4. Algorithm efficiency

We have compared the efficiency of our sorting algorithms. Studying algorithm efficiency and having a language to describe it is an important part of the science of Computer Science.

From the IB CS syllabus:

Students should understand and explain the difference in efficiency between a single loop, nested loops, a loop that ends when a condition is met or questions of similar complexity. Students should also be able to suggest changes in an algorithm that would improve efficiency, for example, using a flag to stop a search immediately when an item is found, rather than continuing the search through the entire list. Examination questions will involve specific algorithms (in pseudocode/flowcharts), and students may be expected to give an actual number (or range of numbers) of iterations that a step will execute.

As you can see from the above, it is important to understand the efficiency of different algorithms. One measure used in industry is called Big O notation.

Big-O describes the relationship of how the runtime will scale with respect to certain input variables.

Some common examples of Big-O expressions:

● `O(1)`	constant
● `O(log(n))`	logarithmic
● `O(n)`	linear
● `O(n^2)`	quadratic
● `O(n^c)`	polynomial
● `O(c^n)`	exponential

If the run time will increase linearly with respect to the size of the input data, this would be `O(n)`. An example is to loop through the values of an array.

If the run time will increase exponentially with respect to the size of the input data, this would be `O(n^2)`. An example is to have a loop instead a loop, where both iterate through the values of an array.

A good introduction to Big O notation can be found with the following video:

HackerRank (2016), Big O notation
https://www.youtube.com/watch?v=v4cd1O4zkGw

There were four important rules from the video:

Rule 1 - If you have two important steps in your algorithm, you add those steps.

```
function something() {
    doTask1()        // O(a)
    doTask2()       // O(b)
}
// Overall result = O(a+b)
```

Rule 2 - Drop constants.

```
function something() {
    for each item in array:     // O(n)
        min = MIN(item, min)
    for each item in array:     // O(n)
        max = MAX(item, max)
}
// The overall run time will still increase linearly, so it is O(n) not O(2n).
```

Rule 3 - Different inputs usually use different variables to represent them in the O() relationship.

```
function something() {
    for each item in A:
       total = total + item
    for each item in B:
       total = total + item
    return total
}
// The overall runtime would be O(a*b)
```

Rule 4 - Drop non-dominant terms.

For example, if an algorithm as a nested for-loop which would be O(n^2) and a regular for-loop O(n), it is not O(n + n^2) because the O(n) is going to be completely dominated by the O(n^2) to such a degree that it is meaningless to worry about.

Finally,

Here is also a great analogy from the comments section of that video: Let's say you're making dinner for your family. O is the process of following a recipe, and n is the number of times you follow a recipe.

- `O(1)` - you make one dish that everyone eats whether they like it or not. You follow one recipe from top to bottom, then serve (1 recipe). <-- How I grew up
- `O(n)` - you make individual dishes for each person. You follow a recipe from top to bottom for each person in your family (recipe times the number of people in your family).
- `O(n^2)` - you make individual dishes redundantly for every person. You follow all recipes for each person in your family (recipe times the number of people squared).
- `O(log n)` - you break people into groups according to what they want and make larger portions. You make one dish for each group (recipe times request)

For a graphical comparison that illustrates the difference in processing load of different O() algorithms, check the https://bigocheatsheet.com website.

4.1 Big-O Practice

What is the Big O in the following?

Question 1

```
boolean containsValue(int[] list, int val) {
   for (int item : list) {
      if (item==val) {
         return true;
      }
   }
   return false;
}
```

Question 2

```
boolean containsDuplicates(int[] a, int[] b) {
   for (int x : a) {
      for (int y : b) {
         if (x==y) { return true; }
      }
   }
   return false;
}
```

Question 3

```
int fibonacci(int n) {
   if (n <= 1) { return n; }
   int fib = 1;
   int prev = 1;
   for (int i=2; i<n; i++) {
      int temp = fib;
      fib += prev;
      prev = temp}
   return fib
}
```

Question 4

```
boolean isFirstElementZero(int[] a) {
   if (a[0] == 0) {
      return true;
   } else {
      return false;
```

```
    }
}
```

Question 5

Which is algorithmically more efficient?

You are maintaining a customer contact details database for a sales business, currently with 1+ million records. On any given day, several hundred customers will notify a change of address for the database. Additionally, on any given day, the support staff will receive several thousand calls through the switchboard, where they will have to look up the customer data.

Should your program...?

- Save new changes as they occur, and have the support desk app sequentially search through the records? ... or ...
- Bubble sort after every save, and have the support desk app binary search through the records.

Justify your choice with reference to the O(n) of each method. [5 marks]

Question 6

A supermarket inventory system needs to be updated to keep an accurate record of the various stock on hand. The supermarket carries a range of about 10 000 different items on its shelves. The hard disk of the computer used is efficient at performing data lookups, but takes about 100 times longer to write to disk than a lookup.

With approximately 1000 transactions per day, would it be more efficient to:

- Selection sort the data after each individual sale? or
- Bubble sort the data after closing each day?