Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания  
 к лабораторной работе № 2.9  
 по курсу «Основы алгоритмизации   
и программирования»**

«Сортировка и поиск»

Преподаватель: Войтехович   
Агния Витольдовна

Составитель: Войтехович   
Агния Витольдовна

Полоцк, 2017

**ЦЕЛЬ РАБОТЫ**

Изучить быстрые и медленные алгоритмы сортировки. Изучить алгоритмы линейного и бинарного поиска.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В мире компьютеров сортировка и поиск принадлежат к числу наиболее распространенных и хорошо изученных задач. Процедуры сортировки и поиска используются почти во всех программах управления базами данных, (а также в компиляторах, интерпретаторах и операционных системах. В настоящей главе представлены основные алгоритмы сортировки и поиска. Как вы сможете убедиться, они также иллюстрируют некоторые важные приемы программирования на языке С. Вообще говоря, поскольку целью сортировки, является облегчение и ускорение поиска данных, алгоритмы сортировки рассматриваются в первую очередь.

# 1. Сортировка

Сортировка — это упорядочивание набора однотипных данных по возрастанию или убыванию. Сортировка является одной из наиболее приятных для умственного анализа категорией алгоритмов, поскольку процесс сортировки очень хорошо определен. Алгоритмы сортировки были подвергнуты обширному анализу, и способ их работы хорошо понятен. К сожалению, вследствие этой изученности сортировка часто воспринимается как нечто само собой разумеющееся. При необходимости отсортировать данные многие программисты просто вызывают стандартную функцию **qsort()**, входящую в стандартную библиотеку **<cstdlib>**. Однако различные подходы к сортировке обладают разными характеристиками. Несмотря на то, что некоторые способы сортировки могут быть в среднем лучше, чем другие, ни один алгоритм не является идеальным для всех случаев. Поэтому широкий набор алгоритмов сортировки — полезное добавление в инструментарий любого программиста.

Будет полезно кратко остановиться на том, почему вызов **qsort()** не является универсальным решением всех задач сортировки. Во-первых, функцию общего назначения вроде **qsort()** невозможно применить во всех ситуациях. Например, **qsort()** сортирует только массивы в памяти. Она не может сортировать данные, хранящиеся в связанных списках. Во-вторых, **qsort()** - параметризованная функция, благодаря чему она может обрабатывать широкий набор типов данных, но вместе с тем вследствие этого она работает медленнее, чем эквивалентная функция, рассчитанная на какой-то один тип данных. Наконец, как вы увидите, хотя алгоритм быстрой сортировки, примененный в функции **qsort()**, очень эффективен в общем случае, он может оказаться не самым лучшим алгоритмом в некоторых конкретных ситуациях.

Существует две общие категории алгоритмов сортировки: алгоритмы, сортирующие объекты с произвольным доступом (например, массивы или дисковые файлы произвольного доступа), и алгоритмы, сортирующие последовательные объекты (например, файлы на дисках и лентах или связанные списки[1]). В данной лабораторной работе рассматриваются только алгоритмы первой категории, поскольку они наиболее полезны для среднестатистического программиста.

Чаще всего при сортировке данных лишь часть их используется в качестве ключа сортировки. Ключ — это часть информации, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. Для простоты в нижеследующих примерах будет производиться сортировка массивов символов, в которых ключ и данные совпадают. Далее вы увидите, как адаптировать эти методы для сортировки структур данных любого типа.

# 1.1 Классы алгоритмов сортировки

Существует три общих метода сортировки массивов:

Обмен

Выбор (выборка)

Вставка

Чтобы понять, как работают эти методы, представьте себе колоду игральных карт. Чтобы отсортировать карты методом обмена[1], разложите их на столе лицом вверх и меняйте местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе выбора разложите карты на столе, выберите карту наименьшей значимости и положите ее в руку. Затем из оставшихся карт снова выберите карту наименьшей значимости и положите ее на ту, которая уже находится у вас в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом вставки, возьмите все карты в руку. Выкладывайте их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.

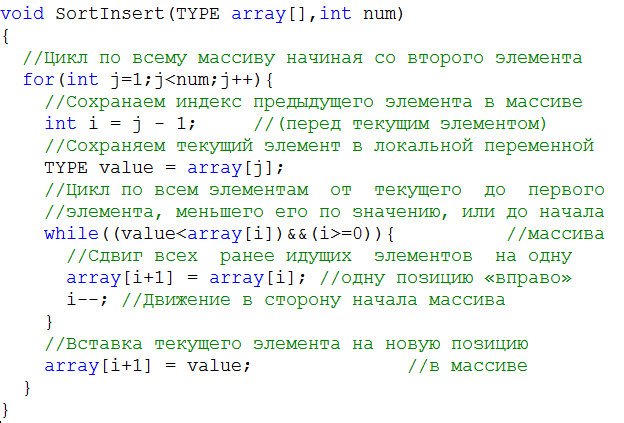
# 1.2 Медленные алгоритмы сортировки

Среди медленных алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

* алгоритм сортировки вставками;
* алгоритм сортировки пузырьком;
* минимаксные алгоритмы;
* алгоритм сортировки перечислениями.

Алгоритм сортировки вставками заключается в прохождении по всем элементам массива, начиная со второго, при котором каждый элемент массива вставляется на свое место. Для этого, начальный элемент каждой итерации цикла помещается во временную переменную и его значение сравнивается со всеми элементами, стоящими до этого элемента. Если их значение больше текущего, то они сдвигаются на один элемент вправо. После достижения элемента меньшего текущего, или начала массива, текущий элемент вставляется на новое место. Так продолжается до последнего элемента массива. Листинг функции сортировки вставками:

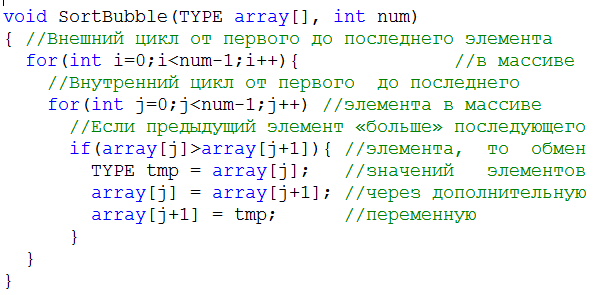
**ПРИМЕЧАНИЕ**: Здесь и далее во всех функциях сортировки:



 первый параметр *array* - массив, который необходимо упорядочить;

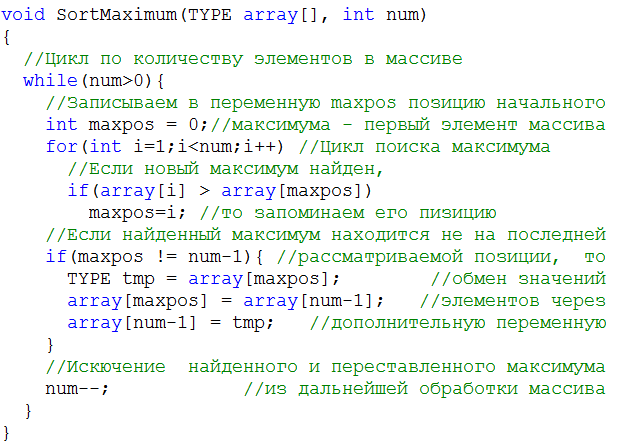
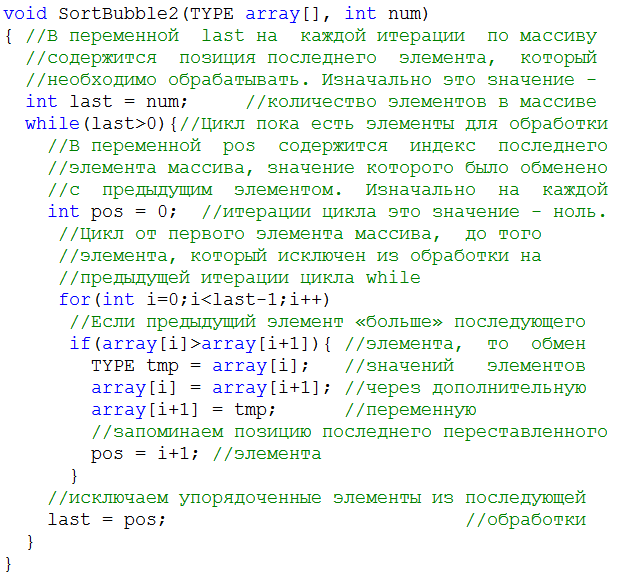
 второй параметр *num* - количество элементов в массиве.

Алгоритм пузырьковой сортировки является наиболее распространенным среди медленных алгоритмов сортировки. Он заключается в сравнении двух элементов попарно и их перестановкой согласно направлению сортировки. Существует несколько реализаций. Листинг функции сортировки полным пузырьком (алгоритм полного перебора):

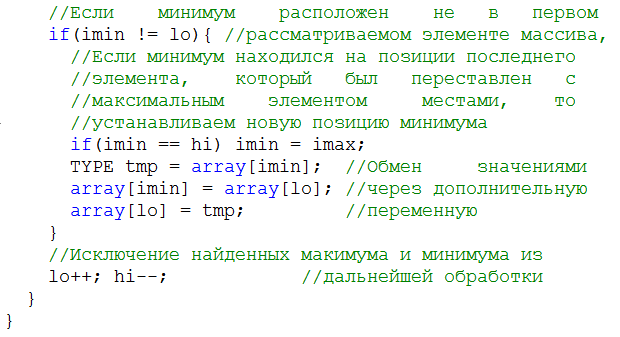
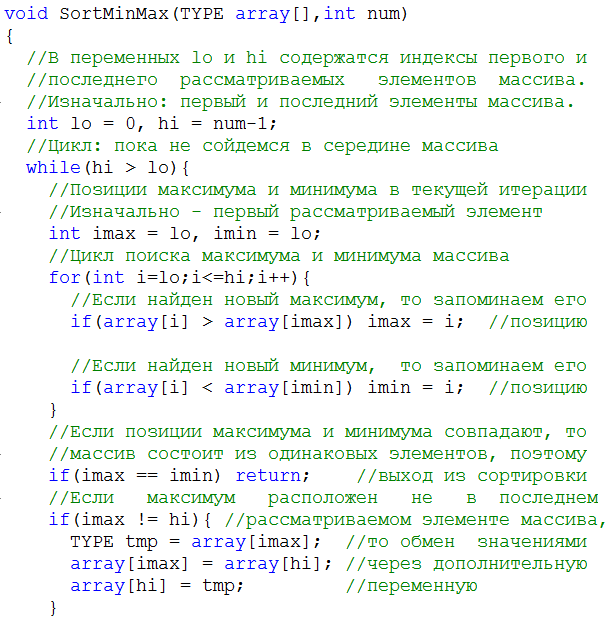
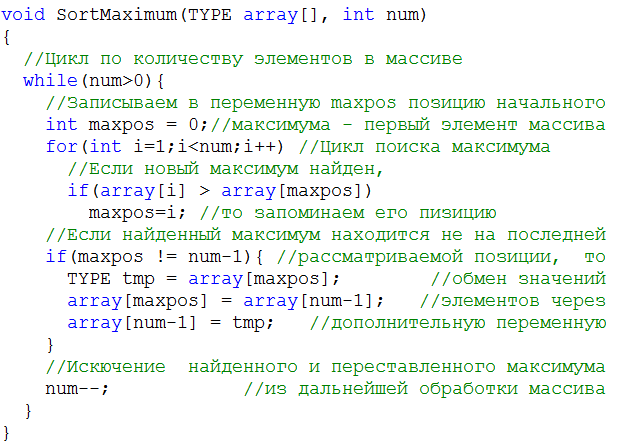


Приведенный алгоритм сортировки полным пузырьком не учитывает «эффекта пузырька»: самый легкий (тяжелый) элемент за одну итерацию по массиву поднимается (опускается) до самого конца массива, поэтому его следует исключить из обработки на последующих итерациях по массиву. Далее представлен листинг функции сортировки, реализующей алгоритм сортировки простым пузырьком учитывающий «эффект пузырька»:

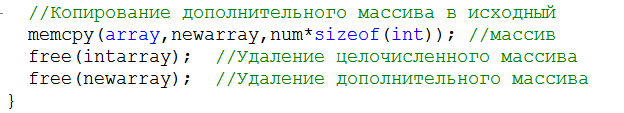
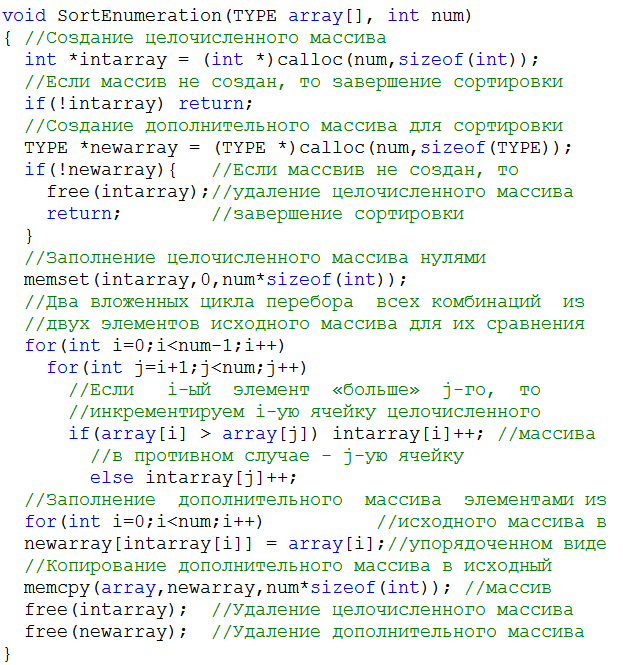
Суть минимаксных алгоритмов сортировки заключается в поиске максимального и/или минимального элементов в массиве и обмен значений найденных элементов с последним и/или первым элементами в массиве. После каждой итерации границы массива сужаются. Например, листинг функции сортировки массива, реализующей максимальный алгоритм сортировки (поиск максимума и помещение его в конец массива):



Далее приведен алгоритм,шелл реализованный в виде функции, осуществляющий сортировку минимаксным способом: ищутся и максимальный и минимальный элементы и помещаются в конец и начало рассматриваемого массива соответсвенно.



Идея сортировки перечислением состоит в том, чтобы сравнить попарно все элементы и подсчитать, сколько из них меньше каждого отдельного элемента. Для подсчета используется вспомогательный массив, в котором, после завершения подсчета, находится окончательное положение элементов исходного массива в упорядоченной последовательности. Листинг функции сортировки перечислением:



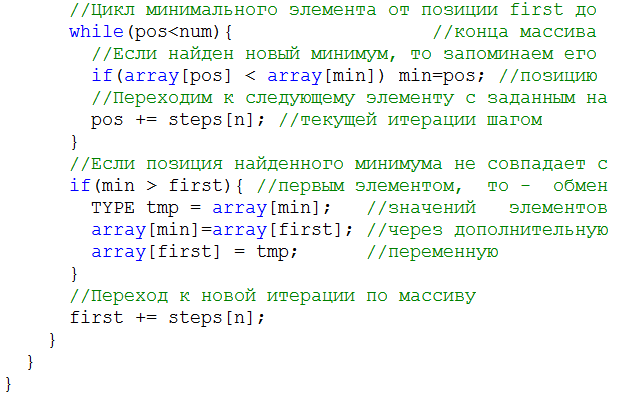
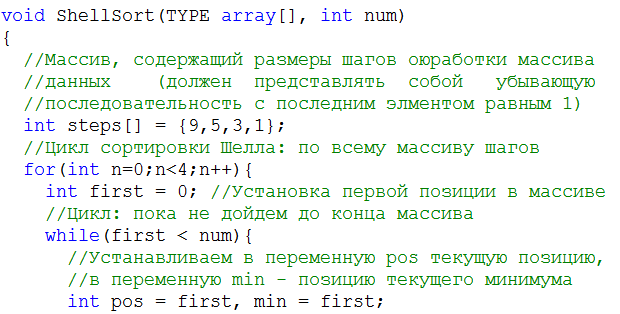
# 1.3 Быстрые алгоритмы сортировки

Среди быстрых алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

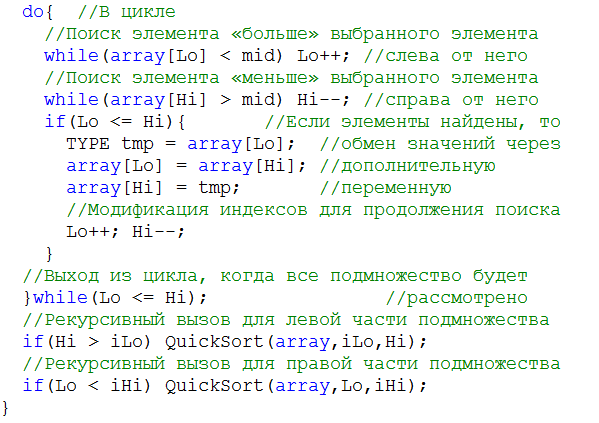
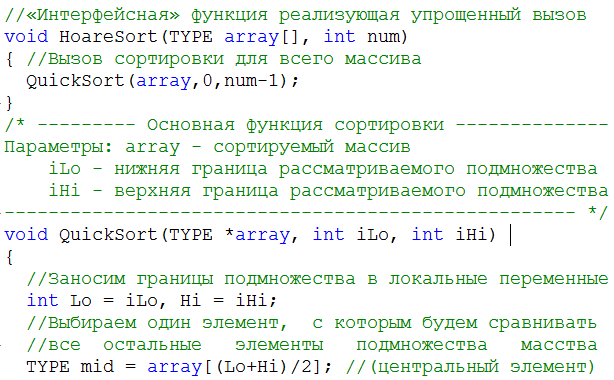
* алгоритм Шелла;
* алгоритм Хоара;
* алгоритм Флойда.

Идея алгоритма Шелла состоит в том, что в исходном наборе элементов (массиве) сначала упорядочиваются элементы, расположенные на расстоянии друг от друга, т.е. итерации по массиву осуществляются через несколько элементов. Затем шаг уменьшается и производится сортировка с новым размером шага. Так продолжается до тех пор, пока шаг не станет равным единице. Таким образом, алгоритм Шелла представляет собой обычный медленный алгоритм сортировки, в котором ускорение процесса осуществляется за счет того, что на первых этапах (с большим шагом) обрабатывается меньше элементов, и они упорядочиваются быстрее, перемещаясь по массиву с большими шагами. На практике скорость данного алгоритма редко достигает указанной ранее зависимости (n×log2n), поэтому данный алгоритм относят к классу быстрых алгоритмов скорее условно, чем в действительности. Сама сортировка в алгоритме Шелла может осуществляться любым из рассмотренных ранее медленных алгоритмов.

Например, листинг функции сортировки алгоритмом Шелла, в которой в качестве основного (внутреннего) алгоритма сортировки используется алгоритм сортировки с поиском минимума и вставкой его на позицию первого из рассматриваемых элементов:



Алгоритм быстрой сортировки Хоара заключается в следующем: берется массив, выбирается один элемент, а все остальные элементы делятся на два подмножества - элементы, большие выбранного или равные ему, и элементы, меньшие выбранного. Затем процедура применяется рекурсивно к каждому из двух подмножеств. Если в подмножестве меньше двух элементов, сортировка ему не требуется, и рекурсия на этом прекращается. Листинг функции сортировки алгоритмом Хоара:



Алгоритм Флойда является самым оптимальным из алгоритмов сортировки. В нем активно используется упорядоченное двоичное дерево, пример которого представлен на рисунке 1.



Рисунок 1 - Пример упорядоченного двоичного дерева

Значение в каждой из его вершин не меньше, чем значения в его дочерних вершинах. Двоичное дерево называется частично упорядоченным, если свойство упорядоченности выполняется для каждой из его вершин, однако для корня это свойство нарушается. Пример частично упорядоченного дерева приведен на рисунке 2.



Рисунок 2 - Пример частично упорядоченного двоичного дерева

Структура дерева позволяет сохранить состояние процесса сортировки последовательности на каждом его шаге, с целью использования этого состояния в дальнейших расчетах и уменьшении числа операций сравнения при поиске наибольшего (наименьшего) из оставшихся элементов.

В алгоритме сортировки Флойда исходная последовательность данных представляется в виде дерева на смежной памяти. В таком дереве ребра присутствуют неявно и вычисляются с помощью арифметических операций над индексами элементов массива. Корень дерева a[0], за каждой вершиной a[k] следуют вершины a[2k+1] и a[2k+2]. На рисунке 3 приведен пример нумерации вершин двоичного дерева на смежной памяти.



Рисунок 3 - Пример двоичного дерева на смежной памяти

Основу алгоритма составляет функция SURFACE(a[i..k]) всплытия Флойда, которая за Q(n)~(log2n) сравнений преобразует почти упорядоченное поддерево в упорядоченное. Поддерево представляется на одномерно массиве a[i..k], где a[i] - корень дерева, a[k] - максимальный элемент массива, который еще может принадлежать поддереву (рисунок 4).



Рисунок 4 - Двоичное дерево на смежной памяти a[i..k]

Функция заключается в том, что значение корня (здесь может нарушаться условие упорядоченности) всплывает по направлению к листьям (последний уровень вершин в дереве) до тех пор, пока дерево не преобразуется в упорядоченное. Функция SURFACE всплытия Флойда позволяет в почти упорядоченном дереве найти наибольший (наименьший) элемент за число сравнений Q(log2n), преобразуя дерево к упорядоченному виду. В результате найденный элемент будет располагаться в вершине дерева. Для сортировки множества элементов, из них сначала организуется почти упорядоченное двоичное дерево при помощи повторного применения функции SURFACE сначала к самым мелким его поддеревьям от листьев и затем ко все более крупным. Листья тривиально упорядочены, поэтому можно начинать с минимальных поддеревьев, содержащих несколько вершин и укрупнять их, каждый раз полностью, применяя алгоритм всплытия до тех пор, пока не будет достигнут корень дерева. Заметим, что каждое из поддеревьев, к которому применяется алгоритм всплытия, удовлетворяет условию почти упорядоченности, поскольку упорядочивание происходит от листьев к корню.

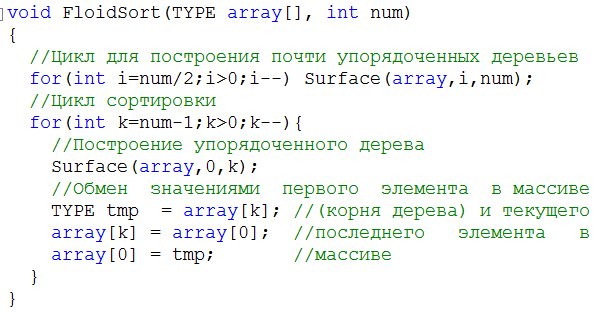
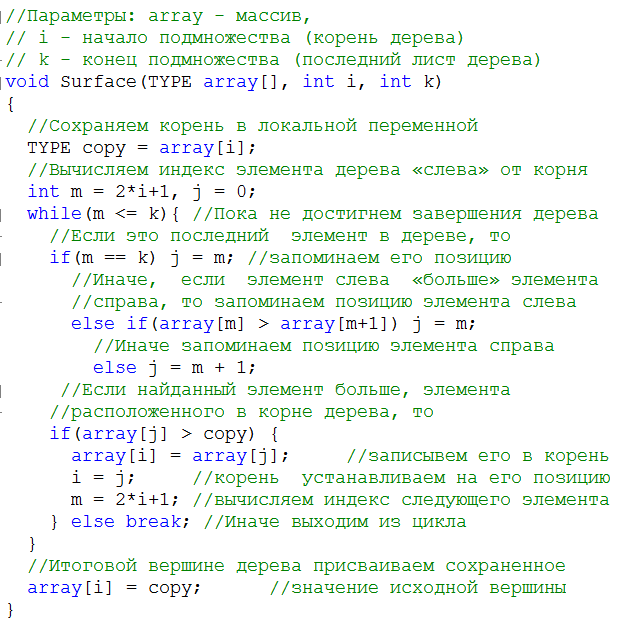
После того, как дерево упорядочено, наибольший (наименьший) элемент оказывается в его корне. Найденный элемент меняют местами с самым последним листом в дереве (последний элемент рассматриваемого массива), дерево уменьшается на одну вершину и все готово для определения нового наибольшего (наименьшего) элемента множества. На рисунке 5 показана полная последовательность перестановок и всплытий, которые происходят после формирования из исходного множества почти упорядоченного дерева и вплоть до того, как в этом дереве останется всего одна вершина, а исходное множество окажется отсортированным.



Рисунок 5 - Пример сортировки чисел 18,4,56,65,37,63,66 методом Флойда

Листинг функции SURFACE:

Листинг основной функции сортировки Флойда:



При выборе того или иного алгоритма сортировки следует учитывать то, что быстрые алгоритмы выгодны для сортировки достаточно больших массивов (размер которых исчисляется сотнями и даже тысячами элементов). Также необходимо обращать внимание на то, что сортируемые данные могут иметь большой объем. Поэтому алгоритмы сортировки следует реализовывать так, чтобы минимизировать число перестановок элементов в процессе сортировки.

# 2. Поиск

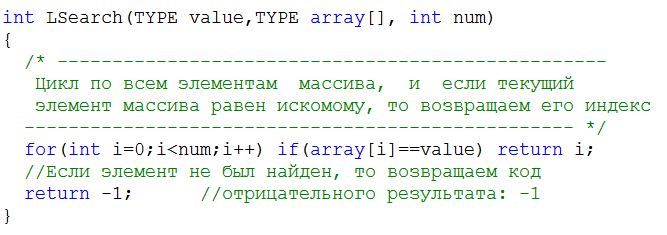
Базы данных существуют для того, чтобы время от времени пользователи могли найти нужную запись, введя ее ключ. Существует один метод поиска информации в неупорядоченном массиве, и другой для поиска в упорядоченном массиве. В набор стандартной библиотеки **<cstdlib>** входит стандартная функция **bsearch()**. Тем не менее, как и в случае сортировки, процедуры общего назначения иногда совсем не эффективны при использовании в критических ситуациях из-за накладных расходов, связанных с их обобщением. Кроме того, функцию **bsearch()** невозможно применить к неупорядоченным данным.

# 2.1 Методы поиска

Для нахождения информации в неупорядоченном массиве требуется последовательный поиск, начинающийся с первого элемента и заканчивающийся при обнаружении подходящих данных либо при достижении конца массива. Этот метод применим для неупорядоченной информации, но также можно использовать его и на отсортированных данных. Однако если данные уже отсортированы, можно применить двоичный поиск, который находит данные быстрее.

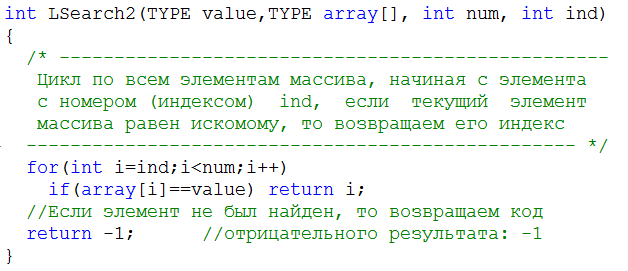
# 2.2 Последовательный поиск

Сущность этого алгоритма заключается в последовательном просмотре всех элементов множества и сравнении каждого из них с искомым значением. Как только элемент множества будет найден (его значение совпадет с искомым согласно условиям поиска), то поиск завершается и возвращается номер найденного элемента во множестве. Если все множество было просмотрено, а искомый элемент не найден, то возвращается код отрицательного результата поиска. Листинг функции линейного поиска:



Первый параметр value - искомое значение, второй параметр array - множество элементов в котором осуществляется поиск (массив), третий параметр num - количество элементов во множестве. Функция возвращает: положительное значение или ноль - индекс найденного элемента множества, -1 - если элемент не найден.

Приведенная функция позволяет найти только первый элемент во множестве. Для организации продолжения поиска элементов необходимо модифицировать данную функцию, добавив еще один параметр: номер элемента, с которого необходимо начинать поиск. Листинг модифицированной функции:



# 2.3 Двоичный поиск

Если данные, в которых производится поиск, отсортированы, для нахождения элемента можно применять метод, намного превосходящий предыдущий — двоичный поиск[1]. В нем применяется метод половинного деления. Сначала проверим средний элемент. Если он больше, чем искомый ключ, проверим средний элемент первой половины, в противном случае — средний элемент второй половины. Будем повторять эту процедуру до тех пор, пока искомый элемент не будет найден либо пока не останется очередного элемента.

Например, чтобы найти число 4 в массиве

1 2 3 4 5 6 7 8 9

при двоичном поиске сначала проверяется средний элемент — число 5. Поскольку оно больше, чем 4, поиск продолжается в первой половине:

1 2 3 4 5

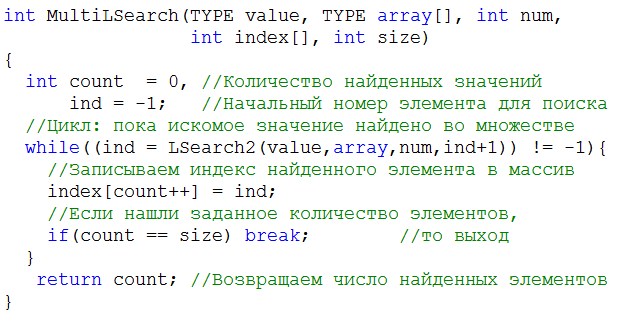
Средний элемент теперь равен 3. Это меньше, чем 4, поэтому первая половина отбрасывается. Поиск продолжается в части

4 5

На этот раз искомый элемент найден.

В двоичном поиске количество сравнений в худшем случае равно log2n. В среднем случае количество немного ниже, а в лучшем — количество сравнений равно 1.

Для реализации множественного поиска необходимо в функцию множественного поиска передать массив, куда будут занесены индексы найденных элементов, а также размер этого массива (чтобы не допустить выхода за его границы). Реализация функции множественного линейного поиска:



Параметры функции:

* value - искомое значение;
* array - множество, в котором осуществляется поиск;
* num - количество элементов во множестве;
* index - массив, в котором будут содержаться индексы найденных элементов;
* size - размер массива, ограничение количества искомых элементов.

Функция возвращает количество найденных элементов.

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ № 2\_9

**Задание 1.** Организуйте выгрузку данных из Вашей структуры данных в динамический массив.

**Задание 2.** Организуйте выгрузку данных из динамического массива в Вашу структуру данных.

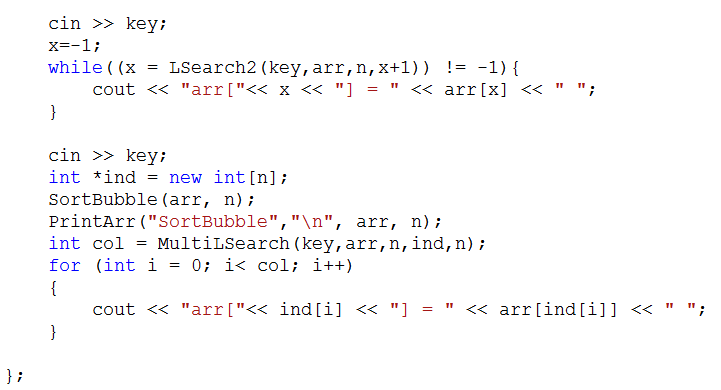
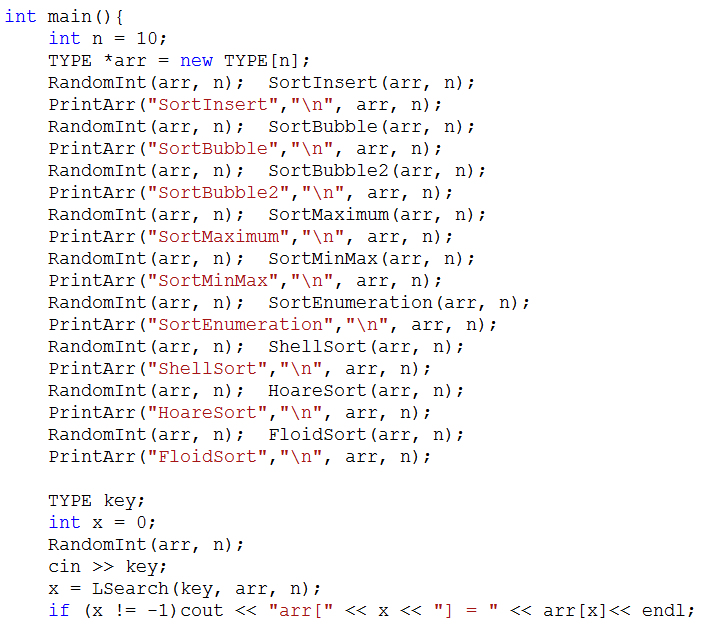
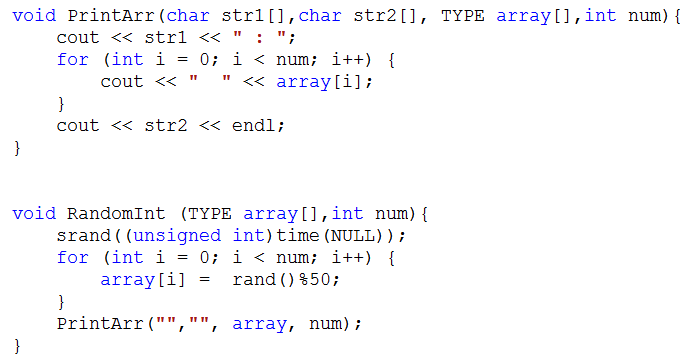
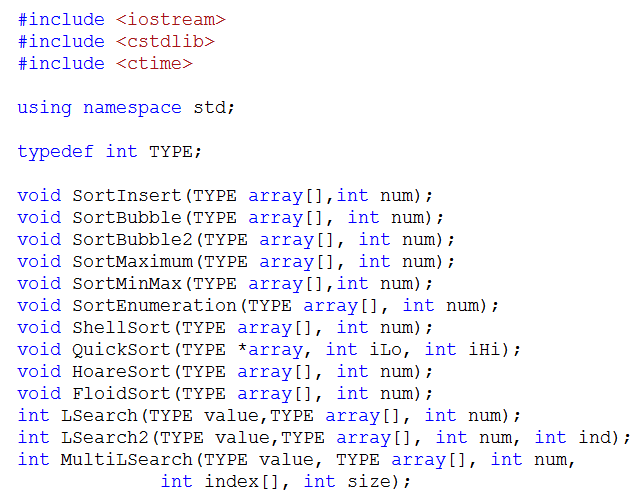
**Задание 3.** Организуйте сортировку элементов массива по любому полю любым алгоритмом.

**Задание 4.** Организуйте медленный поиск записей в Вашей структуре данных (или быстрый поиск в предварительно выгруженном и отсортированном динамическом массиве) по любому полю.

**Задание 5.** Добавьте написанные Вами функции в курсовую работу.

# ДОПОЛНЕНИЕ

Для проверки корректности работы описанных выше функций необходимо в файл исходного кода добавить следующее содержимое:



Результат работы программы представлен на рисунках 6, 7, 8.

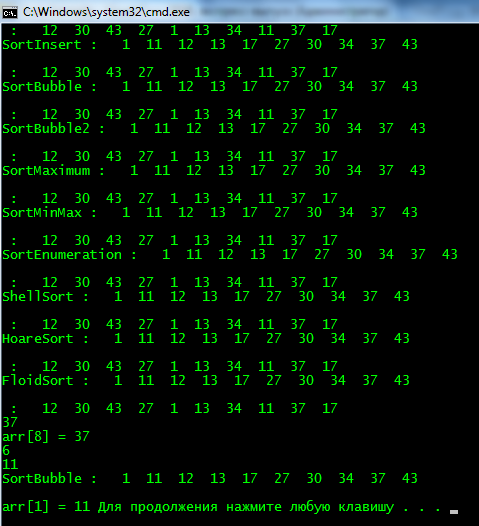


Рисунок 6 – Результат первого запуска тестирования функций

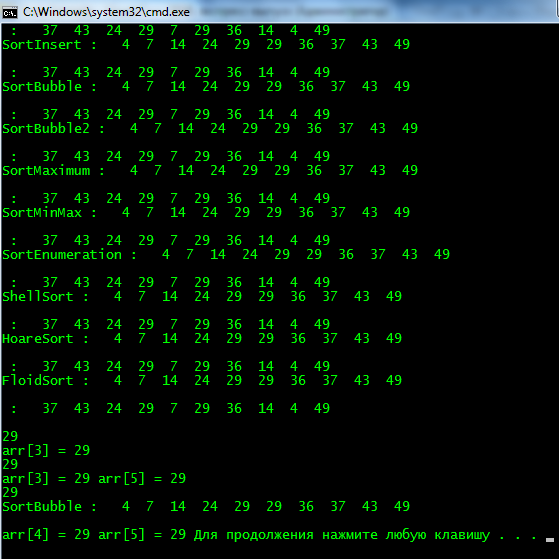


Рисунок 7 – Результат второго запуска тестирования функций

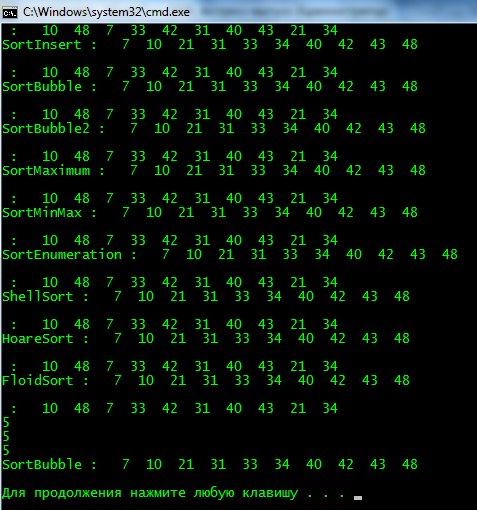


Рисунок 8 – Результат третьего запуска тестирования функций