# Measuring complexity

- Goals in designing programs
  1. It returns the correct answer on all legal inputs
  2. It performs the computation efficiently
- Typically (1) is most important, but sometimes (2) is also critical, e.g., programs for collision detection
- Even when (1) is most important, it is valuable to understand and optimize (2)

## Computational complexity

- How much time will it take a program to run?
- How much memory will it need to run?

- Need to balance minimizing computational complexity with conceptual complexity
  - Keep code simple and easy to understand, but where possible optimize performance

# How do we measure complexity?

- Given a function, would like to answer: “How long will this take to run?”
- Could just run on some input and time it.
- Problem is that this depends on:
  1. Speed of computer
  2. Specifics of Python implementation
  3. Value of input
- Avoid (1) and (2) by measuring time in terms of number of basic steps executed

## Measuring basic steps

- Use a **random access machine (RAM)** as model of computation
  - Steps are executed sequentially
  - Step is an operation that takes constant time
    - Assignment
    - Comparison
    - Arithmetic operation
    - Accessing object in memory
- For point (3), measure time in terms of size of input

## But complexity might depend on value of input?

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

- If x happens to be near front of L, then returns True almost immediately
- If x not in L, then code will have to examine all elements of L
- Need a general way of measuring

## Cases for measuring complexity

- **Best case:** minimum running time over all possible inputs of a given size
  - For linearSearch – constant, i.e. independent of size of inputs
- **Worst case:** maximum running time over all possible inputs of a given size
  - For linearSearch – linear in size of list
- **Average (or expected) case:** average running time over all possible inputs of a given size
- We will focus on worst case – a kind of **upper bound** on running time

# Example

```
def fact(n):
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

- Number of steps
  - 1 (for assignment)
  - 5*n (1 for test, plus 2 for first assignment, plus 2 for second assignment in while; repeated n times through while)
  - 1 (for return)
- 5*n + 2 steps
- But as n gets large, 2 is irrelevant, so basically 5*n steps

## Example

- What about the multiplicative constant (5 in this case)?
- We argue that in general, multiplicative constants are not relevant when comparing algorithms

## Example

```
def sqrtExhaust(x, eps):
    step = eps**2
    ans = 0.0
    while abs(ans**2 - x) >= eps and ans <= max(x, 1):
        ans += step
    return ans
```

- If we call this on 100 and 0.0001, will take one billion iterations of the loop
  - Have roughly 8 steps within each iteration

# Example

```
def sqrtBi(x, eps):
    low = 0.0
    high = max(1, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= eps:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans
```

- If we call this on 100 and 0.0001, will take thirty iterations of the loop
  - Have roughly 10 steps within each iteration
- 1 billion or 8 billion versus 30 or 300 – it is size of problem that matters

## Measuring complexity

- Given this difference in iterations through loop, multiplicative factor (number of steps within loop) probably irrelevant
- Thus, we will focus on measuring the complexity as a function of input size
  - Will focus on the largest factor in this expression
  - Will be mostly concerned with the worst case scenario