1. # INSERTION SORT

This is an efficient algorithm for sorting small no of elements. It works in the same way as cards are sorted we suppose that we have empty left hand and the cards are ~~limited~~ lined on the table we then remove one card at the time from the table and insert it into the correct position in the left hand to find a correct position for a card we compare it with each of the cards already in hand from right to left.

Insertion Sort (A).

```
1. for j ← 2 to length(A)
2.   do key ← A[j]
3. // insert A[j] into the sorted sequence
4.   i ← j-1
5. while i>0 and A[j] > key
6.   do A[j+1] ← A[i]
7.   i ← j-1
8.   A[i+1] ← key
```

The index j indicates the current card being inserted into hand.

Array elements 1 to j-1 constitute currently sorted hands.

A[j+1 ... n] corresponds to the pile of cards still left on the table.

The index j moves from left to right.

Eg. 5, 2, 4, 6, 1, 3

↓ ↓

i j

Page No.
Date : / /

Key ← A[j] Key ← 2

j = 2 i = j-1 = 2-1 = 1

1 > 0 and 5 > 2 true

A[i+1] = A[i]

A[2] = 5 5, 5, 4, 6, 1, 3

i = i-1 = 1-1 = 0 while i > 0 false

A[i+1] ← Key

A[1] = 2

So, 2, 5, 4, 6, 1, 3

j = 3 i = j-1 = 3-1 = 2

2 > 0 and 5 > 4 true

A[i+1] ← A[i]

A[3] ← A[2] A[3] = 5 2, 5, 5, 6, 1, 3

i = i-1 = 2-1 = 1 i > 0 && A[i] > Key

∴ A[i+1] ← key i 2 > 4 false

A[2] ← 4

So, 2, 4, 5, 6, 1, 3

j = 4 i = j-1 = 3

3 > 0 and 5 > 6 false

2, 4, 5, 6, 1, 3

i j

~~i < j~~ > 1

j = 5 i = j-1 = 4

4 > 0 and 6 > 1 true

A[i+1] ← A[i]

A[5] ← A[4]

A[5] ← 6 2, 4, 5, 6, 6, 3

i = i-1 = 4-1 = 3

Date : / /

Analysing Algorithm :-

→ It means predicting the resources that the algorithm requires such as memory communication bandwidth, CPU time etc.

→ The time taken by the insertion sort procedure depends on the input (Sorting more nos., algorithm takes more time).

→ Insertion sort can take different amount of time to sort two input sequences of the same size depending on how much it is sorted.

→ Therefore the running time of a program is defined as a function of the size of its input.

→ The running time of an algorithm on a particular input is the number of operations it performs.

→ A constant amount of time is required to execute each line of code. One line may take different amount of time than another line.

→ We assume that each execution of the i^{th} line takes c_i time. (Constant time).

→ The analysis of insertion sort procedure include the analysis with the time that is cost of each statement and the no. of times each statement is executed.

→ For each $j = 2$ to n where n = length of the array, let t_j be the number of time the while loop is executed, for that value of j.

→ When a for or while loop exists, the test is executed one time more than the loop body.

→ The comments are not included while analysing.

1.	for ($j=2$ to length[A])	C_1	n
2.	do Key $\leftarrow$ A[j]	C_2	$n-1$
3	$i \leftarrow j-1$	C_3	$n-1$
4.	while ($i>0$ and $A[j] > Key$)	C_4	$\sum_{j=2}^{n} t_j$
5.	do $A[i+1] \leftarrow A[i]$	C_5	$\sum_{j=2}^{n} t_j - 1$
6.	$i \leftarrow i-1$	C_6	$\sum_{j=2}^{n} t_j - 1$
7	$A[i+1] \leftarrow Key$	C_7	$n-1$

Total running time is the sum of running times for each statement and is represented as $T(n)$.

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4\left(\sum_{j=2}^{n} t_j\right) + C_5\left(\sum_{j=2}^{n} t_j - 1\right) + C_6\left(\sum_{j=2}^{n} t_j - 1\right) + C_7(n-1)$$

Best Case:

The best case occurs when the array is already sorted. In this case for each $j=2$ to n the $A[i] \leq Key$. is found. In line 4, when i has the initial value of $j-1$ in line no. 3.

$\therefore t_j = 1$ for $j = 2$ to n and the best case running time will be

$$T(n) = C_1(n) + C_2(n-1) + C_3(n-1) + C_7(n-1)$$
$$= C_1(n) + C_2(n) - C_2 + C_3(n) - C_3 + C_7(n) - C_7$$
$$= (C_1 + C_2 + C_3 + C_4)n - (C_2 - C_3 - C_7)$$
$$= an - b$$

The above function can be written as $an-b$, where a and b are constants, the running time is a linear function in terms of n.

Page No.
Date : / /

Worst Case:

The worst case occurs when the array is in reverse order or decreasing order.
In this case each element A[j] is compared with each element in the entire sorted sub array (A[1 ... j-1]) and $t_j = j$ (for $j = 2$ to n)
Therefore,

$\sum_{j=2}^{n} t_j$ can be written as $\sum_{j=2}^{n} j$

$$= 2 + 3 + 4 \cdots n + 1 - 1$$

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

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

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

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

$$T(n) = C_1(n) + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n^2+n-2}{2}\right) + C_5\left(\frac{n^2-n}{2}\right)$$

$$+ C_6\left(\frac{n^2-n}{2}\right) + C_7(n-1)$$

$$= (C_1+C_2+C_3+C_4-C_5-C_6+C_7)n + \left(\frac{C_4+C_5+C_6}{2}\right)n^2 - (C_2+C_3+C_6+C_7)$$

$$\boxed{T(n) = an^2 + bn - c} \quad \approx n^2$$

Average Case:

For insertion sort average case is as bad as the worst case. In this case we ~~chose~~ choose n number and

Page No.
Date : / /

apply insertion sort. First we have to determine, where in the subarray A[1 ... to j-1] to here to insert the element A[j].

On average half the elements, in A[i to j-1] are less than A[j] and half are greater.

Therefore, half of the array is checked and thus,

$t_j = j/2$.

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

$$= \frac{2}{2} + \frac{3}{2} + \frac{4}{2} + \frac{5}{2} \cdots + \frac{n}{2}$$

$$= \frac{1}{2}[2 + 3 + 4 \cdots n + n - 1]$$

$$= \frac{1}{2}\left[\frac{n(n+1)}{2} - 1\right]$$

$$= \frac{n(n+1)}{4} - \frac{1}{2}$$

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

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

$$= c_6 \sum_{j=2}^{n} t_j - 1 + c_7(n-1)$$

$$T(n) = n(c_1 + c_2 + c_3 + c_4 + c_4 - c_5 - c_6) - (c_2 + c_3 + c_7)$$

$$+ n^2(c_4 + c_5 + c_6)$$

$$\therefore T(n) \approx n^2$$

Page No.
Date : / /

MERGE SORT

The recursive algorithms follow divide and conquer approach. In divide and conquer approach a big problem is divided into, several sub problems and then we solve the sub ~~array~~ problem recursively and then combine these solutions, to create a solution to the original problem.

The merge sort algorithm follows divide and conquer approach and involves 3 steps:

divide - divide n elements sequence, to be sorted into two sub sequences of n/2 elements each.

Conquer - solve two sub sequences recursively using merge sort.

combine - merge the 2 sorted sub sequences to produce the sorted answer.

The major task in merge sort is the merging of two sorted sequences.

The procedure that performs this task is **Merge (A, p, q, r)** where A is the array

p, q, r are the indices such that $p \leq q \leq r$

Two sorted sub arrays are $A[p \cdots q]$ and $A[q+1 \cdots r]$

Page No.
Date : / /

Merge (A, p, q, r)

```
1.  n1 ← q - p + 1
2.  n2 ← r - q
3.  Create arrays L[1 ... n1+1] and R[1 ... n2+1]
4.  for i ← 1 to n1
5.  do L[i] ← A[p - i - 1]
6.  for j ← 1 to n2
7.  do R[j] ← A[q + j]
8.    L[n1+1] ← ∞
9.    R[n2+1] ← ∞
10.     i ← 1
11.     j ← 1
12.   for K ← p to r
13.   do if L[i] ≤ R[j]
14.   then A[K] ← L[i]
15.        i ← i + 1
16.   else A[K] ← R[j]
17.     j ← j + 1
```

→ line 1 computes the length of sub array A[p ... q] and
→ line 2 computes length of sub array A[q+1 ... r).
→ In line 3 left and right arrays are created with length $n_1 + 1$ and $n_2 + 1$ respectively.
→ In line 4 and 5, the array elements are copied from A[p ... q] into L[1 to ... n_1]
→ Similarly in line 6 and 7, A[q+1 ... r] are copied to R[1 ... n_2]
→ line 8 and 9 are used to put the sentinal values at the end of the left and right a sub array.

→ In line 10 and 11 indices are initialised to 1.

→ In line 12 the for loop is used to add the values from the left and right array into the original array.

→ The no. of elements ~~are~~ are represented as $n = r - p + 1$.

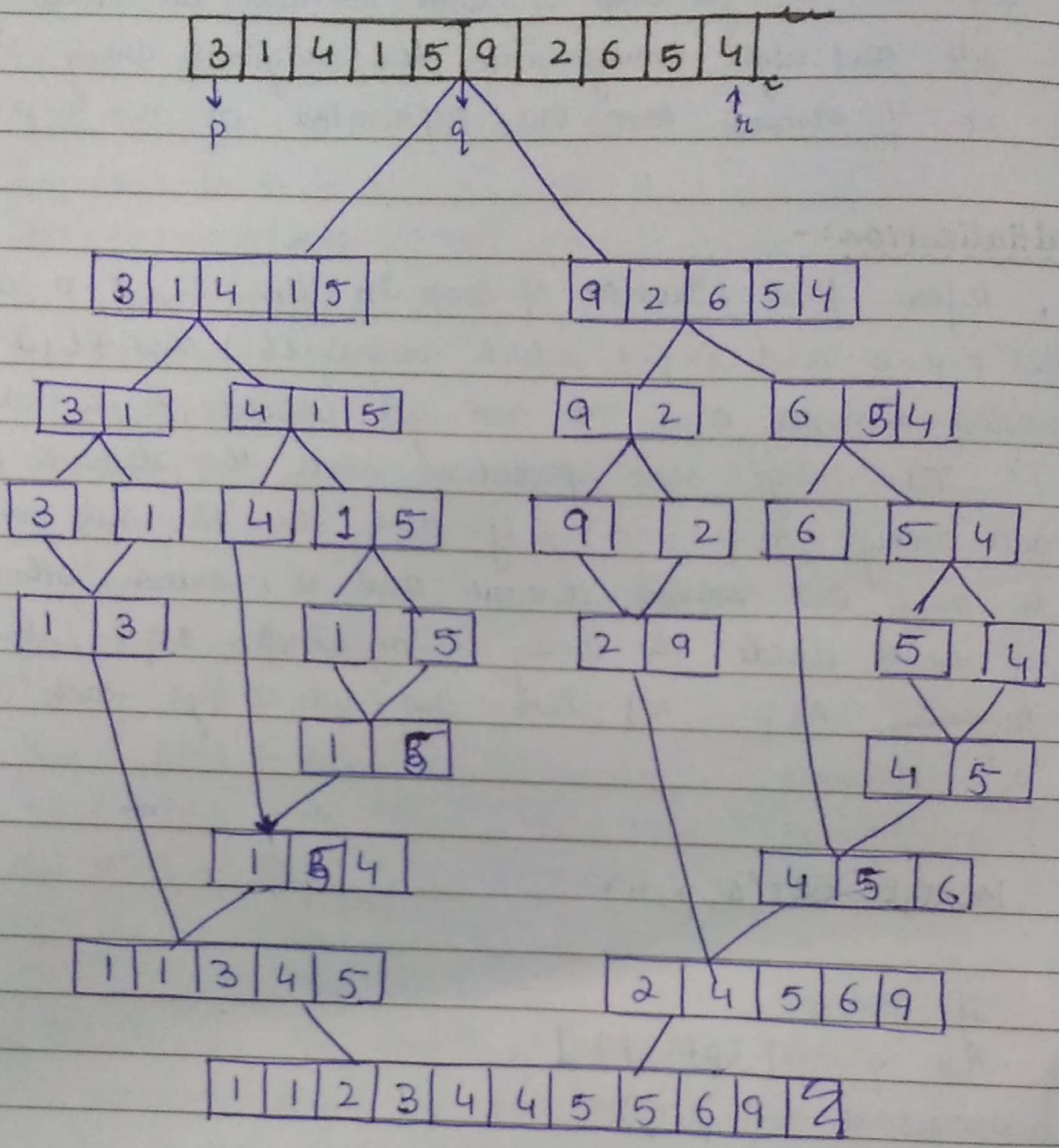
initialization:-

- Before first iteration of loop in line 12, $k = p$ such that $k - p = 0$ and $i = j = 1$ which means $L[i]$ and $R[j]$ contains smallest elements and are not yet copied to the array A.
- The merge sort procedure sorts the elements in the sub array $A[p \ldots r]$, if $p \geq r$ then the sub array has only one ~~marble~~ element and is sorted, otherwise we ~~write~~ divide the array using index 'q', that partitions $A[p \ldots r]$ into two sub arrays each containing $n/2$ elements.

MERGE_SORT(A, p, r)

1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. mergesort (A, p, q)
4. mergesort (A, p+1, r)
5. Merge~~sort~~ (A, p, q, r)

3, 1, 4, 1, 5, 9, 2, 6, 5, 4



analysis of merge sort

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

A recurrence of divide and conquer algorithm is based on 3 steps:

Page No.
Date : / /

- let, $T(n)$ be the running time of a problem of size n.
- If the problem size is small $n \leq c$, where c is some constant the total running time can be written as $\Theta(1)$ i.e. constant time.
- Suppose division of the problem yields 'a' sub problems of size $1/b$ each in merge sort, a and b both are 2.
- $D(n)$ be the time taken to divide the problem into subproblem.
- $C(n)$ be the time taken to combine the sol^n of the sub-problem.

So the recurrence relation can be defined as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ a\,T\left(\frac{n}{b}\right) + D(n) + C(n) & \text{other} \end{cases}$$

when no. of elements are greater than 1 ~~than~~ then the running time is divided into 3 steps.

* **divide**

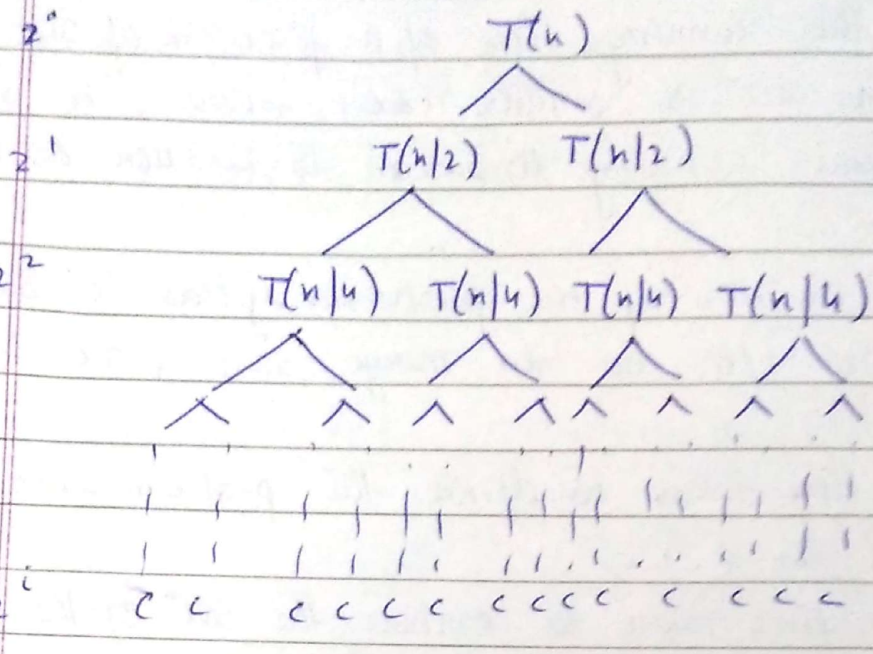
It helps to compute middle of sub array and takes constant time ~~is~~ i.e. $D(n) = 1$

* **Conquer**

In this step 2 sub problems are solved each of ~~#~~ size $n/2$, which contributes to $2T(n/2)$.

* **Combine**

The sol^n of sub problems are combined together, which means, n elements are combined,
So combine time $C(n)$ will be $\Theta(n)$.



The total cost is computed by adding the cost involved at all the levels is $cn + cn + cn + cn \cdots + cn \log_2$

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

The level i has 2^i nodes each contributing to the cost of $c \cdot \frac{n}{2^i}$ so the total cost at each level be $2^i \cdot \frac{cn}{2^i}$

$= cn$

let the total no of levels be m $\therefore$ cost of each node at m is $c \cdot \frac{n}{2^m}$

$$\frac{cn}{2^m} = c \Rightarrow cn = c 2^m$$

$$n = 2^m$$

$$\log n = m \log 2$$

$$\log n = m$$

no. of levels $= \log n + 1$

$\therefore$ total cost $= cn[\log n + 1]$

$= cn \log n + cn$

$= n \log n + n$

$\approx n \log n$

Page No.
Date : / /

3. # HEAP SORT

- Heap sort algorithm uses a data structure called a heap which is a binary tree with some special properties.

- An array A[] represents a heap, having 2 attributes :
  - → length[A] – no of elements in the array.
  - → heap-size[A] – the no. of elements in the heap stored, within the array A[].
  - → heap-size[A] ≤ length[A]
- The root of the tree is A[1], the indicies of parent ie. parent(i).
  for left child = left(i)
  right child = right(i).

* parent(i) — return ⌊i/2⌋
left(i) — return 2i
right(i) — return 2i+1

eg:

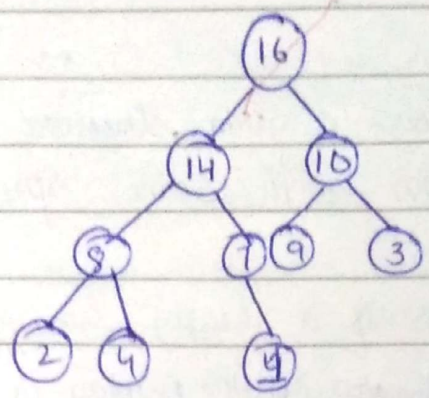
16	14	10	8	7	9	3	2	4	1

root = A[1] = 16

To check the parent of any index will calculate i/2 factor.

index	parent(i)	left(i)	right(i)
1	⌊1/2⌋ = 0	2×1 = 2	2×1+1 = 3
2	⌊2/2⌋ = 1	2×2 = 4	2×2+1 = 5
3	⌊3/2⌋ = 1	2×3 = 6	2×3+1 = 7
4	⌊4/2⌋ = 2	2×4 = 8	2×4+1 = 9
5	⌊5/2⌋ = 2	2×5 = 10	2×5+1 = 11
6	⌊6/2⌋ = 3	2×6 = 12	2×6+1 = 13
7	⌊7/2⌋ = 3	2×7 = 14	2×7+1 = 15
8		no child	no child
9			
10			

Date : / /



There are two type of binary heaps, max heap and m
The values in the node satisfy ~~all~~ the heap property

The max heap property is that for every node i, oth
than the root.

$$A[Parent(i)] \geq A[i]$$

The min heap property

$$A[Parent(i)] \leq A[i]$$

In a tree representation of a heap, the height of a no
in a heap is defined as the number of edges on the
longest downward path from node to a leaf.

The height of the entire heap is defined by the height
of its root.
There are 4 procedures that are used to sort :

- Max-heapify - It runs in O(log n) time and maintains
  max-heap property.
- build-max-heap - It runs in linear time, produces
  max-heap from unsorted input array.
- heap-sort - It runs in O(n log n) time sorts an
  array in place.

- Max-heap-insert - heap extract-max, heap-increase-key, maximum. They all run in O(log n) time.

maintaining the heap property -

Max-heapify is the procedure, that manipulates the max-heap. Inputs are array A[] and index i.

When Max-heapify is called, it is assumed that left of i and right of i are max-heaps. But roots may violate the max heap property.

Max-heapify (A, i)

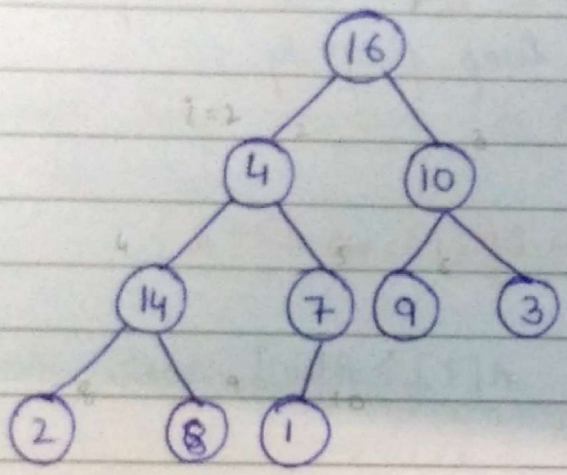
1. l ← left (i) l ← left(i) ⇒ l ← 2 i=2, l=4, r=5
2. r ← right (i)
3. if l ≤ heapsize [A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heapsize [A] and A[r] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. Max-heapify [A, largest];

At each step the largest of elements
A[i]
A[left(i)]
A[right(i)]
is determined and its index is stored in largest variable. If A[i] is largest then subtree at node i is a max heap and the procedure terminates.

else one of the children has the largest element and A[i] is swapped with A[largest].

At the end Max-Heapify procedure is recursively called at the largest index.

Max-Heapify on a sub tree of size 'n' of node i, takes $O(i)$ times. Time taken to run max-heapify on sub tree.

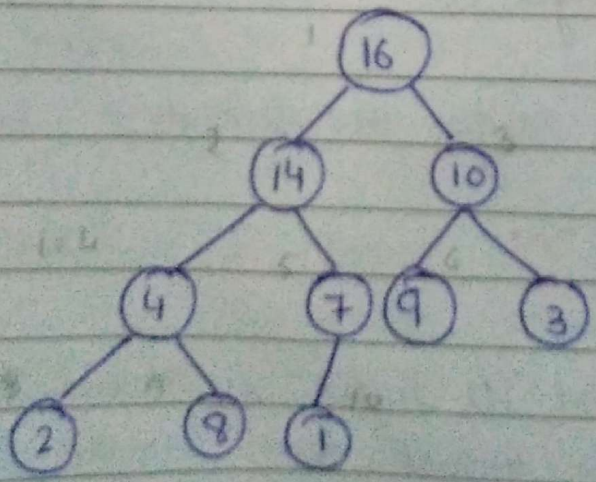


$A[i] < A[left(i)]$ $4 < 14$

largest = 14

~~left~~ $A[i] > A[right(i)]$ $14 > 7$

largest $4 \leftrightarrow 14$



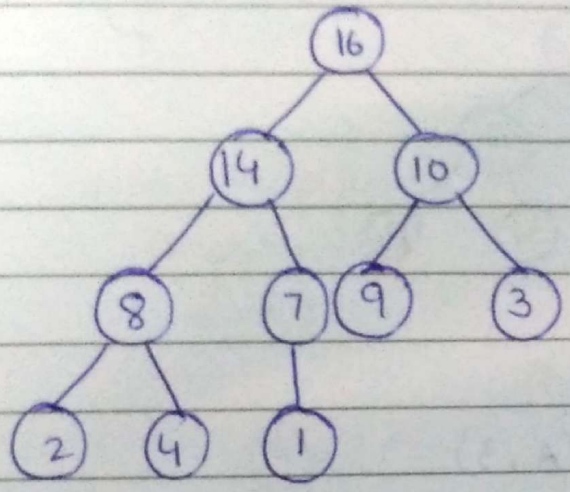
4 > 2

largest = 4

4 > 8

largest = 9

Swap 4 ↔ 8



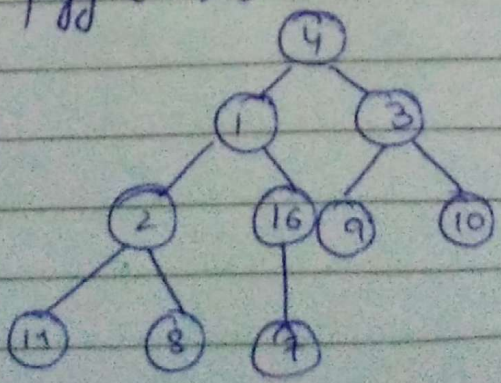
Build-max-heap.

Max-heapify in a bottom up manner is used to convert an array into a max-heap because, the elements in the sub array are all leaves of the tree.

Build-max-heap goes through the remaining nodes and runs heapify on each of them.

Build-max-heap(A)

1. heapSize[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ down to 1
3. do max-heapify(A, i)



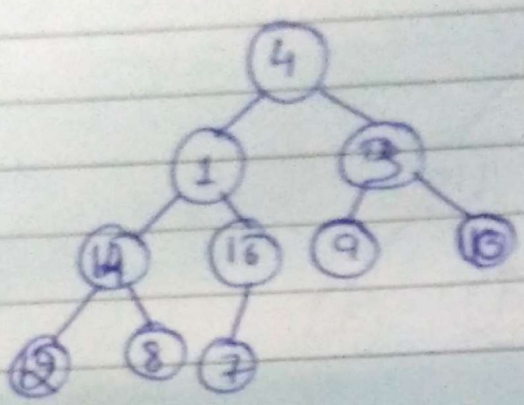
heapSize = 10

for i = 5 to 1

i = 5 Max. heapify (A, 5)

no change

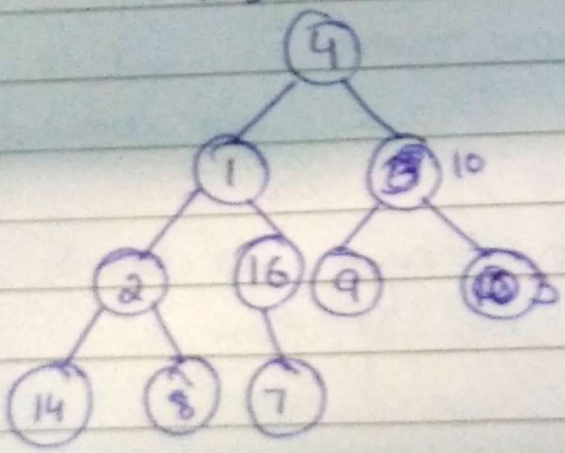
2 i = 4 max heapify (A, 4)



3. i = 3 max_heapify (A, 3)

largest = 7

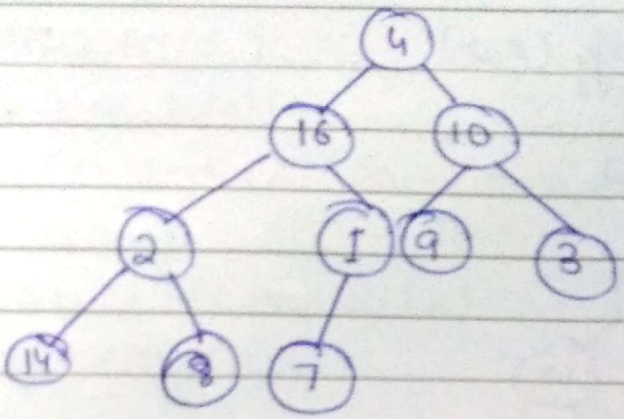
A[3] ↔ A[7]



4. i = 2 max_heapify (A, 2)

largest = 5

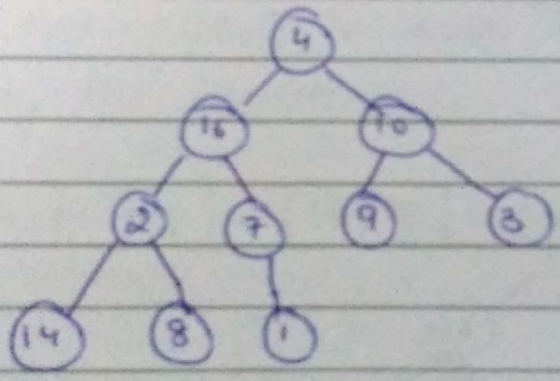
A[5] ↔ A[2]



5 $i = 2$ max-heapify (A, 5)

largest = 16

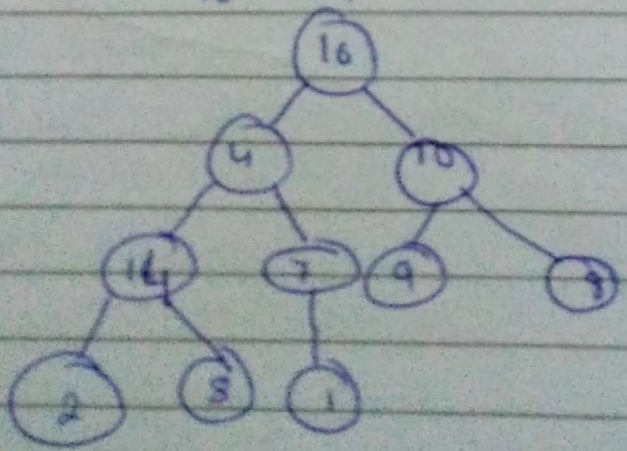
$i \leftrightarrow 1$



6. $i = 1$ max-heapify (A, 1)

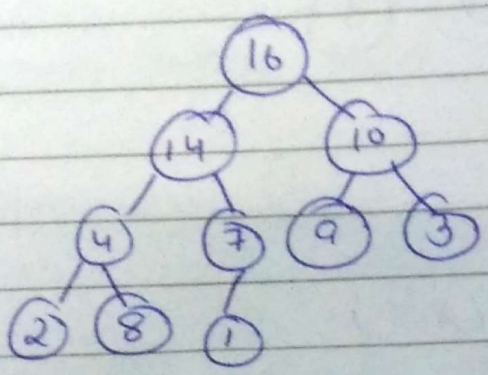
largest = 2

$16 \leftrightarrow 4$

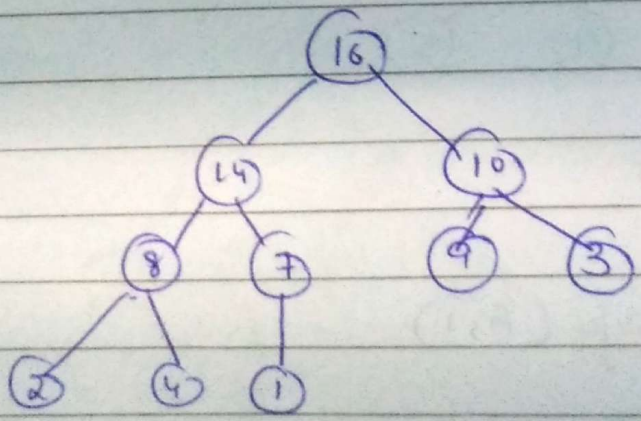


Page No.
Date : / /

7. i = 1 max-heapify (A, 2)
largest = 4
14 ↔ 2



8. i = 1 max-heapify (A, 4)
largest = 9
8 ↔ 4



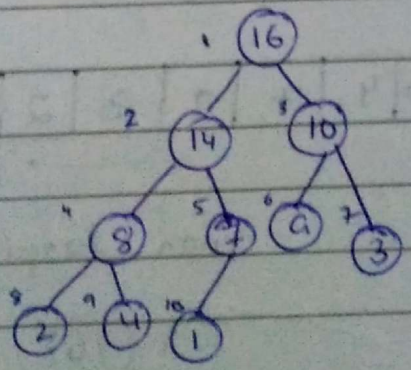
HEAP_SORT(A)

1. BUILD-MAX-HEAP(A).
2. for i ← length[A] down to 2
3. do exchange A[1] ↔ A[i]
4. HeapSize[A] ← HeapSize[A] - 1
5. MAX-HEAPIFY(A, 1).

Heap sort algo starts with BUILD-MAX-HEAP to build a max-heap on input array A[1---n]. The max element of the array is stored at the root i.e A[1] it can be put to its ~~Cor~~ correct position by exchanging it with A[n]

Decrease the heap-size by 1 and then call MAX-HEAPIFY(A,1) to restore the heap property of new binary tree created.

A	4	1	3	2	16	9	10	14	8	7

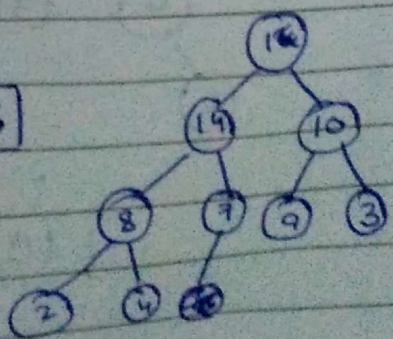


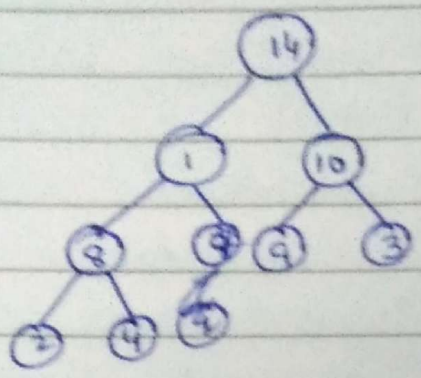
i = 10 A[1] ↔ A[10]

16 ↔ 1

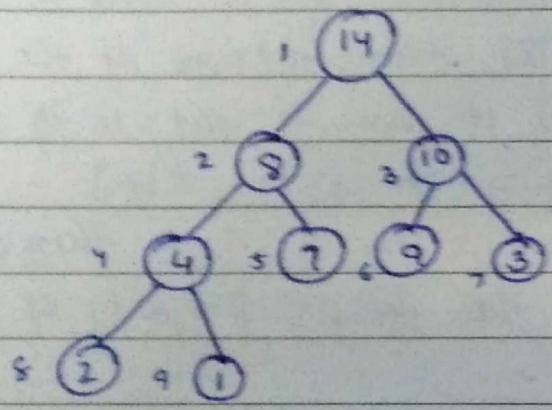
1	14	10	8	7	9	3	2	4	16

heapsize = 9





max heapify:



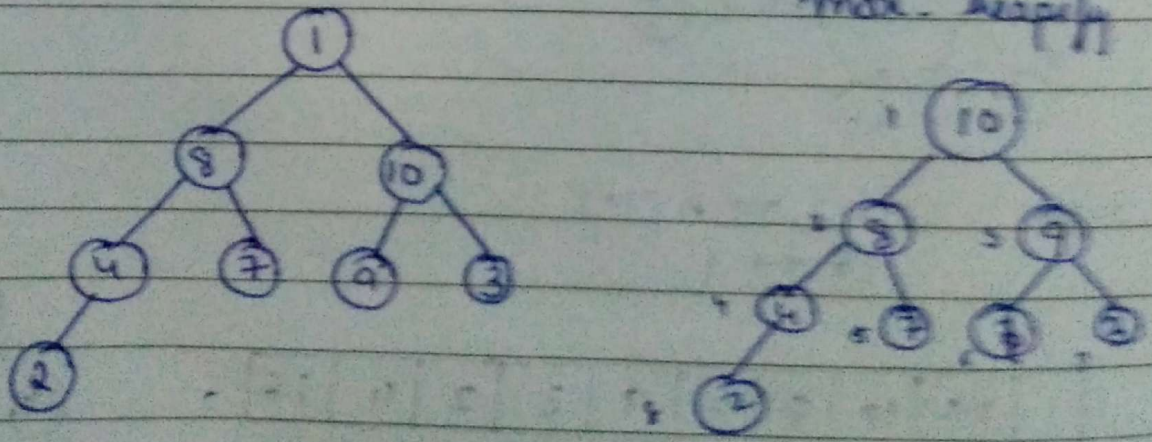
i = 9 $A[1] \leftrightarrow A[9]$

$14 \leftrightarrow 1$ ✓

i	1	8	10	4	7	9	3	2	14	16

heapsize = 8

max. heapify



1	2	3	4	5	6	7	8
10	8	9	4	7	1	3	2

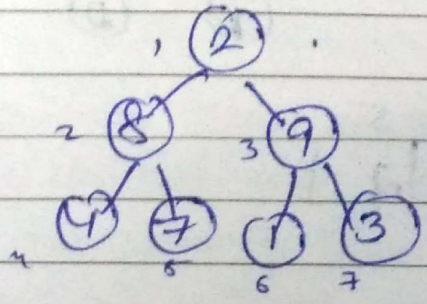
i=8

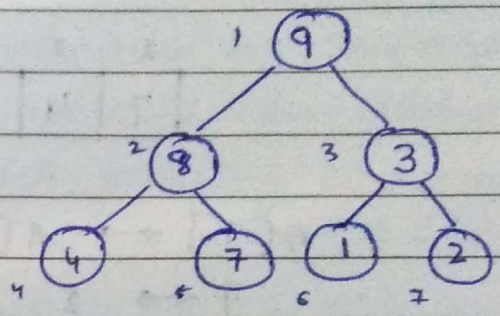
A[1] ↔ A[8]

1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16

heapsize = 7

max-heapify





1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

i=7

~~heapsize = 6~~

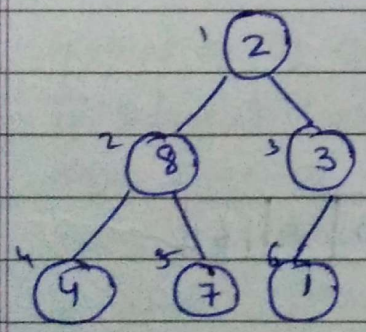
A[1] ↔ A[7]

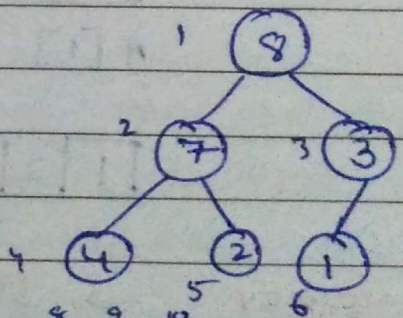
9 ↔ 2

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

heapsize = 6.

max-heapify



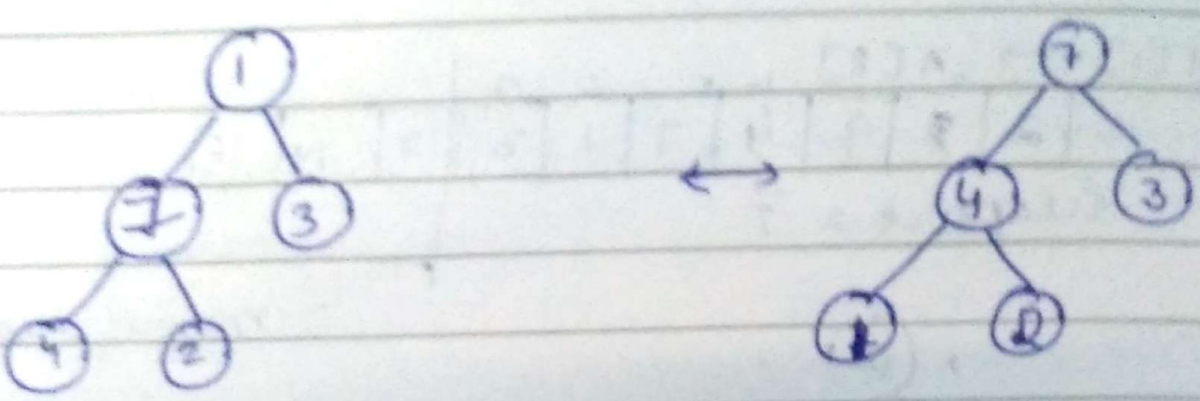


1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

A[1] ↔ A[6]

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

heap size = 5



1	2	3	4	5
7	4	3	1	2

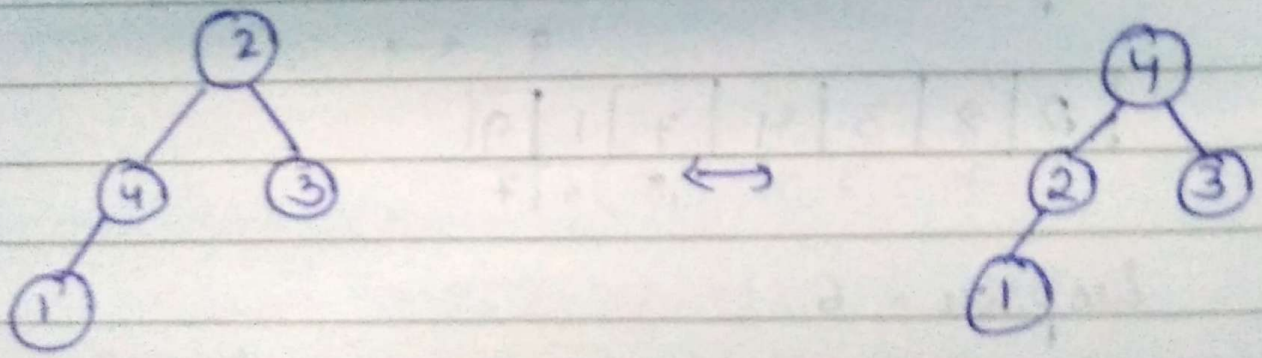
A[1] ↔ A[5]

7 ↔ 2

2	4	3	1	7

heapsize = 4

max_heapify



4	2	3	1

A[1] ↔ A[4]

1	2	3	4	7	8	9	10	M	16

Page No.
Date : / /

PRIORITY QUEUES.

Application of heap as priority queue can be of 2 types

max priority queue
min priority queue.

→ A priority queue is a data structure for maintaining set S ~~with~~ of elements each with an associated value called a key.

→ A max priority queue supports following operations

- Insert(S, x) – Inserts the element x into set S i.e.

$$S \leftarrow S \cup \{x\}$$

- Maximum (S) - It returns the element of x with the largest Key.
- EXTRACT ~~Abstract~~ _MAX(S) - It removes and returns the element of S with the largest Key.
- INCREASE_KEY(S, x, K) - It increases the value of element x's key to the new value K. which is assumed to be atleast as large as x's current Key value.

Application of max priority queue.

- To schedule the jobs.
- Keeps track of the jobs to be performed and their relative priorities. when a job is finished, the highest priority job is selected from those pending using extract-max()

HEAP. MAXIMUM(A).
return A[1]

This procedure is implemented constant time O(1)

Page No.
Date : / /

HEAP_EXTRACT_MAX(A)

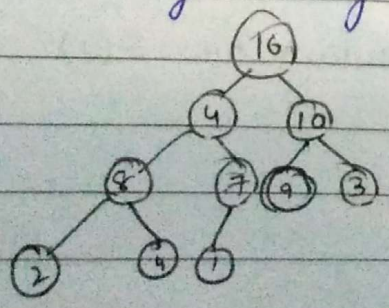
1. if heapSize[A] < 1 // no elements in the heap
2. then error "heap underflow"
3. max ← A[1]
4. A[1] ← A[heapsize[A]] // last element of array is copied to A[1]
5. heapsize[A] ← heapsize[A] - 1
6. MAX-HEAPIFY(A, 1)
7. Return max.

Running time - log n which is the max time taken by max-heapify.

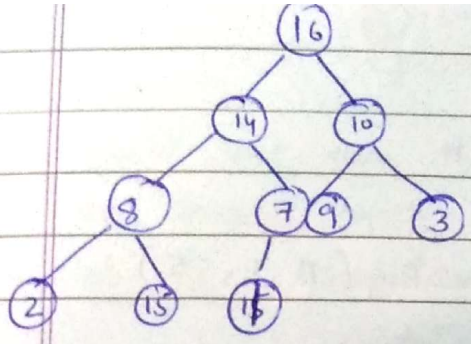
HEAP_INCREASE_KEY(A, i, key).

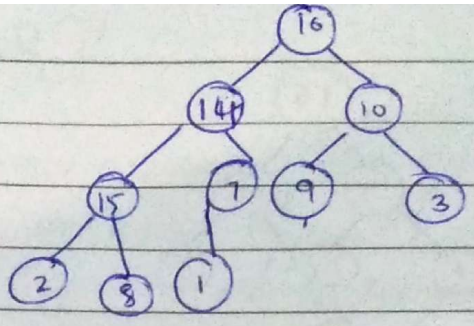
1. if key < A[i]
2. then error "new key is smaller"
3. A[i] ← key
4. while i > 1 and A[parent(i)] < A[i]
5. do exchange A[i] ↔ A[parent(i)]
6. i ← parent(i)

The priority key element whose key is to be increased is identified by index 'i'. The procedure first update the key of the element A[i] to its k new value. This procedure traverses a path from the node i to the root and repeatedly compares an element to its parent exchanging their keys and continuing if the element's key is larger.



The value of index is to be increased to 15 i.e. key = 15 and i = 9.





A[parent(9)]

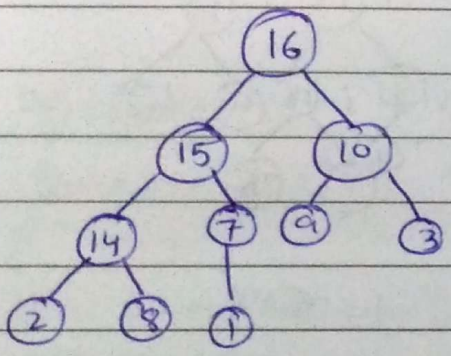
A[4] ↔ A[9]

8 < 15 True.

A[9] ↔ A[4]

i ← parent(i)

i ← 4



Running time - log n.

A[parent(4)]

A[4] < A[2]

14 < 15 true

exchange. A[4] ↔ A[2]

15 ↔ 15

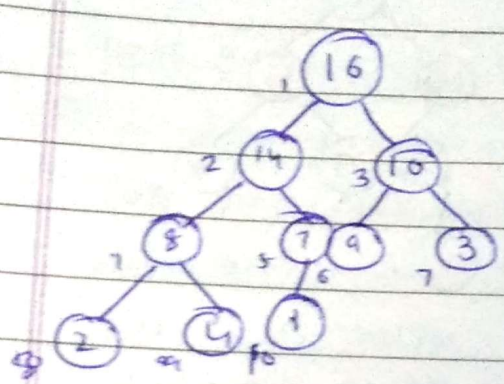
MAX_HEAP_INSERT (A, Key).

heapSize[A] ← heapsize[A] + 1

A[heapsize[A]] ← -∞

Heap_increase_Key (A, Heap size[A], Key)

The above procedure does insertion, It takes as input the key of the new element which is to be inserted into max heap. To do this it first adds (-∞) in a new leaf and then calls heap-increase key to set the key of new node to its correct value.

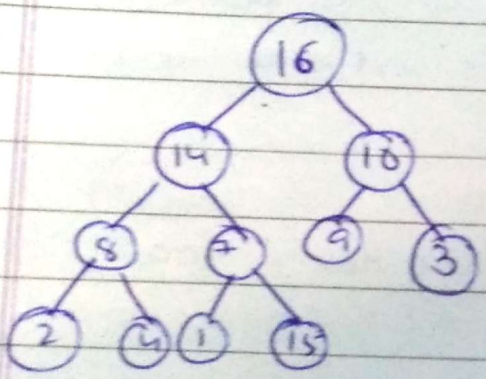


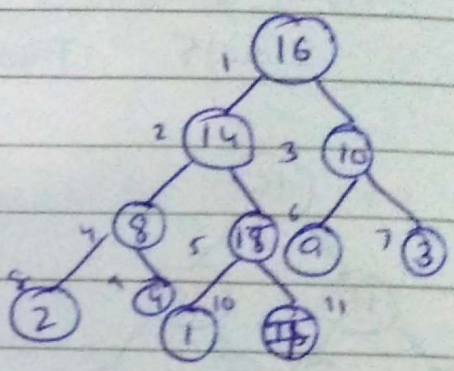
key = 15

heapsize = 11

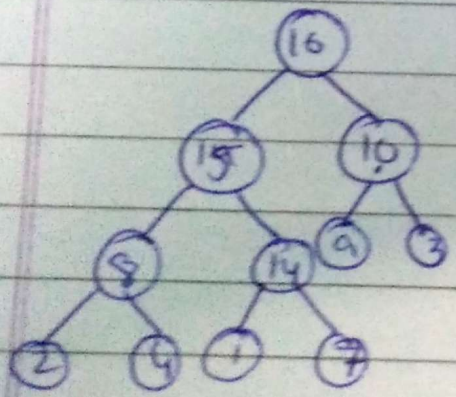
A[11] ← -∞

Heap_increase_key (A, 11, 15)





i = 5



i = 2

18/1/17

Page No.
Date : / /

QUICK SORT

Quick sort uses divide and conquer approach. Three steps of divide and conquer process for sorting an array $A[p \cdots r]$ are

- **divide** → partition the array $A[p \cdots r]$ into 2 subarrays $A[p \cdots q-1]$ & $A[q+1 \cdots r]$ such that each element of $A[p \cdots q-1] \leq A[q]$ and $A[q] \leq A[q+1 \cdots r]$
- **Conquer** → Sort the two subarrays by recursive call to quicksort
- **Combine** → Subarrays are sorted inplace and therefore no work is needed to combine them.

PROCEDURE

```
QuickSort (A, p, r)
1. if   p < r
2. then q ← PARTITION (A, P, r)
3.      QuickSort (A, P, q-1)
4.      QuickSort (A, q+1, r)

PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p-1
3.  for j ← p to r-1
4.  do if A[j] ≤ x
5.     then i ← i+1
6.  exchange A[i] ↔ A[j]
7.  exchange A[i+1] ↔ A[r]
8.  return i+1.
```

Date : / /

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

Initially quick Sort procedure is called with the values as Quicksort (A, 1, 8). Then p is check against r ie $1 < 8$ which means there are more than one element. The array is then partitioned using partition procedure with a call to PARTITION (A, 1, 8).
This procedure selects an element x as pivot element around which to partition the subarray $A[p, \ldots r]$.

QuickSort (A, 1, 8)

$1 < 8$

PARTITION (A, 1, 8)

$x \leftarrow A[8]$. $x = 4$

$i = p - 1 = 0$

$j = 1$

$A[1] \leq 4$

$2 \leq 4$ $\quad A[1] \leftrightarrow A[1]$

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

$j = 2$ $\quad A[2] \leq 4$

$8 \leq 4$ false

$j = 3$ $\quad A[3] \leq 4$

$7 \leq 4$ false

$j = 4$ $\quad A[4] \leq 4$

$i = 2$

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

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

$i \rightarrow$ $\quad$ $\leftarrow j$

$j = 5$ $\quad A[5] \leq 4$

$3 \leq 4$ true

$i = 3$

$A[3] \leftrightarrow A[5]$

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

Page No.
Date : / /

j=6 , i=3. A[6] ≤ 4

5 ≤ 4 false

j=7 , A[7] ≤ 4

6 ≤ 4 false

j=8 ~~A[8]~~

i = 1

8-1 = 7

for (j=p to r-1

A[i+1] ↔ A[r]

A[4] ↔ A[8]

8 ↔ 4

q = 4

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

(A, p, q-1) → A(A, 1, 3)

x ← A[3]

x ← 3

2	1	3
1	2	3

i = 0

j = 1 A[1] ≤ 3 => 2 ≤ 3 true

i = 1

A[1] ↔ A[1]

2	1	3
1	2	3

j = 2 A[2] ≤ 3 => 1 ≤ 3 true

i = ~~2~~ 2

A[2] ↔ A[2]

2	1	3
1	2	3
	i	

~~A~~ (A, p, q-1) → A(1, 2)

1 < 2 true

x ← A[2]

x ← 1

i = 0

2	1
1	2

j = 1 A[1] ≤ 1 => 2 ≤ 1 false

~~i = 1~~

~~A[1] ↔ A[1]~~

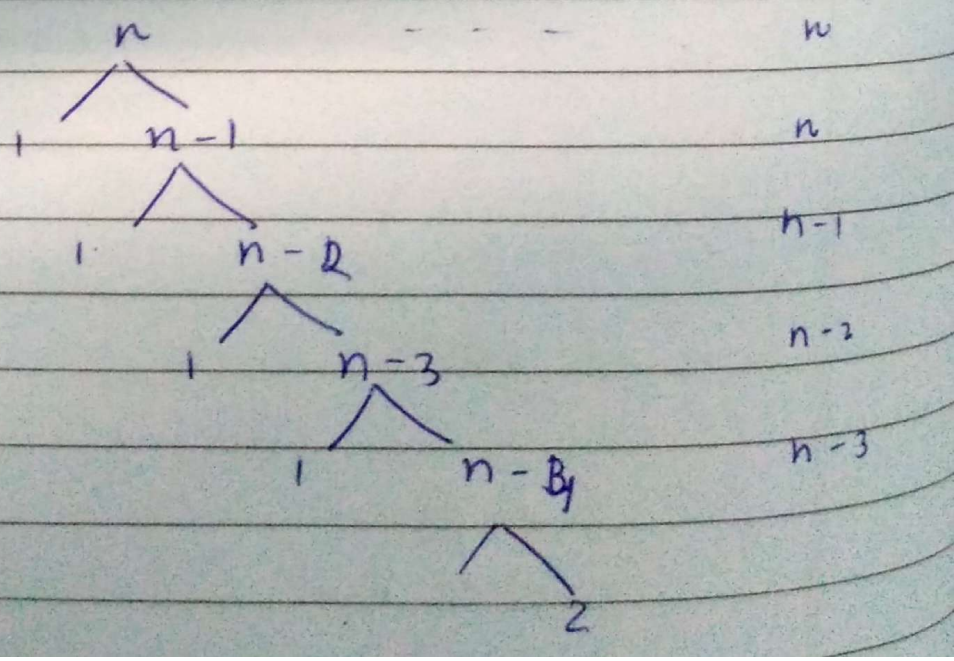
performance analysis

The running time of quick sort depends on whether the partitioning is balanced or unbalanced. If the partitioning is balanced the algorithm asymptotically as fast as merge sort.
If the partitioning is balanced the algorithm runs as slow as insertion sort.

worst case occurs when partitioning procedure produces one sub problems with $n-1$ elements and another with only one with 0 element.
let this unbalanced partitioning arises at each step of the algorithm.
Partitioning costs $\theta(n)$ time
The recursive call on a array of size = 0 returns
The recurrence equation for total running time will be

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

The recursion tree for worst case execution of quick sort



Total = $n + n + (n-1) + (n-2) + (n-3) + \cdots$

$+ 3 + 2 + 1 - 1$

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

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

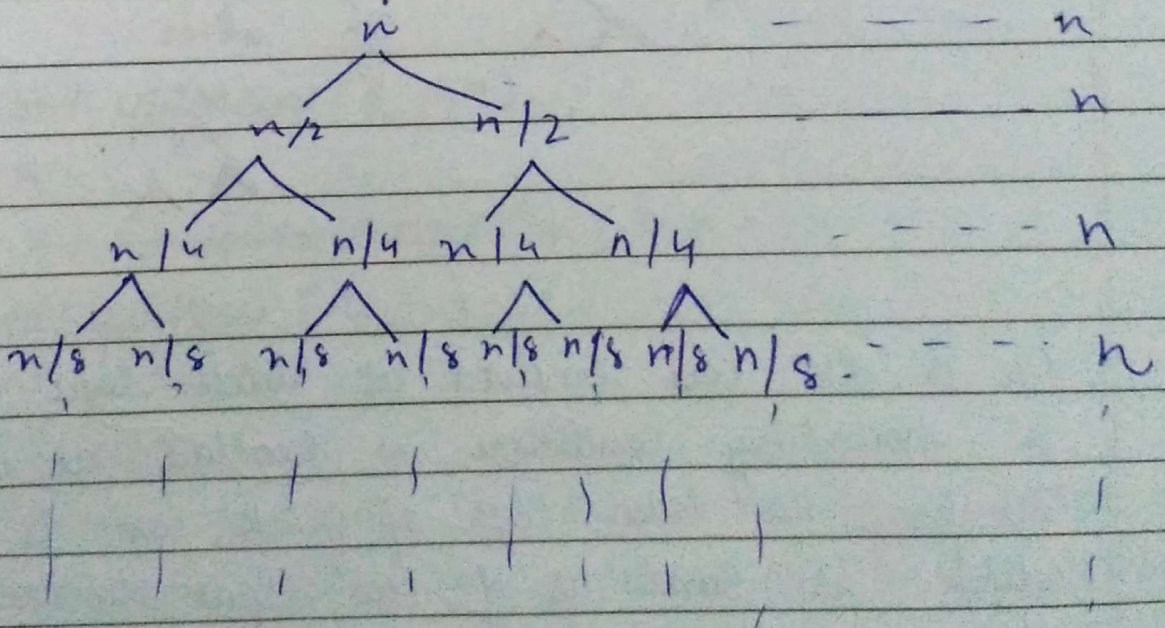
$= O(n^2)$

Best case partitioning occurs when partitioning ~~procedure~~ procedures 2 regions ie. each of size no more than $n/2$.

The recurrence equation

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The recursion tree for the best case



Total = $n + n + n \cdots \log n$

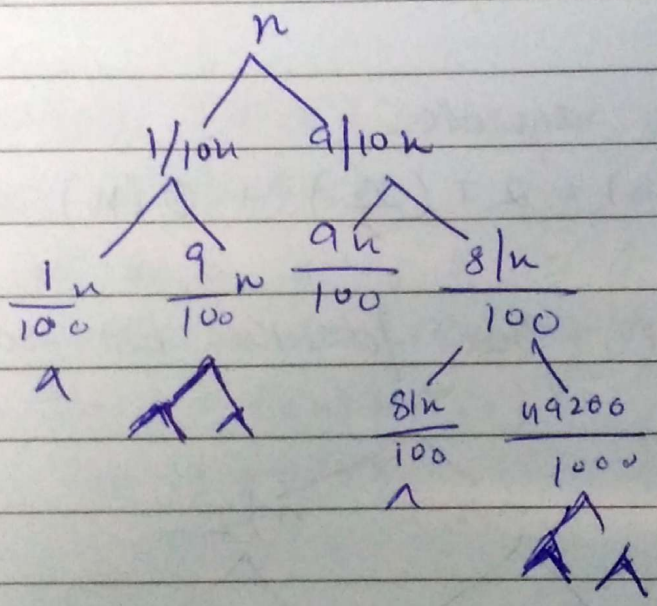
$= n \log n$

The average case running time of quick sort is close to the best case

Let us assume that partitioning algorithm always produ proportional split, which seems unbalanced at first sigh

The recurrence equation can be defined as

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + Cn$$



C_n is the cost involved at each level of the tree, until a boundary condition is reached at depth $\log n$. Therefore, the total cost of quick cost is $O(n \log n)$, which is same as if we split the array from the middle.

Page No.
Date : / /

Randomised version of Quick Sort.

Randomisation is added to an algorithm in order to obtain good average case performance, Instead of using A[r] as the pivot we will use a randomly choosen element from the sub array. Then we exchange the element A[r] with an element choosen at random. This modification ensures that, the pivot element is equally likely to be any of the element in the sub array.

Randomized-partition (A, p, r).

1. i ← random (p, r)
2. exchange A[r] ↔ A[i]
3. return PARTITION (A, p, r).

Randomized-quicksort (A, p, r).

1. if p < r
2. then q ← randomized-partition (A, p, r)
3. randomized-quicksort (A, p, q-1)
4. randomized-quicksort (A, p+1, r)

We know that the worst case split at every level of recursion in quick sort produces $O(n^2)$ running time. Let T(n) be the worst case running time for procedure for quicksort on an input of size n.

∴ Recurrance for worst case

$$T(n) = \max (T(q) + T(n-q-1) + O(n))$$

because partition procedure produces 2 sub problems with total size of (n-1), q ranges from (0 to n-1)

Page No.
Date : / /

We make a guess that,

$$T(n) \leq cn^2$$

for some c, now substitute the guess to into recurrance

$$T(n) = \max_{0 \leq q \leq (n-1)} (q^2 + (n-q-1)^2) + \Theta(n)$$

The maximum value of $q^2 + (n-q-1)^2$ occurs when q is either '0' or $(n-1)$.

This means that

$$\max (q^2 + (q-n-1)^2) \leq (n-1)^2$$

$$= n^2 - 2n + 1$$

Thus, the running time is dependent on the constant, how much it dominates $\Theta(n)$ which makes the running time little less than $\Theta(n^2)$.

Page No.
Date : / /

SORTING IN LINEAR TIME

Comparison sort :-

In this the sorted order is determined based on the comparison b/w the input elements. Our aim is to prove that any comparison sort must sort the numbers in less than $(n \log n)$ time.

Sorting algorithm like counting sort, radix sort and bucket sort runs in linear time. Therefore, these algo. use operation other than comparisons to determine the sorted order and running time will be in linear terms.

In comparison sort we use only comparison b/w elements to gain order information about an input seq. Given two elements a_i and a_j we perform following tests.

$a_i \leq a_j$ $\quad a_i > a_j$ $\quad a_i < a_j$ $\quad a_i \geq a_j$ $\quad a_i = a_j$

The decision ~~so~~ tree model.

Comparison sort can be reviewed in terms of decision tree which is a full binary tree, that represents, the comparison b/w the elements that are performed by a particular sorting algorithm.

In a decision tree, each internal node is represented by i, j for some i and j in the range $1 \leq i$ and $j \leq n$ each leaf is represented by a permutation for eg :- $S = \{a, b, c\}$

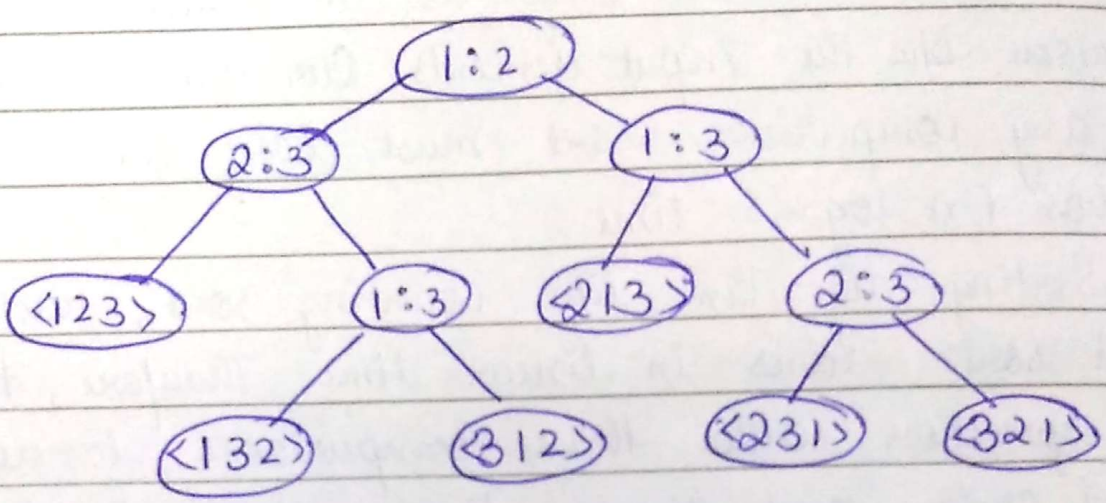
31. are 6 $\{abc, acb, bac, bca, cab, cba\}$

The execution of sorting algorithm corresponds to tracing a path from root of a decision tree to the leaf.

At each node a comparison of $a_i \leq a_j$ is performed.

Page No.
Date : / /

The left subtree shows comparisons for $a_i \leq a_j$ and the right subtree shows comparisons for $a_i > a_j$.
At the leaf level the sorting algorithm



COUNTING SORT

Counting sort assumes that each of the n input elements is an integer in the range of 0 to k for some integer k.

In Counting sort we determine for each input element, x the number of elements less than k.

The information is used to place an element directly into the position in output array.

Page No.
Date : / /

Counting Sort (A, B, K)

```
1.  for i ← 0 to k
2.  do C[i] ← 0
3.  for j ← 1 to length[A]
4.    do c[A[j]] ← c[A[j]] + 1
5.  ▷ C[i] contains the no. of elements equal to i
6.  for i ← 1 to k
7.  do C[i] ← c[i] + C[i-1]
8.  ▷ c[i] now contains the no. of elements less than or equal to i
9.    for j ← length[A] down to 1
10.     do B[c[A[j]]] ← A[j]
11.    c[A[j]] ← c[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

i=0 to k=5

C[0]=0 and C[1]=0

	0	1	2	3	4	5
C	0	0	0	0	0	

j = 1 to 8

i=1

C[A[1]] ← C[A[1]] + 1

C[2] ← C[2] + 1

C[2] ← 1

	0	1	2	3	4	5
C	0	0	1	0	0	0

j=2

C[A[2]] ← C[A[2]] + 1

C[5] ← 1

	0	1	2	3	4	5
C	0	0	1	0	0	1

j=3

C[3] ← C[3] + 1

C[3] ← 1

	0	1	2	3	4	5
C	0	0	1	1	0	1

Date : / /

j=4 C[A[4]] ← C[A[4]]+1
C[0] ← C[0]+1
C[0] ← 1

	0	1	2	3	4	5
C	1	0	1	1	0	1

j=5 C[A[5]] ← C[A[5]]+1
C[2] ← 2

	0	1	2	3	4	5
C	1	0	2	1	0	1

j=6 C[3] ← C[3]+1
C[3] ← 2

	0	1	2	3	4	5
C	1	0	2	2	0	1

j=7 C[0] ← C[0]+1
C[0] ← 2

	0	1	2	3	4	5
C	2	0	2	2	0	1

j=8 C[3] ← C[3]+1
C[3] ← 2+1 = 3

	0	1	2	3	4	5
C	2	0	2	3	0	1

The C array shows the occurences of each element in the input array A.

i=1 to 5

i=1 C[i] ← C[i]+C[i-1]
C[1] ← C[1]+C[0]

	0	1	2	3	4	5
C	2	2	2	3	0	1

i=2 C[i] ← C[i]+C[i-1]
C[2] ← C[2]+C[1]
C[2] ← 2+2 = 4

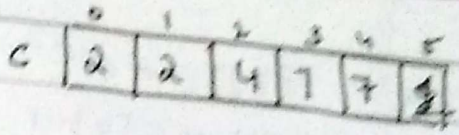
	0	1	2	3	4	5
C	2	2	4	3	0	1

i=3 C[3] ← C[3]+C[2]
C[3] ← 3+3

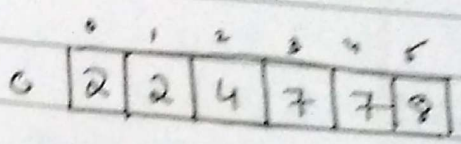
	0	1	2	3	4	5
C	2	2	4	7	0	1

Page No.
Date : / /

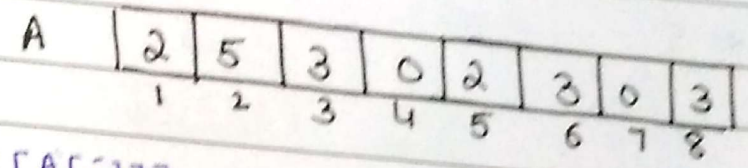
i=4 C[4] ← C[4] + C[3]
C[4] ← 0 + 7



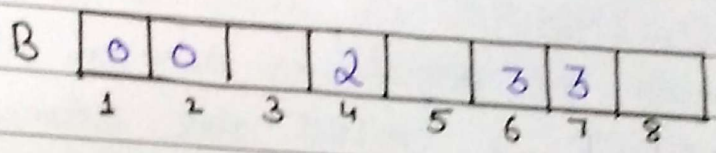
i=5 C[5] ← C[5] + C[4]
C[5] ← 1 + 7



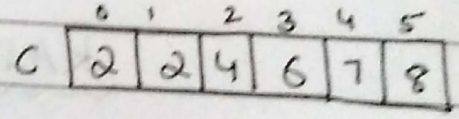
for j = 8 to 1



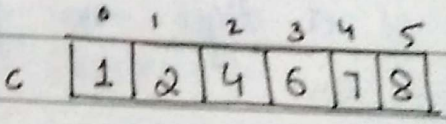
B[C[A[j]]] ← A[j]
C[A[j]] ← C[A[j]] - 1



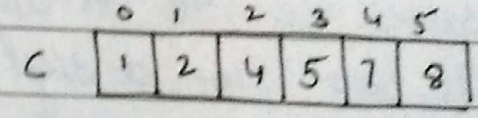
j = 8 B[C[A[8]]] ← A[8]
B[C[3]] ← 3
B[7] ← 3
C[3] ← 6



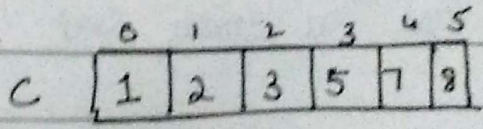
j = 7 B[2] ← 0
C[0] ← 2 - 1



j = 6 B[6] ← 3
C[3] ← 6 - 1



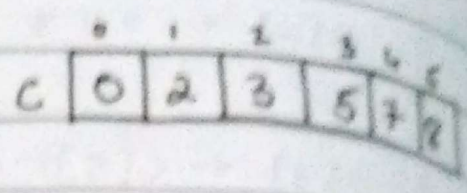
j = 5 B[4] ← 2
C[2] ← C[2] - 1
C[2] ← 3



j=4 B[1] ← 0

C[0] ← C[0] - 1

C[0] ← 1 - 1



j=3 B[6] ← 3

C[3] ← C[3] - 1

C[3] ← 5

j=2

j=1 B[ ] ←

C[3] ← C[ ] - 1

C[ ] ← ~~←[ ]~~

RADIX SORT

Radix Sort works by sorting the least significant digit of d-digit no. first, then second least significant and so on.

This process continues until the no. have been sorted on all d-digits. Thus only d passes are required to sort ~~to~~ d-digit no.

Radix_Sort(A, d)

1. for i ← 1 to d
2. use a stable sort to sort an array A on d-digit

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

BUCKET SORT

We assume that the input ~~lies~~ lies in the interval [0, 1). The idea is to divide the interval [0, 1) in n equal sized sub intervals or buckets and then distribute 'n' input numbers into buckets.
To produce the output, we sort the numbers in each bucket and go through the bucket in order listing the elements in each.

BUCKET SORT

1. n ← length[A]
2. for i ← 1 to n
3. do insert A[i] into list B[⌊n A[i]⌋]
4. for i ← 0 to n-1
5. do sort list B[0 ... n-1] with insertion Sort.
6. Concatenate the lists B[0], B[1] ..... B[n-1]