CS 24 AP Computer Science A Review

Week 7: Recursion and Algorithm Analysis

DR. ERIC CHOU
IEEE SENIOR MEMBER

SECTION 1

Problem Solving

Developing Algorithms – Problem Solving

Problem Solving

- Sorting
- Approximate Methods (ad hoc solutions)
- Randomized Solutions
- Dynamic Programming
- Linear Programming
- Cryptography
- Fourier Transform
- Computational Geometry
- Graph Theory

Algorithm Efficiency Analysis (Time Complexity)

Big-O Notation

NP-Completeness

Data Structure (Memory Complexity/Time Complexity Trade-offs)

Binary

B-Tree and other Tree

Stack, Queue, List, Map, Set, Heap, Sparse Matrix

Design Tools

- English – Like Pseudo Code (Using Java-Like pseudo code instead)
- Drawings, Visualization.
- Developing procedures (Flow-Chart)
- Developing data structure and classes (UML)
- GUI development – Design Patterns (Non-AP)
- Unit Testing (JUnit) (Non-AP)

Written Essay in Exams

(1) No computer allowed.

(2) Test on the programming ideas.

(3) Time limited.

(4) Design-centric. (will not be data processing centric or GUI-centric)

Purpose of this Lecture

A Collection of Algorithms and Patterns for Exams

This lecture works as a program collection.
It may grow over the time. (I will keep expanding).

When a certain big exam comes, you may always come there fore the latest version.

Memorize Algorithms and Design Patterns

Study a few Java Program patterns:
(1) Swap (Ch. 7)
(2) Shuffle (Ch. 7)
(3) Finding max/min (Ch. 7)
(4) Sum/average (Ch. 7)
(5) Sort (Ch. 15)
(6) 2-D array for shortest distance among a group of points. (Ch. 8)

Memorize Algorithms and Design Patterns

(7) 2-D traversal. (Ch.8)
(8) conversion of 1-D Array to 2-D, 2-D array to 1-D (Ch.8)
(9) split and merging of arrays (Ch. 15, Ch. 7)
(10) insertion, deletion, replacement of array and arraylist items. (Ch. 7, 8)
(11) copy a block in 2-D array (Ch. 8, Lab 2)
(12) String matching and operations (Ch. 3, Ch. 6, 7)
(13) recursive calls. (Ch. 14)

SECTION 2

Complexity Analysis

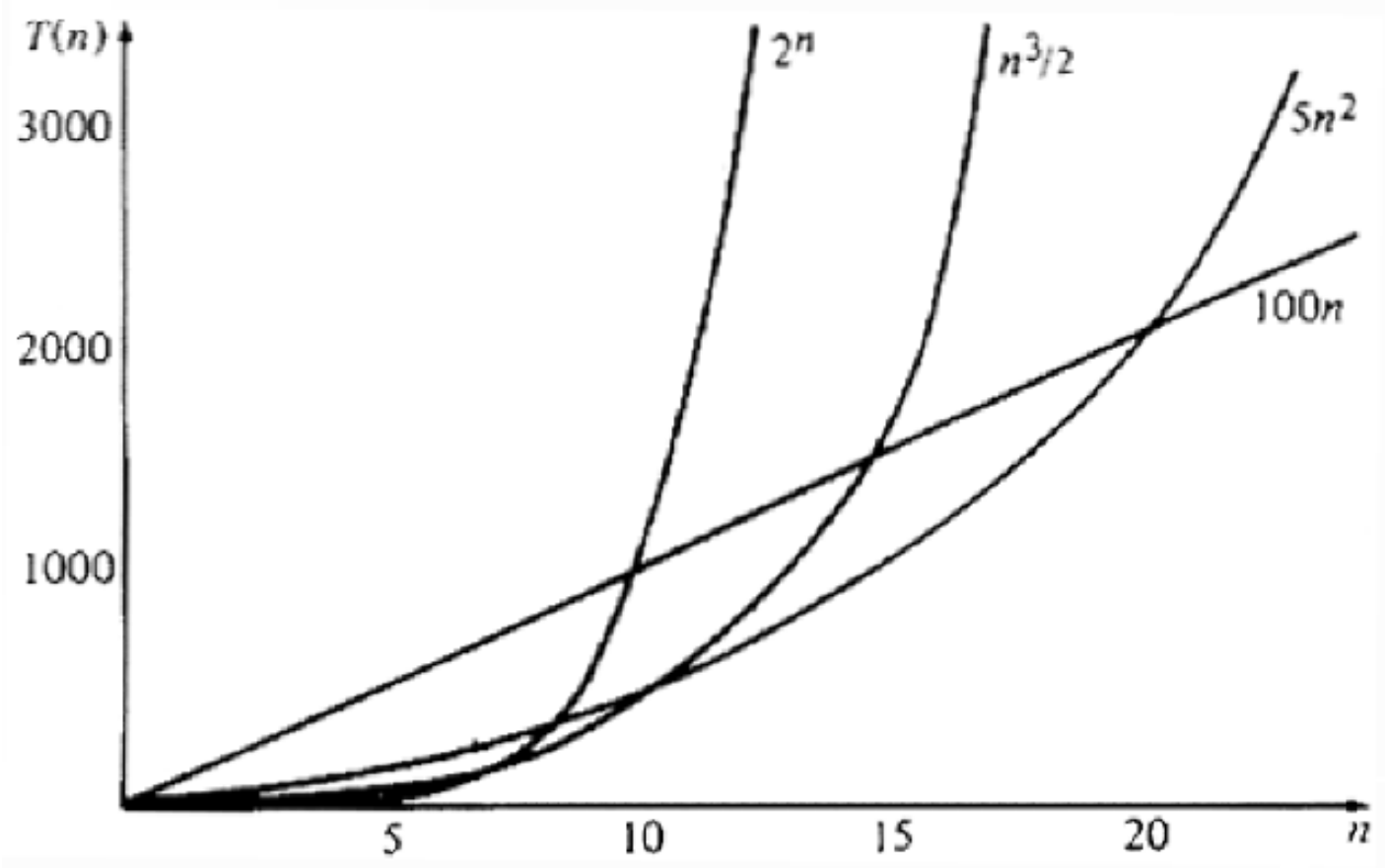
Commonly asked Questions

- Number of Recursive Calls
- The result of Recursion

Big O Notation

- Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires *n* comparisons for an array of size *n*. If the key is in the array, **it requires *n/2* comparisons on average**.
- The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of *n*. Computer scientists use the Big *O* notation to abbreviate for "order of magnitude." Using this notation, the complexity of the linear search algorithm is ***O(n)***, pronounced as "***order of n***."

Approximate Growth Rate *T(n)*



Best, Worst, and Average Cases

- For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.
- **An average-case analysis attempts to determine the average amount of time among all possible input of the same size.** Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

Ignoring Multiplicative Constants

- The linear search algorithm requires n comparisons in the worst-case and **n/2** comparisons in the average-case. Using the Big O notation, both cases require **O(n)** time. The multiplicative constant **(1/2)** can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.
- The growth rate for n/2 or 100n is the same as n, i.e., **O(n) = O(n/2) = O(100n).**

Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes n-1 times of comparisons to find maximum number in a list of n elements.
- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As n grows larger, the n part in the expression n-1 dominates the complexity.
- The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression n-1) and highlight the important part (e.g., n in the expression n-1). So, the complexity of this algorithm is O(n).

Constant Time

- The Big **O(n)** notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation **O(1)**.
- For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

SECTION 3

Recursion

Steps to write a Recursive Program

1. The Termination Condition (Base Case)
2. Recursive Formula (Recursive Call)
3. Step by Step Improvement

Sum of 1 to n

```
public class SumForward
{
    public static int Sum(int n){
        if (n == 1) return 1;

    }
}
```

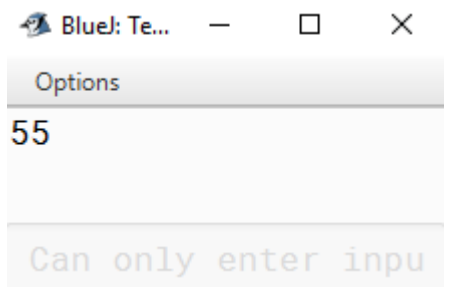
(1) Base Case

```
public class SumForward
{
    public static int Sum(int n){
        if (n == 1) return 1;    // base case
        return Sum(n-1) + n ;    // recursive formula
    }
}
```

(2) Recursive Formula

```
1  public class SumForward
2  {
3      public static int Sum(int n){
4          if (n == 1) return 1;    // base case
5          return Sum(n-1) + n ;    // recursive formula
6      }
7      public static void main(String[] args){
8         System.out.println(Sum(10));
9      }
10 }
```

(3) Test Setup



Sum of 1 to n

Helper Method with Accumulator (For tail-recursive methods)

```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
     }
}
```

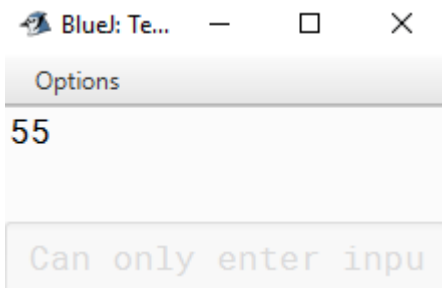
(1) Base Case

```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
     }
    public static void main(String[] args){
        System.out.println(Sum(10, 0));
     }
}
```

(2) Recursive Formula

```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
     }
    public static int Sum(int n){ return Sum(n, 0); }
    public static void main(String[] args){
        System.out.println(Sum(10));
     }
}
```

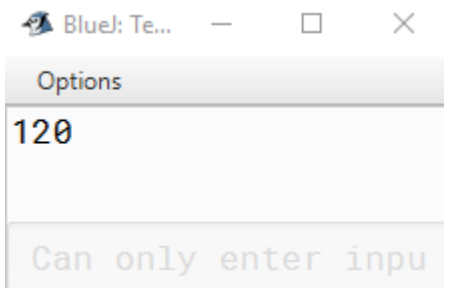
(3) Test Case with Normal Method



Factorial

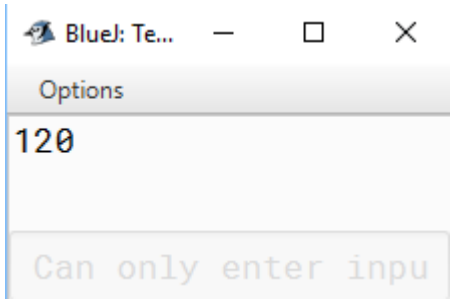
```java
public class Factorial{
    public static int f(int n){
      if (n==0) return 1;
      return f(n-1)*n;
    }
    public static void main(String[] args){
        System.out.println(f(5));
    }
}
```

(1) Normal Version




```java
public class FactorialHelper{
    public static int f_helper(int n, int product){
      if (n==0) return product;
      return f_helper(n-1, product * n);
    }
    public static int f(int n){ return f_helper(5, 1);}
    public static void main(String[] args){
        System.out.println(f(5));
    }
}
```

(2) Tail-recursive Version



Conversion of Sum to Iterative

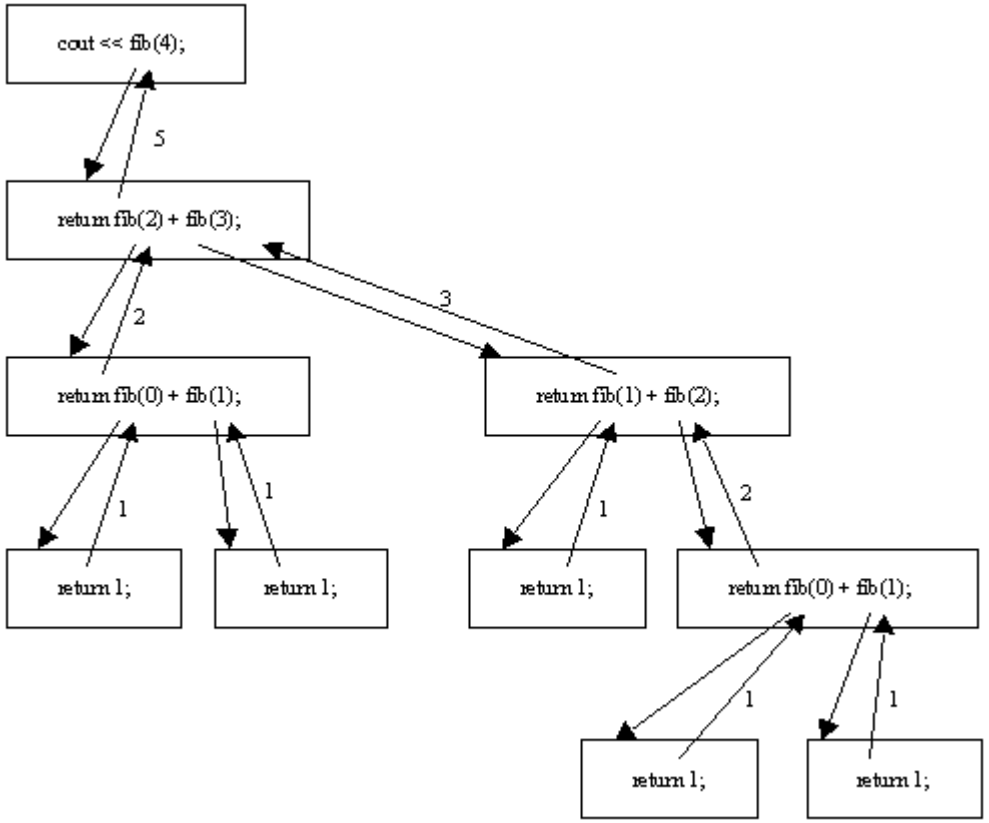
```
public static int Sum(int n){
    if (n == 1) return 1;   // base case
    return Sum(n-1) + n ;   // recursive formula
}
```

Base condition

Recursive Formula

```
public static int Sum(int n){
  int s = 0;
  for (int i=0; i<=n; i++){
    s = s + i;
  }
  return s;
}
```

Tracing Recursive Functions



Recursive Versus Iterative

```
static int fibonacciI(int n){
 int[] a = new int[n+1];
 a[0] = 1;
 a[1] = 1;
 for (int i = 2; i<=n; i++){
    a[i] = a[i-1] + a[i-2];
  }
 return a[n];
} // runs faster
static int fibonacciR(int n){
if (n == 0 || n == 1) return 1;
return fibonacciR(n-1) + fibonacciR(n-2);
} // shorter code
```

Base Case

Recursive Formula

```
n=21  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=22  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=23  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=24  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=25  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=26  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=27  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=28  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=29  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=30  Iterative Time: 0  Recursive Time: 4   Difference: 4
n=31  Iterative Time: 0  Recursive Time: 6   Difference: 6
n=32  Iterative Time: 0  Recursive Time: 10   Difference: 10
n=33  Iterative Time: 0  Recursive Time: 18   Difference: 18
n=34  Iterative Time: 0  Recursive Time: 32   Difference: 32
n=35  Iterative Time: 0  Recursive Time: 42   Difference: 42
n=36  Iterative Time: 0  Recursive Time: 75   Difference: 75
n=37  Iterative Time: 0  Recursive Time: 115   Difference: 115
n=38  Iterative Time: 0  Recursive Time: 181   Difference: 181
n=39  Iterative Time: 0  Recursive Time: 321   Difference: 321
n=40  Iterative Time: 0  Recursive Time: 481   Difference: 481
```

Tracing Recursive Functions

- Write function calls one by one.
- Write function name in a letter if possible.
- Be aware of what need to be added and what need to be returned.
- Trace them step by step.

39. Consider the following recursive method.

```
public int recur(int n)
{
  if (n <= 10)
    return n * 2;
  else
    return recur(recur(n / 3));
}
```

What value is returned as a result of the call `recur(27)` ?

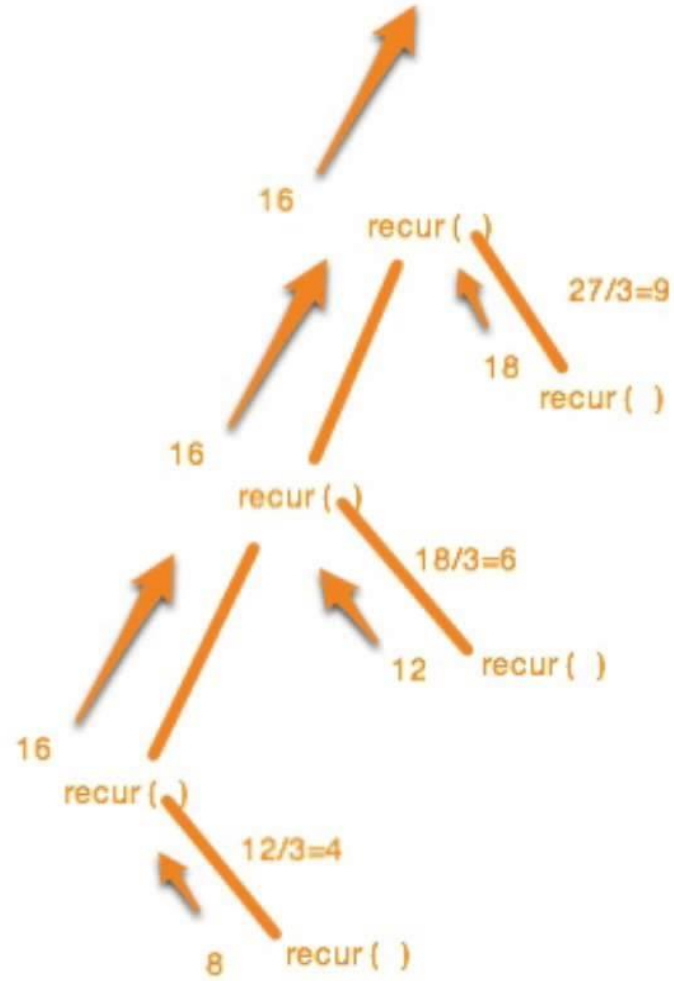
(A) 8

(B) 9

(C) 12

(D) 16

(E) 18



Write down the recursive steps one by one like a robot!!!
recur(27)-> recur(recur(9))->
recur(18)->recur(recur(6))->
recur(12)-> recur(recur(4))->
recur(8) -> return 16

SECTION 4

Recursive Functions

Palindrome Examples

- A palindrome is a word, phrase, number or sequence of words that reads the same backwards as forwards. Punctuation and spaces between the words or lettering is allowed (or not in CS).

Single Word Palindromes

- Anna
- Civic
- Kayak
- Level
- Madam
- Mom
- Noon
- Racecar
- Radar
- Redder
- Refer
- Repaper
- Rotator
- Rotor
- Sagas
- Solos
- Stats
- Tenet
- Wow

Recursive Palindrome

- Recursively check the string's characters layer by layer. If two characters of the same layer are matched, then continue to check otherwise reporting false.
- If no characters (or one characters left) then declare that the input string is indeed a palindrome word.

```java
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  private static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

Adding a Helper to Remove Unrelated Characters and Allow Case-Insensitiveness

- Convert all letters to lowercase (check palindrome under case-insensitive criteria)
- Remove non-digit and non-letter characters.

```
//helper function
public String helper(String s){
    s = s.toLowerCase();
    String result = "";
    //int j = 0;
    //for loop
    for(int i = 0; i < s.length(); i++ ){
        if(Character.isDigit(s.charAt(i))|| Character.isLetter(s.charAt(i)))result += s.charAt(i);
    }
    return result;
}
```

```java
public class Palindrome
{
    //helper function
    public String helper(String s){
        s = s.toLowerCase();
        String result = "";
        //int j = 0;
        //for loop
        for(int i = 0; i < s.length(); i++ ){
            if(Character.isDigit(s.charAt(i))|| Character.isLetter(s.charAt(i)))result += s.charAt(i);
        }
        return result;
    }

    //isPalindrome function
    public boolean isPalindrome(String s){
        s = helper(s);
        // if loop
        if(s.length()== 0)return true;
        if(s.length()== 1)return true;
        if(s.length()== 2)return s.charAt(0)== s.charAt(1);
        return (s.charAt(0)== s.charAt(s.length()-1))&& isPalindrome(s.substring(1, s.length()-1));
    }

}
```

```java
import java.util.Scanner;
public class PalindromeTest
{
    //main function
     public static void main(String[] args){
         //Scanner function
        Scanner input = new Scanner(System.in);
        boolean done = false;
        //while loop
        while(!done){
            System.out.println("Please enter a String or a Word: ");
            String word = input.nextLine();
            Palindrome rp = new Palindrome();
            System.out.println(rp.isPalindrome(word));
            System.out.println("Do you want to continue(Y/N)? ");
            String yes = input.nextLine();
            //if loop
            if(yes.length()!= 0 && (yes.charAt(0)=='Y' || yes.charAt(0)== 'y')) done = false;
            else done = true;
        }
    }

}
```

SECTION 5

Advanced Accumulate Algorithm

Advanced Algorithms

- In the previous Algorithms section (Week 4), you learned basic algorithms for swapping, copying, searching, accumulating, and finding the highest or lowest value in as list.
- What is the information being processed is buried in some complex data structure that is also buried in a class hierarchy?
- Or what if you are asked to solve a problem that you've never seen before?
- This concept focuses on algorithms that require you to algorithms for different situations.

Answering Un-seen Problem?

- Don’t panic!!!
- The problem is designed to be finished within 25 mins (90/4 = 22.5 mins)
- The answer won’t be longer than 30 lines.
- The answer can only be a method, a class or a program unit. It won’t involve 10 methods or classes.
- So, find out what the exam designers want! It must be quite simple.

Tips

- If there is a method to be called in the problem description, even if the implementation is omitted, just use it.
- If part a has a method designed, but you do not know how to answer part a, you can still use the method for part b or part c. They won't deduct points from you in part b or c.
- If you have written for more than 20 lines of code and still do not know the right way to finish it, then you are moving into a wrong direction. Try some a new way.

Advanced Accumulate Algorithm

- You have been hired by the high school baseball team to write a program that calculates statistics for the players on the team. You decide to have a class called **BaseballPlayer**.
- Every **BaseballPlayer** has a **name**, a **numberOfHits**, and a **numberOfAtBats**. The BaseballRunner has an array of BaseballPlayer objects called roster. **Your goal is to find the team batting average.**

Read the Program Structure First

```
1  public class BaseballPlayer{
2    private String name;
3    private int hits;
4    private int atBats;
5    BaseballPlayer(String n, int a, int h){
6        name=n; hits=h; atBats=a;
7      }
8    public String getName(){ return name; }
9    public int getHits()   { return hits;}
10   public int getAtBats() {return atBats; }
11   public double getBattingAverage() {
12       /* implementation not shown */
13       return (double) getHits()/getAtBats();
14     }
15 }
```

- Read this class and understand that this class is a class for each individual player. Some data structure may organize it. But the getter, setter maybe very useful.
- I filled in the getBattingAverage() but it is not related to the final solution for the problem.

Read the Program Structure First

```
1  public class BaseballRunner{
2      public static void main(String[] args){
3          BaseballPlayer[] roster; // array of players
4          /* Additional implementation now shown*/
5       }
6
7       public static double findTeamAverage(BaseballPlayer[] arr){
8          /* implementation to be compelted */
9          return 0.0;  //use for compiler
10      }
11 }
```

- This problem is in

  ProgramTester -> Data Class

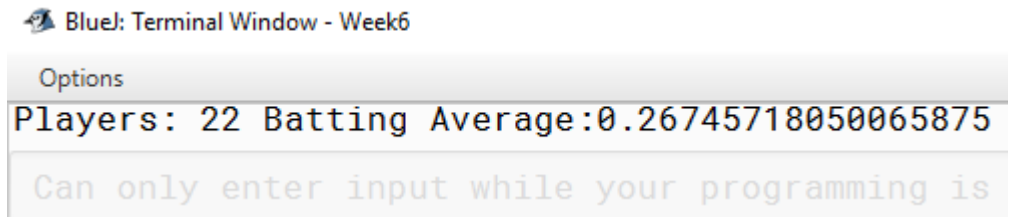
  format.

- Only the team average need to be found. The additional implementation in the main() should be preparing the data.

- The findTeamAverage is the function to be implemented. Just work on this.

Testing Result Using Yankees' Roster Info

```
BaseballPlayer[] roster = {
    new BaseballPlayer("Eric Kratz",     2,2),
    new BaseballPlayer("Miguel Andujar", 7,4),
    new BaseballPlayer("Michael Pineda", 3,1),
    new BaseballPlayer("Garrett Cooper", 43,14),
    new BaseballPlayer("Starlin Castro", 443,133),
    new BaseballPlayer("Ronald Torreyes", 315,92),
    new BaseballPlayer("Didi Gregorius", 534,154),
    new BaseballPlayer("Aaron Judge", 542,154),
    new BaseballPlayer("Gary Sanchez", 471,131),
    new BaseballPlayer("Chase Headley", 512,140),
    new BaseballPlayer("Ji-Man Choi", 15,4),
    new BaseballPlayer("Aaron Hicks", 301,80),
    new BaseballPlayer("Brett Gardner", 594,157),
    new BaseballPlayer("Jacoby Ellsbury", 356,94),
    new BaseballPlayer("Mason Williams", 16,4),
    new BaseballPlayer("Clint Frazier", 134,31),
    new BaseballPlayer("Matt Holliday", 373,86),
    new BaseballPlayer("Tyler Austin", 40,9),
    new BaseballPlayer("Todd Frazier", 194,43),
    new BaseballPlayer("Austin Romine", 229,50),
    new BaseballPlayer("Chris Carter", 184,37),
    new BaseballPlayer("Luis Severino", 5,1),
  };
```



http://www.espn.com/mlb/team/stats/batting/_/name/nyy/new-york-yankees

SECTION 6

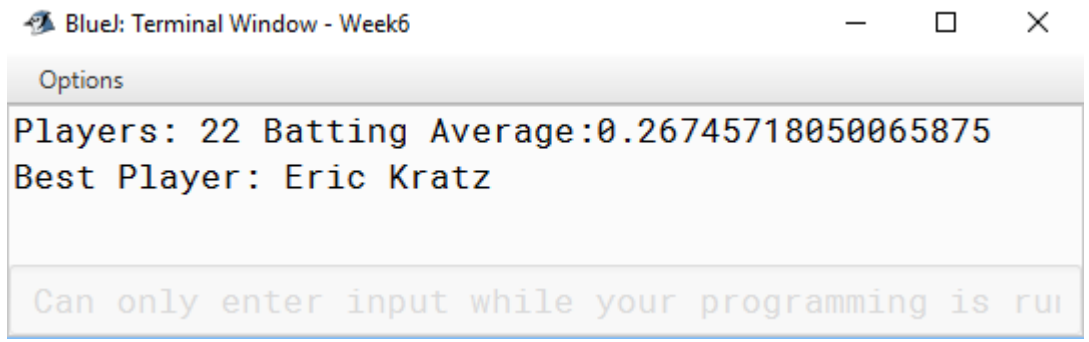
Advanced Highest (Min/Max) Algorithm

Advanced Find-Highest Algorithm

- General Problem: Given a roster of baseball players, find the name of the player that has the highest batting average.
- Final Problem: Write a method called findBestPlayer that has one parameter: a BaseballPlayer array. The method should return a String that is the name of the player that has the highest batting average. Make sure a player exists before processing him. If two players have the same high average, only select the first player. Also, adapt your algorithm to work with an arrayList of BaseballPlayer objects.

Using the getBattingAvarage() as Key Value. Keep track of a Value for the Highest Value

```
public static String findBestPlayer(BaseballPlayer[] arr){
      String bestPlayer="";
      double maxBA = Double.MIN_VALUE;
      for (int i=0; i<arr.length; i++){
            if (arr[i].getBattingAverage() > maxBA){
                maxBA = arr[i].getBattingAverage();
                bestPlayer = arr[i].getName();
             }
        }
      return bestPlayer;
 }
```



SECTION 7

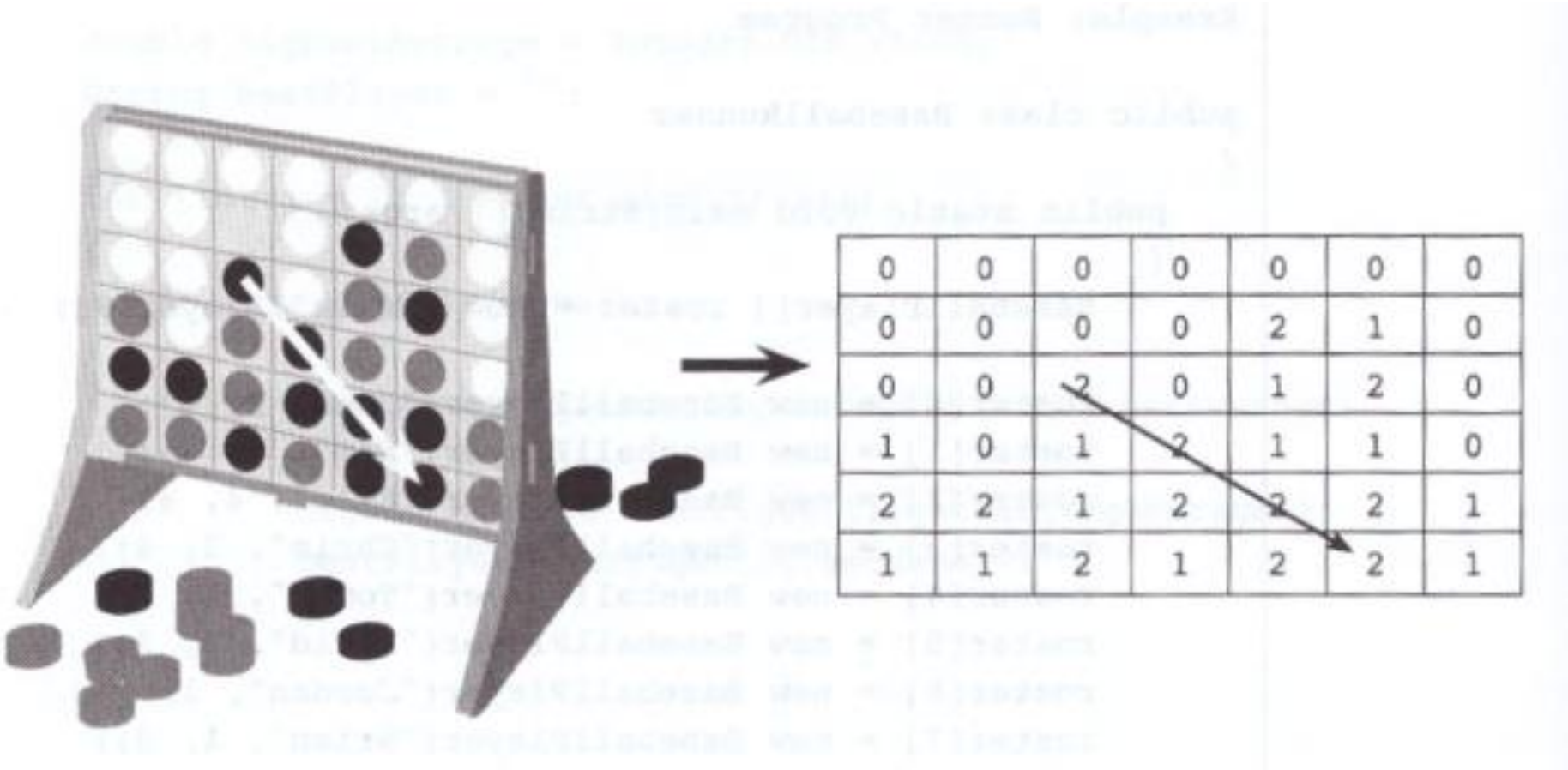
The Connect Four Advanced Algorithm

General Problem: Determine if a player has won the game of Connect Four.

Refined Problem: The Connect Four app version uses a 2-D array that is 6 × 7. After a player makes a move, determine if he or she has won by checking all possible ways to win. If four of the same color disks are found in a line, then the game is over. Decide who won the game (player one or player two). A way to win can be vertical, horizontal, diagonally downward, or diagonally upward.

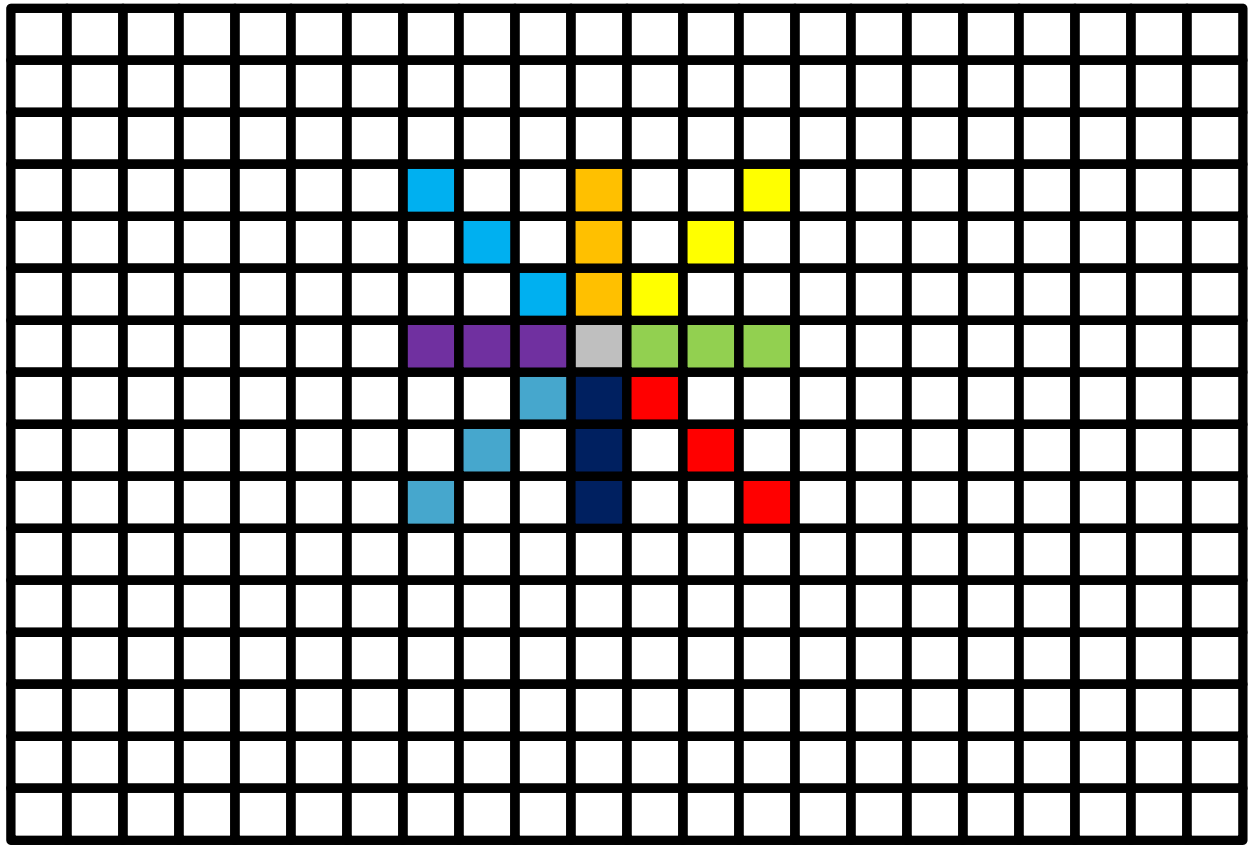
Final Problem: Write a method called checkForWinner that has one parameter: a 2-D array of int. The 2-D array consists of 1s and 2s to represent a move by player one or player two. Cells that do not contain a move have a value of 0. This method checks all four directions to win (vertical, horizontal, diagonally downward, and diagonally upward). The method should return the number of the player who won or return 0 if nobody has won.

chekForWinner()

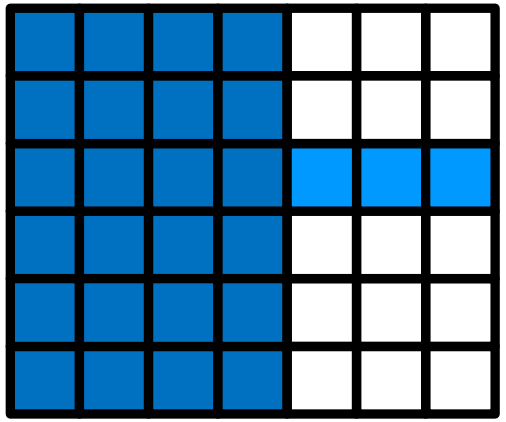


0	0	0	0	0	0	0
0	0	0	0	2	1	0
0	0	2	0	1	2	0
1	0	1	2	1	1	0
2	2	1	2	2	2	1
1	1	2	1	2	2	1

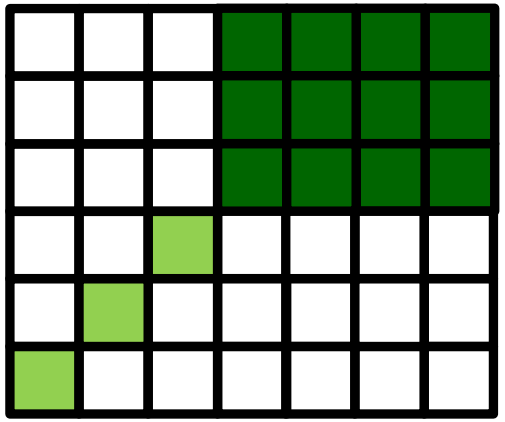
What is the winner condition?



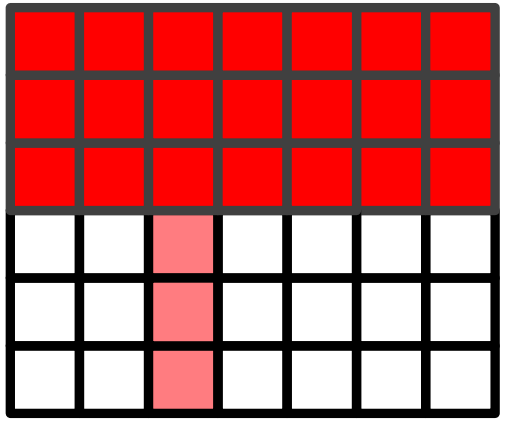
1. One block can be involved in 32 possible combinations.
2. But, in reality, we just need to check 4 that started from a block.
3. Domain check, range check.
4. Be careful about boundary conditions.



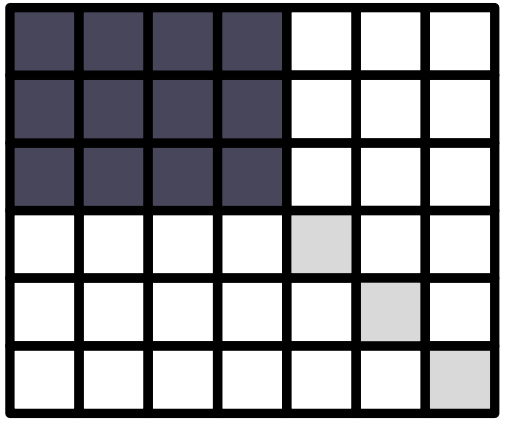
Horizontal Check



Slash Diagonal Check



Vertical Check



Back-Slash Diagonal Check

Runner Program

```
1  public class ConnectFourRunner{
2        public static void main(String[] args){
3              int[][] board = {
4                {0, 0, 0, 0, 0, 0, 0},
5                {0, 0, 0, 0, 2, 1, 0},
6                {0, 0, 2, 0, 1, 2, 0},
7                {1, 0, 1, 2, 1, 1, 0},
8                {2, 2, 1, 2, 2, 2, 1},
9                {1, 1, 2, 1, 2, 2, 1}
10              };
11
12             System.out.println("Winner:" + checkForWinner(board));
13          }
14       public static int checkForWinner(int[][] board){ return 0; }
15 }
```

```
// Horizontal check
for (int i=0; i<board.length; i++){
    for (int j=0; j<board[i].length-3; j++){
        if (
         board[i][j] != 0 && board[i][j]==board[i][j+1] &&
         board[i][j]==board[i][j+2] && board[i][j]==board[i][j+3]
        ) return board[i][j];
    }
}
// Vertical check
for (int i=0; i<board.length-3; i++){
    for (int j=0; j<board[i].length; j++){
        if (
         board[i][j] != 0 && board[i][j]==board[i+1][j] &&
         board[i][j]==board[i+2][j] && board[i][j]==board[i+3][j]
        ) return board[i][j];
    }
}
```

```
// slash check
for (int i=0; i<board.length-3; i++){
    for (int j=3; j<board[i].length; j++){
        if (
         board[i][j] != 0 && board[i][j]==board[i+1][j-1] &&
         board[i][j]==board[i+2][j-2] && board[i][j]==board[i+3][j-3]
        ) return board[i][j];
      }
   }
// back slash check
for (int i=0; i<board.length-3; i++){
    for (int j=0; j<board[i].length-3; j++){
        if (
         board[i][j] != 0 && board[i][j]==board[i+1][j+1] &&
         board[i][j]==board[i+2][j+2] && board[i][j]==board[i+3][j+3]
        ) return board[i][j];
      }
   }
```

SECTION 8

The Twitter-Sentiment-Analysis

The Twitter-Sentiment-Analysis Advanced Algorithm

- Twitter has become very popular and Twitter Sentiment Analysis has grown in popularity. The idea behind Twitter Sentiment is to pull emotions and/or draw conclusions from the tweets.
- A large collections of tweets can be used to make general conclusions of how people feel about something. The important words in the tweet are analyzed against a library pf words to give a tweet a sentiment score.

The Twitter-Sentiment-Analysis Advanced Algorithm

- One of the first steps in processing a tweet is to remove all of the words that don't add any real value to the tweet. These words are called **stop words**, and this step is typically part of a phase called **preprocessing**.
- For this problem, you job is to remove all the stop words from the tweet. There are many different ways to find meaningless words, but for our example, the stop words will be all words that are three characters long or less.

Read the requirement

- **General Problem:** Given a tweet, remove all of the meaningless words.
- **Refined Problem:** You are given a tweet as a list of words. Find all of the words in the list that are greater than three characters and add them to a new list. Leave the original tweet unchanged. The new list will contain all the words that are not stop words.
- **Final Problem:** Write a method called **removeStropWords** that has one parameter: an ArrayList of Strings. The method should return a new ArrayList of Strings that contains only the words from the original list that are **greater than three characters long**. **Do not** modify the original ArrayList. Also, modify the code to work on a String array.

Requirement for the Method

- Keep string longer than 3.
- Non-destructive copy.

The Runner Class

```
1  import java.util.ArrayList;
2  public class TweetSentimentRunner{
3      public static void main(String[] args){
4          ArrayList<String> tweet = new ArrayList<String>();
5          tweet.add("If");
6          tweet.add("only");
7          tweet.add("Bradley's");
8          tweet.add("arm");
9          tweet.add("was");
10         tweet.add("longer");
11         tweet.add("best");
12         tweet.add("photo");
13         tweet.add("ever");
14         ArrayList<String> processedTweet = removeStopWords(tweet);
15         System.out.println(processedTweet);
16     }
17
18     public static ArrayList<String> removeStopWords(ArrayList<String> arr){ return null; }
19 }
```

```java
import java.util.ArrayList;
public class TweetSentimentRunner{
    public static void main(String[] args){
        ArrayList<String> tweet = new ArrayList<String>();
        tweet.add("If");
        tweet.add("only");
        tweet.add("Bradley's");
        tweet.add("arm");
        tweet.add("was");
        tweet.add("longer");
        tweet.add("best");
        tweet.add("photo");
        tweet.add("ever");
        ArrayList<String> processedTweet = removeStopWords(tweet);
        System.out.println(processedTweet);
     }

    public static ArrayList<String> removeStopWords(ArrayList<String> arr){
        ArrayList<String> a = new ArrayList<String>();

        for (String s: arr){ if (s.length()>3) a.add(s); }
        return a;
     }
}
```

BlueJ: Terminal Window - Week6

Options

```
[only, Bradley's, longer, best, photo, ever]
```

Can only enter input while your programming is running