BITS Pilani
Pilani Campus

Data Structures & Algorithms Design- SS ZG519
Lecture - 5

Dr. Padma Murali

Lecture 5 Topics

- Solving recurrences
- Sorting Algorithms

Recurrences

When an algorithm contains a recursive call to itself, its running time can be described by a recurrence equation or recurrence.

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Three methods for solving recurrences -(i.e) for obtaining asymptotic "Θ" or "O" bounds on the solution.

(1) Substitution method

→ Guess a solution

→ use mathematical induction to prove our guess is correct.

(2) Recursion tree method

→ converts recurrences into a tree whose nodes represent the costs incurred at various levels of the recursion.

→ use techniques for bounding summations to solve the recurrences.

(3) Master method

→ provide bounds for recurrences of the form

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

where $a \geq 1$, $b > 1$ & $f(n)$ is a given function.

Assumptions

→ we assume that the input n is always an integer.

→ we ignore boundary conditions.

→ while stating & solving recurrences, floors, ceilings are omitted.

... case running

floor, ...

For example: The worst case running time of Merge sort is given by

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

The solution of the above is claimed to be $\Theta(n \log n)$.

When $n > 1$, we have

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{— (2)}$$

But, from our assumption, (1) & (2) is usually written as

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

since although changing the value of T(1) changes the solution to the recurrence, the solution typically doesn't change by more than a constant factor, so the order of growth is unchanged.

Substitution method

(1) Guess the form of the solution.

(2) use mathematical induction to find the constants and show that the solution works.

→ The method is powerful, but can be applied only in cases when we can guess the solution.

→ The method can be used to establish either upper or lower bounds on a recurrence.

For example:

Find an upper bound on the recurrence

$T(n) = 2T(\lfloor n/2 \rfloor) + n$ — (3)

Sol: This is similar to the recurrence which we saw in (1) & (2).

we guess that the solution is

$T(n) = O(n \log n)$.

we have to prove that

$T(n) \le cn \log n$ for an appropriate choice of the constant $c > 0$.

we will use mathematical induction to prove this. ... holds for $\lfloor n/2$

to prove this.
Assume that this bound holds for $\lfloor n/2 \rfloor$.
(ie) $T(\lfloor n/2 \rfloor) \le c(\lfloor n/2 \rfloor) \log \lfloor n/2 \rfloor$.
Substituting into the recurrence ③,
we get

$$
\begin{aligned}
T(n) &\le 2c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + n \\
&\le 2c\frac{n}{2} \log n/2 + n \\
&= cn \log n/2 + n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&= cn \log n + n(1-c) \\
&\le cn \log n \quad , \text{for } c \ge 1
\end{aligned}
$$

(2) Show that the solution of the recurrence
$T(n) = 2T(\lfloor n/2 \rfloor + 15) + n$ is $T(n) = O(n \log n)$.

<u>Solution</u>:

Assume that

$T(m) \le cm \log m$ for all values $m < n$.

$\therefore T(n) = 2(T\lfloor n/2 \rfloor + 15) + n$

$\le 2c(\lfloor n/2 \rfloor + 15) \log(\lfloor \frac{n}{2} \rfloor + 15) + n$

$\le 2c(\frac{n}{2} + 15) \log(\frac{n}{2} + 15) + n$

Now, $\log(\frac{n}{2} + 15) = \log\left[\frac{n}{2}\left(1 + \frac{30}{n}\right)\right]$

$$= \log \frac{n}{2} + \log\left(1 + \frac{30}{n}\right) \qquad \left[\because \log(ab) = \log a + \log b\right]$$

$$= \log \frac{n}{2} + \frac{30}{n} - \frac{1}{2}\left(\frac{30}{n}\right)^2 + \frac{1}{3}\left(\frac{30}{n}\right)^3 - \cdots$$

$$\left[\because \log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots\right]$$

$$\therefore T(n) \le 2c\left(\frac{n}{2} + 15\right) \log\left(\frac{n}{2} + 15\right) + n$$

$$= 2c\left(\frac{n}{2} + 15\right)\left[\log \frac{n}{2} + \frac{30}{n} - \frac{1}{2}\left(\frac{30}{n}\right)^2 + \frac{1}{3}\left(\frac{30}{n}\right)^3 \cdots\right] + n$$

$$\leq 2c\left(\frac{n}{2}+15\right)\left[\log\left(\frac{n}{2}\right)+\frac{30}{n}\right]+n$$

$$\leq 2c\left(\frac{n}{2}\log\frac{n}{2}+15\log\frac{n}{2}+15+\frac{450}{n}\right)+n$$

Choose $n \geq n_0 \geq 30$.

$$\therefore \leq 2c\left[\frac{n}{2}\log\frac{n}{2}+\frac{n}{2}\log\frac{n}{2}+\frac{n}{2}\log\frac{n}{2}+\frac{n}{2}\log\frac{n}{2}\right]+n$$

$$= 2c\cdot\frac{4n}{2}\log n/2 + n$$

$$= 4cn\log n/2 + n$$

$$= 2c \cdot \frac{4n}{2} \log n/2 + n$$

$$= 4cn \log n/2 + n$$

$$= 4cn(\log n - \log 2) + n$$

$$= 4cn \log n - 4cn + n$$

$$= 4cn \log n + n(1 - 4c)$$

$$\leq cn \log n \quad \text{for } c \leq 1/4$$

$$\Rightarrow T(n) = O(n \log n)$$

Example 1

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

Guess: $T(n) = O(\lg n)$

- Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
- Induction hypothesis: $T(n/2) \le d \lg(n/2)$

Proof of induction goal:

$$T(n) = T(n/2) + c \le d \lg(n/2) + c$$

$$= d \lg n - d + c \le d \lg n$$

if: $-d + c \le 0$, $d \ge c$

Example 2

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

Guess: $T(n) = O(n^2)$

- Induction goal: $T(n) \le c\, n^2$, for some c and $n \ge n_0$
- Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$

Proof of induction goal:

$$T(n) = T(n-1) + n \le c\,(n-1)^2 + n$$

$$= cn^2 - (2cn - c - n) \le cn^2$$

$$\text{if: } 2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$$

- For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

Example 3

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

Guess: $T(n) = O(n\lg n)$

- Induction goal: $T(n) \leq cn \lg n$, for some c and $n \geq n_0$
- Induction hypothesis: $T(n/2) \leq cn/2 \lg(n/2)$

Proof of induction goal:

$$T(n) = 2T(n/2) + n \leq 2c\,(n/2)\lg(n/2) + n$$

$$= cn \lg n - cn + n \leq cn \lg n$$

$$\text{if: } -cn + n \leq 0 \Rightarrow c \geq 1$$

(3) Solve the recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

By changing variable.

Let $m = \log n$

$\Rightarrow n = 2^m$

$\therefore T(2^m) + 2T(2^{m/2}) + m.$

Let $S(m) = T(2^m).$

$\Rightarrow S(m) = 2S(m/2) + m$ — (1)

This recurrence is similar to the recurrence

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

whose solution is $T(n) = O(n \log n)$.

$\therefore$ solution of (1) is

$S(m) = O(m \log m)$

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

$= O(\log n \log(\log n))$

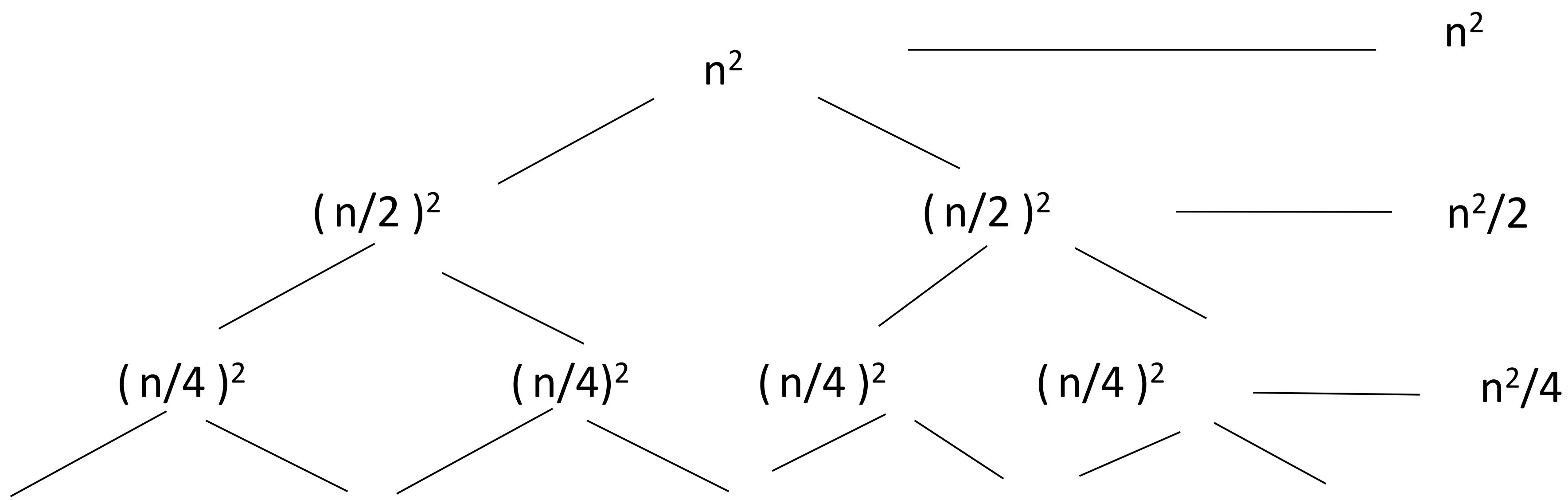
Recursion-tree method

- Convert the recurrence into a tree:
  - Each node represents the cost incurred at that level of recursion
  - Sum up the costs of all levels

- Used to “guess” a solution for the recurrence

SOLVE

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

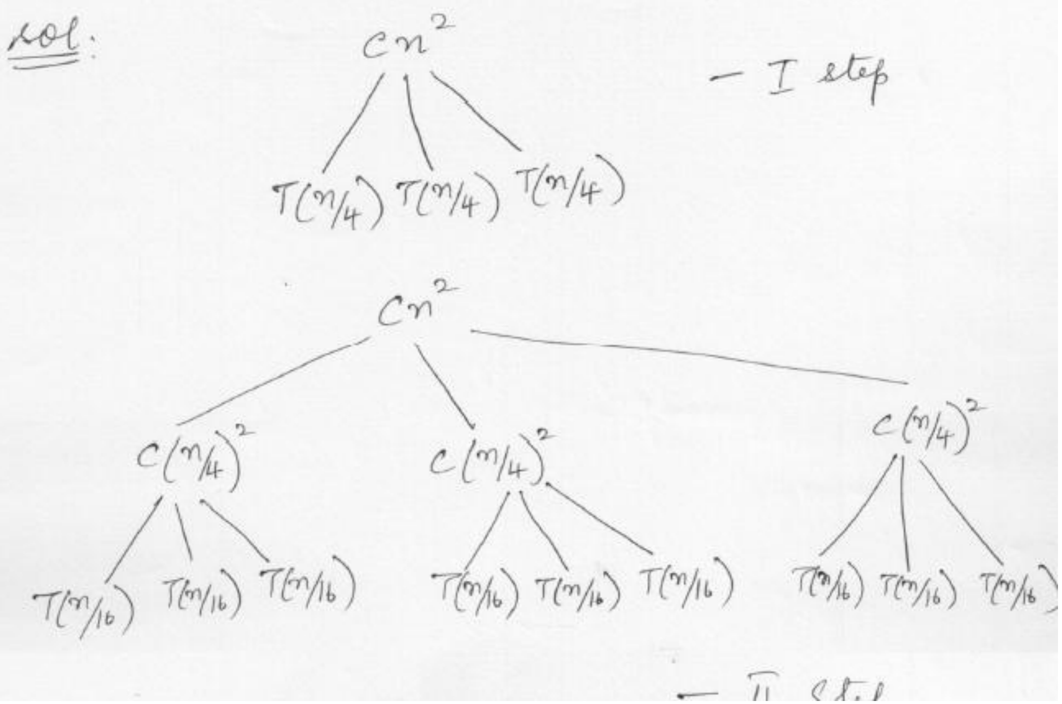


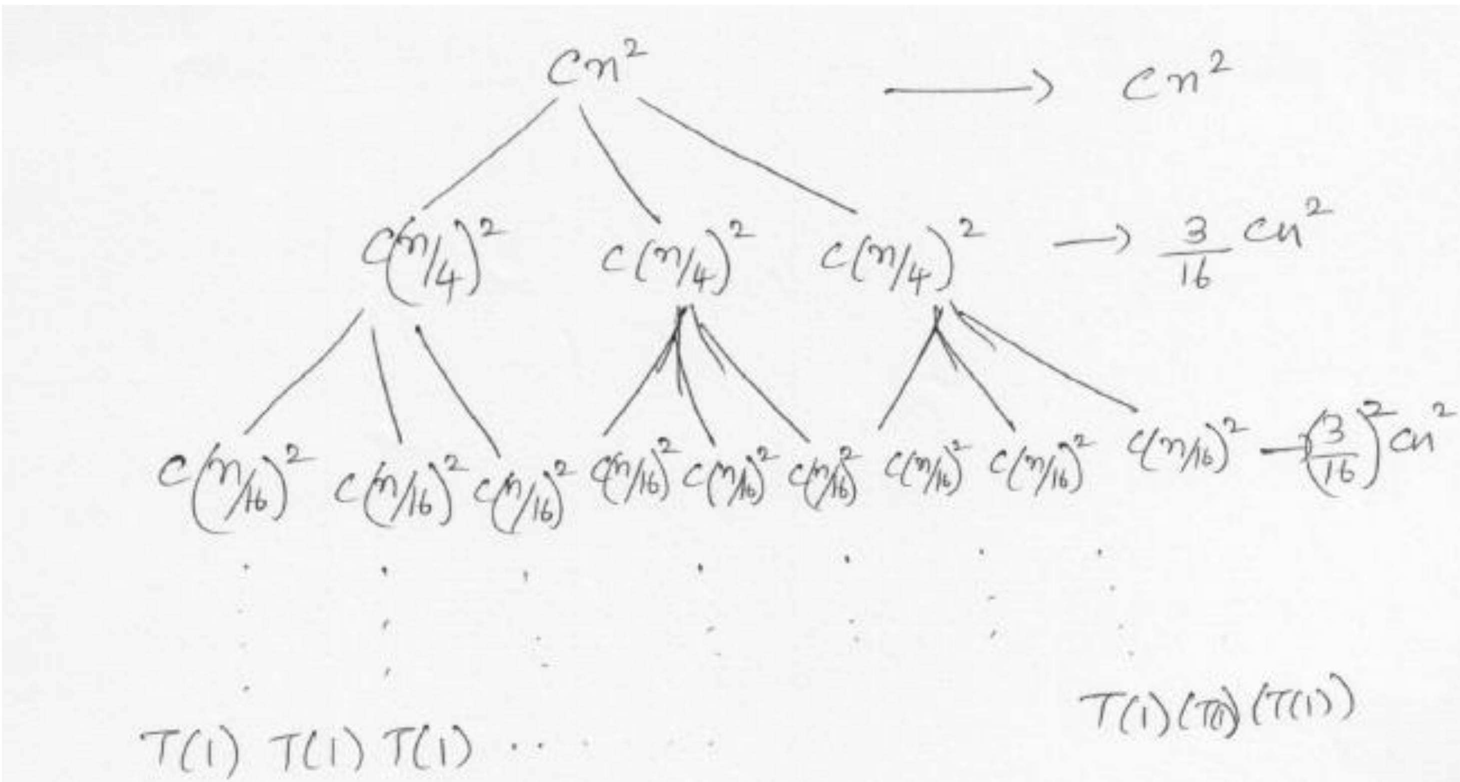
$T(n) = \theta(n^2)$

Recursion Tree Method

(1) Prove that the solution of the recurrence $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + cn^2$ is $O(n^2)$ using a recursion tree.

sol:




cn^2
cn^2
c(n/4)^2 c(n/4)^2 c(n/4)^2
3/16 cn^2
c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2 c(n/16)^2
(3/16)^2 cn^2
T(1) T(1) T(1) ...
T(1) (T(1)) (T(1))

$T(1)\ T(1)\ T(1) \cdots$ $T(1)(T(1))(T(1))$

At the last level, we have

$\frac{n}{4^k} = 1$ (ie) at the k^{th} level

$\Rightarrow n = 4^k$ or $n = (2^2)^k$

or $\Rightarrow n = 2^{2k}$

$k = \log_4 n$

or $2k = \log_2 n$

or $k = \frac{1}{2} \log_2 n$

No. of nodes at each level.

$$3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^k,$$

(ie) 3^k nodes at the k th level

or $k = \log_4 n$

or $3^k = 3^{\log_4 n}$

No. of nodes at each level	cost of node at each level
3^0	cn^2
3^1	$c(n/4)^2$
3^2	$c\left(\frac{n}{4^2}\right)^2$
$\vdots$	$\vdots$
3^{k-1}	$c\left(\frac{n}{4^{k-1}}\right)^2$
3^k	$c\left(\frac{n}{4^k}\right)^2$

Adding up we get

$$T(n) = cn^2 + 3c\left(\frac{n}{4}\right)^2 + 3^2 c\left(\frac{n}{4^2}\right)^2 + \cdots + 3^{\log_4 n - 1} c\left(\frac{n}{4^{\log_4 n - 1}}\right)^2 + 3^{\log_4 n} \cdot c\left(\frac{n}{4^{\log_4 n}}\right)^2$$

$\Rightarrow$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \frac{3^2}{16^2}cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$\left[\because a^{\log_c b} = b^{\log_c a}\right]$$

$\Rightarrow$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \frac{3^2}{16^2}cn^2$$

$$+ \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$\left[\because a^{\log_c b} = b^{\log_c a}\right]$$

$$\leq cn^2\left(\frac{1}{1-\frac{3}{16}}\right) + \Theta(n^{\log_4 3})$$

$$= cn^2 \cdot \frac{16}{13} + \Theta(n^{\log_4 3})$$

since $\log_4 3 < 1$, omitting the term, we can write

$$T(n) \leq \frac{16}{13}cn^2$$

$$\Rightarrow T(n) = O(n^2).$$

Master's Theorem

In Master's theorem, the function $f(n)$ is compared with the function $n^{\log_b a}$.

The solution to the recurrence is determined by the larger of the two functions.

innovate achieve lead

To solve recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

Master's Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the nonnegative integers by the recurrence

$T(n) = a\,T(n/b) + f(n)$ where n/b can be $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then, $T(n)$ can be bounded asymptotically as follows.

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Solve using master's method

(1) $T(n) = 9T(n/3) + n$

Sol. Here, $a = 9$, $b = 3$, $f(n) = n$.

$\therefore n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

Since, $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, applying case 1 of the Master theorem, we have

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

(2) $T(n) = 3T(n/4) + n \log n$.

Sol. $a = 3$, $b = 4$, $f(n) = n \log n$.

$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$.

For, large n,

$$af(n/b) = 3(n/4)\log(n/4) \le \left(\frac{3}{4}\right) n \log n = c f(n)$$

for $c = 3/4$.

$\therefore$ By case 3, solution is

$$T(n) = \Theta(n \log n).$$

(3) $T(n) = T(2n/3) + 1$

Sol.: $a = 1,\ b = 3/2,\ f(n) = 1.$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

Case 2 applies,

since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$,

$\therefore$ the solution is

$T(n) = \Theta(\log n)$.

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.

Three common cases

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

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

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

Three common cases

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

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

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

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

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

Three common cases (cont.)

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

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

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

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

Examples

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

$a = 2,\ b = 2,\ \log_2 2 = 1$

Compare $n^{\log_2 2}$ with $f(n) = n$

$\Rightarrow f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n\lg n)$

Examples (cont.)

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

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\, f(n/b) \le c\, f(n)$

$\Leftrightarrow 2\, n^2/4 \le c\, n^2 \Rightarrow c = \frac{1}{2}$ is a solution $(c<1)$

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

Examples (cont.)

$$T(n) = 2T(n/2) + \sqrt{n}$$

$$a = 2,\ b = 2,\ \log_2 2 = 1$$

Compare n with $f(n) = n^{1/2}$

$\Rightarrow f(n) = O(n^{1-\varepsilon})$ Case 1

$\Rightarrow T(n) = \Theta(n)$

Examples (cont.)

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ Case 3

Check regularity condition:

$$3*(n/4)\lg(n/4) \leq (3/4)n\lg n = c*f(n),\ c=3/4$$

$\Rightarrow T(n) = \Theta(n\lg n)$

The master method

Solve the following

1. $T(n) = T(2n/3) + 1$

2. $T(n) = 9T(n/3) + n$

The master method

1. $T(n) = T(2n/3) + 1$

   $T(n) = \theta(\lg n)$

2. $T(n) = 9T(n/3) + n$

   $T(n) = \theta(n^2)$

Sorting

Iterative methods:

Insertion sort

Bubble sort

Selection sort

Divide and conquer

- Merge sort
- Quicksort

Non-comparison methods

- Counting sort
- Radix sort
- Bucket sort

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

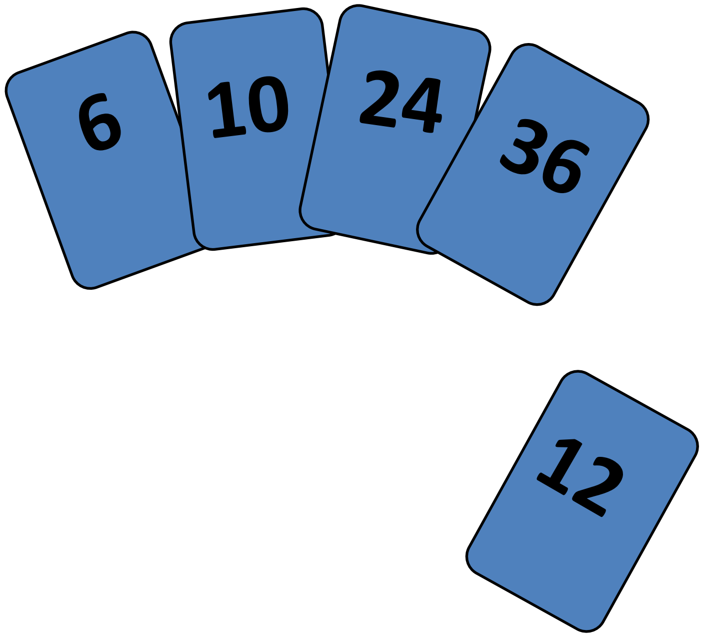
Example:

Input: 8 2 4 9 3 6

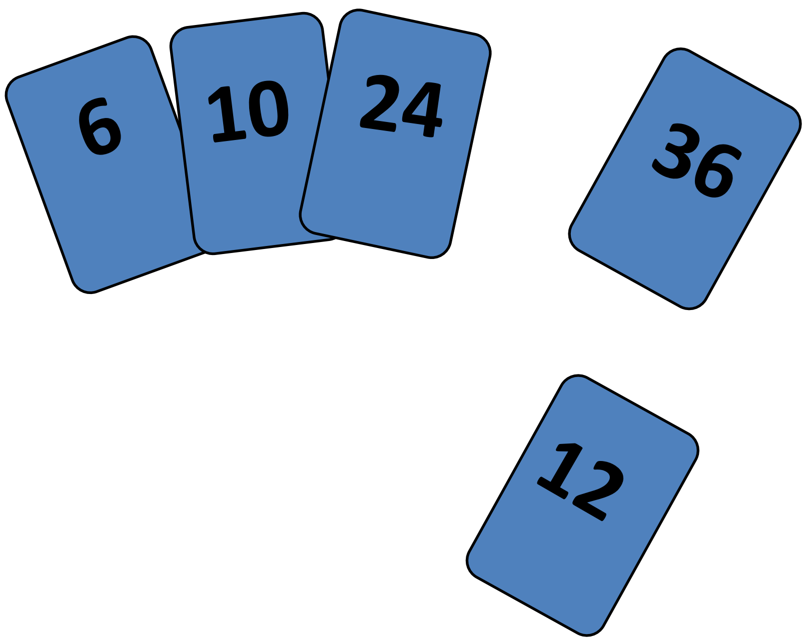
Output: 2 3 4 6 8 9

Insertion Sort

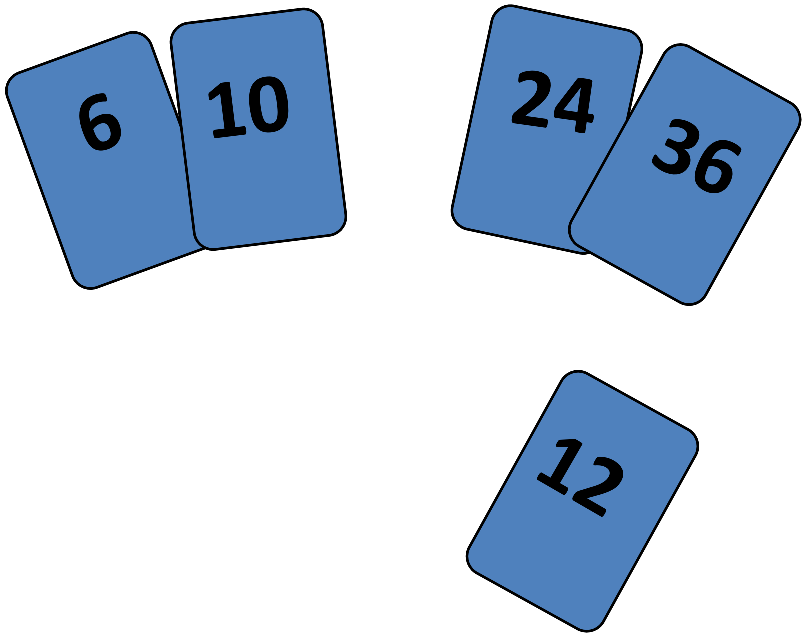
To insert 12, we need to make room for it by moving first 36 and then 24.



Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

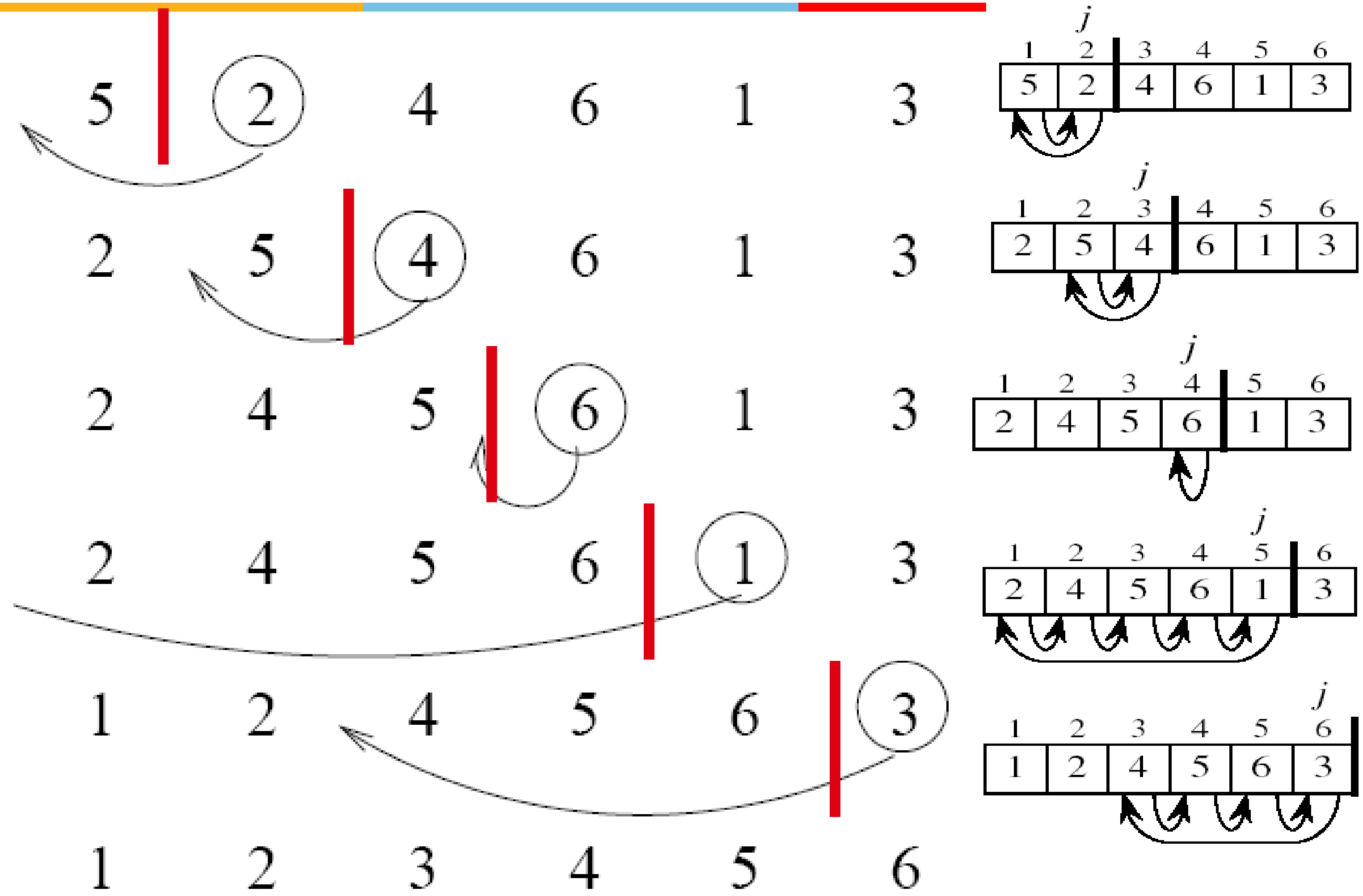
right sub-array

2 5 | 4 6 1 3

sorted

unsorted

Insertion Sort



Insertion Sort

1	2	3	4	5	6	7	8
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

key

Alg.: INSERTION-SORT(*A*)

for j ← 2 **to** n

  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 – sorts the elements in place

Analysis of Insertion Sort

innovate achieve lead

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

t_j: # of times the while statement is executed at iteration j

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

Best Case Analysis

The array is already sorted   **"while** i > 0 and A[i] > key"

- $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$)
- $t_j = 1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$$

$$= an + b = \Theta(n)$$

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

Worst Case Analysis

The array is in reverse sorted order

- Always $A[i] > key$ in **while** loop test
- Have to compare key with all elements to the left of the j-th position $\Rightarrow$ compare with $j-1$ elements $\Rightarrow t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

→ $T(n) = \Theta(n^2)$ order of growth in n^2

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

Sorting Problem

- Input : A sequence of n numbers

  <a1, a2,.....,an>

- Output : A permutation (reordering)

  <a'1, a'2,.....,a'n> of the input sequence such that a'1 <= a'2<=.....<=a'n

- Solutions : Many!
- First Solution : "Insertion Sort"

Insertion Sort

- Big idea:
  - Inserting an element into a sorted list in the appropriate position retains the order.
  - Works the way many people sort a hand of playing cards.
  - Start with an empty left hand and the cards face down on the table.
  - We remove one card from the table and insert it in the correct position in left hand.

Insertion Sort

- To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

- Important :

  At all times the cards in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Insertion Sort

Crucial Idea

– Start with a singleton list – sorted trivially.

– Repeatedly insert elements – one at a time – while keeping it sorted.

– Initially, x will need to be the second element and a[1] the 'sorted part'.

– Sorted part is extended by first inserting the 2^{nd} element, then the 3^{rd} & so on.

Insertion Sort (Con't)

- Example:

5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6 DONE

Insertion Sort – Pseudo Code

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
    *insert A[i] into the sorted sequence A[1,..,i-1]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

Insertion Sort - Analysis

- Best Case Analysis

  The best case for insertion sort occurs when the list is already sorted.
  In this case, insertion sort requires n-1 comparisons
  i.e., O(n) complexity.

- Worst Case Analysis
  for each value of i, what is the maximum number of key comparisons possible?
  - Answer: i -1

- Thus, the total time in the worst case is
  $T(n) = 1+2+3+\ldots\ldots+(n-1)$
  $= n(n-1)/2$
  $= O(n^2)$

Insertion Sort - Analysis

- Average Case Analysis

  - We assume that all permutations of the keys are equally likely as input.

  - We also assume that the keys are distinct.

  - We first determine how many key comparisons are done on average to insert one new element into the sorted segment.

Insertion Sort – Average Case

- When we deal with entry *i*, how far back must we go to insert it?

  Answer:

  There are *i* possible positions: not moving at all, moving by one position up to moving by *i – 1* positions.

- Given randomness, these are equally likely.

Insertion Sort – Average Case

Average no. of comparisons

$$= \frac{1}{i}\sum_{j=1}^{i-1} j + \frac{i-1}{i} = \frac{i-1}{2} + 1 - \frac{1}{i}$$

Total =

$$\sum_{i=1}^{n-1}\left(\frac{i-1}{2} + 1 - \frac{1}{i}\right) = \frac{(n-1)(n-2)}{4} + n - 1 - \sum_{i=1}^{n-1}\frac{1}{i}$$

Insertion Sort – Average Case

- Well known

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

- Thus, the total number of comparisons

  = O(n^2)

Selection Sort

Selection Sort Algorithm (ascending)

1. Find smallest element (of remaining elements).
2. Swap smallest element with current element (starting at index 0).
3. Finished if at the end of the array. Otherwise, repeat 1 and 2 for the next index.

Selection Sort Example(ascending)

37 61 70 75 89

- Smallest is 75
- Swap with index 3
  - Swap with itself

37 61 70 75 89

- Don't need to do last element because there's only one left

37 61 70 75 89

70 75 89 61 37

- Smallest is 37
- Swap with index 0

37 75 89 61 70

- Smallest is 61
- Swap with index 1

37 61 89 75 70

- Smallest is 70
- Swap with index 2

Selection Sort Example(ascending)

Write out each step as you sort this array of 7 numbers (in ascending order)

72 4 17 5 5 64 55

4   72 17 5 5 64 55

4 5   17 72 5 64 55

4 5 5   72 17 64 55

4 5 5 17   72 64 55

4 5 5 17 55   64 72

4 5 5 17 55 64   72

4 5 5 17 55 64 72

Swapping

a = b; b = a; //Does this work?

- a gets overwritten with b's data
- b get overwritten with the new data in a (same data now as b)

Need a temporary variable to store a value while we swap.

```
temp = a;
a = b;
b = temp;
```

Selection Sort Code (ascending)

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIndex = i;
        int min = arr[minIndex];
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < min) {
                minIndex = j;
                min = arr[minIndex];
            }
        }
        int temp = arr[minIndex]; // swap
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Selection Sort Algorithm (descending)

1. Find <u>largest</u> element (of remaining elements).
2. Swap largest element with current element (starting at index 0).
3. Finished if at the end of the array. Otherwise, repeat 1 and 2 for the next index.

98 84 67 1 35

- Largest is 35
- Swap with index 3

98 84 67 35 1

- Don't need to do last element because there's only one left

98 84 67 35 1

Selection Sort Example(descending)

84 98 35 1 67

- Largest is 98
- Swap with index 0

98 84 35 1 67

- Largest is 84
- Swap with index 1
  - Swap with itself

98 84 35 1 67

- Largest is 67
- Swap with index 2

Selection Sort Example(descending)

Write out each step as you sort this array of 7 numbers (in descending order)

72 4 17 5 5 64 55

72 4 17 5 5 64 55

72 64 17 5 5 4 55

72 64 55 5 5 4 17

72 64 55 17 5 4 5

72 64 55 17 5 4 5

72 64 55 17 5 5 4

72 64 55 17 5 5 4

Selection Sort Code (ascending)

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int maxIndex = i;
        int max = arr[maxIndex];
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] > max) {
                maxIndex = j;
                max = arr[maxIndex];
            }
        }
        int temp = arr[maxIndex]; // swap
        arr[maxIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Selection Sort Efficiency

$$n^2 \; comparisons$$

- n is the number of elements in array

$$O(n^2) \; time \; complexity$$

- Big O notation, will talk about this later

Inefficient for large arrays

Why use it?

Memory required is small

- Size of array (you're using this anyway)
- Size of one variable (temp variable for swap)

Selection sort is useful when you have limited memory available

- Inefficient otherwise when you have lots of extra memory

Relatively efficient for small arrays

Selection Sort

Alg.: SELECTION-SORT(*A*)

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

8	4	6	9	2	3	1

Analysis of Selection Sort

innovate achieve lead

Alg.: SELECTION-SORT(*A*)

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Divide-and-Conquer

Divide the problem into a number of sub-problems

- Similar sub-problems of smaller size

Conquer the sub-problems

- Solve the sub-problems recursively
- Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner

Combine the solutions to the sub-problems

- Obtain the solution for the original problem

Merge Sort Approach

To sort an array $A[p \,..\, r]$:

Divide

- Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each

Conquer

- Sort the subsequences recursively using merge sort
- When the size of the sequences is 1 there is nothing more to do

Combine

- Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(A, p, r)

p			q				r
1	2	3	4	5	6	7	8
5	2	4	7	1	3	2	6

if p < r ▷ Check for base case

  then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

    MERGE-SORT(A, p, q) ▷ Conquer

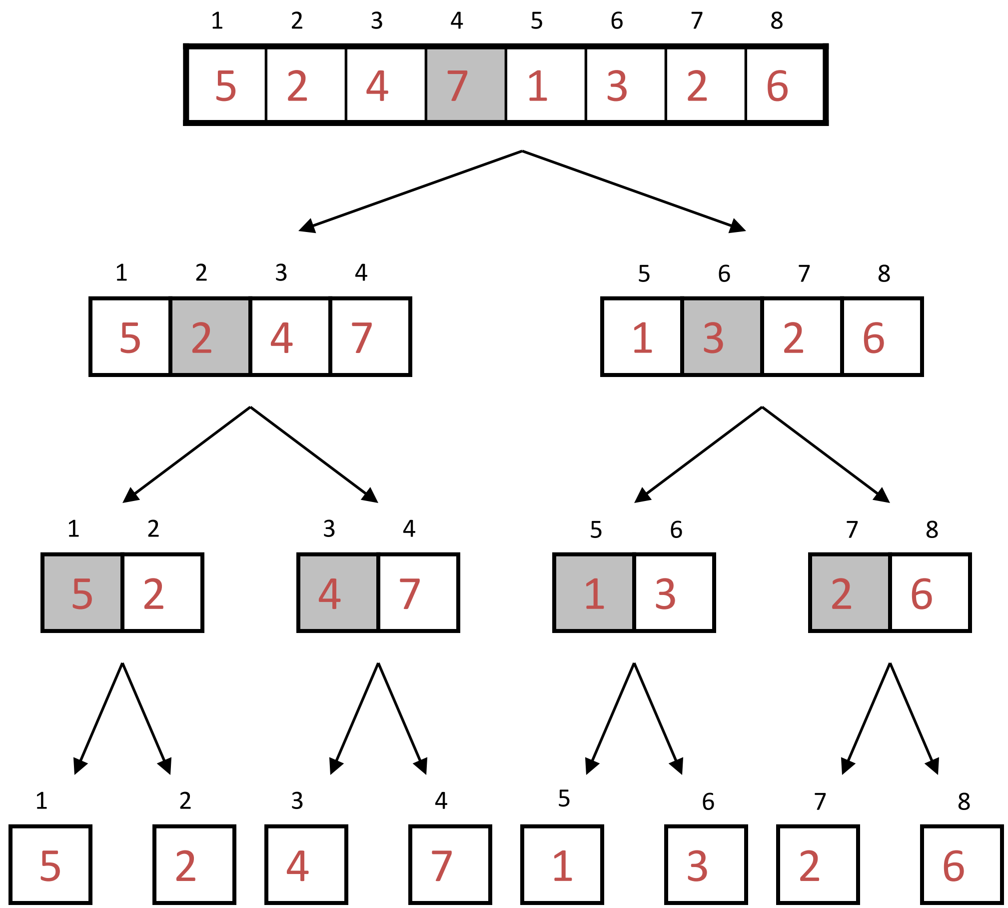
    MERGE-SORT(A, q + 1, r) ▷ Conquer

    MERGE(A, p, q, r) ▷ Combine

Initial call: MERGE-SORT(A, 1, n)

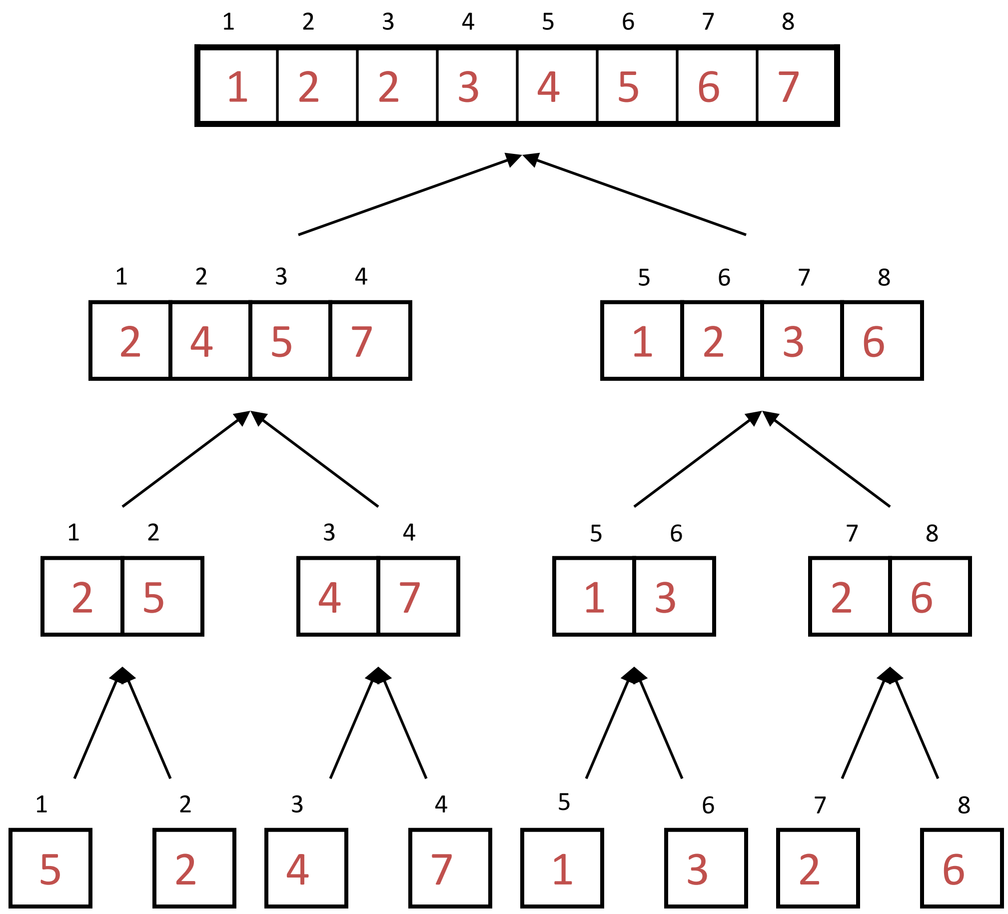
Example – *n* Power of 2

[Example](#)

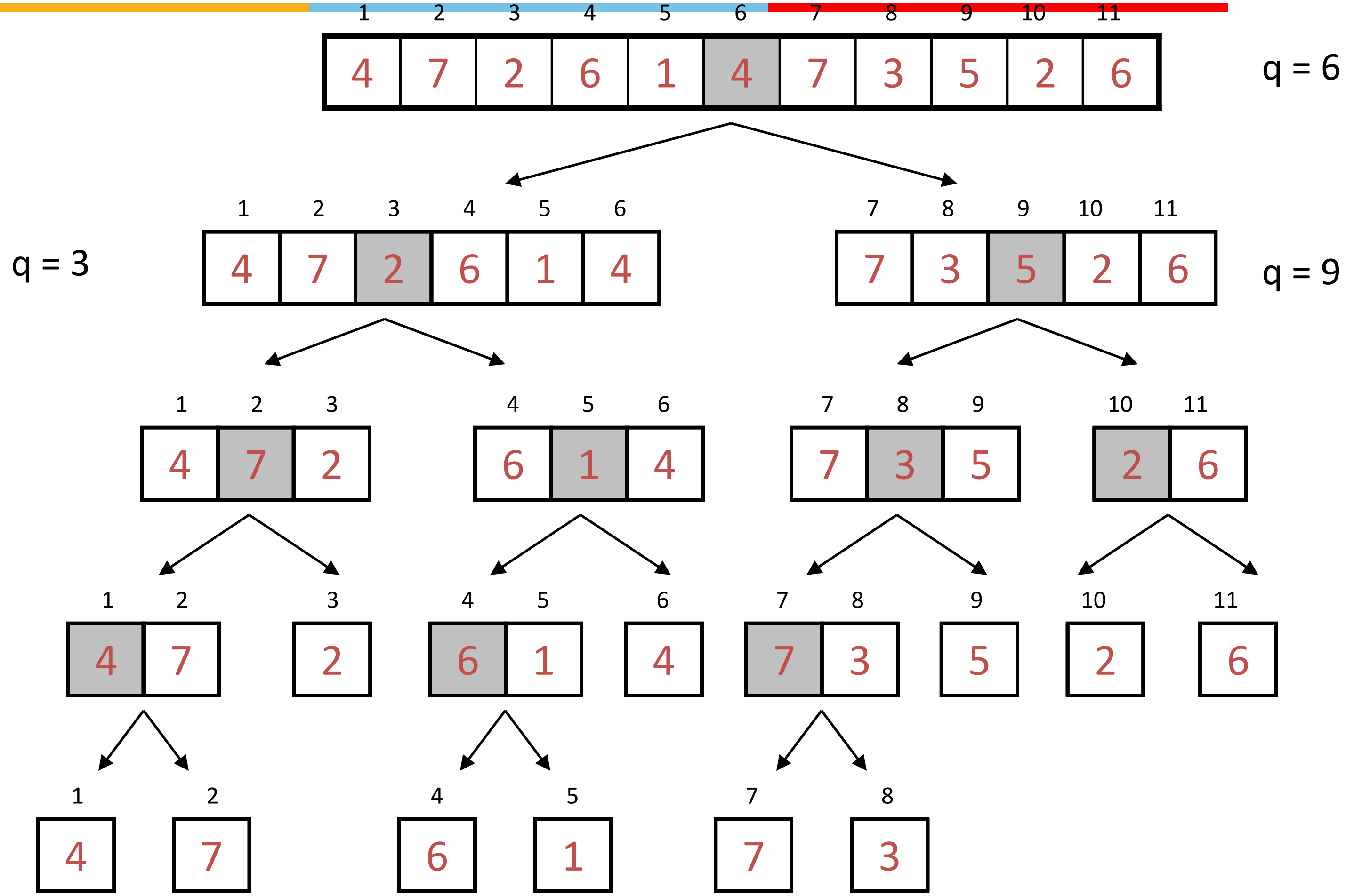


q = 4

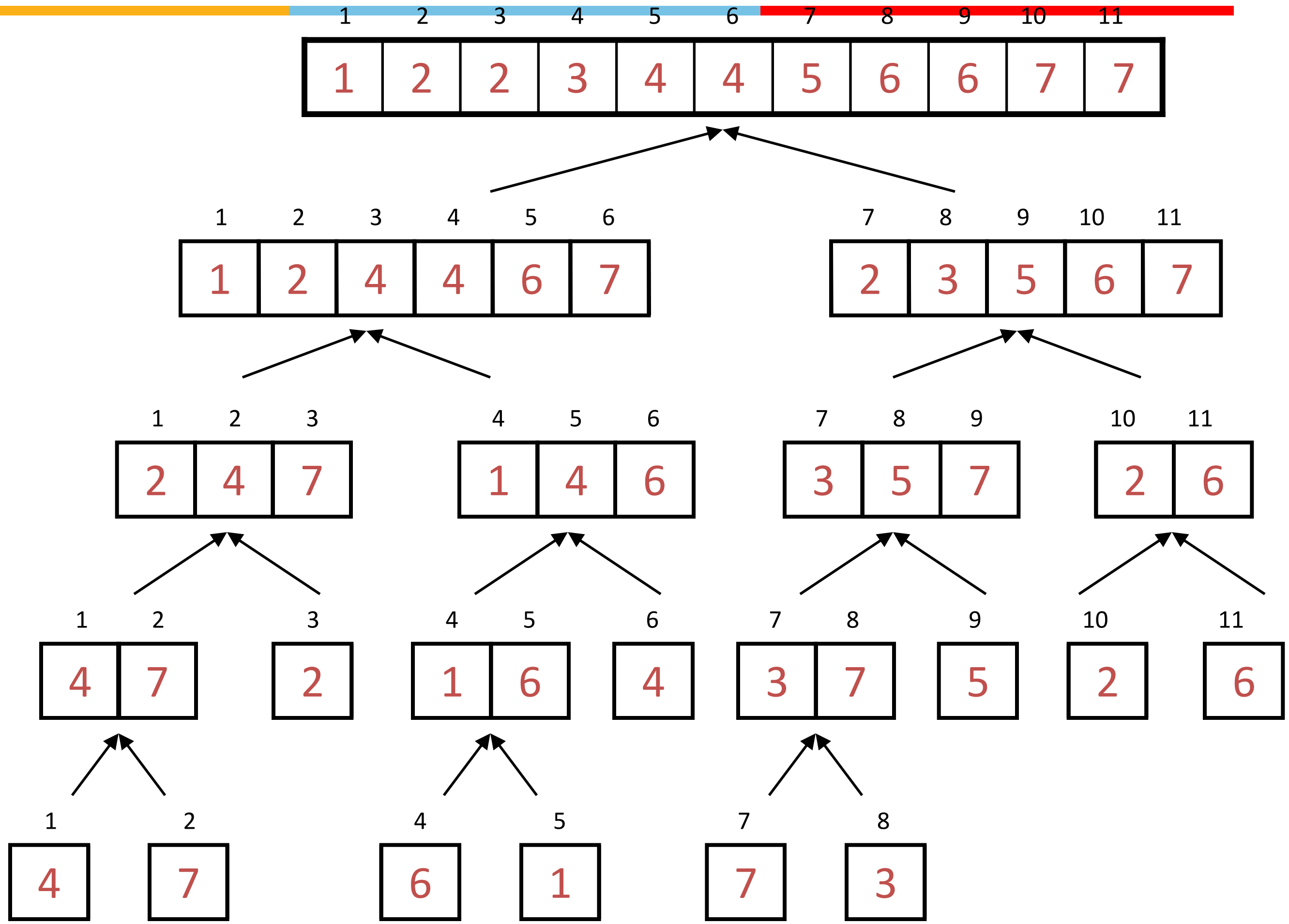
Example – *n* Power of 2



Example – *n* Not a Power of 2



Example – *n* Not a Power of 2

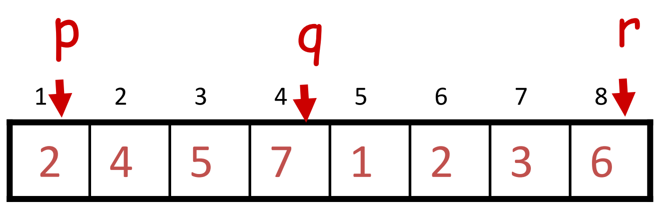


Merging

Input: Array A and indices p, q, r such that $p \le q < r$

- Subarrays $A[p \, . \, . \, q]$ and $A[q + 1 \, . \, . \, r]$ are sorted

Output: One single sorted subarray $A[p \, . \, . \, r]$

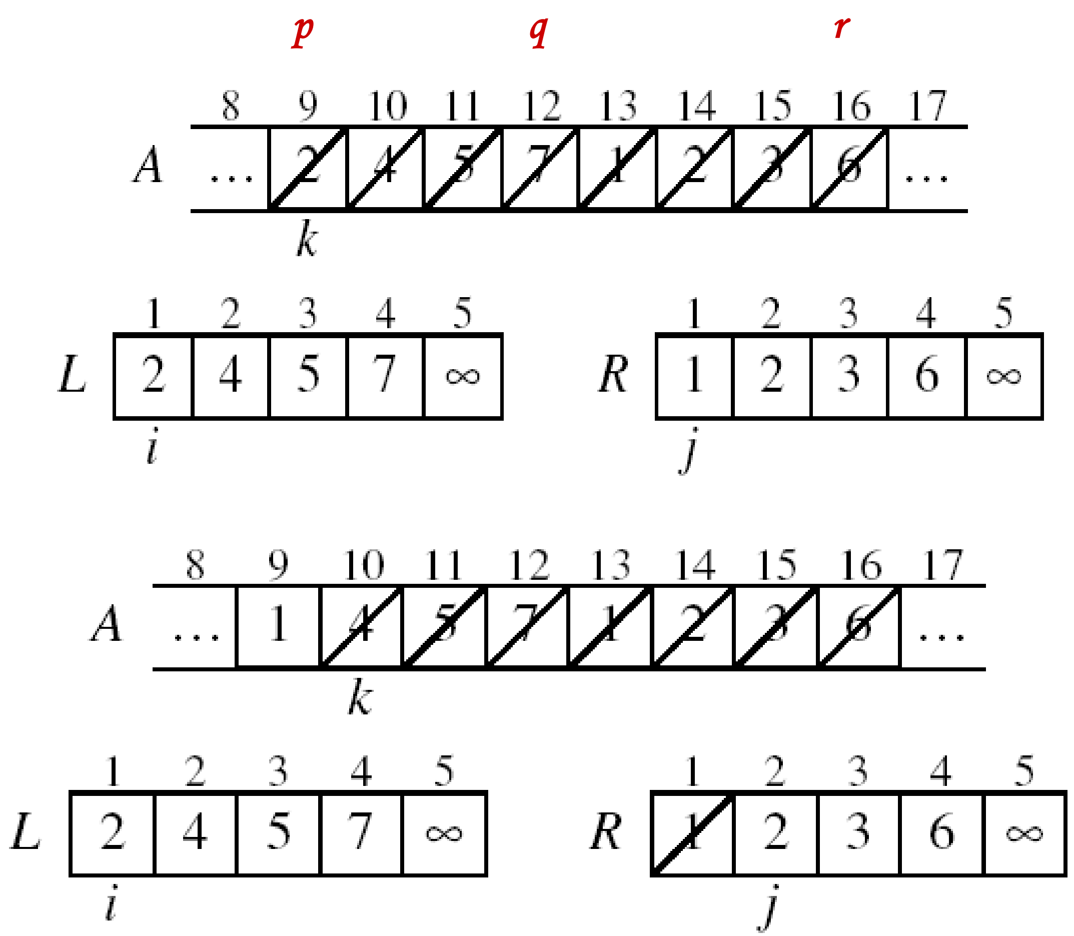


Merging

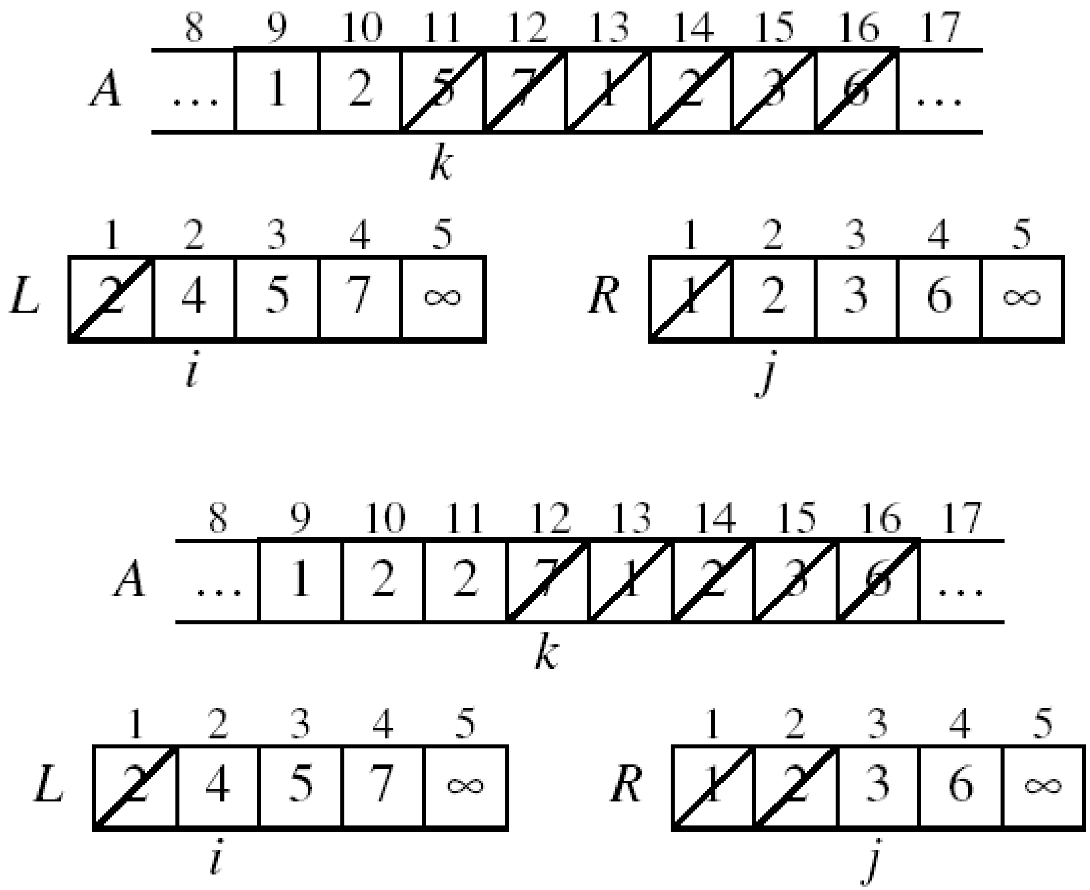
Idea for merging:

- Two piles of sorted cards
  - Choose the smaller of the two top cards
  - Remove it and place it in the output pile
- Repeat the process until one pile is empty
- Take the remaining input pile and place it face-down onto the output pile

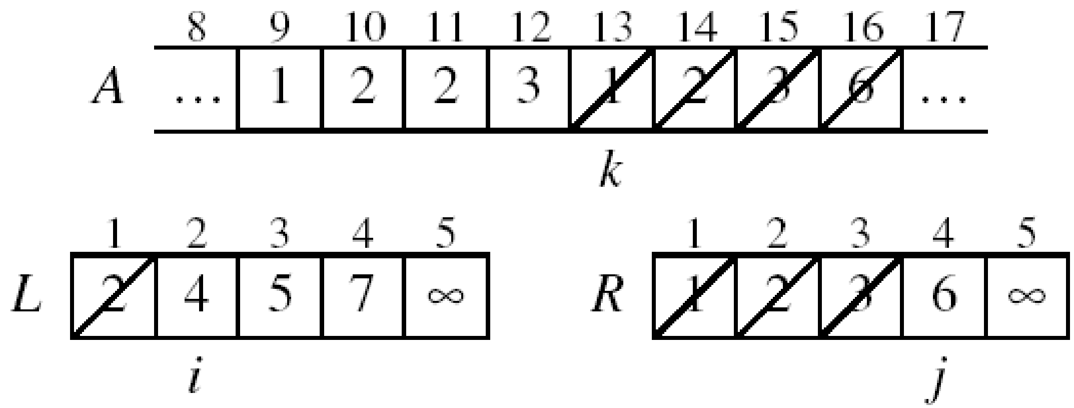
MERGE(A, 9, 12, 16)

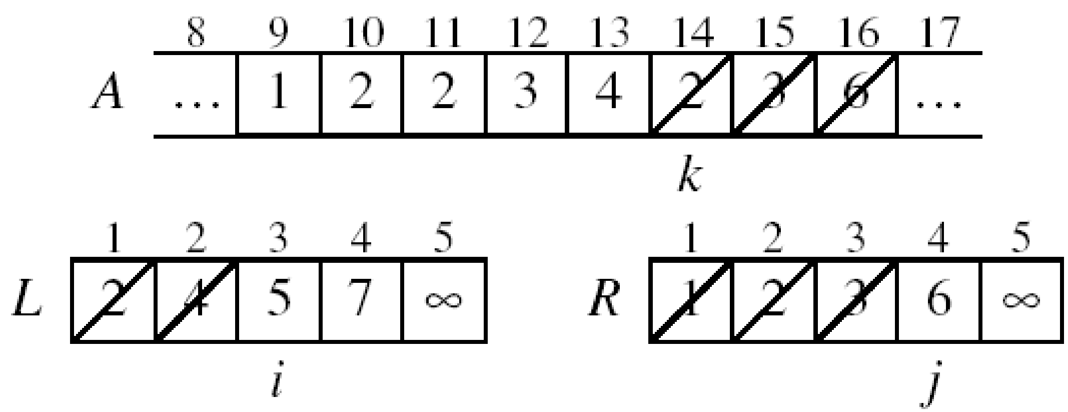


Example: MERGE(A, 9, 12, 16)

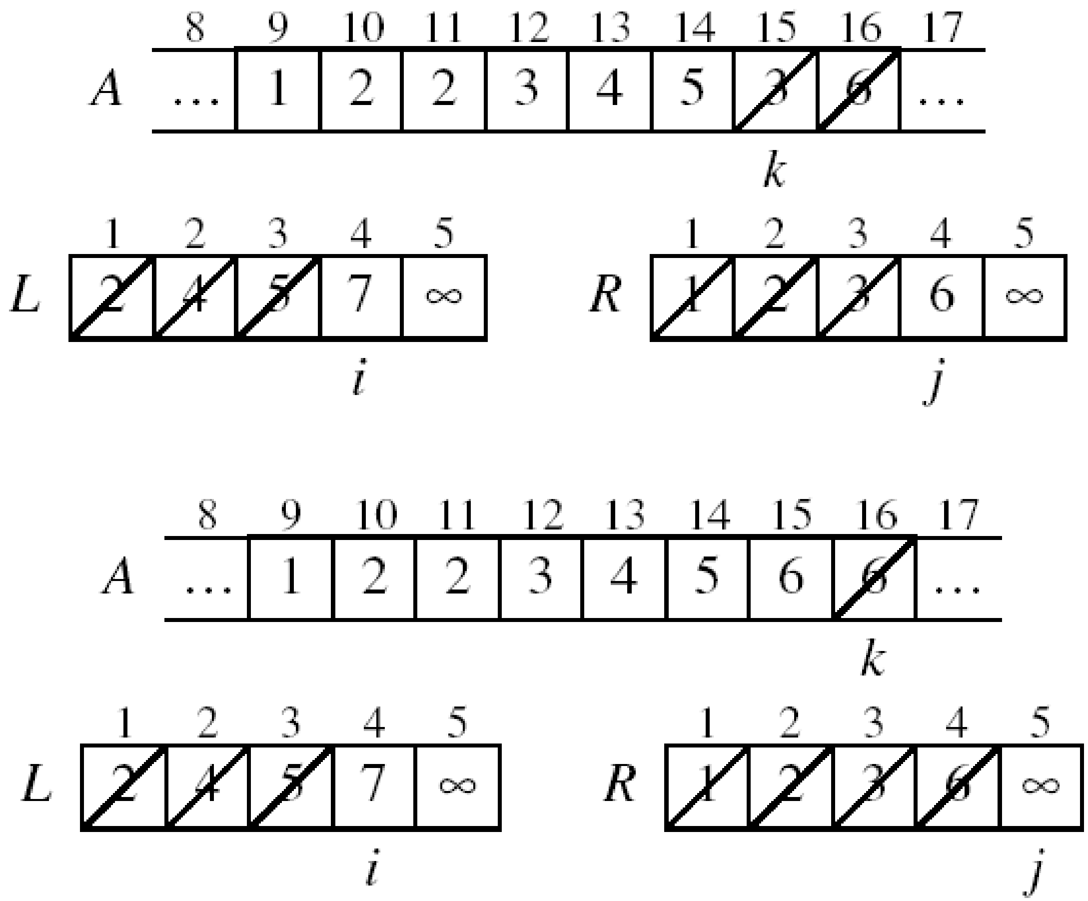


Example (cont.)

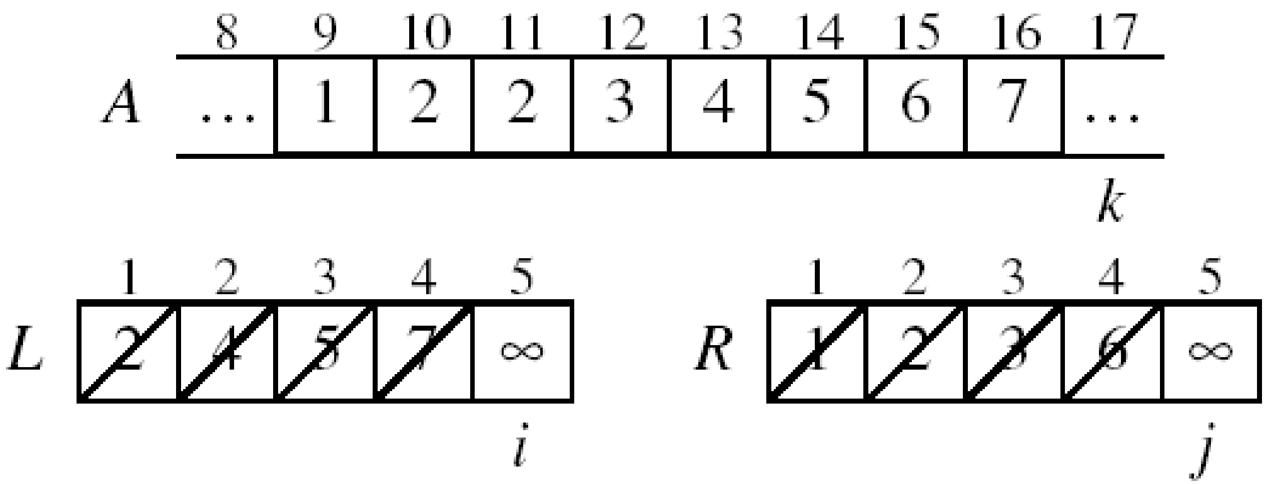




Example (cont.)

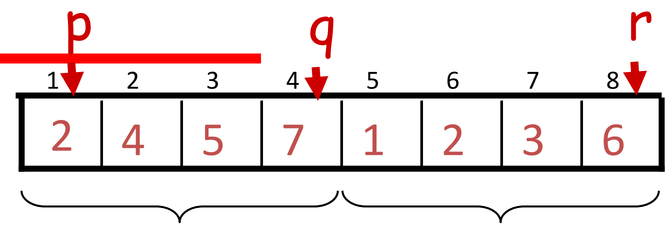


Example (cont.)



Done!

Merge - Pseudocode



Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \ldots n_1 + 1]$ and the next n_2 elements into $R[1 \ldots n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$

	p			q	
L	2	4	5	7	∞

	q + 1			r	
R	1	2	3	6	∞

Running Time of Merge

Initialization (copying into temporary arrays):

- $\Theta(n_1 + n_2) = \Theta(n)$

Adding the elements to the final array (the last **for** loop):

- n iterations, each taking constant time $\Rightarrow \Theta(n)$

Total time for Merge:

- $\Theta(n)$

Analyzing Divide-and Conquer Algorithms

The recurrence is based on the three steps of the paradigm:

- $T(n)$ – running time on a problem of size n
- **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
- **Conquer** (solve) the subproblems $aT(n/b)$
- **Combine** the solutions $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT Running Time

Divide:

– compute q as the average of p and r: $D(n) = \Theta(1)$

Conquer:

– recursively solve 2 subproblems, each of size $n/2 \Longrightarrow 2T(n/2)$

Combine:

– MERGE on an n-element subarray takes $\Theta(n)$ time $\Longrightarrow C(n) = \Theta(n)$

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

Solve the Recurrence

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

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Merge Sort

- **Divide:** Divide the n_element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce a sorted list.
- The general algorithm for the merge sort is as follows:
- If the list is of size greater than 1, then

  a. Find the mid-position of the list.
  b. Merge sort the first sublist.
  c. Merge sort the second sublist.
  d. Merge the first sublist and the second sublist.

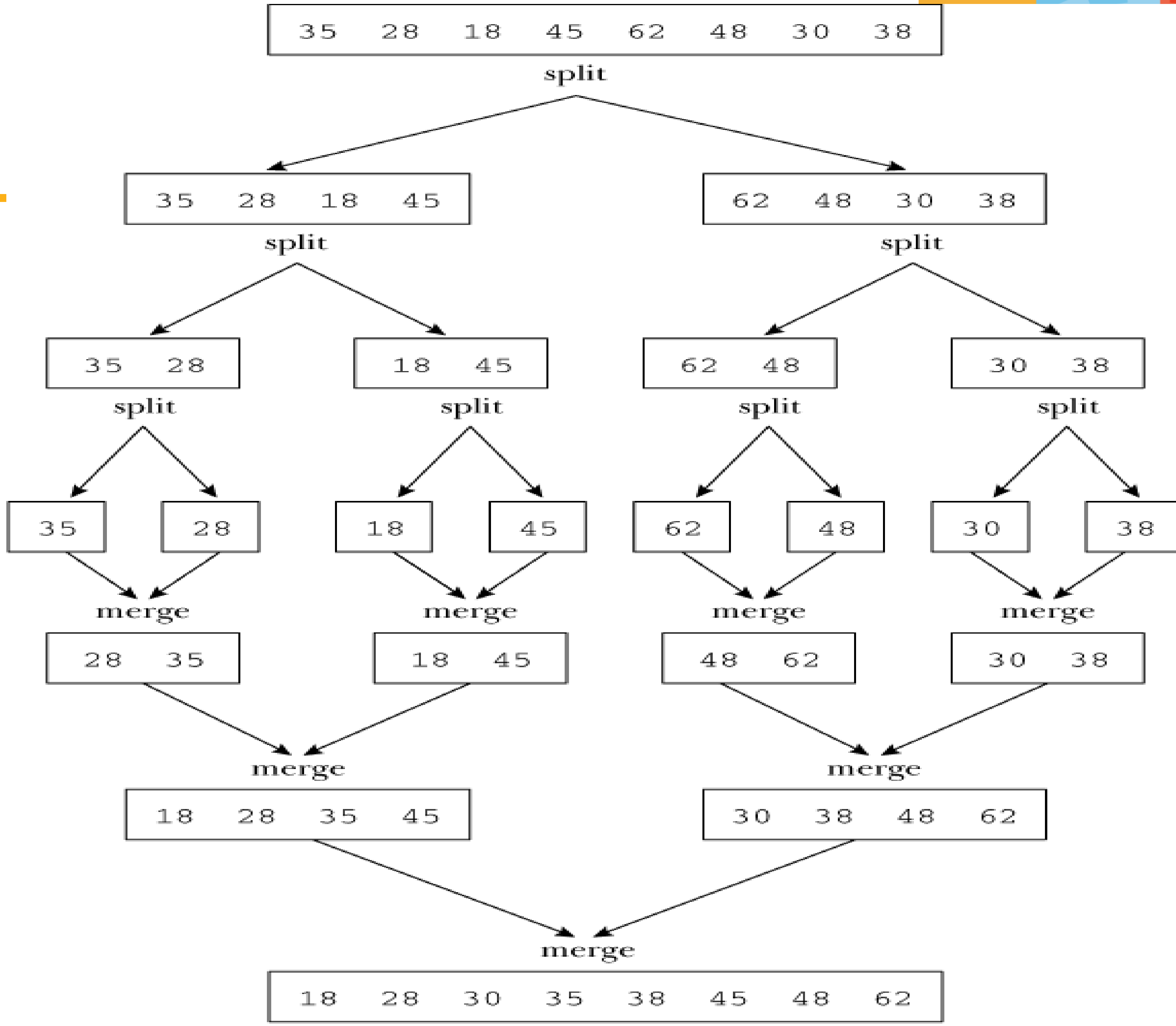


FIGURE 9.22 Merge sort process

Merging two sorted Array

- Once the sublists are sorted, the next step in the merge sort algorithm is to merge the sorted sublists.
- Suppose L_1 and L_2 are two sorted lists as follows:
  - L_1: 2, 7, 16, 35
  - L_2: 5, 20, 25, 40, 50
- Merge L_1 and L_2 into a third list, say L_3.
- The merge process is as follows:

  repeatedly compare, using a loop, the elements of L_1 with the elements of L_2 and copy the smaller element into L_3.

Example

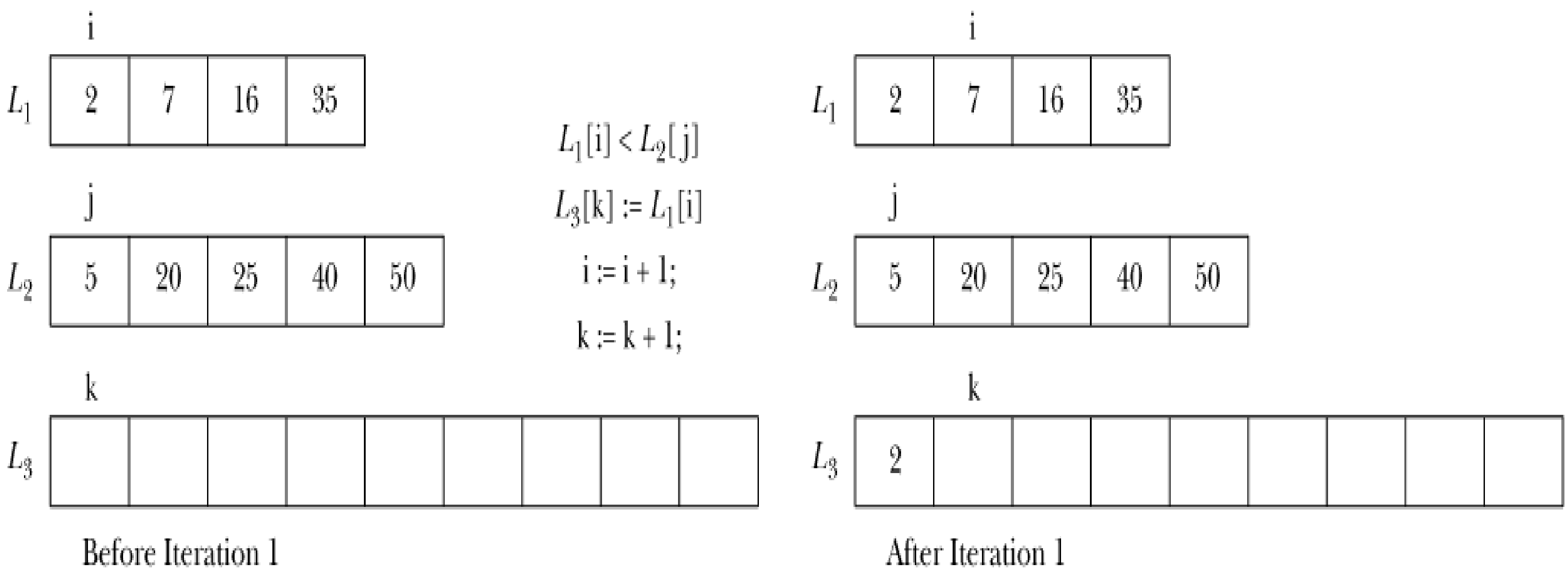


FIGURE 9.23 L_1, L_2, and L_3 before and after the first iteration

- First compare $L_1[1]$ with $L_2[1]$ and see that $L_1[1] < L_2[1]$, so copy $L_1[1]$ into $L_3[1]$

Time : If both the list has n elements each then tne merging process takes 2n time in the worst case.

Merge Sort (Complexity)

- Running time analysis:
  - $T(n)$: worst-case running time of merge sort to sort n numbers (assume n is a power of 2)

Running time analysis can be modeled as an recurrence equation:

$$T(1) = 1, \quad \text{if } n=1$$

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

Merge Sort Complexity (Con't)

- Running time analysis (Con't):

$$
\begin{aligned}
T(1) &= 1 && \textit{Initial condition} \\
T(n) &= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 4T(\frac{n}{4}) + 2n \\
&= 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n \\
&= 8T(\frac{n}{8}) + 3n \\
&\vdots \\
&= 2^k T(\frac{n}{2^k}) + kn && Since\ n = 2^k,\ we\ have\ k = log_2 n \\
&= nT(1) + n\ log_2\ n && Since\ T(1) = 1 \\
&= n + n\ log_2\ n \\
&= O(n\ log\ n)
\end{aligned}
$$

Design Strategy

Divide and Conquer

- is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S

- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Merge-Sort Review

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm $mergeSort(S, C)$
  Input sequence S with n elements, comparator C
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1, C)$
    $mergeSort(S_2, C)$
    $S \leftarrow merge(S_1, S_2)$