Introduction to Algorithmic Complexity Analysis

Guttag Chapter 11

Goal

- Programmers and clients want algorithms to be correct and fast
- Analyzing the **complexity** of an algorithms will give us an idea of speed

How should one go about answering the question "How long will the following function take to run?"

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

Two approaches:

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

Analytical Approach to Complexity Analysis

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

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

Analytical Approach to Complexity Analysis

- count number of steps in algorithm

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

`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)

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                    ← 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)
```

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                ← 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

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                 ← 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

Linear

the number of steps depends on the size of the input

- scaling does not matter
- other constants do not matter
- think about approaching an input that is length ∞
    - **+ 1 does not matter**
    - *** 3 does not matter**
- ^^^ Asymptotic analysis

Big O notation

linear: O(n), where n is the size of the input

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

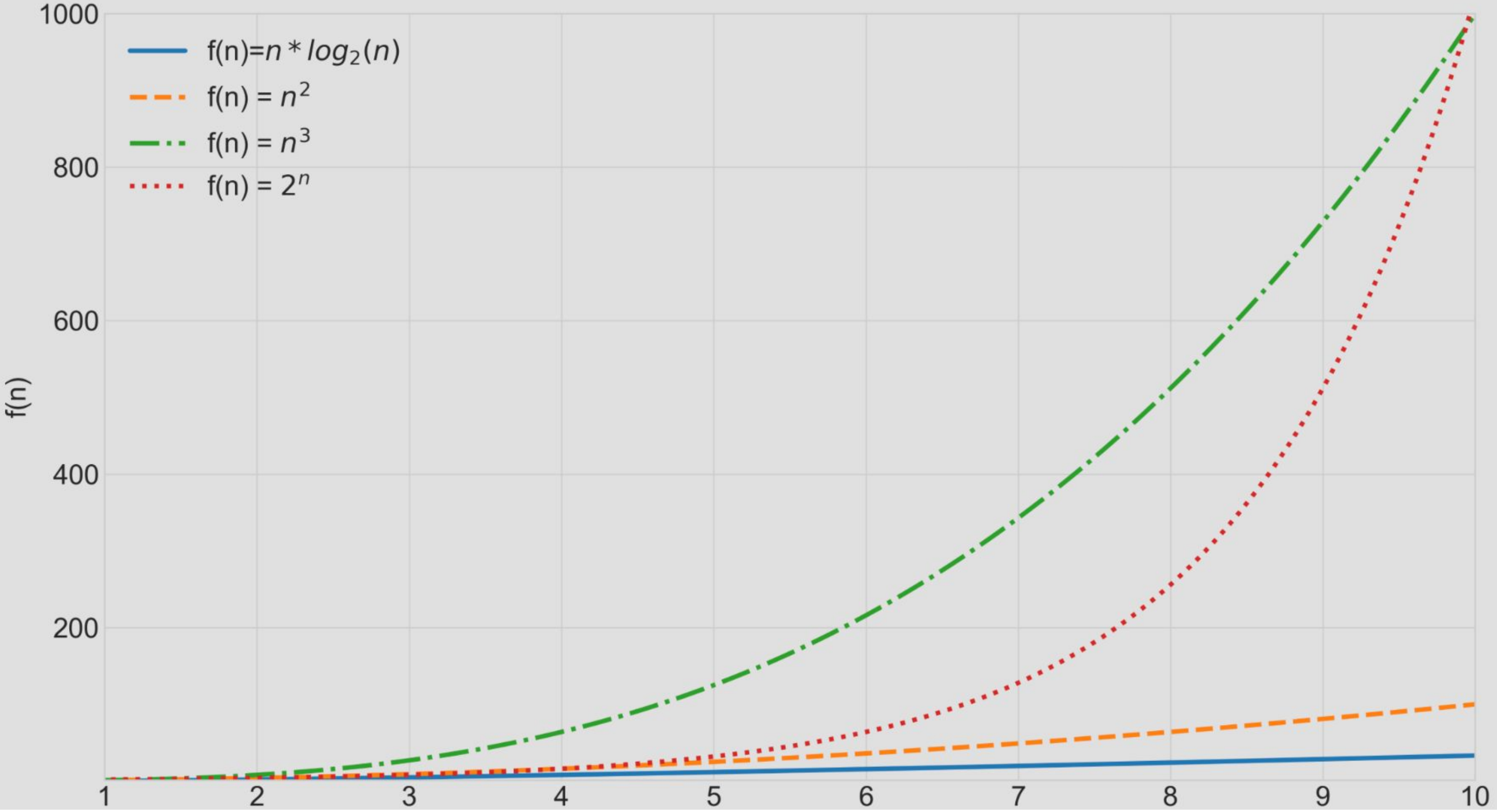
Big O notation

linear: O(n), where n is the size of the input

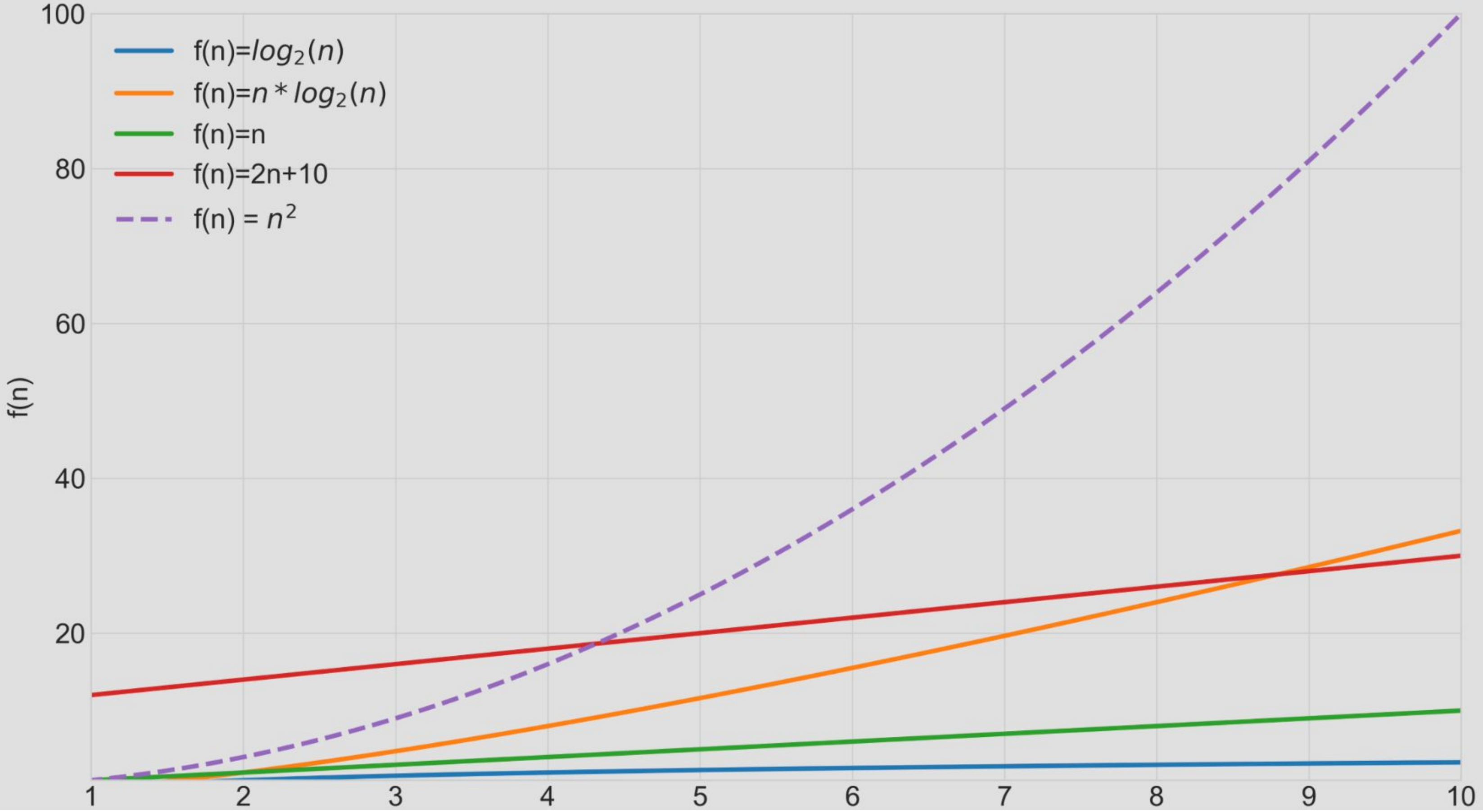
quadratic: O(n^2), where n is the size of the input

- $O(1)$ 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.

"Fast" Order of Growth Functions



"Slow" Order of Growth Functions



Example 3: Analytical Approach to Complexity Analysis

Exhaustive search

- $x/(\text{epsilon}^2)$ times through the loop…technically linear but $1/(\text{epsilon}^2)$ dominates

Bisection Search

- approx $\mathbf{log_2(x)}$ times through the loop

For x = 100, epsilon = 0.0001,
Exhaustive search → 10 billion steps
Bisection search → approx 10 steps

```
def square_root_exhaustive(x, epsilon):
    """Assumes x and epsilon are positive floats & epsilon < 1
       Returns a y such that y*y is within epsilon of x"""
    step = epsilon**2
    ans = 0.0
    while abs(ans**2 - x) >= epsilon and ans*ans <= x:
        ans += step
    if ans*ans > x:
        raise ValueError
    return ans
```

```
def square_root_bi(x, epsilon):
    """Assumes x and epsilon are positive floats & epsilon < 1
       Returns a y such that y*y is within epsilon of x"""
    low = 0.0
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans
```

Example 4: Analytical Approach to Complexity Analysis

- Powerset function
- for a set of {A,B,C,D,E,F} the powerset is a set with all possible subsets:
- {{}, {A}, {B}, {A,B}, {C}, {A,C}, {B,C}, {A,B,C},{D}, {A,D}, {B,D}, {A,B,D}, {C,D}, {A,C,D}, {B,C,D}, {A,B,C,D}, {E}, {A,E}, {B,E}, {A,B,E}, {C,E}, {A,C,E}, {B,C,E}, {A,B,C,E},{D,E}, {A,D,E}, {B,D,E}, {A,B,D,E}, {C,D,E}, {A,C,D,E}, {B,C,D,E}, {A,B,C,D,E}, {F}, {A,F}, {B,F}, {A,B,F}, {C,F}, {A,C,F}, {B,C,F}, {A,B,C,F},{D,F}, {A,D,F}, {B,D,F}, {A,B,D,F}, {C,D,F}, {A,C,D,F}, {B,C,D,F}, {A,B,C,D,F}, {E,F}, {A,E,F}, {B,E,F}, {A,B,E,F}, {C,E,F}, {A,C,E,F}, {B,C,E,F}, {A,B,C,E,F},{D,E,F}, {A,D,E,F}, {B,D,E,F}, {A,B,D,E,F}, {C,D,E,F}, {A,C,D,E,F}, {B,C,D,E,F}, {A,B,C,D,E,F}}
- order does not matter in set (combinations, not permutations)
- **Growth of output is exponentially related to input size: $O(2^n)$**

Pros and Cons for Analytical Approach

- Neutral: Assumption that every step takes the same amount of time, like **Random Access Machine**
- Pro: does not depend on specific laptop or OS
- Pro: makes algorithms comparable
- Con: does not relate to real world time
- Con: typically relates to worst case, not average case

Challenges of analytical algorithm evaluation?

- ! Operations may not require equivalent time
- ! Operation counting may be tough with loops
- ! Counting with recursion requires alternative

Worst case analysis

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

Suppose that `L` is a list containing a million elements, and consider the call `linear_search(L, 3)`. If the first element in `L` is `3`, `linear_search` will return `True` almost immediately. On the other hand, if `3` is not in `L`, `linear_search` will have to examine all one million elements before returning `False`.

Questions

What is the relationship between growth function and program's performance?

- Slow growth functions → fast programs
- Fast growth functions → slow programs

Empirical Tests

- Use profiles, or other timers to run doubling experiments

Doubling Experiment: Linear

Double the size of the program's input

14.98 seconds

31.45 seconds

Doubling ratio is approximately 2

Likely worst-case time complexity is O(n)

Doubling Experiment: Quadratic

Double the size of the program's input

12.63 seconds

51.48 seconds

Doubling ratio is approximately 4

Likely worst-case time complexity is O(n^2)

Doubling Experiment: Cubic

Double the size of the program's input

11.23 seconds

89.72 seconds

Doubling ratio is approximately 8

Likely worst-case time complexity is O(n^3)

Suitable complexity class for an algorithm?

- Constant is exceptional but rarely attainable
- Logarithmic, linear, or linearithmic are very good
- Quadratic or greater suggests likely infeasibility