Design and Analysis of Algorithms

Xin Li
East China Normal University

This course

An intermediate-level yet rigorous introduction to the design and analysis of algorithms

- Basics of algorithms and algorithm analysis
- Advanced sorting
- Graph algorithms
- Major algorithm design paradigms
- Elementary computational complexity theory
- Coping with hard problems

Course information

Instructor: 李鑫

Office: Science Building B1104

Lecture time: Tuesdays 10:00 – 11:40 am

Email: xinli@sei.ecnu.edu.cn

Office hours:

please drop me (or TAs) a line with your problem for making an appointment

Course information

Teaching assistants:

- 陈吟凯

Email: 51205902081@stu.ecnu.edu.cn

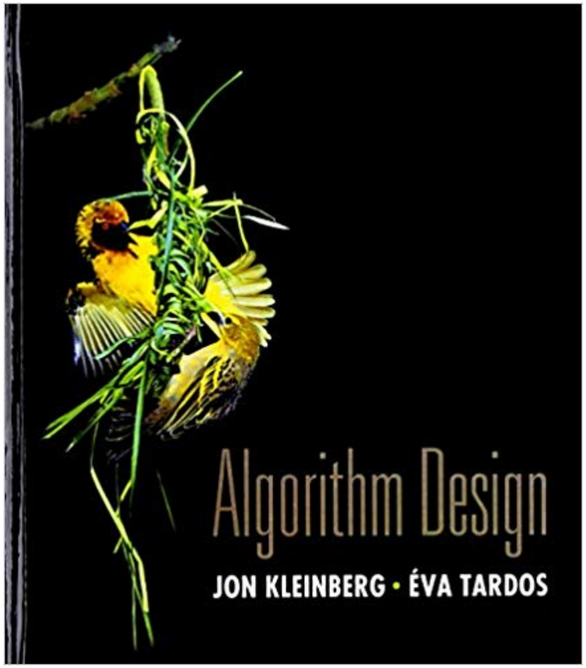
- 张奕岑

Email: 51194501210@stu.ecnu.edu.cn

References

- *Algorithm Design* by Jon Kleinberg and Éva Tardos. Addison-Wesley, 2005.
  - http://www.cs.princeton.edu/~wayne/kleinberg-tardos/
- *Introduction to Algorithms (Third Edition)* by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. MIT Press, 2009.

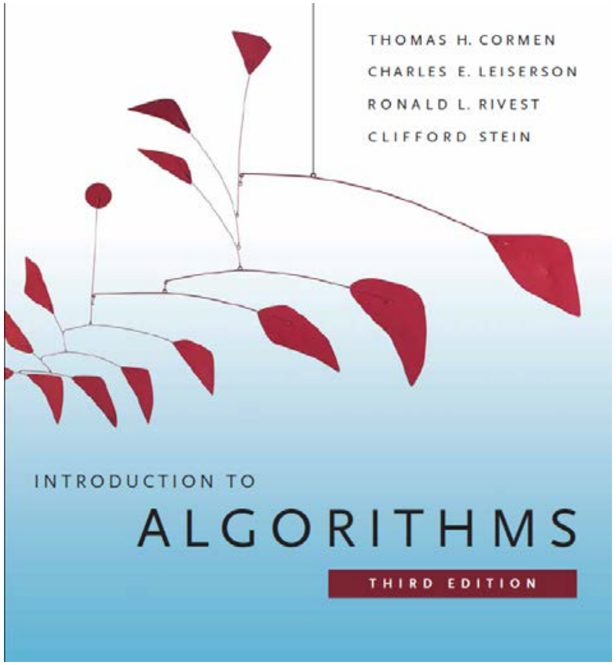
Reference: Algorithm Design



Algorithm Design by Jon Kleinberg and Éva Tardos. Addison-Wesley, 2005.

- http://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Reference: Introduction to Algorithms



Introduction to Algorithms (Third Edition) by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. MIT Press, 2009.

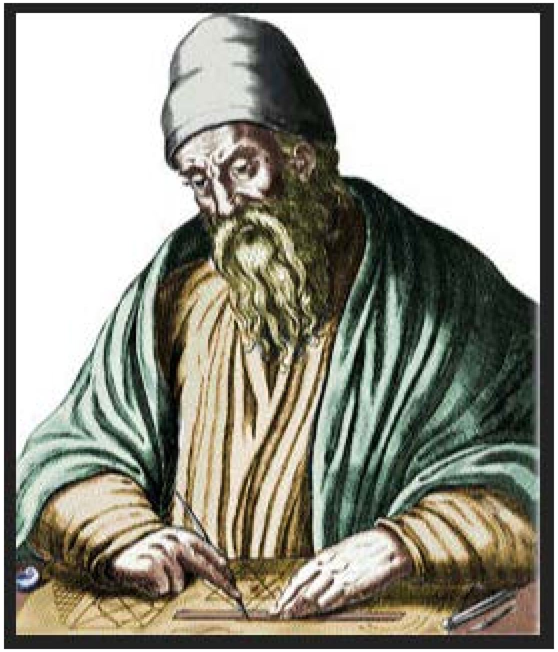
Acknowledgement

Some slides in this lecture are based on official slides that accompany the reference textbooks, provided by

- *Prof. Kevin Wayne*
- *Prof. Charles E. Leiserson*

Prologue

Euclidean algorithm



Euclid (active during c.300 B.C.)

- Greek mathematician
- Father of geometry

$$\gcd(m,n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n, m \bmod n) & \text{o.w.} \end{cases}$$

Here given that $m \geq n$

Study of algorithms at least dates back to Euclid, and one of the oldest algorithms is Euclid's method for computing the greatest common divisor of two natural numbers.

Al Khwarizmi

Muhammad Al Khwarizmi
(c.780 – 850)

- Persian mathematician
- Father of algebra and algorithm

The word "algorithm" is derived from the Latinization of his name to honor him for laying out the basic methods of

- adding, multiplying, dividing numbers
- extracting square roots, calculating digits of π, etc.

in his book on the Indian numbers, introducing the decimal system to the Western world.

Modern notion of algorithms

Alan Turing (1912-1954)

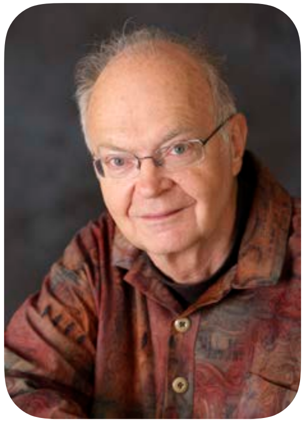
- English computer scientist, mathematician, logician, etc.
- Father of theoretical computer science and AI

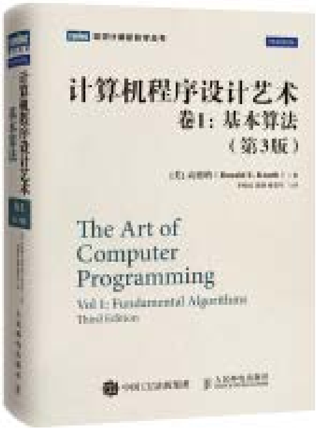
Algorithms and computation were formalized by Church (with λ-calculus) and Turing (with *Turing machine*) independently in 1930s, to answer David Hilbert's *Entscheidungsproblem* (1928) asking whether all functional calculus are solvable by some *effective method*.

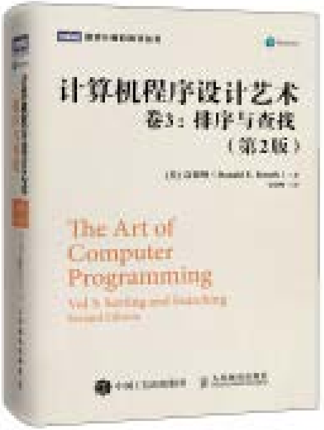
What is an algorithm?

“ *An* *algorithm* *is a finite, definite, effective procedure, with some input and some output.* ”

— *Donald E. Knuth*







TeX

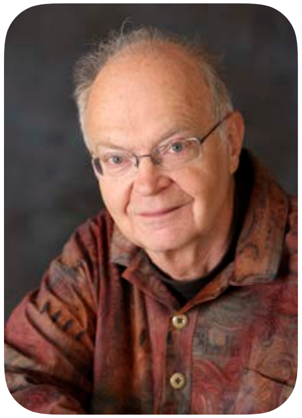
What is an algorithm?

Taking any instance of a problem as inputs, an algorithm terminates in a finite number of steps and returns a correct answer for solving the problem.

- Solve a *well-specified* computational problem
- A finite sequence of *precisely-defined* operations that transforms the input into the output
- Terminates in a *finite* number of execution steps

Why study algorithms?

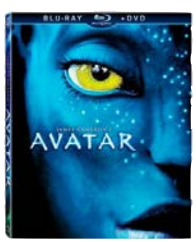
“ Algorithms are the life-blood of computer science…
the common denominator that underlies and unifies the different branches. ” — *Donald Knuth*

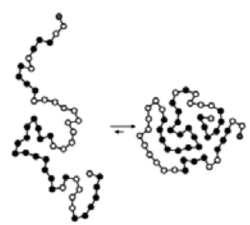


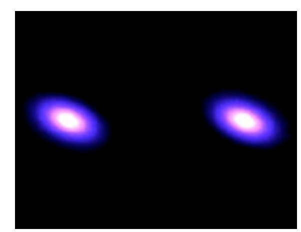
TEX

Why study algorithms?

Internet. Web search, packet routing, distributed file sharing ...
Biology. Human genome project, protein folding …
Computers. Circuit layout, databases, network, compilers …
Computer graphics. Movies, video games, virtual reality …
Security. Cell phones, e-commerce, voting machines …
Multimedia. MP3, JPG, DivX, HDTV, face recognition …
Social networks. Recommendations, news feeds, advertisements…
Physics. Particle collision simulation, *n*-body simulation …
Artificial intelligence. Decision trees, k-means, neural networks…





We emphasize algorithms and techniques useful in practice.

Example of sorting

The sorting problem

Input: a sequence $\langle a_1, \cdots, a_n \rangle$ of n numbers

Output: $\langle a_{i_1}, \ldots, a_{i_n} \rangle$ such that $a_{i_1} \leq \cdots \leq a_{i_n}$ where $\{i_{1,} \cdots, i_n\}$ is a permutation of $\{1, \cdots, n\}$.

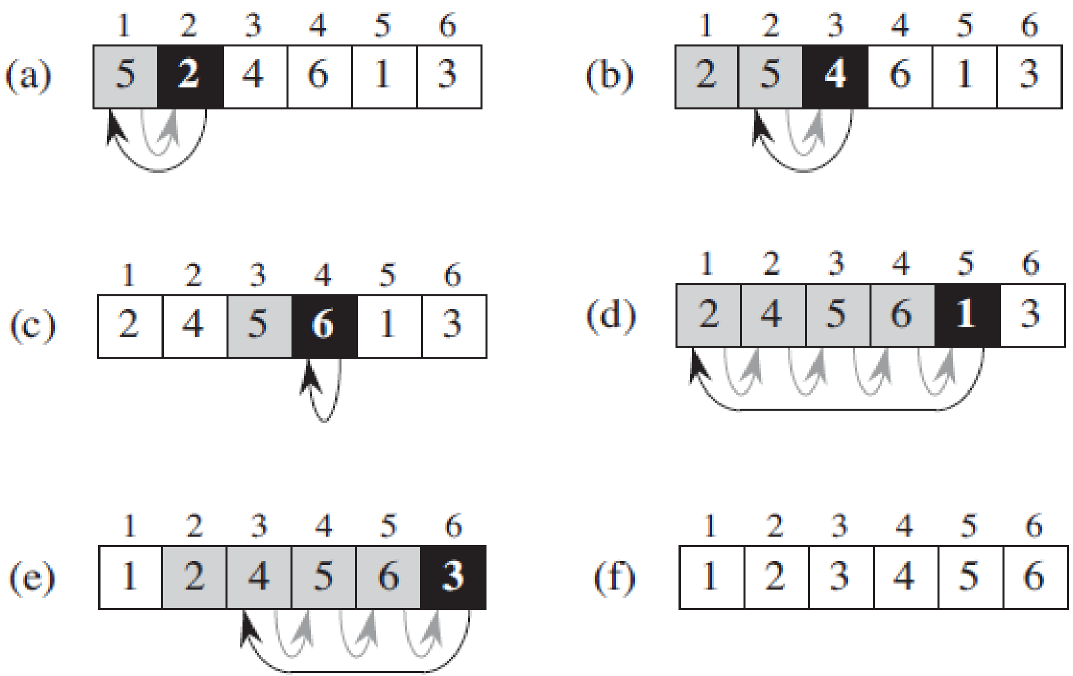
A problem instance

Input: $\langle 31, 41, 59, 26, 41, 58 \rangle$

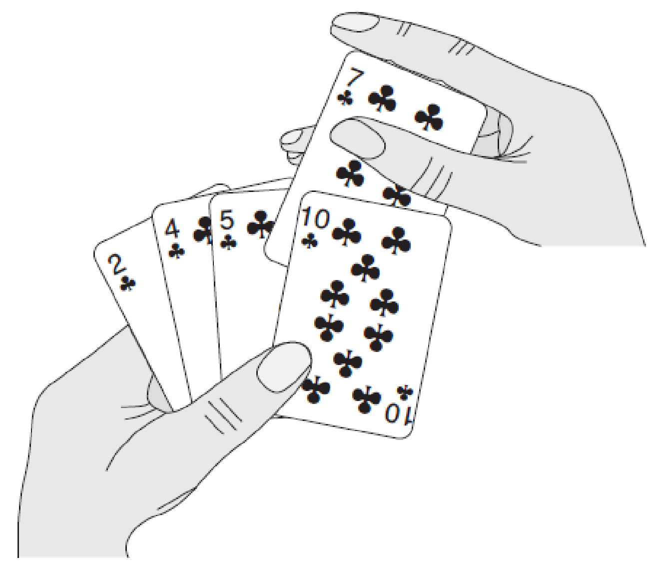
Output: $\langle 26, 31, 41, 41, 58, 59 \rangle$

Example of insertion sort

Input: $\langle 5, 2, 4, 6, 1, 3 \rangle$



Output: $\langle 1, 2, 3, 4, 5, 6 \rangle$



Pseudocode of insertion sort

```
InsertionSort (A[n])
    for j = 2 to n
        key = A[j]
        //insert A[j] into the sorted sequence A[j − 1]
        i = j − 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = key
```

Analysis of algorithms

- Is an algorithm correct?
- How much time does it take?
- How can we do it more efficiently?
- What's the best algorithm?
- Simplicity, robustness, extensibility…

Here we are concerned with *correctness, time and space performance* of algorithms

Analysis of correctness

Often use *loop invariants I + termination conditions* to help understand why an algorithm is correct

- **Initialization:** *I* is true before 1st 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, *I* gives us a useful property that helps show the algorithm is correct.

Bearing a similarity to mathematical induction

- **Base case:** $P(0)$ *holds*
- **Inductive step:** $\forall i(P(i) \rightarrow P(i+1))$ *holds*

Correctness of insertion sort

$\boldsymbol{InsertionSort}\ (\boldsymbol{A}[\boldsymbol{n}])$

for $j = 2$ **to** n

  $key = A[j]$

  //……

  $i = j - 1$

  while $i > 0$ and $A[i] > key$

    $A[i+1] = A[i]$

    $i = i - 1$

  $A[i+1] = key$

Loop invariant

$A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ yet in sorted order

Correctness of insertion sort

$\boldsymbol{InsertionSort}\ (\boldsymbol{A}[\boldsymbol{n}])$

for $j = 2$ **to** n
    $key = A[j]$
    $//\ldots\ldots$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$

Termination condition

Each loop increases j by 1 and $j = n + 1$ causing the loop to terminate

What matters to running time?

Input size

-short sequences are usually easier to sort than long ones.

Various inputs of a given size

-a sorted sequence is easier to sort for instance.

Computing resources

-clock rates, cache size, 32-bit vs 64-bit

……

Seek a machine-independent time characterization of an algorithm's efficiency and ignore machine-dependent constants

Primitive computer steps

Assume each primitive instruction (in the RAM model) and basic operation takes a constant amount of time

arithmetic (add, subtract, multiply, divide) over small numbers like 32-bit numbers

data movement (load, store, copy)

control (conditional and unconditional branch)

element comparison in sorting and searching

multiplication *of* each pair of matrix elements

……

Note that sometime one may need more refined model (e.g., for multiplying n-bit integers).

Input size

Sorting and searching. The number of input items

Multiplying integers. The total number of bits needed to represent the input in binary notation

Matrix multiplication. The number of rows and columns m, s, n for $A_{m\times s} \times B_{s\times n}$

Graph. The number of vertices and edges

……

The input size depends on the problem being studied.

The running time of insertion sort

$\boldsymbol{InsertionSort}\ (\boldsymbol{A}[\boldsymbol{n}])$	**cost**	**times**
1 **for** $j = 2$ **to** n	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad //\ldots\ldots$		
4 $\quad i = j - 1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

t_j: the number of times the while loop test (line 5) is executed for the jth iteration

The running time of insertion sort

$T(n)$ is the sum of running times for each statement

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

where t_j depends on which kind of inputs is given.

Best case. $\boldsymbol{A}[\boldsymbol{n}]$ is already sorted $t_j = 1$

Worst case. $\boldsymbol{A}[\boldsymbol{n}]$ is in reverse sorted order $t_j = j$

The running time of insertion sort

Best case. $\boldsymbol{A}[\boldsymbol{n}]$ is already sorted when $t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

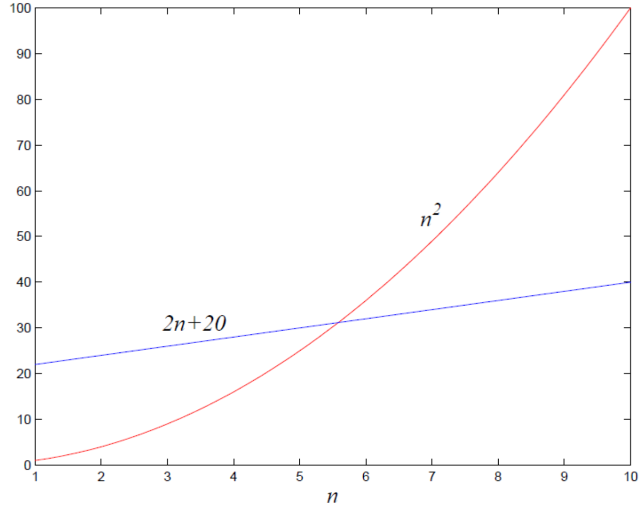
Worst case. $\boldsymbol{A}[\boldsymbol{n}]$ is in reverse sorted order when $t_j = j$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

Which running time is better?

Suppose we are choosing between two algorithms for a given computational task:

- A_1 takes $T_1(n) = n^2$ steps
- A_2 takes $T_2(n) = 2n + 20$ steps



The answer depends on n

$$\lim_{n\to\infty} \frac{2n+20}{n^2} = 0$$

T_2 scales much better as n grows, and therefore is superior

Which running time is better?

How about $T_2(n) = 2n + 20$ vs $T_3(n) = n + 1$?

Certainly, $T_3(n)$ is better than $T_2(n)$ but only by a constant factor

$$\lim_{n \to \infty} \frac{2n + 20}{n + 1} = 2$$

The discrepancy between $T_2(n)$ and $T_3(n)$ is tiny compared to the hug gap between $T_1(n)$ and $T_2(n)$

Asymptotic analysis

Count the number of *primitive operations or steps*, parameterize $T(n)$ as a function of input size n.

Consider *the order of growth* of *T(n)* as the *input size* becomes large enough.

Drop lower-order terms and ignore constant coefficients in the leading terms.

Ex. Just say that insertion sort has a worst-case running time of $O(n^2)$ (“big theta of n-squared”).

Apply to analyze other aspects of algorithms like space.

Space complexity

The number of *auxiliary* memory cells an algorithm needs to run, usually do not count the following memory

- taken by input/output that are irrelevant to the algorithm
- for storing the algorithm itself that is usually fixed

Space complexity is also a function of input size n.

In-place algorithm. only use a constant amount of extra space (e.g., $O(1)$ for the insertion sort).

Time-space-tradeoff. one needs a compromise.

Asymptotic analysis

Three kinds of common analyses

Worst-case

- $T(n)$: maximum time of an algorithm on any input of some size n

Best-case

- $T(n)$: cheat with a slow algorithm that works fast on some input

Average-case

- $T(n)$: expected time of an algorithm over all inputs of some size n

- Need assumption of statistical distribution of inputs

Worst-case analysis

Worst case analysis. Running time guarantee for any input of a given size *n*.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

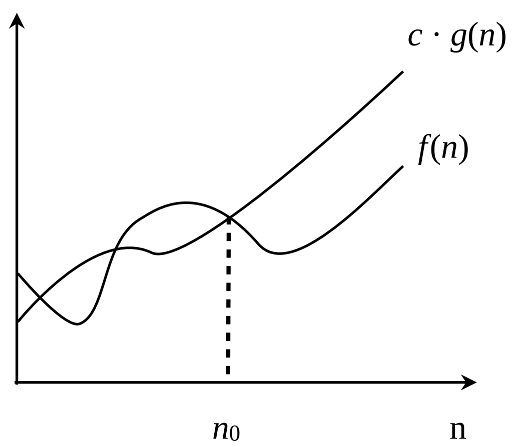
What about average-case analysis? Very hard to generate "random" input instances and need to consider the statistical distributions of inputs.

Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances don't arise.

- simplex algorithm, Linux grep, k-means algorithm, etc.

Big O notation

Upper bounds. $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $O(n^2)$. ⟵ choose $c = 50$, $n_0 = 1$
- $f(n)$ is neither $O(n)$ nor $O(n \log n)$.

Typical usage. Insertion sort makes $O(n^2)$ compares to sort n elements in the worst case.

Big O notation

One-way "equality." $O(g(n))$ is *a set of functions*, often written as $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

Ex. Consider $g_1(n) = 5n^3$ and $g_2(n) = 3n^2$.

- We have $g_1(n) = O(n^3)$ and $g_2(n) = O(n^3)$.
- But, do not conclude $g_1(n) = g_2(n)$.

Domain & codomain. f and g are real-valued functions.

- The domain is typically the natural numbers: $\mathbb{N} \to \mathbb{R}$.
- Sometimes we extend to the reals: $\mathbb{R}_{\geq 0} \to \mathbb{R}$.

Big O properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then $c\, f$ is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.

Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

(Here, we abbreviate $f(n)$ and $g(n)$ by f and g.)

Ex. $f(n) = 5n^3 + 3n^2 + n + 1234$ is $O(n^3)$.

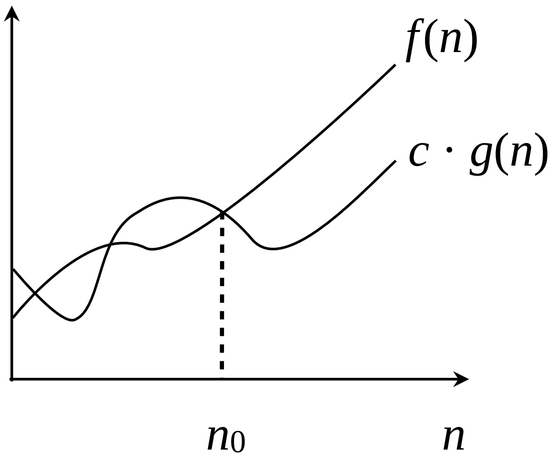
Big Omega notation

Lower bounds. $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.

Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is both $\Omega(n)$ and $\Omega(n^2)$.
- $f(n)$ is not $\Omega(n^3)$.

choose $c = 32$, $n_0 = 1$



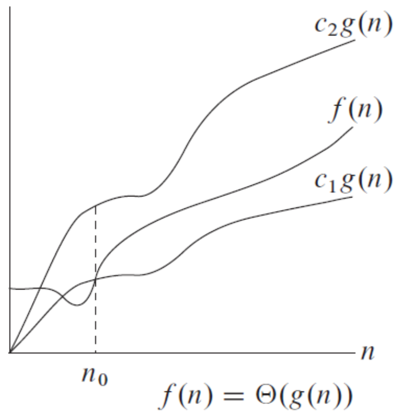
Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

Big Theta notation

Tight bounds. $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $\Theta(n^2)$.
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.



Typical usage. Mergesort makes $\Theta(n \log n)$ compares to sort n elements in the worst case.

Big O notation with multiple variables

Upper bounds. $f(m, n)$ is $O(g(m, n))$ if there exist constants $c > 0$, $m_0 \geq 0$, and $n_0 \geq 0$ such that $f(m, n) \leq c \cdot g(m, n)$ for all $n \geq n_0$ and $m \geq m_0$.

Ex. $f(m, n) = 32mn^2 + 17mn + 32n^3$.

- $f(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $f(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Typical usage. Breadth-first search takes $O(m + n)$ time to find a shortest path from s to t in a digraph with n nodes and m edges.

Some useful theorems

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c > 0$, then $f(n) = \Theta(g(n))$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n))$

$f(n)$ is $o(g(n))$ if for any constant $c > 0$, there exists constant $n_0 \geq 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) = \omega(g(n))$

$f(n)$ is $\omega(g(n))$ if for any constant $c > 0$, there exists constant $n_0 \geq 0$ such that $f(n) > c \cdot g(n) \geq 0$ for all $n \geq n_0$.

Some useful theorems

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c$ for some constant $c > 0$, then $f(n) = \Theta(g(n))$

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Omega(g(n))$.

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty$, then $f(n)$ is $\Omega(g(n))$ but not $O(g(n))$.

Proof

By definition, for any arbitrarily small $\varepsilon = c/2$, there exists some n_0, when $n \geq n_0$, we have

$$\left|\frac{f(n)}{g(n)} - c\right| < \varepsilon \Rightarrow c - \varepsilon < \frac{f(n)}{g(n)} < c + \varepsilon$$

$$\Rightarrow \frac{c}{2} < \frac{f(n)}{g(n)} < \frac{3c}{2} < 2c$$

(1) for any $n \geq n_0$, $f(n) \leq 2cg(n)$. Then we have $f(n) = O(g(n))$

(2) For any $n \geq n_0$, $f(n) \geq (c/2)g(n)$. Then we have $f(n) = \Omega(g(n))$.

It follows that $f(n) = \Theta(g(n))$

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Logarithms.

- $\log_a n = \Theta(\log_b n)$ for any constants $a, b > 1$.

can avoid specifying the base

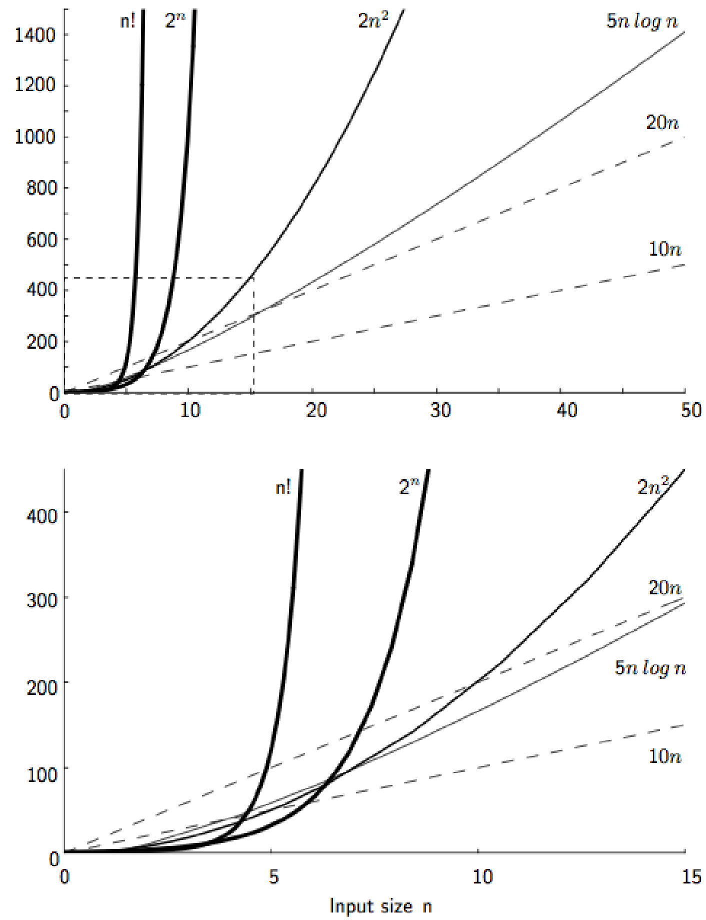
- For any $x > 0$, $a > 1$, $\log_a n = O(n^x)$.

log grows slower than every polynomial

Exponentials. For any $r > 1$ and any $d > 0$, $n^d = O(r^n)$.

every exponential grows faster than every polynomial

Example of growth rate graph



Why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Factorials

Stirling formula. $$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$n! = o(n^n)$

$n! = \Omega(2^n)$

$\log(n!) = \Theta(n\log n)$

$$\lim_{n\to+\infty} \frac{\log(n!)}{n\log n} = \lim_{n\to+\infty} \frac{\ln(n!)/\ln 2}{n\ln n/\ln 2} = \lim_{n\to+\infty} \frac{\ln(n!)}{n\ln n}$$

$$= \lim_{n\to+\infty} \frac{\ln\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \left(\frac{c}{n}\right)\right)\right)}{n\ln n} = \lim_{n\to+\infty} \frac{\ln\sqrt{2\pi n} + n\ln\frac{n}{e}}{n\ln n} = 1$$

Here c is some constant.

A survey of
common running times

Constant time - $O(1)$

Constant time. Bounded by a constant which does not depend on input size n.

Examples

- Conditional branch.
- Arithmetic/logic operation.
- Declare/initialize a variable.
- Follow a link in a linked list.
- Access element i in an array.
- Compare/exchange two elements in an array.

…

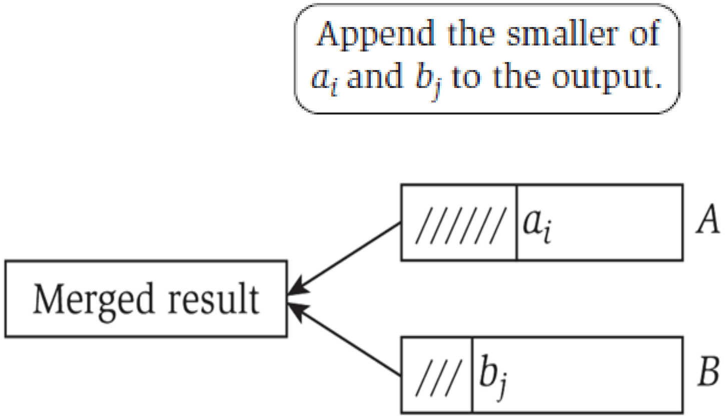
Linear time - $O(n)$

Linear time. Running time is proportional to input size.

Computing the maximum. Compute the maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a₁
for i = 2 to n {
   if (aᵢ > max)
      max ← aᵢ
}
```

Merging two sorted arrays. Combine two sorted arrays $A[1..n]$ and $B[1..n]$ into a sorted one.



Merge demo

Given two sorted lists *A* and *B*, merge them into a sorted list *C*.

`sorted list A`

3	7	10	14	18

`sorted list B`

2	11	16	20	23

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 2

sorted list C

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 3

sorted list C

2									

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 7

sorted list C

2	3								

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 10

sorted list C

2	3	7							

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 11

sorted list C

2	3	7	10						

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 14

sorted list C

2	3	7	10	11					

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 16

sorted list C

2	3	7	10	11	14				

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 18

sorted list C

2	3	7	10	11	14	16			

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 20

sorted list C

2	3	7	10	11	14	16	18		

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Compare minimum entry in each list: copy 23

sorted list C

2	3	7	10	11	14	16	18	20	

Merge demo

sorted list A

3	7	10	14	18

sorted list B

2	11	16	20	23

Done!

sorted list C

2	3	7	10	11	14	16	18	20	23

Logarithmic time - $O(\log n)$

Search in a sorted array. Given a sorted array A of n distinct integers and an integer x, find index of x in the array.

O(log n) algorithm. Binary search.

Compare key against middle entry.

$lo \leftarrow 1$; $hi \leftarrow n$
WHILE $(lo \leq hi)$
  $mid \leftarrow \lfloor (lo + hi) / 2 \rfloor$
  IF $(x < A[mid])$ $hi \leftarrow mid - 1$
  ELSE IF $(x > A[mid])$ $lo \leftarrow mid + 1$
  ELSE RETURN mid
RETURN -1

Invariant: If x is in the array, then x is in A[lo .. hi].

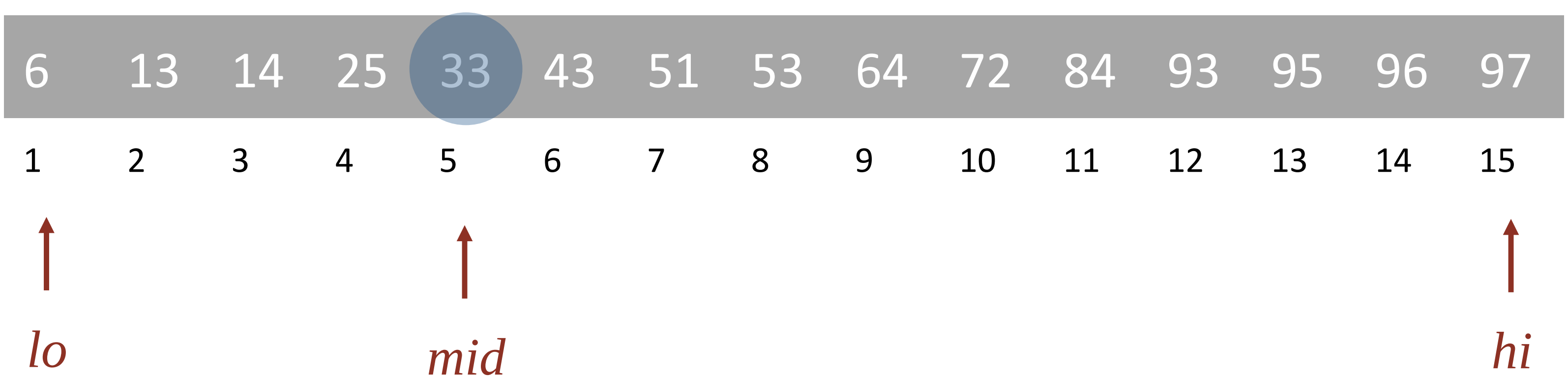
After k iterations of WHILE loop,
$(hi - lo + 1) \leq n / 2^k$
$\Rightarrow k \leq 1 + \log_2 n$.

Binary search demo

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

`Successful search for 33`



Binary search demo

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

`Successful search for 33`

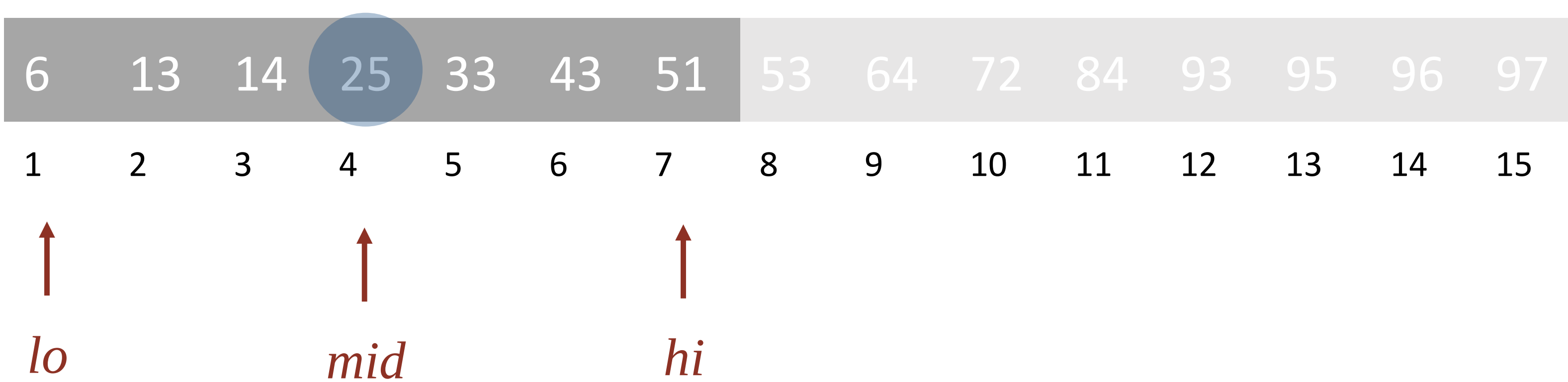
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
lo							*mid*							*hi*

Binary search demo

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

`Successful search for 33`

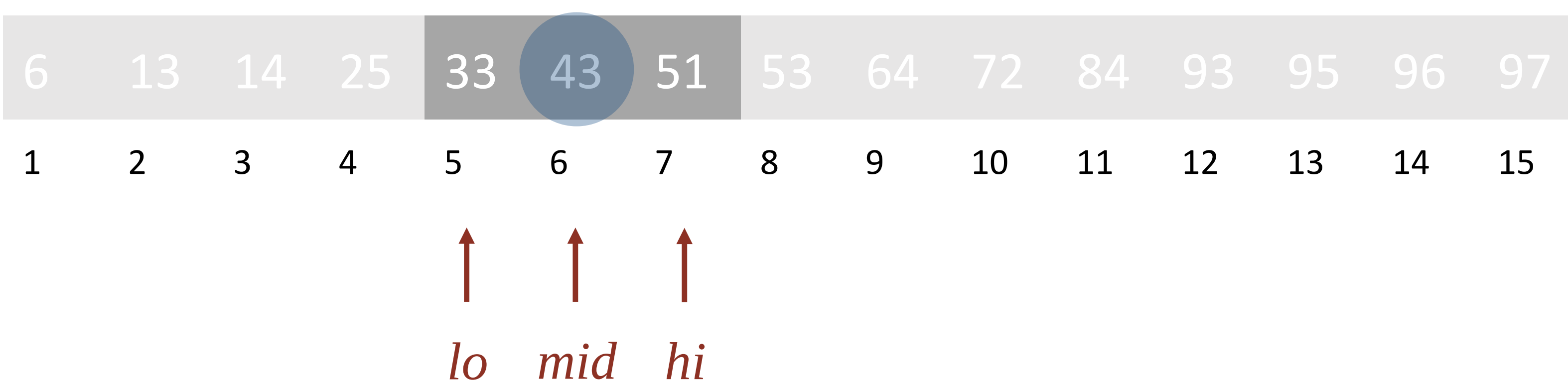


Binary search demo

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

`Successful search for 33`



Binary search demo

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

`Successful search for 33`

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

lo, mid, hi

Linearithmic time - $O(n \log n)$

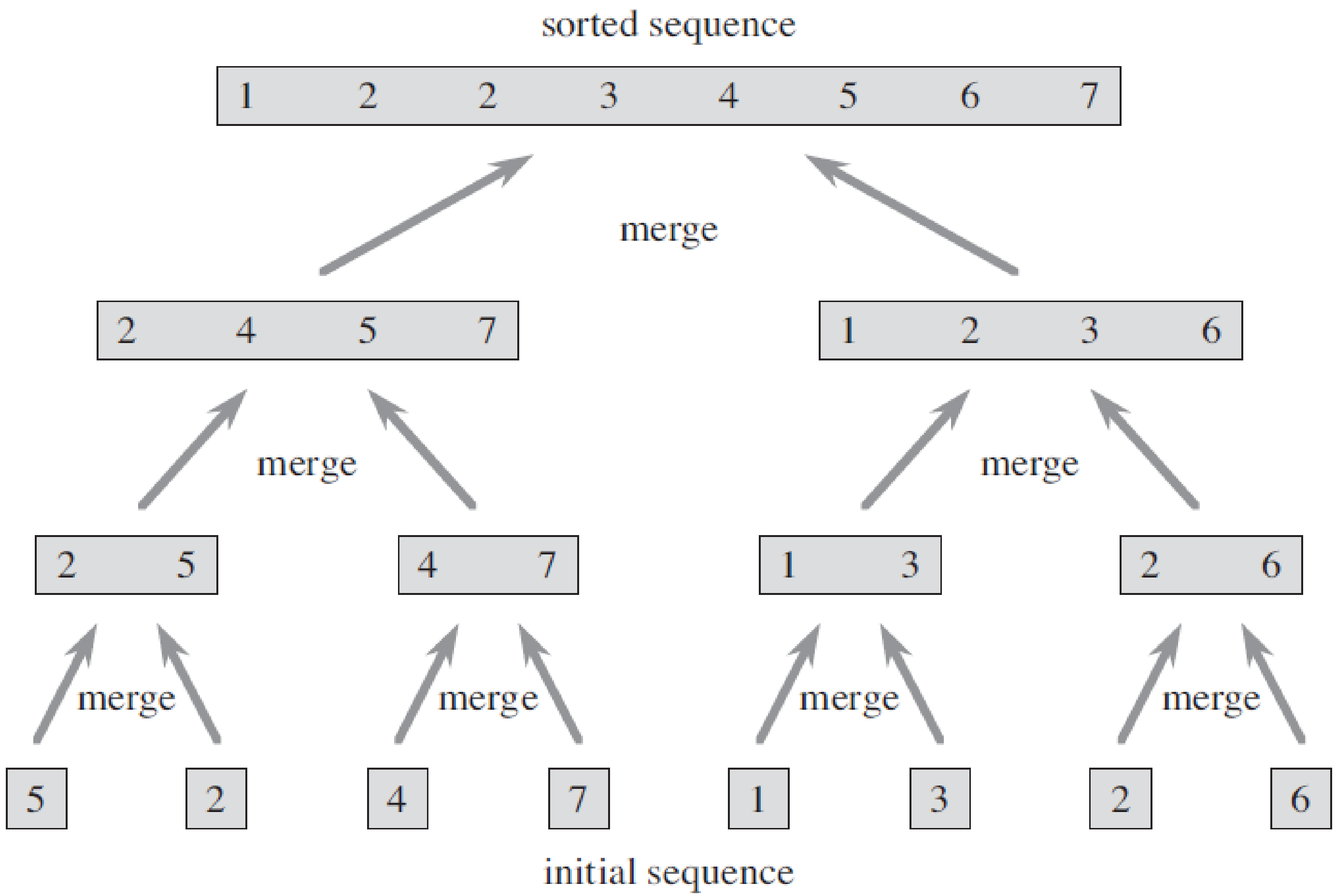
O(n log n) algorithm of sorting. *MergeSort*

Divide-and-conquer paradigm

- Divide $A[n]$ into two subarray of $n/2$ elements each
- Sort the two subarray recursively using *MergeSort*
- *Merge* the two sorted subarray to a sorted whole $\leftarrow O(n)$

$\boldsymbol{MergeSort(A[p..r])}$
    IF $\boldsymbol{p < r}$
        $\boldsymbol{q = \lfloor (p+r)/2 \rfloor}$
        $\boldsymbol{MergeSort(A[p..q])}$
        $\boldsymbol{MergeSort(A[q+1..r])}$
        $\boldsymbol{Merge(A, p, q, r)}$

Merge sort demo



Analyzing merge sort

$\boldsymbol{MergeSort}(\boldsymbol{A}[\boldsymbol{p}..\boldsymbol{r}])$

IF $\boldsymbol{p} < \boldsymbol{r}$

$\boldsymbol{q} = \lfloor(\boldsymbol{p}+\boldsymbol{r})/\mathbf{2}\rfloor$

$\boldsymbol{MergeSort}(\boldsymbol{A}[\boldsymbol{p}..\boldsymbol{q}])$ ← $T(n/2)$

$\boldsymbol{MergeSort}(\boldsymbol{A}[\boldsymbol{q}+\mathbf{1}..\boldsymbol{r}])$ ← $T(n/2)$

$\boldsymbol{Merge}(\boldsymbol{A},\boldsymbol{p},\boldsymbol{q},\boldsymbol{r})$ ← $\Theta(n)$

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

Recursion-tree method. Converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

Recursion tree

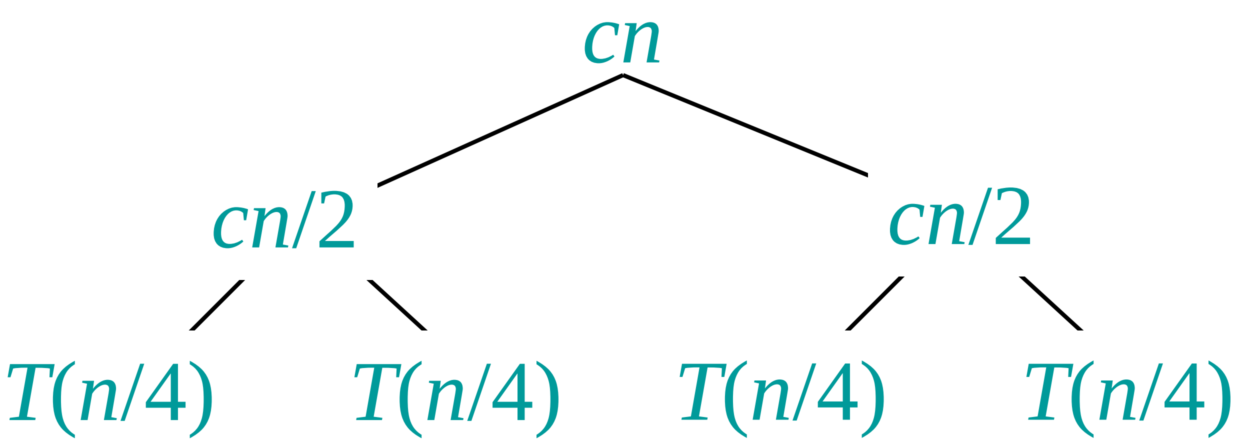
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$cn$$

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

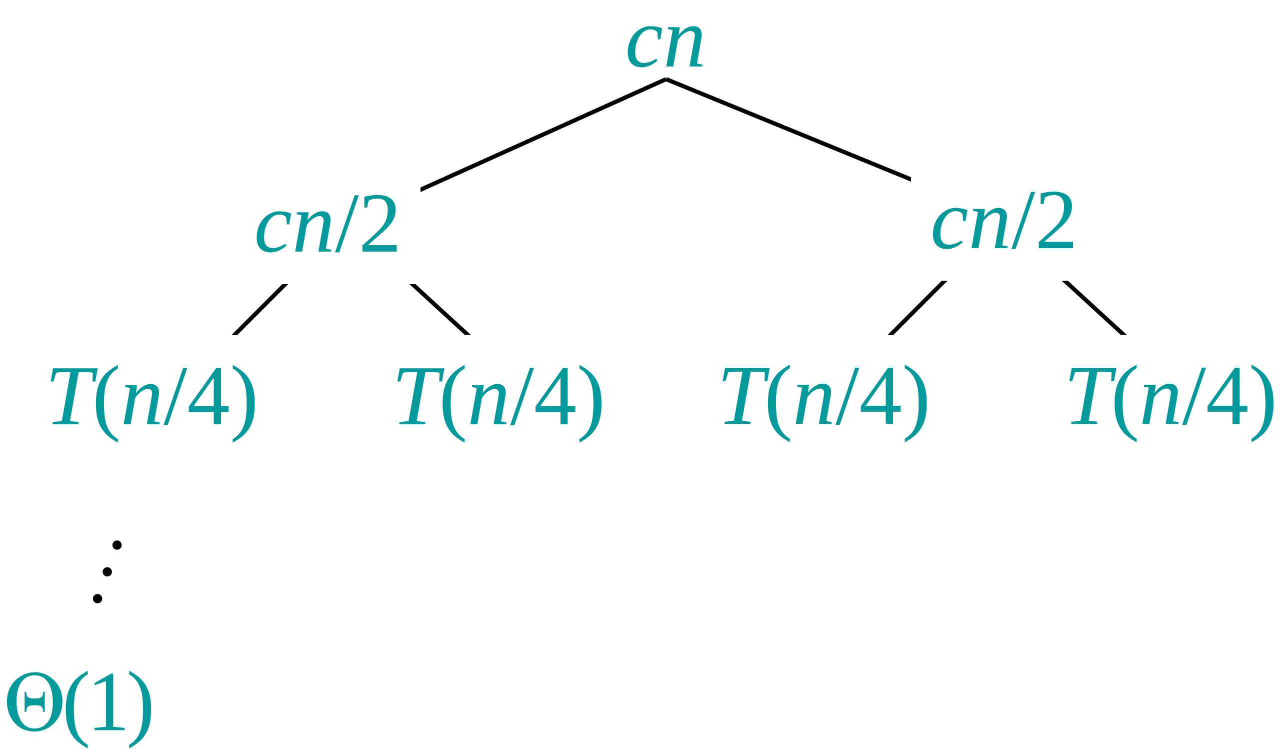
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



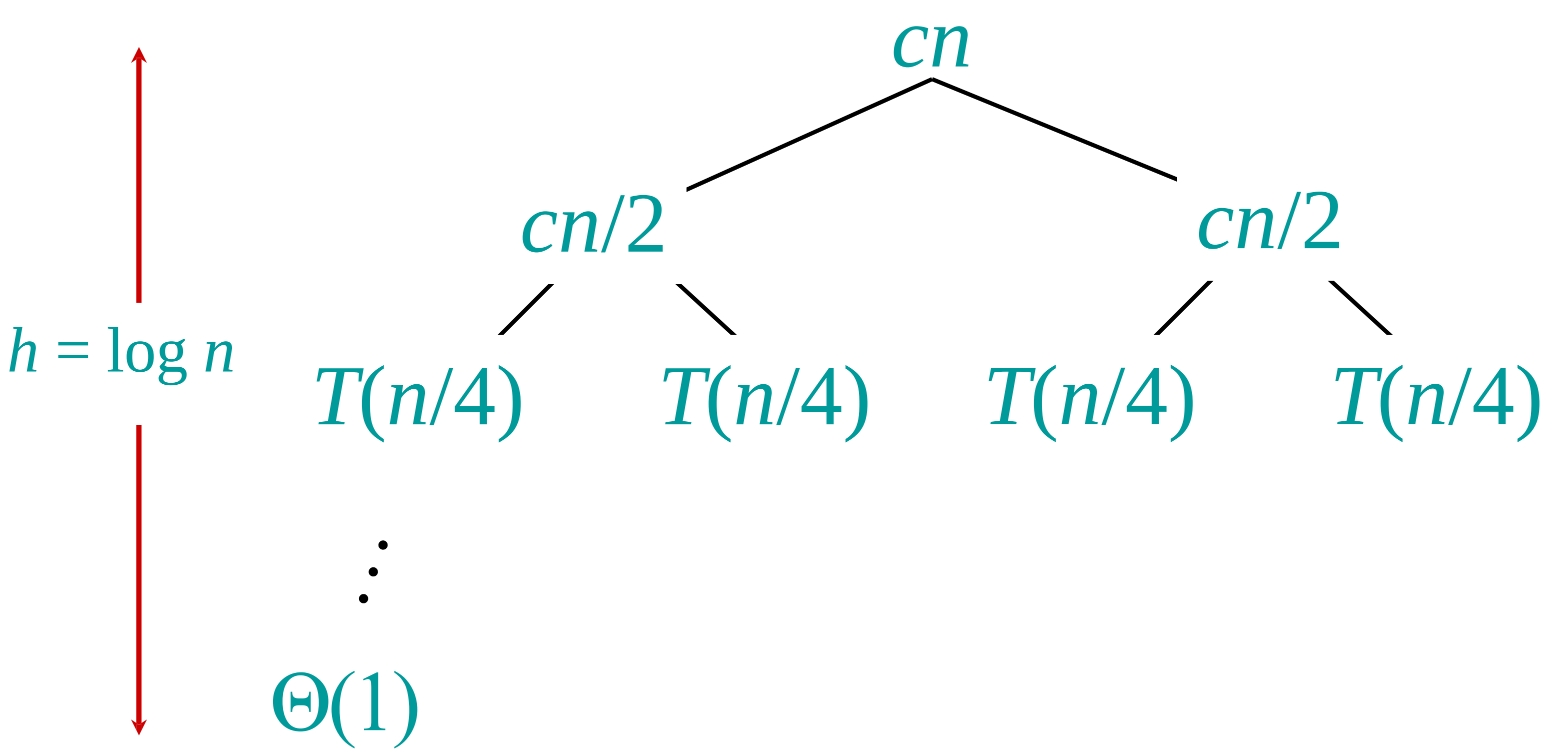
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



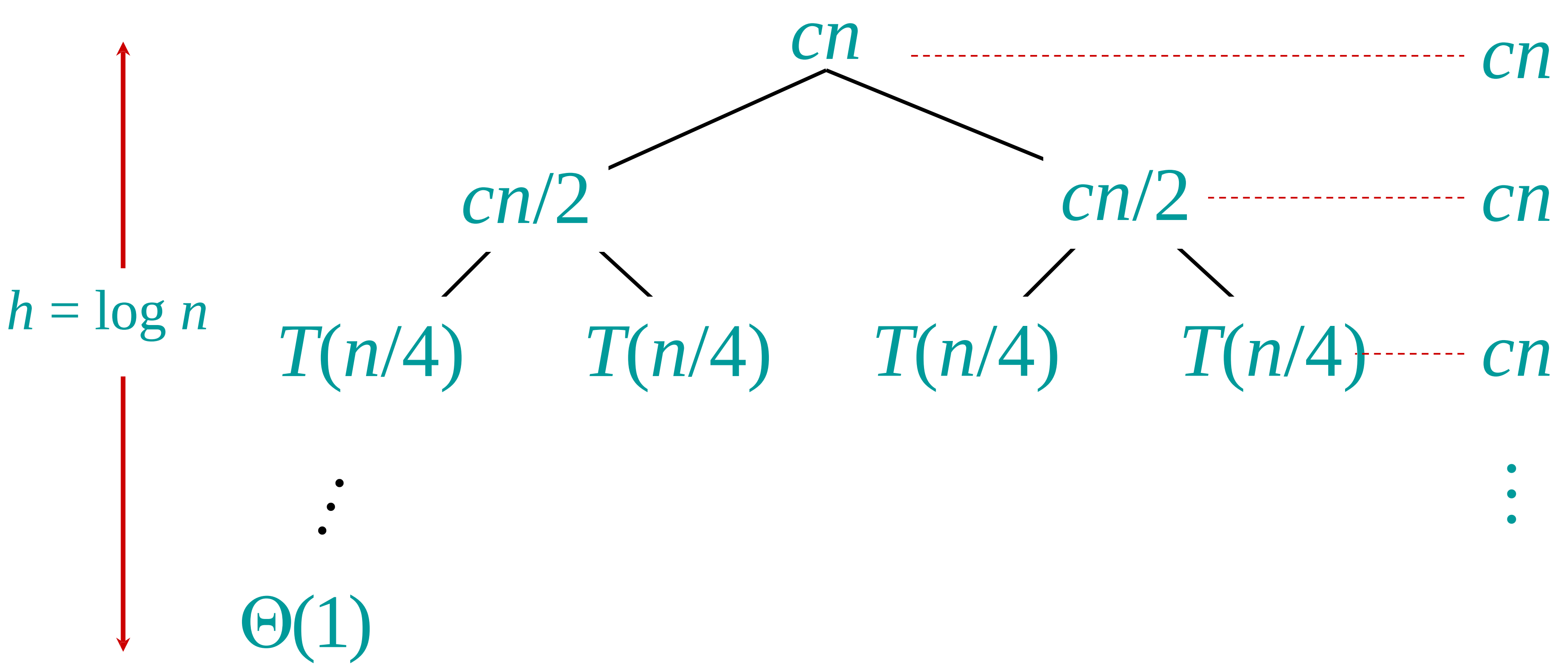
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



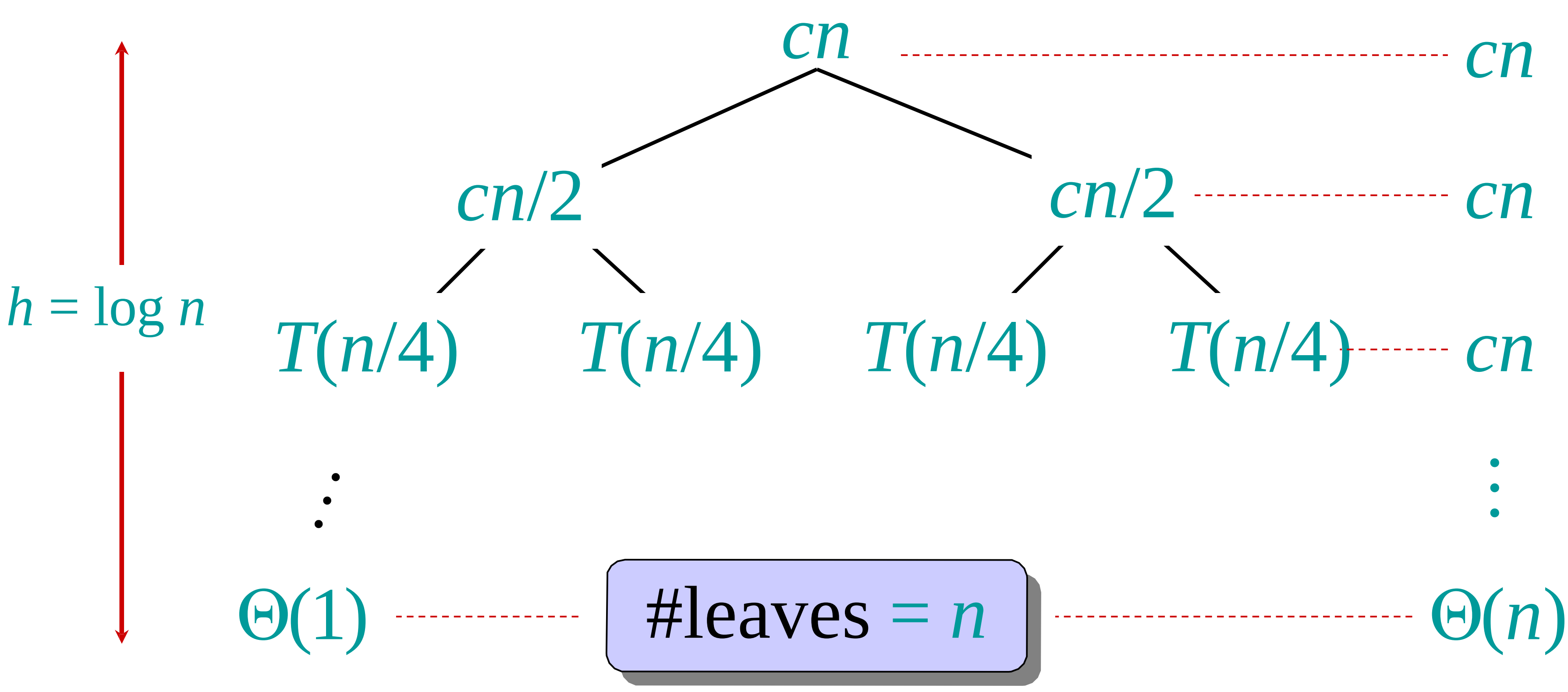
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



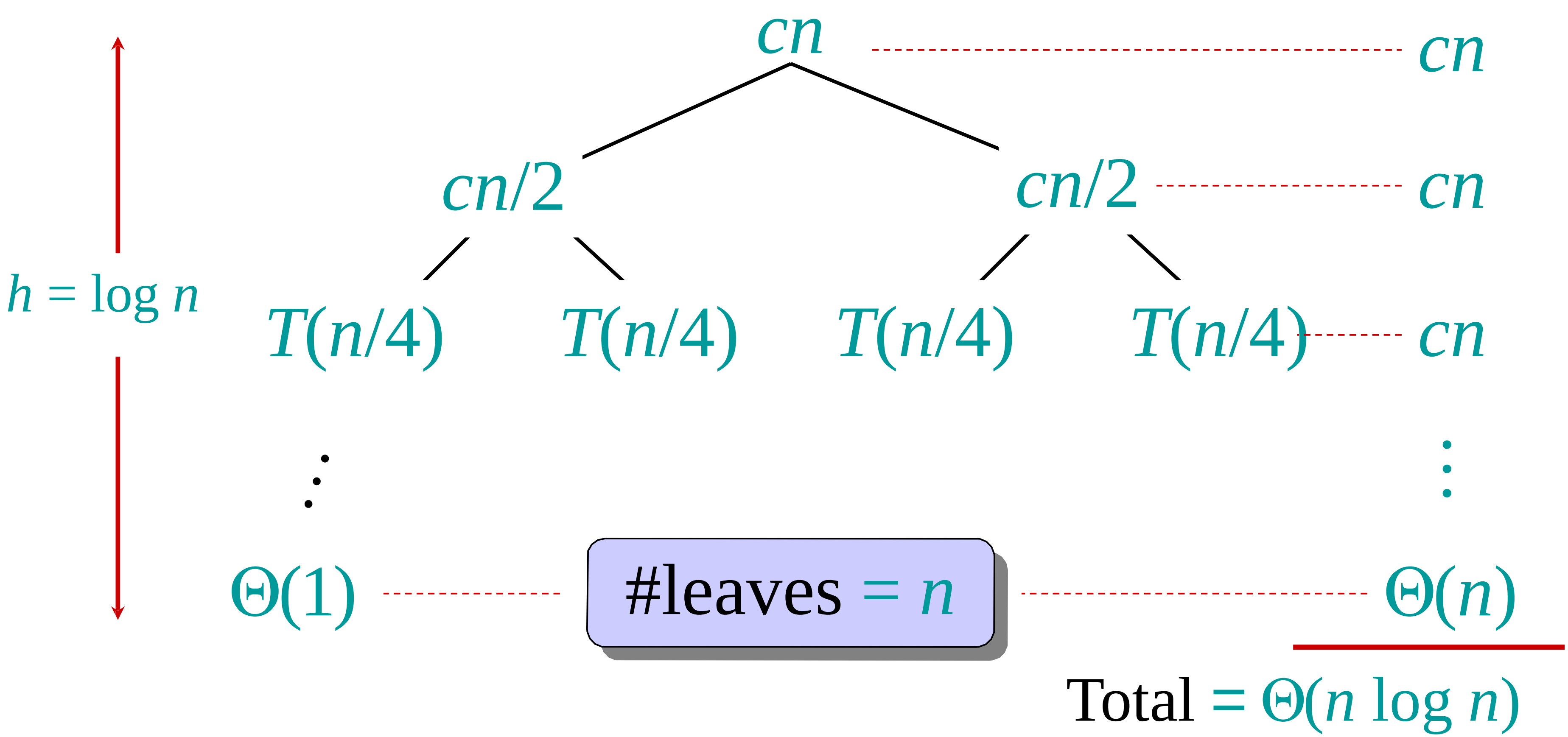
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Quadratic time - $O(n^2)$

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair closest to each other.

$O(n^2)$ algorithm. Enumerate all pairs of points (with $i < j$).

brute force

```
min ← ∞.
FOR i = 1 TO n
    FOR j = i + 1 TO n
        d ← (xi − xj)^2 + (yi − yj)^2.
            IF (d < min)
                min ← d.
```

Remark. $O(n^2)$ seems inevitable, but this is just an illusion.

It can be done in $O(n \log n)$ time and even better.

Cubic time

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

O(n^3) solution. For each pairs of sets, determine if they are disjoint.

```
FOREACH set Si {
   FOREACH other set Sj {
      FOREACH element p of Si {
         determine whether p also belongs to Sj
      }
      IF (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
}}
```

Polynomial time

Running time is $O(n^k)$ for some constant $k > 0$.

Independent set of size k. Given a graph, find k nodes such that no two are joined by an edge.

$O(n^k)$ algorithm. Enumerate all subsets of k nodes.

- Check whether S is an independent set of size k takes $O(k^2)$ time.
- Number of k-element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2) \times \cdots \times (n-k+1)}{k(k-1)(k-2) \times \cdots \times 1} \leq \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

```
FOREACH subset S of k nodes:
    Check whether S is an independent set.
    IF (S is an independent set)
        RETURN S.
```

Exponential time

Running time is $O(2^{n^k})$ for some constant $k > 0$.

Independent set. Given a graph, find a independent set of max cardinality.

$O(n^2 2^n)$ algorithm. Enumerate all subsets.

$S^* \leftarrow \varnothing$.
FOREACH subset S of nodes:
    Check whether S is an independent set.
    IF (S is an independent set and $|S| > |S^*|$)
        $S^* \leftarrow S$.
RETURN S^*.

Polynomial running time

Desirable scaling property. When the input size doubles, the algo. should slow down by at most some constant factor *c*.

There exist constants $c > 0$ and $d > 0$ such that, for every input of size n, the running time of the algorithm is bounded above by $c\,n^d$ primitive computational steps.

Polynomial-time algorithm. We say that an algorithm is polynomial time if the above scaling property holds.

We say that an algorithm is efficient if it has a polynomial running time.

Polynomial running time

Computation model independent. The notion is (relatively) insensitive to the model of computation that may have different notion of *primitive computational steps*.

- Any polynomial-time bound has the scaling property we are looking for.
- Lower-degree polynomials scales better than higher-degree ones

It works in practice. The poly-time algorithms that people develop have both small constants and small exponents.

Some poly-time algorithms in the wild have galactic constants and/or huge exponents.

Q. Which would you prefer: $20\,n^{120}$ or $n^{1\,+\,0.02\,\ln n}$?

Solving recurrences

Recap merge sort

$MergeSort(A[p..r])$

IF $p < r$

$q = \lfloor (p+r)/2 \rfloor$

$MergeSort(A[p..q])$ $T(n/2)$

$MergeSort(A[q+1..r])$ $T(n/2)$

$Merge(A, p, q, r)$ $\Theta(n)$

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

Solving recurrences

1. Substitution method
2. Recursion tree
3. Master theorem

Substitution method

The most general method:

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

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

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

whenever $(c/2)n^3 - n \ge 0$, for example, if $c \ge 2$ and $n \ge 1$.

residual

Example (continued)

- 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.

Example (continued)

- 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 $1 \leq n < n_0$, we have "$\Theta(1)$" $\leq cn^3$, if we pick c big enough.

This bound is not tight!

A tighter upper bound?

We shall prove that $T(n) = O(n^2)$.

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 $T(k) \le ck^2$ for $k < n$:

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^2 + n \\
&= cn^2 + n \\
&= \cancel{O(n^2)} \quad \textbf{\textit{Wrong!}} \text{ We must prove the I.H.} \\
&= cn^2 - (-n) \quad [\text{ desired} - \text{residual }] \\
&\le cn^2 \quad \text{for } \textbf{\textit{no}} \text{ choice of } c > 0. \text{ Lose!}
\end{aligned}
$$

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4\big(c_1(n/2)^2 - c_2(n/2)\big) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\le c_1 n^2 - c_2 n \quad \text{if } c_2 \ge 1. \end{aligned}$$

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 $k < n$.

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

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

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

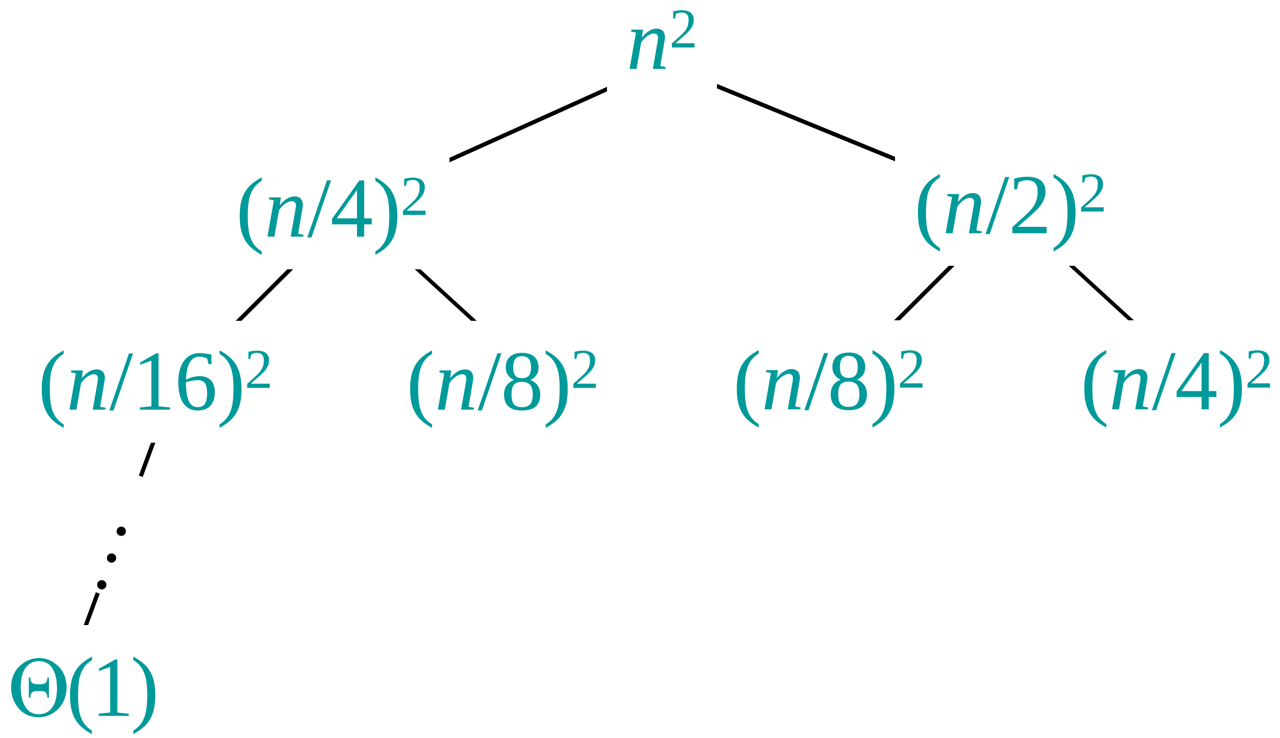
$$n^2$$

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

$$T(n/16) \quad T(n/8) \qquad T(n/8) \quad T(n/4)$$

Example of recursion tree

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



Example of recursion tree

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

$$n^2 \quad \cdots\cdots \quad n^2$$

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

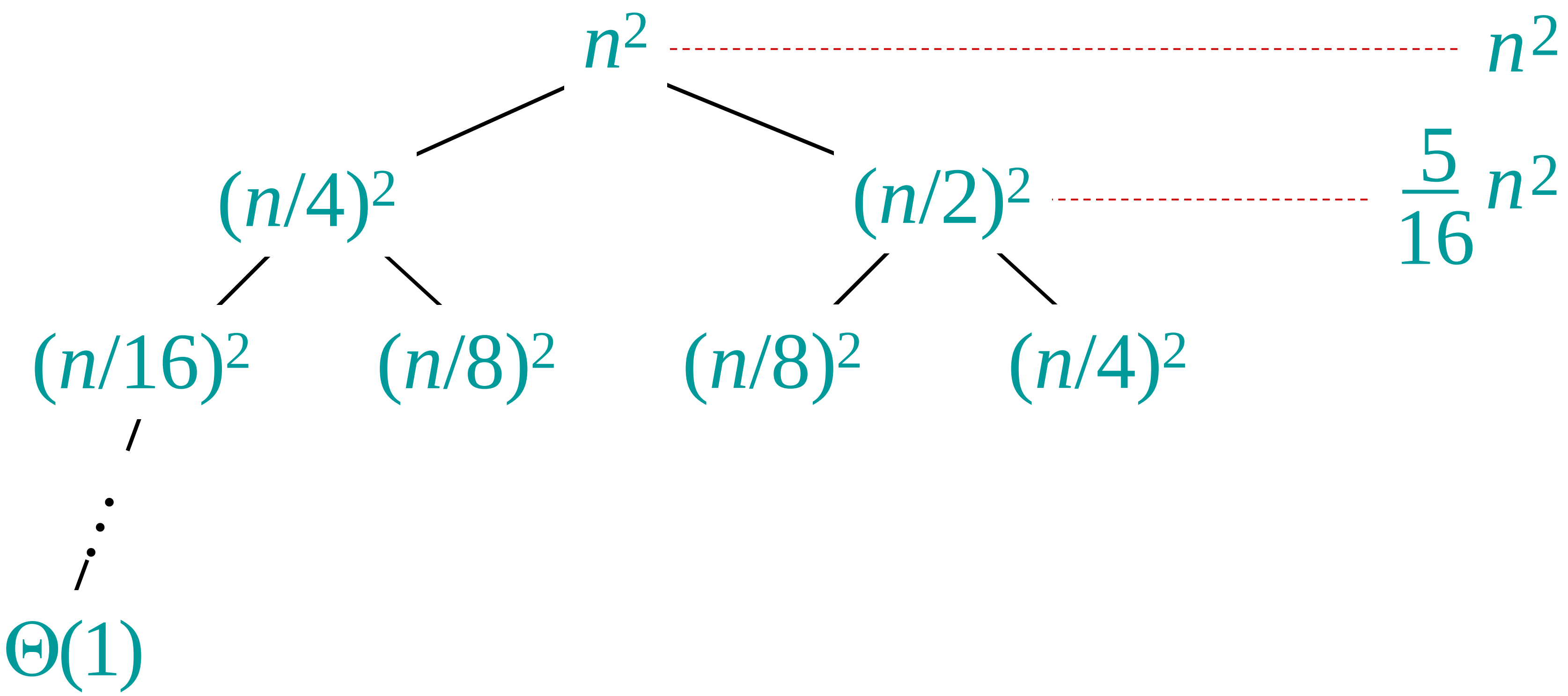
$$(n/16)^2 \qquad (n/8)^2 \qquad (n/8)^2 \qquad (n/4)^2$$

$$\vdots$$

$$\Theta(1)$$

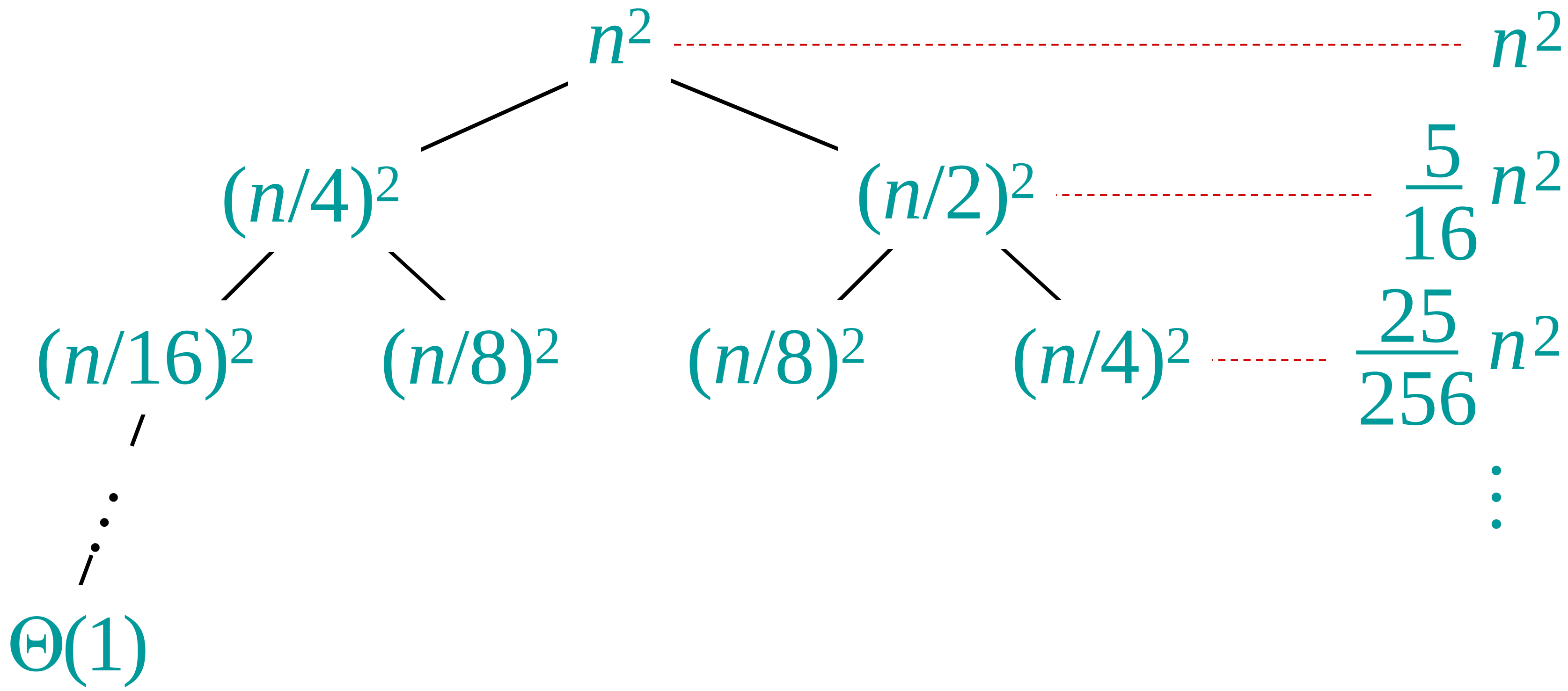
Example of recursion tree

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



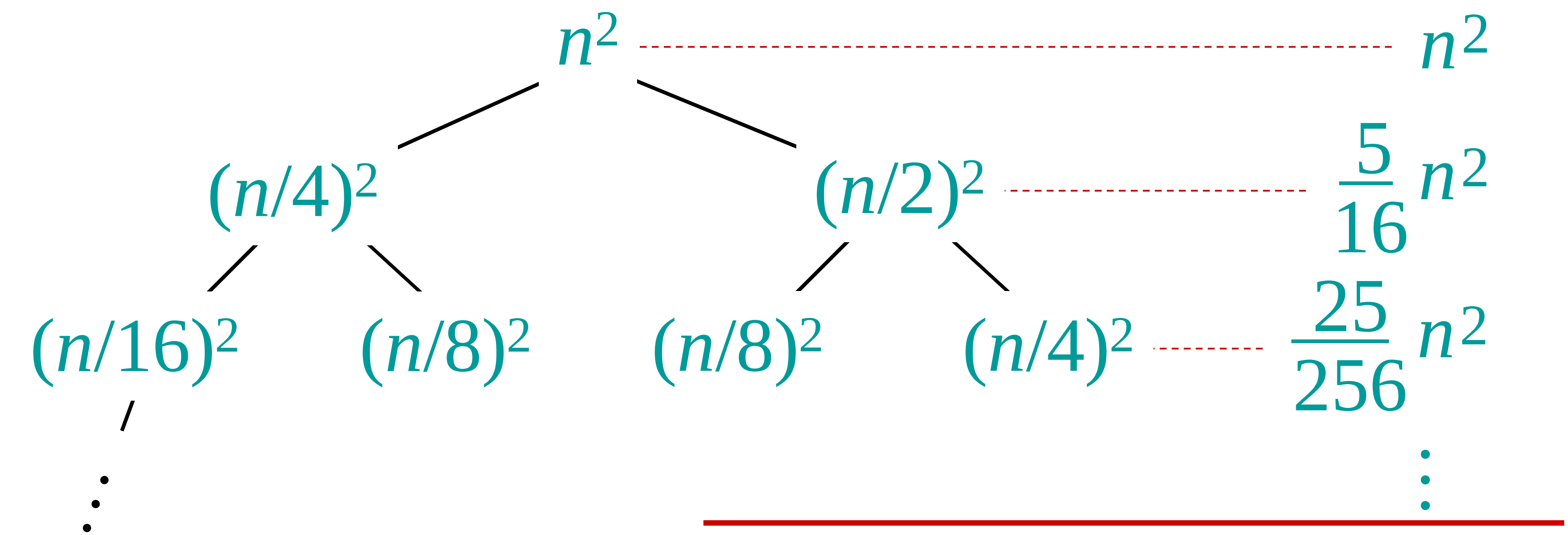
Example of recursion tree

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



Example of recursion tree

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



$\Theta(1)$

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

$= \Theta(n^2)$

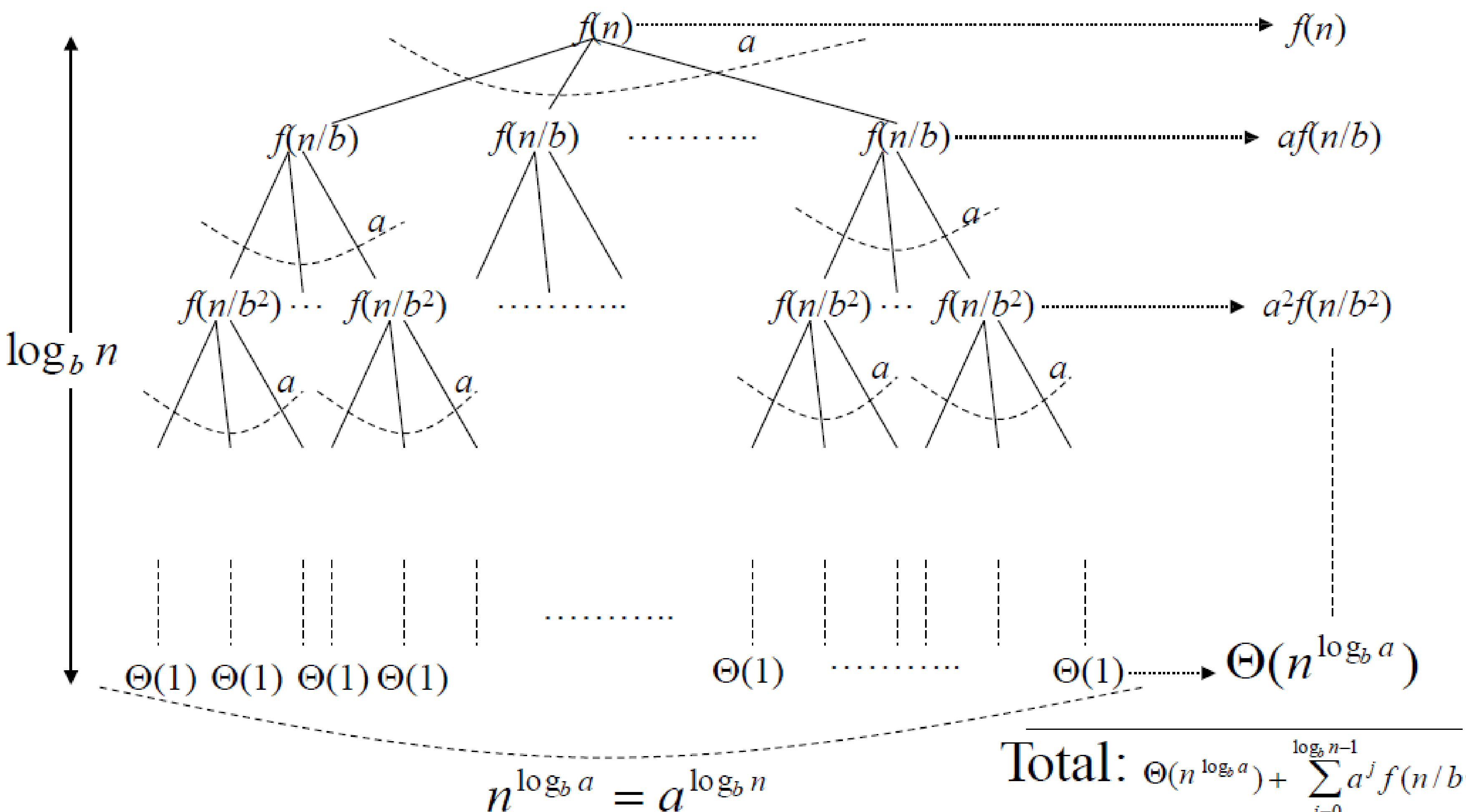
The master method

The master method applies to recurrences of the form

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

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

The Recursion Tree of T(n)



Total: $\Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

Three common cases

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

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

- $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

Solution: $T(n) = \Theta(n^{\log_b a})$.

Three common cases

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

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

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: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Three common cases (cont.)

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

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a f(n/b) \le c f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Examples

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

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

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

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

Examples

Ex. $T(n) = 4T(n/2) + n$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.
CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
$\therefore\ T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.
CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.
$\therefore\ T(n) = \Theta(n^2 \lg n)$.

Examples

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

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

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

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

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

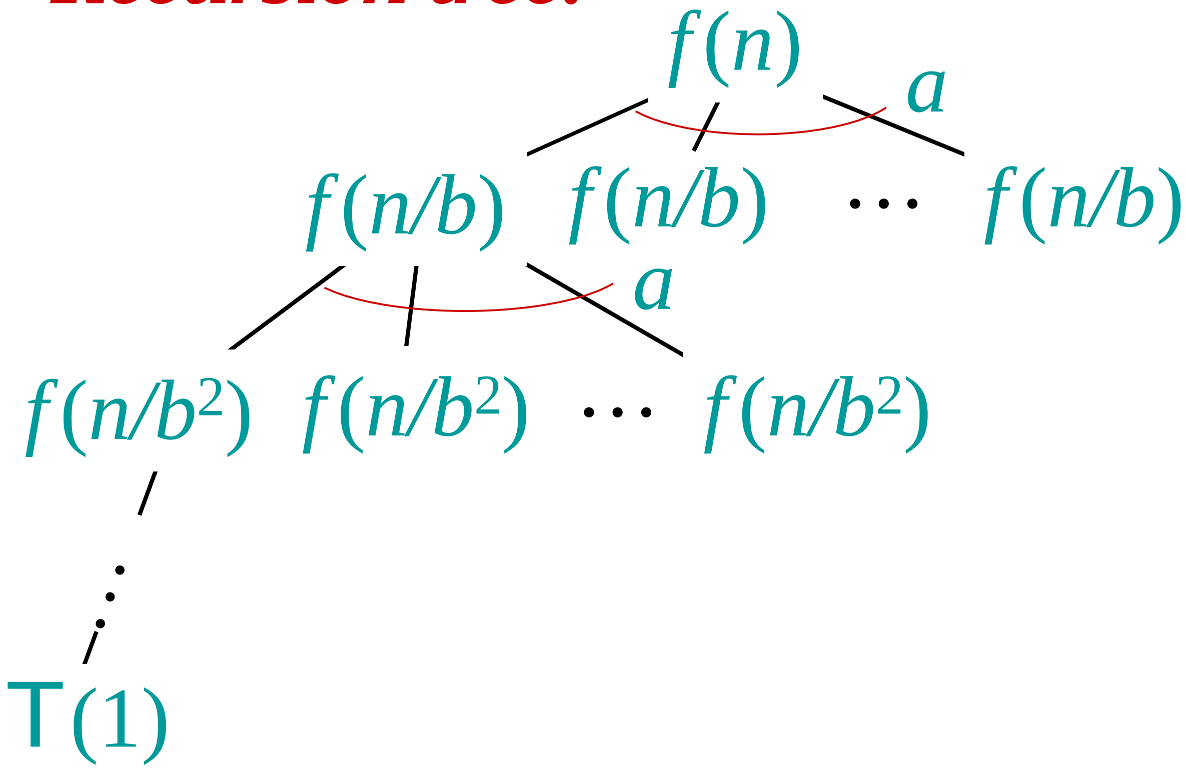
Examples

Ex. $T(n) = 4T(n/2) + n^3$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.
Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$
and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.
$\therefore T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2/\lg n$.
Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

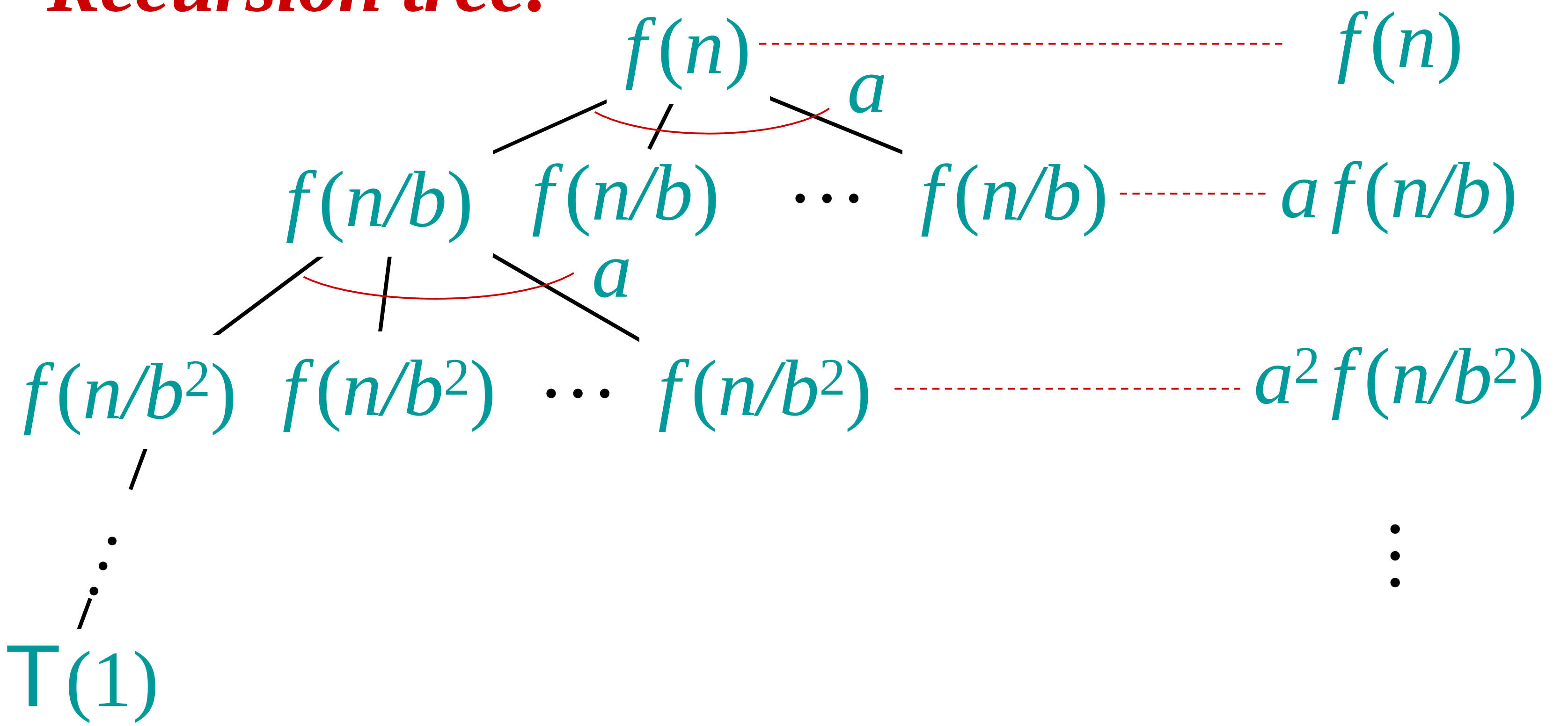
Idea of master theorem

Recursion tree:



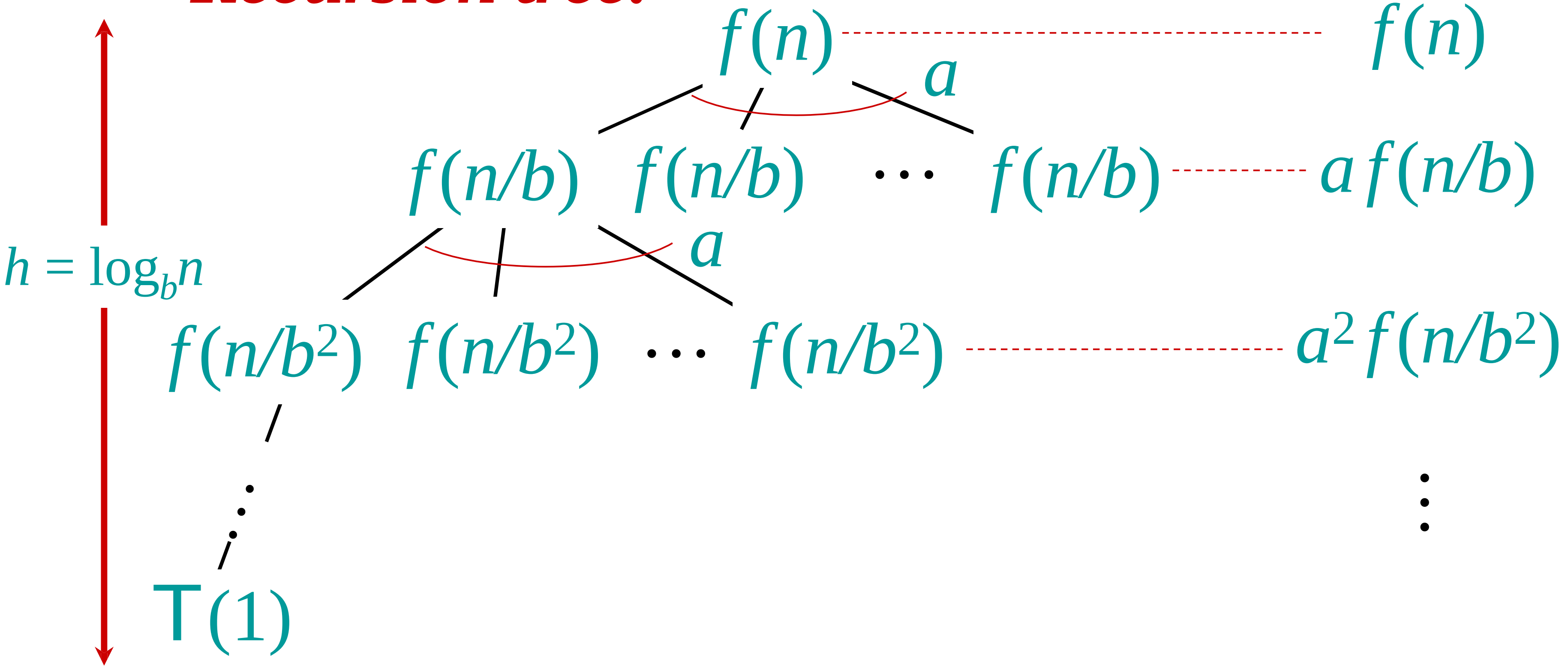
Idea of master theorem

Recursion tree:

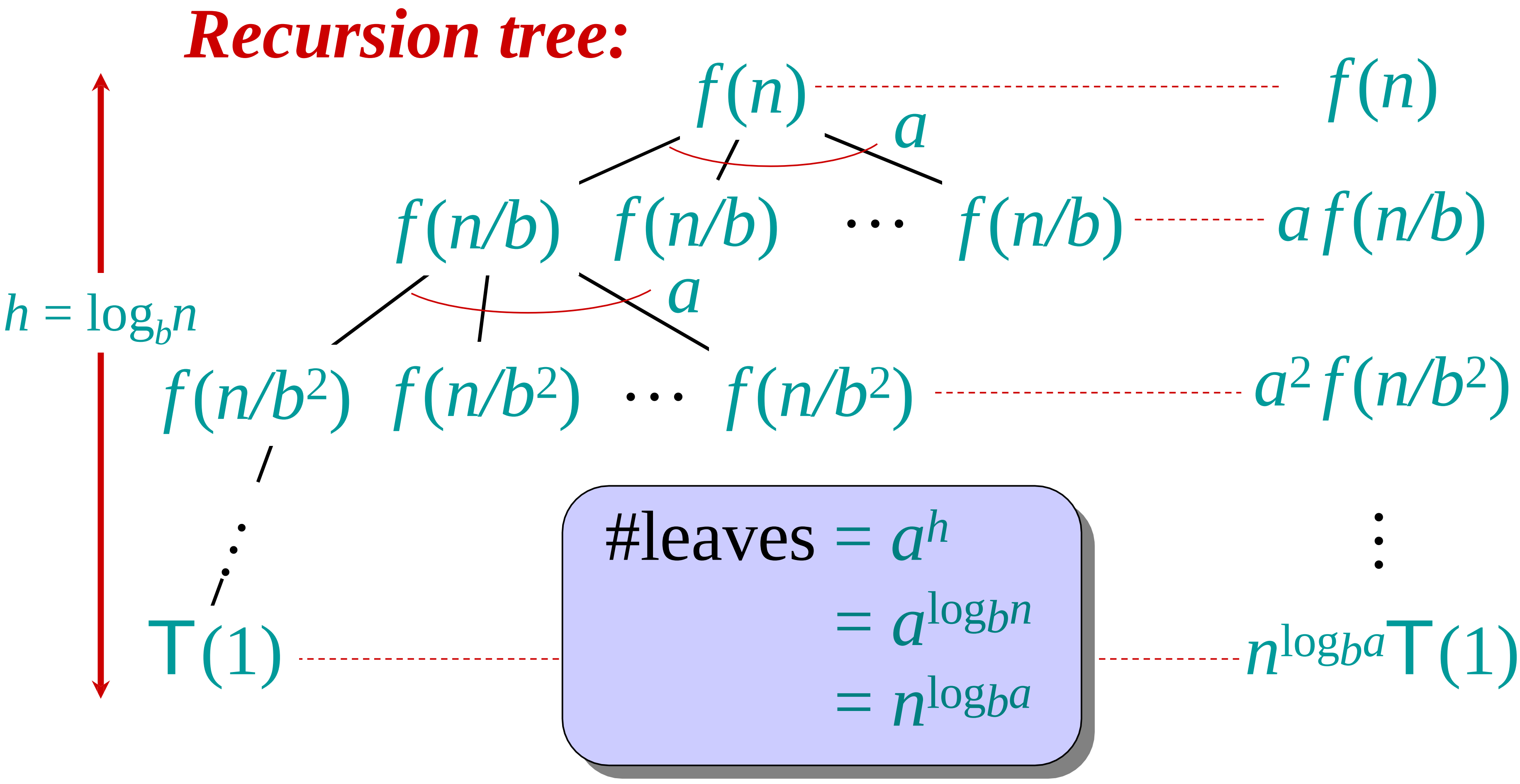


Idea of master theorem

Recursion tree:

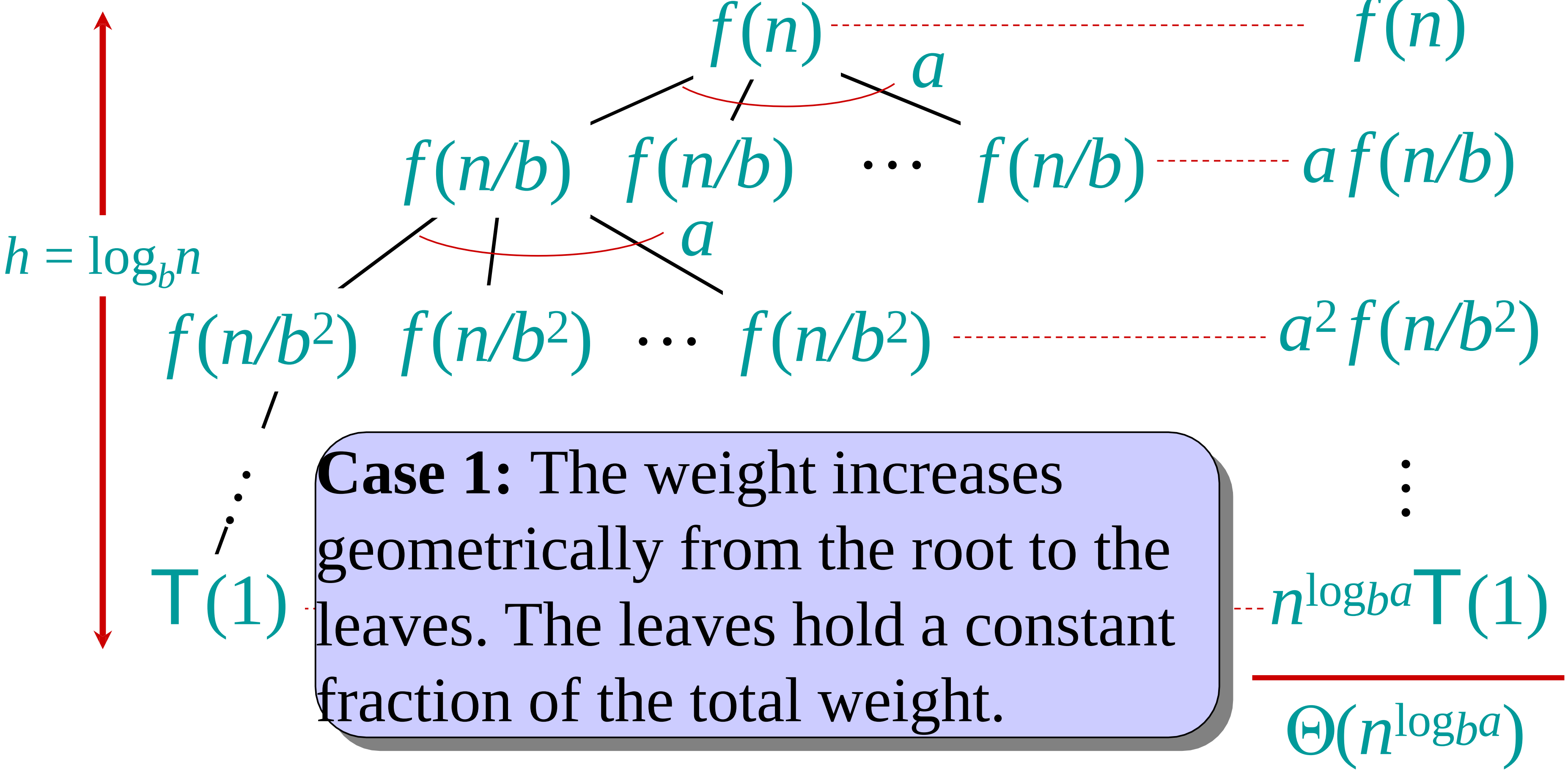


Idea of master theorem



Idea of master theorem

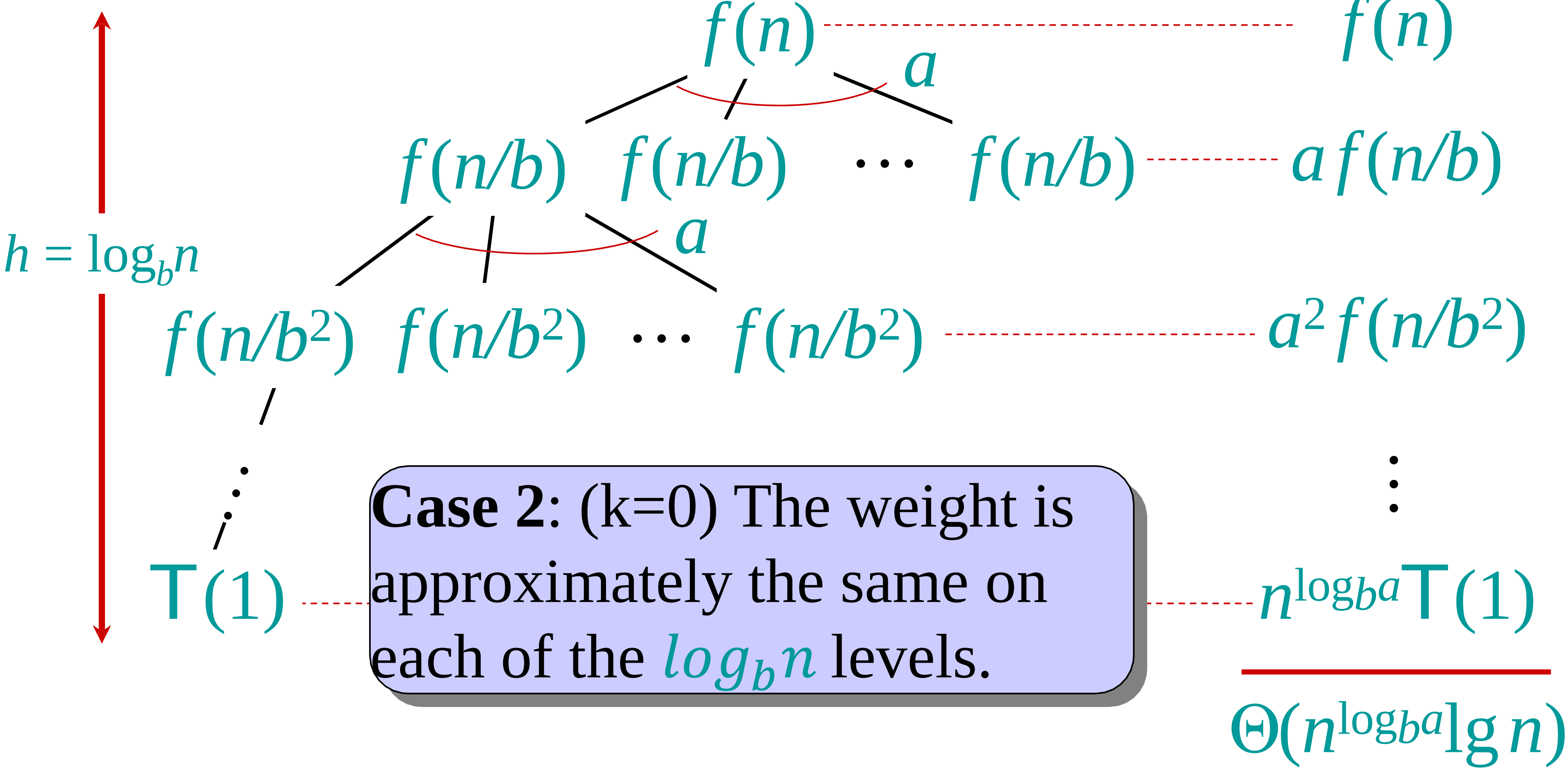
Recursion tree:



Case 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Idea of master theorem

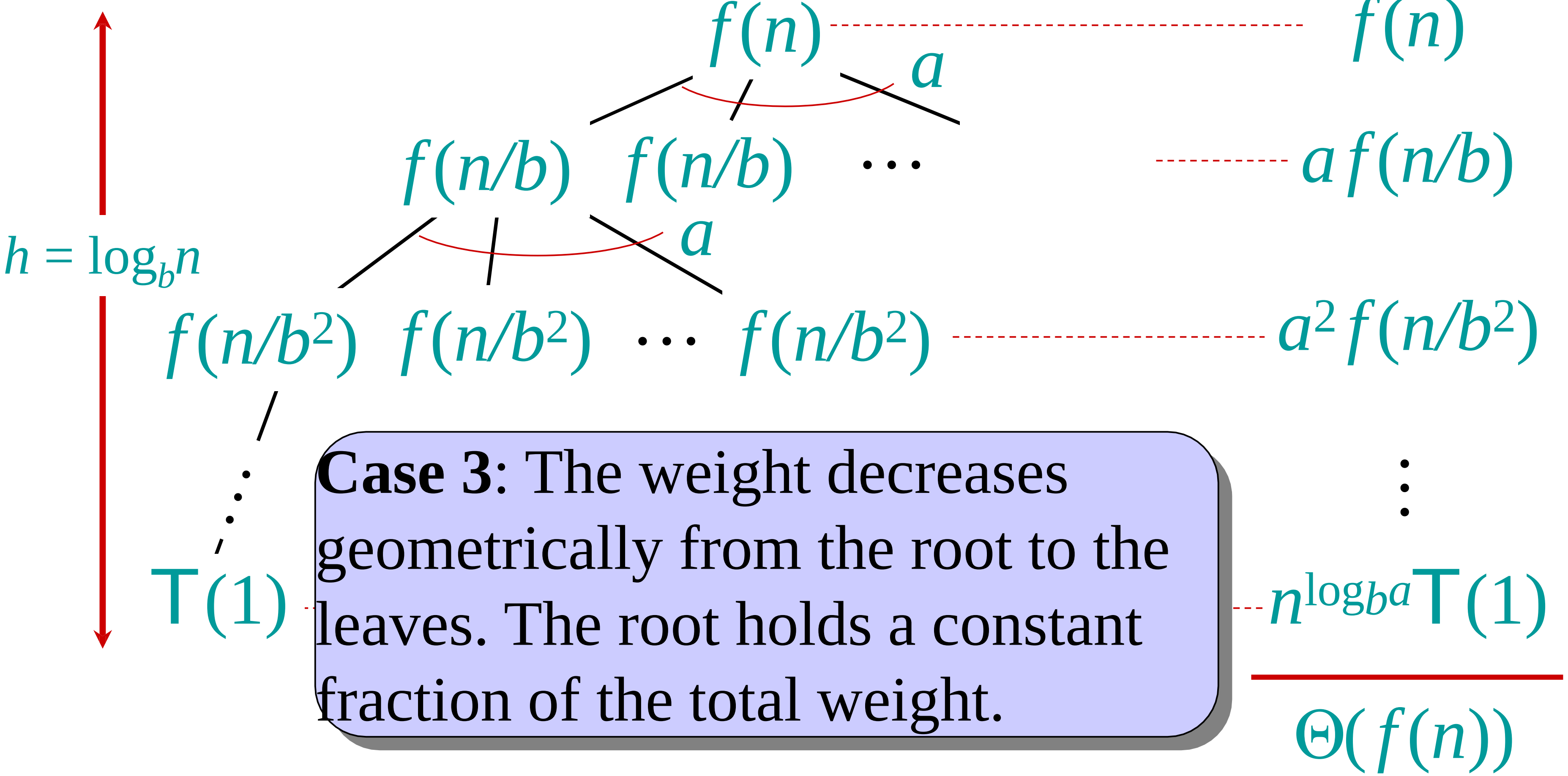
Recursion tree:



Case 2: (k=0) The weight is approximately the same on each of the $\log_b n$ levels.

Idea of master theorem

Recursion tree:



Case 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Five representative problems

Interval Scheduling

You have a resource—it may be a lecture room, a supercomputer, or an electron microscope—and many people request to use the resource for periods of time.

A request takes the form: Can I reserve the resource starting at time *s*, until time *f* ? We will assume that the resource can be used by at most one person at a time.

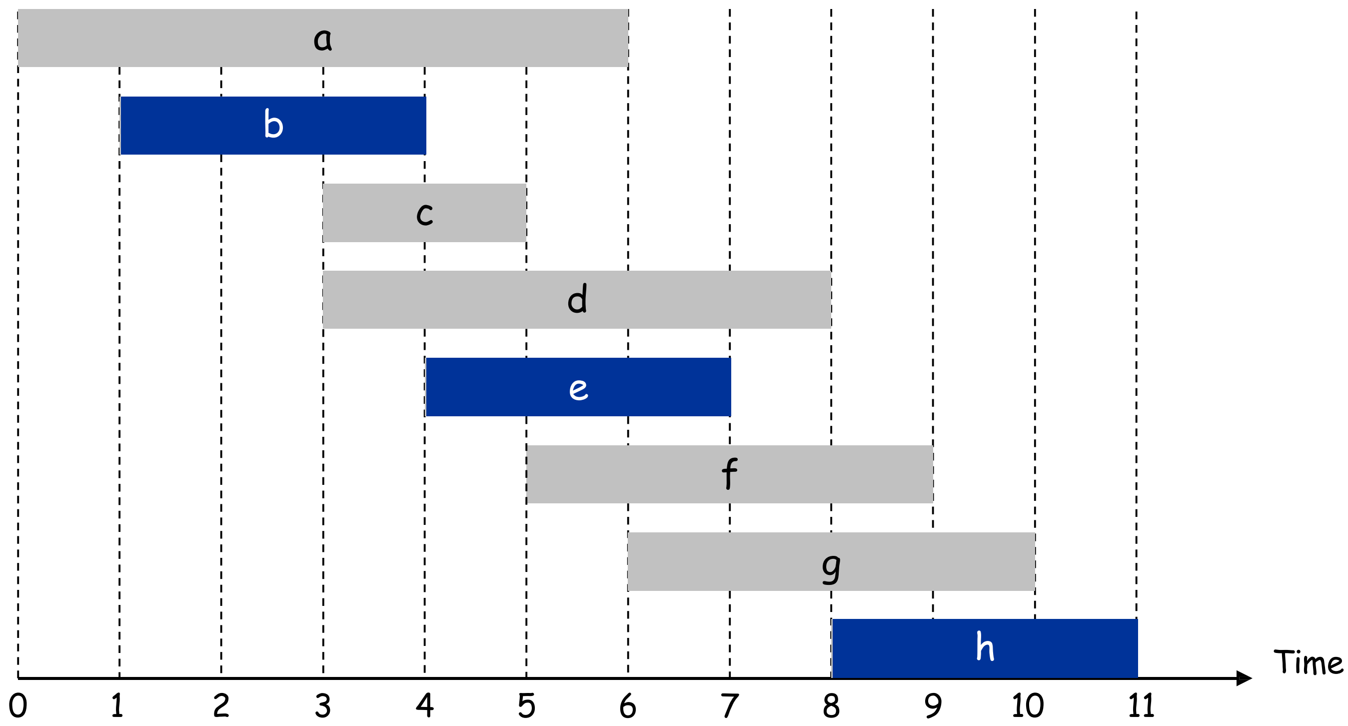
A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time.

The goal is to maximize the number of requests accepted.

Interval Scheduling

Input. Set of jobs with start times and finish times.
Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap



Weighted Interval Scheduling

In the interval scheduling problem, we sought to maximize the number of requests that could be accommodated simultaneously.

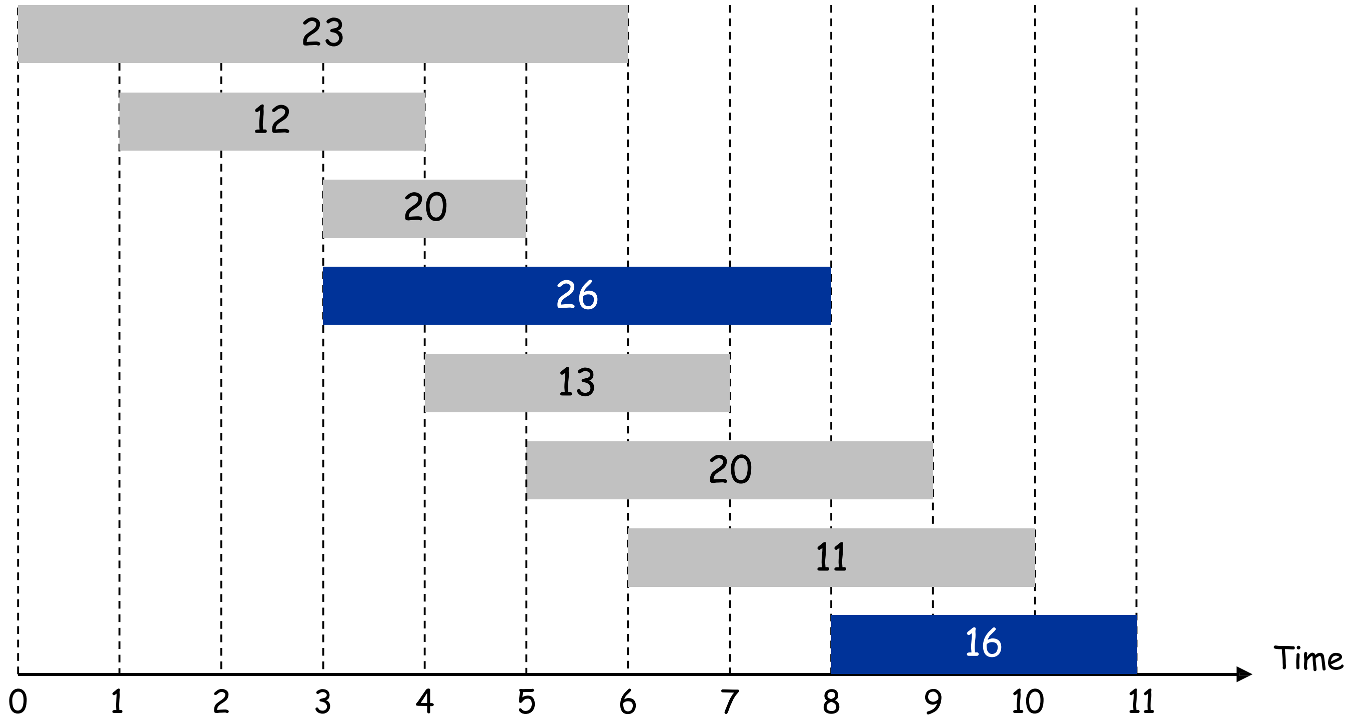
Suppose more generally that each request interval i has an associated value or weight, *vi* > 0. We could picture this as the amount of money we will make from the *ith* individual if we schedule his or her request.

The goal is to find a compatible subset of intervals of maximum total value.

Weighted Interval Scheduling

Input. Set of jobs with start times, finish times, and weights.

Goal. Find maximum weight subset of mutually compatible jobs.



Bipartite Matching

Def. A graph $G = (V, E)$ is *bipartite* if its node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y.

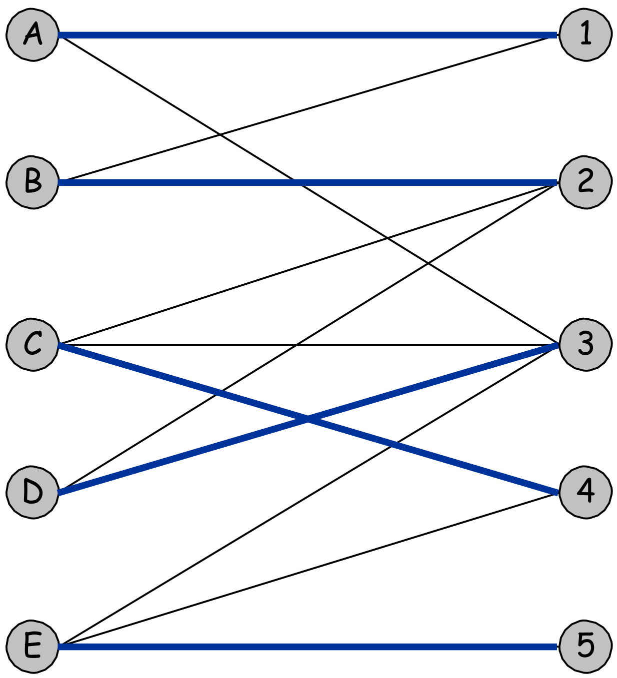
Def. A *matching* in G is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M. M is a *perfect matching* if every node appears in exactly one edge of M.

Matchings in bipartite graphs can model situations in which objects are being *assigned* to other objects.

Bipartite Matching Problem

Input. Bipartite graph.

Goal. Find maximum cardinality matching.



If $|X| = |Y| = n$, then there is a perfect matching if and only if the maximum matching has size n.

Independent Set

Def. Given a graph $G = (V, E)$, we say a set of nodes $S \subseteq V$ is ***independent*** if no two nodes in S are joined by an edge.

An extremely general problem, which includes most of these earlier problems as special cases.

The independent set problem encodes any situation in which you are trying to choose from among a collection of objects and there are pairwise *conflicts* among some of the objects.

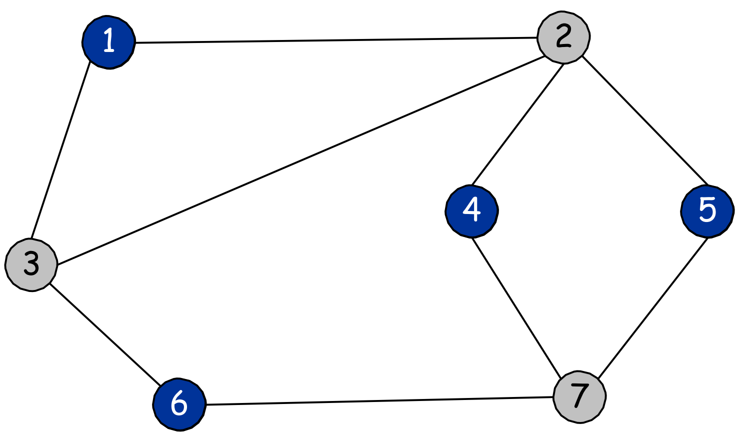
Independent Set Problem

Input. Graph.

Goal. Find maximum cardinality independent set.

↑

subset of nodes such that no two joined by an edge



Interval Scheduling and Bipartite Matching can both be encoded as special cases of the Independent Set Problem.

Competitive Facility Location

A two player game. Consider two large companies that operate cafe franchises across the country—let's call them *JavaPlanet* and *Queequeg's* Coffee—and they are currently competing for market share in a geographic area.

- First JavaPlanet opens a franchise; then Queequeg's Coffee opens a franchise; then JavaPlanet; then Queequeg's; and so on.
- Suppose they must deal with zoning regulations that require no two franchises be located too close together, and each is trying to make its locations as convenient as possible.

Q. Who will win?

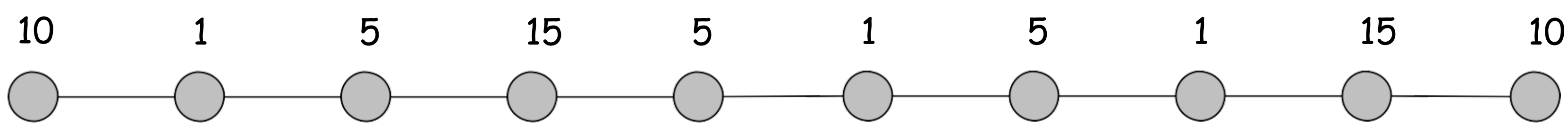
Competitive Facility Location

Input. Graph with weight on each node.

Game. Two competing players alternate in selecting nodes.

Not allowed to select a node if any of its neighbors have been selected.

Goal. Select a maximum weight subset of nodes.



Second player can guarantee 20, but not 25.

Note that. Not only is it computationally difficult to determine whether a player has a winning strategy; on a reasonably sized graph, it would even be hard for us to convince you that it has a winning strategy.

Five Representative Problems

Variations on a theme. independent set

Interval scheduling. $O(n \log n)$

Weighted interval scheduling. $O(n \log n)$

Bipartite matching. $O(n^k)$

Independent set. NP-complete

Competitive facility location. PSPACE-complete