CS151 Intro to Data Structures

Interfaces

Algorithm Analysis

Announcements

- HW02
  - Linked Lists
  - due Tuesday February 25th
  - start early!

Outline

- Objects
  - Comparing Objects
- Interfaces
- Algorithm Analysis

Java Memory Diagrams

Given a segment of code:

- Trace the program by going through line by line
  - Draw a memory diagram
  - As program variables are updated, update their values in the diagram

Example 2

Given a segment of code:

- Trace the program by going through line by line
  - Draw a memory diagram
  - As program variables are updated, update their values in the diagram

Interfaces

- An interface is <u>a contract </u>- A set of shared methods that users **must** implement
- create a program to calculate the area of different shapes, such as circles, rectangles, triangles etc.
- For each shape, you should be able to print the shape name and area
- Every time someone adds a new shape, they **must** include the methods for getName() and getArea()

Interfaces

- For any new shape that is created, we want to **enforce** that these methods are also implemented.

```
interface Shape {
    public double getArea();
    public String getName();
}
```

```
class Circle implements Shape {
```

Interfaces

<u>A contract </u>- A set of shared methods that users **must** implement

A collection of method signatures with no bodies

A class can implement more than one interface

Interfaces

An interface is not a class!

A class is what an object **is**

An interface is what an object **does**

- can not be instantiated
- no constructors
- incomplete methods

Interface

No modifier - implicitly `public`

No instance variables except for constants (`static final`)

Object Comparison

Object Equality

A custom class must define (override) its own `equals`

Object Comparison

- What if we wanted to compare two students by GPA?

```
int compareTo(T o)
```

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

`compareTo`

`compareTo` returns an `int`, not a `Boolean`

Why?

because it needs to convey three outcomes:

- `negative` if smaller compared to the parameter
- `0` if equal
- `positive` if larger compared to the parameter

`Comparable` interface

The `Comparable` interface is designed for objects that have an ordering

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

`Comparable` interface

When would we want to use this? **Let’s see in code :)**

Now, what if we wanted to sort from highest to lowest GPA

Custom Exceptions

Making Custom Exceptions

Often times we need to raise a custom exception

Extend `Exception` or `RuntimeException`

Custom Exceptions

What is the difference between extending from `Exception` rather than `RuntimeException`?

Subclass of `Exception` are checked exceptions – must be treated/caught

Subclass of `RuntimeException` are not checkable during compile time

Computational Complexity

Run Time Complexity

- Mathematical notation used to describe the performance or complexity of an algorithm.

- Hardware independent

- Represents the upper bound of the time complexity in the **worst-case scenario.**

- Helps us understand how the runtime of an algorithm grows ***as the input size increases.***

Big-O Example 1

```
int n = Integer.parseInt(args[0]);
int power = 1;
while (power < n) {
    System.out.print(power + " ");
    power *= 2;
}
```

How does the runtime grow as a function of the input size?

O(logn)

Big-O Example 2

```
int fetchFirstElement(int[] arr) {
    return arr[0];
}
```

How does the runtime grow as a function of the size of arr?

O(1)

Big-O Example 3

```
int n = Integer.parseInt(args[0]);
int tot = 0;
int i = 0;

while (i < n) {
   tot = tot * i;
   i++;

    for (int j=0; j<10000; j++) {
        System.out.println("hello");
    }
}
```

How does the runtime grow as a function of the input size?

Linearly!

O(n)

Big-O Example 4

```
int n = Integer.parseInt(args[0]);

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

How does the runtime grow as a function of the input size?

Quadratically!

O(n^2)

We do n operations n times

Big-O Example 5

```
String[] lst =
   {"19", "12", "20", "15"};

for (int i=0; i<100; i++)  {
   System.out.println(getNum(lst));
}

int getNum(int[] arr) {
   return Integer.parseInt(arr[0]);
}
```

How does the runtime grow as a function of the size of lst?

Constant! The runtime is not affected by the number of elements in `lst`

O(1)

Big-O Example 6

```
int[] lst = {1,2,3,4,5,6,7};

for (int i=0; i<lst.length; i++)  {
    findMax(lst);
}

int findMax(int[] arr) {
    int max = Integer.MIN_VALUE;
    for (int i=0; i<arr.length; i++) {
        if (arr[i] > max) {
            max =  arr[i];
        }
    }
    return max;
}
```

How does the runtime grow as a function of the size of lst?

O(n^2)

Space (Memory) Complexity

How much memory a program needs

The space requirements time typically grows with input size. Expressed as a size of the input. (Big O notation)

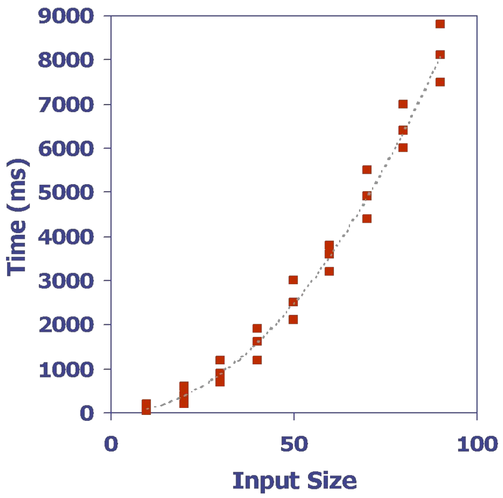
We focus on *worst case* analysis

- how much space will it take in the worst case?

Big O Notation and Theoretical Analysis

- Why do we express runtime notation with Big O notation? Why not just say the run time in number of seconds?

```
long startTime = System.currentTimeMillis();      // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();        // record the ending time
long elapsed = endTime − startTime;               // compute the elapsed time
```



- Answer: comparing two algorithms requires exact same hardware and software environments

Constant Time Operations

- Constant time operations require the same amount of time, regardless of the size of the input

- Examples:
  - Basic computations: Assigning variables, adding, multiplying, boolean operators
  - What were some constant time operations in ExapandableArray?
  - LinkedList?

Linear Time Algorithms: $O(n)$

- The runtime grows linearly as the size of the input grows
- Processes the input in a single pass spending constant time on each item
- Examples:
  - A single loop over an array
  - ExpandableArray?
  - LinkedList?

Example: Find Max

Worst case: 4n +1 ==> O(n)

Best case: 3n + 2 ==> O(n)

Quadratic Time: $O(n^2)$

Nested loops...

Example:
worst case: $4 + 3n^2$
best case: 7

$O(nlogn)$ time

Example: Binary Search!

find 0

-5	0	1	7	9	100	117	200

How many elements did we touch?

3 = log(8)

Where did the n come from?

$O(nlogn)$ time

Example: Binary Search!

Best case?

-5	0	1	7	9	100	117	200

Exponential Time: $O(2^n)$

- Generate all possible subsets

{a, b, c } = ...

How many subsets are there?

{∅}, {a}, {b}, {c}, {a,b}, {b,c}, {a,c}, {a,b,c}

8

2^3 = 8

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296

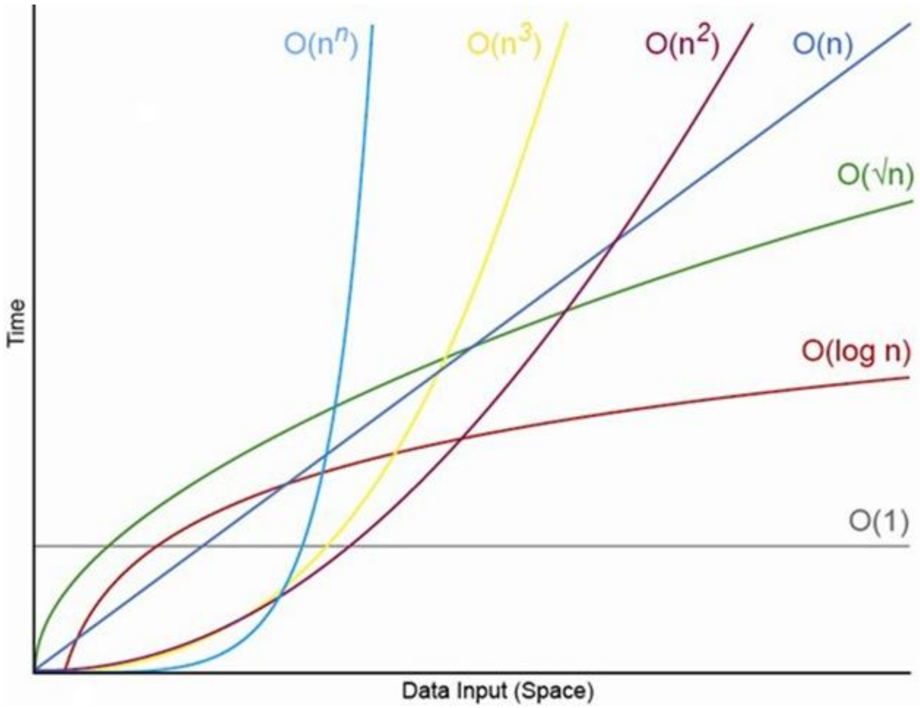
Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Asymptotic Notation



As the number of elements approaches infinity, only the dominant term matters

That is why we simplify O(n+1) to O(n) etc.

Big-O Analysis

1. Write a polynomial in terms of input size n
   - Only loops contribute
   - Each nested factor is multiplied
   - Each sequential factor is summed

2. Simplify the polynomial
   - Identify dominant term – highest degree polynomial
   - Polynomials beat polylogs
   - Exponentials beat polynomials
   - Discard constants

Summary

- Every non-primitive is an OBJECT in Java
- Every Object is a REFERENCE

- Interfaces are a *contract* of which methods you will implement

- You can compare objects with compareTo