# Introduction to Algorithms

## Lecture 1: Analysis of Algorithms

**Shouzhen Gu**

# Outline

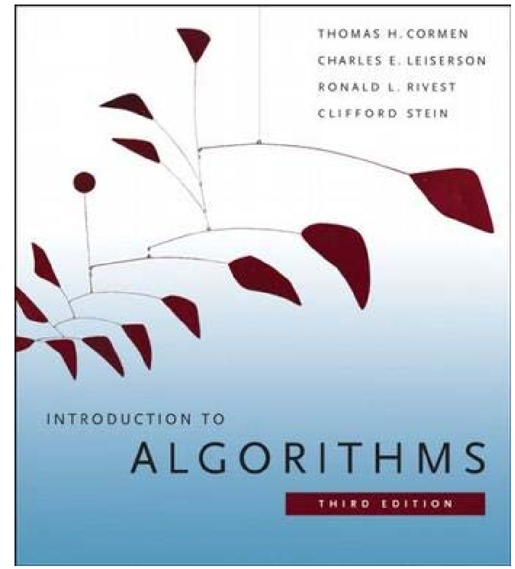
- Course information
- Algorithmic thinking
- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

# Outline

- Course information
- Algorithmic thinking
- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

# Course Information

- Instructor:
  - Shouzhen Gu (谷守珍)
  - Email: szgu@sei.ecnu.edu.cn
  - Office: East 213, Mathematics Building
- Tutor：
  - Xiongpeng Hu (胡雄鹏)
  - Email: 51215902089@stu.ecnu.edu.cn




- Grade Determinates:
  - 10% Attendance + 15% Quiz + 15% Homework
  - 60% Final Report
- Textbook
  - “Introduction to Algorithms (3.ed.)”, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

# Course Overview

- The course is divided into 8 modules
  - Algorithmic Thinking
  - Sorting & Trees
  - Hashing
  - Numerics
  - Graphs
  - Shortest Paths
  - Dynamic Programming
  - Advanced Topics

# Outline

- Course information
- Algorithmic thinking
- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

# Peak Finder: One-dimensional Version

- Position 2 is a peak if and only if b ≥ a and b ≥ c. Position 9 is a peak if i ≥ h.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | h | i |

- Problem: Find a peak if it exists
- Does it always exist?

# Straightforward Algorithm

- Start from left

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 4 | 3 | 2 | 1 | 3 | 4 | 5 |

- Could look at $n$ elements in the worst case

# Straightforward Algorithm

- Start from left

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 4 | 3 | 2 | 1 | 3 | 4 | 5 |

- Could look at *n* elements in the worst case

Can we do better?

# Divide & Conquer

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 4 | 3 | 2 | 1 | 3 | 4 | 5 |

- Look at *n/2* position
  - If a[n/2] < a[n/2 − 1] then only look at left half 1…n/2-1 to look for peak
  - Else if a[n/2] < a[n/2 + 1] then only look at right half n/2 + 1 . . . n to look for peak
  - Else n/2 position is a peak:

$$a[n/2] \geq a[n/2 - 1]$$

$$a[n/2] \geq a[n/2 + 1]$$

# What is the complexity?

$$T(n) = T(n/2) + \Theta(1)$$

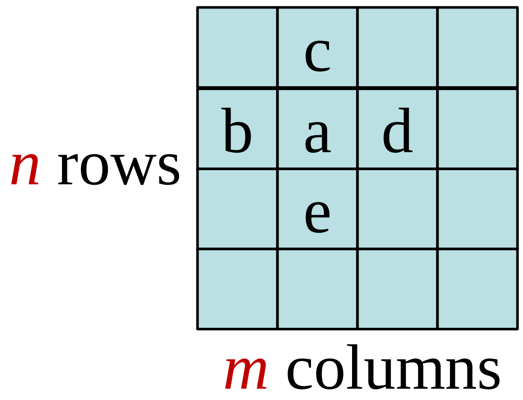
to compare a[n/2] to neighbors

Then,

$$T(n) = \Theta(1) + \ldots + \Theta(1) \text{ } (\log_2(n) \text{ times}) = \Theta(\log_2(n))$$

- If $n = 1000000$,
  - $\Theta(n)$ algorithm needs 13 sec in python
  - $\Theta(\log n)$ algorithm only need 0.001 sec in python

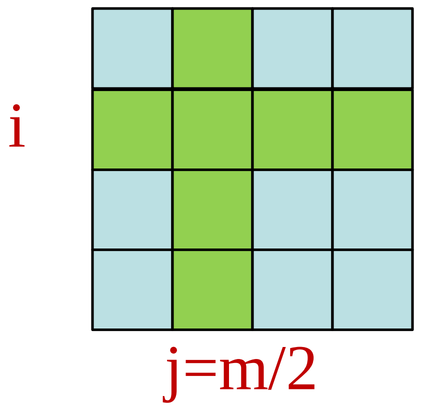
# Peak Finder: Two-dimensional Version

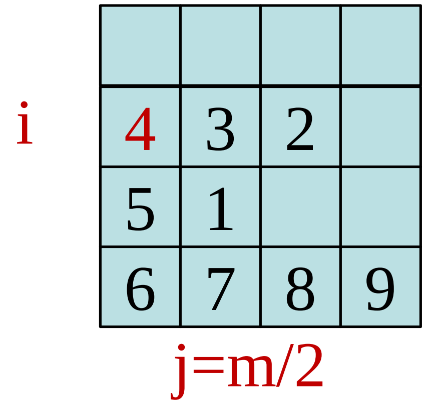



- $a$ is a 2D-peak iff $a \geq b$, $a \geq d$, $a \geq c$, $a \geq e$
- Greedy Ascent Algorithm: $\Theta(nm)$ complexity,
- $\Theta(n^2)$ algorithm if $m = n$

# Extend 1D Divide and Conquer to 2D

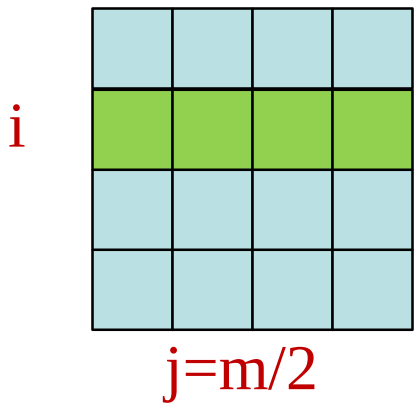
- Attempt #1:
  - Pick middle column $j = m/2$
  - Find a 1D-peak at $i, j$
  - Use $(i, j)$ as a start point on row $i$ to find 1D-peak on row $i$

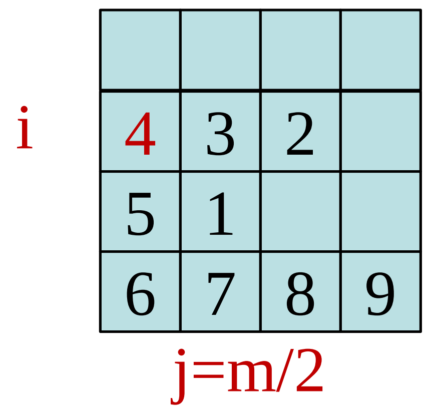


# Extend 1D Divide and Conquer to 2D

- Attempt #1:
  - Pick middle column j = m/2
  - Find a 1D-peak at i, j
  - Use (i, j) as a start point on row i to find 1D-peak on row i




End up with 4 which is not a 2D-peak

Problem: 2D-peak may not exist on row i

# Extend 1D Divide and Conquer to 2D

- Attempt #2:
  1. Pick middle column $j = m/2$
  2. Find global maximum on column $j$ at $(i, j)$
  3. Compare $(i, j - 1)$, $(i, j)$, $(i, j + 1)$
  4. Pick left columns of $(i, j - 1) > (i, j)$
  5. Pick right columns of $(i, j + 1) > (i, j)$
  6. $(i, j)$ is a 2D-peak if neither condition holds
  7. Solve the new problem with half the number of columns
  8. When you have a single column, find global maximum and you're done

# Example of Attempt #2

| | | | |
|---|---|---|---|
| 10 | 8 | 10 | 10 |
| 14 | 13 | 12 | 11 |
| 15 | 9 | 11 | 21 |
| 16 | 17 | 19 | 20 |

pick this column 17 global max for this column

| | |
|---|---|
| 10 | 10 |
| 12 | 11 |
| 11 | 21 |
| 19 | 20 |

pick this column 19 global max for this column

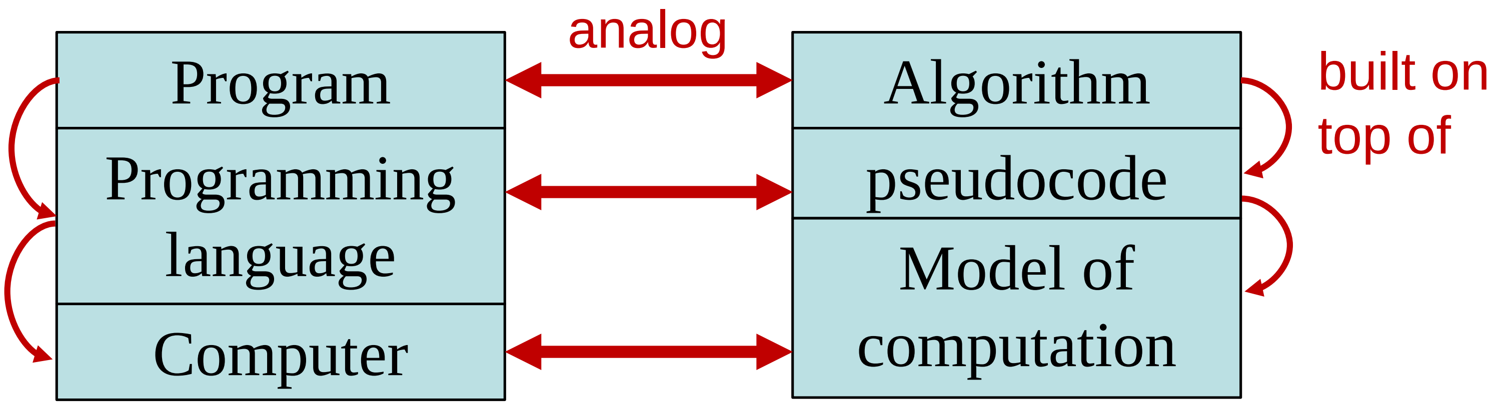
find 21

| |
|---|
| 10 |
| 11 |
| 21 |
| 20 |

- Complexity
  - $T(n, m) = T(n, m/2) + \Theta(n)$ (to find global maximum on a column——n rows)
  - $T(n, m) = \Theta(n) + \ldots + \Theta(n)$
  - $T(n, m) = \Theta(n \log m) = \Theta(n \log n)$ if $m = n$

# What is an Algorithm?

- Mathematical abstraction of computer program
- Computational procedure to solve a problem




- Model of computation specifies:
  - what operations an algorithm is allowed
  - cost (time, space, . . .) of each operation
  - cost of algorithm = sum of operation costs

# Why study algorithms and performance?

- Algorithms help us to understand **scalability**.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a **language** for talking about program behavior.
- Performance is the **currency** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

# Outline

- Course information
- Algorithmic thinking
- **Insertion sort**
- Asymptotic analysis
- Merge sort
- Recurrences

# The problem of sorting

- ***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers
- ***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \ldots \le a'_n$.
- Example:
  - ***Input:*** 8 2 4 9 3 6
  - ***Output:*** 2 3 4 6 8 9

# Insertion sort

“pseudocode”

$$
\begin{array}{l}
\textsc{Insertion-Sort}\,(A, n) \quad \rhd\ A[1\,.\,.\,n] \\
\quad \textbf{for } j \leftarrow 2 \textbf{ to } n \\
\qquad \textbf{do } key \leftarrow A[\,j] \\
\qquad\quad i \leftarrow j - 1 \\
\qquad\quad \textbf{while } i > 0 \text{ and } A[i] > key \\
\qquad\qquad \textbf{do } A[i+1] \leftarrow A[i] \\
\qquad\qquad\quad i \leftarrow i - 1 \\
\qquad\quad A[i+1] = key
\end{array}
$$

# Loop invariants & correctness of insertion sort

- **Loop invariant** help us to understand why an algorithm is correct. We must show three things:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps us show that the algorithm is correct.
- Prove a base case and an inductive step, such that we can prove the first two properties hold.
- We stop the “inductive” when the loop terminates, and prove the third property hold.

# Example of insertion sort

8 2 4 9 3 6

# Example of insertion sort

8 2 4 9 3 6

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |

# Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |

# Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 3 | 4 | 8 | 9 | 6 |

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 3 | 4 | 8 | 9 | 6 |

# Example of insertion sort

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 | |
| 2 | 8 | 4 | 9 | 3 | 6 | |
| 2 | 4 | 8 | 9 | 3 | 6 | |
| 2 | 4 | 8 | 9 | 3 | 6 | |
| 2 | 3 | 4 | 8 | 9 | 6 | |
| 2 | 3 | 4 | 6 | 8 | 9 | *done* |

# Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

# Kinds of analyses

- **Worst-case:** (usually)
  - $T(n)$ =maximum time of algorithm on any input of size $n$.
- **Average-case:** (sometimes)
  - $T(n)$ =expected time of algorithm over all inputs of size $n$.
  - Need assumption of statistical distribution of inputs.
- **Best-case:** (bogus)
  - Cheat with a slow algorithm that works fast on *some* input.

# Machine-independent time

- What is insertion sort's worst-case time?
- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

**BIG IDEA:**

- Ignore machine-dependent constants
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

**"Asymptotic Analysis"**

# Outline

- Course information
- Algorithmic thinking
- Insertion sort
- **Asymptotic analysis**
- Merge sort
- Recurrences

# Asymptotic notation

- $O$-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

- EXAMPLE: $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

*funny, "one-way" equality*

*functions, not values*

- Engineering: Drop low-order terms; ignore leading constants

# Set definition of O-notation

$O(g(n)) = \{f(n)$ : there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

- EXAMPLE: $2n^2 \in O(n^3)$

# Macro substitution

- ***Convention:*** A set in a formula represents an anonymous function in the set.
- **EXAMPLE:**

$f(n) = n^3 + O(n^2)$

means

$f(n) = n^3 + h(n)$

for some $h(n) \in O(n^2)$

$\mathrm{n}^2 + \mathrm{O}(\mathrm{n}) = \mathrm{O}(\mathrm{n}^2)$

means

for any $f(n) \in O(n)$:

$n^2 + f(n) = h(n)$

for some $h(n) \in O(n^2)$

# Ω-notation (lower bounds)

- $\Omega$-notation is an *lower-bound* notation. It makes no sense to say $f(n)$ is at least $\Omega(n^2)$.

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

- EXAMPLE: $\sqrt{n} = \Omega(\lg n)$ $(c = 1, n_0 = 16)$

# Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

- EXAMPLE: $n^2 + 2n = \Theta(n^2)$

# o-notation and ω-notation

- $O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.
- $o$-notation and $\omega$-notation are like < and >.

$o(g(n)) = \{f(n)$ : there exist constants $c > 0$,
$n_0 > 0$ such that $0 \leq f(n) < cg(n)$
for all $n \geq n_0\}$

- EXAMPLE: $2n^2 = o(n^3)$ $(n_0 = 2/c)$

# o-notation and ω-notation

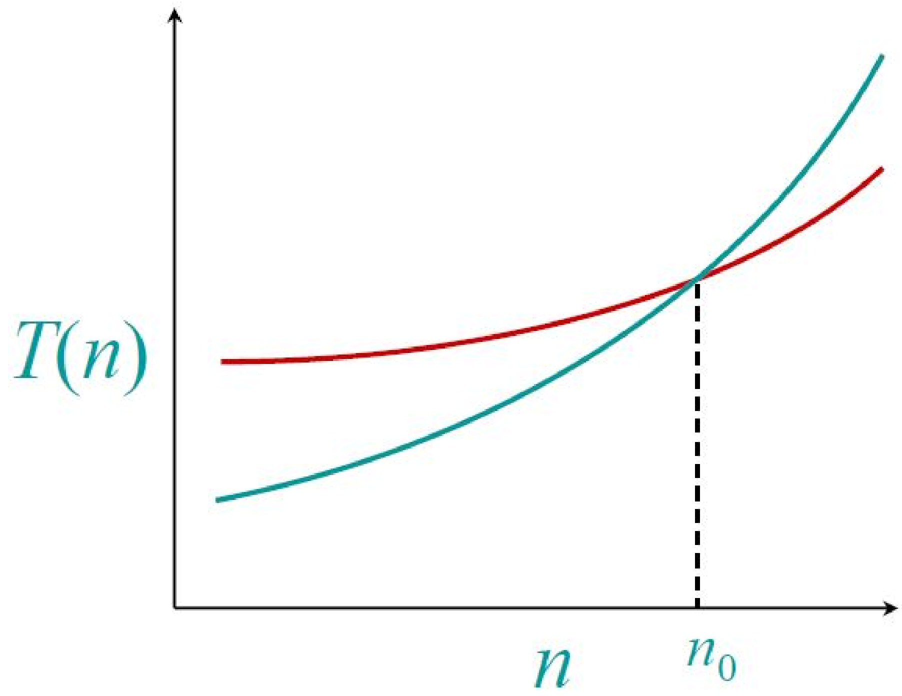
- $O$-notation and Ω-notation are like $\leq$ and $\geq$.
- $o$-notation and ω-notation are like < and >.

$\omega(g(n)) = \{f(n)$ : there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

- EXAMPLE: $\sqrt{n} = \omega(\lg n)$ $(n_0 = 1+1/c)$

# Asymptotic performance

- When $n$ gets large enough, a $\Theta(n^2)$ algorithm ***always*** beats a $\Theta(n^3)$ algorithm.
- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.




- Asymptotic analysis is a useful tool to help to structure our thinking.

# Insertion sort analysis

- *Worst case*: Input reverse sorted

$$\mathrm{T}(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

- *Average case*: All permutations equally likely

$$\mathrm{T}(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

- Is insertion sort a fast sorting algorithm?
  - Moderately so, for small *n*.
  - Not at all, for large *n*

# Outline

- Course information
- Algorithmic thinking
- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

# Merge sort

MERGE-SORT $A[1\,.\,.\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,.\,.\,\lceil n/2 \rceil]$
   and $A[\,\lceil n/2 \rceil + 1\,.\,.\,n\,]$ .
3. "***Merge***" the 2 sorted lists.

***Key subroutine:*** **MERGE**

# Merging two sorted arrays

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

# Merging two sorted arrays

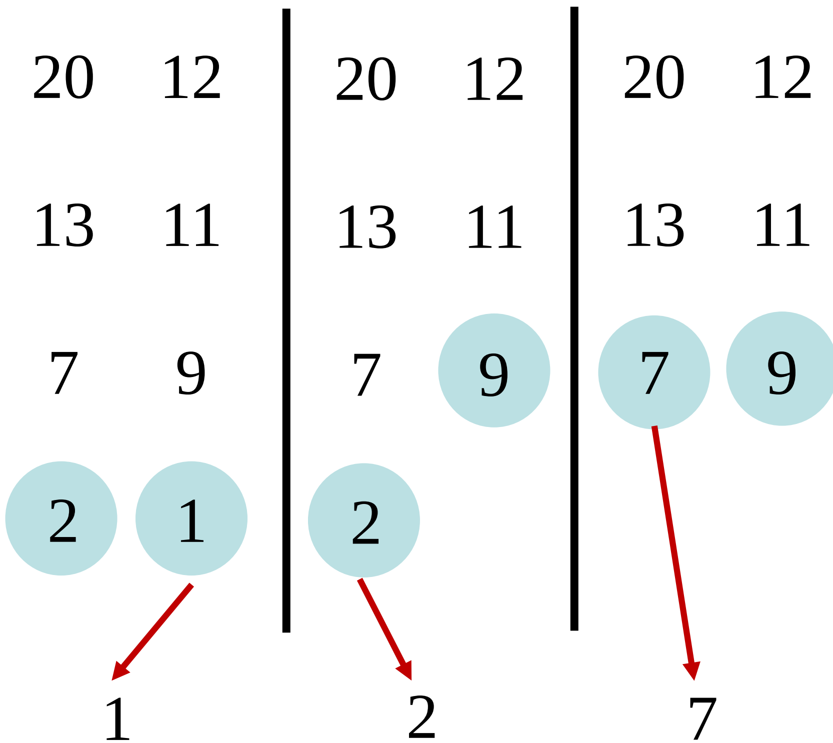
| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

1

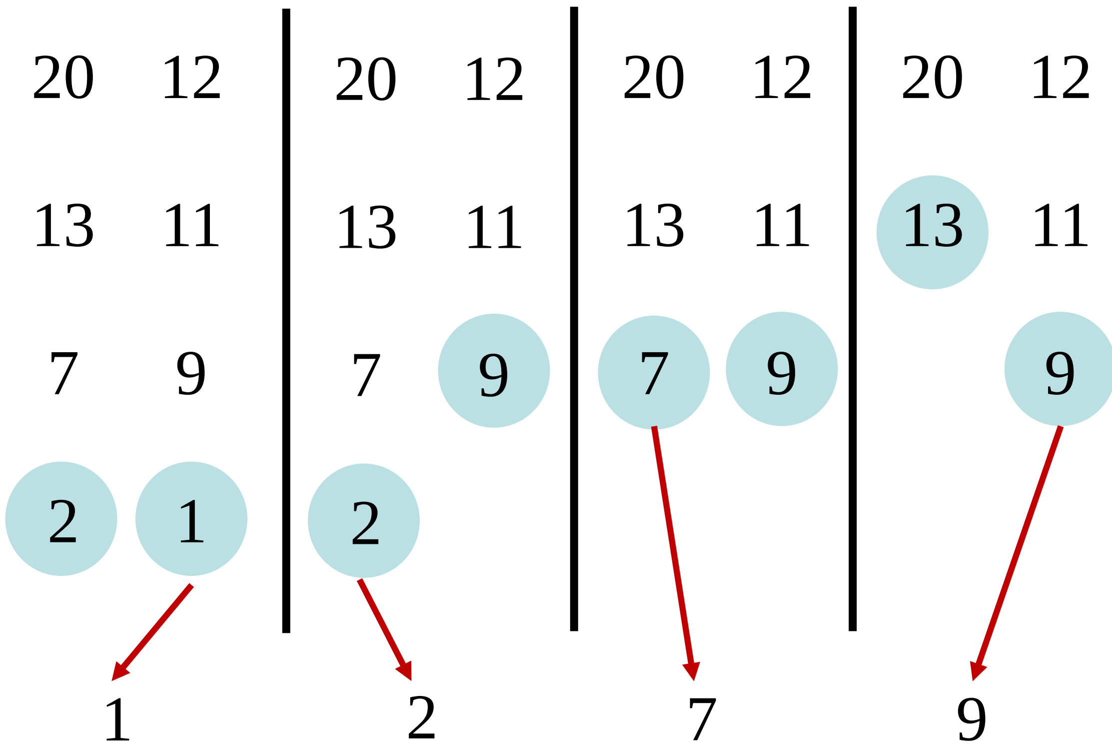
# Merging two sorted arrays

| | | | | |
|---|---|---|---|---|
| 20 | 12 | | 20 | 12 |
| 13 | 11 | | 13 | 11 |
| 7 | 9 | | 7 | 9 |
| 2 | 1 | | 2 | |
| | 1 | | | 2 |

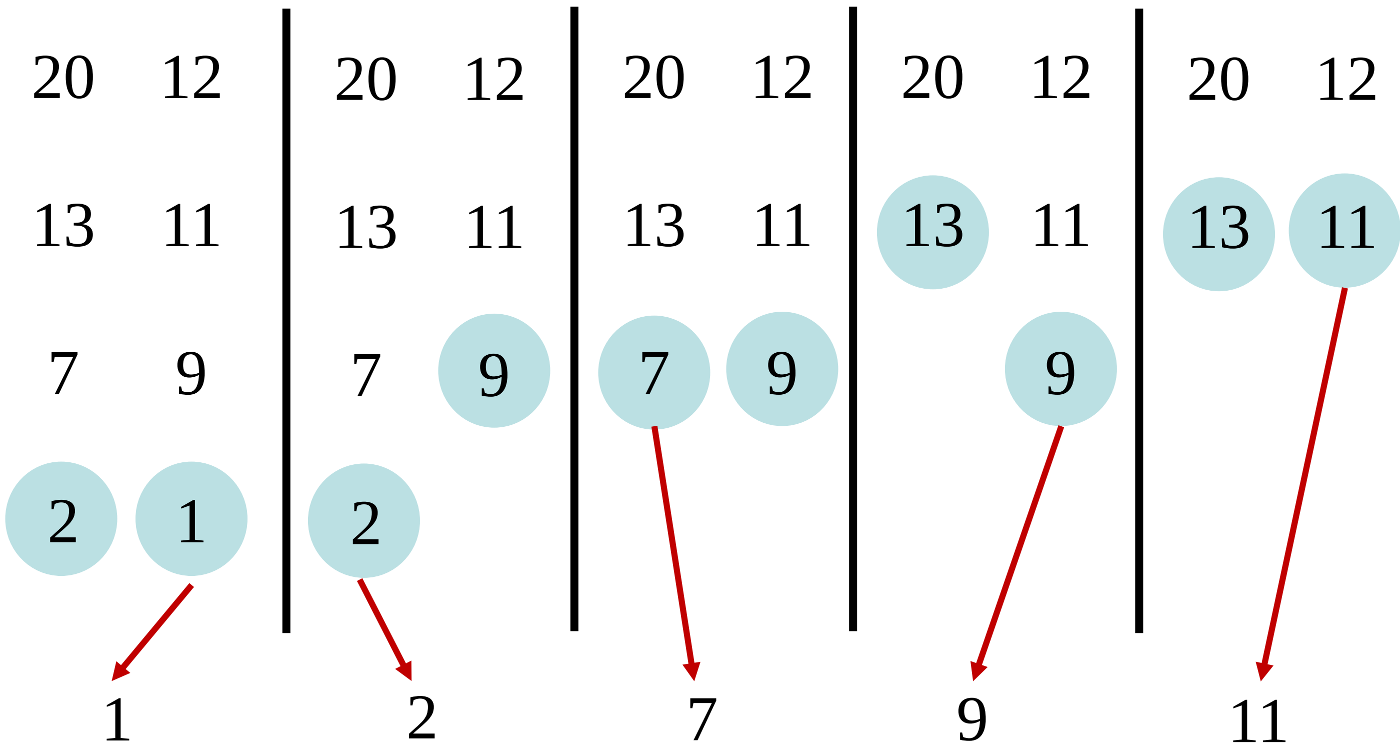
# Merging two sorted arrays



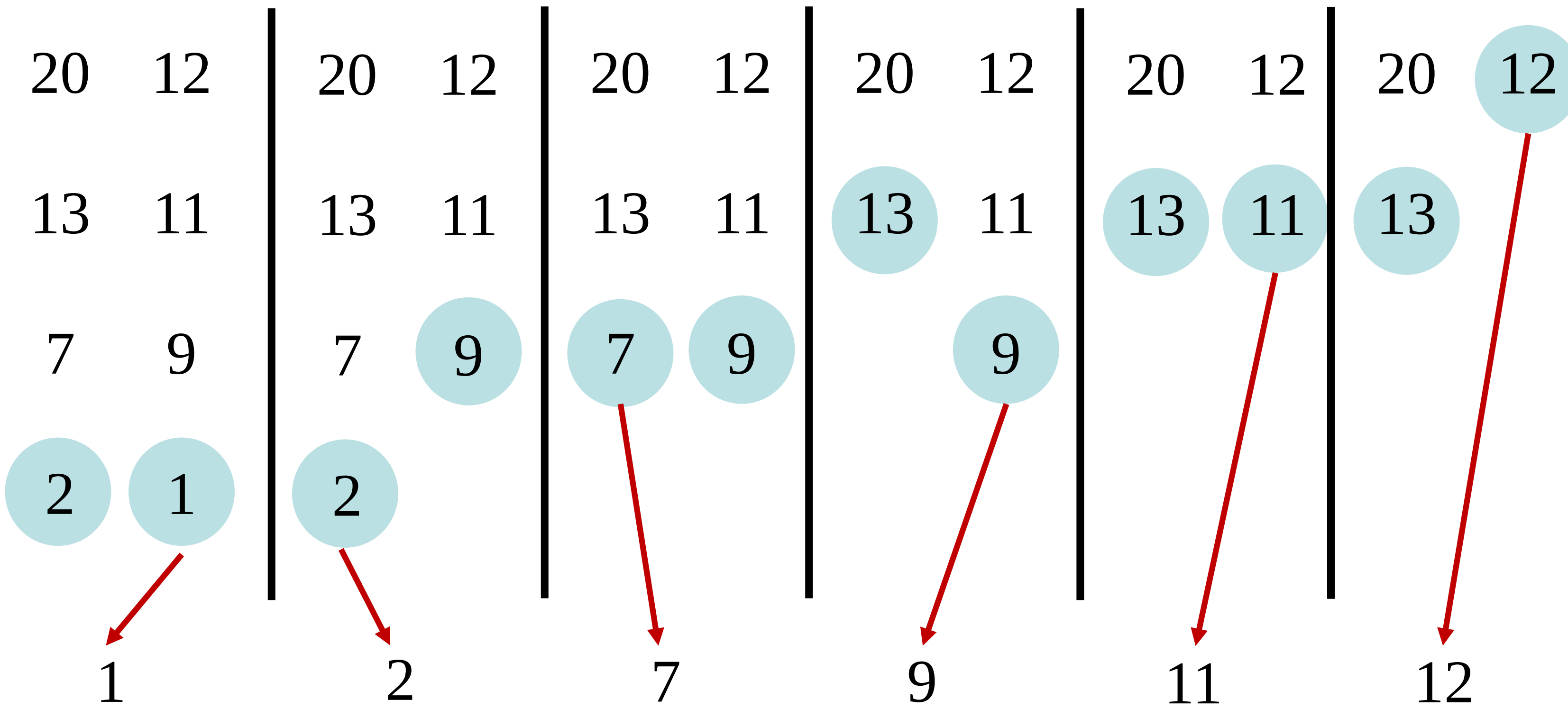
# Merging two sorted arrays



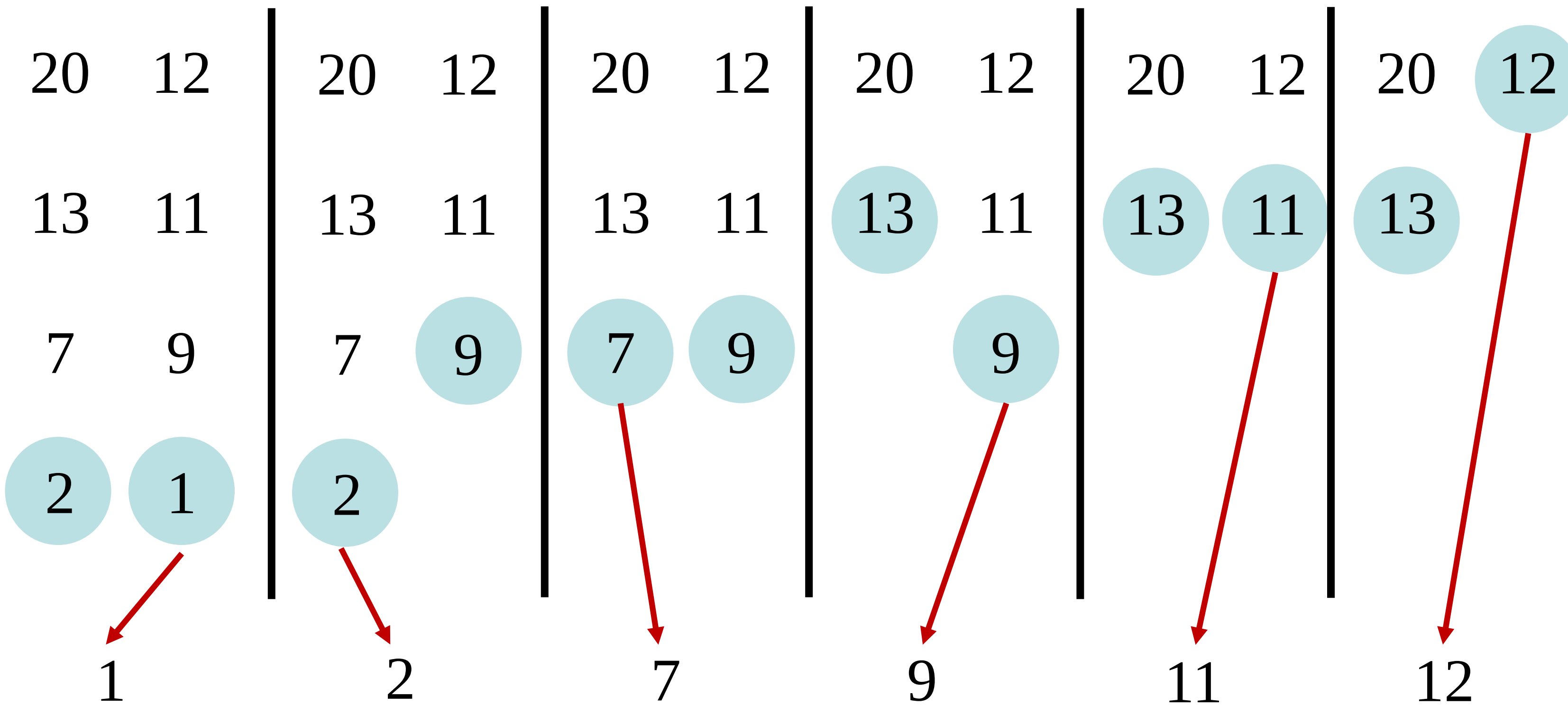
# Merging two sorted arrays



# Merging two sorted arrays



# Merging two sorted arrays




Time = $\Theta(n)$ to merge a total of $n$ elements (linear time)

# Analyzing merge sort

| | |
|---|---|
| $T(n)$ | **MERGE-SORT** $A[1 \,..\, n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done. |
| $2T(n/2)$ | 2. Recursively sort $A[1 \,..\, \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \,..\, n]$. |
| $\Theta(n)$ | 3. "***Merge***" the 2 sorted lists. |

***Sloppiness:*** Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically

# Outline

- Course information
- Algorithmic thinking
- Insertion sort
- Asymptotic analysis
- Merge sort
- **Recurrences**

# Recurrence for merge sort

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

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.

# Solving recurrences

- The analysis of merge sort required us to solve a recurrence.
- Recurrences are like solving integrals, differential equations, etc.
  - Learn a few tricks.

# Substitution method

- The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

- **EXAMPLE:** $T(n) = 4T(n/2) + n$
  - [Assume that $T(1) = \Theta(1)$.]
  - Guess $O(n^3)$. (Prove O and Ω separately.)
  - Assume that $T(k) \le ck^3$ for $k < n$ .
  - Prove $T(n) \le cn^3$ by induction.

# Example of substitution

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^3 + n \\
&= (c/2)n^3 + n \\
&= cn^3 - ((c/2)n^3 - n) \longleftarrow \text{desired–residual} \\
&\le cn^3 \longleftarrow \text{desired}
\end{aligned}
$$

- whenever $(c/2)n^3 - n \ge 0$, for example, if $c \ge 0$ and $n \ge 1$
- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base***: $T(n) = \Theta(1)$for all $n < n_0$, where $n_0$ is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$"$\le cn^3$, if we pick $c$ big enough.

**This bound is not tight!**

# A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \leq ck^2$ for $k < n$ :

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4c(n/2)^2 + n \\
&= cn^2 + n \\
&= O(n^2)
\end{aligned}
$$

# A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \le ck^2$ for $k < n$ :

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^2 + n \\
&= cn^2 + n \\
&= \cancel{O(n^2)}
\end{aligned}
$$

***Wrong!*** We must prove the I.H.

# A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \leq ck^2$ for $k < n$ :

$$\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4c(n/2)^2 + n \\
&= cn^2 + n \\
&= \cancel{O(n^2)}
\end{aligned}$$

***Wrong!*** We must prove the I.H.

$$= cn^2 - (-n)$$ [ desired–residual]

$$\leq cn^2$$ for ***no*** choice of c > 0. ***Lose!***

# A tighter upper bound!

- **IDEA:** Strengthen the inductive hypothesis
- ***Subtract*** a low-order term
- Inductive hypothesis: $T(k) \leq c_1 k^2 – c_2 k$ for $k < n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\
&= c_1 n^2 - 2c_2 n + n \\
&= c_1 n^2 - c_2 n - (c_2 n - n) \\
&\leq c_1 n^2 - c_2 n \text{ if } c_2 \geq 1
\end{aligned}
$$

- Pick $c_1$ big enough to handle the initial conditions.

# Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method

# Example of recursion tree

- Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$T(n)$$

# Example of recursion tree

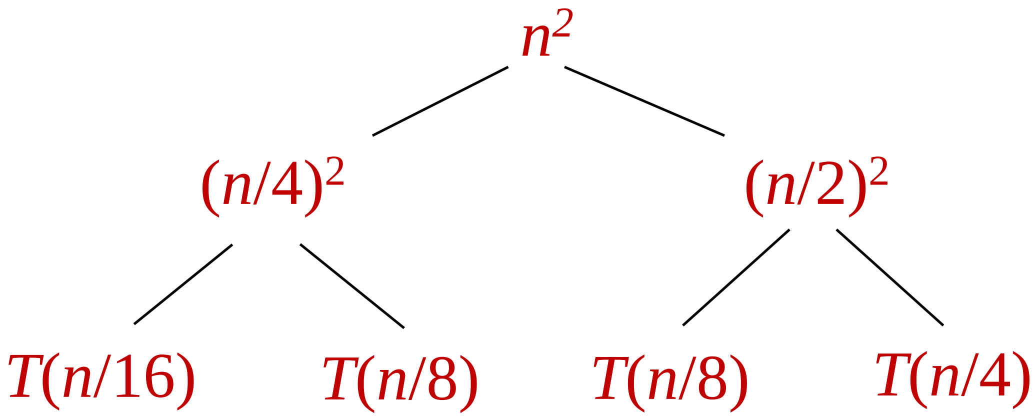
- Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$n^2$$

$$T(n/4) \qquad\qquad\qquad T(n/2)$$

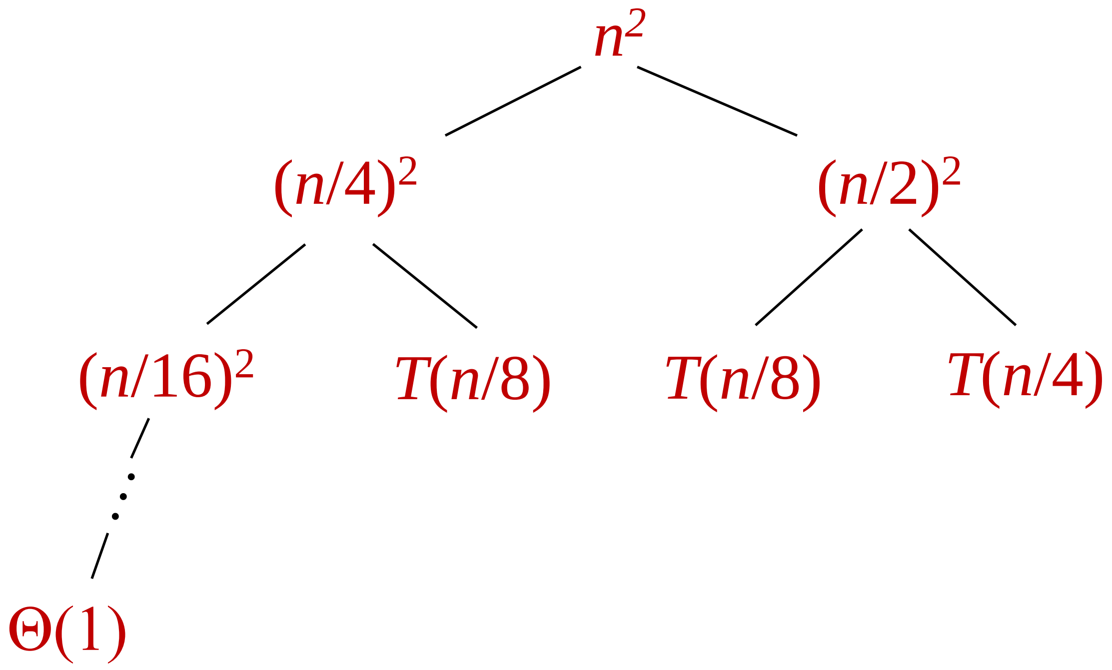
# Example of recursion tree

- Solve $T(n) = T(n/4) + T(n/2) + n^2$:



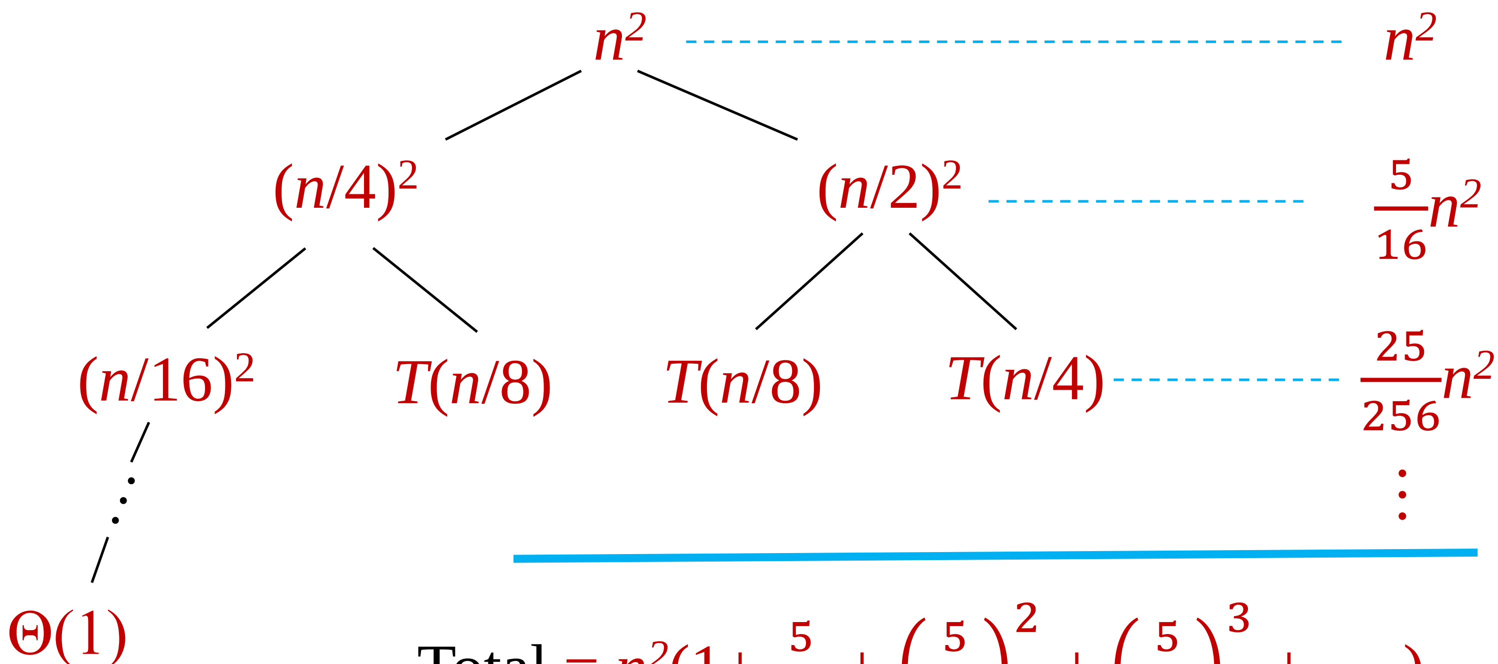
# Example of recursion tree

- Solve $T(n) = T(n/4) + T(n/2) + n^2$:



# Example of recursion tree

- Solve $T(n) = T(n/4) + T(n/2) + n^2$:




$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \dots\right)$$
$$= \Theta(n^2)$$

# The master method

- The master method applies to recurrences of the form

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

- where $a \geq 1$, $b > 1$, and $f$ is asymptotically positive.

# Three common cases

- Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an $n^{\varepsilon}$ factor)
   - **Solution:** $T(n) = \Theta(n^{\log_b a - \varepsilon})$
2. $f(n) = \Theta(n^{\log_b a} lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - **Solution:** $T(n) = \Theta(n^{\log_b a - \varepsilon} lg^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an $n^{\varepsilon}$ factor)
   - ***and*** $f(n)$ grows satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.
   - **Solution:** $T(n) = \Theta(f(n))$

# Examples

$T(n) = 4T(n/2) + n$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.

**Case1**: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore T(n) = \Theta(n^2)$

$T(n) = 4T(n/2) + n^2$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.

**Case2**: $f(n) = O(n^2 \lg^0 n)$ , that is, $k = 0$.

$\therefore T(n) = \Theta(n^2 \lg n)$

# Examples

$T(n) = 4T(n/2) + n^3$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.

**CASE 3**: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

***and*** $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore T(n) = \Theta(n^3)$

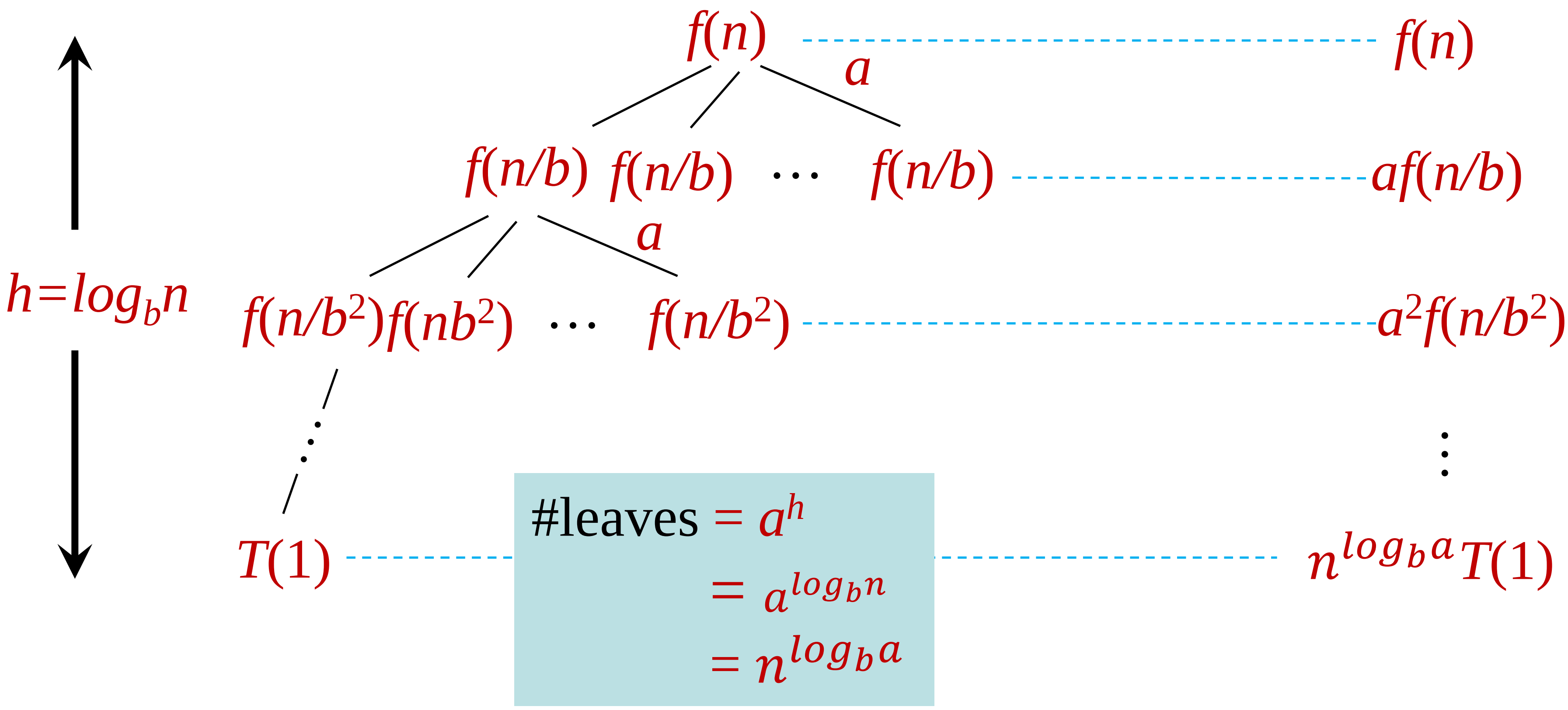
$T(n) = 4T(n/2) + n^2/\lg n$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2/\lg n$.

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

# Idea of master theorem

➢ ***Recursion tree:***



# Idea of master theorem

- ***Recursion tree:***

$f(n)$ ------------------------------------- $f(n)$

$a$

$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$ ------------------------ $af(n/b)$

$a$

$h = \log_b n$

$f(n/b^2)$ $f(nb^2)$ $\cdots$ $f(n/b^2)$ ------------------------ $a^2 f(n/b^2)$

$\vdots$

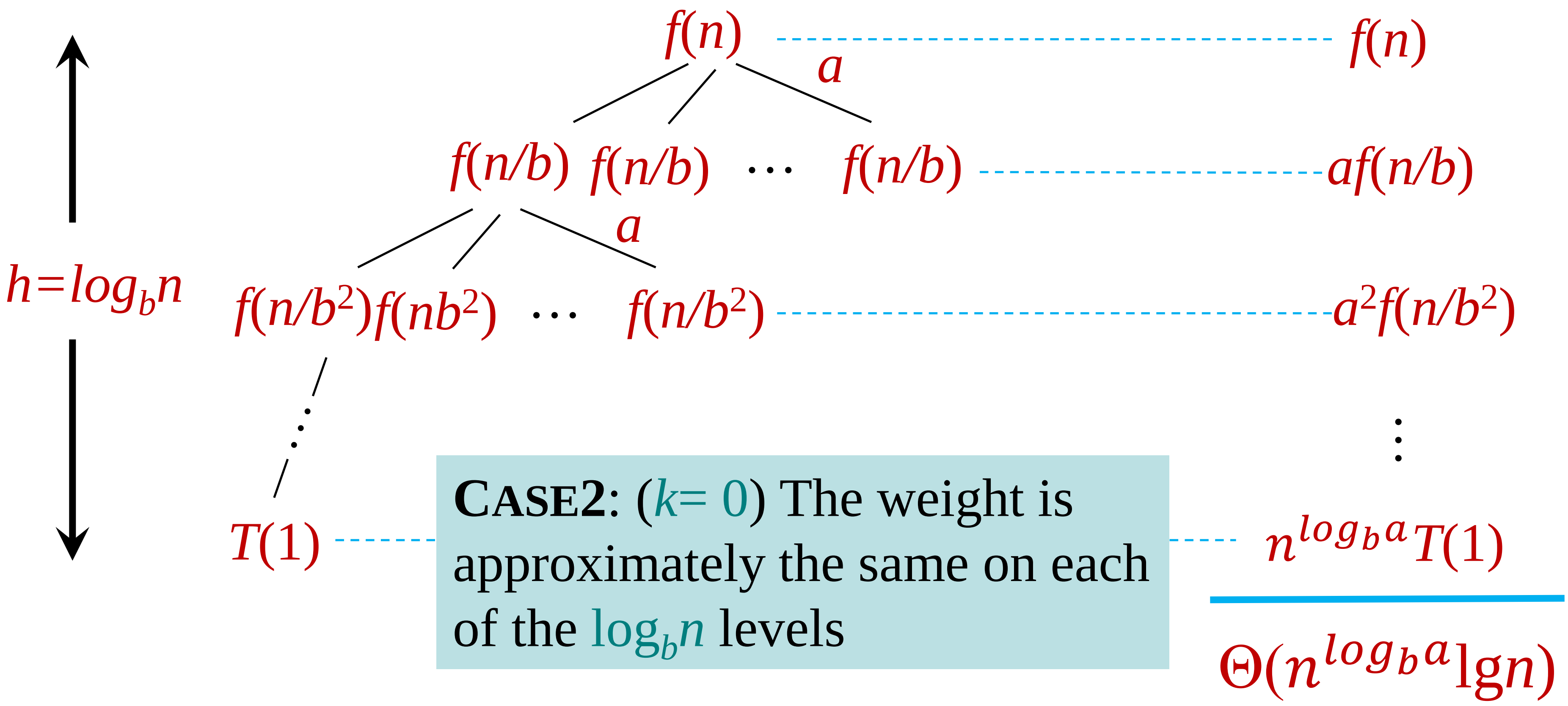
$T(1)$ ------ $n^{\log_b a} T(1)$

**CASE1**: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

$\Theta(n^{\log_b a})$

# Idea of master theorem

➢ ***Recursion tree:***




**CASE2**: ($k$= 0) The weight is approximately the same on each of the $\log_b n$ levels

# Idea of master theorem

➢ ***Recursion tree:***

$h = \log_b n$

| | |
|---|---|
| $f(n)$ (branching $a$) | $f(n)$ |
| $f(n/b)\ \ f(n/b)\ \cdots\ f(n/b)$ (branching $a$) | $af(n/b)$ |
| $f(n/b^2)\ f(nb^2)\ \cdots\ f(n/b^2)$ | $a^2 f(n/b^2)$ |
| $\vdots$ | $\vdots$ |
| $T(1)$ | $n^{\log_b a} T(1)$ |
| | $\Theta(f(n))$ |

**CASE3**: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.