Introduction to Algorithms

Part II

Today

- How do we evaluate/compare algorithms
  - Bubblesort/Mergesort revisited
  - Complexity family
  - Function growth
- New algorithms/algorithmic pattern
  - Finding power set
  - Search

How do we evaluate an algorithm?

- The wrong question: is *mergesort* better than *bubblesort*?
- The right question: ***when*** is *mergesort* better than *bubblesort*?

How do we evaluate an algorithm?

- The wrong question: is *mergesort* better than *bubblesort*?
- The right question: ***when*** is *mergesort* better than *bubblesort*?

Metric of interests:

- **Time**
- Space
- Simplicity/elegance/readability (“kissability” ☺)

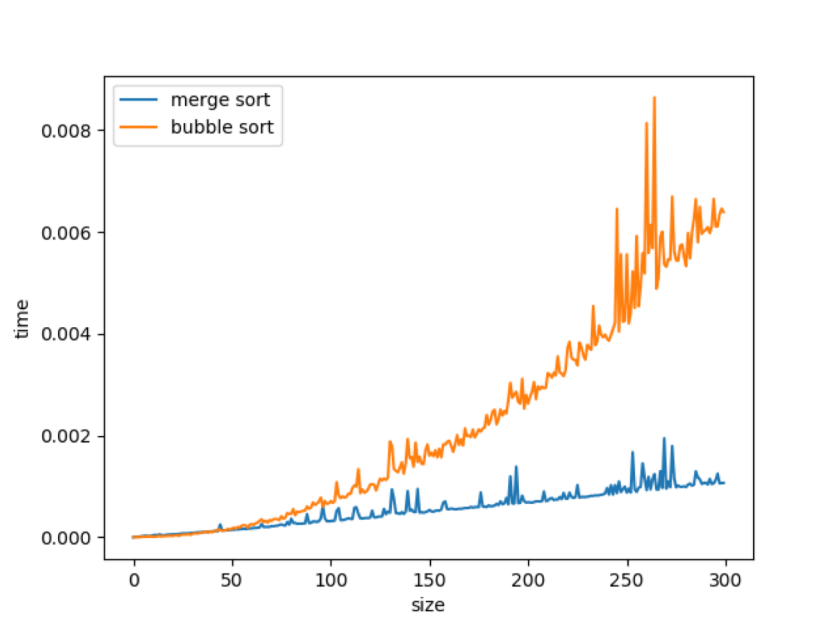
Timing the running time

- Try it out on the sorting code, compare merge sort and bubble sort

```
# Function runtime timer
def check_func_time(func, *args):
    start_time = time.time()
    func(*args)
    end_time = time.time()
    return end_time - start_time
```

In Python, a function can be an argument to another function.

- The check_func_time() returns the runtime that func takes.



Count the # of pair-comparisons

For a list with n numbers: (in the worst case)

- Bubble sort: (n-1) + (n-2) + … + 1

```
11 # Optimized bubble sort function
12 def bubble_sort(my_list):
13     N = len(my_list)
14     for i in range(1,N):
15         swapped = False
16         for j in range(0,N-i):
17             if my_list[j] > my_list[j+1]:
18                 my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
19                 swapped = True
20         if swapped == False:
21             break
22     return my_list
```

- Merge sort:

The Big-O notation & algorithm complexity

- One single metric that describes the algorithm's ***limiting behavior***
- ... as a function of input size *n*
- E.g.: bubblesort/mergesort as a function of list size

- How does the algorithm's time grow with *n*? **Order-of**...
  - O(1): constant, doesn't care (e.g. takes the *first* element of a list)
  - O(n): linear with input size (e.g. compute the *sum* of a list)

The Big-O notation & algorithm complexity

- Let f, g be real valued functions, both defined on some unbounded subset of the real positive numbers, such that g(x) is strictly positive for all large enough values of x.

$$f(x) = O(g(x))$$

$$\exists M \; and \; x_0, \;\; s.t. \; |f(x)| \leq M * g(x) \; for \; \forall x \geq x_0$$

Some important complexity classes

- $O(1)$: Constant complexity
- $O(\log n)$: Logarithmic complexity
- $O(n)$: Linear complexity
- $O(n \log n)$: Log-linear complexity
- $O(n^k)$: Polynomial complexity
- $O(C^n)$: Exponential complexity

Each represents a growth rate of a function family.

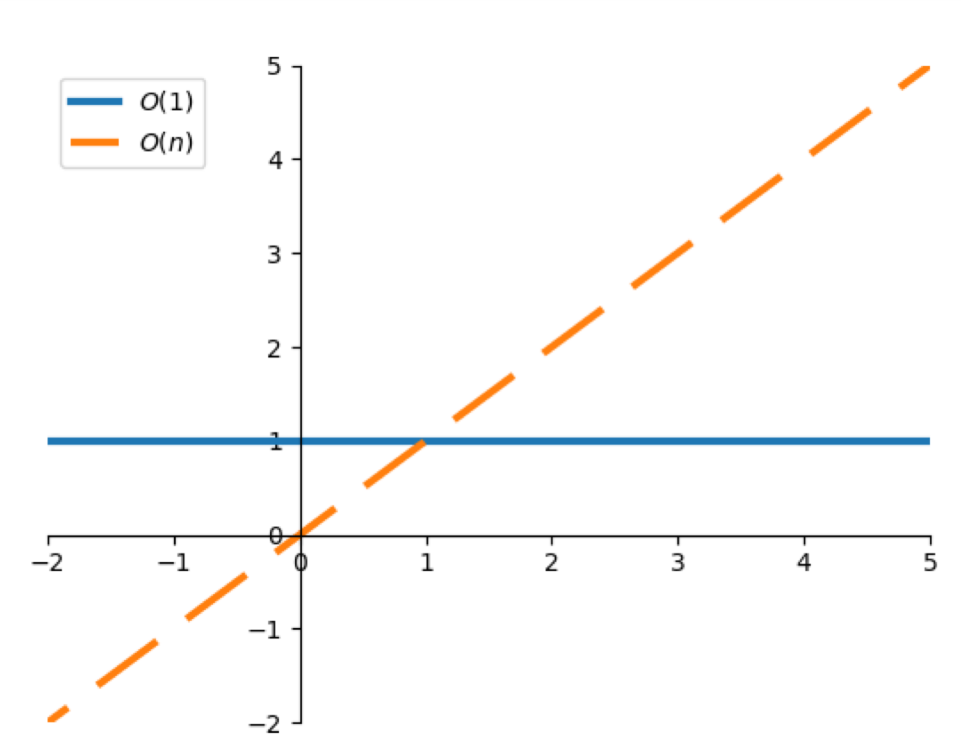
Constant complexity

O(1):

the complexity does not change with the increasing of n

```
def func(a_list):
    a = 1
    b = 2
    return a+b
```

Whatever the length of "a_list", the number of operations in func() are fixed.

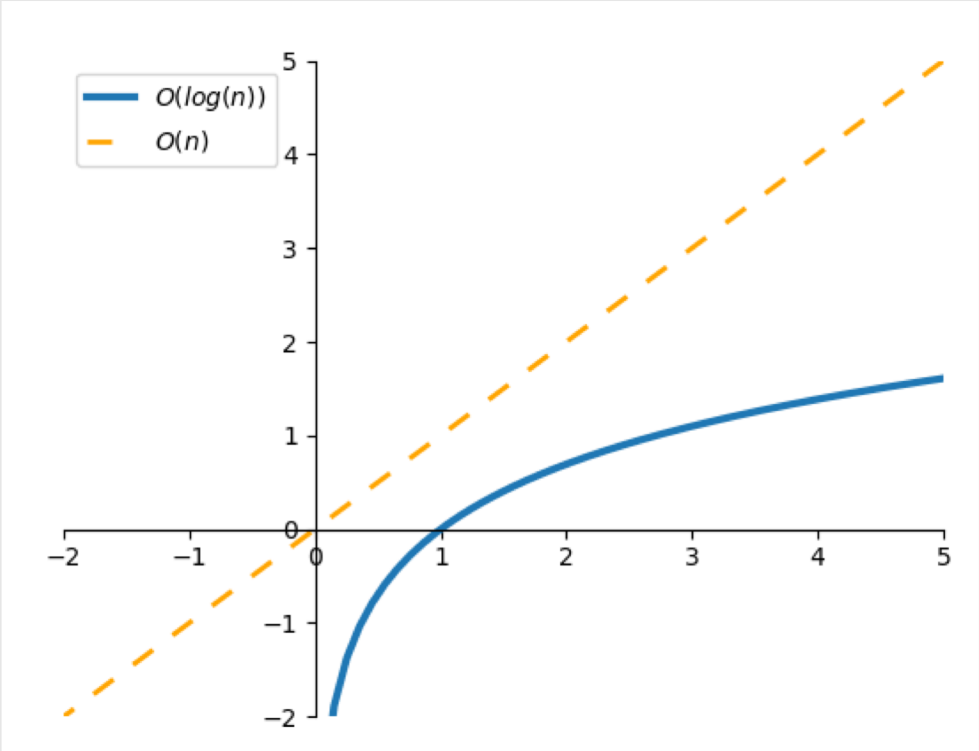


Logarithmic complexity

O(log(n)): the base of log does not matter. (Recall the logarithmic identities.)

```
def func(n):
    while n > 1:
        n /= 10
    return
```

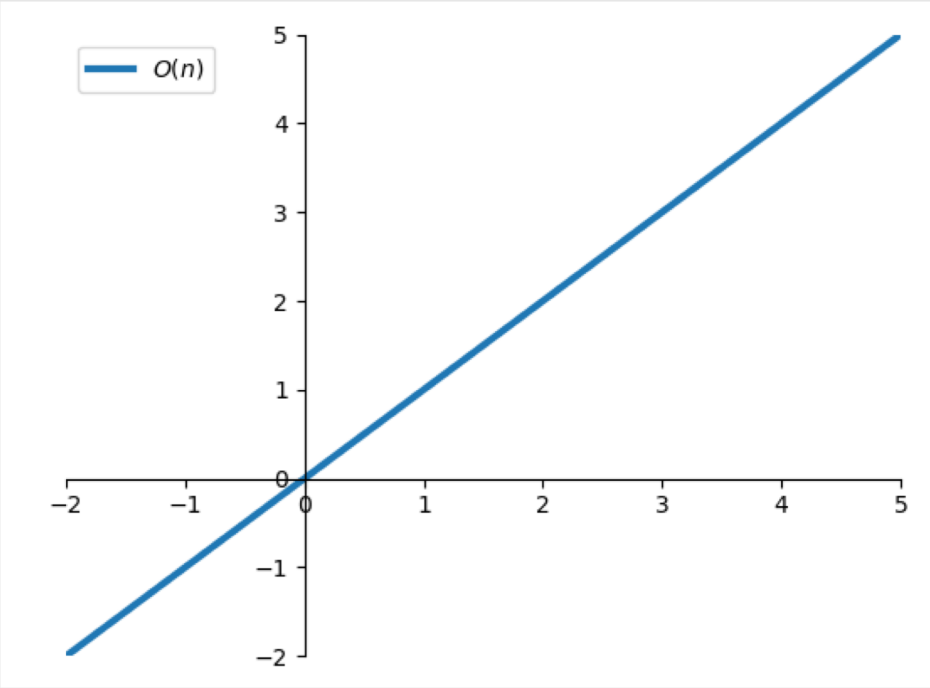
If n = 1000, how many iteration will the above while loop has?



Linear complexity

O(n)

```
def func(n):
    j = 0
    for i in range(n):
        j += 1
    return
```

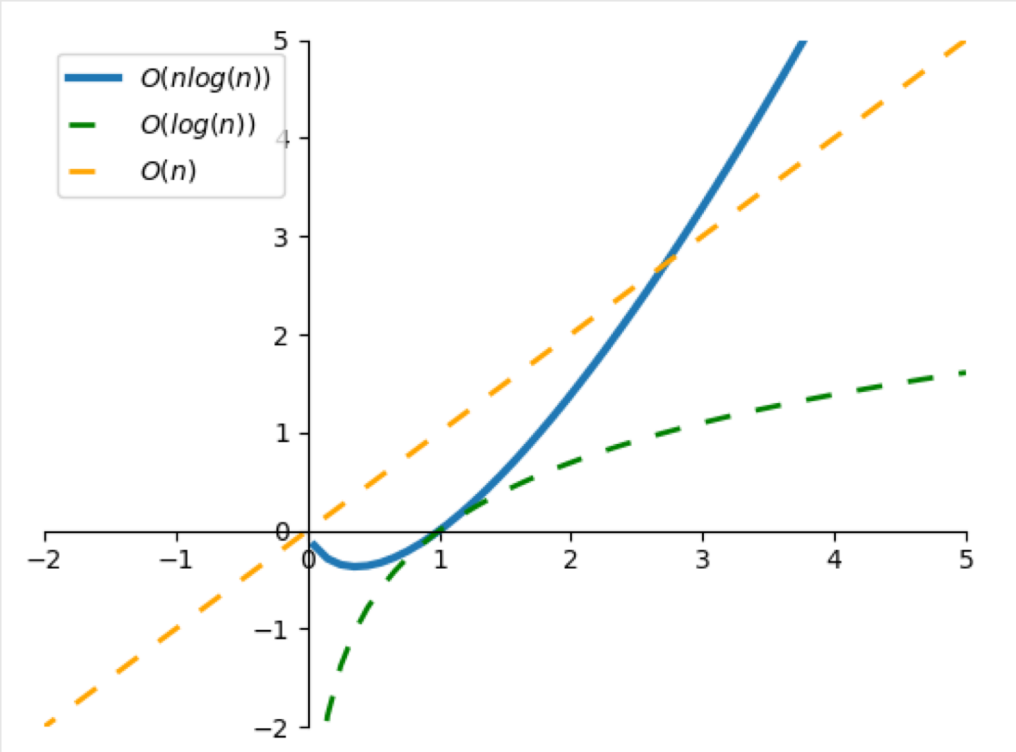


Log-linear complexity

```
def fun(n):
    while n>1:
        n = n//2
        m = n
        for i in range(m):
            a = 1
    return
```

Another example:

the merge sort.



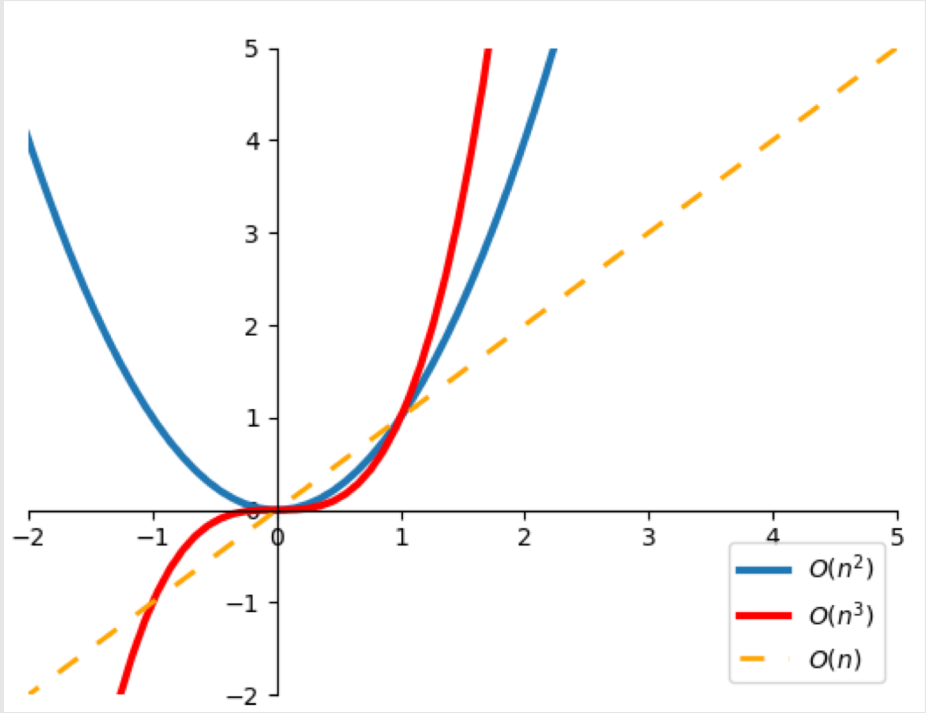
Polynomial complexity

$O(n^2)$

```
def func(n):
    for i in range(1, n):
        for j in range(1, n):
            print(i*j)
    return
```

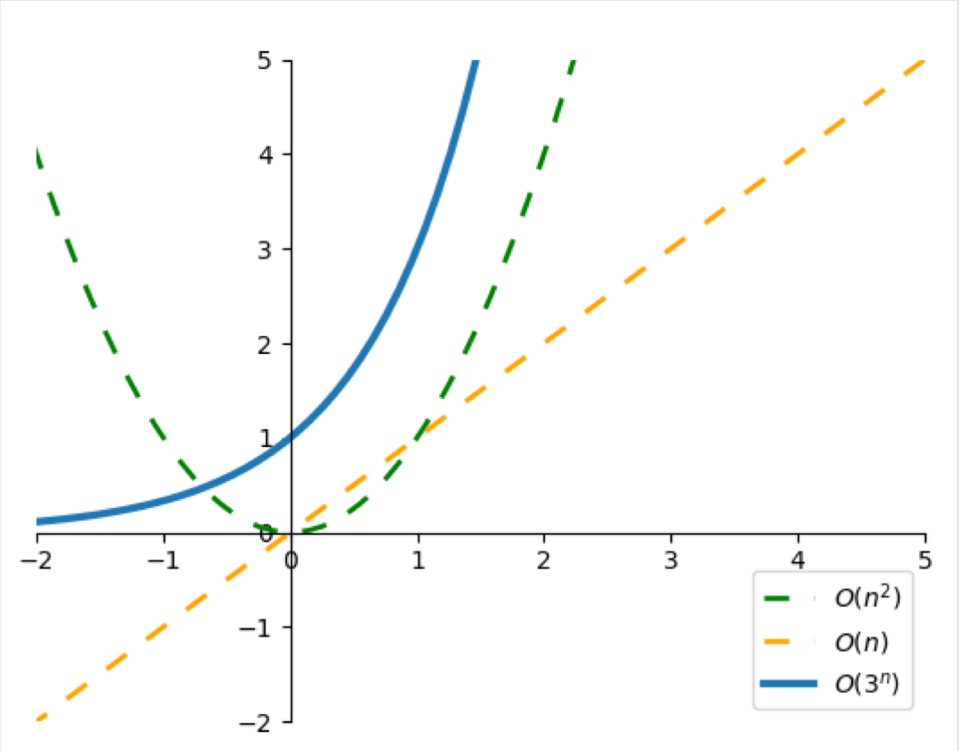
$O(n^3)$

```
def func(n):
    for i in range(1, n):
        for j in range(1, n):
            for k in range(1, n):
                print(i+j+k)
    return
```



Exponential complexity

Rarely a company is willing to pay for a program that requires exponential time to run.



How does it look like:

- Please plot the curves of x**2 and 2**x in [0, 10].
- Exponential growth and epidemics

Comparison of complexity classes

$O(C^n)$

$O(n^k)$

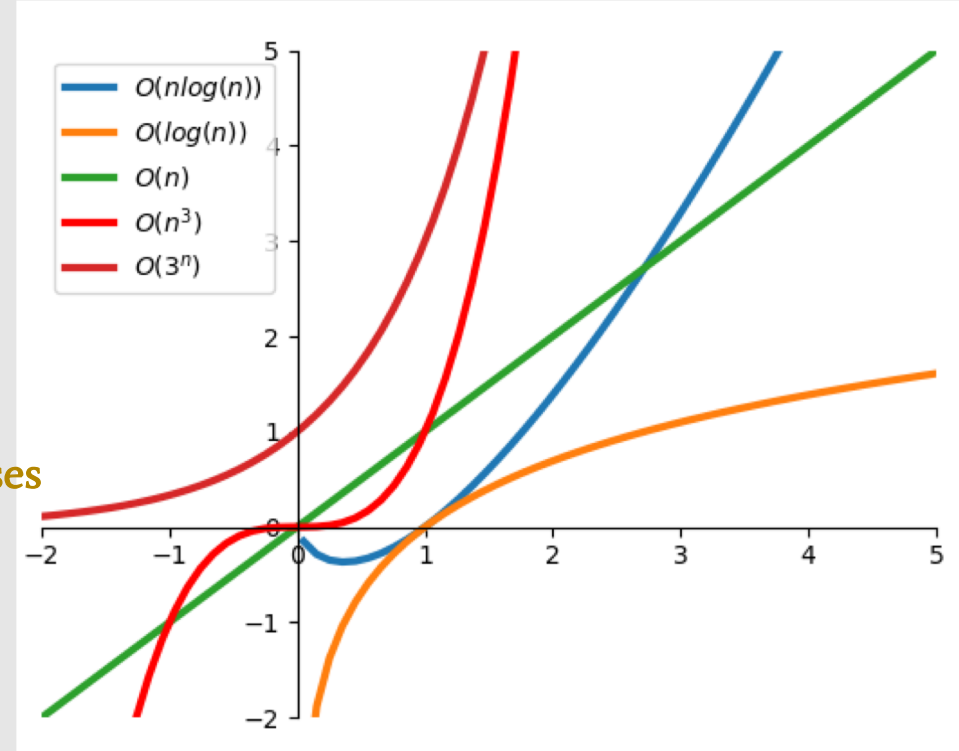
$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$

Grows faster!
Complexity increases



Complexity of real code

The complexity is:

```
def fact(n):
    """Assumes n is a natural number
       Returns n!"""
    answer = 1
    while n > 1:
        answer *=  n
        n -= 1
    return answer
```

$O(n^2)$

Complexity of real code

$O(1)$ ←

$O(n)$ ←

$O(n^2)$ ←

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0
    #Loop that takes constant time
    for i in range(1000):
        ans += 1
    print 'Number of additions so far', ans
    #Loop that takes time x
    for i in range(x):
        ans += 1
    print 'Number of additions so far', ans
    #Nested loops take time x**2
    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print 'Number of additions so far', ans
    return ans
```

The Big-O notation & algorithm complexity

An algorithm may have many phases. In practice, two simplifications:

- If a sum of several factors, only include the one that grows the largest
- Always drop a constant in a factor

$$f(n) = 9\log n + 5(\log n)^3 + 3n^2 + 2n^3 = O(n^3) \qquad n \to \infty$$

Complexity of real code

- This is $O(n^2)$
- We only care when problem size grows to infinity

Type equation here.

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0
    #Loop that takes constant time
    for i in range(1000):
        ans += 1
    print 'Number of additions so far', ans
    #Loop that takes time x
    for i in range(x):
        ans += 1
    print 'Number of additions so far', ans
    #Nested loops take time x**2
    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print 'Number of additions so far', ans
    return ans
```

Order of functions

$$\lim_{n\to\infty} \frac{n^4 + 2n^2 + 4}{n^4} = ???$$

- We say the function $f(n) = n^4 + 2n^2 + 4$ is $O(n^4)$ because the two is asymptotically the same
- What about $f(n) = 2^n + n^2$ in the big-O notation?
- When we say the complexity of an algorithm, we ask its order of growth as input size, therefore using big-O notation.

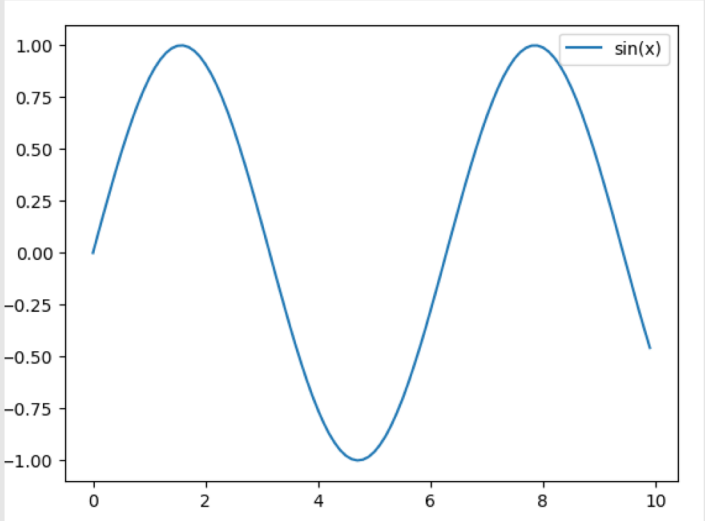
Functions: some examples

- Linear: $f(x) = x; f(x) = 3x + 5$
- Quadratic: $f(x) = x^2; f(x) = 4x^2 + 3x - 6$
- Cubical: $f(x) = x^3; f(x) = x^3 + 4x - 5$
- Logarithmic: $f(x) = \log_2(x)$;$f(x) = \log_2(8x + 9)$
- Log-linear: $f(x) = x\log_2(x)$
- Exponential: $f(x) = 4^x$

First, some coding warmup

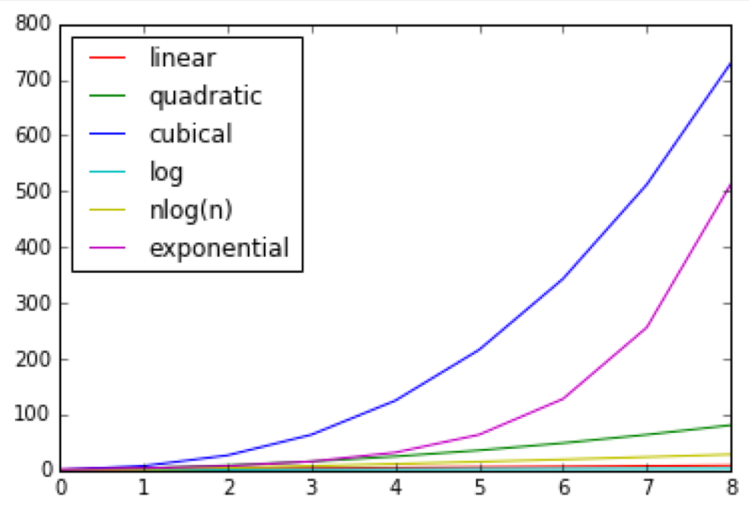
- Recall plotting a line in Python

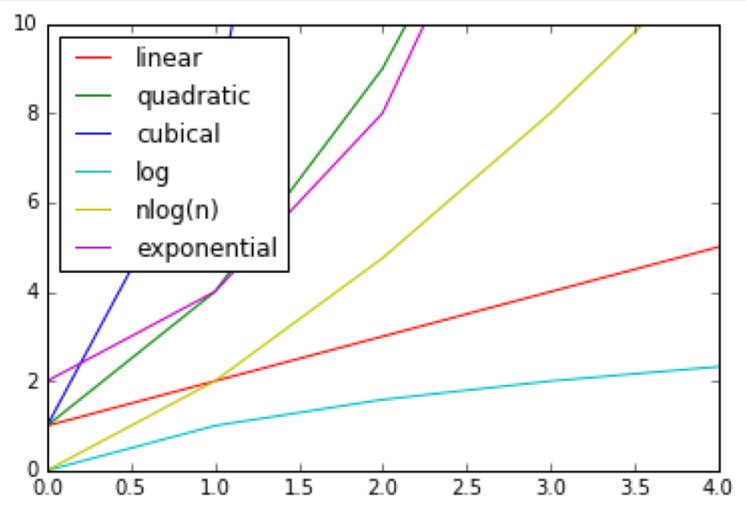
```
8 import matplotlib.pyplot as plt
9 from math import *
.0
.1 x = [i*0.1 for i in range(100)]
.2 y = [sin(t) for t in x]
.3 plt.plot(x, y, label='sin(x)')
.4 plt.legend()
```



First, some coding warmup

- Plotting a line in Python
- Exercise: complete function_plot_student.py





Bubblesort

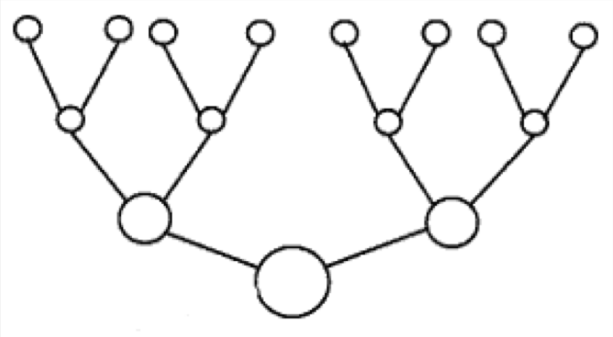
```
11 # Optimized bubble sort function
12 def bubble_sort(my_list):
13     N = len(my_list)
14     for i in range(1,N):
15         swapped = False
16         for j in range(0,N-i):
17             if my_list[j] > my_list[j+1]:
18                 my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
19                 swapped = True
20         if swapped == False:
21             break
22     return my_list
```

- Time complexity: $O(n^2)$

Mergesort

$O(\log(n))$ “layers”

```
26 def merge_sort(m):
27
28     if len(m) <= 1:
29         return m
30
31     middle = len(m) // 2
32     left = m[:middle]
33     right = m[middle:]
34
35     left = merge_sort(left)
36     right = merge_sort(right)
37     return merge(left, right)
```



Mergesort

$O(\log(n))$ passes

```
26 def merge_sort(m):
27
28     if len(m) <= 1:
29         return m
30
31     middle = len(m) // 2
32     left = m[:middle]
33     right = m[middle:]
34
35     left = merge_sort(left)
36     right = merge_sort(right)
37     return merge(left, right)
```

- Time: $O(n\log(n))$

$O(n)$ each pass

```
40 # Merge function definition
41 def merge(left, right):
42     result = []
43     left_idx, right_idx = 0, 0
44
45     while left_idx < len(left) and right_idx < len(right):
46         # change the direction of this comparison
47         # to change the direction of the sort
48         if left[left_idx] <= right[right_idx]:
49             result.append(left[left_idx])
50             left_idx += 1
51         else:
52             result.append(right[right_idx])
53             right_idx += 1
54
55     if left:
56         result.extend(left[left_idx:])
57     if right:
58         result.extend(right[right_idx:])
59
60     return result
```

Powerset: back to steam sale (again)

Question: why don't we just list all possible choices, and then rank them?

Answer: we cannot afford, when n is large!!

Powerset(L):

- L is a list of all items, return a list of list: all possible combinations of the items
  - Powerset(['a', 'b']): [[], ['a'], ['b'], ['a', 'b']]

```
In [6]: powerset_comb(list(range(3)))
Out[6]: [[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2]]
```

Powerset size is exponential

- $len(powerset(L)) \triangleq 2^{len(L)}$

- But, WHY? Can you prove it?

Proof by combination

- L has n items
- Put n coin on top of each item
- If a coin is heads-up, the item is in.

- Make a n-bit binary number (i.e. 000, 001, 010, ... 111)
- There are 2^n possible values. Therefore there are 2^n combinations.

Proof by induction

- If L has 0 items, then there is only 1 (i.e. 2^0) combinations

- Suppose L has n items, and powerset(L) has 2^n combinations
- Now adding a new item x into L (i.e. L.append(x)):
  - We can partition the new powerset into two kinds: with or without x
- Therefore L with n+1 items has $2\times(2^n)$, or 2^{n+1} combinations

Exercise

- Choose one of the approaches to implement powerset(L)
- Time the execution time and graph it, as a function of len(L)
- **Caution: don't use large list!**

```
 9 def timeit(f, *args):
10     t1 = time.time()
11     f(*args)
12     t2 = time.time()
13     return t2 - t1
```

Expected results: a list of list

ipython (Python)

```
In [5]: powerset_comb(range(4))
Out[5]:
[[],
 [3],
 [2],
 [2, 3],
 [1],
 [1, 3],
 [1, 2],
 [1, 2, 3],
 [0],
 [0, 3],
 [0, 2],
 [0, 2, 3],
 [0, 1],
 [0, 1, 3],
 [0, 1, 2],
 [0, 1, 2, 3]]
```

```
In [6]: powerset_add(range(4))
Out[6]:
[[],
 [0],
 [1],
 [0, 1],
 [2],
 [0, 2],
 [1, 2],
 [0, 1, 2],
 [3],
 [0, 3],
 [1, 3],
 [0, 1, 3],
 [2, 3],
 [0, 2, 3],
 [1, 2, 3],
 [0, 1, 2, 3]]
```

Coding tips: induction approach

- $L = [0, 1, 2, 3]$
- *pset* = [[], [0], [1], [0, 1]]; now you are considering adding "2"
- The result should be [[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2]]
- Mind the mutable list, use deepcopy!

```
18 from copy import deepcopy
19 def powerset_add(l):
20     pset = [[]]
21     for item in l:
22 # note the use of deepcopy, list is mutable!
23         new_subset = deepcopy(pset)
24 # implement logic to add new subset
```

Complete the code (~3lines each)

```
18 from copy import deepcopy
19 def powerset_add(l):
20     pset = [[]]
21     for item in l:
22 # note the use of deepcopy, list is mutable!
23         new_subset = deepcopy(pset)
24 # implement logic to add new subset
25 # for each existing list, add the item, then you need to
26 # extend pset with new_subsets
27         pass
28 # your code ends here
29     return pset
```

Complete the code (~3lines)

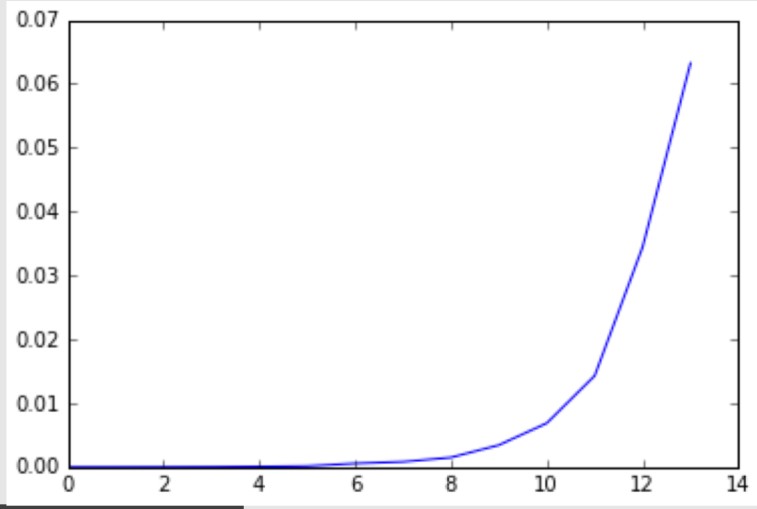
```
39 def powerset_comb(l):
40     pset = []
41     total_items = len(l)
42     for i in range(2 ** total_items):
43         is_in = get_bin_str(i, total_items)
44         subset = []
45 # implement logic to insert into the subset below
46 # if is_in[i] is '1', then i-th item is in it
47         pass
48 # your code ends here
49         pset.append(subset)
50     return pset
```

Convert an integer from 0 to 2^n to a binary code:

Makes one choice:

Timing a function

```
10 # timing the execution of a function
11 def timeit(f, *args):
12     t1 = time.time()
13     f(*args)
14     t2 = time.time()
15     return t2 - t1
```




```
67     for i in range(14):
68         my_list = list(range(i + 1))
69         takes = timeit(powerset_comb, my_list)
70         exec_time.append(takes)
```

Steam sale: complexity compared

	Value ($)	Size(GB)
Civilization V	50	10
CS:GO	30	15
The Elder Scrolls V	20	5
Torchlight II	20	2
Age of Empires II	35	5
Banished	20	0.5

The *greedy* algorithmic approach is $O(n\log n)$

The brute force search is $O(n2^n)$

Computer science: algorithm research

- *Very* important branch
- Algorithms are ***building blocks*** of many important applications
- A related area is *complexity analysis*

Typical loop:

- Analyze and prove the complexity of the optimal solution
- Then propose one that works well under a certain context

Searching

Linear search: the brute-force way

- What's the complexity of exhaustive search?

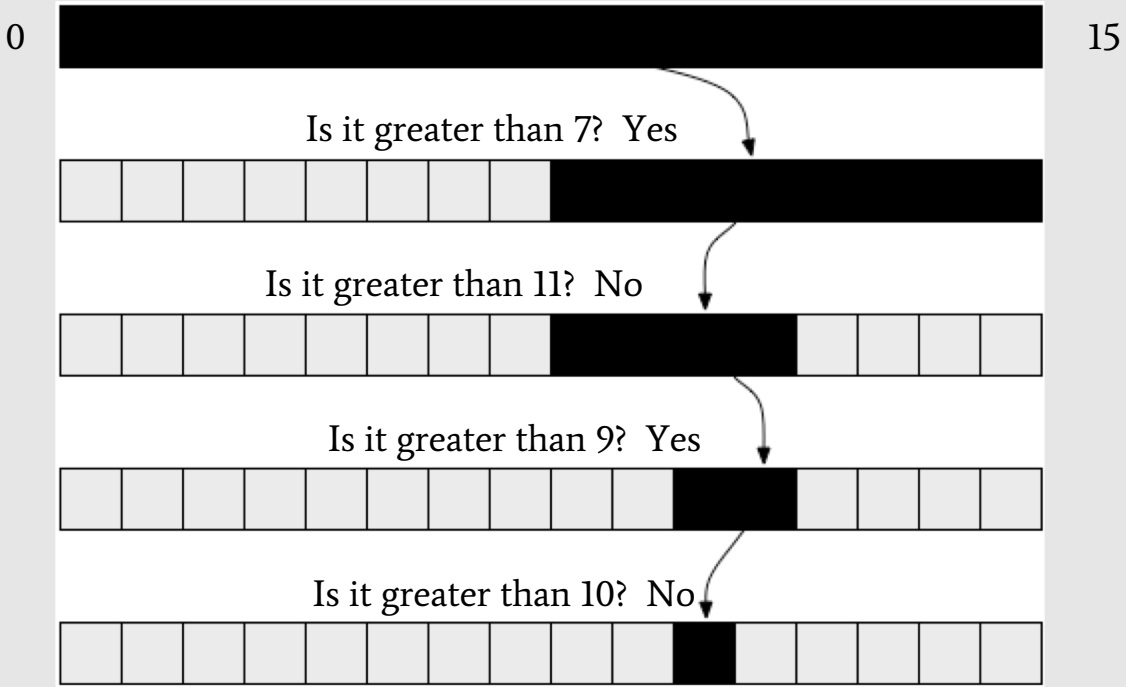
```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

The complexity is linear, i.e., O(n). (assuming the operations inside the loop can be done in constant time.)

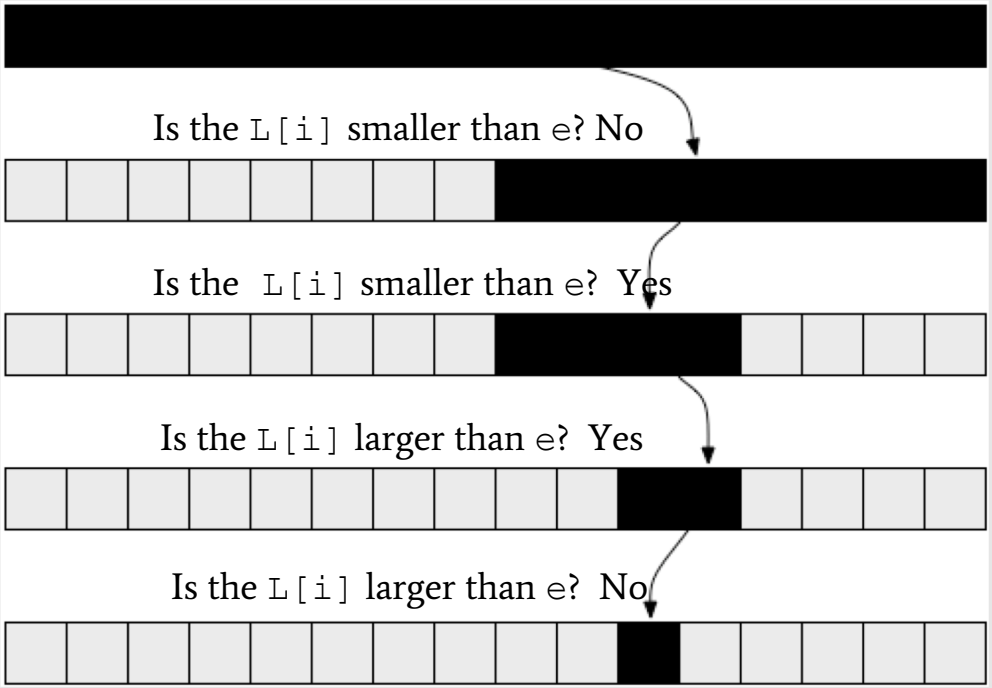
This looks good … but when there are millions of items, you have to wait for minutes or even hours.

If the list is sorted, can you do better?

Search a sorted list: guess a number



Binary search: a recursive search



The search task can be splitted into several sub-tasks; each is conducted by the same way (assume the list is ordered):

1. Pick an index, `i`, that divides the list `L` roughly in half.
2. Ask if `L(i) == e`
3. If not, ask whether `L[i]` is larger or smaller than `e`.
4. Depending on the answer in step 3, search either the left or right half of `L` for `e`.

Binary search: if the list is sorted

What is the complexity of the binary search?

The algorithm stops when the length of the interval is 1.

Each time the interval shrinks by a half.

The time complexity is O(log(n)).

```
def search(L, e):
    """Assumes L is a list, the elements of which are in
          ascending order.
       Returns True if e is in L and False otherwise"""

    def bSearch(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bSearch(L, e, low, mid - 1)
        else:
            return bSearch(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

When to use binary search?

- Given an unsorted list of length n
- You will search m times
- What's the complexity using naive search, and binary search?

When to use binary search?

- Given an unsorted list of length n
- You will search m times
- What's the complexity using naive search, and binary search?
  - Naive: $O(m * n)$
  - Binary search: $O(n\log n + m\log n)$
  - Typically, you do search many, many times

Search complexity revisited

- m: number of searches
- n: the size the list

Brute-force search	Binary search
$O(m \times n)$	$O(\mathrm{n} \times \log n + m \times \log n) \approx$ $O(m \log n)$ # if n << m

Summary

- How do we evaluate/compare algorithms
  - Function growth
  - Complexity family
  - Bubblesort/Mergesort revisited
- New algorithms/algorithmic pattern
  - Finding powerset
  - Search

Readings

Object-oriented Programming

- MIT Ch. 8 (OOP) 8.1, 8.3, 8.4
- Python OOP Tutorial 1: Classes and Instances, & tutorial 2, 3

Dynamic Programming

- MIT Ch. 17 (Knapsack Probs.) 17
- MIT Ch. 18 (Dynamic Prog.). 18.1, 18.2