ALGORITHMS FOR SORTING

CITS1001
Listen to the sound of sorting

• Various algorithms

• [http://www.youtube.com/watch?v=t8g-iYGHpEA](http://www.youtube.com/watch?v=t8g-iYGHpEA)

• Quicksort

• [http://www.youtube.com/watch?v=m1PS8lR6Td0](http://www.youtube.com/watch?v=m1PS8lR6Td0)

• Or Google for sound of sorting
Scope of this lecture

- Linear Search
- Sorting and Bubble Sort

References:
- Wirth, *Algorithms + Data Structures = Programs*, Chapter 2

This lecture is based on powerpoint slides by Gordon Royle UWA
SEARCHING
Searching

- **Searching** refers to the process of finding data items that match certain criteria; we may either want a yes/no answer to a question or additional details as well
  - Find out *whether* any students got a mark of 49
  - Find out *which* students got a mark of 49

- The simplest searching technique is called *linear search*, which simply involves looking through each element in turn until we find one that matches the criteria
Our favourite Student class

```java
public class Student {
    private String studentID;
    private int mark;

    public Student(String studentID, int mark) {
        this.studentID = studentID;
        this.mark = mark;
    }

    public String getStudentID() {
        return studentID;
    }

    public int getMark() {
        return mark;
    }
}
```

A skeleton version of a possible Student class in a student records system
Collections of Students

- We consider a class list being stored as an ArrayList
- The question that we will consider is how to retrieve the data for a student with a given student number

- So, we will write a method with the following signature

```java
public Student findStudent(ArrayList<Student> classlist, String id)
```

The method returns a (reference to a) Student object
The arraylist of students is a parameter
The student ID we want is the other parameter
public Student findStudent( ArrayList<Student> classlist, String id) {
    for (Student s : classlist) {
        String sid = s.getStudentID();
        if (sid.equals(id)) {
            return s;
        }
    }
    return null;
}
Comments

• If the arraylist *does* contain the desired value, then the method returns *true* as soon as it is found
• If the arraylist *does not* contain the desired value, then the method will return *false* after checking every element of the arraylist without success
• We have shown the general situation of finding an *object* in a collection of objects
Performance of linear search

• How fast does linear search work on an array of \( n \) items?
• We can identify three situations
  • Best case, when the input is the most convenient possible
  • Worst case, when the input is the least convenient possible
  • Average case, averaged over all the inputs
• In the \textit{best} case, linear search finds the item at the first position of the array, so it needs 1 comparison
• In the \textit{worst} case, linear search does not find the item and so must perform \( n \) comparisons unsuccessfully
• To calculate the \textit{average} case performance we would need some problem-specific assumptions about the input data
Linear search is too slow

- If we have very large amounts of data, then linear search is not feasible
- For example, we can view the telephone directory as a very large array of objects, with each object consisting of a name and a number
  - If you are asked to find out which person has phone number 9388 6105 then how long would it take you to do this by linear search?
- However, if I ask you to find out the phone number of a specific person, then you can do it much, much faster
  - How do you do it?
  - How can we program a computer to do this?
Sorted Collections

• The reason that is quick, while is slow is because the “collection” (i.e. phone book) is \textit{sorted} into alphabetical order, and somehow this allows us to find an entry much more quickly (we will see why later)

• Most useful databases are \textit{sorted} – dictionaries, indices etc.
Sorting

• Before we examine how to efficiently *search* in a sorted collection, we must consider how to *sort* the collection.

• We will again start with the “plain vanilla” example – sorting an array of integers into increasing order.

<table>
<thead>
<tr>
<th>before</th>
<th>6</th>
<th>8</th>
<th>1</th>
<th>15</th>
<th>12</th>
<th>2</th>
<th>7</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>after</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

• Later we will extend this to sorting arrays of objects according to various other criteria (alphabetical etc).
The basic set up

- We will implement a number of sorting methods, all of which operate on an array of integers
- We develop these as a utility class called Sorter – a class with no instance variables, but just class (static) methods (e.g. Math)
- Each method will have a similar signature, where the only thing that will vary is the name of the sorting technique

```
public static void nameSort(int[] a)
```
- Each method receives a (reference to an) array as a parameter and will sort that array in place
BUBBLE SORT
bubbleSort

• The idea behind *bubbleSort* is to systematically compare pairs of elements, exchanging them if they are out of order
• If the array contains *n* elements, then we view the algorithm as consisting of *n* “passes”
• In the first pass we compare

  • Element 0 with element 1, exchange if necessary
  • Element 1 with element 2, exchange if necessary
  • ...
  • Element *n*−2 with element *n*−1, exchange if necessary
The first pass

• After the first pass, the largest element will be at the end

```
6 8 1 15 12 2 7 4

6 1 8 15 12 2 7 4

6 1 8 12 15 2 7 4

6 1 8 12 2 15 7 4

6 1 8 12 2 7 4 15
```
The second pass

- The second pass doesn’t need to make the last comparison
The third pass

- The third pass can omit the last two comparisons
The fourth pass

- The fourth pass is even shorter
The fifth and sixth passes

1 2 6 4 7 8 12 15

1 2 6 4 7 8 12 15

1 2 6 4 7 8 12 15

1 2 4 6 7 8 12 15

1 2 4 6 7 8 12 15

1 2 4 6 7 8 12 15
Why does it work?

• We need to have some argument or “proof” that this works
• We claim that

_after the i-th pass, the last i elements in the array are in their correct positions_

• This is true after the first pass, because the largest element in the array is encountered at some stage and then “swapped all the way to the end” of the array
• The same argument – applied to the remainder of the array – shows that the second pass puts the 2\textsuperscript{nd} largest element into place; repeating this argument \(i\) times gives the result
Coding Bubblesort

```java
public static void bubbleSort(int[] a) {
    for (int pass=1; pass<a.length; pass++) {
        for (int j=0; j<a.length-pass; j++) {
            if (a[j] > a[j+1]) {
                int swap = a[j+1];
                a[j+1] = a[j];
                a[j] = swap;
            }
        }
    }
}
```
Sorting Students

public static void bubbleSort(Student[] a) {

    for (int pass=1; pass<a.length; pass++) {
        for (int j=0; j<a.length-pass; j++) {
            if (/* a[j] and a[j+1] out of order */) {
                Student swap = a[j+1];
                a[j+1] = a[j];
                a[j] = swap;
            }
        }
    }
}

Almost identical code except that we need to get the right boolean condition to check when two students are in the “wrong order”
What order do we want?

- The precise form of the statement depends on whether we want to sort students:
  - *Alphabetically* according to their `studentId`
  - *Numerically* according to their `mark`
- In addition, the desired sort could be *ascending* (lower values first) or *descending* (lower values last)
- Suppose that we want to sort the students into normal (ascending) alphabetical order by `studentId`
For alphabetic order

- The comparison between the two Student objects $a[j]$ and $a[j+1]$ first needs to obtain the two ids to compare, so it will involve the two Strings
  - String $s1 = a[j].getStudentID()$;
  - String $s2 = a[j+1].getStudentID()$;
- To compare two Strings we use the `compareTo` method

```java
if (s1.compareTo(s2) > 0) {
    // Swap the two Students
}
```
SELECTION SORT AND INSERTION SORT
When sorting $n$ items, *Selection Sort* has the following properties:

- The procedure has $n$ stages
- Select the smallest element in the array and swap it with the element in position 0.
- Then calculate the position of the smallest element in the array starting from position 1, and swap that element into position 1
- After the $i$’th stage, the first $i$ items are sorted in order
- At the $(i+1)$’st stage, the item originally in position $i+1$ is incorporated into the sorted list and placed in position
**Sorting cards**

- In card games, it is common to pick up the cards as they are dealt and to sort them into order as they arrive
- For example, suppose your first three cards are

![Playing cards](image)

- Next you pick up a 9 of clubs
Inserting a card

• The new card is then *inserted* into the correct position
We can develop this notion into a sorting algorithm called *Insertion Sort*

When sorting $n$ items, *Insertion Sort* has the following properties:

- The procedure has $n$ stages
- After the $i$’th stage, the first $i$ items are sorted in order
- At the ($i+1$)’st stage, the item originally in position $i+1$ is incorporated into the sorted list and placed in position
Example

- Initial array

\[
\begin{array}{cccccccc}
6 & 8 & 1 & 15 & 12 & 2 & 7 & 4 \\
\end{array}
\]

- Stage 1: Move the first element into position (do nothing)

\[
\begin{array}{cccccccc}
6 & 8 & 1 & 15 & 12 & 2 & 7 & 4 \\
\end{array}
\]

- Stage 2: Examine the second element and insert it into position (again do nothing)

\[
\begin{array}{cccccccc}
6 & 8 & 1 & 15 & 12 & 2 & 7 & 4 \\
\end{array}
\]
Stage 3

- This element is out of position so will have to be *inserted*
Stages 4 and 5

Stage 4

Stage 5
Stage 6

1 6 8 12 15 2 7 4

1 6 8 12 15 → 7 4

1 6 8 12 → 15 7 4

1 6 8 → 12 15 7 4

1 6 → 8 12 15 7 4

1 → 6 8 12 15 7 4

1 2 6 8 12 15 7 4
# Stage 7

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>15</th>
<th>7</th>
<th>4</th>
</tr>
</thead>
</table>

1. **Stage 7**

2. **Stage 6**

3. **Stage 5**

4. **Stage 4**

5. **Stage 3**

6. **Stage 2**

7. **Stage 1**
Final stage
Code for InsertionSort

```java
public static void insertionSort(int[] a) {

    for (int pass=1;pass<a.length;pass++) {
        int tmp = a[pass];
        int pos = pass-1;

        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }

        a[pos+1] = tmp;
    }
}
```
public static void insertionSort(int[] a) {

    for (int pass=1; pass<a.length; pass++) {

        int tmp = a[pass];
        int pos = pass - 1;

        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }

        a[pos+1] = tmp;

    }
}
public static void insertionSort(int[] a) {
    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass-1;
        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }
}

The variable tmp stores the value that is to be inserted; the variable pos will eventually contain the position where it should be inserted.
Code dissection

```java
public static void insertionSort(int[] a) {
    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass-1;
        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }
}
```

This code does the work of shifting each element in turn one space along if it is bigger than the value to be inserted. We also need to ensure that we don’t fall off the left-hand end of the array!
Code dissection

public static void insertionSort(int[] a) {

    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass - 1;
        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }

    The while loop finishes when we have found the correct position for the element, and so it is now inserted into this position.
Efficiency experiment

• Is there any difference between the performance of these sorting algorithms?
• Which one(s) are more efficient?
• Why?

• Experiment: use the provided Java Sorter class and its test harness SorterTest to estimate the execution time of each algorithm for sorting a large, unsorted array.

• Graph your results on the next page
Performance Comparison
A note on the accuracy of this test

• WARNING: testing the execution time of Java code this way is not completely accurate, in that you will not always get the same results
• Activities such as garbage collection and other aspects of the Java Virtual Machine may affect run time
• I have used random data and multiple runs to "average out" unrepresentative runs
• For more on the efficiency of Java programs, see for ex,
• Highly recommended: PhD Thesis (CORE winner)
  Garbage Collection and the Case for High-level Low-level Programming
Summary

• Searching for an element in a collection is a common requirement
• Searching can be more efficient if the collection is sorted
• We study sorting algorithms because they provide good examples of many of the features that affect the run-time of program code.
• Checking the efficiency of your own code? Consider:
  • Number of loops and depth of nesting
  • Number of comparison operations
  • Number of swap (or similar) operations