Problem 2

6 мая 2019 г.

Ed. 1.3 as of 07.05.2019

General notes

The goal of the current task is to experimentally estimate the time complexity of different sorting algorithms on different data structures.

During the first ADS classes we considered the following sorting algorithms.

1. InsertionSort sorts a given sequence of comparable elements in $O(n^2)$.
2. MergeSort sorts a given sequence of comparable elements in $O(n \log n)$, with a constant factor C_1.
3. QuickSort sorts a given sequence of comparable elements on the average in $O(n \log n)$, with a constant factor C_2.
4. CountingSort, BucketSort, RadixSort sort a given sequence of non-negative integers in $O(n)$.

Most of these algorithms can be implemented slightly differently, they can also use different data structures for storing elements which are to be be sorted. We consider exactly two following data structures:

- a dynamic array implemented as `std::vector<T>`;
- a doubly linked list implemented by students themselves as a class `LinkedList<T>` (see the problem set for «Workshop 8»).

The complexity of these algorithms is estimated in a series of experiments on some randomly created data (sequences of numbers/strings to be sorted) following the approach provided for Problem 1 (study on complexity of multiplication algorithms).

You have to create a testbed application that prepares data, applies a set of sorting algorithms, collects average running time for every "algo modification + data" setting, stores the produced results in a persistent memory (external CSV file). Finally, the resulting data **must** be visualized by using Python features, such as *matplotlib* (we do not mean using Excel for this problem).

You have to prepare a brief report and present it in the form of PDF document. The report contains the problem statement, samples of data sorted by the algorithms, and graphs/charts drawn by *matplotlib*, and a brief discussion on the obtained results.

The program code and the report must be uploaded in the conformed GIT repository.

Algorithm variants

The complete work must contain a study for the following algorithms and their variants.

1. InsertionSort implementation for vector and linked list sequences. The algorithm does not use any additional storage, so it implements in-place sorting. You have to provide experiments on both integers and string sequences.

   *Total number of implementations: 4 ({integers, strings} * {array, list}).*

2. MergeSort implementation for vector and linked list sequences. The algorithm may use additional storage for a vector version but must sort a given linked list w/o creating a copy of any of its parts (it is well known that MergeSort effectively sorts lists). You have to provide experiments on both integers and string sequences.

*Total number of implementations: 4 ({integers, strings} * {array, list}).*

3. QuickSort implementation for vector and linked list sequences. The algorithm does not use any additional storage, so it implements in-place sorting (according to Cormen's approach). You have to provide experiments on both integers and string sequences.

   *Total number of implementations: 4 ({integers, strings} * {array, list}).*

4. CountingSort implementation for vector of integers. The algorithm uses only vectors both for input/output and as an intermediate storage.

   Total number of implementations: 1.

5. BucketSort implementation sorting a given vector of integers by using a linked list for representing buckets internally.

   Total number of implementations: 1.

6. RadixSort implementation sorting a given vector of integers using BucketSort as a stable sorting algorithm applied for individual digits. You have to implement it for base r = 2 and r = 10.

   Total number of implementations: 2.

Note that for CountingSort and BucketSort algorithms you have to report the size of the intermediate arrays related to the difference between the maximum and the minimum value in a given sequence of integers.

*The total number of all algorithm implementations is **16**.*

Implementation notes

It is necessary to apply all compatible (with regard to input data) algorithms exactly to the same input data. For instance, you can create a method `makeRandomNumbers(size_t n)` generating up to `n` non-negative integers and returning them as a `vector<int>` object. Then you are able to apply all mentioned algorithms for such input data. Then you may convert them from `vector<int>` to a `LinkedList<int>` object by providing a method called, e.g., `convertToList(const vector<int>& src)`. Such a list can be used as input data for comparison-based sorting algorithms. The same idea can be implied for a vector of strings and a list of strings.

Recommended procedure for conducting experiments

For a given k:

1. Generate a sequence of k-random numbers (as a vector).
2. Apply sorting algorithm "A" to a copy of the given vector. Measure time needed for the algorithm to be executed.
3. Repeat the third step not less than three times, calculate the everage time. Store it in persistent memory.
4. Repeat steps 2-3 for all appropriate algorithms ("B", "C") etc.
5. Convert the vector otained at step 1 to a linked list.
6. Apply steps 2-4 for algorithms "A", "B", etc. able to deal with a list as input data.

Steps 1-6 are repeated for $k = 1$ to N, where N is experimentally determined value such that sorting of N randomly generated numbers takes a reasonable time.