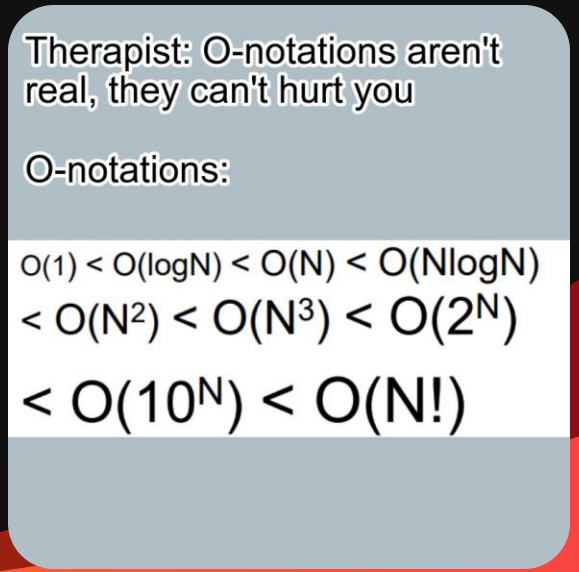
Day 3: Algorithms

+ - % *

TOC

Big O Notation

-



Big O Notation:

What is Big-O Notation?

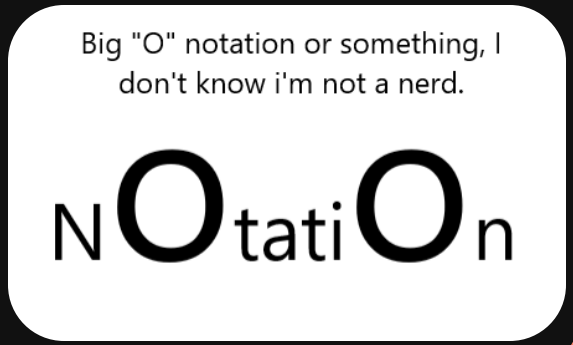
- Used to measure the efficiency of algorithms in terms of time complexity (time it takes to run/number of operations) and space complexity (memory used)

Purpose: Optimize efficiency of algorithm and select the best algorithm for the specific problem and input size

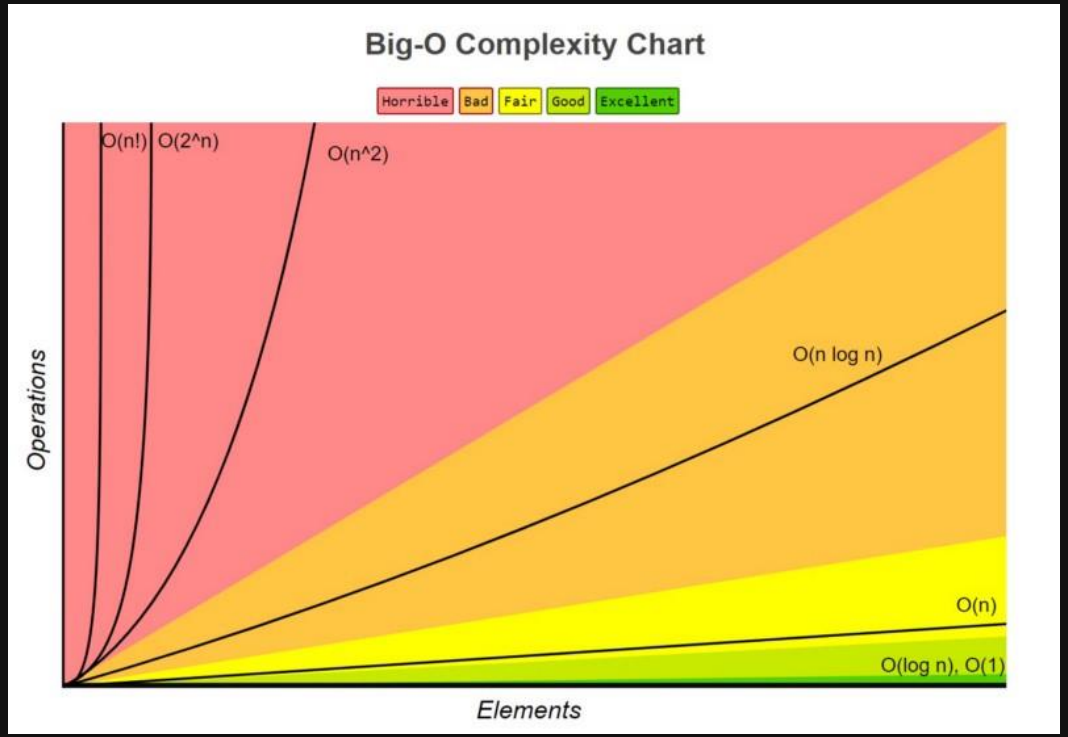
O(x n)

x = constant

n = input size



Big O Notation Chart:



Time Complexity:

O(1) -> **Constant**

O(log n) -> **Logarithmic**

O(n) -> **Linear**

O(n log n) -> **Log-Linear**

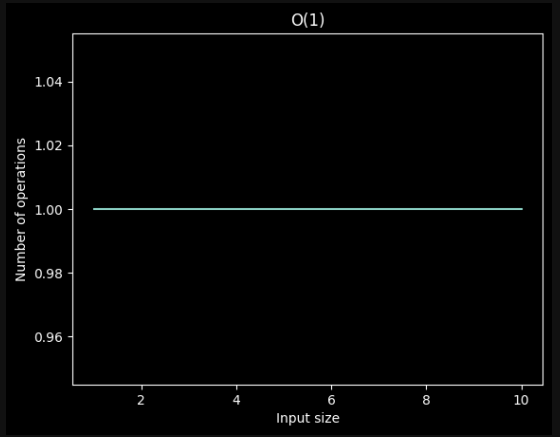
O(n^2) -> **Quadratic**

O(2^n) -> **Exponential**

O(n!) -> **Factorial**

O(1): Constant Time

Algorithm will always take the same amount of time to complete

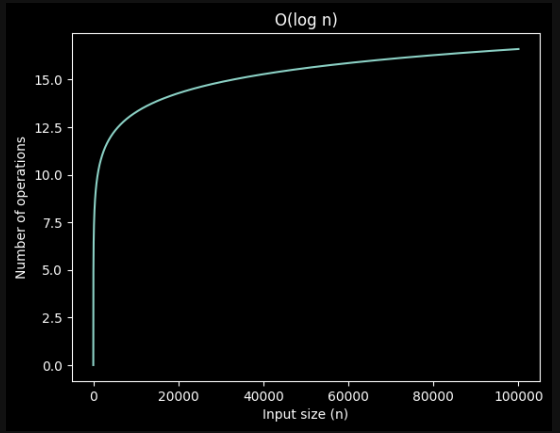


Code Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int x = arr[3];
```

O(log n): Logarithmic Time

As the input size grows larger, the time it takes for the algorithm to run increases at a slower rate

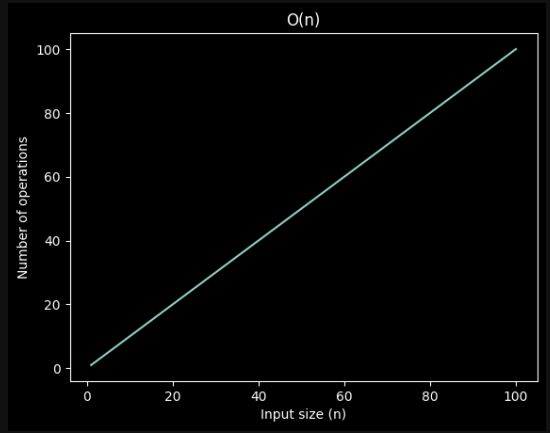


Code Example:

```
int binarySearch(int arr[], int l, int r, int x)
{   while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) {
            return mid;
        }
        if (arr[mid] < x) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return -1;
}
```

O(n): Linear Time

Time taken for algorithm to run will increase linearly

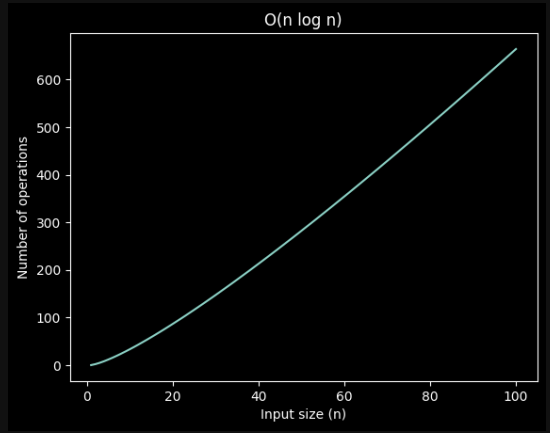


Code Example:

```
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += arr[i];
}
```

O(n log n): Log-Linear Time

As the input size grows, the algorithm takes longer to run, but the rate at which the running time increases slows down due to the logarithm growing much more slowly than the input size



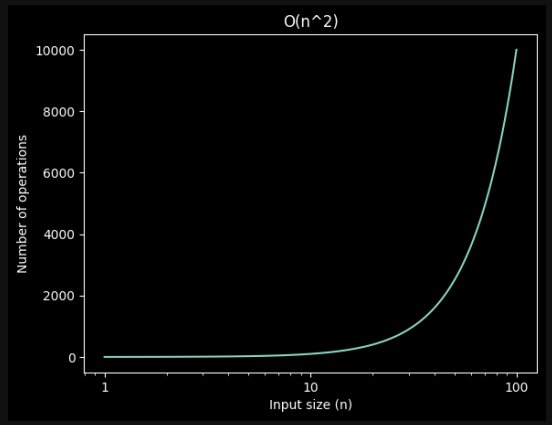
Code Example:

```cpp
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 5;
    int index = binarySearch(arr, 0, n - 1, x);
    if (index != -1) {
        cout << "Element " << x << " found at index " << index << endl;
    } else {
        cout << "Element " << x << " not found in the array" << endl;
    }
    return 0;
}
```

binarySearch has a time complexity of log n (code in O(log n) slide)

O(n^2): Quadratic Time

Running time of the algorithm increases as the square of the input size

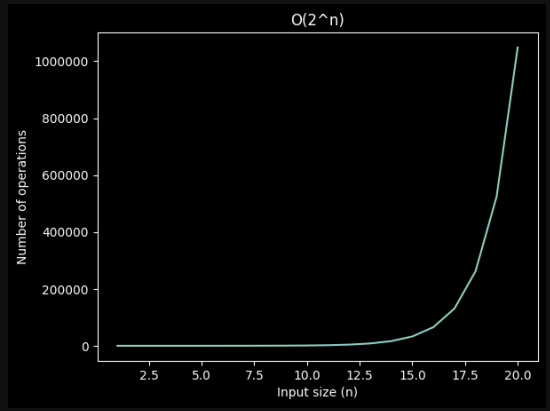


Code Example:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

O(2^n): Exponential Time

Algorithm's running time doubles with every additional input element

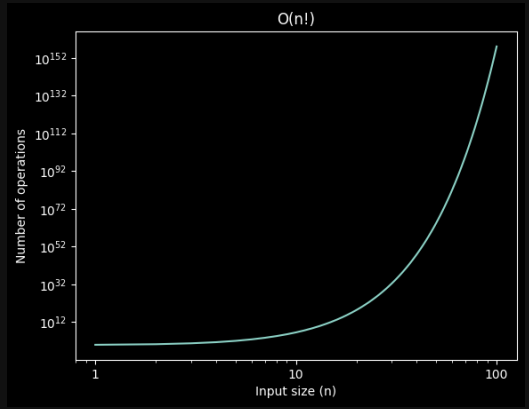


Code Example:

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

O(n!): Factorial Time

Algorithm's running time increases as a factorial of the input size



Code Example:

```
void generatePermutations(string str, int start, int end) {
    if (start == end) {
        cout << str << endl;
    } else {
        for (int i = start; i <= end; i++) {
            swap(str[start], str[i]);
            generatePermutations(str, start + 1, end);
            swap(str[start], str[i]);
        }
    }
}
```

Case Study: Fibonacci Sequence

O(1) vs O(n) vs O(2^n)

Fibbonaci Sequence

O(1) Solution:

This solution is the most efficient approach to find the n^{th} Fibonacci number in the Fibonacci sequence as it only requires constant time to compute any Fibonacci number

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

int fibonacci_constant(int n) {
    double phi = (1 + sqrt(5)) / 2;
    return round(pow(phi, n) / sqrt(5));
}
```

Fibbonaci Sequence

O(n) Solution:

Less efficient compared to O(1)

Calculation:

The time complexity of the loop dominates the time complexity of the function.

Time complexity of loop:

- Loop iterates from 2 to n: O(n-2)

- Perform constant number of operations: O(1)

- Total: O(n-2) * O(1) = O(n-2) = O(n)

```
int fibonacci_linear(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        int Fn_2 = 0;
        int Fn_1 = 1;
        int Fn = 0;
        for (int i = 2; i <= n; i++) {
            Fn = Fn_2 + Fn_1;
            Fn_2 = Fn_1;
            Fn_1 = Fn;
        }
        return Fn;
    }
}
```

Fibbonaci Sequence

O(2^n) Solution:

Least efficient compared to the previous two solutions

The number of operations double with every function call as two additional recursive calls are made each time

Calculation:

T(n)

= (Recursive Call 1 + Recursive Call 2) + Base Case

= (T(n-1) + T(n-2)) + O(1)

= O(2^n)

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

int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Sorting Algorithms

Comparison Based + Non-Comparison Based Sorts

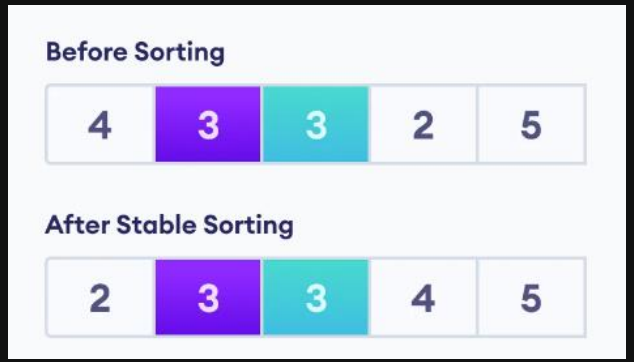
Sorting Algorithms

- Sorting algorithms are used to arrange elements in an array/list in a specific order
- Some of the common sorting algorithms are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quicksort
  - Merge Sort
  - Counting Sort
  - Radix Sort
- Based on our needs, we will implement different sorting algorithms to perform the sort

Sorting Algorithms

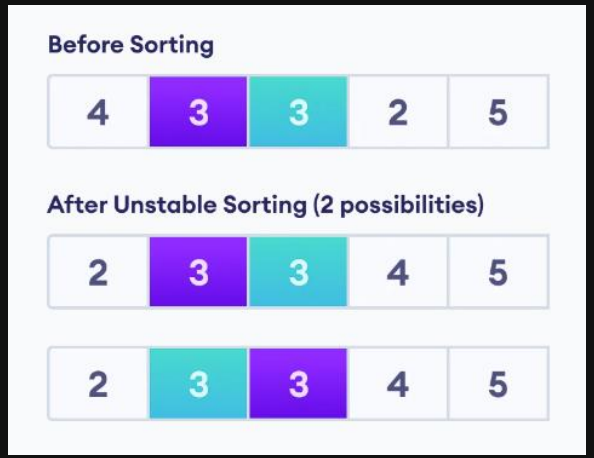
- Different sorting algorithms have different time complexity and stability
- Stable sorting algorithms allow items to remain in the same order as they are in the unsorted array after sorting while items might not be in the same order after sorting when using unstable sorting algorithm

Stable Sort



vs

Unstable Sort

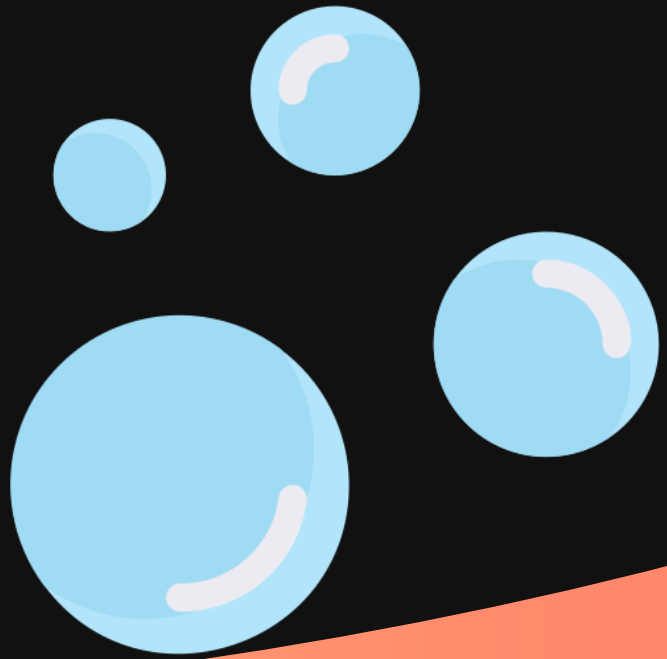


Sorting Algorithms

- Sorting algorithms are usually implemented in one of the following ways:
  - Iterative Method: Make use of loops to repeat some parts of code
  - Recursive Method: Call the function repeatedly to repeat the code
- Common approach for the following few sorting algorithms:
  - Bubble Sort: Iterative
  - Selection Sort: Iterative
  - Insertion Sort: Iterative
  - Quicksort: Recursion
  - Merge Sort: Recursion
  - Counting Sort: Iterative
  - Radix Sort: Iterative

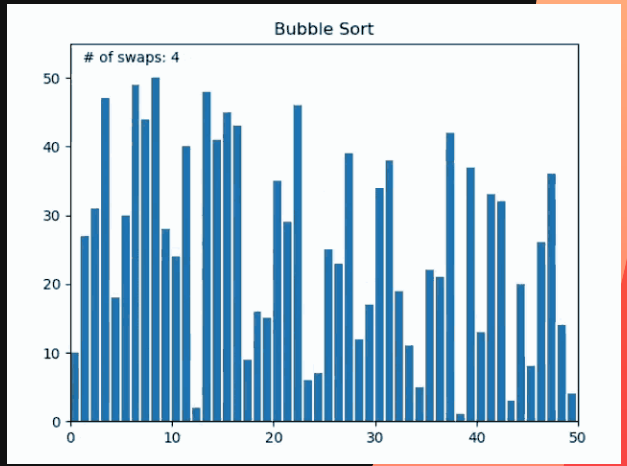
VS

Bubble Sort



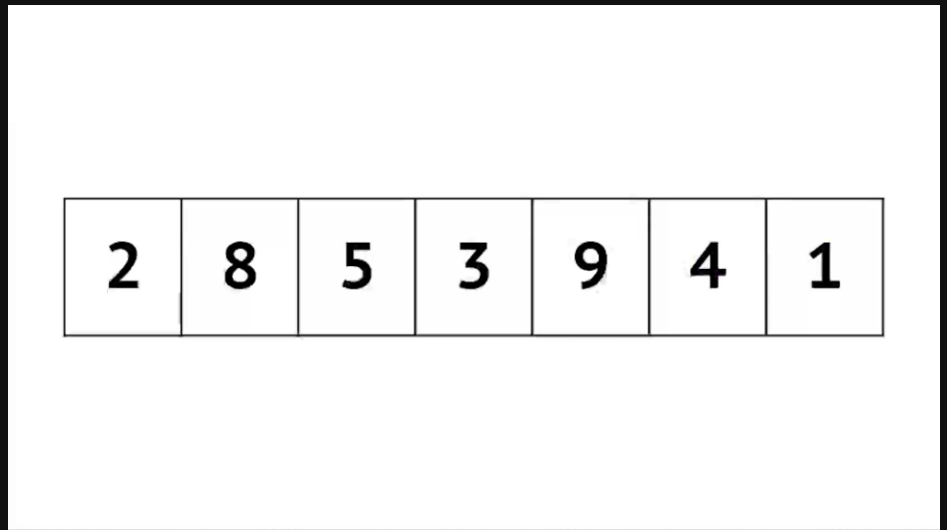
Bubble Sort: Comparison-Based

- Bubble sort iterates throughout the list and compare two adjacent elements and swap them when needed
- It would loop through the entire array/list until it is completely sorted
- Stability: Stable
- Time Complexity:
  - Best Case: O(n)
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$
- Use Cases: Most efficient on smaller sets of elements



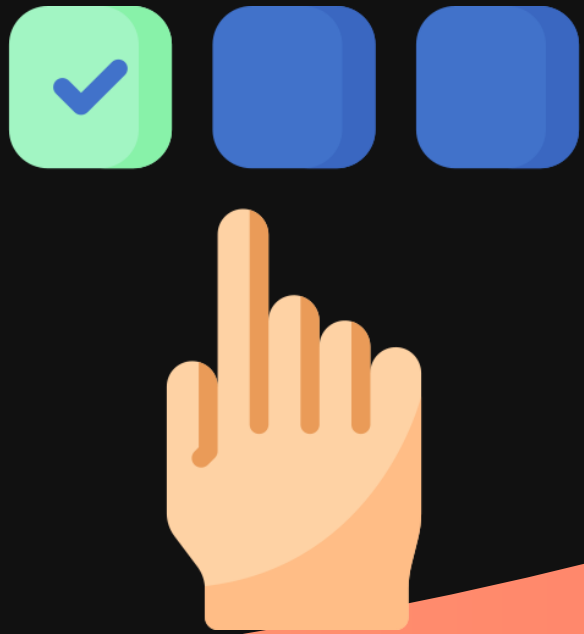
Bubble Sort - Pseudocode

- Loop to access each item in array
  - Inner loop to compare unsorted array items
    - For ascending order, swap each item if the item on the left is greater than the item on the right



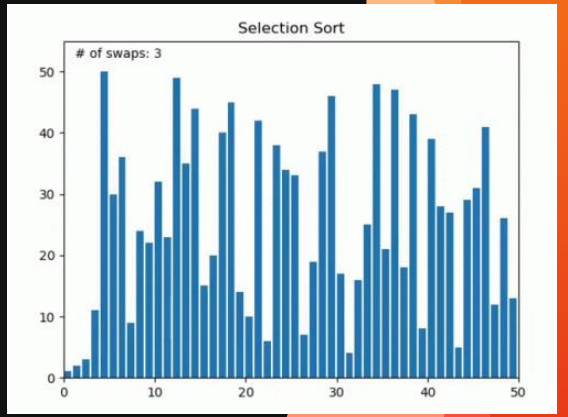
Bubble Sort Demo

Selection Sort



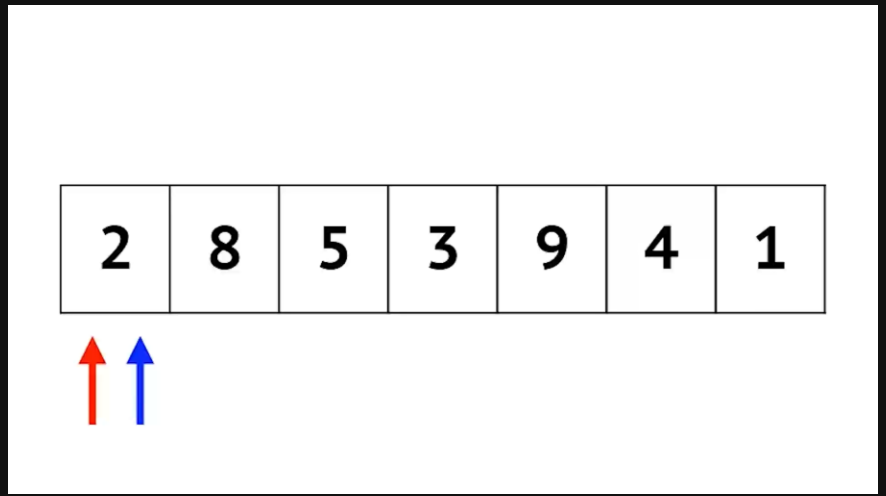
Selection Sort: Comparison-Based

- In each iteration, selection sort **selects the smallest item** from the unsorted part of the list and **places it at the beginning** of the unsorted part of the list
- Stability: Unstable
- Time Complexity: $O(n^2)$
- Use Case:
  - Small data set
  - Need to check through all elements
  - Original sequence of elements not needed to remain the same



Selection Sort - Pseudocode

- Loop for (size of array – 1) times
  - Set first item of unsorted array as minimum
  - Inner loop to iterate over every item in the unsorted part of array
    - Find the index of the minimum item and set it as new minimum
  - Swap the first item of unsorted array with the minimum item



Current minimum

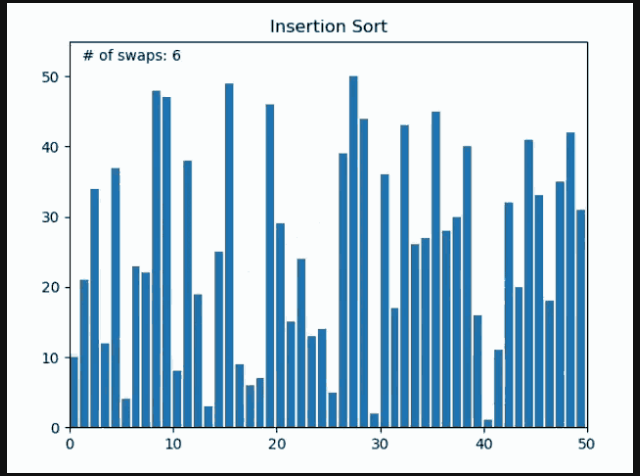
Current item

Selection Sort Demo

Insertion Sort

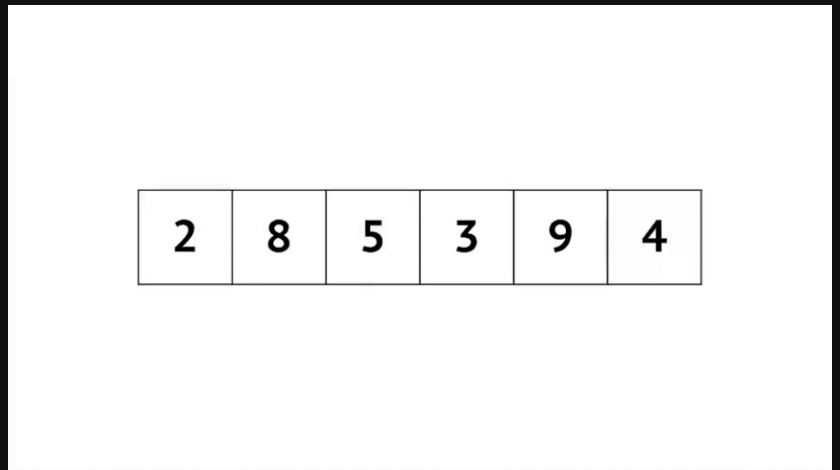
Insertion Sort: Comparison-Based

- In each iteration of insertion sort, the algorithm **selects an item** from the unsorted part of the list and **inserts it into its correct position** within the sorted part of the list.
- Stability: Stable
- Time Complexity:
  - Best Case: O(n)
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$
  - Use Case:
  - Small data set
  - Only a few elements to be sorted



Insertion Sort - Pseudocode

- Assume the first element is sorted
- Loop from i = 1 to i = (size of array – 1)
  - Select the first element in the unsorted part of array
  - Loop to iterate over every item in the unsorted part of array and place it behind the element smaller than it

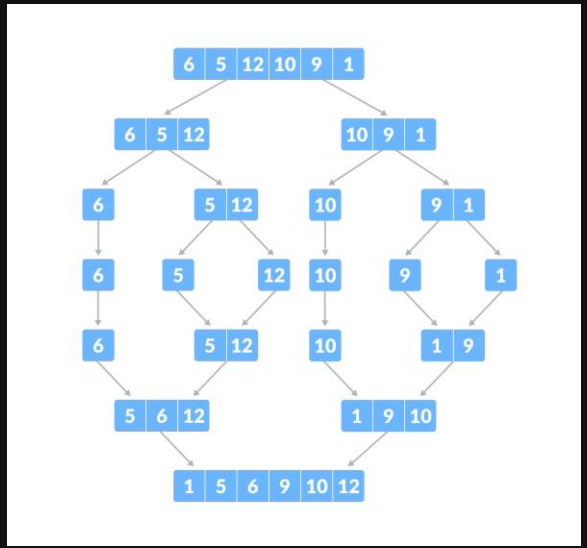


Insertion Sort Demo

Merge Sort

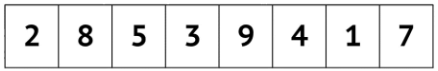
Merge Sort: Comparison-Based

- Merge sort would split an array in half
- These subarrays would then continue to be split in half until it cannot be further divided
- Once the base case is reached, it would start merging into a sorted array
- Stability: Stable
- Time Complexity: O(n logn)
- Use Case: inversion count problem, external sorting



Merge Sort - Pseudocode

- Find the middle of the array
- Store first half of array in a variable, and second half in another variable
- Call the merge sort function to sort the first half
- Call the merge sort function to sort the second half
- Use while loop to merge both sorted arrays into the final array
- Check if any items are left in both arrays, and add them into the final array



Merge Sort Demo

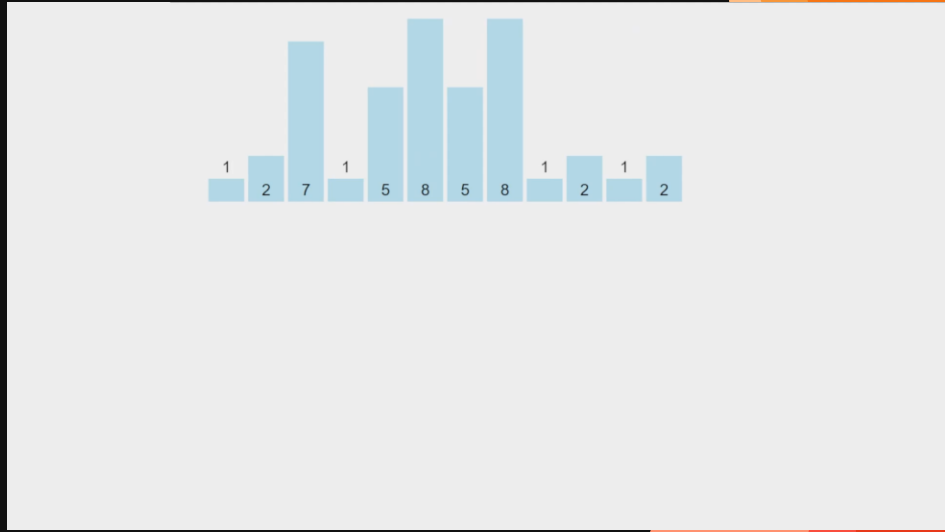
Counting Sort

Counting Sort: Non-Comparison-Based

- The algorithm sorts the elements of an array by **counting the number of occurrences of each unique element** in the array
- Stability: Stable
- Time Complexity: O(n + k)
- Use Case:
  - Sorting integers within a small range
  - Need linear complexity

Counting Sort - Pseudocode

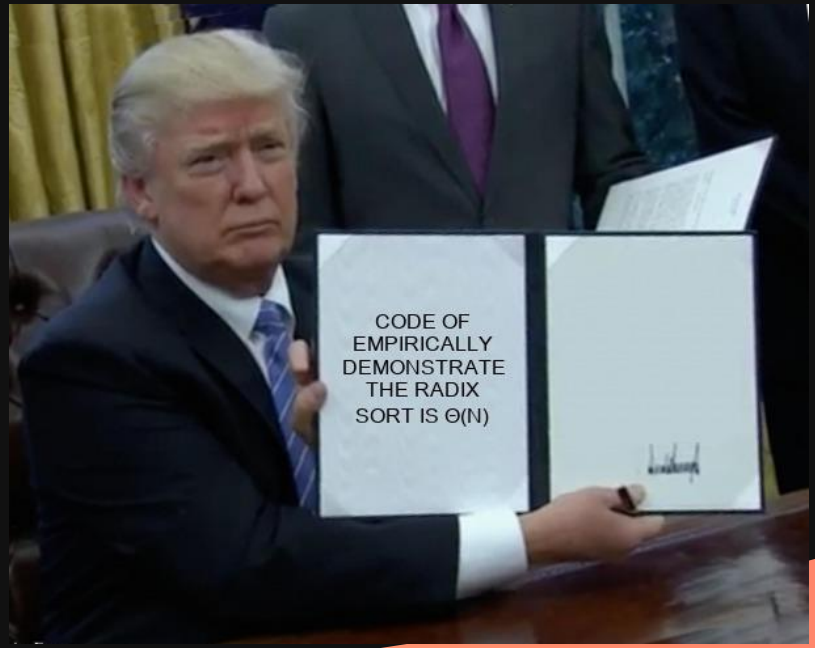
- Count the number of occurrences of each element in the input array
- Compute the cumulative sum of each element in the input array and update the count stored in another array accordingly
- Create an empty array for output and populate it
- Copy sorted array back to original array



Counting Sort Demo

Break (10mins)

Radix Sort

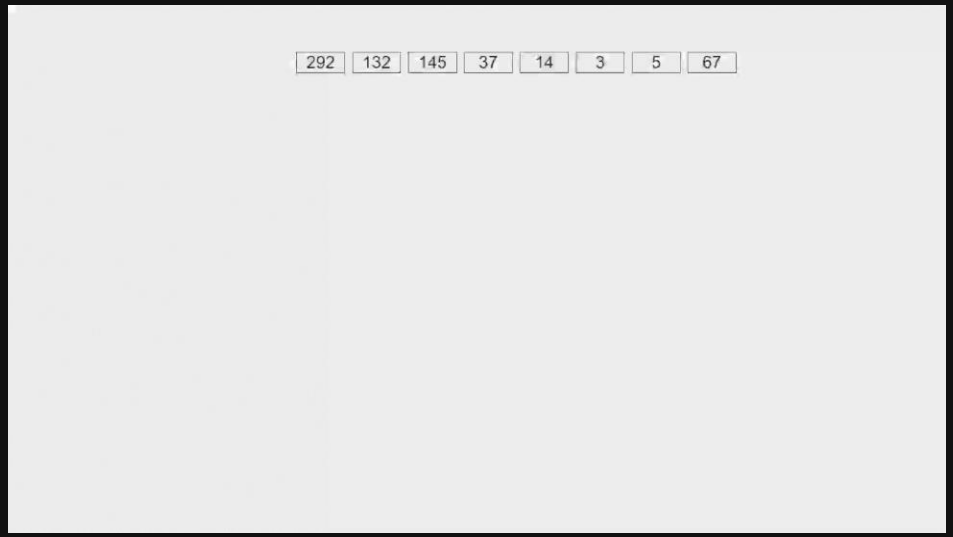


Radix Sort: Non-Comparison-Based

- The algorithm sorts the elements by first **grouping the individual digits of the same place value.** For example, starting with the unit place and followed by the tens place
- Stability: Stable
- Time Complexity: O(n + k)
- Use Case:
  - Sorting integers with a large range
  - Need linear complexity

Radix Sort - Pseudocode

- Start with finding the largest element
- Perform counting sort on each digit, starting from the least significant digit being the unit place



Radix Sort Demo

Searching Algorithms

Linear Search, Binary Search

Linear Search

- Linear Search is the simplest searching algorithm which searches elements by traversing the entire array
- Time Complexity: O(n)
- Use Case:
  - For searching elements in small arrays (e.g. 100 items)

Code Example:

```
int linearSearch(int arr[], int n, int target) {
    // traverse the array from the beginning to the end
    for(int i = 0; i < n; i++) {
        // if the current element is equal to the target, return its index
        if(arr[i] == target) {
            return i;
        }
    }
    // if the target is not found, return -1
    return -1;
}
```

Binary Search

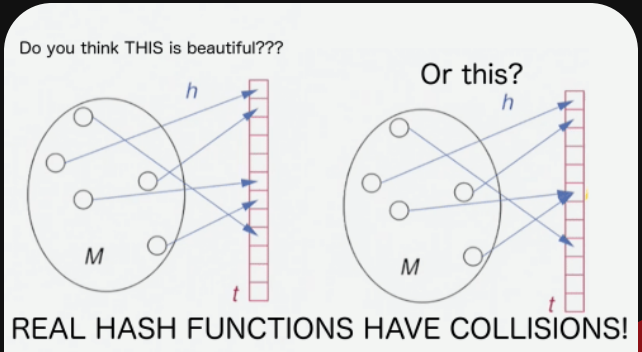
- Binary Search is an efficient searching algorithm as it repeatedly half the search interval until either the target is found or when the search interval becomes empty
- Time Complexity:
  - Best Case: O(1)
  - Average Case: O(log n)
  - Worst Case: O(log n)

Binary Search

```
int binarySearch(vector<int> arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    // Loop until the search interval is empty.
    while (left <= right) {
        int mid = (left + right) / 2;

        // If target is found, return its index.
        if (arr[mid] == target) {
            return mid;
        }
        // If target is greater than mid, search the right half.
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        // If target is less than mid, search the left half.
        else {
            right = mid - 1;
        }
    }

    // Target value not found in array.
    return -1;
}
```

Hashing

###



Real hash functions have collisions! Don't listen to the propaganda of what a "perfect" hash function should look like!

Hashing

- Hashing is the process of using a hash function to convert input data into a hash code to represent the original data which then are used to map input data to their corresponding index positions in the table.
- **Hash Table:**
  - An array of fixed size
  - Each element is the array is called a bucket
  - Each bucket can contain one or more key-value pairs
  - Data are converted into hash code and the hash code is used as an index into the array and data inserted into the corresponding bucket

  - **Advantages:** Fast access to data
  - **Disadvantages:** Potential of collision, cost of memory allocation and rehashing when table becomes full

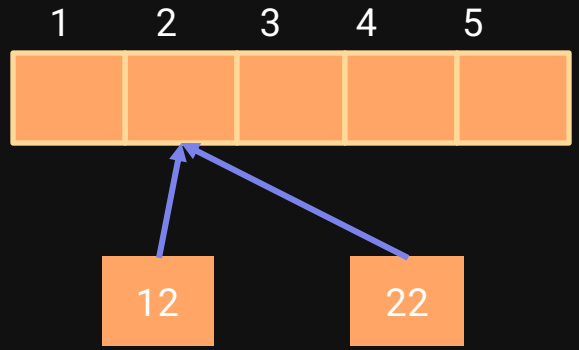
Hashing: Hash Functions

- Hash functions should be easy and fast to compute
- A perfect hash function would generate a unique hash code for each unique input key, but it is difficult to practice in reality

- When two unique input key are map to the same hash code, it is called a **collision**

Collision Example->

Hash Function: input % 5



Hashing: Handling Collisions

- There are multiple ways to handle collisions like increasing table size, choosing a better hash function, linear probing, double hashing etc.
- We will be focusing on making use of **separate chaining** to handle collisions

- This suggest that every array element/bucket is a linked list. When any collision occurs, we will add them to their respective linked list.

Hashing: Hashing Strings

- **There are several techniques to hash strings.**
- **Some of the common ones are:**
  - Polynomial Hashing
  - Rolling Hashing
  - Cryptographic Hashing
  - **Character-Based Hashing**

- **Example of Character-Based Hashing:**

Let ‘A' = 0, 'B' = 1, ‘C’=2, ‘D’=4 …, 'Z' = 25, 'a' = 26, …, 'z' = 51

hash(“Tim") = 19 * 523^2 + 34 * 523^1 + 38 * 523^0

= 1002883

Hashing: Hashing Functions

- These are some common hashing functions for numeric values
- Every hash functions have their desired use case like desired level of collisions and performance

- Some Common Hashing Functions and Suggested Use Cases:
  - **The Mod Method:** Simple and fast method and when the input data is already evenly distributed
  - **The Multiplication Method:** High degree of uniformity in the hash values is desired
  - **The Mid Square Method:** Input data is evenly distributed and uniformly random
  - **The Folding Method:** Input data is not uniformly random, but still exhibits some regularity (e.g. Phone Numbers)

Hashing Demo

Dynamic Programming

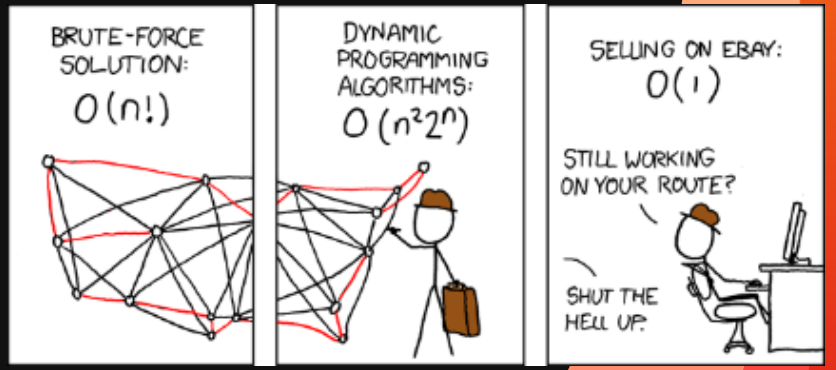
Make things fast vroom vroom

Dynamic programming

- Technique to solve an optimization problem by breaking it into simpler subproblems
- Also, avoid computing the same problems repeatedly by storing the results
- Results are stored in a "lookup table"

There are two forms of results storage in DP:

- Tabulation (bottom-up)
- Memoization (top-down)



Tabulation vs Memoization

Memoization	Tabulation
Top to Down	Down to Top
Lookup Table filled on demand	Lookup table filled one by one
Slower due to many recursive calls	Fast, access Previous state

Fibonacci

- Fibonacci is a simple example to illustrate DP
- Each sequence is the sum of the previous two sequences

Fib(0) = 0

Fib(1) = 1

Fib(2) = Fib(1) + Fib(0) = 1

Fib(3) = Fib(2) = Fib(1) = 2

Fib(4) = Fib(3) = Fib(2) = 3

Fibonacci memoization

- Let’s say we want to solve Fib(10)

  Fib(10) = Fib(9) + Fib(8)

  Fib(9) = Fib(8) + Fib(7)

  Fib(8) = Fib(7) + Fib(6)

  ...

- Notice duplicate calculations required
- By storing the values, we can avoid recomputing them

```cpp
// Dynamic programming implementation of Fibonacci sequence
int fibonacci_dp(int n, std::unordered_map<int, int>& memo) {
    if (memo.find(n) != memo.end())
        return memo[n];
    if (n <= 1)
        return n;
    int result = fibonacci_dp(n - 1, memo) + fibonacci_dp(n - 2, memo);
    memo[n] = result;
    return result;
}
```

Fibonacci tabulation

- Let’s say we want to solve Fib(10)
- To calculate Fib(10), we need Fib(1), Fib(2) … Fib(9)
- We can calculate from the bottom up, and store the values

Fibonacci tabulation

```
int fib(int n) {
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}

int main () {
    int n = 9;
    cout << fib(n);
    return 0;
}
```

Thank you!