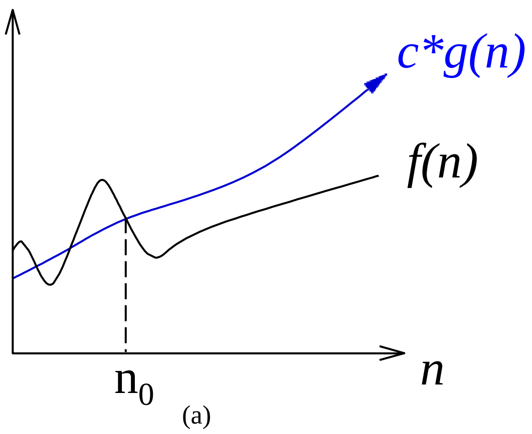
Exploring Big-Oh

Lecture 01.03

by Marina Barsky

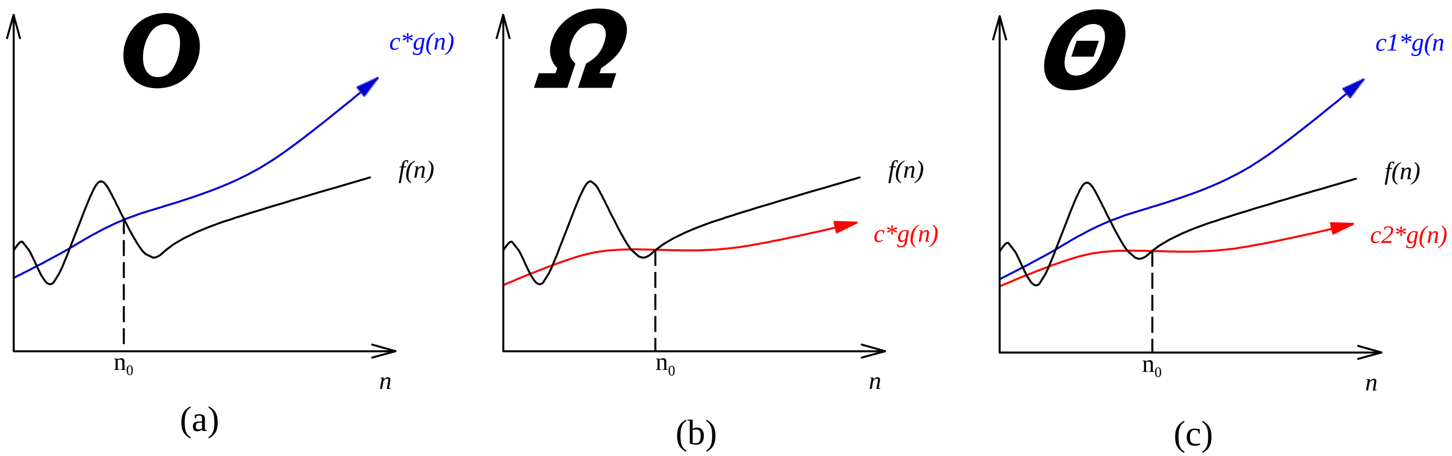
[Big Oh formally]

$f(n) = \boldsymbol{O}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 the value of $f(n)$ always lies on or below $c \cdot g(n)$



For Big-O Notation analysis, we care more about the part that grows fastest as the input grows, because everything else is quickly eclipsed as n gets very large

[Why Big Oh – and not Big Theta]



- *Theta* represents a tight bound on the performance of the algorithm – it is the best characteristic of the running time.
- BUT: It is not easy to find a single bounding function *g(n)* that bounds *f(n)* both from above and from below for all possible inputs:

  For example: bubble sort is not always >= c_2 n^2, that is $f(n) \neq \Theta(n^2)$
- It is easier to give an upper bound, which might not always be tight, but is easier to find.

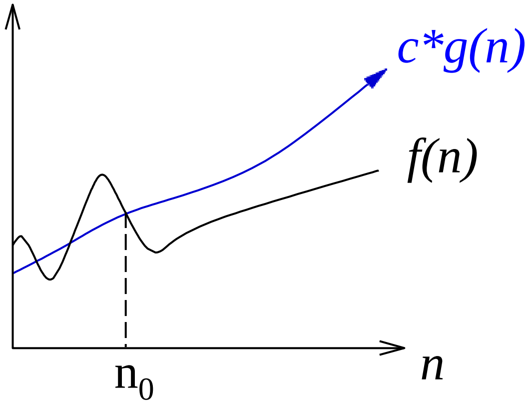
[Big Oh –the rate of growth]

- We use Big O Notation to talk about **how quickly the runtime grows**
- Big O guarantees that for a given input size *n* the algorithm never exceeds the value some function on *n*
- Big O bounds the speed of growth from above:

  so we can say things like the runtime grows "on the order of the size of the input" (O(n)) or "on the order of the square of the size of the input" ($O(n^2)$)

Big Oh – in practice

$f(n) = \boldsymbol{O}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 the value of $f(n)$ always lies on or below $c \cdot g(n)$

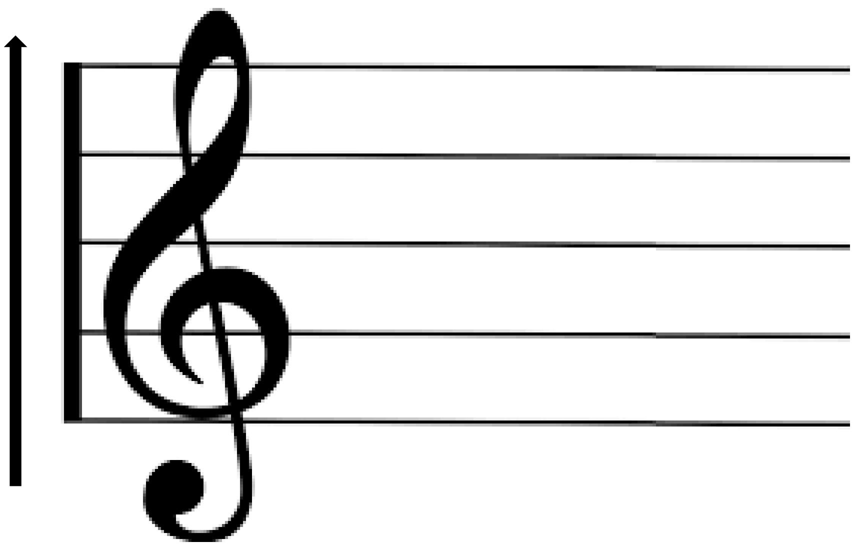


Big-oh is an upper bound that does two things:

- Removes lower order (ie slower growing) terms.
- Removes constant factors.

Example: Let's show that $f(n) = ½ n^2 + 3n \leq cn^2$
Divide both sides by n^2
$½ + 3/n \leq c$
Then, starting with $n_0=6$ and any $c \geq 1$, $f(n) \leq cn^2$
Let $c=2$, then $f(n) < 2n^2$ for any $n>6$, $f(n) = O(n^2)$

Classifying algorithms with Big-Oh



Doubly-Exponential Functions: 2^{2^n}

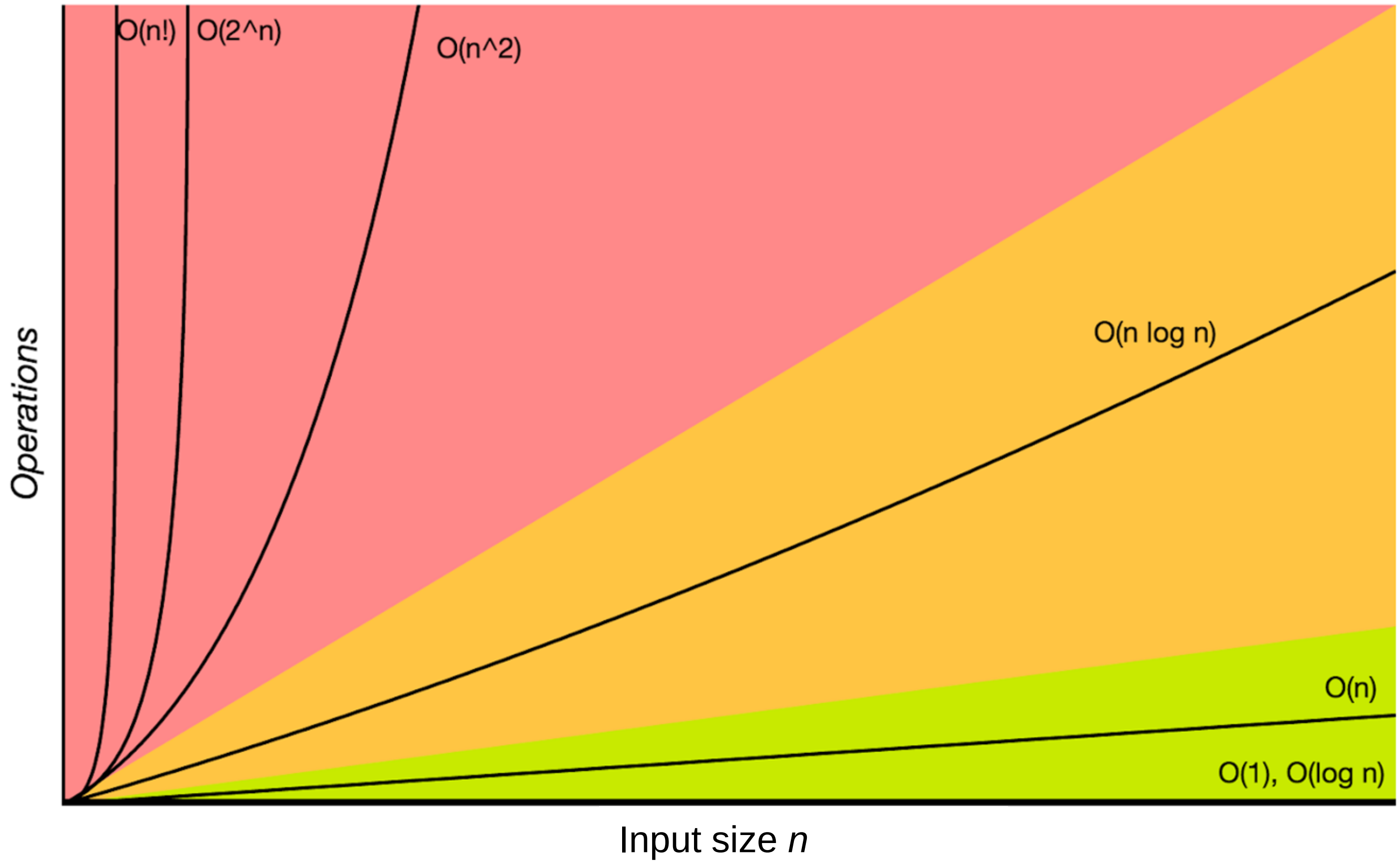
Exponential Functions: 2^n, 3^n, $n \cdot 2^n$

Polynomial Functions: n, n^2, n^3, $n^2 \cdot \log(n)$, $\sqrt{n} = n^{0.5}$

Logarithmic Functions: $\log(n) = \log_2(n)$, $\log_3(n)$

Doubly-Logarithmic Functions: $\log \log n = \log_2(\log_2(n))$

Big Oh matters



n bytes	log n	n	n^2	2^n
10 B	1	10	100	$\sim 1*10^3$
100 B	2	100	10000	$\sim 1*10^{30}$
1 KB	3	1,000	1000000	$\sim 1*10^{300}$
10 KB	4	10,000	100000000	$\sim 1*10^{3000}$
100 KB	5	100,000	10000000000	$\sim 1*10^{30,000}$
1 MB	6	1,000,000	1.00E+12	$\sim 1*10^{300,000}$
10 MB	7	10,000,000	1.00E+14	n/a
100 MB	8	100,000,000	1.00E+16	n/a
1 GB	9	1,000,000,000	1.00E+18	n/a
10 GB	10	10,000,000,000	1.00E+20	n/a
100 GB	11	100,000,000,000	1.00E+22	n/a
1 TB	12	1,000,000,000,000	1.00E+24	n/a

CPU with a clock speed of 2 gigahertz (GHz) can carry out two thousand million (**$2*10^9$**) cycles (operations) **per second**.

- Algorithm which runs in **$O(2^n)$** time will process **1 KB** of input in **$\sim 10^{22}$ years** (more than 7 millenia)
- Processing **1 GB** of input will take **<0.001 ms** by O(log n) algorithm, **< 1 sec** by **O(n)** algorithm, and **>32 years** by **$O(n^2)$** algorithm

Reasoning about time complexity

- When you *intuitively* understand an algorithm, the reasoning about the run-time of an algorithm can be done in your head
- But it is usually much easier to estimate complexity given a **precise-enough pseudocode**

Big Oh: Multiplication by Constant

Multiplication by a constant does not change Big Oh:

The “old constant” C from the Big Oh becomes $c \cdot C$

$$O(c \cdot f(n)) \rightarrow O(f(n))$$

Big Oh: Multiplication by Function

- But when both functions in a product depend on *n*, both are important
- This is why the running time of two nested loops is $O(n^2)$.

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$

Loops

The running time of a loop is, at most, the running time of the statements inside the loop (including if tests) multiplied by the number of iterations.

```
m:= 0
for i from 1 to n:                  #repeat n times
    m:= m + 2                       #constant time c
```

Total time = constant c × n = c n = O(n).

Nested loops

Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
for i from 1 to n:             # outer loop - n times
      for j from 1 to n:       # inner loop - n times
                  k:= k+1      # constant time
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

Consecutive statements

Add the time complexity of each statement.

```
x:= x + 1                          # constant time
for i from 1 to n:                 # executes n times
      m:= m+2                      # constant time

for i from 1 to n:                 # outer loop - n times
      for j from 1 to n:           # inner loop - n times
            k:= k+1                # constant time
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

If-then-else statements

Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
if len(t) = 0:                       # test: constant
      return false                   # then part: constant c0
else:
      for n from 0 to len(t):        # else part: (c1+c2)*n
            if t[n] = p[n]:          # if: c1 + c2 (no else)
                  return false
      return true
```

Total time = $c_0 + (c_1 + c_2) * n = O(n)$.

Logarithmic complexity

An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
i:= 1
while i<=n:
      i:= i*2
```

- If we observe carefully, the value of *i* is doubling every time: Initially i = 1, in next step i = 2, and in subsequent steps i = 4, 8 and so on

Logarithmic complexity

An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
i:= 1
while i<=n:
      i:= i*2
```

- Let us assume that the loop is executing some *k* times - before i becomes > n
- At k-th step $\mathbf{2^k = n}$, and at (*k* + 1)-th step we come out of the loop
- Taking logarithm on both sides:

$\log(2^k) = \log n$

$k \log 2 = \log n$

$\mathbf{k = \log n}$

Logarithmic complexity

The same logic holds for the decreasing sequence as well:

```
i:= n
while i >= 1:
      i:= i/2
```

Example: ***binary search*** (finding a word in a sorted list of size *n*)

- Look at the center point in the sorted list
- Is the word towards the left or right of center?
- Repeat the process with the left or right part of the list until the word is found.

Commonly used Logarithm Rules

Rule or special case	Formula
Product	$\log(xy)=\log(x)+\log(y)$
Quotient	$\log(x/y)=\log(x)-\log(y)$
Log of power	$\log(x^y)=y\log(x)$
Log of one	$\log(1)=0$
Log reciprocal	$\log(1/x)=-\log(x)$
Changing base	$\log_{10}(x) = \log_2(x)/\log_2(10)$

Constant.

Base of the logarithm does not matter in complexity analysis!

Commonly used summations

Arithmetic series

$$\sum_{k=1}^{n} k = 1 + 2 + \ \ldots + n = \frac{n(n-1)}{2} = O(n^2)$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + \ x^2 + \cdots + x^n = \frac{x^{n+1}-1}{x-1} = O(x^{n+1} - 1) = O(x^n), x \neq 1$$

x is a constant, for example 2.
If $x < 1$, then the above sum = 1/(1-x) ≤ 2 = O(1).

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} \ldots + \frac{1}{n} = O(\log n)$$

Example: reasoning about complexity

```
Algorithm2(n)

i ← 1
s ← 1

while s <= n:
        i ← i + 1
        s ← s + i
```

- *i* is going through 1,2,3 …
- Our goal is to determine how many times *i* should increase until *s* hits *n:* let's call this number *k*
- *s* on the other hand contains a sum of $1 + 2 + 3 + \ldots k = O(k^2)$
- So when $k^2 = n$ the loop stops
- Thus after $k=\sqrt{n}$ steps the algorithm terminates → the complexity of the algorithm is $O(\sqrt{n})$

Real-life performance

- How do we compare algorithms which belong to the same big-Oh class?
- Some of them may contain a very large constant: but we already got rid of all constants in our analysis
- Some of the algorithms may use a faulty data structure:

  an example would be an ancient version of the Sieve of Eratosthenes, where we removed an element from the middle of the list: expensive operation
- The implementation quality and the programming language also matter:

  good implementation can make an algorithm run for up to 1000 times faster for the same input
- For these reasons, we run comparative performance tests

Class activity 4

Big O