**1.2. Постановка задачи сортировки массивов**.

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться ***на том же месте*** т. е. методы, в которых элементы из массива *а* передаются в результирующий массив *b,* представляют существенно меньший интерес. Мы будем сначала классифицировать методы по их экономичности, т. е. по времени их работы. Хорошей мерой эффективности может быть *C* – число необходимых сравнений ключей и *M* – число пересылок (перестановок) элементов. Эти числа суть функции от *n* – числа сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка *n\*log n* сравнений, мы сначала разберем несколько простых и очевидных методов, их называют ***прямыми***, где требуется порядка *n*2 сравнений ключей. Начинать разбор с прямых методов, не трогая быстрых алгоритмов, нас заставляют такие причины:

1.      Прямые методы особенно удобны для объяснения характерных

черт основных принципов большинства сортировок.

2.      Программы этих методов легко понимать, и они коротки.

3.      Усложненные методы требуют большого числа операций, и

поэтому для достаточно малых *n* прямые методы оказываются быстрее, хотя при больших *n* их использовать, конечно, не следует.

Методы сортировки “ *на том же месте* “ можно разбить в соответствии с определяющими их принципами на три основные категории:

        Сортировки с помощью включения ( by insertion ).

        Сортировки с помощью выделения ( by selection ).

        Сортировка с помощью обменов ( by exchange).

Теперь мы исследуем эти принципы и сравним их. Все программы оперируют массивом *а.*

*Const n=<длина массива>*

*a: array[1..n] of integer;*

**2. Методы сортировки массивов**

**2.1. Простые методы сортировки массивов**

**2.1.1. Сортировка с помощью прямого включения**.

Такой метод широко используется при игре в карты. Элементы мысленно делятся на уже “готовую” последовательность  *а1, … , аi-1* и исходную последовательность. При каждом шаге, начиная с *I = 2* и увеличивая *i* каждый раз на единицу, из исходной последовательности извлекается *i*- й элементы и перекладывается в готовую последовательность, при этом он вставляется на нужное место.

В таблице 2.1. показан в качестве примера процесс сортировки с помощью включения восьми случайно выбранных чисел. Алгоритм этой сортировки таков:

*FOR I: = 2 TO n DO*

*x: = a[I];*

*включение x на соответствующее место среди а[1], … , a[I]*

***END***

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать *x* , т. е. *x* сравнивается с очередным элементом *aj* , а затем либо x вставляется на свободное место, либо *aj*сдвигается вправо и процесс “уходит” влево. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1.     Найден элемент *aj*  с ключом, меньшим чем ключ у *x*.

2.     Достигнут левый конец готовой последовательности

ПРОГРАММА 2.1. ССОРТИРОВКА С ПОМОЩЬЮ ПРЯМОГО ВКЛЮЧЕНИЯ.

PROGRAM SI;

VAR

I,J,N,X:INTEGER;

A:ARRAY[0..50] OF INTEGER;

BEGIN

WRITELN(‘Введите длину массива’);

READ(N);

WRITELN(‘Введите массив’);

FOR I:=1 TO N DO READ(A[I]);

FOR I:=2 TO N DO BEGIN

X:=A[I];

A[0]:=X;

J:=I;

WHILE X<A[J-1] DO BEGIN

A[J]:=A[J-1];

DEC(J)

END;

A[J]:=X

END;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

Такой типичный случай повторяющегося процесса с двумя условиями окончания позволяет нам воспользоваться хорошо известным приемом “барьера” (sentinel).

***Анализ метода прямого включения****.* Число сравнений ключей (*Ci*) при *i*- м просеивании самое большее равно *i-1*, самое меньшее – *1*; если предположить, что все перестановки из *n* ключей равновероятны, то среднее число сравнений – *I/2*. Число же пересылок *Mi* равно *Ci + 2* (включая барьер). Поэтому общее число сравнений и число пересылок таковы:

*Cmin = n-1*  (2.1.) *Mmin = 3\*(n-1)* (2.4.)

*Cave = (n2+n-2)/4* (2.2.) *Mave = (n2+9\*n-10)/4* (2.5.)

*Cmax = (n2+n-4)/4* (2.3.) *Mmax = (n2+3\*n-4)/2*  (2.6.)

Алгоритм с прямыми включениями можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественно остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка включения. Такой модифицированный алгоритм сортировки называется ***методом с двоичным включением*** ( ***binary insertion***).

ПРОГРАММА 2.2. СОРТИРОВКА МЕТОДОМ ДЕЛЕНИЯ ПОПОЛАМ.

PROGRAM BI;

VAR I,J,M,L,R,X,N:INTEGER;

A:ARRAY[0..50] OF INTEGER;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

FOR I:=2 TO N DO BEGIN

X:=A[I];L:=1;R:=I;

WHILE L<R DO BEGIN

M:=(L+R) DIV 2;

IF A[M]<=X THEN L:=M+1 ELSE R:=M

END;

FOR J:=I DOWNTO R+1 DO A[J]:=A[J-1];

A[R]:=X

END;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

***Анализ двоичного включения****.* Место включения обнаружено, если *L = R.* Таким образом, в конце поиска интервал должен быть единичной длины; значит, деление его пополам происходит *I\*log I* раз. Таким образом:

*C = Si: 1<=i<=n: [log I ]* (2.7.)

Аппроксимируем эту сумму интегралом

*Integral (1:n) log x dx = n\*(log n – C) + C* (2.8.)

Где *C = log e = 1/ln 2 = 1.4426…* . (2.9.)

**2.1.2.Сортирвка с помощью прямого выбора**

Этот прием основан на следующих принципах:

1.                  Выбирается элемент с наименьшим ключом.

2.                  Он меняется местами с первым элементом *а1*.

3.                  Затем этот процесс повторяется с оставшимися *n-1* элементами, *n-2* элементами и т.д. до тех пор, пока не останется один, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в таблице 2.1., приведен в таблице 2.2. Алгоритм формулируется так:

*FOR I:= 1 TO n-1 DO*

*Присвоить k индекс наименьшего из a[I] … a[n];*

*Поменять местами a[I] и a[k];*

*END*

Такой метод сортировки – его называют прямым выбором – в некотором смысле противоположен прямому включению. Полностью алгоритм прямого выбора приводится в программе 2.3.

ПРОГРАММА 2.3. СОРТИРВКА С ПОМОЩЬЮ ПРЯМОГО ВЫБОРА.

PROGRAM SS;

VAR I,J,R,X,N:INTEGER;

A:ARRAY[0..50] OF INTEGER;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

FOR I:=1 TO N-1 DO BEGIN

R:=I;

X:=A[I];

FOR J:=I+1 TO N DO IF A[J]<X THEN BEGIN

R:=J;

X:=A[R]

END;

A[R]:=A[I];

A[I]:=X

END;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

***Анализ прямого выбора.*** Число сравнений ключей (*С*), очевидно не зависит от начального порядка ключей. Для *С* мы имеем

*C = (n2 – n)/2*  (2.10.)

Число перестановок минимально: *Mmin = 3\*(n-1)*  (2.11.)

в случае изначально упорядоченных ключей и максимально

*Mmax = n2/4 + 3\*(n-1)*  (2.12.)

Определим *Мavg* . Для достаточно больших *n* мы можем игнорировать дробные составляющие и поэтому аппроксимировать среднее число присваиваний на *i*-м просмотре выражением

*Fi = ln i + g + 1* (2.13.)

где *g = 0.577216…* - константа Эйлера.

Среднее число пересылок *Mavg* в сортировке с выбором есть сумма *Fi* с *i* от *1* до *n*:

*Mavg = n\*(g + 1) + (Si: 1<=i<=n: ln i)* (2.14.)

Вновь аппроксимируя эту сумму дискретных членов интегралом

*Integral (1:n) ln x dx = x\*(ln x – 1) = n\*ln (n) – n + 1*  (2.15.)

Получаем приблизительное значение

*Mavg = n\*(ln (n) + g)* . (2.16.)

**2.1.3. Сортировка с помощью прямого обмена**.

Как и в методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рассматривать как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу (см. таб. 2.3.).

Таблица 2.3. Пример пузырьковой сортировки.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **I = 1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 44 | 06 | 06 | 06 | 06 | 06 | 06 | 06 |
| 55 | 44 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 55 | 44 | 18 | 18 | 18 | 18 | 18 |
| 42 | 12 | 55 | 44 | 42 | 42 | 42 | 42 |
| 94 | 42 | 18 | 55 | 44 | 44 | 44 | 44 |
| 18 | 94 | 42 | 42 | 55 | 55 | 55 | 55 |
| 06 | 18 | 94 | 67 | 67 | 67 | 67 | 67 |
| 67 | 67 | 67 | 94 | 94 | 94 | 94 | 94 |

Такой метод сортировки известен под именем ***“пузырьковая сортировка”***. Он представлен в программе 2.4.

ПРОГРАММА 2.4. ПУЗЫРЬКОВАЯ СОРТИРОВКА.

PROGRAM BS;

VAR I,J,X,N:INTEGER;

A:ARRAY[0..50] OF INTEGER;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

FOR I:=2 TO N DO

FOR J:=N DOWNTO I DO

IF A[J-1]>A[J] THEN BEGIN

X:=A[J-1];

A[J-1]:=A[J];

A[J]:=X

END;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

Улучшения этого алгоритма напрашиваются сами собой:

1.   Запоминать, были или не были перестановки в процессе

некоторого прохода.

2.   Запоминать не только сам факт, что обмен имел место, но и

положение (индекс) последнего обмена.

3.   Чередовать направление последовательных просмотров.

Получающийся при этом алгоритм мы назовем ***“шейкерной”*** сортировкой (***ShakerSoft***). Таблица 2.4. иллюстрирует сортировку этим способом.

Таблица 2.4. Пример шейкерной сортировки.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| L = 2 | 3 | 3 | 4 | 4 |
| R = 8 | 8 | 7 | 7 | 4 |
| Dir = |  |  |  |  |
| 44 | 06 | 06 | 06 | 06 |
| 55 | 44 | 44 | 12 | 12 |
| 12 | 55 | 12 | 44 | 18 |
| 42 | 12 | 42 | 18 | 42 |
| 94 | 42 | 55 | 42 | 44 |
| 18 | 94 | 18 | 55 | 55 |
| 06 | 18 | 67 | 67 | 67 |
| 67 | 67 | 94 | 94 | 94 |

***Анализ пузырьковой и шейкерной сортировок****.* Число сравнений в строго обменном алгоритме

*C = (n2 – n)/2*, (2.17.)

а минимальное, среднее и максимальное число перемещений элементов (присваиваний) равно соответственно

***M = 0, Mavg = 3\*(n2 – n)/2, Mmax = 3\*(n2 – n)/4* (2.18.)**

Анализ же улучшенных методов, особенно шейкерной сортировки довольно сложен. Минимальное число сравнений

*Cmin = n – 1*. Кнут считает, что улучшенной пузырьковой сортировки среднее число проходов пропорционально *n – k1n1/2*, а среднее число сравнений пропорционально *½(n2 – n(k2 +ln n))*.

ПРОГРАММА 2.5. ШЕЙКЕРНАЯ СОРТИРОВКА.

PROGRAM SS;

VAR

J,L,K,R,X,N,I:INTEGER;

A:ARRAY[0..50] OF INTEGER;

BEGIN

WRITELN('Введи длину массива’);

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

L:=2;

R:=N;

K:=N;

REPEAT

FOR J:=R DOWNTO L DO IF A[J-1]>A[J] THEN BEGIN

X:=A[J-1];

A[J-1]:=A[J];

A[J]:=X;

K:=J

END;

L:=K+1;

FOR J:=L TO R DO IF A[J-1]>A[J] THEN BEGIN

X:=A[J-1];

A[J-1]:=A[J];

A[J]:=X;

K:=J

END;

R:=K-1

UNTIL L>R;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

Фактически в пузырьковой сортировке нет ничего ценного, кроме ее привлекательного названия. Шейкерная же сортировка широко используется в тех случаях, когда известно, что элементы почти упорядочены – на практике это бывает весьма редко.

Далее мы рассмотрим три улучшенных метода: по одному для каждого из основных методов сортировки – включения, выбора и обмена.

**2.2. Улучшенные методы сортировки массивов**

**2.2.1.Метод Шелла**

В 1959 году Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения. Сам метод представлен на нашем стандартном примере в таблице 2.5. Сначала отдельно сортируются и группируются элементы, отстоящие друг от друга на расстояние 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов и каждая группа состоит из двух элементов. После первого прохода элементы перегруппировываются – теперь каждый элемент группы отстоит от другого на две позиции – и вновь сортируются. Это называется двойной сортировкой. На третьем подходе идет обычная или одинарная сортировка. На каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуют сравнительно немного перестановок.

Таблица2.5. Сортировка с помощью включений с уменьшающимися расстояниями.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 | |
| Четверная сортировка дает | | | | | | | | |
|  | 44 | 18 | 06 | 42 | 94 | 55 | 12 | 67 | |
| Двойная сортировка дает | | | | | | | | |
|  | 06 | 18 | 12 | 42 | 44 | 55 | 94 | 67 | |
| Одинарная сортировка дает | | | | | | | | |
|  | 06 | 12 | 18 | 42 | 44 | 55 | 67 | 94 | |
|  |  |  |  |  |  |  |  |  |  |

Приводимая программа не ориентирована на некоторую определенную последовательность расстояний. Все *t* расстояний обозначаются соответственно *h1, h2, … ,ht* , для них выполняются условия

*ht = 1, hi+1 < hi .*

**Описание массива при этом выглядит так**

*A: ARRAY [-h1..n] OF INTEGER*

Сам алгоритм для *t = 4* описывается в программе 2.6.

ПРОГРАММА 2.6. СОРТИРОВКА ШЕЛЛА..

PROGRAM SHELLS;

CONST T=4;

H: ARRAY[1..4] OF INTEGER = (15,7,3,1);

VAR

I,J,K,S,X,N,M:INTEGER;

A:ARRAY[-16..50] OF INTEGER;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

FOR M:=1 TO T DO BEGIN

K:=H[M];

S:=-K;

FOR I:=K+1 TO N DO BEGIN

X:=A[I];

J:=I-K;

IF S=0 THEN S:=-K;

INC(S);

A[S]:=X;

WHILE X<A[J] DO BEGIN

A[J+K]:=A[J];

J:=J-K

END;

A[J+K]:=X

END;

END;

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

***Анализ сортировки Шелла.***Нам не известно, какие расстояния дают наилучший результат. Но они не должны быть множителями один другого. Справедлива такая теорема: если *k*-отсортированную последовательность *i*-отсортировать, то она остается *k*-отсортированной. Кнут показывает, что имеет смысл использовать такую последовательность, в которой  *hk-1 = 3hk + 1*, *ht = 1* и *t = [log2 n] – 1.* (2.19.)

**2.2.2.Сортировка с помощью дерева**

Метод сортировки с помощью прямого выбора основан на повторяющихся поисках наименьшего ключа среди *n* элементов, среди *n-1* оставшихся элементов и т. д. Как же усовершенствовать упомянутый метод сортировки? Этого можно добиться, действуя согласно следующим этапам сортировки:

1.оставлять после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Проделав *n-1* сравнений, мы можем построить дерево выбора вроде представленного на рисунке 2.1.

2. спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент в самом низу, либо на элемент из соседний ветви в промежуточных вершинах (см. рисунки 2.2 и 2.3.).

Д. Уилльямсом был изобретен метод ***Heapsort***, в котором было получено существенное улучшение традиционных сортировок с

помощью деревьев. Пирамида определяется как последовательность ключей *a[L], a[L+1], … , a[R],* такая, что

*a[i] <= a[2i] и a[i] <= a[2i+1] для i = L…R/2* .

Р. Флойдом был предложен некий “лаконичный” способ построения пирамиды на ”*том же месте*”. Здесь *a[1] … a[n]* – некий массив, причем *a[m]…a[n] (m = [n DIV 2] + 1 )* уже образуют пирамиду, поскольку индексов i и j, удовлетворяющих соотношению  *j = 2i (или j = 2i + 1)*, просто не существует.

Эти элементы образуют как бы нижний слой соответствующего двоичного дерева (см. рисунок 2.4.), для них никакой упорядоченности не требуется. Теперь пирамида расширяется влево; каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент. Таблица 2.6. иллюстрирует этот процесс, а получающаяся пирамида показана на рисунке 2.4.

Таблица 2.6. Построение 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 |

Каждый раз будем брать последнюю компоненту пирамиды (скажем *х*), прятать верхний элемент пирамиды в освободившемся теперь месте, а *х* сдвигать в нужное место. В таблице 2.7. приведены необходимые в этом случае *n-1* шагов.

Таблица 2.7. Примеры процесса сортировки с помощью *Heapsort*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 06 |
| 94 | 67 | 55 | 44 | 42 | 18 | 12 | 06 |

Пример из таблицы 2.7. показывает, что получающийся порядок фактически является обратным. Однако это можно легко исправить, и мы получаем программу 2.7.

ПРОГРАММА 2.7. HEARSORT.

PROGRAM HS;

VAR I,X,L,N,R:INTEGER;

A:ARRAY[0..50] OF INTEGER;

PROCEDURE SIFT(L,R: INTEGER);

VAR

I,J,X: INTEGER;

BEGIN

I:=L;

J:=2\*L;

X:=A[L];

IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);

WHILE (J<=R)AND(X<A[J]) DO BEGIN

A[I]:=A[J];

A[J]:=X;

I:=J;

J:=2\*J;

IF (J<R)AND(A[J]<A[J+1]) THEN INC(J);

END

END;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

L:=(N DIV 2)+1;

R:=N;

WHILE L>1 DO BEGIN

DEC(L);

SIFT(L,N)

END;

WHILE R>1 DO BEGIN

X:=A[1];

A[1]:=A[R];

A[R]:=X;

DEC(R);

SIFT(1,R)

END;

WRITELN(‘Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

***Анализ Heapsort*.** *Heapsort* очень эффективна для большого числа элементов *n*; чем больше *n*, тем лучше она работает.

В худшем случае нужно *n/2* сдвигающих шагов, они сдвигают элементы на *log(n/2), log(n/2 – 1), … ,log(n – 1)* позиций (логарифм по [основанию 2] «обрубается» до следующего меньшего целого). Следовательно фаза сортировки будет *n – 1* сдвигов с самое большое *log(n-1), log(n-2), … , 1* перемещениями.

Можно сделать вывод, что даже в самом плохом из возможных случаев *Heapsort* потребуется n\*log n шагов. Великолепная производительность в таких случаях – одно из привлекательных свойств *Heapsort.*

**2.2.3. Сортировка с помощью разделения.**

Этот улучшенный метод сортировки основан на обмене. Это самый лучший из всех известных на данный момент методов сортировки массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар назвал этот метод ***быстрой сортировкой*** (***Quicksort***) . В *Quicksort* исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, что у нас есть *n* элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за *n/2* обменов, сначала поменять местами самый левый с самым правым, а затем последовательно сдвигаться с двух сторон. Это возможно в том случае, когда мы знаем, что порядок действительно обратный. Однако полученный при этом алгоритм может оказаться и не удачным, что, например, происходит в случае *n* идентичных ключей: для разделения нужно *n/2* обменов. Этих необязательных обменов можно избежать, если операторы просмотра заменить на такие:

*WHILE a[i] <= x DO i := i + 1 END;*

WHILE x <= a[i] DO j := j – 1 END

В этом случае *x* не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут к переходу через границы массива.

Наша цель – не только провести разделение на части исходного массива элементов, но и отсортировать его. Будем применять процесс разделения к получившимся двум частям, до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются в программе 2.8.

ПРОГРАММА 2.8. QUICKSORT.

PROGRAM QS;

VAR N,I:INTEGER;

A:ARRAY[0..50] OF INTEGER;

PROCEDURE SORT(L,R: INTEGER);

VAR

I,J,X,W: INTEGER;

BEGIN

I:=L;

J:=R;

X:=A[(L+R) DIV 2];

REPEAT

WHILE A[I]<X DO INC(I);

WHILE X<A[J] DO DEC(J);

IF I<=J THEN BEGIN

W:=A[I];

A[I]:=A[J];

A[J]:=W;

INC(I);

DEC(J)

END

UNTIL I>J;

IF L<J THEN SORT(L,J);

IF I<R THEN SORT(I,R);

END;

BEGIN

WRITELN('Введи длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ(A[I]);

SORT(1, N);

WRITELN('Результат:');

FOR I:=1 TO N DO WRITE(A[I],' ')

END.

***Анализ Quicksort***. Процесс разделения идет следующим образом: выбрав некоторое граничное значение *x*, мы затем проходим целиком по всему массиву. При этом выполняется точно *n* сравнений. Ожидаемое число обменов есть среднее этих ожидаемых значений для всех возможных границ *x*.

*M = [Sx:1 <= x <= n:(x-1)\*(n-(x-1))/n]/n*

*= [Su:0 <= u <= n-1: u\*(n-u)]/n2*

*= n\*(n-1)/2n-(2n2-3n+1)/6n = (n-1/n)/6*  (2.20.)

В том случае, если бы нам всегда удавалось выбирать в качестве границы медиану, то каждый процесс разделения расщеплял бы массив на две половинки, и для сортировки требовалось бы всего  *n\*log n* подходов. В результате общее число сравнений было бы равно *n\*log n*, а общее число обменов – *n\*log(n) /6*. Но вероятность этого составляет только *1/n*.

Главный из недостатков *Quicksort* – недостаточно высокая производительность при небольших *n*, впрочем этим грешат все усовершенствованные методы, одним из достоинств является то, что для обработки небольших частей в него можно легко включить какой-либо из прямых методов сортировки.

**2.2.4. Нахождение медианы и k-й статистики массива**

***Медианой***  для *n* элементов называется элемент, меньший (или равный) половине из *n* элементов и больший (или равный) другой половине из *n* элементов. Например, медиана для элементов *16 12 99 95 18 87 10* равна 18. Метод определения медианы заключается в том, чтобы отсортировать *n* элементов, а затем выбрать средний элемент. Прием отыскивания медианы можно легко обобщить и для поиска среди *n* элементов *k*-го наименьшего числа. В этом случае поиск медианы – просто частный случай *k = n/2*. Алгоритм, предложенный Ч. Хоаром, работает следующим образом: сначала применяется операция разделения из  *Quicksort* с  *L = 1* и  *R = n*, в качестве разделяющего значения *x* берется *a[k]*. В результате получаем индексы *i* и *j*, удовлетворяющие таким условиям:

*1.       a[h] < x для всех h < i* (2.21.)

*2.       a[h] > x для всех h > j* (2.22.)

*3.       i > j*  (2.23.)

При этом мы сталкиваемся с одним из таких трех случаев:

1.     Разделяющее значение *x* было слишком мало, и граница между

двумя частями лежит ниже нужной величины *k*. Процесс разделения повторяется для элементов *a[i] … a[R]* (рис. 2.5.).

2.     Выбранная граница  *x* была слишком большой. Операции

разделения следует повторить для элементов *a[L] … a[j]* (рис. 2.6.).

3.     *j < k < i*: элемент *a[k]* разделяет массив на две части в нужной

пропорции, следовательно, это то, что нужно (рис. 2.7.)

Процессы разделения повторяются до тех пор, пока не возникнет третий случай. Теперь получаем программу поиска *k*-го наибольшего элемента (см. программу 2.9.).

ПРОГРАММА 2.9. ПОИСК k-ГО НАИБОЛЬШЕГО ЭЛЕМЕНТА.

PROGRAM F;

VAR

L,R,I,J,N,W,X,K: INTEGER;

A: ARRAY [1..1000] OF INTEGER;

BEGIN

WRITELN('Введите длину массива');

READ(N);

WRITELN('Введи массив');

FOR I:=1 TO N DO READ (A[I]);

L:=1;R:=N;

WRITELN('Введи требуемый номер статистики');

READ(K);

WHILE L<R DO BEGIN

X:=A[K];I:=L;J:=R;

REPEAT

WHILE A[I]<X DO INC(I);

WHILE X<A[J] DO DEC(J);

IF I<=J THEN BEGIN

W:=A[I];

A[I]:=A[J];

A[J]:=W;

INC(I);

DEC(J)

END

UNTIL I>J;

IF J<K THEN L:=I;

IF K<I THEN R:=J;

END;

WRITELN('Результат = ',A[K])

END.

Если предположить, что каждое разделение в среднем разбивает часть, где находится желаемая величина пополам, то число требуемых сравнений равно

*n + n/2 + n/4 + … = 2n* . (2.24.)

**2.3.Сравнение методов сортировки массивов**

Заканчивая наш обзор методов сортировки массивов, мы попытаемся сравнить их эффективность. Как и раньше, *n* – число сортируемых элементов, а *C* и *M* соответственно число необходимых сравнений ключей и число обменов. Для всех прямых методов сортировки можно дать точные аналитические формулы. Они приводятся в таблице 2.8. Столбцы *Min, Avg, Max* определяют соответственно минимальное, усредненное и максимальное по всем *n!* перестановкам из *n* элементов значений. Для усовершенствованных методов нет простых и точных формул. Существенно, что в случае сортировки Шелла вычислительные затраты составляют *c\*n1,2* , а для *Heapsort* и *Quicksort* - *c\*n\*log n* , где *c* – соответствующие коэффициенты.

Таблица 2.8. Сравнение прямых методов сортировки.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Min | Avg | Max |
| Прямое  включение | C =  M = | n-1  2(n-1) | (n2+n-2)/4  (n2-9n-10)/4 | (n2-n)/2-1  (n2-3n-4)/2 |
| Прямой  Выбор | C =  M = | (n2-n)/2  3(n-1) | (n2-n)/2  n\*(ln n + 0.57) | (n2-n)/2  n2/4 +3(n-1) |
| Прямой  обмен | C =  M = | (n2-n)/2  0 | (n2-n)/2  (n2-n)\*0.75 | (n2-n)/2  (n2-n)\*1.5 |

Для практических целей полезно иметь некоторые экспериментальные данные, способные пролить свет на те коэффициенты *с*, которыми один метод отличается от другого. В таблице 2.9. собраны времена (в секундах) работы, обсуждавшихся выше методов сортировки. Три столбца содержат времена сортировки уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. В начале приводятся цифры для 256 элементов, а ниже – для 2048. Четко прослеживается отличие квадратичных (прямых) методов от логарифмических (усложненных). Кроме того, заслуживают внимания следующие особенности:

1. Улучшение двоичного включения по сравнению с прямым включением действительно почти ничего не дает, а в случае упорядоченного массива даже получается отрицательный эффект.

**2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых. Ее усовершенствованная версия, шейкерная сортировка, продолжает оставаться плохой по сравнению с прямым включением и прямым выбором (за исключением случая уже упорядоченного массива).**

3. *Quicksort* лучше в 2 – 3 раза, чем *Heapsort* . Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

Таблица 2.9. Время работы различных программ.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Упорядоченный** | **Случайный** | В обратном порядке |
| n = 256 | | | |
| StraightInsertion | 0.02 | 0.82 | 1.64 |
| BinaryInsertion | 0.12 | 0.70 | 1.30 |
| StraightSelection | 0.94 | 0.96 | 1.18 |
| BubbleSelection | 1.26 | 2.04 | 2.80 |
| ShakerSort | 0.02 | 1.66 | 2.92 |
| ShellSort | 0.10 | 0.24 | 0.28 |
| HeapSort | 0.20 | 0.20 | 0.20 |
| QuickSort | 0.08 | 0.12 | 0.08 |
| NonRecQuickSort | 0.08 | 0.12 | 0.08 |
| StraightMerge | 0.18 | 0.18 | 0.18 |
| n = 2048 | | | |
| StraightInsertion | 0.22 | 50.74 | 103.80 |
| BinaryInsertion | 1.16 | 37.66 | 76.06 |
| StraightSelection | 58.18 | 58.34 | 73.46 |
| BubbleSelection | 80.18 | 128.84 | 178.66 |
| ShakerSort | 0.16 | 104.44 | 187.36 |
| ShellSort | 0.80 | 7.08 | 12.34 |
| HeapSort | 2.32 | 2.22 | 2.12 |
| QuickSort | 0.72 | 1.22 | 0.76 |
| NonRecQuickSort | 0.72 | 1.32 | 0.80 |
| StraightMerge | 1.98 | 2.06 | 1.98 |

**Заключение.**

Закончили разработка лабораторного практикума по курсу “Теоретические основы информатики”. Вообще говоря, внедрение электронного лабораторного практикума предполагает совершенствование процесса преподавания, повышение его эффективности и качества, сокращение времени на изучение учебного материала, тиражирование передовых педагогических технологий.

В данном практикуме мы рассмотрели некоторые простые и улучшенные методы внутренней сортировки массивов: прямое включения, двоичное включение, прямой выбор, прямой обмен, метод Шелла, метод Heapsort, быструю сортировку (Quicksort), и т. д. Их алгоритмы представлены в виде программ, написанных на языке Pascal. Так же мы проанализировали эти методы, и выяснили, какие из них более эффективны и в каких случаях.

В дальнейшем возможна разработка аналогичных электронных методических пособий для обучения методам внешней сортировки массивов, а также для обучения алгоритмам на графах.