UNDERSTANDING PROGRAM EFFICIENCY

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- computers are fast and getting faster – so maybe efficient programs don't matter?
  - but data sets can be very large
  - thus, simple solutions may simply not scale with size in acceptable manner
- so how could we decide which option for program is most efficient?

- separate **time and space efficiency** of a program
- tradeoff between them – will focus on time efficiency

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

Challenges in understanding efficiency of solution to a computational problem:

- a program can be **implemented in many different ways**
- you can solve a problem using only a handful of different **algorithms**
- would like to separate choices of implementation from choices of more abstract algorithm

HOW TO EVALUATE EFFICIENCY OF PROGRAMS

- measure with a **timer**
- **count** the operations
- abstract notion of **order of growth**

will argue that this is the most appropriate way of assessing the impact of choices of algorithm in solving a problem; and in measuring the inherent difficulty in solving a problem

TIMING A PROGRAM

- use time module
- recall that importing means to bring in that class into your own file
- **start** clock
- **call** function
- **stop** clock

```
import time

def c_to_f(c):
    return c*9/5 + 32

t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
Print("t =", t, ":", t1, "s,”)
```

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- running time **varies between algorithms**
- running time **varies between implementations**
- running time **varies between computers**
- running time is **not predictable** based on small inputs

- time varies for different inputs but cannot really express a relationship between inputs and time

COUNTING OPERATIONS

- assume these steps take **constant time**:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32
```

3 ops

```
def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

1 op

loop x times

1 op

2 ops

mysum → 1+3x ops

COUNTING OPERATIONS IS BETTER, BUT STILL…

- GOAL: to evaluate different algorithms
- count **depends on algorithm** ✓
- count **depends on implementations** ✗
- count **independent of computers** ✓
- no real definition of **which operations** to count ✗

- count varies for different inputs and can come up with a relationship between inputs and the count ✓

STILL NEED A BETTER WAY

- timing and counting **evaluate implementations**
- timing **evaluates machines**

- want to **evaluate algorithm**
- want to **evaluate scalability**
- want to **evaluate in terms of input size**

NEED TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- want to express **efficiency in terms of input**, so need to decide what your input is
- could be an **integer**
  -- `mysum(x)`
- could be **length of list**
  -- `list_sum(L)`
- **you decide** when multiple parameters to a function
  -- `search_for_elmt(L, e)`

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- a function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- when `e` is **first element** in the list → BEST CASE
- when `e` is **not in list** → WORST CASE
- when **look through about half** of the elements in list → AVERAGE CASE
- want to measure this behavior in a general way

BEST, AVERAGE, WORST CASES

- suppose you are given a list `L` of some length `len(L)`
- **best case**: minimum running time over all possible inputs of a given size, `len(L)`
  - constant for `search_for_elmt`
  - first element in any list
- **average case**: average running time over all possible inputs of a given size, `len(L)`
  - practical measure
- **worst case**: maximum running time over all possible inputs of a given size, `len(L)`
  - linear in length of list for `search_for_elmt`
  - must search entire list and not find it

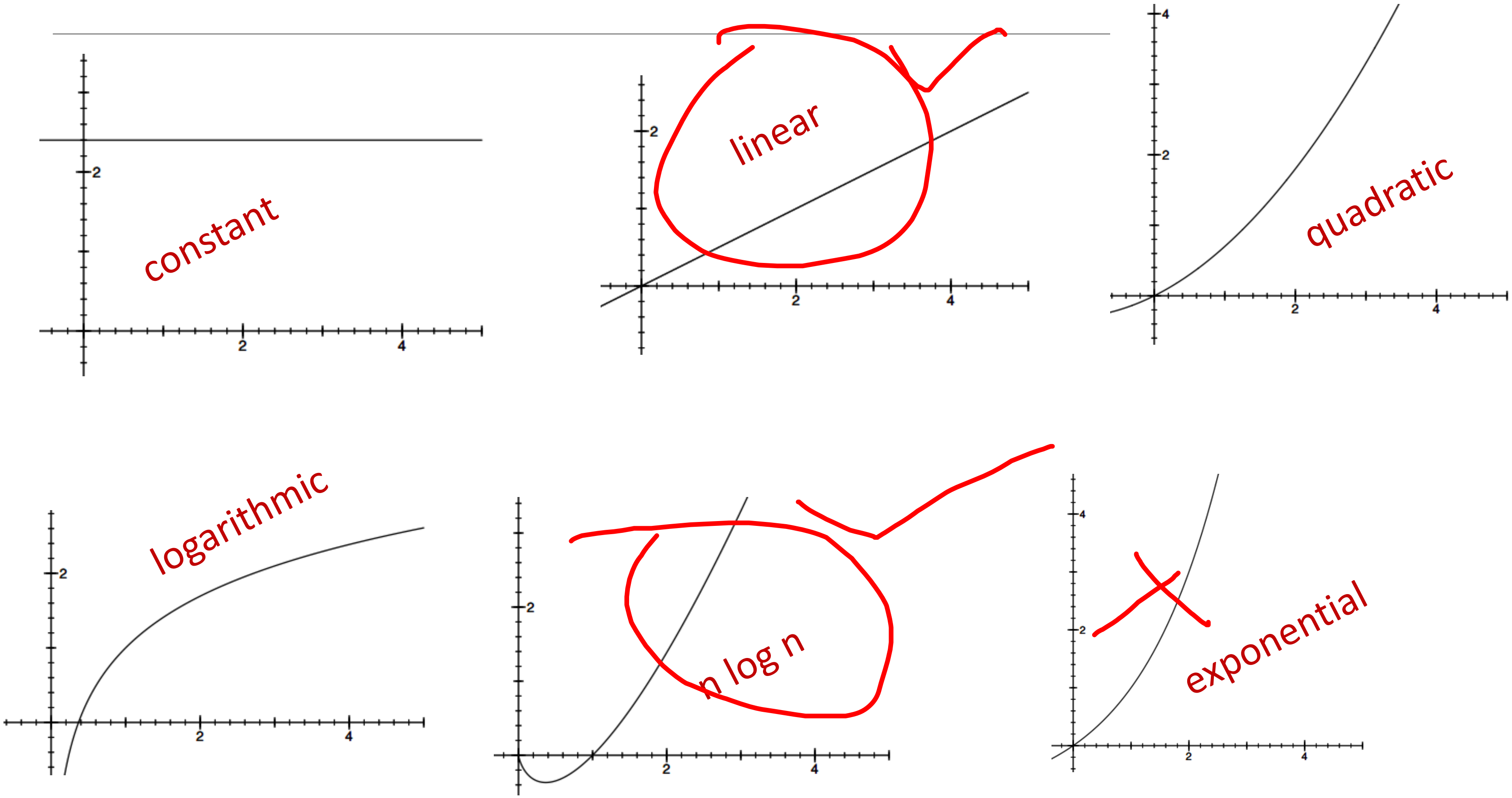
generally will focus on this case

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)

TYPES OF ORDERS OF GROWTH



MEASURING ORDER OF GROWTH: BIG OH NOTATION

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth

- **Big Oh or O()** is used to describe worst case
  - worst case occurs often and is the bottleneck when a program runs
  - express rate of growth of program relative to the input size
  - evaluate algorithm not machine or implementation

EXACT STEPS vs O()

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

temp = n-1
n = temp

- computes factorial
- number of steps: $1 + 5n + 1$
- worst case asymptotic complexity: $O(n)$
  - ignore additive constants
  - ignore multiplicative constants

SIMPLIFICATION EXAMPLES

- drop constants and multiplicative factors
- focus on **dominant terms**

O(n²) : $n^2 + 2n + 2$

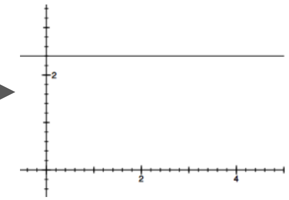
O(n²) : $n^2 + 100000n + 3^{1000}$

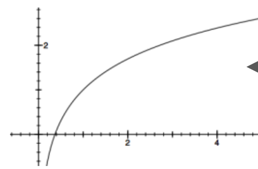
O(n) : $\log(n) + n + 4$

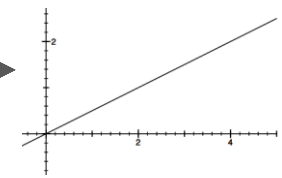
O(n log n) : $0.0001*n*\log(n) + 300n$

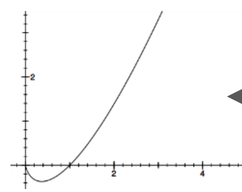
O(3ⁿ) : $2n^{30} + 3^n$

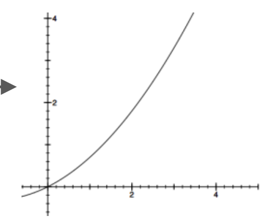
COMPLEXITY CLASSES ORDERED LOW TO HIGH

$O(1)$: constant → 

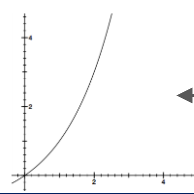
$O(\log n)$:  ← logarithmic

$O(n)$: linear → 

$O(n \log n)$:  ← loglinear

$O(n^c)$: polynomial → 

c is a constant

$O(c^n)$:  ← exponential

ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
  - analyze statements inside functions
  - apply some rules, focus on dominant term

Law of Addition for O():

- used with **sequential** statements
- O(f(n)) + O(g(n)) is O(f(n) + g(n))
- for example,

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

is $O(n) + O(n*n) = O(n+n^2) = O(n^2)$ because of dominant term

ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
  - analyze statements inside functions
  - apply some rules, focus on dominant term

Law of Multiplication for O():

- used with **nested** statements/loops
- O(f(n)) * O(g(n)) is O(f(n) * g(n))
- for example,

```
for i in range(n):
    for j in range(n):
        print('a')
```

is $O(n)*O(n) = O(n*n) = O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

COMPLEXITY CLASSES

- *$O(1)$* denotes constant running time
- *$O(\log n)$* denotes logarithmic running time
- *$O(n)$* denotes linear running time
- *$O(n \log n)$* denotes log-linear running time
- *$O(n^c)$* denotes polynomial running time (c is a constant)
- *$O(c^n)$* denotes exponential running time (c is a constant being raised to a power based on size of input)

CONSTANT COMPLEXITY

- complexity independent of inputs
- very few interesting algorithms in this class, but can often have pieces that fit this class
- can have loops or recursive calls, but number of iterations or calls independent of size of input

LOGARITHMIC COMPLEXITY

- complexity grows as log of size of one of its inputs
- example:
  - bisection search
  - binary search of a list

Efficient

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10  ⟶ Log
    return result
```

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return result
```

const

only have to look at loop as no function calls

within while loop, constant number of steps

how many times through loop?

- how many times can one divide i by 10?
- *O(log(i))*

LINEAR COMPLEXITY

- searching a list in sequence to see if an element is present
- add characters of a string, assumed to be composed of decimal digits

```
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

const

n loops

- *O(len(s))*

LINEAR COMPLEXITY

- complexity can depend on number of recursive calls

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

- number of times around loop is n
- number of operations inside loop is a constant
- overall just *O(n)*

O() FOR RECURSIVE FACTORIAL

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

2

- computes factorial recursively
- if you time it, may notice that it runs a bit slower than iterative version due to function calls
- still **O(n)** because the number of function calls is linear in n
- **iterative and recursive factorial** implementations are the **same order of growth**

LOG-LINEAR COMPLEITY

- many practical algorithms are log-linear
- very commonly used log-linear algorithm is merge sort
- will return to this

POLYNOMIAL COMPLEXITY

- most common polynomial algorithms are quadratic, $O(n^2)$ i.e., complexity grows with square of size of input
- commonly occurs when we have nested loops or recursive function calls

QUADRATIC COMPLEXITY

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

QUADRATIC COMPLEXITY

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

outer loop executed len(L1) times

each iteration will execute inner loop up to len(L2) times

*O(len(L1)*len(L2))*

worst case when L1 and L2 same length, none of elements of L1 in L2

$O(len(L1)^2)$

QUADRATIC COMPLEXITY

find intersection of two lists, return a list with each element appearing only once

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

QUADRATIC COMPLEXITY

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

first nested loop takes *len(L1)*len(L2)* steps

second loop takes at most *len(L1)* steps

latter term overwhelmed by former term

*O(len(L1)*len(L2))*

O() FOR NESTED LOOPS

```
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

- computes n^2 very inefficiently
- when dealing with nested loops, **look at the ranges**
- nested loops, **each iterating n times**
- **$O(n^2)$**

Most expensive/ indeed many programs are exponential

EXPONENTIAL COMPLEXITY

- recursive functions where more than one recursive call for each size of problem
  - Towers of Hanoi
- many important problems are inherently exponential
  - unfortunate, as cost can be high
  - will lead us to consider approximate solutions more quickly

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]] #list of empty list
    smaller = genSubsets(L[:-1]) # all subsets without
last element
    extra = L[-1:] # create a list of just last element
    new = []
    for small in smaller:
        new.append(small+extra)  # for all smaller
solutions, add one with last element
    return smaller+new  # combine those with last
element and those without
```

Solution of smallest problem/ recursive method

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

assuming append is constant time

time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

but important to think about size of smaller

know that for a set of size k there are 2^k cases

so to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps

math tells us this is *$O(2^n)$*

COMPLEXITY CLASSES

- *$O(1)$* denotes constant running time
- *$O(\log n)$* denotes logarithmic running time
- *$O(n)$* denotes linear running time
- $\boxed{O(n \log n) \text{ denotes log-linear running time}}$
- *$O(n^c)$* denotes polynomial running time (c is a constant)
- *$O(c^n)$* denotes exponential running time (c is a constant being raised to a power based on size of input)

EXAMPLES OF ANALYZING COMPLEXITY

TRICKY COMPLEXITY

```
def h(n):
    """ assume n an int >= 0 """
    answer = 0
    s = str(n)
    for c in s:
        answer += int(c)
    return answer
```

linear O(len(s))
but what in terms
of input n?

- adds digits of a number together
- tricky part
  - convert integer to string
  - iterate over **length of string**, not magnitude of input n
  - think of it like dividing n by 10 each iteration
- **O(log n)** – base doesn't matter

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant O(1)

constant O(1)

linear O(n)

constant O(1)

- Best case:

  O(1)

- Worst case:

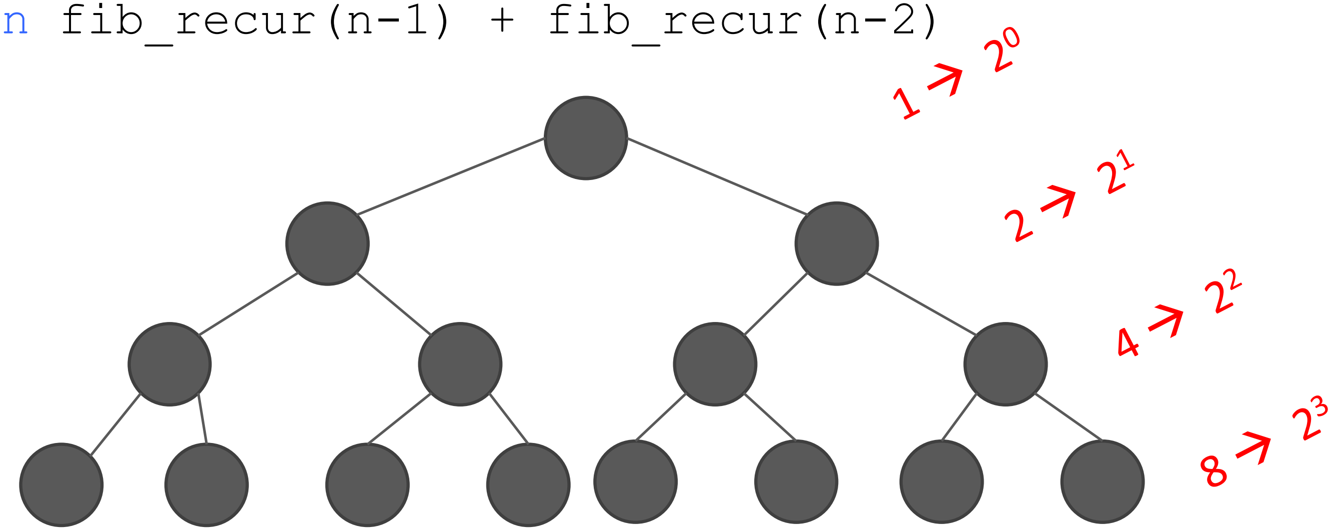
  O(1) + O(n) + O(1) → **O(n)**

COMPLEXITY OF RECURSIVE FIBONACCI

```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

$O(2^n)$



WHEN THE INPUT IS A LIST…

```
def sum_list(L):
    total = 0
    for e in L:
        total = total + e
    return total
```

- O(n) where n is the length of the list
- O(len(L))
- must **define what size of input means**
  - previously it was the magnitude of a number
  - here, it is the length of list

BIG OH SUMMARY

- compare **efficiency of algorithms**
  - notation that describes growth
  - **lower order of growth** is better
  - independent of machine or specific implementation

- use Big Oh
  - describe order of growth
  - **asymptotic notation**
  - **upper bound**
  - **worst case** analysis

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)