10. АЛГОРИТМИ СОРТИРОВКИ

10.1 Задача сортировки

Пусть есть последовательность a_0, a_1... a_n и функция сравнения, которая на любых двух элементах последовательности принимает одно из трех значений: меньше, больше или равно. Задача сортировки состоит в перестановке членов последовательности таким образом, чтобы выполнялось условие: $a_i <= a_{i+1}$, для всех i от 0 до n.

Возможна ситуация, когда элементы состоят из нескольких полей:


```
struct element
{
  field x;
  field y;
}
```

Если значение функции сравнения зависит только от поля x, то x называют ключом, по которому производится сортировка. На практике, в качестве x часто выступает число, а поле y хранит какие-либо данные, никак не влияющие на работу алгоритма.

Не существует некий "универсальный", наилучший алгоритм. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, работающий оптимальным образом.

Для того, чтобы обоснованно сделать такой выбор, рассмотрим параметры, по которым будет производиться оценка алгоритмов.

1. *Время* сортировки - основной параметр, характеризующий быстродействие алгоритма.
2. *Память* - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.
3. *Устойчивость* - устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из нескольких полей, как на рис.10.1, а сортировка происходит по одному из них, например, по x.

Взаимное расположение равных элементов с ключом 1 и дополнительными полями "a", "b", "c" осталось прежним: элемент с полем "a", затем - с "b", затем - с "c".

Пример работы неустойчивой сортировки.

Взаимное расположение равных элементов с ключом 1 и дополнительными полями "a", "b", "c" изменилось.

Рис. 10.1.Примеры работы устойчивой и неустойчивой сортировок

4. *Естественность поведения* - эффективность метода при обработке уже отсортированных, или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Еще одним важным свойством алгоритма является его сфера применения. Здесь основных позиций две:

- внутренние сортировки работают с данным в оперативной памяти с произвольным доступом;
- внешние сортировки упорядочивают информацию, расположенную на внешних носителях. Это накладывает некоторые дополнительные ограничения на алгоритм:
    - доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим
    - объем данных не позволяет им разместиться в ОЗУ

Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

Данный класс алгоритмов делится на два основных подкласса:

Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно сортируются на том же месте, без дополнительных затрат.

Внешняя сортировка оперирует с запоминающими устройствами большого объема, но с доступом не произвольным, а последовательным (сортировка файлов), т.е в данный момент 'видно' только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это приводит к специальным методам сортировки, обычно использующим дополнительное дисковое пространство.

10.1 Сортировка методом прямого включения.

Основная идея данного алгоритма заключается в том, что массив сортируемых объектов условно разбивается на две части: отсортированную и не отсортированную. На каждом шаге, начиная со второго элемента массива из исходной последовательности извлекается один элемент и перемещается в готовую (отсортированную) половину последовательности в соответствующую позицию так, чтобы упорядоченность массива не нарушилась. Демонстрация алгоритма сортировки методом прямого включения приведена на рисунке 10.2.

Пример 10.1. Программная реализация алгоритма сортировки методом прямого включения.

```
        void main()
{
        setlocale (LC_CTYPE,"rus");
        int arr [] =  {8,7,6,0,4,3,2,1};
        const int N = 8;
      //Вывод массива
        cout<< "Массив:\n";
        for (int i=0; i < N; i++)
                cout<< arr [i]<< " ";
        cout<<"\n";
        //сортировка массива
        for (int i=1; i< N; i++)
        {
                int tmp = arr [i];
                int j;
                for (j=i - 1; j >=0 && arr [j] > tmp; j--)
                        arr [j + 1] = arr [j]; //сдвиг вправо

                arr [j + 1] = tmp;
        }
        cout<< "\nОтсортированный массив:\n";   //Вывод массива
        for (int i=0; i < N; i++)
                cout<< arr [i]<< " ";
}
```

Условием завершения вложенного цикла по “j” будет либо ситуация, когда с **выделенным** элементом дошли до начала отсортированной последовательности, не найдя ни одного элемента, который бы был меньше, чем выделенный, а следовательно, новый элемент должен быть вставлен в самое начало ряда. Либо цикл будет завершен, когда будет найден такой элемент с номером “j”, который оказался меньше, чем выделенный, и, значит, после него в позицию “j+1” необходимо поместить выделенный элемент.

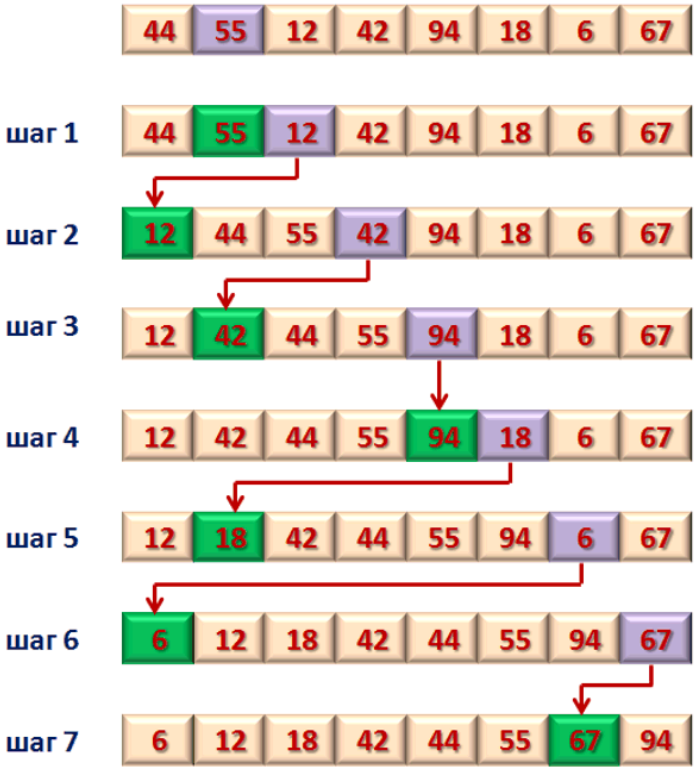


Рис. 10.2. Демонстрация алгоритма сортировки методом прямого включения

Анализ выполнения

Число сравнений ключей C_i при i-м просеивании составляет самое большое i-1, самое меньшее – 1. Если предположить, что все перестановки из n ключей равновероятны, то среднее число сравнений – i/2. Число пересылок

$$M_i = C_i+2.$$

Поэтому общее число сравнений и пересылок таковы:

$$Cmin=n-1;\ Mmin=3(n-1);$$

$$Ccp=(n^2+n-2)/4;\ Mcp=(n^2+9n-10)/4;$$

$$Cmax=(n^2+n-4)/4;\ Mmax=(n^2+3n-4)/2.$$

Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие оценки — когда элементы первоначально расположены в обратном порядке.

Резюме: сортировка методом прямого включения – не очень подходящий метод для компьютера, поскольку включение элемента с последующим сдвигом на одну позицию целой группы элементов неэффективно.

10.2 Сортировка методом прямого выбора.

Алгоритм сортировки прямым выбором в некотором смысле противоположен сортировке прямыми включениями.

При прямом включении на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готовой последовательности для нахождения места включения.

При прямом выборе для поиска одного элемента с наименьшим ключом просматриваются все элементы входной последовательности и найденный элемент помещается как очередной элемент в конец готовой последовательности.

Данный алгоритм сортировки основан на многократном выполнении следующего набора шагов:

1) в исходном массиве находится элемент с наименьшим значением;

2) найденный элемент меняется местами с самым первым элементом;

3) эти два шага повторяются многократно для оставшейся неупорядоченной части массива, начиная со второго, потом третьего и так далее до тех пор, пока не дойдем до конца массива и не отсортируем его целиком.

Демонстрация алгоритма сортировки методом прямого выбора приведена на рисунке 10.3.

Пример 10.2. Программная реализация алгоритма сортировки методом прямого выбора.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int arr [] =  {8,7,0,5,4,3,2,9};
	const int N = 8;
	cout<< "Массив:\n";
	for (int i=0; i < N; i++)
		cout<< arr [i]<< " ";
	cout<<"\n";
	//сортировка массива
	for (int i=0; i< N; i++)
	{
		int posMin = i;
		for (int j=i + 1; j < N; j++)
			if (arr [j] < arr[posMin])
				posMin = j;
		int tmp = arr [posMin];
		arr [posMin] = arr [i];
		arr [i] = tmp;
	}
	cout<< "\nОтсортированный массив:\n";
	for (int i=0; i < N; i++)
		cout<< arr [i]<< " ";
}
```

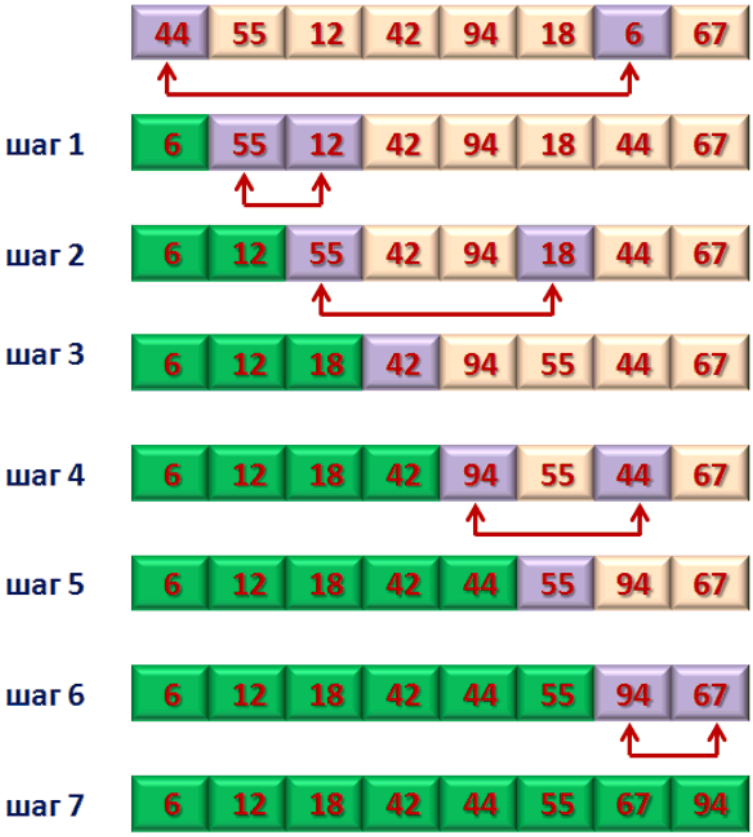


Рис. 10.3. Демонстрация алгоритма сортировки методом прямого выбора

Анализ алгоритма

Число сравнений ключей C не зависит от порядка ключей:

$$C=½(n^2-n).$$

Число перестановок минимально в случае изначально упорядоченных ключей

$$Mmin=3(n-1)$$

и максимально, если первоначально ключи располагаются в обратном порядке.

$$Mmax= n2/4 +3(n-1),$$

Среднее число пересылок

$$Mcp≈n(\ln n + g),$$

где **g = 0,577216...** — константа Эйлера.

Как правило, сортировка прямым выбором предпочтительнее алгоритма прямого включения, однако, если ключи в начале упорядочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

10.3 Пузырьковая сортировка.

Алгоритм сортировки прямым обменом основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут отсортированы все элементы. Как и в методе прямого выбора совершаются проходы по

массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива.

Если рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в банке с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу. Такой метод известен под именем «пузырьковая сортировка» (рис.10.4).

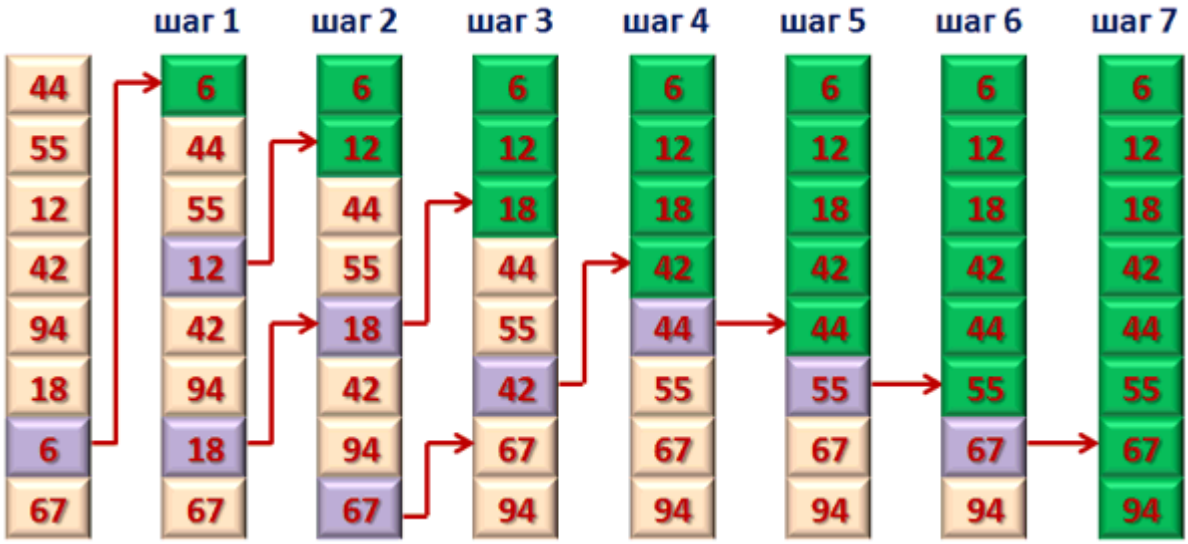


Рис. 10.4. Демонстрация алгоритма пузырьковой сортировки

Смысл пузырьковой перестановки заключается в сравнении двух рядом расположенных элементов и в случае необходимости перестановки их местами. После одного такого прохода на последней n-й позиции массива будет стоять максимальный элемент, если выполняется сортировка по возрастанию (минимальный элемент, если выполняется сортировка по убыванию). Поскольку максимальный (минимальный) элемент уже стоит на своей последней позиции, то второй проход по массиву выполняется до (n-1)-го элемента, и так далее.

Пример 10.3. Программная реализация алгоритма пузырьковой сортировки.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int arr [] =  {8,7,0,5,4,3,2,1};
	const int N = 8;
	//Вывод массива
	cout<< "Массив:\n";
	for (int i=0; i < N; i++)
		cout<< arr [i]<< " ";
	cout<<"\n";
```

```
        //сортировка массива
        for (int i=N-1; i > 0; --i)
                for (int j=0; j < i; ++j)
                        if (arr [j] < arr [j+1])
                        {
                                int tmp = arr [j];
                                arr [j] = arr [j+1];
                                arr [j+1] = tmp;
                        }
        cout<< "\nОтсортированный массив:\n";   //Вывод массива
        for (int i=0; i < N; i++)
                cout<< arr [i]<< " ";
}
```

10.4. Шейкер-сортировка

Шейкер-сортировка **я**вляется усовершенствованным методом пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства:

- если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.
- при движении от конца массива к началу минимальный элемент "всплывает" на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к модификациям в методе пузырьковой сортировки.

- От последней перестановки до конца (начала) массива находятся отсортированные элементы. Учитывая данный факт, просмотр осуществляется не до конца (начала) массива, а до конкретной позиции. Границы сортируемой части массива сдвигаются на 1 позицию на каждой итерации.
- Массив просматривается поочередно справа налево и слева направо.
- Просмотр массива осуществляется до тех пор, пока все элементы не встанут в порядке возрастания (убывания).
- Количество просмотров элементов массива определяется моментом упорядочивания его элементов.

Рассмотрим алгоритм Шейкер-сортировки на примере (рис.10.5).

Рис. 10.5. Демонстрация алгоритма Шейкер-сортировки

Пример 10.4. Программная реализация алгоритма Шейкер-сортировки

```
void main()
{
	SetConsoleOutputCP(1251);
	int arr [] =  {8,7,0,5,4,3,2,1};
	const int N = 8;
	//Вывод массива
	cout<< "Массив:\n";
	for (int i=0; i < N; i++)
		cout<< arr [i]<< " ";
	cout<<"\n";
	 int left= 0, right = N-1;
	 double t;
	 int flag = 1;  // флаг наличия перемещений
	 while((left < right) && flag > 0)
	 {
		flag = 0;
		for(int i=left; i<right; i++) //двигаемся слева направо
		 if(arr[i]>arr[i+1])
		{
		  t=arr[i];
		  arr[i]=arr[i+1];
		  arr[i+1]=t;
		  flag = 1;
		}
		 right--;
```

```
            for(int i=right; i>left; i--) //двигаемся справа налево
            if(arr[i-1]>arr[i])
             {
               t=arr[i];
               arr[i]=arr[i-1];
               arr[i-1]=t;
              flag = 1;

             }
            left++;
     }
    cout<<"\nОтсортированный массив:\n";    //Вывод массива
    for (int i=0; i < N; i++)
          cout<< arr [i]<< " ";
}
```

Каждое повторение цикла while() представляет собой шаг сортировки.

10.5 Сортировка включениями с убывающими приращениями

В 1959 г. Д. **Шеллом** было предложено усовершенствование сортировки с помощью прямого включения.

Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на 4 позиции. Такой процесс называется ***четвертной сортировкой***.

После первого прохода элементы перегруппировываются — теперь каждый элемент группы отстоит от другого на 2 позиции — и вновь сортируются (***двойная сортировка***).

На третьем проходе идет обычная сортировка (рис.10.6).

Кажется, что необходимость нескольких проходов сортировки, в каждом из которых участвуют все элементы, потребует большего количества машинных ресурсов, чем обычная сортировка. Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Такой метод в результате дает упорядоченный массив, и каждый проход от предыдущих только выигрывает (так как каждая i-сортировка объединяет две группы, уже отсортированные 2i-сортировкой).

Расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным. В самом плохом случае последний проход сделает всю работу.

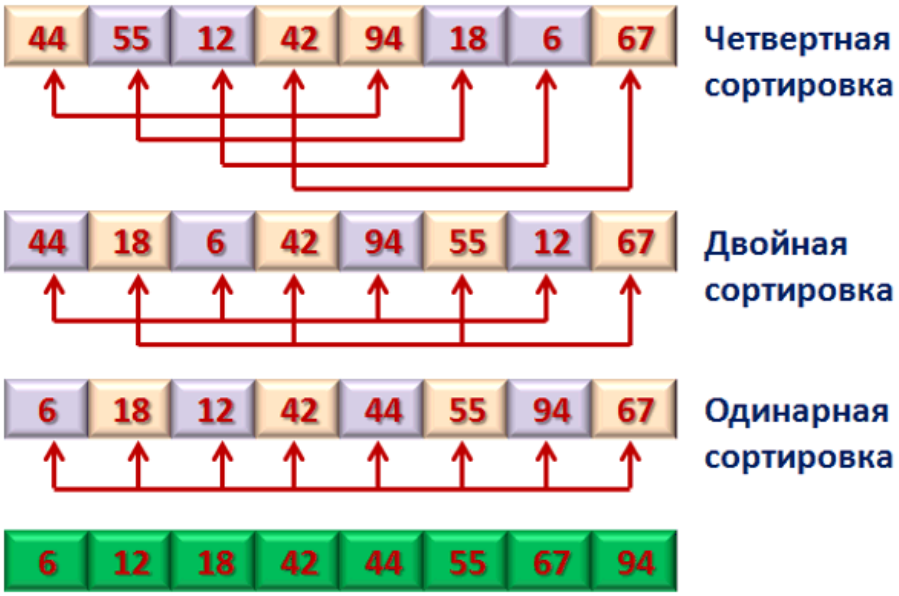


Рис. 10.6. Демонстрация алгоритма сортировки включениями с убывающими приращениями

Пример 10.5. Программная реализация алгоритма сортировки Шелла

```
void main()
{
SetConsoleOutputCP(1251);
        int arr [] =  {8,7,0,10,4,3,2,1};
        const int N = 8;
        int i, j, increment= 4, temp;
        //Вывод массива
        cout<< "Массив:\n";
        for (i=0; i < N; i++)
                cout<< arr [i]<< " ";
        cout<<"\n";
    while (increment > 0)
 {
        for (i=0; i < N; i++)
        {
                j = i;
                temp = arr[i];
                while((j>=increment) && (arr[j-increment]>temp))
                {
                        arr[j] = arr[j - increment];
                        j = j - increment;
                }
                 arr[j] = temp;
        }
         if (increment/2 != 0)
         increment = increment/2;
        else if (increment == 1)
```

```
                increment = 0;
                else
                        increment = 1;
  }
        cout<<"\nОтсортированный массив:\n";    //Вывод массива
        for (int i=0; i < N; i++)
                cout<< arr [i]<< " ";
}
```

Анализ алгоритма

Приводимая программа не ориентирована на некую определенную последовательность расстояний. Все **t** расстояний обозначаются соответственно $\mathbf{h_1, h_2, ..., h_t}$, для них выполняются условия

$$\mathbf{h_t=1;}$$

$$\mathbf{h_{i+1}<h_i.}$$

Каждая **h**-сортировка программируется как сортировка с помощью прямого включения.

В алгоритме не известно, какие расстояния дают наилучшие результаты.

Дональд Кнут рекомендует такую последовательность:

1, 3, 7, 15, 31, ...,

то есть

$$\mathbf{h_{k-1}=2h_k+1,}$$

$$\mathbf{h_t=1} \text{ и } \mathbf{t = [\log_2 n] - 1.}$$

10.6 Сортировка с помощью дерева

Сортировка с помощью дерева осуществляется на основе бинарного дерева поиска. ***Бинарное (двоичное) дерево поиска*** – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Для сортировки с помощью дерева исходная сортируемая последовательность представляется в виде структуры данных "дерево".

Например, исходная последовательность имеет вид:

4, 3, 5, 1, 7, 8, 6, 2

Корнем дерева будет начальный элемент последовательности. Далее все элементы, меньшие корневого, располагаются в левом поддереве, все элементы, большие корневого, располагаются в правом поддереве (рис.10.7). Причем это правило должно соблюдаться на каждом уровне.

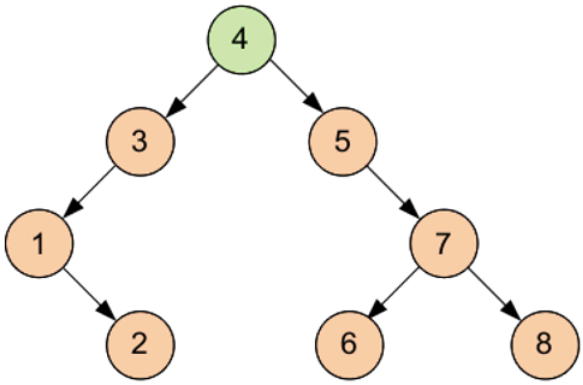


Рис. 10.7. Бинарное (двоичное) дерево поиска

После того, как все элементы размещены в структуре "дерево", необходимо вывести их, используя инфиксную форму обхода.

Пример 10.6. Программная реализация алгоритма сортировки с помощью дерева

```
#include <iostream>
using namespace std;// Структура - узел дерева
struct tnode
{
        int field;          // поле данных
        struct tnode *left;  // левый потомок
        struct tnode *right; // правый потомок
};
// Обход в инфиксной форме
void treeprint(tnode *tree)
{
         if (tree!=NULL)
        { //Пока не встретится пустой узел
        treeprint(tree->left); //Рекурсивная функция вывода левого поддерева
        cout << tree->field << " "; //Отображаем корень дерева
        treeprint(tree->right); //Рекурсивная функция вывода правого поддерева
        }
}
//                Добавление                узлов                в                дерево
struct tnode * addnode(int x, tnode *tree)
 {
   if (tree == NULL)
```

```
        { // Если дерева нет, то формируем корень
                tree =new tnode; //память под узел
                tree->field=x;   //поле данных
                tree->left = NULL;
                tree->right=NULL; //ветви инициализируем пустотой
         }
  else
  if (x < tree->field)
        {  //Если элемент x меньше корневого, уходим влево
                tree->left = addnode(x,tree->left); //Рекурсивно добавляем элемент
        } else
        { //иначе уходим вправо
                tree->right     =     addnode(x,tree->right); //Рекурсивно     добавляем
//элемент
        }
  return(tree);
}
//Освобождение памяти дерева
void freemem(tnode *tree)
{
  if(tree!=NULL)
{
    freemem(tree->left);
    freemem(tree->right);
    delete tree;  }
}
// Тестирование работы
int main()
 {
        struct tnode *root=0;
        system("chcp 1251");
        system("cls");
        int a;
        for (int i=0;i< 8;i++)
        {
                cout << "Введите узел " << i+1 <<": ";
                cin >> a;
                root = addnode(a,root);
        }
        treeprint(root);
        freemem(root);
        cin.get();  cin.get();
        return 0;
}
```

10.7. Пирамидальная сортировка

Метод пирамидальной сортировки, изобретенный Д. Уилльямсом, является улучшением традиционных сортировок с помощью дерева.
Пирамидой (***кучей***) называется двоичное дерево (рис.10.8) такое, что

a[i] ≤ a[2i+1];
a[i] ≤ a[2i+2].

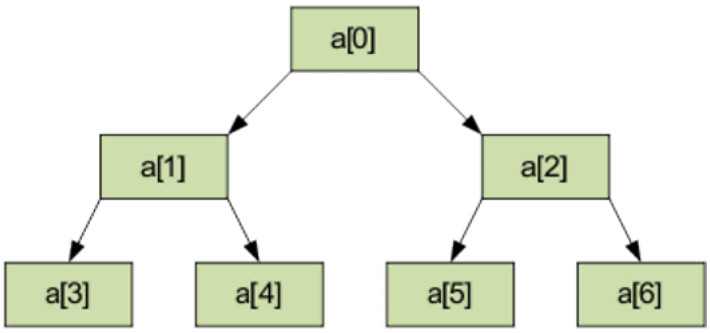


Рис. 10.8. Пирамида

a[0] — минимальный элемент пирамиды.

Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

Выполнение алгоритма разбивается на два этапа:
1 этап. Построение пирамиды. Определяем правую часть дерева, начиная с **n/2-1** (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираете путь через меньший элемент.
Например, массив для сортировки
24, 31, 15, 20, 52, 6

Расположим элементы в виде исходной пирамиды; номер элемента правой части (6/2-1)=2 - это элемент 15 (рис.10.9).

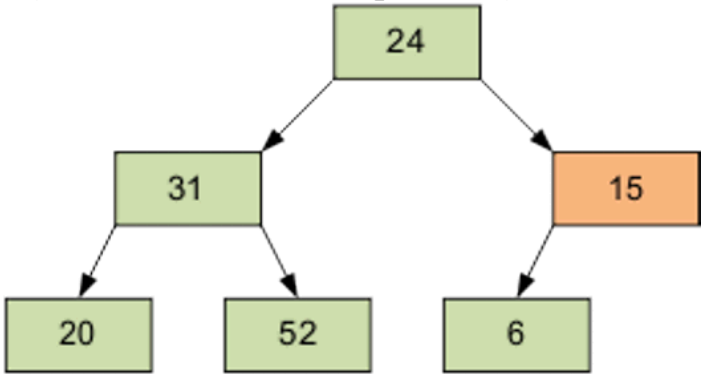


Рис. 10.9. Исходная пирамида

Результат просеивания элемента 15 через пирамиду (рис.10.10).

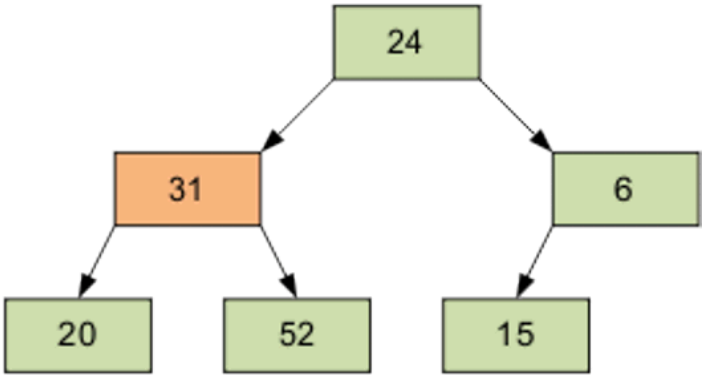


Рис. 10.10. Результат просеивания элемента 15

Следующий просеиваемый элемент – 1, равный 31 (рис.10.11).

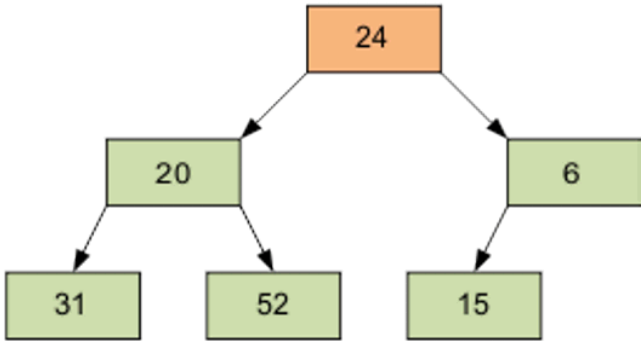


Рис. 10.11. Результат просеивания элемента 31

Затем – элемент 0, равный 24 (рис.10.12).

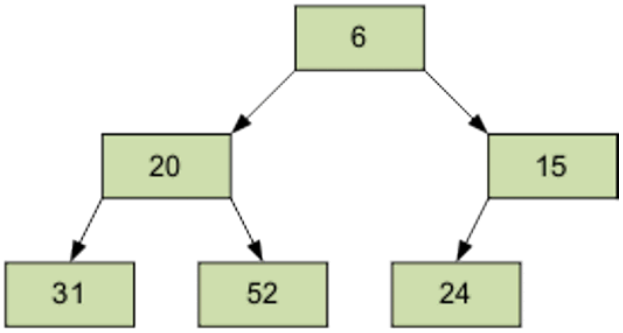


Рис. 10.12. Результат просеивания элемента 24

Разумеется, полученный массив еще не упорядочен. Однако процедура просеивания является основой для пирамидальной сортировки. В итоге просеивания наименьший элемент оказывается на вершине пирамиды.

2 этап. Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент

массива и текущий местами (рис.10.13). Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду (рис.10.14). Затем берем предпоследний элемент и т.д.

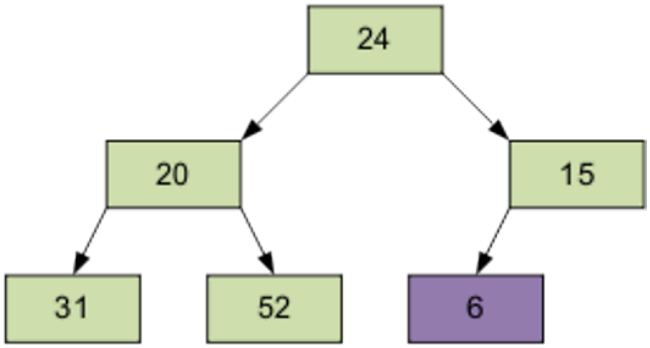


Рис. 10.13. Результат замены верхнего (наименьшего) элемента массива и текущего местами

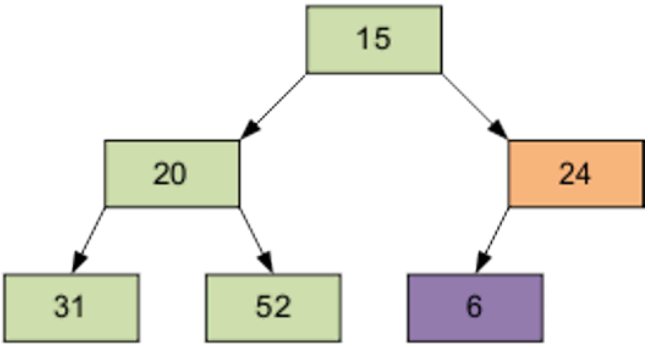
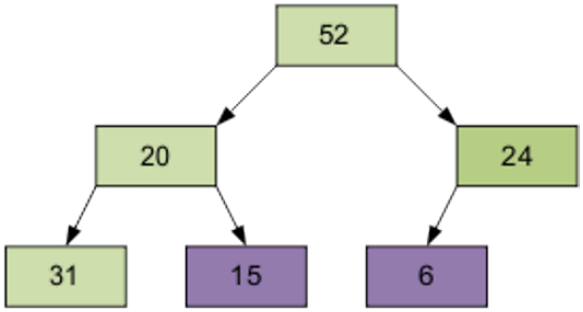
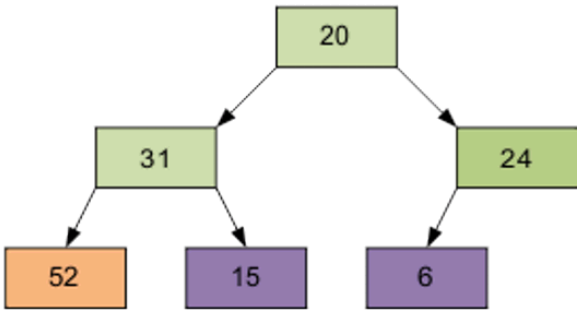
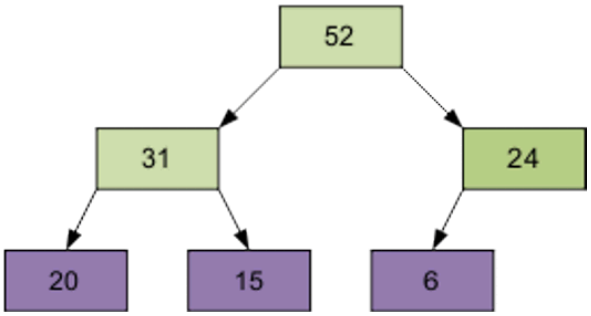


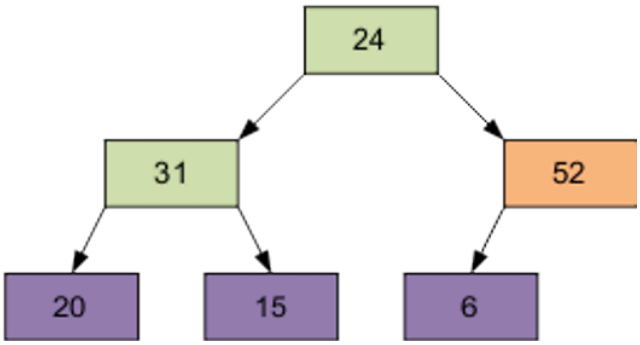
Рис. 10.14. Результат просеивания сквозь n-1 элементную пирамиду

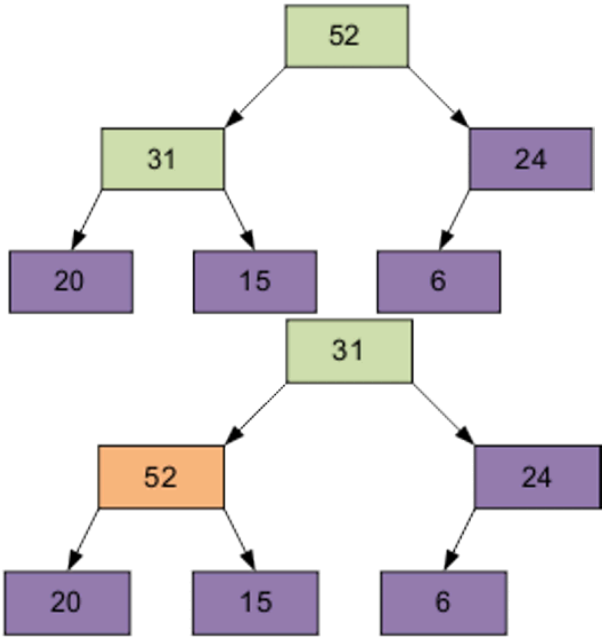
Продолжим процесс. В итоге массив будет отсортирован по убыванию (рис.10.15).

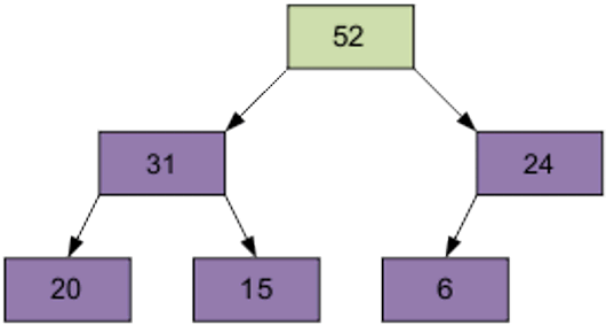





52
31
24
20
15
6


24
31
52
20
15
6


52
31
24
20
15
6
31
52
24
20
15
6


52
31
24
20
15
6

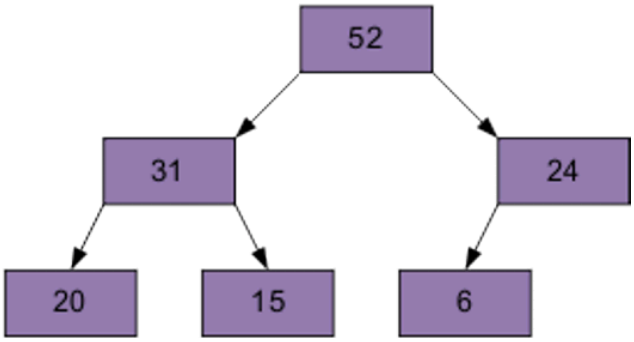


Рис. 10.15. Этапы сортировки массива по убыванию

Пример 10.7. Программная реализация алгоритма пирамидальной сортировки.

```
#include <stdio.h>
#include <stdlib.h>
void siftDown(int *numbers, int root, int bottom)
{
        int done, maxChild, temp;
        done = 0;
        while ((root*2 <= bottom) && (!done))
        {
          if (root*2 == bottom)
          maxChild = root * 2;
         else if (numbers[root * 2] > numbers[root * 2 + 1])
                maxChild = root * 2;
                else
                maxChild = root * 2 + 1;
        if (numbers[root] < numbers[maxChild])
        {
           temp = numbers[root];
           numbers[root] = numbers[maxChild];
           numbers[maxChild] = temp;
           root = maxChild;
        }  else
                done = 1;
     }
}
void heapSort(int *numbers, int array_size)
{
         int i, temp;
         for (i = (array_size / 2)-1; i >= 0; i--)
```

```
        siftDown(numbers, i, array_size);
        for (i = array_size-1; i >= 1; i--)
    {
            temp = numbers[0];
            numbers[0] = numbers[i];
            numbers[i] = temp;
            siftDown(numbers, 0, i-1);
    }
}
int main()
{
    int a[10];
    for(int i=0;i<10;i++)  a[i]=rand()%20-10;
    for(int i=0;i<10;i++)  printf("%d ",a[i]);
    printf("\n");
    heapSort(a,10);
    for(int i=0;i<10;i++)  printf("%d ",a[i]);
    printf("\n");
    getchar();
    return 0;
}
```

Анализ алгоритма пирамидальной сортировки

Несмотря на некоторую внешнюю сложность, пирамидальная сортировка является одной из самых эффективных. Алгоритм сортировки эффективен для больших **n**. В худшем случае требуется $\mathbf{n \cdot \log_2 n}$ шагов, сдвигающих элементы. Среднее число перемещений примерно равно $\mathbf{(n/2) \cdot \log_2 n}$, и отклонения от этого значения относительно невелики.

10.8. Быстрая сортировка

Быстрая сортировка представляет собой усовершенствованный метод сортировки, основанный на принципе обмена. Пузырьковая сортировка является самой неэффективной из всех алгоритмов прямой сортировки. Однако усовершенствованный алгоритм является лучшим из известных методом сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель Ч. Хоар назвал его быстрой сортировкой.

Для достижения наибольшей эффективности желательно производить обмен элементов на больших расстояниях. В массиве выбирается некоторый элемент, называемый ***разрешающим***. Затем он помещается в то место

массива, где ему полагается быть после упорядочивания всех элементов. В процессе отыскания подходящего места для разрешающего элемента производятся перестановки элементов так, что слева от них находятся элементы, меньшие разрешающего, и справа — большие (предполагается, что массив сортируется по возрастанию). Тем самым массив разбивается на две части:

- не отсортированные элементы слева от разрешающего элемента;
- не отсортированные элементы справа от разрешающего элемента.

Чтобы отсортировать эти два меньших подмассива, алгоритм рекурсивно вызывает сам себя.

Если требуется сортировать больше одного элемента, то нужно

- выбрать в массиве разрешающий элемент;
- переупорядочить массив, помещая элемент на его окончательное место;
- отсортировать рекурсивно элементы слева от разрешающего;
- отсортировать рекурсивно элементы справа от разрешающего.

Ключевым элементом быстрой сортировки является ***алгоритм переупорядочения***.

Рассмотрим сортировку на примере массива:

10, 4, 2, 14, 67, 2, 11, 33, 1, 15.

Для реализации алгоритма переупорядочения используем указатель left на крайний левый элемент массива. Указатель движется вправо, пока элементы, на которые он показывает, остаются меньше разрешающего. Указатель right поставим на крайний правый элемент массива, и он движется влево, пока элементы, на которые он показывает, остаются больше разрешающего.

Пусть крайний левый элемент — разрешающий. Установим на следующий за ним элемент; right — на последний (рис.10.16). Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.

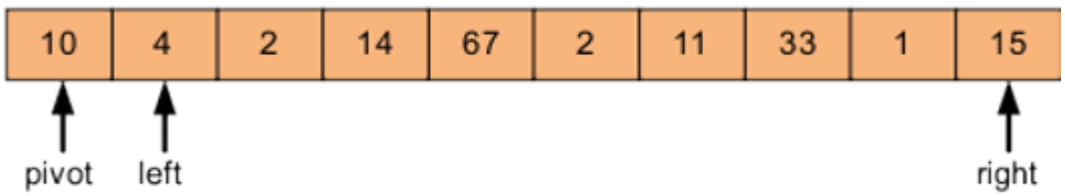


Рис. 10.16. Исходное состояние

Движение указателей останавливается, как только встречаются элементы, порядок расположения которых относительно разрешающего элемента неправильный.

Указатель left перемещается до тех пор, пока не покажет элемент больше 10; right движется, пока не покажет элемент меньше 10 (рис.10.17).

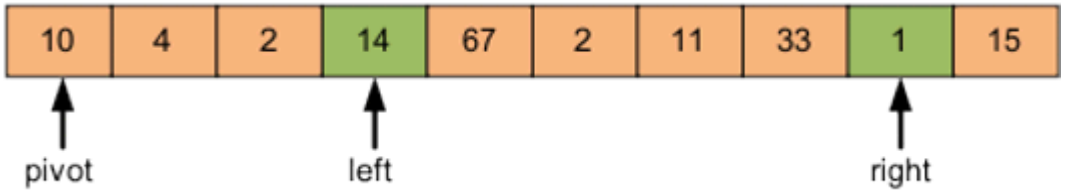


Рис. 10.17. Движение указателей. Шаг 1.

Эти элементы меняются местами и движение указателей возобновляется (рис.10.18).

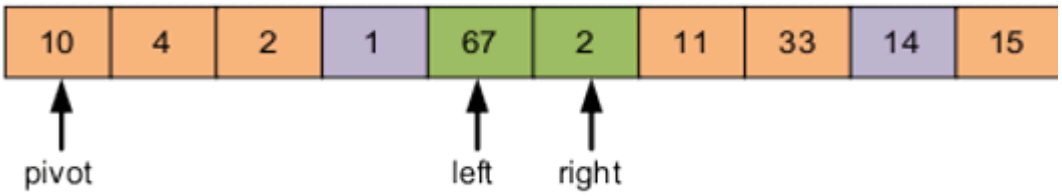


Рис. 10.18. Движение указателей. Шаг 2.

Процесс продолжается до тех пор, пока right не окажется слева от left (рис.10.19).

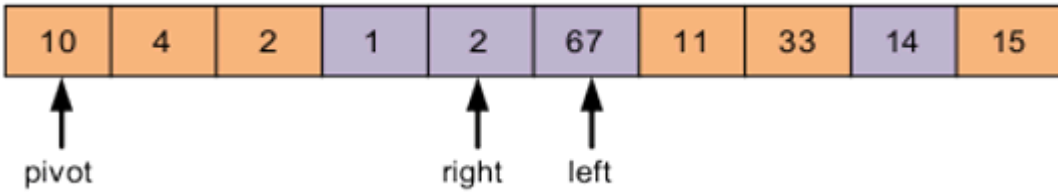


Рис. 10.19. Движение указателей. Шаг 3.

Тем самым будет определено правильное место разрешающего элемента.

Осуществляется перестановка разрешающего элемента с элементом, на который указывает (рис.10.20).

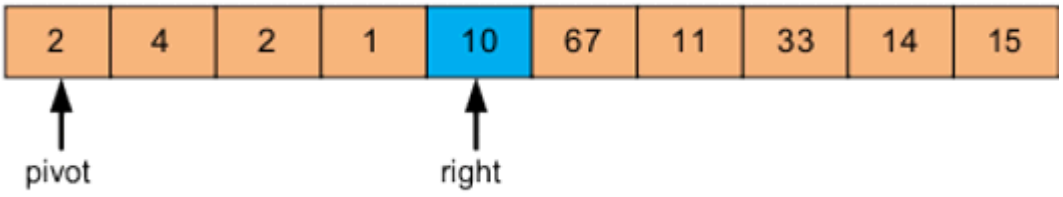


Рис. 10.20. Движение указателей. Шаг 4.

Разрешающий элемент находится в нужном месте: элементы слева от него имеют меньшие значения; справа — большие. Алгоритм рекурсивно вызывается для сортировки подмассивов слева от разрешающего и справа от него.

Пример 10.8. Программная реализация алгоритма быстрой сортировки.

```
#include <stdio.h>
#include <stdlib.h>
void q_sort(int *numbers, int left, int right)
{
  int pivot, l_hold, r_hold;
  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
    {
          while ((numbers[right] >= pivot) && (left < right))
         right--;
         if (left != right)
          {
           numbers[left] = numbers[right];
           left++;
           }
         while ((numbers[left] <= pivot) && (left < right))
         left++;
         if (left != right)
         {
          numbers[right] = numbers[left];
          right--;
         }
      }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
  q_sort(numbers, pivot+1, right);
}
int main()
{
  int a[10];
  for(int i=0;i<10;i++)  a[i]=rand()%100;
```

```
  for(int i=0;i<10;i++)  printf("%d ",a[i]);
  q_sort(a,0,9);
  printf("\n");
  for(int i=0;i<10;i++)  printf("%d ",a[i]);
  getchar();
  return 0;
}
```

10.9 Вопросы для самоконтроля

1. Классификация алгоритмов сортировки.
2. Алгоритм сортировки вставкой.
3. Алгоритм сортировки выбором.
4. Алгоритм сортировки обменом.
5. Алгоритм Шейкер-сортировки.
6. Алгоритм сортировки Шелла.