**CS161 Lecture 13** Chain Matrix Multiplication, Knapsack, Independent Set
**Scribe:** Wilbur Yang (2016) **Date:** November 9, 2016

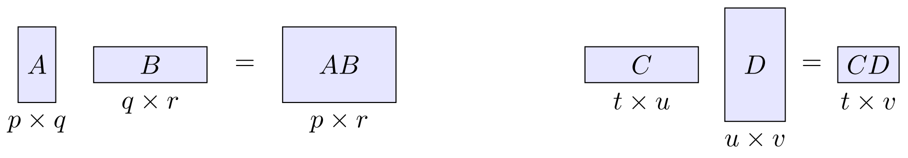
## 1 Overview

Last lecture, we talked about dynamic programming (DP), a useful paradigm and one technique that you should immediately consider when you are designing an algorithm. We covered the Bellman-Ford algorithm for solving the single source shortest path problem, and we talked about the Floyd-Warshall algorithm for solving the all pairs shortest path problem. Also, we explored the longest common subsequence problem, which has important applications in biocomputation.

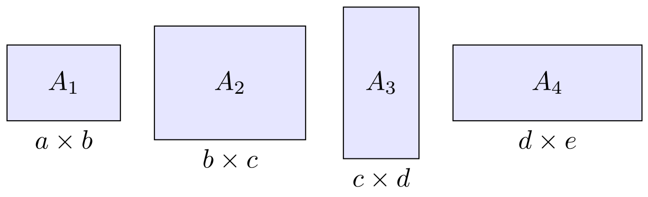
This lecture, we will cover some more examples of dynamic programming. We will talk about three problems today: chain matrix multiplication, knapsack, and maximum weight independent set in trees.

## 2 The Chain Matrix Multiplication Problem

Recall that if you have a matrix $A$ with dimensions $p \times q$ and a matrix $B$ with dimensions $q \times r$, then $AB$ is a $p \times r$ matrix, and calculating $AB$ (naively) takes $pqr$ multiplications. Also recall that in general, matrix multiplication is not commutative; that is, $AB \neq BA$. In fact, it doesn't even make sense to multiply matrices if their dimensions are not compatible. However, matrix multiplication is associative, which means that $(AB)C = A(BC)$.



The problem is as follows: given matrices $A_1, A_2, \ldots, A_n$, compute $A_1 \times A_2 \times \ldots \times A_n$ using the fewest possible scalar multiplications.



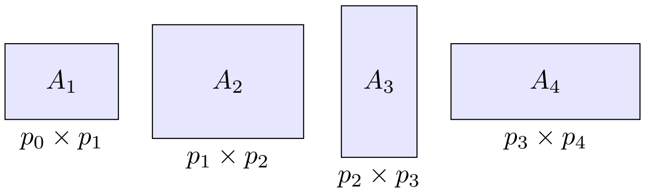
How would you compute this matrix product? If $\{A_i\}$ were all square $k \times k$ matrices, it wouldn't matter which order you multiply them in: there would be $n - 1$ matrix multiplications, each requiring $k^3$ scalar multiplications. However, the problem becomes interesting when these matrices are not all square matrices. We want to save operations by multiplying matrices in an efficient order.

Let's first see how exactly we can multiply a matrix chain in a variety of different ways. Suppose we have four matrices $A_1, A_2, A_3, A_4$. There are five ways to multiply them, yielding different operation counts:

1. $(A_1(A_2(A_3A_4)))$ takes $cde + bce + abe$ multiplications.
2. $(A_1((A_2A_3)A_4))$ takes $bcd + bde + abe$ multiplications.
3. $((A_1A_2)(A_3A_4))$ takes $abc + cde + ace$ multiplications.
4. $((A_1(A_2A_3))A_4)$ takes $bcd + abd + ade$ multiplications.
5. $(((A_1A_2)A_3)A_4)$ takes $abc + acd + ade$ multiplications.

**Remark 1.** *The number of ways to multiply a matrix chain is very closely tied to the number of ways to write valid nested parentheses. This sequence is called the Catalan numbers.*

Let's say that the $i$th matrix has dimensions $p_{i-1} \times p_i$. How do you compute this matrix product, minimizing the number of multiplications? The number of ways to compute the product is $\Omega(2^n)$, so we can't simply list them out and try them all.



One idea is to get rid of large dimensions first by multiplying wide matrices with tall matrices first. Unfortunately, that intuition isn't very helpful here as it's not clear how exactly to do that. As with all dynamic programming approaches, the key is to consider the subproblems that we need to solve in order to solve the overall problem.

Here's an idea: at the very last step, we've multiplied $(A_1, \ldots, A_k)$ and $(A_{k+1}, \ldots, A_n)$. So, we need to solve the subproblems of finding the minimum cost to compute $A_i \times \ldots \times A_j$. Let $A_{i\ldots j} = A_i \times \ldots \times A_j$. We must split the chain at some position $k$, where $i \leq k < j$. That is, we need to compute $A_{i\ldots k}$ and $A_{k+1\ldots j}$ and multiply them. The optimal solution for $A_{i\ldots j}$ must use the optimal solutions for $A_{i\ldots k}$ and $A_{k+1\ldots j}$. Then, we have the recurrence $\textsc{Cost}(A_{i\ldots j}) = \min_k \left(\textsc{Cost}(A_{i\ldots k}) + \textsc{Cost}(A_{k+1\ldots j}) + p_{i-1}p_kp_j\right)$.

If we were to simply use this recursion without memoization to solve the problem, then our algorithm would take exponential time to finish. However, because the same subproblems show up multiple times, we can either memoize our subproblem solutions in a top-down way or build up solutions from smaller subproblems to larger subproblems in a bottom-up fashion. Now let's turn this recurrence into a bottom-up dynamic programming algorithm. Let $m[i, j]$ be the minimum cost for computing $A_{i\ldots j}$. If $k$ is the optimal splitting point, then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

Letting $m[i, j]$ represent products of the $(j - i + 1)$ matrices from $i$ to $j$, the expression for $m[i, j]$ needs to know the costs for products of strictly less than $(j - i + 1)$ matrices. Specifically, $m[i, j]$ is determined by $m[i, k]$ and $m[k + 1, j]$, $i \leq k < j$. So,

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

We compute entries of $m[i, j]$ in increasing order of $j - i$, starting with our base cases that $m[i, i] = 0$ for all $i$. At the end, $m[1, n]$ will be the answer for the overall problem.

**Algorithm 1:** $\textsc{MinCostMatrixChain}(n, p)$

```
for $i = 1, \ldots, n$ do
    $m[i,i] \leftarrow 0$
for $\ell = 1, \ldots, n-1$ do
    for $i = 1, \ldots, n - \ell$ do
        $j \leftarrow i + \ell$
        $m[i,j] \leftarrow \infty$
        for $k = i, \ldots, j-1$ do
            $m[i,j] \leftarrow \min\{m[i,j], m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$
return $m[1,n]$
```

Also, if we do some extra bookkeeping, we can recover the multiplication order at the end. While filling out the $(i, j)$th entry of $m$, we can keep track of the optimal splitting point in another table $s$ at $s[i, j]$. At the end, we can traverse $s$ backwards using these "pointers" at each $s[i, j]$ to determine how exactly to multiply our matrices.

The running time is $O(n^3)$ since we take $O(n)$ time to compute each entry in the DP table $m$.

## 3 The Knapsack Problem

This is a classic problem, defined as the following:

We have $n$ items, each with a value and a positive weight. The $i$th item has weight $w_i$ and value $v_i$. We have a knapsack that holds a maximum weight of $W$. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

| Item | Weight | Value |
|---|---|---|
| $A$ | 6 | 25 |
| $B$ | 3 | 13 |
| $C$ | 4 | 15 |
| $D$ | 2 | 8 |

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

### 3.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item $B$ and two of item $D$. Then, the total weight is 10, and the total value 42.

We define $K(x)$ to be the optimal solution for a knapsack of capacity $x$. Suppose $K(x)$ happens to contain the $i$th item. Then, the remaining items in the knapsack must have a total weight of at most $x - w_i$. The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

$$K(x) = \max_{i : w_i \leq x} \left( K(x - w_i) + v_i \right).$$

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize $K(0) = 0$, and then we compute $K(x)$ values from $x = 1, \ldots, W$. The overall runtime is $O(nW)$.

**Algorithm 2:** UnboundedKnapsack$(W, n, w, v)$

```
$K[0] \leftarrow 0$
for $x = 1, \ldots, W$ do
    $K[x] \leftarrow 0$
    for $i = 1, \ldots, n$ do
        if $w_i \leq x$ then
            $K[x] \leftarrow \max\{K[x], K[x - w_i] + v_i\}$
return $K[W]$
```

**Remark 2.** *This solution is not actually polynomial in the input size because it takes* $\log(W)$ *bits to represent* $W$*. We call these algorithms "pseudo-polynomial." If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.*

### 3.2 The 0-1 Knapsack Problem

Now we consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item $A$ and item $C$, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension to the DP table). Let $K(x,j)$ be the maximum value that we can get with a knapsack of capacity $x$ considering only items at indices from $1,\ldots,j$. Consider the optimal solution for $K(x,j)$. There are two cases:

1. Item $j$ is used in $K(x,j)$. Then, the remaining items that we choose to put in the knapsack must be the optimum solution for $K(x-w_j, j-1)$. In this case, $K(x,j) = K(x-w_j, j-1) + v_j$.
2. Item $j$ is not used in $K(x,j)$. Then, $K(x,j)$ is the optimum solution for $K(x, j-1)$. In this case, $K(x,j) = K(x, j-1)$.

So, our recurrence relation is: $K(x,j) = \max\{K(x-w_j, j-1) + v_j, K(x, j-1)\}$. Now, we're done: we simply calculate each entry up to $K(W,n)$, which gives us our final answer. Note that this also runs in $O(nW)$ time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do $O(1)$ work.

**Algorithm 3:** ZeroOneKnapsack($W, n, w, v$)

```
for $x = 1, \ldots, W$ do
    $K[x, 0] \leftarrow 0$
for $j = 1, \ldots, n$ do
    $K[0, j] \leftarrow 0$
for $j = 1, \ldots, n$ do
    for $x = 1, \ldots, W$ do
        $K[x, j] \leftarrow K[x, j-1]$
        if $w_j \leq x$ then
            $K[x, j] \leftarrow \max\{K[x, j], K[x - w_j, j-1] + v_j\}$
return $K[W, n]$
```

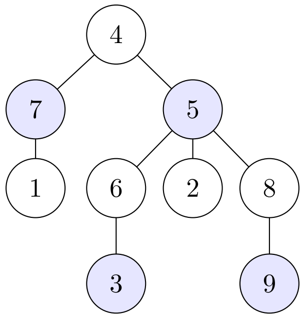
## 4 The Independent Set Problem

This problem is as follows:

Say that we have an undirected graph $G = (V, E)$. We call a subset $S \subseteq V$ of vertices "independent" if there are no edges between vertices in $S$. Let vertex $i$ have weight $w_i$, and denote $w(S)$ as the sum of weights of vertices in S. Given $G$, find an independent set of maximum weight $\text{argmax}_{S \subseteq V} w(S)$.

Actually, this problem is NP-hard for a general graph $G$. However, if our graph is a tree, then we can solve this problem in linear time. In the following figure, the maximum weight independent set is highlighted in blue.

**Remark 3.** *Dynamic programming is especially useful to keep in mind when you are solving a problem that involves trees. The tree structure often lends itself to dynamic programming solutions.*



As usual, the key question to ask is, "What should our subproblem(s) be?" Intuitively, if the problem has to do with trees, then subtrees often play an important role in identifying our subproblems. Let's pick any vertex $r$ and designate it as the root. Denoting the subtree rooted at $u$ as $T_u$, we define $A(u)$ to be the weight of the maximum weight independent set in $T_u$. How can we express $A(u)$ recursively? Letting $S_u$ be the maximum weight independent set of $T_u$, there are two cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children $v$ of $u$.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v A(v)$ for all grandchildren $v$ of $u$.

To avoid solving the subproblem for trees rooted at grandchildren, we introduce $B(u)$ as the weight of the maximum weight independent set in $T_u \setminus \{u\}$. That is, $B(u) = \sum_v A(v)$ for all children $v$ of $u$. Equivalently, we have the following cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children $v$ of $u$.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v B(v)$ for all children $v$ of $u$.

So, we can calculate the weight of the maximum weight independent set:

$$A(u) = \max \left\{ \sum_{v \in \textsc{Children}(u)} A(v), w_u + \sum_{v \in \textsc{Children}(u)} B(v) \right\}$$

To create an algorithm out of this recurrence, we can compute the $A(u)$ and $B(u)$ values in a bottom-up manner (a post-order traversal on the tree), arriving at the answer, $A(r)$. This takes $O(|V|)$ time.

```
Algorithm 4: MaxWeightIndependentSet(G)        ▷ G is a tree
r ← ArbitraryVertex(G)
T ← RootTreeAt(G, r)
Procedure SolveSubtreeAt(u)
    if Children(T, u) = ∅ then
        A(u) ← $w_u$
        B(u) ← 0
    else
        for v ∈ Children(T, u) do
            SolveSubtreeAt(v)
        A(u) ← $\max\left\{\sum_{v \in \textsc{Children}(T,u)} A(v), w_u + \sum_{v \in \textsc{Children}(T,u)} B(v)\right\}$
        B(u) ← $\sum_{v \in \textsc{Children}(T,u)} A(v)$
SolveSubtreeAt(r)
return A(r)
```