

Algorithm Complexity

Written and composed by Laurent Haan

<http://www.progressive-coding.com>

Introduction to Algorithm Complexity

*[...] an **algorithm** is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.*

Wikipedia: <http://en.wikipedia.org/wiki/Algorithm>

In Computer Science, it is important to measure the quality of algorithms, especially the specific amount of a certain resource an algorithm needs. Examples of such resources would be time or memory storage. Nowadays, memory storage is almost a non-essential factor when designing algorithms but be aware that several systems still have memory constraints, such as Digital Signal Processors in embedded systems.

Different algorithms may complete the same task with a different set of instructions in less or more time, space or effort than other. The analysis and study of algorithms is a discipline in Computer Science which has a strong mathematical background. It often relies on theoretical analysis of pseudo-code.

To compare the efficiency of algorithms, we don't rely on abstract measures such as the time difference in running speed, since it too heavily relies on the processor power and other tasks running in parallel. The most common way of qualifying an algorithm is the Asymptotic Notation, also called Big O.

Asymptotic Notation

The symbol O is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, simpler function.

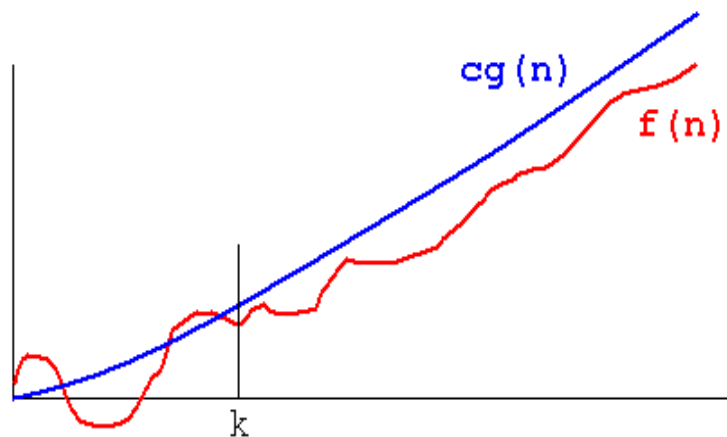
$$f(x) \text{ is } O(g(x)) \text{ as } x \rightarrow \infty$$

This means that for $x > k$, when x tends to infinity, the value $f(x)$ will always be inferior to $C * g(x)$ (with C a constant).

The idea behind this notation is that it allows us to qualify the efficiency of an algorithm by telling us how often a certain algorithm will execute operations before terminating. Let's start with a simple example:

```
void f ( int a[], int n )
{
    int i;
    printf ( "N = %d\n", n );

    for ( i = 0; i < n; i++ )
        printf ( "%d ", a[i] );
    printf ( "\n" );
}
```



In this function, the only part that takes longer as the size of the array grows is the loop. Therefore, the two `printf` calls outside of the loop are said to have a constant time complexity, or $O(1)$, as they don't rely on N . The loop itself has a number of steps equal to the size of the array, so we can say that the loop has a linear time complexity, or $O(N)$. The entire function f has a time complexity of $2 * O(1) + O(N)$, and because constants are removed, it's simplified to $O(1) + O(N)$.

Now, asymptotic notation also typically ignores the measures that grow more slowly because eventually the measure that grows more quickly will dominate the time complexity as N moves toward infinity. So by ignoring the constant time complexity because it grows more slowly than the linear time complexity, we can simplify the asymptotic bound of the function to $O(N)$, so the conclusion is that f has linear time complexity. If we say that a function is $O(N)$ then if N doubles, the function's time complexity at most will double. It may be less, but never more. That's the upper bound of an algorithm, and it's the most common notation.

Please note that this notation doesn't imply that an algorithm will always need the maximum complexity, since the termination condition to stop an algorithm might be fulfilled very early. An example would be a sort algorithm in $O(n^2)$ who would try to sort an already sorted array. In this case we're talking about best-case complexity and even that might vary from algorithm to algorithm.

Thanks to Julienne Walker from <http://www.etrnallyconfuzzled.com/> for providing a wonderful explanation about asymptotic notations.

Linear Search

In Computer Science, linear search is a search algorithm that tries to find a certain value in a set of data. It operates by checking every element of a list (or array) one at a time in sequence until a match is found:

```
int find ( int a[], int n, int x )
{
    int i;

    for ( i = 0; i < n; i++ ) {
        if ( a[i] == x )
            return i;
    }

    return 0;
}
```

1. What is the complexity of this algorithm ?
2. What is the average number of comparisons that are needed ?
3. What is the best/worst case complexity ?
4. Is there a difference if the array is already sorted ?

Binary Search

Binary Search only works on sorted lists (or arrays). It finds the median, makes a comparison to determine whether the desired value comes before or after it and then searches the remaining half in the same manner.

```
int find ( int a[], int n, int x )
{
    int i = 0;

    while ( i < n ) {
        int mid = ( n + i ) / 2;

        if ( a[mid] < x )
            n = mid;
        else if ( a[mid] > x )
            i = mid + 1;
        else
            return mid;
    }

    return 0;
}
```

We can call this an $O(N)$ algorithm and not be wrong because the time complexity will never exceed $O(N)$. But because the array is split in half each time, the number of steps is always going to be equal to the base-2 logarithm of N , which is considerably less than $O(N)$. So an even better choice would be to set the upper bound to $\log N$, which is the upper limit that we know we're guaranteed never to cross. Therefore, a more accurate claim is that binary search is a logarithmic, or $O(\log_2 N)$, algorithm.

Bubble Sort

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical or lexicographical order. There exist numerous sort algorithms and their efficiency varies greatly!

Bubble sort is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Although simple, this algorithm is highly inefficient and is rarely used except in education. For sorting small numbers of data (e.g. 20) it is better than Quicksort.

```
void bubbleSort(int *array, int size)
{
    int swapped = 0;
    int x;
    do
    {
        swapped = 0;
        for (x = 0; x < size-1;x++)
        {
            if (array[x]>array[x+1])
            {
                swap(&array[x], &array[x+1]);
                swapped = 1;
            }
        }
    } while (swapped);
}
```

1. What is the worst case performance ?
2. What is the best case performance ?
3. Can you think of a way to improve the performance ?

Selection Sort

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear search and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on. Selection sort is unique compared to almost any other algorithm in that its running time is not affected by the prior ordering of the list, it performs the same number of operations because of its simple structure.

```
void selectionSort(int *array, int size)
{
    int x,y,min;
    for (x = 0; x < size-1; x++)
    {
        min = x;
        for (y=x+1; y<size; y++)
        {
            if (array[y] < array[min])
            {
                min = y;
            }
        }
        /* swap the places */
        swap(&array[x], &array[min]);
    }
}
```

1. What is the worst case performance ?
2. What is the best case performance ?
3. What advantage does Selection Sort have ?

Insertion Sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list.



```
void insert(int *array, int pos, int value);
```

```
void insertionSort(int *array, int size)
{
    int x;
    for (x = 0; x < size; x++)
    {
        insert(array, x, array[x]);
    }
}
```

```
void insert(int *array, int pos, int value)
{
    pos--;
    while (pos >= 0 && array[pos] > value)
    {
        array[pos+1] = array[pos];
        pos--;
    }
    array[pos+1] = value;
}
```

1. What is the worst case performance ?
2. What is the best case performance ?
3. Can you think of a way to improve the performance ?

Heap Sort

Heapsort is a much more efficient version of Selection Sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. It is removed and placed at the end of the list, then the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes $O(\log n)$ time, instead of $O(n)$ for a linear scan as in simple selection sort. This allows Heapsort to run in $O(n \log n)$ time.

Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by dividing the list into two sublists which are then sorted again with Merge Sort. This continues until the size of the list equals one (which is sorted). Then it merges those sublists again into bigger lists and only a few simple steps are required to merge two sorted lists.

```
function mergesort(m, start, end)
  if length(m) ≤ 1
    return m
  else
    middle = start+end / 2
    left = mergesort(m, start, middle)
    right = mergesort(m, middle, end)
    result = merge(left, right)
    return result
```

Since Merge Sort is a recursive implementation working on two independent sublists, it can be efficiently implemented in parallel using threads. The speed gain of a parallel execution is doubled on dual core processors.

Quick Sort

Quicksort is a divide and conquer algorithm, which relies on a partition operation: to partition an array, we choose an element, called a *pivot*, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in linear time and in-place. We then recursively sort the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest $O(\log n)$ space usage, this makes quicksort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower ($O(n^2)$) performance, but if at each step we choose the *median* as the pivot then it works in $O(n \log n)$.

Exercise: Stone Age boulders

Two stone age men have gathered an impressive collection of boulders in their cave, all of different size and weight, standing neatly one after the other, in the order they have been collected. To restore some order in the room, they want to arrange the boulders from the smallest to the largest, with the smallest at the entrance of the cave and the largest close to the back wall.

Each boulder is only represented by its weight, so the heavier it is, the larger it is (we assume that they are all made of the same material). As there are only 2 stone age men, and the space inside the cave is limited, they are only allowed to swap two boulders at a time. Additionally, to save their energy, they want to use a method that allows them to move the minimum necessary weight only.

Write an algorithm that takes an array of boulders and orders it from the smallest to the largest, by only swapping two boulders at a time but with the least effort in terms of kilos moved.

Example:

{5, 3, 1}

-> {1, 3, 5} and necessary effort: $1+5 = 6$

{6, 4, 1, 2}

-> {6, 1, 4, 2}, effort 5

-> {6, 2, 4, 1}, effort 3

-> {1, 2, 4, 6}, effort 7

total effort: 15