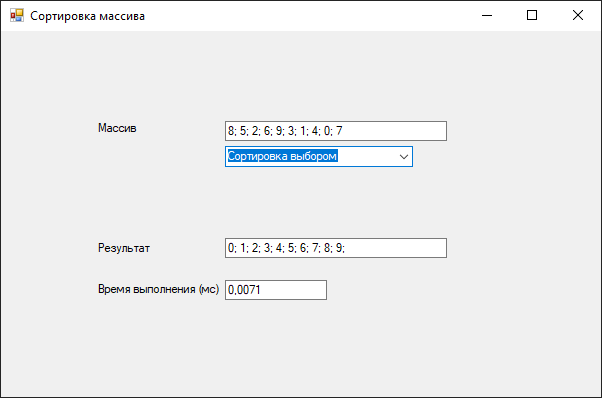
Методы сортировки массива

Интерфейс приложения



Пользователь вводит массив и выбирает из выпадающего списка метод сортировки, после массив сортируется соответствующими способом. Отсортированный массив выводится в результирующее поле. Также в соответствующее поле выводится время работы алгоритма.

Реализация приложения

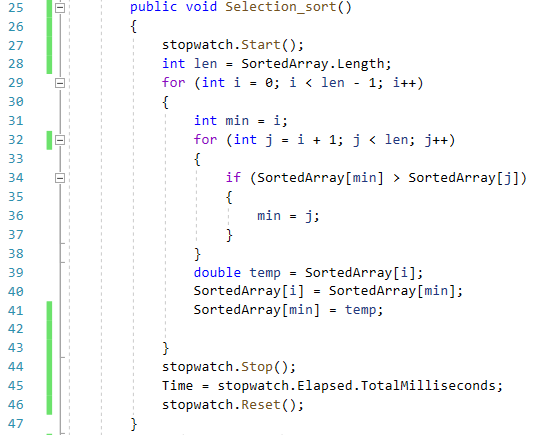
Приложение разработано на языке C# в IDE MS Visual Studio. Программа включает два основных класса *MainFom.cs* (Windows Forms) и *AlgorithmsClass.cs,* который содержит реализацию методов сортировки.

* В классе *MainFom.cs* содержится выпадающий список (ComboBox), в который добавлены алгоритмы сортировки.
* В переменную Item записывается название выбранного алгоритма.
* Данные из поля массив (tbInput) записываются в массив inputDate (string).
* Определяется переменная n, равная длине массива inputDate, и создается массив inputArray(double) с длиной n.
* В цикле осуществляется запись элементов inputDate, преобразованных в тип double, в массив inputArray. Порядок элементов сохраняется.
* Вызывается метод класса algorithmsClass CopyArray. Аргументом метода является массив inputArray. Таким образом создается копия SortedArray в объекте класса *AlgorithmsClass.cs*
* Выбор метода сортировки из выпадающего список реализован конструкцией switch. В каждом case вызывается соответствующий метод *algorithmsClass.*
* Методы класса *algorithmsClass.cs*, представляющие собой реализацию алгоритмов сортировки массива, записывают обработанные массивы в SortedArray. SortedArray преобразовывается в строку и записывается в поле Результат (tbOutput)
* Время выполнения алгоритма записывается в поле Время выполнения (tbTime)
* При вводе пользователем данных, не являющихся массивом, или значений содержащие не только числа, в tbOutput выводится сообщение об ошибке

1. Сортировка выбором
   1. Описание алгоритма
2. Находим номер минимального значения в текущем массиве
3. Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
4. Сортируем оставшуюся часть списка (правая часть), исключив из рассмотрения уже отсортированные элементы

Пример:

* {8; 5; 2; 6; 9; 3; 1; 4; **0**; 7}
* {**0**; 5; 2; 6; 9; 3; **1**; 4; 8; 7}
* {**0**; **1**; 2; 6; 9; **3**; 5; 4; 8; 7}
* …
* {**0**; **1**; **2**; **3**; **4**; **5**; **6**; **7**; **8**; **9**}
  1. Реализация



* Во внешнем цикле по i определяется индекс элемента массива, для которого осуществляется поиск соответствующего значения в отсортированном массиве.
* Во вложенном цикле по j осуществляется поиск локального минимума по ещё не отсортированный элементам массива
* Если текущий минимум (SortedArray[min]) больше j-го элемента, то он принимается в качестве минимума
* Найденный во внутреннем цикле локальный элемент записывается в i-ую элемент массива. На место же локального минимума записывается старое значение i-ого элемента.

1. Сортировка вставками

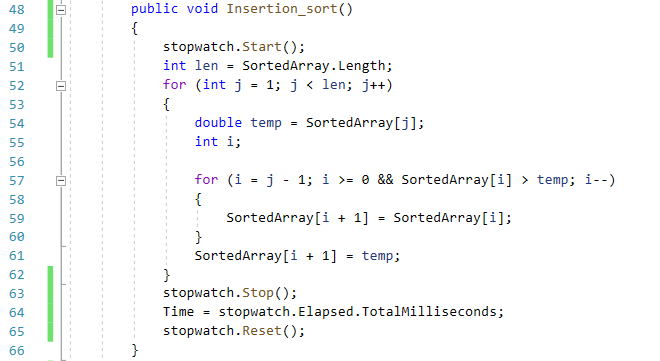
### 2.1. Описание алгоритма

1. Перебираются значения в неотсортированной части массива.
2. Каждый элемент сравнивается с отсортированным и вставляется после (правее) того отсортированного элемента, который оказался меньше

Пример:

* {**8**; **5**; 2; 6; 9; 3; 1; 4; 0; 7}
* {**5**; **8**; **2**; 6; 9; 3; 1; 4; 0; 7}
* {**2**; **5**; **8**; **6**; 9; 3; 1; 4; 0; 7}
* …
* {**0**; **1**; **2**; **3**; **4**; **5**; **6**; **7**; **8**; **9**}

### 2.2. Реализация



* Выполняется внешний цикл перебора не отсортированных элементов, пока не будет рассматриваться последний элемент
  + Выполняется вложенный цикл смещения отсортированных элементов вправо пока рассматриваемый элемент не найдет свое место (пока не найдется элемент меньший по значению или рассматриваемы не достигнет начала массива)
  + Рассматриваемый элемент занимает свое место

## Сортировка перестановками

## 

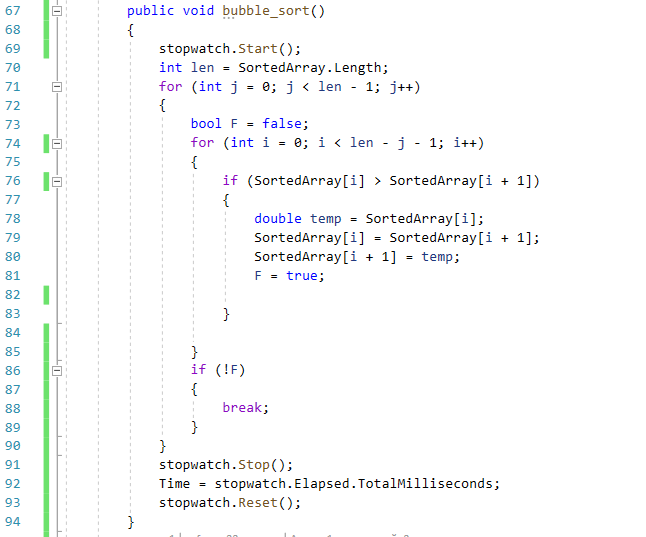
### 3.1. Описание алгоритма

После первого завершения внутреннего цикла максимальный элемент массива всегда находится на последней позиции. При втором проходе, следующий по значению максимальный элемент находится на предпоследнем месте. И так далее.

Пример

* {**8**; **5**; 2; 6; 9; 3; 1; 4; 0; 7}
* {5; **8**; **2**; 6; 9; 3; 1; 4; 0; 7}
* …
* {**5**; **2**; 6; 8; 3; 1; 4; 0; 7; **9**}
* …
* {**0**; **1**; **2**; **3**; **4**; **5**; **6**; **7**; **8**; **9**}

3.2. Реализация



* Во внешний цикл выполняется, пока все элементы не займут соответствующие места (сортировка идет с конца массива)
* Вложенный цикл смещения больших значений выполняется, пока не будет достигнут отсортированный элемент в конце массива:
  + если текущий элемент больше следующего, они меняются местами

## Сортировка быстрая

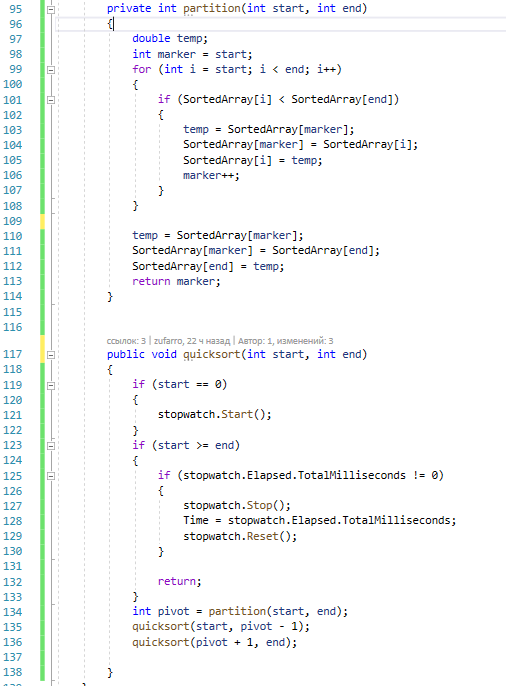
### 4.1. Описание алгоритма

* Выбирается опорный элемент массива. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. В нашем случае в качестве опроного элемента всегда рассматривается последний элемент массива.
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные и большие».
* Для отрезков «меньших» и «равных и больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Пример

* {8; 5; 2; 6; 9; 3; 1; 4; 0; 7}
* {5; 2; 6; 3; 1; 4; 0; **7**; 9; 8}
* {**0**; 2; 6; 3; 1; 4; 5; **7**; 9; 8}
* {**0**; 2; 6; 3; 1; 4; 5; **7**; **8**; **9**}
* …
* {0; 1; 2; 3; 4; 5; 6; 7; 8; 9}

### 4.2. Реализация



Для реализации алгоритма были написаны 2 функции: *partition* и *quicksort*. параметрами функция являются переменный *start* и *end*, содержащие значения индексов начала и конца рассматриваемой части массива.

*partition* - возвращает искомую позицию опорного элемента и тем самым разделяет массив на 2 части: «элементы меньшие опорного», «равные и большие».

* Вводится переменная *marker*, в которую в конце цикла запишется индекс элемента, имеющего граничное значение между двумя искомыми частями массива( равный или больший опорного элемента). Начальным значением для этой переменной послужит самый левый элемент.
* Осуществляется обход массива.
  + Если находится меньший элемент, чем опорный, то его значение и элемента массива с индексом *marker* обмениваются. также в этом случае значение переменной *marker* увеличивается на единицу.
  + В противном случае мы игнорируем текущий элемент.
* После выхода из цикла элемент с индексом *marker* и опорный элемент обмениваются значениями.

Пример *partition*

* {**8**; 5; 2; 6; 9; 3; 1; 4; 0; 7}
* {**5**; **8**; 2; 6; 9; 3; 1; 4; 0; 7}
* {**5**; **2**; **8**; 6; 9; 3; 1; 4; 0; 7}
* …
* {5; 2; 6; 3; 1; 4; 0; **7**; 9; 8}

*quicksort* - рекурсивная функция, имплементирующая алгорит.

* Вызывает функцию partition для определения позиции опорного элемента (последнего элемента в начальном массиве). После вызова этой функции опорный элемент будет находится на нужной позиции.
* Далее функция quicksort рекурсивно вызывается для частей массива находящихся слева и справа от опорного элемента.