Unitedworld Institute Of Technology

शिक्षणतः शिद्धि

B.Tech. Computer Science & Engineering

Semester-3

Data Structures and Algorithms

Course Code: 71203002002

Analyzing Control Statements

For Loop # Input : int A[n], array of n integers # Output : Sum of all numbers in array A

```
Algorithm: int Sum(int A[], int n)
{
                for (int
1               i=0; i<n;
                i++)   s = s
                + A[i];
}                return s;     n
  int s=0;                     n+1
```

Running Time of Algorithm

- The time complexity of the algorithm is : �� �� = �� · �� + ��
- Estimated running time for different values of �� :

�� = ���� 23 steps

�� = ������ 203 steps

�� = �������� 2,003 steps

�� = ���������� 20,003 steps

- As �� grows, the number of steps grow in linear proportion to �� for the given algorithm Sum.
- The dominating term in the function of time complexity is ��: As �� gets large, the +3 becomes insignificant.

▯ The time is linear in proportion to ��.

Analyzing Control Statements

Example 1:

������� = �� + ��;

- Statement is executed once only
- So, The execution time ��(��) is some constant �� ≈ ��(��)

������� �� = 1 ���� �� ����

������� = �� + ��;

- Analysis

��

������� ��

���� �� + �� ����*

�� * ��

Example 2:

������� �� = 1 ���� �� ����

������� = �� + ��;

Example 3:

= 1 ���� �� ����

���� �� + ��

���� * �� + ��

�� �� = $��_1$ �� + 1 + $��_2$�� �� + 1 + $��_3$�� �� ��(��) = $��_1$�� + $��_1$ + $��_2$$��^2$ + $��_2$�� + $��_3$$��^2$ ��(��) = $��^2$($��_2$

$+ ��_3) + ��(��_1 + ��_2) + ��_1$

$��_{��}$

- Total time is denoted as, ��

$�� = ��_{��}�� + ��_{��} + ��_{��}��$

$* ��$

$��(��) = ����^2 + ���� + ��$ $��(��) = ��$ ■$^{��}$

$$��(��) = ��(��_{��} + ��_{��}) + ��_{��} \approx ��(��)$$

Analyzing Control Statements

Example 4:

$�� = 0$

$������ �� = 1 ���� ��$

$����$

$������ �� = 1 ���� ��$

$����$

$������ �� = �� ����$

$�� ����$

$�� = �� + 1$

$�� �� = ��$ ■$^{��}$

Example 5:

$�� = 0$

$������ �� = 1 ���� ��$

$����$

$������ �� = 1 ���� ��^2$

$����$

$������ �� = 1 ���� ��^3$

$����$

$�� = �� + 1$

$�� �� = ��$ ■$^{��}$

Example 6:

$������ �� = 1 ���� ��$

���� + 1

������ �� = 1 ���� �� ����

```
printf(“sum is now %d”,������)
```

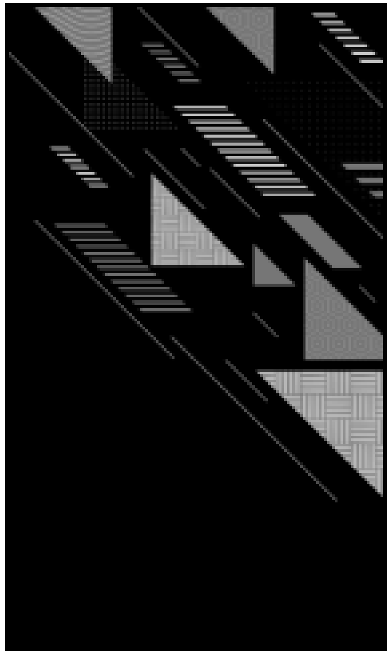
������ = ������ + �� * ��

������ �� = 1 ���� �� ����

������ = ������ − ��

�� �� = �� ■�� + �� �� + �� �� �� �� = �� ■��

�� ■�� �� �� �� �� ��



Sorting Algorithms

Bubble Sort, Selection Sort, Insertion Sort

Introduction

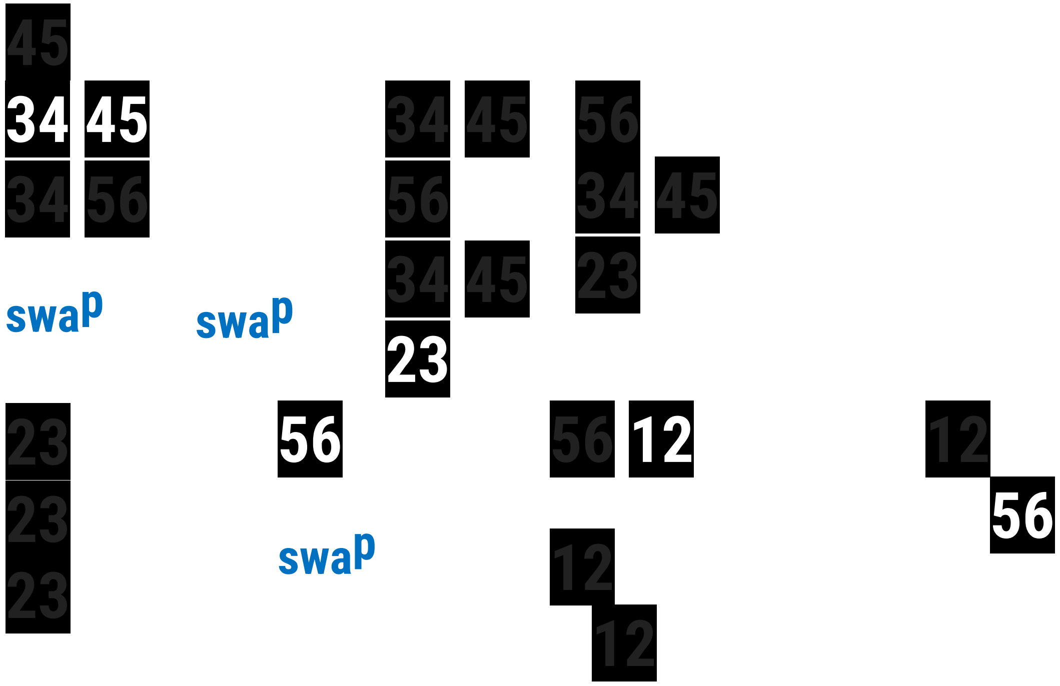
- Sorting is any process of arranging items systematically or arranging items in a sequence ordered by some criterion.
- Applications of Sorting
  1. Phone Bill: the calls made are date wise sorted.
  2. Bank statement or Credit card Bill: transactions made are date wise sorted.
  3. Filling forms online: “select country” drop down box will have the name of countries sorted in Alphabetical order.
  4. Online shopping: the items can be sorted price wise, date wise or relevance wise.
  5. Files or folders on your desktop are sorted date wise.

Bubble Sort – Example

Sort the following array in Ascending order

45	34	56	23	12

Pass 1 :



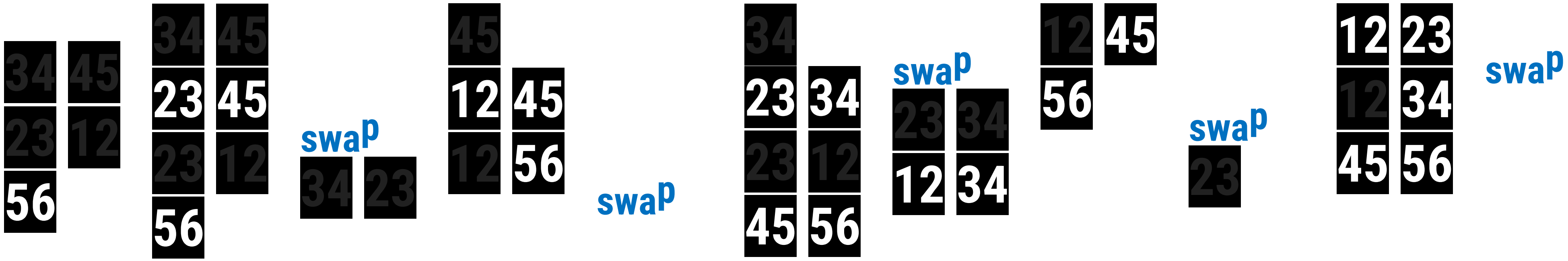
����(��[��] > ��[�� + 1]) ���������(�� [��], ��[�� + 1])

Bubble Sort – Example

Pass 2 :

Pass 3 :

Pass 4 :



����(��[��] > ��[�� + 1]) ���������(�� [��], ��[�� + 1])

Bubble Sort - Algorithm

```
# Input:
Array A
```

```
# Output: Sorted array A

Algorithm: Bubble_Sort(A) for i ← 1 to n-1 do
    for j ← 1 to n-i do  if A[j] > A[j+1] then
        �� ��
        temp ← �� ��
        A[j] ← A[j+1]
        A[j+1] ← temp
        swap(A[j], A[j]   A[j+1])
```

Bubble Sort

- It is a simple sorting algorithm that works by comparing each pair of adjacent items and swapping them if they are in the wrong order.

- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use.

- The time complexity of bubble sort is �� ��

Algorithm: Bubble_Sort(A)

Bubble Sort Algorithm – Best Case Analysis

```
int flag=1;
```

[redacted]

Input: Array A

Output: Sorted array A

Pass 1 :

i = 1

j = 1 j = 2 j = 3

```
for i ← 1 to  n-1 do for j ← 1 to n-i do
```

true

Condition never becomes [redacted]

```
j = 4
cout<<"already sorted"<<endl
if A[j] > A[j+1] then flag = 0;
break;
swap(A[j],A[j+1])
if(flag == 1)
```

Best case time complexity = �� ��

Selection Sort – Example 1

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14

1 2 3 4 5 6 7 8

Step 2 :

Unsorted Array (elements 2 to 8)

▪ **Minj** denotes the current index and **Minx** is the value stored at current index.

-5 5

Swap

Index = 4, value = -5

5 1 12 -5 16 2 12 14 1 2 3 4 5 6 7 8

▪ **So, Minj = 1, Minx = 5**

▪ Assume that currently **Minx** is the smallest value. ▪ Now find the smallest value from the remaining entire Unsorted array.

Selection Sort – Example 1

Step 3 :

Unsorted Array (elements 3 to 8)

▪ **Now Minj = 2, Minx = 1** ▪ Find min value from remaining

-5 1 12 5 16 2 12 14

1

unsorted array

1 2 3 4 5 6 7 8

Index = 2, value = 1

No Swapping as min value is already at right place

Step 4 :

Unsorted Array (elements 4 to 8)

-5 1 12 5 16 2 12 14
2 12

1 2 3 4 5 6 7 8

- **Minj = 3, Minx = 12**
- Find min value from remaining unsorted array

Index = 6, value = 2

Swap

Selection Sort – Example 1

8) min value from remaining

Step 5 :

Unsorted Array (elements 5 to 8)

▪ **Now Minj = 4, Minx = 5** ▪ Find min value from remaining unsorted array

-5 1 2 5 16 12 12 14 5

Index = 4, value = 5

1 2 3 4 5 6 7 8

Step 6 :

No Swapping as min value is already at right place ▪ **Minj = 5, Minx = 16**

Unsorted Array (elements 6 to 8)

-5 1 2 5 16 12 12 14 12 16

1 2 3 4 5 6 7 8 Swap

▪ Find min value from remaining unsorted array

Index = 6, value = 12

Selection Sort – Example 1

8)

Step 7 :

Unsorted Array (elements 7 to

▪ **Now Minj = 6, Minx = 16** ▪ Find min value from remaining

-5 1 2 5 12 16 12 14

Index = 7, value = 12

12 16

1 2 3 4 5 6 7 8 Swap

Step 8 :

Unsorted Array (element 8)

- **Minj = 7, Minx = 16**
- Find min value from remaining unsorted array

-5 1 2 5 12 12 16 14

Index = 8, value = 14

14 16

1 2 3 4 5 6 7 8 Swap

The entire array is sorted now.

unsorted array

Selection Sort

- Selection sort divides the array or list into two parts,
  1. The sorted part at the left end
  2. and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- Then it finds the second smallest element and exchanges it with the element in the second leftmost position.
- This process continues until the entire array is sorted.
- The time complexity of selection sort is �� ��

Selection Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
minj ← i;
minx ← A[i];
```

```
for j ← i + 1 to n do
if A[j] < minx then
minj ← j;
minx ← A[j];
A[minj] ← A[i];
A[i] ← minx;
```
�� ��

�� ��

Selection Sort – Example 2

```
Algorithm: Selection_Sort(A) for i ← 1 to
n-1 do
minj ← i; minx ← A[i];
for j ← i + 1 to n do
```

i = 1

2

Pass 1 :

```
if A[j] < minx then A[j];
minj ← j ; minx ←
```

minj ← 1 34

minx ← 45

No Change

j = 2 3

```
A[minj] ← A[i];
A[i] ← minx;
```

Sort in Ascending order

A[j] = 3456

45	34	56	23	12

1 2 3 4 5

Selection Sort – Example 2

Pass 1 :

Algorithm: Selection_Sort(A) **for** i ← 1 to n-1 **do**

i = 1

minj ← i; minx ← A[i];

4

for j ← i + 1 to n **do**

if A[j] < minx **then** minx ← 34 12

minj ← j ; minx ← A[j]; 23

minj ← 2 5

A[i]; 3 4 5

A[minj] ← j = 2

A[i] ← minx;

A[j] = 2312

Unsorted Array

Sort in Ascending order

45	34	56	23	12

1 2 3 4 5

Swap

45	

45 12	34	56	23	1245

1 2 3 4 5

			34 56	

Insertion Sort – Example

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 : **Unsorted Array**

5	1	12	-5	16	2	12	14

1 2 3 4 5 6 7 8

Step 2 :

�� 1

�� = ��, �� = ��   �� > ��

�� = �� − �� ������

[redacted] 1 12 -5 16 2 12 14 1 2 3 4 5 6 7 8

while �� < �� �� do
�� �� + 1 ← �� ��

�� — — Shift down

Insertion Sort – Example

Step 3 :

��   ����   ������ �� > ��

�� = ��, �� =   �� = �� − ��

1 5 12 -5 16 2 12 14 1 2 3 4 5 6 7 8

while �� < �� �� do �� �� +
1 ← �� ��
�� − −

No Shift will take place

Step 4 :

−��
�� = �� – ��
������ �� >
��

�� = ��, �� =

��

��

-5

while �� < �� �� do

�� �� + 1 ← �� ��
�� − −

1 5 12 -5 16 2 12 14 1 2 3 4 5 6 7 8

Shift down Shift down Shift down

Insertion Sort – Example

Step 5 :

��

�� = ��, �� = ����

�� = �� – �� ������ �� > ��

-5 1 5 12 16 2 12 14 1 2 3 4 5 6 7 8

while �� < �� �� do �� �� + 1 ← �� ��
�� – –

No Shift will take place

Step 6 :

��

�� = �� – �� ������ �� > ��

�� = ��, �� =

�� ��

1 2 3 4 5 6 7 8

while �� < �� �� do �� �� + 1 ← �� ��
�� – –

-5 1 5 12 16 2 12 14

2

down n
Shif Shift
t down
Shift dow

Insertion Sort – Example

Step 7 : ��

-5 1 2 5 12 16 12 14
12

1 2 3 4 5 6 7 8 Shift down

�� = ��, �� = ���� �� = �� – ��
������ �� > ��
�� �� + 1 ← �� ��
�� − −

Step 8 :

while �� < �� �� do

�� = ��, �� = ���� �� = �� – ��
������ �� > ��

�� while �� < �� �� do

�� − −

14

-5 1 2 5 12 12 16 14 1 2 3 4 5 6 7 8

The entire array is sorted now.

Shift down

�� �� + 1 ← �� ��

j ← i − 1;

while x < T[j] and j > 0 do

Insertion Sort - Algorithm

```
# Input: Array T
# Output: Sorted array T

Algorithm:
Insertion_Sort(T[1,…,n]) for i ← 2 to n do
x ← T[i];
```

�� ��

j ← j − 1;
T[j+1] ← x;

T[j+1] ← T[j];

Insertion Sort Algorithm – Best Case Analysis

Input: Array T
Output: Sorted array T

Insertion_Sort(T[1,…,n])

Algorithm:

for i ← 2 to n do x ← T[i];
j ← i − 1;

12

Pass 1 :

23 x=23 i=2

T[j]=12

i=3 i=4 i=5

T[j]=23

x=34 x=45

T[j]=34

34 45 59

x=59

T[j]=45

```
while x < T[j] and j > 0 do
        T[j+1] ← T[j];
        j ← j - 1;
T[j+1] ← x;
```

The best case time complexity of Insertion sort is �� ��

The average and worst case time complexity of Insertion sort is �� ��

Heap & Heap Sort Algorithm

Introduction

- A heap data structure is a binary tree with the following two properties.

1. It is a complete binary tree: Each level of the tree is completely filled, except possibly the bottom level. At this level it is filled from left to right.
2. It satisfies the **heap order** property: the data item stored in each node is **greater than or equal to** the data item stored in its children node.

a

b c

d e f

Binary Tree but not a Heap

a

b c d e f

9

Complete Binary Tree - Heap

6 7

2 4 8

9

Not a Heap

6 7 2 4 1

Heap

Array Representation of Heap Heap can be implemented using an Array.

 An array �� that represents a heap is an object with two attributes: 1. ����������h[��], which is the number of elements in the array, and 2. h�������−���������[��], the number of elements in the heap stored within array ��

16

14 10

8 7 9 3

Array representation of heap

Heap

16	14	10

2 4 1

Array Representation of Heap In the array ��, that represents a heap

1. length[��] = heap-size[��]
2. For any node �� the parent node is ��/��
3. For any node ��, its left child is ���� and right child is ����+��

For node �� $= 4$, parent node is $4/2 = 2$

��
16

�� ��

14 10

�� �� �� �� 8 7 9 3

��

�� ����

2 4 1 **Heap**

For node �� $= 4$,

Left child node is $2 * 4 =$ node 8

Right child is $2 * 4 + 1 =$ node 9

�� �� �� �� �� �� �� �� �� ����

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

Types of Heap

1. **Max-Heap** – Where the value of the root node is greater than or equal to either of its children.

9

6 7 2 4 1

1

2 4 6 7 9

2. **Min-Heap** – Where the value of the root node is less than or equal to either of its children.

Introduction to Heap Sort

1. Build the complete binary tree using given elements.
2. Create Max-heap to sort in ascending order.
3. Once the heap is created, swap the last node with the root node and delete the last node from the heap.
4. Repeat step 2 and 3 until the heap is empty.

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 1 : Create Complete Binary Tree

4

1 2 3 4 5

4	10	3	5	1

10 3 5 1

Now, a binary tree is created and we have to convert it into a Heap.

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3

Step 2 : Create Max Heap

4

1 2 3 4 5

10

10 is greater than 4 So, swap 10 & 4

4	10	3	5	1
10	4			

10 3 4

Swap

5 1

In a Max Heap, parent node is always greater than or equal to the child nodes.

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 2 : Create Max Heap

10

1 2 3 4 5

10	4 5	3	5 4	1

5

5 is greater than 4 So, swap 5 & 4

4 3

Swap

In a Max Heap, parent node

4

is always greater than or equal
to the child nodes.

Max Heap is created

5 1

⚠

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1

1 2 3 4 5

10 1	5	3	4	110

1. Swap the first and the last nodes and
2. Delete the last node.

10

Swap

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

10

4 1

5 3

5

1

1 2 3 4 5 **Max Heap Property is**

violated so, create a

4
1

1 5	5 1 4	3	4 1	**10**

1
4

Swap

Max Heap again.

5 3

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1

Heap Sort – Example 1

Sort the following elements in Ascending order

1 2 3 4 5 **Max Heap is created** 5

5 1	4	3	1 5	**10**

Swap

1. Swap the first and the last nodes and
2. Delete the last node.

1

4 3

5

4	10	3

1

Create Max Heap

Step 3 : Apply Heap Sort

3

1	2	3	4	5
1 4 3	4 1	3 4	**5**	**10**

1. Swap the first and the last nodes and
2. Delete the last node.

4

4 3 1 4

Swap

again

Max Heap is created

Already a Max Heap

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3

Step 3 : Apply Heap Sort

1

3

1	2	3	4	5
3 1	1 3	**4**	**5**	**10**

3

1

Swap

1. Swap the first and the last nodes and
2. Delete the last node.

Step 3 : Apply Heap Sort

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3

Remove the last element from heap and the sorting is over.

1

1 2 3 4 5

1 1	3	4	5	10

Already a Max Heap

Heap Sort – Example 2

Sort given element in ascending order using heap sort. 19, 7, 16, 1, 14, 17

19	7	16	1	14	171

	14	17		7	6

Step 1: Create binary tree Step 2: Create Max-heap

19

1 14 17

7 16 17

7 16

19

1 14 167 7

14

16 17	14	17 16	1	7	

Heap Sort – Example 2 Step 3 Step 4

19 16	14	17	1	7	161 9

16 Create Max-heap

19

Swap & remove the last element

17

14 17 16

14 17

16

1 7

1 7 16 19

Heap Sort – Example 2

Step 5 Step 6

17 7	14	16	1	7 17	19

7 16	14	16 7	1	17	

7 Create Max-heap

17

Heap Sort – Example 2

Step 7 Step 8

Swap & remove

the last element

14 16

16

7

7

17

17

16

14 1

1

14

16

16 1	14	7	1 16	17	19

Create Max-heap

1 14	14 1	7	16	17	

Heap Sort – Example 2

Step 9 Step 10

14 7	1	7 14	16	17	19

Swap & remove the last element

16

1

14 7

Swap & remove the last element

14 7 1

7 1	1 7	14	16	17	

Already a Max-heap

7

1

Swap & remove the last

14

7

1 7 14

7

1

Step 11

element

1 | Remove the last element

The entire array is sorted now.

Exercises

Sort the following elements using Heap Sort Method.

1. 34, 18, 65, 32, 51, 21
2. 20, 50, 30, 75, 90, 65, 25, 10, 40

Sort the following elements in Descending order using Hear Sort Algorithm.

1. 65, 77, 5, 23, 32, 45, 99, 83, 69, 81

Binary Tree Analysis

Heap Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
     BUILD-MAX-HEAP(A)
```

```
for i ← length[A] downto 2
        do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MAX-HEAPIFY(A, 1, n)
```

4	

Heap Sort – Algorithm

Algorithm:

```
BUILD-MAX-HEAP(A) heap-size[A] ←
length[A] for i ← ⌊length[A]/2⌋ downto
1  do MAX-HEAPIFY(A, i)
```

1

4

2 3

1 3

4 5 6

2 9 7

heap-size[A] = 6

4	1	7	2	9	3

4	9	7	2	1	3

9	4	7	2	1	3

i = 3

1

i = 2 i = 1

1

4

1

4

4 9

2 3

2 3

2 3

1 3

1 7

9 7

7

9

4

4 5 6 2 9 7

4 5 6 2 9 3

4 5 6 2 1 3

3

1

Heap Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A
```

39	4	7	2	1	93

1

9

```
Algorithm: Heap_Sort(A[1,…,n])
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2
```

Heap Sort – Algorithm

Algorithm: Max-heapify(A,i, n)

3	4	7	2	1	9

do exchange A[1] A[i]

heap-size[A] ← heap-size[A] –

1 MAX-HEAPIFY(A, 1, n)

2 3

4 7

4

5 6 **2 1 3**

l← LEFT(i) r← RIGHT(i)

l ←2 1

r ←3

ifl≤n and A[l]> A[i]

Yes

3

1

then largest←l

else largest←i

largest← 2

23

47

4 5

ifr≤n and A[r]> A[largest]

Yes

21

then largest←r

largest← 3

if largest≠i

Yes

then exchange A[i] A[largest]
MAX-HEAPIFY(A, largest, n)

Output: Sorted array A

3	4	7	2	1	9

Heap Sort – Algorithm

Input: Array A

1

Algorithm: Heap_Sort(A[1,...,n])

7

BUILD-MAX-HEAP(A)

for i ← length[A] downto 2

do exchange A[1] A[i]

heap-size[A] ← heap-size[A] -

1 MAX-HEAPIFY(A, 1, n)

4 5

2 3

4 3

2 1

1

Algorithm:
Heap_Sort(A[1,…,n])

Heap Sort Algorithm – Analysis

Input: Array A
Output: Sorted array A

��T��

heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto

(A)

BUILD-MAX-HEAP

��(��

��������)

i)

do MAX-HEAPIFY(A, ��(���������)

for i ← length[A] downto 2

do exchange A[1] A[i]

heap-size[A] ←
heap-size[A] − 1

�� − ��

��(�� − ��)
(���������)

MAX-HEAPIFY(A, 1, n)

Running time of heap sort algorithm is:

�� �� ������� �� + ��(������ ��) �� − �� + ��(�� − ��) = ��(�� ������� ��)

Sorting Algorithms

Radix Sort, Bucket Sort, Counting Sort

Radix Sort

- Radix Sort puts the elements in order by comparing the digits of the numbers.
- Each element in the ��-element array �� has �� digits, where digit 1 is the lowest-order

digit and digit �� is the highest order digit.

Algorithm: RADIX-SORT(A, d)

for i ← 1 to d

do use a stable sort to sort array A on digit i

- Sort following elements in Ascending order using radix sort.

363, 729, 329, 873, 691, 521, 435, 297

Radix Sort - Example

3 6 3
7 2 9

3 2 9
8 7 3
6 9 1
5 2 1
4 3 5

2 9 7

6 9 1 5 2 1 3 6 3 8 7 3 4 3
5 2 9 7 7 2 9 3 2 9

5 2 1 | 7 2 9 | 3 2 9 | 4 3 5 | 3 6 3 | 8 7 3 | 6 9 1 | 2 9 7

Sort on column 3

Sort on column 2

Sort on column 1

The entire array is sorted now.

Bucket Sort – Introduction

 Sort the following elements in Ascending order using bucket sort.

45	96	29	30	27	12	39	61	91

1. Create �� empty buckets.
2. Add each input element to appropriate bucket as,
   a. Bucket �� holds values in the half-open interval,

$$�� * ���� \leq ��[��] < (�� + ��) * ����$$

3. Sort each bucket queue with insertion sort.

4. Merge all bucket queues together in order.

 Expected running time is ��(�� + ��), with �� = size of original sequence. If �� is ��(��) then sorting algorithm in ��(��).

Bucket Sort – Example

45	96	29	30	27	12	39	61	91

45 96 29 30 27 12 39 61 91

0 1 2 3 4 5 6 7 8 9

Sort each bucket queue with insertion sort

Merge all bucket queues together in order

12	27	29	30	39	45	61	91	96

Bucket Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bucket-Sort(A[1,...,n])
    n ← length[A]
    for i ← 1 to n do
        insert A[i] into bucket B[⌊A[i] ÷ n⌋]
```

```
for i ← 0 to n - 1 do
     sort bucket B[i] with insertion sort
concatenate the buckets B[0], B[1], . . ., B[n - 1] together in
order.
```

⚠

Counting Sort – Example Sort the following elements in Ascending order using counting sort.

3	6	4	1	3	4	1	4	2

Step 1
Given elements are stored in an input array ��[1, … , 9]

1	2	3	4	5	6	7	8	9
3	6	4	1 3		4	1	4	2

Step 2
Define a temporary array ��.
The size of an array �� is

Index

Elements

equal to the **maximum element** , 6] to 0.
in array ��. Initialize ��[1, ...

Index

Elements

0	0	0	0	0	0

1	2

Counting Sort – Example

 Sort the following elements in Ascending order using counting sort.

Input array ��

3	6	4

Step 3

Update an array C with the occurrences of each value of array ��

Index

Elements

1	2	3	4	5	6
02	10	20	03	00	10

++

Step 4

In array ��, from index 2 to ��

add the value with previous element

2	3	5	8	8	9

Index		
Elements	1	2

Counting Sort – Example

 Create an output array ��[1…9]. Start positioning elements of Array �� ���� �� as shown below.

Input array ��	1	2	3	4	5	6	7	8	9
	3	6	4	1	3	4	[redacted]	[redacted]	[redacted]

Temporary Array C

1	2
[redacted]	[redacted]

Output Array B

1	2	3	4	5	6	7	8	9
1	1	2	3	3	4	4	4	

Counting Sort - Procedure

- Counting sort assumes that each of the �� input elements is an integer in the range 0 to ��, for some integer ��.
- When ��=��(��), the counting sort runs in ��(��) time.
- The basic idea of counting sort is to determine, for each input element ��, the number of elements less than ��.
- This information can be used to place element �� directly into its position in the output array.

Counting Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Counting-Sort(A[1,...,n], B[1,...,n], k)
for i ← 1 to k do
    c[i] ← 0
for j ← 1 to n do
    c[A[j]] ← c[A[j]] + 1
for i ← 2 to k do
    c[i] ← c[i] + c[i-1]
for j ← n downto 1 do
    B[c[A[j]]] ← A[j]
    c[A[j]] ← c[A[j]]
```

Thank You!