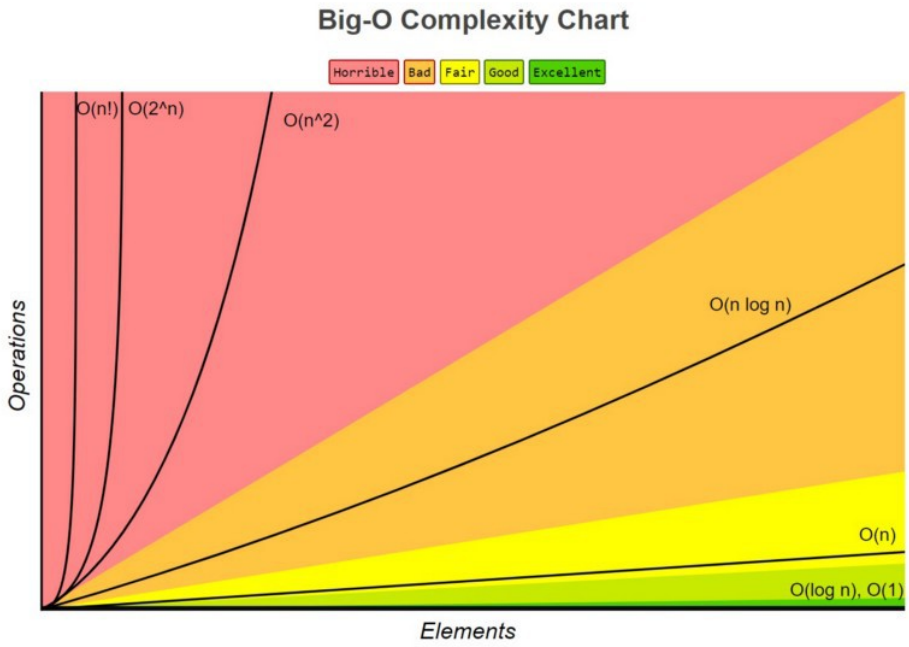
Algorithms

Contents

Time/Space Complexity

How the complexity of an algorithm influences the time or space it takes to complete it's task. Use Big O notation to calculate complexities.

O(n) – where O is the order of magnitude (time / space), and n is the amount of data input. In almost all cases, as n grows, so will O. The degree to which O grows according to n determines if it is a fast or slow algorithm.



Time Complexity

The time it takes for the algorithm to finish, compared with the number of elements.

Space Complexity

The space, in memory, which the algorithm needs to complete the set task. As the number of elements increases, so too can the memory requirement. However, this is not true for in-place algorithms which need no extra memory.

e.g. the merge sort algorithm requires more memory as it is a 'divide-and-conquer' algorithm; it divides the data set into many smaller data sets. It's time complexity is *O(n)*

On the other hand, bubble sort requires no extra memory, so it can be referred to as an in-place algorithm; it's space complexity will be *O(1)*

Constant complexity - O(1)

No matter the input length, the algorithm's run time/required memory will stay the same.

This function is of constant complexity:

```
void constantPrint(int *array) {
      printf("%i", array[i]);
}
```

Linear complexity - O(n)

The run time or memory required will increase as the input length, n, increases.

Consider this function:

```
void printArray(int *array) {
      int n = sizeof(array) / sizeof(int);

      for (int i = 0; i < n; i++) {
            printf("[%i]: %i", i, array[i]);
      }
}
```

Quadratic complexity - O(n^2)

Requires a number of steps equal to the square of the input length.

The following function is of quadratic complexity, as it iterates n^2 times

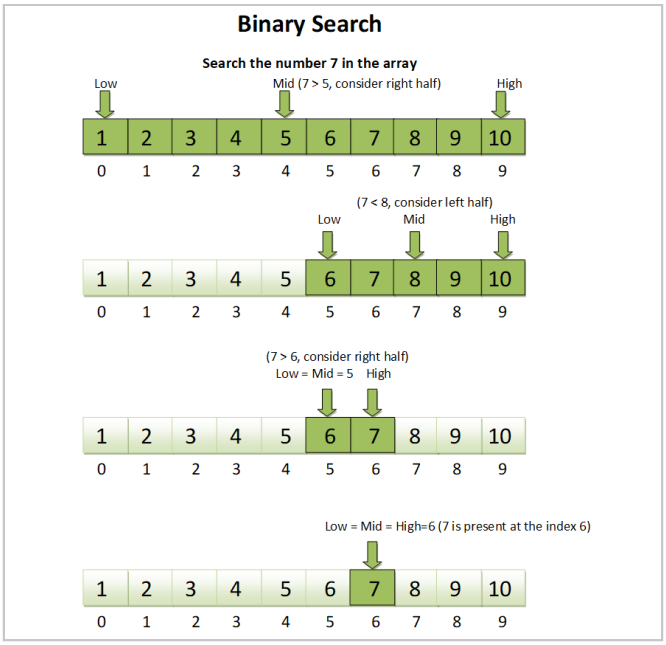
```
void overlyComplexFunction(int *array) {
      int n = sizeof(array) / sizeof(int);

      for (int x = 0; x < n; x++)
            for (int y = 0; y < n; y++)
                  printf("X:%i Y:%i", x, y);
}
```

Logarithmic complexity – O(log n)

Those algorithms with logarithmic time complexity are often the best suit for large data sets.

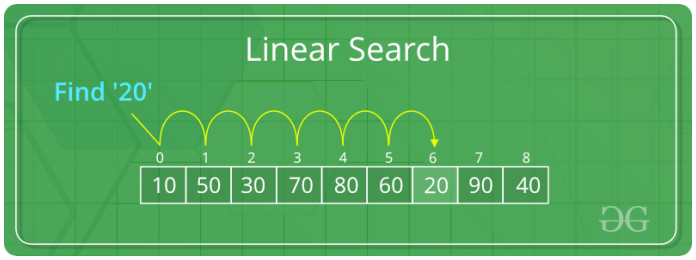
Binary search is an example of a logarithmic algorithm, as the data being searched gets half as long each iteration, making it quicker each subsequent step. “Each iteration, the time to find the target element is decreased by a magnitude inversely proportional to n”



Searching

Linear Search

A very inefficient way of searching. Should only be used for small data sets, as this technique requires each element to be compared when iterating through the whole set.




```
int search(int array[], int target) {
        for (int i = 0; i < array.length; i++) {
                if (array[i] == target) {
                        return i; // found index of the target in the array
                }
        }
        return -1; // not found
}
```

Time Complexity		
Best-Case	Average-Case	Worst-Case
O(1) – found on first element	O(n/2)	O(n) – not found or last element

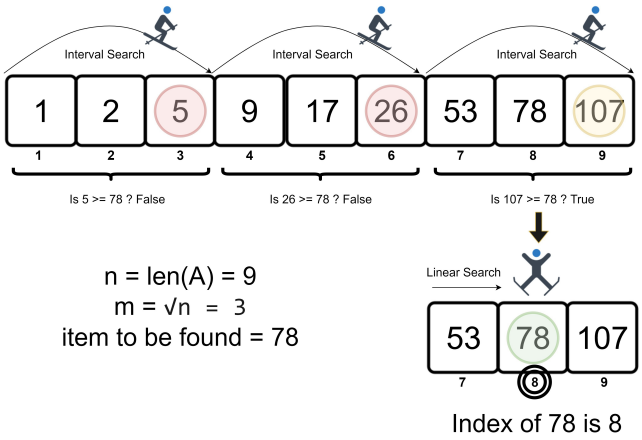
Binary Search

To perform a binary search, the data set must already be sorted. A binary search will split the data in halves each iteration, choosing the portion which the target is in. Because of this, it's a very efficient search algorithm, as even with large data sets you will have a low time complexity. This algorithm can be implemented iteratively or recursively.

Time Complexity		
Best-Case	Average-Case	Worst-Case
O(1)	O(log n)	O(log n)

Jump Search

The Jump Search is similar to a linear search, just with extra steps. First, the data set must be sorted. The Jump Search works by jumping through the data, then going back when the current value is greater than the target. Instead of stepping one index at a time, the amount of indexes to jump depends on the number of elements, and is usually the square root of n.

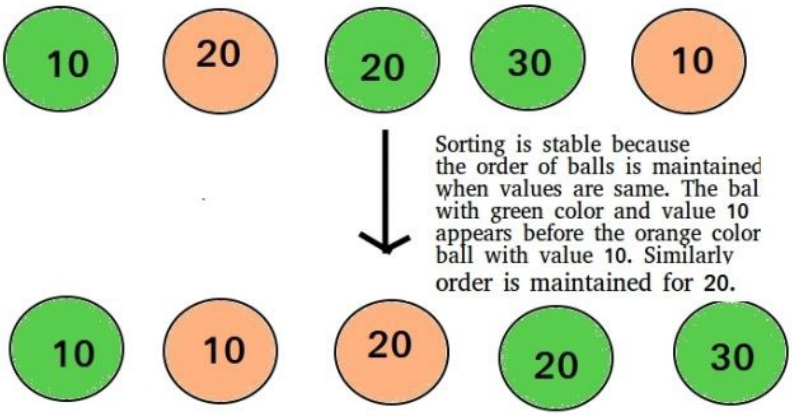


Time Complexity	
Best-Case	Worst-Case
O(1)	O(√n)

Sorting

There are many algorithms for sorting data. See a comparison of the most used algorithms. There are different classifications of algorithms, including Exchanging, Merging, Insertion, Selection and Partitioning. The algorithm you choose can depend on your memory availability and the order(s) you want to sort by.

When an algorithm is stable, this means that if there are duplicate values in a set, the duplicates will stay in their original order. Sometimes, you may need to sort a set of data by two values, e.g. if a Student (object) has the fields Name and Classroom, you want the Classroom field to be sorted first, and then you want the students to be sorted alphabetically within their class. This is when you need to use a Stable algorithm.



First, the students are sorted by Name, alphabetically. Then they are sorted by Classroom, which should result in every student from Class A being listed at the top, alphabetically, and so on.

See the difference below of a Stable vs Unstable algorithm when sorting by multiple fields.

Unstable algorithm result after sorting first by Name, then by Classroom	***Stable algorithm result***
Carol, A	Carol, A
Dave, A	Dave, A
Ken, A	Ken, A
Eric, B	Alice, B
Alice, B (should be before Eric)	Eric, B

Since a stable algorithm will ignore when two values are the same, it becomes possible to sort by multiple values. On the other hand, an unstable algorithm will not ignore when two values are the same, and can produce inconsistencies.

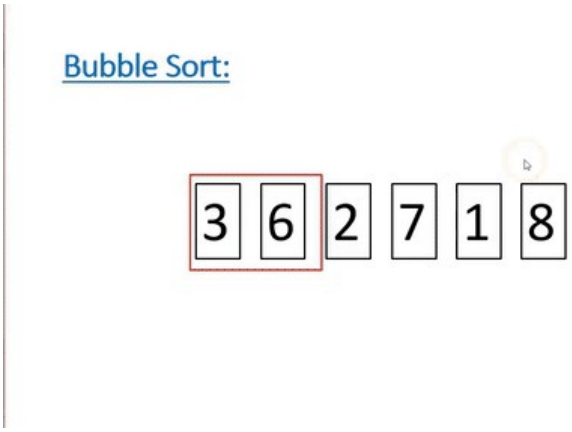
Insertion Sort

Insertion sort virtually splits the data into a sorted and unsorted part. The values from the unsorted part are taken and placed into the sorted part, similar to how we sort playing cards in our hands.

Stable?	Memory	Best-Case	Average-Case	Worst-Case
Yes	*O(1)*	*O(n)*	*O(n^2)*	*O(n^2)*

Bubble Sort

Is an **exchanging** method of sorting. It isn't a very efficient algorithm for medium-large data sets. It works by comparing adjacent elements and swapping them if they are in the wrong order.

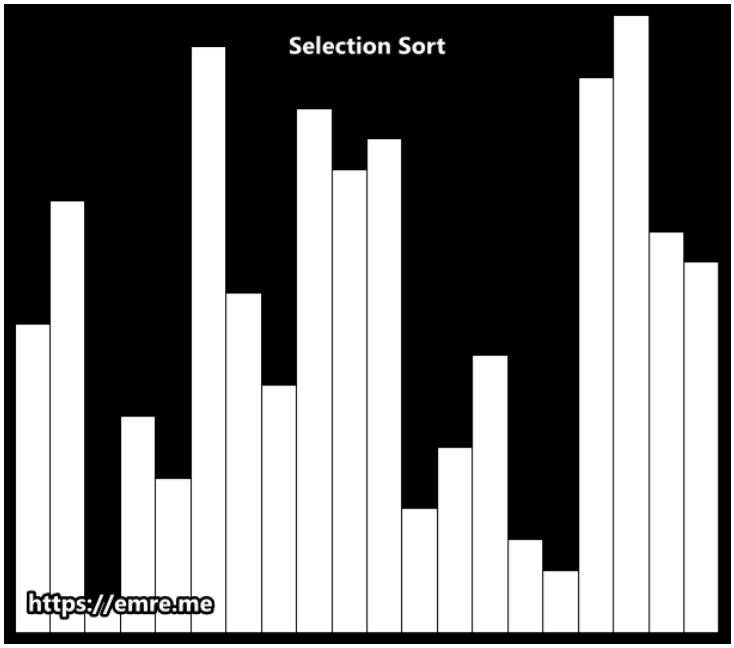


Stable?	Memory	Best-Case	Average-Case	Worst-Case
Yes	*O(1)*	*O(n)*	*O(n^2)*	*O(n^2)*

Selection Sort

Should only really be used on smaller data sets, in environments where memory is limited.

Similar to the insertion sort, it divides the set into sorted and unsorted parts. It scans from left to right, picking the smallest value from the unsorted part and placing it on the end of the sorted part.

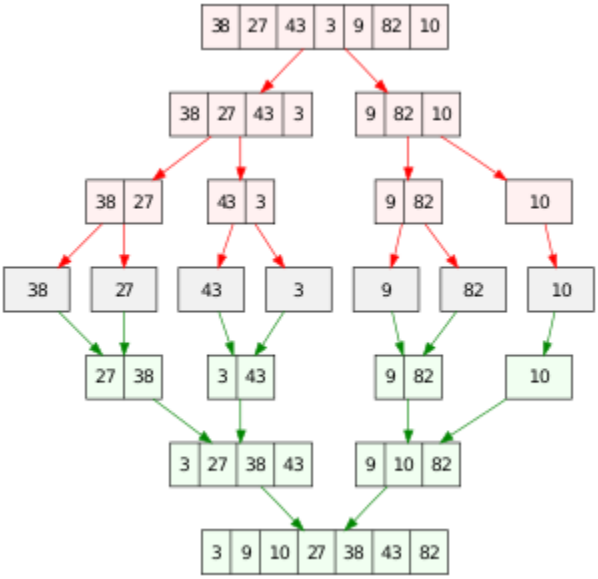


Stable?	Memory	Best-Case	Average-Case	Worst-Case
No	*O(1)*	*O(n^2)*	*O(n^2)*	*O(n^2)*

Merge Sort

Most implementations are stable. It works by dividing the unsorted set into *n* subsets, each containing one value, as a list with only one element is considered to be sorted. Then you repeatedly merge the subsets to produce new sorted sets. Do this until there is only one set remaining. There are multiple variants of the merge sort.

As you'd expect, this takes quite a lot of memory, especially for large data sets.



Merge Sort

Stable?	Memory	Best-Case	Average-Case	Worst-Case
Yes	*O(n)*	*O(n log n)*	*O(n log n)*	*O(n log n)*

Block Sort

There is an in-place variant of merge sort named Block Sort, or Block Merge Sort which combines merge operations with insertion sort. It is stable.

Stable?	Memory	Best-Case	Average-Case	Worst-Case
Yes	*O(1)*	*O(n)*	*O(n log n)*	*O(n log n)*

Quick Sort

An algorithm which uses partitioning. It is in-place, and when implemented well, can be faster than merge sort and heap sort.