What is efficiency in programming?

Why efficiency is important?

Types of efficiency

Space and Time Efficiency

Our focus – Time

Techniques to measure time efficiency

Techniques

1. Measuring **time** to execute
2. **Counting** operations involved
3. Abstract notion of **order of growth**

1. Measuring Time

Problems with this approach

Different time for different algorithm	✔
Time varies if implementation changes	✖
Different machines different time	✖
Does not work for extremely small input	✖
Time varies for different inputs, but can’t establish a relationship	✖

2. Counting Operations

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

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+3x ops

Problems with this approach

Different time for different algorithm	✔
Time varies if implementation changes	✖
Different machines different time	✔
No clear definition of which operation to count	✖
Time varies for different inputs, but can’t establish a relationship	✔

What do we want

1. We want to evaluate the algorithm
2. We want to evaluate scalability
3. We want to evaluate in terms of input size

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

3. Orders of Growth

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**

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

So the idea is simple

$n^2 + 2n + 2$

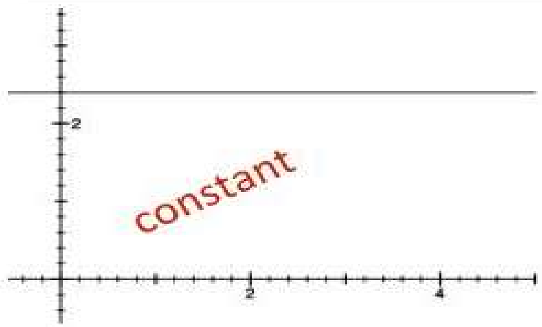
$n^2 + 100000n + 3^{1000}$

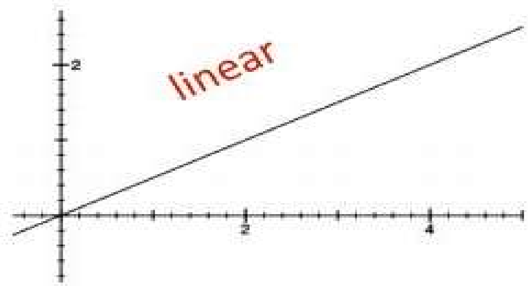
$\log(n) + n + 4$

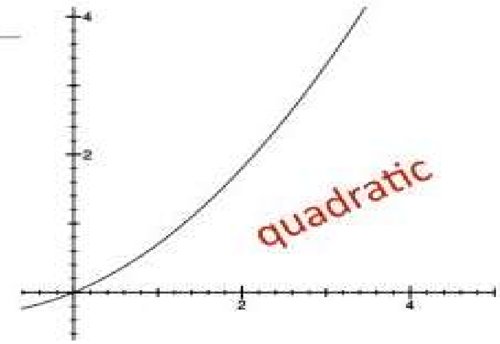
$0.0001 * n * \log(n) + 300n$

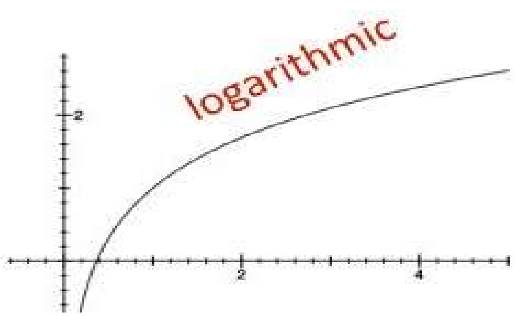
$2n^{30} + 3^n$

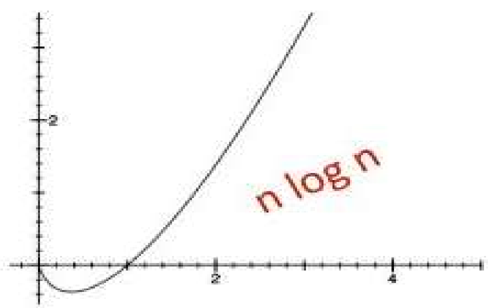
TYPES OF ORDERS OF GROWTH

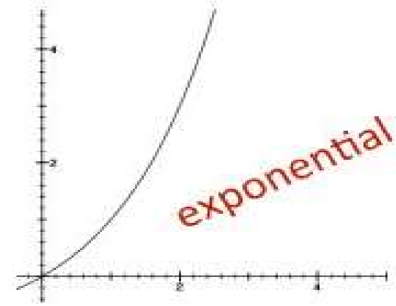












Law of addition

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

Law of multiplication

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.

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 64776865421676604298316 526243868372056680693 76	**Good luck!!**