Date .....

Data Structures And Algorithms

Data structures are the efficient way of storing data. Data structures are the containers for storing data so that operations can be performed on the data stored in the data structure.

Need of Data Structure

With enormous amount of data in today's world, there are three problems that applications face:

1) Data Search: Consider an inventory of 1 million data, if the application is to search a data, it has to search 1 million times slowing down the search. As a result, search will become slower.

2) Processor Speed: Processor speed although being very high, falls limited if data grows to billion records.

3) Multiple requests: As thousand of users search for data simultaneously on web server, even the server fails while searching the data.

Algorithm

Algorithm is a step by step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are created independent of underlying language i.e algorithms can be implemented in more than one programming language.

Date .....

Algorithms are written in english language during Design Phase.
After writing the algorithm, algorithm needs to be analysed to check whether the results can be achieved or not, is the algorithm efficient in terms of time and space complexity.

Attributes / Characteristics of Algorithm

There are five characteristics of algorithms

1) Input
2) Output
3) Definiteness
4) Finiteness
5) Effectiveness

1) Input → Algorithms take input. Some algorithms may or may not take input. Algorithm can take 0 or more input

2) Output → Algorithm must generate atleast 1 output otherwise there is no need of writing algorithm.

3) Definiteness → Every statement of algorithm must be unambiguous, it should have a meaning and should be clear.

4) Finiteness → Algorithm must have finite set of statements.

5) Effectiveness → Algorithms should not have unnecessary statements, every statement written should contribute towards output.

Date .....

Algorithm Design Techniques

There are seven Algorithm Design Techniques.

1) Brute Force Technique
2) Divide and Conquer
3) Greedy Method
4) Dynamic Programming
5) Divide And Conquer
6) [illegible]
7) BackTracking

1) Brute Force Techniques

A brute force technique finds all possible solutions to find satisfactory solution to given problem. The brute force technique tries out all possibilities till a satisfactory solution is not found.

Example:- Consider a 3 digit padlock, and you have to open the padlock, you have to try all 3 digit possible combination to unlock it, ie 1000 combinations.

The time complexity of brute force is O(mn) which can also be written as O(m*n) ~~where~~ which means that if we need to search of n characters in a string of m characters, then it would take n*m tries.

Example:- Sequential search, String matching algorithm, Traveling salesman problem, Knapsack problem etc.

Date .....

2) Divide And Conquer Technique

Divide And Conquer Technique works on top down approach and is preferred for largeer problems.
It has following steps.

1) Divide the problem into several subproblems.
2) Conquer the problem ie solve the problem
3) Combine the solution of each subproblem to find the solution of larger problem.

3) Greedy Method

A greedy method is an approach for solving a problem by selecting the best option available at the moment. It works in a top down approach.
This algorithm may not produce the best results for all the problems

4) Dynamic Programming

Dynamic Programming is mainly an optimisation problem for recursion. Whenever we see a recursive function we can optimise it using Dynamic Programming.
The idea is simply to store the results of subproblems so that we do not have to recompute them again and again whenever there is a function call. It is a bottom up approach.

5) Back Tracking

Back Tracking means if the current solution isnot working you should go back and attempt another option. This approach is used

Speral

Date .....

to resolve problems having multiple solutions.

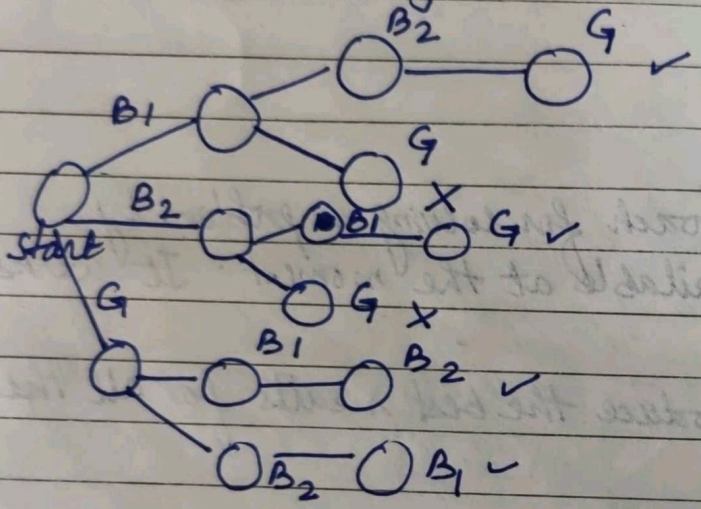
For ex:- If you want to arrange 2 boys and 1 girl on 3 benches such that girl should not be sitting in the middle bench.

B_1	B_2	G
G	B_1	B_2
B_2	B_1	G
G	B_2	B_1

} Possible ways

Actual ways

B_1	B_2	G ✓
G	B_1	B_2 ✓
B_1	G	B_2 ✗
B_2	G	B_1 ✗
G	B_2	B_1
B_2	B_1	G ✓



Time Complexity

There are two criterias used to analyse the performance of algorithm:-

1. Time complexity
2. Space complexity

Time Complexity

It is the amount of time needed by CPU to execute the program's instructions.

Date .....

Space Complexity

It is the amount of memory needed by algorithm to run till its completion.

The time taken for an algorithm is compromised of two times:

1. Compilation Time
2. Run Time

Compilation Time

It is the time taken to compile an algorithm. During compilation it checks for syntax and semantic errors and links the function calls with its standard libraries.

Run Time

It is the time taken to execute the compiled program. The run time of an algorithm depends upon the no. of instructions present in the program.

*Run Time is calculated only for executable statements and not for declarative statements.

Date .....

Time complexity of an algorithm is generally categorised of three types. These are also called as Asymptotic analysis or assymptotic notations.

1. Worst Case (Longest Time)
2. Average case (Average Time)
3. Best Case (Shorter Time)

In Technical terms worst case is known as Big Oh Notation (Upper bound) Theta
Average case is known as ~~Theta~~ ~~Omega~~ Notation (Lower bound) Omega
Best case is known as ~~Omega~~ notation (tighter bound)

Time complexity of an algorithm is measured using these standard analysis techniques.

1. Analysing Loops
2. Constant time statements
3. Analysing nested Loops
4. Analysing sequence of statements
5. Analysing conditional statements

Simple Example to calculate Time complexity

```
int sum(int A[], int n)
{
  int s = 0;  -(1)
  for (int i = 0; i<n; i++)
  { (2)        (3)      (4)
    s = s + A[i];
  }  (5) (6)  (7)
  return s;
}  (8)
```

A = {1, 2, 3}

i = 0
s = 0 + 1
s = 1

i = 1
s = 1 + 2
s = 3

i = 2
s = 3 + 3

Spiral : 6

Date .....

Statement ①, ② and ⑥ are constant statements and will be executed only once.
statement ③, ④, ⑤, ⑥ and ⑦ they will be executed once for each iteration of the loop. There are total N iterations.
Hence time complexity of the function is: $f(N) = \boxed{5N+3}$.

Space Complexity of an Algorithm

Space complexity of a program or algorithm is the amount of memory needed by the algorithm until it completes its execution.

The space occupied by the program is generally categorised into following:-

① A fixed amount of memory is occupied by the Data types associated with variables.

② Space occupied by variables used in the program.
③ The space increases or decreases depending upon whether the program uses iterative or recursive procedures.

Example of Space complexity

```
void f()
{
int z = a+b+c;
return z;
}
```

a will take 4 bytes of memory, b will take 4 bytes, c will take 4 bytes. 12 bytes of memory required. The sum is stored in z which will again occupy 4 bytes of

Date .....

memory ie in total 16 bytes of memory. After that z is returned which will take 4 bytes of memory in environment space. Hence total of 20 bytes of memory is required.

```
int sum (int a[], int n)
{
 int x=0;
for(int i=0; i<n; i++)
  {
   x = x+a[i];
   }
  return x;
   }
```

n, x and i each will take 4 bytes of memory and for loop will be executed N times where each variable is integer type. So it will take total 4N memory. Total memory occupied will be 12+4N bytes.

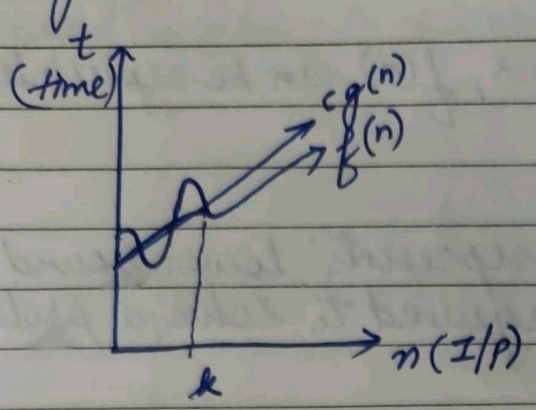
Asymptotic Notations

Asymptotic Notation is the mathematical way of representing time complexity of an algorithm.

1) Big Oh (O)
2) Big Omega (Ω)
3) Theta (Θ)

Date .....

(1) Big-Oh (O) Notation → It represents upper bound value (almost)



$f(n)$ is the time taken to solve the problem.

Here $f(n)$ is represented as

$$f(n) = O(g(n))$$

where $f(n) \leq c\,g(n)$

c is the constant time which is greater than 0.

$c > 0$ and $n \geq k$ and $k \geq 0$

Big Oh notation is used to calculate the maximum amount of time taken by algorithm.

For ex:- Time taken to solve a particular problem is

$f(n) = 2n^2 + n$

$f(n) = O(?)$

$2n^2 + n \leq c\,g(?)$

→ This n should be greater than equal to n^2

$2n^2 + n \leq c\,g(n^2)$

Now we have to decide the value of c

therefore c's min value is $c = 3$

$2n^2 + n \leq 3n^2$

On solving

$n \leq n^2$

Dividing both sides we get,

$1 \leq n$ ie $\boxed{n \geq 1}$

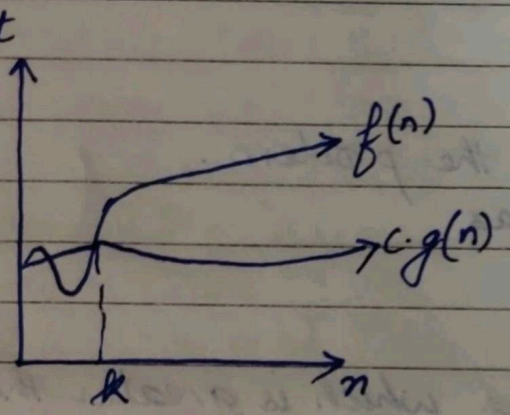
Spiral

Date .....

It means that for all values of input $n \geq 1$ and $c = 3$, $f(n) \leq c\,g(n)$.

For all values of $n \geq 1$ and $c = 3$, $f(n)$ can be represented as $f(n) = O(n^2)$

(2) Big Omega Notation (Ω) → It represents lower bound value ie minimum time required to solve a problem



$f(n)$ is the time taken to solve the problem.

Here $f(n)$ is represented as

$f(n) = \Omega\, g(n)$

where $f(n) \geq c\,g(n)$

Let $f(n) = 2n^2 + n$

$2n^2 + n \geq c(?)$

$2n^2 + n \geq cn^2$

Take $c = 1$

$2n^2 + n \geq n^2$

$n \geq -n^2$

$n \leq n^2$

$1 \leq n$

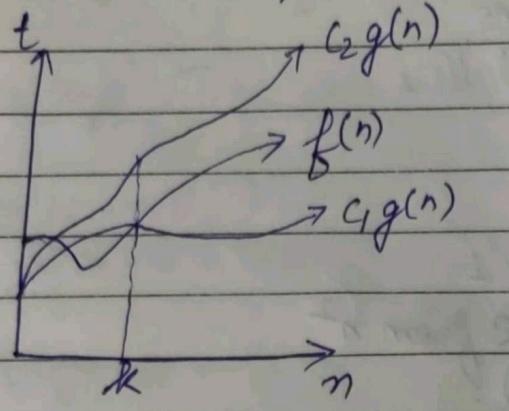
It means that for all values of $n \geq 1$, and $c = 1$

$f(n) \geq c\,g(n)$

For all values of $n \geq 1$ and $c = 1$, $f(n)$ can be represented as $f(n) = \Omega(n^2)$

Date .....

(3) Theta Notation (Θ) → It represents average case time required to solve a problem.



$f(n)$ is the time taken to solve a problem.

Here $f(n)$ is represented as

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

Let $f(n) = 2n^2 + n$

$c_1 n^2 \leq 2n^2 + n \leq c_2 n^2$

Take $c_1 = 2$ and $c_2 = 3$

$2n^2 \leq 2n^2 + n \leq 3n^2$

$2n^2 \leq 2n^2 + n$	$2n^2 + n \leq 3n^2$
$0 \leq n$	$n \leq n^2$
$n \geq 0$	$1 \leq n$
	$n \geq 1$

It means that for all $n \geq 0$ and $c_1 = 2$, $f(n) \geq c_1 g(n)$ and for all $n \geq 1$ and $c_2 = 3$, $f(n) \leq c_2 g(n)$.

Date .....

Rules for calculating Time complexity of an algorithm

(1) Drop the lower order terms
(2) Drop the coefficients multipliers.

For ex: $T(n) = 2n^2 + 3n + 1$

We will drop $3n+1$

We will drop 2 from n^2

So we get $T(n) = O(n^2)$

(1) Time complexity of Loop

```
for(i=1; i<=n; i++)
  {
    x = y+z; → This statement once executed will take
  }          O(1) time.
```

But this statement will be executed n times inside the loop.

So Time complexity would be $O(n)$.

(2) Time complexity of nested loops

```
for(i=1; i<=n; i++) → n times
  {
    for(j=1; j<=n; j++) → n times
      {
        x = y+z; → This statements once executed will
      }            take O(1) time.
  }
```

But this statement is written inside nested for loop. For each iteration of outer loop

Date .....

inner loop will also be executed n times.
So Time complexity of the statement would be $O(n^2)$

(3) Time complexity of Sequential statements

```
a=a+b → O(1)
for(i=1; i<=n; i++)        } O(n)
{
  x=y+z;
}
for(j=1; j<=n; j++)        } O(n)
{
  c=d+e;
}
```

For sequential statements we add all the time complexities to get the final time complexity.

Time complexity would be $O(1)+O(n)+O(n)$
$= O(n)$

4) If else statements time complexity

```
if (condition)
{
  — O(n)
}
else
{ — O(n^2)
}
```

We will consider the time complexity as $O(n^2)$ as n^2 is higher order term **Spiral**

Date .....

Comparison of Time complexities

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n!)$$

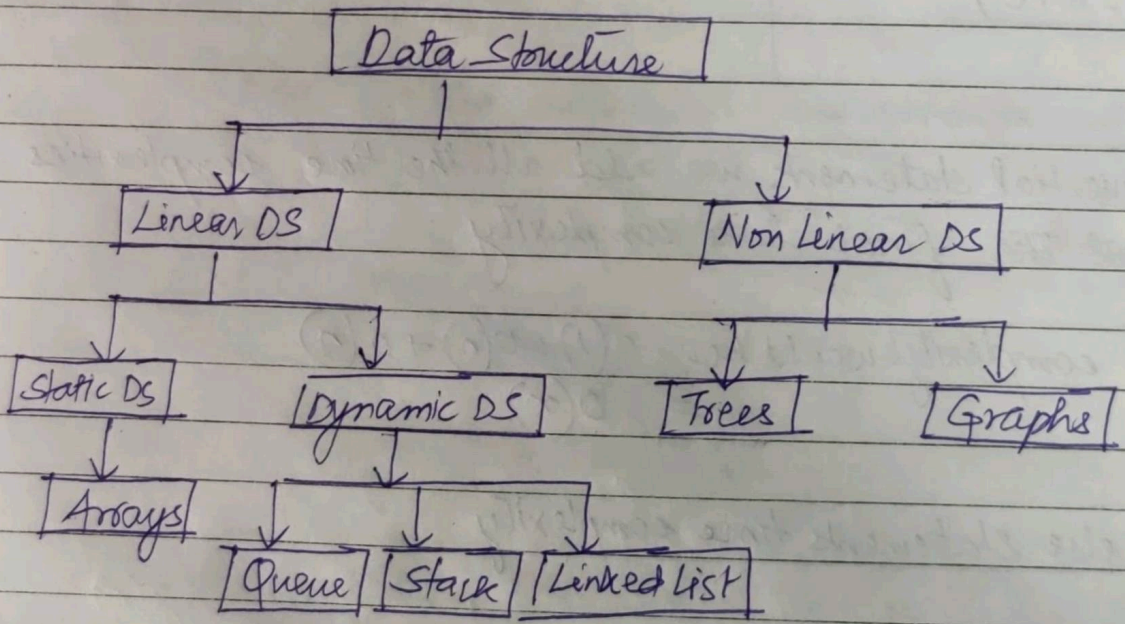
$O(1)$ ↓ min time

$O(n!)$ ↓ max time

* Space efficiency and Time efficiency are reverse to each other.

An algorithm which executes faster will take a lot of memory space and vice versa.

Classification of Data Structures



Linear Data Structure → Data Structure in which elements are stored sequentially or linearly. Ex:→ Array, stack, queue, linked list.

Static Data Structure → Static data structure has fixed memory size. It is easier to access elements in static data structure. Ex: Array.

Date .....

Dynamic Data Structure → In Dynamic Data structure, the size is not fixed. It can be randomly updated during run time.
Ex: Queue, Stack, Linked List

Non Linear Data Structure → Data structure where data elements are not placed sequentially or linearly are called non linear DS. Ex: Trees & Graphs.

Various Operations that can be performed on Data Structure

① Traversing: Traversing a Data structure means visiting each element of the DS. We can do traversal on array, queue, stack, linkedlist.

② Searching: Searching means to find a particular element in the given DS. It is considered as successful when the required element is found. Searching can be performed on Arrays, linkedlist, trees, graphs etc.

③ Insertion :- Insertion means to add an element in the DS. The insertion operation is successful, when the element is added to the required DS. It is unsuccessful, when the size of the DS is full and there is no more space.

④ Deletion → Deletion means to delete an element from the given DS. Deletion is successful when the required element is deleted from DS. It is unsuccessful, when the DS is empty.

⑤ Create → It reserves memory for DS by declaring them. The creation of DS can be done in 2 ways:

Spiral

Date .....

(1) Compile Time
(2) Run Time

(6) Selection → It selects specific data from DS.

(7) Update → It updates the data in DS.

(8) Sort → Sorting data in particular order (Ascending or Desc)

(9) Merge → Merging data of two different orders in a specific order either ascending or descending.

(10) Split Data → Dividing data into two different sub parts.

Date .....

Arrays

An array is a collection of elements of similar data type. The size of the array needs to be declared beforehand and it cannot be changed during run time.

Ex: `int a[60];` — 60: size of array → Array Declaration

int: datatype; a: array name

There are three types of Array

1. One Dimensional Array (1D Array)
2. Two Dimensional Array (2D Array)
3. MultiDimensional Array

Elements of array are stored in contiguous memory locations i.e one after another. Indexing would always be starting from 0.

Array Initialisation

Arrays can be initialised at two times:-

1. Compile Time
2. Run Time

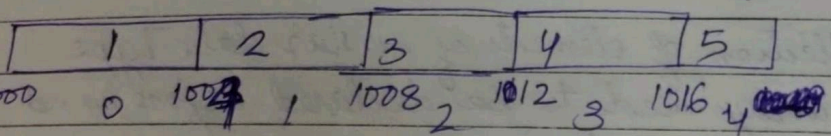
`int a[5] = {1,2,3,4,5};` → Initialisation at compile time

You can specify less than 5 elements but not more than 5 elements.

These elements will be stored in contiguous memory locations and will occupy 20 bytes in memory since

Date .....

each element will occupy 4 bytes of memory.



How to calculate addres of element

Address = B + i x sizeof data Type

For ex i = 2

1000 + 2 x 4

1000 + 8

Address = 1008

You can access array elements randomly.

Time taken to access array elements is O(1).

Drawback of array is that there could be wastage of space because size is declared before compile time so if less elements are inserted, remaining memory locations will get wasted

Array initialisation at runtime

```
int a[5];
printf("Enter array elements");
for(int i=0; i<n; i++)
  {
  scanf("%d", &a[i]);
  }
```

To access Array element :→ arrayname[index]

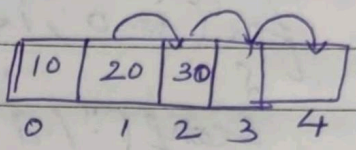
Date .....

Operations on Array

① Insertion of an element in an Array

To insert at ~~index~~ specific position

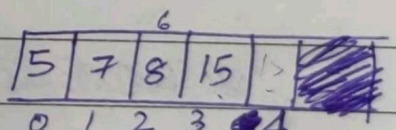
```
int a[50], size, num, pos;
printf("Enter array size");
scanf("%d", &size);
if(size>50)
{
printf("Overflow");
}
else
{printf("Enter array elements");
for(int i=0; i<size; i++)
{
scanf("%d", &a[i]);
}
printf("Enter element you want to insert");
scanf("%d", &num);
printf("Enter the position");
scanf("%d", &pos);
for(int i=size-1; i>=pos-1; i--)
{
a[i] = a[i-1];
}
a[pos-1] = num;
}
```



size = 5

To insert 15 at index 1 ie 2nd position

a[4] = a[3]
a[3] = a[2]
a[2] = a[1]



size = 5
pos = 3
ie i = 2

Date .....

(2) Deletion of an element from an array.

```
int a[50], size, item, pos;
printf("Enter array size");
scanf("%d", &size);
printf("Enter array elements");
for(i=0; i<n; i++)
{
scanf("%d", &a[i]);
}
printf("Enter pos of element");
scanf("%d", &pos);
for(i=pos; i<size; i++)
{
a[i] = a[i+1];
}
```

10	20	30	50	40
0	1	2	3	4

To del: pos = 2 ie i = 1

a[1] = a[2]

a[2] = a[3]

Two Dimensional Arrays

A two dimensional Array can be represented in the form of rows and columns.

Declaration :- `int a[3][3];`

(3 → row, 3 → column)

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}_{3\times3}$$

2D Array is an array of arrays.

Initialisation of 2D Array can also be done at compile time and run time.

```
int a[2][3] = {1,2,3,4,5,6}
```

↗ Compile time

i →	j → 0	1	2
0	1	2	3
1	4	5	6

```
int a[2][3] = { {1,2,3}, {4,5,6} };
int a[ ][3];  → valid
int a[2][ ];  → invalid
```

At run time array initialisation

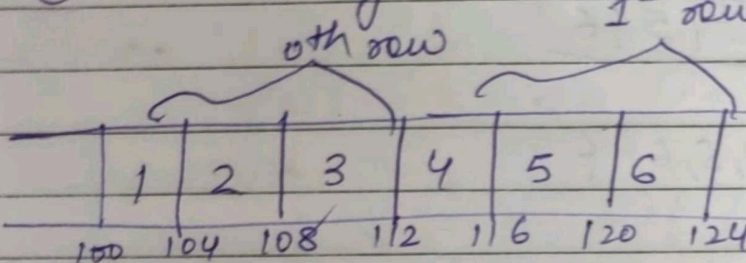
```
int a[2][3];
printf("Enter the array elements");
for(int i=0; i<2; i++)
  {
   for(j=0; j<3; j++)
    {
     scanf("%d", &a[i][j]);
    }
  }
For printing
for(i=0; i<2; i++)
  {
   for(j=0; j<3; j++)
    {
     printf("%d", a[i][j]);
    }
  }
```

To access array element: array_name[row no][col no]

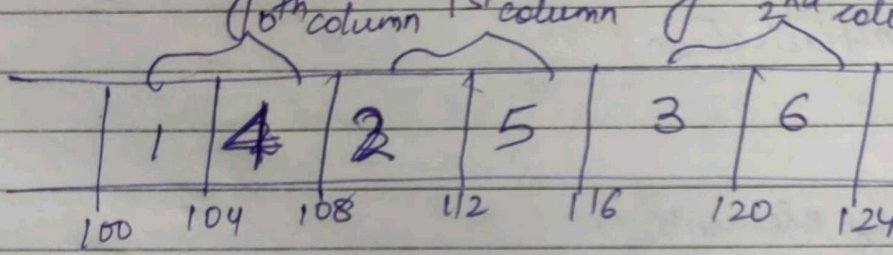
Space

How 2D Arrays are stored in memory

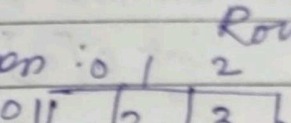
There are two ways of memory representation :
1. Row Major order
2. Column Major order



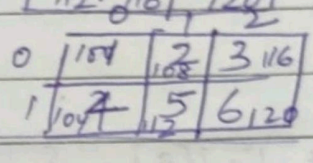
storing elements according to row major order



Row Major



Column Major



The address of the first element is known as base address.

To find out the address of a[1][1] ie 5 using Row Major Order
Here i = 1, j = 1, B = 100, size = 4 bytes.

Address of a[1][1] = B + ((i × n) + j) * size
= 100 + (1 * 3 + 1) * 4
= 100 + 4 * 4
= 100 + 16
= 116

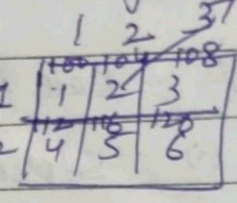
To find out the address of a[1][1] using column Major.

Address of a[1][1] = B + ((j × m) + i) * size
= 100 + ((1 * 2) + 1) * 4
= 100 + 3 * 4
= 100 + 12
= 112

Spiral

Date .....

If index is starting from 1 modified formula for row major and column major would be:-



Row Major

Address of a[i][j] = B + w * ((i-1) * n + (j-1))

Column Major

Address of a[i][j] = B + w * ((j-1) * m + (i-1))

Ques:- Consider the linear array A[16:30]. If base address is 100 and space required to store each element is 4 words, then find the address of A[27].

Sol:- B = 100, w = 4

Given array: A[16:30]

lb = 16, ub = 30

To find address of A[27].

Address = B + w(i - lb)

= 100 + 4(27 - 16)

= 100 + 4 X 11

= 100 + 44

Address = 144.

Date .....

Address calculation of 2D Array

B = Base Address
W = Storage size of an element
L_r = Lower bound of row
L_c = Lower bound of column
m = no. of rows
n = no. of columns
i = row subscript of element
j = column subscript of element

Row Major Order Formula

Address of $A[i][j] = B + W(n(i - l_r) + (j - l_c))$

Column Major Order Formula

Address of $A[i][j] = B + W(m(j - l_c) + (i - l_r))$

Ques:- An array M[1:10, 1:10] required 2 bytes of the storage for each element and the beginning location is 500. so determine the location of M[5][6] Use Row Major & Column Major.

Row Major

Sol:- B = 500, W = 2

$l_r = 1, l_c = 1, i = 5, j = 6$

Address of $M[5][6] = B + W[n(i - l_r) + (j - l_c)]$

$= 500 + 2[10(5-1) + (6-1)]$

$= 500 + 2[10 \times 4 + 5]$

$= 500 + 2[40 + 5]$

$= 500 + 90 = 590$

Speral

Date .....

Column Major

Addres of $M[5][6] = B + W(m(j - l_c) + (i - l_r))$

$= 500 + 2(10(6-1) + (5-1))$

$= 500 + 2(10 \times 5 + 4)$

$= 500 + 2(54)$

$= 500 + 108$

$= 608$

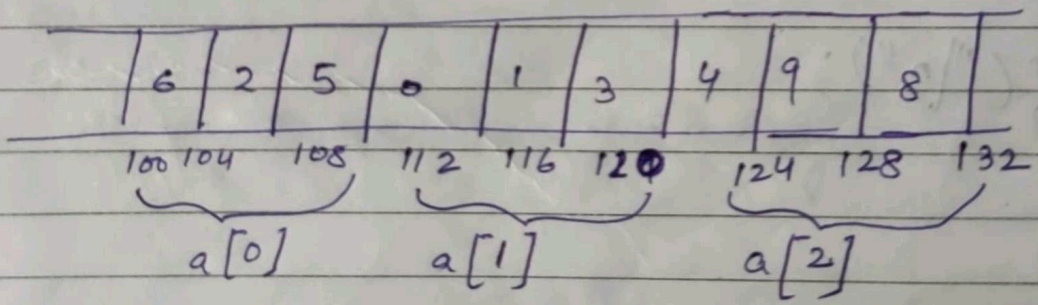
~~Multidimensional Array~~

Pointers and 2D Arrays

int a[3][3] = {6, 2, 5, 0, 1, 3, 4, 9, 8}

	0	1	2
0	6	2	5
1	0	1	3
2	4	9	8

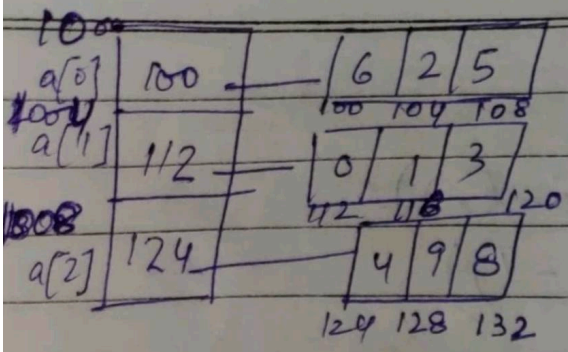
3x3



100	a[0]
112	a[1]
124	a[2]

a[0] will contain ~~first~~ first three integer values and so on

Date .....



int a[3][3] = {6, 2, 5, 0, 1, 3, 4, 9, 8}

int i, j;

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

printf("%d", a[i][j]);

printf("%d", *(*(a+i)+j))

*(1000)

*(100+0)

*100

= 6

}

printf("\n");

}

Date .....

MultiDimensional Array

Declaration :- datatype name of array [size1][size2] ... [sizeN]

Ex :- int A[2][3][3]

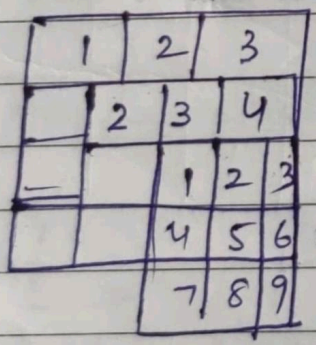
Total elements that can be accomodated = 2×3×3
= 18 elements

MultiDimensional Array can either be :-

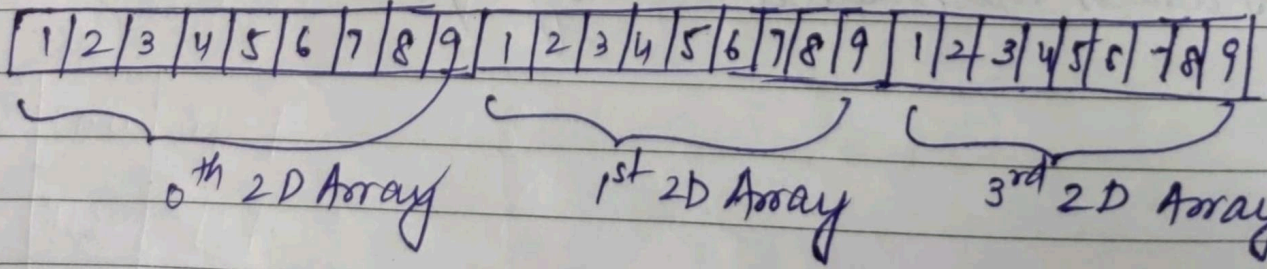
1. 2D Array
2. 3D Array

3D Array

It is array of 2D arrays.



Memory Representation of 3D Array



Spiral

Date .....

Initialisation of 3D Array

```
int a[2][3][4] = {{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
                  {{7,8,9,10},{11,12,13,14},{15,16,17,18}}};
```

↓ we will create 2 2D Arrays of size 3×4

Declaration

```
data type array_name[table][row][column];
```

Sparse Matrix

Sparse matrix is a matrix which contains maximum no of zero elements.

Ex:- $A = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ → Sparse

A matrix having less no. of zero elements is called dense matrix.

Ex: $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 1 & 0 \end{bmatrix}$ → Dense

Sparse matrix is used for reducing time to access non zero elements and for reducing memory space.

Using sparse matrix we will store only non zero elements which will reduce memory space.

Date .....

Memory Representation of Sparse Matrix

1. Array Representation (Triplet)
2. Linked Representation

In both types of representation we will store only non zero elements.

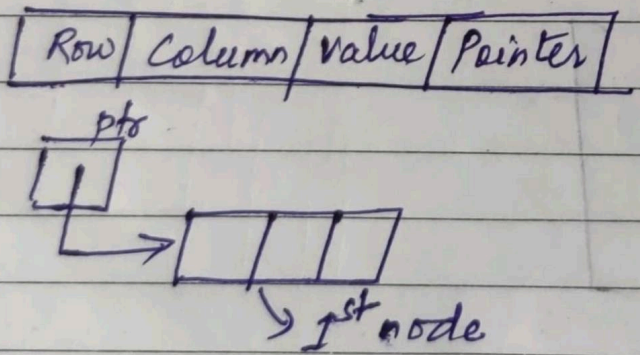
Array Representation

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 3 \\ 3 & 0 & 0 \end{bmatrix} \quad (r_0, r_1, r_2 \text{ rows}; c_0, c_1, c_2 \text{ columns})$$

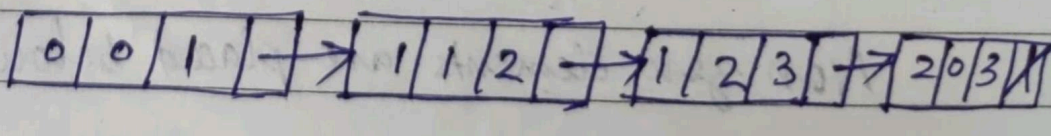
Row	Col	Value (Total no. of non zero values)
3	3	4
0	0	1
1	1	2
1	2	3
2	0	3

We will write the row and column entry and value of non zero element.

Linked Representation



$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 3 \\ 3 & 0 & 0 \end{bmatrix} \Rightarrow$$



Types of sparse Matrix

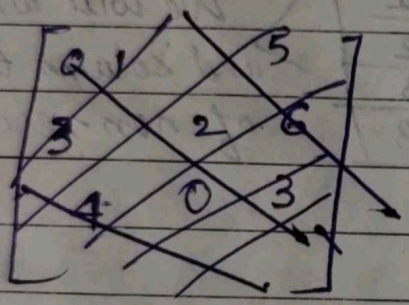
① Diagonal Matrix

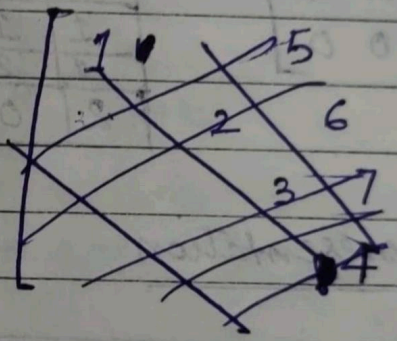
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$ → Only Diagonal contains non zero elements

② Tri Diagonal Matrix

Non zero elements are placed on, or above, or below diagonal.

Ex





$$\begin{bmatrix} 1 & 8 & 0 & 0 \\ 5 & 2 & 9 & 0 \\ 0 & 6 & 3 & 10 \\ 0 & 0 & 7 & 4 \end{bmatrix}$$

③ Lower Triangular Matrix

Non zero elements are placed below the main diagonal.

Ex. $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 3 & 5 & 3 & 0 \\ 4 & 6 & 7 & 4 \end{bmatrix}$$

Date .....

(4) Upper Triangular Matrix

Non zero elements are placed above the main diagonal.

Ex:

$$\begin{bmatrix} 1 & 5 & 8 & 6 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Date .....

Sorting

Sorting in Data Structure means arrangement of elements either in ~~[illegible]~~ ascending order or descending order.

① Bubble Sort

Bubble sort is the simplest algorithm that works by swapping the adjacent elements if they are in wrong order. This algorithm is not suitable for large data sets as its time complexity is very high.

Ex:- a[] = | 2 | 1 | 0 | 3 |
0 1 2 3

There will be required (n-1) passes to sort the elements.
Here n = 4, so total passes = 3

Pass 1:-

| 2 | 1 | 0 | 3 |

2 ↓ 2 3 ∵ 1 < 2

| 1 | 2 | 0 | 3 |

| 1 | 0 | 2 | 3 | ∵ 0 < 2

| 1 | 0 | 2 | 3 | No swapping

Pass 2:-

| 1 | 0 | 2 | 3 |

| 0 | 1 | 2 | 3 | No swapping

| 0 | 1 | 2 | 3 |

No swapping

| 0 | 1 | 2 | 3 |

Date .....

Pass 3: | 0 | 1 | 2 | 3 | No swapping

| 0 | 1 | 2 | 3 | No swapping

| 0 | 1 | 2 | 3 | No swapping

| 0 | 1 | 2 | 3 |

Hence final sorted array will be: | 0 | 1 | 2 | 3 |
0 1 2 3

Ex 2 :- | 6 | 3 | 0 | 5 |
0 1 2 3 $3 < 6$

Pass 1

| 3 | 6 | 0 | 5 |
0 1 2 3 $0 < 6$

| 3 | 0 | 6 | 5 |

| 3 | 0 | 5 | 6 |

Pass 2 : | 3 | 0 | 5 | 6 |

$0 < 3$

| 0 | 3 | 5 | 6 |

| 0 | 3 | 5 | 6 |

| 0 | 3 | 5 | 6 |

Pass 3: | 0 | 3 | 5 | 6 |

| 0 | 3 | 5 | 6 |

| 0 | 3 | 5 | 6 |

| 0 | 3 | 5 | 6 |

Spiral

Date .....

In bubble sort ~~we sort minimize~~ the no of comparisons in each pass is minimised.

Ex:- a[] = | 3 | 0 | 5 | 6 |
0 1 2 3

Pass 1:- | 3 | 0 | 5 | 6 |

| 0 | 3 | 5 | 6 |

0 < 3

| 0 | 3 | | |

Ex:- a[] = | 64 | 34 | 25 | 12 | 22 | 11 | 90 |
0 1 2 3 4 5 6 n = 7 Total passes: 6

Pass 1:- | 64 | 34 | 25 | 12 | 22 | 11 | 90 |

| 34 | 64 | 25 | 12 | 22 | 11 | 90 |

| 34 | 25 | 64 | 12 | 22 | 11 | 90 |

| 34 | 25 | 12 | 64 | 22 | 11 | 90 |

| 34 | 25 | 12 | 22 | 64 | 11 | 90 |

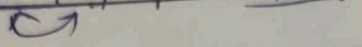
| 34 | 25 | 12 | 22 | 11 | 64 | 90 | NO swapping

| 34 | 25 | 12 | 22 | 11 | 64 | 90 |

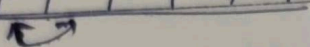
Date .....

Pass 2:- 34 25 12 22 11 64 90

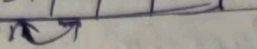
25 34 12 22 11 64 90



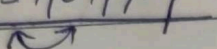
25 12 34 22 11 64 90



25 12 22 34 11 64 90



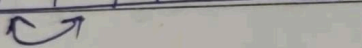
25 12 22 11 34 64 90



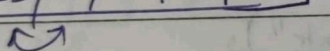
25 12 22 11 34 64 90

Pass 3: 25 12 22 11 34 64 90

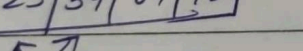
12 25 22 11 34 64 90



12 22 25 11 34 64 90



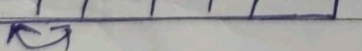
12 22 11 25 34 64 90



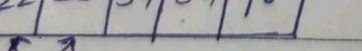
12 22 11 25 34 64 90

Pass 4:- 12 22 11 25 34 64 90

12 22 11 25 34 64 90



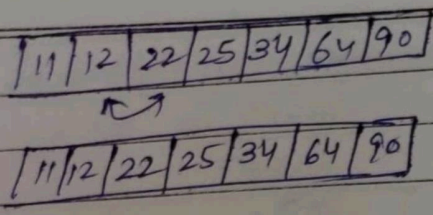
12 11 22 25 34 64 90



12 11 22 25 34 64 90

~~In Pass 5, no~~ Pass 5: 12 11 22 25 34 64 90

Spiral



In pass 6 ~~~~ our array is already sorted.

In bubble sort, sometimes array gets sorted in less no. of passes. so we can optimise bubble sort algorithm

Bubble sort Algorithm (Unoptimised)

```
void main()
{
int a[10];
int n, temp;
printf("\n Enter array size");
scanf("%d", &n);
  printf("\n Enter array elements");
for(int i=0; i<n; i++);
  {
  for(int j=0; j<(n-1)-i; j++)
    {
      if( a[j] > a[j+1])
        {
          temp = a[j];
          a[j] = a[j+1];
          a[j+1] = temp;
        }
    }
printf("\n Sorted Array is");
```

Spiral

```
for(int i=0; i<n; i++)
{
printf("%d", a[i]);
}
}
```

Ex:- a[ ] = | 4 | 3 | 2 | 1 |
0 1 2 3

For i = 0, j = 0

Pass 1:- a[0] > a[1] temp = 4
a[0] = 3 a[1] = 4

| 3 | 4 | 2 | 1 |
0 1 2 3

j = 1

a[1] > a[2]

| 3 | 2 | 4 | 1 | j = 2

| 3 | 2 | 1 | 4 | Total 3 comparisons

Pass 2:- | 3 | 2 | 1 | 4 | j = 0, i = 1

| 2 | 3 | 1 | 4 | j = 1

| 2 | 1 | 3 | 4 | Total 2 comparisons

i = 2

Pass 3:- | 2 | 1 | 3 | 4 | j = 0

| 1 | 2 | 3 | 4 |

Optimised Bubble Sort Algorithm

```
void main()
{
 int a[10];
 int size;
 int flag=0;
  int temp;
 printf("\n Enter array size");
  scanf("%d", &size);
  printf("\n Enter array elements");
    for(int i=0; i<size; i++)
      {
      scanf("%d",&a[i]);
      }
 for(int i=0; i<n-1; i++)
    {
    [illegible]
  for(int j=0; j<(n-1)-i ; j++)
    { flag=0;
      if (a[j] > a[j+1])
        {
          temp = a[j];
          a[j] = a[j+1]
          a[j+1] = temp; flag=1;
        [illegible]}
        }
        if(flag==0)
        {
          break;
        }
        }
```

Spiral

Date .....

```
printf("\n Array elements are");
for(int i=0; i<n; i++)
  {
  printf("%d", a[i]);
  }
}
```

Ex: | 2 | 1 | 3 | 4 | flag = 0
0 1 2 3

i = 0, j = 0

| 1 | 2 | 3 | 4 | flag = 1
↑

j = 1

| 1 | 2 | 3 | 4 | flag = 0

i = 2

| 1 | 2 | 3 | 4 | flag = 0

Sorted Array:- | 1 | 2 | 3 | 4 |

Time Complexity of Bubble Sort Algorithm:- n times x n times
Worst Case = $O(n^2)$

Time Complexity of Bubble Sort Algorithm in best case:- (Optimised)
1 x n
= O(n) times.

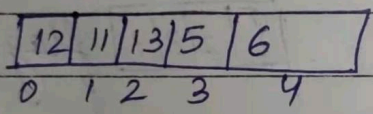
Date .....

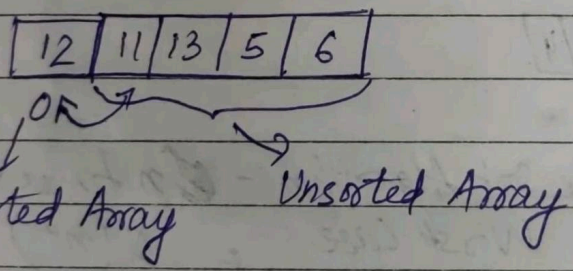
Insertion sort

Insertion sort is a simple sorting algorithm that works similar to sorting playing cards. The array is virtually split into a sorted and unsorted part. Values from unsorted array are picked and placed at the correct position in the sorted array. This algorithm works efficiently for small data sets.

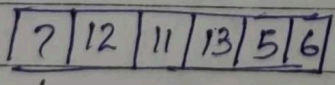
Ex:- a[ ] = {12, 11, 13, 5, 6}

Initially we consider first element as the part of sorted array.

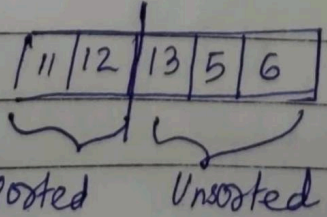




∵ 12 > 11 so we will shift (right) 12 by one position.



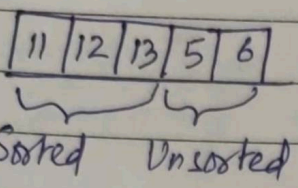
Again we will compare 11 with a[0] since a[0] is blank, so we will place 11 at a[0].



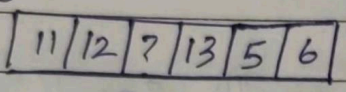
Spiral

Date .....

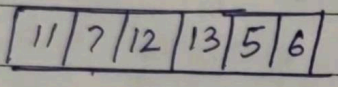
Now we will compare 13 with the 2 elements in the sorted array.
∵ 13 > 12, so no right shifting.



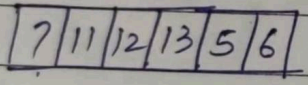
Now we will compare 5 with 13, since 5<13, so we will right shift 13.



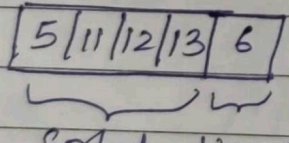
Now compare 5 with 12, since 5<12 so, we will right shift 12.



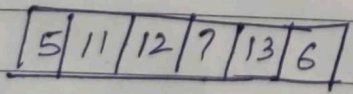
Now compare 5 with 11, since 5<11, so we will right shift 11.



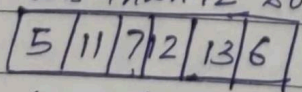
Since a[0] is empty, so we will place 5 at a[0].



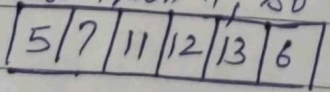
Since 6 is less than 13, so we will right shift 13.



Since 6 is less than 12 so we will right shift 12.



Since 6 is less than 11, so we will right shift 11.



5<6, so no need to right shift and place 6 at a[1].

Spiral

Date .....

Sorted Array:-

5	6	11	12	13
0	1	2	3	4

Code

```
void main()
{
 int a[10];
 int size; int j i
 int temp; printf("\n Enter array size"); &scanf("%d", &size);
 printf("\n Enter array elements");
 scanf("%d", &a[i]);
 for( i=0; i<size; i++)
  {
  scanf("%d", &a[i]);
  }
  = i
  = for(int i=1; i<=n; i++) { j=i-1;
  while(j>=0 && a[j] temp) temp =a[i];
  {
   temp = [j];
   =
   temp =
   a[j+1]= a[j];
   j--;
  }
  a[j+1] = temp;
 }
printf("\n Sorted array is: ");
 for(int i=0; i<n; i++)
  {
```

Date .....

```
printf("\n %d", a[i]);
}
}
```

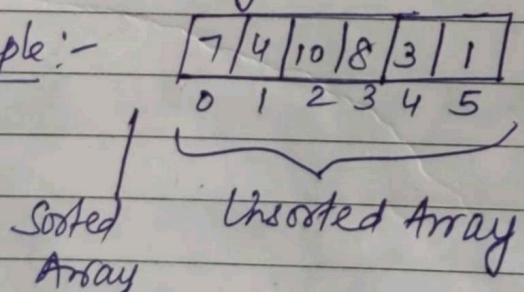
Time complexity in worst case: $O(n^2)$
Time Complexity in best case: $O(n)$.

Selection sort Algorithm

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest element from unsorted array and moving it to the sorted array.
This algorithm repeatedly selects the smallest element from the unsorted array and swaps it with the first element of the unsorted array. This process is repeated with the remaining unsorted array elements till the entire list is sorted.

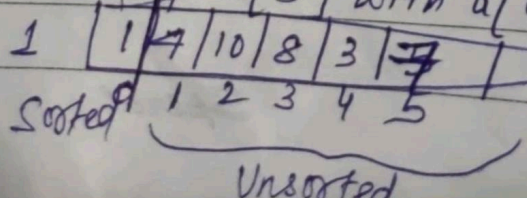
This algorithm maintains two sub arrays:
① The subarray which is already sorted
② The remaining unsorted subarray.

Example:-



We will find the minimum element from the unsorted array and swap it with first element and put it into sorted array.

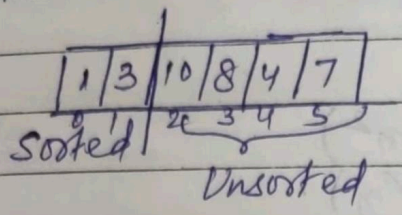
Min = 1 ie a[5].
Swap a[5] with a[0].



Speral

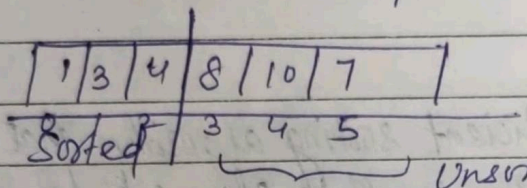
Date .....

Pass 2: Min is 3 ie a[4] so swap a[4] with a[1].



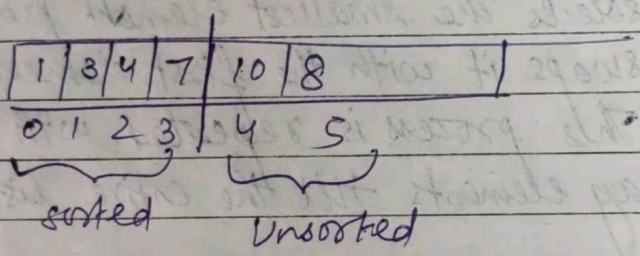
Min is 4 so swap a[4] with a[2]

Pass 3:



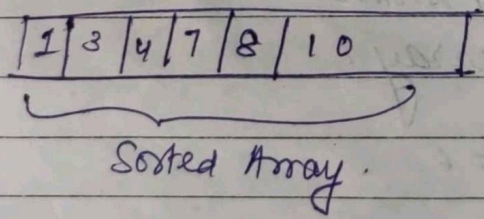
Min is 7 so swap a[5] with a[3].

Pass 4:



Min is 8 so swap a[5] with a[4].

Pass 5:



Code

```
void main()
{
int a[10];
int size; scanf("%d", &size);
printf("\n Enter array elements");
for(int i=0; i<size; i++)
  {
scanf("%d", &a[i]);
}
```

Spiral

```
for(int i=0; i<n-1; i++) → For passes
{
   int min = i;
for(int j = i+1; j<n; j++)
{
      if(a[j] < a[min])
      {
         min = j;
      }
}
if(min != i)
{
   temp = a[min];
   a[min] = a[i];
   a[i] = temp;
}
}
printf("\n Sorted array is");
for(int i=0; i<size; i++)
{
   printf("%d", a[i]);
   }
}
```

Time complexity of selection sort in worst case: $O(n^2)$

Spiral

Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each sub array and then merging the sorted subarrays to form the final sorted array. It works on divide and conquer tech.

Ex:-

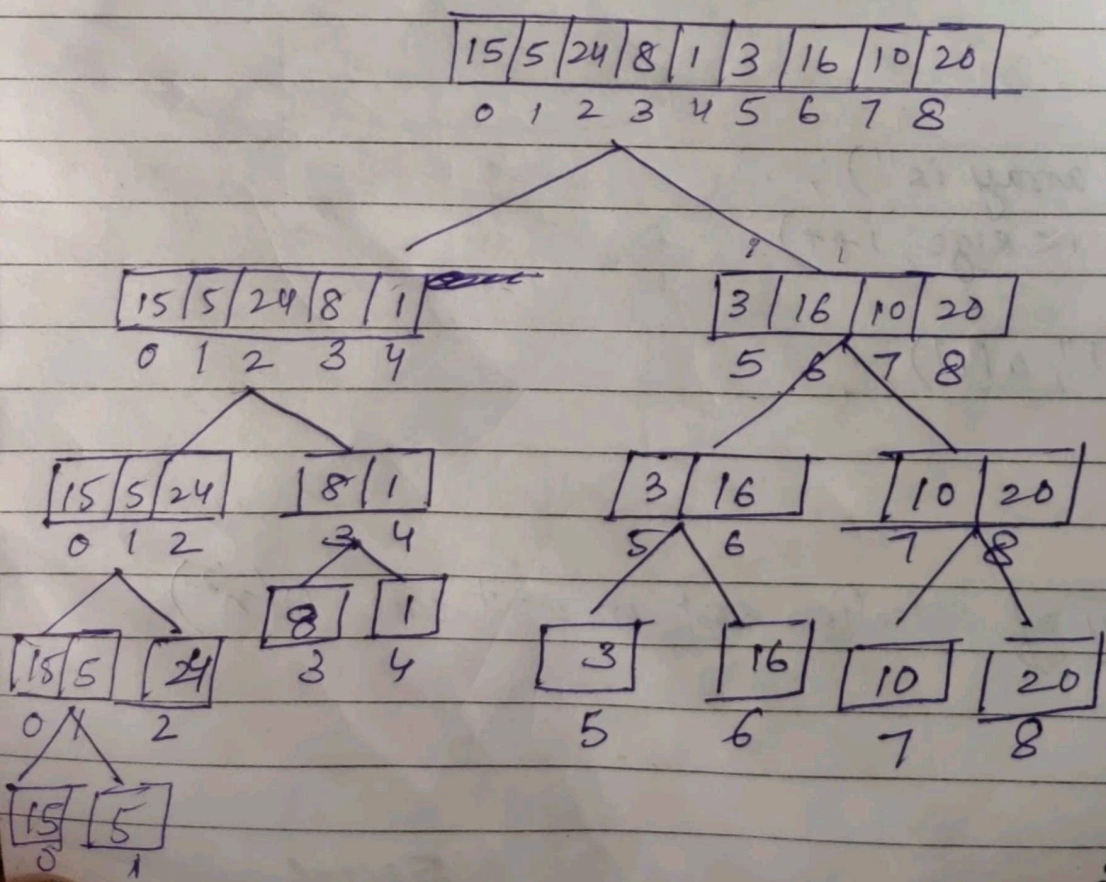
15	5	24	8	1	3	16	10	20
0	1	2	3	4	5	6	7	8

First we have to find the mid position of the array.

$$mid = \frac{lb + ub}{2}$$

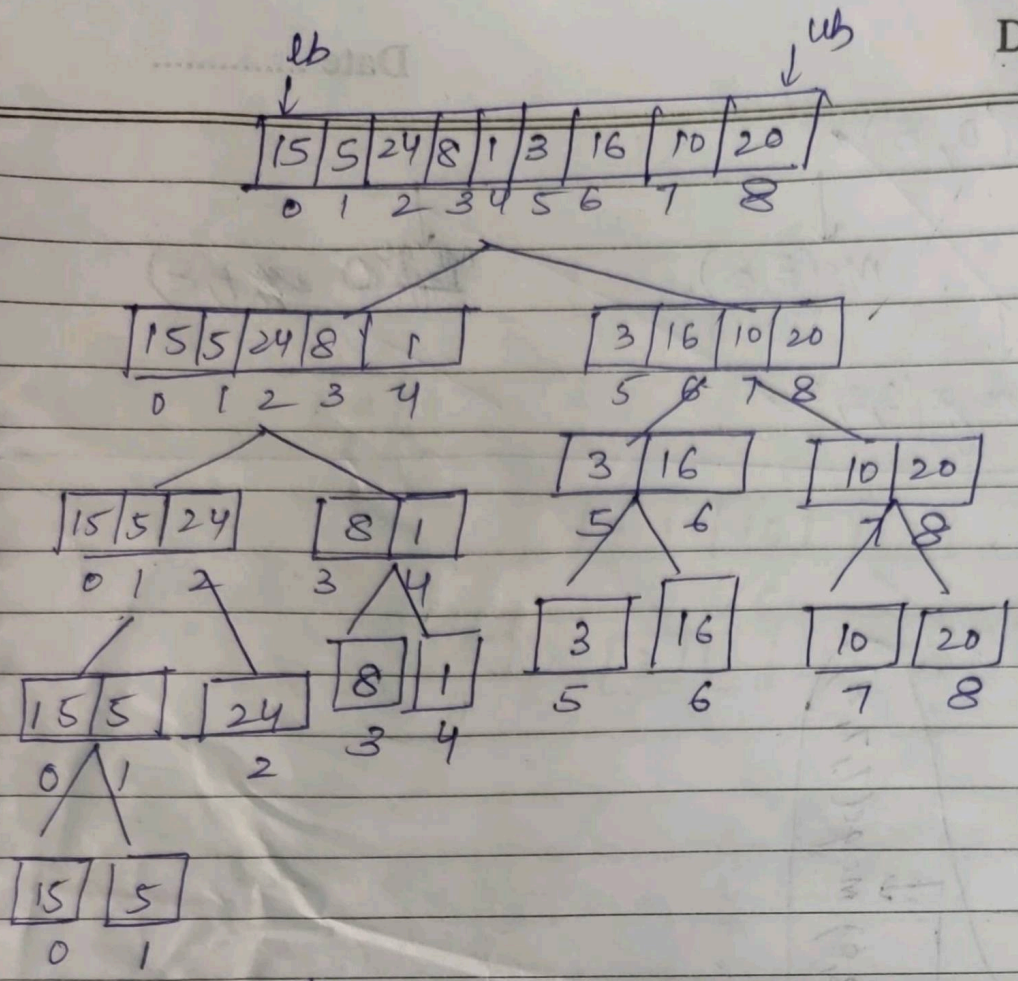
$$= \frac{0+8}{2} = 4$$

First sub list will contain elements till index 0 to 4 and second sublist will contain elements till index 5 to 8.



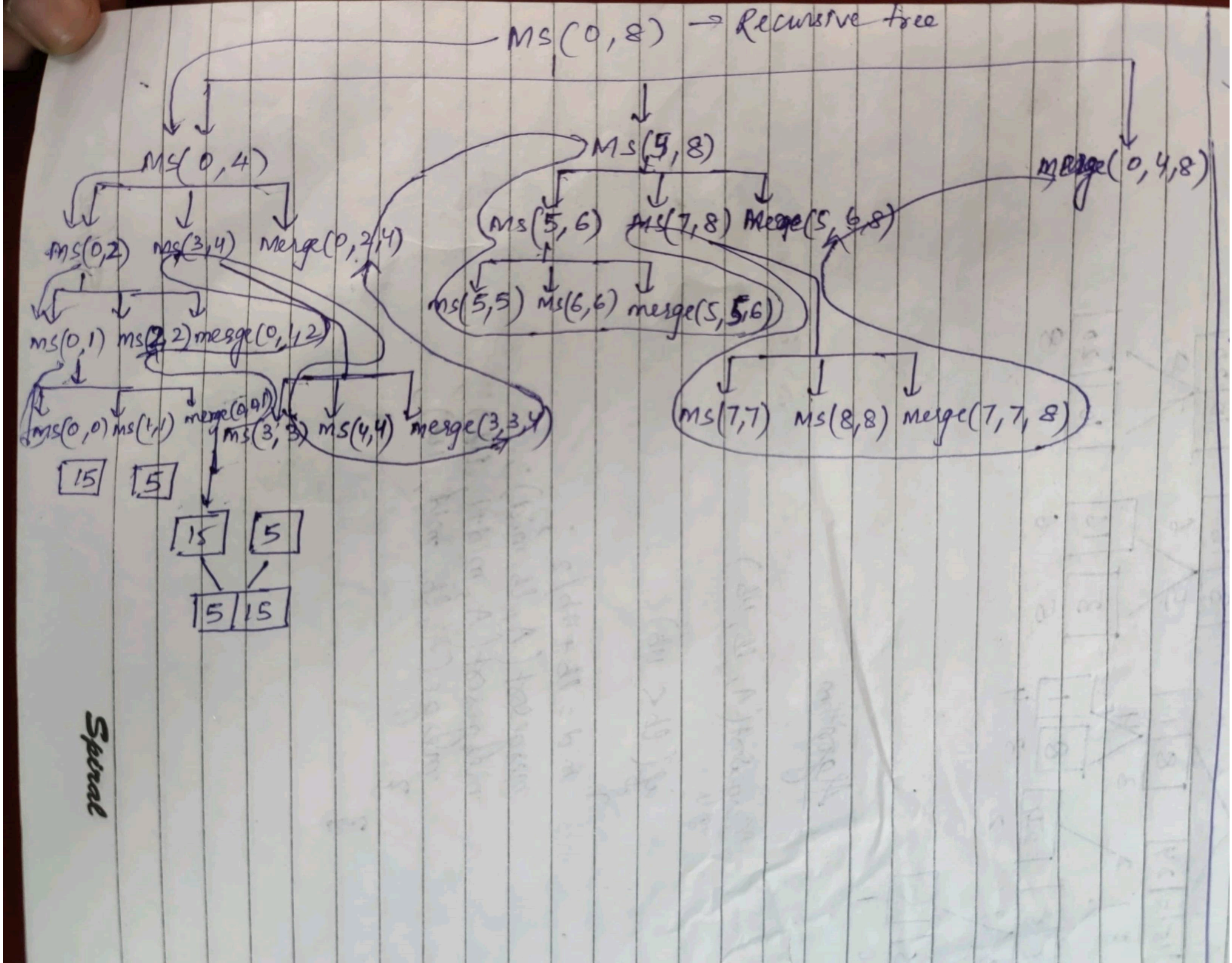
Spiral

Date .....

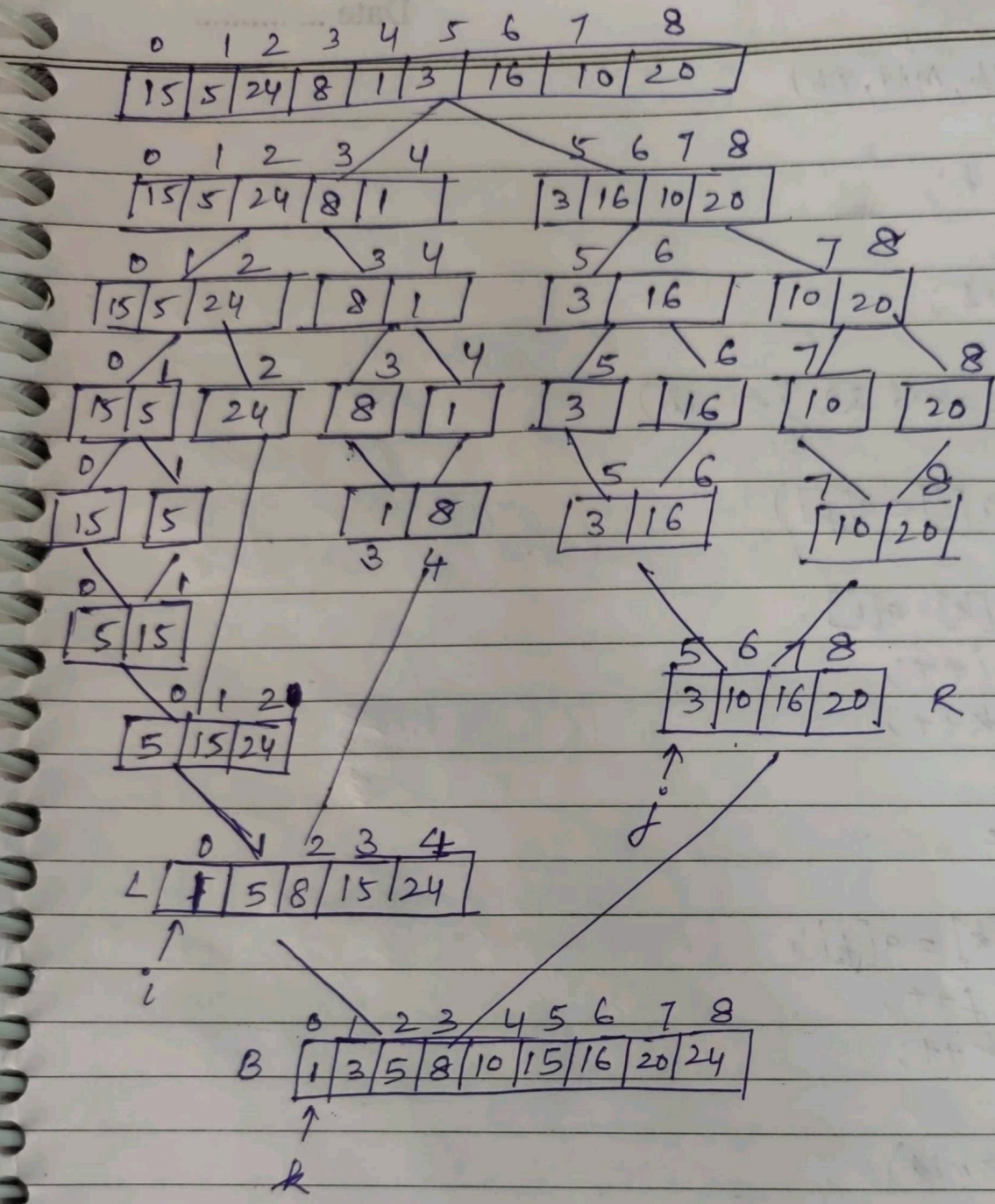


Algorithm

```
MergeSort(A, lb, ub)
{
  if(lb < ub)
  {
   mid = lb+ub/2;
  mergesort(A, lb, mid);
  mergesort(A, mid+1, ub);
   merge(A, lb, mid, ub);
  }
}
```


MS(0,8) → Recursive tree
MS(0,4)
MS(5,8)
merge(0,4,8)
MS(0,2)
MS(3,4)
Merge(0,2,4)
MS(5,6)
MS(7,8)
Merge(5,6,8)
MS(0,1)
MS(2,2)
merge(0,1,2)
MS(5,5)
MS(6,6)
merge(5,5,6)
MS(0,0)
MS(1,1)
merge(0,0,1)
MS(3,3)
MS(4,4)
merge(3,3,4)
MS(7,7)
MS(8,8)
Merge(7,7,8)
15
5
15
5
5
15

Date .....



```
Merge(A, lb, mid, ub)
{
  int i, j, k;
  i = lb;
  j = mid+1;
  k = lb;
  while(i<=mid && j<=ub)
  {
    if(a[i]<=a[j])
    {
      b[k]=a[i];
      i++;
      k++;
    }
    else
    {
      b[k]=a[j];
      j++;
      k++;
    }
    if(i>mid)
    {
      while(j<=ub)
      {
        b[k]=a[j];
        j++;
        k++;
      }
    }
    else
    {
      while(i<=mid)
```

```
    { b[k] = a[i];
      i++;
      k++;
    }
  }
  for(k=lb; k<=ub; k++)
    {
      a[k] = b[k];
    }
  printf("\n Sorted Array is");
  for(int i=0; i<n; i++)
    {
      printf("%d\t", a[i]);
    }
  }
}
```

~~Time Complexity~~

Date .....

Linear Search

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of list till the desired element is found, otherwise the search continues till the end of data set.
In best case, its time complexity is O(1), If the item to be searched for is present at 0th index.
In worst case, its time complexity is O(n), if the element to be searched is present at the last index.

Ex:-

int a[9] = | 10 | 50 | 30 | 70 | 80 | 60 | 20 | 90 | 40 |
0 1 2 3 4 5 6 7 8

To search for item: 20

Item to be found will be ~~searched~~ matched with each element at every ~~all~~ indexes.

Program

```
void main ()
{
int a[9] = {10, 50, 30, 70, 80, 60, 20, 90, 40}
int item, ____; flag;
printf("\n Enter item to be searched for");
scanf("%d", &item);
for(int i = 0; i < 9; i++)
  {
    if (a[i] == item)
    {
      flag = 1;
      break; } }
```

Spiral

```
if (flag == 1)
{
printf("\n Element found at pos %d", i+1);
}
else
{
printf("\n Element not found");
}
}
```

Binary Search

Binary search is searching algorithm that searches the item by dividing the array into two subarrays. Its time complexity is O(log n). For binary search algorithm to be performed array must be sorted.

Ex:- int a[9] = | 10 | 20 | 30 | 50 | 70 | 80 | 90 | 120 | 140 |
0 1 2 3 4 5 6 7 8

To search for 20

beg = 0, end = 8

$$mid = \frac{beg + end}{2} = \frac{8}{2} = 4$$

$a[4] = 80 \neq 20$ ∵ $20 < 70$

Our new sub array will be | 10 | 20 | 30 | 50 |
0 1 2 3

beg = 0, end = 3

$$mid = \frac{0+3}{2} = 1$$

$a[1] = 20 = 20$ item found at Spiral index 1.

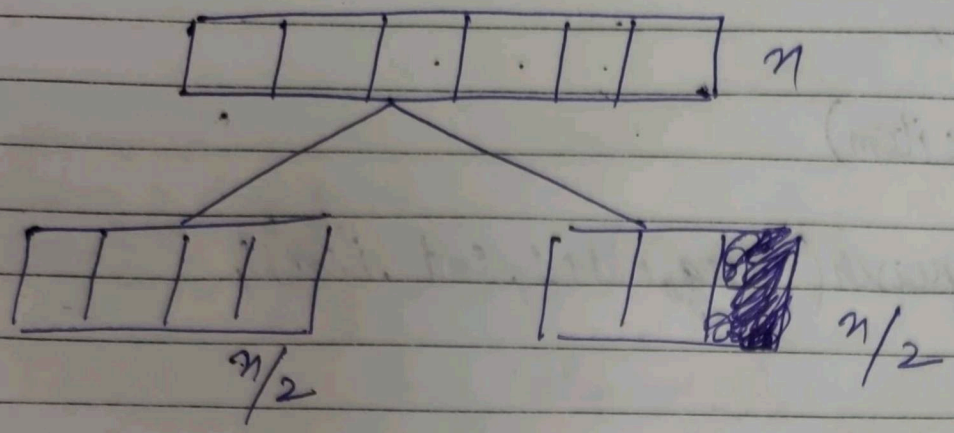
Code

```
void main()
{
int a[10];
int n, item, res;
printf("\n Enter array size");
scanf("%d", &n);
printf("\n Enter array elements");
for(int i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
printf("\n Enter item to be searched for");
scanf("%d", &item);
res = binarysearch(a, 0, n-1, item);
if (res == 0)
  {
    printf("\n Item not found");
  }
else
  {
  printf("\n Item found at pos %d", res);
  }
}
int binarysearch (int a[], int beg, int end, int item)
  {
    if(beg <= end)
      {
        int mid = (beg+end)/2;
```

Date .....

```
if (a[mid] == item)
{
    return mid+1;
}
else if (a[mid] < item)
  {
    return binarysearch(a, beg, mid+1, end, item);
  }
else
{
return binarysearch(a, beg, mid-1, item);
}
return 0;
}
```

Time Complexity of merge sort



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

Now put $n = \frac{n}{2}$ in $T\left(\frac{n}{2}\right)$

$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

$$= 2^2 T\left(\frac{n}{4}\right) + n + n$$

$$= 2^2 T\left(\frac{n}{4}\right) + 2n$$

$$= 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 2^3 T\left(\frac{n}{8}\right) + 3n$$

We will continue till $T(1)$.

In generalised form it can be written as $2^k T\left(\frac{n}{2^k}\right) + kn$

For $T(1)$

$$\frac{n}{2^k} = 1$$

Date .....

$n = 2^k$

Take $\log_2$ on both sides

$\log_2 n = \log_2 2^k$

$\log_2 n = k \log_2 2$

$\log_2 n = k$

Our generalised eq^n will be

$$2^{\log_2 n} T(1) + \log_2 n \cdot n$$

$$= 2^{\log_2 n} \cdot 1 + n \log_2 n$$

$$= n + n \log_2 n$$

We will consider the larger value as time complexity.

Therefore time complexity of Merge sort will be $n \log_2 n$.

Time Complexity of Binary search

$$T(n) = c + T\left(\frac{n}{2}\right) - (1)$$

$$T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{4}\right) - (2)$$

Put (2) in (1)

$$T(n) = c + c + T\left(\frac{n}{4}\right)$$

$$T(n) = 2c + T\left(\frac{n}{4}\right) - (3)$$

$$T\left(\frac{n}{4}\right) = c + T\left(\frac{n}{8}\right) - (4)$$

Put (4) in (3)

$$T(n) = 2c + c + T\left(\frac{n}{8}\right)$$

$$T(n) = 3c + T\left(\frac{n}{8}\right)$$

Generalized form will be

$$T(n) = kc + T\left(\frac{n}{2^k}\right) - (5)$$

Now make $T(1)$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

Taking log on both sides,

$$\log n = \log 2^k$$

Date .....

$\log n = k \log_2 2$

$k = \log n$

Substitute k in (5)

$T(n) = c \log n + T\left(\frac{n}{2^{\log_2 n}}\right)$

$T\left(\frac{n}{\log_2 n 2}\right)$

$T(n) = c \log n + T(1)$

Now ignoring the constants, time complexity would be $O(\log n)$.

Date .....

Difference between Searching & Sorting Algorithms

Searching Algorithms are used to search for an element in the ~~array~~ data structure. Searching can be classified into two types:

(1) Sequential search: Sequential search is basic and simple searching algorithm. Sequential search starts at the beginning of the list ~~of~~ or array. It traverses the array or list sequentially and matches the element to be searched for with every element of the array. Linear search is an example of sequential search.

(2) Interval search: These algorithms are designed to search for an element in sorted data structure. These type of searching algorithms are much more efficient than linear search algorithm. Binary search is an example of interval search.
A binary search searches the given element in the array by dividing the array into two halves.

Sorting Algorithm is used for arranging the data of list or array either in ascending order or descending order. Bubble sort, Insertion sort, selection sort, merge sort etc are examples of sorting algorithms.
There are 2 categories of sorting:
(1) Internal sorting: When all data to be sorted is placed in memory then it is called internal sorting.
(2) External sorting: When all data to be sorted cannot be placed in memory at a time. External Sorting is used for massive amounts of data. Merge sort is an example of

Date .....

external sorting algorithm. External storages like hard disk, CD is used for external ~~sotage~~ storage.

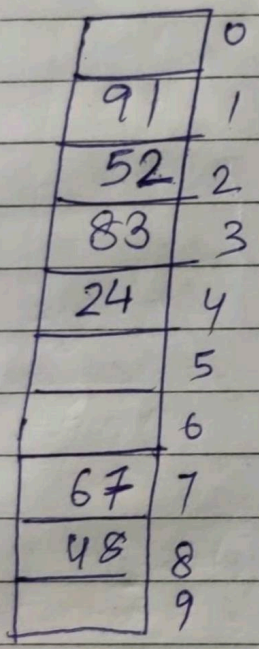
Date .....

Hashing

Hashing is the technique or process of mapping values with its keys in hash table by using hash function. It is done for faster access of elements. The efficiency of mapping depends on efficiency of hash function used. Using hashing we can retrieve elements in O(1) time.

Searching of data is based on key.

Hash table is a data structure which stores elements in respective keys.



Hash table

For ex: To insert values: 24, 52, 91, 67, 48, 83

Let hash function: k mod 10

24 mod 10

= 4 → key value

We will put 24 at key 4.

To insert 52

52 mod 10

2

To insert 91 91 mod 10 = 1

Date .....

To insert 67

67 mod 10 = 7

To insert 48

48 mod 10 = 8

To insert 83

83 mod 10 = 3

Insertion of elements will take O(1) time because we only have to calculate hash value and store element at that index.

To search 67.

67 mod 10 = 7

Searching will also take O(1) time.

Deletion will also take O(1) time.

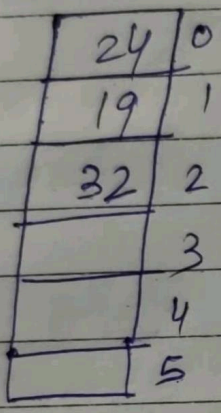
In hashing, there can occur collisions.

Collisions means when element is already stored at particular key and we want to store another element at that key.

Collision Resolution Techniques

- Chaining (Open Hashing)
- Open Addressing (Closed Hashing)
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Example of collision



$m = 6$ (No of slots in hash table)

Let hash function = $k \bmod m$

To insert: 24, 19, 32, 44

$24 \bmod 6 = 0$

$19 \bmod 6 = 1$

$32 \bmod 6 = 2$ } Collision

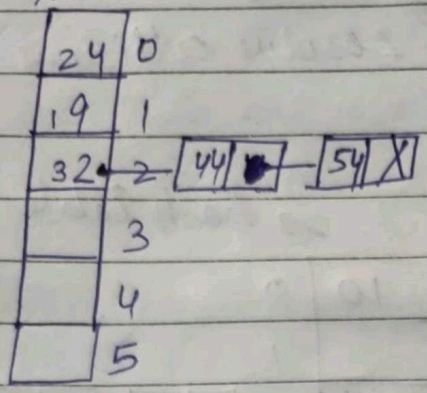
$44 \bmod 6 = 2$ }

Chaining

We will create chain of elements at a particular index in the form of linked list

Date .....

For ex:-



In this we are using available hash space plus external space that's why it is open hashing.

In closed hashing we will try to occupy all the slots of hash table.

Linear Probing

In linear probing, if the slot is not empty then it will put element in next slot sequentially.

Quadratic Probing

Hash function :- $(h + i^2) \bmod m$, i = probe no. i.e no. of times we have checked for empty slot

To put 30 in hash table

$h = 30 \bmod m$ $\quad i = 1$

$= 30 \bmod 6$

$= 0 + (1)^2$

$= 1 \bmod 6$

$= 1$

1 slot is already filled. $i = 2$

$= 0 + (2)^2$

$= 4 \bmod 6 = 4$

30 will be put at index 4.

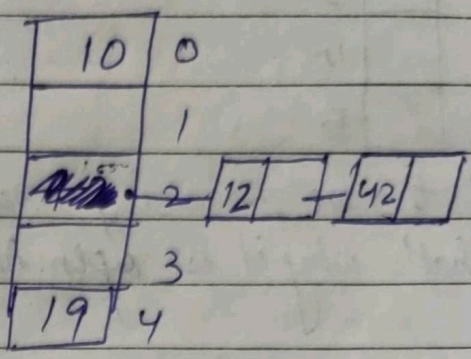
Spiral

Date .....

In Double hashing we use two hash functions.
One to insert data and second to resolve collisions.

Chaining (Open hashing)

Values :- 42, 19, 10, 12
Hash function : k mod 5



To insert : 42
42 mod 5
= 2

To insert : 19
19 mod 5
= 4

To insert : 10
= 10 mod 5
= 0

To insert : 12
= 12 mod 5
= 2

2nd slot of hash table is already fulled so we will link 12 with 42 ~~using~~ in the form of linked list.

Advantages of chaining

1. Deletion is easy.
2. Insertion is easy and takes O(1) time.

Disadvantage

Searching will be difficult and take O(n) time

Spiral

If all the elements are forming large link list at a particular slot.

Although slots are available in the hash table but still it is taking extra space.

Open Addressing

(1) Linear Probing

$h(k) = k \bmod m$

$h(k) = k \bmod 10$

19	0
	1
72	2
43	3
23	4
135	5
82	6
	7
	8
99	9

To resolve collision:

$h'(k, i) = (h(k) + i) \bmod m$

To insert :- 43, 135, 72, 23, 99, 19, 82

$43 \bmod 10 = 3$

$135 \bmod 10 = 5$

$72 \bmod 10 = 2$

$23 \bmod 10 = 3 \Rightarrow$ this slot is already filled.

So we will use $h'(k, i) = (h(k) + i) \bmod m$

Here $i = 1$

$(h(k) + i) \bmod m$

$= (3 + 1) \bmod 10$

$= 4 \bmod 10 = 4$

So we will put 23 at index 4

$99 \bmod 10 = 9$

Date .....

19 mod 10 = 9 → this slot is already filled

$h'(k,i) = (h(k)+i) \bmod m \quad i = 1$

$= (9+1) \bmod 10$

$= 10 \bmod 10$

$= 0$

82 mod 10 = 2 → this slot is already filled

$h'(k,i) = (h(k)+i) \bmod m \quad i = 1$

$= (2+1) \bmod 10$

$= 3 \bmod 10$

$= 3$ → this slot is already filled

$h'(k,i) = (h(k)+i) \bmod m \quad i = 2$

$= (2+2) \bmod 10$

$= 4 \bmod 10$

$= 4$ → this slot is already filled

$i = 3$

$= (2+3) \bmod 10$

$= 5 \bmod 10$

$= 5$ → this slot is already filled.

$i = 4$

$(2+4) \bmod 10$

$= 6 \bmod 10$

$= 6$

Put 82 at slot 6.

Advantage

No extra space required.

Disadvantage

Searching will take time and it will be O(n) in worst case.

Deletion is also difficult.
Primary clustering is created means elements are grouped at a particular place.

Quadratic Probing

$h(k) = k \bmod 10$

If collision occurs, $h'(k, i) = (h(k) + i^2) \bmod m$

To insert: 42, 16, 91, 33, 18, 27, 36, 62

$42 \bmod 10 = 2$

$16 \bmod 10 = 6$
$91 \bmod 10 = 1$
$33 \bmod 10 = 3$
$18 \bmod 10 = 8$
$27 \bmod 10 = 7$
$36 \bmod 10 = 6 \rightarrow$ slot is already filled

36	0
91	1
42	2
33	3
	4
	5
16	6
27	7
18	8
	9

$(h(k) + i^2) \bmod m$ $\quad i = 1$
$(6 + 1) \bmod 10$
$7 \bmod 10 = 7 \rightarrow$ slot is already filled
$i = 2$
$(h(k) + i^2) \bmod m$
$= (6 + 4) \bmod 10$
$= 10 \bmod 10$
$= 0$
$62 \bmod 10 = 2 \rightarrow$ slot is already filled
$i = 1$
$(2 + 1) \bmod 10 = 3 \bmod 10 = 3 \rightarrow$ slot already filled.

Spiral

Date .....

$i = 2$

$(2 + (2)^2) \bmod 10$

$(2+4) \bmod 10$

$6 \bmod 10 = 6 \rightarrow$ slot already filled

$i = 3$

$(2 + (3)^2) \bmod 10$

$(2+9) \bmod 10$

$11 \bmod 10$

$= 1 \rightarrow$ slot already filled

$i = 4$

$(2 + (4)^2) \bmod 10$

$18 \bmod 10$

$= 8 \rightarrow$ slot already filled

$i = 5$

$(2 + 25) \bmod 10$

$27 \bmod 10 = 7 \rightarrow$ slot already filled.

$i = 6$

$(2 + 36) \bmod 10$

$38 \bmod 10 = 8 \rightarrow$ slot already filled.

62 will never get slot in hash table ~~also~~ although slots are free.

Advantage

No extra space Required
Primary clustering resolved.

Disadvantage

Insertion, deletion and searching of element in worst case will take $O(n)$ time.

No gurantee of finding slot if Hash Table is half filled.

Secondary clustering → If two or more elements are trying to ~~occupy~~ form the same probing sequence i.e trying to occupy same slot.

Double Hashing

In this, we use two hash functions.

$h_1(k) = k \bmod 11$

$h_2(k) =$ ~~prime mod prime~~ $= 8 - k \bmod 8$ — Relatively prime

For collision : $(h_1(k) + i\,h_2(k)) \bmod m$

Values :- 20, 34, 45, 70, 56

$20 \bmod 11 = 9$

$34 \bmod 11 = 1$

$45 \bmod 11 = 1 \rightarrow$ slot already filled

	0
34	1
	2
56	3
45	4
	5
70	6
	7
	8
20	9
	10

$h_2(k) = 8 - 45 \bmod 8$

$= 8 - 5 = 3$

$i = 1$

$(1 + 1 \times 3) \bmod 11$

$= (1 + 3) \bmod 11$

$= 4 \bmod 11$

$= 4$

Date .....

70 mod 11 = 4 → already filled

$h_2(k) = 8 - 70 \bmod 8$

$= 8 - 6 = 2$

$i = 1$

$(4 + 1 \times 2) \bmod 11$

$6 \bmod 11 = 6$

56 mod 11 = 1 → already filled

$h_2(k) = 8 - 56 \bmod 8$

$8 - 0 = 8$

$i = 1$

$(1 + 1 \times 8) \bmod 11$

$= 9 \bmod 11 = 9$ → already filled

$i = 2$

$(1 + 2 \times 8) \bmod 11$

$(1 + 16) \bmod 11$

$17 \bmod 11 = 6$ → already filled

$i = 3$

$(1 + 3 \times 8) \bmod 11$

$25 \bmod 11$

$= 3$

Double hashing will effectively utilise hash table. The keys will be distributed uniformly.

Advantage	Disadvantage
No extra space	Insertion, Deletion, searching will take $O(n)$ time.
No primary clustering.	
No secondary clustering	

Spiral

Types of Hash Functions

1. Division Method
2. Mid square Method
3. Digit Folding Method or Pairing Method
4. Multiplicative Method

1. Division Method

This is the most common hash function using which we can insert data in hash table. In this we perform modulo method

Ex: 33, 11, 66, 44

$h(k) = k \bmod m$, m = size of hash table

33 mod 10
= 3

11 mod 10
= 1

66 mod 10
= 6

44 mod 10
= 4