EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Dynamic Programming

Readings:

- Weiss 10.3
- Horowitz 5.1
- **Cormen 15**

Short list of Algorithm Designs

- Brute force algorithms
- Simple recursive algorithms
- Divide and conquer algorithms
- Greedy algorithms
- **Dynamic programming algorithms**
- Backtracking algorithms
- Branch and bound algorithms

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- "Programming" here means "planning"

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- "Programming" here means "planning"

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Backstory: Divide-and-Conquer

- **Divide-and-conquer** method for algorithm design:

- **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems

Backstory: Divide-and-Conquer

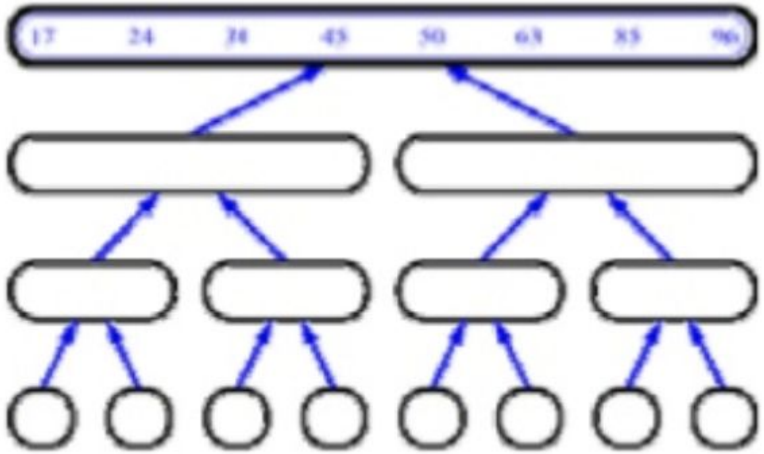
- **Divide-and-conquer** method for algorithm design:
- **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
- **Conquer**: conquer recursively to solve the subproblems
- **Combine**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

Backstory: Divide-and-Conquer

- For example, **MergeSort**

```
Merge-Sort(A, p, r)
   if p < r then
      q←(p+r)/2
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

- The **subproblems** are **independent**, **all different.**



DP: Efficient Divide-and-Conquer

- **Dynamic programming** is a way of improving on **inefficient** *divide-and-conquer* algorithms.

- By "*inefficient*", we mean that ***the same recursive call is made over and over.***

DP: Efficient Divide-and-Conquer

- **Dynamic programming** is a way of improving on **inefficient** *divide-and-conquer* algorithms.

- By "*inefficient*", we mean that ***the same recursive call is made over and over.***

- If **same** *subproblem* is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.

DP: Efficient Divide-and-Conquer

- **Dynamic programming** is a way of improving on **inefficient** *divide-and-conquer* algorithms.

- By "*inefficient*", we mean that ***the same recursive call is made over and over.***

- If **same** *subproblem* is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.

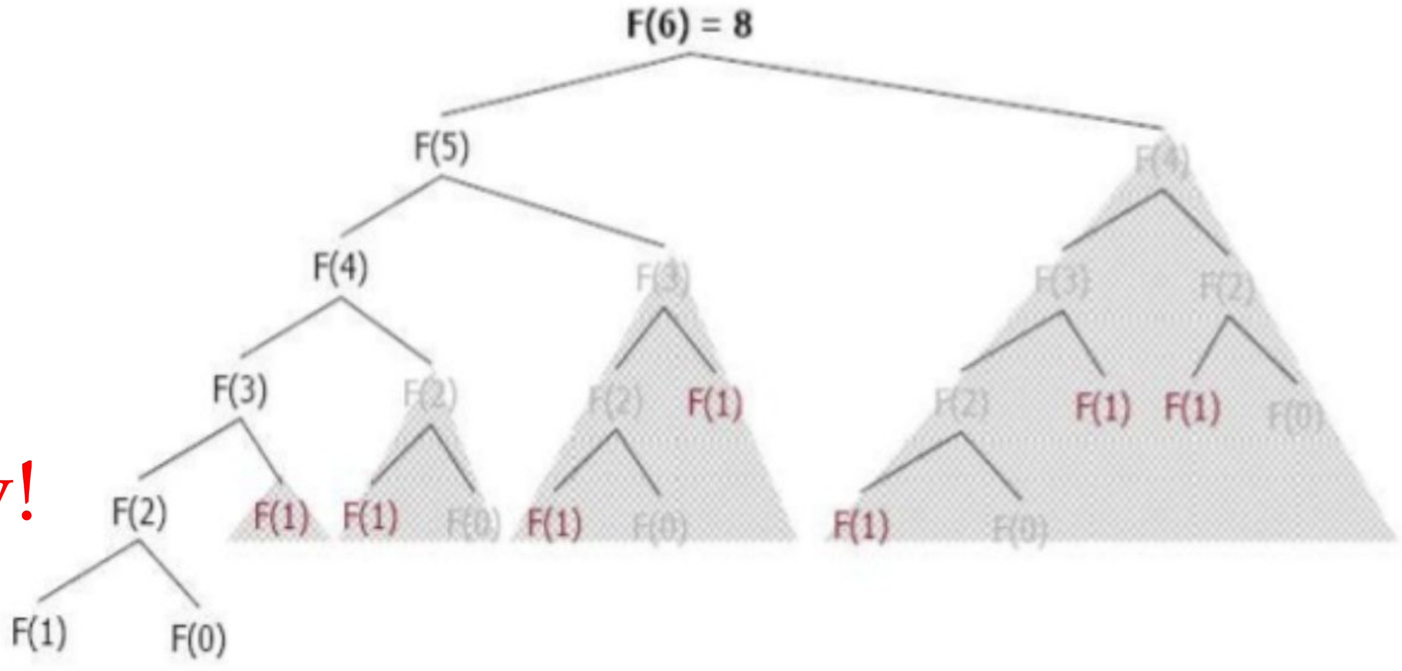
- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share subsubproblems.

Example 1: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

Exponential time complexity!



- We keep calculating the same value over and over!

Example 1: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- We can calculate *Fn* in ***linear* time** by remembering solutions to the solved subproblems – *dynamic programming*

- Compute solution in a bottom-up fashion

- In this case, only two values need to be remembered at any time

```
Fibonacci(n)
    F0←0
    F1←1
    for i ← 2 to n do
        Fi ← Fi-1 + Fi-2
```

Linear Fibonacci Code

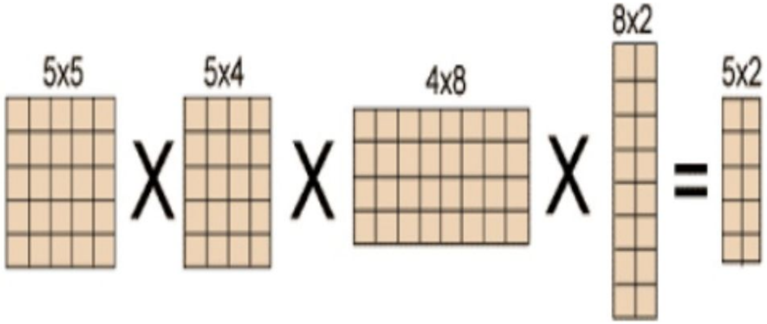
```
long fibonacci( int n ) {
   if( n == 1 )
      return 1;

   // initialize the "table"
   long last = 1;
   long nextToLast = 0;
   long answer = 1;

   for( int i = 2; i <= n; ++i ) {
      answer = last + nextToLast; // combine subproblem solutions
      nextToLast = last; // update subproblem solutions
      last = answer;
   }

   return answer;
}
```

Sequential Viewpoint on DynProg

- **Problem:** compute a sequence of decisions with the minimal cost
  - An **optimal solution** minimizes the cost
  - E.g., which edges to take for a shortest path in a graph
- *Brute force:* try all sequences of decisions
- *Greedy approach:* use the local information to make **one decision at a time**
  - What if the *"local information"* is **insufficient** for that?
- *Dynamic programming:* work out the **optimal decision subsequences**, even if not from the start (often from the end)
  - Need to know how the optimal sequence & costs are <u>structured</u>
  - <u>Optimal sequence</u> is composed of <u>optimal subsequences</u>
  - Suboptimal subsequences are not considered

Matrix Chain Ordering Problem

A1 A2 A3 A4



- Given : a chain of matrices $\{A_1, A_2, \ldots, A_n\}$.
- Once all pairs of matrices are *parenthesized*, they can be multiplied by using the standard algorithm as a sub-routine.
- A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. [Note: since matrix multiplication is associative, all parenthesizations yield the same product.]

((A1 x A2) x A3) x A4

A1 x (A2 x (A3 x A4))

(A1 x A2) x (A3 x A4)

Matrix Chain Ordering Problem

The way the chain is parenthesized can have a dramatic impact on the cost of evaluating the product.

Fact: multiplying *p x q* and *q x r* matrices leads to a *p x r* matrix and takes *pqr* scalar products

- Example: A[30][35], B[35][15], C[15][5]

  minimum of A*B*C

  A*(B*C) = 30*35*5 + 35*15*5 = 7,585

  (A*B)*C = 30*35*15 + 30*15*5 = 18,000

Problem: Find the parenthesization of matrices with the **least** number of scalar products for computing their product

Towards a Good Solution

Step 1: Let's first do a recursive characterization

High-Level Parenthesization for $A_{i..j}$

For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k. That is,

$$A_{i..j} = (A_i \cdots A_k)(A_{k+1} \cdots A_j) = A_{i..k} A_{k+1..j}.$$

Question: how many ways to multiply 4 matrices? What are the k's?

Product of matrices from A_i to A_j

Algorithm's goal: find the best ***k*** where to make a split

Towards a Good Solution

Step 2: For a decomposition at k, we can develop a formula for the **cost of multiplication**

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

Cost of multiplying matrices from A_i to A_j

Numbers of columns in matrices: columns of A_{i-1} (aka the rows of A_i). columns of A_k, and columns of A_j

We don't know what the **best** ***k*** is, but there are only (j – i) possible choices of k, so we can try them all and find the one with the smallest cost

Good Solution

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Good Solution

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.
$m[i, j]$ only defined for $i \leq j$.

Good Solution

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.
$m[i, j]$ only defined for $i \leq j$.

The important point is that when we use the equation

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j)$$

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k+1, j]$. For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$. Hence, the algorithm should fill the table in increasing order of the length of the matrix-chain.

Good Solution

$$m[1,2], m[2,3], m[3,4], \ldots, m[n-3,n-2], m[n-2,n-1], m[n-1,n]$$

$$m[1,3], m[2,4], m[3,5], \ldots, m[n-3,n-1], m[n-2,n]$$

$$m[1,4], m[2,5], m[3,6], \ldots, m[n-3,n]$$

$$\vdots$$

$$m[1,n-1], m[2,n]$$

$$m[1,n]$$

Optimal Matrix Ordering Code

```
// p: column counts of matrices, except p[0] is the row count for the first matrix
int optMatrix( const vector<int> & p ) {
    int n = c.size( ) - 1;
    // initialize the table
    matrix<int> m;
    for( int i = 1; i <= n; ++i )
        m[i][i] = 0;

    for( int row = 1; row < n; ++row ) // row is right - left
        for( int left = 1; left <= n - row; ++left ) {
            // for each position
            int right = left + row;
            m[left][right] = INT_MAX; // same as infinity
            for( int k = left; k < right; ++k ) {
                int thisCost = m[left][k] + m[k+1][right] + p[left-1]*p[k]*p[right];
                if( thisCost < m[left][right] ) // update min cost
                    m[left][right] = thisCost;
            }
    }
    return m[1][n];
}
```

Divide & Conquer / Recursion vs DynProg

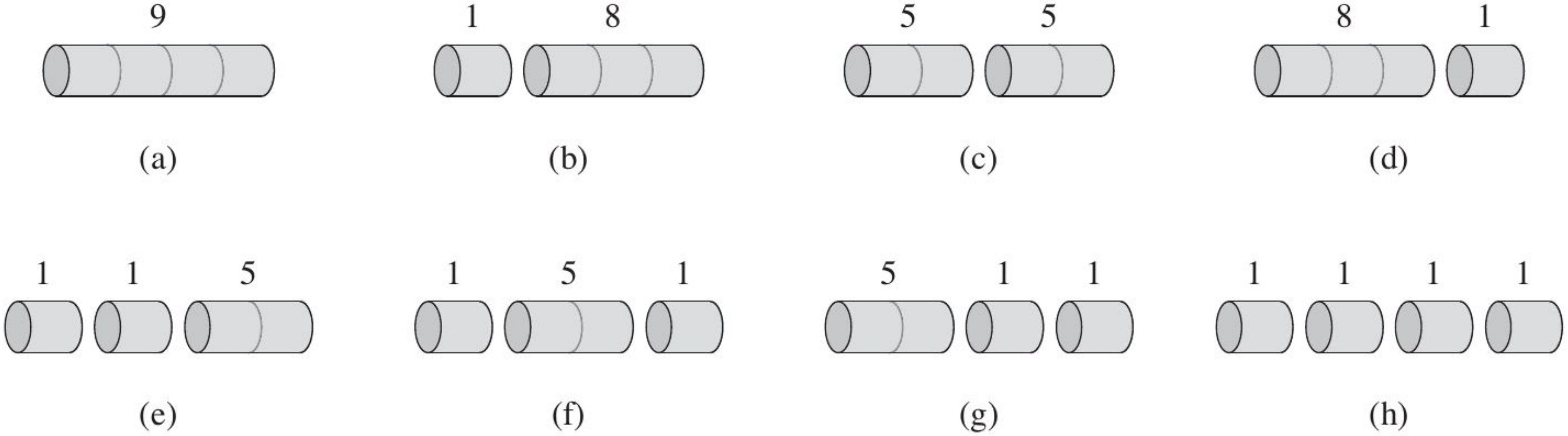
Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Class Exercise: Rod Cutting

- Given a **rod** of N inches (integer)
- Given a table of **prices** for pieces of different length:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- *Goal:* cut the rod into pieces with the **most expensive total**



Class Exercise: Rod Cutting

Knapsack Problem

- Given **N** items with **weights** $w_1 \ldots w_N$ and **prices** $p_1 \ldots p_N$
- How to fit items with **maximum total price** into a knapsack, which holds max weight **W**?
    a. *Solution format:* an array of N booleans, indicating items to take
- *Naive approach:* try all combinations of items, 2^N time complexity
- *Greedy approach* won't ensure an optimal solution

Example: weights {1, 2, 3}, prices {10, 15, 40}, W = 6

Recursion for Knapsack Problem

- Given **N** items with **weights** $w_1 \ldots w_N$ and **prices** $p_1 \ldots p_N$
- Maximize **maximum total price** into under max weight **W**?
- *Recursion with saving intermediate results (aka* ***memoization****):*
  a. Suppose we solved the problem for the first m items: `price(m, W)`
  b. **Price recurrence** for item m+1: `price(m+1, W)`. Choose the max of:
     - p_{m+1} + `price(m, W-w`$_{m+1}$`) // add item m+1`
     - 0 + `price(m, W-w`$_{m+1}$`) // do not add item m+1`
  c. Maintain a 2D table of best prices with indices:
     `M[X first items considered][Y unused weight]=max(..)`

Time complexity **N*W**, space complexity **N*W**

Recursion for Knapsack Problem

- **Example:** weights {1, 2, 3}, prices {10, 15, 40}, W = 6
- Work through the *recursion with* ***memoization***:

```
if (M[m+1][W] < infinity)
   return M[m+1][W]
price(m+1, W) = max (p_{m+1} + price(m, W-w_{m+1}), price(m, W-w_{m+1}))
M[m+1][W] = price(m+1, W)
```

DynProg for Knapsack Problem

1. *Subproblem representation:* price of a subset of items with a reduced knapsack capacity, so the matrix is as in the recursion:

   `[X first items considered] [Y unused weight]`

2. Fill out the matrix row-by-row the lower item counts

Example: weights {1, 2, 3}, prices {10, 15, 40}, W = 6

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Time complexity **N*W**, space complexity **N*W**