

CMPUT 403: Dynamic Programming



Zachary Friggstad

January 23, 2016

Dynamic Programming by Example

Example: Matrix Chain Multiplication

Given matrices M_1, M_2, \dots, M_n where matrix M_i is an $r_i \times c_i$ matrix and $c_i = r_{i+1}$.

Goal: Compute $M_1 \cdot M_2 \cdot \dots \cdot M_n$, an $r_1 \times c_n$ matrix.

Associativity tells us there are many ways to do this:

$$A \cdot (B \cdot C) \quad \text{vs} \quad (A \cdot B) \cdot C$$

or

$$A \cdot (B \cdot (C \cdot D)) \quad \text{vs} \quad (A \cdot B) \cdot (C \cdot D) \quad \text{vs} \quad (A \cdot (B \cdot C)) \cdot D$$

What is the **cheapest** way.

Dynamic Programming by Example

Say the cost of multiplying an $a \times b$ matrix S with a $b \times c$ matrix T is $abc = \#$ of element-by-element multiplications when computing $S \times T$ naively.

Example:

- $A : 1 \times 100$
- $B : 100 \times 100$
- $C : 100 \times 100$
- $D : 100 \times 1$

Cost of $(A \cdot (B \cdot C)) \cdot D$ is $100^3 + 100^2 + 100 = 1010100$.

Cost of $(A \cdot B) \cdot (C \cdot D)$ is $2 \cdot 100^2 + 100 = 20100$.

Dynamic Programming by Example

Recall we want to compute $A_1 \cdot \dots \cdot A_k$ where A_i is an $r_i \times c_i$ matrix.

What order of multiplication/“parenthesizing” results in the cheapest calculation?

Idea:

Let $\text{cost}(i, j)$ be the cheapest way to compute $A_i \cdot \dots \cdot A_j$.

If $i = j$, nothing to compute (answer is 0).

Otherwise “guess” the outermost multiplication:

$$(A_i \cdot \dots \cdot A_k) \cdot (A_{k+1} \cdot \dots \cdot A_j)$$
$$\text{cost}(i, j) = \min_{i \leq k \leq j-1} \text{cost}(i, k) + \text{cost}(k+1, j) + r_i \cdot c_k \cdot c_j.$$

The Recurrence

Concisely,

$$\text{cost}(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j-1} \text{cost}(i, k) + \text{cost}(k+1, j) + r_i \cdot c_k \cdot c_j & \text{if } i < j \end{cases}$$

```
int cost(int i, int j) {
    if (i == j) return 0; //base case
    int best = INT_MAX;
    for (int k = i; k < j; k++)
        best = min(best,
                    cost(i, k) + cost(k+1, j) + r[i]*c[k]*c[j]);
    return best;
}
```

Memoization

This takes exponential time.

Fantastic Idea: Computed each $\text{cost}(i,j)$ entry only once!

```
//table[][] is initialized to contain all -1 entries
int cost(int i, int j) {
    if (table[i][j] == -1) { //first time computing cost(i,j)
        if (i == j) table[i][j] = 0;
        else {
            table[i][j] = INT_MAX;
            for (int k = i; k < j; k++)
                table[i][j] = min(table[i][j],
                    cost(i,k)+cost(k+1,j) + r[i]*c[k]*c[j]);
        }
    }
    return table[i][j];
}
```

Memoization

$$\text{cost}(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j-1} \text{cost}(i, k) + \text{cost}(k + 1, j) + r_i \cdot c_k \cdot c_j & \text{if } i < j \end{cases}$$

There are $O(n^2)$ different subproblems (i.e. (i, j) pairs).

Computing each requires $O(n)$ time within the current recursive call.

Running time: $O(n^3)$.

General Recipe

Formulate a mathematical *recurrence* that solves the problem.

Ingredients:

- “Base cases” that are trivial to solve.
- A clear way to break larger problems into “smaller” subproblems plus some easy to compute part.

Sort of a “computation by induction”. We can easily compute the solution if we know the solutions to smaller subproblems.

Running Time Analysis Template:

(# of possible subproblems) \times (time in one recursive call).

Bottom-Up

The approach just shown is **Top-Down**: check a table before completing the calculation to see if it already done.

Bottom-Up: solve subproblems from “smallest” to “largest”.

```
for (int len = 1; len <= n; len++)
  for (int i = 0, j = len - 1; j < n; i++, j++) {
    if (i == j) cost[i][j] = 0;
    else {
      cost[i][j] = INT_MAX;
      for (int k = i; k < j; k++)
        cost[i][j] = min(cost[i][j],
                        cost[i][k] + cost[k+1][j] + r[i] * c[k] * c[j]);
    }
  }
```

Comparison


Top-Down Advantage:

- Never have to think about how to cleverly build the table from the bottom up. Just memoize the natural recurrence.

Bottom-Up Advantage:

- Runs a bit faster. Not asymptotically faster, though.
- No concern about filling the call stack by recursing too deep.
- Can sometimes use a smaller table (thanks for the reminder Jason!)

In the vast majority of contest problems, the top-down approach works fine.



Recovering a Solution

This computes the cost of the cheapest way to multiply the chain of matrices. But it doesn't give the exact sequence!

To recover the sequence, just trace through the DP table.

The outermost multiplication in the cheapest way to compute $A_i \cdot \dots \cdot A_j$ is at the index k achieving the min-value in the recurrence.

For this k , we compute $(A_i \cdot \dots \cdot A_k) \cdot (A_{k+1} \cdot \dots \cdot A_j)$.

More Examples

Longest path in a DAG

Let G be a DAG, a **D**irected **A**cylic **G**raph with vertices V and edges E . Let s be a vertex.

Find the *longest* path starting at s .

Let $f(v)$ be the length of the longest path starting from v and $N(v)$ all nodes u such that vu is an edge.

$$f(v) = \begin{cases} 0 & \text{if } N(v) = \emptyset \\ 1 + \max_{u \in N(v)} f(u) & \text{otherwise} \end{cases}$$

Running Time: $O(|V|^2)$: $|V|$ vertices, each with $\leq |V|$ neighbours.

Maybe Better Analysis: $O(|V| + |E|)$ if $N(v)$ is stored as an array.

Longest Common Subsequence

Let $s = s_0s_1 \dots s_{n-1}$ and $t = t_0t_1 \dots t_{m-1}$ be two sequences.

Find a *longest common subsequence* (LCS) between s and t . For example, a LCS of $s = \text{human}$ and $t = \text{chimp}$ is hm .

Let $f(i, j)$ be the length of the LCS of $s_0 \dots s_{i-1}$ and $t_0 \dots t_{j-1}$.

$$f(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + f(i - 1, j - 1) & \text{if } s_i = t_j \\ \max(f(i - 1, j), f(i, j - 1)) & \text{otherwise} \end{cases}$$

Running Time: $O(nm)$

Subset Sum

Given integers $a_1, \dots, a_n \geq 0$ and some integer $\alpha \geq 0$, is there some $S \subseteq \{1, \dots, n\}$ with $\sum_{i \in S} a_i = \alpha$?

Let $f(k, b)$ be a boolean value that is true if and only if there is some $A \subseteq \{1, \dots, k\}$ with $\sum_{i=1}^k a_i = b$.

$$f(k, b) = \begin{cases} \text{TRUE} & \text{if } k = 0, b = 0 \\ \text{FALSE} & \text{if } k = 0, b \geq 1 \\ f(k-1, b) & \text{if } k \geq 1, b \leq a_k - 1 \\ f(k-1, b) \text{ OR } f(k-1, b - a_k) & \text{if } k \geq 1, b \geq a_k \end{cases}$$

Running Time: $O(n \cdot \alpha)$