

MA/CSSE 473

Day 14

**Strassen's
Algorithm: Matrix
Multiplication**

**Decrease and
Conquer**

DFS



MA/CSSE 473 Day 14

- HW 6 is due Tuesday
- In-class exam (Tuesday, October 5) will cover through Chapter 4 and HW 7.
- **Student Questions**
- Matrix Multiplication (Strassen)
- Decrease and Conquer examples
- DFS



Reminder: The Master Theorem

- The Master Theorem for Divide and Conquer recurrence relations:

- Consider

$$T(n) = aT(n/b) + \Theta(n^k)$$

- The solution is

- $\Theta(n^k)$ if $a < b^k$
- $\Theta(n^k \log n)$ if $a = b^k$
- $\Theta(n^{\log_b a})$ if $a > b^k$



Ordinary Matrix Multiplication

How many additions and multiplications are needed to compute the product of two 2x2 matrices?

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$



Q1

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

Values of M_1, M_2, \dots, M_7 are on the next slide



Formulas for Strassen's Algorithm

$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

How many additions
and multiplications?



Q2

The Recursive Algorithm

- We multiply square matrices whose size is a power of 2 (if not, pad with zeroes)
- Break up each matrix into four $N/2 \times N/2$ submatrices.
- Recursively multiply the parts.
- How many additions and multiplications?
 - If we do "normal matrix multiplication" recursively using divide and conquer?
 - If we use Strassen's formulas?



Analysis of Strassen's Algorithm

If N is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

$$M(N) = 7M(N/2) + C, \quad M(1) = 1$$

Solution: $M(N) = \Theta(N^{\log_2 7}) \approx N^{2.807}$

vs. N^3 of brute-force algorithm.

What if we also count the additions?

Algorithms with better asymptotic efficiency are known but they are even more complex.



DECREASE AND CONQUER



Decrease and Conquer Algorithms

- What does the term mean?
 - Reduce problem instance to smaller instance of the same problem
 - Solve smaller instance
 - Extend solution of smaller instance to obtain solution to original instance
- Also referred to as *inductive* or *incremental* approach
- Can be implemented either top-down or bottom-up
- Three variations. Decrease by
 - constant amount
 - constant factor
 - variable amount



Decrease by constant vs by half

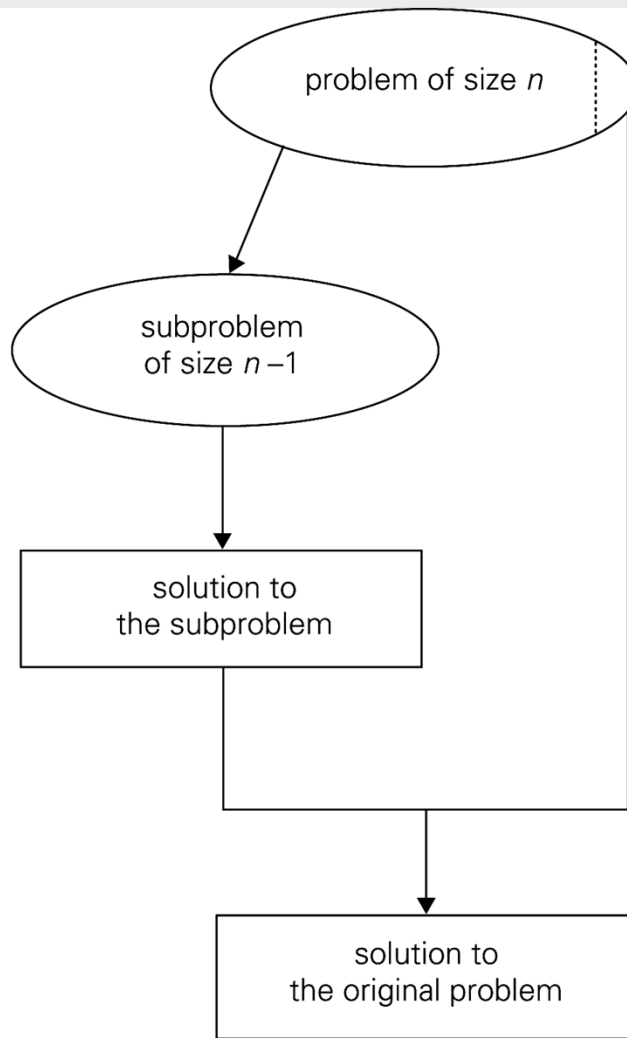


FIGURE 5.1 Decrease (by one)-and-conquer technique

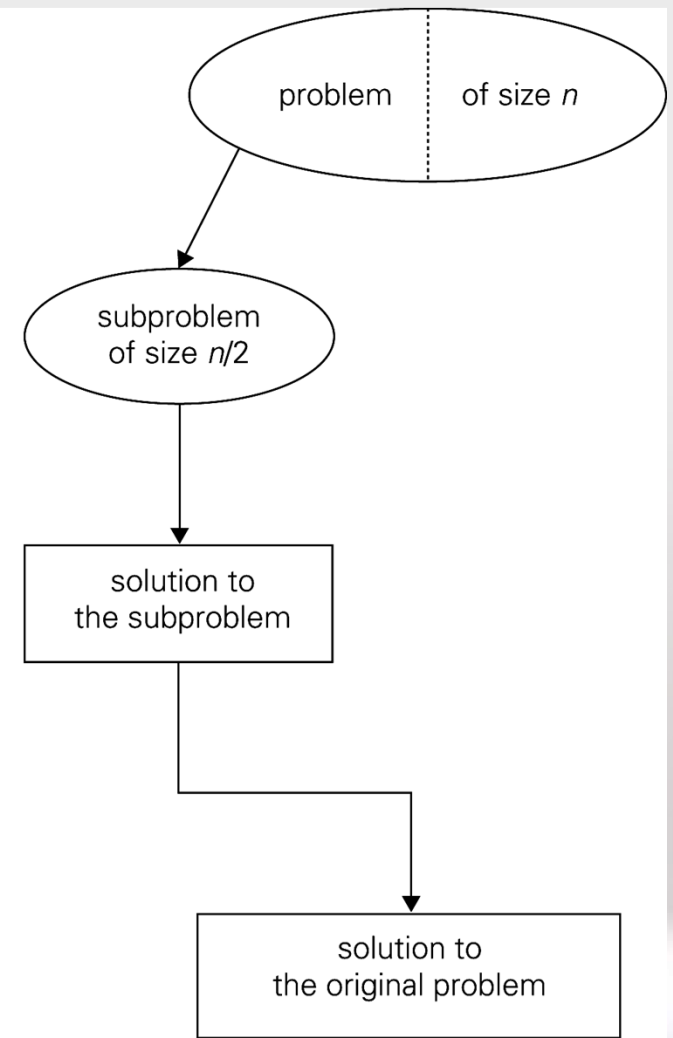


FIGURE 5.2 Decrease (by half)-and-conquer technique

Variable Decrease Simple Example

- Euclid's algorithm
- b and $a \% b$ are smaller than a and b , but not by a constant amount or constant factor



What's the difference?

- Consider the problem of exponentiation:
Compute a^n , where n is a power of 2
 - Brute Force:
 - Divide and conquer:
 - Decrease by one:
 - Decrease by constant factor:



Insertion Sort

- How does "decrease and conquer" apply to insertion sort?
 - Reduce problem instance to smaller instance of the same problem
 - Solve smaller instance
 - Extend solution of smaller instance to obtain solution to original instance
- Example: Sort 6, 4, 1, 8, 5

```
6 | 4 1 8 5
4 | 6 1 8 5
1 4 | 6 8 5
1 4 6 | 8 5
1 4 5 6 | 8
```



Analysis of Insertion Sort

- Time efficiency

$$C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$$

$$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$$

$$C_{best}(n) = n - 1 \in \Theta(n)$$

(also fast on almost-sorted arrays)

- Space efficiency: in-place
(constant extra storage)
- Stable: yes
- Binary insertion sort
 - use Binary search, then move elements to make room for inserted element



Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:

- Depth-first search (DFS)
- Breadth-first search (BFS)



Depth-First Search (DFS)

- Visits a graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available
- Uses a stack
 - a vertex is pushed onto the stack when it's reached for the first time
 - a vertex is popped off the stack when it becomes a dead end, i.e., when there are no adjacent unvisited vertices
- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)
 - A back edge is an edge of the graph that goes from the current vertex to a previously visited vertex (other than the current vertex's parent).



Pseudocode for DFS

ALGORITHM $DFS(G)$

//Implements a depth-first search traversal of a given graph

//Input: Graph $G = \langle V, E \rangle$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they've been first encountered by the DFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

$count \leftarrow 0$

for each vertex v in V **do**

if v is marked with 0

$dfs(v)$

$dfs(v)$

//visits recursively all the unvisited vertices connected to vertex v by a path

//and numbers them in the order they are encountered

//via global variable $count$

$count \leftarrow count + 1$; mark v with $count$

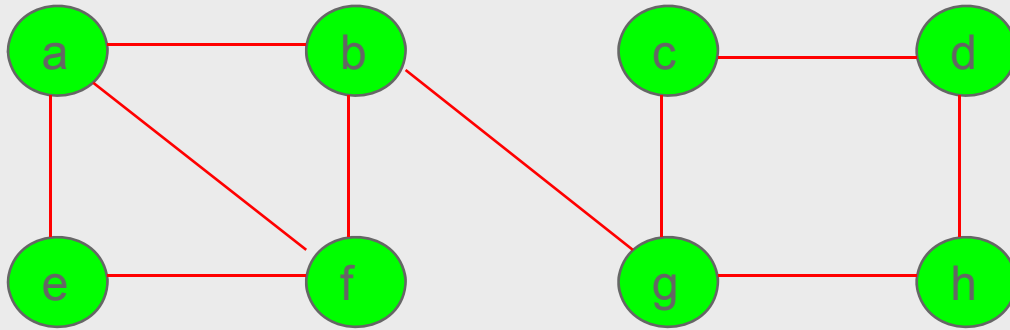
for each vertex w in V adjacent to v **do**

if w is marked with 0

$dfs(w)$



Example: DFS traversal of undirected graph



DFS traversal stack:

DFS tree:



Notes on DFS

- DFS can be implemented with graphs represented as:
 - adjacency matrix: $\Theta(V^2)$
 - adjacency list: $\Theta(|V| + |E|)$
- Yields two distinct ordering of vertices:
 - order in which vertices are first encountered (pushed onto stack)
 - order in which vertices become dead-ends (popped off stack)
- Applications:
 - checking connectivity, finding connected components
 - checking acyclicity
 - finding articulation points
 - searching state-space of problems for solution (AI)

