Runtime (and Memory) Analysis

Problem

- ► Code can take an unfeasibly long time to run.
- ► (Code can require more memory than available.)

Solution

- ▶ Analyze the runtime of your code to make it more efficient.
- ► (Analyze the memory requirement of your code to make it more efficient.)

How long does this take to run?

```
for i in range(n):
    print(i)
```

Runtime is expressed as a function of n, the size of the code's input.

Example 1 We'll assume a unit of execution: step to simplify things. sometime is spend on the initialization and wrap up: for i in range(n): print(i) getting the next element with 1+2n steps

```
for i in range(n): # 1 + n steps
    print(i) # n steps
```

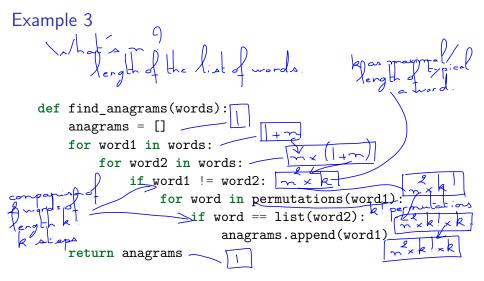
1 step + 2n steps $\approx 2n$ steps for large value of nbut Joes the 2 really matter of steps ?

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

```
for i in xrange(n): # 1 + n steps
for j in xrange(n): # n * (1 + n) steps
print(i, j) # n \ge steps
```

1 step + 2n steps $+ 2n^2$ steps $\approx 2n^2$ steps for large value of n

```
How long does this take to run?
def find_anagrams(words):
    anagrams = []
    for word1 in words:
        for word2 in words:
            if word1 != word2:
                 for word in permutations(word1):
                     if word == list(word2):
                         anagrams.append(word1)
    return anagrams
```



$$\frac{3}{2}$$
 + 2n + n² + n²k + n²k! + 2n²(k + 1)! steps

Is this a useful expression?

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

$$ightharpoonup$$
 (n, k) = (10, 5)

$$ightharpoonup$$
 (n, k) = (100, 10)

$$ightharpoonup$$
 2 + 200 + 1E4 + 1E5 + 4E10 + **8E11**

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

Simplifying Runtime Analysis: Big O Notation

- ▶ 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: $\mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in O(g(n))$$

(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 $n \geq n_0$,

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

Big O Intuitive Summary

$$f = O(g)$$

"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(2n).
- ▶ Saying O(2n) is highly unconventional.

Big O Example

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

$$2 + 2n + n^{2} + n^{2}k + n^{2}k! + 2n^{2}(k+1)! \text{ steps}$$

$$2 + 2n + n^{2} + n^{2}k + n^{2}k! + 2n^{2}(k+1)! + \frac{1}{(k+1)!} + \frac{1}{2(k+1)!} + \frac{2}{2(k+1)!} + \frac{2}{2(k+1)!}$$

Big O Example

```
How long does this take to run?
def find_anagrams(words):
    anagrams = []
    d = defaultdict(list)
    for word in words:
        d[tuple(sorted(word))].append(word)
    for _, words in d.items():
        if len(words) > 1:
            anagrams.extend(words)
    return anagrams
```

```
def find_anagrams(words):
   anagrams = []
   d = defaultdict(list) O(1)
for word in words: O(~)
      d[tuple(sorted(word))].append(word);
      _, words in d.items():
       f len(words) > 1:
          anagrams.extend(words)
        return anagrams
```

```
def find_anagrams(words):
 anagrams = []
                                        # 0(1)
 d = defaultdict(list)
                                        # 0(1)
 for word in words:
                                        \# O(n)
  d[tuple(sorted(word))].append(word) # O(n * k * log(k))
  # line is executed n times; then in series,
  \# - sorted() is O(k * log(k))
  # - d / 7 is O(1)
  \# - .append() is O(1)
 for _, words in d.items():
                                        # υ
  if len(words) > 1:
                                        # υ
   anagrams.extend(words)
                                        \# O(n)
 return anagrams
```

Common runtimes to remember

```
▶ O(1) - constant

▶ O(n) - linear

▶ O(nlog(n)) - "n log n" - e.g. Sorting

▶ O(n^2) - quadratic

▶ O(n^3) - cubic - e.g. matrix multiplication

▶ O(2^n) - exponential (base is not necessarily 2)

▶ O(n!) - factorial
```

Runtime in practical terms

n	Linear runtime	Quadratic runtime	
100	1 s	1s	
1000	10 s	100 s	
10,000	100 s	$10,000 \; \mathrm{s} = 167 \; \mathrm{min}$	
100,000	$1000\;\mathrm{s}=17\;\mathrm{min}$	$1,000,000 \; \mathrm{s} = 11 \; \mathrm{days}$	

Quadratic (n^2) algorithms are already really slow.

Linear (n) or nlog(n) are ideal; anything bigger is usually too slow for large data.

How much memory does this take to run?

for in range(n):

print(i)

Grange(n):

Thom 3:

generator O(1)

Memory is expressed as a function of n, the size of the code's input.

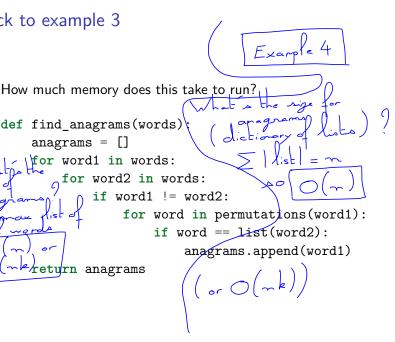
```
How much memory does this take to run?

for i in (range(n): list of generators)

print(i, j)

print(i, j)

one for j exist.
```



def find_anagrams(words) anagrams = for word1 in words: eturn anagrams

```
How long does this take to run?
def find_anagrams(words):
    anagrams = []
    d = defaultdict(list)
    for word in words:
        d[tuple(sorted(word))].append(word)
    for _, words in d.items():
        if len(words) > 1:
            anagrams.extend(words)
    return anagrams
```

Important data structures

- Lists
- Dictionaries
- ► Graphs
 - ► Trees

Runtime of typical Python list functions

Appending (1)
Adding to the beginning or middle (n)
Popping from the end (1)
Popping from the beginning or middle (n)
Looking up by index (1)
Searching an unsorted list (log n)
Searching a sorted list (log n)

Runtime of typical Python list functions (Time Complexity)

- ► Appending:
 - ► *O*(1)
- ▶ Adding to the beginning or middle:
 - ightharpoonup O(n) (have to shift elements over!)
- ▶ Popping from the end:
 - ▶ O(1)
- Popping from the beginning or middle:
 - ightharpoonup O(n) (again, have to shift elements over!)
- Looking up by index:
 - ► O(1)
- Searching an unsorted list:
 - ► 3 O(n) (have to look at every item)
- Searching a sorted list:
 - ► O(logn) (binary search)



Runtime of typical Python dictionary functions

```
► Inserting an item
► Removing an item
► Looking up by key
► Looking up by value
```

Runtime of typical Python dictionary functions (Time Complexity)

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

More

Algorithms Notes for Professionals book

- Chapter 2: Algorithm Complexity
- ► Chapter 22: Big-O Notation

Important algorithms: Sorting

Algorithm	Best	Average	Worst	Memory
Bubble sort	n	n^2	n^2	1
Selection sort	n^2	n^2	n^2	1
Insertion sort	n2	n^2	n^2	1
Merge sort	nlog(n)	nlog(n)	nlog(n)	n (worst)
In-place	-	-	$n(\log(n))^2$	1
merge sort				
Quicksort	nlog(n)	nlog(n)	n^2	logn (average) n (worst)
Heapsort	nlog(n)	nlog(n)	nlog(n)	1

Important algorithms: Sorting

Algorithms Notes for Professionals book

- Chapters 23: Sorting
 - Chapters 24: Bubble Sort
 - ► Chapters 25: Merge Sort
 - ► Chapters 26: Insertion Sort
 - ► Chapters 27: Bucket Sort
 - Chapters 28: Quick Sort
 - ▶ Chapters 29: Counting Sort
 - Chapters 30: Heap Sort
 - ► Chapters 31: Cycle Sort
 - Chapters 32: Odd-Even Sort
 - ► Chapters 33: Selection Sort

Other important algorithms

Algorithms Notes for Professionals book

- Chapter 3: Graph
 - Chapter 4: Graph Traversals
 - Chapter 5: Dijkstra's Algorithm
- Chapter 34: Trees
 - Chapter 35: Binary Search Trees
 - Chapter 37: Binary Tree traversals
 - Chapter 41: Breadth-First Search
 - Chapter 41: Depth First Search