# Array

arr = [10, 15, 20, 30, 40]

→ Contagious Memory

| 10 | 15 | 20 | 30 | 40 |
|---|---|---|---|---|
| x | x+y | x+2y | x+3y | x+4y |

Advantages:

1) Random access
2) cache friendliness

Types of array (based on size)

→ Fixed size Arrays
→ Dynamic size Arrays

Fixed size Arrays

int arr[100] ] stack allocated →
int arr[n] ]
int *arr = new int[n] ] Heap allocated →
int arr[] = {10, 15, 30, 40} ] stack allocated →

Dynamic size Arrays

Resize automatically

c++ : Vector
Java : ArrayList
python : list

# Operations on Arrays

→ **Search (unsorted Array)**

I/p : arr[ ] = {20, 5, 7, 25}
n = 5

O/p : 1

I/p : arr[] : {20, 5, 7, 25}

O/p : -1

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

→ **Insert**

I/p : arr[] = {5, 10, 20, —, —}
x = 7
pos = 2

O/p : arr[] : {5, 7, 10, 20, —}

I/p arr[] = {5, 7, 10, 20, —}
x = 3
pos = 2

O/P arr[] = {5, 3, 7, 10, 20}

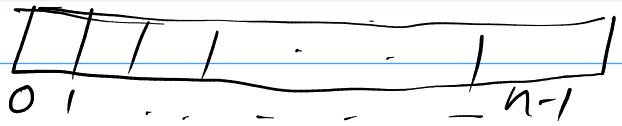
Algo

```
int insert(int arr[], int n,
           int x, int cap
           , int pos)
{ if (n == cap)
    return n;
  int idx = pos-1;
  for(int i=n-1; i>=idx; i--)
    arr[i+1]=arr[i];
  arr[idx] = x;
  return (n+1);
}
```

Time complexity : O(n)
Insert at the End : O(1)
Insert at the Beginning O(n)

# Insert at the End for Dynamic Sized Array

Intial capacity




Time complexity of every insert $= O(1)$
for first n inserts:

Average time complexity for n+1 $= \frac{O(1)+O(1)\cdots O(1)+O(1)}{n+1}$

$= O(1)$

## Delete:

i/p arr[] = {3, 8, 12, 5, 6}
x = 12

o/p arr[] = {3, 5, 5, 6, –}

```
int delete(int arr[],
           int size, int x)
{
  for(int i=0; i<size; i++)
  { if(arr[i]==x)
    { for(int j=i+1; j<size;
          j++)
        arr[j-1]=arr[j];
      size = size-1;
      break; }
  return size;
}
```

Note:

Insert : $O(n)$

Search : $O(n)$ for unsorted

$O(\log n)$ for sorted

Delete : $O(n)$

Get ith Element : $O(1)$

update ith Element : $O(1)$

Insert at the end and delete from the end can be done in $O(1)$ time

Arrays are cache friendly

why because of "contagecous memory"

prob\ Largest Element in an Array

I/p : arr[] = {10, 5, 2, 9, 8}

O/P : 2

I/p : arr[] = {40, 8, 50, 100}

O/p : 3

my solution:

```
int largest_element(int arr[], int size)
{
    int pos=0;
    for(int i=0; i<size; i++)
    {   if(arr[i] > arr[pos])
            pos = i;
    }
    return pos;
}
```

$O(n)$ : Time

$\Theta(1)$ : space

Naive approach

→ check each element with other elements

code:

```
int get largest (int arr[], int n)
{   for(int i=0; i<n; i++)
    {   bool flag = true;
        for(int j=0; j<n; j++)
        { if(arr[j] > arr[i])
          { flag = false;
            break;
          }
        }
    if(flag == true)
        return i;
    }
    return -1;
}
```

$O(n^2)$

prob 2 Second largest Element

I/p : arr[] = {10, 5, 8, 20}

O/p : 0 //Index of 10

I/p : arr[] = {20, 10, 20, 8, 12}

O/p : 4

I/p : arr[] = {10, 10, 10}

O/p : -1 //no second largest.

Solution: (Naive)

1) Find maximum first
2) again traverse find second

```
int second_largest(int arr[], int size)
{
    int pos = 0;
    for (int i=1; i<size; i++)
    {   if (arr[i] > arr[pos])
            pos = i;
    }
    int flpos = pos;
    pos = 0;  bool flag = false;
    for (int i=1; i<size; i++)
    {   if (arr[i] > arr[pos] && arr[flpos] != arr[i])
            pos = i; flag = true;
    }
    if (flag) return pos;
    else return 0;
}
```

<u>efficient Solution</u>

```
int slpos = -1, flpos = 0;
for (int i=0; i<size; ++i)
{
    if (arr[i] > arr[flpos])
    {   slpos = flpos;
        flpos = i;
    }
```

```
else if (arr[i] < arr[flpos])
{   if ( slpos == -1 || arr[i] > arr[slpos])
        slpos = i;
}
return slpos;
}
```

O(n) : one traversal
Θ(1) space complexity

Prob move zero to end.

I/p arr[] = {8, 5, 0, 10, 0, 20}.
O/p arr[] = {8, 5, 10, 20, 0, 0}

<u>naive solution</u>

① first find zero element
② then search for next non zero element and swap that zero and non zero element

Time complexity $O(n^2)$ -
Space complexity $\Theta(1)$ - constant time

Source code

```
for(int i=0; i<size; ++i)
{   int j=0;
```

```
if(arr[i]==0)
{
    for( j=i+1; j<size; ++j)
    {
        if(arr[j]!=0) break; }

    Swap(arr[i], arr[j]);
}
```

efficient solution:

```
void MoveZeroestoend(int arr[], int size)
{
    int NZcount=0;
    for(int i=0; i<n; ++i)
    {
        if (arr[i]!=0)
        {   swap(arr[i], arr[NZcount]);
            NZcount++;
        }
    }
}
```

Algorithm.

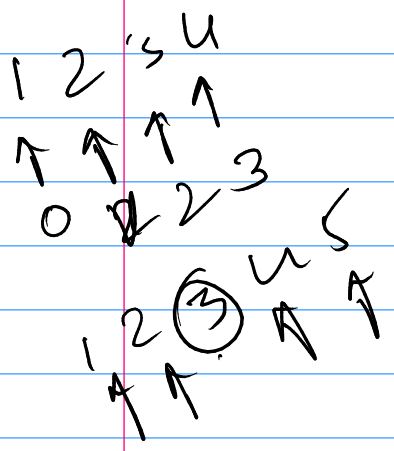
Here we need to keep track of nonzero elements i.e first zero. we need to swap current nonzero element with first zero.

Problem
Reverse an array

Algorithm:
1) Take 2 pointers one at begining and one at the end
2) Now swap and increment beggining point and decrement the end point.

code:

```
void reverse (int arr[], int size)
{
    int i=0, j=size-1;
    while (i<j)
    {
        swap(arr[i++], arr[j--]);
    }
}
```



Prob Left Rotate an Array by one.

I/p {1, 2, 3, 4, 5}

O/p {2, 3, 4, 5, 1}

Algo :

```
void LRAO(int arr[], int size)
{
    int temp = arr[0];
    for(int i=1; i<size; ++i)
    {
        swap(arr[i-1], arr[i]);
    }
    arr[size-1] = temp;
}
```