

# Introduction

By  
Shital Dongre  
Asst. Prof.  
IT Dept,  
VIT, Pune



Program

Data

Algo

- ▶ Algo– Ways for Data transformation
- ▶ Data structure–
  - Stores data
  - makes algorithm simpler
  - easier to maintain & often faster.

# Why Data Structure

- ▶ Sophisticated data str– simpler the algo
- ▶ Simple algo– less expensive, less code
- ▶ Logic is simple– modifications are less likely to introduce errors
- ▶ Easier to repair defects, make modifications, or add enhancements
- ▶ Ex– 1. array 2. Stack ex– pile of plates, box of books 3. Non-Linear data str– Tree– used for indexing, routing table

# Syllabus

## Section 1: Arrays , Stack , Queue, Linked List

- ▶ Single and Multidimensional arrays, Time & Space Complexity Analysis.
- ▶ **Sorting Techniques**: Insertion, Bucket, Merge, Quick and heap sort.
- ▶ **Search techniques** Binary search, Fibonacci search.
- ▶ **Linked Lists**: Dynamic memory allocation, Singly Linked Lists, Doubly linked Lists, Circular linked lists, and Generalized linked lists, Applications of Linked list.
- ▶ **Stack**: stack representation using array and Linked list. Applications of stack: Recursion, Validity of parentheses, Expression conversions and evaluations, mazing problem.
- ▶ **Queue**: representation using array and Linked list, Types of queue, Applications of Queue: Job Scheduling, Josephus problem etc.

## Section2: Trees, Graphs, Hashing

- ▶ Trees:- Basic terminology, representation using array and linked list, Tree Traversals: Recursive And Non recursive, Operations on binary tree: Finding Height, Leaf nodes, counting no of Nodes etc., Construction of binary tree from traversals, Binary Search trees(BST): Insertion, deletion of a node from BST. Threaded Binary tree (TBT): Creation and traversals on TBT, AVL tree.
- ▶ Graph:-Terminology and representation, Traversals, Connected components and Spanning trees: Prims and Kruskal's Algorithm, Shortest Paths and Transitive Closures: Single Source All destinations (Dijkstra'sAlgorithm), all pair shortest path algorithm, Topological Sort.
- ▶ Hasing:- Hashing techniques: Hash table, Hash functions, and Collision, Cuckoo Hashing.

# Arrays

- ▶ An Array is a collection of variables of the same type that are referred to through a common name.
- ▶ Declaration  
type var\_name[size]

e.g

```
int A[6];  
double d[15];
```

# Array Initialization

After declaration, array contains some garbage value.

## Static initialization

```
int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

## Run time initialization

```
int i;  
int A[5];  
for(i = 0; i < 5; i++)  
    A[i] = 6 - i;
```



# Array – Accessing an element

int A[6];

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
0x100	0x100	0x100	0x101	0x101	0x102
0	4	8	2	6	0
6	5	4	3	2	1

6 elements of 4 bytes each,

total size = 6 x 4 bytes = 24 bytes

Read an element

```
int tmp = A[2];
```

Write to an element

```
A[3] = 5;
```

# Strings in C

- ▶ No "Strings" keyword
- ▶ A string is an array of characters.

OR

```
char string[] = "hello world";  
char *string = "hello world";
```

A C String of Characters with Addresses

1234:0000	1234:0001	1234:0002	1234:0003	1234:0004	1234:0005	1234:0006	1234:0007	1234:0008	1234:0009	1234:000A	1234:000B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
H	e	l	l	o		W	o	r	l	d	\0

# Significance of NULL character '\0'

```
char string[] = "hello world";  
printf("%s", string);
```

- Compiler has to know where the string ends
- '\0' denotes the end of string

Some more characters (do \$man ascii):

'\n' = new line, '\t' = horizontal tab, '\v' = vertical tab, '\r' = carriage return  
'A' = 0x41, 'a' = 0x61, '\0' = 0x00

# Pointers in C

- A char pointer points to a single byte.
- An int pointer points to first of the four bytes.
- A pointer itself has an address where it is stored in the memory. Pointers are usually four bytes.

```
int *p; ⇔ int* p;
```

- ▶ \* is called the dereference operator
- \*p gives the value pointed by p

```
int i = 4;  
p = &i;
```



- & (ampersand) is called the reference operator
- &i returns the address of variable i

# Pointer Arithmetic

- ▶ A 32-bit system has 32 bit address space.
- ▶ To store any address, 32 bits are required.
- ▶ Pointer arithmetic :  $p+1$  gives the next memory location assuming cells are of the same type as the base type of  $p$ .

# Pointer Arithmetic: Example

```
int *p, x = 20;  
p = &x;  
printf("p      = %p\n", p);  
printf("p+1 = %p\n", (int*)p+1);  
printf("p+1 = %p\n", (char*)p+1);  
printf("p+1 = %p\n", (float*)p+1);  
printf("p+1 = %p\n", (double*)p+1);
```

**Sample output:**

```
p      = 0022FF70  
p+1 = 0022FF74  
p+1 = 0022FF71  
p+1 = 0022FF74  
p+1 = 0022FF78
```

# Pointers and arrays

- ▶ Pointers and arrays are tightly coupled.

```
char a[] = "Hello World";
```

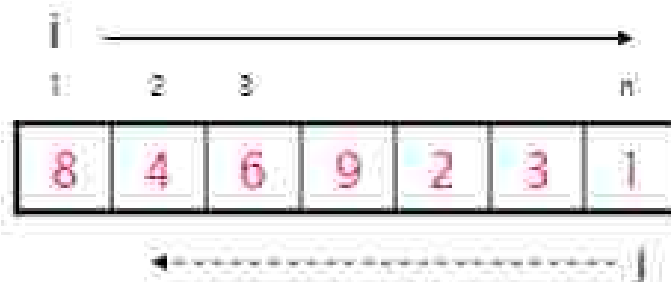
```
char *p = &a[0];
```

char a[12], *p = &a[0];											
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)	*(p+10)	*(p+11)
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
H	e	l	l	o		W	o	r	l	d	'\0'

# Bubble Sort

## ▶ Idea:

- Repeatedly pass through the array
- Swaps adjacent elements that are out of order



- ▶ Easier to implement, but slower than Insertion sort



# Example

8	4	6	9	2	1	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

1	4	6	9	2	8	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

8	4	6	9	1	2	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

8	4	6	1	9	2	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

8	1	4	6	9	2	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

8	1	4	6	9	2	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

8	1	4	6	9	2	3
---	---	---	---	---	---	---

$i = 1$   $\leftarrow$   $\dots$   $j$

1	8	4	6	9	2	3
---	---	---	---	---	---	---

$i = 2$   $\leftarrow$   $\dots$   $j$

1	2	8	4	6	9	3
---	---	---	---	---	---	---

$i = 3$   $\leftarrow$   $\dots$   $j$

1	2	3	8	4	6	9
---	---	---	---	---	---	---

$i = 4$   $\leftarrow$   $\dots$   $j$

1	2	3	4	8	6	9
---	---	---	---	---	---	---

$i = 5$   $\leftarrow$   $\dots$   $j$

1	2	3	4	6	8	9
---	---	---	---	---	---	---

$i = 6$   $\leftarrow$   $\dots$   $j$

1	2	3	4	6	8	9
---	---	---	---	---	---	---

$i = 7$   $\leftarrow$   $\dots$   $j$

$j$

# Bubble Sort

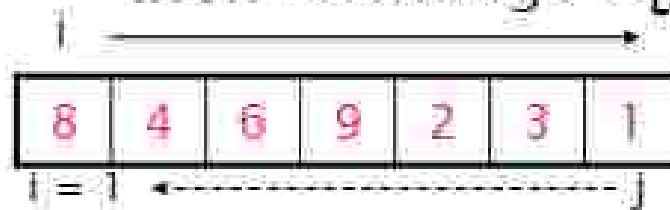
*Alg.* BUBBLESORT(A)

for  $i \leftarrow 1$  to  $\text{length}[A]$

do for  $j \leftarrow \text{length}[A]$  downto  $i + 1$

do if  $A[j] < A[j - 1]$

then exchange  $A[j] \leftrightarrow A[j - 1]$



## 2-Dimensional Arrays (Array of arrays)

```
int d[3][2];
```

Access the point 1, 2 of the array:

```
d[1][2]
```

Initialize (without loops):

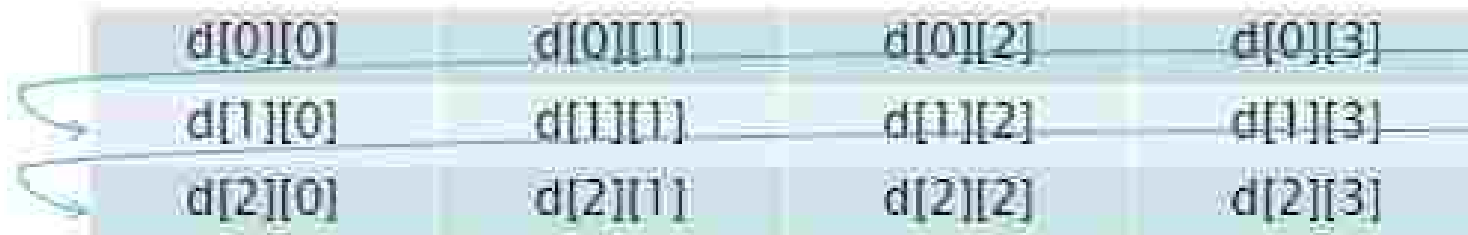
```
int d[3][2] = {{1, 2}, {4, 5}, {7, 8}};
```

# More about 2-Dimensional arrays

A Multidimensional array is stored in a row major format.

A two dimensional case:

→ next memory element to  $d[0][3]$  is  $d[1][0]$



What about memory addresses sequence of a three dimensional array?

→ next memory element to  $t[0][0][0]$  is  $t[0][0][1]$

# Sorting Algorithms

# Sorting

- ▶ Arrangement of data items in ascending or descending order.
- ▶ For unstructured data or records, keys are used to distinguish or sort items.
- ▶ Ex. Insertion, selection, bubble, merge etc.

# Insertion Sort

- ▶ Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

# 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

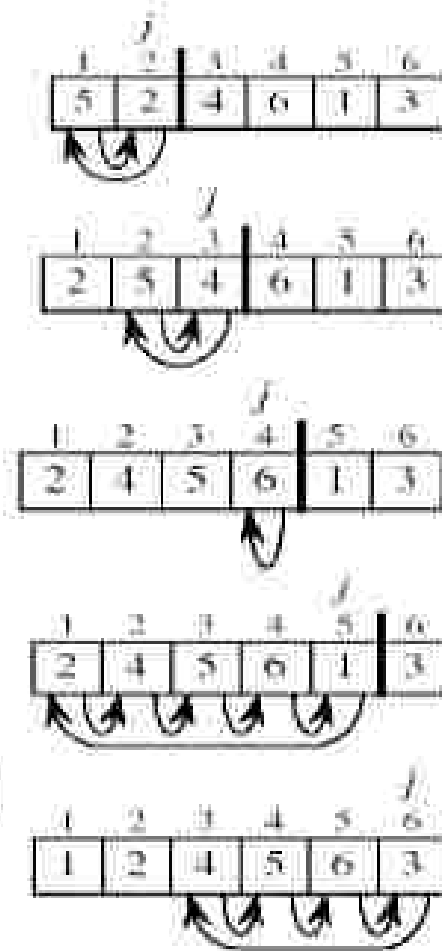
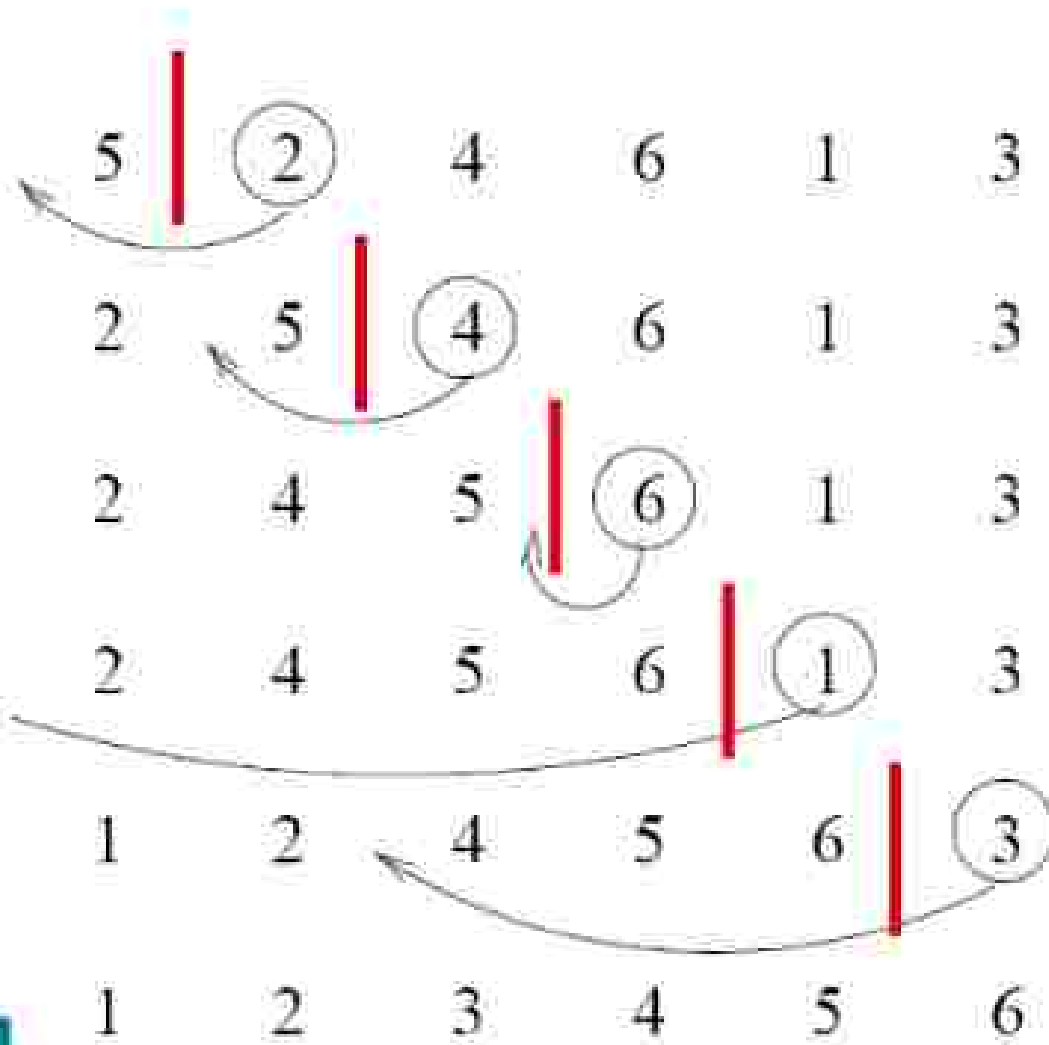
at each iteration, the array is divided in two sub-arrays:

left sub-array

right sub-array



# Insertion Sort



# INSERTION-SORT

*Alg.:* INSERTION-SORT( $A$ )

for  $j \leftarrow 2$  to  $n$

do  $key \leftarrow A[j]$

Insert  $A[j]$  into the sorted sequence  $A[1 \dots j-1]$

$i \leftarrow j - 1$

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

do  $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$



- ▶ Insertion sort - sorts the elements in place

# Analysis of Insertion Sort

INSERTION-SORT(A)

cost times

for  $j \leftarrow 2$  to  $n$

$c_1$   $n$

do  $key \leftarrow A[j]$

$c_2$   $n-1$

▷ Insert  $A[j]$  into the sorted sequence  $A[1] \dots [j-1]$

$\theta(i)$   $n-1$

$i \leftarrow j - 1$

$c_4$   $n-1$

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

$c_5$   $\sum_{j=2}^n t_j$

do  $A[i + 1] \leftarrow A[i]$

$c_6$   $\sum_{j=2}^n (t_j - 1)$

$i \leftarrow i - 1$

$c_7$   $\sum_{j=2}^n (t_j - 1)$

$A[i + 1] \leftarrow key$

$c_8$   $n-1$

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

$$T(n) = c_1(n-1) + c_2(n-1) + c_3 \sum_{j=2}^n t_j + c_4 \sum_{j=2}^n (t_j - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

# Selection Sort

- ▶ Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

- ▶ Disadvantage:

- Running time depends only slightly on the amount of order in the file

# Example

8	4	6	9	2	3	1
---	---	---	---	---	---	---

1	4	6	9	2	3	8
---	---	---	---	---	---	---

1	2	6	9	4	3	8
---	---	---	---	---	---	---

1	2	3	9	4	6	8
---	---	---	---	---	---	---

1	2	3	4	9	6	8
---	---	---	---	---	---	---

1	2	3	4	6	9	8
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---



# Selection Sort

*Alg.:* SELECTION-SORT( $A$ )

$n \leftarrow \text{length}[A]$

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

  do  $\text{smallest} \leftarrow j$

    for  $i \leftarrow j + 1$  to  $n$

      do if  $A[i] < A[\text{smallest}]$

        then  $\text{smallest} \leftarrow i$

    exchange  $A[j] \leftrightarrow A[\text{smallest}]$

8	4	6	9	2	3	1
---	---	---	---	---	---	---

# Analysis of Selection Sort

*Alg.*: SELECTION-SORT(*A*)

$n \leftarrow \text{length}[A]$

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

do smallest  $\leftarrow j$

$\approx n^2/2$  for  $i \leftarrow j + 1$  to  $n$

comparisons

do if  $A[i] < A[\text{smallest}]$

then smallest  $\leftarrow i$

$\approx n$

exchanges

exchange  $A[j] \leftrightarrow A[\text{smallest}]$

cost

times

$c_1$

1

$c_2$

$n$

$c_3$

$n-1$

$c_4$

$\sum_{j=1}^{n-1} (n-j+1)$

$c_5$

$\sum_{j=1}^{n-1} (n-j)$

$c_6$

$\sum_{j=1}^{n-1} (n-j)$

$c_7$

$n-1$

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

# Sorting

## ▶ Insertion sort

- Design approach: incremental
- Sorts in place: Yes
- Best case:  $\Theta(n)$
- Worst case:  $\Theta(n^2)$

## ▶ Bubble Sort

- Design approach: incremental
- Sorts in place: Yes
- Running time:  $\Theta(n^2)$

# Sorting

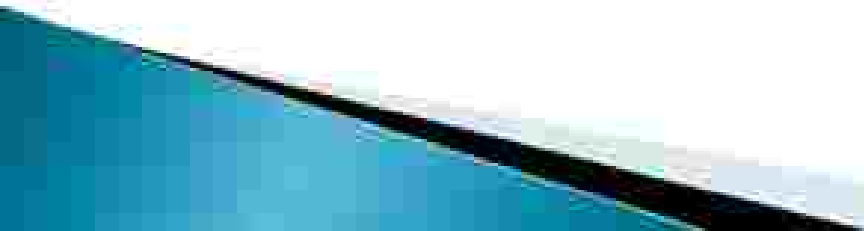
## ▶ Selection sort

- Design approach: incremental
- Sorts in place: Yes
- Running time:  $\Theta(n^2)$

## ▶ Merge Sort

- Design approach: divide and conquer
- Sorts in place: No
- Running time:

# Bucket Sort

- ▶ Bucket sort works by partitioning the elements into buckets and then return the result
  - ▶ Buckets are assigned based on each element's search key
  - ▶ To return the result, concatenate each bucket and return as a single array
- 

# Bucket Sort

- ▶ Some variations

- Make enough buckets so that each will only hold one element, use a count for duplicates
- Use fewer buckets and then sort the contents of each bucket

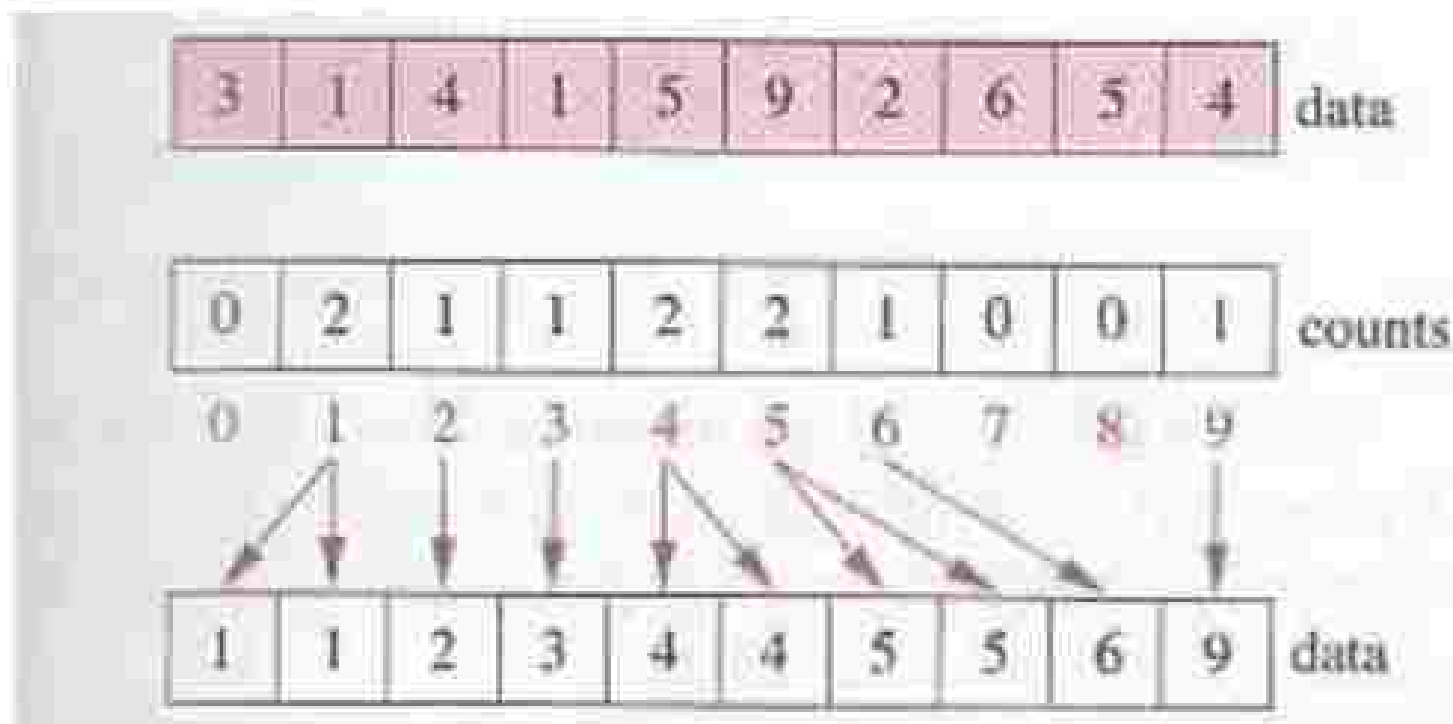
- ▶ The more buckets you use, the faster the algorithm will run but it uses more memory

# Bucket Sort

- ▶ Time complexity is reduced when the number of items per bucket is evenly distributed and as close to 1 per bucket as possible
- ▶ Buckets require extra space, so we are trading increased space consumption for a lower time complexity
- ▶ In fact Bucket Sort beats all other sorting routines in time complexity but can require a lot of space

# Bucket Sort

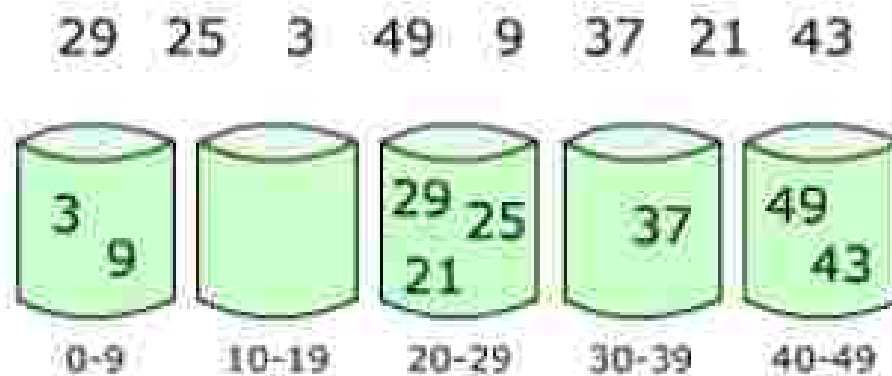
- ▶ One value per bucket:





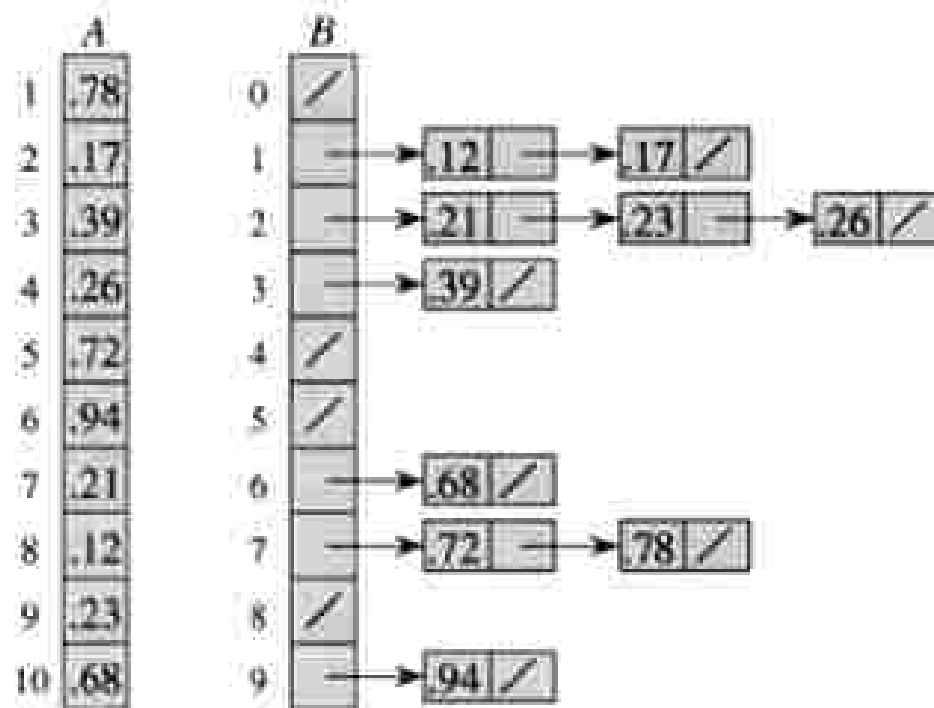
# Bucket Sort

Multiple items per bucket:



# Bucket Sort

In array form:



# 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 of the sub-problems
  - Obtain the solution for the original problem

# Merge Sort Approach

- ▶ To sort an array  $A[p \dots r]$ :
- ▶ **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( $A, p, r$ )

if  $p < r$

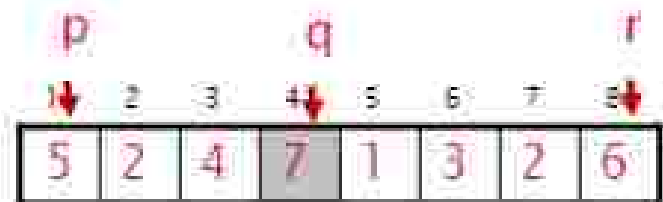
then  $q \leftarrow \lfloor (p + r) / 2 \rfloor$

MERGE-SORT( $A, p, q$ )

MERGE-SORT( $A, q + 1, r$ )

MERGE( $A, p, q, r$ )

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



▷ Check for base case

▷ Divide

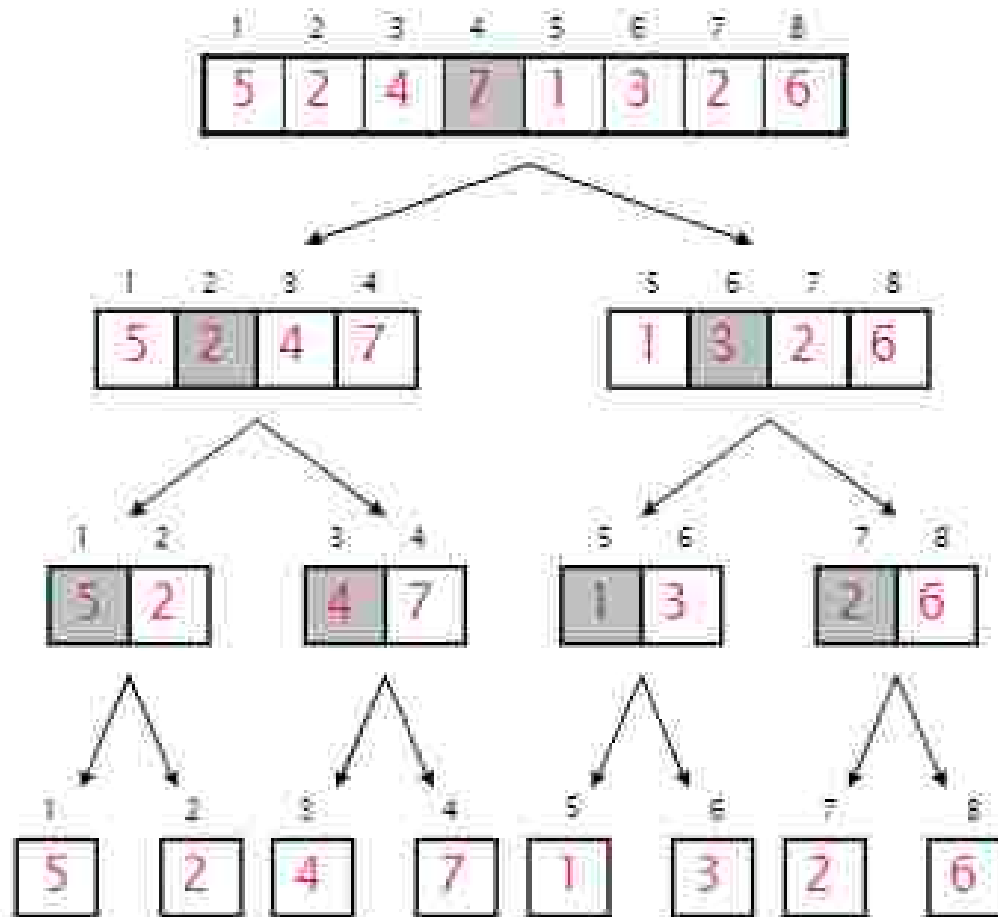
Conquer

▷ Conquer

▷ Combine

# Example - n Power of 2

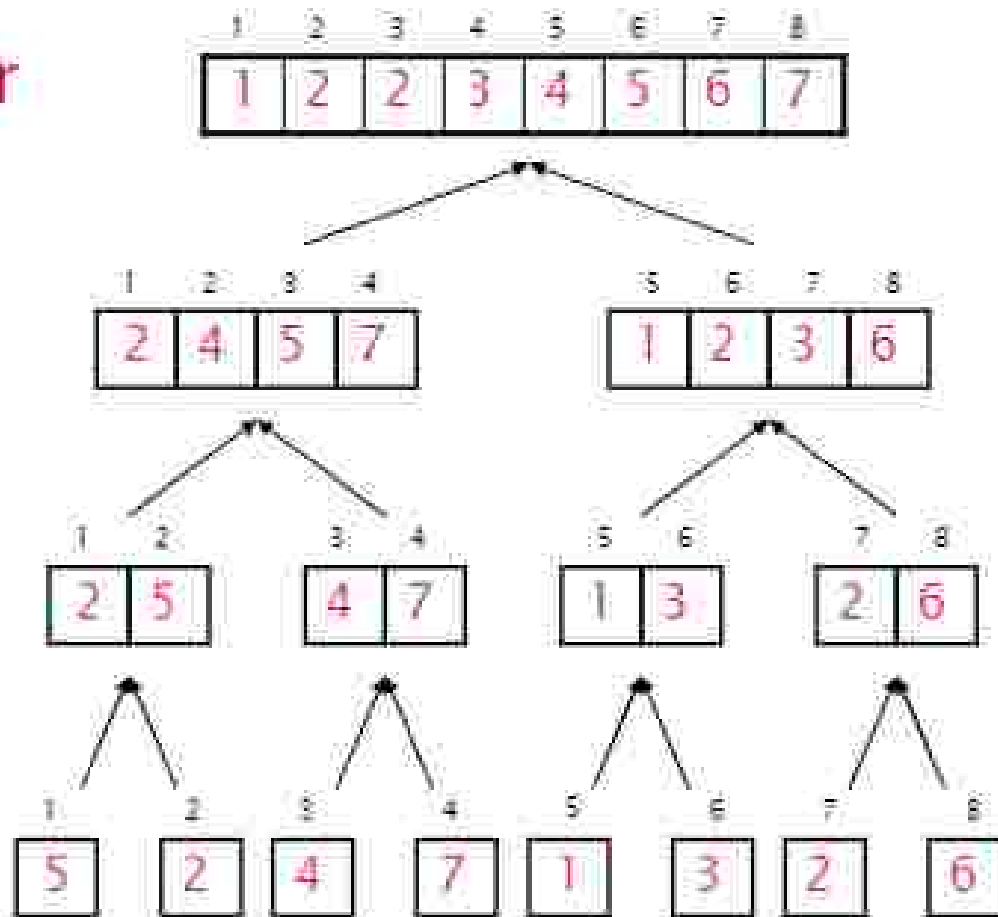
Divide



$q = 4$

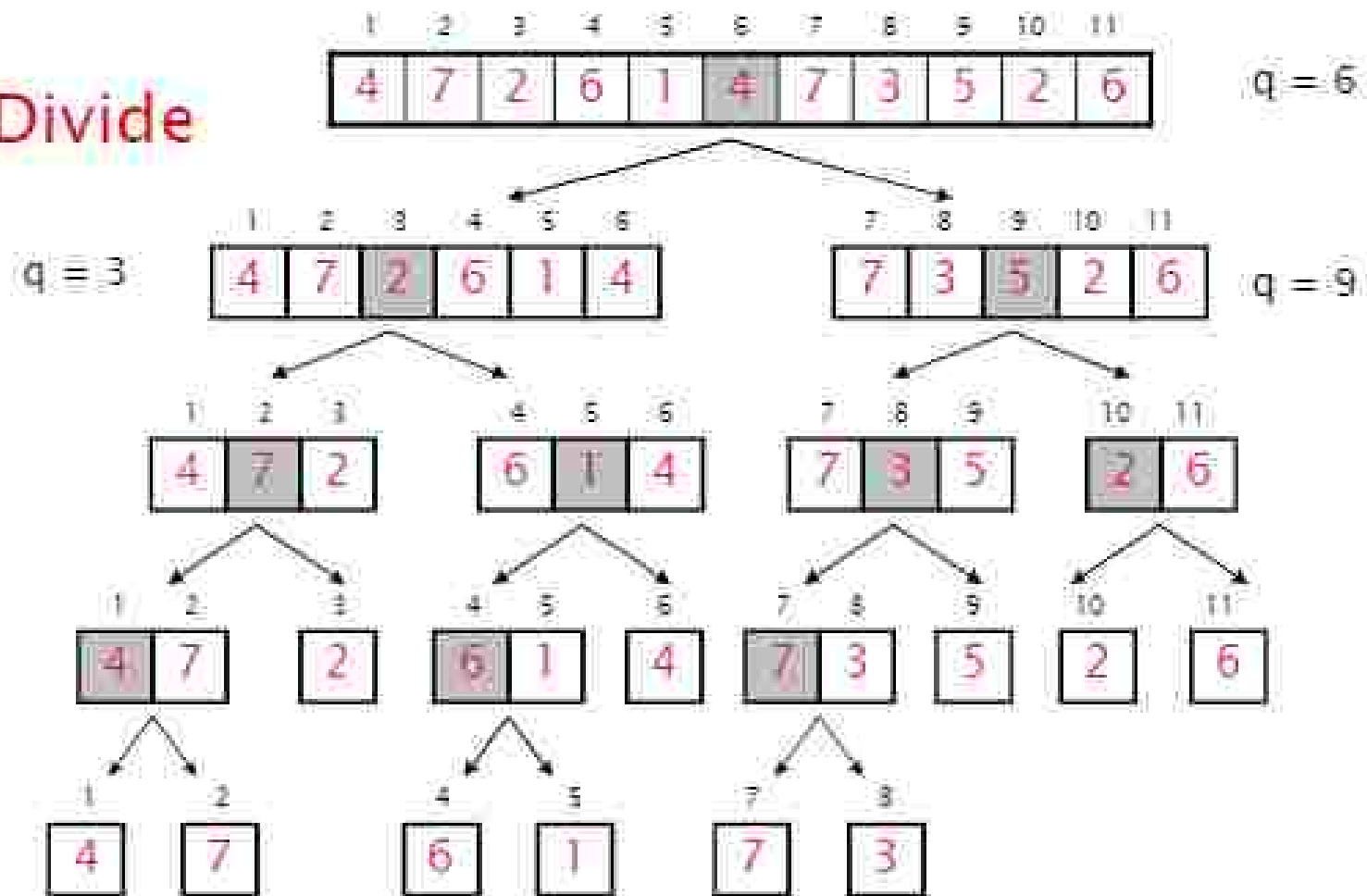
# Example - n Power of 2

Conquer  
and  
Merge



# Example - n Not a Power of 2

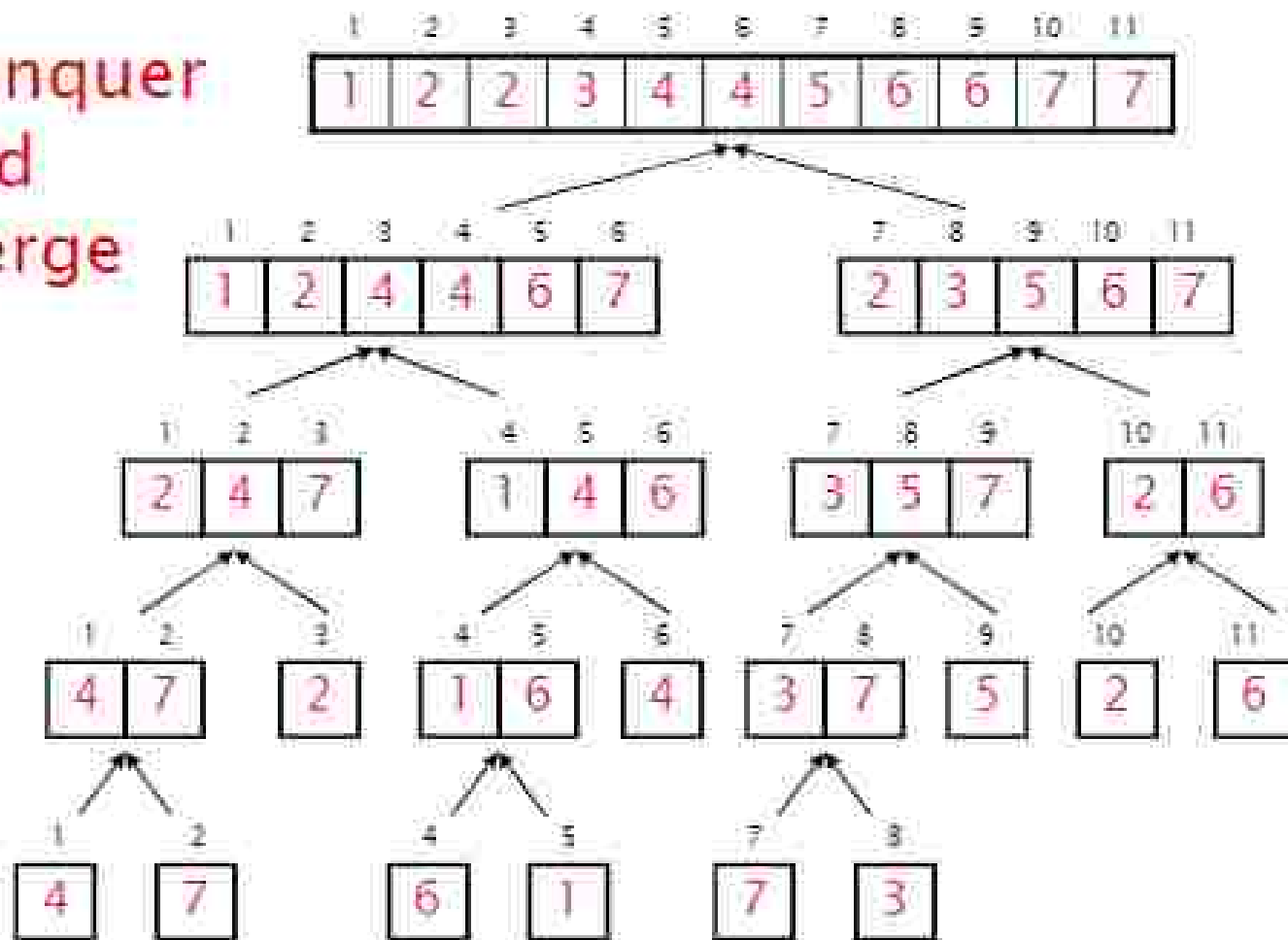
Divide



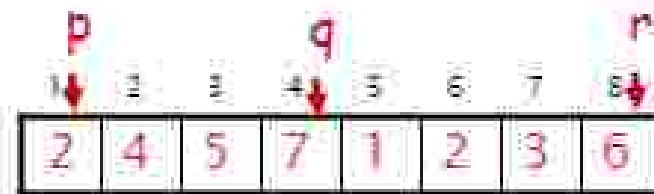


# Example - n Not a Power of 2

Conquer  
and  
Merge



# Merging



- ▶ **Input:** Array  $A$  and indices  $p, q, r$  such that  $p \leq q < r$ 
  - 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



$A1 \leftarrow A[p, q]$



$A2 \leftarrow A[q+1, r]$



choose the smaller  
element from the subarrays

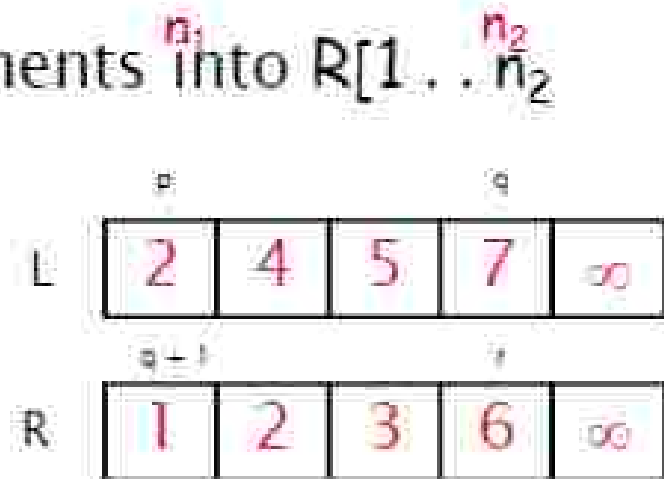
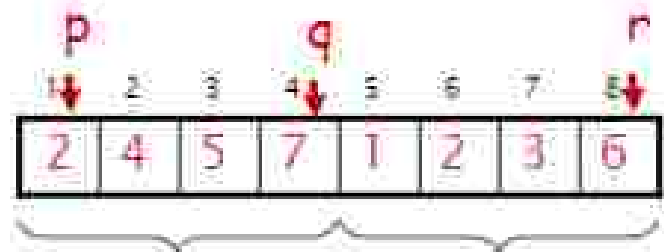
$A[p, r]$



# Merge – Pseudocode

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

1. Compute  $n_1$  and  $n_2$
2. Copy the first  $n_1$  elements into  $L[1 \dots n_1 + 1]$  and the next  $n_2$  elements into  $R[1 \dots n_2 + 1]$
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  $L[i] \leq R[j]$
7.         then  $A[k] \leftarrow L[i]$
8.              $i \leftarrow i + 1$
9.         else  $A[k] \leftarrow R[j]$
10.              $j \leftarrow j + 1$



```

void merge(int a[], int low, int
high, int mid)
{
    int i,j,k,c[max];
    i=low;
    j=mid+1;
    k=0;
    while(i<=mid) && (j<=high)
    {
        if(a[i]<a[j])
            c[k]=a[i++];
        else
            c[k]=a[j++];
        k++;
    }
    while(i<=mid)
        c[k++]=a[i++];
    while(j<=high)
        c[k++]=a[j++];
    for(i=low,j=0;i<=high;i++,j++)
    {
        a[i]=c[j];
    }
}

```

```

void mergesort(int a[], int low, int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,high,mid);
    }
}

```

# Quicksort Algorithm

Given an array of  $n$  elements (e.g., integers):

- ▶ If array only contains one element, return
- ▶ Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

# Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

# Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:





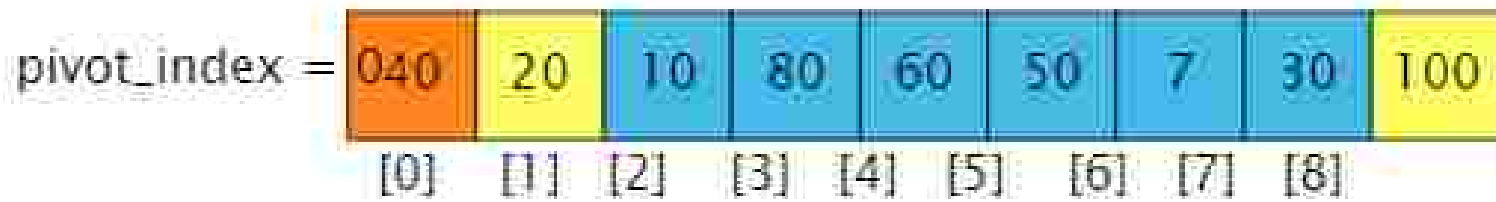
# Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements  $\geq$  pivot
2. Another sub-array that contains elements  $<$  pivot

The sub-arrays are stored in the original data array.

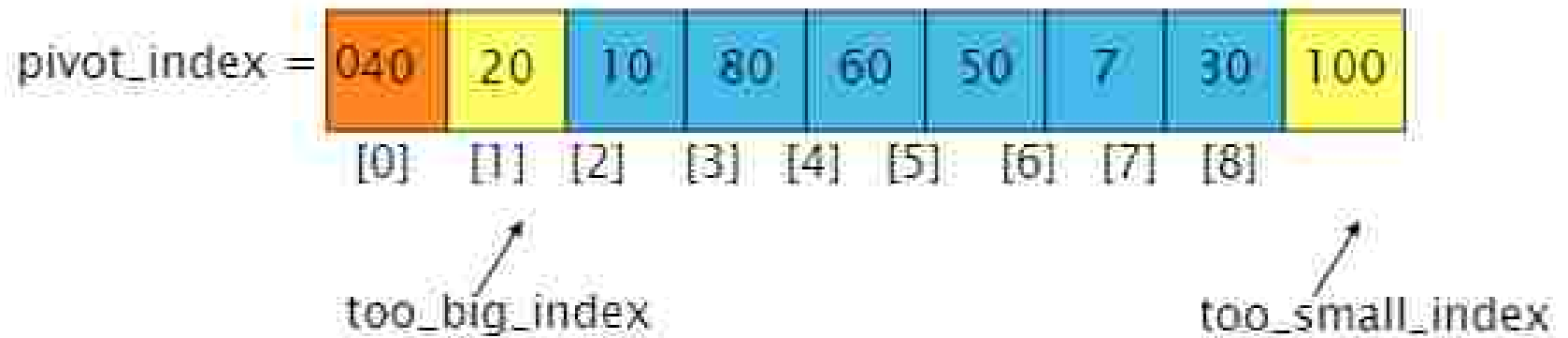
Partitioning loops through, swapping elements below/above pivot.



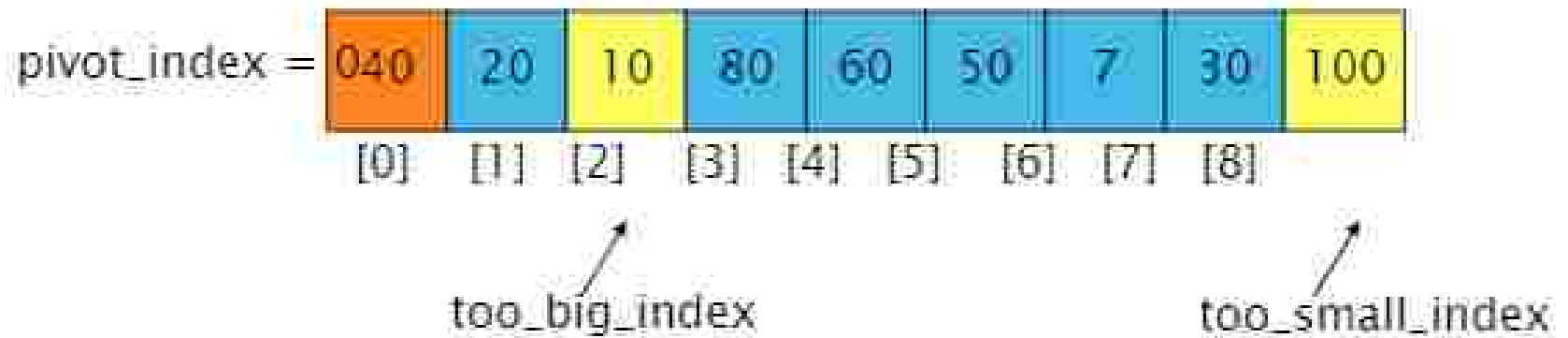
too\_big\_index

too\_small\_index

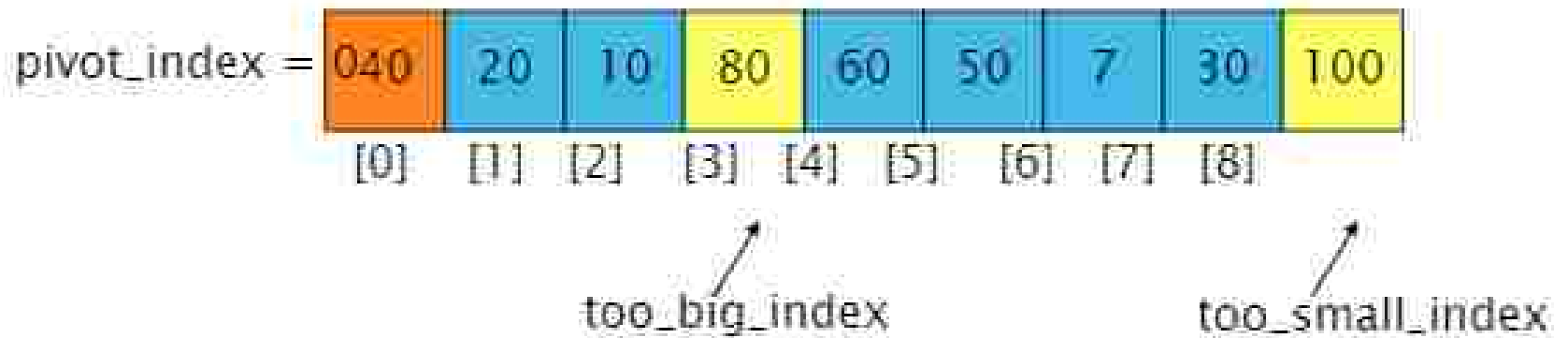
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



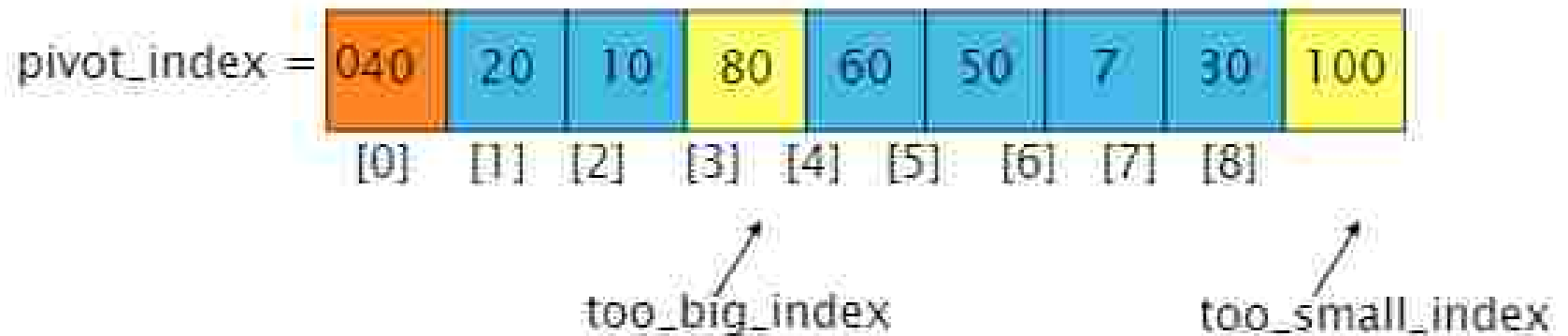
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



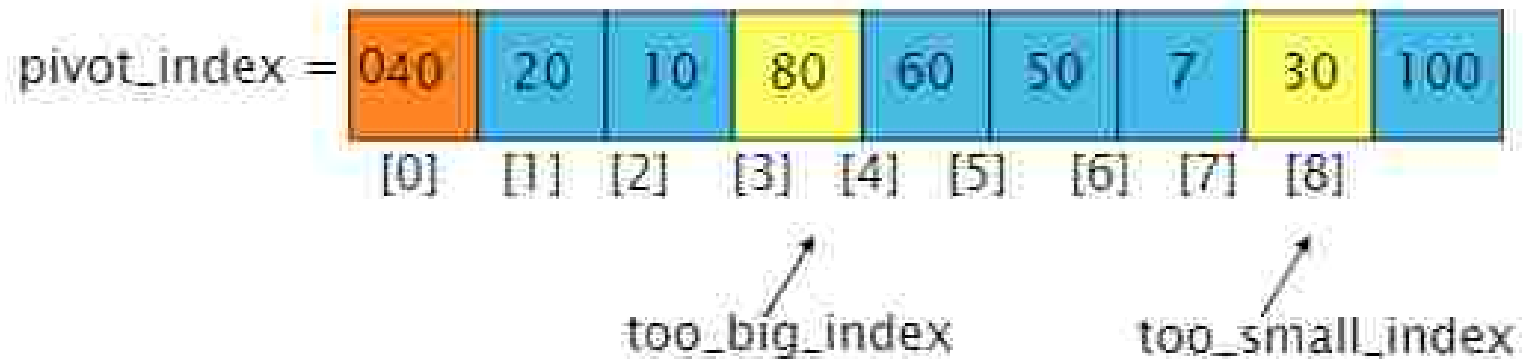
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



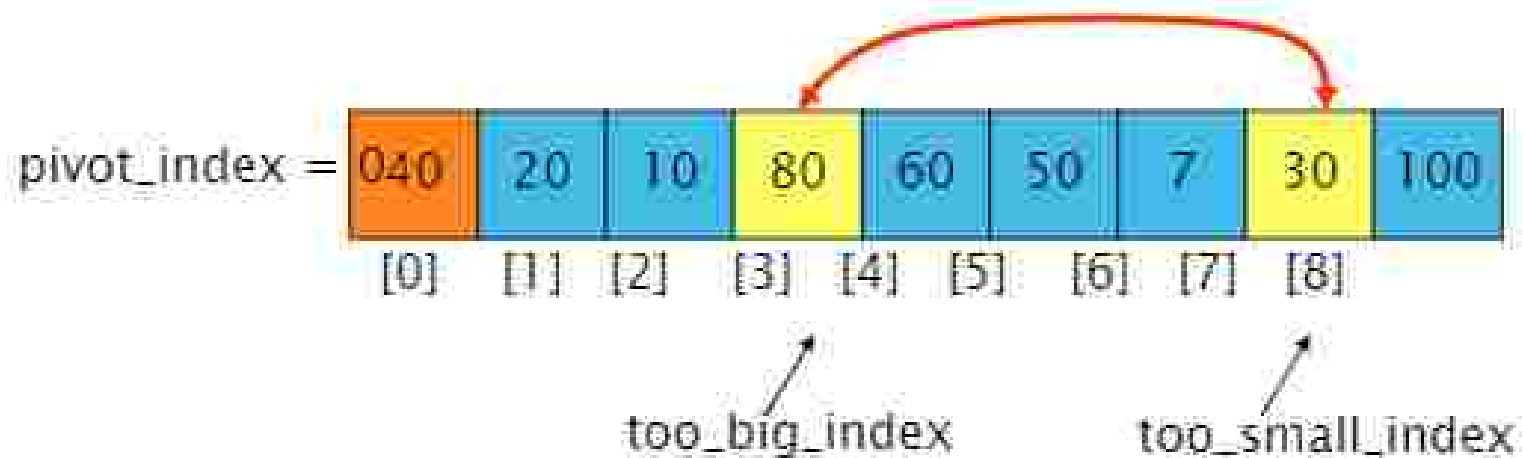
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$

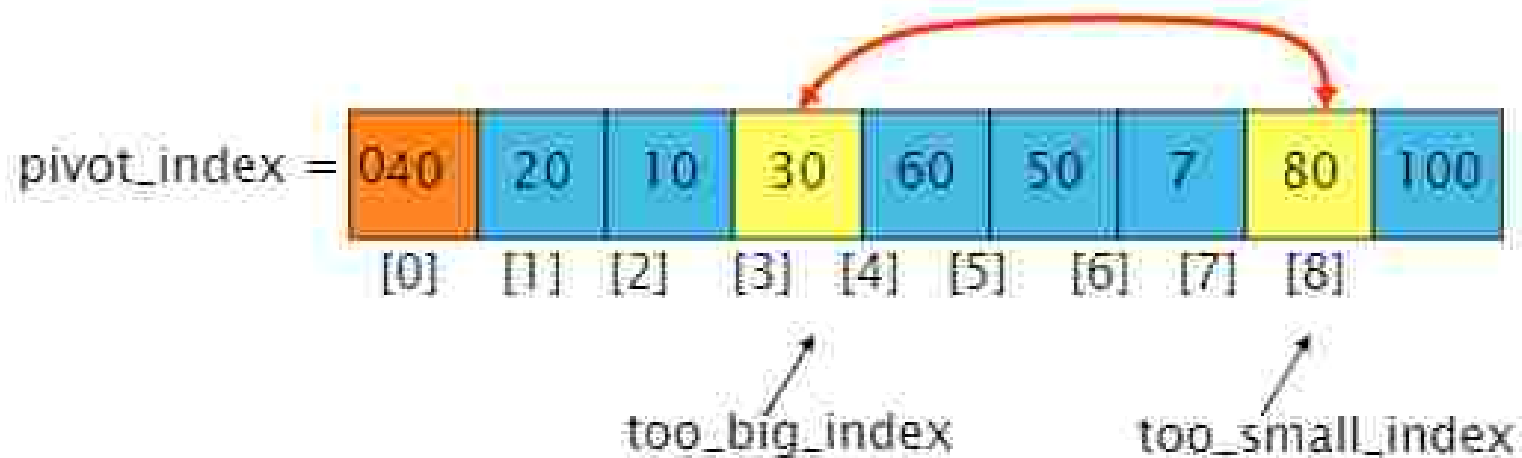


1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $--\text{too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$

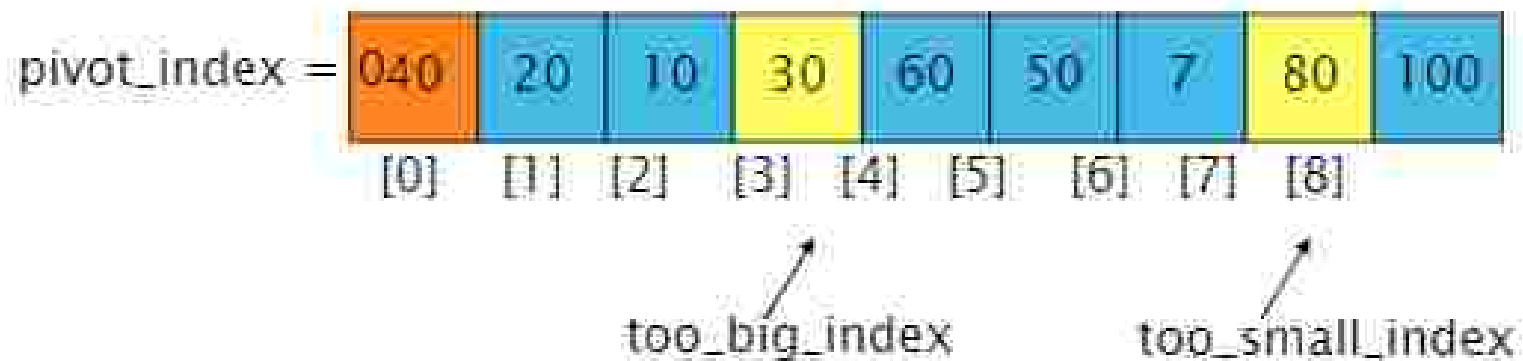




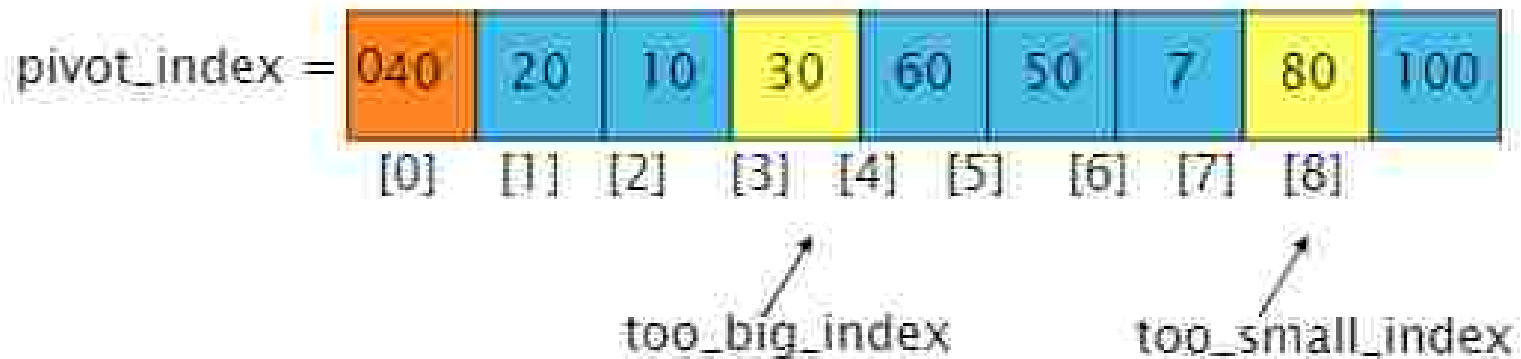
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$



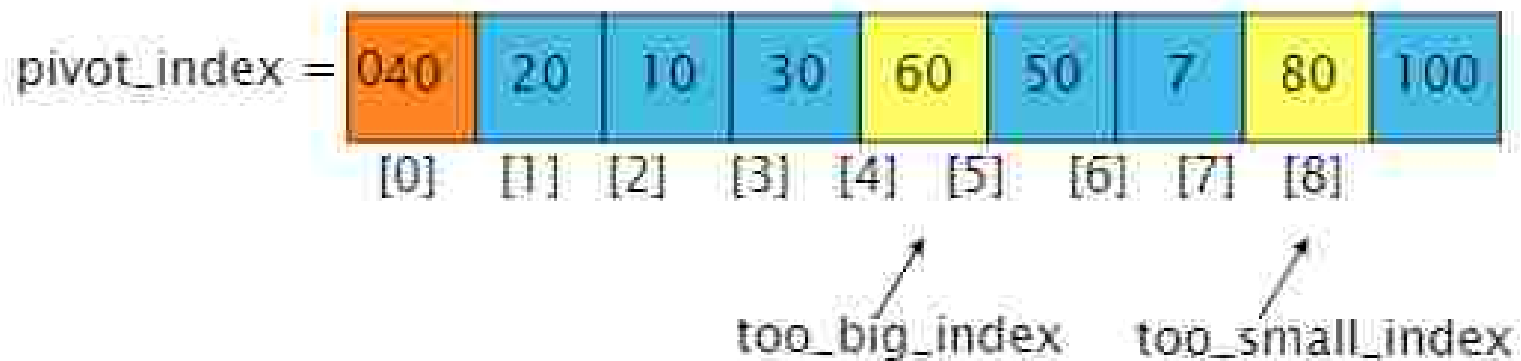
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



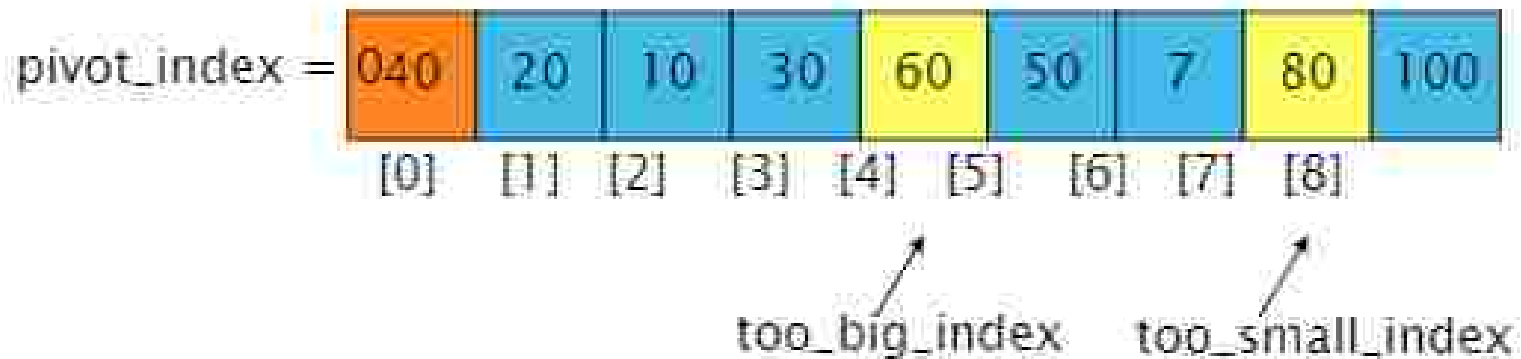
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



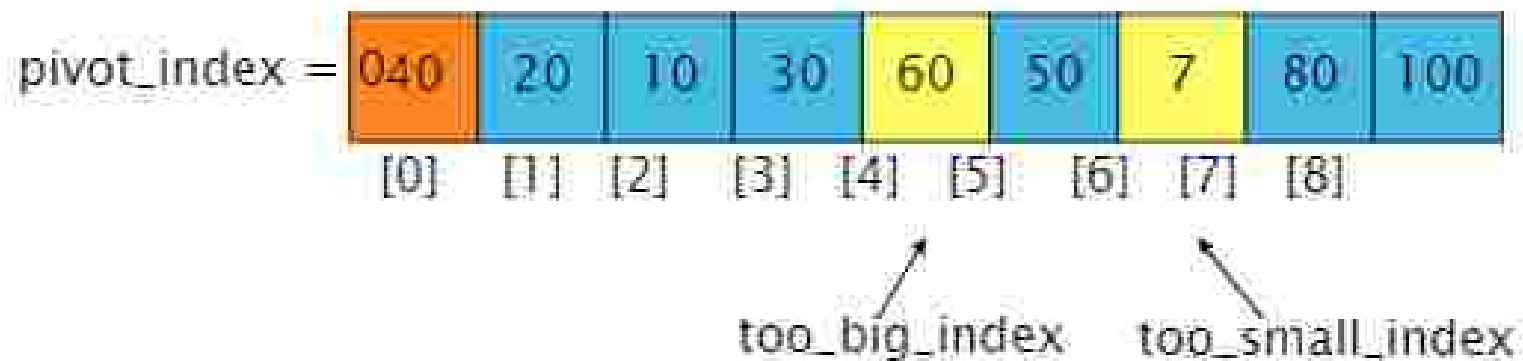
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



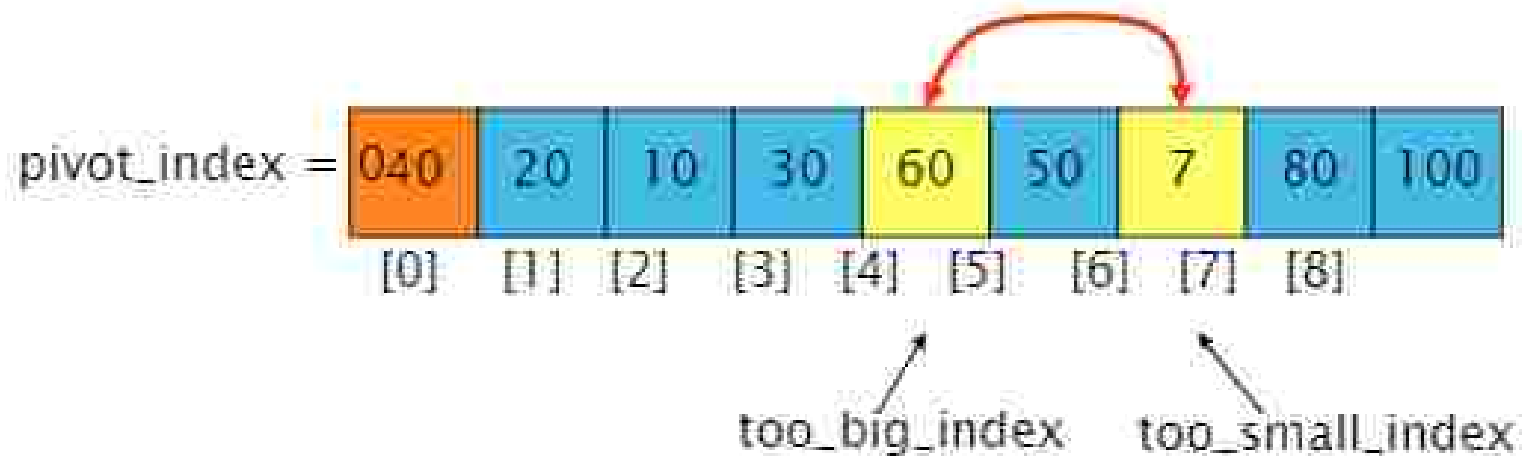
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



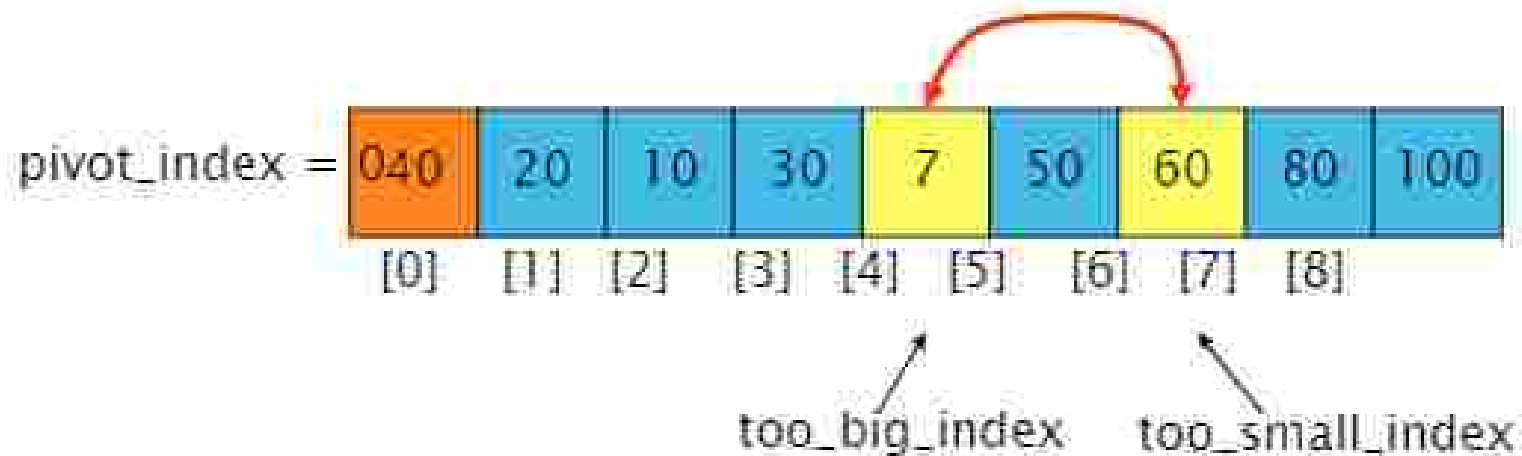
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $--\text{too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

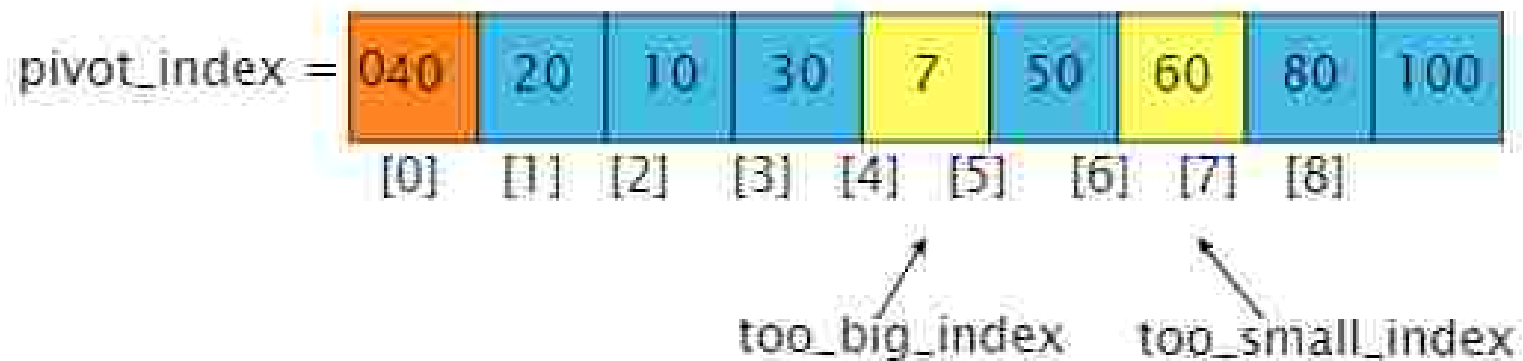


1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

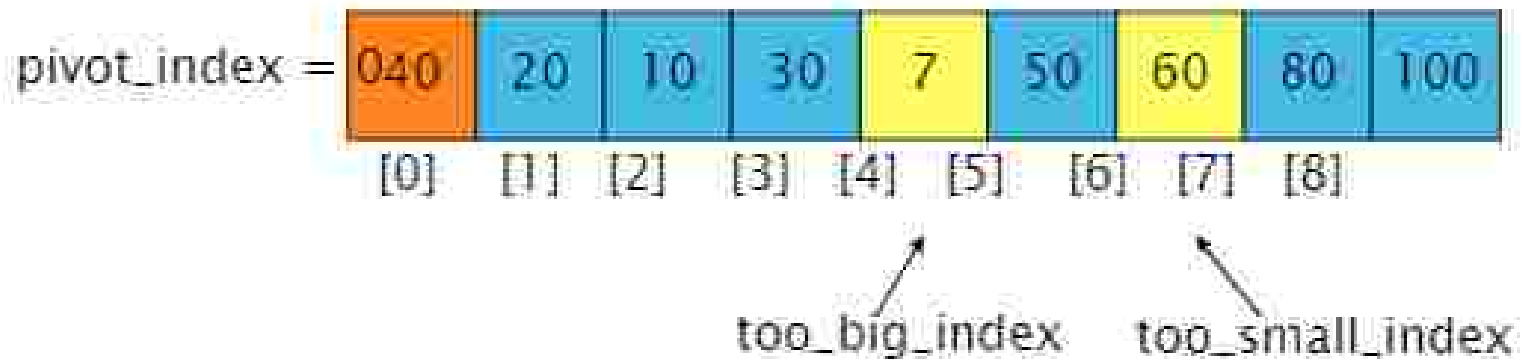




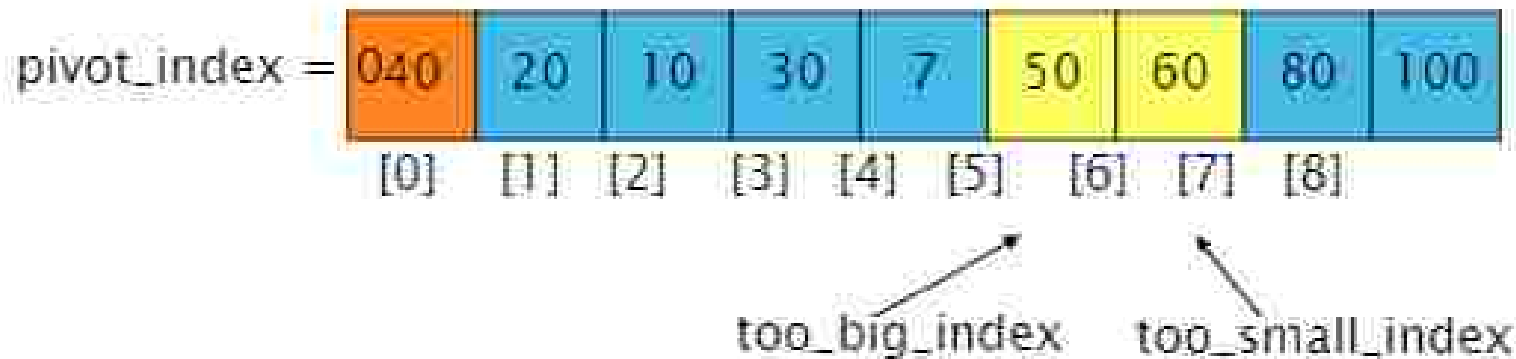
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $--\text{too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



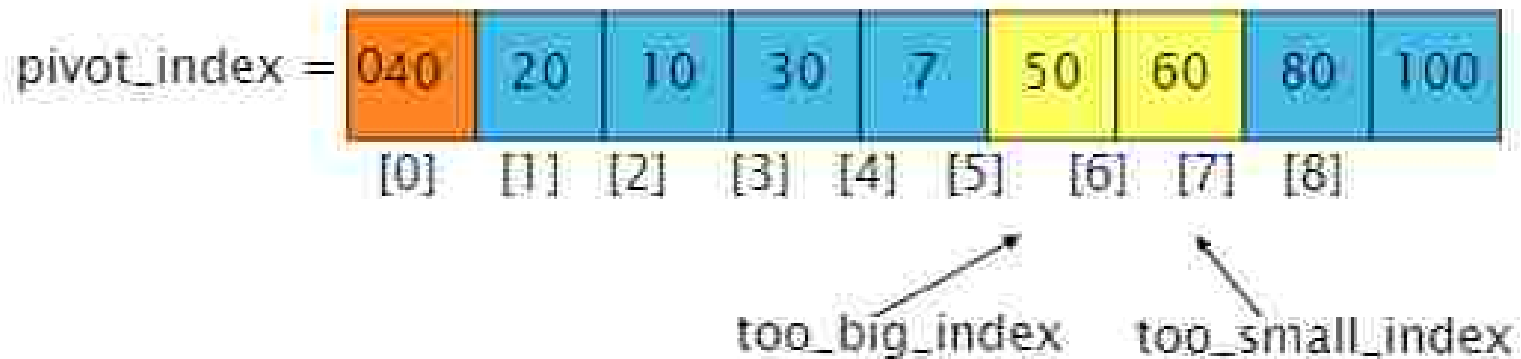
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



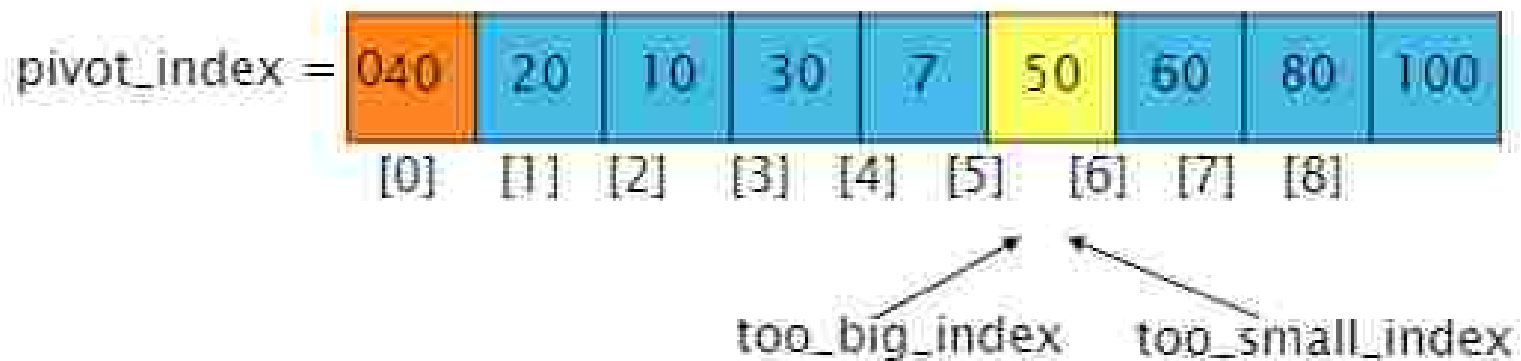
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



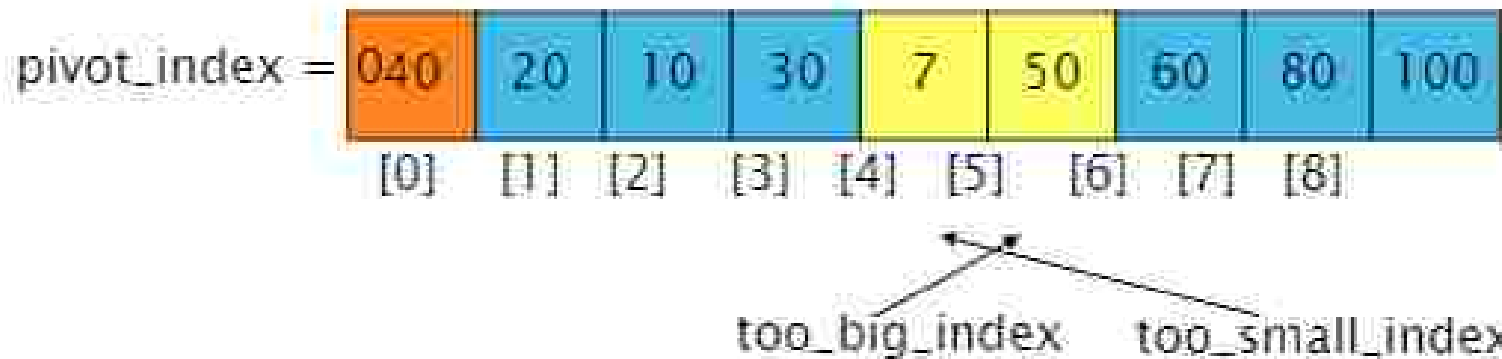
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



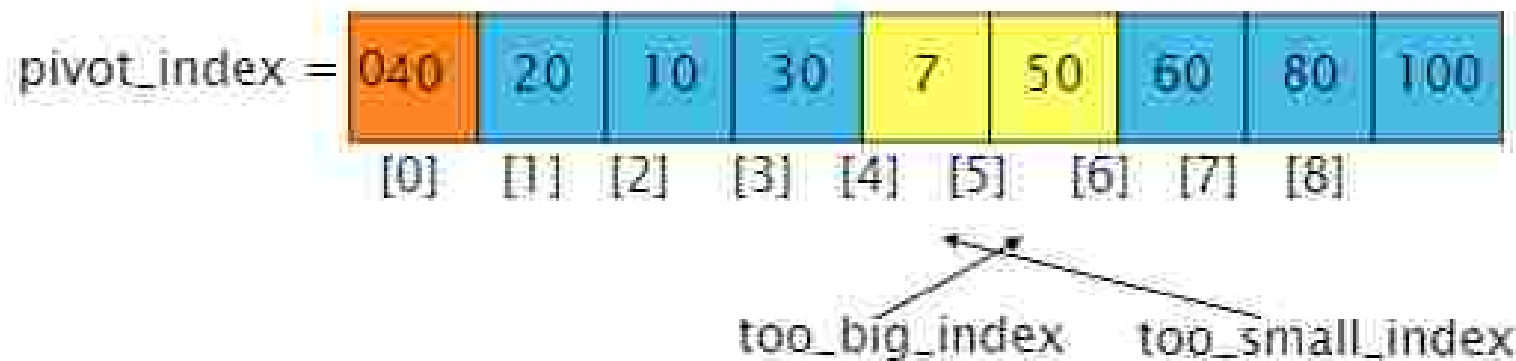
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



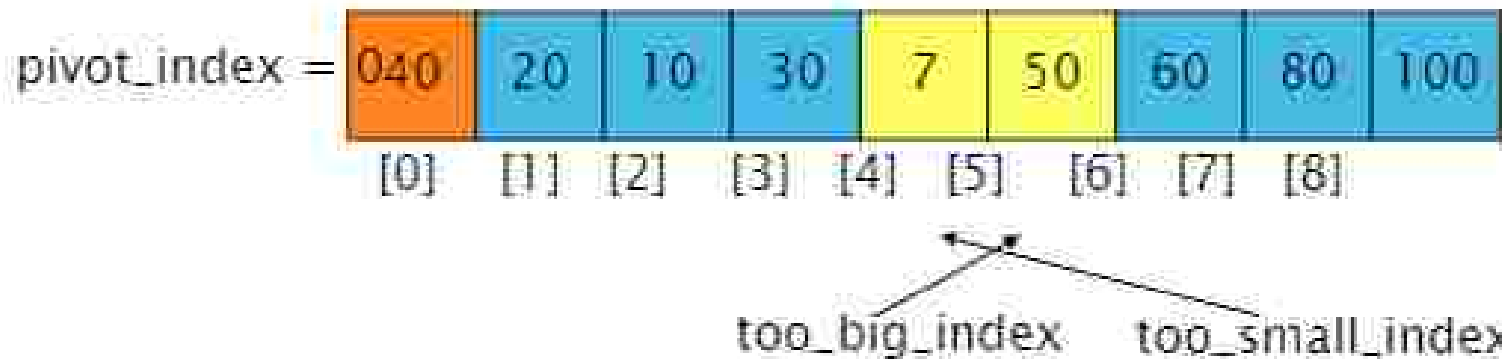
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

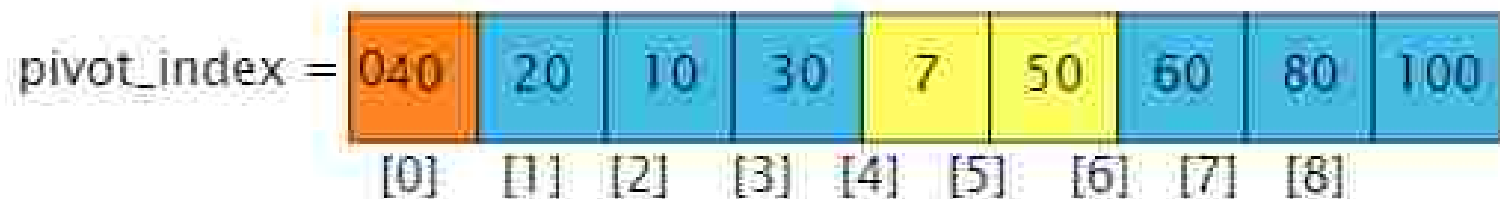


1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

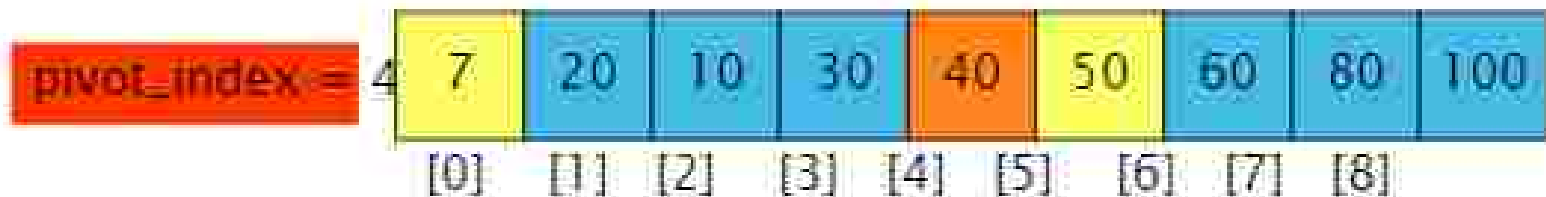




1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



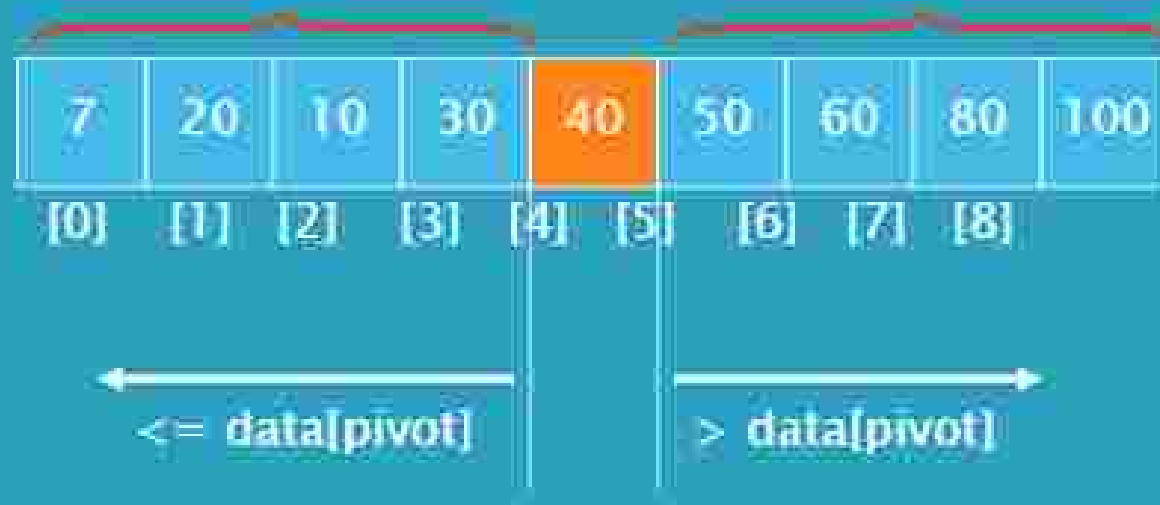
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Partition Result

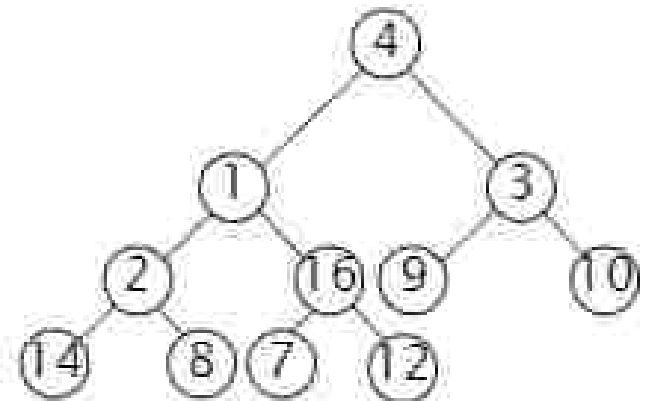


# Recursion: Quicksort Sub-arrays



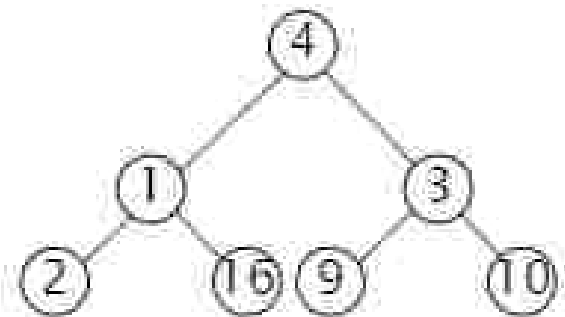
# Special Types of Trees

- ▶ *Def.* Full binary tree = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

- ▶ *Def.* Complete binary tree = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



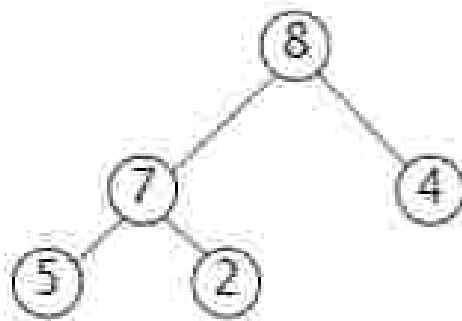
Complete binary tree

# The Heap Data Structure

▶ *Def.* A heap is a nearly complete binary tree with the following two properties:

- **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
- **Order (heap) property:** for any node  $x$

$$\text{Parent}(x) \geq x$$



Heap

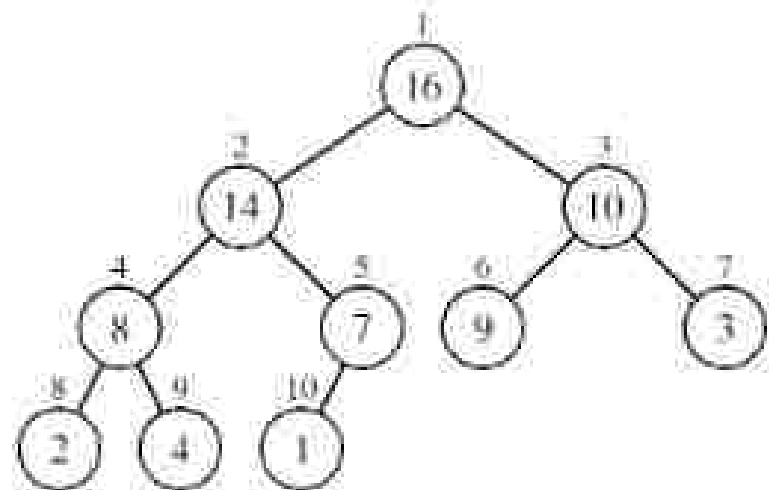
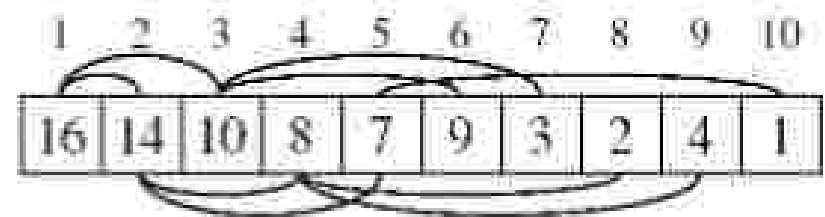
From the heap property, it follows that:

**“The root is the maximum element of the heap!”**

**A heap is a binary tree that is filled in order**

# Array Representation of Heaps

- ▶ A heap can be stored as an array  $A$ .
  - Root of tree is  $A[1]$
  - Left child of  $A[i] = A[2i]$
  - Right child of  $A[i] = A[2i + 1]$
  - Parent of  $A[i] = A[\lfloor i/2 \rfloor]$
  - $\text{Heapsize}[A] \leq \text{length}[A]$
- ▶ The elements in the subarray  $A[(\lfloor n/2 \rfloor + 1) .. n]$  are leaves



# Heap Types

- ▶ **Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes  $i$ , excluding the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

- ▶ **Min-heaps** (smallest element at root), have the *min-heap property*:

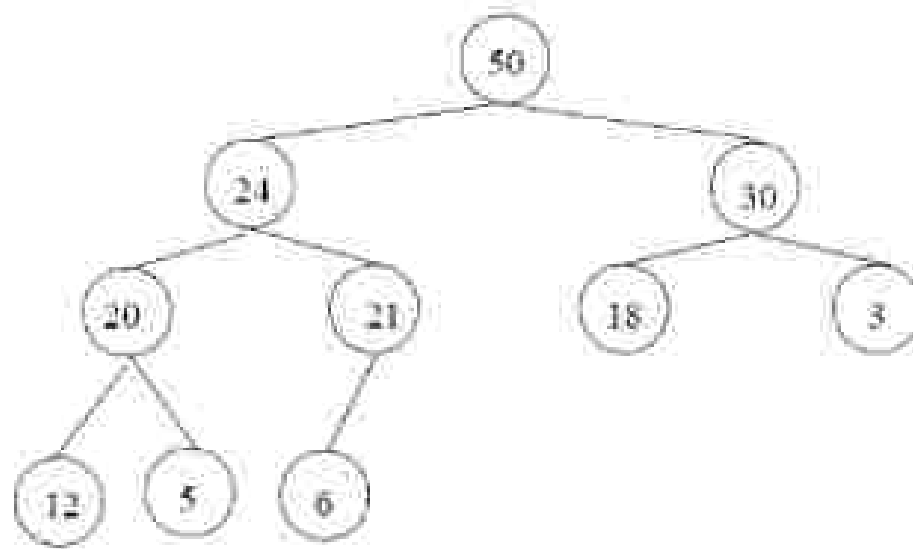
- for all nodes  $i$ , excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$



# Adding / Deleting Nodes

- ▶ New nodes are always inserted at the bottom level (left to right)
- ▶ Nodes are removed from the bottom level (right to left)

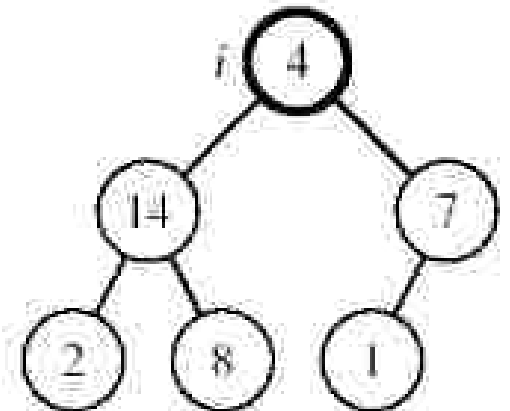


# Operations on Heaps

- ▶ Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- ▶ Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- ▶ Sort an array in place
  - HEAPSORT
- ▶ Priority queues

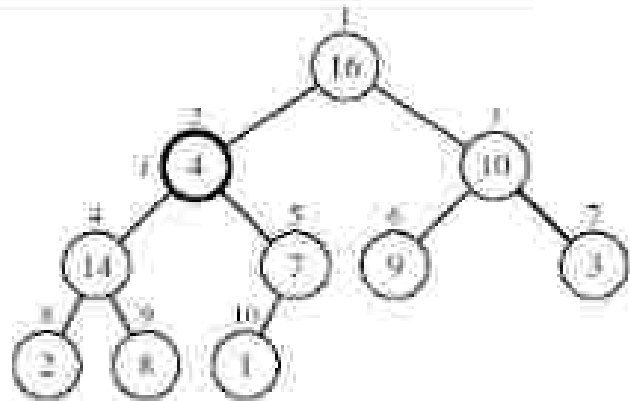
# Maintaining the Heap Property

- ▶ Suppose a node is smaller than a child
  - Left and Right subtrees of  $i$  are max-heaps
- ▶ To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



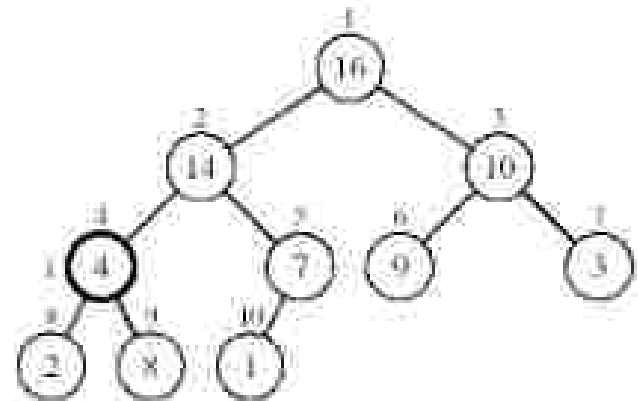
# Example

MAX-HEAPIFY(A, 2, 10)



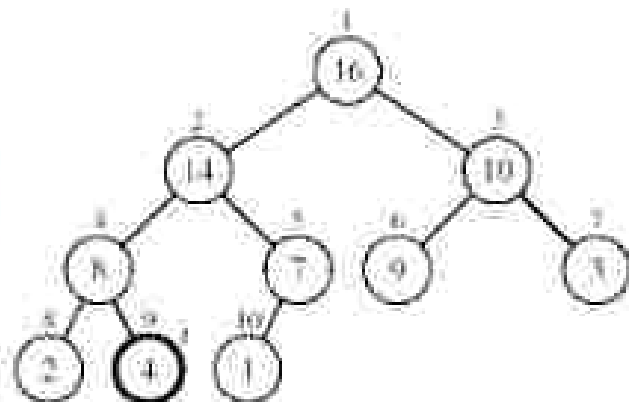
A[2] violates the heap property

$A[2] \leftrightarrow A[4]$



A[4] violates the heap property

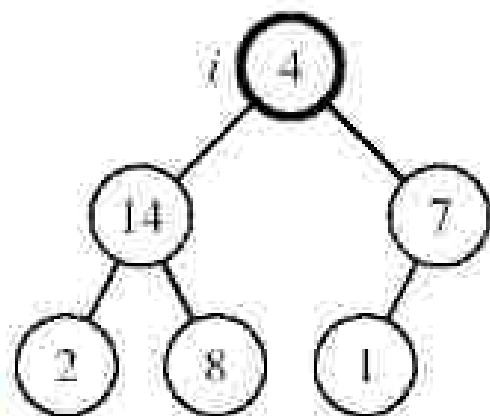
$A[4] \leftrightarrow A[9]$



Heap property restored

# Maintaining the Heap Property

- ▶ Assumptions:
  - Left and Right subtrees of  $i$  are max-heaps
  - $A[i]$  may be smaller than its children



*Alg:* MAX-HEAPIFY( $A, i, n$ )

1.  $l \leftarrow \text{LEFT}(i)$
2.  $r \leftarrow \text{RIGHT}(i)$
3. if  $l \leq n$  and  $A[l] > A[i]$
4.     then  $\text{largest} \leftarrow l$
5.     else  $\text{largest} \leftarrow i$
6. if  $r \leq n$  and  $A[r] > A[\text{largest}]$
7.     then  $\text{largest} \leftarrow r$
8. if  $\text{largest} \neq i$
9.     then exchange  $A[i] \leftrightarrow A[\text{largest}]$
10.         MAX-HEAPIFY( $A, \text{largest}, n$ )

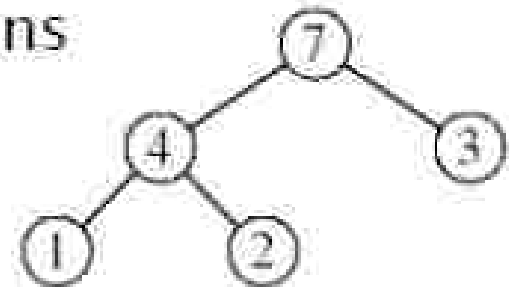
# Heapsort

## ▶ Goal:

- Sort an array using heap representations

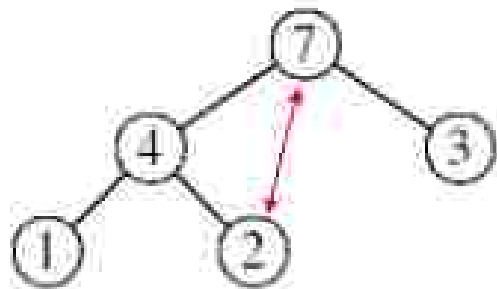
## ▶ Idea:

- Build a **max-heap** from the array
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Call **MAX-HEAPIFY** on the new root
- Repeat this process until only one node remains

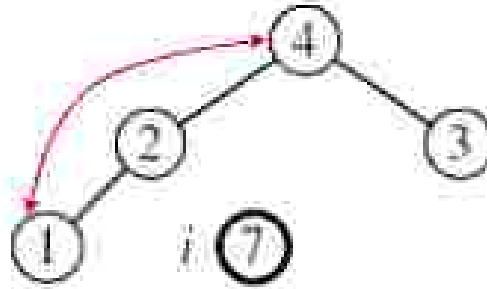


# Example:

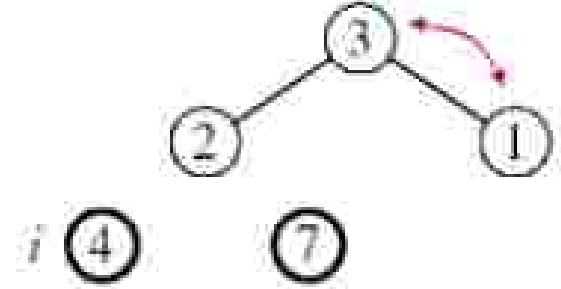
# $A = [7, 4, 3, 1, 2]$



MAX-HEAPIFY(A, 1, 4)



MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)

A 

1	2	3	4	7
---	---	---	---	---

# Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)  $O(n)$
  2. for  $i \leftarrow \text{length}[A]$  downto 2
  3.     do exchange  $A[1] \leftrightarrow A[i]$
  4.     MAX-HEAPIFY(A, 1,  $i - 1$ )  $O(\lg n)$
- $\left. \begin{array}{l} O(n) \\ O(\lg n) \end{array} \right\} n-1 \text{ times}$

- ▶ Running time:  $O(n \lg n)$  --- Can be shown to be  $\Theta(n \lg n)$





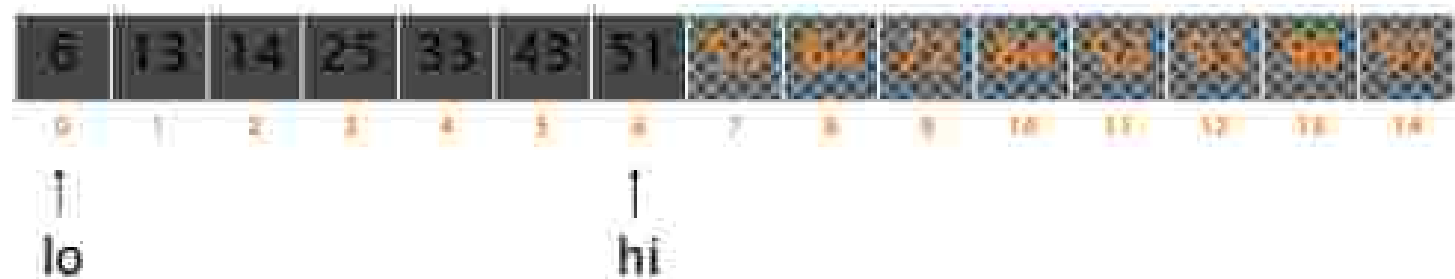
# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains `a[lo] ≤ value ≤ a[hi]`.
- ▶ Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi

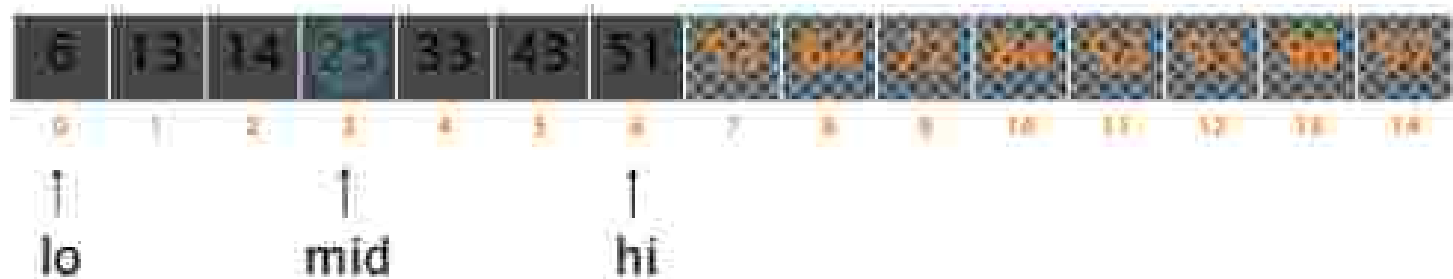
# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] == value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains  $a[lo] \leq value \leq a[hi]$ .
- ▶ Ex. Binary search for 33.



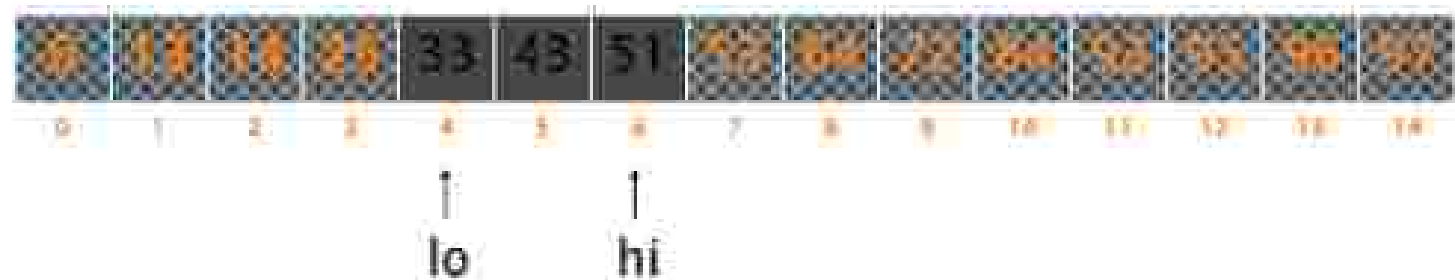
# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains  $a[lo] \leq value \leq a[hi]$ .
- ▶ Ex. Binary search for 33.



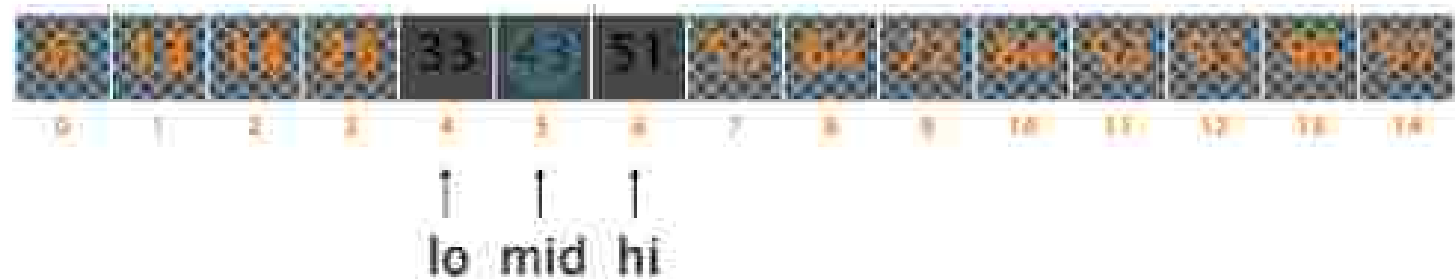
# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains `a[lo] ≤ value ≤ a[hi]`.
- ▶ Ex. Binary search for 33.



# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains  $a[lo] \leq value \leq a[hi]$ .
- ▶ Ex. Binary search for 33.



# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains  $a[lo] \leq value \leq a[hi]$ .
- ▶ Ex. Binary search for 33.







# Binary Search

- ▶ Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.
- ▶ Invariant. Algorithm maintains  $a[lo] \leq value \leq a[hi]$ .
- ▶ Ex. Binary search for 33.



```
low = 0;
high = length - 1;

while (low <= high) {
    mid = (low + high) / 2;
    if (a[mid] < target) {
        low = mid + 1;
    } else if (a[mid] > target) {
        high = mid - 1;
    } else {
        return mid;    // target found
    }
}
```

# Fibonacci Search

## Similarities with Binary Search:

- ▶ Works for sorted arrays
- ▶ A Divide and Conquer Algorithm.
- ▶ Has  $\log n$  time complexity.

## Differences with Binary Search:

- ▶ Fibonacci Search divides given array in unequal parts
- ▶ Binary Search uses division operator to divide range. Fibonacci Search doesn't use  $/$ , but uses  $+$  and  $-$ . The division operator may be costly on some CPUs.
- ▶ Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

- ▶ Fibonacci Numbers are recursively defined as  $F(n) = F(n-1) + F(n-2)$ ,  $F(0) = 0$ ,  $F(1) = 1$ .  
First few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- ▶ Below observation is used for range elimination, and hence for the  $O(\log(n))$  complexity.
- ▶  $F(n - 2) \approx (1/3)*F(n)$  and  $F(n - 1) \approx (2/3)*F(n)$ .

# Algorithm

- ▶ Let the searched element be  $x$ .
- ▶ The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of given array. Let the found Fibonacci number be  $\text{fib}$  ( $m$ 'th Fibonacci number). We use  $(m-2)$ 'th Fibonacci number as the index (If it is a valid index). Let  $(m-2)$ 'th Fibonacci Number be  $i$ , we compare  $\text{arr}[i]$  with  $x$ , if  $x$  is same, we return  $i$ . Else if  $x$  is greater, we recur for subarray after  $i$ , else we recur for subarray before  $i$ .

- ▶ Below is the complete algorithm  
Let  $arr[0..n-1]$  be the input array and element to be searched be  $x$ .
- ▶ Find the smallest Fibonacci Number greater than or equal to  $n$ . Let this number be  $fibM$  [ $m$ 'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be  $fibMm1$  [ $(m-1)$ 'th Fibonacci Number] and  $fibMm2$  [ $(m-2)$ 'th Fibonacci Number].
- ▶ While the array has elements to be inspected:
  - Compare  $x$  with the last element of the range covered by  $fibMm2$ .
  - If  $x$  matches, return index
  - Else If  $x$  is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
  - Else  $x$  is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
- ▶ Since there might be a single element remaining for comparison, check if  $fibMm1$  is 1. If Yes, compare  $x$  with that remaining element. If match, return index.

▶ Ex

$A = \{10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100\}$ ,

1	2	3	4	5	6	7	8	9	10	11
10	22	35	40	45	50	80	82	85	90	100

$X = 85$

$N = 11$

Fib = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

- ▶  $\text{Fib}(7) = 13 > 11$
- ▶  $(m-1) = 8, (m-2) = 5$

- ▶  $i = \min(\text{offset} + m2, n)$
- ▶ Offset-It marks the range that has been eliminated, starting from the front. We will

<i>fibMma</i>	<i>fibMmr</i>	<i>fibM</i>	<i>offset</i>	$i = \min(\text{offset} + \text{fib}(i), n)$	<i>arr[i]</i>	<i>Consequence</i>
5	8	13	0	5	45	Move one down, reset offset
3	5	8	5	8	82	Move one down, reset offset
2	3	5	8	10	90	Move two down
1	1	2	8	9	85	Return i



# Analysis of Algorithms



An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

# Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm
  - 1) Is it easy to implement, understand, modify?
  - 2) How long does it take to run it to completion?
  - 3) How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria
- In this course we are interested in the second and third criteria

# Comparing Algorithms

- ▶ **Time complexity**
  - = The amount of time that an algorithm needs to run to completion
- ▶ **Space complexity**
  - = The amount of memory an algorithm needs to run
- ▶ We will occasionally look at space complexity, but we are mostly interested in time complexity in this course
- ▶ Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

# How to Calculate Running time

- ▶ Most algorithms transform input objects into output objects



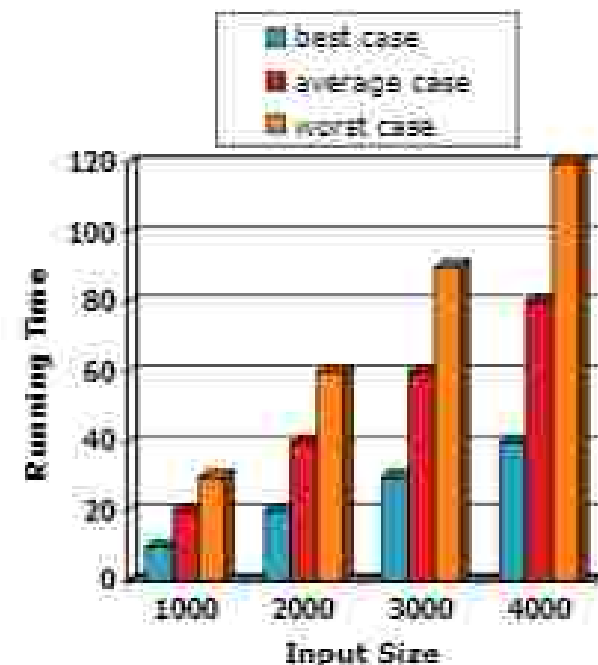
- The running time of an algorithm typically grows with the input size
  - idea: analyze running time as a function of input size

# How to Calculate Running Time

- ▶ Even on inputs of the same size, running time can be very different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right
- Idea: analyze running time in the
  - best case
  - worst case
  - average case

# How to Calculate Running Time

- ▶ Best case running time is usually useless
- ▶ Average case time is very useful but often difficult to determine
- ▶ We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



# Analysis of Algorithms

- ▶ When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*
- ▶ To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

# The Execution Time of Algorithms

- ▶ Each operation in an algorithm (or a program) has a cost.
  - ➔ Each operation takes a certain of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

*A sequence of operations:*

<code>count = count + 1;</code>	Cost: $c_1$
<code>sum = sum + count;</code>	Cost: $c_2$

➔ Total Cost =  $c_1 + c_2$



# The Execution Time of Algorithms (cont.)

*Example: Simple If-Statement*

	<u>Cost</u>	<u>Times</u>
<code>if (n &lt; 0)</code>	$c_1$	1
<code>absval = -n</code>	$c_2$	1
<code>else</code>		
<code>absval = n;</code>	$c_3$	1

$$\text{Total Cost} \leq c_1 + \max(c_2, c_3)$$

# The Execution Time of Algorithms (cont.)

## *Example: Simple Loop*

	<u>Cost</u>	<u>Times</u>
<code>i = 1;</code>	$c_1$	1
<code>sum = 0;</code>	$c_2$	1
<code>while (i &lt;= n) {</code>	$c_3$	$n+1$
<code>i = i + 1;</code>	$c_4$	$n$
<code>sum = sum + i;</code>	$c_5$	$n$
<code>}</code>		

$$\text{Total Cost} = c_1 + c_2 + (n+1)*c_3 + n*c_4 + n*c_5$$

→ The time required for this algorithm is proportional to  $n$

# The Execution Time of Algorithms (cont.)

## *Example: Nested Loop*

	<u>Cost</u>	<u>Times</u>
<code>i=1;</code>	<code>c1</code>	<code>1</code>
<code>sum = 0;</code>	<code>c2</code>	<code>1</code>
<code>while (i &lt;= n) {</code>	<code>c3</code>	<code>n+1</code>
<code>j=1;</code>	<code>c4</code>	<code>n</code>
<code>while (j &lt;= n) {</code>	<code>c5</code>	<code>n*(n+1)</code>
<code>sum = sum + i;</code>	<code>c6</code>	<code>n*n</code>
<code>j = j + 1;</code>	<code>c7</code>	<code>n*n</code>
<code>}</code>		
<code>i = i + 1;</code>	<code>c8</code>	<code>n</code>
<code>}</code>		

$$\text{Total Cost} = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5 + n*n*c6 + n*n*c7 + n*c8$$

→ The time required for this algorithm is proportional to  $n^2$

# General Rules for Estimation

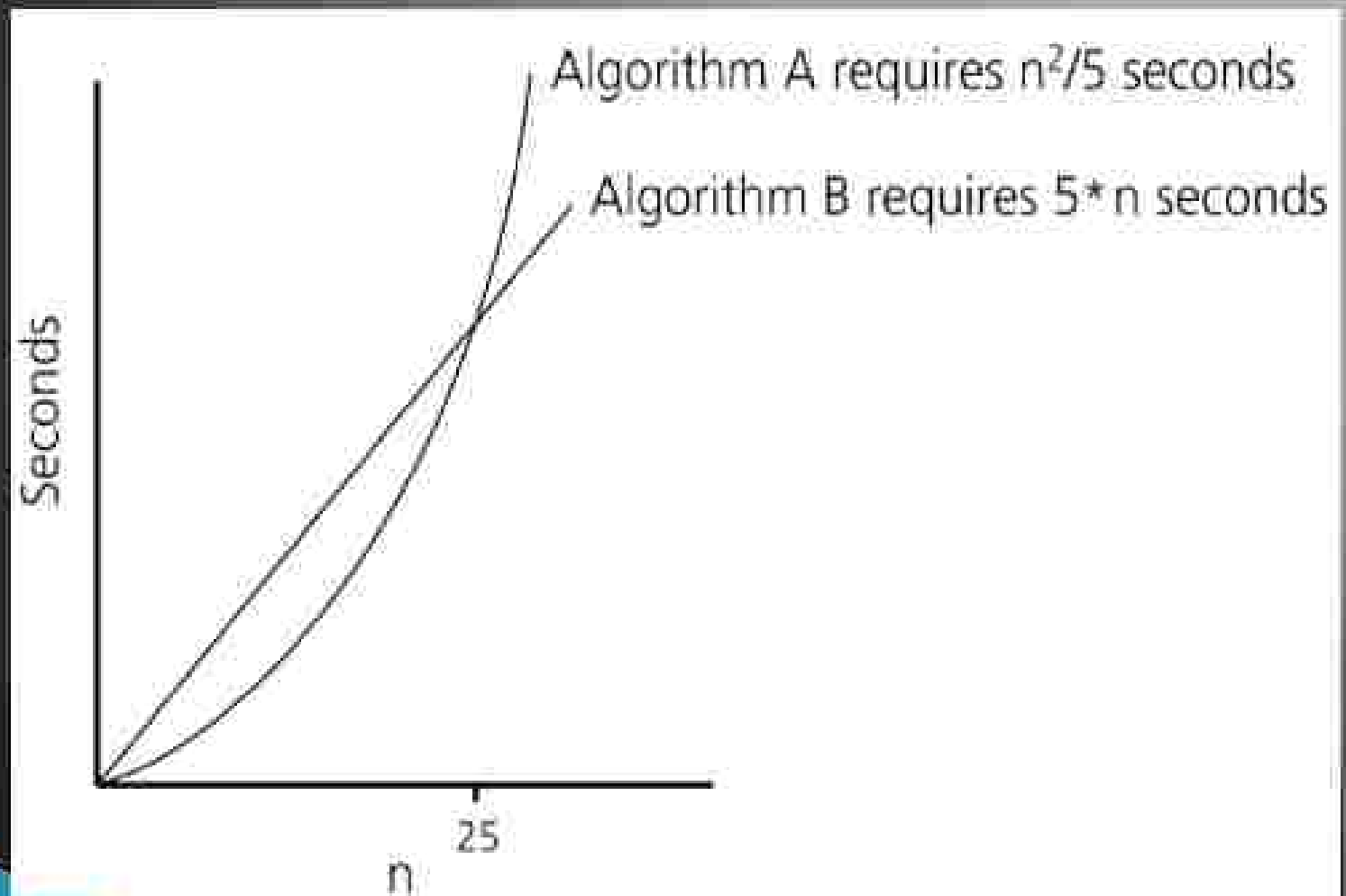
- ▶ **Loops:** The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- ▶ **Nested Loops:** Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- ▶ **Consecutive Statements:** Just add the running times of those consecutive statements.
- ▶ **If/Else:** Never more than the running time of the test plus the larger of running times of  $S1$  and  $S2$ .

# Algorithm Growth Rates

- ▶ We measure an algorithm's time requirement as a function of the *problem size*.
  - Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- ▶ So, for instance, we say that (if the problem size is  $n$ )
  - Algorithm A requires  $5 \cdot n^2$  time units to solve a problem of size  $n$ .
  - Algorithm B requires  $7 \cdot n$  time units to solve a problem of size  $n$ .
- ▶ The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to  $n^2$ .
  - Algorithm B requires time proportional to  $n$ .
- ▶ An algorithm's proportional time requirement is known as *growth rate*.
- ▶ We can compare the efficiency of two algorithms by comparing their growth rates.

# Algorithm Growth Rates (cont.)

*Time requirements as a function of the problem size  $n$*



# Common Growth Rates

Function	Growth Rate Name
$c$	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
$N$	Linear
$N \log N$	
$N^2$	Quadratic
$N^3$	Cubic
$2^N$	Exponential

# Big-Oh and Growth Rate

- ▶ The big-Oh notation gives an upper bound on the growth rate of a function
- ▶ The statement “ $f(n)$  is  $O(g(n))$ ” means that the growth rate of  $f(n)$  is no more than the growth rate of  $g(n)$
- ▶ We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes