ECS 32B - Algorithms and Computational Complexity

Aaron Kaloti

UC Davis - Summer Session #2 2020

This content is protected and may not be shared, uploaded, or distributed.

[CC BY-NC-SA 4.0 International Licence]

Overview

- What is an algorithm?
- How do we analyze algorithms?
  - What ***characteristics*** of algorithms do we make measurements about?
  - ***How*** do we make these measurements?

Algorithm

Definition

- Many definitions[1]:
  - "Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output."
  - "Sequence of computational steps that transform the input into the output."
  - "Tool for solving a well-specified computational problem", where the computational problem specifies input/output relationship.

1. Source for all of these: p.5 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition)

Computational Problems

Example #1: Searching

- Given a list[1], find the first occurrence of the target.
- One solution:

```
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

1. Note that in any book/source about algorithms, you may see the term "array" instead of "list". This is because in C/C++, arrays are similar to lists in Python. For now, think of them as interchangeable. However, since we are doing Python in this course, I will usually say "list".

Computational Problems

Example #2: Sorting

- Given a list of values (not necessarily integers), sort them from lowest to highest.
- We'll see many solutions later:
  - Insertion sort.
  - Selection sort
  - Heapsort.
  - Mergesort.
  - Quicksort.
  - Etc.

Computational Problems

- Searching and sorting are well-known, highly studied examples, but not all computational problems are.

Example #3: Example from My ECS 32A Course

Write a function called `get_key_to_min` that takes a dictionary and returns the key of the smallest value in the dictionary that is greater than or equal to 10.

About the values in the given dictionary, you may assume:

1. They are all integers that do not exceed 1000.
2. They are all unique (i.e. different from each other).
3. At least one of the values is at least 10.

Here are some examples of how your program should behave.

```
>>> get_key_to_min({'a':5,'b':8,'abc':13,'def':18,'xyz':10})
'xyz'
>>> get_key_to_min({'a':5,'b':8,'abc':13,'def':18,'xyz':9})
'abc'
>>> get_key_to_min({'a':12,'b':8,'abc':13,'def':18,'xyz':9})
'a'
```

Algorithm Analysis

- From overview: how do we analyze algorithms?

Definition

- **analyzing an algorithm**: "predicting the resources that the algorithm requires"[1].

Resources?

- Could be:
  - Memory.
  - Communication bandwidth.
  - Computer hardware.
  - Computational time.
- From overview: What ***characteristics*** of algorithms do we make measurements about?
  Most common: computational time (followed by memory).

1. p.23 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition)

Measuring Time

Complications: Time Depends on Nature of Input

Example

- Which algorithm is "better" in terms of time?
  - Algorithm A: Takes consistent, "okay" amount of time on all inputs.
  - Algorithm B: Very fast on certain inputs but horribly long on all other inputs.
  - Algorithm C: Slightly faster than algorithm A on most inputs, but prohibitively long on a few other inputs.
- With hypothetical, exaggerated numbers:

Portion of Possible Inputs	Algo A's Speed	Algo B's Speed	Algo C's Speed
First 20%	5 minutes	30 minutes	1 minute
Second 20%	5 minutes	30 minutes	1 minute
Third 20%	5 minutes	30 minutes	1 minute
Fourth 20%	5 minutes	5 seconds	1 minute
Last 20%	5 minutes	5 seconds	1 hour

Measuring Time

Complications: Time Depends on Hardware

Example: Finding a Target in a List[1]

```python
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

```
>>> find([5,8,17,25,3],17)
2
>>> find([5,8,17,25,3],25)
3
>>> find([5,8,17,25,3],22)
-1
```

1. There is already a method called `index()` for lists that does this; see https://www.w3schools.com/python/ref_list_index.asp

Measuring Time

Complications: Time Depends on Hardware

Example: Finding a Target in a List

- `timeit(number=1000)` returns time in seconds to run 1000 times (which mathematically is milliseconds to run 1 time).

```
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1

from timeit import Timer
vals = list(range(100001))  # [0, 1, 2, ..., 99998, 99999, 100000]
t1 = Timer("find(vals, 0)", "from __main__ import find, vals")
print("find(vals, 0)", t1.timeit(number=1000), "milliseconds")
t2 = Timer("find(vals, 50000)", "from __main__ import find, vals")
print("find(vals, 50000)", t2.timeit(number=1000), "milliseconds")
t3 = Timer("find(vals, 100000)", "from __main__ import find, vals")
print("find(vals, 100000)", t3.timeit(number=1000), "milliseconds")
t4 = Timer("find(vals, -1)", "from __main__ import find, vals")
print("find(vals, -1)", t4.timeit(number=1000), "milliseconds")
```

Measuring Time

Complications: Time Depends on Hardware

Example: Finding a Target in a List

- Running on my laptop:

```
find(vals, 0) 0.0002686820225790143 milliseconds
find(vals, 50000) 1.7249889109516516 milliseconds
find(vals, 100000) 3.487803687923588 milliseconds
find(vals, -1) 3.52738544694148 milliseconds
```

```
find(vals, 0) 0.00027292093727737665 milliseconds
find(vals, 50000) 1.8430724210338667 milliseconds
find(vals, 100000) 3.7169026780175045 milliseconds
find(vals, -1) 3.723948363913223 milliseconds
```

```
find(vals, 0) 0.00026883999817073345 milliseconds
find(vals, 50000) 1.762271368992515 milliseconds
find(vals, 100000) 3.60138825699687 milliseconds
find(vals, -1) 3.5785278249531984 milliseconds
```

Measuring Time

Complications: Implementation-Dependence

Example: Finding a Target in a List

- Running on my other laptop:

```
find(vals, 0) 0.0010832000000391417 milliseconds
find(vals, 50000) 3.3695142999999916 milliseconds
find(vals, 100000) 5.425747300000012 milliseconds
find(vals, -1) 5.079429000000005 milliseconds
```

```
find(vals, 0) 0.0005251999999700274 milliseconds
find(vals, 50000) 2.716031500000099 milliseconds
find(vals, 100000) 6.437083600000051 milliseconds
find(vals, -1) 5.15661799999998 milliseconds
```

```
find(vals, 0) 0.0004897999999684544 milliseconds
find(vals, 50000) 2.6653554000000668 milliseconds
find(vals, 100000) 5.3474549000000025 milliseconds
find(vals, -1) 5.152637099999993 milliseconds
```

Measuring Time

Complications: Time Depends on Hardware

Example: Finding a Target in a List

Comparison: Averages of Three (Thousand) Attempts

	First Laptop	Second Laptop
`find(vals, 0)`	0.0003	0.0007
`find(vals, 50000)`	1.7768	2.9170
`find(vals, 100000)`	3.6020	5.7368
`find(vals, -1)`	3.6100	5.1296

Conclusion

- Specifying the absolute time an algorithm takes to run on a certain input has flaws:
    - Hardware concerns: variations (even when on same hardware), other users/programs.
    - Heavily dependent on the input (e.g. what the target is in the above example).

Measuring Time

Complications: Implementation-Dependence

Example: Finding a Target in a List

	for Loop	**while Loop**	**while Loop with Stored Length**
`find(vals, 0)`	0.0003	0.0001	0.0001
`find(vals, 50000)`	1.7768	4.7736	2.8654
`find(vals, 100000)`	3.6020	9.8410	5.8228
`find(vals, -1)`	3.6100	9.9125	5.9048

`for` Loop:

```
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

`while` Loop:

```
def find(lst, target):
    i = 0
    while i < len(lst):
        if lst[i] == target:
            return i
        i += 1
    return -1
```

`while` Loop with Stored Length:

```
def find(lst, target):
    i = 0
    l = len(lst)
    while i < l:
        if lst[i] == target:
            return i
        i += 1
    return -1
```

- Even on same hardware and same programming language, how you code it matters.

Measuring Time

Complications: Programming Language

- Programming languages have different speeds.

	Time
(c) C	$\boxed{1.00}$
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	$\boxed{1.89}$
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	$\boxed{71.90}$
(i) Lua	82.91

- We don't want this noise in our analysis.

Image source: https://jaxenter.com/energy-efficient-programming-languages-137264.html

Measuring Time

Back to the Overview

- Analyzing algorithms with the below is too difficult or misleading:
  - Characteristics to measure: time.
  - How: absolute time.

Addressing the Complications

- What we end up doing:
  - Characteristics to measure: order of growth worst-case running time.
    - Secondary: average-case running time, best-case running time, space.
  - How: big-O of cost function.
- Called **worst-case time complexity**.

Worst-Case Time Complexity

Informal Overview

- Classifies order of magnitude of a mathematical function.
- Captures dominant part.
  - Informally, as n increases, which term of $f(n)$ will matter most?
- Ignores leading constants.
- Captures rate/order of growth.

Example #1

- Function: $f(n) = 2n^2 + 3n + 8$.
- Classification: $f(n)$ is $O(n^2)$.

Example #2

- Function: $f(n) = 3n^4 + 8n^2 + 4n^7 + 9 + 2n^7$.
- Classification: $f(n)$ is $O(n^7)$.

Example #3

- Function: $f(n) = 8n^2 + 2^n + n^3$.
- Classification: $f(n)$ is $O(2^n)$.

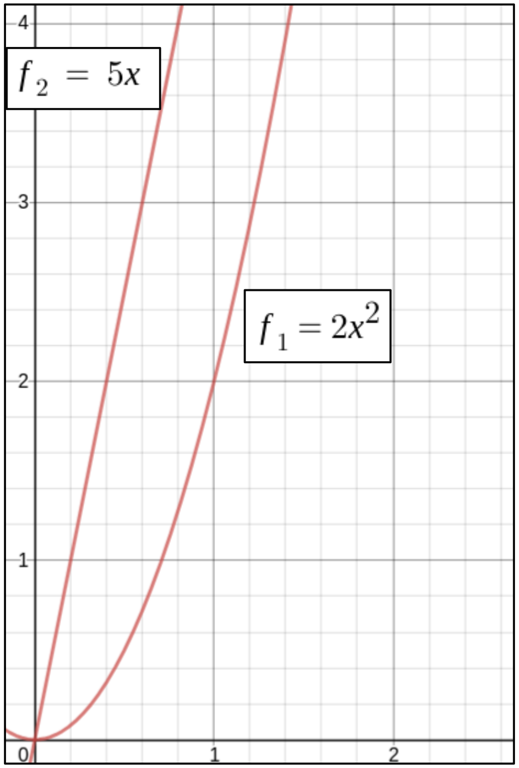
Worst-Case Time Complexity

Formal Definition[1]

- $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
    - T is *asymptotically upper-bounded* by f.

Long-Term Picture

- Can we say that f_1 is $O(f_2)$ (i.e. that f_1 is *asymptotically upper-bounded* by f_2?



1. Definition from p.36 of *Algorithm Design* by Jon Kleinberg and Éva Tardos.

Worst-Case Time Complexity

Formal Definition

- $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
    - T is *asymptotically upper-bounded* by f.

Long-Term Picture

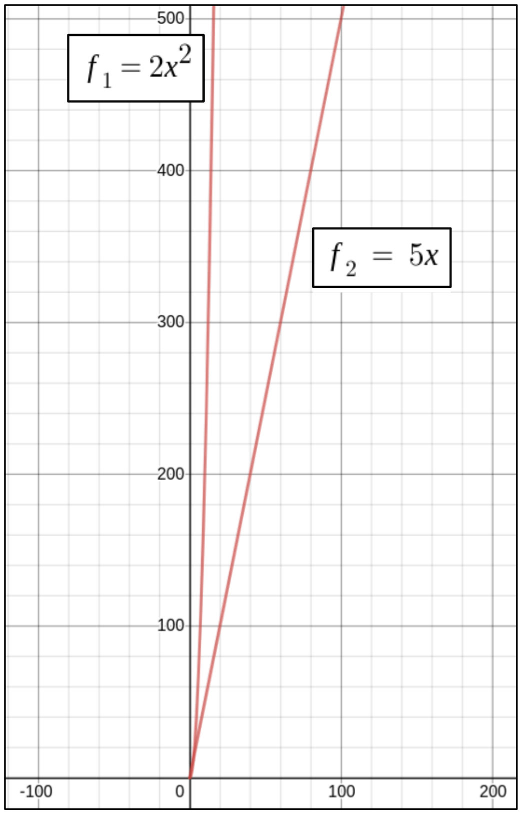
- f_2 is $O(f_1)$. (i.e. f_2 is *asymptotically upper-bounded* by f_1)

Proof

- Set $c = 1$ and $n_0 = 3$.
- If we set $c = \frac{1}{2}$, would n_0 need to change?

Final Remarks

- Only need to find one valid pair of c and n_0.



Worst-Case Time Complexity

Formal Definition

- $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
  - T is *asymptotically upper-bounded* by f.
- Put another way: some constant multiple of $f(n)$ is an asymptotic upper bound on $T(n)$[1].

Another Proof

- **Prompt**: Prove (i.e. find appropriate values of c and n_0 to show) that $f(n) = 8n^2 + 2^n + n^3$ is $O(2^n)$.
- Set $n_0 = 10$, because that's when 2^n begins exceeding n^2 and n^3.
  - Could pick a higher n_0 (or lower n_0 and higher c), but we'll keep it simple.
- Observe that $f(n) = 8n^2 + 2^n + n^3 \leq 8 \cdot 2^n + 2^n + 2^n = 10 \cdot 2^n$ for all $n \geq 10$.
- **Final answer**: $n_0 = 10$, $c = 10$.

n	n^2	n^3	2^n
1	1	1	2
2	4	8	4
3	9	27	8
4	16	64	16
5	25	125	32
6	36	216	64
7	49	343	128
8	64	512	256
9	81	729	512
10	100	1000	1024
11	121	1331	2048

1. From p.47 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Worst-Case Time Complexity

Formal Definition

- $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
  - T is *asymptotically upper-bounded* by f.

One Final Example Proof

- **Prompt**: Prove that $5n^2 + 3n + n$ is $O(n^2)$.
- Let $n_0 = 1$.
- Observe that, for all $n \geq 1$, $5n^2 + 3n + n \leq 5n^2 + 3n^2 + n^2 = 9n^2$.
- **Final answer**: set $n_0 = 1$ and $c = 9$.

Worst-Case Time Complexity

Big-O is an Upper Bound

Example

- Function: $f(n) = 2n^2 + 3n + 8$.
- Classification: $f(n)$ is $O(n^2)$.
  - That is, there are constants c and n_0 such that for all $n \geq n_0$, $f(n) = 2n^2 + 3n + 8 \leq cn^2$.
  - **Also true**: $f(n)$ is $O(n^3)$, $f(n)$ is $O(n^4)$, $f(n)$ is $O(n^5)$, $f(n)$ is $O(n^5 \lg n)$, $f(n)$ is $O(3^n)$, and many other statements...
  - $f(n)$ is *not* $O(n)$.

Different Classes of Functions

- Logarithms (regardless of base) are upper-bounded by polynomials.
  - Example: $\lg n$ is $O(n)$.
- Polynomials are upper-bounded by exponential functions.
  - Example: $5n^{100}$ is $O(2^n)$.
- Exponential functions are upper-bounded by factorial.
  - Example #1: 2^n is $O(n!)$.
    - Informal explanation: $2 \cdot 2 \cdot \ldots \cdot 2 \leq 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.
  - Example #2: 3^n is $O(n!)$.
- $n!$ is $O(n^n)$.
  - Informal explanation: $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n \cdot n$.

Worst-Case Time Complexity

Common Big-Os

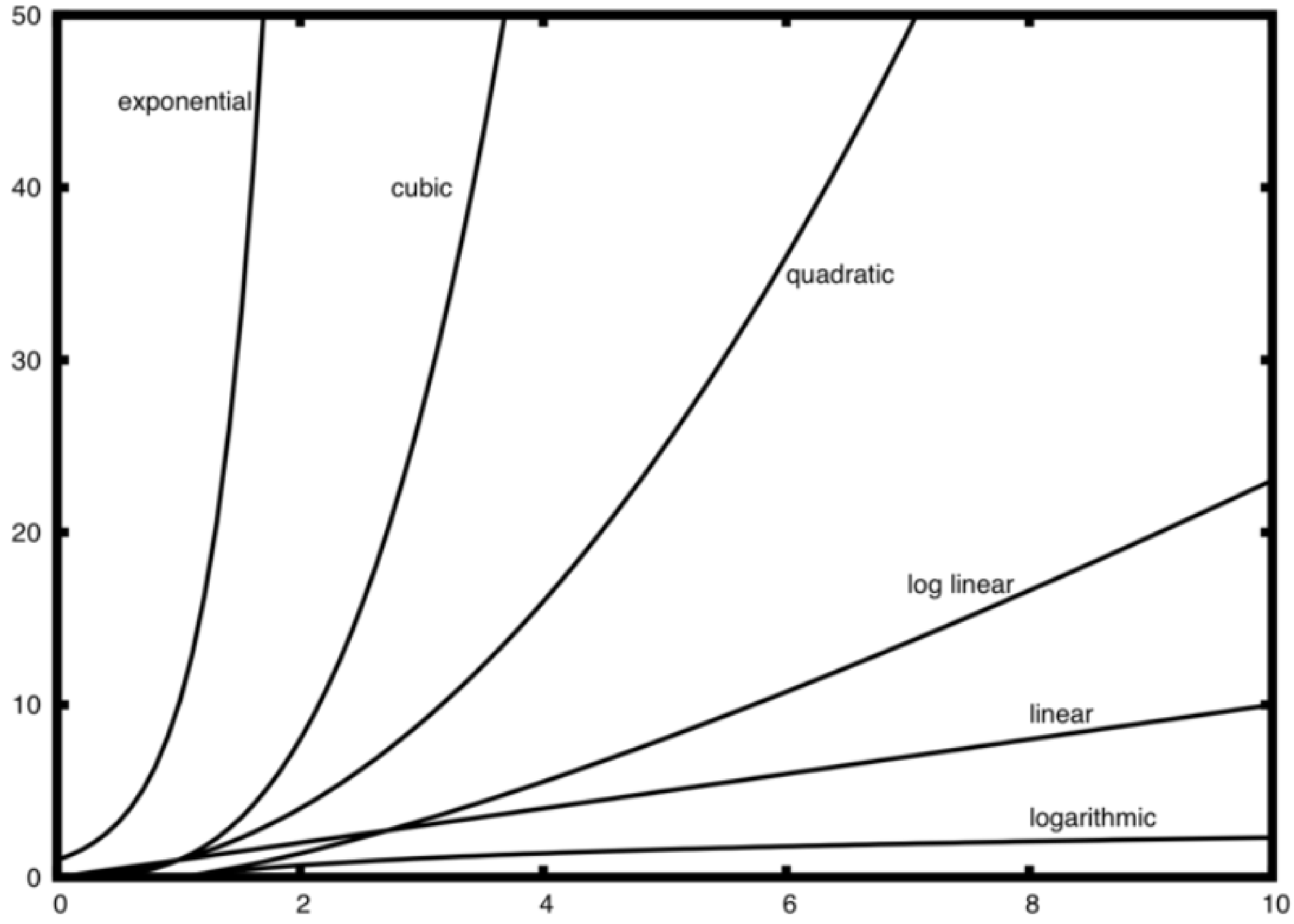


Figure 1: Plot of Common Big-O Functions

Source: Section 3.3 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Worst-Case Time Complexity

Applying Big-O Notation to Code

- Weren't we talking about code? Big-O is math. Code is... not math.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- Let $T(n)$ represent *worst-case* running time of algorithm on input of size n.
  - Let "running time" be "number of computational steps".

Example: `find()` (Revisited)

```python
def find(lst, target):
    i = 0
    l = len(lst)
    while i < l:
        if lst[i] == target:
            return i
        i += 1
    return -1
```

- n is `len(lst)`.
- Worst case: `target` is not in `lst`.
- Could say $T(n) = 1 + 1 + 2n + 1 = 2n + 3$.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example (Continued): Observing the Scaling

- $T(n)$ is $O(n)$ (linear time). Doubling n doubles the runtime.

```
def find(lst, target):
    i = 0
    l = len(lst)
    while i < l:
        if lst[i] == target:
            return i
        i += 1
    return -1
```

- Order of growth ("scaling"):

n	$T(n) = 2n + 3$	$f(n) = n$	**Runtime (Code)**
1000	2003	1000	0.0522
2000	4003	2000	0.1099
3000	6003	3000	0.1699
4000	8003	4000	0.2278

```
from timeit import Timer

vals = list(range(1000))
t1 = Timer("find(vals, -1)", "from __main__ import find, vals")
print("find(vals, -1)", t1.timeit(number=1000), "milliseconds")
vals = list(range(2000))
t2 = Timer("find(vals, -1)", "from __main__ import find, vals")
print("find(vals, -1)", t2.timeit(number=1000), "milliseconds")
vals = list(range(3000))
t3 = Timer("find(vals, -1)", "from __main__ import find, vals")
print("find(vals, -1)", t3.timeit(number=1000), "milliseconds")
vals = list(range(4000))
t4 = Timer("find(vals, -1)", "from __main__ import find, vals")
print("find(vals, -1)", t4.timeit(number=1000), "milliseconds")
```

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example (Continued): Why This is OK

```python
def find(lst, target):
    i = 0
    l = len(lst)
    while i < l:
        if lst[i] == target:
            return i
        i += 1
    return -1
```

- $T(n) = 1 + 1 + 2n + 1 = 2n + 3.$

Addressing Complications

- Why do `i = 0`, `l = len(lst)`, `i += 1`, and `return -1` each cost the same? (Answer: It doesn't matter, since they all take constant time.)
- Why does code run slower if we do `while i < len(lst)`?
- With worst-case time complexity, such details are irrelevant, and they *should* be, because we're concerned with order of growth. (If `len(lst)` were very high, the cost of `i = 0` would be negligible.)

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example (Continued): Alternative Implementations are Viewed Similarly

	`for` Loop	`while` Loop	`while` Loop with Stored Length
`find(vals, 0)`	0.0003	0.0001	0.0001
`find(vals, 50000)`	1.7768	4.7736	2.8654
`find(vals, 100000)`	3.6020	9.8410	5.8228
`find(vals, -1)`	3.6100	9.9125	5.9048

`for` Loop:

```
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

`while` Loop:

```
def find(lst, target):
    i = 0
    while i < len(lst):
        if lst[i] == target:
            return i
        i += 1
    return -1
```

`while` Loop with Stored Length:

```
def find(lst, target):
    i = 0
    l = len(lst)
    while i < l:
        if lst[i] == target:
            return i
        i += 1
    return -1
```

- All of these run in $O(n)$ time.
    - Minor details (e.g. programming language, different computers) are ignored.
    - Doubling input size approximately doubles time.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- n needn't be size of a list[1].

Example: `sum_to_n()`

```
def sum_to_n(n):
    i = 1
    s = 0
    while i <= n:
        s += i
        i += 1
    return s
```

- n is integer.
- Could say $T(n) = 2 + 2n + 1 = 2n + 3$.
- $T(n)$ is $O(n)$.

```
from timeit import Timer
import_line = "from __main__ import sum_to_n"
for n in range(1000, 9000, 1000):
    timer = Timer("sum_to_n({})".format(n),
        import_line)
    print("n={}: {} milliseconds".format(
        n, round(timer.timeit(number=1000), 4)))
```

n	$T(n) = 2n + 3$	$f(n) = n$	**Runtime (milliseconds)**
1000	2003	1000	0.0539
2000	4003	2000	0.109
3000	6003	3000	0.1655
4000	8003	4000	0.216
5000	10003	5000	0.2716
6000	12003	6000	0.3331
7000	14003	7000	0.3804
8000	16003	8000	0.441

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- Cost function needn't involve n.

Example: Constant Time (`get_third()`)

```
# @x is a string or list.
def get_third(x):
    return x[2]

from timeit import Timer
import_line = "from __main__ import get_third, vals"
for n in range(1000, 9000, 1000):
    vals = list(range(n))  # list of n elems
    timer = Timer("get_third(vals)", import_line)
    print("n={}: {} milliseconds".format(n, round(timer.timeit(number=1000), 6)))
```

- $T(n) = 1 \Rightarrow T(n)$ is $O(1)$. (Constant time.)
    - Doubling n does nothing to runtime of `get_third()`.
- Time doesn't seem to change as n increases:

```
n=1000: 5.5e-05 milliseconds
n=2000: 5.5e-05 milliseconds
n=3000: 5.5e-05 milliseconds
n=4000: 5.5e-05 milliseconds
n=5000: 5.5e-05 milliseconds
n=6000: 5.6e-05 milliseconds
n=7000: 5.4e-05 milliseconds
n=8000: 5.4e-05 milliseconds
```

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- We should get used to determining worst-case time complexity without time measurements.

Examples

```
def foo(vals):
    x = 1
    for i in range(len(vals)):
        x *= vals[i]
    for i in range(len(vals)):
        x *= vals[i]
    return x
```

- n is `len(vals)`.
- $T(n) = 1 + n + n + 1 = 2n + 2 \Rightarrow T(n)$ is $O(n)$.
  - e.g. if `len(vals)` doubles from 20 to 40, then the number of loop iterations doubles.

```
def bar(a):
    s = 0
    for i in range(100, 100 + 2 * a):
        s += i
    return s
```

- n is `a`.
- $T(n) = 1 + 2n + 1 = 2n + 2 \Rightarrow T(n)$ is $O(n)$.
  - i.e. if `a` doubles, then the number of iterations doubles.

```
# Assumes len(vals1) == len(vals2).
# Returns first index at which @vals1
# and @vals2 differ.
def find_diff(vals1, vals2):
    for i in range(len(vals1)):
        if vals1[i] != vals2[i]:
            return i
    return -1  # not found
```

- n is `len(vals1)` or `len(vals2)` (same).
- (Worst case) $T(n)$ is $O(n)$.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- Needn't be constant or linear time.

Example: 2D List

```
# Assumes @vals is square, 2D list.
def print_each(vals):
    for row in vals:
        for val in row:
            print(val, end=' ')
        print()
```

- Example usage:

```
>>> print_each([[5,8,2],[4,1,8],[9,12,2]])
5 8 2
4 1 8
9 12 2
```

- n is height and width[1] of 2D list.
- `print_each()` runs in $O(n^2)$ time in the worst case.
  - Quadratic time: doubling n quadruples runtime of `print_each()`.

1. I think of the "height" as the number of rows and the "width" as the number of elements per row. We are assuming each row has the same width.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: 2D List (Continued)

- Quadratic scaling is significant compared to linear scaling[1].

```
# Assumes @vals is square, 2D list.
def print_each(vals):
    for row in vals:
        for val in row:
            pass # print(val, end=' ')
        # print()

from timeit import Timer
import_line = "from __main__ import print_each, vals"
for n in range(100, 900, 100):
    # Create n-by-n 2D list of arbitrary values.
    vals = [[0] * n for i in range(n)]
    timer = Timer("print_each(vals)", import_line)
    print("n={}: {} milliseconds".format(
        n, round(timer.timeit(number=1000), 4)))
```

1. That's why this particular example's measurements were done with increments of 100 instead of 1000.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: 2D List (Continued)

n	$f(n) = n$	$f(n) = n^2$	**Runtime (milliseconds)**
100	100	10000	0.0801
200	200	40000	0.3358
300	300	90000	0.7579
400	400	160000	1.3536
500	500	250000	2.1543
600	600	360000	2.7772
700	700	490000	3.8337
800	800	640000	5.2233

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: Linear Time vs. Constant Time vs. Quadratic Time

```
def lin(n):
    for i in range(n):
        pass

def con(n):
    pass

def quad(n):
    for i in range(n):
        for j in range(n):
            pass
...
```

```
...
from timeit import Timer
import_lin = "from __main__ import lin, n"
import_con = "from __main__ import con, n"
import_quad = "from __main__ import quad, n"
round_num_digits = 4
for n in range(500, 4500, 500):
    print("=== n = {} ===".format(n))
    timer = Timer("lin(n)", import_lin)
    print("lin({}): {} milliseconds".format(
        n, round(timer.timeit(number=1000),
            round_num_digits)))
    timer = Timer("con(n)", import_con)
    print("con({}): {} milliseconds".format(
        n, round(timer.timeit(number=1000),
            round_num_digits)))
    timer = Timer("quad(n)", import_quad)
    print("quad({}): {} milliseconds".format(
        n, round(timer.timeit(number=1000),
            round_num_digits)))
```

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: Linear Time vs. Constant Time vs. Quadratic Time (Continued)

```
=== n = 500 ===
lin(500): 0.0054 milliseconds
con(500): 0.0001 milliseconds
quad(500): 3.0308 milliseconds
=== n = 1000 ===
lin(1000): 0.0147 milliseconds
con(1000): 0.0001 milliseconds
quad(1000): 15.3717 milliseconds
=== n = 1500 ===
lin(1500): 0.0241 milliseconds
con(1500): 0.0001 milliseconds
quad(1500): 38.0262 milliseconds
=== n = 2000 ===
lin(2000): 0.0331 milliseconds
con(2000): 0.0001 milliseconds
quad(2000): 72.0931 milliseconds
...
```

```
...
=== n = 2500 ===
lin(2500): 0.0417 milliseconds
con(2500): 0.0001 milliseconds
quad(2500): 110.2678 milliseconds
=== n = 3000 ===
lin(3000): 0.0535 milliseconds
con(3000): 0.0001 milliseconds
quad(3000): 166.7163 milliseconds
=== n = 3500 ===
lin(3500): 0.0784 milliseconds
con(3500): 0.0001 milliseconds
quad(3500): 234.8094 milliseconds
=== n = 4000 ===
lin(4000): 0.0736 milliseconds
con(4000): 0.0001 milliseconds
quad(4000): 331.0241 milliseconds
```

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

- Would need multiple variables if height and width of 2D list differed.

Example: 2D List (Modified)

```
def print_each(vals):
    for row in vals:
        for val in row:
            pass # print(val, end=' ')
        # print()

from timeit import Timer
import_line = "from __main__ import print_each, vals"
for m in range(100, 500, 100):
    for n in range(100, 500, 100):
        # Create m-by-n 2D list of arbitrary values.
        vals = [[0] * n for i in range(m)]
        timer = Timer("print_each(vals)", import_line)
        print("m={}, n={}: {} milliseconds".format(
            m, n, round(timer.timeit(number=1000), 4)))
```

- m is height and n is width[1].
- `print_each()` runs in $T(n)$ is $O(mn)$ time in the worst case.

1. We are still assuming that each row has the same length (which would be the width).

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: 2D List (Continued)

```
m=100, n=100: 0.0669 milliseconds
m=100, n=200: 0.13 milliseconds
m=100, n=300: 0.191 milliseconds
m=100, n=400: 0.2546 milliseconds
m=200, n=100: 0.1334 milliseconds
m=200, n=200: 0.2584 milliseconds
m=200, n=300: 0.39 milliseconds
m=200, n=400: 0.5181 milliseconds
m=300, n=100: 0.204 milliseconds
m=300, n=200: 0.3973 milliseconds
m=300, n=300: 0.5754 milliseconds
m=300, n=400: 0.7687 milliseconds
m=400, n=100: 0.2736 milliseconds
m=400, n=200: 0.5261 milliseconds
m=400, n=300: 0.7693 milliseconds
m=400, n=400: 1.0305 milliseconds
```

Worst-Case Time Complexity

Applying Big-O Notation to Code: Cost Functions

Example: 2D List (Continued)

- Runtime in milliseconds.

m	n	$f(m,n) = mn$	**Runtime**
100	100	10000	0.0669
100	200	20000	0.13
100	300	30000	0.191
100	400	40000	0.2546
200	100	20000	0.1334
200	200	40000	0.2584
200	300	60000	0.39
200	400	80000	0.5181

m	n	$f(m,n) = mn$	**Runtime**
300	100	30000	0.204
300	200	60000	0.3973
300	300	90000	0.5754
300	400	120000	0.7687
400	100	40000	0.2736
400	200	80000	0.5261
400	300	120000	0.7693
400	400	160000	1.0305

Worst-Case Time Complexity

Applying Big-O Notation to Code: Common Big-Os

Constant Time[1]

- $O(1)$

```
a = 5
```

Nested Loops

- $O(N^2)$
- quadratic time

```
for i in range(n):
    for j in range(n):
        # do stuff
```

More Than Two Loops

- $O(N^3)$ (in this example)

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            # do stuff
```

1. You might also see $O(n^0)$ instead. $O(1)$ is considered an abuse, since it doesn't specify the variable that's approaching infinity; see p.46-47 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein for more information. You may also see $O(k)$ instead, as mentioned here. The vast majority of times, $O(1)$ is used regardless.

Worst-Case Time Complexity

Applying Big-O Notation to Code: Common Big-Os

- Worst-case time complexity?

Logarithmic

- $O(\lg N)$, a.k.a $O(\log_2 N)$
- logarithmic time

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

Worst-Case Time Complexity

Drawbacks of Using Big-O Notation

Example

- For our hypothetical application, n never exceeds 100. Which algorithm should you prefer?
    - Algorithm A: Worst-case running time given by $T_a(n) = 5n^2 \Rightarrow T_a(n)$ is $O(n^2)$.
    - Algorithm B: Worst-case running time given by $T_b(n) = 1000n \Rightarrow T_b(n)$ is $O(n)$.
- The big-Os suggest algorithm B is faster. (Don't forget: lower big-O is better!)
- Chart on the right disagrees ⇒
- Sometimes, big-O is misleading or not worth considering. Think of big-O as a (commonly used) *tool*; when used appropriately, it helps.

n	$5n^2$	$1000n$
10	500	10000
20	2000	20000
30	4500	30000
40	8000	40000
50	12500	50000
60	18000	60000
70	24500	70000
80	32000	80000
90	40500	90000
100	50000	100000

Worst-Case Time Complexity

- The most important thing about big-O notation is not the mathematical aspects but that we can identify the worst-case time complexity of any piece of code.
- Let's make sure we can do this well.

Example #1

```python
# @vals is a list of ints.
def sum_every_other(vals):
    s = 0
    for i in range(0, len(vals), 2):
        s += vals[i]
    return s
```

- n is `len(vals)`.
- $T(n) = \frac{n}{2} = \frac{1}{2}n$ is $O(n)$.

Worst-Case Time Complexity

- A loop that executes a constant number of times (in relation to the input size[1]) contributes a *constant* term.

Example #2

```
for i in range(n):
    # constant time stuff...

a = 3
b = 4

for i in range(n):
    for j in range(a + b):
        # constant time stuff...
```

- n is the input.
- Is this $O(n^2)$?
- Yes. But also: $T(n) = n + 1 + 1 + n \cdot 7 \cdot 1 = 8n + 2$ is $O(n)$.

1. Given the example on this slide, you may find it odd that I say "input size", given that n is *one* integer. The explanation for this is that, as you will learn in ECS 50, all values (including integers) are represented as bits, and as n increases, it will require more bits in its representation; thus, the input size is increasing.

Worst-Case Time Complexity

- Below is a common big-O analysis you'll encounter.

Example #3

```
for i in range(n):
    for j in range(i,n):
        # do stuff that takes constant time
```

- Outer loop iterates n times.
- Inner loop:

i	Number of Inner Loop Iterations
0	n
1	n - 1
2	n - 2
3	n - 3
...	...
n - 2	2
n - 1	1

- Worst-case time complexity dictated by number of inner loop iterations.
- Can't find a constant number of inner loop iterations per outer loop iterations, so calculate the *total* number of inner loop iterations across *all* outer loop iterations.
- Total: $n + (n - 1) + (n - 2) + \ldots + 3 + 2 + 1$ $= (n + 1)\frac{n}{2}$, which is $O(n^2)$.

Worst-Case Time Complexity

Example #4

```
for i in range(n):
    if n % 2 == 0:
        # constant time stuff...
```

- n, an integer, is the input.
- Every other value of `i` results in additional *constant time* work.
- Takes $O(n)$ time.
  - Possible cost function: $T(n) = \frac{n}{2} + 2\frac{n}{2}$.

Worst-Case Time Complexity

The `in` Operator

- With lists, although no loop can be seen, the `in` operator takes linear time in worst case.
    - Worst case: target element is not in list.

```
from timeit import Timer
nums = None
def check_in(vals):
    return -1 in vals

def time_list_in():
    import_line = "from __main__ import check_in, nums"
    global nums
    for n in range(10000, 80001, 10000):
        nums = list(range(n))
        timer = Timer("check_in(nums)", import_line)
        print("n={}: {} milliseconds".format(n, round(timer.timeit(number=1000), 4)))
```

```
n=10000: 0.0584 milliseconds
n=20000: 0.1189 milliseconds
n=30000: 0.1838 milliseconds
n=40000: 0.2474 milliseconds
n=50000: 0.3084 milliseconds
n=60000: 0.369 milliseconds
n=70000: 0.4282 milliseconds
n=80000: 0.4928 milliseconds
```

Worst-Case Time Complexity

The `in` Operator

- With dictionaries, the `in` operator takes constant time regardless of size of dictionary.
  - Due to *hashing* (a later concept).

```
from timeit import Timer
d = None
def check_in(vals):
    return -1 in vals

def time_dict_in():
    import_line = "from __main__ import check_in, d"
    global d
    for n in range(10000, 80001, 10000):
        d = {}
        for i in range(n):
            d[i] = i
        print("d has {} keys; ".format(len(d)), end='')
        timer = Timer("check_in(d)", import_line)
        print("n={}: {} milliseconds".format(n, round(timer.timeit(number=1000), 8)))

time_dict_in()
```

```
d has 10000 keys; n=10000: 5.56e-05 milliseconds
d has 20000 keys; n=20000: 5.56e-05 milliseconds
d has 30000 keys; n=30000: 5.812e-05 milliseconds
d has 40000 keys; n=40000: 5.979e-05 milliseconds
d has 50000 keys; n=50000: 5.728e-05 milliseconds
d has 60000 keys; n=60000: 5.672e-05 milliseconds
d has 70000 keys; n=70000: 5.756e-05 milliseconds
d has 80000 keys; n=80000: 5.7e-05 milliseconds
```

Big-Ω Notation

Definition

- $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.
  - T is *asymptotically lower-bounded* by f.
- Big-Ω is the opposite of big-O.

Example #1

- $T(n) = 8n^2 + 6$
  - $T(n)$ is $\Omega(n)$.
  - $T(n)$ is $\Omega(1)$.
  - $T(n)$ is $\Omega(n^2)$.
  - $T(n)$ is *not* $\Omega(n^3)$.
  - $T(n)$ is *not* $\Omega(n^{100})$.
  - $T(n)$ is *not* $\Omega(2^n)$.

Example #2

- $T(n) = 5n^3 + 2n^2 + n$
  - $T(n)$ is $\Omega(n)$. ($c = 8$, $n_0 = 1$)
  - $T(n)$ is $\Omega(n^3)$. ($c = 5$, $n_0 = 1$)
    - Think of big-O and big-Ω as putting upper/lower bounds on the *dominant* term.
  - $T(n)$ is $\Omega(n^2)$.
  - $T(n)$ is *not* $\Omega(n^4)$.

Big-Θ Notation

Definition

- $T(n)$ is $\Theta(f(n))$ if *both* of the following are true:
    1. $T(n)$ is $O(f(n))$.
    2. $T(n)$ is $\Omega(f(n))$.
- T is *asymptotically tight-bounded* by f.

Example #1

- $T(n) = 8n^2 + 6$
    - $T(n)$ is $\Theta(n^2)$.
    - $T(n)$ is *not* $\Theta(n^3)$, because it is not $\Omega(n^3)$.
    - $T(n)$ is *not* $\Theta(n)$, because it is not $O(n)$.

Example #2

- $T(n) = n \lg n + \frac{n}{2}$.
    - Dominant term is $n \lg n$.
        - $n \lg n$ is "between" n and n^2 in order of growth.
    - $T(n)$ is $O(n^2)$ but not $\Theta(n^2)$.
    - $T(n)$ is $\Theta(n \lg n)$.

Best-Case Analysis

- We needn't just focus on worst-case.

Example #1

- Best-case time complexity?

```
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

- Best case scenario is finding `target` immediately.
- Takes $O(1)$ time.

Caution: Best-Case Time Complexity ≠ Best-Case Input Size

- Don't say the best-case above is when `lst` is empty (i.e. `[]`).
- Best-case and worst-case analyses differ in their assumption about the *nature* of the input, not the size.

Best-Case Analysis

Example #2

- Worst-case and best-case time complexities?

```
# @vals is 2D list, @target is integer. Assume @vals is a square.
# foo() returns row index of the first row in @vals whose
# values sum up to less than @target.
def foo(vals, target):
    for r in range(len(vals)):
        s = 0
        for c in range(len(vals[r])):
            s += vals[r][c]
        if s < target:
            return r
    return -1
```

Worst-Case

- Worst case scenario is not finding such a row (or the last row is such a row).
- Need to consider every value in `vals`.
- Takes $O(n^2)$ (and $\Theta(n^2)$) time, where n is `len(vals)`.

Best-Case

- Best case scenario is finding satisfactory row immediately.
- Only need to consider every value in first row.
- Takes $O(n)$ (and $\Theta(n)$) time, where n is `len(vals)`.

Average-Case Analysis

Overview[1]

- Average performance of the algorithm over all possible inputs.
- Typically involves randomly generating inputs (when the number of possible inputs is huge).
  - Issue: Risks describing how the inputs were randomly generated instead of the how the algorithm truly behaves on average.
- In many cases, average-case and worst-case are same due to constants being ignored.

Example

```
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

- On average, `find()` will need to go *halfway* through `lst`.
- Average-case takes $O(\frac{n}{2}) = O(n)$ time.

1. Further reading: p.31 of *Algorithm Design* by Jon Kleinberg and Éva Tardos.

Notational Issues

Common Usage Issue #1: Underuse

Example

```
for i in range(4, n, 2):
    pass
```

- Worst-case time complexity?
  - $T(n) = \frac{n-4}{2} = \frac{n}{2} - 2$.
  - Can say $T(n)$ is $\Theta(n)$.
  - Can say $T(n)$ is $O(n)$.

- Can also say $T(n)$ is $O(n \lg n)$, $O(n^2)$, $O(n^2 \lg n)$, $O(2^n)$, $O(n^{n^{n^{n^{n^{n^{n^{n^{n}}}}}}}}!)$, etc.

Conclusion

- Big-Θ provides more information than big-O.
- Unfortunately, many resources (e.g. books, online tutorials, interviewers, instructors) say "What is the big-O of the above example?" (meaning trivial answers such as $O(n!)$ and $O(4^n)$ are technically correct) and ***mean to say*** "What is the **big-Θ** of the above example?"[1].
  - In this course, we'll try to always use big-Θ when appropriate (which would be a lot of the time, when we're looking at code). However, don't be surprised if other instructors use big-O all over the place.
  - The directions will always specifically say which one (e.g. big-O, big-Θ) is desired.

1. See the Appendices at the end of this slide set for sentences from a well-respected algorithms textbook that backs me up on this.

Notational Issues

Common Usage Issue #2a: Abuse of Notation

- Common to erroneously equate big-O with worst-case time complexity.

Example

```
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

- Some may say, "The running time of `find()` is $O(n)$," where n is `len(lst)`.
- What they *should* say: "The ***worst-case*** running time of `find()` is $O(n)$ (or $\Theta(n)$).".
  - Technically an abuse to say this (see footnote), because the running time doesn't just depend on the size of the list (but also the list's contents), but it is common.
- To be clear:
  - *In general*, `find()` takes $O(n)$ time (but not $\Omega(n)$ time).
  - *In the worst-case*, `find()` takes $\Theta(n)$ time.
  - *In the best-case*, `find()` takes $\Theta(1)$ time.

See the Appendices at the end of this slide set for sentences from a well-respected algorithms textbook that backs me up on this.

Notational Issues

Common Usage Issue #2b: Big-O = Worst-Case?

- Don't say, "Big-O is for worst-case and big-Θ is for average-case."
  - The worst-case has an objective runtime. How you choose to express this runtime is your choice. Big-O and big-Θ are two tools for this expression.
- Recall two of our starting questions:
  - What ***characteristics*** of algorithms do we make measurements about?
  - ***How*** do we make these measurements?
- Characteristics: If measuring time, must specify which of best-case time, worst-case time, or average-case time.
- How: Regardless of chosen characteristic, can choose between big-O, big-Ω, or big-Θ.
  - Can use big-O with best-case (as we did earlier).
  - Can use big-Ω with worst-case.

Notational Issues

Common Usage Issue #2b: Big-O = Worst-Case? (Continued)

Example #1

```python
# Returns the index of the first occurrence of
# @target in @lst or -1 otherwise.
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

	Worst-Case	**Average-Case**	**Best-Case**
Big-O	$O(n)$, $O(n^2)$, $O(n^3)$, etc.	$O(n)$, $O(n^2)$, $O(n^3)$, etc.	$O(1)$, $O(\lg n)$, $O(n)$, $O(n^2)$, $O(n^3)$, etc.
Big-Ω	$\Omega(1)$, $\Omega(\lg n)$, $\Omega(n)$	$\Omega(1)$, $\Omega(\lg n)$, $\Omega(n)$	$\Omega(1)$
Big-Θ	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

1. In the above table, I'm only using "increments" of n or $\lg n$, but there are (useful) bounds out there that sometimes involve more complicated terms like $\lg \lg n$ or $\alpha(n)$, where $\alpha(n)$ is the inverse Ackermann function.

Notational Issues

Common Usage Issue #3: (Another) Abuse of Notation

- Typical notation:
  - $T(n)$ is $O(f(n))$
    - I will use this one in this class.
  - $T(n) \in O(f(n))$
    - Don't worry about fully understanding this until after ECS 20.
- $T(n) = O(f(n))$ is not technically correct.
  - Why not?
    - $O(f(n))$ describes a *set* of functions.
    - $T(n) = O(f(n))$ is a one-way equality
      - e.g. $n = O(n^2), n^2 = O(n^2) \nRightarrow n = n^2$.
- Use whichever one that you prefer.

Notational Issues

Common Usage Issue #3: (Another) Abuse of Notation

- One well-respected textbook[1] argues that abuses like $T(n) = O(f(n))$ permit certain conveniences.

Placeholder for Anonymous Function

- May have a formula containing an anonymous function that need not be named.

Examples

- $f_1(n) = 2n^2 + \Theta(n)$ is $\Theta(n^2)$.
- If $f_2(n) = 3n^4 + \Theta(n^3)$ and $f_3(n) = n + 6 + O(n^2)$, then:
  - $f_2(n)$ is $\Theta(n^4)$.
  - $f_3(n)$ is $O(n^2)$ (but might not be $\Theta(n^2)$).
  - f_3 is $O(f_2)$.
- $T(n) = T(n-1) + O(1)$ is $\Theta(n)$.

1. See p.49 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition).

Noteworthy Properties

- *Symmetry*: $f(n)$ is $\Theta(g(n))$ if and only if $g(n)$ is $\Theta(f(n))$.
- *Transpose symmetry*: $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

Space Complexity

- Again, recall two of our starting questions:
  - What ***characteristics*** of algorithms do we make measurements about?
  - ***How*** do we make these measurements?
- Can measure space (i.e. worst-case space complexity, average-case space complexity, and best-case space complexity), and can use big-O, big-Ω, and/or big-Θ.
- We'll focus on worst-case space complexity.

Space Complexity

Caveat: Definitions

- **space complexity**: too much ambiguity regarding whether the input size is included or not.
- **auxiliary space**: space used by code, as a function of n (or whatever the input variables are), ignoring the input size.

Example #1

- Worst-case space complexity?

```
def find(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

- Auxiliary space is $O(1)$; the only additional variables created is `i`, which is always a single integer.
- Space complexity is arguably $\Theta(n)$ due to `len(lst)`, since `n` is `len(lst)`.
  - This obscures the good auxiliary space usage.

Remarks

- We will use auxiliary space, and I will always explicitly specify it, e.g. "What is the space complexity (auxiliary space) of ...?"

Space Complexity

Example #2

- Worst-case space complexity (auxiliary space)?

```
# @vals is list.
def compute_sum(vals):
    s = 0
    for x in vals:
        s += x
    return s
```

- `compute_sum()` only creates `s` and `x` $\Rightarrow \Theta(1)$.

Space Complexity

Example #3

- Worst-case time complexity and space complexity (auxiliary space)?

```
# @vals is a 2D list.
def sum_each_row(vals):
    sums = []
    for row in vals:
        s = 0
        for val in row:
            s += val
        sums.append(s)
    return sums
```

- Time complexity: $\Theta(mn)$, where m is `len(vals)` and n is length of any row in `vals`. (The rows would all be same length in worst-case scenario.)
- Auxiliary space: `sum_each_row()` creates a list `sums` and two variables `row` and `s`; `sums` reaches length m. Thus, $\Theta(m)$.

Remarks

- Impossible for space complexity to be worse than time complexity[1].

1. https://cs.stackexchange.com/questions/115309/is-space-complexity-always-less-than-or-equal-to-time-complexity

Space Complexity

Example #4

```
# @n is integer.
# Creates @n-by-@n 2D list and asks user for every integer to put in it.
def create_2d_list(n):
    vals = []  # 2d list
    for i in range(n):
        row = []
        for j in range(n):
            row.append(int(input("Enter integer: ")))
        vals.append(row)
    return vals
```

- Time complexity: $O(n^2)$.
- Auxiliary space: $O(n^2)$.

References / Further Reading

- **Primary reference**: Chapter 3 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.
- Chapters 1-3 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition)

Appendix: Little-o and Little-ω Notations

- $T(n)$ is $o(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.
- A similar definition follows for $\omega(f(n))$ (i.e. "little/small omega").

Analogy[1]

- Analogy between asymptotic comparison of two functions f and g and two real numbers a and b.

Asymptotic Comparison	Comparison of Real Numbers
$f(N)$ is $O(g(N))$	$a \leq b$
$f(N)$ is $\Omega(g(N))$	$a \geq b$
$f(N)$ is $\Theta(g(N))$	$a = b$
$f(N)$ is $o(g(N))$	$a < b$
$f(N)$ is $\omega(g(N))$	$a > b$

1. Source: Page 52 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Appendix: Source on Common Usage Issues

- Here are sentences from p.47-48 of a well-respected algorithms textbook called *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein that support what I said about common usage issues earlier.

Common Issue #1

If you have seen O-notation before, you might find it strange that we should write, for example, $n = O(n^2)$. In the literature, we sometimes find O-notation informally describing asymptotically tight bounds, that is, what we have defined using Θ-notation. In this book, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds is standard in the algorithms literature.

Common Issue #2

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n, the actual running time varies, depending on the particular input of size n. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.