**АЛГОРИТМЫ ОБРАБОТКИ ИНФОРМАЦИИ**

Ответы на вопросы

1. ***Постановка задачи сортировки данных. Примеры использования отсортированных последовательностей.***

Задача сортировки часто возникает при создании различных систем автоматизации обработки данных на базе ЭВМ. В настоящее время известно множество алгоритмов сортировки, свойства которых достаточно хорошо изучены. Прежде чем рассмотреть основные из них, необходимо ввести базовые исходные понятия и сформулировать задачу.

Под данными в системах обработки данных понимаются описания фактов и понятий предметной области на формализованном входном языке описания данных. Средствами языка данные представляются в виде наборов (совокупностей) различных символов из некоторого исходного конечного множества символов, называемого алфавитом. Отдельные символы, входящие в эти наборы, являются элементами данных (простейшими данными). Они могут группироваться (объединяться), образуя различные структуры данных.

Под структурой данных в широком смысле понимается группа простейших данных, между которыми по определенному принципу установлены различные отношения. Структура данных, состоящая из элементов, связанных друг с другом по смыслу и обрабатываемых совместно, называется записью данных. Записи обычно делятся на части, которые называются полями. В ряде задач обработки данных к записям добавляется специальное поле, служащее для идентификации (определения) каждой записи по отношению к другим записям. Содержимое такого поля называется ключом записи. Конечная совокупность нескольких поименованных записей, в каждой из которых хранится однотипная информация на некотором внешнем носителе, например, магнитном, образует структуру данных, называемую файлом. Подобная же совокупность записей, хранящаяся в памяти, называется массивом.

Задача сортировки данных, решаемая для массивов и файлов, может быть сформулирована следующим образом.

Пусть имеется массив (файл) записей R1, R2, …, RN с ключами K1, K2, …, KN . На множестве ключей Ki, i=1, N вводится такое отношение порядка типа “≤”, что для любых трех значений ключей a, b, c выполняются следующие условия:

1) рефлексивность: a≤a;

2) антисимметричность: если a≤b, b≤a, то a=b;

3) транзитивность: если a≤b, b≤c, то a≤c;

4) линейность: для произвольных a и b

Любое множество записей, ключи которых удовлетворяют свойствам 1-4, называется полностью (линейно) упорядоченным (или совершенно упорядоченным).

Задача упорядочения (сортировки) данных состоит в том, чтобы найти такую перестановку записей, т.е. такую комбинацию их взаимного расположения, при которой ключи Ki расположились бы в неубывающем порядке:

K1 ≤ K2 ≤ … KN.

Если имеется возможность переставлять записи в произвольном порядке (в случае, когда все Ki различны, получится N! таких перестановок), то процесс переразмещения записей в упорядоченную последовательность называется сортировкой. Сортировка называется устойчивой, если она сохраняет упорядоченность записей и в случае равенства ключей отдельных записей. Сортировка записей, хранящихся в памяти, называется внутренней. Сортировка, которая выполняется для записей в файлах, называется внешней.

В дальнейшем рассматриваются методы внутренней сортировки на примере упорядочения простой последовательности целых чисел (x1, x2, …,xn), для которых значение ключа совпадает со значением элемента последовательности.

Примечание. Помимо понятия линейного упорядочения существует понятие лексикографического упорядочения, которое используется в задачах обработки буквенно-символьной информации. В узком смысле лексикографический порядок – это порядок слов в словаре, определяемый последовательностью букв некоторого алфавита. В общем случае рассматривается некоторое множество символов S={xi}, i=1, n, на котором существует отношение порядка “≤”. Для n>0 вводится множество T n-кортежей (x1, x2, …,xn), для которых определяется отношение упорядочения:

(x1, x2, …,xn) < (y1, y2, …,yn)

тогда и только тогда, когда существует некоторое значение k, 1≤k≤n, для которого xi=yi при 1≤i Если все кортежи расположены в соответствии с указанным отношением, то множество T считается лексикографически упорядоченным. Рассмотренное понятие может быть обобщено и для кортежей (строк) неодинаковой длины. При этом порядок строк будет совпадать с порядком слов в словаре.

1. ***Задача сортировки. Метод прямого выбора.***

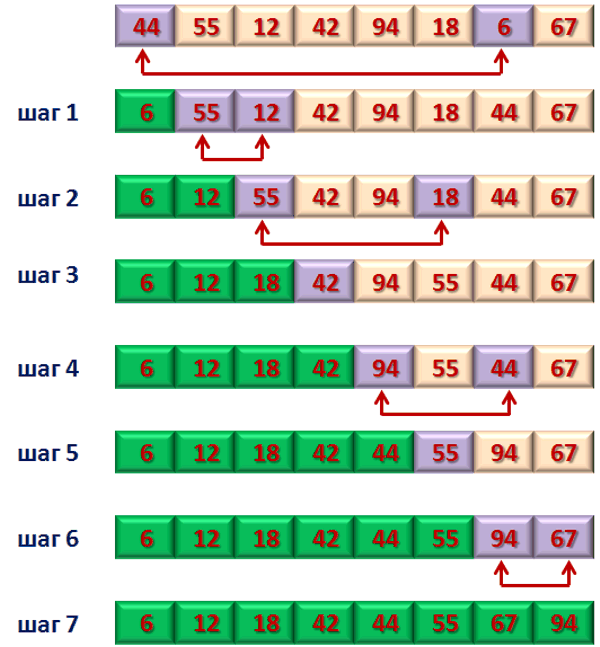
Алгоритм сортировки прямым выбором в некотором смысле противоположен сортировке прямыми включениями.

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.

Метод сортировки прямым выбором основан на следующих правилах:

* Выбирается элемент с наименьшим ключом.
* Он меняется местами с первым элементом a0.
* Затем эти операции повторяются с оставшимися n-1 элементами, n-2 элементами и так далее до тех пор, пока не останется один, самый большой элемент.



void selectSort(int arr[], int n)

{

int pos\_min,temp;

for (int i=0; i < n-1; i++)

{

pos\_min = i;//set pos\_min to the current index of array

for (int j=i+1; j < n; j++)

{

if (arr[j] < arr[pos\_min])

pos\_min=j;

//pos\_min will keep track of the index that min is in, this is needed when a swap happens

}

//if pos\_min no longer equals i than a smaller value must have been found, so a swap must occur

if (pos\_min != i)

{

temp = arr[i];

arr[i] = arr[pos\_min];

arr[pos\_min] = temp;

}

}

}

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

1. ***Задача сортировки. Метод прямого включения.***

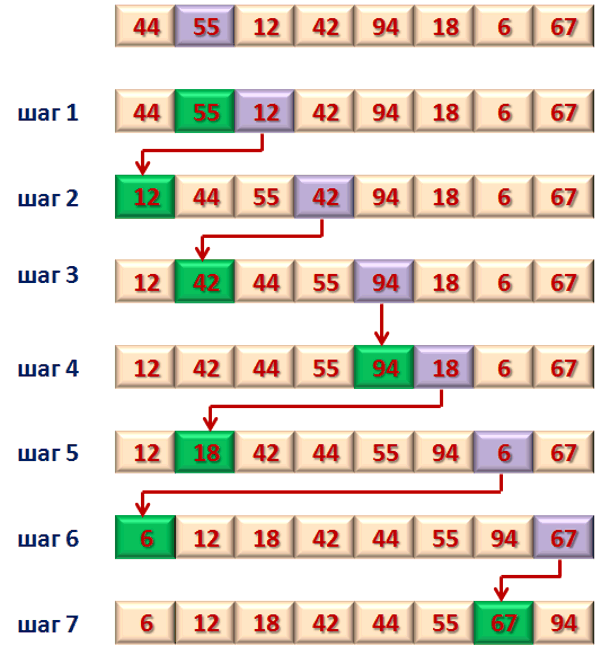
Элементы массива условно разделяются на готовую последовательность

**a1, a2, ..., ai-1**

и входную последовательность

**ai, ai+1, ..., an**.

На каждом шаге i-й элемент помещается на подходящее место в готовую последовательность.

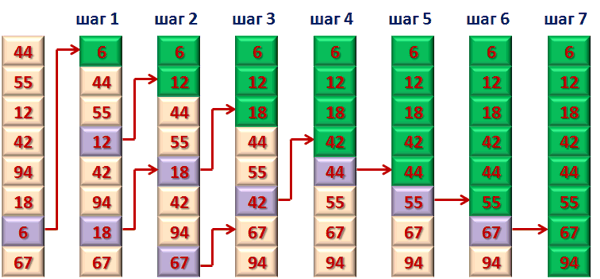


void inclusionSort(int \*num, int size)  
{  
  // Для всех элементов кроме начального  
  for (int i = 1; i < size; i++)  
  {  
    int value = num[i]; // запоминаем значение элемента  
    int index = i;     // и его индекс  
    while ((index > 0) && (num[index - 1] > value))  
    {   // смещаем другие элементы к концу массива пока они меньше index  
      num[index] = num[index - 1];  
      index--;    // смещаем просмотр к началу массива  
    }  
    num[index] = value; // рассматриваемый элемент помещаем на освободившееся место  
  }  
}

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.

1. ***Задача сортировки. Метод прямого обмена.***

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](https://prog-cpp.ru/sort-select/), совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу 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».  
  
  
***Резюме***: «обменная сортировка» представляет собой нечто среднее между сортировками с помощью включений и с помощью выбора; фактически в пузырьковой сортировке нет ничего ценного, кроме привлекательного названия.

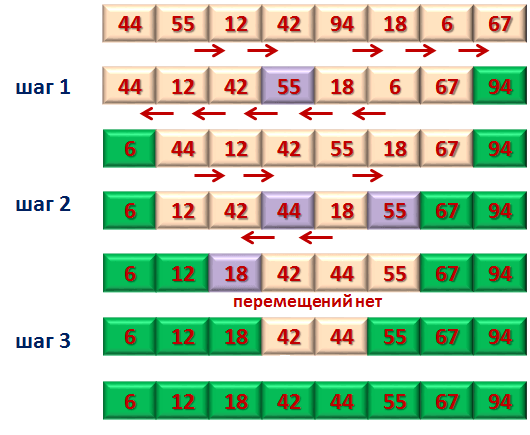
1. ***Улучшенная сортировка. Шейкерная сортировка.***

***Шейкер-сортировка*** является усовершенствованным методом [пузырьковой сортировки](https://prog-cpp.ru/sort-bubble/).   
  
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.

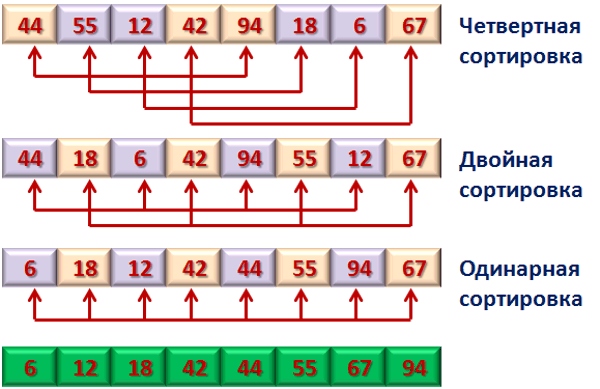
Эти две идеи приводят к модификациям в методе пузырьковой сортировки.

* 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. Дана последовательность  


void shekerSort(double \*mass, int count)  
{  
  int left = 0, right = count - 1; // левая и правая границы сортируемой области массива  
  int flag = 1;  // флаг наличия перемещений  
  // Выполнение цикла пока левая граница не сомкнётся с правой  
  // или пока в массиве имеются перемещения  
  while ((left < right) && flag > 0)  
  {  
    flag = 0;  
    for (int i = left; i<right; i++)  //двигаемся слева направо  
    {  
      if (mass[i]>mass[i + 1]) // если следующий элемент меньше текущего,  
      {             // меняем их местами  
        double t = mass[i];  
        mass[i] = mass[i + 1];  
        mass[i + 1] = t;  
        flag = 1;      // перемещения в этом цикле были  
      }  
    }  
    right--; // сдвигаем правую границу на предыдущий элемент  
    for (int i = right; i>left; i--)  //двигаемся справа налево  
    {  
      if (mass[i - 1]>mass[i]) // если предыдущий элемент больше текущего,  
      {            // меняем их местами  
        double t = mass[i];  
        mass[i] = mass[i - 1];  
        mass[i - 1] = t;  
        flag = 1;    // перемещения в этом цикле были  
      }  
    }  
    left++; // сдвигаем левую границу на следующий элемент  
  }  
}

1. ***Улучшенная сортировка. Сортировка Шелла.***

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.  
  
  
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.

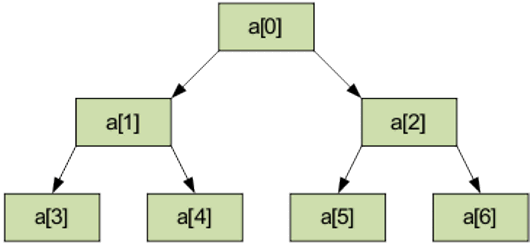
void shellSort(int \*num, int size)  
{  
  int increment = 3;    // начальное приращение сортировки  
  while (increment > 0)  // пока существует приращение  
  {  
    for (int i = 0; i < size; i++)  // для всех элементов массива  
    {  
      int j = i;          // сохраняем индекс и элемент  
      int temp = num[i];  
      // просматриваем остальные элементы массива, отстоящие от j-ого  
      // на величину приращения  
      while ((j >= increment) && (num[j - increment] > temp))  
      {  // пока отстоящий элемент больше текущего  
        num[j] = num[j - increment]; // перемещаем его на текущую позицию  
        j = j - increment;       // переходим к следующему отстоящему элементу  
      }  
      num[j] = temp; // на выявленное место помещаем сохранённый элемент  
    }  
    if (increment > 1)      // делим приращение на 2  
      increment = increment / 2;  
    else if (increment == 1)   // последний проход завершён,  
      break;  // выходим из цикла  
  }  
}

1. ***Улучшенная сортировка. Пирамидальная сортировка.***

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

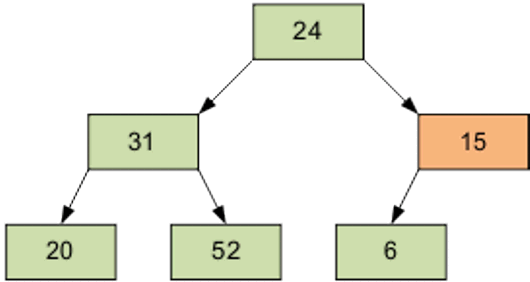
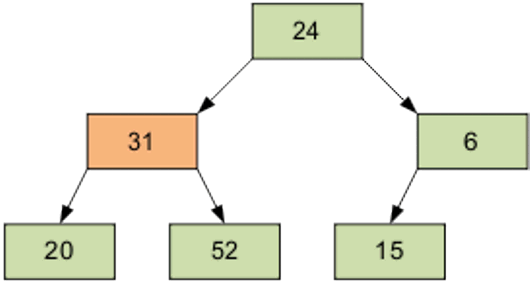
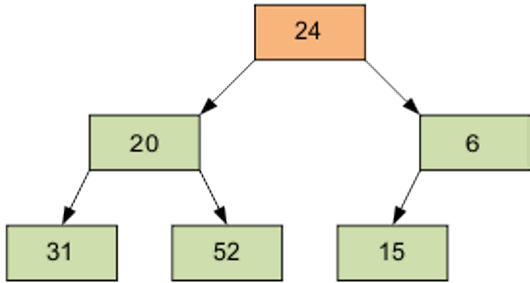
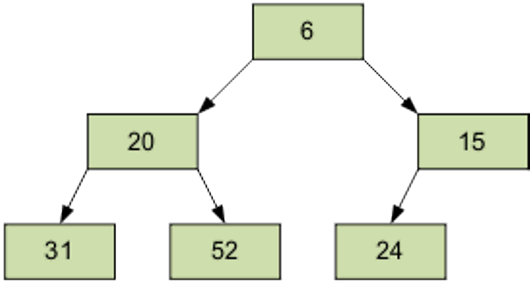
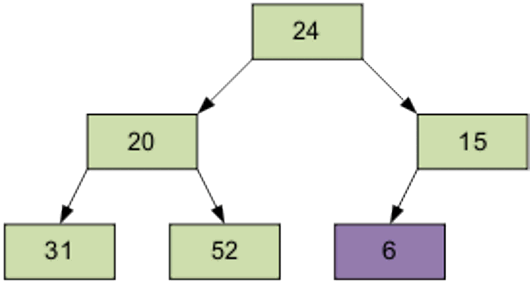
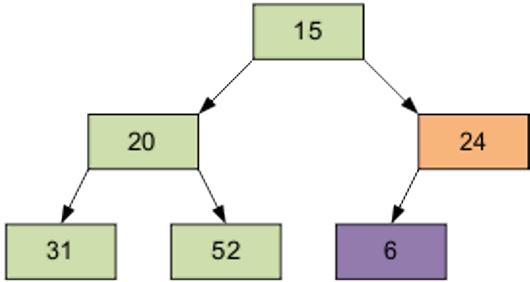
**a[i] ≤ a[2i+1];**

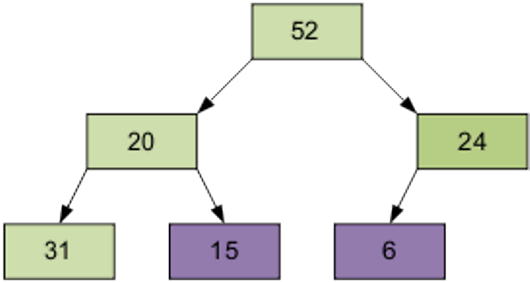
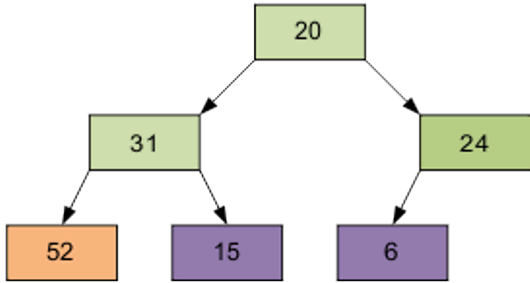
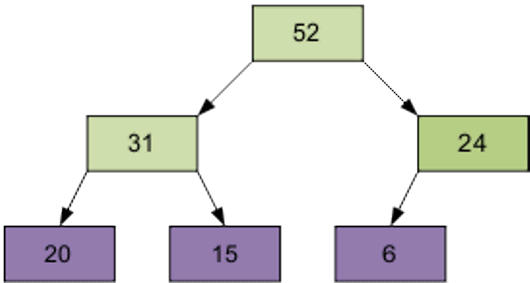
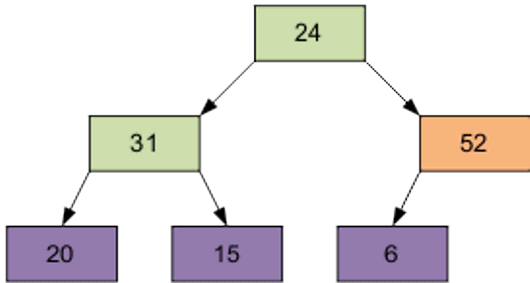
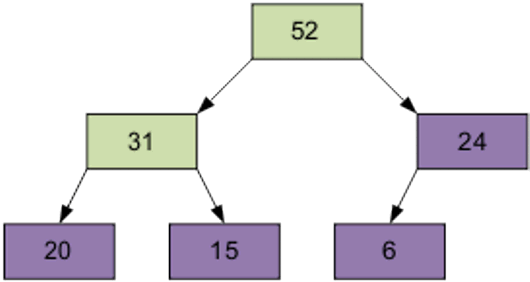
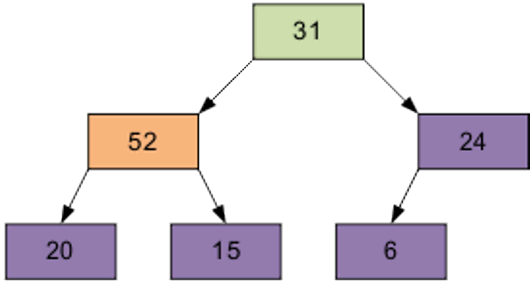
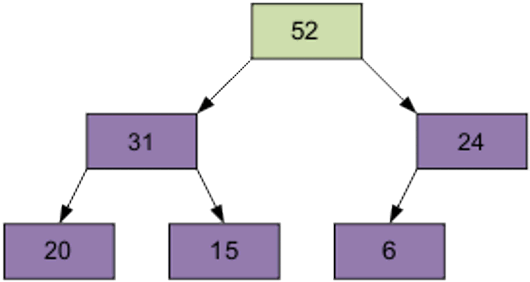
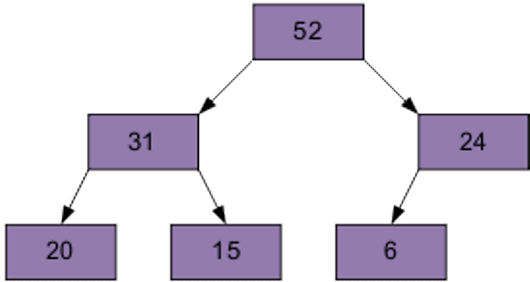
**a[i] ≤ a[2i+2].**

[Подробнее](https://prog-cpp.ru/data-heap/)  


**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.  
  
  
Результат просеивания элемента 15 через пирамиду.  
  
  
Следующий просеиваемый элемент – 1,  равный 31.  
  
  
Затем – элемент 0, равный 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. Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду. Затем берем предпоследний элемент и т.д.  
  
  
  
  
Продолжим процесс. В итоге массив будет отсортирован по убыванию.

void siftDown(int \*numbers, int root, int bottom)  
{  
  int maxChild; // индекс максимального потомка  
  int done = 0; // флаг того, что куча сформирована  
  // Пока не дошли до последнего ряда  
  while ((root \* 2 <= bottom) && (!done))   
  {  
    if (root \* 2 == bottom)    // если мы в последнем ряду,   
      maxChild = root \* 2;    // запоминаем левый потомок  
    // иначе запоминаем больший потомок из двух  
    else if (numbers[root \* 2] > numbers[root \* 2 + 1])  
      maxChild = root \* 2;  
    else  
      maxChild = root \* 2 + 1;  
    // если элемент вершины меньше максимального потомка  
    if (numbers[root] < numbers[maxChild])   
    {  
      int temp = numbers[root]; // меняем их местами  
      numbers[root] = numbers[maxChild];  
      numbers[maxChild] = temp;  
      root = maxChild;  
    }  
    else // иначе  
      done = 1; // пирамида сформирована  
  }  
}  
// Функция сортировки на куче  
void heapSort(int \*numbers, int array\_size)   
{  
  // Формируем нижний ряд пирамиды  
  for (int i = (array\_size / 2) - 1; i >= 0; i--)  
    siftDown(numbers, i, array\_size);  
  // Просеиваем через пирамиду остальные элементы  
  for (int i = array\_size - 1; i >= 1; i--)  
  {  
    int temp = numbers[0];  
    numbers[0] = numbers[i];  
    numbers[i] = temp;  
    siftDown(numbers, 0, i - 1);  
  }  
}

Несмотря на некоторую внешнюю сложность, пирамидальная сортировка является одной из самых эффективных. Алгоритм сортировки эффективен для больших **n**. В худшем случае требуется **n·log2n** шагов, сдвигающих элементы. Среднее число перемещений примерно равно

**(n/2)·log2n**,

и отклонения от этого значения относительно невелики.

1. ***Улучшенная сортировка. 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:

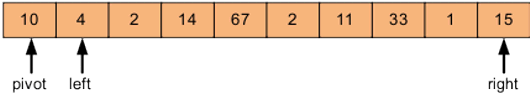
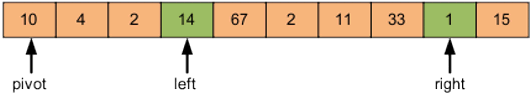
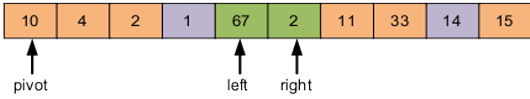
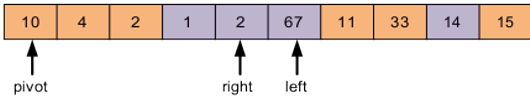
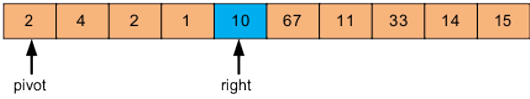
* не отсортированные элементы слева от разрешающего элемента;
* не отсортированные элементы справа от разрешающего элемента.

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 pivot. Установим указатель left на следующий за ним элемент; right — на последний. Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.  
  
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.  
  
  
Эти элементы меняются местами и движение указателей возобновляется.  
  
  
Процесс продолжается до тех пор, пока right не окажется слева от left.  
  
  
Тем самым будет определено правильное место разрешающего элемента.  
  
Осуществляется перестановка разрешающего элемента с элементом, на который указывает right.  
  
  
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.

void quickSort(int \*numbers, int left, int right)  
{  
  int pivot; // разрешающий элемент  
  int l\_hold = left; //левая граница  
  int r\_hold = right; // правая граница  
  pivot = numbers[left];  
  while (left < right) // пока границы не сомкнутся  
  {  
    while ((numbers[right] >= pivot) && (left < right))  
      right--; // сдвигаем правую границу пока элемент [right] больше [pivot]  
    if (left != right) // если границы не сомкнулись  
    {  
      numbers[left] = numbers[right]; // перемещаем элемент [right] на место разрешающего  
      left++; // сдвигаем левую границу вправо   
    }  
    while ((numbers[left] <= pivot) && (left < right))  
      left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]  
    if (left != right) // если границы не сомкнулись  
    {   
      numbers[right] = numbers[left]; // перемещаем элемент [left] на место [right]  
      right--; // сдвигаем правую границу вправо   
    }  
  }  
  numbers[left] = pivot; // ставим разрешающий элемент на место  
  pivot = left;  
  left = l\_hold;  
  right = r\_hold;  
  if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива  
    quickSort(numbers, left, pivot - 1);  
  if (right > pivot)  
    quickSort(numbers, pivot + 1, right);  
}