Data Structures & Algorithms

Lecture 5

Recap

- MAP ADT
- Hashmap
- Time complexity of a hashmap

Objectives

- What is an algorithmic strategy?

- Learn about commonly used Algorithmic Strategies
  - ❖ Brute-force
  - ❖ Divide-and-conquer
  - ❖ Dynamic programming
  - ❖ Greedy algorithms

- You will also see an example of how classical algorithmic problems can appear in daily life

Algorithm Classification

- Based on problem domain

- Based on algorithmic strategy

Algorithmic Strategies

- Approach to solving a problem
- Algorithms that use a similar problem solving approach can be grouped together
- Classification scheme for algorithms

Brute Force

Brute Force

- Straightforward approach to solving a problem based on the simple formulation of the problem
- Often, does not require deep analysis of the problem
- Perhaps the easiest approach to apply and is useful for solving problems of small-size

-

Brute Force

- May results in naïve solutions with poor performance

- Examples
  - ❖Computing $a^n (a > 0, n$ a non negative integer) by repetitive multiplication $a * a * \ \ldots * a$
  - ❖Computing $n!$
  - ❖Sequential search
  - ❖Selection sort

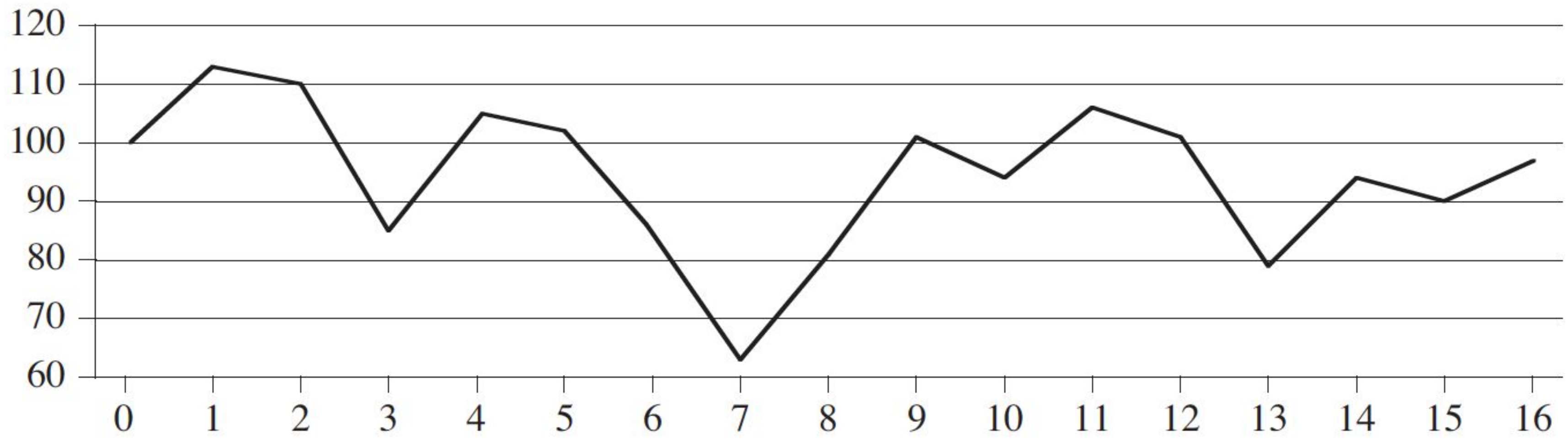
-

Example Algorithmic Problem

- Maximum subarray problem
  - Given a sequence of integers $i_1, i_1, \ldots, i_n$ find the sub-sequence with the maximum sum
    - If all the numbers are negative, the result is 0
  - Examples:
    - 1, -3, 4, -2, -1, 6 gives the solution ?

Max Subarray in Real-life

- Information about the price of stock in a Chemical manufacturing company after the close of trading over a period of 17 days

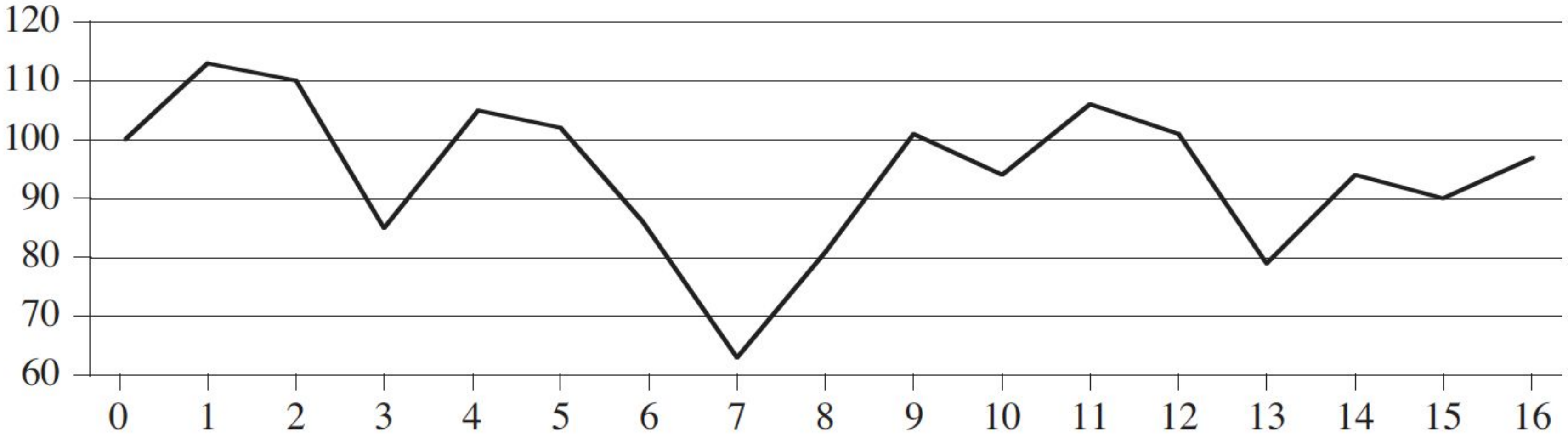


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- When to buy the stock and when to sell it to maximize the profit?

Max Subarray in Real-life

- Transformation to convert this problem into the max-subarry problem

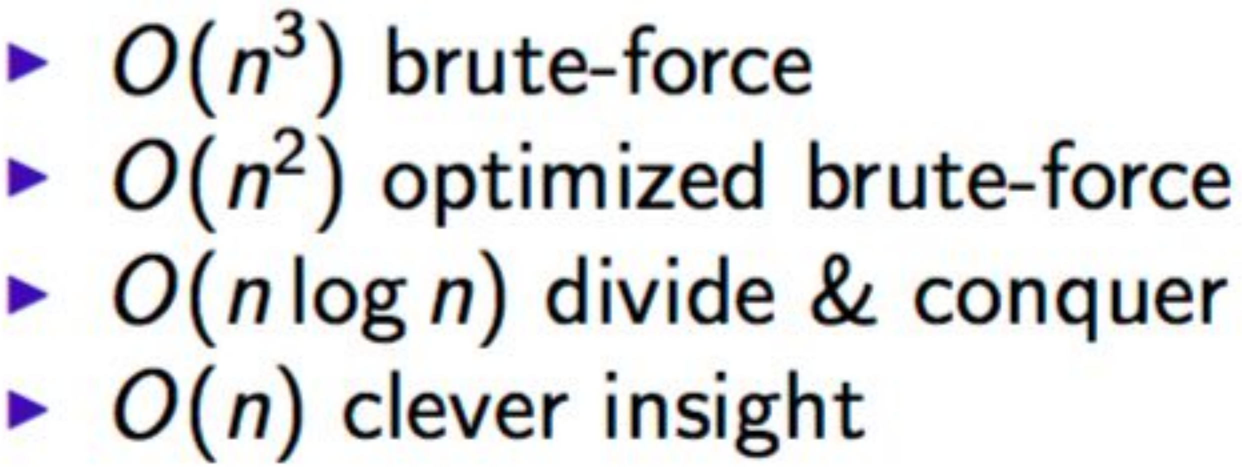


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- When to buy the stock and when to sell it to maximize the profit? *Now we can answer this by finding the sequence of days over which the net change is maximum*

Example Algorithmic Problem

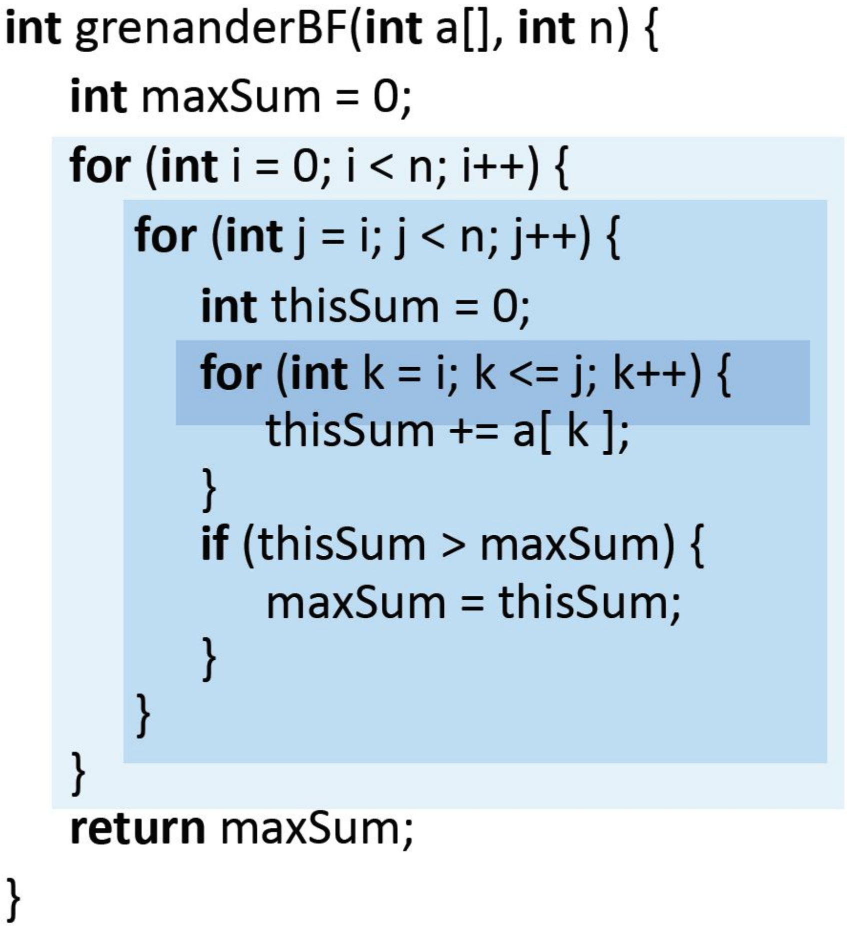
- Maximum subarray problem



- $O(n^3)$ brute-force
- $O(n^2)$ optimized brute-force
- $O(n \log n)$ divide & conquer
- $O(n)$ clever insight

Brute Force Approach to Our Problem

- We can easily devise a brute-force solution to this problem – O(?)



```
int grenanderBF(int a[], int n) {
    int maxSum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                thisSum += a[ k ];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Brute Force

- The most straightforward and the easiest of all approach
- Often, does not required deep analysis of the problem
- May results in naïve solutions with poor performance, but easy to implement

Divide-and-Conquer

Divide-and-Conquer

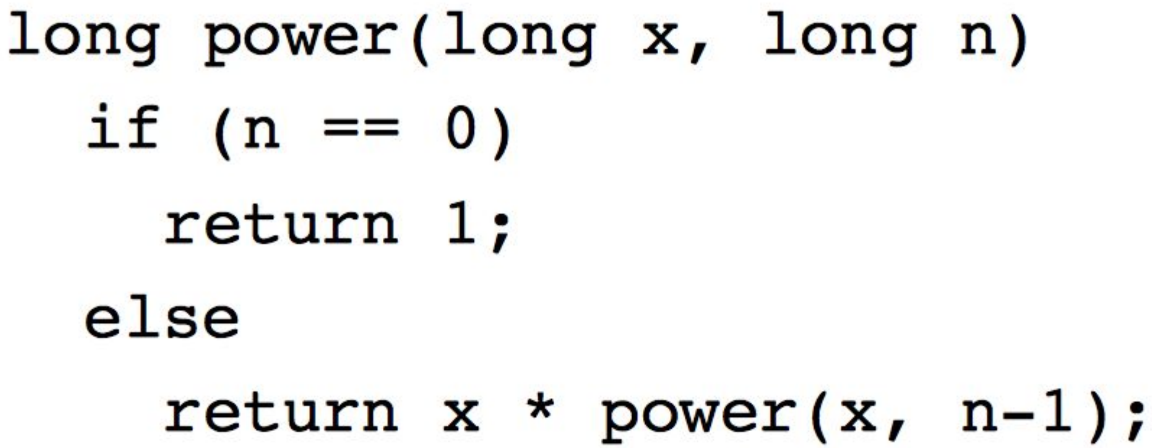
- Solving a problem *recursively*, applying three steps at each level of *recursion*
  - **Divide** the problems into a number a sub-problems that are smaller instances of the same problem
  - **Conquer** the sub-problems by solving them recursively. If the sub-problems size is small enough, just solve it in a straightforward manner
  - **Combine** the solutions to the sub-problems into the solution for the original problem

Recursion and Recurrence Relations

- Recursion
  - A wonderful programming tool
  - A function is said to be recursive if it calls itself – usually with "*smaller or simpler*" inputs
  - Two properties:
    a) A problem should be solvable by utilizing the solutions to the smaller versions of the same problem,
    b) The smaller versions should reduce to easily solvable cases

Recursion and Recurrence Relations

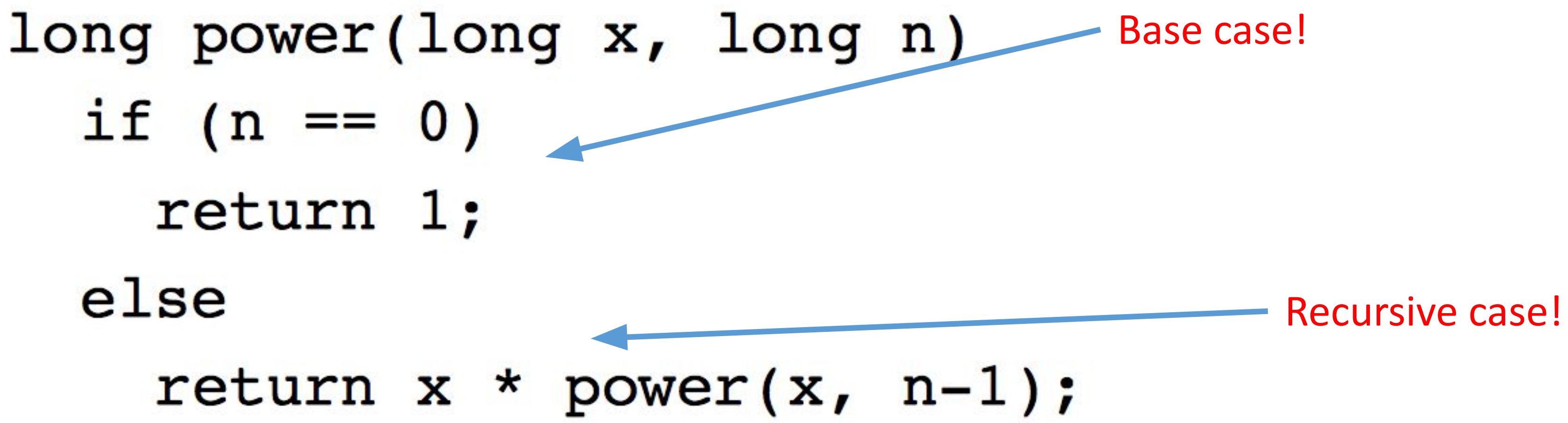
- Recursion



```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

Recursion and Recurrence Relations

- Recursion



```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

Base case!

Recursive case!

Recursion and Recurrence Relations

- Recurrence Relations
  - Are used to determine the running time of recursive algorithms
  - Recurrence relations are themselves recursive
  - Let $T(n)$ = Time required to solve the problem of size n

$T(0) =$ time to solve problem of size 0
– Base Case

$T(n) =$ time to solve problem of size n
– Recursive Case

Recursion and Recurrence Relations

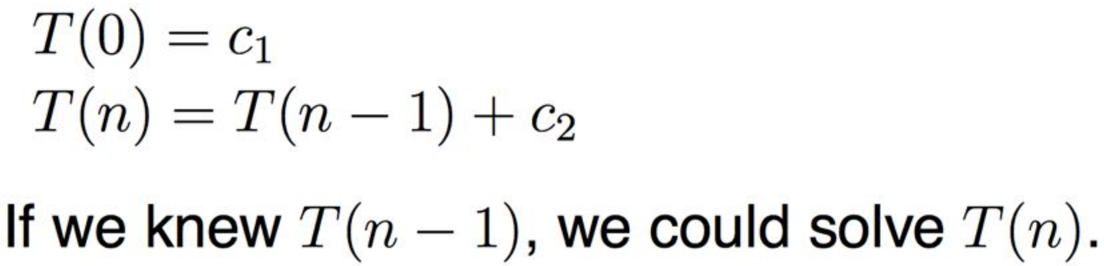
- Recurrence Relations

```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

$$\begin{aligned} T(0) &= c_1 && \text{for some constant } c_1 \\ T(n) &= c_2 + T(n-1) && \text{for some constant } c_2 \end{aligned}$$

Recursion and Recurrence Relations

- Recurrence Relations



$$T(0) = c_1$$
$$T(n) = T(n-1) + c_2$$

If we knew $T(n-1)$, we could solve $T(n)$.

Recursion and Recurrence Relations

- Recurrence Relations

$$
\begin{aligned}
T(0) &= c_1 \\
T(n) &= T(n-1) + c_2
\end{aligned}
$$

If we knew $T(n-1)$, we could solve $T(n)$.

$$
\begin{array}{lll}
T(n) & = T(n-1) + c_2 & T(n-1) = T(n-2) + c_2 \\
 & = T(n-2) + c_2 + c_2 & \\
 & = T(n-2) + 2c_2 & T(n-2) = T(n-3) + c_2 \\
 & = T(n-3) + c_2 + 2c_2 & \\
 & = T(n-3) + 3c_2 & T(n-3) = T(n-4) + c_2 \\
 & = T(n-4) + 4c_2 & \\
 & = \ldots & \\
 & \boxed{= T(n-k) + kc_2} &
\end{array}
$$

Recursion and Recurrence Relations

- Recurrence Relations

$$T(0) = c_1$$
$$T(n) = T(n-k) + k * c_2 \quad \text{for all } k$$

If we set $k = n$, we have:

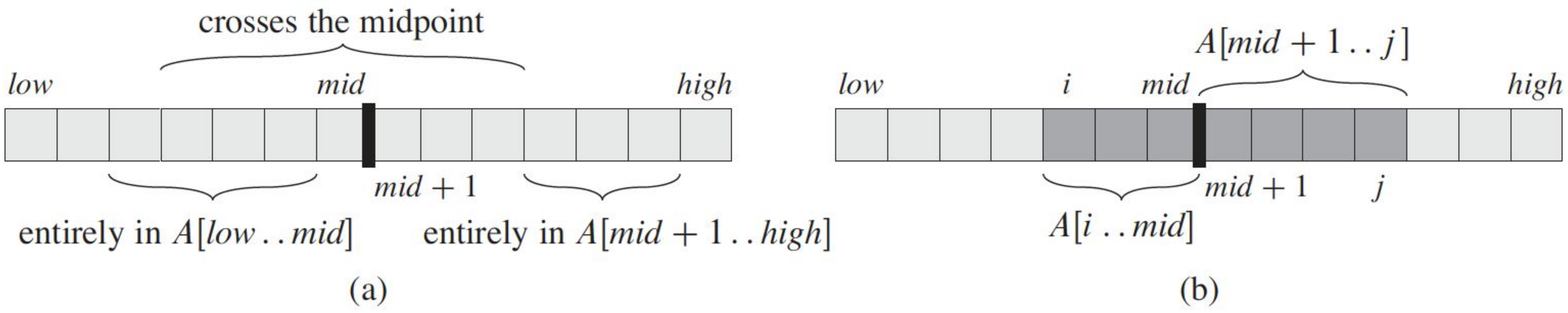
$$\begin{aligned} T(n) &= T(n-n) + nc_2 \\ &= T(0) + nc_2 \\ &= c_1 + nc_2 \end{aligned}$$

Back to Divide-and-Conquer

- Solving a problem *recursively*, applying three steps at each level of *recursion*
  - **Divide** the problems into a number a sub-problems that are similar instances of the same problem
  - **Conquer** the sub-problems by solving them recursively. If the sub-problems size is small enough, just solve it in a straightforward manner
  - **Combine** the solutions to the sub-problems into the solution for the original problem
- Examples: Quicksort, Mergesort, etc.

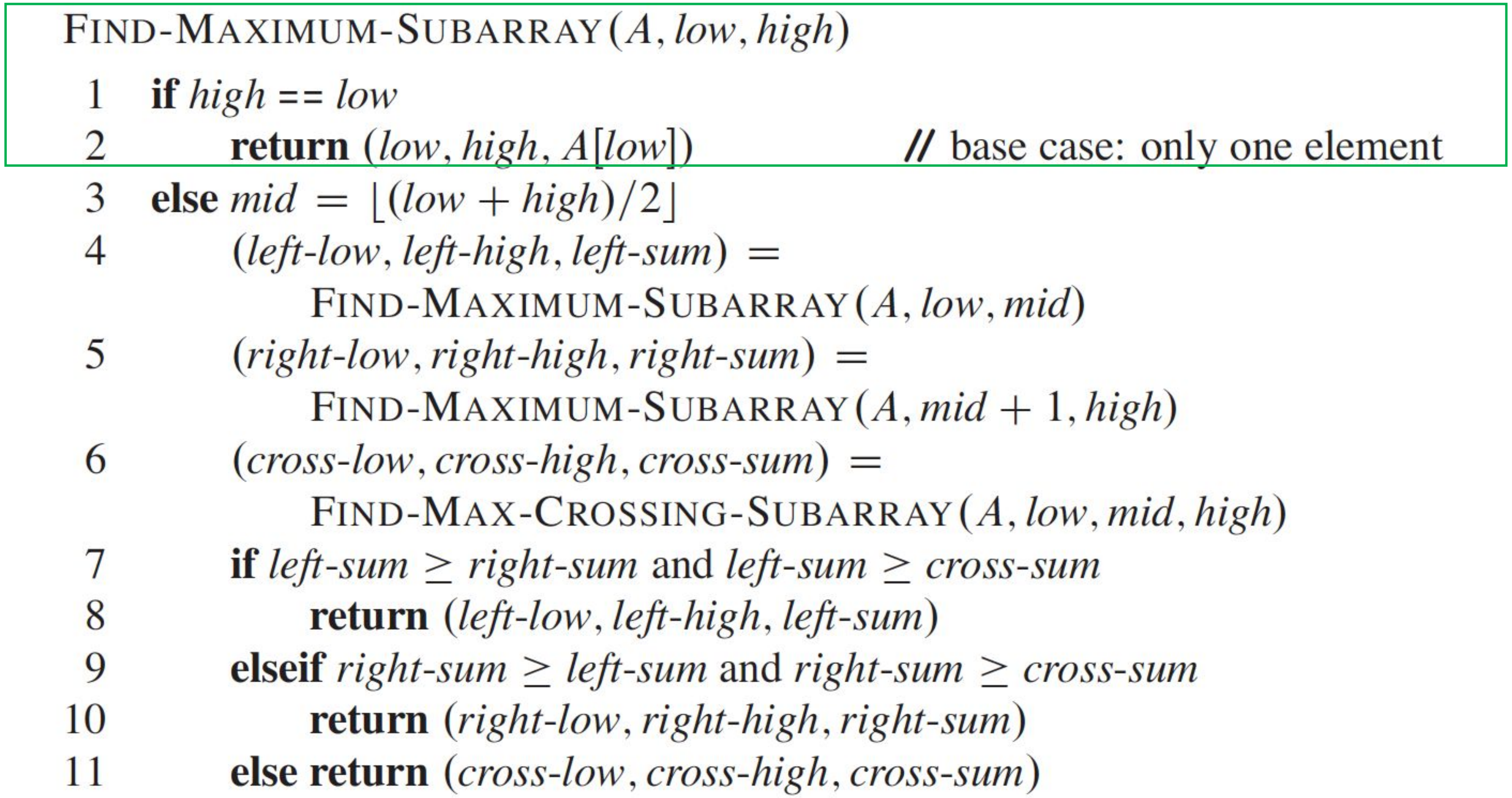
Divide-and-Conquer

- Max Subarray Problem



Divide-and-Conquer

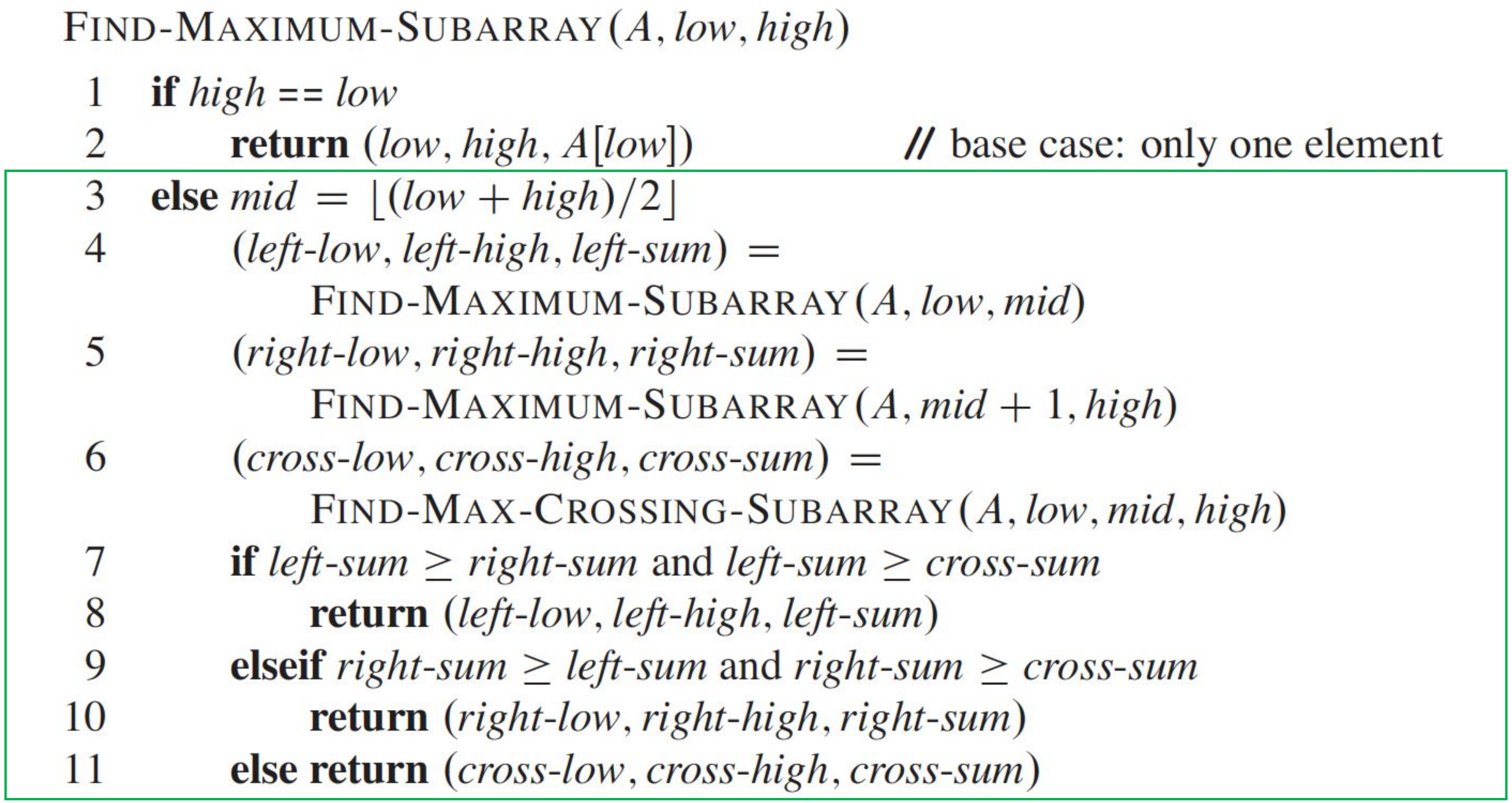
- Max Subarray Problem



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = $\lfloor (low + high)/2 \rfloor$
    (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 $\geq$ right-sum and left-sum $\geq$ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum $\geq$ left-sum and right-sum $\geq$ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

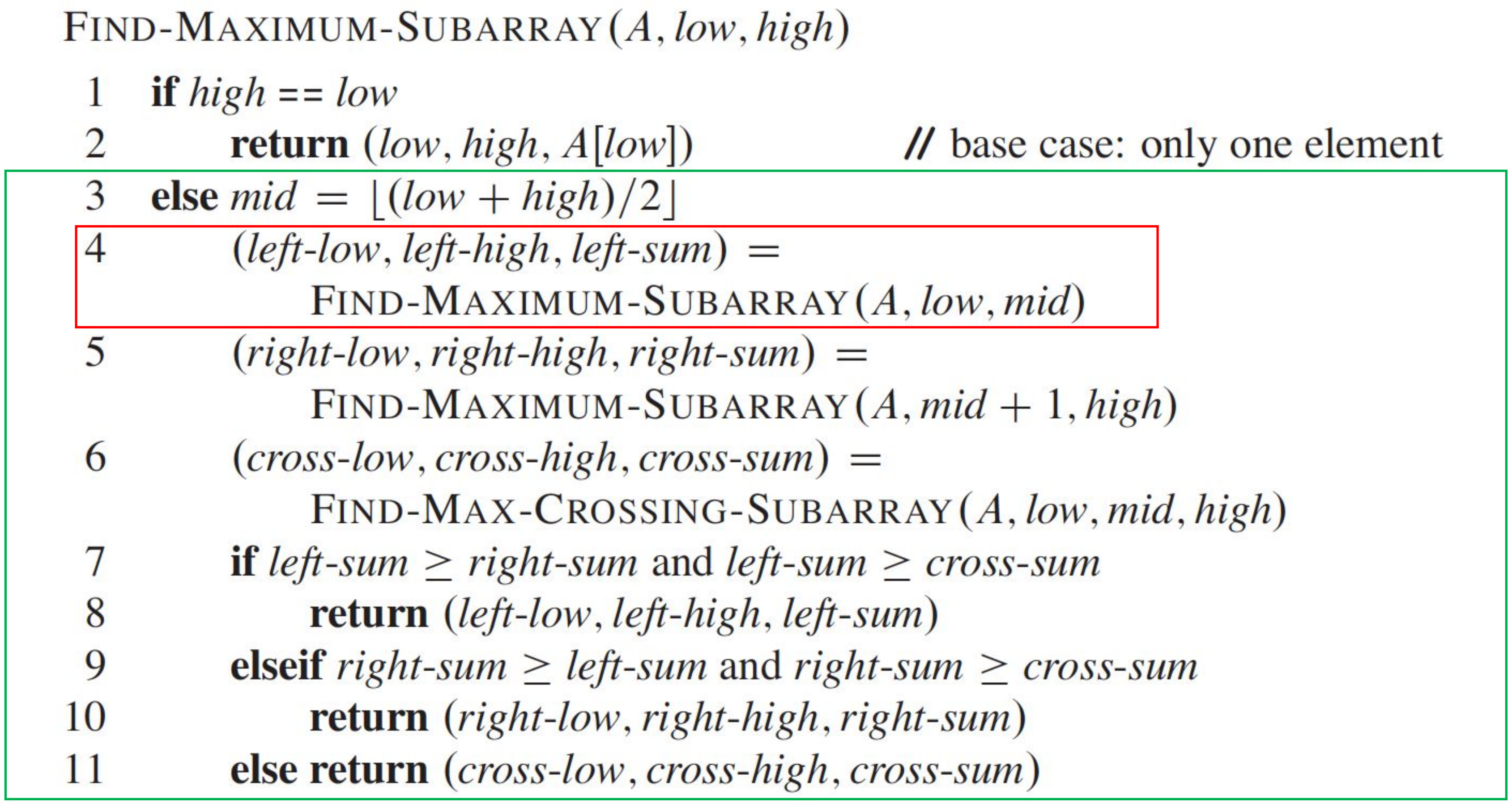
- Max Subarray Problem



```
FIND-MAXIMUM-SUBARRAY($A, low, high$)
if $high == low$
    return ($low, high, A[low]$)            // base case: only one element
else $mid = \lfloor (low + high)/2 \rfloor$
    ($left\text{-}low, left\text{-}high, left\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, low, mid$)
    ($right\text{-}low, right\text{-}high, right\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$)
    ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low, left\text{-}high, left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low, right\text{-}high, right\text{-}sum$)
    else return ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$)
```

Divide-and-Conquer

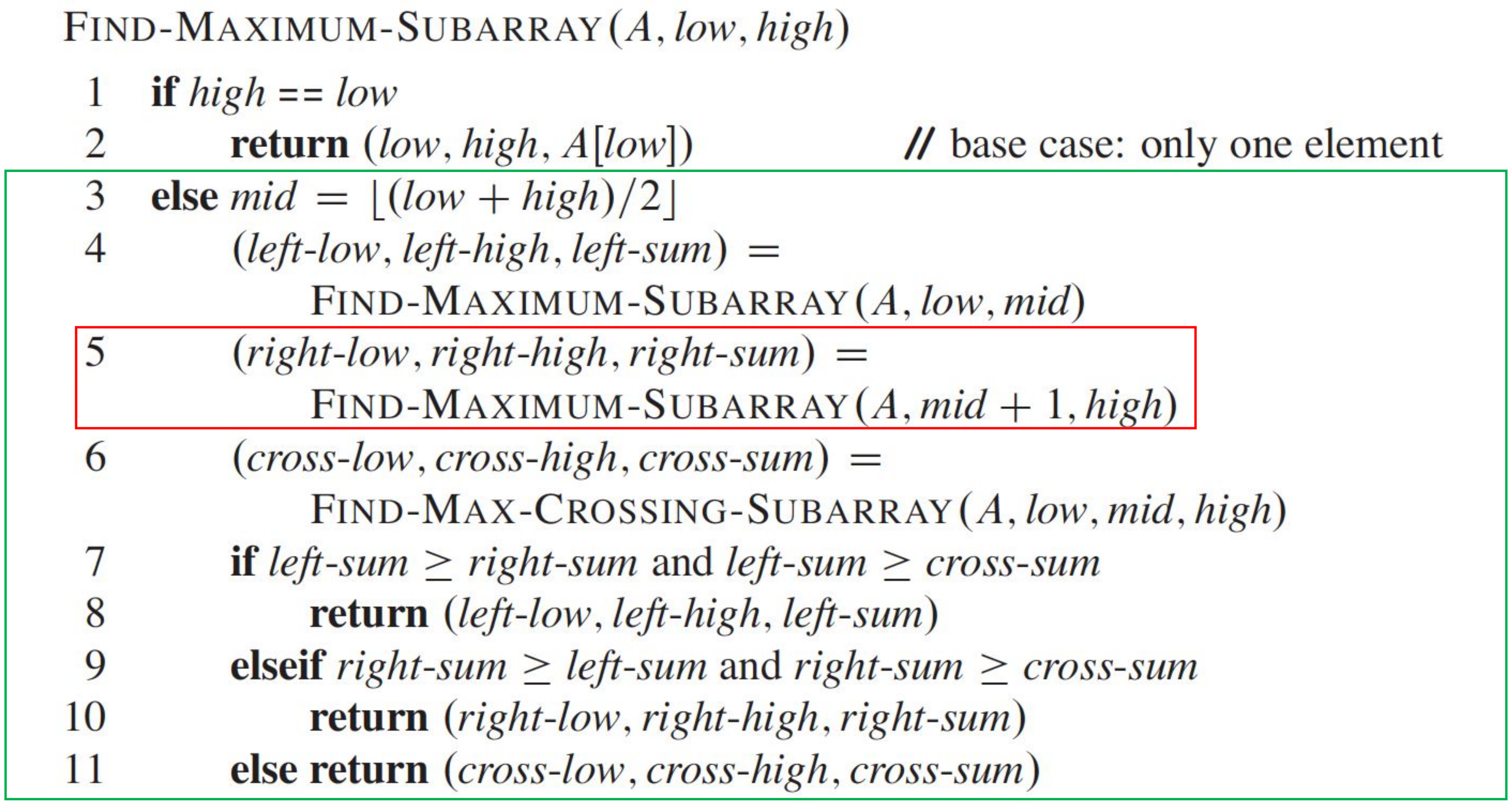
- Max Subarray Problem



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = $\lfloor (low + high)/2 \rfloor$
    (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 $\geq$ right-sum and left-sum $\geq$ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum $\geq$ left-sum and right-sum $\geq$ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

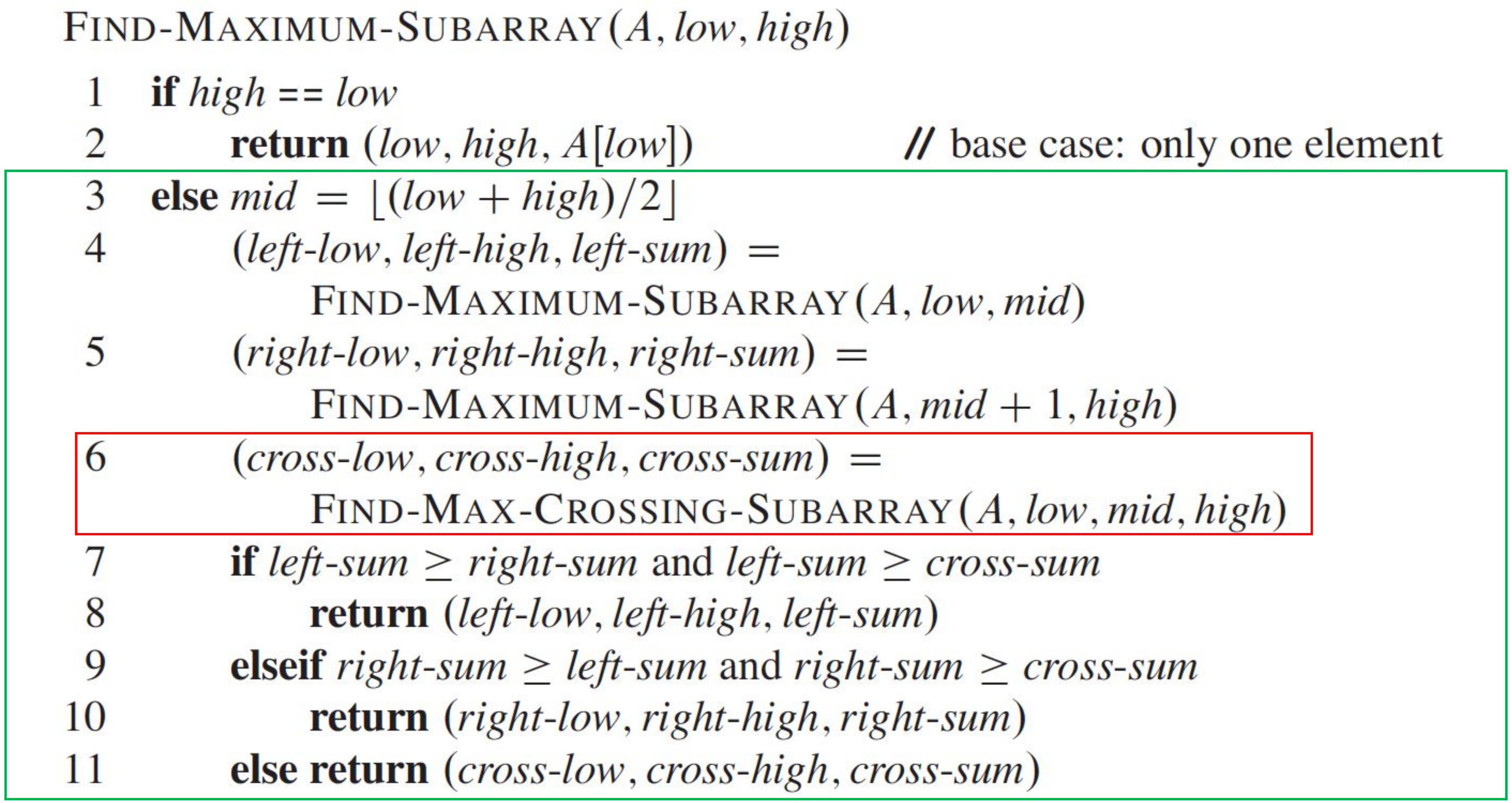
- Max Subarray Problem



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = $\lfloor (low + high)/2 \rfloor$
    (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 $\geq$ right-sum and left-sum $\geq$ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum $\geq$ left-sum and right-sum $\geq$ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem



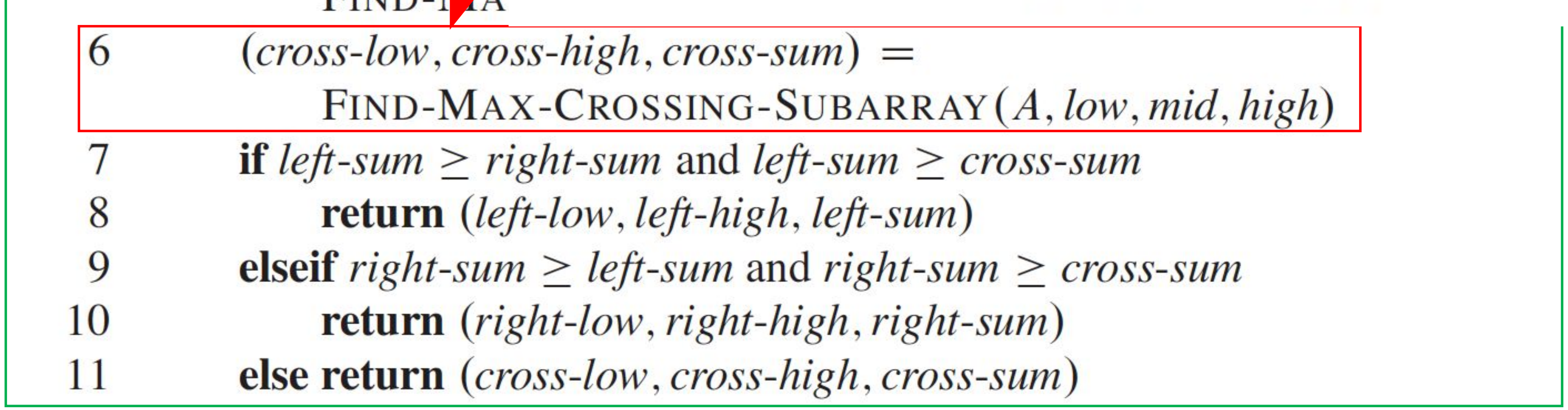
```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = $\lfloor (low + high)/2 \rfloor$
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A,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

Divide

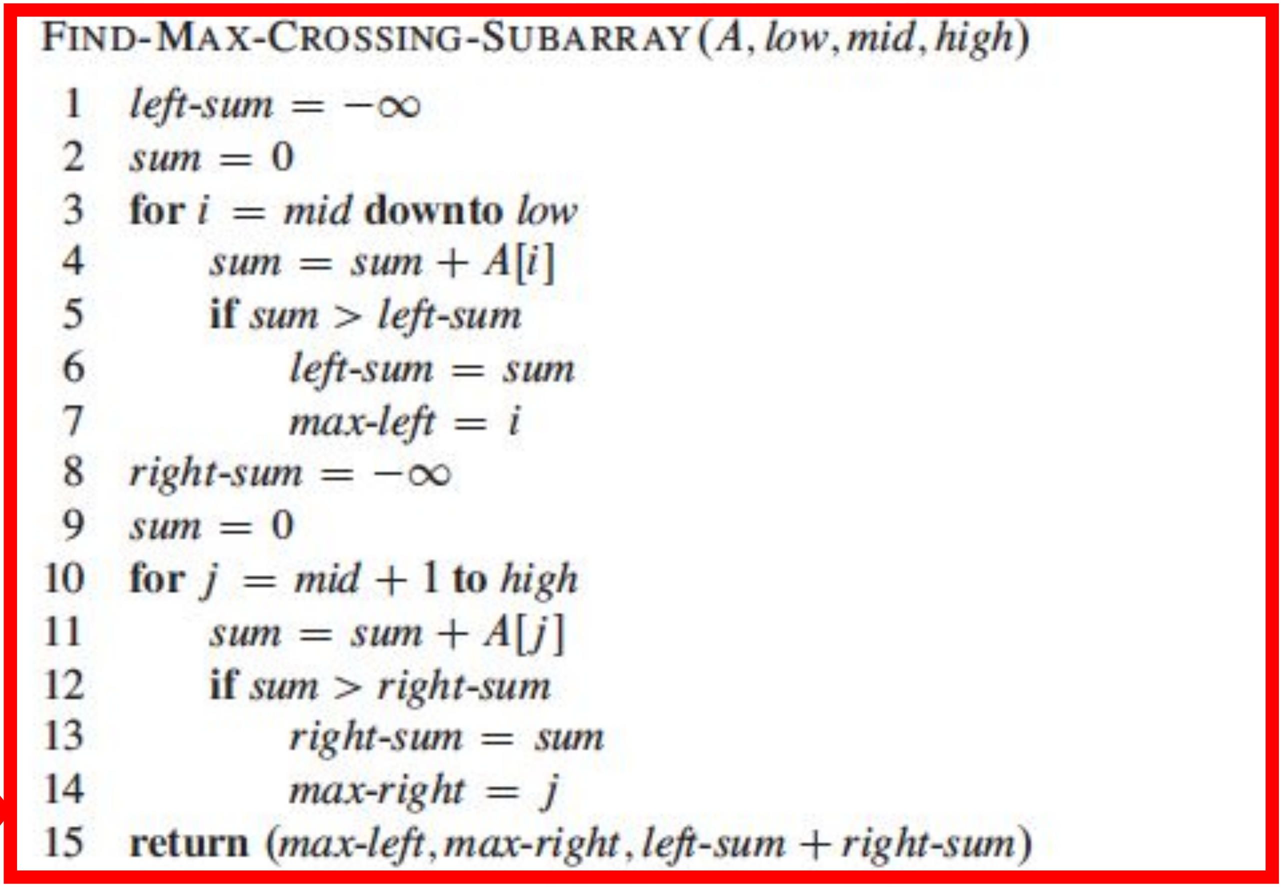
- Max Subarray Problem

FIND-MAXIMUM-SUBA

```
if $high == low$
    return ($low$, $hi$
else $mid = \lfloor(low$ -
    ($left\text{-}low$, $left\text{-}h$
        FI
    ($right\text{-}l$
        FIND-MA
```



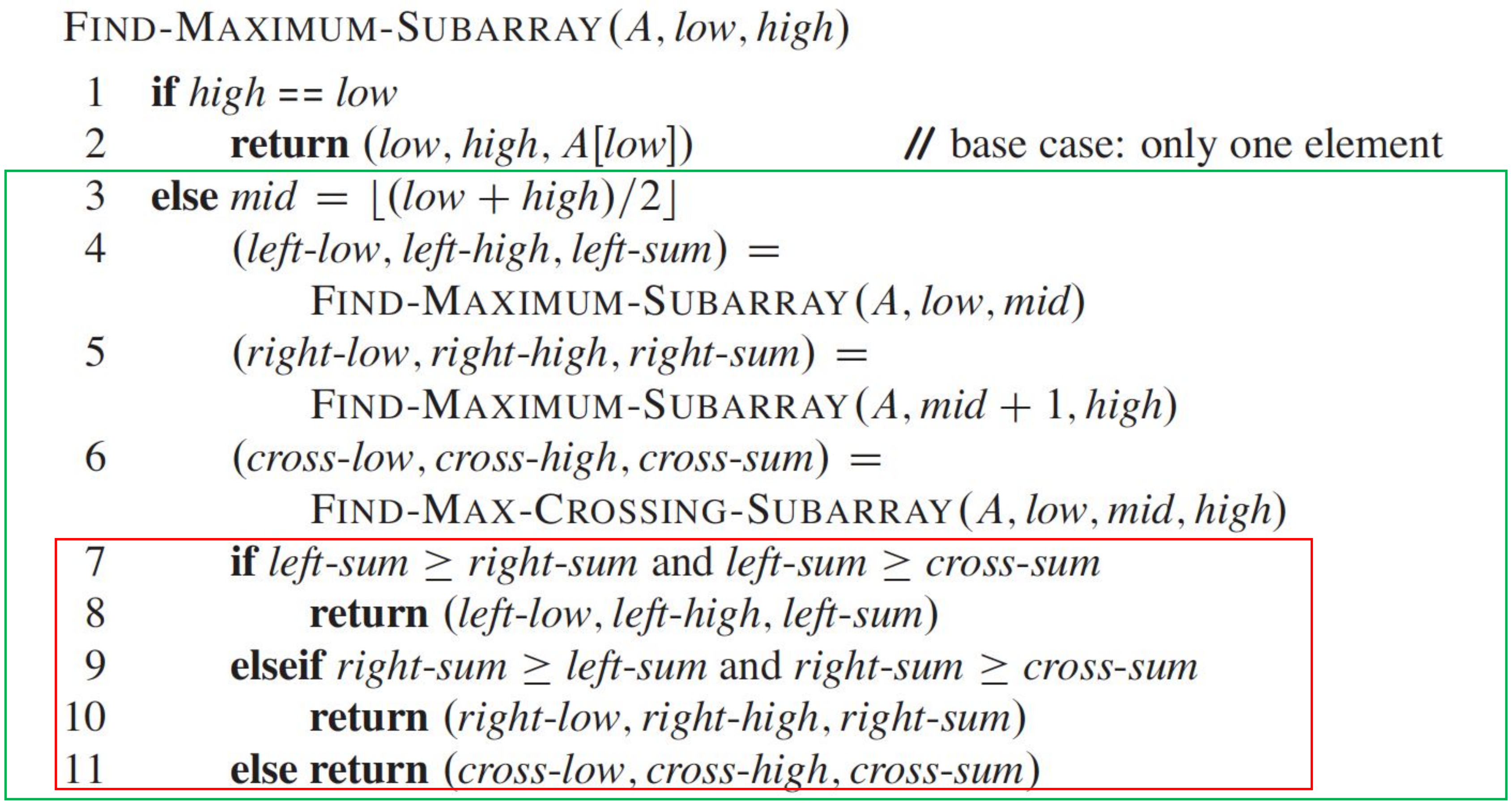
```
    ($cross\text{-}low$, $cross\text{-}high$, $cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low$, $left\text{-}high$, $left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low$, $right\text{-}high$, $right\text{-}sum$)
    else return ($cross\text{-}low$, $cross\text{-}high$, $cross\text{-}sum$)
```



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

Divide-and-Conquer

- Max Subarray Problem



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])                  // base case: only one element
else mid = $\lfloor (low + high)/2 \rfloor$
    (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 $\geq$ right-sum and left-sum $\geq$ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum $\geq$ left-sum and right-sum $\geq$ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide-and-Conquer

- Max Subarray Problem – Time Complexity

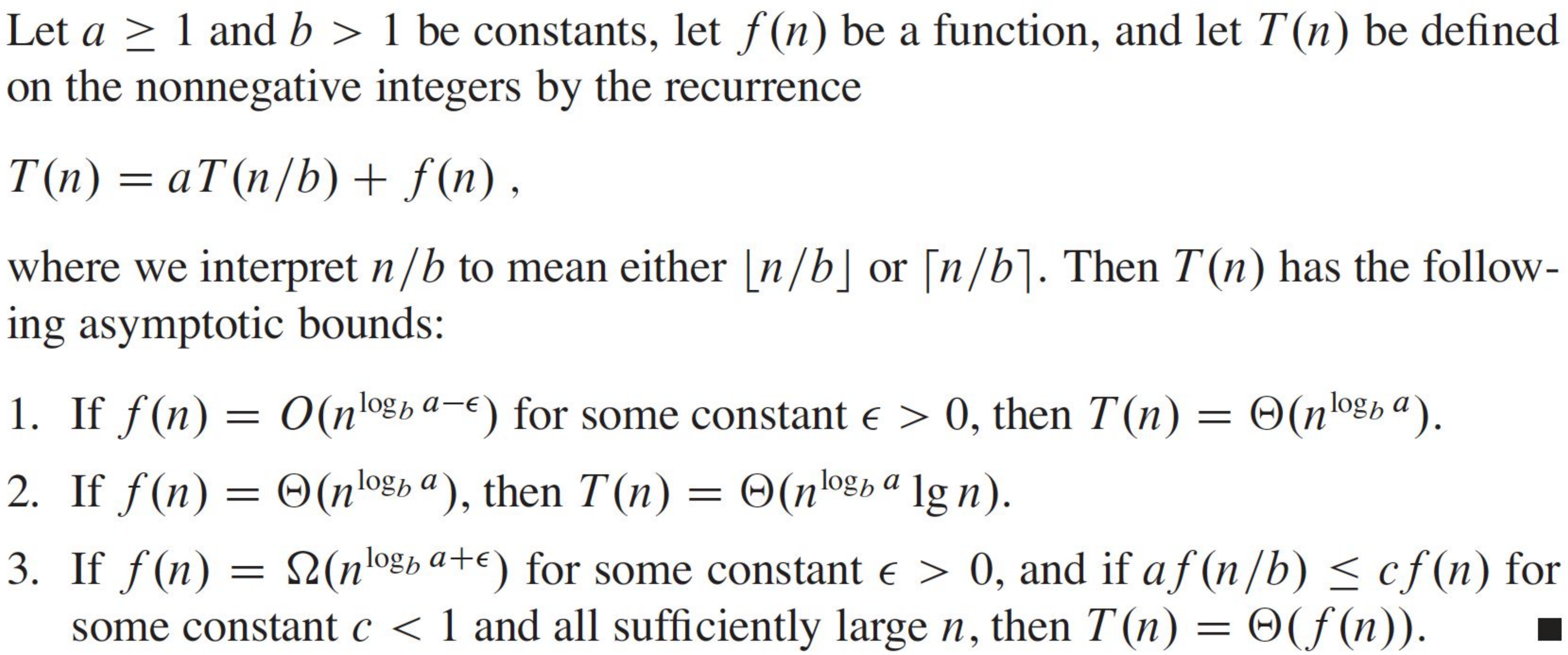
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

- This type of recurrence is called "*Divide-and-Conquer*" recurrence

We can solve this recurrence using the **"Master Theorem"** **--**
Cormen's, Chapter 4

Divide-and-Conquer

- Master Theorem



Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

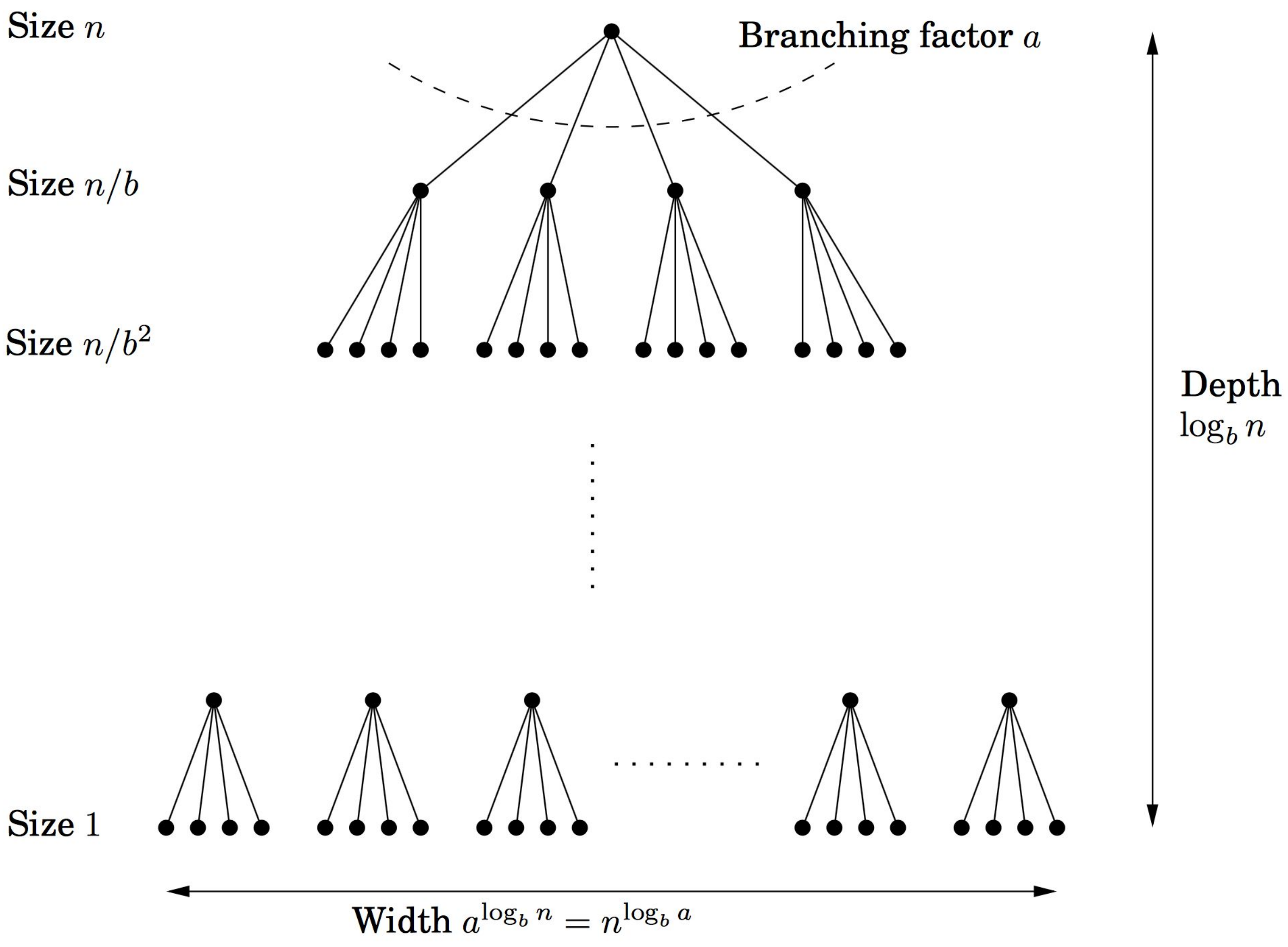
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Divide-and-Conquer

- Master Theorem

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



Divide-and-Conquer

- Max Subarray Problem – Time Complexity

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

- Case 2 from Master Theorem applies, thus we have the solution

$$T(n) = \Theta(n \lg n).$$

Master Theorem

- You will get back to it in your tutorial today

- With some examples

Dynamic Programming

Dynamic Programming

- Similar to divide-and-conquer, it solves the problem by combining solutions to the sub-problems
- But it applies when sub-problems overlap
- That is, sub-problems **share sub-sub-problems**!
- To avoid solving the same sub-problems more than once, the results are stored in a data structure that is updated dynamically

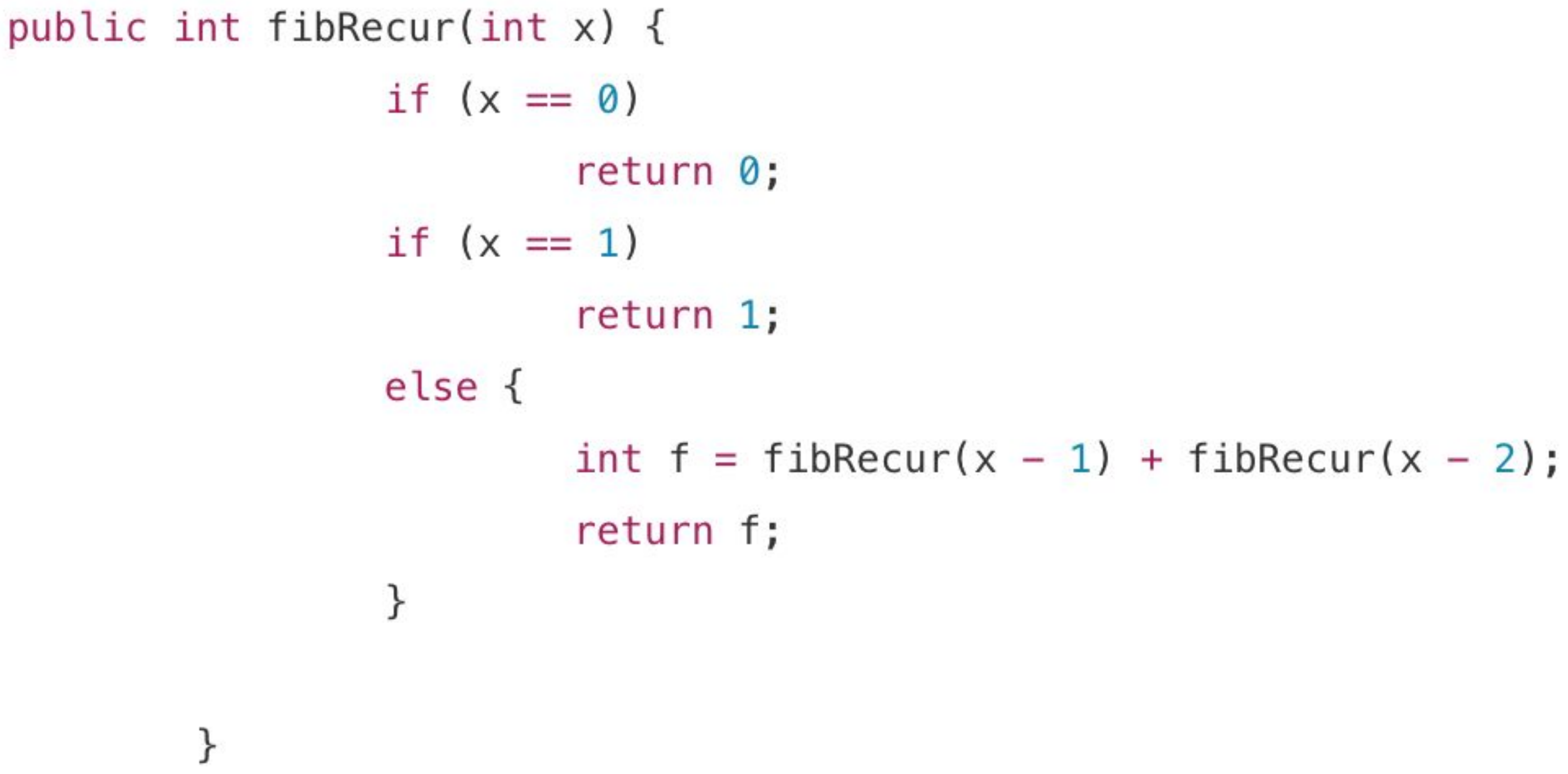
Dynamic Programming

- Fibonacci Numbers

```
Fibonacchi(N) = 0                                   for n=0
       = 1                                   for n=1
       = Fibonacchi(N-1)+Finacchi(N-2)       for n>1
```

Dynamic Programming

- Fibonacci Numbers

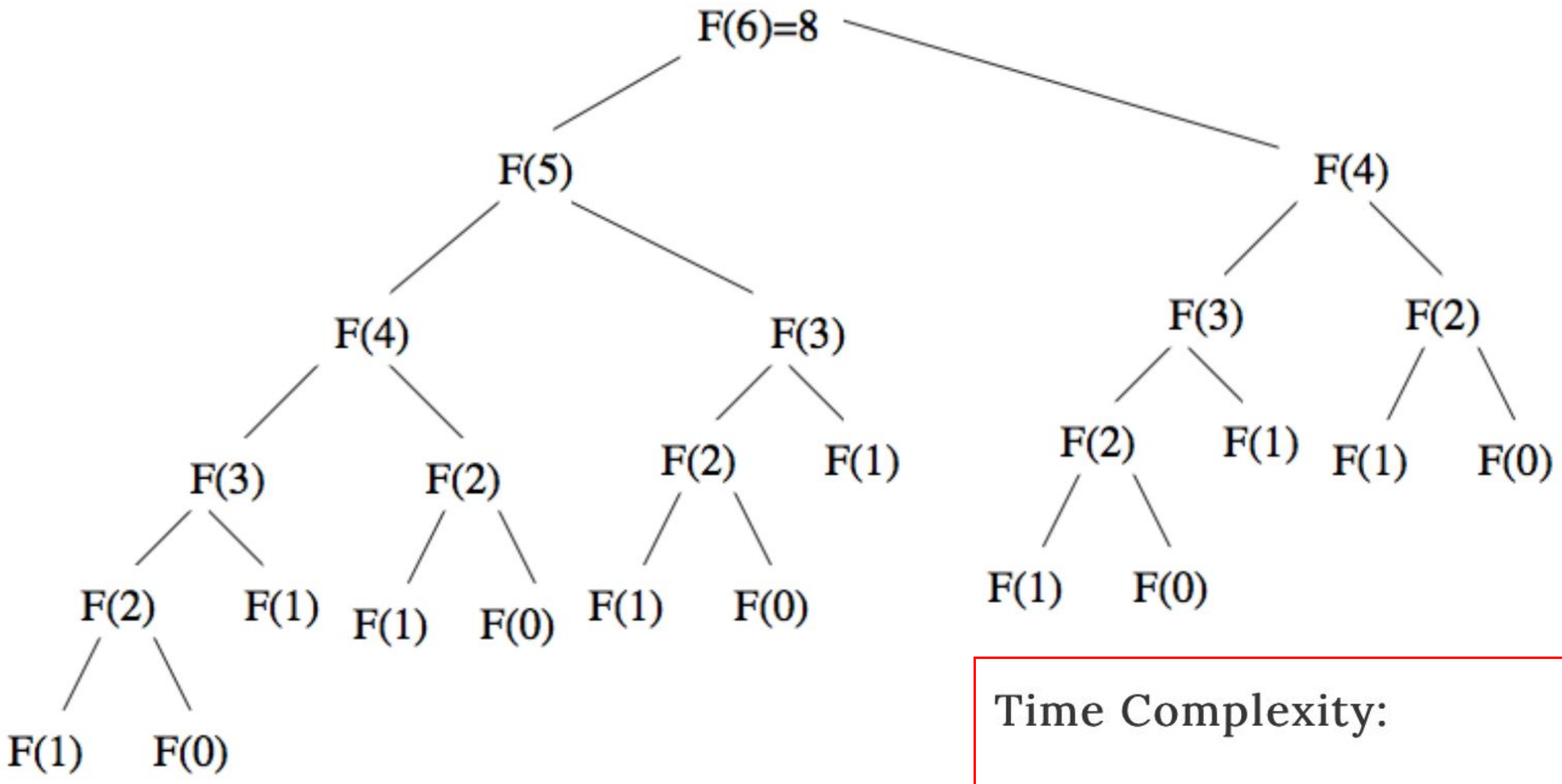


```
public int fibRecur(int x) {
                if (x == 0)
                        return 0;
                if (x == 1)
                        return 1;
                else {
                        int f = fibRecur(x - 1) + fibRecur(x - 2);
                        return f;
                }

        }
```

Dynamic Programming

- $n-th$ Fibonacci Numbers

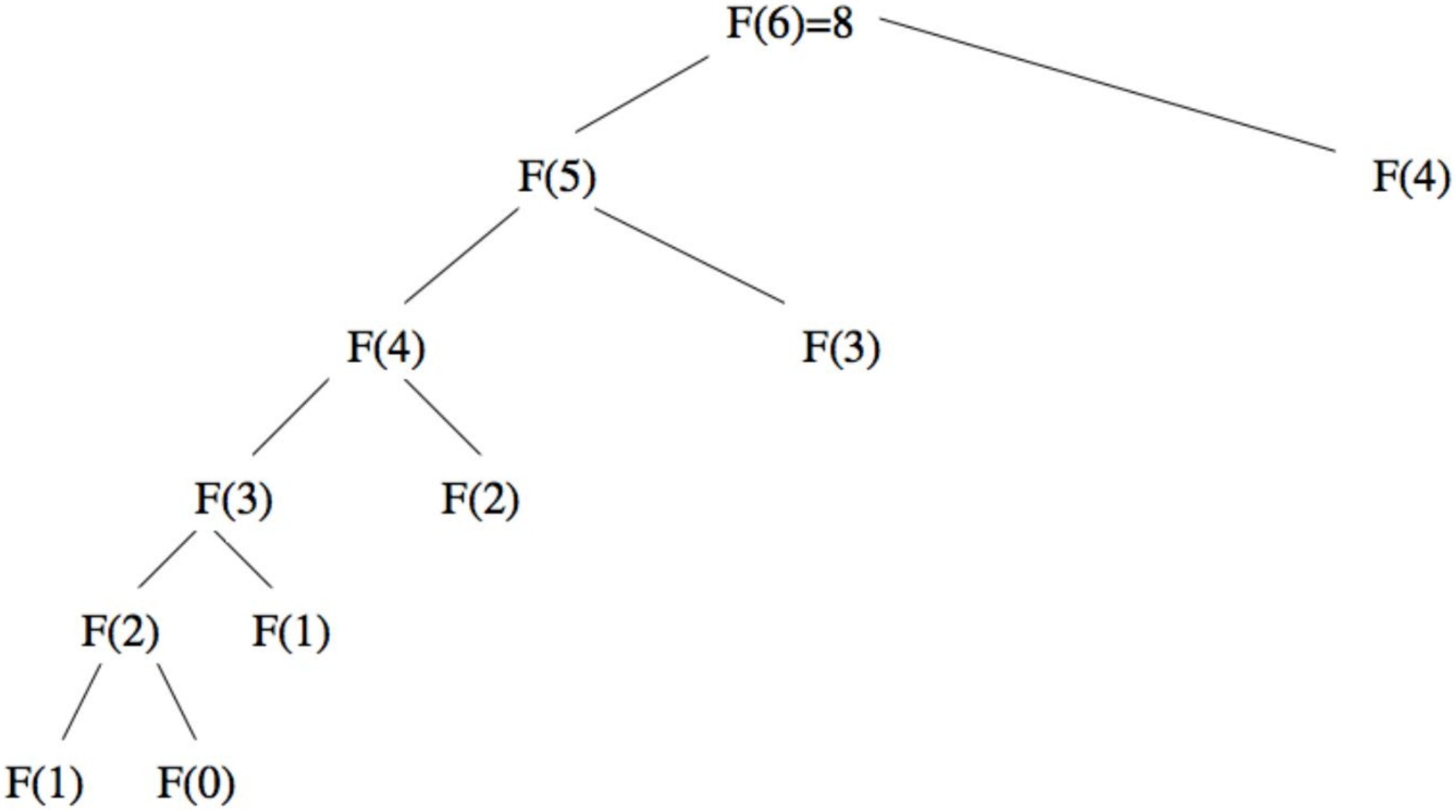


Time Complexity:

$T(n) = T(n-1) + T(n-2) + 1 = 2^n = O(2^n)$

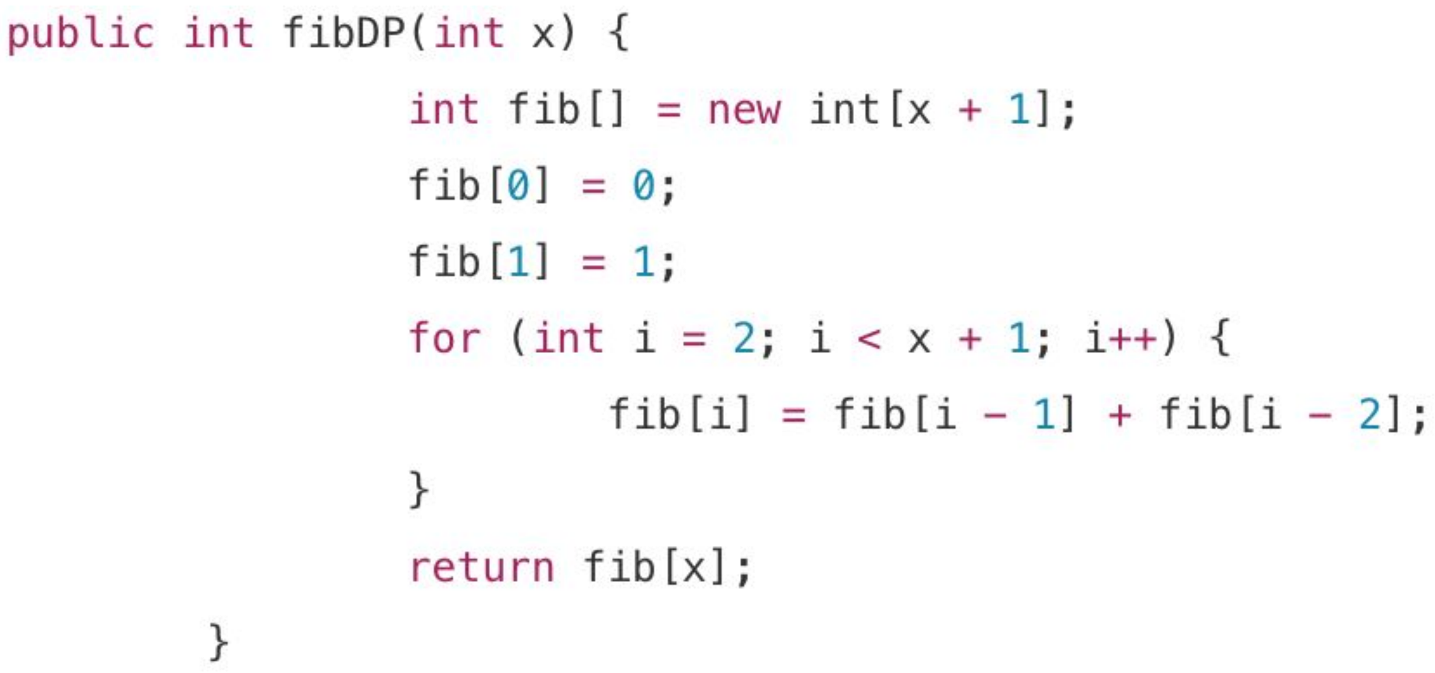
Dynamic Programming

- $n - th$ Fibonacci Numbers



Dynamic Programming

- Fibonacci Numbers – Bottom-up Fashion



```
public int fibDP(int x) {
                int fib[] = new int[x + 1];
                fib[0] = 0;
                fib[1] = 1;
                for (int i = 2; i < x + 1; i++) {
                        fib[i] = fib[i - 1] + fib[i - 2];
                }
                return fib[x];
        }
```

Time Complexity: O(n) , Space Complexity : O(n)

Dynamic Programming

- Key is to relate the solution of the whole problem and the solutions of subproblems.

  - ❖ Same is true of divide & conquer, but here the subproblems need not be disjoint. – they need not divide the input (i.e., they can “overlap”)

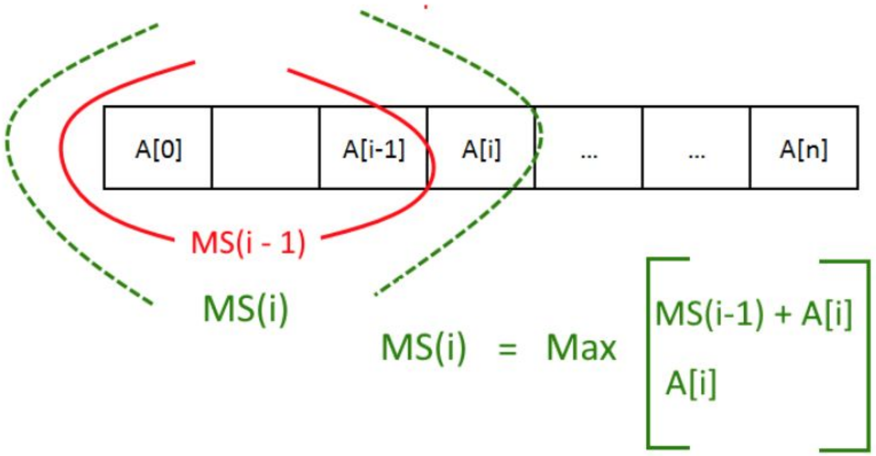
- A dynamic programming algorithm computes the solution of every subproblem needed to build up the solution for the whole problem.
  - compute each solution using the above relation
  - store all the solutions in an array (or matrix)
  - algorithm simply fills in the array entries in some order

Dynamic Programming

- Max Subarray Problem

- Let $S(i)$ be the sum at ith-index

A[0]		A[i-1]	A[i]	...		A[n]



- Then it can be recursively defined as

$$S(i) = \max((S(i-1) + A[i]), A[i])$$

-

Dynamic Programming

- Max Subarray Problem

$$S(i) = \max((S(i-1) + A[i]), A[i])$$

- Apply this definition to solve the problem for the following sequence

-2, -3, 4, -1, -2, 1, 5, -3

Dynamic Programming

- Max Subarray Problem

```
Max-Subarray-Sum (A, n)
opt $\leftarrow$ 0, opt' $\leftarrow$ 0
for i $\leftarrow$ 1 to n
    opt' $\leftarrow \max\{0, opt' + A[i]\}$
opt $\leftarrow \max\{opt, opt'\}$
return opt
```

Elements of Dynamic Programming

- So we just learned how DP works

- But, given a problem, how do we know:
    - Whether we can use DP
    - How to attack the problem with DP

Will be covered in detail in the tutorial

Greedy Algorithms

Greedy Algorithms

- Finding solutions to problem **step-by-step**
- A partial solution is incrementally expanded towards a complete solution
- In each step, there are several ways to expand the partial solution
- The best alternative for the moment is chosen, the others are discarded.
- Thus, at each step the choice must be **locally optimal** – this is the central point of this technique

Greedy Algorithms

- For example, counting to a desired value using the least number of coins

- Let’s say, we are given coins of value 1, 2, 5 and 10 of some currency. And the target value is 16 in that currency

- How will you proceed?

Greedy Algorithms

- Not always gives the optimal solution
- Let’s say, a monetary system consists of only coins of worth 1, 7 and 10.
- How would a greedy approach count out the value of 15?

Greedy Algorithms

- Examples
  - Finding the minimum spanning tree of a graph (Prim’s algorithm)
  - Finding the shortest distance in a graph (Dijkstra’s algorithm)
  - Using Huffman trees for optimal encoding of information
  - The Knapsack problem

- We will go through the first two algorithms in detail when we learn about Graphs; therefore, I will end today’s lecture here.

- You are strongly advised to read about the discussed topics, as well as other algorithmic strategies such as
  - “Combinatorial search & Backtracking”
  - “Branch and Bound”

Did we achieve today's objectives?

- What is an algorithmic strategy?

- Learn about commonly used Algorithmic Strategies
  - ❖ Brute-force
  - ❖ Divide-and-conquer
  - ❖ Dynamic programming
  - ❖ Greedy algorithms

- You also saw an example of how classical algorithmic problems can appear in daily life