; k++){
            printf("%d ", ans[k]);
        }
        printf("%d\n", ans[total - 1]);
    }
    else{
        printf("No Solution\n");
    }
    return 0;
}
```

Backtracking:

```
#include <iostream>
#include <algorithm>
using namespace std;
int N, M;                            //N-the total number of coins, M-the amount
of money Eva has to pay
int *ValueOfCoins = new int[N];    //ValueOfCoins--store the value of all
coins eva has
int OutPutPoint = 0;                //OutPutPoint--to record the outputarray
head
/*check whether eva can pay the money and record coins eva uses to pay*/
bool calCoins(int n, int m, int *output){
    bool flag;
    if (n == 0){
        if (m == 0)
            return true;
        else
            return false;
    }
    else{

        if (m < 0)  return false;
        else if (m == 0) return true;
        else{
            for (int i = N - n; i < N; i++){

                flag = calCoins(N - i - 1, m - ValueOfCoins[i], output);
                if (flag == true){
                    output[OutPutPoint++] = ValueOfCoins[i];
                    return flag;
                }
            }
            return flag;
        }
    }
}

int main(void){
    cin >> N >> M;
    int *output = new int[N];     //output array
    bool CanPay;                 //eva can pay the money
    int j = 1;
    int sum = 0;
    /*InPut*/
    for (int i = 0; i < N; i++)
        cin >> ValueOfCoins[i];
    /*sort the coins eva has*/
    sort(ValueOfCoins, ValueOfCoins + N);
    //for(int i=0;i<N;i++)
    //cout<<ValueOfCoins[i]<<' ';
    //cout<<endl;
    /*check whether eva can pay the money and record coins eva uses to pay*/
    CanPay = calCoins(N, M, output);


    /*OutPut*/
    if (CanPay == true){
        for (int i = OutPutPoint - 1; i > 0; i--){
            sum += output[i];
            cout << output[i] << ' ';
            j = i;
        }
        cout << output[j - 1] << endl;
        sum += output[j - 1];
        //cout<<sum<<endl;
    }
    else{
        cout << "No Solution" << endl;
    }
    return 0;
}
```

Author List

Specify ***who did what*** to show that particular contributors deserve to have their names printed in the cover page of your report.

Programming: Write the program **Wan Yuan & Min Sitao**

Testing: Provide a set of test cases to fill in a test report. **Chen Siya**

Documentation: a complete report. **Wan Yuan & Min Sitao**

Declaration

We hereby declare that all the work done in this project titled "XXX" is of our independent effort as a group.

Signatures

闵思韬

万园

陈思雅

2016.4.18