

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 10 – Sorting and Searching

(Week 12, starting 4 April 2016)

1. High-Tech Library

A new high-tech library is opening near your home – And you have been tasked to help design a system that facilitates searching of titles. Being a smart CS1020 student, you know that you should sort the books in ascending order, so that they can be found quickly using Binary Search.

- (a) The librarian Mary suggests using Radix Sort to sort the books by their title. She argues that Radix sort is the best candidate because it is an $O(n)$ algorithm. Do you agree? What are some potential problems of using this algorithm?

Answer: Radix sort may not be very suitable for this purpose. The biggest reason for this is its very high memory use. For each pass of the algorithm, the input needs to be sorted into buckets. Since we are sorting by title, we will need to set aside space for **many buckets** – As many as 128 buckets every iteration, if we want to support the full ASCII character set.

In addition, Radix Sort is not necessarily $O(n)$. While often quoted in literature as $O(n)$, the time complexity of radix sort is more correctly $O(dn)$, where d is the maximum length of the elements of the input data. “ d ” is often left out because it is usually a known small constant.

However, in this case, we do not know what d is – Some book titles can be very long. The length of the longest book title determines Radix Sort’s run time!

- (b) Suggest a modification to Mary’s idea to make using Radix Sort feasible. Discuss any possible drawbacks in your solution.

Answer: There are many different suggestions that could work. This list is not exhaustive!

- Sort by Dewey Decimal Number or ISBN instead: These numbers have a very consistent length, and only need 10 buckets since the possible values are 0-9. However, the drawback of this is that users can only search by DDN/ISBN.
- (Advanced answer) Break down the strings into bits instead, so that for each pass of Radix sort, there will only ever be two buckets (0 or 1). While this helps with the memory problem, d becomes *much* larger. Speed will be adversely affected.

For more information on the pros and cons of Radix sort, please refer to the following:

<http://www.cprogramming.com/tutorial/computersciencetheory/radix.html>

- (c) After much brainstorming, you’ve narrowed down your choices of sorting algorithms to Quicksort and Merge Sort. Which would you pick, and why?

Answer: You can pick either, as long as you justify your choice! Quicksort is not stable and has an $O(n^2)$ worst case time complexity. However, the worst case time rarely happens, and the algorithm has the advantage of being in-place. Mergesort is stable, but is not in-place.

(Advanced) In practical applications, quicksort is usually preferred:

<https://stackoverflow.com/questions/70402/why-is-quicksort-better-than-mergesort>

(Hint: Remember while answering this question note that speed is not the only concern!)

2. Taking Advantage of Java Sorting

In practical applications, we do not usually have to code our own sorting algorithms. Instead, we can simply make use of what is provided by Java, namely the `Collections.sort()` function. What's really interesting about this function is that it can sort a list of any generic type! In this question, we explore how and why that works.

To do this, we have a `Coordinate` Class, which holds `x`, `y` and `z` coordinates of a point in space. We want to be able to sort a list of `Coordinates` by their distance from the origin `(0,0,0)`. The following is a simplified skeleton of the coordinate class:

```
public class Coordinate {
    private double _x;
    private double _y;
    private double _z;

    public Coordinate(double x, double y, double z) {
        _x = x;
        _y = y;
        _z = z;
    }

    public double getX() { return _x; }
    public double getY() { return _y; }
    public double getZ() { return _z; }
}
```

In order for a list of `Coordinates` to be sorted the coordinates need to be **Comparable** to each other. Look into the Java API for the **Comparable** class, and implement what is required to make the `Coordinate` class sortable!

Suggested Solution:

```
public class Coordinate implements Comparable<Coordinate> {
    // Setup stuff already in the question is omitted

    public int compareTo(Coordinate other) {
        double distanceOfSelf = Math.sqrt( this.getX()*this.getX()
                                           + this.getY()*this.getY()
                                           + this.getZ()*this.getZ() );

        double distanceOfOther = Math.sqrt( other.getX()*other.getX()
                                           + other.getY()*other.getY()
                                           + other.getZ()*other.getZ() );

        if (distanceOfSelf < distanceOfOther) {
            return -1;
        }
        else if (distanceOfSelf > distanceOfOther) {
            return 1;
        }
        else {
            // Condition here: distanceOfSelf == distanceOfOther
            return 0;
        }
    }
}
```

Things you need to know about making a class comparable:

- You must implement the Comparable interface, with the correct generic type.
- The return value is from the “point of view” of the object the function is invoked on, i.e., what is returned is the answer to the question “Should *myself* be considered larger than, smaller than, or equal to *the other object*?”
- Return -1 if the current object is smaller, 1 if the object is larger, and 0 if they are equal.

3. Search Algorithms

After learning about Binary Search in CS1020, you decide to build a better version of the algorithm, which partitions the array into three parts instead of just two.

(a) In English, explain what you would need to change about Binary Search to make it partition a list into three instead of two. Then, modify the following code snippet to implement your idea.

```
public static int binSearchRecur(int[] input, int searchTerm,
                                int low, int high)
                                throws Exception {
    if (low > high) {
        throw new Exception("Item Not Found");
    }

    int mid = (low + high) / 2;
    if (searchTerm > input[mid]) {
        return binSearchRecur(input, searchTerm, mid+1, high);
    }
    else if (searchTerm < input[mid]) {
        return binSearchRecur(input, searchTerm, low, mid-1);
    }
    else {
        return mid;
    }
}
```

Answer: Generate two “midpoints” instead of just one. If neither of the midpoints refers to the correct item, the result (if present) must belong in one of the three sublists: From the beginning to the first midpoint, from the first midpoint to the second midpoint, or from the second midpoint to the end of the list.

The solution in code is as follows:

```
public static int enhancedSearchRecur(int[] input, int searchTerm,
                                     int low, int high)
    throws Exception {
    if (low > high) {
        throw new Exception("Item Not Found");
    }

    int interval = (high - low) / 3;
    int mid1 = low + interval;
    int mid2 = low + 2*interval;

    if (searchTerm < input[mid1]) {
        // Result is in lowest 1/3
        return enhancedSearchRecur(input, searchTerm, low, mid1-1);
    }
    else if (searchTerm > input[mid1] && searchTerm < input[mid2]) {
        // Result is in middle 1/3
        return enhancedSearchRecur(input, searchTerm, mid1+1, mid2-1);
    }
    else if (searchTerm > input[mid2]) {
        // Result is in highest 1/3
        return enhancedSearchRecur(input, searchTerm, mid2+1, high);
    }
    else if (searchTerm == input[mid1]) {
        // mid1 is the result
        return mid1;
    }
    else {
        // mid2 is the result
        return mid2;
    }
}
```

- (b) Since your algorithm is able to discard more information per iteration, it appears to be more efficient than Binary Search. Discuss whether this is the case using the big-O notation.

Answer: We begin by first analyzing how long Binary Search takes to run, and why:

1 st Run	2 nd Run	3 rd Run	...	2 nd last run	Last run
n items	n/2 items	n/4 items	...	2 items	1 item

Therefore, in addition to the first call, the number of steps required by Binary Search in the worst case is the number of times you have to divide the size of the input by two before it becomes 1.

Let the number of steps required be x. Therefore,

$$n \div 2^x = 1$$

$$n = 2^x$$

$$\log(n) = x \log(2)$$

$$x = \log(n) \div \log(2)$$

$$x = \log_2 n$$

Hence, under the big-O notation, the time complexity of Binary Search is $O(\log_2 n)$.

Each subsequent invocation of Enhanced Search reduces the search range to $\frac{1}{3}$, i.e.

1 st Run	2 nd Run	3 rd Run	...	2 nd last run	Last run
n items	n/3 items	n/9 items	...	3 items	1 item

Therefore, in addition to the first call, the number of steps required by Enhanced Search in the worst case is the number of times you have to divide the size of the input by three before it becomes 1.

Let the number of steps required be x . Therefore,

$$n \div 3^x = 1$$

$$n = 3^x$$

$$\log(n) = x \log(3)$$

$$x = \log(n) \div \log(3)$$

$$x = \log_3 n$$

$$x = (\log_2 n) \div (\log_2 3)$$

We discard constant coefficients under the big-O notation. Therefore, the worst case time complexity of Enhanced Search is still $O(\log n)$ – It is the same as traditional Binary Search under the big-O notation!

- (c) As we have now seen, Binary Search is fast and thus preferable over linear search. However, this is not always the case. Give an example of such a context, and support it with discussions on time complexity.

Answer: Sorting a list takes $O(n \log n)$ time, while a linear search takes $O(n)$ time. Therefore if we want to find just one item in a list, it is faster and simpler to just use linear search. Similarly, if our list keeps changing and we have to keep re-sorting it, performance will be compromised as well.

Pre-sorting for binary search works best when we want to perform many searches on the same list. Sorting it once incurs $O(n \log n)$ time at first, but all future searches on the list become very efficient.