INFO0004-2 : Project 1 (20%)
Sorting by prefix reversal

Submission before November 12

You will implement, in C++, two sorting algorithms that sort a vector of integers by prefix reversals only.

Sorting by prefix reversal is also known by the more popular name "pancake sorting": imagine that you have a stack of pancakes, where all the pancakes have different sizes. Your goal is to sort the pancakes, so that the largest one is at the bottom of the stack (the plate) and the smallest at the top of the stack. And the only way you have to move the pancakes within the stack is by inserting a spatula under any pancake you like, and flip over the (sub-)stack of pancakes above the spatula.

For example, if we represent the stack of pancakes as an vector on integers, with the top of the stack at index 0, then `[1, 4, 2, 0, 3]` becomes `[0, 2, 4, 1, 3]` by flipping over the top 4 pancakes.

To simplify, in this project, we will only consider stacks of unique pancakes (each pancake has a different size), where sizes range from 0 to $n-1$, if there are n pancakes.

1 Solution sketch

1.1 Simple approach

A simple approach to pancake sorting is to find the largest pancake in the stack that is not yet at its place, flip this pancake to the top of the stack, then flip that pancake in place.

For instance, in `[1, 4, 2, 0, 3]`, we can flip the largest pancake, not yet in its place, to the top of the stack by flipping the top two pancakes (`[4, 1, 2, 0, 3]`), then we can flip the largest pancake to its place, by flipping the whole stack (`[3, 0, 2, 1, 4]`).

We can then repeat this procedure for the next largest pancake.

1.2 Less flipping

We will use the "A* search algorithm", in the hope of sorting our pancakes with less flipping than in the simple approach above.

The idea, is that from a given stack, we can generate several “children” stacks, simply by flipping the “parent” stack at various places. As such, stacks of pancakes actually form a huge graph of stacks. But if we can somehow associate a cost with each stack we generate, we can then easily use a “best-first search” strategy , in order to only explore the parts of the stack graph that look promising.

More specifically, starting from the given stack, we can generate all of its children, and then pick the child with the smallest cost for further consideration and development, that is generate its children. Then, we can pick the stack with the smallest cost *among all of the generated stacks so far*, and so on, until we pick the goal stack which is the stack where all the pancakes are sorted.

By following back the “path” found from the goal to the start stack of pancakes, you can then reconstruct the sequence of flips needed to sort the start stack.

There are a few important considerations here. First, for the A* algorithm to work well (or even at all), the cost associated with successive stacks must meet specific properties. Without entering into the details, the following cost works well: consider the cost of a stack to be composed of a *real flip cost* and an *estimated end cost.* The real flip cost represents the cost of reaching the stack from the start stack, and can be computed by equating the cost of a flip to the number of pancakes flipped. As such, the real flip cost of a child stack is the real flip cost of its parent, plus the number of pancakes that must be flipped to transform the parent into the child.

The estimated end cost of a particular stack represents an estimate of the cost that must still be charged to this stack to reach the goal stack. Defining this cost as the size of the largest pancake that is not yet in its place in the stack, works well.

The *total cost* associated with a stack is then the sum of the real flip cost and the estimated end cost. It is that total cost that you should use to pick stacks to explore.

Second, the A* algorithm explores several paths from the start stack to the goal stack simultaneously. It is therefore important to properly keep track of which path leads to any given stack. This also means that a same stack can actually be generated several times during the exploration.

But third, you should try and persuade yourself that because costs accumulate along the exploration paths, once a stack has been picked for exploration (because it was the cheapest one so far), there is no point in ever pursuing exploration through that stack again later, as its later cost will necessarily be higher than when it was first explored.

Fourth, you will very probably want to use a priority queue, at the heart of your implementation, in order to easily find the cheapest stack for exploration, at each iteration of your exploration. The standard library does provide a `priority_queue` data structure, but beware that this container *only* provides access to its top element. On the other hand, if you need to also be able to search your priority queue data structure, the `set` data structure, provided by the standard library, gives the guarantee to internally keep its elements ordered, so the smallest element can be efficiently accessed through `set::begin()`.

Finally, it is important not to declare victory when you generate the goal stack, but only when you pick the goal stack from your priority queue. This is because another path could actually generate a cheaper goal later in the exploration.

2 Interface

Your two implementations of pancake sorting will be accessed through the following API:

```
typedef std::vector<int> stack_type;
typedef std::vector<stack_type::size_type> flip_type;

void simple_pancake_sort(const stack_type& pancakes, flip_type& flips);

void astar_pancake_sort(const stack_type& pancakes, flip_type& flips);
```

In this API, the `pancakes` parameter represents the stack to be sorted.

The `moves` parameter is a "result" parameter, and represents the sequence of flips needed to sort the stack. Each entry of this sequence is simply the index of the pancake "under which" the spatula was inserted for the flip. In other words, if you flip the top 2 pancakes, the corresponding entry in the sequence of flips should show the value 1.

You might be surprised to see that the given stack of pancakes is passed to your algorithm as a constant reference, but this is because we actually do know what the sorted stack will look like, and because we are actually more interested in the sequence of flips than the result of the sorting itself.

These function declarations, as well as the `typedef`s defining the `stack_type` and `flip_type` types, must be contained in a header file called `pancakes.hpp`. This file may also contain other lines of code, *but only the strict minimum required* to support the use of this interface.

All the code required for the implementation of this interface must be in a file called `pancakes.cpp`. Note that this file *must NOT* contain a `main` function.

You can consult the Standard C++ Library Reference for further documentation about the STL data structures and algorithms, at `http://www.cplusplus.com/reference/`.

3 Remarks

In this project, you **must NOT** define any new class or new structure. Doing so will result in a mark of 0.

Submission must scrupulously follow the interface (data structure, function signatures and return contracts, input format and code organisation) defined above. You can, nevertheless, define your own functions, as you see fit, *as long as* these are properly hidden from your users and do not interfere with the good operations of the interface specified above.

3.1 Readability

Your code must be readable. Use indentation, one of the common naming conventions for variable names, and comments.

3.2 Robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, programmes must run to completion without crash.

4 Evaluation

This project counts towards 20% of your final mark. In this project, you will be evaluated on: the organization of your code, usage of the STL, correctness and correct use of the `const` keyword, and efficiency of the solution.

5 Submission

Projects must be submitted before Sunday November 12, 23:59pm.

After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where N is the number of started days after the deadline).

You will submit your code (`pancakes.hpp` and `pancakes.cpp`) through the automated submission system. Your `pancakes.cpp` *must NOT* contain a `main` function.

Basic tests will be automatically run on your submissions and feedback will be automatically sent to you. This feedback is for information *only* and does not necessarily reflect your final mark. You can "submit" as often as you wish until the deadline. Submission instructions will be made available in due course, both via myULg and
`http://courses.run.montefiore.ulg.ac.be/cppoop/`.

Your programme must compile on the ms8** computers, without error or warning. Failure to compile will result in an awarded mark of 0. Warnings systematically result in lost marks.

Bon travail...