Last Week

Introduction

- Lecturer
- Course Details
- A.M. Turing Award Winners for Algorithms
- What Is This Course About
- What Are Algorithms
- What Does It Mean to Analyze An Algorithm
- Comparing Time Complexity

Asymptotic Notations and Recurrences

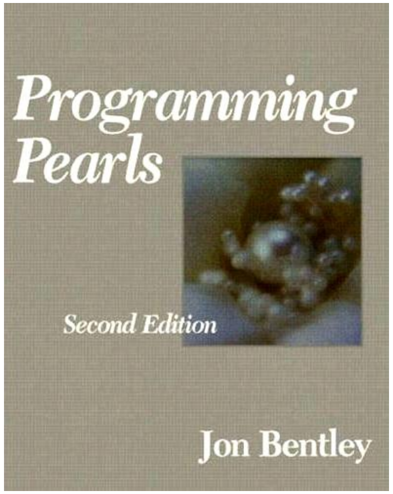
- Asymptotic Notations（渐近记号）
  - Big-Oh
  - Big-Omega
  - Big-Theta
  - Algorithm Design and Algorithm Turing
- Solving Recurrences
  - Recursion-tree Method（递归树法）
  - Substitution Method（代入法/替代法）
  - Master Method and Master Theorem (主方法)

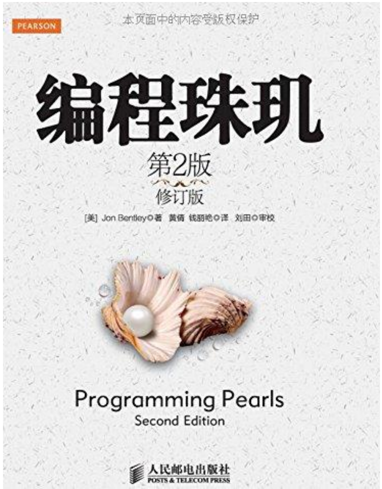
Some Thoughts on Algorithm Design

- Algorithm Design, as taught in this class, is mainly about designing algorithms that have big-Oh running times.
- As n gets larger and larger, O(nlogn) algorithms will run faster than $O(n^2)$ ones and O(n) algorithms will beat O(nlogn) ones.
- Good algorithm design & analysis allows you to identify the hard parts of your problem and deal with them effectively.
- Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code!
- A few hours of abstract thought devoted to algorithm design often results in faster, simpler, and more general solutions.

Algorithm Tuning

- After algorithm design one can continue on to Algorithm tuning
  - concentrate on improving algorithms by cutting down on the constants in the big O() bounds.
  - needs a good understanding of both algorithm design principles and efficient use of data structures.
- In this course we will not go further into algorithm tuning
  - For a good introduction, see chapter 9 in Programming Pearls, 2nd ed by Jon Bentley





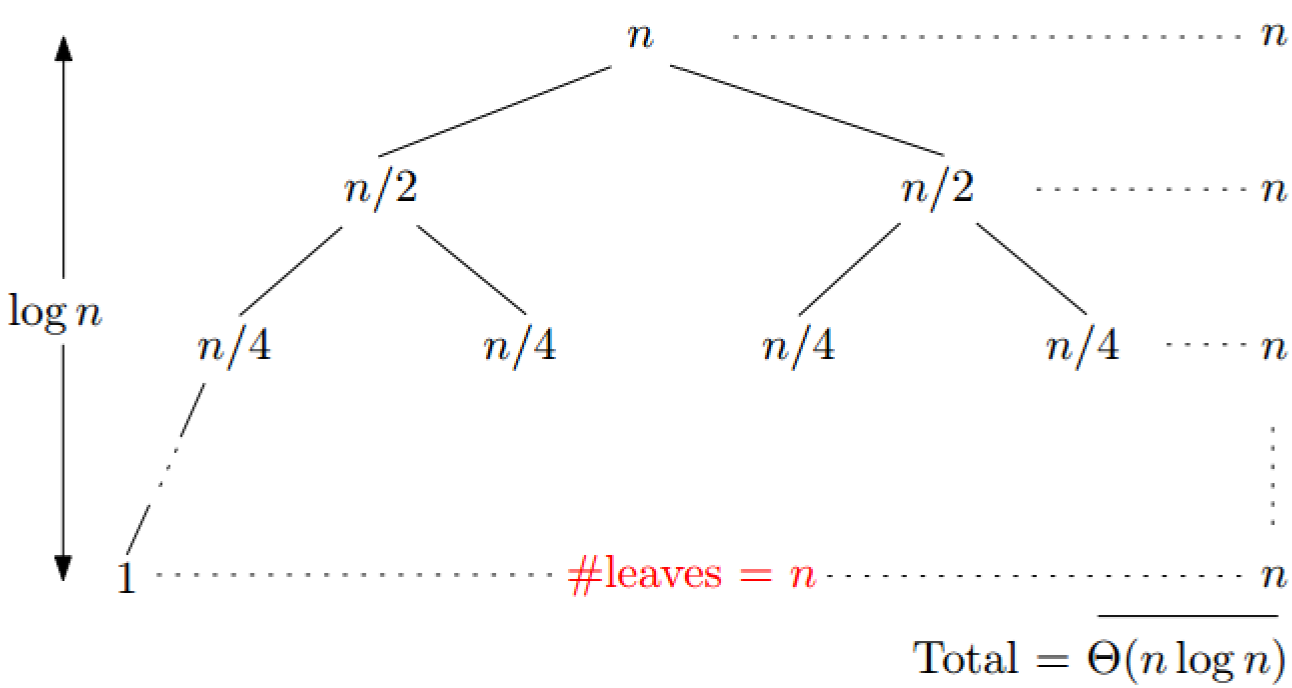
Outline

- Asymptotic Notations（渐近记号）
  - Big-Oh
  - Big-Omega
  - Big-Theta
  - Algorithm Design and Algorithm Turing
- Solving Recurrences
  - Recursion-tree Method（递归树法）
  - Substitution Method（代入法/替代法）
  - Master Method and Master Theorem (主方法)

Solving recurrences: Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
  - Each node represents the cost of a single subproblem.

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



Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

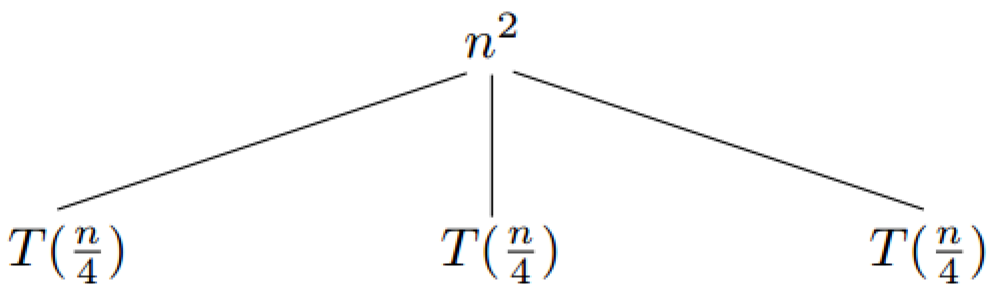
Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

$$T(n)$$

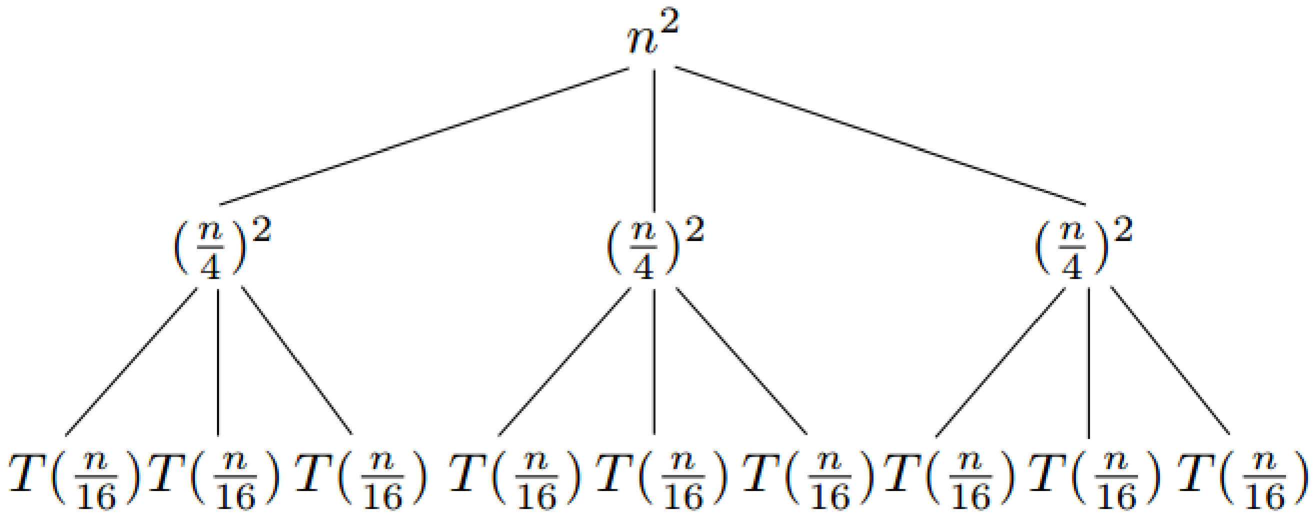
Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



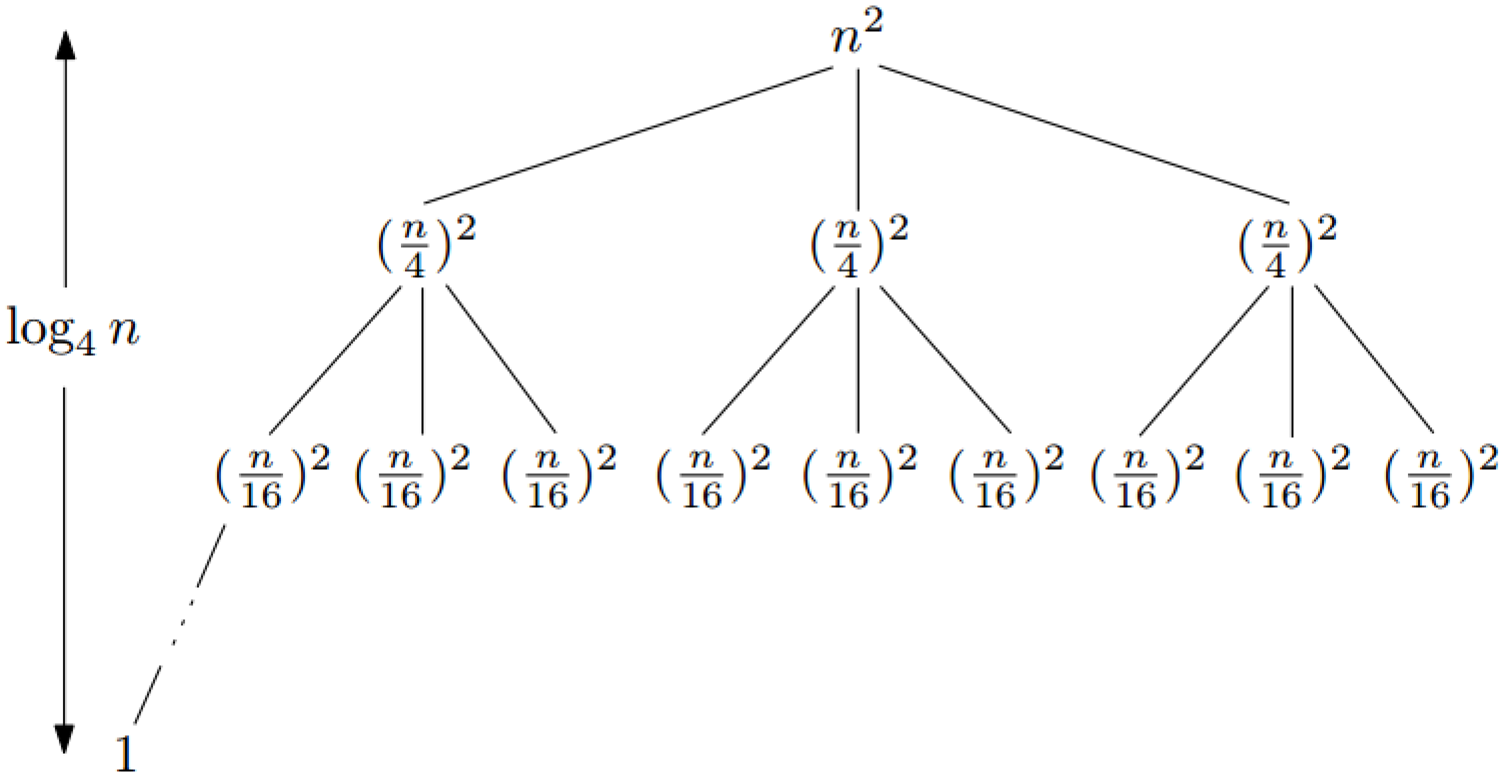
Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



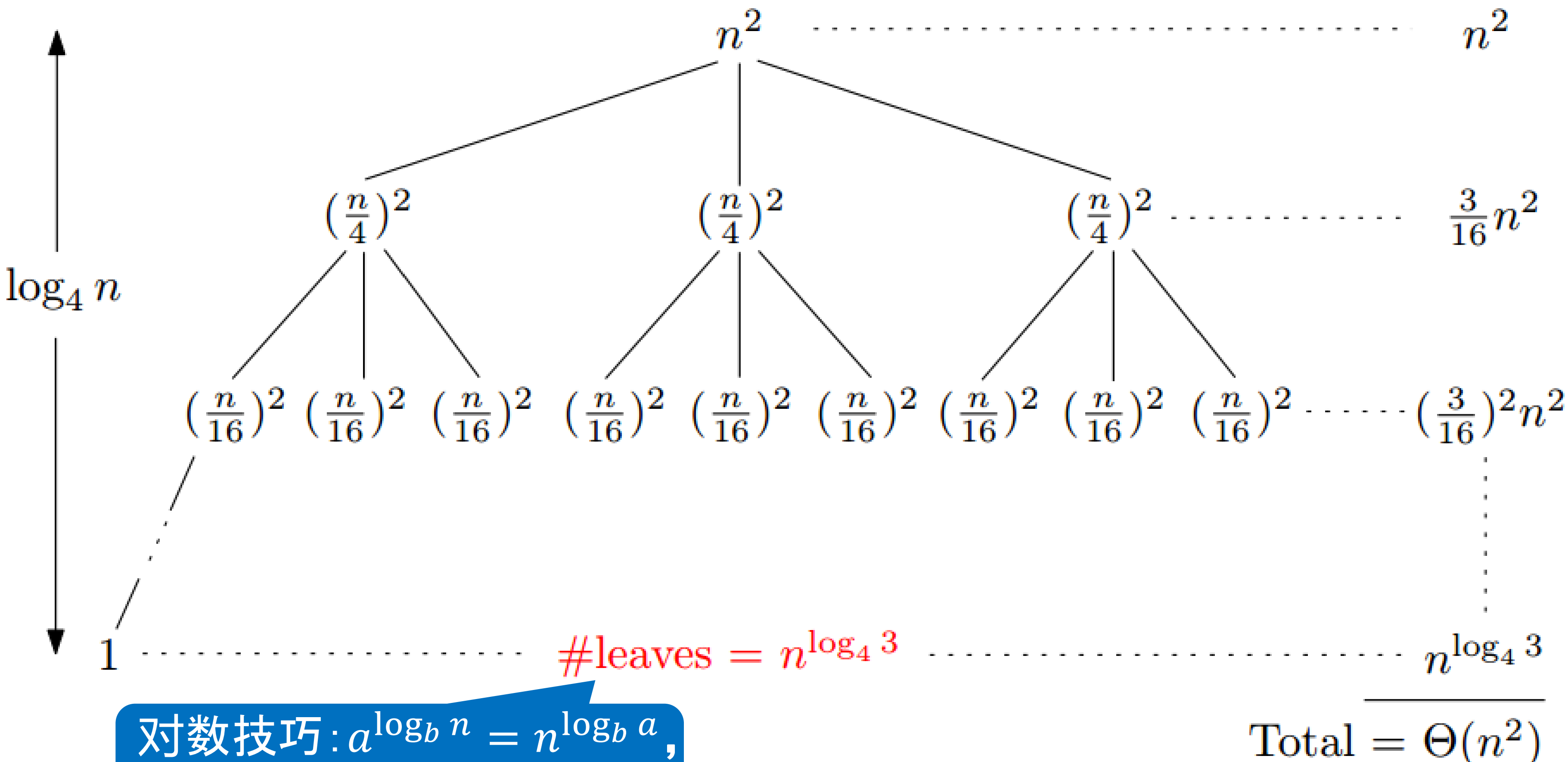
Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



对数技巧：$a^{\log_b n} = n^{\log_b a}$,
$3^{\log_4 n} = n^{\log_4 3}$

Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

$$\begin{aligned} T(n) &\le n^2 + \frac{3}{16}n^2 + \left(\frac{3}{16}\right)^2 n^2 + \cdots \\ &= O(n^2). \end{aligned}$$

geometric series

几何级数（又称为等比级数）

Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

$$\begin{aligned} T(n) &\le n^2 + \frac{3}{16}n^2 + \left(\frac{3}{16}\right)^2 n^2 + \cdots \\ &= O(n^2). \quad \text{geometric series} \end{aligned}$$

- Since $T(n) = 3T(n/4) + n^2$, it follows that $T(n) \geq n^2$

Recursion-tree method: Example

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

$$\begin{aligned} T(n) &\le n^2 + \frac{3}{16}n^2 + \left(\frac{3}{16}\right)^2 n^2 + \cdots \\ &= O(n^2). \quad \text{geometric series} \end{aligned}$$

- Since $T(n) = 3T(n/4) + n^2$, it follows that $T(n) \geq n^2$
- So, $T(n) = \Omega(n^2)$.
- Thus, $T(n) = \Theta(n^2)$

Outline

- Asymptotic Notations（渐近记号）
  - Big-Oh
  - Big-Omega
  - Big-Theta
  - Algorithm Design and Algorithm Turing
- Solving Recurrences
  - Recursion-tree Method（递归树法）
  - Substitution Method（代入法/替代法）
  - Master Method and Master Theorem (主方法)

Substitution method: Example 1

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Prove $T(n) \leq cn^2$ by induction, where c is a large constant.

Substitution method: Example 1

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Prove $T(n) \le cn^2$ by induction, where c is a large constant.

Proof.

- Base (n=1) : obviously holds for any $c \ge 1$

$\square$

Substitution method: Example 1

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Prove $T(n) \leq cn^2$ by induction, where c is a large constant.

Proof.

- Base (n=1) : obviously holds for any $c \geq 1$
- Induction:

$$\begin{aligned} T(n) &= 3T(n/4) + n^2 \\ &\leq 3c(n/4)^2 + n^2 \\ &= cn^2 - (13c/16 - 1)n^2 \\ &\leq cn^2, \end{aligned}$$

whenever $13c/16 - 1 \geq 0$, or $c \geq 16/13$.

□

Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$

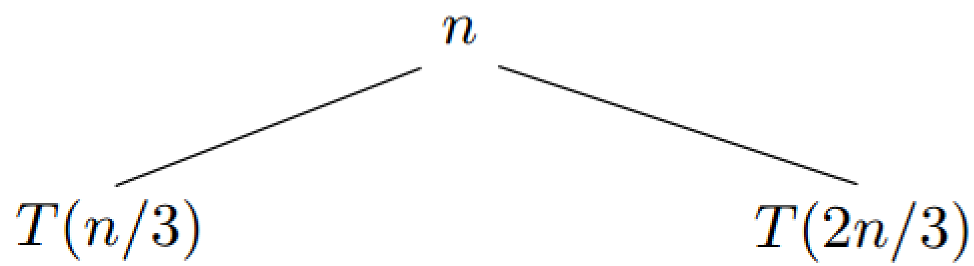
Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$

$$T(n)$$

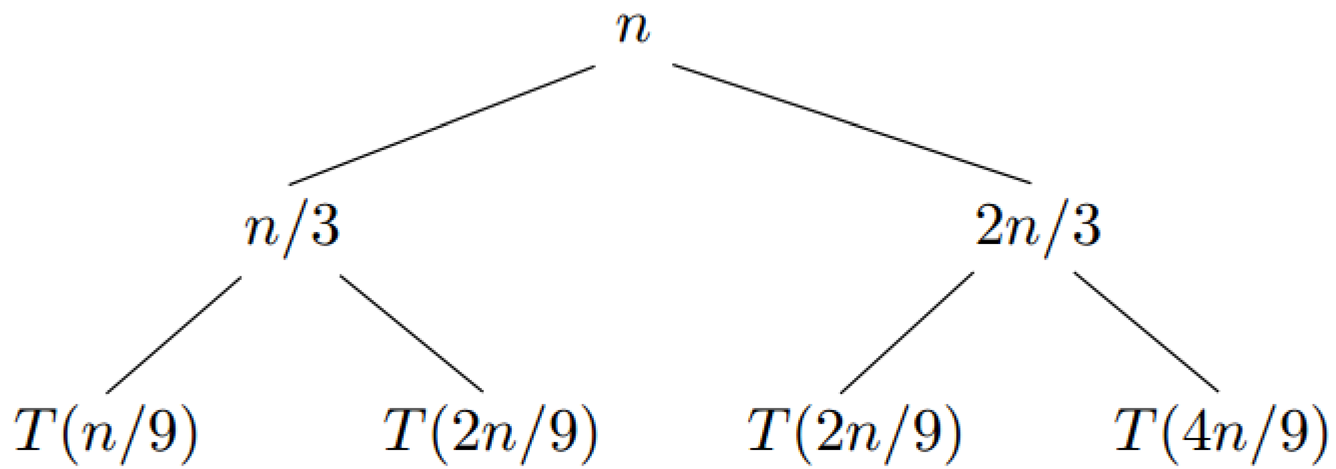
Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$



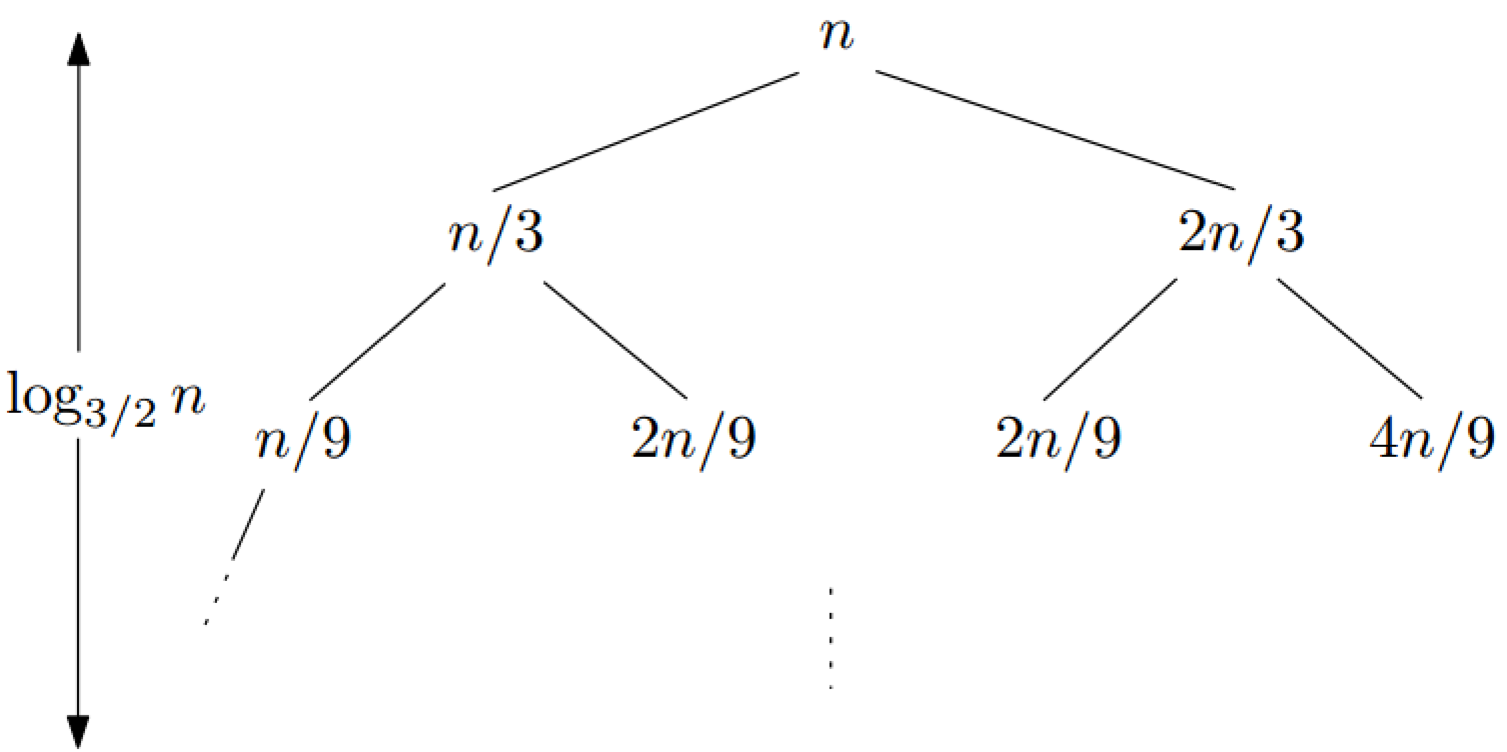
Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$



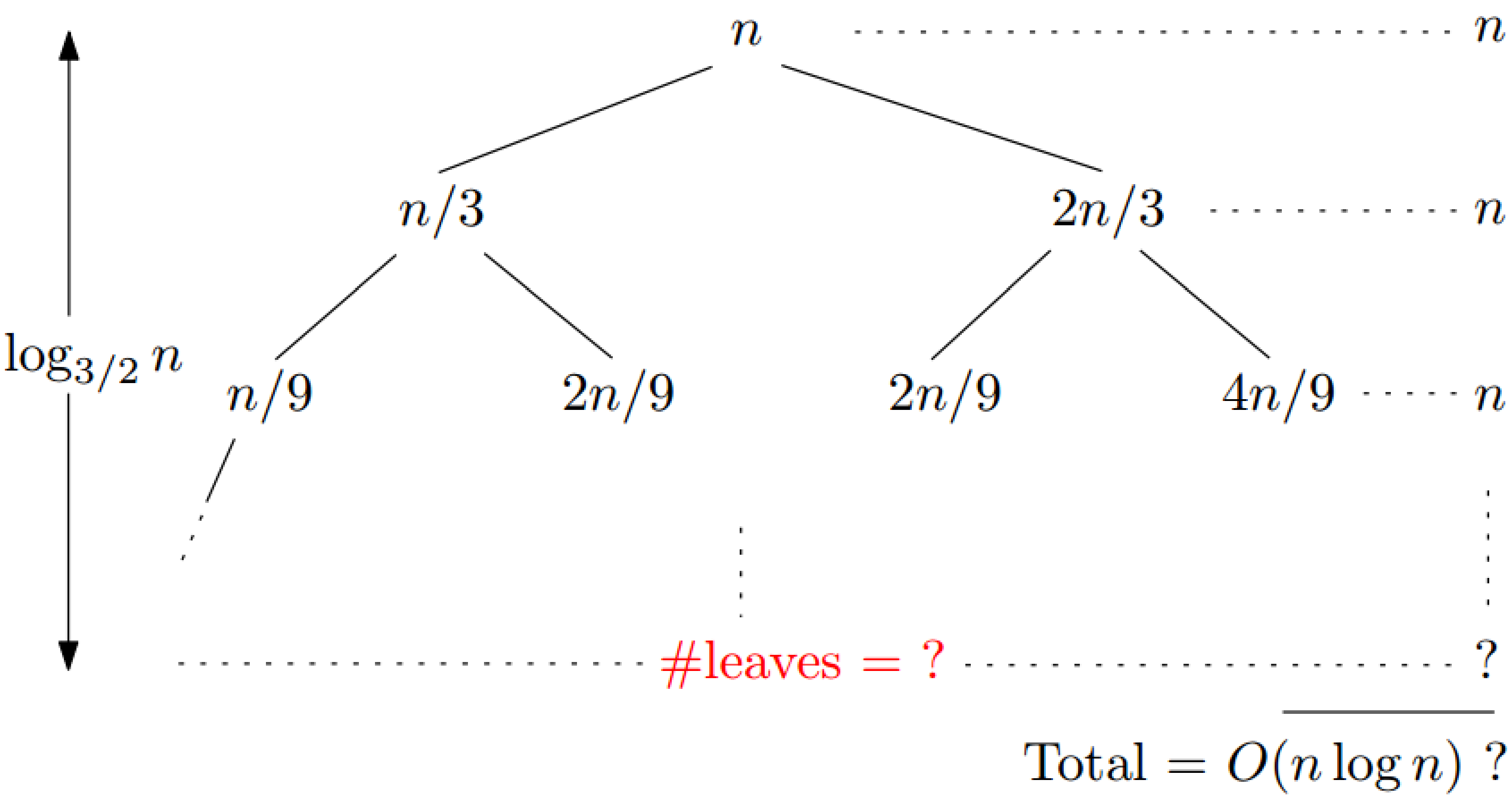
Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$



Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$



Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$

Prove $T(n) \leq cn \log n$ by induction, where c is a large constant.

Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$

Prove $T(n) \leq cn \log n$ by induction, where c is a large constant.

Proof.

- Base (n=2) : obviously holds for any $c \geq 1/2$

$\square$

Substitution method: Example 2

$$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & \text{if } n > 2, \\ 1, & \text{if } n = 1, 2. \end{cases}$$

Prove $T(n) \leq cn \log n$ by induction, where c is a large constant.

Proof.

- Base (n=2) : obviously holds for any $c \geq 1/2$
- Induction:

$$\begin{aligned} T(n) &= T(n/3) + T(2n/3) + n \\ &\leq c(n/3)\log(n/3) + c(2n/3)\log(2n/3) + n \\ &= cn\log n - c((n/3)\log 3 + (2n/3)\log(3/2)) + n \\ &= cn\log n - cn(\log 3 - 2/3) + n \\ &\leq cn\log n, \end{aligned}$$

as long as $c \geq 1/(\log 3 - 2/3)$.

$\square$

Outline

- Asymptotic Notations（渐近记号）
  - Big-Oh
  - Big-Omega
  - Big-Theta
  - Algorithm Design and Algorithm Turing
- Solving Recurrences
  - Recursion-tree Method（递归树法）
  - Substitution Method（代入法/替代法）
  - Master Method and Master Theorem (主方法)

Master Theorem

If $T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

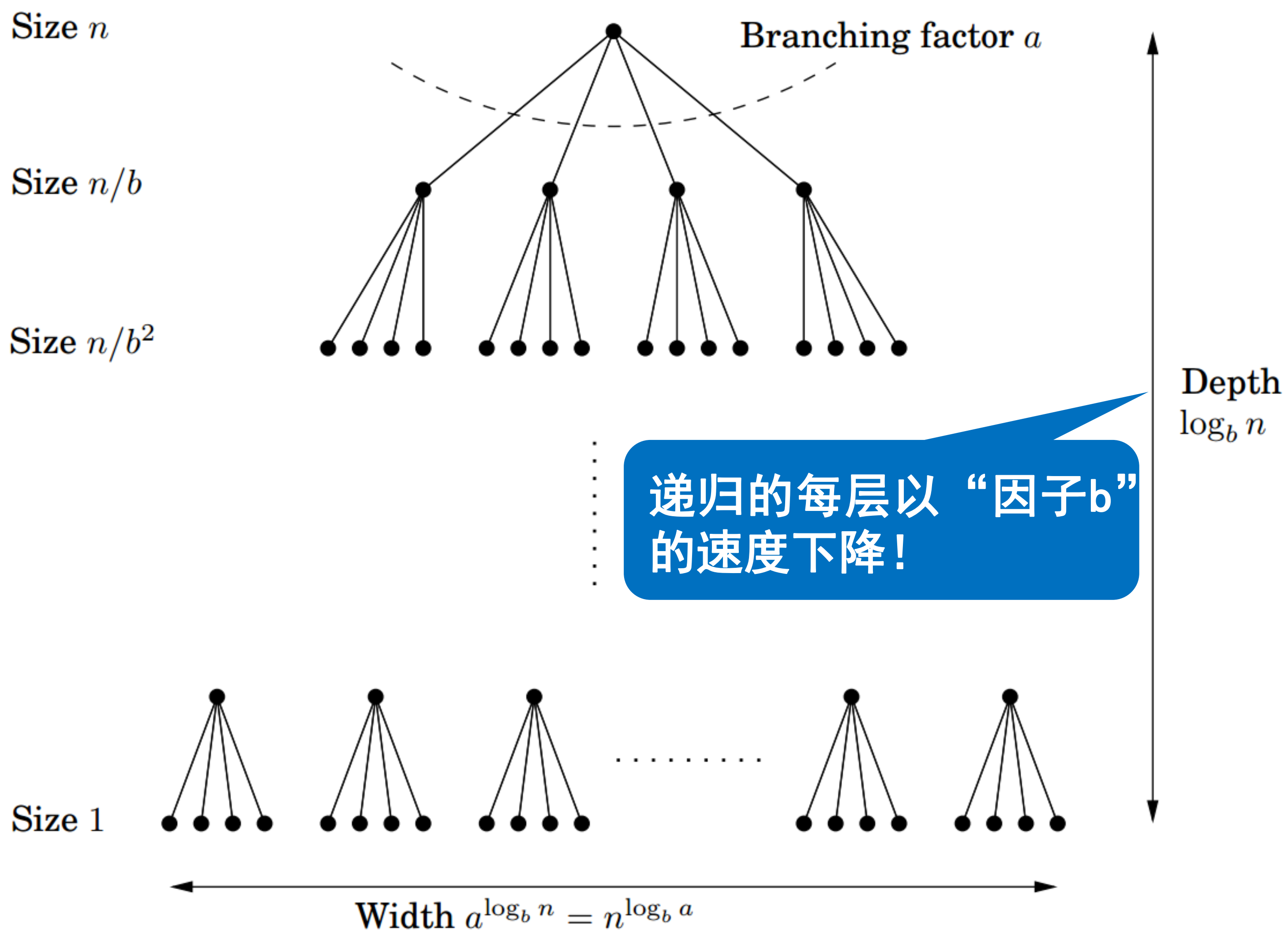
Proof of the Master Theorem

If $T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

For the sake of convenience, we assume that n is a power of b. This will not influence the final bound in any important way—n is at most a multiplicative factor of b away from some power of b—and it will allow us to ignore the rounding effect in $\lceil\frac{n}{b}\rceil$.

Proof of the Master Theorem



$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k$$

Proof of the Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

- The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree.
- The k-th level of the tree is made up of a^k subproblems, each of size n/b^k
- The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O\left(n^d\right) \times \left(\frac{a}{b^d}\right)^k$$

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 (a/b^d <1)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 (a/b^d <1)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- **The ratio a/b^d is greater than 1 (a/b^d >1)**.
  The series is increasing and its sum is given by its last term, $O\left(n^{\log_b a}\right)$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} =$$

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 (a/b^d <1)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- **The ratio a/b^d is greater than 1 (a/b^d >1)**.
  The series is increasing and its sum is given by its last term, $O\left(n^{\log_b a}\right)$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) =$$

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 (a/b^d <1)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- **The ratio a/b^d is greater than 1 (a/b^d >1)**.
  The series is increasing and its sum is given by its last term, $O\left(n^{\log_b a}\right)$:

$$n^d\left(\frac{a}{b^d}\right)^{\log_b n} = n^d\left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)}$$

$$=$$

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 ($a/b^d<1$)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- **The ratio a/b^d is greater than 1 ($a/b^d>1$)**.
  The series is increasing and its sum is given by its last term, $O\left(n^{\log_b a}\right)$:

$$n^d\left(\frac{a}{b^d}\right)^{\log_b n} = n^d\left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)}$$

$$= n^{\log_b a}$$

Proof of the Master theorem

It comes down to the following three cases.

- **The ratio a/b^d is less than 1 (a/b^d <1)**.
  Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- **The ratio a/b^d is greater than 1 (a/b^d >1)**.
  The series is increasing and its sum is given by its last term, $O\left(n^{\log_b a}\right)$:

$$n^d\left(\frac{a}{b^d}\right)^{\log_b n} = n^d\left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)}$$

$$= n^{\log_b a}$$

- **The ratio a/b^d is exactly 1 (a/b^d =1)**.
  In this case all O(log n) terms of the series are equal to $O(n^d)$.

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 1: $T(n) = 3T\left(\frac{n}{2}\right) + n$

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 1: $T(n) = 3T\left(\frac{n}{2}\right) + n$

- a = 3, b = 2, d = 1, d < $\log_b$ a

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 1: $T(n) = 3T\left(\frac{n}{2}\right) + n$

- a = 3, b = 2, d = 1, d < $\log_b$ a
- $T(n) = O\left(n^{\log_b a}\right) = O(n^{\log 3})$

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 2: $T(n) = 3T\left(\frac{n}{4}\right) + n^5$

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 2: $T(n) = 3T\left(\frac{n}{4}\right) + n^5$

- a = 3, b = 4, d = 5, d > $\log_b$ a

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 2: $T(n) = 3T\left(\frac{n}{4}\right) + n^5$

- a = 3, b = 4, d = 5, d > $\log_b$ a
- $T(n) = O\left(n^d\right) = O(n^5)$

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 3: $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 3: $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

- a = 4, b = 2, d = 2, d = $\log_b$ a

Master theorem: Example

If $T(n) = aT\left(\lceil\frac{n}{b}\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Example 3: $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

- a = 4, b = 2, d = 2, d = $\log_b$ a
- $T(n) = O\left(n^d \log n\right) = O(n^2 \log n)$

Part I: Divide and Conquer

Maximum Contiguous Subarray Problem and Counting Inversion Problem

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Introduction to Part I

- Divide-and-conquer (D&C) is an important algorithm design paradigm.

Introduction to Part I

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**
    Dividing a given problem into two or more subproblems (ideally of approximately equal size)

Introduction to Part I

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)
  - **Conquer**

    Solving each subproblem (directly if small enough or recursively)

Introduction to Part I

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**
    Dividing a given problem into two or more subproblems (ideally of approximately equal size)
  - **Conquer**
    Solving each subproblem (directly if small enough or recursively)
  - **Combine**
    Combining the solutions of the subproblems into a global solution

Introduction to Part I

- In Part I, we will illustrate Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Integer Multiplication (整数乘法)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)
  - Deterministic and Randomized Selection (确定性与随机化选择)

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

[1] A Company that Makes Everything

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

Between years 1 and 9:

- ACME earned $-3+2+1-4+5+2-1+3-1=4$ M$

[1] A Company that Makes Everything

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

Between years 1 and 9:

- ACME earned $-3+2+1-4+5+2-1+3-1=4$ M$

Between years 2 and 6:

- ACME earned $2+1-4+5+2=6$ M$

[1] A Company that Makes Everything

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

Between years 1 and 9:

- ACME earned $-3+2+1-4+5+2-1+3-1=4$ M$

Between years 2 and 6:

- ACME earned $2+1-4+5+2=6$ M$

Between years 5 and 8:

- ACME earned $5+2-1+3=9$ M$

[1] A Company that Makes Everything

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

Between years 1 and 9:

- ACME earned $-3+2+1-4+5+2-1+3-1=4$ M$

Between years 2 and 6:

- ACME earned $2+1-4+5+2=6$ M$

Between years 5 and 8:

- ACME earned $5+2-1+3=9$ M$

Problem: Find the span of years in which ACME earned the most

[1] A Company that Makes Everything

Maximum Contiguous Subarray (MCS) Problem

ACME Corp[1] – PROFIT HISTORY

Year	1	2	3	4	5	6	7	8	9
Profit M$	-3	2	1	-4	5	2	-1	3	-1

Between years 1 and 9:

- ACME earned $-3+2+1-4+5+2-1+3-1=4$ M$

Between years 2 and 6:

- ACME earned $2+1-4+5+2=6$ M$

Between years 5 and 8:

- ACME earned $5+2-1+3=9$ M$

如果所有数组元素都是非负数，整个数组和肯定是最大

Problem: Find the span of years in which ACME earned the most

Answer: Year 5-8, 9 M$

[1] A Company that Makes Everything

Formal Definition

- Input: An array of reals $A[1\ldots n]$

Formal Definition

- Input: An array of reals *A*[1...n]
- The value of subarray *A*[i...j] is

$$V(i,j) = \sum_{x=i}^{j} A(x)$$

Formal Definition

- Input: An array of reals $A[1...n]$
- The value of subarray $A[i...j]$ is

$$V(i,j) = \sum_{x=i}^{j} A(x)$$

Definition (Maximum Contiguous Subarray Problem)

Find $i \leq j$ such that $V(i,j)$ is maximized.

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

A Brute Force Algorithm

Calculate the value of $V(i, j)$ for each pair $i \leq j$ and return the maximum value

A Brute Force Algorithm

Calculate the value of $V(i, j)$ for each pair $i \leq j$ and return the maximum value

$\text{VMAX} \leftarrow A[1]$;

A Brute Force Algorithm

Calculate the value of $V(i,j)$ for each pair $i \leq j$ and return the maximum value

```
VMAX ← A[1];
for i ← 1 to n do
    for j ← i to n do
        // calculate V(i,j)
        V ← 0;
        for x ← i to j do
            V ← V + A[x];
        end
```

A Brute Force Algorithm

Calculate the value of $V(i,j)$ for each pair $i \le j$ and return the maximum value

```
VMAX ← A[1];
for i ← 1 to n do
    for j ← i to n do
        // calculate V(i,j)
        V ← 0;
        for x ← i to j do
            V ← V + A[x];
        end
        if V > VMAX then
            VMAX ← V;
        end
    end
end
return VMAX
```

A Brute Force Algorithm

Calculate the value of $V(i,j)$ for each pair $i \le j$ and return the maximum value

```
VMAX ← A[1];
for i ← 1 to n do
    for j ← i to n do
        // calculate V(i,j)
        V ← 0;
        for x ← i to j do
            V ← V + A[x];
        end
        if V > VMAX then
            VMAX ← V;
        end
    end
end
return VMAX
```

$O(n^3)$ arithmetic additions

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

A Data-Reuse Algorithm

Idea:

- don't need to calculate each V(*i, j*) from scratch

A Data-Reuse Algorithm

Idea:

- don't need to calculate each V(*i, j*) from scratch
- exploit the fact: $V(i,j) = \sum_{x=i}^{j} A[x] = V(i,j-1) + A[j]$

A Data-Reuse Algorithm

Idea:

- don't need to calculate each V(*i, j*) from scratch
- exploit the fact: $V(i,j) = \sum_{x=i}^{j} A[x] = V(i,j-1) + A[j]$

```
VMAX ← A[1];
for i ← 1 to n do
    V ← 0;
    for j ← i to n do
        // calculate V(i,j)
        V ← V + A[j];
```

A Data-Reuse Algorithm

Idea:

- don't need to calculate each V(*i, j*) from scratch
- exploit the fact: $V(i,j) = \sum_{x=i}^{j} A[x] = V(i,j-1) + A[j]$

```
VMAX ← A[1];
for i ← 1 to n do
    V ← 0;
    for j ← i to n do
        // calculate V(i,j)
        V ← V + A[j];
        if V > VMAX then
            VMAX ← V;
        end
    end
end
return VMAX
```

A Data-Reuse Algorithm

Idea:

- don't need to calculate each V(*i*, *j*) from scratch
- exploit the fact: $V(i,j) = \sum_{x=i}^{j} A[x] = V(i,j-1) + A[j]$

```
VMAX ← A[1];
for i ← 1 to n do
    V ← 0;
    for j ← i to n do
        // calculate V(i,j)
        V ← V + A[j];
        if V > VMAX then
            VMAX ← V;
        end
    end
end
return VMAX
```

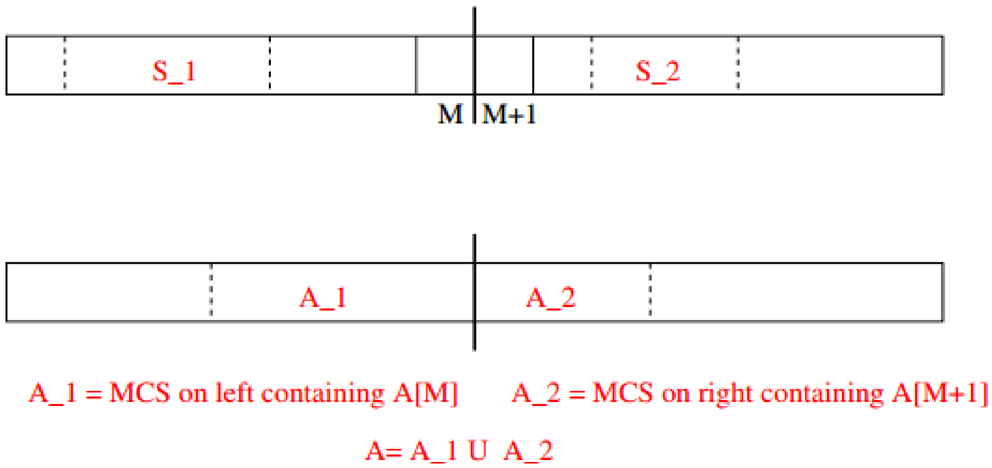
$O(n^2)$ arithmetic additions

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

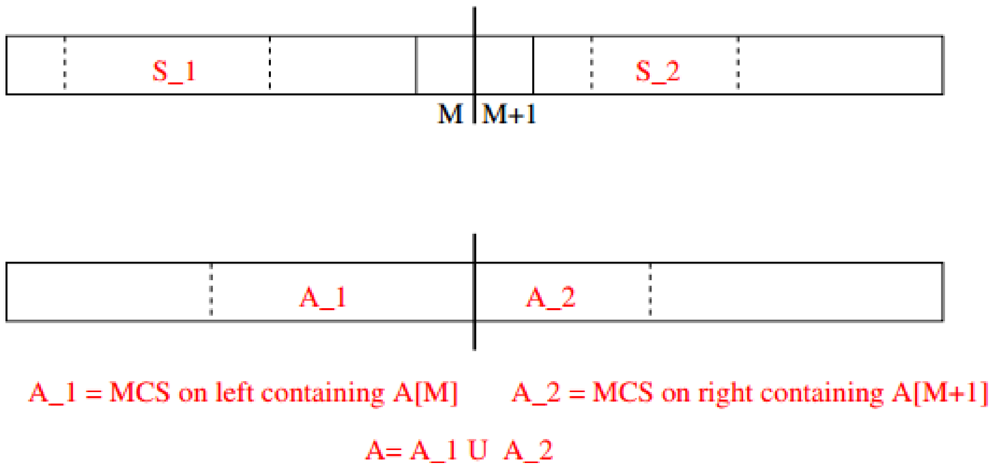
A Divide-and-Conquer Algorithm

Set $m = \lfloor (n+1)/2 \rfloor$



A Divide-and-Conquer Algorithm

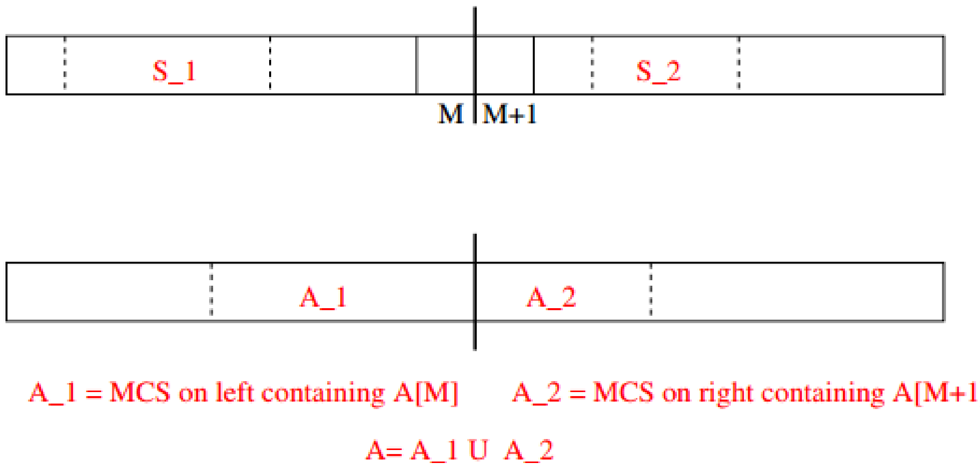
Set $m = \lfloor (n+1)/2 \rfloor$



The MCS S must be one of

A Divide-and-Conquer Algorithm

Set $m = \lfloor (n+1)/2 \rfloor$

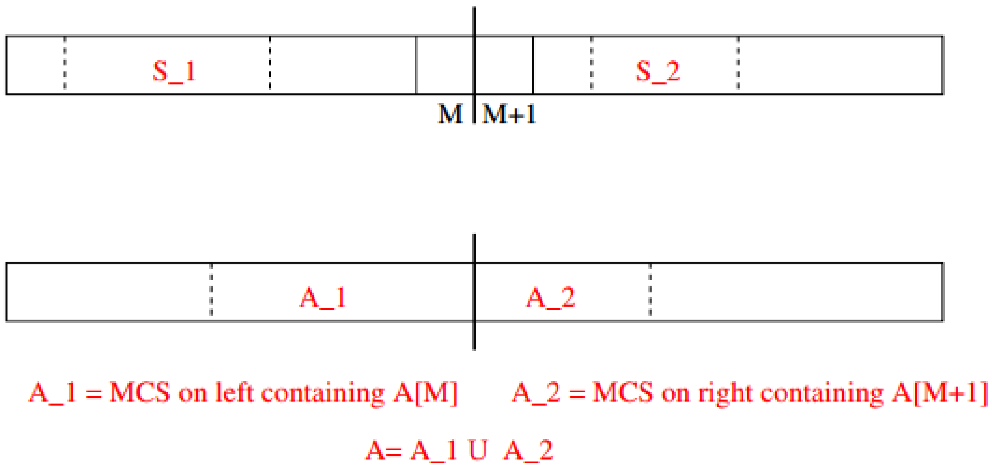


The MCS S must be one of

1. S_1: the MCS in $A[1 \ldots m]$

A Divide-and-Conquer Algorithm

Set $m = \lfloor (n+1)/2 \rfloor$

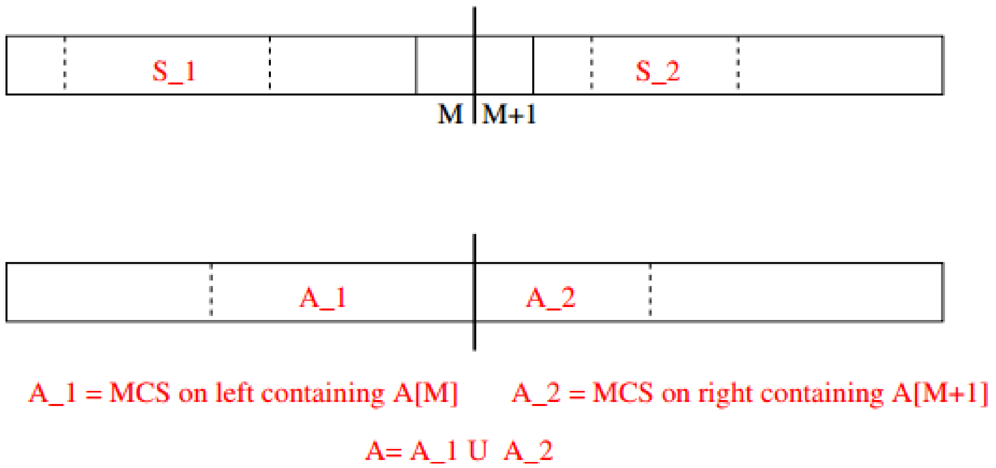


The MCS S must be one of

1. S_1: the MCS in $A[1 \ldots m]$
2. S_2: the MCS in $A[m+1 \ldots n]$

A Divide-and-Conquer Algorithm

Set $m = \lfloor (n+1)/2 \rfloor$

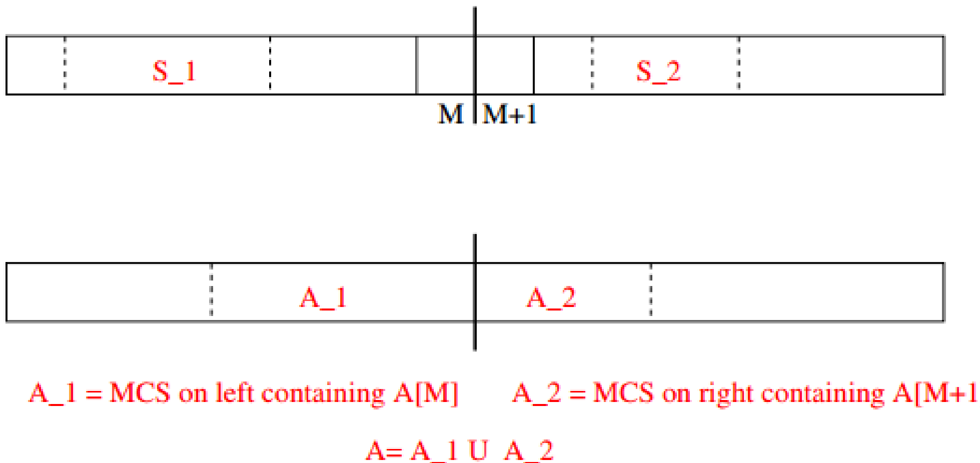


The MCS S must be one of

1. S_1: the MCS in $A[1 \ldots m]$
2. S_2: the MCS in $A[m+1 \ldots n]$
3. A: the MCS across the cut.

A Divide-and-Conquer Algorithm

Set $m = \lfloor (n+1)/2 \rfloor$



The MCS S must be one of

1. S_1: the MCS in $A[1 \dots m]$
2. S_2: the MCS in $A[m+1 \dots n]$
3. A: the MCS across the cut.

So,

最终，在S_1，S_2和A（跨越中点的最大子数组）这三种情况中选取和最大者

$$S = \text{the best among } \{S_1, S_2, A\}$$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 =$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 =$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $A_1 =$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $A_1 = [3, 6, -1]$ and $A_2 =$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

- $A_1 = [3, 6, -1]$ and $A_2 = [2, -4, 7]$
- $A = A_1 \cup A_2 =$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

- $A_1 = [3, 6, -1]$ and $A_2 = [2, -4, 7]$
- $A = A_1 \cup A_2 = [3, 6, -1, 2, -4, 7]$

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $A_1 = [3, 6, -1]$ and $A_2 = [2, -4, 7]$
- $A = A_1 \cup A_2 = [3, 6, -1, 2, -4, 7]$

- $Value(S_1) = 9;\ Value(S_2) = 9;\ Value(A) = 13$
- solution:

An Example of Divide-and-Conquer Algorithm

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $S_1 = [3, 6]$ and $S_2 = [2, 6, 1]$

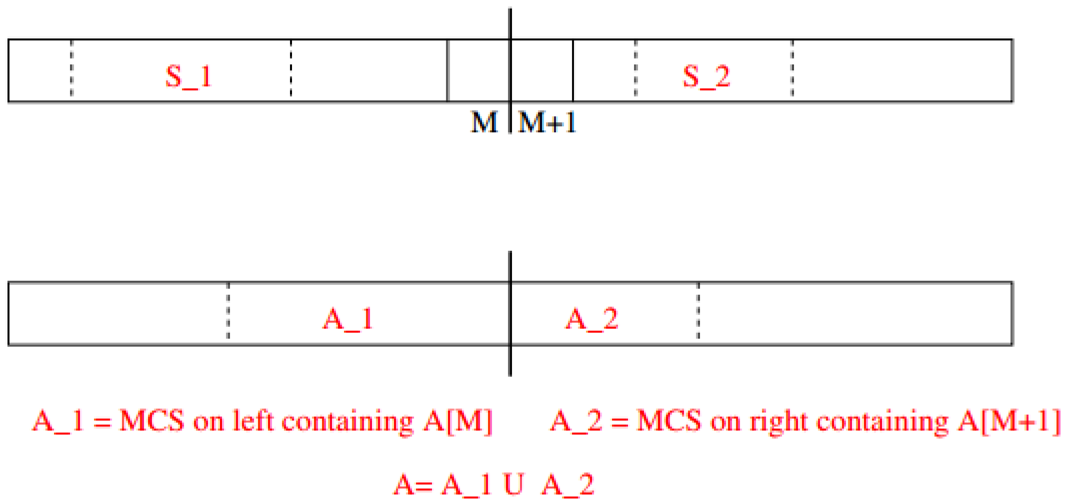
1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

- $A_1 = [3, 6, -1]$ and $A_2 = [2, -4, 7]$
- $A = A_1 \cup A_2 = [3, 6, -1, 2, -4, 7]$

- $Value(S_1) = 9;\ Value(S_2) = 9;\ Value(A) = 13$
- solution: A

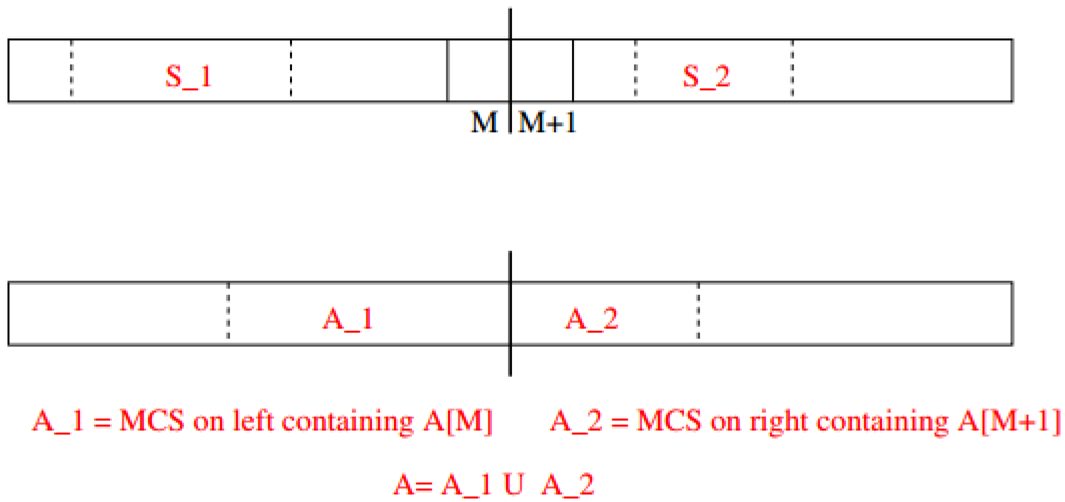
Divide: MCS across The Cut

Set $m = \lfloor (n+1)/2 \rfloor$



Divide: MCS across The Cut

Set $m = \lfloor (n+1)/2 \rfloor$

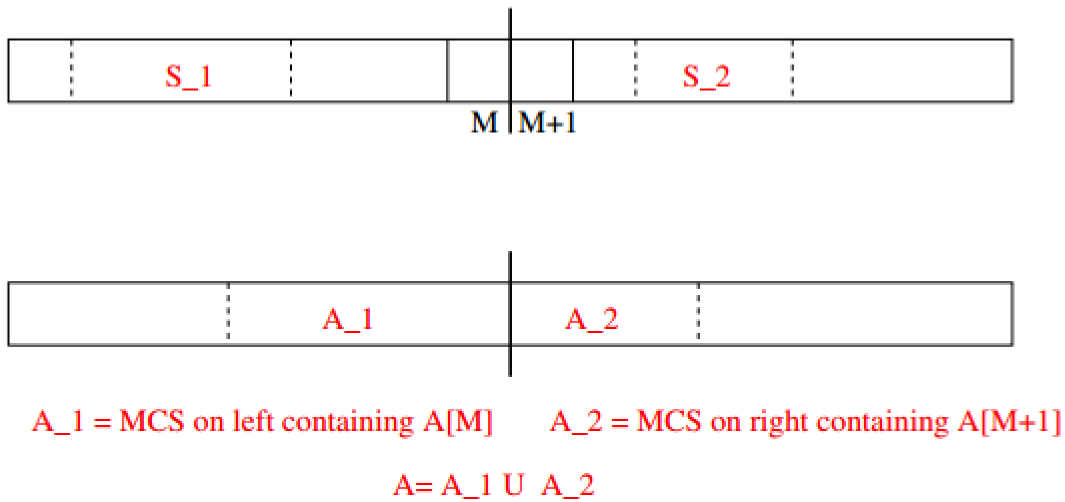


$A = A_1 \cup A_2$

- A_1: MCS among contiguous subarrays ending at A[m]

Divide: MCS across The Cut

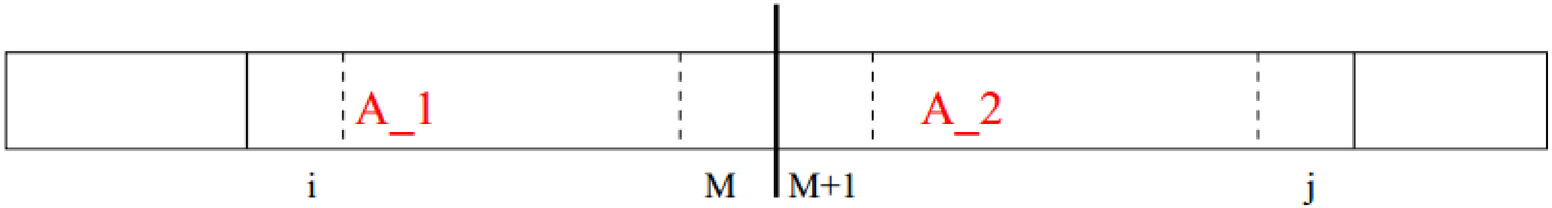
Set $m = \lfloor (n+1)/2 \rfloor$



$A = A_1 \cup A_2$

- A_1: MCS among contiguous subarrays ending at A[m]
- A_2: MCS among contiguous subarrays starting at A[m+1]

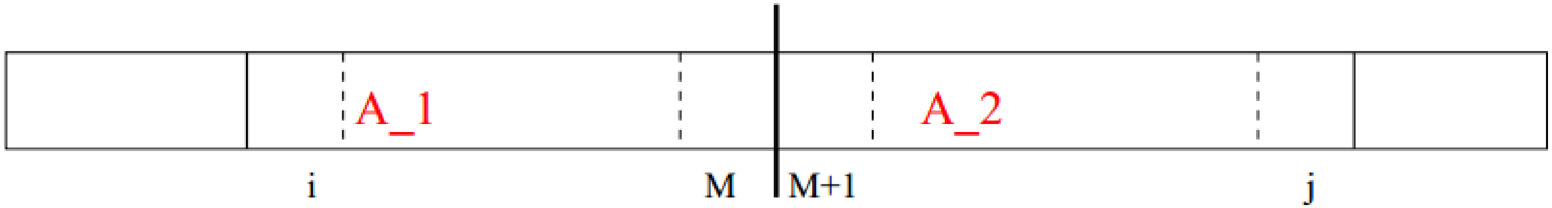
Conquer: Finding the "A_1" Subarrays



A_1 is in the form $A[i \ldots m]$, $V(i, m) = V(i+1, m) + A[i]$

MAX $\leftarrow A[m]$;
SUM $\leftarrow A[m]$;

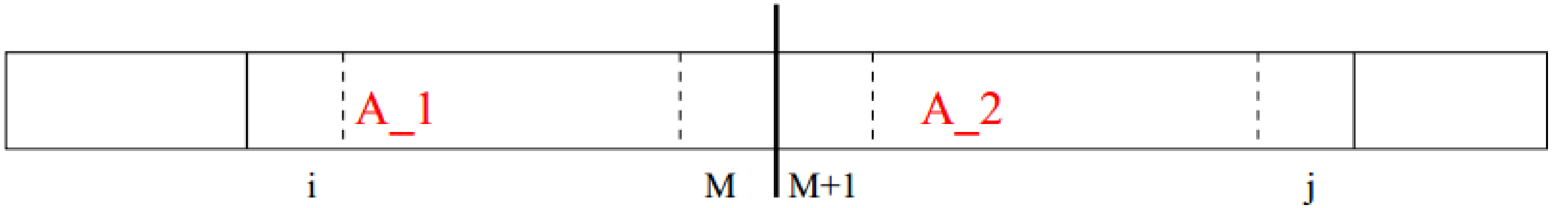
Conquer: Finding the "A_1" Subarrays



A_1 is in the form $A[i \ldots m]$, $V(i, m) = V(i+1, m) + A[i]$

MAX $\leftarrow A[m]$;
SUM $\leftarrow A[m]$;
for $i \leftarrow m-1$ **downto** *1* **do**
    SUM $\leftarrow$ SUM $+A[i]$;

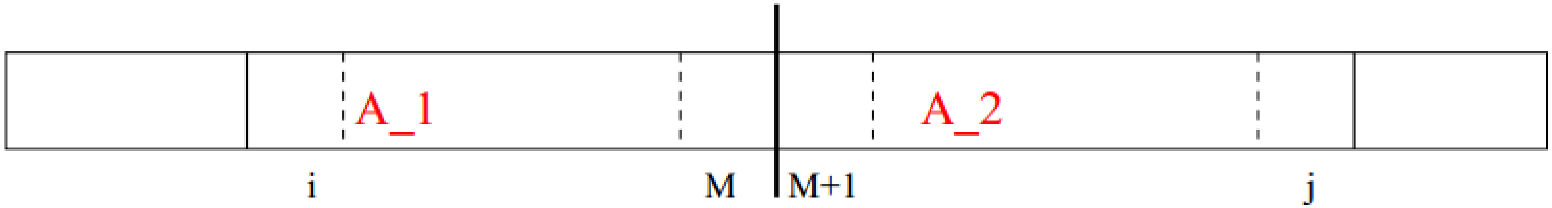
Conquer: Finding the "A_1" Subarrays



A_1 is in the form $A[i \ldots m]$, $V(i, m) = V(i+1, m) + A[i]$

MAX $\leftarrow A[m]$;
SUM $\leftarrow A[m]$;
for $i \leftarrow m - 1$ **downto** *1* **do**
    SUM $\leftarrow$ SUM $+A[i]$;
    if *SUM* $>$ *MAX* **then**
        MAX $\leftarrow$ SUM;
    end
end

Conquer: Finding the "A_1" Subarrays



A_1 is in the form $A[i \ldots m]$, $V(i, m) = V(i+1, m) + A[i]$

MAX $\leftarrow A[m]$;
SUM $\leftarrow A[m]$;
for $i \leftarrow m - 1$ **downto** *1* **do**
    SUM $\leftarrow$ SUM $+A[i]$;
    if *SUM* $>$ *MAX* **then**
        MAX $\leftarrow$ SUM;
    end
end
A_1=MAX;

Conquer: Finding "A" with A Linear Time

- There are only m sequences of the form

Conquer: Finding "*A*" with A Linear Time

- There are only *m* sequences of the form
  - A_1 can be found in $O(m)$ time

Conquer: Finding "A" with A Linear Time

- There are only m sequences of the form
  - A_1 can be found in $O(m)$ time

- Similarly, A_2 is in the form $A[m+1...j]$

Conquer: Finding "*A*" with A Linear Time

- There are only *m* sequences of the form
  - A_1 can be found in $O(m)$ time

- Similarly, A_2 is in the form $A[m+1 \ldots j]$
  - there are only $n-m$ such sequences

Conquer: Finding "A" with A Linear Time

- There are only m sequences of the form
  - A_1 can be found in $O(m)$ time
- Similarly, A_2 is in the form $A[m+1 \ldots j]$
  - there are only $n-m$ such sequences
  - A_2 can be found in $O(n-m)$ time

Conquer: Finding "*A*" with A Linear Time

- There are only m sequences of the form
  - A_1 can be found in $O(m)$ time
- Similarly, A_2 is in the form $A[m+1 \ldots j]$
  - there are only $n-m$ such sequences
  - A_2 can be found in $O(n-m)$ time
- $A = A_1 \cup A_2$ can be found in $O(n)$ time

Conquer: Finding "A" with A Linear Time

- There are only m sequences of the form
  - A_1 can be found in $O(m)$ time
- Similarly, A_2 is in the form $A[m+1 \ldots j]$
  - there are only $n-m$ such sequences
  - A_2 can be found in $O(n-m)$ time
- $A = A_1 \cup A_2$ can be found in $O(n)$ time
  - linear to the input size

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \ldots t]$ with $s \leq t$
Output: MCS of $A[s \ldots t]$

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \dots t]$ with $s \leq t$
Output: MCS of $A[s \dots t]$
begin
  if $s = t$ **then return** $A[s]$;

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \ldots t]$ with $s \leq t$
Output: MCS of $A[s \ldots t]$
begin
  if $s = t$ **then return** $A[s]$;
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$;

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \ldots t]$ with $s \leq t$
Output: MCS of $A[s \ldots t]$
begin
  if $s = t$ **then return** $A[s]$;

  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$;
    Find $MCS(A, m+1, t)$;

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \dots t]$ with $s \leq t$
Output: MCS of $A[s \dots t]$
begin
  if $s = t$ **then return** $A[s]$;
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$;
    Find $MCS(A, m+1, t)$;
    Find MCS that contains both $A[m]$ and $A[m+1]$;

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \dots t]$ with $s \le t$
Output: MCS of $A[s \dots t]$
begin
  if $s = t$ **then return** $A[s]$;

  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$;
    Find $MCS(A, m+1, t)$;
    Find MCS that contains both $A[m]$ and $A[m+1]$;
    return maximum of the three sequences found
  end

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \dots t]$ with $s \le t$
Output: MCS of $A[s \dots t]$
begin
    if $s = t$ **then return** $A[s]$;
    else
        $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
        Find $MCS(A, s, m)$;
        Find $MCS(A, m+1, t)$;
        Find MCS that contains both $A[m]$ and $A[m+1]$;
        return maximum of the three sequences found
    end
end

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

Input: $A[s \dots t]$ with $s \leq t$
Output: MCS of $A[s \dots t]$
begin
    if $s = t$ **then return** $A[s]$;
    else
        $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
        Find $MCS(A, s, m)$;
        Find $MCS(A, m+1, t)$;
        Find MCS that contains both $A[m]$ and $A[m+1]$;
        return maximum of the three sequences found
    end
end

First Call: *MCS*(*A*, *1*, *n*)

A Full Illustration of the D&C Algorithm

6 -4 7 -4 0 1

A Full Illustration of the D&C Algorithm

6 -4 7 | -4 0 1

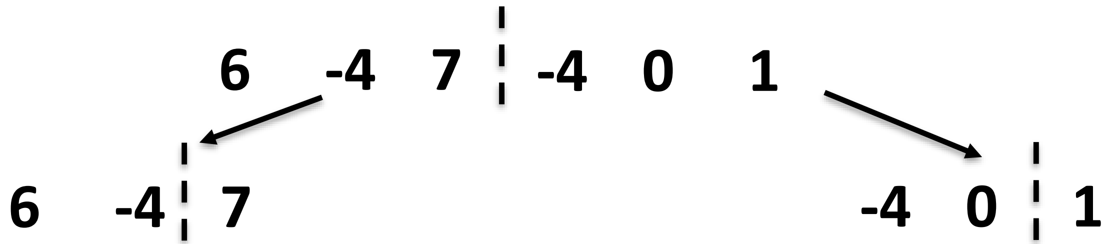
A Full Illustration of the D&C Algorithm

6 -4 7 | -4 0 1

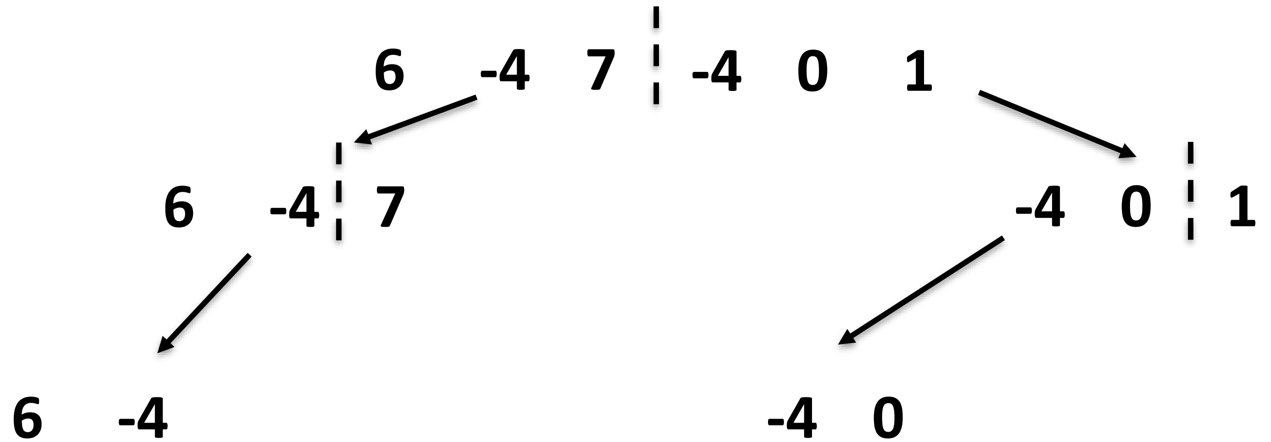
6 -4 7

-4 0 1

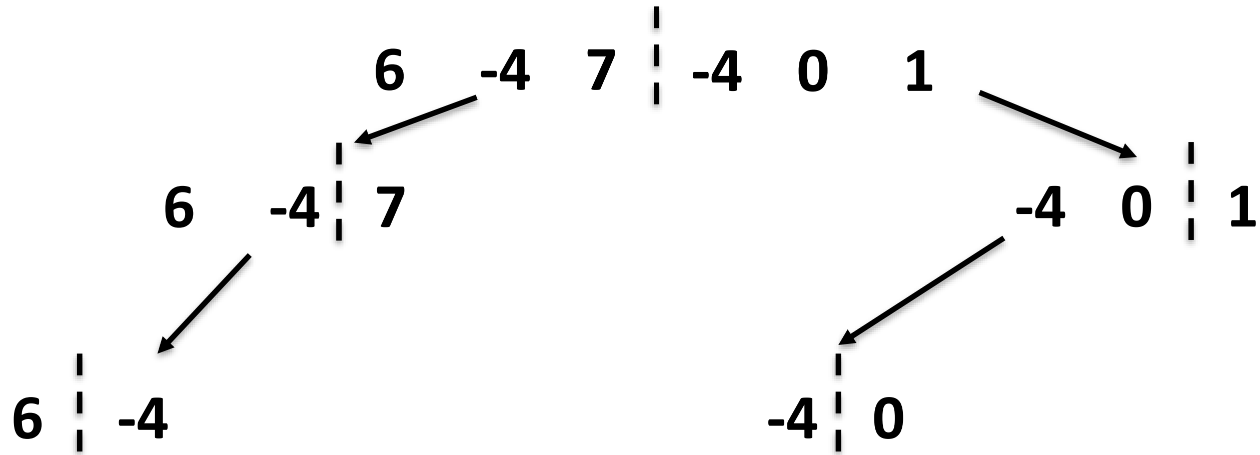
A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm

6 -4 7 | -4 0 1

6 -4 | 7

-4 0 | 1

6 | -4

-4 | 0

Divide

A Full Illustration of the D&C Algorithm

6 -4 7 ¦ -4 0 1

6 -4 ¦ 7

-4 0 ¦ 1

6 ¦ -4

-4 ¦ 0

Divide

Conquer

A Full Illustration of the D&C Algorithm

6 -4 7 ¦ -4 0 1

6 -4 ¦ 7 **-4 0 ¦ 1**

6 ¦ -4 **-4 ¦ 0**

Divide

MCS={6} **MCS={-4}**

Conquer

A Full Illustration of the D&C Algorithm

6 -4 7 ¦ -4 0 1

6 -4 ¦ 7

-4 0 ¦ 1

6 ¦ -4

-4 ¦ 0

Divide

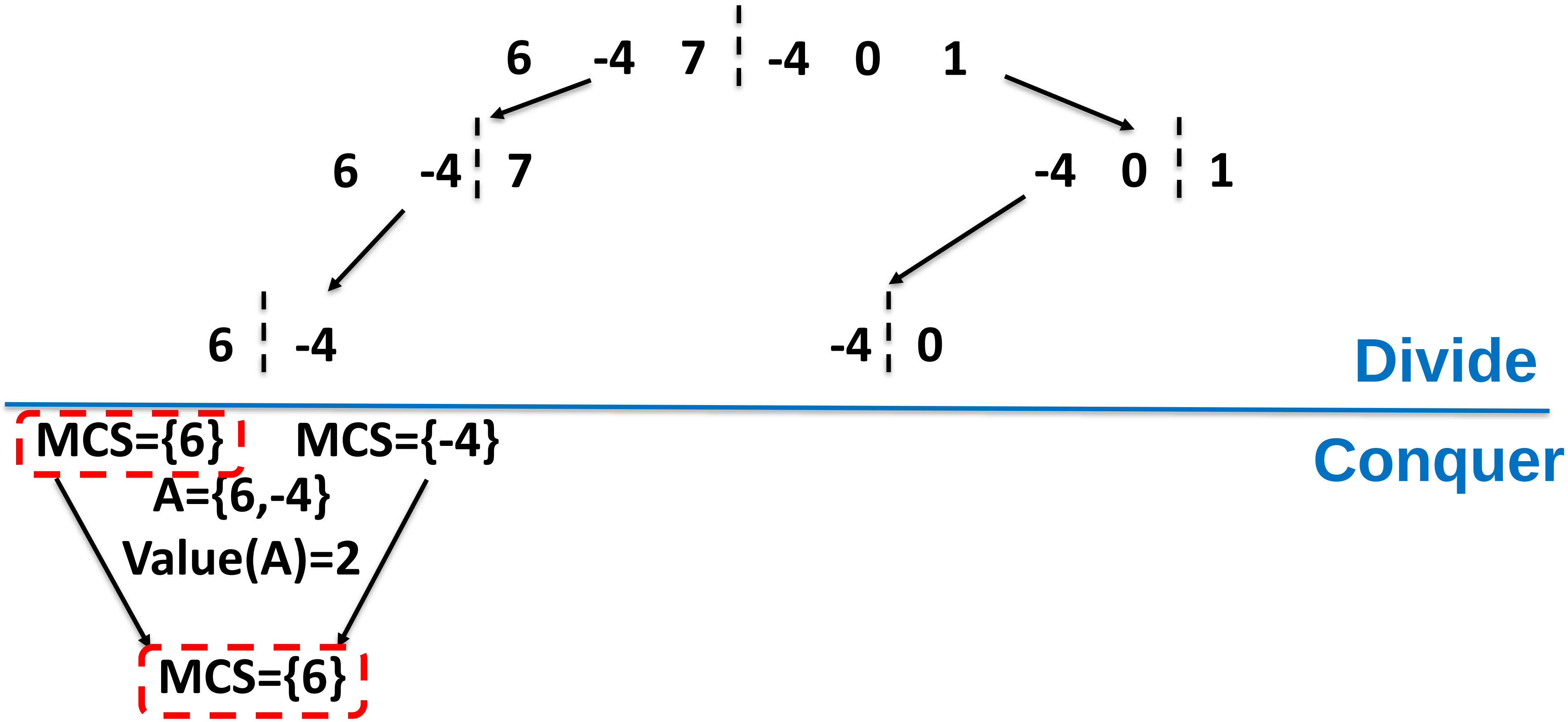
Conquer

MCS={6} **MCS={-4}**

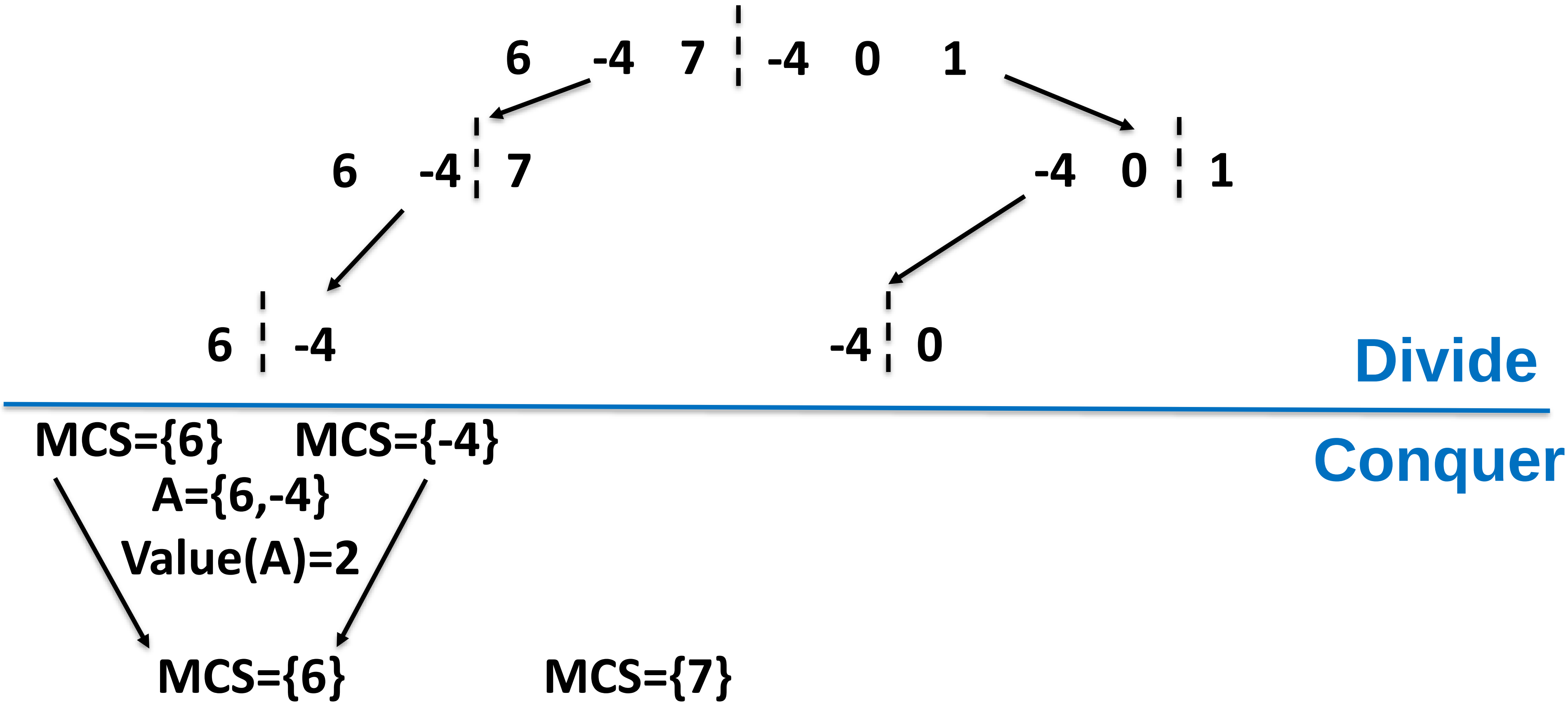
A={6,-4}

Value(A)=2

A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm

6 -4 7 | -4 0 1

6 -4 | 7

-4 0 | 1

6 | -4

-4 | 0

Divide

Conquer

MCS={6} **MCS={-4}**

A={6,-4}

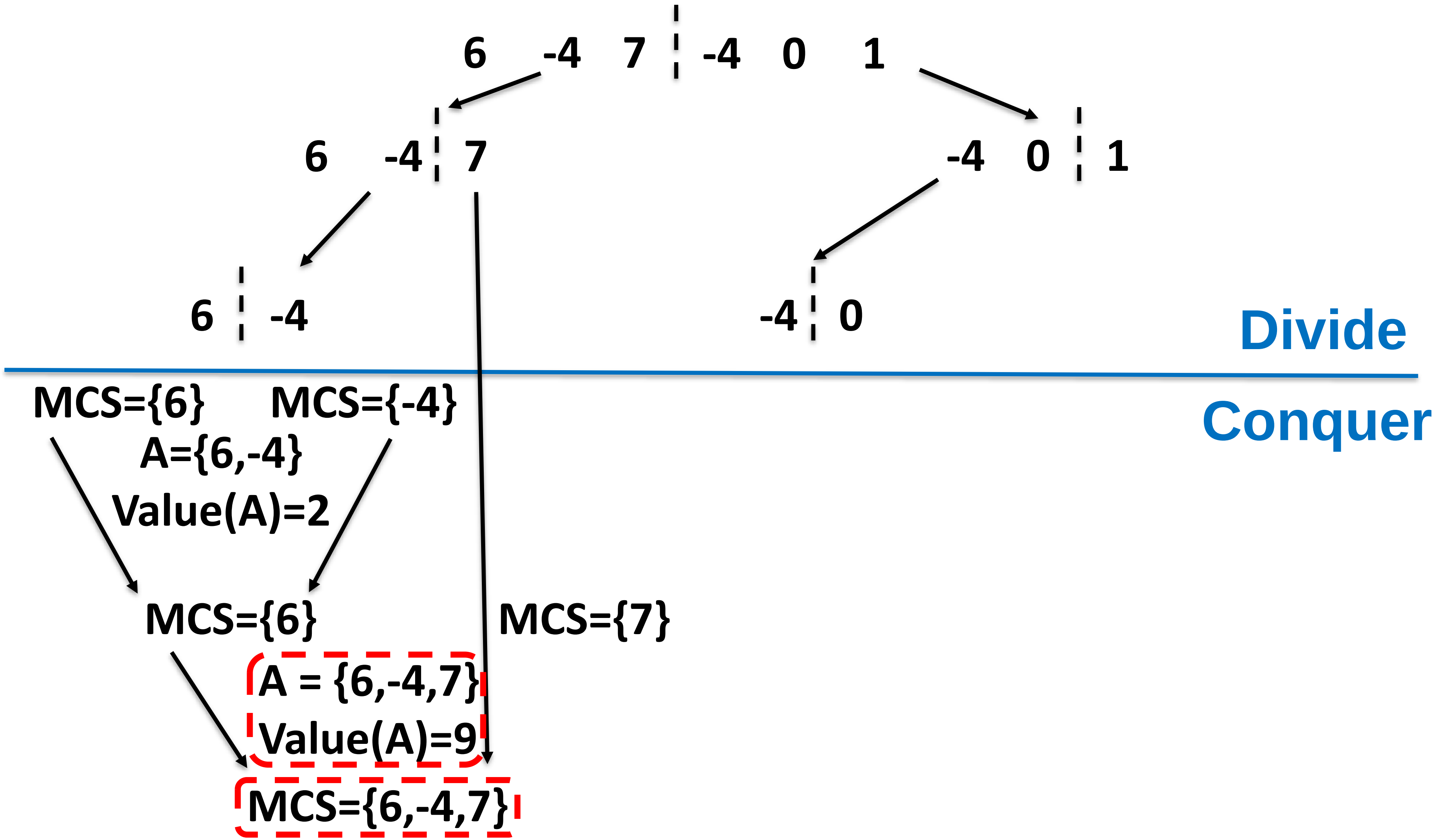
Value(A)=2

MCS={6} **MCS={7}**

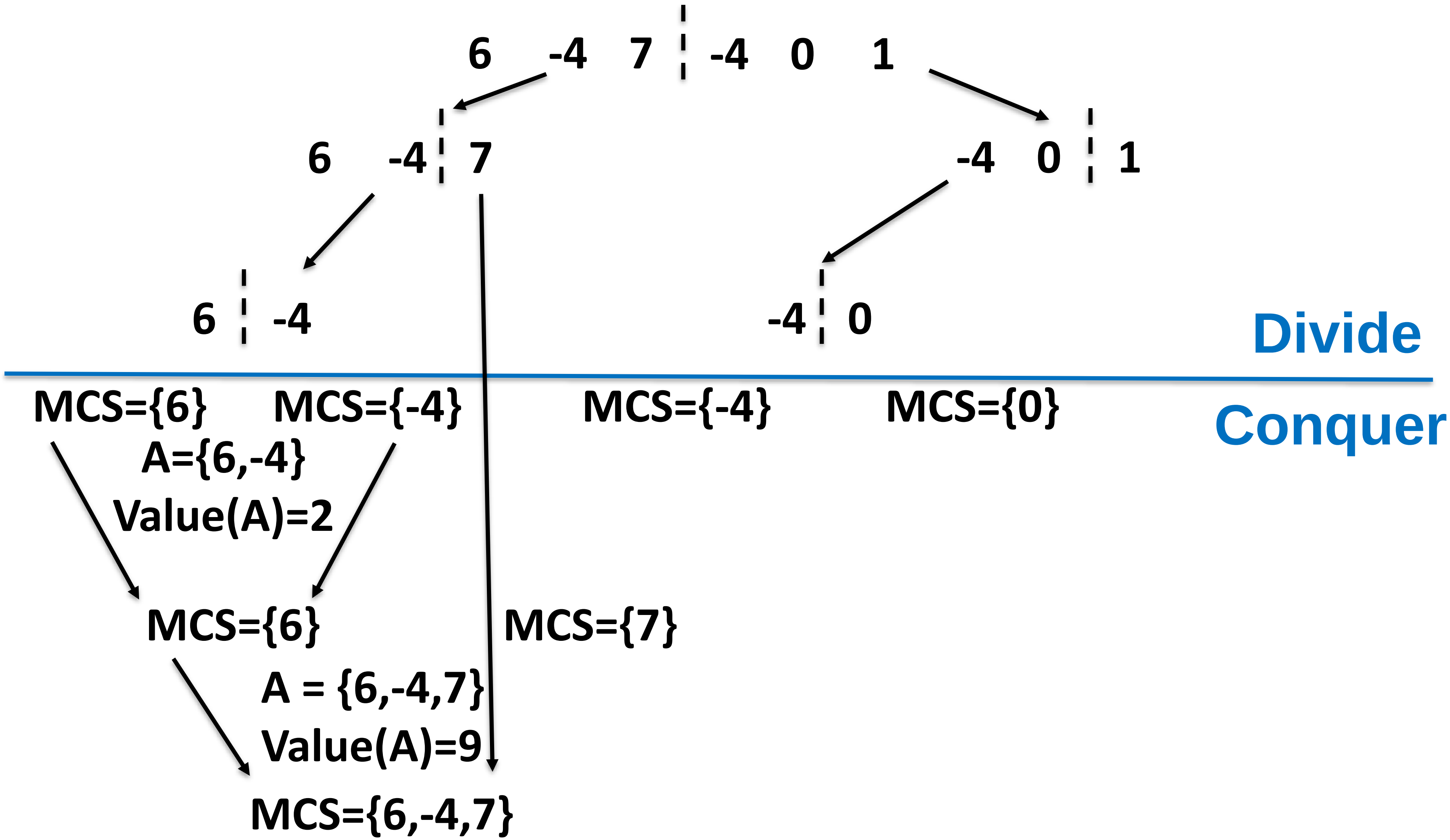
A = {6,-4,7}

Value(A)=9

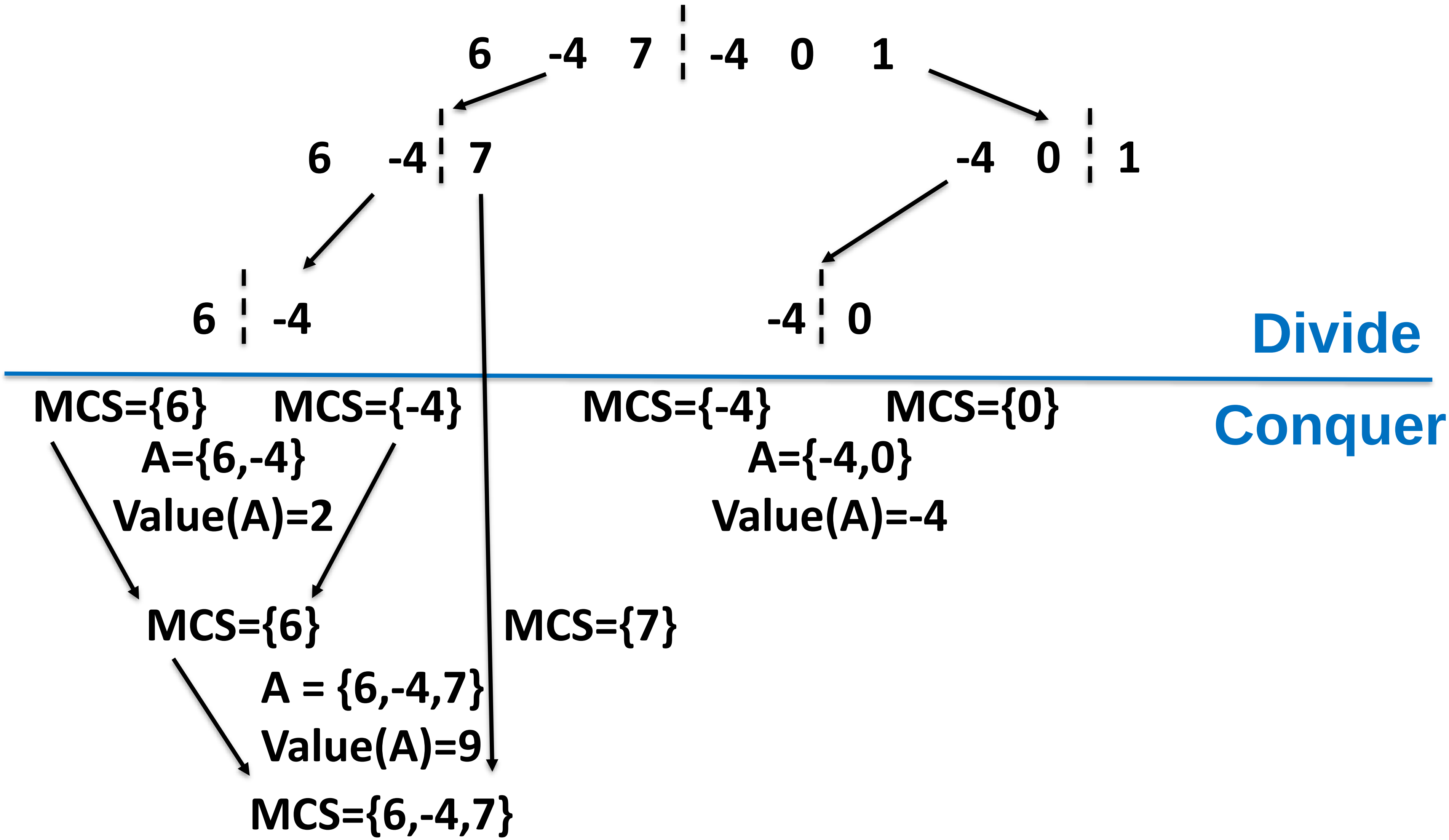
A Full Illustration of the D&C Algorithm



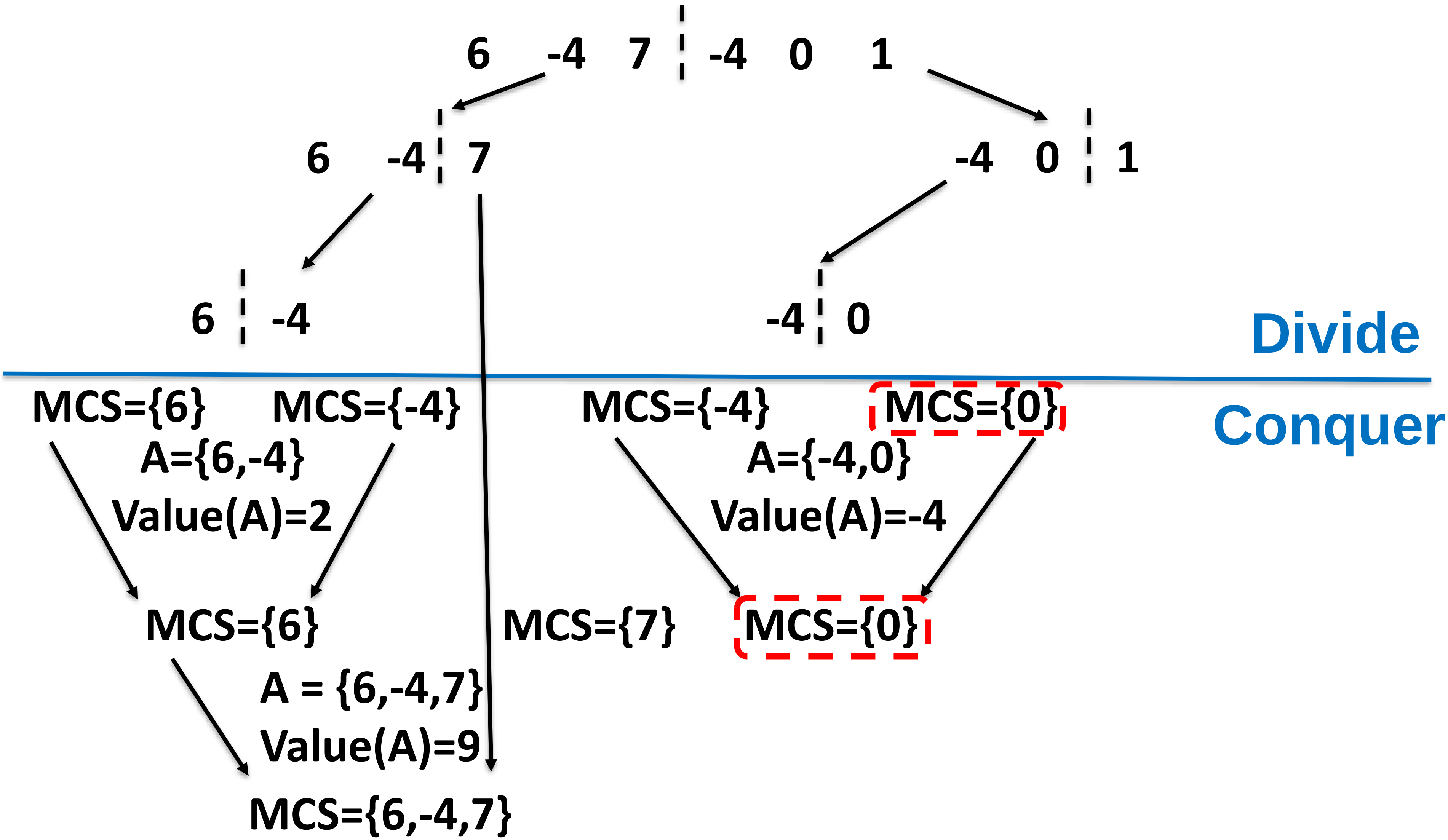
A Full Illustration of the D&C Algorithm



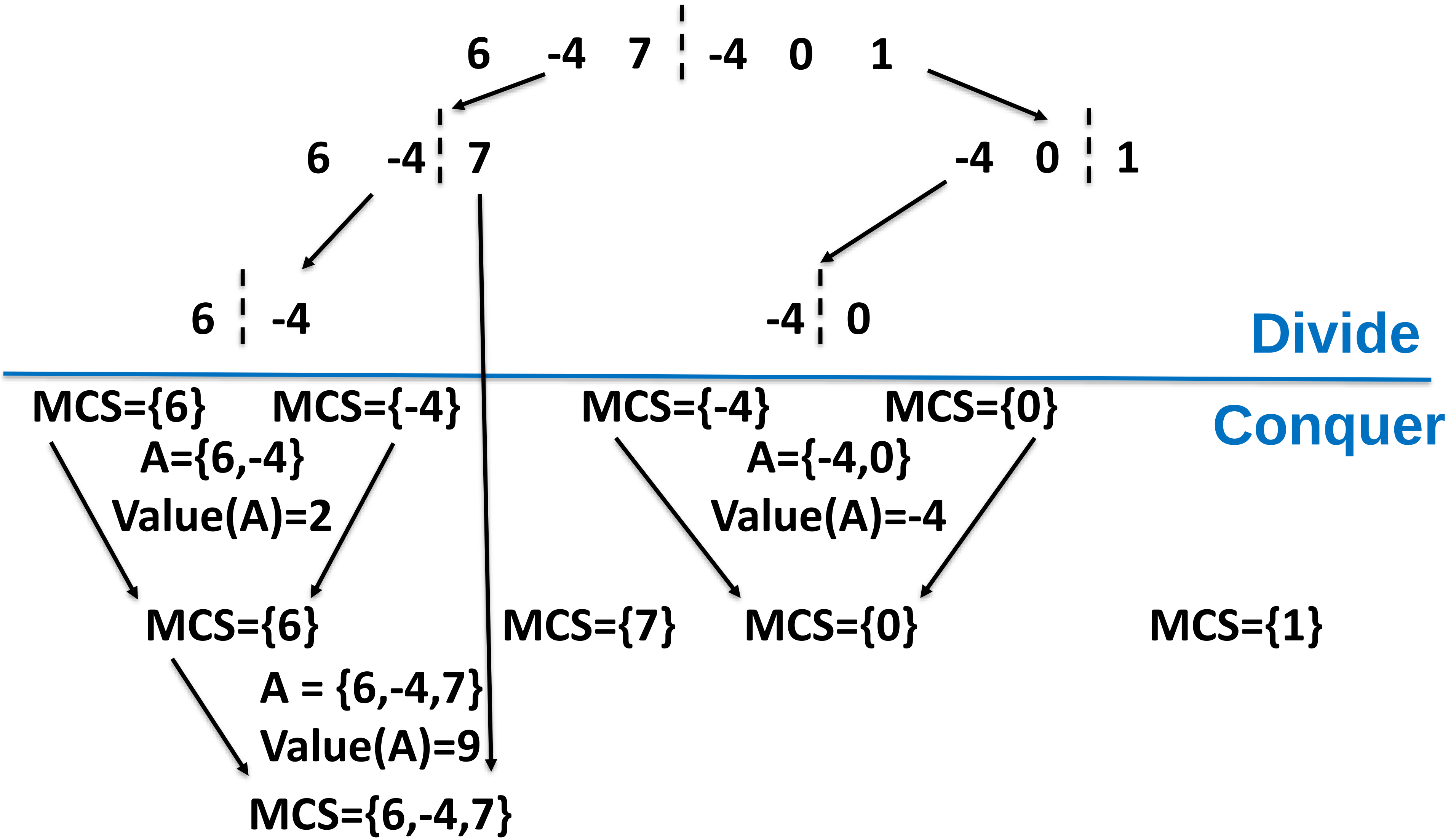
A Full Illustration of the D&C Algorithm



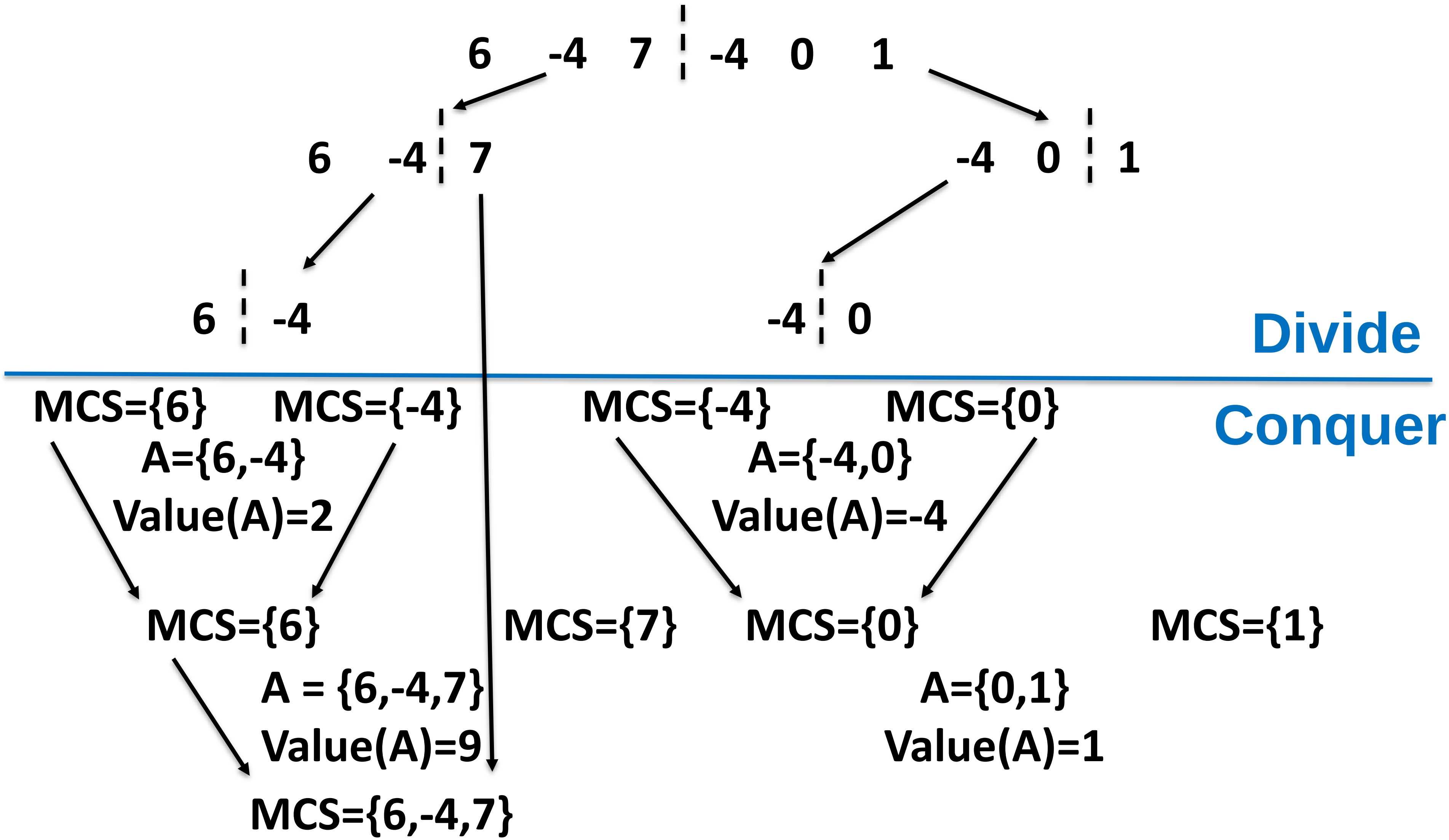
A Full Illustration of the D&C Algorithm



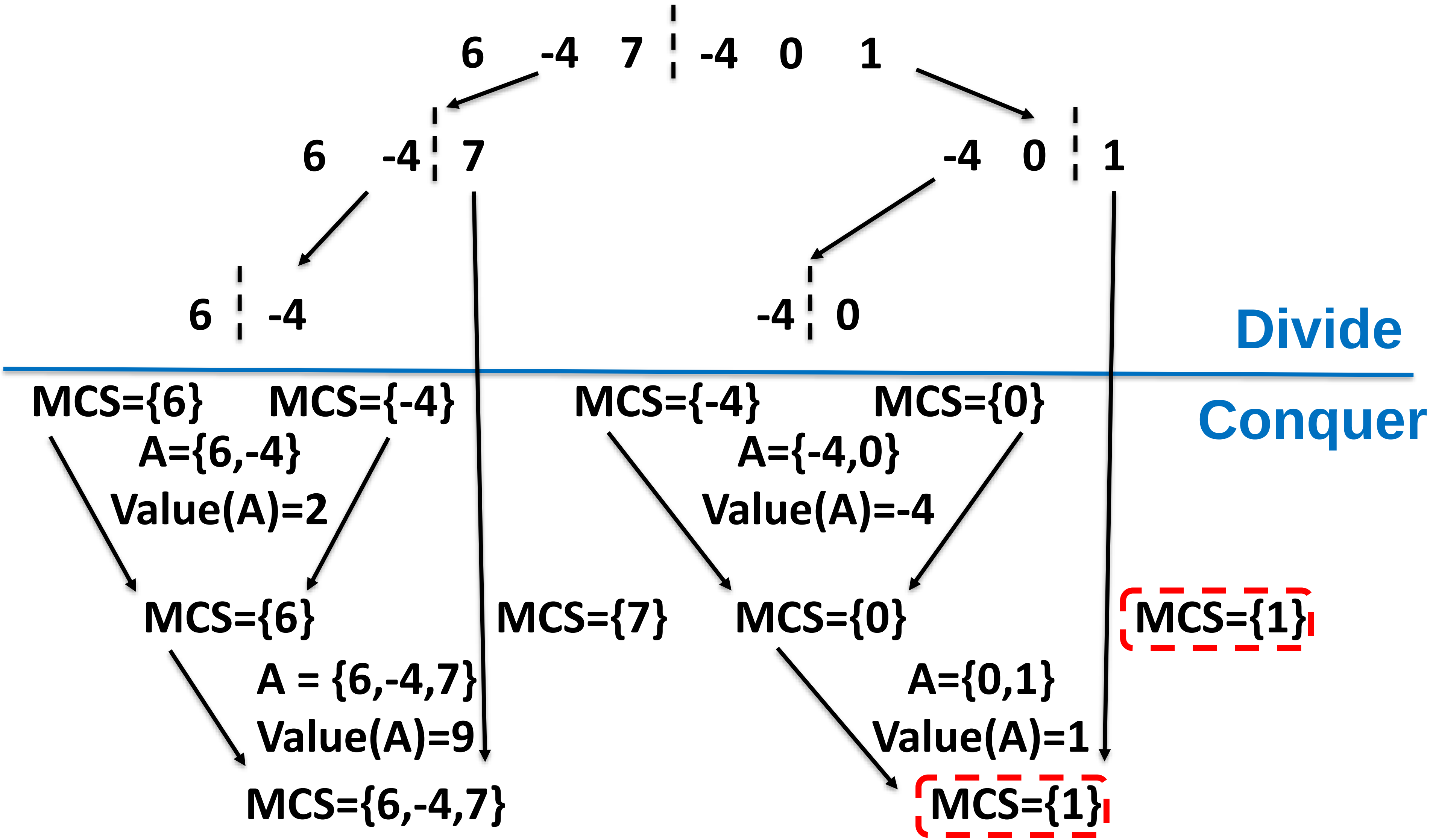
A Full Illustration of the D&C Algorithm



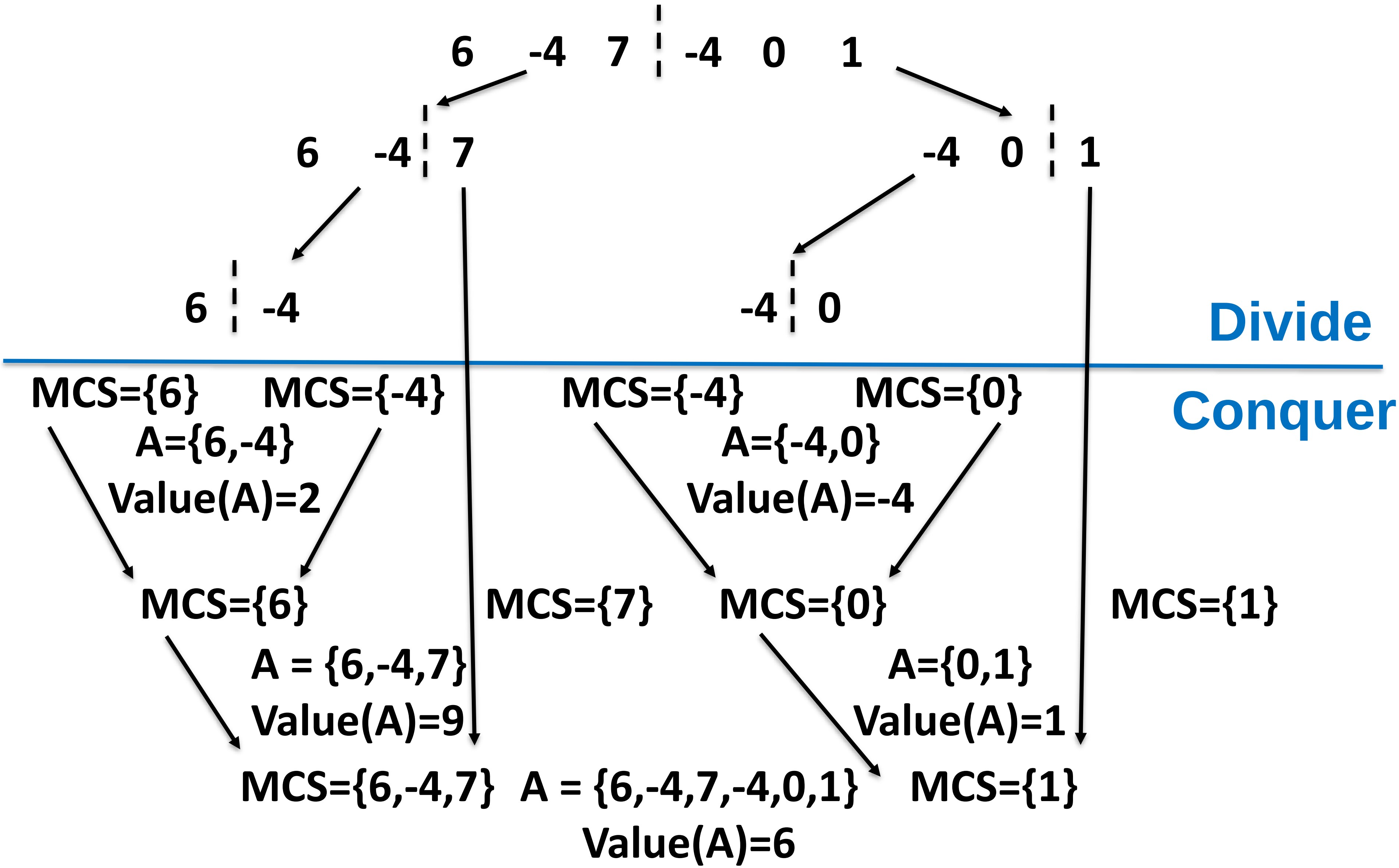
A Full Illustration of the D&C Algorithm



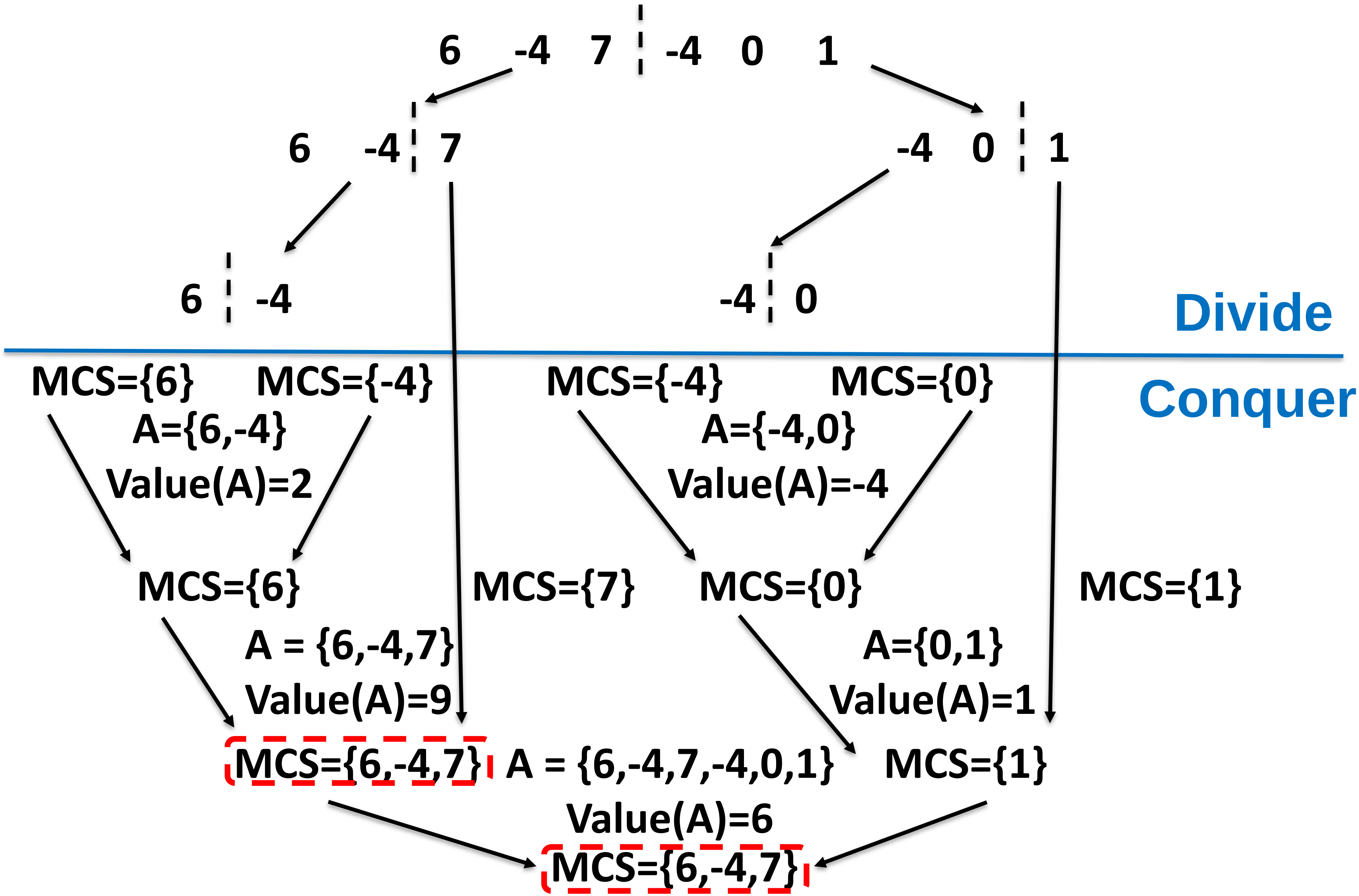
A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm



A Full Illustration of the D&C Algorithm



Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

Analysis of the D&C Algorithm

- *n*: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

```
begin
    if s = t then return A[s] // O(1)
```

Analysis of the D&C Algorithm

- *n*: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run *MCS*(A, s, t)

begin

  if $s = t$ **then return** $A[s]$ // O(1)

  else

    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;

    Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$

.

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

begin
  if $s = t$ **then return** $A[s]$ // O(1)
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$
    Find $MCS(A, m+1, t)$; // $T(\lfloor \frac{n}{2} \rfloor)$

.

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

begin

  if $s = t$ **then return** $A[s]$ `// O(1)`

  else

    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;

    Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$

    Find $MCS(A, m+1, t)$; // $T(\lfloor \frac{n}{2} \rfloor)$

    Find MCS that contains *both* $A[m]$ and $A[m+1]$; `// O(n)`

.

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

```
begin
    if s = t then return A[s] // O(1)
    else
        m ← ⌊(s+t)/2⌋;
        Find MCS(A, s, m); //  T(⌈n/2⌉)
        Find MCS(A, m+1, t); //  T(⌊n/2⌋)
        Find MCS that contains both A[m] and A[m+1]; // O(n)
        return maximum of the three sequences found // O(1)
.
```

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

begin
    if $s = t$ **then return** $A[s]$ // O(1)
    else
        $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
        Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$
        Find $MCS(A, m+1, t)$; // $T(\lfloor \frac{n}{2} \rfloor)$
        Find MCS that contains *both* $A[m]$ and $A[m+1]$; // O(n)
        return maximum of the three sequences found // O(1)
    end
end

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

begin
  if $s = t$ **then return** $A[s]$ `// O(1)`
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$
    Find $MCS(A, m+1, t)$; // $T(\lfloor \frac{n}{2} \rfloor)$
    Find MCS that contains *both* $A[m]$ and $A[m+1]$; `// O(n)`
    return maximum of the three sequences found `// O(1)`
  end
end

$$T(1) = O(\quad)$$

Analysis of the D&C Algorithm

- *n*: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

begin
  if $s = t$ **then return** $A[s]$ // O(1)
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$; // $T(\lceil \frac{n}{2} \rceil)$
    Find $MCS(A, m+1, t)$; // $T(\lfloor \frac{n}{2} \rfloor)$
    Find MCS that contains *both* $A[m]$ and $A[m+1]$; // O(n)
    return maximum of the three sequences found // O(1)
  end
end

$$T(1) = O(1)$$

Analysis of the D&C Algorithm

- n: problem size ($n = t - s + 1$)
- $T(n)$: time needed to run $MCS(A, s, t)$

```
begin
    if s = t then return A[s] // O(1)
    else
        m ← ⌊(s+t)/2⌋;
        Find MCS(A, s, m); // T(⌈n/2⌉)
        Find MCS(A, m+1, t); // T(⌊n/2⌋)
        Find MCS that contains both A[m] and A[m+1]; // O(n)
        return maximum of the three sequences found // O(1)
    end
end
```

$$T(1) = O(1)$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) \quad \text{for } n > 1$$

Analysis of the D&C Algorithm

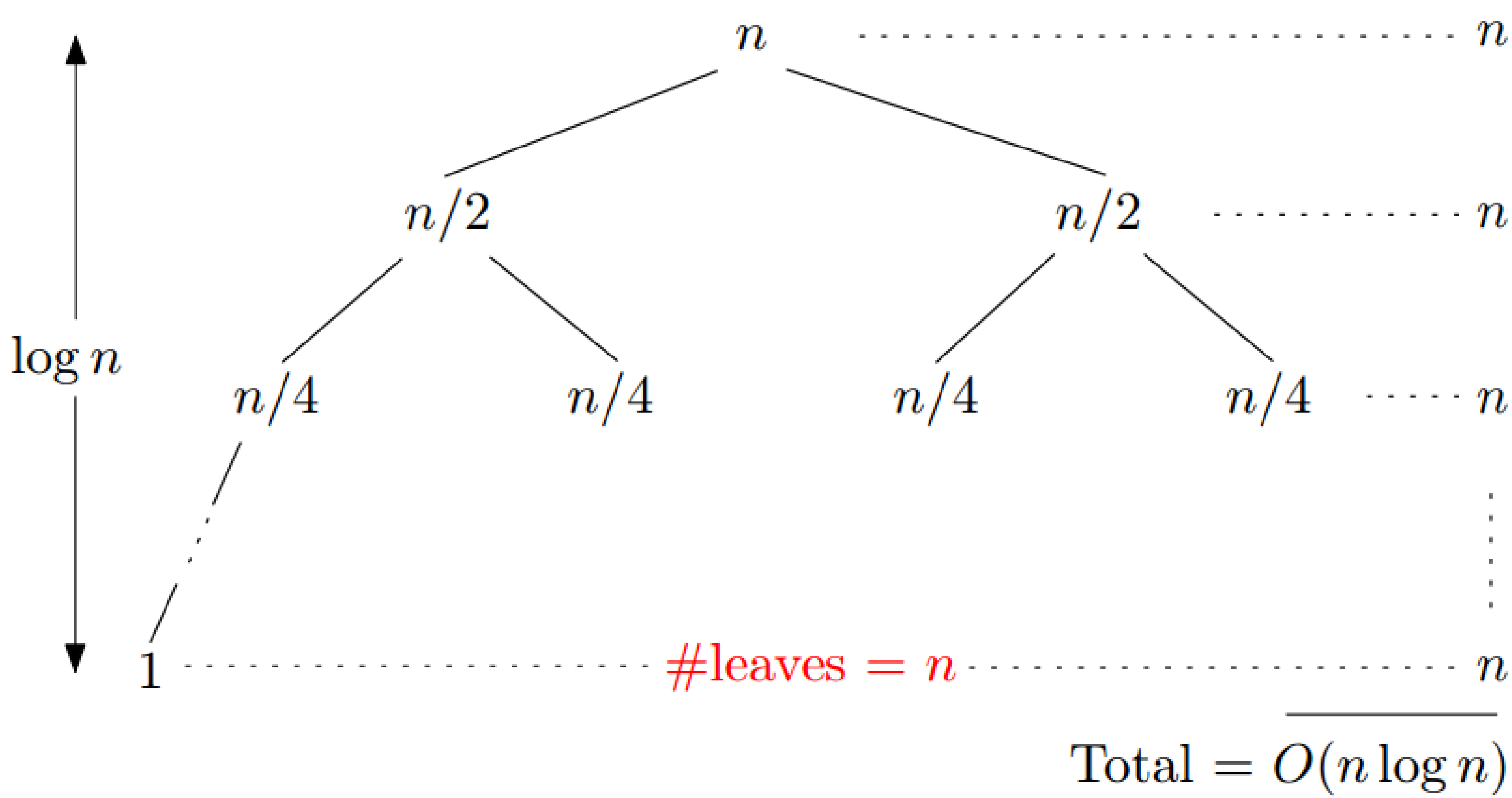
To simplify the analysis, we assume that n is a power of 2

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

Analysis of the D&C Algorithm

To simplify the analysis, we assume that *n* is a power of 2

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



Summary

In the MCS problem, we saw 3 different algorithms for solving the maximum contiguous subarray problem

- A $O(n^3)$ brute force algorithm

Summary

In the MCS problem, we saw 3 different algorithms for solving the maximum contiguous subarray problem

- A $O(n^3)$ brute force algorithm
- A $O(n^2)$ algorithm that reuses data

Summary

In the MCS problem, we saw 3 different algorithms for solving the maximum contiguous subarray problem

- A $O(n^3)$ brute force algorithm
- A $O(n^2)$ algorithm that reuses data
- A $O(n \log n)$ divide-and-conquer algorithm

Summary

In the MCS problem, we saw 3 different algorithms for solving the maximum contiguous subarray problem

- A $O(n^3)$ brute force algorithm
- A $O(n^2)$ algorithm that reuses data
- A $O(n \log n)$ divide-and-conquer algorithm

Can you solve the problem in $O(n)$ time?

Outline

- Introduction to Part I
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm