Algorithm:

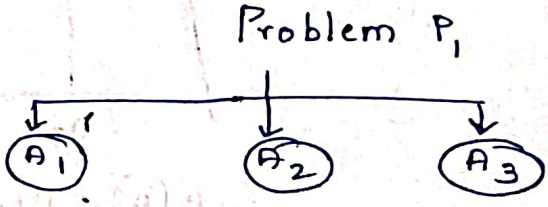
It is a step by step procedure for solving a computation problem.

Algorithm	Program
→ Written at design phase	→ Development / Implementation phase.
→ Written by someone having domain knowledge.	→ Written by a programmer.
→ Written in an English.	→ Written in a Programming language.
→ Hardware and OS independent	→ Depends on Hardware and OS.

How to write an algorithm

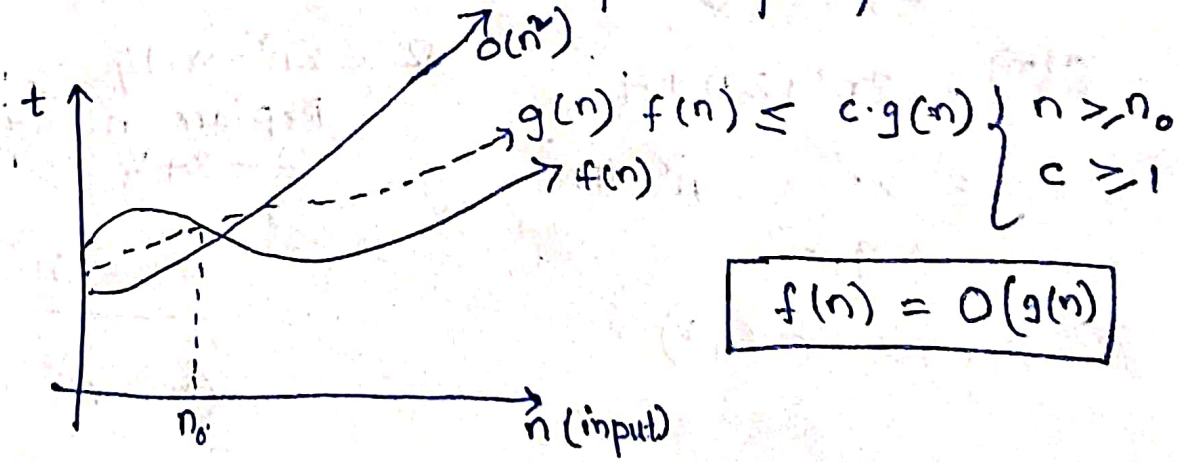
Summation of two numbers:

```
Algorithm_Sum_of_two_numbers (a, b) → no datatype{
    Sum ← a+b
}
```



A_1 Time and Space Complexity

Big O:



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

$f(n) = 3n+2$, $g(n) = n$

$3n+2 \leq c \cdot n$ for ex $c = 10$.

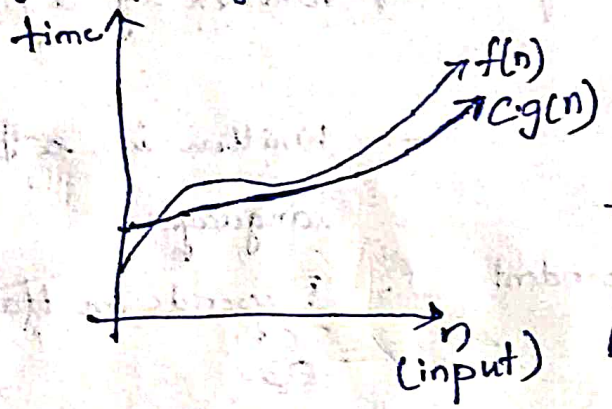
$3n+2 \leq 10n \longrightarrow \forall\ n \geq 1$

$3n + 2(n) = 5n$

$3n+2 \leq 5n$.

Big (O) denotes tighest upper bound.

Big Omega:- Tighest lower bound.



$f(n) = 3+2$

$g(n) = n$

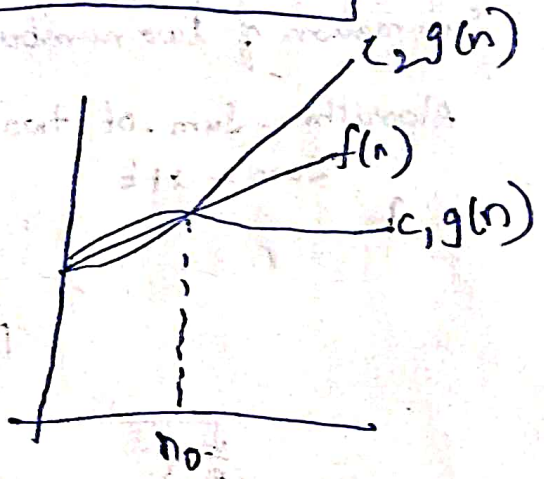
find if $f(n) = \Omega(g(n))$

$f(n) \geq cg(n)$, $\forall\ n > 1$

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

$f(n) = \theta(n)$

→ Average.



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

$f(n) = 2n^2 + 3n + 4$

O ; Ω ; θ

$O(n) = 2n^2 + 3n + 4$

$= 2n^2 + 3n^2 + 4n^2$

$\rightarrow 9n^2$

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

$\Omega = 2n^2 + 3n + 4 \geq n^2$

Replace n with 1

$2 + 3 + 4 \geq n^2$

$\Omega = n^2$

$\theta = n^2$

$f(n) = n!$

$\Omega(f(n)) = 1$

$O(f(n)) = n^n$

we cannot determine the Average Case

$\underline{\underline{\Theta}}$

$f(n) = \log n!$

$= \log(1 \times 2 \times 3 \dots n)$ ⇒ replace everything with n.

$= \log n^n$

Cannot deterime $\underline{\underline{\Theta}}$

$\log 1 \leq \log n! \leq \log n^n$

Ω ✓ $O(n)$ ✓

$\log 1 \leq \log n! \leq \underline{\underline{n \log n}}$

1. 2^n, n^2

apply log

$\log_2 n$ $\quad \log n^2$

$n \log 2$ $\quad 2 \log n$

$\Rightarrow n$, $\log_2 n$

$\Rightarrow 128$, $\log 128$

$\Rightarrow 128$, $\log 2^7$

$128 > 7$

So $\boxed{2^n > n^2}$

2. 3^n, 2^n ⟶ log

$\log 3^n$ $\quad \log_2 n$

$n \log 3$ $\quad n \log 2$

$\log 3 > \log 2$

$3^n > 2^n$

3. n^2 $\quad n \log n$

$2 \log n$ $\quad n^{\log}$

4. $n^{\log n}$ $\quad \log n^{100}$

$\log n$ $\quad 100 \log(\log n)$

$\log_2 128$ $\quad 100 \log(\log_2 128)$

$128 < 700$

5. n^2 $\quad n \log n$

$n \times n$ $\quad n \log n$

n $\quad \log n$

128 $\quad \log_2 2^7$

$128 > 7 \Rightarrow \boxed{n^2 > n \log n}$

$n = 1024$

$\log_2^{1024}$ $\quad$ $100\log(\log_2 1024)$

1024 $\quad$ $100\log_2(2^{10})$

1024 , 1000

so $\boxed{n > \log_n^{100}}$

6. $n^{\log n}$ $\quad$ $n\log n$

$\log[n^{\log n}]$ $\quad$ $\log[n\log n]$

$\log n \times \log n$ $\quad$ $\log n + \log[\log n]$

$\log_2^{1024} \times \log_2^{1024}$ $\quad$ $\log_2^{1024} + \log[\log_2^{1024}]$

$1024 \times 1024 > 1024 + \log_2 1024$

$\boxed{n^{\log n} > n\log n}$

7. $\sqrt{\log n}$ $\quad$ $\log(\log n)$

$n = 2^{64}$

$\sqrt{\log_2^{64}}$ $\quad$ $\log(\log_2^{64})$

$\sqrt{64}$ $\quad$ $\log_2(64)$

8 $\quad$ $\log_2 2^6$

$8 > 6$

$\boxed{\sqrt{\log n} > \log(\log n)}$

8. $n\log n$ $\quad$ n^2

$\cancel{n}\log n$ $\quad$ $\cancel{n} \times n$

$\log_2^{64}$ $\quad$ 2^{64}

$64 < 2^{64}$

$\boxed{n^2 > n\log n}$

Summary for function:

$C < \log(\log n) < \log n < n^{1/3} < n^{1/2} < n < n\log n < n^2 < n^3 < n^4 < 2^n < n^n$

Algorithm

- Iterative

```
for (int i = 0 to n) {
    S.O.pln ("abc");
}
```

- Recursive

```
f(x) {
    f(x-1);
}
```

1.
```
for(int i = 0; i<n; i++){
    s.o.pln ("abc");
}
```
T.C → O(n)

2.
```
for(int i = 0; i<n; i++){
  for(int j = 0; j<n; j++){
    s.o.pln.("abc")
  }
}
```
→ we are concerned the last loop will execute.

i	1	2	3	....	n
j	n	n	n		n

⇒ n * n

⇒ T.C = $O(n^2)$

3.
```
for(int i=1; i<=n; i++){
  for(int j = 1; j<=i; j++){
    for(int k=1; k<=100; k++){
      s.o.pln ("abc");
    }
  }
}
```

i	1	2	3	.....	n
j	1	2	3	.....	n
k	1×100	2×100	.....		n×100

$= 100[1 + 2 + \dots + n]$

$= 100\left[\frac{n(n+1)}{2}\right]$

→ **T.C = $O(n^2)$**

4.
```
for(int i=1; i<n; i++){
  for(int j=1; j<i^2; j++){
    for(int k=1; k<=n/2; k++){
      s.o.pln("abc");
    }
  }
}
```

i	1	2	3	....	n
j	1	4	9	....	n^2
k	$1\times\frac{n}{2}$	$4\times\frac{n}{2}$	$9\times\frac{n}{2}$	....	$n^2\times\frac{n}{2}$

$= \frac{n}{2}[1 + 2^2 + 3^2 + \dots + n^2]$

$= \frac{n}{2}\left[\frac{n(n+1)(2n+1)}{6}\right]$

T.C = $O(n^4)$

5. for (int i=1; i<n; i=i*2){
a) S.o.pln ("abc");
}

i = 1, i = 2, i = 4, ..., i = 8
S S S S

The stopping condition when
$i \geq n$
$2^k >= n$
$\log_2 2^k = \log n$
$k = o(\log n)$

b) for (int i = 1; i < n; i = i*3){
S.o.pln ("abc");
}
T.C = $O(\log_3 n)$

6. for (int i=0; i^2<n; i++){
S.o.pln ("abc");
}

$i^2 < n$
$i < \sqrt{n}$
$\Rightarrow$ T.C = $O(\sqrt{n})$

7. for (int i = n/2; i<=n; i++){
for (int j = 1; j <= n/2; j++){
for (int k=1; k <= n; k=k*2){
S.o.pln ("abc");
}
}
}

$i \rightarrow n/2$
$j \rightarrow n/2$
$k \rightarrow \log n$
T.C = $n^2 \log n$

8. for (int i = n/2; i <= n; i++){
for (int j = 1; j <= n; j=j*2)
{
for (int k = 1; k <= n; k=k*2)
{ S.o.pln ("abc");
}
}
}

$i \rightarrow n$
$j \rightarrow \log n$
$k \rightarrow \log n$
T.C = $n \log n^2$

9.
```
for(int i=0; i<=n; i++){
    for (int j=1; j<=n; j=j+i)
    { for(int k=1; k<n; k=k
            s.o.pln("abc");
    }
}
```

$= n\left[1+\frac{1}{2}+\frac{1}{3}+\cdots+1\right]$

$= n(\log n)$

i	0	1	2	3 ...	n
j	0	n/0	n/2	n/3	... n/n

```
for(int i=0; i<n; i=i+2)
{
    s.o.pln("abc");
}
```

i	0	2	4	6	8	... n/2

$O(n/2) \Rightarrow O(n)$

Summary:

1. for(int i=0; i<n; i++) $\rightarrow O(n)$
2. for(i=0; i<n; i=i+2) $\rightarrow O(n/2) \rightarrow O(n)$
3. for(i=1; i<n; i=i*2) $\rightarrow O(\log_2 n)$
4. for(i=1; i<n; i=i*3) $\rightarrow O(\log_3 n)$
5. for(i=n; i>1; i=i/2) $\rightarrow O(\log_2 n)$

1)
```
i = 0
while (i < n) {
    stmt;
    i++;
}
```

0, 1, 2, ... k

$k > n$

$\Rightarrow O(n)$

2)
```
i = 1, k = 1
while (k < n) {
    stmt;
    k = k + i;
    i++;
}
```

i	1	2	3	.....	m
k	1	1+2	1+2+3	.....	1+2+3+...m

$\Downarrow$

$\frac{m(m+1)}{2} > n$

$m^2 > n$

$m = \sqrt{n}$

$\boxed{O(\sqrt{n})}$

3)
```
a = 1
while (a < b)
{
    stmt;
    a = a * 2;
}
```

1, 2, 4, ... 2^k

$2^k > b$

$T.C = (\log_2 b)$

Time Analysis of Recursive Programs:-

→
```
func(int x) {
    func(x-1)
}
```

* whenever we have a function like the function calling itself is called Recursive function.

1)
```
A(n)
{
  if (n>1)
  return A(n/2) + A(n/2)
```

Step 1:- Derive Recurrance Relation; Substitute A with T

$T(n) = T(n/2) + T(n/2)$

$T(n) = 2T(n/2) + c$

2)
```
A(n)
{ if (n>1)  ⇒ c=1
    return A(n-1);
}
```

$T(n) = T(n-1) + c \Rightarrow T(n-1) + 1$

Step 2: Calculate Time complexity by using any of one below methods.

1. Back Substitution method.
2. Recurssion Tree.
3. Masters Theorem.

Back Substitution:

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

Substitute (or) Calculate $n-1$

$T(n-1) = T(n-2) + 1 \longrightarrow (2)$

Calculate $n-2$ from eq (1)

$T(n-2) = T(n-3) + 1 \longrightarrow (3)$

Now, Substitute eq ② in eq ①

$$T(n) = T(n-2) + 1 + 1 \rightarrow ④$$

Sub eq ③ in eq ④

$$T(n) = T(n-3) + 1 + 1 + 1$$

$$T(n) = T(n-3) + 3.$$

$$T(k) = T(n-k) + k \qquad n-k = 1, \quad k = n-1$$

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

$$= n-1+1$$

$$T(n-1) = O(n)$$

3) $T(n) = n + T(n-1); \; n > 1 \rightarrow ①$

$$= 1 \; ; \quad n = 1$$

$$T(n-1) = n-1 + T(n-2) \rightarrow ②$$

$$T(n-2) = n-2 + T(n-3) \rightarrow ③$$

Sub ② in ①

$$T(n) = n + n-1 + T(n-2)$$

$$T(n) = n + n-1 + n-2 + T(n-3)$$

$$T(n) = 3n-3 + T(n-3)$$

$$= nk - k + T(n-k)$$

$$n-k = 1 \Rightarrow k = n-1$$

$$T(n-1) = k(n-1) + T(n-k)$$

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

$$= (n-1)^2 + T(1)$$

$$= O(n^2)$$

T.C

(or)

$$T(n) = n + n-1 + n-2 + \cdots + T(n-(k+1))$$

$$n-(k+1) = 1 \Rightarrow n-k-1 = 1$$

$$\Rightarrow n-k = 2$$

$$\Rightarrow k = n-2$$

$$T(n) = n + (n-1) + (n-2) + \cdots + T(n-(n-2+1))$$

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

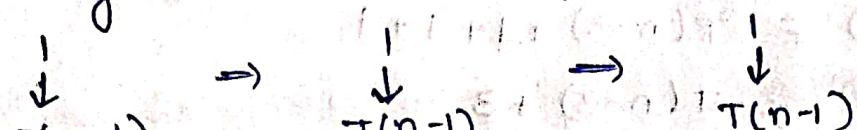
$$= \frac{n(n+1)}{2}$$

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

Recursion Tree:

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

we are doing a constant amount of work and then calling $T(n-1)$



1, 1, 1, ... T(1)

T.C ⇒ O(n)

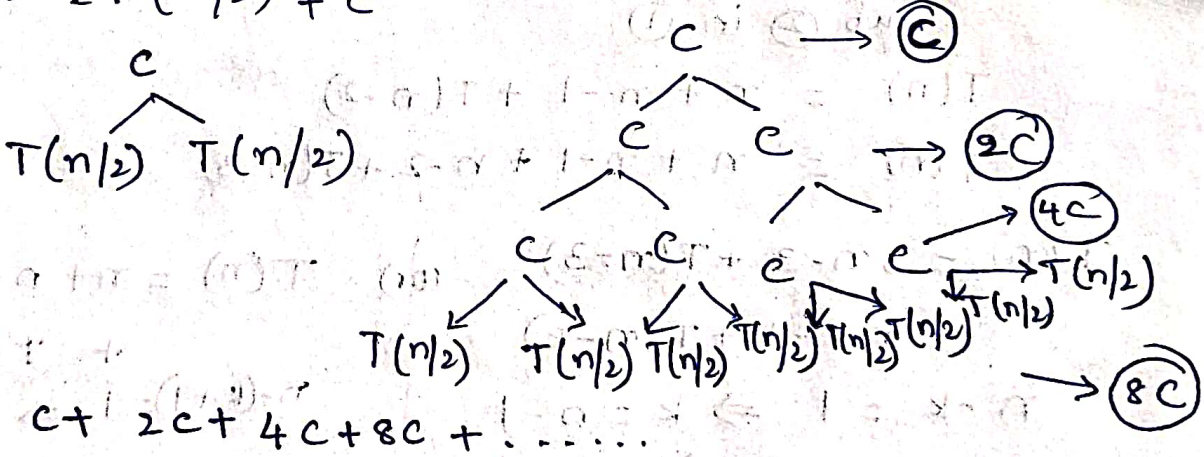
2) $T(n) = n + T(n-1)$; $n > 1$

1 ; $n = 1$

$\Rightarrow n + (n-1) + (n-2) + \cdots + 1$

$\Rightarrow \frac{n(n+1)}{2} \Rightarrow O(n^2)$

3) $T(n) = 2T(n/2) + c$



$\Rightarrow c + 2c + 4c + 8c + \cdots$

$\Rightarrow c[1 + 2 + 4 + 8 + \cdots]$

$\Rightarrow c[2^0 + 2^1 + 2^2 + 2^3 + \cdots]$

Sum of first G.P numbers $= \frac{a(r^n - 1)}{r - 1}$, say $n = k + 1$

$a = 2^0 = 1$, $r = 2$, $n = k+1$

$= c\left[\frac{a(r^{k+1} - 1)}{r - 1}\right]$

Sum (or) $n = 2^{1(K+1)} - 1$

$n + 1 = 2^{1(K+1)}$

$\log_2(n+1) = \log_2 2^{(K+1)} \Rightarrow K + 1 = \log_2(n+1)$

$k = \log_2(n+1) - 1$

Substituting the value of k

$T.C = C * 2^{\log_2(n+1)-1}$

$= C * (n+1-1)$

$T.C = O(n)$.

Masters Theorem:

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + \theta(n^k \log^p n)}$$

where n = size of the problem

a = number of sub problems in the recursion and $a >= 1$

n/b = size of each sub problem

$b > 1$, $k >= 0$ and p is a real number.

Then

1. if $a > b^k$, Then $T(n) = \theta(n \log_b a)$
2. if $a = b^k$, Then
   a) if $p > -1$ then $T(n) = \theta(n \log_b a \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \theta(n \log_b a \log\log n)$
   c) if $p < -1$ then $T(n) = \theta(n \log_b a)$
3. if $a < b^k$, then
   a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = \theta(n^k)$

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

$a = 3$, $b = 2$, $k = 2$, $p = 0$

$b^k = 2^2 = 4$ $\quad a < b^k$ & $p = 0$

So, $T(n) = \theta(n^k)$

$= \theta(n^2)$

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

$a = 2, b = 2, k = 0, P = 0$

$b^k = 2^0 = 1$

$\Rightarrow \Theta(n^{\log_b a}) \Rightarrow \Theta(n \log 2^2)$

$\Rightarrow \Theta(n)$

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

$a = 4, b = 2, P = 0, k = 2$

$a = b^k$

$\Theta(n^{\log_b a} \log^{P+1} n) \Rightarrow \Theta(n^{\log_2 4} \log n)$

$\Rightarrow \Theta(n^2 \log n)$

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

$a = 1, b = 2, P = 0, k = 2$

$1 < 2^2 \Rightarrow 1 < 4 \Rightarrow a < b^k$

$\Theta(n^k \log^P n) \Rightarrow \Theta(n^2 \log^0 n)$

$\Rightarrow \Theta(n^2)$

5) $T(n) = 16T\left(\frac{n}{4}\right) + n$

$a = 16, b = 4, P = 0, k = 1$

$16 > 4^1$

$\Rightarrow \Theta(n \log_b a)$

$\Rightarrow \Theta(n \log_4 16)$

$\Rightarrow \Theta(n \log_4 4^2)$

$\Rightarrow \Theta(n^2)$

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

$T(n-1) = T(n/2) = 2^{n/2} T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^{\frac{n}{2}}$

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

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

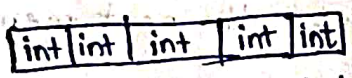
$$= 2^{\frac{3n}{2}} T\left(\frac{n}{4}\right) + n^{\frac{n}{4}} \times 2^{\frac{3n}{4}} + n^n$$

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

So $O(n) = 2^n$

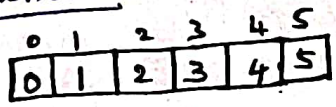
Arrays:

Collection of similar data elements stored at contiguous memory location.




```
int [ ] arr = new int [5];
```

Accessing array Elements:




```
int [ ] arr = new int [6]
```

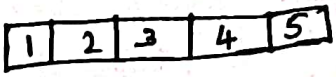
i.e create an array of size 6.

Elements in an Array can be Accessed through indexing

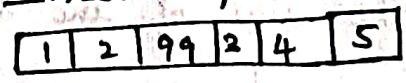
arr [1] :

Different Operations on Array:

a) Insertion



Insert 99



If We need insert an element at particular index, we need to right shift the elements, So T.C is O(n)

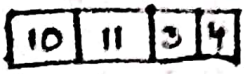
b) Searching:-

We need to iterate over the array.

Best Case Scenario T.C → O(1)

Worst Case Scenario T.C → O(n)

c) Deletion:



d) Accessing:

Elements can be accessed through indexing.

So T.C = O(1)

What happens in memory?

→ Memory address are hexadecimal. For simplicity we can consider it as decimal.

int [] arr = new int [4];

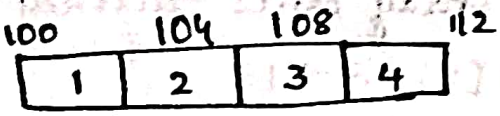
Reserve 4 integer spaces.

→ Integer takes 4 bytes, the next memory Address can be calculated using

= Initial MemAddress + (size of datatype × Element Number)

= 100 + 4 × 1

= 104



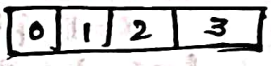
→ If the heap size is 100, but I have declared Max size as the size of the array, then we will encounter Out of Memory error.

i.e If the size of the array is more than the available heap size, then expect Out of Memory Error.

→ If we are trying to access an element index which is more than the array size then expect Array Index Out of Bound Exception.

Problems on Arrays:-

→ Print an Array :-



→ Iterate through the Array

→ Access the element through the index and print it.

```
for(int i=0; i<A.length; i++){
    System.out.pln("Element at"i +"is" + A[i]);
}
```

→ Sum of Array:

→ Max of Array:

```
A[] = [1, 4, 2, 3]
int max = Integer.MIN_VALUE;
for(int i=0; i<A.length; i++){
    if(A[i] > max){
        max = A[i]
    }
}
return max;
```

Second Max of Array:

1	4	2	5	5

```
max = Integer.MIN_VALUE;
SecondMax = Integer.MIN_VALUE;
for(int i=0; i<A.length; i++){
    if (arr[i] > max){
        SecondMax = max;
        max = arr[i];
    }
    else if (arr[i] > SecondMax && arr[i] < max)
    {
        Second Max = arr[i];
    }
}
```

Dry Run:

1) Smax = -∞
   max = 1

2) 4 > 1
   Smax = 1
   max = 4

3) 2 < 4,
   So in else if
   2 > 1 && 2 < 4
   Sm = 2, max = 4

4) 5 > 4
   Smax = 4
   max = 5

5) 5 == 5
   So in else if
   5 > 4 && 5 ≠ 5
   So
   Smax = 4

Third Max Element:

```
max = Integer.MIN_VALUE;
Second max = Integer.MIN_VALUE;
thirdMax = Integer.MIN_VALUE;
for(int i=0; i<A.length; i++){
    if (A[i] > max) {
        thirdMax = SecondMax;
        SecondMax = max;
        max = A[i];
    }
    else if(A[i] > SecondMax && A[i] < max)
    {
        thirdMax = SecondMax;
        SecondMax = A[i];
    }
    else if (A[i] > ThirdMax && A[i] < SecondMax)
    {
        thirdMax = A[i];
    }
}
```

→ IsAscending

[1, 2, 3, 4, 5] o/p: true

```
for(int i=0; i<A.length; i++){
    if(A[i] < A[i+1]){
        p++;
        q++;
    }
    else{
        return false;
    }
return true;
```

→ nth element of Fibonacci Series:-

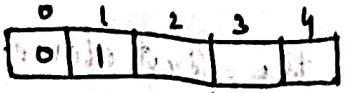
Fibonacci Series - 0, 1, 1, 2, 3, 5, 8 ...

if n = 4

If we create a new array

```
int[] ans = new int[n+1];
```

i.e with extra space and ans[0] = 0, ans[1] = 1



ans[2] = ans[1] + ans[0]

ans[x] = ans[x-1] + ans[x-2]

```
for(int i=2; i<ans.length; i++)
    {
      ans[i] = ans[i-1] + ans[i-2];
    }
return ans[n];
```

T.C → O(n)

S.C → O(n)

Without Extra Space:

```
firstElement = 0
secondElement = 1
if (n == 0) return 0; else if (n == 1) return 1;
for (int i = 2; i < n; i++)
{
    thirdElement = firstElement + secondElement;
    firstElement = secondElement;
    secondElement = thirdElement;
}
return thirdElement;
```

Merge Two Sorted Arrays:

arr1 | 1 | 3 | 5 | 7 | arr2 | 2 | 4 | 6 | 8 |

O/P: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | → To sort

T.C → O(n) + O(n) = O(2n) = O(n) + ~~O(n log n)~~

S.C → O(n) = O(n) ~~log n~~

→ Use 2 Pointer Approach:

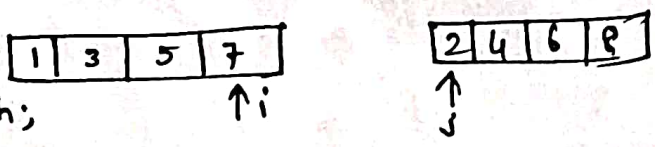
1. Create 2 pointers pointing to the first element of both arrays.
2. Compare the elements being held by the pointers:

```
if (arr1[i] < arr2[j]) {
    add the element to resultant array
    i++;
}
else {
    add the element to array
    j++;
}
```

3. Handle the case if any of the pointer reaches the end of array by appending all the values of the other array.

```
int[] result = new int[m+n];
int i=0; int j=0; int k=0;
while (i < m && j < n) {
  if (arr1[i] < arr2[j])
  {
    result[k] = arr1[i];
    k++; i++;
  }
  else
    result[k] = arr2[j];
    k++; j++;
  }
while (i < m)
  {
    result[k] = arr1[i];
    i++; k++;
  }
while (j < n)
  {
    result[k] = arr2[j];
    j++; k++;
  }
```

Without Extra Space:-




```
int i= array1.length;
  j= 0;
while(j < array1.length && i > 0) {
  if(array1[i] > array2[j]{
    int temp = array1[i];
    array1[i] = array2[j];
    array2[j] = temp
    i--; j++;
  else{ break; }}
```

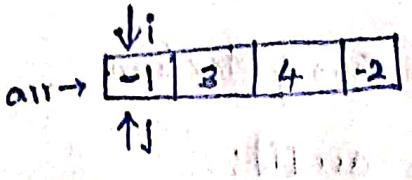
Arrays.sort(array1);

Arrays.sort(array2);

O(n) + O(n log n)

T.C = O(n log n), S.C = O(1)

Maximum SubArray Sum (or) Largest Continuous Sum:



maxSum, Sum

Initial val of maxSum = $-\infty$

Sum = arr[0];

Algo:-

1. Find out Sub Arrays
2. Calculate Sum of each Subarray
3. And update the maxSum.

Seudo Code:

```
maxSum = 0
for (i = 0 to len(arr),)        ;i++
  {
    Sum = 0                     ;j++
  for (j = i to len(arr))
    {
      Sum += arr[j];
      maxSum = Math.max(sum, maxSum);
    }
  }
  return maxSum;
```

T.C → $O(n^2)$, S.C $O(1)$

Sum = -1+3 = -2

maxSum = $-\infty$ < -2

= -2

Sum = -1+3+4 = 6

maxSum = -2 < 6

= 6.

Sum = -1+3+4-2

= 4

maxSum = 6 < 4

= (6)

j = 1

sum = 3;

maxSum = 6 > 3

= 6

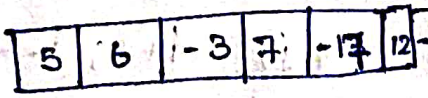
sum = 3+4 = 7

maxSum = 6 < 7

= (7)

Kadane's Algorithm:-

↳ To find the Largest Sum in a SubArray with T.C O(n)



Initial maxSum, sum = arr[0]

max Sum = 5

① = 11 > 5, So max = 11

② = 11 < 8 So max = 11

③ = 11 < 15 So max = 15

④ = 15 > -2 So max = 15

① 5 + 6 = 11

② 5 + 6 - 3 = 8

③ 5 + 6 - 3 + 7 = 15

④ 5 + 6 - 3 + 7 - 17 = -2

⑤ As Sum is -ve, So Sum = 0

Codes & Algo:

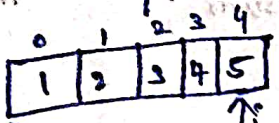
① Calculate cummulative Sum, sum = arr[0], maxSum = arr[0]

② update MaxSum by Comparing Sum & MaxSum

③ If Sum is -Ve value, Sum = arr[i];

```
int Sum = arr[0];
int maxSum = arr[0];
for (int i = 1; i<arr.length; i++){
    if (sum > 0) {
     Sum + = arr[i];
    }
    else {
      sum = arr[1];
    }
    maxSum = Math.max (maxSum, sum);
}
return maxSum;
```

Pair Sum: * No Duplicates.



K = 6

o/p:- [0, 4], [1, 3]

Return the indexes of the pairs.

```
int i=0; j=arr.length-1;
while (i < arr.length && j>1) {
    int value = arr[i] + arr[j];
    if (value == k){
    S.O.plm (i + " " + j); i++; j--;
    }
    else if (value < k) {
        i++;
    }
    else
        j--;
```

Count Frequency of Each Element:

→ There is a concept called Frequency distribution.

i/p: 1, 2, 2, 1, 1, 2, 5, 2

o/p: 1 → 3
2 → 4
5 → 1

② Algorithm:

→ Create a frequency array with max value of array.
→ Traverse the array and update the value in freq. Array with index from main array.
→ Traverse the array if count > 0.

```
int max = Integer.MIN_VALUE;
for(int i=0; i< arr.length; i++){
    max = Math.max(arr[i], max);
}
int[] freqArray = new int[max+1];
for(int i=0; i< arr.length; i++){
    int index = arr[i];
    freq[index] += 1;
}
for(int i=0; i< freqArray.length; i++){
    if(freqArray[i] > 0)
    {
        S.O.pln(i + " :" + freqArray[i]);
    }
}
```

Best Time to Buy and Sell Stock

I/P → 7 1 5 3 6 4 O/P: 0

M T W Th F Sat → Price of Stock on each day

e.p = 1, s.p = 6, P = 6 - 1 = 5

minArray[] = | 7 | 1 | 1 | 1 | 1 | 1 | → Before each day what was the minimum price of each stock.

ansArray[] = | 0 | 0 | 4 | 2 | 5 | 3 |

maxProfit

→ Buy it on that day price & Sell it with minimum day price of stock of the previous day.

i.e Buy it on lowest Price and Sell it on the highest Price.

| 7 | 4 | 5 | 3 | 1 |

| 7 | 4 | 4 | 3 | 1 | minArray

| 0 | 0 | 1 | 0 | 0 | ansArray

7 1 5 3 6 4

| 7 | 1 | 1 | 1 | 1 | 1 |

| 0 | 0 | 4 | 2 | 5 | 3 |

```
for (int i = 0; i < arr.length; i++)
{
    minSoFar = Math.min (minSoFar, arr[i]);
    int profit = arr[i] - minSoFar.
    maxProfit = Math.max (maxProfit, profit);
}
```

O(n)

Segregate the Array:

I/P: -1, 1, -2, 3, 4, -5

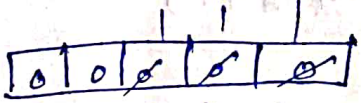
O/P:- -1, -2, 5, 1, 3, 4

Similar approach as Move Zeros to End.

Missing First Positive Number:

I/p:- -1, -1, 2, 3, 4

use frequency distribution approach:



TC → O(n)

ans: (1) S.C → O(n)

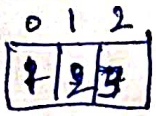
10 | 11 | 12 → missing number → (1)

1 | 2 | 3 → missing number → length+1

→ 3+1=4

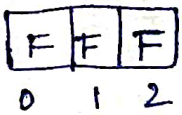
First missing +ve number will always lie b/w 1 to n+1.

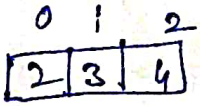
Coloring:



int index = arr[i]

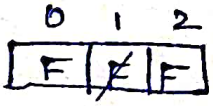
= 2



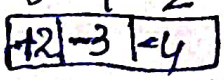


abs = 2

index = 2-1 = 1



T



3-1 = 2

4-1 = 3

index +1

0+1

= 1

```
int n = nums.length;
for (i to n){
  if (arr [i] <= 0 || arr[i]> n)
    {
      arr [i] = n+1;
    }
}
    for (i = to n) {
      int id = Math.abs (arr[i]);
        if (id >n) continue; id--;
        if (arr[id]>0) arr[id] = -arr[id];
```

```
for (int i=0; i<n; i++)
{
    if (arr[i] > 0)
        return i+1;
}
return n+1;
```

Majority Element:

i.e element present more than $n/2$ times

i/p → {5, 1, 4, 1, 1} $n = 5 \Rightarrow n/2 \Rightarrow 5/2 = 2$

Count of 1 is 3

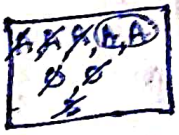
So majority element is 3.

a) Using Frequency Distribution:

	0	1	2	3	4	5
freqArr	0	3	0	0	1	1

```
→ for(int i=0; i<freqArr.length; i++)
  {
      if (arr[i] > n/2) {
          return i;
      }
  }
```

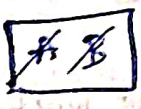
b) Moore's Voting Algorithm:-

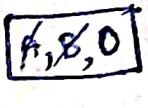


8 elements are there in the bag.

$8/2 = 4$

→ If one element occurs $n/2$ times, all the other elements will occur less than $n/2$ times.





⇒ no majority element.

```
arrIndex = 0,
count = 1
for (int i=0; to ?)
{  if (arr[i] == arr[arrIndex]) {
       count ++;
   }
   count--;
   if (count == 0) {
       arrIndex = i;
       count = 1;
   }
}
```

2, 2, 1, 3, 1, 2, 2

checking the count of majority elem

```
ansCount = 0;
for (int i = 0; i<n; i++)
{
if(arr[arrIndex] = arr[i])
   ansCount++;
}
if(ansCount > n/2)
   arr[arrIndex];
```

Sort an Array of 0, 1, 2 (Dutch National Flag)

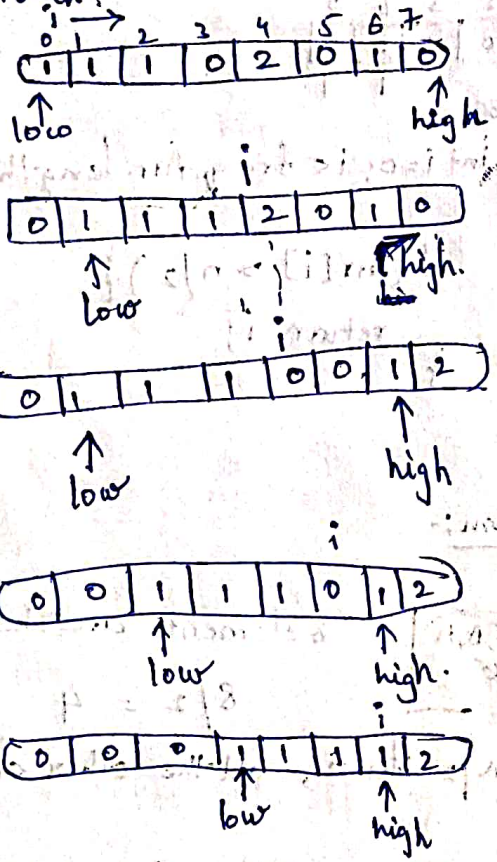
{1, 1, 1, 0, 2, 0, 1, 0}

a) Approach 1:

Count Zero = 3, Count 1 = 4, Count 2 = 1

and then update the new array

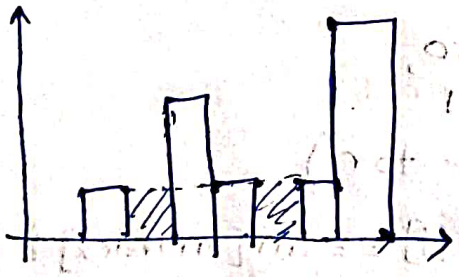
b) Two Pointer Approach:

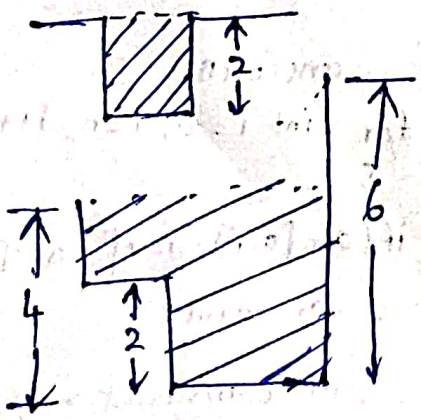


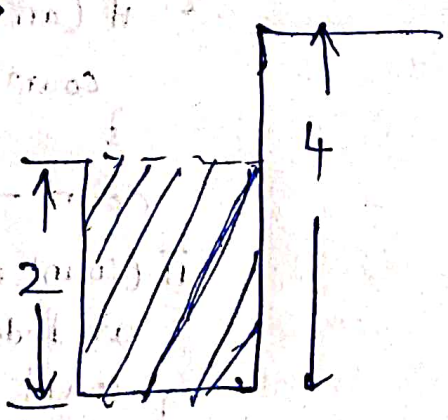

```
while(i<=high) {
if (arr[i]==1)
{
  i++;
}
else if(arr[i]==0)
{
  swap(arr[i], arr[low]
  i++;
  low++;
}
else {
  swap(arr[i], arr[high]
  high--;
}
}
```

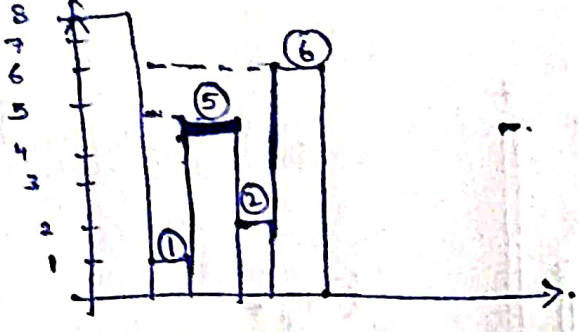
Trapping Rain Water:

How much water can be trapped after raining?









min(6,6) - 1 ⇒ 5

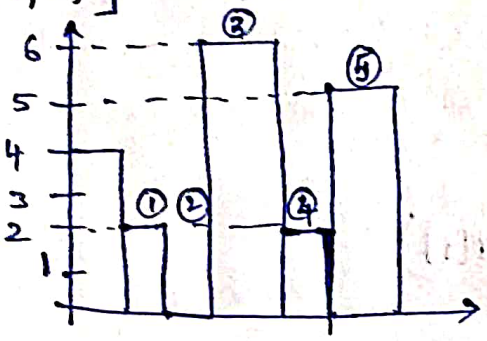
min(6, 6) - 5 ⇒ 1

min(5, 6) - 2 ⇒ 3

i.e $\min(l_{max}, r_{max}) - arr(i)$

$5 + 1 + 3 = 9$

[4,2,0,6,2,5]



1) min (4, 6) - 2 = 4 - 2 = 2
2) min (4, 6) - 0 = 4 - 0 = 4
3) min (6, 5) - 6 = 5 - 6 = -1 ✗
4) min (6, 5) - 2 = 5 - 2 = 3
5) min (6, 5) - 5 = 5 - 5 = 0

$2 + 4 + 3 = 9$

	0	1	2	3	4	5
arr[] =	4	2	0	6	2	5
l_{max} =	4	4	4	6	6	6
r_{max} =	6	6	6	6	5	5

```
ans += Math.min(lmax(i) - rmax(i)) - arr[i];
```

T.C → O(n), S.C → O(n)

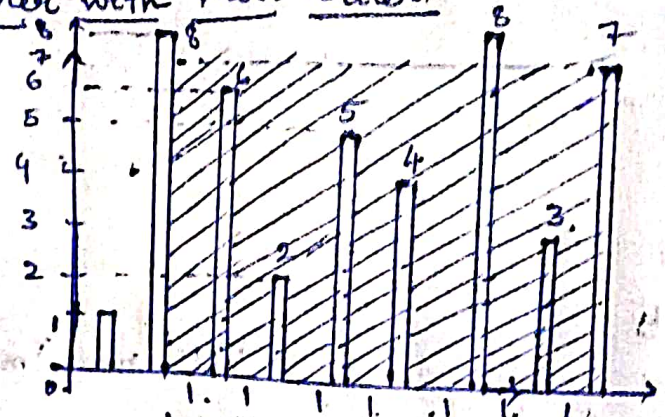
b) Two Pointer Approach:

leftmax = 0, rightmax = arr.length-1; ans = 0

4	2	0	6	2	5
↑ l					↑ r

```
while (left < right)
{
  leftMax = Math.max(leftMax, height[left]);
  rightMax = Math.max(rightMax, height[right]);
  if(leftMax < rightMax)
    {
      ans += leftMax - height[left];
      left++;
    }
  else
    {
      ans += rightMax - height[right];
      right--;
    }
}
return ans;
```

Container with Most Water:



height[] = [1, 8, 6, 2, 5, 4, 8, 3, 7]

l = 7
b = 7
a = l * b
= 7 * 7 = 49

Approach 1:

The max area of water

```
for(int i=0; i<arr.length; i++){
  for(int j=i+1; j<arr.length; j++){
    int length= Math.min(arr[i], arr[j])
    int breadth = j-i
    currentArea = l x b;
    MaxArea = Math.max(CurrentArea, maxArea);
  }
}
```

$O(n^2)$

Approach: ②

```
while(left<right){
  int
  if(height[left] < height[right]){
    left++;
  }
  else{
    right++;
  }
}
return maxArea;
```

$O(n)$

30/9/2023

Sliding Window:

arr = | 2 | 3 | 5 | 2 | 9 | 7 | 1 |

→ Find sum of all sub-arrays of size 3.

Sub-array:- any continuous elements form a sub-array.

2, 3, 5 = 10
3, 5, 2 = 10
5, 2, 9 = 16
2 9 7 = 18
9 7 1 = 17

Brute Force:

```
for(int i=0; i<arr.length-k; i++){
    for(j=i; j<i+k; j++) {
        sum += arr[j];
    }
    S.O.println(sum);
}
```

2 + [3 + 5] = 10
3 + [5 + 2] = 10
5 + [2 + 9] = 16
2 + [9 + 7] = 18
9 + 7 + 1 = 17

Step1:- Make your window reach till the window size.

Step2: Once you have reached the window size, do your calculation.

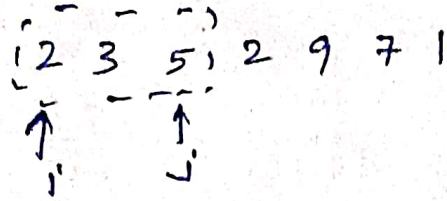
| 2 | 3 | 5 | 2 | 9 | 7 | 1 |

a) 2+3+5 = 10
b) 2+3+5 + 2 - 2 = 10 (newvalue = 2; subtract the value at window start.)
c) 3+5+2+9-3 = 16
d) 5+2+9+7-5 = 18
e) 2+9+7+1-2 = 17

→ Consider Sliding Window Technique, You see Sub-array, Maximize (or) Minimize.

→

Sliding Window
- Fixed (window size K=3)
- Variable (window size, needs to be calculated)



→ Initialize two variables

i → window start, j → window end (Traversing variable)

```
while(j<arr.length){
    sum+=array[j];
    if(j-i+1==k){ → Print the Sum here.
    sum-=array[i];
    i++;
    }
    j++;
}
```

→ Maximum Sum of k Consecutive Elements:

```
int i=0, j=0, sum=0, Max=Integer.MIN_VALUE;
    while(j<arr.length){
        sum+=arr[j];
        if(j-i+1==k){
            max=Math.max(sum,max);
            sum-=arr[i]; i++;
        }
        j++;
    }
    return max;
```

→ Given a String, and non-empty string P, find all P's anagrams in S. String contains only lowercase letters.

I/P: s = "cbaebabcd", P = "abc", O/P: 2.

slient → listen

→ You are able to create a meaningful word then the word is said to be an anagram.

→ S = cabaebabcd , P = abc

Count = 0 * (2)

ex ①:

a a b c → a = 2, b = 1, c = 1

b a c a → a = 2, b = 1, c = 1

These are anagrams.

ex ②:

a b c d → a = 1, b = 1, c = 1, d = 1

a b b c → a = 1, b = 2, c = 1

Not anagrams.

→ P. Arr

0	1	2
1	1	1
a	b	c

→ update the frequency of character

S arr

0	1	2		
1	1	1		
a	b	c	d	

→ We need to iterate over the P array & check the count of each character.

Approach ①:

```
for(int i=0; i<PArr.length; i++){
    if(PArr[i] != SArr[i]){
        return false;
    }
}
return true;
```

Approach (2):

```
int k = p.length(); int j=0; int i=0;
int[] sArr = new int [26];
int[] pArr = new int [26];
int count = 0;
for(int i=0; i< p.length(); i++){
    pArr [p.charAt(i) - 'a']++;
}
while (j < s.length()){
    sArr [s.charAt(j) - 'a']++;
    if (j - i + 1 == k) {
        if (isAnagram (sArr, pArr)
        {
            count++;
        }
        // slide the window
        sArr [s.charAt(i) - 'a']--;
        i++;
    }
    j++;
}
```

p = abc

s = cabaebabcd

k = p.length = 3

```
boolean isAnagram(sArr, pArr){
    for (int i=0; i< pArr.length; i++)
    { if (pArr[i] != sArr[i]){
        return false;
    }
    return true;
}
```

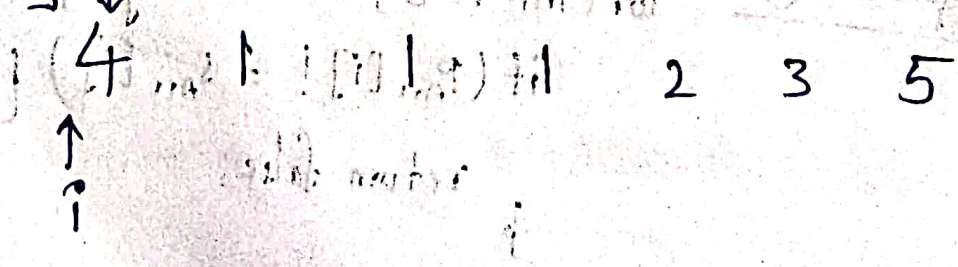
→ Longest SubArray of Size k: (Variable Window Size)

→ Given an array, find the longest subarray of size k.

I/P: {4, 1, 1, 1, 2, 3, 5} k=8 , O/P: 5

↳ k is sum

That longest array is 1, 1, 1, 2, 3, so the length is 5.



```
int i=0; int j=0; int currentSum=0;
int longestWindow=Integer.MIN_VALUE;
while(j<arr.length){
  currentSum+=arr[j];
  if(currentSum==k){
    longestWindow=Math.max(longestWindow, j-i+1);
  }
  else{
    while(currentSum>k){
      currentSum-=arr[i];
      i++;
      if(currentSum==k){
        longestWindow=Math.max(longestWindow, j-i+1);
      }
    }
  }
  j++;
}
return longestWindow;
```

→ Smallest SubArray:

Find the length of Smallest SubArray whose length is >k.

Arr = {4, 2, 2, 7, 8, 1, 10};

k = 8

O/P: 1

```
int i=0; int j=0;
int minLength = Integer.MAX_VALUE;
int currentWindowSum = 0;
while (j < arr.length) {
    currentWindowSum += arr[j];
    while (currentWindowSum > k)
    {
        minLength = Math.min(minLength, j-i+1);
        currentWindowSum -= arr[i];
        i++;
    }
    j++;
}
return minLength;
```

→ Max Consecutive One's if at most 1 flip is allowed.

I/P: {1, 1, 0, 0, 1, 1, 1, 1, 1} k = 1, O/P: 6

```
int zeroCount = 0; int i = 0; int j = 0;
int maxConsecutiveOne = 0;
while (j < arr.length) {
    if (arr[j] == 0) {
        zeroCount++;
    }
    while (zeroCount > k) {
        if (arr[i] == 0) {
            zerCount--;
        }
        i++;
    }
    maxConsecutiveOne = Math.max(maxConsecutiveOne, j-i+1);
    j++;
}
return maxConsecutiveOne;
```