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 time 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 time 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.
* If xz = y then log xy = z.

20 = 1 therefore: log21 = 0

21 = 2 therefore: log22 = 1

22 = 4 therefore: log24 = 2

23 = 8 therefore: log28 = 3

24 = 16 therefore: log216 = 4

25 = 32 therefore: log232 = 5

NOTE: That each number that were calculating the log of is twice as big as the previous number (log22, log24, log28 etc…). But each log (answer) is only 1 bigger than the previous value (1,2, 3 etc..).

Binary Search

* Used to search an ordered list for a particular value.
* Takes a “divide and conquer” approach.
* Target is compared with the middle value of the list, then half of the list is discarded repeatedly, until the target is found.
* Very efficient for large sorted lists.

iLow = lBound(DataArray) //assign the list’s lowest value (its first element) to this var

iHigh = UBound(DataArray) //assign the list’s highest value (its last element) to this var

The above vars (pointers) define the section of the array that the program is looking at. (All of it initially)

Do While iLow <= iHigh

iMiddle = (iLow + iHigh)/2 //the above pointers calculate the middle position of the array

If Target = DataArray(iMiddle) then //check if middle pointer contains the target value

bFound = True

Exit Do

//if middle pointer doesn’t contain the target value:

ElseIf Target < DataArray(iMiddle) Then //check if target value is less than middle pointer

iHigh = (iMiddle - 1) //redefine iHigh to discard the top half of the list (moving its pos -1)

Else //OR:

iLow = (iMiddle+1)//redefine iLow to discard the bottom half of the list (moving its pos +1)

End if

Loop

Binary Search Complexity

* Doubling the data of the list requires only one extra chop.
* This makes the binary search **very** efficient for very large data sets (if the data is already sorted).
* A binary search’s Big O time complexity is known as Logarithmic. O(log n).

Merge Sort

* Sorts the data in a list.
* Uses the “divide and conquer” approach.
* It splits a list into several sub lists. Each of which contains only one item, and is therefore by definition sorted.
* Pairs of these sub lists are then merged together, sorting as it goes.
* Best implemented with a recursive function.

8 \* 3 append operations are done on the example below:

Initial unordered list, split into 8 individual lists:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 66 |  | 44 |  | 63 |  | 8 |  | 10 |  | 12 |  | 144 |  | 106 |

8 append operations to group above lists into 4 lists of 2. 1 to group 2 lists, then one to swap their values:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 66 |  | 8 | 63 |  | 10 | 12 |  | 106 | 144 |

8 append operations to group above lists into 2 lists of 4. 2 to group the lists, then 3 to swap the values of one list, and 3 to swap the values of the other:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 44 | 63 | 66 |  | 10 | 12 | 106 | 144 |

8 append operations to group above lists into 1 sorted lists of 8. 1 to group the lists together, then 7 to swap all their values:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 10 | 12 | 44 | 63 | 66 | 106 | 144 |

Merge Sort Complexity

* If n = 8 (like in example above), then the merge sort does n\*3 append operations.
* We know that log2n = 3, this is n\*log2n append operations.
* The Big O time complexity of the merge sort is a cross between Linear and Logarithmic, which is Linearithmic. O(n log n).

<https://www.youtube.com/watch?v=a8he-_n2B-Q&list=PLTd6ceoshprcXjTkoI9_6zev-yU04vol1&index=13>

**Splitting a List:**

Middle = (UBound(SourceArray))/2 //create pointer to middle of array

REPEAT

ArrLeft(Ptr) = SourceArray(Ptr) //use a pointer to copy elements from first half of source array into left array

Ptr++

UNTIL Ptr = Middle //stop when ptr hits middle

PtrFill = 0 //another pointer with 0 index to fill right array with second half of source array

REPEAT

ArrRight(PtrFill) = SouceArray(Ptr) //copy elements from 2nd half of source array into right array

Ptr++

PtrFill++

UNTIL Ptr = Unbound(SourceArray) //stop when ptr hits last element of source array

**Merging two Lists:**

REPEAT

//compare list values at their respective current positions:

IF Arr1(Ptr1) < Arr2(Ptr2) THEN //if list 1’s value is less than list 2’s

Arr3(Ptr3) = Arr1(Ptr1) //assign value to new list

Ptr1++ //increment ptr1

ELSE //list 2’s value is bigger

Arr3(Ptr3) = Arr2(Ptr2) //assign value to new list

Ptr2++ //increment ptr2

END IF

Ptr3++ //increment ptr3

UNTIL Ptr3 = UBound(Arr1) + UBound(Arr2) + 1 //stop when reach end of both lists combined. (+1 for 0 indexing)

Summary

* **Constant** O(1)
  + No matter how big the data structure, the number of operations (and therefore the time taken) is fixed. In relation to the operation itself and what resources it’s using (type of computer etc). Therefore:
  + The time taken is **independent** of the amount of data that the program is given to work on.
  + Examples: Stack push, pop and peek; Queue enqueuer and dequeuer; Inserting a node into a linked list.
* **Linear** O(n)
  + The time taken is directly proportional to the amount of data it’s given to work on (doubling the number of data items, doubles the time taken). Therefore, slow with large data structures.
  + Examples: Linear search (traversing a linked list); Counting items in a list; Comparing a pair of strings.
* **Quadratic** O(n2)
  + Time taken is proportional to the square of the amount of data that it starts with. (twice as much data takes 4 times as long to process. 3 times as much data takes 9 times longer to process). Therefore, slow with large amounts of data.
  + Examples: Bubble sort; selection sort; insertion sort; traversing a 2D array.
* **Polynomial** O(nk)
  + Time taken is proportional to the amount of data raised to the power of a constant (k).
  + If k is 2, this is quadratic (shown above). If K is 3 this is cubic complexity etc…
* **Logarithmic** O(log n)
  + The time taken is proportional to the base 2 logarithm of the amount of data. (number of items in the data structure).
  + This means that if the amount of data doubles, the number of operations (and therefore time taken) grows by only one unit. Making it great for scalability.
  + This means that the impact of increasing the amount of data that the program has to work with lessens when asked to work with very large data structures.
  + Examples: Binary search a sorted list; Searching a binary tree.
* **Linearithmic** O(n log n)
  + The time taken is proportional to the logarithm of the amount of data, multiplied by the amount of data.
  + This complexity is on a par with linear. The increase in time is reasonable for small increases in the amount of data. But the impact on vary large data structures can be unacceptable.
  + Examples: Marge sort; Quick sort.
* **Exponential** O(Kn)
  + The time taken is proportional to a constant raised to the power of the number of items in the data structure.
  + This means that if the constant is 2, then the program takes twice as long when asked to work with only one extra unit of data. If the constant is 10, then one extra item of data will slow it down 10 times!
  + Examples: n-Queens problem; Traveling salesman.

Final Word

* Remember that Big O complexity is NOT about the raw performance of a program. It’s about how well a program **scales**. IE how well it maintains its performance when given more data to work with.