C++ Algorithmen

Numeric

```
void iota() {
  std::vector<int> out1(8);
  std::vector<int> expected{0, 1, 2, 3, 4, 5, 6, 7};

  std::iota(
  std::begin(out1),
  std::end(out1),
  0);

  ASSERT_EQUAL(expected, out1);
}

void accumulate() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6};
  int expected = 121;

  int res = std::accumulate(
    std::begin(in1),
    std::end(in1),
    100);

  ASSERT_EQUAL(expected, res);
}

void adjacent_difference() {
  std::vector<int> in1{1, 2, 4, 3, 9, 5, 7};
  std::vector<int> out1(in1.size());
  std::vector<int> expected{1, 1, 2, -1, 6, -4, 2};

  std::adjacent_difference(
  std::begin(in1),
  std::end(in1),
  std::begin(out1));

  ASSERT_EQUAL(expected, out1);
}

void partial_sum() {
  std::vector<int> in1{1, 20, 300, 4000, 50000};
  std::vector<int> out1(in1.size());
  std::vector<int> expected{1, 21,321,4321, 54321};

  std::partial_sum(std::begin(in1), std::end(in1),
      std::begin(out1));

  ASSERT_EQUAL(expected, out1);
}

void inner_product() {
  std::vector<int> in1{1, 2, 3, 2, 1};
  std::vector<char> in2{'a', 'b', 'c', 'd', 'e'};
  std::string expected{"begin, 1a, 2b, 3c, 2d, 1e"};

  std::string res = std::inner_product(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::string{"begin"},
    [](std::string l, std::string r) {return l + ", " + r;},
    [](int i, char c) {return std::to_string(i) + c;});

  ASSERT_EQUAL(expected, res);
}
```

Property Checking

```
void any_of() {
  std::vector<unsigned> in1{2, 3, 5, 6, 7};

  bool res = std::any_of(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT(res);
}


void is_permutation() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> in2{1, 5, 7, 4, 2, 6, 3, 8};
  auto expected = true;

  auto res = std::is_permutation(
    std::begin(in1),
    std::end(in1),
    std::begin(in2));

  ASSERT_EQUAL(expected, res);
}


void mismatch() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> in2{1, 2, 3, 4, 0, 6, 7, 8};
  auto expected = std::make_pair(std::begin(in1) + 4,
      std::begin(in2) + 4);

  auto res = std::mismatch(
    std::begin(in1),
    std::end(in1),
    std::begin(in2));

  ASSERT_EQUAL(expected, res);
}


void all_of() {
  std::vector<unsigned> in1{2, 3, 5, 6, 7};

  bool res = std::all_of(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT(!res);
}

void equal() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> in2{1, 2, 3, 4, 0, 6, 7, 8};
  auto expected = false;

  auto res = std::equal(
    std::begin(in1),
    std::end(in1),
    std::begin(in2));

  ASSERT_EQUAL(expected, res);
}


void none_of() {
  std::vector<unsigned> in1{1, 4, 6, 8, 9};

  bool res = std::none_of(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT(res);
}
```

Find

```
void search() {
  std::vector<int> in1{1, 2, 1, 2, 1, 2, 3, 1, 2, 3};
  std::vector<int> in2{1, 2, 3};
  auto expected = std::begin(in1) + 4;

  auto res = std::search(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2));

  ASSERT_EQUAL(expected, res);
}

void count() {
  std::vector<int> in1{1, 2, 3, 2, 1, 2, 3, 4, 3, 2};
  int expected = 4;

  int res = std::count(
    std::begin(in1),
    std::end(in1),
    2);

  ASSERT_EQUAL(expected, res);
}

void find() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7};
  auto expected = std::begin(in1) + 4;

  auto res = std::find(
```

```
    std::begin(in1),
    std::end(in1),
    5);

  ASSERT_EQUAL(expected, res);
}

void find_first_of() {
  std::vector<int> in1{5, 6, 4, 7, 6, 2, 1};
  std::vector<int> in2{1, 2, 3};
  auto expected = std::begin(in1) + 5;

  auto res = std::find_first_of(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2));

  ASSERT_EQUAL(expected, res);
}

void search_n() {
  std::vector<int> in1{1, 1, 2, 2, 2, 1, 1, 1, 3, 3};
  auto expected = std::begin(in1) + 5;

  auto res = std::search_n(
    std::begin(in1),
    std::end(in1),
    3,
    1);

  ASSERT_EQUAL(expected, res);
}

void find_if() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7};
  auto expected = std::begin(in1) + 1;

  auto res = std::find_if(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT_EQUAL(expected, res);
}

void adjacent_find() {
  std::vector<int> in1{5, 6, 4, 7, 7, 2, 2};
  auto expected = std::begin(in1) + 3;

  auto res = std::adjacent_find(
    std::begin(in1),
    std::end(in1));

  ASSERT_EQUAL(expected, res);
}

void find_if_not() {
  std::vector<int> in1{2, 3, 5, 7, 11, 13, 17};
  auto expected = std::end(in1);

  auto res = std::find_if_not(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT_EQUAL(expected, res);
}

void count_if() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int expected = 4;

  int res = std::count_if(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT_EQUAL(expected, res);
}

void find_end() {
  std::vector<int> in1{1, 2, 3, 1, 2, 3, 1};
  std::vector<int> in2{1, 2, 3};
  auto expected = std::begin(in1) + 3;

  auto res = std::find_end(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2));

  ASSERT_EQUAL(expected, res);
}
```

Copy/Replace

```
void transform() {
  std::vector<int> in1{5, 6, 7, 8, 0, 10};
  std::vector<int> out1{};
  std::vector<int> expected{32, 64, 128, 256, 1, 1024};

  std::transform(
  std::begin(in1),
  std::end(in1),
  std::back_inserter(out1),
  [](int i){return std::pow(2, i);});

  ASSERT_EQUAL(expected, out1);
}


void replace() {
  std::vector<int> in_out1{1, 2, 3, 2, 1, 2, 3, 2};
  std::vector<int> expected{1, 4, 3, 4, 1, 4, 3, 4};

  std::replace(
  std::begin(in_out1),
  std::end(in_out1),
  2,
  4);

  ASSERT_EQUAL(expected, in_out1);
}

void replace_copy() {
  std::vector<int> in1{1, 2, 3, 2, 1, 2, 3, 2};
  std::vector<int> out1{};
  std::vector<int> expected{1, 4, 3, 4, 1, 4, 3, 4};

  std::replace_copy(
  std::begin(in1),
  std::end(in1),
  std::back_inserter(out1),
  2,
  4);

  ASSERT_EQUAL(expected, out1);
}


void swap_ranges() {
  std::vector<int> in_out1{1, 2, 3, 4};
  std::vector<int> in_out2{5, 6, 7, 8};
  std::vector<int> expected1{5, 6, 7, 8};
  std::vector<int> expected2{1, 2, 3, 4};

  std::swap_ranges(
  std::begin(in_out1),
  std::end(in_out1),
  std::begin(in_out2));

  ASSERT_EQUAL(std::tie(expected1, expected2),
      std::tie(in_out1, in_out2));
}


void replace_if() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> expected{1, 0, 0, 4, 0, 6, 0, 8};

  std::replace_if(
  std::begin(in_out1),
  std::end(in_out1),
  is_prime,
  0);

  ASSERT_EQUAL(expected, in_out1);
}

void copy_backward() {
  std::vector<int> in_out1{5, 6, 3, 7, 4, 0, 0, 0};
  std::vector<int> expected{5, 6, 3, 5, 6, 3, 7, 4};

  std::copy_backward(
  std::begin(in_out1),
  std::begin(in_out1) + 5,
  std::end(in_out1));
```

```
  ASSERT_EQUAL(expected, in_out1);
}

void copy() {
  std::vector<int> in1{5, 6, 3, 7, 9, 1, 5};
  std::vector<int> out1{};
  std::vector<int> expected{5, 6, 3, 7, 9, 1, 5};

  std::copy(
  std::begin(in1),
  std::end(in1),
  std::back_inserter(out1));

  ASSERT_EQUAL(expected, out1);
}

void copy_if() {
  std::vector<int> in1{5, 6, 3, 7, 10, 10, 5};
  std::vector<int> out1{};
  std::vector<int> expected{5, 3, 7, 5};

  std::copy_if(
  std::begin(in1),
  std::end(in1),
  std::back_inserter(out1),
  [](int const & i) {return i % 2;});

  ASSERT_EQUAL(expected, out1);
}

void copy_n() {
  std::vector<int> in1{5, 6, 3, 7, 9, 1, 5};
  std::vector<int> out1{};
  std::vector<int> expected{5, 6, 3, 7, 9, 1};

  std::copy_n(
  std::begin(in1),
  6,
  std::back_inserter(out1));

  ASSERT_EQUAL(expected, out1);
}

void replace_copy_if() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> out1{};
  std::vector<int> expected{1, 0, 0, 4, 0, 6, 0, 8};

  std::replace_copy_if(
  std::begin(in1),
  std::end(in1),
  std::back_inserter(out1),
  is_prime,
  0);

  ASSERT_EQUAL(expected, out1);
}
```

Remove, Unique, Rotate

```
void reverse_copy() {
  std::vector<int> in1{5, 6, 7, 8, 0, 10};
  std::vector<int> out1{};
  std::vector<int> expected{10, 0, 8, 7, 6, 5};

  std::reverse_copy(
    std::begin(in1),
    std::end(in1),
    std::back_inserter(out1));

  ASSERT_EQUAL(expected, out1);
}

void prev_permutation() {
  std::vector<int> in_out1{4, 1, 2, 3};
  std::vector<int> expected{3, 4, 2, 1};

  std::prev_permutation(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void unique() {
  std::vector<int> in_out1{1, 1, 3, 3, 4, 2, 2, 2};
  std::vector<int> expected{1, 3, 4, 2};

  auto new_end = std::unique(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL_RANGES(std::begin(expected), std::end(expected),
      std::begin(in_out1), new_end);
}

void remove() {
  std::vector<int> in_out1{1, 2, 3, 2, 1, 2, 3, 2};
  std::vector<int> expected{1, 3, 1, 3};

  auto new_end = std::remove(
    std::begin(in_out1),
    std::end(in_out1),
    2);

  ASSERT_EQUAL_RANGES(std::begin(expected), std::end(expected),
      std::begin(in_out1), new_end);
}

void rotate_copy() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> out1{};
  std::vector<int> expected{5, 6, 7, 8, 9, 1, 2, 3, 4};

  std::rotate_copy(
    std::begin(in1),
    std::begin(in1) + 4,
    std::end(in1),
    std::back_inserter(out1));

  ASSERT_EQUAL(expected, out1);
}

void reverse() {
  std::vector<int> in_out1{5, 6, 7, 8, 0, 10};
  std::vector<int> expected{10, 0, 8, 7, 6, 5};

  std::reverse(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void next_permutation() {
  std::vector<int> in_out1{4, 1, 2, 3};
  std::vector<int> expected{4, 1, 3, 2};

  std::next_permutation(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void rotate() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> expected{5, 6, 7, 8, 9, 1, 2, 3, 4};

  std::rotate(
    std::begin(in_out1),
    std::begin(in_out1) + 4,
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}


void remove_if() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> expected{1, 4, 6, 8};

  auto new_end = std::remove_if(
    std::begin(in_out1),
    std::end(in_out1),
    is_prime);

  ASSERT_EQUAL_RANGES(std::begin(expected), std::end(expected),
      std::begin(in_out1), new_end);
}

void remove_copy_if() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> out1{};
  std::vector<int> expected{1, 4, 6, 8};

  std::remove_copy_if(
```

```
    std::begin(in1),
    std::end(in1),
    std::back_inserter(out1),
    is_prime);

  ASSERT_EQUAL(expected, out1);
}


void remove_copy() {
  std::vector<int> in1{1, 2, 3, 2, 1, 2, 3, 2};
  std::vector<int> out1{};
  std::vector<int> expected{1, 3, 1, 3};

  std::remove_copy(
    std::begin(in1),
    std::end(in1),
    std::back_inserter(out1),
    2);

  ASSERT_EQUAL(expected, out1);
}

void unique_copy() {
  std::vector<int> in1{1, 1, 3, 3, 4, 2, 2, 2};
  std::vector<int> out1{};
  std::vector<int> expected{1, 3, 4, 2};

  std::unique_copy(
    std::begin(in1),
    std::end(in1),
    std::back_inserter(out1));

  ASSERT_EQUAL(expected, out1);
}
```

Fill/Generate

```
void fill_n() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> expected{42, 42, 42, 42, 5, 6, 7, 8};

  std::fill_n(
    std::begin(in_out1),
    4,
    42);

  ASSERT_EQUAL(expected, in_out1);
}


void generate_n() {
  std::vector<int> out1{};
  std::vector<int> expected{100, 101, 102, 103, 104};
  int start = 100;

  std::generate_n(
    std::back_inserter(out1),
    5,
    [start]() mutable {return start++;});

  ASSERT_EQUAL(expected, out1);
}

void generate() {
  std::vector<int> out1(5);
  std::vector<int> expected{100, 101, 102, 103, 104};
  int start = 100;

  std::generate(
    std::begin(out1),
    std::end(out1),
    [start]() mutable {return start++;});

  ASSERT_EQUAL(expected, out1);
}

void fill() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
  std::vector<int> expected{42, 42, 42, 42, 42, 42, 42, 42};

  std::fill(
    std::begin(in_out1),
    std::end(in_out1),
    42);

  ASSERT_EQUAL(expected, in_out1);
}
```

Partition

```
void partition_point() {
  std::vector<int> in1{2, 3, 5, 7, 1, 4, 6, 8, 9};
  auto expected = std::begin(in1) + 4;

  auto res = std::partition_point(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT_EQUAL(expected, res);
}

void stable_partition() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> expected{2, 3, 5, 7, 1, 4, 6, 8, 9};

  std::stable_partition(
    std::begin(in_out1),
    std::end(in_out1),
    is_prime);

  ASSERT_EQUAL(expected, in_out1);
}

void is_partitioned() {
  std::vector<int> in1{2, 3, 5, 7, 1, 4, 6, 8, 9};

  bool res = std::is_partitioned(
    std::begin(in1),
    std::end(in1),
    is_prime);

  ASSERT(res);
}


void partition_copy() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> out1{};
  std::vector<int> out2{};
  std::vector<int> expected1{2, 3, 5, 7};
  std::vector<int> expected2{1, 4, 6, 8, 9};

  std::partition_copy(
    std::begin(in1),
    std::end(in1),
    std::back_inserter(out1),
    std::back_inserter(out2),
    is_prime);

  ASSERT_EQUAL(std::tie(expected1, expected2),
      std::tie(out1, out2));
}

void partition() {
  std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> expected{7, 2,  3, 5, 4, 6, 1, 8, 9};

  std::partition(
    std::begin(in_out1),
    std::end(in_out1),
    is_prime);

  ASSERT_EQUAL(expected, in_out1);
}
```

Sorting

```
void sort() {
  std::vector<int> in_out1{2, 3, 5, 7, 1, 4, 6, 8, 9};
  std::vector<int> expected{1, 2, 3, 4, 5, 6, 7, 8, 9};

  std::sort(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void partial_sort_copy() {
  std::vector<int> in1{2, 5, 3, 7, 1, 4, 6, 8, 9};
  std::vector<int> out1{0, 0, 0, 0, 0};
  std::vector<int> expected{1, 2, 3, 4, 5};

  std::partial_sort_copy(
    std::begin(in1),
    std::end(in1),
    std::begin(out1),
    std::end(out1));
```

```
  ASSERT_EQUAL(expected, out1);
}


void is_sorted() {
  std::vector<unsigned> in1{2, 3, 5, 6, 7};

  bool res = std::is_sorted(
    std::begin(in1),
    std::end(in1));

  ASSERT(res);
}


void partial_sort() {
  std::vector<int> in_out1{2, 5, 3, 7, 1, 4, 6, 8, 9};
  std::vector<int> expected{1, 2, 3, 4};

  std::partial_sort(
    std::begin(in_out1),
    std::begin(in_out1) + 4,
    std::end(in_out1));

  ASSERT_EQUAL_RANGES(std::begin(expected), std::end(expected),
      std::begin(in_out1), std::begin(in_out1) + 4);
}


void stable_sort() {
  std::vector<std::pair<int, int>> in_out1{{2, 1}, {1, 0}, {1, 2},
      {1, 4}, {2, 3}};
  std::vector<std::pair<int, int>> expected{{1, 0}, {1, 2}, {1, 4},
      {2, 1}, {2, 3}};

  std::stable_sort(
    std::begin(in_out1),
    std::end(in_out1),
    // Achtung nächste zwei Zeilen sind zusammen
    [](std::pair<int, int> l, std::pair<int, int> r)
        {return l.first < r.first;});

  ASSERT_EQUAL(expected, in_out1);
}


void nth_element() {
  std::vector<unsigned> in_out1{45, 27, 73, 15, 95,
      64, 44, 0, 99};

  std::nth_element(
    std::begin(in_out1),
    std::begin(in_out1) + 3,
    std::end(in_out1));

  ASSERT_EQUAL(44, *(std::begin(in_out1) + 3));
}
```

Sorted Sequence

```
void inplace_merge() {
  std::vector<int> in_out1{2, 3, 8, 9, 10, 16, 1, 3, 7, 13, 15};
  std::vector<int> expected{1, 2, 3, 3, 7, 8, 9, 10, 13, 15, 16};}

  std::inplace_merge(
    std::begin(in_out1),
    std::begin(in_out1) + 6,
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void binary_search() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};

  auto res = std::binary_search(
    std::begin(in1),
    std::end(in1),
    7);

  ASSERT(res);
}

void merge() {
  std::vector<int> in1{1, 3, 7, 13, 15};
  std::vector<int> in2{2, 3, 8, 9, 10, 16};
  std::vector<int> out{};
  std::vector<int> expected{1, 2, 3, 3, 7, 8, 9, 10, 13, 15, 16};

  std::merge(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2),
    std::back_inserter(out));

  ASSERT_EQUAL(expected, out);
}

void equal_range() {
  std::vector<unsigned> in1{1, 1, 1, 2, 2, 2, 3, 4, 4};
  auto expected = std::make_pair(std::begin(in1) + 3,
      std::begin(in1) + 6);

  auto res = std::equal_range(
    std::begin(in1),
    std::end(in1),
    2);

  ASSERT_EQUAL(expected, res);
}

void lower_bound() {
  std::vector<unsigned> in1{1, 1, 1, 2, 2, 2, 3, 4, 4};
  auto expected = std::begin(in1) + 3;

  auto res = std::lower_bound(
    std::begin(in1),
    std::end(in1),
    2);

  ASSERT_EQUAL(expected, res);
}

void upper_bound() {
  std::vector<unsigned> in1{1, 1, 1, 2, 2, 2, 3, 4, 4};
  auto expected = std::begin(in1) + 6;

  auto res = std::upper_bound(
    std::begin(in1),
    std::end(in1),
    2);

  ASSERT_EQUAL(expected, res);
}
```

Set

```
void set_intersection() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
  std::vector<int> out{};
  std::vector<int> expected{4, 5, 6, 9};

  std::set_intersection(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2),
    std::back_inserter(out));

  ASSERT_EQUAL(expected, out);
}

void set_difference() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
  std::vector<int> out{};
  std::vector<int> expected{1, 2, 3, 7, 8};

  std::set_difference(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2),
    std::back_inserter(out));

  ASSERT_EQUAL(expected, out);
}

void set_union() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
  std::vector<int> out{};
  std::vector<int> expected{1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12};

  std::set_union(
```

```
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2),
    std::back_inserter(out));

  ASSERT_EQUAL(expected, out);
}


void set_symmetric_difference() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
  std::vector<int> out{};
  std::vector<int> expected{1, 2, 3, 7, 8, 10, 11, 12};

  std::set_symmetric_difference(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2),
    std::back_inserter(out));

  ASSERT_EQUAL(expected, out);
}


void includes() {
  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::vector<int> in2{2, 3, 4, 7, 8, 9};

  auto res = std::includes(
    std::begin(in1),
    std::end(in1),
    std::begin(in2),
    std::end(in2));

  ASSERT(res);
}
```

Heap

```
void sort_heap() {
  std::vector<int> in_out1{10, 6, 9, 1, 2, 3, 5};
  std::vector<int> expected{1, 2, 3, 5, 6, 9, 10};

  std::sort_heap(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}


void push_heap() {
  std::vector<int> in_out1{9, 6, 5, 1, 2, 3, 10};
  std::vector<int> expected{10, 6, 9, 1, 2, 3, 5};

  std::push_heap(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void is_heap() {
  std::vector<int> in1{10, 6, 9, 1, 2, 3, 5};

  auto res = std::is_heap(
    std::begin(in1),
    std::end(in1));

  ASSERT(res);
}

void pop_heap() {
  std::vector<int> in_out1{10, 6, 9, 1, 2, 3, 5};
  std::vector<int> expected{9, 6, 5, 1, 2, 3, 10};

  std::pop_heap(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT_EQUAL(expected, in_out1);
}

void is_heap_until() {
  std::vector<int> in1{9, 6, 5, 1, 2, 3, 10};
  auto expected = std::begin(in1) + 6;

  auto res = std::is_heap_until(
    std::begin(in1),
    std::end(in1));

  ASSERT_EQUAL(expected, res);
}

void make_heap() {
  std::vector<int> in_out1{3, 1, 9, 2, 5, 6, 10};

  std::make_heap(
    std::begin(in_out1),
    std::end(in_out1));

  ASSERT(std::is_heap(std::begin(in_out1), std::end(in_out1)));
}
```

Min/Max

```
void min() {
  auto expected = 1;

  auto res = std::min({9, 6, 5, 1, 2, 10, 3, 8});

  ASSERT_EQUAL(expected, res);
}

void min_element() {
  std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
  auto expected = std::begin(in1) + 3;

  auto res = std::min_element(std::begin(in1), std::end(in1));

  ASSERT_EQUAL(expected, res);
}

void minmax_element() {
  std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
  auto expected = std::make_pair(std::begin(in1) + 3,
      std::begin(in1) + 5);

  auto res = std::minmax_element(std::begin(in1), std::end(in1));

  ASSERT_EQUAL(expected, res);
}

void max() {
  auto expected = 10;

  auto res = std::max({9, 6, 5, 1, 2, 10, 3, 8});

  ASSERT_EQUAL(expected, res);
}

void minmax() {
  auto expected = std::make_pair(1, 10);

  auto res = std::minmax({9, 6, 5, 1, 2, 10, 3, 8});

  ASSERT_EQUAL(expected, res);
}

void max_element() {
  std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
  auto expected = std::begin(in1) + 5;

  auto res = std::max_element(std::begin(in1), std::end(in1));

  ASSERT_EQUAL(expected, res);
}
```

Original auf
https://www.sharelatex.com/project/5876079a99a83e0a533601e6