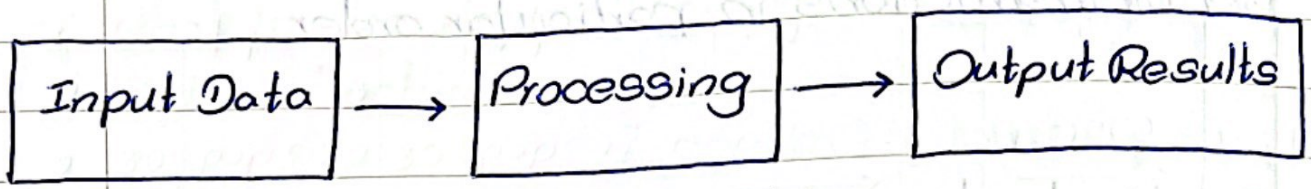
Date No

Context of Computing.



Computer Program

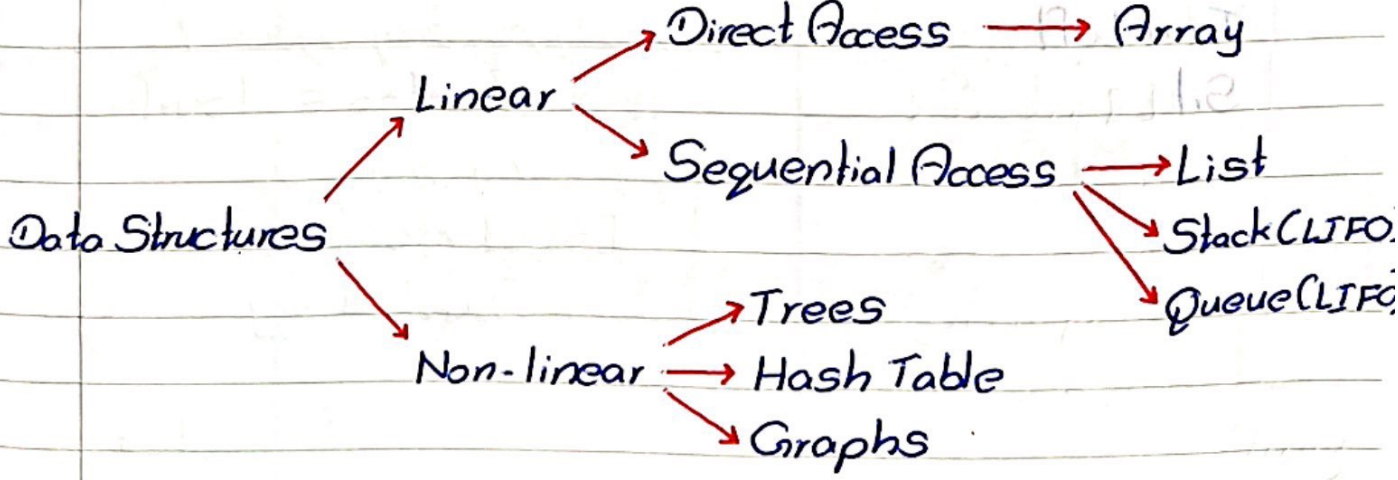
Instructions + Data

- Algorithms
- Data Structures

Data Structures.

A way of storing data or it's a memory arrangement

Data Structures Classification.



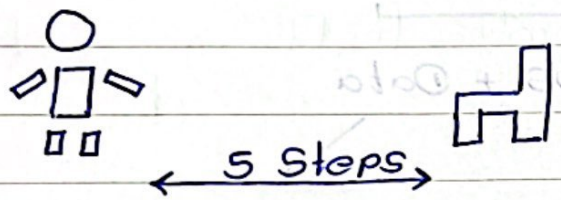
* LIFO → Last In First Out.
* FIFO → First In First Out

Date No

Algorithms.

Set of instructions in particular order.

Give instructions to OTTO to sit on the chair.



Turn Left
Walk
Walk
Walk
Walk
Walk
Turn Left
Turn Left
Sit

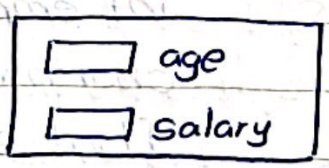
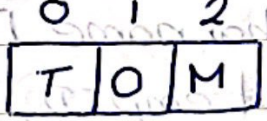
Date No

Structures in C

01. What is the main difference between an array and a structure?

02. What is the keyword used to declare a structure?

Variable	Structure	Array

int age;
↑ data type ↑ variable name
 
age
	struct employee {
	int age;
	Float salary;
	 
	
	};
		char arr [3];
		 

Date No

Assume we need to store empID, name and salary of each employee.
Declare a structure to store above components.

```
struct employee {
        int empID;
        char name [10];
        Float salary;
};
```

* Do not initialize values inside the declaration.

Use the above declared structure to store structure variables for each employee.
There're 3 employees working.

Method 01: (with structure definition)

```
struct employee {
        int empID;
        char name [10];
        Float salary;
} emp1, emp2, emp3;
```

Method 02: (seperatly)

```
struct employee {
        int empID;
        char name [10];
        float salary;
};

struct employee emp1, emp2, emp
```

Date No

Use the above declared structure and structure variables to store details of each employee.

```
emp1.empID = 1290;
strcpy(emp1.name, "John");
emp1.salary = 345.67;

Struct employee emp1 = {1290, "John", 345.67};
```

```
#include<stdio.h>
#include<string.h>

struct employee {
    int empID;
    char name[10];
    Float salary;
};

int main() {
    struct employee emp1;

    emp1.empID = 1290;
    strcpy(emp1.name, "John");
    emp1.salary = 345.67;
```

```
    printf("EmpID of emp1 is %d \n", emp1.empID);
    printf("Emp Name of emp1 is %s \n", emp1.name);
    printf("Emp Salary of emp1 is %f \n", emp1.salary);

return 0;
}
```

```
#include <stdio.h>
#include <string.h>

struct employee {
        int empID;
        char name[10];
        float salary;
};

int main()
    struct employee emp1 = {1290, "John", 345.67};

    printf("EmpID of emp1 is %d \n", emp1.empID);
    printf("Emp Name of emp1 is %s \n", emp1.name);
    printf("Emp Salary of emp1 is %f \n", emp1.salary);

return 0;
}
```

Date No

```
#include <stdio.h>
#include <string.h>

typedef struct {

    int empID;
    char name[10];
    float salary;
}employee;

int main() {
    employee emp1;

        emp1.empID = 100;
        strcpy(emp1.name, "John");
        emp1.salary = 350.90;

    employee emp3 = {200, "John2", 450.80};

    printf("Details: %s", emp3.name); //just to check
                                      whether values
                                      are stored.

return 0;
}
```

Date No

Structure.

A structure is a user defined data type in C language that allows us to combine different types.

Structure helps to construct a complex data type that is more meaningful.

① Defining a Structure.

```
struct [structure_tag] {
        //member variable 01
        //member variable 02
        //member variable 03
        ...
} [structure_variables];
```

Example:

```
struct Employee {
        char name [50];
        int empID;
        char name [50];
        float salary;
};
```

Date No

Array of Structure

We can also declare an array of structure variable in which each element of the array will represent a structure

Example:

struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type employee

#include <stdio.h>

struct employee {
int id;
char name[10];
float salary;
} emp[5];

int i;

void main() {

Date No

Array of Structure.

We can also declare an array of structure variables, in which each element of the array will represent a structure.

Example:

```
struct employee emp [5];
```

The below program defines an array emp of size 5. Each element of the array emp is of type employee.

```
#include <stdio.h>

struct employee {
        int sal;
        char name [10];
};

struct employee emp [5];
int i, j;

void ask () {
```

```
    for (i=0; i<3; i++) {
        printf ("\n Enter %dst employee record \n",
                                        i+1);

        printf ("\n Employee name \t");
        scanf(" %s",&emp[i].name);
        printf("\n Salary \t");
        scanF("%d",&emp[i].sal);
}

printf("\n Displaying Employee Record \n");

For (i=0; i<3; i++)

{

printf("\n Employee name is %s", emp[i].name);
printf("\n Salary is %d", emp[i].sal);

}

}

void main()
{
   ask();
}
```

Date No

```
#include <stdio.h>

struct employee {
        int empID;
        char name [10];
        float salary;
};

//struct employee emp1 = {1290, "John", 345.67};

struct employee emp[5];

int i;
void in () {

    for (i= 1; i<=3; i++) {

            printf ("Enter emp %d ID", i);
            scanf("%d", &emp [i].empID);
            printf("Enter emp %d name", i);
            scanf("%s", &emp [i].name);
            printf ("Enter emp%d salary", i);
            scanf("%f", &emp[i].salary);

    }
}
```

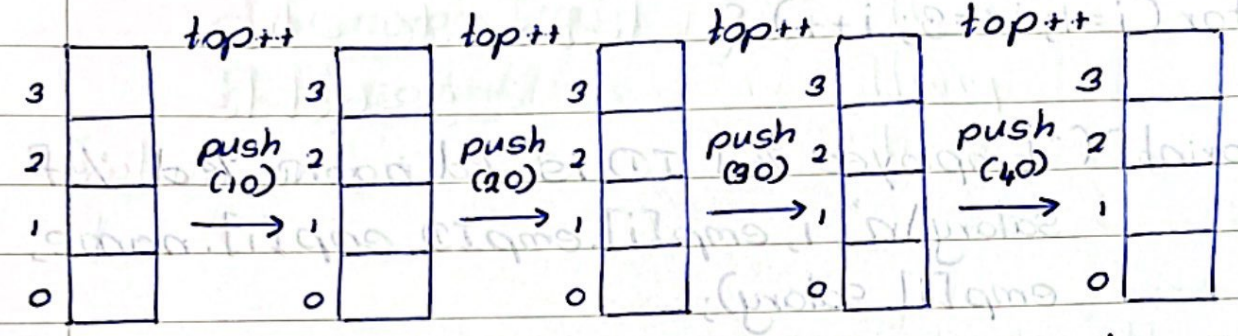
Date No

```
void out () {

printf ("Printing Employee Details \n");

for (i = 1; i <= 3; i++) {

printf ("Employee %d ID, %d name, %f
        salary \n", i, emp[i].empID, emp[i].name,
        emp[i].salary);

}

}

int main () {

in();
out();

return 0;

}
```

Date No

Stacks in C (LIFO) - Last In First Out

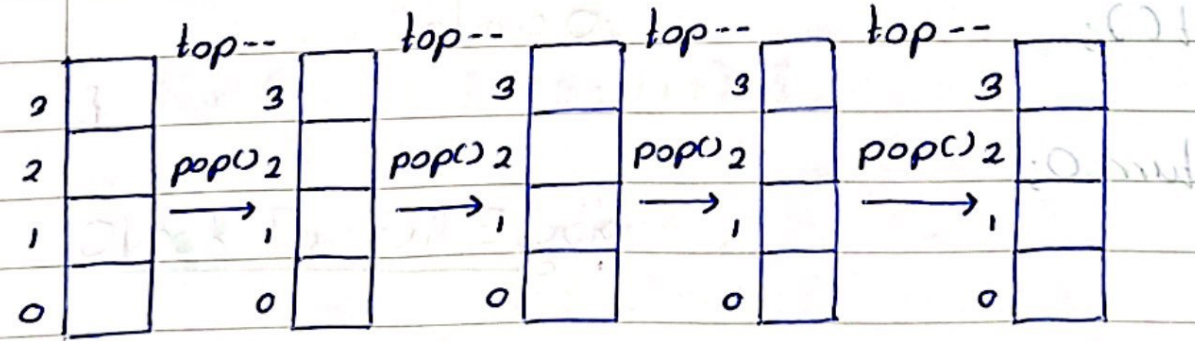
Push Operation



top = (-1) (initial) top = 0 top = 1 top = 2 top = 3

push (ele)

Pop Operation



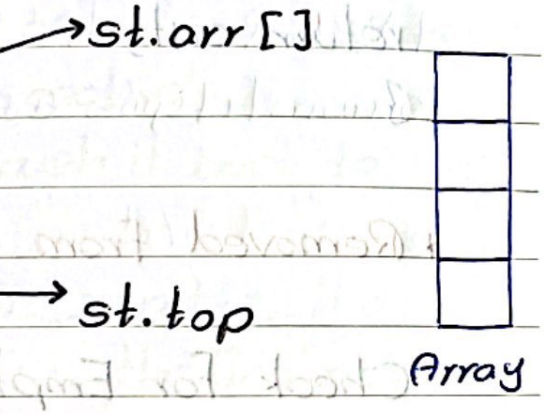
top = 3 top = 2 top = 1 top = 0 top = (-1)

pop ()

Date No

Creating a New Stack

```
#define size 5

//Stack structure

struct stack {

        int arr [size];
        int top;

} st;
```



Inserting an Element - Push

```
void push (int ele) {
        st.top++;
        st.arr [st.top] = ele;
}
```

* Added to the top of the stack.

Date No

Removing an Element - Pop

```
int pop () {
    int out;
    out = st.arr [st.top];
    st.top--;
return out;
}
```

* Removed from the top of the stack.

Check for Empty Stack.

```
int stempty () {
    if (st.top == -1)
        return 1;
    else
        return 0;
}
```

Check For Full Stack.

```
int stfull () {
    if (st.top >= size-1)
        return 1;
    else
        return 0;
```

Date No

Evaluate a PostFix Notation Using Stacks.

Algorithm: maintain a stack and scan the postfix expression from left to right.
If the element is a number, push it into the stack.
If the operator is an operator O, pop twice and get A and B respectively.
Calculate BOA and push it back to the stack.
When the expression is ended, the number in the stack is the final answer.

Date No

ProMate

Date No

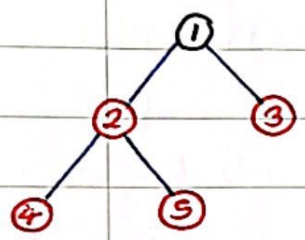
Using Depth First Search (DFS) in Stacks.

Approach: Depth First Search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there

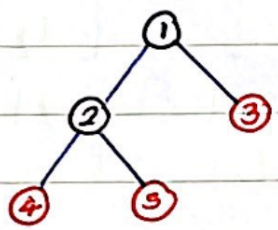
Date No

is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

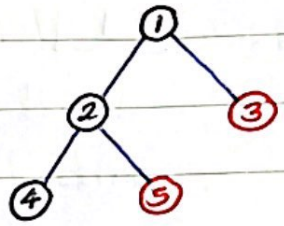
1



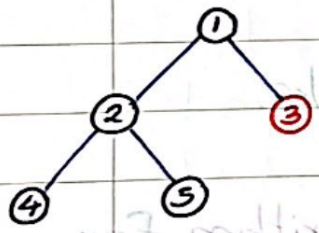
1 → 2



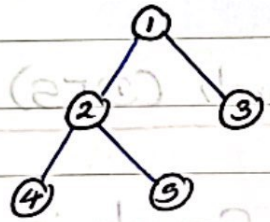
1 → 2 → 4



1 → 2 → 4 → 5



1 → 2 → 4 → 5 → 3



Date No

Queues in C (FIFO) - First In First Out

There're 2 types of Queues.

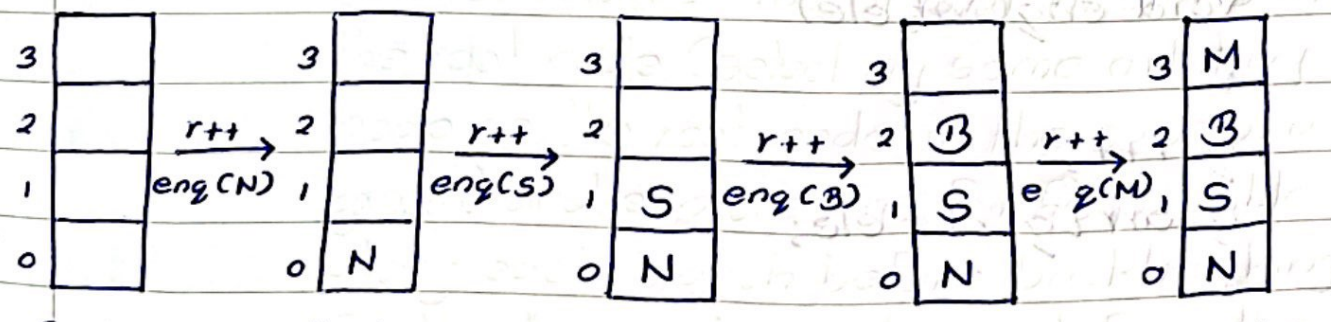
01. Linear Queue.
02. Circular Queue.

01) Linear Queue.

* Insert element → enq
* Remove element → deq

N, S, B, M

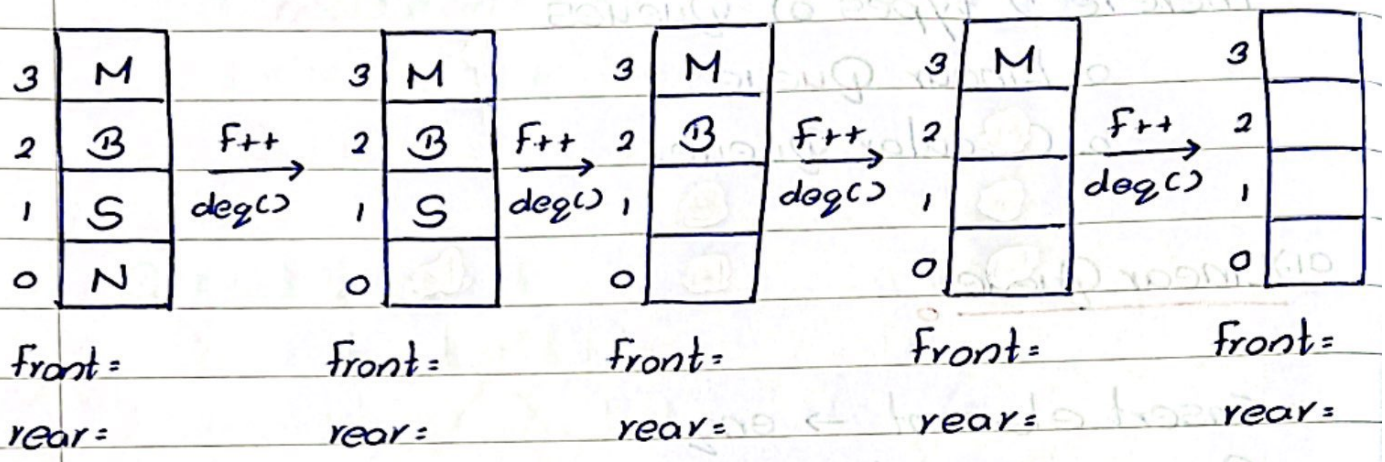
Enq Operation.



front=0 rear=-1 | front=0 rear=0 | front=0 rear=1 | Front=0 rear=2 | front=0 reor=3

Date No

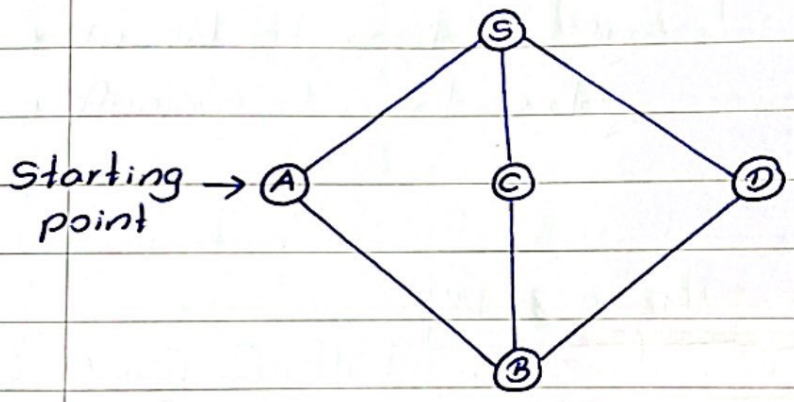
① Deq Operation.




```
#define size 5
struct queue {
int arr [size];
int r,f;
} q;

void enq(int ele)
{
q.r++;
q.arr[q.r] = ele;
}

int deq()
{
int out;
```

```
out = q.arr[q.f];
q.f++;
return out;
}
```

Breadth First Search



Date No

Date No

Date No

Creating a new Queue.

```
#define size 5

//Queue structure.

struct queue {
        int q [size];
        int rear = -1, front = 0;
} st;
```

Inserting an Element ; enqueue

```
void enqueue (int item)
{
    st.rear++;
    st.q [st.rear] = item;
}
```

* Added to the rear of the queue.

* Before you should check for the queue is full or not.

Date No

Removing an element; dequeue.

```
int dequeue()
{
    int item;
    item = st.s [st.front];
    st.front++;
    return(item);
}
```

* Removed the front item from the queue.

Check for Empty Stack

```
(rear == MAX)
```

Check for Full Stack.

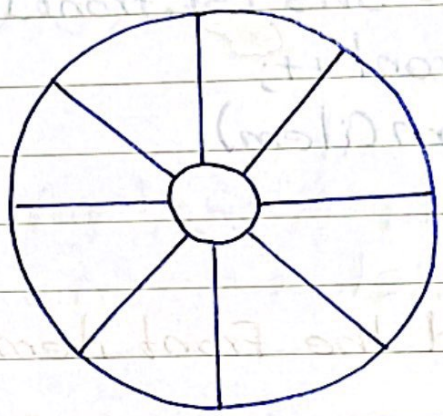
```
(front == rear)
```

* Efficiency of insert / remove from queue is O(1)

Date No

02.) Circular Queue.

Linear queue $\xrightarrow{\text{wrapping}}$ Circular queue.



* ① Declare all the elements.

- Array
- Front = -1
- rear = -1

* Function to perform enque.

- Empty
- Full
- Few elements

Date No

* Function to perform deque

- Full / Few
- Empty
- Single.

```
#include <stdio.h>

int arr [5];
int front = -1;
int rear = -1;

//function to perform enq

void enqueue (int x) {
    if (front == -1 && rear == -1) {
            front = 0;
            rear = 0;

//access the rear

            arr[rear] = x;
            }
    if else ((rear + 1) % 5 == front) {
            printf ("Queue is Full");
}
```

```
    else {
            rear = (rear + 1) % N;
            arr[rear] = x;
    }

// function to perform deq

void dequeue () {
    if (front == -1 && rear == -1) {
        printf ("No elements");
    }
    else if (rear == front) {
        printf ("%d", arr[front]);

        front = -1;
        rear = -1;
    }
```

Queue in Action

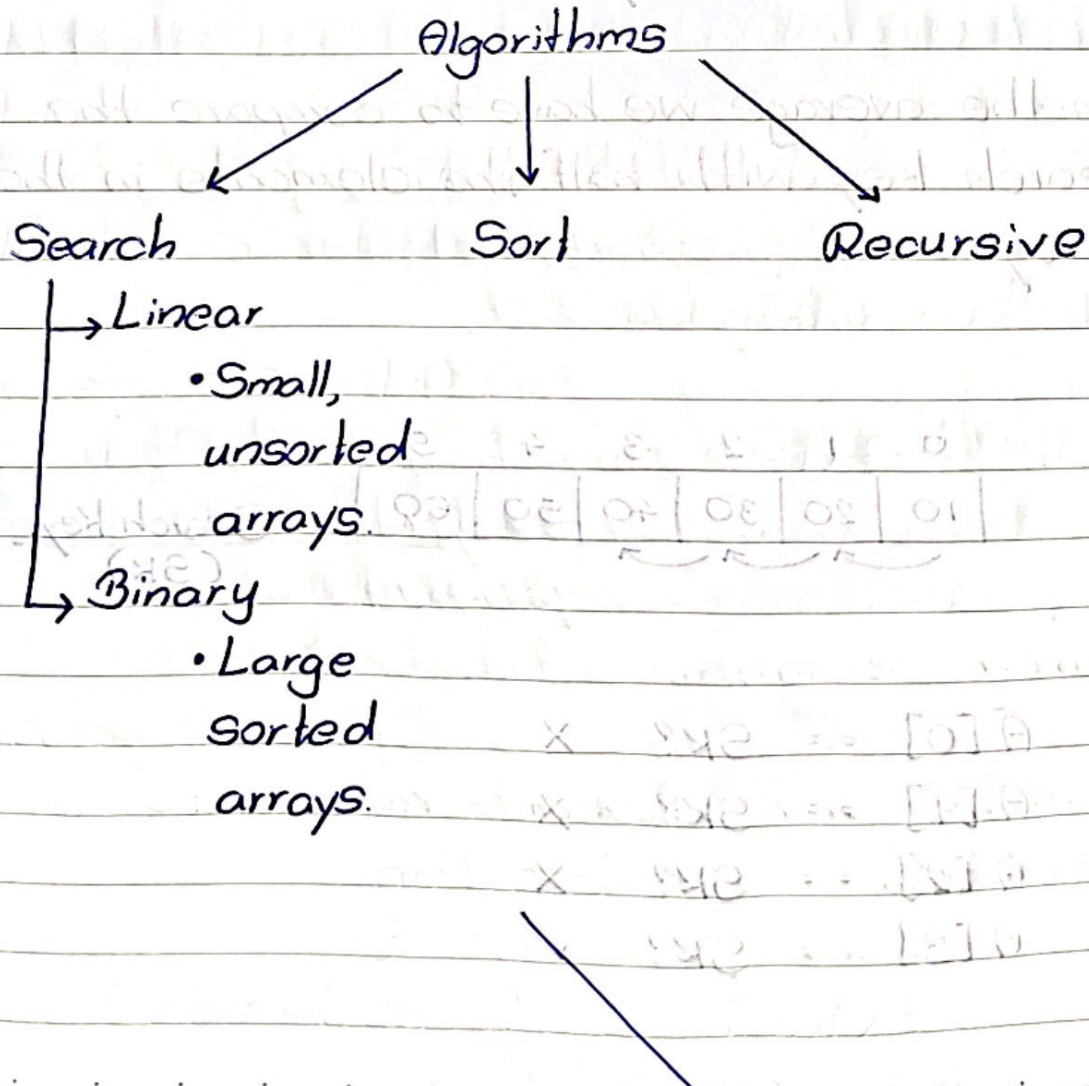
Active part of the queue keeps on moving and hit the array limit at some stage

Date No

Applications.

- Print Queue.
- Buffers for communication between two entities.
  - Network cards need a packet buffer.
  - Secondary storage write through buffer.

Searching Algorithms



Date No

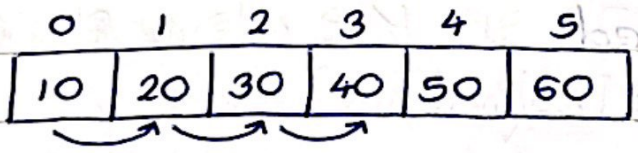
Linear Search Algorithm

01.) Start at the first element of array.

02.) Compare value to value (key) for which you're searching.

03.) Continue with next element of the array until you find a match or reach the last element in the array.

* On the average we have to compare the search key with half the elements in the array.



Search Key (SK) = 40

A[0] == SK? X
A[1] == SK? X
A[2] == SK? X
A[3] == SK? ✓

Date No

case \ number	10	100	1000
Best	0th last		
Average	4th mid		
Worst	9th (end) not found		

t_1 t_2 t_3

Worst case:

$10 \rightarrow t_1$

$100 \rightarrow t_2$ $\quad t_1 < t_2 < t_3$

$1000 \rightarrow t_3$

$t \rightarrow$ time

$N \rightarrow$ number of elements

$$t \propto N$$

Date No

* For a larger data set, linear search won't work. Because when the time increase, the performance of the algorithm decrease.

* Linear search algorithm is suitable for small data sets. (sorted or unsorted.)

Algorithm

```
int lsearch (int arr[], int sk, int N) {

        bool found = false;
        int position = -1;
        int index = 0;

    while (index < N && !found) {
            if (arr[index] == sk) {
                position = index;
                found = true;
            }
            index++;
    }
    return position;
}
```

Date No

Ex: -1

0	1	2	3	4	5
10	2	15	25	18	30

SK = 18

index	arr[index]	SK	found	position
0	10	18	F	-1
1	2	18	F	-1
2	15	18	F	-1
3	25	18	F	-1
4	18	18	T	4

* if SK ≠ arr[index], found & position will remain same.

A Linear Search Function.

```
int searchlist(int list[], int numelems, int value){
    int index = 0; //used as a subscript to search array
    int position = -1; //to record position of search value
    bool found = false; //Flag to indicate if value was found.
while (index < numelems && !found) {
    if (list[index] == value) //if the value is found {
        found = true; //set the Flag
        position = index; //record the value's subscript
```

```
    }
    index++; //go to the next element
  }
return position; //return the position, or -1
}
```

Binary Search Algorithm

01.) May only be used on a sorted array.
02.) Eliminates one half of the elements after each comparison.
03.) Locate the middle of the array.
04.) Compare the value at that location with the search key.
05.) If they're equal - DONE!
06.) Otherwise, decide which half of the array contains the search key.
07.) Repeat the search on that half of the array and ignore the other half.
08.) The search continues until the key is matched or no elements remain to be searched.

Date No

0	1	2	3	4	5
10	2	15	25	18	30

SK = 18

sorting

midI

f l

0	1	2	3	4	5
2	10	15	18	25	30

lower boundry (LB) upper boundry (UB)

$$midI = \frac{f + l}{2} = \frac{0 + 5}{2} = 2.5 \rightarrow 2$$

01.) arr[midI] == SK

15 18

02.) arr[midI] > SK || LB

15 18

l = midI - 1;

ProMate

Date No

03.) II UB

10	2	15	25	18	30

f = midI + 1

```
while(!found && first <= last) {
        int midI =(first + last) / 2;
  if (arr[midI] == sk) {
        found = true;
        position = midI;
  } else if (arr[midI] > sk) {
        last = midI - 1;
  } else {
        first = midI + 1;
   }
 }
return position;
}
```

Date No

Ex:

-1	0	1	2	3	4	5
	2	10	15	18	25	30

SK = 18

First	last	midI	arr[midI]	SK	found	position
0	5	2	15	18	F	-1
3	5	4	25	18	F	-1
3	3	3	18	18	T	3

if the SK = 12 (within the boundry (2-30))

First	last	midI	arr[midI]	SK	found	position
0	5	2	15	12	F	-1
0	1	0	2	12	F	-1
1	1	1	10	12	F	-1

if the SK = 35 Coutside the boundry)

first	last	midI	arr[midI]	SK	Found	position
0	5	2	15	35	F	-1
3	5	4	25	35	F	-1
5	5	5	30	35	F	-1

Date No

Efficiency of Binary Search.

* Searching an array of 1024 elements will take at most 10 passes to find a match or determine that the element does not exist in the array.

  512, 256, 128, 64, 32, 16, 8, 4, 2, 1

* An array of one billion elements takes a maximum of 30 comparisons.

* The bigger the array the better a binary search is as compared to a linear search.

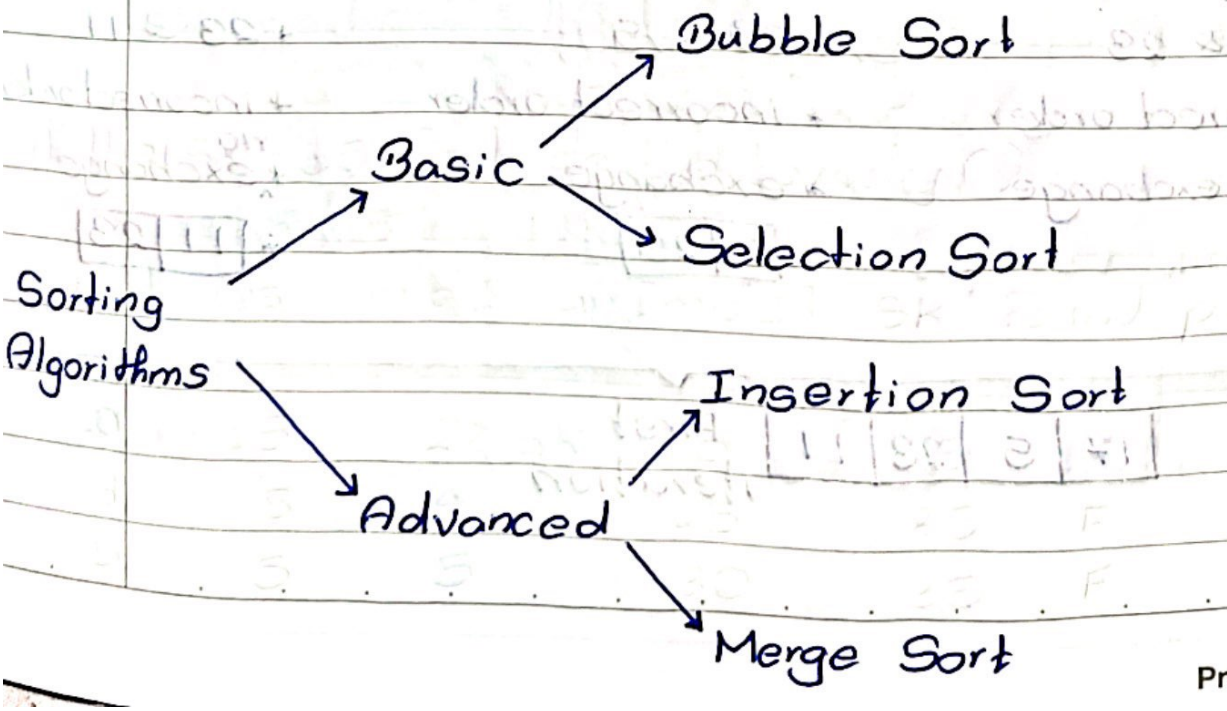
A Binary Search Function.

```
int binarySearch(int arr[], int size, int value){
    int first = 0, // first array element
        last = size - 1, // last array element
        middle, // mid point of search
        position = -1; // position of search value
    bool found = false; // flag
while (!found && first <= last){
    middle = (first + last) / 2; // calculate mid point
    if (arr[middle] == value) { // if value is found at mi
        found = true;
```

```
            position = middle;
        }
        else if (arr[middle] > value)
            last = middle - 1; //if value is in lower half
        else
            first = middle + 1; // if value is in upper half
        }
        return position;
    }
```

Introduction to Sorting Algorithms.

* Sorting methods : Alphabetical.
  Ascending numeric.
  Descending numeric.

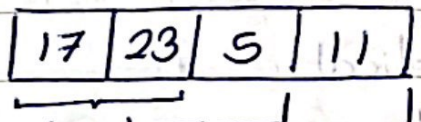


Date No

Bubble Sort

01.) Compare 1st two elements.
* If out of order, exchange them to put in order.
02.) Move down one element, compare 2nd and 3rd elements, exchange if necessary.
03.) Continue until end of array.
04.) Pass through array again, exchanging as necessary.
05.) Repeat until pass made with no exchanges.

Ex:



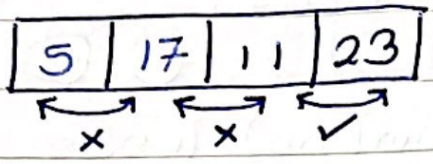
* compare values
* 17 & 23
* correct order
* no exchange

* compare values
* 23 & 5
* incorrect order
* exchange
* 5 | 23

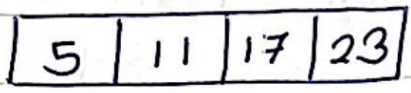
* compare value
* & 11
* incorrect orde
* no exchange

First iteration.

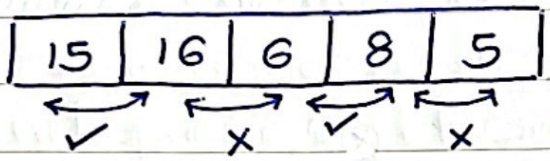
→ Second iteration

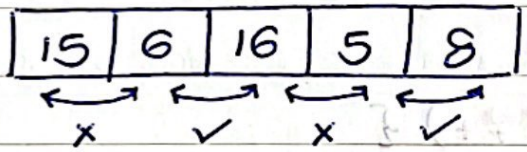


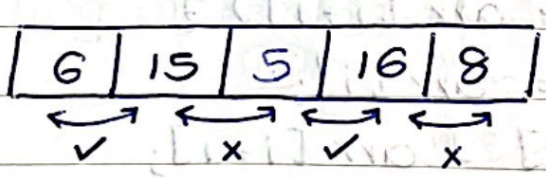
→ iteration.

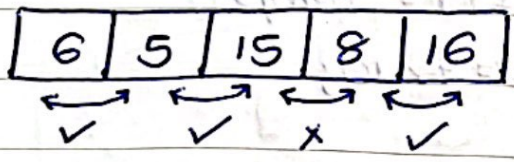


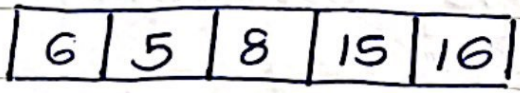
Ex:











```
int temp;
if (arr[i] > arr[i+1])
{
temp = arr[i];
arr[i] = arr[i+1];
arr[i+1] = temp;
}
```

```
do {
    swap = false;
    for (i=0; i<n; i++) {
        if (arr[i] > arr[i+1]) {
            temp = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = temp
            swap = true;
        }
    }
}while (swap)
```

if conditional check
arr[i] > arr[i+1] → i<n

moves to swap

Date No

iteration	i	i+1	arr[i]		arr[i+1]	swap	
						F	
	0	1	12	<	20	F	
	1	2	20	>	18	T	
	2	3	20	>	10	(T)	→ 12, 18, 10, 20

* Benifit of Bubble Sort:
  - Easy to understand & implement.

* Disadvantage of Bubble Sort:
  - Inefficient: slow for large arrays.

Date No

Selection Sort

Concept For sort in ascending order:

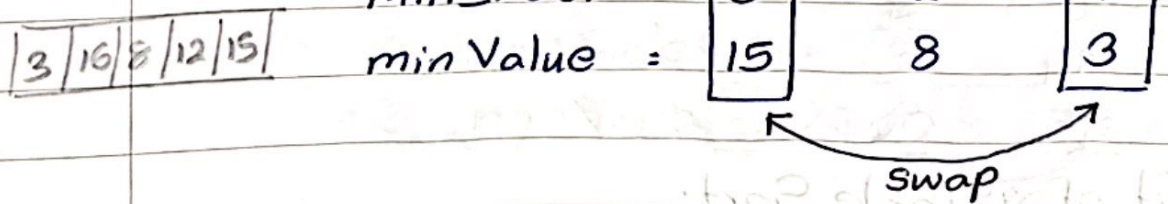
01.) Locate smallest element in array.
02.) Exchange it with element in position (index) 0.
03.) Locate next smallest element in array.
04.) Exchange it with element in position (index) 1.
05.) Continue until all elements are arranged in order.

Ex1:-

0	1	2	3	4
15	16	8	12	3

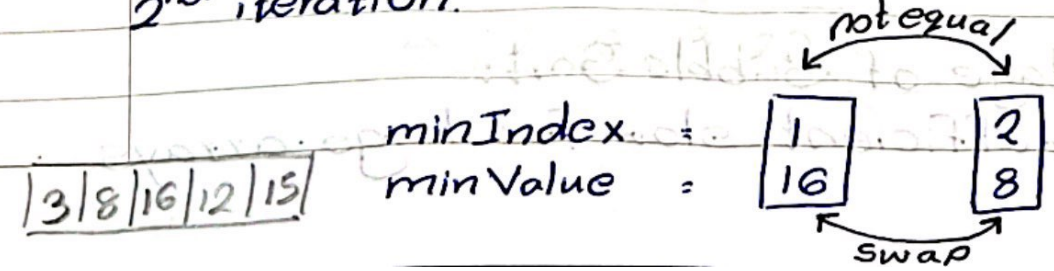
Assume

1st iteration.



3	16	8	12	15

2nd iteration.



3	8	16	12	15

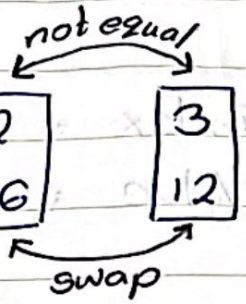
ProMate

Date No

3rd iteration.

3	8	12	16	15

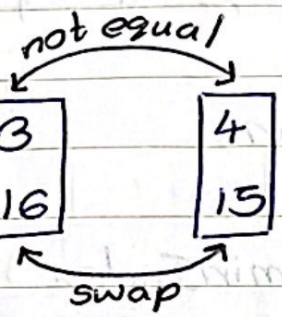
minIndex = 2 3
minValue = 16 12



4th iteration.

3	8	12	15	16

minIndex = 3 4
minValue = 16 15



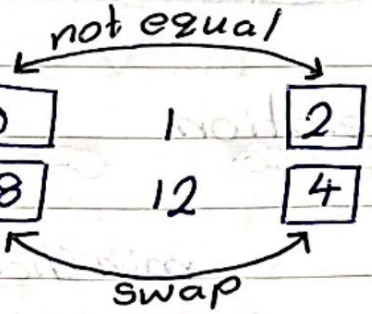
Ex2:-

0	1	2	3	4
18	12	4	20	16

Assume

1st iteration.

minIndex = 0 1 2
minValue = 18 12 4



0	1	2	3	4
4	12	18	20	16

Date No

2nd iteration

same

minIndex = 1

minValue = 12

0	1	2	3	4
4	12	18	20	16

3rd iteration

not equal

minIndex = 2 4

minValue = 18 16

swap

0	1	2	3	4
4	12	16	20	18

4th iteration.

not equal

minIndex = 3 4

minValue = 20 18

swap

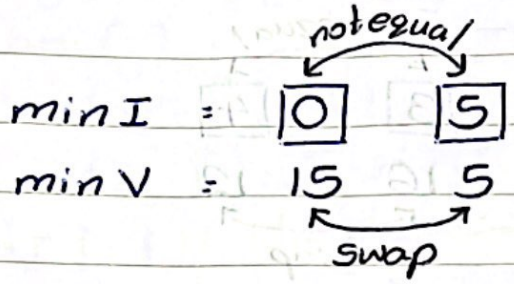
0	1	2	3	4
4	12	16	18	20

Date No

Ex3:-

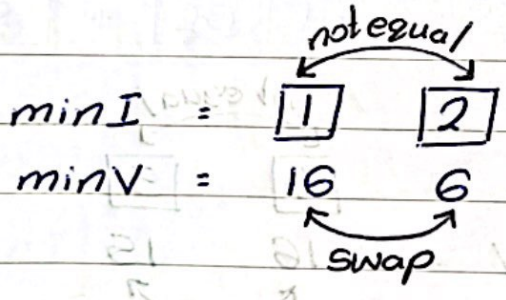
0	1	2	3	4	5
15	16	6	8	12	5

1st iteration.



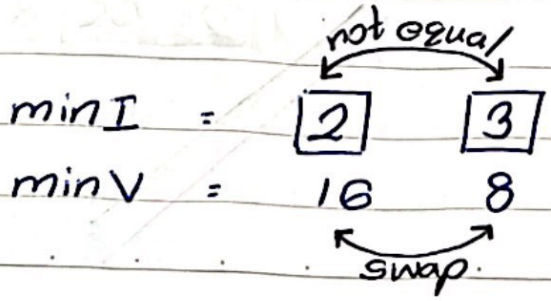
0	1	2	3	4	5
5	16	6	8	12	15

2nd iteration.



0	1	2	3	4	5
5	6	16	8	12	15

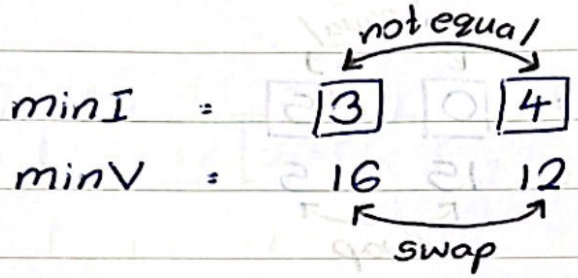
3rd iteration.



Date No

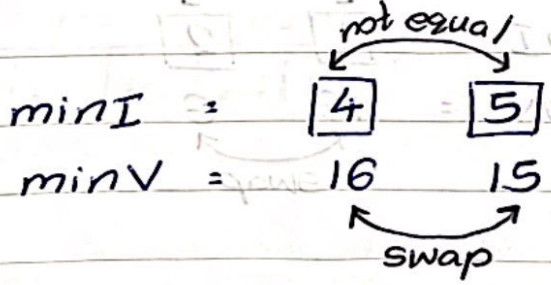
0	1	2	3	4	5
5	6	8	16	12	15

4th iteration



0	1	2	3	4	5
5	6	8	12	16	15

5th iteration



0	1	2	3	4	5
5	6	8	12	15	16

Date No

```
for (int i=0; i<n-1; i++) {
    int minI = i;
  for (int j = i+1; j<n; j++) {
     if (arr[j] < arr[minI])
         minI = j
  }
  if (minI != i)
         swap (arr[i], arr[minI]);
}
```

① Desk Search

Ex:

0	1	2	3
18	19	20	25

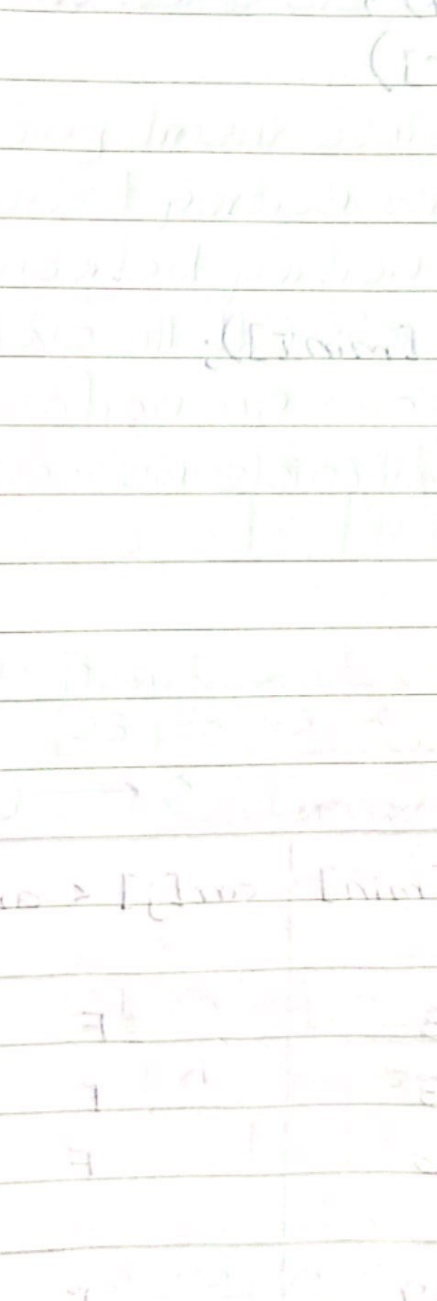
outer loop ↘ inner loop ↙

i	j	minI	arr[j]	arr[min]	arr[j] < arr[min]
0	1	0	19	18	F
	2	0	20	18	F
	3	0	25	18	F
1	2	1	20	19	F
	3	1	25	19	F
2	3	2	25	20	T

Date No

(01.)

0	1	2	3
18	17	20	16



Date No

```
void selectionSort (int array[], int size)
{
   int startScan, minIndex, minValue;
   for (startScan=0; startScan<(size-1); startScan++)
   {
      minIndex = startScan;
      minValue = array[startScan];
      for (int index=startScan+1; index<size; index++)
      {
         if (array[index] < minValue)
         {
            minValue = array[index];
            minIndex = index;
         }
      }
      array[minIndex] = array[startScan];
      array[startScan] = minValue;
   }
}
```

* Benifits of Selection Sort.
  - More efficient than Bubble sort, since fewer exchanges.

* Disadvantages of Selection Sort.
  - May not be as easy as Bubble Sort to understand.

only for partial data sets

Date No

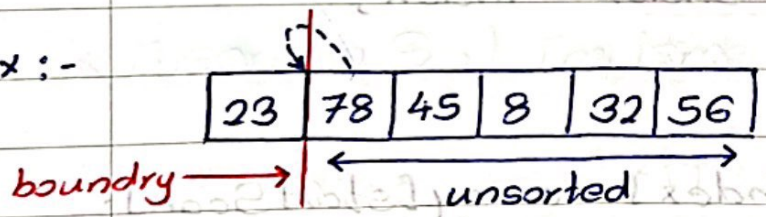
Insertion Sort

While some elements unsorted:

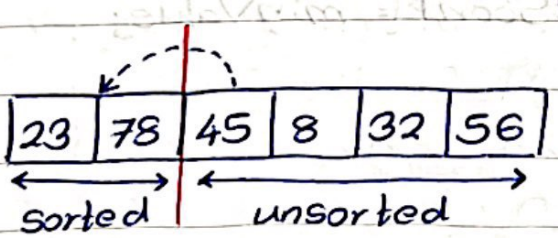
01.) Using linear search, find the location in the sorted portion where the first element of the unsorted portion should be inserted.

02.) Move all the elements after the insertion location up one position to make space for the new element.

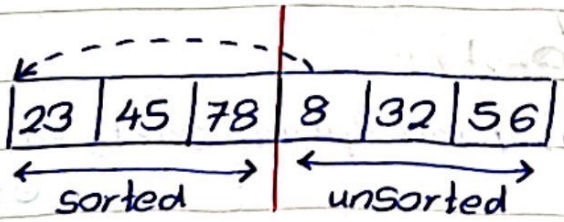
Ex :-



Original list

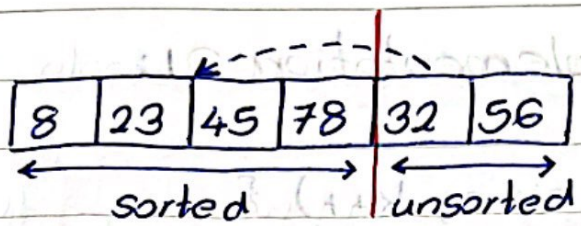


After Pass 1

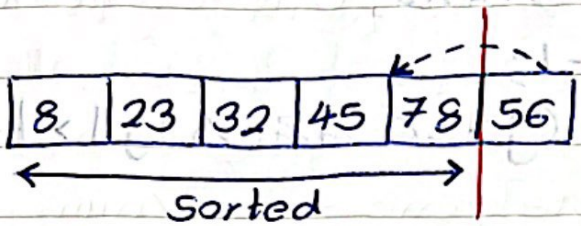


After Pass 2

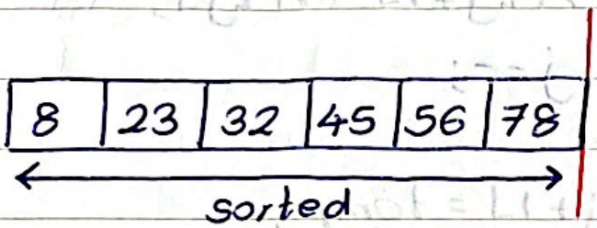
Date No



After Pass 3

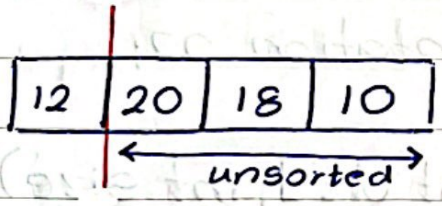


After Pass 4

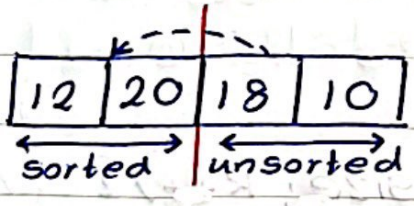


After Pass 5

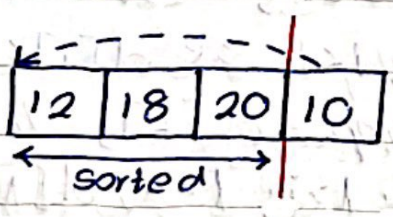
Ex:-



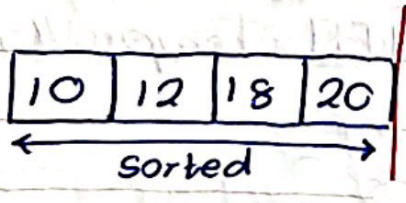
Original list



After Pass 1



After Pass 2



After Pass 3

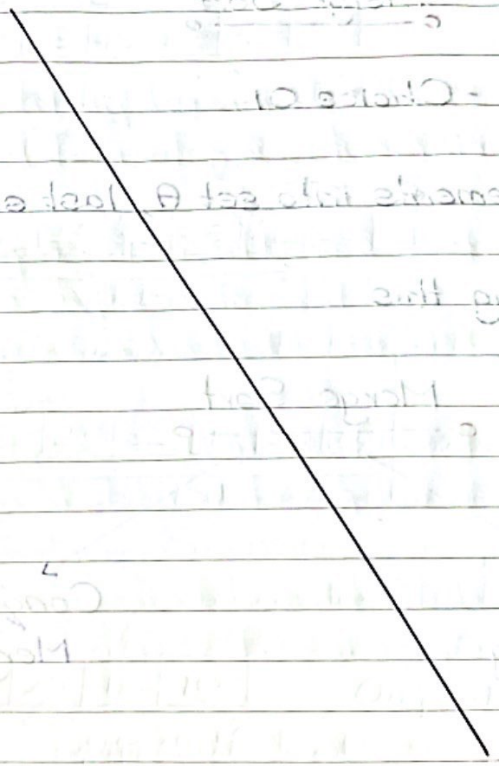
Date No

* Insertion Sort Implementation 01:

```
for (k=1; k<size; k++) {
        temp = A[k];
        j = k-1;
        while (j>=0 && A[j] > temp)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = temp;
}
```

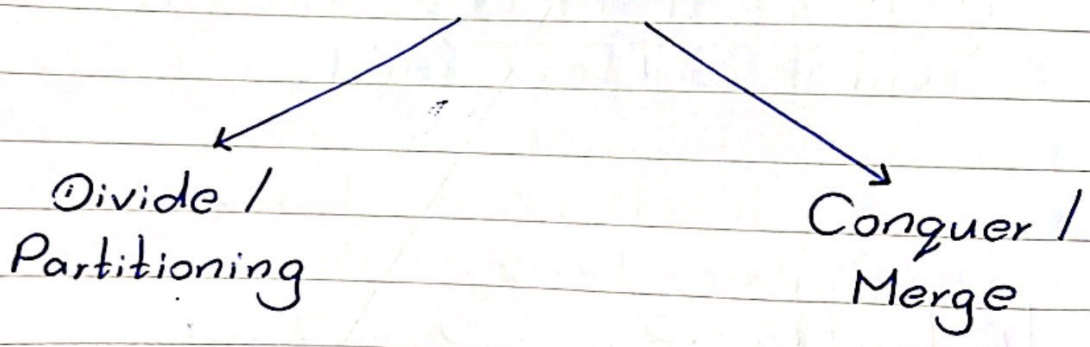
* Insertion Sort Implementation 02:

```
void insort (float d[], int size){
    int i, k, l;
    float temp;
    For (k=1; k<size; k++) {   //insert elements starting from 2nd
        for (i=0; i<k && d[i]<=d[k]; i++);   // find where to insert - 1
            temp = d[k];   // copy kth element to temp
        For (l=k; l>i; l--) d[l] = d[l-1];   // make room to insert at i
            d[i] = temp;   // insert kth element to ith place
    }
}
```



Date No

Merge Sort



Merge Sort algorithm uses divide and conquer programming technique.

Merge Sort execute in 2 phases.

01.) Dividing / Partitioning
(The entire data set is partitioning into single element level)

02.) Conquering / Merging.
(The actual sorting and merging happens here)

Date No

How to divide:

$n/2$, balanced part

* Partitioning - Choice 01

- First n-1 elements into set A, last element set B.
- Sort A using this partitioning scheme recursively
  - B already sorted.
- Combine A and B using method Insert() (=insertion into sorted array)
- Leads to recursive version of InsertionSort()

* Partitioning - Choice 02

- Put elements with largest key in B, remaining elements in A.
- Sort A recursively.
- To combine sorted A and B, append B to sorted A.
  - Use Max() to find largest element → recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position → recursive BubbleSort()

Date No

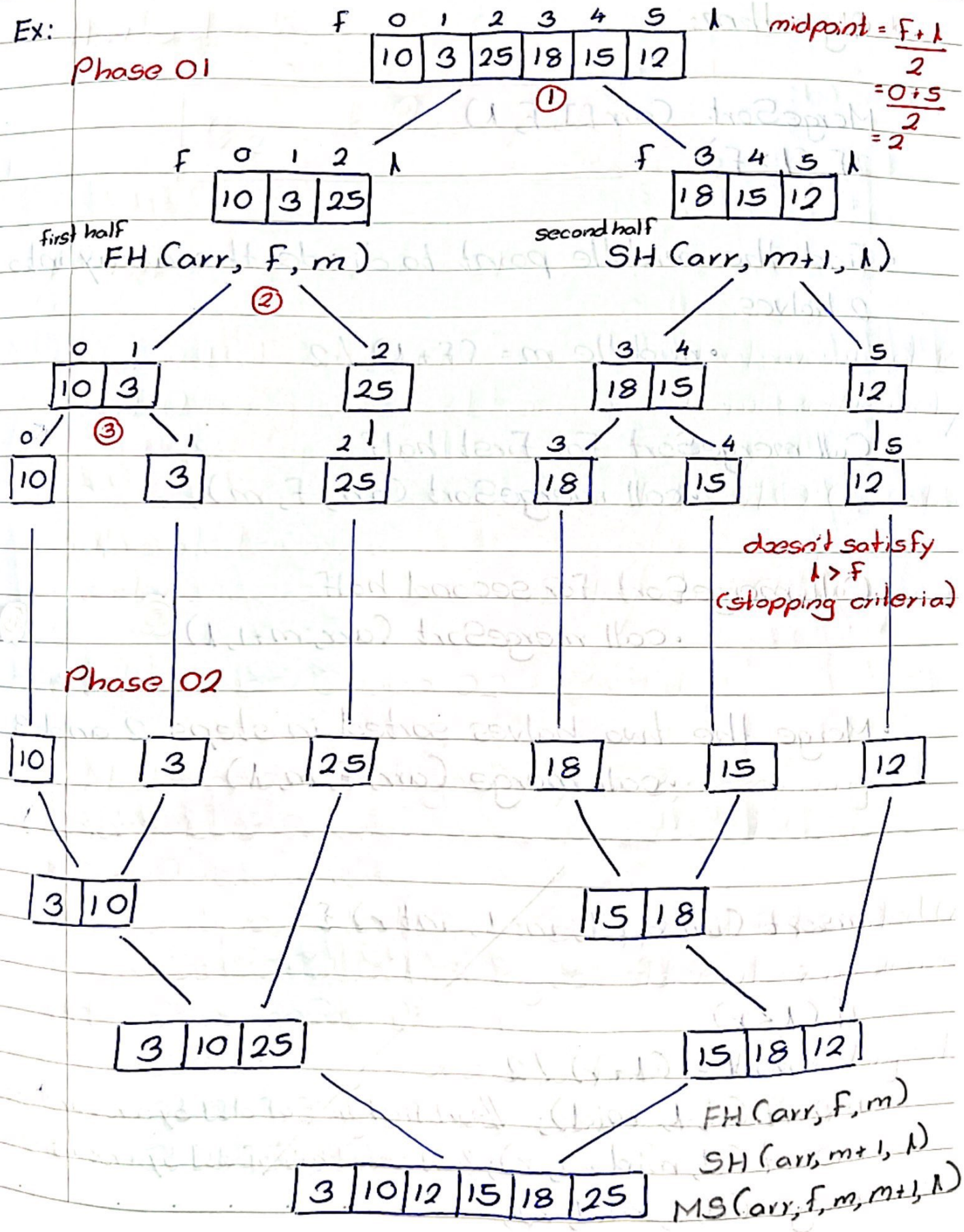
* Partitioning - Choice 03

- Let's try to achieve balanced partitioning.
- A gets $n/2$ elements, B gets rest half.
- Sort A and B recursively.
- Combine sorted A and B using a process called merge, which combines two sorted list into one.

①, ②... → indicates the order in which steps are processed

Date No

Ex:



Date No

* Algorithm:

MergeSort (arr[], f, l)
If (l > f)

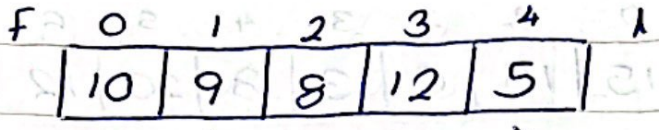
- Find the middle point to divide the array into 2 halves.
  - middle m = (f + l) / 2
- Call mergeSort for First half
  - call mergeSort (arr, f, m)
- Call mergeSort for second half
  - call mergeSort (arr, m+1, l)
- Merge the two halves sorted in steps 2 and 3.
  - call merge (arr, f, m, l)

```
msort (int d[], int l, int r) {

if (l < r)
    mid = (l+r) / 2
msort (d, l, mid); //sort first half of d[] by a recursive ca
msort (d, mid + 1, r); //sort second half of d[] by a recursive ca
merge (d, l, mid, r);
}
```

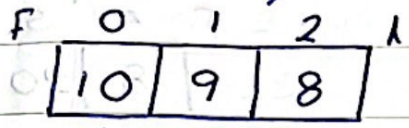
ProMate

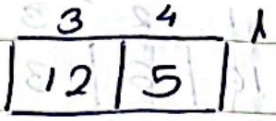
Date No

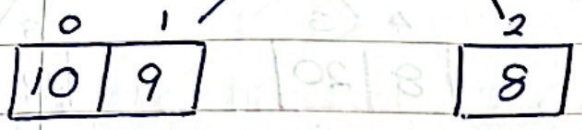
(01.)

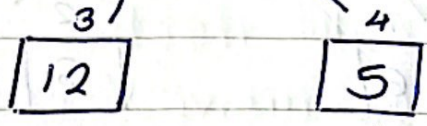


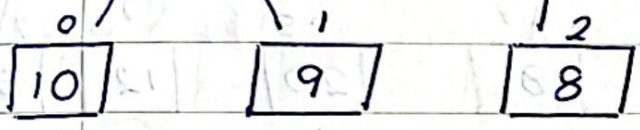
Phase 01:

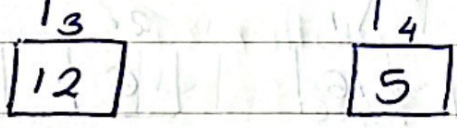




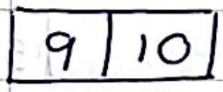


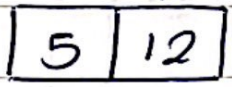


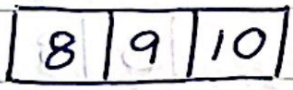


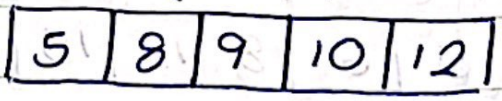


Phase 02:



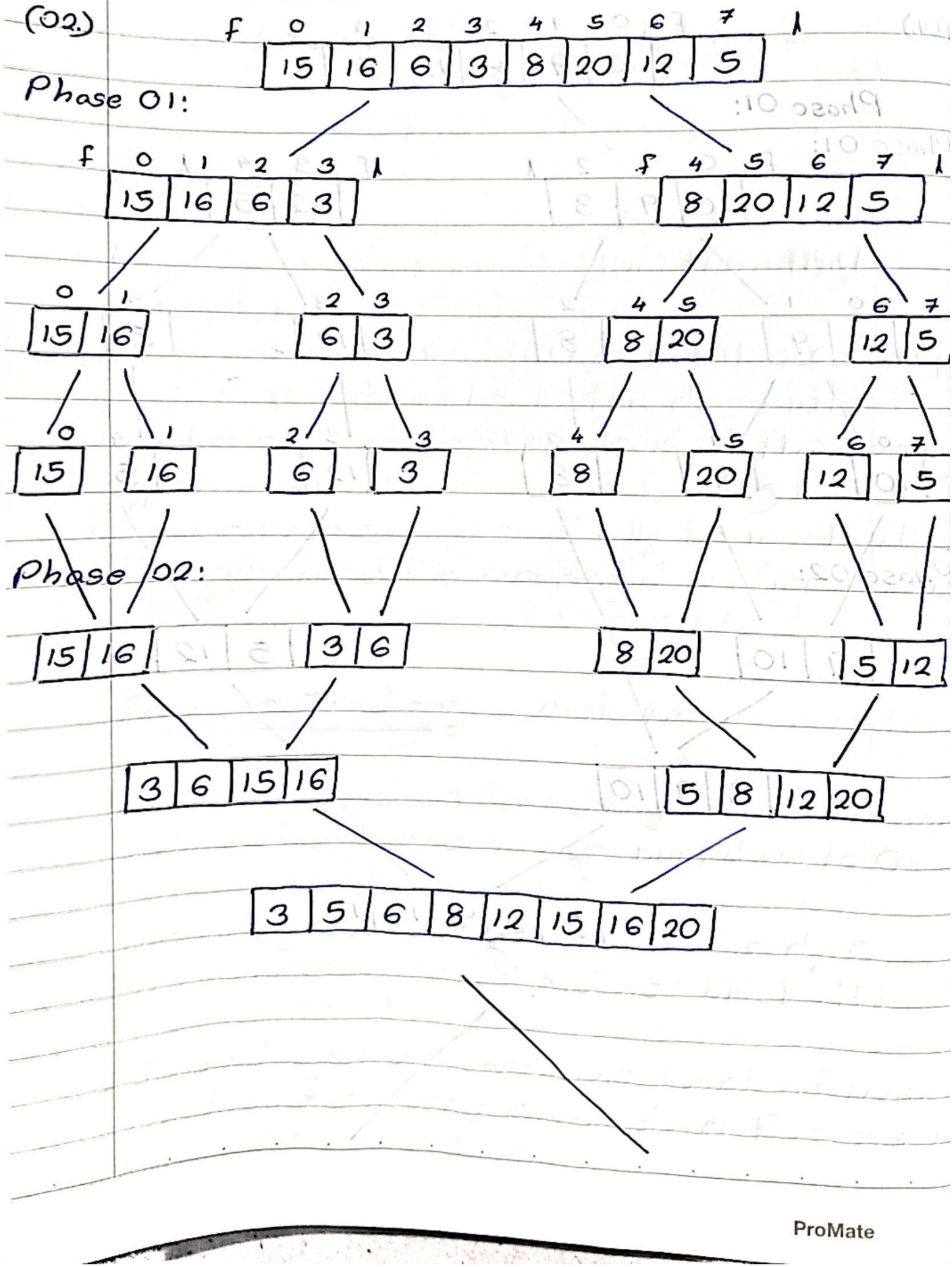






Date No

(02.)



Date No

* Merge Sort Implementation using Recursion:

```
void msort ( int d[], int size) {
int i,j, k, result [size], mid = size/2;
if (size == 1) return;
msort (d, mid);
msort (&d[mid], size - mid);
for (k=0, i=0, j= mid; i<mid && j<size; k++)
result [k] = d [i] < d [j]? d [i++]: d [j++];
while (i<mid) result [k++] = d [i++];
while (j<size) result [k++] = d [j++];
for (k=0; k<size; k++) d[k] = result [k];
}
```

How MergeSort Algorithm Works Internally.

01.) Divide the array into 2 parts.
02.) Divide the array into 2 parts again.
03.) Break each element into single parts.
04.) Sort the elements from smallest to largest.
05.) Merge the divided sorted arrays together.
06.) The array has been sorted.

Date No

Recursion & Recursive Algorithms

Recursion.

* A recursive function is a function that calls itself.
* Recursive Functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type.
* A base case should eventually be reached, at which time the breaking down (recursion) will stop.

Recursive Functions.

Consider a function for solving the count-down problem from some number num down to 0:

* The base case is when num is already 0 : the problem is solved and we "blast off!"
* If num is greater than 0, we count off num and then recursively count down from num-1.

Date No

Base Case / Recursive Call

A recursive function for counting down to 0:

```
void countDown(int num)
{
    if (num == 0)
        print "Done!";
    else
    {
        print num;
        countDown(num - 1); //recursive call
    }
}
```

What happens when called?

If a program contains a line like countDown(2);

01.) countDown(2) generates the output 2..., then it calls countDown(1)
02.) countDown(1) generates the output 1..., then it calls countDown(0)
03.) countDown(0) generates the output "Done!", then returns to countDown(1)
04.) countDown(1) returns to countDown(2).
05.) countDown(2) returns to the calling function.

Date No

Stopping the Recursion.

* A recursive function should include a test for the base cases.

* In the sample program, the test is:

```
if (num == 0)
```

```
void countDown (int num)
{
    if (num == 0)
        print "Done!"; // test
    else
    {
        print n;
        countDown (num-1) //recursive call
    }
}
```

value passed to recursive call is closer to base case of num = 0.

* With each recursive call, the parameter controlling the recursion should move closer to the base case.

* Eventually, the parameter reaches the base case and the chain of recursive calls terminates.

Date No

What Happens When Called?

* Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables being created.
* As each copy finishes executing, it returns to the copy of the function that called it.
* When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function.

Types of Recursion.

01.) Direct Recursion.

* A function calls itself.

02.) Indirect Recursion.

* function A calls function B and B calls function A.

or

* function A calls function B, which calls..., which calls function A.

Date No

The Recursive Factorial Function.

* The factorial of a nonnegative integer n is the product of all positive integers less or equal to n.

* Factorial of n is denoted by n!

* The factorial of 0 is 1

$$0! = 1$$
$$n! = n \times (n-1) \times \ldots \times 2 \times 1 \quad \text{if } n > 0$$

* Factorial of n can be expressed in terms of the factorial of n-1.

$$0! = 1$$
$$n! = n \times (n-1)!$$

* Recursive Function.

```
int factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial (n-1);
```

Date No

* Any given recursion function has 2 components.
  01.) Base case / Stopping condition / Termination point.
  02.) The Recursive Call. (recursive call runs towards the base case.

01.) Write the outputs of following functions.

I)
```
void countDown (int num)        if num = 5
{
    if (num == 0)
        print Done!;
    else                                    5 4 3 2 1 Done!
    {
        print num;
        countDown (num-1);
    }
}
```

II)
```
void countDown (int num)
{
    if (num == 0)                           Done! 1 2 3 4 5
        print Done!;
    else
    {
        countDown (num-1);
        print num;
    }
}
```

ProMate