6.5 Chain matrix multiplication

Multiply two matrices at a time

Not commutative but it is associative (Ax(BxC)=(AxB)xC)

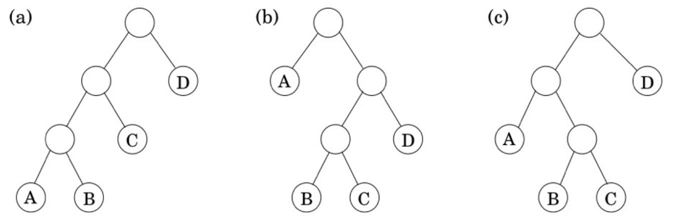
Multiplying an mxn matrix by an nxp matrix takes mnp multiplications

Order of multiplications makes a big difference in the final running time

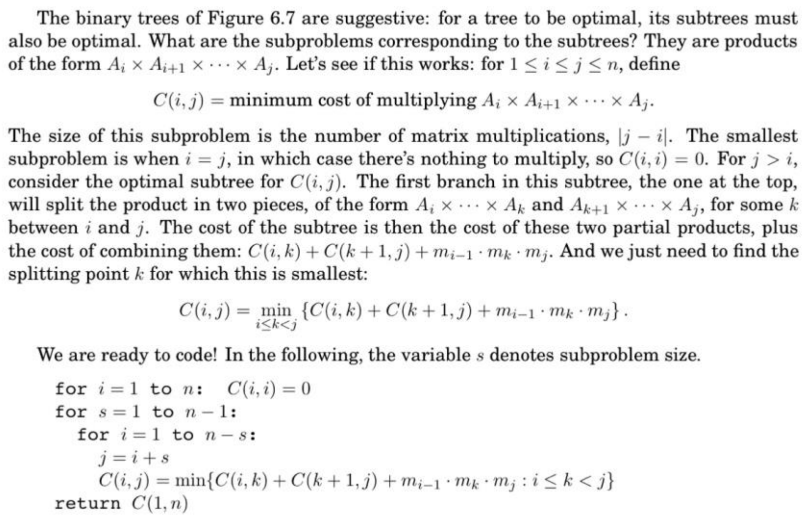
How to determine the optimal order?

Various full binary tress with n leaves whose number is exponential in n

Figure 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



Can't try each each tree, so turn to dynamic programming



The binary trees of Figure 6.7 are suggestive: for a tree to be optimal, its subtrees must also be optimal. What are the subproblems corresponding to the subtrees? They are products of the form $A_i \times A_{i+1} \times \cdots \times A_j$. Let's see if this works: for $1 \le i \le j \le n$, define

$$C(i, j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j.$$

The size of this subproblem is the number of matrix multiplications, $|j - i|$. The smallest subproblem is when $i = j$, in which case there's nothing to multiply, so $C(i, i) = 0$. For $j > i$, consider the optimal subtree for $C(i, j)$. The first branch in this subtree, the one at the top, will split the product in two pieces, of the form $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$, for some k between i and j. The cost of the subtree is then the cost of these two partial products, plus the cost of combining them: $C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j$. And we just need to find the splitting point k for which this is smallest:

$$C(i, j) = \min_{i \le k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

We are ready to code! In the following, the variable s denotes subproblem size.

```
for $i = 1$ to $n$:   $C(i, i) = 0$
for $s = 1$ to $n - 1$:
  for $i = 1$ to $n - s$:
    $j = i + s$
    $C(i, j) = \min\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j : i \le k < j\}$
return $C(1, n)$
```

6.6 Shortest paths

Shortest reliable paths

In dynamic programming, choose subproblems so that all vital information is remembered and carrier forward.

Let us define, for each vertex v and each integer i<=k, dist(v, i) to be the length of the shortest path from s to v that uses i length

The starting values dist(v, 0) are infinite for all vertices except s, for which it is 0

General update equation is, naturally enough

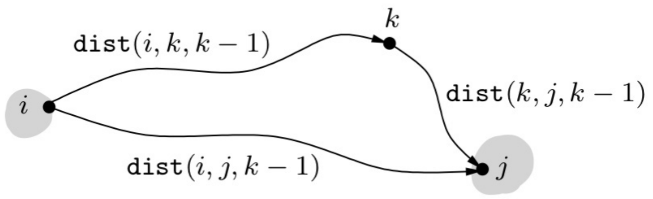
dist(v, i) = min {dist(u, i-1) + l(u, v)}

All-pairs shortest paths

Shortest path not just between s and t but between all pairs of vertices

If execute our general shortest-path algorithm |V| times, total running time would be O(|V|^2|E|)

Better alternative, the O(|V|^3) dynamic programming-based *Floyd-Warshall algorithm*



Using k gives us a shorter path from I to j if and only if

Dist(i, k, k-1) + dist(k, j, k-1) < dist(i, j, k-1)

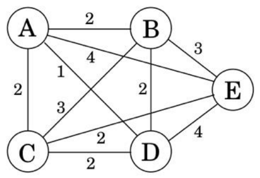
in which case dist(I, j, k) should be updated accordingly.

Here is the Floyd-Warshall algorithm—and as you can see, it takes $O(|V|^3)$ time.

```
for $i = 1$ to $n$:
   for $j = 1$ to $n$:
      dist$(i, j, 0) = \infty$
for all $(i, j) \in E$:
   dist$(i, j, 0) = \ell(i, j)$
for $k = 1$ to $n$:
   for $i = 1$ to $n$:
      for $j = 1$ to $n$:
         dist$(i, j, k) = \min\{$dist$(i, k, k-1) +$ dist$(k, j, k-1),$ dist$(i, j, k-1)\}$
```

The traveling salesman problem

Figure 6.9 The optimal traveling salesman tour has length 10.



Best order to visit cities to minimize the overall distance traveled?

Cities: 1, .., n (hometown as 1)

Matrix of intercity distances: D(dij)

Goal: A tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length

Notorious computational tasks.. unlikely to be solvable in polynomial time

Evaluate every possible tour and return the best one

Since there are (n-1)! Possibilities, this strategy takes O(n!) time

Dynamic programming way faster than polynomial

What is the appropriate subproblem for the TSP?

For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S|>1, we define C(S, 1) = infinite since the path cannot both start and end at 1

C(S, j) in terms of similar problems

$$C(S,j) = \min_{i \in S: i \neq j} C(S-\{j\}, i) + d_{ij}.$$

The subproblems are ordered by $|S|$. Here's the code.

```
$C(\{1\},1) = 0$
for $s = 2$ to $n$:
   for all subsets $S \subseteq \{1,2,\ldots,n\}$ of size $s$ and containing 1:
      $C(S,1) = \infty$
      for all $j \in S, j \neq 1$:
         $C(S,j) = \min\{C(S-\{j\},i) + d_{ij} : i \in S, i \neq j\}$
return $\min_j C(\{1,\ldots,n\},j) + d_{j1}$
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

6.7 Independent sets in trees

A subset of nodes S ⊂ V is an independent set of graph G = (V, E) if there are no nodes between them.

Finding the largest independent set in a graph is believed to be intractable, ***but when the graph happens to be a tree, the problem can be solved in linear time, using dynamic programming***

Chain matrix multiplication problem) Layered structure of a tree provides a natural definition of a subproblem-as long as one node of the tree has been identified as a root

Rooting the tree at any node r

Each node defines a subtree-the one hanging from it

i(u) = size of largest independent set of subtree hanging from u

Final goal is I(r)

Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree. Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know $I(w)$ for all descendants w of u. How can we compute $I(u)$? Let's split the computation into two cases: any independent set either includes u or it doesn't (Figure 6.11).

$$I(u) \;=\; \max\left\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w),\; \sum_{\text{children } w \text{ of } u} I(w)\right\}.$$

If the independent set includes u, then we get one point for it, but we aren't allowed to include the children of u—therefore we move on to the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

The number of subproblems is exactly the number of vertices. With a little care, the running time can be made linear, $O(|V| + |E|)$.

Figure 6.11 $I(u)$ is the size of the largest independent set of the subtree rooted at u. Two cases: either u is in this independent set, or it isn't.