Algorithms and Data Structures
Lecture X

This Lecture

- Dynamic programming
  - Fibonacci numbers example
  - Optimization problems
  - Matrix multiplication optimization
  - Principles of dynamic programming
  - Longest Common Subsequence

Divide and Conquer

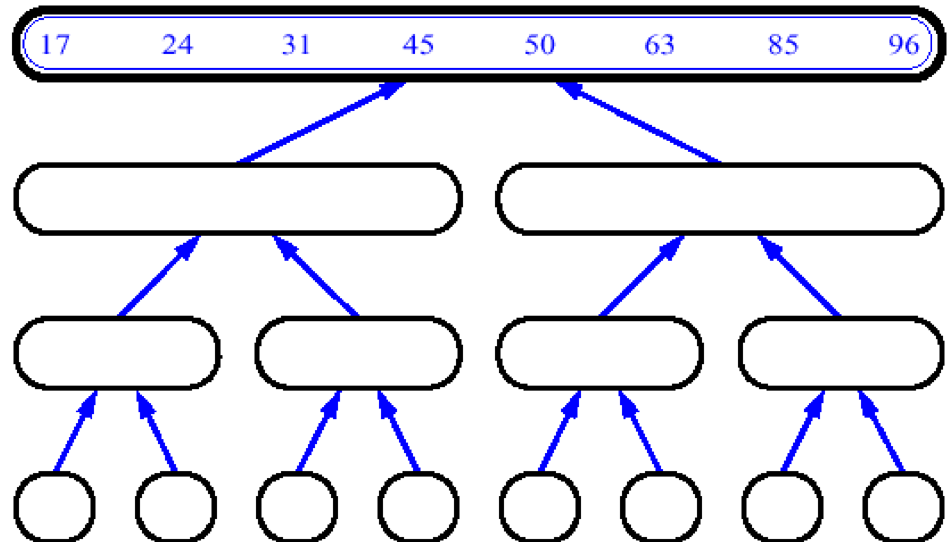
- *Divide and conquer* method for algorithm design:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - **Conquer**: Use divide and conquer recursively to solve the subproblems
  - **Combine**: Take the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

Divide and Conquer (2)

- 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)
```

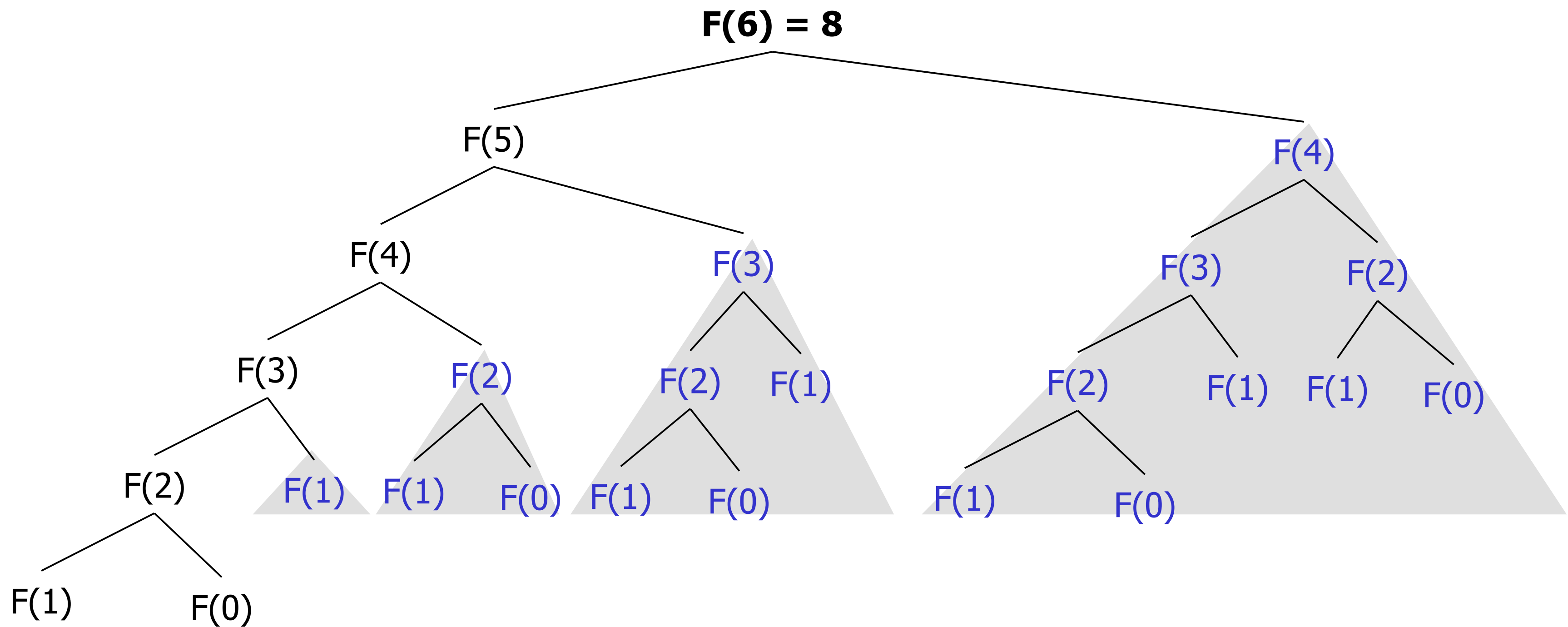
- The subproblems are independent, all different



Fibonacci Numbers

- $F_n = F_{n-1} + F_{n-2}$
- $F_0 = 0, F_1 = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …
- Straightforward recursive procedure is slow!
- Why? How slow?
- Let’s draw the recursion tree

Fibonacci Numbers (2)



- We keep calculating the same value over and over!

Fibonacci Numbers (3)

- How many summations are there?
- Golden ratio $\frac{F_{n+1}}{F_n} \approx \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803...$
- Thus $F_n \approx 1.6^n$
- Our recursion tree has only 0s and 1s as leaves, thus we have $\approx 1.6^n$ summations
- Running time is *exponential!*

Fibonacci Numbers (4)

- We can calculate F_n in *linear* time by remembering solutions to the solved subproblems – *dynamic programming*
- Compute solution in a bottom-up fashion
- Trade space for time!
  - In this case, only two values need to be remembered at any time

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

Optimization Problems

- We have to choose one solution out of many – a one with the optimal (minimum or maximum) value.
- A solution exhibits a structure
  - It consists of a string of choices that were made – what choices have to be made to arrive at an optimal solution?
- The algorithms computes the optimal value plus, if needed, the optimal solution

Multiplying Matrices

- Two matrices, *A – n×m* matrix and *B – m×k* matrix, can be multiplied to get C with dimensions *n×k*, using *nmk* scalar multiplications

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \dots & \dots & \dots \\ \dots & c_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \qquad c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$$

- Problem: Compute a product of many matrices efficiently
- Matrix multiplication is *associative*
  - *(AB)C = A(BC)*

Multiplying Matrices (2)

- The parenthesization matters
- Consider *A*×*B*×*C*×*D*, where
  - *A* is 30×1, *B* is 1×40, *C* is 40×10, *D* is 10×25
- Costs:
  - *(AB)C)D* = 1200 + 12000 + 7500 = 20700
  - *(AB)(CD)* = 1200 + 10000 + 30000 = 41200
  - *A((BC)D)* = 400 + 250 + 750 = 1400
- We need to optimally parenthesize

$A_1 \times A_2 \times \ldots \times A_n$, where A_i is a $d_{i-1} \times d_i$ matrix

Multiplying Matrices (3)

- Let $M(i,j)$ be the *minimum* number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$
- Key observations
  - The outermost parenthesis partition the chain of matrices (i,j) at some k, ($i \le k < j$): $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$
  - The optimal parenthesization of matrices (i,j) has optimal parenthesizations on either side of k: for matrices (i,k) and $(k+1,j)$

Multiplying Matrices (4)

- We try out all possible *k*. Recurrence:

$$M(i,i) = 0$$

$$M(i,j) = \min_{i \le k < j} \{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\}$$

- A direct recursive implementation is exponential – there is a lot of duplicated work (why?)
- But there are only $\binom{n}{2} + n = \Theta(n^2)$ different subproblems *(i,j)*, where $1 \le i \le j \le n$

Multiplying Matrices (5)

- Thus, it requires only $\Theta(n^2)$ space to store the optimal cost *M(i,j)* for each of the subproblems: half of a 2d array *M[1..n,1..n]*

```
Matrix-Chain-Order(d0…dn)
1   for i←1 to n do
2       M[i,i] ← 0
3   for l←2 to n do
4       for i←1 to n-l+1 do
5           j ← i+l-1
6           M[i,j] ← ∞
7           for k←i to j-l do
8               q ← M[i,k]+M[k+1,j]+d(i-1)d(k)d(j)
9               if q < M[i,j] then
10              M[i,j] ← q
11              c[i,j] ← k
12  return M, c
```

Multiplying Matrices (6)

- After execution: *M[1,n]* contains the value of the optimal solution and *c* contains optimal subdivisions (choices of *k*) of any subproblem into two subsubproblems
- A simple recursive algorithm **Print-Optimal-Parens**(*c, i, j*) can be used to reconstruct an optimal parenthesization
- Let us run the algorithm on
  d = [10, 20, 3, 5, 30]

Multiplying Matrices (7)

- Running time
  - It is easy to see that it is $O(n^3)$
  - It turns out, it is also $\Omega(n^3)$
- From exponential time to polynomial

Memoization

- If we still like recursion very much, we can structure our algorithm as a recursive algorithm:
  - Initialize all *M* elements to ∞ and call **Lookup-Chain**(d, i, j)

```
Lookup-Chain(d,i,j)
1  if M[i,j] < ∞ then
2     return m[i,j]
3  if i=j then
4     m[i,j] ←0
5  else for k ←i to j-1 do
6           q ←Lookup-Chain(d,i,k)+
              Lookup-Chain(d,k+1,j)+d_{i-1}d_kd_j
7           if q < M[i,j] then
8              M[i,j] ←q
9  return M[i,j]
```

Dynamic Programming

- In general, to apply dynamic programming, we have to address a number of issues:
  - 1. Show **optimal substructure** – an optimal solution to the problem contains within it optimal solutions to subproblems
    - Solution to a problem:
      - Making a choice out of a number of possibilities (look what possible choices there can be)
      - Solving one or more subproblems that are the result of a choice (characterize the space of subproblems)
    - Show that solutions to subproblems must themselves be optimal for the whole solution to be optimal (use "cut-and-paste" argument)

Dynamic Programming (2)

- 2. Write a recurrence for the value of an optimal solution
  - $M_{opt} = Min_{\text{over all choices } k}$ {(Sum of M_{opt} of all subproblems, resultnig from choice *k*) + (the cost associated with making the choice *k*)}
- Show that the number of different instances of subproblems is bounded by a polynomial

Dynamic Programming (3)

- 3. Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary subresults precomputed (or use memoization)
- See if it is possible to reduce the space requirements, by "forgetting" solutions to subproblems that will not be used any more
- 4. Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution)

Longest Common Subsequence

- Two text strings are given: *X* and *Y*
- There is a need to quantify how similar they are:
  - Comparing DNA sequences in studies of evolution of different species
  - Spell checkers
- One of the measures of similarity is the length of a Longest Common Subsequence (LCS)

LCS: Definition

- Z is a subsequence of X, if it is possible to generate Z by skipping some (possibly none) characters from X
- For example: X =“ACGGTTA”, Y=“CGTAT”, LCS(X,Y) = “CGTA” or “CGTT”
- To solve LCS problem we have to find “skips” that generate LCS(X,Y) from X, and “skips” that generate LCS(X,Y) from Y

LCS: Optimal Substructure

- We make Z to be empty and proceed from the ends of X_m="$x_1 x_2 \ldots x_m$" and Y_n="$y_1 y_2 \ldots y_n$"
  - If $x_m = y_n$, append this symbol to the beginning of Z, and find optimally LCS(X_{m-1}, Y_{n-1})
  - If $x_m \neq y_n$,
    - Skip either a letter from X
    - or a letter from Y
    - Decide which decision to do by comparing LCS(X_m, Y_{n-1}) and LCS(X_{m-1}, Y_n)
  - "Cut-and-paste" argument

LCS: Reccurence

- The algorithm could be easily extended by allowing more "editing" operations in addition to *copying* and *skipping* (e.g., changing a letter)
- Let c[i,j] = LCS(X_i, Y_j)

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Observe: conditions in the problem restrict subproblems (What is the total number of subproblems?)

LCS: Compute the Optimum

```
LCS-Length(X, Y, m, n)
1   for i←1 to m do
2       c[i,0] ← 0
3   for j←0 to n do
4       c[0,j] ← 0
5   for i←1 to m do
6       for j←1 to n do
7           if x_i = y_j then
8               c[i,j] ← c[i-1,j-1]+1
9               b[i,j] ← "copy"
10          else if c[i-1,j] ≥ c[i,j-1] then
11                  c[i,j] ← c[i-1,j]
12                  b[i,j] ← "skipx"
13              else
14                  c[i,j] ← c[i,j-1]
15                  b[i,j] ← "skipy"
16  return c, b
```

LCS: Example

- Lets run: X ="ACGGTTA", Y="CGTAT"
- How much can we reduce our space requirements, if we do not need to reconstruct LCS?

Next lecture

- Graphs:
  - Representation in memory
  - Breadth-first search
  - Depth-first search
  - Topological sort