Dynamic Memory Allocation in C++ !! <stdlib.h>

Note: We can use the malloc(), calloc(), realloc() & free() functions in C++ as well by utilising the #include <cstdlib> header file.

But for C++, we mostly make use of these two keywords for DMA :→
(i) new → For Allocation
(ii) delete → For De-allocation

1D Array
n size

```
int * arr = new int[n]
```

This becomes an array of n elements.
ex: [1, 2, 3, 4, 5]

```
delete [ ] arr;
```

2D - Array
Square Matrix
(n×n)

```
int ** arr = new int *[n]
```

n → rows n → cols

The above line only defines the number of rows, so we need to allocate memory for 'n' cols as well.
Suppose row → i → 0 to n

```
arr[i] = new int[n]
```

cols

5×5
n n

```
delete[] arr;
```

2D - Array
Non-Square Matrix
(n×m)

```
int ** arr = new int *[n]
```

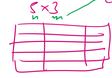
n → rows m → cols

The above line only defines the number of rows, so we need to allocate memory for 'm' cols as well.
Suppose row → i → 0 to n

```
arr[i] = new int[m]
```

cols

5×3
n m



```
delete[] arr;
```

Data Structures & Algorithms (DSA)
[Logic]

* Why?
  Data → Order → Ascending / Descending (Arrange) (Sort)
  ↳ Access → Less Time (Max / Min)
  ↳ Google Maps — Shortest Distance
  [Dijkstra's Algo graph]

Algorithms:

Searching Algorithms
* Linear Search
* Binary Search
* Recursive Binary Search
* Jump Search
* Interpolation Search

[Big O Notation]
Time Complexity

Sorting Algorithms
* Bubble Sort
* Selection Sort
* Insertion Sort
* Merge Sort
* Quick Sort
* Heap Sort (Heap)
* Count Sort
* Radix Sort
* Wave Sort
* Shell Sort
* Bucket Sort

Special Algorithms
* Kadane's Algorithm (array)
* Rabin Karp's Algorithm (string)
* Bellman Ford
* Topological Sort (graphs)

Searching Algos :→

(1) Linear Search :

int Key = 3

```
int arr[] = {9, 6, 1, 4, 3, 7, 0}
```

pos: 1 2 3 4 5 6 7
ind: 0 1 2 3 4 5 6

[unsorted array allowed]

return index or pos.

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

index position

invalid

K = 9
K O(1)
O(n)

Binary Search :→ Sorted Array (S & E)

key = 13

m = (S+E)/2 = 3

2	6	9	13	17	19
0	1	2	3	4	5
S		mid			E

→ sorted

mid value
m = (S+E)/2 = (0+5)/2 = 2

S = 2+1 = 3
e = 5
m = (3+5)/2 = 8/2 = 4

```
if (arr[mid] == key) {
    return mid;
}
else if (arr[mid] < key) {
    S = m+1;
}
else {
    e = m-1;
}
return -1;
```

13 == 13
9 == 13
17 == 13

9 < 13
17 < 13

(Move to right)
17 < 13

S = 3
e = 3

Time Complexity of Binary Search:

n

n/2 n/2

n/4 n/4

n/8 n/8

mid = (S+e)/2

K = log n

K = 0, 1, 2, 3, ... O(log n)

n → n/2^0
n/2 → n/2^1
n/4 → n/2^2
n/8 → n/2^3
⋮
n/2^K

Recursive Algo :→ Recursive Binary Search

Either the start value or the end value changes, therefore we can update these values & pass back to the same function (Recursion).

```
int recursiveBinarySearch(int * arr, int n,
                          int s, int e, int key)
{
}
```

size