Zadanie 0 – sortowanie przez scalanie (1pkt)

Napisz szablon funkcji

```
template <class RandomIter>
void mergeSort(RandomIter begin, RandomIter end)
```

wykonującej sortowanie przez scalanie pomiędzy dwoma iteratorami swobodnego dostępu.
Algorytm **rekurencyjny** sortowania przez scalanie:

1. Jeżeli sortowany ciąg nie jest jednoelementowy wykonaj następne kroki.
2. Znajdź środkowy element ciągu. Podziel ciąg na dwa podciągi: pierwszy od elementu pierwszego do środkowego, drugi od elementu środkowego do ostatniego elementu.
3. Wykonaj sortowanie przez scalanie dla obu podciągów.
4. Scal posortowane podciągi.

Najtrudniejszym elementem sortowania jest procedura scalania posortowanych podciągów. Zamiast pisać własną możesz wykorzystać algorytm biblioteki standardowej. Do scalania posortowanych podciągów (pierwszy od iteratora first do middlle, drugi od middle do last) możesz użyć $std :: inplace\_merge(first, middle, last)$.
Zadanie z użyciem tej funkcji staje się bardzo proste.
Jeżeli napiszesz własną procedurę scalania otrzymasz dodatkowy punkt! Warunkiem jest, żeby nie używała żadnych wysokopoziomowych funkcji scalających oraz, oczywiście, musi działać poprawnie dla różnych danych, nie tylko dla przykładowych. Ponieważ jest to zadanie dodatkowe, nie będę w nim pomagał i trzeba poradzić sobie samemu.
Do znalezienia iteratora na środkowy element możesz wykorzystać arytmetykę iteratorów (analogicznie jak dla wskaźników), operator $\pm int$ daje oterator przesunięty o int w prawo dla + i w lewo dla -. Odległość między dwoma iteratorami można znaleźć odejmując je od siebie i rzutując na typ int, albo z użyciem funkcji $std :: distance(first, second)$.
Szablon:

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


template <class RandomIter>
void mergeSort(RandomIter begin, RandomIter end);

int main() {
std::vector<int> v = {6, 4, 2, 2, 3, 1, 24, -12, 0, 3, 1};
mergeSort(v.begin(), v.end());
std::for_each(v.begin(), v.end(), [](auto x) { std::cout << x << " "; });
return 0;
}
```

Output: Posortowana rosnąco tablica.
Uwaga: Nie korzystać z gotowych implementacji sortowania przez scalanie. Trzymać się algorytmu i nie łączyć różnych metod sortowania. Nie tworzyć nowych tablic czy kolekcji – posortować istniejący wektor.

Zadanie 1 – sortowanie kopcowe (2pkt)

Napisz szablon funkcji implementujący sortowanie kopcowe (ang. heap sort)

```
template<typename I>
void heapSort(I begin, I end);
```

Algorytm jest następujący (dla sortowania malejącego):

1. Nadaj sortowanej tablicy strukturę kopca, tj. drzewa binarnego w którym każdy rodzic jest nie mniejszy od swoich dzieci.
2. Największym elementem w tablicy jest korzeń kopca. Weź go i umieść na końcu tablicy.
3. Przywróć strukturę kopca na wycinku tablicy nie zawierającym ostatniego elementu.
4. Powtarzaj procedurę zwiększając posortowaną część tablicy kosztem kopca.
5. Zakończ w momencie, gdy wszystkie elementy znajdą się w posortowanej części tablicy.

Będziesz potrzebować drugiego szablonu funkcji, którego zadaniem jest tworzenie struktury kopca w tablicy:

```
template<typename I>
void heapify(I begin, I end, std::size_t current);
```

Iteratory begin oraz end wskazują odpowiednio na początek i koniec tablicy. Zmienna current wskazuje na indeks rodzica (na raz rozważamy jedynie rodzica i jego nie więcej niż 2 dzieci). Uwaga: Funkcja powinna być rekurencyjna. Rozwiązania bez rekurencji będą mogły otrzymać max 70% dostępnych punktów. Należy pamiętać o poprawnym zakończeniu rekurencji.
Szablon:

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

template<typename I>
void heapify(I begin, I end, std::size_t current);

template<typename I>
void heapSort(I begin, I end);

int main()
{
        std::vector<int> v = {25, 1, 3, 13, 2, 8, 0, -2, 4,
         -2, 2, -3, -1, 1, 24};
        heapSort(v.begin(), v.end());
        std::for_each(v.begin(), v.end(), [](auto x)
         { std::cout << x << " "; });
        std::cout<<std::endl;
        return 0;

}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.
Podpowiedź: Pomocne mogą być funkcje do operacji na iteratorach, które poznaliśmy w trakcie poprzednich zajęć. Może się również przydać *std* :: $distance(Iit1, Iit2)$, która liczy odległość między dwoma elementami wskazywanymi przez iteratory it1 i it2.
Dla chętnych: Dodaj opcję, żeby przy wywołaniu szablonu można było zdecydować, czy sortowanie ma być rosnące czy malejące.

1 Zadanie 2 – lambdy (2pkt)

Celem tego zadania jest oswojenie się ze składnią funkcji lambda. W tym celu trzeba napisać 10 trywialnych funkcji, każda będzie warta 0.1+0.1 punktów (poprawność i terminowość).
Szbalon

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

using namespace std;

int main()
{
        vector<double> v =
        {12.54, -7.2, 9.09, 3.561, 0.0, -4.2135, 6.0009, 12.1, 2.45, -3.1};

        auto wypisz = [&v]()->void
        {
                for(auto em : v)
                        cout << em << " ";
                cout << endl;
        };

        auto sum = ...
        auto av = ...
        auto std = ...
        auto max = ...
        auto min = ...
        auto square = ...
        auto inverse = ...
        auto multKtimes = ...
        auto toInt = ...
        auto modify = ...

        wypisz();

        cout << setprecision(3) << "SUMA=" << sum()
        << " SREDNIA=" << av()
        << " ODCHYLENIE=" << std()
        << " MAX=" << max()
```

```
        << " MIN=" << min()
        << endl;

    square();
        wypisz();
        inverse();
        wypisz();
        multKtimes(1241.78);
        wypisz();
        toInt();
        wypisz();
        modify(-13, 50);
        wypisz();
        return 0;
}
```

Funkcje sum, av, std, max, min zwracają kolejno sumę, średnią, odchylenie standardowe, element największy, element najmniejszy z wektora v. **Te funkcje mają być napisane tak, żeby programistycznie nie były w stanie zmienić wektora v.**
Teraz funkcje, które **modyfikują** wektor v: Funkcja square zamienia wszystkie liczby w wektorze v na ich kwadraty. Funkcja inverse zamienia liczby na ich odwrotności, chyba, że napotka 0, wtedy zostawia 0 (a nie np. inf). Funkcja multKtimes mnoży wszystkie elementy wektora v przez liczbę podaną jako argument. Funkcja toInt pozbywa się części ułamkowej liczb z wektora (typ wektora się ostatecznie nie zmienia).
Funkcja modify przyjmuje dwa argumenty typu double, pierwszy określa nową liczbę, którą dodamy do wektora v, drugi ustala próg, po przekroczeniu którego mamy dodać nowy element. Działanie jest następujące: iterujemy po wszystkich elementach wektora v, jeżeli napotkamy element większy od drugiego argumentu to zmieniamy mu znak na przeciwny i dodajemy **za nim** nowy element wektora równy pierwszemu argumentowi (patrz przykładowy output). Efektem działania jest dodanie nowych elementów do wektora oraz zmiana znaku niektórych starych elementów. Aby dodać element VAL **przed** elementem wskazywanym przez iterator ITER, można użyć funkcji $v.insert(ITER, VAL)$.
Output:
12.54 -7.2 9.09 3.561 0 -4.2135 6.0009 12.1 2.45 -3.1
SUMA=31.2 SREDNIA=3.12 ODCHYLENIE=6.5 MAX=12.5 MIN=-7.2
157 51.8 82.6 12.7 0 17.8 36 146 6 9.61
0.00636 0.0193 0.0121 0.0789 0 0.0563 0.0278 0.00683 0.167 0.104
7.9 24 15 97.9 0 69.9 34.5 8.48 207 129
7 23 15 97 0 69 34 8 206 129
7 23 15 -97 -13 0 -69 -13 34 8 -206 -13 -129 -13

Uwaga: W tym zadaniu nie piszemy standardowych funkcji tylko funkcje lambda. W outpucie można zauważyć, że konwersja na liczbę całkowitą zmieniła 207 na 206. Nie ma tutaj błędu, bo tak na prawdę nie ma lcizby 207 tylko 206.X gdzie $X >= 5$. Z powodu ustawionej precyzji nie wyświetliły się cyfry po przecinku.
Podpowiedź: Nie trzeba tutaj pisać ogólnego kodu, w szczególności nie trzeba przekazywać wektora jako argumentu, tylko dać funkcji lambda dostęp do zmiennej v.

Zadanie 3 – przeciążanie operatorów (1pkt)

Klasa Wektor reprezentuje wektor w 3D i posiada trzy składowe prywatne odpowiadające współrzędnym w układzie kartezjańskim. Zaimplementuj dla tej klasy następujące operatory:

- + – dwuargumentowy operator dodawania wektorów
- - – dwuargumentowy operator odejmowania wektorów
- * – dwuargumentowy operator mnożenia skalarnego wektorów
- == – dwuargumentowy operator sprawdzający, czy wektory są identyczne
- - – jednoargumentowy operator odpowiadający mnożeniu wektora przez -1

Szablon:

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

class Wektor
{
private:
        double x;
        double y;
        double z;
public:
        void setX(double a){x=a;}
        void setY(double a){y=a;}
        void setZ(double a){z=a;}
        double getX(){return x;}
        double getY(){return y;}
        double getZ(){return z;}
        Wektor(const double a=0., const double b=0., const double c=0.)
        :x(a), y(b), z(c){}
        double getModule(){return sqrt(x*x+y*y+z*z);}
        friend std::ostream& operator<<(std::ostream &out, const Wektor& c);
};

std::ostream& operator<<(std::ostream &os, const Wektor& w)
{
        os << "(" << w.x << "," << w.y << "," << w.z << ")";
        return os;
};

int main()
{
        Wektor w(5., -1., 3.);
        Wektor v;
        v.setX(1.);
        v.setY(2.);
        v.setZ(-3.);
```

```
        std::cout << w << std::endl << v << std::endl;
        std::cout << -w << std::endl;
        std::cout << w+v << std::endl;
        std::cout << w-v << std::endl;
        std::cout << w*v << std::endl;
        std::cout << (w==v) << std::endl;
        std::cout << (Wektor(5., -1., 3.)==w) << std::endl;
        return 0;
}
```

Output:
(5,-1,3)
(1,2,-3)
(-5,1,-3)
(6,1,0)
(4,-3,6)
-6
0
1