ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

ALGORITHMS AND COMPLEXITY

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ucl.ac.uk

Algorithms

- Any computing problem can be solved by executing a series of actions in a specific order
- An algorithm is the process of solving a problem in terms of:
  - The actions to execute
  - The order in which these actions are executed
- We are going to examine in detail search and sort algorithms
- We are also going to examine and understad the time complexity of an algorithm as a function of the problem size

Search Algorithms

- Looking up a phone number electronically, searching the Web or looking-up a word in an online dictionary all involve data searching
- A linear search algorithm searches each element sequentially and, as such, it is relatively inefficient
- A binary search algorithm requires the data to be sorted and it is more efficient – it works by splitting the data set in the middle in an iterative fashion
- We are going to examine linear and binary search algorithms searching through an array of integers

Integer Array

```
public class IntegerArray {

    protected int[] data;

    public IntegerArray (int[] newData) {
        data = newData;
    }

    @Override
    public String toString () {
        String output = new String();
        for (int i = 0; i < data.length; i++) {
            output += data[i];
            if (i != (data.length – 1))   // add a space but not after the last element
                output += " ";
        }
        return output;
    }

} // end class IntegerArray
```

Linear Search Algorithm

```
public class LinearArray extends IntegerArray {

    public LinearArray (int[] data) {
        super(data);
    }

    public int linearSearch (int searchKey) {
        // look through data sequentially in a for loop
        for (int i = 0; i < data.length; i++)
            if (data[i] == searchKey)
                return i;
        return -1;       // searchKey not found
    }

} // end class LinearArray
```

Linear Search Test

```
import java.util.Random;
public class LinearArrayTest {
    public static void main (String args[]) {
        if (args.length != 2) {
            System.err.println(“usage: LinearSearchTest arraySize searchKey");
            System.exit(1);
        }
        // here we need exception handling but we omit it for brevity
        int size = Integer.parseInt(args[0]);
        int key  = Integer.parseInt(args[1]);
        if (key < 1 || key > size) { System.err.println(“key out of range”); System.exit(1); }

        int[] data = new int[size];  Random randGen = new Random();
        for (int i = 0; i < size; i++)
            data[i] = randGen.nextInt(size)+1; // random numbers 1..size

        LinearArray searchArray = new LinearArray(data);
        int position = searchArray.linearSearch(key);

        System.out.println(“linear array is:\n” + searchArray);
        System.out.println(“key ” + key + “ was found in position ” + position);
    }
} // end class LinearArrayTest
```

Binary Search Algorithm

```
import java.util.Arrays;

public class BinaryArray extends IntegerArray {
   public BinaryArray (int[] data) {
      super(data); Arrays.sort(data); // use static sort method of Arrays class
   }
   public int binarySearch (int searchKey) {
      int low      =   0;                          // initial low
      int high     = data.length-1;                // initial high

      while (high >= low) {
         int middle  = (low+high) / 2;            // middle for splitting in two
         if (searchKey == data[middle])           // key exactly in the middle
            return middle;
         else if (searchKey < data[middle])  // key possibly in low half
            high = middle-1;                         // new high
         else // (searchKey > data[middle])  -  key possibly in high half
            low = middle+1;                          // new low
      }
      return -1;  // key not found
   }
} // end class BinaryArray
```

Binary Search Test

```
import java.util.Random;
public class BinaryArrayTest {
    public static void main (String args[]) {
        if (args.length != 2) {
            System.err.println(“usage: BinarySearchTest arraySize searchKey");
            System.exit(1);
        }
        // here we need exception handling but we omit it for brevity
        int size = Integer.parseInt(args[0]);
        int key  = Integer.parseInt(args[1]);
        if (key < 1 || key > size) { System.err.println(“key out of range”); System.exit(1); }

        int[] data = new int[size];  Random randGen = new Random();
        for (int i = 0; i < size; i++)
            data[i] = randGen.nextInt(size)+1; // random numbers 1..size

        BinaryArray searchArray = new BinaryArray(data);
        int position = searchArray.binarySearch(key);

        System.out.println(“binary array is:\n” + searchArray);
        System.out.println(“key ” + key + “ was found in position ” + position);
    }
} // end class BinaryArrayTest
```

Sort Algorithms

- Sorting data is one of the most important computer applications, e.g. banks sort cheques by account number, phone companies sort customer accounts by name, etc.
- A selection sort algorithm finds the smallest element and swaps it with the first, it then finds the second smallest and swaps it with the second, etc.
- A bubble sort algorithm compares adjacent elements and swaps them if they are in the wrong order. It goes through the array continuously until no swap is done
  - Bigger elements "bubble" their way to the top, hence "bubble sort"
- Other types of sort algorithms that we do *not* examine here are insertion sort, which is of similar complexity to selection & bubble sort but better, and merge and quick sort, which are more complex but also more efficient

Selection Sort

```
public class Sort extends IntegerArray {
    public Sort (int[] data) {
        super(data);
    }

    public void selectionSort () {
        for (int i = 0; i < data.length-1; i++) {
            int smallest = i; // initialised to first index of remaining array

            // loop to find smallest element in [i+1 .. data.length]
            for (int j = i+1; j < data.length; j++)
                if (data[j] < data[smallest])
                    smallest = j;

            if (smallest != i)  {  // found a smaller element in the rest of the array
                int tmp = data[i];
                data[i] = data[smallest];  // swap data[i] with data[smallest]
                data[smallest] = tmp;
            }
        }
    }
    // continues overleaf
```

Bubble Sort

```
// class Sort continued

public void bubbleSort () {
    boolean swapped = true;

    while (swapped) {
        swapped = false;
        for (int i = 0; i < data.length-1; i++)
            if (data[i] > data[i+1]) {
                int tmp = data[i];
                data[i] = data[i+1];
                data[i+1] = tmp;
                swapped = true;
            }
    } // end while
}

} // end class Sort
```

Sort Test

```
import java.util.Random;

public class SortTest {
    public static void main (String args[]) {
        if (args.length != 1) {
            System.err.println(“usage: SortTest arraySize");
            System.exit(1);
        }
        // here we need exception handling but we omit it for brevity
        int size = Integer.parseInt(args[0]);

        int[] data = new int[size];
        Random randGen = new Random();
        for (int i = 0; i < size; i++)
            data[i] = randGen.nextInt(2*size)+1; // random numbers 1..2*size

        Sort sortArray = new Sort(data);
        System.out.println(“unsorted array is:\n” + sortArray);
        sortArray.selectionSort();  // could also use bubbleSort
        System.out.println(“sorted array is:\n” + sortArray);
    }
} // end class SortTest
```

Complexity

- There typically exist different algorithms to solve the same problem, e.g. linear and binary search, the key difference being the effort and time required
- We describe algorithm complexity using the Big O Notation, which indicates the worst case run time the algorithm requires
  - i.e. how hard the algorithm will have to work to find the solution
- Algorithm complexity in the big O notation is expressed as a function of the problem size n, e.g. the size of the array to search or sort
  - For example, $O(n)$, $O(n^2)$, etc.

Constant Complexity

- Let's consider an algorithm that checks to see if the first is equal to the last element of an array
  - This algorithm always requires 1 comparison (data[0] == data[data.length-1]), independently of the array size
- This algorithm is said to have a constant run time which is represented in big O notation as *O(1)*
- An *O(1)* algorithm does not require only one comparison but a constant number of comparisons, i.e. a number independent of the problem size n
  - For example, an algorithm that checks to see if the first element is equal to the last four elements of an array requires 4 comparisons but it is still of complexity *O(1)*

Linear Complexity

- Let's now consider an algorithm that checks to see if the first element is equal to any other array element
  - This algorithm requires at most n-1 comparisons
- As n grows larger, the n part dominates the n-1 expression, with -1 becoming inconsequential
- In fact, Big O is designed to highlight the dominant terms and ignore unimportant terms as n grows
- As such, an algorithm that requires at most n-1 comparisons is said to be *O(n)*
- This is referred to as linear run time and pronounced "of the order of n" or simply "order n"

Linear Search Complexity

- A linear search requires at most n comparisons, so it is of *O(n)* complexity
  - At most n for the case that the search key is not present or it is located in the very last element
- If the search key is closer to the beginning of the array, linear search can be much faster in terms of searches/time required but the worst case is *O(n)*
- This is why we look for algorithms that can possibly perform better, hence the binary search algorithm

Binary Search Complexity

- In binary search, we split the array in the middle and check in which of the two parts the search key is, then we continue like this until we find it (or not)
  - For an array of 1024 elements (2^{10}), we need to split it 10 times until it we get 1: 512, 256, 128, 64, 32, 16, 8, 4, 2, 1
  - We perform a constant number of comparisons per split, which is the same in terms of complexity as 1 comparison
- As such, the max number of comparisons is the first power of 2 greater than the array size n, which is represented as $\log_2 n$
- Given that all logarithms grow in roughly the same rate, the base 2 can be omitted, resulting in complexity of O(logn) or logarithmic run time
  - This is a tremendous improvement, e.g. if n is 2^{20}=1048576 binary search needs at maximum 20 comparison cycles

Quadratic Complexity

- Let's now consider an algorithm that checks to see if any element is duplicated in the array
  - The first element must be compared with the rest n-1 elements, the second with the rest n-2, etc.
  - The total number of comparisons is (n-1)+(n-2)+...+2+1 = $(n^2-n)/2$. It is reminded that 1+2+...+(n-1)+n = $(n^2+n)/2$
- As n increases, the n^2 part dominates the expression with n becoming inconsequential (which this also the case for the denominator 2)
- As such, the algorithm requires at most n^2 comparisons and is said to be $O(n^2)$
- This is referred to as quadratic run time and pronounced "order n squared"

Selection Sort Complexity

- The selection sort algorithm contains two nested for loops
  - The outer loop controls the iterations of the inner loop over the remaining array elements
  - The inner loop does n-1 comparisons in the first iteration, n-2 comparisons in the second, then n-3, …, 2, 1 (with potentially one element swap at the end of every loop)
  - The total comparison cycles are 1+2+…+(n-2)+(n-1) = $(n^2-n)/2$ (see previous slide)
- As such the selection sort algorithm is $O((n^2-n)/2)$ which is effectively $O(n^2)$ as explained previously, i.e. of quadratic complexity

Bubble Sort Complexity

- The bubble sort algorithm contains one inner loop which is run within an outer while(forever) loop until no item is swapped
  - The worst case is that the list is “inversely” sorted, in which case the inner loop executes n times
  - The inner loop does n-1 comparisons in the first pass, n-1 comparisons in the second, n-1 in the third, …, and n-1 for the n-th pass
  - The total comparison cycles will be $n*(n-1) = n^2-n$
- As such, the bubble sort algorithm is $O(n^2-n)$ hence also $O(n^2)$

Comparison of Bubble & Selection Sort Complexity

- Bubble sort is though more inefficient than selection sort:
  - In the worst case it makes more comparisons, n^2 vs $(n^2-n)/2$
  - Also selection sort swaps once per inner loop while bubble sort swaps potentially many times per inner loop
- On the other hand, if an array is already sorted then bubble sort complexity drops to *O(n)* as the inner loop will execute only once (n comparisons) – no swap required
  - Selection sort complexity will still remain $O(n^2)$ as the inner loop will still have to do $(n^2-n)/2$ comparisons
  - Bubble sort works well of arrays that are "nearly sorted"
- This means that the complexity is not the only measure, algorithms of the same complexity may perform slightly differently
  - Insertion better than selection better than bubble sort

Algorithms & Complexity

- In summary, the complexity of the algorithms we examined or mentioned is as follows:

Algorithm	Complexity
Linear Search	O(n)
Binary Search	O(logn)
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Merge Sort	O(nlogn)
Quick Sort	O(nlogn)

Summary

- Algorithms involve detailed steps or actions in the process of solving a problem
- Different algorithms may solve the same problem with different degrees of efficiency
- We measure algorithm complexity as a function of the problem size through the Big O Notation
- Linear search is of linear complexity while binary search is of logarithmic complexity
- Selection and bubble sort are of quadratic complexity while other sort algorithms, e.g. merge sort / quick sort, are more efficient - O(nlogn) - but also more complex