

Proof of Claim

Given ALG we construct ALG_i .

Given a set of pages X and a page p , let $X + p$ denote the set $X \cup \{p\}$.

Without loss of generality, assume that the i th request resulted in a page fault.

Assume that after processing the i th request, let the caches of ALG and ALG_i contain the page sets $X + v$ and $X + u$ respectively, where X is some set of $k - 1$ common pages and v and u are any pages.

Assume that $u \neq v$, otherwise, ALG_i simply mimics ALG , i.e., makes the same evictions as ALG .

We consider two cases:

Case 1: Till v is requested: ALG_i mimics ALG to serve subsequent requests. If ALG evicts v then ALG_i evicts u , unless u itself has been requested. The number of common pages become k and ALG_i mimics ALG from now on.

Also, number of page misses of ALG_i is at most that of ALG .

Case 2: If v is requested and assume that ALG and ALG_i have not yet identified. ALG_i will evict u and the two algorithms identify. But, ALG_i will incur a page fault and ALG will not.

However, by the time v was requested, there must have been at least one request for u after the i th page fault. The first such request incurs a page fault to ALG but not to ALG_i . Thus the total number of page faults for ALG_i after servicing v is equal to that for ALG .

Dynamic Programming

A powerful technique for solving **optimization** problems. Useful when the solution to a problem can be viewed as a result of a sequence of decisions.

For example, the solution to the Knapsack problem can be viewed as a sequence of decisions, namely, deciding the values of x_i , $1 \leq i \leq n$. If x_i 's are restricted to be 0/1, then a greedy solution will not work.

One way to solve such problems is to enumerate all possible decision sequences and pick the best. Dynamic programming often drastically reduces the amount of enumeration by appealing to the **Principle of Optimality**:

“An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.”

Ingredients of Dynamic Programming

1. **Optimal substructure:** The optimal solution is computed from optimal solutions to **independent** sub-problems.

The solution to one subproblem does not affect the solution to another subproblem of the same problem.

2. **Overlapping** subproblems.

The same subproblem occurs in the solution of many different (sub)-problems.

3. **Dynamic Programming:** Write a **recursive** formula to evaluate the optimal value. Solve the recursion in a **bottom-up** manner.

4. **Memoization:** We compute the solution of a subproblem **only once** and then storing it in a table and doing a **look up** for subsequent computations of the **same** subproblem. A **top-down strategy**.

Matrix Multiplication

Given a chain of n matrices: $\langle A_1, A_2, \dots, A_n \rangle$ where matrix A_i has dimension $p_{i-1} \times p_i$, for $i = 1, 2, \dots, n$.

Find a way to paranthesize the product $A_1 A_2 \dots A_n$ which **minimizes** the number of multiplications.

Recursive solution

For $1 \leq i \leq j \leq n$, let $m[i, j]$ be the minimum cost of paranthesizing the product $A_i A_{i+1} \dots A_j$.

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

A Naive Algorithm

cost(p, i, j)

Input: A sequence of matrix dimensions: $p = \langle p_0, p_1, \dots, p_n \rangle$.

Output: the optimal cost to multiply $A_i \dots A_j$.

1 **if** $i = j$ **return** 0

2 $m[i, j] = \infty$

3 **for** $k = i$ **to** $j - 1$

4 $q = \text{cost}(p, i, k) + \text{cost}(p, k + 1, j) + p_{i-1}p_kp_j$

5 **if** $q < m[i, j]$

6 $m[i, j] = q$

7 **return** $m[i, j]$

Memoization: Top-down Algorithm

```
1 for  $i = 1$  to  $n$ 
2   for  $j = i$  to  $n$ 
3      $m[i, j] = \infty$ 
4 return lookup( $p, 1, n$ )
```

lookup(p, i, j)

```
1 if  $m[i, j] < \infty$  return  $m[i, j]$ 
2 if  $i = j$ 
    $m[i, j] = 0$ 
3 else for  $k = i$  to  $j - 1$ 
4    $q = \mathbf{lookup}(p, i, k) + \mathbf{lookup}(p, k + 1, j) + p_{i-1}p_kp_j$ 
5   if  $q < m[i, j]$ 
      $m[i, j] = q$ 
6 return  $m[i, j]$ 
```

Bottom-up Algorithm

Matrix-cost(p)

Input: A sequence of matrix dimensions: $p = \langle p_0, p_1, \dots, p_n \rangle$.

Output: the optimal cost to multiply $A_1 \dots A_n$.

```
1 for  $i = 1$  to  $n$ 
     $m[i, i] = 0$ 
2 for  $l = 2$  to  $n$ 
3     for  $i = 1$  to  $n - l + 1$ 
4          $j = i + l - 1$ 
5          $m[i, j] = \infty$ 
6         for  $k = i$  to  $j - 1$ 
7              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
8             if  $q < m[i, j]$ 
9                  $m[i, j] = q$ 
10 return  $m[1, n]$ 
```

A Problem from Computational Biology

Given two (DNA or protein) sequences $s[1 \dots m]$ and $t[1 \dots n]$ we want the “best” **alignment** between the two sequences.

Example:

$$s = GACGGATTAG$$
$$t = GATCGGAATAG$$

An alignment is an insertion of spaces in arbitrary locations along the sequences so that they end up with the same size.

A space in one sequence should not align with a space in the other. But spaces can be inserted at the beginning or at the end.

An alignment can be **scored** by a scoring scheme. We assume a scoring matrix *score* where the entry

$score[x, y]$ gives the alignment score for characters x and y .

The score for an alignment is the sum of the scores of its aligned characters.

A best alignment is one which receives the maximum score called the **similarity** — $sim(s, t)$.

Recursive formula

$$\text{sim}(s[1 \dots i], t[1 \dots j])$$

$$= \max \begin{cases} \text{sim}(s[1 \dots i - 1], t[1 \dots j - 1]) + \text{score}(s[i], t[j]) \\ \text{sim}(s[1 \dots i], t[1 \dots j - 1]) + \text{score}(-, t[j]) \\ \text{sim}(s[1 \dots i - 1], t[1 \dots j]) + \text{score}(s[i], -) \end{cases}$$

Bottom-up Algorithm

Input: sequences $s[1 \dots m]$ and $t[1 \dots n]$

Output: $sim(s, t)$

for $i = 0$ to m

$a[i, 0] = i \times score(-, t[1])$

for $j = 0$ to n

$a[0, j] = j \times score(s[1], -)$

for $i = 1$ to m

for $j = 1$ to n

$a[i, j] = \max\{a[i - 1, j - 1] + score(s[i], t[j]),$
 $a[i, j - 1] + score(-, t[j]),$
 $a[i - 1, j] + score(s[i], -)\}$

return $a[m, n]$

0/1 Knapsack

Let $KNAP(l, j, y)$ represent the problem:

$$\text{maximize } \sum_{l \leq i \leq j} p_i x_i$$

$$\text{subject to } \sum_{l \leq i \leq j} w_i x_i \leq y$$

$$x_i \in \{0, 1\}, l \leq i \leq j$$

The Knapsack problem is $KNAP(1, n, m)$. For simplicity, we will assume that all the weights, profits, and m are integers.

Let $f_j(y)$ be the optimal solution to $KNAP(1, j, y)$. Then,

$$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

because of the principle of optimality.

And for arbitrary $f_i(y)$, $i > 0$,

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

Using, $f_0(y) = 0$ for all $y \geq 0$ and

$f_i(y) = -\infty$ if $y < 0$

we compute $f_n(m)$ in a bottom up manner.

The time complexity is $O(nm)$.

Elements of Dynamic Programming

Optimal subproblems: A k -stage optimal solution is computed from $k - 1$ -stage optimal solutions.

Overlapping subproblems: the same $k - 1$ -stage subproblem is used in the computation of a number of k -stage subproblems.