ARRAY ADT

↳ Abstract Datatype

↓

1. Representation of data
2. Operations on data

DATA

1. Array Space
2. Size
3. Length (No of elements)

① `int A[10];`

② `int *A;`
`A = new int[size];`

OPERATIONS

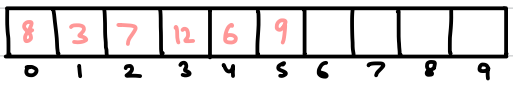
1. Display()
2. Add(x) / Append(x)
3. Insert(index, x)
4. Delete(index)
5. Search(x)
6. Get(index)
7. Set(index, x)
8. Max(), Min()
9. Reverse()
10. Shift() / Rotate()

1. DISPLAY

pseudo code

```
for(i=0; i<length; i++)
{
        print(A[i])
}
```

Array Size = 10
Length = 6



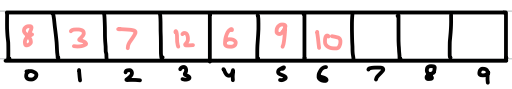
2. Add(x) / Append(x)

```
A[Length] = x;      ——— 1
Length++;           ——— 1
```

f(x) = 2
f(n) = 2

O(n°) = O(1)

Array Size = 10
Length = 7

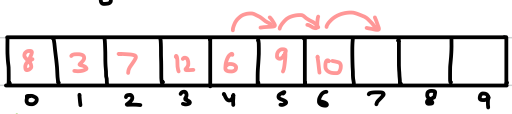


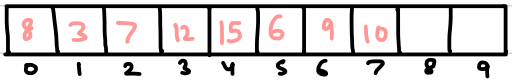
3. Insert (4, 15) — index = 4, x = 15

```
for (i = length ; i > index ; i--)
{
    A[i] = A[i-1] ;        — 0-n
}
A[index] = x ;             — 1
Length++ ;
```

Inserting at index 8 ↵ ↓ Inserting at 0

Array Size = 10
Length = 8





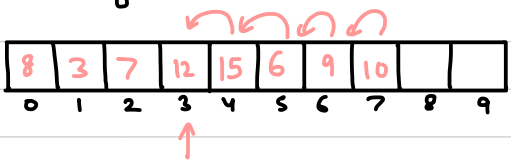
O(1) min O(n) max

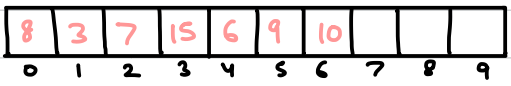
4. Delete (3) — index = 3

```
x = A[index] ;             — 1
for ( i = index ; i <= Length-1 ; i++)
    A[i] = A[i+1] ;        — 0-n
Length-- ;                 — 1
```

min = 2
max = n+2

Array Size = 10
Length = 7



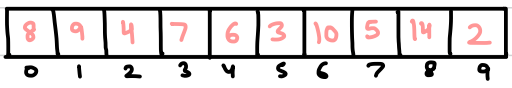


Best O(1) Worst O(n)

Index should be in range of length
we cannot leave empty space between two elements in an array.

LINEAR SEARCH → Searching each element and incrementing

Array Size = 10
Length = 10



Key = 5 ← Succesful

Key = 12 ← Unsuccesful

```
for (i=0 ; i < length ; i++)
    if (key == A[i])
        return i ;
return -1 ;
```

Best — O(1)
Worst — O(n)
Average — O(n)

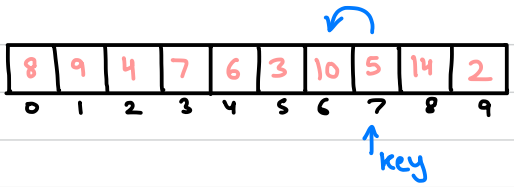
Average

found at location 1
found at 2

$$\frac{1+2+3+\dots n}{n} = \frac{(n+1)n}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

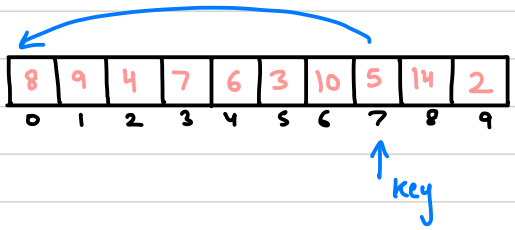
FASTER WAY OF LINEAR SEARCH

1. Transposition : Moving the searched element one step back




```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[i-1]);
            return i-1;
        }
}
```

2. Move to front / head : The searched element is brought to first position in array.

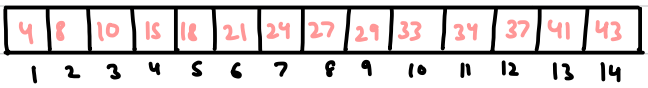



```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[0 ]);
            return 0;
        }
}
```

BINARY SEARCH

Size = 15
Length = 15

A



Sorted Array

ITERATIVE VERSION

```
Algorithm BinSearch (l, h, key)
{
    while (l <= h)
    {
        mid = [(l+h)/2];

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            h = mid - 1;
        else
            l = mid + 1;
    }
    return -1;      # key not found
}
```

Binary search is faster than linear search

RECURSIVE VERSION

Tail Recursion

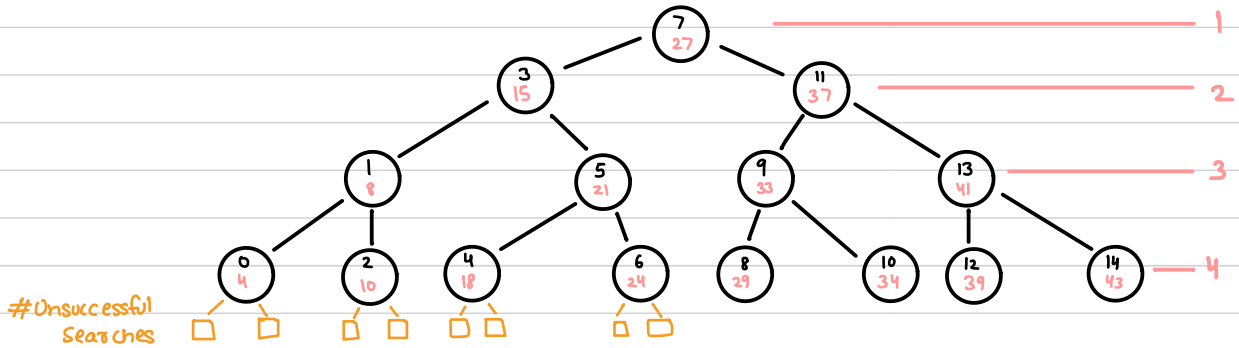
```
Algorithm RBinSearch (l, h, key)
{
    if (l <= h)
    {
        mid = [(l+h)/2];

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            return RBinsearch (l, mid - 1, key);
        else
            return RBinsearch (mid + 1, h, key);
    }
    return -1;      # key not found
}
```

ANALYSIS OF BINARY SEARCH

Size = 15
Length = 15

A

1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	8	10	15	18	21	24	27	29	33	34	37	41	43



Best — min — O(1)
Worst — max — O(logn)
Unsuccesful — max — O(logn)

Why logn ?

logn

Let size of array be 16

16 / 2 / 2 / 2 / 2

$2^4 = 16$

$4 = \log_2 16$

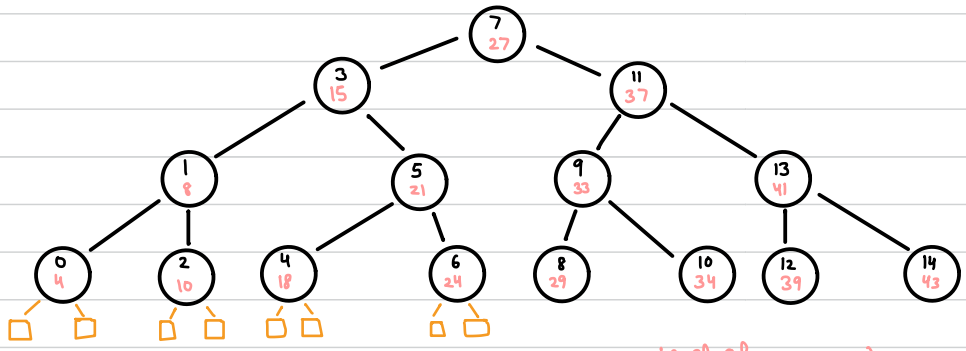
$\log_2 n$

Inverse of power is log.

16 is divided by 2 multiple times

AVERAGE CASE ANALYSIS OF BINARY SEARCH *

$$= \frac{\text{Total time taken in all possible cases}}{\text{Number of cases}}$$



$1 + 1\times2 + 2\times4 + 3\times8$

No of ↵ ↳ No of elements Comparisons

$1 + 1\times2^1 + 2\times2^2 + 3\times2^3$

$$\sum_{i=1}^{\log n} \frac{i \times 2^i}{n} = \frac{\log n \times 2^{\log n}}{n}$$

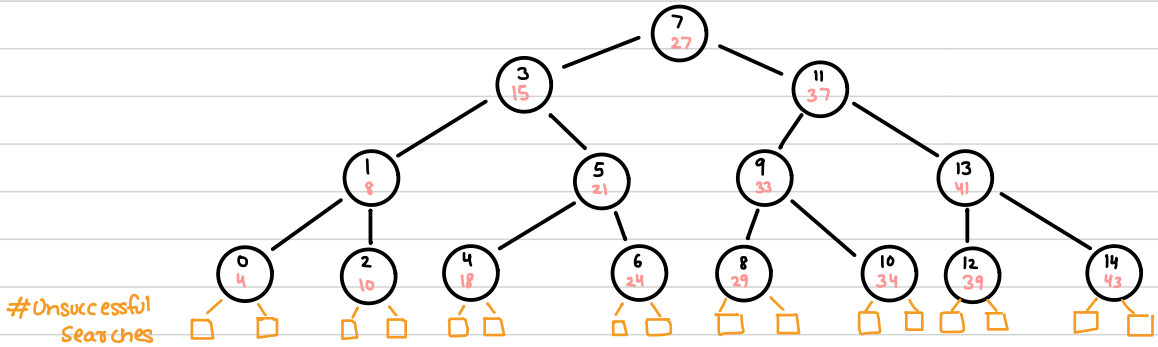
logn → Level of tree (Here 4)

i is replaced with logn

Same as formula *

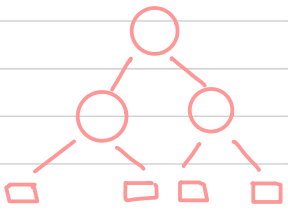
$$= \frac{\log n \times n^{\log 2}}{n}$$

$$= \log n$$



I = sum of all internal nodes
E = sum of all external nodes

$E = I + 2n$



n = 3
I = 2
E = 2x4 = 8

$E = I + 2n$

i = num of internal nodes
e = num of external nodes

n = num of nodes

$e = i + 1$

Average successful time for n elements

$$A_S(n) = 1 + \frac{I}{n} \Rightarrow 1 + \frac{E-2n}{n} \Rightarrow 1 + \frac{n\log n}{n} - 2$$

$$\Rightarrow 1 + \frac{E}{n} - 2 \Rightarrow \log n$$

Average unsuccessful time

$$A_U(n) = \frac{E}{n+1} = \frac{n\log n}{n+1} = \log n$$

$E = n\log n$
$E = I + 2n$
$I = E - 2n$

6. **Get (index) :**

```
if (index >= 0 && index < length)        O(1)
    return A[index];
```

7. **Set (index, x)**

```
if (index >= 0 && index < length)        O(1)
    A[index] = x;
```

8. Max()

```
max = A[0];                          ——— 1
for (i = 1; i < length; i++)         ——— n
{
        if (A[i] > max)              ——— n-1
            max = A[i];
}
return max;                          ——— 1
                                     2n+1   O(n)
```

9. Min()

```
min = A[0];
for (i = 1; i < length; i++)
{
        if (A[i] < min )
            min = A[i];
}
return min;
```

10. Sum()

```
Total = 0;                           ——— 1
for (i = 0; i < length; i++)         ——— n+1
        Total += A[i]                ——— n
return total                         ——— 1
                                     2n+3
                                     O(n)
```

$$Sum(A,n) = \begin{cases} 0 & n < 0 \\ Sum(A,(n-1)) + A[n] & n >= 0 \end{cases}$$

11. Avg()

```
Total = 0;
for (i = 0; i < length; i++)
        Total += A[i]
return total/n;
```

RECURSIVE CALL

```
int Sum (A, n)
{
    if (n < 0)
        return 0;
    else
        return sum (A, n-1) + A[n];
}
```

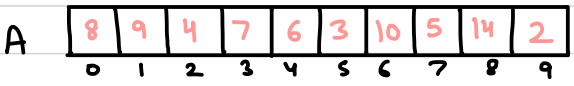
Sum (A, length-1) — call

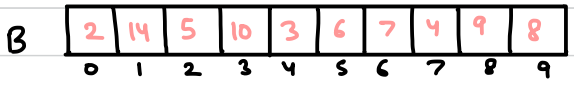
REVERSE AND SHIFT AN ARRAY

1. Reverse
2. Left Shift
3. Left Rotate
4. Right Shift
5. Right Rotate

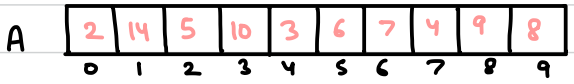
1. Reversing an array

Reversing using auxillary array B.





A → B n



B → A n

2n

O(n)

```
for (i = length-1; j=0; i>=0 ; i-- , j++)
        B[j] = A[i];
```

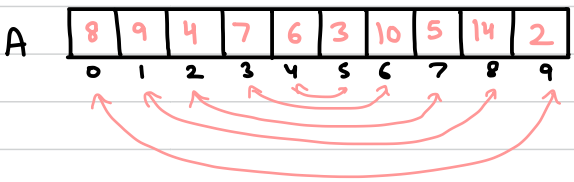
] Reverse copying array

```
for (i=0 ; i < length ; i++)
        A[i] = B[i];
```

ANOTHER METHOD OF REVERSING ARRAY

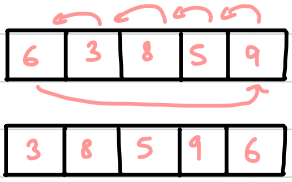
```
for ( i=0 ;  j = length-1 ; i<j ; i++ ; j--)
{
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
}
```

O(n)



2. Left Shift

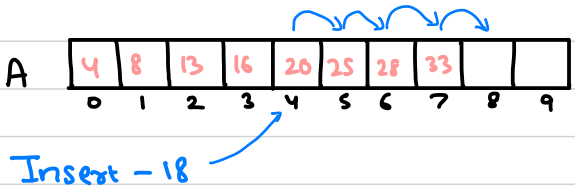
3. Left Rotate



CHECK IF ARRAY IS SORTED

1. Inserting in an sorted array
2. Checking if array is sorted
3. Arranging -ve elements on left side and +ve on right side.

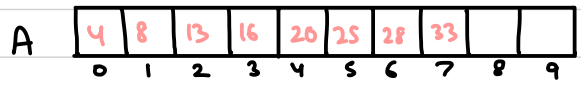
1. Inserting in an sorted array



Starting from last element, loop will run until the element is greater than the element to be inserted

```
x = 18 ;
i = length - 1 ;
while ( A[i] > x )
{
      A[i+] = A[i];
      i-- ;
}
A[i+1] = x ;
```

2. Checking if array is sorted




```
Algorithm isSorted ( A, n)
{
      for (i=0 ; i<n-1 ;i++)
      {
            if ( A[i] > A[i+1])
                  return false;
      }
      return true;
}
```

We will check false condition first by checking if any element is greater than its next element.

O(n) - Max Worst
O (1) - Min Best

3. Arranging -ve elements on left side and +ve on right side.

```
i=0;
j= length-1;

while (i<j)
{
        while (A[i]<0)
            i++;
        while (A[j]≥0)
            j--

        if (i<j)
            swap(A[i], A[j])
}
```

A

-6	3	-8	10	5	-7	-9	12	-4	2
0	1	2	3	4	5	6	7	8	9

i, j

A

-6	-4	-8	-9	-7	5	10	12	3	2
0	1	2	3	4	5	6	7	8	9

j, i

$O(n)$ $n+2$ comparisons are made

MERGING ARRAYS

1. Append / Concat
2. Compare
3. Copy

```
i=0;
j=0;
k=0;
while (i<m && j<n) ⟶ # When either of i or j have reached end of array.
{
    if (A[i]<B[j])
    {
        C[k]=A[i];
        k++;
        i++;
    }
```

Can also be written as `C[k++]=A[i++];`

m

A

3	8	16	20	25
0	1	2	3	4

i

n

B

4	10	12	22	23
0	1	2	3	4

j

C

3	4	8	10	12	16	20	22	23	25
0	1	2	3	4	5	6	7	8	9

k

```
    else
    {
        C[k]=B[j];
        k++;
        j++;
    }
```

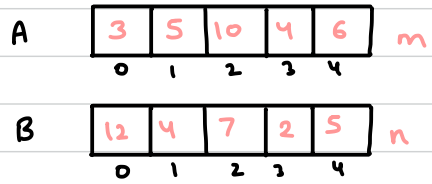
`C[k++]=B[j++]`

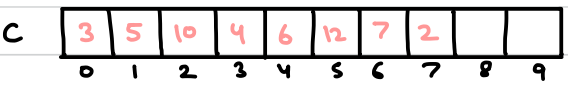
$\Theta = (m+n)$ ↳ Time is known

```
    for(; i<m; i++) ⟶ # When some elements in either of array are remaining to be copied.
        C[k++]=A[i];
    for(; j<n; j++)
        C[k++]=B[j];
}
```

SET OPERATIONS

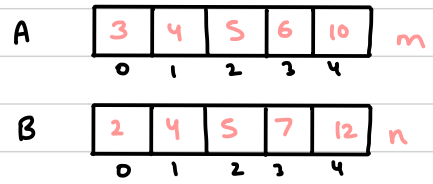
1. UNION

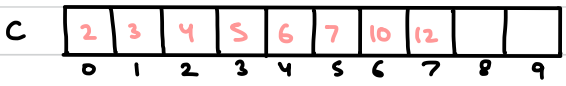




- Copy all elements of A in C
- Then copy elements of B which are not there in C.

$m + m*n$
$n + n*n$
$n + n^2$
$O(n^2)$

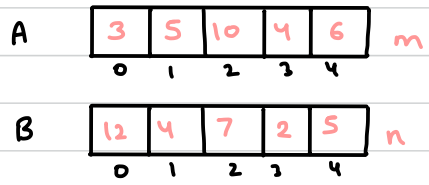


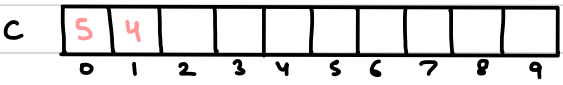


- Use merge procedure
- Copy the smaller element to C
- If same element, copy once
- Use i, j, k method

$\Theta(m+n)$
$\Theta(n+n)$
$\Theta(n)$

2. INTERSECTION

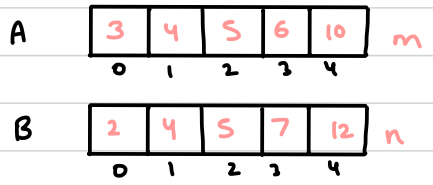


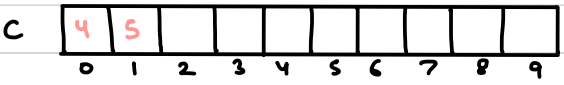


Copy the common elements of A and B to C

- One by one, take each element of A If it is present in B, copy it to C otherwise move on to next element

$n*m$
$n*n$
$O(n^2)$

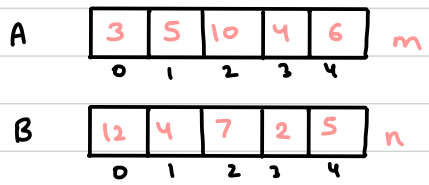




- Use merge method
- If element is smaller, don't copy.
- If element is same, copy
- Not merging but similar to it.

$\Theta(m+n)$
$\Theta(n)$

3. DIFFERENCE



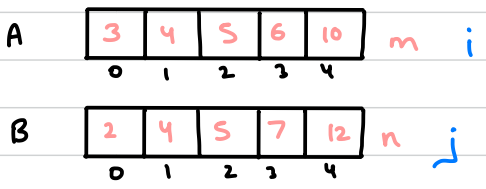
A-B

- We want elements of A which are not there in B.
- One by one, take each element of A If it is not present in B, copy it to C, otherwise move to next element

$m * n$

m^2

$O(n^2)$



- Use merge procedure.
- Compare A[i] and B[j], if A[i] is small, copy it to C otherwise increment j.
- If same, don't copy.

$\theta(m+n)$

$\theta(n)$

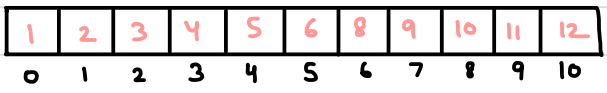
FIND MISSING ELEMENT

1. Single missing element in an sorted array.
2. Multiple missing element in an sorted array.
3. Missing element in unsorted array.

1. Single missing element in an sorted array.

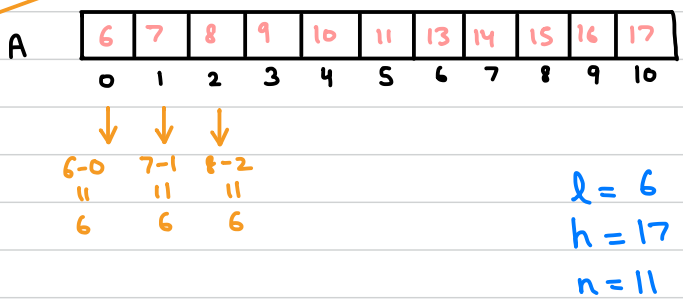
METHOD 1

ITERATIVE METHOD




```
for (i = 0 ; i < 11 ; i++)
        sum += A[i];
S =  n*(n+1)/2 ;      # Formula for sum of
                        n natural numbers
S - sum
78 - 71 = 7  → Missing num
```

METHOD 2



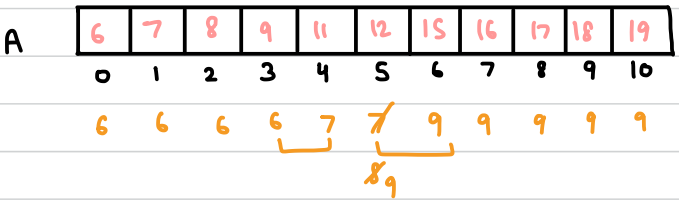
l = 6
h = 17
n = 11
To be known

```
diff = l-0;
for (i=0; i<n; i++)
{
    if (A[i]-i != diff)
    {
        printf(" missing element : %d", i+diff);
        break;
    }
}
```

O(n)

2. Multiple missing element in an sorted array.

METHOD 1




```
diff = 6-0;
for (i=0; i<n; i++)
{
    if (A[i]-i != diff)
    {
        while (diff < A[i]-i)
        {
            printf("%d", i+diff);
            diff++;
        }
    }
}
```

negligible (few elements)

O(n)

METHOD 2

A

6	7	8	9	11	12	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9	10

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

Hash Table / Bit Set

A table is created of size equal to the largest element of array A. The array is initialized with 0 One by one, each element present in array A, the index of hash array is initialized by 1 corresponding to it.

```
for (i=0; i<n; i++)                    ---------- n
    H[A[i]]++;

for (i=1; i<=h; i++)                   ---------- n
    if (H[i]==0)                       ----------
        printf("%d", i);               2n
                                       O(n)
```

FINDING DUPLICATE IN A SORTED ARRAY

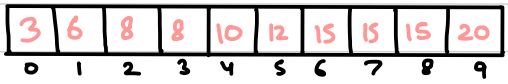
3	6	8	8	10	12	15	15	15	20
0	1	2	3	4	5	6	7	8	9

```
lastDuplicate = 0;

for (i=0; i<n; i++)
{
    if (A[i]==A[i+1] && A[i]!=lastDuplicate)
    {
        printf("%d\n", A[i]);
        lastDuplicate = A[i];
    }
}
```

So that if there are more than 1 duplicate, it will print only one time.

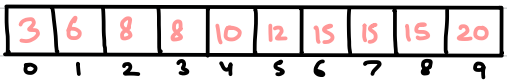
COUNTING NO OF TIMES OF DUPLICATE ELEMENT

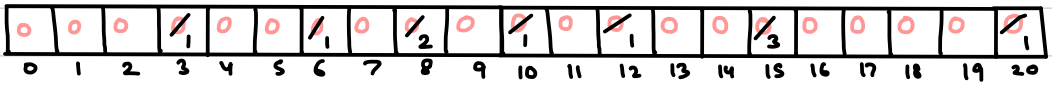
```
for (i=0; i<n-1; i++)
{
    if (A[i] == A[i+1])
    {
        j = i+1;

        while (A[j] == A[i])
            j++;

        printf("%d is appearing %d times", A[i], j-1);
        i = j-1;
    }
}
```



O(n)

FINDING DUPLICATES IN SORTED ARRAY USING HASHING





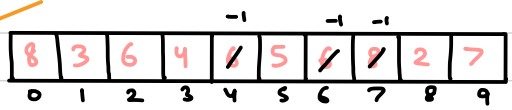

```
for (i=0; i<n; i++)
    H[A[i]]++;

for (i=0; i<=max; i++)
    if (H[i]>1)
        printf("%d %d", i, H[i]);
```

n+n = 2n

O(n)

FINDING DUPLICATES IN AN UNSORTED ARRAY

METHOD 1

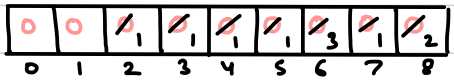



```
for (i=0; i<n-1; i++)
{
        count = 1;

        if( A[i] != -1)
        {
                for (j = i+1; i<n; j++)
                {
                        if (A[i] == A[j])
                        {
                                count ++;
                                A[j] = -1;
                        }
                }

                if (count > 1)
                        printf("%d %d ", A[i], count);
        }
}
```

$O(n^2)$

METHOD 2

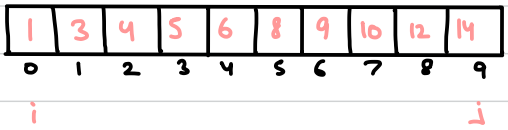
USING HASH TABLE



$$\frac{n + n}{O(n)}$$

FIND PAIR WITH SUM K (a+b=k)

SORTED



Let a+b = 10 (to be searched)

i+j

If i+j > 10
decrement j

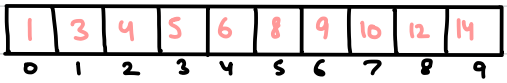
i+j < 10
increment i

i+j = 10
increment i
decrement j

```
i = 0, j = n-1
while (i<j)
{
    if (A[i] + A[j] == k)
    {
        printf("%d+ %d = %d", A[i], A[j], k);
        i++;
        j--;
    }
    else if (A[i] + A[j] < k)
        i++;
    else                                  O(n)
        j--;
}
```

FIND PAIR WITH SUM K (a+b=k)

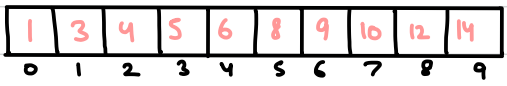
UNSORTED




```
for (i=0; i<n-1; i++)
{                                         O(n²)
    for (j= i+1; j<n; j++)
    {
        if (A[i] + A[j] == k)
            printf("%d+ %d = %d", A[i], A[j], k);
    }
}
```

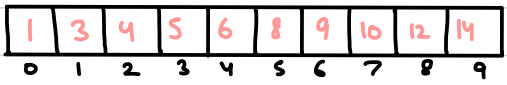
FIND PAIR WITH SUM K (a+b=k)

USING HASHING




```
for (i=0; i<n; i++)
{
        if (H[k-A[i]] != 0)
                printf ("%d + %d = %d", A[i], k-A[i], k);
        H[A[i]]++;
}
```

FIND MAX AND MIN IN SINGLE SCAN




```
min = A[0];
max = A[0];

for (i=1; i<n; i++)
{
        if (A[i]< min)
                min = A[i];
        else if (A[i] > max)
                max = A[i];
}
```

n=10

O(n)

Best = ← 10, 9, 8, 7, 2, 1

comp = n-1

worst = → 1, 2, 3, 5, 8, 9

comp = 2(n-1)