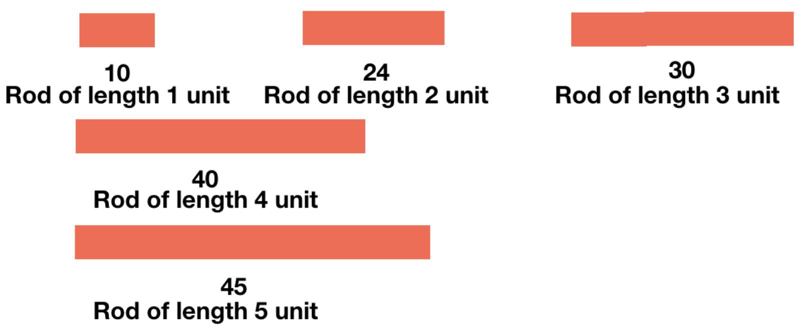
(/add_ques

Rod Cutting | Dynamic Programming

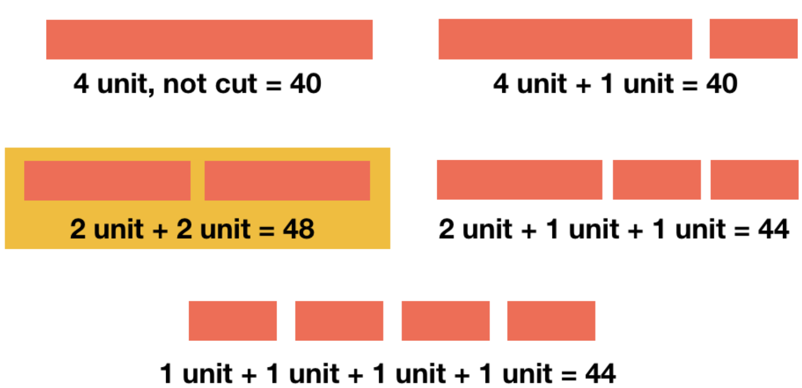
Rod cutting is another kind of problem which can be solved without using dynamic programming (/course/algorithms-dynamic-programming/) approach but we can optimize it greatly by using it. According to the problem, we are provided with a long rod of length n units. We can cut the rod in different sizes and each size has a different cost associated with it i.e., a rod of i units length will have a cost of $c_i$$. Let's look at the table given below showing the cost v/s length of the rod.

Length	1	2	3	4	5
Price	10	24	30	40	45



From the above table, you can see that the rod of length 1 unit has a price of 10$ and a rod of length 2 has a price of 24$, etc. Now, our task is to generate the pieces of the rod in such a way that the revenue generated by selling all the pieces is maximum (let's say this maximum revenue is r_n for a rod of length n units).

Let's take a case when our rod is 4 units long, then we have the following different ways of cutting it:

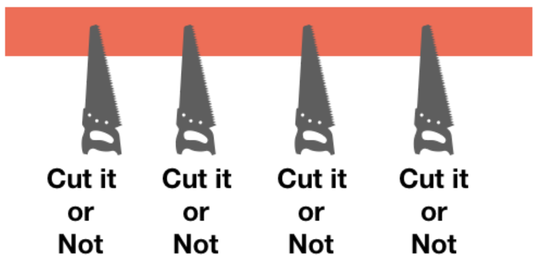


Thus, you can see that for a rod of 4 units long, cutting it into two pieces of 2 units length generates the maximum profit for us.

Our task is to find the value of r_n (maximum revenue that can be generated from a rod of length n units).

Before moving further, let's first analyze the brute force way of solving it i.e., by checking all the possible solutions.

For a rod of n units long, for every i units, we have two choices - either make that cut or not.



So, we have to choose between two option for a total of n-1 times and thus the total possible number of solutions are $\underbrace{2 * 2 * \ldots * 2}_{n-1 \text{ times}} = 2^{n-1}$.

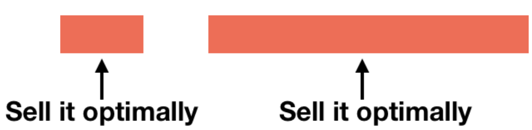
Like the 0-1 knapsack problem, this is also exponential and we need to optimize this to make it useable.

Approach to Solve Rod Cutting Problem

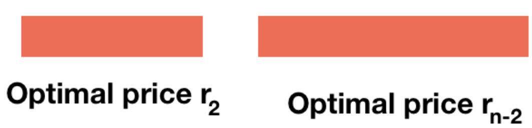
First Approach

Let's say we have a rod of n units long. Now, we can make cuts at 1, 2, 3, ... n-1 units of length or even no cut at all. In the case of no cutting at all, the rod will be sold at $c_n\$$.

Let's say we have made a cut at 1 unit of length. Now, we have two pieces. So we will generate maximum revenue from these two pieces i.e., by selling both the pieces optimally. Thus, we will sell the first piece at $r_1\$$ and the second at $r_{n-1}\$$.



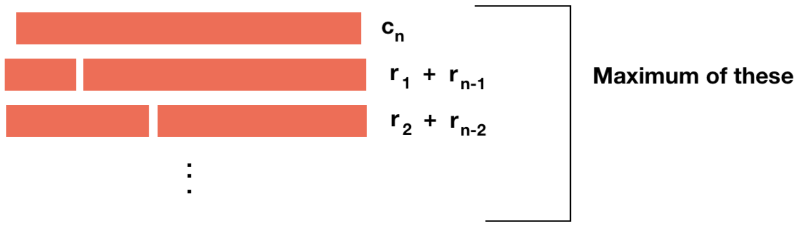
Similarly, we can make a cut at 2 unit length and then sell both the pieces optimally at $r_2\$$ and $r_{n-2}\$$ to generate a maximum revenue.



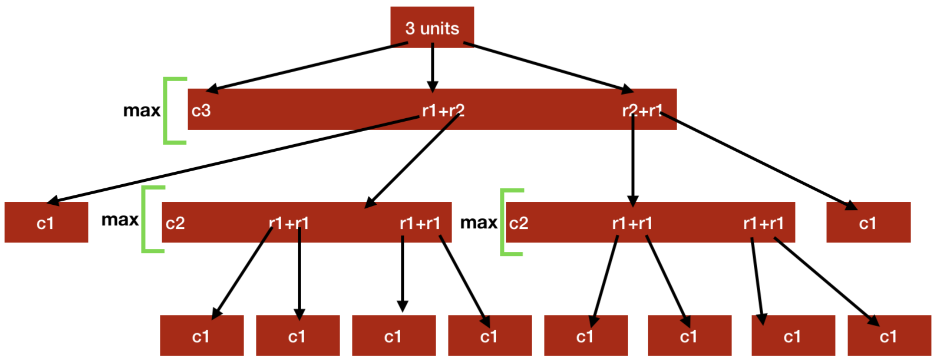
You can see that the optimal solution of the problem is incorporating the optimal solutions of the subproblems also. For example, by selling the smaller pieces at the optimal price, we are generating maximum profit from the bigger piece. This property is called **optimal substructure**.

We can say that when making a cut at i unit length, the maximum revenue can be generated by selling the first unit at $r_i\$$ and the second unit at $r_{n-i}\$$.

The maximum revenue for a rod of length n (r_n) will be the maximum of all these revenues.



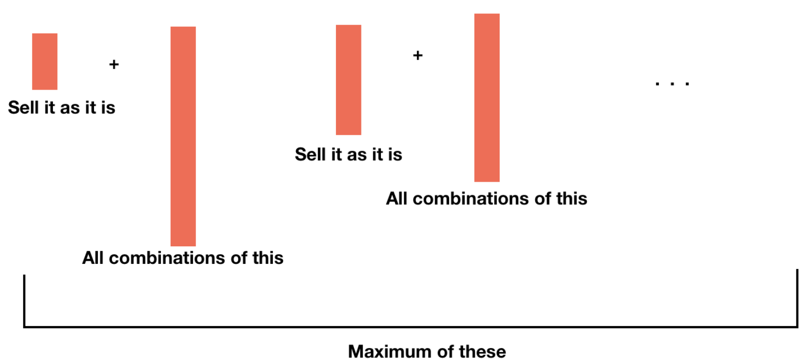
$$r_n = max\{c_n, (r_1 + r_{n-1}), (r_2 + r_{n-2}), \ldots, (r_{n-1} + r_1)\}$$



From the above picture, you can see that there are many overlapping subproblems i.e., subproblems are repeated many times. We can reduce significantly by thinking the solution of this problem in a slightly different way.

Second Approach

There are different ways in which the rod can be cut. All these different ways can be classified having two parts - one part containing the uncut part of the rod of length i and the other part containing all different possible combination of the rest of the rod (n-i). For example, let's take the case of the first part having the length of 1 unit, then we are going to sell this first piece as it is and the second part will contain all the other combination made by cutting the rod of length n-i differently.



In this case, the maximum revenue generated will be $c_1 + r_{n-1}$ i.e., selling the first piece uncut and selling the second piece by cutting in a way to get the optimal value.

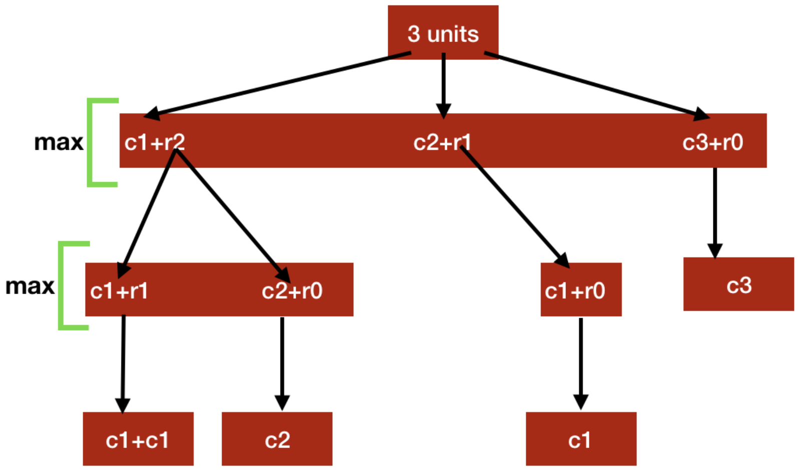
Next, we can have the rod of length 2 units uncut on the first part and all the other possible combinations made by cutting the rod of length n-2 units differently on the other part.

In this case, the maximum revenue will be $c_2 + r_{n-2}$ i.e., the first piece of length 2 units will be sold uncut and the second piece will be cut in such a way to generate the maximum revenue.

Similarly, we can generate all the possibilities that can be made by cutting the rod differently and the optimal revenue will be the maximum of these.

Thus, the formula for the optimal revenue can be written as:

$$r_n = \max_{1 \le i \le n} \{c_i + r_{n-i}\}$$



You can see that we have reduced the number of subproblems by using this formula.

Let's take the price table given above and find the optimal revenue for each length.

Length	1	2	3	4	5
Price	10	24	30	40	45

r1 = 10
r2 = 24
r3 = 34
r4 = 48
r5 = 58

After finding the solution of the problem, let's code the solution.

Code for Rod cutting problem

Let's look at the top-down dynamic programming code first.

Top Down Code for Rod Cutting

We need the cost array (c) and the length of the rod (n) to begin with, so we will start our function with these two - `TOP-DOWN-ROD-CUTTING(c, n)`

We already know that we are going to use dynamic programming, so we will start by making an array to store the maximum revenue that can be generated by different lengths i.e., `r[n+1]` so that we don't have to recalculate this value again and again.

Initially, we don't know the value of the maximum revenue that can be generated by these lengths, so we will set all the elements of this array to something negative i.e., $-\infty$. We can't initialize all the elements with 0 because the maximum revenue generated by the rod of length 0 is 0, so we will make the `r[0] = 0`.

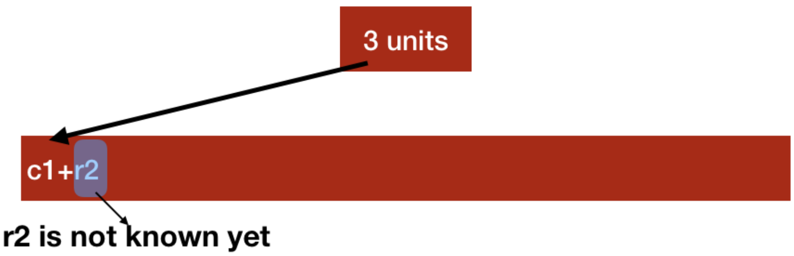
```
r[0] = 0
for i in 1 to n
  r[i] = -INF
```

(/add_ques

In the top down approach, we just start solving the problem naturally, so we will just start checking if the there is already a solution stored in the array or not.

```
if r[n] >= 0
  return r[n]
```

If the revenue is not already in the array r, then we will have to calculate it and for that, we need to calculate the maximum value among $(c_i + r_{n-i})$ for i ranging from 1 to n. Basically, $(c_i + r_{n-i})$ means the c_i + maximum revenue that can be generated by rod of length n-i i.e., `TOP-DOWN-ROD-CUTTING(c, n-i)`. Take note that we can't directly use the array r to get the value of r[n-i] because the array r doesn't store this value yet or at least we are not sure that it does or doesn't. This would have been in the case of bottom-up implementation in which we start by generating the smaller values first.



Now, the calculation of the maximum of $(c_i + r_{n-i})$ is similar to calculating the sum of all the elements of an array. We will have a variable to store the maximum revenue and then we will iterate from 1 to n and compare the current maximum revenue stored in the variable with the revenue `c[i] + TOP-DOWN-ROD-CUTTING(c, n-i)`.

```
maximum_revenue = -INF
for i in 1 to n
  maximum_revenue = max(maximum_revenue, c[i] + TOP-DOWN-ROD-CUTTING(c, n-i))
```

As stated, we started by having a variable to store the maximum revenue i.e., `maximum_revenue = -INF` and then we iterate from 1 to n to find the maximum among `c[i]+TOP-DOWN-ROD-CUTTING(c, n-i)`.

At last, we have to just fill up our array r and return this value.

```
r[n] = maximum_revenue
return r[n]
```

```
r[n+1]
r[0] = 0
for i = 1 to n
  r[i] = -INF
TOP-DOWN-ROD-CUTTING(c, n)
  if r[n] >= 0
    return r[n]

  maximum_revenue = -INF
  for i in 1 to n
    maximum_revenue = max(maximum_revenue, c[i] + TOP-DOWN-ROD-CUTTING(c, n-i))

  r[n] = maximum_revenue
  return r[n]
```

C **Python** **Java**

?

(/add_ques

```
class Rod {
  static final int INF = 100000;
  static int[] r = new int[5+1];

  public static int max(int x,int y) {
    if(x > y)
      return x;
    return y;
  }

  public static void initR(){
    r[0] = 0;
    for(int i=1; i<=5; i++) {
      r[i] = -1*INF;
    }
  }

  public static int topDownRodCutting(int c[], int n) {
    if (r[n] >= 0) {
      return r[n];
    }

    int maximumRevenue = -1*INF;
    int i;

    for(i=1; i<=n; i++) {
      maximumRevenue = max(maximumRevenue, c[i] + topDownRodCutting(c, n-i));
    }

    r[n] = maximumRevenue;
    return r[n];
  }

  public static void main(String[] args) {
    initR();
    // array starting from 1, element at index 0 is fake
    int c[] = {0, 10, 24, 30, 40, 45};
    System.out.println(topDownRodCutting(c, 5));
  }
}
```

We just saw the top down implementation. Let's look at the bottom-up implementation next.

Bottom-Up Code for Rod Cutting

In the bottom-up technique, we start by filling the array from start. So, we will first initialize an array r and then we will iterate over it to fill it.

```
r[n+1]
r[0] = 0
for i in 1 to j
```

Now, the array r contains the maximum revenue that can pe generated by each length. For example, the 4th element of the array r contains the maximum revenue that can be generated by a rod of 4 units long. So to calculate it, we need to calculate the maximum of $(c_i + r_{n-i})$ for i ranging from 1 to 4. And thus, we need to iterate again for the different values of i.

```
for i in 1 to j
   max_revenue = -INF
   for i in 1 to j
```

(/add_ques

```
      max_revenue = max(max_revenue, c[i] + r[j-i])
    r[j] = max_revenue
  return r[n]
```

Take a note that we can directly use the value stored in the array r for r_{n-i} because we know that we have already filled all the smaller values of the array.

```
BOTTOM-UP-ROD-CUTTING
  r[n+1]
  r[0] = 0
  for j = i to n
    maximum_revenue = -INF
    for i in 1 to j
      maximum_revenue = max(maximum_revenue, c[i] + r[j-i])
    r[j] = maximum_revenue
  return r[n]
```

C **Python** **Java**

```
class Rod {
  static final int INF = 100000;

  public static int max(int x,int y) {
    if(x > y)
      return x;
    return y;
  }

  public static int bottomUpRodCutting(int c[], int n) {
    int[] r = new int[n+1];
    r[0] = 0;
    int i,j;

    for(j=1; j<=n; j++) {
      int maximumRevenue = -1*INF;
      for(i=1; i<=j; i++) {
        maximumRevenue = max(maximumRevenue, c[i] + r[j-i]);
      }
      r[j] = maximumRevenue;
    }
    return r[n];
  }

  public static void main(String[] args) {
    // array starting from 1, element at index 0 is fake
    int c[] = {0, 10, 24, 30, 40, 45};
    System.out.println(bottomUpRodCutting(c, 5));
  }
}
```

Analysis of Rod Cutting

The analysis of the bottom up code is simple. We are using nested loops, the first loop is iterating from 1 to n and the second loop is iterating from 1 to j (j ranging from 1 to n). So, it will result in $\Theta(n^2)$ time. You can do the full analysis yourself if you are not convinced or can read the Analyze Your Algorithm (/course/algorithms-analyze-your-algorithm/) chapter to see the examples of analysis of algorithms. You can even raise a doubt in the discussion section (https://www.codesdope.com/discussion/) and find help from other members. The top-down approach also gives us a $\Theta(n^2)$ running time.

?

Among the algorithms we studied so far, you can see that our first task was to come up with a solution for the problem and then we were focusing on making a code for it and this is what one should do while dealing with an unknown problem that is, focus on finding a solution for the problem first.

(/add_ques

> Who you are tomorrow begins with what you do today.
>
> \- Tim Fargo

PREV (/course/algorithms-knapsack-problem/) (/course/algorithms-coin-change/) NEXT

?