COP 3530: Computer Science III
Summer 2005

Divide-and-Conquer Algorithms

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
CSB 242, (407)823-2790

Course Webpage:
http://www.cs.ucf.edu/courses/cop3530/sum2005

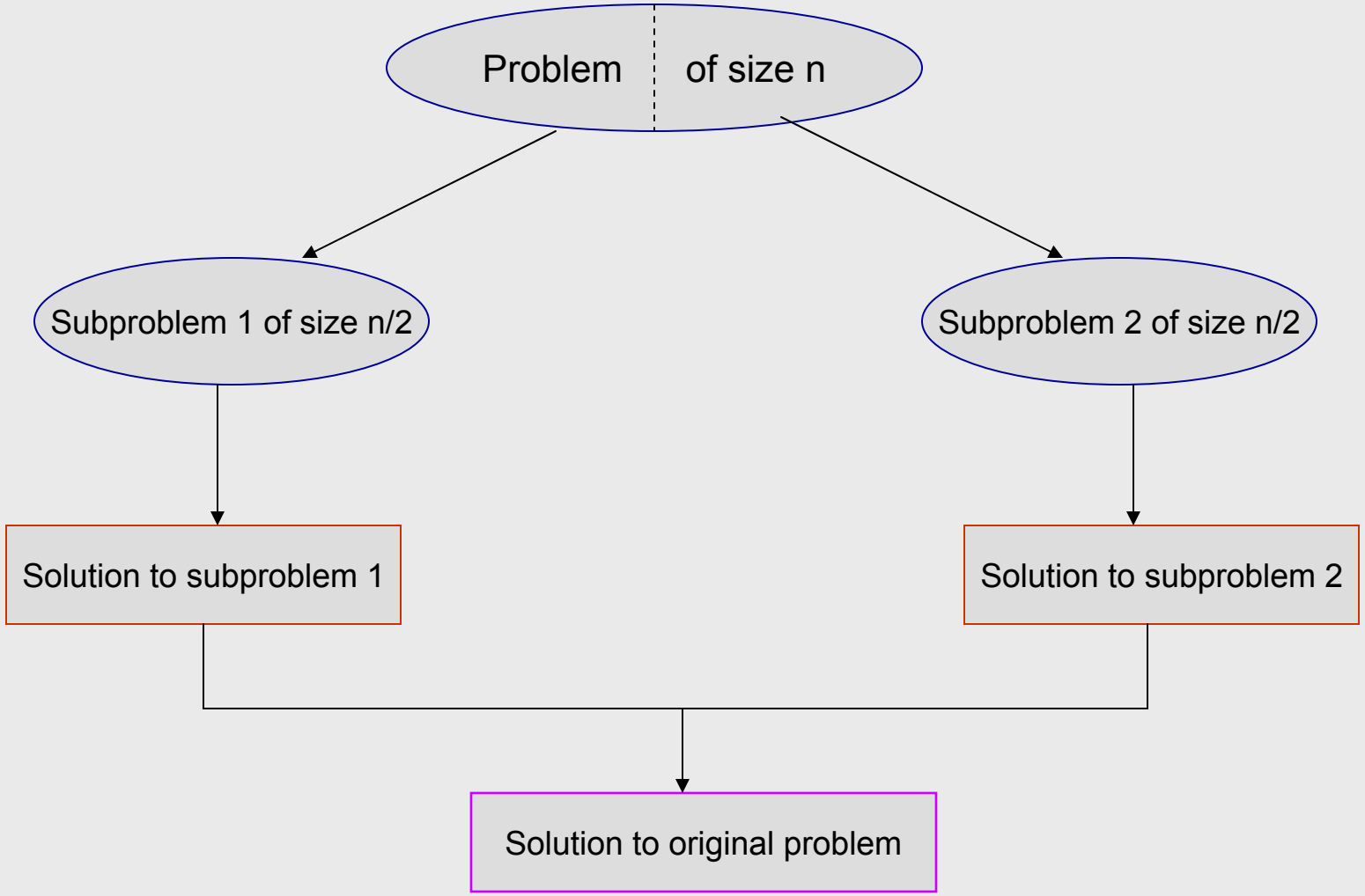
School of Computer Science
University of Central Florida

What Is A Divide-and-Conquer Algorithm?

- Divide-and-conquer is probably the best-known general algorithm design technique.
- A large number of very efficient algorithms are specific implementations of this general strategy.
- Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when the instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original problem.

Typical Divide-and-Conquer Strategy



Divide-and-Conquer Algorithms (cont.)

- The divide-and-conquer technique as diagrammed on the previous page, depicts the case of dividing a problem into two smaller subproblems.
  - This is by far the most commonly occurring case, at least for divide-and-conquer algorithms designed to be executed on a single-processor computer.
- As an example, let's consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n=1$, we simply return a_0 as the answer.)

Divide-and-Conquer Algorithms (cont.)

- Once each of these two sums is computed (by applying the same method recursively), we can add their values together to get the sum in question:

  $a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$

- Is this an efficient way to compute the sum of *n* numbers? Is it more/less efficient than brute force?

  – Consider 1+2+3+4+5+6+7+8+9+10

  = (1+2+3+4+5) + (6+7+8+9+10) = [(1+2) + (3+4+5)] + [(6+7) + (8+9+10)]

  = 1 + 2 + 3 + (4+5) + 6 + 7 + 8 + (9+10)

  = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 (total of 9 addition operations plus 8 splits)

- Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute force solution. However, often it is the case that a divide-and-conquer approach is more efficient than another approach.

Divide-and-Conquer Algorithms (cont.)

- At this point in time, we will consider only sequential algorithms. Later in the semester we will introduce parallel algorithms.
- Until that time, keep in mind that the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.
- The sum example illustrates the most typical case of divide-and-conquer: a problem's instance of size *n* is divided into two instances of size *n/2*.
- More generally, an instance of size *n* can be divided into several instances of size *n/b*, with *a* of them needing to be solved (*a* ≥ 1 and b > 1).

Divide-and-Conquer Algorithms (cont.)

- If size *n* is a power of *b* the following recurrence for the running time T(n) holds:

```
T(n) = aT(n/b) + f(n)
```

- The function *f(n)* accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.
- For the summation example, *a = b = 2* and *f(n) = 1*.
- The recurrence shown above is called the general Divide-and-Conquer recurrence. Obviously, the order of growth of its solution T(n) depends on the values of the constants *a* and *b* as well as the order of growth of the function *f(n)*.

Divide-and-Conquer Algorithms (cont.)

- The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in $T(n) = aT(n/b) + f(n)$, then

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log n\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

Divide-and-Conquer Algorithms (cont.)

- Returning to our summation example, the recurrence equation for the number of additions *A(n)* made by the divide-and-conquer approach on inputs of size $n = 2^k$ is:

```
A(n) = 2A(n/2) + 1
```

- Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$ we have:

$$A(n) \in \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

- Thus, we were able to find the solution's efficiency class without going through the drudgery of solving the recurrence. However, this approach can only establish a solution's order of growth to within an unknown multiplicative constant while solving a recurrence equation with a specific initial condition yields an exact answer (at least for *n*'s that are powers of *b*.)

Computing integer power a^n

- Brute force algorithm

```
Algorithm power(a,n)
    value ← 1
    for i ← 1 to n do
        value ← value × a
return value
```

$$a^n = a \times a \ldots \times a$$

- Complexity: O(n)

Computing integer power a^n

- Divide-and-Conquer algorithm

```
Algorithm power(a,n)
    if (n = 1)
      return a
    partial ← power(a,floor(n/2))
    if n mod 2 = 0
      return partial × partial
    else
      return partial × partial × a
```

$$a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{if } n \text{ is even} \\ a^{\frac{n}{2}} \times a^{\frac{n}{2}} \times a & \text{if } n \text{ is odd} \end{cases}$$

- Complexity: $T(n) = T(n/2) + O(1) \Rightarrow T(n)$ is $O(\log n)$

Integer Multiplication

- Multiply two n-digit integers *I* and *J*.

  ex: 61438521 × 94736407

- Complexity: $O(n^2)$

```
  6 1 4 38521
×9 4 7 3 6 4 0 7
– – – – – – – – – – – – – – –
      4 3 0 0 6 9 6 4 7
      0 0 0 0 0 0 0 0
    2 4 5 7 5 4 0 8 4
…
– – – – – – – – – – – – – – – – –
5 8 2 0 4 6 4 7 3 0 9 3 4 0 4 7
```


Integer Multiplication

- Divide : Split I and J into high-order and low-order digits.
  - ex: $I = 61438521$ is divided into $I_h = 6143$ and $I_l = 8521$
  - *i.e.* $I = 6143 \times 10^4 + 8521$

$$I = I_h\, 10^{n/2} + I_l$$
$$J = J_h\, 10^{n/2} + J_l$$

- Conquer : define $I \times J$ by multiplying the parts and adding

$$I \times J = \left(I_h\, 10^{n/2} + I_l\right) \times \left(J_h\, 10^{n/2} + J_l\right)$$
$$= \underbrace{\left(I_h \times J_h\right)}_{\text{subProblem}_1} 10^n + \left[\underbrace{\left(I_h \times J_l\right)}_{\text{subProblem}_3} + \underbrace{\left(I_l \times J_h\right)}_{\text{subProblem}_4}\right] 10^{n/2} + \underbrace{\left(I_l \times J_l\right)}_{\text{subProblem}_2}$$

Complexity: T(n) = 4T(n/2) + n $\Rightarrow$ T(n) is $O(n^2)$.

Integer Multiplication

- Improved Algorithm

$$I \times J = (I_h \times J_h)10^n + [(I_h \times J_l) + (I_l \times J_h)]10^{n/2} + (I_l \times J_l)$$

$$= \underbrace{(I_h \times J_h)}_{\text{subProblem}_1}10^n + [\underbrace{(I_h - I_l)\times(J_l - J_h)}_{\text{subProblem}_3} + \underbrace{(I_h \times J_h)}_{\text{subProblem}_1} + \underbrace{(I_l \times J_l)}_{\text{subProblem}_2}]10^{n/2} + \underbrace{(I_l \times J_l)}_{\text{subProblem}_2}$$

  - Complexity: T(n) = 3T(n/2) + cn,
    $\Rightarrow$ T(n) is $O(n^{\log_2 3})$, by the Master Theorem

  - Thus, T(n) is $O(n^{1.585})$.

Mergesort

- Mergesort is a perfect example of a successful application of the divide-and-conquer technique.
- It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor ..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- Algorithms for performing the merge and the mergesort are presented on the next two pages.

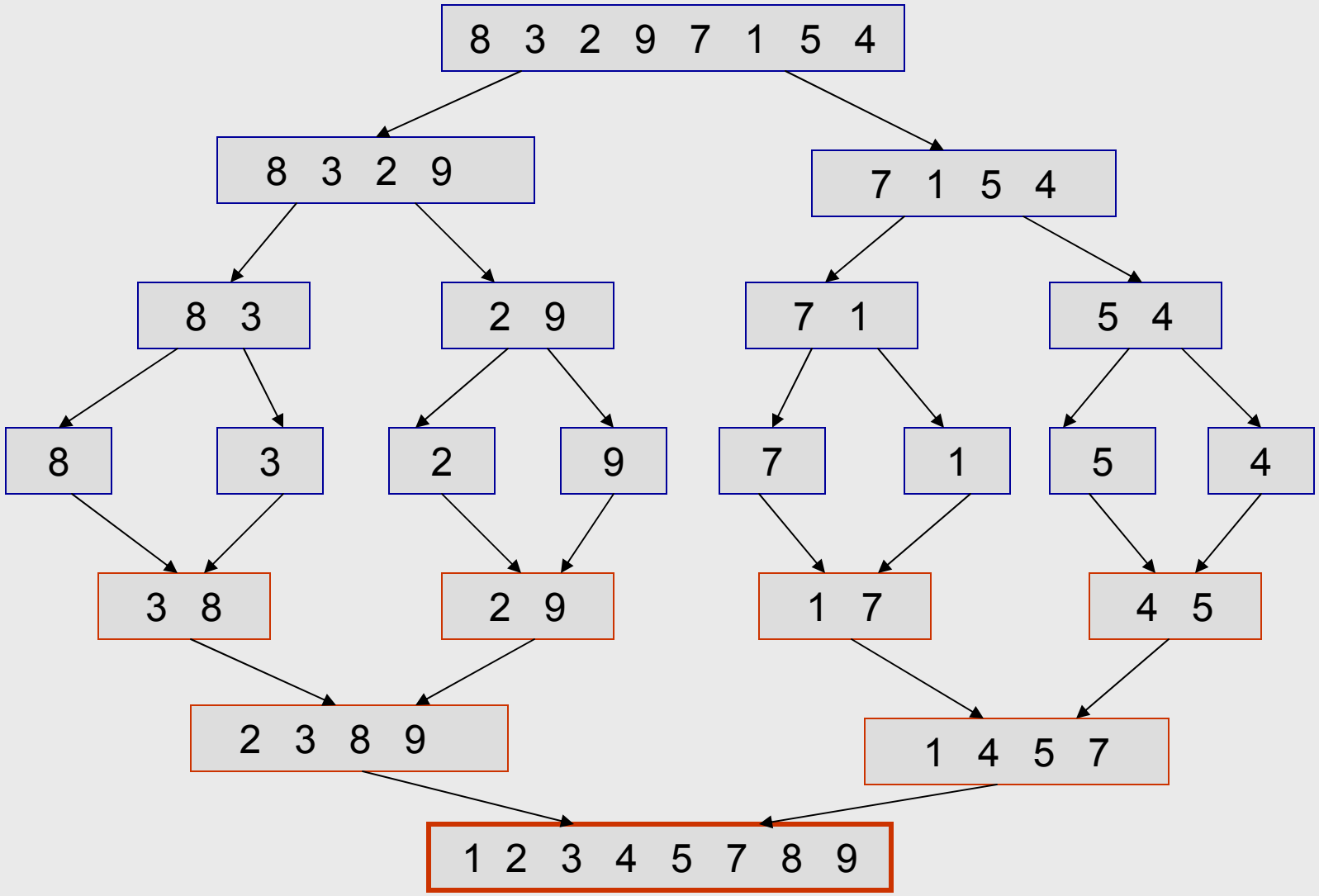
Mergesort Algorithm

```
Algorithm Mergesort( A[0..n-1])
//sorts array A[0..n-1] by recursive mergesort
//Input: An array A[0..n-1] of orderable elements
//Output: Array A[0..n-1] sorted in nondecreasing order

if n > 1
       copy A[0..⌊n/2⌋-1] to B[0..⌊n/2⌋-1]
       copy A[⌊n/2⌋..n-1] to C[0..⌊n/2⌋-1]
       Mergesort(B[0..⌊n/2⌋-1)
       Mergesort(C[0..⌊n/2⌋-1)
       Merge(B, C, A)
```


Merge Algorithm

```
Algorithm Merge( B[0..p-1], C[0..q-1], A[0..p+q-1])
//merges two sorted arrays into one sorted array
//Input: Arrays B[0..p-1] and C[0..q-1], both sorted
//Output: Sorted array A[0..p+q-1] of the elements of B and C

i ← 0;   j ←0;  k  ← 0;
while I < p and j < q do
      if B[i] ≤  C[j]
            A[k] ← B[i];  i ← i + 1;
      else  A[k] ← C[j];  j ← j + 1;
      k ← k + 1;
if i = p
      copy C[j..q-1] to A[k..p+q-1]
else copy B[i..p-1] to A[k..p+1-1]
```


An Example Mergesort Operation



Efficiency of Mergesort

- How efficient is mergesort?
- For simplicity, lets assume that *n* is a power of 2. The recurrence relation for the number of key comparisons *C*(n) is:

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1,\ C(1) = 0$$

- Now we need to determine $C_{merge}(n)$, which is the number of key comparisons performed during the merging stage (no key comparisons are performed during the splitting stage).
- At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one.

Efficiency of Mergesort (cont.)

- In the worst case, neither of the two arrays becomes empty before the other one contains just one element.
- Therefore, for the worst case, $C_{merge}(n) = n-1$, and the recurrence becomes:

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1,\ C_{worst}(1) = 0$$

- According to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$.
- In fact, it is easy to find the exact solution to the worst case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1$$

Efficiency of Mergesort (cont.)

- The number of key comparisons made by mergesort in the worst case comes very close to the theoretical minimum that any general comparison-based sorting algorithm can achieve.
  - The theoretical minimum is $\lceil \log_2 n! \rceil \cong \lceil n \log_2 n - 1.44n \rceil$
- There is, however, a shortcoming of the mergesort algorithm. Can you think what it might be?
- It is the fact that it requires a linear amount of additional memory (the arrays B and C used to hold the halves of the original array A). To overcome this problem requires merging in place, however, this algorithm is very complicated and has a significantly larger multiplicative constant making it of theoretical interest only.

Matrix Multiplication

- Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, $0 \leq i, j \leq n-1$, recall that the product AB is defined to be the $n \times n$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

- The straightforward algorithm based on this definition clearly performs n^3 (scalar) multiplications.
- In 1969 Strassen devised a divide-and-conquer algorithm for matrix multiplication of complexity $O(n^{\log_2 7})$ using certain algebraic identities for multiplying 2×2 matrices.

Matrix Multiplication (cont.)

- The classic method of multiplying 2×2 matrices performs 8 multiplications as follows:

$$AB = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{01} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{01} & a_{01}b_{01} + a_{11}b_{11} \end{bmatrix}$$

- Strassen discovered a way to carry out the same matrix product *AB* using only the following seven multiplications:

$$\begin{aligned} m_1 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11})b_{00} \\ m_3 &= a_{00}(b_{01} - b_{11}) \\ m_4 &= a_{11}(b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01})b_{11} \\ m_6 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned}$$

Matrix Multiplication (cont.)

- Using these identities, the matrix product is then given by:

$$AB = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- Now consider the case of two $n \times n$ matrices where, for convenience, we'll assume that $n = 2^k$.
- Strassen's divide and conquer approach begins by partitioning the matrices *A* and *B* into four $(n/2) \times (n/2)$ submatrices, as follows:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \qquad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Matrix Multiplication (cont.)

- The product *AB* can be expressed in terms of eight matrix products as follows:

$$AB = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{01} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{01} & A_{01}B_{01} + A_{11}B_{11} \end{bmatrix}$$

- In complete analogy with the 2×2 case, the following matrix multiplications will produce the matrix product *AB*.

$$\begin{aligned} M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ M_2 &= (A_{10} + A_{11})B_{00} \\ M_3 &= A_{00}(B_{01} - B_{11}) \\ M_4 &= A_{11}(B_{10} - B_{00}) \\ M_5 &= (A_{00} + A_{01})B_{11} \\ M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned}$$

7 multiplication operations
10 +/- operations

Matrix Multiplication (cont.)

- As in the case of the 2×2 matrices, the matrix product *AB* is then given by:

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

8 more +/- operations = total 18 +/- operations

- The complexity of Strassen's algorithm satisfies the recurrence:

$$T(n) = 7T(n/2), \; n > 1 \qquad \text{initial condition } T(1) = 1$$

- By the Master Theorem (see page 8), $T(n) \in \Theta(n^{\log_2 7})$.
- Since $\log_2 7$ is approximately 2.81, Strassen's algorithm provides a matrix multiplication algorithm with complexity $\Theta(n^{\log_2 7})$ a significant improvement over the $\Theta(n^3)$ classical algorithm.

Matrix Multiplication (cont.)

- As an aside, you may be interested to know that Winograd discovered the following set of identities, which leads to a method of multiplying 2×2 matrices using only 15 +/- operations in addition to the 7 multiplication operations.

$$m_1 = (a_{10} + a_{11} - a_{00})(b_{11} - b_{01} + b_{00})$$
$$m_2 = a_{00}b_{00}$$
$$m_3 = a_{01}b_{10}$$
$$m_4 = (a_{00} - a_{10})(b_{11} - b_{01})$$
$$m_5 = (a_{10} + a_{11})(b_{01} - b_{00})$$
$$m_6 = (a_{01} - a_{10} + a_{00} - a_{11})b_{11}$$
$$m_7 = a_{11}(b_{00} - b_{11} - b_{01} + b_{10})$$

7 multiplication operations and 24 +/- operations.

NOTE: Only 15 of the +/- operations are distinct!

- Winograd's product matrix is given by:

$$AB = \begin{bmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{bmatrix}$$

Special Note On Binary Search

- Binary search is often presented (especially in CS2-level texts) as the quintessential example of a divide-and-conquer algorithm.
- This interpretation is flawed because, in fact, binary search is a very atypical case of divide-and-conquer.
- According to the definition, the divide-and-conquer technique divides a problem into several subproblems, each of which need to be solved. This is not the case for binary search where, instead, only one of the two subproblems needs to be solved.
- Therefore, if binary search is to be considered as a divide-and-conquer algorithm, it should be looked on as a degenerative case of the technique.
- As a matter of fact, binary search fits better into the class of decrease-by-half algorithms.