

PROGRAM EFFICIENCY

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

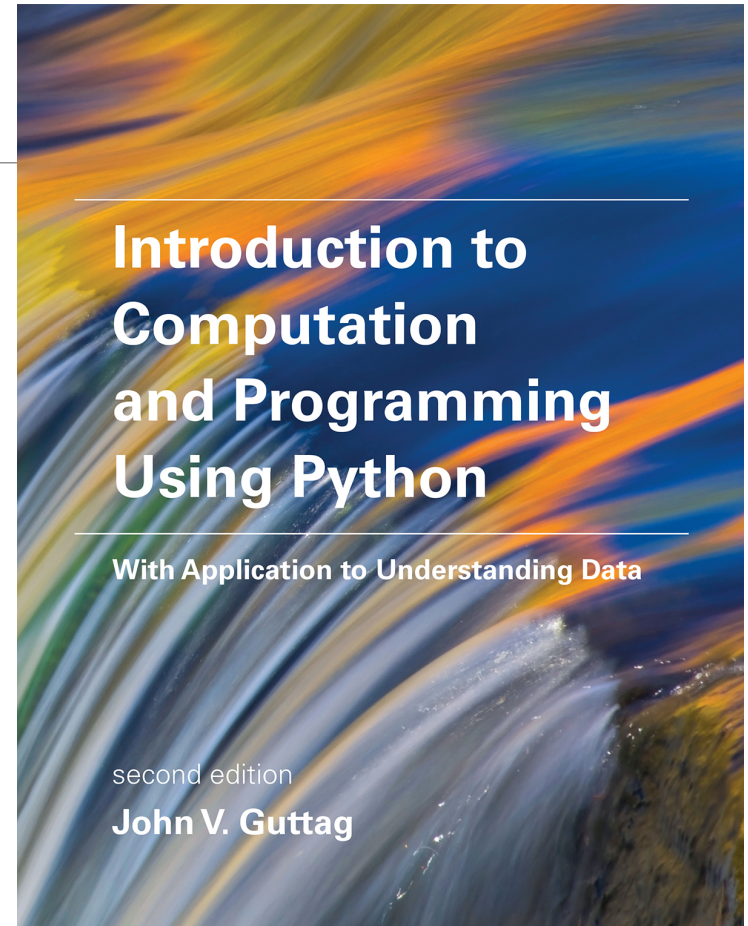
6.0001 LECTURE 9

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

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 ALGORITHMS

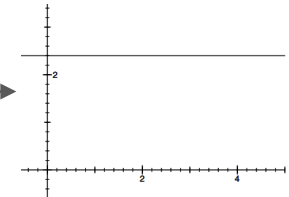
- Focus on idea of counting operations in an algorithm, but **not worry about small variations in implementation**
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- Look at the **worst case asymptotic run time** of a program, as the input grows to a large value

COMPLEXITY CLASSES ORDERED LOW TO HIGH

$O(1)$

:

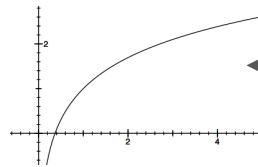
constant



$O(\log n)$

:

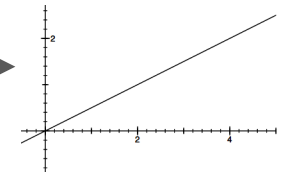
← logarithmic



$O(n)$

:

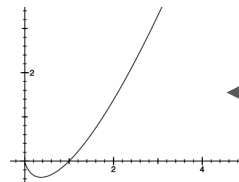
linear



$O(n \log n)$

:

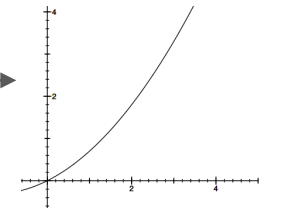
← loglinear



$O(n^c)$

:

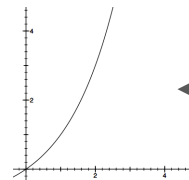
polynomial



$O(c^n)$

:

← exponential



*c is a
constant*

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

- Add x to y

```
def add(x, y):  
    return x+y
```

- **$O(1)$**

CONSTANT COMPLEXITY: EXAMPLE 2

- 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

- 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
```

- **$O(\text{len}(s))$**
- **$O(n)$ where n is $\text{len}(s)$**

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**

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

LINEAR SEARCH ON **SORTED** LIST

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

- Must only look until reach a number greater than e
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- Overall complexity is **$O(n)$ – where n is $\text{len}(L)$**
- **$O(\text{len}(L))$**

POLYNOMIAL COMPLEXITY

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 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
 $\text{len}(L1)$ times

Each iteration will execute
inner loop up to $\text{len}(L2)$
times

$O(\text{len}(L1) * \text{len}(L2))$

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

$O(\text{len}(L1)^2)$

QUADRATIC COMPLEXITY:

EXAMPLE 3

- 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)  
    unique = []  
    for e in tmp:  
        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
 $O(\text{len}(L1) * \text{len}(L2))$ steps.

Second loop takes at most
 $O(\text{len}(L1) * \text{len}(L2))$ steps.
Typically not this bad.

Overall **$O(\text{len}(L1) * \text{len}(L2))$**

EXPONENTIAL COMPLEXITY

EXPONENTIAL COMPLEXITY

- Recursive functions where have more than one recursive call for each size of problem
 - Fibonacci
- 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 $2^{n-1} + 2^{n-2} + \dots + 2^0$ 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
```

constant
 $O(1)$

```
    else:
```

```
        fib_i = 0  
        fib_ii = 1
```

constant
 $O(1)$

```
        for i in range(n-1):  
            tmp = fib_i  
            fib_i = fib_ii  
            fib_ii = tmp + fib_i
```

linear
 $O(n)$

```
        return fib_ii
```

constant
 $O(1)$

- Best case:

$O(1)$

- Worst case:

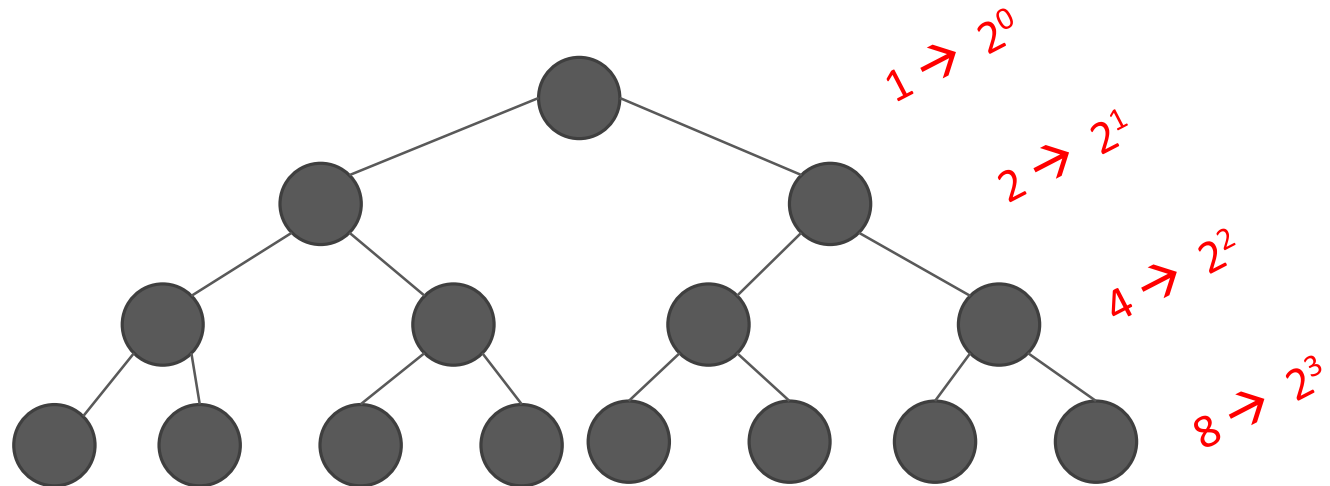
$O(1) + O(n) + O(1) \rightarrow O(n)$

COMPLEXITY OF RECURSIVE FIBONACCI

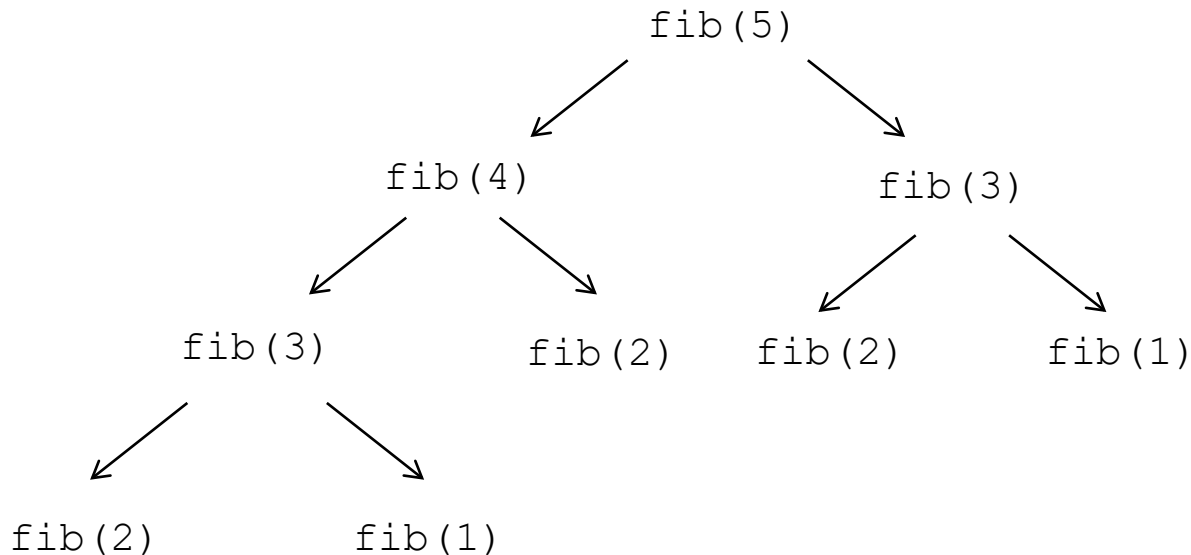
```
def fib_recur(n):  
    """assumes n an int >= 0 """  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib_recur(n-1) + fib_recur(n-2)
```

- 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



TEST YOURSELF

```
def all_digits(nums):  
    """ nums is a list of numbers """  
    digits = [0,1,2,3,4,5,6,7,8,9]  
    for i in nums:  
        isin = False  
        for j in digits:  
            if i == j:  
                isin = True  
                break  
        if not isin:  
            return False  
    return True
```

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

5 Min Break, then Quiz Time!

- Sit at a seat, **not on the floor**
- No aids allowed, **only MITx and your IDE**
- If you finish early, **stay in your seat** (no phones, external websites, etc)
- **Checkout password given in the last 2 mins of exam**