Assignment week 10: Order of Run-time Complexity

Course ‘Imperative Programming’ (IPC031)

Make sure to start working on the assignment **before the lab** session, otherwise you will be too late for the deadline

1 Background

In this assignment you analyze the order of run-time complexity of a number of functions. In addition, to relate the order of run-time complexity of the sorting algorithms selection sort, insertion sort, bubble sort, and heap sort, you count the number of comparison operations that are used during sorting and visualize these results. For this purpose, use the “`main.cpp`” of your solution of assignment 9. For visualization, you are going to generate *comma-separated value* text files[1] and one of the programs *Office Excel*, *LibreOffice*, or *Numbers* to generate a chart out of the data (see instructions in Part 3 of this assignment).

2 Learning objectives

After doing this assignment you are able to:

- Analyze algorithms for their order of run-time complexity;
- Count the number of comparison operations of the sorting algorithms and relate them to their run-time order of complexity.

3 Assignment

Part 1: Analyze the order of run-time complexity

Consider the algorithms below. Identify their input and derive the worst case order of run-time complexity in terms of the input size. Give a short(!) explanation for your answers. Include your answers in “`main.cpp`” as a comment using the provided template.

(a)

```
void read (ifstream& inputfile, int data [N][N][N])
{
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        inputfile >> data[i][j][k];
}
```

(b)

```
void read (ifstream& inputfile, int data [L][M][N])
{
  for (int i = 0; i < L; i++)
    for (int j = 0; j < M; j++)
      for (int k = 0; k < N; k++)
        inputfile >> data[i][j][k];
}
```

(c)

```
bool occurs (string text, char x)
{
  int i = 0;
  while (i < ssize (text) && text.at (i) != x)
    i++;
  return i < ssize (text);
}
```

(d)

```
bool occurs (char text [], char x)
{
  int i = 0;
  while (i < strlen (text) && text[i] != x)
    i++;
  return i < strlen (text);
}
```

[1]For more information about *.csv* files, see `https://en.wikipedia.org/wiki/Comma-separated_values`

(e)

```
bool is_prime (int x, int& divisor)
{
  if (x <= 1)
    return false;
  for (divisor = 2; divisor <= sqrt (static_cast<double> (x)); divisor++)
    if (x % divisor == 0)
      return false;
  return true;
}
```

(f)

```
void add1 (int source [], int length, vector<int>& dest)
{
  for (int i = 0; i < length; i++)
    dest.push_back (source[i]);
}
```

(g)

```
int add2 (int dest [], int length, vector<int>& source)
{
  int no_added = 0;
  while (no_added < length && ssize (source) > 0)
  {
    dest[no_added] = source.at (ssize (source)-1);
    source.pop_back ();
    no_added++;
  }
  return no_added;
}
```

(h)

```
void add3 (int n, vector<int>& source)
{
  while (n>0)
  {
    const int N = ssize (source);
    for (int i = 0; i < N; i++)
      source.push_back (source.at (i));
    n--;
  }
}
```

(i)

```
bool might_be_sorted (const vector<int>& data)
{
  return ssize (data) == 0 || data.at (0) <= data.at (ssize (data)-1);
}
```

Part 2: Measuring the sorting algorithms

In order to get a feeling about how much work is done by the sorting algorithms, we are going to count how many times the algorithms perform `<` and `==` while sorting an increasing number of elements of the music database. To do this, add a *global* `int` variable `g_count` that is set to zero each time before sorting a portion of the music database starts, and change your implementation of `<` and `==` on `Track` values such that they increment `g_count`. In this way you obtain a measure of the number of steps that are involved in sorting a portion of the music database.

The number of comparisons are counted in a gradual fashion, for growing slices of index values: 0–500, 0–1000, 0–1500, *etc.* up until 0–6500 (in the case of the current music database). This means you first sort the first 500 tracks, then the first 1000, *etc.* You must start with the same unaltered database before sorting each slice. Do not sort the first 500 tracks, then sort the first 1000 tracks with the first 500 tracks already having been sorted.

To give a numerical example; suppose we wish to sort `4, 2, 1, 6, 3, 5` with growing slices of size 2, i.e. 0–2, 0–4, up until 0–6. We would first sort `4, 2`, then `4, 2, 1, 6`, and finally `4, 2, 1, 6, 3, 5`. Note that after sorting `4, 2` we do not try to sort `2, 4, 1, 6`, where the first two elements have already been sorted. Instead we try to sort `4, 2, 1, 6` as this is the next slice of the unaltered `4, 2, 1, 6, 3, 5` data.

In short: obtain the next slice by making a copy of part of the unaltered database, and remember to set `g_count` to zero before sorting each slice.

A *.csv* file generated by your program must be formatted as follows:

- print the text `"sorting␣algorithm"`, followed by the slice numbers `500 1000 1500 ... 6500` all separated by a single `,` followed by `endl`;
- print the text `"insertion␣sort"`, followed by the counts of *insertion sorting* 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* music database all separated by a single `,` followed by `endl`;
- print the text `"selection␣sort"`, followed by the counts of *selection sorting* 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* music database all separated by a single `,` followed by `endl`;
- print the text `"bubble␣sort"`, followed by the counts of *bubble sorting* 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* music database all separated by a single `,` followed by `endl`;
- print the text `"heap␣sort"`, followed by the counts of *heap sorting* 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* music database all separated by a single `,` followed by `endl`;

Implement the function

```
void generate_csv (const vector<Track>& tracks, ofstream& os)
```

which given `tracks`, the music database, will generate a *.csv* file as outlined above, and write it to the output file stream `os`.

Hint: Introduce a helper function `get_slice`, which given a vector of tracks `t` and a slice `s` returns a new vector containing the elements `t.at(first(s)) ... t.at(last(s))`.

On Brightspace you can find the file "`assignment-10-mandatory-files.zip`", which in addition to our familiar music database "`Tracks.txt`" contains the following files:

- "`Tracks_sorted.txt`": The music database in a sorted order.
- "`Tracks_random.txt`": The music database in a random order.
- "`Tracks_reverse.txt`": The music database in a sorted, and then reversed order.

In the provided "`main.cpp`" you find a `main` function that will generate measurement *.csv* files for these four music databases using your `generate_csv` function, and save them as:

- "`measurements_tracks.csv`"
- "`measurements_sorted.csv`"
- "`measurements_random.csv`"
- "`measurements_reverse.csv`"

After implementing and testing your code, run your program to generate these measurement files. They will be written in the directory containing your program executable, which is most likely the "`build`" directory. Copy these files to the directory containing your "`assignment_10.txt`" file.

Important: In order to have a correct counting, you must deactivate the execution of all `assert` statements. Do this by writing `#define NDEBUG` before any `#include <...>` statements. By commenting out this line you can control whether assertions are enabled or not. Enable assertions to verify your code works, but be sure to disable them while measuring.

Part 3: Creating charts

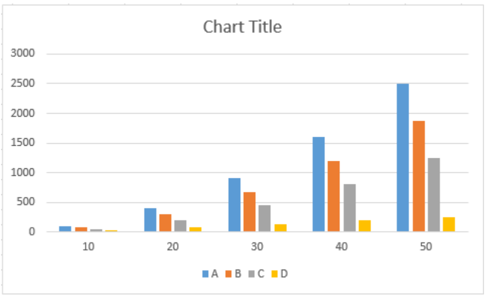
Now that we have obtained our raw measurement data in the form of *.csv* files from Part 2, we are going to create charts in order to visualize them. Instructions to create a chart from a file called "`sorting1.csv`" are as follows.

- Office Excel: Open "`sorting1.csv`". Select all 5 rows of all columns of data. Select command *Insert*, *Recommended Chart*, chart type *Clustered Column*. Press *Ok* to add the chart to the opened spreadsheet.
- LibreOffice: Open "`sorting1.csv`". Check that the *Text Import - [sorting1.csv]* window has selected *"Comma"* in *"Separator Options"*. Press *Ok*. Select all 5 rows of all columns of data. Select command *Insert*, *Chart...*. In the *Chart Wizard*, in *Steps 2. Data Range* select *Data series in rows* and *First row as label* and *First column as label*. Press *Finish* to add the chart to the opened spreadsheet.
- Numbers: Open "`sorting1.csv`". Select all 5 rows of all columns of data. Insert a '2D-column' diagram.

Add an appropriate title, and save the chart as "`sorting1.png`" (e.g. via a screen capture and an image cropping tool).

Note: To check if you can generate a proper chart image file, copy-and-paste the below left lines into an empty *.csv* file and follow the above instructions to obtain a *.png* file that should look similar to the image below right (generated on a Windows machine with Office Excel):

```
algorithm,10,20,30,40,50
A,100,400,900,1600,2500
B,75,300,675,1200,1875
C,50,200,450,800,1250
D,30,80,120,200,250
```



Convert all four *.csv* files obtained in Part 2 into charts, and save them as *.png* images named:

- "`measurements_tracks.png`"
- "`measurements_sorted.png`"
- "`measurements_random.png`"
- "`measurements_reverse.png`"

Part 4: Analyzing the measurements

Do the measured results of the sorting algorithms match with your expectations according to their order of run-time complexities? Discuss this in "`main.cpp`" using the provided comment. In particular, include the following aspects in your analysis on each algorithm:

- What is the worst/best/average case run-time complexity of the algorithm?
- When do you expect each case to occur?
- Does the order of the database significantly impact the measured number of operations?
- If so, explain why this difference occurs.

For your analysis you can refer to both the raw measurement data obtained in Part 2, and the charts produced in Part 3. You may generate additional graphs if this aids your analysis, but clearly state how they were obtained.

4 Products

As product-to-deliver you upload to Brightspace the following files:

- "`main.cpp`" with your solutions for Part 2 of the assignment, and Parts 1 and 4 as comments.
- The four *.csv* files with raw measurement data generated in Part 2.
- The four *.png* files with charts created in Part 3.
- Any additional charts or measurement data used for your analysis in Part 4.

Deadline

Lab assignment: Friday, November 22, 2024, 23:59h

Important notes:

1. check that you have actually submitted your solution in Brightspace.
2. the deadline is firm, and it is impossible to upload a solution after expiry. In that case you fail the assignment.
3. you can upload solutions to Brightspace several times, but only the last submission is stored.
4. identify yourself and your lab partner in every uploaded document. The identification consists of your first and last names, student numbers, and number of (sub) assignment. By identifying yourself, you declare that the uploaded work has been created solely by you and your lab partner.
5. your work will be judged and commented upon. We expect that you obtain the feedback, read it, and use it to for the next exercises.

6. it is essential that you only submit your own solution, never copy somebody/something else’s solution, and never share your solution—in particular: **AI tools (including but not limited to Github Copilot or ChatGPT) are not permitted**, solutions from previous year cannot be reused, and finally, you and your lab partner take joint responsibility for the assignment you submit.