Dynamic Programming

Dynamic Programming is a programming technique that can dramatically reduces the runtime of some algorithms (but not all problem has DP characteristics) from exponential to polynomial. Many (and still increasing) real world problems only solvable within reasonable time using DP.

To be able to use DP, the original problem must have:

1. Optimal sub-structure : Optimal solution to the problem contains within it optimal solutions to sub-problems

2. Overlapping sub-problems : We accidentally recalculate the same problem twice or more.

There are 2 types of DP: We can either build up solutions of sub-problems from small to large (bottom up) or we can save results of solutions of sub-problems in a table (top down + memoization).

Let's start with a sample of Dynamic Programming (DP) technique. We will examine the simplest form of overlapping sub-problems. Remember Fibonacci? A popular problem which creates a lot of redundancy if you use standard recursion $f_n = f_{n-1} + f_{n-2}$.

Top-down Fibonacci DP solution will record the each Fibonacci calculation in a table so it won't have to re-compute the value again when you need it, a simple table-lookup is enough (memorization), whereas Bottom-up DP solution will build the solution from smaller numbers.

Now let's see the comparison between Non-DP solution versus DP solution (both bottom-up and top-down), given in the C source code below, along with the appropriate comments

```
// the slowest, unnecessary computation is repeated
 int Non_DP(int n) {
 if (n==1 || n==2)
    return 1;
 else
    return Non_DP(n-1) + Non_DP(n-2);
 }

 // top down DP
 int DP_Top_Down(int n) {
 // base case
 if (n == 1 || n == 2)
    return 1;

 // immediately return the previously computed result
 if (memo[n] ≠ 0)
    return memo[n];

 // otherwise, do the same as Non_DP
 memo[n] = DP_Top_Down(n-1) + DP_Top_Down(n-2);
 return memo[n];
```

```
}

// fastest DP, bottom up, store the previous results in array
int DP_Bottom_Up(int n) {
memo[1] = memo[2] = 1; // default values for DP algorithm

// from 3 to n (we already know that fib(1) and fib(2) = 1
for (int i=3; i ≤ n; i++)
    memo[i] = memo[i-1] + memo[i-2];

return memo[n];
}
```

Matrix Chain Multiplication (MCM)

Let's start by analyzing the cost of multiplying 2 matrices:

```
Matrix-Multiply(A,B):
 if columns[A] ≠ columns[B] then
      error "incompatible dimensions"
 else
      for i = 1 to rows[A] do
           for j = 1 to columns[B] do
                C[i,j]=0
                for k = 1 to columns[A] do
                     C[i,j] = C[i,j] + A[i,k] * B[k,j]
 return C
```

Time complexity = O(pqr) where |A| = p x q and |B| = q x r. The result is matrix C with |C| = p x r.

Matrix Chain Multiplication Problem

Input: Matrices $A_1, A_2, \ldots, A_n$, each A_i of size $P_{i-1} \times P_i$

Output: Fully parenthesized product $A_1.A_2 \ldots A_n$ that minimizes the number of scalar multiplications.

A product of matrices is fully parenthesized if it is either

(1.) a single matrix

(2.) the product of 2 fully parenthesized matrix products surrounded by parentheses

Example of MCM problem:

We have 3 matrices and the size of each matrix:

A_1 (10 x 100), A_2 (100 x 5), A_3 (5 x 50)

We can fully parenthesized them in two ways:

(1.) $(A_1 (A_2 A_3))$ = 100 x 5 x 50 + 10 x 100 x 50 = 75000

(2.) $((A_1 A_2) A_3)$ = 10 x 100 x 5 + 10 x 5 x 50 = 7500 (10 times better)

See how the cost of multiplying these 3 matrices differ significantly. The cost truly depend on the choice of the fully parenthesization of the matrices. However, exhaustively checking all possible

parenthesizations take exponential time.

Now let's see how MCM problem can be solved using DP.

Step 1: characterize the optimal sub-structure of this problem.

Let $A_{i..j}$ (i < j) denote the result of multiplying $A_iA_{i+1}..A_j$. $A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the subproducts. There are j-i possible splits (i.e. k=i,...,j-1)

Within the optimal parenthesization of $A_{i..j}$:

(a) the parenthesization of $A_{i..k}$ must be optimal

(b) the parenthesization of $A_{k+1..j}$ must be optimal

Because if they are not optimal, then there exist other split which is better, and we should choose that split and not this split.

Step 2: Recursive formulation

Our final solution is $A_{1..n}$. So we need to find $A_{1..n}$

Let m[i,j] = minimum number of scalar multiplications needed to compute $A_{i..j}$

Since $A_{i..j}$ can be obtained by breaking it into $A_{i..k}$ $A_{k+1..j}$, we have

$$m[i,j] = \begin{cases} 0 & ; \text{if } i=j \\ \min\limits_{k=i \text{ to } j} \{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & ; \text{if } i<j \end{cases}$$

let s[i,j] be the value k where the optimal split occurs for $A_{i..j}$.

Step 3: Computing the Optimal Costs

```
Matric-Chain-Order(p)
n = length[p]-1
for i = 1 to n do
    m[i,i] = 0
    for r = 2 to n do
        for i = 1 to n-r+1 do
            j = i+r-1
            m[i,j] = ∞
            for k = i to j-1 do
                q = m[i,k] + m[k+1,j] + pi-1*pk*pj
                if q < m[i,j] then
                    m[i,j] = q
                    s[i,j] = k
return m and s
```

Step 4: Constructing an Optimal Solution

```
Print-MCM(s,i,j)
if i=j then
    print Ai
else
    print "(" + Print-MCM(s,1,s[i,j]) + "*" + Print-MCM(s,s[i,j]+1,j) + ")"
```

Note: As any other DP solution, MCM also can be solved using Top Down recursive algorithm using memoization. Sometimes, if you cannot visualize the Bottom Up, approach, just modify your original Top Down recursive solution by including memoization. You'll save a lot of time by avoiding repetitive calculation of sub-problems.

Source:

http://www.learnalgorithms.in/#