COR-IS1702:
COMPUTATIONAL THINKING
WEEK 3: COMPLEXITY

2020/21 Term 1

(03) Complexity Part 1a

Video (14 mins):

https://www.youtube.com/watch?v=9V4QIICezeg&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=21&t=0s

Road Map

Algorithm Design and Analysis

- Week 1: Counting, Programming
- Week 2: Programming
- This week → Week 3: Complexity
- Week 4: Iteration & Decomposition
- Week 5: Recursion

Fundamental Data Structures

(Weeks 6 - 10)

Computational Intractability and Heuristic Reasoning

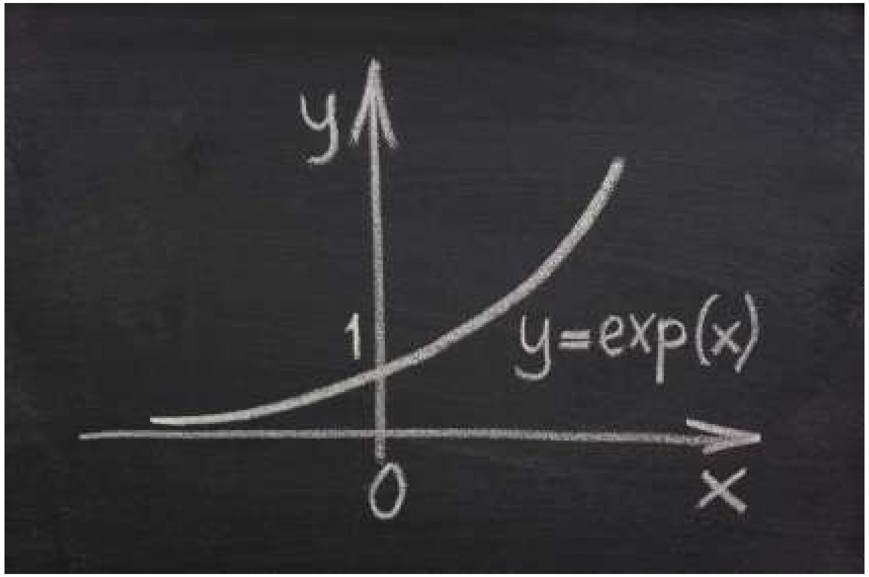
(Weeks 11 - 13)

References

- Supplementary materials:
  - Prichard and Carrano, Chapter 10.1 “Measuring the Efficiency of Algorithms”
  - Course Reserve (2 hours) at SMU Library:
    - http://catalogue.library.smu.edu.sg/record=b1105798
- Online sources:
  - http://en.wikipedia.org/wiki/Big_O_notation
  - http://en.wikipedia.org/wiki/Time_complexity
  - http://web.mit.edu/16.070/www/lecture/big_o.pdf
  - http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html

Complexity

Characterizing the efficiency of algorithms



- Measurement of Efficiency
- Big O notation
- Orders of Complexity

Understanding *Complexity*

2018 Winter Olympics Torch Relay

https://en.wikipedia.org/wiki/2018_Winter_Olympics_torch_relay

- Huge undertaking
  - 80 localities in every corner of South Korea from Incheon to Pyeongchang
  - 7,500 runners covering 2,018 km
- How to compute the “optimal” route?
  - $80! = 7.2 \times 10^{118}$ possible permutations
  - With 1 Billion permutations per second, it will take 10^{100} centuries to try every one.

Algorithms defined

- A specification for how to carry out a computation
  - an algorithm can be thought of as a “prescription”, follow these steps and you will solve your problem
- An algorithm contains a complete description of
  - the set of ***inputs***, or starting conditions
    - a full specification of the problem to be solved
  - the set of ***outputs***
    - descriptions of valid solutions to the problem
  - a sequence of ***operations*** that will eventually produce the output
    - steps must be simple and precise
- An algorithm may be described as a program, pseudo-code or a less formal step-by-step explanation.

Attributes of Algorithms

- What do we mean by "a sequence of simple and precise steps" ?
  - **precise**: they must be written in terms understandable by anyone
    - but what does "precise" mean? how precise does a step have to be?
  - **effective**: a step must help the algorithm progress to the final goal
    - but how effective? is there a formal definition of "effective"?
  - **practical**: a sequence of precise and effective steps may not be useful in practice
    - example (from Knuth): a hypothetical algorithm for winning a chess tournament: "for each game, consider all possible moves, choose the best"

Algorithm Supports Abstraction

- Input-Output contract:
  - accepts a set of allowable inputs
  - returns a valid output

There can be more than one algorithm to transform the input to the desired output

Problem: Greatest Common Divisor (`gcd`)

Given two numbers not prime to one another, to find their greatest common measure.

Proposition VII.2 in the Elements

- gcd(4, 8) = ?
- gcd(81, 36) = ?
- gcd(1989, 867) = ?

Problem: Greatest Common Divisor (`gcd`)

- **Inputs:**
  - two non-negative integers `a` and `b`
- **Algorithm:**

?

- **Output:**
  - The `gcd` of `a` and `b`

First Algorithm: Brute Force Algorithm

For two numbers `a` and `b`, its `gcd` is between `1` and the minimum of `a` and `b`.

Brute Force: try all reasonable possibilities.

First Algorithm: Brute Force Algorithm

- **Inputs:**
  - two non-negative integers `a` and `b`
- **Algorithm:**
  - set `t` to be the minimum of `a` and `b`
  - repeat until `t` equals 1:
    - if `a` and b are both divisible by `t`, return `t` as output
    - else, subtract `1` from `t`
- **Output:**
  - `gcd` of the original `a` and `b`

How many repetitions are needed to compute `gcd`(81, 36)?

28 times

Second Algorithm: Dijkstra's Algorithm

gcd of two numbers are unchanged if the smaller number is subtracted from the larger number

For two integers a and b, if a > b, then gcd(a, b) = gcd(a−b, b).

gcd(81, 36) = gcd(81 - 36, 36) = gcd(45, 36)
= gcd(45 - 36, 36) = gcd(9, 36)
= gcd(9, 36 - 9) = gcd(9, 27)
= gcd(9, 27 - 9) = gcd(9, 18)
= gcd(9, 18 - 9) = gcd(9, 9) = 9

Second Algorithm: Dijkstra's Algorithm

- **Inputs:**
  - two non-negative integers `a` and `b`
- **Algorithm:**
  - repeat until `a` is equal to `b`:
    - if `a` is larger than `b`, subtract `b` from `a`
    - else, subtract `a` from `b`
  - return `a` as output
- **Output:**
  - `gcd` of the original `a` and `b`

How many repetitions are needed to compute `gcd`(81, 36)?

5 times

Exercise: Comparisons

	Number of Repetitions		
	`gcd`(81,36)	`gcd`(330, 231)	`gcd`(1989, 867)
Brute Force	28		
Dijkstra's	5		

Same input. Same output.
Different number of operations.

Exercise: Comparisons

	Number of Repetitions		
	`gcd`(81,36)	`gcd`(330, 231)	`gcd`(1989, 867)
Brute Force	28	199	817
Dijkstra’s	5	5	8

Same input. Same output.
Different number of operations.

What makes a good algorithm?

- Correctness:
  - Produces the right output given the input
- Efficiency:
  - Uses as little memory as possible (efficient in space)
    - Related to data structures, which will be discussed later in the course
  - Runs as fast as possible (efficient in time)
    - Focus of this lesson

How NOT to measure efficiency

- C++ vs. Java
  - efficiency is not about the programming language used
- PC vs. Mac
  - efficiency is not about the operating system
- Intel Core i5 vs. i7
  - efficiency is not about the processor speed
- Andy's implementation vs. Barbara's implementation
  - efficiency is not about different implementations of the same algorithm
- Using 50% of the data vs. using 100% of the data
  - efficiency is not about the relative sizes of input

Performance is a function of input size

- An algorithm's time performance is a function of the input size
  - Most of the time, we use the letter **n** to represent the input size
- Example:
  - Problem: Find the largest number in an array `a` of `n` numbers
  -
    ```
    def findMax(a):
        max = a[0]
        for i in range(1, len(a)):
            if max < a[i]:
                max = a[i]
        return max
    ```
  - Number of 'comparison' operations is (n – 1)
  - Number of 'assignment' operations is at most n

In-Class Exercises: How many operations?

(a) Given an array `a` of `n` numbers, where `n > 10`, find out which of the first `10` numbers is the largest.

(b) Given an array `a` of `n` numbers, find the smallest difference between any two numbers in the array `a`.

In-Class Exercises: How many operations?

Given an array `a` of `n` numbers, where `n > 10`, find out which of the first `10` numbers is the largest.

```
def findMaxTen(a):
    max = a[0]
    for i in range(1, 10):
        if max < a[i]:
            max = a[i]
    return max
```

- Number of 'comparison' operations is …

In-Class Exercises: How many operations?

Given an array `a` of `n` numbers, find the smallest difference between any two numbers in the array `a`.

```
def findMinDiff(a):
    mindiff = abs(a[0]-a[1])
    for i in range(0, len(a)-1):
        for j in range(i+1, len(a)):
            diff = abs(a[i]-a[j])
            if mindiff > diff:
                mindiff = diff
    return mindiff
```

- Number of 'comparison' operations is …
- Number of 'assignment' operations is …
- Number of 'subtraction' operations is …

(03) Complexity Part 1b

Solution to In-class Exercise

Video (7 mins):

https://www.youtube.com/watch?v=n7kugeKYvVE&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=21