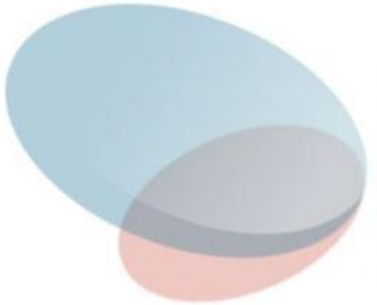


Алгоритмы и структуры данных
Лекция 1
Алгоритмы сортировки

Сортировка – это процесс упорядочения некоторого множества элементов, на котором определено отношение порядка >, <, ≥, ≤, =. Когда говорят о сортировке, подразумевают упорядочение множества элементов по возрастанию или убыванию.

Традиционно различают

внутреннюю сортировку, в которой предполагается, что

- данные находятся в оперативной памяти, и
- важно оптимизировать число действий программы (для методов, основанных на сравнении, число сравнений, обменов элементов и пр.), и

внешнюю, в которой

- данные хранятся на внешнем устройстве и,
- прежде всего, требуется снизить число обращений к этому устройству.

Оценка сложности алгоритма сортировки

- **Алгоритмы сортировки ведут себя по-разному в различных обстоятельствах.**

  Например, пузырьковая сортировка опережает быструю сортировку по скорости работы, если сортируемые элементы уже были почти упорядочены, но работает медленнее, если элементы были расположены хаотично.

- **Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти.**

Время — основной параметр, характеризующий быстродействие алгоритма. Называется также ***вычислительной сложностью***. Для упорядочения важны ***худшее***, ***среднее*** и ***лучшее*** поведение алгоритма в терминах мощности входного множества A. Если на вход алгоритму подаётся множество A, то обозначим n = | A |. Для типичного алгоритма хорошее поведение — это *O(n log n),* и плохое поведение — это *O(n^2)*. Идеальное поведение для упорядочения — *O(n)*.

Алгоритмы сортировки, использующие только абстрактную операцию *сравнения ключей*, всегда нуждаются по меньшей мере в *O(n log n)* сравнениях.

Память – второй параметр, характеризующий эффективность алгоритма**. Ряд алгоритмов требует выделения *дополнительной* памяти под временное хранение данных**. Как правило, эти алгоритмы требуют *O(log n)* памяти. Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к **сортировкам на месте**.

Классификация алгоритмов сортировки

- ***Устойчивость*** (stability) — устойчивая сортировка **не меняет взаимного расположения равных элементов.**
- ***Естественность поведения*** — эффективность метода **при обработке уже упорядоченных**, или частично упорядоченных данных. **Алгоритм** ведёт себя естественно, если учитывает эту характеристику входной последовательности и **работает быстрее**.
- ***Использование операции сравнения***. Алгоритмы, использующие для сортировки **сравнение элементов между собой**, называются **основанными на сравнениях**. **Минимальная** трудоемкость худшего случая для этих алгоритмов составляет *O(n log n)*, но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Общие принципы преобразования данных при использовании алгоритмов сортировки

- ***Таблицы указателей***

При сортировке элементов данных программа перестраивает их в некоторую структуру данных. Скорость этого процесса зависит от **типа** элементов. Перемещение целого числа на новое место в массиве может быть намного быстрее, чем перемещение определенной пользователем структуры данных. **Для повышения производительности при сортировке больших объектов**, например, записей, можно **помещать ключевые поля данных, используемые для сортировки, в таблицу индексов (указателей).**

Замечание

- При добавлении или удалении записи необходимо обновлять каждую таблицу индексов независимо.
- Таблицы индексов занимают дополнительную память. Если создать таблицу индексов для каждого из полей данных, объем занимаемой памяти более чем удвоится.

Общие принципы преобразования данных при использовании алгоритмов сортировки

- ***Объединение и сжатие ключей***

- В некоторых случаях можно **хранить ключи списка в комбинированной или сжатой форме.**

  Например, **можно объединять (combine) в программе два поля,** соответствующих имени и фамилии, в один ключ. Это позволит упростить и ускорить сравнение.

- **Иногда можно сжимать (compress) ключи.** Сжатые ключи занимают меньше места, уменьшая размер таблиц индексов. Это позволяет сортировать списки большего размера без перерасхода памяти, быстрее перемещать элементы в списке.

- **Один из методов сжатия строк — кодирование их целыми числами или данными другого числового формата**. Числовые данные занимают меньше места, чем строки и сравнение двух численных значений также происходит намного быстрее, чем сравнение двух строк.

Пример. Требуется **закодировать строки, состоящие из заглавных латинских букв**. Закодируем каждый символ числом по основанию 27 (26 латинских букв и еще одна цифра для обозначения конца слова).

Пусть сравниваются слова максимальной длины 3.

Код по основанию 27 для строки из трех символов дает формула

27^2 * (ord(первая буква) - ord(A) + 1) + 27 * (ord(вторая буква) - ord(A) + 1) + (ord(третья буква) - ord(A) + 1).

Если в строке меньше трех символов, вместо значения (ord(третья буква) - ord(A) + 1) подставляется 0. Например, строка FOX кодируется так:

27^2 * (ord(F) - ord(A) + 1) + 27 * (ord(O) - ord(A) + 1) + (ord(X) - ord(A) +1) = 4803

Строка NO кодируется следующим образом:

27^2 * (ord(N) - ord(A) + 1) + 27 * (ord(O) - ord(A) + 1) + (0) = 10611

Заметим, что 10611 больше 4803, поскольку NO > FOX.

ord(A) = 65 ; // ASCII код для символа "A".

1. Сортировка выбором

Сортировка выбором (selection sort) — **простой алгоритм** сортировки, является **неустойчивым.** На массиве из n элементов имеет **время выполнения в худшем, среднем и лучшем случае O(n^2),** (предполагается, что сравнения осуществляются за постоянное время).

Идея состоит в

- **поиске наименьшего (наибольшего) элемента** в списке, **который затем меняется местами с элементом в начале (в конце) списка**.
- Далее находится наименьший (наибольший) элемент из оставшихся, и меняется местами со вторым элементом (предпоследним).
- Процесс продолжается до тех пор, пока все элементы не займут свое конечное положение.

Усовершенствование алгоритма

Использовать **двунаправленный вариант сортировки методом выбора**, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

1. Сортировка выбором

Вычислительная сложность алгоритма

При поиске i-го наибольшего (наименьшего) элемента алгоритму приходится перебрать n-i элементов, которые еще не заняли свое конечное положение. Время выполнения алгоритма пропорционально (n - 1) + (n - 2) + … + 1=n*n/2, или порядка O(n^2).

Особенности алгоритма

- Сортировка выбором **неплохо работает со списками, элементы в которых расположены случайно или в прямом порядке**, но несколько **хуже**, если список изначально отсортирован **в обратном порядке.**
- **Алгоритм чрезвычайно прост**. Это не только облегчает его разработку и отладку,
- но и делает сортировку выбором **достаточно быстрой для задач небольшой размерности.**

1. Сортировка выбором

Пример упорядочения по возрастанию

СОРТИРОВКА ПОСРЕДСТВОМ ПРОСТОГО ВЫБОРА

503 087 512 061 **908** 170 **897** 275 653 426 154 509 612 677 **765** **703**|
503 087 512 061 703 170 **897** 275 653 426 154 509 612 677 **765**|908
503 087 512 061 703 170 **765** 275 653 426 154 509 612 **677**|897 908
503 087 512 061 **703** 170 **677** 275 **653** 426 154 509 **612**|765 897 908
503 087 512 061 612 170 **677** 275 **653** 426 154 **509**|703 765 897 908
503 087 512 061 612 170 509 275 **653** **426** **154**|677 703 765 897 908
...
061|087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

2. Сортировка вставкой

Идея сортировки вставками:

очередная запись вставляется среди ранее отсортированных записей.

К i-у шагу (i=2,…,n) первые i-1 записей исходного массива отсортированы. Если запись k(i) < k(i-1) (в случае, когда массив сортируется по возрастанию), запись k(i) необходимо вставить в отсортированную часть массива, не нарушая упорядочивания.

1. Запись k(i) сохраняется во вспомогательной переменной temp.
2. Последовательно сравниваются k(j), j= i-1, …, 1, со значением temp, пока k(j) > temp, при этом записи k(j+1) присваивается значение k(j). В результате находится такое j, что либо k(j) <= temp, либо j=0 (значение temp меньше любого k(j)).
3. Записи k(j+1) присваивается значение temp.

i-й шаг алгоритма завершен.

2. Сортировка вставкой

Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. **Обычно** (и с целью получения устойчивого алгоритма сортировки), **элементы вставляются по порядку их появления во входном массиве**. С целью сохранения **естественности** алгоритма поиск места для вставки в упорядоченной части массива осуществляется от конца этой части к началу.

Алгоритм может сортировать список по мере его получения.

На каждом шаге алгоритма

- **выбираем** один из элементов входных данных и
- **вставляем** его на нужную позицию в уже отсортированном списке,
- до тех пор, пока набор входных данных не будет исчерпан.

2. Сортировка вставкой

Сортировка вставкой (insertion sort) — алгоритм со **сложностью порядка O(n^2).**

Особенности алгоритма

- **прост** в реализации;
- **эффективен на небольших наборах данных**, на наборах данных до десятков элементов может оказаться лучшим;
- **эффективен на наборах данных, которые уже частично отсортированы**;
- устойчивый алгоритм сортировки (**не меняет порядок элементов, которые уже отсортированы**);
- не требует временной памяти, даже под стек;
- **может сортировать список по мере его получения.**

2. Сортировка вставкой

Пример

ПРИМЕР СОРТИРОВКИ МЕТОДОМ ПРОСТЫХ ВСТАВОК

```
^503 : 087
 087  503 ^: 512
^087  503  512 : 061
 061  087  503  512 ^: 908
 061  087 ^503  512  908 : 170
 061  087  170  503  512 ^908 : 897
   . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 061  087  154  170  275  426  503  509  512  612  653  677 ^765  897  908 : 703
 061  087  154  170  275  426  503  509  512  612  653  677  703  765  897  908
```

2. Сортировка вставкой

Полное число шагов, которые потребуется выполнить, составляет 1 + 2 + 3 + … + (n - 1), то есть O(n^2). Фактически, этот алгоритм не слишком быстр даже в сравнении с другими алгоритмами порядка O(n^2), такими как сортировка выбором.

Достаточно много времени тратится на поиск правильного положения для нового элемента.

Усовершенствование алгоритма

Применение эффективного алгоритма поиска (в уже отсортированной части массива!) ускоряет выполнение алгоритма сортировки вставкой.

Пример. Отсортировать массив 6, 1, 8, 2, 5, 3 методом вставки.

Вставка в связных списках

Можно использовать вариант сортировки вставкой для упорядочения элементов не в массиве, а в связном списке. Алгоритм ищет требуемое положение элемента в возрастающем связном списке, и затем помещает туда новый элемент, используя операции работы со связными списками.

Наилучший случай для этого алгоритма достигается, когда исходный список первоначально отсортирован в обратном порядке. В этом случае каждый последующий элемент меньше, чем предыдущий, поэтому алгоритм помещает его в начало отсортированного списка, при условии, что *список просматривается с начала*. При этом требуется выполнить только одну операцию сравнения элементов, и **в наилучшем случае время выполнения алгоритма будет порядка O(n).**

В **наихудшем и среднем случаях** вычислительная сложность алгоритма **порядка O(n^2).**

Преимущество использования связных списков для вставки в том, что при этом *перемещаются только указатели, а не сами записи данных*. **Передача указателей может быть быстрее, чем копирование записей целиком, если элементы представляют собой большие структуры данных.**

3. Сортировка пузырьком

Сортировка простыми обменами, сортировка пузырьком (***bubble sort***) — **простой** алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но **эффективен он лишь для небольших массивов**. Сложность алгоритма: $O(n^2)$.

Алгоритм

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. За каждый проход

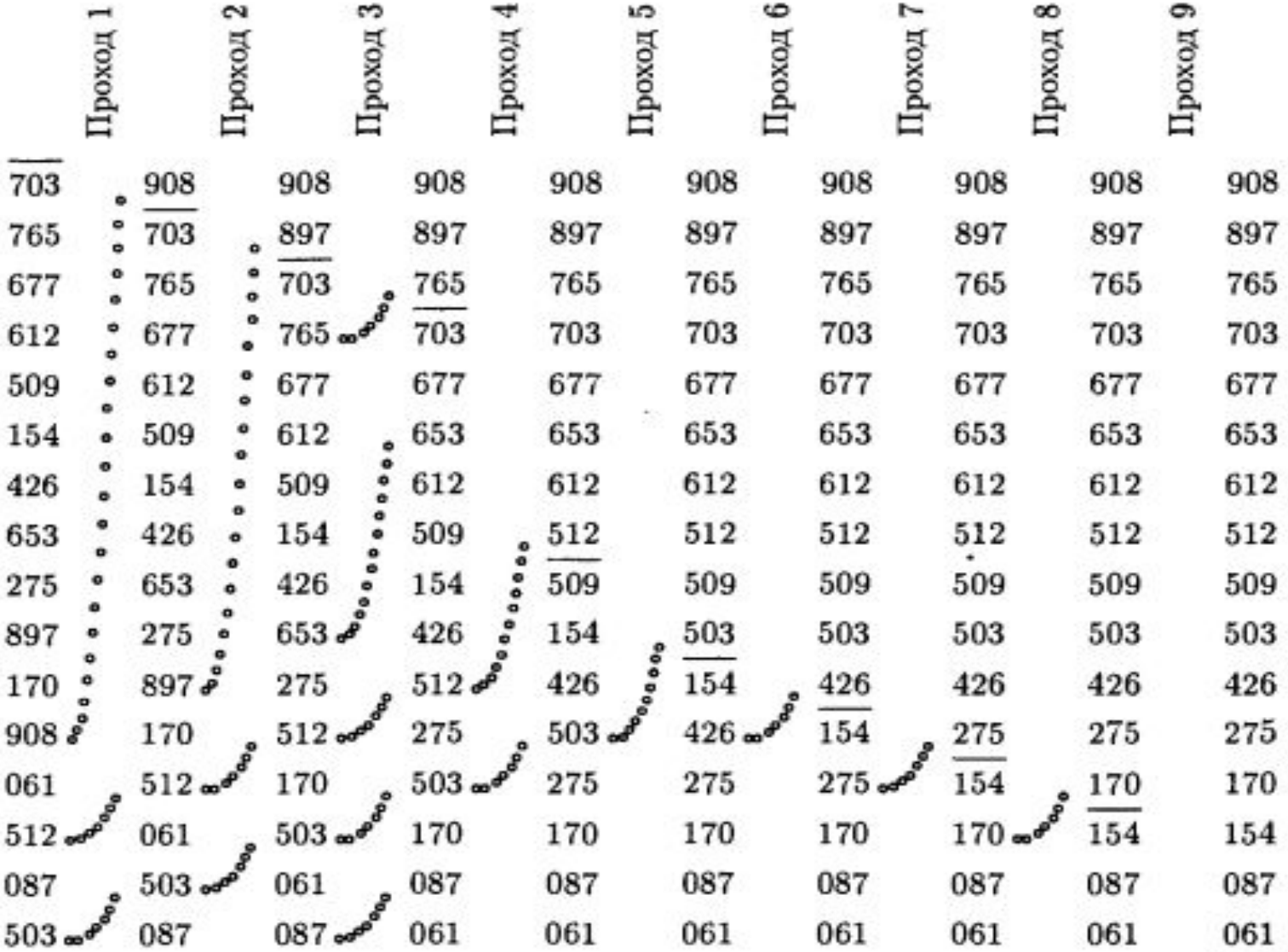
- **элементы последовательно сравниваются попарно** и, если порядок в паре неверный, **выполняется обмен** элементов.
- Проходы по массиву повторяются до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

При проходе алгоритма от начала к концу массива, больший элемент, стоящий не на своём месте, «всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма. В противоположность «методу погружения» (методу простых вставок), в котором элементы погружаются на соответствующий уровень.

3. Сортировка пузырьком

Пример. Сортировка пузырьком

Последовательность чисел представлена вертикально: первый элемент – внизу, последний – вверху.



Процесс сортировки методом пузырька.

Усовершенствование алгоритма сортировки пузырьком (переход к шейкерной сортировке)

1.При просмотре массива снизу вверх(от начала к концу при упорядочении по возрастанию (неубыванию)), **элементы, которые перемещаются вниз (в направлении, противоположном проходу), сдвигаются всего на одну позицию.**

Элементы, которые перемещаются вверх, то есть *в направлении прохода*, сдвигаются в общем случае на несколько позиций за один проход.

Этот факт можно использовать для ускорения работы алгоритма.

Вместо выполнения нескольких отдельных перестановок, **можно сохранить перемещаемое значение во временной переменной** до тех пор, пока не будет найдено конечное положение элемента.

Это может сэкономить большое число шагов алгоритма, если элементы перемещаются на большие расстояния внутри массива.

Усовершенствование алгоритма сортировки пузырьком (переход к шейкерной сортировке)

2. При просмотре массива **сверху вниз** (от конца к началу), **элементы, перемещаемые вверх (в направлении, противоположном обходу), сдвигаются всего на одну позицию. Элементы, которые перемещаются вниз, то есть *в направлении прохода*, сдвигаются на несколько позиций за один проход.** Этот факт можно использовать для ускорения работы алгоритма.

Если чередовать просмотр массива снизу вверх и сверху вниз, то перемещение элементов в прямом и обратном направлениях будет одинаково быстрым.

Если m элементов списка расположены не на своих местах, алгоритму потребуется не более m проходов для того, чтобы расположить элементы по порядку. Если в списке n элементов, алгоритму потребуется n шагов для каждого прохода. Таким образом, **полное время выполнения** для этого алгоритма будет **порядка O(m * n).**

Усовершенствование алгоритма сортировки пузырьком (переход к шейкерной сортировке)

3. **Ограничение проходов массива**. После просмотра массива, ***последние переставленные*** **элементы ограничивают часть массива, которая содержит неупорядоченные элементы.** При проходе, например, наибольший элемент перемещается в конечное положение. Поскольку нет бо́льших его элементов, которые необходимо поместить за ним, то можно начать очередной проход сверху вниз с этой позиции и на ней же заканчивать следующие проходы снизу вверх.

4. Шейкерная сортировка

Сортировка перемешиванием (Шейкерная сортировка) (Cocktail sort) — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент “всплывает” на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

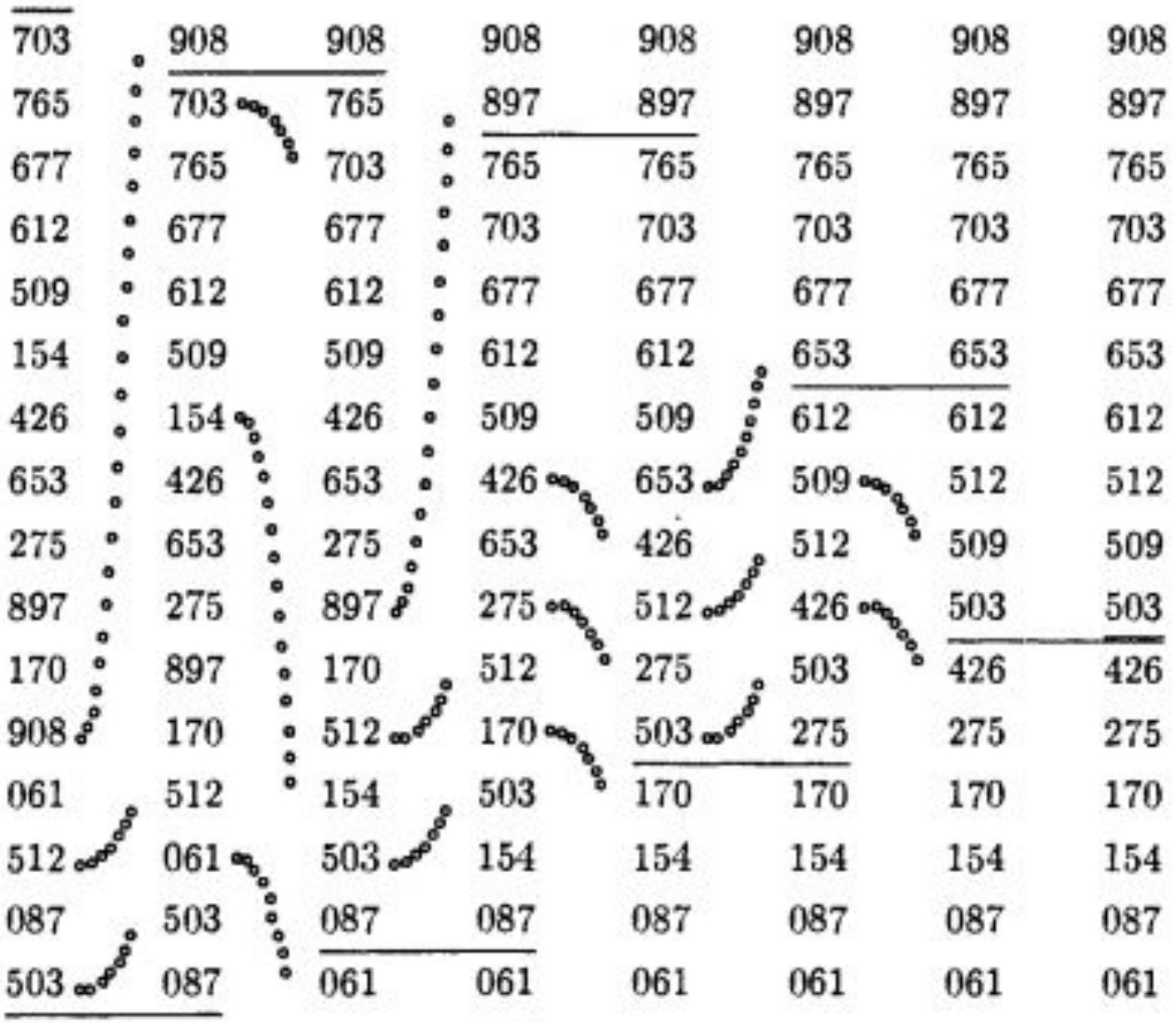
Эти две идеи приводят к следующим ***модификациям в методе пузырьковой сортировки***.

- Границы рабочей части массива (т.е. части массива, где происходит движение) устанавливаются **в месте последнего обмена** на каждой итерации.
- Массив просматривается поочередно слева направо и справа налево.
- Перемещаемое значение хранится в буфере памяти.

Лучший случай для этой сортировки — отсортированный массив (O(n)),
худший — отсортированный в обратном порядке ($O(n^2)$).

4. Шейкерная сортировка

Пример. Шейкерная сортировка



Шейкер-сортировка.

4. Шейкерная сортировка

```
procedure TForm1.Bubblesort(list1: PlongIntarray; min, max: longint);
var
    i, j, tmp, last_swap: longint;
begin
  while min<max do
  begin     //погружение
    last_swap:= min-1;
    i:=min+1;
    while i<=max do
    begin       //нахождение пузырька
      if list1[i-1] > list1[i] then
```

4. Шейкерная сортировка

```
begin
      //куда сдвинуть пузырек
      tmp:=  list1[i-1];
      j:=i;
      repeat
        list1[j-1]:=list1[j] ;
        //'погружение’
        j:=j+1;
        if j>max then break;
      until list1[j]>tmp;
      list1[j-1]:=tmp;
      // 'погружение’ ;
       last_swap:=j-1;
      i:=j+1;
    end else
      i:=i+1;
  end;   //конец погружения
```

4. Шейкерная сортировка

```
//обновление max
  max:= last_swap-1;
  //всплытие
  last_swap:= max+1;
  i:=max-1;
  while i>=min do
  begin       //нахождение пузырька
    if list1[i+1] < list1[i]then
    begin         //куда сдвинуть пузырек
      tmp:=  list1[i+1];
      j:=i;
      repeat
        list1[j+1]:=list1[j] ;
        j:=j-1;
        if j<min then break;
      until list1[j]<tmp;
```

4. Шейкерная сортировка

```
list1[j+1]:=tmp;
        last_swap:=j+1;
        i:=j-1;
      end else
        i:=i-1;
    end;  //конец всплытия
    //обновление min
    min:=last_swap+1;
  end;
end;
```

5. Быстрая сортировка

Быстрая сортировка — рекурсивный алгоритм, который использует подход «разделяй и властвуй». Если сортируемый массив имеет больше, чем минимальный заданный размер, алгоритм быстрой сортировки разбивает его на два подмассива, а затем рекурсивно вызывает себя для сортировки этих двух подмассивов независимо друг от друга.

Суть алгоритма заключается в процессе разбиения массива элементом key на два подмассива таким образом, что выполняются условия:

1. Элемент k(p)= key для некоторого p занимает окончательную позицию в массиве.
2. Каждый из элементов k(0), …, k(p-1) не превышает значения k(p).
3. Каждый из элементов k(p+1), …, k(n-1) не меньше значения k(p).

Полная сортировка достигается путем деления массива на подмассивы с последующим применением к ним этого же метода. Если алгоритм вызывается для подмассива, содержащего не более одного элемента, то подмассив уже отсортирован, и алгоритм для данного подмассива завершает работу.

5. Быстрая сортировка

Описание рекурсивного алгоритма

Пусть текущий подмассив содержит элементы

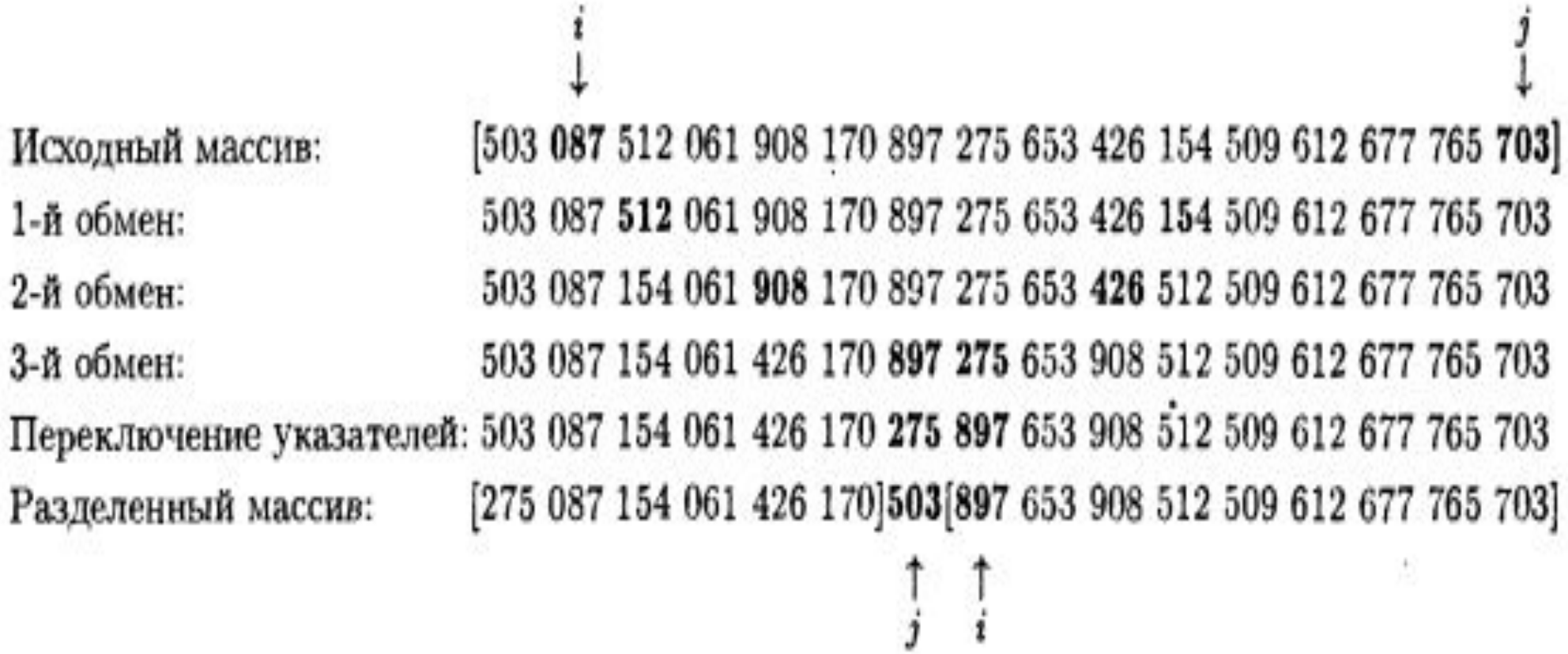
k(left), k(left+1),…, k(right).

- Полагаем x=k(left), i=left+1, j=right.
- Просматриваем массив с левого конца, увеличивая i на 1, до тех пор, пока не будет найден элемент, бо́льший или равный x.
- Просматриваем массив с правого конца, уменьшая j на 1, до тех пор, пока не будет найден элемент меньше x.
- Элементы k(i) и k(j) меняются местами, если i<=j. i=i+1. j=j-1.
- Когда i станет больше j, меняем местами k(j) и k(left).
- В результате k(j) окажется на своем месте, а подмассив разделится на две части: k(left), k(left+1),…, k(j-1) и k(j+1), k(j+2) …, k(right). Рекурсивный алгоритм продолжается для этих подмассивов.
- Условие выхода из рекурсии: left=right.

5. Быстрая сортировка

Пример. Быстрая сортировка

Предполагается, что первый элемент принадлежит первому подмассиву. После разделения на 2 подмассива последний элемент первого подмассива и его первый элемент меняются местами.



5. Быстрая сортировка

Особенности версии алгоритма

1. В качестве **разделителя** используется **первый элемент в списке**.
2. После разбиения массива на два подмассива разделяющий элемент *не входит ни в один подмассив*. Это означает, что в двух подмассивах содержится на один элемент меньше, чем в исходном массиве. Т.к. число рассматриваемых элементов уменьшается, то **в конечном итоге алгоритм завершит работу**.

Пример. Рассмотрите работу алгоритма быстрой сортировки на примере массива из 12 элементов 75, 80, 90, 5, 10, 4, 60, 2, 15, 0, 85, 65.

5. Быстрая сортировка

Способы выбора разделяющего элемента

1. **Можно использовать элемент из середины списка**. Но он может оказаться наименьшим или наибольшим элементом списка. При этом один подсписок будет намного больше, чем другой, что приведет к снижению производительности до порядка O(n^2) и глубокому уровню рекурсии.

2. **Просмотреть весь список, вычислить среднее арифметическое всех значений**, и использовать его в качестве разделительного значения. Дополнительный проход со сложностью порядка O(n) не изменит теоретическое время выполнения алгоритма, но снизит общую производительность.

3. Выбрать **средний (медианный) из элементов в начале, конце и середине списка**. Потребуется выбрать всего три элемента. Гарантируется, что этот элемент не является наибольшим или наименьшим в списке, и вероятно окажется где-то в середине списка.

4. Выбор **среднего элемента из списка случайным образом**.

5. Быстрая сортировка

Особенности алгоритма быстрой сортировки

- Если **данные имеют небольшой диапазон значений** (много дубликатов нескольких значений), то алгоритм при каждом вызове будет помещать много идентичных значений в один список. Это **приводит к большому уровню вложенности рекурсии**.
- Быстрая сортировка — **не самый лучший выбор для сортировки небольших списков**. Благодаря своей простоте, сортировка выбором быстрее при сортировке примерно десятка записей.

Усовершенствование алгоритма

Можно **улучшить производительность быстрой сортировки**, если прекратить рекурсию до того, как подсписки уменьшатся до нуля, и **использовать для завершения работы сортировку выбором**.

5. Быстрая сортировка

Быстрая сортировка (quicksort), сортировка Хоара, часто называемая qsort по имени реализации в стандартной библиотеке языков — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром. Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем O(n log n) обменов при упорядочении n элементов), хотя и имеющий ряд недостатков.

Краткое описание алгоритма

- выбрать элемент, называемый опорным.
- сравнить все остальные элементы с опорным, на основании сравнения разбить множество на *три* — «меньшие опорного», «равные опорному» и «большие опорного», расположить их в порядке меньшие-равные-большие.
- повторить рекурсивно для «меньших» и «больших».

5. Быстрая сортировка

```
void Q_sort_help(Iter left, Iter right) {
    if (right - left <= 1) return;
    auto separator = *(left + (right - left) / 2);
    auto t_left = left;
    auto t_right = right - 1;
    while (t_left < t_right) {
        while (*t_left < separator) ++t_left;
        while (*t_right > separator) --t_right;
        if (t_left <= t_right) {
            std::swap(*t_left, *t_right);
            ++t_left;
            --t_right;
        }
        if (left < t_right) Q_sort_help(left, t_right + 1);
        if (t_left < right) Q_sort_help(t_left, right);
    }
}
```

6. Сортировка слиянием

Сортировка слиянием (merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Для сортировки массива эти три этапа выглядят так:

- Сортируемый **массив разбивается на две части примерно одинакового размера.**
- **Каждая из получившихся частей сортируется отдельно**, например — тем же самым алгоритмом.
- **Два упорядоченных массива половинного размера соединяются в один**.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

6. Сортировка слиянием

Этап слияния

Подсписки сливаются во **временный массив**, и **результат копируется в первоначальный список**. Работа с временным массивом приводит к тому, что бо́льшая часть времени уходит на копирование элементов между массивами.

Пример. Слияние двух упорядоченных массивов

$$\begin{cases} 503\ 703\ 765 \\ 087\ 512\ 677 \end{cases},$$

$$087 \begin{cases} 503\ 703\ 765 \\ 512\ 677 \end{cases},$$

$$087\ 503 \begin{cases} 703\ 765 \\ 512\ 677 \end{cases}$$

$$087\ 503\ 512 \begin{cases} 703\ 765 \\ 677 \end{cases}$$

6. Сортировка слиянием

Усовершенствование алгоритма

Как и в случае с быстрой сортировкой, можно ускорить выполнение сортировки слиянием, остановив рекурсию, когда подсписки достигают определенного минимального размера. Затем можно использовать сортировку выбором для завершения работы.

Преимущества алгоритма сортировки слиянием

- **Время работы алгоритма сортировки слиянием остается одним и тем же для различных представлений данных и начального распределения.** Его **можно использовать** и в случае, когда в списке имеется **много дублированных** значений.
- Поскольку **сортировка слиянием** (простого двухпутевого) делит список на равные части, она **никогда не входит в глубокую рекурсию**. Для списка из n элементов сортировка слиянием достигает **глубины рекурсии** всего O(log n).
- Время работы алгоритма сортировки слиянием порядка O(n*log n) (быстрая сортировка тоже алгоритм порядка O(n*log n), но только для лучшего случая).
  Расход памяти выше, чем для быстрой сортировки.

6. Сортировка слиянием

Пример. Рекурсивный алгоритм простого двухпутевого слияния. Используются дополнительные массивы

9 2 5 4 | 7 1 8 0

9 2 | 5 4 | 7 1 | 8 0

9 | 2 | 5 | 4 | 7 | 1 | 8 | 0

2 9 | 4 5 | 1 7 | 0 8

2 4 5 9 | 0 1 7 8

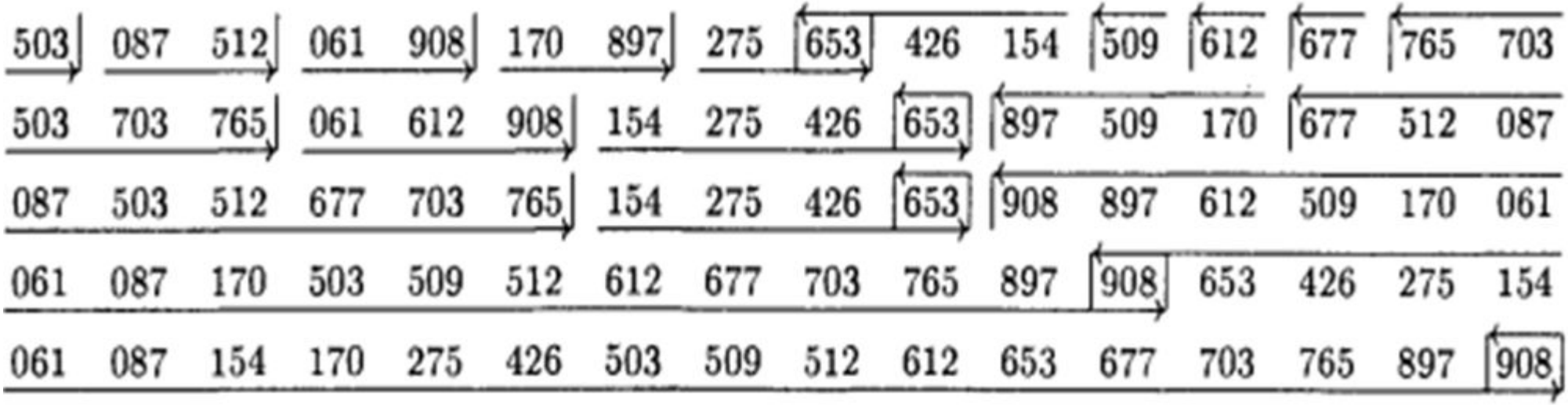
0 1 2 4 5 7 8 9

6. Сортировка слиянием

Пример. Не рекурсивный алгоритм. Сортировка методом простого двухпутевого (сбалансированного) слияния.

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	512	677	509	908	426	897	653	275	170	154	612	061	765	087
087	503	703	765	154	170	509	908	897	653	426	275	677	612	512	061
061	087	503	512	612	677	703	765	908	897	653	509	426	275	170	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Пример. Не рекурсивный алгоритм. Сортировка методом естественного двухпутевого слияния.



В обоих случаях используется дополнительный массив той же размерности, что и исходный.

6. Сортировка слиянием

Рекурсивный алгоритм

```
void my_sort::merge_sort(int l, int r)
{
        int i_mid, i1, i2, i3;
        if (l < r)
        {
//этап распределения
           i_mid = (l + r)/2;
           merge_sort(l,i_sred);
           merge_sort(i_sred + 1, r);
        // этап слияния
        int *tmp_mas = new int[r-l+1];  // буфер для слияния
           i1 = l;          // начало первого подмассива в arr
           i2 = i_mid + 1;   // начало второго подмассива в arr
           i3 = 0;          // начало буфера
```

6. Сортировка слиянием

Рекурсивный алгоритм

```
while (i1 <= i_mid && i2 <= r)
        if (arr[i1] < arr[i2])
        {
            tmp_mas[i3] = arr[i1];
            i1++;
            i3++;
        }
        else
        {
            tmp_mas[i3] = arr[i2];
            i2++;
            i3++;
        }
```

6. Сортировка слиянием

Рекурсивный алгоритм

```
        while(i2 <= r)
                {
                        tmp_mas[i3] = arr[i2];
                        i2++;
                        i3++;
                }
        while(i1 <= i_mid)
                {
                        tmp_mas[i3] = arr[i1];
                        i1++;
                        i3++;
                }
          for (i3 = 0; i3 < r-l+1; i3++)
                arr[i3+l] = tmp_mas[i3];
          delete []tmp_mas;
        }
}
```