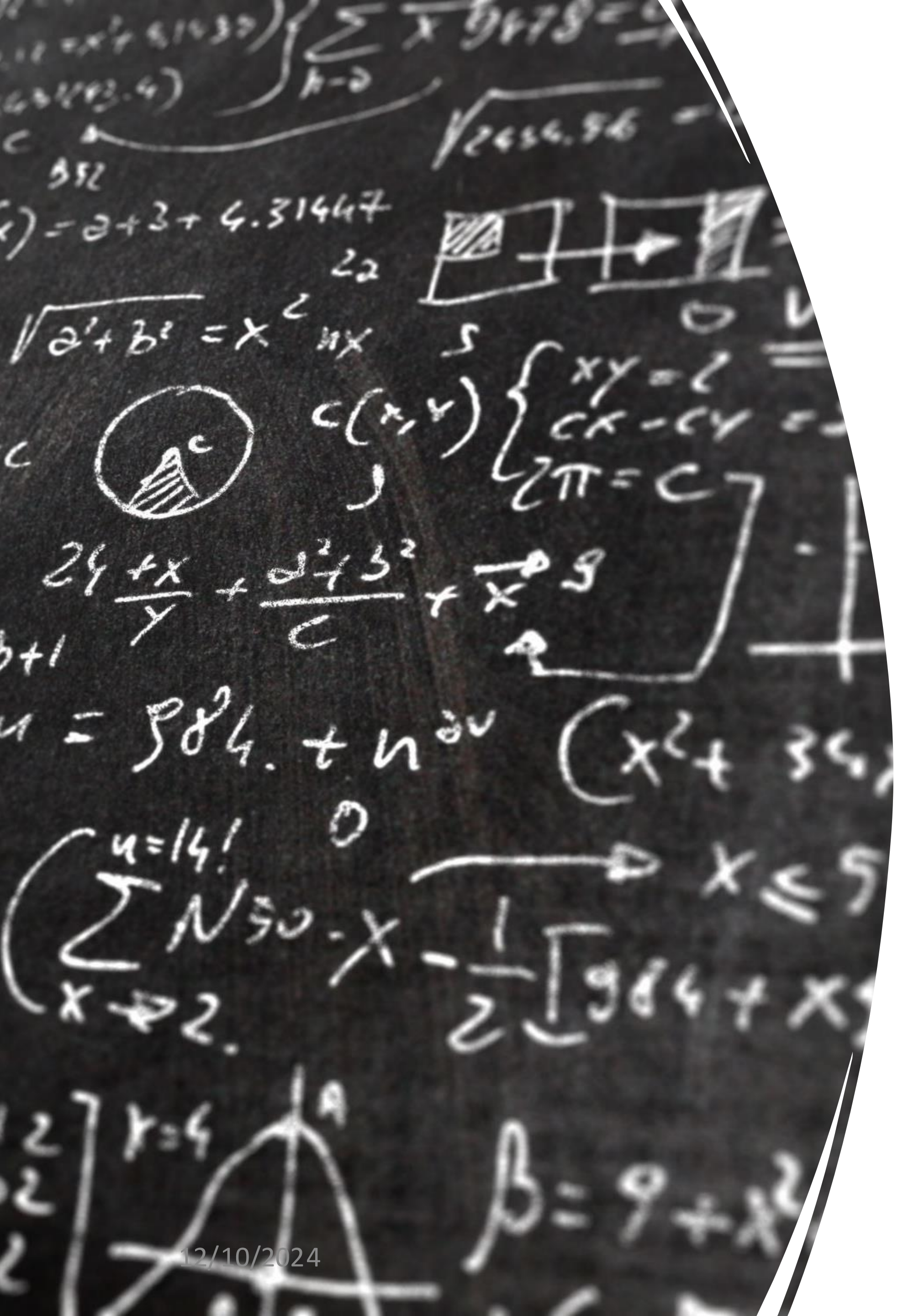


CS 113 – Computer Science I

Lecture 24 – Runtime Analysis

Tuesday 12/10/2024

Announcements

Mid-semester feedback survey

HW11 – due Thursday 12/12

Office hours Thursday: 2:45-4pm, and by appointment

Final: Wednesday 12/18 9:30am-12:30pm Park 238

Agenda

Midterm 2 Overview

Run time analysis

Midterm 2

	Denominator	Max	Median %	Mean %
Fall ‘24	92	89.4	86.84	80.28
Fall ‘23	77	75.9	69.48	67.17

Interfaces & Classes

Imagine <u>class</u> C **implements** <u>interface</u> A.

Is C a ***<u>subclass</u>*** of A?

No, because A is not a class, it is an interface

Instead, C is a ***<u>type of</u>*** A

LocationsOf

Write a method locationsOf that takes in a string and a character. The method should return a list of all locations where the character is located in the string.

LocationsOf

Approach 1:

initialize an empty array of indices: *locs*

Loop through the array

If item at index i == needle:

create a new *tmp* array of length *locs*.length + 1

copy over every element from *locs* to *tmp*

assign the value at last location of *tmp* to *i*

locs <- *tmp* // reassign *tmp* to *locs*

Steps to compute Big-O

How to compute Big O

1. Identify the input size: look at the number of data points (usually *n*)
2. Break down the algorithm:
    1. Analyze loops, nested loops, function calls
3. Calculate each component
    1. Count how many time operations are executed in terms of *n* or other components
4. Focus on dominant Terms
    1. Keep the fastest-grown term and ignore constants

Example

```
for (int i = 0; i < n; i++) {        O(n)
  for (int j = 0; j < d; j++) {      O(d)
    System.out.println(i, j);        O(1)
  }
}
```

Runtime:

O(n * d)

Common Patterns

Single loop through j items:

$$O(j)$$

Nested loop: outside loops f times and inner loops e times

$$O(f * e)$$

Nested loop: outside loops m times and inner loops m times

$$O(m * m) = O(m^2)$$

Divide and conquer through a list originally containing q items:

$$O(\log_2 q)$$

Example:

```
for (int i = 0; i < n; i++) {
  if (arr[i] == needle) {
    return true;
  }
}
```

Runtime:

Loop runs n times

Each operation inside of loop is $O(1)$

Total runtime: $O(n)$

Example: Matrix Multiplication

$$\begin{bmatrix} 1 & 2 \\ -6 & 5 \\ 3 & -4 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < k; j++) {
        for (int p = 0; p < d; p++) {
            result[i][j] += A[i][p] * B[p][j];
        }
    }
}
```

Runtime:

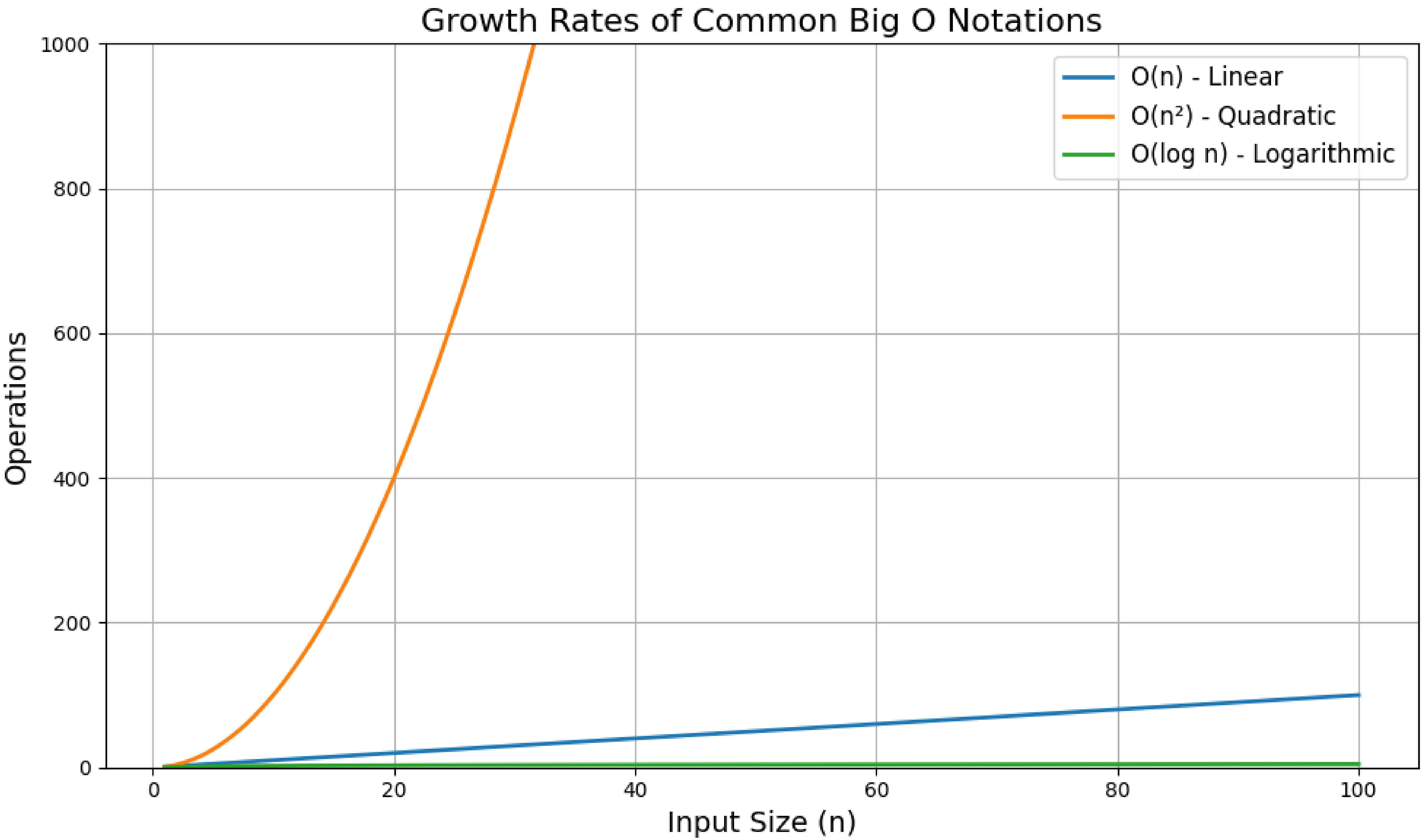
Outside loop runs n times

Middle loop runs k times

Inside loop runs d times

Each operation inside of loop is $O(1)$

Total runtime: $O(n * k * d)$



Growth Rates of Common Big O Notations
Operations
Input Size (n)
0
200
400
600
800
1000
0
20
40
60
80
100
O(n) - Linear
O(n²) - Quadratic
O(log n) - Logarithmic

Example:

```
for i from 1...n
  for j from 1...n
    print(i,j)
```

Runtime:

Outer loop runs n times

Inner loop runs n times

Each operation inside of loop is $O(1)$

Total runtime: $O(n * n) = O(n^2)$

Example:

```
for i from 1...n
  for j from 1...d
    if i == j
      return
```

Runtime:

Outer loop runs n times

Inner loop runs d times

Each operation inside of loop is $O(1)$

Total runtime: $O(n * d)$

But, algorithm will always stop after first check

Total runtime: $O(1)$

LocationsOf

Approach 1:

- initialize an empty array of indices: *locs*
- Loop through the array
- If item at index i == needle:
  - create a new *tmp* array of length *locs*.length + 1
  - copy over every element from *locs* to *tmp*
  - assign the value at last location of *tmp* to *i*
  - *locs* <- *tmp* // reassign *tmp* to *locs*

LocationsOf

Approach 2:

```
initialize idxs: an empty Boolean array that is the same size as the haystack array
initialize empty counter c
Loop through the haystack array
If item at index i == needle:
        idxs[i] = true
        c = c + 1

initialize a new array result of length c
pointer = 0
for i in 1… length of idxs:
  if idxs[i] == true:
        result[pointer] = i
        pointer += 1
```

Why care about Big-O

Why analyze runtimes?

- Predict how algorithms scale with larger inputs
- Compare performance of different algorithms
- Avoid inefficient solutions for real world problems
- Can compare algorithms before implementing them

Key Takeaways

- Big O helps measure algo efficiency
- Break algo into steps and count operations
- Focus on dominant terms (ignore constants)
- Practice analyzing real code examples to build intuition