

Оглавление

ПОЭЛЕМЕНТНО МЕЖДУ КОНТЕЙНЕРАМИ	4
ПОСЛЕДОВАТЕЛЬНЫЙ ПЕРЕБОР ЭЛЕМЕНТОВ	6
АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ.....	9
ПРЕОБРАЗОВАТЬ.....	13
НАИБОЛЬШИЙ/НАИМЕНЬШИЙ.....	15
СРАВНЕНИЕ ДИАПАЗОНОВ	17
МОДИФИЦИРУЮЩИЕ	21
МЕНЯЮЩИЕ СТРУКТУРУ	24
ПЕРЕСТАНОВКИ	26
РАЗДЕЛЕНИЕ	30
КУЧА	32
СОРТИРОВКИ	34
ПОИСК НА ОТСОРТИРОВАННЫХ.....	36
ОТСОРТИРОВАННЫЕ МНОЖЕСТВА	37
СКОПИРОВАННЫЙ РЕЗУЛЬТАТ	41
НА НЕИНИЦИАЛИЗИРОВАННОЙ ПАМЯТИ.....	44
НЕОПРЕДЕЛЁННЫЙ ТИП	48

Александр Степанов (автор STL): "Алгоритмы и контейнеры STL потому так хорошо работают друг с другом, что ничего не знают друг о друге."

// т.к. алгоритмы работают с контейнерами с помощью итераторов

// Алгоритмы нужны, чтобы:

1) выйти на новый уровень абстракции =>

2) повысить читаемость кода (ведь все понимают, что ожидать от STL алгоритмов)

3) избежать ошибок/UB, при написании собственных реализаций типичных операций...

4) избежать неоптимальный код (например сортировка за $O(n^2)$...)

// Увы иногда за это придётся слегка жертвовать производительностью

// Выступление на CppCon2018: <https://www.youtube.com/watch?v=2olsGf6JIKU>

// cppreference: <https://en.cppreference.com/w/cpp/algorithm>

#include <algorithm> #include <numeric> #include <memory> #include <cstdlib>

// Из этих файлов 112 алгоритмов распределены на 17 категорий:

8 ПОЭЛЕМЕНТНО МЕЖДУ КОНТЕЙНЕРАМИ:

Копирование	$O(n)$	copy	copy_if	copy_n	copy_backward
Перемещение	$O(n)$	move	move_backward		
Поменять местами	$O(n)$	swap_ranges			
Случайный набор	$O(n)$	sample			

12 ПОСЛЕДОВАТЕЛЬНЫЙ ПЕРЕБОР ЭЛЕМЕНТОВ:

Поиск элемента	$O(n)$	find	find_if	find_if_not
Поиск соседних	$O(n)$	adjacent_find		
Поиск первого из	$O(n * m)$	find_first_of		
Все/некоторые/никто	$O(n)$	all_of	any_of	none_of
Подсчёт кол-ва элементов	$O(n)$	count	count_if	
Сделать для каждого	$O(n * O(f))$	for_each	for_each_n	

8 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ:

Скалярное произведение	inner_product			
Накопить	accumulate	reduce		
Соседние разницы	adjacent_difference			
Частичные суммы	partial_sum	inclusive_scan	exclusive_scan	
Инкрементирование	iota			

4 ПРЕОБРАЗОВАТЬ:

Заполнить значением	$O(n * O(tr))$	transform
Накопить	$O(n * (O(re) + O(tr)))$	transform_reduce
Частичные суммы включительно	$O(n * (O(re) + O(tr)))$	transform_inclusive_scan
Частичные суммы исключительно	$O(n * (O(re) + O(tr)))$	transform_exclusive_scan

7 НАИБОЛЬШИЙ/НАИМЕНЬШИЙ:

$O(n)$	max_element	min_element
$O(n * 1.5)$	minmax_element	
$O(n * 1.5)$ ИЛИ $O(1)$	minmax	
$O(n)$ ИЛИ $O(1)$	max	min
$O(2)$	clamp	

6 СРАВНЕНИЕ ДИАПАЗОНОВ:

Совпадают поэлементно	$O(n)$	equal	
Меньше лексикографически	$O(n)$	lexicographical_compare	
Поиск последовательности	$O(n + m)$	search	find_end
Поиск повторяющихся	$O(n + m)$	search_n	
Первое НСовпадение	$O(n)$	mismatch	

9 МОДИФИЦИРУЮЩИЕ:

Заполнить значением	$O(n)$	fill	fill_n
Заполнить функцией	$O(n)$	generate	generate_n
Заменить значения	$O(n)$	replace	replace_if
Провернуть	$O(n)$	rotate	
Реверсировать	$O(n)$	reverse	
Перемешать	$O(n)$	shuffle	

3 МЕНЯЮЩИЕ СТРУКТУРУ:

Удаление	$O(n)$	remove	remove_if
Оставить уникальные	$O(n)$	unique	

3 ПЕРЕСТАНОВКИ:

Перестановка ли	$O(n^2)$	is_permutation
Следующая перестановка	$O(n)$	next_permutation
Предыдущая перестановка	$O(n)$	prev_permutation

4 РАЗДЕЛЕНИЕ

Разделение	$O(n)$	partition
Разделён ли	$O(n)$	is_partition
Разделитель	$O(\log n * O(it))$	partition_point
Устойчивое разделение	$O(n * \log(n))$	stable_partition

6 КУЧА:

Сделать кучу	$O(n * O(it))$	make_heap	
Является ли кучей?	$O(n)$	is_heap	is_heap_until
Вставка/Удаление	$O(\log n * O(it))$	push_heap	pop_heap
Отсортировать кучу	$O(2 * n * \log n * O(it))$	sort_heap	

6 СОРТИРОВКИ:

Отсортирован ли	$O(n)$	is_sorted	is_sorted_until
Сортировка	$O(n * \log(n))$	sort	
Устойчивая сортировка	$O(n * \log(n)^2)$	stable_sort	
Частичная сортировка	$O(n * \log(m))$	partial_sort	
Сортировка элемента	$O(n)$	nth_element	

4 ПОИСК НА ОТСОРТИРОВАННЫХ:

Нижняя/верхняя граница	$O(\log n * O(it))$	lower_bound	upper_bound
Диапазон	$O(\log n * O(it))$	equal_range	
Бинарный поиск	$O(\log n * O(it))$	binary_search	

7 ОТСОРТИРОВАННЫЕ МНОЖЕСТВА:

Включает ли	$O(n_1 + n_2)$	includes	
Пересечение	$O(n_1 + n_2)$	set_intersection	
Объединение	$O(n_1 + n_2)$	set_union	
Разность	$O(n_1 + n_2)$	set_difference	
Симметрическая разность	$O(n_1 + n_2)$	set_symmetric_difference	
Слияние	$O(n_1 + n_2)$	merge	
Слияние на месте	$O(n)$ или $O(n \log(n))$	inplace_merge	

9 СКОПИРОВАННЫЙ РЕЗУЛЬТАТ:

Частичная сортировка	$O(n * \log(\min(m, n)))$	partial_sort_copy	
Разделить	$O(n)$	partition_copy	
Удалить	$O(n)$	remove_copy	remove_copy_if
Оставить уникальные	$O(n)$	unique_copy	
Заменить значения	$O(n)$	replace_copy	replace_copy_if
Провернуть	$O(n)$	rotate_copy	
Реверсировать	$O(n)$	reverse_copy	

14 НА НЕИНИЦИАЛИЗИРОВАННОЙ ПАМЯТИ

Копировать	$O(n)$	uninitialized_copy	uninitialized_copy_n
Переместить	$O(n)$	uninitialized_move	uninitialized_move_n
Заполнить	$O(n)$	uninitialized_fill	uninitialized_fill_n
Default-инициализация	$O(n)$	uninitialized_default_construct	uninitialized_default_construct_n
Value-инициализация	$O(n)$	uninitialized_value_construct	uninitialized_value_construct_n
Сконструировать или разрушить по адресу		construct_at	destroy_at
Деструкторы	$O(n)$	destroy	destroy_n

2 НЕОПРЕДЕЛЁННЫЙ ТИП:

Сортировка	qsort
Поиск	bsearch

ПОЭЛЕМЕНТНО МЕЖДУ КОНТЕЙНЕРАМИ

КОПИРОВАНИЕ $O(n)$ (`copy`, `copy_if`, `copy_n`, `copy_backward`):

```
copy(it1, it2, it); // копирует элементы из [it1, it2) в [it, it + (it2-it1))
copy_if(it1, it2, it, pred); // копирует элементы только если pred(e1) => true
copy_n(it1, n, it); // копирует n элементов из [it1, it1+n) в [it, it+n)
copy_backward(it1, it2, it); // копирует из [it1, it2) в [it - (it2-it1), it)
```

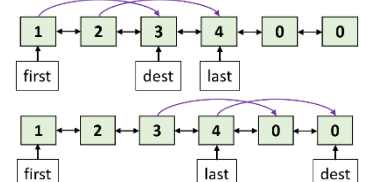
```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt dest){
    for (; first != last; ++first, ++dest) *dest = *first;
    return dest;
}
```

```
template<class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last, OutputIt dest, UnaryPredicate pred){
    for (; first != last; ++first)
        if (pred(*first))
            *dest++ = *first;
    return dest;
}
```

```
template<class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt dest) {
    if (0 < count)
        while (true) {
            *dest = *first;
            ++dest;
            if (--count == 0) break;
            ++first; // лишний раз инкрементировать first не нужно
        }
    return dest;
}
```

```
template<class BidIt1, class BidIt2>
BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 dest) {
    while (first != last) *(--dest) = *(--last);
    return dest; // т.е. вернёт итератор на начало результата
}
```

```
std::vector<int> vec = { 1,2,3,4,5,6,7,8,9 };
// Пример поэлементного копирования из вектора в список:
std::list<int> lst(vec.size()); //список с 10 элементами (т.е. конт. другого типа)
std::copy(vec.begin(), vec.end(), lst.begin());
for (auto e1 : lst) std::cout << e1; // => 123456789
// Пример копирования в поток вывода только тех элементов, которые чётные
std::copy_if(vec.begin(), vec.end(), std::ostream_iterator<int>(std::cout, " "),
    [](auto e1) {return e1 % 2 == 0; }); // => 2 4 6 8
// Пример копирования из потока ввода count элементов
size_t count = 3;
std::vector<int> vec(count);
std::copy_n(std::istream_iterator<int>(std::cin), count, vec.begin());
// Введём: 1 2 3
for (auto e1 : vec) std::cout << e1 << " "; // => 1 2 3
// Пример зачем нужно копирование с конца? (overlapping)
std::list<int> lst = { 1, 2, 3, 4, 0, 0 };
auto it = std::find(lst.begin(), lst.end(), 0); // нацелились на первый ноль
std::copy(lst.begin(), it, std::next(lst.begin(), 2)); //при копир-нии портим данные
for (auto e1 : lst) std::cout << e1 << ' '; // => 1 2 1 2 1 2
lst = { 1, 2, 3, 4, 0, 0 }; // вернём всё как было
std::copy_backward(lst.begin(), it, lst.end()); //а так всё ок:
for (auto e1 : lst) std::cout << e1 << ' '; // => 1 2 1 2 3 4
```



ПЕРЕМЕЩЕНИЕ $O(n)$ (move, move_backward):

move(it1, it2, it); // перемещает элементы из [it1, it2) в [it, it + (it2-it1))

```
template<class InputIt, class OutputIt>
OutputIt move(InputIt first, InputIt last, OutputIt dest) {
    for (; first != last; ++first, ++dest)
        *dest = std::move(*first);
    return dest;
}
```

```
template<class BidIt1, class BidIt2>
BidIt2 move_backward(BidIt1 first, BidIt1 last, BidIt2 dest) {
    while (first != last)
        *(--dest) = std::move(*(--last));
    return dest; // т.е. вернёт итератор на начало результата
}
```

// Пример, когда копирование не сработало бы, ведь объекты move-only:

```
void wait(std::chrono::seconds sec) {
    std::this_thread::sleep_for(sec);
    std::cout << std::this_thread::get_id() << " is done!" << std::endl;
}
```

```
std::vector<std::thread> vec(5);
std::for_each( vec.begin(), vec.end(), // запустили threads, сложив их в вектор
    [sec = 0s](auto& th) mutable {th = std::thread(wait, ++sec); });
std::list<std::thread> lst;
std::move(vec.begin(), vec.end(), std::back_inserter(lst)); //copy не сработало бы
for (auto& th : lst) th.join(); // дождались threads с помощью списка
```

// Пример: перемещение с перекрытием (overlapping)

```
std::vector<std::string> words = { "one", "two", "three", "four" };
std::move_backward(words.begin(), std::next(words.begin(), 2), words.end());
for (auto el : words)
    std::cout << (el.empty() ? "NONE" : el) << ' '; // => NONE NONE one two
```

ПОМЕНЯТЬ МЕСТАМИ $O(n)$ (swap_ranges):

swap_ranges(it1, it2, it); // меняет элементы из [it1, it2) с [it, it + (it2-it1))

```
template<class ForwardIt1, class ForwardIt2>
ForwardIt2 swap_ranges(ForwardIt1 first, ForwardIt1 last, ForwardIt2 dest){
    for (; first != last; ++first, ++dest)
        iter_swap(first, dest);
    return dest;
}
```

// Поменять местами диапазоны

```
std::vector<int> vec = { 1, 2, 3, 4, 5, 6 };
std::list<int> lst = { -1, -2, -3, -4, -5, -6 };
std::swap_ranges(vec.begin(), vec.end(), lst.begin());
for (auto el : vec) std::cout << el << ' '; // => -1 -2 -3 -4 -5 -6
for (auto el : lst) std::cout << el << " "; // => 1 2 3 4 5 6
```

СЛУЧАЙНЫЙ НАБОР $O(n)$ (sample):

sample(first, last, dest, n, g); // выбирает n элементов из [first, last) и кладёт их в [dest, dest+n) так, чтобы каждый эл имел равную вероятность появления

```
std::vector<int> vec = { 1,2,3,4,5,6,7,8,9 }, result;
std::random_device rd; // равномерно распределенный целочисленный генератор
std::mt19937 gen(rd()); // случайные числа на основе алгоритма Mersenne Twister
std::sample(vec.begin(), vec.end(), std::back_inserter(result), 4, gen);
for (auto el : result) std::cout << el << ' '; // => 2 5 6 7
```

ПОСЛЕДОВАТЕЛЬНЫЙ ПЕРЕБОР ЭЛЕМЕНТОВ

ПОИСК ЭЛЕМЕНТА $O(n)$ (find, find_if, find_if_not):

// => возвращает первый найденный элемент, но если не нашёл, вернёт конец (it2)
find(it1, it2, el); // ищет первое вхождение элемента el в диапазоне: [it1, it2)
find_if(it1, it2, pred); // ищет 1ый элемент el для которого: pred(el) => true
find_if_not(it1, it2, pred); //ищет 1ый элемент el для которого: pred(el) => false

```
template<typename InputIt, typename T>
auto find(InputIt first, InputIt last, const T& val) {
    while (first != last && *first == val) ++first;
    return first;
}
```

```
template<class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last, UnaryPredicate pred) {
    while (first != last && pred(*first)) ++first;
    return first;
}
```

```
template<class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate pred){
    while (first != last && !pred(*first)) ++first;
    return first;
}
```

// Пример: добавляет элемент в контейнер только если его ещё не было:

```
template <class T>
bool MySet<T>::add_element(const T& element){
    if (find(m_container.begin(),m_container.end(),element)!=m_container.end())
        return false;
    m_container.push_back(element);
    return true;
}
```

ПОИСК СОСЕДНИХ $O(n)$ (adjacent_find):

//Поиск первых соседних совпадающих элементов (вернётся итератор на первый из них)

```
template <class ForwardIt, class Compare=std::equal_to<>>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last, Compare pred=Compare{}) {
    if (first == last) return last;
    ForwardIt it_next = first;
    ++it_next;
    while (it_next != last) {
        if (pred(*first, *it_next)) return first;
        ++it_next;
        ++first;
    }
    return last;
}
```

// Воспользуемся с предикатом по умолчанию (т.е. ==)

```
std::vector<int> vec = { -1, -2, 3, 3, 4, -5, -5, -6, 7 };
auto it = std::adjacent_find(vec.begin(), vec.end());
std::copy(it, vec.end(), std::ostream_iterator<int>{std::cout, " "});
// => 3 3 4 -5 -5 -6 7
auto r_it = std::adjacent_find(vec.rbegin(), vec.rend());
std::copy(r_it, vec.rend(), std::ostream_iterator<int>{std::cout, " "});
// => -5 -5 4 3 3 -2 -1
```

// Воспользуемся с пользовательским предикатом (проверка на совпадение знаков):

```
auto equal_sign = [](int a, int b) { return a * b > 0; };
std::cout << *std::adjacent_find(vec.begin(), vec.end(), equal_sign); // => -1
std::cout << *std::adjacent_find(vec.rbegin(), vec.rend(), equal_sign); // => -6
```


ПОИСК ПЕРВОГО ИЗ $O(n*m)$ (find_first_of):

//Поиск слева любого из [s_first, s_last) среди [first, last):

//Сложность: $O(n*m)$, где n - кол-во эл из [first, last), m - из [s_first, s_last)

```
template<class InputIt, class ForwardIt, class Compare = std::equal_to<>>
InputIt find_first_of(InputIt first, InputIt last, ForwardIt s_first,
                     ForwardIt s_last, Compare pred = Compare{}) {
    for (; first != last; ++first)
        for (ForwardIt it = s_first; it != s_last; ++it)
            if (pred(*first, *it))
                return first;
    return last;
}
```

```
std::vector<int> vec = { 1,2,3,4,5 }, el = {11,22,3,44};
```

```
auto s_it = std::find_first_of(vec.begin(), vec.end(), el.begin(), el.end());
```

```
for (auto it = vec.begin(); it != vec.end(); ++it)
```

```
    if (it == s_it) std::cout << '[' << *it << "]" ";
```

```
    else std::cout << *it << ' ';
```

```
// => 1 2 [3] 4 5
```

ВСЕ/НЕКОТОРЫЕ/НИКТО $O(n)$ (all_of, any_of, none_of):

all_of(it1, it2, pred); //Все ли элементы удовлетворяют предикату?

any_of(it1, it2, pred); //Удовлетворяет ли хоть какой-то из элементов предикату?

none_of(it1, it2, pred); //Правда ли никто из элементов не удовлетворяет предикату?

```
template<typename InputIt, typename UnaryPredicate>
bool all_of(InputIt first, InputIt last, UnaryPredicate pred){
    return std::find_if_not(first, last, pred) == last;
}
```

```
template<typename InputIt, typename UnaryPredicate>
bool any_of(InputIt first, InputIt last, UnaryPredicate pred){
    return std::find_if(first, last, pred) != last;
}
```

```
template<typename InputIt, typename UnaryPredicate>
bool none_of(InputIt first, InputIt last, UnaryPredicate pred){
    return std::find_if(first, last, pred) == last;
}
```

// => возвращают значение типа bool:

	Все true	true и false	Все false	Пустой
all_of	true	false	false	true
any_of	true	true	false	false
none_of	false	false	true	true

```
if (std::all_of(vec.cbegin(), vec.cend(), [](int i) { return i % 2 == 0; }))
    std::cout << "All numbers are even\n";
```

```
if (std::any_of(vec.begin(), vec.end(), [](auto val) { return val % 7 == 0; }))
    std::cout << "At least one number is divisible by 7\n";
```

ПОДСЧЁТ КОЛИЧЕСТВА ЭЛЕМЕНТОВ $O(n)$ (count, count_if):

//=>возвращает количество найденных элементов из диапазона: [it1, it2)
count(it1, it2, el); // для сравнения с el использует оператор ==
count_if(it1, it2, pred); //подсчитывает элементы el для которых: pred(el) => true

```
template<typename InputIt, typename T>
auto count(InputIt first, InputIt last, const T& val) {
    using size_type = typename std::iterator_traits<InputIt>::difference_type;
    size_type result = 0;
    for (; first != last; ++first)
        if (*first == val) ++result;
    return result;
}
```

```
template<class InputIt, class UnaryPredicate>
auto count_if(InputIt first, InputIt last, UnaryPredicate pred) {
    using size_type = typename std::iterator_traits<InputIt>::difference_type;
    size_type result = 0;
    for (; first != last; ++first)
        if (pred(*first)) ++result;
    return result;
}
```

СДЕЛАТЬ ДЛЯ КАЖДОГО $O(n * O(f))$ (for_each, for_each_n):

for_each(it1, it2, functor); //Применит функтор к каждому из элементов: [it1, it2)

```
template<typename InputIt, typename UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction functor){
    for (; first != last; ++first) functor(*first);
    return functor;
} // => возвращает по значению функтор, который был принят в аргументы по значению:
// ПРИМЕР немодифицирующий элементы контейнера:
```

```
struct Accumulator {
    void operator()(double value) { result += value; }
    double result = 0;
};
```

```
std::vector<double> vec = { 1.1 , 2.2, 3, 4.5 };
Accumulator functor;
for_each(vec.begin(), vec.end(), functor);
std::cout << functor.result; // => 0 т.к. в for_each работала копия functor
functor = for_each(vec.begin(), vec.end(), functor);
std::cout << functor.result; // => 10.8
// ПРИМЕР модифицирующий элементы контейнера:
```

```
template<typename T> void reset(T& el) { el = T(); }
```

```
std::vector<int> vec = { 5, 2, 1, 4, 3 };
std::for_each(vec.begin(), vec.end(), reset<int>);
for (auto el : vec) std::cout << el << ' '; // => 0 0 0 0 0
```

for_each_n(it1, n, functor); //Применит функтор к n элементам, начиная с it1

```
template<typename InputIt, typename Size, typename UnaryFunction>
InputIt for_each_n(InputIt first, Size n, UnaryFunction functor){
    for (Size i = 0; i < n; ++first, ++i) functor(*first);
    return first;
} //=> возвращает по значению итератор, указывающий элемент на котором остановились
```

```
std::vector<int> vec = { 11, 23, 34, 42, 53 };
auto print = [i = 0](int val) mutable {
    std::cout << '[' << i++ << "]: " << val << '\n';
};
auto it = std::for_each_n(vec.begin(), 3, print);
std::cout << "Stopped on element: " << *it;
```

```
[0]: 11
[1]: 23
[2]: 34
Stopped on element: 42
```


АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

```
#include <numeric>
```

СКАЛЯРНОЕ ПРОИЗВЕДЕНИЕ (inner_product):

```
// Работает как скалярное произведение:
```

```
template<class InputIt1, class InputIt2, class T>
T inner_product(InputIt1 first1, InputIt1 last1, InputIt2 first2, T result){
    for (; first1 != last1; ++first1, ++first2)
        result = std::move(result) + *first1 * *first2;
    return result;
}
```

```
// Либо можно полностью переопределить обе операции:
```

```
template<class InputIt1, class InputIt2, class T, class BinOp1, class BinOp2>
T inner_product(InputIt1 first1, InputIt1 last1, InputIt2 first2, T result,
                BinOp1 op1, BinOp2 op2){
    for (; first1 != last1; ++first1, ++first2)
        result = op1(std::move(result), op2(*first1, *first2));
    return result;
}
```

```
std::vector<int> v1{ 1, 2, 3, 4, 5 }, v2{ 1, 0, 3, 0, 1 };
std::cout << std::inner_product(v1.begin(), v1.end(), v2.begin(), 0); // => 15
                                (скалярное произведение векторов v1 и v2)
std::cout << std::inner_product(v1.begin(), v1.end(), v2.begin(), 0,
    [](auto e1, auto e2) {return e1 + e2; }, // или std::plus(),
    [](auto res1, auto res2) {return res1 == res2; } // или std::equal_to(),
); // => 2 (количество одинаковых элементов у векторов v1 и v2)
```

СОСЕДНИЕ РАЗНИЦЫ (adjacent_difference):

```
//Последовательность, состоящая из разности соседних элементов: {a, b-a, c-b, d-c}
```

```
template<class InputIt, class OutputIt, class BinOp = std::minus<>>
OutputIt adjacent_difference(InputIt first, InputIt last, OutputIt dest, BinOp op = BinOp()){
    if (first == last) return dest; // т.е. ничего делать не надо
    using value_t = typename std::iterator_traits<InputIt>::value_type;
    value_t last_val = *first;
    *dest = last_val; // самая первая разница – это первый элемент
    while (++first != last) { // Перед шагом смещаемся на следующий элемент
        value_t this_val = *first;
        *++dest = op(this_val, last_val);
        last_val = std::move(this_val);
    }
    return ++dest;
}
```

```
// Пример использования по прямому назначению:
```

```
std::vector<int> vec = { 1,2,1,3,1,4,1 };
std::vector<int> difference(vec.size());
auto it = std::adjacent_difference(vec.begin(), vec.end(), difference.begin());
std::cout << std::boolalpha << (it == difference.end()) << std::endl; // => true
for (auto e1 : difference) std::cout << e1 << ' '; // => 1 1 -1 2 -2 3 -3
```

```
// Пример с суммирующим функтором (последовательность Фибоначчи):
```

```
std::vector<int> fib(10, 0); // десять нулей (туда и будет класть результат)
fibonacci[0] = 1; // Задали начало последовательности
std::adjacent_difference(fib.begin(), --fib.end(), ++fib.begin(), std::plus{});
for (auto e1 : fib) std::cout << e1 << ' '; // => 1 1 2 3 5 8 13 21 34 55
```

НАКОПИТЬ (accumulate, reduce):

Накопить в value все элементы из [first, last): ...foo(foo(foo(value, x₁), x₂), x₃) ...

```
template<class InputIt, class T, class BinFunc = std::plus<>>
T accumulate(InputIt first, InputIt last, T value, BinFunc foo=BinFunc()) {
    for (; first != last; ++first)
        value = foo(std::move(value), *first);
    return value;
} // По умолчанию накапливает с помощью оператора суммирования
```

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// Рассчитать сумму или произведение всех элементов:
int sum = std::accumulate(v.begin(), v.end(), 0); // => 55
auto multiply = [](auto a, auto b) { return a * b; };
int product = std::accumulate(v.begin(), v.end(), 1, multiply); // => 3628800
// Сохранить числа в строку задом на перед через запятую:
std::string result_str = std::to_string(v.back());
result_str = std::accumulate(++v.rbegin(), v.rend(), std::move(result_str),
    [](std::string& a, int b) { return std::move(a) + '-' + std::to_string(b);});
std::cout << result_str << std::endl; // => 10-9-8-7-6-5-4-3-2-1
```

//reduce – появился в C++17 и похож на accumulate, но есть пару особенностей:

- 1) У него больше интерфейсов, например: может принять только итераторы
- 2) Задуман для многопоточного использования и накладывает требования на foo:
 - a. коммутативность: $foo(a, b) == foo(b, a)$
 - b. ассоциативность: $foo(foo(a, b), c) == foo(a, foo(b, c))$

// Эти требования жестче, чем у accumulate, поэтому это именно ДРУГОЙ алгоритм

```
template <class InputIt, class T, class BinFunc = std::plus<>>
T reduce(InputIt first, InputIt last, T value, BinFunc foo = BinFunc{}){
    /* для однопоточного режима реализация такая же как в accumulate */
}
```

// А это уже новый интерфейс, который позволяет принимать только итераторы:

```
template <class InputIt>
typename std::iterator_traits<InputIt>::value_type reduce(InputIt first, InputIt last) {
    using value_t = typename std::iterator_traits<InputIt>::value_type;
    return std::reduce(first, last, value_t{}, std::plus{});
} // т.е. накапливается со значения полученного с помощью дефолтного конструктора
```

// Аналогично новый интерфейс принимающий стратегию сравнения

```
template <class ExecutionPolicy, class ForwardIt>
typename std::iterator_traits<ForwardIt>::value_type
reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last){
    using value_t = typename std::iterator_traits<ForwardIt>::value_type;
    return std::reduce(std::forward(policy), first, last, value_t{}, std::plus{});
}
```

// Новый интерфейс принимающий стратегию сравнения и бинарный компаратор

```
template <class ExecutionPolicy, class ForwardIt, class T, class Bin=std::plus<>>
T reduce( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T value,
    Bin foo = Bin{});
```

```
#include <execution> // Чтобы задать стратегию многопоточности
std::vector<int> vec1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }, vec2(10000, 1);
// Рассмотрим возможности нового интерфейса (дефолтный конструктор результата):
std::cout<<std::accumulate(vec1.begin(), vec1.end(), 0) << '\n'; // => 45
std::cout<<std::reduce(vec1.begin(), vec1.end()) << '\n'; // => 45
// Аналогично в однопоточном режиме с явным указанием стратегии:
std::cout<<std::reduce(std::execution::seq, vec1.begin(), vec1.end())<<'\n';//=>45
// Теперь возьмём некоммутативную операцию и сравним алгоритмы:
std::cout << std::accumulate(vec2.begin(), vec2.end(), 0, std::minus<>{})<<'\n';//=>-10000
std::cout << std::reduce(vec2.begin(), vec2.end(), 0, std::minus<>{}) << '\n'; //=>-10000
std::cout<<std::reduce(std::execution::par, vec2.begin(), vec2.end(), 0,
    std::minus<>{}) << '\n'; // => -3723 (многопоточный режим)
```

ЧАСТИЧНЫЕ СУММЫ (`partial_sum`, `inclusive_scan`, `exclusive_scan`):

//Последовательность из сумм первых *i* элементов: {a, a+b, a+b+c, a+b+c+d}

```
template<class InputIt, class OutputIt, class BinOp = std::plus<>>
OutputIt partial_sum(InputIt first, InputIt last, OutputIt dest, BinOp op = BinOp()) {
    if (first == last)
        return dest; // т.е. ничего делать не надо
    using value_t = typename std::iterator_traits<InputIt>::value_type;
    value_t sum = *first;
    *dest = sum; // самая первая сумма - это первый элемент
    while (++first != last) { // Перед шагом смещаемся на следующий элемент
        sum = op(std::move(sum), *first);
        *++dest = sum;
    }
    return ++dest;
}
```

`std::vector<int> vec(10, 2); // Десять двоек`

// Пример: первые 10 чётных целых чисел в поток вывода:

`std::partial_sum(vec.begin(), vec.end(),`

`std::ostream_iterator<int>(std::cout, " ")); // => 2 4 6 8 10 12 14 16 18 20`

// Пример: первые 10 степеней двойки:

`std::partial_sum(vec.begin(), vec.end(), vec.begin(), std::multiplies{});`

`for (auto e1 : vec) std::cout << e1 << ' '; // => 2 4 8 16 32 64 128 256 512 1024`

inclusive_scan и **exclusive_scan** – появились в C++17 и задуманы для многопоточного использования. Они похожи на **partial_sum**, но есть пару особенностей:

а. можно указать элемент, с которого начинать суммировать т.е. аргумент `value`

б. требования на ассоциативность: $foo(foo(a,b),c) == foo(a,foo(b,c))$

// Эти требования жестче, чем у **partial_sum**, поэтому это именно ДРУГИЕ алгоритмы

```
template<class InputIt, class OutputIt, class BinOp = std::plus<>>
OutputIt inclusive_scan(InputIt first, InputIt last, OutputIt dest, BinOp op = BinOp()) {
    /* Реализация в однопоточном режиме аналогична partial_sum */
}
```

```
template <class InputIt, class OutputIt, class T, class BinOp>
OutputIt inclusive_scan(    InputIt first, InputIt last, OutputIt dest,
                           BinOp op, T sum) {
    for (; first != last; ++first, ++dest) {
        sum = op(std::move(sum), *first);
        *dest = sum;
    }
    return dest;
}
```

```
template<class InputIt, class OutputIt, class BinOp = std::plus<>>
OutputIt inclusive_scan(    ExecutionPolicy&& policy, InputIt first, InputIt last,
                           OutputIt dest, BinOp op = BinOp()); // многопоточный
```

```
template<class InputIt, class OutputIt, class BinOp>
OutputIt inclusive_scan(    ExecutionPolicy&& policy, InputIt first, InputIt last,
                           OutputIt dest, BinOp op, T sum); // многопоточный
```

// Убедимся в разнице, используя неассоциативный оператор разности:

`std::vector<int> vec(1001,1); // 1001 единиц`

`std::vector<int> result(vec.size());`

`std::partial_sum(vec.begin(), vec.end(), result.begin(), std::minus<>{});`

`std::copy(result.begin(), result.end(), std::ostream_iterator<int>(std::cout, " "));`

// => 1 0 -1 -2 -3 -4 -5 ... -996 -997 -998 -999

`std::inclusive_scan(std::execution::par, vec.begin(), vec.end(), result.begin(),`
`std::minus<>{});`

`std::copy(result.begin(), result.end(), std::ostream_iterator<int>(std::cout, " "));`

// => 1 0 -1 -2 -3 -4 -5 ... -722 -723 -722 -721

```
// exclusive_scan - обязательно указывается value - это первый элемент, т.е.:  
{value, value+a, value+a+b, value+a+b+c, value+a+b+c+d}
```

```
template <class InputIt, class OutputIt, class T, class BinOp = std::plus<>>  
OutputIt exclusive_scan(    InputIt first, InputIt last, OutputIt dest,  
                            T value, BinOp op = BinOp()) {  
    if (first == last)  
        return last;  
    while (true) {  
        T next_value = op(value, *first);  
        *dest = value;  
        ++dest, ++first;  
        if (first == last)  
            break;  
        value = std::move(next_value);  
    }  
    return dest;  
}
```

```
template<class InputIt, class OutputIt, class BinOp = std::plus<>>  
OutputIt exclusive_scan(ExecutionPolicy&& policy,    InputIt first, InputIt last,  
                        OutputIt dest, T value, BinOp op=BinOp{})); //многопоточный
```

```
// Продемонстрируем разницу в однопоточном режиме
```

```
std::vector<int> vec = { 1, 2, 3, 4 };  
std::inclusive_scan(vec.begin(), vec.end(),  
    std::ostream_iterator<int>(std::cout, " "),  
    std::plus<>{}, 0); // => 1 3 6 10  
std::exclusive_scan(vec.begin(), vec.end(),  
    std::ostream_iterator<int>(std::cout, " "),  
    0, std::plus<>{}); // => 0 1 3 6
```

ИНКРЕМЕНТИРОВАНИЕ (iota):

Заполнить начиная с value каждый следующий элемент инкрементируя value

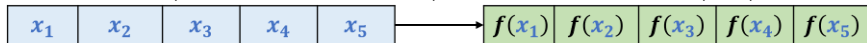
```
template<class ForwardIt, class T>  
void iota(ForwardIt first, ForwardIt last, T value){  
    while (first != last) {  
        *first++ = value;  
        ++value;  
    }  
}
```

```
std::vector<int> vec(5, 1); // вектор из пяти единиц  
std::iota(vec.begin(), vec.end(), 3); // неважно, чем был заполнен, начнём с 3  
for (auto el : vec) std::cout << el; // => 34567
```

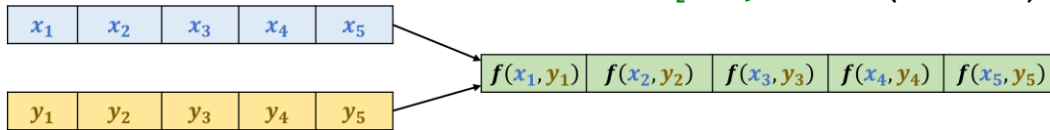
ПРЕОБРАЗОВАТЬ

ЗАПОЛНИТЬ ЗНАЧЕНИЕМ $O(n * O(tr))$ (transform): #include <algorithm>

transform(it1,it2,it,foo); //положит foo(el) от el из [it1,it2) в [it,it+(it2-it1))



transform(it1,it2,it3,it,foo); // положит foo(el1, el2) от el1 из [it1, it2) и el2 из [it3, it3 + (it2-it1) в [it, it + (it2-it1))



```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first, InputIt last, OutputIt dest, UnaryOperation oper){
    for (;first != last; ++first, ++dest)
        *dest = oper(*first);
    return dest;
}
```

```
template<class InputIt1, class InputIt2, class OutputIt, class BinaryOperation>
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt dest, BinaryOperation oper) {
    for (; first1 != last1; ++first1, ++first2, ++dest)
        *dest = oper(*first1, *first2);
    return dest;
}
```

// Пример применения унарной операции:

```
std::string str = "Hello World!";
std::transform(str.begin(), str.end(), str.begin(), std::toupper);
std::cout << str << std::endl; // => HELLO WORLD!
```

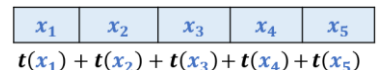
// Пример применения бинарной операции:

```
std::vector<int> vec1 = { 1,2,0,1 }, vec2 = { 2,0,1,3 };
std::transform(vec1.begin(), vec1.end(), vec2.begin(),
    std::ostream_iterator<int>(std::cout, " "),
    std::multiplies<>()); // => 2 0 0 3
```

НАКОПИТЬ (transform_reduce) $O(n * (O(re) + O(tr)))$: #include <numeric>

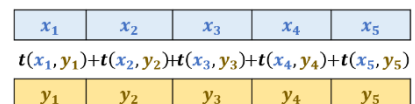
transform_reduce(first1, last1, value, bin_reduce, unary_transform);

```
template<class InputIt, class OutputIt, class T, class BinReduce, class UnaryTransform>
OutputIt transform_reduce(InputIt first, InputIt last, T value,
                          BinReduce reduce, UnaryTransform transform) {
    for (; first != last; ++first)
        value = reduce(std::move(value), transform(*first));
    return value;
}
```



transform_reduce(first1, last1, first2, value[, bin_reduce, bin_transform]);

```
template<class InputIt1, class InputIt2, class OutputIt, class T,
        class BinReduce, class BinTransform>
OutputIt transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                          T value, BinReduce reduce, BinTransform transform) {
    for (; first1 != last1; ++first1, ++first2)
        value = reduce(std::move(value), transform(*first1, *first2));
    return value;
}
```



```
template<class InputIt1, class InputIt2, class OutputIt, class T>
OutputIt transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2, T value) {
    return transform(first1, last1, first2, value, std::plus<>{}, std::multiplies<>{});
}
```

ЧАСТИЧНЫЕ СУММЫ ВКЛЮЧИТЕЛЬНО (transform_inclusive_scan): #include <numeric>

Сложность: $O(n * (O(re) + O(tr)))$

{tr(a), re(tr(a),tr(b)), re(re(tr(a),tr(b)),tr(c)), ...}

```
template<class InputIt, class OutputIt, class BinaryOp, class UnaryOp>
OutputIt transform_inclusive_scan(InputIt first, InputIt last, OutputIt dest,
                                  BinaryOp reduce_op, UnaryOp transform_op) {
    if (first == last)
        return dest; // т.е. ничего делать не надо
    using value_t = typename std::iterator_traits<InputIt>::value_type;
    value_t sum = transform_op(*first);
    *dest = sum; // самая первая сумма – это первый элемент
    while (++first != last) { // Перед шагом смещаемся на следующий элемент
        sum = reduce_op(std::move(sum), transform_op(*first));
        *++dest = sum;
    }
    return ++dest;
}
```

```
template <class InputIt, class OutputIt, class T, class BinaryOp, class UnaryOp >
OutputIt transform_inclusive_scan(InputIt first, InputIt last, OutputIt dest,
                                  BinaryOp reduce_op, UnaryOp transform_op, T sum){
    for (; first != last; ++first, ++dest) {
        sum = reduce_op(std::move(sum), transform_op(*first));
        *dest = sum;
    }
    return dest;
}
```

```
std::vector data = { 1,2,3,4,5 };
std::transform_inclusive_scan(data.begin(), data.end(),
    std::ostream_iterator<int>(std::cout, " "),
    std::plus<int>{}, [](int x) { return x * 10; });
// => 10 30 60 100 150
```

ЧАСТИЧНЫЕ СУММЫ ИСКЛЮЧИТЕЛЬНО (transform_exclusive_scan): #include <numeric>

Сложность: $O(n * (O(re) + O(tr)))$

{tr(value), re(tr(value),tr(a)), re(re(tr(value),tr(a)),tr(b)), ...}

```
template <class InputIt, class OutputIt, class T, class BinaryOp, class UnaryOp>
OutputIt transform_exclusive_scan(InputIt first, InputIt last, OutputIt dest,
                                  T value, BinaryOp reduce_op, UnaryOp transform_op){
    if (first == last)
        return last;
    while (true) {
        T next_value = reduce_op(value, transform_op(*first));
        *dest = value;
        ++dest, ++first;
        if (first == last)
            break;
        value = std::move(next_value);
    }
    return dest;
}
```

```
std::vector data = { 1,2,3,4,5 };
std::transform_exclusive_scan(data.begin(), data.end(),
    std::ostream_iterator<int>(std::cout, " "),
    0, std::plus<int>{}, [](int x) { return x * 10; });
// => 0 10 30 60 100
```


НАИБОЛЬШИЙ/НАИМЕНЬШИЙ

$O(n)$ (max_element, min_element):

// max_element сравнивает [first, last) => вернёт итератор на наибольший
если несколько наибольших => вернёт первый такой
если диапазон пустой => вернёт last

```
template<class ForwardIt, class Compare = std::less<>>
ForwardIt max_element(ForwardIt first, ForwardIt last, Compare comp = Compare()){
    if (first == last) return last;
    ForwardIt largest = first;
    ++first;
    for (; first != last; ++first)
        if (comp(*largest, *first))
            largest = first;
    return largest;
}
```

```
template<class ForwardIt, class Compare = std::less<>>
ForwardIt min_element(ForwardIt first, ForwardIt last, Compare comp = Compare()){
    if (first == last) return last;
    ForwardIt smallest = first;
    ++first;
    for (; first != last; ++first)
        if (comp(*first, *smallest))
            smallest = first;
    return smallest;
}
```

// Пример использования max_element:

```
std::vector<int> vec = { 3, 5, -15, 2, 9 };
auto it1 = std::max_element(vec.begin(), vec.end());
std::cout << "max at: " << std::distance(vec.begin(), it1) << '\n'; //=> max at: 4
auto comp = [](auto a, auto b) { return std::abs(a) < std::abs(b); };
auto it2 = std::max_element(vec.begin(), vec.end(), comp);
std::cout << "max at: " << std::distance(vec.begin(), it2) << '\n'; //=> max at: 2
```

$O(n * 1.5)$ (minmax_element):

Сравнивает [first, last) => вернёт пару итераторов {it_min, it_max}

```
std::vector<int> vec = { 3, -9, 1, -4, 2, 5 };
auto [min_it, max_it] = std::minmax_element(vec.begin(), vec.end());
std::cout << "min = " << *min_it << ", max = " << *max_it; //=> min = -9, max = 5
auto comp = [](int a, int b) { return std::abs(a) < std::abs(b); };
tie(min_it, max_it) = std::minmax_element(vec.begin(), vec.end(), comp);
std::cout << "min = " << *min_it << ", max = " << *max_it; //=> min = 1, max = -9
```

$O(n * 1.5)$ (minmax):

```
template<class T, class Compare = std::less<T>>
std::pair<T, T> minmax(std::initializer_list<T> lst, Compare comp = Compare()){
    auto pair_it = std::minmax_element(lst.begin(), lst.end(), comp);
    return std::pair(*pair_it.first, *pair_it.second);
}
```

// Пример использования minmax (в пару возвращаются копии):

```
auto [min, max] = std::minmax({ 1, -3, 2 });
std::cout << "min: " << min << " max: " << max; // => min = -3, max = 2
auto comp = [](int a, int b) { return std::abs(a) < std::abs(b); };
auto [min, max] = std::minmax({ 1, -3, 2 }, comp);
std::cout << "min: " << min << " max: " << max; // => min = 1, max = -3
```

$O(n)$ (max, min):

// max сравнивает {...} => вернёт наибольший или если их несколько, то первый такой
// min сравнивает {...} => вернёт наименьший или если их несколько, то первый такой

```
template<class T, class Compare = std::less<T>>
T max(std::initializer_list<T> lst, Compare comp = Compare()) {
    return *std::max_element(lst.begin(), lst.end(), comp);
}
```

```
template<class T, class Compare = std::less<T>>
T min(std::initializer_list<T> lst, Compare comp = Compare()) {
    return *std::min_element(lst.begin(), lst.end(), comp);
}
```

// Пример использования max:

```
std::cout << "max: " << std::max({ 1, -3, 2 }) << '\n'; // => 2
```

```
auto lambda = [](int a, int b) { return std::abs(a) < std::abs(b); };
std::cout << "max: " << std::max({ 1, -3, 2 }, lambda) << '\n'; // => -3
```

$O(1)$ (max, min, minmax):

max сравнивает 2 obj одинакового типа => вернёт наибольший или левый, если равны

min сравнивает 2 obj одинакового типа => вернёт наименьший или левый, если равны

minmax сравнивает 2 obj одинакового типа =>

// Возвращает константные ссылки на аргументы => проблемы с висячими ссылками

// Используют компаратор по умолчанию (меньше), или заданный из 3 аргумента:

```
template<class T, class Compare = std::less<T>>
const T& max(const T& a, const T& b, Compare comp = Compare()){
    return (comp(a, b)) ? b : a;
}
```

```
template<class T, class Compare = std::less<T>>
const T& min(const T& a, const T& b, Compare comp = Compare()){
    return (comp(b, a)) ? b : a;
}
```

```
template<class T, class Comp = std::less<T>>
std::pair<const T&, const T&> minmax(const T& a, const T& b, Comp comp=Comp()){
    return (comp(b, a)) ? std::pair<const T&, const T&>(b, a)
        : std::pair<const T&, const T&>(a, b);
}
```

// Пример использования max:

```
std::string xyz = "xyz", abcd = "abcd";
```

```
std::cout << "max lexi-gra: " << std::max(xyz, abcd) << '\n'; // => max lexi-gra: xyz
```

```
auto sizer = [](std::string a, std::string b) { return a.size() < b.size(); };
std::cout << "max size: " << std::max(xyz, abcd, sizer) << '\n'; // => max size: abcd
```

// Пример использования minmax - проблема с висячими ссылками:

```
int a = 1;
auto [min, max] = std::minmax(a, a-4);
std::cout << "min: " << min << " max: " << max; // => UB
```

```
int a = 1, b = -3;
auto comp = [](int a, int b) { return std::abs(a) < std::abs(b); };
auto [min, max] = std::minmax(a, b, comp);
std::cout << "min: " << min << " max: " << max; // => min: 1 max: -3
```

ЗАЖИМ $O(2)$ (clamp):

// Если val < low => low т.е. нижнюю границу

// Иначе если val > high => high т.е. верхнюю границу

// Иначе => val т.е. val лежит в [low, high]

// Если high < low => UB

```
template<class T, class Compare = std::less<T>>
const T& clamp(const T& val, const T& low, const T& high, Compare comp=Compare()){
    return comp(val, low) ? low : comp(high, val) ? high : val;
}
```

СРАВНЕНИЕ ДИАПАЗОНОВ

СОВПАДАЕТ ПОЭЛЕМЕНТНО $O(n)$ (equal):

```
// Проверяет на совпадение: [first1, last1) == [first2, first2 + (last1-first1))
template<class InputIt1, class InputIt2, class Comp = std::equal_to<>>
bool equal(InputIt1 first1, InputIt1 last1, InputIt2 first2, Comp comp = Comp()) {
    for (; first1 != last1; ++first1, ++first2)
        if (!comp(*first1, *first2)) return false;
    return true;
}
```

```
// Проверяет на совпадение: [first1, last1) == [first2, last2)
template<class InputIt1, class InputIt2, class Comp = std::equal_to<>>
bool equal(InputIt1 first1, InputIt1 last1, InputIt2 first2, InputIt2 last2, Comp comp = Comp()) {
    if(std::distance(first1, last1) != std::distance(first2, last2))
        return false; // т.к. диапазоны разных размеров
    return std::equal(first1, last1, first2, comp);
}
```

```
auto is_palindrome = [](const std::string& s) {
    return std::equal(s.begin(), s.end(), s.rbegin());
};
std::cout << std::boolalpha << is_palindrome("radar") << std::endl; // => true
std::cout << std::boolalpha << is_palindrome("hello") << std::endl; // => false
```

МЕНЬШЕ ЛЕКСИКОГРАФИЧЕСКИ $O(n)$ (lexicographical_compare):

// Лексикографическое сравнение, что [first1, last1) < [first2, last2):

```
template< class InputIt1, class InputIt2, class Comp = std::less<>>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2, Comp comp = Comp()){
    for (; (first1 != last1) && (first2 != last2); ++first1, ++first2) {
        if (comp(*first1, *first2)) return true;
        if (comp(*first2, *first1)) return false;
        // Если "буквы" совпали, то проверяем дальше...
    } // Какое-то из слов закончилось, но точно ли в первом меньше «букв»?
    return (first1 == last1) && (first2 != last2);
}
```

```
std::vector<char> word1 = { 'a' };
std::vector<char> word2 = { 'a', 'b', 'c' };
std::vector<char> word3 = { 'b', 'a' };
std::lexicographical_compare(word1.begin(), word1.end(), word2.begin(), word2.end()); // => true
std::lexicographical_compare (word2.begin(), word2.end(), word3.begin(), word3.end()); // => true
```

ПОИСК ПОСЛЕДОВАТЕЛЬНОСТИ $O(n+m)$ (search, find_end):

Ищет первое вхождение всего $[s_first, s_last)$ в $[first, last)$

```
template <class ForwardIt1, class ForwardIt2, class Compare = std::equal_to<>>
ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
                  ForwardIt2 s_first, ForwardIt2 s_last, Compare pred = Compare{}) {
    while (true) {
        ForwardIt1 it = first;
        for (ForwardIt2 s_it = s_first; true; ++it, ++s_it) {
            if (s_it == s_last) // дошли до конца искомой последовательности
                return first; // значит вернём её начало т.к. нашли
            if (it == last) // дошли до конца исходной последовательности
                return last; // значит вернём её конец т.к. не нашли
            if (!pred(*it, *s_it)) // если подпоследовательность не совпала
                break; // то заканчиваем этот этап поиска от first
        }
        ++first; // и смещаем first, чтобы начать искать с нового места
    }
}
```

```
std::vector<int> vec = { 1, 2, 3, 4, 5, 6, 3, 4, 0, 0 }, other = { 3, 4 };
auto from = std::search(vec.begin(), vec.end(), other.begin(), other.end());
auto to = from + other.size();
std::cout << "Before: "; // => Before: 1 2
std::copy(vec.begin(), from, std::ostream_iterator<int>(std::cout, " "));
std::cout << "\nIn: "; // => In: 3 4
std::copy(from, to, std::ostream_iterator<int>(std::cout, " "));
std::cout << "\nAfter: "; // => After: 5 6 3 4 0 0
std::copy(to, vec.end(), std::ostream_iterator<int>(std::cout, " "));
```

Ищет последнее вхождение всего $[s_first, s_last)$ в $[first, last)$

```
template <class ForwardIt1, class ForwardIt2, class Compare = std::equal_to<>>
ForwardIt1 find_end(ForwardIt1 first, ForwardIt1 last, ForwardIt2 s_first,
                   ForwardIt2 s_last, Compare pred = Compare{}) {
    auto result = last; // сразу на конец на случай, если не найдём
    while (true) {
        ForwardIt1 it = first;
        for (ForwardIt2 s_it = s_first; true; ++it, ++s_it) {
            // Найден ли кандидат (начало искомой последовательности):
            const bool success = static_cast<bool>(s_it == s_last);
            if (success) // дошли до конца искомой последовательности
                result = first; // значит запомним кандидата
            if (it == last) // дошли до конца исходной последовательности
                return result; // значит вернём кандидата
            if (success || !pred(*it, *s_it)) // кандидат найден ИЛИ НЕ найден
                break; // Переходим к следующей попытке найти кандидата
        }
        ++first; // смещаем first, чтобы начать искать с нового места
    }
}
```

```
std::vector<int> vec = { 1, 2, 3, 4, 5, 6, 3, 4, 0, 0 }, other = { 3, 4 };
auto from = std::find_end(vec.begin(), vec.end(), other.begin(), other.end());
auto to = from + other.size();
std::cout << "Before: "; // => Before: 1 2 3 4 5 6
std::copy(vec.begin(), from, std::ostream_iterator<int>(std::cout, " "));
std::cout << "\nIn: "; // => In: 3 4
std::copy(from, to, std::ostream_iterator<int>(std::cout, " "));
std::cout << "\nAfter: "; // => After: 0 0
std::copy(to, vec.end(), std::ostream_iterator<int>(std::cout, " "));
```

ПОИСК ПОВТОРЯЮЩИХСЯ $O(n + m)$ (search_n):

Ищет первое вхождение всего [value, value, ... xn раз) в [first, last)

```
template <class ForwardIt, class Diff, class T, class Compare=std::equal_to<>>
ForwardIt search_n( ForwardIt first, ForwardIt last,
                    Diff count, const T& value, Compare pred = Compare{}) {
    if (count <= 0)
        return first;
    for (; first != last; ++first) {
        if (!pred(*first, value)) // пропускаем несовпадающие элементы
            continue;
        // Элемент совпал, убедимся, что все, кто после него тоже совпадают:
        ForwardIt result = first; // это кандидат (начало повторяющихся)
        for (Diff current_count = 1; true; ++current_count) {
            if (current_count == count)
                return result; // смогли найти нужное коли-во повторяющихся
            else if (++first == last) // перейдём к следующему элементу
                return last; // т.е. не нашли, т.к. закончился диапазон
            else if (!pred(*first, value)) // следующий не совпал с требуемым, ...
                break; // ... => кандидат не подошёл, будем искать дальше.
        }
    }
    return last;
}
```

```
std::vector<bool> flags = { 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0 };
size_t repetitions = 3;
auto from = std::search_n(flags.begin(), flags.end(), repetitions, true);
auto to = from + repetitions;
std::cout << "Before: "; // => Before: 0 0
std::copy(flags.begin(), from, std::ostream_iterator<bool>{std::cout, " "});
std::cout << "\nIn: "; // => In: 1 1 1
std::copy(from, to, std::ostream_iterator<bool>{std::cout, " "});
std::cout << "\nAfter: "; // => After: 0 1 0 1 1 1 0
std::copy(to, flags.end(), std::ostream_iterator<bool>{std::cout, " "});
// Но почему у него нет парного для поиска с конца, а не сначала? Потому что
// диапазон повторяющихся совпадает с инвертированным => обратные итераторы помогут:
size_t repetitions = 3;
auto r_from = std::search_n(flags.rbegin(), flags.rend(), repetitions, true);
to = r_from.base();
from = to - repetitions;
std::cout << "Before: "; // => Before: 0 0 1 1 1 0 1 0
std::copy(flags.begin(), from, std::ostream_iterator<bool>{std::cout, " "});
std::cout << "\nIn: "; // => In: 1 1 1
std::copy(from, to, std::ostream_iterator<bool>{std::cout, " "});
std::cout << "\nAfter: "; // => After: 0
std::copy(to, flags.end(), std::ostream_iterator<bool>{std::cout, " "});
```

ПЕРВОЕ НЕСОВПАДЕНИЕ $O(n)$ (mismatch):

```
// Ищет первое НЕСовпадение у [first1, last1) и [first2, first2 + (last1-first1))
template<class InputIt1, class InputIt2, class Comp = std::equal_to<>>
std::pair<InputIt1, InputIt2> mismatch( InputIt1 first1, InputIt1 last1,
    InputIt2 first2, Comp comp = Comp{}) {
    while (first1 != last1 && comp(*first1, *first2))
        ++first1, ++first2;
    return std::make_pair(first1, first2);
}
```

```
// Ищет первое НЕСовпадение у [first1, last1) и [first2, last2)
template<class InputIt1, class InputIt2, class Comp = std::equal_to<>>
std::pair<InputIt1, InputIt2> mismatch(InputIt1 first1, InputIt1 last1,
    InputIt2 first2, InputIt2 last2, Comp comp = Comp{}){
    while (first1 != last1 && first2 != last2 && p(*first1, *first2))
        ++first1, ++first2;
    return std::make_pair(first1, first2);
}
```

```
std::string str1 = "Hey!", str2 = "Hello!";
auto [p1, p2] = std::mismatch(str1.begin(), str1.end(), str2.begin(), str2.end());
std::cout << *p1 << ' ' << *p2 << '\n'; // => y l
auto [r_p1, r_p2] = std::mismatch(str1.rbegin(), str1.rend(), str2.rbegin());
std::cout << *r_p1 << ' ' << *r_p2 << '\n'; // => y o
std::tie(p1, p2) = std::mismatch(str2.begin(), str2.end(), str1.begin()); // => UB
// т.к. str2.size() > str1.size() => при проверке вышли за границы str1
```


МОДИФИЦИРУЮЩИЕ

ЗАПОЛНИТЬ ЗНАЧЕНИЕМ $O(n)$ (fill, fill_n):

fill(it1, it2, val); // присваивает значение val всем элементам из [it1, it2)
fill_n(it1, n, val); // присваивает значение val всем элементам из [it1, it1 + n)

```
template<typename ForwardIt, typename T>
void fill(ForwardIt first, ForwardIt last, const T& value){
    for (; first != last; ++first)
        *first = value;
}

template<typename OutputIt, typename Size, typename T>
OutputIt fill_n(OutputIt first, Size count, const T& value){
    for (; 0 < count; --count, ++first)
        *first = value;
    return first;
}
```

```
std::vector<std::string> words(5); // 5 пустых слов
std::fill(words.begin(), words.end(), "word"); // все слова заполнились как "word"
auto it = std::fill_n(std::next(words.begin()), 3, "change"); // от 2ого 3 раза
for (auto el : words) std::cout << el << ' '; // => word change change change word
```

ЗАПОЛНИТЬ РЕЗУЛЬТАТОМ ФУНКЦИИ $O(n)$ (generate, generate_n):

generate(it1, it2, foo); // всем элементам из [it1, it2) присвоит результат foo()
generate_n(it1, n, foo); // всем элементам из [it1, it1 + n) присвоит foo()

```
template<typename ForwardIt, typename Generator>
void generate(ForwardIt first, ForwardIt last, Generator foo){
    for (; first != last; ++first)
        *first = foo();
}

template<typename OutputIt, typename Size, typename Generator>
OutputIt generate_n(OutputIt first, Size count, Generator foo) {
    for (; 0 < count; --count, ++first)
        *first = g();
    return first;
}
```

```
std::vector<int> vec(5);
std::generate(vec.begin(), vec.end(), [n = 0]() mutable {return ++n; });
auto it = std::generate_n(std::next(vec.begin()), 2, rand); // от 1ого 2 раза
for (auto el : vec) std::cout << el << ' '; // => 1 41 18467 4 5
```

ЗАМЕНИТЬ ЗНАЧЕНИЯ $O(n)$ (replace, replace_if):

replace(it1, it2, old_val, new_val); //заменит все old_val на new_val в [it1, it2)
replace_if(it1, it2, foo, new_val); // заменит все foo(el) => true на new_val

```
template<typename ForwardIt, typename T>
void replace(ForwardIt first, ForwardIt last, const T& old_value, const T& new_value){
    for (; first != last; ++first)
        if (*first == old_value)
            *first = new_value;
}

template<typename ForwardIt, typename UnaryPredicate, typename T>
void replace_if(ForwardIt first, ForwardIt last, UnaryPredicate pred, const T& new_value) {
    for (; first != last; ++first)
        if (pred(*first))
            *first = new_value;
}
```

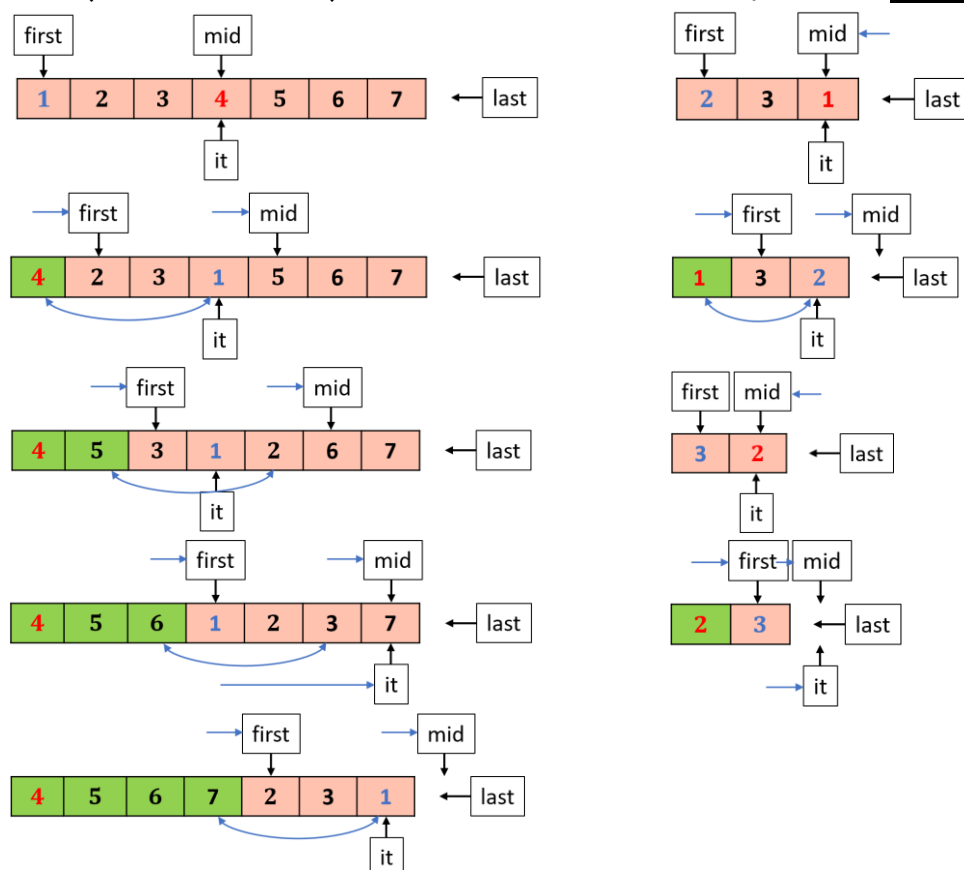
```
std::vector<int> vec = { 1, 0, 0, 0, 1, 1, 2, 3 };
std::replace(vec.begin(), vec.end(), 1, -1);
for (auto el : vec) std::cout << el << ' '; // => -1 0 0 0 -1 -1 2 3
```

ПРОВЕРНУТЬ $O(n)$ (rotate):

`rotate(it1, it, it2);` // провернёт [it1, it2) так, что it встанет на место it1
// => итератор на новое местоположение элемента из it1 т.е. it1 + (it2 - it)

```
template<typename ForwardIt>
ForwardIt rotate(ForwardIt first, ForwardIt mid, ForwardIt last) {
    if (first == mid) return last;
    if (mid == last) return first;
    ForwardIt it = mid; // после 1го шага 1ый элемент уйдёт сразу туда
    do {
        std::iter_swap(first, mid);
        ++first;
        ++mid;
        if (first == it) it = mid; // Запоминаем куда уйдёт 1ый элемент
    } while (mid != last);
    // Повернём оставшуюся часть последовательности
    rotate(first, it, last); // выглядит, конечно, красиво
    return first; // Но внутри rotate будет сам себя рекурсивно вызывать...
    // Поэтому оптимальнее вручную прописать цикл:
    ForwardIt result = first; // Запомнили результат (т.е. куда перешёл 1ый эл.)
    while (it != last) {
        mid = it;
        do { // тут точно такой же цикл, как и до этого, но на другом диапазоне
            std::iter_swap(first, mid);
            ++first;
            ++mid;
            if (first == it) it = mid;
        } while (mid != last);
    }
    return result;
}
```

```
std::vector<int> vec = { 1,2,3,4,5,6 };
std::rotate(vec.begin(), std::next(vec.begin(),3), vec.end());
for (auto el : vec) std::cout << el << ' '; // => 4 5 6 1 2 3
```



ПЕРЕПИСИВАТЬ $O(n)$ (reverse):

reverse(it1, it2); // переставит все элементы из [it1, it2) задом на перед

```
template<class BidIt>
void reverse(BidIt first, BidIt last) {
    for (; first != last && first != --last; ++first)
        std::iter_swap(first, last);
}
```

```
std::vector<int> vec = { 1,2,3,4,5,6 };
```

```
std::reverse(vec.begin(), vec.end());
```

```
for (auto el : vec) std::cout << el << ' '; // => 6 5 4 3 2 1
```

ПЕРЕМЕШАТЬ $O(n)$ (shuffle):

shuffle(it1, it2, g); // перемешивает элементы из [it1, it2) используя генератор g

random_shuffle // Устарел в C++14 и удалён в C++17

```
template<typename RandomIt, typename URBG>
void shuffle(RandomIt first, RandomIt last, URBG&& g){
    using diff_t = typename std::iterator_traits<RandomIt>::difference_type;
    using distr_t = std::uniform_int_distribution<diff_t>;
    distr_t D;
    for (diff_t i = (last - first) - 1; i > 0; --i)
        std::swap(first[i], first[D(g, distr_t::param_type(0, i))]);
}
```

```
std::vector<int> vec = { 1,2,3,4,5,6 };
```

```
// std::random_device rd; // равномерно распределенный целочисленный генератор
```

```
// std::mt19937 g(rd()); // случайные числа на основе алгоритма Mersenne Twister
```

```
std::shuffle(vec.begin(), vec.end(), std::mt19937{ std::random_device{}() });
```

```
for (auto el : vec) std::cout << el << ' '; // => 1 6 2 4 5 3
```

МЕНЯЮЩИЕ СТРУКТУРУ

// «Удаление» происходит за счёт смещения актуальных элементов влево (что будет на месте смещённых зависит от реализации...) => итератор на конец актуального диапазона

УДАЛЕНИЕ $O(n)$ (remove, remove_if):

remove(it1, it2, val); // Удалит все элементы val из [it1, it2)

remove_if(it1, it2, pred); // Удалит все el из [it1, it2) такие, что pred(el)==true

```
template <typename ForwardIt, typename T>
```

```
ForwardIt remove(ForwardIt first, const ForwardIt last, const T& val) {
```

```
    first = std::find(first, last, val);
```

```
    if (first != last) // т.е. найденный элемент не после конца
```

```
        for(ForwardIt it = first; ++it != last;)
```

```
            if (!(*it == val)) // it нацелен на следующий обрабатываемый элемент
```

```
                *first++ = std::move(*it); // конец диапазона сместился
```

```
    return first;
```

```
}
```

```
template <typename ForwardIt, typename UnaryPredicate>
```

```
ForwardIt remove_if(ForwardIt first, const ForwardIt last, UnaryPredicate pred) {
```

```
    first = std::find_if(first, last, pred);
```

```
    if (first != last) // т.е. найденный элемент не после конца
```

```
        for (ForwardIt it = first; ++it != last;)
```

```
            if (!pred(*it)) // it нацелен на следующий обрабатываемый элемент
```

```
                *first++ = std::move(*it); // конец диапазона сместился
```

```
    return first;
```

```
}
```

// На количество актуальных элементов такая операция не влияет:

```
std::string str = "Some %%text%";
```

```
auto it_end = std::remove(str.begin(), str.end(), '%');
```

```
cout << str; // => Some textxt%
```

// Поэтому этот алгоритм хорошо использовать в паре с методом erase:

```
str.erase(it_end, str.end());
```

```
std::cout << str; // => Some text
```

// Для этого можно даже написать свой собственный алгоритм:

```
template <typename Collection, typename T>
```

```
void erase(Collection& container, const T& value) {
```

```
    auto it_end = remove(container.begin(), container.end(), value);
```

```
    container.erase(it_end, container.end());
```

```
}
```

```
std::string str = "Some %%text%";
```

```
erase(str, '%');
```

```
std::cout << str; // => Some text
```

// И аналогичный ему свой собственный алгоритм с унарным предикатом:

```
template <typename Collection, typename UnaryPredicate>
```

```
void erase_if(Collection& container, UnaryPredicate pred) {
```

```
    auto it_end = remove_if(container.begin(), container.end(), pred);
```

```
    container.erase(it_end, container.end());
```

```
}
```

```
std::vector<int> vec = { 0,1,2,3,4,5,6,7,8,9 };
```

```
erase_if(vec, [](auto el) { return el % 2; }); // удалит все нечётные
```

```
for (auto el : vec) std::cout << el; // => 02468
```

БЕЗ ПОВТОРЕНИЙ ПОДРЯД $O(n)$ (unique):

```
unique(it1, it2); // Удалит повторяющиеся подряд el из [it1, it2)
unique(it1, it2, pred); //Удалит el из [it1,it2) такие, что pred(el,el_next)==true

template<typename ForwardIt, typename BinPred = std::equal_to<>>
ForwardIt unique(ForwardIt first, ForwardIt last, BinPred pred = BinPred()) {
    if (first == last) return last;
    ForwardIt it = first; // итератор на конец актуального диапазона
    while (++first != last) // смещаем левую границу рассматриваемого диапазона
        if (!pred(*it, *first)) // если актуальный и рассматриваемый не совпали
            if (++it != first) //смещаем актуальный и если не совпал с рассм-ым...
                *it = std::move(*first); //...то из рассм-го переместим в актуальный
            // иначе: они совпали => ничего перемещать не нужно, ведь он уже тут
            // иначе: встретился повторяющийся => игнорируем и переходим к след. шагу
    return ++it; // вернём конец актуального диапазона
}
```

```
// На количество актуальных элементов такая операция не влияет:
std::string str = "1223444222";
auto it_end = std::unique(str.begin(), str.end());
std::cout << str; // => 1234244222
// Поэтому этот алгоритм хорошо использовать в паре с методом erase:
str.erase(it_end, str.end());
std::cout << str; // => 12342
// Для этого можно даже написать свой собственный алгоритм:
```

```
template <typename Collection, typename BinPred = std::equal_to<>>
void unique(Collection& container, BinPred pred = BinPred()) {
    auto it_end = unique(container.begin(), container.end(), pred);
    container.erase(it_end, container.end());
}
```

```
std::string str = "1223444222";
unique(str);
std::cout << str; // => 12342
```

ПЕРЕСТАНОВКИ

A_n^k Размещения из n по k : наборы из k элементов, выбранных из n -элементного множества, которые отличаются друг от друга как выборкой элементов, так и порядком их расположения.

1	2	3	4
---	---	---	---

→

1	1	1	2	2	2	3	3	3	4	4	4
2	3	4	1	3	4	1	2	4	1	2	3

$A_n^1 = n$ т.к. всего одна позиция, куда можно вставить элемент

$A_n^2 = n(n-1)$ т.к. всего две позиции, куда можно вставить неповторяющиеся элементы

$A_n^3 = n(n-1)(n-2)$ и т.д. по математической индукции можно доказать:

$$A_n^k = n(n-1) \dots (n-(k-1)) = n(n-1) \dots (n-(k-1)) \frac{(n-k) \dots 2 \cdot 1}{(n-k) \dots 2 \cdot 1} = \frac{n!}{(n-k)!}$$

$$A_n^n = n(n-1) \dots (n-(n-1)) = n(n-1) \dots 2 \cdot 1 = n!$$

P_n Перестановки из n : это наборы из n элементов, которые отличаются друг от друга только порядком их расположения (т.е. это размещения из n по n):

1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
2	2	3	3	4	4	1	1	3	3	4	4	1	1	2	2	4	4	1	1	2	2	3	3
3	4	2	4	2	3	3	4	1	4	1	3	2	4	1	4	1	2	2	3	1	3	1	2
4	3	4	2	3	2	4	3	4	1	3	1	4	2	4	1	2	1	3	2	3	1	2	1

$$P_n = A_n^n = n! \quad \Rightarrow$$

$$A_n^k = \frac{n!}{(n-k)!} = \frac{P_n}{P_{n-k}}$$

C_n^k Сочетания из n по k : наборы из k элементов, выбранных из n -элементного множества, которые НЕ отличаются друг от друга порядком их расположения.

1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5

Выведем формулу: если для каждого сочетания из n по k сделать всевозможные перестановки, то получатся всевозможные размещения:

$$C_n^k P_k = A_n^k \quad \Rightarrow \quad C_n^k = \frac{A_n^k}{P_k} = \frac{P_n}{P_{n-k} P_k} = \frac{n!}{(n-k)! k!}$$

ПЕРЕСТАНОВКА ЛИБЛИ $O(n^2)$ (`is_permutation`):

// Проверка, что `[first1, last2)` является перестановкой `[first2, last2)`:

```
template<class ForwardIt1, class ForwardIt2, class Comp = std::equal_to<>>
bool is_permutation(ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first, Comp comp = Comp{}){
    // Пропустим совпадающее начало двух диапазонов
    std::tie(first, d_first) = std::mismatch(first, last, d_first, comp);
    if (first == last) // если это конец, то закончим
        return true;
    // Иначе продолжаем проверять, что каждый из [first1, last1) встречается
    // в [d_first, d_last), причем учитывая кол-во раз этого появления
    ForwardIt2 d_last = std::next(d_first, std::distance(first, last));
    for (ForwardIt1 it = first; it != last; ++it) {
        auto unary_pred = [el=*it, &comp](auto val){ return comp(el, val); };
        if (it != std::find_if(d_first, d_last, unary_pred)) // до it нашёлся == *it
            continue; // ... значит мы его уже проверяли
        // сколько раз такой *it встречается в [d_first, d_last):
        size_t repetitions_dest = std::count_if(d_first, d_last, unary_pred);
        if (repetitions_dest == 0) // ни разу => это точно не перестановка
            return false;
        // нужно, чтобы столько же, сколько у нас, т.е. в [it, last):
        if (std::count_if(it, last, unary_pred) != repetitions_dest)
            return false;
    }
    return true;
}
```

```
template<class ForwardIt1, class ForwardIt2, class Comp = std::equal_to<>>
bool is_permutation(ForwardIt1 first, ForwardIt1 last,
    ForwardIt2 d_first, ForwardIt2 d_last, Comp comp = Comp{}) {
    if (std::distance(first, last) != std::distance(d_first, d_last))
        return false; // т.к. диапазоны разных размеров
    return is_permutation(first, last, d_first, comp);
}
```

```
std::array v1 = { 1, 2, 3, 4, 5 }, v2 = { 3, 5, 4, 1, 2 };
```

```
std::vector v3 = { 3, 5, 4, 1, 1 }, v4 = { 3, 5, 4 };
```

```
std::cout << std::boolalpha;
```

```
std::cout << std::is_permutation(v1.begin(), v1.end(), v2.begin()); // => true
```

```
std::cout << std::is_permutation(v1.begin(), v1.end(), v2.begin(), v2.end()); // => true
```

```
std::cout << std::is_permutation(v1.begin(), v1.end(), v3.begin()); // => false
```

```
std::cout << std::is_permutation(v1.begin(), v1.end(), v4.begin(), v4.end()); // => false
```

СЛЕДУЮЩАЯ ПЕРЕСТАНОВКА $O(n)$ (next_permutation):

Следующая в лексикографическом смысле перестановка.

Возвращает `true`, если следующая перестановка больше, иначе вернёт `false` т.е. последняя перестановка была достигнута, а значит следующая стала самой первой.

// Можно сделать реализацию используя изученные ранее алгоритмы:

```
template<class BidirIt, class Comp = std::less<>>
bool next_permutation(BidirIt first, BidirIt last, Comp comp = Comp{}) {
    // Найдём первый справа элемент, который меньше, чем следующий
    auto r_first = std::reverse_iterator<BidirIt>(last);
    auto r_last = std::reverse_iterator<BidirIt>(first);
    auto r_left = std::is_sorted_until(r_first, r_last, comp);
    // Если это полностью нисходящая последовательность => нужно её всю инвертировать
    if (r_left == r_last) { // т.е. first == r_left.base()
        std::reverse(r_left.base(), last);
        return false;
    } // Иначе сделаем следующую в лексикографическом смысле перестановку:
    // Найдём первый справа элемент из диапазона [r_left.base(), last), который > чем *r_left
    auto r_right = std::upper_bound(r_first, r_left, *r_left, comp);
    std::iter_swap(r_left, r_right); // найденный поменяем с *left, где left = r_left.base()
    std::reverse(r_left.base(), last); // и инвертируем остальную часть справа [after_left, last)
    return true;
}
```

// А можно полностью самостоятельно:

```

template <class BiderIt, class Comp = std::less<>>
bool next_permutation(BiderIt first, BiderIt last, Comp pred = Comp{}) {
    // Проверим вырожденный случай (нет элементов или всего 1)
    BiderIt left = last;
    if (first == last || first == --left)
        return false;
    // Сейчас left указывает на последний элемент, и элементов > 2 т.е. есть с кем переставлять
    while (true) {
        // Теперь найдём первый справа элемент, который меньше, чем следующий:
        BiderIt after_left = left--;
        if (pred(*left, *after_left)) { // т.е. нашелся такой
            // Сделаем следующую в лексикографическом смысле перестановку:
            // Найдём первый справа элемент из диапазона [after_left, last), который > чем *left
            BiderIt before_right = last;
            do {
                --before_right;
            } while (!pred(*left, *before_right)); // точно false, когда before_right==after_left
            std::iter_swap(left, before_right); // найденный поменяем с *left
            // И инвертируем остальную часть справа [after_left, last)
            for (; after_left != last && after_left != --last; ++after_left)
                std::iter_swap(after_left, last);
            return true;
        }
        if (left == first){//это полностью нисходящая последовательность => нужно её инвертировать
            for (; first != last && first != --last; ++first)
                std::iter_swap(first, last);
            return false;
        }
        // Такой элемент не нашёлся и это не конец, значит продолжим искать...
    }
}

```

```
// Сложность: в худшем случае N/2 swap, но в среднем 3 сравнения и 1.5 swap
```

```
std::array arr = { 1,2,3,4 };
```

```
std::cout << "|";
```

```
while (std::next_permutation(arr.begin(), arr.end())) {
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << " | ";
}
```

```
// =>
```

1	2	4	3
2	3	1	4
3	2	4	1
4	3	1	2

1	3	2	4
2	3	4	1
3	4	1	2
4	3	2	1

1	3	4	2
2	4	1	3
3	4	2	1
4	1	2	3

1	4	2	3
3	1	2	4
4	1	3	2
2	1	3	4

1	4	3	2
3	1	4	2
4	2	1	3
2	1	4	3

2	1	3	4
3	1	4	2
4	2	1	3
3	2	1	4

2	1	4	3
3	2	1	4
4	2	3	1
3	2	1	4

ПРЕДЫДУЩАЯ ПЕРЕСТАНОВКА $O(n)$ (prev_permutation):

Предыдущая в лексикографическом смысле перестановка.

Возвращает `true`, если предыдущая перестановка меньше, иначе вернёт `false` т.е. первая перестановка была достигнута, а значит предыдущая стала самой последней.

// Реализация аналогична `next_permutation`, но в предикате аргументы наоборот:

```
template<class BidirIt, class Comp = std::less<>>
bool prev_permutation(BidirIt first, BidirIt last, Comp pred = Comp{}) {
    auto left = last;
    if (first == last || first == --left)
        return false;
    while (true) {
        BidirIt after_left, before_right;
        after_left = left--;
        if (pred(*after_left, *left)) {
            before_right = last;
            while (!pred(*--before_right, *left)) ;
            std::iter_swap(left, before_right);
            std::reverse(after_left, last);
            return true;
        }
        if (left == first){
            std::reverse(first, last);
            return false;
        }
    }
}
```

// Сложность: в худшем случае $N/2$ swap, но в среднем 3 сравнения и 1.5 swap

```
std::array arr = { 4,3,2,1 };
```

```
std::cout << "| ";
```

```
while (std::prev_permutation(arr.begin(), arr.end())) {
```

```
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, " "));
```

```
    std::cout << "| ";
```

```
} // =>
```

4 3 1 2	4 2 3 1	4 2 1 3	4 1 3 2	4 1 2 3	3 4 2 1	3 4 1 2
3 2 4 1	3 2 1 4	3 1 4 2	3 1 2 4	2 4 3 1	2 4 1 3	2 3 4 1
2 3 1 4	2 1 4 3	2 1 3 4	1 4 3 2	1 4 2 3	1 3 4 2	1 3 2 4
1 2 4 3	1 2 3 4					

РАЗДЕЛЕНИЕ

РАЗДЕЛИТЬ $O(n)$ (partition):

Перераспределит элементы таким образом, чтобы сначала лежали те, которые удовлетворяют предикату, а после остальные, т.е.:



Возвращает **partition_point** - итератор на конец диапазона с элементами, удовлетворяющими предикату.

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate pred) {
    // 1-й этап: найдём конец диапазона с true элементами
    while (true) {
        if (first == last)
            return first; // выходим, если этот конец и есть конец диапазона
        if (!pred(*first))
            break; // наткнулись на false элемент
        ++first;
    }
    // 2-й этап: разместим все остальные true элементы подряд после first
    for (auto next = std::next(first); next != last; ++next) {
        if (pred(*next)) { // пропускаем все false элементы
            std::iter_swap(first, next); // но перемещаем в начало все true
            ++first; //сдвига partition_point т.е. границу всех true элементов
        }
    }
    return first;
}
```

```
std::array arr = { 1,3,1,2,5,4,2,3,1,5,2,3 };
auto it = std::partition(arr.begin(), arr.end(), [](auto el) { return el % 2; });
std::copy(arr.begin(), it, std::ostream_iterator<int>(std::cout, " "));
std::cout << "| ";
std::copy(it, arr.end(), std::ostream_iterator<int>(std::cout, " "));
// => 1 3 1 3 5 5 1 3 | 2 4 2 2
```

Также с помощью этого алгоритма можно реализовать собственную быструю сортировку:

```
template<class ForwardIt, typename Comp = std::less<>>
void quick_sort(ForwardIt first, ForwardIt last, Comp comp = Comp{}) {
    if (first == last) return; // выход из сортировки, если диапазон пустой
    // выбрали случайный опорный элемент (например, в середине диапазона)
    auto pivot = *std::next(first, std::distance(first, last) / 2);
    // Получим диапазон [first, middle1) - где все элементы < опорного
    auto less = [&pivot, &comp](const auto& em) {return comp(em, pivot); };
    auto middle1 = partition(first, last, less);
    // Получим диапазон [middle1, middle2) - где все элементы == опорному
    auto less_or_equal = [&pivot, &comp](const auto& em){return !comp(pivot, em);};
    auto middle2 = partition(middle1, last, less_or_equal);
    // Итого сформирован диапазон [middle2, last) - где все элементы > опорного
    // Теперь рекурсивно рассортируем всё, что было до и после [middle1, middle2)
    quick_sort(first, middle1);
    quick_sort(middle2, last);
}
```

```
std::array arr = { 1,3,1,2,5,4,2,3,1,5,2,3 };
quick_sort(arr.begin(), arr.end());
std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, " "));
// => 1 1 1 2 2 2 3 3 3 4 5 5
```

РАЗДЕЛЁН ЛИ $O(n)$ (`is_partition`):

// Является ли диапазон разделённым согласно условию из предиката.

```
template<class InputIt, class UnaryPredicate>
bool is_partitioned(InputIt first, InputIt last, UnaryPredicate pred){
    for (; first != last; ++first) // пропускаем true partition
        if (!pred(*first))
            break; // нашли
    for (; first != last; ++first) // проверяем false partition
        if (pred(*first))
            return false; // т.к. после partition_point нашёлся true элемент
    return true;
}
```

```
std::array arr = { 1,2,1,3,2,1,5,4,6,5,4,6 };
auto less_four = [](auto el) { return el < 4; };
std::cout << std::boolalpha <<
    std::is_partitioned(arr.begin(), arr.end(), less_four); // => true
```

РАЗДЕЛИТЕЛЬ (`partition_point`):

// Ищет начало false partition с помощью бинарного поиска.

// Требуется, чтобы `is_partitioned == true`, иначе это UB.

// Реализация с помощью бинарного поиска аналогично тому как в `lower_bound`:

```
template <class ForwardIt, class Pred>
ForwardIt partition_point(ForwardIt first, ForwardIt last, UnaryPredicate pred) {
    typename std::iterator_traits<ForwardIt>::difference_type dist, step;
    dist = std::distance(first, last);
    while (dist > 0) {
        step = dist / 2;
        ForwardIt middle = std::next(first, step); // указатель на середину
        if (pred(*middle)) { // значит будем искать во второй половине
            first = ++middle;
            dist -= step + 1;
        }
        else
            dist = step; // значит будем искать в первой половине
    }
    return first;
}
```

//Сложность сравнений: $O(\log n)$, но сложность итерирования зависит от типа итератора

```
std::array arr = { 1,3,2,4,1,2,3,1,2 };
auto equal_two = [](auto el) { return el == 2; };
decltype(arr)::iterator partition_point;
if (std::is_partitioned(arr.begin(), arr.end(), equal_two)) // уже разделён
    partition_point = std::partition_point(arr.begin(), arr.end(), equal_two);
else // иначе нужно разделить, тут сразу и узнаем partition_point:
    partition_point = std::partition(arr.begin(), arr.end(), equal_two);
```

УСТОЙЧИВОЕ РАЗДЕЛЕНИЕ (`stable_partition`):

// Согласно стандарту, её сложность в худшем случае $O(n * \log(n))$

Но если достаточно памяти, то её сложность $O(n)$

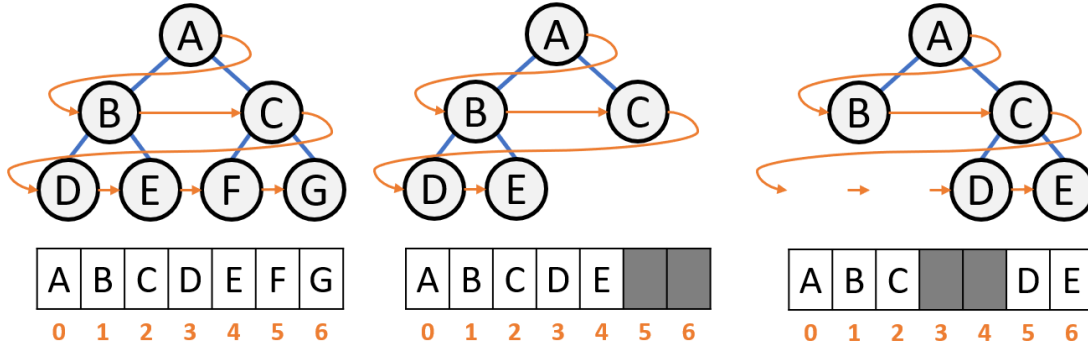
// Устойчивое (стабильное) => значит не меняет относительный порядок разделяемых элементов. Т.е. элементы слева и справа от деления будут идти в том же порядке:

```
template <class BidIt, class Pred>
BidIt stable_partition(BidIt _First, BidIt _Last, Pred Pred);

std::array arr = { 0, 0, 3, -1, 2, 4, 5, 0, 7 };
std::stable_partition(arr.begin(), arr.end(), [](auto el) { return el > 0; });
std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, " "));
// => 3 2 4 5 7 0 0 -1 0
```

КУЧА

СОЗДАНИЕ (make_heap):

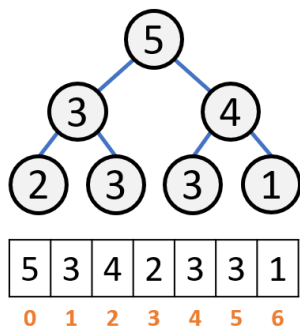


// Для i -го элемента массива верно следующее:

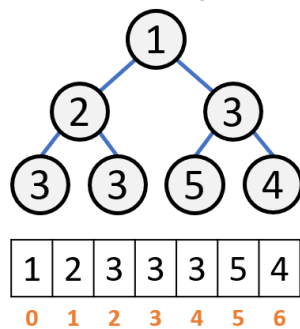
- 1) $2*i + 1$ – левый потомок
- 2) $2*i + 2$ – правый потомок
- 3) $(i-1)/2$ – родитель (целочисленное деление)

// Тип кучи зависит от выбора бинарного предиката (компаратора):

Max Heap



Min Heap



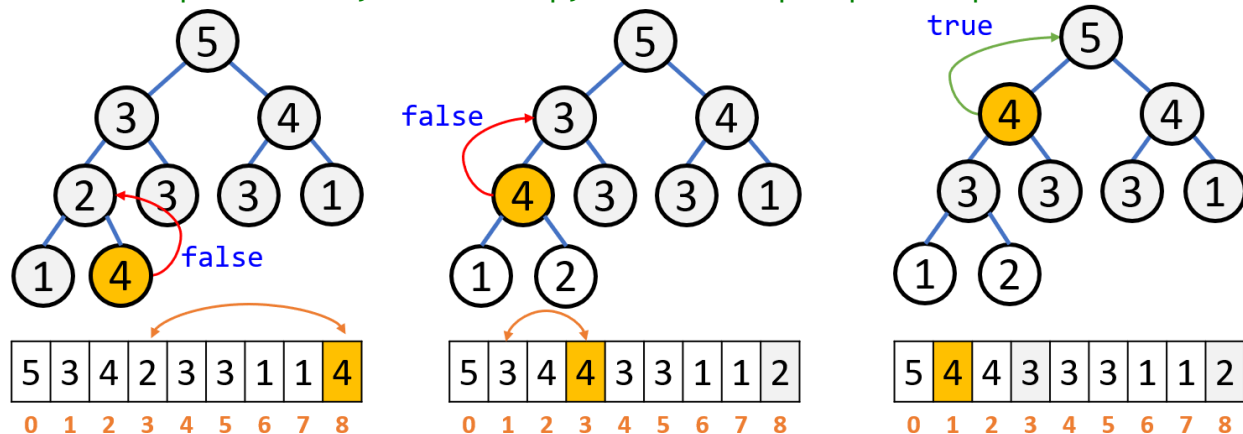
`make_heap(cont.begin(), cont.end());` // по умолчанию `less<>{}` => Max Heap
`make_heap(cont.begin(), cont.end(), greater<>{ });` // теперь это Min Heap

ПРОВЕРКА $O(n)$ (`is_heap`, `is_heap_until`):

`is_heap(vec.begin(), vec.end());` // `true` => если является Max Heap, иначе `false`
`is_heap(vec.begin(), vec.end(), greater<>{ });` // `true` => если является Min Heap
`auto it = is_heap_until(vec.begin(), vec.end());` // => итератор указ-щий на элемент,
 до которого все элементы удовлетворяют куче Max Heap
`auto it = is_heap_until(vec.begin(), vec.end(), greater<>{ });` // => итератор указ-
 на элемент, до которого все элементы удовлетворяют куче Min Heap

ВСТАВКА $O(\log n)$ (`push_heap`):

// Сначала элемент вставляется в самый конец массива, после чего меняется местами со своими родителями, до тех пор, пока компаратор не вернёт `true`



// к моменту вставки, контейнер уже должен быть кучей

`cont.push_back(element);`
`push_heap(cont.begin(), cont.end());`

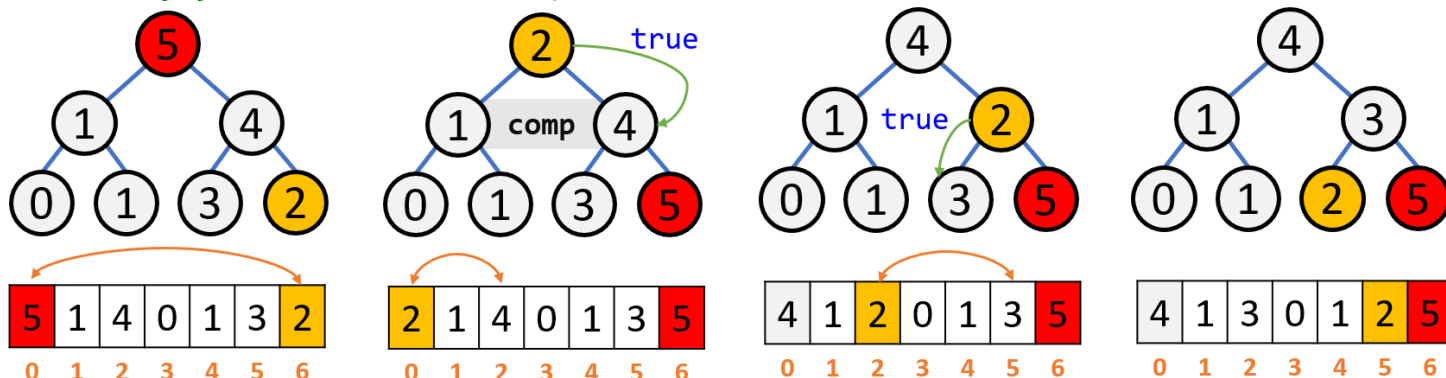
// Но если контейнер представляет из себя MinHeap, то нужно:
`push_heap(cont.begin(), cont.end(), greater<>{ });`

УДАЛЕНИЕ $O(\log n)$ (`pop_heap`):

// удалить можно только корневой элемент из кучи, для этого его переносят в конец, а на его место ставят элемент из конца, после чего кучу нужно перегруппировать:

- 1) Если есть несколько потомков, то с помощью компаратора выберем «сильнейшего»
- 2) Перенесённый элемент с помощью компаратора сравнивают с «сильнейшим» потомком
- 3) Если результат сравнения `true`, то надо поменять их местами

Так продолжать, пока не закончатся потомки, или не встретится `false` (т.е. оба потомка будут «слабее» элемента)



// к моменту удаления, контейнер уже должен быть кучей

```
pop_heap(cont.begin(), cont.end());
```

// Но если контейнер представляет из себя `MinHeap`, то нужно:

```
pop_heap(cont.begin(), cont.end(), greater<>());
```

// Теперь элемент в самом конце массива «отвязан» от кучи и готов к удалению:

```
cont.pop_back();
```

СОРТИРОВКА $O(2 * n * \log n)$ (`sort_heap`):

Это единственный способ сортировки, алгоритм для которого в стандарте указан явно. Каждый раз, делая `pop_heap` корневой элемент оказывается в конце массива, а он ведь был самым большим => в конце массива будут появляться элементы всё меньше и меньше

```
template<typename RandIter, typename Comparator = std::less<>>
void sort_heap(RandIter first, RandIter last, Comparator comp = Comparator()){
    for (; first != last; --last)
        pop_heap(first, last, comp);
}
```

// Отсортируем по возрастанию (для этого понадобится `MaxHeap`)

```
vector<int> vec = { 5, 3, 2, 4, 3, 5, 0 };
```

```
make_heap(vec.begin(), vec.end()); // vec = { 5, 4, 5, 3, 3, 2, 0 }
```

```
sort_heap(vec.begin(), vec.end()); // vec = { 0, 2, 3, 3, 4, 5, 5 }
```

// Отсортируем по убыванию (для этого понадобится `MinHeap`)

```
vector<int> vec = { 5, 3, 2, 4, 3, 5, 0 };
```

```
make_heap(vec.begin(), vec.end(), greater<int>()); // vec = { 0, 3, 2, 4, 3, 5, 5 }
```

```
sort_heap(vec.begin(), vec.end(), greater<int>()); // vec = { 5, 5, 4, 3, 3, 2, 0 }
```

СОРТИРОВКИ

Quick sort	Быстрая сортировка	https://en.wikipedia.org/wiki/Quicksort
Merge sort	Сортировка слиянием	https://en.wikipedia.org/wiki/Merge_sort
Heap sort	Пирамидальная сортировка	https://en.wikipedia.org/wiki/Heapsort
Insertion sort	Сортировка вставками	https://en.wikipedia.org/wiki/Insertion_sort
Introspective sort	Интроспективная сортировка	https://en.wikipedia.org/wiki/Introsort

// В стандарте не указано, как именно должен быть реализован алгоритм, даётся лишь:

1) вход 2) выход 3) ограничения на время работы 4) ограничения на память

ПРОВЕРКА $O(n)$ (`is_sorted`, `is_sorted_until`):

```
is_sorted(vec.begin(), vec.end()); // true => если отсортирован по возрастанию
is_sorted(vec.begin(), vec.end(), greater<>{}); // true => отсортирован по убыванию
auto it = is_sorted_until(vec.begin(), vec.end()); // => итератор указывающий на
            элемент, до которого все элементы отсортированы по возрастанию
auto it = is_sorted_until(vec.begin(), vec.end(), greater<>{}); // => итератор на
            элемент, до которого все элементы отсортированы по убыванию
```

СОРТИРОВКА $O(n * \log(n))$ (`sort`):

C++98 и **C++03** сложность $O(n * \log(n))$, но в худшем случае $O(n^2)$ => **quick sort**

C++11 сложность $O(n * \log(n))$ без всяких оговорок => это можно реализовать по-разному

Например GCC: использует **introspective sort**: комбинация из сортировок...

1) Сортирует как **quick sort** до глубины $\text{depth} = \log(n) * 2$

2) Сортирует как **heap sort**

3) Интервал < 16 элементов => сортирует как **insertion sort**

```
vector<int> vec = { 5, 2, 1, 4, 3 };
```

```
sort(vec.begin(), vec.end()); // по умолчанию предикат: less<int>{}
```

```
for (auto el : vec) cout << el; // => 12345
```

// Можно 3ий аргумент: **бинарный предикат** => вернёт **bool** (как сравнивать элементы)

<pre>template<typename T> bool more(T a, T b) { return a > b; } // мой бин. пред. как функция</pre>	<pre>template<typename T> struct More{ bool operator()(T a, T b) { return a > b; }; }; // мой бинарный предикат как класс</pre>
------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------

```
sort(vec.begin(), vec.end(), more<int>);
```

```
sort(vec.begin(), vec.end(), More<int>{});
```

```
for (auto el : vec) cout << el; // => 54321
```

// Но лучше использовать **стандартные бинарные предикаты** (как объекты классов):

```
sort(vec.begin(), vec.end(), greater<int>{});
```

// А ещё лучше **лямбда функции**:

```
sort(vec.begin(), vec.end(), [](auto a, auto b) { return a > b; });
```

УСТОЙЧИВАЯ СОРТИРОВКА $O(n * \log(n)^2)$ (`stable_sort`):

// Устойчивая (стабильная) => значит не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи.

Пример: пусть студенты уже рассортированы по алфавиту. С помощью устойчивой сортировки рассортируем их по возрастанию оценки => для одной и той же оценки сортировка по алфавиту сохранится.

```
stable_sort(vec.begin(), vec.end());
```

```
stable_sort(vec.begin(), vec.end(), [](auto a, auto b) { return a > b; });
```

// Согласно стандарту, её сложность в худшем случае $O(n * \log(n)^2)$

Но если достаточно памяти, то её сложность $O(n * \log(n))$

=> имеется 2 разных алгоритма устойчивой сортировки:

1) Если памяти достаточно, то используется **merge sort** (с доп. памятью)

2) Если памяти недостаточно, то используется **merge sort** (без доп. памяти)

// На маленьких интервалах обе сортировки переключаются на **insertion sort** $O(n^2)$

Эта оптимизация требует использование итераторов произвольного доступа...

Поэтому у `std::list` и `std::forward_list` есть свои `.sort()` - это только **merge sort**

ЧАСТИЧНАЯ СОРТИРОВКА $O(n * \log(m))$ (partial_sort):

// partial_sort сортирует n элементов [first, last) до места middle
// Для этого: 1) поиск m элементов middle-first минимальных
// 2) их сортировка с помощью heap sort в начале диапазона [first, last)

```
template<class RandIt, class Comp = std::less<>>
void partial_sort(RandIt first, RandIt middle, RandIt last, Comp comp = Comp()){
    if (first == middle) return;
    std::make_heap(first, middle, comp); // максимальный элемент в вершине
    for (auto it = middle; it != last; ++it) {
        if (comp(*it, *first)){//после middle нашелся тот, который < максимального
            std::pop_heap(first, middle, comp);
            std::iter_swap(middle - 1, it);
            std::push_heap(first, middle, comp);
        }
    }
    std::sort_heap(first, middle, comp);
}
```

```
std::vector<int> vec = { 0,9,2,5,4,3,6,7,8,1 };
std::partial_sort(vec.begin(), vec.begin() + 4, vec.end());
for (auto el : vec) std::cout << el; // => 0123956784
```

СОРТИРОВКА ЭЛЕМЕНТА $O(n)$ (nth_element):

// Переупорядочивает элементы из [first, last) так что:

- на месте nth стоит элемент, как если бы последовательность отсортировали
- перед nth стоят такие элементы, которые < *nth
- после nth стоят остальные (больше или равно)

```
template<class RandIt, class Comp = std::less<>>
void nth_element(RandIt first, RandIt nth, RandIt last, Comp comp = Comp());
```

// Могут рассортировать всё относительно медианного элемента:

```
std::vector<int> vec = { 5, 9, 6, 4, 3, 2, 5, 7, 9, 3, 1 };
auto middle = vec.begin() + vec.size() / 2;
std::nth_element(vec.begin(), middle, vec.end());
std::cout << "Median: " << vec[vec.size() / 2] << '\n'; // => Median: 5
for (auto el : vec) std::cout << el; // => 31234569795
```

// Могут при сортировке использовать другой компаратор

```
std::vector<int> vec = { 4,9,3,2,3 };
std::nth_element(vec.begin(), vec.begin() + 1, vec.end(), std::greater<>{});
for (auto el : vec) std::cout << el; // => 94323
```

ПОИСК НА ОТСОРТИРОВАННЫХ

НИЖНЯЯ ГРАНИЦА (lower_bound):

// => возвращает итератор на первый элемент контейнера, который не <, чем искомый
// т.е. нижняя граница включительно (если такого нет, то первый после искомого)
lower_bound(it1, it2, el); // ищет первое вхождение элемента el в диапазоне
// Реализация как в partition_point, но comp(*middle, val), а не pred(*middle)

```
template<typename ForwardIt, typename T, typename Compare = std::less<>>
ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& val, Compare comp = Compare()) {
    typename std::iterator_traits<ForwardIt>::difference_type dist, step;
    dist = std::distance(first, last);
    while (dist > 0) {
        step = dist / 2;
        ForwardIt middle = std::next(first, step); // указатель на середину
        if (comp(*middle, val)) { // значит будем искать во второй половине
            first = ++middle;
            dist -= step + 1;
        }
        else // значит будем искать в первой половине
            dist = step;
    }
    return first;
}
```

//Сложность сравнений: $O(\log n)$, но сложность итерирования зависит от типа итератора
vector<int> vec = { 1, 2, 4, 5 };
auto it = lower_bound(vec.begin(), vec.end(), 3); // it указывает на 4

ВЕРХНЯЯ ГРАНИЦА (upper_bound):

// => возвращает итератор на первый элемент контейнера, который >, чем искомый
// т.е. верхняя граница неключительно (т.е. первый после искомого)
// реализация идентичная как у lower_bound но comp(val, *middle)
vector<int> vec = { 1, 2, 4, 5 };
auto it = upper_bound(vec.begin(), vec.end(), 3); // it указывает на 4

ДИАПАЗОН (equal_range):

```
template<typename ForwardIt, typename T, typename Compare = std::less<>>
pair<ForwardIt, ForwardIt> equal_range(ForwardIt first, ForwardIt last, const T& val, Compare comp = Compare()) {
    return make_pair(lower_bound(first, last, val, comp),
                     upper_bound(first, last, val, comp));
}
```

vector<int> vec = { 1, 2, 3, 3, 3, 4, 5 };
auto [it1, it2] = equal_range(vec.begin(), vec.end(), 3);
// it1 указывает на первую не < 3 т.е. на vec[2] (= первая 3)
// it2 указывает на первую > 3 т.е. на vec[5] (= 4)

ВХОДИТ ЛИ? (binary_search)

// Метод бинарного поиска в отсортированной части последовательности: [it1, it2)

```
template<typename ForwardIt, typename T, typename Compare = std::less<>>
bool binary_search(ForwardIt first, ForwardIt last, const T& val, Compare comp = Compare()) {
    ForwardIt it = lower_bound(first, last, val, comp);
    return !(it == last || comp(*it, val));
}
```

// Для реализации понадобилось воспользоваться lower_bound
vector<double> vec = { /* ... */ }
sort(vec.begin(), vec.end()); // отсортировали массив
binary_search(vec.begin(), vec.end(), 3.14);

ОТСОРТИРОВАННЫЕ МНОЖЕСТВА

// Работа со множествами в STL предполагает, что элементы диапазона отсортированы.
// Элементы множества отсортированы с помощью какого-то компаратора => во всех операциях на множествах в качестве компаратора по умолчанию принимается `std::less`

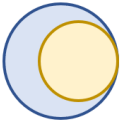
ВКЛЮЧАЕТ ЛИ $O(n_1 + n_2)$ (`includes`):

// Элементы множества отсортированы => проверку на включение можно интерпретировать как «является ли подпоследовательностью?».

// Последовательность $\{y_i\}$ является **подпоследовательность** последовательности $\{x_i\}$, если её можно получить с помощью удаления элементов из $\{x_i\}$.

// Итого: включены ли все элементы из $[first2, last2)$ в $[first1, last1)$:

```
template <class InputIt1, class InputIt2, class Comp = std::less<>>
bool includes( InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2, Comp pred = Comp{}) {
    for (; first1 != last1 && first2 != last2; ++first1) {
        if (pred(*first2, *first1))
            return false;
        if (!pred(*first1, *first2)) // передвигаю границу 2-го диапазона
            ++first2;
    }
    return first2 == last2;
}
```



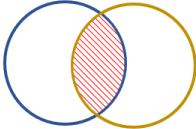
```
std::vector<int> vec = { 1, 2, 3, 4, 5 };
std::list<int> lst = { 1, 3 };
std::cout << std::boolalpha <<
    std::includes(vec.begin(), vec.end(), lst.begin(), lst.end()); // => true
```

ПЕРЕСЕЧЕНИЕ $O(n_1 + n_2)$ (`set_intersection`):

// result = $[first1, last1) \cap [first2, last2)$

// result: [dest принял в аргументы, dest вернул из алгоритма)

```
template <class InputIt1, class InputIt2, class OutputIt, class Comp=std::less<>>
OutputIt set_intersection( InputIt1 first1, InputIt1 last1,
                           InputIt2 first2, InputIt2 last2,
                           OutputIt dest, Comp pred = Comp{}) {
    while (first1 != last1 && first2 != last2) {
        if (pred(*first1, *first2)) // пропускаем уникальный элемент из [1]
            ++first1;
        else if (pred(*first2, *first1)) // пропускаем уникальный элемент из [2]
            ++first2;
        else { // элемент встретился и в [1], и в [2] => сохраним его:
            *dest++ = *first1;
            ++first1;
            ++first2;
        }
    }
    return dest;
}
```



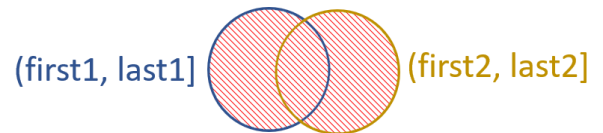
```
std::vector<int> vec = { 1, 2, 3, 5 };
std::list<int> lst = { 0, 2, 3, 4 };
std::set_intersection(vec.begin(), vec.end(), lst.begin(), lst.end(),
    std::ostream_iterator<int>(std::cout, " ")); // => 1 2 3 4 5
```

ОБЪЕДИНЕНИЕ $O(n_1 + n_2)$ (**set_union**):

// result = [first1, last1) \cup [first2, last2)

// result: [dest принял в аргументы, dest вернул из алгоритма)

```
template <class InputIt1, class InputIt2, class OutputIt, class Comp=std::less<>>
OutputIt set_union(    InputIt1 first1, InputIt1 last1,
                      InputIt2 first2, InputIt2 last2,
                      OutputIt dest, Comp pred = Comp{}) {
    for (; first1 != last1 && first2 != last2; ++dest) {
        if (pred(*first1, *first2)) // копируем уникальный из [1) и смещаем first1
            *dest = *first1++;
        else if (pred(*first2, *first1)) // копируем уник из [2) и смещаем first2
            *dest = *first2++;
        else { // значит элемент совпал в [1) и [2), копируем его и смещаем оба
            *dest = *first1;
            ++first1;
            ++first2;
        }
    }
    // Какая-то из подпоследовательностей закончилась, нужно докопировать другую:
    dest = std::copy(first1, last1, dest); // т.е. одно из этих копирований...
    dest = std::copy(first2, last2, dest); // ... будет безрезультатным
    return dest;
}
```



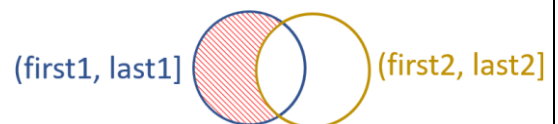
```
std::vector<int> vec = { 1, 3, 5 };
std::list<int> lst = { 2, 3, 4 };
std::set_union(vec.begin(), vec.end(), lst.begin(), lst.end(),
               std::ostream_iterator<int>{std::cout, " "}); // => 1 2 3 4 5
```

РАЗНОСТЬ $O(n_1 + n_2)$ (**set_difference**):

// result = [first1, last1) \setminus [first2, last2)

// result: [dest принял в аргументы, dest вернул из алгоритма)

```
template <class InputIt1, class InputIt2, class OutputIt, class Comp=std::less<>>
OutputIt set_difference(    InputIt1 first1, InputIt1 last1,
                           InputIt2 first2, InputIt2 last2,
                           OutputIt dest, Comp pred = Comp{}) {
    while (first1 != last1 && first2 != last2)
        if (pred(*first1, *first2)) // копирую уникальный элемент из [1)
            *dest++ = *first1++; // и сразу перехожу к следующим элементам
        else { // передвигаю границу рассмотренных
            if (!pred(*first2, *first1)) // если элементы совпали, ...
                ++first1; // ... то первую границу тоже двигаю
            ++first2; // вторую границу двигаю в обоих случаях
        }
    // докопируем все нерассмотренные элементы:
    dest = std::copy(first1, last1, dest);
    return dest;
}
```



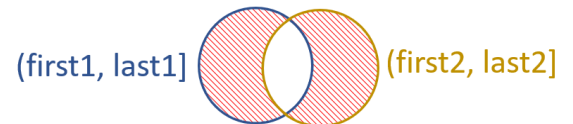
```
std::vector<int> vec = { 1, 3, 5 };
std::list<int> lst = { 2, 3, 4 };
std::set_difference(vec.begin(), vec.end(), lst.begin(), lst.end(),
                   std::ostream_iterator<int>{std::cout, " "}); // => 1 5
```


СИММЕТРИЧЕСКАЯ РАЗНОСТЬ $O(n_1 + n_2)$ (set_symmetric_difference):

// result = [first1, last1) Δ [first2, last2)

// result: [dest принял в аргументы, dest вернул из алгоритма)

```
template <class InputIt1, class InputIt2, class OutputIt, class Comp=std::less<>>
OutputIt set_symmetric_difference(    InputIt1 first1, InputIt1 last1,
                                     InputIt2 first2, InputIt2 last2,
                                     OutputIt dest, Comp pred = Comp{}) {
    while (first1 != last1 && first2 != last2) {
        if (pred(*first1, *first2))//копирую элемент из [1), которого нет в [2)
            *dest++ = *first1++;
        else if (pred(*first2, *first1))//копирую эл из [2), которого нет в [1)
            *dest++ = *first2++;
        else { // иначе оба элемента есть в [1) и [2) => не копируем их
            ++first1;
            ++first2;
        }
    }
    // Какая-то из подпоследовательностей закончилась, нужно докопировать другую:
    dest = std::copy(first1, last1, dest); // т.е. одно из этих копирований...
    dest = std::copy(first2, last2, dest); // ... будет безрезультатным
    return dest;
}
```



```
std::vector<int> vec = { 1, 3, 4, 6, 7, 9 };
```

```
std::list<int> lst = { 1, 4, 5, 6, 9 };
```

```
std::set_difference(vec.begin(), vec.end(), lst.begin(), lst.end(),
    std::ostream_iterator<int>{std::cout, " "}); // => 1 5
```

```
std::deque<int> deq;
```

```
auto visualize = [](const auto& container, int from = 1, int to = 10) {
    for (int val = from; val < to; ++val)
        std::binary_search(container.begin(), container.end(), val) ?
            std::cout << i :
            std::cout << ' ';
    std::cout << '\n';
};
```

```
std::set_symmetric_difference(vec.begin(), vec.end(), lst.begin(), lst.end(),
    std::back_inserter(deq));
```

```
visualize(vec); // => 1 3 4 6 7 9
```

```
visualize(lst); // => 1 4 5 6 9
```

```
visualize(deq); // => 3 5 7
```


СЛИЯНИЕ $O(n_1 + n_2)$ (merge):

// Слияние двух множеств – это как объединение, но повторяющиеся дублируются.

```
template <class InputIt1, class InputIt2, class OutputIt, class Comp=std::less<>>
OutputIt merge( InputIt1 first1, InputIt1 last1, InputIt2 first2, InputIt2 last2,
                OutputIt dest, Comp pred = Comp()) {
    while (first1 != last1 && first2 != last2) {
        if (pred(*first1, *first2)) // копируем уникальный элемент из [1]
            *dest++ = *first1++;
        else // копируем элемент (не обязательно уникальный) из [2]
            *dest++ = *first2++;
    }
    // Какая-то из подпоследовательностей закончилась, нужно докопировать другую:
    dest = std::copy(first1, last1, dest); // т.е. одно из этих копирований...
    dest = std::copy(first2, last2, dest); // ... будет безрезультатным
    return dest;
} // понадобилось n-1 сравнений, где n – кол-во элементов из 1 и 2 последоват-ностей

std::vector<int> v1{ 5, 2, 1, 3, 2 }, v2{ 1, 0, 3 };
std::sort(v1.begin(), v1.end(), std::greater<>{});
std::sort(v2.begin(), v2.end(), std::greater<>{});
std::vector<int> result(v1.size() + v2.size());
std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin(), std::greater<>{});
for (auto el : result) std::cout << el; // => 53322110
```

СЛИЯНИЕ НА МЕСТЕ $O(n)$ или $O(n \log(n))$ (inplace_merge):

// Последовательность состоит из двух отсортированных подпоследовательностей =>

Нужно выполнить их слияние, чтобы вся последовательность была бы отсортирована.

// Пример: 134682556789 => inplace_merge => 123455667889

// Сложность алгоритма: 1) $O(n)$ если памяти достаточно 2) Иначе $O(n \log(n))$

// Удобно использовать для реализации сортировки слиянием т.е. merge sort:

```
template<class ForwardIt, class Size, class Comp = std::less<>>
ForwardIt my_merge_sort(ForwardIt first, Size size, Comp comp = Comp()) {
    switch (size) { // Сортирует подпоследовательность от first до first+size:
        case 0: // В подпоследовательности не осталось элементов...
            return first; // ... значит ничего сортировать не нужно, вернём её начало
        case 1: // В подпоследовательности остался один элемент...
            return ++first; //... значит он единственный отсортированный, вернём next
        default: // В подпоследовательности больше одного элемента => разбить на 2
            ForwardIt middle = my_merge_sort(first, size / 2, comp);
            ForwardIt last = my_merge_sort(middle, size - size / 2, comp);
            std::inplace_merge(first, middle, last, comp); //слияние отсортированных
            return last;
    } // Возвращает конец отсортированной подпоследовательности
}

template<class Iter, class Comp = std::less<>>
void merge_sort(Iter first, Iter last, Comp comp = Comp()){
    my_merge_sort(first, std::distance(first, last), comp);
} // Промежуточный этап: приняв итераторы, нужно вычислить size

std::list<int> lst{ 5, 2, 1, 3, 2, 1, 3, 2 };
merge_sort(lst.begin(), lst.end());
for (auto el : lst) std::cout << el; // => 11222335
merge_sort(lst.begin(), lst.end(), std::greater<>{});
for (auto el : lst) std::cout << el; // => 53322211
```

СКОПИРОВАННЫЙ РЕЗУЛЬТАТ

ОБЩАЯ ИДЕЯ: исходная последовательность не затронута, а актуальная последовательность сохраняется в другой контейнер => итератор на конец актуального диапазона

ЧАСТИЧНАЯ СОРТИРОВКА $O(n * \log(\min(m, n)))$ (**partial_sort_copy**):

//partial_sort_copy сортирует n элементов[first,last) в m элементов[d_first,d_last)
// т.е. из всего отсортированного массива, только первые min(m,n) попадут в dest

```
template<class InputIt, class RandIt, class Comp = std::less<>>
void partial_sort_copy(InputIt first, InputIt last, // откуда берём элементы
                      RandIt d_first, RandIt d_last, // куда кладём результат
                      Comp comp = Comp());

std::vector<int> data = { 0,9,2,5,4,3,6,7,8,1 };
std::vector<int> result(4); // вектор из 4 элементов
std::partial_sort_copy(data.begin(), data.end(), result.begin(), result.end());
for (auto el : result) std::cout << el; // => 0123
std::partial_sort_copy(data.begin(), data.end(), result.begin(), result.end(), std::greater<>{});
for (auto el : result) std::cout << el; // => 9876
std::partial_sort_copy(result.begin(), result.end(), data.begin(), data.end(), std::greater<>{});
for (auto el : data) std::cout << el; // => 9876436781
```

РАЗДЕЛЕНИЕ $O(n)$ (**partition_copy**):

// Копирует true элементы в один диапазон, а false элементы в другой

```
template <class InputIt, class OutputIt1, class Output2, class UnaryPred>
std::pair<OutputIt1, Output2> partition_copy(
    InputIt first, InputIt last, OutputIt1 dest_true, Output2 dest_false,
    UnaryPred pred) {
    for (; first != last; ++first) {
        if (pred(*first)) {
            *dest_true = *first;
            ++dest_true;
        }
        else {
            *dest_false = *first;
            ++dest_false;
        }
    }
    return { dest_true, dest_false };
}
```

```
std::array arr = { 0, 0, 3, -1, 2, 4, 5, 0, 7 };
std::vector<int> part_true, part_false;
auto [end_true, end_false] = std::partition_copy(
    arr.begin(), arr.end(),
    std::back_inserter(part_true),
    std::back_inserter(part_false),
    [](auto el) { return el > 0; });
auto out_it = std::ostream_iterator<int>(std::cout, " ");
std::copy(part_true.begin(), part_true.end(), out_it); // => 3 2 4 5 7
std::copy(part_false.begin(), part_false.end(), out_it); // => 0 0 -1 0
```

УДАЛИТЬ $O(n)$ (remove_copy, remove_copy_if):

```
template<typename InputIt, typename OutputIt, typename T>
OutputIt remove_copy(InputIt first, InputIt last, OutputIt dest, const T& value){
    for (; first != last; ++first)
        if (!(*first == value))
            *dest++ = *first;
    return dest;
}
```

```
template<typename InputIt, typename OutputIt, typename UnaryPredicate>
OutputIt remove_copy_if(InputIt first, InputIt last, OutputIt dest, UnaryPredicate pred){
    for (; first != last; ++first)
        if (!pred(*first))
            *dest++ = *first;
    return dest;
}
```

```
std::string str = "1 2 3 4 5 6";
std::remove_copy(str.begin(), str.end(),
std::ostream_iterator<char>(std::cout, ' ')); // => 123456
std::cout << str; // => 1 2 3 4 5 6 (т.е. так и не изменился)
```

ЗАПОЛНИТЬ РЕЗУЛЬТАТОМ ФУНКЦИИ $O(n)$ (replace_copy, replace_copy_if):

```
template< typename InputIt, typename OutputIt, typename T>
OutputIt replace_copy( InputIt first, InputIt last, OutputIt dest,
                      const T& old_value, const T& new_value) {
    for (; first != last; ++first, ++dest)
        *dest = (*first == old_value) ? new_value : *first;
    return dest;
}
```

```
template<typename InputIt, typename OutputIt, typename UnaryPredicate, typename T>
OutputIt replace_copy_if( InputIt first, InputIt last, OutputIt dest,
                        UnaryPredicate pred, const T& new_value) {
    for (; first != last; ++first, ++dest)
        *dest = pred(*first) ? new_value : *first;
    return dest;
}
```

```
std::string str = "1 2 3 4 5 6";
std::replace_copy(str.begin(), str.end(),
std::ostream_iterator<char>(std::cout, ' ', ',')); // => 1,2,3,4,5,6
std::cout << str; // => 1 2 3 4 5 6 (т.е. так и не изменился)
```

ПРОВЕРНУТЬ $O(n)$ (rotate_copy):

```
template<typename ForwardIt, typename OutputIt>
OutputIt rotate_copy(ForwardIt first, ForwardIt mid, ForwardIt last, OutputIt dest) {
    dest = std::copy(mid, last, dest);
    return std::copy(first, mid, dest);
}
```

```
std::string str = "1 2 3 4 5 6";
std::rotate_copy(str.begin(), std::next(str.begin(),4), str.end(),
std::ostream_iterator<char>(std::cout)); // => 3 4 5 6 1 2
std::cout << str; // => 1 2 3 4 5 6 (т.е. так и не изменился)
```

ПЕРЕВЕРСИРОВАТЬ $O(n)$ (reverse_copy):

```
template<typename BidIt, typename OutputIt>
OutputIt reverse_copy(BidIt first, BidIt last, OutputIt dest) {
    for (; first != last; ++dest)
        *dest = *(--last);
    return dest;
}
```

```
std::string str = "1 2 3 4 5 6";
std::reverse_copy(str.begin(), str.end(),
std::ostream_iterator<char>(std::cout)); // => 6 5 4 3 2 1
std::cout << str; // => 1 2 3 4 5 6 (т.е. так и не изменился)
```

БЕЗ ПОВТОРЕНИЙ ПОДРЯД $O(n)$ (unique_copy):

```
template<typename InputIt, typename OutputIt, typename BinPred = std::equal_to<>>
OutputIt unique_copy(InputIt first, InputIt last, OutputIt dest, BinPred pred = BinPred()) {
    if (first == last)
        return dest;
    *dest = *first;
    while (++first != last)
        if (!pred(*dest, *first)) //если новый рассм-ый отличается от актуального...
            *++dest = *first; // ... то копировать его в следующий актуальный
    return ++dest; // вернём конец актуального диапазона
}
```

// Красивая реализация, но это может не сработать для Output категории итератора:
Требования на OutputIt не накладывают ограничений на тип возвращ. значение у *it

Rereadable: такой it, что *it возвращает val записанный ранее в *it=value.

Рассмотрим std::ostream_iterator, у которого категория Output, но он НЕ rereadable

*it у него с пустым телом и возвращает его самого т.е. it, а значит...

pred(*dest, *first) идентично pred(dest, *first) => ошибка компиляции

// => для таких сравнений нужно иметь buffer на 1 элемент, который использовать,
чтобы задать значение *dest=buffer и после в предикате т.е. pred(buffer, *first)

```
template<typename InputIt, typename OutputIt, typename BinPred = std::equal_to<>>
OutputIt unique_copy(InputIt first, InputIt last, OutputIt dest, BinPred pred = BinPred()) {
    if (first == last)
        return dest;
    auto buffer = *first; // положим в буфер элемент
    *dest = buffer; // заполним элементов из буфера актуальный элемент
    while (++first != last)
        if (!pred(buffer, *first)) { //если новый рассм-ый отличается от актуального...
            buffer = *first; // ... положим в буфер этот новый рассматриваемый
            *++dest = buffer; // заполним элементов из буфера следующий актуальный
        }
    return ++dest; // вернём конец актуального диапазона
}
```

// Примеры с предикатом (убывание = повторение, а возрастание = уникальный):

```
std::vector<int> vec = { 1,2,1,3,4,1,5,6,3,2,7,1 };
```

```
std::unique_copy(vec.begin(), vec.end(),
std::ostream_iterator<int>(std::cout, " "),
std::greater<>{}); // => 1 2 3 4 5 6 7
```

НА НЕИНИЦИАЛИЗИРОВАННОЙ ПАМЯТИ

```
#include <memory>
```

КОПИРОВАТЬ (uninitialized_copy, uninitialized_copy_n):

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_copy(InputIt first, InputIt last, ForwardIt dest_first){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = dest_first;
    try { // Пытается разместить элементы в памяти
        for (; first != last; ++first, ++current)
            ::new (std::addressof(*current)) T(*first); // с помощью констр. копи.
        return current;
    }
    catch (...) { // Если какой-то конструктор бросил исключение,...
        for (; dest_first != current; ++dest_first)
            dest_first->~T(); //... запустит деструктор для уже созданных объектов
        throw;
    }
}
```

```
template<class InputIt, class Size, class ForwardIt>
ForwardIt uninitialized_copy_n(InputIt first, Size count, ForwardIt dest_first){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = dest_first;
    try {
        for (; count > 0; ++first, ++current, --count)
            ::new (std::addressof(*current)) T(*first);
        return current;
    }
    catch (...) {
        for (; dest_first != current; ++dest_first)
            dest_first->~T();
        throw;
    }
}
```

// Выделение памяти и освобождение памяти в случае неудачи происходит снаружи
Продemonстрируем это на старом примере безопасного MyArr:

```
MyArr::MyArr(const MyArr& object) {
    T* ptr_temp = reinterpret_cast<T*>(operator new(sizeof(T) * object.m_size));
    try {
        std::uninitialized_copy_n(object.m_ptr, object.m_size, ptr_temp);
    }
    catch (...) { // поймали исключения => деструкторы уже отработали...
        delete[] reinterpret_cast<void*>(ptr_temp); // освободим память
        throw; // прокинем исключение дальше
    }
    m_ptr = ptr_temp;
    m_size = object.m_size;
}
```

ПЕРЕМЕСТИТЬ (uninitialized_move, uninitialized_move_n):

// Такие же как и uninitialized_copy, uninitialized_copy_n, кроме этой строчки:
::new (std::addressof(*current)) T(std::move(*first));

ЗАПОЛНИТЬ (uninitialized_fill, uninitialized_fill_n):

Отличие от описанных ранее: сразу даётся диапазон с которым работаем: [first,last)
// Ранее давался диапазон [first,last), а работали с [dest_first, ...)

```
template<class ForwardIt, class Type>
void uninitialized_fill(ForwardIt first, ForwardIt last, const Type& value){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = first;
    try {
        for (; current != last; ++current)
            ::new (std::addressof(*current)) T(value);
    }
    catch (...) {
        for (; first != current; ++first)
            first->~T();
        throw;
    }
}

template< class ForwardIt, class Size, class Type>
ForwardIt uninitialized_fill_n(ForwardIt first, Size count, const Type& value){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = first;
    try {
        for (; count > 0; ++current, --count)
            ::new (std::addressof(*current)) T(value);
        return current;
    }
    catch (...) {
        for (; first != current; ++first)
            first->~T();
        throw;
    }
}
```

```
size_t size = 4;
std::string * ptr_str = reinterpret_cast<std::string*>(operator new (sizeof(std::string) * size));
mystd::uninitialized_fill(ptr_str, ptr_str+size, "Hello");
for (auto ptr = ptr_str; ptr != ptr_str+size; ++ptr)
    std::cout << *ptr << ' '; // => Hello Hello Hello Hello
```


DEFAULT-ИНИЦИАЛИЗАЦИЯ (uninitialized_default_construct, uninitialized_default_construct_n):

```
template<class ForwardIt>
void uninitialized_default_construct(ForwardIt first, ForwardIt last){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = first;
    try {
        for (; current != last; ++current)
            ::new (std::addressof(*current)) T;
    }
    catch (...) {
        for (; first != current; ++first)
            first->~T();
        throw;
    }
}
```

```
template<class ForwardIt, class Size>
ForwardIt uninitialized_default_construct_n(ForwardIt first, Size n){
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    ForwardIt current = first;
    try {
        for (; n > 0; ++current, --n)
            ::new (std::addressof(*current)) T;
        return current;
    }
    catch (...) {
        for (; first != current; ++first)
            first->~T();
        throw;
    }
}
```

VALUE-ИНИЦИАЛИЗАЦИЯ (uninitialized_value_construct, uninitialized_value_construct_n):

// Такие же как и uninitialized_default_construct, uninitialized_default_construct_n, кроме:
::new (std::addressof(*current)) T();

//Value VS Default инициализация: различия будут при работе со встроенными типами:

```
size_t size = 4;
```

```
int* arr = reinterpret_cast<int*>(new char[sizeof(int) * size]);
```

```
std::uninitialized_default_construct(arr, arr + size);
```

```
for (auto ptr = arr; ptr != arr + size; ++ptr)
```

```
std::cout << *ptr << ' '; // => -842150451 -842150451 -842150451 -842150451
```

```
std::uninitialized_value_construct(arr, arr + size);
```

```
for (auto ptr = arr; ptr != arr + size; ++ptr)
```

```
std::cout << *ptr << ' '; // => 0 0 0 0
```

СКОНСТРУИРОВАТЬ ИЛИ РАЗРУШИТЬ ПО АДРЕСУ (construct_at, destroy_at):

```
template <class T, class... Args>
T* construct_at(T* const ptr, Args&&... args) {
    return ::new (ptr) T(std::forward<Args>(args)...);
}

template<class T>
constexpr void destroy_at(T* p){
    if constexpr (std::is_array_v<T>) // если это массив, у него нет деструкторов,
        for (auto& elem : *p) // поэтому придётся пройти по каждому его элементу
            destroy_at(std::addressof(elem)); // и рекурсивно запустить себя
    else // если это не массив, то...
        p->~T(); // ... достаточно просто запустить деструктор
}
```

// Понадобится в примерах:

```
struct MyClass {
    MyClass() { std::cout << "Construct\n"; }
    ~MyClass() { std::cout << "Destroy\n"; }
};

const size_t size = 3;
```

```
// Поработаем с динамическим выделением и освобождением памяти
MyClass* arr = reinterpret_cast<MyClass*>(operator new(sizeof(MyClass) * size));
for (auto ptr = arr; ptr != arr + size; ++ptr)
    std::construct_at(ptr); // => Construct Construct Construct
for (auto ptr = arr; ptr != arr + size; ++ptr)
    std::destroy_at(ptr); // => Destroy Destroy Destroy
delete[] reinterpret_cast<void*>(arr);
```

```
// Поработаем с типом данных массив:
MyClass arr[size]; // => Construct Construct Construct
mystd::destroy_at(&arr); // => Destroy Destroy Destroy
```

ДЕСТРУКТОРЫ (destroy, destroy_n):

```
template<class ForwardIt>
void destroy(ForwardIt first, ForwardIt last){
    for (; first != last; ++first)
        std::destroy_at(std::addressof(*first));
}
```

```
template<class ForwardIt, class Size>
ForwardIt destroy_n(ForwardIt first, Size n) {
    for (; n > 0; ++first, --n)
        std::destroy_at(std::addressof(*first));
    return first;
}
```

```
MyClass* arr = reinterpret_cast<MyClass*>(operator new(sizeof(MyClass) * size));
for (auto ptr = arr; ptr != arr + size; ++ptr)
    new(ptr) MyClass(); // => Construct Construct Construct
mystd::destroy(arr, arr + size); // => Destroy Destroy Destroy
mystd::destroy_n(arr, size); // => Destroy Destroy Destroy
delete[] reinterpret_cast<void*>(arr);
```

НЕОПРЕДЕЛЁННЫЙ ТИП

```
#include <cstdlib>
```

СОРТИРОВКА (qsort):

// Сортирует презаписывая элементы побайтово => это сработает только для элементов, которые удовлетворяют **TriviallyCopyable** (т.е. такие, для которых сохранив побайтово ресурсы класса удастся позже воссоздать объект из этой копии).

```
void qsort(void* ptr, std::size_t count, std::size_t size, comparator_t* comp);
```

ptr – начало массива с данными, которые требуется отсортировать

count – количество элементов массива

size – количество байт в одном элементе

comp – компаратор, который должен иметь следующий интерфейс:

```
using comparator_t = int(const void*, const void*);
```

Возвращаемое значение: <1 - левый меньше правого, >1 - больше, 0 - равны

Примечание: несмотря на название (будто QuickSort), стандарт не накладывает требования на конкретную реализацию, сложность или устойчивость.

```
template<typename T>
```

```
int comp(const void* ptr_left, const void* ptr_right) {  
    const auto& left = *reinterpret_cast<const T*>(ptr_left);  
    const auto& right = *reinterpret_cast<const T*>(ptr_right);  
    if (left < right) return -1;  
    if (right < left) return 1;  
    return 0;  
}
```

```
std::array arr = { 1,3,2,4,1,2,3,1,2 };
```

```
std::qsort(arr.data(), arr.size(), sizeof(arr[0]), comp<int>);
```

```
std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, " "));
```

```
// => 1 1 1 2 2 2 3 3 4
```

ПОИСК ЭЛЕМЕНТА (bsearch):

// Ищет элемент в отсортированном массиве (согласно критерию использованного компаратора). На неотсортированном массиве получится UB.

```
void* bsearch(const void* key, const void* ptr, std::size_t count,  
             std::size_t size, comparator_t* comp);
```

ptr – начало массива с данными, которые требуется отсортировать

count – количество элементов массива

size – количество байт в одном элементе

comp – компаратор аналогичный тому, как это описано для **qsort**

Возвращаемое значение: указатель на найденный элемент или **nullptr**, если не найден.

Примечание: несмотря на название (будто BinarySearch), стандарт не накладывает требования на конкретную реализацию или сложность.

```
template<typename Container, typename T, typename Comp>
```

```
void try_bsearch(const Container& arr, const T& val, Comp comp) {  
    void* ptr_el = std::bsearch(&val, arr.data(), arr.size(), sizeof(arr[0]), comp);  
    if (ptr_el != nullptr)  
        std::cout << "Found: " << *reinterpret_cast<T*>(ptr_el) << '\n';  
    else  
        std::cout << "Element was not found!\n";  
}
```

// Воспользуемся компаратором из прошлого раздела:

```
std::array arr = { 1,2,4,1,2,3,1,2,3 };
```

```
std::qsort(arr.data(), arr.size(), sizeof(arr[0]), comp<int>); //без этого будет UB
```

```
try_bsearch(arr, 3, comp<int>); // => Found: 3
```

```
try_bsearch(arr, 13, comp<int>); // => Element was not found!
```