PROGRAM EFFICIENCY

(download slides and .py files to follow along!)

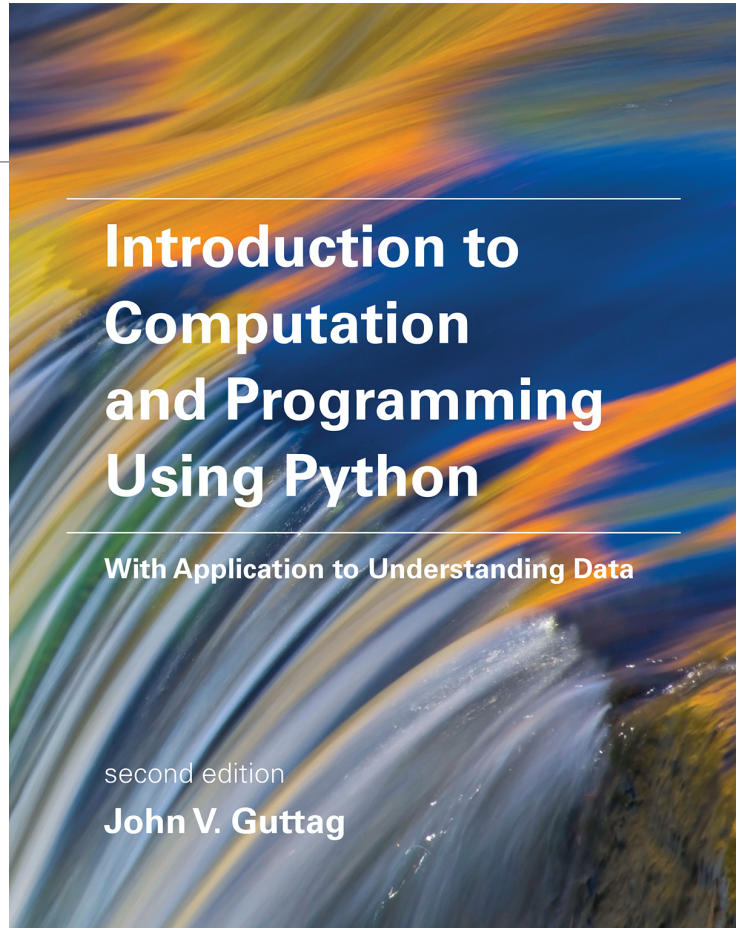
6.0001 LECTURE 9

TODAY

- Formally evaluate programs
- Efficiency in time
- Orders of growth, big Oh notation
- Examples of different complexity cases

Assigned Reading

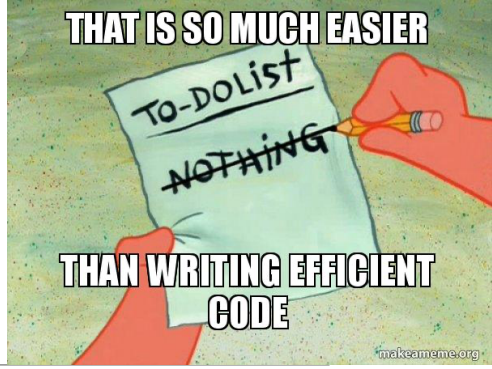
- Today
  - Chapter 9
  - 10.1 – 10.2
- Monday
  - 10.3
  - Chapter 11



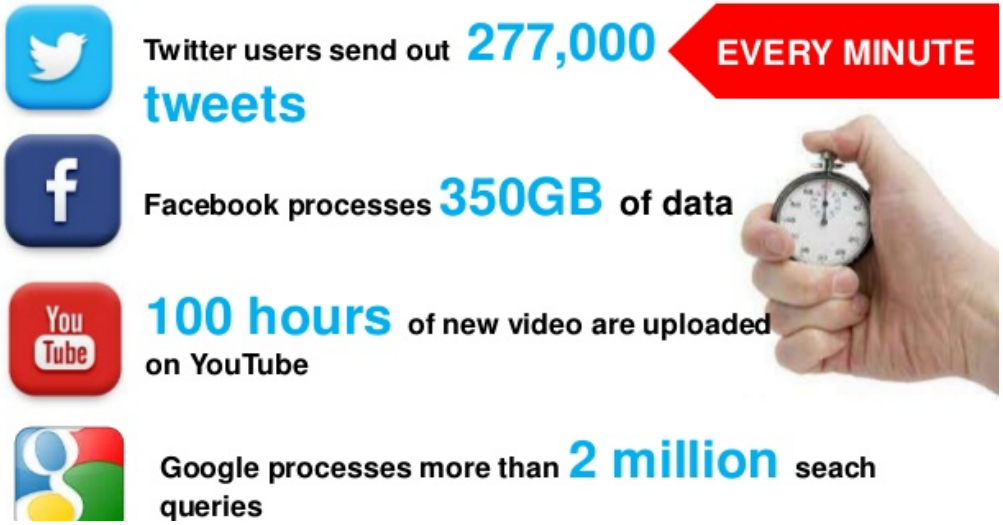
https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

PROGRAM EFFICIENCY

WRITING EFFICIENT PROGRAMS



- So far, we have emphasized correctness. It is the first thing to worry about!
- But sometimes that is not enough
- Problems can be very complex (as we shall see when we get to optimization in 6.0002)
- But data sets can be very large: in 2014 Google served 30,000,000,000,000 pages covering 100,000,000 GB of data



EFFICIENCY IS IMPORTANT

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - You can solve a problem using only a handful of different **algorithms**
- Want to separate choice of implementation from choice of more abstract algorithm

EVALUATING PROGRAMS

- Measure with a **timer**
- **Count** the operations
- Abstract notion of **order of growth**

Aside: MODULES

- A module is a set of python definitions and statements stored in a file
- You first need to “import” the module
- call functions inside the module using the module’s name and dot notation
- module.function()

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 convert_to_km(m):
    return m * 1.609

t0 = time.perf_counter()
c_to_f(100000)
t1 = time.perf_counter() - t0
print("t =", t1, "s,")
```

Example: Convert, compound

```
def convert_to_km(m):
    return m * 1.609

def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
       total = total * interest + invest
    return total
```

Measure time: convert

```
L_N = [1]
for i in range(7):
    L_N.append(L_N[-1]*10)


for N in L_N:
    t = time.perf_counter()
    km = convert_to_km(N)
    dt = time.perf_counter()-t
    print ("convert(", N, ") took ", dt, "seconds")
```

Measure time: convert multiple samples

```
### run the computation n_samples times for better measurement

n_samples = 50

for N in L_N:
    t = time.perf_counter()
    for i in range(n_samples):
        km = convert_to_km(N)
    dt = (time.perf_counter()-t)/n_samples
    print ("convert(", N, ") took ", dt, "seconds")
```

Measure time: compound

```
def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
       total = total * interest + invest
    return total
```

Measure time: sum

```
def sum_of(L):
    total = 0.0
    for elt in L:
        total = total + elt
    return total


L_N = [1]
for i in range(8):
    L_N.append(L_N[-1]*10)

for N in L_N:
    L = range(N)
    t = time.perf_counter()
    s = sum_of(L)
    print ("sum of ", N, "elements took ", time.perf_counter()-t, "seconds")
```

Measure time: is_in

```
def is_in(L, x):
    for elt in L:
        if elt==x: return True
    return False

def binary_search(L, x):
    """ returns True if x is in L
        L must be sorted """
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        mid = (hi+lo) // 2
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
    return L[lo] == x
```

Measure time: diameter

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                              + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

Measure time: binary numbers

```
def all_binary_numbers(N):
    def helper (prefix, N):
        if N==0:
            return [prefix]
        return helper(prefix+'0', N-1) + helper(prefix+'1', N-1)
    return helper('', N)
```

Two different machines

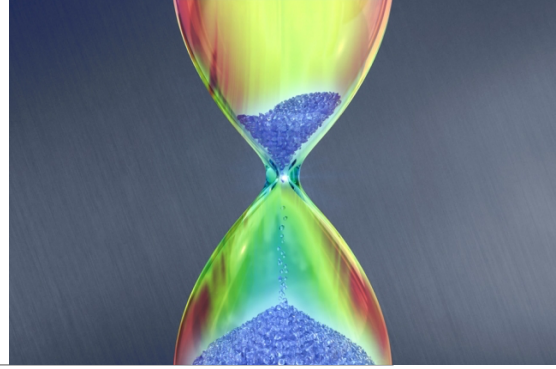
Fredo's laptop

```
convert( 1 ) took  0.0919969081879 seconds
convert( 10 ) took  0.0812351703644 seconds
convert( 100 ) took  0.0810060501099 seconds
convert( 1000 ) took  0.0786969661713 seconds
convert( 10000 ) took  0.0776309967041 seconds
convert( 100000 ) took  0.0800149440765 seconds
convert( 1000000 ) took  0.0772659778595 seconds
convert( 10000000 ) took  0.0839469432831 seconds
convert( 100000000 ) took  0.0802690982819 seconds
convert( 1000000000 ) took  0.0796220302582 seconds
compound( 1 ) took  0.0781879425049 seconds
compound( 10 ) took  0.0791871547699 seconds
compound( 100 ) took  0.0802779197693 seconds
compound( 1000 ) took  0.0811159610748 seconds
compound( 10000 ) took  0.079794883728 seconds
compound( 100000 ) took  0.0803499221802 seconds
compound( 1000000 ) took  0.180749893188 seconds
compound( 10000000 ) took  0.713826179504 seconds
compound( 100000000 ) took  6.48052787781 seconds
compound( 1000000000 ) took  63.5682651997 seconds
```

Fredo's (old) desktop

```
convert( 1 ) took  0.0651700496674 seconds
convert( 10 ) took  0.0838208198547 seconds
convert( 100 ) took  0.0830719470978 seconds
convert( 1000 ) took  0.0816540718079 seconds
convert( 10000 ) took  0.0824558734894 seconds
convert( 100000 ) took  0.0837979316711 seconds
convert( 1000000 ) took  0.0837349891663 seconds
convert( 10000000 ) took  0.0843281745911 seconds
convert( 100000000 ) took  0.0838270187378 seconds
convert( 1000000000 ) took  0.0844709873199 seconds
compound( 1 ) took  0.083487033844 seconds
compound( 10 ) took  0.0834701061249 seconds
compound( 100 ) took  0.083163022995 seconds
compound( 1000 ) took  0.0843181610107 seconds
compound( 10000 ) took  0.0845410823822 seconds
compound( 100000 ) took  0.099858045578 seconds
compound( 1000000 ) took  0.183917045593 seconds
compound( 10000000 ) took  1.38667988777 seconds
compound( 100000000 ) took  12.7653880119 seconds
compound( 1000000000 ) took  126.978576899 seconds
```

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** for small inputs ✗

- Time varies for different inputs but cannot really express a relationship between inputs and time ✗
- Can only be measured a-posteriori

Don’t get me wrong

- Timing is a critical tool to assess the performance of programs
  - At the end of the day, it is is unreplaceable for real-world assessment

- But we will learn a complementary tool (asymptotic complexity) that has other advantages
  - A priori evaluation (before writing or running code)
  - Assesses algorithm independently of machine and implementation
  - Provides direct insight to the **design** of efficient algorithms

COUNTING OPERATIONS

- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of operations executed as function of size of input

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

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

3 ops

1 op

loop x times

1 op

2 ops

mysum → 1+3(x+1) ops

Count oeprations

```
def is_in_counter(L, x):
    global count
    for elt in L:
        count += 1 #if == test
        if elt==x: return True
    return False
```

Count operations, binary search

```
def binary_search_counter(L, x):
    """ returns True if x is in L
        L must be sorted """
    global count
    lo = 0
    hi = len(L)
    while hi-lo > 1:
        count +=1 #while test
        mid = (hi+lo) // 2
        count += 2 #mid + and /
        if L[mid] <= x:
            lo = mid
        else:
            hi = mid
        count += 1 #if test
    count += 1 #return == test
    return L[lo] == x
```

COUNTING OPERATIONS IS BETTER, BUT …

- 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 and counting **evaluate machines**

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

A BETTER WAY

- Focus on idea of counting operations in an algorithm, but **not worry about small variations in implementation**
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- Want to **relate time** needed to complete a computation, measured this way, **against the size of the input** to the problem
- Need to decide what to measure, given that actual number of steps may depend on specifics of trial

HOW TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- Want to express **efficiency in terms of input**, so need to decide what is your input
- 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

- Consider that 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
  - Focus on **worst case** in this class

ORDERS OF GROWTH

- Want to evaluate programs when **input is very big**
- Want to express the **growth of program's run time**
- 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?)

MEASURING ORDER OF GROWTH: BIG O() 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 tends to occur often and is the bottleneck when a program runs
  - Express rate of growth of program relative to the input
  - Evaluate algorithm not machine or implementation
- A technicality
  - When we say that the complexity of f is O(n), we mean that its asymptotic growth is not worse than linear in n.
  - It is an **upper bound**, not necessarily a **tight bound**
  - In practice, we are usually looking for something close to a tight bound

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 + 7n + 1
- Worst case asymptotic complexity: O(n)
  - Ignore additive constants
  - Ignore multiplicative constants

WHAT DOES O(N) MEASURE?

- Interested in describing how amount of time needed grows as size of (input to) problem grows
- Given an expression for the number of operations needed to compute an algorithm, want to know **asymptotic behavior as size of problem gets large**
- Will focus on term that grows most rapidly
- Ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input

SIMPLIFICATION EXAMPLES

- Drop constants and multiplicative factors
- Focus on **dominant term**

: $n^2 + 2n + 2$

: $n^2 + 100000n + 3^{1000}$

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

SIMPLIFICATION EXAMPLES

- Drop constants and multiplicative factors
- Focus on **dominant term**

$O(n^2)$: $n^2 + 2n + 2$

$O(n^2)$: $n^2 + 100000n + 3^{1000}$

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

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):          O(n)
    print('a')
for j in range(n*n):
    print('b')              O(n^2)
```

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,

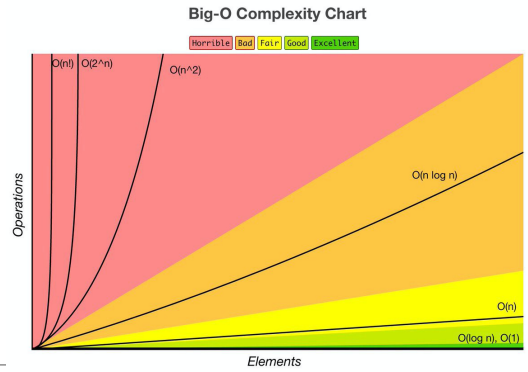
```
for i in range(n):
    for j in range(n):
        print 'a'
```

O(n)

O(n) for each outer loop iteration

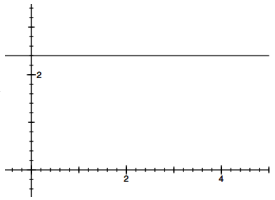
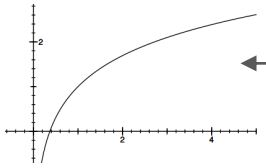
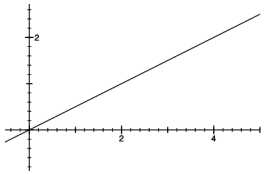
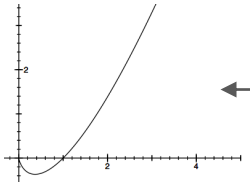
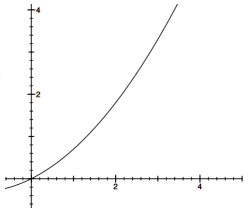
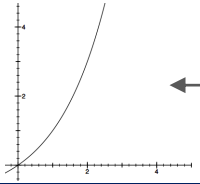
is O(n)*O(n) = O(n*n) = O(n^2) because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

COMPLEXITY CLASSES

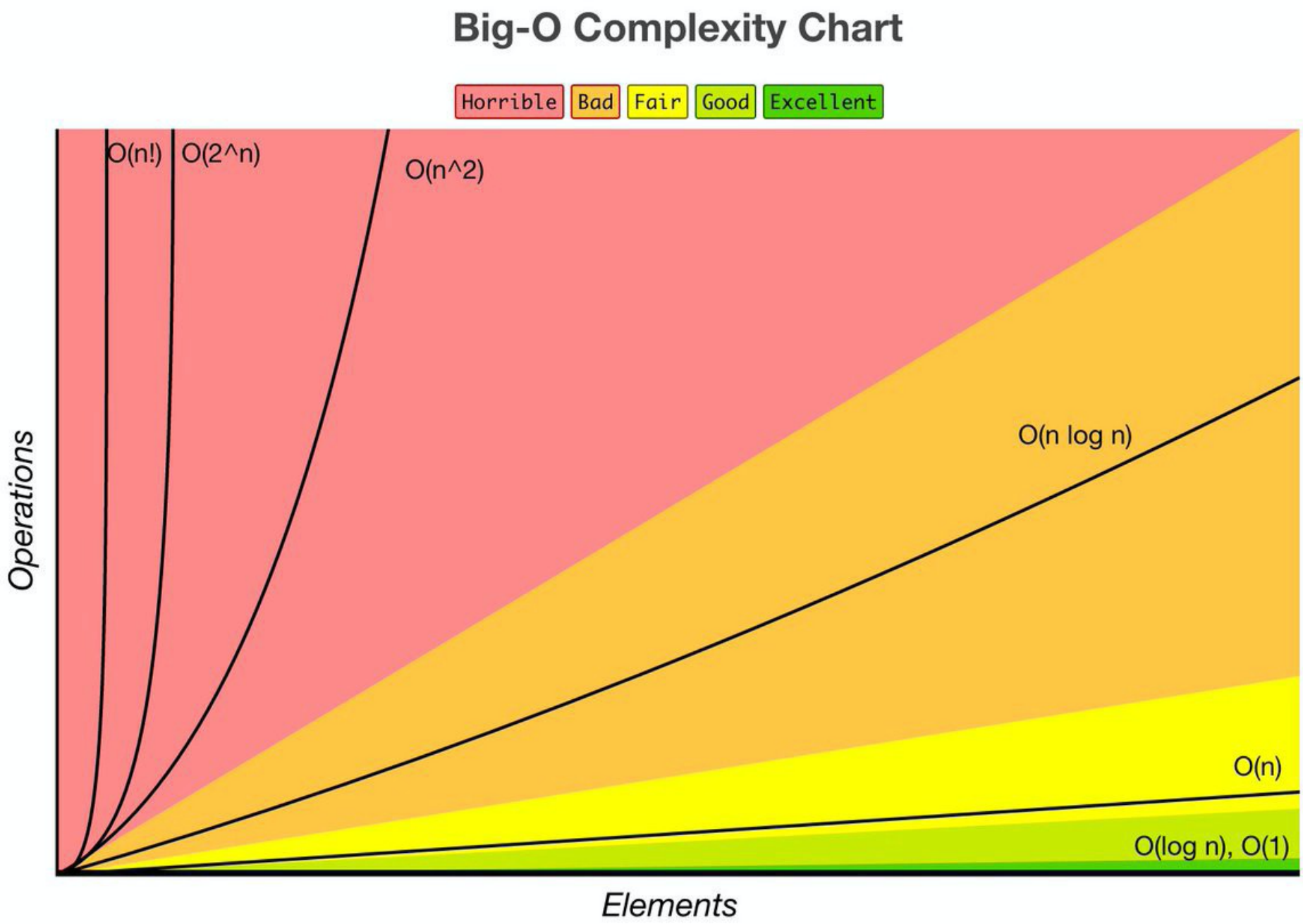


- *O(1)* denotes **constant** running time
- *O(log n)* denotes **logarithmic** running time
- *O(n)* denotes **linear** running time
- *O(n log n)* denotes **log-linear** running time
- $O(n^c)$ denotes **polynomial** running time (c is a constant)
- $O(c^n)$ denotes **exponential** running time (c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

COMPLEXITY CLASSES ORDERED LOW TO HIGH

`O(1)`	:	constant →	
`O(log n)`	:	 ← logarithmic	
`O(n)`	:	linear →	
`O(n log n)`	:	 ← log linear	
`O(n^c)`	:	polynomial →	
`O(c^n)`	:	 ← exponential	

c is a constant


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

COMPLEXITY GROWTH

CLASS	N = 10	N = 100	N = 1000	N = 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
$O(n^2)$	100	10000	1000000	1000000000000
$O(2^n)$	1024	1267650600228229401496703205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937676	Good Luck!!

CONSTANT COMPLEXITY

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

CONSTANT COMPLEXITY: EXAMPLE 1

```
def convert_to_km(m):
    return m * 1.609
```

CONSTANT COMPLEXITY: EXAMPLE 2 (with a twist)

- Multiply x by y

```
def mul(x, y):
    tot = 0
    for i in range(y):
        tot += x
    return tot
```

- complexity in terms of x: **O(1)**
- complexity in terms of y: **O(y)**

LINEAR COMPLEXITY

LINEAR COMPLEXITY

```
def compound(invest, interest, n_months):
    total=0
    for i in range(n_months):
       total = total * interest + invest
    return total
```

```
compound( 1 ) took  2.1050000214017928e-06 seconds
compound( 10 ) took  2.3679986043134704e-06 seconds
compound( 100 ) took  5.580001015914604e-06 seconds
compound( 1000 ) took  5.0598000598256476e-05 seconds
compound( 10000 ) took  0.0005178840001462959 seconds
compound( 100000 ) took  0.005193626000618678 5 seconds
compound( 1000000 ) took  0.05187488499905157 4 seconds
compound( 10000000 ) took  0.4647158330008096 5 seconds
```

LINEAR COMPLEXITY

- Simple **iterative loop** algorithms
- Loops must be a **function of input**
- Linear search a list to see if an element is present
- Recursive functions with one recursive call and constant overhead for call

LINEAR COMPLEXITY: EXAMPLE 1

- Add characters of a string, assumed to be composed of decimal digits

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

- **O(len(s))**
- **O(n) where n is len(s)**

LINEAR COMPLEXITY: EXAMPLE 2

- Loop to find the factorial of a number

```
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)**

LINEAR COMPLEXITY: EXAMPLE 3

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

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

Funny thing about factorial and python

```
iterative fact( 1 ) took  1.8179998733103275e-06 seconds
iterative fact( 10 ) took  1.899001290439628e-06 seconds
iterative fact( 100 ) took  8.716999218449928e-06 seconds
iterative fact( 1000 ) took  0.00029863599957025144 seconds
iterative fact( 10000 ) took  0.02207461900704843 seconds
iterative fact( 100000 ) took  2.204656157000497 seconds
```

- Eventually grows faster than linear
- Because Python increases the size of integers, which yields more costly operations
- For this class: ignore such effects

LINEAR SEARCH ON UNSORTED LIST

```
def is_in(L, x):
    for elt in L:
        if elt==x: return True
    return False
```

Must look through all elements to decide it's not there

- O(len(L)) for the loop * O(1) to test if e == L[i]
- Overall complexity is **O(n) – where n is len(L)**
- **O(len(L))**

LINEAR SEARCH ON UNSORTED LIST

```
def is_in(L, x):
    for elt in L:
        if elt==x: return True
    return False
```

speed up a little by returning True here, but speed up doesn't impact worst case

Must look through all elements to decide it's not there

- O(len(L)) for the loop * O(1) to test if e == L[i]
- Overall complexity is **O(n) – where n is len(L)**
- **O(len(L))**

POLYNOMIAL COMPLEXITY

QUADRATIC COMPLEXITY

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                              + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

```
diameter of 10 points took 3.1518999094259925e-05 seconds
diameter of 100 points took 0.002390421001109644 seconds
diameter of 1000 points took 0.2340913150001142 6 seconds
diameter of 10000 points took 16.50804504900043 seconds
```

Predictable without running

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                              + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

POLYNOMIAL COMPLEXITY (OFTEN QUADRATIC)

- Most **common polynomial algorithms are quadratic**, i.e., complexity grows with square of size of input
- Commonly occurs when we have **nested loops** or some recursive function calls

QUADRATIC COMPLEXITY: EXAMPLE 1

```
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**
- **$O(n^2)$**

QUADRATIC COMPLEXITY: EXAMPLE 2

- Find if L1 is a subset of L2, if all elements in L1 are in L2

```
def is_subset(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: EXAMPLE 2

```
def is_subset(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: EXAMPLE 3

```
def diameter(L):
    """ assumes that L is a list of pairs of Cartesian coordinates
    compares all pairs of points and returns the largest distance """
    farthest_dist = 0
    for i in range(len(L)):
        # next loop only goes from i+1 so that pairs are not compared twice
        for j in range(i+1, len(L)):
            p1 = L[i]
            p2 = L[j]
            dist = math.sqrt( (p1[0]-p2[0])**2
                              + (p1[1]-p2[1])**2 )
            if dist > farthest_dist:
                farthest_dist = dist
    return farthest_dist
```

```
            if not(e in unique):
                unique.append(e)
        return unique
```

QUADRATIC COMPLEXITY: EXAMPLE 3

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

First nested loop takes **O(len(L1)*len(L2))** steps.

Second loop takes at most **O(len(L1)*len(L2))** steps. Typically not this bad.

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

Cubic complexity

- Matrix-Matrix multiply
- Cubic in matrix length N
  - N*N output coefficients,
    each coefficient is a sum over N values

EXPONENTIAL COMPLEXITY

EXPONENTIAL COMPLEXITY

```
def all_binary_numbers(N):
    def helper (prefix, N):
        if N==0:
            return [prefix]
        return helper(prefix+'0', N-1) + helper(prefix+'1', N-1)
    return helper('', N)
```

```
binary numbers of  5 digits took  3.156400089210365e-05 s
binary numbers of  10 digits took  0.0007744319991616067 s
binary numbers of  15 digits took  0.016560387999561 5 s
binary numbers of  20 digits took  0.4674571899995499 s
binary numbers of  25 digits took  18.066407528000127 s
```

EXPONENTIAL COMPLEXITY

- Recursive functions where have more than one recursive call for each size of problem
  - (bad algorithm for) Fibonacci, power set
- Many important problems are inherently exponential
  - Unfortunate, as cost can be high
  - Will lead us to consider approximate solutions more quickly

EXPONENTIAL COMPLEXITY GENERATE SUBSETS

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

Go until reach list of empty list

all subsets without last element

create a list of just last element

for all smaller solutions, add one with last element

combine those with last element and those without

EXPONENTIAL COMPLEXITY GENERATE SUBSETS

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    smaller = gen_subsets(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 GENERATE SUBSETS

```
def gen_subsets(L):
    if len(L) == 0:
        return [[]]
    smaller = gen_subsets(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 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:

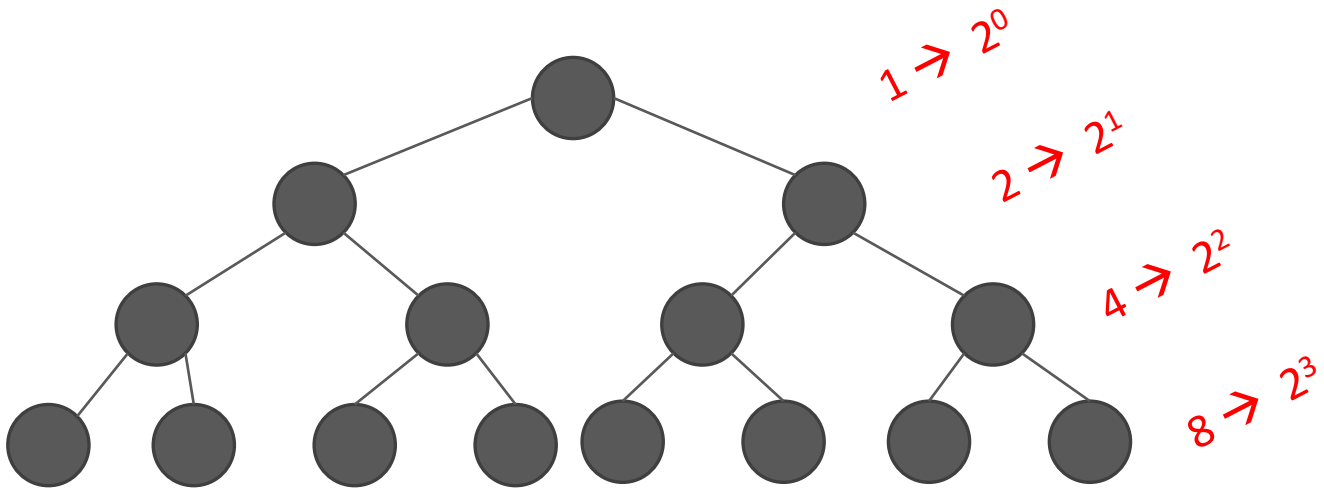
  O(1) + O(n) + O(1) ➔ **O(n)**

COMPLEXITY OF RECURSIVE FIBONACCI

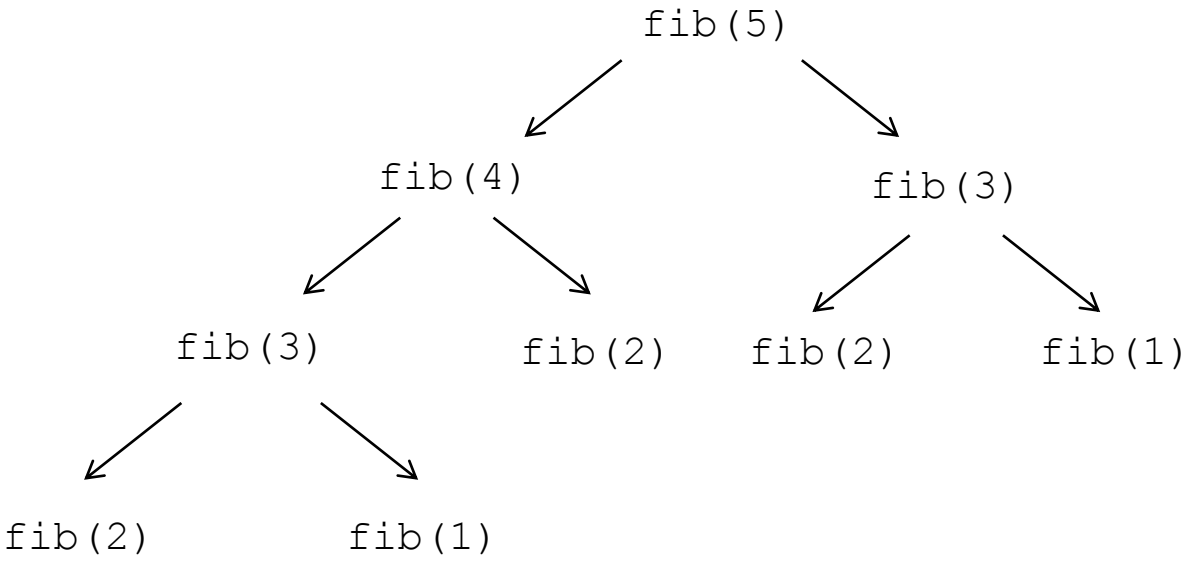
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

  $O(2^n)$



COMPLEXITY OF RECURSIVE FIBONACCI



- Can do a bit better than 2^n since tree thins out to the right
- But complexity is still order exponential

BIG OH SUMMARY

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

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

IN SUMMARY

- Empirical Timing is a critical tool to assess the performance of programs
  - Only reliable way to take into account complexities of machine, implementation, and inputs

- But asymptotic complexity has other advantages
  - A priori evaluation (before writing or running code)
  - Assesses algorithm independently of machine and implementation
  - Provides direct insight to the **design** of efficient algorithms