

# UNDERSTANDING PROGRAM EFFICIENCY: 2

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

---

6.0001 LECTURE 11

# TODAY

---

- Classes of complexity
- Examples characteristic of each class

# 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**

# COMPLEXITY CLASSES: RECAP

---

- 
- $O(1)$  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

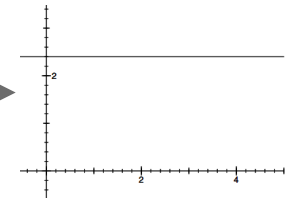


*c is a  
constant*

$O(1)$

:

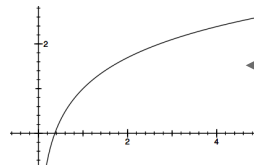
constant



$O(\log n)$

:

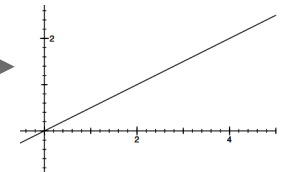
logarithmic



$O(n)$

:

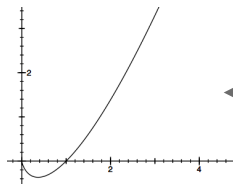
linear



$O(n \log n)$

:

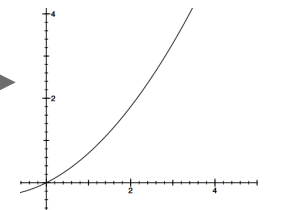
loglinear



$O(n^c)$

:

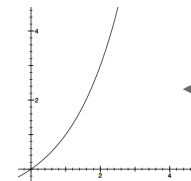
polynomial



$O(c^n)$

:

exponential



# 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<br>00228229<br>40149670<br>3205376 | 1071508607186267320948425049060<br>0018105614048117055336074437503<br>8837035105112493612249319837881<br>5695858127594672917553146825187<br>1452856923140435984577574698574<br>8039345677748242309854210746050<br>6237114187795418215304647498358<br>1941267398767559165543946077062<br>9145711964776865421676604298316<br>52624386837205668069376 | Good luck!!   |



# 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

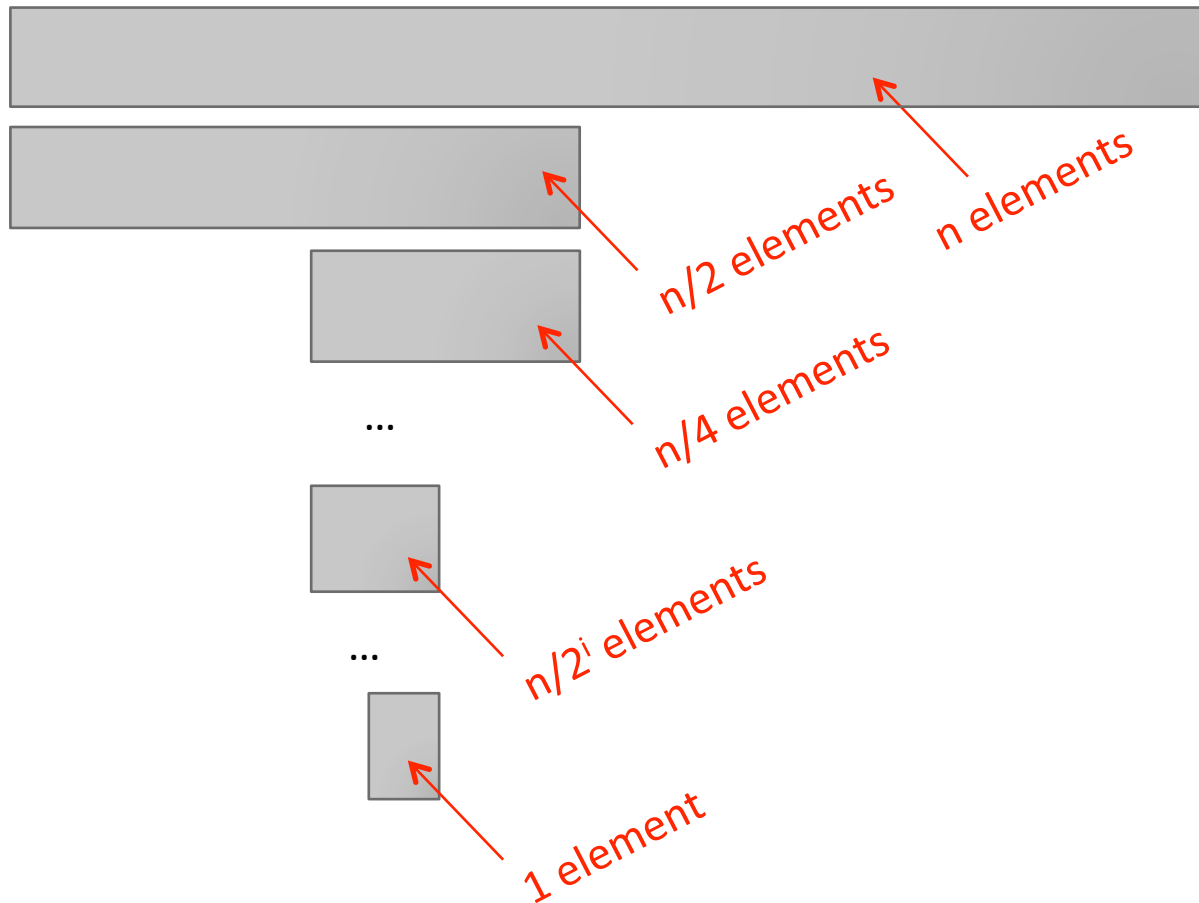
---

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

$$1 = n/2^i$$

$$\text{so } i = \log n$$

- complexity of recursion is  **$O(\log n)$**  – where  $n$  is  $\text{len}(L)$

# BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
```

```
    if L == []:
```

```
        return False
```

*constant  
 $O(1)$*

```
    elif len(L) == 1:
```

```
        return L[0] == e
```

*constant  
 $O(1)$*

```
    else:
```

```
        half = len(L)//2
```

*constant  
 $O(1)$*

```
        if L[half] > e:
```

```
            return bisect_search1(L[:half], e)
```

*NOT constant,  
copies list*

```
        else:
```

```
            return bisect_search1(L[half:], e)
```

*NOT constant  
NOT constant*

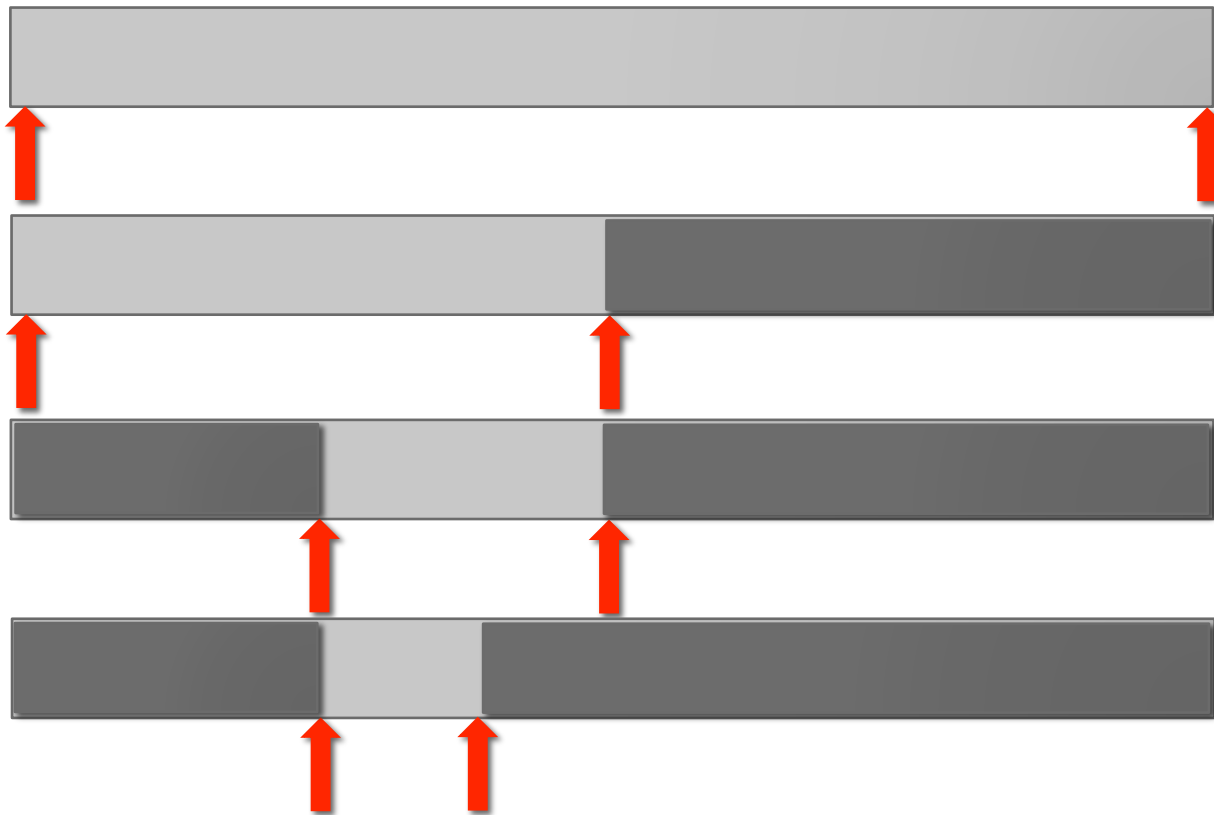
# COMPLEXITY OF FIRST BISECTION SEARCH METHOD

---

## ■ **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
- $O(\log n) * O(n) \rightarrow \mathbf{O(n \log n)}$
- 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  $\mathbf{O(n)}$  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  $\text{len}(L)$

# 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
  - $O(\log n) * O(1) \rightarrow \mathbf{O(\log n)}$

# 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?
- $O(\log(i))$

# LINEAR COMPLEXITY

---

- saw this last time
  - searching a list in sequence to see if an element is present
  - iterative loops

# 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
- will return to this next lecture

# 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
- saw this last time

# 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  $n$

- $t_n = 2t_{n-1} + 1$

- $= 2(2t_{n-2} + 1) + 1$

- $= 4t_{n-2} + 2 + 1$

- $= 4(2t_{n-3} + 1) + 2 + 1$

- $= 8t_{n-3} + 4 + 2 + 1$

- $= 2^k t_{n-k} + 2^{k-1} + \dots + 4 + 2 + 1$

- $= 2^{n-1} + 2^{n-2} + \dots + 4 + 2 + 1$

- $= 2^n - 1$

- so order of growth is  $O(2^n)$

Geometric growth

$$a = 2^{n-1} + \dots + 2 + 1$$

$$2a = 2^n + 2^{n-1} + \dots + 2$$

$$a = 2^n - 1$$

# 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)
- $t_n = t_{n-1} + 2^{n-1} + c$
- $= t_{n-2} + 2^{n-2} + c + 2^{n-1} + c$
- $= t_{n-k} + 2^{n-k} + \dots + 2^{n-1} + kc$
- $= t_0 + 2^0 + \dots + 2^{n-1} + nc$
- $= 1 + 2^n + nc$

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_i
```

```
        return fib_ii
```

constant  
 $O(1)$

constant  
 $O(1)$

linear  
 $O(n)$

constant  
 $O(1)$

- Best case:

$O(1)$

- Worst case:

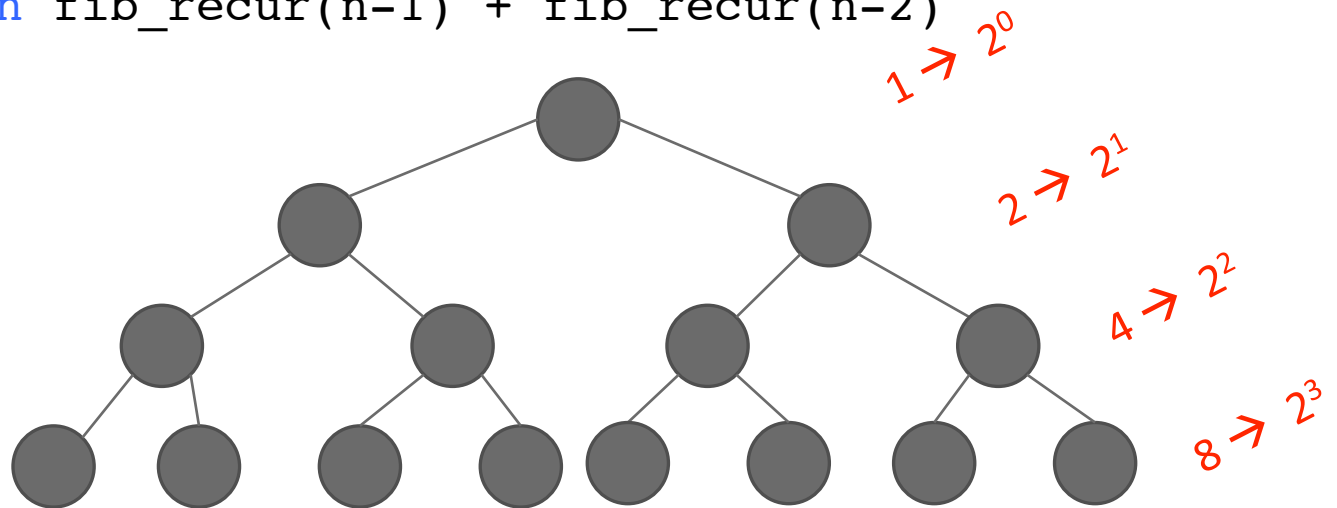
$O(1) + O(n) + O(1) \rightarrow \mathbf{O(n)}$

# 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)
```

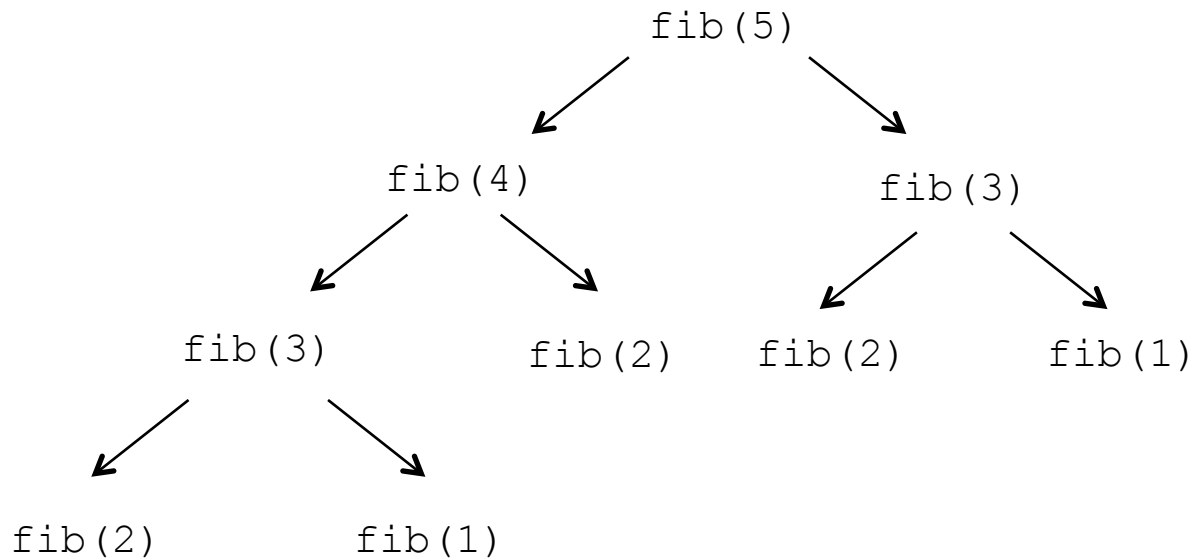
- Worst case:

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



# 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  $O(n)$

MIT OpenCourseWare  
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python  
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.