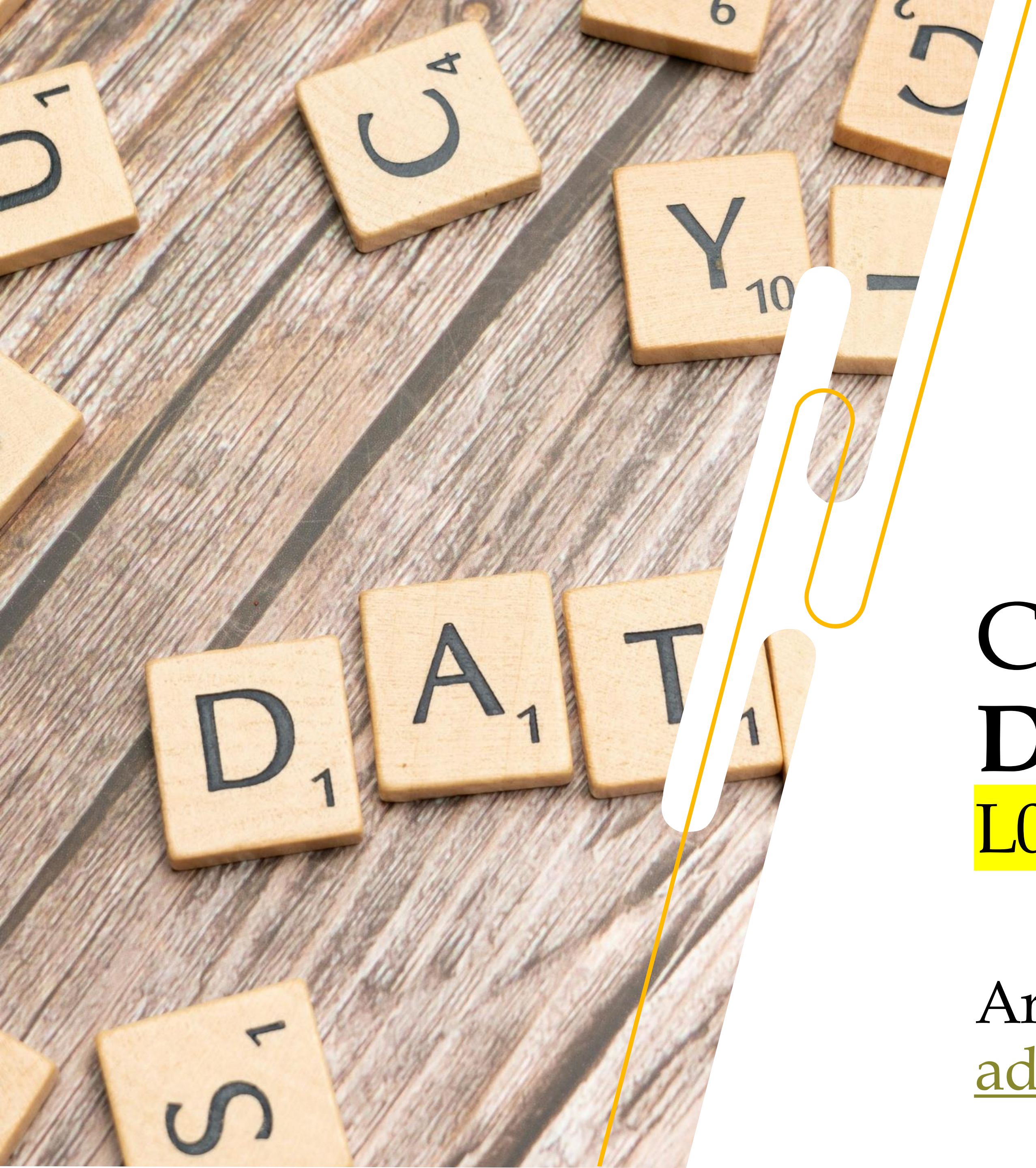


GEORGE MASON UNIVERSITY

CS 310 – Fall 2025
Data Structures
L03-Complexity, Dynamic Arrays

Archange G. Destiné
adestine@gmu.edu

Part of slides is from Dr. Russell and from Dr. Socrates

Outline for Today (Lecture 03)

1. **Recap from last Lecture (Generics)**
2. **Today:**
   - **Efficient Programming**
   - **Static Arrays**
   - **Linked List (intro… more on Lecture 05)**
   - **Dynamic Arrays**
3. **Notes:**
   - **Coding Warm-Up (DUE Sept 5 / MUST pass all tests)**
   - **Project 1 (Will be released later this week)**
     - **Individual Effort ! Reminder on Academic Standards**
   - **Participation Activities**

Academic Standards (from the syllabus)

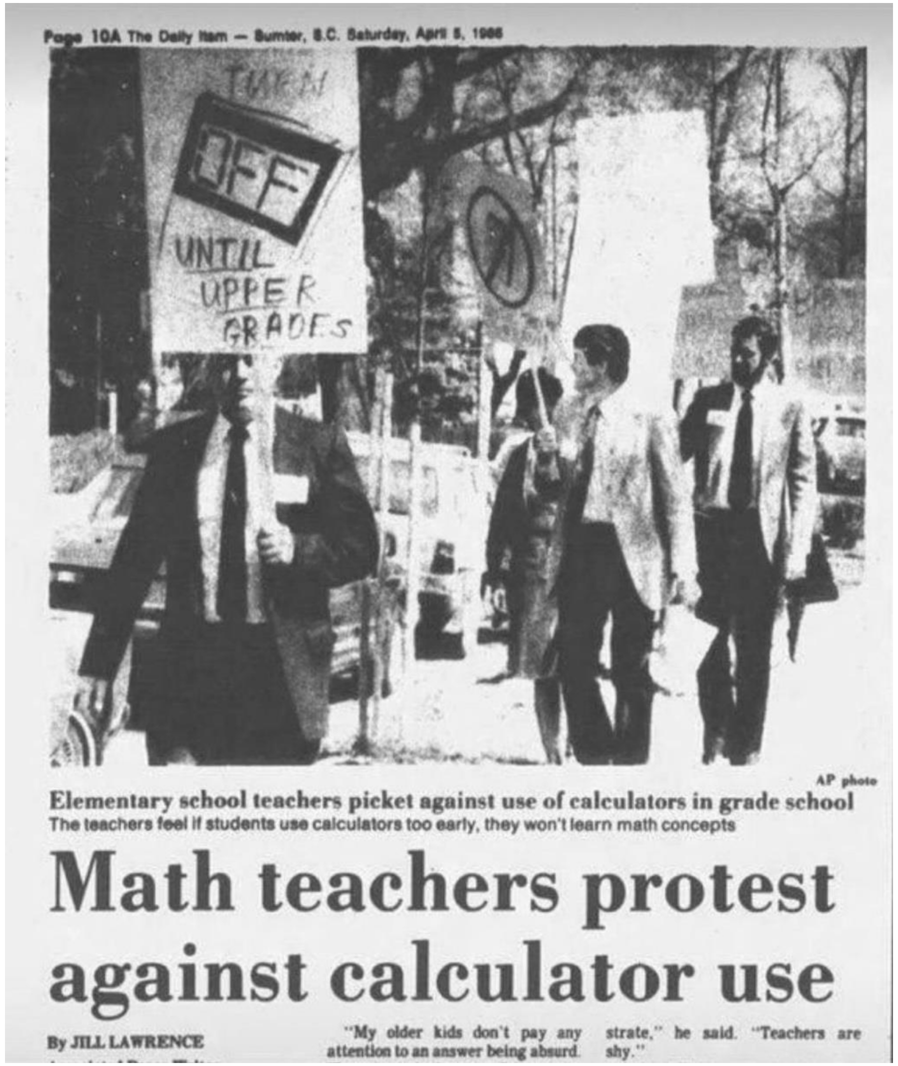
Programming projects are considered **individual efforts**, therefore no sharing of code and/or discussion of problem solutions are allowed with anyone except the TA or the professor.

Student projects will be manually and automatically assessed for cheating.
You may not use any Internet resources to create code or algorithms, besides the textbooks, the slides, and Piazza, unless otherwise specified.

However, you are free **to look up the syntax errors** your encounter online, to gain an understanding of what the syntax error means. The projects we're doing this semester **can be directly solved using techniques discussed in class**, and no outside material is needed unless otherwise noted.

It is your responsibility to **lock your computers** with a password, to not **post your code to websites** that are publicly accessible, to **guard your USB drives and computers**, to not upload your files to someone else's computer, etc. **You will be liable for any access gained to your code.**

Academic Standards



AP photo

Elementary school teachers picket against use of calculators in grade school

The teachers feel if students use calculators too early, they won't learn math concepts

Math teachers protest against calculator use

By JILL LAWRENCE

"My older kids don't pay any attention to an answer being absurd.

strate," he said. "Teachers are shy."

Need Help. How to request appointment?

Link:

Book time with Archange Giscard Destine

Choose a meeting type

General Appointment

15 MIN

This Booking is a 15-minute General Appointment with Professor Destine. For CS310 or CS330, choose the corresponding meeting type (not this one).

In-Person CS330 Appointments

30 MIN

The Booking is for CS 330 Appointment with Professor Destine. Please, make sure to bring in your filled-out advising forms.

CS310 Office Hours

15 MIN

No appointment is required if you plan to join on Monday from 12 to 2pm (Regular Office Hours). This is for CS 310 Office Hours appointment with Prof....

Recap - Java Basics - Quick Review

Object and References…
Illustration using whiteboard.

```
Car c = new Car();
```

What is **`c`**?
What happen in the memory when **`new Car()`** is executed?

What if we do **`Car d = c;`** ?

Generics (Quick review)

Problem 1: inefficient overloading

Ideally, we would like to create just **one** method that can dynamically accept any type **T**

```
public double print(T[] vector)
{
	for (T v : vector)
		System.out.print(v);
}
```

```
public double print(Float[] vector)
{
	for (Float v : vector)
		System.out.print(v);
}
```

don't try this code, it won't compile as is

```
public double print(Double[] vector)
{
	for (Double v : vector)
		System.out.print(v);
}
```

```
public double print(Byte[] vector)
{
	for (Byte v : vector)
		System.out.print(v);
}
```

Generics (Quick review)

Problem 2: "lost" types

This issue would always arise when we retrieve things from the ArrayList

It's even more annoying with the for-each loop:

```
ArrayList personList = new ArrayList();
// add many Person objects

//NOT ALLOWED:
for (Person p : personList)
        p.whatever();
```

Instead, we must use **downcasting** after we retrieve with **Object type**, like this:

```
// allowed, but annoying, harder to read, and error-prone
for (Object p : personList)
{
        ((Person) p).whatever();
}
```

Generics (Quick review)

Generics: Establish & Remember Types

- Generics allow us to define **type parameters** – we can parameterize blocks of code with types!

- Where can we add type parameters?
  - **at class declarations** → available for entire class definition
  - **at method signatures** → available throughout just this method

- instead of **just** having the regular parameter list where we supply values, we can **also** give a listing of type parameters, which can then show up as the types of our formal parameters

- Think of it as an extra level of overloading but more powerful and dynamic

Generics (Quick review)

Declaring Generic Classes

We can add a generic type to a class definition:

```
public class Foo <T>
{
      // T can be anywhere: like field types.
      public T someField;

      public Foo(T t)// T used as parameter type
      {
            this.someField = t;
      }

      // T used as return type and param type
      public T doStuff(T t, int x) { … }
}
```

Simply add the ***<T>*** or ***<E>*** or whatever name you like right after the class name, and then use ***T*** instead of a specific type in your code

Generics (Quick review)

Declaring Generic Methods

- In a generic method the **<T>** notation must go **before the return type**
- It may then be used as a type anywhere in the method: parameter types, local definitions' types… even the return type!
- All we know about **u1** or **u2** is that it is a value of the **U** type. That's not much info! But we can still write useful, highly re-usable code this way

```
public <T> void echo (T t1, T t2)
{
    System.out.print(t1 + t2);
}

public <U> U choose (U u1, U u2, boolean b)
{
    return (b ? u1 : u2);
}
```

Generics (Quick review)

Declaring both Generic Class and Generic Method

We can declare new generic types that are only visible with one method, like **<U>**, and have a different generic type for the class, like **<T>**

```
public class Foo <T>
{
    …
    public <U> void choose (U u1, U u2, boolean b, T var)
    {
        return (b ? u1 : u2);
    }
}
```

Generics (Quick review)

```java
public class Pair <R,S> {
    public R v1;
    public S v2;

    public Pair(R r, S s) {
        v1 = r;
        v2 = s;
    }

    public String toString() {
        return ("("+v1+","+v2+")");
    }
}

public class RunMe {
    public static void main (String[] args) {
        Pair<Integer, Double> a = new Pair<Integer, Double>(1, 2.0);
        Pair<Integer, Double> b = new Pair<>(3, 4.0);   // since Java 7
    }
}
```

Generics (Quick review)

Calling Generic Methods

Given a generic method (which happens to be static in this case):

```java
public class Foo {
    public static <U> U choose (U u1, U u2, boolean b) {
        return (b ? u1 : u2);
    }
}
```

We instantiate the parameters and can call it like this:

```java
    String s = Foo.<String>choose("yes","no",true);
    String t = Foo.choose("yes","no",true);
```

If it were non-static, we'd need an object to call it:

```java
    Foo f = new Foo();
    String s = f.<String>choose("yes","no",true);
    String t = f.choose("yes","no",true);
```

Since Java 7 we only need to specify the generic type when it's not clear from the parameters

Questions?

More on Generics
Consider those 2 problems…

Write a non-generic class that:

1. Has a generic method that returns a subarray of the first three items of a generic array

```
public <U> U[] subArray(U[] arr)
```

2. Has a generic method that returns the max value of a generic array

```
public <T> T maxValue(T[] arr)
```

Are those solutions correct ?

```
public class IssuesWithGenerics
{
  public <U> U[] subArray(U[] arr)
  {
    U[] subArray = new U[3];
    for(int i=0; i<subArray.length; i++)
        subArray[i] = arr[i];
    return subArray;
  }

  public <T> T maxValue(T[] arr)
  {
    T max = arr[0];
    for (int i = 1; i < arr.length; i++)
    {
      if(arr[i] > max)
      {
        max = arr[i];
      }
    }
    return max;
  }
}
```

Are those solutions correct ?

```
public class IssuesWithGenerics
{
  public <U> U[] subArray(U[] arr)
  {
    U[] subArray = new U[3];
    for(int i=0; i<subArray.length; i++)
        subArray[i] = arr[i];
    return subArray;
  }

  public <T> T maxValue(T[] arr)
  {
    T max = arr[0];
    for (int i = 1; i < arr.length; i++)
    {
      if(arr[i] > max)
      {
        max = arr[i];
      }
    }
    return max;
  }
}
```

ERROR

ERROR

Are those solutions correct ?

```java
public <U> U[] subArray(U[] arr)
{
  U[] subarray = new U[3];  // this instantiation is not allowed
  ...
}
```

```java
@SuppressWarnings("unchecked")  // to avoid compiler warnings
public <U> U[] subArray(U[] arr)
{
  U[] subarray = (U[]) new Object[3];    // downcasting
  // note that the cast type is U[] not just U
  ...
}
```

Are those solutions correct ?

```
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```

```
public <T> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

Are those solutions correct ?

```java
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```

```java
public <T> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

Unfortunately this doesn't work because compiler can't tell
if T has a **compareTo()** method

Are those solutions correct ?

```
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```

```
public <T extends Comparable<T>> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

The compiler now does know that T comes with a compareTo() method because T implements the Comparable interface

Bounded Type Parameters – Upper Bound

- Sometimes you want to restrict the types that can be used as type arguments
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses
- To declare an **upper bound** we use the **extends** keyword: `<T extends SomeType>`
- It means that T is a sub-class of SomeType or implements the SomeType interface

Bounded Type Parameters – Upper Bound

- We can even add **multiple extensions**:

```
<T extends A & B & C>
```

- The same way that classes/interfaces gave us subtypes, we're now saying "any class that's a subtype of SomeType". But we can make multiple claims at once this way

- In multiple extensions, **if one of the bounds is a class**, it must be specified first (i.e. before the interfaces)

Sometimes Upper bound is not enough

The following doesn’t compile. Can you see why?

```
class A {
     public static <T extends Comparable<T>> void someMethod(List<T> list) {

     }

     public static void main(String[] args) {
          ArrayList<Truck> al = new ArrayList<>();
          al.add(new Truck());
          al.add(new Truck());
          someMethod(al);
     }
}

class Vehicle implements Comparable<Vehicle> {
     public int compareTo(Vehicle v) {
          return 0;
     }
}

class Truck extends Vehicle {
}
```

We must replace `Comparable<T>` with `Comparable<? super T>`

Blank page

Bounded Type Parameters – Lower Bound

- We can use generics with a lower bound, indicating that it's acceptable for a type parameter to be any type that is a supertype of something particular. Example:

```
<? super PickupTruck>
```

  In this case, we can use any type that can accept PickupTruck values like PickupTruck, Truck, and Vehicle

- Look for instance at Collections.sort

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

  We could have limited ourselves to Comparator<T> that would only allow Comparator<PickupTruck> values to sort a list of PickupTruck objects. But if we also had a Comparator<Truck>, or a Comparator<Vehicle> it would make perfect sense to use them as another way to sort trucks or vehicles, which certainly includes PickupTrucks.

  By using the **super** keyword we can set a **lower bound** and accept all these different comparators when sorting a list of PickupTrucks.

Combining Lower Bounds and Upper Bounds

- We can also mix upper and lower bounds in the same declaration. Example:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

It means that the items in the list must be descendants of **`Comparable`** (otherwise it's impossible to compare them and then sort them), but the implementation of the Comparable interface itself (i.e. the implementation of the **`compareTo`** method) doesn't necessarily have to come from **`T`** directly, it can be inherited from its ancestor(s).

- How did we come up with the above declaration. Three attempts:

```
public static <T> void sort(List<T> list) // error
```

```
public static <T extends Comparable<T>> void sort(List<T> list) // ok but limited
```

```
public static <T extends Comparable<? super T>> void sort(List<T> list) // best
```

Questions?

Algorithm Analysis

Algorithm: Describe the **steps to solving** problems.

Algorithm Analysis: **Evaluate the efficiency** of your approach.

How to evaluate the efficiency?

Two types of complexities

Time-complexity

- How does the processing time increase with larger data sets?
- Poor performance: If sorting 10 names takes a second, but sorting 10,000 names takes several hours.

Space-complexity

- How does the memory usage increase with larger data sets?
- Poor example: If processing 10 images requires 5 megabytes, but processing 1,000 images needs 5 gigabytes.

Time Complexity

We will focus on **Time Complexity**.

Space Complexity analysis is similar.

Ex: How long dose it take to process **n** images?

How do we measure Time Complexity?

Time Complexity

	Figure 5.4	Figure 5.5	Figure 7.20	Figure 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.000001	0.000000	0.000001	0.000000
100	0.000288	0.000019	0.000014	0.000005
1,000	0.223111	0.001630	0.000154	0.000053
10,000	218	0.133064	0.001630	0.000533
100,000	NA	13.17	0.017467	0.005571
1,000,000	NA	NA	0.185363	0.056338

figure 5.10

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

Time Complexity

How do we measure Time Complexity?

- **Empirical** approach… Pro/Cons
- **Mathematical** approach… Pro/Cons
- **Approximation** analysis

Analysis Approaches: Empirical

– Run the program with different n
– Reasonable approach if no access to the code
– Can be used to predict performance

– Machine specific and lacks understanding

Analysis Approaches: Mathematical

- Can be used to predict performance
- Independent of machine (instructions->cycles->time)

- Very difficult to perform

Modeling Code Mathematically

- Represents complexity as a **function** of the input
  - **f(n)** where **n** is some input.
  - Example: for a sort, $f(n) = n^2$
    - where n is the **number of items to sort**
  - So… if you need to sort n = 5things,
    - it will take $\mathbf{n^2 = 5^2 = 25}$ **units** (of time/space)
  - Compare to **n=100** things, and it will take
    - $\mathbf{n^2 = 100^2 = 10{,}000}$ **units** (of time/space)
- Instead of using **f(n)**
  - When referring to time complexity, will use **T(n)**

Simple Problems

Let's look at a simple program and compute T(n)

```
void method1(int n, int m) {
        int sum = n + m;
        int diff = n -m;
}
```

```
void method2(int n, int m) {
        int total =0;
        for(int i=0; i<n; i++) {
                total += m;
        }
}
```

Simple Problems

T(n) for sequential loops?

```
void method3(int n, int m) {
        for(int i=0; i<n; i++) {
                System.out.println(i);
        }
        for(int i=0; i<n; i++) {
                System.out.println(i);
        }
}
```

T(n) for a one-factor nested loop?

```
void method4(int n, int m) {
        System.out.println(n+"x"+n);
        for(int i=0; i<n; i++) {
                for(int j =0; j <n; j++) {
                        System.out.println("banana");
                }
        }
}
```

Approximation Analysis

– Observation: For large n, **the lower order terms of T(n) are insignificant** and can be discarded

– idea is from calculus, as x tends to infinity

– (1) **Throw out low order terms** and (2) **approximate the time complexity** by considering upper/lower bounds.

– greatly simplify analysis

Approximation Analysis

– **Eliminates details to simplify model**
  – **e.g. Tilde(~), Big-Oh, Theta, Omega, etc.**
– **Independent of machine**
– **Can make statements concerning bounds**

– **Typically** cannot make predictions

Big-O Over-Simplified

Big-O(micron) simplified version!

– Define a function to describe your algorithm

  – $2n^2 + 27$

– Drop the low order terms (+27)

  – $2n^2$

– Drop the leading constants (2)

  - n^2

- Put an O next to it

  – $O(n^2)$

Big-O Identities

The **time complexity** of a **sequence of statements** in an algorithm or program is the **sum of the statements' individual complexities**.

$O(k\ g(n)) = O(g(n))$ - for a constant k

$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$

$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$

$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n)))$

$O(\max(g_1(n), g_2(n), \ldots, g_m(n)) =$

$\max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$

Categories of Functions

Name	Leading Term	Big-Oh	Example
Constant	$1, 5, c$	$O(1)$	$2.5, 85, 2c$
Log-Log	$\log(\log(n))$	$O(\log\log n)$	$10 + (\log\log n + 5)$
Log	$\log(n)$	$O(\log(n))$	$5\log n + 2$
			$\log(n^2)$
Linear	n	$O(n)$	$2.4n + 10$
			$10n + \log(n)$
N-log-N	$n\log n$	$O(n\log n)$	$3.5n\log n + 10n + 8$
Super-linear	$n^{1.x}$	$O(n^{1.x})$	$2n^{1.2} + 3n\log n - n + 2$
Quadratic	n^2	$O(n^2)$	$0.5n^2 + 7n + 4$
			$n^2 + n\log n$
Cubic	n^3	$O(n^3)$	$0.1n^3 + 8n^{1.5} + \log(n)$
Exponential	a^n	$O(2^n)$	$8(2^n) - n + 2$
		$O(10^n)$	$100n^{500} + 2 + 10^n$
Factorial	$n!$	$O(n!)$	$0.25n! + 10n^{100} + 2n^2$

Quick Rules of Thumb for Industry

– **O(1)** -usually doing something that takes a fixed amount of time, no matter how long that time is
– **O(log n)** -dividing a problem in half repeatedly and working on only one half each time
– **O(n)** -doing something with each item of data (or a faction of the data, like n/2)
– **O(n log n)** -dividing a problem in half repeatedly and working on both halves each time

– **O(n^2)** -nested loops that both go through all data
– **O(n^3)** -three nested loops that each go through all data
– **O([anything more than n^x])**-you're usually doing it wrong

Blank page

Approximation Analysis

Big-O (Omicron) formal version!

T(n) is O(F(n)) if there are positive constants **c** and **n_0** such that when **n** >= **n_0**

T(n) <= c.F(n)

– Let's show that:

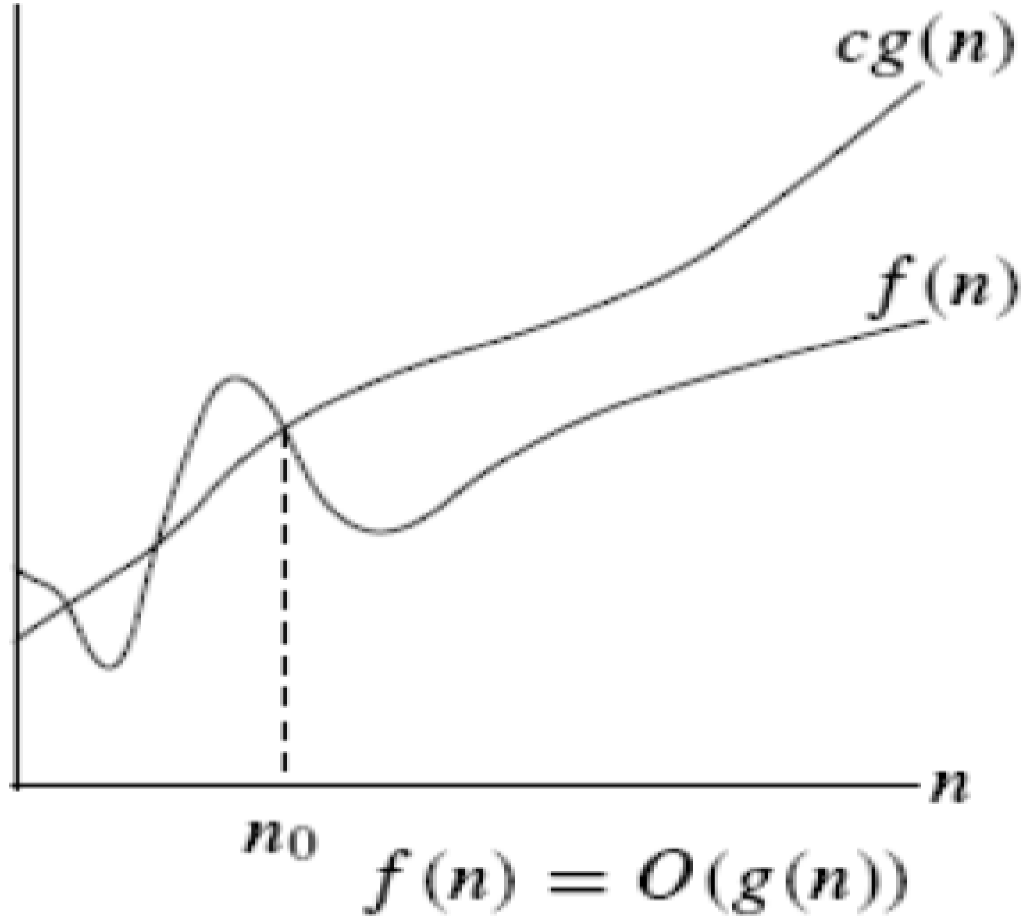
2n+1 is O(n)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \in [0, \infty)$$

Approximation Analysis

f(n) is O(g(n)) if there are positive constants **c** and **n_0** such that when **n** >= **n_0**

$f(n) <= c.g(n)$



Other Notations

Big O(Omicron): **Upper bound**, “big-oh”

– T(n) is O(F(n)) if there are positive constants c and n_0 such that

– when n >= n_0

– T(n) <= cF(n) Ex: $n^2 - 2n + 5 = O(n^3)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in [0, \infty)$$

Big Ω(Omega): **Lower bound**

– T(n) is Ω(F(n)) if there are positive constants c and n_0 such that

– when n >= n_0

– T(n) >= cF(n) Ex: $n^2 + 1 = \Omega(n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty]$$

Big Θ (Theta): **Upper and lower bound**

– If something is O(F(n)) and Ω(F(n)) it is Θ(F(n))

Ex: $n^2 + 3n + 4 = \Theta(n^2)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty)$$

Little o(Omicron): T(n) grows **much slower** than F(n)

Little ω(Omega): T(n) grows **much faster** than F(n)

Questions?

Static List vs Dynamic List

What is a **List**?

Collection of items that have a position.

Set vs Sequence

Static List vs Dynamic List

ADT vs Data Structure

Static Array (White Board)

Static List

Dynamic List (ADT)

- Invariants

- How do those invariants dictate the algorithms?

Dynamic List Implementation (Data Structures)

Dynamic List:

- Array-based
- Pointer-based (next week)

Dynamic Array Lists

Some operations I might perform on a list:

- **set** a value (at some index)
- **get** a value (at some index)
- **insert** a value (to the end)
- **insert** a value (at some index)
- **remove** a value (at some index)
- **search** for a value (and return the index)

We can have clear specifications (ADT/API) answering those questions.
Now what implementations, algorithms (Data Structures) solve these problems.

Basic Outline…

set()

- parameter: int **index**, **Object** to put there
- return: ??

get()

- parameter: int **index**
- return: **Object** at index

append()

- parameter: **Object** to add
- return: ??

Basic Outline…

insert()

- parameter: int **index**, **Object** to put there
- return: ??
- Other types of add…. parameters: Object?

remove()

- parameter: int **index**
- return: ??
- Other types of remove… parameter: Object?

search()

- parameter: **Object** to find
- return: int index found (or **-1** for not found)

Your Turn! Big-Oh

What is the Big-O of the following operations:

- **set** a value (at some index) -- ?
- **get** a value (at some index) -- ?
- **append** a value (to the end) -- ?
- **add/insert** a value (at some index) -- ?
- **remove** a value (at some index) -- ?
- **search** for a value (and return the index) -- ?

Hint: is adding/removing different from the front/middle/end?

- Think about best/worst case?

Blank

Data Structure Operations Big-O

- Limitations of these structures?

Operation Implementation	Get set	Add Remove end	Insert Remove begin	Insert Remove middle	search	Grow?
Array	1	-	-	-	-	no
List (Static Array)	1	1	n	n	n	No

Dynamic Arrays

Java: ArrayList

Not enough space?

- Increase the capacity (not size !!!)
- copy things over to the new (larger) array

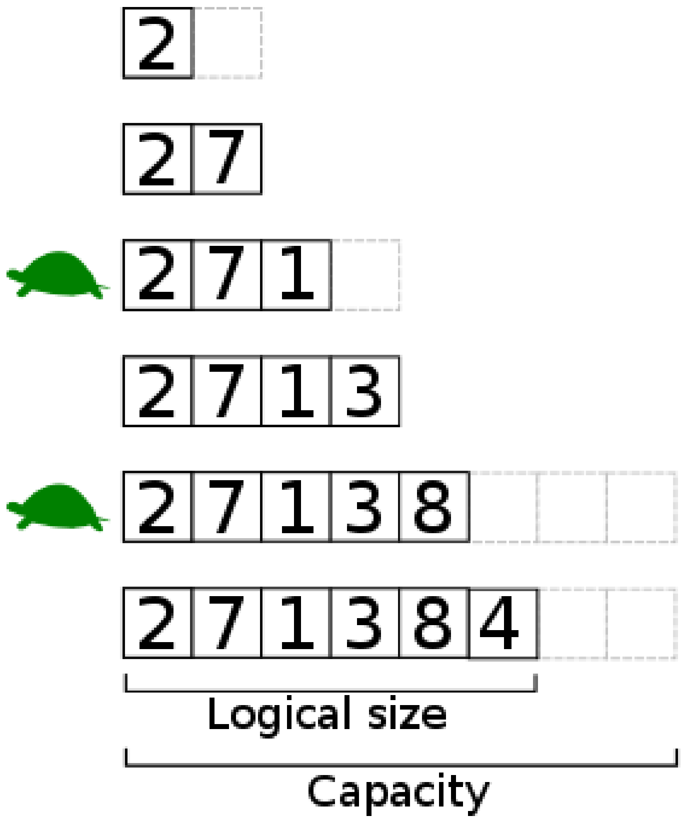


Image Sources: http://en.wikipedia.org/wiki/File:Dynamic_array.svg

Questions?

Data Structure Operations Big-O

–Later This Semester: We'll talk about the "n?" using a new analysis technique!

Operation Implementation	Get set	Add Remove end	Insert Remove begin	Insert Remove middle	search	Grow?
Array	1	-	-	-	-	no
List (Static Array)	1	1	n	n	n	no
Dynamic Array	1	n?	n	n	n	Yes

Example of implementation… and code analysis.

ArrayList implementation

`remove(index)`, O(n), why?

What about `remove_first()`?

Algorithm remove(i):

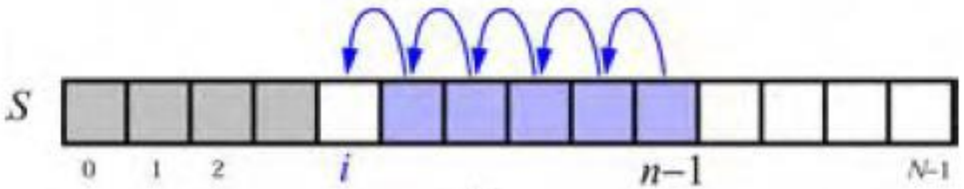
$e \leftarrow A[i]$ {e is a temporary variable}

for $j = i, i+1, \ldots, n-2$ **do**

$A[j] \leftarrow A[j+1]$ {fill in for the removed element}

$n \leftarrow n - 1$

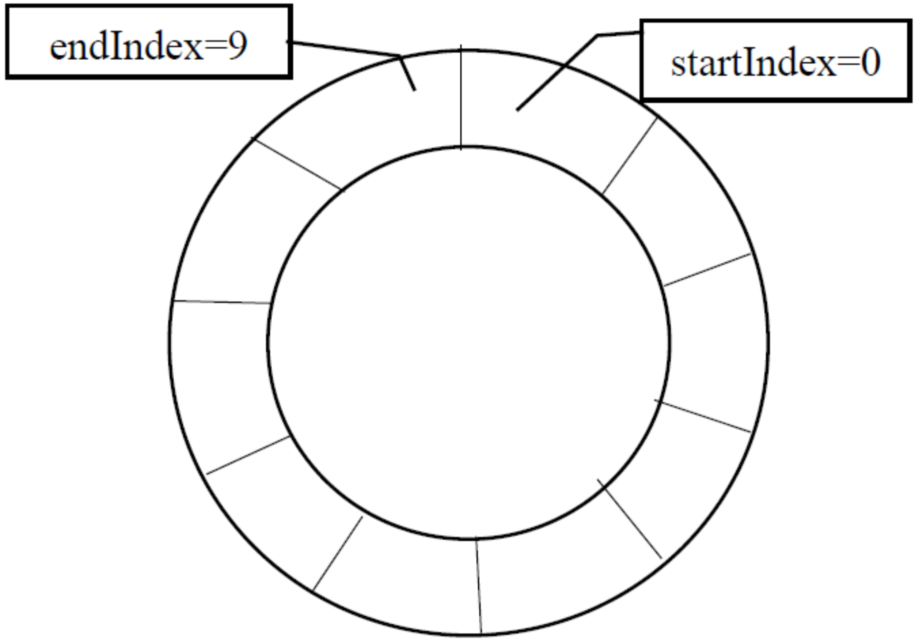
return e

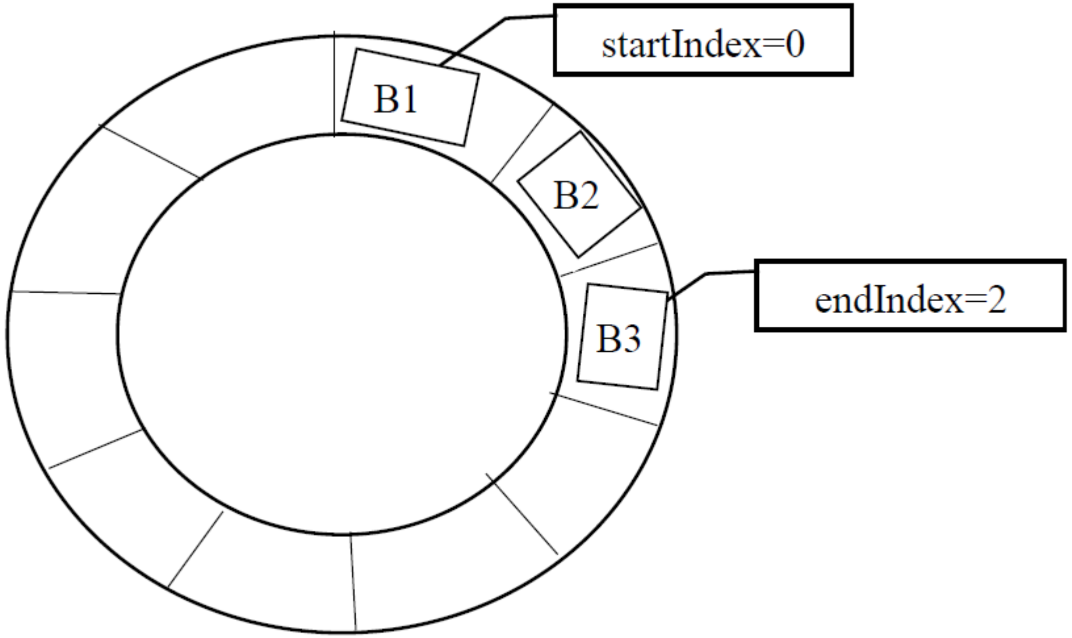


Dynamic Arrays… -> Circular Array

A Special implementation

A data structure using an array as if it is connected end-to-end…
Think about Search function in a text document.
What is the time complexity of remove_first()?





Circular Array (Motivation)

Operation Implementation	Get set	Add Remove end	Insert Remove begin	Insert Remove middle	search	Grow?
Array	1	-	-	-	-	no
List (Static Array)	1	1	n	n	n	no
Dynamic Array	1	n?	n	n	n	Yes
Circular Array	?	?	?	?	?	?

In what situation would a circular array be **better than a Dynamic Array**?

Questions?

Next Lecture (Lecture 04)

1. More on Dynamic Arrays (Recap)
2. Iterator
3. Linked List

Reminders:

You must pass all tests on Coding Warmup.

Project_1 will be released by Friday

Make sure you have access to Piazza