Algorithms & Analysis of Algorithms

CS 331: Data Structures and Algorithms
Recall: we are *problem solvers*

... our “solutions” — in abstract form — are often expressed as *algorithms*
def., Algorithm: an *unambiguous description* of how to perform a *computation* that can be *completed in finite time*
typically, an algorithm accepts *initial input*, is defined in terms of intermediate *transformations/states*, and produces *output*
e.g., input = integers n, m;

output = greatest common denominator of n, m
e.g., input = list of sorted values;
output = max value
e.g., input = list of *unsorted* values;
output = max value
e.g., \( \text{input} = \text{list of unsorted values} \); 
\( \text{output} = \text{list of sorted values} \)
e.g., input = chess board state & turn;
output = next best move
e.g., input = Sudoku puzzle;
output = solved Sudoku grid
e.g., input = square $s$ in Sudoku grid; output = list of units for $s$
an algorithm is a *recipe* for solving a specific *problem*

... but a given problem can potentially be solved by many different algorithms!
to choose the appropriate algorithm, we need to analyze and compare them.

what metrics?
- correctness
- complexity
- execution time ← focus on this first
- space (memory) requirements
- special considerations (?)
Digression: time measurement
>>> import time
>>> time.time()
1443029788.348523
>>> from time import time
>>> time()
1443029795.575447
>>> time() / (365 * 24 * 60 * 60)
45.758176356261224
>>> 2015 - time() // (365 * 24 * 60 * 60)
1970.0
```python
>>> import random
>>> l = list(range(1000))
>>> random.shuffle(l)
>>> l[0:10]
[644, 40, 918, 875, 261, 437, 780, 998, 843, 92]
>>> start = time()
>>> s = sorted(l)
>>> end = time()
>>> end - start
11.23031497001648
```
all_units = ([[r+c for c in cols] for r in rows] +
            [[r+c for r in rows] for c in cols] +
            [[r+c for r in rs for c in cs]
             for rs in ('ABC', 'DEF', 'GHI')
             for cs in ('123', '456', '789')])

square_units = [(s, [u for u in all_units if s in u])
                for s in squares]

for entry in square_units:
    if entry[0] == 'C2':
        c2_units = entry[1]
        break
all_units = ([[r+c for c in cols] for r in rows] +
            [[r+c for r in rows] for c in cols] +
            [[r+c for r in rs for c in cs]
             for rs in ('ABC', 'DEF', 'GHI')
             for cs in ('123', '456', '789')])

square_units = [(s, [u for u in all_units if s in u])
                for s in squares]

def search_units_list_list(sq):
    for entry in units_list_list:
        if entry[0] == sq:
            return entry[1]
```python
def search_units_list(sq):
    for entry in units_list:
        if entry[0] == sq:
            return entry[1]

def timed_search_units_list(sq):
    start = time()
    for entry in units_list:
        if entry[0] == sq:
            rval = entry[1]
            break
    end = time()
    return rval, end - start
```
```python
def timed_search_units_list(sq):
    start = time()
    for entry in units_list:
        if entry[0] == sq:
            rval = entry[1]
            break
    end = time()
    return rval, end-start

print(timed_search_units_list('A1')[1])
print(timed_search_units_list('I9')[1])
```

```
1.597404799804688e-05
1.6927719116210938e-05
```
```python
def timed_call(fn, *args):
    start = time()
    rval = fn(*args)
    end = time()
    return rval, end - start

print(timed_call(search_units_list, 'A1')[1])
print(timed_call(search_units_list, 'I9')[1])
```

1.9073486328125e-06
1.1920928955078125e-05
```python
def timed_call(fn, *args):
    start = time()
    rval = fn(*args)
    end = time()
    return rval, end - start

print([timed_call(search_units_list, s)[1]
       for s in squares])
```

```
  5.00e-06, 5.96e-06, 5.96e-06, 6.19e-06, 5.96e-06, 5.96e-06, 7.15e-06, 6.91e-06, 6.91e-06, 5.96e-06, ...
  9.05e-06, 9.05e-06, 9.05e-06, 9.05e-06, 9.05e-06, 9.05e-06, 1.00e-05, 1.00e-05]
```
```python
def timed_calls(fn, *args, number=1000000):
    start = time()
    for _ in range(number):
        fn(*args)
    end = time()
    return end - start

print(timed_calls(search_units_list, 'I1'))
print(timed_calls(search_units_list, 'I9'))
```

4.227874994277954
4.534433841705322
timeit module already does this!

```python
import timeit
timeit.timeit(stmt='pass',
               setup='pass',
               number=1000000,
               globals=None)
```
>>> timeit('random.shuffle(l); sorted(l)',
    'import random; l=list(range(100))',
    number=100000)
8.864088952999737
def search_units_list(sq):
    for entry in units_list:
        if entry[0] == sq:
            return entry[1]

def search_units_dict(sq):
    return units_dict[sq]

assert all(search_units_list(s) == search_units_dict(s)
            for s in squares)
```python
def search_units_list(sq):
    for entry in units_list:
        if entry[0] == sq:
            return entry[1]

def search_units_dict(sq):
    return units_dict[sq]

print(timeit('search_units_list(random.choice(squares))',
             number=10**6,
             globals=globals()))
print(timeit('search_units_dict(random.choice(squares))',
             number=10**6,
             globals=globals()))
```

```
3.176056673997664
1.0228661349974573
```
§Algorithm Design
(a case study)
def search_units_list(sq):
    for entry in units_list:
        if entry[0] == sq:
            return entry[1]
← ideas for improving?
idea: “binary” search algorithm

- given a list sorted by square names, start search in the middle

- if not found, search left or right half if square to find comes alphabetically before or after the square in the middle
def binary_search_units_list(sq):
    bot = 0
    top = len(units_list) - 1
    mid = (bot + top) // 2
    while units_list[mid][0] != sq:
        if sq < units_list[mid][0]:
            top = mid - 1
        else:
            bot = mid + 1
        mid = (bot + top) // 2
    return units_list[mid][1]
print(timeit('search_units_list(random.choice(squares))', number=10**6, globals=globals()))
print(timeit('binary_search_units_list(random.choice(squares))', number=10**6, globals=globals()))
print(timeit('search_units_dict(random.choice(squares))', number=10**6, globals=globals()))

3.2335270900002797
2.7439835610020964
1.0194689620002464
to generalize binary search, need to:

- accept all sequence types
- deal with case where item is not found
- decide what to return when item is found (True?, the value itself?)
def binary_search(lst, to_find):
    bot = 0
    top = len(lst) - 1
    mid = (bot + top) // 2
    while lst[mid] != to_find and bot <= top:
        if to_find < lst[mid]:
            top = mid - 1
        else:
            bot = mid + 1
        mid = (bot + top) // 2
    return lst[mid] if bot <= top else None

assert(all(binary_search(range(100), n) is not None for n in range(100)))
assert(binary_search(range(100), -1) is None)
assert(binary_search(range(100), 100) is None)
>>> binary_search([1, 3, 9, 10, 21], 3)
3

(return value seems a bit pointless)
would like to find an item in a collection based on some *search key*

- support via a *key extraction function*
```python
def search(lst, to_find, key_fn):
    """Linear search, using a key""
    for item in lst:
        if key_fn(item) == to_find:
            return item
    else:
        return None

superheroes = [
    ('Batman', 'Bruce Wayne'),
    ('Ironman', 'Tony Stark'),
    ('Mr. Fantastic', 'Reed Richards'),
    ('Professor X', 'Charles Xavier'),
    ('Spiderman', 'Peter Parker'),
    ('Superman', 'Clark Kent'),
    ('The Hulk', 'Bruce Banner'),
    ('Thing', 'Benjamin Grimm')]
```
superheroes = [ ('Batman', 'Bruce Wayne'), ('Ironman', 'Tony Stark'), ('Mr. Fantastic', 'Reed Richards'), ('Professor X', 'Charles Xavier'), ('Spiderman', 'Peter Parker'), ('Superman', 'Clark Kent'), ('The Hulk', 'Bruce Banner'), ('Thing', 'Benjamin Grimm') ]

print(search(superheroes, 'Batman', first))
print(search(superheroes, 'Thing', first))
print(search(superheroes, 'Peter Parker', second))
print(search(superheroes, 'Michael Lee', second))

('Batman', 'Bruce Wayne')
('Thing', 'Benjamin Grimm')
('Spiderman', 'Peter Parker')
None
superheroes = [('Batman', 'Bruce Wayne'),
               ('Ironman', 'Tony Stark'),
               ('Mr. Fantastic', 'Reed Richards'),
               ('Professor X', 'Charles Xavier'),
               ('Spiderman', 'Peter Parker'),
               ('Superman', 'Clark Kent'),
               ('The Hulk', 'Bruce Banner'),
               ('Thing', 'Benjamin Grimm')]

print(search(superheroes, 'Batman',
              lambda pair: pair[0]))  # anonymous fn

print(search(superheroes, 'peter parker',
              lambda pair: pair[1].lower()))  # anonymous fn

('Batman', 'Bruce Wayne')
('Spiderman', 'Peter Parker')
Can use a lambda to provide a *sane default* to the key function:

```python
def search(lst, to_find, key_fn=lambda x:x):
    for item in lst:
        if key_fn(item) == to_find:
            return item
    else:
        return None
```

`lambda x:x ← “identity function”`
```python
def search(lst, to_find, key_fn=lambda x: x):
    for item in lst:
        if key_fn(item) == to_find:
            return item
    return None

print(search(range(100), 5))
print(search(range(100), -1))
print(search(range(100), 36, lambda x: x**2))
```

5
None
6
keyed binary search function:

def binary_search(lst, to_find, key=lambda x: x):
    bot = 0
    top = len(lst) - 1
    mid = (bot + top) // 2
    while key(lst[mid]) != to_find and bot <= top:
        if to_find < key(lst[mid]):
            top = mid - 1
        else:
            bot = mid + 1
    mid = (bot + top) // 2
    return lst[mid] if bot <= top else None
superheroes = [ ('Batman', 'Bruce Wayne'),
 ('Ironman', 'Tony Stark'),
 ('Mr. Fantastic', 'Reed Richards'),
 ('Professor X', 'Charles Xavier'),
 ('Spiderman', 'Peter Parker'),
 ('Superman', 'Clark Kent'),
 ('The Hulk', 'Bruce Banner'),
 ('Thing', 'Benjamin Grimm') ]

print(binary_search(superheroes, 'The Hulk',
    key=lambda pair: pair[0]))
print(binary_search(superheroes, 'Dr. Doom',
    key=lambda pair: pair[0]))

('The Hulk', 'Bruce Banner')
None
superheroes = [('Batman', 'Bruce Wayne'),
('Ironman', 'Tony Stark'),
('Mr. Fantastic', 'Reed Richards'),
('Professor X', 'Charles Xavier'),
('Spiderman', 'Peter Parker'),
('Superman', 'Clark Kent'),
('The Hulk', 'Bruce Banner'),
('Thing', 'Benjamin Grimm')]

print(binary_search(superheroes, 'Charles Xavier',
        key=lambda pair: pair[1]))
print(binary_search(superheroes, 'Bruce Banner',
        key=lambda pair: pair[1]))

('Professor X', 'Charles Xavier') ← (got lucky)
None

beware: binary search requires that the list be sorted (by key)!
def binary_search(lst, to_find, key=lambda x: x):
    def binary_search_rec(bot, top):
        """Recursive helper function""
        mid = (bot + top) // 2
        if bot > top:
            return None
        elif to_find == key(lst[mid]):
            return lst[mid]
        elif to_find < key(lst[mid]):
            return binary_search_rec(bot, mid-1)  # recurse
        else:
            return binary_search_rec(mid+1, top)  # recurse
    return binary_search_rec(0, len(lst)-1)

Yet another viable implementation!
(more on this later)
comparing linear and binary search:

```python
print('search takes',
      timeit('search(l, random.randrange(100))',
             'l=list(range(100))',
             globals=globals()))

print('binary search takes',
      timeit('binary_search(l, random.randrange(100))',
             'l=list(range(100))',
             globals=globals()))
```

search takes 6.235889658004453
binary search takes 3.7670211430013296
hard numbers — as obtained via timing — are useful, but difficult to generalize

consider:

- how might they be affected by different platforms and implementations?

- what happens when we increase the input size (i.e., # items to search)?
for n in range(100, 1000+1, 100):
    print('for', n, 'values, search takes',
    timeit('search(l, random.randrange(n))',
    'l=list(range({}))'.format(n),
    globals=globals()))

for 100 values, search takes 6.244413117005024
for 200 values, search takes 11.09209362600086
for 300 values, search takes 16.19163142999605
for 400 values, search takes 22.83937353899819
for 500 values, search takes 27.96151282599748
for 600 values, search takes 33.72683835899806
for 700 values, search takes 38.81316025299748
for 800 values, search takes 43.92521726700215
for 900 values, search takes 49.25267040800099
for 1000 values, search takes 54.73558082799718
for n in range(100, 1000+1, 100):
    print('for', n, 'values, search takes',
    timeit('search(l, random.randrange(n))',
            'l=list(range({}))'.format(n),
        globals=globals()))
for n in range(100, 1000+1, 100):
    print('for', n, 'values, binary search takes',
          timeit('binary_search(l, random.randrange(n))',
                 'l=list(range({}))'.format(n),
                 globals=globals()))

for 100 values, binary search takes 4.047461039001064
for 200 values, binary search takes 4.708190935001767
for 300 values, binary search takes 5.204135913998471
for 400 values, binary search takes 5.435238691003178
for 500 values, binary search takes 5.519282907996967
for 600 values, binary search takes 5.981077292998644
for 700 values, binary search takes 5.981077292999544
for 800 values, binary search takes 5.994120800998644
for 900 values, binary search takes 6.140121288997761
for 1000 values, binary search takes 6.050188294000691
for n in range(100, 1000+1, 100):
    print('for', n, 'values, binary search takes',
    timeit('binary_search(l, random.randrange(n))',
    'l=list(range({}))'.format(n),
    globals=globals()))
Runtime of binary search is not directly proportional to the size of the input list.
def search(lst, to_find, key_fn=lambda x:x):
    iters = 0
    for item in lst:
        iters += 1
        if key_fn(item) == to_find:
            return item, iters
    else:
        return None, iters

def binary_search(lst, to_find, key=lambda x:x):
    bot = 0
    top = len(lst) - 1
    mid = (bot + top) // 2
    iters = 0
    while key(lst[mid]) != to_find and bot <= top:
        iters += 1
        if to_find < key(lst[mid]):
            top = mid - 1
        else:
            bot = mid + 1
        mid = (bot + top) // 2
    return (lst[mid] if bot <= top else None), iters
superheroes = [ ('Batman', 'Bruce Wayne'),
 ('Ironman', 'Tony Stark'),
 ('Mr. Fantastic', 'Reed Richards'),
 ('Professor X', 'Charles Xavier'),
 ('Spiderman', 'Peter Parker'),
 ('Superman', 'Clark Kent'),
 ('The Hulk', 'Bruce Banner'),
 ('Thing', 'Benjamin Grimm') ]

print(search(superheroes, 'Batman', lambda e:e[0]))
print(binary_search(superheroes, 'Batman', lambda e:e[0]))
print(search(superheroes, 'Thing', lambda e:e[0]))
print(binary_search(superheroes, 'Thing', lambda e:e[0]))

(('Batman', 'Bruce Wayne'), 1)
(('Batman', 'Bruce Wayne'), 2)
(('Thing', 'Benjamin Grimm'), 8)
(('Thing', 'Benjamin Grimm'), 3)
n_elems  = 1200
n_trials = 10000
l = list(range(n_elems))

tot_iters = sum(search(l, random.randint(0, n_elems)) for _ in range(n_trials))
print('Average iters for search', tot_iters / n_trials)

tot_iters = sum(binary_search(l, random.randint(0, n_elems)) for _ in range(n_trials))
print('Average iters for binary search', tot_iters / n_trials)
binary search *halves* the number of search items in each iteration

... so to increase the number of required iterations to find the search item, must *double* the input size!
for n in (2**e for e in range(7, 17)):
    l = list(range(n))
    iters = binary_search(l, -1)[1]  # -1 doesn't exist!
    print('for', n, 'values, binary search needs max of', iters, 'iters')

for 128 values, binary search needs max of 7 iters
for 256 values, binary search needs max of 8 iters
for 512 values, binary search needs max of 9 iters
for 1024 values, binary search needs max of 10 iters
for 2048 values, binary search needs max of 11 iters
for 4096 values, binary search needs max of 12 iters
for 8192 values, binary search needs max of 13 iters
for 16384 values, binary search needs max of 14 iters
for 32768 values, binary search needs max of 15 iters
for 65536 values, binary search needs max of 16 iters
# math review: exponent-logarithm relationship

```python
for e in range(7, 17):
    print('2^{:<2} = {:<5} ; log_2({:>5}) = {:>2.0f}')
    .format(e, 2**e, 2**e, math.log2(2**e))
```

\[
\begin{align*}
2^7 &= 128 \quad ; \quad \log_2(128) = 7 \\
2^8 &= 256 \quad ; \quad \log_2(256) = 8 \\
2^9 &= 512 \quad ; \quad \log_2(512) = 9 \\
2^{10} &= 1024 \quad ; \quad \log_2(1024) = 10 \\
2^{11} &= 2048 \quad ; \quad \log_2(2048) = 11 \\
2^{12} &= 4096 \quad ; \quad \log_2(4096) = 12 \\
2^{13} &= 8192 \quad ; \quad \log_2(8192) = 13 \\
2^{14} &= 16384 \quad ; \quad \log_2(16384) = 14 \\
2^{15} &= 32768 \quad ; \quad \log_2(32768) = 15 \\
2^{16} &= 65536 \quad ; \quad \log_2(65536) = 16
\end{align*}
\]

i.e., \[ b^x = y \iff \log_b y = x \]
for n in (2**e for e in range(7, 17)):
    print('for', n, 'values, binary search takes',
          timeit('binary_search(l, random.randrange(n))',
                 'l=list(range({}))'.format(n),
                 globals=globals()))

for 128 values, binary search takes 3.65576458799478
for 256 values, binary search takes 4.01973975000146
for 512 values, binary search takes 4.56810258099722
for 1024 values, binary search takes 5.01432827299868
for 2048 values, binary search takes 5.54799817300227
for 4096 values, binary search takes 5.94262130399874
for 8192 values, binary search takes 6.51392442900396
for 16384 values, binary search takes 6.96816744899842
for 32768 values, binary search takes 7.34519273899786
for 65536 values, binary search takes 8.07719282899779
for n in (2**e for e in range(7, 17)):
    print('for', n, 'values, binary search takes',
          timeit('binary_search(l, random.randrange(n))',
                 'l=list(range({}))'.format(n),
                 globals=globals()))
Runtime of binary search is proportional to the *exponent* of the size of the input.

I.e., it is not only *faster* than regular search for a given input size, it *scales much better* to larger inputs!
Next: how to *formally express* these ideas?
I.e., we’d like a succinct way to describe:

- algorithm *runtime* based on input size
- the *relative performance* of algorithms
- algorithm *growth rate*
§ Runtime Analysis
Reframing the problem:

Given an algorithm that takes *input size* $n$, we want a function $T(n)$ that describes the *running time* of the algorithm.
input size might be the number of items in the input (e.g., as in a list), or the magnitude of the input value (e.g., for numeric input).

an algorithm may also be dependent on the size of more than one input.
```python
def sort(vals):
    # input size = len(vals)

def factorial(n):
    # input size = n

def gcd(m, n):
    # input size = (m, n)
```
running time is based on \# of primitive operations (e.g., statements, computations) carried out by the algorithm.

ideally, machine independent!
```python
def factorial(n):
    prod = 1
    for k in range(2, n+1):
        prod *= k
    return prod
```

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod = 1</td>
<td>$c_1$</td>
<td>1</td>
</tr>
<tr>
<td>for k in range(2, n+1):</td>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>prod *= k</td>
<td>$c_3$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>return prod</td>
<td>$c_4$</td>
<td>1</td>
</tr>
</tbody>
</table>

$$T(n) = c_1 + (n - 1)(c_2 + c_3) + c_4$$

Messy! Per-instruction costs obscure the “big picture” runtime function.
def factorial(n):
    prod = 1
    for k in range(2, n+1):
        prod *= k
    return prod

times

T(n) = 2(n - 1) + 2 = 2n

Simplification #1: ignore actual cost of each line of code.
Runtime is clearly linear w.r.t. input size.
Next: a sort algorithm — *insertion* sort

Inspiration: sorting a hand of cards
Insertion sort; intuitive description:

- split hand into a sorted side (initially just the leftmost card) and unsorted side (rest of cards, on right)

- one at a time, *insert* one unsorted card into the sorted side, shifting sorted cards to make room as needed
\[
\begin{align*}
[5, & 2, 3, 1, 4] \\
[2, & 5, 3, 1, 4] \\
[2, & 3, 5, 1, 4] \\
[1, & 2, 3, 5, 4] \\
[1, & 2, 3, 4, 5]
\end{align*}
\]

(insertion procedure not shown)
insertion procedure for 1 into sorted side:

\[
\begin{bmatrix}
2, & 3, & 5, & 1, & 4
\end{bmatrix}
\]

to_insert = 1

\[
\begin{bmatrix}
2, & 3, & _, & 5, & 4
\end{bmatrix}
\]

\[
\begin{bmatrix}
2, & _, & 3, & 5, & 4
\end{bmatrix}
\]

\[
\begin{bmatrix}
_, & 2, & 3, & 5, & 4
\end{bmatrix}
\]

\[
\begin{bmatrix}
1, & 2, & 3, & 5, & 4
\end{bmatrix}
\]
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert

init: [5, 2, 3, 1, 4]

insertion: [2, 3, 5, 1, 4] to_insert = 1
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert

?’s will vary based on initial “sortedness”
... useful to contemplate worst case scenario
```python
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert

times
7
6
5
4
3
2
1

worst case arises when list values start out in reverse order! (exercise: what if sorted?)
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert

\textit{worst case} analysis — this is our default analysis hereafter unless otherwise noted
Review (or crash course) on *arithmetic series*

e.g., $1+2+3+4+5$ ($=15$)

Sum can be found by:

- adding first and last term ($1+5=6$)
- multiplying by num of values ($6\times 5=30$)
- then dividing by two ($30/2=15$)
e.g., $1+2+3+4+5 = 15$

Intuition: values to the left and right of the mean “balance” each other out ... so just compute the mean, then multiply it by the number of values!
i.e., $1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{(n + 1)n}{2}$
```python
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert
```
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert
def insertion_sort(vals):
    for j in range(1, len(vals)):
        to_insert = vals[j]
        i = j - 1
        while i >= 0 and vals[i] > to_insert:
            vals[i+1] = vals[i]
            i -= 1
        vals[i+1] = to_insert

T(n) = 4(n – 1) + \frac{2n(n – 1)}{2} + \frac{(n + 2)(n – 1)}{2}
    = \frac{8n – 8 + 2n^2 – 2n + n^2 + n – 2}{2}
    = \frac{3n^2 + 7n – 10}{2}
\[ T(n) = \frac{3n^2 + 7n - 10}{2} \]

i.e., runtime of insertion sort is a \textit{quadratic function} of its input size.

Simplification #2: only consider \textit{leading term}; i.e., with the \textit{highest order of growth}

Simplification #3: \textit{ignore constant coefficients}
\[ T(n) = \frac{3n^2 + 7n - 10}{2} \]

... we conclude that insertion sort has a \textit{worst-case order of growth} of \( n^2 \)

we write: \( T(n) = \Theta(n^2) \)

read: “is big-O of”
formally, $f(n) = O(g(n))$

means that there exists constants $c, n_0$

such that $0 \leq f(n) \leq c \cdot g(n)$

for all $n \geq n_0$
i.e., \( f(n) = O(g(n)) \)

intuitively means that \( g \) (multiplied by a constant factor) sets an upper bound on \( f \) as \( n \) gets large — i.e., an asymptotic bound
3.1 Asymptotic notation

Figure 3.1 Graphic examples of the \( \Theta \), \( O \), and \( \Omega \) notations. In each part, the value of \( n_0 \) shown is the minimum possible value; any greater value would also work.

(a) \( \Theta \)-notation bounds a function to within constant factors. We write \( f(n) / \Theta g(n) \) if there exist positive constants \( n_0 \), \( c_1 \), and \( c_2 \) such that at and to the right of \( n_0 \), the value of \( f(n) \) always lies between \( c_1 g(n) \) and \( c_2 g(n) \) inclusive.

(b) \( O \)-notation gives an upper bound for a function to within a constant factor. We write \( f(n) / O g(n) \) if there are positive constants \( n_0 \) and \( c \) such that at and to the right of \( n_0 \), the value of \( f(n) \) always lies on or below \( c g(n) \).

(c) \( \Omega \)-notation gives a lower bound for a function to within a constant factor. We write \( f(n) / \Omega g(n) \) if there are positive constants \( n_0 \) and \( c \) such that at and to the right of \( n_0 \), the value of \( f(n) \) always lies on or above \( c g(n) \).

A function \( f(n) \) belongs to the set \( \Theta g(n) \) if there exist positive constants \( c_1 \) and \( c_2 \) such that it can be "sandwiched" between \( c_1 g(n) \) and \( c_2 g(n) \), for sufficiently large \( n \). Because \( \Theta g(n) \) is a set, we could write "\( f(n) / \Theta 2 g(n) \)" to indicate that \( f(n) \) is a member of \( \Theta g(n) \). Instead, we will usually write "\( f(n) / \Theta g(n) \)" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

Figure 3.1(a) gives an intuitive picture of functions \( f(n) \) and \( g(n) \), where \( f(n) / \Theta g(n) \). For all values of \( n \) at and to the right of \( n_0 \), the value of \( f(n) \) lies at or above \( c_1 g(n) \) and at or below \( c_2 g(n) \). In other words, for all \( n \), \( f(n) / \Theta g(n) \). We say that \( g(n) \) is an asymptotically tight bound for \( f(n) \).

The definition of \( \Theta g(n) \) requires that every member \( f(n) / \Theta g(n) \) be asymptotically nonnegative, that is, that \( f(n) \) be nonnegative whenever \( n \) is sufficiently large. (An asymptotically positive function is one that is positive for all sufficiently large \( n \).) Consequently, the function \( g(n) \) itself must be asymptotically nonnegative, or else the set \( \Theta g(n) \) is empty. We shall therefore assume that every function used within \( \Theta \)-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

(from Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms)
\[ f(x) = \frac{3x^2 + 7x - 10}{2} \]

\[ 2g(x) = 2x^2 \]
technically, \( f = O(g) \) does not imply a asymptotically \emph{tight bound}

e.g., \( n = O(n^2) \) is true, but there is no constant \( c \) such that \( cn^2 \) will approximate the growth of \( n \), as \( n \) gets large
but in this class we will use big-O notation
to signify asymptotically tight bounds
i.e., there are constants $c_1$, $c_2$ such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for } n \geq n_0$$

(there’s another notation: $\Theta$ — big-theta
— but we’re avoiding the formalism)
asymptotically tight bound: $g$ “sandwiches” $f$

(from Cormen, Leiserson, Riest, and Stein, Introduction to Algorithms)
So far, we've seen:

- binary search = $O(\log n)$
- factorial, linear search = $O(n)$
- insertion sort = $O(n^2)$
```python
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)
```

\[ = O(?) \]
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)

Always a fixed (constant) number of LOC executed, regardless of input.

= O(?)
```python
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)
```

Always a fixed (constant) number of LOC executed, regardless of input.

\[ T(n) = C = O(1) \]
\[
\text{def } \text{foo}(m, n): \\
\quad \text{for } _\_ \text{ in range}(m): \\
\quad \quad \text{for } _\_ \text{ in range}(n): \\
\quad \quad \quad \text{pass}
\]

\[
= \mathcal{O}(?)
\]
```python
def foo(m, n):
    for _ in range(m):
        for _ in range(n):
            pass
```

\[= O(m \times n)\]
def foo(n):
    for _ in range(n):
        for _ in range(n):
            for _ in range(n):
                pass

= \text{O}(?)
def foo(n):
    for _ in range(n):
        for _ in range(n):
            for _ in range(n):
                pass

= O(n^3)
\[
\begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
a_{20} & a_{21} & a_{22}
\end{bmatrix} \times \begin{bmatrix}
b_{00} & b_{01} & b_{02} \\
b_{10} & b_{11} & b_{12} \\
b_{20} & b_{21} & b_{22}
\end{bmatrix} = \begin{bmatrix}
c_{00} & c_{01} & c_{02} \\
c_{10} & c_{11} & c_{12} \\
c_{20} & c_{21} & c_{22}
\end{bmatrix}
\]

\[c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{in}b_{nj}\]

i.e., for \(n \times n\) input matrices, each result cell requires \(n\) multiplications
def square_matrix_multiply(a, b):
    dim = len(a)
    c = [[0] * dim for _ in range(dim)]
    for row in range(dim):
        for col in range(dim):
            for i in range(dim):
                c[row][col] += a[row][i] * b[i][col]
    return c

= O(dim^3)
using “brute force” to crack an $n$-bit password $= O(?)$
1 character (8 bits) \(\{\) possible values \(\}
\[
\begin{align*}
00000000 \\
00000001 \\
00000010 \\
00000011 \\
00000100 \\
00000101 \\
00000110 \\
00000111 \\
00001000 \\
00001001 \\
00001010 \\
00001011 \\
00001100 \\
00001101 \\
00001110 \\
00001111 \\
00010000 \\
00010001 \\
00010010 \\
00010011 \\
00010100 \\
00010101 \\
00010110 \\
00010111 \\
00011000 \\
00011001 \\
00011010 \\
00011011 \\
00011100 \\
00011101 \\
00011110 \\
00011111 \\
00100000 \\
00100001 \\
00100010 \\
00100011 \\
00100100 \\
00100101 \\
00100110 \\
00100111 \\
00101000 \\
00101001 \\
00101010 \\
00101011 \\
00101100 \\
00101101 \\
00101110 \\
00101111 \\
00110000 \\
00110001 \\
00110010 \\
00110011 \\
00110100 \\
00110101 \\
00110110 \\
00110111 \\
00111000 \\
00111001 \\
00111010 \\
00111011 \\
00111100 \\
00111101 \\
00111110 \\
00111111 \\
11100000 \\
11100001 \\
11100010 \\
11100011 \\
11100100 \\
11100101 \\
11100110 \\
11100111 \\
11101000 \\
11101001 \\
11101010 \\
11101011 \\
11101100 \\
11101101 \\
11101110 \\
11101111 \\
11110000 \\
11110001 \\
11110010 \\
11110011 \\
11110100 \\
11110101 \\
11110110 \\
11110111 \\
11111000 \\
11111001 \\
11111010 \\
11111011 \\
11111100 \\
11111101 \\
11111110 \\
11111111
\end{align*}
\]

= \(O(?)\)
using “brute force” to crack an $n$-bit password $= O(2^n)$
<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
<td>Compute discriminant</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log n)$</td>
<td>Binary search</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>$O(n \log n)$</td>
<td>Merge sort (coming!)</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(n^2)$</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(n^3)$</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^c)$</td>
<td>Generally, $c$ nested loops over $n$ items</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(c^n)$</td>
<td>Brute forcing an $n$-bit password</td>
</tr>
<tr>
<td>Factorial</td>
<td>$O(n!)$</td>
<td>“Traveling salesman” problem</td>
</tr>
</tbody>
</table>

Common order of growth classes