Zadanie 0 – metoda Simpsona (1pkt)

Napisz funkcję:

```
double simpsonIntegration(double(*f)(double), double from, double to, int n);
```

która liczy całkę oznaczoną z funkcji *f* na przedziale od *from* do *to*. Pamiętaj o sprawdzeniu poprawności danych. Jeżeli dane są złe, możesz np. wyświetlić komunika to błędzie i zwrócić 0. Pamiętaj, że metoda Simpsona ma sens jedynie dla parzystych n. Nie zakładaj $from < to$.
Wykorzystaj poniższy szablon:

```
#include <cmath> //nie ma spacji przed >
#include <iostream>

double simpsonIntegration(double (*f)(double), double from, double to, int n);

double square(double x) { return x * x; };
double xexp(double x) { return x * exp(x); };
double poly(double x)
 {
        double sum = 0;
        for (int i = 1; i != 9; ++i)
        {
                sum += (i + 1) * pow(x, i);
        }
        return sum;
}

int main()
{
        std::cout << simpsonIntegration(square, -1, 1, 6) << std::endl;
        std::cout << simpsonIntegration(nullptr, -1, 1, 6) << std::endl;
        std::cout << simpsonIntegration(xexp, 0, 1, 6) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, 6) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, 33) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, -33) << std::endl;
        std::cout << simpsonIntegration(poly, 1, 0, 100) << std::endl;
        return 0;
}
```

Output:// 0.666667// [ERROR] Nullptr!// 0// 1.00003// 8.02462// [ERROR] Wrong value of n!// 0// [ERROR] Wrong value of n!// 0// -8 Uwaga: Nie tworzyć żadnych tablic ani kolekcji. Nie wykorzystywać żadnych gotowych algorytmów.
Podpowiedź: Zadanie jest proste, wystarczy zaimplementować odpowiedni wzór i sprawdzić, czy dane są poprawne.

Zadanie 1 – inne zastosowanie metody Simpsona (1pkt)

Skopiuj funkcję całkującą metodą Simpsona, którą napisałeś w poprzednim zadaniu.
Napisz funkcję:

```
double logarithm(double x, int n)
```

która wykorzysta napisaną w Zadaniu 0 funkcję simpsonIntegration(), żeby policzyć wartość logarytmu z liczby x przy n podziałach w metodzie Simpsona. Dla x mniejszego niż 0 zwróć *NAN*. *NAN* jest stałą używaną do określania liczb, które są niezdefiniowane albo nie można ich reprezentować. Zazwyczaj, jeżeli funkcja wykonująca działania numeryczne napotyka błędne argumenty, np. funkcja $1/x$ dla $x = 0$, to w zależności od intencji programisty robi jedną z kilku rzeczy:

1. Wyświetla komunikat o błędzie i przerywa działanie, najczęściej z wykorzystaniem wyjątków, które pozwalają wychwycić błąd i kontynuować pracę programu pomimo błędu.
2. Wyświetla komunikat o błędzie, ale nie przerywa działania, zwracając jakąś wartość domyślną (tak jak to zrobiliśmy w Zadaniu 0).
3. Nie wyświetla błędu ale zwraca NAN. Jeżeli wynik ma być wykorzystywany gdzieś dalej, to należy sprawdzić czy nie jest NAN za pomocą funkcji isnan().

Wykorzystaj poniższy przykład:

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

double simpsonIntegration(double (*f)(double), double from, double to, int n);

double logarithm(double x, int n);

int main()
{
        std::cout << "Ln(7)=" << logarithm(7, 16) << std::endl;
        std::cout << "Ln(e)=" << logarithm(2.718281828459, 10) << std::endl;
        std::cout << "Ln(0.5)=" << logarithm(0.5, 6) << std::endl;
        std::cout << "Ln(-1)=" << logarithm(-1, 6) << std::endl;
        std::cout << "Ln(0)=" << logarithm(0, 6) <<std::endl;
        auto val = isnan(logarithm(-2, 2)) ? "Yes!" : "No!";
        std::cout << "Is ln(-2) nan? " << val << std::endl;
        return 0;
}
```

Output:
Ln(7)=1.94642
Ln(e)=1.00003
Ln(0.5)=-0.69317
Ln(-1)=nan
Ln(0)=-inf
Is ln(-2) nan? Yes!

Uwaga: Nie tworzyć tablic ani kolekcji. Nie modyfikować funkcji całkującej.
Podpowiedź: Cała trudność polega na wymyśleniu jak policzyć logarytm wykorzystując całkowanie. Przyda się napisanie jeszcze jednej prostej funkcji (albo użycie funkcji lambda jeśli ktoś zna).

Zadanie 2 – Pochodna w punkcie

Napisz funkcje

```
double forwardDiff(double(*f)(double), double x, double h);
double backwardDiff(double(*f)(double), double x, double h);
double centralDiff(double(*f)(double), double x, double h);
double richardsonDiff(double(*f)(double), double x, double h);
double centralSecondDiff(double(*f)(double), double x, double h);
```

Obliczające pochodne funkcji f w punkcie x z wykorzystaniem różnic odmiennych rodzajów. Wykorzystaj poniższe wzory:

$$f_f'(x) = \frac{f(x+h) - f(x)}{h} \tag{1}$$

$$f_b'(x) = \frac{f(x) - f(x-h)}{h} \tag{2}$$

$$f_c'(x) = \frac{f(x+h) - f(x-h)}{2h} \tag{3}$$

$$f_r'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \tag{4}$$

$$f_c''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{5}$$

Porównaj dokładność metod liczenia pierwszej pochodnej.
Wykorzystaj poniższy szablon:

```
#include <cmath> //bez spacji przed >
#include <iostream>

double forwardDiff(double (*f)(double), double x, double h);
double backwardDiff(double (*f)(double), double x, double h);
double centralDiff(double (*f)(double), double x, double h);
double richardsonDiff(double (*f)(double), double x, double h);
double centralSecondDiff(double (*f)(double), double x, double h);

double poly(double x)
{
        double sum = 0;
        for (int i = 1; i != 9; ++i)
        {
                sum += (i + 1) * pow(x, i);
        }
        return sum;
}

int main()
{
        auto h = {0.01, 0.00001};
        auto v = {poly, cos};
        for (auto hi : h)
        {
                std::cout << "h=" << hi << std::endl;
                for (auto f : v)
                {
                        std::cout << forwardDiff(f, 0, hi) << std::endl;
```

```
                        std::cout << backwardDiff(f, 0, hi) << std::endl;
                        std::cout << centralDiff(f, 0, hi) << std::endl;
                        std::cout << richardsonDiff(f, 0, hi) << std::endl;
                        std::cout << centralSecondDiff(f, 0, hi) << std::endl;
                        std::cout << std::endl;
                }
        }
        return 0;
}
```

Output:
h=0.01
2.03041
1.9704
2.0004
2
6.001
-0.00499996

0.00499996
0
-1.85037e-15
-0.999992
h=1e-05

2.00003
1.99997
2
2
6
-5e-06

5e-06
0
0
-1

Zadanie 3 – różniczkowanie funkcji (1pkt)

Uzupełnij funkcje

```
std::vector<double>* centralDiff(const std::vector<double>* x,
const std::vector<double>* y);

std::vector<double>* secondCentralDiff(const std::vector<double>* x,
const std::vector<double>* y);
```

Które przyjmują wskaźniki na wektory x oraz y niosące informacje o wartościach funkcji w kolejnych punktach. Funkcje tworzą nowy wektor liczb double na stosie (z użyciem operatora new), wypełniają go odpowiednio pierwszą lub drugą pochodną (różnicą) centralną i zwracają go poprzez wskaźnik. Użyj poniższych wzorów:

$$f' = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{6}$$

$$f'' = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(x_{i+1} - x_i)(x_i - x_{i-1})} \tag{7}$$

Zastanów się jaki rozmiar powinien mieć zwracany wektor. Możesz założyć, że przekazywane punkty są już posortowane po współrzędnej x.

Szablon programu:

```
#include <iostream> //bez spacji przed >
#include <vector>

std::vector<double>* centralDiff(const std::vector<double>* x,
const std::vector<double>* y);

std::vector<double>* secondCentralDiff(const std::vector<double>* x,
const std::vector<double>* y);

int main()
{
        std::vector<double> x = {1, 1.101, 1.202, 1.303, 1.404,
        1.505, 1.606, 1.707, 1.808, 1.909};

        std::vector<double> y = {1.975, 1.85955, 1.72085, 1.55678,
         1.36518, 1.14394, 0.890923, 0.603991, 0.281013, -0.0801417};

        std::cout << "first:" << std::endl;
        std::vector<double>* v = centralDiff(&x, &y);
        for (auto i : (*v) )
        {
                std::cout << i << " ";
        }
        std::cout << std::endl;
        delete v;

        v = secondCentralDiff(&x, &y);
        std::cout << "second:" << std::endl;
        for (auto i : *v)
        {
                std::cout << i << " ";
        }
        std::cout << std::endl;
        delete v;

        return 0;
}
```

Output
first:

-1.25817 -1.49886 -1.76074 -2.04376 -2.34781 -2.67301 -3.01936 -3.3868
second:
-2.27919 -2.48701 -2.69876 -2.9056 -3.11509 -3.32467 -3.53358 -3.74245

Uwaga: Nie kopiować wektorów x i y. Nie używać gotowych implementacji algorytmów różniczkujących.
Podpowiedź: Alokowanie pamięci na stosie odbywa się poprzez operator *new*. Aby dodać element o nazwie *el* do obiektu *std* :: *vector* wskazywanego przez wskaźnik o nazwie v, należy użyć *v->push_back(el);*.

Zadanie 4 – sortowanie przez scalanie (1pkt)

Sortowanie przez scalanie to bardzo pomysłowy sortujący algorytm rekurencyjny, którego autorstwo przypisuje się Johnowi von Neumannowi. Algorytm wygląda następująco:

1. Dana tablica n nieposortowanych liczb.
2. Jeżeli $n == 1$ to tablica jest posortowana, koniec.
3. Jeżeli $n > 1$ to podziel tablice na dwie możliwie równe połowy.
   (a) Wywołaj rekurencyjnie algorytm dla każdej połowy z osobna.
   (b) Scal obydwie połowy, które są już posortowane. Zrób to, patrząc na kolejne elementy w obu tablicach i porównując je ze sobą.

Np. dla tablicy [23, 5, 1], którą chcemy posortować rosnąco, najpierw podzielimy ją na dwie tablice: [23, 5], [1]. Tablicę pierwszą ponownie podzielimy,([23], [5]), [1], a tablica 2 już była jednoelementowa. Teraz scalamy dostając [5, 23], [1]. Ponownie scalamy: patrzymy na pierwsze elementy w obu tablicach, mamy $1 < 5$ więc bierzemy jedynkę. W tym momencie wyczerpaliśmy wszystkie liczby z tablicy drugiej, więc teraz przepisujemy liczby z tablicy pierwszej które pozostały, w takiej kolejności w jakiej są (bo są posortowane). Czyli dostajemy [1, 5, 23].
W zadaniu proszę zaimplementować sortowanie rosnące dla tablicy liczb całkowitych typu int, gdzie posługujemy się wskaźnikami na początek i koniec tablicy. Można założyć, że kolejność wskaźników jest właściwa.
Uzupełnij poniższe funkcje

```
void splitMerge(int* a, int* b)
```

Wskaźniki a i b wskazują odpowiednio na poczatek i koniec sortowanej tablicy intów. W tej funkcji należy zaimplementować rekurencyjną część algorytmu, czyli dzielenie tablicy na dwie części, wołanie splitMerge() dla każdej części z osobna i na koniec obu części. Trzeba pamiętać o warunku na koniec rekurencji!

```
void mergeSort(int* a, int* b, int* c)
```

Ta funkcja ma za zadanie scalić dwie tablice w taki sposób, żeby po scaleniu wynikowa tablica była posortowana. Wskaźnik a wskazuje na początek pierwszej tablicy, wskaźnik c na koniec drugiej tablicy. Wskaźnik b może wskazywać na koniec pierwszej tablicy, albo jesli wolisz, na początek drugiej tablicy. Ponieważ tablice, które będziemy scalać są ułożone jedna po drugiej, to nie ma potrzeby przekazywać 4 wskaźników, za pomocą wskaźnika b można dostać zarówno koniec 1. tablicy jak i początek 2. tablicy.
Przy scalaniu bierz kolejne elementy z już posortowanych tablic. Rób to pojedynczo, wybierając mniejszy z nich. Pamiętaj, że tablice mogą być różnych rozmiarów. Nieprawdziwe jest założenie, że kolejne elementy będą wybierane naprzemiennie, raz z 1. a potem z 2. tablicy, a potem znów z 1. itd. Potrzebna będzie arytmetyka wskaźników.
Końcowym efektem działania funkcji mergeSort() ma być zmiana wartości w oryginalnej tablicy!

Wykorzystaj poniższy szablon:

```cpp
#include <iostream> //bez spacji przed >

void mergeSort(int* a, int* b, int* c)
{
        //uzupelnij
}

void splitMerge(int* a, int* b)
{
        //uzupelnij
}

int main()
{
        int tab[7] = {23, 5, -1, 2, 3, 4, -1 };
        int arr[10] = {-4, -6, 0, 1, -2, 8, -11, 2, 0, 1};
        splitMerge(tab, tab+6);
        std::cout<<"[ ";
        for(auto val : tab)
                std::cout << val << " ";
        std::cout << "]" << std::endl << "[ ";
        splitMerge(arr+9, arr);
        for(auto val : arr)
                std::cout << val << " ";
        std::cout << "]" << std::endl;
}
```

Output:
[**-1 -1 2 3 4 5 23**]
[**-11 -6 -4 -2 0 0 1 1 2 8**]

Uwaga: Algorytm jest rekurencyjny i tak ma być zaimplementowany. Efektem działania ma być zmiana oryginalnej tablicy. Proszę posortować ją rosnąco.
Podpowiedź: W funkcji mergeSort() przydatne może być utworzenie pomocniczej tablicy do której wpiszesz posortowane wartości. Wtedy na koniec w pętli for można wykorzystać arytmetykę wskaźników, żeby przepisać posortowane wartości do oryginalnej tablicy. Stworzoną na stosie tablicę należy na koniec bezwzględnie usunąć operatorem *delete*[]. W trakcie wypełniania tablicy przydać się mogą nowe wskaźniki, do których skopiujemy oryginalne, a następnie będziemy je zwiększać/zmniejszać.
Dla chętnych: Przerób to tak, żeby działało dla tablic dowolnego typu dla których zdefiniowano operatory $>, <, ==, >=, <=$ Użyj szablonów funkcji.