Big O  
  
<https://www.youtube.com/watch?v=ZO6GjoXEwYQ&list=PLTd6ceoshprfdzLWovxULl8Lt7RAFKTe5&index=6&t=0s>

* Big O describes how the time taken, or memory used by a program scales with the amount of data it has to work on.
* Big O describes the ‘complexity’ of a program.
* Complexity describes the way that the time taken grows according to the amount of data.
* Common sense tells us that a program takes longer when there is more data to work on… But not necessary

Linear Search

* Sometimes referred to as a sequential search.
* An unordered list is searched for a particular value.
* Each value in the list is compared with the target value. (A brute force approach).
* Usually implemented with a single loop.
* If the target value is at the end of the list (worst case scenario), then increasing the size of the list will increase the size of the search time. The search time is directly proportional to the amount of data.

For I = 0 TO n-1

If ArrayToSearch(i) = Target THEN

bFound = True

EXIT FOR

END IF

NEXT i

Linear Search Complexity

* For n data items, the time taken is equal to some constant multiplied by n.
* The Big O time complexity is known as Linear. O(n)

Stack

* Items are pushed onto and popped off the top of a stack.
* A stack might also include a Peek operation, which looks at the top item without removing it.
* Known as a Last in, first out data structure (LIFO).
* Usually implemented with an array and a pointer to the top item.
* Adding or removing from a stack takes the same amount of operational time, regardless of how many items are in the stack.

**Push (adding to a stack) example:**

Procedure Push

IF Top = MaximumSize THEN //check if memory available

OUTPUT “Stack overflow”

ELSE

Top = Top + 1 //increment top pointer

ArrayStack(Top) = new item //new item placed in the pos of pointer

END IF

END Procedure

//----------------------------------

**Pop (removing from stack) example:**

Procedure Pop

IF Top = 0 THEN //check if stack has elements for removal

OUTPUT “Stack is empty”

ELSE

copy item = ArrayStack(Top) //grab item from top by copying it

Top = Top - 1 //decrement top pointer

END IF

END Procedure

NOTE: We don’t necessarily have to delete the item form the stack, just change the pos of the pointer. (The item will be overridden next time a new element is popped onto this pos).

Stack Operations Complexity

* Increasing the amount of data makes no difference to the time taken by pushing or popping.
* The Big O time complexity is known as Constant. O(1)

The Dominant Term

* An algorithm working on a data structure of size n might take 5n3 + n2 + 4n + 3 steps.
* The larger n becomes (5n3 in the example above), the less significant the smaller terms become, so we ignore everything except 5n3.
* We can also ignore any constants (fixed numbers. ‘5’ in this case), so the Big O time complexity of the above algorithm is O(n3)

Bubble Sort

* Sorts a list of items into numeric or alphabetical order.
* It scans a list, comparing pairs of values and swapping their positions if necessary.
* For n amount of data items, the list is scanned like this n-1 times.
* Various enhancements are possible, to improve its performance.
* Not very efficient. Doubling the amount of data with this sort, **quadruples** the amount of time taken!

FOR j = 1 to n - 1 //from 1 to n-1 so that the final current is checking the final element\*\*

FOR i = 0 TO n - 2 //loop up to 2nd last pos, as current element checks next element

IF ArrayToSort(i) > ArrayToSort(i + 1) THEN //check if next element is bigger than current

Swap ArrayToSort(i) with ArrayToSort(i + 1) //swap if so

END IF

NEXT i

NEXT j

\*\* example list: 0123

length = 4. Requires 3 searches: 01, 12, 23. This is done with “1 to n – 1” (1 to 3).

Bubble Sort Complexity

* For n data items to check, a simple implementation performs (n - 1) \* (n – 1) operations to sort.
* This can be written as n2 – 2n + 1, and the dominant term is n2.
* The Big O complexity is known as Quadratic. O(n2)

Enhanced Bubble Sort

* After the first pass of sorting, the largest item will always be in the correct position (the last position in the list).
* Second largest item is in the correct place after the next pass (second last pos)
* And so on…
* The inner loop can therefore run one less time with each pass of the outer loop.

FOR j = 1 to n-1

FOR i=0 to (n-1) - j //sort up to one less position for each pass

IF ArrayToSort(i) > ArrayToSort(i+1) THEN

Temp = ArrayToSort(i) //assign current to temp

ArrayToSort(i) = ArrayToSort(i+1) //make current the value of next

ArrayToSort(i+1) = Temp //make next the value of temp (current)

END IF

NEXT i

NEXT j

Enhanced Bubble Sort Complexity

* For n items of data, the enhanced bubble sort algorithm performs (n-1) + (n-2) + (n-3) + … + 3 +2 + 1 operations.
* This can be shown to be (n2 - n)/2.
* This is a 50% reduction in the time taken, but…
* The dominant term is still n2.
* The complexity is still Quadratic. O(n2).

Alternative Enhanced Bubble Sort

* If the inner loop performs no swaps, then the data must now be in the correct order.
* So we can check for swaps with a Boolean variable.
* And force an early exit from the bubble sort, when there’s no more work to do.

REPEAT

Swapped = False //swapped starts false

FOR i=0 TO Length(ArrayToSort) -2

IF ArrayToSort(i) > ArrayToSort(i+1) THEN

Swap ArrayToSort(i) with ArrayToSort(i + 1)

Swapped = True //turned true if a swap occurred

END IF

NEXT i

UNTIL Swapped = False //repeat another pass through the list as a swap previously occurred.

Best versus Worst Case Scenario

* Best case scenario - Data is already in the right order. In example above that means the inner loop will only run once.
* This means in the best case scenario the bubble sort above has Linear Complexity O(n).
* Worst case scenario – Data is in completely the wrong order, so every item has to be moved.
* This means in the worst case scenario the bubble sort above has Quadratic Complexity O(n2).

Logarithms

* Logarithms are the inverse of exponentiation. (like how multiplication is the inverse of division).
* Example 1: 23 = 8 (2x2x2 = 8). We say that 2 is the base and 3 is the exponent.
* Therefore, the logarithm of 8 is the exponent (power) to which the base of 2 must be raised in order to give us 8.
* We say the base 2 log of 8 is equal to 3: log28 = 3.
* Example 2: 104 = 10,000 (10x10x10x10). The base 10 log of 10,000 = 4: log1010,000 = 4.