Contents

Introduction

Data structures is the study of how to organize information in a computer so as to ensure efficiency. Note that I am not taking this class purely of my own volition, so I will be much more sarcastic in these notes than even the PDEs notes.

Reintroduction to Java

Everyone here has learned how to write code in Java,[1] so we're going to go over a quick review of everything we learned in Java.

The variable types are as follows:

- `String`: text, like `"Hello"`;
- `int`: integers, like `123`;
- `double`: floating point numbers, like `19.99`;
- `char`: characters, like `'a'`;
- `boolean`: stores the states `true` or `false`.

Hello World

```
public class Main{
  public static void main(String[] args) {
    System.out.println("Hello, World.");
  }
}
```

Note that unlike Python, we need to specify the data type of each variable. For instance,

Values to Variables

```
String message;
message = "Hello, World";

int value1;
value1 = 15;

double value2;
value2 = 24.8;
```

To obtain values from user inputs, we need to use the `Scanner` library.

[1] Well, "learned" is a strong word.

User Input

```java
import java.util.Scanner;
public class Main{
  public static void main(String[] args){
    Scanner input = new Scanner(System.in);

    System.out.println("Integer:");
    int a=input.nextInt();

    System.out.println("Double:");
    double b = input.nextDouble();

    System.out.println("Text:");
    input.nextLine(); //Need this, else will return blank line
    String c=input.nextLine();
  }
}
```

We can also include if/else statements.

Using If/Else Statements

```java
  public class Main{
    public static void main(String[] args){
      int a=10;
      int b=2;
      if(a > b){
        System.out.println("a is greater than b");
      } else if (a < b){
        System.out.println("b is greater than a");
      } else {
        System.out.println("a and b are equal");
      }
    }
  }
```

The loop syntaxes[II] are as follows:

While Loop

```java
    int a=0;
    int b=0;
    int c=5;

    while (a < c){
      b = b + 10;
      a = a + 1;
    }
```

For Loop

```java
  int a;
  int b=0;
  int c = 5;
  for(a=0; a < c; a= a + 1){
    b = b+10;
  }
```

The next most important structure we use a lot is the Array/Array List.

Arrays and Array List

```java
import java.util.*;
public class PlayingWithArrays{
  public static void main(String[] args){
    List<Integer> a = new ArrayList<>();
```

[II] Syntices?

```
    a.add(10);
    a.add(11);
    a.add(12);
    System.out.println(a);
    a.set(0,20);
    System.out.println(a.get(0));

    int[] b = {30,31,32};
    System.out.println(b[0]);
    System.out.println(Arrays.toString(b));
  }
}
```

Note that array lists are data structures, as well as arrays.

Java also admits functions (but, in classic Java fashion, they are called methods).

Functions and Methods

```
public class Main{
  public static double area (int base, int height){
    double result;
    result = base * height/2;
    return result;
  }
}
```

Java is an object-oriented language, so there are all the fun parts of OOP, like classes, instances, etc.

Classes and Instances

```
class Professor{
  String first_name;
  String last_name;
  String email_address;
  String office_location;
} // A class.
public class Main{
  public static void main(String[] args){
    // Instances
    Professor the_reader = new Professor();
    Professor not_the_reader = new Professor();
  }
}
```

The two most important types of methods in Java are the getter and the setter.

Getting and Setting

```
class Movie{
  private String title;
  public void setTitle(String title){
    this.title = title; //use of this keyword tells us that we want to change the title that is
        part of the class Movie rather than the title that is our argument.
  } // our title setter
  public String getTitle(){
    return this.title; //Similarly, this returns the value of our previously set title.
  }
}
public class Main{
  public static void main(String[] args){
    Movie our_american_programming_language = new Movie();
    our_american_programming_language.setTitle("Our American Programming Language"); //Use setter
         to set title
    System.out.println(our_american_programming_language.getTitle()); //Use getter to access
        title (without changing it)
  }
}
```

Data Structures

We use data structures to achieve efficiency, in the sense that it uses the least time, fewest instructions, and least memory usage. When evaluating an algorithm, we need to understand all of these methods.

Consider a guessing game where one participant chooses a number and the other participant tries to guess. There are two ways to guess:

- guessing directly (i.e., just guessing in increasing order);
- guessing via binary search.

The latter approach is significantly more efficient than the former.

Lab 1: Playing with Arrays

Finding Maximum Value in an Array

```java
import java.util.*;
public class FindingMaximum {
  public static void main(String[] args) {
    int[] main_array = new int[6];
    Scanner input = new Scanner(System.in);
    for(int i = 0; i < main_array.length; i++){
      System.out.print("Please provide an integer to go into the array: ");
      main_array[i] = input.nextInt();
      System.out.println();

    }
    int currentMax = main_array[0];
    for(int i = 0; i < main_array.length; i++){
      if(main_array[i] > currentMax){
        currentMax = main_array[i];
      }
      System.out.println("Current maximum value: " + currentMax);
    }
    System.out.println("The maximum value of the array is "+ currentMax);
  }
}
```

Finding Maximum Value in an Array

```java
import java.util.*;
public class FindingMinimum {
  public static void main(String[] args) {
    int[] main_array = new int[6];
    Scanner input = new Scanner(System.in);
    for(int i = 0; i < main_array.length; i++){
      System.out.print("Please provide an integer to go into the array: ");
      main_array[i] = input.nextInt();
      System.out.println();

    }
    int currentMax = main_array[0];
    for(int i = 0; i < main_array.length; i++){
      if(main_array[i] < currentMax){
        currentMax = main_array[i];
      }
      System.out.println("Current minimum value: " + currentMax);
    }
    System.out.println("The minimum value of the array is "+ currentMax);
  }
}
```

Finding the Maximum n Values in an Array

```
import java.util.*;
public class MaximumValues{
  public static void main(String[] args){
    // want to find the n maximum values of an array.
    int[] numbers = new int[6];
    Scanner input = new Scanner(System.in);
    for(int i = 0; i < numbers.length; i++){
      System.out.print("Please provide an integer to go into the array (from highest to lowest):
          ");
      numbers[i] = input.nextInt();
      System.out.println();
    }
    System.out.print("Please provide how many maximum values you wish to retrieve: ");
    int inputLength = input.nextInt();
    System.out.println();
    System.out.println(outputValues(numbers,inputLength));
  }
  public static String outputValues(int[] inputArray, int inputIndex){
    String outputString = "The maximum ";
    if(inputIndex <= 1){
      outputString = outputString + " value of the array is " + inputArray[0];
      return outputString;
    } else{
      int arrayIndex = inputIndex;
      if(inputIndex > inputArray.length){
        arrayIndex = inputArray.length;
      }
      outputString = outputString + arrayIndex + " values of the array are ";
      String valueString = "";
      for(int i = 0; i < arrayIndex; i++){
        valueString = valueString + inputArray[i] +", ";
      }
      valueString = valueString.substring(0,valueString.length()-2);
      outputString = outputString + valueString;
      return outputString;
    }
  }
}
```

Finding the Minimum n Values in an Array

```
import java.util.*;
public class MinimumValues{
  public static void main(String[] args){
    // want to find the n maximum values of an array.
    int[] numbers = new int[6];
    Scanner input = new Scanner(System.in);
    for(int i = 0; i < numbers.length; i++){
      System.out.print("Please provide an integer to go into the array (from lowest to highest):
          ");
      numbers[i] = input.nextInt();
      System.out.println();
    }
    System.out.print("Please provide how many minimum values you wish to retrieve: ");
    int inputLength = input.nextInt();
    System.out.println();
    System.out.println(outputValues(numbers,inputLength));
  }
  public static String outputValues(int[] inputArray, int inputIndex){
    String outputString = "The minimum ";
    if(inputIndex <= 1){
      outputString = outputString + " value of the array is " + inputArray[0];
      return outputString;
    } else{
      int arrayIndex = inputIndex;
      if(inputIndex > inputArray.length){
```

```
25          arrayIndex = inputArray.length;
26        }
27        outputString = outputString + arrayIndex + " values of the array are ";
28        String valueString = "";
29        for(int i = 0; i < arrayIndex; i++){
30          valueString = valueString + inputArray[i] +", ";
31        }
32        valueString = valueString.substring(0,valueString.length()-2);
33        outputString = outputString + valueString;
34        return outputString;
35      }
36    }
37  }
```

In Java, we use the `final` keyword to render it impossible to change something or the other.

- A final variable cannot have its value changed.
- A final method cannot be overriden by any child methods.
- A final class cannot be extended or inherited.

Finally, one more thing is that if we do not care about indices, and just want to access all the elements of an array, we may use the following code snippet.

Accessing Array

```
1     int[] numbers = {1,2,3,4,5,6,7,8};
2     for(int n : numbers){
3       System.out.println(n);
4     }
```

Abstract Data Types and Complexity

When we discuss abstract data types, we are focused on higher-level programming, rather than the low-level understanding of how the data type works. Typically, if we have an array with data, we need to find

Abstract Data Type	Description	Underlying Primitive Data Structure(s)
list	ordered data	Array, linked list
dynamic array	ordered data with indexed access	Array
stack	items are only inserted or removed at the top of the stack	linked list, array
queue	items are inserted at the end of the queue and removed from the front of the queue	linked list, array
deque	stands for "double ended queue," items can be inserted and removed from the front and back	linked list, array
bag	order does not matter, duplicate items are allowed	array, linked list
set	order does not matter, distinct items	binary search tree, hash table
priority queue	queue where each item has a designated priority, those with higher priority are closer to front of the queue	heap
dictionary	associates keys with values	hash table, binary search tree

Table 1: Abstract Data Types

where in the array a specific value is being held. Consider the following array.

Searching An Array

```
1     int[] num = {2,4,7,10,11,32,45,87};
```

If we want to search for an entered key, we can go about this by two fashions. The first is linear search.

Linear Search

```
    int[] num = {2,4,7,10,11,32,45,87};
    int key; // asked for by system
    // insert necessary lines here
    for(int i = 0; i < num.length; i++){
      if(num[i] == key){
        System.out.println("We have found the key value of " + key + " at index " + i);
        break;
      }
    }
```

Writing a Linear Search

```

import java.util.*;
public class LinearSearch{
  public static void main(String[] args){
    int[] numbers = {2, 4, 7, 10, 11, 32, 45, 87};
    System.out.println("The array is {2, 4, 7, 10, 11, 32, 45, 87}");
    Scanner input = new Scanner(System.in);
    System.out.print("Please specify a search key: ");
    int key = input.nextInt();
    System.out.println();
    int targetIndex = -1;
    for(int i = 0; i < numbers.length; i++){
      System.out.println("Searching for key value " + key + " at index " + i + "...");
      if(numbers[i] == key){
        targetIndex = i;
        break;
      }
    }
    if(targetIndex == -1){
      System.out.println("Key value not found");
    } else{
      System.out.println("Found key value " + key + " at index " + targetIndex);
    }
  }
}
```

The binary search shown here was improved based on troubleshooting by ChatGPT. The full transcript can be found here.

Binary Search

```
import java.util.*;
public class BinarySearch{
  public static void main(String[] args){
    int[] numbers = {2, 4, 7, 10, 11, 32, 45, 87};
    System.out.println("The array is {2, 4, 7, 10, 11, 32, 45, 87}");
    Scanner input = new Scanner(System.in);
    int upperIndex = numbers.length - 1;
    int lowerIndex = 0;

    System.out.print("Please specify a search key: ");
    int key = input.nextInt();
    System.out.println();
    boolean notFound = true;

    while(notFound && (upperIndex >= lowerIndex)){ //need greater than or equal to
      int searchIndex = (upperIndex + lowerIndex)/2; // initialize search index to be average of
          lower and upper values
      System.out.println("Searching for key value " + key + " at index " + searchIndex + "...");
      if(key == numbers[searchIndex]){
        System.out.println("Found key value " + key + " at index " + searchIndex);
        notFound = false;
```

```
      } else if (key > numbers[searchIndex]){
        lowerIndex = searchIndex + 1;
      } else if (key < numbers[searchIndex]){
        upperIndex = searchIndex - 1;
      }
      // Assistance of ChatGPT was used to troubleshoot errors in previous version of code
    }

    if(notFound == true){
      System.out.println("Key value not found.");
    }
    /* while(notFound && notAtEdgeIndex){
    *   if(searchIndex == numbers.length-1 || searchIndex == 0){
    *     notAtEdgeIndex = false;
    *   }
    *   System.out.println("Searching for key value " + key + " at index " + searchIndex + "...")
        ;
    *   if(key == numbers[searchIndex]){
    *     notFound = false;
    *   } else if (key > numbers[searchIndex]){
    *     searchIndex = boundaryIndex - (boundaryIndex - searchIndex)/2;
    *   } else if (key < numbers[searchIndex]){
    *     boundaryIndex = boundaryIndex / 2;
    *     searchIndex = searchIndex / 2;
    *   }
    */ }
    // The above snippet was the first attempt.
  }
}
```

Sorting Algorithms

A computer that wishes to sort an array must access individual values of the array and compare two values, swapping values until the array is sorted.

Selection Sort

```
    import java.util.*;
    public class SelectionSort{
      public static void main(String[] args){
        // initialize array and stuff with boilerplate here
        for(int i = 0; i< numbers.length; i++){
          //start by choosing your current index
          smallIndex = i;
          //our loop here finds the smallest index
          for(int j = i + 1; j < numbers.length; i++){
            if(numbers[j] < numbers[smallIndex]){
              smallIndex = j;
            }
          }
          //now, swap with the current index
          int temp = numbers[i];
          numbers[i]= numbers[smallIndex];
          numbers[smallIndex] = temp;
        }
      }
    }
```

One of the issues with selection sort is that we *always* run the comparison and the swapping, regardless of if our array is currently sorted, scrambled, or almost sorted.

This is not an issue with insertion sort.

Insertion Sort

```
import java.util.*;
public class InsertionSort{
  public static void main(String[] args){
    //initialize array and stuff with boiilerplate here
    for(int i = 1; i < numbers.length; i++){ //start with current index with i =1 and "go down"
         rather than starting from current index and selecting smallest from everything above
      int j = i; //start from current index
      while(j > 0 && numbers[j] < numbers[j-1]){ //move down swapping our array until we hit a
          lower limit

        // we swap downward if we're satisfied
        int temp = numbers[j];
        numbers[j] = numbers[j-1];
        numbers[j-1] = temp;
        j--; //move down in index and check again
      }
    }
  }
}
```

Insertion sort swaps downward only if we need to do so; it performs the same amount of comparisons, but it performs way fewer swaps.

It may be shocking, then, to see that both selection sort and insertion sort have the same time complexity of $O(n^2)$. However, the advantage of insertion sort over selection sort is that it works well with nearly sorted data much better than selection sort does.

Now, we want to understand more efficient sorting algorithms. Specifically, we will understand quicksort.

The quicksort begins by a partitioning algorithm that "sorts" our array by selecting a pivot and moving elements to one side or the other of the pivot.

Partition Algorithm

```
int partition(int[] numbers, int lowIndex, int highIndex) {
 // Pick middle element as the pivot
 int midpoint = lowIndex + (highIndex - lowIndex) / 2;
 int pivot = numbers[midpoint];

 boolean done = false;
 while (!done) {
    while (numbers[lowIndex] < pivot) {
       lowIndex++;
    }
    while (pivot < numbers[highIndex]) {
       highIndex--;
    }

    // If lowIndex and highIndex have met or crossed each
    // other, then partitioning is done
    if (lowIndex >= highIndex) {
       done = true;
    }
    else {
       // Swap numbers[lowIndex] and numbers[highIndex]
       int temp = numbers[lowIndex];
       numbers[lowIndex] = numbers[highIndex];
       numbers[highIndex] = temp;

       // Update lowIndex and highIndex
       lowIndex++;
       highIndex--;
    }
 }
 return highIndex; // highIndex is weirdly named here; it refers to the upper bound of the
     lower partition
```

```
32  }
```

After we implement our partition algorithm, our algorithm recursively sorts each sub-array using the partition method until we finish.

Quicksort Algorithm

```
1   void quicksort(int[] numbers, int lowIndex, int highIndex) {
2    if (highIndex <= lowIndex) {
3       return;
4    }
5
6    int lowEndIndex = partition(numbers, lowIndex, highIndex); //lowEndIndex denotes the upper
         bound of the lower partition
7    quicksort(numbers, lowIndex, lowEndIndex);//now, we partition each partite set in our array
8    quicksort(numbers, lowEndIndex + 1, highIndex);
9   }
```

Note that worst case run time for quicksort is $O(n^2)$, but this only occurs if every partition only consists of one element.