COMP26120
Academic Session: 2018-19

Lab Exercise 5: Iterative Sort

Duration: 1 lab session

For this lab exercise you should do all your work in your COMP26120/ex5 directory.

Learning Objectives

At the end of this lab you should be able to:

- Explain how four different iterative sorting algorithms work
- Find the Big O best, worst and average case of a given iterative sorting algorithm
- Interpret experimental results and discuss what they mean with respect to theoretical complexity results

Introducing Four Iterative Sorting Algorithms

Note that there is a tool on the course website visualising some of the algorithms from this week and next week. There are lots of similar visualisations available online elsewhere.

Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly processes the input list, comparing two elements at a time and exchanging them if they are in the wrong order. The sorting is finished if no more swaps are needed.

The name "Bubble sort" comes from the way smaller or bigger elements would "bubble" to the top of the list through swaps.

The idea of bubble sort is given as follows:

1. Compare adjacent elements. Swap those in the wrong order;
2. Repeat step 1 through the given list. After this step, the largest or smallest item would bubble to the top;
3. Repeat above steps each time with one less item, until there's no other pair of items that needs to be compared.

Insertion Sort

Insertion sort is an intuitive algorithm that constructs a sorted output list directly. At each iteration, one element from the input is taken and inserted into the right position in the ordered list.

The idea of insertion sort is given as follows:

1. Initialize the output sorted list with the first element;
2. Take the next item, scanning the sorted list from back to front;
3. Find the right place to insert this element;
4. Insert into that position;
5. Repeat steps 2 to 4.

Scanning the sorted list from back to front ensures stability. The sort can also be in-place by dividing the initial list into the sorted and unsorted part.

Bucket Sort

Bucket sort works by partitioning a the data into a finite number of buckets. The data needs to be easily bucketed e.g. numeric data can be bucketed using modular arithmetic; any data can be bucketed using bit information. Elements within each bucket are sorted individually, then merged together in order. It is a Distribution sort. The computational complexity estimates involve the number of buckets.

The idea of bucket sort with n buckets is given as follows:

1. Set up a fixed array of n buckets;
2. Go through the given list, putting each element into corresponding bucket;
3. Sort each non-empty bucket (e.g. using insertion sort). You don't need to sort buckets if they only contain one element;
4. Put elements from non-empty buckets back to the original list.

Pseudocode for bucket-sort is provided in Figure 1.

```
function bucket-sort(array, length, n){
  buckets ← new array of n initially empty lists
  for each item i in array do
    // The key must be between 0 and n-1
    let k be the key of item i
    insert i into buckets[k]
  for i = 0 to n - 1 do
    // Assuming an in-place sort
    sort(buckets[i])
  concatenate buckets[0], ..., buckets[n-1]
}
```

Figure 1: Algorithm for Bucket Sort

Radix Sort

Radix sort is a non-comparative integer sorting algorithm that processes integer along individual digits, either starting from the least significant digit (LSD, i.e., the rightmost digit) or from the most significant digit (MSD, i.e., the leftmost digit).

Processing each digit is usually done using bucket sort, which is efficient since there are usually only a small number of digits.

The idea of **least significant digit radix sort** is given as follows:

1. Take the least significant digit of each key.
2. Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
3. Repeat the grouping process with each more significant digit.
4. Concatenate the buckets together in order.

The sequence in which digits are processed by a most significant digit (MSD) radix sort is the opposite of the sequence in which digits are processed by an LSD radix sort.

Pseudocode is provided in Figure 2.

```
function radix-sort(array, length) {
  MAX_NUM ← the maximum number in array
  DIGIT_NUM ← number of digits needed to represent MAX_NUM
  buckets ← new array of BASE(e.g. 10) initially empty lists
  for each i in DIGIT_NUM
    for each item in array
      d ← $i^{th}$ least significant digit of item
      insert item into bucket[d]
    for j = 0 to BASE - 1 do
      sort(buckets[j])
  concatenate buckets[0], ..., buckets[BASE-1]
}
```

Figure 2: Algorithm for LSD Radix Sort

Description

For this exercise you need to implement three sorting algorithms - **Bubble Sort, Insertion Sort, Bucket Sort.** You do **NOT** need to implement Radix Sort, although you could do so as a practice.

Write C programs for each sorting algorithm separately. Note that if you use code from outside sources for an algorithm, you can only get partial credit for its part. See marking scheme for details.

Your code should read an integer array and its length from standard input, then output the sorted array in ascending order. The input and output format is given below.

Input

The first line of input contains a single integer n ($1 \leq n \leq 10^6$) - the length of array.

The second line contains n space-separated integers: a_1, a_2, ..., a_n with $0 \leq a_i \leq$ 32,767.

Output

Print the sorted list on a single line in ascending order with items separate by a single space.

Sample Input & Output

Sample Input 1	Sample Output 1
5 26391 18088 27865 32649 31711	18088 26391 27865 31711 32649
Sample Input 2	**Sample Output 2**
5 23836 7258 17581 11457 22474	7258 11457 17581 22474 23836

Comparison and Analysis

The final part of this exercise is to provide a comparison of the sorting algorithms. In terms of:

- The worst and best case for each sorting algorithm.
- The time complexity under worst, average and best cases.

To help you do this we have provided a set of data of different sizes covering three cases: *ascending*, *descending*, and *random*. You can access this data as a zip file at `/opt/info/courses/COMP26120/problems/ex5`. There is a README in the same directory with some hints on how to use this.

You should complete two text files (also available in the above directory) called `times.txt` and `complexities.txt`. These text files reflect the tables given in Figures 3 and 4 - we've filled in one of the boxes for you! For the first table you should record the execution time of the different data sets. For the second table you should give the time complexity for the different sorting algorithms in worst, average and best case. You may need to look up this last information but **the TA will ask you to explain why some of these complexities hold.** It does not matter how you layout the information in these files as long as it is readable.

Implementation Hints

Insertion sort and bubble sort should be straightforward. For bucket sort you will need to use arrays to be competitive (e.g. whilst you can use linked-lists if you want it is unlikely to perform well). You have two possible approaches with bucket sort. The first approach would be to use buckets of size one, the second approach is to define a two-dimensional array. Note that the testing server has a time-limit per problem and your solution may fail due to this.

Edit: Here are some further hints on the two suggestions for bucket sort:

- *One number per bucket.* Note that the range of numbers that we can expect in the input is relatively limited. Given this limited range we can create a 'bucket' per possible number (e.g. an array of size 32,767) and store in each index of the array the number of number in the list of that value. This can then be used to produce a sorted list. It is important to note that this solution solves a special case of the sorting problem. Can we do this for integer lists in general? What about lists of strings?

- *n buckets.* Here you should dynamically allocate n buckets and define which numbers go in which (e.g. if you had two buckets the even numbers could go in one and the odd numbers in the other but 2 isn't very many buckets). How big do they need to be? (think about the worst case here). You will need to keep track of how many numbers are in each bucket. Once the input has been put into buckets you can use one of your other algorithms to sort those smaller arrays.

Submission

You should submit five files (*bubble.c*, *insertion.c*, *bucket.c*, *times.txt*, *complexities.txt*. As usual, you should check the report for errors and to view performance graphs for your submitted code. The data used for testing is not as big as the data we want you to benchmark your code with – this is to reduce the load on the testing server.

Algorithm	Length	Ascending	Descending	Random
Bubble Sort	1000			
Insertion Sort	1000			
Bucket Sort	1000			
Bubble Sort	10,000			
Insertion Sort	10,000			
Bucket Sort	10,000			
Bubble Sort	100,000			
Insertion Sort	100,000			
Bucket Sort	100,000			
Bubble Sort	1,000,000			
Insertion Sort	1,000,000			
Bucket Sort	1,000,000			

Figure 3: Comparison of execution time

	Worst case	Average case	Best case
Bubble Sort	O(n^2)		
Insertion Sort			
Bucket Sort			
Radix Sort			

Figure 4: Comparison of time complexity in Big O notation

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Implementation of sorting algorithms	
All algorithms are implemented correctly (all test cases pass) without reusing code from the internet	(3)
All algorithms are complete but a few test cases fail	(2.5)
All algorithms are implemented correctly (all test cases pass) but depends in part on code from the internet	(2)
Only two algorithms are complete but they are fully correct	(2)
At least one algorithm is complete and correct	(1)
An attempt has been made but it does not compile or fails most tests	(0.5)
No attempt has been made	(0)

Understanding of sorting algorithms	
The student can explain all algorithms well and in detail and how their code implements each algorithm. They can identify and justify the design decisions they made. They can explain what it means for a sorting algorithm to be stable and how it applies to the given sorting algorithms. Their understanding is impressive.	(3)
The student gives a good explanation of all algorithms and how their code implements each algorithm.	(2.5)
The student can explain bubble sort and insertion sort well but struggles with bucket sort.	(2)
The student can explain one of the algorithms.	(1)
The student cannot explain how any of the algorithms work	(0)

Quality of code	
The code is of good quality (e.g. appropriate comments, good layout throughout).	(0.5)
The quality of code is good enough but not great (e.g. only a few comments)	(0.25)
The code is not of acceptable quality (e.g. no comments, bad layout or variable names)	(0)

Record of execution time	
Correctly recorded execution times for completed algorithms and can reflect on what these numbers mean with respect to the known complexities of the algorithms i.e. do they agree?	(1.5)
Recorded some execution times can reflect on what these numbers mean	(1)
Correctly recorded execution times for completed algorithms but cannot reflect on what they mean	(1)
Made an attempt at recording execution time but made some mistakes e.g. forgot to consider the different cases	(0.5)
Did not attempt	(0)

Comparison of complexities	
Complete the table and can adequately explain why two complexities hold for two randomly chosen complexities in the table. Can explain the relation between bucket sort and radix sort.	(2)
Complete the table and can adequately explain why two complexities hold for two randomly chosen complexities in the table	(1.75)
Table may not be complete but can explain at least two randomly chosen complexities	(1.5)
Completed the table but cannot adequately explain it	(1)
Did not attempt	(0)