1. Sortowanie
   - bąbelkowe (bubble sort),
   - sortowanie przez wstawianie (insertion sort)

Sortowanie znajduje zastosowanie w problemach wymagających uporządkowania danych, np. uporządkowania:
- studentów względem wyników z przedmiotu,
- klientów względem ich kodu pocztowego,
- domów względem cen sprzedaży,
- miast względem populacji,
- krajów względem ich PNB (Produktu Narodowego Brutto, ang. GNP – Gross National Product),
- gwiazd względem ich jasności, etc.

Sortowanie może też być wstępnym krokiem do wyszukiwania (z poprzednich ćwiczeń wiemy, że wyszukiwanie binarne może być zastosowane tylko do posortowanych danych i jest znacznie szybsze niż wyszukiwanie liniowe).

Prawdopodobnie najprostszą metodą sortowania jest metoda bąbelkowa (bubble sort).
Algorytm bubble sort składa się z dwóch kroków:
1. Porównania dwóch sąsiednich elementów.
2. Jeśli to konieczne zamiana ich miejscami.

Przykładowo sortowanie zaczynamy od lewego krańca tablicy porównując dwa sąsiednie elementy i jeśli lewy okaże się większy od prawego to dokonujemy zamiany. W przeciwnym razie przesuwamy się o jeden element w prawo. Postępujemy tak aż do osiągnięcia prawego końca tablicy. W ten sposób po przejściu przez całą tablicę element o największej wartości trafia na koniec tablicy (tzn. jeśli tablica składa się z n elementów, największa wartość znajduje się pod indeksem n-1). Procedurę tą stosujemy n-1 razy za każdym razem zmniejszając prawy zakres sortowania o 1 (tzn. za drugim razem sortowaniu podajemy zakres tablicy o indeksach od 0 do n-2, za trzecim od 0 do n-3, itd.).

Przykład 1 (C++) Przykład klasy (ArrayBub) realizującej sortowanie bąbelkowe.

[bubbleSort.cpp]

```
//bubbleSort.cpp
//demonstrates bubble sort
#include <iostream>
#include <vector>
using namespace std;
//--------------------------------------------------------------
class ArrayBub
{
    private:
        vector<double> v;                   //vector v
        int nElems;                         //number of data items
//--------------------------------------------------------------
        void swap(int one, int two)         //private member function
        {
            double temp = v[one];
            v[one] = v[two];
            v[two] = temp;
```

```
        }
//--------------------------------------------------------------
    public:
//--------------------------------------------------------------
        ArrayBub(int max) : nElems(0)       //constructor
        {
            v.resize(max);                  //size the vector
        }
//--------------------------------------------------------------
        void insert(double value)           //put element into array
        {
            v[nElems] = value;              //insert it
            nElems++;                       //increment size
        }
//--------------------------------------------------------------
        void display()                      //displays array contents
        {
            for(int j=0; j<nElems; j++)     //for each element,
                cout << v[j] << " ";        //display it
            cout << endl;
        }
//--------------------------------------------------------------
        void bubbleSort()                   //sorts the array
        {
            int out, in;

            for(out=nElems-1; out>1; out--)   //outer loop (backward)
                for(in=0; in<out; in++)       //inner loop (forward)
                    if( v[in] > v[in+1] )     //out of order?
                        swap(in, in+1);       //swap them
        }  //end bubbleSort()
//--------------------------------------------------------------
};  //end class ArrayBub
////////////////////////////////////////////////////////////////
int main()
{
    int maxSize = 100;                      //array size
    ArrayBub arr(maxSize);                  //create the array

    arr.insert(77);                         //insert 10 items
    arr.insert(99);
    arr.insert(44);
    arr.insert(55);
    arr.insert(22);
    arr.insert(88);
    arr.insert(11);
    arr.insert(00);
    arr.insert(66);
    arr.insert(33);

    arr.display();                          //display items
    arr.bubbleSort();                       //bubble sort them
    arr.display();                          //display them again
    return 0;
}  //end main()
```

Invariants (niezmienniki, loop invariant – niezmiennik pętli)

W wielu algorytmach znajdują się warunki które pozostają niezmienione w trakcie wykonywania algorytmu. Warunki te nazywa się niezmiennikami (invariants). Rozpoznawanie niezmienników może być użyteczne w zrozumieniu algorytmu. W pewnych

warunkach mogą być również pożyteczne przy debugowaniu; możesz wielokrotnie sprawdzać czy invariant ma wartość logiczną true i sygnalizować wystąpienie błędu jeśli jest false.

W programie bubbleSort.cpp znajduje się jeden invariant: elementy o indeksach większych od outer są posortowane. Warunek ten pozostaje prawdziwy w trakcie działania algorytmu (przy pierwszym przebiegu, liczba posortowanych elementów wynosi zero (zero elementów na prawo od outer gdyż w pierwszy przebiegu outer oznacza również koniec tablicy)).

Efektywność algorytmu sortowania metodą bubble sort
Dla tablicy o 10 elementach potrzeba 9 porównań przy pierwszym przebiegu tablicy, 8 przy drugim, 7 przy trzecim, itd. Stąd dla 10 elementów mamy

$$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$$

porównań. W ogólności dla N elementów potrzeba

$$(N-1) + (N-2) + (N-3) + \ldots + 1 = N*(N-1)/2$$

porównań.
Oznacza to, że w notacji O() efektywność algorytmu jest typu O(N^2) (w notacji O() pomija się wszelkie stałe).

Sortowanie przez wstawianie (insertion sort)
Sortowanie metodą insertion sort jest zasadniczo lepsze niż metodą bubble sort. Nadal wykonuje się w czasie O(N^2) lecz jest dwa razy szybsze od bubble sort.

Opis działania algorytmu.
Zakładamy, że mamy tablice częściowo już posortowanych elementów. Niech posortowane elementy (w porządku rosnącym) znajdują się na lewo od elementu o indeksie out. Element o indeksie out zapisujemy do zmiennej tymczasowej temp i porównujemy elementy posortowane zaczynając od największego (indeks out-1) z wartością przechowywaną w temp. Jeśli element e(out-1) jest większy niż temp przesuwamy e(out-1) w górę tablicy o jedno miejsce (tzn. w miejsce e(out); wówczas wolnym miejscem staje się e(out-1)). Procedurę powtarzamy tak długo aż napotkamy element e(index) którego wartość będzie mniejsza niż temp. Wówczas wartość temp zapisujemy do e(index+1) i zwiększamy out o jeden (out=out+1). Powtarzamy wszystkie kroki do momentu gdy out będzie równe N-1 (gdzie N – rozmiar tablicy).

Przykład 2 (C++) Przykład klasy (ArrayIns) realizującej sortowanie przez wstawianie.

[insertSort.cpp]

```
//insertSort.cpp
//demonstrates insertion sort
#include <iostream>
#include <vector>
using namespace std;
//--------------------------------------------------------------
class ArrayIns
{
    private:
        vector<double> v;                   //vector v
        int nElems;                         //number of data items
```

```
//--------------------------------------------------------------
    public:
        ArrayIns(int max) : nElems(0)       //constructor
        {
            v.resize(max);                  //size the vector
        }
//--------------------------------------------------------------
        void insert(double value)           //put element into array
        {
            v[nElems] = value;              //insert it
            nElems++;                       //increment size
        }
//--------------------------------------------------------------
        void display()                      //displays array contents
        {
            for(int j=0; j<nElems; j++)     //for each element,
                cout << v[j] << " ";        //display it
            cout << endl;
        }
//--------------------------------------------------------------
        void insertionSort()
        {
            int in, out;
            for(out=1; out<nElems; out++)       //out is dividing line
            {
                double temp = v[out];           //remove marked item
                in = out;                       //start shifts at out
                while(in>0 && v[in-1] >= temp)  //until one is smaller,
                {
                    v[in] = v[in-1];            //shift item to right
                    --in;                       //go left one position
                }
                v[in] = temp;                   //insert marked item
            }  //end for
        }  //end insertionSort()
//--------------------------------------------------------------
};  //end class ArrayIns
////////////////////////////////////////////////////////////////
int main()
{
    int maxSize = 100;              //array size
    ArrayIns arr(maxSize);          //create array

    arr.insert(77);                 //insert 10 items
    arr.insert(99);
    arr.insert(44);
    arr.insert(55);
    arr.insert(22);
    arr.insert(88);
    arr.insert(11);
    arr.insert(00);
    arr.insert(66);
    arr.insert(33);

    arr.display();                  //display items
    arr.insertionSort();            //insertion-sort them
    arr.display();                  //display them again
    return 0;
}  //end main()
```

Invariants w algorytmie sortowania insertion sort
Pod koniec każdego przebiegu, po wstawieniu elementu do temp, elementy o mniejszych indeksach niż out są częściowo posortowane.

Efektywność algorytmu sortowania metodą insertion sort
Przy pirwszym przebiegu potrzebne jest porównanie z maksimum jednym elementem. Przy drugim przebiegu potrzebne jest porównanie z maksimum dwoma elementami (maksimum gdyż może się okazać, że już pierwszy porównywany jest mniejszy niż ten przechowywany w zmiennej temp), i tak dalej aż do maksimum N-1 porównań przy ostatnim przebiegu. W ten sposób potrzebujemy maksimum

$$1 + 2 + 3 + \ldots + (N-1) = N*(N-1)/2$$

porównań.
W rzeczywistości porównań będzie mniej (średnio w połowie przypadków element porównywany z temp będzie znajdował się w pierwszej połowie tablicy na lewo od out).

Efektywność algorytmu sortowania metodą przez wstawianie jest typu O(N^2).

Sortowanie obiektów
Do tej pory stosowaliśmy sortowanie do danych typu double. W rzeczywistości bardziej prawdopodobnym jest sortowanie obiektów, np. sortowanie względem nazwisk tablicy wskaźników do obiektów przechowujących dane osobowe (m.in. nazwiska).

Przykład 3 (C++) Przykład realizacji sortowania obiektów metodą przez wstawianie.

[objectSort.cpp]

```
//objectSort.cpp
//demonstrates sorting objects (uses insertion sort)
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Person
{
    private:
        string lastName;
        string firstName;
        int age;

    public:
        Person(string last, string first, int a) :   //constructor
            lastName(last), firstName(first), age(a)
        {  }

        void displayPerson()
        {
            cout << "   Last name: " << lastName;
            cout << ", First name: " << firstName;
            cout << ", Age: " << age << endl;
        }

        string getLast()                              //get last name
        { return lastName; }
```

```
};  //end class Person
//--------------------------------------------------------------
class ArrayInOb
{
    private:
        vector<Person*> v;                  //vect of ptrs to Persons
        int nElems;                         //number of data items

    public:
        ArrayInOb(int max) : nElems(0)      //constructor
        {
            v.resize(max);                  //size the vector
        }

        //put person into array
        void insert(string last, string first, int age)
        {
            v[nElems] = new Person(last, first, age);
            nElems++;                       //increment size
        }

        void display()                      //displays array contents
        {
            for(int j=0; j<nElems; j++){    //for each element,
                cout << v[j]; v[j]->displayPerson();       //display it
            }
        }

        void insertionSort()
        {
            int in, out;
            for(out=1; out<nElems; out++)
            {
                Person* temp = v[out];      //out is dividing line
                in = out;                   //start shifting at out
                                            //until smaller one found,
                while( in>0 && v[in-1]->getLast() > temp->getLast() )
                {
                    v[in] = v[in-1];        //shift item to the right
                    --in;                   //go left one position
                }
                v[in] = temp;               //insert marked item
            }  //end for
        }  //end insertionSort()

};  //end class ArrayInOb
//--------------------------------------------------------------
int main()
{
    int maxSize = 100;              //array size
    ArrayInOb arr(maxSize);         //create array

    arr.insert("Evans", "Patty", 24);
    arr.insert("Smith", "Doc", 59);
    arr.insert("Smith", "Lorraine", 37);
    arr.insert("Smith", "Paul", 37);
    arr.insert("Yee", "Tom", 43);
    arr.insert("Hashimoto", "Sato", 21);
    arr.insert("Stimson", "Henry", 29);
    arr.insert("Velasquez", "Jose", 72);
    arr.insert ("Vang", "Minh", 22);
```

```
    arr.insert("Creswell", "Lucinda", 18);

    cout << "Before sorting:" << endl;
    arr.display();                  //display items

    arr.insertionSort();            //insertion-sort them

    cout << "After sorting:" << endl;
    arr.display();                  //display them again
    return 0;
}  //end main()
```

Inne własności algorytmów sortowania: stabilność

Załóżmy, że chcemy posortować dane pracowników alfabetycznie względem nazwisk. Następnie chcemy jeszcze raz posortować dane lecz według kodów pocztowych (tzn. chcemy aby elementy o tym samym kodzie pocztowym zachowały porządek nazwisk z poprzedniego, alfabetycznego sortowania). O algorytmach sortowania zachowujących porządek z poprzedniego sortowania mówimy, że są stabilne. Oba algorytmy sortowania (insertion sort i bubble sort) są algorytmami stabilnymi.

Opracowano na podstawie:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"