# Course Description

**Weekly Overview**

This week begins a more formal presentation of algorithms by looking at searching algorithms and complexity analysis. Students should be able to code linear and binary search and should be able to conceptually define the order of growth of functions to describe the efficiency of an algorithm.

# Institutional Learning Outcomes

**Main Objectives**

* Write linear search.
* Write binary search.
* Describe the efficiency of algorithms using order of growth.
* Classify linear search and binary search in terms of the order of growth.

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Searching

Day 2: The Binary Search

Day 3: Searching Lab

Day 4: Complexity Analysis

Day 5: Flexible

**Included Resources**

Searching Exploration Student Activity

Lecture Notes: The Binary Search

Lab Assignment: Searching Lab

Java Class: Timer.java

Java Class: SearchingMachine.java

Lecture Notes: Complexity Analysis

Homework Assignment: Complexity Analysis

**In-Class Activity: Searching through an Array**

Part I: Programming a searching algorithm

Work with a partner to program the following method that determines whether a particular value occurs in an array.

//A is an array of ints, v is a search value

//returns true if v occurs at least once in A

//returns false if v does not occur at least once in A

public static boolean search(int[] A, int v){

}

Part II: Analyzing the speed of the algorithm

How fast does your algorithm work? Complete the chart below. “Best Case” is defined to be the smallest number of values that the algorithm needs to check in order to return the correct boolean. “Worst Case” is defined to be the greatest number of values that the algorithm needs to check in order to return the correct boolean. “Average Case” is defined to be the average expected number of values that the algorithm needs to check in order to return the correct boolean.

|  |  |  |  |
| --- | --- | --- | --- |
| **Length of the Array** | **Best Case** | **Worst Case** | **Average Case** |
| 10 |  |  |  |
| 50 |  |  |  |
| 100 |  |  |  |
| 10,000 |  |  |  |
| 1,000,000 |  |  |  |
| *n* |  |  |  |

There are approximately 450 million social security numbers that have been issues. If a computer can perform 85 million “looks” per second, how long will it take (worst, best, average) to search through an array that has every possible SSN to determine if a particular value (v) is a validly issued number?

Part III: Brainstorming

Is there anyway that you can think to speed up your algorithm?

**Lecture Notes: The Binary Search**

**Bell Work (5 minutes)**

Which algorithm is “faster” on average? Why?

//Algorithm A

public static boolean search(int[] A, int v){

for(int i=0; i<A.length; i++)

if(A[i] == v)

return true;

return false;

}

//Algorithm B

public static boolean search(int[] A, int v){

boolean b = false;

for(int i=0; i<A.length; i++)

if(A[i] == v)

b = true;

return b;

}

***Answer:*** Algorithm A is faster on average because it returns immediately upon finding the required value *v*. Algorithm B goes through the entire array every time, even if it find the value in the first spot.

**Main Lecture: Part 1 (10 minutes)**

This is the famous “dictionary presentation.” Have a rather lengthy dictionary on hand, and ask someone in the class to choose a “main entry” word. (In other words, the student cannot pick a plural of a noun or a past tense of a verb, unless these are explicitly listed as a main entry word.

Ask the students to guess how many “yes or no” questions you, the teacher, will need in order to guess their word. Inform the students that you think you can accomplish this in twenty questions or less.

The algorithm is to go to the middle of the dictionary (approximately half way is okay), and ask, “Does your words come alphabetically *before* the word \_\_\_\_\_\_\_\_\_\_.” You will continue cutting the number of entries in half until you have the correct word. Two hints are helpful. First, keep track of the “boundary” words on the board so that you do not lose track of where the word is located. Second, make sure, especially towards the end, that if you ask, “Does your word come alphabetically before the word *geography*?” and the word *is* geography, the student will say, “No.” A final hint may also be helpful: practice this ahead of time with a family member or friend a few times.

The algorithm in the Warm Up is known as “Linear Search.” It is simple – start at the beginning, and search until you find the required value. The first version of it has something called an “early bailout”, which halts the algorithms upon finding the value. It is a way to speed up the average time (and certainly best-case time) for the search, but it does not fundamentally change the algorithm itself. The worst-case scenarios are identical: for an array of size *n*, it will take *n* “looks”.

The dictionary algorithm is known as “Binary Search.” Note that it requires that the array be *ordered*. That is essential. Without an ordered array, you must perform a Linear Search.

**Main Lecture – Part 2 (25 minutes)**

Work out the code with students for Binary Search. If there is time, you can have them start to code it (on paper) based on the demonstration. If not, you can simply go through it Socratically with them. For reference, the code is:

public static boolean binarySearch(int[] A, int v){

int low = 0;

int high = A.length - 1;

while(high >= low) {

int middle = (low + high) / 2;

if(A[middle] == v) {

return true;

}

if(A[middle] < v) {

low = middle + 1;

}

if(A[middle] > v) {

high = middle - 1;

}

}

return false;

}

Step through the code and make sure that students see the parallel with what you did using the dictionary.

**Main Lecture – Part 3 (10 minutes)**

The question is, how fast is the Binary Search? It seems much faster than Linear Search.

Have students fill out the following chart:

|  |  |  |
| --- | --- | --- |
| **Length of the Array** | **Best Case** | **Worst Case** |
| 8 |  |  |
| 4096 |  |  |
| 131,072 |  |  |
| 1,048,576 |  |  |
| 1,000,000,000 |  |  |
| *n* |  |  |

The “answer” is that Binary Search takes approximately log2*n* “looks” for an array of size *n*. Because the logarithm function grows very slowly, this is a *very* fast algorithm.

Students previously explored how long it would take to search through all 450,000,000 valid SSNs for find whether a given number is a valid SSN. Given that the computer can look at 85 million values per second, how long would it take the computer to search using Binary Search? Compute this value with students.

The answer is about 29 “looks”, which would take 3.4 x 10–7 seconds. Compare this with the 5.3 seconds for the Linear Search.

**Homework:** Pass out the StopWatch.java file and ask students to read it and try to understand how to use it to time code. It will be used in tomorrow’s lab.

**Linear and Binary Search Lab Assignment**

1. Pull down the StopWatch.java code from GitHub.
2. Pull down the SearchingMachine.java code from GitHub.
3. Fill in the code for Linear Search and Binary Search.
4. Time the code on various values of *n*, and fill in the chart below.
5. Use the points to generate a linear function for Linear Search and a logarithmic function for Binary Search. Use these functions to figure out how much time it would take to search through 100 trillion values.
6. Submit your code through GitHub, but your graphs and calculations on paper.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Length of the Array** | **Linear Search Time (milliseconds)** |  | **Length of the Array** | **Binary Search Time (milliseconds)** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*f*(*n*) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *g*(*n*) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*f*(1,000,000,000,000) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_ *g*(1,000,000,000,000) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Coding Hint: Binary Search is so fast that you may have to imbed the search into a* for *loop that runs several (lots of) times in a row, and then divide the total time by the number of runs.*

//Stopwatch.java

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac Stopwatch.java

\* Execution: java Stopwatch n

\* Dependencies: none

\*

\* A utility class to measure the running time (wall clock) of a program.

\*

\* % java8 Stopwatch 100000000

\* 6.666667e+11 0.5820 seconds

\* 6.666667e+11 8.4530 seconds

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* The {@code Stopwatch} data type is for measuring

\* the time that elapses between the start and end of a

\* programming task (wall-clock time).

\*

\* See {@link StopwatchCPU} for a version that measures CPU time.

\*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

public class Stopwatch {

private final long start;

/\*\*

\* Initializes a new stopwatch.

\*/

public Stopwatch() {

start = System.currentTimeMillis();

}

/\*\*

\* Returns the elapsed CPU time (in seconds) since the stopwatch was created.

\*

\* @return elapsed CPU time (in seconds) since the stopwatch was created

\*/

public double elapsedTime() {

long now = System.currentTimeMillis();

return (now - start) / 1000.0;

}

/\*\*

\* Unit tests the {@code Stopwatch} data type.

\* Takes a command-line argument {@code n} and computes the

\* sum of the square roots of the first {@code n} positive integers,

\* first using {@code Math.sqrt()}, then using {@code Math.pow()}.

\* It prints to standard output the sum and the amount of time to

\* compute the sum. Note that the discrete sum can be approximated by

\* an integral - the sum should be approximately 2/3 \* (n^(3/2) - 1).

\*

\* @param args the command-line arguments

\*/

public static void main(String[] args) {

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

// sum of square roots of integers from 1 to n using Math.sqrt(x).

Stopwatch timer1 = new Stopwatch();

double sum1 = 0.0;

for (int i = 1; i <= n; i++) {

sum1 += Math.sqrt(i);

}

double time1 = timer1.elapsedTime();

System.out.println("%e (%.2f seconds)\n" + sum1 + "\n" + time1);

// sum of square roots of integers from 1 to n using Math.pow(x, 0.5).

Stopwatch timer2 = new Stopwatch();

double sum2 = 0.0;

for (int i = 1; i <= n; i++) {

sum2 += Math.pow(i, 0.5);

}

double time2 = timer2.elapsedTime();

System.out.println("%e (%.2f seconds)\n" + sum2 + "\n" + time2);

}

}

//SearchingMachine.java

/\*\*

\* The searching machine fills an array with random numbers and

\* searches through the array for a value using both

\* linear search and binary search

\*

\* @author (Jake Tawney)

\* @version (1.0)

\*/

import java.util.Random;

public class SearchingMachine

{

static Random randomGenerator = new Random();

public static void fillArray(int[] data){

for(int i=0; i<data.length; i++){

data[i] = randomGenerator.nextInt();

}

}

public static boolean linearSearch(int[] data, int value){

return true; //delete and fill in code for linear search

}

public static boolean binarySearch(int[] data, int value){

return true; //delete and fill in code for binary search

}

//Okay, let's use these methods

public static void main(String[] args){

int size = 10000;

int[] A = new int[size];

fillArray(A);

//Search for a value that is likely not in the array

int searchValue = randomGenerator.nextInt();

boolean isThere = linearSearch(A, searchValue);

System.out.println(searchValue + ": " + isThere);

//Search for a value that is definitely in the array

int index = randomGenerator.nextInt(size);

isThere = linearSearch(A, A[index]);

System.out.println(A[index] + ": " + isThere);

}

}

**Lecture Notes: Complexity Analysis**

**Bell Work (5 minutes)**

How many times does the following while loop execute?

n = 3000

while(n > 1){

n = n / 3;

}

***Answer:*** 8 times

**Main Lecture: Part 1 (25 minutes)**

Computer Scientists are very interested in classifying algorithms based on their speeds. This has great theoretical, but also practical concerns. An algorithm is no good if it takes too long to complete. The trick is to develop ways to categorize speed for *large values only*. For small values, the speed is very quick and dependent almost entirely on hardware. As the amount of data gets bigger, it is the algorithm that becomes more important.

There are two ways to describe the functions that classify algorithm speed:

We use *tilde notation* to develop simpler approximate expressions. First, we work with the *leading term* of mathematical expressions by using a mathematical device known as the tilde notation. The idea is similar to the leading term of a polynomial, but we can have other terms than just powers.

For example. The dominating term in *n*3 – 5*n*5 + 2*n* is 2*n*. For large values of *n*, the other terms are insignificant. Therefore, *n*3 – 5*n*5 + 2*n* ~ 2*n*.

Officially, ** if .

With this notation, we can ignore complicated parts of an expression that represent small values.

For computer code, or algorithms, *n* is often the number of data points, and *f*(*n*) is the number of instructions that need to be executed in order to complete the algorithm. One reason for simplifying this using *tilde* is that it is too complicated to count *actual* instructions. For example, the first three lines of pseudo-code in the following chunk are insignificant compared to the loop:

print(“Enter a value of n:”);

int n = getValue();

for(int i=0; i< ***n***; i++)

System.out.println(i);

It is the loop that governs the amount of time it takes for the code to run when values of *n* are large.

However, even the *tilde* notation is a little too specific for certain things. For example, 6*n*100 and *n*100 are not related by *tilde*. However, they are in the same general category, or at least they are much more similar to one another than either one is to *n*2 or 2*n*.

The most common way that computer scientists classify algorithms are to group them together using an **order of magnitude** or **order of growth**.

The key point in analyzing the running time of a program is this: for a great many programs, the running time satisfies the relationship *T*(*n*)∼*cf*(*n*)*T*(*n*)∼*cf*(*n*) where *c* is a constant and *f*(*n*) is a function known as the *order of growth* of the running time.

For typical programs, *f*(*n*) is a function such as log2*n*, *n*, *n*log2*n*, *n*2, log2*⁡n*, *n*, *n*2, or *n*3.

*Note to teacher: There are several formal constructs for algorithm analysis, such as O(n) and o(n). These are to provide things like upper and lower bounds for the order of growth. For this course, we will not focus on those, but will instead just talk about the general category of “order of growth”. So, for example, while mathematically it is true that 4n = O(n) and 4n = O(n2), because n2* is *and upper bound for 4n*, *the grouping of 4n with n is much more significant for our purposes than the statement 4n = O(n2).*

**Main Lecture: Part 2 (15 minutes)**

We will need to give several examples of code that has various orders of growth. Begin with a nested loop:

for(int i=0; i<n; i++)

for(int j=0; j<i; j++)

System.out.println(“\*”);

This is tricky because the number of statements executed in the inner loop decreases as *i* increases. Students should be led through how to calculate the number of stars printed, and then they should see that this is an *n*2 algorithm.

Two things should be pointed out before the class ends:

1. Linear Search, in the worst case, is an order *n* algorithms. Binary Search is an order log2*n* algorithm.
2. Of the more “famous” order of magnitude, the following is the “order of their orders”:

1 < log2*n* < *n* < *n*log2*n* < *n*2 < *n*3 < … < *nk* < 2*n* < 3*n* < … < *kn* < *n*! < *nn*

(Note that “1” is a constant time algorithm. It does not depend on the size of the date. For example, printing the first element in an array – it does not matter how large the array is, it still takes the same amount of time to access the and print the first element. The complexity category can either be described as “1” or “*c*”, for “constant.”)

This ordering is very important and has tremendous implications for future weeks. We would *much* rather have a *n*3 algorithm than a 2*n* algorithm. Likewise, we would rather have a 2*n* algorithm than an *n*! algorithm.

To drive the point home, it would be helpful to look at the actual functional values for numbers for as *n* = 100. How big is 1002? What about 100log2100? What about 2100? What about 100!? What about 100100? Can you prove that 100100 > 100! or that 2100 < 100!?

**Homework:** Complexity Analysis worksheet.

**Homework: Complexity Analysis**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Give the order of magnitude for each of the following functions and algorithms.

1. 2*n*2­ ­­– ­3*n* + 7*n*3 + 8
2. log(*n*) + 3sqrt(*n*)
3. 18
4. log2(*n*) – 4
5. 2*n* + 5*n* + 3*n*
6. *n*2009 + 2*n*
7. *n*! + 2*n*
8. *n*! + *nn*

9. for(int i=0; i<n; i++)

System.out.println(i);

10. for(int i=0; i<n; i++)

linearSearch(myList,"17");

11. for(int i=0; i<n; i++)

binarySearch(myList,"17");

12. for(int i=0; i<10; i++)

System.out.println(i);

13. for(int i=0; i<2^n; i++)

System.out.println(i);

14. for(int i=0; i<n; i++)

for(int j=0; j<i; j++)

System.out.println(i);

15. for(int i=0; i<n; i++)

for(int j=i; j<n; j++)

for(int k=n; k>j; k--)

System.out.println(i);

16. for(int i=0; i<n; i++)

for(int j=0; j<n; j++)

for(int k=0; k<99; k++)

System.out.println(i);

17. for(int i=0; i<9876; i++)

for(int j=0; j<n; j++)

for(int k=0; k<99; k++)

System.out.println(i);

18. for(int i=n; i>0; i=i/10)

System.out.println(i);

19. for(int i=1; i<2^n; i=i\*2)

System.out.println(i);

20. for(int i=0; i<n; i++){

linearSearch(myList, "17");

binarySeachr(myList, "18");

}

21. for(int i=0; i<100; i++)

for(int j=0; j<100; j++)

for(int k=0; k<100; k++)

for(int g=0; g<100; g++)

binarySearch(myList, "17");

22. Use limits to show that all logarithms, regardless of base, are in the same order of magnitude category.

23. Use limits to show that *an* and *bn* are *not* in the same complexity category (unless *a* = *b*).