Algorithms Handbook

Vladimir Bolshakov

Contents

Bubble sort

Bubble Sort: A Simple Sorting Algorithm

Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.

How Bubble Sort Works:

1. **Comparing Adjacent Elements:**
   - Bubble Sort starts by comparing the first two elements of an array. If the first element is greater than the second, they are swapped. If not, they remain in their positions.
2. **Iterative Process:**
   - This process is then repeated for every pair of adjacent elements throughout the entire array. After the first iteration, the largest element will have "bubbled up" to the last position.
3. **Subsequent Passes:**
   - The algorithm then repeats the process for the remaining elements (excluding the already sorted ones at the end of the array). In each pass, the next largest element is placed in its correct position.
4. **Termination:**
   - The algorithm terminates when a pass through the entire array is made without any swaps, indicating that the array is now sorted.

Time Complexity: - Bubble Sort has a time complexity of O(nˆ2) in the worst and average cases, where 'n' is the number of elements in the array. This makes it inefficient for large datasets but is useful for educational purposes due to its simplicity.

```
function bubbleSort(array: number[] | string[]) {
    for (let i = 0; i < array.length; i++) {
        for (let j = 0; j < array.length - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                [array[j], array[j + 1]] = [array[j + 1], array[j]];
            }
        }
    }
    return array;
}

console.log(bubbleSort([2,5,2,6,7,2,22,5,7,9,0,2,3]))
```

```
    public static void bubbleSort(int[] array) {
        for(int i = 0; i < array.length - 1; i++) {
            for(int j = 0; j < array.length - i - 1; j++) {
                if(array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
```

Selection sort

Selection Sort is a straightforward sorting algorithm that works by dividing the input array into two parts: the sorted and the unsorted subarrays. The algorithm repeatedly selects the minimum (or maximum, depending on the sorting order) element from the unsorted subarray and swaps it with the first unsorted element. This process is iteratively applied until the entire array is sorted.

How Selection Sort Works:

1. **Dividing the Array:**
    - The algorithm starts with the entire array considered as unsorted.
2. **Finding the Minimum Element:**
    - In each iteration, Selection Sort finds the minimum element from the unsorted part of the array.
3. **Swapping:**
    - Once the minimum element is identified, it is swapped with the first element in the unsorted part, effectively extending the sorted subarray.
4. **Iterative Process:**
    - The above steps are repeated for the remaining unsorted part of the array until the entire array is sorted.

```
function selectionSort(array: any[]) {
  for (let i = 0; i < array.length - 1; i++) {
    let min = i;
    for (let j = i + 1; j < array.length; j++) {
      if (array[min] > array[j]) min = j;
    }
    [array[i], array[min]] =[array[min], array[i]]
  }
  return array;
}

console.log(selectionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));
```

```
    public static void selectionSort(int[] array) {
        for(int i = 0; i < array.length - 1; i++) {
            int min = i;
            for(int j = i + 1; j < array.length; j++) {
                if(array[min] > array[j]) {
                    min = j;
                }
            }
            int temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
```

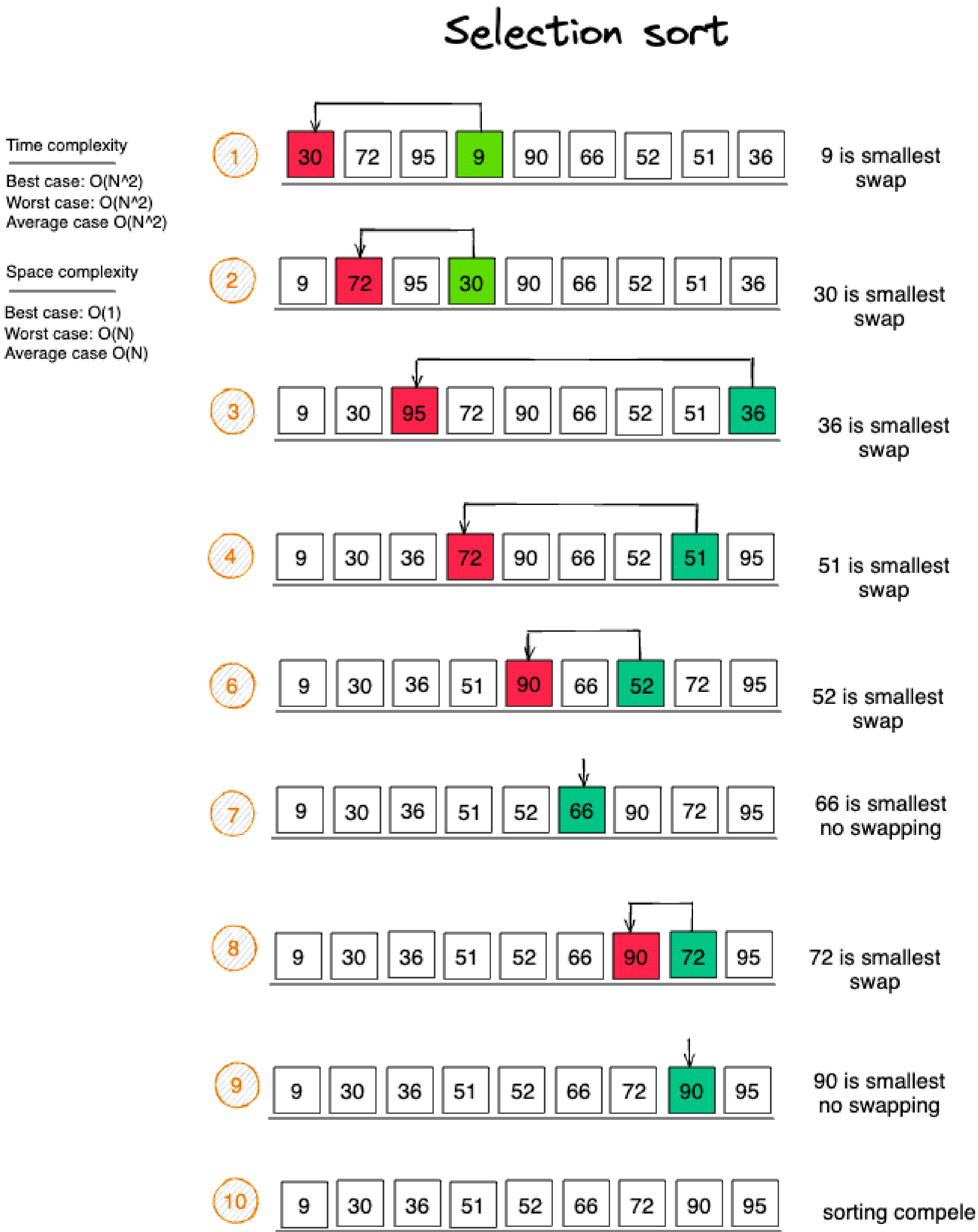

Figure 1: Selection sort

```python
print('This is selection sort')

def find_smallest(arr):
    smallest = arr[0]
    smallest_index = 0
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index

def selection_sort(arr):
    newArr = []
    for i in range(len(arr)):
        smallest = find_smallest(arr)
        newArr.append(arr.pop(smallest))
    return newArr

print(selection_sort([5,4,6,2,1,123, 2, 3,1,23 ,1,1,]))
```

Insertion sort

Insertion Sort is a straightforward sorting algorithm that builds the sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has some advantages: it is simple to implement, efficient for small datasets, and performs well for partially sorted arrays.

How Insertion Sort Works:

1. **Dividing the Array:**
   - The algorithm starts with the first element of the array considered as the sorted part.
2. **Inserting Elements:**
   - For each element in the unsorted part of the array, Insertion Sort compares it with the elements in the sorted part.
   - It then inserts the element into its correct position in the sorted part, shifting the other elements if necessary.
3. **Iterative Process:**
   - This process is repeated until all elements are sorted.

Time Complexity: - Insertion Sort has a time complexity of O(n^2) in the worst case, where 'n' is the number of elements in the array. Despite its quadratic time complexity, Insertion Sort is often more efficient on small datasets or partially sorted arrays compared to other quadratic sorting algorithms. It's also an in-place sorting algorithm, meaning it doesn't require additional memory.

```
function insertionSort(array: number[] | string[]) {
    for (let i = 1; i < array.length; i++) {
        let curr = array[i];
        let j = i - 1;
        for (j; j >= 0 && array[j] > curr; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = curr;
    }
    return array;
}

console.log(insertionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));
```

```
    class Solution {
        void insertionSort (int[] arr) {
            int n = arr.length;
            for(int i = 1; i < n; i++) {
                int current = arr[i];
                int position = i - 1;
                while(position >= 0 && arr[position] > current) {
                    arr[position + 1] = arr[position];
                    position--;
                }
                arr[position + 1] = current;
            }
        }
    }
```

Quick sort

Quick Sort is an efficient, comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

How Quick Sort Works:

1. **Choosing a Pivot:**
   - The algorithm selects a pivot element from the array. The choice of pivot can affect the efficiency of the algorithm.
2. **Partitioning:**
   - Elements smaller than the pivot are moved to its left, and elements greater than the pivot are moved to its right. The pivot is now in its final sorted position.
3. **Recursive Sorting:**
   - The algorithm is applied recursively to the sub-arrays on the left and right of the pivot until the entire array is sorted.

Time Complexity: - Quick Sort has an average and best-case time complexity of O(n log n), where 'n' is the number of elements in the array. In the worst case, it is O(n^2), but this is rare when a good pivot selection strategy is used. Quick Sort is often faster in practice than other O(n log n) algorithms, and it is widely used in various applications due to its efficiency.

```
class Solution {

    int makePartition(int [] arr, int low, int high) {
        int pivot = arr[high];
        int currentIndex = low - 1;
        for(int i = low; i < high; i++) {
            if(arr[i] < pivot) {
                currentIndex++;
                int temp = arr[i];
                arr[i] = arr[currentIndex];
                arr[currentIndex] = temp;
            }

        }
        int temp = arr[high];
        arr[high] = arr[currentIndex + 1];
        arr[currentIndex + 1] = temp;
        return currentIndex + 1;
    }

    void quicksort(int[] arr, int low, int high) {
        if(low < high) {
            int pivot = makePartition(arr, low, high);
            quicksort(arr, low, pivot - 1);
            quicksort(arr, pivot + 1, high);
        }
    }
```

```java
    void quickSort (int[] arr) {
        int n = arr.length;
        quicksort(arr, 0, n - 1);
    }
}
```

```python
def quicksort(arr):
    if len(arr) < 2:
        return arr
    else:
        pivot = arr[len(arr)/2]
        less = [i for i in arr[1:] if i <= pivot]
        greater = [i for i in arr[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10,2,3,1,5,4]))
```

```java
class Solution {
    static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void quickSort(int[] array, int start, int end) {
        if(end <= start) return; // base case

        int pivot = partition(array, start, end);

        quickSort(array, start, pivot -1);
        quickSort(array, pivot + 1, end);
    }

    private static int partition(int[] array, int start, int end) {
        int pivot = array[end];

        int i = start - 1;

        for(int j = start; j <= end -1; j++) {
            if(array[j] < pivot) {
                i++;
                swap(array, i, j);
            }
        }
        i++;
        swap(array, i, end);

        return i;
    }
}
```

```typescript
function quicksort(arr: number[]): number[] {
  if (arr.length < 2) {
    return arr;
  } else {
    const pivot = arr[Math.floor(arr.length / 2)];
    const less = arr.slice(1).filter((i) => i <= pivot);
    const greater = arr.slice(1).filter((i) => i > pivot);
    return [...quicksort(less), pivot, ...quicksort(greater)];
  }
}
```

- Go back

Merge sort

Merge Sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by dividing the unsorted array into 'n' sub-arrays, each containing one element. It then repeatedly merges these sub-arrays to produce new sorted sub-arrays until there is only one sub-array remaining – the fully sorted array.

How Merge Sort Works:

1. **Divide:**
   - The unsorted array is recursively divided into two halves until each sub-array contains only one element. This is the base case of the recursion.
2. **Conquer:**
   - The adjacent sub-arrays are then recursively merged to produce new sorted sub-arrays. This process continues until there is only one sub-array remaining – the fully sorted array.
3. **Merge:**
   - The key operation in Merge Sort is the merging of two sorted sub-arrays to produce a single, sorted sub-array. This involves comparing elements from the two sub-arrays and placing them in the correct order.

Time Complexity: - Merge Sort has a consistent time complexity of O(n log n) in all cases, where 'n' is the number of elements in the array. It is a stable sorting algorithm, meaning that equal elements maintain their relative order in the sorted output. While Merge Sort has a slightly higher space complexity due to the need for additional memory, its stability and predictable performance make it a widely used and reliable sorting algorithm.

```
class Solution {

    void merge(int[] arr, int low, int mid, int high) {
        int subArr1Size = mid - low + 1;
        int subArr2Size = high - mid;

        int [] subArr1 = new int[subArr1Size];
        int [] subArr2 = new int[subArr2Size];

        for (int i = 0; i < subArr1Size; i++) {
           subArr1[i] = arr[low + i];
         }
         for (int i = 0; i < subArr2Size; i++) {
           subArr2[i] = arr[mid + 1 + i];
        }
        int i = 0, j = 0, k = low;

        while(i < subArr1Size && j < subArr2Size) {
            if(subArr1[i] <= subArr2[j]) {
                arr[k] = subArr1[i];
                i++;
            } else {
                arr[k] = subArr2[j];
                j++;
            }
```

```java
            k++;
        }
        while(i < subArr1Size) {
            arr[k++] = subArr1[i++];
        }
        while (j < subArr2Size) {
           arr[k++] = subArr2[j++];
       }
    }

    void mergesort(int[] arr, int low, int high){
        if(high > low) {
            int mid = (high + low) / 2;
            mergesort(arr, low, mid);
            mergesort(arr, mid + 1, high);
            merge(arr, low, mid, high);
        }
    }

    void mergeSort (int[] arr) {
        int n = arr.length;
        mergesort(arr, 0, n - 1);
    }
}
```

```typescript
function mergeSort(arr: number[]): number[] {
  if (arr.length <= 1) {
    return arr;
  }

  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}

function merge(left: number[], right: number[]): number[] {
  let result: number[] = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }

  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
```

```
}
```

Linear search

Linear Search, also known as sequential search, is a simple searching algorithm that finds the position of a target value within a list or array. It works by iterating through the elements one by one until the target value is found or the entire list has been searched.

How Linear Search Works:

1. **Start at the Beginning:**
   - Linear Search begins by looking at the first element in the list.
2. **Compare with Target:**
   - It compares the current element with the target value that we are searching for.
3. **Search Iteratively:**
   - If the current element is equal to the target value, the search is successful, and the index or position of the element is returned.
   - If the current element is not equal to the target value, the search continues by moving to the next element in the list.
   - This process is repeated until either the target value is found or the end of the list is reached.

Time Complexity:

The time complexity of Linear Search is O(n), where 'n' is the number of elements in the array. In the worst case, the algorithm may need to iterate through the entire list to find the target value. While Linear Search is simple, it may not be the most efficient for large datasets, especially when compared to more advanced search algorithms like binary search on sorted lists. However, it is easy to understand and implement.

```
function linearSearch(arr: number[], target: number): number {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }

  return -1;
}
```

Interval search

An interval search algorithm typically refers to searching for overlapping or containing intervals in a collection of intervals. One common approach for this task is to use an interval tree. Here's an explanation of the Interval Search algorithm using an interval tree:

How Interval Search Works:

1. **Construct the Interval Tree:**
   - Begin by constructing an interval tree from the given set of intervals.
   - Each node in the interval tree represents an interval, and the tree is recursively built to efficiently organize and store these intervals.
2. **Search for Overlapping Intervals:**
   - When searching for intervals that overlap with a given interval (query interval), start at the root of the interval tree.
3. **Traverse the Tree:**
   - Traverse the tree, comparing the query interval with the intervals represented by each node.
   - If there is an overlap, the algorithm can either return the overlapping interval(s) immediately or continue searching in both left and right subtrees.
4. **Recursive Search:**
   - Recursively search in the left or right subtree based on the relationship between the query interval and the intervals represented by the current node.
   - Continue this process until all potential overlapping intervals are found.

Time Complexity: - The time complexity of searching for overlapping intervals using an interval tree is typically O(log n + k), where 'n' is the number of intervals in the tree and 'k' is the number of intervals overlapping with the query interval. The construction of the interval tree initially takes O(n log n) time, but subsequent searches are more efficient. Interval trees are particularly useful when there are many queries for overlapping intervals in a set.

```
type Interval = [number, number];

function intervalSearch(intervals: Interval[], queryInterval: Interval): number[] {
  const result: number[] = [];

  for (let i = 0; i < intervals.length; i++) {
    const [start, end] = intervals[i];
    const [queryStart, queryEnd] = queryInterval;

    if (start <= queryEnd && end >= queryStart) {
      result.push(i);
    }
  }

  return result;
}
```

Binary search

Steps:

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then returns -1;

Time Complexity:

- Worst case: O(log n)
- Average case: O(log n)
- Best case: O(1)

```
function binarySearch(nums: number[], target: number): number {
  let left: number = 0;
  let right: number = nums.length - 1;

  while (left <= right) {
    const mid: number = Math.floor((left + right) / 2);

    if (nums[mid] === target) return mid;
    if (target < nums[mid]) right = mid - 1;
    else left = mid + 1;
  }

  return -1;
}
```

```
class Solution {
    private static int binarySearch(int[] array, int target) {

        int low = 0;
        int high = array.length - 1;

        while(low <= high) {
            int middle = low + (high - low) / 2;
            int value = array[middle];
```

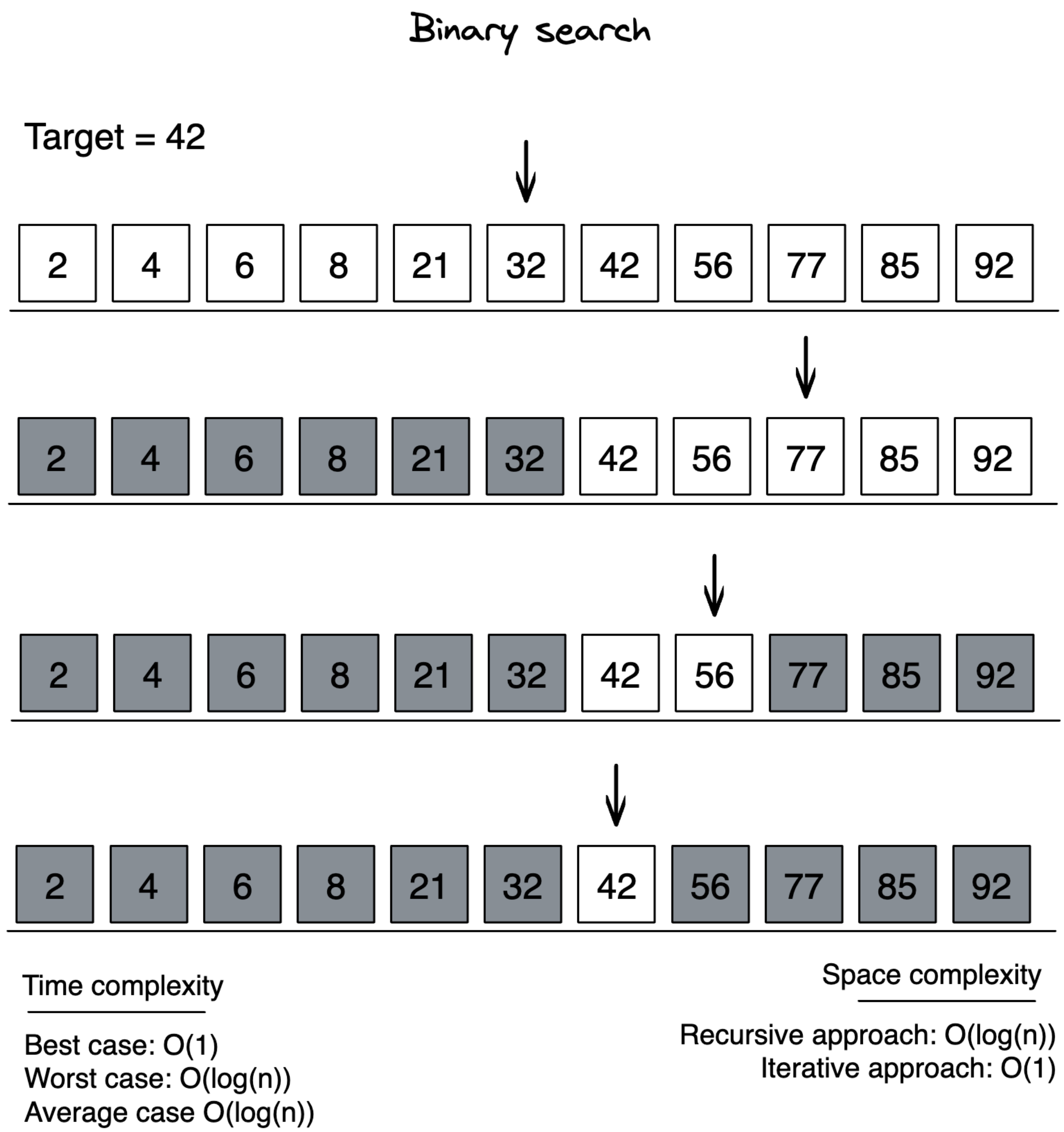

Figure 2: Binary search

```
            if(value < target) {
                low = middle + 1;
            } else if(value > target) {
                high = middle - 1;

            } else {
                return middle;
            }
        }
        return -1;
    }
}
```

```
def binary_search(list, item):
    low = 0
    high = len(list) - 1
    while low <= high:
        mid = (low+high)/2
        guess = list[mid]
        if guess == item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid +1
    return None

my_list = [1, 3, 5, 7, 9]

res = binary_search(my_list, 3)

print(my_list[res])
```

Diffie hellman algorithm

```
function power(a: any, b: any, p: any) {
    if(b === 1) {
        return 1
    } else {
        return Math.pow(a,b) % p
    }
}


function DiffieHellman() {

    let P, G, x, a, y, b, ka, kb;

    P = 23

    console.log("The value of P :", P);

    G = 9;

    console.log("The value of G :", G);

    a = 4;

    console.log("The private key a for Alice : ", a);

    x = power(G,a,P);

    b = 3;

    console.log("The private key a for Bob : ", b);

    y = power(G,b,P);


    ka = power(y, a, P);
    kb = power(x, b, P);

    console.log("Secret key for the Alice is : ", ka);
    console.log("Secret key for the Bob is : ", kb);
}


DiffieHellman()
```

Ternary search

Ternary Search is a divide-and-conquer algorithm designed for efficiently finding the position of a target value in a sorted array. It operates by dividing the array into three parts and recursively narrowing down the search space until the target is found or determined to be absent.

How Ternary Search Works:

1. **Divide the Array:**
   - Ternary Search starts by dividing the sorted array into three parts.
2. **Compare with the Target:**
   - It then compares the target value with the elements at two points within the array, dividing it into three segments.
   - If the target is found at one of these points, the search is successful.
3. **Determine Search Space:**
   - Based on the comparisons, Ternary Search identifies whether the target lies in the first, second, or third segment of the array.
4. **Recursive Search:**
   - The algorithm then recursively applies the same process to the identified segment.
   - This recursion continues until the target is found or the search space is reduced to an empty array, indicating that the target is not present.

Time Complexity: - Ternary Search has a time complexity of O(log3 n), where 'n' is the size of the array. This is an improvement over binary search when the search space can be significantly reduced at each step. However, it's worth noting that constant factors play a role, and in practice, binary search might be faster for smaller datasets due to simpler arithmetic operations. Ternary Search is particularly beneficial when the dataset is large and the search space can be significantly reduced with each iteration.

```
function ternarySearch(func: (x: number) => number, left: number, right: number, epsilon: number): numbe
  while (right - left > epsilon) {
    const mid1 = left + (right - left) / 3;
    const mid2 = right - (right - left) / 3;

    const value1 = func(mid1);
    const value2 = func(mid2);

    if (value1 < value2) {
      left = mid1;
    } else {
      right = mid2;
    }
  }

  return (left + right) / 2;
}
```

Interpolation search

Interpolation Search is a an algorithm designed for finding a specific target value in a sorted array. Unlike linear or binary search, this algorithm utilizes the characteristics of the data distribution to make more informed decisions about where to look for the target. It is particularly effective when the data has a uniform distribution.

How Interpolation Search Works:

1. **Linear Interpolation:**
   - Interpolation Search utilizes linear interpolation to estimate the likely position of the target value in the array.
2. **Estimate Position:**
   - Instead of evenly dividing the search space, as in binary search, Interpolation Search estimates the probable position of the target based on its value relative to the minimum and maximum values in the array.
3. **Calculation of Position:**
   - It calculates an estimate of the target's position by considering the relative location of the target with respect to the minimum and maximum values in the current search space.
4. **Refine Search:**
   - Based on the calculated estimate, the algorithm narrows down the search space and repeats the process until the target is found or the search space is exhausted.

Time Complexity:

The time complexity of Interpolation Search is O(log log n) on average, where "n" is the number of elements in the array. In the best case, it can be O(1), and in the worst case, it can be O(n). However, the average case is often more relevant, and it is O(log log n) under certain assumptions about the distribution of the data.

```
class Solution {

    private static int interpolationSearch(int[] array, int value) {
        int low = 0;
        int high = array.length - 1;

        while(value >=array[low] && value <= array[high] && low <= high) {
            int probe = low + (high - low) * (value - array[low]) / (array[high] - array[low]);
            if(array[probe] == value) {
                return probe;
            } else if(array[probe] > value) {
                low = probe + 1;
            } else {
                high = probe -1;
            }

        }
```

```
        return -1;
    }

}
```

```
function interpolationSearch(array: number[], value: number): number {
  let low = 0;
  let high = array.length - 1;

  while (value >= array[low] && value <= array[high] && low <= high) {
    const probe =
      low + ((high - low) * (value - array[low])) / (array[high] - array[low]);
    const roundedProbe = Math.floor(probe);

    if (array[roundedProbe] === value) {
      return roundedProbe;
    } else if (array[roundedProbe] < value) {
      low = roundedProbe + 1;
    } else {
      high = roundedProbe - 1;
    }
  }

  return -1;
}
```

Breadth-first search

```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }

  bfs(startingVertex: string) {
    const visited: Set<string> = new Set();
    const queue: string[] = [];

    visited.add(startingVertex);
    queue.push(startingVertex);

    while (queue.length > 0) {
      const currentVertex = queue.shift()!;
      console.log(currentVertex);

      const neighbors = this.adjacencyList.get(currentVertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");
```

```
graph.bfs("A");
```

Depth-first search

```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }

  dfs(startingVertex: string) {
    const visited: Set<string> = new Set();

    const dfsHelper = (vertex: string) => {
      console.log(vertex);
      visited.add(vertex);

      const neighbors = this.adjacencyList.get(vertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          dfsHelper(neighbor);
        }
      }
    };

    dfsHelper(startingVertex);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.dfs("A");
```

Dijkstra’s algorithm

```
class Graph {
  private adjacencyList: Map<string, Map<string, number>>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, new Map());
    }
  }

  addEdge(vertex1: string, vertex2: string, weight: number) {
    this.adjacencyList.get(vertex1)?.set(vertex2, weight);
    this.adjacencyList.get(vertex2)?.set(vertex1, weight);
  }

  dijkstra(startingVertex: string) {
    const distances: Map<string, number> = new Map();
    const previous: Map<string, string | null> = new Map();
    const priorityQueue = new PriorityQueue();

    for (const vertex of this.adjacencyList.keys()) {
      distances.set(vertex, vertex === startingVertex ? 0 : Infinity);
      previous.set(vertex, null);
      priorityQueue.enqueue(vertex, distances.get(vertex)!);
    }

    while (!priorityQueue.isEmpty()) {
      const currentVertex = priorityQueue.dequeue()!;
      const neighbors = this.adjacencyList.get(currentVertex);

      if (neighbors) {
        for (const neighbor of neighbors.keys()) {
          const distance = distances.get(currentVertex)! + neighbors.get(neighbor)!;

          if (distance < distances.get(neighbor)!) {
            distances.set(neighbor, distance);
            previous.set(neighbor, currentVertex);
            priorityQueue.enqueue(neighbor, distance);
          }
        }
      }
    }

    return { distances, previous };
  }

  shortestPath(startingVertex: string, targetVertex: string) {
    const { distances, previous } = this.dijkstra(startingVertex);
```

```typescript
    const path: string[] = [];
    let currentVertex = targetVertex;

    while (currentVertex !== null) {
      path.unshift(currentVertex);
      currentVertex = previous.get(currentVertex)!;
    }

    return { path, distance: distances.get(targetVertex) };
  }
}

class PriorityQueue {
  private items: [string, number][] = [];

  enqueue(element: string, priority: number) {
    this.items.push([element, priority]);
    this.sort();
  }

  dequeue() {
    return this.items.shift();
  }

  isEmpty() {
    return this.items.length === 0;
  }

  private sort() {
    this.items.sort((a, b) => a[1] - b[1]);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B", 1);
graph.addEdge("A", "C", 4);
graph.addEdge("B", "C", 2);
graph.addEdge("B", "D", 5);
graph.addEdge("C", "D", 1);

const { path, distance } = graph.shortestPath("A", "D");
console.log("Shortest Path:", path); // Output: Shortest Path: [ 'A', 'B', 'C', 'D' ]
console.log("Distance:", distance); // Output: Distance: 4
```

Floyd-Warshall algorithm

```
class Graph {
  private adjacencyMatrix: number[][];

  constructor(numVertices: number) {
    this.adjacencyMatrix = Array.from({ length: numVertices }, () =>
      Array(numVertices).fill(Infinity)
    );

    // Set diagonal elements to 0
    for (let i = 0; i < numVertices; i++) {
      this.adjacencyMatrix[i][i] = 0;
    }
  }

  addEdge(source: number, destination: number, weight: number) {
    this.adjacencyMatrix[source][destination] = weight;
  }

  floydWarshall() {
    const numVertices = this.adjacencyMatrix.length;

    for (let k = 0; k < numVertices; k++) {
      for (let i = 0; i < numVertices; i++) {
        for (let j = 0; j < numVertices; j++) {
          if (
            this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j] <
            this.adjacencyMatrix[i][j]
          ) {
            this.adjacencyMatrix[i][j] =
              this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j];
          }
        }
      }
    }

    return this.adjacencyMatrix;
  }
}

// Example usage:
const graph = new Graph(4);

graph.addEdge(0, 1, 3);
graph.addEdge(0, 2, 6);
graph.addEdge(1, 2, 1);
graph.addEdge(1, 3, 4);
graph.addEdge(2, 3, 2);

const result = graph.floydWarshall();

console.log("Shortest Path Matrix:");
```

```
for (const row of result) {
  console.log(row);
}
```

Ford Fulkerson algorithm

```
class FordFulkerson {
  private graph: number[][];
  private numVertices: number;

  constructor(graph: number[][]) {
    this.graph = graph;
    this.numVertices = graph.length;
  }

  fordFulkerson(source: number, sink: number): number {
    let maxFlow = 0;

    // Create a residual graph and initialize it with the original capacities.
    const residualGraph = this.graph.map((row) => [...row]);

    while (true) {
      const path = this.bfs(source, sink, residualGraph);
      if (!path) {
        break; // No augmenting path found, terminate the algorithm
      }

      // Find the minimum capacity along the augmenting path
      let minCapacity = Number.POSITIVE_INFINITY;
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        minCapacity = Math.min(minCapacity, residualGraph[u][v]);
      }

      // Update residual capacities and reverse edges along the path
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        residualGraph[u][v] -= minCapacity;
        residualGraph[v][u] += minCapacity;
      }

      // Add the flow of the augmenting path to the total flow
      maxFlow += minCapacity;
    }

    return maxFlow;
  }

  bfs(source: number, sink: number, graph: number[][]): number[] | null {
    const visited: boolean[] = new Array(this.numVertices).fill(false);
    const queue: number[] = [source];
    const parent: number[] = new Array(this.numVertices).fill(-1);

    while (queue.length > 0) {
      const u = queue.shift()!;
```

```
      for (let v = 0; v < this.numVertices; v++) {
        if (!visited[v] && graph[u][v] > 0) {
          queue.push(v);
          parent[v] = u;
          visited[v] = true;
        }
      }
    }

    if (!visited[sink]) {
      return null; // No augmenting path found
    }

    const path: number[] = [];
    for (let v = sink; v !== source; v = parent[v]) {
      path.unshift(v);
    }
    path.unshift(source);

    return path;
  }
}

// Example usage:
const graph = [
  [0, 16, 13, 0, 0, 0],
  [0, 0, 10, 12, 0, 0],
  [0, 4, 0, 0, 14, 0],
  [0, 0, 9, 0, 0, 20],
  [0, 0, 0, 7, 0, 4],
  [0, 0, 0, 0, 0, 0],
];

const fordFulkerson = new FordFulkerson(graph);
const maxFlow = fordFulkerson.fordFulkerson(0, 5);
console.log("Maximum Flow:", maxFlow);
```