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

this is Quadratic because there's ^3 then ^2 then ^1 or a regular value at the end
O(f(n)) = O(n3)

The following run in constant time: O(1)

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

`:=` 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)

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

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 implicity declaired as 0 unless previously
stated. (it hasnt)
    for j in range(n):  # j goes from 0 to n-j. again implicitly declairing the value of j
        pass  # replace with whatever ud 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 ammount 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_ + ... + 3 + 2 + 1 ??

Remarkabley this turns out to be n(n+1)/2, so
O(n(n+1)/2 = O(((n^2) / 2) + n/2 = (0(n2)

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 t ofind the index of a particular value in the array if it exists. waht is the time complexity of the following algorithm?

heres 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: 0(log_2(n)) = 0(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 ajusts 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 2024-03-25 11:18:47 UTC by naruzkurai
Updated 2024-03-25 11:41:41 UTC by naruzkurai