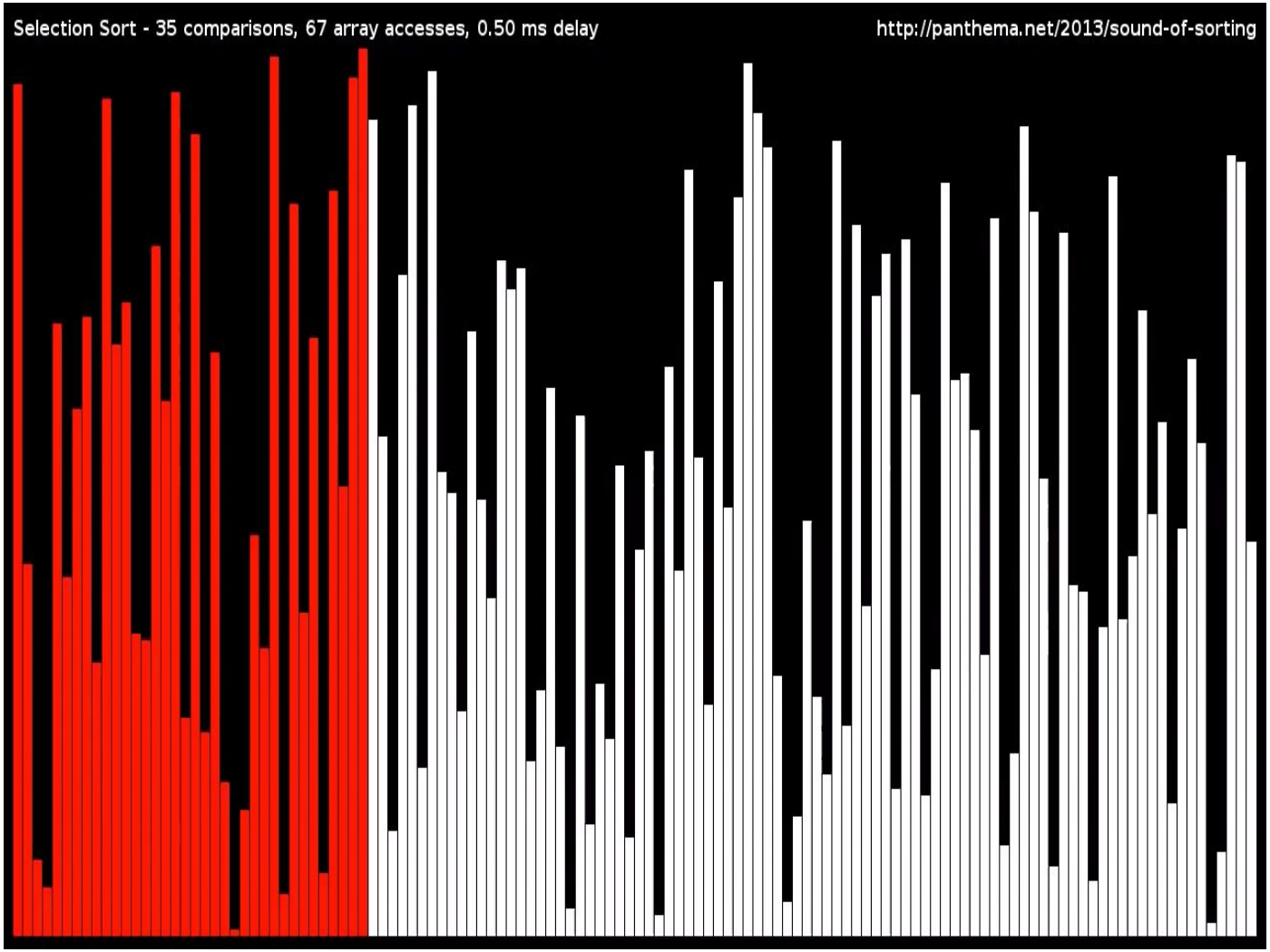
Алгоритмы сортировки



- В вашем шкафу множество рубашек, и потому очень трудно быстро отыскать ту, которую вы хотите надеть. Как следует развесить рубашки, чтобы вам было легко отыскать нужную?

Алгоритмы сортировки

Алгоритм сортировки – это алгоритм упорядочивания элементов списка. Упорядочивание производится в соответствии со значением ключа сортировки по возрастанию или убыванию его значения.

Задача алгоритмов сортировки

Задача сортировки состоит в том, что в произвольной последовательности элементов списка, нужно расположить их таким образом, чтобы каждое последующее значение ключа сортировки было больше предыдущего, в случае сортировки по возрастанию. Если производится сортировка по убыванию, каждое последующее значение ключа, соответственно, должно быть меньше предыдущего.

Характеристики алгоритмов сортировки

Время – главный параметр, показывающий скорость упорядочения элементов от исходного произвольного порядка к конечному результату. В основном, на время, влияет количество шагов, за которое алгоритм придет к результату. Под шагами подразумевается количество сравнений и замен элементов. Причем скорость упорядочения одного вида алгоритма может существенно отличатся для разных наборов сортируемых данных. Различают худший, средний и лучший результат работы алгоритма. При максимально возможном количестве шагов, получим худший результат. Средний результат, соответственно при среднем количестве возможного количества шагов алгоритма. Лучший результат чаще всего можно получить, если исходная последовательность уже упорядочена или близка к этому.

Память – занимаемое дополнительное место в памяти. Некоторые алгоритмы для своей работы дублируют исходный массив или его части. Учитывая что сам массив может быть очень большим, то затраты памяти на работу таких алгоритмов могут быть очень существенными, а иногда могут являться и основным критерием при выборе алгоритма сортировки. Память, затрачиваемая на хранение исходного массива, в учет этого критерия не берется.

Устойчивость – характеристика, определяющая производится ли замена элементов с одинаковым значением ключа сортировки.

Сфера применения – определяет назначение алгоритмов и характеризуется двумя типами сортировки – внутренней и внешней. Внутренняя сортировка, т.е. работа с массивом данных, целиком расположенном в памяти и возможным доступом к его любому элементу. Внешняя сортировка характеризуется работой с данными большого

Критерии выбора алгоритма сортировки

Критерий	Алгоритм сортировки
Только несколько элементов	Сортировка вставками
Элементы уже почти отсортированы	Сортировка вставками
Важна производительность в наихудшем случае	Пирамидальная сортировка
Важна производительность в среднем случае	Быстрая сортировка
Элементы с равномерным распределением	Блочная сортировка
Как можно меньший код	Сортировка вставками
Требуется устойчивость сортировки	Сортировка слиянием

Сортировка вставками (Insertion Sort)

В этом методе из неупорядоченной последовательности выбирается поочередно каждый элемент, сравнивается с предыдущими (уже упорядоченными) и помещается на соответствующее место.

Это похоже на то, как карточные игроки сортируют имеющиеся у них карты. Каждая карта, начиная со второй, ставится сразу на свое место по старшинству среди стоящих левее карт. При получении (выборе) каждой следующей карты, имеющиеся на руках (или стоящие левее) оказываются уже отсортированы.

Метод хорош для сортировки данных, получаемых из входного потока.

Наилучший случай: O (n) Средний, наихудший случаи: $0(n^2)$

6 5 3 1 8 7 2 4

```
//сортировка вставками
void InsertionSort(int *arr, int size){
    int buf;
    int i, j;
    for(i=1; i<size; i++){
        buf=arr[i];
        for(j=i-1; j>=0 && buf<arr[j]; j--)
            arr[j+1]=arr[j];
        arr[j+1]=buf;
    }
}
```

Пирамидальная сортировка (Heap Sort)

6 5 3 1 8 7 2 4

Наилучший, средний, наихудший случаи: 0(n*logn)

Краткое описание алгоритма:

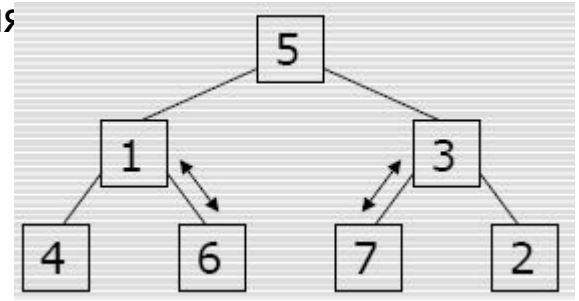
1. Подготовка (просеивание) – вершина дерева должна быть больше любого элемента в поддеревьях.

2. Выбор – выкидываем вершину

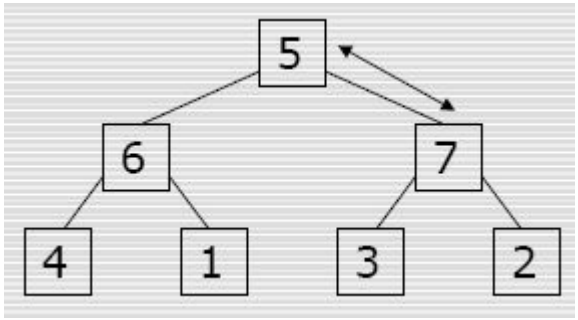
3. Повтор – переходим к 1 с меньшим количеством вершин

Сортировка двоичной кучей – подготовка

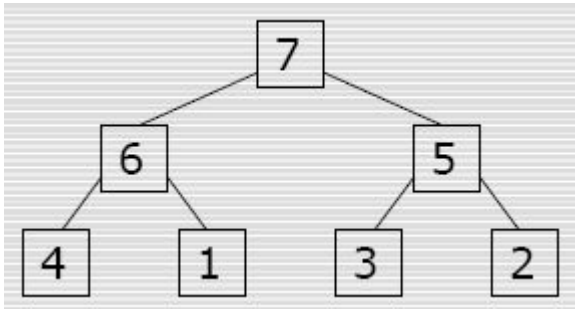
Если корень какого-либо поддерева имеет индекс N, то поддеревья нижнего уровня имеют индексы 2N+1 и 2N+2. Аналогично для не корневого узла с меткой N его родительский узел имеет метку $\lfloor (N-1)/2 \rfloor$. С помощью такой схемы можно хранить пирамиды в массиве, сохраняя значение элемента узла в позиции массива, которая определя



5	1	3	4	6	7	2

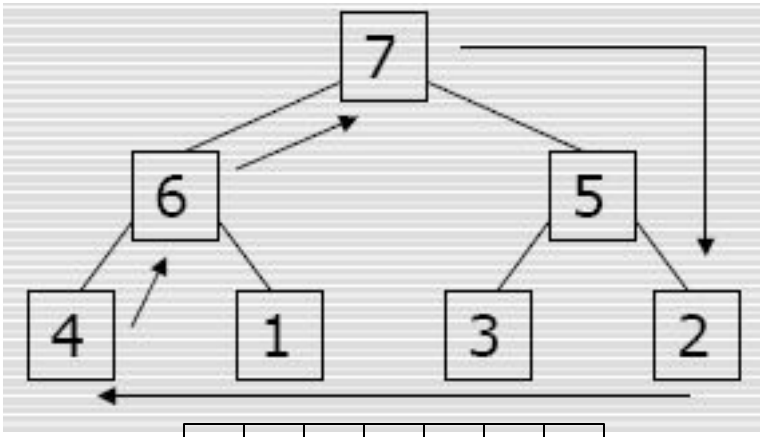


5	6	7	4	1	3	2

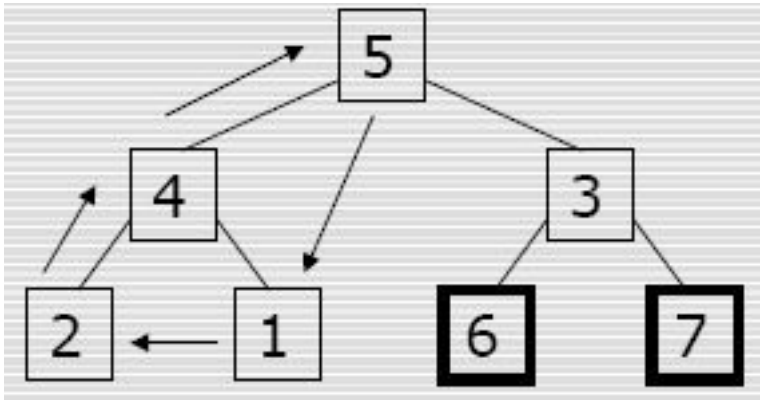


7	6	5	4	1	3	2

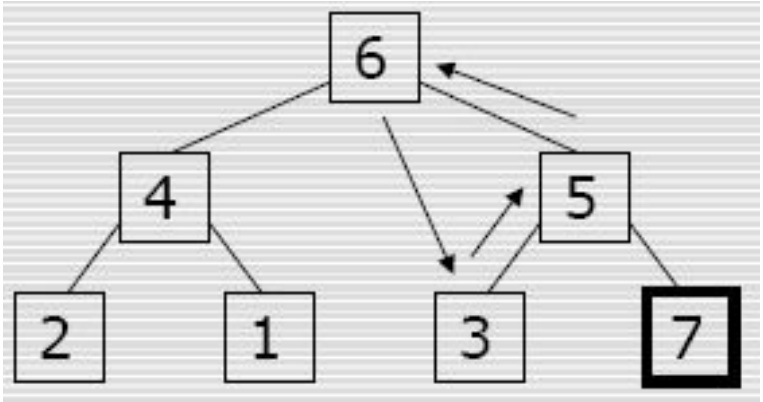
Сортировка двоичной кучей – выбор



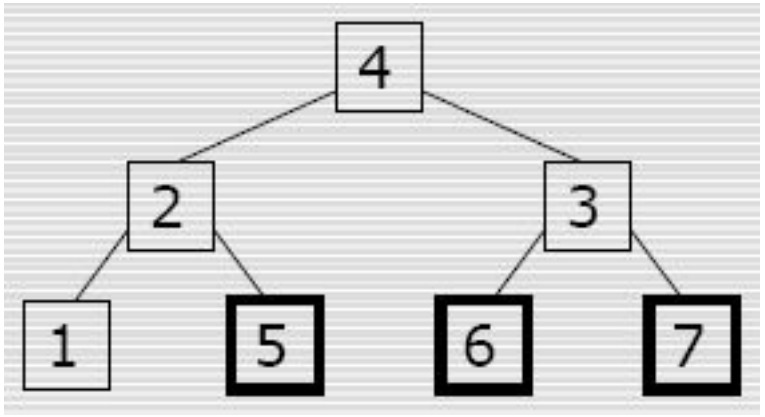
7	6	5	4	1	3	2



5	4	3	2	1	**6**	**7**

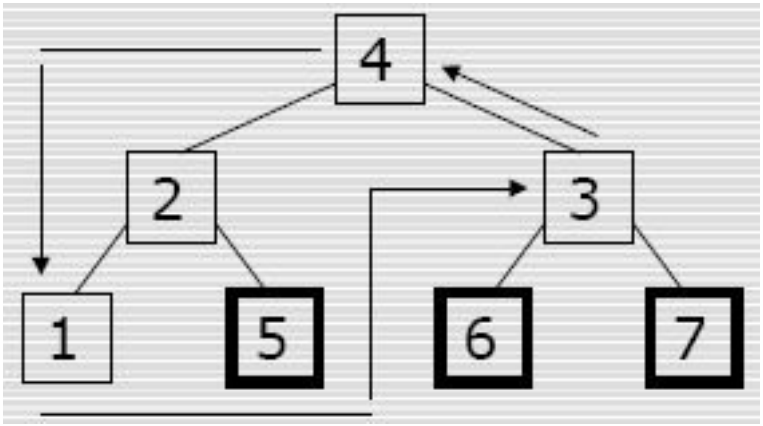


6	4	5	2	1	3	**7**

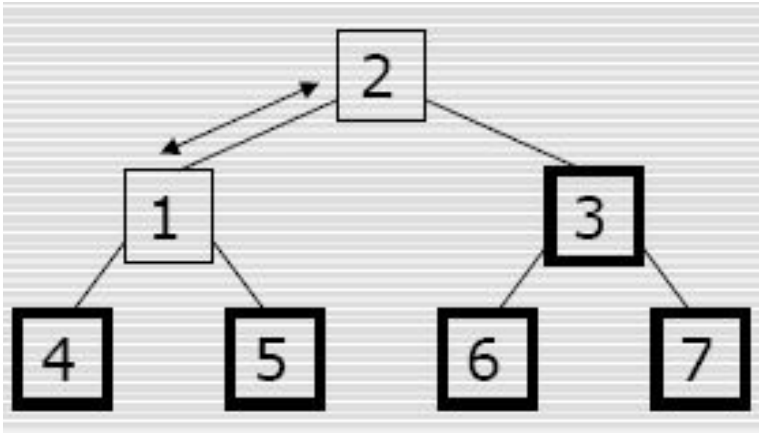


4	2	3	1	**5**	**6**	**7**

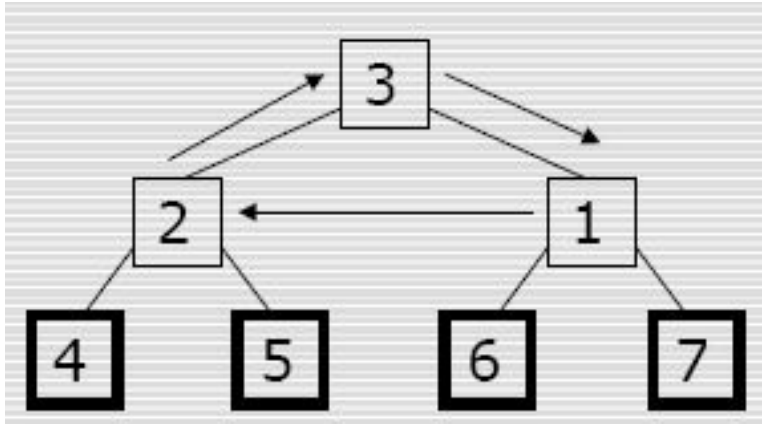
Сортировка двоичной кучей – выбор



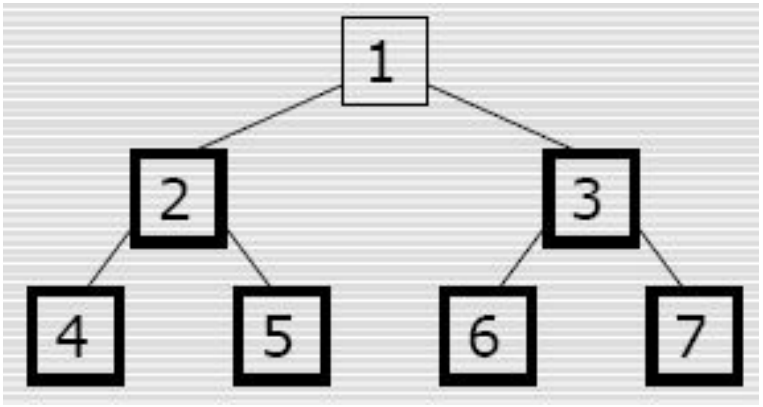
4	2	3	1	5	6	7



2	1	3	4	5	6	7



3	2	1	4	5	6	7



1	2	3	4	5	6	7

```
//пирамидальная сортировка
void HeapSort(int *arr, int size){
    int buf;
    for(int i=size; i>1; i--){
    //построение двоичного дерева
        bool flag = false;
        int sh = 0; //смещение
        for(int j=0; j<i/2; ){
            int largest=j, left=2*j+1, right=2*j+2;
            if(arr[left]>arr[largest]){
                buf=arr[left];
                arr[left]=arr[largest];
                arr[largest]=buf;
                flag = true;
            }
            if(right<i){
                if(arr[right]>arr[largest]){
                    buf=arr[right];
                    arr[right]=arr[largest];
                    arr[largest]=buf;
                    flag = true;
                }
            }
            if(flag && j>0){
            //возврат к корню для упорядочения ветки
                j=(j-1)/2;
                sh++;
                flag = false;
            }
            else{
            //продолжение построения дерева
                j=j+1+sh;
                sh=0;
            }
        }
    //перестановка первого и последнего элементов
        buf=arr[i-1];
        arr[i-1]=arr[0];
        arr[0]=buf;
    }
}
```

Разновидностью пирамидальной сортировки является "плавная сортировка", разработанная Э. Дейкстрой в 1981 г. В отличие от пирамидальной сортировки, используется не двоичная куча, а специальная куча, полученная с помощью чисел Леонардо. Преимущество плавной сортировки в том, что её сложность приближается к *O(n), если* входные данные частично отсортированы.

Пирамидальная сортировка не является устойчивой. Она позволяет избежать многих неприятных (почти неловких) ситуаций, которые приводят к плохой производительности быстрой сортировки. Тем не менее в среднем случае быстрая сортировка превосходит пирамидальную.

Быстрая сортировка (Quicksort)

6 5 3 1 8 7 2 4

Наилучший и средний случаи: (n*logn), наихудший случай: $O(n^2)$

Алгоритм сортировки Хоара (разделяй и властвуй)

1. Выбираем «медиану»
   - наиболее эффективный способ – случайным образом
2. Создаем левый и правый индекс
3. Пока индексы не перекрылись
   - Двигаем левый индекс вправо, пока не найдем элемент >= «медианы»
   - Двигаем правый индекс влево, пока не найдем элемент <= «медианы»
   - Если индексы не перекрылись, меняем элементы местами и сдвигаем индексы еще на один элемент
4. Повторяем сортировку для двух меньших последовательностей (переходим к пункту 1)

Переупорядочивание

- 1. Выбираем опорный элемент $x := a[(l+r) / 2]$
- 2. $i = l$
- 3. $j = r$
- 4. Двигаясь вправо, просматриваем поочерёдно все элементы массива, пока не встретим некоторый $a[i] \geq x$.
- 5. Двигаясь влево, просматриваем поочерёдно все элементы массива, пока не встретим некоторый $a[j] \leq x$.
- 6. Если $i \leq j$,
  - a) меняем местами $a[i]$ и $a[j]$,
  - b) i увеличиваем на единицу, продвигаясь на 1 шаг вправо,
  - c) j уменьшаем на единицу, продвигаясь на 1 шаг влево.
- 7. Если $i > j$, то цель достигнута, иначе перейти к пункту 4

Быстрая сортировка. Переупорядочивание

44	55	12	*42*	94	6	18	67

Опорным элементом будет $x = 42$.
Выполняя пункт 4 алгоритма, получим $i = 1$.
Выполняя пункт 5, получим $j = 7$.
Обмениваем $a[1]$ и $a[7]$ и продвигаемся к следующим элементам. Новые значения $i = 2$ и $j = 6$

18	55	12	*42*	94	6	44	67

Уже пройденная часть массива показана заливкой.
$i \le j$, поэтому продолжаем выполнение операторов тела цикла — переходим к пункту 4.
В результате выполнения п. 4 и п. 5 значения переменных i и j не изменятся.
Выполняя п. 6, обмениваем $a[2]$ с $a[6]$ и переходим к $i = 3$ и $j = 5$.

18	6	12	*42*	94	55	44	67

Опять $i \le j$, поэтому переходим к п. 4.
В результате п. 4 получаем $i = 4$.
В результате п. 5 получаем $j = 4$.
Операторы п. 6 приведут к тому, что $a[4] = 42$ будет обменян сам с собой, а также (и это главное) изменятся значения переменных i и j. Теперь $i = 5$, а $j = 3$.

18	6	12	*42*	94	55	44	67

Теперь $i > j$, а это значит, что работа процедуры разделения завершена. Теперь все элементы в части массива левее индекса $i = 5$ меньше или равны x, а правее $j = 3$ — больше или равны x. Цель достигнута.

Быстрая сортировка (Quicksort)

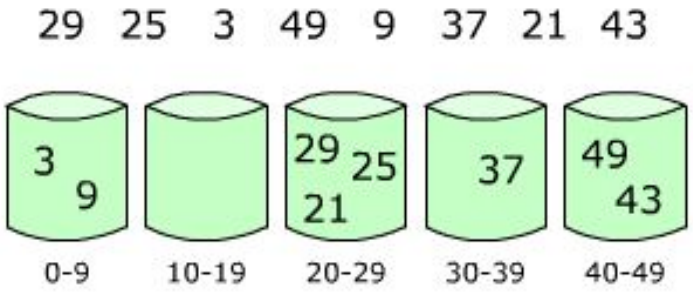
```
void QuickSort(int *arr, int first, int last){
    int i = first, j = last;
    int buf, comp;
    comp = arr[(first + last) / 2];
    do{
        while (arr[i] < comp && i < last)
            i++;
        while (arr[j] > comp && j > first)
            j--;
        if(i <= j){
            if (arr[i] > arr[j]){
                buf=arr[i];
                arr[i]=arr[j];
                arr[j]=buf;
            }
            i++; j--;
        }
    }while (i <= j);
    if(first < j)
        QuickSort(arr, first, j);
    if (i < last)
        QuickSort(arr, i, last);
}
```

Одним из таких недостатков является чувствительность алгоритма к степени упорядоченности входных данных. В худшем случае или близком к нему (что может случиться при неудачных входных данных) алгоритм сильно деградирует по скорости, а прямая реализация в виде функции с двумя рекурсивными вызовами может привести при этом к ошибке переполнения стека. Последней можно избежать, путем модификации алгоритма, устраняющей одну ветвь рекурсии. То есть вместо того, чтобы после разделения массива вызывать рекурсивно процедуру разделения для обоих найденных подмассивов, рекурсивный вызов делается только для меньшего подмассива, а больший обрабатывается в цикле в пределах этого же вызова процедуры.

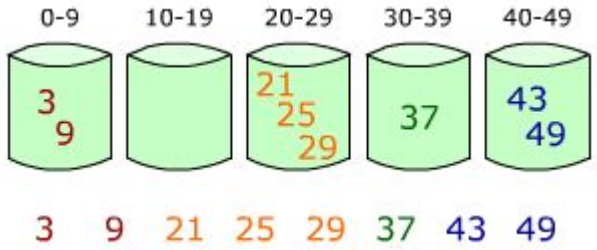
Переполнение также не произойдет, если разбивать массив не на две, а на три части, либо при достижении нежелательной глубины рекурсии переходить на сортировку другими методами, не требующими рекурсии.

Блочная сортировка (Bucket Sort)

Блочная сортировка (Карманная сортировка, корзинная сортировка) — алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив. Этот тип сортировки может обладать линейным временем исполнения.



Элементы распределяются по корз



Затем элементы в каждой корзине сортируются

```
void bucketSort(int *arr, int n){
    int **b = new int*[n];
    int *k = new int[n];
    for(int i=0; i<n; i++){
        b[i] = new int[n];
        k[i] = 0;
    }

    for (int i=0; i<n; i++){
       int x = arr[i] / 10;
       b[x][k[x]++] = arr[i];
    }

    for (int i=0; i<n; i++){
        InsertionSort(arr, k[i]-1);
//      QuickSort(b[i], 0, k[i]-1);
    }

    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < k[i]; j++)
          arr[index++] = b[i][j];
}
```

Вместо *n блоков сортировка хешированием (Hash Sort) создает достаточно большое* количество блоков *к на которые подразделяются элементы; с ростом к производительность* сортировки хешированием растет.

Сортировка слиянием (Merge Sort)

Алгоритм разработал Джон фон Нейман в 1945 году.

6 5 3 1 8 7 2 4

Используется для эффективной сортировки данных, которые хранятся во внешнем файле

- Алгоритм имеет индуктивный характер
  - Разобьём массив пополам
  - Отсортируем каждую из половин **тем же способом**
    - Один ключ отсортирован автоматически
  - Сольём две отсортированные половины вместе
- Слияние половин
  - Для каждой половины заводим индекс, вначале он указывает на наименьший элемент
  - Выбираем больший элемент из двух, переносим в результирующий массив
  - Увеличиваем индекс и повторяем процедуру

Сортировка слиянием (Merge Sort)

```
sort (A)
    if A имеет меньше 2 элементов then
     return A
    else if A имеет два элемента then
     Обменять элементы A, если они не в порядке
     return A

     sub1 = sort(left half of A)
     sub2 = sort(right half of A)

     Объединить sub1 и sub2 в новый массив B
     return B
end
```

```
void MergeSort(int arr[], int size){
    int i; //BlockSizeIterator
    int j; //BlockIterator
    int lbi; //LeftBlockIterator
    int rbi; //RightBlockIterator
    int mi; //MergeIterator
    int lb; //LeftBorder
    int mb; //MidBorder
    int rb; //RightBorder

    for (i=1; i < size; i*=2){
        for (j=0; j < size-i; j+=2*i){
    //Производим слияние с сортировкой пары блоков начинающуюся с элемента j
    //левый размером i, правый размером i или меньше
            lbi=0;
            rbi=0;
            lb=j;
            mb=j+i;
            rb=j+2*i;
            rb=(rb < size)? rb : size;
            int* SortedBlock = new int[rb-lb];
    //Пока в обоих массивах есть элементы выбираем меньший из них и заносим в отсортированный блок
            while (lb+lbi < mb && mb+rbi < rb){
                if (arr[lb+lbi] < arr[mb+rbi]){
                    SortedBlock[lbi+rbi] = arr[lb+lbi];
                    lbi++;
                }
                else{
                    SortedBlock[lbi+rbi] = arr[mb+rbi];
                    rbi++;
                }
            }
    //После этого заносим оставшиеся элементы из левого или правого блока
            while (lb+lbi < mb){
                SortedBlock[lbi+rbi] = arr[lb+lbi];
                lbi++;
            }
            while (mb+rbi < rb){
                SortedBlock[lbi+rbi] = arr[mb+rbi];
                rbi++;
            }
            for (mi=0; mi<lbi+rbi; mi++){
                arr[lb+mi] = SortedBlock[mi];
            }
            delete SortedBlock;
    }}
}
```

Анализ методов

Фундаментальным научным результатом является тот факт, что не существует алгоритма сортировки сравнением элементов, который имел бы лучшую производительность, чем *O(n*logn), в среднем или наихудшем случае.*
Если имеется n *элементов, то всего есть n! перестановок этих элементов.*
Каждый алгоритм, который сортирует элементы с помощью попарных сравнений, соответствует бинарному дереву решений. Листья дерева соответствуют перестановкам, и каждая перестановка должна соответствовать хотя бы одному листу дерева. Узлы на пути от корня к листу соответствуют последовательности сравнений. Высота такого дерева определяется как число узлов сравнения на самом длинном пути от корня к листьям.
Для любого заданного бинарного дерева решений для сравнения n *элементов можно указать его минимальную высоту h, т.е. должен быть некоторый* лист, который требует *h узлов сравнений на пути от корня до этого листа дерева.*
Рассмотрим полное двоичное дерево высотой *h, в котором все узлы, не являющиеся* листьями, имеют как левый, так и правый дочерние узлы. Это дерево содержит $n = 2^h - 1$ *узлов, а его высота равна* $h = log(n+1)$. *Если дерево не является полным, оно*
может быть несбалансированным любым, пусть даже самым странным, образом, но
мы знаем, что $h >= log(n+1)$. *Любое бинарное дерево решений с n! конечных узлов*
уже имеет минимум n! узлов в общей сложности. Нам осталось только

Воспользуемся следующими свойствами логарифмов:

$$\log(a \cdot b) = \log(a) + \log(b) \text{ и } \log(x^y) = y \cdot \log(x).$$

Тогда

$$h = \log(n!) = \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1)$$

$$h > \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot n/2)$$

$$h > \log\left((n/2)^{n/2}\right)$$

$$h > (n/2) \cdot \log(n/2)$$

$$h > (n/2) \cdot (\log(n) - 1)$$

Что же означает последнее неравенство? Для заданных n *элементов, которые необходимо* отсортировать, будет хотя бы один путь от корня к листу длиной *h, а значит,* алгоритм, который сортирует с помощью сравнений, требует как минимум указанного количества сравнений для сортировки n *элементов. Обратите внимание, что h вычисляется как функция f(n); в частности, в данном случае f(n) = (1l2)*n*log(n) - n/2.*

Таким образом, любой алгоритм с использованием сравнений потребует выполнения

0(nlog(n)) сравнений для сортировки.