Bonus assignment 10: Hybrid sorting

Course 'Imperative Programming' (IPC031)

1 Background

In this assignment we are going to compare our heap sort implementation against the C++ standard library sort function `std::sort`.[1] We will count and compare the number of operations required, as done in the mandatory assignment, but also the wall-time—how much real-world time is needed to sort a slice.

The standard only mandates that the `std::sort` function has a run-time complexity of $\mathcal{O}(N \log N)$. As such the actual implementation of this function is up to your C++ standard library implementation, i.e. it is implementation defined. There are several popular C++ standard library implementations, such as:

- `libstdc++` (GCC/MinGW).
- `libc++` (LLVM/Clang[2]).
- `STL` (MSVC).

You are most likely using `libstdc++`, or `libc++` if you are using an Apple product. The implementation of all these libraries are open source, and as such we can inspect how they implement the `std::sort` function.[3] [4] [5]

Apparently these libraries all use a hybrid sorting algorithm called intro(spective) sort.[6] This algorithm is not a sorting algorithm in itself, but uses a combination of heap, quick, and insertion sort to sort data.

2 Assignment

Part 1: Understanding hybrid sorting

Give a high-level explanation of how introspective sort works, and how it guarantees a worst-case run-time complexity of $\mathcal{O}(N \log N)$. In addition, explain what the advantages of such a hybrid approach are over using either heap, quick, or insertion sort directly. Write your answers to a text file called "`bonus_assignment_10.txt`".

Part 2: Measuring number of operations

On Brightspace you can find "`assignment-10-bonus-files.zip`". It includes "`main.cpp`" with a `main` function that uses `generate_count_csv` to generate two files:

1. "`measurements_count_heapsort.csv`": The number of operations required to sort growing slices of our music databases using heap sort.
2. "`measurements_count_stdsort.csv`": The number of operations required to sort growing slices of our music databases using `std::sort`.

This function is defined as follows:

```
void generate_count_csv (
    const vector<Track>& tracks, const vector<Track>& sorted,
    const vector<Track>& random, const vector<Track>& reverse,
    ofstream& os, SortFunc sort)
```

[1] `https://en.cppreference.com/w/cpp/algorithm/sort`
[2] On Linux `libstdc++` is often used
[3] `https://github.com/gcc-mirror/gcc/blob/b1f91819e312d1e92d88a693718d791693cdf26c/libstdc%2B%2B-v3/include/bits/stl_algo.h#L1942`
[4] `https://github.com/llvm/llvm-project/blob/e31d27e46048ccc3294d6b215dc778b3390e7834/libcxx/include/__algorithm/sort.h#L627`
[5] `https://github.com/microsoft/STL/blob/60f18856c56389001c38a9929ef37bd5c01c4e47/stl/inc/algorithm#L8047`
[6] `https://en.wikipedia.org/wiki/Introsort`

where `tracks`, `sorted`, `random`, and `reverse` are the contents of the music databases used in the mandatory assignment, `os` the *.csv* file to write to, and `sort` the sorting function to use. Note that the `sort` parameter is a function parameter, as explained in the prior bonus assignment. This is a `void` function that accepts a single `vector<Track>&` parameter, which it will sort. This allows you to pass the actual sorting algorithm to use in `generate_count_csv` via a parameter, and call it as `sort(my_vector)`.

Implement the `generate_count_csv` function such that it generates a *.csv* file with the following format:

- the text `"database"`, followed by the slice numbers `500 1000 1500` … `6500`, all separated by a single `,` followed by `endl`;
- the text `"tracks"`, followed by the counts of sorting 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `tracks` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"sorted"`, followed by the counts of sorting 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `sorted` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"random"`, followed by the counts of sorting 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `random` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"reverse"`, followed by the counts of sorting 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `reverse` music database using `sort`, all separated by a single `,` followed by `endl`;

The counting of operations is done in exactly the same way as the mandatory assignment. Once you have your *.csv* files, save them to the same directory as your "`bonus_assignment_10.txt`" file, and convert them to *.png* chart files—as explained in the mandatory assignment—called:

- "`measurements_count_heapsort.png`"
- "`measurements_count_stdsort.png`"

Part 3: Measuring time

Besides measuring the number of operations required, we will also look at the actual amount of real-world time it takes to sort each slice. In order to measure time you can use the following snippet:

```
chrono::time_point start = chrono::high_resolution_clock::now();
// the code to measure
chrono::time_point end = chrono::high_resolution_clock::now();
cout << "This␣took␣" << chrono::duration_cast<chrono::microseconds>(end - start).count() << "␣us" << endl;
```

This will measure the number of microseconds[7] that have passed between the calls to `now`. The accuracy of this measurement depends on the resolution of the clock available on your system. Be aware that microseconds are an incredibly small time interval, and as such highly sensitive to background noise on your system. Make sure to close as many background programs as possible to avoid odd spikes in your measurements, and consider running it several times to see if you can obtain a representative measurement.[8]

Implement the `generate_time_csv` function such that it generates a *.csv* file with the following format:

- the text `"database"`, followed by the slice numbers `500 1000 1500` … `6500`, all separated by a single `,` followed by `endl`;
- the text `"tracks"`, followed by the time needed to sort 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `tracks` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"sorted"`, followed by the time needed to sort 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `sorted` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"random"`, followed by the time needed to sort 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `random` music database using `sort`, all separated by a single `,` followed by `endl`;
- the text `"reverse"`, followed by the time needed to sort 0–500, 0–1000, 0–1500, . . . , 0–6500 elements of the *unsorted* `reverse` music database using `sort`, all separated by a single `,` followed by `endl`;

Use the above snippet to measure how long each `sort` call takes, and output this as a plain integer representing the number of microseconds it took. Once you have your *.csv* files, save them to the same directory as your "`bonus_assignment_10.txt`" file, and convert them to *.png* chart files—as explained in the mandatory assignment—called:

- "`measurements_time_heapsort.png`"
- "`measurements_time_stdsort.png`"

[7] one millisecond is 1000 microseconds

[8] In a proper measurement setup we would repeat the measurement many times and take the median to eliminate outliers.

Part 4: Analysis

Use your results from parts 2 and 3 to analyze the differences between your heap sort implementation and `std::sort`. Document your analysis in "`bonus_assignment_10.txt`". Be sure to discuss the following aspects in your analysis:

- How does the database order impact the number of operations required by heap sort?
- How does the database order impact the number of operations required by `std::sort`?
- Is the time required to sort each slice consistent with the number of operations required by heap sort? Does the database order have any impact?
- Is the time required to sort each slice consistent with the number of operations required by `std::sort`? Does the database order have any impact?
- Based on the above points, is there a significant difference between the two algorithms?
- Do your measurements reflect the advantages you mentioned in part 1?

3 Products

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

- "`main.cpp`" that you have created with solutions for parts 2 and 3 of the assignment.
- "`bonus_assignment_10.txt`" with the discussions of parts 1 and 4.
- The four *.csv* files with raw measurement data generated in parts 2 and 3.
- The four *.png* files with charts created in parts 2 and 3.
- Any additional charts or measurement data used for your analysis in part 4.

Deadline

Bonus assignment: Monday November 27, 2023, 15:30h