What is Dynamic Programming?

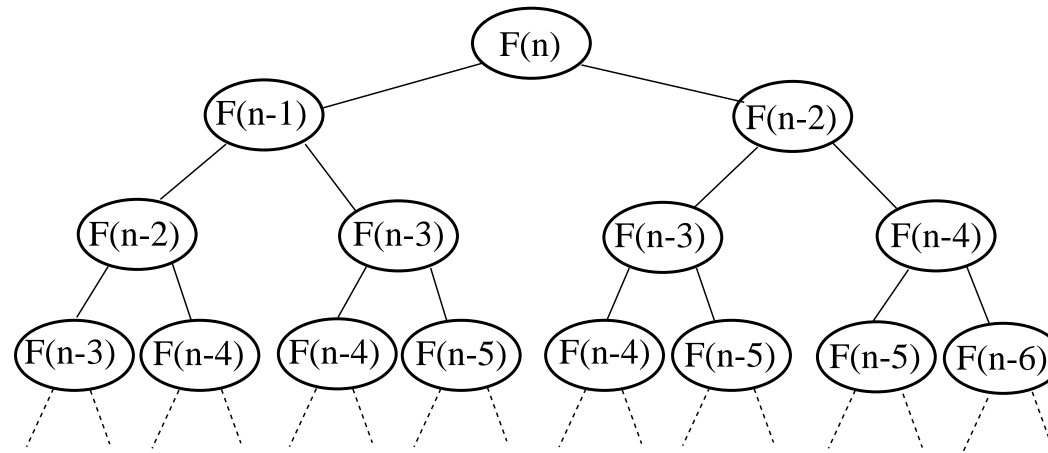
- Dynamic programming is a method for solving optimization problems by combining solutions of subproblems.
- Unlike divide-and-conquer methods, dynamic programming is best suited when a problem has many subproblems which overlap.
- Dynamic programming is usually used in optimization problems.
- The basic steps are:
  - Define the optimal solution.
  - Give a recursive definition for computing the optimal solution based on optimal solutions of smaller problems.
  - Compute the optimal solutions and/or the value of the optimal solution in a bottom-up fashion.
- We'll give more details later.

A Trivial example

- We have seen the **Fibonacci sequence** before:

$$F(n) = F(n-1) + F(n-2)$$

- We have seen that it can be computed recursively:



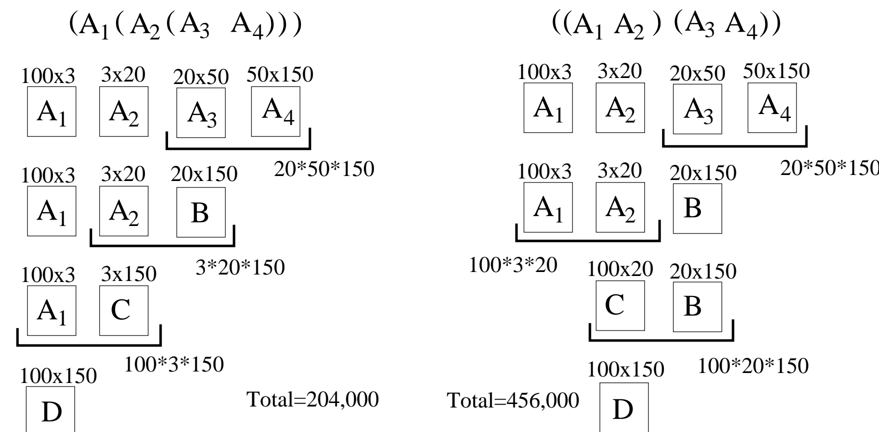
- We have also seen that this takes exponential time.
- The problem is that we solve the same subproblems repeatedly.
- With dynamic programming, we solve each subproblem only once.

A Dynamic Programming Example

- **The Matrix-Chain Multiplication Problem (MCM Problem)**
  - **Given:** A sequence of matrices, $(A_1, A_2, \ldots, A_n)$, where matrix A_i has size $p_{i-1} \times p_i$.
  - **Solution:** A full parenthesization of the product $A_1 A_2 \cdots A_n$.
  - **Cost of a Solution:** The number of operations required to compute $A_1 A_2 \cdots A_n$ given the parenthesization.
  - **Optimal solution:** A Solution with minimal cost.
- Recall that to compute the product of an $a \times b$ matrix with a $b \times c$ requires abc operations.
- We can express the sizes of the matrices as a sequence of $n + 1$ numbers, since the number of columns of A_i is the number of rows of A_{i+1} for each i.

MCM Example

- Given 4 matrices A_1, A_2, A_3, and A_4 with sizes 100×3, 3×20, 20×50, and 50×150.
- This gives us the sequence of p values $\{100, 3, 20, 50, 150\}$
- We want to cheapest way to compute $A_1 A_2 A_3 A_4$.
- Computing $(A_1(A_2(A_3 A_4)))$ requires 204,000 operations.
- Computing $((A_1 A_2)(A_3 A_4))$ requires 456,000 operations.
- There are 3 other ways to compute the product.



Solving the MCM Problem

- One way to find the optimal solution would be to try all possible solutions and find the one with the least cost.
- The number of parenthesization of a sequence of n matrices can be shown to be

$$P(n) = \frac{1}{n}\left(\begin{array}{c} 2n-2 \\ n-1 \end{array}\right) = \Omega\left(\frac{4^n}{(n-1)^{3/2}}\right).$$

- Since this is clearly exponential, this is not a feasible solution.
- We can solve this problem much faster than this, however.
- The key is to notice that an optimal solution contains optimal solutions to subproblems.

An Optimal Solution

- An optimal solution will look like $(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$, where $1 \leq k < n$, and the two halves are parenthesized in some way.
- **Example:** When computing $A_1 A_2 A_3 A_4 A_5$, the final product will be one of:
  - $(A_1)(A_2 A_3 A_4 A_5)$
  - $(A_1 A_2)(A_3 A_4 A_5)$
  - $(A_1 A_2 A_3)(A_4 A_5)$
  - $(A_1 A_2 A_3 A_4)(A_5)$
- The cost of the optimal solution is the cost of each of the two subsolutions plus the cost of multiplying the final two matrices.
- **Example:** If we use $(A_1 A_2)(A_3 A_4 A_5)$, the cost is the cost to compute $A_1 A_2$ plus the cost to compute $A_3 A_4 A_5$ plus the cost to compute the product of these two matrices.

An Optimal Solution

- If the optimal solution has the final product

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

  then the parenthesizations of $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$ must be optimal.
  - Prove it by contradiction.
- That is, subsolutions within an optimal solution are optimal solutions to subproblems (or subsolutions).
- Thus, to compute the optimal solution, we can compute all optimal subsolutions.
- We will start with all optimal subsolutions of size 1, then compute all optimal subsolutions of size 2 *based on* subsolutions of size 1. We continue in this fashion until we have the solution for n.

Recursive Cost Function

- Let $M[i, j]$ be the cost of the optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$.
- The ultimate goal is to compute $M[1, n]$.
- **Example:**
  - Consider the optimal way to compute $A_1 A_2 A_3 A_4 A_5$, where A_i is a $p_{i-1} \times p_i$ matrix.
  - If it has the final split $(A_1 A_2)(A_3 A_4 A_5)$, then the optimal cost is $M[1, 2] + M[3, 5] + p_0 p_2 p_5$, where A_1 is $p_0 \times p_1$
  - However, there are 3 other possible final splits as we saw earlier.
- To compute $M[i, j]$, we need to consider all of the possible splits. since we don't know which is optimal.
- Since the split occurs at some k, where $i \leq k < j$, we can compute

$$M[i, j] = \min_{i \leq k < j} \{M[i, k] + M[k + 1, j] + p_{i-1} p_k p_j\}.$$

Solving MCM

- We just saw that we can define $M[i,j]$ recursively by

$$M[i,j] = \min_{i \le k < j} \{M[i,k] + M[k+1,j] + p_{i-1}p_k p_j\}.$$

- Also notice that $M[i,i] = 0$ for all i.
- We have a **base case** and a **recursive definition** for $M[i,j]$.
- Based on these facts, it is natural to consider using a simple recursive algorithm to compute $M[1,n]$.
- Let's consider how well this will work.

MCM Recursion Tree

- The recursion tree for the MCM problem with 4 matrices:

M[1,4]

=Already Computed

M[1,1] M[2,4] M[1,2] M[3,4] M[1,3] M[4,4]

M[2,2] M[3,4] M[2,3] M[4,4] M[1,1] M[2,2] M[3,3] M[4,4] M[1,1] M[2,3] M[1,2] M[3,3]

M[3,3] M[4,4] M[2,2] M[3,3] M[2,2] M[3,3] M[1,1] M[2,2]

- Notice that many of the calls are repeated.
- In general, the recursive algorithm has complexity

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \geq 2^{n-1}$$

Bottom-Up Solution

- If we have the optimal solutions to all subproblems involving k matrices, we can use the recurrence to find optimal solutions to subproblems involving $k+1$ matrices without any recursive calls. Why?
- In this case, we can compute each optimal subsolution in at most n steps. Why?
- There are about $n^2/2$ subproblems.
- Thus, computing all of the optimal subsolutions, including the solution itself, from the bottom up requires $O(n^3)$ time.
- In summary, to solve the problem, we compute all subproblems involving 1 matrix (trivial), then 2 matrices, then 3 matrices, . . ., then n matrices.
- All will be made clear by the following example.

MCM Example

- We are given matrices with sizes 4×10, 10×3, 3×12, 12×20, and 20×7.
- The sequence of p values is $\{4, 10, 3, 12, 20, 7\}$ ($p_0 = 4$, $p_1 = 10$, etc.)
- We need to compute $M[i, j]$, $0 \leq i, j \leq 5$.
- We know $M[i, i] = 0$ for all i.
- We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

	1	2	3	4	5
1	0	120			
2		0	360		
3			0	720	
4				0	1680
5					0

MCM Example (continued)

- Now products of 3 matrices. $p = \{4, 10, 3, 12, 20, 7\}$

$$M[1,3] = \min \begin{cases} M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \\ M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \end{cases}$$

$$M[2,4] = \min \begin{cases} M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \\ M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \end{cases}$$

$$M[3,5] = \min \begin{cases} M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \\ M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \end{cases}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example (continued again)

- Now products of 4 matrices. $p = \{4, 10, 3, 12, 20, 7\}$

$$M[1,4] = \min \begin{cases} M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \end{cases}$$

$$M[2,5] = \min \begin{cases} M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \end{cases}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example (continued *again*)

- Now the product of 5 matrices. $p = \{4, 10, 3, 12, 20, 7\}$

$$M[1,5] = \min \begin{cases} M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \end{cases}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example: Optimal Cost and Solution

- The optimal cost is $M[1,5] = 1344$.
- What is the optimal parenthesization?
- We didn't keep track of enough information to find that out.
- We can modify the algorithm slightly and keep enough information to get the optimal parenthesization.
- Each time we find the optimal value for $M[i,j]$, we also store the value of k that we used.
- Then we can just work backwards, partitioning the matrices according to the optimal split.

MCM Example: Optimal Solution

- If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

- The k value for the solution is 2, so we have $((A_1A_2)(A_3A_4A_5))$.
- The first half is done.
- The optimal solution for the second half comes from entry $M[3,5]$.
- The value of k here is 4, so now we have $((A_1A_2)((A_3A_4)A_5))$.
- Thus the optimal solution is to parenthesize as $((A_1A_2)((A_3A_4)A_5))$,

Dynamic Programming

- As we have seen, dynamic programming is a technique that can be used to find optimal solutions to certain problems.
- Dynamic programming works when a problem has the following characteristics:
  - **Optimal Substructure:** If an optimal solution contains optimal subsolutions, then a problem exhibits *optimal substructure*.
  - **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has *overlapping subproblems*.
- Dynamic programming takes advantage of these facts to solve problems more efficiently.
- It works well only for problems which have *both* properties. Let's consider why next.

Why It Works

- If a problems has *optimal substructure*, then we can recursively define an optimal solution.
- If a problem has *overlapping subproblems*, then we can improve on a recursive implementation by computing each subproblem only once.
- If a problem doesn't have *optimal substructure*, there is no basis for defining a recursive algorithm to find the optimal solutions.
- If a problem doesn't have *overlapping subproblems*, we really don't have anything to gain by using dynamic programming.
- If the space of subproblems is small enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Memoization

- So far we have talked about implementing dynamic programming in a bottom-up fashion.
- Dynamic programming can also be implemented using **memoization**.
- With memoization, we implement the algorithm recursively, but we keep track of all of the subsolutions.
  - If we encounter a subproblem that we have seen, we look up the solution.
  - If we encounter a subproblem that we have not seen, we compute it, and add it to the list of subsolutions we have seen.
- When is it better to use memoization?
- When is it better to use the bottom-up approach?

0-1 Knapsack Problem

- A thief walks into a house to steal stuff.
- In the house there are n objects.
- The ith object weighs w_i pounds and is worth v_i dollars, where w_i and v_i are integers.
- The thief wants to maximize the value of the stuff he steals, but he can only carry W pounds, where W is an integer.
- Which objects should the thief take to get the most dollars?
- This is called the **0-1 knapsack problem** because the thief must either take or not take each object.
- This is different than the **fractional knapsack problem** in which the thief is allowed to take fractions of objects.
- We will see the fractional knapsack problem again when we talk about greedy algorithms.

Solving the 0-1 Knapsack

- Notice that this is an optimization problem.
- Does it exhibit the desired properties?
- We need to show two things
  - An optimal solution contains optimal subsolutions.
  - The subproblems are solved repeatedly in a recursive implementation.
- Let $\{x_1, x_2, \ldots, x_k\}$ be the objects in an optimal solution.
- The optimal value is $V = v_{x_1} + v_{x_2} + \cdots + v_{x_k}$.
- It is clear that $w_{x_1} + w_{x_2} + \cdots + w_{x_k} \leq W$.

Optimal Substructure

- **Claim:** If $\{x_1, x_2, \ldots, x_k\}$ is an optimal solution to the knapsack problem with weight W, then $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.
- **Proof:** Assume $\{x_1, x_2, \ldots, x_{k-1}\}$ is not an optimal solution to the subproblem. Then there are objects $\{y_1, y_2, \ldots, y_l\}$ such that

$$w_{y_1} + w_{y_2} + \cdots + w_{y_l} \leq W',$$

and

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}}.$$

Then

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} + v_{x_k} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}} + v_{x_k}.$$

But then the set $\{x_1, x_2, \ldots, x_k\}$ is not an optimal solution to the knapsack problem with weight W. This contradicts our assumption. Thus $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Recursive Formula

- Let $K[i, j]$ be the maximal value for the knapsack problem involving the first i objects and weight j, where $1 \leq i \leq n$, and $0 \leq j \leq W$.
- It is not hard to see that

$$K[1, j] = \begin{cases} v_1 & \text{if } w_1 \leq j \\ 0 & \text{if } w_1 > j. \end{cases}$$

- To compute $K[i, j]$, notice that
  - if we add the ith element to the knapsack, the sack had weight $j - w_i$ before it was added, and
  - if we don't add the ith element, then $K[i, j] = K[i - 1, j]$.
- Thus,

$$K[i, j] =$$

$$\begin{cases} K[i-1, j] & \text{if } w_i > j \\ max\{K[i-1, j], K[i-1, j - w_i] + v_i\} & \text{if } w_i \leq j. \end{cases}$$

- The maximal value is $K[n, W]$.

Overlapping Subproblems

- We have seen that the maximal value is $K[n, W]$.
- Computing $K[n, W]$ recursively is not efficient.
- To see this, notice that to find $K[i, j]$, one has to solve 2 problems of size $i - 1$.
- Thus, the time to compute $K[n, W]$ is given by $T(n) = 2T(n - 1) + C$.
- Solving this yields $T(n) = \Omega(2^n)$.
- Notice that the number of subproblems is nW.
- Thus, if $nW < 2^n$, then the 0-1 knapsack problem will certainly have overlapping subproblems, and dynamic programming is appropriate.

0-1 Knapsack Algorithm

- The dynamic programming algorithm is now straightforward.
- The objects are numbered 0 through $n-1$ in this implementation.
- The algorithm:

```
int 0-1-Knapsack(int *w,int *v,int n, int W) {
   int K[n][W+1];
   int i=0;
   while(w[0] > i) K[0][i++]=0;
   while(i<n)      K[0][i++]=v[0];

   for(int i=1;i<n;i++) {
      for(int j=0;j<=W;j++)
         {
         if(j>=w[i] && K[i-1][j-w[i]] + v[i] > K[i-1][j])
             K[i][j] = K[i-1][j-w[i]] + v[i];
         else
             K[i][j] = K[i-1][j];
         }
      }
   return K[n,W];
}
```

0-1 Knapsack Example

- We have the following instance of 0-1 knapsack, with $W = 10$:

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

- We solve it from the bottom up:

$i\backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

- So the optimal value is 26.
- Which items do we take?

Finding the Knapsack

- How do we compute an optimal knapsack?
- With this problem, we don't have to keep track of anything extra.
- Let $K[n, k]$ be the maximal value.
- If $K[n, k] \neq K[n-1, k]$, then $K[n, k] = K[n-1, k-w_n] + v_n$, and the nth item is in the knapsack.
- Otherwise, we know $K[n, k] = K[n-1, k]$, and we assume that the nth item is not in the optimal knapsack. (Could it be?)
- In either case, we have an optimal solution to a subproblem.
- Thus, we continue the process with either $K[n-1, k]$ or $K[n-1, k-w_n]$, depending on whether n was in the knapsack or not.
- When we get to the $K[1, k]$ entry, we take item 1 if $K[1, k] \neq 0$ (equivalently, when $k \geq w[1]$)

Finishing the Example

- Recall we had:

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

- We work backwards through the table

$i\backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	**7**	7	7	7	7	7	7	7
2	0	0	10	10	10	**17**	17	17	17	17	17
3	0	0	10	10	10	**17**	17	17	17	17	17
4	0	3	10	13	13	17	**20**	20	20	20	20
5	0	3	10	13	13	17	**20**	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	**26**

- It is easy to see that the optimal knapsack should contain $\{1, 2, 4, 6\}$