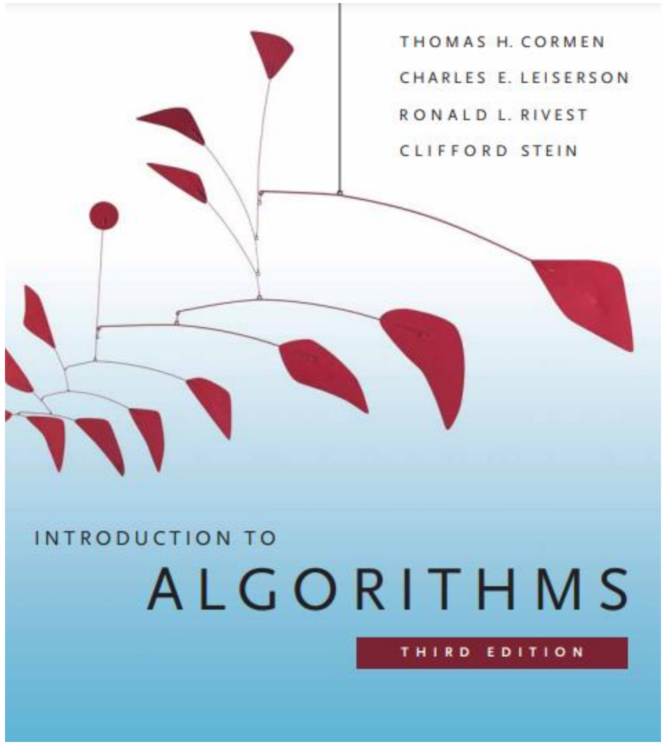
ANALYSIS OF ALGORITHMS
CPS 5440

UNIT 4: DIVIDE AND CONQUER

DIVIDE AND CONQUER

- ❑ Binary search
- ❑ Powering a number
- ❑ Fibonacci numbers
- ❑ Matrix multiplication
- ❑ Maximum subarray problem



THE DIVIDE-AND-CONQUER DESIGN PARADIGM

1. ***Divide*** the problem (instance) into subproblems that are smaller instances of the same problem.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

RECURRENCES

- ***Recurrences go hand in hand with the divide − and − conquer paradigm***
- ***A recurrence describes a function in terms of its value on smaller inputs***
- Recurrences can take many forms:
  - size of subproblems may differ. $e.g., T(n) = 1.T(2n/3) + 1.T(n/3) + \Theta(n)$
  - not necessarily constant fraction $e.g., T(n) = T(n-1) + \Theta(1)$
- ***A recurrence can be an equation or inequality***
  - $T(n) \leq T(n/2) + \Theta(n)$
  - such a recurrence states only an upper bound on $T(n)$
  - Its solution is couched using $O - notation$ rather than $\Theta - notation$
  - *similarly*, $T(n) \geq T(n/2) + \Theta(n)$ *gives a lower bound and couched using* $\Omega - notation$

RECURRENCES

- In practice, we often neglect certain technical details
- , e.g., in merge sort, we neglect the case of n is odd
- The running time on a constant-sized input is a constant
- Hence, we usually omit statements of the boundary conditions of recurrences
- We often omit floors, ceilings, and boundary conditions

MERGE SORT

1. ***Divide***: Trivial.
2. ***Conquer***: Recursively sort 2 subarrays.
3. ***Combine***: Linear-time merge.

$$\underbrace{T(n) = 2}_{\#\ \text{subproblems}}\ T\left(\underbrace{\frac{n}{2}}_{\text{subproblem size}}\right) + \underbrace{\Theta(n)}_{\text{time to combine}}$$

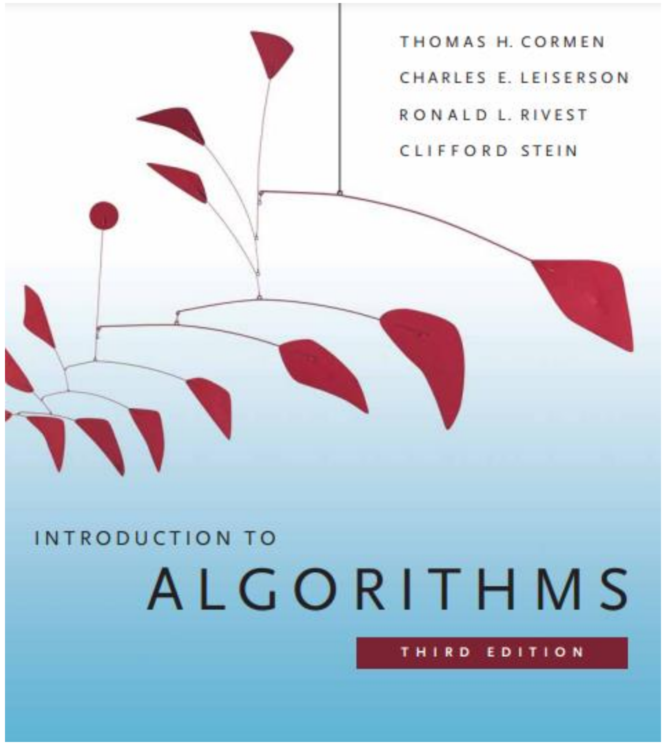
MASTER THEOREM (RECAP)

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Case 1: $f(n) = O(n^{log_b a - \varepsilon}), \varepsilon > 0 \Longrightarrow T(n) = \Theta(n^{log_b a})$
- Case 2: $f(n) = \Theta(n^{log_b a}), \Longrightarrow T(n) = \Theta(n^{log_b a} \lg n)$
- Case 3: $f(n) = \Omega\,(n^{log_b a + \varepsilon}),\ \varepsilon > 0 \wedge \text{regularity condition} \Longrightarrow T(n) = \Theta\big(f(n)\big)$

- **Merge Sort**
  - $a = 2;\ b = 2; f(n) = n;$
  - $\Longrightarrow n^{log_b a} = n^{log_2 2} = n$
  - $\Longrightarrow$ Case 2 $\Longrightarrow T(n) = \Theta(n \lg n)$

- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Maximum subarray problem



BINARY SEARCH

- Problem: Find an element in a sorted array

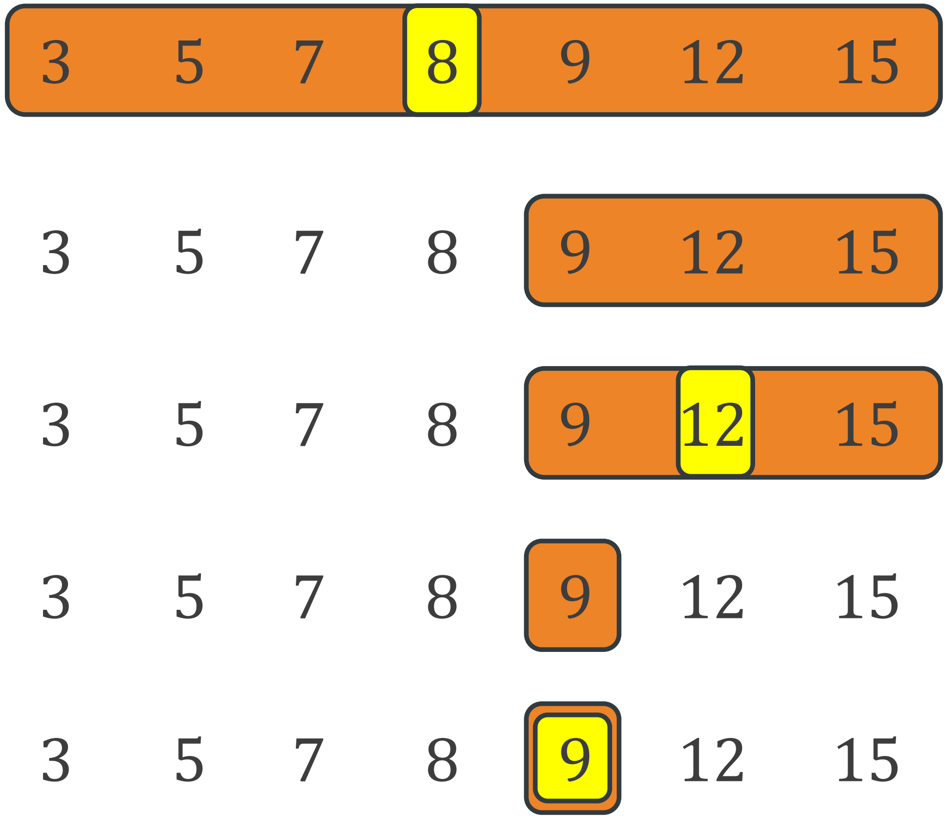
1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine*****:** Trivial.

BINARY SEARCH

- Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

- Example: Find 9



BINARY SEARCH

```
binarySearch(arr, x, low, high)
    if low > high
        return False

    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid

        else if x > arr[mid]        // x is on the right side
            return binarySearch(arr, x, mid + 1, high)

        else                        // x is on the right side
            return binarySearch(arr, x, low, mid - 1)
```

RECURRENCE BINARY SEARCH

$$T(n) = \mathbf{1}\ T\left(\frac{\boldsymbol{n}}{\mathbf{2}}\right) + \boldsymbol{\Theta}(\mathbf{1})$$

subproblems

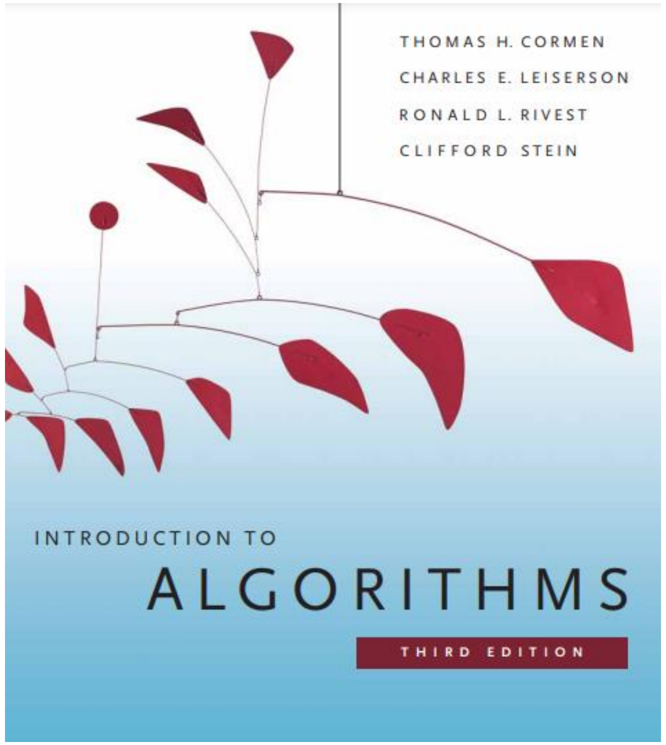
subproblem size

Time to combine

- $a\ = 1;\ b\ = 2; f(n) = n^0$
- $\Longrightarrow\ n^{log_b a} = n^{log_2 1} = n^0$
- $f(n) = n^{log_b a} \Longrightarrow$ **case 2** $[f(n) = \Theta(n^{log_b a})]$
- $\Longrightarrow T(n)\ = \Theta(n^{log_b a} \lg n) = T(n) = \Theta(\lg n)$

DIVIDE AND CONQUER

- ❑ Binary search
- ❑ Powering a number
- ❑ Fibonacci numbers
- ❑ Matrix multiplication
- ❑ Maximum subarray problem



POWERING A NUMBER

- Problem: Compute a^n, where $n \in \mathbb{N}$

1. ***Naive algorithm***: $\Theta(n)$.
2. ***Divide − and − conquer algorithm***:

$$a^n = \begin{cases} a^{n/2} . a^{n/2} & if\ n\ is\ even; \\ a^{(n-1)/2} . a^{(n-1)/2} . a & if\ n\ is\ odd. \end{cases}$$

$$T(n) = \mathbf{1} . T(n/2) + \boldsymbol{\Theta}(\mathbf{1}) \Longrightarrow T(n) = \Theta(\lg n).$$

POWERING A NUMBER

```
power(int a, int n){
    prod = 1;

    for(i=0; i<n; i++)
        prod = prod * a;

    return prod;
}
```

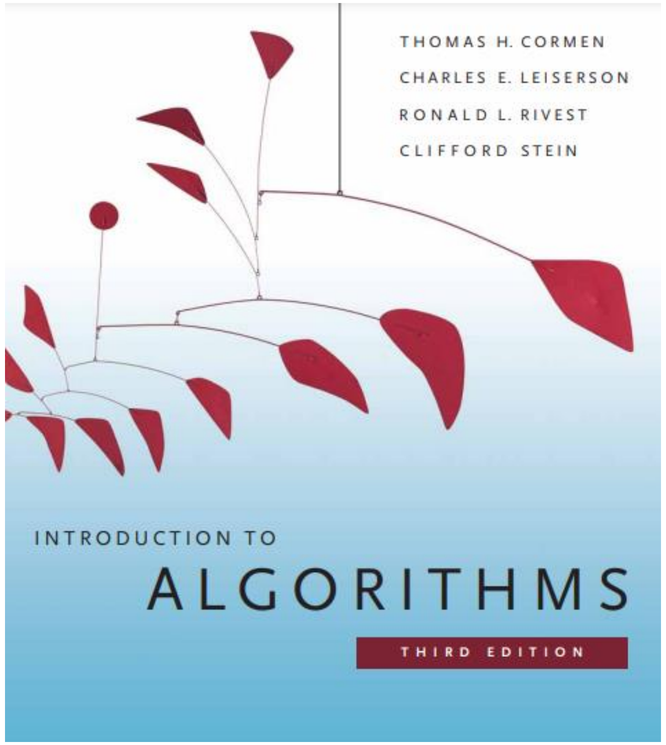
A naive iterative approach
$O(n)$ time.

```
power(int x, int y){
    int temp;
    if (y == 0)
        return 1;
    temp = power(x, y / 2);
    if (y % 2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
```

An Optimized Divide and Conquer Solution:
$\Theta(\lg n)$ time.

DIVIDE AND CONQUER

- ❑ Binary search
- ❑ Powering a number
- ❑ Fibonacci numbers
- ❑ Matrix multiplication
- ❑ Maximum subarray problem



- Recursive Definition:

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$

1. ***Naive recursive algorithm***:
   - $\Omega(\phi^n)$. (exponential time),
   - where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803398875$
   - = the golden ratio.

```
fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

2. Bottom-up:

- **Compute** $F_0, F_1, F_2, \cdots, F_n$ **in order,**
- ***forming each number by summing the two previous***
- *Running Time*: $\Theta(n)$

```
fib(int n) {
    int a = 0, b = 1, c;
    if (n == 0)    return a;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

3. ***Naïve Recursive Squaring:***
   - $F_{\boldsymbol{n}} = \phi^n / \sqrt{5}$ ***rounded to the nearest integer.***
   - $Running\ Time: \Theta(\lg n)$ ***time.***
   - ***This method is unreliable,***
   - ***since floating-point arithmetic is prone to round-off errors.***

4. Algorithm: Recursive Squaring .

- *Time* $= \Theta(\lg n)$ *time*

- *Theorem:* $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$.
- *Proof of theorem. (Induction on n)*
- *Base* ($n = 1$)*:* $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$

- 0 1 1 2 3 5 8 13 21 34 ⋯
- F_0 F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 ⋯

RECURSIVE SQUARING

- ***Theorem:*** $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$

- ***Inductive step*** $(n \geq 2)$

- $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} . \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$$= \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} . \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

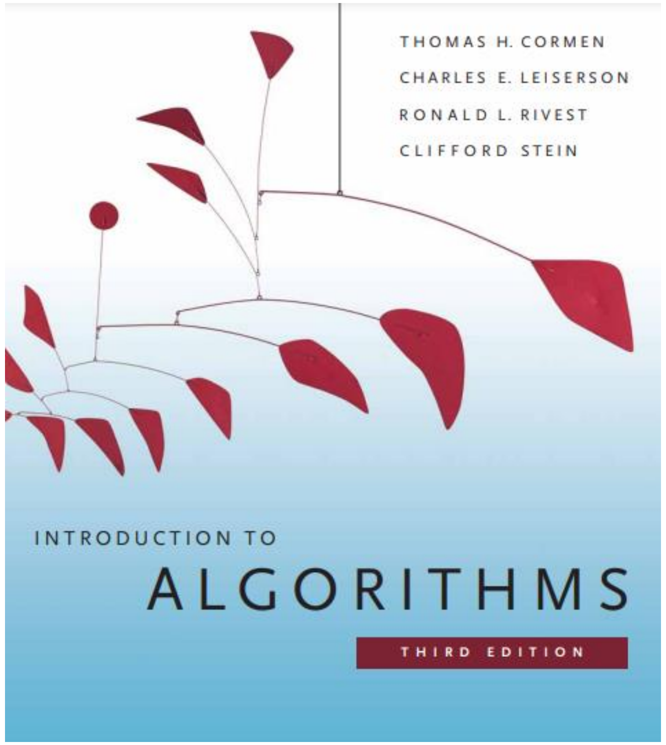
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} . \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n}$$

✓ $\begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}.$

DIVIDE AND CONQUER

- ❑ Binary search
- ❑ Powering a number
- ❑ Fibonacci numbers
- ❑ Matrix multiplication
- ❑ Maximum subarray problem



MATRIX MULTIPLICATION

$$\left.\begin{array}{l}\textbf{\textit{Input}}\text{: } A = [a_{ij}], B = [b_{ij}].\\ \textbf{\textit{Output}}\text{: } C = [c_{ij}] = A.B.\end{array}\right\} i, j = 1, 2, \dots, n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{12} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{12} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} . \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{12} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

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

1. *Standard Algorithm:*

- We must compute n^2 *matrix entries, and each is the sum of* n *values*

SQUARE−MATRIX−MULTIPLY ***(A,B)***

$n = A.rows$

$let\ C\ be\ a\ new\ n \times n\ matrix$

$\boldsymbol{for}\ i = 1\ to\ n$

$\quad \boldsymbol{for}\ j = 1\ to\ n$

$\quad\quad c_{ij} = 0$

$\quad\quad \boldsymbol{for}\ k = 1\ to\ n$

$\quad\quad\quad c_{ij} = c_{ij} + a_{ik}.\, b_{kj}$

$\boldsymbol{return}\ C$

Running Time: $\Theta(n^3)$

- You might at first think that any matrix multiplication algorithm must take $\Omega(n^3)$ time, since the natural definition of matrix multiplication requires that many multiplications.
- You would be incorrect, however: we have a way to multiply matrices in $o(n^3)$.

2. A simple divide-and-conquer algorithm

- *Assume* n *is an exact power of 2 in each of the* $n \times n$ *matrices*
- $\boldsymbol{Idea}: \boldsymbol{n \times n\ matrix = 2 \times 2\ matrix\ of\ (n/2) \times (n/2) submatrices}$:

- $\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$

$$\boldsymbol{C \quad = \quad A \quad . \quad B}$$

recursive

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} = \begin{aligned} &8\ mults\ of\ \boldsymbol{(n/2) \times (n/2) submatrices} \\ &\boldsymbol{4\ adds\ of\ (n/2) \times (n/2) submatrices} \end{aligned}$$

2. A simple divide-and-conquer algorithm

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} . \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C \quad = \quad A \quad . \quad B$$

- **Implementation:**
  - Partitioning does involve creating $12\ new (n/2) \times (n/2)$ matrices
  - Otherwise, we would spend $\Theta(n^2)$ copying entries
  - Instead, we use index calculations
  - We identify a matrix, by a range of row and column indices of the original matrix

MATRIX MULTIPLICATION

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

MATRIX MULTIPLICATION

$$T(n) = \mathbf{8}\, T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n^2)$$

submatrices

submatrix size

work adding submatrices

- $a = 8;\ b = 2; f(n) = n^2$
- $\Longrightarrow\ n^{log_b a} = n^{log_2 8} = n^3$
- Case 1: $f(n) = O(n^{log_b a - \varepsilon}), \varepsilon > 0 \Longrightarrow T(n) = \Theta(n^{log_b a})$
- $\Longrightarrow T(n) = \Theta(n^3)$
- No better than the ordinary algorithm.

3. *Strassen's Idea*

- *Make the recursion tree slightly less bushy*
- *Multiply* 2×2 *matrix with only* 7 *recursive mults instead of 8.*
- *The cost of eliminating one mult will be several new additions of* $(n/2) \times (n/2)$ *matrices*

Strassen's Algorithm (1969)

1. Partition each of the matrices into four $n/2 \times n/2$ submatrices
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$
4. Compute $n/2 \times n/2$ submatrices of C by adding and subtracting various combinations of the P_i.

3. *Strassen's Idea*

- *Make the recursion tree slightly less bushy*
- *Multiply* 2×2 *matrix with only* 7 *recursive mults instead of 8.*
- *The cost of eliminating one mult will be several new additions of* $(n/2) \times (n/2)$ *matrices*

- $P_1 = a.(f - h)$
- $P_2 = (a + b).h$
- $P_3 = (c + d).e$
- $P_4 = d.(g - e)$
- $P_5 = (a + d).(e + h)$
- $P_6 = (b - d).(g + h)$
- $P_7 = (a - c).(e + f)$

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

$7\ mults, 18\ adds/subs$

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} . \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$
$$C = A . B$$

3. *Strassen's Idea*

- $P_1 = a.(f - h)$
- $P_2 = (a + b).h$
- $P_3 = (c + d).e$
- $P_4 = d.(g - e)$
- $P_5 = (a + d).(e + h)$
- $P_6 = (b - d).(g + h)$
- $P_7 = (a - c).(e + f)$

$$\left.\begin{aligned} r &= ae + bg \\ \checkmark\; s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\}$$

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= [(a + d).(e + h)] + \\ &\quad [d.(g - e)] - \\ &\quad [(a + b).h] + \\ &\quad [(b - d).(g + h)] \\ &= ae + ah + de + dh \\ &\quad + dg - de \\ &\quad - ah - bh \\ &\quad + bg + bh - dg - dh \\ &= ae + bg \end{aligned}$$

STRASSEN'S ALGORITHM

1. ***Divide***:
   - Partition A and B into $(\boldsymbol{n/2}) \times (\boldsymbol{n/2})$ submatrices.
   - Form term to be multiplied using $+$ and $-$.
2. ***Conquer***: Perform 7 multiplications of $(\boldsymbol{n/2}) \times (\boldsymbol{n/2})$ submatrices recursively.
3. ***Combine***: Form C using $+$ and $-$ on $(\boldsymbol{n/2}) \times (\boldsymbol{n/2})$ submatrices.

$$T(n) = \mathbf{7}\, T\left(\frac{\boldsymbol{n}}{\mathbf{2}}\right) + \boldsymbol{\Theta}(\boldsymbol{n}^{\mathbf{2}})$$

$T(n) = \mathbf{7}\, T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(\boldsymbol{n^2})$

- $a = 7;\ b = 2; f(n) = n^2$
- $\Longrightarrow\ n^{log_b a} = n^{log_2 7} = n^{2.81}$
- Case 1: $f(n) = O(n^{log_b a - \varepsilon}), \varepsilon > 0 \Longrightarrow T(n) = \Theta(n^{log_b a})$
- $\Longrightarrow T(n) = \Theta(n^{2.81})$

➢ *The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant.*

➢ *In fact, Strassen's algorithm beats the ordinary algorithm (i.e,* $\Theta(n^3)$*) on today's machines for* $n \geq 32$ *or so.*

➢ ***Best to date (of theoretical interest only):*** $\Theta(n^{2.376\ldots})$

CURRENT STATE-OF-THE-ART

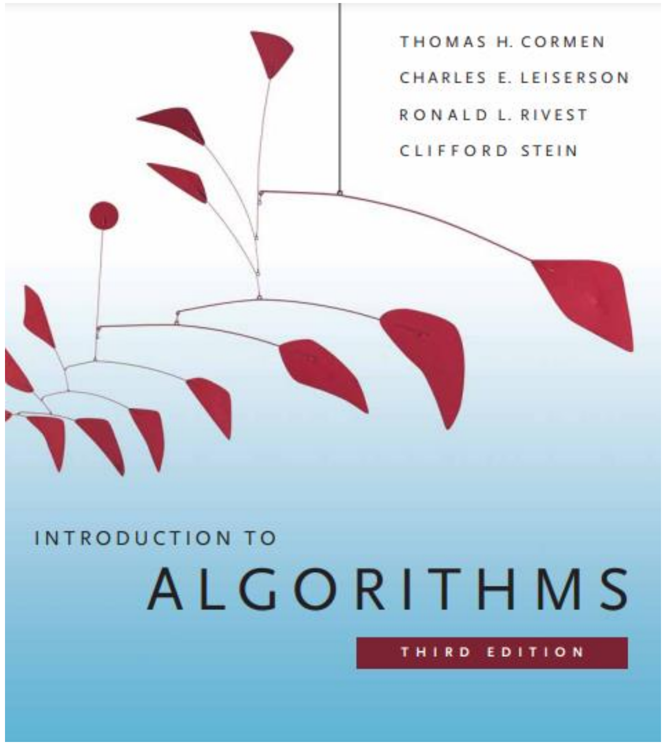
Asymptotic Complexities:

- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
- $O(n^{2.796})$, Pan (1978)
- $O(n^{2.522})$, Schönhage (1981)
- $O(n^{2.517})$, Romani (1982)
- $O(n^{2.496})$, Coppersmith and Winograd (1982)
- $O(n^{2.479})$, Strassen (1986)
- $O(n^{2.376})$, Coppersmith and Winograd (1989)
- $O(n^{2.374})$, Stothers (2010)
- $O(n^{2.3728642})$, V. Williams (2011)
- $O(n^{2.3728639})$, Le Gall (2014)
- ...

Strassen’s Matrix Multiplication

DIVIDE AND CONQUER

- ❑ Binary search
- ❑ Powering a number
- ❑ Fibonacci numbers
- ❑ Matrix multiplication
    - ❑ Maximum subarray problem



Problem statement:

- Input: an array $A[1 \cdots n]$ of (positive /negative) numbers
- Output:
    - (1) indices $i\ and\ j$ such that the subarray $A[i \cdots j]$ has the greatest sum of any nonempty contiguous subarray of A, and
    - (2) the sum of the values in $A[i \cdots j]$

- Note: maximum subarray might not be unique, though its value is, so we speak of a maximum rather than the maximum subarray problem
- If all the array entries were nonnegative, the entire array would give the greatest sum!

THE MAXIMUM-SUBARRAY PROBLEM

Example 1:

$$A[1 \cdots 4] = \begin{array}{|c|c|c|c|} \hline 1 & -4 & 3 & -4 \\ \hline \end{array}$$

Maximum-subarray: $A[3 \cdots 3] (i = \mathrm{j} = 3)$, and sum $= 3$

Example 2:

$$A[1 \cdots 6] = \begin{array}{|c|c|c|c|c|c|} \hline 1 & -4 & 3 & 4 & -2 & 6 \\ \hline \end{array}$$

Maximum-subarray: $A[3 \cdots 6] (i = 3, \mathrm{j} = 6)$, and sum $= 11$

Example 2:

$$A[1 \cdots 16] = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 13 & -3 & -25 & 20 & -3 & -16 & -23 & 18 & 20 & -7 & 12 & -5 & -22 & 15 & -4 & 7 \\ \hline \end{array}$$

Maximum-subarray: $A[8 \cdots 11] (i = 8, \mathrm{j} = 11)$, and sum $= 43$

THE MAXIMUM-SUBARRAY PROBLEM

$\boldsymbol{Algoritm1: brute - force}$

Idea: check all subarrays

Total number of subarrays $A[i \cdots j]$:

- $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{1}{2}n(n-1) = \Theta(n^2)$
- plus the arrays of length $= 1$

- Cost $T(n) = \Theta(n^2)$

THE MAXIMUM-SUBARRAY PROBLEM

Algoritm2: Divide and Conquer

- Generic problem: Find a maximum subarray of $A[\text{low} \cdots high]$ with initial call:

$$low = 1, and\ high = n$$

- DC strategy:
  - *Divide*: $A[\text{low} \cdots high]$ into two subarrays of as equal size as possible by finding the midpoint mid
  - Conquer:
    - (a) finding maximum subarrays of $A[\text{low} \cdots mid]$ and $A[mid + 1 \cdots \text{high}]$
    - (b) finding a max-subarray that crosses the midpoint
  - Combine: returning the max of the three
- Correctness: this strategy works because any subarray must either lie entirely in one side of midpoint or cross the midpoint.

Algoritm2: Divide and Conquer

- $A[\text{low} \cdots high]$
- Divide in the middle: $A[\text{low} \cdots mid], A[mid + 1 \cdots \text{high}]$
- Any subarray $A[i \cdots j]$ is

  (1) entirely in $A[\text{low} \cdots mid]$ so that $low \le i \le j \le mid,$

  (2) entirely in $A[mid + 1 \cdots \text{high}]$ so that $mid < i \le j \le high$

  (3) in both so that $low \le i \le mid < j \le high$
- (1) and (2) can be found recursively
- (3) need to find max-subarray of $A[i \cdots mid], A[mid + 1 \cdots j]$
- Take subarray with largest sum of $(1), (2)$, and (3)

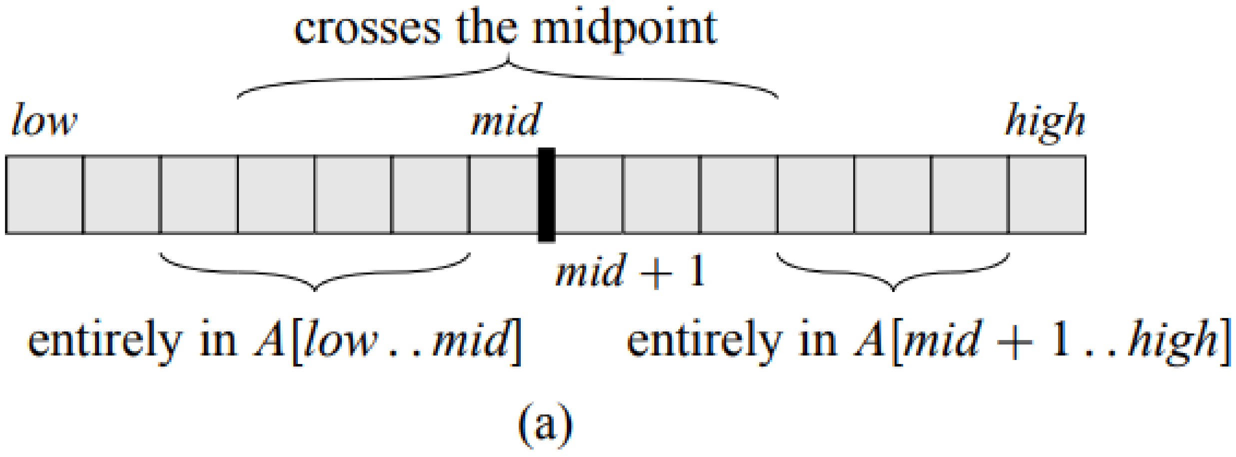
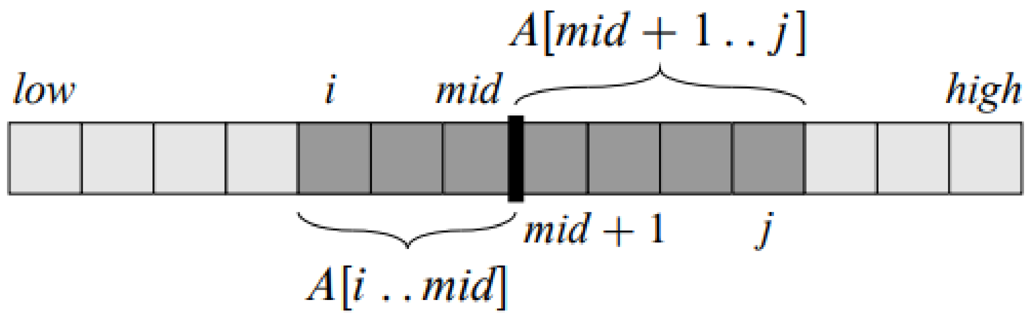


Figure 4.4 **(a)** Possible locations of subarrays of $A[low \,..\, high]$: entirely in $A[low \,..\, mid]$, entirely in $A[mid + 1 \,..\, high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low \,..\, high]$ crossing the midpoint comprises two subarrays $A[i \,..\, mid]$ and $A[mid + 1 \,..\, j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

THE MAXIMUM-SUBARRAY PROBLEM

$FindMaxCrossSubarray(A, low, mid, high)$

$left\text{-}sum = -\infty$
$sum = 0$
$\textbf{for } i = mid \textbf{ downto } low$
$\quad sum = sum + A[i]$
$\quad \textbf{if } sum > left\text{-}sum \textbf{ then}$
$\quad\quad left\text{-}sum = sum$
$\quad\quad \text{max-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
$\textbf{for } j = mid + 1 \textbf{ to } high$
$\quad sum = sum + A[j]$
$\quad \textbf{if } sum > right\text{-}sum \textbf{ then}$
$\quad\quad right\text{-}sum = sum$
$\quad\quad \text{max-}right = j$
$\textbf{return } (\text{max-}left, \text{max-}right, left\text{-}sum + right\text{-}sum)$



THE MAXIMUM-SUBARRAY PROBLEM

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Remarks

- *Initial call:* $MaxSubarray(A, 1, n)$
- *Base case is when the subarray has only 1 element.*
- *Divide* by computing mid
- Conquer by the two recursive calls to $MaxSubarray$ and a call to $FindMaxCrossSubarray$
- Combine by determining which of the three results gives the maximum sum.
- Complexity:
  - $T(n) = \ 2.T\left(\frac{n}{2}\right) + \ \Theta(n) + \Theta(1)$
  - $\qquad = \Theta(n \log n)$

Algoritm2: Divide and Conquer

- *Find-Max-Cross-Subarray:* $O(n)$ *time*
- *Two recursive calls on input size* $n/2$
- *Thus:*

$$T(n) = 2T(n/2) + O(n)$$
$$T(n) = O(n \log n)$$

Maximum Subarray Sum

Divide & Conquer strikes back: maximum-subarray

Conclusions

- Divide and conquer is just one of several powerful techniques for algorithm design
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math.)
- The divide-and-conquer strategy often leads to efficient algorithms

QUESTIONS/ANSWERS