Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Image Source: https://www.bigocheatsheet.com/

If we look at the worst case upper bound for all sorting algorithms, **Radix Sort** and **Counting Sort** look optimal with **O(nk)** and **O(n+k)**. However, **Radix Sort** and **Counting Sort are not** considered as **comparison-based** sorting algorithm (**compares two number A[i] and A[j]** to get the output sorted order). Among those comparison-based algorithms, Merge Sort, Heap Sort, and Tim Sort are optimal as those algorithms meet the lower bound for sorting.

> *Lower bound for any comparison-based sorting algorithm is* ***O(nlogn)****.*

Obviously, no matter what algorithm it is, if the algorithm uses comparison based technique (A[i] > A[j] or A[i] ≤A[j]) to determine the order of A[i] and A[j] during sorting, that optimal algorithm will always have time complexity of **O(nlogn)**

This is mainly because of 4 reasons:

1. We can view comparison based sorting algorithms as a decision tree. Each node is a yes/no to a question: Does A[i] ≤ A[j] ?

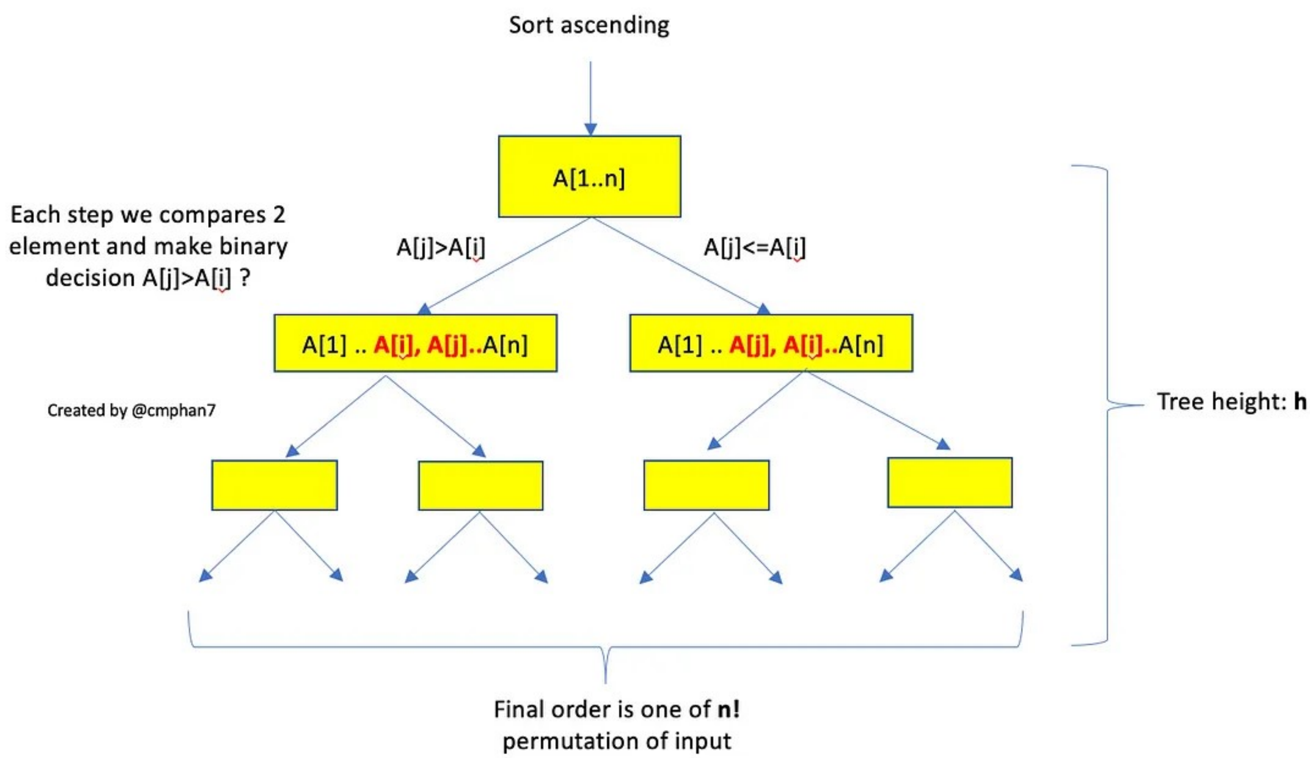


Image Created By Author

2. Given an array A[1..n], the output of the sorting algorithms is one of n! permutation of A[1..n].

3. The height of a tree, which is the longest path from the root to any of its leaves, represents the highest number of comparisons that the sorting algorithms have to perform.

4. We know that for any binary tree of height h, there are no more than 2^h leaves

```
=> n! ≤ 2^h
=> h ≥ log(n!)
log(n!) = log(1) + log(2) + ...+ log(n/2) + ... + log(n) ≥ log(n/2)
+ ... + log(n) ≥ log(n/2) + ... + log(n/2)  ≥ n/2log(n/2) =
n/2log(n) - n/2
```

```
=> h = Ω(nlogn)
```

Bubble Sort

The basic idea is to compares the adjacent elements and swap them if they are in the wrong order.

Animation Source: Wikipedia

```
BubbleSort(array){
  for i -> 0 to arrayLength
     for j -> 0 to (arrayLength - i - 1)
      if arr[j] > arr[j + 1]
        swap(arr[j], arr[j + 1])
}
```

There is a nested loop so bubble sort takes **O(n)** in **time complexity**. The **space complexity** is **O(1)** since we do not create any extra space.

Selection Sort

The key idea is to repeatedly select the next smallest element and move it to the front.

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
```

```
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Example Source: Wikipedia

```
function selectionSort(arr,  n)
{
    var i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(arr,min_idx, i);
    }
}
```

selectionSort hosted with ❤ by **GitHub** **view raw**

Source: geeksforgeeks

Again, we see a nested loop and an in-place swapping in selection sort. Therefore, the **time complexity** is **O(n^2)** and the **space complexity** is **O(1)**

Insertion Sort

The key idea for insertion sort is to sort one element at a time. It's similar to the way you're playing cards. You just try to find a position to insert a card.

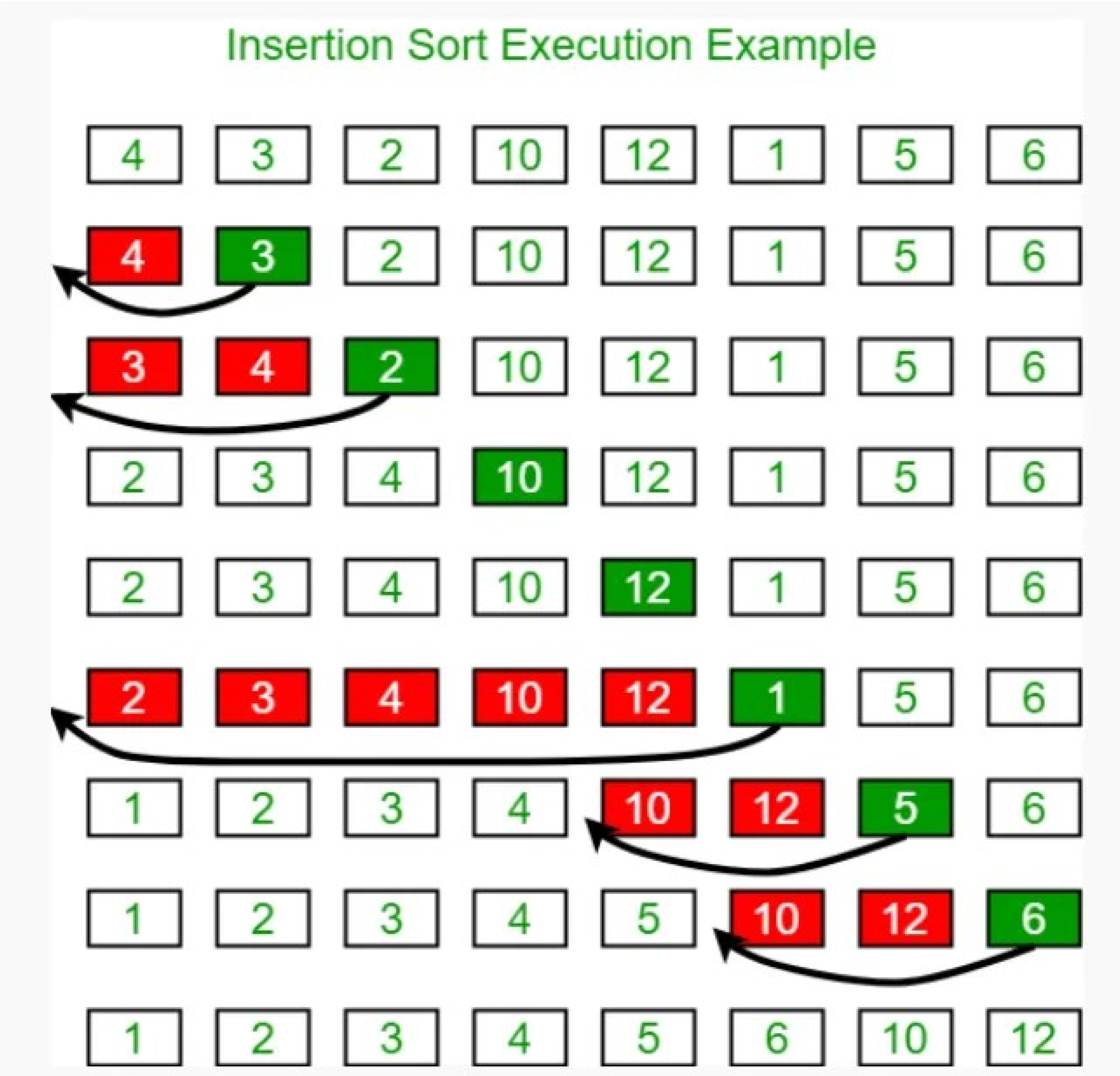


Image Source: geeksforgeeks

```
// Function to sort an array using insertion sort
function insertionSort(arr, n)
{
    let i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

```

insertionSort hosted with ❤ by **GitHub** **view raw**

Insertion sort still requires nested loop so the **time complexity** is **O(n^2)** and **space complexity** is **O(1)** since we do not create any extra space to store the output.

Merge Sort

Merge sort is a **divide-and-conquer** algorithm. **First** the algorithm **divides** the array in halves at each step until it reaches the base case of one element. **Then**, the algorithm **combine** and compare elements at each step to place them in a sorted order.

Animation Source: Wikipedia

```
MergeSort(arr[], l,  r)
If r > l
   middle m = l + (r – l)/2 //Find middle to split array
   mergeSort(arr, l, m) //Recursively sort left half
   mergeSort(arr, m + 1, r) //Recursively sort right half
   merge(arr, l, m, r) //Merge two halves for the final sorted
output
```

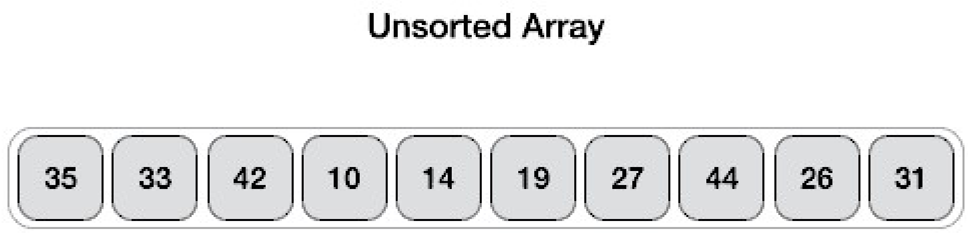
Since we keep diving the input n into 2 equal halves at each step, merge sort can be expressed as the following recurrence relation

```
T(n) = 2T(n/2) + θ(n)
```

The solution to the recurrence relation is **O(nlogn)**, which is the **time complexity** of merge sort. In merge sort, all elements are copied into auxiliary array, so the **space complexity** is **O(n)** for merge sort.

Quick Sort

Similar to Merge Sort, Quick Sort is a **divide-and-conquer** algorithm. However, instead of keep splitting the array in 2 equal halves like merge sort, quick sort pick a last element as a pivot and **partition** the array **around** that **pivot.**



Animation Source: Tutorialspoint

```javascript
// Javascript implementation of QuickSort


// A utility function to swap two elements
function swap(arr, i, j) {
    let temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
function partition(arr, low, high) {

    // pivot
    let pivot = arr[high];

    // Index of smaller element and
    // indicates the right position
    // of pivot found so far
    let i = (low - 1);

    for (let j = low; j <= high - 1; j++) {

        // If current element is smaller
        // than the pivot
        if (arr[j] < pivot) {

            // Increment index of
            // smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}

/* The main function that implements QuickSort
```

```
          arr[] --> Array to be sorted,
          low --> Starting index,
          high --> Ending index
 */
function quickSort(arr, low, high) {
    if (low < high) {

        // pi is partitioning index, arr[p]
        // is now at right place
        let pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

quickSort hosted with ❤ by GitHub view raw

On **average case**, the **time complexity** for quick sort is **O(nlogn).** In the **best case**, quick sort will pick the middle element as pivot, which result in the following recurrence:

$$T(n) = 2T(n/2) + O(n)$$

The recurrence relation is like merge sort, so the time complexity is **O(nlogn)**.

However, in a **worst case**, quick sort will always pick the **smallest/largest** element as pivot, which results in the following recurrence:

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

This recurrence has the solution $O(n^2)$. Therefore, in **worst case**, quick sort takes **$O(n^2)$** in **time complexity**.

Heap Sort

Heap sort is a sorting technique based on Binary Heap data structure. The Binary Heap maintain the order by "heapify" operation.

Example of a Max Heap:

```
       30(0)

       /   \
   70(1)   50(2)
```

*Child (**70(1)**) is greater than the parent (**30(0)**)*

*Swap Child (**70(1)**) with the parent (**30(0)**)*

```
        70(0)
       /   \
 30(1)   50(2)
```

Heap Sort Algorithm for sorting in increasing order:
1. Build a max heap from the input data.
2. Largest item is at the root. Replace it with the last item of the heap. Reduce heap size by 1
3. Heapify the root of the tree.
4. Repeat step 2 while heap size > 1

Illustration:
Input data: *{4, 10, 3, 5, 1}*

```
          4(0)
         /   \
    10(1)   3(2)
       /      \
   5(3)       1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
          4(0)
         /   \
      10(1)  3(2)
        /     \
     5(3)    1(4)
```

Applying heapify procedure to index 0:

```
        10(0)
         /  \
      5(1)  3(2)
```

```
      /    \
    4(3)   1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

The time of heapify is O(logn) and the time to build heap is O(n) so the overall **time complexity** is **O(nlogn).**

Counting Sort

Counting sort sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array. Count sort is a **non-comparative sorting** algorithm.

```
function CountingSort(input, k)

    count ← array of k + 1 zeros
    output ← array of same length as input

    for i = 0 to length(input) - 1 do
        j = key(input[i])
        count[j] += 1

    for i = 1 to k do
        count[i] += count[i - 1]

    for i = length(input) - 1 down to 0 do
        j = key(input[i])
        count[j] -= 1
        output[count[j]] = input[i]

    return output
```

Source: Wikipedia

There are two sequential loop: one from 0 to n-1 and the other one from 1 to k. Therefore, the **time** complexity **O(n+k).** The algorithm uses arrays of length k+ 1 and n so the space complexity is **O(n+k).**

Radix Sort

Radix sort is a **non-comparative sorting** algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using
countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of
elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Tim Sort

Tim sort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. Tim Sort is known to be used in Java.sort() and Python sort() function.

The main idea is that the array are divided into blocks known as **Run.** We sort those runs using insertion sort one by one and then merge those runs using the combine function used in merge sort.

```javascript
// Javascript program to perform TimSort.
let MIN_MERGE = 32;

function minRunLength(n)
{

    // Becomes 1 if any 1 bits are shifted off
    let r = 0;
    while (n >= MIN_MERGE)
    {
        r |= (n & 1);
        n >>= 1;
    }
    return n + r;
}

// This function sorts array from left index to
// to right index which is of size atmost RUN
function insertionSort(arr,left,right)
{
    for(let i = left + 1; i <= right; i++)
    {
        let temp = arr[i];
        let j = i - 1;

        while (j >= left && arr[j] > temp)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}

// Merge function merges the sorted runs
function merge(arr, l, m, r)
{

    // Original array is broken in two parts
    // left and right array
    let len1 = m - l + 1, len2 = r - m;
    let left = new Array(len1);
```

```
    let right = new Array(len2);
    for(let x = 0; x < len1; x++)
    {
```

Source: geeksforgeeeks

Algorithms Interview Cheatsheet JavaScript Programming

Conclusion

Sorting is an essential part of programming. One key takeaway is that any comparison-based algorithm has O(nlogn) time complexity lower bound. Note that Count Sort and Radix Sort are not comparing elements to generate the final output.

I hope you find this article helpful and feel much confident about sorting algorithms by now :)

Follow

Written by Cuong Phan

346 Followers · Writer for InterviewNoodle

Software Engineer | Master CS Student @ UT Austin | https://www.linkedin.com/in/cmphan

More from Cuong Phan and InterviewNoodle