პრაქტიკული 4: სწრაფი დახარისხების ალგორითმი - qSort

სააუდიტორიო სამუშაო:

- ისეთი იტერატორის ძებნა, სადაც მოცემული თვისება არ სრულდება
- დიაპაზონში ობიექტების გადანაწილება თვისების ქონა/არქონის მიხედვით >>>
- სწრაფი დახარისხება >>>
- სავარჯიშოები >>>

იტერატორის ძებნა, სადაც მოცემული თვისება არ სრულდება. `[first,last)` ფრაგმენტში მოვძებნოთ დასაწყისიდან პირველივე ისეთი ობიექტის იტერატორი, რომელიც არ აკმაყოფილებს გარკვეულ პირობას.

C++11-ის სტანდარტულ ბიბლიოთეკაში ამ საქმეს აკეთებს შემდეგი ალგორითმი:

```
template<class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q)
{
	for (; first != last; ++first) {
		if (!q(*first)) {
			return first;
		}
	}
	return last;
}
```

სხვადასხვა მაგალითზე ვნახოთ, თუ როგორ მუშაობს იგი ნაბიჯ-ნაბიჯ, ალგორითმის ძირითადი სტრიქონების შესაბამისად. სხვა სიტყვებით, ვნახოთ მისი ტრასირება.

მაგალითად, დავუშვათ რომ `[first,last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5 -23, 55, 6. მოვძებნოთ ისეთი ობიექტის იტერატორი, რომლიც არაა დადებითი. ამ შემთხვევაში, `q(*first)` არის ჭეშმარიტი მაშინ, როდესაც `*first` არის დადებითი. `for` განმეორებამ უნდა შეწყვიტოს მუშაობა მაშინ, როდესაც ჭეშმარიტი არის `(!q(*first)`, ანუ `*first` არის უარყოფითი. შესაბამისად, როდესაც `first` მიუთითებს -23-ის პოზიციას, მაშინ განმეორება შეწყდება და ალგორითმი დააბრუნებს იტერატორის ამ მნიშვნელობას.

თუ ამ დიაპაზონისთვის გვენდომება ისეთი იტერატორის მოძებნა, სადაც არ სრულდება პირობა (პრედიკატი): `[](int n){return n<10;};` მაშინ დაბრუნდება დიაპაზონის პირველივე მონაცემის პოზიცია, ანუ იტერატორი.

<<< დიაპაზონში ობიექტების გადანაწილება თვისების ქონა/არქონის მიხედვით. მოცემულია `[first,last)` ფრაგმენტი და უნარული პრედიკატი (თვისება). ფრაგმენტში ობიექტები უნდა გადანაწილდეს ორ ჯგუფად: პირველი ჯგუფის ყველა ელემენტს აქვს მოცემული თვისება, ხოლო მეორე ჯგუფის არცერთ ელემენტს არ აქვს იგივე თვისება. ალგორითმმა უნდა დააბრუნოს მეორე ჯგუფის პირველივე ობიექტის იტერატორი.

C++11-ის სტანდარტულ ბიბლიოთეკაში ამ საქმეს აკეთებს შემდეგი ალგორითმი:

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
	first = std::find_if_not(first, last, p);
	if (first == last) return first;

	for (ForwardIt i = std::next(first); i != last; ++i) {
		if (p(*i)) {
			std::iter_swap(i, first);
			++first;
		}
	}
	return first;
```

}

იტერატორების ეს კატეგორია აუცილებელია, რადგან ალგორითმი იმახსოვრებს იტერატორებს დიაპაზონის ფრაგმენტების ხელმეორედ გავლის მიზნით.

სხვადასხვა მაგალითზე აღვწეროთ, თუ როგორ მუშაობს იგი ნაბიჯ-ნაბიჯ, ალგორითმის ძირითადი სტრიქონების შესაბამისად (დავაკვირდეთ მის ტრასირებას).

დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5, -23, 55, 6, 17. თვისება, რომლის მიხედვითაც გადავანაწილებთ, არის: „ელემენტი ნაკლები უნდა იყოს 10-ზე“.

პირველი სტრიქონი განსაზღვრავს, რომ `first` იგივე ელემენტის, 10-იანის პოზიციას მიუთითებს, რადგან პირველივეზე დაირღვა თვისება.

განმეორების შეტყობინებაში ვნახოთ თუ როგორ იცვლება იტერატორები და მიმდევრობა (იტერატორს ფრჩხილებში ვუწერთ იმ მნიშვნელობას, რომელსაც მიუთითებს):

`i(5)`	`p(*i)) = true`	5, 10, -23, 55, 6, 17	`++first: first(10)`
`i(-23)`	`p(*i)) = true`	5, -23, 10, 55, 6, 17	`++first: first(10)`
`i(55)`	`p(*i)) = false`	5, -23, 10, 55, 6, 17	`: first(10)`
`i(6)`	`p(*i)) = true`	5, -23, 6, 55, 10, 17	`++first: first(55)`
`i(17)`	`p(*i)) = false`	5, -23, 6, 55, 10, 17	`: first(55)`

ალგორითმი დააბრუნებს იტერატორს, რომელიც მიუთითებს 55-ზე (გადანაწილებულ მიმდევრობაში).

<<< სწრაფი დახარისხება. ალგორითმის მიზანია `[first,last)` ფრაგმენტში ელემენტების დახარისხება ზრდადობის მიხედვით. ვიგულისხმოთ, რომ დიაპაზონში ჩაწერილი ელემენტებისთვის განსაზღვრულია წრფივი დალაგების მიმართება „ნაკლები ან ტოლი“.

სწრაფი დახარისხების იდეა ასეთია: საწყის მიმდევრობაში მონაცემებს ორ ჯგუფად ისე, რომ მარცხენა მხარიდან ნებისმიერი ელემენტი ნაკლებია ან ტოლი მარჯვენა მხარის ნებისმიერ ელემენტზე. შემდეგ, პროცესი რეკურსიულად გაგრძელდება:

```
template<typename ForwaedIt>
void qSort(ForwaedIt first, ForwaedIt last)
{
	if ( distance(first, last) > 1 )
	{
		auto x = *first;
		auto lm = [=](decltype(x) y) {return (y < x); };
		auto q = partition(first, last, lm);
		if (q == first) q = std::next(q);

		qSort(first, q);
		qSort(q, last);
	}
}
```

დიაპაზონის ორ ნაწილად გადანაწილებისთვის შეგვიძლია გამოვიყენოთ ზემოთ მოყვანილი `partition` ალგორითმი. თავის მხრივ, `partition` ალგორითმს სჭირდება თვისება, რომელიც ორად გაყოფს დიაპაზონს. სწრაფი დახარისხების ალგორითმის ჩვენს მარტივ იმპლემენტაციაში, თვისება არის ასეთი: ელემენტი ნაკლებია დიაპაზონის პირველ ელემენტზე“. გადანაწილების შემდეგ,

$$[first,last) = [first,q) \cup [q,last).$$

იმისათვის რომ რეკურსიამ იმუშაოს, ყოველი დიაპაზონის ორივე ნაწილი უნდა იყოს არაცარიელი. თუ შემთხვევით აღმოჩნდა, რომ რომელიღაც ბიჯზე დიაპაზონის პირველი ელემენტი უმცირესიცაა, მაშინ `q = first`, შედეგად `[first,last) = [q,last)` და ალგორითმი დაიციკლება. ამისგან თავდასაცავად არის სტრიქონი `if (q == first) q = std::next(q);`

კიდევ ერთი სტრიქონი: `auto lm = [=](decltype(x) y) {return (y < x); };` ქმნის თვისებას. აქ `x` არის დიაპაზონის პირველი ელემენტი, ხოლო განაცხადი `decltype(x) y` ნიშნავს რომ `y` ცვლადი იგივე ტიპისაა რაც `x`.

განვიხილოთ ალგორითმი კერძო მონაცემებზე. ვთქვათ, გვინდა 10, 5, 55, 6 მიმდევრობის სწრაფი დახარისხება.

```
qSort[10, 5, 55, 6]
{
	distance[10, 5, 55, 6]= 4 > 1  ☺
	{
		x = 10;
		lm = [=](int y) {return y < 10; };
		partition[10, 5, 55, 6] = [5, 6] ∪ [55, 10];
			// q = არის 55-ის პოზიცია მიმდევრობაში,
			// ამ მომენტში მიმდევრობა ასეთია:  5, 6, 55, 10

		qSort[5, 6]
			distance[5,6] = 2 > 1 ☺
			{
				x = 5;
				lm = [=](decltype(x) y) {return (y < 5); };
				partition[5, 6] = {} ∪ [5, 6];    //*q = 5
				if (q == first) q = std::next(q); ☺
				// გადანაწილება გადავახალისეთ: [5, 6] = [5] ∪ [6];
				//შესაბამისად, იყო*q = 5 მაგრამ q გახდა 6-ის პოზიცია
				qSort[5] {};  // distance = 0
				qSort[6] {};  // distance = 1
				// qSort[5, 6] გამოძახებამ არაფერი შეცვალა,
			}
		qSort[55, 10];
			distance[55, 10] = 2 > 1 ☺
			// ამ ჯერზე,  *fisrst = 55
			{
				x = 55;
				lm = [=](decltype(x) y) {return (y < 55); };
				partition[55,10] = [10] ∪ [55]
				// 55, 10 მიმდევრობა გახდა 10, 55,  ხოლო *q = 55
				qSort[10] {};  // distance = 1
				qSort[55] {};  // distance = 1
			}
	}
}
```

შესაძლოა, უფრო მოსახერხებელი სახეა:

10	5	55	6
5	6	55	10
5	6		
5			
	6		
		10	55
		10	
			55
5	6	10	55

<<< სავარჯიშოები:

1. დავუშვათ რომ `[first,last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 15, 23, 55, 76. მოვძებნოთ ისეთი ობიექტის იტერატორი, რომლიც არაა:

   ა) 76-ზე ნაკლები; ბ) 15-ზე ნაკლები; გ) არაა ლუწი; დ) არაა ორნიშნა.

2. დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 55, 15, 10, 76, 23. მოვძებნოთ ისეთი ობიექტის იტერატორი, რომლიც არაა:

   ა) 76-ზე ნაკლები; ბ) 15-ზე ნაკლები; გ) არაა ლუწი; დ) არაა ორნიშნა.

3. დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5, -23, 55, 6, 17. თვისება, რომლის მიხედვითაც გადავანაწილებთ, არის: „ელემენტი ნაკლები უნდა იყოს 55-ზე“. აჩვენეთ, თუ როგორ იმუშავებს გადანაწილების ალგორითმი, ძირითადი შესრულებადი შეტყობინებების შედეგების მითითებით.

4. დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5, -23, 55, 6, 17. თვისება, რომლის მიხედვითაც გადავანაწილებთ, არის: „ელემენტი ნაკლები უნდა იყოს -23-ზე“. აჩვენეთ, თუ როგორ იმუშავებს გადანაწილების ალგორითმი, ძირითადი შესრულებადი შეტყობინებების შედეგების მითითებით.

5. დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5, -23, 55, 6, 17. თვისება, რომლის მიხედვითაც გადავანაწილებთ, არის: „ელემენტი ნაკლები უნდა იყოს 17-ზე“. აჩვენეთ, თუ როგორ იმუშავებს გადანაწილების ალგორითმი, ძირითადი შესრულებადი შეტყობინებების შედეგების მითითებით.

6. დავუშვათ რომ `[first, last)` დიაპაზონში ჩაწერილია რიცხვები: 10, 5, -23, 55, 6, 17. თვისება, რომლის მიხედვითაც გადავანაწილებთ, არის: „ელემენტი უნდა იყოს კენტი“. აღწერეთ ალგორთმის ტრასირება.

7. რამდენიმე 3 და 4 ელემენტიან რიცხვით მიმდევრობაზე გასინჯეთ, თუ როგორ მუშაობს სწრაფი დახარისხების ალგორითმის კოდი.

8. მოიყვანეთ `find_if_not` ალგორითმის ისეთი იმპლემენტაცია, რომელიც გამოიყენებს `while` შეტყობინებას.

9. ინტერნეტში მოძებნეთ შემდეგი ალგორითმები: `search_N, fill, fill_N, inner_product`. გაეცანით მათ კოდებს და გამოყენების მაგალითებს.

10. შეადგინეთ ალგორითმი, რომელიც დაითვლის მოცემულ დიაპაზონში ელემენტების ჯამს. მოიყვანეთ მისი გამოყენების მაგალითები.

11. შეადგინეთ ალგორითმი, რომელიც დაითვლის მოცემულ დიაპაზონში გარკვეული თვისების მქონე ელემენტების ჯამს. მოიყვანეთ მისი გამოყენების მაგალითები.