Data structures and Algorithms

Complexity (Big O)

Complexity of an algorithm measures time and memory. How long does one algorithm takes to finish? How much memory does it use?

We measure these values in terms of the input size (Usually for the input size we use the letter n)

We can analyze algorithms in the Best case, Average case and Worst case.

Best case measures the quickest way for an algorithm to finish.

Average case measures what time does the algorithm need to finish on average.

Worst case measures the slowest time for an algorithm to finish.

General idea

1. Big O: $\mathcal{O}(n^2)$ - the algorithm takes **at most** n^2 steps to finish in the **WORST CASE**.
2. Little o: $o(n^2)$ - the algorithm takes **less than** n^2 steps to finish in the **WORST CASE**.
3. Big Omega: $\Omega(n^2)$ - the algorithm takes **at least** n^2 steps to finish in the **BEST CASE**.
4. Little Omega: $\omega(n^2)$ - the algorithm takes **more than** n^2 steps to finish in the **BEST CASE**.
5. Big Theta: $\Theta(n^2)$ - the combination of $\mathcal{O}(n^2)$ and $\Omega(n^2)$. Meaning the algorithm takes at least n^2 steps to finish and takes at most n^2 steps to finish, i.e. it always takes n^2 steps.

Comparison of orders of common functions

$$\mathcal{O}(1) < \mathcal{O}(loglog(n)) < \mathcal{O}(log(n)) < \mathcal{O}(log^2(n)) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(nlog(n)) < \mathcal{O}(n^2) < \\ < \mathcal{O}(n^3) < \mathcal{O}(2^n) < \mathcal{O}(3^n) < \mathcal{O}(n!) < \mathcal{O}(n^n) < \mathcal{O}(3^{n^2}) < \mathcal{O}(2^{n^3})$$

Formal definitions

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0, \exists n_0 > 0 \mid 0 \leq g(n) \leq cf(n)\ \forall n \geq n_0\}$$

$$\Theta(f(n)) = \{g(n) : \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0 \mid 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\ \forall n \geq n_0\}$$

$$\Omega(f(n)) = \{g(n) : \exists c > 0, \exists n_0 > 0 \mid 0 \leq cf(n) \leq g(n)\ \forall n \geq n_0\}$$

$$o(f(n)) = \{g(n) : \forall c > 0\ \exists n_0 > 0 \mid 0 \leq g(n) < cf(n)\ \forall n \geq n_0\}$$

$$\omega(f(n)) = \{g(n) : \forall c > 0\ \exists n_0 > 0 \mid 0 \leq cf(n) < g(n)\ \forall n \geq n_0\}$$

Examples

$\mathcal{O}(34n^4 + 5) = n^4$

$\mathcal{O}(2^n + n^{1024} + log(n)) = 2^n$

$\mathcal{O}(5N + \frac{1}{2}M) = N + M$

$\mathcal{O}(n^2)$ means that if our algorithm takes input of 10 elements it will need 100 steps to finish in the worst case. If the input is of 100 elements it will need 10000 steps to finish in the worst case.

Sorting

Sorting means ordering a set of elements in a sequence.

We can sort a set of elements whose elements are in partial order. Partial order is a relation with the following properties:

1. Antisymmetry - For every 2 elements in the set (A, B), if $A \leq B$ and $B \leq A$ then $A = B$.
2. Transitivity - For every 3 elements in the set (A, B, C), if $A \leq B$ and $B \leq C$ then $A \leq C$.
3. Reflexivity - For every element in the set A, $A \leq A$.

Sorting properties

Stable sort

A sorting algorithm is stable if two equal objects appear in the same order in the ordered output as they appeared in the unsorted input.

> Example input: $1, 2, 3_a, 8, 5, 3_b$. Here 3_a and 3_b are simply a 3 but we have marked them to follow what happens with them after the sorting.
>
> Example output: $1, 2, 3_a, 3_b, 5, 8$
>
> Unstable sort output: $1, 2, 3_b, 3_a, 5, 8$

In place sort

Uses only a small constant amount of extra memory. In place sort means the elements are swapped around. Out of place means we use another extra array to swap items around.

Number of comparisons

How many times do we need to compare 2 elements. For most sorts this represents the time complexity.

Adaptive sort

Determines whether the sorting algorithm runs faster for inputs that are partially or fully sorted. If an algorithm is unadaptive it runs for the same time on sorted and unsorted input. If an algorithm is adaptive it runs faster on sorted input than on unsorted input.

Online

Determines if an algorithm needs all the items from the input to start sorting. If an algorithm is online it can start sorting input that is given in parts. If an algorithm is offline it can start sorting only after it has the whole input.

External sort

Allows sorting data which cannot fit into memory. Such algorithms are designed to sort massive amounts of data (Gigabytes, Terabytes, etc.)

Parallel sort

An algorithm which can be ran on multiple threads at the same time, speeding the running time.

Helper code

```
void swap(int & a, int & b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Bubble sort

Bubble sort	
Time complexity	$\mathcal{O}(length^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	Yes
Stable	Yes
Number of comparisons	$\mathcal{O}(length^2)$
Number of swaps	$\mathcal{O}(length^2)$
Online	No
In place	Yes

Clean code

```
void bubbleSort(int * array, int length) {
    for (int bubbleStartIndex = 0; bubbleStartIndex < length;
bubbleStartIndex++) {
        for (int bubbleMovedIndex = 0; bubbleMovedIndex < length - 1;
bubbleMovedIndex++) {
            if (array[bubbleMovedIndex] > array[bubbleMovedIndex+1]) {
                swap(array[bubbleMovedIndex], array[bubbleMovedIndex+1]);
            }
        }
    }
}
```

Short code

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        for (int k = 0; k < length - 1; k++) {
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
            }
        }
    }
}
```

Optimization 1 - make it faster

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        for (int k = 0; k < length - 1 - i; k++) { // length - 1 - i = 2x less
iterations
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
            }
        }
    }
}
```

Optimization 2 - make it adaptive

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        bool swappedAtLeastOnce = false; // add a flag to check if a swap has
occurred

        for (int k = 0; k < length - 1 - i; k++) {
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
                swappedAtLeastOnce = true;
            }
        }

        if (!swappedAtLeastOnce) { // if there were no swaps, it's ordered
            break;
        }
    }
}
```

Selection sort

Selection sort	
Time complexity	$\mathcal{O}(length^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	No
Stable	Yes
Number of comparisons	$\mathcal{O}(length^2)$
Number of swaps	$\mathcal{O}(length)$
Online	No
In place	Yes

Clean code

```
void selectionSort(int * array, int length) {
    for (int currentIndex = 0; currentIndex < length; currentIndex++) {
        int smallestNumberIndex = currentIndex;
```

```
        for (int potentialSmallerNumberIndex = currentIndex + 1;
                potentialSmallerNumberIndex < length;
                potentialSmallerNumberIndex++) {
            if (array[potentialSmallerNumberIndex] < array[smallestNumberIndex])
{
                smallestNumberIndex = potentialSmallerNumberIndex;
            }
        }

        swap(array[currentIndex], array[smallestNumberIndex]);
    }
}
```

Short code

```
void selectionSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        int index = i;

        for (int k = i + 1; k < length; k++) {
            if (array[k] < array[index]) {
                index = k;
            }
        }

        swap(array[i], array[index]);
    }
}
```

Insertion sort

Insertion sort	
Time complexity	$\mathcal{O}(length^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	Yes
Stable	Yes
Number of comparisons	$\mathcal{O}(length^2)$
Number of swaps	$\mathcal{O}(length^2)$
Online	Yes
In place	Yes

Clean code

```
void insertionSort(int * array, int length) {
    for (int nextItemToSortIndex = 1; nextItemToSortIndex < length;
nextItemToSortIndex++) {
        for (int potentiallyBiggerItemIndex = nextItemToSortIndex;
                potentiallyBiggerItemIndex > 0 &&
                array[potentiallyBiggerItemIndex] <
array[potentiallyBiggerItemIndex - 1];
                potentiallyBiggerItemIndex--) {
            swap(array[potentiallyBiggerItemIndex],
array[potentiallyBiggerItemIndex-1]);
        }
    }
}
```

Short code

```
void insertionSort(int * array, int length) {
    for (int i = 1; i < length; i++) {
        for (int k = i; k > 0 && array[k] < array[k - 1]; k--) {
            swap(array[k], array[k-1]);
        }
    }
}
```