Efficiency & Big-O Runtime

15-112 2/26/19

Big Ideas

Why do we care about efficiency?

Facets to Consider

Time

Space

Bandwidth

...

Time varies based on processing power, input size, random factors…

To smooth out these factors, consider **how the time changes as the size of the input changes**.

Or, more specifically: **how does the amount of work done change as the size of the input grows?**

Measuring the Work Done

We can measure work done on an input by counting **how many algorithmic steps a program takes.**

Algorithmic step: a simple action done in a program, like adding two numbers. (This is an approximation, but good enough for 112)

How do we choose the input? In 112, try to consider the **worst-case input** for a given program, an input that leads to the most possible work being done.

Big-O Notation

Big-O Definition

A mathematical system to demonstrate how a function's **number of algorithmic steps** grows as its **input size** grows.

Input size: N, where N is the length of a list/string, or an integer.

Example:

```
def addOneToEach(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] + 1

# Operations: 1 + 3*N
# O(3N + 1)
```

Simplifying Big-O

We only care about the part of the Big-O equation that has the **largest impact.** Therefore, we only consider the **highest-order term** of the equation. In O(3N + 1), that's 3N.

Additionally, we only care about how the equation **grows with the input.** Therefore, we remove all constant factors- O(3N) becomes O(N).

Examples:

O(3N^2 - 2N + 25) -> O(N^2)

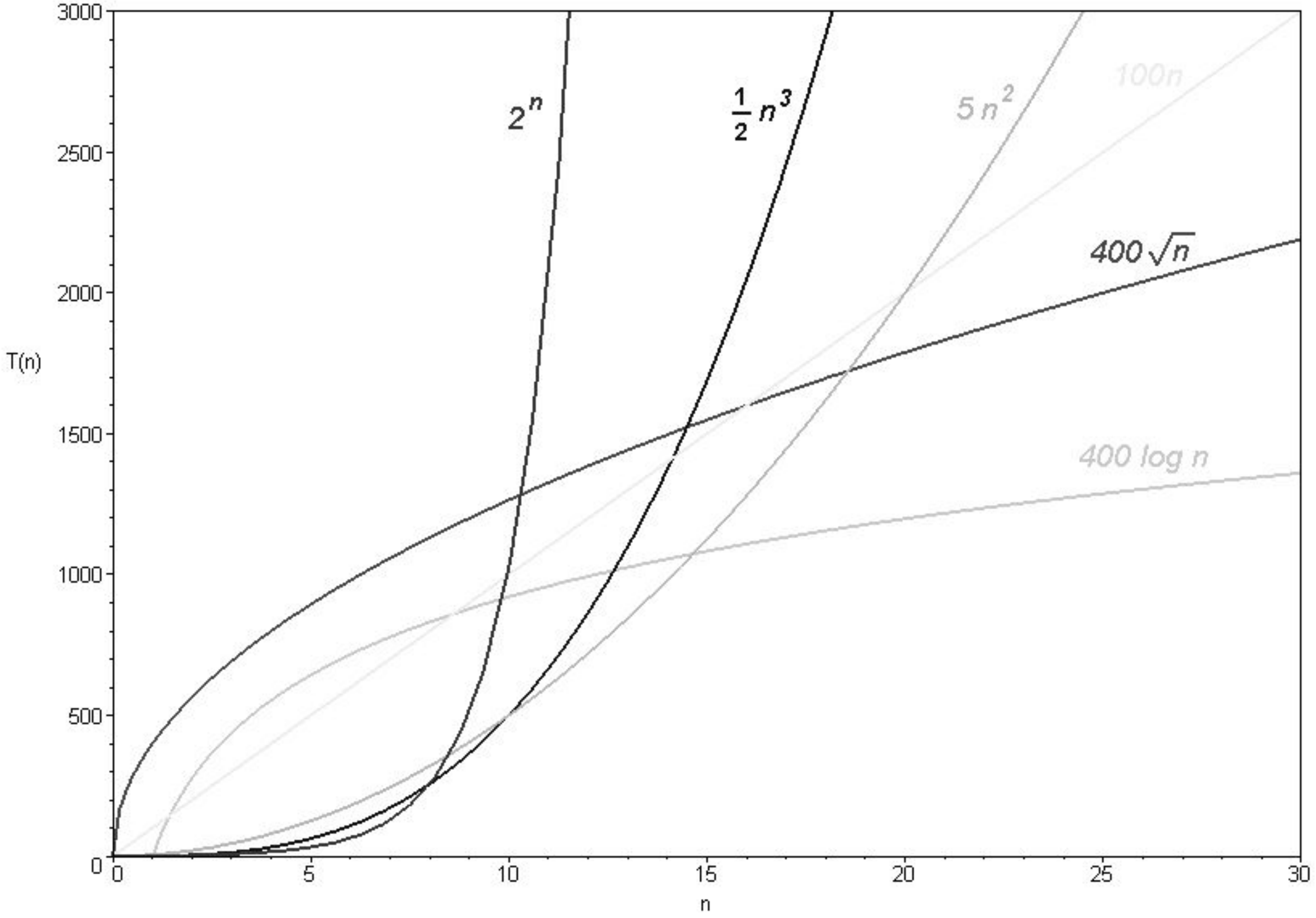
O(0.00000000001N + 123456789) -> O(N)

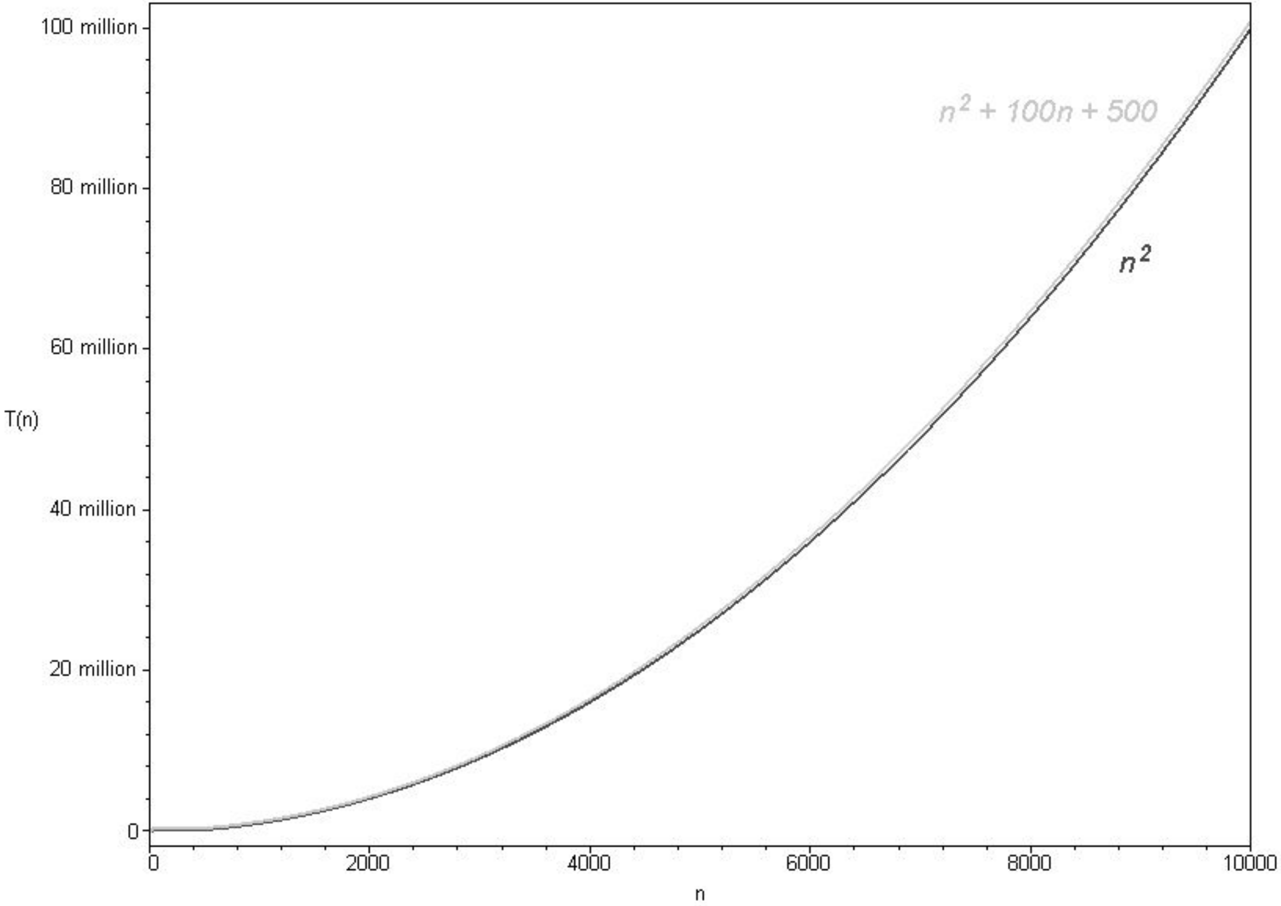
Function Families

Simplifying Big-O equations lets us consider primarily which **function family** a program belongs to.

Grows **slowly**: Contant [O(1)], Logarithmic [O(logN)], Linear [O(N)]

Grows **quickly**: Quadratic [O(N^2)], Polynomial [O(N^K)], Exponential [O(K^N)]


3000
2500
2000
1500
1000
500
0
T(n)
0
5
10
15
20
25
30
n
2^n
$\frac{1}{2}n^3$
$5n^2$
$100n$
$400\sqrt{n}$
$400 \log n$


100 million
80 million
60 million
40 million
20 million
0
T(n)
0
2000
4000
6000
8000
10000
n
$n^2 + 100n + 500$
n^2

Calculating Big-O

Big-O of a Statement

To find the Big-O class of a single Python statement, determine how many **algorithmic steps** it takes.

Some built-in functions take multiple steps. You can find the mapping of function to number of steps here: https://www.cs.cmu.edu/~112/notes/notes-efficiency-builtin-runtime-table.html

Example:

```
# len(lst) = N
2 + lst.count("foo")
```

1 operation for addition + N operations for .count -> O(N)

Big-O of a Sequence of Statements

When computing the Big-O of a sequence of statements, **add** the individual Big-Os together. The highest-order term wins.

Addition should also be used to combine multiple actions in a single statement.

Example:

```
# len(lst) = N
L.sort()
L.sort(reverse=True)
L[0] -= 5
print(L.count(L[0]) + sum(L))
```

O(NlogN) + O(NlogN) + O(1) + O(N) + O(N) -> O(NlogN)

Big-O of Nested Statements - Loops

When determining the Big-O of a loop, consider **how many times the loop iterates**. Then **multiply** that # of iterations by the Big-O of the loop's body.

In a for loop, the values in the range/iterator will be computed **once.**

In a while loop, the values in the condition check are computed in **each loop**.

Example:

```
for c in L:
    L[0] += c
    L.sort()
print(L)
```

N * (O(1) + O(NlogN)) + O(N) -> O(N^2 logN)

Big-O of Nested Statements - Conditionals

When determining the Big-O of an if statement, add the **condition check** (which happens once), then determine logically whether the Big-O of the **body** should be added.

If statements, though nested, are **sequential!**

Example:

```
# len(lst) = N
if len(lst) == -1:
    lst.sort()
else:
    lst.append(4)
```

O(1) + O(1) -> O(1)

Big-O of Composed Statements

When functions are **composed,** pay attention to how the size of the input changes based on the function calls.

This can also happen by changing an input inside a function- always pay attention to the input size!

Example:

```
def f(L):
    L1 = sorted(L)  #sorting->NlogN
    return L1
def g(L):
    L1 = L * len(L)
    return L1
result = f(g(L)) # len(L) = N
```

O(N^2) + O(N^2 logN^2) -> O(N^2 logN)

Let’s Practice!

What’s the Big-O?

```
def f(lst):
    if len(lst) > 0:
        return lst[0]
    else:
        return None
```

What’s the Big-O?

```
def f(n):
    result = 0
    i = n
    while i > 0:
        result = result + i
        i = i // 10
    return result
```

What’s the Big-O?

```
def f(lst):
    result = 0
    for i in range(len(lst)):
        result += lst[i]
    return result
```

What’s the Big-O?

```
def f(lst):
    result = True
    for i in range(len(lst)):
        for j in range(i+1, len(lst)):
            if lst[i] == lst[j]:
                result = False
    return result
```

Checking work: time.time()

```
def f(lst):
    result = True
    for i in range(len(lst)):
        for j in range(i+1, len(lst)):
            if lst[i] == lst[j]:
                result = False
    return result

import time
n = 1000
lst1 = [42] * n
lst2 = [42] * (10 * n)
t1 = time.time()
f(lst1)
t2 = time.time()

t3 = time.time()
f(lst2)
t4 = time.time()
print("Time of N: " + str(t2 - t1))
print("Time of 10*N: " + str(t4 - t3))
print("Ratio:" + str((t4-t3)/(t2-t1)))
```

You do!

```
def foo(s): #s is a string of length N
    result = 0
    for char in string.ascii_lowercase:
        if char in s:
            s = s[1:]
            result += 1
    return result
```