Arrays

Qurra-tul-ann

Lecturer

Array

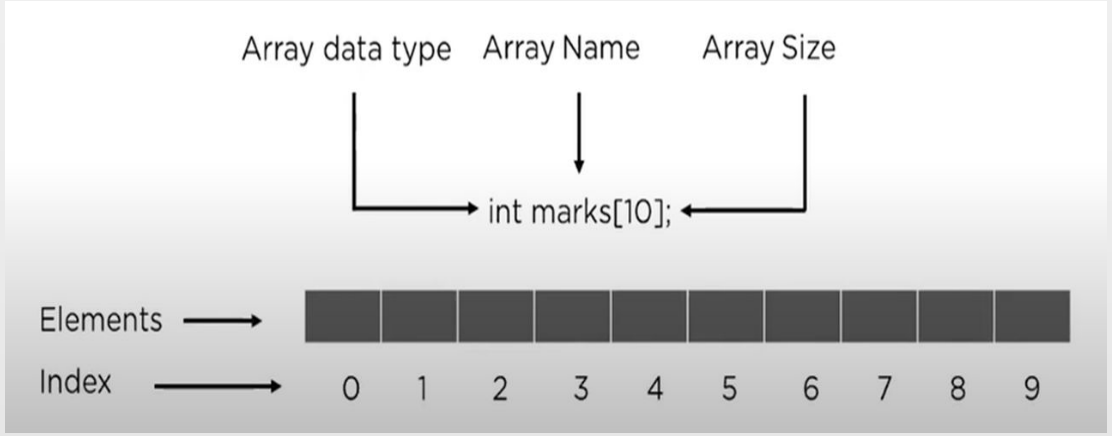
- Array is a container which can hold fix number of items and these items should be of same type. Most of the data structure make use of array to implement their algorithms.
- Following are important terms to understand the concepts of Array.

  Element − Each item stored in an array is called an element.

  Index − Each location of an element in an array has a numerical index which is used to identify the element.

- Consider following program for Array declaration and its representation in memory

```
main( )
{
    int x[10]={35,33,42,10,14,19,27,44,26,31};
    int j;
    for(j = 0; j < 6; j++)
        x[j] = 2 * j;
}
```



Array Declaration/Initialization

- **Declaration :**

```
int a[20], b[3],c[7];
float f[5], c[2];
char m[4], n[20];
```

- **Initialization :**

```
float, b[3]={2.0, 5.5, 3.14};
char name[4]= {‘E’,’m’,’r’,’e’};
int c[10]={0};
int a[]={1,2,3,4,5};
Int a[3];  a[0]=1; a[1]=1; a[2]=2;
```

ACCESSING ELEMENT OF THE ARRAY

- TO access array element :

  a[5];

- Operations on array:
  - traversal
  - insertion
  - Deletion
  - searching
  - sorting

TRAVERSAL

```
#include <iostream>
using namespace std;
#define size 10     // another way int const size = 10
int main(){
   int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0,LB=0, UB=size;
    float av;
   for(i=LB;i<UB;i++)
      {     sum = sum + x[i];
            av = (float)sum/size;
            cout<< "The average of the numbers= "<<av<<endl;
     }
return 0;
}
```

INSERT

```
void insertElement(int arr[], int &size, int pos, int element) {
    if (pos > size || pos < 0) {
        cout << "Invalid Position!" << endl;
        return;
    }
for (int i = size; i > pos; i--) {    // Shift elements to the right
        arr[i] = arr[i - 1];
    }
    arr[pos] = element;          // Insert the new element
    size++;  // Increase size
for (int i = 0; i < size; i++) {    // Print updated array
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

```
int main() {
    int arr[10] = {1, 2, 3, 5, 6};    // Initial array
    int size = 5;            // Current size
    int element = 4, position = 3;      // Insert 4 at index 3
    insertElement(arr, size, position, element);
 return 0;
}
```

Update

```
void update(int arr[], int size, int index, int newValue) {
    if (index < 0 || index >= size) {
        cout << "Invalid index!" << endl;
        return;
    }
    arr[index] = newValue;
}
```

Deletion

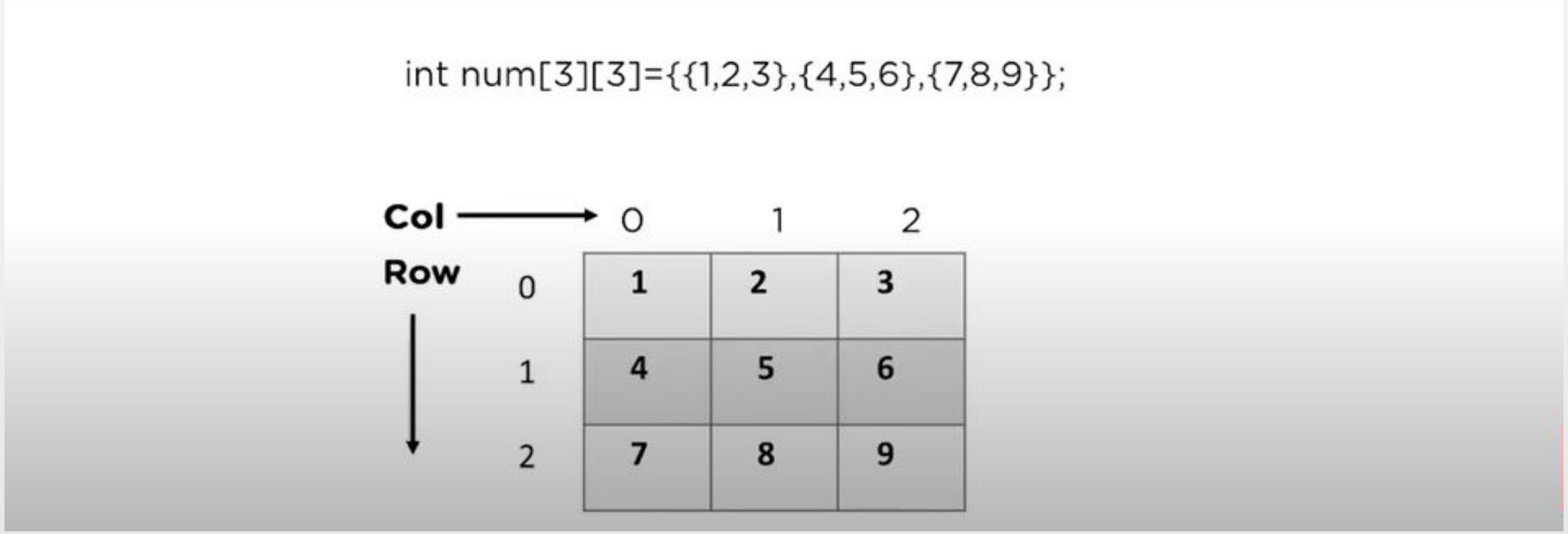
```
void deleteElement(int arr[], int &size, int pos) {
    if (pos >= size || pos < 0) {
        cout << "Invalid Position!" << endl;
        return;
    }
for (int i = pos; i < size - 1; i++) {// Shift elements to the left
        arr[i] = arr[i + 1];
    }
    size--; // Reduce size
for (int i = 0; i < size; i++) {      // Print updated array
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

```
int main() {
    int arr[10] = {1, 2, 3, 4, 5};
    int size = 5;    // Current size

    int position = 2; // Delete element at index 2 (value = 3)
    deleteElement(arr, size, position);
    return 0;
}
```

2-D Array

- The two dimensional (2D) array is also known as matrix.
- A matrix can be represented as a table of rows and columns.



Display a matrix

```
#include <iostream>
using namespace std;
int main() {
int x[3][3]={{3,4,5},
             {6,7,8},
             {9,1,2}};
     for(int i = 0; i< 3; i++)
     {
       for(int j = 0; j< 3; j++)
             {
                 cout<<x[i][j];
             }
             cout<<"\n";
     }
}
```

Sparse Matrix

It is a matrix in which most of the elements are **zero**. To efficiently store and manipulate a sparse matrix, we use a **compressed storage format** such as a 3-column representation (row, column, value).

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

2D array is used to represent a sparse matrix in which there are three rows named as

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row, column)

Sparse Matrix $\rightarrow$

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 3 & 6 \\ 2 & 0 & 0 & 2 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

EXAMPLE:

```
#include<iostream>
using namespace std;
int main () {
   int a[10][10] = { {0, 0, 9} , {5, 0, 8} , {7, 0, 0} };
   int i, j, count = 0;
   int row = 3, col = 3;
   for (i = 0; i < row; ++i) {
      for (j = 0; j < col; ++j){
         if (a[i][j] == 0)
         count++;
      }
   }
cout<<"The matrix is:"<<endl;
   for (i = 0; i < row; ++i) {
      for (j = 0; j < col; ++j) {
         cout<<a[i][j]<<" ";
      }
      cout<<endl;
   }
   cout<<"The number of zeros in the matrix are "<<
count <<endl;
   if (count > ((row * col)/ 2))
   cout<<"This is a sparse matrix"<<endl;
   else
   cout<<"This is not a sparse matrix"<<endl;
   return 0;
}
```

Issues with Array

- Want to use an array data structure but may lack the information about the size of the array at compile time
- For example you have to store telephone directory or store the names of total population of a city or a country
- If you initially assign a very large chunk of memory and store very few information what will happen?
- If you initially store very small chunk of memory and your requirement increases with the passage of time – what will happen?
- Misuse of resource
- What are the possible solutions?

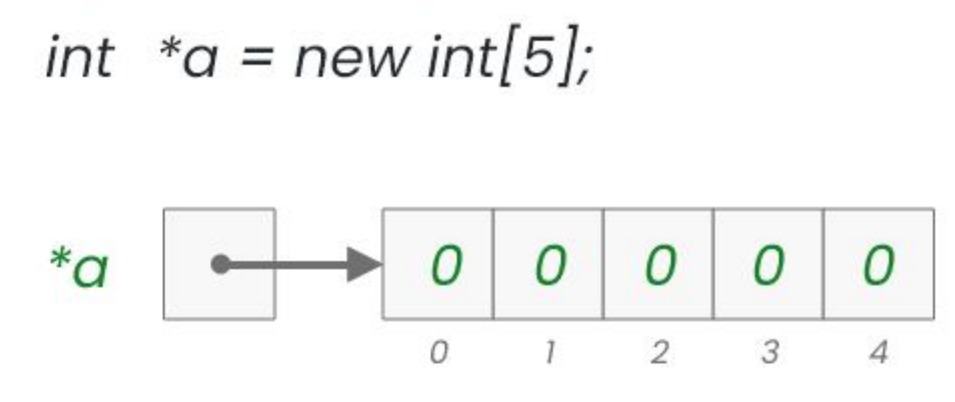
Dynamic Array

- Dynamic array allocation is actually a combination of pointers and dynamic memory allocation.
- Dynamic arrays are created in the heap using the ***new*** and released from the heap using ***delete*** operators.

int* y = new int[20];

- Computer will allocate twenty memory location at the time of execution.
- The **new** key word returns the memory address of first of twenty locations and store that address into y.
- Use following statement to release memory because y is allocated memory using new

delete[] y;



EXAMPLE:

```
#include<iostream>
using namespace std;
int main() {
    int x, n;
    cout << "Enter the number of items:" << "\n";
    cin >>n;
    int *arr = new int[n];
    cout << "Enter " << n << " items" << endl;
    for (x = 0; x < n; x++) {
        cin >> arr[x];
    }
cout << "You entered: ";
    for (x = 0; x < n; x++) {
        cout << arr[x] << " ";
    }
 delete [] arr;; // freeing-up space
    return 0;
}
```

DYNAMIC ALLOCATION USING VECTORS

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;
vector<int> arr(size);    // Create dynamic array using vector

cout << "Enter " << size << " elements: ";     // Taking input
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }
cout << "Array elements: "; // Display array
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

ASYMPTOTIC ANALYSIS

- Used to describe the growth of functions in terms of input size (n).
- Asymptotic notation is a shorthand way to represent the time complexity.
- Compares different algorithms effectively.

- **Types of Asymptotic Notation**
  - **Big-O (O)** – Upper bound (worst-case).
  - **Omega (Ω)** – Lower bound (best-case).
  - **Theta (Θ)** – Tight bound (average-case).

Big-O Notation (O): Worst –case

- **Definition:**

  $O(g(n))$ represents an *upper bound* on the growth rate of a function $f(n)$. There exist positive constants c and n_0 such that $\mathbf{f(n) \le c * g(n)}$ for all $n \ge n_0$.

- **Explanation:** function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

- **Example:**

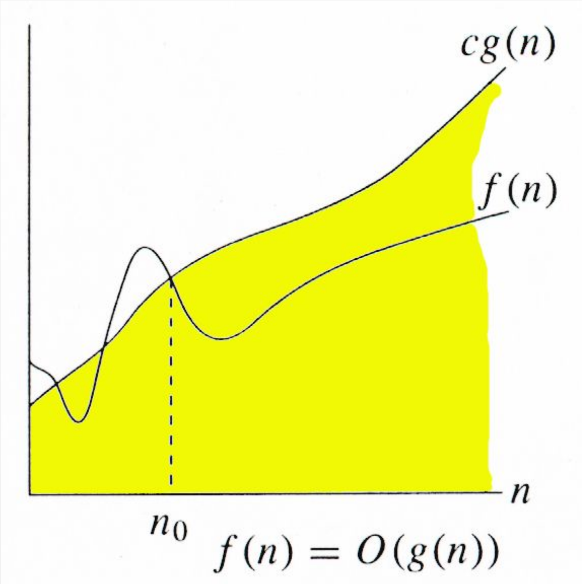
  $f(n) = 2n^2 + 5n + 1$

  $g(n) = n^2$

  We can find $c=3$ and $n_0=1$

  such that $2n^2 + 5n + 1 \le 3n^2$ for all $n \ge 1$.

  Therefore, $f(n)$ is $\mathbf{O(n^2)}$.



Big-O Notation (O)

- Example:

```
void example(int n) {
    for (int i = 0; i < n; i++) {
        cout << i << endl;
    }
}
```

- Complexity: **O(n)** (Linear Time)

Omega Notation (Ω): Best-case

- **Definition:**

  $\Omega(g(n))$ represents a *lower bound* on the growth rate of a function f(n). There exist positive constants c and n_0 such that $\mathbf{c * g(n) \le f(n)}$ for all $n \ge n_0$.

- **Explanation:** It means function ***g*** is a lower bound for function ***f*** ; after a certain value of ***n, f*** will never go below ***g***..

- **Example:**

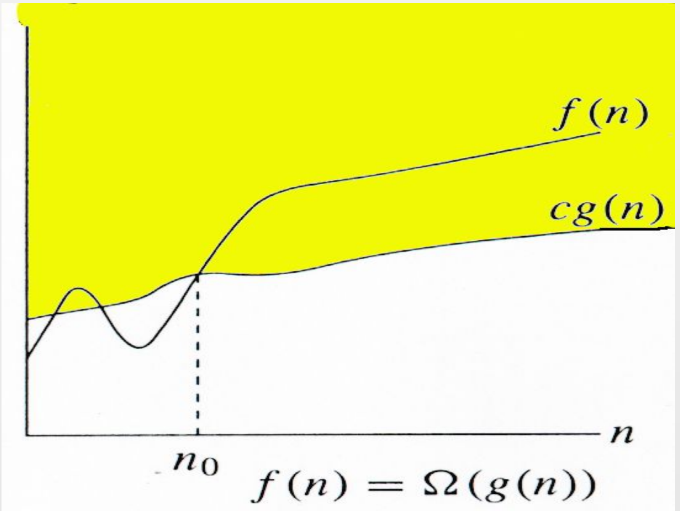
  - $f(n) = n^3 - 2n$

    $g(n) = n^3$

    $c=1/2$ and $n_0=2$

    $(1/2)n^3 \le n^3 - 2n$ for all $n \ge 2$.

    Therefore, f(n) is $\mathbf{\Omega(n^3)}$.



- Example:

```
int findMin(int arr[], int n) {
    return arr[0]; // Best case when the minimum is the firs
}
```

- Complexity: **Ω(1)** (Constant Time)

Theta Notation (Θ): Average case

- **Definition:**

  $\Theta(g(n))$ represents both an *upper and lower bound* i.e (tight bound)on the growth rate of a function f(n). There exist positive constants c_1, c_2, and n_0 such that $\mathbf{c_1 * g(n) \leq f(n) \leq c_2 * g(n)}$ for all $n \geq n_0$.

- **Explanation:** f(n) grows *at the same rate* as g(n) (up to constant

  factors) for sufficiently large inputs.

- **Example:**

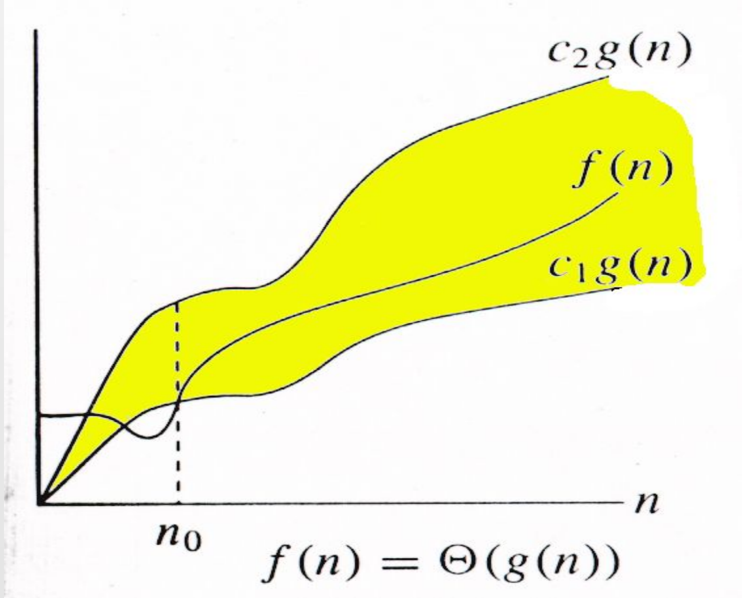
- $f(n) = 3n^2 + 7n - 2$

  $g(n) = n^2$

  We can find $c_1=3/2$, $c_2=5$ and $n_0=7$

  $(3/2)n^2 \leq 3n^2 + 7n - 2 \leq 5n^2$ for all $n \geq 7$.

  Therefore, f(n) is $\mathbf{\Theta(n^2)}$.



Theta Notation (Θ)

```
void example(int n) {
    for (int i = 0; i < n; i++) {
        cout << i << endl;
    }
}
```

- Complexity: **Θ(n)** (Linear Time)

Common Complexity Classes

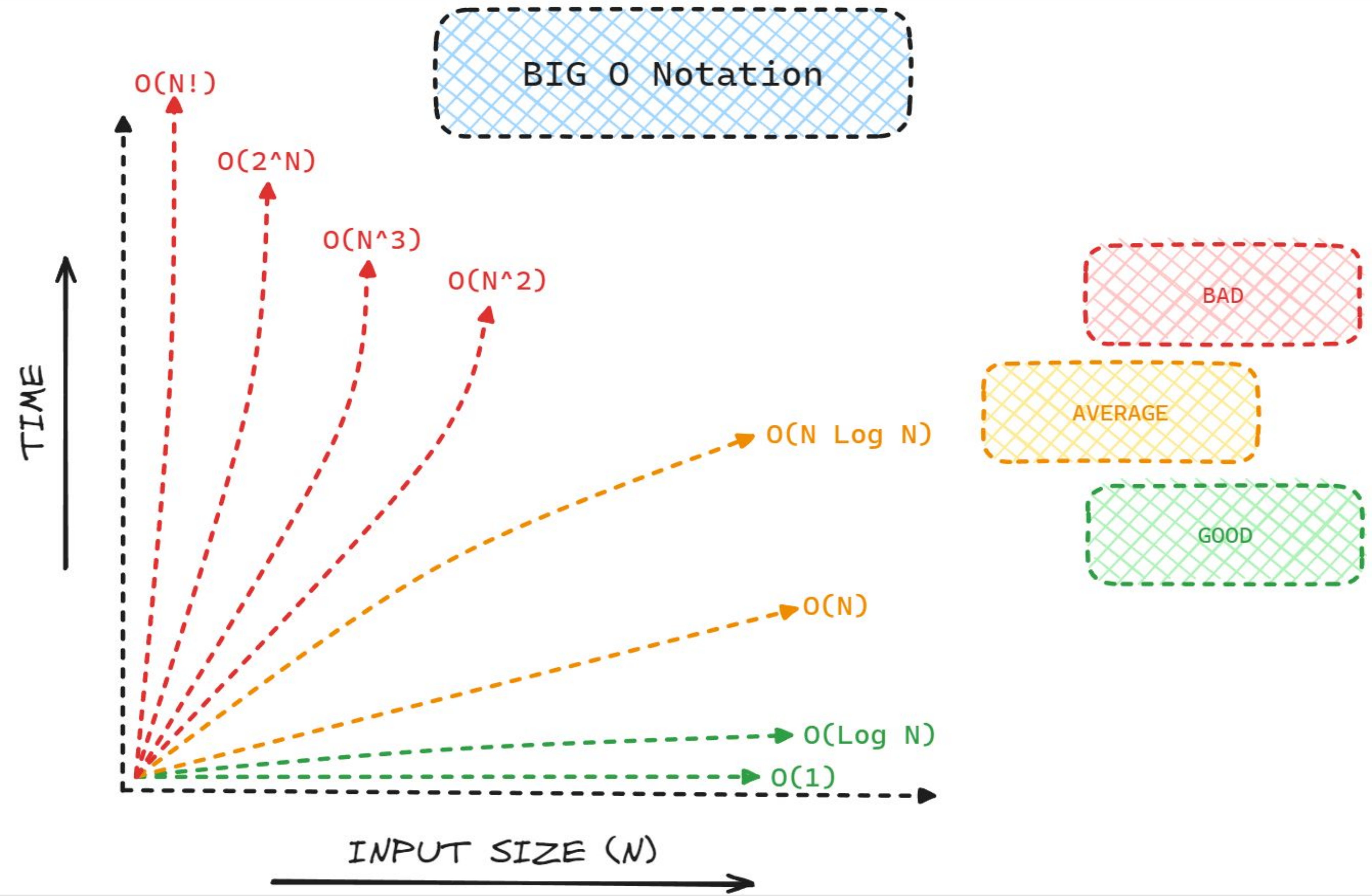
Growth Rate	Name	Code Example	description
1	Constant	`a= b + 1;`	statement (one line of code)

log(n)	Logarithmic	`while(n>1){`
		`n=n/2;`
		`}`
			Divide in half (binary search)

n	Linear	`for(c=0; c<n; c++){`
		`a+=1;`
		`}`
			Loop

n*log(n)	Linearithmic	Mergesort, Quicksort, …	Effective sorting algorithms

n^2	Quadratic	`for(c=0; c<n; c++){`
		`for(i=0; i<n; i++){`
		`a+=1;`
		`}`
			Double loop


BIG O Notation
O(N!)
O(2^N)
O(N^3)
O(N^2)
O(N Log N)
O(N)
O(Log N)
O(1)
TIME
INPUT SIZE (N)
BAD
AVERAGE
GOOD

Example 1

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World";
    return 0;
}
```

time complexity is **constant: O(1)** i.e. every time a constant amount of time is required to execute code

Example 2: O(n) - Linear time

Linear time complexity *O(n)* means that the algorithms take proportionally longer to complete as the input grows.

```
#include <iostream>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 1; i <= n; i++) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

- "Hello World !!!" is printed only **n times** on the screen, as the value of n can change. So it is **O(n)**
- Eg:
    1. Get the max/min value in an array.
    2. Find a given element in a collection.
    3. Print all the values in a list.

Example 3: O(n^2) - Quadratic time

- A function with a quadratic time complexity has a growth rate of n^2. If the input is size 2, it will do four operations. If the input is size 8, it will take 64, and so on

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {           // Outer loop (n)
        for (int j = 0; j < n - i - 1; j++) {  // Inner loop (n)
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);  // Swap if elements are in wrong order
            }
        }
    }
}
```

Example 4

```
#include <iostream>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

Time Complexity: $O(\log_2(n))$

Example 5

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 2; i <= n; i=pow(i,2)) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

Time Complexity: O(log(log n))

Finding the complexities

```
int list_Sum(int A[], int n)
{
    int sum = 0;     // cost=1  no of times=1
    for(int i=0; i<n; i++)     // cost=2  no of times=n+1 (+1 for the end
false condition)
        sum = sum + A[i]  ;     // cost=2  no of times=n
    return sum ;                // cost=1  no of times=1
}
```

Therefore the total cost to perform sum operation

Tsum=1 + 2 * (n+1) + 2 * n + 1 = 4n + 4 =C1 * n + C2 = O(n)

Finding the complexities

1	Algorithm Message(n)	0
2	{	0
3	for i=1 to n do	n+1
4	{	0
5	write("Hello");	n
6	}	0
7	}	0
	total frequency count	2n+1

While computing the time complexity we will neglect all the constants, hence ignoring 2 and 1 we will get n. Hence the time complexity becomes **O(n).**

Finding the complexities

1	Algorithm add(A,B,m,n)	0
2	{	0
3	for i=1 to m do	m+1
4	for j=1 to n do	m(n+1)
5	C[i,j] = A[i,j]+B[i,j]	mn
6	}	0
	total frequency count	2mn+2m+1

f(n) = Og(n). => O(2mn+2m+1)// when m=n; = O(2n2 +2n+1); By neglecting the constants, we get the time complexity as O(n2). The maximum degree of the polynomial has to be considered

Computational Complexity of Linear Search

It is the maximum number of comparisons you need to search the array. As you are visiting all the array elements in the worst case, then, the number of comparisons required is:

n (n is the size of the array)

Example:

If a given an array of 1024 elements, then the maximum number of comparisons required is:

n-1 = 1023 (As many as 1023 comparisons may be required)

Computational Complexity of Binary Search

- The searched array is divided by 2 for each comparison/iteration.

  Therefore, the maximum number of comparisons is measured by:

  ***log*2(*n*)** where n is the size of the array

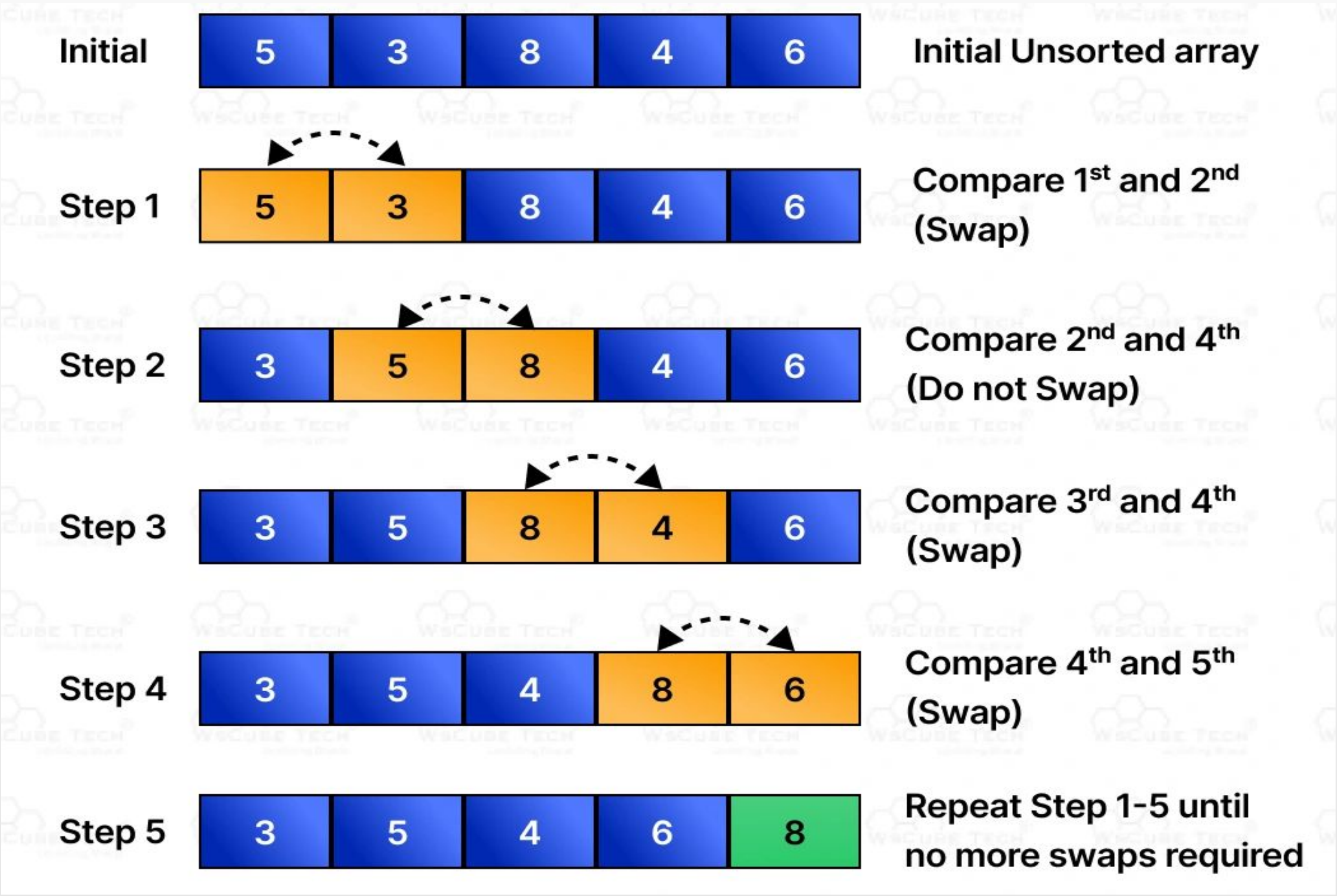
Example:

If a given sorted array 1024 elements, then the maximum number of comparisons required is:

$log2(1024) = 10$ (only 10 comparisons are enough)

Bubble Sort

- **Idea:**
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order
  - Easier to implement, but slower than Insertion sort



Algo of bubble sort

```
BubbleSort(A)
n = Length[A];
for j = 0 to n-2
    for i = 0 to n-j-2
        if A[i] > A[i+1]
            temp = A[i]
          A[i] = A[i+1]
          A[i+1] = temp
return A;
```

BUBBLE SORT

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {        // (1) O(n) - Outer loop runs (n-1) times
        for (int j = 0; j < n - i - 1; j++) { // (2) O(n) - Inner loop runs (n-i-1) times
            if (arr[j] > arr[j + 1]) {       // (3) O(1) - Constant time comparison
                swap(arr[j], arr[j + 1]);    // (4) O(1) - Constant time swap operation
            }
        }
    }
}
```

Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A] c_1

do for j ← length[A] **downto** i + 1 c_2

Comparisons: $\approx n^2/2$

do if A[j] < A[j -1] c_3

Exchanges: $\approx n^2/2$

then exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

$$where \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

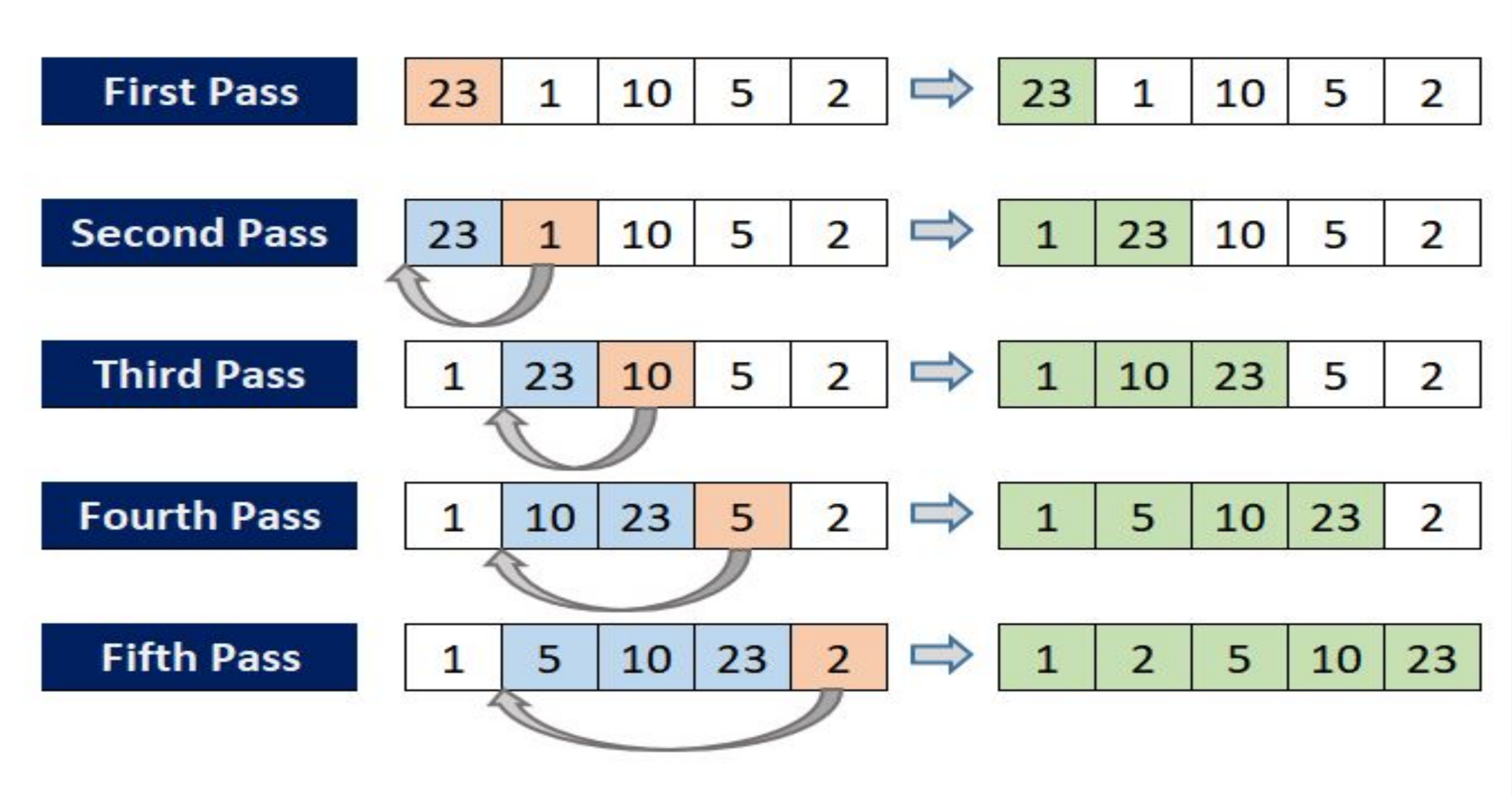
Thus, $T(n) = \Theta(n^2)$

Insertion Sort

Insertion Sort

- **Idea:** like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort



Algo

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
do *key* $\leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted		
$\triangleright$ sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

INSERTION SORT

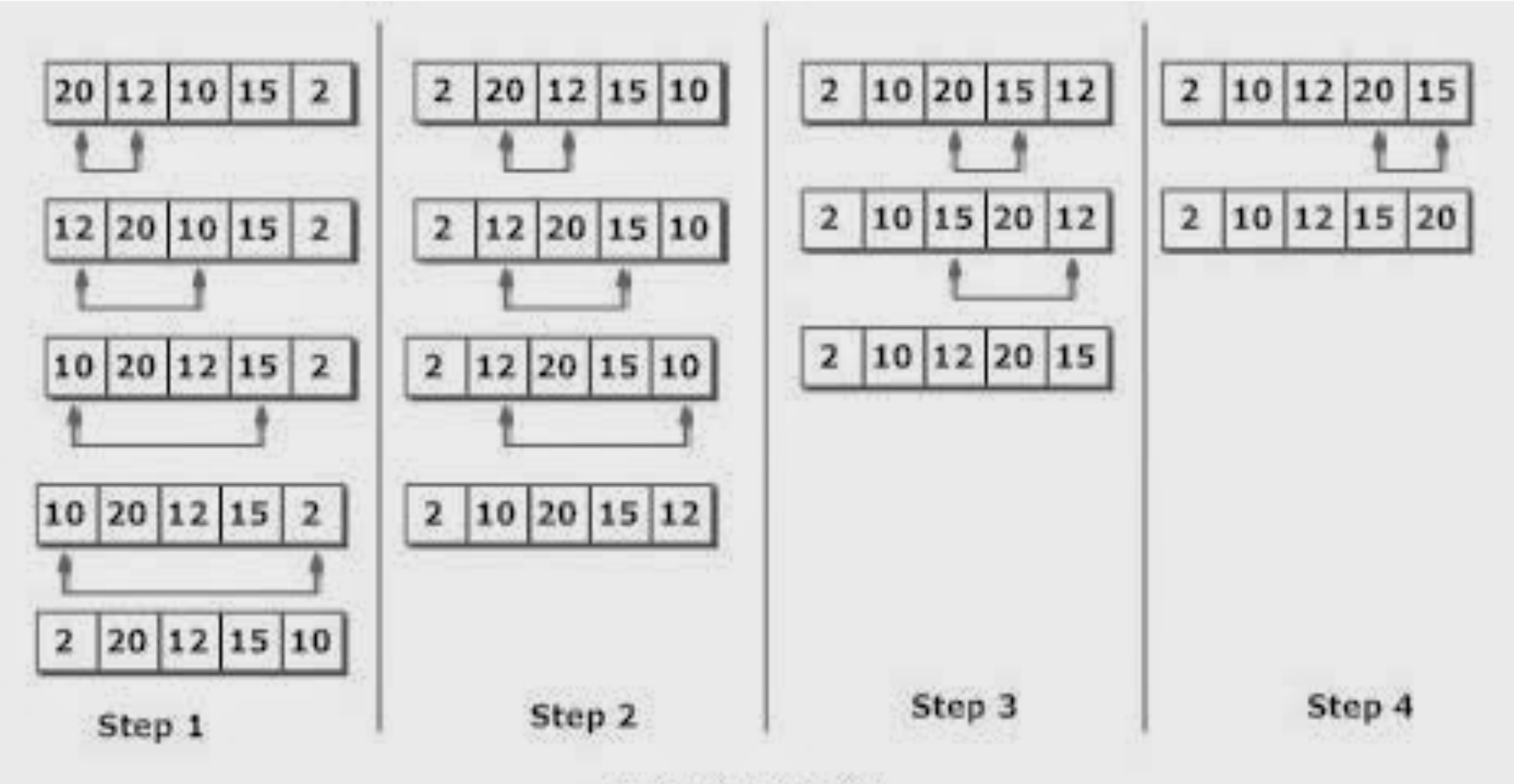
```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {          // (1) O(n) - Outer loop runs (n-1) times
        int key = arr[i];                  // (2) O(1) - Constant time assignment
        int j = i - 1;                     // (3) O(1) - Constant time assignment

        while (j >= 0 && arr[j] > key) {   // (4) O(j) - Worst case: runs j times
            arr[j + 1] = arr[j];           // (5) O(1) - Constant time shift
            j--;                           // (6) O(1) - Constant time decrement
        }
        arr[j + 1] = key;                  // (7) O(1) - Constant time assignment
    }
}
```

Selection Sort

- **Idea:**
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- **Disadvantage:**
  - Running time depends only slightly on the amount of order in the file

Selection Sort



Analysis of Selection Sort

	cost	times
Alg.: SELECTION-SORT*(A)*	c_1	1
n ← length[A]	c_2	n
for j ← 1 **to** n - 1	c_3	n-1
do smallest ← j	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
for i ← j + 1 **to** n	c_5	$\sum_{j=1}^{n-1}(n-j)$
do if A[i] < A[smallest]	c_6	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i		
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Selection Sort

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {     // (1) O(n) - Outer loop runs (n-1) times
        int minIndex = i;                 // (2) O(1) - Constant time assignment

        for (int j = i + 1; j < n; j++) { // (3) O(n) - Inner loop runs (n-i-1) times
            if (arr[j] < arr[minIndex]) { // (4) O(1) - Constant time comparison
                minIndex = j;             // (5) O(1) - Constant time assignment
            }
        }
```