Algoritms and data structures
Laboratory assignment, 1.5 hp

University of Skövde

November 12, 2021

Aim

The aim of this lab exercise is to help students get familiar with the design and analysis of different types of data structures and algorithms. The lab exercise includes a number of tasks related to several main questions. It is important to remember that the lab exercise is not only a programming task. Instead, the focus of our aim is problem solving which can even be carried out before starting to write your code (with a pencil and paper).

Examination

The lab is examined by a written report and a demonstration of the implemented algorithms. Working in groups of two people is encouraged.Only one report is submitted per group. However, the demonstration is assessed individually. The implementation of the algorithms is expected in C / C ++ and the demonstration takes place following an announced schedule by tutors. Poorly explained solutions, or unnecessarily high time-complexity, result in a resubmission chance (only once), even-though the algorithm is correct. A reasoning that shows the correctness of the algorithm is expected to be included in the report. Each task specifies what the report must actually include to be considered complete. As a rule of thumb, all tasks need to be completed but in some cases, a high quality solution with good reasoning for one task can make up for a low-quality solution for another task. **There will only be one chance to supplement the report, so please be thorough and include solutions for all tasks in the first submission.** During the lab sessions, you can get help with, for example, the structure of the report, how to write pseudo-code, or how to create an expression for the time-complexity. All information about how to submit your report as well as the deadline for submissions are available in Canvas. **You must participate in the oral exam to get credits for the lab exercise, please pay attention to your time-slot on the Canvas website.** Read the next section before beginning your lab exercise.

Describing algorithms

1. **Intuitive description**

    - Firstly you have to explain how the algorithm solves the problem. To do this, it is good to use both text and figures. The description should be intuitive, easy to understand, also brief and concise. For example, using a series of figures with correspondent text which describes how the algorithm works is often a good way to convey an intuitive description.

2. **Abstract algorithm in pseudo code**

   - This is a descriptive programming language in which you can use abstract data types (ADT), for example, lists, queues and graphs as well as the expression type such as "for each node in Graph G" and "if x is in the list L". The description should be programming-language independent. The goal of this step is to demonstrate a) a clear structure of the algorithm, and b) you have a good understanding on how the algorithm could be implemented in a programming language. **Note that the description of an algorithm cannot be too abstract so that some critical parts are missing.** For example, if you are required to print-out all the leaves of a binary tree, "Traversing the tree" is not a good abstraction because it is the critical part of printing out a tree. You have to describe in details how the traversal in a tree works. Only the expressions that can be immediately implemented could be abstracted in a pseudo-code.

3. **Time complexity analysis**

   - Time complexity analysis is an approach to determine how an algorithm uses computer resource (processing-time) with mathematical expressions. It is very important for you to clearly reason and demonstrate how time-complexity expressions can be derived from your algorithm (to show you actually can carry-out the analysis). It is not enough to simply write a time-complexity expression as a result. This also applies if the algorithm is known. You may of course use the textbook (and other resources) as a reference but it is important that you describe the reasoning behind the time complexity in your own words (it is absolutely not acceptable to write that the time complexity is a certain specific expression "as it says so in the textbook").

Problems

Here is the description of the four main problems with corresponding tasks, that you are expected to carry out as part of the lab exercise.

Problem 1 – Sorting (Related lectures: F1, F2, F3)

A modified version of the sorting algorithm "Bucket Sort" can be explained according to the following intuitive descriptions (**v** is the vector to be sorted):

1. Find the greatest element **k** in the vector **v**
2. Create a new vector **w** with length of $\mathbf{k} + \mathbf{1}$
3. Initiate $\mathbf{w}[i] = 0$ for all $0 \leq i \leq k$.
4. Iterate and update the vector **w** according to: $\mathbf{w}[\mathbf{v}[i]] = \mathbf{w}[\mathbf{v}[i]] + 1$ for every $0 \leq i < \mathbf{v}.size$
5. Use **w** to write out the vector **v** in a sorted order.

Task 1.1 – *Describe the above algorithm with pseudo code and implement the algorithm in C/C++.*

In the report, it must include the following parts:

- *Intuitive description of the algorithm steps*
- *Abstract algorithm in pseudo code*
- *Implementation in C/C++ code*

Task 1.2 –*Suppose that the largest number in the above vector* **v** *is p and the length of vector* **v** *is* **n**. *Make an analysis of the time complexity for this algorithm by using* **p** *and* **n**. *In doing so, the following parts must be included in the report:*

- *Time complexity analysis expressed in $O(\cdot)$-notation*

Task 1.3 –*There is an obvious disadvantage in this modified version of "Bucket Sort" algorithm dependent on input data. Describe the disadvantage.*

The following parts should be included in the report:

- *An explanation or example to show the disadvantage of the above "Bucket Sort" algorithm*

Task 1.4 – *Implement insertion sort algorithm, and justify the time complexity based on empirical measurements for different vectors of distinct lengths. Plot a graph to show the time as a function of vector length n. One recommended*

approach to justify the time complexity $T(n) = O(f(n))$ from the measurements is to show a series $\frac{T(n_1)}{f(n_1)} ... \frac{T(n_k)}{f(n_k)}$ converges to a constant.

The following parts must be included in the report:

- *A graph showing the time as a function of the vector length*
- *A time complexity expression in big O-notation, with justification of the mathematical expression based on the measurement data.*

Problem 2 – Social Network (Related lectures: F4, F5, F9)

The concept of social networks in which members can make friends in various ways has become increasingly popular. You will learn a new innovative idea to help members find new friends in a social network. The idea is to allow all members of the network to specify their adversaries and model this as a graph, i.e. an *adversary graph*. Then this graph can be used for a member to find new friends according to following description:

- An adversary's adversary is a friend.
- A friend's adversary is an adversary.

All members in the graph are classified according to the above description. Note that, based on the adversary graph, a member can be classified as a friend or an adversary and can choose to classify a person according to "the shortest distance" rule, i.e. the path going through the least number of edges (an edge is a connection between two members). Below is an example of an adversary graph where members are classified as friends or adversaries based on the above recursive description (all directed edges in the graph below models adversary relations):

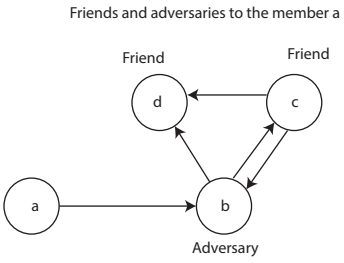

Figure 1: A example to show the friends and adversaries of a from outside the adversary graph.

Note that member d is classified as a's adversary via the path $a \to b \to c \to d$. However, there is a shortest path $a \to b \to d$ where d is classified as a friend.

Task 2.1 (The algorithm must be demonstrated to the lab tutor) – *Design an algorithm based on* breadth-first search *so that a specific individual can find all friends by using the* **adversarygraph** *. Implement the algorithm in C/C++. It should be easy to test whether the algorithm works for any adversary graph.* **The algorithm should use a queue from the STL library.**

The following parts must be included in the report:

- *Intuitive description of the algorithm.*
- *Abstract algorithm in pseudo code.*
- *Time complexity analysis with $O(\cdot)$-notation*
- *C/C++ code of the allgorithm implmentation.*

Problem 3 – Compression by Huffman encoding (Related lectures: F6, F8)

A method of compressing a text string is to use so-called Huffman coding which is to encode the characters that occur most frequently in the string with a short sequence of bits as possible. We can take the text string `aaaabbbc` as a simple example. Each letter in this sequence contains eight bits (according to ASCII table) making a total of $8 * 8$ bits i.e. eight bytes for the entire sequence. However, we can see that this string consists of only three characters in which case we do not need eight bits to represent a single character, we could instead make use of bit encoding `a - 0, b - 11, c - 10`, which means we could represent the string `aaaabbbc` by the bit sequence `000011111110` with only 12 bits (2 bytes). Compared to the previous example, we saved 6 bytes in this simple example by using this coding approach, which could be used for the purpose of compressing a text string (or file). This coding approach is called Huffman coding. An important feature of Huffman coding is that the characters that occur most frequently in the text string always get shortest bit sequence to obtain the most efficient compression as possible (e.g. `a` is encoded by only one bit since the character appeared most frequently in the string of the above example). The actual Huffman coding can be represented effectively through a tree representation, as illustrated next.

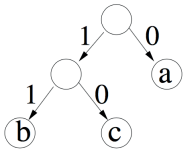

Figure 2: Tree representation of Huffman coding

One can use the above tree for coding the bit-sequence `000011111110` by following the branches from the root of the tree and print each character in

the leaves when it is reached, and then repeat the process for each character. For example, we reach `a` four times, followed by `b` etc., to get the encoded bit-sequence of the compressed text. The algorithm for constructing Huffman trees could use a min-priority queue ADT, as described below:

1. count the occurrence of the various characters that occur in the string and construct a tree T consisting of a node containing the character and a weight w equal to the occurence of the character in the string
2. Insert each node in a min priority queue where the min relation is determined by weight w
3. Pick the current "top" element T_1 (a tree) from the priority queue (i.e. that is the element with the minimum priority, which will be retrieved and removed)
4. Pick the current "top" element T_2 (a tree) from the priority queue (i.e. it will be retrieved and removed, as well)
5. Construct a new tree $T_{1,2}$ by creating a new node with weight $w_1 + w_2$ (i.e. the sum of the weights of trees T_1, and T_2). T_1 and T_2 are subtrees of $T_{1,2}$. Place the tree $T_{1,2}$ in the priority queue.
6. Repeat from step 3 as long as the priority queue contains more than one trees.
7. Return the remaining tree in the priority queue (Huffman coding).

The figure below illustrates how the above procedure works on the string `aaaabbbc` (where the numbers indicate the tree weights – occurrence of each character in the tree):

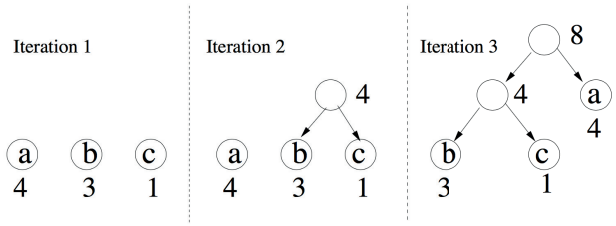

Figure 3: Trees in the priority queue after each iteration.

Note that, in each iteration you should pick the trees that have minimum weights, assemble them into a new tree, and then you put this tree in the priority queue.

Task 3.1 *Implement the above-mentioned Huffman coding algorithm by* **using a priority queue from STL** *and a tree class with the following member functions:*

```
#include <vector>
using namespace std;

class Tree {
public:
    Tree(int w, char c);
    Tree(int w, Tree* t1, Tree* t2);
    ~Tree();
    int getWeight() const;
    void printTree(vector<char>& bitString) const;
private:
    Tree* left;
    Tree* right;
    int weight;
    char c;
};
```

To avoid problems of handling pointers in the priority queue, it is easier to use the following "wrapper structure":

```
#ifndef NULL
#define NULL 0
#endif

struct TreeWrapper {
    TreeWrapper() {
        tree = NULL;
    }

    TreeWrapper(Tree* t) {
        tree = t;
    }

    bool operator<(const TreeWrapper &tw) const {
        return tree->getWeight() > tw.tree->getWeight();
    }

    Tree* tree;
};
```

This structure is used before inserting anything into the priority queue, for example

```
    #include <queue>
    priority_queue<TreeWrapper> q;
    q.push(TreeWrapper(new Tree(4, 'a')));
```

```
q.push(TreeWrapper(new Tree(3, 'b')));
q.push(TreeWrapper(new Tree(1, 'c')));
```

The function `printTree(...)` *to print Huffman coding of the different characters like the following example:*

```
0 : a
1 0 : c
1 1 : b
```

where Huffman coding is based on the inserted values in the above example using the priority queue. `printTree(...)` *is used to print out recursively (with input argument* `vector<char>& bitString)).` *Here is an example of how to invoke the* `printTree(...)`

```
// t is a tree
vector<char> bitString;
t->printTree(bitString);
```

The following parts should be included in the report:

- *Intuitive description of the algorithm*
- *Abstract algorithms in pseudo code*
- *Time complexity analysis in $O(\cdot)$-notation*
- *C/C++ code*

Problem 4 – Time complexity (Related lectures: F10, F11)

One way to analyze the time complexity of a recursive algorithm is to first describe the algorithm in terms of a recursive function. For example, you can describe the time complexity $T(n)$ with n being the size of the input, of a "divide and conquer " technique, using the expression:

$$\begin{aligned} T(n) &= 2T(n/2) + kn \\ T(1) &= k \end{aligned} \tag{1}$$

The rationale of the above expression is the result of dividing the problem into sub-problems of size n/2 (for example binary search). This process adds some linear complexity kn. The above expression can be interpreted as the time it takes to solve the problem of size n is twice the time it takes to solve the same problem for input-size $n/2$ plus a term kn. For the base case, i.e, $T(1) = k$, means that a problem of size 1 takes a constant time k to solve.

From the above equations, you can deduce the mathematical expressions of the time complexity: $O(nlog(n))$.

Now, suppose you have deduced the following recursive expressions for the time-complexity of an algorithm:

$$\begin{aligned} T(n) &= T(n-1) + T(\lceil n/2 \rceil) + n \\ T(1) &= 1 \end{aligned} \tag{2}$$

where $\lceil n/2 \rceil$ is the least integer greater than $n/2$.

Task 4.1 *Design a trivial (the most obvious solution) recursive algorithm that calculates $T(n)$ in the expression (2). Implement the algorithm in C / C ++ code.*

The following parts should be included in the report:

- *Intuitive description of the algorithm.*
- *Abstract algorithm in pseudo code.*
- *C/C++ code*

Task 4.2 (The algorithm must be demonstrated to the lab tutor) *Design an algorithm based on the design principles of dynamic programming, which calculates $T(n)$ in expression (2). Check lecture material materials about dynamic programming principles, and implement the algorithm in C / C ++.*

The following parts must be included in the report:

- *Intuitive description of the algorithm.*
- *Abstract algorithm in pseudo-code.*
- *Time complexity analysis with $O(\cdot)$-notation*
- *C/C++ code*

Task 4.3 Which algorithm out of the previous tasks, i.e. Task 4.2, or the trivial Algorithm of Task 4.1 calculates $T(n)$ more effectively? Explain via some reasoning and examples why there is a difference. Test some different values for your implemented algorithms and see what makes the difference.

The following parts must be included in the report:

- A reasoning process and an example that show the cause of the difference. It is not enough that you only state which algorithm runs faster via measurements. You have to explain the reasons which prove that you have understood the cause of the difference.