Runtime Analysis

Problem Motivation

- Code can take an unfeasibly long time to run
- Code can require more memory than available

→ Analyze the “runtime” of your code to make it more efficient

Example 1

How long does this take to run?

```
for number in xrange(n):
    print number
```

Runtime is expressed as a function of “n”--the size of the code’s input.

Example 2

How long does this take to run?

```
for number1 in xrange(n):
    for number2 in xrange(n):
        print number1, number2
```

Example 3

How long does this take to run?

```
def find_anagrams(lst):
    result = []
    for word1 in lst:
        for word2 in lst:
            if word1 != word2:
                for perm in permutations(word1):
                    if perm == list(word2):
                        result.append(word1)
    return result
```

Example 3

How long does this take to run?

```
def find_anagrams(lst):
    result = []                                       1 step
    for word1 in lst:                                 n step
        for word2 in lst:                             n steps
            if word1 != word2:                        k steps
                for perm in permutations(word1):      k! steps
                    if perm == list(word2):           k steps
                        result.append(word1)          1 steps
    return result
```

$$1 + n(n(k + k!(k + 1))) = 1 + n^2k + n^2k!k + n^2k!$$

Example 3

Is this a useful expression?

$$1 + n^2k + n^2k!k + n^2k!$$

(n, k) = (10, 5) → steps ~= 1 + 5E2 + **6E5** + 12E4

(n, k) = (100, 10) → steps ~= 1 + 1E5 + **4E11** + 4E10

→ **With large inputs, only the largest terms of the expression are relevant.**

Simplifying Runtime Analysis: “Big O”

- Often, a single term dominates the runtime expression
- Runtime analysis can be simplified by eliminating lower order terms
- Big O notation provides a principled way to do this

Big O Definition

Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \mathcal{O}(g(n))$$

(read: f is Big-"O" of g) if there exists a constant $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

Image credit: Christopher M. Bourke 2006

Big O Intuitive Summary

- $f = O(g) \rightarrow$ “f is bounded by g” (roughly speaking)
  - f is bounded by some constant multiple of g
  - Only true for sufficiently large input values
- Practical runtime analysis only cares about large input values and approximate bounds

Big O Conventions

Be Precise

- Technically, if a function is O(n), then it is also O(n^2)
- But, a more precise upper bound is more useful
- Therefore, give the lowest upper bound possible
  - I.e. don't say O(n^2) when you could instead say O(n)

Ignore Constants

- If a function is O(n), then it is also O(2*n)
- Saying O(2*n) is unnecessary and highly unconventional

Big O Example

What’s the big O analysis of this expression? (from earlier anagrams example)

$$1 + n^2k + n^2k!k + n^2k!$$

Example 4

How long does this take to run?

```
def find_anagrams(lst):
    result = []
    d = defaultdict(list)
    for word in lst:
        d[tuple(sorted(word))].append(word)
    for key, value in d.iteritems():
        if len(value) > 1:
            result.extend(value)
    return result
```

Example 4

How long does this take to run?

`def find_anagrams(lst):`	
`    result = []`	O(1)
`    d = defaultdict(list)`	O(1)
`    for word in lst:`	O(n)
`        d[tuple(sorted(word))].append(word)`	O(k*log(k))
`    for key, value in d.iteritems():`	O(n)
`        if len(value) > 1:`	O(1)
`            result.extend(value)`	O(n)
`    return result`	

Common Runtimes to Remember

O(1) - constant (practically no time at all)
O(n) - linear
O(n*log(n)) - "n log n" (typical sorting runtime)
O(n^2) - quadratic
O(n^3) - cubic
O(2^n) - exponential (base is not necessarily 2)
O(n!) - factorial

Runtime of Typical Python Functions

List operations

- Appending: O(1)
- Adding to the beginning or middle: O(n) (have to slide everything over!)
- Popping from the end: O(1)
- Popping from the beginning or middle: O(n)
- Looking up by index: O(1)
- Searching: O(n) (have to look at every item)
- Searching a sorted list: O(log n) (binary search)

https://wiki.python.org/moin/TimeComplexity

Runtime of Typical Python Functions

Dictionary operations

- Inserting an item: O(1)
- Removing an item: O(1)
- Looking up by key: O(1)
- Looking up by value: O(n) (have to look at every item)

https://wiki.python.org/moin/TimeComplexity

Runtime in Practical Terms

n	Linear (n) runtime in seconds	Quadratic (n^2) runtime in seconds
100	1 sec	1 sec
1000	10 sec	100 sec
10,000	100 sec	10,000 sec = 167 min
100,000	1000 sec = 17 min	1,000,000 sec = 11 days

Quadratic (n^2) algorithms are *really slow*