Dynamic Programming

February 14, 2018

Contents

1 Introduction

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another. You can think as

1. Sending a recursive function to do different jobs.
2. Then, at the top of the recursion decide which job is the best one.

Actually the name comes from two notions:

- **Dynamic** was chosen by Bellman to capture the temporal part of the problem.
- **Programming** referred to finding the optimal program in military logistic.

2 Steps in Dynamic Programming

Dynamic programming follows a series of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

We will use the following rod cutting problem to exemplify the four steps.

3 Rod cutting

Given a rod of length n inches and a table of prices p_i (Example table 1) for $i = 1, 2, ..., n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Clearly, we can cut the rod in 2^{n-1} different ways starting at the left end with distance $i = 1, 2, ..., n-1$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Table 1: Rod Cutting table

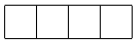
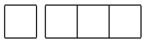
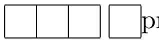
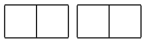
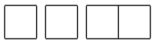
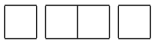
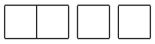
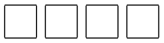
For example for a rod of size 10, we could cut the rod in 3 parts, 10=4+3+3. Then, we can assume that an optimal solution cuts the rod in k pieces, $1 \leq k \leq n$, then

$$n = i_{1,} + i_2 + ... + i_k$$

gives the maximum revenue

$$r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}.$$

For length $n = 4$, we have the following ways of partitioning the rood:

1.  price equal to 9
2.  price equal to 1+8
3.  price equal to 8+1
4.  price equal to 5+5
5.  price equal to 1+1+5
6.  price equal to 1+5+1
7.  price equal to 5+1+1
8.  price equal to 1+1+1+1

Note Can you device a recursion for the process? And which is the base case?

For $n = 4$, we have that it is possible, using a brute force approach, to obtain the solution. This brute force approach tells us that no so small pieces can be cut to obtain more revenue. Still, the problem become intractable by brute force once n become larger.

However, we can use the recursion property that we observed during the brute force approach to frame the final revenue by a series of decisions

$$r_n = \max\left\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1\right\}.$$

In this solution, we have the following parts:

- p_n corresponds to making no cuts at all.
- The other $n - 1$ arguments correspond to cut the rod in two pieces of size i and $n - i$.

This means that we need to solve smaller problems if we want to be able to solve the larger one. This means that the optimal solution incorporates the solutions of two related sub-problems. This means that the rod cut problem exhibits a **optimal substructure:**

- Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

Therefore, we can rewrite the previous equation as (Assuming $r_0 = 0$)

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

This can be implemented as (fig. 1).

```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

Figure 1: Cut Rod Code

This code has the following recursion:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0 \end{cases}$$

It can be proved by induction that $T(n) = 2^n$. Not so surprising when you consider that CUT-ROD is considering all the 2^{n-1}ways of cutting the rod. The recursive tree (fig.) for $n = 4$ shows those possible ways in the form of leaves.

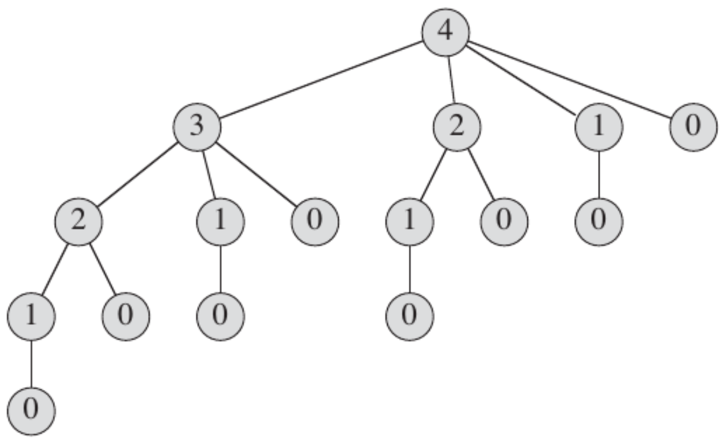


Figure 2: Recursion Tree for $n = 4$

Now, once we observed that the naive solution is clearly inefficient, we decide to use a dynamic programming approach. This is done by computing each subproblem only once and storing its solution in some way. This is known as **time-memory trade-off**, and the savings may be dramatic. It is more, a dynamic-programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

We have two ways of solving the problem:

- Top-down with memoization.
- Bottom-up.

These are the classical ways of solving dynamic programming problems.

3.1 Top-down with memoization.

Here, the solution is written in a recursive way, but before entering the recursion, it checks if the solution already exists. This is know as memoization. The cut rod solution for this is in (fig. 3).

```
MEMOIZED-CUT-ROD$(p, n)$
1  let $r[0..n]$ be a new array
2  for $i = 0$ to $n$
3      $r[i] = -\infty$
4  return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
1  if $r[n] \geq 0$
2      return $r[n]$
3  if $n == 0$
4      $q = 0$
5  else $q = -\infty$
6      for $i = 1$ to $n$
7          $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
8  $r[n] = q$
9  return $q$
```

Figure 3: Top-down solution for cut rod

This code has the following properties:

- It solves each subproblem just once. It solves subproblems for sizes $i = 0, 1, ..., n$
- To solve a problem of size i the for loop in line 6 of MEMOIZED-CUT-ROD-AUX iterates i times.

Then, we have the total number of steps is $0 + 1 + 2 + ... + n = \frac{n(n+1)}{2}$. Then, the code has complexity $\Theta\left(n^2\right)$.

3.2 Bottom-Up

The solution is even simpler (fig. 4).

```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

Figure 4: Bottom-Up Solution

From the two nested loop we realize that the complexity is $\Theta(n)$.

3.3 How to See Everything: Subproblem Graphs

In dynamic programing, it is necessary to understand how subproblems depend on each other. This information can be found in the subproblem graph (Example fig. 5). In it, a directed edge from subproblem x to subproblem y signify that in order to solve x, it is necessary to consider the solution of y.

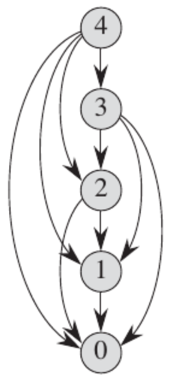


Figure 5: Subproblem Graph for $n = 4$

The fact that dynamic programming solves each problem once means that the total running time is the sum of the times needed to solve each subproblem. It is more, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph.

3.4 Reconstructing the Solution

Now, because we are already storing the solution of each subproblem, we store the path of solutions for the problem too. This can bee seen in (fig. 6).

```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```

Figure 6: Extended Version of Bottom-Up

With code in (fig.) to rebuild the stored solution

```
PRINT-CUT-ROD-SOLUTION($p, n$)
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

Figure 7: Print Solution

Now, we are ready too look for more interesting problems.

4 The Elements of Dynamic Programming

4.1 Optimal Substructure

As a first step toward the solution, in dynamic programming, is characterizing the problem and finding the optimal substructure. For this, we have the following steps

1. The problem consists in making choices.
2. Given each problem, you are given a choice that leads to a solution.

3. Each solution allows us to determine which subproblems need to be solved, and how to best characterize the resulting space of subproblems.

4. Use cut-and-paste to prove by contradiction that the optimal subproblem structure exists.

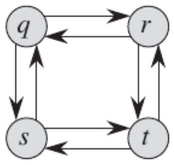
4.2 Subtleties of the proofs

For example, we have that

1. **Unweighted shortest path** has an optimal substructure:

   (a) Given an optimal shortest path t between p and q. Then, assume an intermediate point z such that there are two paths t_1 and t_2, $t = t_1 \cup t_2$. Thus, by contradiction, assume that there is a shorter path between z and q, t_2^1. Then, $\left|t_1 \cup t_2^1\right| < t \perp$ QED.

2. But the longest unweighted path does not have the optimal substructure by the simpler counter-example.



   This last part happens because the problems are not independent. This means that the problems are sharing resources in order to find a solution, in this case nodes.

4.3 Overlapping Sub-problems

This happens because the recursive solution revisits the same subproblem multiple times. Then, dynamic programming takes advantage of this by solving and storing the solution. This can be done in two fashions:

1. Bottom-Up - This takes the inefficient recursive solution and adds a memory to improve efficiency.

2. Top-Down with Memoization.

5 Matrix Multiplication

In this kind of problems, we are given a sequence of matrices:

Input $\langle A_1, A_2, ..., A_n \rangle$

Output, we want a fully parenthesized product, where the final result is a single matrix or the product of two fully parenthesized matrix products.

Now, consider the cost of the product of two matrices:

```
MATRIX-MULTIPLY($A, B$)
if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let $C$ be a new $A.rows \times B.columns$ matrix
    for $i = 1$ to $A.rows$
        for $j = 1$ to $B.columns$
            $c_{ij} = 0$
            for $k = 1$ to $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return $C$
```

This algorithm has a cost of, A is a matrix of pxq and B is a matrix of qxr, pqr operations.

Now, given the original problem, if we count the parenthezization using

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Solving this recursion, we get $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ lower bound.

5.1 The Optimal Substructure

Use the cut-and paste approach, by simple looking at one the full parentheza-tions and assume that you have something better. Then, by cut-and paste you find a contradiction.

1. Give $A_i, A_{i+2}, ..., A_k, A_{k+1}, ..., A_j$
2. Then, assume an optimal solution that is composed of two full parenthez-izations for $A_i, A_{i+2}, ..., A_k$ and $A_{k+1}, ..., A_j$.
3. Then, assume that you can find something better for $A_{k+1}, ..., A_j$.
4. Use that to build a more optimal solution.
5. This is a contradiction.

5.2 The Recursive Solution

When looking at the problem, we realized that we have

1. $m[i,k]$ cost to solve one side of the problem.
2. $m[k+1,j]$ for the other
3. $p_{i-1}p_kp_j$ to multiply the resulting matrices

Therefore, we have the following cost $m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$. The final recursive solution is in the slides, and an initial recursive solution is the following one:

```
RECURSIVE-MATRIX-CHAIN($p, i, j$)
1  if $i == j$
2      return 0
3  $m[i, j] = \infty$
4  for $k = i$ to $j - 1$
5      $q$ = RECURSIVE-MATRIX-CHAIN($p, i, k$)
             + RECURSIVE-MATRIX-CHAIN($p, k + 1, j$)
             + $p_{i-1} p_k p_j$
6      if $q < m[i, j]$
7          $m[i, j] = q$
8  return $m[i, j]$
```

This yields the following bounded recurrence.

$$\begin{aligned} T(1) &\geq 1 \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} \left[T(n-k) + T(k) + 1\right] \text{ for } n > 1. \end{aligned}$$

Which can be proved by substitution to be really bad, $T(n) = \Omega\left(2^n\right)$. Better if we look at the Bottom-Up Approach!!!

5.3 The Bottom-Up Approach and the Reconstruction

They are at the slides in the class.

6 Longest Common Subsequence (LCS)

In Biological Sciences, we want to compare strands of DNA to see how similar they are. A measure of similarity is one is a substring of the other one. In our case, we look at the longest common between to strings under an alphabet.

6.1 Characterizing the LCS

For this, we have the proof for Theorem 15.1

Proof:

1. if $z_k \neq x_m$, then we could append $x_m = y_n$to Z to build a longest sequence of $k+1$ elements. This contradict that Z is the longest sequence. Therefore, $x_m = y_n = z_k$ and $Z_{k-1}$is a common subsequence of X_{m-1} and Y_{n-1}. We prove by contradiction that Z_{k-1}. This is done by assuming a W sequence with at least k elements on it, and again we create a longer subsequence than Z by appending $x_m = y_n$ to it, which is a contradiction.

2. If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. Again by contradiction, if there a W longest than Z, this contradict that Z is LCS of Y and X.

3. By Symmetry with 2.

The rest is at the slides.