Academic Research in Educational Sciences
ISSN: 2181-1385
DOI: 10.24412/2181-1385-2022-3-525-528

Volume 3 | Issue 3 | 2022
Cite-Factor: 0,89 | SIS: 1,12
SJIF: 5,7 | UIF: 6,1

DYNAMIC PROGRAMMING BY SUBSEGMENTS

I. Z. Iskandarov
B. B. Nurmetova
B. I. Sobirov
Urgench Branch of Tashkent University of Information Technologies
named after Muhammad al-Khwarizmi
E-mails: islom.iskandarov@ubtuit.uz , bonuraxon20102018@gmail.com, bahrombek0960@mail.ru

ABSTRACT

This article discusses dynamic programming and one of its types dynamic programming by subsegments, as well as several examples and solutions to them.

Keywords: dynamic programming, algorithms, greedy algorithm

INTRODUCTION

In problems in which it is required to find the optimal answer, different solution methods are used: the "brute force" method, "greedy" algorithms, etc. The brute force method is based on enumerating all the solutions and choosing the optimal one. In most cases, this approach is relatively easy to implement and finds a solution, but cannot be applied with large input data. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution[1]. This approach is faster than the brute force method and can be applied to large inputs, but sometimes it may not find the optimal solution. Dynamic programming(DP) can be another method for solving the problem. The idea behind dynamic programming is to split a problem into subtasks and solve them separately using the results of other subtasks or previous steps. In order for the problem to be solved by the dynamic programming method, the following conditions must be met: in the problem, subtasks of a similar structure of a smaller size can be distinguished; among the selected subtasks there are trivial ones, that is, having a "small size" and an obvious solution; an optimal solution to a larger subproblem can be constructed from optimal solutions to subproblems; subproblem solutions are stored in tables with reasonable sizes [2]. Based on the options for using dynamic programming in solving problems, the following types can be distinguished: linear, DP by sets, DP by profile, etc. Another type is DP by subsegments. This type is used when solving problems in which we can take the calculation of values on subsegments of a some array as a subtask, and the answer to the problem is the result of calculating a segment of the dimension of the array.

METHOD DESCRIPTION

The essence of the method consists in splitting an array of values into subsegments and finding a solution for subsegments and combining the results. When solving the problem, we calculate the results in the order of increasing the length of the subsegments. Thus, when calculating the results for subsegments of length L, we use the values for subsegments of lengths in the interval $[0; L-1]$. Usually we use a two-dimensional array dp to store the results, where $dp\ [left, right]$ will store the answer for the segment

from *left* to *right*. In most cases, due to the use of nested loops, we get asymptotics in time $O(n^2)$ or $O(n^3)$, and in memory $O(n^2)$. Below is the pseudocode of the implementation.

```
for length=1 to n
for left=1 to n-length+1
        right=left+length-1
        // calculate the value dp[left,right]
```

EXAMPLES OF USING

Problem 1. Let there be a sequence (chain) consisting of *n* matrices and their product

$$A_1 A_2 \cdots A_n$$

Matrix multiplication has the property of associativity, so the result does not depend on the order of multiplication, but the time spent on calculating the product can strongly depend on this. You need to find the minimum number of operations required to calculate the product. The sizes of matrices are given in the form of *n* pairs of arrays *h[]* and *w[]*,and *w[i]=h[i+1]* for all indices *i* that satisfy the condition *i<n*.

Note. For more details on the problem and solution, see[3]

Solution. As a subproblem we take *dp[i,j]* – the minimum number of operations required for matrix multiplication in the interval *[i; j]*.

Base: *dp [i,i]=0.*

Recurrent formula:

$$dp[i,j] = min(dp[i,k] + dp[k+1,j] + h[i] * w[i] * w[j]), where\ k\ \in\ [i,j)$$

Pseudocode.

```
for i=1 to n
  dp[i,i]=0
for int length=2 to n
  for i=1 to n-length+1
    j=i+length-1
    dp[i][j]=INF
    for k=i to j-1
  dp[i,j]=min(dp[i,j],dp[i,k]+dp[k+1,j]+h[i]*w[i]*w[j])
output dp[1,n]
```

Time complexity: $O(n^3)$

Problem 2. A **palindrome** is a string that reads in the same way both from left to right and from right to left. Given a string s, you need to find the length of the longest subsequence that can be obtained by deleting some letters from the given sequence in such a way that the remaining subsequence will be a palindrome.

Solution. Subproblem *dp[i,j]* is the length of the longest subsequence of the palindrome in the subsegment *[i;j]*.

Base: *dp[i,i]=1*

Recurrent formula:

$$dp[i,j] = \begin{cases} dp[i+1][j-1] + 2, if\ s[i] = s[j] \\ max(dp[i][j-1], dp[i+1][j]), if\ s[i] \neq s[j] \end{cases}$$

Pseudocode.

```
for i=1 to n
dp[i,i]=1
for length=1 to n
  for i=1 to n-length+1
    j=i+length-1
      if s[i] == s[j]
        dp[i][j]=dp[i+1][j-1]+2
      else
        dp[i][j]=max(dp[i][j-1],dp[i+1][j])
output dp[1,n]
```

Time complexity: $O(n^2)$

Problem 3. A sequence of *n* integers is written on the board. Two are playing. On the next move, the player selects a number from the right or left side of the sequence, then this number is erased and the sequence becomes one number less, and the move goes to the opponent. The winner is the one who collects more in total. Find the points that players will gain when playing optimally.

Solution. Subproblem *dp[i, j]* – points that the first player will score in the optimal game on the subsegment *[i; j]*. Points are given as an array of numbers w*[]*. Let's define the function *sum(l, r)* – the sum of the numbers of the array *w[]* in the range of indices *[l; r]*.

Base: *dp[i,i]=w[i]*

Recurrent formula:

$$\begin{cases} dp[l,r] = max(w[l], w[r]), if\ l+1 = r \\ dp[l,r] = max\begin{pmatrix} w[l] - dp[l+1,r] + sum(l+1,r), \\ w[r] - dp[l,r-1] + sum(l,r-1) \end{pmatrix}, else \end{cases}$$

Pseudocode.

```
for i to n
dp[i,i]=w[i]
for i=1 to n
  for j=1 to n-i+1
    l=j
    r=i+j-1
    if l+1==r
      dp[l,r]=max(w[l],w[r])
    else
                dp[l,r]=max(w[l]- dp[l+1,r]+sum(l+1,r),    w[r]-dp[l,r-1]+sum(l,r-1))
                output dp[1,n] // number of points scored by the first player
output sum(1,n)-dp[1,n]  // number of points scored by the second player
```

Time complexity: $O(n^2)$

CONCLUSION

The method we have considered can be used in the problems of finding the optimal answer or calculating the number of options that satisfy a certain condition. To use the method for a task, it is necessary that we can take a subsegment as a subtask for DP. Thus, this approach can be effective for a number of specific tasks.

REFERENCES

[1]. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms”, 3rd ed, pp. 414

[2]. Национальный исследовательский университет «Высшая школа экономики». Факультет компьютерных наук. Летняя школа по компьютерным наукам. Август. 2016. Параллель С. Шуйкова И.А. http://shujkova.ru/sites/default/files/lec5.pdf

[3]. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms”, 3rd ed, pp. 370-377.