Sorting and Searching

Basic sorting and searching algorithms

Lecturer: Guoqing Lu

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Department of Computing and Mathematics
http://www.setu.ie/

Main concepts to be covered in this section

- Arrays
  - Swapping Values
- Sorting
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Searching
  - Linear Search (unsorted list)
  - Binary Search (sorted list)

Arrays – Swapping Values

- Before we start discussing sorting, we will look at swapping two values in an array.

- This swap algorithm is needed during searching.

Arrays – Swapping Values

- An array of integer values has been created called intArray:

```
private int[] intArray = new int[20];
```

- **Exercise 1:** write a few lines of code that will swap the contents of index position 2 with index position 3.

Arrays – Swapping Values

Solution 1:

```
int temp = intArray[2];
intArray[2] = intArray[3];
intArray[3] = temp;
```

Arrays – Swapping Values

- Note if we want to make this code more usable, we will write a method that takes as parameters the name of the array, and the positions of the values you wish to swap e.g.

Arrays – Swapping Values

```
private void swap(int[] intArray,int i,int j)
{

    int temp = intArray[i];
    intArray[i] = intArray[j];
    intArray[j] = temp;


}
```

Sorting

- The human brain can very quickly sort a collection e.g. it would be straightforward to sort a group of people in height order.
- For a computer to carry out this process is more difficult.

Sorting

- A computer is quite limited in the data it can handle simultaneously and is restricted to:
    1. Comparing two items
    2. Swapping the items or copying one item.
- In the sorting algorithms we will investigate, we will see the above two steps repeatedly used.

Sorting

- In this introduction to sorting algorithms we will compare three basic algorithms
  1. Bubble sort
  2. Selection sort
  3. Insertion sort

.

What is a “Sorted List”?

- We can define a sorted list (which we will implement as an integer array to start with) as a list in which each element is in its correct position, its correct position being defined as (if we see the list as a horizontal list and assuming ascending order) :

  – *All values to the left of the element are less than or equal to it and all values to the right of the element are greater than or equal to it.*

Sorting – Bubble Sort

- In the bubble sort, a collection of elements is rearranged by repeatedly comparing neighbouring elements and either swapping the elements or leaving them unchanged depending on their relative sizes (for numbers).

Sorting – Bubble Sort

- At the end of the first cycle through the array, 1 item will be in its correct position (the largest, or smallest, bubbles up to one end of the array).
- For example in an array of integers arranged in increasing integer size, the largest integer should be in the highest index position in the array.

Sorting – Bubble Sort

- In the next pass through the array, each element is once again compared to its neighbour but there is now no need to compare the final element in the array because this is already known to be the largest.

Sorting – Bubble Sort

Exercise 2:

- Write a method called bubbleSort()
  - which takes an array of integers as its argument
  - and prints out the elements of a sorted array with integer size increasing with index position.

Solution 2:

```
public void bubbleSort(int[] intArray, int size)
{
   for(int j = size - 1; j >= 0; j--)
   {
      for(int i = 0; i< j; i++)
      {
         if (intArray [i] > intArray [i+1])
            swap(intArray, i, i+1);
      }
   }
   print(intArray);
}
```

Sorting – Bubble Sort

- ***What if the Array was already sorted?***
- ***This solution does not take into consideration that the list could already be sorted.***

- The following solution does by checking is there a swap performed during a pass. If there was no swap on a previous run, it means the list is already sorted and we can exit or stop sorting.

Solution 2:

```
public int[] bubbleSort(int[] array)
{
  boolean swappedOnPrevRun = true;
  while(swappedOnPrevRun)
  {
    swappedOnPrevRun = false;
    for (int i = 0; i < array.length - 1;i++)
    {
      if (array[i] > array[i+1])
      {
         swappedOnPrevRun = true;
         swap(array, i, i+1);        }
    }
  }
  return array;
}
```

Efficiency in Sorting?

- The bubble sort algorithm is one of the most inefficient sorting algorithms – however, it could be used where performance isn’t an issue.
- Selection sort is more efficient than Bubble sort.

Selection Sort Algorithm

1. Iterate through all elements of the array and select the largest one (this is stored in a temporary variable).
2. The largest one is then swapped with the element at the end of the array.
3. Then reduce the size of the array by 1 and work through the array to locate the next largest element.
4. Repeat the above steps until the size of your array is 1.

The Selection Sort Algorithm

1. Firstly we need to find the position of the largest element.

   – Exercise: Write a method called ***findPosOfLargest***, that accepts:
     - an array of ints
     - an int called size (depicting the size of the array)

     as parameters. This method returns the position of the largest element in the array of int passed in as the parameter.

Solution:

```
private int findPosOfLargest(int[] intArray, int size)
{
    int largestPosSoFar = 0;

    for (int i = 1; i < size; i++)
    {
      if (intArray[i] > intArray[largestPosSoFar])
                              largestPosSoFar = i;
    }
    return largestPosSoFar;
}
```

The Selection Sort Algorithm

2. Secondly, we need to write the method that reduces the size of the array by 1 and works through the array to locate the next largest element. This method will be repeated until the size of your array is 1.

```
public void selectionSort(int[] intArray)
{
  for (int i = intArray.length; i > 0; i--)
  {
     int posLargest = findPosOfLargest(intArray, i);
     swap(intArray, posLargest, i-1);
  }
}
```

Selection Sort in one method

```
private int[] selectionSortOneMethod(int[] intArray)
{
   for (int i = intArray.length - 1; i >= 0; i--)
   {
      int highestIndex = i;
      for (int j = i; j >= 0; j--)
      {
         if (intArray[j] > intArray[highestIndex])
         {
            highestIndex = j;
         }
      }
      swap(intArray, i, highestIndex);
   }
   return intArray;
}
```

Selection Sort Algorithm

1. Our selection sort algorithm searches for the largest element and places it at the end of the array.
2. It can also be written so that you select the smallest element and place it at the start of the array.
3. Homework – re-write the above code so that it looks for the smallest element and places it at the start of the array!

Insertion Sort

1. This approach essentially splits the array into two sections – a sorted section initially empty and an unsorted section.

2. The basic idea is that the sorted section grows by 1 element with each cycle through the code (the unsorted section must therefore reduce in size by 1).

Insertion Sort contd.

1. Within the sorted section, the elements are in order.

2. We will spilt up the code into two parts, firstly we write a method that inserts an element into an already sorted list.

Insertion Sort contd.

```
public void insertElem(int[] intArray, int size, int newnumber)
{
    int i = size-1;
    while (i >=0 && intArray[i] > newnumber)
    // need to ensure it doesn't fall over the start(i = -1)
    {
       intArray[i+1] = intArray[i];
       i--;
    }
    intArray[i+1] = newnumber;
    size++;
}
```

Insertion Sort contd.

```
public void insertionSort(int[] intArray, int size)
{
     for (int i = 1; i <size; i++)
          insertElem(intArray, i, intArray[i]);
}
```

We then repeatedly call this method to insert the first (leftmost) element of the unsorted list into the ‘so-far’ sorted list.

Linear and binary searching

Searching

- Generally a search method will look something like (ignore access modifier):

```
int search(int[] a, int size, int looking_for)
```

- This method would return the position in the array of the value ‘looking_for’ if it exists and –1 otherwise.

Searching

- Before we decide on how to search through a list we need to know whether or not the list is sorted.
  - Not Sorted → Linear Search
  - Sorted → Binary Search

Linear Search

```
public int linearsearch(int[] intArray, int size, int looking_for)
{
   int i = 0;

   while (i < size)
   {
      if (intArray[i] == looking_for)
         return i; //this terminates the method
      i++;
   }
   return -1;  // we only get to here if we have got to
               // end of list and never found item
}
```

Binary Search

- If the list is sorted we make use of this (same way we search through telephone book)
- The binary search is such a search and is a particularly efficient search.

Binary Search Algorithm

- The overall algorithm works by **always** looking at the **midpoint** of the list.
- If we have found what we are looking for, great.
- If not…

Binary Search Algorithm contd.

- If not...
- …and the value we are looking for is **less** than the midpoint value…
- …then, because the list is sorted, we can ‘throwaway’ all the values greater than the midpoint. So we ‘redraw’ our list to be the left-hand part of the list and repeat the procedure.

Binary Search Algorithm contd.

- If not…
- …and the value we are looking for is **greater** than the midpoint value…
- …then (using the same logic) we ‘redraw’ the list to just include the right-hand part of the list and repeat the procedure.

Binary Search Code

```
public int binarysearch(int[] intArray, int size, int looking_for)
{

  int mid;
  int left =  0;
  int right = size-1;

  while (left <= right)
  {
    mid = (left + right) / 2;
    if (intArray[mid] == looking_for)
       return mid;                          //this terminates the method
    else if (intArray[mid] > looking_for) // element in left hand of list, if is exists.
       right = mid - 1;
    else if (intArray[mid] < looking_for) // element in right hand of list, if is exists.
       left = mid + 1;
    }
    return -1;
}
```