CSCI 120
Introduction to CompSci and Programming I
Lec 5: Algorithms II

Jetic Gū

Overview

- Focus: Python Programming
- Architecture: von Neumann
- Core Ideas:
    1. Time Complexity & Big-O Notation
    2. New lab to be released on Friday, with Tutorial videos to make up for yesterday

Time Complexity and Big-O

Summary

Time Complexity

- Method of algorithm analysis: how efficient is an algorithm?
- Time complexity: **estimation of amount of time** it takes to finish up an execution
- Why?
  - Different algorithms might lead to different complexity, and usually we want the most efficient algorithm
  - Time complexity analysis allows us to compare different algorithms scientifically

Search Algorithm (1)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

- An array contains n unique elements. Design an algorithm to search for the second largest number in an array.

- How can you solve it?

Example

Search Algorithm (1)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

- Solution 1:
  - Search for the largest number by going through the entire array.
  - Knowing the largest number, search again for the second largest.

Search Algorithm (1)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

```
# searching for the largest
lar = -1
for item in arr:
    if item > lar:
        lar = item
# lar is now the largest num
# this is how max(arr) works
```

```
# searching for the 2nd
lar2 = -1
for item in arr:
    if item > lar2:
        if item < lar:
            lar2 = item
return lar2
```

- How many steps does it take to execute this algorithm?

Example

Search Algorithm (1)

```
# searching for the largest
lar = -1
```

1 step

```
for item in arr:
    if item > lar:
        lar = item
```

an **steps**

```
# lar is now the largest num
# searching for the 2nd
lar2 = -1
```

1 step

```
for item in arr:
    if item > lar2:
        if item < lar:
            lar2 = item
```

bn **steps**

```
return lar2
```

- Assuming each comparison in the first `for` loop takes a steps
    - the first `for` loop in total takes an steps
- Assuming each comparison in the second `for` loop takes b steps
    - the second `for` loop in total takes bn steps
- a and b are constants

Example

Search Algorithm (1)

```
# searching for the largest
lar = -1                            1 step
for item in arr:
    if item > lar:
        lar = item                  an steps
# lar is now the largest num
# searching for the 2nd
lar2 = -1                           1 step
for item in arr:
    if item > lar2:
        if item < lar:
            lar2 = item             bn steps
return lar2
```

- In total:
  - $1 + an + 1 + bn = (a + b)n + 2$
  - In reality you will never be certain what these constants are, since different programming languages are different
  - We call algorithms that take $c_1 n + c_0$ time to be **linear**, and we say it's time complexity $O(n)$

Example

Search Algorithm (2)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

- An array contains n unique elements. Design an algorithm to search for the second largest number in an array.
- Consider this same problem, do we have other solutions? Can we look for the largest and 2nd largest at the same time?

Example

Search Algorithm (2)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

```
lar = -1  # largest
lar2 = -1 # second largest
for item in arr:
    if item > lar:
        lar2 = lar
        lar = item
    else if item > lar2:
        lar2 = item
```

every time a larger number is found,
`lar` **takes it, and** `lar2` **becomes the second largest**

- How many steps does it take to execute this algorithm?

Search Algorithm (2)

10	99	32	7	12	1	56	33	64	78	9	5	3	27

```
lar = -1   # largest
lar2 = -1 # second largest
```

2 step

```
for item in arr:
    if item > lar:
        lar2 = lar
        lar = item
```

cn **steps**

- In total:
  - $cn + 2$
  - This is also **linear**, also $O(n)$
  - is it faster or slower?

Example

P1
Complexity

Time Complexity Analysis

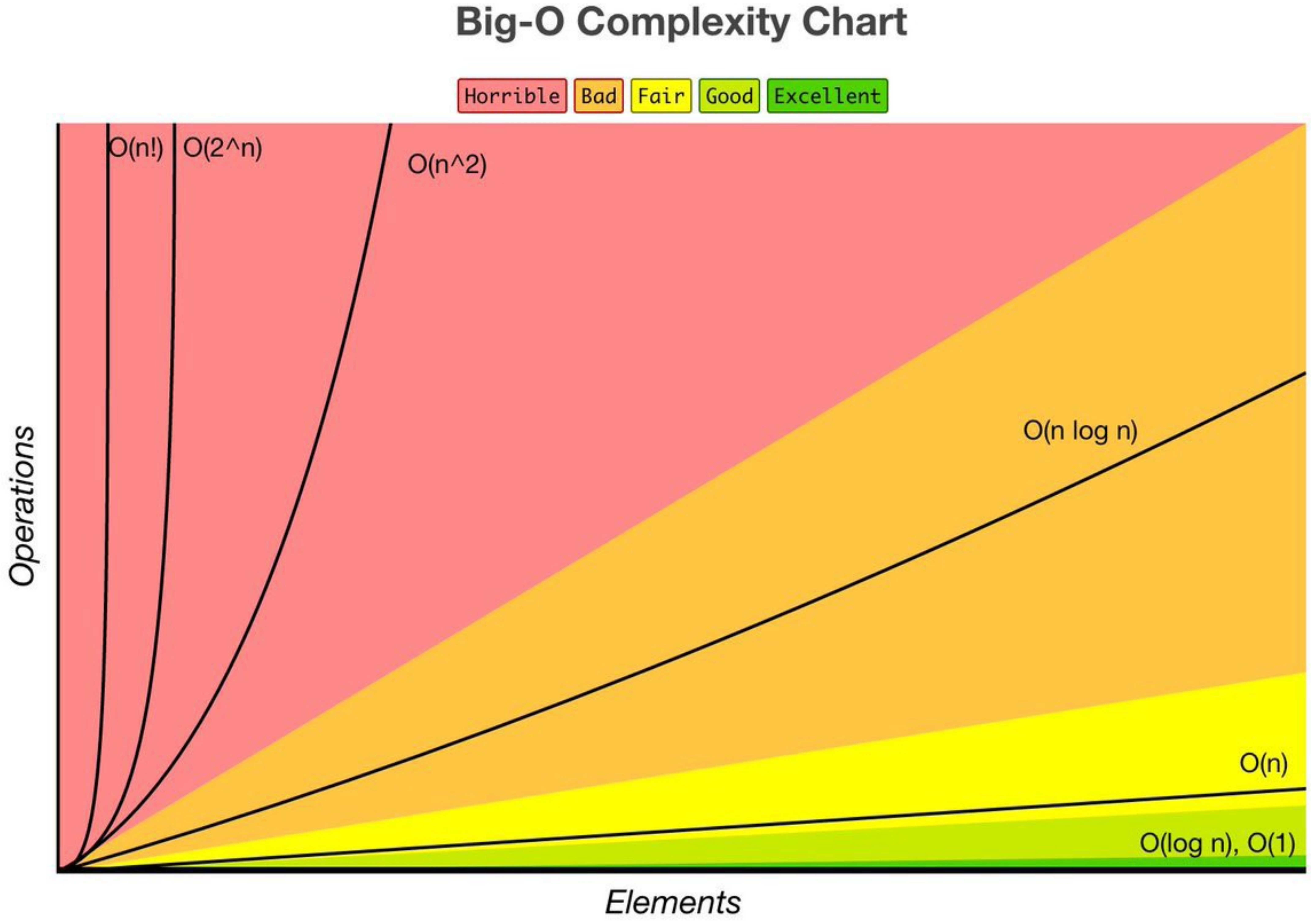
- There's a few principles in time complexity analysis of algorithms
  - we don't care about constants $ax + b = O(n)$
  - we only care about the element with highest power $ax^2 + bx + c = O(n^2)$
    - Why? Because an element with higher power will always out grow those with lower power. i.e. $O(2^n) > O(n^{50}) > O(n^2) > O(n \log n) > O(n) > O(\log n)$
  - This is called **Big-O Notation**.
    mathematical notation that describes the **limiting behaviour** of a function when the **argument tends towards a particular value or infinity**.

Constants are implementation details, not algorithm themselves

Example

P1
Complexity

Time Complexity Analysis



Chart

What is the complexity?

```
a = []
for i in range(n):
    a.append([])
    for j in range(n):
        a[i].append(int(input()))
```

- What is this programme doing?
- What is its complexity?

What is the complexity?

```
# a and b are matrices of n×n
c = []
for i in range(n):
    c.append([])
    for j in range(n):
        c[i].append(0)
        for k in range(n):
            c[i][j] += a[i][k] * b[k][j]
```

- What is this programme doing?
- What is its complexity?