CME 211 Lecture 4: Functions and Complexity Analysis

We study the complexity of algorithms to understand the resources required to run a program.

Analysis of algorithms

Key questions when considering the performance of algorithms:

- **Time/computational complexity**: How does the number of computational operations in an algorithm scale with the size of the input?
- **Space complexity**: How do the storage (memory) requirements of the algorithm scale?
- **Communication complexity**: Many modern applications in high performance computing are limited by memory communication bandwidth and/or latency. This can be a challenging area of study, because there are many types of communication to consider (e.g. main memory to CPU, CPU to GPU, GPU memory to GPU, or computer to computer over a network). Also modern computing hardware has many levels of caching (e.g. L1, L2, and L3), making it difficult to to predict the performance of a single memory transaction.

These notes focus on **time complexity**.

Empirical approach

Let's measure the running time of Pythons `sort` method on a random list of integers. See `listsort.py`. Here is the code modified to suit the notebook:

```
import random
import sys
import time
```

```
n = 1000

# Setup a list of random values and record the time required to sort it
v = random.sample(range(n), n)
t0 = time.time()
v.sort()
t1 = time.time()

print("Sorting {} values took {:.3} seconds.".format(n, t1-t0))
```

Collect data

Let's run the script with increasing list length:

```
$ python3 listsort.py
Usage:
  listsort.py nvalues
$ python3 listsort.py 1000000
Sorting 1000000 values took 0.87 seconds.
$ python3 listsort.py 2000000
Sorting 2000000 values took 2.91 seconds.
$ python3 listsort.py 4000000
Sorting 4000000 values took 5.01 seconds.
$ python3 listsort.py 8000000
```

```
Sorting 8000000 values took 10.8 seconds.
$ python3 listsort.py 16000000
Sorting 16000000 values took 23.6 seconds.
```

IPython `%timeit` magic command

IPython has a magic command called `%timeit` to help benchmark Python statements.

```
# use %timeit to benchmark sorted function
# Setup a list of random values and record the time required to sort it
n = 10000
v = random.sample(range(n), n)
%timeit sorted_v = sorted(v)
```

Problems with empirical measurement

Empirical performance testing is an important endeavor. It is an aspect of "profiling" your code to see what parts take longer. Empirical performance testing has some drawbacks, namely:

- Results are computer dependent
- You need to have the code before you can do the analysis. You may spend time implementing something that turns out to be slow

Time complexity

- *Time complexity* is an estimate of the number of operations as a function of the input size (usually denoted as n)
- Input size examples:
- length of list
- for an m by m matrix, we say the input size is m even though the matrix as m^2 entries
- number of non-zero entries in a sparse matrix
- number of nodes in a graph or network structure (sometimes the number of edges is also important)
- Typically characterized in terms of Big O notation, e.g. an algorithm is $O(n \log n)$ or $O(n^2)$.

```
| order notation | in English          |
|----------------+---------------------|
| O(1)           | Constant time       |
| O(log n)       | Logarithmic time    |
| O(n)           | Linear time         |
| O(n log n)     | Linearithmitic time |
| O(n^2)         | Quadratic time      |
| O(n^3)         | Cubic time          |
| O(2^n)         | Exponential time    |
```

Visualization

Big O notation

- Big O notation represents growth rate of a function in the limit of argument going to infinity

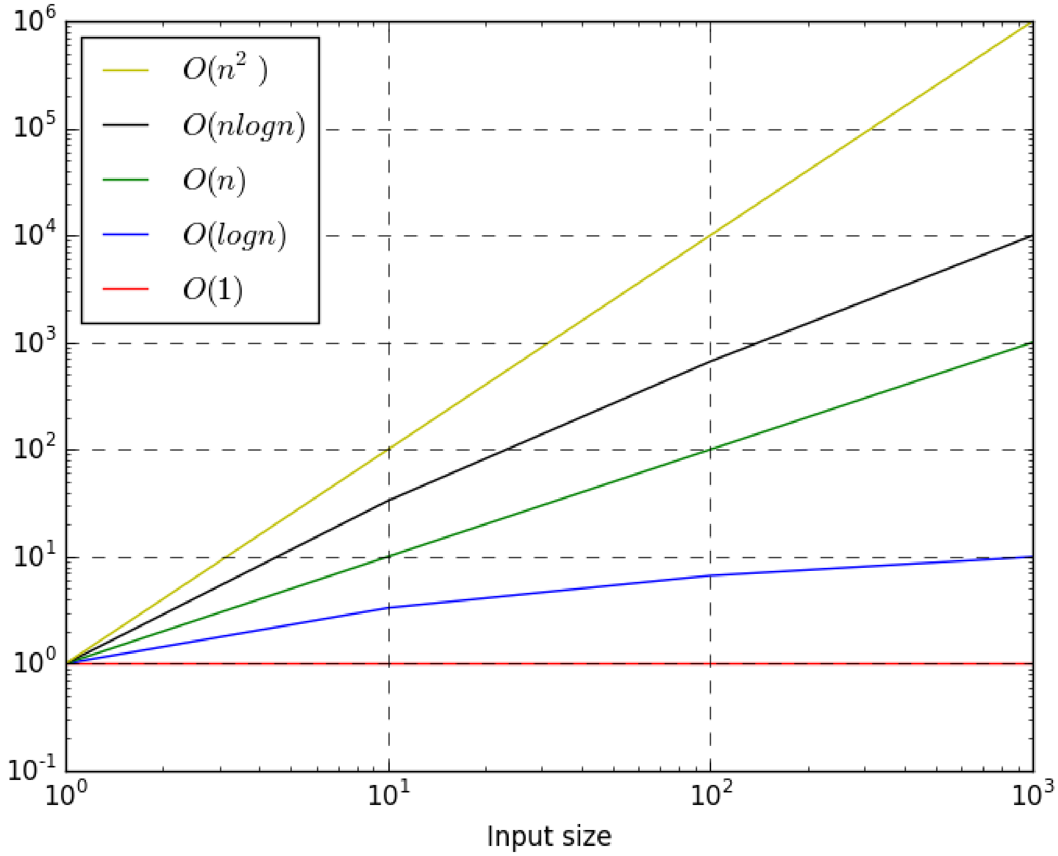


Figure 1: order chart

- Excludes coefficients and lower order terms

$$2n^2 + 64n \rightarrow O(n^2)$$

- Often some simplifying assumptions will need to be made about the nature of the input data in order to carry out analysis

Complexity analysis examples

Basic linear algebra

- Adding two vectors? $O(n)$

```python
# define data
a = [1.0, 2.0, 3.0, 4.0]
b = [1.0, 1.0, 1.0, 1.0]
c = [0.0, 0.0, 0.0, 0.0]

# c = a + b
# assume all the same length
n = len(a)
for i in range(n):
    c[i] = a[i] + b[i]
```

- Multiplying two matrices? Assuming the matrices are both $n \times n$, it's $O(n^3)$

```
# assume all matrices are n x n
# indexing notation below comes from numpy
# this will not work with standard python
# C = A*B
for i in range(n):
    for j in range(n):
        C[i,j] = 0
        for k in range(n):
            C[i,j] += A[i,k]*B[k,j]
```

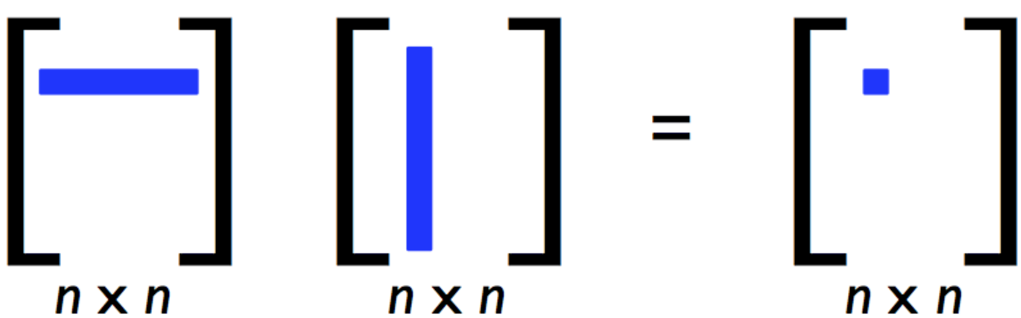


Figure 2: matmul

Computing one value in the output matrix requires $O(n)$ operations, and there are n^2 values in the output matrix.

Linear search

Linear search is searching through a sequential data container for a specified item. An example of this is finding the start index of a given sub-string in a longer string.

Exercise: Find the number x in your data:

```
|---+----+-----+----+-----+----+-----+-----|
| 4 | 17 | 100 | 73 | 120 | 42 | 999 | -17 |
|---+----+-----+----+-----+----+-----+-----|
```

Is it $O(1)$, or $O(n)$, or something else?

Linear search: best and worst case

```
|---+----+-----+----+-----+----+-----+-----|
| 4 | 17 | 100 | 73 | 120 | 42 | 999 | -17 |
|---+----+-----+----+-----+----+-----+-----|

  ^                                     ^
  |                                     |
 O(1)                                  O(n)
```

- Best case: `x = 4` and we find the index with only one comparison
- Worst case: `x = -17` and we scan the entire list to find the last element

Linear search: average case

```
|---+----+-----+----+-----+----+-----+-----|
| 4 | 17 | 100 | 73 | 120 | 42 | 999 | -17 |
|---+----+-----+----+-----+----+-----+-----|

                    ^
                    |
                  O(n/2)
```

Given random data and a random input (in the range of the data) we can **on average** expect to search through half of the list. This would be $O(n/2)$. Remember that Big O notation is not concerned with constant terms, so this becomes $O(n)$.

Binary search

If we know that the list is sorted, we can apply binary search. Let's look at an example:

Goal: Find the index of `17` in the following list:

```
|-----+---+----+----+----+-----+-----+-----|
| -17 | 4 | 17 | 42 | 73 | 100 | 120 | 999 |
|-----+---+----+----+----+-----+-----+-----|
```

Start by looking half way through the list:

```
|-----+---+----+----+----+-----+-----+-----|
| -17 | 4 | 17 | 42 | 73 | 100 | 120 | 999 |
|-----+---+----+----+----+-----+-----+-----|
                  ^
                  U
```

42 is not 17, but 42 is greater than 17 so continue searching the left (lower) part of the list. The index associated with 42 becomes an upper bound on the search.

```
|-----+---+----+----+----+-----+-----+-----|
| -17 | 4 | 17 | 42 | 73 | 100 | 120 | 999 |
|-----+---+----+----+----+-----+-----+-----|
        ^         ^
        L         U
```

4 is not 17, but 4 is less than 17 so continue searching to the right part of the list up to the upper bound. Turns out in this example that we only have one entry to inspect:

```
|-----+---+----+----+----+-----+-----+-----|
| -17 | 4 | 17 | 42 | 73 | 100 | 120 | 999 |
|-----+---+----+----+----+-----+-----+-----|
        ^   ^     ^
        L   *     U
```

We have found 17. It is time to celebrate and return the index of 2. (Remember Python uses 0-based indexing.)

Binary search: analysis For illustration, lets say we have a list with 16 elements. Each iteration of binary search cuts the list in half.

- Start with 16 elements
- Iteration 1: cut in half: 8 elements
- Iteration 2: cut in half: 4 elements
- Iteration 3: cut in half: 2 elements
- Iteration 4: cut in half: 1 element

(The algorithm would stop early if it finds the element, let's assume it does not until the very end.)

In each iteration we do a single comparison and update one index, that is $O(1)$ work. So the main question is how many iterations. In the above example we had an input size of $n = 16 = 2^4$. It's the power of 2 that determines how many operations must be performed in binary search. Thus the number of operations for binary search is proportional to $\log_2 n$ where n is the input size. Thus, we say that binary search has time complexity of $O(\log n)$.

Note that the base of the logarithm is not important because

$$\log_2 n = \frac{\log_{10} n}{\log_{10} 2}$$

and $\log_{10} 2$ is constant (independent of n).

Summary: Binary search

- Requires that the data first be sorted, but then:
- Best case: $O(1)$
- Average case: $O(\log n)$
- Worst case: $O(\log n)$

Sorting algorithms

There are many sorting algorithms and this is a worthy area of study. Here are few examples of names of sorting algorithms:

- Quicksort
- Merge sort
- Heapsort
- Timsort
- Bubble sort
- Radix sort

The internet is full of examples of how sorting algorithms work

- http://www.youtube.com/watch?v=lyZQPjUT5B4
- http://www.youtube.com/user/AlgoRythmics

Name	Best	Average	Worst	Memory	Stable
Quicksort	*n* log *n*	*n* log *n*	n^2	Average log *n*, worst *n*	Usually not
Merge sort	*n* log *n*	*n* log *n*	*n* log *n*	Worst *n*	Yes
Heapsort	*n* log *n*	*n* log *n*	*n* log *n*	1	No
Bubble sort	*n*	n^2	n^2	1	Yes

Figure 3: sorting algo table

Sorting algorithms reference See: https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

Finding the maximum

What's the order of the algorithm to find the maximum value in an *unordered* list?

```
|----+------+----+-----+----+----+-----+-----+---|
| 17 | 1325 | -3 | 346 | 73 | 19 | 999 | 120 | 0 |
|----+------+----+-----+----+----+-----+-----+---|
```

Idea: let's sort

- Sort the list ascending / descending and take the last / first value
- Cost of the algorithm will be the cost of the sorting plus one more operation to take the last / first value

- Sorting algorithms are typically $O(n \log n)$ or $O(n^2)$
- Overall order of algorithm will clearly be the order of the sorting algorithm

Idea: linear search Algorithm:

- scan through the list sequentially
- keep track of max element seen so far
- compare each element and update if needed

Step 1:

```
|----+------+----+-----+----+----+-----+-----+---|
| 17 | 1325 | -3 | 346 | 73 | 19 | 999 | 120 | 0 |
|----+------+----+-----+----+----+-----+-----+---|
   ^
   |
  17
```

Step 2: move to next element, compare and update

```
|----+------+----+-----+----+----+-----+-----+---|
| 17 | 1325 | -3 | 346 | 73 | 19 | 999 | 120 | 0 |
|----+------+----+-----+----+----+-----+-----+---|
          ^
          |
        1325
```

Repeat:

```
|----+------+----+-----+----+----+-----+-----+---|
| 17 | 1325 | -3 | 346 | 73 | 19 | 999 | 120 | 0 |
|----+------+----+-----+----+----+-----+-----+---|
                ^
                |
              1325
```

And so on:

```
|----+------+----+-----+----+----+-----+-----+---|
| 17 | 1325 | -3 | 346 | 73 | 19 | 999 | 120 | 0 |
|----+------+----+-----+----+----+-----+-----+---|
                                               ^
                                               |
                                             1325
```

Question: what is the order of this algorithm?

Find two largest values

Question: What's the complexity to find the two largest values in an *unordered* list of n values?

Now we need to keep track of two values during the traverse of the list. We will also need to sort the pair of numbers that we keep along the way.

Start by looking at the first two elements:

```
|----+----+-----+-----+----+------+-----+---|
| 17 | 73 | 417 | 346 | 73 | 1325 | 120 | 0 |
|----+----+-----+-----+----+------+-----+---|
```

```
 ^     ^
 |     |
(17,  73)
(73,  17) <- sorted
```

Move down by one:

```
|----+----+-----+-----+----+------+-----+---|
| 17 | 73 | 417 | 346 | 73 | 1325 | 120 | 0 |
|----+----+-----+-----+----+------+-----+---|
       ^     ^
       |     |
      (73,  417)
      (417,  73) <- sorted
```

Repeat:

```
|----+----+-----+-----+----+------+-----+---|
| 17 | 73 | 417 | 346 | 73 | 1325 | 120 | 0 |
|----+----+-----+-----+----+------+-----+---|
            ^      ^
            |      |
           (417,  346)
           (417,  346) <- sorted
```

Repeat (in this case no update is needed):

```
|----+----+-----+-----+----+------+-----+---|
| 17 | 73 | 417 | 346 | 73 | 1325 | 120 | 0 |
|----+----+-----+-----+----+------+-----+---|
                  ^      ^
                  |      |
                 (417,  346)
                 (417,  346) <- sorted
```

Notes:

- For each of n input elements you will do a comparison, potentially a replacement, and a sort
- Time complexity is $O(n)$

Question:

- Does that mean that finding the two largest values will take the same amount of time as finding the single largest value?

m largest values

What if I want to find the m largest values in an unordered list of n elements?

This is an example of a more complicated algorithm. We have two components to consider:

- the length of the list n
- number number of largest values that we want m

Thus, it may not be appropriate to characterize an algorithm in terms of one parameter n.

- Time complexity for finding the m largest values in an unordered list of n elements could be characterized as $O(nm \log m)$ for a sorting algorithm that is $O(m \log m)$

Question:

- For what values of m and n is it better just to sort the list?

Finding sub-strings

Important procedure. We are using it in Homework 1.

Example:

```
TGTAGAATCACTTGAAAGGCGCGCAGTCTGGGGCGCTAGTCGTGGT
          CTTGAAAGG
          ^        ^
          |        |
```

- String has length m, and sub-string has length n
- Different algorithms:
- $O(mn)$ for a naive implementation
- $O(m)$ for typical algorithms
- $O(n)$ for a search that uses the Burrows-Wheeler transform

List operations in Python

```python
a = []
a.append(42)
print(a)
a.insert(0, 7)
print(a)
a.insert(1, 19)
print(a)
```

Python lists use contiguous storage. As we are inserting into the list, the memory layout will look something like:

```
a.append(42)

|----+---+---+---|
| 42 | ? | ? | ? |
|----+---+---+---|

a.insert(0, 7)

|---+----+---+---|
| 7 | 42 | ? | ? |
|---+----+---+---|

a.insert(1, 19)

|---+----+----+---|
| 7 | 19 | 42 | ? |
|---+----+----+---|
```

List vs Set in python

Let's compare Python's `list` and `set` objects for a few operations:

```python
def load_set(filename):
    names_set = set()
    with open(filename,'r') as f:
        for line in f:
            names_set.add(line.split()[0])
    return names_set

def load_list(filename):
    names_list = []
    with open(filename,'r') as f:
        for line in f:
            names_list.append(line.split()[0])
    return names_list
```

```python
names_list = load_list('../lecture-04/dist.female.first')
names_set = load_set('../lecture-04/dist.female.first')
```

Let's test:

```python
'JANE' in names_list
```

```python
'LELAND' in names_list
```

```python
'JANE' in names_set
```

```python
'LELAND' in names_set
```

Which container is better for insertion and existence testing?

Exercise: use IPython's `%timeit` magic command.

Documentation

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

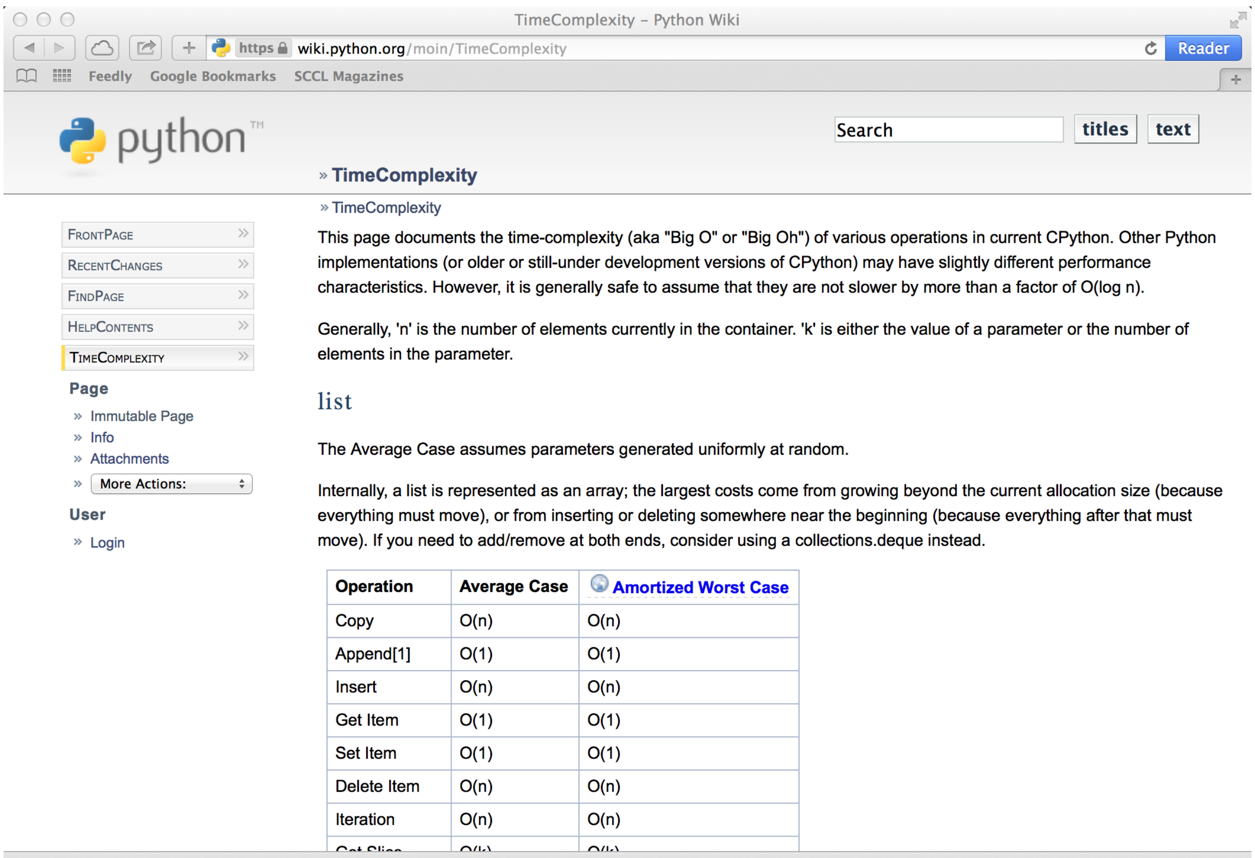
List operations

Set operations

Dictionary operations

Space complexity

- What additional storage will I need during execution of the algorithm?
- Doesn't include the input or output data
- Really just refers to temporary data structures which have the life of the algorithm
- Process for determining the space complexity is analogous to determining time complexity



TimeComplexity - Python Wiki

https wiki.python.org/moin/TimeComplexity

Reader

Feedly Google Bookmarks SCCL Magazines

python™

Search titles text

» **TimeComplexity**

» TimeComplexity

- FrontPage
- RecentChanges
- FindPage
- HelpContents
- TimeComplexity

Page

- » Immutable Page
- » Info
- » Attachments
- » More Actions:

User

- » Login

This page documents the time-complexity (aka "Big O" or "Big Oh") of various operations in current CPython. Other Python implementations (or older or still-under development versions of CPython) may have slightly different performance characteristics. However, it is generally safe to assume that they are not slower by more than a factor of O(log n).

Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter.

list

The Average Case assumes parameters generated uniformly at random.

Internally, a list is represented as an array; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a collections.deque instead.

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)

Figure 4: time complexity

list

The Average Case assumes parameters generated uniformly at random.

Internally, a list is represented as an array; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a collections.deque instead.

Operation	**Average Case**	**Amortized Worst Case**
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Figure 5: list

set

See dict -- the implementation is intentionally very similar.

Operation	Average case	Worst Case
x in s	O(1)	O(n)
Union s|t	O(len(s)+len(t))	
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))
Difference s-t	O(len(s))	
s.difference_update(t)	O(len(t))	
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))
s.symmetric_difference_update(t)	O(len(t))	O(len(t) * len(s))

- As seen in the source code the complexities for set difference s-t or s.difference(t) (`set_difference()`) and in-place set difference s.difference_update(t) (`set_difference_update_internal()`) are different! The first one is O(len(s)) (for every element in s add it to the new set, if not in t). The second one is O(len(t)) (for every element in t remove it from s). So care must be taken as to which is preferred, depending on which one is the longest set and whether a new set is needed.
- To perform set operations like s-t, both s and t need to be sets. However you can do the method equivalents even if t is any iterable, for example s.difference(l), where l is a list.

Figure 6: set

dict

The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.

Note that there is a fast-path for dicts that (in practice) only deal with str keys; this doesn't affect the algorithmic complexity, but it can significantly affect the constant factors: how quickly a typical program finishes.

Operation	Average Case	Amortized Worst Case
Copy[2]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[2]	O(n)	O(n)

Figure 7: dict

Summary: complexity analysis

- Good framework for comparing *algorithms*
- Understanding individual algorithms will help you understand performance of an application made up of multiple algorithms
- Also important for understanding data structures
- Caveats:
- There is no standard definition of what constitutes an operation
- It's an asymptotic limit for large n
- Says nothing about the constants
- May make assumptions about dataset (random distribution, etc.)