Journal of Technical Science and Technologies; ISSN 2298-0032; Volume 3, Issue 1, 2014

Methodology of Teaching Dynamic Programming

George MANDARIA*

Abstract

In this article the methodology of learning dynamic programing is described. Mentioned methodology is one of the main methods of solving some programming problems and foresees dividing one problem into such sub problems that using their solutions will be possible to build the solution of initial problem. The realization of this method needs memorizing the solutions of sub problems, thus, the dynamic tables are used. As well as that the solutions of sub problems and recursive dependence are explained in this article. It is showed how to use recursive dependence in order to divide a problem into sub problems. Moreover, some sample problems and their solutions are included.

Keywords: Dynamic programming methodology, Dynamic tables, right recursive dependencies, Sub problems

Introduction

Dynamic programming methodology is one of the main tools used to solve many problems in the field of informatics. It actually is mentioned in a lot of algorithmic books, but in a way, that for a beginner programmer it will be quite hard to understand the essence of it. That's why while learning this method, it is crucial to explain the essence and the practical use of method understandably.

While formulating this or that problem it is essential to determine the initial data of it, which is also called the parameters of the problem.

For example, while solving linear equation ax=b, it means that this problem is determined by two parameters a and b.

If we want to solve the problem of calculating the arithmetic mean of some numbers, then the parameters of the problem will be the amount of numbers and their values.

However, we're not yet interested in a concrete algorithm of solving problems. Our goal is to learn solving the problem by finding the solutions of sub problems. In this case it is suggested to consider the algorithm as the function of converting the input data into output, which will be the solution of the initial problem.

Thus the above mentioned approach explains that any problem can be formalized in a function, the arguments of which might be:

- the amount of parameters
- the values of parameters

Now and later we will consider as parameters positive integers only.

Usually, one of the arguments of the problem is considered to be the amount of its parameters. In case when by value of this parameter it is possible to determine the concrete values of other parameters, the last mentioned can be ignored. It usually occurs when parameters are given by table. For example, if we want to find the sum of first K elements in the table, to solve this problem is enough to know just the value of one K element, the other parameters can be chosen from the table (Mandaria, 2012).

As soon as the problem is formalized as a function with some arguments, we can bring in the statements of sub problems. By saying "sub problem" we mean the same problem but with the fewer amounts of parameters, or with the same amount of parameters, but at least one of them should have the value less than in initial.

Methodology

As an example, let's consider the problem of looking for the heaviest coin from ten given. To formulate the problem let's determine the function "the heaviest coin", the arguments of which are the amount of coins (10) and the weight of each one, thus the function will have 11 arguments. For now, we are not interested in this function concretely. The most important here is that this function gives us the correct solution.

We can consider 9 sub problems for the given initial problem. The sub problems will have the fewer amounts of arguments:

"the heaviest coin" from one coin,
"the heaviest coin" from first two coins,
"the heaviest coin" from first three coins;

"the heaviest coin" from first nine;

Thus, in our function ("the heaviest coin") argument is believed to be the amount of coins, considering which it is possible to determine the weight of each coin. Above mentioned problems have the fewer amount of arguments than the initial problem. To be more precise in the sub problems

* Assoc. Prof. Dr., Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia
E-mail: gmandaria@ibsu.edu.ge

we just have one argument instead of 11, which makes the function way easier. Decreasing the amount of arguments is achieved by the correct formulation of sub problems: "the heaviest coin from **first K** coins". That makes determining the exact weight of first K coins possible and thus there is no need in consideration of these weights as a separate argument. In case if the sub problem was formulated so "the heaviest coin from K coins", their weights would be separate arguments, because we would not know from N coins which K coin is considered. That's exactly why it is crucial to formulate sub problems correctly while using dynamic programming methods.

It should be mentioned, "sub problems" must not be understood as some stages of solving the initial problem, like organizing data input and output, data sorting or solving any part of the given problem.

As we already mentioned, one of the main methods of solving the problems is dividing one problem into such sub problems that using their solutions will be possible to build the solution of initial problem.

Thus, to solve the problem it might be needed to solve one or more sub problems.

The method of solving the initial problem on the base of the solutions of sub problems can be given in a dependence form, where the initial problem's corresponding function's values are determined by the sub problem's corresponding function's values.

Dependencies which connect the same functions, but with different arguments are called **recursive dependencies, or recursive equations**.

Recursive dependencies (equations), in which the amount of input functions' arguments or values from the right side are less than those from the left side are called **right recursive dependencies**. In case of having multiple arguments, decreasing the value of at least one of them, would be enough.

We should pay attention to the fact that dependencies should be determined for each possible value of arguments. Thus, the values of a function should be determined for initial value of parameters.

For example, it should be mentioned that recursive dependency: $S(i)=S(i-1)+ai$, $i \geq 1$, which connects two functions with different parameters: S(i) and S(i-1), also a(i) and a(i-1) for any value of i would be incorrect without S(0) and a(0) initial values, as it is not determined for i=1.

Of course, there are more difficult dependencies that connect more than two functions to one another.

The next important step after the problem is divided into sub problems and the recursive equations are determined, is the method of building of solution of the initial problem by using the solutions of sub problems.

One effective way of "memorizing" the solutions of sub problems is using the tables. The method of solving problems this way is called dynamic programming method. *The main point of this method is that for each following value of parameter in order to find the solution of sub problem, and consequently the solution of initial task, we are using not the given initial data, but we find it using already found solutions of sub problems for the previous values of parameter and preliminarily determined recursive equations* (Mandaria, 2013).

Sub problem is formulized in function form, which has one or more arguments. If we consider a table with the amount of elements equal to the amount of groups of all possible different values of function arguments, then we can match each group to the table element. When all elements of the table (solutions of sub problems) are known, we can find the solution of initial task. However, we need to consider more or less rational method of finding the table elements.

Generally, for one dimensional array this type of method is represented by sequential calculation, either started from the first element, or the second one. For two-dimensional array the method of calculating the elements is a little bit more complex and it can differ depending on a given problem.

Also it should be mentioned, that in case of two-dimensional arrays the solution of an initial problem might not always be the finally calculated value: it may be the maximum element of dynamically constructed array, the last element of the first line and etc. (Cormen Thomas, Leiserson Charles, Rivest Ronald, Stein Clifford, 2001).

Let's discuss several problems, which are solved using the methods of dynamic programming.

Problem # 1. Find the number of sequences with the size of N consisting of 0-es and 1-s, where none of the two 1-s are standing next to each other (Berov, Lapunov, Matiukhin, Ponomarev, 2000).

The algorithm for solving this problem is the following: we should mark the number of sequences having length K with F_k and try to find this number by using of the numbers of similar sequences with lengths less than K. If the last symbol of such sequence of having length K is 0, then for the previous K-1symbol we can take any sequence having the length K-1 and the number of such sequences is F_{k-1}. But if the last symbol is 1, then, according to the conditions, the previous symbol has to be 0 and as its previous K-2 symbol we can take any sequence with the length of K-2, and its number is F_{k-2}. Therefor we get the following: $F_k = F_{k-1}+F_{k-2}$.

If N=10, then the corresponding single-element dynamic table will be the following:

2	3	5	8	13	21	34	55	89	144

Problem # 2. From the given numeric sequence A[1..N] delete the minimal amount of elements, so that the elements left in the sequence create strongly increasing sequence (or, we can say, you need to find the maximal strongly increasing subsequence of sequence A).

For better explanation, let's discuss clearer, but, as it always happens, less affective (very slow) algorithm. Let's generate every subsequence of the sequence containing N elements and for each of them check whether it is the maximal, strongly increasing subsequence or not.

Let's generate every number from 0 to 2^N -1, find their binary representations and create subsequences from the elements of array A, with the indexes corresponding to the unit bits in this representation.

We'll have 2^N such subsequences, that's why even for not so big N, this algorithm will work too slowly.

Let's say, that while generating those subsequences we found strongly increasing subsequence with K element in it. After that we only need to check the subsequences containing more then K elements.

Let's discus the initial sequence containing N elements. If it isn't our target, then we should generate a subsequence with N-1 elements. If the target still isn't found, we should check a subsequence with N-2 elements and so on. In the worst case we'll have to discus 2^N different situations (Kotov, Lapo, 2000).

Journal of Technical Science and Technologies; ISSN 2298-0032; Volume 3, Issue 1, 2014

For solving this problem faster, we can use Dichotomy method for K (number of elements in subsequence).

Now, let's discus more effective method for solving the given problem. Let's initialize A, B and C arrays having the length N. In A[1..N] we keep the numbers from the initial sequence; element B[i] is the length of the target subsequence with the ending element A[i]. C[i] is an index of an element, standing directly before of A[i]. (C[i]=0, if the previous element doesn't exist).

If N=1, then A[1] is a subsequence we're looking for. Also B[1]=1 and C[1]=0. Let's assume that B and C arrays are already filled with elements from first till (i-1)th. Let's try to get B[i] and C[i] elements (which means determining recursive dependencies). For this we should go through the array A from first till (i-1)th element and look for K index, for which this statements will be true:

1. A[k]<A[i];
2. B[k] is maximum.

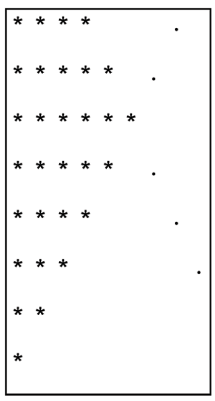
It is clear, that finding the sub sequence of maximum length which will be ending with A[i] element is possible if we write this element in the end of such subsequence that has the A[k] element as last. Thus, these will be the recursive formulas: B[i]=B[k]+1 and C[i]=k.

Let's say we went through all N elements of A array and found the maximum element in B array. Let's call the index of this element *IndexMax*. The value of this element is the length of longest subsequence.

The method of finding the needed subsequence is as follows. Let's say we want the output to be starting from the end of the subsequence and thus j is its current index. First of all, we state: j= IndexMax and we write A[j], which is the last element, as an output. The previous element of it in the subsequence will be C[j], That's why its index from end will be determined as j= C[j]. Above mentioned operations should be repeated until j value will become 0 (which will mean that we've reached the beginning of the subsequence). The algorithm, written in C++ will be as follows:

```
C[1]=0; B[1]=1; Max=1; IndexMax=1;
For (i=2; i<=N; i++)
   {  Max1=0;
      p=0;
      for(k=1; k<=i-1; k++)
         if (A[k]<A[i] && Max1<B[k])
         {  Max1=B[k];
            p=k; }
         C[i]=p;
         B[i]=Max1+1;
         if (B[i]>Max)
         {   Max=B[i];
            IndexMax=I; } }
j=IndexMax;
While (j<>0)
{   Cout<<A[j]<<" ";   j=C[j];  }
```

Problem #3. Make a new B matrix from the given A matrix with size NxN, which will have the same size. Bij is element equal to the maximum value taken from the area of array A, wich is surrounded from the right side by diagonal passing Aij element.



The obvious solution of the problem is in using such procedure that by (i, j) coordinates (numbers of rows and columns) of elements is looking for elements with maximum values in the corresponding part of matrix A.

Thus, it is not hard to notice that this statement is true for the first column of B matrix:

$$B[i][1]=A[i][1],\ i=1, 2, ..., N$$

The elements of other columns can be found as following:

$$B[i][j]=\max(A[i][j], B[i-1][j-1], B[i][j-1], B[i+1][j-1])$$

Apart from that it should be mentioned that the matrix indexes should not exceed of the matrix boarders.

Thus, we are building square NxN dynamic matrix, which is filled by columns. The elements of this matrix are found by comparing already known "neighbor" elements and corresponding element from the initial matrix.

Problem #4. Two strings of symbols are given: $x=a_1a_2...a_m$ and $y=b_1b_2...b_n$. d(x, y) is used to determine the minimum amount of taken out, changed and put in symbols needed to transform x string into y string. For example, d(ptslddf, tsglddss)=3.

taking out p putting g in changing f

ptslddf ⟶ tslddf ⟶ tsglddf ⟶ tsglddss

For the given x and y strings find d(x, y).

For x=a1…am and y=b1…bn, ai and bj are symbols, $1 \le i \le m$, $1 \le j \le n$. Finding d(x, y) using dynamic programming method is possible as follows:

We determine d[m][n] matrix elements of wich

$$d[i][j]=d(a_1...a_i, b_1...b_j),\ 0 \le i \le m,\ 0 \le j \le n$$

It is clear that d[0, j]=j; d[i, 0]=i.

As well as that it is not hard to see that:

$$d[i][j]=\min(d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+P_{ij}),$$

where $P_{ij}=1$ if $a_i \neq b_j$ and $P_{ij}=0$, if $a_i=b_j$. In the above mentioned representation the first element of min is corresponded to the deleting operation of last element ai in $a_1...a_{i-1}a_i$ string, after what within d[i-1][j] operations $a_1...a_{i-1}$ is transformed into $b_1...b_j$ string. Second element is corresponded

to the insertion of element bi in the end of b1...bj_1 string which is a result of transformation of $a_1...a_i$ row within d[i][j-1] operations. The third element corresponds to the change of ai element by bj element. The change occurs when $a_i \neq b_j$ (in this case P_{ij}=1) and not occurs when ai=bj.

d[m][n] is the minimum amount of operations needed for the transformation of x string into y.

The dynamic two-dimensional table for the example used in problem#4, where m=7, n=7, x=ptslddf, y=tsgldds will have form as is shown below:

		0	1	2	3	4	5	6	7
			t	s	g	l	d	d	s
0		0	1	2	3	4	5	6	7
1	p	1	1	2	3	4	5	6	7
2	t	2	1	2	3	4	5	6	7
3	s	3	2	1	2	3	4	5	6
4	l	4	3	2	2	2	3	4	5
5	d	5	4	3	3	3	2	3	4
6	d	6	5	4	4	4	3	2	3
7	f	7	6	5	5	5	4	3	**3**

The solution of the initial problem is the last, d[7][7] element of the table - 3. The algorithm written in C++ will be as follows:

```
for(i=1; i<=m; i++)
  d[i][0]=i;
for(j=1; j<=n; j++)
  d[0][j]=j;
for(i=1; i<=m; i++)
  for(j=1; j<=n; j++)
      d[i][j]=min(d[i-1][j]+1, d[i][j-1]+1);
      d[i][j]=min(d[i][j], d[i-1][j-1]+P);
```

Conclusion

It is crucial to formulate the sub problem correctly while using dynamic programming method for solving problems or determining recursive dependencies. We must try to decrease to minimum the amount of parameters used in the sub problems, so that the problem is formalized in a form of function with the minimum amount of arguments. It should be mentioned that dependencies should be determined for all possible values of arguments. That's why the function values should be determined for the initial values of parameters.

The corresponding to the initial problem function is determined by the functions corresponding to sub problems. Apart from that, it is important that rational method is used to find the elements of dynamic table. When the elements of dynamic table are known, we can find the solution of the initial problem.

References

Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Stein Clifford. (2001). *Introduction to Algorithms (2nd Edition).* Boston: Massachusetts Institute of Technology.

Mandaria, G. (2012). Solution of some Computer Studies Problems by Reducing it to a Sub-Problems - Recurrent Relations. Periodical Scientific Journal "INTELECTI" #3(44), pp. 49-51.

Mandaria, G. (2013). Method of the Organization of Dynamic Tables. *Periodical Scientific Journal "INTELECTI" #1(45)*, pp. 89-91.

Berov V.I., Lapunov A.V., Matiukhin V.A., Ponomarev A.E. (2000). *Features of National Tasks in Informatics.* Kirov.

Kotov V.M., Lapo I.A. (2000). *Algorithmic Methods.* Minsk.