COMP1819
Algorithms and Data Structures

Lecture 02: Analysis of Algorithm

Dr. Tuan Vuong

25/01/2021

Content

- Lab 01 Discussion
- What is Algorithm Analysis?
- BigO
- A true winner for Lab01
- Reinforcement

Lab 01

You can check for sample solutions here: https://github.com/vptuan/COMP1819ADS

1. MinMax function

Write a short Python *function*, minmax(data), that takes a sequence of one or more numbers, and returns the smallest and largest numbers, in the form of a tuple of length two.

Examples

Input	Output
`1 2 3 5`	`(1, 5)`
`-2 0 1`	`(-2, 1)`
`3`	`(3, 3)`

Hints

- You can use the built-in functions min or max in implementing your solution.
- Can you try **NOT** to use the built-in functions min or max in implementing your solution?

Good programs: correct, finite, terminate, unambiguous. We should focus on solving problems efficiently.

2. Staircase

Consider a staircase of size **n** = 3:

```
  #
 ##
###
```

Observe that its base and height are both equal to **n**, and the image is drawn using `#` symbols and spaces.

Write a program that prints a staircase of size **n**.

Constraints: 0 < n <= 20

Examples

Input	Output
2	# ##

Hints

- Input can come hardcoded or from keyboard.

Debug to understand code.

3. Lucky Winner

Now you are provided with a text file that has 20 lucky Banner IDs randomly selected for today. Write a program to check if your ID are listed in this file.

Examples:

Input	Output	Comments

`A text file` **`lucky_ids.txt`**`:`
`001059317`
`001086770`
`001087316`
…
	`Yes - if your id is listed`
	`No - if your id is NOT listed`
		`… means there are more ids followed`

Hints

- You can create your own text file and try IDs as positive integer first to simplify the problem.
- **`lucky_ids.txt`** file is provided in Moodle with this lab instruction.

4. Top 3 Max Lucky Winners

Now you are provided with a text file that has 20 lucky Banner ids randomly selected for today. Write a program to output the top 3 max ids without rearranging the list.

Examples

Input	Output	Comments
1 6 3 8 9 10 7 4	10 9 8	Ids are Banner IDs provided in the **`lucky_ids.txt`**, `each ids in its own line`

Extra

- Discuss with your friends if your algorithm is good for one million records? How can you improve your algorithm?

"Without rearranging the list" = no sorting

5. Duplicated Banner Id

Note that the lucky ids are selected but there was a mistake. There is at least one duplicated id in the file. Write a program to detect the duplicated value.

Examples

Input	Output	Comments
1 2 3 4 5 3 7 8	3	Ids are provided in the **`lucky_ids.txt`**`, each ids in its own line`

Extra

- Discuss with your friends if your algorithm is good for one million records? How can you improve your algorithm?
- Did you use rearrangement (sorting) of the list? Can you do it without sorting?

Built-in function vs. your own code solution.

5. Duplicated Banner Id

Note that the lucky ids are selected but there was a mistake. There is at least one duplicated id in the file. Write a program to detect the duplicated value.

Examples

Input	Output	Comments
1 2 3 4 5 3 7 8	3	Ids are provided in the **`lucky_ids.txt`**, `each ids in its own line`

Extra

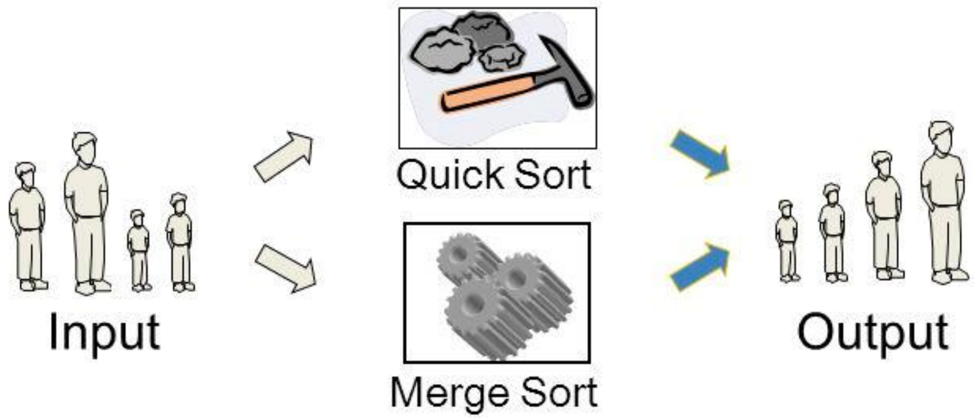
- Discuss with your friends if your algorithm is good for one million records? How can you improve your algorithm?
- Did you use rearrangement (sorting) of the list? Can you do it without sorting?

Built-in function vs. your own code solution.

Analysis of Algorithms

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

How to evaluate algorithms?

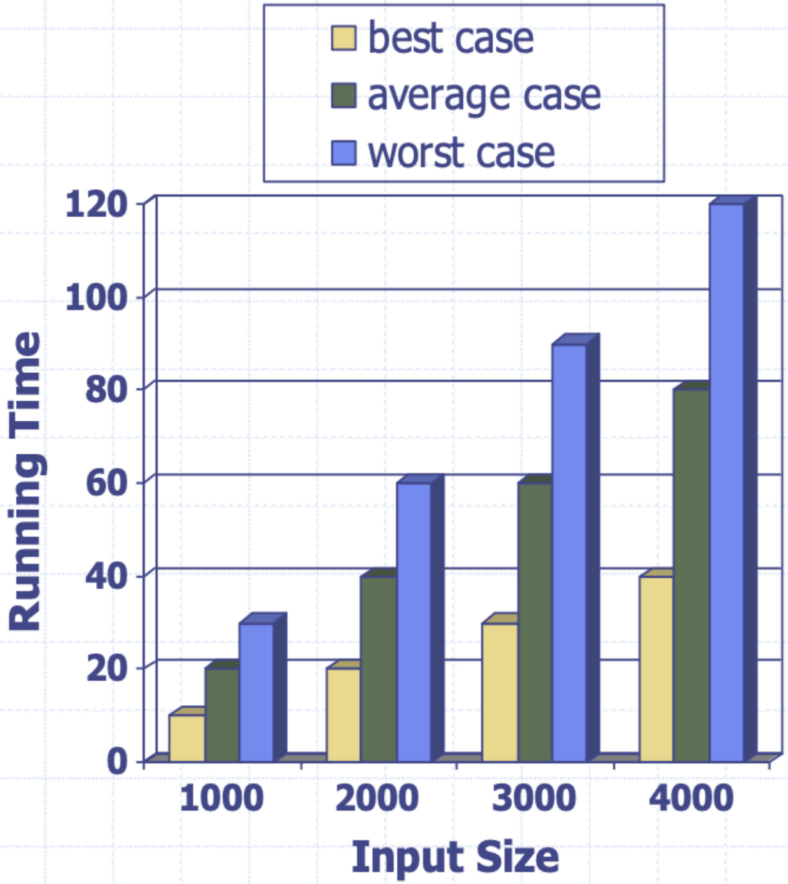


- Which one is better?
- What are the criteria?

Good programs: correct, finite (meaning?), terminate, unambiguous. We should focus on solving problems efficiently.

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Running time for Lab 01 – Exercise 3: Lucky winner run.py util.py

```
"""Using time function."""


import time

def test_run():
    t1 = time.time()
    print "ML4T"
    t2 = time.time()
    print "The time taken by print statement is ",t2 - t1," seconds"

if __name__ == "__main__":
    test_run()
```

Experimental Studies

“Running time”/”execution time”: **time.time()** method

```
# The "timeit" module lets you measure the execution
# time of small bits of Python code

>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))',
                  number=10000)

0.3412662749997253

>>> timeit.timeit('"-".join([str(n) for n in range(100)])',
                  number=10000)

0.2996307989997149

>>> timeit.timeit('"-".join(map(str, range(100)))',
                  number=10000)

0.24581470699922647
```

“Running time”/”execution time”: timeit method

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Same results between two runs?

Theoretical Analysis

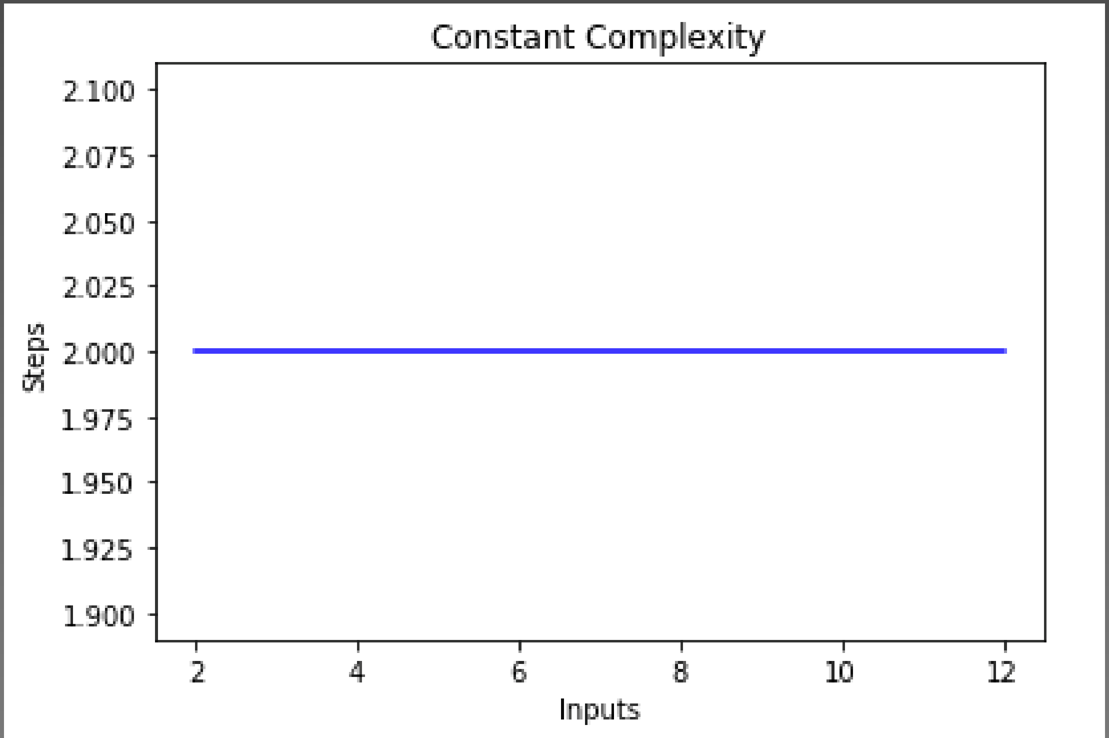
- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Have you heard of Big O Notation?

n	Constant O(1)	Logarithmic O(log n)	Linear O(n)	Linear Logarithmic O(n log n)	Quadractic $O(n^2)$	Cubic $O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824

Big O Notation: the order of magnitude for a useful approximation to the actual steps in the computation.

```
def constant_algo(items):
    result = items[0] * items[0]
    print ()

constant_algo([4, 5, 6, 8])
```

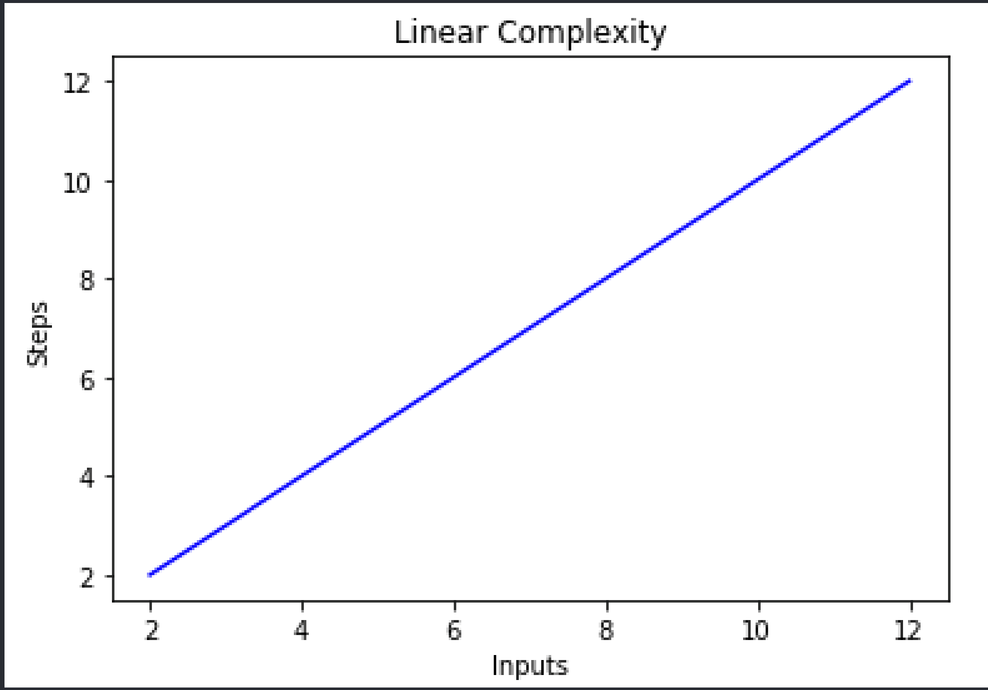


By Usman Malik

Constant Complexity (O(1))

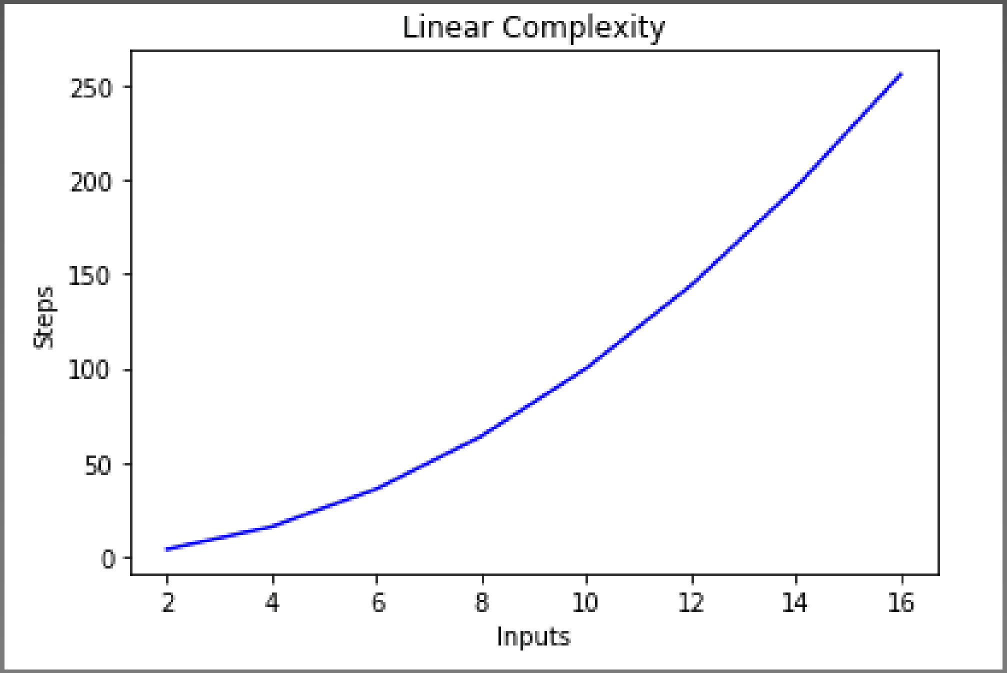
```python
def linear_algo(items):
    for item in items:
        print(item)

linear_algo([4, 5, 6, 8])
```

```python
import matplotlib.pyplot as plt
import numpy as np

x = [2, 4, 6, 8, 10, 12]

y = [4, 8, 12, 16, 20, 24]

plt.plot(x, y, 'b')
plt.xlabel('Inputs')
plt.ylabel('Steps')
plt.title('Linear Complexity')
plt.show()
```



Linear Complexity (O(n)) with plot code.

```
def quadratic_algo(items):
    for item in items:
        for item2 in items:
            print(item, ' ' ,item)

quadratic_algo([4, 5, 6, 8])
```



Quadratic Complexity (O(n^2))

```python
def search_algo(num, items):
    for item in items:
        if item == num:
            return True
        else:
            return False
nums = [2, 4, 6, 8, 10]

print(search_algo(2, nums))
```

Worst vs Best Case Complexity. What’s wrong with this function?
How about ”Lucky winner” in Lab 01?

```python
def return_squares(n):
    square_list = []
    for num in n:
        square_list.append(num * num)

    return square_list

nums = [2, 4, 6, 8, 10]
print(return_squares(nums))
```

Space Complexity

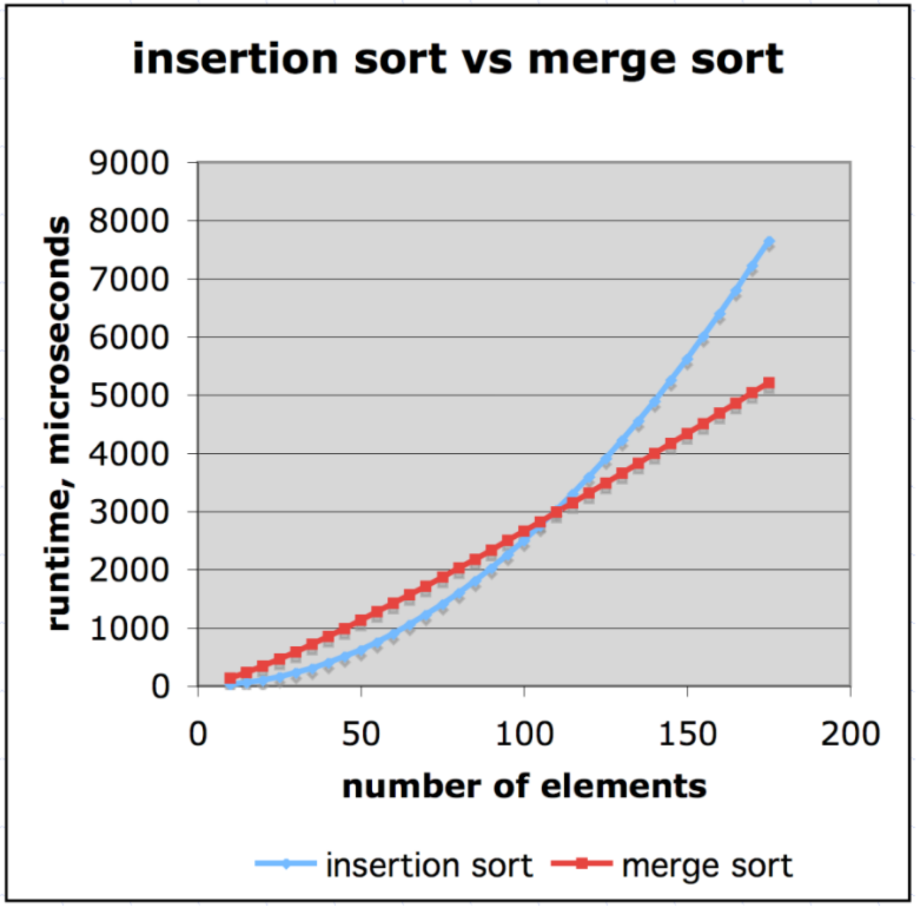
Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$

Big-Omega: the tight lower bound notation
Big-Theta: asymptotically tight bound (for a large n)

Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



insertion sort is

$n^2 / 4$

merge sort is

2 n lg n

sort a million items?

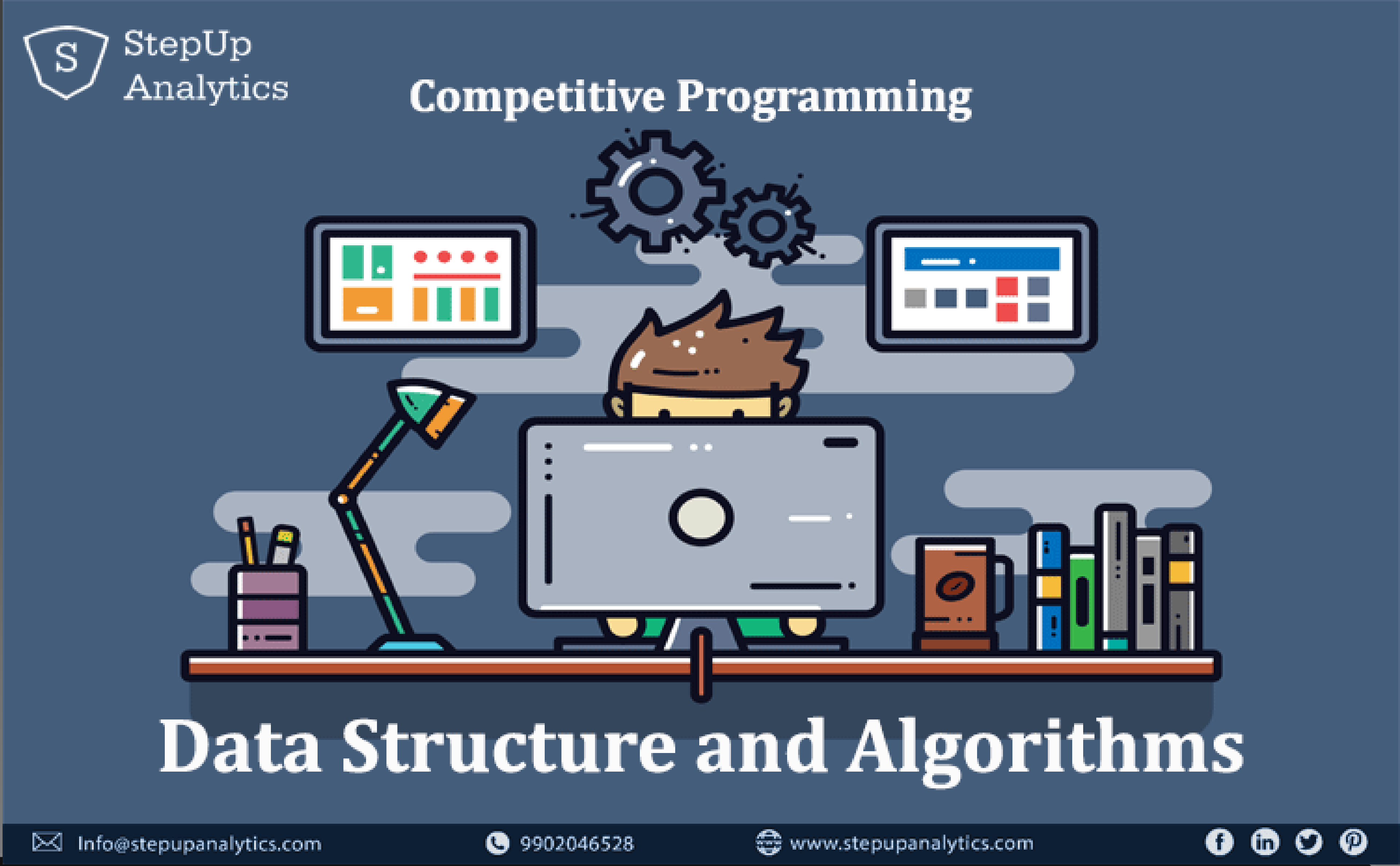
insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it’s 40 minutes versus less than 0.5 seconds

Which one is faster?



Basic Mathematics
Data Structures (Beginner to Expert)
Algorithms.

Reinforcement

Discussion Questions

Give the Big-O performance of the following code fragment

```
for i in range(n):
   for j in range(n):
      k = 2 + 2
```

```
i = n
while i > 0:
   k = 2 + 2
   i = i // 2
```

```
for i in range(n):
   for j in range(n):
      for k in range(n):
         k = 2 + 2
```

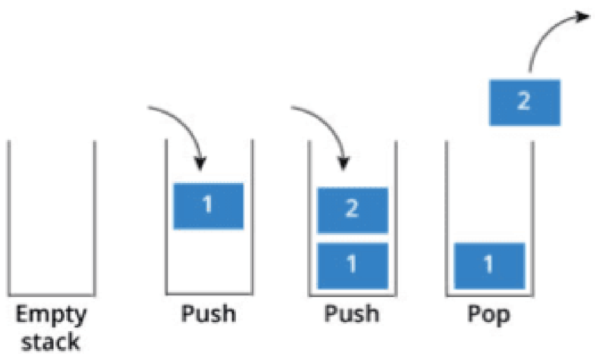
```
for i in range(n):
   k = 2 + 2
for j in range(n):
   k = 2 + 2
for k in range(n):
   k = 2 + 2
```

Quick overview

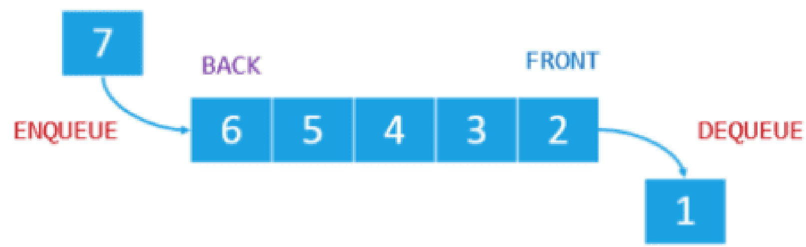
- Algorithm analysis is an implementation-independent way of measuring an algorithm.
- Big-O notation allows algorithms to be classified by their dominant process with respect to the size of the problem.

For next week!

Data Structure Basics



Stack



Queue