Dynamic Programming

6

Dynamic programming is an algorithm design technique that can be used to find optimal solutions to problems and to count the number of solutions. This chapter is an introduction to dynamic programming, and the technique will be used many times later in the book when designing algorithms.

Section 6.1 discusses the basic elements of dynamic programming in the context of a coin change problem. In this problem we are given a set of coin values and our task is to construct a sum of money using as few coins as possible. There is a simple greedy algorithm for the problem, but as we will see, it does not always produce an optimal solution. However, using dynamic programming, we can create an efficient algorithm that always finds an optimal solution.

Section 6.2 presents a selection of problems that show some of the possibilities of dynamic programming. The problems include determining the longest increasing subsequence in an array, finding an optimal path in a two-dimensional grid, and generating all possible weight sums in a knapsack problem.

6.1 Basic Concepts

In this section, we go through the basic concepts of dynamic programming in the context of a coin change problem. First we present a greedy algorithm for the problem, which does not always produce an optimal solution. After this, we show how the problem can be efficiently solved using dynamic programming.

6.1.1 When Greedy Fails

Suppose that we are given a set of coin values $\texttt{coins} = \{c_1, c_2, \ldots, c_k\}$ and a target sum of money n, and we are asked to construct the sum n using as few coins as

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_6

possible. There are no restrictions on how many times we can use each coin value. For example, if $\texttt{coins} = \{1, 2, 5\}$ and $n = 12$, the optimal solution is $5 + 5 + 2 = 12$, which requires three coins.

There is a natural greedy algorithm for solving the problem: always select the largest possible coin so that the sum of coin values does not exceed the target sum. For example, if $n = 12$, we first select two coins of value 5, and then one coin of value 2, which completes the solution. This looks like a reasonable strategy, but is it always optimal?

It turns out that this strategy does *not* always work. For example, if $\texttt{coins} = \{1, 3, 4\}$ and $n = 6$, the optimal solution has only two coins ($3 + 3 = 6$) but the greedy strategy produces a solution with three coins ($4 + 1 + 1 = 6$). This simple counterexample shows that the greedy algorithm is not correct.[1]

How could we solve the problem, then? Of course, we could try to find another greedy algorithm, but there are no other obvious strategies that we could consider. Another possibility would be to create a brute force algorithm that goes through all possible ways to select coins. Such an algorithm would surely give correct results, but it would be very slow on large inputs.

However, using dynamic programming, we can create an algorithm that is almost like a brute force algorithm but it is also *efficient*. Thus, we can both be sure that the algorithm is correct and use it for processing large inputs. Furthermore, we can use the same technique for solving a large number of other problems.

6.1.2 Finding an Optimal Solution

To use dynamic programming, we should formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is to calculate values of a function $\texttt{solve}(x)$: what is the minimum number of coins required to form a sum x? Clearly, the values of the function depend on the values of the coins. For example, if $\texttt{coins} = \{1, 3, 4\}$, the first values of the function are as follows:

$$\begin{aligned}
\texttt{solve}(0) &= 0 \\
\texttt{solve}(1) &= 1 \\
\texttt{solve}(2) &= 2 \\
\texttt{solve}(3) &= 1 \\
\texttt{solve}(4) &= 1 \\
\texttt{solve}(5) &= 2 \\
\texttt{solve}(6) &= 2 \\
\texttt{solve}(7) &= 2 \\
\texttt{solve}(8) &= 2 \\
\texttt{solve}(9) &= 3 \\
\texttt{solve}(10) &= 3
\end{aligned}$$

[1] It is an interesting question when exactly does the greedy algorithm work. Pearson [24] describes an efficient algorithm for testing this.

For example, $\texttt{solve}(10) = 3$, because at least 3 coins are needed to form the sum 10. The optimal solution is $3 + 3 + 4 = 10$.

The essential property of `solve` is that its values can be recursively calculated from its smaller values. The idea is to focus on the *first* coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following recursive formula to calculate the minimum number of coins:

$$\begin{aligned}\texttt{solve}(x) = \min(&\texttt{solve}(x-1)+1,\\ &\texttt{solve}(x-3)+1,\\ &\texttt{solve}(x-4)+1).\end{aligned}$$

The base case of the recursion is $\texttt{solve}(0) = 0$, because no coins are needed to form an empty sum. For example,

$$\texttt{solve}(10) = \texttt{solve}(7)+1 = \texttt{solve}(4)+2 = \texttt{solve}(0)+3 = 3.$$

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum x:

$$\texttt{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \texttt{coins}} \texttt{solve}(x-c)+1 & x > 0 \end{cases}$$

First, if $x < 0$, the value is infinite, because it is impossible to form a negative sum of money. Then, if $x = 0$, the value is zero, because no coins are needed to form an empty sum. Finally, if $x > 0$, the variable c goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++ (the constant `INF` denotes infinity):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Still, this function is not efficient, because there may be a large number of ways to construct the sum and the function checks all of them. Fortunately, it turns out that there is a simple way to make the function efficient.

Memoization The key idea in dynamic programming is *memoization*, which means that we store each function value in an array directly after calculating it. Then, when the value is needed again, it can be retrieved from the array without recursive calls. To do this, we create arrays

```
bool ready[N];
int value[N];
```

where `ready`[x] indicates whether the value of `solve`(x) has been calculated, and if it is, `value`[x] contains this value. The constant N has been chosen so that all required values fit in the arrays.

After this, the function can be efficiently implemented as follows:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    ready[x] = true;
    value[x] = best;
    return best;
}
```

The function handles the base cases $x < 0$ and $x = 0$ as previously. Then it checks from `ready`[x] if `solve`(x) has already been stored in `value`[x], and if it is, the function directly returns it. Otherwise the function calculates the value of `solve`(x) recursively and stores it in `value`[x].

This function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of `solve`(x) has been stored in `value`[x], it can be efficiently retrieved whenever the function will be called again with the parameter x. The time complexity of the algorithm is $O(nk)$, where n is the target sum and k is the number of coins.

Iterative Implementation Note that we can also *iteratively* construct the array `value` using a loop as follows:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

In fact, most competitive programmers prefer this implementation, because it is shorter and has smaller constant factors. From now on, we also use iterative implementations in our examples. Still, it is often easier to think about dynamic programming solutions in terms of recursive functions.

Constructing a Solution Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. To construct an optimal solution in our coin problem, we can declare a new array that indicates for each sum of money the first coin in an optimal solution:

```
int first[N];
```

Then, we can modify the algorithm as follows:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

After this, the following code prints the coins that appear in an optimal solution for the sum n:

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

6.1.3 Counting Solutions

Let us now consider another variant of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if $\texttt{coins} = \{1, 3, 4\}$ and $x = 5$, there are a total of 6 ways:

- $1+1+1+1+1$
- $1+1+3$
- $1+3+1$
- $3+1+1$
- $1+4$
- $4+1$

Again, we can solve the problem recursively. Let $\texttt{solve}(x)$ denote the number of ways we can form the sum x. For example, if $\texttt{coins} = \{1, 3, 4\}$, then $\texttt{solve}(5) = 6$ and the recursive formula is

$$\begin{aligned}\texttt{solve}(x) = &\texttt{solve}(x-1)+\\ &\texttt{solve}(x-3)+\\ &\texttt{solve}(x-4).\end{aligned}$$

Then, the general recursive function is as follows:

$$\texttt{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \texttt{coins}} \texttt{solve}(x-c) & x > 0 \end{cases}$$

If $x < 0$, the value is zero, because there are no solutions. If $x = 0$, the value is one, because there is only one way to form an empty sum. Otherwise we calculate the sum of all values of the form $\texttt{solve}(x-c)$ where c is in `coins`.

The following code constructs an array `count` such that `count`$[x]$ equals the value of $\texttt{solve}(x)$ for $0 \leq x \leq n$:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x-c >= 0) {
            count[x] += count[x-c];
        }
    }
}
```

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo m where, for example, $m = 10^9 + 7$. This can be done by changing the code so that all calculations are done modulo m. In the above code, it suffices to add the line

```
        count[x] %= m;
```

after the line

```
        count[x] += count[x-c];
```

6.2 Further Examples

After having discussed the basic concepts of dynamic programming, we are now ready to go through a set of problems that can be efficiently solved using dynamic programming. As we will see, dynamic programming is a versatile technique that has many applications in algorithm design.

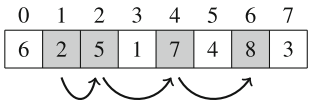

Fig. 6.1 The longest increasing subsequence of this array is [2, 5, 7, 8]

6.2.1 Longest Increasing Subsequence

The *longest increasing subsequence* in an array of n elements is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, Fig. 6.1 shows the longest increasing subsequence in an array of eight elements.

We can efficiently find the longest increasing subsequence in an array using dynamic programming. Let $\texttt{length}(k)$ denote the length of the longest increasing subsequence that ends at position k. Then, if we calculate all values of $\texttt{length}(k)$ where $0 \leq k \leq n-1$, we will find out the length of the longest increasing subsequence. The values of the function for our example array are as follows:

$$\begin{aligned}
\texttt{length}(0) &= 1 \\
\texttt{length}(1) &= 1 \\
\texttt{length}(2) &= 2 \\
\texttt{length}(3) &= 1 \\
\texttt{length}(4) &= 3 \\
\texttt{length}(5) &= 2 \\
\texttt{length}(6) &= 4 \\
\texttt{length}(7) &= 2
\end{aligned}$$

For example, $\texttt{length}(6) = 4$, because the longest increasing subsequence that ends at position 6 consists of 4 elements.

To calculate a value of $\texttt{length}(k)$, we should find a position $i < k$ for which $\texttt{array}[i] < \texttt{array}[k]$ and $\texttt{length}(i)$ is as large as possible. Then we know that $\texttt{length}(k) = \texttt{length}(i)+1$, because this is an optimal way to append $\texttt{array}[k]$ to a subsequence. However, if there is no such position i, then $\texttt{length}(k) = 1$, which means that the subsequence only contains $\texttt{array}[k]$.

Since all values of the function can be calculated from its smaller values, we can use dynamic programming to calculate the values. In the following code, the values of the function will be stored in an array `length`.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k],length[i]+1);
        }
    }
}
```

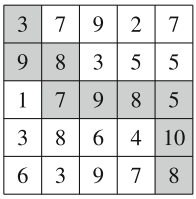

Fig. 6.2 An optimal path from the upper-left corner to the lower-right corner

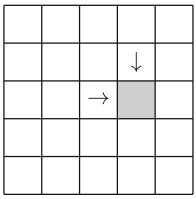

Fig. 6.3 Two possible ways to reach a square on a path

The resulting algorithm clearly works in $O(n^2)$ time.[2]

6.2.2 Paths in a Grid

Our next problem is to find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, with the restriction that we may only move down and right. Each square contains an integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

As an example, Fig. 6.2 shows an optimal path in a 5×5 grid. The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Assume that the rows and columns of the grid are numbered from 1 to n, and $\texttt{value}[y][x]$ equals the value of square (y, x). Let $\texttt{sum}(y, x)$ denote the maximum sum on a path from the upper-left corner to square (y, x). Then, $\texttt{sum}(n, n)$ tells us the maximum sum from the upper-left corner to the lower-right corner. For example, in the above grid, $\texttt{sum}(5, 5) = 67$. Now we can use the formula

$$\texttt{sum}(y, x) = \max(\texttt{sum}(y, x-1), \texttt{sum}(y-1, x)) + \texttt{value}[y][x],$$

which is based on the observation that a path that ends at square (y, x) can come either from square $(y, x-1)$ or from square $(y-1, x)$ (Fig. 6.3). Thus, we select the direction that maximizes the sum. We assume that $\texttt{sum}(y, x) = 0$ if $y = 0$ or $x = 0$, so the recursive formula also works for leftmost and topmost squares.

Since the function sum has two parameters, the dynamic programming array also has two dimensions. For example, we can use an array

[2] In this problem, it is also possible to calculate the dynamic programming values more efficiently in $O(n \log n)$ time. Can you find a way to do this?

```
int sum[N][N];
```

and calculate the sums as follows:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

The time complexity of the algorithm is $O(n^2)$.

6.2.3 Knapsack Problems

The term *knapsack* refers to problems where a set of objects is given, and subsets with some properties have to be found. Knapsack problems can often be solved using dynamic programming.

In this section, we focus on the following problem: Given a list of weights $[w_1, w_2, \ldots, w_n]$, determine all sums that can be constructed using the weights. For example, Fig. 6.4 shows the possible sums for weights [1, 3, 3, 5]. In this case, all sums between $0 \ldots 12$ are possible, except 2 and 10. For example, the sum 7 is possible because we can choose the weights [1, 3, 3].

To solve the problem, we focus on subproblems where we only use the first k weights to construct sums. Let $\texttt{possible}(x, k) = \texttt{true}$ if we can construct a sum x using the first k weights, and otherwise $\texttt{possible}(x, k) = \texttt{false}$. The values of the function can be recursively calculated using the formula

$$\texttt{possible}(x, k) = \texttt{possible}(x - w_k, k - 1) \text{ or } \texttt{possible}(x, k - 1),$$

which is based on the fact that we can either use or not use the weight w_k in the sum. If we use w_k, the remaining task is to form the sum $x - w_k$ using the first $k - 1$ weights, and if we do not use w_k, the remaining task is to form the sum x using the first $k - 1$ weights. The base cases are

$$\texttt{possible}(x, 0) = \begin{cases} \texttt{true} & x = 0 \\ \texttt{false} & x \neq 0, \end{cases}$$

because if no weights are used, we can only form the sum 0. Finally, $\texttt{possible}(x, n)$ tells us whether we can construct a sum x using *all* weights.

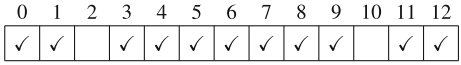

Fig. 6.4 Constructing sums using the weights [1, 3, 3, 5]

	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=0$	✓												
$k=1$	✓	✓											
$k=2$	✓	✓		✓	✓								
$k=3$	✓	✓		✓	✓		✓	✓					
$k=4$	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Fig. 6.5 Solving the knapsack problem for the weights [1, 3, 3, 5] using dynamic programming

Figure 6.5 shows all values of the function for the weights [1, 3, 3, 5] (the symbol "✓" indicates the `true` values). For example, the row $k = 2$ tells us that we can construct the sums [0, 1, 3, 4] using the weights [1, 3].

Let m denote the total sum of the weights. The following $O(nm)$ time dynamic programming solution corresponds to the recursive function:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= m; x++) {
        if (x-w[k] >= 0) {
            possible[x][k] |= possible[x-w[k]][k-1];
        }
        possible[x][k] |= possible[x][k-1];
    }
}
```

It turns out that there is also a more compact way to implement the dynamic programming calculation, using only a one-dimensional array `possible`[x] that indicates whether we can construct a subset with sum x. The trick is to update the array from right to left for each new weight:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = m-w[k]; x >= 0; x--) {
        possible[x+w[k]] |= possible[x];
    }
}
```

Note that the general dynamic programming idea presented in this section can also be used in other knapsack problems, such as in a situation where objects have weights and values and we have to find a maximum-value subset whose weight does not exceed a given limit.

6.2.4 From Permutations to Subsets

Using dynamic programming, it is often possible to change an iteration over permutations into an iteration over subsets. The benefit of this is that $n!$, the number of

permutations, is much larger than 2^n, the number of subsets. For example, if $n = 20$, $n! \approx 2.4 \cdot 10^{18}$ and $2^n \approx 10^6$. Thus, for certain values of n, we can efficiently go through the subsets but not through the permutations.

As an example, consider the following problem: There is an elevator with maximum weight x, and n people who want to get from the ground floor to the top floor. The people are numbered $0, 1, \ldots, n-1$, and the weight of person i is $\texttt{weight}[i]$. What is the minimum number of rides needed to get everybody to the top floor?

For example, suppose that $x = 12$, $n = 5$, and the weights are as follows:

- $\texttt{weight}[0] = 2$
- $\texttt{weight}[1] = 3$
- $\texttt{weight}[2] = 4$
- $\texttt{weight}[3] = 5$
- $\texttt{weight}[4] = 9$

In this scenario, the minimum number of rides is two. One optimal solution is as follows: first, people 0, 2, and 3 take the elevator (total weight 11), and then, people 1 and 4 take the elevator (total weight 12).

The problem can be easily solved in $O(n!n)$ time by testing all possible permutations of n people. However, we can use dynamic programming to create a more efficient $O(2^n n)$ time algorithm. The idea is to calculate for each subset of people two values: the minimum number of rides needed and the minimum weight of people who ride in the last group.

Let $\texttt{rides}(S)$ denote the minimum number of rides for a subset S, and let $\texttt{last}(S)$ denote the minimum weight of the last ride in a solution where the number of rides is minimum. For example, in the above scenario

$$\texttt{rides}(\{3,4\}) = 2 \text{ and } \texttt{last}(\{3,4\}) = 5,$$

because the optimal way for people 3 and 4 to get to the top floor is that they take two separate rides and person 4 goes first, which minimizes the weight of the second ride. Of course, our final goal is to calculate the value of $\texttt{rides}(\{0 \ldots n-1\})$.

We can calculate the values of the functions recursively and then apply dynamic programming. To calculate the values for a subset S, we go through all people who belong to S and optimally choose the last person p who enters the elevator. Each such choice yields a subproblem for a smaller subset of people. If $\texttt{last}(S \setminus p) + \texttt{weight}[p] \leq x$, we can add p to the last ride. Otherwise, we have to reserve a new ride that only contains p.

A convenient way to implement the dynamic programming calculation is to use bit operations. First, we declare an array

```
pair<int,int> best[1<<N];
```

that contains for each subset S a pair ($\texttt{rides}(S)$, $\texttt{last}(S)$). For an empty subset, no rides are needed:

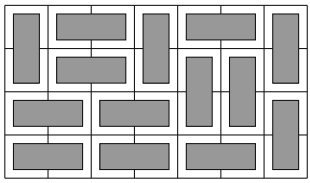

Fig. 6.6 One way to fill the 4×7 grid using 1×2 and 2×1 tiles

```
best[0] = {0,0};
```

Then, we can fill the array as follows:

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // add p to an existing ride
                option.second += weight[p];
            } else {
                // reserve a new ride for p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Note that the above loop guarantees that for any two subsets S_1 and S_2 such that $S_1 \subset S_2$, we process S_1 before S_2. Thus, the dynamic programming values are calculated in the correct order.

6.2.5 Counting Tilings

Sometimes the states of a dynamic programming solution are more complex than fixed combinations of values. As an example, consider the problem of calculating the number of distinct ways to fill an $n \times m$ grid using 1×2 and 2×1 size tiles. For example, there are a total of 781 ways to fill the 4×7 grid, one of them being the solution shown in Fig. 6.6.

The problem can be solved using dynamic programming by going through the grid row by row. Each row in a solution can be represented as a string that contains

m characters from the set {⊓, ⊔, ⊏, ⊐}. For example, the solution in Fig. 6.6 consists of four rows that correspond to the following strings:

- ⊓ ⊏⊐ ⊓ ⊏⊐ ⊓
- ⊔ ⊏⊐ ⊔⊓⊓⊔
- ⊏⊐⊏⊐ ⊔⊔⊓
- ⊏⊐⊏⊐⊏⊐ ⊔

Suppose that the rows of the grid are indexed from 1 to n. Let $\texttt{count}(k, x)$ denote the number of ways to construct a solution for rows $1 \ldots k$ such that string x corresponds to row k. It is possible to use dynamic programming here, because the state of a row is constrained only by the state of the previous row.

A solution is valid if row 1 does not contain the character ⊔, row n does not contain the character ⊓, and all consecutive rows are compatible. For example, the rows ⊔ ⊏⊐ ⊔⊓⊓⊔ and ⊏⊐⊏⊐ ⊔⊔⊓ are compatible, while the rows ⊓ ⊏⊐ ⊓ ⊏⊐ ⊓ and ⊏⊐⊏⊐⊏⊐ ⊔ are not compatible.

Since a row consists of m characters and there are four choices for each character, the number of distinct rows is at most 4^m. We can go through the $O(4^m)$ possible states for each row, and for each state, there are $O(4^m)$ possible states for the previous row, so the time complexity of the solution is $O(n4^{2m})$. In practice, it is a good idea to rotate the grid so that the shorter side has length m, because the factor 4^{2m} dominates the time complexity.

It is possible to make the solution more efficient by using a more compact representation for the rows. It turns out that it suffices to know which columns of the previous row contain the upper square of a vertical tile. Thus, we can represent a row using only the characters ⊓ and □, where □ is a combination of the characters ⊔, ⊏, and ⊐. Using this representation, there are only 2^m distinct rows, and the time complexity is $O(n2^{2m})$.

As a final note, there is also a direct formula for calculating the number of tilings:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right)$$

This formula is very efficient, because it calculates the number of tilings in $O(nm)$ time, but since the answer is a product of real numbers, a problem when using the formula is how to store the intermediate results accurately.