```cpp
#include <iostream>
#include <ctime>
#include <chrono>
#include <stdlib.h>
#include <fstream>
#include <cstdlib>
#include <algorithm>

//klasa input - do ujednolicenia wykonania jednakowych funkcjonalnosci
//              wystepujacych w roznych czesciach programu. Klasa ta jest
//              dziedziczona przez wszystkie pozostale klasy
class Input{
public:
    //funkcja do wprowadzania liczby z oczekiwanego przedzialu
    int inputLoop(int minimum, int maksimum){
        int x;
        // powtarzanie petli jesli x nie poza podanym zakresem lub nie jest liczba calkowita int
        while(!(std::cin >> x) || x < minimum || x > maksimum){
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            printf("Nalezy podac cyfre z zakresu (%d) - (%d):\n",minimum,maksimum);
        }
        return x;
    }

    //funkcja do zwrocenia liczby losowej z oczekiwanego przedzialu
    double giveRandom(int min_v, int max_v){
        double random_num = (rand()/(double)RAND_MAX)*(max_v - min_v)+min_v;
        return random_num;
    }
};

//klasa sortings - klasa ze wszystkimi uwzglednionymi sortowaniami
//                 sortowanie shella z 2 funkcjami po 1 dla kazdego rodzaju
//                 podobnie quick sort: 4 funkcje po 1 dla kazdego rodzaju wyboru pivota
template <typename T>
class Sortings: public Input{
public:
    int chosen_pivot;   //zmienna okreslajaca wybrany sposob doboru pivota w quicksort
    int ShellOption;    //zmienna okreslajaca wybrane odstepy w algorytmie shella

// poczatek sekcji z sortowaniem przez wstawianie
    void insertSort(T tab[], int n){
        int key;
        for(int i=1;i<n;i++){
            key=tab[i];
            int j=i;
            while(j>0 && tab[j-1]>key){
                tab[j]=tab[j-1];
                j--;
            }
            tab[j] = key;
        }
    }

    double prepareInsertSort(T tab[], int n){
        auto start = std::chrono::high_resolution_clock::now();
        insertSort(tab, n);
        auto ending = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
    }

// poczatek sekcji z sortowaniem przez kocpowanie
    void heapFix(T t[], int n, int index){
        int parent = index;
        int left_child = 2 * index + 1;
```

```
        int right_child = 2 * index + 2;
        if (left_child < n && t[left_child] > t[parent]){
            parent = left_child;
        }
        if (right_child < n && t[right_child] > t[parent]){
            parent = right_child;
        }
        if (parent != index) {
            std::swap(t[index], t[parent]);
            heapFix(t, n, parent);
        }
    }

    void heapSort(T tab[], int n){
        //wstepne zbudowanie kopca
        for (int i = n/2 - 1; i >= 0; i--){
            heapFix(tab, n, i);
        }
        //wlasciwa czesc sortowania
        for(int i=n-1;i>0; i--) {
            std::swap(tab[0],tab[i]);
            heapFix(tab,i,0);
        }
    }

    double prepareHeapSort(T tab[], int n){
        auto start = std::chrono::high_resolution_clock::now();
        heapSort(tab, n);
        auto ending = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
    }
// poczatek sekcji z sortowaniami shella
    // z odstepami shella N/2, N/4 itd.
    void ShellSort1(T tab[], int n){
        for (int gap = n/2; gap > 0; gap /= 2) {
            for (int i = gap; i < n; i += 1) {
                int temp = tab[i];
                int j;
                for (j = i; j >= gap && tab[j-gap] > temp; j -= gap){
                    tab[j] = tab[j-gap];
                }
                tab[j] = temp;
            }
        }
    }
    // z odstepami Hibbarda (2^k - 1)
    void ShellSort2(T tab[], int n){
        int interval = 2;                   // wyznaczenie maksymalnej potegi liczby 2
        while(interval < n/2){interval*=2;} // ale mniejszej niz n
        interval--;                         // uzyskanie 2^k - 1
        for (int gap = interval; gap > 0; gap /= 2) {
            for (int i = gap; i < n; i += 1) {
                int temp = tab[i];
                int j;
                for (j = i; j >= gap && tab[j-gap] > temp; j -= gap){
                    tab[j] = tab[j-gap];
                }
                tab[j] = temp;
            }
        }
    }

    void shellSortInfo(int n){
        std::cout << "Wybierz sortowanie shella:\n(1) Odstepy Shella N/2, N/4 itd.\n(2) Odstepy Hibbarda
(2^-1)\n";
        ShellOption = inputLoop(1,2);
```

```
    }
    double prepareShellSort(T tab[], int n){
        auto start = std::chrono::high_resolution_clock::now();
        switch(ShellOption){
            case 1: ShellSort1(tab, n); break;
            case 2: ShellSort2(tab, n); break;
            default: break;
        }
        auto ending = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
    }

// poczatek sekcji z sortowaniami szybkimi
    // z lewym skrajnym pivotem
    int partitionedL(T tab[], int left, int right){
        T pivot = tab[left];
        int l=left; int r = right;
        while(true){
            while(tab[l]<pivot){l++;}
            while(tab[r]>pivot){r--;}
            if(l < r){
                std::swap(tab[l],tab[r]);
                l++;
                r--;
            }
            else{
                if (r == right) {r--;}
                return r;
            }
        }
    }
    void quickSortL(T tab[], int l, int r){
        if(l>=r)    {return;}
        int m = partitionedL(tab, l, r);
        quickSortL(tab, l, m);
        quickSortL(tab, m+1, r);
    }

    // z prawym skrajnym pivotem
    int partitionedR(T tab[], int left, int right){
        T pivot = tab[right];
        int l=left; int r = right;
        while(true){
            while(tab[l]<pivot){l++;}
            while(tab[r]>pivot){r--;}
            if(l < r){
                std::swap(tab[l],tab[r]);
                l++;
                r--;
            }
            else{
                if (r == right) {r--;}
                return r;
            }
        }
    }
    void quickSortR(T tab[], int l, int r){
        if(l>=r)    {return;}
        int m = partitionedR(tab, l, r);
        quickSortR(tab, l, m);
        quickSortR(tab, m+1, r);
    }

    // ze srodkowym pivotem
    int partitionedM(T tab[], int left, int right){
        T pivot = tab[(left+right)/2];
```

```
    int l=left; int r = right;
    while(true){
        while(tab[l]<pivot){l++;}
        while(tab[r]>pivot){r--;}
        if(l < r){
            std::swap(tab[l],tab[r]);
            l++;
            r--;
        }
        else{
            if (r == right) {r--;}
            return r;
        }
    }
}
void quickSortM(T tab[], int l, int r){
    if(l>=r)    {return;}
    int m = partitionedM(tab, l, r);
    quickSortM(tab, l, m);
    quickSortM(tab, m+1, r);
}

// z losowym pivotem
int partitionedRand(T tab[], int left, int right){
    T pivot = tab[int(giveRandom(left,right))];
    int l=left; int r = right;
    while(true){
        while(tab[l]<pivot){l++;}
        while(tab[r]>pivot){r--;}
        if(l < r){
            std::swap(tab[l],tab[r]);
            l++;
            r--;
        }
        else{
            if (r == right) {r--;}
            return r;
        }
    }
}
void quickSortRand(T tab[], int l, int r){
    if(l>=r)    {return;}
    int m = partitionedRand(tab, l, r);
    quickSortRand(tab, l, m);
    quickSortRand(tab, m+1, r);
}

// pomiar czasu sortowania z pivotem skrajnym lewym
double quickSort1(T tab[], int l, int r){
    auto start = std::chrono::high_resolution_clock::now();
    quickSortL(tab, l, r);
    auto ending = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
}

// pomiar czasu sortowania z pivotem skrajnym prawym
double quickSort2(T tab[], int l, int r){
    auto start = std::chrono::high_resolution_clock::now();
    quickSortR(tab, l, r);
    auto ending = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
}

// pomiar czasu sortowania z pivotem srodkowym
double quickSort3(T tab[], int l, int r){
    auto start = std::chrono::high_resolution_clock::now();
```

```
        quickSortM(tab, l, r);
        auto ending = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
    }

    // pomiar czasu sortowania z pivotem losowym
    double quickSort4(T tab[], int l, int r){
        auto start = std::chrono::high_resolution_clock::now();
        quickSortRand(tab, l, r);
        auto ending = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::nanoseconds>(ending-start).count();
    }

    // wybor pivota w sortowaniu szybkim
    void quickSortInfo(){
        std::cout << "Wybierz pozycje pivota:\n(1) Skrajna lewa\n(2) Skrajna prawa\n(3) Srodkowa\n(4) Losowa\n";
        chosen_pivot = inputLoop(1,4);
    }
    // wywolanie danej funkcji zalezne od wyboru z powyzszej funkcji
    double prepareQuickSort(T tab[], int l, int r){
        int time_m = 0;
        switch(chosen_pivot){
            case 1: time_m = quickSort1(tab, l, r); break;
            case 2: time_m = quickSort2(tab, l, r); break;
            case 3: time_m = quickSort3(tab, l, r); break;
            case 4: time_m = quickSort4(tab, l, r); break;
            default: break;
        }
        return time_m;
    }
};

//klasa operations - warstwa druga menu do wyboru rodzaju operacji z wybranym typem danych
template <typename H>
class Operations: public Input{
public:
    Sortings<H> sorting;
    H* arrayToSort = nullptr;    // oryginal generowanej/wczytanej tablicy
    H* sortedArray = nullptr;    // kopia oryginalnej tablicy, na ktorej sa wykonywane algorytmy sortowania
    H* arrayToCompare = nullptr;// kopia oryginalnej tablicy posortowana z uzyciem std::sort w celu
                                // porownania jej z sortedArray (sprawdzenie poprawnosci
                                // wykonania uzytych algorytmow sortowania)

    int array_size;              // rozmiar tablicy do generowania
    std::string text_file;       // nazwa pliku do wczytania
    int min_value, max_value;    // zakres generowania danych do tablicy
    int option;                  // zmienna do okreslenia, ktora opcje z drugiej warstwy menu wykonac
    int algorithm;               // zmienna do okreslenia, ktory algorytm zostal wybrany
    int sorted_ratio;            // zmienna do okreslenia jaki typ wstepnego posortowania tablicy zostal wybrany
    double time_measured;

    bool is_initialized = false;// czy oryginal tablicy zostal zainicjowany
    bool isSorted = false;       // czy zostalo wykonane sortowanie na ostatnio utworzonej tablicy

    //utworzenie tablicy o zadanym rozmiarze - jednakowo dla kazdej tablicy
    //parametr arr przekazywany jest przez referencje
    void createArray(H *&arr, int array_size){
        if (arr != nullptr) {
            delete[] arr;
        }
        arr = new H[array_size];
        this->array_size = array_size;
        is_initialized = true;  // potwierdzenie utworzenia tablicy
    }

    bool optionMenu(){
```

```
        std::cout << "Wybierz polecenie:\n(1) Wczytanie tablicy z pliku o zadanej nazwie\n(2) Wygenerowanie
tablicy o zadanym rozmiarze\n(3) Wyswietlenie ostatnio utworzonej tablicy\n(4) Uruchomienie wybranego algorytmu
sortowania\n(5) Wyswietlenie posortowanej tablicy\n(6) Sprawdzenie poprawnosci algorytmu\n(7) Wykonanie serii
badan\n\n(0) Wyjscie z programu\n";
        option = inputLoop(0, 7);
        if (option == 0)   {return false;}
        return true;
    }
    void options(){
        while(true){
            time_measured = 0;
            // instrukcja switch do wykonania okreslonego polecenia w zaleznosci od wyboru z optionMenu
            if(!optionMenu()) {break;}
            switch(option){
                case 1: if(!loadFile()){return;} break;
                case 2: generateArrayInfo(); break;
                case 3: showLastArray(); break;
                case 4: chooseAlgorithmAndSortInfo();
                        printf("Sortowanie trwalo %f milisekund\n\n" ,time_measured/1000000.);
                        break;
                case 5: showSortedArray(); break;
                case 6: checkSortedArray(); break;
                case 7: serie(); break;
                default: break;
            }
            system("PAUSE");
            system("CLS");
        }
    }

    //funkcja do wczytania danych z pliku tekstowego
    bool loadFile(){
        std::cout << "\nPodaj nazwe pliku tekstowego (razem z '.txt'):\n";
        std::cin >> text_file;
        std::ifstream ifile;
        ifile.open(text_file);
        // petla do obsluzenia przypadku nie odnalezienia pliku o zadanej nazwie
        while(!ifile){
            std::cout << "\nNie mozna odnalezc pliku. Upewnij sie ze podana nazwa jest wlasciwa i sprobuj
ponownie lub:\n(0) Jesli chcesz wyjsc z programu\n(-1) Jesli chcesz sie cofnac do menu\n";
            std::cin >> text_file;
            ifile.open(text_file);
            if(text_file == "-1")       {system("CLS"); return true;}
            else if(text_file == "0")   {return false;}
        }
        std::cout << "\nPlik zostal pomyslne odnaleziony\n";

        try{
            std::string content;
            getline(ifile, content);
            //stworzenie tablicy o rozmiarze rownym wartosci pierwszego elementu w pliku tekstowym
            createArray(arrayToSort, std::stoi(content));
            int index = 0;
            //petla do wypelnienia tablicy danymi z pliku
            while(getline(ifile, content)){
                arrayToSort[index] = std::stod(content);
                index++;
            }
            std::cout << "\n\n";
        }
        catch(std::invalid_argument e){
            std::cout << "Przerwanie odczytu! W pliku znajduje sie niewlasciwy argument!\n"; return false;
        }
        is_initialized = true;  //potwierdzenie zainicjowania tablicy
        isSorted = false;
        return true;
```

```
    }

    // funkcja do zebrania odpowiednich informacji o tworzonej tablicy
    void generateArrayInfo(){
        std::cout << "\nWybierz rozmiar tablicy:\n";
        array_size = inputLoop(1, 2147000000);
        std::cout << "\nPodaj minimalna wartosc w tablicy:\n";
        min_value = inputLoop(-2147000000, 2147000000);
        std::cout << "\nPodaj maksymalna wartosc w tablicy:\n";
        max_value = inputLoop(min_value, 2147000000);
        generateArray(array_size, min_value, max_value);
    }

    // funkcja do utworzenia i wpelnienia tablicy
    void generateArray(int array_size, int min_value, int max_value){
        createArray(arrayToSort, array_size);
        H rand_number;
        for (int i = 0; i < array_size; i++){
            rand_number = giveRandom(min_value, max_value);
            arrayToSort[i] = rand_number;
        }
        is_initialized = true;
        isSorted = false;
    }

    // funkcja do wyswietlenia ostanio utworzonej tablicy
    void showLastArray(){
        if(!is_initialized) {std::cout << "Operacja niemozliwa. Tablica nie zostala jeszcze zainicjowana!\n";
return;}
        std::cout << "\nZawartosc ostaniej utworzonej tablicy:\n";
        for(int i = 0; i < array_size; i++){
            std::cout << arrayToSort[i] << "\n";
        }
    }

    // funkcja do wyboru algorytmu i typu wstepnego posortowania
    void chooseAlgorithmAndSortInfo(){
        if(!is_initialized) {std::cout << "Operacja niemozliwa. Tablica nie zostala jeszcze zainicjowana!\n";
return;}
        system("CLS");
        std::cout << "Wybierz algorytm sortowania:\n(1) przez wstawianie\n(2) przez kopcowanie\n(3) Shella\n(4)
szybkie\n";
        algorithm = inputLoop(1,4);
        system("CLS");
        std::cout << "Wybierz procent posortowania tablicy przed pomiarem czase:\n(1) Pozostaw losowa\n(2) 100%
(Posortowana rosnaco)\n(3) 66%\n(4) 33%\n(5) 0% (Posortowana malejaco)\n";
        sorted_ratio = inputLoop(1,5);
        system("CLS");
        chooseAlgorithm(algorithm, sorted_ratio, true);
    }
    // funkcja do wykonania sortowania z uwzglednieniem informacji z powyzszej funkcji
    void chooseAlgorithm(int algorithm, int sorted_ratio, bool firstRepeat){
        createArray(sortedArray, array_size);
        std::copy(arrayToSort, arrayToSort+array_size, sortedArray);
        // sposoby wstepnego posortowania tablicy w zaleznosci od wyboru
        switch(sorted_ratio){
            case 2: std::sort(sortedArray, sortedArray + array_size); break;
            case 3: std::sort(sortedArray, sortedArray + int(array_size*0.66)); break;
            case 4: std::sort(sortedArray, sortedArray + int(array_size*0.33)); break;
            case 5: std::sort(sortedArray, sortedArray + array_size, std::greater<H>()); break;
            default: break;
        }
        // algorytmy do wykonania w zalznosci od dokonanego wyboru
        switch(algorithm){
            case 1: time_measured = sorting.prepareInsertSort(sortedArray, array_size); break;
            case 2: time_measured = sorting.prepareHeapSort(sortedArray, array_size); break;
```

```
            case 3: if(firstRepeat){sorting.shellSortInfo(array_size);} // odroznienie pierwszego wywolania od
pozostalych (w celu wykonania serii pomiarowej w funkcji serie())
                    time_measured = sorting.prepareShellSort(sortedArray, array_size); break;
            case 4: if(firstRepeat){sorting.quickSortInfo();} // odroznienie pierwszego wywolania od pozostalych
(w celu wykonania serii pomiarowej w funkcji serie())
                    time_measured = sorting.prepareQuickSort(sortedArray, 0, array_size-1); break;
            default: break;
        }
        isSorted = true;
    }

    // funkcja do wyswietlenia posortowanej tablicy
    void showSortedArray(){
        if(!isSorted) {std::cout << "Operacja niemozliwa. Ostatnio utworzona tablica nie zostala jeszcze
posortowana!\n"; return;}
        std::cout << "\nZawartosc ostaniej posortowanej tablicy:\n";
        for(int i = 0; i < array_size; i++){
            std::cout << sortedArray[i] << "\n";
        }
    }

    // funkcja do sprawdzenia poprawnosci wykonanego sortowania
    void checkSortedArray(){
        if(!isSorted) {std::cout << "Operacja niemozliwa. Ostatnio utworzona tablica nie zostala jeszcze
posortowana!\n"; return;}
        // stworzenie kopii oryginalu i posotowanie jej z uzyciem std:sort()
        createArray(arrayToCompare, array_size);
        std::copy(arrayToSort, arrayToSort+array_size, arrayToCompare);
        std::sort(arrayToCompare, arrayToCompare + array_size);

        // sprawdzenie na przykladzie uworzonej wyzej tablicy
        // poprawnosc posortowania sortowanej algorytmem tablicy
        for(int i=0; i < array_size;i++){
            if(arrayToCompare[i] != sortedArray[i]){
                std::cout << "Wykryto blad na pozycji " << i << "\n"; return;
            }
        }
        std::cout << "Nie wykryto bledu w sortowaniu\n";
    }

    // funkcja do wykonywania serii pomiarowych
    void serie(){
        system("CLS");
        std::cout << "Badania zostana przeprowadzone dla 7 tablic o reprezentatywnych rozmiarach,\ndobranych
eksperymentalnie, roznych dla poszczegolnych algorytmow\nprzy czym wartosc kazdego elementu zawiera sie w
przedziale [1, 2000000]\n\n";
        int default_array_size[7];
        int default_min_value = 1;
        int default_max_value = 2000000;

        // dobrane rozmiary tablic dla kazdego algorytmu sortowania
        int insert_array[] = {8000, 12000, 18000, 27000, 40000, 60000, 100000};
        int heap_array[] = {10000, 20000, 50000, 100000, 200000, 500000, 1000000};
        int shell_array[] = {30000, 50000, 100000, 200000, 500000, 1000000, 2000000};
        int quick_array[] = {10000, 20000, 40000, 80000, 160000, 320000, 640000};

        std::cout << "Wybierz algorytm sortowania:\n(1) przez wstawianie\n(2) przez kopcowanie\n(3) Shella\n(4)
szybkie\n";
        algorithm = inputLoop(1,4);
        //instrukcja switch do skopiowania odpowiedniego zestawu rozmiarow tablic do default_array_size[]
        switch(algorithm){
            case 1: std::copy(std::begin(insert_array), std::end(insert_array), std::begin(default_array_size));
                    break;
            case 2: std::copy(std::begin(heap_array), std::end(heap_array), std::begin(default_array_size));
                    break;
            case 3: std::copy(std::begin(shell_array), std::end(shell_array), std::begin(default_array_size));
```

```
                break;
            case 4: std::copy(std::begin(quick_array), std::end(quick_array), std::begin(default_array_size));
                break;
            default: std::cout << "Blad inicjalizacji tablicy!\n"; return;
        }
        std::cout << "Wybierz procent posortowania tablicy przed pomiarem czas:\n(1) Pozostaw losowa\n(2) 100%
(Posortowana rosnaco)\n(3) 66%\n(4) 33%\n(5) 0% (Posortowana malejaco)\n(6) Sprawdzenie wszystkich
powyzszych\n\n";
        sorted_ratio = inputLoop(1,6);
        std::cout << "Wybierz liczbe powtorzen sortowania dla kazdej tablicy: \n";
        int repeats = inputLoop(1,1000); // liczba powtorzen sortowania do wykonania
        system("CLS");

        // otwarcie pliku wyjsciowego do zapisania danych
        std::ofstream outFile("output.txt");
        if (!outFile.is_open()) {
            std::cout << "Blad otwarcia pliku na zapisywanie danych!\n";
            return;
        }
        outFile << "Measured time in nanoseconds:\n";
        double average_time;    // sredni czas wykonania algorytmu sortowania
        int def_array_length = sizeof(default_array_size) / sizeof(default_array_size[0]);
        bool first_repeat = true;

        // liczba prawidlowo zmierzonych czasow. W przypadku bardzo krokiego czasu
        // sortowania (dla Shella o rozmiarze 30000 elementow lub sortowania
        // przez wstawianie na posortowanej rosnaco wstepnie tablicy) zdarzaly sie
        // czasami wyniki rowne 0, zanizajace sredni czas sortowania.
        int properly_measured;
        // liczba tpow sortowan do wykonania. 1 jesli zostal wybrany konkretny typ
        // lub 5 jesli zostala wybrana ostatnia 6-ta opcja
        int sorts_num;
        if (sorted_ratio == 6) {sorts_num = 5;}
        else {sorts_num = 1;}

        // zewnetrzna petla for do sprawdzenia jednego konkretnego typu lub 5 wszystkich typow
        // wstepnie posortowanej tablicy
        for(int k = 1; k <= sorts_num; k++){
            // petla z kolejnymi rozmiarami tablicy dla danego algorytmu sortowania
            for(int j = 0; j < def_array_length;j++){
                average_time = 0;
                properly_measured = 0;
                // petla z liczba powtorzen dla kazdego rozmiaru tablicy
                for(int i = 0;i < repeats; i++){
                    generateArray(default_array_size[j], default_min_value, default_max_value);
                    // sortowanie w przypadku wyboru konkretnego typu wstepnie posortowanej tablicy
                    if(sorts_num == 1)  {chooseAlgorithm(algorithm, sorted_ratio,first_repeat);}
                    // sortowanie w przypadku wyboru zbadania wszystkich typow jednoczesnie
                    else                {chooseAlgorithm(algorithm, k,first_repeat);}
                    first_repeat = false; // falsz dla kazdego powtorzenia od drugiego poczawszy
                    average_time+= time_measured;
                    if(time_measured != 0) {properly_measured++;}
                    if(i % 5 == 0){
                        printf("Sortowanie %d - Tablica %7d: %d/%d ukonczenia\n",k,default_array_size[j], i,
repeats);
                    }
                }
                std::cout << properly_measured << " prob zostalo wlasciwie zmierzonych\n\n";
                average_time/= double(properly_measured); //nanoseconds
                outFile << average_time << "\n";
            }
            outFile << "\n";
        }
        std::cout << "Ukonczono dzialanie.\n";
    }
```

```
    //destruktor klasy options - usuwanie tablic gdy program zostaje zakonczony
    ~Operations(){
        if(arrayToSort != nullptr){
            delete[] arrayToSort;
        }
        if(sortedArray != nullptr){
            delete[] sortedArray;
        }
        if(arrayToCompare != nullptr){
            delete[] arrayToCompare;
        }
    }
};

//klasa menu - warstwa pierwsza menu programu do wyboru typu danych
class menu : public Input{
private:
    void chooseDataType(){
        std::cout << "#########################################################\n           AiZO - zadanie
projektowe nr. 1\n\n";
        std::cout << "      Badanie efektywnosci wybranych algorytmow\n   sortowania ze wzgledu na zlozonosc
obliczeniowa\n";
        std::cout << "#########################################################\n";
        std::cout << "\nStworzony program pozwala na wykonanie roznych sposobow sortownia\nna ponizej
pzedstawionych typach danych. W celu wybrania\ndanego typu danych nalezy podac odpowiednia cyfre:\n";
        std::cout << "\n(1) Int\n(2) Float\n(3) Double\n\n(0) Wyjscie z programu\n";
        data_type = inputLoop(0, 3);
        if (data_type == 0)  {return;}
        system("CLS");
    }
public:
    Operations<int> ints;
    Operations<float> floats;
    Operations<double> doubles;
    int data_type = 0;

    void init_programme(){
        chooseDataType();
        switch(data_type){
            case 1: ints.options(); break;
            case 2: floats.options(); break;
            case 3: doubles.options(); break;
            default: break;
        }
    }
};

//funkcja main - miejsce rozpoczecia programu
int main()
{
    menu klasa;
    srand(time(NULL));
    klasa.init_programme();
    system("PAUSE");
    return 0;
}
```