Sorting and Searching Techniques

Introduction

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

We have:

1. Different Sorting Techniques
2. Different Searching Techniques

Sorting Algorithms

A sorting algorithm is an algorithm that puts elements of a list in a certain order.

The most used orders are numerical order.

Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly

List of Various Sorting Algorithms

1. Bubble Sort
2. Selection Sort
3. Merge Sort
4. Quick Sort
5. Insertion Sort
6. Shell Sort

Why Study Sorting?

When an input is sorted, many problems become easy (e.g. searching, min, max, k-th smallest).

Sorting has a variety of interesting algorithmic solutions that embody many ideas

- Comparison vs non-comparison based
- Iterative
- Recursive
- Divide-and-conquer
- Best/worst/average-case bounds
- Randomized algorithms

Applications of Sorting

1. Uniqueness testing
2. Deleting duplicates
3. Prioritizing events
4. Frequency counting
5. Reconstructing the original order
6. Set intersection/union
7. Finding a target pair x, y such that x+y = z
8. Efficient searching

Bubble Sort

It is also known as ***exchange sort.***

It is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted.

The algorithm gets its name from the way smaller elements "bubble" to the top (i.e., the beginning) of the list via the swaps.

Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

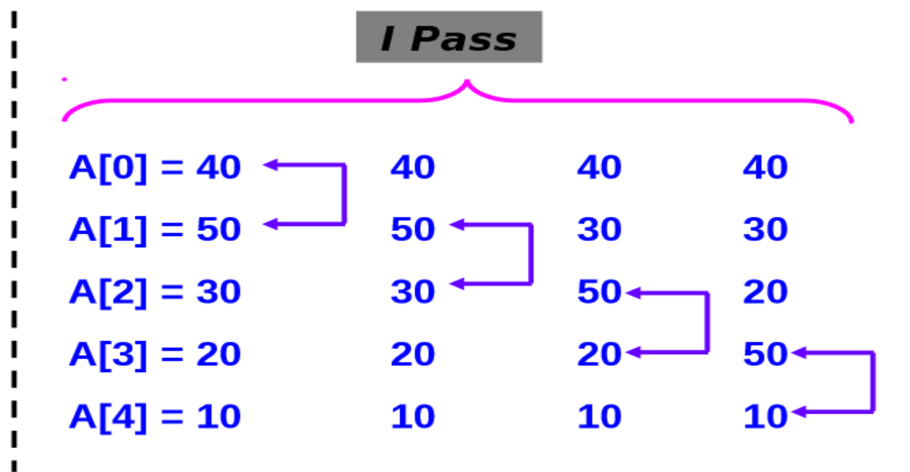
Bubble Sort: Implementation

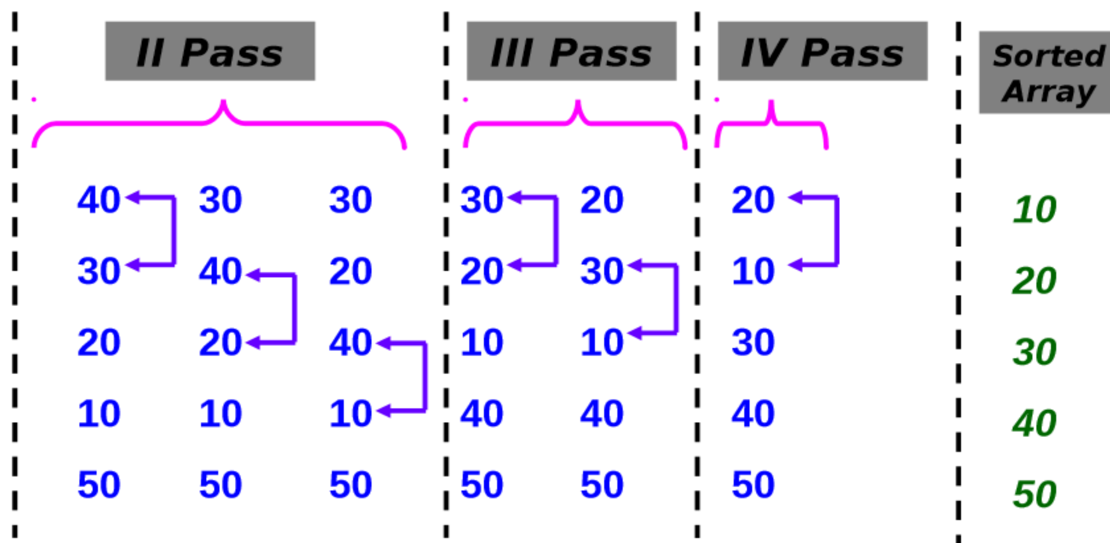
```
void bubbleSort(int a[], int n) {
  for (int i = n-1; i >= 1; i--) {
    for (int j = 1; j <= i; j++) {
      if (a[j-1] > a[j])
        swap(a[j], a[j-1]);
    }
  }
}
```

Step 1: Compare adjacent pairs of numbers

Step 2: Swap if the items are out of order

Trace of a Bubble Sort





Worst Case Performance

1. Bubble sort has worst-case complexity O(n2) on lists of size n.
2. Note that each element is moved no more than one step each time.
3. No element can be more than a distance of n - 1 away from its final sorted position, so we use at most n - 1 = O(n) operations to move an element to its final sorted position, and use no more than (n - 1)2 = O(n2) operations in the worst case.

4. On a list where the smallest element is at the bottom, each pass through the list will only move it up by one step, so we will take n - 1 passes to move it to its final sorted position.
5. As each pass traverses the whole list a pass will take n - 1 = O(n) operations. Thus the number of operations in the worst case is also O(n2).

Best Case Performance

1. When a list is already sorted, bubble sort will pass through the list once, and find that it does not need to swap any elements. This means the list is already sorted.
2. Thus bubble sort will take O(n) time when the list is completely sorted.
3. It will also use considerably less time if the elements in the list are not too far from their sorted places.