Insertion sort

//1. pick one element (second element) of the array
//2. compare picked element with all its left neighbours one by one
//3. if left neighbour is greater than picked element then move left neighbour one position adead
//4. insert picked element at its appropriate position
//5. repeat above steps untill array is sorted

0	1	2	3	4	5
55,	**44,**	**22,**	**66,**	**11,**	**33**
44	55	22	66	11	33
22	44	55	66	11	33
22	44	55	66	11	33

```
for(int i = 1 ; i < N ; i++) {
        int temp = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
```

i	i<6	temp	j	j>=0&&arr[j]>temp
1	T	44	0,-1	T,F
2	T	22	1,0,-1	T,T,F
3	T	66	2	F

Algorithm analysis / Efficiency measurement / Complexities

- finding time and space required to execute an algorithm
    - time - time required to execute on machine (CPU) (nS, uS, mS, S)
    - space - space required to execute algorithm inside memory (bytes, Kb, Mb,..)

1. Exact analysis
    - finding exact time and space requirement of an algorithm
    - exact time and space of the algorithm is dependent on some external factors
    - time is dependent on type of matchine, number of processes running on the system
    - space is dependent on type of machine (architecture), data types

2. Approximate analysis
    - finding time and soace requirement approximately
    - asymptotic analysis - mathematical way of finding time and space complexity of the algorithm
    - it also tell about the behavior of an algorithm for different inputs or for different sequence of input
    - behaviour of an algorithm can be of three types
        1. Best case
        2. Average case
        3. Worst case

Time Complexity

- count the number of iterations for the loop which is used inside the algorithm
- timp required is directly proportional to the iterations of the loop

1. print 1D array on console

```
void print1DArray(int arr[], int n){
        for(int i = 0 ; i < n ; i++)
                sysout(arr[i]);
}
```

no. of iterations = n

Time $\propto$ no. of iteration

Time $\propto$ n

$\boxed{T(n) = O(n)}$

1. print 2D array on console

```
void print2DArray(int arr[][], int m, int n){
        for(int i = 0 ; i < m ; i++)
                for(int j = 0 ; j < n ; j++)
                        sysout(arr[i]);
}
```

no. of iterations (outer loop) = m

no. of iterations (inner loop) = n

Total no. of iteration = m * n

Time $\propto$ m * n

$\boxed{T(n) = O(m*n)}$

$\because$ m = n

iterations = n * n = n^2

Time $\propto$ n^2 $\boxed{T(n) = O(n^2)}$

3. add two numbers

```
int addition(int num1, int num2){
        return num1 + num2;
}
```

- time required is constant because it will not varry according to the values of variable.
- constant time requirement can be denoted as

$$\boxed{T(n) = O(1)}$$

4. print table of given number

```
void printTable(int num){
        for(int i = 1 ; i <= 10 ; i++)
                sysout(i * num);
}
```

- loop is going to iterate constant number of times always
- constant time requirement

$$\boxed{T(n) = O(1)}$$

5. print binary of decimal number

2	9	1
	4	0
	2	0
	1	1

$(9)_{10} = (1001)_2$

```
void printBinary(int num){   (n)
        while(num > 0){
                sysout(num % 2);
                num = num / 2;
        }
}
```

num	num > 0	num % 2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$n = 9, 4, 2, 1, 0$

$n = n, n/2, n/4, n/8 \ldots\ldots$

$= n/2^0, n/2^1, n/2^2, n/2^3 \ldots\ldots n/2^{itr}$

for $n=1$
last time loop condition is true

$n/2^{itr} = 1$

$n = 2^{itr}$

$\log 2^{itr} = \log n$

$itr \log 2 = \log n$

$itr = \frac{\log n}{\log 2}$

Time $\propto$ itr

Time $\propto \frac{\log n}{\log 2}$

$\boxed{T(n) = O(\log n)}$

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), ... O(2^n), ...

modification : '+' / '-' **--> T(n) will be in terms of n**

modification : '*' / '/' **--> T(n) will be in terms of log n**

for(i=0; i<n; i++) → O(n)

for(i=n; i>0; i--) → O(n)

for (i=0; i<n; i+=20) → O(n)

for(i=0; i<20; i++) → O(1)

for(i=n; i>0; i/=2) → O(log n)

for(i=1; i<n; i*=2) → O(log n)

for (i=0; i<n; i++)
for(j=0; j<n; j++) → $O(n^2)$

for(i=0; i<n; i++)
for (j=n; j>0; j/=2) → O(n log n)

for(i=0; i<n; i++); → n
for(j=0; j<n; j++); → n

+ = 2n $T \propto 2n$ T(n) = O(n)

Searching Algorithms Analysis

- for searching and sorting algorithms, we count number of comparisions
- time is directly proportinal to number of comparision

Linear Search

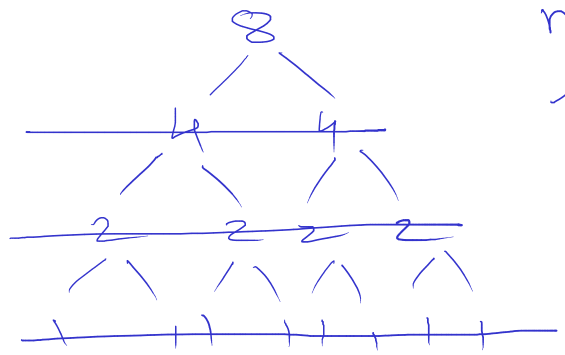
Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(n)
Worst case :	key is found in last few comparitions	:	O(n)
	key is not found		

Binary Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(log n)
Worst case :	key is found in last few comparitions	:	O(log n)
	key is not found		

Iterative : count no. of iterations $\rightarrow O(\log n)$

Recursive : count no. of recursive calls $\rightarrow O(\log n)$



$n = 8$

$l = 3$

$2^3 = 8$

$2^l = n$

$2^c = n$

$c = \frac{\log n}{\log 2}$

Sorting Algorithms Analysis

Array length = n (6)

No. of passes = n - 1

pass 1 : 5   n-1

pass 2 : 4   n-2

pass 3 : 3   n-3

pass 4 : 2   ⋮

pass 5 : 1   1

Total comparisions = 1+2+3+ - - - - - - - +(n-2)+(n-1)

$= 1+2+3+ - - - - - - + n$

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

$\text{comps} = \frac{n^2+n}{2}$

$\text{Time} \propto \frac{n^2+n}{2}$

$\text{Time} \propto n^2$

$T(n) = O(n^2)$

- mathematical polynomial
- Degree of polynomial
  ↳ highest power
- consider only degree term in time complexity because it is highest growing term.

n	n^2
1	1
10	100
100	10000
1000	1000000

Basic Sorting Algorithms Analysis and Comparisions

	Worst case	Avg case	Best case
selection sort –	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort –	$O(n^2)$	$O(n^2)$	$O(n)$
insertion sort —	$O(n^2)$	$O(n^2)$	$O(n)$

Space Complexity

Total space = Input space + Auxillary space

(space of actual input (data))

(space required to process actual data)

find sum of array elements

```
int findSum(int arr[], int size){
	int sum = 0 ;
	for(int i = 0 ; i < size ; i++)
		sum+=arr[i];
	return sum;
}
```

Input variables = arr
Auxillary variable = size, sum, i
input space = n units
Auxillary space = 3 units
Total space = n + 3 units
space $\propto n+3$

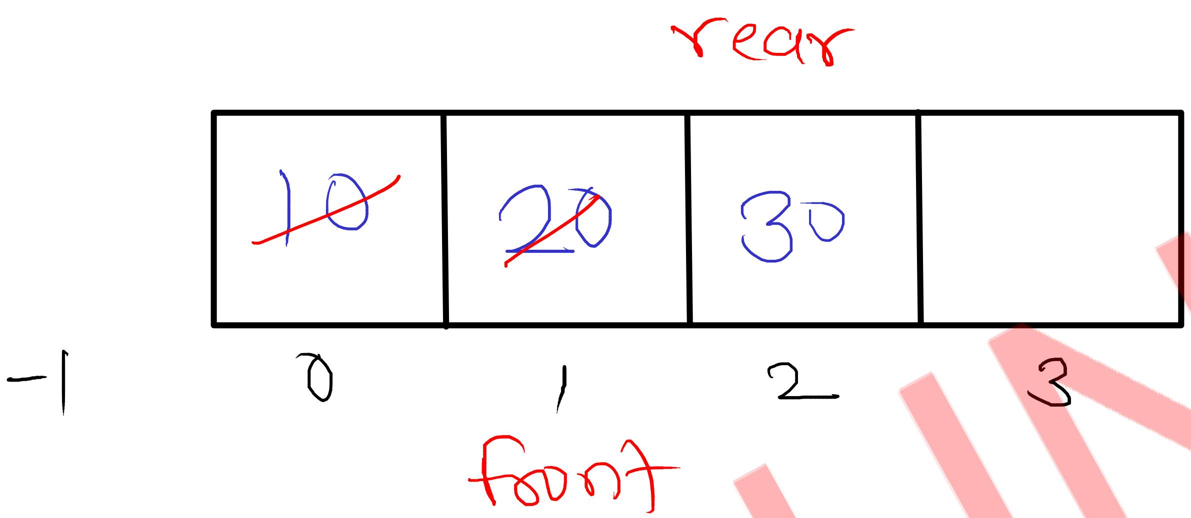
$\boxed{S(n) = O(n)}$

Auxillary space analysis
Auxillary variable = size, sum, i
Auxillary space = 3 units

$\boxed{AS(n) = O(1)}$

Linear Queue

- linear data structure
- insertion of data is allowed from one end (rear)
- removal of data is allowed from another end (front)
- work on principle of "First In First Out" (FIFO)



Operations:

1. Add/Insert/Push/Enqueue:

i. reposition rear (inc) rear++

ii. add value/data at rear index arr[rear]=v

2. Delete/Remove/Pop/Dequeue:

i. reposition front (inc)

3. Peek: (collect)

i. read data/value from front end

Conditions

1. isFull

rear == SIZE - 1

2. isEmpty

rear == front

Time complexities of array implementation of queue are

push - O(1)
pop - O(1)
peek - O(1)

Circular Queue - front and rear repositioning

size = 4

rear = front = -1

rear = (rear + 1) % size

front = (front + 1) % size

front = rear = -1

= (-1+1) % 4 = 0

= (0 + 1) % 4 = 1

= (1+1) % 4 = 2

= (2+1) % 4 = 3

= (3+1) % 4 = 0