Sort

Sorting and verifying the sortedness of your data has been made very easy in C++. Let's find out how.

You can sort a range with `std::sort` or `std::stable_sort` or sort until a position with `std::partial_sort`. In addition `std::partial_sort_copy` copies the partially sorted range. With `std::nth_element` you can assign an element the *sorted* position in the range. You can check with `std::is_sorted` if a range is sorted. If you want to know until which position a range is sorted, use `std::is_sorted_until`.

Per default the predefined function object `std::less` is used a as sorting criterion. However, you can use your sorting criterion. This has to obey the strict weak ordering.

Sorts the elements in the range:

```
void sort(RaIt first, RaIt last)
void sort(ExePol pol, RaIt first, RaIt last)

void sort(RaIt first, RaIt last, BiPre pre)
void sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Sorts the elements in the range stable:

```
void stable_sort(RaIt first, RaIt last)
void stable_sort(ExePol pol, RaIt first, RaIt last)

void stable_sort(RaIt first, RaIt last, BiPre pre)
void stable_sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Sorts partially the elements in the range until `middle`:

```
void partial_sort(RaIt first, RaIt middle, RaIt last)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last)

void partial_sort(RaIt first, RaIt middle, RaIt last, BiPre pre)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last, BiPre pre)
```

Sorts partially the elements in the range and copies them in the destination ranges `result_first` and `result_last`:

```
RaIt partial_sort_copy(InIt first, InIt last, RaIt result_first, RaIt result_last)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,RaIt result_first, RaIt result_las

RaIt partial_sort_copy(InIt first, InIt last, RaIt result_first, RaIt result_last, BiPre pre)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,RaIt result_first, RaIt result_las
```

Checks if a range is sorted:

```
bool is_sorted(FwdIt first, FwdIt last)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last)

bool is_sorted(FwdIt first, FwdIt last, BiPre pre)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

Returns the position to the first element that doesn't satisfy the sorting criterion:

```
FwdIt is_sorted_until(FwdIt first, FwdIt last)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last)

FwdIt is_sorted_until(FwdIt first, FwdIt last, BiPre pre)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

Reorders the range, so that the n-th element has the right (sorted) position:

```
void nth_element(RaIt first, RaIt nth, RaIt last)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last)

void nth_element(RaIt first, RaIt nth, RaIt last, BiPre pre)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last, BiPre pre)
```

Here is a code snippet.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

int main(){

  std::cout << std::boolalpha << std::endl;
  std::string str{"RUdAjdDkaACsdfjwldXmnEiVSEZTiepfgOIkue"};


  std::cout << str <<  std::endl;

  std::cout << "std::is_sorted(str.begin(), str.end()): " <<  std::is_sorted(str.begin(), str

  std::cout << std::endl;

  std::partial_sort(str.begin(), str.begin() + 30, str.end());
  std::cout << str <<  std::endl;
  auto sortUntil=  std::is_sorted_until(str.begin(), str.end());
  std::cout << "Sorted until: " << *sortUntil << std::endl;
  for (auto charIt= str.begin(); charIt != sortUntil; ++charIt) std::cout << *charIt;

  std::cout << "\n\n";

  std::vector<int> vec{1, 0, 4, 3, 5};

  auto vecIt= vec.begin();
  while( vecIt != vec.end() ){
    std::nth_element(vec.begin(), vecIt++, vec.end());
    std::cout << std::distance(vec.begin(), vecIt) << "-th ";
    for (auto v: vec) std::cout << v;
    std::cout << std::endl;
  }

  std::cout << std::endl;

}
```

Sort algorithms