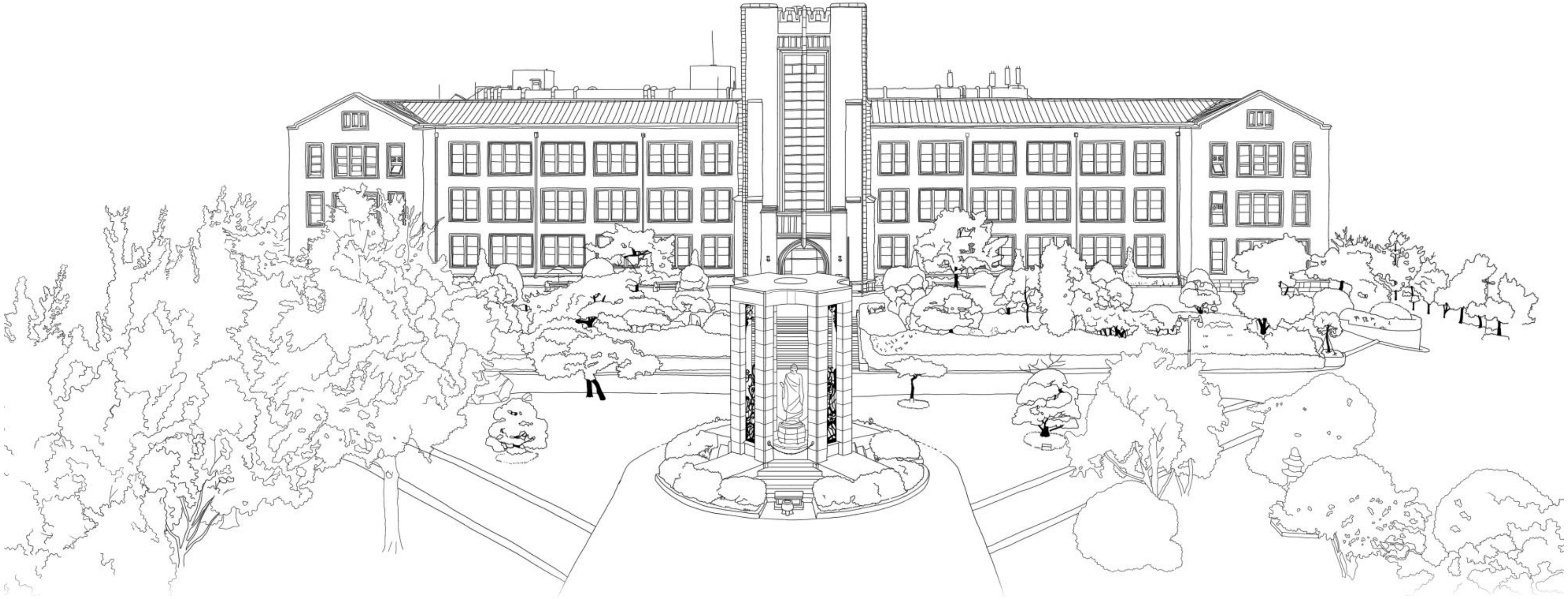


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

Contents

❖ Chapter.1 Foundations

1. The Role of Algorithms in Computing
   1) Algorithms
   2) Algorithms as a technology

2. Getting Started
   1) Insertion sort
   2) Analyzing algorithms
   3) Designing algorithms

3. Growth of Functions
   1) Asymptotic notation
   2) Standard notations and common functions

4. Divide-and-Conquer
   1) The maximum-subarray problem
   2) Strassen's algorithm for matrix multiplication
   3) The substitution method for solving recurrences
   4) The recursion-tree method for solving recurrences
   5) Proof of the master theorem

5. Probabilistic Analysis and Randomized Algorithms
   1) The hiring problem
   2) Indicator random variables
   3) Randomized algorithms
   4) Probabilistic analysis and further uses of indicator random variables

❖ Introduction

- Order of growth
  - ➢ **The order of growth of the running time of an algorithm**
    - ✓ Gives a simple characterization of the algorithm's efficiency
    - ✓ For large enough inputs, the multiplicative constants and lower-order terms are dominated by the effects of the input size itself

      Ex) $an^2 + bn + c \rightarrow \Theta(n^2)$
    - ✓ Enables to compare the relative performance of alternative algorithms

      Ex) Merge sort : $\Theta(\mathrm{n}\log n) <$ Insertion sort : $\Theta(n^2)$

  - ➢ **Asymptotic efficiency of algorithm**
    - ✓ How the running time of an algorithm increases with the size of the input in the limit
    - ✓ Several standard methods for simplifying the asymptotic analysis of algorithms

❖ Asymptotic notation

- **Asymptotic notation, functions, and running times**
  - ➢ Asymptotic notation
    - ✓ Describe the running times of algorithms
    - ✓ Applies to functions

      Ex) What we were writing as $\Theta(n^2)$ was the function about $an^2 + bn + c$

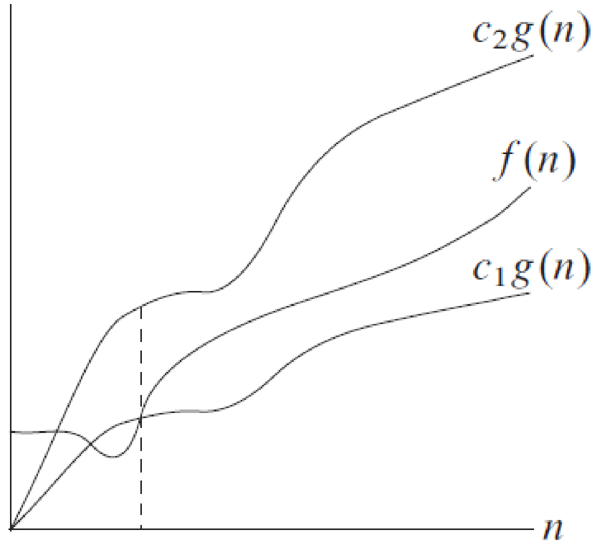
  - ➢ Functions
    - ✓ The functions to which we apply asymptotic notation will usually characterize the running times of algorithms
    - ✓ Characterize other aspect of algorithms – amount of space they use

  - ➢ Running time
    - ✓ Commonly interested in the worst case
    - ✓ Characterize the running time no matter what the input

❖ Asymptotic notation

- **Θ** notation (Big-**Θ** notation)
  - ➢ Definition

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .^1$$



$f(n) = \Theta(g(n))$

- ✓ $\Theta(g(n))$ is the set of functions
- ✓ Function $f(n)$ is running time of algorithm
- ✓ n_0 is the minimum possible value
- ✓ $f(n) = \Theta(g(n))$ means $f(n)$ belongs to the set $\Theta(g(n))$
  ($f(n) \in \Theta(g(n))$)
- ✓ For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$
- ✓ $g(n)$ is asymptotically tight bound for $f(n)$

❖ Asymptotic notation

- $\boldsymbol{\Theta}$ notation
  - Example : Prove $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
    - ✓ Let $f(n) = \frac{1}{2}n^2 - 3n,\ g(n) = \theta(n^2)$
    - ✓ According to definition

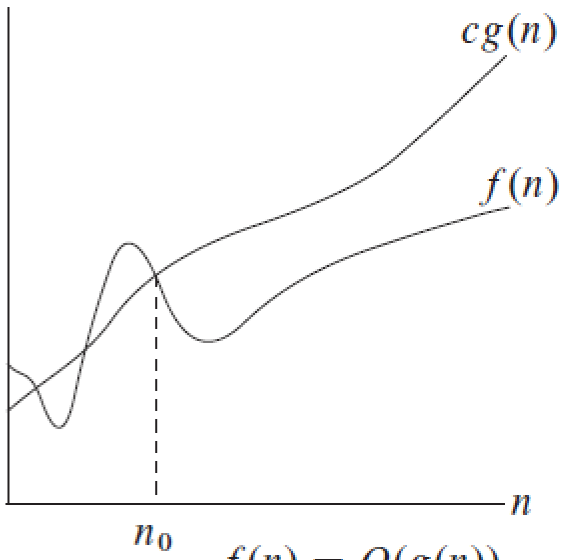
      $c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$
    - ✓ For all $n \ge n_0$, dividing by n^2

      $c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$
    - ✓ First, if $c_1 \le \frac{1}{14}$ & $n \ge 7 \rightarrow c_1 \le \frac{1}{2} - \frac{3}{n}$ is always true
    - ✓ Second, if $c_2 \ge \frac{1}{2}$ & $n \ge 1 \rightarrow c_2 \ge \frac{1}{2} - \frac{3}{n}$ is always true
    - ✓ So there exist c_1, c_2, n_0 so we can verify that $\frac{1}{2}n^2 - 3n = \theta(n^2)$
    - ✓ The important thing is that there exist some set about c_1, c_2, n_0, not which number of c_1, c_2, n_0 choosen

Asymptotic notation

- O notation (Big-O notation)
  - Definition

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$



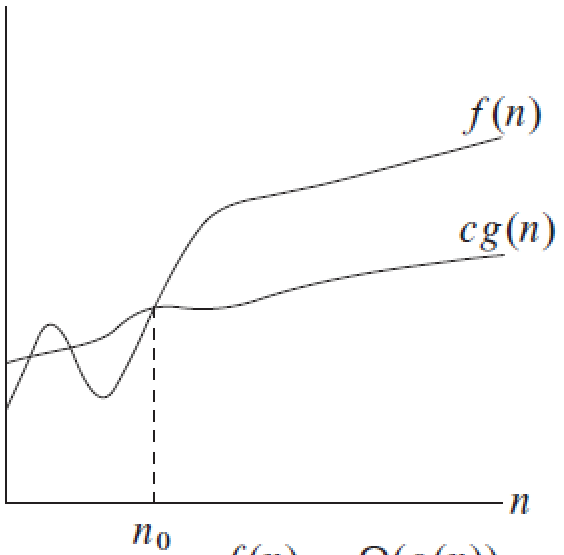
$f(n) = O(g(n))$

- ✓ Only have an asymptotic upper bound
- ✓ $f(n) = O(g(n))$ means $f(n)$ belongs to the set $O(g(n))$
  ($f(n) \in O(g(n))$)
- ✓ $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$
  ($\Theta(g(n)) \subseteq O(g(n))$)
- ✓ For all values of n to the right of n_0, the value of $f(n)$ lies at or below $cg(n)$
- ✓ $g(n)$ is an asymptotic upper bound on $f(n)$

❖ Asymptotic notation

- Ω notation (Big−Ω notation)
  - ➢ Definition

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$f(n) = \Omega(g(n))$

✓ Only have an asymptotic lower bound

✓ $f(n) = \Omega(g(n))$ means $f(n)$ belongs to the set $\Omega(g(n))$ ($f(n) \in \Omega(g(n))$)

✓ For all values of n to the right of n_0, the value of $f(n)$ lies at or above $cg(n)$ and

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Asymptotic notation

- o notation (Little-o notation)
  - Definition

$$o(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \} .$$

- ✓ Denote an upper bound that is not asymptotically tight (Relaxed upper bound)
- ✓ Big-O notation, the bound $0 \le f(n) \le cg(n)$ holds for some constant $c > 0$
- ✓ Little-o notation, the bound $0 \le f(n) < cg(n)$ holds for all constants $c > 0$
- ✓ Ex) $2n = o(n^2),\ but\ 2n^2 \ne o(n^2)$
- ✓ o-notation, function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

❖ Asymptotic notation

- $\boldsymbol{\omega}$ notation (Little–$\boldsymbol{\omega}$ notation)
  - ➢ Definition

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\}.$$

- ✓ Denote a lower bound that is not asymptotically tight (Relaxed lower bound)
- ✓ Big–Ω notation, the bound $0 \le cg(n) \le f(n)$ holds for some constant $c > 0$
- ✓ Little–ω notation, the bound $0 \le cg(n) < f(n)$holds for all constants $c > 0$
- ✓ Ex) $\frac{n^2}{2} = \omega(n),\ but\ \frac{n^2}{2} \ne \omega(n^2)$
- ✓ o–notation, function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

❖ Introduction

- Divide and Conquer
  - ➢ Solve a problem recursively, applying three steps
    - ✓ Divide the problem into some subproblems that are smaller instances of the same problem
    - ✓ Conquer the subproblems by solving them recursively.
      1. If the subproblem sizes are small enough, just solve the subproblems in a straightforward manner
      2. Recursive case: subproblems are large enough to solve recursively
      3. Base case: subproblems become small enough that no longer recurse
    - ✓ Combine the solutions to the subproblems into the original problem
  - ➢ Sometimes, we solve subproblems that are not the same as the original problem
    - ✓ consider solving such subproblems as combine step
  - ➢ We will solve two Divide and Conquer problem
    - ✓ maximum-subarray problem
    - ✓ Multiplying N x N matrices

Divide and Conquer

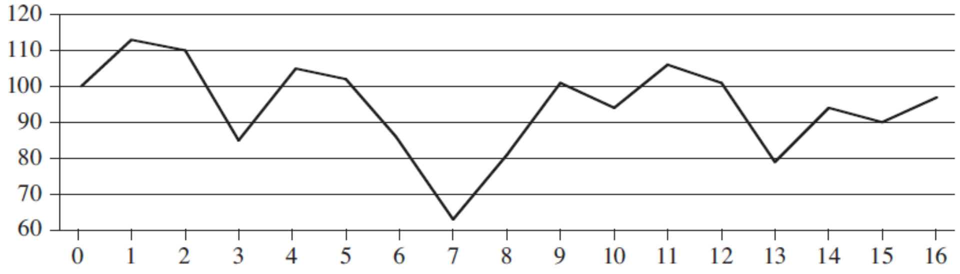
❖ The maximum subarray problem

- Definition
  - ➢ Task to find the series of continuous elements with the maximum sum in any given array
  - ➢ It is trivial if all elements in the array are non-negative

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

- Price of stock problem



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- ✓ X axis: date Y axis: stock price
- ✓ The bottom row of the table gives the change in price from the previous day
- ✓ Have to find maximum profit → find maximum subarray in bottom row of the table

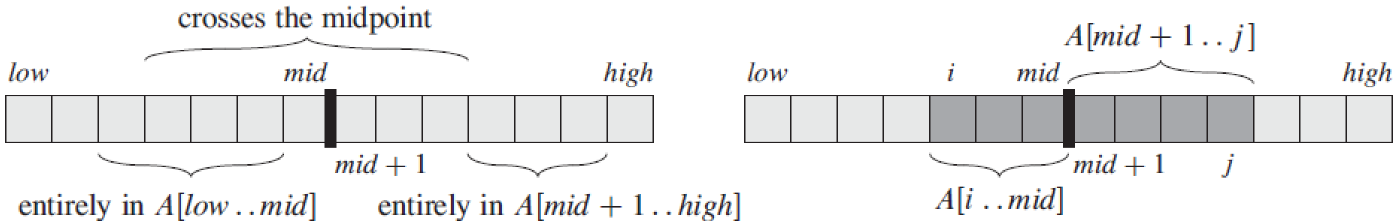
❖ The maximum subarray problem

- Brute Force solution
  - ➢ Just try every possible pair of buy and sell dates.
  - ➢ $\binom{n}{2} = \Theta(n^2)$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

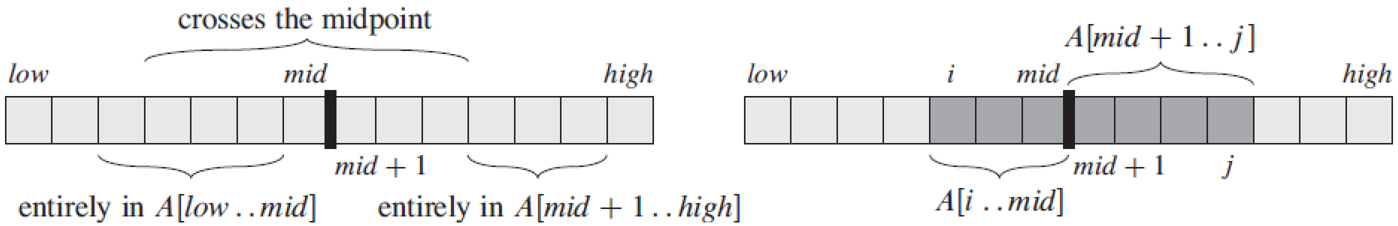
- Divide and Conquer solution



  - ➢ We want to find maximum subarray of the subarray $\boldsymbol{A[low..high]}$
  - ➢ Maximum subarray : $\boldsymbol{A[i..j]}$
  - ➢ We will find maximum subarray in three cases

❖ The maximum subarray problem

- Divide and Conquer Solution



➢ We will find maximum subarray in three cases

- entirely in the subarray $A[low..mid]$, so that $low \leq i \leq j \leq mid$,
- entirely in the subarray $A[mid+1..high]$, so that $mid < i \leq j \leq high$, or
- crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.

✓ Any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places

✓ We can find maximum subarrays of $A[low..mid]$ and $A[mid+1..high]$ recursively

i. Two subproblems are smaller instances of the Original problem

✓ all that left to do is find a maximum subarray that crosses the midpoint

❖ The maximum subarray problem

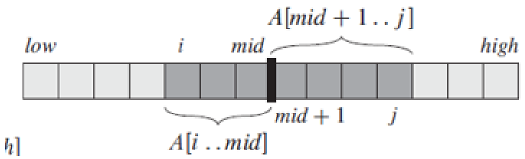
- Divide and Conquer Solution
  - ➢ Find a maximum subarray that crosses the midpoint

FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$

```
1   left-sum = -∞
2   sum = 0
3   for i = mid downto low
4       sum = sum + A[i]
5       if sum > left-sum
6           left-sum = sum
7           max-left = i
8   right-sum = -∞
9   sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

✓ This problem is not a smaller instance of original problem → it has the added restriction that the subarray must cross the midpoint

✓ Find maximum subarrays of the form $A[i..mid]$ and $A[mid+1..j]$



✓ Line 1 ~ 7 : get max−left

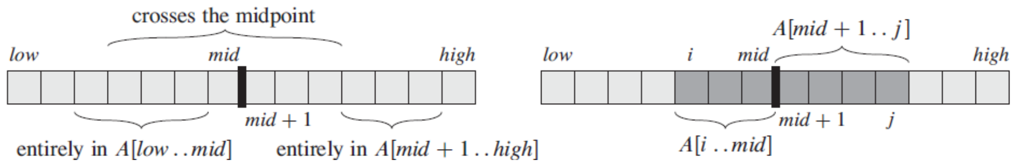
✓ Line 8 ~ 14 get max−right

✓ The total number of iteration : $high - low + 1 = n$

✓ It takes $\Theta(n)$

❖ The maximum subarray problem

- Divide and Conquer Solution
  - ➢ FIND–MAXIMUM–SUBARRAY$(\boldsymbol{A, low, high})$




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

- ✓ Line 4~5: recursively solve the problem (left & right)
- ✓ Line 6: find max crossing subarray
- ✓ Line 7~11: fine maximum subarray out of the three cases

❖ The maximum subarray problem

- Analyzing Divide and Conquer algorithm
  - ➢ Set up a recurrence of the recursive FIND-MAXIMUM-SUBARRAY procedure
    - ✓ $T(n)$: the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements
    - ✓ When $n = 1, T(1) = \theta(1)$
    - ✓ When $\mathrm{n} > 1$
      i. Recursively solve left & right subarray spend $T(n/2)$ time solving each of them
      ii. As we already seen, FIND-MAX-CROSSING-SUBARRAY takes $\theta(n)$ time
      iii. The left line take only $\theta(1)$ time
    - ✓ Therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned} \Rightarrow T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

    - ✓ According to Master method $T(n) = \theta(n \log n) \rightarrow$ asymptotically faster than the brute-force

❖ Strassen's algorithm for matrix multiplication

- Matrix multiplication
  - ➢ Definition
    - ✓ Input: $A = (a_{ij})$ and B $= (b_{ij})$ are square $n \times n$ matrices
    - ✓ Output: $C = A \times B$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad \text{for } i, j = 1, 2, \ldots, n, \text{ by}$$

  - ➢ pseudocode

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

✓ The running time for matrix multiplication

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} c = cn^3 = \theta(n^3)$$

✓ We will study about simple divide-and-conquer algorithm and Strassen's algorithm which runs in $O(n^{2.81})$ time

❖ Strassen's algorithm for matrix multiplication

- A simple divide–and–conquer algorithm
  - ➢ Divide
    - ✓ Divide $n \times n$ matrices into four $n/2 \times n/2$ matrices
    - ✓ Suppose that we partition each of A, B and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

    - ✓ We can rewrite the equation $C = A \times B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

  - ➢ Conquer & Combine
    - ✓ Perform 8 multiplication and 4 addition of $n/2 \times n/2$ submatrices recursively

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, & C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, & C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

❖ Strassen's algorithm for matrix multiplication

- A simple divide-and-conquer algorithm
  - ➢ Pseudocode & analyzing

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

1 $n = A.rows$
2 let C be a new $n \times n$ matrix
3 **if** $n == 1$
4 $\quad c_{11} = a_{11} \cdot b_{11}$
5 **else** partition A, B, and C as in equations (4.9)
6 $\quad C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11}) $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
7 $\quad C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12}) $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
8 $\quad C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11}) $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
9 $\quad C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12}) $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
10 **return** C

✓ Line5: do not create 12 new $n/2 \times n/2$ matrices
→ would spend $\Theta(n^2)$ time to copy entries
→ use index calculation

✓ Index Calculation : identify submatrix by a range of row & column indices of the original matrix
→ spend $\Theta(1)$ time

✓ When $n = 1$, $T(1) = \Theta(1)$

✓ When $n > 1$,
→ call recursive function 8 times takes $8T\left(\frac{n}{2}\right)$
→ four matrix additions takes $\Theta(n^2)$ times

❖ Strassen's algorithm for matrix multiplication

- A simple divide-and-conquer algorithm
  - ➢ Pseudocode & analyzing

$$\begin{aligned} T(n) &= \underbrace{\Theta(1)}_{\text{Index Calculation}} + \underbrace{8T(n/2)}_{\text{8 Multiplications}} + \underbrace{\Theta(n^2)}_{\text{4 Additions}} \\ &= 8T(n/2) + \Theta(n^2)\,. \end{aligned}$$

$$\Rightarrow \quad T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

- ✓ According to master method $T(n) = \theta(n^3)$
- ✓ This simple divide- and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure
- ✓ So now introduce Strassen's method

❖ Strassen's algorithm for matrix multiplication

- Strassen's method
  - ➢ Differences from the previous method
    - ✓ The key to Strassen's method is to make the recursion tree less complicated
    - ✓ It performs only seven recursive multiplications (not eight)
  - ➢ How does it work?
    - ✓ Divide the input matrices A, B and C into $n/2 \times n/2$ submatrices
      i. takes $\Theta(1)$ time by index calculation
    - ✓ Create 10 matrices s_1, s_2, .. , s_{10}, each of which is $n/2 \times n/2$ submatrices
      i. Matrices are the sum or difference of two matrices created in step 1
      ii. Takes $\Theta(n^2)$ times

$$
\begin{aligned}
S_1 &= B_{12} - B_{22}, & S_6 &= B_{11} + B_{22}, \\
S_2 &= A_{11} + A_{12}, & S_7 &= A_{12} - A_{22}, \\
S_3 &= A_{21} + A_{22}, & S_8 &= B_{21} + B_{22}, \\
S_4 &= B_{21} - B_{11}, & S_9 &= A_{11} - A_{21}, \\
S_5 &= A_{11} + A_{22}, & S_{10} &= B_{11} + B_{12}.
\end{aligned}
$$

❖ Strassen's algorithm for matrix multiplication

- Strassen's method
  - How does it work?
    - ✓ Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1, P_2, .. , P_7 which are $n/2 \times n/2$ submatrices
      i. Require us to perform seven multiplications of $n/2 \times n/2$ matrices
      ii. Takes $7T(n/2)$ times

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} ,
\end{aligned}
$$

$$
\begin{aligned}
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

    - ✓ Compute the desired submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding and subtracting carious combinations of the P_i matrices.
      i. Takes $\theta(n^2)$ times

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad C_{21} = P_3 + P_4$$

$$C_{12} = P_1 + P_2 \qquad C_{22} = P_5 + P_1 - P_3 - P_7$$

❖ Strassen's algorithm for matrix multiplication

- Strassen's method
  - ➢ Analyzing algorithm

✓ T(n) = $\underbrace{\theta(1)}_{\text{Index Calculation}} + \underbrace{\theta(n^2)}_{\text{Create 10 } s_i} + \underbrace{7T\left(\frac{n}{2}\right)}_{\text{Calculate 7 } P_i} + \underbrace{\theta(n^2)}_{\text{Compute } c_i}$

$= 7T\left(\frac{n}{2}\right) + \theta(n^2)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

✓ According to master method $T(n) = \theta\left(n^{\log 7}\right) = \theta(n^{2.8})$

✓ Strassen's algorithm, comparing with previous methods, is asymptotically faster!!

Thank You!