Dynamic Programming

Adam W. Bargteil

February 3, 2022

We typically apply dynamic programming to **optimization problems**. We follow a sequence of 4 steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of the optimal solution.
4. Construct the optimal solution from computed information.

Rod Cutting Problem: Given a rod of length n and a table pf prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting the rod and selling the pieces. There are 2^{n-1} different ways to cut the rod since we can choose to include or not include cut i.

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1}, r_1).$$

Optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independantly.

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

Cut-Rod (p, n)

1: **if** n == 0 **then**
2: **return** 0
3: **end if**
4: $q = -\infty$
5: **for** $i = 1$ **to** n **do**
6: $q = \max(q, p[i] +$ Cut-Rod $(p, n - i))$
7: **end for**
8: **return** q

Easy to prove correct by induction. Running time?

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n$$

Both $\Theta(n^2)$.

Subproblem Graph: A "reduced" or "collapsed" version of call tree. Bottom-up uses reverse topological sort. Typically, the time to compute the solution of a subproblem is proportional to out-degree and the number of subproblems is the number of vertices.

Memoized-Cut-Rod (p, n)

```
1: r = new int[n]
2: for i = 0 to n do
3:     r[i] = −∞
4: end for
5: return Memoized-Cut-Rod-Aux (p, n, r)
```

Memoized-Cut-Rod-Aux (p, n, r)

```
1: if r[n] ≥ 0 then
2:     return r[n]
3: end if
4: if n == 0 then
5:     q = 0
6: else
7:     q = −∞
8: end if
9: for i = 1 to n do
10:    q = max(q, p[i]+ Memoized-Cut-Rod-Aux (p, n − i))
11: end for
12: r[j] = q
13: return r[n]
```

Bottom-Up-Cut-Rod (p, n)

```
1: r = new int[n]
2: r[0] = 0
3: if n == 0 then
4:     return 0
5: end if
6: q = −∞
7: for j = 1 to n do
8:     q = −∞
9:     for i = 1 to j do
10:        q = max(q, p[i] + r[j − 1])
11:        r[j] = q
12:    end for
13: end for
14: return r[n]
```

Extended-Bottom-Up-Cut-Rod (p, n)

```
1:  r = new int[n]
2:  s = new int[n]
3:  r[0] = 0
4:  if n == 0 then
5:      return 0
6:  end if
7:  q = -∞
8:  for j = 1 to n do
9:      q = -∞
10:     for i = 1 to j do
11:         if q < p[i] + r[j - i] then
12:             q = p[i] + r[j - i]
13:             s[j] = i
14:         end if
15:         r[j] = q
16:     end for
17: end for
18: return r[n]
```

Print-Cut-Rod-Solution(p, n)

```
1:  (r, s) = Extended-Bottom-Up-Cut-Rod(p, n)
2:  while n > 0 do
3:      print s[n]
4:      n = n - s[n]
5:  end while
```

Matrix-chain Multiplication: Given a sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications. Matrix multiplaction rules: $p \times q$ times $q \times r$ gives $p \times r$, cost pqr.

Number of possible parenthesizations, $P(n)$,

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k).$$

Catalan numbers $\Omega(4^n/n^{3/2})$.

Dynamic Programming:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of the optimal solution.
4. Construct the optimal solution from computed information.

Structure: $A_{i..j} = A_i A_{i+1} \ldots A_j$. We must break the problem up into $A_{i..k}$ and $A_{k+1..j}$ for some k. The cost is computing $A_{i..k}$ and $A_{k+1..j}$ and then multiplying them together.

Optimal substructure: the optimal paranthesization of the subchain $A_{i..k}$ within the optimal parenthesization of $A_{i..j}$ must be an optimal parenthesization of $A_{i..k}$. Because if there were a better way to parenethesize $A_{i..k}$ then we would substitute that and find a better parenthesization of $A_{i..j}$.

Given optimal substructure, we can build the optimal solution by optimally splitting the problem into two subproblems.

Let $m[i, j]$ be the **minimum** cost of computing $m[i, j]$.

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

$$m[i, j] = \min_{i \le k < j} m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

Number of subproblems: one for each choice of i, j s.t. $1 \le i \le j \le n = \binom{n}{2} + n = \Omega(n^2)$. **Overlapping subproblems.**

Elements of Dynamic Programming:

Optimal Substructure: Characterize the structure of an optimal solution. **Optimal Substructure** if an optimal solution contains optimal solutions to subproblems.

1. Solution requires a choice that results in subproblems.
2. Assume you are given the choice that leads to an optimal solution.
3. determine ensuing subproblems and how to characterize the space of subproblems.
4. Show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by the "cut-and-paste" technique.Assume the the subrpoblem solutions are not optimal and derive contradiction. "Cut out" the nonoptimal solution to subproblem, "paste in" the optimal one and shoe that you get a better solution resulting in a contradiction, since we assumed an optimal solution.

Matrix-Chain-Order(p)

```
1:  n = p.length − 1
2:  m = newint[1..n][1..n]
3:  s = newint[1..n − 1][2..n]
4:  for 1 = 1 to n do
5:      m[i, i] = 0
6:  end for
7:  for l = 2 to n do
8:      for i = 1 to n − l + 1 do
9:          j = i + l − 1
10:         m[i, j] = ∞
11:         for k = i to j − 1 do
12:             q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_k
13:             if q < m[i, j] then
14:                 m[i, j] = q
15:                 s[i, j] = k
16:             end if
17:         end for
18:     end for
19: end for
20: return m and s
```

Print-Optimal-Parens(p, i, j)

```
1:  if i == j then
2:      print "A_i"
3:  else
4:      print "("
5:      Print-Optimal-Parens(s, i, s[i, j])
6:      Print-Optimal-Parens(s, i[i, j] + 1, j])
7:      print ")"
8:  end if
```

Keep the space of subproblems as simple as possible. We could work from the left on rod cutting, but had to allow "both sides" for matrix-mult.

Optimal substructure varies in two ways:

1. how many subproblems an optimal solution uses
2. how many choices we have in determining which subproblems to use in an optimal solution

Rod-cutting: one subproblem, but n choices. Matrix-chain-mult: 2 subproblems, $j - i$ choices.

Running time depends on two factors: number of subproblems and choices for each subproblem. Rod cutting $\Theta(n)$ subproblems, and n choices for each, $O(n^2)$. Matrix-chain had $\Theta(n^2)$ subproblems and $n - 1$ choices, $O(n^3)$

Subproblem graph gives alternate analysis, each node is a subproblem and each edge is a subproblem we must consider.

Cost of a problem is the cost of subproblems plus the cost of the decision itself.

Optimal substructure does not always apply. Unweighted longest simple path: The path u to w includes v, but the longest simple path from u to v is not necessarily part of the path from u to w because there could be edges in the longest simple path from u to v that are **also** on the longest simple path from v to w, creating a cycle. The subproblems must be **indpendent**.

Overlapping Subproblems: The space of subproblems is "small," polynomial in the problem size and a recursive algorithms solves the same subproblems repeatedly. Divide-and-conquer usually generates **new** subproblems at each step of recursion.

It is important to note, we only care about the *optimal* solution.

We often have to store additional information to reconstruct the optimal solution.

Longest-common-subsequence problem : given two sequences $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ find a maximum-length common subsequence of X and Y.

Naive: check whether every subsequence of X is in Y. There are an exponential number of subsequences of X.

Define **prefix**, X_i as the first i entries in X.

Given X and Y, let Z be any LCS.

1. if $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. if $x_m \neq y_n$ then $z_k \neq x_m$ and Z_{k-1} is an LCS of X_{m-1} and Y
3. if $x_m \neq y_n$ then $z_k \neq y_n$ and Z_{k-1} is an LCS of X and Y_{n-1}

If $z_k \neq x_m$ then we could append $x_m = y_n$ to Z to obtain a longer subsequence, contraditing that Z is longest. Z_{k-1} is a subsequence of X_{m-1} and Y_{n-1} (length $k - 1$). Show it is an LCS. Suppose (for contradiction) that there is a common subsequence W of X_{m-1} and Y_{n-a} greater length than $k - 1$. Then appending $x_m = y_n$ produces a common subsequence greater than k, which is a contradiction.

If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

Symmetry for (3).

Let $c[i, j]$ be the length of the LCS of X_i and Y_i.

$$c[i, j] = c[i - 1, j - 1] + 1$$

if $x_i = y_i$, otherwise

$$c[i, j] = \max(c[i, j - 1], c[i - 1, j])$$

LCS-Length(X, Y)

```
1:  m = X.length
2:  n = Y.length
3:  b = new int[1..m][1..n]
4:  c = new int[0..m][0..n]
5:  for 1 = 1 to m do
6:      c[i, 0] = 0
7:  end for
8:  for 1 = 1 to n do
9:      c[o, j] = 0
10: end for
11: for i = 1 to m do
12:     for i = 1 to n do
13:         if x_i == y_j then
14:             c[i, j] = c[i − 1, j − 1] + 1
15:             b[i, j] = 0
16:         else if c[i − 1, j] ≥ c[i, j − 1] then
17:             c[i, j] = c[i − 1, j]
18:             b[i, j] = 2
19:         else
20:             c[i, j] = c[i, j − 1]
21:             b[i, j] = 2
22:         end if
23:     end for
24: end for
25: return c and b
```

Print-LCS(b, X, i, j)

```
1:  if i == 0 or j == 0 then
2:      return
3:  end if
4:  if b[i, j] == 0 then
5:      Print-LCS(b, X, i − 1, j − 1)
6:      print x_i
7:  else if b[i, j] = 1 then
8:      Print-LCS(b, X, i − 1, j)
9:  else
10:     Print-LCS(b, X, i, j − 1)
11: end if
```
