



Dedicated to **Abū al-Hasan ibn Ali al-Qalasādi** (1412-1482),
Pioneer of **Symbolic algebra**.



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting – Fundamentals – I

By Dr. Bilal Wajid





Definition

Sorting: Process of **taking a list of objects** which could be stored in a linear order e.g., numbers:

$$(a_0, a_1, \dots, a_{n-1})$$

and returning a reordering

$$(a'_0, a'_1, \dots, a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$$

Sorting: Conversion of an Abstract List into an Abstract Sorted List



Definition

Sorting: Process of **taking a list of objects** which could be stored in a linear order *e.g.*, numbers:

$$(a_0, a_1, \dots, a_{n-1})$$

and **returning a reordering**

$$(a'_0, a'_1, \dots, a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$$

Sorting: Conversion of an Abstract List into an Abstract Sorted List



Definition

Sorting: Process of **taking a list of objects** which could be stored in a linear order *e.g.*, numbers:

$$(a_0, a_1, \dots, a_{n-1})$$

and **returning a reordering**

$$(a'_0, a'_1, \dots, a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$$

Sorting: Conversion of an Abstract List into an Abstract Sorted List



Definition

Sorting: Process of taking a list of objects which could be stored in a linear order e.g., numbers:

$$(a_0, a_1, \dots, a_{n-1})$$

and returning a reordering

$$(a'_0, a'_1, \dots, a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$$

Sorting: Conversion of an Abstract List into an Abstract Sorted List



Example

- List 1: [4, 6, 88, 10, 33, 21, 0, -1]



Example

- List 1: [4, 6, 88, 10, 33, 21, 0, -1]
- Sorted List 1: [-1, 0, 4, 6, 10, 21, 33, 88]



Example

- List 1: [4, 6, 88, 10, 33, 21, 0, -1]
- Sorted List 1: [-1, 0, 4, 6, 10, 21, 33, 88]
- List 2: [z, c, a, e, t, r]



Example

- List 1: [4, 6, 88, 10, 33, 21, 0, -1]
 - Sorted List 1: [-1, 0, 4, 6, 10, 21, 33, 88]
-
- List 2: [z, c, a, e, t, r]
 - Sorted List 2: [a, c, e, r, t, z]



Example

- List 1: [4, 6, 88, 10, 33, 21, 0, -1]
- Sorted List 1: [-1, 0, 4, 6, 10, 21, 33, 88]

- List 2: [z, c, a, e, t, r]
- Sorted List 2: [a, c, e, r, t, z]

We will focus on sorting a list of numbers



Types of Input





Types of Input – Best Case





Types of Input – Worst Case





Types of Input – Average Case





Classifications

The operations of a sorting algorithm are based on the actions performed:

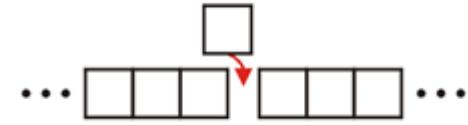
- Insertion
- Exchanging
- Selection
- Merging
- Distribution



Classifications

The operations of a sorting algorithm are based on the actions performed:

- **Insertion**



- Exchanging
- Selection
- Merging
- Distribution



Classifications

The operations of a sorting algorithm are based on the actions performed:

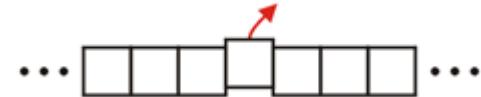
- Insertion
- **Exchanging** 
- Selection
- Merging
- Distribution



Classifications

The operations of a sorting algorithm are based on the actions performed:

- Insertion
- Exchanging
- **Selection**
- Merging
- Distribution

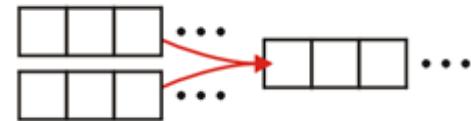




Classifications

The operations of a sorting algorithm are based on the actions performed:

- Insertion
- Exchanging
- Selection
- **Merging**
- Distribution

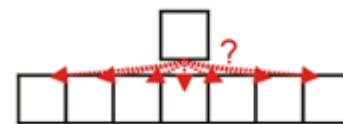




Classifications

The operations of a sorting algorithm are based on the actions performed:

- Insertion
- Exchanging
- Selection
- Merging
- **Distribution**

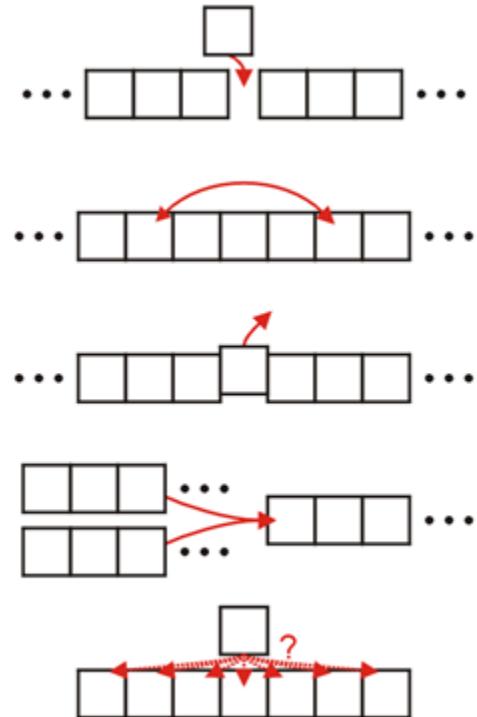




Classifications

The operations of a sorting algorithm are based on the actions performed:

- Insertion
- Exchanging
- Selection
- Merging
- Distribution





Sorting Algorithms

- Incremental comparison sorting
 - 1. Selection sort
 - 2. Bubble sort
 - 3. Insertion sort
- Recursive comparison sorting
 - 1. Merge sort
 - 2. Quick sort
 - 3. Heap sort
- Non-comparison linear sorting
 - 1. Count sort
 - 2. Radix sort
 - 3. Bucket sort



Sorting Algorithms

- Incremental comparison sorting
 - 1. Selection sort
 - 2. Bubble sort
 - 3. Insertion sort
- Recursive comparison sorting
 - 1. Merge sort
 - 2. Quick sort
 - 3. Heap sort
- Non-comparison linear sorting
 - 1. Count sort
 - 2. Radix sort
 - 3. Bucket sort



Sorting Algorithms

- Incremental comparison sorting
 - 1. Selection sort
 - 2. Bubble sort
 - 3. Insertion sort
- Recursive comparison sorting
 - 1. Merge sort
 - 2. Quick sort
 - 3. Heap sort
- Non-comparison linear sorting
 - 1. Count sort
 - 2. Radix sort
 - 3. Bucket sort



Sorting Algorithms

- Incremental comparison sorting
 - 1. Selection sort
 - 2. Bubble sort
 - 3. Insertion sort
- Recursive comparison sorting
 - 1. Merge sort
 - 2. Quick sort
 - 3. Heap sort
- Non-comparison linear sorting
 - 1. Count sort
 - 2. Radix sort
 - 3. Bucket sort



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.



IBN SIRIN'S

DICTIONARY OF DREAMS

As per Islamic Tradition

Dedicated to Abu Bakr Muhammad **Ibn Sirin** Al-Ansari (33-110 AH)

Pioneer of dream interpretation.



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting Fundamentals – II – Inversions

By Dr. Bilal Wajid





Definition – Sorting

$$(a_0, a_1, \dots, a_n) \xrightarrow{\text{Sorting}} a'_0 \leq a'_1 \leq \dots \leq a'_n$$



Types of Input





Types of Input

Input



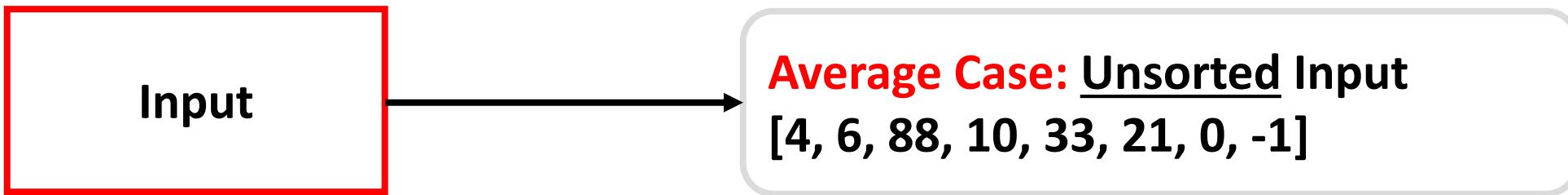
Types of Input

Input

Best Case: Sorted Input
[-1, 0, 4, 6, 10, 21, 33, 88]



Types of Input





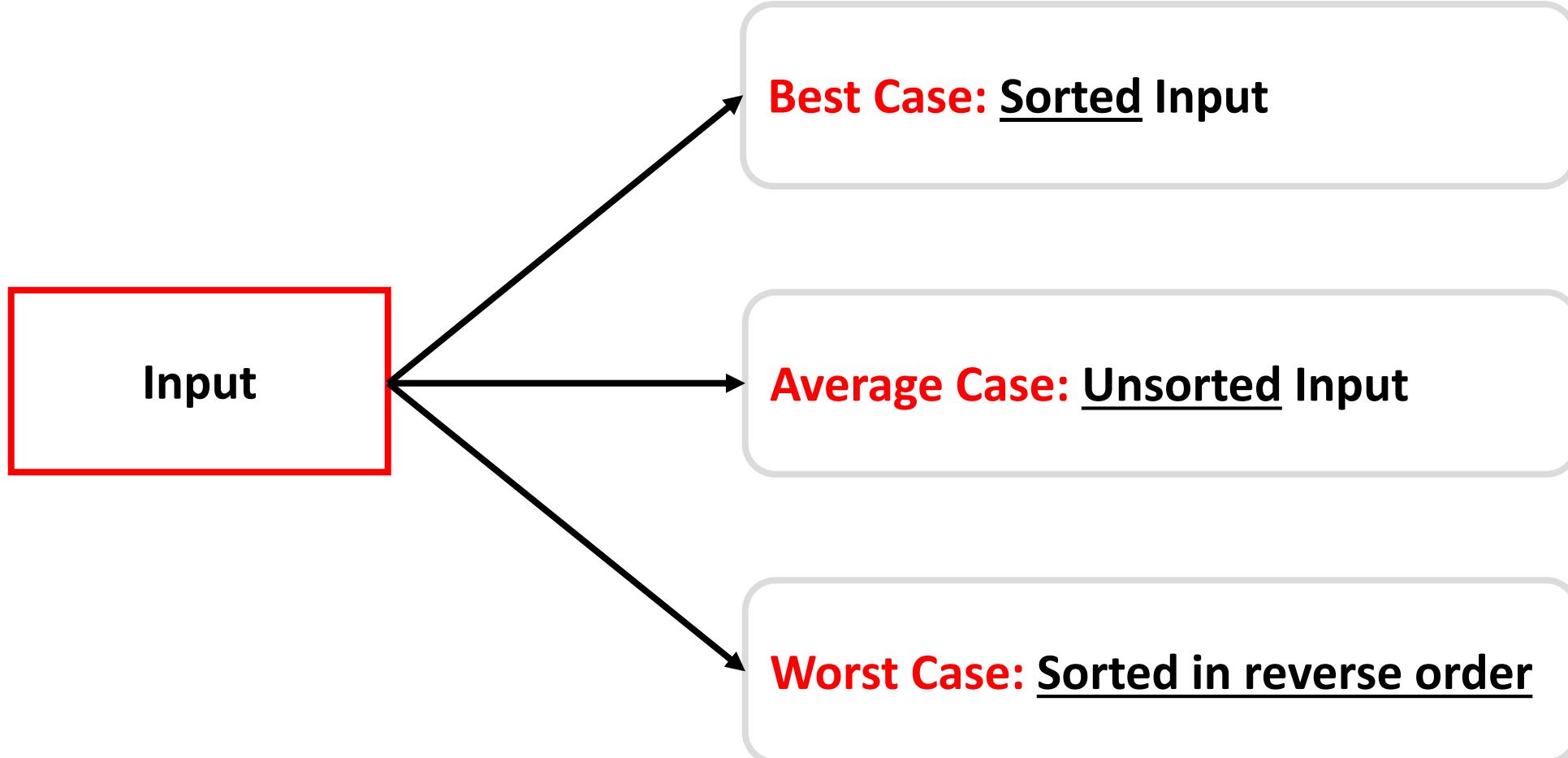
Types of Input

Input

Worst Case: Sorted in reverse order
[88, 33, 21, 10, 6, 4, 0, -1]



Types of Input





Example

*Let us consider an example, where we **compare three inputs***



Example: Compare 3 lists

Consider the following three lists:

1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?



Example: Compare 3 lists

Consider the following three lists:

1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?



Example: Compare 3 lists

Consider the following three lists:

1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?



Example: Compare 3 lists

Consider the following three lists:

1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?



Example: Compare 3 lists

Consider the following three lists:

1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

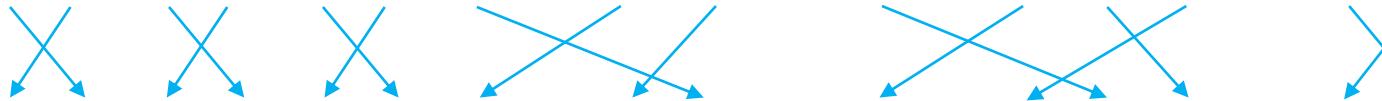
3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?



Comparison: List 1

The first list requires only a few exchanges to make it sorted

- 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

- 1 12 16 25 26 33 35 42 45 56 58 67 74 75 81 83 86 88 95 99



Comparison: List 2

The second list has two entries significantly out of order

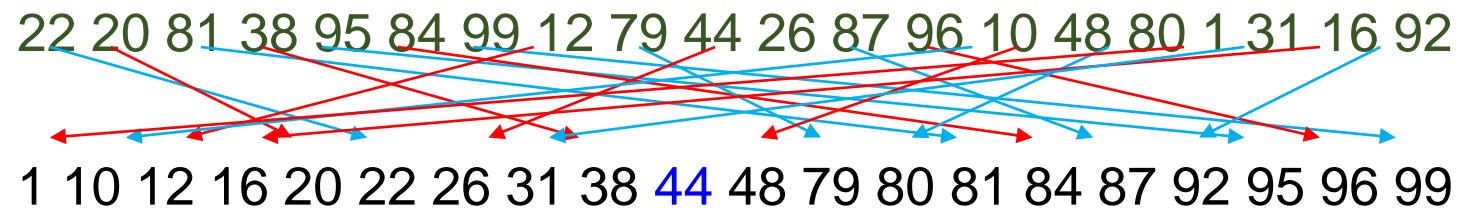
- 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
- 1 17 21 23 24 27 32 35 42 45 47 57 66 69 70 76 85 87 95 99

However, most entries (13) are in place



Comparison: List 3

The third list would, by any reasonable definition, be significantly unsorted





Combinations

Given any list of n numbers, there are

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

pairs of numbers



Combinations

Given any list of n numbers, there are

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

pairs of numbers

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = \frac{6 \cdot (6-1)}{2} = 3 \cdot 5 = 15$ pairs



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = \frac{6 \cdot (6-1)}{2} = 3 \cdot 5 = 15$ pairs:

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
$(5, 4)$	$(5, 2)$	$(5, 6)$		
	$(4, 2)$	$(4, 6)$		
		$(2, 6)$		



Combinations – Ordered

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 pairs, of which **11** are ordered:

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
$(5, 4)$	$(5, 2)$	$(5, 6)$		
	$(4, 2)$	$(4, 6)$		
		$(2, 6)$		



Combinations – Inverted

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 of which **4 are inverted**

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
		$(5, 4)$	$(5, 2)$	$(5, 6)$
			$(4, 2)$	$(4, 6)$
				$(2, 6)$



Combinations – Inverted

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 of which **4 are inverted**

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
		$(5, 4)$	$(5, 2)$	$(5, 6)$
			$(4, 2)$	$(4, 6)$
				$(2, 6)$

Given a pair (a_j, a_k) of entries, an **inversion is** defined as an entry which is reversed such that $(a_j > a_k)$



Inversion

Given a pair (a_j, a_k) of entries, an **inversion is** defined as an entry which is reversed such that $(a_j > a_k)$

Next lecture we focus on these ***inversions*** and how can they help to measure the ***degree of unsortedness***



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Dedicated to Abu Yusuf Yaqub ibn Ishaq Aş-Şabbaḥ **al-Kindi** (801–873 AD)
Pioneer of **Psychotherapy** and **Music therapy**



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting

(III) – Inversions – Example

By Dr. Bilal Wajid





Combinations

Given any list of n numbers, there are

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

pairs of numbers



Combinations

Given any list of n numbers, there are

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

pairs of numbers

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = \frac{6 \cdot (6-1)}{2} = 3 \cdot 5 = 15$ pairs



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = \frac{6 \cdot (6-1)}{2} = 3 \cdot 5 = 15$ pairs:

(1, 3) (1, 5) (1, 4) (1, 2) (1, 6)
(3, 5) (3, 4) (3, 2) (3, 6)
(5, 4) (5, 2) (5, 6)
(4, 2) (4, 6)
(2, 6)



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

$(1, 3)$ $(1, 5)$ $(1, 4)$ $(1, 2)$ $(1, 6)$
 $(3, 5)$ $(3, 4)$ $(3, 2)$ $(3, 6)$
 $(5, 4)$ $(5, 2)$ $(5, 6)$
 $(4, 2)$ $(4, 6)$
 $(2, 6)$



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
$(5, 4)$	$(5, 2)$	$(5, 6)$		
$(4, 2)$	$(4, 6)$			
	$(2, 6)$			



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

(1, 3) (1, 5) (1, 4) (1, 2) (1, 6)
(3, 5) (3, 4) (3, 2) (3, 6)
(5, 4) **(5, 2)** **(5, 6)**
(4, 2) (4, 6)
(2, 6)



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

(1, 3) (1, 5) (1, 4) (1, 2) (1, 6)
(3, 5) (3, 4) (3, 2) (3, 6)
(5, 4) (5, 2) (5, 6)
(4, 2) (4, 6)
(2, 6)



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

(1, 3) (1, 5) (1, 4) (1, 2) (1, 6)
(3, 5) (3, 4) (3, 2) (3, 6)
(5, 4) (5, 2) (5, 6)
(4, 2) (4, 6)
(2, 6)



Combinations

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains $\binom{6}{2} = 15$ pairs:

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
$(5, 4)$	$(5, 2)$	$(5, 6)$		
	$(4, 2)$	$(4, 6)$		
		$(2, 6)$		



Combinations – Ordered

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 pairs: of which **11** are ordered:

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
$(5, 4)$	$(5, 2)$	$(5, 6)$		
	$(4, 2)$	$(4, 6)$		
		$(2, 6)$		



Combinations – Inverted

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 of which **4 are inverted**

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
		$(5, 4)$	$(5, 2)$	$(5, 6)$
			$(4, 2)$	$(4, 6)$
				$(2, 6)$



Combinations – Inverted

For e.g., the list $(1, 3, 5, 4, 2, 6)$ contains 15 of which **4 are inverted**

$(1, 3)$	$(1, 5)$	$(1, 4)$	$(1, 2)$	$(1, 6)$
$(3, 5)$	$(3, 4)$	$(3, 2)$	$(3, 6)$	
		$(5, 4)$	$(5, 2)$	$(5, 6)$
			$(4, 2)$	$(4, 6)$
				$(2, 6)$

Given a pair (a_j, a_k) of entries, an **inversion is** defined as an entry which is reversed such that $(a_j > a_k)$



Inversions

- There are pairs $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ of numbers in any set of n objects



Inversions

- There are pairs $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ of numbers in any set of n objects
- Consequently, each pair contributes to
 - the set of ordered pairs, or
 - the set of inversions



Inversions

- There are pairs $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ of numbers in any set of n objects
- Consequently, each pair contributes to
 - the set of ordered pairs, or ← 50%
 - the set of inversions ← 50%

random ordering



Example

- Let us consider the number of inversions in our first three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92



Example

- Let us consider the number of inversions in our first three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

- Each list has **20** entries, and therefore:

- There are $\binom{20}{2} = 190$ pairs,
- On average, $\frac{190}{2} = 95$ pairs would form inversions



Example

- Let us consider the number of inversions in our first three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

- Each list has 20 entries, and therefore:

- There are $\binom{20}{2} = 190$ pairs,

- On average, $\frac{190}{2} = 95$ pairs would form inversions



Example

- Let us consider the number of inversions in our first three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

- Each list has 20 entries, and therefore:

- There are $\binom{20}{2} = 190$ pairs,
- On average, $\frac{190}{2} = 95$ pairs would form inversions



Example: List I

The first list

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

has 13 inversions:

(16, 12) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (45, 42)
(83, 75) (83, 74) (83, 81) (75, 74) (86, 81) (99, 95)

This is well below 95, the expected number of inversions. Therefore, this is likely not to be a *random* list.



Example: List I

The first list

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

has 13 inversions:

(16, 12) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (45, 42)
(83, 75) (83, 74) (83, 81) (75, 74) (86, 81) (99, 95)

This is well below 95, the expected number of inversions. Therefore, this is likely not to be a *random* list.



Example: List I

The first list

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

has 13 inversions:

(16, 12) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (45, 42)
(83, 75) (83, 74) (83, 81) (75, 74) (86, 81) (99, 95)

This is **well below 95**, the expected number of inversions Therefore, this is likely not to be a *random* list



Example: List II

The second list

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

also has 13 inversions:

(42, 24) (42, 27) (42, 32) (42, 35) (42, 23) (24, 23) (27, 23)
(32, 23) (35, 23) (45, 23) (47, 23) (57, 23) (87, 85)

This, too, is not a random list



Example: List II

The second list

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

also has 13 inversions:

(42, 24) (42, 27) (42, 32) (42, 35) (42, 23) (24, 23) (27, 23)
(32, 23) (35, 23) (45, 23) (47, 23) (57, 23) (87, 85)

This, too, is not a random list



Example: List III

The third list

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

has 100 inversions:

(22, 20) (22, 12) (22, 10) (22, 1) (22, 16) (20, 12) (20, 10) (20, 1) (20, 16) (81, 38)
(81, 12) (81, 79) (81, 44) (81, 26) (81, 10) (81, 48) (81, 80) (81, 1) (81, 16) (81, 31)
(38, 12) (38, 26) (38, 10) (38, 1) (38, 16) (38, 31) (95, 84) (95, 12) (95, 79) (95, 44)
(95, 26) (95, 87) (95, 10) (95, 48) (95, 80) (95, 1) (95, 16) (95, 31) (95, 92) (84, 12)
(84, 79) (84, 44) (84, 26) (84, 10) (84, 48) (84, 80) (84, 1) (84, 16) (84, 31) (99, 12)
(99, 79) (99, 44) (99, 26) (99, 87) (99, 96) (99, 10) (99, 48) (99, 80) (99, 1) (99, 16)
(99, 31) (99, 92) (12, 10) (12, 1) (79, 44) (79, 26) (79, 10) (79, 48) (79, 1) (79, 16)
(79, 31) (44, 26) (44, 10) (44, 1) (44, 16) (44, 31) (26, 10) (26, 1) (26, 16) (87, 10)
(87, 48) (87, 80) (87, 1) (87, 16) (87, 31) (96, 10) (96, 48) (96, 80) (96, 1) (96, 16)
(96, 31) (96, 92) (10, 1) (48, 1) (48, 16) (48, 31) (80, 1) (80, 16) (80, 31) (31, 16)



Example: List III

The third list

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

has 100 inversions:

(22, 20) (22, 12) (22, 10) (22, 1) (22, 16) (20, 12) (20, 10) (20, 1) (20, 16) (81, 38)
(81, 12) (81, 79) (81, 44) (81, 26) (81, 10) (81, 48) (81, 80) (81, 1) (81, 16) (81, 31)
(38, 12) (38, 26) (38, 10) (38, 1) (38, 16) (38, 31) (95, 84) (95, 12) (95, 79) (95, 44)
(95, 26) (95, 87) (95, 10) (95, 48) (95, 80) (95, 1) (95, 16) (95, 31) (95, 92) (84, 12)
(84, 79) (84, 44) (84, 26) (84, 10) (84, 48) (84, 80) (84, 1) (84, 16) (84, 31) (99, 12)
(99, 79) (99, 44) (99, 26) (99, 87) (99, 96) (99, 10) (99, 48) (99, 80) (99, 1) (99, 16)
(99, 31) (99, 92) (12, 10) (12, 1) (79, 44) (79, 26) (79, 10) (79, 48) (79, 1) (79, 16)
(79, 31) (44, 26) (44, 10) (44, 1) (44, 16) (44, 31) (26, 10) (26, 1) (26, 16) (87, 10)
(87, 48) (87, 80) (87, 1) (87, 16) (87, 31) (96, 10) (96, 48) (96, 80) (96, 1) (96, 16)
(96, 31) (96, 92) (10, 1) (48, 1) (48, 16) (48, 31) (80, 1) (80, 16) (80, 31) (31, 16)



Example

- Consider the number of inversions in our three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95 → **13**
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99 → **13**
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92 → **100**

- List 1 and 2 are equally sorted.
- List 3 is highly unsorted.



Example

- Consider the number of inversions in our three lists:

- 1) 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95 → **13**
- 2) 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99 → **13**
- 3) 22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92 → **100**

- List 1 and 2 are equally sorted.
- List 3 is highly unsorted.



Next lecture we focus on **Bubble Sort**



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





**Dedicated to Abu Jaafar Mohammad Ibn Mousa Al Khwarizmi (780-850),
Pioneer of Algebra.**



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting – Fundamentals – I

By Dr. Bilal Wajid





Basic Idea – I

Suppose we have an array of data which is unsorted:

- Starting at the front, traverse the array, find the largest item, and move (or *bubble*) it to the top



Basic Idea – II

Suppose we have an array of data which is unsorted:

- Starting at the front, traverse the array, find the largest item, and move (or *bubble*) it to the top
- With each subsequent iteration, find the next largest item and *bubble* it up towards the top of the array



Observations – I

- Bubble sort is a simple algorithm with:
 - a memorable name, and
 - a simple idea



Observations – II

- Bubble sort is a simple algorithm with:
 - a memorable name, and
 - a simple idea
- Bubble sort is ***“the generic bad algorithm”***



Observations – III

- Bubble sort is a simple algorithm with:
 - a memorable name, and
 - a simple idea
- Bubble sort is “*the generic bad algorithm*”
- Donald Knuth comments that “*the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems*”



Observations – III

- Bubble sort is a simple algorithm with:
 - a memorable name, and
 - a simple idea
- Bubble sort is *“the generic bad algorithm”*
- Donald Knuth comments that *“the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems”*



Implementation – I

- Given a list of N elements, repeat the following steps $N-1$ times:



Implementation – II

- Given a list of N elements, repeat the following steps $N-1$ times:
 - For each pair of adjacent numbers, if number on the left is greater than the number on the right, swap them.





Implementation – III

- Given a list of N elements, repeat the following steps $N-1$ times:
 - For each pair of adjacent numbers, if number on the left is greater than the number on the right, swap them.
 - “Bubble” the largest value to the end using pair-wise comparisons and swapping.



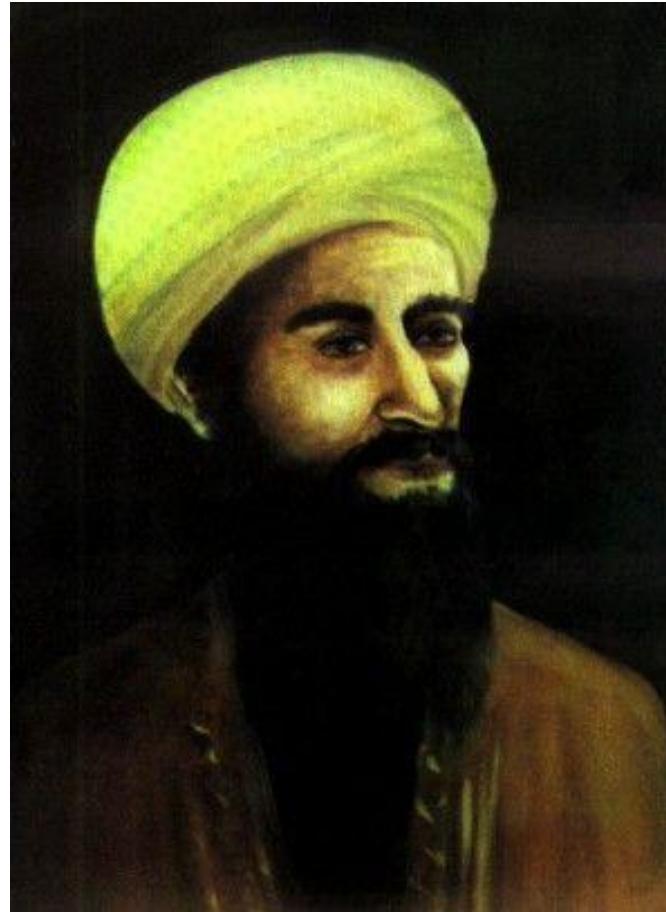
Example

16	16	22	22	22	22
6	12	18	18	17	22
6	18	18	14	17	22
6	8	12	14	17	22



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Dedicated to **Jabir Bin Hayan** (721-813 AH)
Pioneer of **Chemistry**.



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting

Bubble Sort – II

By Dr. Bilal Wajid





Arithmetic Series

- Arithmetic Series Examples:
 - $1 + 2 + 3 + 4 + \cdots + n$
 - $2 + 4 + 6 + 8 + \cdots + n$
 - $1 + 3 + 5 + 7 + \cdots + n$
 - $100 + 200 + 300 + 400 + \cdots + n$
 - $a + (a + d) + (a + 2d) + (a + 3d) + \cdots [a + (n - 1)d]$
 - a = 1st term
 - d = difference between two terms
 - n = last term
- $S_N = \left(\frac{N}{2}\right) \times \{2a + (N - 1) \cdot d\}$





Arithmetic Series

- Arithmetic Series Examples:
 - $1 + 2 + 3 + 4 + \cdots + n$
 - $2 + 4 + 6 + 8 + \cdots + n$
 - $1 + 3 + 5 + 7 + \cdots + n$
 - $100 + 200 + 300 + 400 + \cdots + n$
 - $a + (a + d) + (a + 2d) + (a + 3d) + \cdots [a + (n - 1)d]$
 - a = 1st term
 - d = difference between two terms
 - n = last term
- $S_N = \left(\frac{N}{2}\right) \times \{2a + (N - 1) \cdot d\}$





Algorithm – II

BubbleSort (A)

1. **n = Length [A] ;**

Costs

1. c_1

Times

1. 1



Algorithm – III – A

BubbleSort (A)

1. $n = \text{Length}[A]$;
2. **for** $j = 0$ **to** $n-2$

Costs

1. c_1
2. c_2

Times

1. 1
2. $n - 1$



Algorithm – III – B

BubbleSort (A)

1. $n = \text{Length}[A]$;
2. **for** $j = 0$ **to** $n-2$

Costs

1. c_1
2. c_2

Times

1. 1
2. $n - 1$

- $j = 1 \rightarrow 5: \times 5$
- $j = 0 \rightarrow 5: \times 6$
- $j = 0 \rightarrow n: \times (n + 1)$
- $j = 0 \rightarrow 3: \times 4$
- $j = 0 \rightarrow (n - 2): \times (n - 1)$



Algorithm – III – F

BubbleSort (A)

1. $n = \text{Length}[A]$;
2. **for** $j = 0$ **to** $n-2$

Costs

1. c_1
2. c_2

Times

1. 1
2. $n - 1$

- $j = 1 \rightarrow 5: \times 5$
- $j = 0 \rightarrow 5: \times 6$
- $j = 0 \rightarrow n: \times (n + 1)$
- $j = 0 \rightarrow 3: \times 4$
- $j = 0 \rightarrow (n - 2): \times (n - 1)$



Algorithm – III

BubbleSort (A)

1. $n = \text{Length}[A]$;
2. **for** $j = 0$ **to** $n-2$

Costs

1. c_1
2. c_2

Times

1. 1
2. $n - 1$



Algorithm – IV – A

BubbleSort (A)	Costs	Times
1. $n = \text{Length}[A]$;	1. c_1	1. 1
2. for $j = 0$ to $n-2$	2. c_2	2. $n - 1$
3. for $i = 0$ to $n-j-2$	3. c_3	$\sum_{j=0}^{n-2} T_j$



Algorithm – IV – B

- for $i = 0 \rightarrow (n - 2)$
- for $j = 0 \rightarrow (n - i - 2)$

- $i = 0$
- $i = 1$
- $i = 2$
- $i = 3$
- ...
- $i = n - 3$
- $i = n - 2$

- $j = 0 \rightarrow n - 2 \quad : \times (n - 1)$
- $j = 0 \rightarrow n - 3 \quad : \times (n - 2)$
- $j = 0 \rightarrow n - 4 \quad : \times (n - 3)$
- $j = 0 \rightarrow n - 5 \quad : \times (n - 4)$
- ...
- $j = 0 \rightarrow n - (n - 3) - 2$
 $= 0 \rightarrow n - n + 3 - 2$
 $= 0 \rightarrow 1 \quad : \times 2$
- $j = 0 \rightarrow 0 \quad : \times 1$



Algorithm – IV – H

- $S_j = T_0 + T_1 + \dots + T_{(n-2)}$

- $i = 0$
- $i = 1$
- $i = 2$
- $i = 3$
- ...
- $i = n - 3$
- $i = n - 2$

- $j = 0 \rightarrow n - 2 \quad : \times (n - 1)$
- $j = 0 \rightarrow n - 3 \quad : \times (n - 2)$
- $j = 0 \rightarrow n - 4 \quad : \times (n - 3)$
- $j = 0 \rightarrow n - 5 \quad : \times (n - 4)$
- ...
- $j = 0 \rightarrow n - (n - 3) - 2$
 $= 0 \rightarrow n - n + 3 - 2$
 $= 0 \rightarrow 1 \quad : \times 2$
- $j = 0 \rightarrow 0 \quad : \times 1$



Algorithm – IV – I

- $S_j = T_0 + T_1 + \dots + T_{(n-2)}$
- $S_j = T_{(n-2)} + T_{(n-3)} + \dots + T_1 + T_0$
- $i = 0$
- $i = 1$
- $i = 2$
- $i = 3$
- ...
- $i = n - 3$
- $i = n - 2$
- $j = 0 \rightarrow n - 2 \quad : \times (n - 1)$
- $j = 0 \rightarrow n - 3 \quad : \times (n - 2)$
- $j = 0 \rightarrow n - 4 \quad : \times (n - 3)$
- $j = 0 \rightarrow n - 5 \quad : \times (n - 4)$
- ...
- $j = 0 \rightarrow n - (n - 3) - 2$
 $= 0 \rightarrow n - n + 3 - 2$
 $= 0 \rightarrow 1 \quad : \times 2$
- $j = 0 \rightarrow 0 \quad : \times 1$



Algorithm – IV – K

- $S_j = T_0 + T_1 + \dots + T_{(n-2)}$
- $S_j = T_{(n-2)} + T_{(n-3)} + \dots + T_1 + T_0$
- $S_j = 1 + 2 + \dots + (n - 1)$
- $S_j = \sum_{j=0}^{n-2} T_j$
- S_j = Arithmetic Series
- $S_j = \frac{N}{2} \times [2(a) + (N - 1)d]$
- $S_j = \frac{N}{2} \times [2(1) + (N - 1)1]$
- $S_j = \frac{N}{2} \times [2 + N - 1]$
- $S_j = \frac{N(N+1)}{2} = \frac{(n-1) \times n}{2} = \frac{n(n-1)}{2}$

- $j = 0 \rightarrow n - 2 \quad : \times (n - 1)$
- $j = 0 \rightarrow n - 3 \quad : \times (n - 2)$
- $j = 0 \rightarrow n - 4 \quad : \times (n - 3)$
- $j = 0 \rightarrow n - 5 \quad : \times (n - 4)$
- \dots
- $j = 0 \rightarrow n - (n - 3) - 2$
 $= 0 \rightarrow n - n + 3 - 2$
 $= 0 \rightarrow 1 \quad : \times 2$
- $j = 0 \rightarrow 0 \quad : \times 1$



Algorithm – IV – R

- $S_j = T_0 + T_1 + \dots + T_{(n-2)}$
- $S_j = T_{(n-2)} + T_{(n-3)} + \dots + T_1 + T_0$
- $S_j = 1 + 2 + \dots + (n - 1)$
- $S_j = \sum_{j=0}^{n-2} T_j$
- S_j = Arithmetic Series
- $S_j = \frac{N}{2} \times [2(a) + (N - 1)d]$
- $S_j = \frac{N}{2} \times [2(1) + (N - 1)1]$
- $S_j = \frac{N}{2} \times [2 + N - 1]$
- $S_j = \frac{N(N+1)}{2} = \frac{(n-1) \times n}{2} = \frac{n(n-1)}{2}$

- $j = 0 \rightarrow n - 2 \quad : \times (n - 1)$
- $j = 0 \rightarrow n - 3 \quad : \times (n - 2)$
- $j = 0 \rightarrow n - 4 \quad : \times (n - 3)$
- $j = 0 \rightarrow n - 5 \quad : \times (n - 4)$
- \dots
- $j = 0 \rightarrow n - (n - 3) - 2$
 $= 0 \rightarrow n - n + 3 - 2$
 $= 0 \rightarrow 1 \quad : \times 2$
- $j = 0 \rightarrow 0 \quad : \times 1$



Algorithm – IV – S

BubbleSort (A)

```
1. n = Length [A] ;  
2. for j = 0 to n-2  
3.   for i = 0 to n-j-2  
4.     if A[i] > A[i+1]  
5.       temp = A[i]  
6.       A[i] = A[i+1]  
7.       A[i+1] = temp  
8. return A;
```

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8

Times

1. 1
2. $n - 1$
3. $\sum_{j=0}^{n-2} T_j = \frac{n(n-1)}{2}$
4. $\sum_{j=0}^{n-2} T_j$
5. $\sum_{j=0}^{n-2} T_j$
6. $\sum_{j=0}^{n-2} T_j$
7. $\sum_{j=0}^{n-2} T_j$
8. 1



Algorithm – X

BubbleSort (A)	Costs	Times
1. n = Length [A] ;	1. c_1	1. 1
2. for j = 0 to n-2	2. c_2	2. $n - 1$
3. for i = 0 to n-j-2	3. c_3	3. $\sum_{j=0}^{n-2} T_j$
4. if A[i] > A[i+1]	4. c_4	4. $\sum_{j=0}^{n-2} T_j$
5. temp = A[i]	5. c_5	5. $\sum_{j=0}^{n-2} T_j$
6. A[i] = A[i+1]	6. c_6	6. $\sum_{j=0}^{n-2} T_j$
7. A[i+1] = temp	7. c_7	7. $\sum_{j=0}^{n-2} T_j$
8. return A ;	8. c_8	8. 1



Algorithm – XI

Total Times = Costs × Times	Costs	Times
1. $c_1 \cdot 1$	1. c_1	1. 1
2. $c_2 \cdot (n - 1)$	2. c_2	2. $n - 1$
3. $c_3 \cdot (\sum_{j=0}^{n-2} T_j)$	3. c_3	3. $\sum_{j=0}^{n-2} T_j$
4. $c_4 \cdot (\sum_{j=0}^{n-2} T_j)$	4. c_4	4. $\sum_{j=0}^{n-2} T_j$
5. $c_5 \cdot (\sum_{j=0}^{n-2} T_j)$	5. c_5	5. $\sum_{j=0}^{n-2} T_j$
6. $c_6 \cdot (\sum_{j=0}^{n-2} T_j)$	6. c_6	6. $\sum_{j=0}^{n-2} T_j$
7. $c_7 \cdot (\sum_{j=0}^{n-2} T_j)$	7. c_7	7. $\sum_{j=0}^{n-2} T_j$
8. $c_8 \cdot 1$	8. c_8	8. 1



Algorithm – XII

Total Times = Costs × Times

1. $c_1 \cdot 1$
2. $c_2 \cdot (n - 1)$
3. $c_3 \cdot (\sum_{j=0}^{n-2} T_j)$
4. $c_4 \cdot (\sum_{j=0}^{n-2} T_j)$
5. $c_5 \cdot (\sum_{j=0}^{n-2} T_j)$
6. $c_6 \cdot (\sum_{j=0}^{n-2} T_j)$
7. $c_7 \cdot (\sum_{j=0}^{n-2} T_j)$
8. $c_8 \cdot 1$

$$T(n) = c_1 + c_2 \cdot (n - 1) + c_3 \cdot \left(\sum_{j=0}^{n-2} T_j \right) + c_4 \cdot \left(\sum_{j=0}^{n-2} T_j \right) + c_5 \cdot \left(\sum_{j=0}^{n-2} T_j \right) + c_6 \cdot \left(\sum_{j=0}^{n-2} T_j \right) + c_7 \cdot \left(\sum_{j=0}^{n-2} T_j \right) + c_8$$



Algorithm – XIII

Total Times = Costs × Times

1. $c_1 \cdot 1$
2. $c_2 \cdot (n - 1)$
3. $c_3 \cdot (\sum_{j=0}^{n-2} T_j)$
4. $c_4 \cdot (\sum_{j=0}^{n-2} T_j)$
5. $c_5 \cdot (\sum_{j=0}^{n-2} T_j)$
6. $c_6 \cdot (\sum_{j=0}^{n-2} T_j)$
7. $c_7 \cdot (\sum_{j=0}^{n-2} T_j)$
8. $c_8 \cdot 1$

$$T(n) = c_1 + c_2 \cdot (n - 1) + c_3 \cdot \left[\frac{n(n - 1)}{2} \right] + c_4 \cdot \left[\frac{n(n - 1)}{2} \right] \\ + c_5 \cdot \left[\frac{n(n - 1)}{2} \right] + c_6 \cdot \left[\frac{n(n - 1)}{2} \right] + c_7 \cdot \left[\frac{n(n - 1)}{2} \right] + c_8$$



Algorithm – XIII

Total Times = Costs × Times

1. $c_1 \cdot 1$
2. $c_2 \cdot (n - 1)$
3. $c_3 \cdot (\sum_{j=0}^{n-2} T_j)$
4. $c_4 \cdot (\sum_{j=0}^{n-2} T_j)$
5. $c_5 \cdot (\sum_{j=0}^{n-2} T_j)$
6. $c_6 \cdot (\sum_{j=0}^{n-2} T_j)$
7. $c_7 \cdot (\sum_{j=0}^{n-2} T_j)$
8. $c_8 \cdot 1$

$$T(n) = (c_1 + c_8) + c_2 \cdot (n - 1) + (c_3 + c_4 + c_5 + c_6 + c_7) \cdot \left[\frac{n(n - 1)}{2} \right]$$



Algorithm – XIV

Total Times = Costs × Times

1. $c_1 \cdot 1$
2. $c_2 \cdot (n - 1)$
3. $c_3 \cdot (\sum_{j=0}^{n-2} T_j)$
4. $c_4 \cdot (\sum_{j=0}^{n-2} T_j)$
5. $c_5 \cdot (\sum_{j=0}^{n-2} T_j)$
6. $c_6 \cdot (\sum_{j=0}^{n-2} T_j)$
7. $c_7 \cdot (\sum_{j=0}^{n-2} T_j)$
8. $c_8 \cdot 1$

$$T(n) = (c_1 + c_8 + c_2) \cdot (n - 1) + (c_3 + c_4 + c_5 + c_6 + c_7) \cdot \left[\frac{n(n - 1)}{2} \right]$$
$$T(n) = an^2 + bn + c \approx an^2 \rightarrow O(n^2)$$



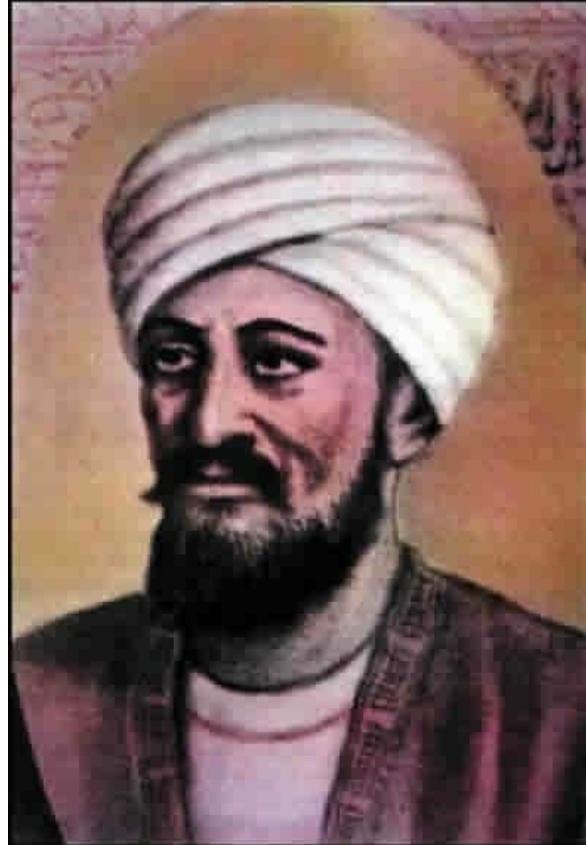
Algorithm – XVI

- Hence Time Complexity of Bubble Sort is $O(n^2)$
- Since the algorithm does not care how sorted the input is therefore, the nested loop will execute completely.
- Hence:
 - Best Case: $O(n^2)$
 - Average Case: $O(n^2)$
 - Worst Case: $O(n^2)$



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Dedicated to **Abu Qasim Khalaf Ibn Abbas Al Zahrawi** (936–1013 AD)
Pioneer of **Modern Surgery**



Sabz-Qalam
Pakistan's premier data-science research institute.

Sorting

Bubble Sort — III

By Dr. Bilal Wajid





Algorithm – I

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.   for i = 0 to n-j-2  
4.     if A[i] > A[i+1]  
5.       temp = A[i]  
6.       A[i] = A[i+1]  
7.       A[i+1] = temp  
8. return A;
```

Analysis

The algorithm does not care how sorted the input is therefore, the nested loop will execute completely.

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$



Algorithm – I

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.   for i = 0 to n-j-2  
4.     if A[i] > A[i+1]  
5.       temp = A[i]  
6.       A[i] = A[i+1]  
7.       A[i+1] = temp  
8. return A;
```

Analysis

Best Case: The array is already sorted.

- Keep track of no. of swaps done in an iteration
- Can finish early if no swapping occurs
- $t_j = 1 \rightarrow n - 1$ comparisons in a single iteration



Algorithm – I

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.   for i = 0 to n-j-2  
4.     if A[i] > A[i+1]  
5.       temp = A[i]  
6.       A[i] = A[i+1]  
7.       A[i+1] = temp  
8. return A;
```

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.   swap = 0;  
4.   for i = 0 to n-j-2  
5.     if A[i] > A[i+1]  
6.       temp = A[i]  
7.       A[i] = A[i+1]  
8.       A[i+1] = temp  
9.     swap = swap + 1;  
10.    if swap = 0  
11.      break;  
12. return A;
```



Algorithm - II

Bubble Sort(A)

[2, 5, 7, 10, 75, 99]



```
1. n = Length[A];  
2. for j = 0 to n-2  
3.     swap = 0;  
4.     for i = 0 to n-j-2  
5.         if A[i] > A[i+1]  
6.             temp = A[i]  
7.             A[i] = A[i+1]  
8.             A[i+1] = temp  
9.             swap = swap + 1;  
10.        if swap = 0  
11.            break;  
12.    return A;
```



Algorithm - II

Bubble Sort(A)

Will only occur if items are not sorted



```
1. n = Length[A];  
2. for j = 0 to n-2  
3.     swap = 0;  
4.     for i = 0 to n-j-2  
5.         if A[i] > A[i+1]  
6.             temp = A[i]  
7.             A[i] = A[i+1]  
8.             A[i+1] = temp  
9.         swap = swap + 1;  
10.        if swap = 0  
11.            break;  
12.    return A;
```



Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.     swap = 0;  
4.     for i = 0 to n-j-2  
5.         if A[i] > A[i+1]  
6.             temp = A[i]  
7.             A[i] = A[i+1]  
8.             A[i+1] = temp  
9.             swap = swap + 1;  
10.    if swap = 0  
11.        break;  
12.    return A;
```

If nothing is inverted then, the list is sorted. Hence, no need to continue





Best Case - Analysis

Cost x Times	Times	Costs
• c_1	• 1	1. c_1
• $c_2(n - 1)$	• $(n - 1)$	2. c_2
• $c_3(n - 1)$	• $(n - 1)$	3. c_3
		4. c_4
		5. c_5
		6. c_6
		7. c_7
		8. c_8
		9. c_9
		10. c_{10}
		11. c_{11}
• c_{12}	• 1	12. c_{12}

Algorithm - II

Bubble Sort(A)

1. $n = \text{Length}[A];$

2. $\text{for } j = \boxed{0} \text{ to } \boxed{n-2}$

3. $\text{swap} = 0;$

4. $\text{for } i = 0 \text{ to } n-j-2$

5. $\text{if } A[i] > A[i+1]$

6. $\text{temp} = A[i];$

7. $A[i] = A[i+1];$

8. $A[i+1] = \text{temp};$

9. $\text{swap} = \text{swap} + 1;$

10. $\text{if } \text{swap} = 0$

11. break;

12. $\text{return } A;$

Upper Limit (UL)

Lower Limit (LL)

$$\begin{aligned} &= \text{UP} - \text{LL} + 1 \\ &= (n-2) - 0 + 1 \\ &= n - 1 \end{aligned}$$



Best Case - Analysis

Two elements will swap only if they are inverted, i.e., $A[i] > A[j]$.

Otherwise, if elements are sorted then $A[i] < A[j]$ and this will not run.

Hence, number of times = 0

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

1. $n = \text{Length}[A];$
2. $\text{for } j = 0 \text{ to } n-2$
3. $\text{swap} = 0;$
4. $\text{for } i = 0 \text{ to } n-j-2$
5. $\text{if } A[i] > A[i+1]$
6. $\text{temp} = A[i]$
7. $A[i] = A[i+1]$
8. $A[i+1] = \text{temp}$
9. $\text{swap} = \text{swap} + 1;$
10. $\text{if } \text{swap} = 0$
11. break;
12. $\text{return } A;$

$\text{temp} = A[i]$
 $A[i] = A[i+1]$
 $A[i+1] = \text{temp}$
 $\text{swap} = \text{swap} + 1;$



Best Case - Analysis

Cost x Times	Times	Costs
• c_1	• 1	1. c_1
• $c_2(n - 1)$	• $(n - 1)$	2. c_2
• $c_3(n - 1)$	• $(n - 1)$	3. c_3
		4. c_4
		5. c_5
	• 0	6. c_6
	• 0	7. c_7
	• 0	8. c_8
	• 0	9. c_9
		10. c_{10}
		11. c_{11}
• c_{12}	• 1	12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Cost x Times	Times	Costs
• c_1	• 1	1. c_1
• $c_2(n - 1)$	• $(n - 1)$	2. c_2
• $c_3(n - 1)$	• $(n - 1)$	3. c_3
		4. c_4
		5. c_5
• 0	• 0	6. c_6
• 0	• 0	7. c_7
• 0	• 0	8. c_8
• 0	• 0	9. c_9
		10. c_{10}
		11. c_{11}
• c_{12}	• 1	12. c_{12}

Algorithm - II

Bubble Sort(A)

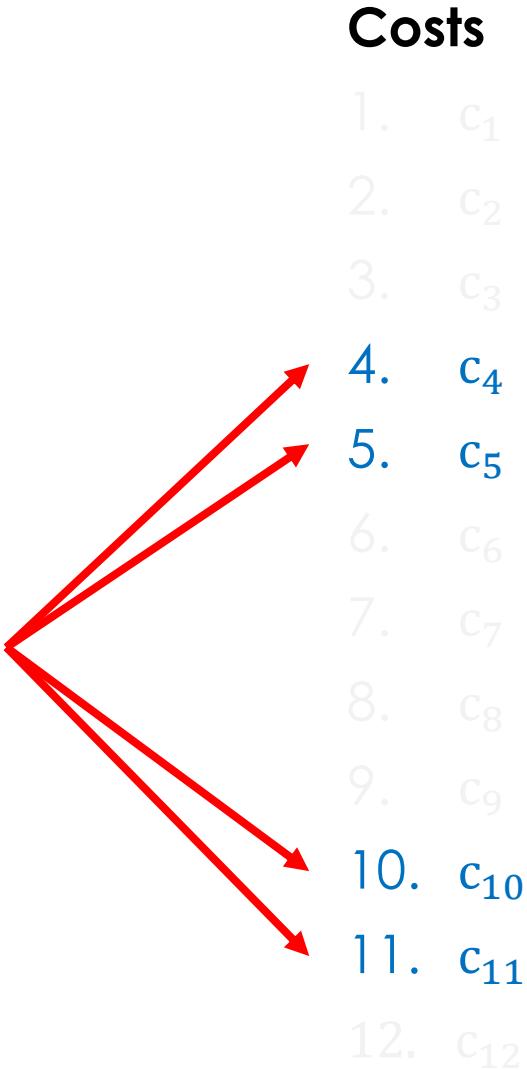
```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.



Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.     swap = 0;  
4.     for i = 0 to n-j-2  
5.         if A[i] > A[i+1]  
6.             temp = A[i]  
7.             A[i] = A[i+1]  
8.             A[i+1] = temp  
9.         swap = swap + 1;  
10.        if swap = 0  
11.            break;  
12.    return A;
```



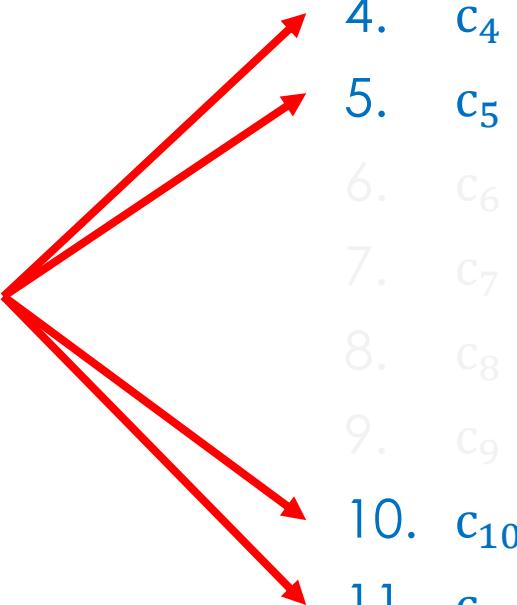
Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

```
1. .
2. for j = 0 to n-2
3. .
4.   for i = 0 to n-j-2
5.   .
6.   .
7.   .
8.   .
9.   .
10.  .
11.  .
12. return A;
```

Outer loop ranges from 0 to n-2



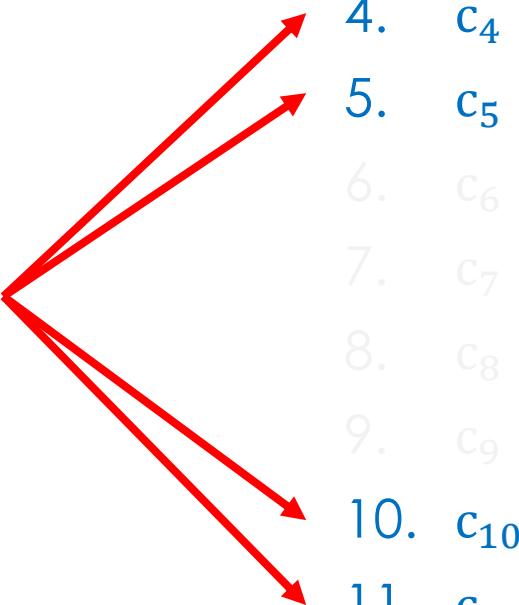
Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

1. .
2. for $j = 0$ to $n-2$ Outer loop ranges from 0 to $n-2$
3. .
4. for $i = 0$ to $n-j-2$ Inner loop depends upon 'j'
5. .
6. .
7. .
8. .
9. .
10. .
11. .
12. return A;



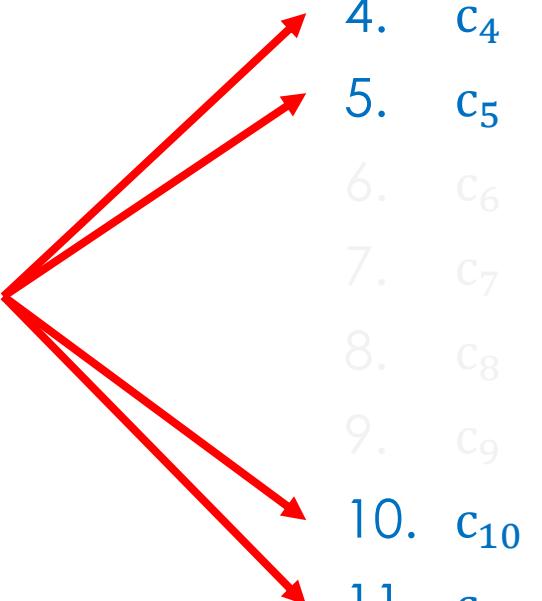
Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

1. .
2. for $j = 0$ to $n-2$ Outer loop ranges from 0 to $n-2$
3. .
4. for $i = 0$ to $n-j-2$ Inner loop depends upon 'j'
 5. .
 6. .
 7. .
 8. .
 9. .
 10. .
 11. .
12. return A;

$j = 0, i = 0$ to $n-2$
 $j = 1, i = 0$ to $n-3$
 $j = 2, i = 0$ to $n-4$
...
 $j = n-3, i = 0$ to 1
 $j = n-2, i = 0$ to 0



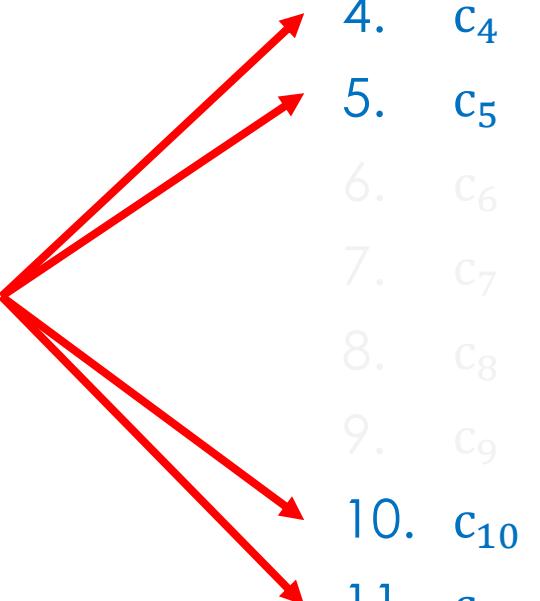
Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

1. .
2. for $j = 0$ to $n-2$
3. .
4. for $i = 0$ to $n-j-2$
5. .
6. .
7. .
8. .
9. .
10. .
11. .
12. return A;

In other words, the inner loop will run a different number of time each time the outer loop runs

$j = 0, i = 0$ to $n-2$
 $j = 1, i = 0$ to $n-3$
 $j = 2, i = 0$ to $n-4$
...
 $j = n-3, i = 0$ to 1
 $j = n-2, i = 0$ to 0



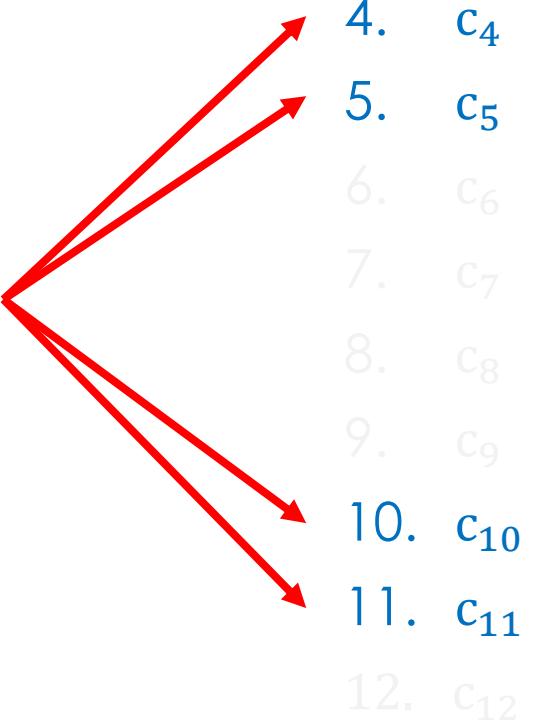
Best Case - Analysis

Lines 4, 5, 10, 11 will always run.

Moreover, they will run equal number of times.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}



Algorithm - II

Bubble Sort(A)

```
1. .
2. for j = 0 to n-2
```

```
3. .
```

```
4. for i = 0 to n-j-2
```

```
5. .
```

```
6. .
```

In other words, the inner loop will run a different number of time each time the outer loop runs

```
7. .
```

```
8. .
```

```
9. .
```

$$10. . \sum_{j=0}^{n-2} T_j$$

```
11. .
```

```
12. return A;
```

$j = 0, i = 0 \text{ to } n-2$
 $j = 1, i = 0 \text{ to } n-3$
 $j = 2, i = 0 \text{ to } n-4$
 \dots
 $j = n-3, i = 0 \text{ to } 1$
 $j = n-2, i = 0 \text{ to } 0$



Best Case - Analysis

Times	Costs
• $\sum_{j=0}^{n-2} T_j$	1. c_1
• $\sum_{j=0}^{n-2} T_j$	2. c_2
	3. c_3
• $\sum_{j=0}^{n-2} T_j$	4. c_4
• $\sum_{j=0}^{n-2} T_j$	5. c_5
	6. c_6
	7. c_7
	8. c_8
	9. c_9
• $\sum_{j=0}^{n-2} T_j$	10. c_{10}
• $\sum_{j=0}^{n-2} T_j$	11. c_{11}
	12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Cost x Times	Times	Costs
• $c_4 \sum_{j=0}^{n-2} T_j$	• $\sum_{j=0}^{n-2} T_j$	1. c_1
• $c_5 \sum_{j=0}^{n-2} T_j$	• $\sum_{j=0}^{n-2} T_j$	2. c_2
		3. c_3
		4. c_4
		5. c_5
		6. c_6
		7. c_7
		8. c_8
		9. c_9
• $c_{10} \sum_{j=0}^{n-2} T_j$	• $\sum_{j=0}^{n-2} T_j$	10. c_{10}
• $c_{11} \sum_{j=0}^{n-2} T_j$	• $\sum_{j=0}^{n-2} T_j$	11. c_{11}
		12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

In a sorted list there will be no swaps.

Hence (`swap == 0`) will be TRUE.

Hence, the **for loop** will break and **run only once**.

Costs

1. c_1
2. c_2
3. c_3
4. c_4
5. c_5
6. c_6
7. c_7
8. c_8
9. c_9
10. c_{10}
11. c_{11}
12. c_{12}

Algorithm - II

Bubble Sort(A)

1. $n = \text{Length}[A];$
2. $\text{for } j = 0 \text{ to } n-2$
3. $\text{swap} = 0;$
4. $\text{for } i = 0 \text{ to } n-j-2$
5. $\text{if } A[i] > A[i+1]$
6. $\text{temp} = A[i]$
7. $A[i] = A[i+1]$
8. $A[i+1] = \text{temp}$
9. $\text{swap} = \text{swap} + 1;$
10. $\text{if } \text{swap} = 0$
11. break;
12. $\text{return } A;$



Best Case - Analysis

Cost x Times	Times	Costs
• $c_4 \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	1. c_1
• $c_5 \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	2. c_2
		3. c_3
		4. c_4
		5. c_5
		6. c_6
		7. c_7
		8. c_8
		9. c_9
• $c_{10} \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	10. c_{10}
• $c_{11} \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	11. c_{11}
		12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Cost x Times	Times	Costs
• c_1	• 1	1. c_1
• $c_2(n - 1)$	• $(n - 1)$	2. c_2
• $c_3(n - 1)$	• $(n - 1)$	3. c_3
• $c_4 \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	4. c_4
• $c_5 \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	5. c_5
• 0	• 0	6. c_6
• 0	• 0	7. c_7
• 0	• 0	8. c_8
• 0	• 0	9. c_9
• $c_{10} \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	10. c_{10}
• $c_{11} \sum_{j=0}^{n-2} 1$	• $\sum_{j=0}^{n-2} T_j$	11. c_{11}
• c_{12}	• 1	12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Cost x Times	Times	Costs
• c_1	• 1	1. c_1
• $c_2(n - 1)$	• $(n - 1)$	2. c_2
• $c_3(n - 1)$	• $(n - 1)$	3. c_3
• $c_4(n - 1)$	• $\sum_{j=0}^{n-2} T_j$	4. c_4
• $c_5(n - 1)$	• $\sum_{j=0}^{n-2} T_j$	5. c_5
	• 0	6. c_6
	• 0	7. c_7
	• 0	8. c_8
	• 0	9. c_9
• $c_{10}(n - 1)$	• $\sum_{j=0}^{n-2} T_j$	10. c_{10}
• $c_{11}(n - 1)$	• $\sum_{j=0}^{n-2} T_j$	11. c_{11}
• c_{12}	• 1	12. c_{12}

Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];
2. for j = 0 to n-2
3.   swap = 0;
4.   for i = 0 to n-j-2
5.     if A[i] > A[i+1]
6.       temp = A[i]
7.       A[i] = A[i+1]
8.       A[i+1] = temp
9.   swap = swap + 1;
10.  if swap = 0
11.    break;
12.  return A;
```



Best Case - Analysis

Cost x Times

- c_1
- $c_2(n - 1)$
- $c_3(n - 1)$
- $c_4(n - 1)$
- $c_5(n - 1)$

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$

- $c_{10}(n - 1)$
- $c_{11}(n - 1)$
- c_{12}



Best Case - Analysis

Cost x Times

- c_1
- $c_2n - c_2$
- $c_3n - c_3$
- $c_4n - c_4$
- $c_5n - c_5$
- $c_{10}n - c_{10}$
- $c_{11}n - c_{11}$
- c_{12}

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$



Best Case - Analysis

Cost x Times

- c_1
- $c_2n - c_2$
- $c_3n - c_3$
- $c_4n - c_4$
- $c_5n - c_5$
- $c_{10}n - c_{10}$
- $c_{11}n - c_{11}$
- c_{12}

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$

$$a = c_1 - c_2 - c_3 - c_4 - c_5 - c_{10} - c_{11} + c_{12}$$



Best Case - Analysis

Cost x Times

- c_1
- $c_2n - c_2$
- $c_3n - c_3$
- $c_4n - c_4$
- $c_5n - c_5$

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$

$$a = c_1 - c_2 - c_3 - c_4 - c_5 - c_{10} - c_{11} + c_{12}$$

$$bn = c_2n + c_3n + c_4n + c_5n + c_{10}n + c_{11}n$$
$$(b) \cdot n = (c_2 + c_3 + c_4 + c_5 + c_{10} + c_{11}) \cdot n$$

- $c_{10}n - c_{10}$
- $c_{11}n - c_{11}$
- c_{12}



Best Case - Analysis

Cost x Times

- c_1
- $c_2n - c_2$
- $c_3n - c_3$
- $c_4n - c_4$
- $c_5n - c_5$

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$

$$a = c_1 - c_2 - c_3 - c_4 - c_5 - c_{10} - c_{11} + c_{12}$$

$$bn = c_2n + c_3n + c_4n + c_5n + c_{10}n + c_{11}n$$
$$(b) \cdot n = (c_2 + c_3 + c_4 + c_5 + c_{10} + c_{11}) \cdot n$$

- $c_{10}n - c_{10}$
- $c_{11}n - c_{11}$
- c_{12}

$$T(n) = \sum_{i=1}^{12} c_i \times T_i = a + bn$$



Best Case - Analysis

Cost x Times

- c_1
- $c_2n - c_2$
- $c_3n - c_3$
- $c_4n - c_4$
- $c_5n - c_5$

Total Cost

$$T(n) = \sum_{i=1}^{12} c_i \times T_i$$

$$a = c_1 - c_2 - c_3 - c_4 - c_5 - c_{10} - c_{11} + c_{12}$$

$$\begin{aligned} bn &= c_2n + c_3n + c_4n + c_5n + c_{10}n + c_{11}n \\ (b) \cdot n &= (c_2 + c_3 + c_4 + c_5 + c_{10} + c_{11}) \cdot n \end{aligned}$$

- $c_{10}n - c_{10}$
- $c_{11}n - c_{11}$
- c_{12}

$$T(n) = \sum_{i=1}^{12} c_i \times T_i = a + bn \approx bn \rightarrow O(n)$$



Algorithm - II

Bubble Sort(A)

```
1. n = Length[A];  
2. for j = 0 to n-2  
3.   swap = 0;  
4.   for i = 0 to n-j-2  
5.     if A[i] > A[i+1]  
6.       temp = A[i]  
7.       A[i] = A[i+1]  
8.       A[i+1] = temp  
9.       swap = swap + 1;  
10.      if swap = 0  
11.        break;  
12.  return A;
```

Analysis

The algorithm cares how sorted the input is therefore, the best case improves significantly.

- Best Case: $O(n)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$



Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Selection Sort – I

Dr. Bilal Wajid





Basic Idea – I

- *Find the smallest element in the array*





Basic Idea – II

- Find the smallest element in the array
- *Exchange it with the element in the first position*





Basic Idea – III

- 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*





Basic Idea – IV

- 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*





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
---	---	---	---	---	---	---





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Selection Sort – II

Dr. Bilal Wajid





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
---	---	---	---	---	---	---





Algorithm – I

SELECTION-SORT(A)

1. $n = \text{length}[A]$
2. $\text{for } j = 0 \text{ to } n - 2$
3. $\text{smallest} = j$
4. $\text{for } i = (j + 1) \text{ to } (n - 1)$
5. $\text{if } A[i] < A[\text{smallest}]$
6. $\text{smallest} = i$
7. $\text{swap } A[j] \leftrightarrow A[\text{smallest}]$

Costs

Times

Cost x Times





Algorithm – II

SELECTION-SORT(A)

```
1.  n = length[A]
2.  for j = 0 to n - 2
3.      smallest = j
4.      for i = (j + 1) to (n - 1)
5.          if A[ i ] < A [ smallest ]
6.              smallest = i
7.      swap A [ j ] ↔ A [ smallest ]
```

Costs	Times	Cost x Times
1. c_1	1. 1	1. c_1





Algorithm – III

SELECTION-SORT(A)

```
1.  n = length[A]
2.  for j = 0 to n - 2
3.      smallest = j
4.      for i = (j + 1) to (n - 1)
5.          if A[ i ] < A [ smallest ]
6.              smallest = i
7.      swap A [ j ] ↔ A [ smallest ]
```

Costs	Times	Cost x Times
1. c_1	1. 1	1. c_1
2. c_2	2. $(n-1)$	2. $c_2(n - 1)$





Algorithm – IV

SELECTION-SORT(A)

```
1. n = length[A]
2. for j = 0 to n - 2
3.     smallest = j
4.     for i = (j + 1) to (n - 1)
5.         if A[ i ] < A [ smallest ]
6.             smallest = i
7.     swap A [ j ] ↔ A [ smallest ]
```

Costs	Times	Cost x Times
1. .	1. .	1. .
2. c_2	2. $(n-1)$	2. $c_2(n - 1)$
3. c_3	3. $(n-1)$	3. $c_3(n - 1)$
4. .	4. .	4. .
5. .	5. .	5. .
6. .	6. .	6. .
7. c_7	7. $(n-1)$	7. $c_7(n - 1)$





Algorithm – III

SELECTION-SORT(A)

```
1.  n = length[A]
2.  for j = 0 to n - 2
3.      smallest = j
4.      for i = (j + 1) to (n - 1)
5.          if A[ i ] < A [ smallest ]
6.              smallest = i
7.      swap A [ j ] ↔ A [ smallest ]
```

Costs	Times	Cost x Times
1. .	1. .	1. .
2. .	2. .	2. .
3. .	3. .	3. .
4. c_4	4. $\sum_{j=0}^{n-2} T_j$	4. $c_4 \sum_{j=0}^{n-2} T_j$
5. c_5	5. $\sum_{j=0}^{n-2} T_j$	5. $c_5 \sum_{j=0}^{n-2} T_j$
6. c_6	6. $\sum_{j=0}^{n-2} T_j$	6. $c_6 \sum_{j=0}^{n-2} T_j$
7. .	7. .	7. .





Algorithm – III

Costs	Times	Cost x Times
1. c_1	1. 1	1. c_1
2. c_2	2. $(n-1)$	2. $c_2(n-1)$
3. c_3	3. $(n-1)$	3. $c_3(n-1)$
4. c_4	4. $\sum_j^{n-2} T_j$	4. $c_4 \sum_j^{n-2} T_j$
5. c_5	5. $\sum_j^{n-2} T_j$	5. $c_5 \sum_j^{n-2} T_j$
6. c_6	6. $\sum_j^{n-2} T_j$	6. $c_6 \sum_j^{n-2} T_j$
7. c_7	7. $(n-1)$	7. $c_7(n-1)$





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Merge Sort – I

Dr. Bilal Wajid





Types of Sorting Algorithms

- Non-recursive/Incremental comparison sorting
 - Selection sort
 - Bubble sort
 - Insertion sort
- **Recursive comparison sorting**
 - Merge sort
 - Quick sort
 - Heap sort
- Non-comparison linear sorting
 - Count sort
 - Radix sort
 - Bucket sort





Merge Sort – Basic Idea – I

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.





Merge Sort – Basic Idea – II

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.
- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the smaller sub-lists, then recombine them in a sorted manner.





Merge Sort – Basic Idea – III

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each.
 - **Conquer:**
 - **Combine:**





Merge Sort – Basic Idea – IV

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each.
 - **Conquer:** Sort the two subsequences to produce the sorted answer.
 - **Combine:**





Merge Sort – Basic Idea – V

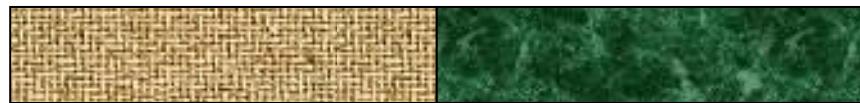
- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each.
 - **Conquer:** Sort the two subsequences to produce the sorted answer.
 - **Combine:** Merge the two sorted sub sequences to produce the sorted answer.





Merge Sort – Basic Idea – VI

Divide, Conquer, Combine



mergeSort(0, n/2-1)



sort

mergeSort(n/2, n-1)



sort

merge(0, n/2, n-1)





Merge Sort – Pseudocode – I

- Merge-Sort (A, p, r)
 - If $p < r$
 - $q = \text{floor}\left(\frac{p+r}{2}\right)$
 - Merge-Sort (A, p, q)
 - Merge-Sort (A, q+1, r)
 - Merge (A, p, r)





Merge Sort – Pseudocode – II

- Merge-Sort (A, p, r)



- If $p < r$

- $q = \text{floor} \left(\frac{p+r}{2} \right)$

- Merge-Sort (A, p, q)



- Merge-Sort (A, q+1, r)



- Merge (A, p, r)

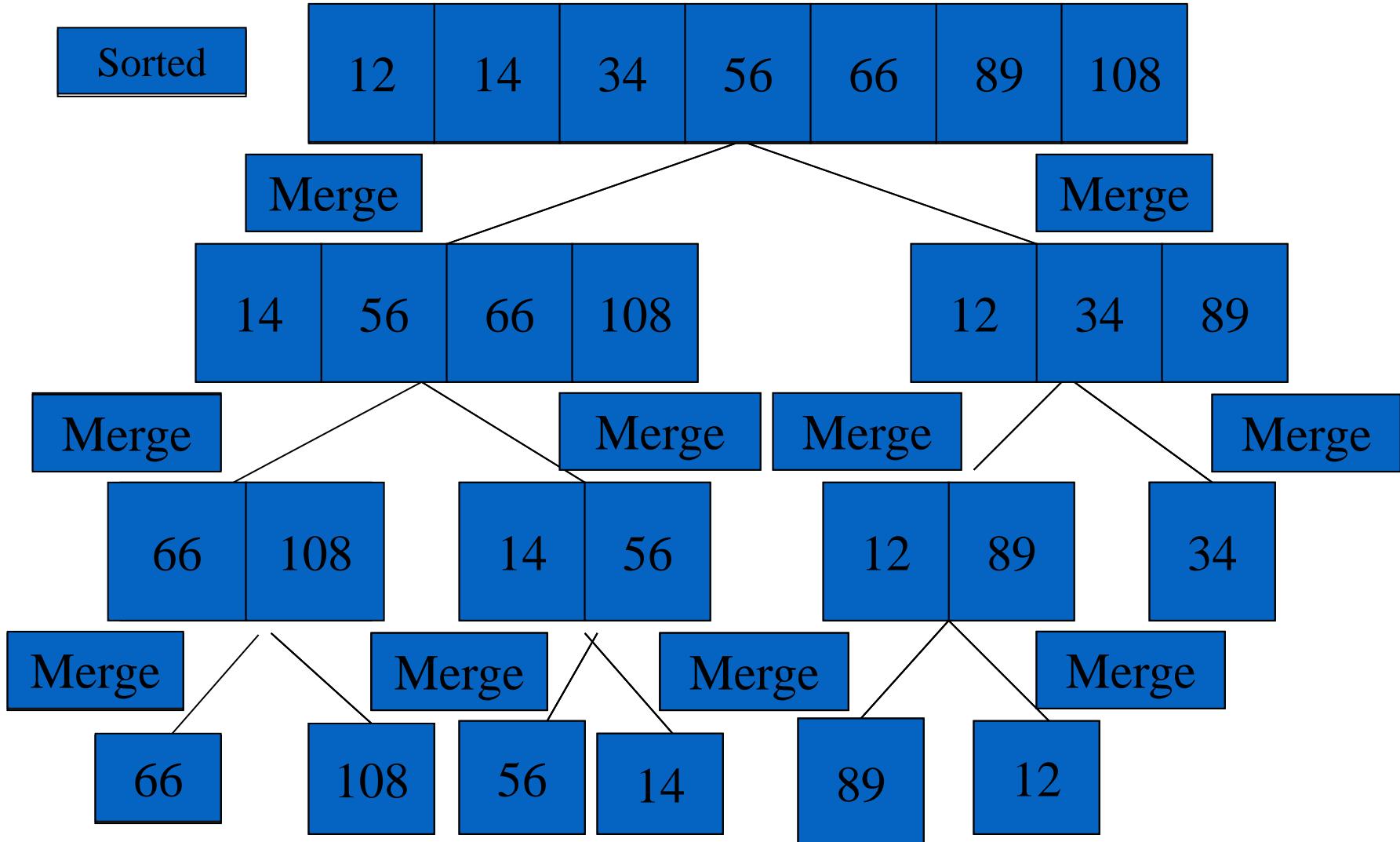
Recursive component





Merge Sort – Example (Idea I)

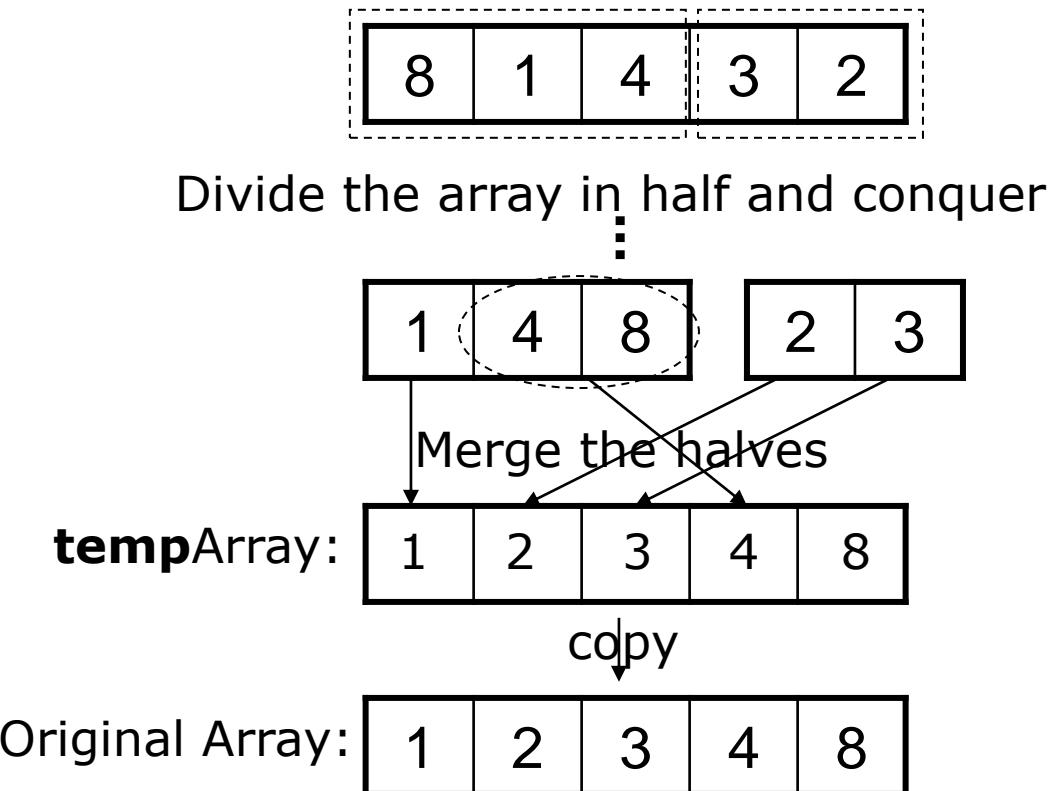
Base Case: When the sequences to be sorted has length 1.





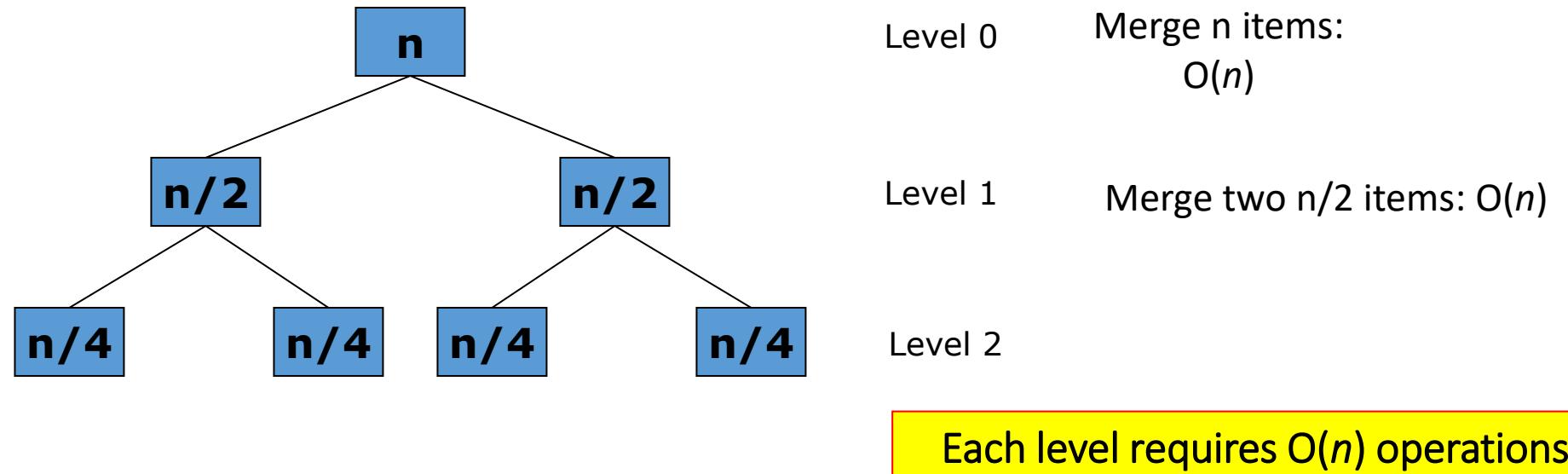
Merge Function

Given two sorted arrays, *merge* operation produces a sorted array with all the elements of the two arrays





Merge Sort Analysis – I



Tree Height : $\log_2 n$

Each level $O(n)$ operations & $O(\log_2 n)$ levels $\rightarrow O(n \times \log_2 n)$





Merge Sort Analysis – II

- Best Case: $O(n \times \log_2 n)$
- Average case: $O(n \times \log_2 n)$
- Worst case: $O(n \times \log_2 n)$





Merge Sort Analysis – III

- Best Case: $O(n \times \log_2 n)$
- Average case: $O(n \times \log_2 n)$
- Worst case: $O(n \times \log_2 n)$
- Performance is independent of the initial order of the array items.





Merge Sort Analysis – IV

- Best Case: $O(n \times \log_2 n)$
- Average case: $O(n \times \log_2 n)$
- Worst case: $O(n \times \log_2 n)$
- Performance is independent of the initial order of the array items.
- **Advantage:** Mergesort is an extremely fast algorithm.
- **Disadvantage:** Mergesort requires a second array as large as the original.





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Merge Sort – II

Dr. Bilal Wajid





Types of Sorting Algorithms

- Non-recursive/Incremental comparison sorting
 - Selection sort
 - Bubble sort
 - Insertion sort
- **Recursive comparison sorting**
 - Merge sort
 - Quick sort
 - Heap sort
- Non-comparison linear sorting
 - Count sort
 - Radix sort
 - Bucket sort





Merge Sort – Basic Idea – I

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.





Merge Sort – Basic Idea – II

- **Idea I:** Recursively dividing the list in half until each sub-list has one element, then recombine them in a sorted manner.
- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the **smaller sub-lists**, then recombine them in a sorted manner.





Merge Sort – Basic Idea – III

- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the smaller sub-lists, then recombine them in a sorted manner.





Merge Sort – Basic Idea – IV

- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the **smaller sub-lists**, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each. Continue dividing until only a few elements remain.
 - **Conquer (Sort):**
 - **Combine (Merge):**





Merge Sort – Basic Idea – V

- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the smaller sub-lists, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each. Continue dividing until only a few elements remain.
 - **Conquer (Sort):** Sort the two subsequences.
 - **Combine (Merge):**





Merge Sort – Basic Idea – VI

- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the smaller sub-lists, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each. Continue dividing until only a few elements remain.
 - **Conquer (Sort):** Sort the two subsequences.
 - **Combine (Merge):** Merge the two sorted sub sequences to produce the sorted answer.





Merge Sort – Basic Idea – VII

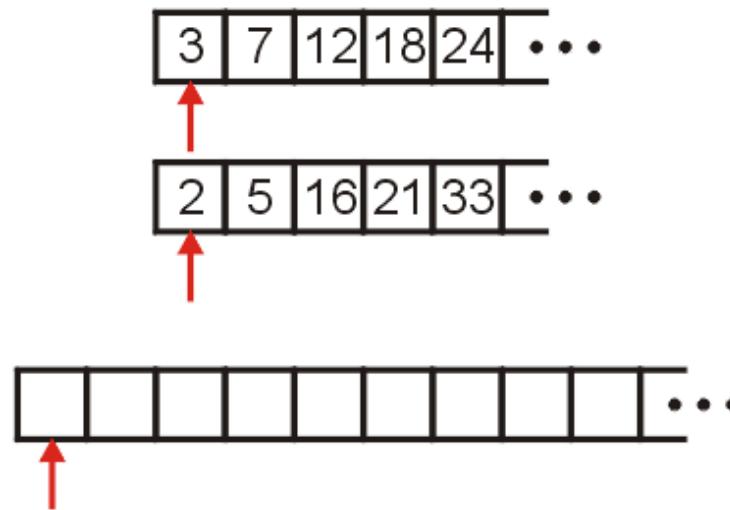
- **Idea II:** Recursively dividing the list in half until each sub-list has **only a small number of elements**. Then sort the smaller sub-lists, then recombine them in a sorted manner.
 - **Divide:** the n element sequence to be sorted into two subsequences of $n/2$ elements each. Continue dividing until only a few elements remain.
 - **Conquer (Sort):** Sort the two subsequences.
 - **Combine (Merge):** Merge the two sorted sub sequences to produce the sorted answer.
 - **How do you Merge Sorted Lists?**





Merging Example – I

Consider the two sorted arrays and an empty array



Define three indices at the start of each array

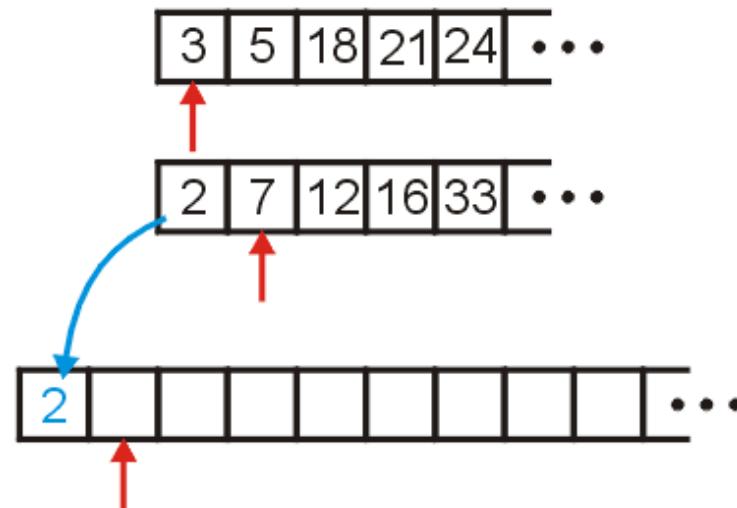




Merging Example – II

We compare 2 and 3: $(2 < 3)$

- Copy 2 down
- Increment the corresponding indices

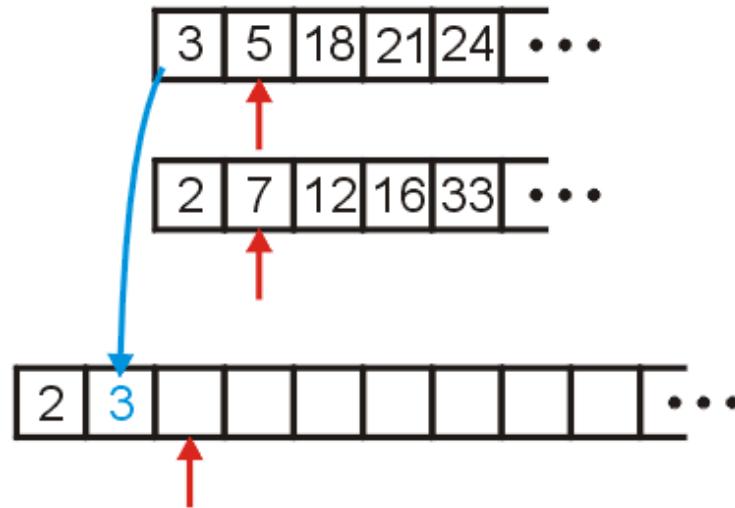




Merging Example – III

We compare 3 and 7

- Copy 3 down
- Increment the corresponding indices

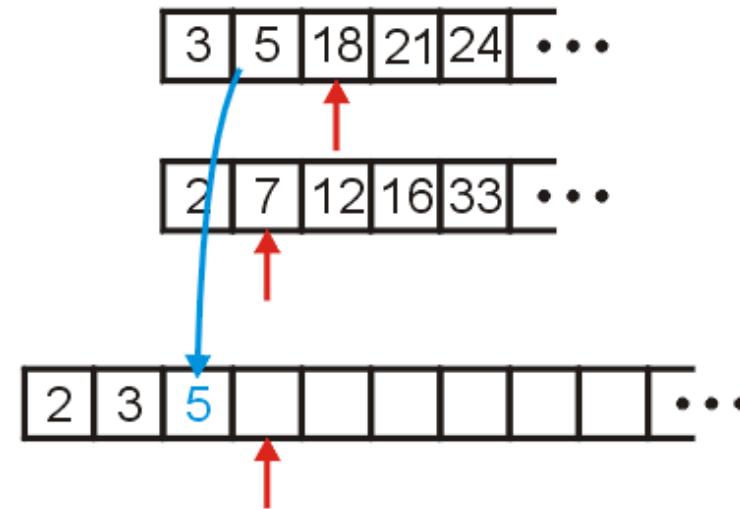




Merging Example – IV

We compare 5 and 7

- Copy 5 down
- Increment the appropriate indices

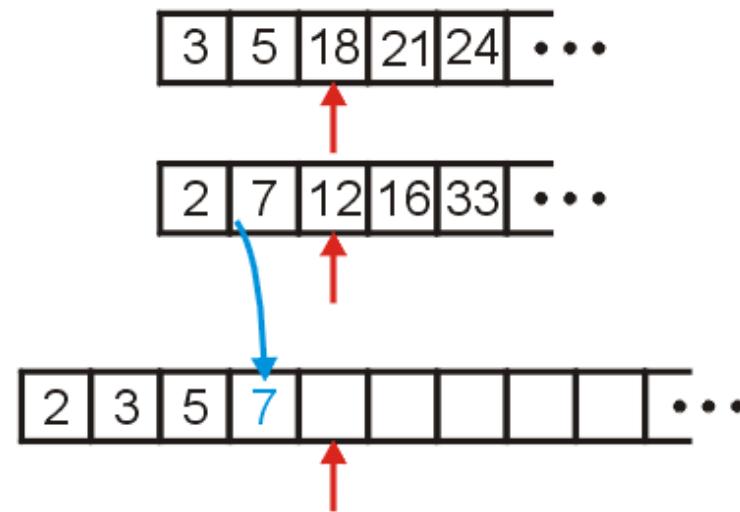




Merging Example – V

Compare 18 & 7

- Copy 7 down
- Increment...

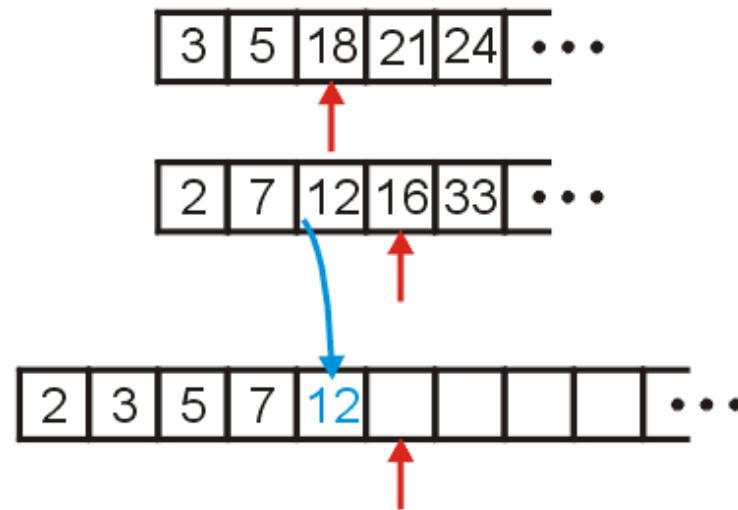




Merging Example – VI

Compare 18 & 12

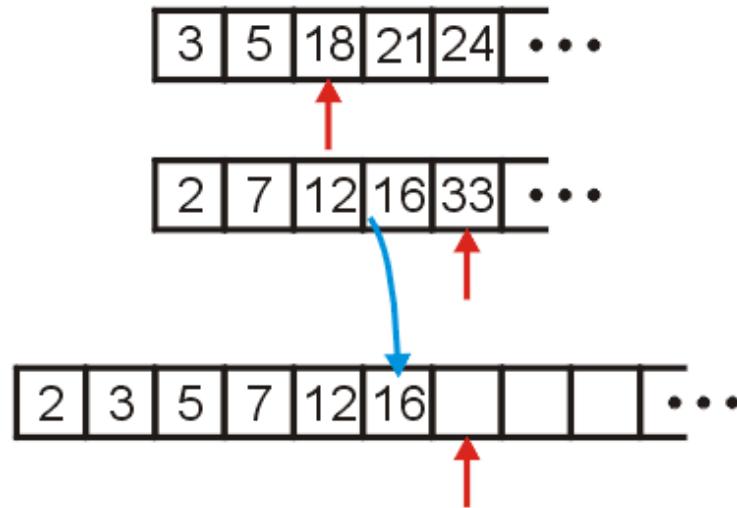
- Copy 12 down
- Increment...





Merging Example – VII

Compare 18 and 16
• Copy 16 down
• Increment...

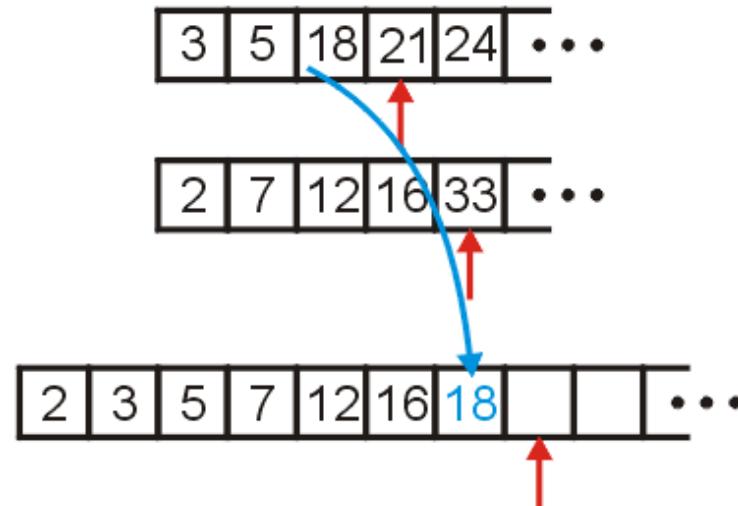




Merging Example – VIII

Compare 18 & 33

- Copy 18 down
- Increment...

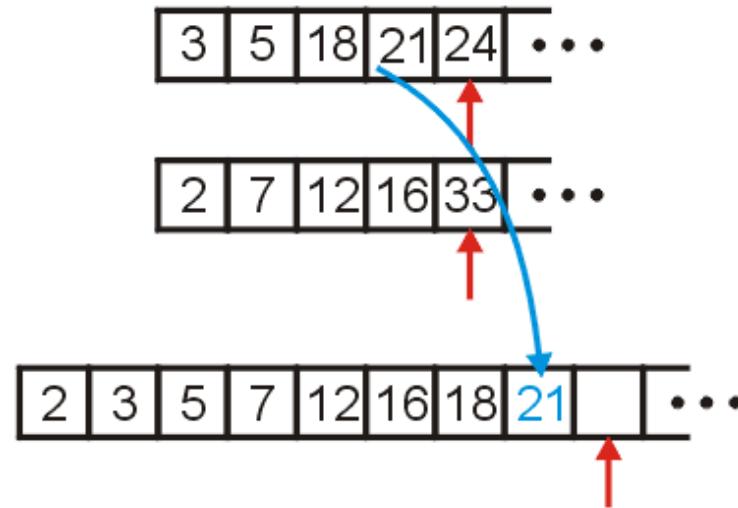




Merging Example – IX

Compare 21 & 33

- Copy 21 down
- Increment...

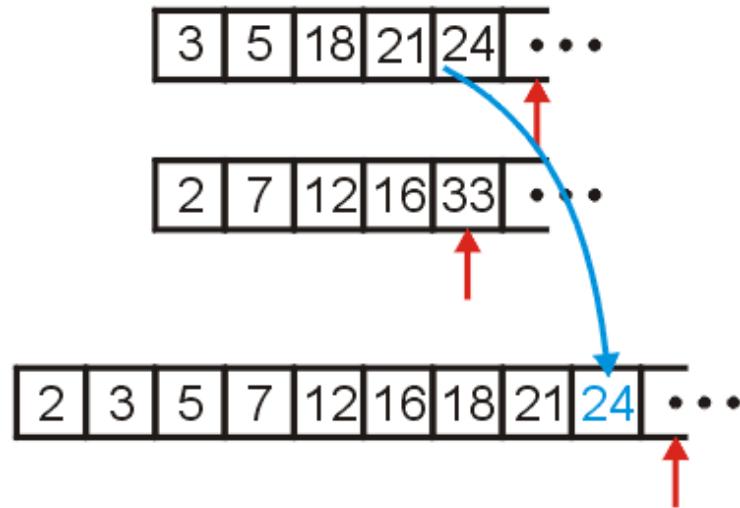




Merging Example – X

Compare 24 & 33

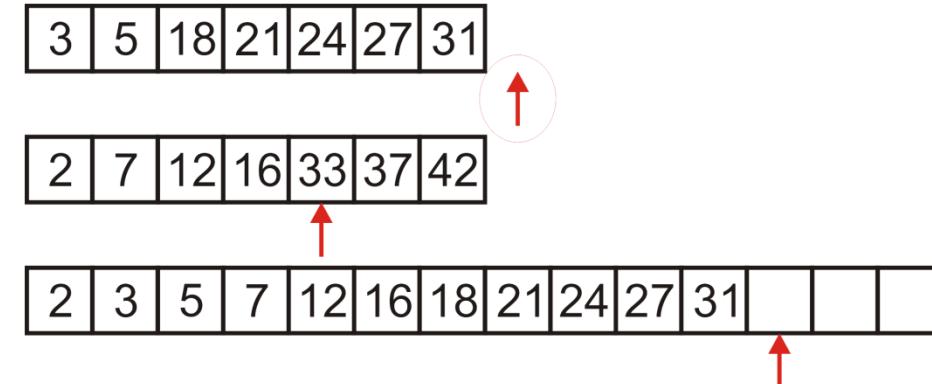
- Copy 24 down
- Increment...



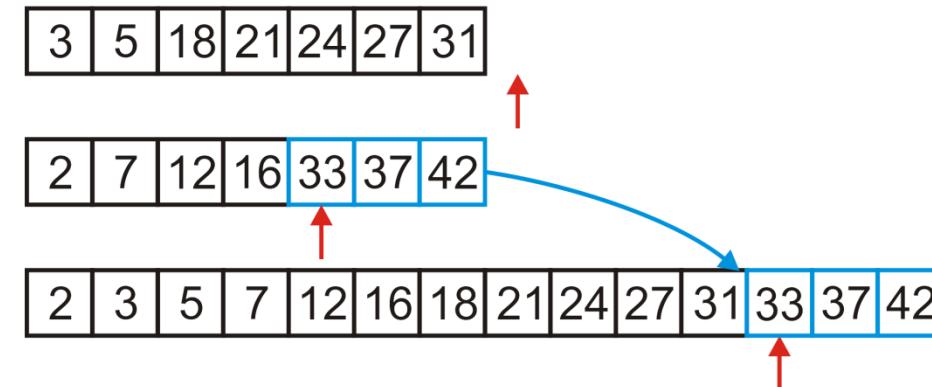


Merging Example – XI

We would continue until we have passed beyond the limit of one of the two arrays



After this, we simply copy over all remaining entries in the non-empty array





Analysis of merging – I

In order to merge two lists of size n_1, n_2

- The body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $O(n_1 + n_2)$ time





Analysis of merging – II

In order to merge two lists of size n_1, n_2

- The body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $O(n_1 + n_2)$ time

If the arrays are approximately the same size, $n = n_1$ and $n_1 \approx n_2$, we can say that:

- run time is $O(n + n) \rightarrow O(2n) \rightarrow O(n)$





Analysis of merging – III

In order to merge two lists of size n_1, n_2

- The body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $O(n_1 + n_2)$ time

If the arrays are approximately the same size, $n = n_1$ and $n_1 \approx n_2$, we can say that:

- run time is $O(n + n) \rightarrow O(2n) \rightarrow O(n)$

Problem: We cannot merge two arrays in-place

- This algorithm always required the allocation of a new array
- Therefore, the memory requirements are also $O(n)$



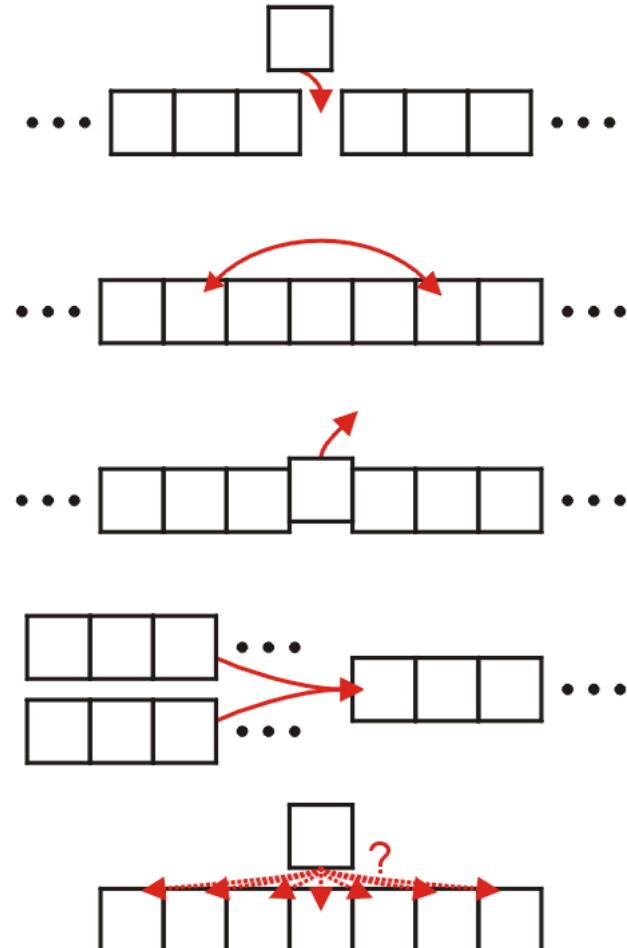


Types of Sorting Algorithm

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- **Merging**
- Distribution

Clearly merge sort falls into the fourth category





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Merge Sort – III

Dr. Bilal Wajid



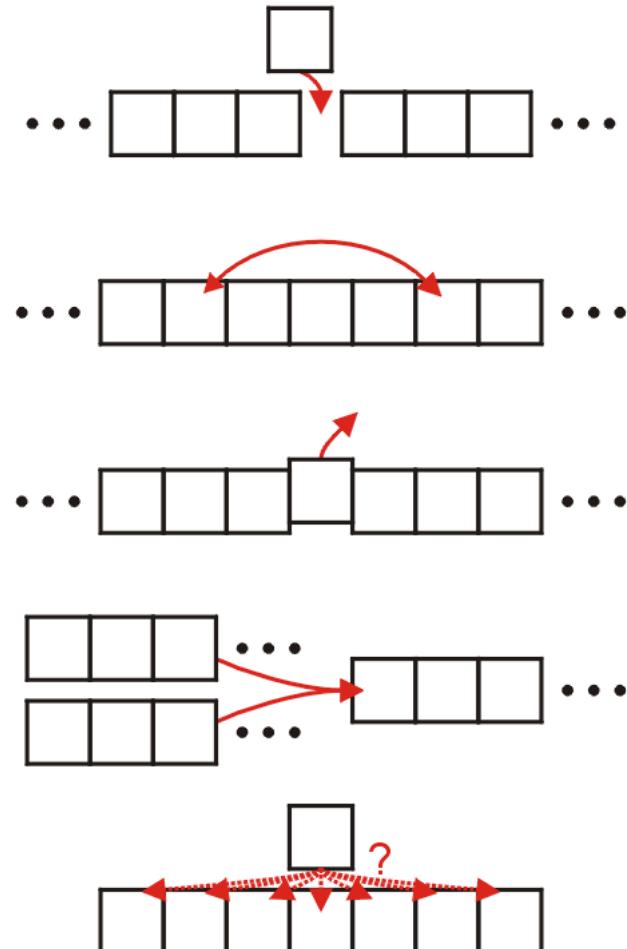


Types of Sorting Algorithm

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- **Merging**
- Distribution

Clearly merge sort falls into the fourth category





The Algorithm – I

Question:

- we split the list into two sub-lists and sorted them
- how should we sort those lists?

Answer (theoretical):

- if the size of these sub-lists is > 1 , use merge sort again
- if the sub-lists are of length 1, do nothing: a list of length one is sorted





The Algorithm – II

However, just because an algorithm has excellent asymptotic properties, this does not mean that it is practical at all levels

Answer (practical):

- If the sub-lists are less than some threshold length, **use an algorithm like insertion sort** to sort the lists
- Otherwise, use merge sort, again





Implementation

The actual body is quite small:

```
template <typename Type>
void merge_sort( Type *array, int first, int last )
{
    if ( last - first <= N )
        insertion_sort( array, first, last );
    else
    {
        int midpoint = (first + last)/2;

        merge_sort( array, first, midpoint );
        merge_sort( array, midpoint, last );
        merge( array, first, midpoint, last );
    }
}
```





Example

Consider the following is of unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less





Example

We call `merge_sort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

`merge_sort(array, 0, 25)`





Example

We are calling `merge_sort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
```

`merge_sort(array, 0, 25)`





Example

We are now executing `merge_sort(array, 0, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $12 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
```

```
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

We are now executing `merge_sort(array, 0, 6)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $6 - 0 \leq 6$, so find we call insertion sort

```
merge_sort( array, 0, 6 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 0 to 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 0, 6 )
merge_sort( array, 0, 6 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 0 to 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

- This function call completes and so we exit

```
insertion_sort( array, 0, 6 )  
merge_sort( array, 0, 6 )  
merge_sort( array, 0, 12 )  
merge_sort( array, 0, 25 )
```





Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array, 0, 6 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 0, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
```

```
merge_sort( array, 0, 6 );
```

```
merge_sort( array, 6, 12 );
```

```
merge_sort( array, 0, 12 )
```

```
merge_sort( array, 0, 25 )
```





Example

We are now executing `merge_sort(array, 6, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $12 - 6 \leq 6$, so find we call insertion sort

```
merge_sort( array, 6, 12 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 6, 12 )  
merge_sort( array, 6, 12 )  
merge_sort( array, 0, 12 )  
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

- This function call completes and so we exit

```
insertion_sort( array, 6, 12 )
merge_sort( array, 6, 12 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array, 6, 12 )
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 0, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

```
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 0, 6, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together

```
merge( array, 0, 6, 12 )  
merge_sort( array, 0, 12 )  
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 0, 6, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together

- This function call exists

```
merge( array, 0, 6, 12 )  
merge_sort( array, 0, 12 )  
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 0, 12)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We are finished calling this function as well

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

Consequently, we exit

```
merge_sort( array, 0, 12 )
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
```

`merge_sort(array, 0, 25)`





Example

We are now executing `merge_sort(array, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 12 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (12 + 25)/2; // == 18
```

```
merge_sort( array, 12, 18 );
```

```
merge_sort( array, 12, 25 )
```

```
merge_sort( array, 0, 25 )
```





Example

We are now executing `merge_sort(array, 12, 18)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $18 - 12 \leq 6$, so find we call insertion sort

```
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 12 to 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 12, 18 )  
merge_sort( array, 12, 18 )  
merge_sort( array, 12, 25 )  
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 12 to 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

- This function call completes and so we exit

```
insertion_sort( array, 12, 18 )
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

```
merge_sort( array, 12, 18 )  
merge_sort( array, 12, 25 )  
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

We continue calling

```
midpoint = (12 + 25)/2; // == 18
```

```
merge_sort( array, 12, 18 );
```

```
merge_sort( array, 18, 25 );
```

```
merge_sort( array, 12, 25 )
```

```
merge_sort( array, 0, 25 )
```





Example

We are now executing `merge_sort(array, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

First, $25 - 18 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 18, midpoint );
merge_sort( array, 12, 25 );
merge_sort( array, 0, 25 );
```





Example

We are now executing `merge_sort(array, 18,`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

Now, $21 - 18 \leq 6$, so find we call insertion sort

```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 18 to 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 18 to 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

- This function call completes and so we exit

```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

We continue calling

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We are now executing `merge_sort(array, 21, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

Now, $25 - 21 \leq 6$, so find we call insertion sort

```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

We continue calling

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 18, 21, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

These two sub-arrays are merged together

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 18, 21, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

These two sub-arrays are merged together

- This function call exists

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

We are finished calling this function as well

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

Consequently, we exit

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

We continue calling

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 12, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

These two sub-arrays are merged together

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 12, 18, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

These two sub-arrays are merged together

- This function call exists

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

We are finished calling this function as well

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

Consequently, we exit

```
merge_sort( array, 12, 25 )
merge_sort( array, 0, 25 )
```





Example

We return to continue executing `merge_sort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

We continue calling

```
midpoint = (0 + 25)/2; // == 12
```

```
merge_sort( array, 0, 12 );
```

```
merge_sort( array, 12, 25 );
```

```
merge( array, 0, 12, 25 );
```

```
merge_sort( array, 0, 25 )
```





Example

We are executing `merge(array, 0, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

These two sub-arrays are merged together

`merge(array, 0, 12, 25)`
`merge_sort(array, 0, 25)`





Example

We are executing `merge(array, 0, 12, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

These two sub-arrays are merged together

- This function call exists

```
merge( array, 0, 12, 25 )  
merge_sort( array, 0, 25 )
```





Example

We return to executing `merge_sort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

We are finished calling this function as well

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

Consequently, we exit

```
merge_sort( array, 0, 25 )
```





Example

The array is now sorted

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

- Question: What is the run-time of this algorithm?





Comments

In practice, merge sort is faster than heap sort, though they both have the same asymptotic run times

Merge sort requires an additional array

- Heap sort does not require

Next we see quick sort

- Faster, on average, than either heap or quick sort
- Requires $\Theta(n)$ additional memory

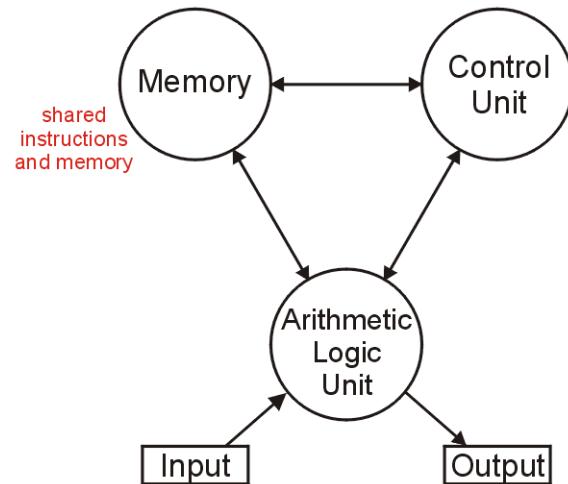




Merge Sort

The (likely) first implementation of merge sort was on the ENIAC in 1945 by John von Neumann

- The creator of the *von Neumann architecture* used by all modern computers:



http://en.wikipedia.org/wiki/Von_Neumann





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Heap Sort – I

Dr. Bilal Wajid





Heap Sort

Uses *Heap* data structure to manage data during algorithm execution

A *(Max/Min)-Heap* is a balanced, left-justified binary tree in which no node has a value *greater/lesser* than the value in its parent

- $A[\text{parent}[i]] \geq A[i]$ (max-heap)
- $A[\text{parent}[i]] \leq A[i]$ (min-heap)

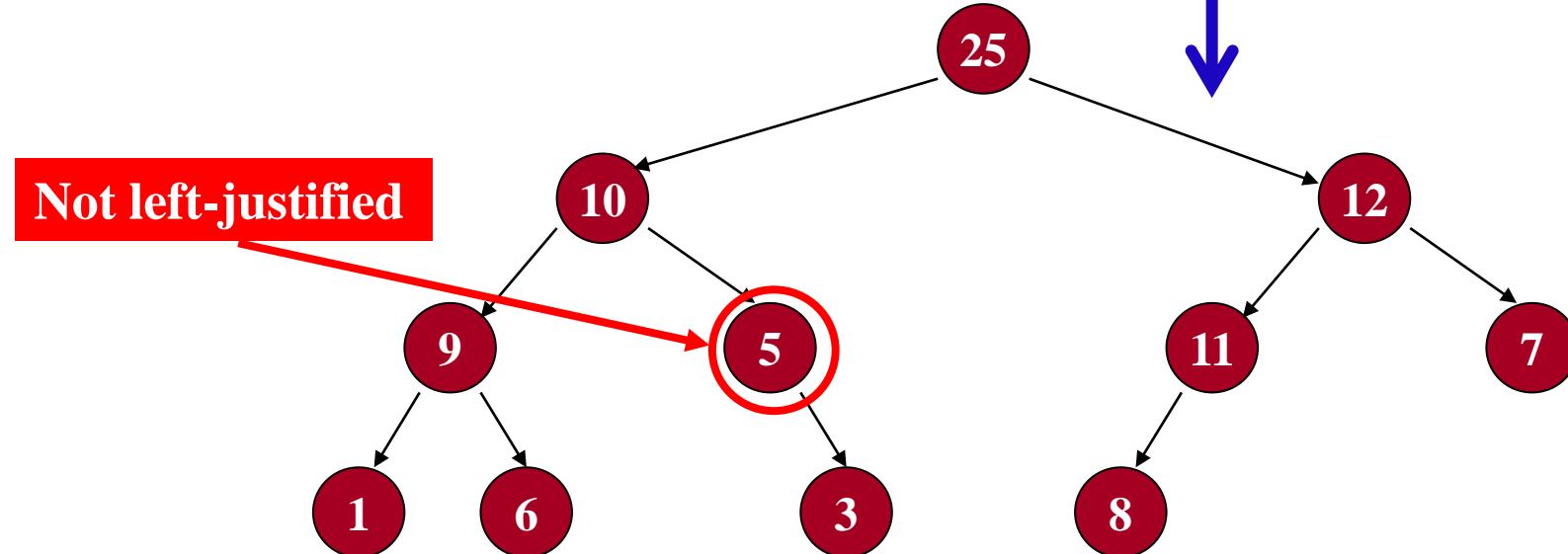
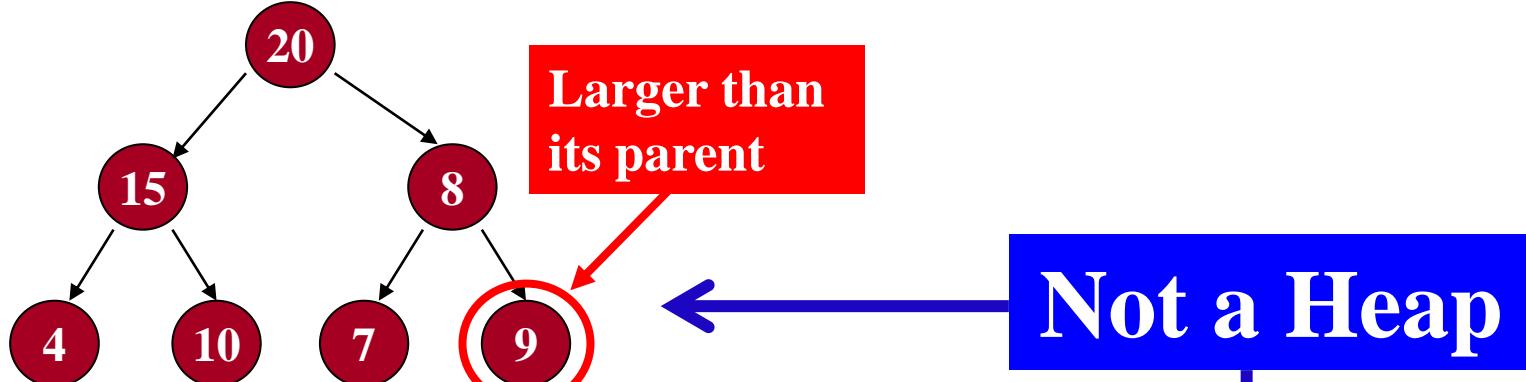
Largest/Smallest element is *stored at the root*.

Subtree rooted at a node contains values no *larger/smaller* than that at the node.





Heap Property



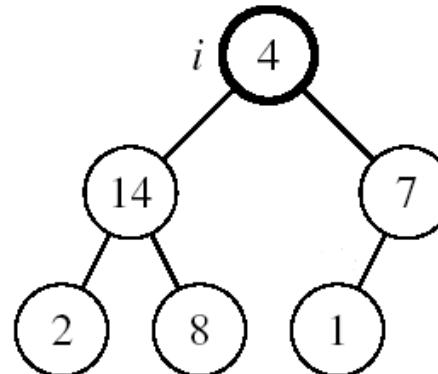


Maintaining the Heap Property

Suppose a node is smaller than a child, violating heap property. To maintain heap property, exchange the node with the larger child, moving the node down the tree until it is not 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 $A[r] > A[\text{largest}]$ *Complexity of MAX-HEAPIFY () ?*
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$ *$O(\lg n)$ where $\lg n$ is the height of the tree*
10. MAX-HEAPIFY($A, \text{largest}, n$)





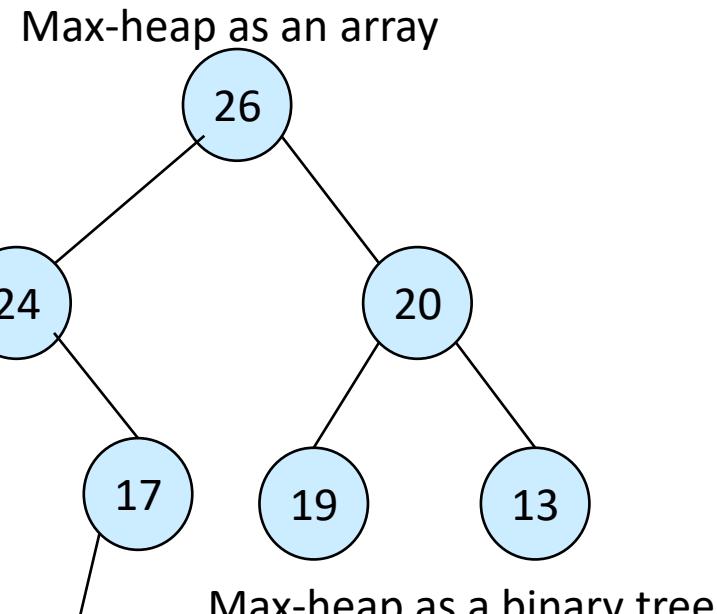
Array Representation of Heaps

A heap can be stored as an array A .

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

- 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



Node Insertion: Insert in last row from left to right
Node Deletion: Delete from last row from right to left





Building a Heap

A given array containing n elements can be converted into a max-heap by applying MAX-HEAPIFY on all interior nodes

- The elements in the subarray $A[1 \dots \lfloor n/2 \rfloor]$ are interior nodes
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves

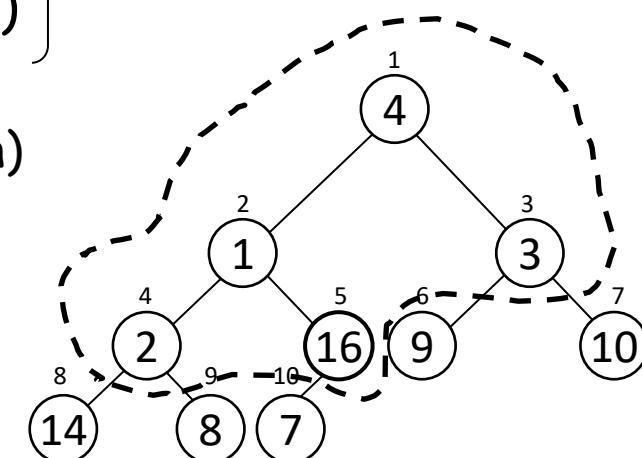
Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
 2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
 3. **do** MAX-HEAPIFY(A, i, n) $O(\lg n)$
- $O(n)$

Complexity of BUILD-MAX-HEAP? $O(n \lg n)$

A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

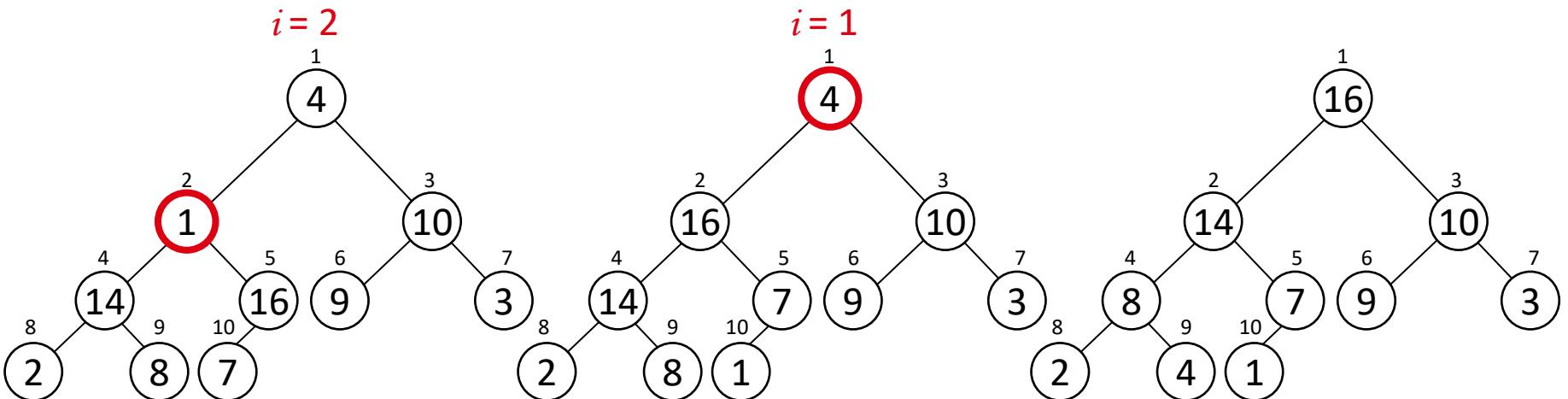
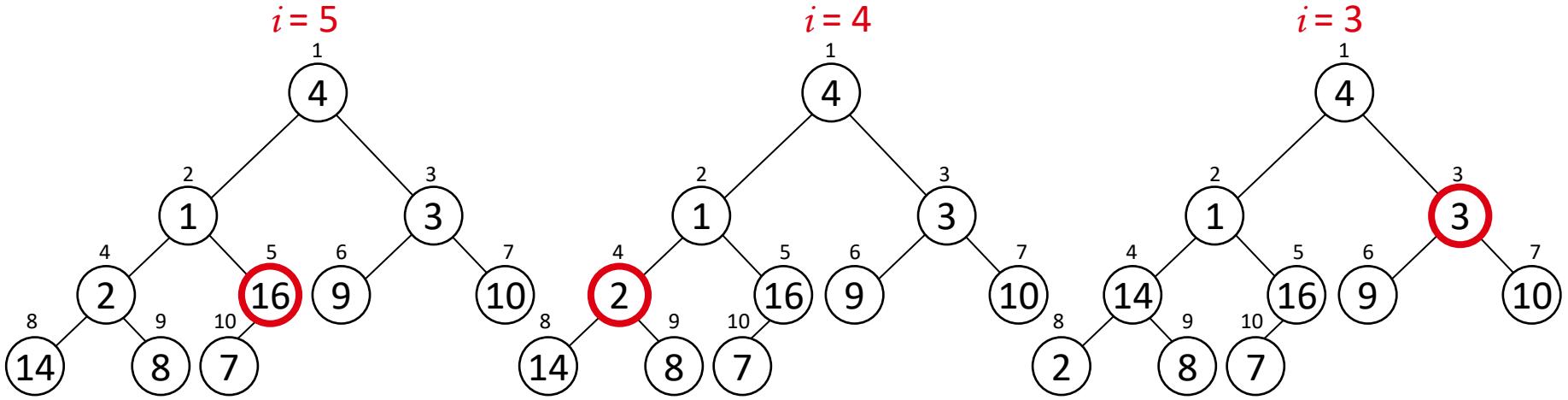




Example:

A:

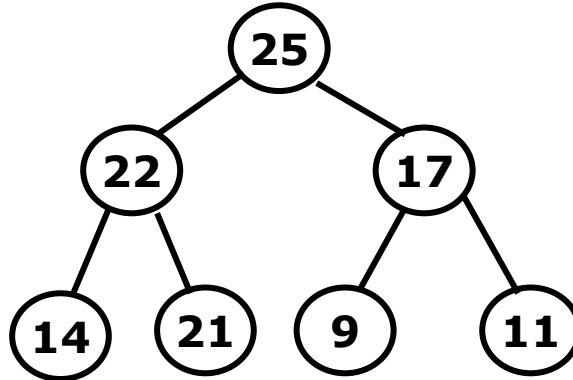
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---





Heap Sort

A given array which has been heapified does not mean it is sorted



Notice that the largest number is at the root. It can be swapped with the rightmost leaf at the deepest level (its original position) and heap size decreased.

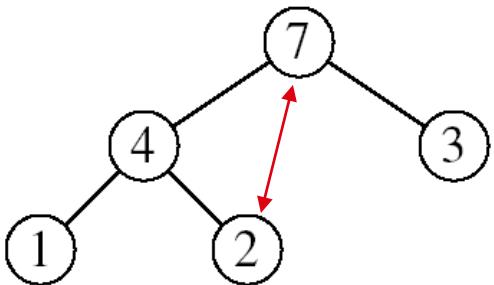
The new root has lost heap property. **Heapify** the new root and repeat this process until only one node remain.



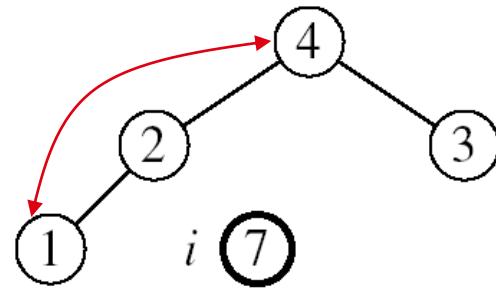


Example:

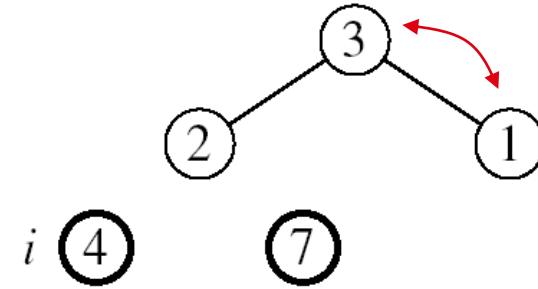
$A = [7, 4, 3, 1, 2]$



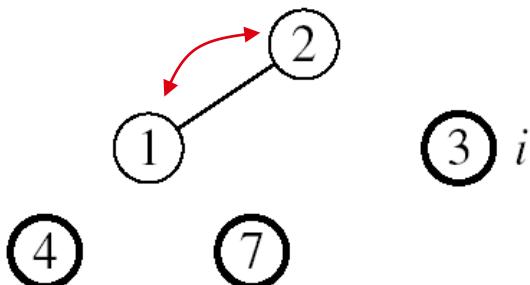
MAX-HEAPIFY($A, 1, 4$)



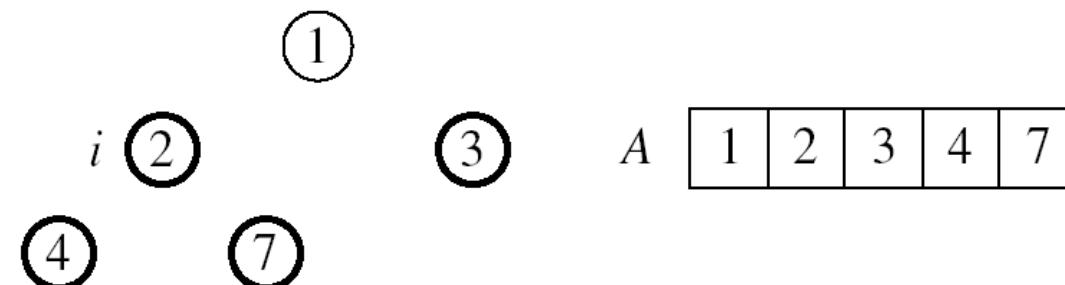
MAX-HEAPIFY($A, 1, 3$)



MAX-HEAPIFY($A, 1, 2$)



MAX-HEAPIFY($A, 1, 1$)





Heap Sort Pseudocode

HEAPSORT(A)

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

*What is the running time of **HEAPSORT ()** ? $O(n \lg n)$*

Why Heap Sort?

- Heap Sort is always $O(n \lg n)$ and is in-place
- Quicksort is usually $O(n \lg n)$ but in the worst case slows to $O(n^2)$
- Quicksort is generally faster, but Heapsort is better in time-critical applications





How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort → $O(n^2)$
- Merge Sort, Quick Sort, Heap Sort → $O(n \lg n)$
- What is common to all these algorithms?
 - Make **comparisons** between input elements
- Lower bound for comparison based sorting
 - **Theorem:** To sort n elements, comparison sorts **must** make $\Omega(n \lg n)$ comparisons in the worst case.
 - **Proof:** Draw decision tree and calculate asymptotic height of any decision tree for sorting n elements



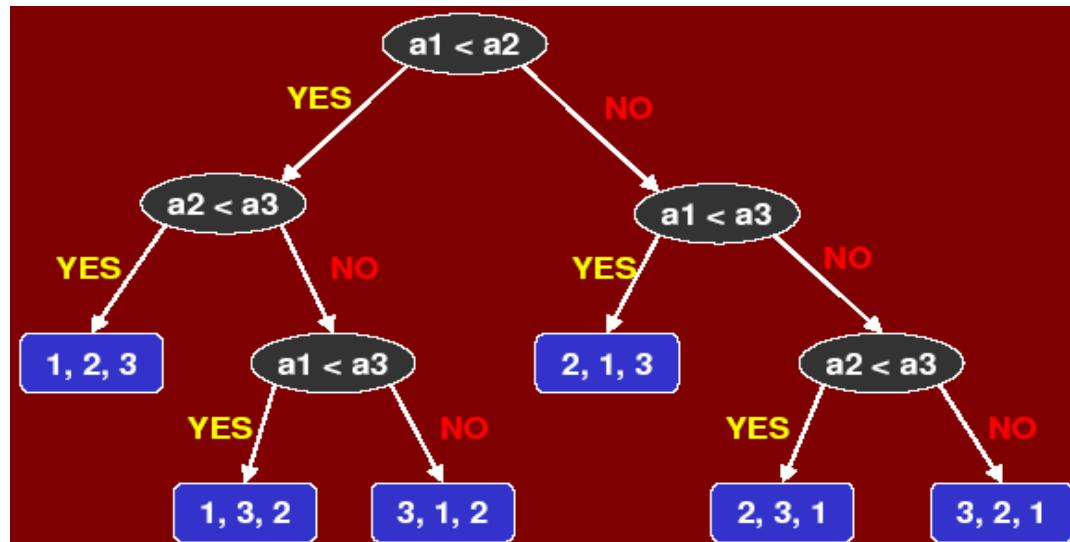


Lower Bound For Comparison Sorting

Consider sorting three numbers a_1, a_2, a_3 .

There are $3! = 6$ possible combinations:

$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_2, a_1) (a_3, a_1, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1)$



- *What's the minimum # of leaves? $n!$*
- *What's the maximum # of leaves of a binary tree of height h ? 2^h*
- minimum # of leaves \leq maximum # of leaves





Lower Bound For Comparison Sorting

- So we have...

$$n! \leq 2^h$$

- Taking logarithms:

$$\lg(n!) \leq h$$

- Stirling's approximation tells us: $n! > \left(\frac{n}{e}\right)^n$ $h \geq \lg\left(\frac{n}{e}\right)^n$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$ therefore the time to comparison sort n elements is $\Omega(n \lg n)$

- Corollary: Quick Sort, Heap Sort and Merge Sort are asymptotically optimal comparison sorts. *Can we do any better than $\Omega(n \lg n)$?*





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Heap Sort- II

Dr. Bilal Wajid





Heap Sort

Recall that inserting n objects into a min-heap and then taking n objects will result in them coming out in order

Strategy: given an unsorted list with n objects, place them into a heap, and take them out





In-place Implementation

Problem:

- This solution requires additional memory, that is, a min-heap of size n

This requires $\Theta(n)$ memory and is therefore not in place

Is it possible to perform a heap sort in place, that is, require at most $\Theta(1)$ memory (a few extra variables)?

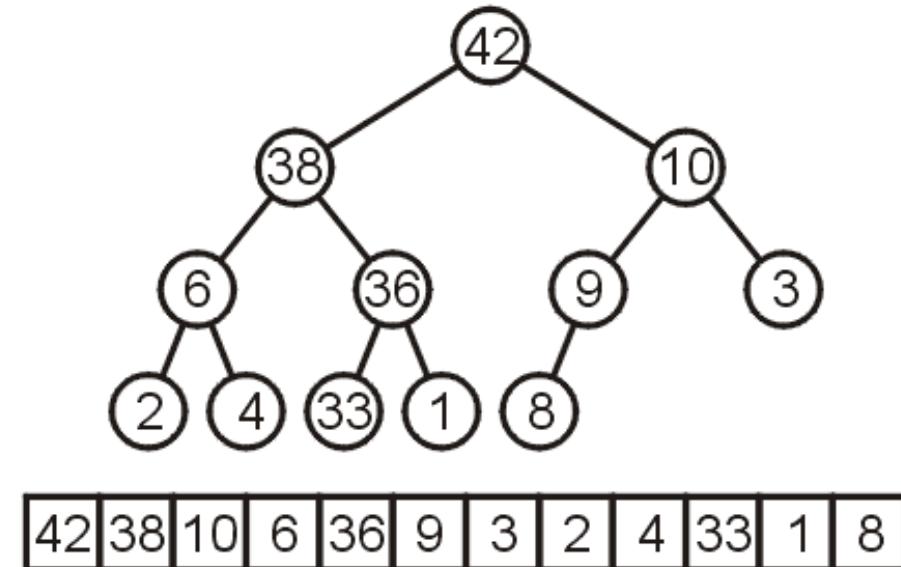




In-place Implementation

Instead of implementing a min-heap, consider a max-heap:

- A heap where the maximum element is at the top of the heap and the next to be popped



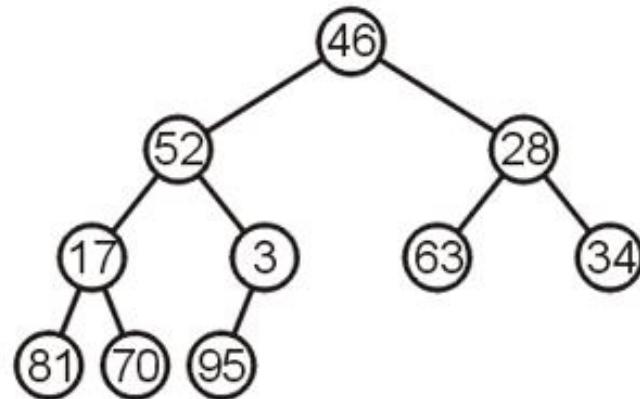


In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95
----	----	----	----	---	----	----	----	----	----

This array represents the following complete tree:



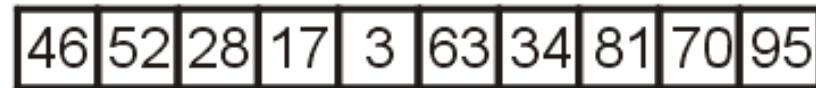
This is neither a min-heap, max-heap, or binary search tree



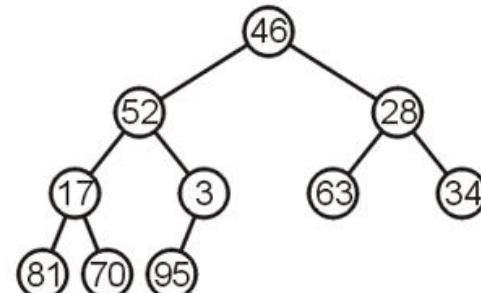


In-place Heapification

Now, consider this unsorted array:



Additionally, because arrays start at 0 (we started at entry 1 for binary heaps) , we need different formulas for the children and parent



The formulas are now:

$$\text{Children} \quad 2*k + 1 \quad 2*k + 2$$

$$\text{Parent} \quad (k + 1)/2 - 1$$



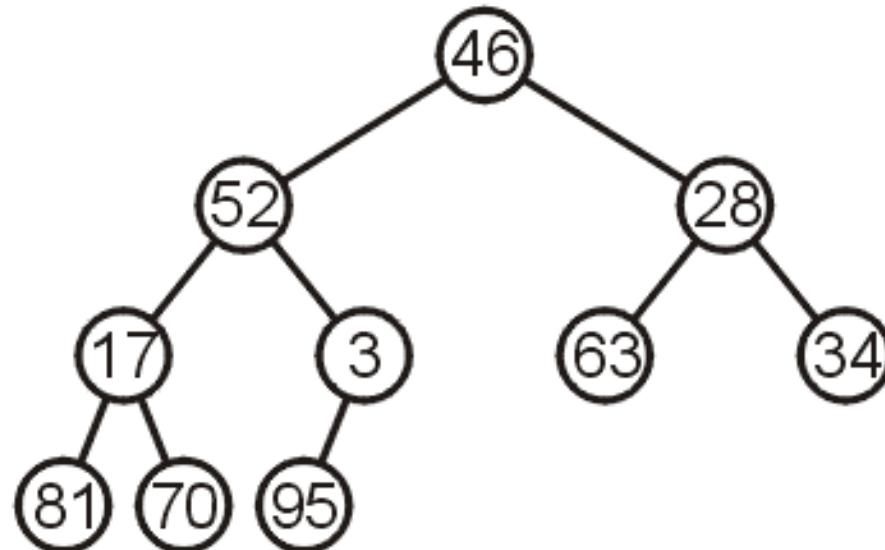


In-place Heapification

Can we convert this complete tree into a max heap?

Restriction:

- The operation must be done in-place

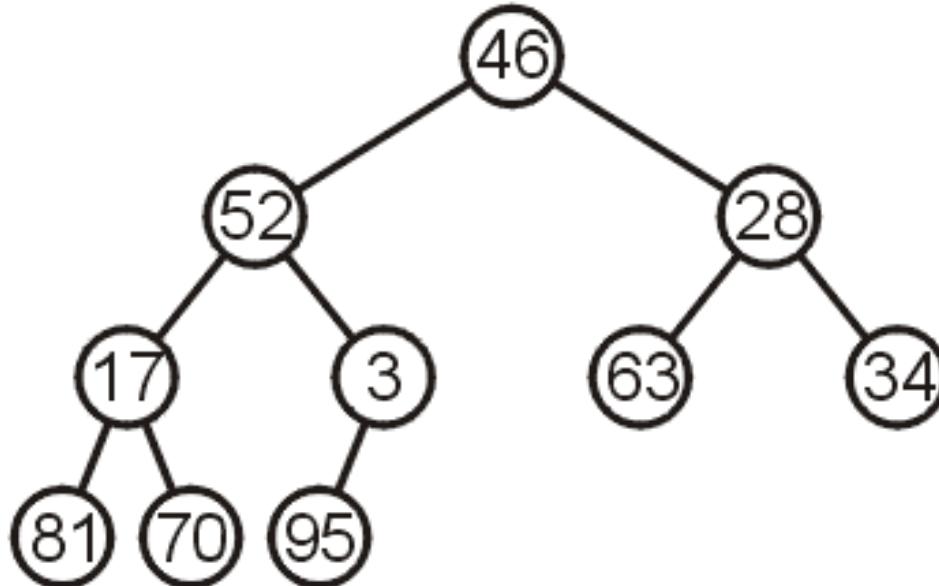




In-place Heapification

Two strategies:

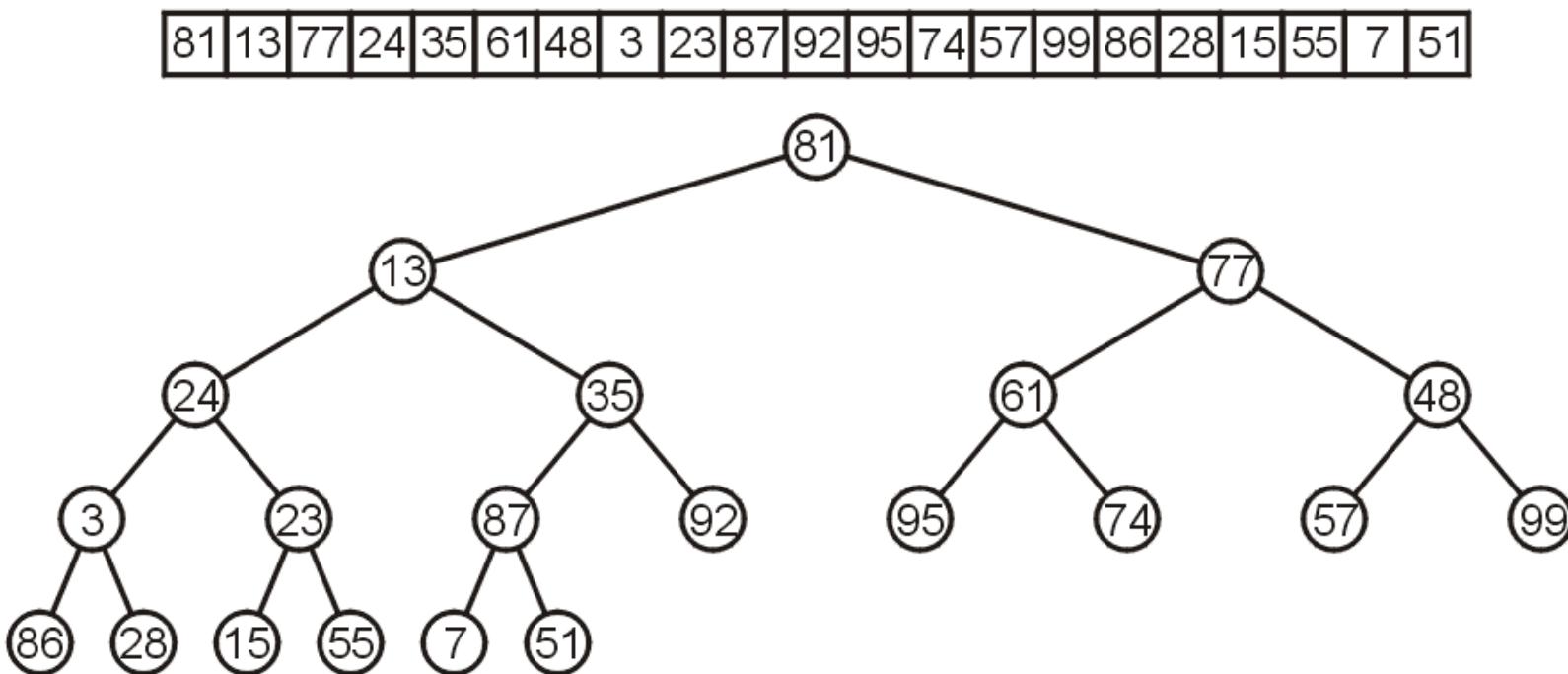
- Assume 46 is a max-heap and keep inserting the next element into the existing heap (similar to the strategy for insertion sort)
- Start from the back: note that all leaf nodes are already max heaps, and then make corrections so that previous nodes also form max heaps





In-place Heapification

Let's work bottom-up: each leaf node is a max heap on its own

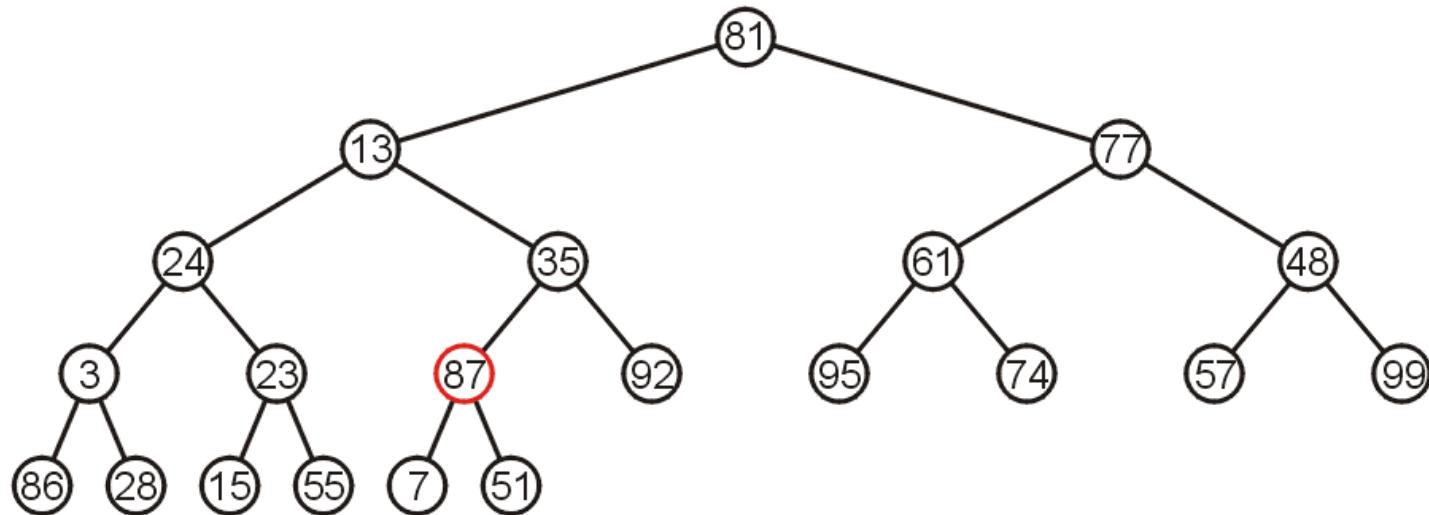




In-place Heapification

Starting at the back, we note that all leaf nodes are trivial heaps

Also, the subtree with 87 as the root is a max-heap

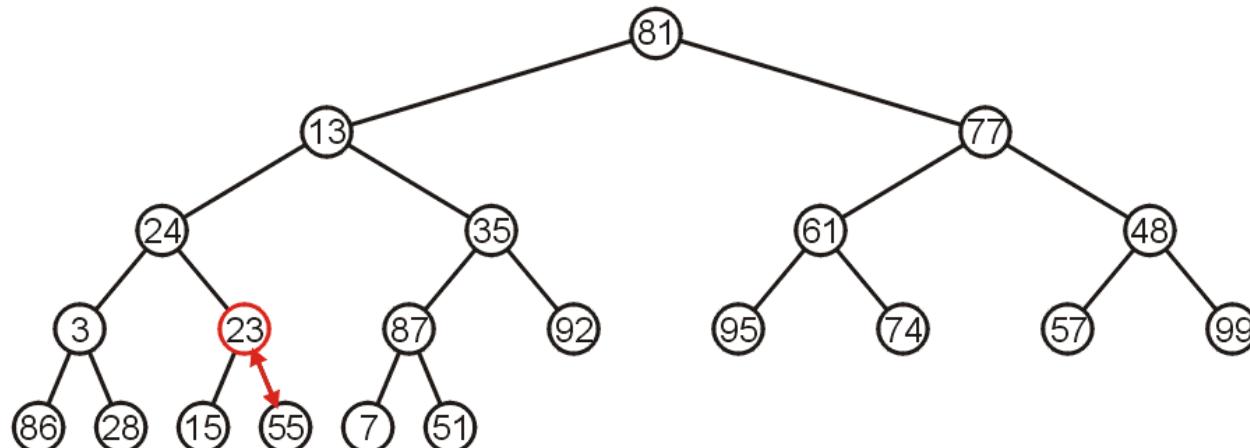




In-place Heapification

The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

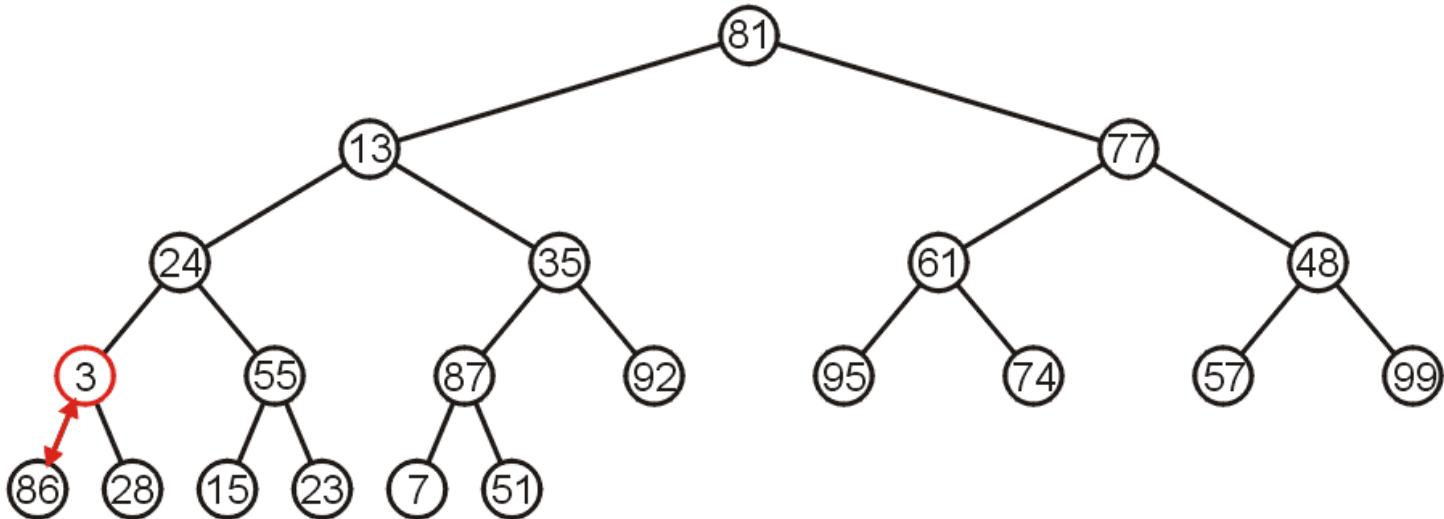
This process is termed *percolating down*





In-place Heapification

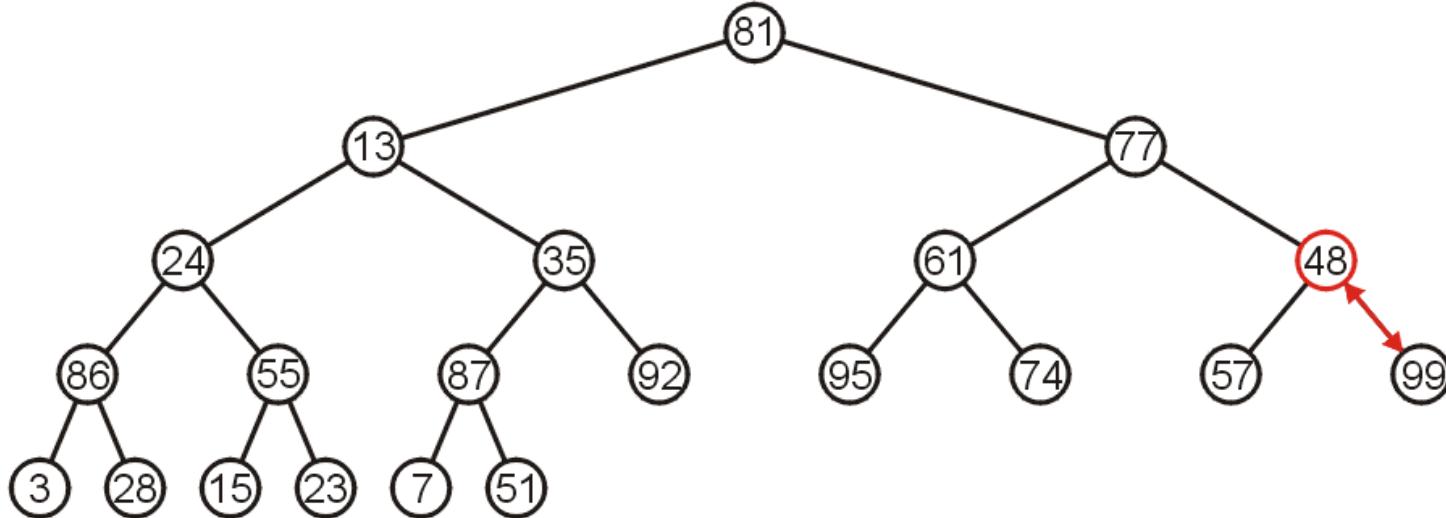
- The subtree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86





In-place Heapification

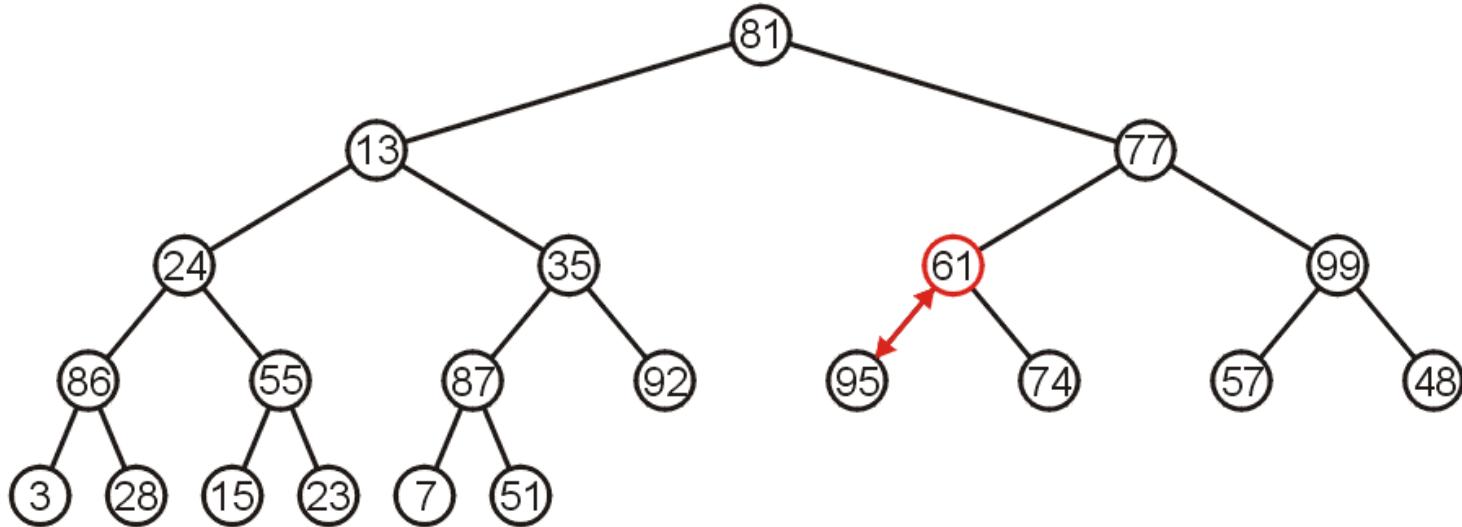
- Starting with the next higher level, the subtree with root 48 can be turned into a max-heap by swapping 48 and 99





In-place Heapification

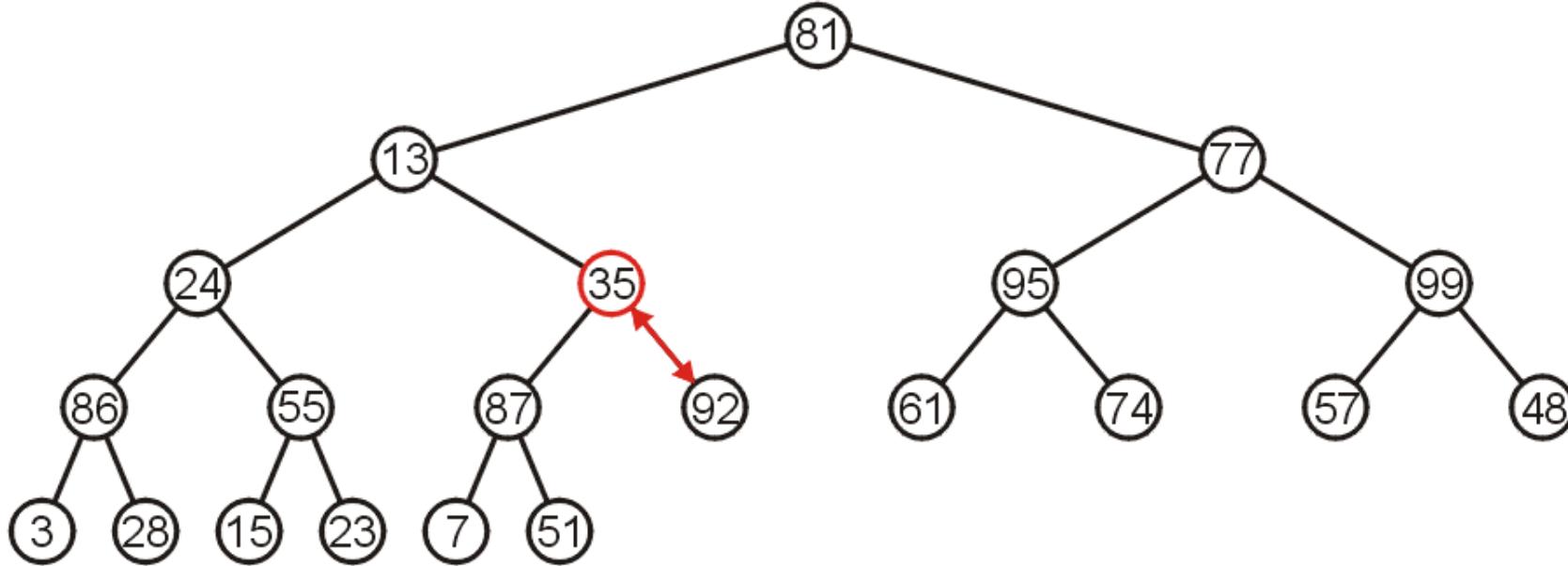
Similarly, swapping 61 and 95 creates a max-heap of the next subtree





In-place Heapification

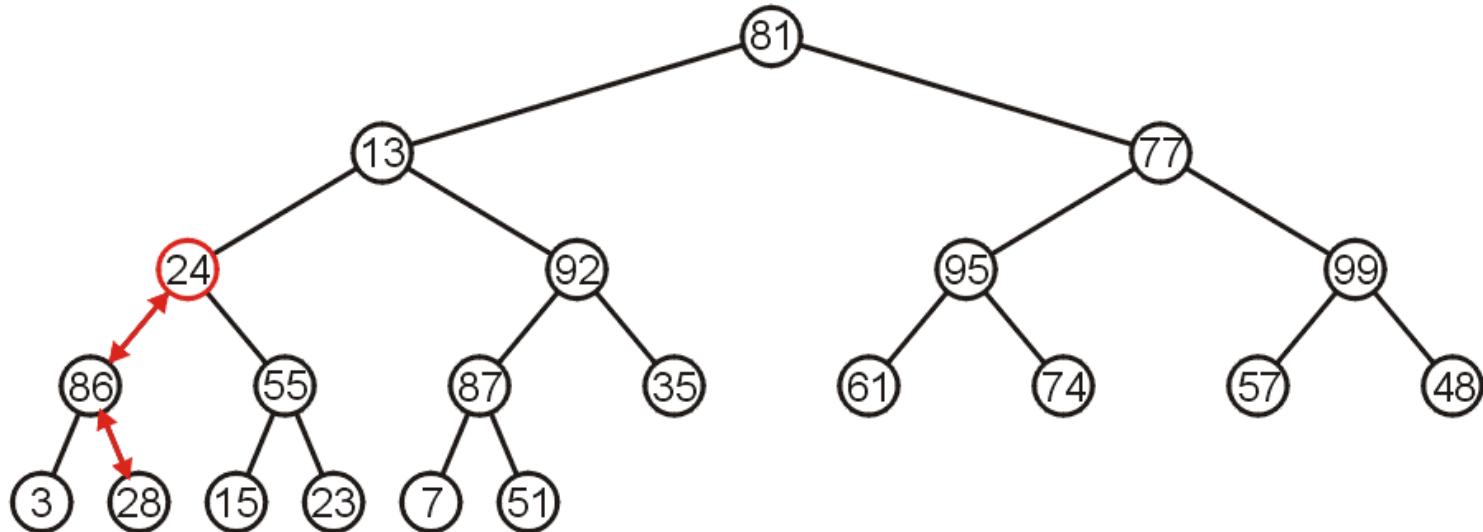
As does swapping 35 and 92





In-place Heapification

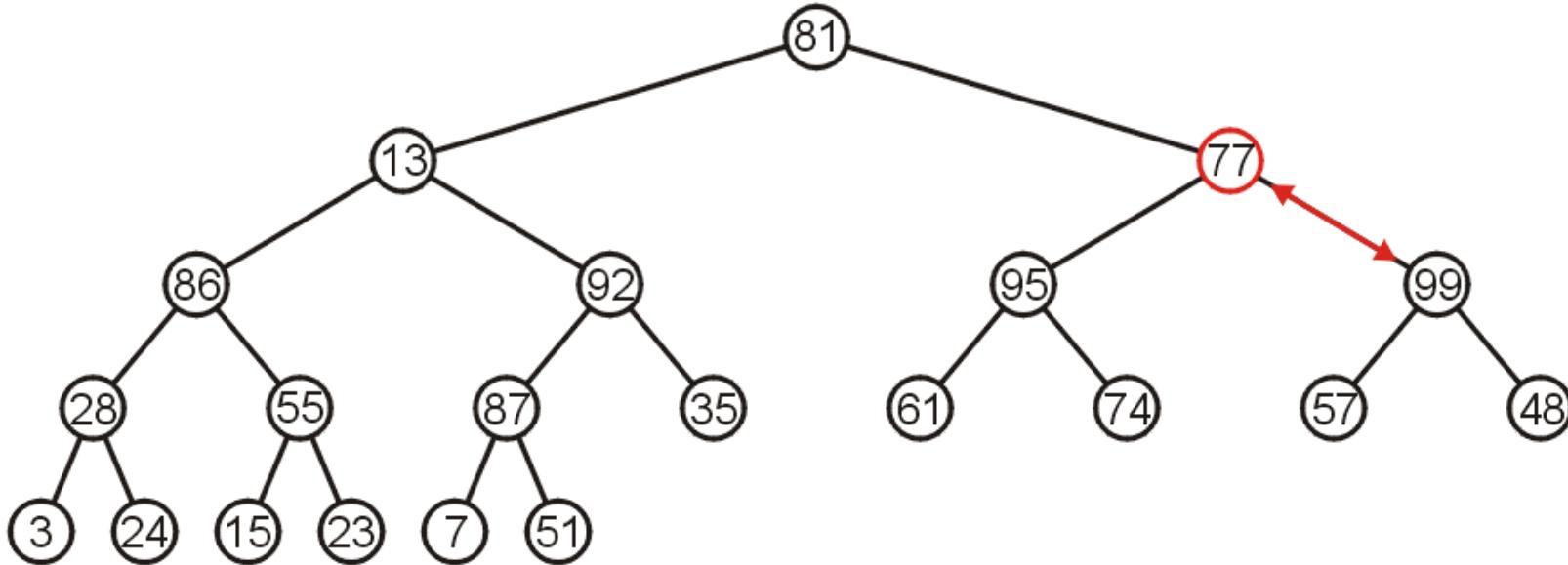
The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28





In-place Heapification

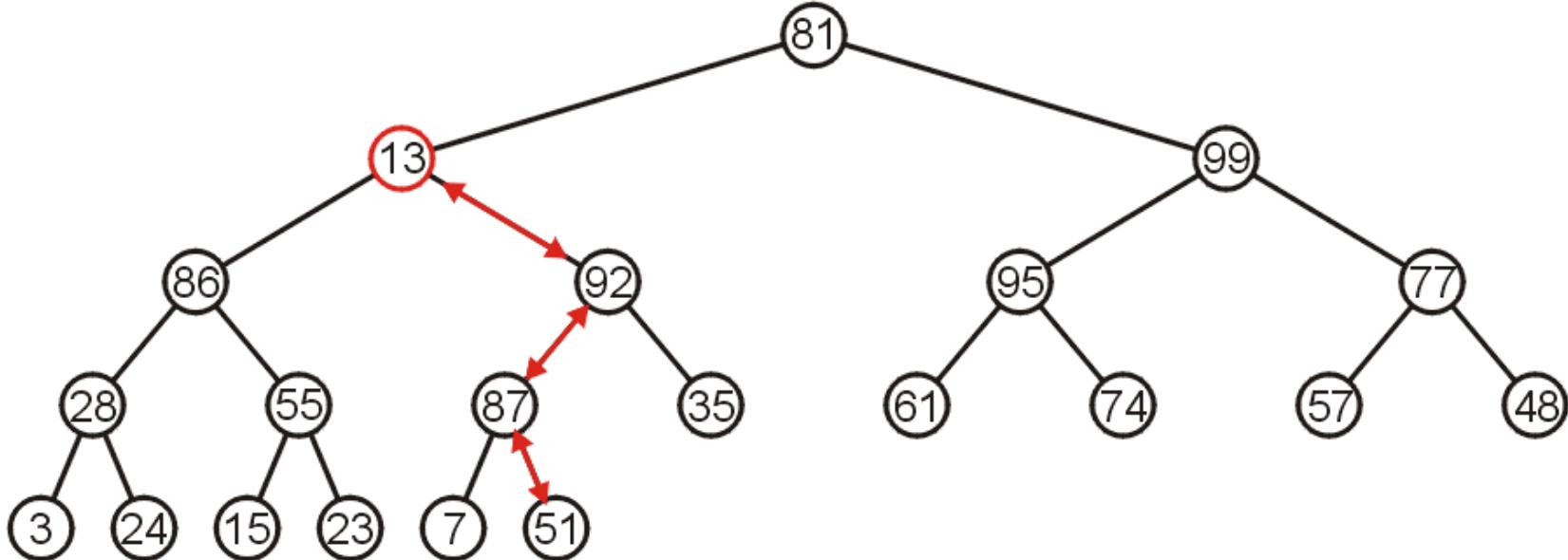
The right-most subtree of the next higher level may be turned into a max-heap by swapping 77 and 99





In-place Heapification

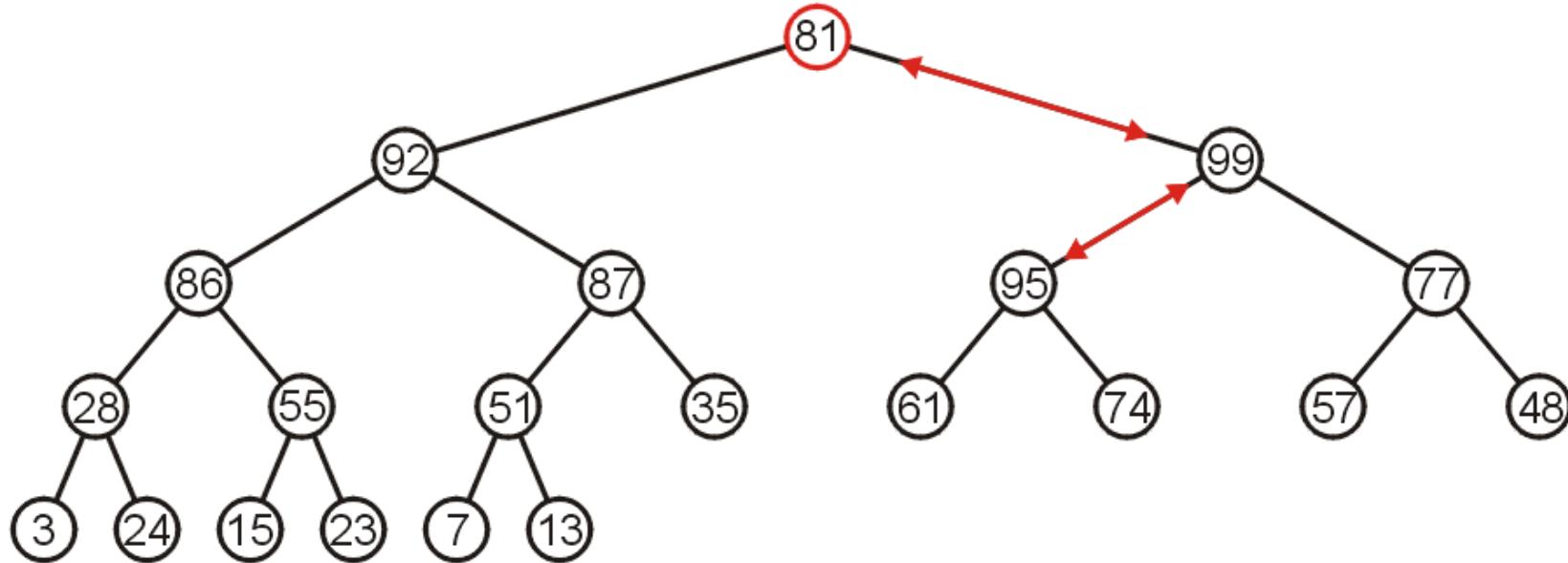
However, to turn the next subtree into a max-heap requires that 13 be percolated down to a leaf node





In-place Heapification

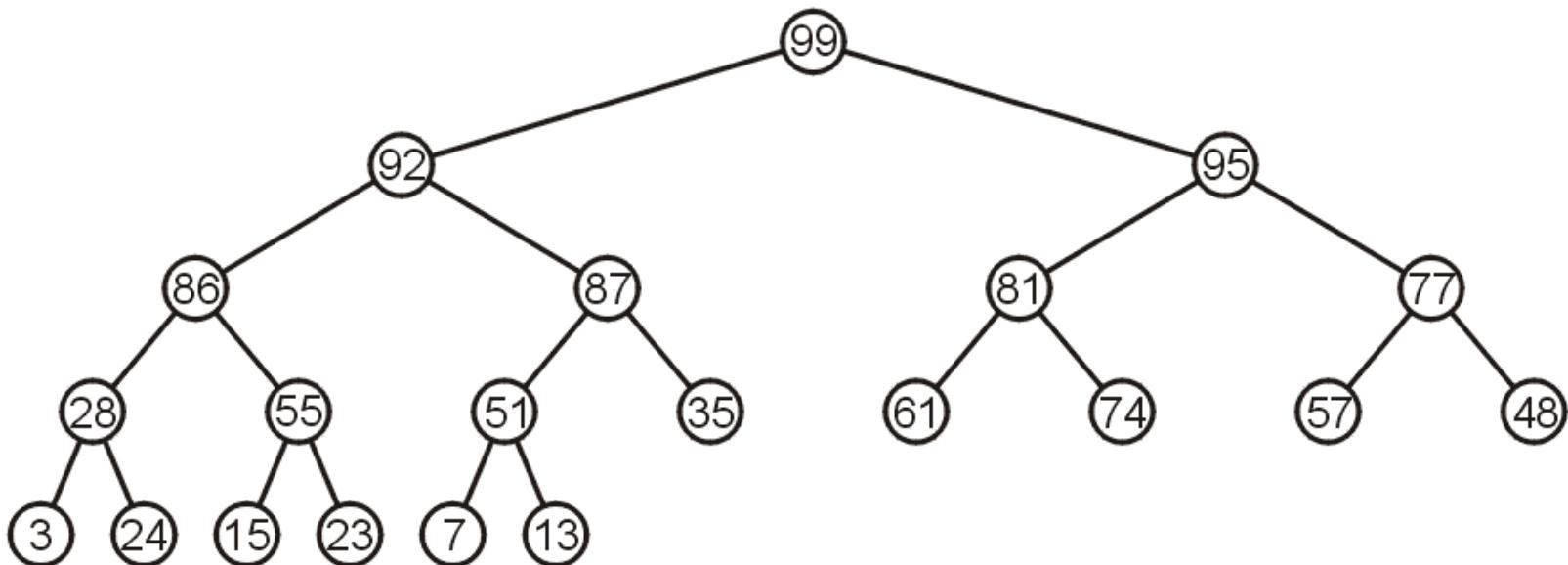
The root need only be percolated down by two levels





In-place Heapification

The final product is a max-heap

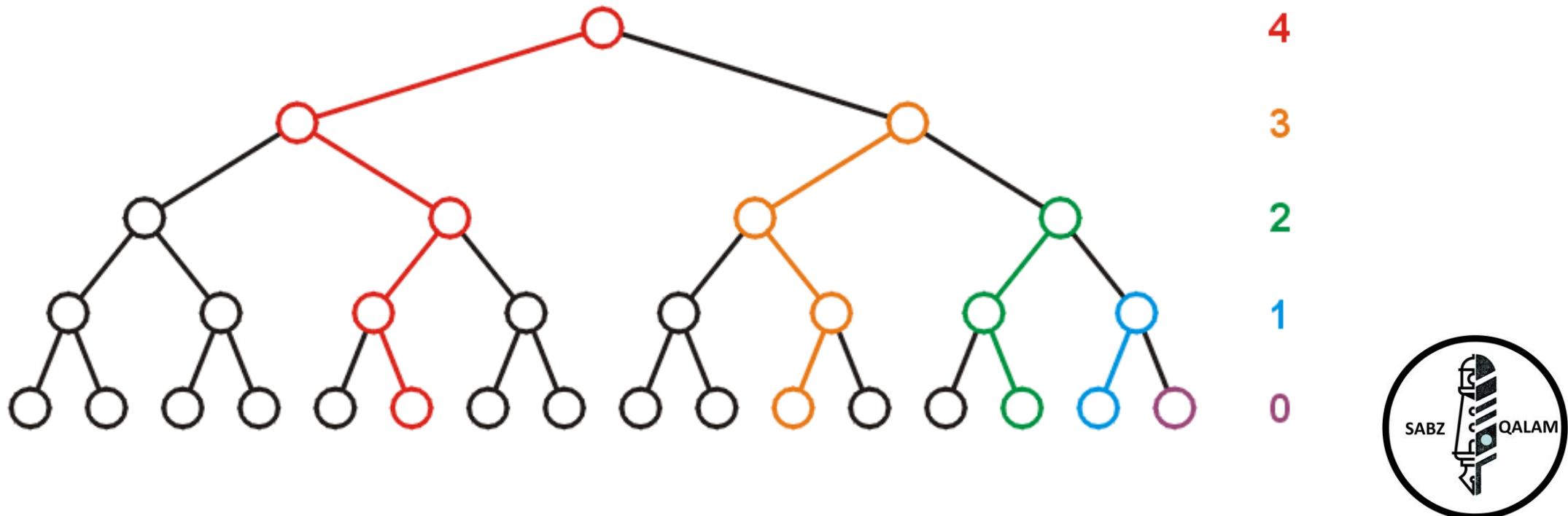




Run-time Analysis of Heapify

Considering a perfect tree of height h :

- The maximum number of swaps which a second-lowest level would experience is 1, the next higher level, 2, and so on





Run-time Analysis of Heapify

At depth k , there are 2^k nodes and in the worst case, all of these nodes would have to percolated down $h - k$ levels

- In the worst case, this would requiring a total of $2^k(h - k)$ swaps

Writing this sum mathematically, we get:

$$\sum_{k=0}^h 2^k (h - k) = (2^{h+1} - 1) - (h + 1)$$





Run-time Analysis of Heapify

Recall that for a perfect tree, $n = 2^{h+1} - 1$ and $h + 1 = \lg(n + 1)$, therefore

$$\sum_{k=0}^h 2^k (h-k) = n - \lg(n+1)$$

Each swap requires two comparisons (which child is greatest), so there is a maximum of $2n$ (or $\Theta(n)$) comparisons





Run-time Analysis of Heapify

Note that if we go the other way (treat the first entry as a max heap and then continually insert new elements into that heap, the run time is at worst

$$\begin{aligned}\sum_{k=0}^h 2^k k &= 2^{h+1}(h-1) + 2 \\ &= (2^{h+1} + 1)(h-1) - (h-1) + 2 \\ &= n(\lg(n+1) - 2) - \lg(n+1) + 4 = \Theta(n \ln(n))\end{aligned}$$

- It is significantly better to start at the back





Sorting

Heap Sort- III

Dr. Bilal Wajid





Example Heap Sort

Let us look at this example: we must convert the unordered array with $n = 10$ elements into a max-heap

46	52	28	17	3	63	34	81	70	95
----	----	----	----	---	----	----	----	----	----

None of the leaf nodes need to be percolated down, and the first non-leaf node is in position $n/2$

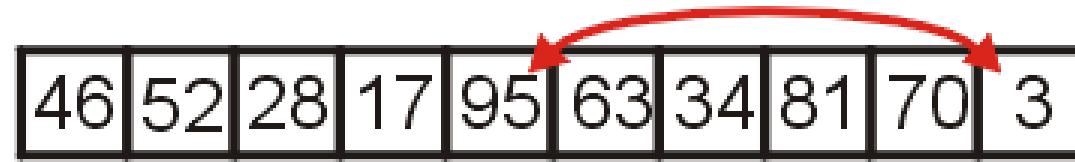
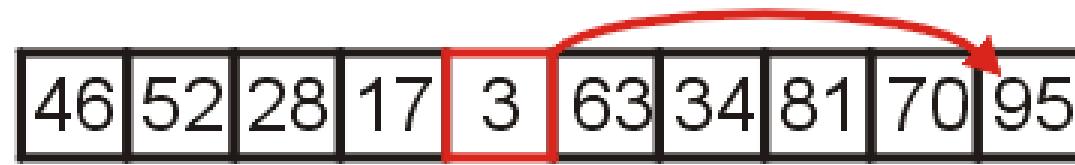
Thus we start with position $10/2 = 5$





Example Heap Sort

We compare 3 with its child and swap them





Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (70)





Example Heap Sort

We compare 28 with its two children, 63 and 34, and swap it with the largest child





Example Heap Sort

We compare 52 with its children, swap it with the largest

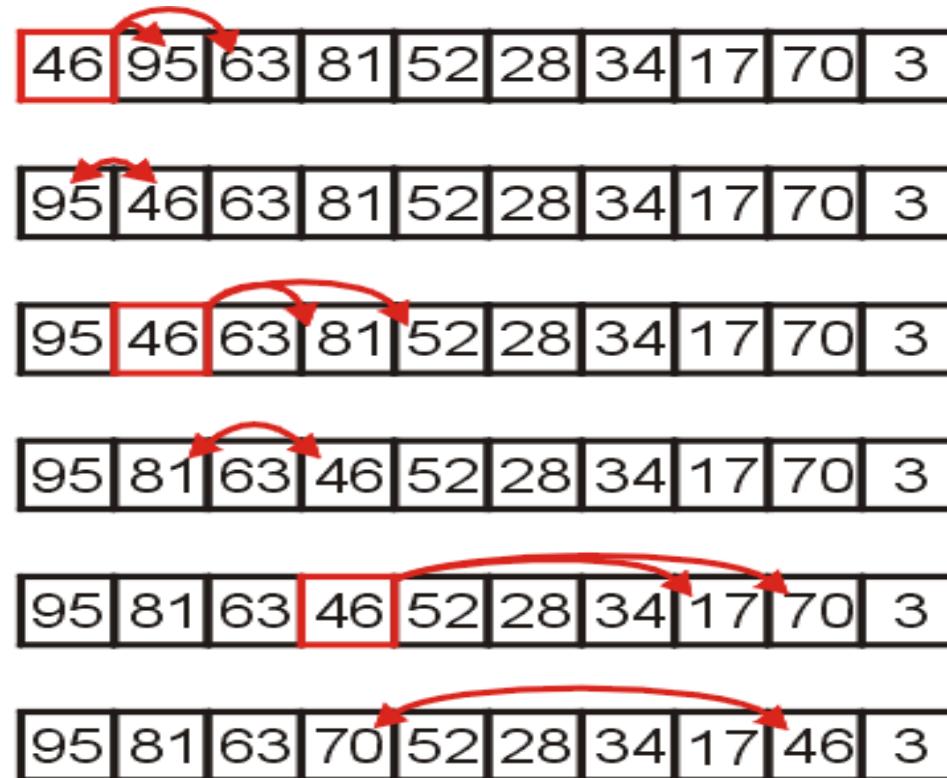
- Recursing, no further swaps are needed





Example Heap Sort

- Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70





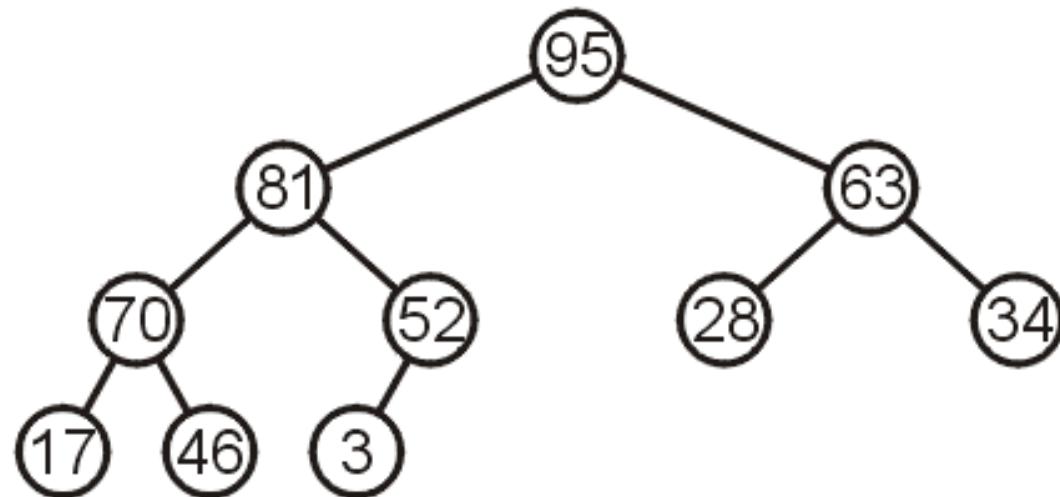
Heap Sort Example

We have now converted the unsorted array

46	52	28	17	3	63	34	81	70	95
----	----	----	----	---	----	----	----	----	----

into a max-heap:

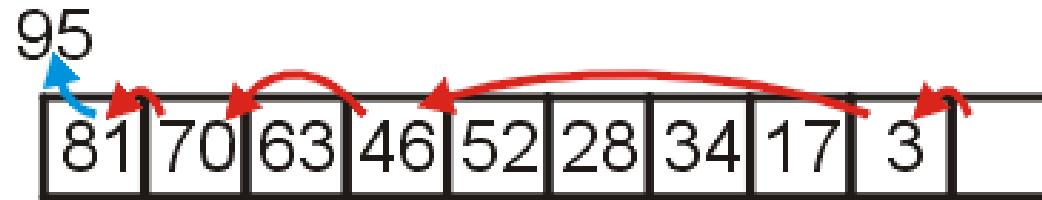
95	81	63	70	52	28	34	17	46	3
----	----	----	----	----	----	----	----	----	---



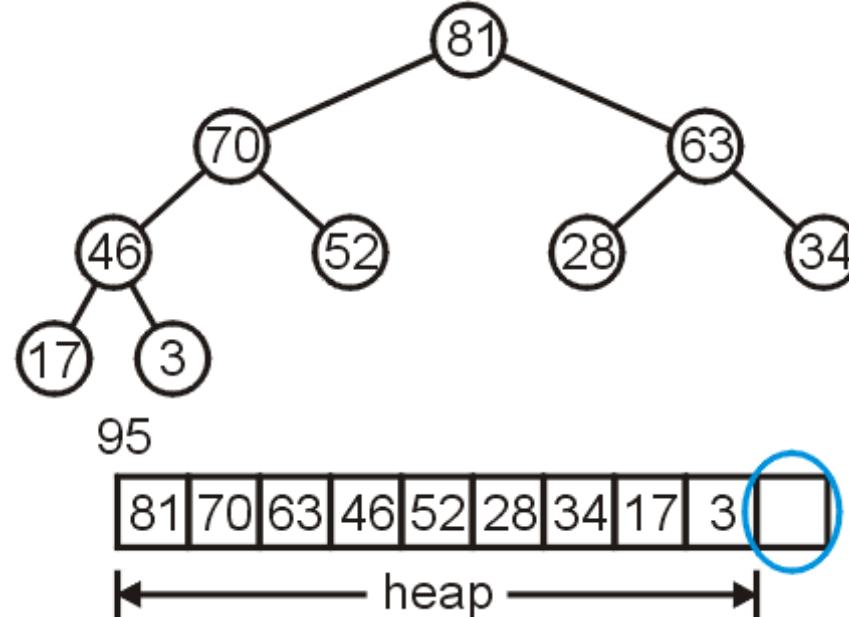


Heap Sort Example

Suppose we pop the maximum element of this heap



This leaves a gap at the back of the array:



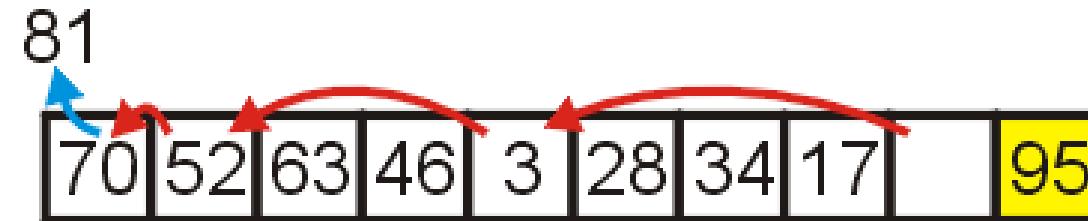


Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



Repeat this process: pop the maximum element, and then insert it at the end of the array:

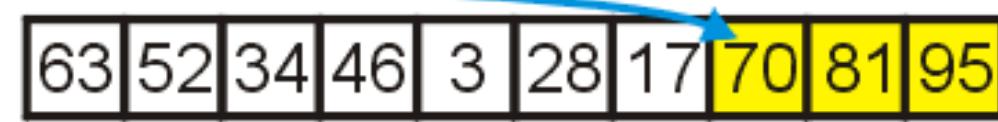
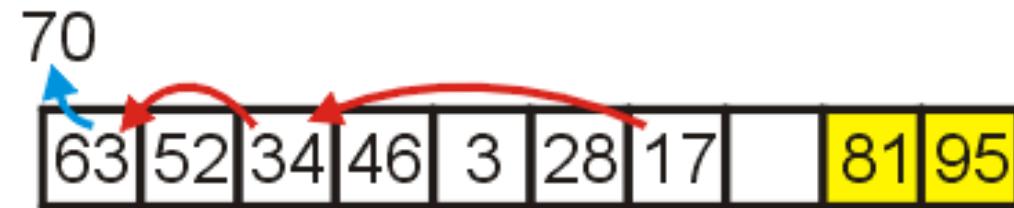




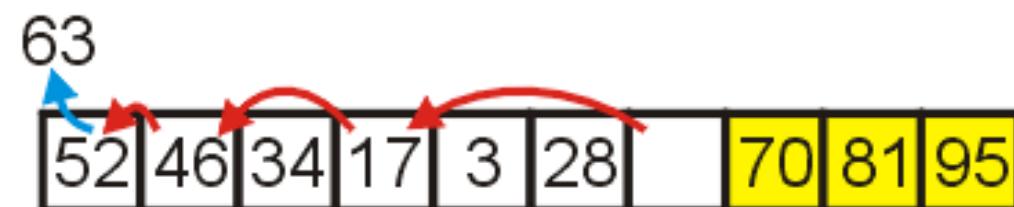
Heap Sort Example

Repeat this process

- Pop and append 70



- Pop and append 63

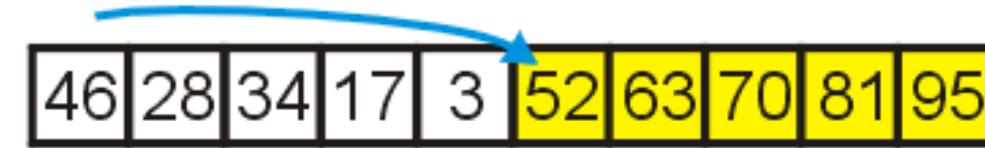
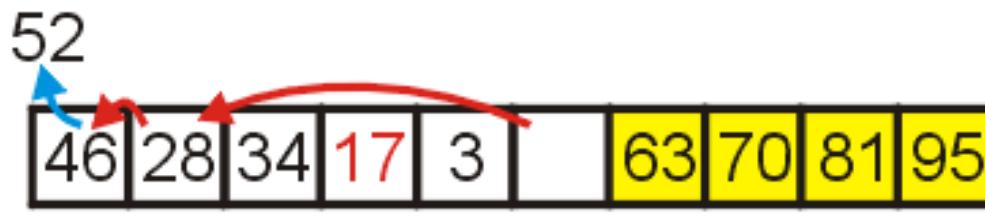




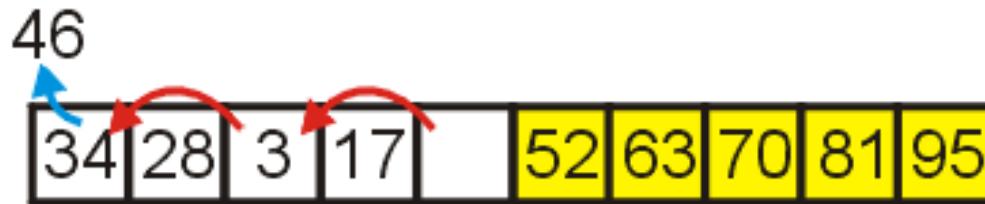
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



- Pop and append 46

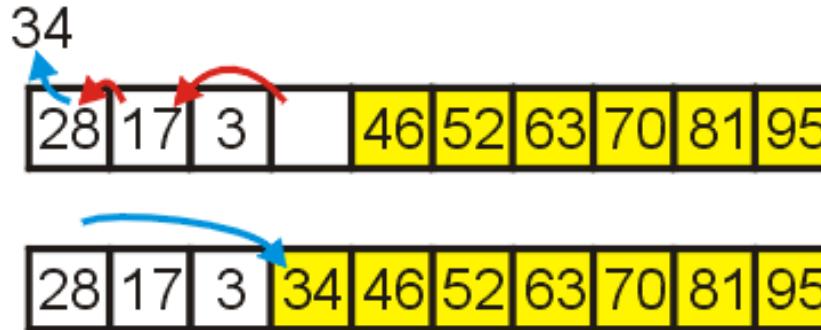




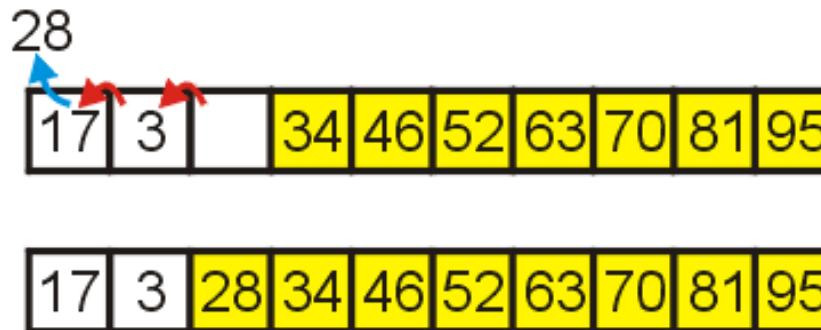
Heap Sort Example

Continuing...

- Pop and append 34



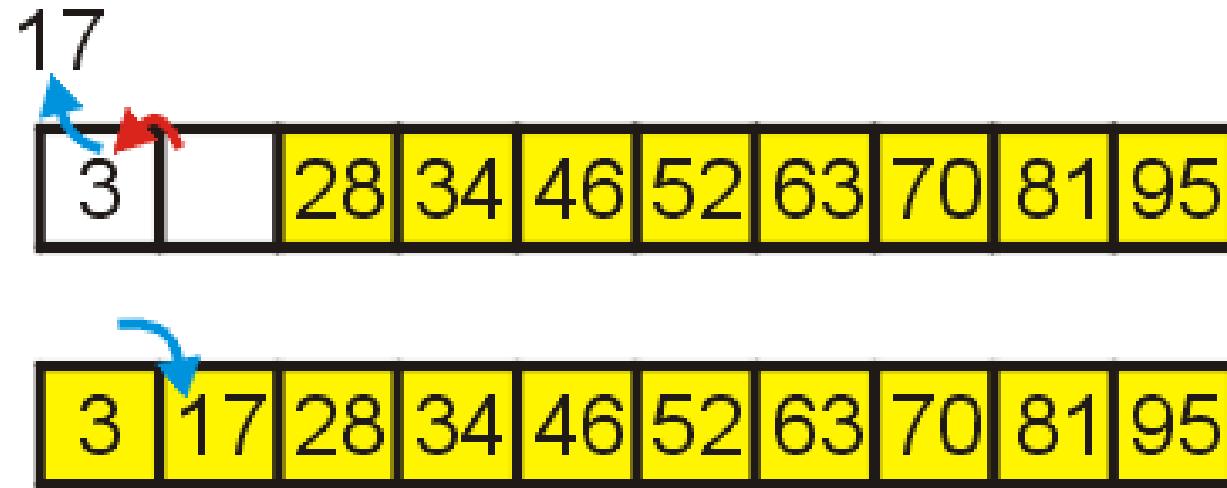
- Pop and append 28





Heap Sort Example

Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted





Heap Sort

Heapification runs in $\Theta(n)$

Popping n items from a heap of size n , as we saw, runs in $\Theta(n \ln(n))$ time

- We are only making one additional copy into the blank left at the end of the array

Therefore, the total algorithm will run in $\Theta(n \ln(n))$ time





Heap Sort

There are no worst-case scenarios for heap sort

- Dequeueing from the heap will always require the same number of operations regardless of the distribution of values in the heap

There is one best case: if all the entries are identical, then the run time is $\Theta(n)$

The original order may speed up the *heapification*, however, this would only speed up an $\Theta(n)$ portion of the algorithm





Run-time Summary

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n)$	All or most entries are the same





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Bucket Sort – I

Dr. Bilal Wajid





Types of Sorting Algorithms

- Non-recursive/Incremental comparison sorting
 - Selection sort
 - Bubble sort
 - Insertion sort
- Recursive comparison sorting
 - Merge sort
 - Quick sort
 - Heap sort
- Non-comparison linear sorting
 - Count sort
 - Radix sort
 - Bucket sort





Bucket Sort – Basic Idea – I

- **Idea I:**

- Input numbers are uniformly distributed in $[0,1)$.
- Suppose input size is n .





Bucket Sort – Basic Idea – II

- **Idea I:**

- Input numbers are uniformly distributed in $[0,1)$.
- Suppose input size is n .

- **Ideal II:**

- Divide $[0,1)$ into n equal-sized buckets ($k = \Theta(n)$)
- Distribute the n input values into the buckets
- Sort each bucket (insertion sort as default).
- Go through the buckets in order, listing elements in each one

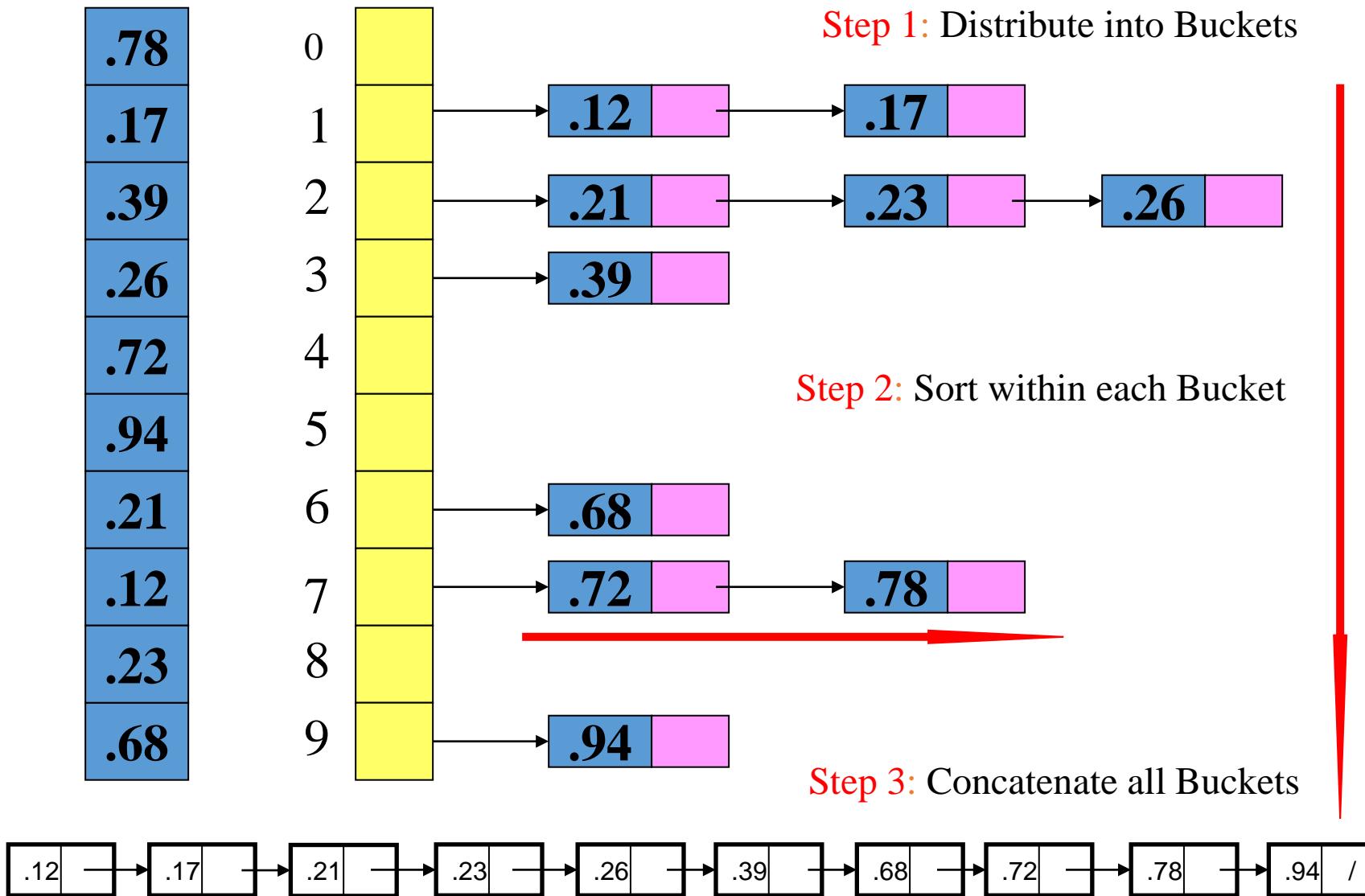


Bucket Sort – Pseudocode – I

Input: $A[1 \dots n]$, where $0 \leq A[i] < 1$ for all i

Output: elements $A[i]$ sorted

Bucket Sort – Pseudocode II(Example)





Bucket Sort – Pseudocode – III

BUCKET-SORT(A, n)

for $i \leftarrow 0$ **to** $n-1$

 insert $A[i]$ into list $B[\lfloor n A[i] \rfloor]$

for $i \leftarrow 0$ **to** $n-1$

 sort list $B[i]$ with insertion sort

 concatenate the lists $B[0], B[1], \dots, B[n - 1]$
 in order

}

$O(n)$

$O(n_i^2)$

$O(n)$

Where n_i is the size of bucket $B[i]$.

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





Bucket Sort Analysis – I

- Best Case: $O(n + k)$
- Average Case: $O(n)$
- Worst Case: $O(n^2)$





Summary: Sorting Algorithms

Insertion Sort: Suitable only for small $n \leq 50$ or nearly sorted inputs

Merge Sort: Guaranteed to be fast even in its worst case; stable

Quick Sort: Most useful general-purpose sorting for very little memory requirement and fastest average time. (Choose the median of three elements as pivot in practice)

Heap Sort: Requiring minimum memory and guaranteed to run fast; average and maximum time both roughly *twice* the average time of quick sort. Useful for time-critical applications

Counting Sort: Very useful when the keys have small range; stable; memory space for counters and for $2n$ records.

Radix Sort: Appropriate for keys either rather short or with an lexicographic collating sequence.

Bucket Sort: Assuming keys to have uniform distribution.





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Radix Sort- I

Dr. Bilal Wajid





Types of Sorting Algorithms

- Non-recursive/Incremental comparison sorting
 - Selection sort
 - Bubble sort
 - Insertion sort
- Recursive comparison sorting
 - Merge sort
 - Quick sort
 - Heap sort
- Non-comparison linear sorting
 - Count sort
 - Radix sort
 - Bucket sort





Radix Sort – Basic Idea – I

- **Idea I:**

- Represents keys as d -digit numbers in some base- k
- $\text{key} = x_1x_2\dots x_d$ where $0 \leq x_i \leq k-1$





Radix Sort – Basic Idea – II (Example)

- **Idea I:**
 - Represents keys as d-digit numbers in some base-k
 - $\text{key} = x_1x_2\dots x_d$ where $0 \leq x_i \leq k-1$
- Example: $\text{key}=15$
 - $\text{key}_{10} = 15$, $d=2$, $k=10$ where $0 \leq x_i \leq 9$
 - $\text{key}_2 = 1111$, $d=4$, $k=2$ where $0 \leq x_i \leq 1$





Radix Sort – Basic Idea – III

- Idea: $d=O(1)$ and $k =O(n)$
- Sorting looks at one column at a time
 - For a d digit number, sort the least significant digit first
 - Continue sorting on the next least significant digit, until all digits have been sorted
 - Requires only d passes through the list



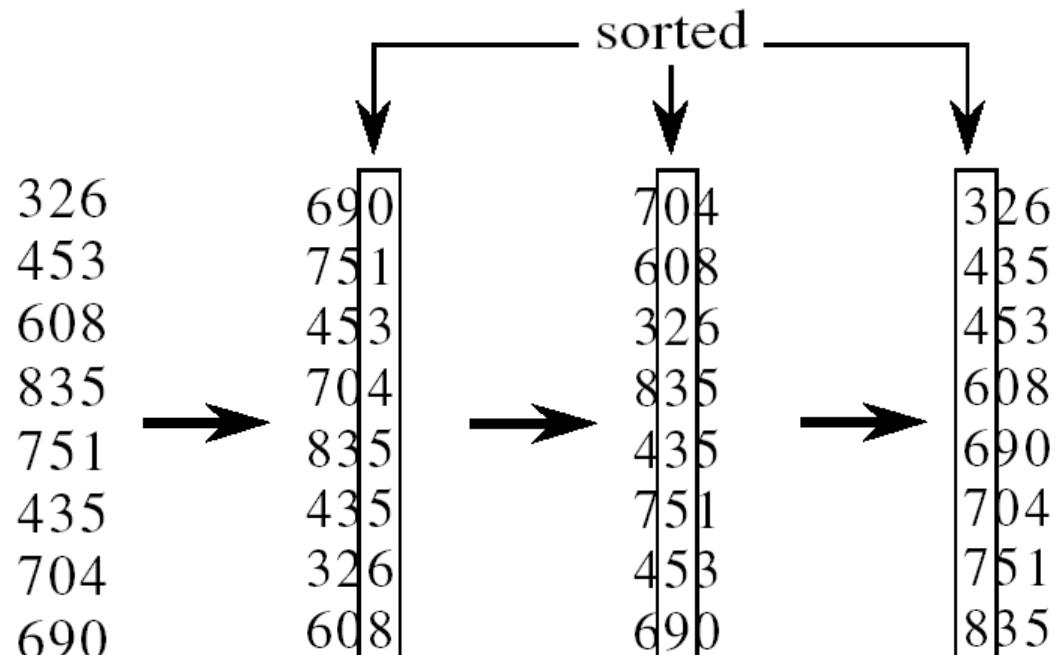
Radix Sort – Pseudocode – I

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i     $\Theta(n+k)$  } d times
```

Running time is $\Theta(dn+dk)$ (or $\Theta(n)$ if $k = O(n)$ and $d = O(1)$)



Radix Sort – Pseudocode – II



(stable sort: preserves order of identical elements)





Radix Sort – Problems

Problem: sort 1 million 64-bit numbers

Treat as four-digit radix 2^{16} numbers

Can sort in just four passes with radix sort!

So why would we ever use anything but radix sort?





Radix Sort Analysis – I

- Best Case: $O(kn)$
- Average Case: $O(kn)$
- Worst Case: $O(kn)$

Radix sort complexity is $O(kn)$ for n keys which are integers of word size k .





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Quick Sort- I

Dr. Bilal Wajid





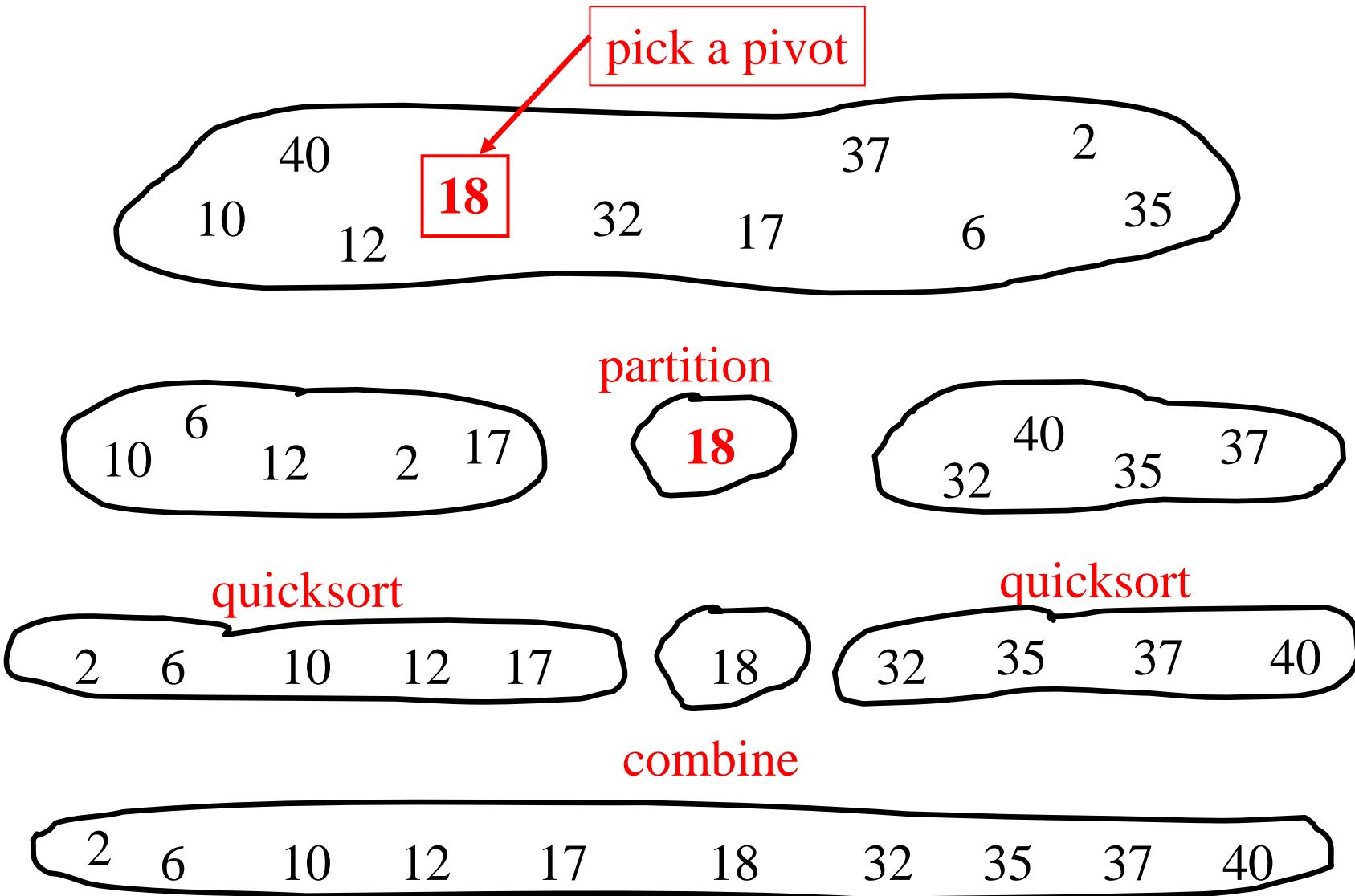
Quick Sort

- **Quick Sort:** orders a list of values by partitioning the list around one element called a ***pivot***, then sorting each partition. It is based on divide and conquer approach.
- **Key Points:** Given an array **S** to be sorted
 - Pick any element **v** in array **S** as the **pivot** (i.e. **partition element**)
 - Partition the remaining elements in **S** into two groups
 - **S1** = {all elements in **S** - {**v**} that are smaller than **v**}
 - **S2** = {all elements in **S** - {**v**} that are larger than **v**}
 - apply the quick sort algorithm (**recursively**) to **both partitions**
- Trick lies in handling the partitioning
 - i.e. picking a good pivot is essential to performance





Quick Sort Illustrated





Quick Sort Pseudocode

QUICKSORT(A, p, r)

- 1 **if** $p < r$
- 2 $q = \text{PARTITION}(A, p, r)$
- 3 QUICKSORT($A, p, q - 1$)
- 4 QUICKSORT($A, q + 1, r$)





Partitioning Algorithm

Original input : $S = \{6, 1, 4, 9, 0, 3, 5, 2, 7, 8\}$

Pick the first element as pivot



Have two 'iterators' – i and j

- i starts at first element and moves forward
- j starts at last element and moves backwards



While ($i < j$)

- Move i to the right till we find a number greater than **pivot**
- Move j to the left till we find a number smaller than **pivot**

If ($i < j$) **swap**($S[i]$, $S[j]$)

- The effect is to push larger elements to the right and smaller elements to the left

Swap the **pivot** with $S[i]$





Partitioning Algorithm Illustrated

	i	8	1	4	9	0	3	5	2	j	7	6	pivot
Move	i	8	1	4	9	0	3	5	j	2	7	6	pivot
swap	i	2	1	4	9	0	3	5	j	8	7	6	pivot
move		2	1	4	j	9	0	3	j	5	8	7	pivot
swap		2	1	4	j	5	0	3	j	9	8	7	pivot
move		2	1	4	5	0	j	j	3	9	8	7	pivot
Swap S[i] with pivot		2	1	4	5	0	j	j	3	6	8	7	9
							j	i					pivot





Partitioning Pseudocode

PARTITION(A, p, r)

1. pivot = A[p];
2. leftPointer = p + 1
3. rightPointer = r
4. while (True)
 5. while (A[leftPointer] < pivot)
 6. leftPointer++;
 7. while (A[rightPointer] >= pivot)
 8. rightPointer--;
 9. if leftPointer >= rightPointer
 10. break;
 11. else
 12. A[leftPointer] ↔ A[rightPointer]
 13. A[leftPointer] ↔ pivot

*What is the running time of
partition()?*

***partition()** runs in $O(n)$ time*





Partitioning Algorithm Illustrated

- choose pivot: $\underline{4} \ 3 \ 6 \ 9 \ 2 \ 4 \ 3 \ 1 \ 2 \ 1 \ 8 \ 9 \ 3 \ 5 \ 6$
- search: $\underline{4} \ \mathbf{3} \ \mathbf{6} \ 9 \ 2 \ 4 \ 3 \ 1 \ 2 \ 1 \ 8 \ 9 \ \mathbf{3} \ \mathbf{5} \ 6$
- swap: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ 9 \ 2 \ 4 \ 3 \ 1 \ 2 \ 1 \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$
- search: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ \mathbf{9} \ 2 \ 4 \ 3 \ 1 \ 2 \ \mathbf{1} \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$
- swap: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ \mathbf{1} \ 2 \ 4 \ 3 \ 1 \ 2 \ \mathbf{9} \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$
- search: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ \mathbf{1} \ 2 \ \mathbf{4} \ 3 \ 1 \ \mathbf{2} \ 9 \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$
- swap: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ \mathbf{1} \ 2 \ \mathbf{2} \ 3 \ 1 \ \mathbf{4} \ 9 \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$
- search: $\underline{4} \ \mathbf{3} \ \mathbf{3} \ \mathbf{1} \ 2 \ \mathbf{2} \ \mathbf{3} \ 1 \ \mathbf{4} \ 9 \ 8 \ 9 \ \mathbf{6} \ 5 \ 6$ (left
> right)
- swap with pivot: $\mathbf{1} \ 3 \ 3 \ 1 \ 2 \ \mathbf{2} \ \mathbf{3} \ \underline{4} \ 4 \ 9 \ 8 \ 9 \ 6 \ 5 \ 6$



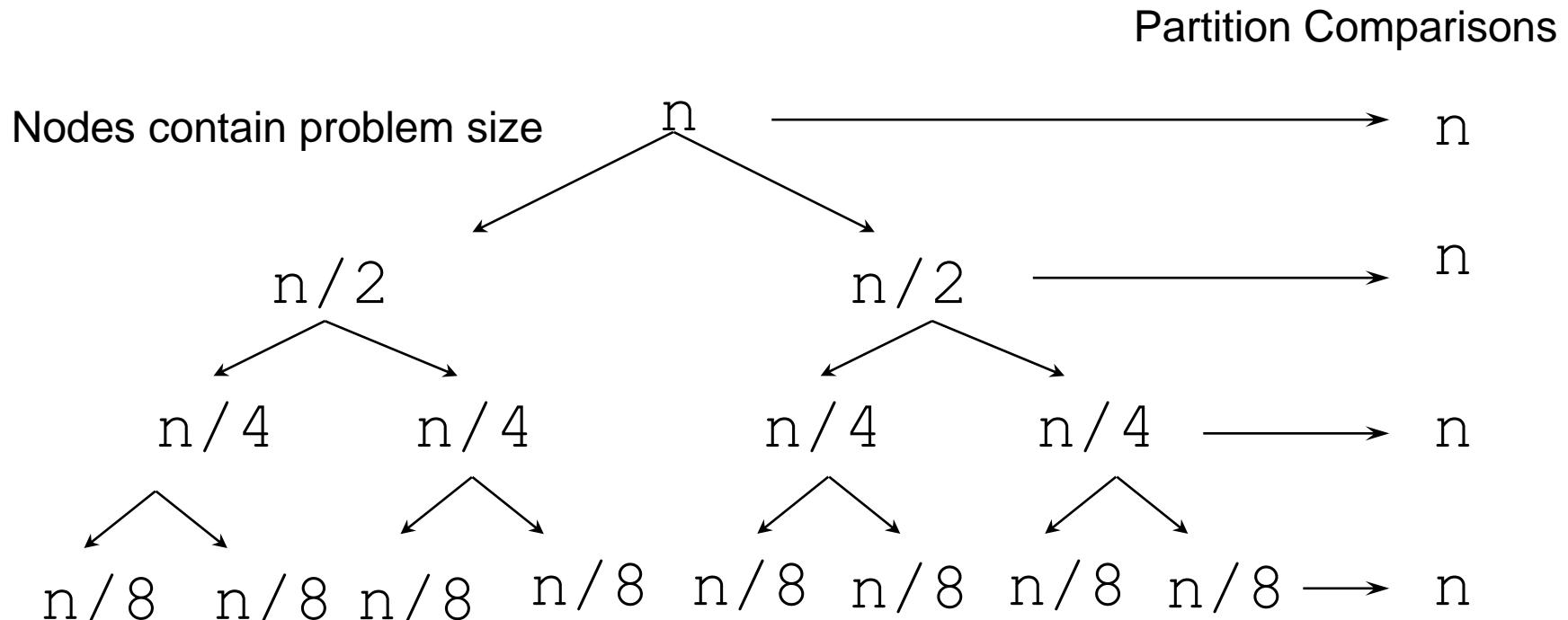


Quick Sort: Best Case Analysis

Assuming that keys are random, uniformly distributed.

The best case running time occurs when pivot is the median

- Partition splits array in two sub-arrays of size $n/2$ at each step



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

$$T(n) = cn \log n + n = O(n \log n)$$





Quick Sort: Worst Case Analysis

Assuming that keys are random, uniformly distributed.

The worst case running time occurs when pivot is the smallest (or largest) element all the time

- One of the two sub-arrays will have zero elements at each step

Height of Tree = n

$$\text{Total Cost} = \sum_{i=1}^{n-1} t_i = \frac{n(n-1)}{2}$$

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

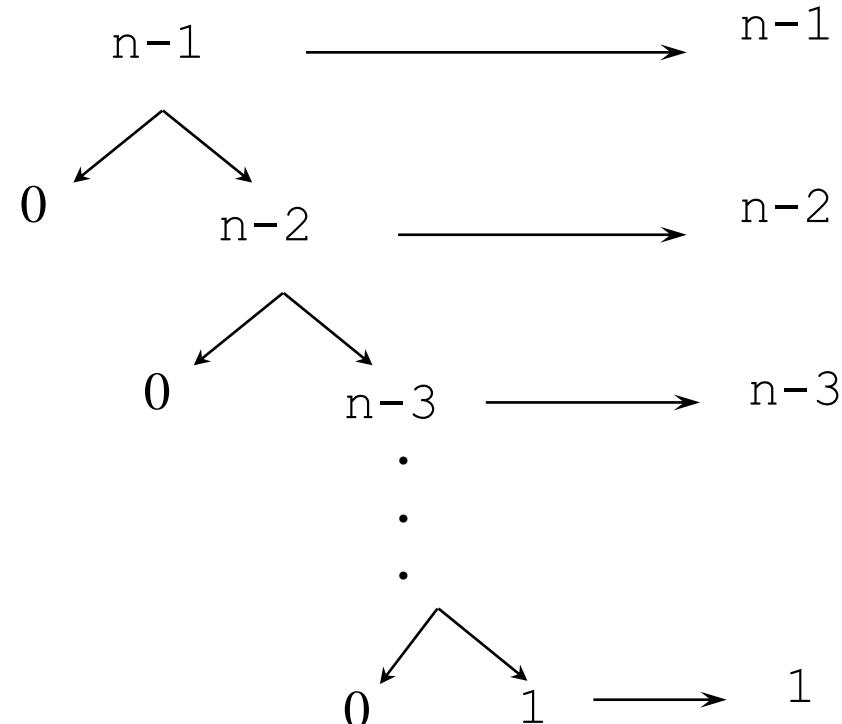
$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

...

$$T(2) = T(1) + 2c$$

$$T(N) = T(1) + c \sum_{i=2}^N i = O(N^2)$$





Quick Sort: Average Case Analysis

Assuming that keys are random, uniformly distributed.

The average case running time occurs when pivot can take any position at each step

- The size of two sub-arrays will vary at each step

Per level Cost = n

Height of Tree = $\log_x n$ (x depends on how the partition is split)

$O(n \lg n)$





Picking the Pivot

- How would you pick one?
- Strategy 1: Pick the **first element** in **S**
 - Works only if input is random
 - What if input **S** is sorted, or even mostly sorted?
 - All the remaining elements would go into either **S1** or **S2**!
 - Terrible performance!
 - Why worry about sorted input?
 - Remember → Quicksort is recursive, so sub-problems could be sorted
 - Plus mostly sorted input is quite frequent





Picking the Pivot (contd.)

- Strategy 2: Pick the pivot **randomly**
 - Would usually work well, even for mostly sorted input
 - Unless the random number generator is not quite random!
 - Plus random number generation is an expensive operation
- Strategy 3: **Median-of-three Partitioning**
 - *Ideally*, the pivot should be the **median** of input array **S**
 - Median = element in the middle of the sorted sequence
 - Would divide the input into two almost equal partitions
 - Unfortunately, harder to calculate median quickly, without sorting first!
 - So find the approximate median
 - Pivot = median of the **left-most**, **right-most** and **center** elements of array **S**
 - Solves the problem of sorted input





Picking the Pivot (contd.)

- Example: **Median-of-three Partitioning**
 - Let input $S = \{6, 1, 4, 9, 0, 3, 5, 2, 7, 8\}$
 - **Left** = 0 and $S[\text{left}] = 6$
 - **Right** = 9 and $S[\text{right}] = 8$
 - **center** = $(\text{left} + \text{right})/2 = 4$ and $S[\text{center}] = 0$
 - Pivot
 - = Median of $S[\text{left}]$, $S[\text{right}]$, and $S[\text{center}]$
 - = median of 6, 8, and 0
 - = $S[\text{left}] = 6$





Quick Sort: Final Comments

- What happens when the array contains many duplicate elements?
 - Not stable
- What happens when the size of the array is small?
 - For small arrays ($N \leq 50$), Insertion sort is faster than quick sort due to recursive function call overheads of quick sort. Use hybrid algorithm; quick sort followed by insertion sort when $N \leq 50$
- However, Quicksort is *usually* $O(n \log_2 n)$
 - Constants are so good that it is generally the fastest algorithm known
 - Most real-world sorting is done by Quicksort
 - For optimum efficiency, the pivot must be chosen carefully
 - “Median of three” is a good technique for choosing the pivot
- However, no matter what you do, some bad cases can be constructed where Quick Sort runs in $O(n^2)$ time





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Quick Sort- II

Dr. Bilal Wajid





Outline

In this topic we will look at quicksort:

- The idea behind the algorithm
- The run time and worst-case scenario
- Strategy for avoiding the worst-case: median-of-three
- Implementing quicksort in place
- Examples





Strategy

We have seen two $\Theta(n \ln(n))$ sorting algorithms:

- Heap sort which allows in-place sorting, and
- Merge sort which is faster but requires more memory

We will now look at a recursive algorithm which may be done *almost* in place but which is faster than heap sort

- Use an object in the array (a pivot) to divide the two
- Average case: $\Theta(n \ln(n))$ time and $\Theta(\ln(n))$ memory
- Worst case: $\Theta(n^2)$ time and $\Theta(n)$ memory

We will look at strategies for avoiding the worst case





Quicksort

Merge sort splits the array sub-lists and sorts them

The larger problem is split into two sub-problems based on *location* in the array

Consider the following alternative:

- Choose an object in the array and partition the remaining objects into two groups relative to the chosen entry





Quicksort

For example, given

80	38	95	84	66	10	79	44	26	87	96	12	43	81	3
----	----	----	----	----	----	----	----	----	----	----	----	----	----	---

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	44	80	95	84	66	79	87	96	81
----	----	----	----	----	---	----	----	----	----	----	----	----	----	----

Notice that 44 is now in the correct location if the list was sorted

- Proceed by applying the algorithm to the first six and last eight entries





Run-time analysis

Like merge sort, we can either:

- Apply insertion sort if the size of the sub-list is sufficiently small, or
- Sort the sub-lists using quicksort

In the best case, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

What happens if we don't get that lucky?





Worst-case scenario

Suppose we choose the first element as our pivot and we try ordering a sorted list:

80	38	95	84	66	10	79	2	26	87	96	12	43	81	3
----	----	----	----	----	----	----	---	----	----	----	----	----	----	---

Using **2**, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3
---	----	----	----	----	----	----	----	----	----	----	----	----	----	---

We still have to sort a list of size $n - 1$

The run time is $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$

- Thus, the run time drops from $n \ln(n)$ to n^2





Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

80	38	95	84	66	10	79	2	26	87	96	12	43	81	3
----	----	----	----	----	----	----	---	----	----	----	----	----	----	---

Unfortunately, it's difficult to find

Alternate strategy: take the median of a subset of entries

- For example, take the median of the first, middle, and last entries





Median-of-three

It is difficult to find the median so consider another strategy:

- Choose the median of the first, middle, and last entries in the list

This will usually give the actual median

80	38	95	84	99	10	79	44	26	87	96	12	43	81	3
----	----	----	----	----	----	----	----	----	----	----	----	----	----	---





Median-of-three

Sorting the elements based on 44 results in two sub-lists, each of which must be sorted (again, using quicksort)

Select the 26



Select 81 to partition the second sub-list:





Median-of-three

If we choose a random pivot, this will, on average, divide a set of n items into two sets of size $1/4 n$ and $3/4 n$

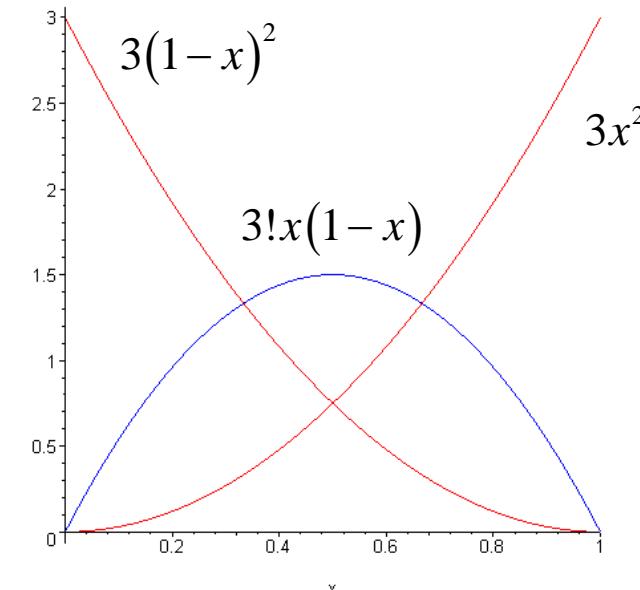
- 90 % of the time the width will have a ratio 1:19 or better

Choosing the median-of-three, this will, on average, divide the n items into two sets of size $5/16 n$ and $11/16 n$

- Median-of-three helps speed the algorithm
- This requires order statistics:

$$2 \int_0^{\frac{1}{2}} x \cdot (6x(1-x)) dx = \frac{5}{16} = 0.3125$$

- Ratio 1:2.2 on average
- 90 % of the time, the width will have a ratio of 1:6.388 or better





Median-of-three

Recall that merge sort always divides a list into two equal halves:

- The median-of-three will require steps
- A single random pivot will require steps

$$\frac{\ln\left(\frac{1}{2}\right)}{\ln\left(\frac{11}{16}\right)} \approx 1.8499 \text{ or } 85\% \text{ more recursive}$$

$$\frac{\ln\left(\frac{1}{2}\right)}{\ln\left(\frac{3}{4}\right)} \approx 2.4094 \text{ or } 141\% \text{ more recursive}$$

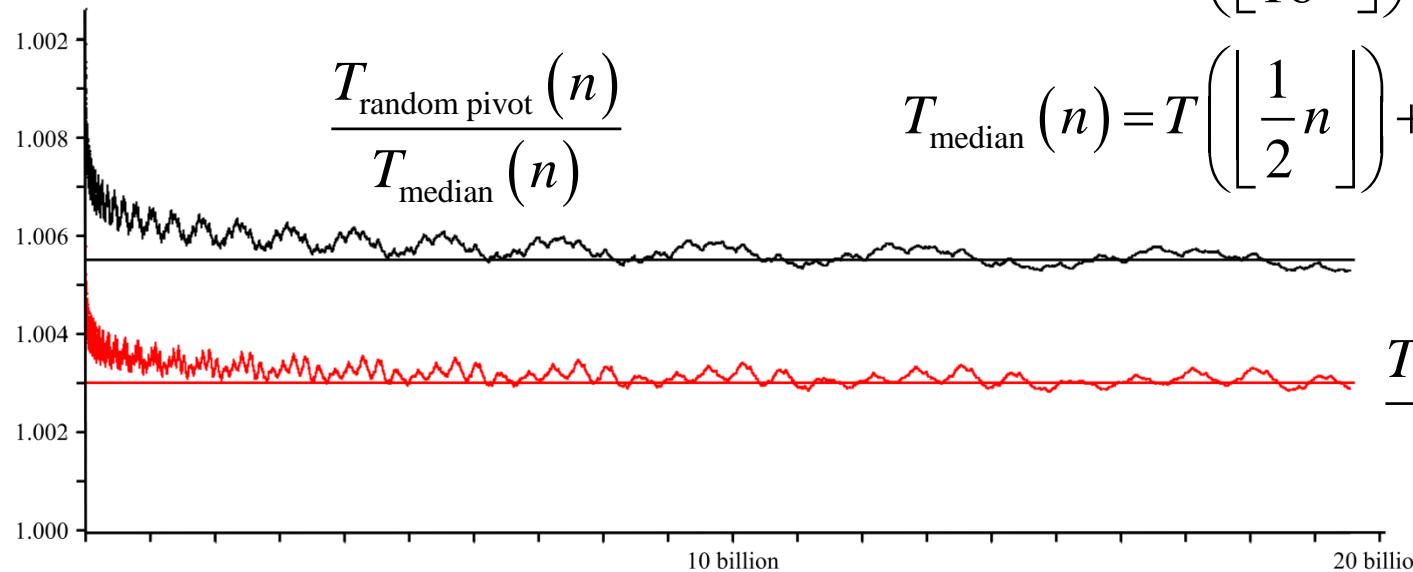




Median-of-three

Question: what is the affect on run time?

- Surprisingly, not so much
- Here we see the ratios of the recurrence relations for large values of n



$$T_{\text{random pivot}}(n) = T\left(\left\lfloor \frac{1}{4}n \right\rfloor\right) + T\left(\left\lceil \frac{3}{4}n \right\rceil\right) + n$$

$$T_{\text{median of 3}}(n) = T\left(\left\lfloor \frac{5}{16}n \right\rfloor\right) + T\left(\left\lceil \frac{11}{16}n \right\rceil\right) + n$$

$$T_{\text{median}}(n) = T\left(\left\lfloor \frac{1}{2}n \right\rfloor\right) + T\left(\left\lceil \frac{1}{2}n \right\rceil\right) + n$$

$$\frac{T_{\text{median of 3}}(n)}{T_{\text{median}}(n)}$$





Implementation

If we choose to allocate memory for an additional array, we can implement the partitioning by copying elements either to the front or the back of the additional array

Finally, we would place the pivot into the resulting hole

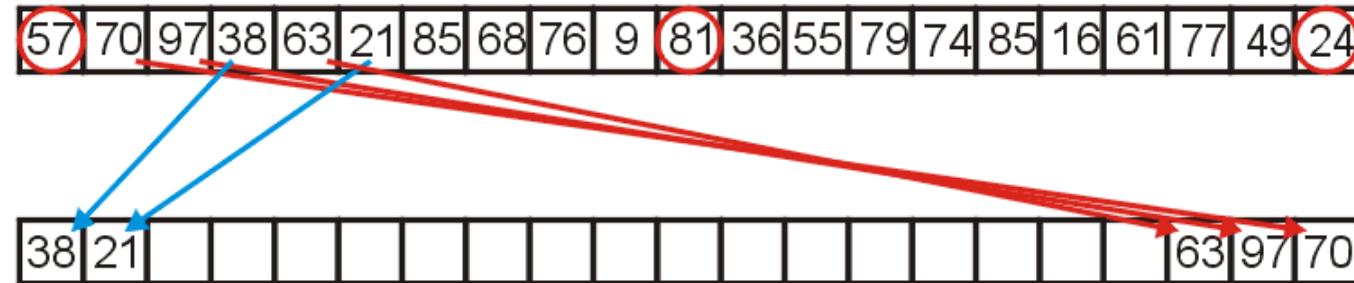




Implementation

For example, consider the following:

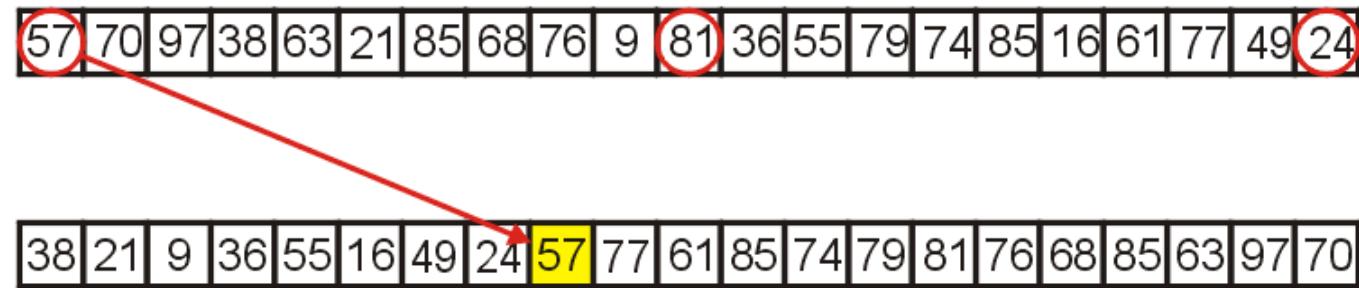
- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array





Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole





Implementation

Note, however, we can do a better job with merge sort, it always divides the numbers being sorted into two equal or near-equal arrays

Can we implement quicksort in place?



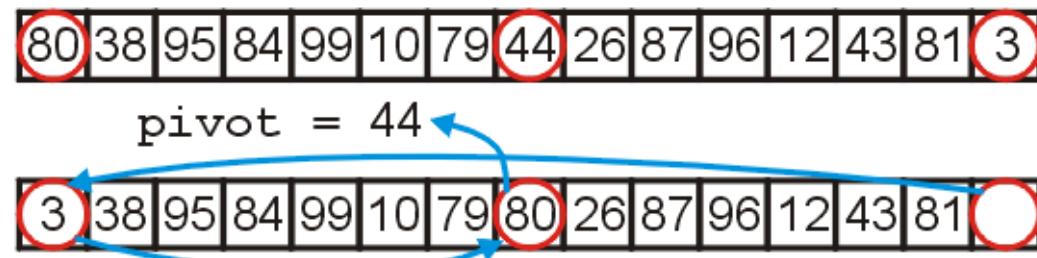


Implementation

First, we have already examined the first, middle, and last entries and chosen the median of these to be the pivot

In addition, we can:

- move the smallest entry to the first entry
- move the largest entry to the middle entry





Implementation

Next, recall that our goal is to partition all remaining elements based on whether they are smaller than or greater than the pivot

We will find two entries:

- One larger than the pivot (staring from the front)
- One smaller than the pivot (starting from the back)

which are out of order and then swap them





Implementation

Continue doing so until the appropriate entries you find are actually in order

The index to the larger entry we found would be the first large entry in the list (as seen from the left)

Therefore, we could move this entry into the last entry of the list
We can fill this spot with the pivot





Implementation

The implementation is straight-forward

```
template <typename Type>
void quicksort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        Type pivot = find_pivot( array, first, last );
        int low   = find_next( pivot, array, first + 1 );
        int high = find_previous( pivot, array, last - 2 );

        while ( low < high ) {
            std::swap( array[low], array[high] );
            low   = find_next( pivot, array, low + 1 );
            high = find_previous( pivot, array, high - 1 );
        }

        array[last - 1] = array[low];
        array[low] = pivot;
        quicksort( array, first, low );
        quicksort( array, high, last );
    }
}
```





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.





Sorting

Quick Sort- III

Dr. Bilal Wajid





Quicksort example

Consider the following unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less





Quicksort example

We call `quicksort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
```

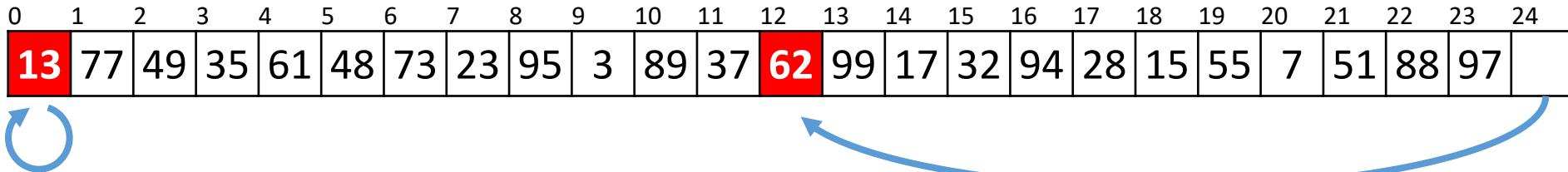
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
pivot = 57;
```

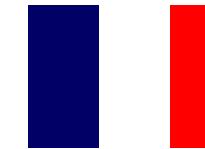
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	51	88	97	



Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

`pivot = 57;`

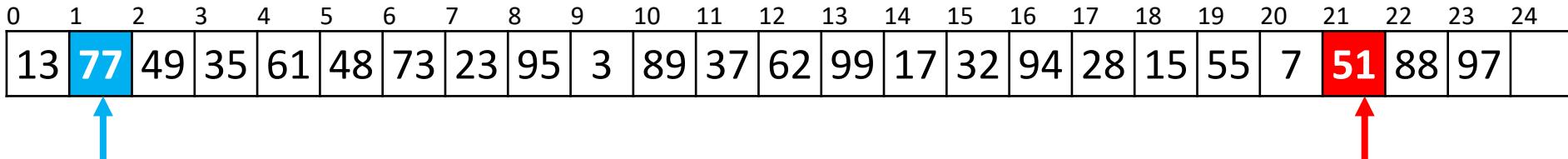
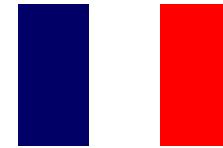
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Searching forward and backward:

`low = 1;`

`high = 21;`

`pivot = 57;`

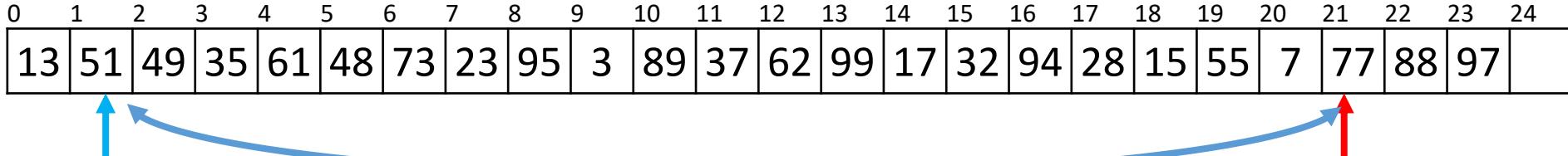
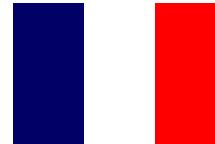
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Searching forward and backward:

`low = 1;`

`high = 21;`

Swap them

`pivot = 57;`

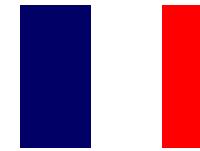
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	77	88	97	

Continue searching

`low = 4;`

`high = 20;`

`pivot = 57;`

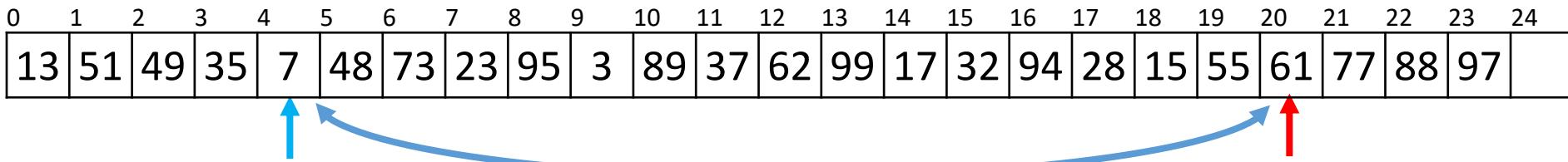
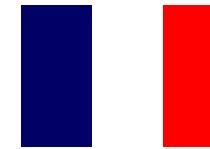
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 4;`

`high = 20;`

Swap them

`pivot = 57;`

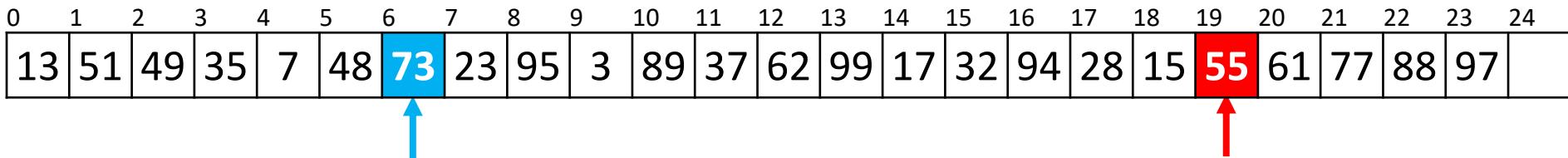
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 6;`

`high = 19;`

`pivot = 57;`

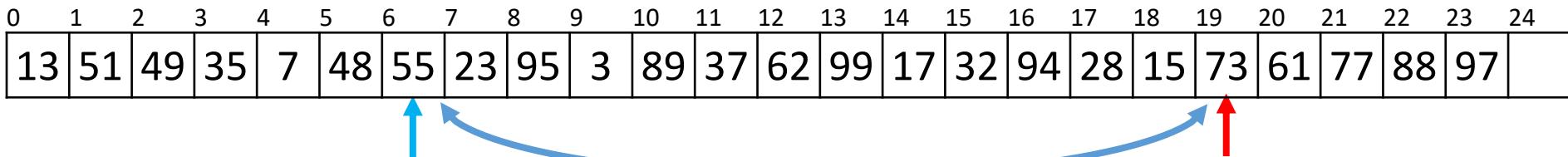
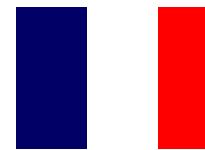
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 6;`

`high = 19;`

Swap them

`pivot = 57;`

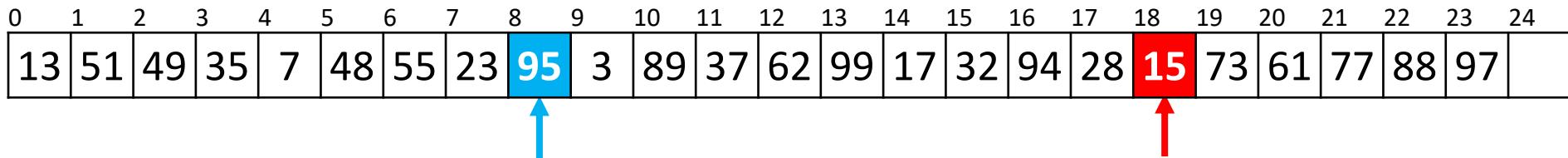
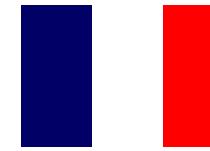
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 8;`

`high = 18;`

`pivot = 57;`

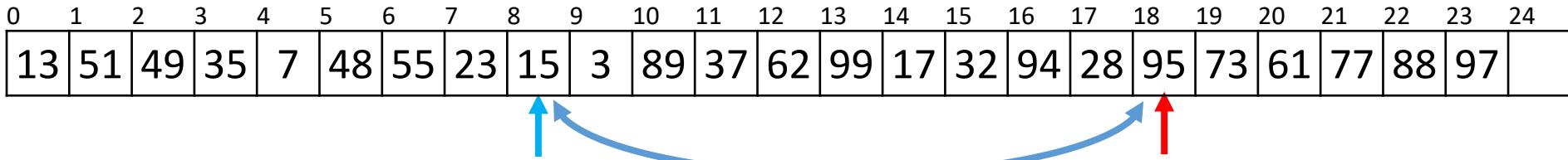
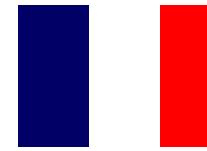
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 8;`

`high = 18;`

Swap them

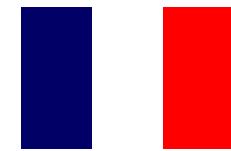
`pivot = 57;`

`quicksort(array, 0, 25)`

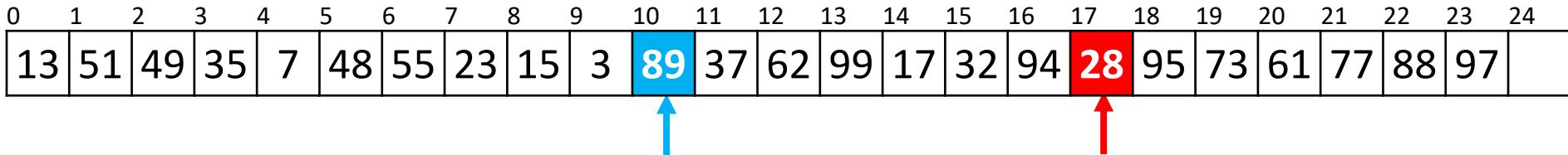




Quicksort example



We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 10;`

`high = 17;`

`pivot = 57;`

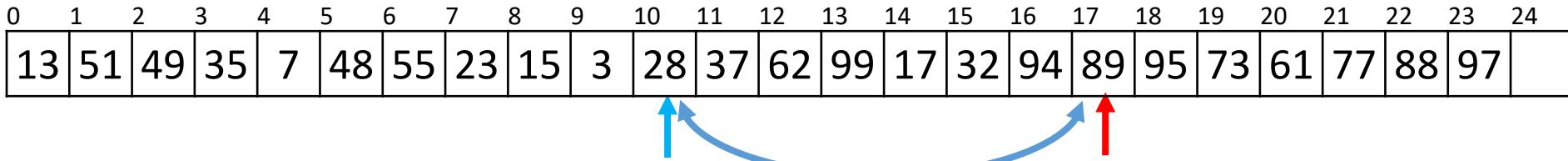
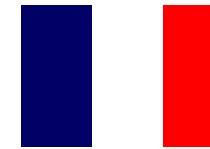
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 10;`

`high = 17;`

Swap them

`pivot = 57;`

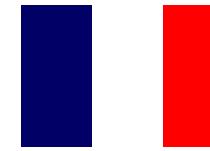
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

A diagram showing an array of 25 integers. The array is indexed from 0 to 24. The element at index 12 is 62 (highlighted in blue), and the element at index 15 is 32 (highlighted in red). A blue arrow points to index 12, and a red arrow points to index 15.

Continue searching

`low = 12;`

`high = 15;`

`pivot = 57;`

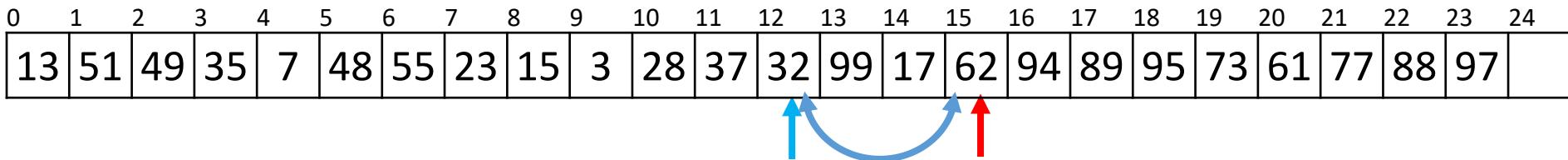
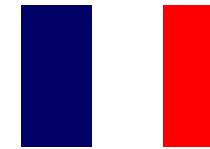
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 12;`

`high = 15;`

Swap them

`pivot = 57;`

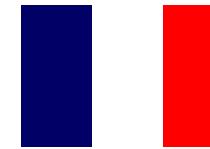
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

A diagram showing an array of 25 integers. The array is indexed from 0 to 24. The elements are: 13, 51, 49, 35, 7, 48, 55, 23, 15, 3, 28, 37, 32, 99, 17, 62, 94, 89, 95, 73, 61, 77, 88, 97, and an empty cell at index 24. The element at index 13 is 99 (highlighted in blue) and the element at index 14 is 17 (highlighted in red). A blue arrow points to index 13 and a red arrow points to index 14, indicating the current partition point.

Continue searching

```
low = 13;  
high = 14;
```

```
pivot = 57;
```

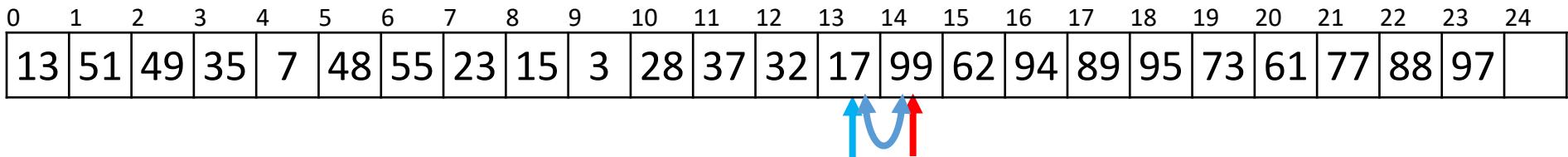
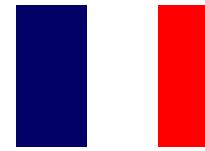
```
quicksort( array, 0, 25 )
```





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 13;`

`high = 14;`

Swap them

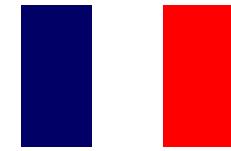
`pivot = 57;`

`quicksort(array, 0, 25)`

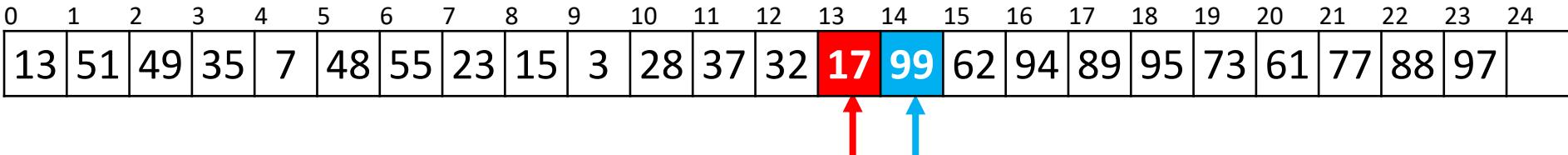




Quicksort example



We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 14;`

`high = 13;`

Now, `low > high`, so we stop

`pivot = 57;`

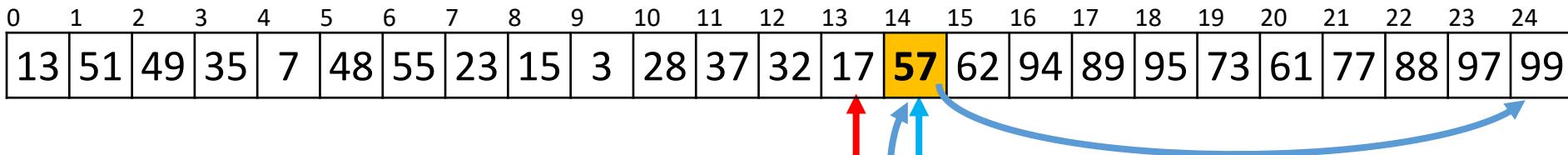
`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`



Continue searching

`low = 14;`

`high = 13;`

Now, `low > high`, so we stop

`pivot = 57;`

`quicksort(array, 0, 25)`





Quicksort example

We are calling `quicksort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half
`quicksort(array, 0, 14);`

`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are executing `quicksort(array, 0, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

`midpoint = (0 + 14)/2; // == 7`

`pivot = 17`

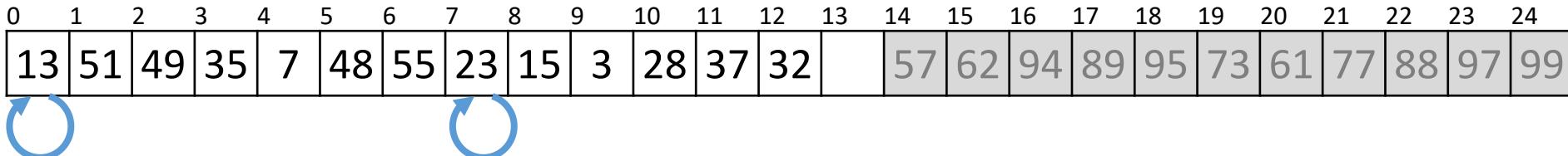
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
pivot = 17;
```

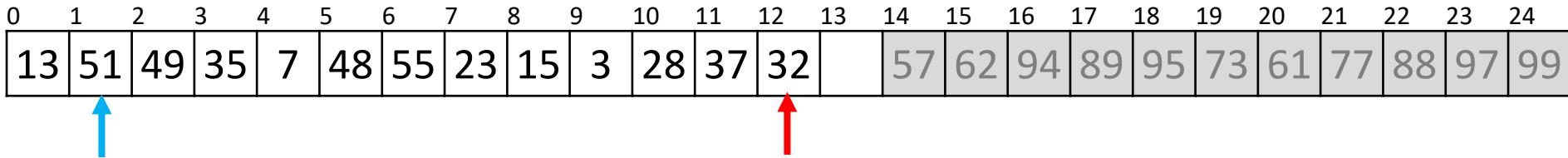
```
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are executing `quicksort(array, 0, 14)`



Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

`pivot = 17;`

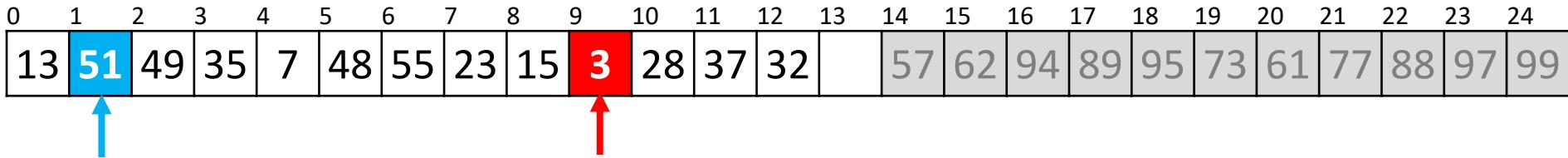
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 1;`

`high = 9;`

`pivot = 17;`

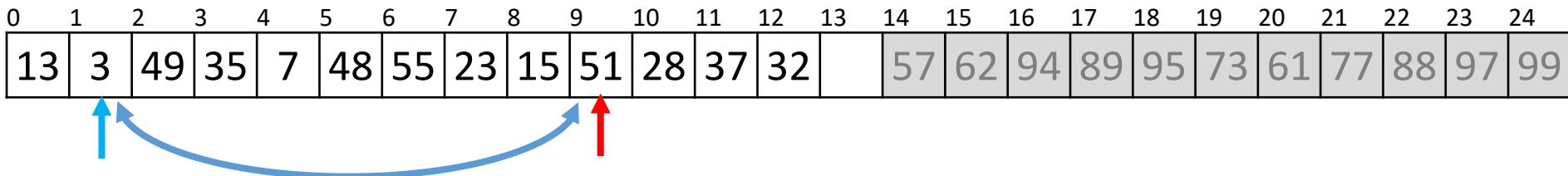
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

```
low = 1;  
high = 9;
```

Swap them

```
pivot = 17;
```

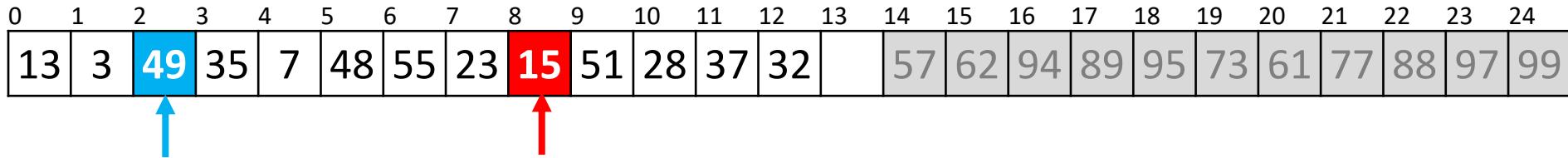
```
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 2;`

`high = 8;`

`pivot = 17;`

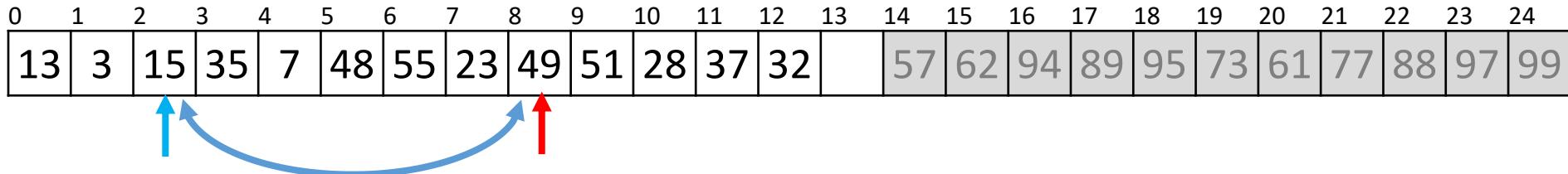
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 2;`

`high = 8;`

Swap them

`pivot = 17;`

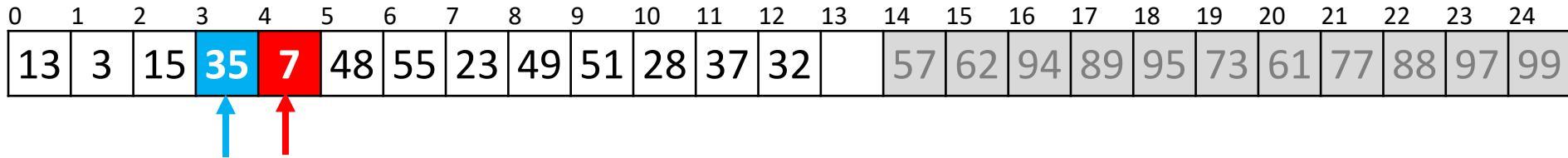
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 3;`

`high = 4;`

`pivot = 17;`

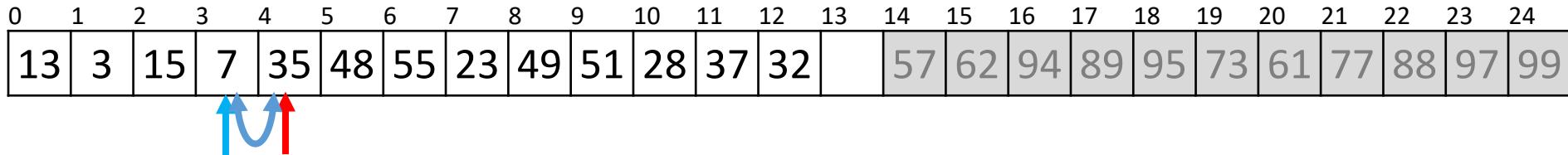
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 3;`

`high = 4;`

Swap them

`pivot = 17;`

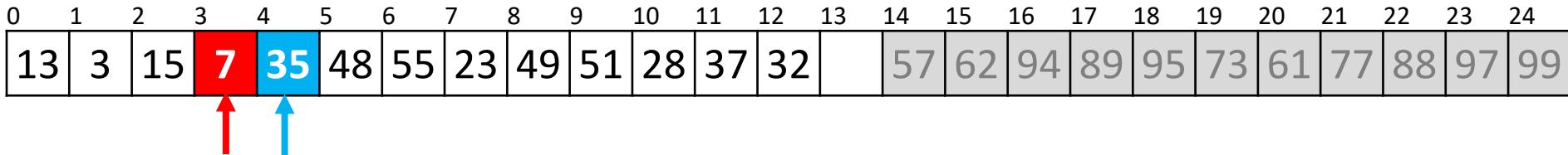
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`



Searching forward and backward:

`low = 4;`

`high = 3;`

Now, `low > high`, so we stop

`pivot = 17;`

`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 0, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively

```
quicksort( array, 0, 4 );
```

```
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are executing `quicksort(array, 0, 4)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array, 0, 4 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 0 to 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 0, 4 )  
quicksort( array, 0, 4 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 0 to 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

- This function call completes and so we exit

```
insertion_sort( array, 0, 4 )  
quicksort( array, 0, 4 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 0, 4 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 0, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

`quicksort(array, 0, 4);`

`quicksort(array, 5, 14);`

`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`





Quicksort example

We now are calling `quicksort(array, 5, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
```

```
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We now are calling `quicksort(array, 5, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
```

```
pivot = 48
```

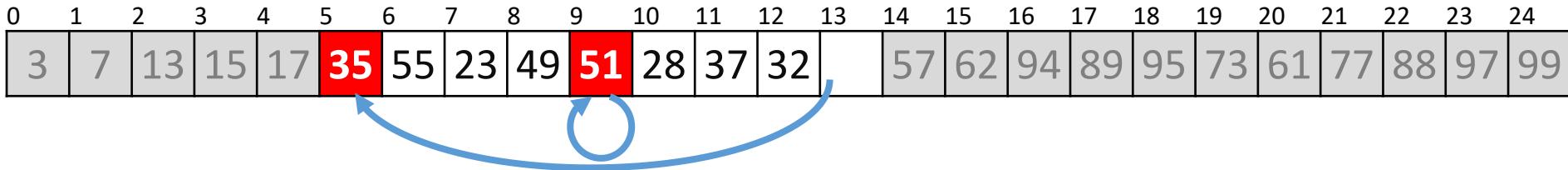
```
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We now are calling `quicksort(array, 5, 14)`



First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
```

```
pivot = 48
```

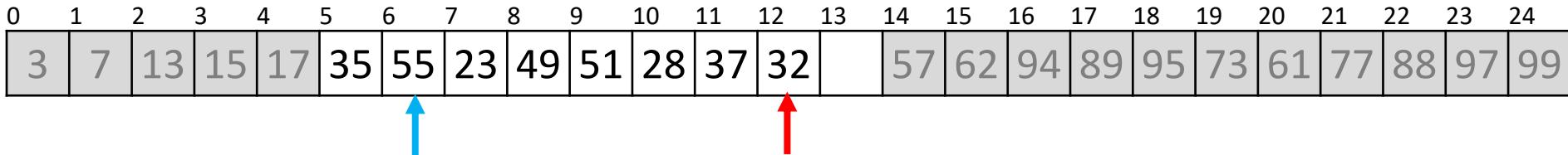
```
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We now are calling `quicksort(array, 5, 14)`



Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

`pivot = 48;`

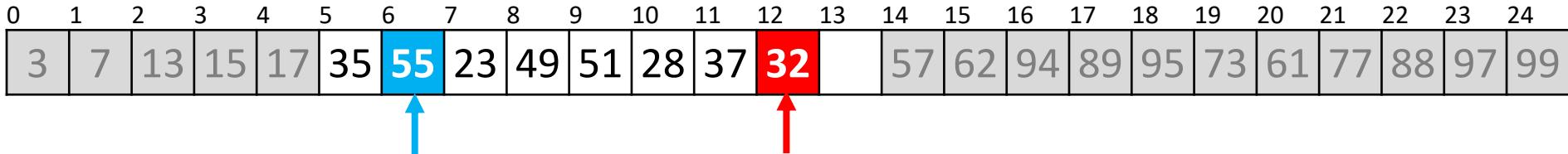
```
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We now are calling `quicksort(array, 5, 14)`



Searching forward and backward:

`low = 6;`

`high = 12;`

`pivot = 48;`

`quicksort(array, 5, 14)`

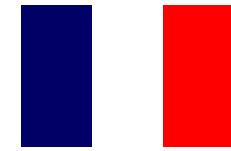
`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`

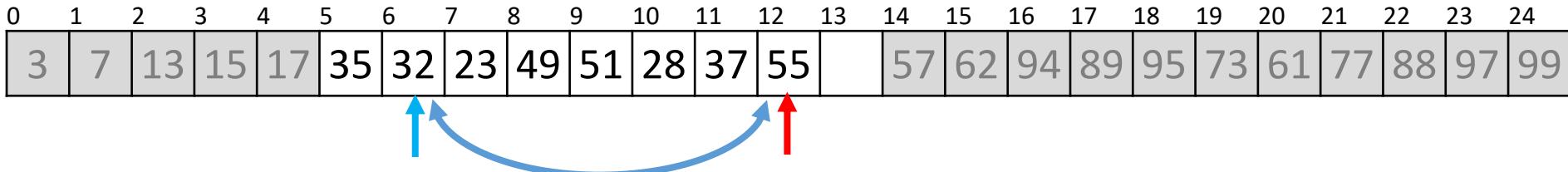




Quicksort example



We now are calling `quicksort(array, 5, 14)`



Searching forward and backward:

`low = 6;`

`high = 12;`

Swap them

`pivot = 48;`

`quicksort(array, 5, 14)`

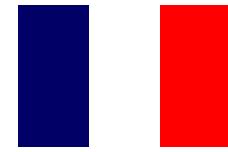
`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`

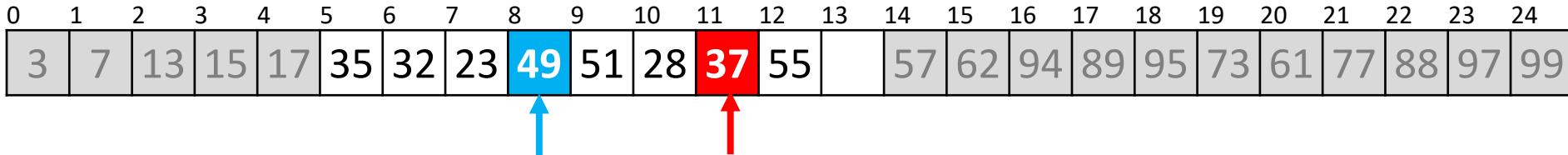




Quicksort example



We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

`pivot = 48;`

`quicksort(array, 5, 14)`

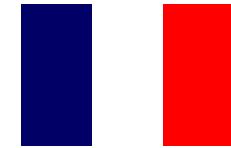
`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`

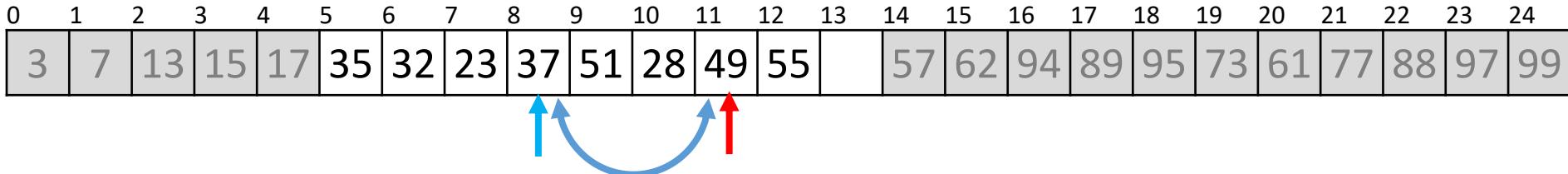




Quicksort example



We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

Swap them

`pivot = 48;`

`quicksort(array, 5, 14)`

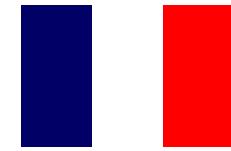
`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`

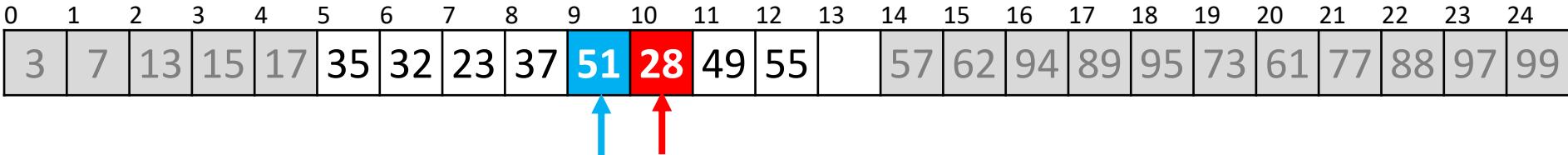




Quicksort example



We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

`pivot = 48;`

`quicksort(array, 5, 14)`

`quicksort(array, 0, 14)`

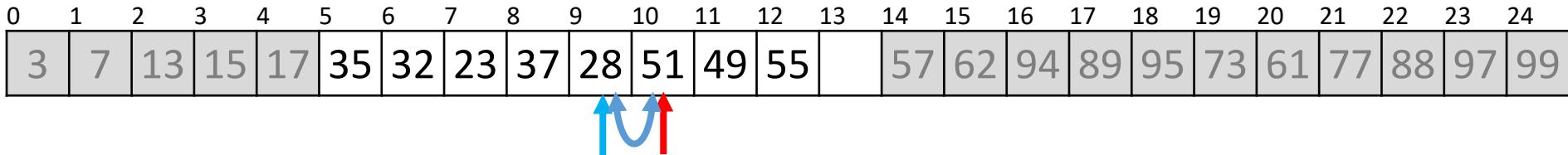
`quicksort(array, 0, 25)`





Quicksort example

We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

Swap them

`pivot = 48;`

`quicksort(array, 5, 14)`

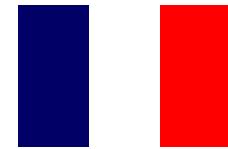
`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`

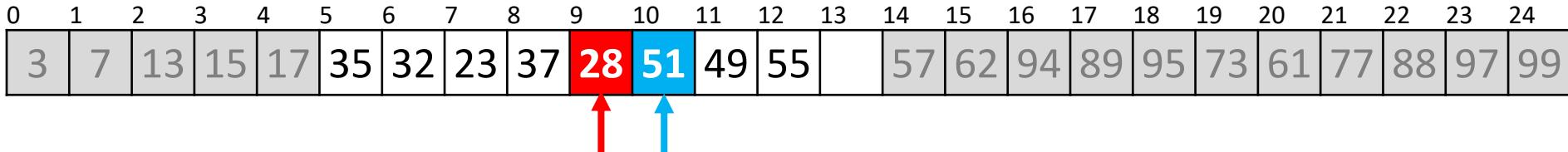




Quicksort example



We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

Now, `low > high`, so we stop

`pivot = 48;`

`quicksort(array, 5, 14)`

`quicksort(array, 0, 14)`

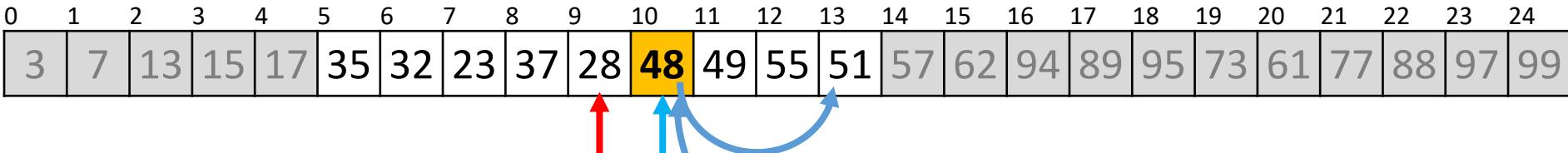
`quicksort(array, 0, 25)`





Quicksort example

We now are calling `quicksort(array, 5, 14)`



Continue searching

`low = 8;`

`high = 11;`

`pivot = 48;`

Now, `low > high`, so we stop

`quicksort(array, 5, 14)`

`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`





Quicksort example

We now are calling `quicksort(array, 5, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half
`quicksort(array, 5, 10);`

`quicksort(array, 5, 14)`
`quicksort(array, 0, 14)`
`quicksort(array, 0, 25)`





Quicksort example

We now are calling `quicksort(array, 5, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively
`quicksort(array, 5, 10);`

```
quicksort( array, 5, 14 )
quicksort( array, 0, 14 )
quicksort( array, 0, 25 )
```





Quicksort example

We are executing `quicksort(array, 5, 10)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, $10 - 5 \leq 6$, so find we call insertion sort

```
quicksort( array, 5, 10 )
quicksort( array, 5, 14 )
quicksort( array, 0, 14 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 5 to 9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 5, 10 )
quicksort( array, 5, 10 )
quicksort( array, 5, 14 )
quicksort( array, 0, 14 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 5 to 9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

- This function call completes and so we exit

```
insertion_sort( array, 5, 10 )  
quicksort( array, 5, 10 )  
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 5, 10 )  
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 5, 14)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

`quicksort(array, 5, 10);`

`quicksort(array, 6, 14);`

`quicksort(array, 5, 14)`

`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 11, 15)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, $15 - 11 \leq 6$, so find we call insertion sort

`quicksort(array, 6, 14)`

`quicksort(array, 5, 14)`

`quicksort(array, 0, 14)`

`quicksort(array, 0, 25)`





Quicksort example

Insertion sort just sorts the entries from 11 to 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 11, 14 )  
quicksort( array, 11, 14 )  
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 11 to 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

- This function call completes and so we exit

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array, 5, 14 )
quicksort( array, 0, 14 )
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 11, 14 )  
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 5, 14 )  
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 0, 14 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 0, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 0, 14 );  
quicksort( array, 15, 25 );
```

`quicksort(array, 0, 25)`





Quicksort example

We are back to executing `quicksort(array, 15, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
```

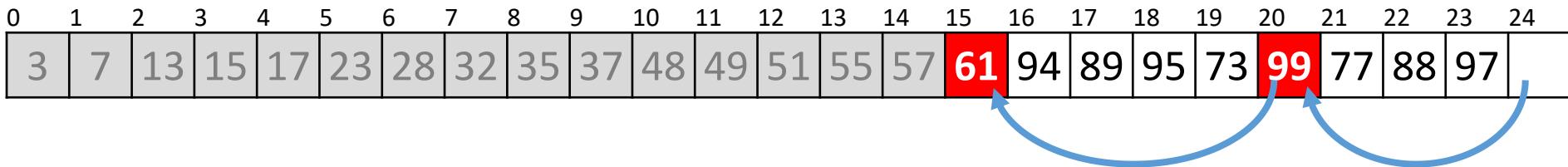
```
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 15, 25)`



First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
```

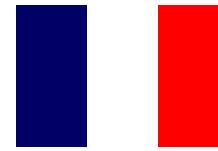
```
pivot = 62;
```

```
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```

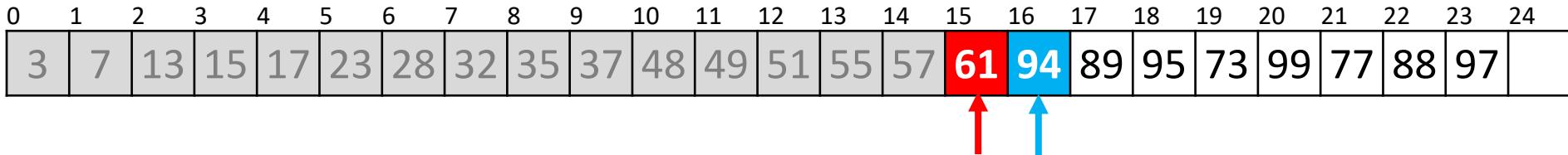




Quicksort example



We are back to executing `quicksort(array, 15, 25)`



Searching forward and backward:

`low = 16;`

`high = 15;`

Now, `low > high`, so we stop

`pivot = 62;`

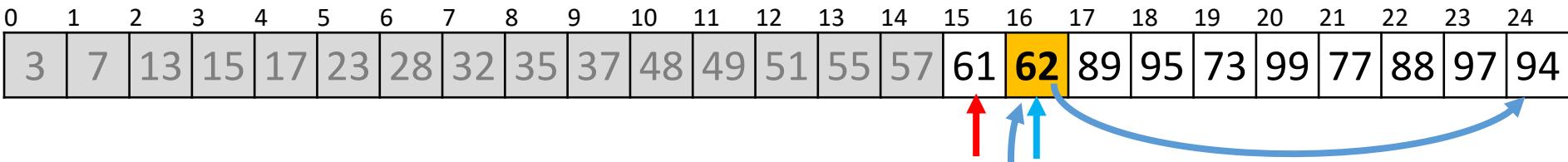
`quicksort(array, 15, 25)`
`quicksort(array, 0, 25)`





Quicksort example

We are back to executing `quicksort(array, 15, 25)`



Searching forward and backward:

`low = 16;`

`high = 15;`

Now, `low > high`, so we stop

- Note, this is the worst-case scenario
- The pivot is the second smallest element

`pivot = 62;`

`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`





Quicksort example

We are back to executing `quicksort(array, 15, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the first half
`quicksort(array, 15, 16);`

`quicksort(array, 15, 16)`
`quicksort(array, 15, 25)`
`quicksort(array, 0, 25)`





Quicksort example

We are executing `quicksort(array, 15, 16)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

Now, $16 - 15 \leq 6$, so find we call insertion sort

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort immediately returns

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

```
insertion_sort( array, 15, 16 )
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

```
quicksort( array, 15, 16 )  
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 15, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the second half

`quicksort(array, 15, 16);`

`quicksort(array, 17, 25);`

`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`





Quicksort example

We are now calling `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
```

```
quicksort( array, 15, 25 )
```

```
quicksort( array, 0, 25 )
```





Quicksort example

We are now calling `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

We are now calling `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

First, $25 - 17 > 6$, so find the midpoint and the pivot

`midpoint = (17 + 25)/2; // == 21`

`pivot = 89`

`quicksort(array, 17, 25)`

`quicksort(array, 15, 25)`

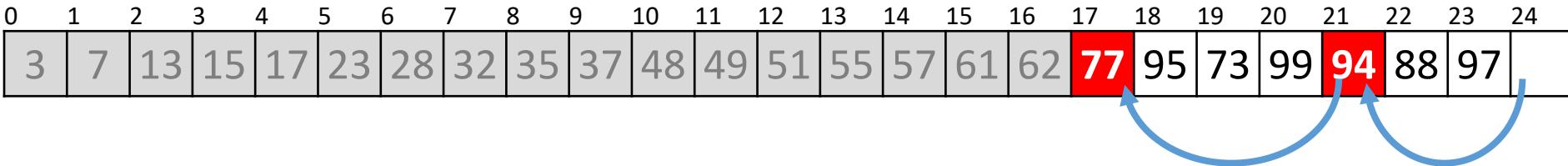
`quicksort(array, 0, 25)`





Quicksort example

We are now calling `quicksort(array, 17, 25)`



First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

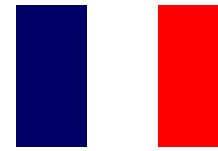
```
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```

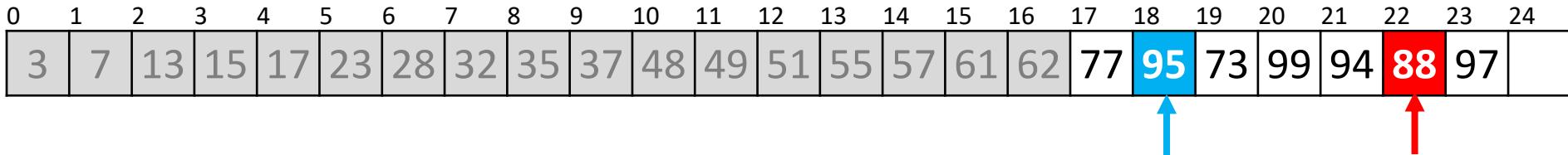




Quicksort example



We are now calling `quicksort(array, 17, 25)`



Searching forward and backward:

`low = 18;`

`high = 22;`

`pivot = 89;`

`quicksort(array, 17, 25)`

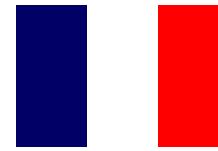
`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`

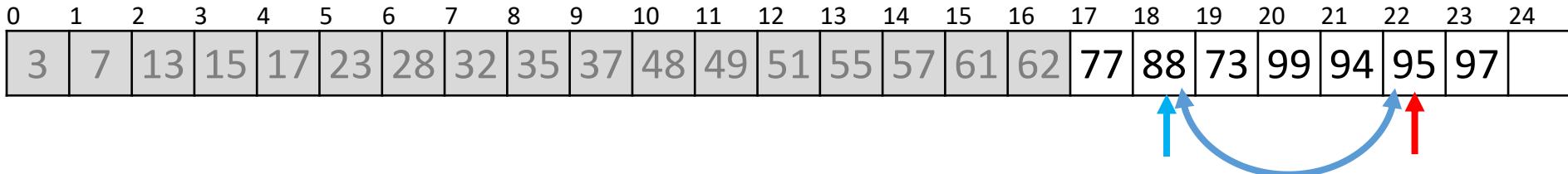




Quicksort example



We are now calling `quicksort(array, 17, 25)`



Searching forward and backward:

`low = 18;`

`high = 22;`

`pivot = 89;`

Swap them

`quicksort(array, 17, 25)`

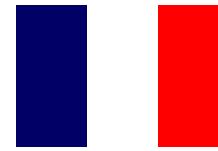
`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`

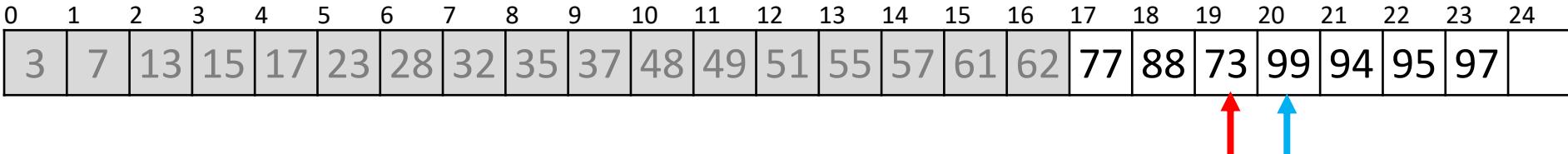




Quicksort example



We are now calling `quicksort(array, 17, 25)`



Searching forward and backward:

`low = 20;`

`high = 19;`

`pivot = 89;`

Now, `low > high`, so we stop

`quicksort(array, 17, 25)`

`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`





Quicksort example

We are now calling `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

Searching forward and backward:

`low = 20;`

`high = 19;`

Now, `low > high`, so we stop

`quicksort(array, 17, 25)`

`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`





Quicksort example

We are now calling `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

We start by calling quicksort recursively on the first half
`quicksort(array, 17, 20);`

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

We are now executing `quicksort(array, 17, 20)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 17 to 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 17 to 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

- This function call completes and so we exit

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

We are back to executing `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

`quicksort(array, 17, 25)`
`quicksort(array, 15, 25)`
`quicksort(array, 0, 25)`





Quicksort example

We are back to executing `quicksort(array, 17, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

We continue by calling quicksort on the second half

```
quicksort( array, 17, 20 );  
quicksort( array, 21, 25 );
```

```
quicksort( array, 17, 25 )  
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

We are now calling `quicksort(array, 21, 25)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Now, $25 - 21 \leq 6$, so find we call insertion sort

`quicksort(array, 21, 25)`

`quicksort(array, 17, 25)`

`quicksort(array, 15, 25)`

`quicksort(array, 0, 25)`





Quicksort example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

- In this case, the sub-array was already sorted
- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 21, 25 )  
quicksort( array, 17, 25 )  
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 25 )  
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 15, 25 )  
quicksort( array, 0, 25 )
```





Quicksort example

This call to quicksort is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

quicksort(array, 0, 25)





Quicksort example

We have now used quicksort to sort this array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99





Black Board Example

Sort the following list using quicksort

- Use insertion sort for any sub-list of size 4 or less

0	1	2	3	4	5	6	7	8	9	10
34	15	65	59	68	42	40	80	50	65	23





Memory Requirements

The additional memory required is $\Theta(\ln(n))$

- In ECE 222, you learned about the memory stack
- Each recursive function call places its local variables, parameters, etc., on a stack
 - The depth of the recursion tree is $\Theta(\ln(n))$
- Unfortunately, if the run time is $\Theta(n^2)$, the memory use is $\Theta(n)$





Run-time Summary

To summarize all three $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Heap Sort		$\Theta(n \ln(n))$		$\Theta(1)$
Merge Sort		$\Theta(n \ln(n))$		$\Theta(n)$
Quicksort	$\Theta(n \ln(n))$	$\Theta(n^2)$	$\Theta(\ln(n))$	$\Theta(n)$





Further modifications

Our implementation is by no means optimal:

An excellent paper on quicksort was written by Jon L. Bentley and M. Douglas McIlroy:

Engineering a Sort Function

found in Software—Practice and Experience, Vol. 23(11), Nov 1993





Further modifications

They detail further suggestions:

- Taking the median of three medians-of-three
 - Expected ratio of 1:1.7292
 - Requires 52 % more depth than merge sort
 - Ratio will be 1:3.3304 or better 90 % of the time
- Better than the median-of-seven but much easier to calculate
- The median of nine would still require 46 % more depth than merge sort

$$F_{\text{median of 3}}(x) = \int_0^1 3! \xi (1 - \xi) d\xi = 3x^2 - 2x^3$$

$$f_{\text{median of medians}}(x) = 3! (3x^2 - 2x^3) \left(1 - (3x^2 - 2x^3)\right) (3!x(1-x))$$

$$2 \int_0^{\frac{1}{2}} x \cdot f_{\text{median of medians}}(x) dx = \frac{469}{1280} = 0.3664$$



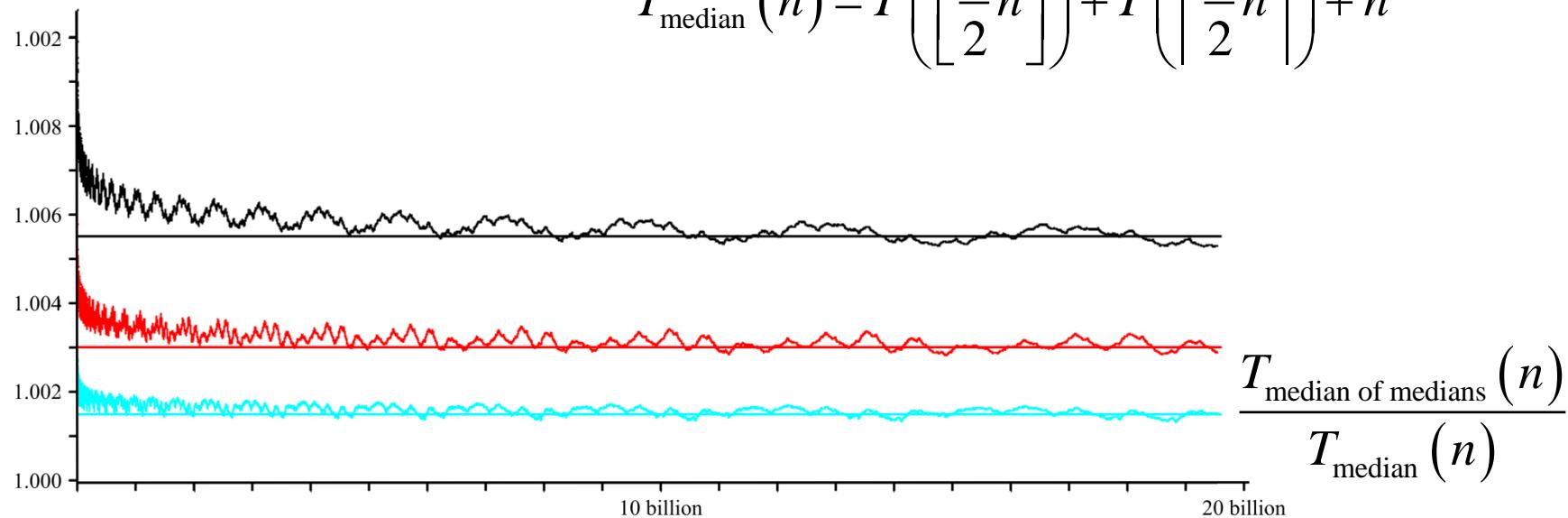


Further modifications

As for the affect on run-time, the ratio of the recurrence relations is now closer to only 0.15 % worse than using the median

$$T_{\text{median of medians}}(n) = T\left(\left\lfloor \frac{469}{1280}n \right\rfloor\right) + T\left(\left\lceil \frac{811}{1280}n \right\rceil\right) + n$$

$$T_{\text{median}}(n) = T\left(\left\lfloor \frac{1}{2}n \right\rfloor\right) + T\left(\left\lceil \frac{1}{2}n \right\rceil\right) + n$$





Further modifications

They detail further suggestions:

- Copy entries equal to the pivot to either end:
 - This requires more tracking of indices (using their notation):



- After the pass, we have:



- Copy the equal entries to the center and only recurs on either side



- Other suggestions are made in the paper, as well...





Summary

This topic covered quicksort

- On average faster than heap sort or merge sort
- Uses a pivot to partition the objects
- Using the median of three pivots is a reasonably means of finding the pivot
- Average run time of $\Theta(n \ln(n))$ and $\Theta(\ln(n))$ memory
- Worst case run time of $\Theta(n^2)$ and $\Theta(n)$ memory





References

Wikipedia, <http://en.wikipedia.org/wiki/Quicksort>

- [1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.1, 2, 3.
- [2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, p.137-9 and §9.1.
- [3] Weiss, *Data Structures and Algorithm Analysis in C++*, 3rd Ed., Addison Wesley, §7.1, p.261-2.
- [4] Gruber, Holzer, and Ruepp, *Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms*, 4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007.

These slides are provided for the ECE 250 *Algorithms and Data Structures* course. The material in it reflects Douglas W. Harder's best judgment in light of the information available to him at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. Douglas W. Harder accepts no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.





Thank you Dr. Muhammad Hasan Jamal and Dr. Douglas Wilhelm Harder (for course material) and flaticon for the nice images.

