Big O notation notes

Given n to be the length of an array/collection/data structure/data set:

**O(n)** 🡪 linear complexity: as many steps as elements in the structure:

* traversing/printing the array/structure
* linear search

**O(n2)** 🡪 quadratic complexity: ***n times n***, comparing each element to all the other elements

If an algorithm has the following structure:

outer loop : n times

inner loop : n times

5 steps

**n × ( n × 5 ) =**

**n × 5n =** (disregard the constant **5**, it doesn’t affect the complexity as much as ***n*** does)

**n × n = n2** (therefore, ***n2*** is the dominant term)

Bubble, selection and insertion sorts = **O(n2)** because they have ***2 nested loops*** depending mostly on the length of the array/collection/data structure.

|  |  |
| --- | --- |
| Tic tac toe board display method  loop for each row:  loop for each column:  output element  output column separator  loop to output row separators | 1|2|3  **-----**  4|5|6  **-----**  7|8|9 |

**n × (n + n) =**

**n × (2n) =**

**n2**

**O(log2 n) 🡪** **logarithmic complexity: “**halving” the problem until solved/done

binary search, as it repeatedly halves the number of items (n) until it finds the key or realises that the key is not in the data set.

**O(1)** 🡪 constant time; solution does not depend on the size of the data set/structure/array

System.out.println(a[0]); System.out.println(a[2]); System.out.println(a[a.length]);

accessing an array element (by index)

getting the length of an array or structure (like when you have a counter to keep track of how many elements are in a list)

accessing RAM 🡪 random access implies the ability to go to a specific memory address without going through all the previous addresses, which would be sequential or linear access (think cassette tapes, VCR tapes, and also lists)

**O(2n)** 🡪 exponential order of growth/time complexity

recursive Fibonacci (HL example)

Some extra links, if you want to read more about algorithm complexity (optional)

* <https://en.wikipedia.org/wiki/Big_O_notation>
* <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/>