# Algorithmic Complexity Continued

### Goals

- Cover concept of doubling experiments
- Analyze common algorithms
- Code Exploration

## Doubling Experiment

## **Doubling Experiment**

#### Definition

 a thought experiment, or actual experiment, where input is doubled, and complexity behavior is characterized empirically!

#### Example

- say an algorithm has an unknown complexity
- when n is 10,000, the timer reports 3 seconds
- when n is doubled to 20,000, the timer reports 6 seconds
- when n is double again to 40,000 the time reports 12 seconds
- the ratio for the outputs was 6/3 = 2 and 12/6 = 2
- therefore, for each doubling, the output time scales linearly  $\rightarrow$  O(n)

## **Doubling Experiment**

#### Definition

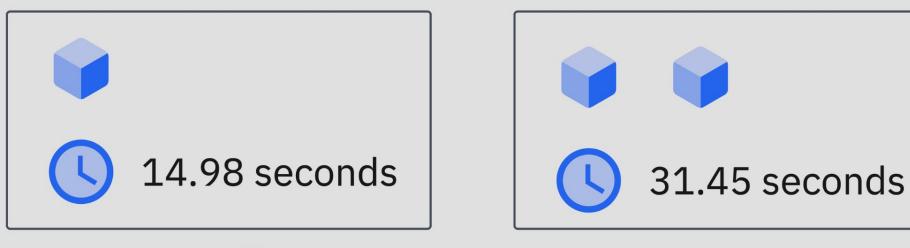
 a thought experiment, or actual experiment, where input is doubled, and complexity behavior is characterized empirically!

#### Example 2

- say an algorithm has an unknown complexity
- when n is 10,000, the timer reports 3 seconds
- when n is doubled to 20,000, the timer reports 12 seconds
- when n is double again to 40,000 the time reports 48 seconds
- the ratio for the outputs was 12/3 = 4 and 48/12 = 4
- therefore, for each doubling, the output time scales by  $2^2 \rightarrow O(n^2)$

## Doubling Experiment: Linear

## Double the size of the program's input



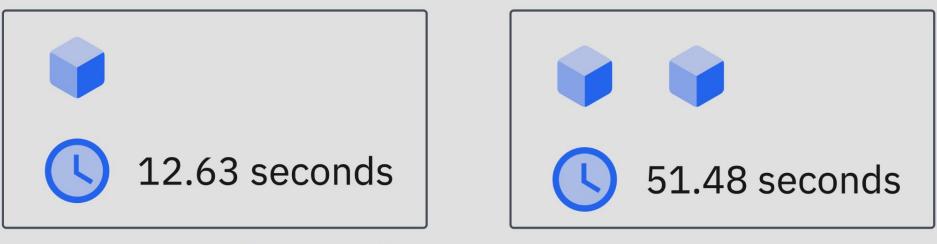
## Doubling ratio is approximately 2



Likely worst-case time complexity is O(n)

## Doubling Experiment: Quadratic

## Double the size of the program's input



## 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





## Doubling ratio is approximately 8



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

## **Critical Thinking**

linear:  $n \rightarrow 2n$  everything doubles

quadratic:  $n^2 \rightarrow (2n)^2 = 2^2 * n^2$  everything scaled by factor of 4

logarithmic:  $\log_2(n) \rightarrow \log_2(2n) = \log_2(2) + \log_2(n) = 1 + \log_2(n)$  shifting ratios

Work out 3 doubling ratios starting with n = 4 going through an  $O(log_2(n))$  algorithm

# Common Algorithms & Complexity

## Basic Loop

```
def example(n: int):
    counter = 0
    for i in range(n):
        # do a constant number of things
        counter += 1
        print(counter)
```

- input was n
- counter goes up to n
- algorithm is O(n)

### **Linear Search**

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

This is a linear algorithm, because in the worst case, every number will have to be examined once!

Complexity analysis is "worst-case" analysis

## Basic Double Loop

```
counter = 0
for i in range(n):
    for j in range(n):
      # do a constant number of things
      counter += 1
      print(counter)
```

- input was n
- but counter goes up to n<sup>2</sup>
- algorithm is O(n<sup>2</sup>)

## Square Root Algorithms (exhaustive, bisection)

#### Exhaustive search

 x\*epsilon² times through the loop...technically linear but epsilon² dominates

#### **Bisection Search**

approx log<sub>2</sub>(x) times through the loop

For x = 100, epsilon = 0.0001, Exhaustive search  $\rightarrow$  10 billion steps Bisection search  $\rightarrow$  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</pre>
```

#### **Powerset**

#### Definition:

- for a given set, the powerset is a set with <u>all</u> possible <u>subsets</u>
- order does not matter

#### Example:

- find the powerset of {A,B,C,D,E,F}
- {{}, {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}, {A,B,F}, {C,F}, {A,C,F}, {B,C,F}, {A,B,C,F}, {B,D,F}, {A,B,D,F}, {C,D,F}, {A,C,D,F}, {B,C,D,F}, {A,B,C,D,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}, {A,D,E,F}, {B,D,E,F}, {A,B,D,E,F}, {C,D,E,F}, {B,C,D,E,F}, {A,B,C,D,E,F}, {A,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}, {A,B,C,D,E,E}, {A,B,C,
- Growth of output is exponentially related to input size: O(2<sup>n</sup>)

## **Explore Code**