Lab session, sorting

February 23, 2017

1 Introduction

In this lab session you will learn how to use the different sorting algorithms on different data and not just integers. For this lab you will need the files:

- **actors.hh** which contains a bunch of actors and movies where they performed.
- **actresses.hh** with actresses and movies where they performed.
- **lab.cc** with all the source code in this document.

2 The modules that you will use

All the data will be stored in a **vector** data structure. As you have seen from previous lectures, this data structure serves to the purpose of representing a sequence $\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}$ of n elements indexed from 0 to $n-1$. The **string** data type is used to represents strings or character sequences.

```
In [ ]: #include <iostream>
        #include <string>
        #include <vector>

        using namespace std;
```

3 The actor class

The attributes of an **Actor** are:

- Its real name,
- the movie where he/she performed, and
- the character he/she played.

Here is the class definition for an actor.

```
In [ ]: class Actor {
        private:
            string movie;
            string name;
            string character;
        public:
            Actor(const string& m, const string& n, const string& c)
                : movie(m), name(n), character(c) {}
            void print() {
                cout << "Name: " << name << " movie: " << movie << endl;
            }
        };
```

As we need some data, the following inclusion will create a vector of actors called **actors** and one with actresses called **actresses**.

```
In [ ]: #include "actors.hh"
        #include "actresses.hh"
```

4 Warming up

Before the core part of this lab we need to warm up our brains. Perform the following exercises.

4.1 Merging both actors and actresses

Write a function to merge both **actors** and **actresses** vectors into a single vector called **people**. Remember to provide a bound for the worst case complexity in the general case.

4.2 Filetring the actors of a specific movie

Write the **filter** function that takes the **people** vector and the name of a movie. **filter** will return a vector with all the actors that participated in that movie.

4.3 Finding performances

Write the **performance** function that takes the **people** vector and computes how many times every actor performed. Provide a bound for the complexity of this function.

5 Bubble sort

Use write a program that sorts the vector **people** using the bubble-sort algorithm seen in the previous lectures. Here is the source code of the algorithm that works to sort a vector of integers. You have to change it to sort the vector of actors. Sort the vector by the actor's name and then by the character he/she played.

```
In [ ]: void swap(vector<int>& v, int p, int q) {
          int tmp = v[p];
          v[p] = v[q];
          v[q] = tmp;
        }
```

```
In [ ]: void bubbleSort(vector<int>& v) {
          int n = v.size();
          for (int i = 1; i <= n - 1; i++) {
            for (int j = 0; j <= n - 2; j++) {
              if (v[j] > v[j + 1]) swap(v, j, j + 1);
            }
          }
        }
```

6 Merge sort

Now use the merge sort algorithm to sort the **people** vector. This time you will sort not by name or movie. Instead, you will sort the actors by the number of participations. Hence, the first element in the vector will be the actor with least number of performances.

As usual, here is the source code of merge sort for a vector of integers.

```
In [ ]: void merge(vector<int>& v,
                   const vector<int>& left,
                   const vector<int>& right) {
          int i = 0, j = 0, k = 0;
          while (i < left.size() && j < right.size()) {
            if (left[i] < right[j]) {
              v[k] = left[i];
              i++;
            } else {
              v[k] = right[j];
              j++;
            }
            k++;
          }
          while (i < left.size()) {
            v[k] = left[i];
            i++;
            k++;
          }
          while (j < right.size()) {
            v[k] = right[j];
            j++;
            k++;
          }
        }
```

```
In [ ]: void mergeSort(vector<int>& v) {
          if (v.size() < 2) return;
          int mid = v.size() / 2;
          vector<int> left(mid, 0);
          vector<int> right(v.size() - mid, 0);
          for (int i = 0; i < left.size(); i++) left[i] = v[i];
```

```
      for (int i = mid; i < v.size(); i++) right[i - mid] = v[i];
      mergeSort(left);
      mergeSort(right);
      merge(v, left, right);
    }
```

7 Quick sort

Analyze the following implementation of the quick sort algorithm and make the changes to use this algorithm to perform the same task as in the previous exercise.

Here is the code for integers.

```
In [ ]: int partition(vector<int>& s, int lo, int hi) {
          int pivot = s[lo];
          int left = lo;
          int right = hi;

          while(left < right) {
            while(s[right] > pivot) right--;
            while((left < right) && (s[left] <= pivot)) left++;
            if(left < right) swap(s,left,right);
          }

          swap(s, right, lo);
          return right;
        }
```

```
In [ ]: void quicksort(vector<int>& s, int lo, int hi) {
          if (lo < hi) {
            int p = partition(s,lo,hi);
            quicksort(s,lo, p-1);
            quicksort(s, p+1, hi);
          }
        }
```