Introduction

This module talks about sorting and searching algorithms. Five sorting algorithms, i.e., Bubble-Sort, Selection-Sort, Insertion-Sort, Merge-Sort and Quick-Sort and their performances will be discussed.

Objectives

- To study several sorting and searching algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To learn how to estimate and compare the performance of algorithms
- To learn how to measure the running time of a program

Bubble-Sort

(This is covered by prerequisite course)

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an abysmal O(n^2).

The algorithm for Bubble Sorting is as follows:

```java
void bubbleSort(int[] numbers)
{
    int i, j, temp;
    for (i = (numbers.length - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (numbers[j-1] > numbers[j])
            {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
```
Selection-Sort

The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

The algorithm for Selection Sorting is as follows:

```java
void selectionSort(int[] numbers)
{
    int i, j;
    int min, temp;
    for (i = 0; i < number.length; i++)
    {
        min = i;
        for (j = i+1; j < number.length; j++)
        {
            if (numbers[j] < numbers[min])
                min = j;
        }
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

Insertion-Sort

Insertion sort is a simple sorting algorithm. It is a comparison sort in which the sorted array (or list) is built one entry at a time. The insertion sort has a complexity of $O(n^2)$.

The algorithm for Insertion-Sort is as follows:

```java
void sort(int[] a)
{
    for (int i = 1; i < a.length; i++)
    {
        int next = a[i];
        // Move all larger elements up
        int j = i;
        while (j > 0 && a[j - 1] > next)
        {
            a[j] = a[j - 1];
            j--;
        }
        // Insert the element
        a[j] = next;
    }
}
```
Merge-Sort

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n \log n)$.

Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. The below algorithm merges the arrays in-place, so only two arrays are required. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

```c
void mergeSort(int numbers[], int temp[], int array_size)
{
    m_sort(numbers, temp, 0, array_size - 1);
}

void m_sort(int numbers[], int temp[], int left, int right)
{
    int mid;
    if (right > left)
    {
        mid = (right + left) / 2;
        m_sort(numbers, temp, left, mid);
        m_sort(numbers, temp, mid+1, right);
        merge(numbers, temp, left, mid+1, right);
    }
}

void merge(int numbers[], int temp[], int left, int mid, int right)
{
    int i, left_end, num_elements, tmp_pos;
    left_end = mid - 1;
    tmp_pos = left;
    num_elements = right - left + 1;
    while ((left <= left_end) && (mid <= right))
    {
        if (numbers[left] <= numbers[mid])
        {
            temp[tmp_pos] = numbers[left];
            tmp_pos = tmp_pos + 1;
            left = left + 1;
        }
        else
        {
            temp[tmp_pos] = numbers[mid];
            tmp_pos = tmp_pos + 1;
            mid = mid + 1;
        }
    }
    while (left <= left_end)
    {
        temp[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1;
    }
    while (mid <= right)
    {
        temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;
    }
    for (i=0; i <= num_elements; i++)
```
Quick-Sort
(See advanced topic 19.3)
Quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps (which closely resemble the merge sort):

1. If there is one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

Efficiency analysis

**Bubble**
Pros: Simplicity and ease of implementation.
Cons: Horribly inefficient.

**Selection**
Pros: Simple and easy to implement.
Cons: Inefficient for large lists

**Merge**
Pros: Faster
Cons: At least twice the memory requirements of the other sorts; recursive.
Searching

Data is sorted so that it is faster and easier to find the piece of information of interest. For example, telephone directories have names arranged in alphabetical order. Why? So that you can locate a telephone number easily. If the names were not arranged in order, you would have to start at the beginning and read through until you found the name of the person you want to contact. On average you’d finish up having to read half the book before you found the name you were after.

Linear search

Consider an array with N entries in random order. On average you’d have to examine N/2 entries to find the one you want. The process of starting at the beginning and reading each item until the required item is found is known as Linear Searching.

Binary search

So how do you use a telephone directory? Human beings are a bit fuzzy in the way they do things. So we will describe a standardized version of what you actually do.

Take the directory and open it in the middle. The name you’re after is either in the half above or the half below the middle. Open that half in the middle. The name you want is now in the upper or lower half of that half; keep on halving the size of the directory until you find what you’re looking for.

As an illustration, suppose you are looking for the letter ‘s’. Got to the middle.

```
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
```

Now ‘s’ is below the middle, so go to the middle of the lower half

```
| n | o | p | q | r | s | t | u | v | w | x | y | z |
```

‘s’ is above the middle, so go to the middle of the upper half.

```
| n | o | p | q | r | s |
```

‘s’ is below, so

```
| q | r | s |
```

‘s’ is below, so

```
| s |
```
And we have our result. It’s taken five goes to find what we were looking for. If you try this process for other letters, you’ll find that the most goes it takes to find the letter is five. Some letters you’ll find quicker, e.g. ‘m’ would be found in one go, but no letter takes more than five goes.

The algorithm described above is known as the Binary Search algorithm.

```java
public class BinarySearch {
  // instance variables
  private int first, middle, last;
  private int[] list;

  /**
   * The main search method.
   * @param list - the list to be sorted
   * @param searchTarget - the value to be searched for
   * @return middle or 0 - the location of the requested element is returned
   * if it exists in the list, otherwise 0 is returned
   */
  public int search(int[] list, int searchTarget) {
    last = list.length - 1;
    first = 0;
    // while there are still elements to search through
    while (first <= last) {
      middle = (first + last) / 2;
      // if current middle value is the search target
      if (list[middle] == searchTarget) {
        return middle;
      } else if (list[middle] < searchTarget) {
        first = middle + 1;
      } else {
        last = middle - 1;
      }
    }
    // return 0 if search target not found
    return 0;
  }
}
```

**Efficiency of the binary search**

When we are assessing the efficiency of a searching algorithm, we examine the number of attempts it takes to locate an element.

Let’s look at some specific cases:

- If there was only one element in the array, we’d find it in one attempt.
- If there were two elements in the array, we’d find it in either one or two attempts. The maximum number of attempts is two.
- If there were three elements in the array, we’d find it in a maximum of two attempts. (Test your understanding of the binary search algorithm by proving this to yourself).
- If there were seven elements in the list, we’d need a maximum of three attempts. After the first attempt, we’d be using an array of length three and we know that a maximum of attempts are needed to locate a value in a three element array. So the maximum number of attempts is 1+2 or 3.
These types of analyses can be continued and the table below summarises the results:

<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Maximum number of trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 or 3</td>
<td>2</td>
</tr>
<tr>
<td>4, 5, 6, or 7</td>
<td>3</td>
</tr>
<tr>
<td>8 … 15</td>
<td>4</td>
</tr>
<tr>
<td>16 … 31</td>
<td>5</td>
</tr>
</tbody>
</table>

A formula for determining the maximum number of trials is given below. (No attempt is made to develop this formula)

If there are N elements in an array, the maximum number of attempts, m, is given by the equation:

\[ 2^m > N \]

Here are some values for N and m:

<table>
<thead>
<tr>
<th>Number of elements (N)</th>
<th>Maximum number of trials (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>1000000</td>
<td>20</td>
</tr>
<tr>
<td>1000000000</td>
<td>30</td>
</tr>
</tbody>
</table>

As you can see, the binary search performs much better than the linear search, particularly for large arrays.

**Sorting Real Data**

Sorting objects are frequently needed in the real application. For example, Employee is a class to hold employees. First name, last name and department are instance fields to the employee. If you need to sort employees by first name, last name or department, you need to call class methods to get the values and to compare the values between each other.

**Reading**

Text book:

Chapter 14: Sorting and Searching (whole chapter)
Review questions

Review exercises:

Page 659- 661: Exercise R14.3, R14.8, R14.9

Programming exercises:


References

Cay Horstmann, CH 2007, Big Java, 3rd Edition

Lab session

1) P14.2. Modify the selection sort algorithm to sort an array of Coins by their value.

```java
/**
   * This class sorts an array of coins, using the selection sort algorithm.
   */
public class SelectionSorter
{
  /**
   * Constructs a selection sorter.
   * @param anArray the array to sort.
   */
  public SelectionSorter(Coin[] anArray)
  {
    a = anArray;
  }

  /**
   * Finds the smallest coin in an array range.
   * @param from the first position in a to compare
   * @return the position of the smallest coin in the
   * range a[from] . . . a[a.length - 1]
   */
  public int minimumPosition(int from)
  {
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
    {
      //fill your stuff here ....
      }
    return minPos;
  }

  /**
   * Sorts an array.
   */
  public void sort()
  {
    for (int i = 0; i < a.length - 1; i++)
    {
```
int minPos = minimumPosition(i);
if (minPos != i)
    swap(minPos, i);
}
/**
 * Swaps two entries of the array.
 * @param i the first position to swap
 * @param j the second position to swap
 */
public void swap(int i, int j)
{
    Coin temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

2) The Sorter class is a modified Selection-Sort algorithm for sorting objects according to their different attributes (instance fields). Study this class and make sure you understand it.

public class Sorter
{
    Employee[] employeeList = new Employee[10];

    public Sorter (Employee[] list)
    {
        employeeList = list;
    }

    public void sort(int by)
    {
        for (int i=0; i<employeeList.length; i++)
        {
            int minPos = minimumPosition(i, by);
            swap(minPos, i);
        }
        display();
    }

    private int minimumPosition(int from, int by)
    {
        int minPos = from;
        for (int i=from+1; i<employeeList.length; i++)
        {
            switch (by)
            {
            case 0:
                if (employeeList[i].getName().compareTo(employeeList[minPos].getName()) < 0)
                {
                    minPos = i;
                }
                break;
            case 1:
                if (employeeList[i].getSalary() < employeeList[minPos].getSalary())
                {
                    minPos = i;
                }
                break;
            default: break;
            }
        }
        return minPos;
    }

    private void swap(int i, int j)
    {
        Employee temp = employeeList[i];
        ...
employeeList[i] = employeeList[j];
employeeList[j] = temp;
}

public void display()
{
    for (int i=0; i<employeeList.length; i++)
    {
        System.out.println(employeeList[i].toString());
    }
}

3) Write an Employee class which contains two instance fields: a String data type for “name” and a double data type for “salary”.

4) Write a Tester class which contain a main method to create ten employees, put these ten employee objects into an array, sort employees based on name and salary by using the Sorter and Employee classes.

5) Supply a class Person that implements the Comparable interface. Compare persons by their names. Ask the user to input 10 names and generate 10 Person objects. Using the compareTo method, determine the first and last person (in dictionary) among them and print them.

```java
import java.util.Scanner;
import java.util.Arrays;

/**
 * This class tests the Person class.
 */
public class PersonDemo
{
    public static void main(String[] args)
    {
        int count = 0;
        Scanner in = new Scanner(System.in);

        boolean done = false;
        Person first = null;
        Person last = null;
        while (!done)
        {
            System.out.println("Please enter the person's name or a blank line to quit");
            String name = in.nextLine();
            if (name.equals(""))
                done = true;
            else
            {
                Person p = new Person(name);
                if (first == null) first = last = p;
                else
                {
                    if (first.compareTo(p) > 0) first = p;
                    if (last.compareTo(p) < 0) last = p;
                }
            }
        }
        System.out.println("First: "+ first);
        System.out.println("Last: "+ last);
    }
```
/**
 * A person.
 */
public class Person implements Comparable {
    /**
     * Construct a Person object.
     * @param aName the name of the person
     */
    public Person(String aName) {
        name = aName;
    }
    public int compareTo(Object otherObject) {
        // fill in blank here
    }
    /**
     * Gets the name of the person.
     * @return the name of the person
     */
    public String getName() {
        return name;
    }
    public String toString() {
        return "Person\[name=\" + name + \\"]\";
    }
    private String name;
}

6) Random generate an array containing 20 integers between 0 to 100. Sorting this array by using Array.sort() method. The ArrayUtil class is provided below.

import java.util.Random;

public class ArrayUtil {
    public static int[] randomIntArray(int length, int n) {
        int[] a = new int[length];
        for (int i = 0; i < a.length; i++)
            a[i] = generator.nextInt(n);
        return a;
    }
    private static Random generator = new Random();
}
7) Practice Collections.sort() method by evaluating the example (CitySorter) in the lecture slides.

```java
import java.util.Collections;
import java.util.ArrayList;

public class CitySorter
{
    public static void main(String[] args)
    {
        ArrayList<City> cities = new ArrayList<City>();
        cities.add(new City("Rockhampton", "QLD");
        cities.add(new City("Orange", "NSW");
        cities.add(new City("Orange", "SA");
        cities.add(new City("Perth", "WA");
        Collections.sort(cities);
        for (City c : cities)
            System.out.println(c);
    }
}

public class City implements Comparable<City>
{
    public City(String name, String state)
    {
        this.name = name;
        this.state = state;
    }

    public String getName()
    {
        return name;
    }

    public String getState()
    {
        return state;
    }

    public int compareTo(City other)
    {
        if (name.compareTo(other.name) == 0)
            return state.compareTo(other.state);
        return name.compareTo(other.name);
    }

    public String toString()
    {
        return "City[" + name + "," + state + "]";
    }

    private String name;
    private String state;
}
```

8) Work on your assignment.