Introduction to Algorithmic Complexity Analysis

Guttag Chapter 11

Goals

- 1. Understand the need for complexity analysis
- 2. Learn approach for deriving complexity
- 3. Review vocabulary
- 4. Orders of growth

Next class

- Concept of doubling experiments
- Analysis of common algorithms

1. Why do we need Complexity Analysis?

Guttag Chapter 11

1. Why Complexity Analysis

Question: How long will it take the function to run?

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

1. Why Complexity Analysis

Two Answers:

- time it → **empirical** analysis (based on observations obtained in experiments)
- reason it out → **analytical complexity** analysis (based on inherent structures/relationships)

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

Critical Thinking

What are the pros and cons of empirical and analytical approaches to characterizing the algorithmic speed?

Critical Thinking

What are the pros and cons of empirical and analytical approaches to characterizing the algorithmic speed?

- empirical pro: hard numbers can be directly compared
- empirical pro: any algorithm can be timed in a straightforward way
- empirical con: have to pick input sizes
- empirical con: have to have a computer
- analytical pro: no computer needed
- analytical pro: no input needed
- analytical con: have to have a deep understanding of data manipulations
- analytical con: hard to know efficiency of all operations

2. Approach for Deriving Complexity

Guttag Chapter 11

2. Analytical Approach to Complexity Analysis

- count number of steps in algorithm

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1                  ← one step (assignment)
    while i >= 1:               ← one step (comparison)
        answer *= i             ← two steps (multiplication and assignment)
        i -= 1                  ← two steps (subtraction and assignment)
    return answer               ← one step (return)
```

2. Analytical Approach to Complexity Analysis

- count number of steps in algorithm
- look for repetition

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

← one step (assignment)

← one step (comparison)

← two steps (multiplication and assignment)

← two steps (subtraction and assignment)

← one step (return)

2. Analytical Approach to Complexity Analysis

- count number of steps in algorithm
- look for repetition
- relate the number of steps inside the algorithm to the size of the input

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

← one step (assignment)

← one step (comparison)

← two steps (multiplication and assignment)

← two steps (subtraction and assignment)

← one step (return)

i repeats, no matter what i is

2. Analytical Approach to Complexity Analysis

- count number of steps in algorithm
- look for repetition
- relate the number of steps inside the algorithm to the size of the input
- simplify the expression

steps = 1 + i*(1+2+2) + 1 + 1
steps is **linearly** related to i
therefore algorithmic complexity is linear

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

← one step (assignment)

← one step (comparison)

← two steps (multiplication and assignment)

← two steps (subtraction and assignment)

← one step (return)

i repeats, no matter what i is

Example 2: Analytical Approach to Complexity Analysis

```
n = 5
steps = 0
for apple in range(n):
    for pear in range(n):
        steps +=1
print("steps:", steps)
```

← one step (assignment)

← one step (assignment)

← two steps? (assignment and range progression)

← two steps? (assignment and range progression)

← two steps (addition and assignment)

steps = 1 + 1 + n*(n*(4))
drop scalars and constants
steps is quadratically related to n
therefore complexity is quadratic, or grows as n^2

3. Terms

Guttag Chapter 11

Linear (Math Version)

Definition:

- output is scaled and/or shifted version of the input

Examples:

- input x, output y
- y = x
- y = 2x
- y = 3x
- y = 0.4x + 2

- scaling does not matter
- other constants do not matter

Linear (Code Complexity Version)

Definition:

- output (number of steps) is scaled and/or shifted version of the input (size of the data container, or repetitions)

Examples:

- input i is number of repetitions, output complexity is number of algorithmic steps

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1                          ← one step (assignment)
    while i >= 1:                       ← one step (comparison)
        answer *= i                     ← two steps (multiplication and assignment)
        i -= 1                          ← two steps (subtraction and assignment)
    return answer                       ← one step (return)
```

i repeats, no matter what i is

- scaling does not matter
- other constants do not matter

Quadratic (Math Version)

Definition:

- output is a squared, and possible scaled and/or shifted version of the input

Examples:

- input x, output y
- $y = x^2$
- $y = 2x^2$
- $y = 3x^2$
- $y = 0.4x^2 + 2$

- scaling does not matter
- other constants do not matter

Quadratic (Code Complexity Version)

Definition:

- output (number of steps) is a squared, and possibly scaled and/or shifted version of the input (size of the data container, or repetitions)

Examples:

- input n is number of repetitions, output complexity is number of algorithmic steps

```
n = 5                                   ← one step (assignment)
steps = 0                               ← one step (assignment)
for apple in range(n):                  ← two steps? (assignment and range progression)
  for pear in range(n):                 ← two steps? (assignment and range progression)
    steps +=1                           ← two steps (addition and assignment)
print("steps:", steps)
```

- scaling does not matter
- other constants do not matter

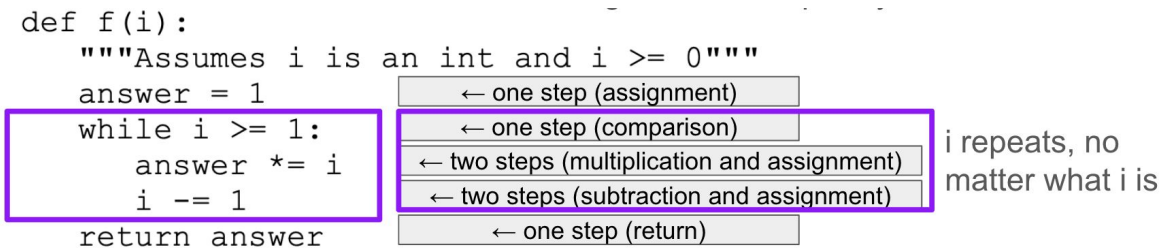
Asymptotic Analysis

Definition:

- fancy term for simplifying mathematical expressions when then get huge or tiny
- in code, we really only think about huge

Example:

- think about looping over an input i that is ∞
- the loop becomes ∞ long
- # steps = 1 + i*(1+2+2) + 1 + 1 = 3 + 5i
  - **+ 3 does not matter**
  - *** 5 does not matter**



Big O notation

Definition:

- way to write down the algorithmic complexity conveniently
- way to tell others about algorithms efficiency

Example:

- **O(n) is said, Oh of N**, linear, where n is the size of the input
- **O(n^2) is sai, Oh of N squared**, quadratic, where n is the size of the input

  - $O(1)$ 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.
  - $O(n^k)$ denotes polynomial running time. Notice that k is a constant.
  - $O(c^n)$ denotes exponential running time. Here a constant is being raised to a power based on the size of the input.

Orders of Growth

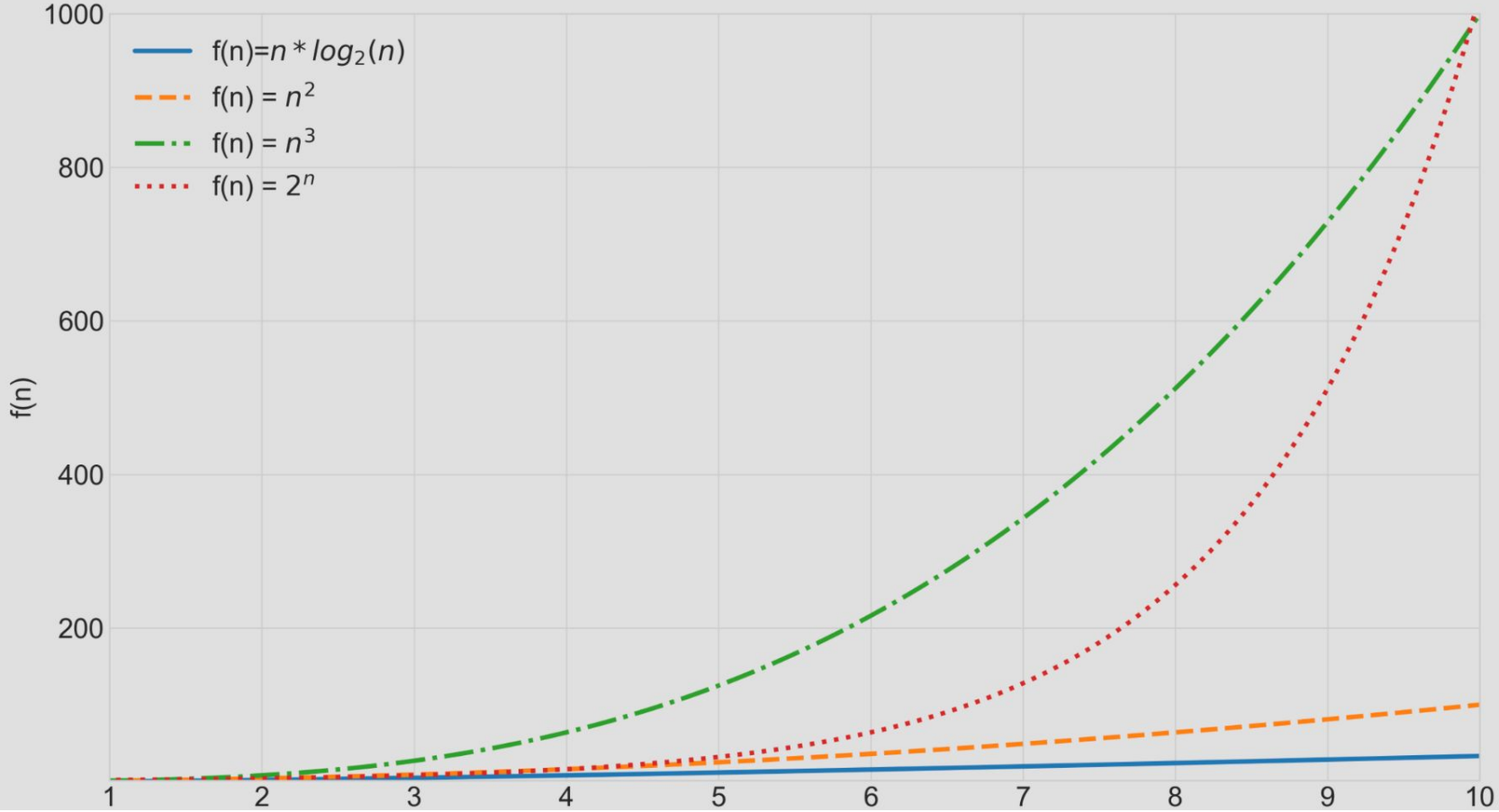
Definition:

- The different complexity categories that algorithms fall into.
- Each order of growth could also be identified with big O

Example:

- **linear** - for O(n) algorithms
- **quadratic** - for $O(n^2)$ algorithms
- **logarithmic** - for O(log(n)) algorithms
- **log-linear** - for O(n log(n)) algorithms
- **polynomial** - for $O(n^c)$ algorithms
- **exponential** - for $O(c^n)$ algorithms

"Fast" Order of Growth Functions



"Slow" Order of Growth Functions