

Big-O Notation

yes it's the letter O not the number 0
Big-O Notation gives an upper bound of the complexity in the worst case, helping to quantify performance as the input size becomes arbitrarily large.

Ex, looking for the number 7 in a list, and it occurring last. this scenario would include all values in that list, if allocating space per number, you'd allocate the entire list.

n - The size of the input
Complexities ordered in from smallest to largest

Constant Time:	$O(1)$
Logarithmic Time:	$O(\log(n))$
Linear Time:	$O(n)$
Linearithmic Time:	$O((n)\log(n))$
Quadratic Time:	$O(n^2)$
Cubic Time:	$O(n^3)$
Exponential time:	$O(b^n), b > 1$
Factorial Time:	$O(n!)$

Big-O Properties

$O(n + c)$	=	$O(n)$
$O(cn)$	=	$O(n), c > 0$

*c = constant, n = number
if your constant is large, like 2 billion its likely to have substantial run-time penalties

Big-O Examples

let f be a function that describes the running time of a particular algorithm for an input of size n:

$$f(n) = 7\log(n)^3 + 15(n)^3 + 15n^2 + 8$$

this is Quadratic because there's $\wedge 3$ then $\wedge 2$ then $\wedge 1$ or a regular value at the end
 $O(f(n)) = O(n^3)$

The following run in constant time: $O(1)$

<code>a := 1</code>	<code>i := 0</code>
<code>b := 2</code>	<code>While i < 11 do</code>
<code>c := a + 5*b</code>	<code> i = i + 1</code>

`:=` is used to explicitly signal the initialization or assignment of a new value to a variable. and is usually used for pseudo code

additionally since c can be run in a single cycle and the while loop is always the same it will always run 11 times,

The Following runs in Linear time: $O(n)$

<pre> :0 While i < n Do i = i + 1 </pre>	<pre> :0 While i < n Do i = i + 3 </pre>
$f(n) = n$ $O(f(n)) = O(n)$	$f(n) = n/3$ $O(f(n)) = O(n)$

since $i + 1$ is $1/3$ the speed of $i + 3$ we finish in appx $1/3$ the time, regardless it will allways finish at a constant rate, if we were to plot it on a graph its a straight line.

Both of the following run in Quadratic time the first may be obvious since n work done n times is $n*n = O(n^2)$, but what about the second one?

--still pseudo code but more lua like coz end = end of loop

```

for (i := 0 ; i < n; i = i + 1 )
    for (j := 0 ; j < n; j = j + 1 )
        end
    end
end

```

$f(n) = n*n = n^2$, $O(f(n)) = O(n^2)$

```

for (i := 0 ; i < n; i = i + 1 )
    for (j := i ; j < n; j = j + 1 )
        -----^ replaced 0 with i
    end
end

```

end

encase you didn't understand the pseudo code. like how i didn't... they are declaring what i is in the for loop, then declaring the condition of the for loop if $i < n$ then giving the condition once the loop ends. eg in python

```
n = 10 # Example value for n
for i in range(n): # i goes from 0 to n-1. i is implicitly declared as 0 unless previously stated. (it hasn't)
    for j in range(n): # j goes from 0 to n-j. again implicitly declaring the value of j
        pass # replace with whatever you do after the math.
```

```
n = 10 # Example value for n
```

```
def foo():
    global i, j, exitcond
    while not exitcond:
        if i < n:
            j = i
            while j < n:
                # whatever code would go after the for j loop

                j += 1
            i += 1
        else:
            exitcond = True
            print("This is now exiting the exit condition if loop.")
```

```
# Initial values
```

```
i = 0
```

```
j = 0
```

```
exitcond = False
```

```
foo()
```

for a moment focus on the second loop....

Since i goes from [0,n) the amount of looping done is directly determined by what i is.

Remark that if $i = 0$, we do n work, we do n-2 work, etc...

so the question then becomes what is:

$(n) + (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$??

Remarkably this turns out to be $n(n+1)/2$, so

$$O(n(n+1)/2) = O((n^2)/2) + n/2 = O(n^2)$$

```
for (i := 0 ; i < n; i = i + 1 )
  for (j := i ; j < n; j = j + 1 )
```

Suppose we have a sorted array and we want to find the index of a particular value in the array if it exists. What is the time complexity of the following algorithm?

Here's a binary search

```
low := 0
high := n-1
while low <= high Do
  mid := (low + high) / 2
  if array[mid] == value: return mid
  else if array[mid] < value: lo = mid + 1
  else if array[mid] > value: hi = mid - 1
return -1 //value not found
```

ans: $O(\log_2(n)) = O(\log(n))$

It starts by making 2 pointers one at the start one at the end then checks if the value we are looking for was found at the midpoint, then has either found it or not, if it has found it it stops. otherwise it discards one half of the array, and adjusts either the high or the low pointer. remark that even in the worst case, we're still discarding half of the array, each each iteration. So very, very quickly, we're going to run out of a range check. so if you do the math, the worst case is you will need to do exactly log base 2 of N iterations, meaning that the binary search will run in logarithmic time. V cool V powerful algorithm.

(go to 14 mins 16 secs future me who needs to write and read all this again :/)

since

Revision #2

Created 25 March 2024 11:18:47 by naruzkurai

Updated 25 March 2024 11:41:41 by naruzkurai