CSC2040 Data Structures and Algorithms, and Programming Languages

Practical 10

Wednesday 31 January 2018

Sorting Algorithms

Create a new empty Project called *Practical10* on the C:\ drive or on your own drive.

1. Testing if an array is sorted

In your main method, create an array of N sorted integers, something like:

```
for(int i = 0; i < N; i++) a[i] = 3 * i + 1;
```

and an array of N unsorted integers, something like:

```
for(int i = 0; i < N; i++) a[i] = rand() % 1000;
```

Now write a function

bool **isSorted** (int* a, int N)

which examines any supplied array, of length N, and returns true if the array is sorted in *ascending* order, and returns false if it isn't.

Test your function for a range of arrays of different length, some sorted and some unsorted.

2. Sorting an array using Insertion Sort

Write a function

void **insertionSort** (int *a, int N)

which will sort the supplied array into ascending order, using the *Insertion Sort* algorithm discussed in lectures. Your function should reuse the original array a for the operation and return the result in a. Be careful about memory management issues.

Test your method first on a small known array, and then on an array of random numbers. Validate your function using the isSorted function above. Then time the operation for several values of N (100, 200, …, 500) and record the times.

3. Sorting an array using QuickSort

Write a function

void **quickSort** (int *a, int first, int last)

which sorts the supplied array, from the first element to the last element, into ascending order, using the *QuickSort* algorithm discussed in lectures. Your function should return the result in the original array a. Again, test your method on an array of random numbers, and validate your function using isSorted. (Note: don't run it on the already-sorted result of insertionSort!)

To check that you understand the algorithm, print out the array inside the function after the partitioning phase, but before the sorting phase (i.e. before the recursive calls to quickSort). Also print out the value of 'M' (the start position of the upper half after partitioning). Check carefully that you understand exactly why each value ended up exactly where it is.

For example, do this exercise on paper. Trace the program from the code (don't run it):

```
int b[] = {41, 8, 33, 16, 14, 56, 50, 5};
```

After 1st partitioning (don't run the program on this array yet! Do it by hand):

b will be: …… M (start of right 'half') will be …….

When sorting the left 'half': after partitioning the left half:

it will be: …… Its M will be …….

When sorting the right 'half' of the original partitioned *b*: after partitioning the right half:

it will be: …… Its M will be …….

Now, run your program to see if your answers are correct. It should output the values of the array segments after each partitioning phase. See if you made any mistakes in your predictions – and learn from them!

Finally, time the operation for N = 100, 200, …, 500 and record the times.