Dynamic Programming

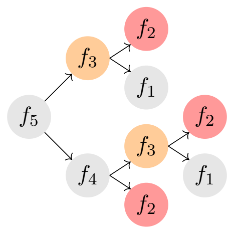
Overview

1. Background
2. Framework
3. Examples

Background

Dyanmic Programming (DP) applies to **optimization** problems. It is used to solve many problems in polynomial time. $O(n^c)$ **instead** of $O(c^n)$. To use Dynamic Programming, a problem must have

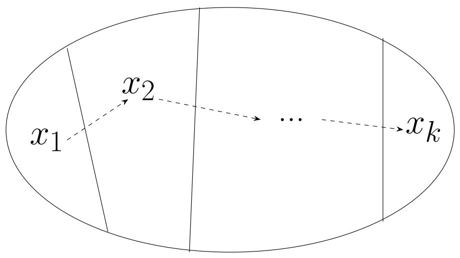
- Overlapping Subproblems



Divide and Conquer is very innefficient when the 'smaller' problem is nearly as large as the original. DP is similar but smaller problems are solved first and then stored for later use.

- Optimal Substructure

Optimal Substructure is a powerful principle that says the optimal solution of a bigger problem can be created from the optimal solutions of its smaller subproblems. It allows us to break a difficult problem into small manageable parts, and combine the results of these small problems to get the best solution to the overall problem.



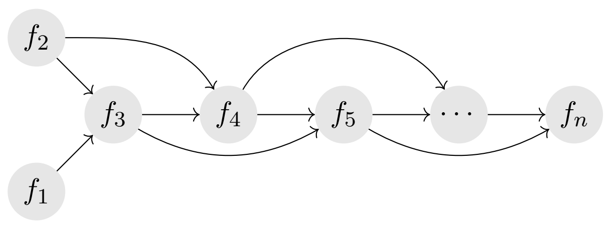
Framework

1. Define the objective function
2. Identify the Base cases (Starting points)
3. Recurrence Relation. Subproblem $\Rightarrow$ Problem
4. Order of computation.
5. Location of the answer, F(n).

The staircase problem is a classic example for the application of Dynamic Programming.
Problem: You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

1. Objective Function
   (a). f(n) = the number of distinct ways to reach the n^{th} step.
2. Base cases
   (a). $f(0) = 1$, as there is only one way to take no steps.
   (b). $f(1) = 1$, as there is only one way to take one step.
   (c). $f(2) = 2$, as you can either take two single steps, or one double step
3. Recurrence Relation
   (a). $f(n) = f(n-1) + f(n-2)$
4. Order of computation
   (a). Bottom-up
5. Location of the answer
   (a). Where to look - $f(n)$.
   (b). Time complexity - $O(n)$
   (c). Space - $O(n)$

The bottom-up approach is being used as each value requires the previous two values to be found. Instead of recomputing them every time like we would with a top-down approach, we can store the previous values. This technique of storing previous values to find the next is called **Memoization**.


$$\begin{tikzcd}
f_2 \arrow[dr] \arrow[drr, bend left] & & & & & \\
& f_3 \arrow[r] \arrow[rr, bend right] & f_4 \arrow[r] \arrow[rr, bend left] & f_5 \arrow[r] \arrow[rr, bend right] & \cdots \arrow[r] & f_n \\
f_1 \arrow[ur] & & & & &
\end{tikzcd}$$

Examples

Fibonacci Sequence

Consider the fibonacci sequence, where the two previous numbers in the sequence are used to calculate the next number.

$$1 + 1 = 2 + 1 = 3 + 2 = 5 + 3 = 8 + 5 = 13 \cdots$$

The Problem: Given a number n, find the n^{th} term in the fibonacci sequence.

This problem is a classic example of how applying dynamic-programming can make finding a solution go from factorial (binomial) time down to polynomial time.

Recursive (Exponential Solution)

As shown above, calculating the n^{th} fibonacci number recursively is very innefficient. Using the Recurrence Relation for it, we can solve for the explicit solution, and find the time complexity for the recursive algorithm

Algorithm 1 Recursive Fibonacci

```
Input
$n \leftarrow$ the $n^{th}$ term in the sequence
Function fib(n) $\rightarrow x \in \mathbb{Z}$
if $n \leq 2$ then
    return 1;
else
    return fib(n-1) + fib(n-2);
end if
```

Since this reursive algorithm has one operation for every call of the function, it's operations can be quantified by the recurrence relation

$$T(n) = T(n-1) + T(n-2)$$

$$T(0) = 0 \qquad T(1) = 1 \qquad T(2) = 1$$

We can then rearrange mathematically to find the explicit solution, and time complexity for our recursive solution.

First rearrange into a Linear Homogenous Recurrence Relation with Constant Coefficients (LHRRCC)

$$T(n) - T(n-2) - T(n-2) = 0$$

Then find the roots using the quadratic formula on the characteristic equation

$$x^2 - x^1 - 1 = 0$$
$$x_1 = \frac{1+\sqrt{5}}{2}$$
$$x_2 = \frac{1-\sqrt{5}}{2}$$

Then apply those roots to the General Solution Equation

$$f_n = Ax_1^n + Bx_2^n$$

Now solve for A and B

$$\begin{aligned}
f_0 = 0 &= A(1) + B(1) & \therefore A = -B \\
f_1 = 1 &= A\left(\frac{1+\sqrt{5}}{2}\right) + B\left(\frac{1-\sqrt{5}}{2}\right) \\
1 &= A\left(\frac{1+\sqrt{5}}{2}\right) - A\left(\frac{1+\sqrt{5}}{2}\right) \\
A &= \frac{1}{5} \;\; B = -\frac{1}{5}
\end{aligned}$$

Which means the explicit solution is

$$T(n) = \frac{1}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

This proves that the recursive approach is an exponential function. An upperbound for the function could be described as $O(2^n)$.

Dynamic Programming Solution

The Fibonacci sequence can be solved using dynamic programming (DP) because it has these charaacteristics

- Optimization Problem (Factorial time)
- Optimal Subproblem Structure (Overlapping subproblems).

1	1	2	3	5	8	13	···

Instead of doing the work from the top-down, and repeating calculations, the previous values should be stored so that future elements can access them instead of recalculating it.

This can be done using an array with the fibonacci sequence. Then, use the recursive definition but instead of recalculating, access the array for the previous values.

$$arr[i] = arr[i-1] + arr[i-2];$$

Note The following solution uses an array to store previous values, which has a space complexity of $O(n)$. However, the optimal solution can use constant space $O(1)$ if only the previous two values were stored with a *Sliding Window* technique.

Algorithm 2 Dynamic Programming Fibonacci

```
**Input**
$n \leftarrow$ the $n^{th}$ term in the sequence
**Initialize**
$f = []$;                                  ▷ Array to store previous values
$f[0] = 1$;
$f[1] = 1$;
**for** $i = 2 : n$ **do**
    $f[i] = f[i-1] + f[i-2]$;
**end for**
**return** $f[n]$;
```

Every-Case Time Analysis

For this algorithm we have one loop, $n = [2 : n]$ with one operation. this can be expressed by

$$\sum_{i=2}^{n} 1 = O(n)$$

Summary

We saw there is an optimal way to calculate the n^{th} turn in the fibonacci sequence. We used dynamic programming to find the solution. The overall DP process is to:

1. Establish a recursive property. In this case,

$$T(n) = T(n-1) + T(n-2)$$

$$T(0) = 0 \qquad T(1) = 1 \qquad T(2) = 1$$

2. Compute the previous values of the sequence in a bottom-up fashion utilizing memoization.
3. Compute the solution.

We saw the time complexity was polynomial, $O(n)$, whereas without DP, it was exponential $O(2^n)$.

Matrix Chain Multiplication

Consider the product of three matrices, $A_1A_2A_3$, with dimensions $d_1 \times d_2$, $d_2 \times d_3$, and $d_3 \times d_4$, respectively. To multiply matrices, A_1A_2 (see Algorithm 1.4 in Section 1.1 on page 9), the dimensions of the products must be compatible, meaning the number of columns of A_1 has to equal the number of rows of A_2. Recall that matrix multiplication is associative, however, *the order in which they are multiplied matters.*

$$A_1(A_2A_3) = (A_1A_2)A_3.$$

For example is A_1 is a 3×2, A_2 is a 2×6, and A_3 is a 6×10, then there are 180 multiplications in $A_1(A_2A_3)$ and 216 for $(A_1A_2)A_3$.

The Problem: Given n matrices, in what order should we multiply the factors?

This problem of determining the optimal number of multiplications in a matrix requires examining how to associate the matrices in the product.

Catalan Number (Binomial Solution)

Catalan numbers arise in several unrelated counting problems. One gives the number of fully parenthesized products. It is the number of ways $n+1$ factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator, as in the matrix chain multiplication problem). The n-th Catalan number

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Theorem: The number of sequences

$$a_1, a_2, a_3, \cdots, a_{2n}$$

of $2n$ terms that can be formed with $n+1$'s and $n-1$'s whose partial sums satisfy

$$a_1 + a_2 + \cdots + a_k \geq 0, \qquad \text{for } k = 1, 2, \ldots, 2n$$

equals the nth Catalan number,

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Proof: Among all the sequences of $a_1, a_2, a_3, \cdots, a_{2n}$, there are acceptable ones and ones that are unacceptable. For example, $(+1)(+1)(-1)(-1)(+1)(-1)(-1)$ is unacceptable. We denote A_n as the acceptable sequences and U_n as unacceptable sequences. An **unacceptable** sequence must have a smallest k such that,

$$a_1 + a_2 + \cdots + a_{k-1} = 0$$

and

$$a_k = -1.$$

Note that k must be odd. By mapping each $a_i \to (-a_i)$ for $i = 1, 2, \ldots, k$ and leave unchanged the remaining terms $k+1, \ldots, 2n$. The resulting sequence has $(n+1)\ +1$'s and $(n-1)\ -1$'s.

Note there are only n matrices (or factors), yet the expression above is for $n+1$ factors. For n matrices, there are $n-1$ multiplications. Therefore, replace the formula above by substituting $n-1$ for n. That is,

$$C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1}$$

Dynamic Programming Solution

The matrix chain problem can be solved using dynamic programming (DP) because it is a problem with these characteristics:

- Optimization Problem (Factorial Time)
- Optimal Subproblem Structure

Example: Let $d_1 = 3, d_2 = 20, d_3 = 10, d_4 = 50$, and $d_5 = 4$. Consider the matrix product,

$$\underset{3\times 20}{A_1} \times \underset{20\times 10}{A_2} \times \underset{10\times 50}{A_3} \times \underset{50\times 4}{A_4} = \underset{d_1\times d_2}{A_1} \times \underset{d_2\times d_3}{A_2} \times \underset{d_3\times d_4}{A_3} \times \underset{d_4\times d_5}{A_4}$$

There are five products:

1. $A_1(A_2(A_3A_4))$
2. $A_1((A_2A_3)A_4)$
3. $(A_1A_2)(A_3A_4)$
4. $(A_1(A_2A_3))A_4$
5. $((A_1A_2)A_3)A_4$

The number of multiplication of each of these products (using brute force) is:

1. $10 \cdot 50 \cdot 4 + 20 \cdot 10 \cdot 4 + 3 \cdot 20 \cdot 4 = 3040$
2. $20 \cdot 10 \cdot 50 + 20 \cdot 50 \cdot 4 + 3 \cdot 20 \cdot 4 = 14240$
3. $3 \cdot 20 \cdot 10 + 10 \cdot 50 \cdot 4 + 3 \cdot 10 \cdot 4 = 2720$
4. $20 \cdot 10 \cdot 50 + 3 \cdot 20 \cdot 50 + 3 \cdot 50 \cdot 4 = 13600$
5. $3 \cdot 20 \cdot 10 + 3 \cdot 10 \cdot 50 + 3 \cdot 50 \cdot 4 = 2700$

The least number of multiplications is the fifth grouping: $((A_1A_2)A_3)A_4$.

Notice how $A_3 \times A_4$ was calculated in the first and the third products, repeating this calculation. We should save this calculation (*memoization*) when we compute it in the first computation and use it in the third. Similar for other pairs such as $A_2 \times A_3$.

Define the $n \times n$ memoization array, M, where $M(i,j)$ stores the minimum number of multiplications in the subproblem product $A_i \cdots A_j$. In the example above, we would have

$$M(2,4) = \min\{20 \cdot 10 \cdot 50 + 20 \cdot 50 \cdot 4, 10 \cdot 50 \cdot 4 + 20 \cdot 10 \cdot 4\} = \min\{14000, 2800\} = 2800$$

In general,

$$M(i,j) = \min_{i \le k \le j-1} \{M(i,k) + M(k+1,j) + d_i d_{k+1} d_{j+1}\}$$

Note The optimal solution to $A_1A_2A_3A_4$ contains $(A_1A_2)A_3$, which is the optimal solution to $A_1A_2A_3$.

Every-Case Time Analysis

We have $d = [1 \cdots n-1]$, $i = [1 \cdots n-d]$, and $k = [i \cdots j-1]$, (or $k = [i...(i+d-1)]$ since $j = [i+d]$). Therefore, we have,

$$\sum_{d=1}^{n-1} (n-d-1+1)(i+d-1-i+1) = \sum_{d=1}^{n-1} (n-d)(d)$$

Simplifying and applying the formula for the sum of squares, $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, we have

$$\sum_{d=1}^{n-1} nd - d^2 = n\frac{(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6} = \frac{3n^3 - 3n^2 - 2n^3 + 3n^2 - n}{6}$$

$$= \frac{n^3 - n}{6} = \frac{n(n^2-1)}{6} = \frac{n(n+1)(n-1)}{6}$$

Algorithm 3 Matrix Chain Algorithm

```
Input:
$D \leftarrow$ Dimensions of the matrices $(d_1, d_2, d_3, ..., d_{n+1})$   ▷ $size(A1) = d_1 \times d_2$, $size(A2) = d_2 \times d_3, \cdots$
Function matchain2(D) $\rightarrow [M, P]$
Initialize:
$n \leftarrow size(d)$
$M \leftarrow [][]$                                   ▷ Initialized with all zeros
$P \leftarrow [][]$                                   ▷ Initialized with all zeros
for $d = [1 : n-1]$ do
    for $i = [1 : n-d]$ do
        $j = i + d$;
        $m = \infty$;
        for $k = [i : j-1]$ do
            if $(m > M(i,k) + M(k+1,j) + D(i) * D(k+1) * D(j+1))$ then
                $m = M(i,k) + M(k+1,j) + D(i) * D(k+1) * D(j+1)$;
                $P(i,j) = k$;
            end if
        end for
        $M(i,j) = m$;
    end for
end for
```

Print parenthesization using P

The entry in $P(i,j)$ is where the matrices should be split. In the example in the book, $P(1,6) = 1$, therefore, $A_1A_2A_3A_4A_5A_6 = A_1(A_2A_3A_4A_5A_6)$. $P(2,6) = 5$, the product should be split at A_5, i.e., $(A_2A_3A_4A_5A_6) = (A_2A_3A_4A_5)A_6$. Combining, so far we have,

$$A_1((A_2A_3A_4A_5)A_6.$$

The product $(A_2A_3A_4A_5) = (A_2A_3A_4)A_5$ since $P(2,5) = 4$. This gives,

$$A_1((A_2A_3A_4)A_5)A_6.$$

Finally, $P(2,4) = 3$, so $A_2A_3A_4 = (A_2A_3)A_4$. The final order is:

$$A_1(((A_2A_3)A_4)A_5)A_6.$$

Summary

We saw there is an optimal way to multiply a chain of matrices. We used dynamic programming to find the solution. The overall DP process is to:

1. Establish a recursive property that gives the optimal solution to an instance of the problem. In this case,

$$M(i,j) = \min_{i \le k \le j-1} \{M(i,k) + M(k+1,j) + d_i d_{k+1} d_{j+1}\} \quad \text{for } i < j$$
$$M(i,i) = 0$$

Algorithm 4 Print Optimal Order

```
Inputs:
$P \leftarrow$ [][] - Matrix of the optimal split points
$i \leftarrow$ starting index of the matrix chain
$j \leftarrow$ ending index of the matrix chain
Function printorder(P,i,j) $\rightarrow y =$ String with optimal parenthesization.
if $i == j$ then
    $y =$ strcat('A', num2str(i));
else
    $k \leftarrow P(i,j)$
    $y =$ strcat(y, ')')
    $y =$ strcat(y, printorder(P,i,k))
    $y =$ strcat(y, printorder(P,k+1,j))
    $y =$ strcat(y, '(')
end if
return y
```

2. Compute the value of an optimal solution in a bottom-up fashion utilizing memoization.
3. Compute the optimal solution.

We saw the time complexity was polynomial, $O(n^3)$, whereas without DP, it was at least exponential. In particular, let $T(n)$:= the number of different orders in which we could multiply n matrices. Then we have,

$$T(n) \geq 2T(n-1)$$

since, $T(n-1)$:= number of different orders of multiplying $A_2 \cdots A_n$ and $A_1 \cdots A_{n-1}$.