**КУРСОВАЯ РАБОТА**

на тему: ***«Сравнительное исследование эффективности методов сортировки»***

**Задание**

Сравнительное исследование эффективности методов сортировки.

Базовая структура данных – вектор

Методы сортировки – метод Шелла, метод Флойда.

Примечание: Сравнение приводиться в виде графиков зависимостей количества сравнений и числа перестановок элементов от объёма данных.

**Введение**

В последние годы программирование для вычислительных машин выделилось в некоторую дисциплину, владение которой стало основным и ключевым моментом, определяющим успех многих инженерных проектов, а сама она превратилась в объект научного исследования. Из ремесла программирование перешло в разряд академических наук. Первый крупный вклад в ее становление сделали Э. Дейкстра и Ч. Хоар. Основное внимание в их работах уделяется построению и анализу программ, а более точно – структуре алгоритмов, представляемых текстом программы. Программы представляют собой конкретные, основанные на некотором реальном представлении и строении данных воплощения абстрактных алгоритмов.

Алгоритм – это формально описанная вычислительная процедура, получающая исходные данные, называемые его аргументом, и выдающая результат вычислений на выход. Алгоритмы строятся для решения тех или иных вычислительных задач. Формулировка задачи описывает, каким требованиям должно удовлетворять решение задачи, а алгоритм, решающий эту задачу, представляет собой метод, применение которого позволяет получить объект, удовлетворяющий этим требованиям. В настоящее время слово «алгоритм» ассоциируется, в основном, с компьютерами и другими средствами вычислительной техники, хотя разработка алгоритмов началась на заре развития математики, задолго до появления вычислительных машин. Формула Герона для вычисления корня квадратного из неотрицательного числа, процесс нахождения наибольшего общего делителя, выявление простых чисел из чисел натурального ряда («решето Эратосфена») всё это алгоритмы, которые можно реализовать посредством любого языка программирования и на любой современной ЭВМ. В последние полвека творческий процесс создания вычислительных алгоритмов стал наиболее интенсивным, это связано с возникновением, совершенствованием и развитием информационных технологий и всей компьютерной индустрии.

Для того чтобы разрабатывать собственные алгоритмы целесообразно сначала изучить уже существующие, методы анализа их параметров и эффективности. Тем более, что мировой опыт программирования насчитывает их великое множество. Рассматривая различные методы решения одной и той же задачи, полезно проанализировать, сколько вычислительных ресурсов они требуют (времени работы, памяти), и выбрать наиболее эффективный. Конечно, в этом случае нужно учитывать какая модель вычислительной системы используется для их выполнения: однопроцессорная ЭВМ или многопроцессорный комплекс. При анализе времени работы алгоритма следует учитывать ряд факторов, оказывающих определенное воздействие на результат: размерность исходных данных, структура данных в которую они организованы, их места хранения и размещения во время выполнения программы.

При сравнении методов сортировки, с точки зрения их эффективности, выполняют многократное тестирование разработанной программы на данных различной длины со всевозможными перестановками. Для каждого входного вектора значений размерности **n** определяется число сравнений sr и число обменов **m**, выполняемых с его координатами в процессе работы алгоритма. Полученные статистические данные отражают средние значения параметров, на основании которых можно сделать вывод об эффективности того или иного метода сортировки или поиска.

При сравнении способов организации некоторой логической структуры данных, например, списка или дерева, в процессе анализа учитывают, насколько быстро и просто выполняется её формирование посредством некоторой физической реализации, сколько времени и вычислительных ресурсов требуется для её модификации (вставки, удаления, перестроения), как быстро осуществляется поиск необходимой информации в такой структуре.

**Алгоритмы сортировки информации**

Хотя в словарях слово «сортировка» определяется как процесс разделение объектов по виду или сорту, программисты традиционно используют его в гораздо более узком смысле, обозначая им такую перестановку предметов, при которой они располагаются в порядке возрастания или убывания.

Под сортировкой массивов понимают процесс перестановки элементов массива в определенном порядке.

Цель сортировки – облегчить последующий поиск элементов в отсортированном массиве.

Методы сортировки важны при обработке данных, с ними связаны многие фундаментальные приемы построения алгоритмов.

Сортировки могут быть выполнены с использованием различных алгоритмов: как простых, так и усложненных (но более эффективных). Основное требование к методам сортировки: экономное использование памяти и быстродействие. Первое требование может быть выполнено, если переупорядочение элементов будет выполняться на том же месте. Хорошие алгоритмы сортировки требуют порядка n \*lognсравнений.



Простые методы сортировки можно разбить на три основных класса в зависимости от лежащего в их основе приема:

1. сортировка выбором;

2. сортировка обменом;

3. сортировка включением.

Простые методы сортировки требуют порядка n\*n сравнений элементов (ключей).

Простые методы сортировки.

Сортировка посредством простого выбора.

Сортировка основана на идее многократного выбора (находится сначала наибольший элемент, затем второй по величине и т.д.) и сводится к следующему:

1. найти элемент с наибольшим значением;

2. поменять значениями найденный элемент и последний;

3. уменьшить на единицу количество просматриваемых элементов;

4. если <количество элементов для следующего просмотра больше единицы> то <повторить пункты, начиная с 1-го>.

Алгоритм:

Цикл по количеству просматриваемых элементов {i:=n, n-1,…, 2}

Найти номер k максимального элемента среди a[1], a[2],…, a[i]

Поменять местами значения элементов a[k] и a[i]

Сортировка обменом (методом пузырька).

Сортировка обменом предусматривает систематический обмен значениями (местами) тех пар, в которых нарушается упорядоченность, до тех пор, пока таких пар не останется.

Алгоритм:

Цикл по количеству просмотров

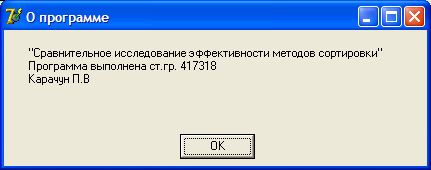
Цикл по количеству сравниваемых значений при очередном просмотре

Если < упорядоченность в паре нарушена > то <выполнить обмен значениями >.

Количество просмотров (повторений) во внешнем цикле равно n-1. Оно может быть уменьшено, если i– й шаг показал, что массив уже упорядочен (во внутреннем цикле не было перестановок).

Сортировка включением.

Сортировка основана на следующем: предполагается, что элементы a[1], a[2], …, a [i-1] упорядочены, a[i] вставляется на соответствующее место, не нарушая свойства упорядоченности. Для этого a[i] сравнивается по очереди с a [i-1], a [i-2], …до тех пор, пока не будет обнаружено, что элемент a[i] следует вставить между a[j], a [j-1] (j – номер элемента в a [1…i-1], за которым следует вставить a[i]).Тогда элементы a [j+1],…, a [i-1] сдвигаются на одну позицию вправо, а новая запись помещается в позицию j+1. Удобно совмещать сравнение и перемещение.



Можно уменьшить количество сравнений при организации внутреннего цикла. Для этого используется метод барьера: вставляемое значение помещается в начало массива на дополнительное 0-е место (a[0]:= a[i]), диапазон индексов расширяется.

**Метод Шелла**

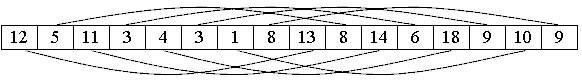
Для алгоритма сортировки, который каждый раз перемещает запись только на одну позицию, среднее время будет в лучшем случае пропорционально N2, потому что в процессе сортировки каждая запись должна пройти в среднем N позиций. Поэтому, если желательно получить метод, существенно превосходящий по скорости метод простых вставок, необходим механизм чтобы записи могли перемещаться большими скачками, а не короткими шажками.



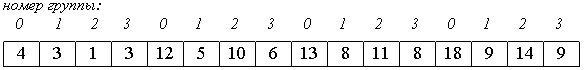
Такой метод предложен в 1959 году Дональдом Л. Шеллом [Donald L. Shell, CACM 2,7 (Java, 1959), 30–32] и известен во всем мире под именем своего автора. Пусть имеется массив записей R1, R2, …., R16.



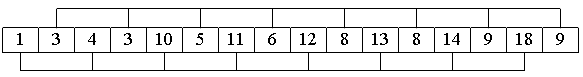
Делим 16 записей на 8 групп по две записи в каждой группе: (R1, R9), (R2, R10), …., (R8, R16). Сортируем выбранные пары записей в порядке, например, возрастания, т.е. если в паре (R2, R10): R2 > R10, то R2 и R10 меняем местами: R1, R10, R3, R4, R5, R6, R7, R8, R9, R2, R11, R12, R13, R14, R15, R16. То же самое выполняется и для других пар записей.



Это сортировка со смещением 8. Этот процесс называется первым проходом. Разделим теперь записи на четыре группы по четыре записи в каждой: (R1, R5, R9, R13), …, (R4, R8, R12, R16). Затем опять рассортируем каждую группу в отдельности. Это сортировка со смещением 4.



На третьем проходе отсортируем 2 группы по 8 записей: (R1, R3, R5, R7, R9, R11, R13, R15) и (R2, R4, R6, R8, R10, R12, R14, R16). Это сортировка со смещением 2.



Процесс завершается четвёртым проходом, во время которого сортируются все 16 записей. Это сортировка со смещением 1.



В каждой из промежуточных стадий сортировки участвуют либо сравнительно короткие массивы, либо уже сравнительно хорошо упорядоченные массивы, поэтому на каждом этапе можно пользоваться методом простых вставок. Метод сортировки Шелла ещё называется с «убывающим смещением», поскольку каждый проход характеризуется смещением h, таким, что сортируются записи, каждая из которых отстоит от предыдущей на h позиций. Последовательность значений смещений 8, 4, 2, 1 не следует считать неизменной, можно пользоваться любой последовательностью смещений, но последнее смещение должно быть равно 1.

Метод сортировки Шелла также известен под именем Shellsort и метода сортировки с «убывающим смещением», поскольку каждый проход характеризуется смещением h, таким, что сортируются записи, каждая из которых отстоит от предыдущей на h позиции. Последовательность значений смещений 8, 4, 2, 1 не следует считать незыблемой; можно пользоваться любой последовательностью ht-1, ht-2, …, h0. но последнее смещение h0 должно быть равно 1. Например, в таблице 2 продемонстрированная сортировка тех же данных со смещениями 7, 5, 3, 1. Как будет показано ниже, выбор значений смещений на последовательных проходах имеет весьма существенное значение для скорости сортировки.

# Сортировка Шелла.

Алгоритм D (сортировка Шелла). Запись R1,…, RN перекомпоновываются в том же адресном пространстве памяти. После завершения сортировки их ключи будут упорядочены: K1 …KN. Для управления процессом сортировки используется вспомогательная последовательность смещений ht-1, ht-2, …, h0, где h0 =1; правильно выбрав эти значения в последовательности, можно значительно сократить время сортировки. При t=1 этот алгоритм сводится к алгоритму S.



D1: [Цикл по s.] Выполнить шаг D2 при s =t –1, t-2, …, 0, после чего завершить процедуру.

D2: [Цикл по j.] Присвоить h ← hs и выполнить шаг от D3 до D6 при h < i < N. (Для сортировки элементов, отстоящих один от другого на h позиций, воспользуемся методом простых вставок и в результате получим Ki Ki+h для 1 i N-h. Шаги от D3 до D6, по существу, такие же, как соответственно от S2 до S5 в алгоритме S.)



D3. [Установка I, K, R.] Присвоить i ← j – h, K ← Ki, R ← Rj.

D4. [Сравнение K: Ki.] Если K Ki, то перейти к шагу D6.



D5. [Перемещение Ri, уменьшение i.] Присвоить Ri+h ← Ri, затем i ← i – h. Если I>0, то возвратиться к шагу D4.

D6. [Перемещение R на место Ri+h.] Присвоить Ri+h ← Ri.

Анализ Метода Шелла.

Для рационального выбора последовательности значений смещений сортировки ht-1,…, h0 для алгоритма D нужно проанализировать время выполнения как функцию от этих смещений. Такой анализ приводит к постановки очень красивых, но еще не до конца решенных математических задач; никому до сих пор не удалось найти наилучшею возможную последовательность смещений для больших N. Тем не менее известно довольно много интересных свойств сортировки методом Шелла с убывающим смещением.

**Метод Флойда**

Данный вид сортировки не рекомендуется для небольшого числа элементов, как, скажем, в нашем программном обеспечении. Однако для большого количества элементов пирамидальная сортировка оказывается очень эффективной, и чем больше число элементов, тем эффективнее.

Пирамидальная сортировка требует N∙Log2N шагов даже в худшем случае. Такие отличные характеристики для худшего случая – одно из самых выгодных качеств пирамидальной сортировки.

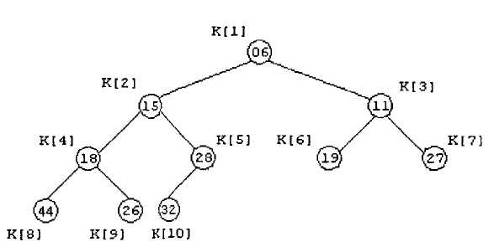
Но в принципе для данного вида сортировки, видимо, больше всего подходят случаи, когда элементы более или менее рассортированы в обратном порядке, т.е. для нее характерно неестественное поведение. Очевидно, что при обратном порядке фаза построения пирамиды не требует никаких пересылок.

**Пирамида**определяется как некоторая последовательность ключей

**K[L],…, K[R],** такая, что

**K[i] ≤ K[2i] & K[i] ≤ K [2i + 1], (1)**

для всякого i = L,…, R/2. Если имеется массив К[1], К[2],…, К[R], который индексируется от 1, то этот массив можно представить в виде двоичного дерева. Пример такого представления при R=10 показан на рисунке 2.

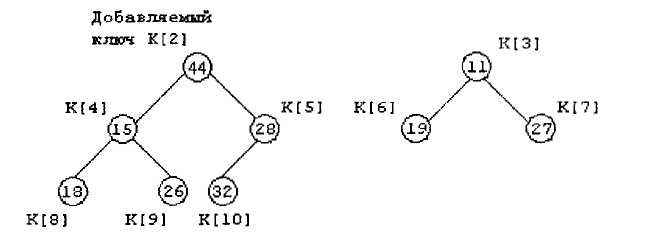


# Рис. 2 –Массив ключей, представленный в виде двоичного дерева

Дерево, изображенное на рисунке 2, представляет собой пирамиду, поскольку для каждого i = 1, 2,…, R/2 выполняется условие (1). Очевидно, последовательность элементов с индексами i = R/2+1, R/2+2,…., R (листьев двоичного дерева), является пирамидой, поскольку для этих индексов в пирамиде нет сыновей.

Способ построения пирамиды «на том же месте» был предложен Р. Флойдом. В нем используется процедура *просеивания* (sift), которую рассмотрим на следующем примере.

Предположим, что дана пирамида с элементами К[3], К[4],…, К[10] нужно добавить новый элемент К[2] для того, чтобы сформировать расширенную пирамиду К[2], К[3], К[4],…, К[10]. Возьмем, например, исходную пирамиду К[3],…, К[10], покачанную на рисунке 3, и расширим эту пирамиду «влево», добавив элемент К[2] =44.

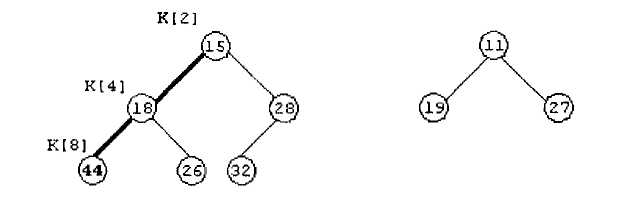


# Рис. 3 – Пирамида, в которую добавляется ключ К[2]=44

Добавляемый ключ К[2] просеивается в пирамиду: его значение сравнивается с ключами узлов-сыновей, т.е. со значениями 15 и 28. Если бы оба ключа-сына были больше, чем просеиваемый ключ, то последний остался бы на месте, и просеивание было бы завершено. В нашем случае оба ключа-сына меньше, чем 44, следовательно, вставляемый ключ меняется местами с наименьшим ключом в этой паре, т.е. с ключом 15. Ключ 44 записываетсяв элемент К[4], а ключ 15 – в элемент К[2]. Просеивание продолжается, поскольку ключи-сыновья нового элемента К[4] оказываются меньше его – происходит обмен ключей 44 и 18. В результате получаем новую пирамиду, показанную на рисунке 4.

В нашем примере получалось так, что оба ключа-сына просеиваемого элемента оказывались меньше его. Это не обязательно: для инициализации обмена достаточно того, чтобы оказался меньше хотя бы один сыновей ключ, с которым и производится обмен.

Просеивание элемента завершается при выполнении любого из двух условий: либо у него не оказывается потомков в пирамиде, либо значение его ключа не превышает значений ключей обоих сыновей.



# Рис. 4 – Просеивание ключа 44 в пирамиду

Алгоритм просеивания в пирамиду допускает рекурсивную формулировку:

1. просеивание элемента с индексом temp,
2. при выполнении условий остановки – выход,
3. определение индекса q элемента, с которым выполняется обмен,
4. обмен элементов с индексами temp и q,
5. tmp:= q,
6. перейти к п. 1.

Этот алгоритм в применении к нашему массиву а реализован в этом коде подпрограммы, который выполняет просеивания в пирамиду с максимальным индексом R:

begin

k:=2\*t;

if k>i then t:=0

else

begin

if k<i then

if Arr[k]>Arr [k-1] then inc(k); Kol\_sravFloud:= Kol\_sravFloud +1;

if Arr [t-1]>=Arr [k-1] then begin t:=0; Kol\_sravFloud:= Kol\_sravFloud +1

end else

begin

Kol\_sravFloud:= Kol\_sravFloud +1;

Tmp:=Arr [k-1];

Arr [k-1]:=Arr [t-1];

Arr [t-1]:=Tmp; // Переустановка Элементов

t:=k;

Kol\_PerFloud:= Kol\_PerFloud +1;

end;

Код учитывает индексацию вектора а от нуля.

Теперь рассмотрим процесс создания пирамиды из массива а[0], а[1],…, a[Highlndex]. Элементы этого массива индексируются от 0, а пирамида от 1. Ясно, что элементы a [N/2], a [N/2+1],…, a[Highlndex] уже образуют пирамиду, поскольку не существует двух индексов i (i= N/2+1, N/2+2, …) и j, таких, что, j=2i (или j=2i+l). Эти элементы составляют последовательность, которую можно рассматривать как листья соответствующего двоичного дерева. Теперь пирамида расширяется влево: на каждом шаге добавляется новый элемент и при помощи просеивания помещается на соответствующее место. Этот процесс иллюстрируется следующим примером.

*Процесс построения пирамиды*

44 55 12 42 **94 18 06 67**

44 55 12 **42 94 18 06 67**

44 55 **06 42 94 18 12 67**

44 **42 06 55 94 18 12 67**

**06 42 12 55 94 18 44 67**

Примечание – жирным шрифтом отмечены ключи, образующие пирамиду на текущем шаге ее построения

Для того, чтобы получить не только частичную, но и полную упорядоченность элементов нужно проделать N сдвигающих шагов, причем после каждого шага на вершину дерева выталкивается очередной (наименьший элемент). Возникает вопрос, где хранить «всплывающие» верхние элементы? Существует такой выход: каждый раз брать последнюю компоненту пирамиды (скажем, это будет х), прятать верхний элемент на место х, а х посылать в начало пирамиды в качестве элемента а[0] и просеивать его в нужное место. В следующей таблице приводятся необходимые в этом случае шаги:

*Пример преобразования пирамиды в упорядоченную последовательность*

**06 42 12 55 94 18 44 67**

**12 42 18 55 94 67 44** 06

**18 42 44 55 94 67** 12 06

**42 55 44 67 94** 18 12 06

**44 55 94 67** 42 18 12 06

**55 67 94** 44 42 18 12 06

**67 94** 55 44 42 18 12 0б

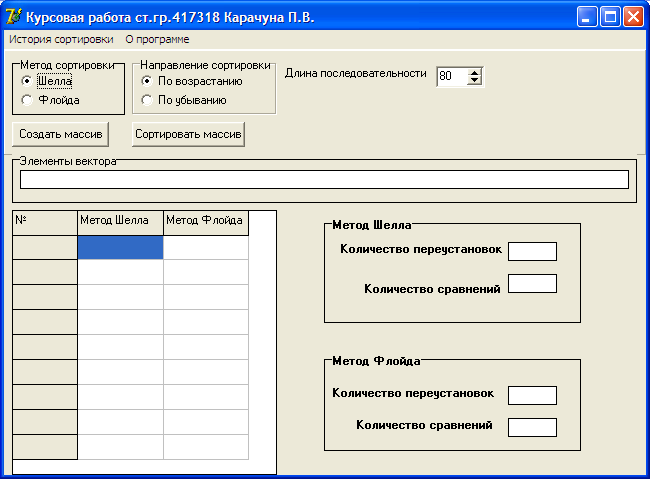
**94** 67 55 44 42 18 12 06 – Результат

Из примера сортировки видно, что на самом деле в результате мы получаем последовательность в обратном порядке. Но это легко можно исправить, изменив направление отношения.

**Состав проекта**

Данный проект состоит из трёх форм и трех модулей: модуля главной интерфейсной формы, модуля формы истории сортировки, модуля формы «О программе».

Главная форма имеет вид:

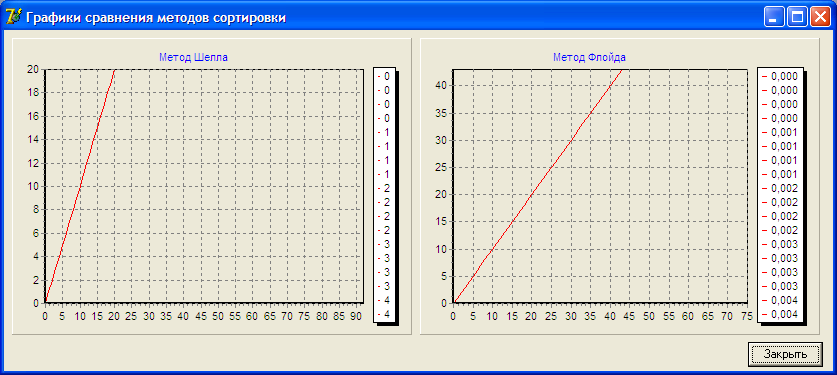


На форме рамположены компоненты: RadioGroup1 служащий для выбора вида сортировки и типа сортировки; SpinEdit1 – для изменения длины сортируемой последовательности; компонент NDgrid, являющийся окном вывода отсортированного массива; MainMenu1 для вызова окна истории сортировки и окна с информацией о создателях программы; Edit1 служит для вывода массива, также на форме имеется несколько компонентов типа TLabel служащих для пояснения назначения других компонентов.

Форма FormAbout имеет вид:

Данная форма служит для отображения информации о данной программе.

Данная форма содержит компонент Button1 для закрытия данной формы, компонент Label1 содержащий название программы и информацию о создателе данной программы. Форма Form2 имеет вид:



Данная форма служит для графиков

На форме располагаются компоненты Chart1 и Chart2, которые служат для отображения графиков.