数据结构
Data Structures

Chapter 4 Basics Of Algorithm

Prof. Yitian Shao
School of Computer Science and Technology

Basics Of Algorithm

Course Overview

- Algorithms
  - Definition
  - Description methods
  - Building blocks of algorithms

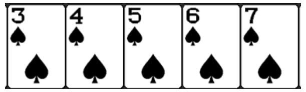
- Evaluation of algorithms
  - Big O
  - Analyze time complexity

- Basic sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort

Previously in High-level Language Programming

- An **algorithm** describes **how to solve a problem**; it is a **procedure** that takes in **input**, follows a certain set of steps, and then produces an **output**

Example problem: Given a set of five cards (randomly shuffled), pick the **largest one**



Input → **Procedure** → **Output**

Input: A set of 5 cards

Output: The card with the largest value

Procedure: **Up to the designer of the algorithm**

Previously in High-level Language Programming

The ways to describe an algorithm

- **Language description**: Might be in the form of text that describes the algorithm; generally does not involve implementation details of the algorithm.

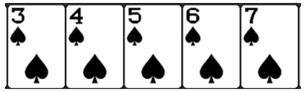
- **Pseudocode**: Loosely formalizing an algorithm, with **general implementation details**; (programming) language-specific details are left out so as not to complicate things.

- **Flowchart**: Visual representation that depicts the step-by-step procedure of an algorithm; can help **simplify complex algorithms** into **visually understandable** forms.

- **Implementation**: An implementation in a given programming language will be a piece of code that is understandable and runnable by a computer; it will fulfill the goals and procedure of the algorithm.

Previously in High-level Language Programming

- **Pseudocode**: Loosely formalizing an algorithm, with general implementation details; language-specific details are left out so as not to complicate things.



Example of pseudo-code:

Input: Given a set of card values $\{c_i\}$, with card index $i = 0,1,2,3,4$

Output: the largest value res

Procedure:

Initialize $\mathrm{res} = c_0$

For each card index i form 1 to 4:

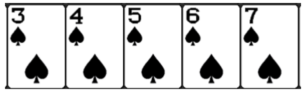
    if $\mathrm{res} < c_i$, then let $\mathrm{res} = c_i$

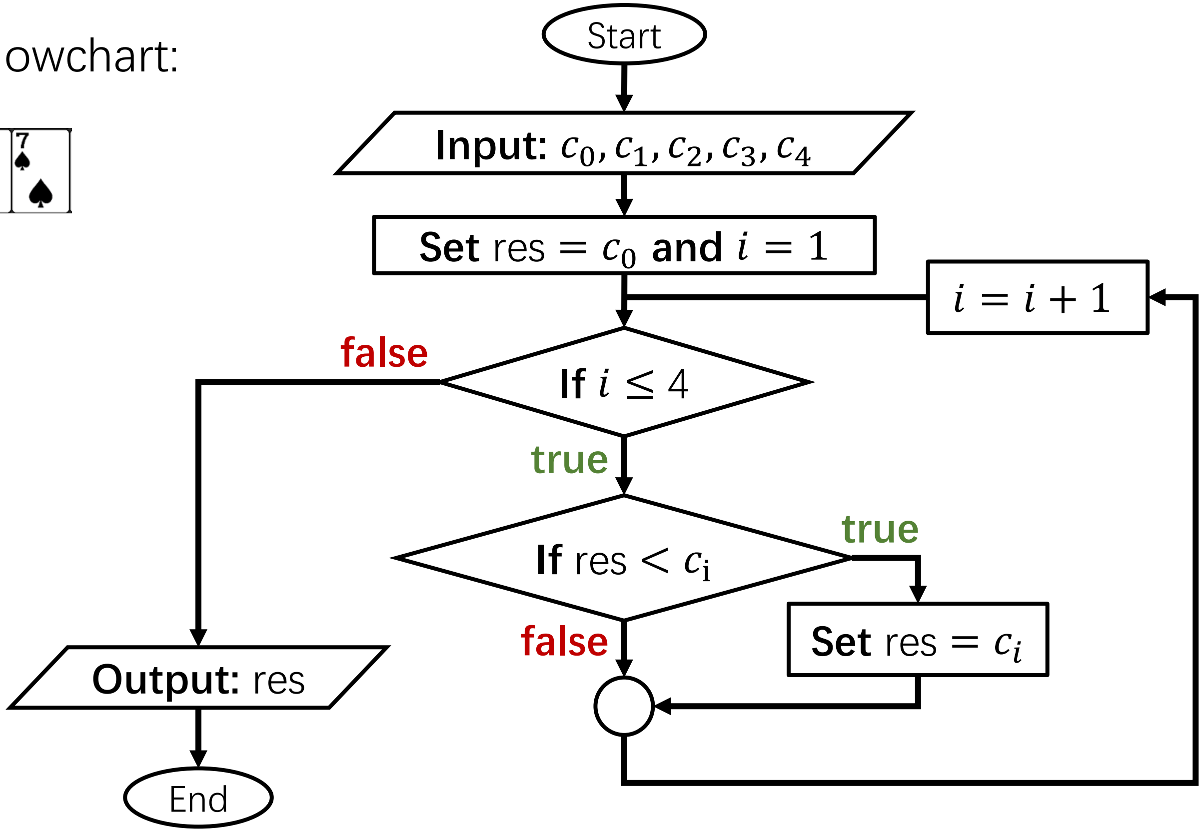
End

Previously in High-level Language Programming

- **Flowchart**: Visual representation that depicts the step-by-step procedure of an algorithm; can help simplify complex algorithms into visually understandable forms.

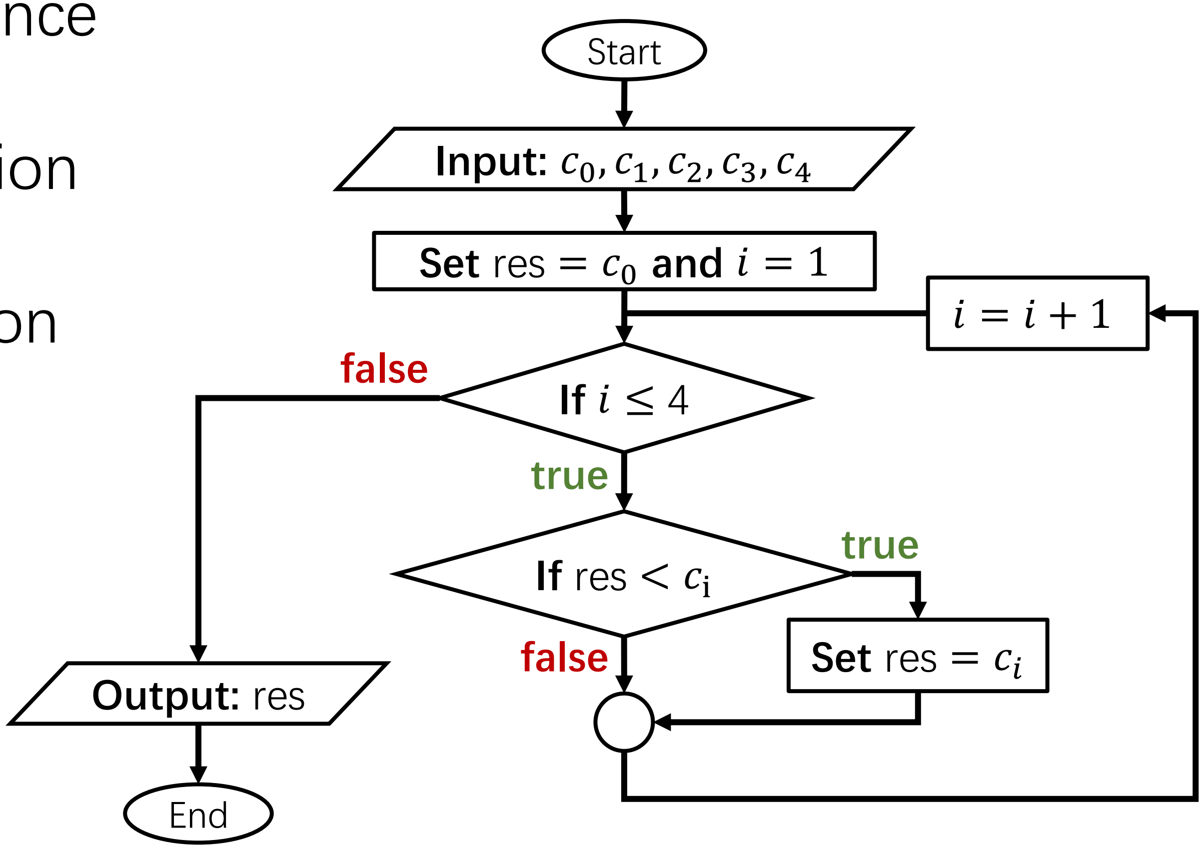
Example of flowchart:





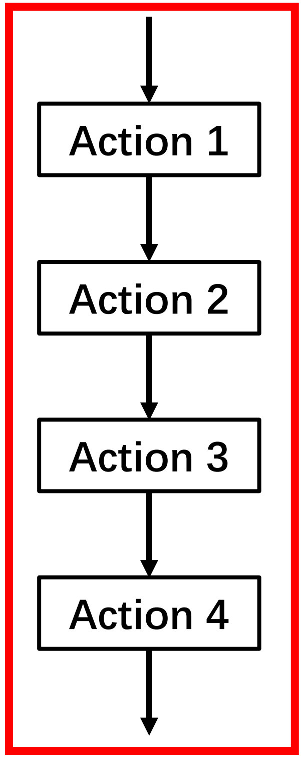
Recall the building blocks of algorithms

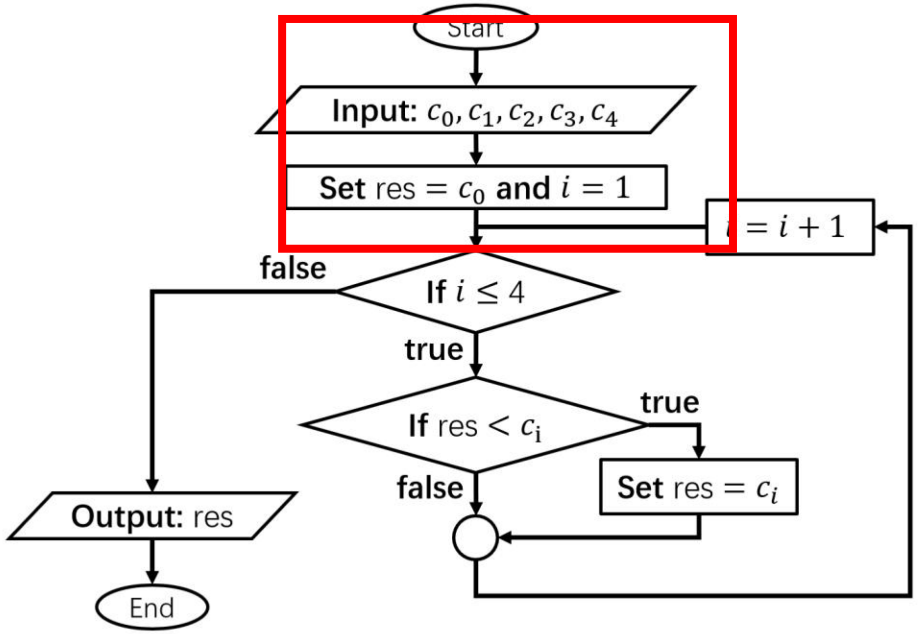
- An **algorithm** is made up of three basic **building blocks**:
  - Sequence
  - Selection
  - Iteration



Sequence

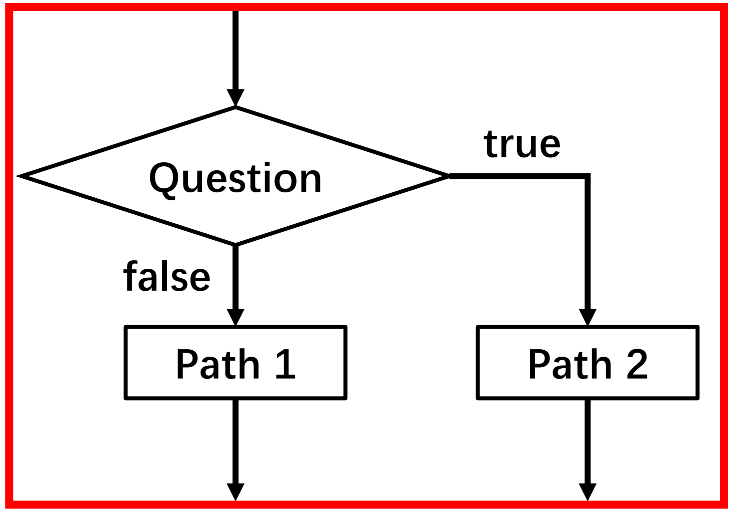
- A **sequence** is a series of actions (code statements) that is completed in a specific order, until all of the actions in the sequence have been carried out

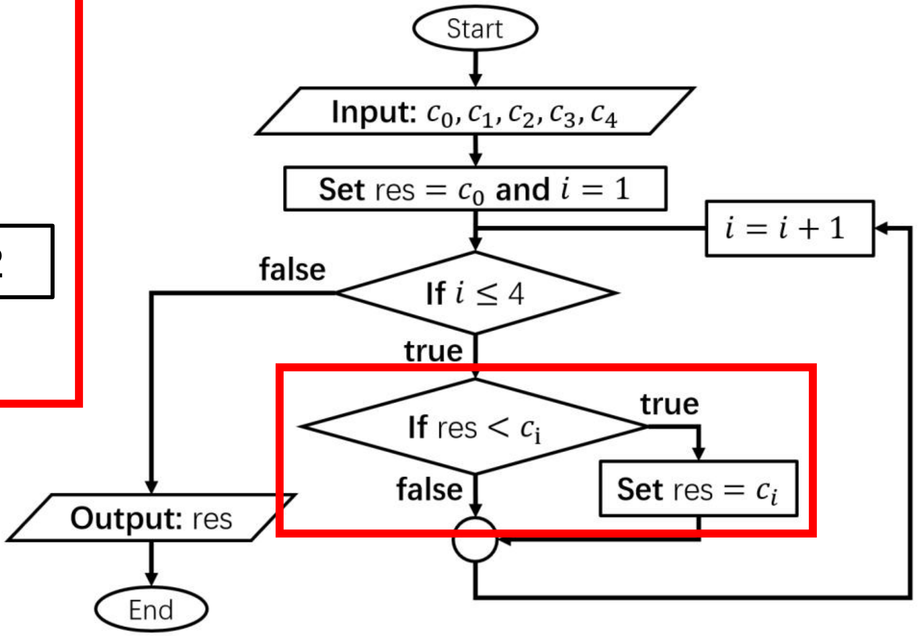




Selection

- Instead of following a specific order of actions, **selection** ask a question in order to figure out which path to take next. (if-else / switch statements…)





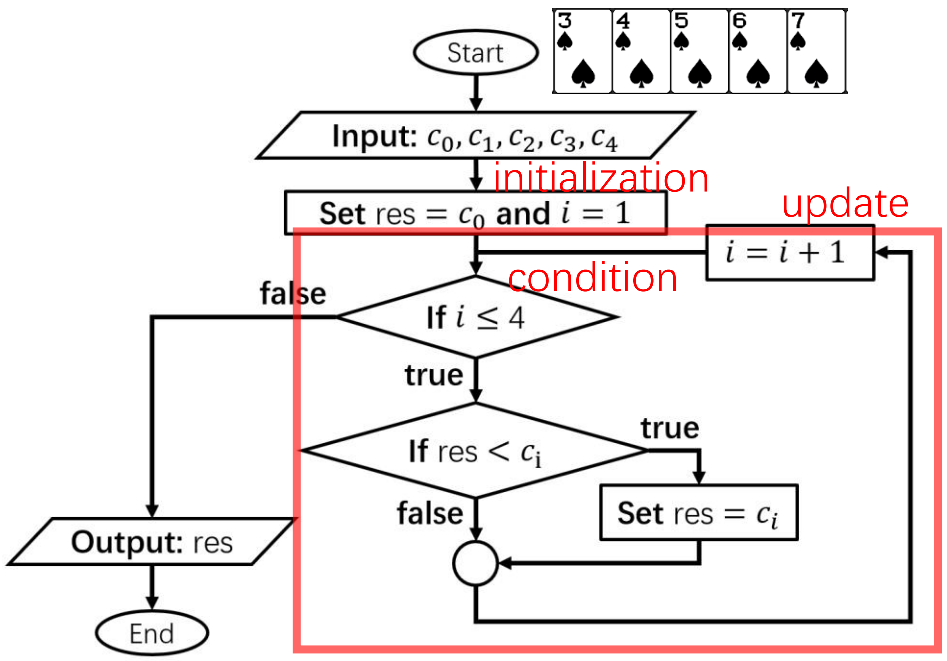
Relational operation

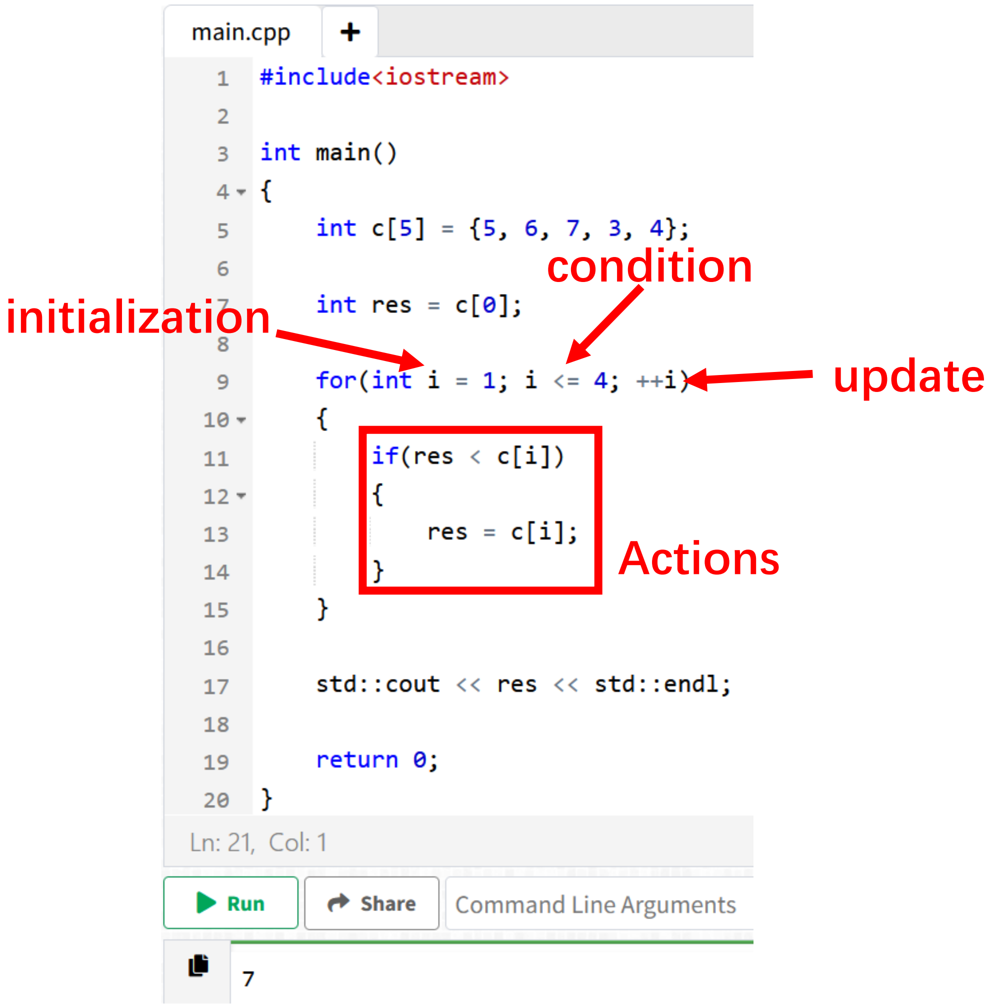
```
if(a > b)
{
    // Path 1
}
else
{
    // Path 2
}
```

```
switch(month)
{
    case 1:
    std::cout << "Jan";
    break;
    case 2:
    std::cout << "Feb";
    break;
    case 3:
    std::cout << "Mar";
    break;
    default:
    std::cout << month;
    break;
}
```

Iteration

- **Loops** are used to repeat a block of code
  - **for** loop





How to evaluate an algorithm?

- What makes an algorithm "good" ?
- How to compare algorithms

Big O analysis!

Big O Intuition

- Lots of different ways to solve a problem
- Measure algorithmic efficiency: resources used (time, memory, etc.)
  - We focus on **time efficiency** in this class
- Idea: algorithms are better if they take less time
- Problem: amount of time a program takes is variable – Depends on what computer you're using, what other programs are running, if your laptop is plugged in, etc..

Big O

- Idea: assume each statement of code takes some **unit of time** – for the purposes of this class, that **unit doesn't matter**
- We can **count the number of units of time** and get the **runtime**
- Sometimes, the number of statements depends on the input – we'll say the input size is N

Big O

```
statement1;          // runtime = 1

for (int i = 1; i <= N; i++) {          // runtime = 3N
        statement3;
        statement4;
        statement5;
}

for (int i = 1; i <= N; i++) {           // runtime = N²
    for (int j = 1; j <= N; j++) {      // runtime = N
        statement2;
    }
}
// total = N² + 3N + 1
```

Big O

- The **actual constant doesn't matter** – so we get rid of the constants: $N^2 + 3N + 1 \text{ -> } N^2 + N + 1$
- Only the **biggest power of N matters**: $N^2 + N + 1 \text{-> } N^2$
  - The biggest term grows so much faster than the other terms that the runtime of that term "dominates"
- We would then say the code snippet has $O(N^2)$ **runtime**

Finding Big O

- Work from the **innermost** indented code out
- Realize that some code statements are more costly than others
  - It takes $O(N^2)$ time to **call a function with runtime $O(N^2)$**, even though calling that function is only one line of code
- Nested code multiplies
- Code at the same indentation level adds

Exercise: What is the Big O?

```
int sum = 0;                              // runtime = 1
for (int i = 1; i < 100000; i++) {        // runtime = 100000 * N
    for (int k = 1; k <= N; k++) {        // runtime = N
        sum++;
    }
}
Vector<int> v;        // runtime = 1
    for (int x = 1; x <= N; x += 2) {     // runtime = N
        v.insert(0, x);
    }
cout << v << endl;    // runtime = 1
```

total = 100000N + 2N + 3
= 100002N + 3

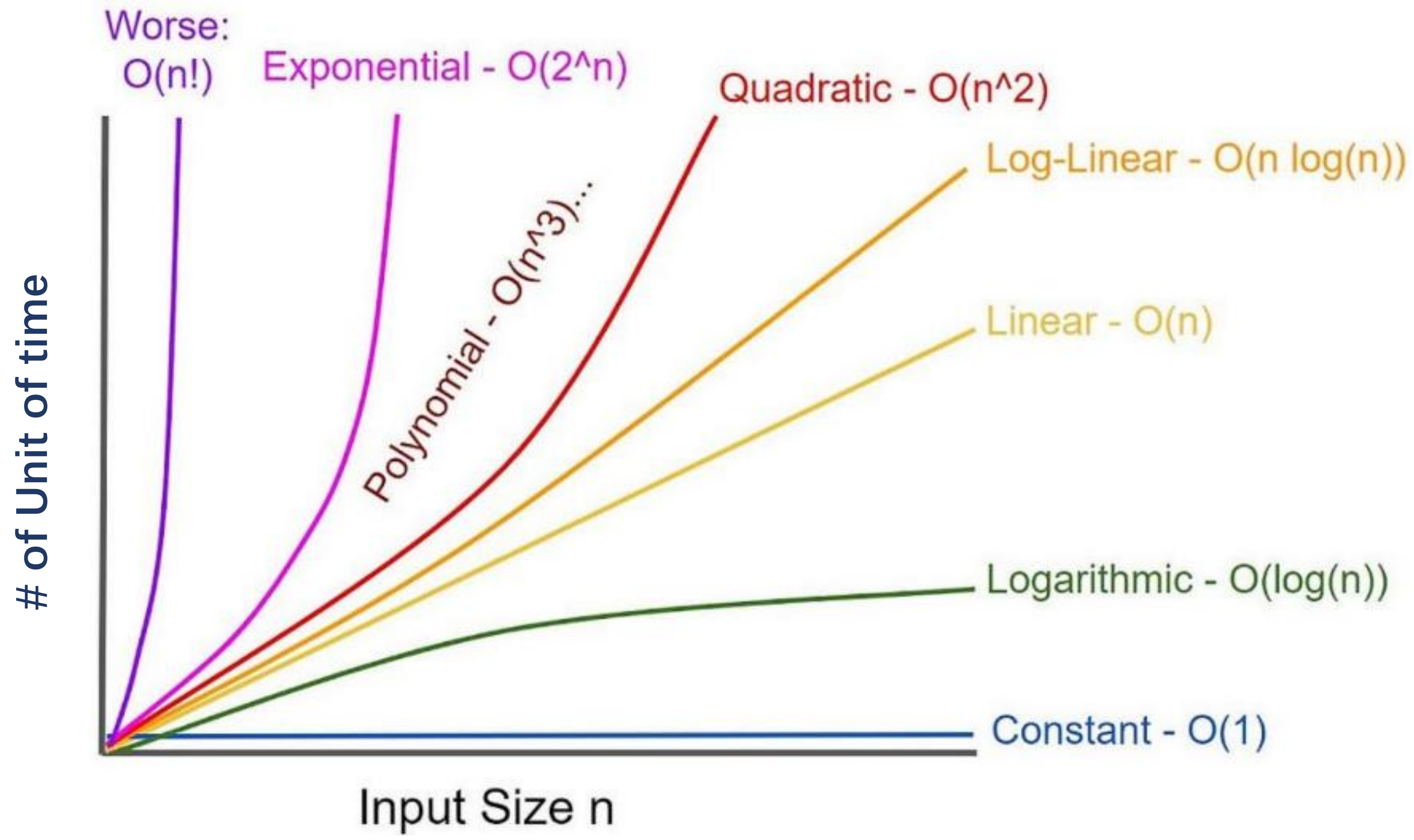
O(N) runtime!

Complexity Classes

- Complexity class: A category of algorithmic efficiency based on the algorithm's relationship to the input size "N".

Class	Big-Oh	If you double N, ...
constant	$O(1)$	unchanged
logarithmic	$O(\log_2 N)$	increases slightly
linear	$O(N)$	doubles
log-linear	$O(N \log_2 N)$	slightly more than doubles
quadratic	$O(N^2)$	quadruples
quad-linear	$O(N^2 \log_2 N)$	slightly more than quadruple
cubic	$O(N^3)$	multiplies by 8
...	...	...
exponential	$O(2^N)$	multiplies drastically
factorial	$O(N!)$	multiplies drastically

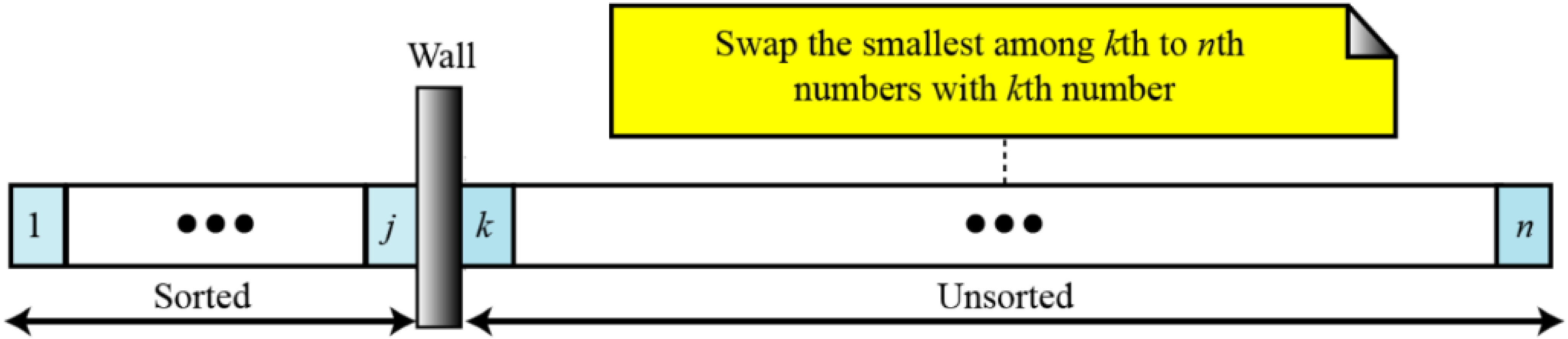
Complexity Classes



Sorting

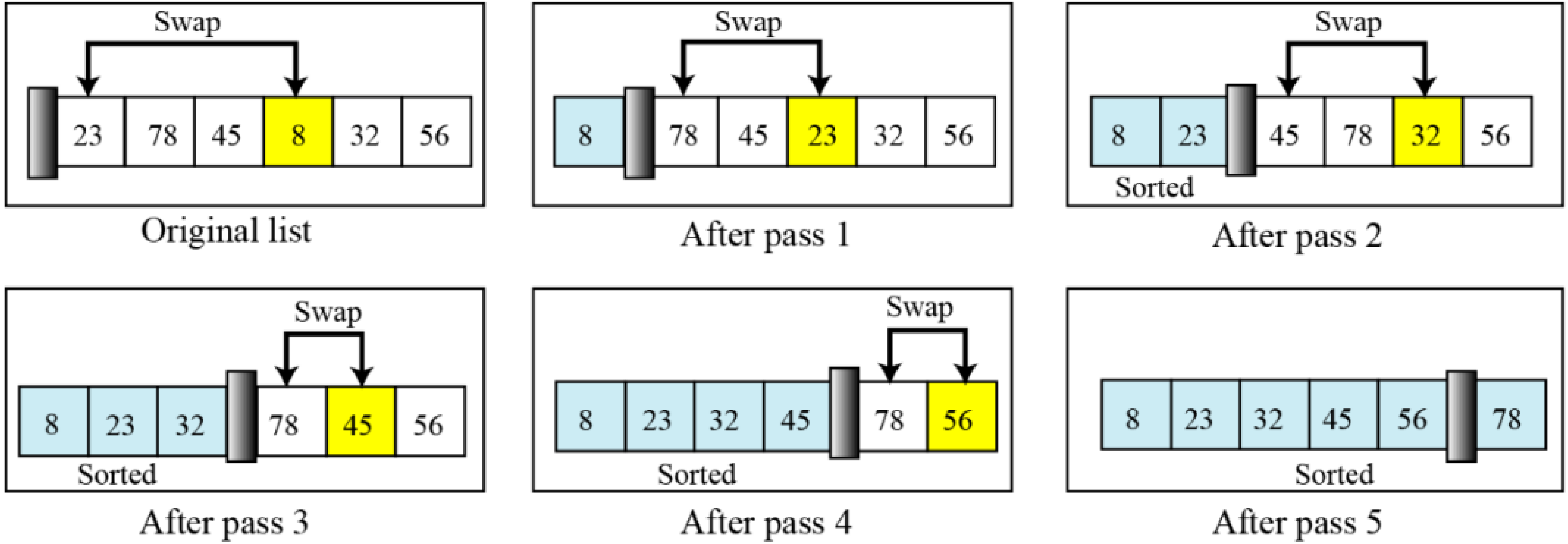
- One of the most common applications in computer science is **sorting**, which is the process by which **data is arranged according to its values**.
- For the beginning, we introduce three sorting algorithms: **selection sort**, **bubble sort** and **insertion sort**.
- These three sorting algorithms are the foundation of faster sorting algorithms used in computer science today.

Selection Sort



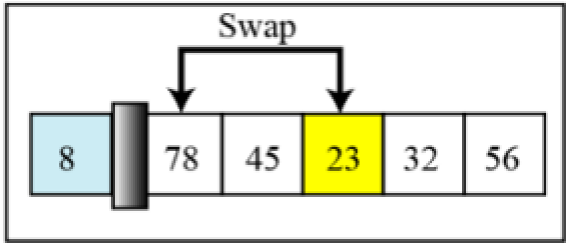
- The list to be sorted is **divided into two sublists**—sorted and unsorted—which are separated by an imaginary wall.
- We **find the smallest element** from the unsorted **sublist** and swap it with the element at the beginning of the unsorted sublist.
- After each selection and swap, **move the imaginary wall** between the two sublists one element ahead

Selection Sort



Implementation of Selection Sort in C++

```
void selectionSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
            int min_idx = i;
            for (int j = i + 1; j < n; ++j) {
                if (arr[j] < arr[min_idx]) {
                        min_idx = j;
                }
            }
            swap(arr[i], arr[min_idx]);
    }
}
```

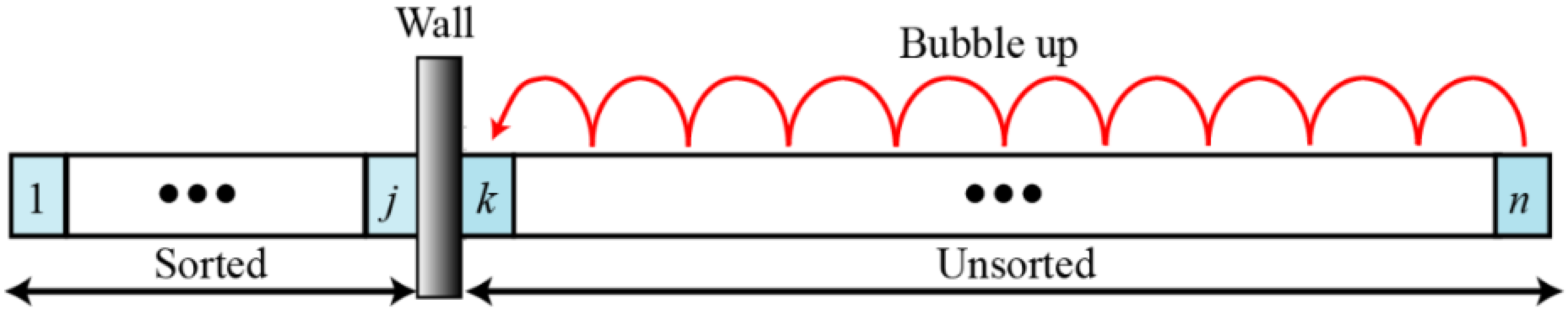


total = 1+(N-1)+(N-1+N-2 +...+1)+(N-1)

Where N-1+N-2 +...+1 = N(N-1)/2 = $N^2/2 - N/2$

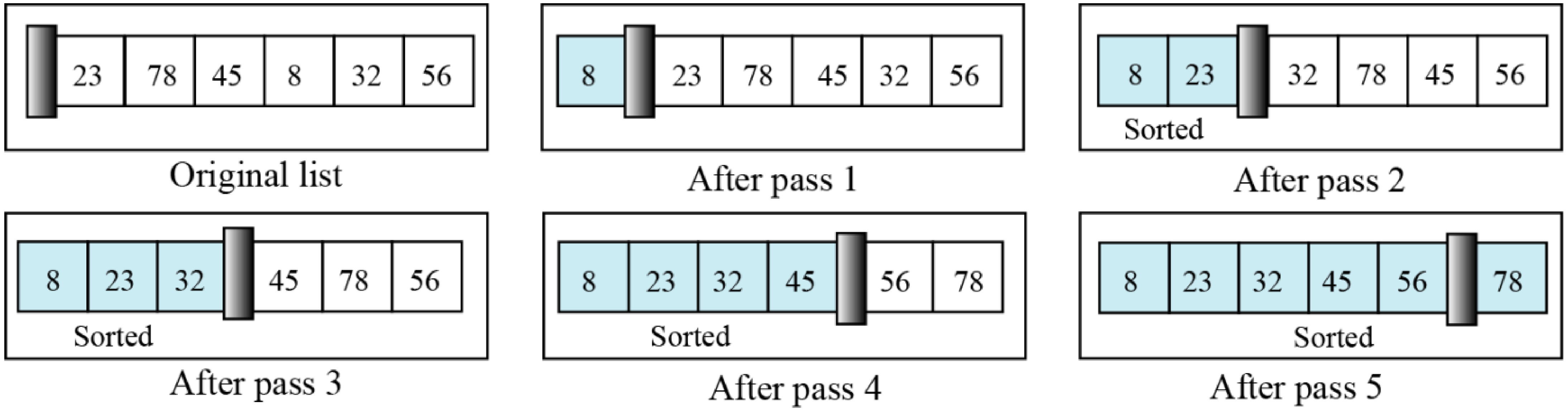
$O(N^2)$ runtime!

Bubble Sort



- The list to be sorted is also **divided into two sublists**—sorted and unsorted.
- The **smallest element is bubbled up** from the unsorted sublist and moved to the sorted sublist.
- After the smallest element has been moved to the sorted list, **move the wall** one element ahead.

Bubble Sort


23 78 45 8 32 56
Original list
8 23 78 45 32 56
After pass 1
8 23 32 78 45 56
Sorted
After pass 2
8 23 32 45 78 56
Sorted
After pass 3
8 23 32 45 56 78
Sorted
After pass 4
8 23 32 45 56 78
Sorted
After pass 5

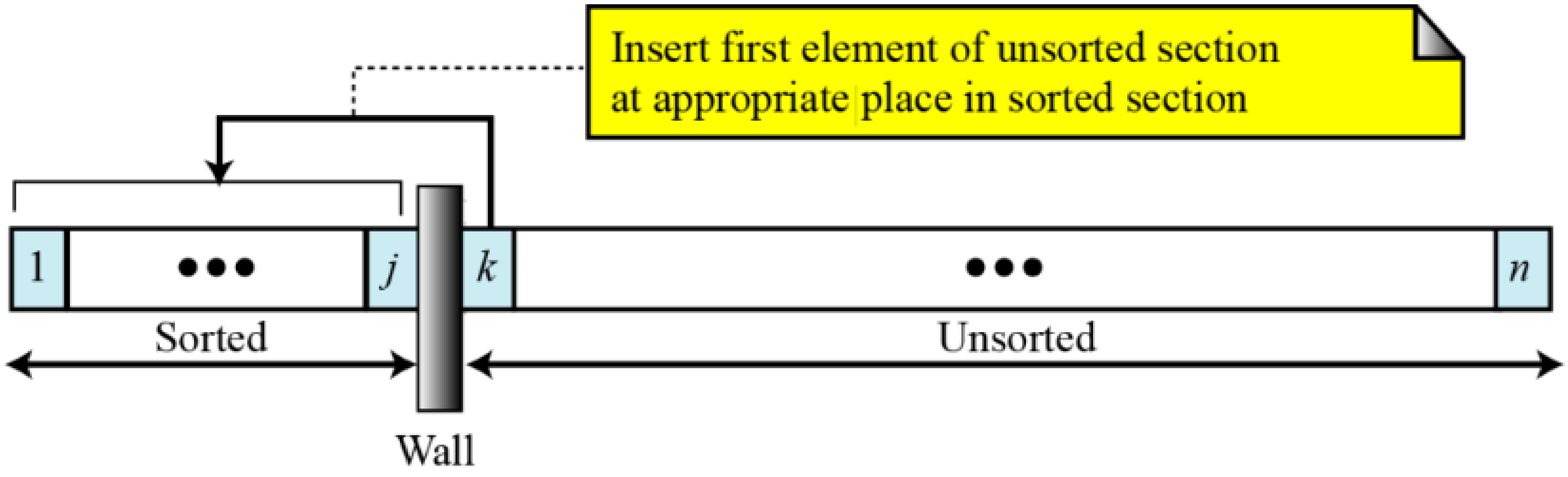
Implementation of Bubble Sort in C++

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

total = 2+(N-1)+(N-1+N-2 +…+1)+(N-1)

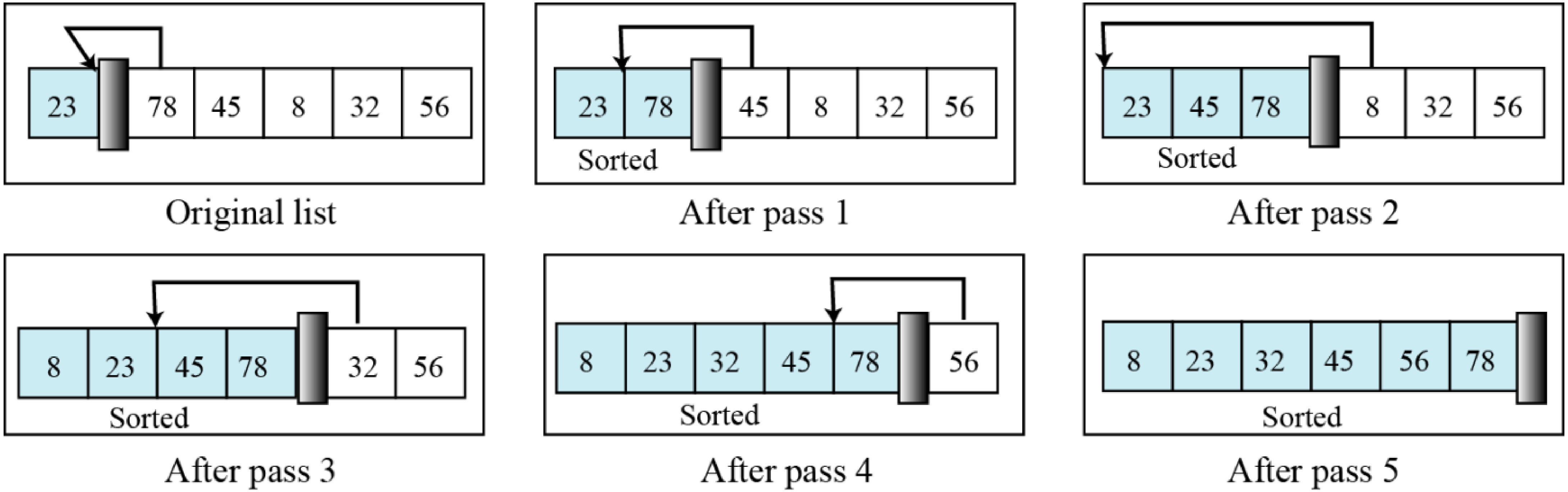
$O(N^2)$ runtime!

Insertion sorts



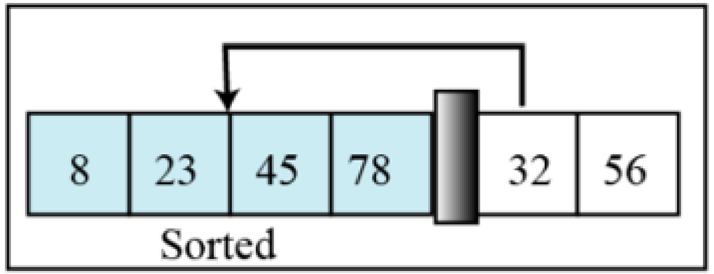
- Common sorting techniques often used by card players.
- Again, the list to be sorted is **divided into two sublists**—sorted and unsorted.
- Each card a player picks up is **inserted into the proper place** in their hand of cards to maintain a particular sequence.

Insertion sorts



Implementation of Insertion Sort in C++

```
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```



total = (N-1)*2+(N-1+N-2+…+1)*2+(N-1)

$O(N^2)$ runtime!

Extra read: More sorting algorithms

Algorithm	Time Complexity (Worst)	Space Complexity (Worst)
Selection Sort	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Heap Sort (Week 12)	$O(n \log_2(n))$	$O(1)$
Quick Sort (Week 12)	$O(n^2)$	$O(n)$
Merge Sort (Week 12)	$O(n \log_2(n))$	$O(n)$
Bucket Sort	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(n + k)$
Count Sort	$O(n+k)$	$O(k)$
Shell Sort	$O(n^2)$	$O(1)$
Tim Sort	$O(n \log_2(n))$	$O(n)$
Tree Sort	$O(n^2)$	$O(n)$
Cube Sort	$O(n \log_2(n))$	$O(n)$

Exercise 4.1

- Complete LeetCode 905

And try analyze the Time Complexity!

905. Sort Array By Parity

Easy Topics Companies

Given an integer array `nums`, move all the even integers at the beginning of the array followed by all the odd integers.

Return ***any array*** *that satisfies this condition.*

Example 1:

```
Input: nums = [3,1,2,4]
Output: [2,4,3,1]
Explanation: The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.
```

Example 2:

```
Input: nums = [0]
Output: [0]
```

Exercise 4.2

- Complete LeetCode 1752

And try analyze the Time Complexity!

1752. Check if Array Is Sorted and Rotated

Easy | Topics | Companies | Hint

Given an array `nums`, return `true` *if the array was originally sorted in non-decreasing order, then rotated* ***some*** *number of positions (including zero)*. Otherwise, return `false`.

There may be **duplicates** in the original array.

Note: An array `A` rotated by `x` positions results in an array `B` of the same length such that `B[i] == A[(i+x) % A.length]` for every valid index `i`.

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: true
Explanation: [1,2,3,4,5] is the original sorted array.
You can rotate the array by x = 3 positions to begin on the element of value 3: [3,4,5,1,2].
```

Example 2:

```
Input: nums = [2,1,3,4]
Output: false
Explanation: There is no sorted array once rotated that can make nums.
```

Exercise 4.3

- Try implementing **selection sort**, **bubble sort**, and **insertion sort** all by yourself **without any hint**!
- Write a testing code to validate your implementation, the code should generate at least three arrays containing unsorted numbers and then check whether your implementations are successful or not.