

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.

Amortized Analysis

Useful when analyzing algorithms that perform a sequence of similar operations.

The time required to perform a sequence of operations is averaged over all the operations performed.

No probabilistic assumptions - not an average case analysis!

Techniques: Aggregate Method, Accounting Method, Potential Function Method.

Stack Operations

Push(S, x) - Push x to the stack S .

Pop(S, x) - Pop the top of stack S .

Multipop(S, k)

1. While not empty and $k \neq 0$ do

1.1 Pop(S);

1.2 $k \leftarrow k - 1$;

A call to Multipop(S, k) takes $O(k)$ time.

A “naive” analysis of a sequence of n operations Push, Pop, and Multipop on an empty stack give $O(n^2)$, since the worst case time of one stack operation is $O(n)$.

Binary Counter

k bit binary counter $A[0, \dots, k - 1]$.

$$k = \text{length}[A]$$

$$x = \sum_{i=0}^{k-1} A[i]2^i$$

Increment(A):

1. $i \leftarrow 0$;
2. While $i < \text{length}[A]$ and $A[i] = 1$ do
 - 2.1 $A[i] \leftarrow 0$
 - 2.2 $i \leftarrow i + 1$
3. If $i < \text{length}[A]$ then $A[i] \leftarrow 1$.

What's the cost of n increments?

A worst case increment takes $O(k)$ steps, thus $O(nk)$ time.

Amortized Analysis - The Aggregate Method

Compute $T(n)$, the total work in n operations and take $T(n)/n$.

Stack Operations: The total amount of work is $O(\text{number of Push operations})$, thus $T(n) = O(n)$, and amortized work is $O(1)$.

Counter increments: In n increments the i -th bit ($i = 0, 1, \dots, k - 1$) is flipped only every 2^i increments. Thus

$$T(n) = \sum_{i=0}^{\log n} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Thus, the amortized work is $T(n)/n = O(1)$.

Amortized Analysis - The Accounting Method

We associate a **charge** with each operation which will represent its **amortized cost**.

If operation's amortized cost exceeds its actual cost, the difference is stored as **credit** associated with specific items in the data structure.

Credit can be used later to pay for operations whose amortized cost is less than the actual cost.

We maintain the invariant that at any step the total charges in the new accounting is not smaller than the total real cost up to that point. Thus total credit stored should never become negative.

Stack Operations:

Real cost: Push - 1; Pop - 1; Multipop(k, S) - $\min(k, s)$.

New charge: Push - 2; Pop - 0; Multipop(k, S) - 0.

Since the number of Pops is bounded by the number of Push operations, the total new charge at any given time is never smaller than the real cost.

A simple $O(n)$ bound on the total amortized cost.

Counter increments:

New charge: 2 for each flip to 1, 0 otherwise.

Since we start with all bits set to 0, this give an upper bound on the real cost.

In each increment operation only one bit is flipped to 1 - $O(1)$ amortized cost.

Amortized Analysis - The Potential Method

Let D_0 be the initial data structure.

Consider a set of n operations: c_i the cost of operation i , D_i the data structure after operation i .

The **potential function**:

$$\Phi : \{D_i \mid i = 0, \dots, n\} \rightarrow R$$

The amortized cost of operation i is

$$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The total amortized cost:

$$\begin{aligned}\sum_{i=1}^n A(c_i) &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

If we can show that $\Phi(D_i) \geq \Phi(D_0)$ for all i , then

$$\sum_{i=1}^n A(c_i) \geq \sum_{i=1}^n c_i$$

Stack Operations

Define $\Phi(D_i)$ = number of items in the stack.

$$\Phi(D_i) \geq \Phi(D_0) = 0.$$

The amortized cost of a push operation (starting with s items in the stack) is:

$$A(c_i) = 1 + (s + 1) - s = 2$$

The amortized cost of a `Multipop(S, k)` ($k' = \min(k, s)$)

$$A(c_i) = k' + (s - k') - s = 0$$

Thus, all operations have amortized cost $O(1)$, the total cost of n operations is $O(n)$.

Counter Increment

Let $\Phi(D_i)$ = number of 1's in the counter.

If counter starts with 0: $\Phi(D_i) \geq \Phi(D_0) = 0$.

Assume that the i -th operation (increment) flipped t_i bits to 0.

The total cost of that operation is $t_i + 1$.

$$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 + 1 - t_i = 2.$$

The total amortized cost of n operations is $O(n)$.

If the counter didn't start with 0:

$$\begin{aligned}\sum_{i=1}^n A(c_i) &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

or

$$\begin{aligned}\sum_{i=1}^n c_i &= \sum_{i=1}^n A(c_i) + \Phi(D_0) - \Phi(D_n) \\ &\leq \sum_{i=1}^n A(c_i) + k\end{aligned}$$

Connected components

A **connected component** of an (undirected) graph is a maximal set V_i such that for vertices $u \in V_i$ and $v \in V_i$, there exists a path from u to v .

Connected-Components(G);

for each vertex $v \in V$ **do**

 Make-Set(v);

for each edge $(u, v) \in E$ **do**

 If Find-Set(u) \neq Find-Set(v) **then** Union(u, v);

- MAKE-SET(v) creates a singleton set containing v .
- UNION(x, y) takes the union of the set containing x and the set containing y .
- FIND-SET(x) returns a pointer to the *representative* of the set containing x .