Stock Cutting Problem

Vikram Singh
IIT2019213
iit2019213@iiita.ac.in

Akshat Agrawal
IIT2019214
iit2019214@iiita.ac.in

Ravutla Ruthvik
IIT2019215
iit2019215@iiita.ac.in

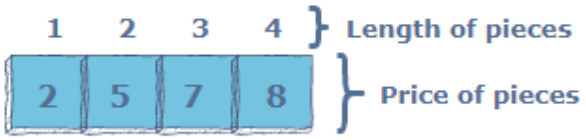
Abstract**—** In this report, we discuss several algorithms and try to figure out which is the best one to solve the stock cutting problem.

Keywords**—** stock, recursion, memoization, dynamic programming, time complexity, space complexity.

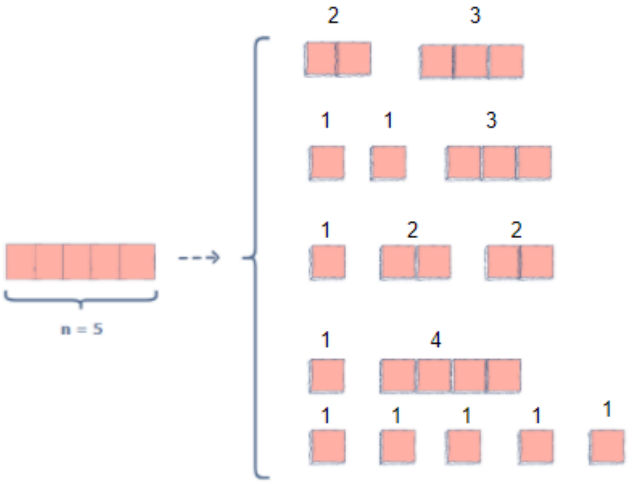
INTRODUCTION

This paper discusses several algorithms that are designed to solve the stock cutting problem.

The problem statement for the stock cutting problem is that given a stock of length n inches and an array of prices that contains prices of all pieces of size smaller than n. We have to determine the maximum value obtainable by cutting up the stock and selling the pieces. Suppose that we have a stock of length 5, and an array containing the length(1,2,3 and 4) and price(2,5,7 and 8) of the pieces.



There are various ways to cut the stock into sub-stocks, each way results in a certain profit.



The answer should be 12 (selling the sub-stocks of length 2+3 gives a 5+7=12 profit or of length 1+2+2 gives a 2+5+5=12 profit).

Algorithm-01

Naive Algorithm - A naive solution for this problem would be to generate all the configurations, of all the different pieces, to find the highest priced configuration.

Optimal configuration: To consider all configurations of pieces of stock, there can be two cases for every piece of stock:

Case 1: The piece of stock is included in the optimal configuration.

Case 2: The piece of stock is not included in the optimal configuration.

Therefore, the maximum value that can be obtained from ‘n’ pieces of stock is the max of the following two values.

1. Maximum value obtained by n-1 pieces of stock and L length (excluding nth piece of stock).

2. Value of nth piece of stock plus maximum value obtained by n pieces of stock (as repetition is allowed) and L minus the length of the nth piece of stock (including nth piece of stock).

If the length of ‘nth’ piece of stock is greater than ‘L’, then the nth piece of stock cannot be included and Case 1 is the only possibility.

PSEUDO CODE:

```
func stockCut(int L, int length[],
int price[], int n)
{
     if (n == 0 || L == 0)
     return 0;  // Base Case

     if (length[n - 1] > L)
     return stockCut(L, length,
price, n - 1);
      else
        return max(price[n - 1]+
stockCut(L - length[n - 1], length,
price, n), stockCut(L, length,
price, n - 1));
}
```

Time Complexity:

This method involves generating all the subsets for a given set of elements. The total number of subsets possible for a given set of elements is given by 2^n (where n is the number of elements).

Thus this solution is **Exponential** in terms of time complexity.

Space Complexity:

There are two arrays of length n which are given to us. The number of variables involved in the recursive function are constant. Thus the space complexity would be **O(n).**

Algorithm-02

Using Memoization - This method uses Memoization Technique (an extension of recursive approach).

This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity.

We can solve this problem by simply creating a 2-D array that can store a particular state (n, l) if we get it the first time.

Now if we come across the same state (n, l) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time.

This method gives an edge over the recursive approach in this aspect.

PSEUDO CODE:

```
int visited[MAX][MAX]=-1;

func stockCut(int L, int length[],
int price[], int n)
{
```

```
      if (n == 0 || L == 0)
     return 0;  // Base Case-1
      if (visited[n][l]!=-1)
     return visited[n][l];          // Base Case-2

      if (length[n - 1] > L)
      return stockCut(L, length,
price, n - 1);
       else
         return max(price[n - 1]+
stockCut(L - length[n - 1], length,
price, n), stockCut(L, length,
price, n - 1));
}
```

The visited matrix obtained after doing these calculations for the example taken above is:

	0	1	2	3	4	5
0	-1	-1	-1	-1	-1	-1
1	-1	2	2	2	2	-1
2	-1	4	5	5	-1	-1
3	-1	6	7	-1	-1	-1
4	-1	8	-1	-1	-1	-1
5	-1	10	12	12	12	12

Time Complexity:

In this method all the redundant calculations are avoided. Thus we are left with two nested recursive calls in which the value of n reduces by 1 in every function call, uptill 0 . Thus the time complexity for this algorithm is **$O(n^2)$.**

Space Complexity:

There are two arrays of length n which are given to us. We then declare a 2-D array of size NxN.

There is some stack memory used in the recursive functions in which the auxiliary space required is significantly lower than $O(n^2)$. Thus the overall space complexity for the algorithm would be $O(n) + O(n^2) \approx$ **$O(n^2)$.**

Algorithm-03

Using Dynamic Programming - Since dynamic programming is mainly an optimization-over-plain recursion, we can use it to optimize the above exponential time algorithm. The idea is to store the results of subproblems so that we do not have to re-compute them when they are needed.

So, in the above example:

In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach.

In a DP[][] table let's consider all the possible number of pieces from '1' to 'n' as the rows and length that can be kept as the columns.

Every element of the DP table contains the maximum profit possible of the sub-matrix upto that element. Thus the element which has row and column both as n (last element) gives the maximum profit possible for the given problem.

First we initialize the matrix for the trivial cases discussed below:

- If the number of pieces are 0, then the profit will obviously be 0.
- If the length of the stock is 0, no matter what the number of pieces are, they all

are of 0 length, so the profit will again be 0.

The state DP[i][j] will denote the maximum value of 'j-length' considering all values from '1 to ith'. So if we consider 'li' (length in 'ith' row) we can fill it in all columns which have 'length values > li'. Now two possibilities can take place:

- Fill 'li' in the given column.
- Do not fill 'li' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' length in 'jth' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the length, DP[i][j] will be equal to the value of 'li'+ price of the column whose length is 'j-li' in the previous row.

So we take the maximum of these two possibilities to fill the current state.

The DP matrix obtained after doing these calculations for the example taken above is:

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	2	4	6	8	10
2	0	2	5	7	10	12
3	0	2	5	7	10	12
4	0	2	5	7	10	12
5	0	2	5	7	10	12

The maximum profit is given by the element present at index dp[n][n] = 12. (Here n=5)

PSEUDO CODE:

```
int dp[n+1][n+1];
int stockCut(int price[], int
length[],int n)
{
 for (int i = 0; i <= n; i++)
 {
  for (int j = 0; j <= n; j++)
   {
      if (i == 0 || j == 0)
      dp[i][j] = 0;
   }
}
for (int i = 1; i <= n; i++)
 {
  for (int j = 1; j <= n; j++)
  {
   if(length[i-1]>j)
    dp[i][j]=dp[i-1][j];
   else
      {
dp[i][j]=max(price[i-1]+dp[i][j-
length[i-1]],dp[i-1][j]);
      }
   }
  }
    return dp[n][n];
}
```

Time Complexity:

This method involves two nested-for loops in which we iterate from 1 to n. Thus the time complexity for this algorithm is **$O(n^2)$.**

Space Complexity:

There are two arrays of length n which are given to us. We then declare a 2-D array of size NxN. Thus the overall space complexity

for the algorithm would be $O(n) + O(n^2) \approx$ **$O(n^2)$.**

CONCLUSION

In this report we have used overall 3 algorithms to solve the Stock cutting problem.

First, we used the simple recursive algorithm to solve the problem. This algorithm, though fairly easy to understand and code, it's time complexity was exponential which was way too much.

Second, we used the memoization technique which was merely an extension of the recursive algorithm. In this method all the redundant calculations are avoided.
Thus the time complexity reduces drastically to $O(n^2)$. Extra auxiliary space required increases from $O(n)$ to $O(n^2)$.

Third, we used dynamic programming. In this method, no recursion is involved. Only two nested for loops and a 2-D matrix are used.
Thus the time complexity involved is $O(n^2)$. Extra auxiliary space required is $O(n^2)$. No extra stack memory is required, which was the case in the memoization technique.

Thus, we conclude that the algorithm involving dynamic programming is the best for solving the stock cutting problem as its time complexity is way better than that of the recursive method and its space complexity is slightly better than that of the memoization technique.

ACKNOWLEDGEMENT

We are very much grateful to our Course instructor Dr Rahul Kala and our mentor Mr. Md. Meraz, who have provided the great opportunity to do this wonderful work on the subject of Data Structure and Algorithm Analysis.

REFERENCES

[1] Introduction to Algorithms / Thomas H. Cormen . . . [et al.]. - 3rd edition.
[2] The Design and Analysis of Algorithms (Pearson) by A V Aho, J E Hopcroft, and J D Ullman
[3] Algorithm Design (Pearson) by J Kleinberg, and E Tard