Solving recurrences

Recap

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

IF $\boldsymbol{p < r}$

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

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

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

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

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

How about apply iteration?

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$= 2^2T(2^{k-2}) + c2^k + c2^k$$

assume $n = 2^k$

$$\cdots\cdots$$

$$= 2^kT(1) + c(k2^k)$$

$$= O(nlgn)$$

It can be difficult to compute sometime.

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 $O(n^3)$.
- Assume that $T(k) \leq ck^3$ for $k < n$.
- Prove $T(n) \leq cn^3$ by induction.

Example of substitution

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

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

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

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 $k < n$:

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

Wrong! We must prove the I.H.

$$= cn^2 - (-n) \quad [\text{ desired} - \text{residual }]$$

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

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

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

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

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

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

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

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

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

Pick c_1 big enough to handle the initial conditions.

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

$$n^2$$

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

Example of recursion tree

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

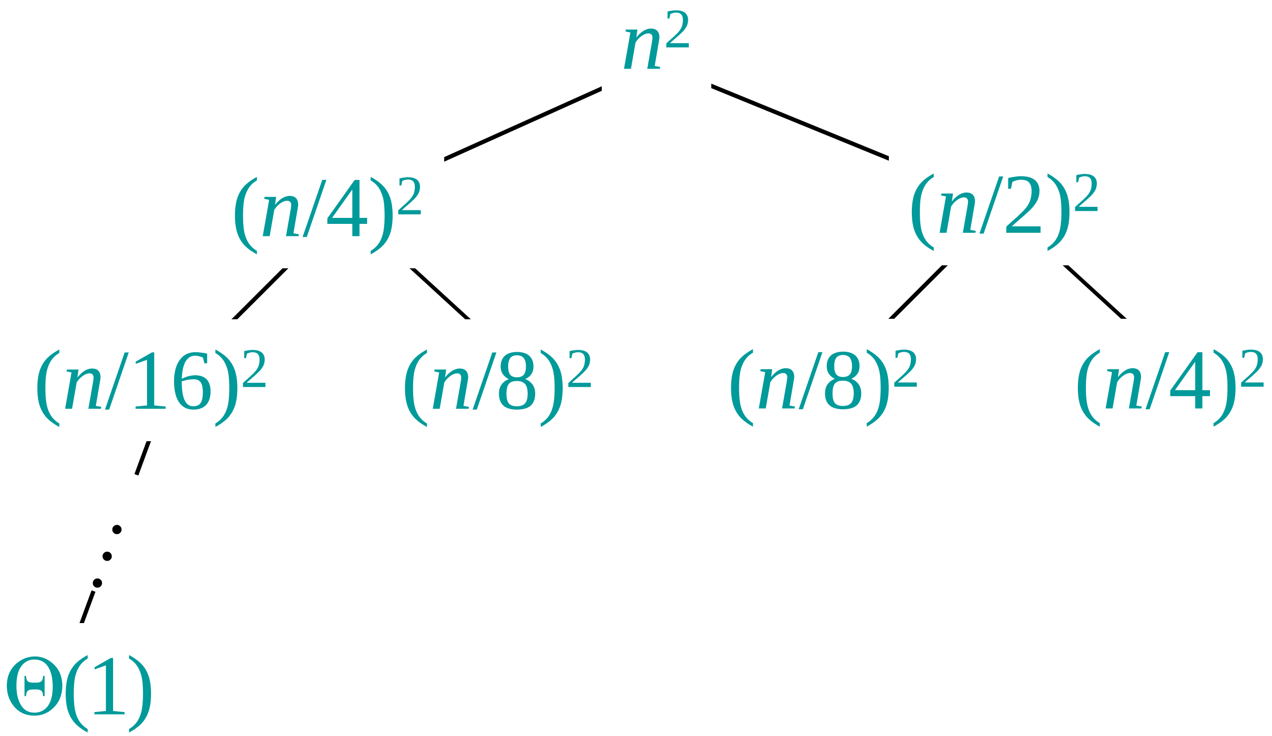
$$n^2$$

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

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

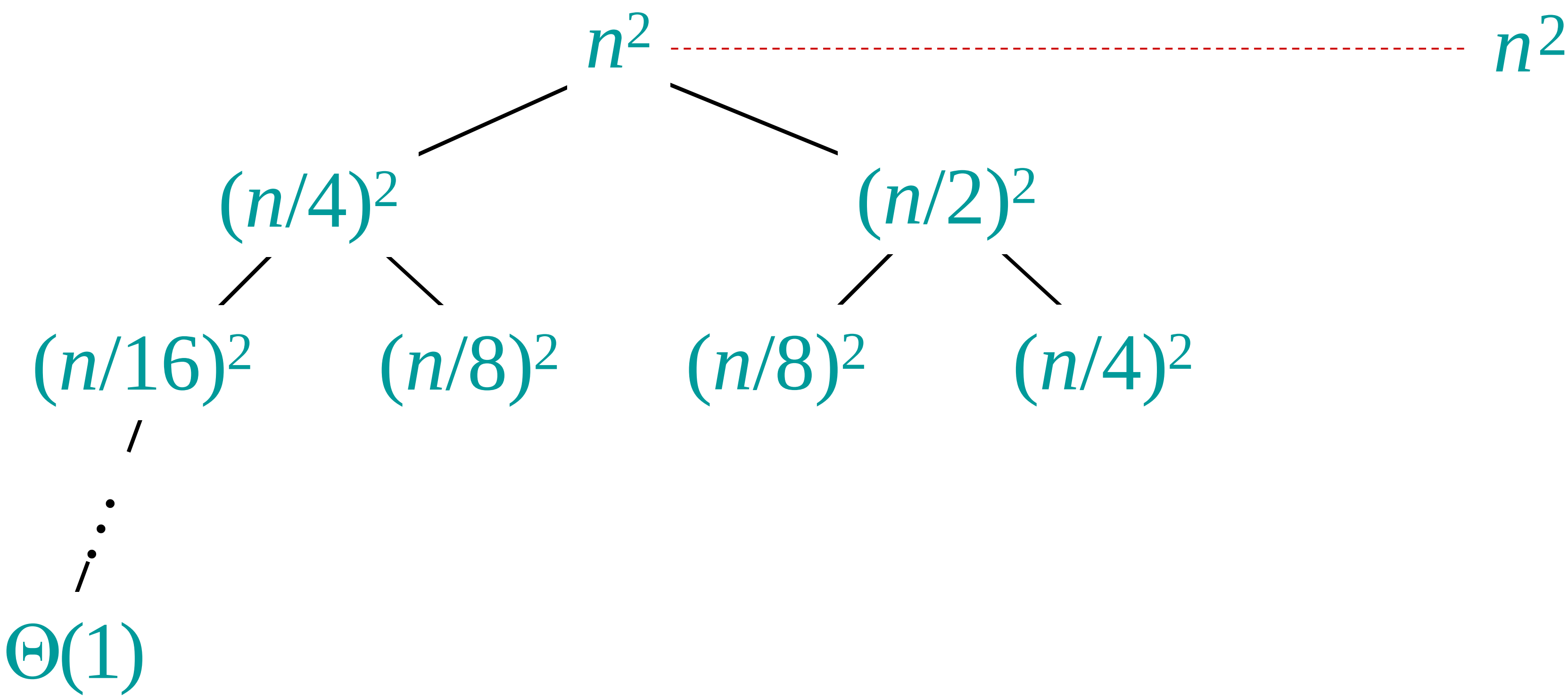
Example of recursion tree

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



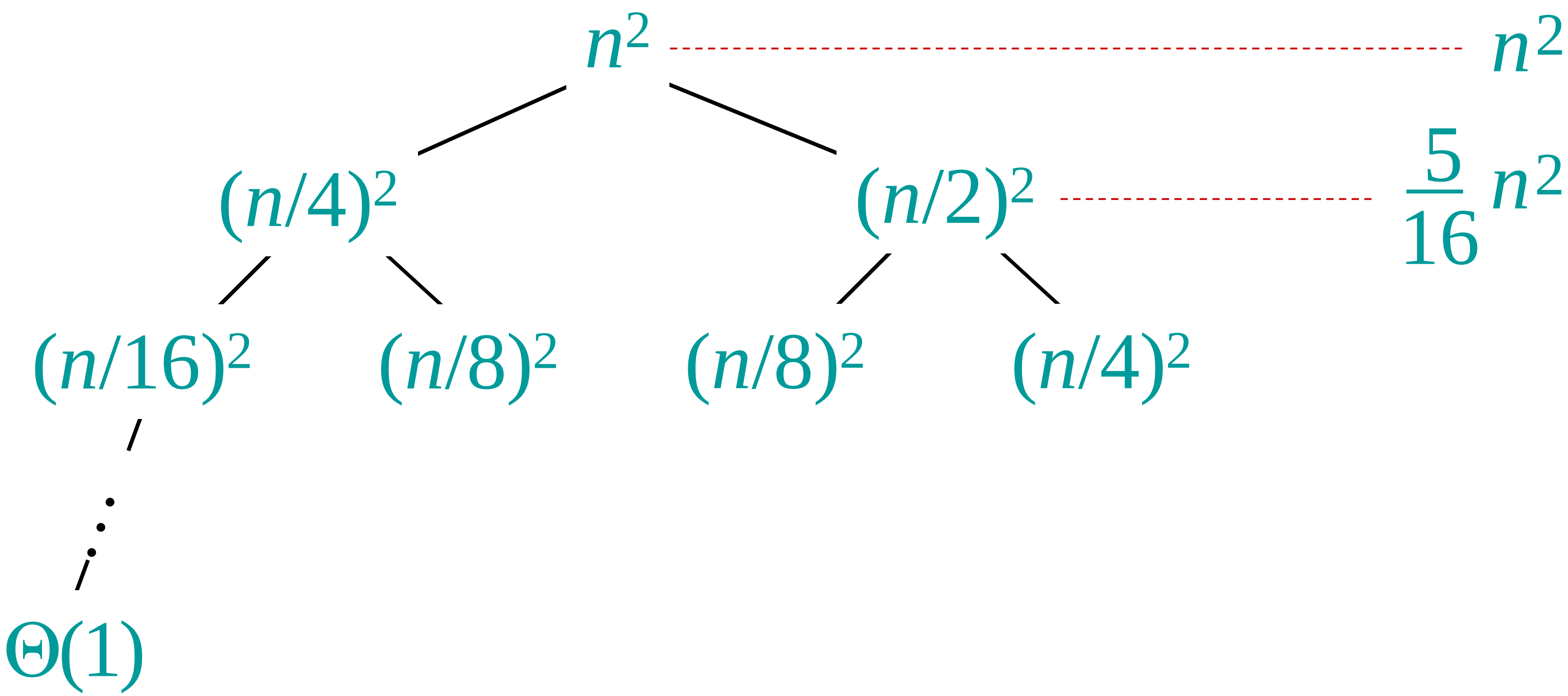
Example of recursion tree

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



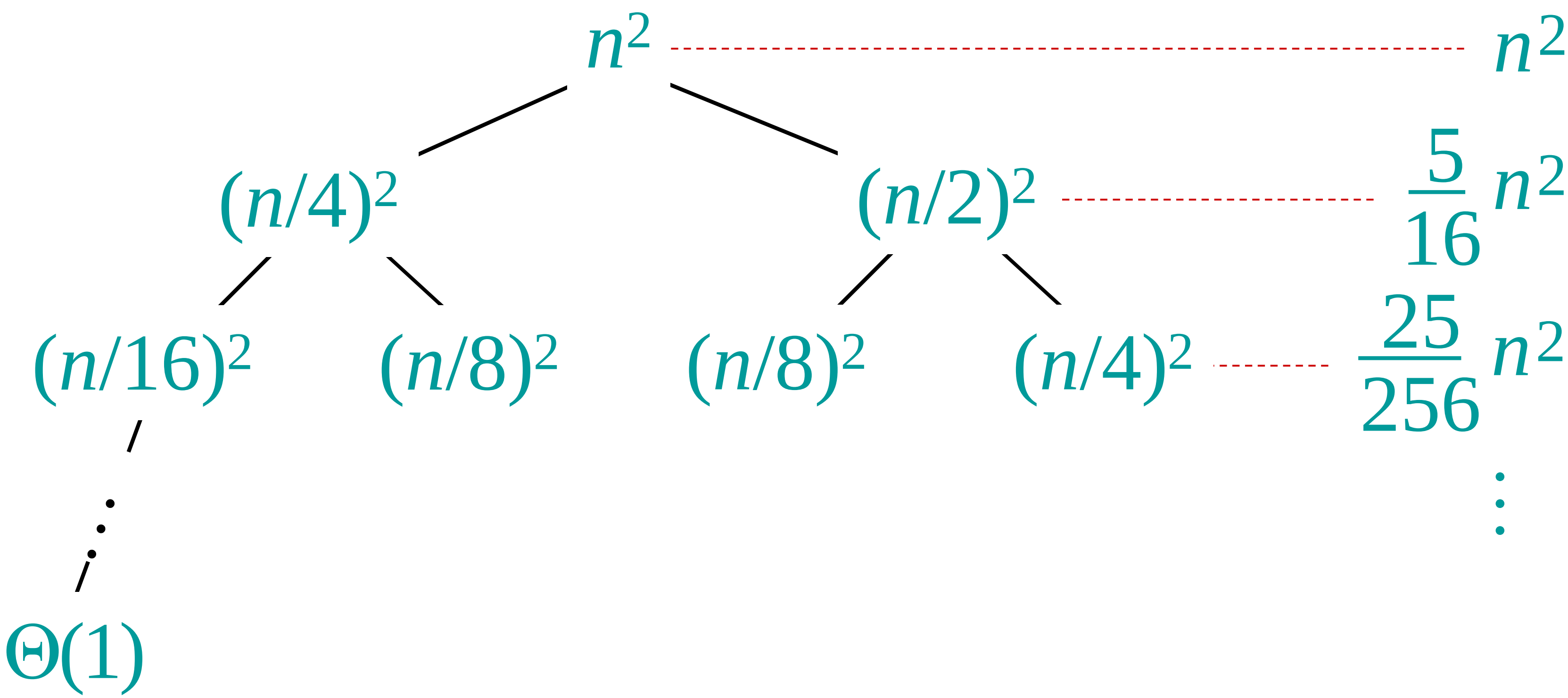
Example of recursion tree

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



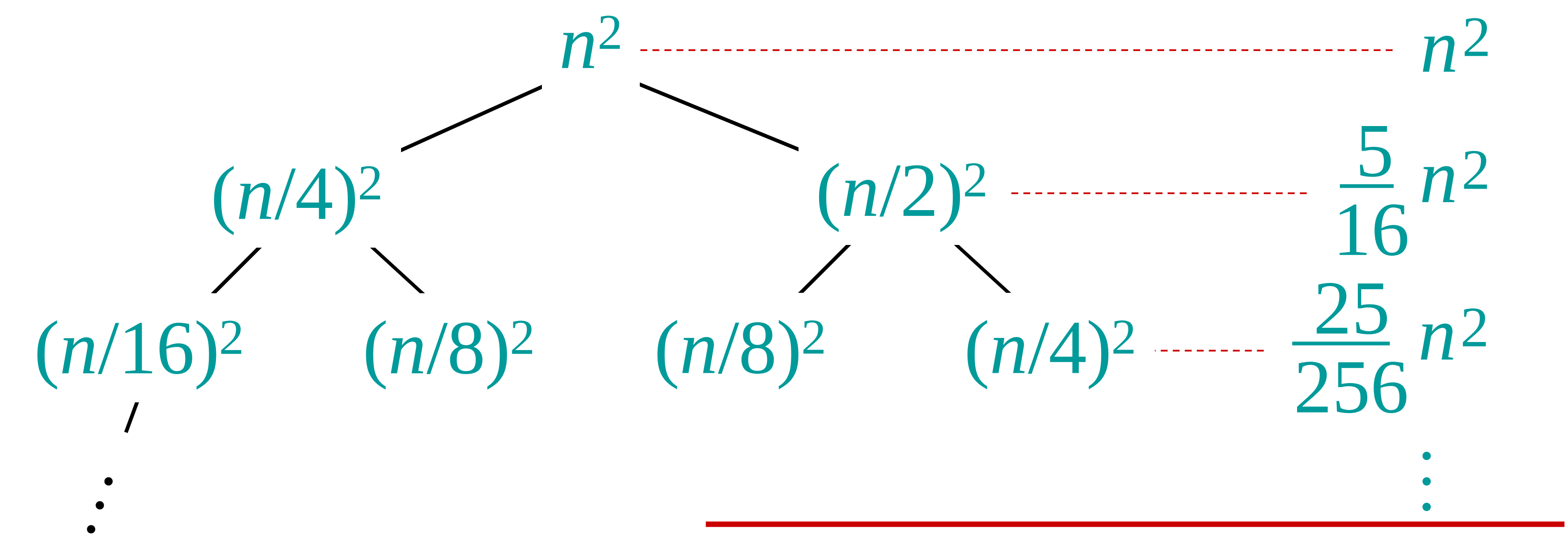
Example of recursion tree

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



Example of recursion tree

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



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

$= \Theta(n^2)$

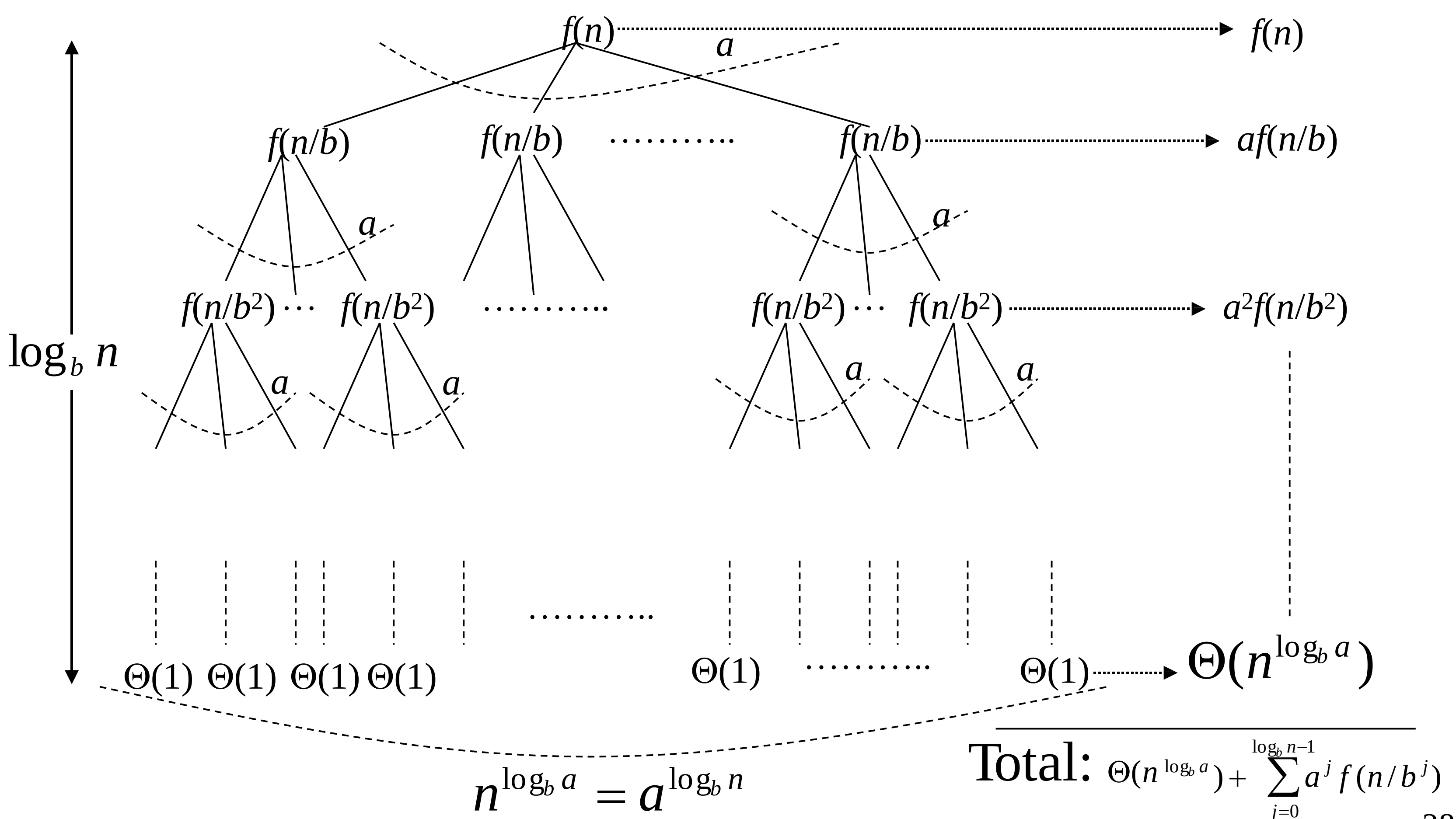
The master method

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)*



Three common cases

The running time *T(n)*

- Dominated by cost at leaves (for solving the minimum subproblems)
- Evenly distributed throughout the tree
- Dominated by cost at the root (for dividing the problem and combining the results)

Solving recurrences amounts to characterizing the dominant term in each case.

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 (within a logarithmic factor to some power).

   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 $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and big enough n.

   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 $c = 1/2$.

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

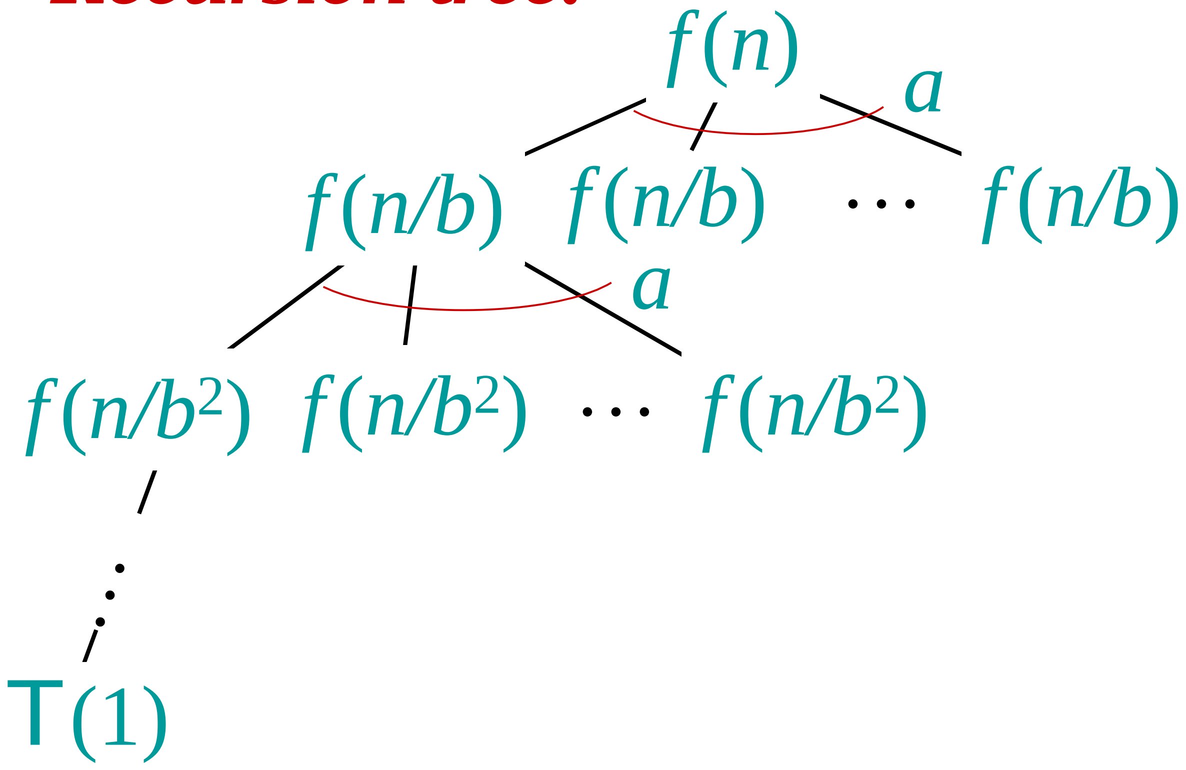
Examples

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 $n^\varepsilon = \omega(\lg n)$.

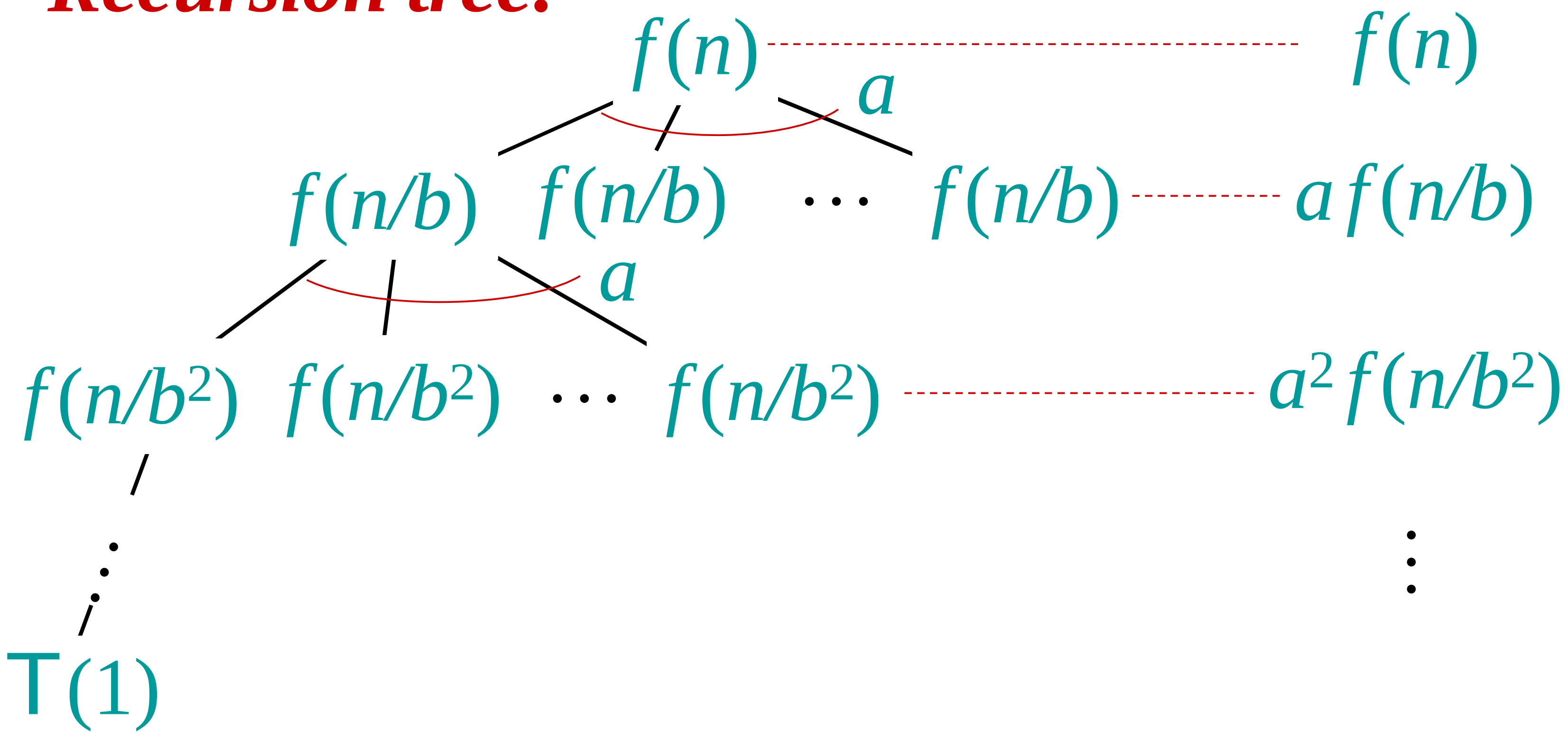
Idea of master theorem

Recursion tree:

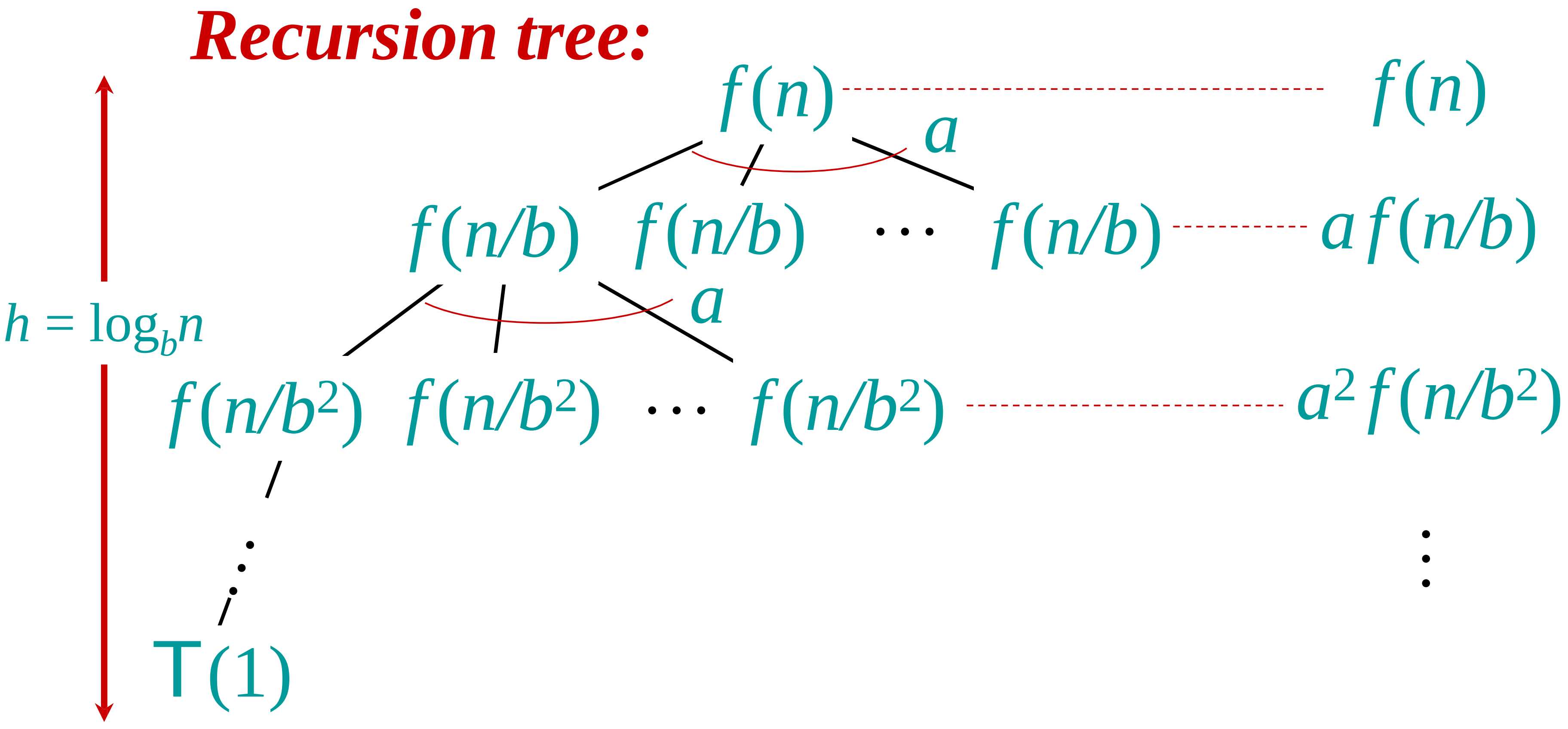


Idea of master theorem

Recursion tree:

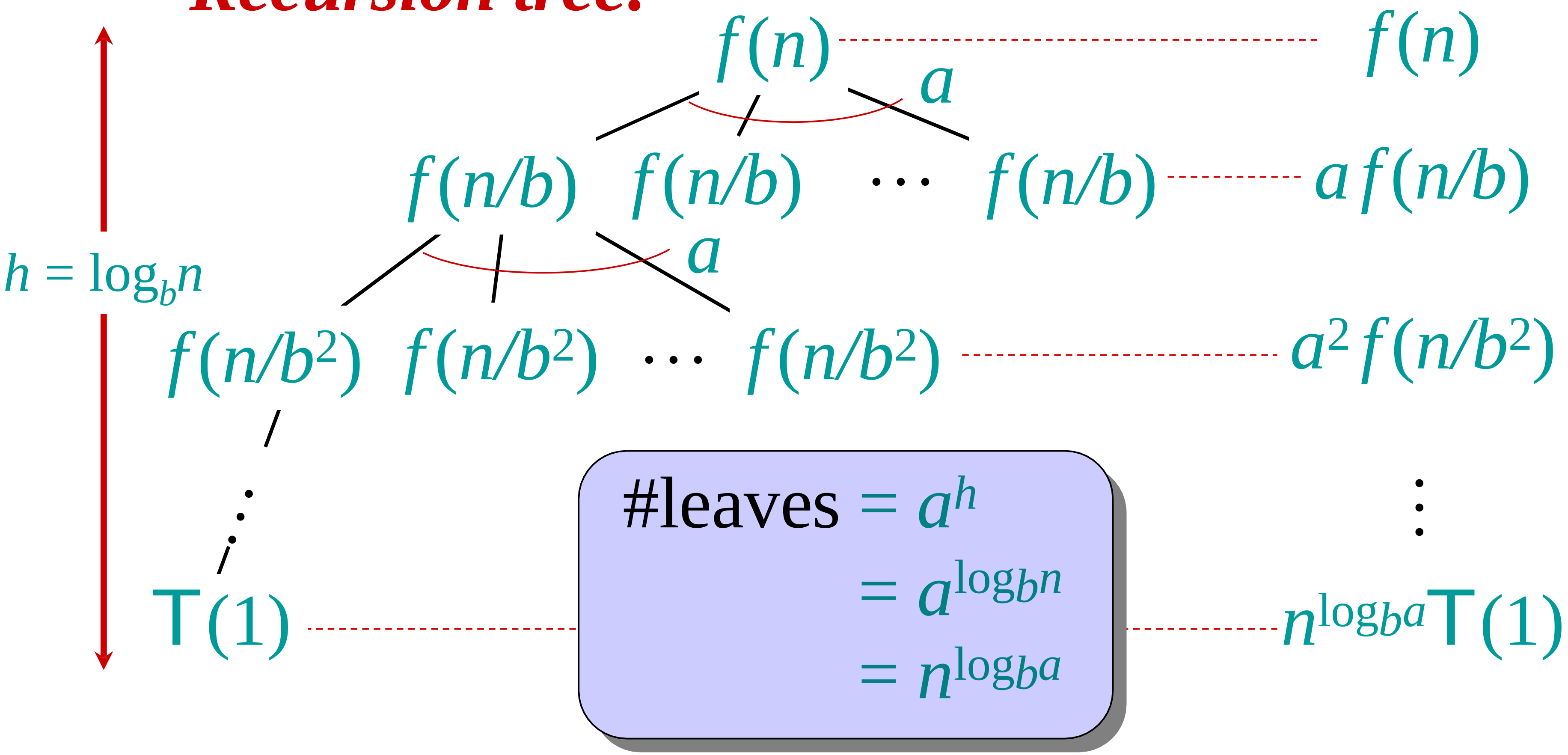


Idea of master theorem



Idea of master theorem

Recursion tree:



Idea of master theorem

Recursion tree:

$$f(n) \quad\cdots\cdots\quad f(n)$$

$$a$$

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

$$a$$

$$h = \log_b n$$

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

$$\vdots$$

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

Case 1: 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:

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

a

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

a

$h = \log_b n$

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

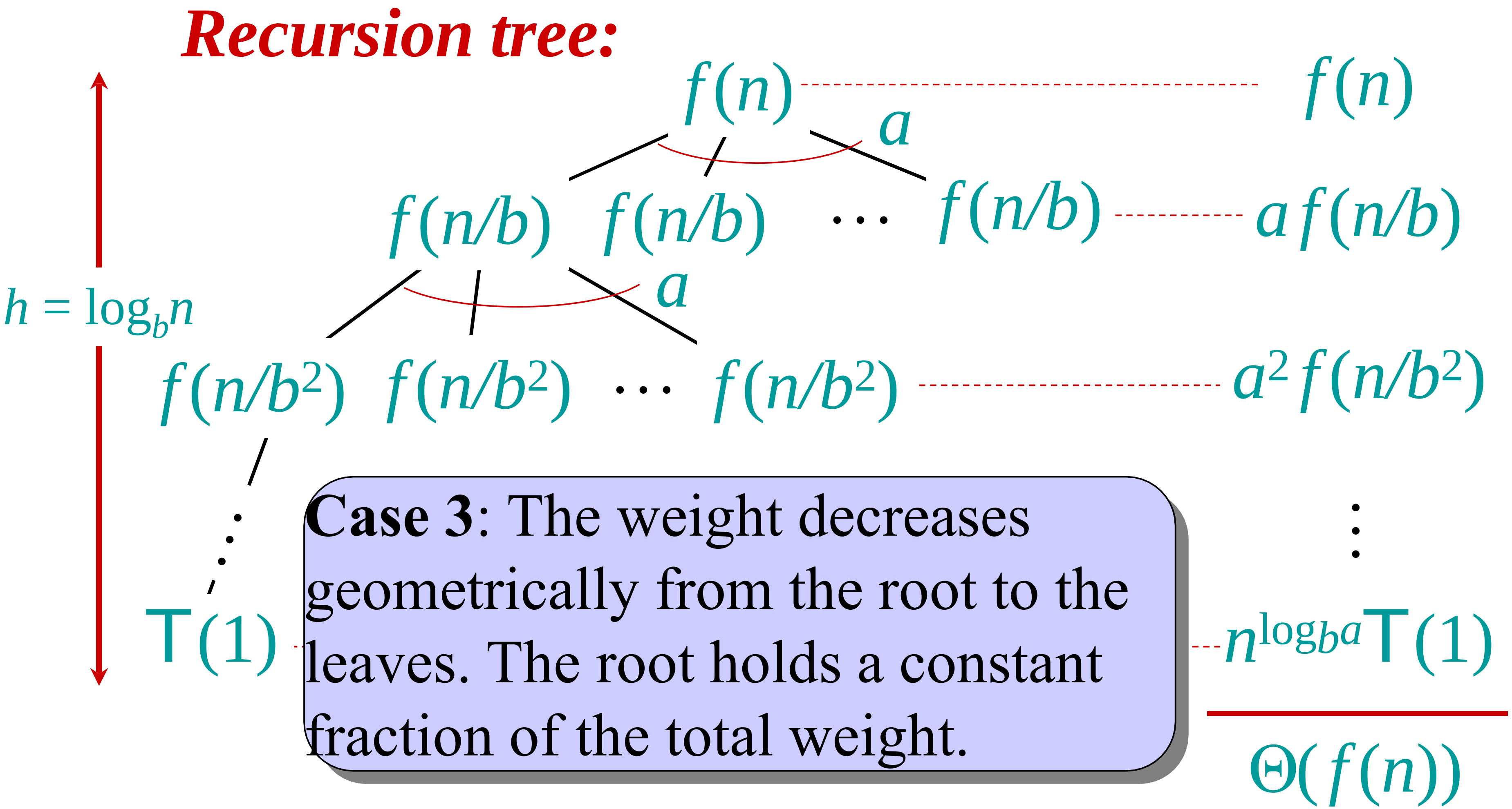
$\vdots$

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

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

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

Idea of master theorem



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

Some advanced sorting algorithms

Recap sorting

Sorting algorithms	Average case	Worst case	Space	Stability	Complexity
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)	Stable	Simple
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)	Stable	Simple
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	Unstable	Simple
Quicksort	O(nlog n)	$O(n^2)$	O(log n)	Unstable	Complex
Heapsort	O(nlog n)	O(nlog n)	O(1)	Unstable	Complex
Mergesort	O(nlog n)	O(nlog n)	O(n)	Stable	Complex

Heapsort

Use data structure *heap* to manage information

Combine the better attributes of insertion sort and merge sort

- $O(n \lg n)$ – like merge sort, unlike insertion sort
- Sorts in place – like insertion, unlike merge sort

Data Structure: Heaps

A *binary* heap data structure *A*

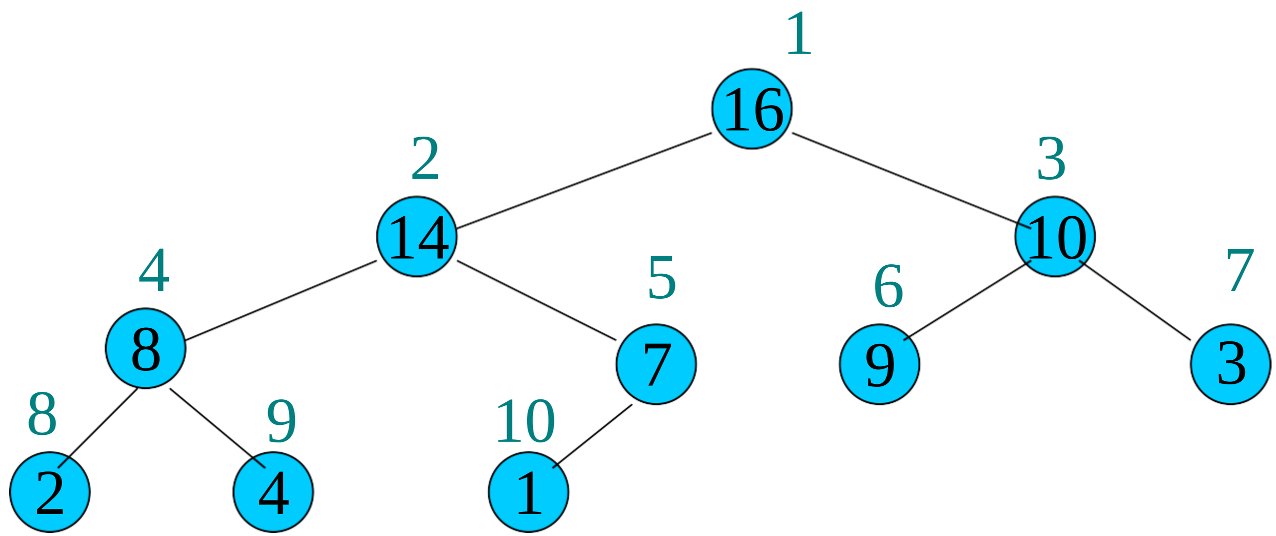
- Simple array
- Viewed as a nearly complete binary tree
- Max-heap property

$$A[\text{parent}(x)] \geq A[x]$$

It means that *the value of a node is at most the value of its parent.*

There are also min-heaps and *k*-ary heaps.

Example of heaps



- Notice the implicit tree links: Children of node i are $2i$ and $2i+1$
- Quickly computed by shifting the binary representation of i left by one bit position and adding 1 as the low-order bit
- Height is $\Theta(\lg n)$ for a heap of size n

Heaps: Extract-Max

Heap-Extract-Max(A)

1. //Removes and returns largest element of A
2. $max \leftarrow A[1]$
3. $A[1] \leftarrow A[n]$
4. $n \leftarrow n - 1$
5. Max-Heapify(A,1) //Remakes heap
6. **return** max

Running time? $\Theta(1)$ + Heapify time.

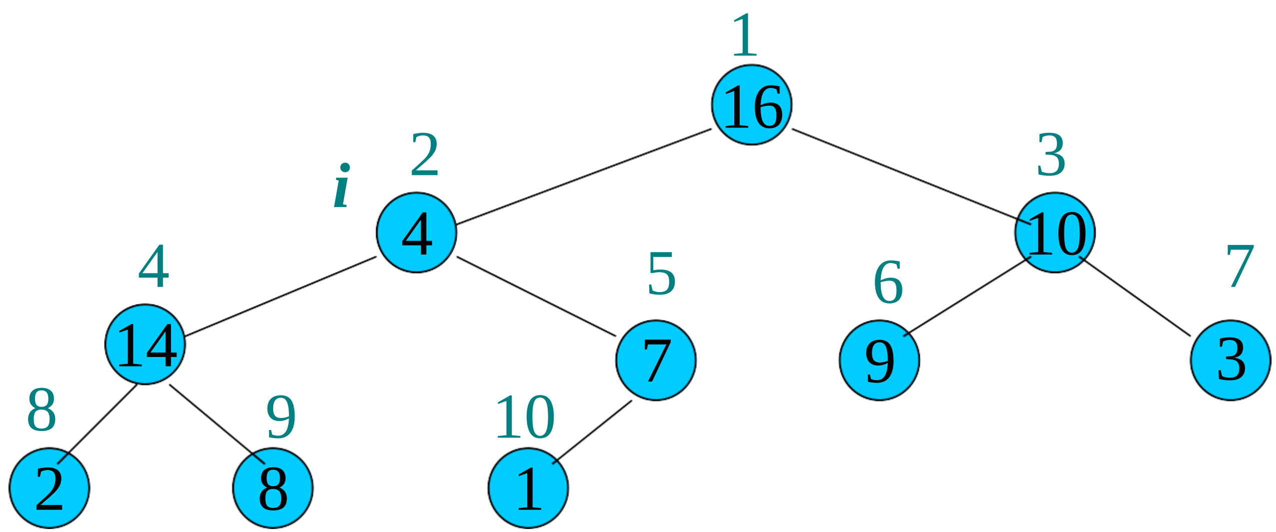
Heaps: Heapify

Max-Heapify (*A, i*)

- *i* is index into array *A.*
- Both binary trees rooted at Left(*i*) and Right(*i*) are max-heaps.
- But, *A*[*i*] may be smaller than its children, thus violating the heap property.
- Heapify makes *A* a heap once more by “floating down” *A*[*i*] in max-heap so that the subtree rooted at *i* obeys the max-heap property.

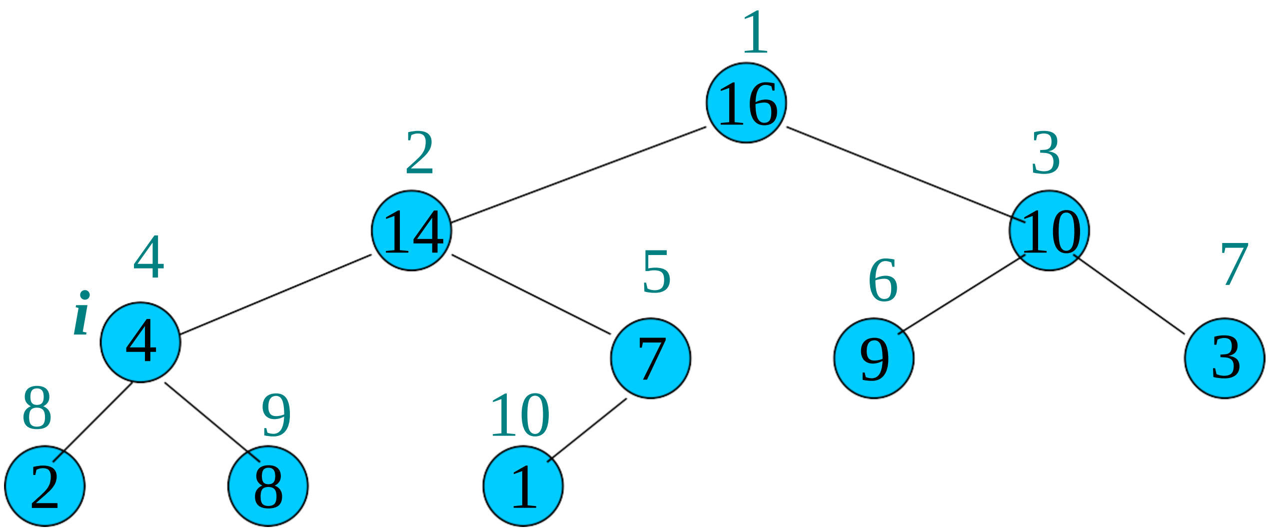
Heaps: Heapify Example

1. Call Heapify(A,2)



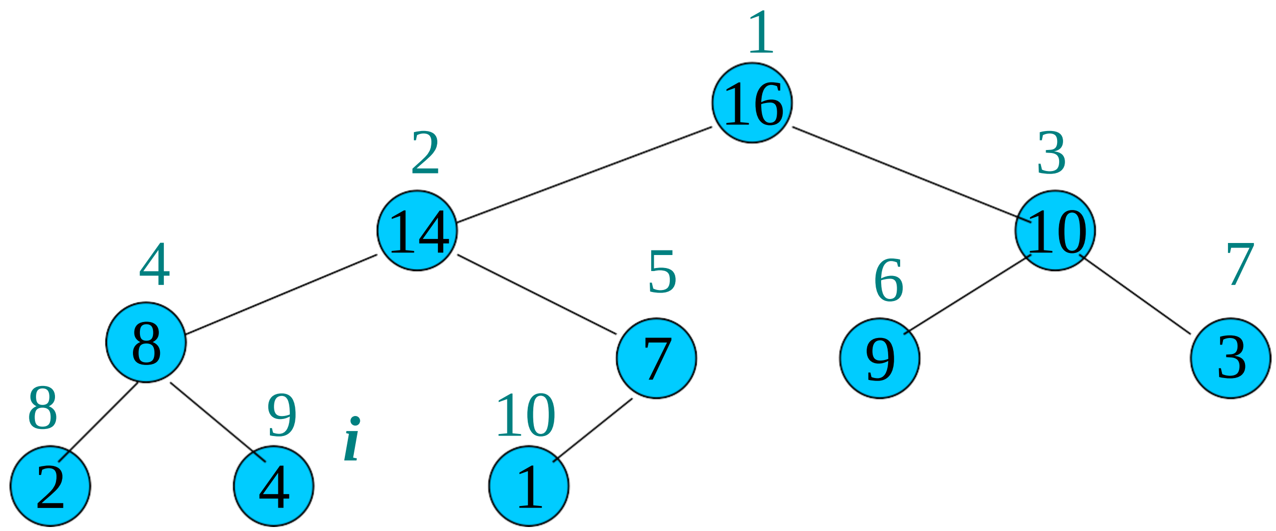
Heaps: Heapify Example (cont.)

2. Exchange *A*[2] with *A*[4] and recursively call Heapify(*A*,4)



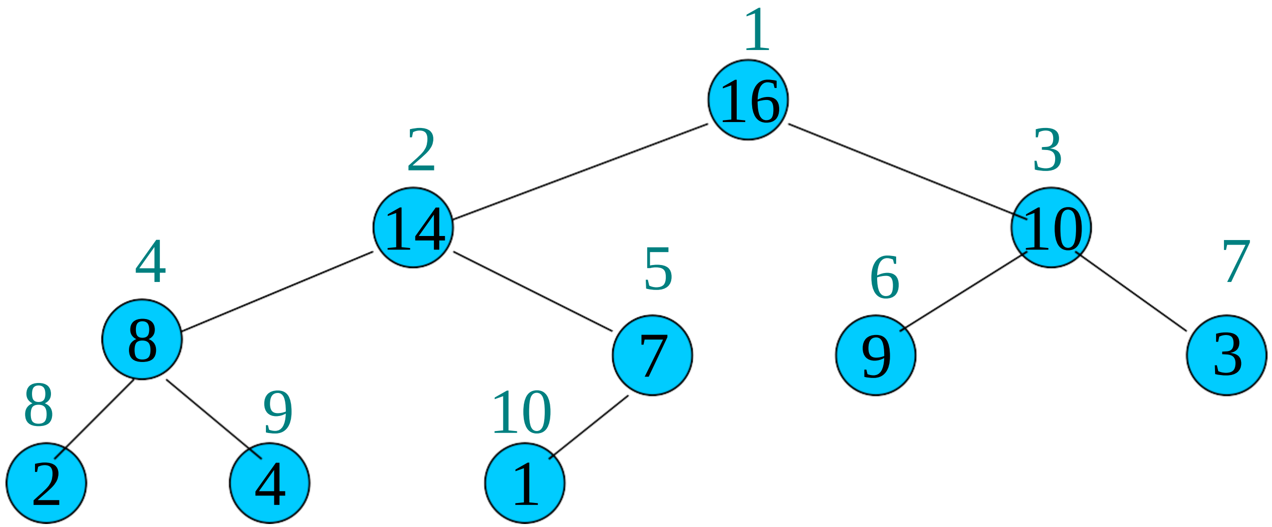
Heaps: Heapify Example (cont.)

3. Exchange *A*[4] with *A*[9] and recursively call Heapify(*A*,9)



Heaps: Heapify Example (cont.)

4. Node 9 has no children, so we are done.



Heaps: Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

- Correctness: induction on the height of i
- The worst-case running time is proportional to the height of $i = O(\lg n)$

Heapsort

Heapsort(A)	Analysis
1. Build-Max-Heap(A)	??
2. for $i \leftarrow n$ **downto** 2	n times
3. **do** exchange $A[1] \leftrightarrow A[i]$	$O(1)$
4. $n \leftarrow n-1$	$O(1)$
5. Heapify(A,1)	$O(\lg n)$

Total Running time:

$O(n\lg n)$ + Build-Heap(A) time

Heapsort: Building a heap

Convert an array $A[1..n]$ where n =length[A], into a heap.

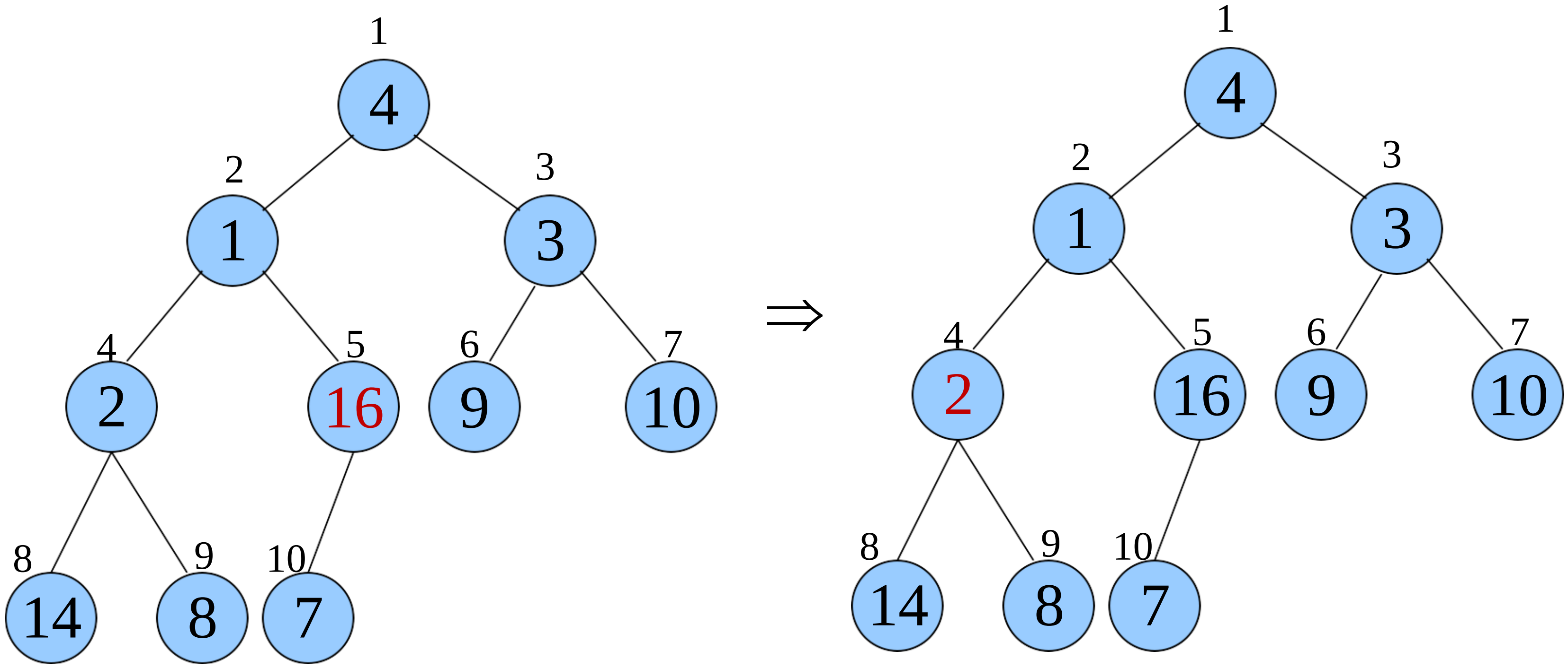
Notice that the elements in the subarray $A[(\lfloor n/2 \rfloor +1)..n]$ are leaves and already 1-element heaps to begin with.

Build-Max-Heap(A)

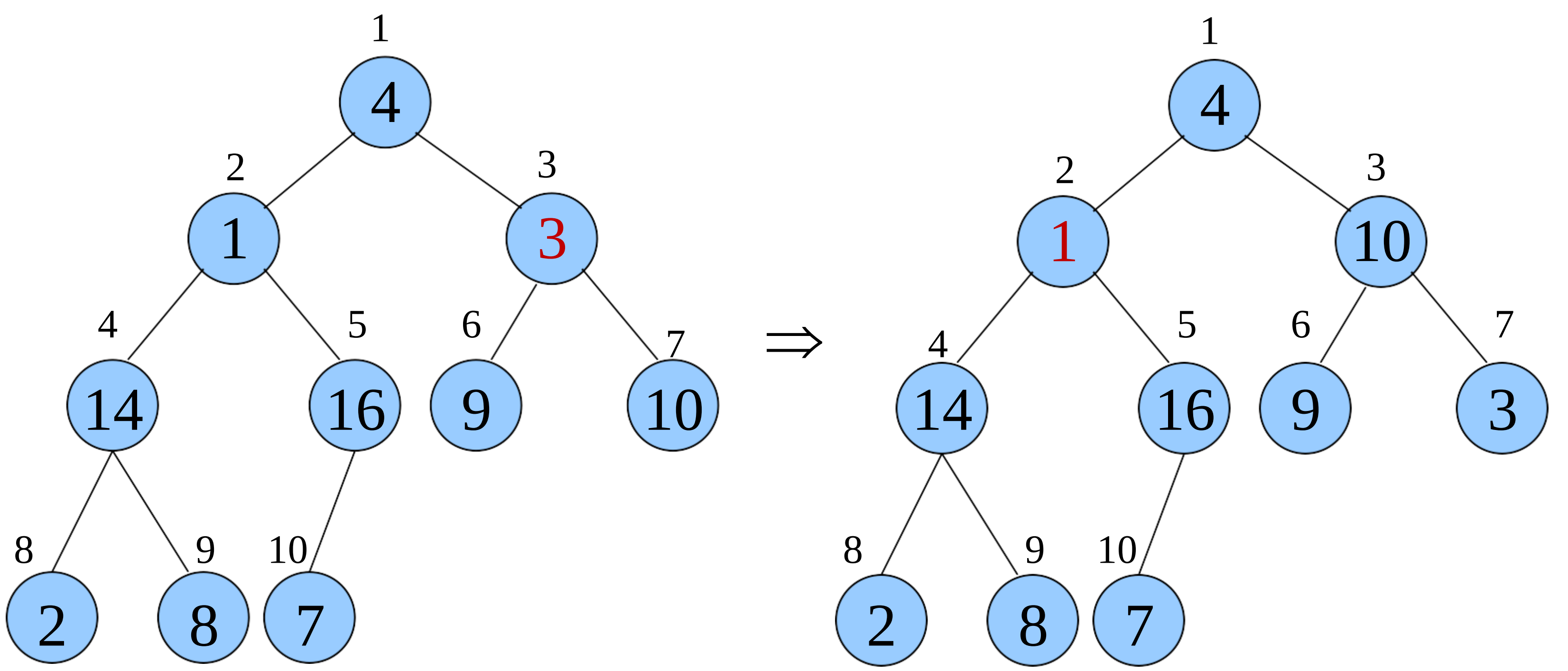
1. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
2. **do** Max-Heapify(A,i)

Build-Max-Heap: Example

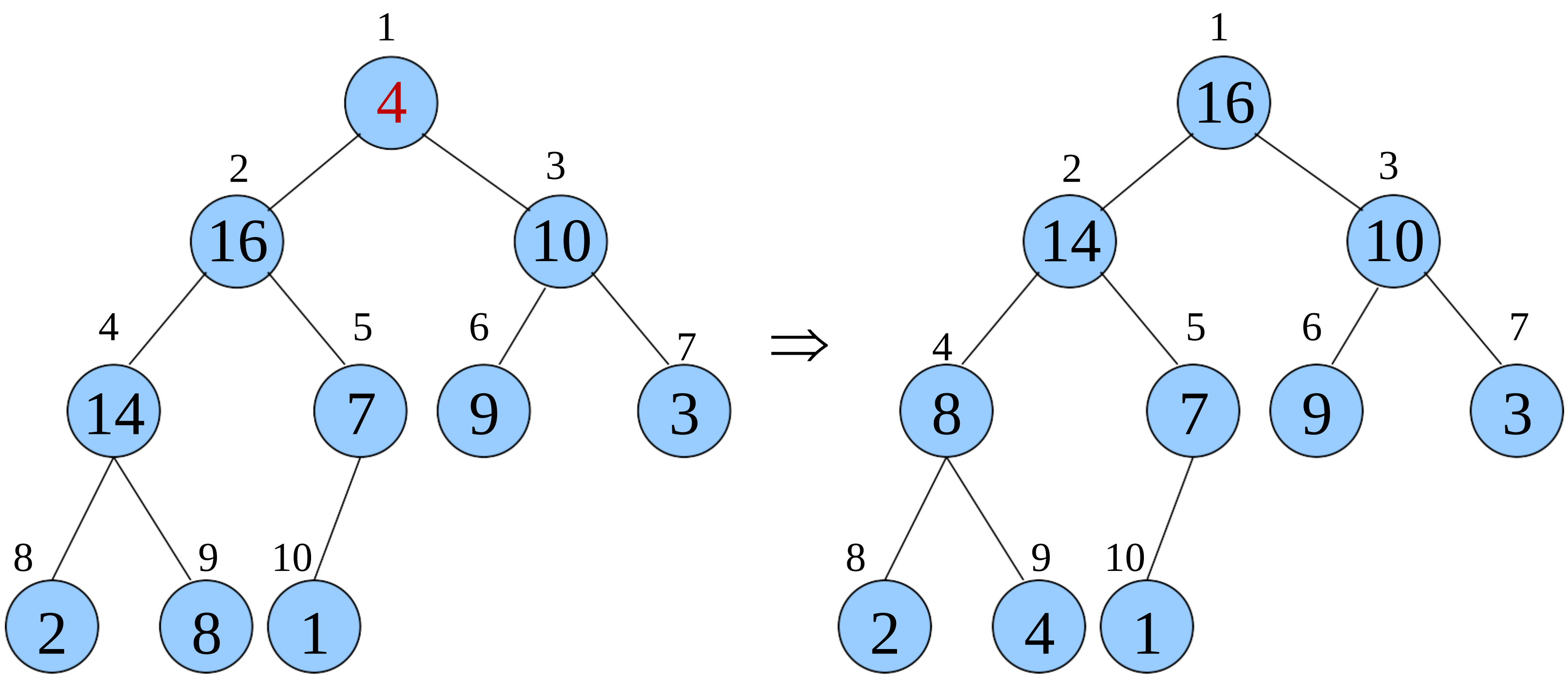
A										
	4	1	3	2	16	9	10	14	8	7



Build-Max-Heap: Example (cont.)



Build-Max-Heap: Example (cont.)



Build-Max-Heap: a simple upper bound

- Correctness: induction on i, all trees rooted at $m > i$ are heaps.
- Running time: makes $O(n)$ calls to Max-Heapify $= O(n \lg n)$
- This is good enough for an $O(n \lg n)$ bound on Heapsort, but sometimes we build heaps for other reasons

Build-max-Heap: a tighter analysis

Time of Heapify = O(height of subtree rooted at i)

An n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\boxed{\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2} \rightarrow \quad = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n) .$$

Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- Remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$

Divide and conquer

Quicksort an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq x$	x	$> x$

2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

Key: *Linear-time partitioning subroutine.*

Partitioning subroutine

PARTITION(A, p, q) $\triangleright$ $A[p\,.\,.\,q]$

  $x \leftarrow A[p]$ $\triangleright$ pivot $= A[p]$

  $i \leftarrow p$

  for $j \leftarrow p + 1$ **to** q

    do if $A[j] \leq x$

      then $i \leftarrow i + 1$

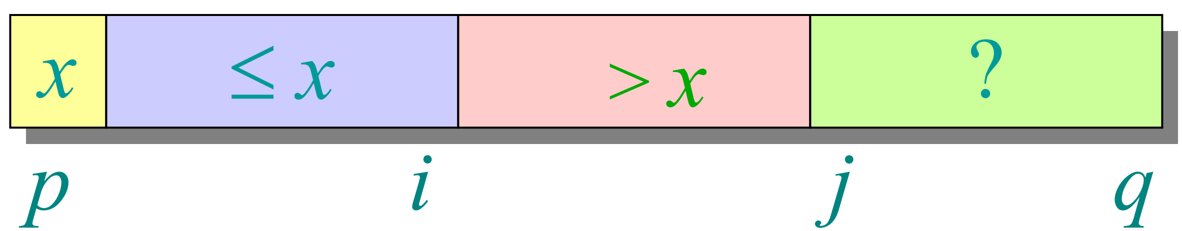
        exchange $A[i] \leftrightarrow A[j]$

  exchange $A[p] \leftrightarrow A[i]$

  return i

Running time $= O(n)$ for n elements.

Invariant:



Example of partitioning

6	10	13	5	8	3	2	11
i	j						

Example of partitioning

6	10	13	5	8	3	2	11
i		j					

Example of partitioning

6	10	13	5	8	3	2	11

i $\qquad\qquad\qquad \bullet\!\!\longrightarrow j$

Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

i j

Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11
	i			j			

Example of partitioning

6	10	13	5	8	3	2	11

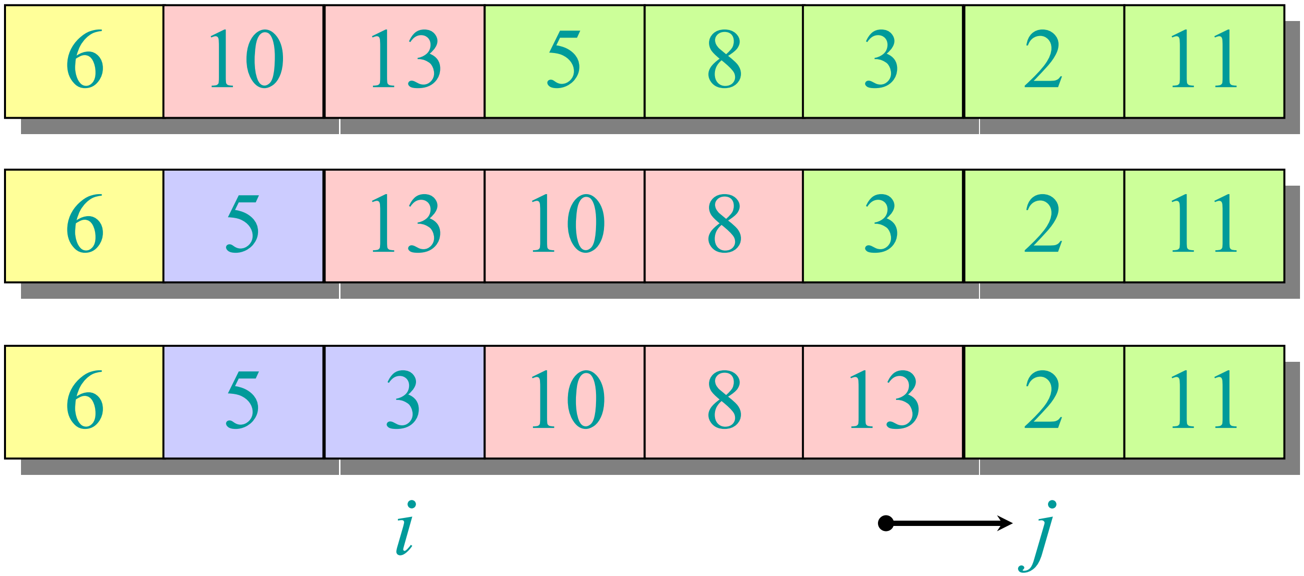
6	5	13	10	8	3	2	11
	i				j		

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11

i j

Example of partitioning



Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

i j

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
			i				$\longrightarrow j$

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

i j

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11

i

Pseudocode for quicksort

QUICKSORT(A, p, r)
  if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r)
      QUICKSORT$(A, p, q-1)$
      QUICKSORT$(A, q+1, r)$

Initial call: QUICKSORT$(A, 1, n)$

Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.

Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

(arithmetic series)

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

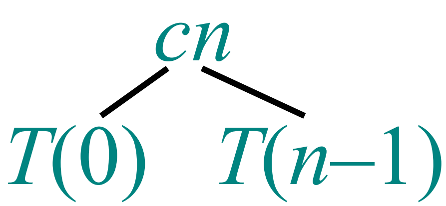
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

$T(n)$

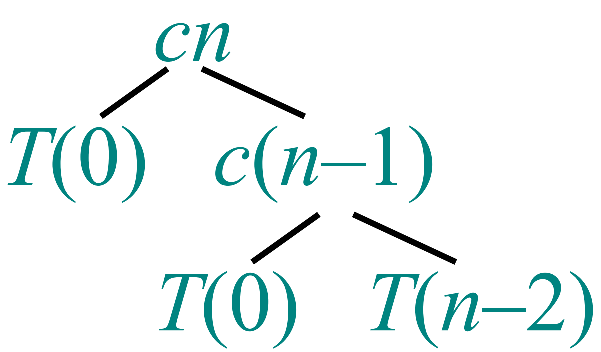
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



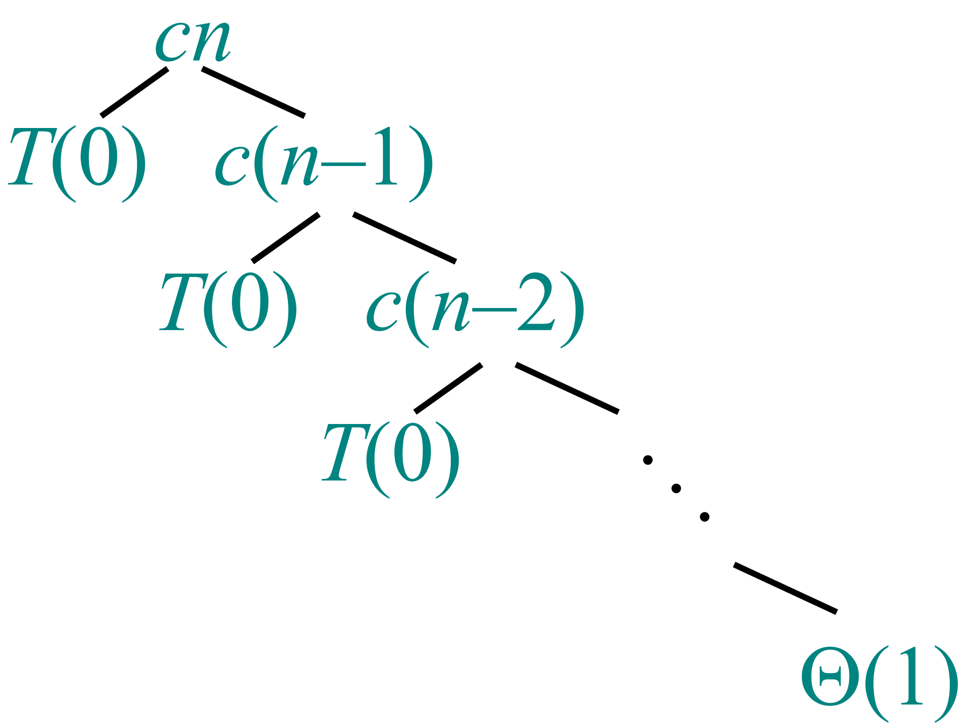
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



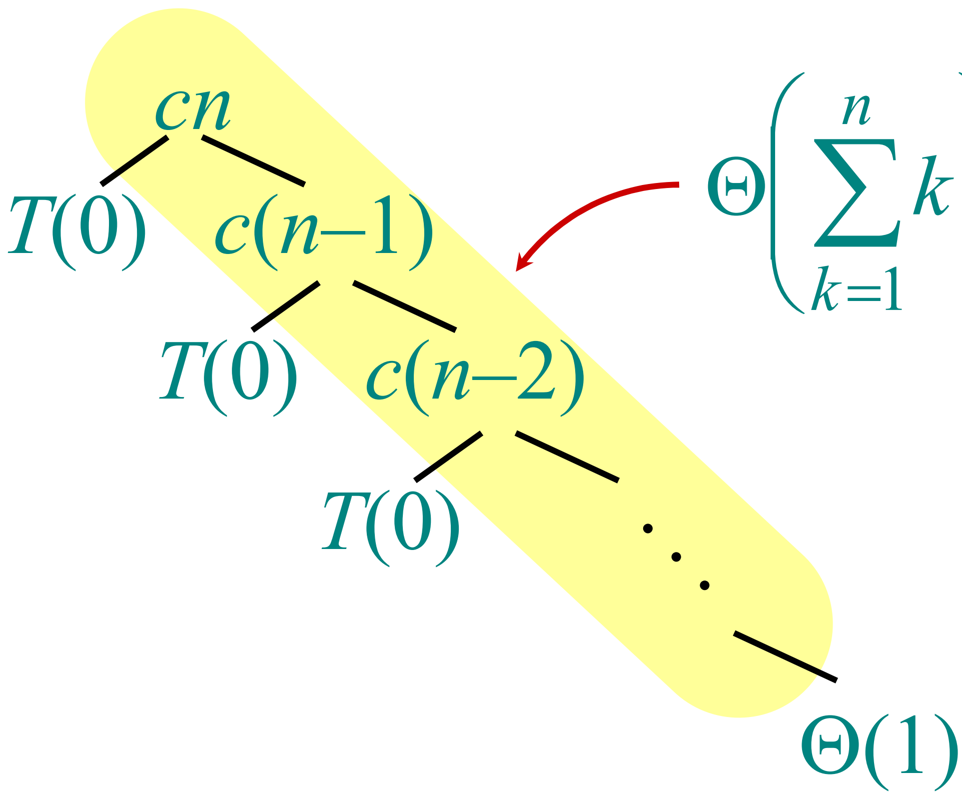
Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$



Worst-case recursion tree

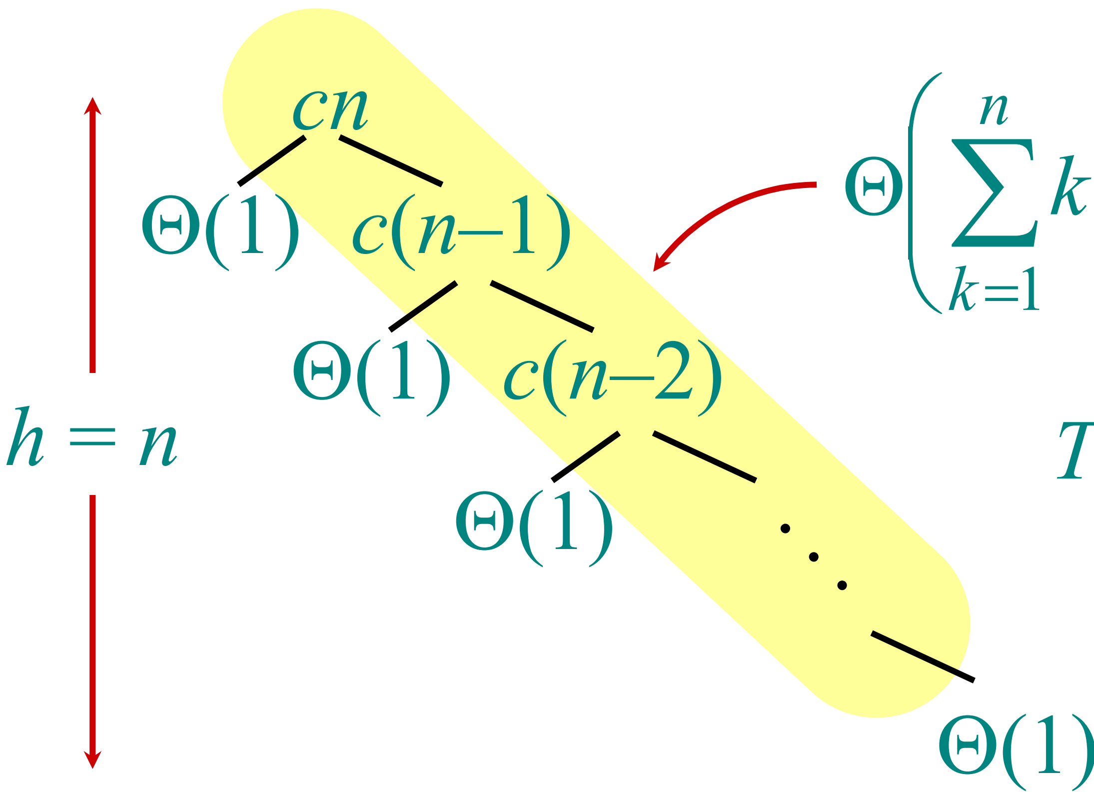
$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

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

Best-case analysis

(For intuition only!)

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$ (same as merge sort)

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

Analysis of “almost-best” case

$T(n)$

Analysis of "almost-best" case

$$cn$$

$$T\left(\tfrac{1}{10}n\right) \qquad\qquad T\left(\tfrac{9}{10}n\right)$$

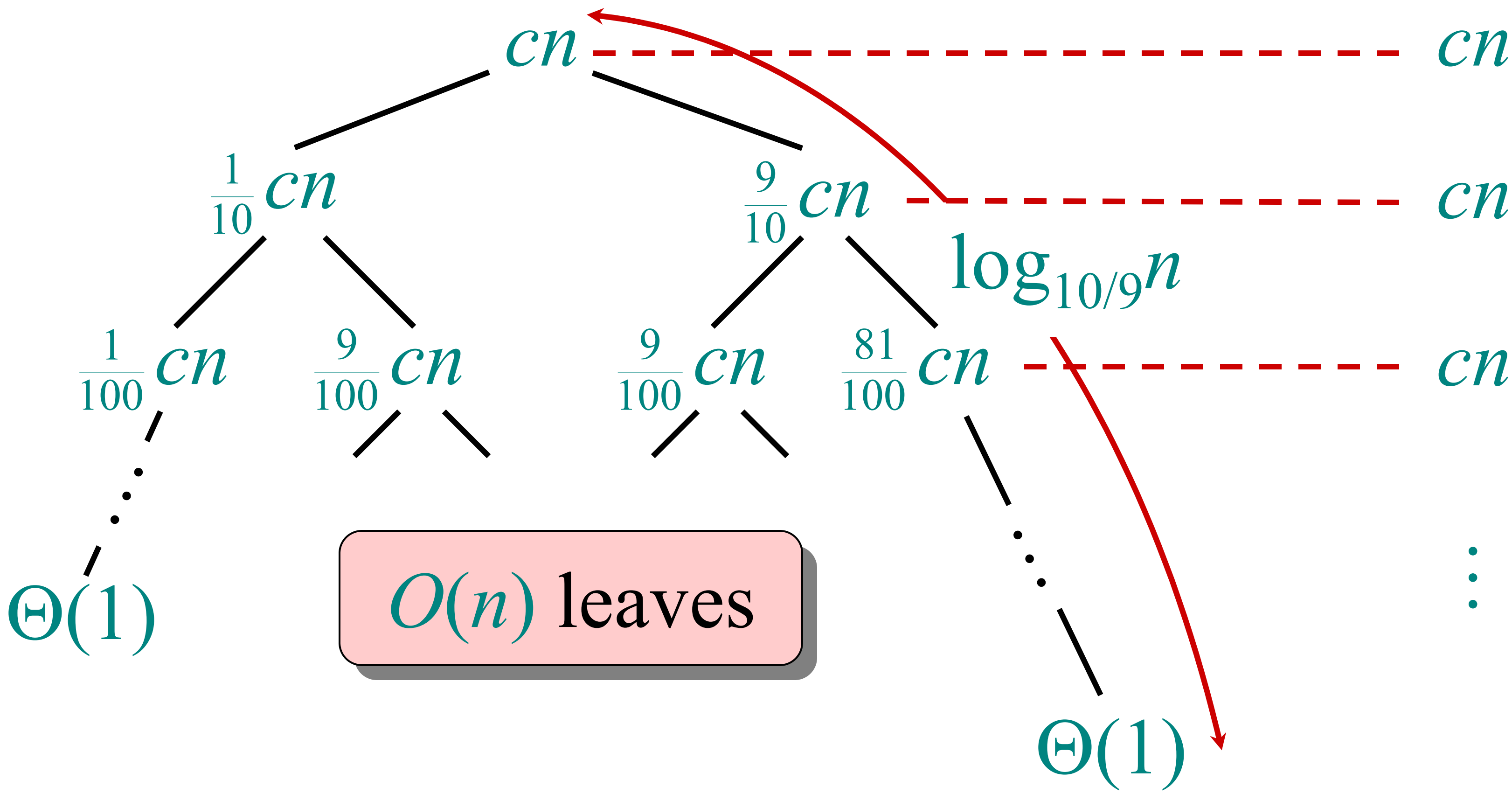
Analysis of “almost-best” case

$$cn$$

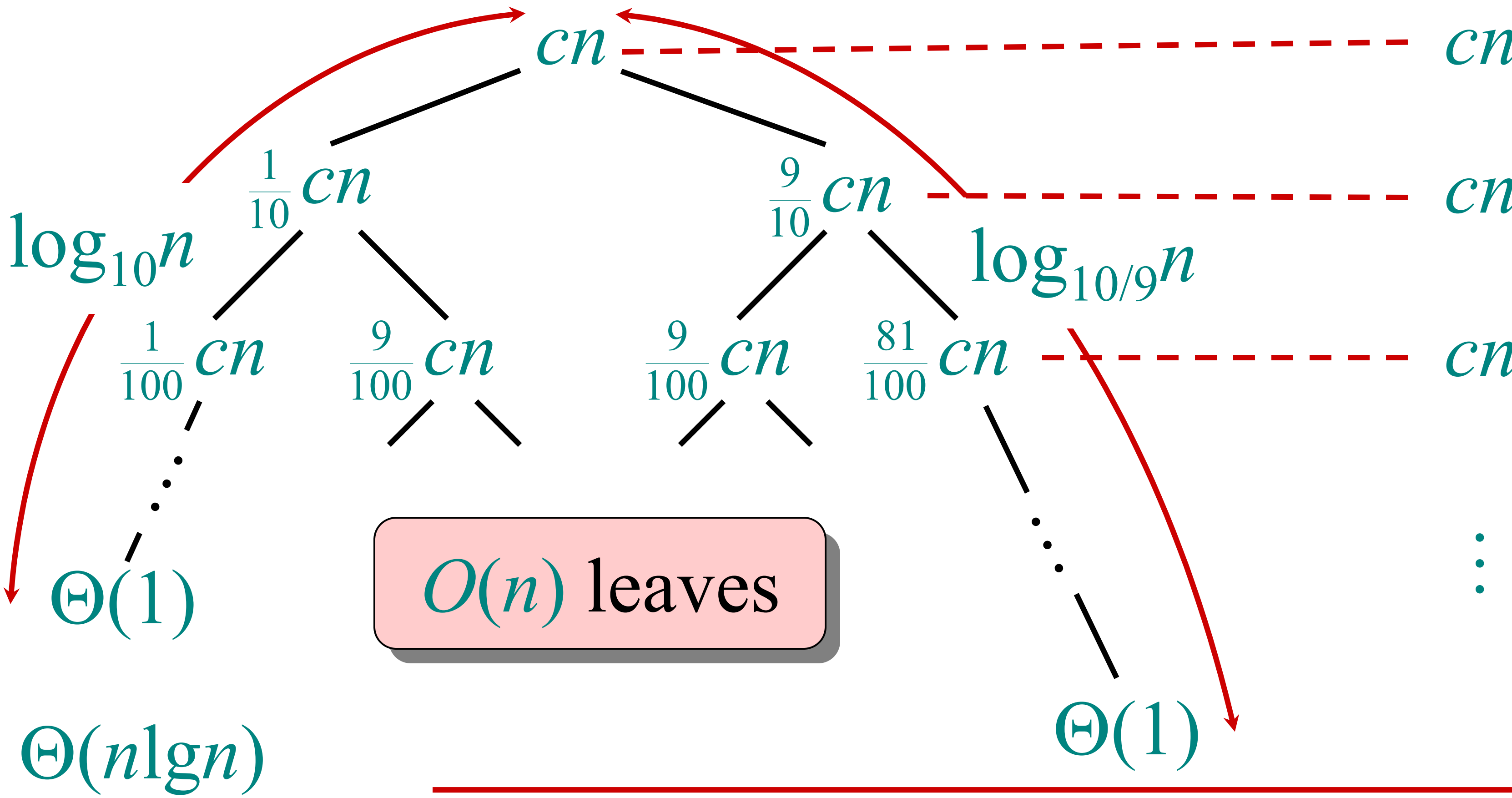
$$\frac{1}{10}cn \qquad \frac{9}{10}cn$$

$$T\left(\frac{1}{100}n\right) \quad T\left(\frac{9}{100}n\right) \qquad T\left(\frac{9}{100}n\right) \quad T\left(\frac{81}{100}n\right)$$

Analysis of "almost-best" case



Analysis of "almost-best" case



$$cn\log_{10}n \le T(n) \le cn\log_{10/9}n + O(n)$$

More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{\textit{lucky}}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{\textit{unlucky}}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

How can we make sure we are usually lucky?

Randomized quicksort

IDEA: Partition around a ***random*** element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Randomized Quicksort

New "randomized" partitioning: swap before actually partitioning.

Randomized-Partition(A, p, r)

1. $i \leftarrow$ Random(p, r)
2. exchange $A[p] \leftrightarrow A[i]$
3. return Partition(A, p, r)

Randomized-Quicksort(A, p, r)

1. **if** $p < r$
2.   **then** $q \leftarrow$ Randomized-Partition(A, p, r)
3.   Randomized-Quicksort($A, p, q–1$)
4.   Randomized-Quicksort($A, q+1, r$)

Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if } \textsc{Partition} \text{ generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Analysis (continued)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \bigl(T(k) + T(n-k-1) + \Theta(n)\bigr).$$

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

Take expectations of both sides.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

Linearity of expectation.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Independence of X_k from other random choices.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = 1/n$.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.

Hairy recurrence

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \le a\,n \lg n$ for constant $a > 0$.

- Choose a large enough so that $a\,n \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Substitution method

$$E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

Substitute inductive hypothesis.

Substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

Use fact.

Substitution method

$$E[T(n)] \leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

Express as ***desired – residual***.

Substitution method

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&= \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an\lg n\,,
\end{aligned}
$$

if a is chosen large enough so that $an/4$ dominates the $\Theta(n)$.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.

How fast can we sort?

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

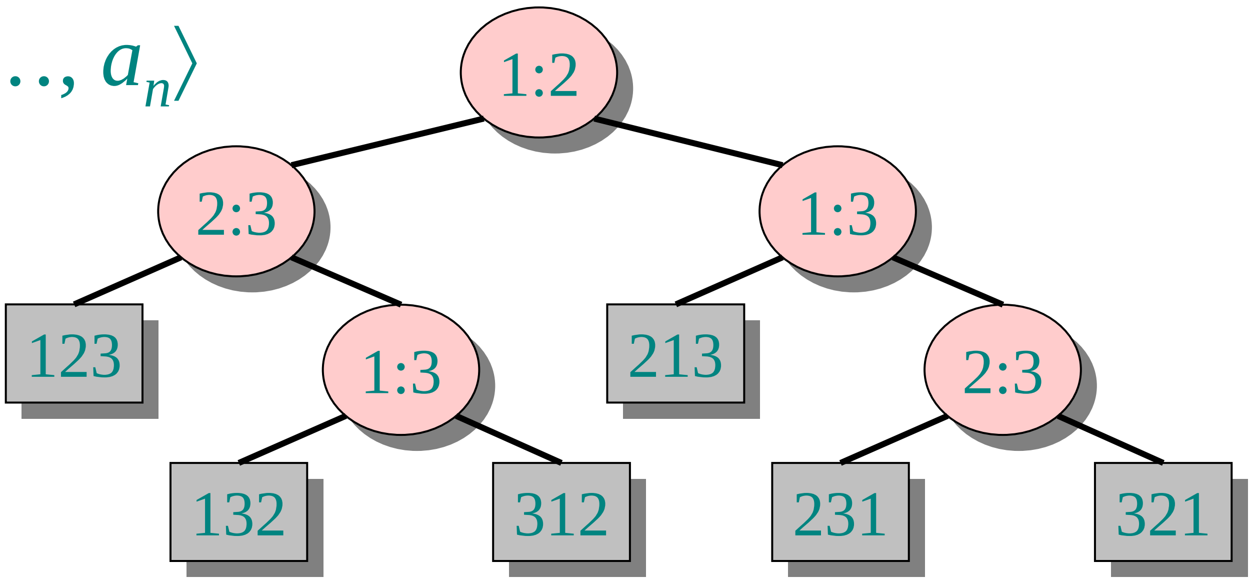
The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.

Decision-tree example

Sort $\langle a_1, a_2, \ldots, a_n \rangle$

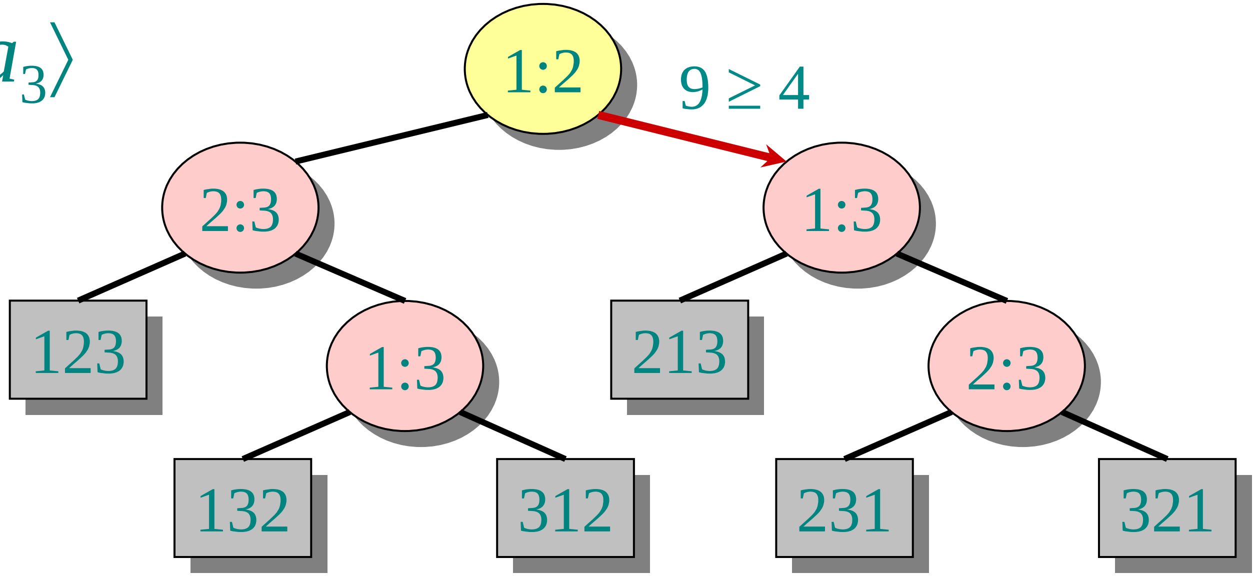


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

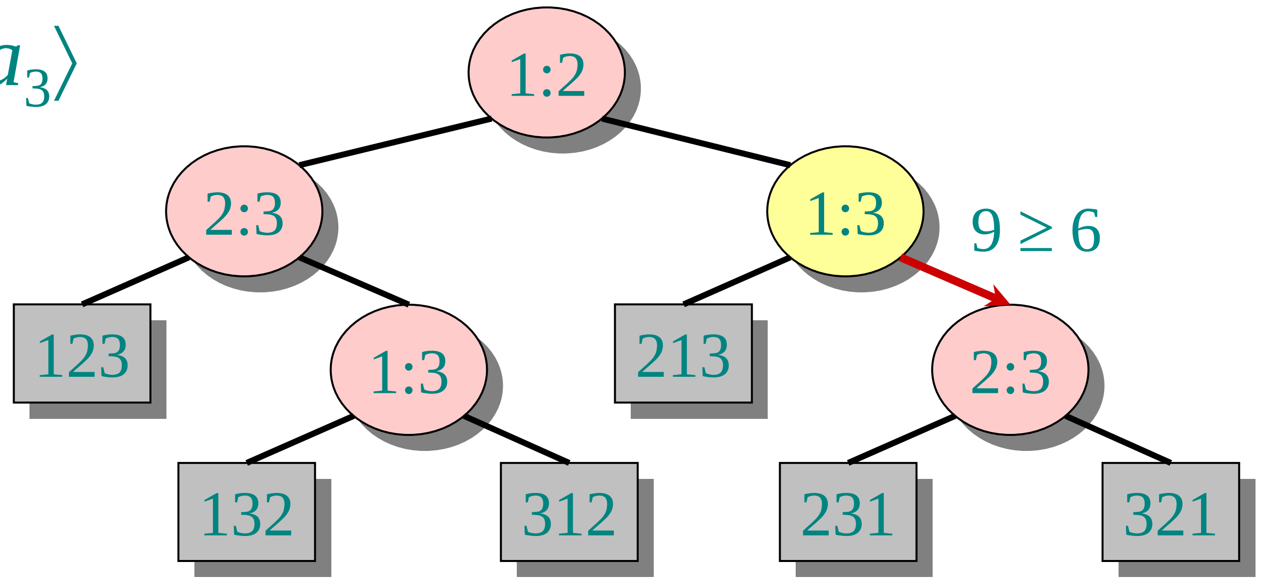


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle\ 9, 4, 6\ \rangle$:

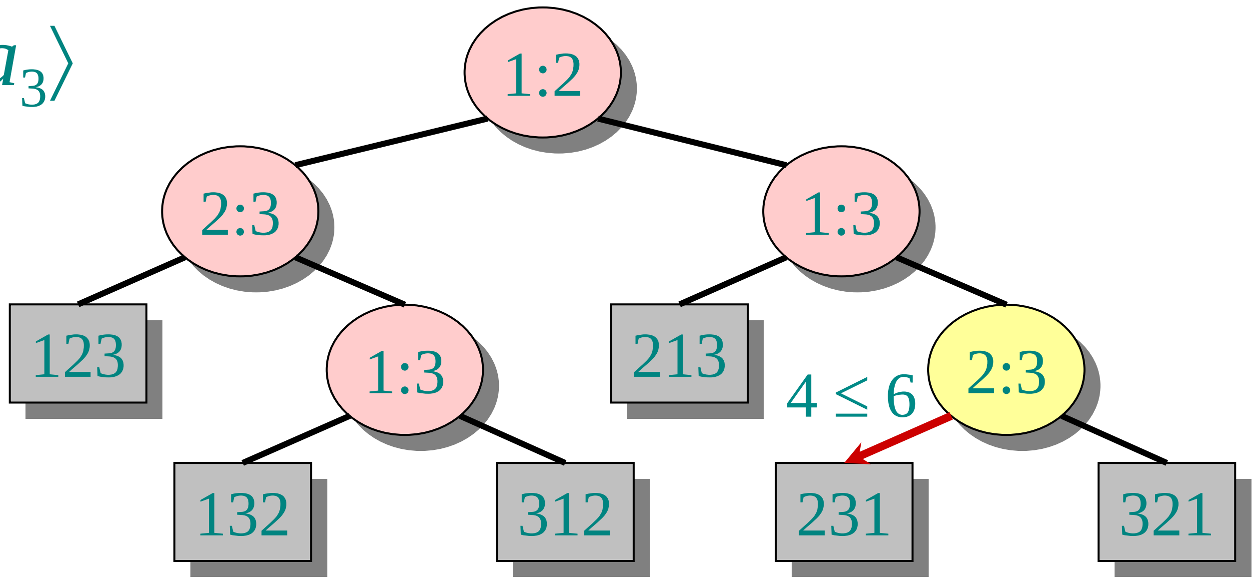


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

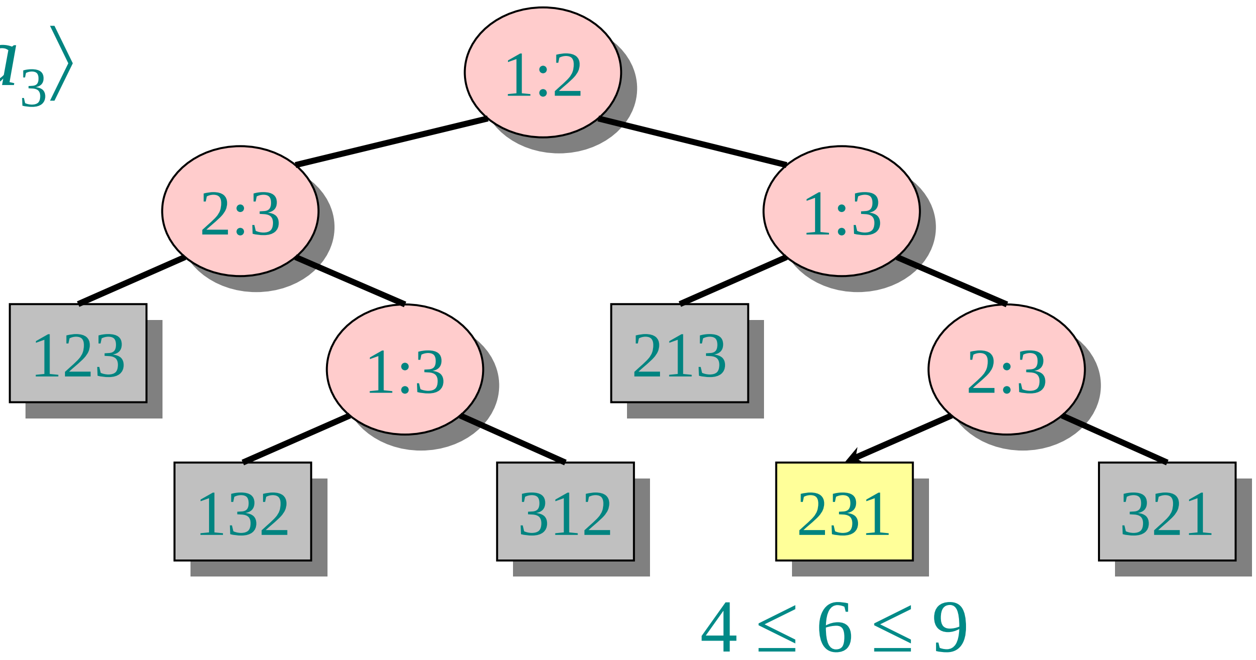


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle\ 9, 4, 6\ \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore\ h &\geq \lg(n!) && \text{(lg is mono. increasing)} \\
&\geq \lg((n/e)^n) && \text{(Stirling's formula)} \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n). \quad \square
\end{aligned}
$$

Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. ■

Sorting in linear time

Counting sort: No comparisons between elements.

- ***Input***: $A[1 \,.\,.\, n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \,.\,.\, n]$, sorted.
- ***Auxiliary storage***: $C[1 \,.\,.\, k]$.

Counting sort

for $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ **to** k
    do $C[i] \leftarrow C[i] + C[i–1]$ $\triangleright$ $C[i] = |\{\text{key} \le i\}|$
for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
        $C[A[j]] \leftarrow C[A[j]] – 1$

Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Loop 1

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	1

B:

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	0	1

B:

for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	2

B:

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i] = |\{\text{key} \leq i\}|$

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i\text{–}1]$ $\triangleright\ C[i] = |\{\text{key} \le i\}|$

Loop 3

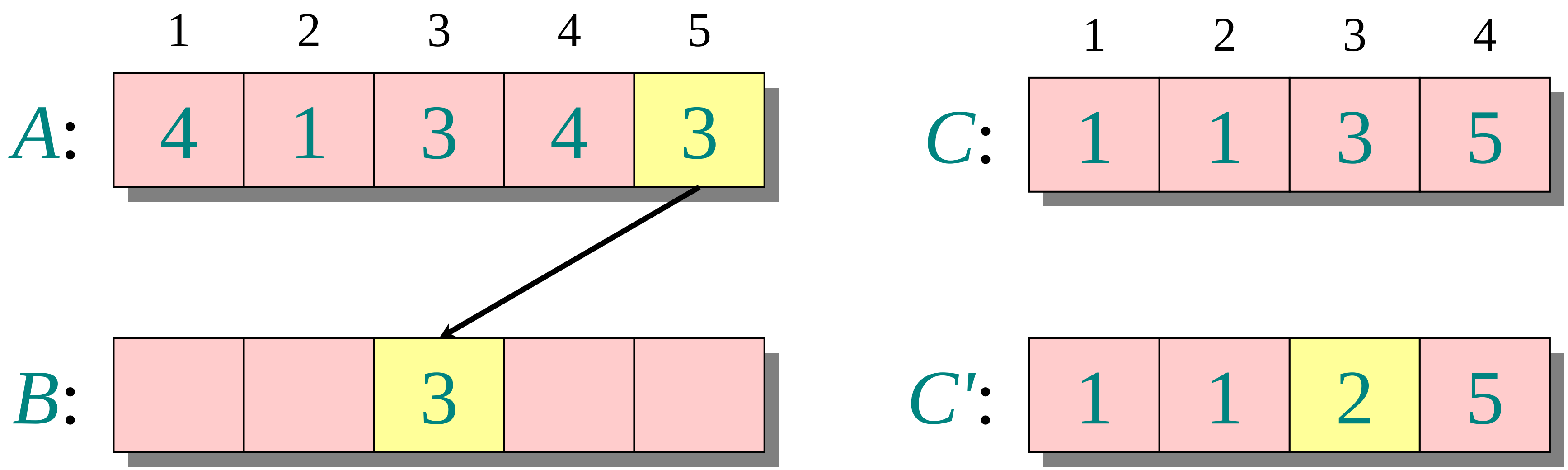
	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright\ C[i] = |\{\text{key} \le i\}|$

Loop 4

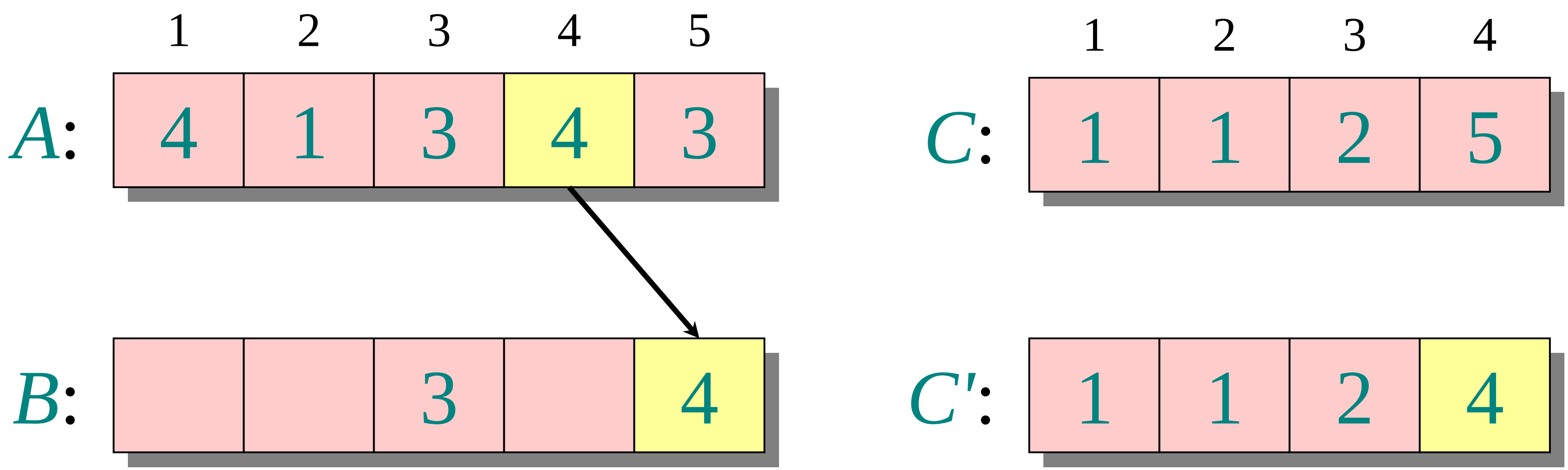


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

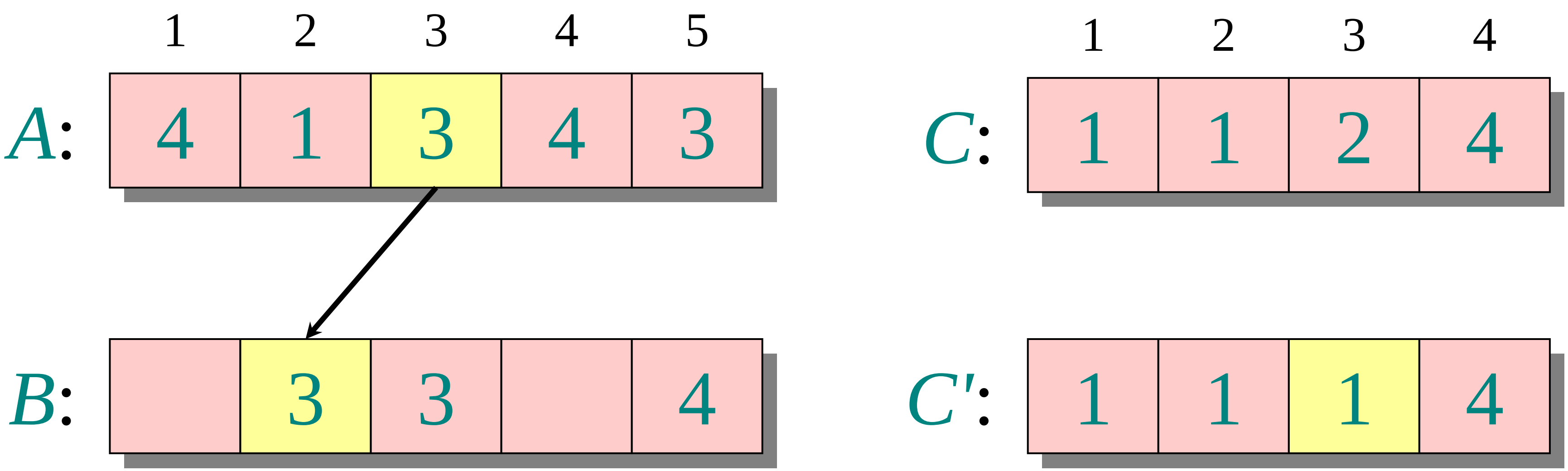


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

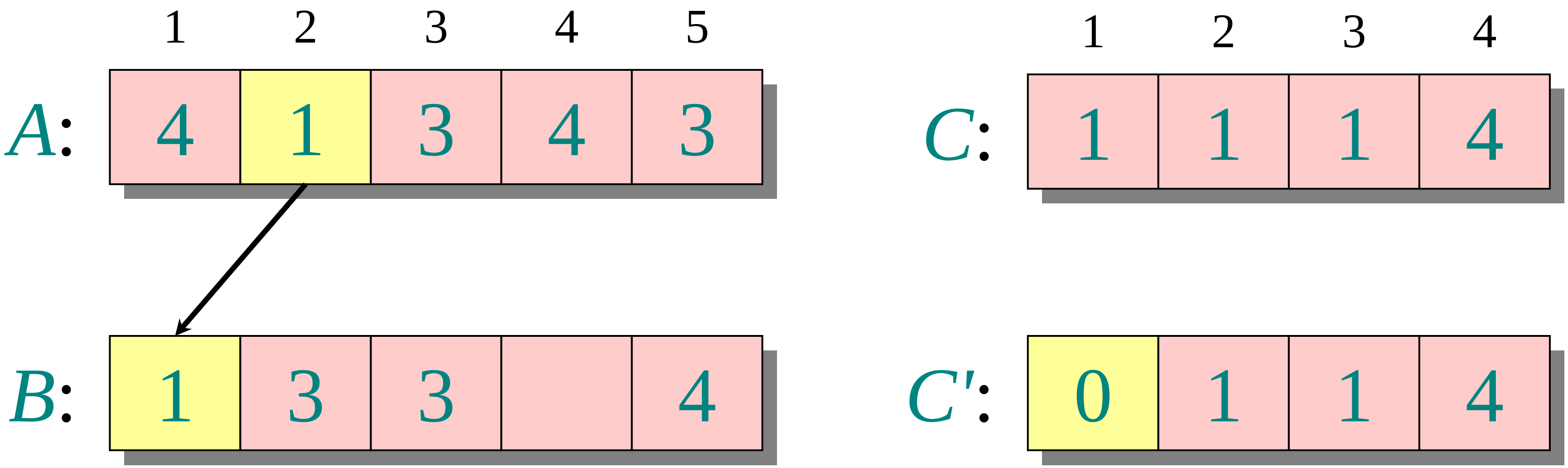


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	1	1	4

B:

1	3	3	4	4

C':

0	1	1	3

for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Analysis

$\Theta(k)$ $\left\{\begin{array}{l}\textbf{for } i \leftarrow 1 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow 0\end{array}\right.$

$\Theta(n)$ $\left\{\begin{array}{l}\textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1\end{array}\right.$

$\Theta(k)$ $\left\{\begin{array}{l}\textbf{for } i \leftarrow 2 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow C[i] + C[i-1]\end{array}\right.$

$\Theta(n)$ $\left\{\begin{array}{l}\textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \quad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \qquad C[A[j]] \leftarrow C[A[j]] - 1\end{array}\right.$

$\Theta(n + k)$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

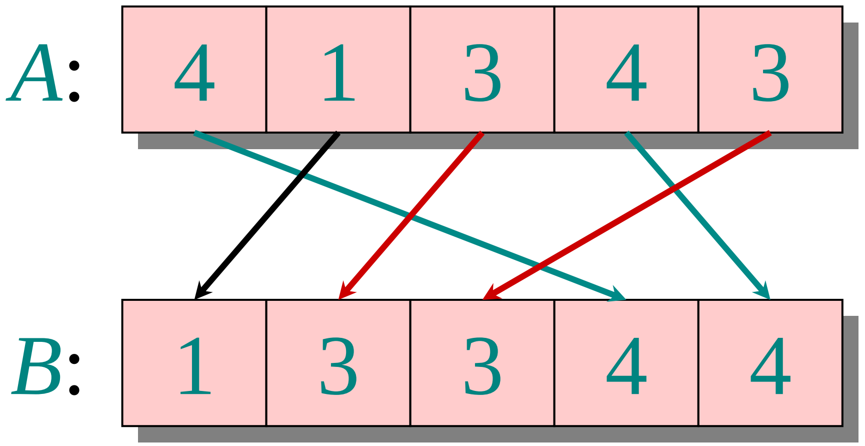
- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.



Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix.)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

Operation of radix sort

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Correctness of radix sort

Induction on digit position

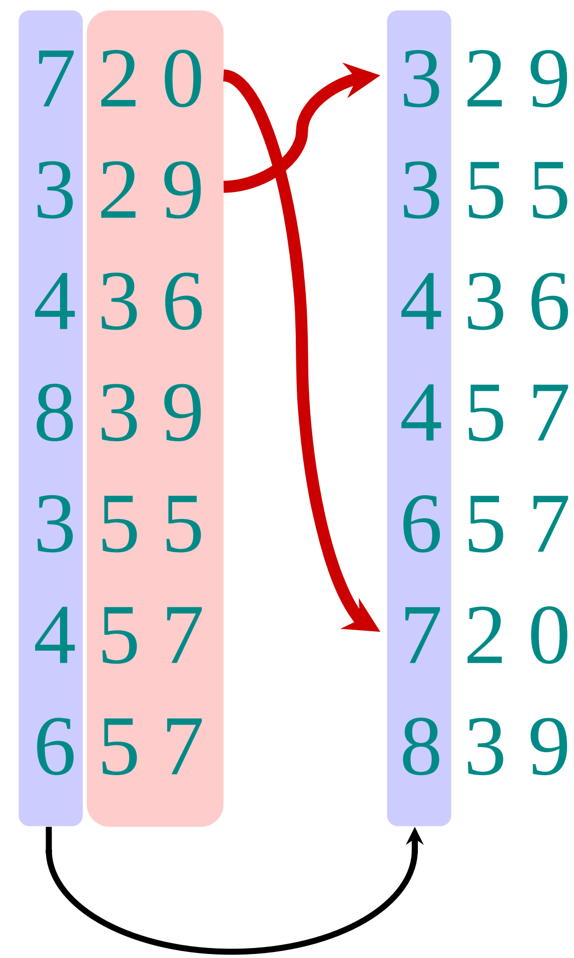
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t

7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

Correctness of radix sort

Induction on digit position

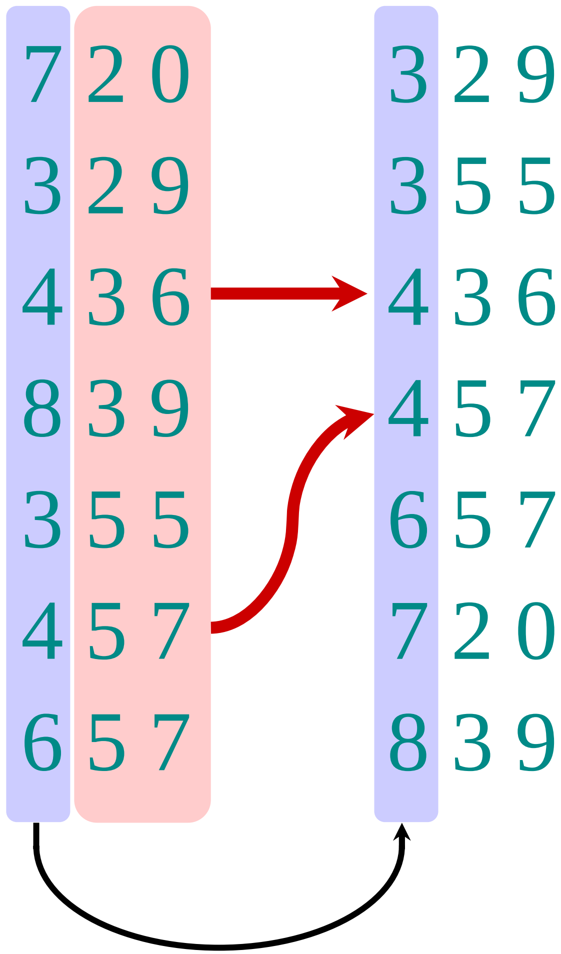
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of radix sort

Induction on digit position

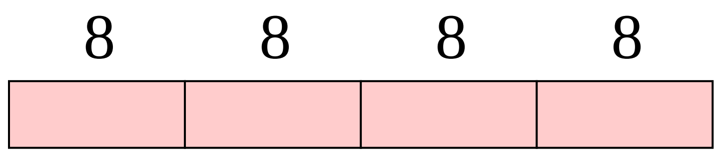
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.