CS 137

Programming Principles

Chapter 12

Searching and Sorting Algorithms

Victoria Sakhnini

Table of Contents

Searching Algorithms

> You are not expected to memorize these algorithms; however, you should understand their behaviour.

In this chapter we will learn some searching and sorting algorithms and use Big-Oh notation to analyse their efficiency and compare results.

Linear Search

The main idea: search from the start of the array until we find what we are looking for, or reach the end.

Code

```
#include <stdio.h>
#include <assert.h>
// Post: Returns the index of the first occurrence of value in a,
//       or -1 if not found.
int lin_search(int a[], int n, int value) {
    for (int i = 0; i < n; i++) {
        if (a[i] == value)  // found it
            return i;       // return index of first occurrence
    }
    return -1;  // value not found
}
int main(void) {
    int a[] = { 19, 4, 2, 4, 6 };
    const int len = sizeof(a) / sizeof(a[0]);
    assert(0  == lin_search(a, len, 19));
    assert(1  == lin_search(a, len, 4));
    assert(4  == lin_search(a, len, 6));
    assert(-1 == lin_search(a, len, 14));
    return 0;
}
```

Time Complexity Analysis

There are three key lines of code: the for loop, the if statement, and the return statement(s). We assume basic operations (arithmetic, comparisons, return statements) run in O(1) constant time.

- **Best case:** The value is found immediately at `a[0]`. All steps are constant time: **O(1)**.
- **Worst case:** The element is not in the array. We execute approximately (1 + 2 + 1 + 2) steps for each of the n elements, plus a final return. Total: **O(n)**.
- **Average case:** Assuming the value is equally likely to be at any position, we need roughly n/2 comparisons on average. This still gives **O(n)**.

> The average/typical case is often hard to reason about precisely. The worst case is a safer and more universal metric and is usually sufficient for our analysis.

Binary Search

If the array is already sorted, linear search wastes information. Binary search exploits the sorted order.

Basic idea:

- Check the middle element `a[m]`.
- If `a[m] == value`, return `m`.
- If `a[m] > value`, search the lower half.
- If `a[m] < value`, search the upper half.
- Stop when `lo > hi` (search space exhausted).

Code

```
#include <stdio.h>
#include <assert.h>

// Post: Returns the index of value in a, or -1 if not found.
int bin_search(int a[], int n, int value) {
    int lo = 0, hi = n - 1;
    while (hi >= lo) {
        // Note: (hi + lo) / 2 is equivalent but may overflow.
        int m = lo + (hi - lo) / 2;
        if (a[m] == value) return m;
        if (a[m] < value)  lo = m + 1;
        if (a[m] > value)  hi = m - 1;
    }
    return -1;  // value not found
}

int main(void)
{
    int a[] = { -10, -7, 0, 2, 11, 14, 38, 42 };
    const int len = sizeof(a) / sizeof(a[0]);
    assert(0  == bin_search(a, len, -10));
    assert(3  == bin_search(a, len, 2));
    assert(7  == bin_search(a, len, 42));
    assert(-1 == bin_search(a, len, 114));
    return 0;
}
```


A trace for the first `assert` (line 29):

```
len=n= 8  value= -10  hi= 7  lo= 0
m=3  hi=2  lo=0
m=1  hi=0  lo=0
m=0  hi=0  lo=0  a[0]==-10
```

A trace for the second `assert` (line 30):

```
len=n= 8  value= 2  hi= 7  lo= 0
m=3  hi=7  lo=0  a[3]==2
```

A trace for the third `assert` (line 31):

```
len=n= 8  value= 42  hi= 7  lo= 0
m=3  hi=7  lo=4
m=5  hi=7  lo=6
m=6  hi=7  lo=7
m=7  hi=7  lo=7  a[7]==42
```

A trace for the fourth `assert` (line 32):

```
len=n= 8  value= 114  hi= 7  lo= 0
m=3  hi=7  lo=4
m=5  hi=7  lo=6
m=6  hi=7  lo=7
m=7  hi=7  lo=8
returning -1
```

Time Complexity Analysis

- **Worst / Average case:** At each step, the search space is halved. We can halve n at most **O(log n)** times, and each step is O(1). Total: **O(log n)**.
- **Best case:** The element is in the middle on the first probe: **O(1)**.

Binary search is extremely fast — even with 1 billion entries, at most 30 iterations are needed in the worst case.

Sorting Algorithms

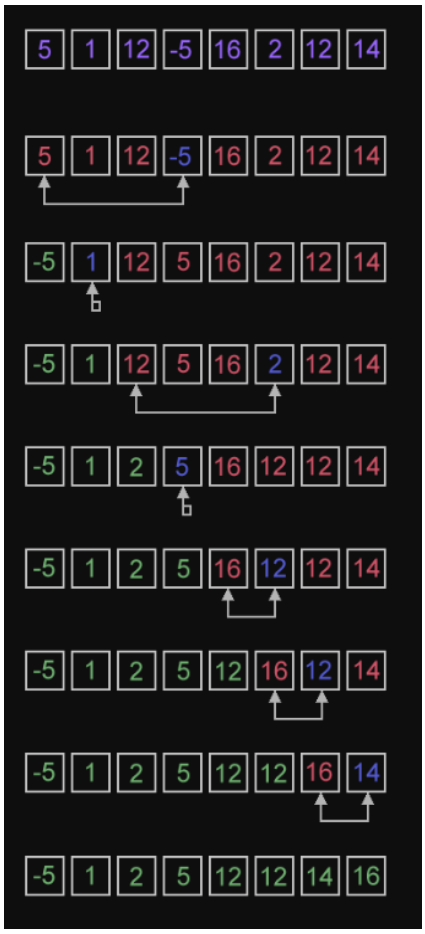
Introduction

The main task: take an unsorted array and sort it. We will discuss Selection Sort, Insertion Sort, Merge Sort, and Quick Sort, and analyse the runtime of each.

Selection Sort

The idea:

- Find the smallest element.
- Swap it with the first element.
- Repeat with the remaining unsorted portion of the array.



source: http://www.algolist.net/img/sorts/selection-sort-1.png

Code

```c
#include <stdio.h>

void selection_sort(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Find the index of the minimum value in a[i..n-1]
        int min = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[min]) min = j;
        // Swap a[i] with a[min]
        int temp = a[min];
        a[min]   = a[i];
        a[i]     = temp;
    }
}
int main(void) {
    int a[] = { 20, 12, 10, 15, 2 };
    const int n = sizeof(a) / sizeof(a[0]);
    selection_sort(a, n);
    for (int i = 0; i < n - 1; i++)
        printf("%d, ", a[i]);
    printf("%d\n", a[n - 1]);
    // Output: 2, 10, 12, 15, 20
    return 0;
}
```

Time Complexity Analysis

All operations are O(1). The outer loop runs n−1 times. For each i, the inner loop runs n−1−i times. The total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} C &= C\sum_{i=0}^{n-2}(n-i-1) = C\sum_{i=0}^{n-2} n - C\sum_{i=0}^{n-2} i - C\sum_{i=0}^{n-2} 1 \\ &= Cn(n-1) - C\frac{(n-1)(n-2)}{2} - C(n-1) \\ &= C(n-1)(n-(n-2)/2) - C(n-1) \\ &= C(n-1)(n/2+1-1) \\ &= O(n^2)\end{aligned}$$

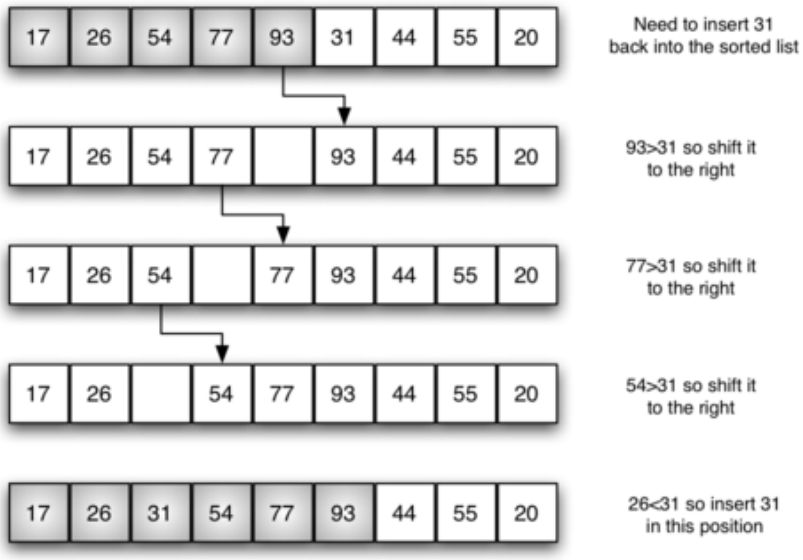
⚠ Significant flaw: even when the array is already sorted, selection sort still performs $O(n^2)$ comparisons. It does not benefit from partial sortedness.

? Can we improve the algorithm in the best case so that a sorted (or nearly sorted) array is handled efficiently? Yes — see Insertion Sort below.

Insertion Sort

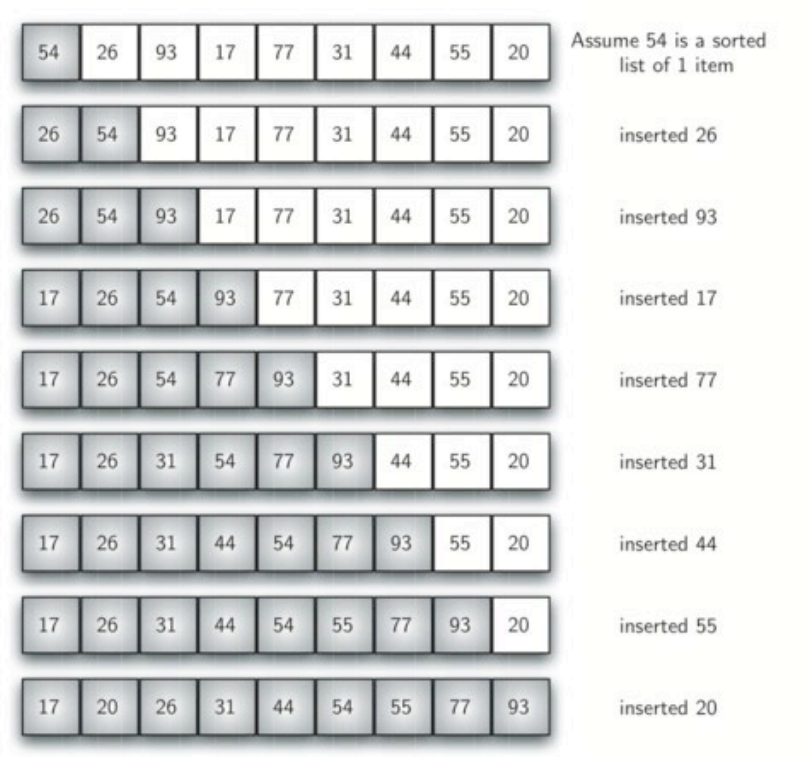
The idea: For each element x in the array, find where x belongs in the already-sorted portion to its left, shift elements greater than x rightward to make room, and insert x.

- For each element x...
- Find where x should go.
- Shift elements greater than x one position right.
- Insert x in its correct position.
- Repeat for the next element.



source: https://runestone.academy/runestone/books/published/pythonds/_images/insertionpass.png

Example Full:



source: https://runestone.academy/runestone/books/published/pythonds/_images/insertionsort.png

Code

```
#include <stdio.h>

void insertion_sort(int *a, int n) {
    int i, j, x;
    for (i = 1; i < n; i++) {
        x = a[i];
        // Shift elements greater than x one position to the right
        for (j = i; j > 0 && x < a[j - 1]; j--)
            a[j] = a[j - 1];
        a[j] = x;  // insert x in its correct position
    }
}

int main(void) {
    int a[] = { -10, 2, 14, -7, 11, 38 };
    const int n = sizeof(a) / sizeof(a[0]);
    insertion_sort(a, n);
    for (int i = 0; i < n - 1; i++)
        printf("%d, ", a[i]);
    printf("%d\n", a[n - 1]);
    // Output: -10, -7, 2, 11, 14, 38
    return 0;
}
```

Time Complexity Analysis

- **Worst case** (reverse-sorted array): The inner loop runs i times for each i from 1 to n−1.

```
∑ (from i=1 to n-1) of i  =  1 + 2 + ... + (n-1)  =  n(n-1)/2  =  O(n²)
```

- **Best case** (already sorted): The inner loop never executes. Only the outer loop runs: **O(n)**.
- **Average case**: The inner loop runs about i/2 times on average.

```
∑ (from i=1 to n-1) of i/2  =  n(n-1)/4  =  O(n²)
```

Insertion Sort vs Selection Sort

Algorithm	Best Case	Average Case	Worst Case	Notes
Insertion Sort	O(n)	O(n²)	O(n²)	Great for nearly-sorted arrays
Selection Sort	O(n²)	O(n²)	O(n²)	No benefit from partial order

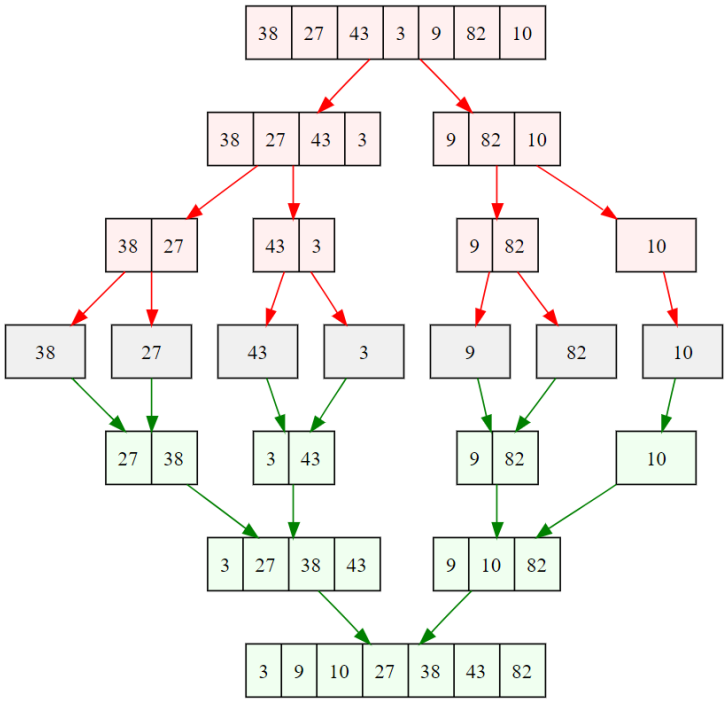
❓ Can we do better than O(n²) in the worst case?

Yes! Merge Sort and Quick Sort both achieve O(n log n) in most cases.

Merge Sort

Basic idea (divide and conquer):

- Divide the array in half.
- Sort each half recursively using the same algorithm.
- Merge the two sorted halves back together.



source: https://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void merge_sort(int a[], int t[], int n)
{
    if (n <= 1) return;  // base case

    // Step 1: divide
    int middle = (n + 1) / 2;
    int *lower = a;
    int *upper = a + middle;

    // Step 2: recursively sort each half
    merge_sort(lower, t, middle);
    merge_sort(upper, t, n - middle);
```

```
    // Step 3: merge
    int i = 0;       // index into lower half
    int j = middle;  // index into upper half
    int k = 0;       // index into temp array
    while (i < middle && j < n)
    {
        if (a[i] <= a[j])  t[k++] = a[i++];
        else               t[k++] = a[j++];
    }
    while (i < middle) t[k++] = a[i++];
    while (j < n)      t[k++] = a[j++];
    for (i = 0; i < n; i++) a[i] = t[i];
}

void sort(int a[], int n)
{
    int *t = malloc(n * sizeof(a[0]));
    assert(t);
    merge_sort(a, t, n);
    free(t);
}

int main(void)
{
    int a[] = { 38, 27, 43, 3, 9, 82, 10 };
    const int n = sizeof(a) / sizeof(a[0]);
    sort(a, n);
    for (int i = 0; i < n - 1; i++) printf("%d, ", a[i]);
    printf("%d\n", a[n - 1]);
    // Output: 3, 9, 10, 27, 38, 43, 82
    return 0;
}
```

📌 The sort() wrapper creates a single temporary array t of size n to store merged results before copying them back to a[]. This requires O(n) extra space.

Time Complexity Analysis

At each level of recursion:

- Two recursive calls: O(1) overhead.
- Merging left and right halves into `t[]`: O(k), where k is the size of the current subarray.
- Copying `t[]` back into `a[]`: O(k).
- Each instance at a given level costs O(k).

At level v of the recursion tree, each subarray has size $k = n / 2^v$ and there are $2^v = n/k$ such subarrays. Total work per level = $O(n/k \times k) = O(n)$.

The number of levels m satisfies $k = n / 2^m = 1$, giving $m = \log_2(n)$.

```
Total time  =  O(n) × log₂(n) levels  =  O(n log n)
```

💡 This O(n log n) bound holds for the best, average, and worst cases — merge sort always splits evenly.

Quick Sort

Quick Sort was invented by Tony Hoare.

Basic idea:

- Pick a **pivot element** p in the array.
- Partition the array so that all elements less than p are to its left and all elements ≥ p are to its right.
- Recursively sort the two partitions.
- **Key benefit:** No temporary array needed.

Lomuto Partition Scheme

There are many partitioning strategies (Lomuto, Hoare, median-of-medians). We use the **Lomuto partition** here:

- Swaps left to right.
- Pivot is the first element.
- `m` — last index in the "less than p" partition.
- `i` — first index of the unpartitioned portion.

Two cases:

- If `a[i] < p`: increment m, swap `a[m]` with `a[i]`, then increment i.
- If `a[i] >= p`: just increment i.
- End: swap `a[0]` (pivot) with `a[m]` to place the pivot between the two partitions.

Code

```
#include <stdio.h>

void swap(int *a, int *b)
{
    int t = *a;
    *a    = *b;
    *b    = t;
}

void quick_sort(int a[], int n)
{
    if (n <= 1) return;
    int m = 0;
    for (int i = 1; i < n; i++)
    {
        if (a[i] < a[0])
        {
            m++;
            swap(&a[m], &a[i]);
        }
    }
    swap(&a[0], &a[m]);  // place pivot between the two partitions
    quick_sort(a, m);
    quick_sort(a + m + 1, n - m - 1);
}
```

```
int main(void)
{
    int a[] = { -10, 2, 14, -7, 11, 38 };
    const int n = sizeof(a) / sizeof(a[0]);
    quick_sort(a, n);
    for (int i = 0; i < n - 1; i++)
        printf("%d, ", a[i]);
    printf("%d\n", a[n - 1]);
    // Output: -10, -7, 2, 11, 14, 38
    return 0;
}
```

Execution Trace

```
quick_sort: -10, 2, 14, -7, 11, 38
  pivot = -10
  m=0 i=1 | m=0 i=2 | m=0 i=3 | m=0 i=4 | m=0 i=5
  after final swap: -10, 2, 14, -7, 11, 38
  base case: -10

quick_sort: 2, 14, -7, 11, 38
  pivot = 2
  m=0 i=1 | m=0 i=2 → a[i]=-7 < 2, so swap: 2, -7, 14, 11, 38, m=1
  m=1 i=3 | m=1 i=4
  after final swap: -7, 2, 14, 11, 38
  base case: -7

quick_sort: 14, 11, 38
  pivot = 14
  m=0 i=1 → a[i]=11 < 14, so swap: 14, 11, 38, m=1
  m=1 i=2
  after final swap: 11, 14, 38
  base case: 11
  base case: 38

Final result: -10, -7, 2, 11, 14, 38
```

Time Complexity Analysis

- **Best case** (pivot always splits evenly): O(n log n). Like merge sort — log n levels, O(n) work per level.
- **Average case** (pivot between 25th and 75th percentile): Still O(n log n). Worst-case split is 3:1, giving log_(4/3) n levels, each O(n).
- **Worst case** (already sorted or reverse-sorted): The pivot is always the smallest element, giving partitions of size 1 and n−1. There are now n levels: **$O(n^2)$**.

```
Worst-case sum:  (n-1) + (n-2) + ... + 1  =  n(n-1)/2  =  O(n²)
```

? If quicksort has a worst case of $O(n^2)$, why is it one of the most commonly used sorting algorithms?

In practice, the worst case is rare. Additionally, intelligent pivot selection strategies (e.g., median-of-medians) can guarantee O(n log n) even in the worst case.

Quick sort also has excellent cache performance and no extra memory allocation (unlike merge sort's O(n) temporary array), which makes it fast in practice despite the theoretical worst case.

Built-in Sorting

In practice, people rarely implement their own sorting algorithms. The C standard library `<stdlib.h>` provides `qsort` (note: this is not necessarily quick sort internally).

```
void qsort(void *base, size_t n, size_t size,
           int (*compare)(const void *a, const void *b));
```

- `base` — pointer to the beginning of the array.
- `n` — number of elements.
- `size` — size in bytes of one element.
- `compare` — function pointer to a comparison function. Must return < 0 if a < b, 0 if a == b, > 0 if a > b.

Example

```
#include <stdio.h>
#include <stdlib.h>

struct student {
    unsigned int id;
    double avg;
};

int compare1(const struct student *a, const struct student *b) {
    unsigned int p = a->id;
    unsigned int q = b->id;
    if (p < q)      return -1;
    else if (p == q) return  0;
    else             return  1;
}

int compare2(const int *a, const int *b) {
    int p = *a;
    int q = *b;
    if (p < q)      return -1;
    else if (p == q) return  0;
    else             return  1;
}

int main(void) {
    struct student st[3] = { {333, 86.7}, {111, 90.7}, {222, 80} };
    qsort(st, 3, sizeof(struct student), compare1);
```

```
    for (int i = 0; i < 2; i++)
        printf("%d, ", st[i].id);
    printf("%d\n", st[2].id);
    // Output: 111, 222, 333

    int a[] = { -10, 2, 14, -7, 11, 38 };
    const int n = sizeof(a) / sizeof(a[0]);
    qsort(a, n, sizeof(int), compare2);
    for (int i = 0; i < n - 1; i++)
        printf("%d, ", a[i]);
    printf("%d\n", a[n - 1]);
    // Output: -10, -7, 2, 11, 14, 38
    return 0;
}
```

Summary

Selection Sort

- Find the smallest element and swap it with the first unsorted element.
- Best, average, and worst case: **$O(n^2)$**.

Insertion Sort

- Find where element i belongs and shift elements to make room.
- Best case: **O(n)**; average and worst case: **$O(n^2)$**.
- Works well for nearly-sorted arrays.

Merge Sort

- Divide and conquer: split into halves, recurse, then merge.
- Best, average, and worst case: **O(n log n)**, but requires **O(n) extra space**.

Quick Sort

- Pick pivot, partition into smaller and larger elements, recurse.
- Best and average case: **O(n log n)**; worst case: $O(n^2)$ (see discussion).
- Does not require a temporary array.

Linear Search

- Scan one by one until the value is found.
- Best case: **O(1)**; average and worst case: **O(n)**.

Binary Search

- Probe the middle element and repeat (requires a sorted array).
- Best case: **O(1)**; average and worst case: **O(log n)**.

Table Summary of Big-Oh Notations

Algorithm	Best Case (Ω)	Average Case (Θ)	Worst Case (O)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort *	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n log n) **
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

** Merge Sort requires O(n) extra space.*

*** See the worst-case discussion in the Quick Sort section.*

Extra Practice Problems

1.

Write a function that takes an array of integers and its length, and returns true if all numbers are unique (no repetitions), false otherwise.

2.

Consider selection sort on the array A: 8, 6, 7, 5, 3, 0, 9.
What is the value of A after the first iteration of the sorting algorithm?

After the first iteration: 0, 6, 7, 5, 3, 8, 9

3.

Consider insertion sort on the array A: 8, 6, 7, 5, 3, 0, 9.
What is the value of A after the first iteration of the sorting algorithm?

After the first iteration: 6, 8, 7, 5, 3, 0, 9

4.

Employees submit meeting bookings with start and end times. Merge all overlapping bookings and return a list of non-overlapping intervals covering all booked periods.

Example 1:
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: [1,3] and [2,6] overlap and merge into [1,6].

Example 2:
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: [1,4] and [4,5] are considered overlapping.

Constraints:
$1 \le bookings.length \le 10^4$
bookings[i].length == 2
$0 \le start_i \le end_i \le 10^4$

5.

Given a list of item codes representing orders, sort items by frequency (most ordered first). Break ties by decreasing item code.

Example 1:
Input: [1,1,2,2,2,3]
Output: [2,2,2,1,1,3]

Example 2:
Input: [2,3,1,3,2]
Output: [3,3,2,2,1]
Explanation: Items 2 and 3 both appear twice; item 3 has higher code so comes first.

Example 3:
Input: [-1,1,-6,4,5,-6,1,4,1]
Output: [1,1,1,4,4,-6,-6,5,-1]

Constraints:
1 ≤ orders.length ≤ 100
-100 ≤ orders[i] ≤ 100

6.

Implement binary search recursively.