Design and Analysis of Algorithm (KCS-503)

By

Raj Kumar

(Assistant Professor)

Department of Computer Science

KIET Group of Institutions, Delhi-NCR, Ghaziabad

- What is an Algorithm?

- How and why do we analyze algorithm?

Algorithms

- ***Informally***,
  - A tool for solving a well-specified computational problem.

- **Example: sorting**

  input: A sequence of numbers.

  output: An ordered permutation of the input.

  issues: correctness, efficiency, storage, etc.

Informal Definition

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness.
  - Definiteness.
  - Input.
  - Output.
  - Effectiveness.

Algorithm Analysis

- Determining performance characteristics. (Predicting the resource requirements.)
  - Time, memory, communication bandwidth etc.
  - **Computation time** (running time) is of primary concern.
- Why analyze algorithms?
  - **Choose** the **most efficient** of several possible algorithms for the same problem.
  - Is the best possible **running time** for a problem ***reasonably finite*** for practical purposes?
  - Is the algorithm **optimal** (best in some sense)? – Is something better possible?

Running Time

- Run time expression should be machine-independent.
  - Use a model of computation or “hypothetical” computer.
  - Our choice – **RAM model** (most commonly-used).
- Model should be
  - Simple.
  - Applicable.

RAM Model

- Generic single-processor model.
- **Supports simple constant-time instructions** found in real computers.
  - Arithmetic (+, –, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy).
  - Control (branch, subroutine call).
- Run time (**cost**) is uniform (**1 time unit**) for all simple instructions.
- Memory is unlimited.
- Flat memory model – no hierarchy.
- Access to a word of memory takes **1 time unit**.
- Sequential execution – **no concurrent operations**.

Running Time – Definition

- Running time of an algorithm for a given input is
  - The **number of steps** executed by the algorithm on that **input**.
- Often referred to as the ***complexity*** of the algorithm.

Complexity and Input

- Complexity of an algorithm generally depends on
  - **Size of input**.
    - Input size depends on the problem.
      - Examples: No. of items to be sorted.
      - No. of vertices and edges in a graph.
  - **Other characteristics of the input data**.
    - Are the items already sorted?
    - Are there cycles in the graph?

Worst, Average, and Best-case Complexity

- Worst-case Complexity
  - **Maximum** steps the algorithm takes for any possible input.
  - Most tractable measure.
- Average-case Complexity
  - **Average** of the running times of all ***possible*** **inputs.**
- Best-case Complexity
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?

Analysis: Examples

Insertion Sort

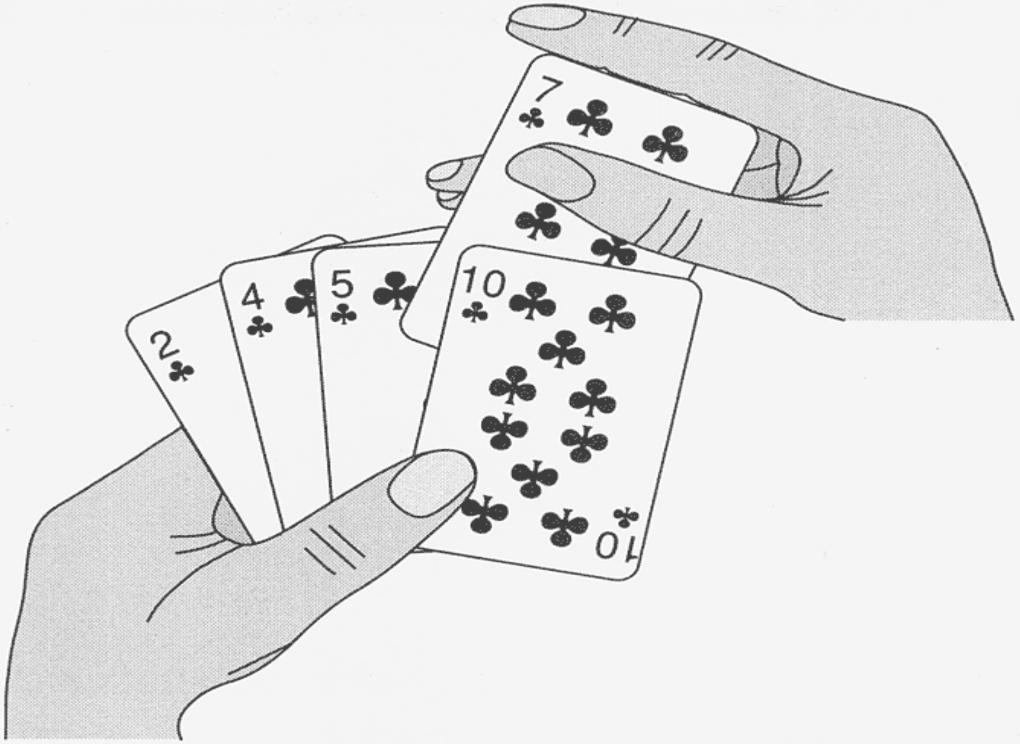
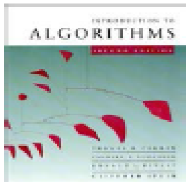


Figure 2.1 Sorting a hand of cards using insertion sort.

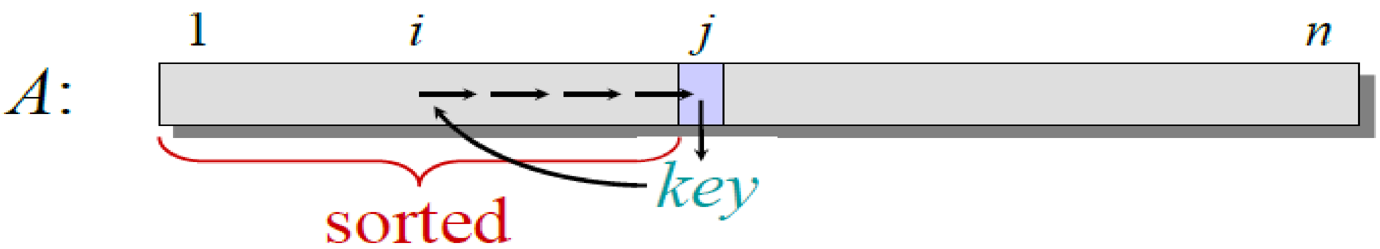
Insertion Sort (contd..)



Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    for j ← 2 to n
         do key ← A[j]
            i ← j – 1
            while i > 0 and A[i] > key
                 do A[i+1] ← A[i]
                    i ← i – 1
            A[i+1] = key
```



Insertion Sort (contd..)

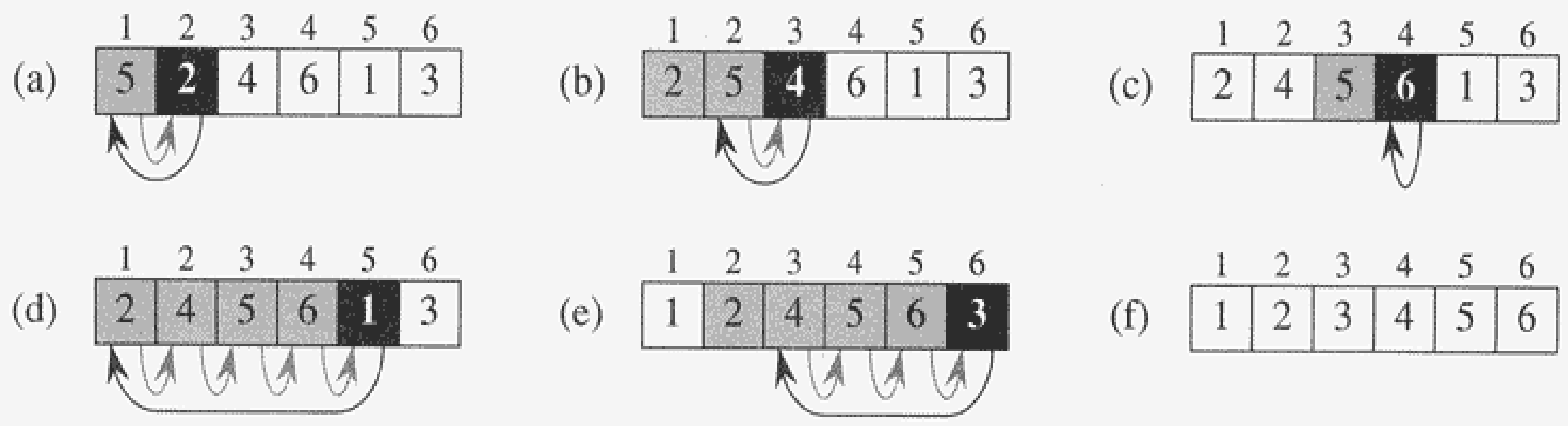


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

Insertion Sort (contd..)

- Algorithm:

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
         A[i + 1] ← key
```

Insertion Sort: Analysis

	INSERTION-SORT(A)	cost	times
1	**for** $j \leftarrow 2$ **to** *length*[A]	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

- t_j is the number of times the while loop test in line 5 is executed
- for that value of j.
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j-1)$
- $+ c_7 \sum_{j=2..n} (t_j-1) + c_8(n-1)$

Insertion Sort: Analysis

- Performance of Insertion sort depends on the value of t_j
- **Ques:** What are the *best* and *worst*-case running times of

  INSERTION-SORT?
- How about *average*-case?

Insertion Sort Analysis: Best case

- It occurs when Array is sorted.
- All tj values are 1.

$$T(n)=C1n+C2(n-1)+0\,(n-1)+C4(n-1)+C5\ \ +C6(\ \ )+C7(\ \ )+C8(n-1)$$

$$=C1n+C2\,(n-1)+0\,(n-1)+C4\,(n-1)+C5\ \ +C8\,(n-1)$$

$$=(C1+C2+C4+C5+C8)\,n-(C2+C4+C5+C8)$$

- Which is of the form an+b.
- Linear function of n. So, linear growth.

Insertion Sort Analysis: Worst Case

- It occurs when Array is reverse sorted, and $t_j = j$

$$T(n)=C_1n+C_2(n-1)+0\ (n-1)+C_4(n-1)+C_5\left(\sum_{j=2}^{n-1} j\right)+C_6\left(\sum_{j=2}^{n} j-1\right)+C_7\left(\sum_{j=2}^{n} j-1\right)+C_8(n-1)$$

$$=C_1n+C_2(n-1)+C_4(n-1)+C_5\left(\frac{n(n-1)}{2}-1\right)+C_6\left(\sum_{j=2}^{n}\frac{n(n-1)}{2}\right)+C_7\left(\sum_{j=2}^{n}\frac{n(n-1)}{2}\right)+C_8(n-1)$$

- which is of the form an^2+bn+c
- Quadratic function.
- So in worst case insertion set grows in n^2.

Insertion Sort Analysis: Average Case

- There may be a mix of best and worst cases
- Roughly as bad as worst case

A Simple Example – *Linear Search*

INPUT: **a sequence of *n* numbers, *key* to search for.**

OUTPUT: ***true* if *key* occurs in the sequence, *false* otherwise.**

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	c_1	1
while $i \leq n$ **and** A[i] != *key*	c_2	x
do i++	c_3	x-1
if $i \leq n$	c_4	1
then return *true*	c_5	1
else return *false*	c_6	1

x ranges between 1 and n+1.

So, the running time ranges between

$c_1 + c_2 + c_4 + c_5$ – **best case**

and

$c_1 + c_2(n+1) + c_3 n + c_4 + c_6$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	1	1
while $i \le n$ **and** A[i] != *key*	1	x
do i++	1	x-1
if $i \le n$	1	1
then return *true*	1	1
else return *false*	1	1

Assign a cost of 1 to all statement executions.

Now, the running time ranges between

$1+ 1+ 1 + 1 = 4$ – **best case**

and

$1+ (n+1)+ n + 1 + 1 = 2n+4$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \leq n$ **and** A[i] != *key*	1	x
3 **do** i++	1	x-1
4 **if** $i \leq n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

If we assume that we search for a random item in the list, on an average, Statements 2 and 3 will be executed $n/2$ times. Running times of other statements are independent of input. Hence, **average-case complexity** is

$$1 + n/2 + n/2 + 1 + 1 = n+3$$

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - **Ex:** $O(n)$, $\Theta(1)$, $\Omega(n^2)$, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - **Linear Search**: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**
- More on O, Θ, and Ω in next class. Use Θ for the present.

Comparison of Algorithms

- Complexity function can be used to compare the performance of algorithms.
- Algorithm *A* is more efficient than Algorithm *B* for solving a problem, if the complexity function of *A* is of lower order than that of *B*.
- Examples:
  - **Linear Search** – $\Theta(n)$ vs. **Binary Search** – $\Theta(\lg n)$
  - **Insertion Sort** – $\Theta(n^2)$ vs. **Quick Sort** – $\Theta(n \lg n)$

Comparisons of Algorithms

- **Multiplication**
  - classical technique: $O(nm)$
  - divide-and-conquer: $O(nm^{\ln 1.5}) \sim O(nm^{0.59})$

  For operands of size 1000, takes 40 & 15 seconds respectively on a Cyber 835.

- **Sorting**
  - insertion sort: $\Theta(n^2)$
  - merge sort: $\Theta(n \lg n)$

  For 10^6 numbers, it took 5.56 hrs on a supercomputer using machine language and 16.67 min on a PC using C/C++.

Why Order of Growth Matters?

- Computer speeds double every two years, so why worry about algorithm speed?
- When speed doubles, what happens to the amount of work you can do?
- What about the demands of applications?

Effect of Faster Machines

No. of items sorted

H/W Speed / *Comp. of Alg.*	1 M*	2 M	Gain
$O(n^2)$	1000	1414	1.414
$O(n \lg n)$	62700	118600	1.9

* Million operations per second.

- Higher gain with faster hardware for more efficient algorithm.
- Results are more dramatic for more higher speeds.

Asymptotic Notations

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - **Ex:** $O(n)$, $\Theta(1)$, $\Omega(n^2)$, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - **Linear Search**: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**

Asymptotic Complexity

- Running time of an algorithm as a function of input size n **for large *n***.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$.
- Describes behavior of function in the limit using ***Asymptotic Notation***.

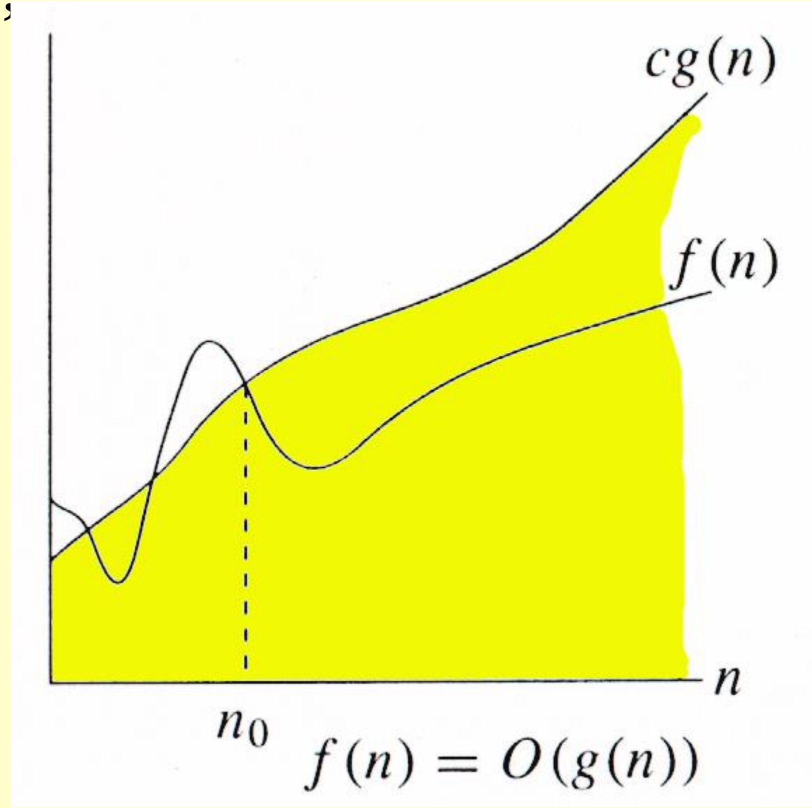
Asymptotic Notation

- **Θ, *O*, Ω, *o*, ω**
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n) :$
$\exists$ **positive constants** c **and** $n_0,$
such that $\forall n \geq n_0,$
we have $0 \leq f(n) \leq cg(n) \}$



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

g*(*n*) is an** ***asymptotic upper bound **for** ***f*(*n*).**

$f(n) = O(g(n))$

Examples

$O(g(n)) = \{f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

- Any linear *function* $an + b$ is in $O(n^2)$. **How?**
- Show that $3n^2+2\text{n}=O(n^2)$ for appropriate c and n_0.

Examples

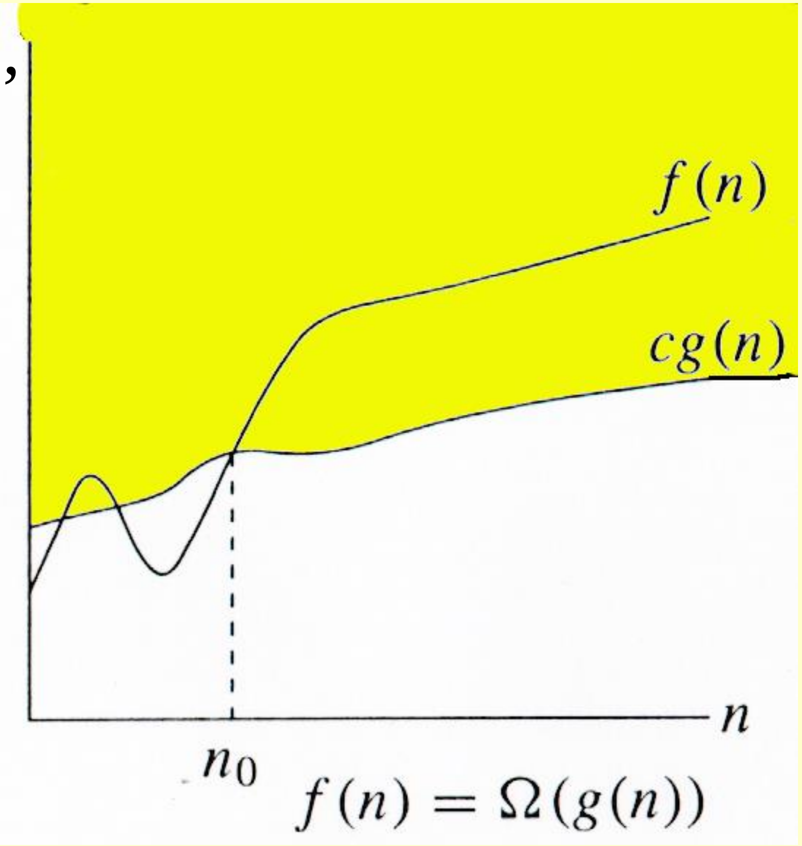
Examples

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n) :$
$\exists$ **positive constants** c **and** n_0,
such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Omega(g(n))$.

Example

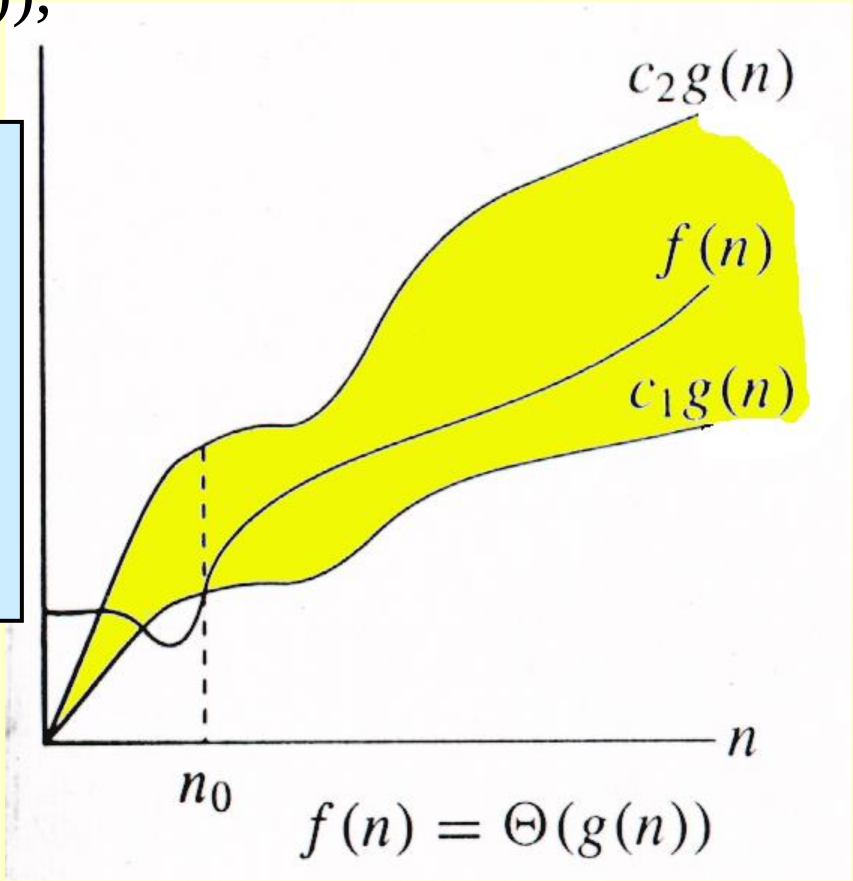
$\Omega(g(n)) = \{f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- $\sqrt{n} = \Omega(\lg n)$. Choose c and n_0.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:
$\exists$ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

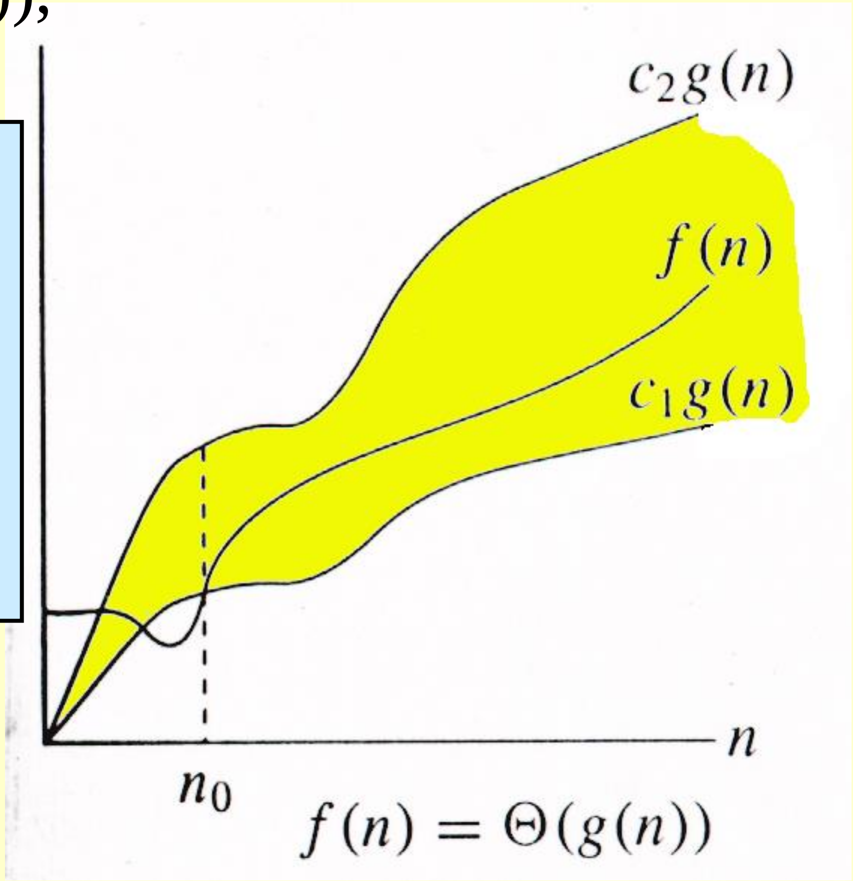
***g*(*n*) is an *asymptotically tight bound* for *f*(*n*).**

Examples

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n) :$
$\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Technically, $f(n) \in \Theta(g(n))$.
Older usage, $f(n) = \Theta(g(n))$.

$f(n)$ and $g(n)$ are nonnegative, for large n.

Example

$\Theta(g(n)) = \{f(n) :$ $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $\quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- $10n^2 - 3n = \Theta(n^2)$
- What constants for n_0, c_1, and c_2 will work?

Example

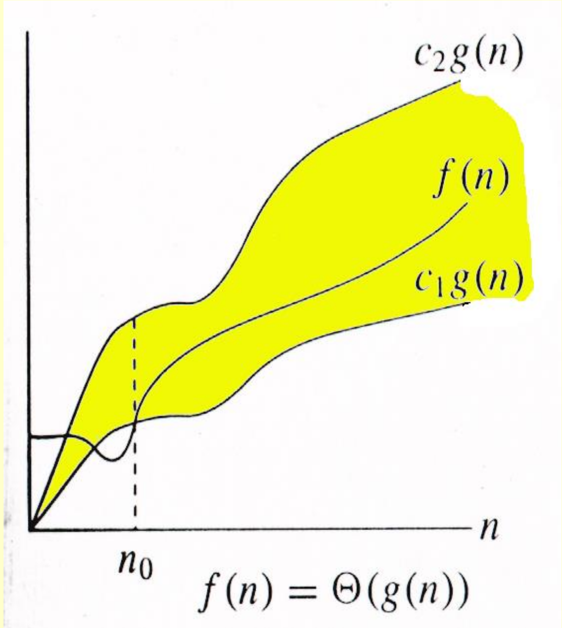
$\Theta(g(n)) = \{f(n) :$ $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $\quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

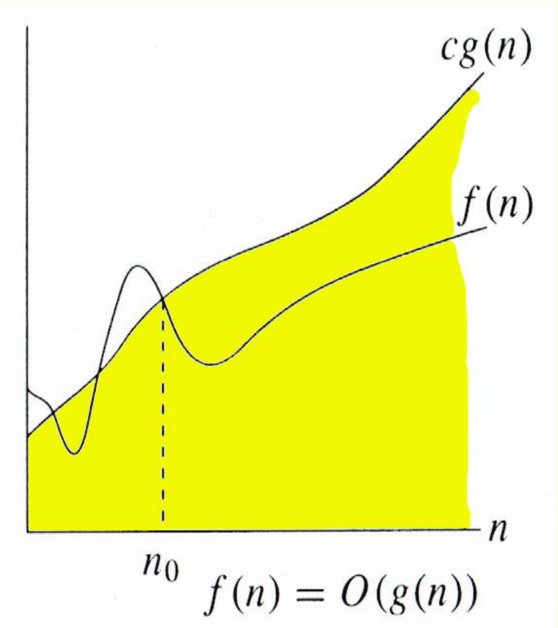
- Is $3n^3 \in \Theta(n^4)$??
- How about $2^{2n} \in \Theta(2^n)$??

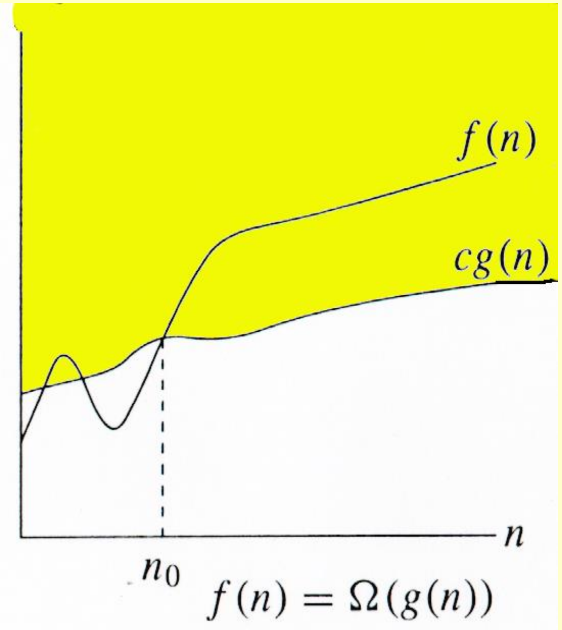
Example

- Exercise: Prove that $n^2/2\text{-}3n = \Theta(n^2)$

Relations Between Θ, *O*, Ω







Relations Between Θ, Ω, O

Theorem : For any two functions $g(n)$ and $f(n)$,

$\boldsymbol{f(n) = \Theta(g(n))}$ iff

$\boldsymbol{f(n) = O(g(n))}$ **and** $\boldsymbol{f(n) = \Omega(g(n))}$.

- I.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Asymptotic Notation in Equations

- Can use asymptotic notation in equations to replace expressions containing lower-order terms.
- For example,

  $4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$

  $= 4n^3 + \Theta(n^2) = \Theta(n^3)$. **How to interpret?**
- In equations, $\Theta(f(n))$ always stands for an ***anonymous function*** $g(n) \in \Theta(f(n))$
  - In the example above, $\Theta(n^2)$ stands for $3n^2 + 2n + 1$.

(small)o-notation

For a given function $g(n)$, the set little-o:

$o(g(n))$ = $\{f(n)$: $\forall\, c > 0$, $\exists\, n_0 > 0$ such that $\forall\, n \geq n_0$, we have $0 \leq f(n) < cg(n)\}$.

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n\to\infty} [f(n) / g(n)] = 0$$

$g(n)$ is an ***upper bound*** for $f(n)$ that is not asymptotically tight.

Observe the difference in this definition from previous ones. **Why?**

ω -notation

For a given function $g(n)$, the set little-omega:

$$\omega(g(n)) = \{f(n): \forall\, c > 0, \exists\, n_0 > 0 \text{ such that } \forall\, n \geq n_0, \text{ we have } 0 \leq cg(n) < f(n)\}.$$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:

$$\lim_{n\to\infty} [f(n) / g(n)] = \infty.$$

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Comparison of Functions

$$f \leftrightarrow g \;\approx\; a \leftrightarrow b$$

$$f(n) = O(g(n)) \;\approx\; a \le b$$

$$f(n) = \Omega(g(n)) \;\approx\; a \ge b$$

$$f(n) = \Theta(g(n)) \;\approx\; a = b$$

$$f(n) = o(g(n)) \;\approx\; a < b$$

$$f(n) = \omega(g(n)) \;\approx\; a > b$$

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Properties

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o(g(n)) \;\&\; g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties

- **Symmetry**

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- **Complementarity**

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ iff } g(n) = \omega((f(n))$$

Common Functions

Monotonicity

- $f(n)$ is
  - **monotonically increasing** if $m \leq n \Rightarrow f(m) \leq f(n)$.
  - **monotonically decreasing** if $m \geq n \Rightarrow f(m) \geq f(n)$.
  - **strictly increasing** if $m < n \Rightarrow f(m) < f(n)$.
  - **strictly decreasing** if $m > n \Rightarrow f(m) > f(n)$.

Exponentials

- **Useful Identities:**

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

- **Exponentials and polynomials**

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Logarithms

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

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

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

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

Logarithms and exponentials – Bases

- If the base of a logarithm is changed from one constant to another, the value is altered by a constant factor.
  - **Ex:** $\log_{10} n * \mathbf{\log_2 10} = \log_2 n$.
  - Base of logarithm is not an issue in asymptotic notation.
- Exponentials with different bases differ by a exponential factor (not a constant factor).
  - **Ex:** $2^n = \mathbf{(2/3)^n} * 3^n$.

Polylogarithms

- **For $a \geq 0$, $b > 0$,** $\lim_{n\to\infty} (\lg^a n / n^b) = 0$, so $\lg^a n = o(n^b)$, and $n^b = \omega(\lg^a n)$
  - Prove using L'Hopital's rule repeatedly

- $\lg(n!) = \Theta(n \lg n)$
  - Prove using Stirling's approximation (in the text) for $\lg(n!)$.

Exercise

Express functions in A in asymptotic notation using functions in B.

A	B	
$5n^2 + 100n$	$3n^2 + 2$	$A \in \Theta(B)$

$A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

$\log_3(n^2)$	$\log_2(n^3)$	$A \in \Theta(B)$

$\log_b a = \log_c a / \log_c b$; $A = 2\lg n / \lg 3$, $B = 3\lg n$, $A/B = 2/(3\lg 3)$

$n^{\lg 4}$	$3^{\lg n}$	$A \in \omega(B)$

$a^{\log b} = b^{\log a}$; $B = 3^{\lg n} = n^{\lg 3}$; $A/B = n^{\lg(4/3)} \to \infty$ as $n \to \infty$

$\lg^2 n$	$n^{1/2}$	$A \in o(B)$

$\lim_{n\to\infty} (\lg^a n / n^b) = 0$ (here $a = 2$ and $b = 1/2$) $\Rightarrow A \in o(B)$

Recurrence Relation

Recurrence Relation

- Equation or an inequality that characterizes/defines a function by its values on smaller inputs.

Ex: Recurrence with exact function

$$T(n) = 1 \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + n \quad \text{if } n > 1$$

- **Solution Methods**
  - » Master Method.
  - » Recursion-tree Method.
  - » Iteration Method
  - » Substitution Method.

- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.
  - » **Ex:** Divide and Conquer.

$$T(n) = \Theta(1) \quad \text{if } n \leq c$$
$$T(n) = a\,T(n/b) + D(n) + C(n) \quad \text{otherwise}$$

Some Technicalities

- We can (almost always) ignore floors and ceilings.
- Exact vs. Asymptotic functions.
  - » In algorithm analysis, both the recurrence and its solution are expressed using asymptotic notation.
  - » Ex: Recurrence with exact function

$$T(n) = 1 \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + n \qquad \text{if } n > 1$$

Solution: $T(n) = n \lg n + n$

Recurrence with asymptotic (BEWARE!)

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

Solution: $T(n) = \Theta(n \lg n)$

The Master Method

- Based on the Master theorem.
- "Cookbook" approach for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

The Master Theorem

Theorem 4.1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and

Let $T(n)$ be defined on nonnegative integers by the recurrence

$T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

$T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,

   and if, for some constant $c < 1$ and all sufficiently large n,

   we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

Examples

Changing Variables

- Use algebraic manipulation to turn an unknown recurrence into one similar to what you have seen before.
  - » Example: $T(n) = 2T(n^{1/2}) + \lg n$
  - » Rename $\boldsymbol{m = \lg n}$ and we have

    $T(2^m) = 2T(2^{m/2}) + m$
  - » Set $S(m) = T(2^m)$ and we have

    $S(m) = 2S(m/2) + m \Rightarrow S(m) = O(m \lg m)$
  - » Changing back from $S(m)$ to $T(n)$, we have

    $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$

2. Recursion Tree Method

- Recursion Trees
  - Show successive expansions of recurrences using trees.
  - A recursion tree is a tree where each node represents the cost of a certain recursive subproblem.
  - Sum up the numbers(cost) in each node to get the cost of the entire algorithm
  - Keep track of the time spent on the subproblems of a divide and conquer algorithm.

- Running time of Merge Sort:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

- Rewrite the recurrence as

$$T(n) = c \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + cn \quad \text{if } n > 1$$

$c > 0$: Running time for the base case and time per array element for the divide and combine steps.

Recursion Tree – Example

- Running time of Merge Sort:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$

$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

- Rewrite the recurrence as

$$T(n) = \boldsymbol{c} \qquad \text{if } n = 1$$

$$T(n) = 2T(n/2) + \boldsymbol{cn} \qquad \text{if } n > 1$$

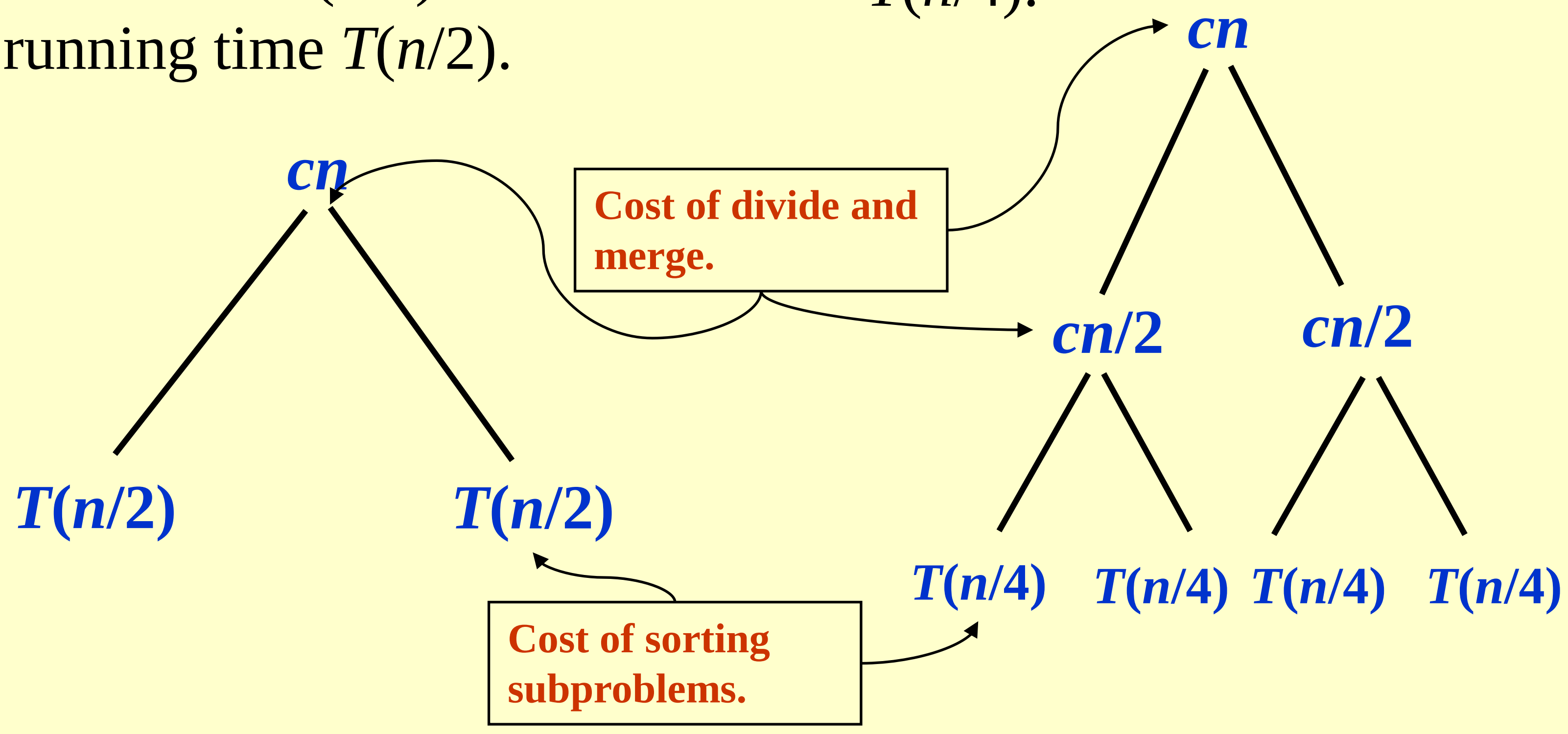
$c > 0$: Running time for the base case and time per array element for the divide and combine steps.

Example

Recursion Tree for Merge Sort

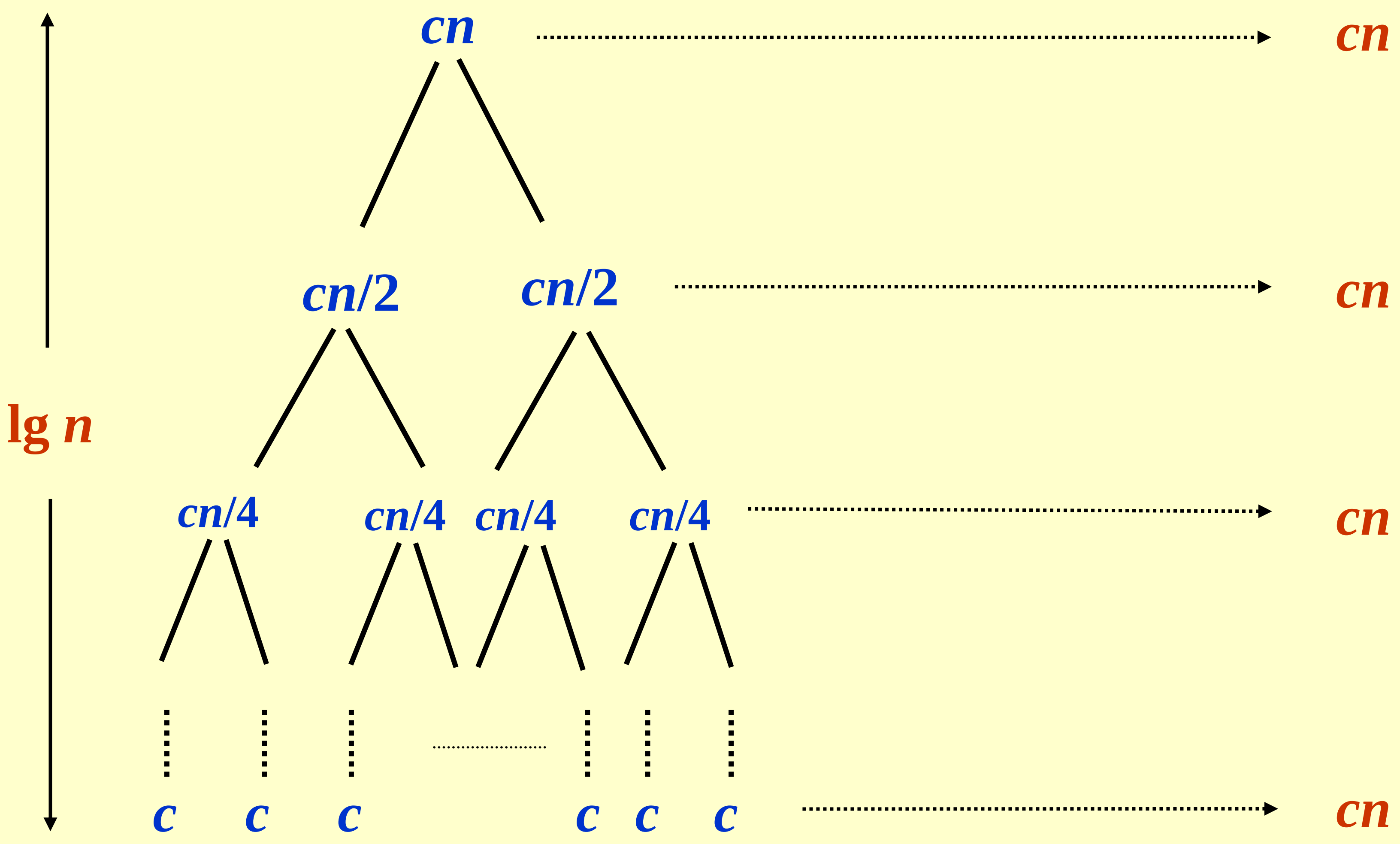
For the original problem, we have a cost of cn, plus two subproblems each of size (n/2) and running time $T(n/2)$.

Each of the size n/2 problems has a cost of cn/2 plus two subproblems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

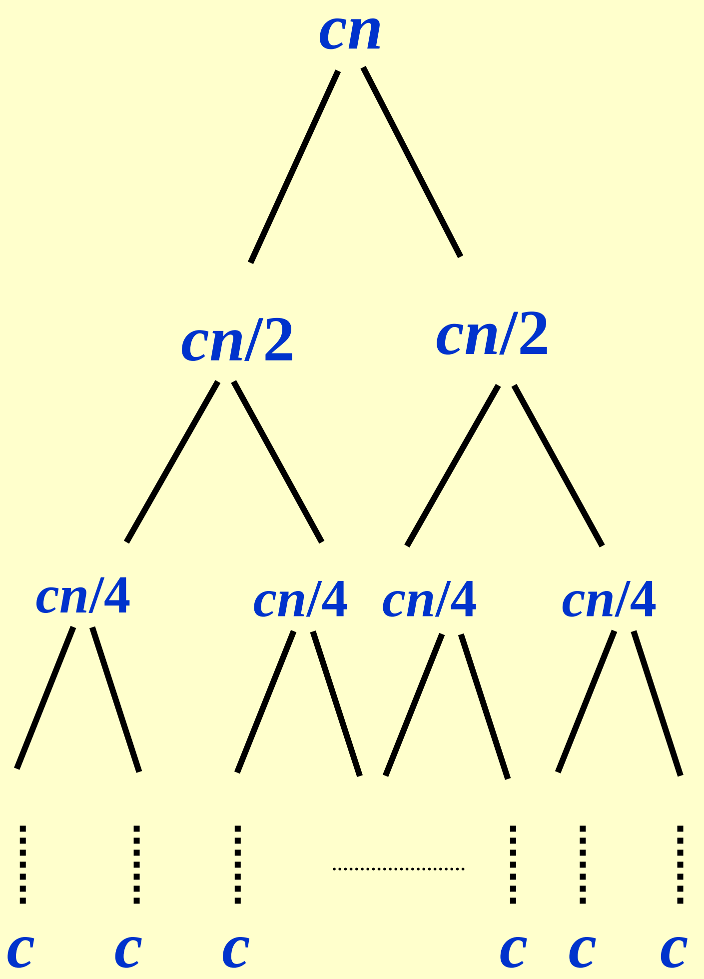
Continue expanding until the problem size reduces to 1.



Total : $cn\lg n + cn$

Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves $\Rightarrow$ *cost per level remains the same*.
- There are lg n + 1 levels, height is lg n. (Assuming n is a power of 2.)
  - Can be proved by induction.
- Total cost = sum of costs at each level = $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$.

Other Examples

- Use the recursion-tree method to determine a guess for the recurrences
  - » $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.
  - » $T(n) = T(n/3) + T(2n/3) + O(n)$.

Example

Example

Substitution Method

- **Guess** the form of the solution, then **use mathematical induction** to show it correct.
  - » Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values – hence, the name.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

Example – Exact Function

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:** $T(n) = 2\,T(n/2) + n$

$$= 2\,((n/2)\lg(n/2) + (n/2)) + n$$

$$= n\,(\lg(n/2)) + 2n$$

$$= n \lg n - n + 2n$$

$$= n \lg n + n$$

Example – With Asymptotics

To Solve: $T(n) = 3T(\lfloor n/3 \rfloor) + n$

- Guess: $T(n) = O(n \lg n)$
- Need to prove: $T(n) \le cn \lg n$, for some $c > 0$.
- Hypothesis: $T(k) \le ck \lg k$, for all $k < n$.
- Calculate:

$$
\begin{aligned}
T(n) &\le 3c \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + n \\
&\le c\, n \lg (n/3) + n \\
&= c\, n \lg n - c\, n \lg 3 + n \\
&= c\, n \lg n - n\,(c \lg 3 - 1) \\
&\le c\, n \lg n
\end{aligned}
$$

(The last step is true for $c \ge 1 / \lg 3$.)

Example – With Asymptotics

To Solve: $T(n) = 3T(\lfloor n/3 \rfloor) + n$

- To show $T(n) = \Theta(n \lg n)$, must show both upper and lower bounds, i.e., $T(n) = O(n \lg n)$ **AND** $T(n) = \Omega(n \lg n)$
- (Can you find the mistake in this derivation?)
- Show: $T(n) = \Omega(n \lg n)$
- Calculate:

$$\begin{aligned} T(n) &\geq 3c \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + n \\ &\geq c\, n \lg (n/3) + n \\ &= c\, n \lg n - c\, n \lg 3 + n \\ &= c\, n \lg n - n\,(c \lg 3 - 1) \\ &\geq c\, n \lg n \end{aligned}$$

(The last step is true for $c \leq 1 / \lg 3$.)

Example – With Asymptotics

If $T(n) = 3T(\lfloor n/3 \rfloor) + O(n)$, as opposed to $T(n) = 3T(\lfloor n/3 \rfloor) + n$, then rewrite $T(n) \le 3T(\lfloor n/3 \rfloor) + cn$, $c > 0$.

- To show $T(n) = O(n \lg n)$, use second constant $\boldsymbol{d}$, different from $\boldsymbol{c}$.
- Calculate:

$$
\begin{aligned}
T(n) &\le 3d \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + c\, n \\
&\le d\, n \lg (n/3) + cn \\
&= d\, n \lg n - d\, n \lg 3 + cn \\
&= d\, n \lg n - n\, (d \lg 3 - \mathrm{c}) \\
&\le d\, n \lg n
\end{aligned}
$$

(The last step is true for $d \ge c / \lg 3$.)

It is <u>OK</u> for d to depend on c.

Making a Good Guess

- If a recurrence is similar to one seen before, then guess a similar solution.
  - » $T(n) = 3T(\lfloor n/3 \rfloor + 5) + n$ (Similar to $T(n) = 3T(\lfloor n/3 \rfloor) + n$)
    - When n is large, the difference between $n/3$ and $(n/3 + 5)$ is insignificant.
    - Hence, can guess $O(n \lg n)$.
- Method 2: Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.
  - » E.g., start with $T(n) = \Omega(n)$ & $T(n) = O(n^2)$.
  - » Then lower the upper bound and raise the lower bound.

Subtleties

- When the math doesn't quite work out in the induction, strengthen the guess by subtracting a lower-order term. Example:
  - » Initial guess: $T(n) = O(n)$ for $T(n) = 3T(\lfloor n/3 \rfloor) + 4$
  - » Results in: $T(n) \leq 3c\lfloor n/3 \rfloor + 4 = c\,n + 4$
  - » Strengthen the guess to: $T(n) \leq c\,n - b$, where $b \geq 0$.
    - What does it mean to strengthen?
    - Though counterintuitive, it works. Why?

$T(n) \leq 3(c\lfloor n/3 \rfloor - b) + 4 \leq c\,n - 3b + 4 = c\,n - b - (2b - 4)$

Therefore, $T(n) \leq c\,n - b$, if $2b - 4 \geq 0$ or if $b \geq 2$.

(Don't forget to check the base case: here $c > b + 1$.)

Avoiding Pitfalls

- Be careful not to misuse asymptotic notation. For example:
  - » We can falsely prove $T(n) = O(n)$ by guessing $T(n) \le cn$ for $T(n) = 2T(\lfloor n/2 \rfloor) + n$

$$T(n) \le 2c\lfloor n/2 \rfloor + n$$
$$\le c\,n + n$$
$$= O(n) \quad \Leftarrow \text{Wrong!}$$

  - » We are supposed to prove that $T(n) \le c\,n$ for all $n>N$, according to the definition of $O(n)$.
- Remember: prove the *exact form* of inductive hypothesis.

Exercises

- Solution of $T(n) = T(\lceil n/2 \rceil) + n$ is $O(n)$
- Solution of $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$
- Solve $T(n) = 2T(n/2) + 1$
- Solve $T(n) = 2T(n^{1/2}) + 1$ by making a change of variables. Don't worry about whether values are integral.

Quicksort

Ack: Several slides from Prof. Jim Anderson's COMP 202 notes.

Performance

- A triumph of analysis by C.A.R. Hoare

- Worst-case execution time – $\Theta(n^2)$.
- Average-case execution time – $\Theta(n \lg n)$.
  - » How do the above compare with the complexities of other sorting algorithms?

- Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its competitors.

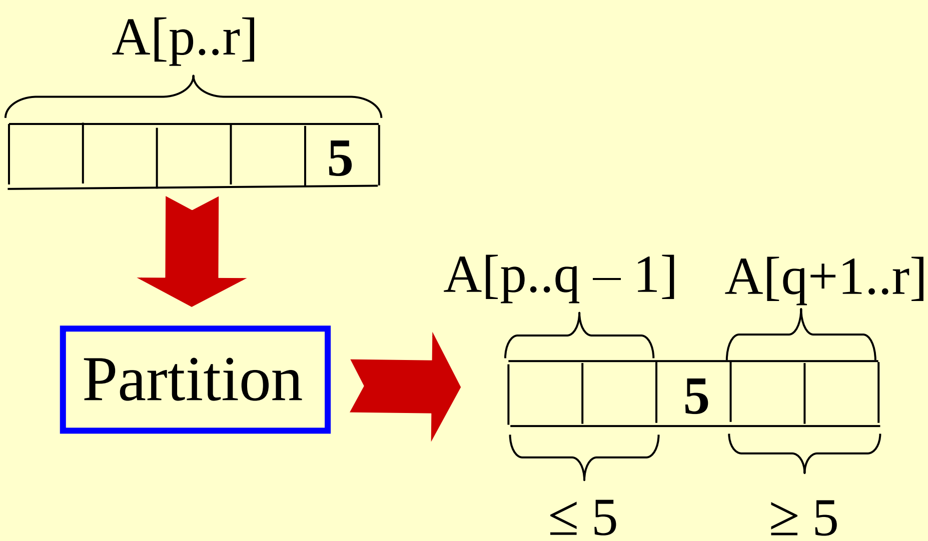
Quicksort

- Follows the **divide-and-conquer** paradigm.
- ***Divide*:** Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - » Each element in $A[p..q-1] \leq A[q]$.
  - » $A[q] \leq$ each element in $A[q+1..r]$.
  - » Index q is computed as part of the partitioning procedure.
- ***Conquer*:** Sort the two subarrays by recursive calls to quicksort.
- ***Combine*:** The subarrays are sorted in place – no work is needed to combine them.

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
PARTITION(A, p, r)
x = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Example

		p									r
initially:		2	5	8	3	9	4	1	7	10	**6**
	i	j									
next iteration:		2	5	8	3	9	4	1	7	10	**6**
		i	j								
next iteration:		2	5	8	3	9	4	1	7	10	**6**
			i	j							
next iteration:		2	5	8	3	9	4	1	7	10	**6**
			i		j						
next iteration:		2	5	3	8	9	4	1	7	10	**6**
				i		j					

note: pivot (x) = 6

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Example (Continued)

next iteration:

2	5	3	8	9	4	1	7	10	**6**
		i		j					

next iteration:

2	5	3	8	9	4	1	7	10	**6**
		i			j				

next iteration:

2	5	3	4	9	8	1	7	10	**6**
			i			j			

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i			j		

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i				j	

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i					j

after final swap:

2	5	3	4	1	**6**	9	7	10	8
				i					j

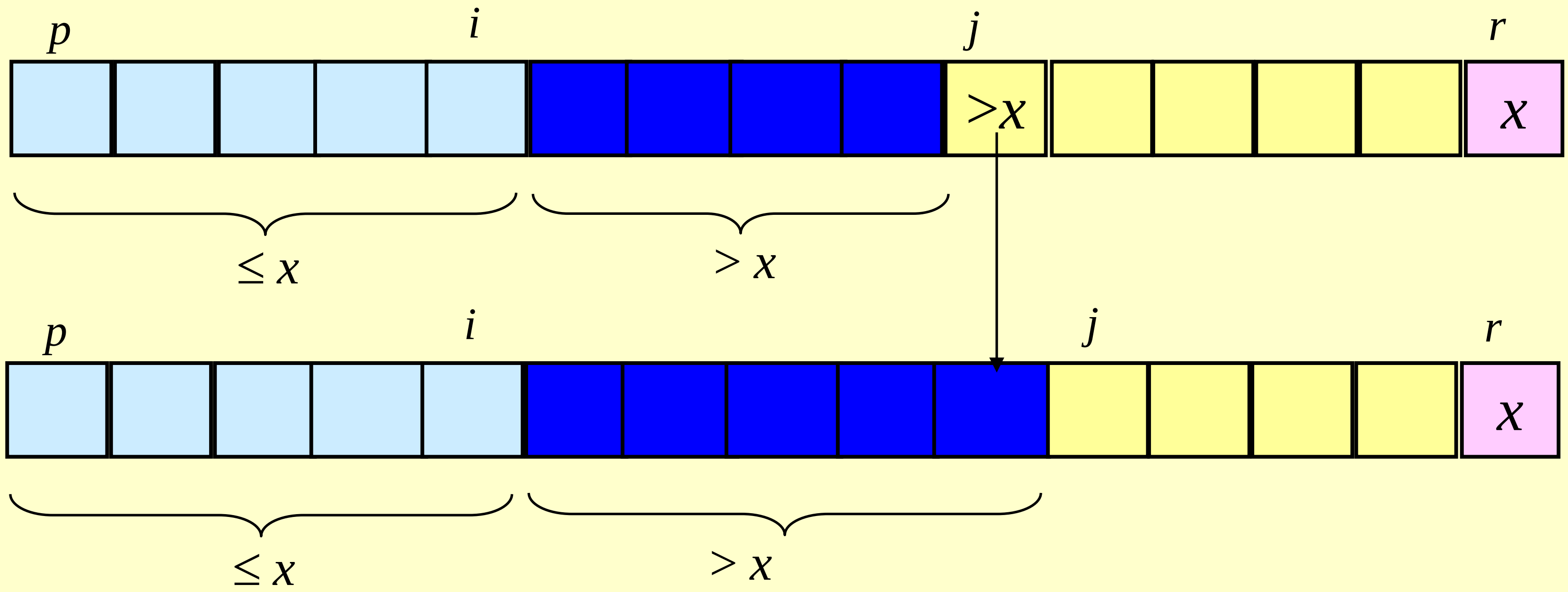
```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Partitioning

- Select the last element A[*r*] in the subarray *A*[*p*..*r*] as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. *A*[*p*..*i*] — All entries in this region are ≤ ***pivot***.
  2. *A*[*i*+1..*j* – 1] — All entries in this region are > ***pivot***.
  3. *A*[*r*] = *pivot*.
  4. *A*[*j*..*r* – 1] — Not known how they compare to *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI.)

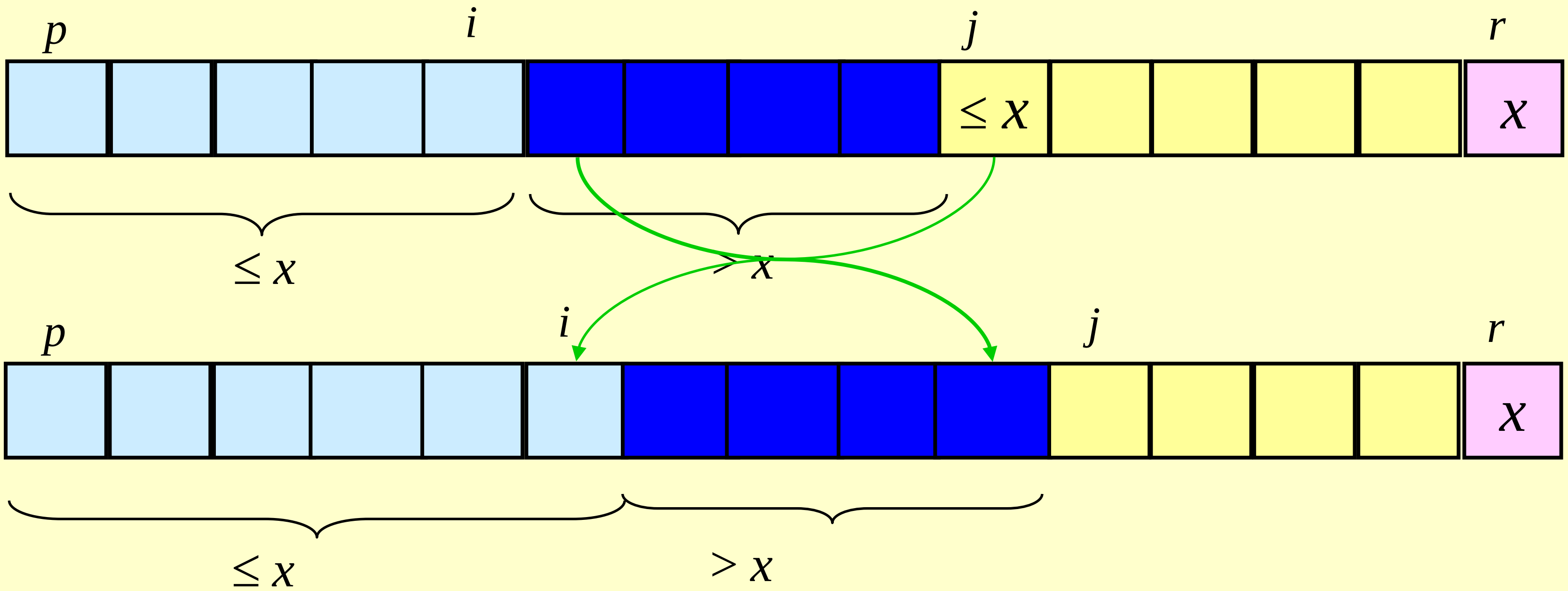
Correctness of Partition

Case 1:



Correctness of Partition

- **Case 2:** $A[j] \le x$
  - » Increment i
  - » Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - » Increment j
    - Condition 2 is maintained.
  - » $A[r]$ is unaltered.
    - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - » When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \le$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - » *Pivot* moves from the end of the array to between the two subarrays.
  - » Thus, procedure *partition* correctly performs the divide step.

Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Algorithm Performance

Running time of quicksort depends on whether the partitioning is balanced or not.

- Worst-Case Partitioning (Unbalanced Partitions):
  - » Occurs when every call to partition results in the most unbalanced partition.
  - » Partition is most unbalanced when
    - Subproblem 1 is of size $n - 1$, and subproblem 2 is of size 0 or vice versa.
    - $pivot \geq$ every element in $A[p..r - 1]$ or $pivot <$ every element in $A[p..r - 1]$.
  - » Every call to partition is most unbalanced when
    - Array $A[1..n]$ is sorted or reverse sorted!

Worst-case Partition Analysis

Recursion tree for
worst-case partition

$$n \left\{ \begin{array}{c} n \\ \downarrow \\ n-1 \\ \downarrow \\ n-2 \\ \downarrow \\ n-3 \\ \vdots \\ 2 \\ \downarrow \\ 1 \end{array} \right.$$

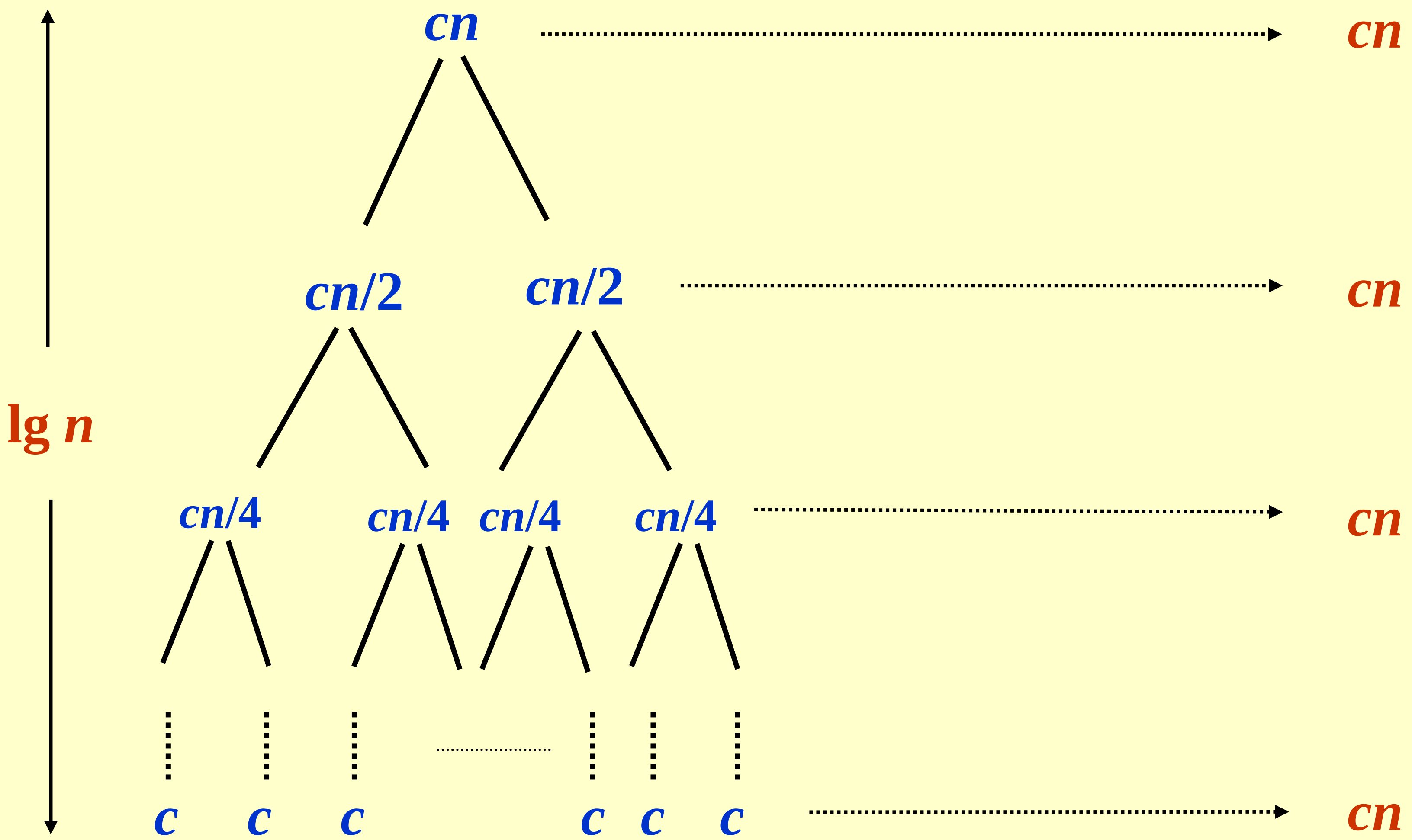
Running time for worst-case partitions at each recursive level:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\ &= T(n-1) + \Theta(n) \\ &= \textstyle\sum_{k=1 \text{ to } n} \Theta(k) \\ &= \Theta(\textstyle\sum_{k=1 \text{ to } n} k) \\ &= \Theta(n^2) \end{aligned}$$

Best-case Partitioning

- Size of each subproblem $\leq n/2$.
  - » One of the subproblems is of size $\lfloor n/2 \rfloor$
  - » The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - » $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$
    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \lg n)}$

Recursion Tree for Best-case Partition



Total : $O(n \lg n)$

Heapsort

Heapsort

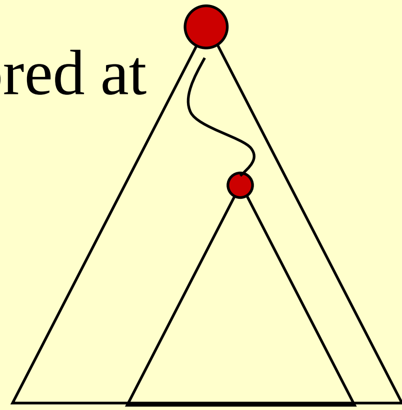
- Combines the better attributes of merge sort and insertion sort.
  - » Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - » Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - » Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - » Priority Queues

Data Structure :Binary Heap

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$
- length[A] – number of elements in array A.
- heap-size[A] – number of elements in heap stored in A.
  - heap-size[A] $\leq$ length[A]

Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



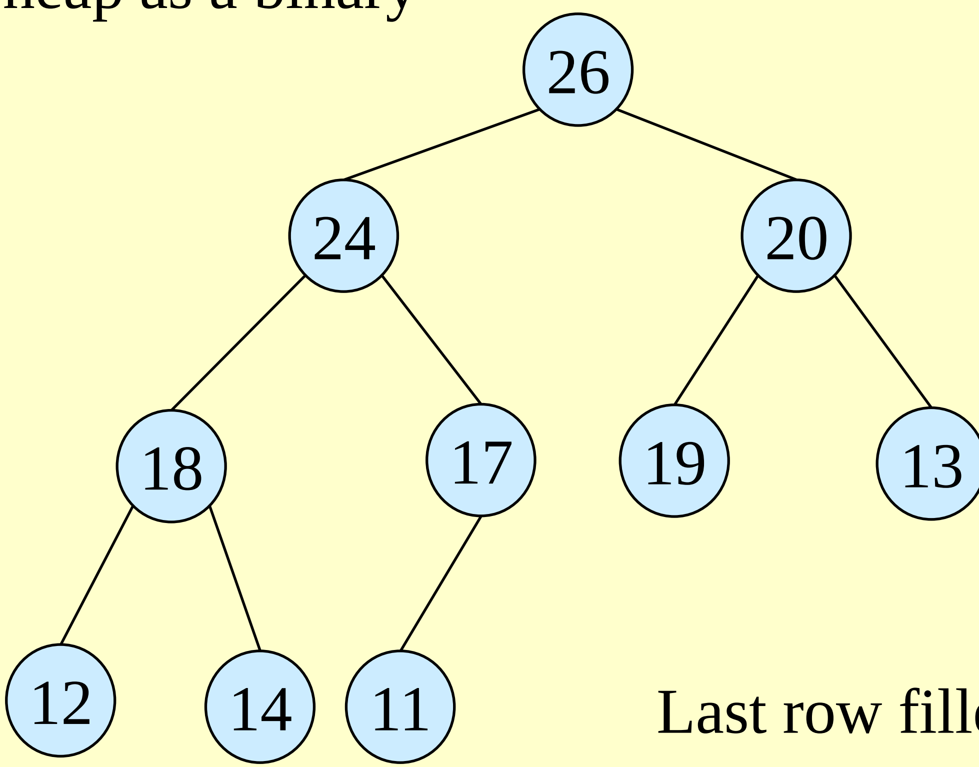
- Min-Heap
  - » For every node excluding the root, value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heaps – Example

1	2	3	4	5	6	7	8	9	10
26	24	20	18	17	19	13	12	14	11

Max-heap as an array.

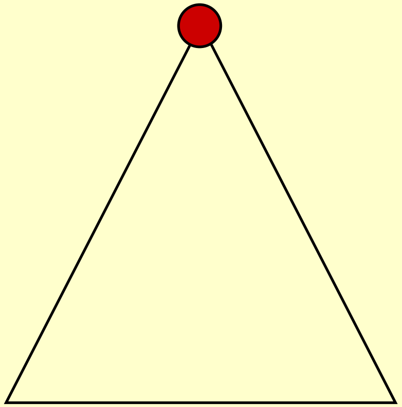
Max-heap as a binary tree.



Last row filled from left to right.

Height

- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- *Height of a tree*: the height of the root.
- Height of a heap: $\lfloor \lg n \rfloor$
  - » Basic operations on a heap run in $O(\lg n)$ time

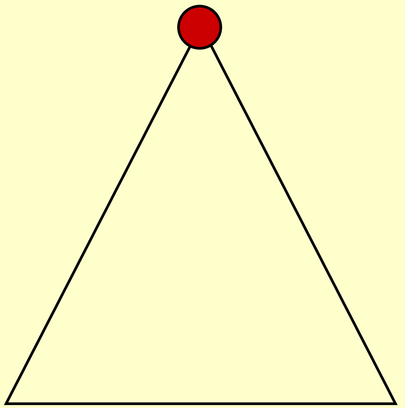


Heaps in Sorting

- Use max-heaps for sorting.
- The array representation of max-heap is not sorted.
- Steps in sorting
  - » Convert the given array of size n to a max-heap (*BuildMaxHeap*)
  - » Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves $n - 1$ elements to be placed in their appropriate locations.
    - However, the array of first $n - 1$ elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

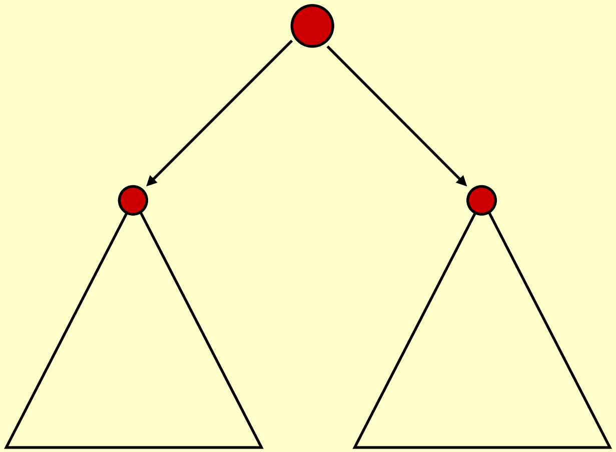
Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height h $\le \lceil n/2^{h+1} \rceil$



Maintaining the heap property

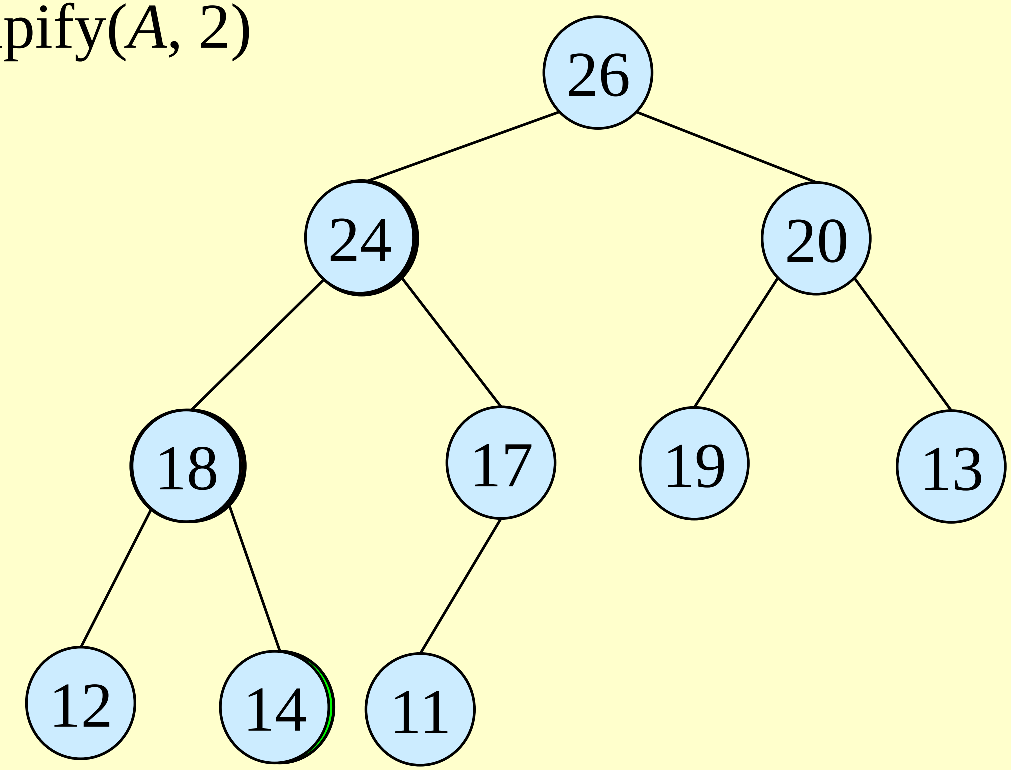
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - » May lead to the subtree at the child not being a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(A, 2)



Procedure MaxHeapify

MaxHeapify(A, i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] **and** $A[r] > A[largest]$
    then *largest* $\leftarrow r$
if $largest \neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
        MaxHeapify(A, *largest*)

Assumption:
Left(i) and Right(i) are max-heaps.

Running Time for MaxHeapify

<u>*MaxHeapify*(*A*, *i*)</u>

$l \leftarrow \text{left}(i)$
$r \leftarrow \text{right}(i)$
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
  then *largest* $\leftarrow l$
  else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] **and** $A[r] > A[largest]$
  then *largest* $\leftarrow r$
if *largest* $\neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MaxHeapify(*A*, *largest*)

Time to fix node *i* and its children = **Θ(1)**

PLUS

Time to fix the subtree rooted at one of *i*'s children = *T*(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

- $T(n) = T(largest) + \Theta(1)$
- $largest \le 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \le T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

<u>*BuildMaxHeap*(*A*)</u>

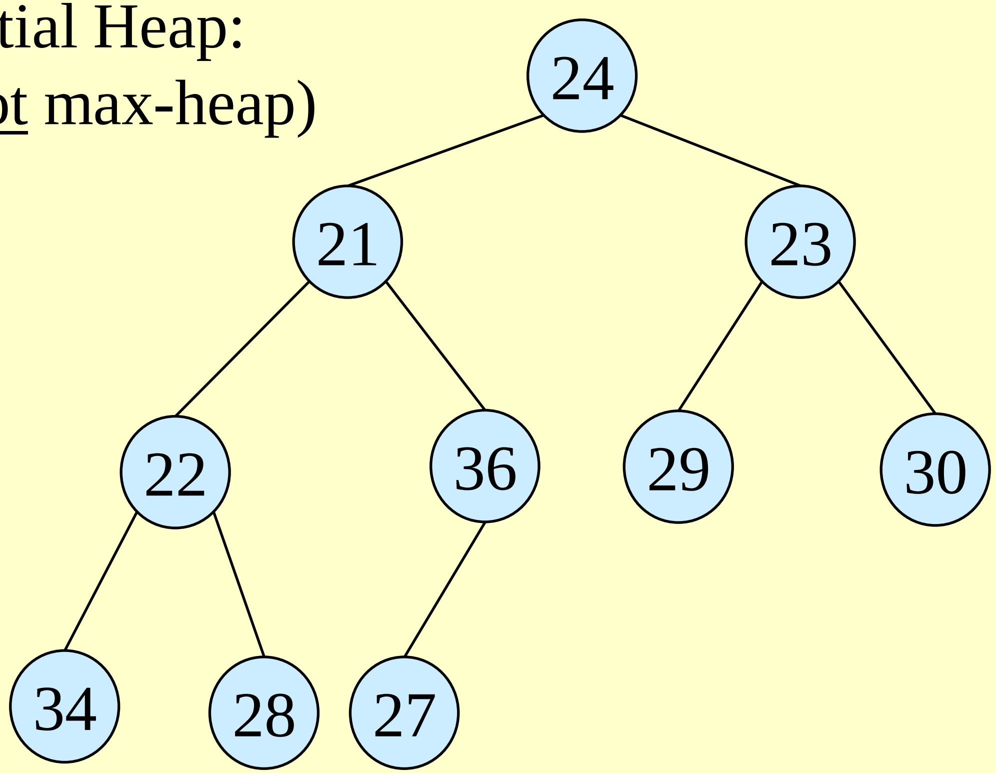
1. $heap\text{-}size[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3.   **do** *MaxHeapify*(*A*, *i*)

BuildMaxHeap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(<u>not</u> max-heap)



BuildMaxHeap – Example

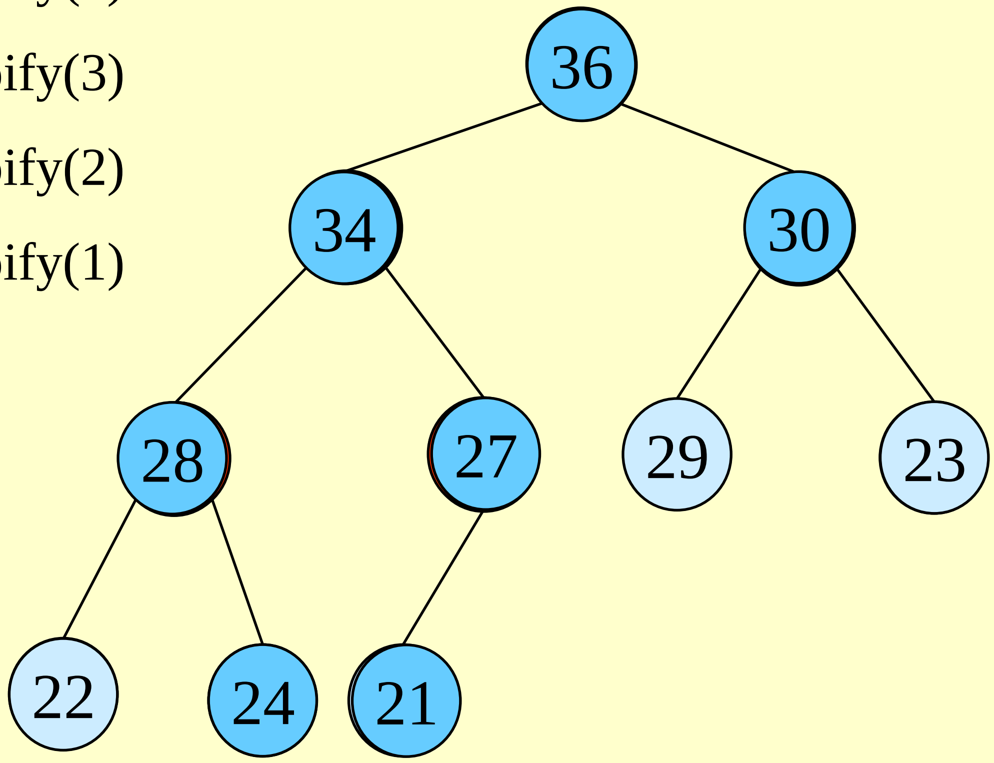
MaxHeapify(⌊10/2⌋ = 5)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)



Correctness of *BuildMaxHeap*

- Loop Invariant: At the start of each iteration of the **for** loop, each node $i+1$, $i+2$, …, n is the root of a max-heap.
- Initialization:
  - » Before first iteration $i = \lfloor n/2 \rfloor$
  - » Nodes $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n are leaves and hence roots of max-heaps.
- Maintenance:
  - » By LI, subtrees at children of node i are max heaps.
  - » Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - » Decrementing i reestablishes the loop invariant for the next iteration.

Running Time of *BuildMaxHeap*

- Loose upper bound:
  - » Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - » $O(\lg n) \times O(n) = O(n\lg n)$
- Tighter bound:
  - » Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - » Height of most nodes smaller than n.
  - » Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - » No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

Heapsort

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in A.
  - » Maximum element is in the root, $A[1]$.
- Move the maximum element to its correct final position.
  - » Exchange $A[1]$ with $A[n]$.
- Discard $A[n]$ – it is now sorted.
  - » Decrement heap-size[A].
- Restore the max-heap property on $A[1..n–1]$.
  - » Call *MaxHeapify*(A, 1).
- Repeat until heap-size[A] is reduced to 2.

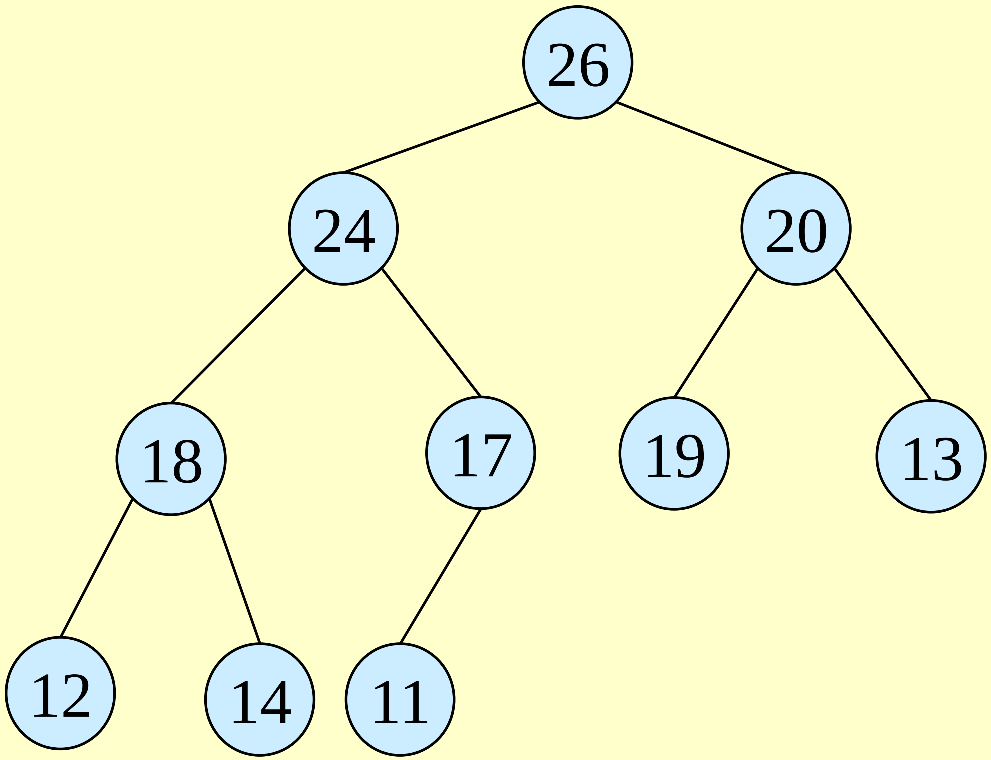
Heapsort(A)

HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i \leftarrow$ *length*[A] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[A] $\leftarrow$ *heap-size*[A] – 1
5. *MaxHeapify*(A, 1)

Heapsort – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10



Algorithm Analysis

<u>*HeapSort*(*A*)</u>

1. Build-Max-Heap(*A*)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. *MaxHeapify*(*A*, 1)

- In-place
- Not Stable
- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting

- MaxHeapify $O(\lg n)$
- BuildMaxHeap $O(n)$
- HeapSort $O(n \lg n)$

Priority Queue

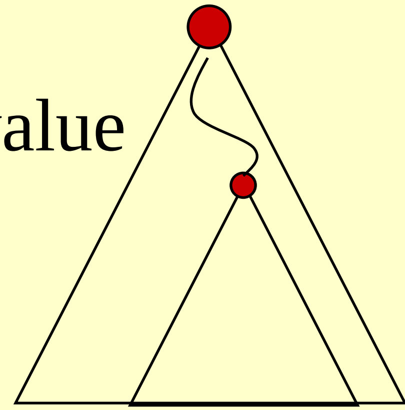
- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - » Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - » In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

- Operations on a max-priority queue:
  - » Insert(S, x) - inserts the element x into the set S
    - $S \leftarrow S \cup \{x\}$.
  - » Maximum(S) - returns the element of S with the largest key.
  - » Extract-Max(S) - removes and returns the element of S with the largest key.
  - » Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



- Min-Heap
  - » For every node excluding the root, value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heap-Extract-Max(A)

Implements the Extract-Max operation.

Heap-Extract-Max(A)
1. if *heap-size*[A] < 1
2. then error "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[heap\text{-}size[A]]$
5. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
6. MaxHeapify(A, 1)
7. return max

Running time : Dominated by the running time of MaxHeapify
$= O(\lg n)$

Heap-Insert(A, key)

Heap-Insert(A, key)

1. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2. $i \leftarrow heap\text{-}size[A]$
4. **while** $i > 1$ **and** $A[\text{Parent}(i)] < key$
5. **do** $A[i] \leftarrow A[\text{Parent}(i)]$
6. $i \leftarrow \text{Parent}(i)$
7. $A[i] \leftarrow key$

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

Heap-Increase-Key(A, i, key)

Heap-Increase-Key(A, i, key)

If $key < A[i]$
    then error "new key is smaller than the current key"
$A[i] \leftarrow key$
while $i > 1$ **and** $A[\text{Parent}[i]] < A[i]$
        do exchange $A[i] \leftrightarrow A[\text{Parent}[i]]$
            $i \leftarrow \text{Parent}[i]$

Heap-Insert(A, key)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
Heap-Increase-Key(A, $heap\text{-}size[A]$, key)

Examples

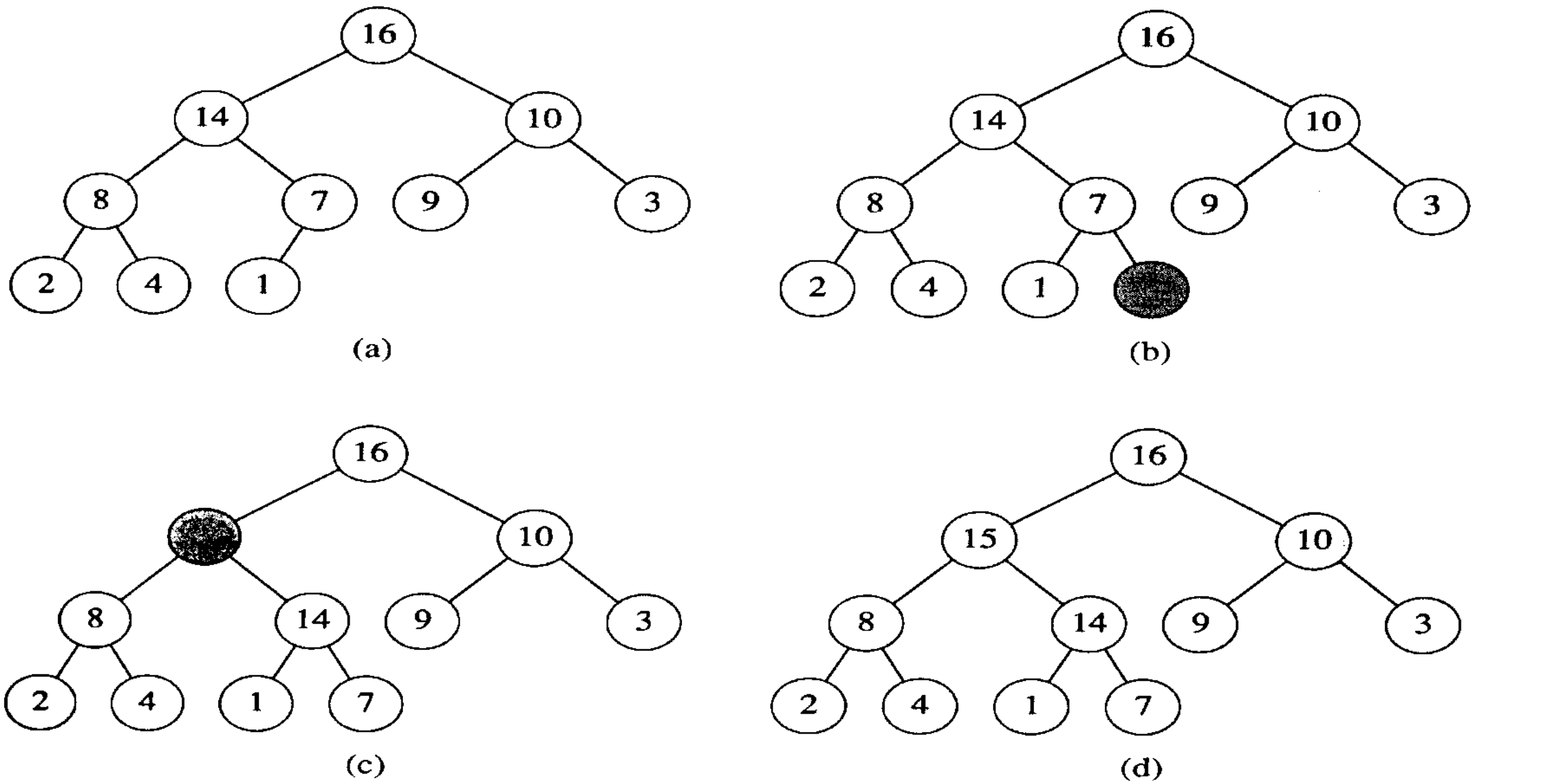


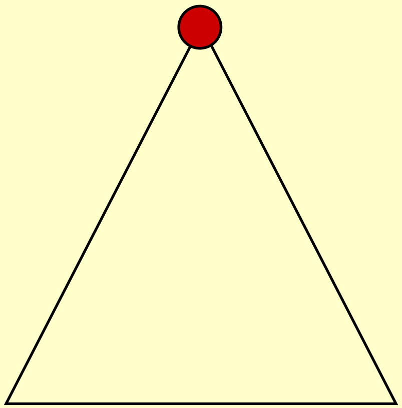
Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Heapsort

(Analysis)

Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height h $\leq \lceil n/2^{h+1} \rceil$



Running Time for MaxHeapify

<u>*MaxHeapify*(*A*, *i*)</u>

1. $l \leftarrow$ left(i)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*[A] and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*[A] **and** $A[r] > A[largest]$
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[largest]$
10. *MaxHeapify*(*A*, *largest*)

Time to fix node *i* and its children = **Θ(1)**

PLUS

Time to fix the subtree rooted at one of *i*'s children = *T*(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

- $T(n) = T(largest) + \Theta(1)$
- $largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a heap

- Use *MaxHeapify* to convert an array A into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(A)

1. $heap\text{-}size[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. **do** *MaxHeapify*(A, i)

Running Time of *BuildMaxHeap*

- Loose upper bound:
  - Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - $O(\lg n) \times O(n) = O(n \lg n)$
- Tighter bound:
  - Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - Height of most nodes smaller than n.
  - Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

Running Time of *BuildMaxHeap*

Tighter Bound for $T(BuildMaxHeap)$

$T(BuildMaxHeap)$

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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$
$$\leq \sum_{h=0}^{\infty} \frac{h}{2^h} \qquad , x = 1/2 \text{ in (A.8)}$$
$$= \frac{1/2}{(1-1/2)^2}$$
$$= 2$$

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

Can build a heap from an unordered array in linear time

Heapsort

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in *A*.
  - » Maximum element is in the root, *A*[1].
- Move the maximum element to its correct final position.
  - » Exchange *A*[1] with *A*[*n*].
- Discard *A*[*n*] – it is now sorted.
  - » Decrement heap-size[*A*].
- Restore the max-heap property on *A*[1..*n*–1].
  - » Call *MaxHeapify*(*A*, 1).
- Repeat until heap-size[*A*] is reduced to 2.

Heapsort(A)

HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i \leftarrow$ *length*[A] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[A] $\leftarrow$ *heap-size*[A] – 1
5. *MaxHeapify*(A, 1)

Algorithm Analysis

<u>*HeapSort*(*A*)</u>

1. Build-Max-Heap(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. $MaxHeapify(A, 1)$

- In-place
- Not Stable
- Build-Max-Heap takes $O(n)$ and each of the n-*1* calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting

- MaxHeapify $O(\lg n)$
- BuildMaxHeap $O(n)$
- HeapSort $O(n \lg n)$

Priority Queue

- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - » Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - » In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

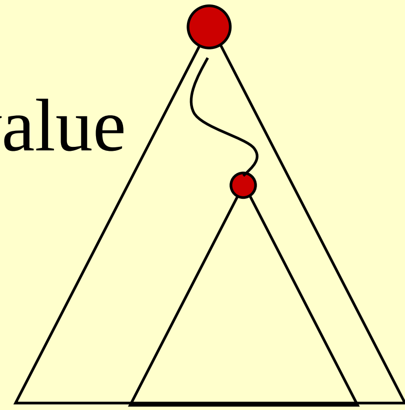
Basic Operations

- Operations on a max-priority queue:
  - » Insert(S, x) - inserts the element x into the set S
    - $S \leftarrow S \cup \{x\}$.
  - » Maximum(S) - returns the element of S with the largest key.
  - » Extract-Max(S) - removes and returns the element of S with the largest key.
  - » Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Shell Sort

Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



- Min-Heap
  - » For every node excluding the root, value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heap-Extract-Max(A)

Implements the Extract-Max operation.

<u>*Heap-Extract-Max*(A)</u>

1. if *heap-size*[A] < 1
2.   then error "heap underflow"
3. *max* $\leftarrow$ A[1]
4. A[1] $\leftarrow$ A[*heap-size*[A]]
5. *heap-size*[A] $\leftarrow$ *heap-size*[A] - 1
6. MaxHeapify(A, 1)
7. return max

Running time : Dominated by the running time of MaxHeapify
$= O(\lg n)$

Heap-Insert(A, key)

Heap-Insert(A, key)

1. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2. $i \leftarrow heap\text{-}size[A]$
4. **while** $i > 1$ **and** $A[\text{Parent}(i)] < key$
5. **do** $A[i] \leftarrow A[\text{Parent}(i)]$
6. $i \leftarrow \text{Parent}(i)$
7. $A[i] \leftarrow key$

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

Heap-Increase-Key(*A*, *i*, *key*)

```
Heap-Increase-Key(A, i, key)
If key < A[i]
   then error "new key is smaller than the current key"
A[i] ← key
while i > 1 and  A[Parent[i]] < A[i]
        do exchange A[i] ↔ A[Parent[i]]
            i ← Parent[i]
```

```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← –∞
Heap-Increase-Key(A, heap-size[A], key)
```

Examples

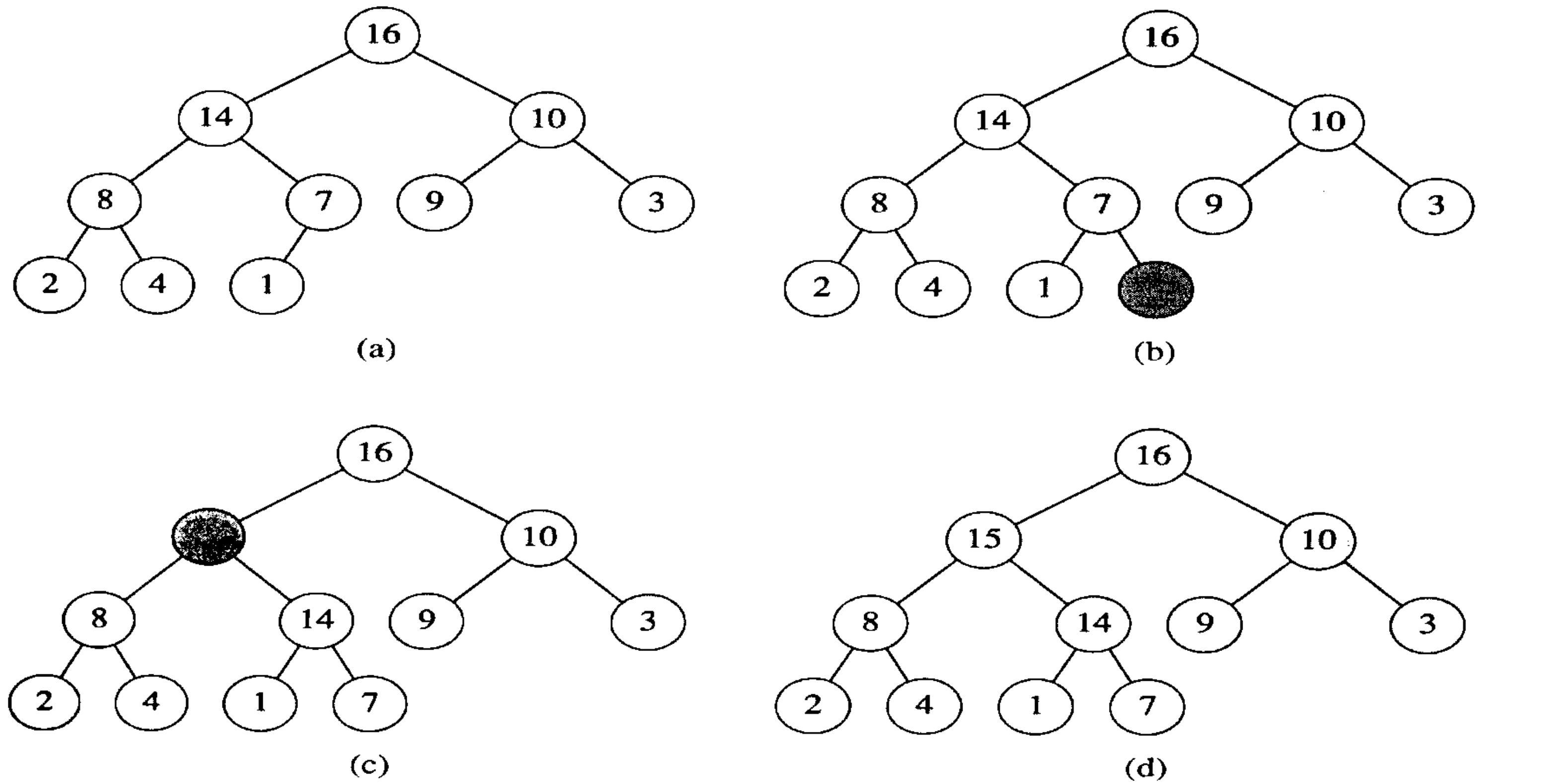


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Lower Bounds &
Sorting in Linear Time

Comparison-based Sorting

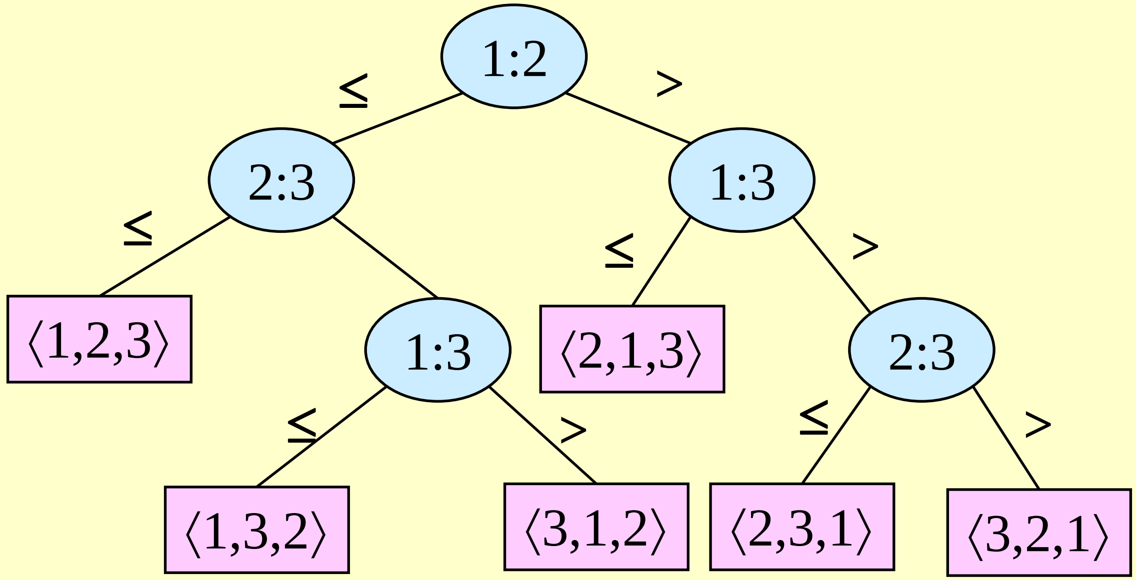
- **Comparison sort**
  - » Only comparison of pairs of elements may be used to gain order information about a sequence.
  - » Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- All our sorts have been comparison sorts
- The best worst-case complexity so far is $\Theta(n \lg n)$ (merge sort and heapsort).
- We prove a lower bound of $n \lg n$, (or $\Omega(n \lg n)$) for any comparison sort, implying that merge sort and heapsort are optimal.

Decision Tree

- Binary-tree abstraction for any comparison sort.
- Represents comparisons made by
  - » a specific sorting algorithm
  - » on inputs of a given size.
- Abstracts away everything else – control and data movement – counting only comparisons.
- Each internal node is annotated by $i{:}j$, which are indices of array elements from their original positions.
- Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ of orders that the algorithm determines.

Decision Tree – Example

For insertion sort operating on three elements.



Contains 3! = 6 leaves.

Decision Tree (Contd.)

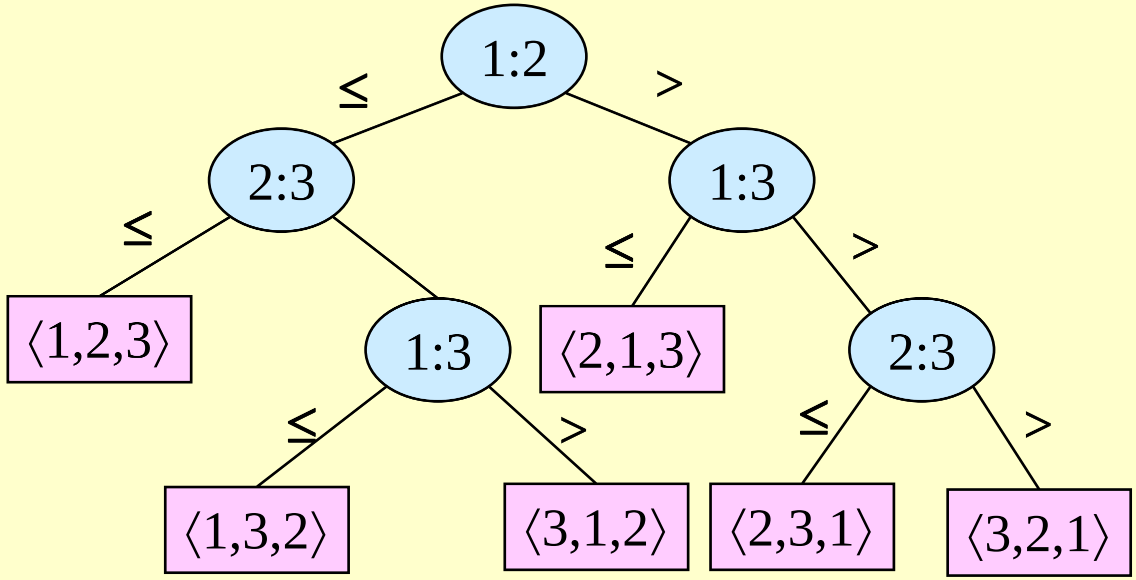
- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \leq a_j$ is made.
  - » If $a_i \leq a_j$, follow left subtree, else follow right subtree.
  - » View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - » Hence, each of the n! permutations must appear at one or more of the leaves of the decision tree.

A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is
  - » Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - Which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - » A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

Optimal sorting for three elements

Any sort of six elements has 5 internal nodes.



There must be a wost-case path of length ≥ 3.

A Lower Bound for Worst Case

***Theorem* 8.1:**
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

- From previous discussion, suffices to determine the height of a decision tree.
- ***h*** – height, ***l*** – no. of reachable leaves in a decision tree.
- In a decision tree for n elements, $l \geq n!$. **Why?**
- In a binary tree of height h, no. of leaves $l \leq 2^h$. **Prove it.**
- Hence, $\mathbf{n! \leq \mathit{l} \leq 2^{\mathit{h}}}$.

Proof – Contd.

- $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Taking logarithms, $h \geq \lg(n!)$.
- $n! > (n/e)^n$. (Stirling's approximation, Eq. 3.19.)
- Hence, $h \geq \lg(n!)$

$$\geq \lg(n/e)^n$$
$$= n \lg n - n \lg e$$
$$= \Omega(n \lg n)$$

Counting Sort

Non-comparison Sorts: Counting Sort

- Depends on a **key *assumption***: numbers to be sorted are integers in $\{0, 1, 2, \ldots, k\}$.
- **Input:** $A[1..n]$, where $A[j] \in \{0, 1, 2, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.
- **Output:** $B[1..n]$ sorted. B is assumed to be already allocated and is given as a parameter.
- **Auxiliary Storage:** $C[0..k]$
- Runs in linear time if $\boldsymbol{k = O(n)}$.

- Example: On board.

Counting Sort: Analysis

Counting-Sort (A, B, k)

CountingSort(A, B, k)

```
for i ← 0 to k
        do C[i] ← 0                                   O(k)
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1                      O(n)
for i ← 2 to k
        do C[i] ← C[i] + C[i –1]                      O(k)
for j ← length[A] downto 1
        do B[C[A[ j ]]] ← A[j]                        O(n)
             C[A[j]] ← C[A[j]]–1
```

Algorithm Analysis

- The overall time is $O(n+k)$. When we have $k=O(n)$, the worst case is $O(n)$.
  - » for-loop of lines 1-2 takes time $O(k)$
  - » for-loop of lines 3-4 takes time $O(n)$
  - » for-loop of lines 5-6 takes time $O(k)$
  - » for-loop of lines 7-9 takes time $O(n)$
- Stable, but not in place.
- No comparisons made: it uses actual values of the elements to index into an array.

Radix Sort

- It was used by the card-sorting machines.
- Card sorters worked on one column at a time.
- It is the algorithm for using the machine that extends the technique to multi-column sorting.
- The human operator was part of the algorithm!
- ***Key idea:*** sort on the "least significant digit" first and on the remaining digits in sequential order. The sorting method used to sort each digit must be "stable".
  - » If we start with the "most significant digit", we'll need extra storage.

An Example

Input		After sorting on LSD		After sorting on middle digit		After sorting on MSD
392		631		928		356
356		392		631		392
446		532		532		446
928	$\Rightarrow$	495	$\Rightarrow$	446	$\Rightarrow$	495
631		356		356		532
532		446		392		631
495		928		495		928
		$\uparrow$		$\uparrow$		$\uparrow$

Radix-Sort(A, d)

RadixSort(A, d)

1. for $i \leftarrow 1$ to d
2. do *use a stable sort to sort array A on digit i*

Correctness of Radix Sort

By induction on the number of digits sorted.

Assume that radix sort works for $d - 1$ digits.

Show that it works for d digits.

Radix sort of d digits $\equiv$ radix sort of the low-order $d - 1$ digits followed by a sort on digit d .

Algorithm Analysis

- Each pass over n d-digit numbers then takes time $\Theta(n+k)$. (Assuming counting sort is used for each pass.)
- There are d passes, so the total time for radix sort is $\Theta(d\,(n+k))$.
- When d is a constant and $k = O(n)$, radix sort runs in linear time.
- Radix sort, if uses counting sort as the intermediate stable sort, does not sort in place.
  - If primary memory storage is an issue, quicksort or other sorting methods may be preferable.

Bucket Sort

- Assumes input is generated by a random process that distributes the elements uniformly over [0, 1).
- **Idea:**
  - » Divide [0, 1) into *n* equal-sized buckets.
  - » Distribute the *n* input values into the buckets.
  - » Sort each bucket.
  - » Then go through the buckets in order, listing elements in each one.

An Example

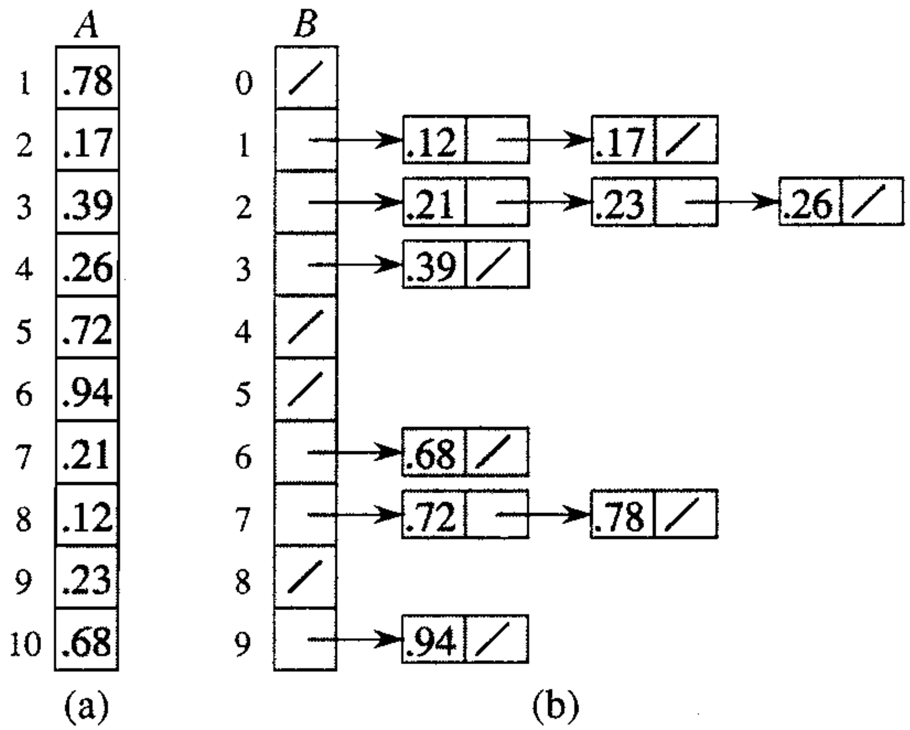


Figure 9.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket-Sort (A)

Input: $A[1..n]$, where $0 \le A[i] < 1$ for all i.
Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(A)

1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ to n
3. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n - 1$
5. **do** sort list $B[i]$ with insertion sort
6. concatenate the lists $B[i]$s together in order
7. **return** the concatenated lists

Correctness of BucketSort

- Consider $A[i]$, $A[j]$. Assume w.o.l.o.g, $A[i] \le A[j]$.
- Then, $\lfloor n \times A[i] \rfloor \le \lfloor n \times A[j] \rfloor$.
- So, $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.
  - » If same bucket, insertion sort fixes up.
  - » If earlier bucket, concatenation of lists fixes up.

Analysis

- Relies on no bucket getting too many values.
- All lines except insertion sorting in line 5 take $O(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow O(n)$ sort time for all buckets.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Analysis – Contd.

- RV n_i = no. of elements placed in bucket $B[i]$.
- Insertion sort runs in quadratic time. Hence, time for bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \qquad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \qquad (E[aX] = aE[X]) \end{aligned} \tag{8.1}$$

Analysis – Contd.

- **Claim:** $E[n_i^2] = 2 - 1/n$. (8.2)
- **Proof:**
- Define indicator random variables.
  - » $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
  - » $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$.
  - » $n_i = \sum_{j=1}^{n} X_{ij}$

Analysis – Contd.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ j\ne k}} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ j\ne k}} E[X_{ij}X_{ik}] \text{ , by linearity of expectation.} \qquad (8.3)$$

Analysis – Contd.

$$E[X_{ij}^2] = 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}$$

$$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}$$

$$= \frac{1}{n}$$

$E[X_{ij}X_{ik}]$ for $j \neq k$:

Since $j \neq k$, X_{ij} and X_{ik} are independent random variables.

$$\Rightarrow E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]$$

$$= \frac{1}{n} \cdot \frac{1}{n}$$

$$= \frac{1}{n^2}$$

Analysis – Contd.

(8.3) is hence,

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}.$$

Substituting (8.2) in (8.1), we have,

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)$$

$$= \Theta(n) + O(n)$$

$$= \Theta(n)$$