МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
“МАТИ” — РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. К.Э. ЦИОЛКОВСКОГО

Кафедра “Моделирование систем и информационные технологии”

ИССЛЕДОВАНИЕ АЛГОРИТМОВ СОРТИРОВКИ В СРЕДЕ ОС LINUX

Методические указания к курсовой работе по предмету
“Алгоритмические языки и технология программирования”

Составитель В.В. Лидовский

Москва 2016

Владимир Викторович Лидовский

ИССЛЕДОВАНИЕ АЛГОРИТМОВ СОРТИРОВКИ В СРЕДЕ ОС LINUX

Методические указания к курсовой работе по предмету
“Алгоритмические языки и технология программирования”
3-е издание, исправленное

Редактор М. А. Соколова

Оригинал-макет подготовлен в пакетах Plain-TEX и XY-pic

Под. в печ. ?.?.2016 Объем 1,25 п.л. Тираж ? экз. Зак. ?

Ротапринт “МАТИ”—РГТУ, Берниковская наб. 14

ВВЕДЕНИЕ

Настоящие методические указания предназначены для обеспечения учебного процесса студентов второго курса дневной формы обучения специальности 220200 “АСОИиУ” при выполнении курсовой работы по предмету “Алгоритмические языки и технология программирования”.

Цели курсовой работы: изучить временные характеристики алгоритмов сортировки данных, приобрести навыки работы в среде ОС Linux.

1. СИСТЕМНЫЕ ТРЕБОВАНИЯ

Для выполнения курсовой работы необходимы следующие системные компоненты:

а) компьютер, работающий под управлением операционной системы Linux;
б) компилятор Free Pascal версии не ранее 2000 года;
в) программный модуль timer.o, обеспечивающий получение точных данных от таймера.

Вследствие того, что Linux является многозадачной ОС, для получения более точных данных от таймера желательно контрольный прогон программы выполнять при минимуме других одновременно выполняющихся программ.

2. ПОДГОТОВКА ИСХОДНЫХ ДАННЫХ

Данные, которые следует отсортировать, в общем случае представляют собой некоторую последовательность с заданной на момент сортировки длиной. Такую последовательность наиболее естественно реализовать в виде массива. Таким образом, не теряя общности, исследование свойств алгоритмов сортировки данных можно проводить на данных, организованных в виде одномерного массива с диапазоном индекса от 1 до количества сортируемых элементов. Последовательность данных считается отсортированной, если в ней каждый последующий элемент не больше (сортировка по убыванию) или не меньше (сортировка по возрастанию) предыдущего. Примеры, приведенные в указании, реализуют сортировку по убыванию. Для выявления характеристик зависимости времени сортировки от объема исходных данных потребуется произвести измерения времени сортировки на данных различного объема: 1000, 2000, 4000, 8000, 16000, 32000 и 64000 элементов в диапазоне от 1 до 70000 (тип longint). Кроме того, для выявления особенностей поведения исследуемых алгоритмов на данных разной степени упорядоченности, сортировка должна проводиться на данных следующих четырех типов:

1) “Упорядоченные” — элемент массива с меньшим индексом строго больше элемента с большим индексом;
2) “Обратного порядка” — элемент массива с меньшим индексом строго меньше элемента с большим индексом;
3) “Вырожденные” — массив заполняется случайным образом числами из диапазона от 1 до 12;
4) “Случайные” — массив заполняется случайными числами из диапазона от 1 до 70000.

Если BaseArray — это имя массива из Size элементов, то реализовать все необходимые способы его заполнения можно при помощи следующей последовательности операторов:

```
randomize;
case t of  (* t задает тип заполнения *)
  1:begin  (* Упорядоченные *)
    BaseArray[1] := MaxNumber; (* MaxNumber - константа 70000 *)
    for i := 2 to Size do
      BaseArray[i] := BaseArray[i-1]-random(MaxNumber div Size)-1
  end;
  2:begin  (* Обратный порядок *)
    BaseArray[1] := 1;
    for i := 2 to Size do
      BaseArray[i] := BaseArray[i-1]+random(MaxNumber div Size)+1
  end;
  3: for i := 1 to Size do  (* Вырожденные *)
      BaseArray[i] := random(12)+1;
  4: for i := 1 to Size do  (* Случайные *)
      BaseArray[i] := random(MaxNumber)+1
end.
```

После выполнения сортировки массива необходимо удостовериться в том, что сортировка выполнена правильно. Следующая последовательность операторов реализует алгоритм проверки правильности сортировки (i — вспомогательная целая переменная):

```
for i := 1 to Size-1 do
  if BaseArray[i] < BaseArray[i+1] then begin
    writeln('Error!');
    halt
  end.
```

3. ХАРАКТЕРИСТИКИ АЛГОРИТМОВ СОРТИРОВКИ

Алгоритмы сортировки используются в практически любой программной системе. Целью алгоритмов сортировки является упорядочение последовательности элементов данных. Поиск элемента в последовательности отсортированных данных занимает время, пропорциональное логарифму количества элементов в последовательности, а поиск элемента в последовательности неотсортированных данных занимает время, пропорциональное количеству элементов в последовательности, т.е. намного большее. Кроме того, отсортированные данные эстетически предпочтительнее неотсортированных. Существует множество различных методов сортировки данных. Однако любой алгоритм сортировки можно разбить на три основные части:

- сравнение, определяющее упорядоченность пары элементов;
- перестановку, меняющую местами пару элементов;
- собственно сортирующий алгоритм, который осуществляет сравнение и перестановку элементов данных до тех пор, пока все эти элементы не будут упорядочены.

Важнейшей характеристикой любого алгоритма сортировки является скорость его работы, которая определяется функциональной зависимостью среднего времени сортировки последовательностей элементов данных, заданной длины, от этой длины. Время сортировки будет пропорционально количеству сравнений и перестановок элементов данных в процессе их сортировки.

В последующих далее примерах процедура Exchange(i,j) меняет местами элементы сортируемого массива с индексами i и j.

3.1. Метод пузырька

Идея этого метода отражена в его названии: самые “легкие” элементы “всплывают” наверх, самые “тяжелые” — “тонут”. Последовательность из N элементов данных просматривается от начала до конца так, что стоящие рядом элементы меняются местами, если первый из них меньше (“легче”) второго. Таким образом, после такого просмотра самый “легкий” элемент “выталкивается” в конец последовательности. Если теперь повторить такой просмотр еще $N - 1$ раз, то, очевидно, что вся заданная последовательность окажется отсортированной. Этот алгоритм можно несколько оптимизировать двумя добавлениями:

1) просматривая на k-м проходе не весь массив, а только элементы с первого до $(N - k + 1)$-го;
2) повторяя просмотр не N раз, а лишь до тех пор, пока при очередном просмотре не произойдет ни одной перестановки.

Если BaseArray — это имя массива из Size элементов, то его сортировку методом пузырька можно реализовать при помощи следующей последовательности операторов (i и UBound — вспомогательные целые переменные, а EndFlag — вспомогательная логическая переменная):

```
UBound := Size-1;
repeat
  EndFlag := true;
  for i := 1 to UBound do
    if BaseArray[i] < BaseArray[i+1] then begin
      Exchange(i, i+1);
      EndFlag := false
    end;
  dec(UBound)
until EndFlag.
```

3.2. **Сортировка выбором**

При этом методе сортировки сначала находится самый большой элемент последовательности из N элементов. Если он больше 1-го, то они меняются местами. Затем эта операция повторяется, но для списка, состоящего из элементов с 2-го по N-ый. Затем эта операция аналогичным образом повторяется опять — всего $N-1$ раз.

Если BaseArray — это имя массива из Size элементов, то его сортировку выбором можно реализовать при помощи следующей последовательности операторов (i, MaxIndex и LBound — вспомогательные целые переменные):

```
LBound := 1;
repeat
  MaxElem := BaseArray[LBound];
  MaxIndex := LBound;
  for i := LBound+1 to Size do
    if MaxElem < BaseArray[i] then begin
      MaxElem := BaseArray[i];
      MaxIndex := i
    end;
  if MaxIndex <> LBound then
    Exchange(LBound, MaxIndex);
  inc(LBound)
until LBound = Size.
```

3.3. **Метод Шелла**

Этот метод является модификацией метода пузырька. Основная его идея заключается в том, чтобы вначале устранить массовый беспорядок

в сортируемой последовательности, сравнивая далеко отстоящие друг от друга элементы. Интервал между сравниваемыми элементами постепенно уменьшают до единицы, т. е. на первом проходе гарантируется, что все элементы, расстояние между которыми $L_1 < N - 1$, упорядочиваются друг относительно друга, на втором то же гарантируется для элементов, расстояние между которыми $L_2 < L_1$ и т. д. до последнего k-го прохода, когда должно выполняться $L_k = 1$. Обычно расстояния L для сортировки Шелла берутся из приблизительного соотношения $L_k \leqslant 2L_{k-1}$ и $L_1 \leqslant N/2$, но лучше для расстояний L брать простые числа, ближайшие к L_k, выбираемым по описанной выше схеме.

Если BaseArray — это имя массива из Size элементов, то его сортировку методом Шелла можно реализовать при помощи следующей последовательности операторов (i, j и gap — вспомогательные целые переменные):

```
gap := Size div 2;
while gap > 0 do begin
  for i := gap to Size-1 do begin
    j := i-gap+1;
    while BaseArray[j] < BaseArray[j+gap] do begin
      Exchange(j, j+gap);
      if j > gap then j := j-gap else break
    end
  end;
  gap := gap div 2
end.
```

3.4. **Быстрая сортировка (метод Хоара)**

Суть этого метода заключается в том, чтобы найти такой элемент сортируемой последовательности, который бы делил последовательность на две части так, что слева от него находились бы элементы не меньшие его, а справа — не большие. Поиск такого элемента можно организовать многими способами. Вот один из них.

Установим два индекса на 1-й (индекс i) и на последний (индекс j) элемент последовательности. Затем, пока элемент с индексом j меньше или равен элементу с индексом i, будем уменьшать j на 1. Если же элемент с индексом j больше элемента с индексом i, то меняем местами элементы с индексами i и j. Затем, пока элемент с индексом j меньше или равен элементу с индексом i, будем увеличивать i на 1. Если же элемент с индексом j больше элемента с индексом i, то меняем местами элементы с индексами i и j. Этот процесс продолжается до тех пор,

пока j не станет равным i. Элемент с индексом $i = j$ и есть искомый. Пример:

1)	10	7	28	49	31	25	17	3	10	43
	i									j
2)	43	7	28	49	31	25	17	3	10	10
	i									j
3)	43	7	28	49	31	25	17	3	10	10
		i								j
4)	43	10	28	49	31	25	17	3	10	7
		i								j
5)	43	10	28	49	31	25	17	3	10	7
		i					j			
6)	43	17	28	49	31	25	10	3	10	7
		i					j			
7)	43	17	28	49	31	25	10	3	10	7.
							i, j			

Далее ищем такой элемент для обоих, полученных в результате построенного разбиения последовательностей, и продолжаем процесс с вновь полученными разбиениями. Разбиение, содержащее один или два элемента, является конечным и далее не делится. Рассмотрим продолжение предыдущего примера:

8)	43	17	28	49	31	25	10	3	10	7.
	i_1					j_1		i_2		j_2

Доказано, что описанная процедура упорядочивает исходную последовательность за конечное число шагов.

Если BaseArray — это имя массива из Size элементов, то его сортировку методом Хоара можно реализовать при помощи следующей рекурсивной процедуры (параметр LBound должен быть равен 1 при запуске этой процедуры, а параметр UBound — Size):

```
Procedure QuickSort(LBound, UBound: word);
  var
    i, j: word;
  begin
    i := LBound;
    j := UBound;
    repeat
      while i <> j do
        if BaseArray[i] >= BaseArray[j] then
          dec(j);
        else begin
          Exchange(i, j);
          break
        end;
```

```
    while i <> j do
      if BaseArray[i] >= BaseArray[j] then
        inc(i);
      else begin
        Exchange(i, j);
        break
      end;
  until i = j;
  if i - 1 > LBound then
    QuickSort(LBound, i - 1);
  if j + 1 < UBound then
    QuickSort(j + 1, UBound)
end.
```

3.5. **Сортировка бинарным деревом**

Бинарным (двоичным) деревом называют упорядоченную структуру данных, в которой каждому элементу данных поставлены в соответствие до трех других элементов: левый и правый преемники и предшественник. Левый преемник должен быть больше, а правый — меньше или равен предшественнику. Единственный элемент, не имеющий предшественника, называется корнем дерева.

Если по исходной последовательности данных построить бинарное дерево, а затем вывести его элементы по определенным правилам обхода дерева, то полученная в результате последовательность окажется отсортированной.

Правила обхода дерева:

1) обход начинается с корня, предыдущим элементом считается верхний;
2) если предыдущий элемент — верхний, то если левый преемник существует, то переход к этому элементу, в противном случае вывод текущего элемента данных и если правый преемник существует, то переход к этому элементу, в противном случае переход к предшественнику;
3) если предыдущий элемент — левый, то вывод текущего элемента и если правый преемник существует, то переход к правому преемнику, в противном случае переход к предшественнику;
4) если предыдущий элемент — правый, то переход к предшественнику;
5) обход заканчивается после вывода последнего элемента, по счетчику.

Если BaseArray — это имя массива из Size элементов, то его сортировку бинарным деревом можно реализовать при помощи следующей

последовательности операторов (i, j и k — вспомогательные целые переменные; dir — вспомогательная переменная перечислимого типа tdirs = (left, right, up), определяющая направление от текущего узла к следующему; Tree — вспомогательный массив, состоящий из 64000 элементов (с индексами от 1 до 64000), каждый из которых — это запись

```
record
  data: longint;
  dirs: array [tdirs] of word
end,
```

где dirs — это индексы узлов-соседей, а data — данные:

```
(* инициализация бинарного дерева *)
for i := 1 to Size do
  for dir := left to right do
    Tree[i].dirs[dir] := 0;
(* построение бинарного дерева *)
Tree[1].data := BaseArray[1];
Tree[1].dirs[up] := 1;
for i := 2 to Size do begin
  j := 1;
  repeat   (* поиск узла для заполнения *)
    k := j;
    if Tree[j].data < BaseArray[i] then
      dir := left
    else
      dir := right;
    j := Tree[j].dirs[dir]
  until j = 0;
  Tree[i].data := BaseArray[i];
  Tree[i].dirs[up] := k;
  Tree[k].dirs[dir] := i
end;
(* вывод отсортированных данных - обход дерева *)
dir := up; (* направление на предшествующий узел *)
i := 1;    (* счетчик неупорядоченных элементов *)
j := 1;    (* индекс корня *)
repeat
  case dir of
    up:begin
         while Tree[j].dirs[left] <> 0 do
           j := Tree[j].dirs[left];
         BaseArray[i] := Tree[j].data;
```

```
          inc(i);
          if Tree[j].dirs[right] <> 0 then
            j := Tree[j].dirs[right]
          else begin
            if Tree[Tree[j].dirs[up]].dirs[left] = j then
              dir := left
            else
              dir := right;
            j := Tree[j].dirs[up]
          end
        end;
      left:begin
             BaseArray[i] := Tree[j].data;
             inc(i);
             if Tree[j].dirs[right] = 0 then begin
               if Tree[Tree[j].dirs[up]].dirs[left] <> j then
                 dir := right;
               j := Tree[j].dirs[up]
             end else begin
               j := Tree[j].dirs[right];
               dir := up
             end
           end;
      right:begin
              if Tree[Tree[j].dirs[up]].dirs[left] = j then
                dir := left;
              j := Tree[j].dirs[up]
            end
    end
  until i > Size.
```

Сортировка бинарным деревом — это нерекурсивная быстрая сортировка. При рекурсивной быстрой сортировке дерево автоматически строится и обходится в стеке.

3.6. **Сортировка массивом (хэширование)**

Сортировка массивом — это самый быстрый метод сортировки, не лишенный, однако, множества существенных недостатков. Суть метода заключается в заполнении вспомогательного массива, содержащего элементов несколько больше, чем исходная последовательность. Заполнение этого вспомогательного массива происходит так: вычисляются значения некоторой монотонной функции, называемой хэш-функция, на элементах сортируемой последовательности и эти значения считаются

индексами этих элементов в заполняемом массиве. Если же окажется, что подлежащий заполнению элемент вспомогательного массива уже занят, то происходит сдвиг соответствующих элементов этого массива так, чтобы образовалось "окно" для вносимого элемента и сохранялась упорядоченность между элементами. Функция должна выбираться так, чтобы ее значения лежали внутри диапазона индексов вспомогательного массива. Например, если известно, что сортируемая последовательность состоит из натуральных чисел от 1 до N, то в качестве искомой функции можно взять $f(n) = N - n + 1$, $n = 1, \ldots, N$. В общем случае, в качестве такой функции рекомендуется взять

$$f(n) = \left\lfloor \frac{A[n] - Min(A)}{Max(A) - Min(A)} (Size(B) - 1) \right\rfloor + 1,$$

где A — это исходная последовательность (массив), $Max(A)$ и $Min(A)$ максимальный и минимальный элементы A, B — это вспомогательный массив, а $Size(B)$ — это его размер. Эта монотонная (почти линейная) функция гарантирует, что ее значения на элементах сортируемого массива будут лежать в диапазоне от 1 до $Size(B)$. Она определена только при $Max(A) \neq Min(A)$. Если же $Max(A) = Min(A)$, то это означает, что массив состоит из одинаковых элементов, т.е. он отсортирован.

Если BaseArray — это имя массива из Size элементов, то его сортировку массивом можно реализовать при помощи следующей последовательности операторов (i, j, l, p, MinElem, MaxElem и UBound — вспомогательные целые переменные; k, m — вспомогательные целые переменные со знаком; AuxArray — вспомогательный массив, состоящий из 64000×factor элементов типа longint с индексами от 1 до 64000 × factor, factor = $Size(B)/Size(A)$ = [рекомендуется] = 1.4):

```
(* нахождение макс. и мин. элементов в BaseArray *)
MinELem := BaseArray[1];
MaxELem := BaseArray[1];
for i := 2 to Size do begin
  if MaxElem < BaseArray[i] then
    MaxELem := BaseArray[i];
  if MinElem > BaseArray[i] then
    MinELem := BaseArray[i]
end;
if MaxElem = MinElem then exit; (* массив уже отсортирован! *)
(* инициализация компонентов вспомогательного массива *)
UBound := round(Size*factor);
for i := 1 to UBound do
  AuxArray[i] := 0; (* нулевые элементы массива - свободны *)
```

```
for i := 1 to Size do begin
  j := (BaseArray[i]-MinElem)*(UBound-1) div (MaxElem-MinElem)+1;
  if AuxArray[j] = 0 then
    AuxArray[j] := BaseArray[i]
  else begin (* создание или поиск "окна" *)
    if AuxArray[j] > BaseArray[i] then begin
      while j > 1 do
        if AuxArray[j-1] > BaseArray[i] then
          dec(j)
        else
          break;
      m := -1
    end else begin
      while j < UBound do
        if (AuxArray[j+1] < BaseArray[i])
              and (AuxArray[j+1] <> 0) then
          inc(j)
        else
          break;
      m := 1
    end;
    k := 0;
    repeat
      if (k+j > 0) and (k+j <= UBound) then
        if AuxArray[k+j] = 0 then
          break;
      if k > 0 then k := -k else k := 1-k
    until false;
    l := j+k;
    if k > 0 then k := 1 else k := -1;
    j := j + (m+k) div 2;
    while l <> j do begin
      AuxArray[l] := AuxArray[l-k];
      l := l-k
    end;
    AuxArray[j] := BaseArray[i]
  end
end;
(* заполнение исходного массива отсортированными
   данными вспомогательного *)
j := 1;
```

```
for i := UBound downto 1 do
  if AuxArray[i] <> 0 then begin
    BaseArray[j] := AuxArray[i];
    inc(j)
  end.
```

4. ПОРЯДОК ВЫПОЛНЕНИЯ И ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ

4.1. Порядок выполнения курсовой работы

4.1.1. Требования к структуре программы

Программа должна содержать шесть основных подпрограмм:

1) заполнение массива;
2) сортировка по первому алгоритму;
3) сортировка по второму алгоритму;
4) проверка правильности сортировки;
5) инициализация таймера;
6) получение текущего значения таймера.

Алгоритм исследовательской программы можно описать следующим образом:

1) выбирается размер и тип заполнения сортируемого массива;
2) заполнение массива;
3) сохранение копии массива;
4) инициализация таймера;
5) выполнение сортировки по 1-му алгоритму;
6) получение времени сортировки по 1-му алгоритму;
7) проверка правильности сортировки;
8) заполнение сортируемого массива копией, сохраненной на шаге 3;
9) инициализация таймера;
10) выполнение сортировки по 2-му алгоритму;
11) получение времени сортировки по 2-му алгоритму;
12) проверка правильности сортировки;
13) если есть еще неисследованная комбинация размера и типа заполнения массива, то переход на шаг 1;
14) вывод результатов.

Процедура init_timer производит инициализацию (обнуление) таймера. Функция get_timer возвращает количество микросекунд, прошедших с момента последнего обнуления таймера. Эти подпрограммы должны быть описаны следующим образом:

```
{$L timer.o}
Procedure init_timer; cdecl; external;
```

`Function get_timer: longint; cdecl; external;`
`{$LinkLib c}`.

4.1.2 **Порядок анализа результатов**

Нарисовать таблицу, отражающую результаты прогонов программы на данных всех размеров и типов (см. Приложение). Построить график зависимости времени сортировки от размера сортируемого массива, заполняемого случайным образом. В случае, если время счета для исследуемых алгоритмов различается очень сильно, то необходимо строить график, использующий логарифмическую шкалу для оси ординат, т.е. эта ось градуируется числами (0.1,) 1, 10, 100, 1000 и т.д., находящимися на равном расстоянии друг от друга (см. Приложение).

Требуется установить характер зависимости времени сортировки от объема входных данных по каждому из четырех типов заполнения. Как правило, эта зависимость приблизительно может быть описана одной из следующих формул: $t(N) \sim N^2$, $t(N) \sim N$, $t(N) \sim N \ln N$, $t(N) \sim N \ln^2 N$, где N — это размер сортируемого массива, $t(N)$ — время сортировки.

На основании полученных результатов требуется оценить сравниваемые алгоритмы сортировки, выявить их области применения, сильные и слабые стороны.

4.2. **Отчет по курсовой работе**

Отчет по работе выполняется на отдельных бланках или листах и должен содержать:

1) цель исследования;
2) краткую характеристику исследуемых алгоритмов сортировки;
3) листинг исследовательской программы;
4) описание исследовательской программы (описание назначения и параметров подпрограмм, описание назначения глобальных переменных, блок-схема программы);
5) таблицу и график результатов исследования;
6) анализ результатов эксперимента;
7) выводы.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назначение алгоритмов сортировки.
2. Характеристики алгоритмов сортировки.
3. Методика расчета скорости работы алгоритмов сортировки.
4. Описание метода пузырька, его достоинств и недостатков.
5. Описание сортировки выбором, ее достоинств и недостатков.
6. Описание метода Шелла, его достоинств и недостатков.

7. Описание метода Хоара (быстрой сортировки), его достоинств и недостатков.
8. Описание метода сортировки бинарным деревом, его достоинств и недостатков.
9. Описание сортировки массивом, ее достоинств и недостатков.

6. ВАРИАНТЫ РАБОТ

Курсовая работа по теме “Изучение характеристик алгоритмов сортировки данных” имеет 15 вариантов заданий, отличающихся друг от друга исследуемыми алгоритмами сортировки:

Вариант	*1-й алгоритм*	*2-й алгоритм*
1	Метод пузырька	Сортировка выбором
2	Метод пузырька	Метод Шелла
3	Метод пузырька	Сортировка бин. деревом
4	Метод пузырька	Сортировка массивом
5	Метод пузырька	Быстрая сортировка
6	Сортировка выбором	Метод Шелла
7	Сортировка выбором	Сортировка бин. деревом
8	Сортировка выбором	Сортировка массивом
9	Сортировка выбором	Быстрая сортировка
10	Метод Шелла	Сортировка бин. деревом
11	Метод Шелла	Сортировка массивом
12	Метод Шелла	Быстрая сортировка
13	Сортировка бин. деревом	Сортировка массивом
14	Сортировка бин. деревом	Быстрая сортировка
15	Сортировка массивом	Быстрая сортировка

ЛИТЕРАТУРА

1. Бородин Ю.С., Вальвачев А.Н., Кузьмич А.И. *Паскаль для персональных компьютеров* — Минск: Вышэйшая школа, БФ ГИТМП “НИКА”, 1991.
2. Зуев Е.А. *Язык программирования Turbo Pascal 6.0* — М.: Унитех, 1992.
3. Кнут Д. *Искусство программирования для ЭВМ* — М.: Мир, 1978.
4. Курковский С. *Алгоритмы сортировки* //Монитор 6–7/92.
5. Петерсен Р. *Linux: полное руководство* — Киев: “Ирина” BHV, 2000.

ПРИЛОЖЕНИЯ

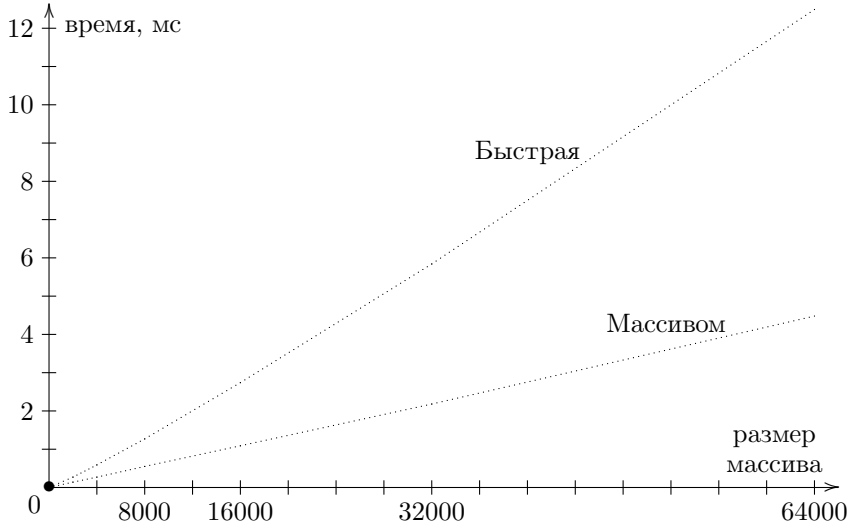
ТАБЛИЦА

Результатов исследования: зависимости времени (в мс) сортировки массива от его размеров и способа заполнения

Алгоритм	Быстрая сортировка/Сортировка массивом			
Тип / Размер	Упорядоченные	Обратный порядок	Вырожденные	Случайные
1000	2.10/0.06	2.09/0.06	0.22/0.42	0.12/0.07
2000	8.30/0.12	8.26/0.12	0.73/1.55	0.27/0.14
4000	33.06/0.22	32.93/0.22	2.78/5.98	0.59/0.27
8000	131.97/0.44	131.47/0.45	10.17/23.08	1.27/0.55
16000	527.54/0.84	525.51/0.85	40.94/91.13	2.74/1.09
32000	2117.28/1.57	2103.18/1.58	153.31/363.64	5.84/2.18
64000	8482.79/2.92	8421.29/2.91	673.89/1457.35	12.49/4.48
N	$\sim N^2/N$	$\sim N^2/N$	$\sim N^2/N^2$	$\sim N \ln N/N$

ГРАФИК

Зависимости времени сортировки от размера сортируемого массива, заполненного случайным образом



ОГЛАВЛЕНИЕ

стр.

Введение . . . 3
1. Системные требования . . . 3
2. Подготовка исходных данных . . . 3
3. Характеристики алгоритмов сортировки . . . 5
3.1. Метод пузырька . . . 5
3.2. Сортировка выбором . . . 6
3.3. Метод Шелла . . . 6
3.4. Быстрая сортировка (метод Хоара) . . . 7
3.5. Сортировка бинарным деревом . . . 9
3.6. Сортировка массивом (хэширование) . . . 11
4. Порядок выполнения и отчет по курсовой работе . . . 14
4.1. Порядок выполнения курсовой работы . . . 14
4.2. Отчет по курсовой работе . . . 15
5. Контрольные вопросы . . . 15
6. Варианты работ . . . 16
Литература . . . 16
Приложения . . . 17