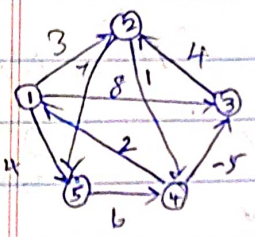
All pairs shortest path:

- It is used to find the shortest paths b/w every pair of vertices from a given weighted graph.
- The Algorithm used for this is "Floyd Warshall Alg"
- It will generate a matrix which will represent the minimum distances from any node to any other node

⇒



Adj Matrix:

Dist cost matrix

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

Step 1:

Dist cost matrix

	1	2	3	4	5
1	0	3	8	∞	4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Step 2:

Keep node 1 as intermediate node & calculate distance for every (i,j) node pair using the formula:

(if A is intermediate node)

Distance[i][j] = MIN (Distance[i][j], Distance[i][A] + Distance[A][j])

Intermediate A = 3

dist[1][2] = MIN (Dist[1][2], Dist[1][3] + Dist[3][2])

= MIN {3, 8+4}

= Min{3, 12} = 3

inter value = ①

inter node ①	1	2	3	4	5
1	0	3	8	∞	4
2	∞	0	?	?	?
3	∞	?	0	0	?
4	2	?	?	0	?
5	∞	?	?	?	0

①

	1	2	3	4	5
1	0	3	8	∞	4
2	∞				
3	∞				
4	2				
5	∞				

②

③

④

⑤

Floyd - Warshall alg:

i/p : the cost matrix

o/p : matrix with shortest path

```
begin
  for k = 0 to n do   // k = intermediate vertex
    for i = 0 to n do   // i = starting vertex
      for j = 0 to n do   // j = ending vertex
        if cost[i,k] + cost[k,j] < cost[i,j] then
            cost[i,j] = cost[i,k] + cost[k,j];
      done
    done
  done
  display the current cost matrix
end
```

↓

Implement this in home

WRITE IN RECORD.

Applications (self-study)

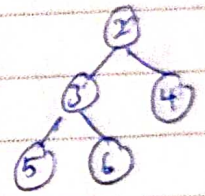
Transitive closure of graph

UNIT - IV

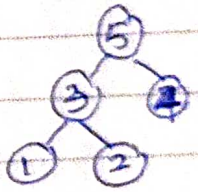
Binary Heaps - Min Max Heaps - operations

Heap - complete binary tree
↓ utmost 2 children
↓
Special Balanced BT where root node is compared with its children & arrange accordingly.

Types — Binary Heap → properties : structural - CBT
Ordering
Min Max
— Binomial heap
— Fibonacci heap.



Min Heap - root node value < value at children node
Max Heap - " > "



Heapification : Action that is taken to rearrange the heap tree when the insertion & deletion oph is involved.

Methods — Shift up
— Shift down.

02/01/24

UNIT-4

DSA

Heap - complete binary tree

Mheap - Min Heap

Heap - insert (A, item)

I/P: Heap A, item to be inserted

O/P: Updated heap.

Begin

n = n+1

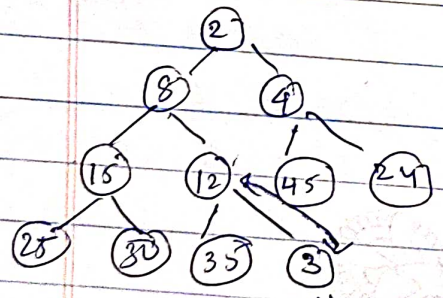
A[n] = item

shift_up(A, n)

return (A)

end

A-Array



New → Not heapified

2	8	4	15	12	45	24	25	30	35	3
1	2	3	4	5	6	7	8	9	10	↓ new

shift_up(A, i)

I/P: Heap A, A[i]

O/P: Updated heap

Begin

j = ⌊i/2⌋ // to find the parent position (floor value)

if (j > 0) and [A[i] < A[j]] then

//check the violation of heap property

A[i] ⟷ A[j] //swap

end if

shift_up(A, j) //recursively

return A

End

[A[i] < A[j]] → violate heap property.

4 < 5

includehelp.com/dsa/program-for-...

Algorithm for delete operation.

Step 1: Return the value of root.

Step 2: Replace the top node with the value of the last slot.

Step 3: Destroy the last slot

Step 4: Perform the shift down operation for the value replacing the root node until the heap satisfies the ordering property

Extract_Min(A)

I/P: Heap

O/P: value at root.

:

Begin

min = A[1] //min value is 1st value

A[i] = A[n] //copy last slot to root value

n = n - 1 // reduce the slot.

Shift_down(A, A[1]) //Heapify

End

Shift_down(A, i)

I/P: Heap A, item A(i)

o/p: updated heap

Begin

j = min(left(i), right(i)) //Min among two children must be replaced

if((j ≤ n) and (A[i] > A[j])) then

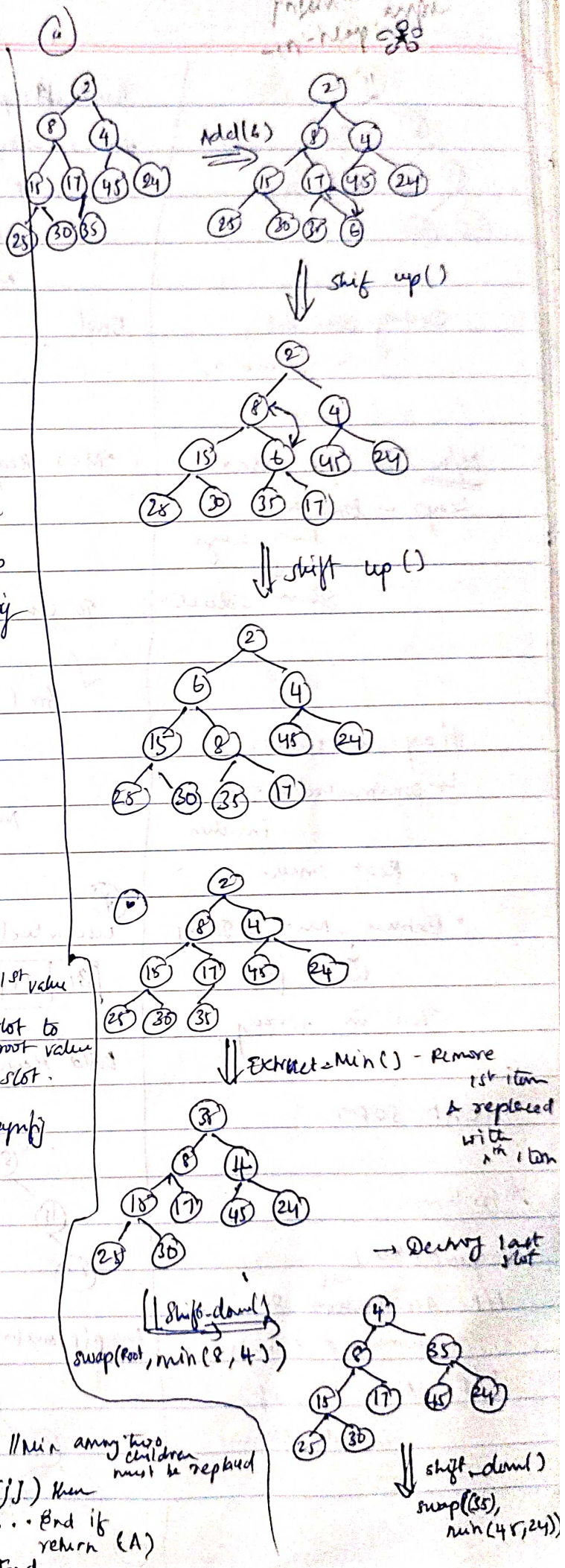
... A[i] ⟷ A[j]

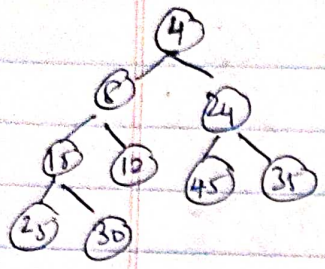
... Shift_down(A, j)

... End if

return (A)

End





Delete: done till n became 0.

H/W: ~~Do~~ for max heap — Similar to min heap
↓
Insert, delete

Heap sort (pair)
↳ constructed — Tree by insertion.

- Root - small.
- Extract_Min ⇒ Delete.
  ↓
  till nth pos = 0.
- Store in array.

HEAP SORT:

Algorithm:

Heapsort (arr)
I/P: Array arr of unsorted elements
O/P: Array arr of sorted element

Build Heap(arr)
for (i = length (arr) to 2)
swap arr[i] with arr[1]
heap-size[arr] = heap-size[arr] - 1
Max Heapify (arr, 1)
End

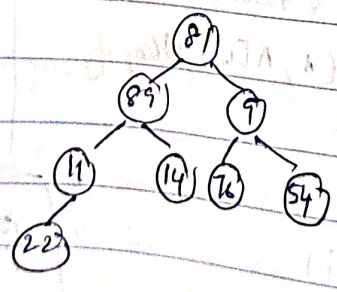
- Max Heap: Root → max
  ↓
  placed in last position in array

To use min heapify
↓
for (i = 2 to length (arr))
⋮
Min heapify (arr, 1)

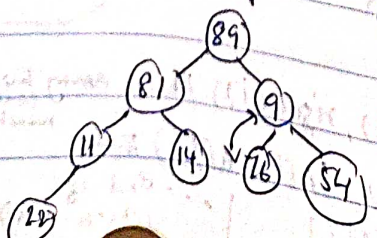
(*)
Unsorted elements:

81	89	9	11	14	76	54	22

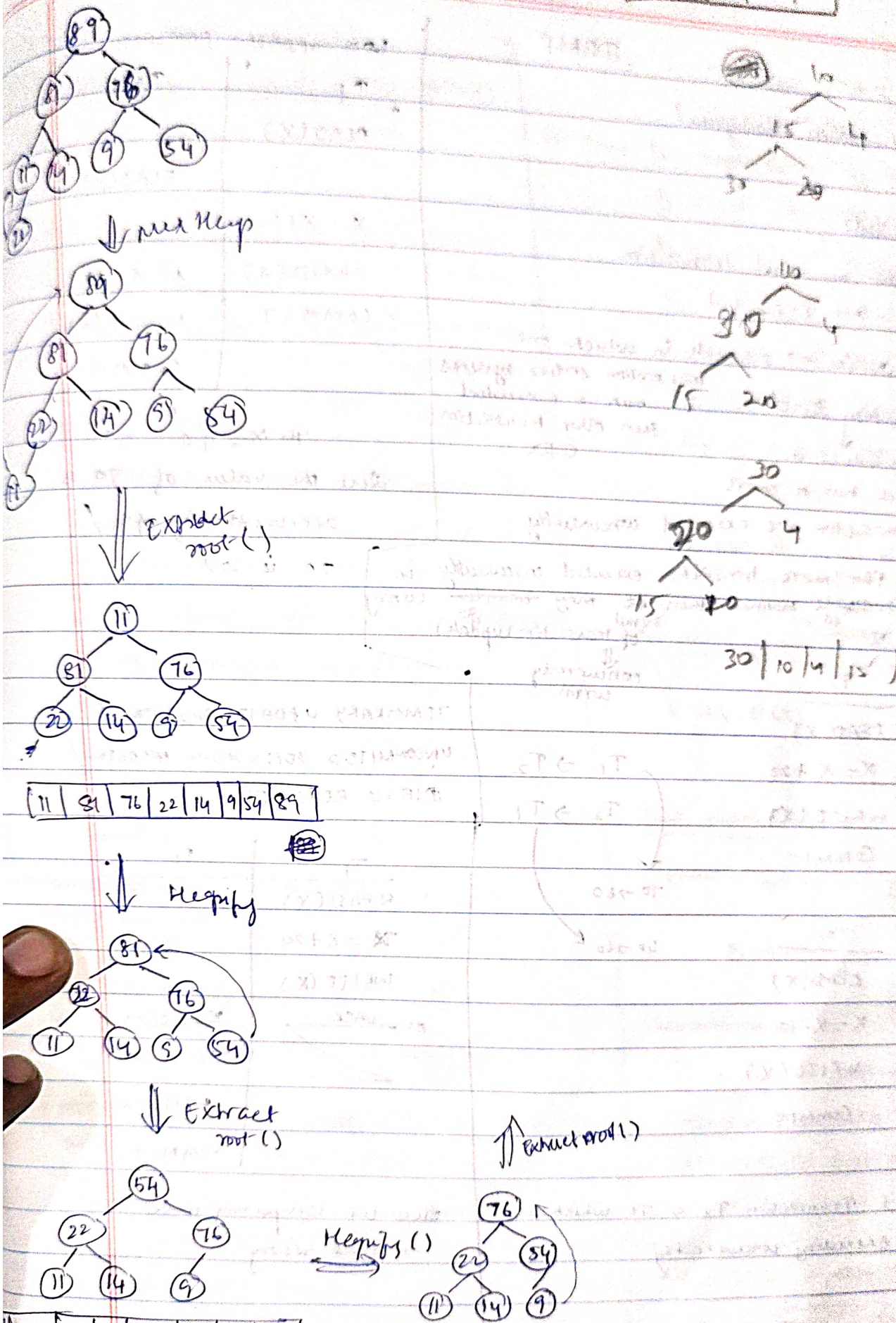
Build Heap



swap (1, max(arr)) || Max Heapify



Max Heap


10 | 15 | 4 | 30 | 20
Max Heap
Extract root()
11 | 81 | 76 | 22 | 14 | 9 | 54 | 89
Heapify
Extract root()
Heapify()
Extract root()
30 | 10 | 4 | 15 | 20

20 April
Tuesday

110-255 / Week 17

April 2021
S M T W T F S
1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

Min Heap:

INSERT:

Heap_Insert (A, item):
I/P: Heap array A, item to be inserted
O/P: updated heap

```
Begin
    n = n+1
    A[n] = item
    shift_up (A,n)
    return A
End

Shift_up(A,i)
I/P: Heap A, A[i]
O/P: updated heap

Begin
    j = ⌊i/2⌋ // find parent pos
    if (j>0) and [A(i) < A(j)]
      //check violation property
      A[i] ⟷ A[j] //swap
    end if
    shift_up (A,j) //recursive
    return A
End
```

DELETE:

Extract_Min (A)
I/P: Heap A
O/P: value at root

```
Begin
    min = A[1]  // min → first value
    A[1] = A[n]  // copy last slot
                    to root value
    n = n-1     // destroy last leaf
    shift_down (A, A[1]) // Heapify
End.

shift_down (A,i)
I/P: Heap A, A[i] (item)
O/P: Updated heap.

Begin
    j = min (left(i), right(j))
    if (j≤n) and (A[i] > A[j]) then
        A[i] ⟷ A[j]
        shift_down (A,j)
    end if
    return A
End
```


May 2021
S M T W T F S
30 31 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29

April 21
Wednesday

111-254 / Week 17

Max Heap:

INSERT:

Heap_Insert (A, item)
I/P: Heap A, item to be inserted
O/P: updated heap

```
BEGIN
    n = n+1
    A[n] = item
    shift_up (A,n)
    return A
END

shift_up (A,i)
I/P: Heap A, A[i]
O/P: Updated Heap

Begin
    j = ⌊i/2⌋
    if (A[j] < A[i])
      A[i] ⟷ A[j]
    end if
    shift_up (A,j)
    return A
End
```

DELETE

Extract_Max (A)
I/P: Heap A
O/P: Value at root

```
Begin
    max = A[1]
    A[1] = A[n]
    n = n-1
    Shift_down (A, A[1])
End

Shft_down (A,i)
I/P: Heap A, A[i]
O/P: updated heap

Begin
    j = max (left(i), right(i))
    if (j≤n) and (A[i] < A[j])
        A[i] ⟷ A[j]
        Shift_down (A,j)
    end if
    return A
End
```

Quick sort:
- recursive procedure
- divide & conquer.
- select any element - pivot

1st Any Last ele

position pivot at its correct position

Lesser ← pivot → greater

quicksort:

```
while(i<j)
{
    i++=1;
    while (i<=r .&& a[i] <pivot)
        i++;
    while (j>=1 .&& a[j]>pivot)
        j--;
    if(i<j && i<=r)
        swap(a,i,j);
}
swap(a,i,j)
quicksort(a,1,j-1);
quicksort(a,j+1,r);
```

Merge sort:
- Divide & conquer method.
- recursively called.
- Divide array until it can't be divided further
- sort & merge the elements.

```
mergesort(array a)
  if (n==1)
    return a
  arrayOne = a[0] ... a[n/2]
  arrayTwo = a[n/2+1] ... a[n]
  arrayOne = mergesort(arrayone)
  arrayTwo = mergesort(arrayTwo)
  merge(arrayone, arrayTwo)

merge(array a, array b)
  array c
  while (a & b have elements)
    if (a[0] > b[0])
      add b[0] to end of c
      remove b[0] from b
    else
      add a[0] to end of c
      remove a[0] from a
  // At this pt either a/b is empty
  while (a has elements)
    add a[0] to the end of c
    remove b[0] from b
  return c
```

24 April

Saturday

114-251 / Week 17

Insertion sort :

```
procedure insertionsort (A: array of item)
    int holeposition
    int value_To_Insert
for i=1 to length(A) inclusive do
  // select value to be inserted
      value_to_insert = A[i]
      holeposition = i
    while holeposition > 0 && A[holeposition-1] > value_to_insert do
        A[holeposition] = A[holeposition-1]
        holeposition = holeposition - 1
    endwhile
        A[holeposition] = value_To_Insert
  end for
end procedure
```

25 Sunday ○

$u(x,y) = (\text{constant})\, e^{-\lambda x} e^{\pm i\lambda y} \quad \textcircled{2}$

$u(x,0) = 0$

$u(\infty,0) = 0$

$\textcircled{2} \Rightarrow u(x,0) = (\text{constant})\, e^{-\lambda x 0} \pm \frac{i}{2\sqrt{2}\alpha} e^{-\alpha^2/4}$

$\text{constant} = \frac{-i}{2\sqrt{2}\alpha} e^{-\alpha^2/4}$

$\textcircled{2} \Rightarrow u(x,y) = \frac{-i}{2\sqrt{2}\alpha} e^{-\alpha^2/4} e^{-\lambda x} e^{i\lambda y} + \frac{i}{2\sqrt{2}\alpha} e^{-\alpha^2/4}$

$= (1 - e^{-\lambda x} e^{i\lambda y}) \frac{e^{-\alpha^2/4}}{2\sqrt{2}\alpha}$

DS:

Merge sort:

```
mergeSort (array a)
if (n==1)
    return a
arrayOne = a[0] ... a[n/2]
arrayTwo = a[n/2+1] .... a[n]
arrayOne = mergeSort (arrayOne)
arrayTwo = mergeSort (arrayTwo)
merge (arrayOne, arrayTwo)
merge (array a, array b) // function
array C
while (a & b have elements)
if (a[0] > b[0])
    add b[0] to the end of c
    remove b[0] from b
else
    add a[0] to the end of c
    remove a[0] from a
// At this point either a or b is empty
while (a has elements)
    add a[0] to the end of C
    remove b[0] from b
return C
```

Insertion Sort

```
Procedure insertionSort (A : array of items)
    int holePosition
    int valueToInsert
for i=1 to length(A) inclusive do:
    // select value to be inserted
    valueToInsert = A[i]
    holePosition = i
    // locate hole position for the element
            to be inserted
    while holePosition > 0 and A[holePosition-1]
            > valueToInsert do:
        A[holePosition] = A[holePosition - 1]
        holePosition = holePosition - 1
    end while
    // insert the number at the hole position
    A[holePosition] = valueToInsert
end for
end Procedure.
```

Quick sort:

```
while (i < j) {
    i++;
while (i<=r && a[i]<pivot)
    i++;
while (j>=l && a[j]>pivot)
    j--;
if (i < j && i<=r)
    swap (a, i, j);
}
swap (a, i, j) // i crossed j
quickSort (a, l, j-1); // we swap with the pivot
quickSort (a, j+1, r),
```

Quick Sort:

- recursive procedure based on devide & conquer.
- select any element - pivot
  - 1st ele
  - any ele
  - last ele

position it at correct position

lesser ← pivot → greater

eg:

9 7 5 14 2 3 6 10

i, L pivot, j, R

a[i] < pivot

7 < 9 ✓ i++

9 7 5 14 2 3 6 10

L Pi i jR

5 < 9 ✓ i++

9 7 5 14 2 3 6 10

LPi i jR

14 < 9 ✗

a[j] > pivot

10 > 9 ✓

j-- j

9 7 5 14 2 3 6 10

9 7 5 6 2 3 14 10

i j

6 > 9 ✗

9 7 5 6 2 3 14 10

i j

a[i] < 9

6 < 9 ✓

9 7 5 6 2 3 14 10

i j

2 < 9 ✓

9 7 5 6 2 3 14 10

i j

3 < 9 ✓

9 7 5 6 2 3 14 10

i j

14 < 9 ✗

14 > 9 ✓

9 7 5 6 2 3 14 10

j i

copy

3 7 5 6 2 9 14 10

LPi i jR pi

7 < 3 ✗

2 > 3 ✗

3 2 5 6 7 9 14 10

LPi i jR

2 < 3 ✓

3 2 5 6 7

LPi i jR

7 > 3 ✓

3 2 5 6 7

i j

6 > 3 ✓

3 2 5 6 7

ij

5 > 3 ✓

3 2 5 6 7

pi j i

2 3 5 6 7 9 14 10

LPi UPi

14 < 14 ✓

10 → 14 ✗

2 3 5 6 7 9 10 14 ✓

Linear Search:

Algorithm

Program

Ex:

Merge sort:

Divide & conquire method

- recursively called → n=1 → split the

5 3 2 7 1 8 4 5

Divide array

5 3 2 7 1 8 4 5

5 3 2 7 1 8 4 5

5 3 2 7 1 8 4 5

2 3 5 7 1 4 5 8

Insertion sort:

(Running cards)

Consider one element & insert it at its correct position.

hole position → position to keep value to insert → value i.e. limited

Binary Search

(1) First sort the elements.

Find mid value & compare.