Zadanie 0 – sortowanie przez wybieranie (1,5 pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void selectionSort(ForwardIt begin, ForwardIt end)
```

wykonującej sortowanie przez wybieranie pomiedzy dwoma iteratorami jednokierunkowymi.
Do zamiany wartości dwóch iteratorów a i b możesz użyć **std::iter swap(a,b)** z **¡algorithm¿** lub **std::swap(*a,*b)**.
Do znalezienia iteratora do najmniejszego elementu pomiedzy iteratorami first i last możesz użyć funkcji **std::min_element(first,last)** z **¡algorithm¿**.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void selectionSort(ForwardIt begin, ForwardIt end);

int main() {
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  selectionSort(v.begin(), v.end());
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  selectionSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //list
  std::list<int> l = { 7, 5, 16, 8 };
  selectionSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.

Zadanie 1 – Sortowanie przez wstawianie (1,5 pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void insertionSort(ForwardIt begin, ForwardIt end);
```

realizującej sortowanie przez wstawianie. Nie będziesz potrzebował(a) iteratorów dwukierunkowych, jeżeli wykorzystasz następujące funkcje z biblioteki standardowej, z nagłówka **¡algorithm¿**:

- **std::upper_bound(start, stop, v)** – zwraca iterator z zakresu od iteratora start do iteratora stop, który odpowiada pierwszej napotkanej wartości większej od v.
- **std::next(it)** – zwraca następny iterator po iteratorze it (odpowiednik użycia ++, nie zmienia it).
- std::rotate(start, val, stop) – dokonuje przesunięcia w lewo dla zakresu od iteratora start do iteratora stop. Przesunięcie będzie takie, żeby po jego dokonaniu iterator val stał na początku zakresu (tam gdzie był start).

Szablon

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void insertionSort(ForwardIt begin, ForwardIt end);

int main() {
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  insertionSort(v.begin(), v.end());
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;

  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  insertionSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;

  //list
  std::list<int> l = { 7, 5, 16, 8, -3 };
  insertionSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.

Iteratory dostępu bezpośredniego

W zadaniach tej serii należy założyć, że iteratory przekazywane do funkcji są *iteratorami dostępu bezpośredniego*, tzn. są dla nich zdefiniowane operatory: $++, --, +, -, +=, -=, ==, !=, <, >, <=, >=$. Tego typu są iteratory dla obiektów klasy *std* :: *vector* <>, których używaliśmy wielokrotnie do tej pory.

Zadanie 2 – sortowanie szybkie (2 pkt)

Napisz szablon funkcji

```
template <class RandomIt>
void quickSort(RandomIt begin, RandomIt end)
```

wykonującej szybkie sortowanie pomiędzy dwoma iteratorami dostępu bezpośredniego. Algorytm rekurencyjny sortowania szybkiego:

1. Jeżeli sortowany ciąg nie jest jednoelementowy to wykonaj natępne kroki.
2. Wybierz jeden z elementów, np. pierwszy. Nazwijmy ten element PIVOT.
3. Przenieś wszystkie elementy mniejsze od PIVOT na lewo od niego, a wszystkie elementy większe od niego na prawo. W ten sposób PIVOT będzie na właściwym miejscu (posortowany).
4. Powtórz rekurencyjnie dla dwóch tablic: pierwszej składającej się z liczb stojących przed PIVOTem, drugiej dla liczb stojących za PIVOTem.

Przydatne będzie użycie pomocniczej funkcji dokonującej zamiany miejscami i zwracającej iterator na PIVOT

```
template <class RandomIt>
RandomIt partition(RandomIt begin, RandomIt end);
```

Szablon:

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

template <class RandomIt>
RandomIt partition(RandomIt begin, RandomIt end);

template <class RandomIt>
void quickSort(RandomIt begin, RandomIt end);

int main()
{
        std::vector<int> v = {11, -3, 2, 4, 0, 6, 4, -2, 2, 3, 1};
        quickSort(v.begin(), v.end());
        std::for_each(v.begin(), v.end(), [](auto x)
                { std::cout << x << " "; });
        std::cout<<std::endl;
        return 0;
}
```

Uwaga: Nie tworzyc nowych tablic ani kolekcji. Algorytm ma być zaimplementowany rekurencyjnie i ma być poprawnym sortowaniem szybkim, a nie innym rodzajem. Nie używać funkcji sortujących z bibliotek, tylko napisać własną implementację.
Podpowiedź: Przydatne będą funkcje, których używaliśmy do działań na iteratorach w zadaniach na poprzednich zajęciach.
Dla chętnych: Dodaj opcję, żeby przy wywołaniu szablonu można było zdecydować, czy sortowanie ma być rosnące czy malejące.

Zadanie 3 – liczby zespolone (1pkt)

Napisz klasę Complex, która będzie reprezentować liczbę zespoloną. Klasa powinna mieć 4 pola: część rzeczywistą, część urojoną, moduł, fazę. Powinna mieć dwa konstruktory: domyślny (bez argumentów), który ustawia wszystkie pola na 0, oraz konstruktor przyjmujący dwa parametry będące częścią rzeczywistą i urojoną liczby zespolonej. Dodatkowo klasa powinna zawierać dwie metody:

```
void Complex::printAlg()
```

drukującą na standardowe wyjście algebraiczną postać liczby zespolonej, tj. $x \pm i * y$. Oraz drugą metodę:

```
void Complex::printExp()
```

drukującą liczbę zespoloną w postaci eksponencjalnej, tj. $R * exp(i * \phi)$. Jeżeli faza jest ujemna, to przed jednostką urojoną powinien stać minus, jeżeli dodatnia to nic nie powinno stać.
Dla ułatwienia, wszystkie pola i metody klasy mogą mieć specyfikator dostępu *public*.
Dodatkowo, uzupełnij maina tak, żeby utworzyć *std* :: *vector* o nazwie v, który będzie zawierał obiekty typu Complex. Następnie dodaj do niego elementy odpowiadające liczbom:
$(\sqrt{3}/2, 1/2), (-1/2, \sqrt{3}/2), (-1, 0), (0, -1)$

Szablon programu:

```
#include <iostream>
#include <cmath>
#include <vector>

//tu napisz klase Complex

int main()
{
        //tu stworz wektor i wypelnij

        for(auto iter = v.begin(); iter!=v.end(); iter++)
        {
                iter -> printAlg();
                iter -> printExp();
                std::cout << std::endl;
        }

        return 0;
}
```

Output:
0.866025 + i*0.5
1*exp(i0.523599)

-0.5 + i*0.866025
1*exp(i2.0944)

-1 + i*0
1*exp(i3.14159)

0 - i*1
1*exp(-i1.5708)

Uwaga: Nie trzeba alokować pamięci dynamicznie, ale jeśli ktoś chce, to musi pamiętać o jej zwolnieniu.
Podpowiedź: Do liczenia fazy może się przydać funkcja *tan*2(). W konstruktorach pomocne może być użycie wskaźnika *this*. Wektor typu Complex tworzy i wypełnia się dokładnie w taki sam sposób jak wektory typów wbudowanych.
Dla chętnych: Napisz funkcje, które będą dodawać, odejmować, mnożyć, dzielić 2 liczby zespolone. Można to zrobić tak, że funkcje będą metodami klasy Complex, przyjmującymi jeden argument typu Complex (albo Complex*). Wtedy dodanie dwóch liczb A i B wyglądać będzie: "A.add(B);" i w wyniku tego wywołania liczba A będzie teraz sumą starego A i B.