Algorithmic Complexity

Lesson 2

Today

- Measuring orders of growth of algorithms
- Big “Oh” notation
- Complexity classes

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 (e.g., in 2014, Google served 30,000,000,000,000 pages, covering 100,000,000 GB – how long to search brute force?)
  - thus, simple solutions may simply not scale with size in acceptable manner
- how can we decide which option for program is most efficient?

- separate **time and space efficiency** of a program
- tradeoff between them:
  - can sometimes pre-compute results are stored; then use "lookup" to retrieve (e.g., memoization for Fibonacci)
  - 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 clear 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**

STILL NEED A BETTER WAY

- Going to focus on idea of counting operations in an algorithm, but not worry about small variations in implementation (e.g., whether we take 3 or 4 primitive operations to execute the steps of a loop)
- Going to 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

NEED TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- want to express **efficiency in terms of size 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 – as tight as possible
- 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?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

stretch . . .

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
```

answer = answer * n

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

WHAT DOES *O(N)* MEASURE?

- Interested in describing how amount of time needed grows as size of (input to) problem grows
- Thus, given an expression for the number of operations needed to compute an algorithm, want to know asymptotic behavior as size of problem gets large
- Hence, will focus on term that grows most rapidly in a sum of terms
- And will 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 terms**

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

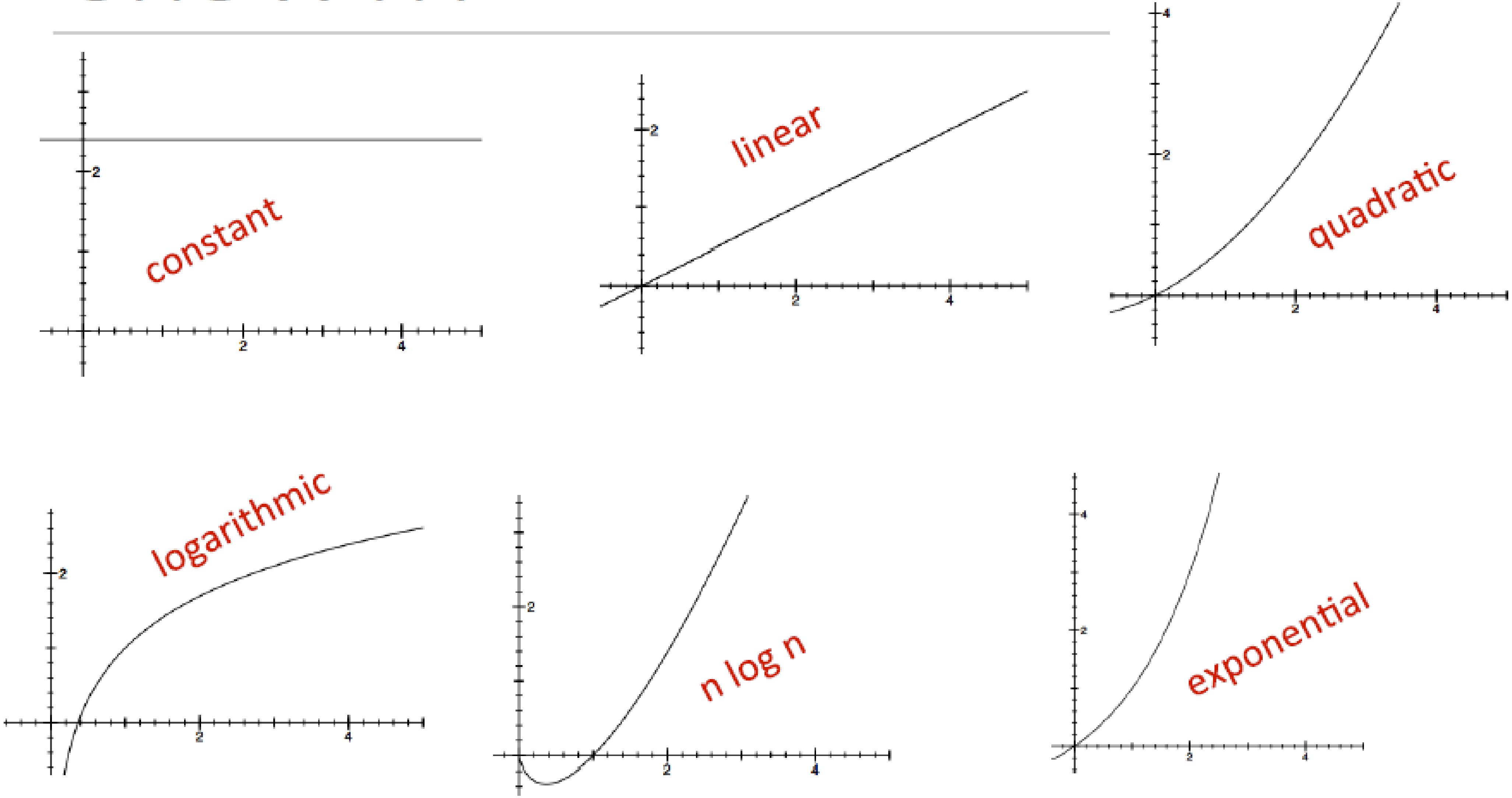
$O(n^2)$: $n^2 + 100000n + 3^{1000}$

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

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

$O(3^n)$: $2n^{30} + 3^n$

TYPES OF ORDERS OF GROWTH



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')
```

O(n)

O(n*n)

O(n) + O(n*n)

is O(n) + O(n*n) = O(n+n²) = O(n²) 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)

n loops, each O(n) →
O(n)*O(n)

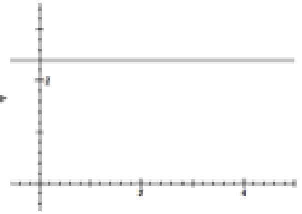
is $O(n)*O(n) = O(n*n) = O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

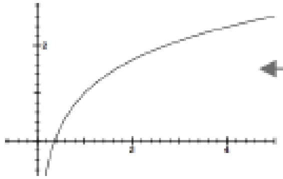
stretch . . .

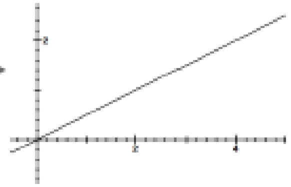
COMPLEXITY CLASSES

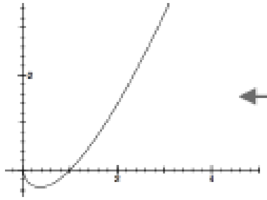
- *$O(1)$* denotes constant running time
- *$O(\log n)$* denotes logarithmic running time
- *$O(n)$* denotes linear running time
- *$O(n \log n)$* denotes log-linear running time
- *$O(n^c)$* denotes polynomial running time (c is a constant)
- *$O(c^n)$* denotes exponential running time (c is a constant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

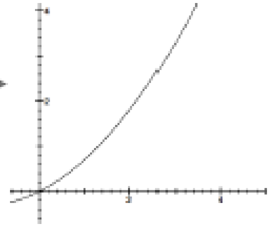
COMPLEXITY CLASSES ORDERED LOW TO HIGH

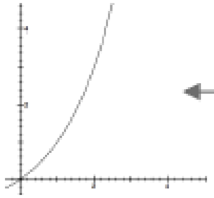
$O(1)$: constant → 

$O(\log n)$:  ← logarithmic

$O(n)$: linear → 

$O(n \log n)$:  ← loglinear

$O(n^c)$: polynomial → 

$O(c^n)$:  ← exponential

c is a constant

COMPLEXITY GROWTH

CLASS	n=10	= 100	= 1000	= 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	12676506 00228229 40149670 3205376	10715086071862673209484250490600 01810561404811705533607443750388 37035105112493612249319837881569 58581275946729175531468251871452 85692314043598457757469857480393 45677748242309854210746050623711 41877954182153046474983581941267 39876755916554394607706291457119 64776865421676604298316526243868 37205668069376	**Good luck!!**

LINEAR COMPLEXITY

- Simple iterative loop algorithms are typically linear in complexity

LINEAR SEARCH ON UNSORTED LIST

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

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]
    - O(1 + 4n + 1) = O(4n + 2) = O(n)
- overall complexity is **O(n) – where n is len(L)**

Assumes we can retrieve element of list in constant time

LINEAR SEARCH ON SORTED LIST

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

- must only look until reach a number greater than e
- $\boxed{\text{O(len(L))}}$ for the loop * O(1) to test if e == L[i]
- overall complexity is **O(n) – where n is len(L)**
- **NOTE:** order of growth is same, though run time may differ for two search methods

worst case will need to look at whole list

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
```

- *O(len(s))*

LINEAR COMPLEXITY

- complexity often depends on number of iterations

```
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 (in this case, 3 – set i, multiply, set prod)
    - O(1 + 3n + 1) = O(3n + 2) = O(n)
- overall just *O(n)*

NESTED LOOPS

- simple loops are linear in complexity
- what about loops that have loops within them?

QUADRATIC COMPLEXITY

determine if one list is subset of second, i.e., every element of first, appears in second (assume no duplicates)

```
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, with constant number of operations

*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
- determining if element in list might take *len(L1)* steps
- if we assume lists are of roughly same length, then
- *O(len(L1)^2)*

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

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 ONLY IF 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

BISECTION SEARCH

- suppose we want to know if a particular element is present in a list
- saw last time that we could just "walk down" the list, checking each element
- complexity was linear in length of the list
- suppose we know that the list is ordered from smallest to largest
    - saw that sequential search was still linear in complexity
    - can we do better?

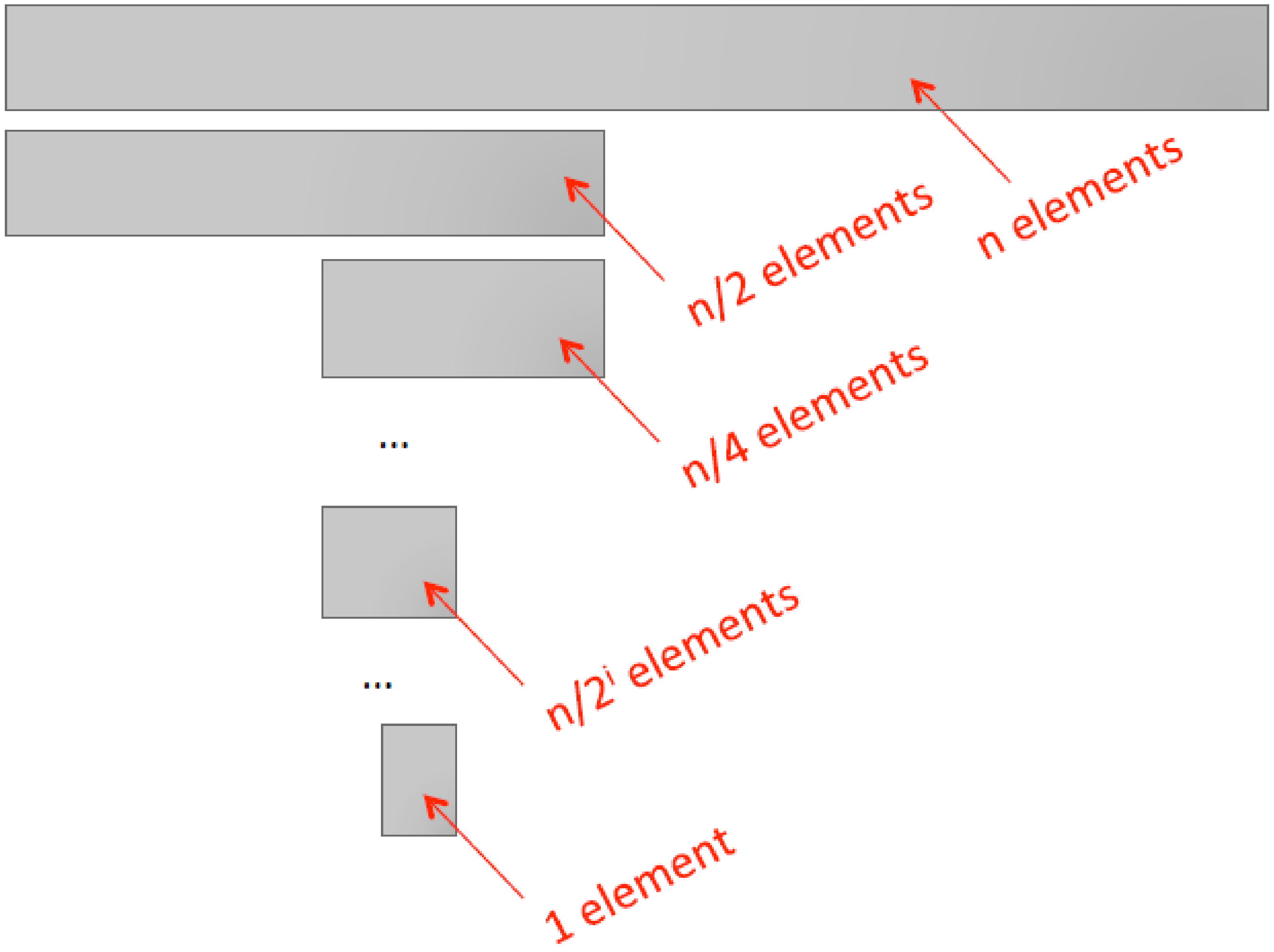
BISECTION SEARCH

1. pick an index, `i`, that divides list in half
2. ask if `L[i] == e`
3. if not, ask if `L[i]` is larger or smaller than `e`
4. depending on answer, search left or right half of `L` for `e`

A new version of a divide-and-conquer algorithm

- break into smaller version of problem (smaller list), plus some simple operations
- answer to smaller version is answer to original problem

BISECTION SEARCH COMPLEXITY ANALYSIS



- finish looking through list when

  $1 = n/2^i$

  so $i = \log n$

- complexity of recursion is **O(log n) – where n is len(L)**

BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1( L[:half], e)
        else:
            return bisect_search1( L[half:], e)
```

constant O(1)

constant O(1)

constant O(1)

NOT constant, copies list

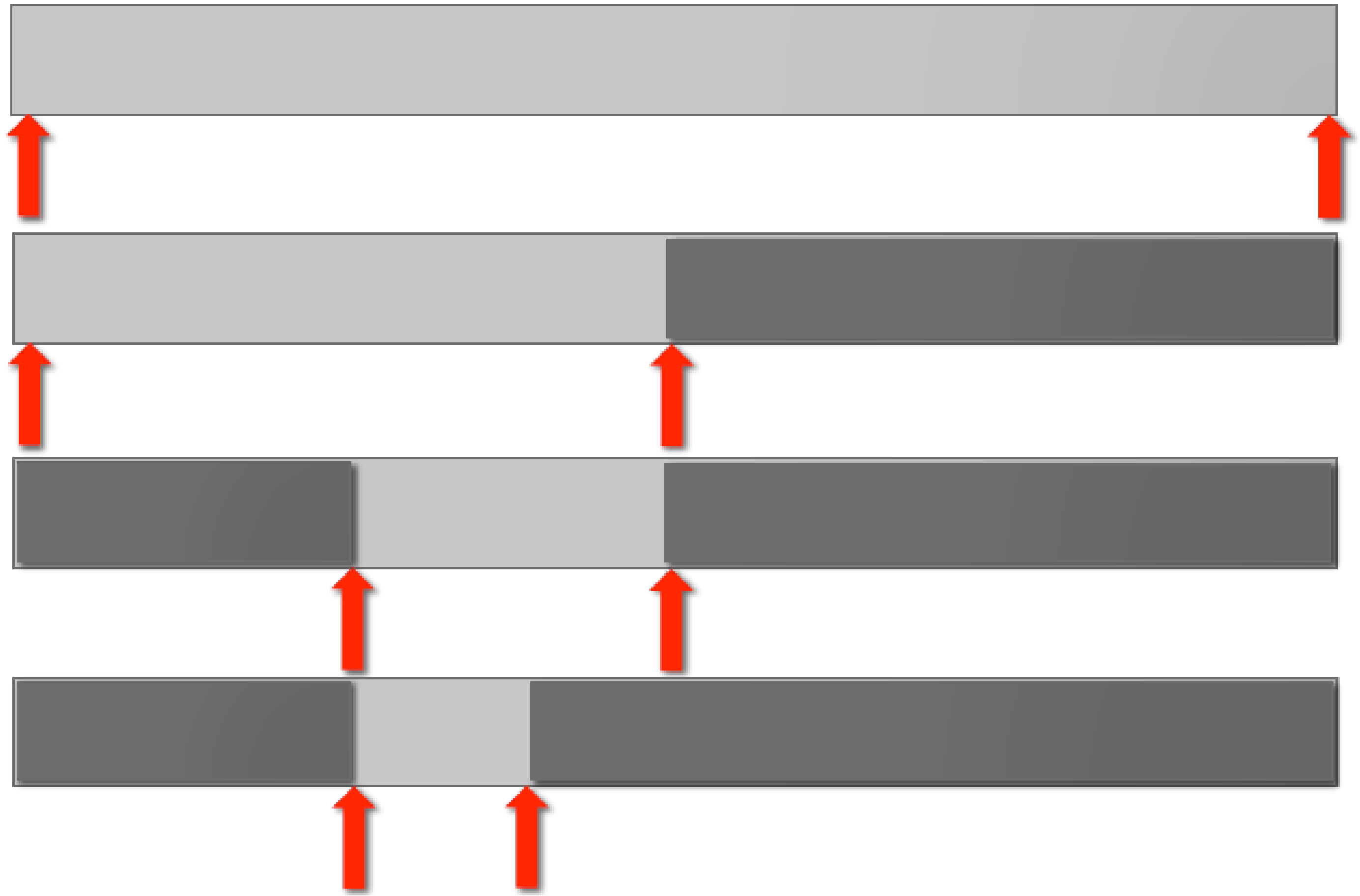
NOT constant

NOT constant

COMPLEXITY OF FIRST BISECTION SEARCH METHOD

- **implementation 1 – bisect_search1**
  - O(log n) bisection search calls
    - On each recursive call, size of range to be searched is cut in half
    - If original range is of size n, in worst case down to range of size 1 when n/(2^k) = 1; or when k = log n
  - O(n) for each bisection search call to copy list
    - This is the cost to set up each call, so do this for each level of recursion
  - O(log n) * O(n) → **O(n log n)**
  - if we are really careful, note that length of list to be copied is also halved on each recursive call
    - turns out that total cost to copy is **O(n)** and this dominates the log n cost due to the recursive calls

BISECTION SEARCH ALTERNATIVE



- still reduce size of problem by factor of two on each step
- but just keep track of low and high portion of list to be searched
- avoid copying the list
- complexity of recursion is again **O(log n) – where n is len(L)**

BISECTION SEARCH IMPLEMENTATION 2

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        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 bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

constant other than recursive call

constant other than recursive call

COMPLEXITY OF SECOND BISECTION SEARCH METHOD

- **implementation 2 – bisect_search2** and its helper
  - O(log n) bisection search calls
    - On each recursive call, size of range to be searched is cut in half
    - If original range is of size n, in worst case down to range of size 1 when n/(2^k) = 1; or when k = log n
  - pass list and indices as parameters
  - list never copied, just re-passed as a pointer
  - thus O(1) work on each recursive call
  - O(log n) * O(1) → **O(log n)**

LOGARITHMIC COMPLEXITY

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

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))*

O() FOR ITERATIVE FACTORIAL

- complexity can depend on number of iterative calls

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

- overall ***O(n)*** – n times round loop, constant cost each time

O() FOR RECURSIVE FACTORIAL

```
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, 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, and constant effort to set up call
- **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

POLYNOMIAL COMPLEXITY

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

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 as may provide reasonable answer more quickly

COMPLEXITY OF TOWERS OF HANOI

- Let t_n denote time to solve tower of size n
- $t_n = 2t_{n-1} + 1$
- $= 2(2t_{n-2} + 1) + 1$
- $= 4t_{n-2} + 2 + 1$
- $= 4(2t_{n-3} + 1) + 2 + 1$
- $= 8t_{n-3} + 4 + 2 + 1$
- $= 2^k t_{n-k} + 2^{k-1} + \ldots + 4 + 2 + 1$
- $= 2^{n-1} + 2^{n-2} + \ldots + 4 + 2 + 1$
- $= 2^n - 1$
- so order of growth is *$O(2^n)$*

Geometric growth

$$\begin{aligned} a &= \phantom{2^n + {}} 2^{n-1} + \ldots + 2 + 1 \\ 2a &= 2^n + 2^{n-1} + \ldots + 2 \\ a &= 2^n \phantom{{}+ 2^{n-1} + \ldots + 2} - 1 \end{aligned}$$

stretch . . .

EXPONENTIAL COMPLEXITY

- given a set of integers (with no repeats), want to generate the collection of all possible subsets – called the power set
- {1, 2, 3, 4} would generate
  - {}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}
- order doesn't matter
  - {}, {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}

POWER SET – CONCEPT

- we want to generate the power set of integers from 1 to n
- assume we can generate power set of integers from 1 to n-1
- then all of those subsets belong to bigger power set (choosing not include n); and all of those subsets with n added to each of them also belong to the bigger power set (choosing to include n)
- {}, {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}
- nice recursive description!

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
```

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

how can we deduce overall complexity?

EXPONENTIAL COMPLEXITY

- let t_n denote time to solve problem of size n
- let s_n denote size of solution for problem of size n
- $t_n = t_{n-1} + s_{n-1} + c$ (where c is some constant number of operations)
- $t_n = t_{n-1} + 2^{n-1} + c$
- $= t_{n-2} + 2^{n-2} + c + 2^{n-1} + c$
- $= t_{n-k} + 2^{n-k} + \ldots + 2^{n-1} + kc$
- $= t_0 + 2^0 + \ldots + 2^{n-1} + nc$
- $= 1 + 2^n + nc$

Thus computing power set is *$O(2^n)$*

COMPLEXITY CLASSES

- *$O(1)$* – code does not depend on size of problem
- *$O(\log n)$* – reduce problem in half each time through process
- *$O(n)$* – simple iterative or recursive programs
- *$O(n \log n)$* – will see next time
- *$O(n^c)$* – nested loops or recursive calls
- *$O(c^n)$* – multiple recursive calls at each level

SOME MORE EXAMPLES OF ANALYZING COMPLEXITY

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:

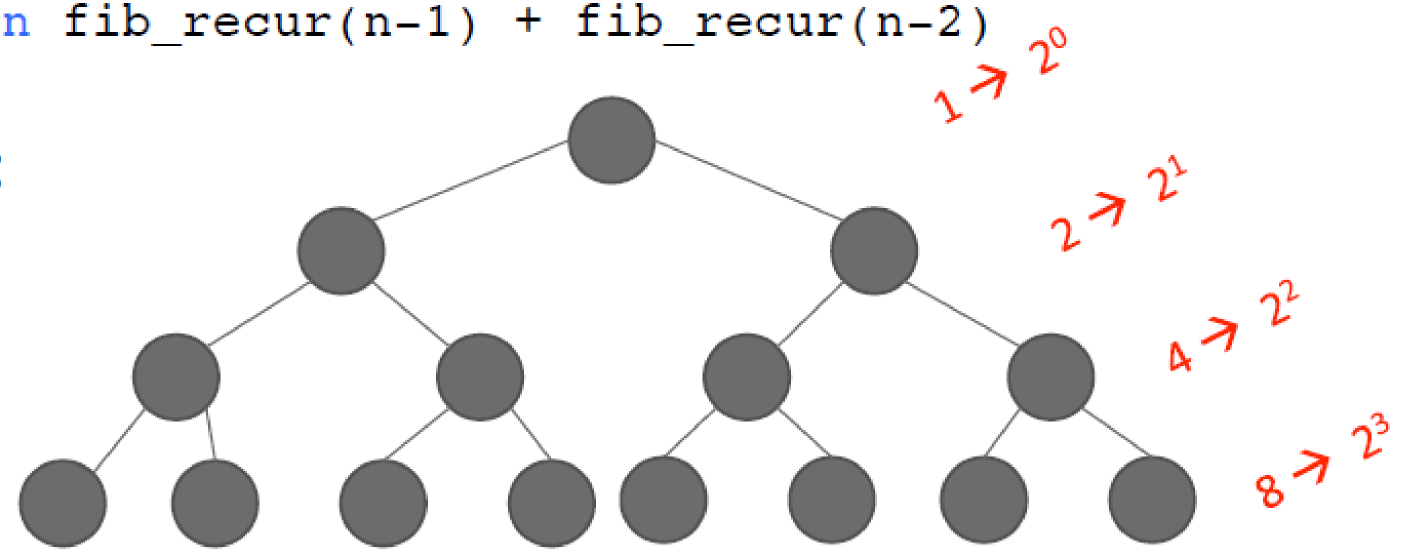
  O(1) + O(n) + O(1) → **O(n)**

COMPLEXITY OF RECURSIVE FIBONACCI

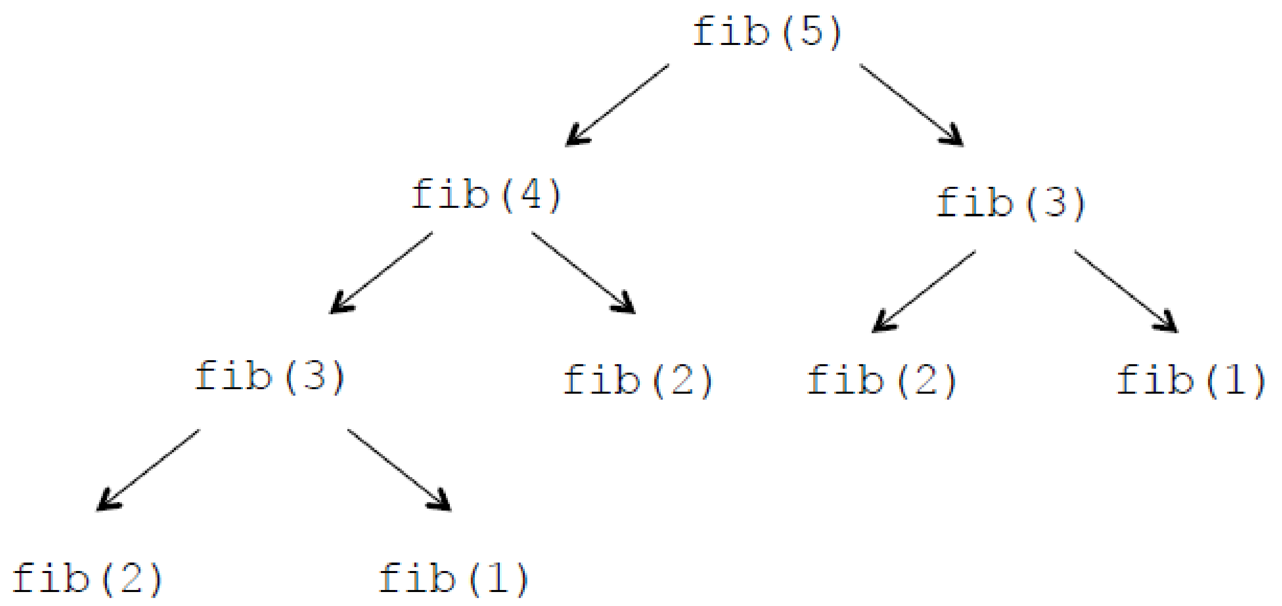
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

  $O(2^n)$



COMPLEXITY OF RECURSIVE FIBONACCI



- actually can do a bit better than 2^n since tree of cases thins out to right
- but complexity is still exponential

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)

WHY WE WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- how can we reason about an algorithm in order to predict the amount of time it will need to solve a problem of a particular size?
- how can we relate choices in algorithm design to the time efficiency of the resulting algorithm?
  - are there fundamental limits on the amount of time we will need to solve a particular problem?

ORDERS OF GROWTH: RECAP

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 – as tight as possible
- 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?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

Exercise

What is the asymptotic complexity of each of the following functions?

```
 def g(L, e):
    """L a list of ints, e is an int"""
    for i in range(100):
        for e1 in L:
            if e1 == e:
                return True
    return False
def h(L, e):
    """L a list of ints, e is an int"""
    for i in range(e):
        for e1 in L:
            if e1 == e:
                return True
    return False
```

Explain why?