Algorithm Design and Analysis

CSE 565

LECTURE 9

Divide and Conquer

- Merge sort
- Counting Inversions
- Binary Search
- Exponentiation

Solving Recurrences

- Recursion Tree Method
- Master Theorem

Sofya Raskhodnikova

Recursion

- Next couple of weeks: recursion as an algorithms design technique
- Three important classes of algorithms
  - Divide and conquer
  - Back tracking
  - Dynamic programming

Recursion in design and analysis

Recursion in design and proof of correctness, but time/space analysis is more "global"

Divide and Conquer

> Divide et impera.
> Veni, vidi, vici.
> - Julius Caesar

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into **two** equal parts of size n/2.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in **linear time**.
- Consequence.
  - Brute force: $\Theta(n^2)$.
  - Divide & conquer: $\Theta(n \log n)$.

Divide and Conquer

> Divide et impera.
> Veni, vidi, vici.
> - Julius Caesar

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

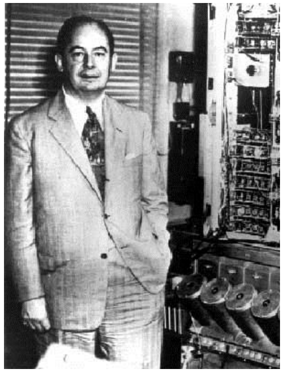
- Examples
  - Mergesort, quicksort, binary search
  - Geometric problems: convex hull, nearest neighbors, line intersection, algorithms for planar graphs
  - Algorithms for processing trees
  - Many data structures (binary search trees, heaps, k-d trees,…)

Analyzing Recursive Algorithms

- Correctness almost always uses strong induction
  1. Prove correctness of base cases
     (typically: $n \leq constant$)
  2. For arbitrary n:
     - Assume that algorithm performs correctly on all input sizes $k < n$
     - Prove that algorithm is correct on input size n
- Time/space analysis: often use recurrence
  – Structure of recurrence reflects algorithm

Mergesort

– Divide array into two halves.
– Recursively sort each half.
– Merge two halves to make sorted whole.

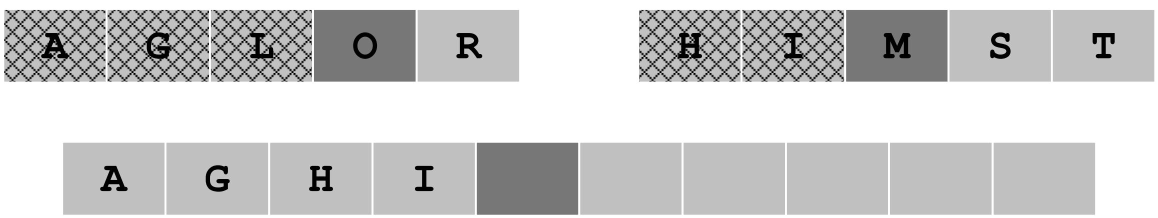


Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S		
A L G O R					I T H M S					divide	$O(1)$
A G L O R					H I M S T					sort	$2T(n/2)$
A	G	H	I	L	M	O	R	S	T	merge	$O(n)$

Merging

- Combine two pre-sorted lists into a sorted whole.
- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



- Challenge for the bored: in-place merge [Kronrud, 1969]

using only a constant amount of extra storage

Recurrence for Mergesort

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

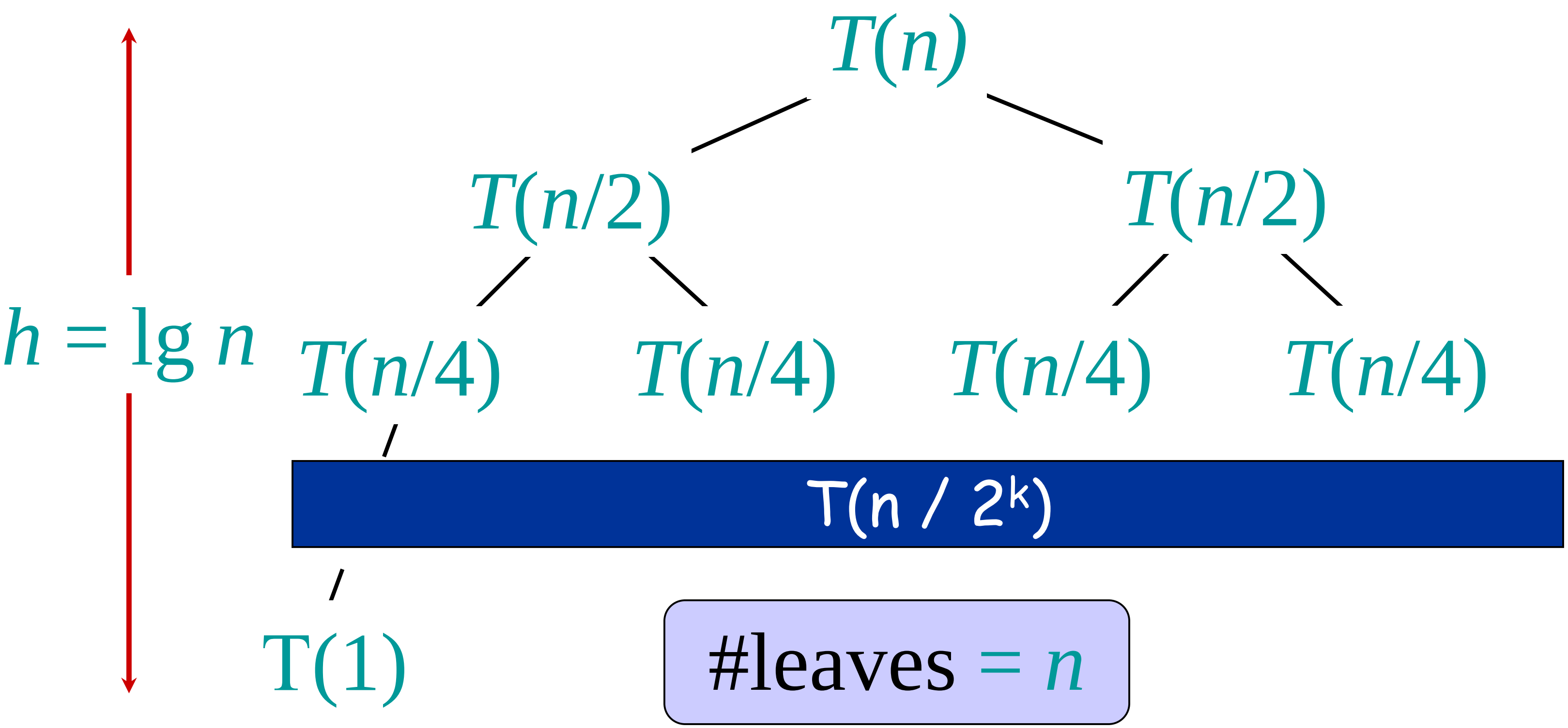
- $T(n)$ = worst case running time of Mergesort on an input of size n.
- Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.
- Usually omit the base case because our algorithms always run in time $\Theta(1)$ when n is a small constant.
- Several methods to find an upper bound on $T(n)$.

Recursion Tree Method

- Technique for guessing solutions to recurrences
  - Write out tree of recursive calls
  - Each node gets assigned the work done during that call to the procedure (dividing and combining)
  - Total work is **sum** of work at all nodes
- After guessing the answer, can prove by induction that it works.

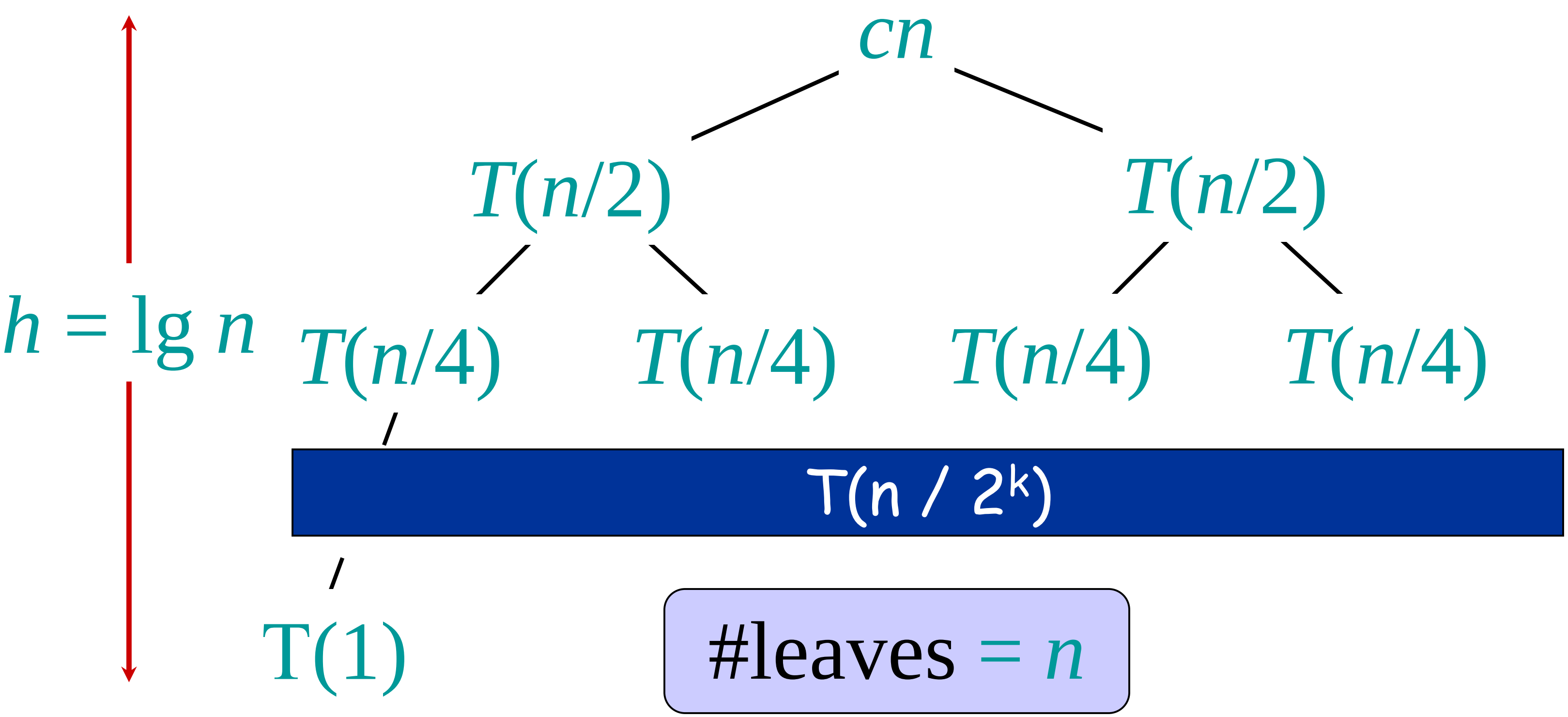
Recursion Tree for Mergesort

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



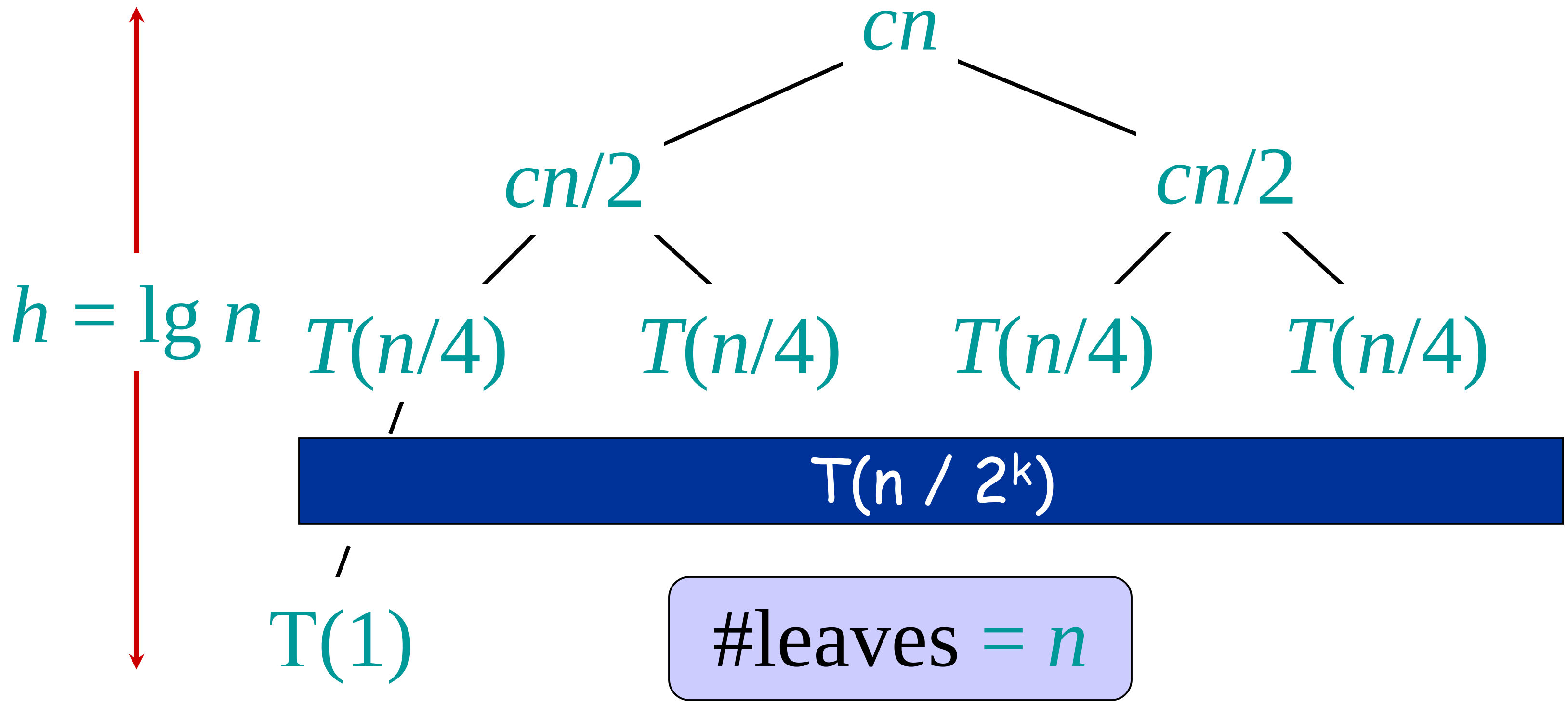
Recursion Tree for Mergesort

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



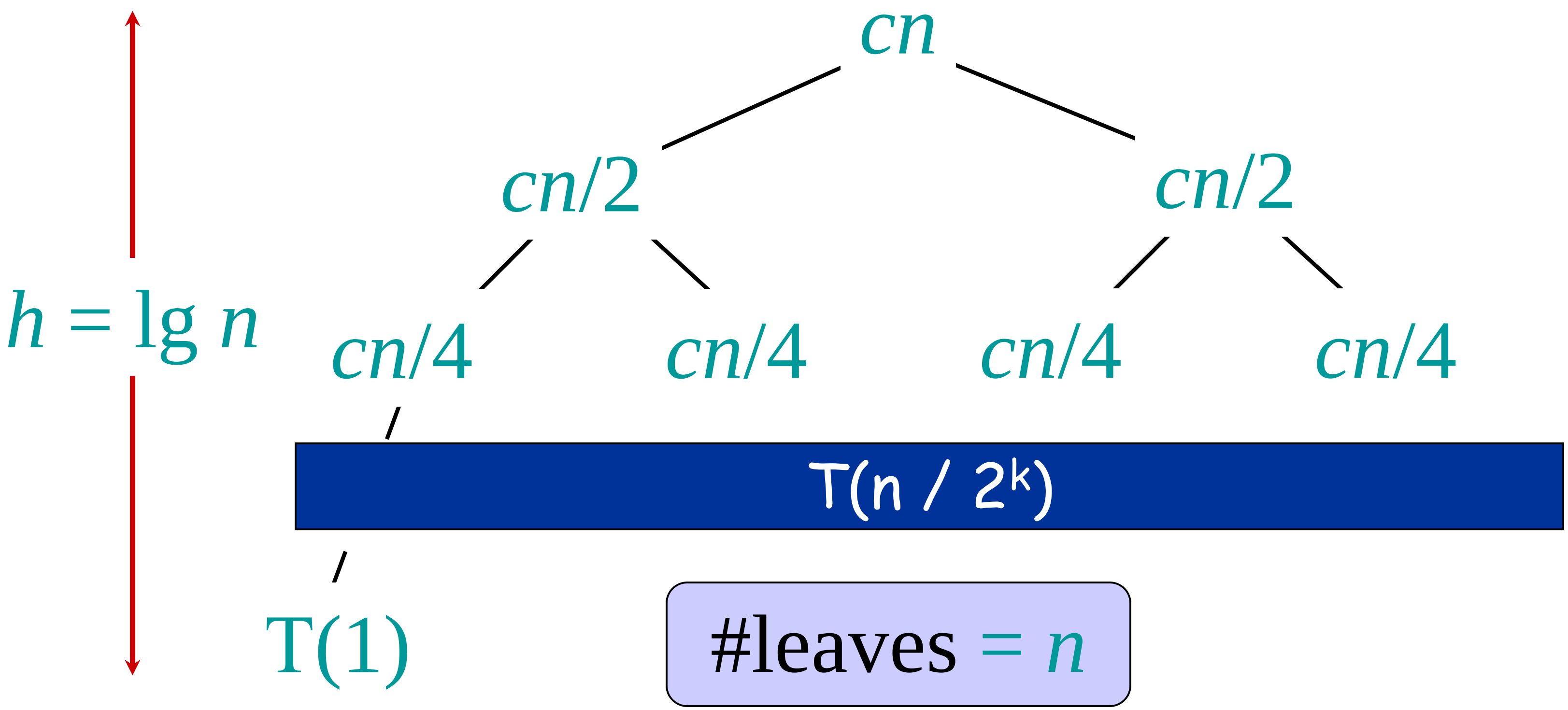
Recursion Tree for Mergesort

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



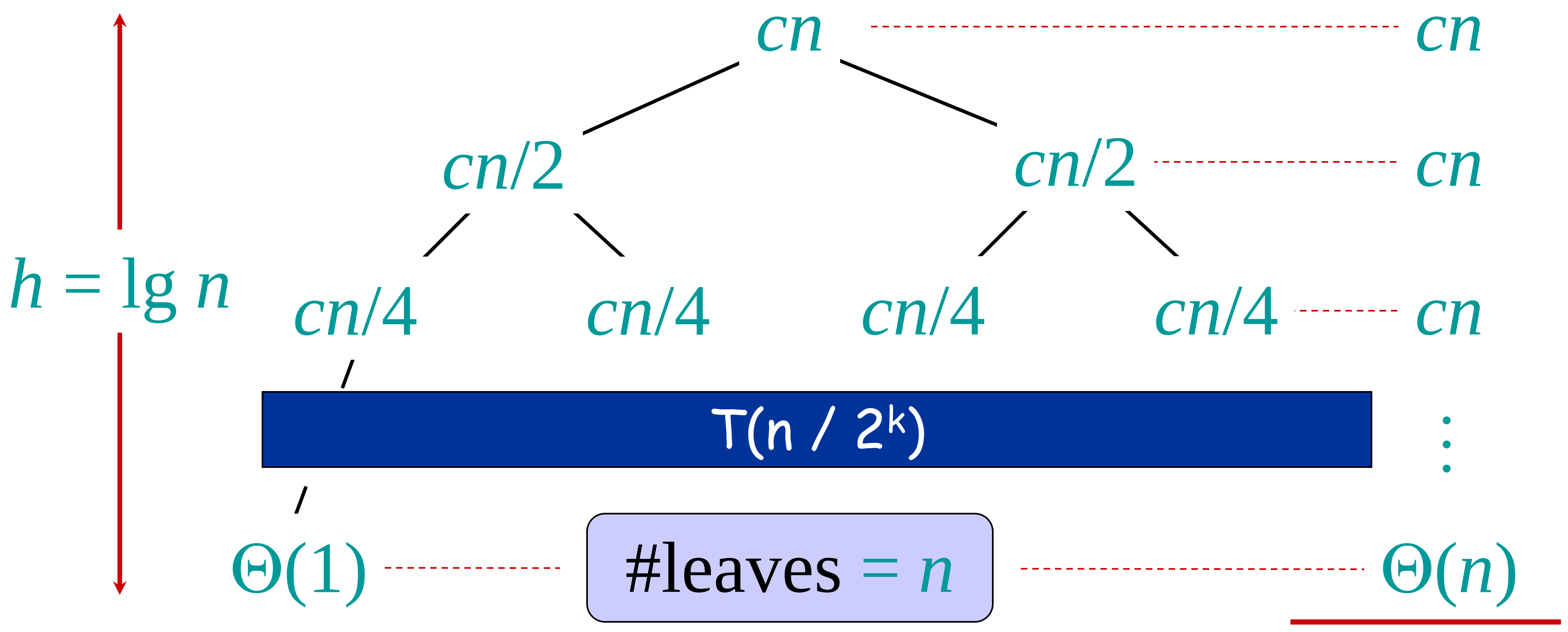
Recursion Tree for Mergesort

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



Recursion Tree for Mergesort

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



Total $= \Theta(n \lg n)$

Counting inversions

Counting Inversions

- **Music site tries to match your song preferences with others.**
  - You rank n songs.
  - Music site consults database to find people with **similar** tastes.
- **Similarity metric:** number of inversions between two rankings.
  - My rank: 1, 2, …, n.
  - Your rank: $a_1, a_2, \ldots, a_n$.
  - Songs i and j **inverted** if $i < j$, but $a_i > a_j$.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions

3-2, 4-2

- **Brute force:** check all $\Theta(n^2)$ pairs i and j.

Counting Inversions: Algorithm

- Divide-and-conquer

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Algorithm

- Divide-and-conquer
  - **Divide**: separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $\Theta(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Counting Inversions: Algorithm

- Divide-and-conquer
  - Divide: separate list into two pieces.
  - **Conquer**: recursively count inversions in each half.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $\Theta(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: $2T(n / 2)$

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Algorithm

- Divide-and-conquer
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - **Combine**: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $\Theta(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: $2T(n / 2)$

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is **sorted**.
- Count inversions where a_i and a_j are in different halves.
- **Merge** two sorted halves into sorted whole.

to maintain sorted invariant

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: $\Theta(n)$

2	3	7	10	11	14	16	17	18	19	23	25

Merge: $\Theta(n)$

$T(n) = 2T(n/2) + \Theta(n)$. Solution: $T(n) = \Theta(n \log n)$.

Implementation

- Pre-condition. [Merge-and-Count] A and B are sorted.
- Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A) ← Sort-and-Count(A)
   (rB, B) ← Sort-and-Count(B)
   (r , L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Recurrence for binary search

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

 # subproblems

subproblem size

work dividing and combining

Recurrence for binary search

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

\# subproblems

subproblem size

work dividing and combining

$$\Rightarrow \quad T(n) = T(n/2) + c = T(n/4) + 2c$$

$$\cdots$$

$$= c\lfloor \log n \rfloor + O(1) = \Theta(\lg n)\,.$$

Review Question: Exponentiation

Problem: Compute a^b, where $b \in \mathbb{N}$ is n bits long.
Question: How many multiplications?

Naive algorithm: $\Theta(b) = \Theta(2^n)$ (exponential in the input length!)

Divide-and-conquer algorithm:

$$a^b = \begin{cases} a^{b/2} \times a^{b/2} & \text{if } b \text{ is even;} \\ a^{(b-1)/2} \times a^{(b-1)/2} \times a & \text{if } b \text{ is odd.} \end{cases}$$

$$T(b) = T(b/2) + \Theta(1) \Rightarrow T(b) = \Theta(\log b) = \Theta(\mathrm{n})\,.$$

So far: 2 recurrences

- Mergesort; Counting Inversions

  $T(n) = 2\ T(n/2) + \Theta(n) \qquad = \Theta(n \log n)$

- Binary Search; Exponentiation

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

Master Theorem: method for solving recurrences.

Master Theorem

The master method

The master method applies to recurrences of the form

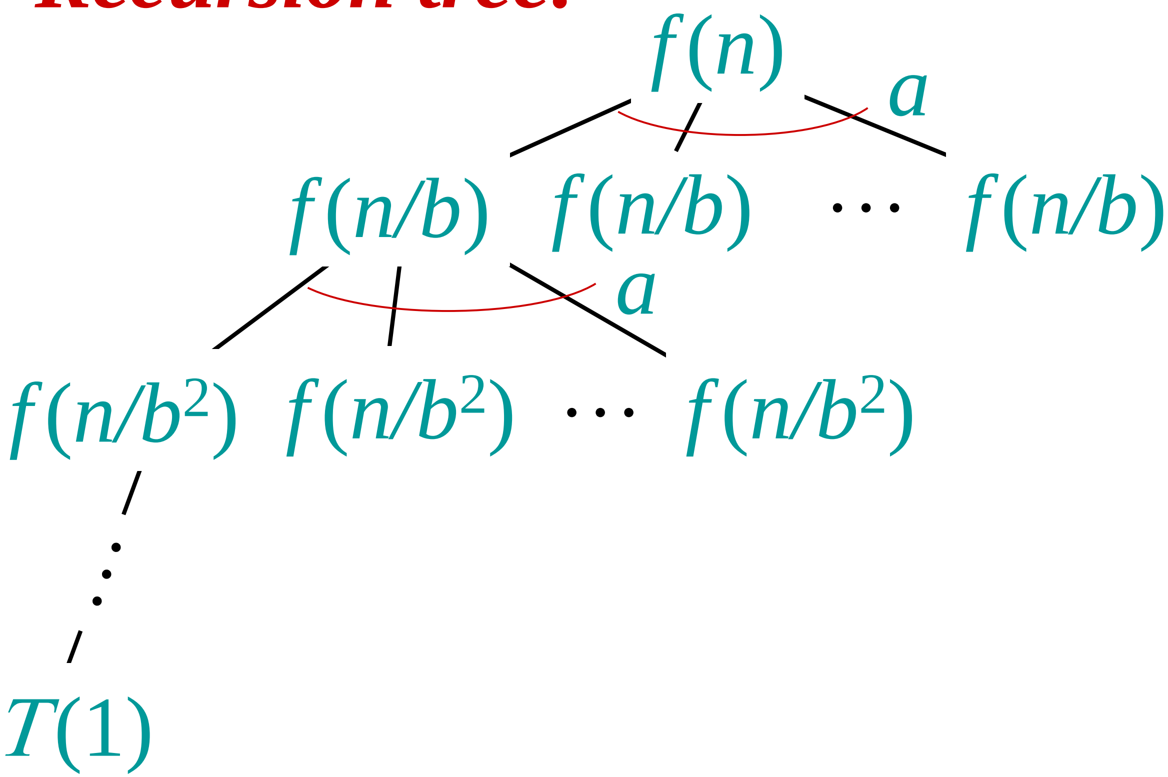
$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive, that is $f(n) > 0$ for all $n > n_0$.

First step: compare $f(n)$ to $n^{\log_b a}$.

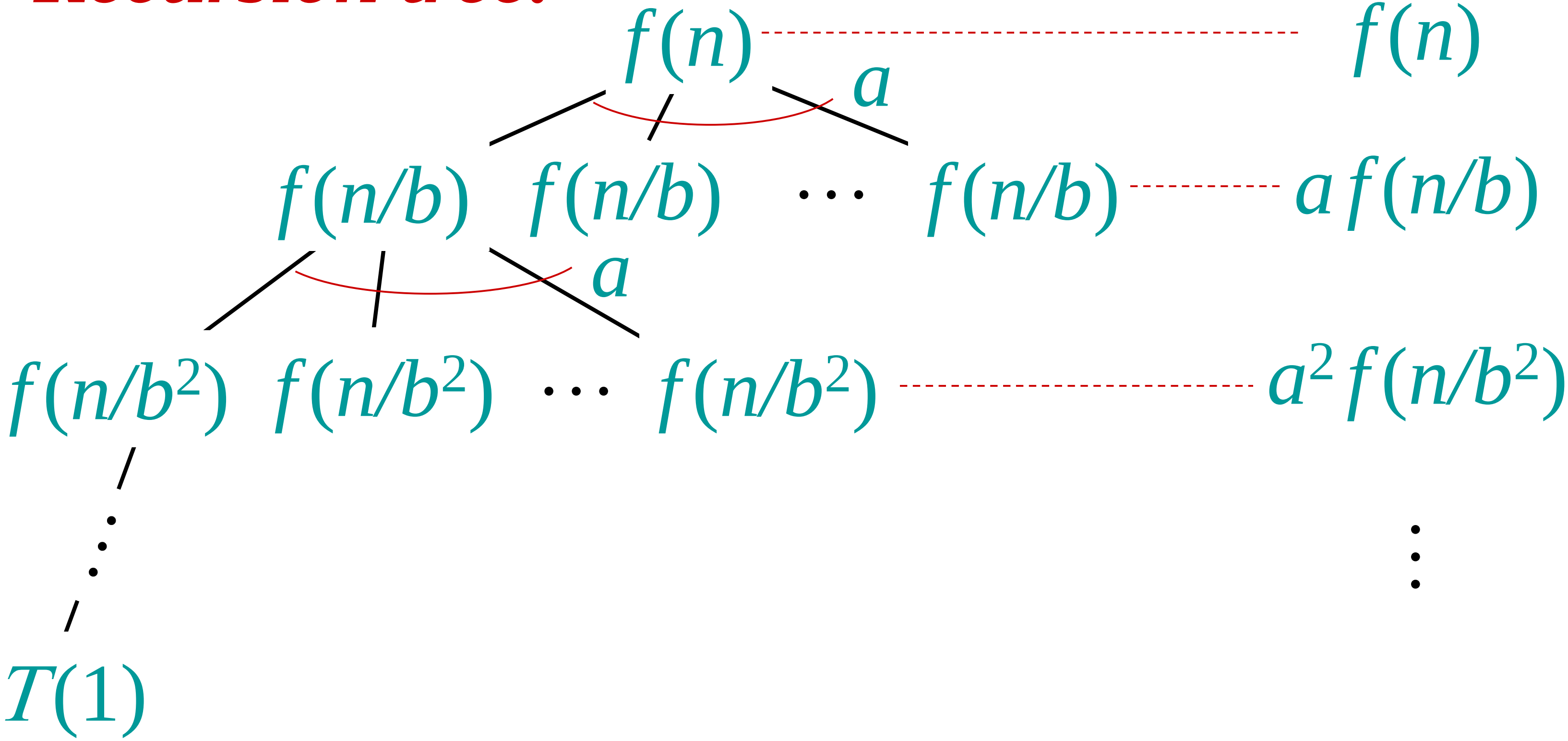
Idea of master theorem

Recursion tree:



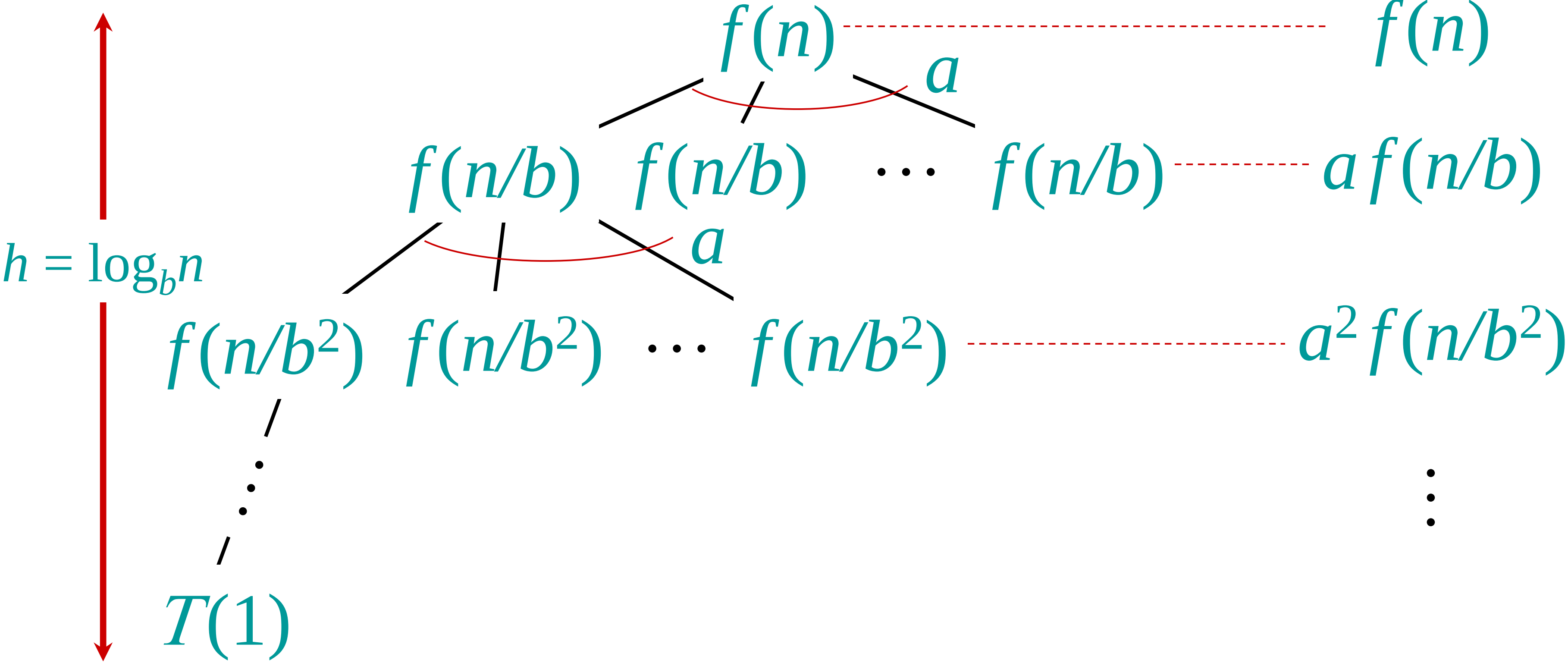
Idea of master theorem

Recursion tree:



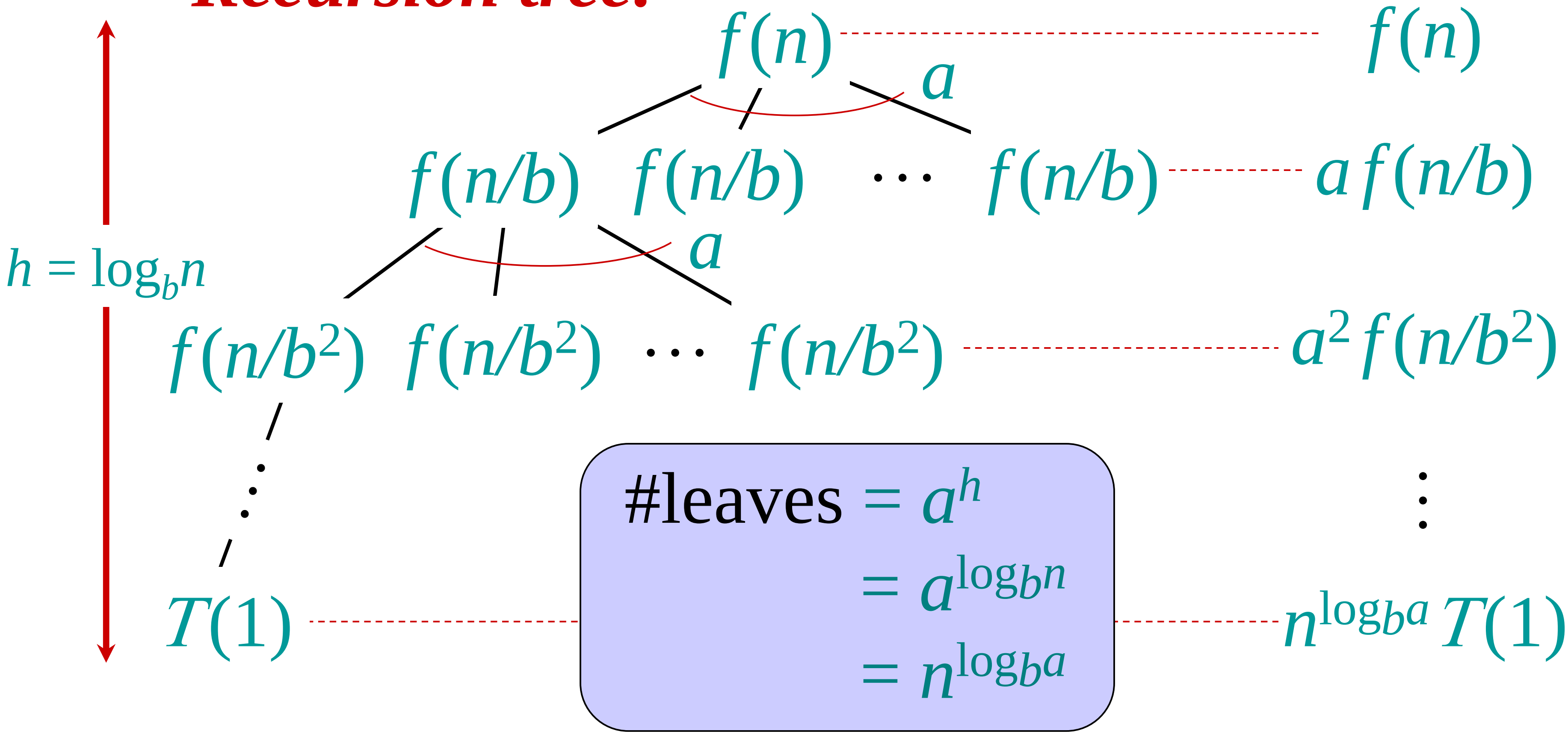
Idea of master theorem

Recursion tree:



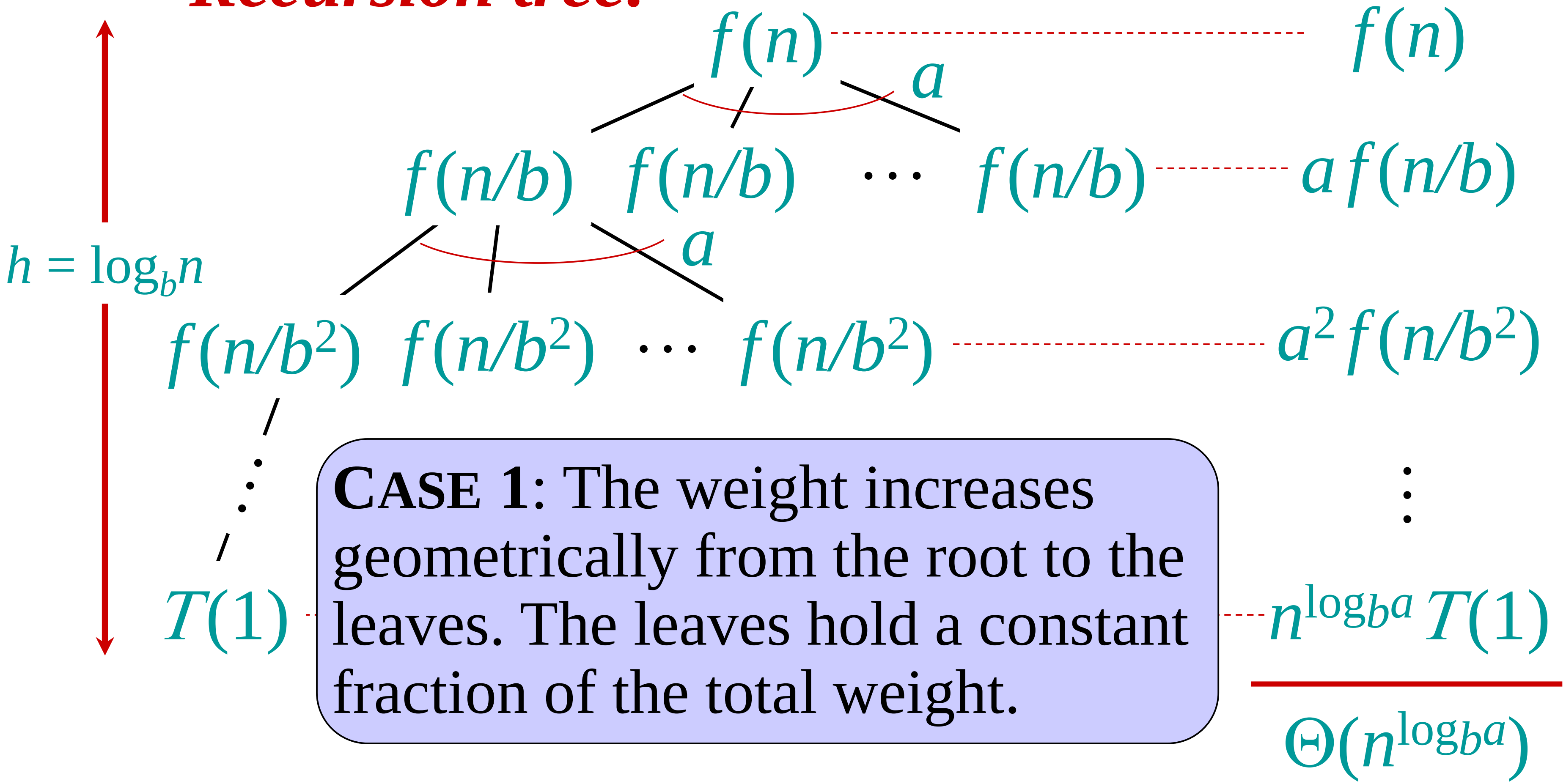
Idea of master theorem

Recursion tree:



Idea of master theorem

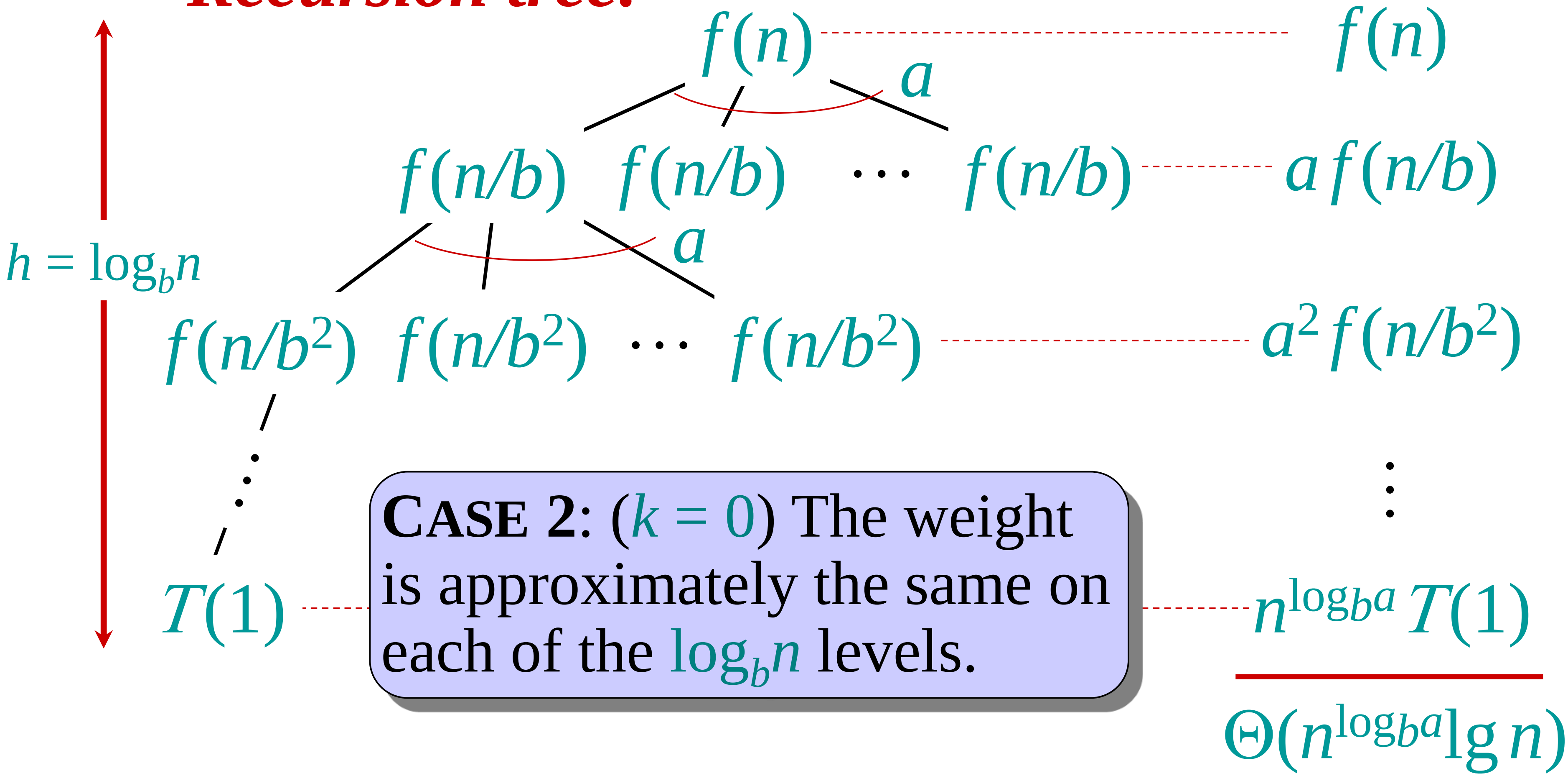
Recursion tree:



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:

$f(n)$ .... $f(n)$

a

$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$ .... $a\,f(n/b)$

a

$h = \log_b n$

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

$\vdots$

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

$\Theta(f(n))$

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

Master Theorem: 3 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: $T(n) = \Theta(n^{\log_b a})$.

Master Theorem: 3 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: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Master Theorem: 3 common cases

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 $4(n/2)^3 \le cn^3$ (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 $n^\varepsilon = \omega(\lg n)$.

Notes on Master Theorem

- Master Thm was generalized by Akra and Bazzi to cover many more recurrences:

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(b_i n + h_i(n))$$

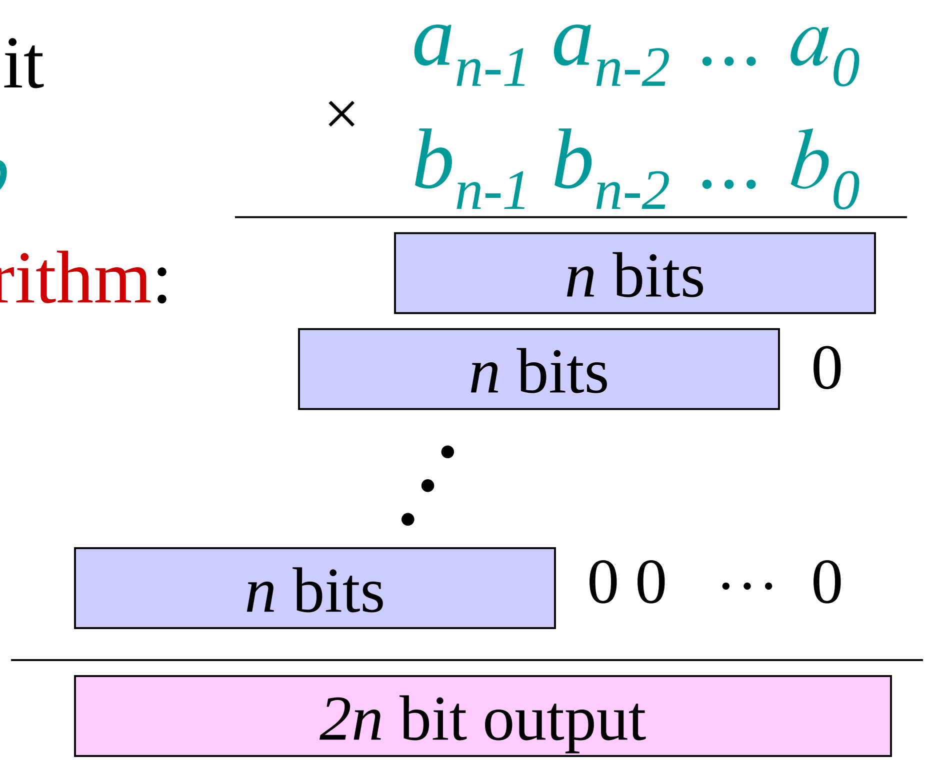
where $h_i(n) = O(\frac{n}{\log^2 n})$

- See the wikipedia article on **Akra–Bazzi method** and pointers from there.

Integer multiplication

Arithmetic on Large Integers

- **Addition**: Given n-bit integers a, b (in binary), compute $c=a+b$
  - O(n) bit operations.
- **Multiplication**: Given n-bit integers a, b, compute $c=ab$
- **Naïve (grade-school) algorithm**:
  - Write a,b in binary
  - Compute n intermediate products
  - Do n additions
  - Total work: $\Theta(n^2)$



Multiplying large integers

- **Divide and Conquer** (warmup):
  - Write $a = A_1\, 2^{n/2} + A_0$
    $b = B_1\, 2^{n/2} + B_0$
  - We want $ab = A_1B_1\, 2^n + (A_1B_0 + B_1A_0)\, 2^{n/2} + A_0B_0$
  - Multiply $n/2$ –bit integers recursively
  - $T(n) = 4T(n/2) + \Theta(n)$
  - Alas! this is still $\Theta(n^2)$ (Master Theorem, Case 1)