

COMP20012 Tutorial 4: Sorting Algorithms – The questions

1. Sort the sequence 8, 1, 4, 1, 5, 9, 2, 6, 5 by using
 - a) Insertion sort
 - b) Mergesort
 - c) Quicksort, with the middle element as pivot
2. An array contains N numbers and you want to determine whether two of the numbers sum to a given number K . For example, if the input is 8, 4, 1, 6 and K is 10, the answer is yes (4 and 6). A number may be used twice.
 - a) Give an $O(N^2)$ algorithm to solve this problem
 - b) Give an $O(N \log N)$ algorithm to solve this problem. (Hint: first sort the array and then solve the problem in linear time.)
3. Investigate the details of the *Shell* sort algorithm, either from a textbook or by finding a suitable web resource.

Explain to your tutor how this algorithm works, illustrating your answer by sorting the array in question 1, using the increment sequence 5, 3, 1.

COMP20012 Tutorial 4: Sorting Algorithms – Tutors notes

Aim: Considerable time is spent in the lectures and the laboratory exercises on the structure and analysis of searching and sorting algorithms, examining a wide range of algorithms. This sheet is an opportunity for the students to (a) exercise their understanding of these algorithms by ‘hand running’ a few examples, and (b) reading up about another family of such algorithms and their analysis.

Question (2) illustrates the use of algorithmic techniques: (a) preconditioning by sorting and (b) a binary division technique, to devise an efficient algorithm. It is not clear at first sight that such an algorithm exists - it depends on properties of arithmetic, and hence the correctness argument is important (it works for addition, but not some other arithmetic operations).

1. The most important thing here is that the students show that they understand how these algorithms work, particularly the recursive algorithms, rather than simulating a Java VM.

Descriptions of the algorithms can be found in any standard text, or in David’s slides, which can be found at <http://www.cs.man.ac.uk/~graham/COMP20012>. Code for the algorithms can be found at `/opt/info/courses/COMP20012/labs/lab2`.

Here are some illustrations of how each of the algorithms work for the data given.

a) Insertsort:

An iterative solution would, perhaps, naturally begin at the beginning:

```
Initial state
8 1 4 1 5 9 2 6 5
Steps
8 * 1 4 1 5 9 2 6 5
1 8 * 4 1 5 9 2 6 5
1 4 8 * 1 5 9 2 6 5
1 1 4 8 * 5 9 2 6 5
1 1 4 5 8 * 9 2 6 5
1 1 4 5 8 9 * 2 6 5
1 1 2 4 5 8 9 * 6 5
1 1 2 4 5 6 8 9 * 5
1 1 2 4 5 5 6 8 9 *
```

A tail recursive version would begin at the end:

```
Initial state
8 1 4 1 5 9 2 6 5
Steps
8 1 4 1 5 9 2 6 * 5
8 1 4 1 5 9 2 * 5 6
8 1 4 1 5 9 * 2 5 6
8 1 4 1 5 * 2 5 6 9
8 1 4 1 * 2 5 5 6 9
8 1 4 * 1 2 5 5 6 9
8 1 * 1 2 4 5 5 6 9
```

```
8 * 1 1 2 4 5 5 6 9
* 1 1 2 4 5 5 6 8 9
```

b) Mergesort

```
Calling for 0 to 8
Initial state: 8 1 4 1 5 9 2 6 5
Calling for 0 to 4
Initial state: 8 1 4 1 5
Calling for 0 to 2
Initial state: 8 1 4
Calling for 0 to 1
Initial state: 8 1
Calling for 0 to 0
Initial state: 8 : result unchanged
Calling for 1 to 1
Initial state: 1 : result unchanged
Result for 0 to 1: 1 8
Calling for 2 to 2
Initial state: 4 : result unchanged
Result for 0 to 2: 1 4 8
Calling for 3 to 4
Initial state: 1 5
Calling for 3 to 3
Initial state: 1 : result unchanged
Calling for 4 to 4
Initial state: 5 : result unchanged
Result for 3 to 4: 1 5
Result for 0 to 4: 1 1 4 5 8
Calling for 5 to 8
Initial state: 9 2 6 5
Calling for 5 to 6
Initial state: 9 2
Calling for 5 to 5
Initial state: 9 : result unchanged
Calling for 6 to 6
Initial state: 2 : result unchanged
Result for 5 to 6: 2 9
Calling for 7 to 8
Initial state: 6 5
Calling for 7 to 7
Initial state: 6 : result unchanged
Calling for 8 to 8
Initial state: 5 : result unchanged
Result for 7 to 8: 5 6
Result for 5 to 8: 2 5 6 9
Result for 0 to 8: 1 1 2 4 5 5 6 8 9
```

c) Quicksort

```
Calling for 0 to 8, pivot 5
Initial state: 8 1 4 1 5 9 2 6 5
  Calling for 0 to 4, pivot 4
  Initial state: 5 1 4 1 2
    Calling for 0 to 2, pivot 1
    Initial state: 2 1 1
    Result for 0 to 2: 1 1 2
    Calling for 3 to 4, pivot 4
    Initial state: 4 5
    Result for 3 to 4: 4 5
  Result for 0 to 4: 1 1 2 4 5
  Calling for 5 to 8, pivot 5
  Initial state: 9 5 6 8
    Calling for 6 to 8, pivot 6
    Initial state: 9 6 8
      Calling for 7 to 8, pivot 9
      Initial state: 9 8
      Result for 7 to 8: 8 9
    Result for 6 to 8: 6 8 9
  Result for 5 to 8: 5 6 8 9
Result for 0 to 8: 1 1 2 4 5 5 6 8 9
```

2. a) Hopefully the $O(N^2)$ algorithm should cause no one any problem.
- b) Once the array is sorted the idea is to look at the first and last elements of the array. If their sum is less than K , we can ignore the first element, because its sum with any other array element will certainly be less than K . Similarly, if the sum is greater than K , we can ignore the last element. Either way we remove one element, and repeat the process until a sum of K is achieved or all elements exhausted. Here is the code

```
public static boolean findSum (int [] a, int k)
{
    quicksort(a,0,a.length -1);
    int lo = 0;
    int hi = a.length -1;

    while (lo <= hi)
    {
        if (a[lo] + a[hi] == k)
        {
            return true;
        }
        else if (a[lo] + a[hi] < k)
            lo++;
        else
            hi--;
    }
    return false;
}
```

3. This algorithm was invented by Donald Shell in 1959. The idea is to rearrange the subarrays of every k -th element into ascending order using Insertion sort and vary k until the whole array is sorted.

There are plenty of web resources describing Shell's sort algorithm, the best of which is probably <http://www.cs.princeton.edu/~rs/shell/paperF.ps>. Others include <http://linux.wku.edu/~lamonml/algor/sort/shell.html> and, one for lovers of arcanery, a PL/1 implementation at http://www.users.bigpond.com/robin_v/shell.htm

Implementations differ in the choice of increment sequence, here is code that uses the sequence suggested in the question.

```
public static void shellsort( int [ ] a )
{
    int [] gaps = {5, 3, 1};
    for ( int k = 0; k < gaps.length; k++)
    {
        int gap = gaps[k];
        for( int i = gap; i < a.length; i++ )
        {
            int tmp = a[ i ];
            int j = i;

            for( ; j >= gap && tmp < a[ j - gap ] ; j -= gap )
                a[ j ] = a[ j - gap ];
            a[ j ] = tmp;
        }
    }
}
```

At for each increment, or gap, the subsequences of elements separated by that gap are sorted, essentially using an insertsort. In the example below the subsequences (for $\text{gap} > 1$) are indicated by having their positions shown above them.

Gap 5 initial state

8 1 4 1 5 9 2 6 5

0 5

8 1 4 1 5 9 2 6 5

1 6

8 1 4 1 5 9 2 6 5

2 7

8 1 4 1 5 9 2 6 5

3 8

8 1 4 1 5 9 2 6 5

Gap 3 initial state

8 1 4 1 5 9 2 6 5

```
0      3      6
1 1 4 8 5 9 2 6 5
  1      4      7
1 1 4 8 5 9 2 6 5
    2      5      8
1 1 4 8 5 9 2 6 5
0      3      6
1 1 4 2 5 9 8 6 5
  1      4      7
1 1 4 2 5 9 8 6 5
    2      5      8
1 1 4 2 5 5 8 6 9
```

```
Gap 1 initial state
1 1 4 2 5 5 8 6 9
```

```
1 1 4 2 5 5 8 6 9
1 1 4 2 5 5 8 6 9
1 1 2 4 5 5 8 6 9
1 1 2 4 5 5 8 6 9
1 1 2 4 5 5 8 6 9
1 1 2 4 5 5 8 6 9
1 1 2 4 5 5 6 8 9
1 1 2 4 5 5 6 8 9
```