**14. АЛГОРИТМЫ СОРТИРОВКИ ДАННЫХ**

Оглавление

[§14.1 Внутренняя сортировка данных 1](#_Toc62412546)

[§14.2 Простейшие методы сортировки: метод обмена 2](#_Toc62412547)

[§14.3 Простейшие методы сортировки: метод вставок 5](#_Toc62412548)

[§14.4 Простейшие методы сортировки: метод выбора 7](#_Toc62412549)

[§14.5 Улучшенные методы сортировки: метод Шелла 10](#_Toc62412550)

[§14.6 Улучшенные методы сортировки: метод быстрой сортировки 13](#_Toc62412551)

[§14.7 Улучшенные методы сортировки: поразрядная сортировка 17](#_Toc62412552)

## §14.1 Внутренняя сортировка данных

Пусть задано некоторое множество элементов а1, а2, а3, ..., аn и требуется выстроить эти элементы по порядку в соответствии с заданной функцией предпочтения (например, по алфавиту). Очень часто значения функции предпочтения явно хранятся в исходных элементах в виде специального ключевого поля целого или строкового типа.

Все задачи сортировки делятся на две большие и принципиально различные группы: задачи внутренней и внешней сортировки. Внутренняя сортировка применима тогда, когда все входные данные можно одновременно разместить в оперативной памяти. Возможность такой загрузки определяется 3 факторами: располагаемым размером памяти, числом обрабатываемых элементов, объемом каждого элемента в байтах. Внутренняя сортировка, как правило, реализуется с помощью массивов и поэтому часто называется сортировкой массивов.

Если исходные данные нельзя одновременно разместить в основной памяти, то приходится использовать дисковую память и алгоритмы обработки файлов. Такая сортировка называется внешней или сортировкой файлов и будет рассмотрена позже. Пока остановимся на задаче сортировки массивов.

Все алгоритмы сортировки основаны на многократном повторении двух базовых операций: сравнение ключей у двух элементов и перестановка двух элементов. Подсчет именно этих операций лежит в основе методов оценивания трудоемкости алгоритмов сортировки.

Методы сортировки массивов можно разделить на две большие группы:

• Универсальные методы, не требующие никакой дополнительной информации об исходных данных и выполняющие сортировку «на месте», т.е. без использования больших объемов дополнительной памяти (например – для размещения копии исходного массива); такие методы в лучшем случае дают оценку трудоемкости порядка

• Специальные методы, которые либо за счет некоторой дополнительной информации об исходных данных, либо за счет использования большой дополнительной памяти позволяют получить более высокую производительность сортировки порядка O(n) (примеры – карманная и поразрядная сортировка)

В свою очередь, универсальные методы сортировки делятся на две подгруппы:

• Простейшие методы с трудоемкостью порядка : сортировка обменом, сортировка выбором и сортировка вставками

• Улучшенные методы с трудоемкостью : метод Шелла, пирамидальная сортировка, быстрая сортировка

Возникает справедливый вопрос: зачем нужны простейшие методы, если есть более быстрые улучшенные методы? Как ни странно, возможны ситуации, когда простейшие методы оказываются лучше улучшенных. Подобные ситуации уже были упомянуты выше: если надо очень много (сотни тысяч или миллионы) раз повторить сортировку весьма небольших массивов (несколько десятков элементов), то использование простейших методов может дать некоторый выигрыш, поскольку компонента оценочной функции при малых n не оказывает решающего влияния на общий результат. Кроме того, простейшие методы сортировки имеют исключительно простую и понятную программную реализацию, что далеко не всегда можно сказать об улучшенных методах.

## §14.2 Простейшие методы сортировки: метод обмена

Данный метод относится к классу простейших, занимая в нем последнее место по производительности. Тем не менее, он очень широко известен, видимо, благодаря своему одному легко запоминающемуся названию – метод **всплывающего пузырька**. Работа алгоритма действительно похожа на всплывание наверх пузырьков воздуха: сначала на самый верх всплывает самый легкий элемент, потом за ним – чуть более тяжелый и т.д.

Пусть имеется n элементов а1 а2, а3, . . ., аn, расположенных в ячейках массива. Для простоты будем считать, что сам элемент совпадает с его ключом. Алгоритм состоит в повторении n-1 шага, на каждом из которых в оставшемся необработанном наборе за счет попарного сравнения соседних элементов отыскивается минимальный элемент.

Шаг 1. Сравниваем аn  с аn-1  и если аn < аn-1 то меняем их местами, потом сравниваем аn-1  с аn-2 и, возможно, переставляем их, сравниваем аn-2 и аn-3 и т.д. до сравнения и, возможно, перестановки а2 и а1. В результате на первом месте в массиве оказывается самый минимальный элемент, который в дальнейшей сортировке не участвует

Шаг 2. Аналогично сравниваем аn с аn-1, аn-1 с аn-2 и т.д., а3 с а2, в результате чего на месте а2 оказывается второй наименьший элемент, который вместе с а1 образует начальную часть упорядоченного массива

Шаг 3. Аналогичными сравнениями и перестановками среди элементов а3, а4, …, аn находится наименьший, который занимает место а3

. . . . .

Шаг n-1. К этому моменту первые n-2 элемента в массиве уже упорядочены и остается “навести порядок” только между двумя последними элементами аn-1 и аn. На этом сортировка заканчивается.

Пример. Дано 6 элементов – целые числа 15, 33, 42, 07, 12, 19.

|  | а1 | а2 | а3 | а4 | а5 | а6 | Выполняемые операции |
| --- | --- | --- | --- | --- | --- | --- | --- |
| шаг 1 | 15 | 33 | 42 | 07 | 12 | 19 | сравнение 19 и 12, обмена нет |
| 15 | 33 | 42 | 07 | 12 | 19 | сравнение 12 и 07, обмена нет |
| 15 | 33 | 07 | 42 | 12 | 19 | сравнение 07 и 42, меняем их |
| 15 | 07 | 33 | 42 | 12 | 19 | сравнение 07 и 33, меняем их |
| 07 | 15 | 33 | 42 | 12 | 19 | сравнение 07 и 15, меняем их; 07 - наименьший |
| шаг 2 | **07** | 15 | 33 | 42 | 12 | 19 | сравнение 19 и 12, обмена нет |
| **07** | 15 | 33 | 12 | 42 | 19 | сравнение 12 и 42, меняем их |
| **07** | 15 | 12 | 33 | 42 | 19 | сравнение 12 и 33, меняем их |
| **07** | 12 | 15 | 33 | 42 | 19 | сравнение 12 и 15, меняем их, 12 –второй наим. |
| шаг 3 | **07** | **12** | 15 | 33 | 19 | 42 | сравнение 19 и 42, меняем их |
| **07** | **12** | 15 | 19 | 33 | 42 | сравнение 19 и 33, меняем их |
| **07** | **12** | 15 | 19 | 33 | 42 | сравнение 19 и 15, обмена нет, 15 – третий наим. |
| шаг 4 | **07** | **12** | **15** | 19 | 33 | 42 | сравнение 42 и 33, обмена нет |
| **07** | **12** | **15** | 19 | 33 | 42 | сравнение 33 и 19, обмена нет, 19 – четвертый элем. |
| шаг 5 | **07** | **12** | **15** | **19** | 33 | 42 | сравнение 42 и 33, обмена нет, сортировка закончена |
|  | **07** | **12** | **15** | **19** | **33** | **42** |  |

Итого, для шести элементов сделано 5+4+3+2+1 = 15 сравнений и 8 перестановок.

В общем случае, на каждом из шагов выполняется в среднем сравнений, поэтому оценка для числа сравнений выражается соотношением , т.е. данный метод относится к классу . Аналогично, число перестановок тоже пропорционально n2. Несмотря на то, что было предложено несколько улучшений данного метода (есть очень красивые названия – например, шейкер-сортировка), он остается самым неэффективным. Уже для 1000 элементов число сравнений выражается внушительной величиной порядка 500 тысяч.

Программная реализация включает двойной цикл: внешний реализует основные шаги алгоритма, внутренний сравнивает и переставляет элементы, начиная с конца массива.

Пример программной реализации сортировки методом обмена представлен в листинге 14.1.

**Листинг 14.1 – Сортировка методом обмена**

|  |  |
| --- | --- |
| 1 | static int[] BubbleSort(int[] mas) |
| 2 | { |
| 3 | int temp; |
| 4 | for (int i = 0; i < mas.Length; i++) |
| 5 | { |
| 6 | for (int j = i + 1; j < mas.Length; j++) |
| 7 | { |
| 8 | if (mas[i] > mas[j]) |
| 9 | { |
| 10 | temp = mas[i]; |
| 11 | mas[i] = mas[j]; |
| 12 | mas[j] = temp; |
| 13 | } |
| 14 | } |
| 15 | } |
| 16 | return mas; |
| 17 | } |

Мы создаём функцию BubbleSort. В неё будет передан массив mas, который мы заполним числами для сортировки.

Здесь мы сравниваем, так сказать, предыдущий элемент (i) с последующим (j).

Если элемент массива под номером i будет больше, чем элемент массива под номером j, то меняем элементы местами и продолжаем сравнение дальше, как в алгоритме.

Для изменения «направления» сортировки, где меньший элемент будет в конце массива, а больший в начале, надо лишь поменять строку

if (mas[i] > mas[j])

на

if (mas[i] < mas[j]).

Данная функция возвращает отсортированный массив mas (строка 16)

## §14.3 Простейшие методы сортировки: метод вставок

Данный метод также относится к классу простейших, но по сравнению с методом пузырька имеет немного лучшие показатели.

Пусть имеется n элементов а1 а2, а3, ..., аn, расположенных в ячейках массива. Сортировка выполняется за шаг, причем шаги удобно нумеровать от 2 до n. На каждом i-ом шаге обрабатываемый набор разбивается на 2 части:

* левую часть образуют уже упорядоченные на предыдущих шагах элементы а1, а2, а3, . . ., аi-1
* правую часть образуют еще не обработанные элементы аi, аi+1, аi+2, . . ., аn

На шаге для элемента аi находится подходящее место в уже отсортированной последовательности. Поиск подходящего места выполняется поэлементными сравнениями и перестановками по необходимости: сравниваем аi с аi-1, если аi < аi-1, то переставляем их, потом сравниваем аi-1 с аi-2 и т. д. Сравнения и, возможно, перестановки продолжаются до тех пор, пока не будет выполнено одно из 2-х следующих условий:

* в отсортированном наборе найден элемент, меньший аi (все остальные не просмотренные элементы будут еще меньше)
* достигнут первый элемент набора а1, что произойдет в том случае, если аi меньше всех элементов в отсортированном наборе и он должен занять первое место в массиве

**Пример**. Возьмем тот же исходный набор целых чисел: 15-33-42-07-12-19

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **а1** | **а2** | **а3** | **а4** | **а5** | **а6** | **Выполняемые операции** |
| шаг 2 | **15** | 33 | 42 | 07 | 12 | 19 | сравнение 15 и 33, обмена нет, 15 – пока первый |
| шаг 3 | **15** | **33** | 42 | 07 | 12 | 19 | сравнение 33 и 42, обмена нет, 15 и 33 пока первые |
| шаг 4 | **15** | **33** | **07** | 42 | 12 | 19 | сравнение 07 и 42, меняем их |
| **15** | **07** | **33** | 42 | 12 | 19 | сравнение 07 и 33, меняем их |
| **07** | **15** | **33** | 42 | 12 | 19 | сравнение 07 и 15, меняем их; 07-15-33 пока первые |
| шаг 5 | **07** | **15** | **33** | **12** | 42 | 19 | сравнение 12 и 42, меняем их |
| **07** | **15** | **12** | **33** | 42 | 19 | сравнение 12 и 33, меняем их |
| **07** | **12** | **15** | **33** | 42 | 19 | сравнение 12 и 15, меняем их |
| **07** | **12** | **15** | **33** | 42 | 19 | сравнение 12 и 07, обмена нет, пока: 07-12-15-33 |
| шаг 6 | **07** | **12** | **15** | **33** | **19** | 42 | сравнение 19 и 42, меняем их |
| **07** | **12** | **15** | **19** | **33** | 42 | сравнение 19 и 33, меняем их |
| **07** | **12** | **15** | **19** | **33** | 42 | сравнение 19 и 15, обмена нет, все готово |

Для данного примера было сделано 12 сравнений и 8 перестановок, что чуть лучше предыдущего метода. В среднем, число сравнений по данному методу примерно в 2 раза меньше, чем в методе пузырька, оставаясь тем не менее пропорциональным величине n2. Наилучший результат этот метод показывает для уже упорядоченного исходного массива – всего сравнение.

Программная реализация включает два вложенных цикла, но в отличие от предыдущего метода, внутренний цикл реализуется как **while** с возможностью остановки при обнаружении меньшего элемента.

Пример программной реализации сортировки методом вставок представлен в листинге 14.2.

**Листинг 14.2 – Сортировка методом вставок**

|  |  |
| --- | --- |
| 1 | void InsertionSort(int n, int[] mass) |
| 2 | { |
| 3 | int newElement, location; |
| 4 | for (int i = 1; i < n; i++) |
| 5 | { |
| 6 | newElement = mass[i]; |
| 7 | location = i - 1; |
| 8 | while(location >= 0 && mass[location] > newElement) |
| 9 | { |
| 10 | mass[location + 1] = mass[location]; |
| 11 | location = location - 1; |
| 12 | } |
| 13 | mass[location + 1] = newElement; |
| 14 | } |
| 15 | } |

## §14.4 Простейшие методы сортировки: метод выбора

Данный метод из группы простейших имеет лучшие характеристики по числу перестановок, хотя он, как и оба ранее рассмотренных метода, в целом имеет трудоемкость O(n2). Его чуть более лучшие показатели связаны с тем, что в некоторых ситуациях выполняется перестановка не соседних элементов, а отстоящих на некотором расстоянии друг от друга.

Пусть имеется n элементов а1 а2, а3, . . ., аn, расположенных в ячейках массива. Сортировка выполняется за шаг, пронумерованных от до . На каждом i-ом шаге обрабатываемый набор разбивается на 2 части:

* левую часть образуют уже упорядоченные на предыдущих шагах элементы а1, а2, а3, . . ., аi-1
* правую часть образуют еще не обработанные элементы аi, аi+1, аi+2, . . ., аn

Суть метода состоит в том, что в необработанном наборе отыскивается наименьший элемент, который меняется местами с элементом аi. На первом шаге (при i = 1), когда необработанным является весь исходный набор, это сводится к поиску наименьшего элемента в массиве и обмену его с первым элементом. Ясно, что поиск наименьшего элемента выполняется обычным попарным сравнением, но соседние элементы при этом не переставляются, что в целом уменьшает число пересылок.

**Пример**. Возьмем тот же исходный набор целых чисел: 15-33-42-07-12-19

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **а1** | **а2** | **а3** | **а4** | **а5** | **а6** | **Выполняемые операции** |
| шаг 1 | 15 | 33 | 42 | 07 | 12 | 19 | сравнение 15 и 33, min = 15 |
| 15 | 33 | 42 | 07 | 12 | 19 | сравнение 15 и 42, min = 15 |
| 15 | 33 | 42 | 07 | 12 | 19 | сравнение 15 и 07, min = 07 |
| 15 | 33 | 42 | 07 | 12 | 19 | сравнение 07 и 12, min = 07 |
| 15 | 33 | 42 | 07 | 12 | 19 | сравнение 07 и 19, min = 07, обмен 15 и 07 |
| шаг 2 | **07** | 33 | 42 | 15 | 12 | 19 | сравнение 33 и 42, min = 33 |
| **07** | 33 | 42 | 15 | 12 | 19 | сравнение 33 и 15, min = 15 |
| **07** | 33 | 42 | 15 | 12 | 19 | сравнение 15 и 12, min = 12 |
| **07** | 33 | 42 | 15 | 12 | 19 | сравнение 12 и 19, min = 12, обмен 33 и 12 |
| шаг 3 | **07** | **12** | 42 | 15 | 33 | 19 | сравнение 42 и 15, min = 15 |
| **07** | **12** | 42 | 15 | 33 | 19 | сравнение 15 и 33, min = 15 |
| **07** | **12** | 42 | 15 | 33 | 19 | сравнение 15 и 19, min = 15, обмен 42 и 15 |
| шаг 4 | **07** | **12** | **15** | 42 | 33 | 19 | сравнение 42 и 33, min = 33 |
| **07** | **12** | **15** | 42 | 33 | 19 | сравнение 33 и 19, min = 19, обмен 42 и 19 |
| шаг 5 | **07** | **12** | **15** | **19** | 33 | 42 | сравнение 33 и 42, min = 33, обмена нет, все готово |

В данном примере сделано 15 сравнений (как и в методе пузырька), но всего 4 перестановки. Эта особенность сохраняется и в целом: по числу сравнений метод выбора близок к методу пузырька, но по числу перестановок существенно превосходит оба рассмотренные выше методы (оценка числа перестановок n\*log 2 n)

Программная реализация включает в себя внешний цикл, который обрабатывает основные шаги и выполняет перестановку минимального элемента, и внутренний цикл, организующий поиск наименьшего элемента в необработанной части массива.

Пример программной реализации сортировки методом выбора представлен в листинге 14.3.

**Листинг 14.3 – Сортировка методом выбора**

|  |  |
| --- | --- |
| 1 | static int[] ViborSort(int[] mas) |
| 2 | { |
| 3 | for (int i = 0; i < mas.Length - 1; i++) |
| 4 | { |
| 5 | //поиск минимального числа |
| 6 | int min=i; |
| 7 | for (int j = i + 1; j < mas.Length; j++) |
| 8 | { |
| 9 | if (mas[j] < mas[min]) |
| 10 | { |
| 11 | min = j; |
| 12 | } |
| 13 | } |
| 14 | //обмен элементов |
| 15 | int temp = mas[min]; |
| 16 | mas[min] = mas[i]; |
| 17 | mas[i] = temp; |
| 18 | } |
| 19 | return mas; |
| 20 | } |

***Общее заключение*** *по простейшим методам сортировки.*

Метод обмена (пузырька) имеет единственное преимущество – нулевое число пересылок в случае, если исходный набор уже отсортирован в нужном порядке. В остальных случаях все его показатели пропорциональны n2.

Метод вставок также дает хорошие результаты для упорядоченных входных данных (число сравнений и пересылок пропорционально n). Во всех остальных случаях его показатели пропорциональны n2, хотя что касается оценки среднего числа сравнений, то она чуть лучше, чем у других методов. Многочисленные эксперименты показывают, что метод вставок дает **наименьшее** время сортировки среди всех **простейших** методов.

Метод выбора, как это и следовало ожидать, имеет лучшие показатели по числу пересылок, особенно – для общего случая, где оценка О(n\*log2n) заметно лучше оценки O(n2). Поэтому его можно рекомендовать к использованию из всех простейших методов в том случае, если именно число перестановок является наиболее важным.

## §14.5 Улучшенные методы сортировки: метод Шелла

Метод Шелла является улучшенным вариантом метода вставок. Поскольку метод вставок дает хорошие показатели качества для небольших или почти упорядоченных наборов данных, метод Шелла использует эти свойства за счет многократного применения метода вставок.

Алгоритм метода Шелла состоит в многократном повторении двух основных действий:

* **объединение** нескольких элементов исходного массива по некоторому правилу
* **сортировка** этих элементов обычным методом вставок

Более подробно, на первом этапе **группируются** элементы входного набора с **достаточно большим шагом**. Например, выбираются все 1000-е элементы, т.е. создаются группы:

группа 1: 1, 1001, 2001, 3001 и т.д.

группа 2: 2, 1002, 2002, 3002 и т.д.

группа 3: 3, 1003, 2003, 3003 и т.д.

. . . . . . . . . . . . . . . . . . . . .

группа 1000: 1000, 2000, 3000 и т.д.

Внутри каждой группы выполняется **обычная** сортировка вставками, что эффективно за счет **небольшого** числа элементов в группе.

На втором этапе выполняется группировка уже с **меньшим** шагом, например - все сотые элементы. В каждой группе опять выполняется обычная сортировка вставками, которая эффективна за счет того, что после первого этапа в каждой группе набор данных будет уже **частично отсортирован**.

На третьем этапе элементы группируются с еще меньшим шагом, например – все десятые элементы. Выполняется сортировка, группировка с еще меньшим шагом и т.д.

На последнем этапе сначала выполняется группировка с **шагом 1**, создающая единственный набор данных размерности n, а затем - сортировка **практически отсортированного** набора.

**Пример**. Исходный набор: 15 – 33 – 42 – 07 – 12 - 19

Выполняем группировку с шагом 3, создаем три группы по 2 элемента и сортируем каждую из них отдельно:

группа 1: 15 – 07 => 07 – 15 (1 сравнение, 1 пересылка)

группа 2: 33 – 12 => 12 – 33 (1 сравнение, 1 пересылка)

группа 3: 42 – 19 => 19 – 42 (1 сравнение, 1 пересылка)

Новый набор чисел: 07 – 15 – 12 – 33 – 19 – 42

Группировка с меньшим шагом 2 дает 2 группы по 3 элемента, которые сортируются отдельно:

группа 1: 07 – 12 – 19 => уже упорядочена (2 сравнения, 0 пересылок)

группа 2: 15 – 33 – 42 => уже упорядочена (2 сравнения, 0 пересылок)

Новый набор чисел: 07 – 12 – 19 – 15 – 33 – 42

Последняя группировка с шагом 1 дает сам набор чисел; к нему применяется сортировка вставками с 5-ю сравнениями и только одной пересылкой, после чего получаем искомый результат.

Итого – 12 сравнений и 4 пересылки, что в общем-то не лучше, чем у простых методов. Однако, здесь надо учесть два фактора.

Фактор 1 (общий). Улучшенные методы показывают свою эффективность именно для **больших** наборов данных (сотни, тысячи и т.д. элементов). Для очень малых наборов (как в примере) они могут давать даже худшие результаты.

Фактор 2 (специфический). Эффективность метода Шелла существенно зависит от выбора **последовательности шагов** группировки. Эта последовательность обязательно должна быть **убывающей**, а последний шаг обязательно **равен 1**. В настоящее время **неизвестна наилучшая** последовательность шагов, обеспечивающая наименьшую трудоемкость. На основе многочисленных экспериментов установлено, что число шагов группировки надо выбирать по формуле [(log 2 n)] – 1, где скобки [ ] используются для обозначения целой части числа, а в качестве самих последовательностей рекомендуется один из следующих наборов (обращаю внимание: для удобства восприятия шаги даются в **обратном порядке**):

1, 3, 5, 9, 17, 33, . . . (общая формула: tk = (2\* tk-1) –1)

1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767 . . . (общая формула: tk = (2\* tk-1) +1, а еще проще – (2k – 1)).

В соответствии с этими рекомендациями, в предыдущем примере надо взять лишь 2 шага группировки со значениями 3 и 1. В этом случае потребуется лишь 8 сравнений и 5 пересылок.

Что касается программной реализации, то по сравнению с методом вставок потребуется организовать еще один **самый внешний** цикл для выполнения группировок элементов с убывающими шагами. Сами шаги можно вычислять по приведенным выше формулам, а можно хранить в предварительно подготовленном вспомогательном массиве. Никакого выделения сгруппированных элементов в отдельные массивы не производится, вся работа выполняется за счет изменения индексов элементов.

Оценка трудоемкости метода Шелла выражается соотношением O(n1,2), что лучше, чем у простейших методов, особенно при больших .

Пример программной реализации сортировки методом Шелла представлен в листинге 14.4.

**Листинг 14.4 – Сортировка методом Шелла**

|  |  |
| --- | --- |
| 1 | void ShellSort(int n, int mass[]) |
| 2 | { |
| 3 | int i, j, step; |
| 4 | int tmp; |
| 5 | for (step = n / 2; step > 0; step /= 2) |
| 6 | for (i = step; i < n; i++) |
| 7 | { |
| 8 | tmp = mass[i]; |
| 9 | for (j = i; j >= step; j -= step) |
| 10 | { |
| 11 | if (tmp < mass[j - step]) |
| 12 | mass[j] = mass[j - step]; |
| 13 | else |
| 14 | break; |
| 15 | } |
| 16 | mass[j] = tmp; |
| 17 | } |
| 18 | } |

В листинге 14.4 n – количество элементов в массиве, а mass[] – упорядочиваемый массив элементов.

## §14.6 Улучшенные методы сортировки: метод быстрой сортировки

Данный метод в настоящее время считается наиболее быстрым универсальным методом сортировки. Как ни странно, он является обобщением самого плохого из простейших методов – обменного метода. Эффективность метода достигается тем, что перестановка применяется не для соседних элементов, а **отстоящих друг от друга на приличном расстоянии**.

Более конкретно, алгоритм быстрой сортировки заключается в следующем.

* пусть каким-то образом в исходном наборе выделен некий элемент **x**, который принято называть **опорным**. В простейшем случае в качестве опорного можно взять **серединный** элемент массива
* просматривается часть массива, расположенная **левее** опорного элемента и находится первый по порядку элемент ai > **x**
* после этого просматривается часть массива, расположенная **правее** опорного элемента, причем - **в обратном порядке**, и находится первый по порядку (с конца) элемент aj < **x**
* производится **перестановка** элементов ai и aj
* после этого в левой части, начиная с ai отыскивается еще один элемент, больший **x**, а в правой части, начиная с aj отыскивается элемент, меньший **х**
* эти два элемента меняются местами
* эти действия (поиск слева и справа с последующим обменом) продолжаются до тех пор, пока не будет достигнут опорный элемент **x**
* после этого слева от опорного элемента **x** будут находиться элементы, **меньшие опорного**, а справа – элементы, **большие опорного**. При этом обе половины скорее всего не будут отсортированными
* после этого массив разбивается на **правую** и **левую** части, и каждая часть обрабатывается **отдельно** по той же самой схеме: определение опорного элемента, поиск слева и справа соответствующих элементов и их перестановка и т.д.

**Пример**. Пусть исходный набор включает 11 чисел:

13-42-28-17-09-25-47-31-39-15-20.

Основные шаги сортировки:

1. Выбор серединного элемента 25 (индекс 6): 13 42 28 17 09 **25** 47 31 39 15 20
2. поиск слева первого элемента, большего 25: 42 (2 сравнения)
3. поиск справа от конца первого элемента, меньшего 25: 20 (1 сравнение)
4. перестановка элементов 42 и 20: 13 20 28 17 09 **25**  47 31 39 15 42
5. поиск слева от 25 еще одного элемента, большего 25: 28 (1 сравнение)
6. поиск справа от 25 еще одного элемента, меньшего 25: 15 (1 сравнение)
7. перестановка элементов 28 и 15: 13 20 15 17 09 **25**  47 31 39 28 42
8. поиск слева от 25 еще одного элемента, большего 25: нет (2 сравнения)
9. поиск справа от 25 еще одного элемента, меньшего 25: нет (3 сравнения)
10. теперь слева от 25 все элементы меньше 25, а справа – больше
11. выделяем отдельно левую часть: 13 20 15 17 09
12. выбираем серединный элемент 15 (индекс 3): 13 20 **15** 17 09
13. поиск слева от 15 элемента, большего 15: 20 (2 сравнения)
14. поиск справа от 15 элемента, меньшего 15: 09 (1 сравнение)
15. перестановка 20 и 09: 13 09 **15** 17 20
16. поиск справа от 15 еще одного элемента, меньшего 15: нет (1 сравнение)
17. теперь слева от 15 все элементы меньше 15, а справа – больше
18. поскольку слева от 15 только 2 элемента, просто сравниваем их друг с другом и переставляем (09 и 13)
19. поскольку справа от 15 только 2 элемента, просто сравниваем их и не переставляем
20. получаем для левой части упорядоченный набор: **09 13 15 17 20**
21. возвращаемся к правой части: 47 31 39 28 42
22. выделяем серединный элемент 39 (индекс в данном поднаборе – 3): 47 31 **39** 28 42
23. поиск слева от 39 элемента, большего 39: 47 (1 сравнение)
24. поиск справа от 39 элемента, меньшего 39: 28 (2 сравнения)
25. переставляем 47 и 28: 28 31 **39** 47 42
26. поиск слева от 39 еще одного элемента, большего 39: нет (1 сравнение)
27. теперь слева от 39 все элементы меньше 39, а справа – больше
28. поскольку слева от 39 только 2 элемента, просто сравниваем их и не переставляем
29. поскольку справа от 39 только 2 элемента, просто сравниваем их и переставляем (42 и 47)
30. получаем для правой части упорядоченный набор: **28 31 39 42 47**
31. вместе с левой частью и серединным элементом 25 получаем окончательный результат

Итого для данного примера потребовалось 22 сравнения и 6 пересылок.

В целом, оценка трудоемкости метода быстрой сортировки является типичной для улучшенных методов и выражается соотношением (n\*log 2 n)/6. Отсюда следует, что данный метод неэффективен при малых n (десятки или сотни элементов), но с ростом n его эффективность резко растет, и при очень больших n метод дает **наилучшие показатели** среди всех универсальных методов сортировки.

К сожалению, есть одна ситуация, когда быстрая сортировка **теряет свою эффективность** и становится пропорциональной n2, т.е. опускается до уровня простых методов. Эта ситуация связана с **правилом выбора опорного элемента**. Эффективность метода сильно зависит от выбора опорного элемента, и использование простейшего способа выбора (серединный элемент массива) часто приводит к падению эффективности метода. Это связано с тем, что каждое разделение массива на две половины в идеале должно давать **примерно** **равное** число элементов слева и справа от опорного элемента (принцип дихотомии!). Если опорный элемент близок к минимальному или максимальному, после попарных перестановок будут получены **существенно** **неравномерные** наборы. Если подобная ситуация возникает на каждом шаге работы алгоритма, общая эффективность резко падает. Для устранения этого недостатка надо уметь **правильно выбирать опорный элемент**.

Наилучшее правило выбора опорного элемента – это так называемая **медиана**. Медиана – это средний элемент массива **не по расположению**, а по **значению**. В приведенном выше примере медианой является число 25, которое также было и серединным элементом (честно говоря, пример был подобран специально). К сожалению, поиск медианы в массиве является задачей, **сопоставимой по трудоемкости** с самой сортировкой, поэтому были предложены другие, более простые правила выбора опорного элемента.

На практике хорошо показал себя следующий способ: выбрать случайно в массиве **три элемента** и взять в качестве опорного **средний из них**. Этот способ очень прост в реализации, т.к. требует только двух сравнений, но, конечно, он может обеспечивать хорошие показатели только в среднем, и не гарантирует идеальное поведение алгоритма абсолютно для ЛЮБЫХ входных данных.

Что касается программной реализации базового алгоритма, то можно заметить его принципиальную особенность: разделение массива на 2 половины, разделение каждой половины на свои половины и т.д. При каждом разделении приходится **запоминать** правую половину (конечно, не сами элементы, а лишь **индексы** левой и правой границы) и **возвращаться** к ней после полной обработки левой половины. Все это как нельзя лучше соответствует **рекурсивному** принципу обработки, и поэтому быстрая сортировка проще всего реализуется рекурсивно.

Пример программной реализации метода быстрой сортировки представлен в листинге 14.5.

**Листинг 14.5 – Метод быстрой сортировки**

|  |  |
| --- | --- |
| 1 | void QuickSort(int[] array, int a, int b) |
| 2 | { |
| 3 | int i = a; |
| 4 | int j = b; |
| 5 | int middle = array[( a + b )/2]; |
| 6 | while(i <= j) |
| 7 | { |
| 8 | while(array[i] < middle) |
| 9 | { |
| 10 | i++; |
| 11 | } |
| 12 | while(array[j] > middle) |
| 13 | { |
| 14 | j--; |
| 15 | } |
| 16 | if(i <= j) |
| 17 | { |
| 18 | int temporaryVariable = array[i]; |
| 19 | array[i] = array[j]; |
| 20 | array[j] = temporaryVariable; |
| 21 | i++; |
| 22 | j--; |
| 23 | } |
| 24 | } |
| 25 | if (a < j) |
| 26 | { |
| 27 | QuickSort(array, a, j); |
| 28 | } |
| 29 | if (i < b) |
| 30 | { |
| 31 | QuickSort(array, i, b); |
| 32 | } |
| 33 | } |

## §14.7 Улучшенные методы сортировки: поразрядная сортировка

Пусть известно, что каждый ключ является **k–разрядным целым числом**. Например, если k = 4, то все ключи находятся в диапазоне 0000 – 9999. Смысл поразрядной сортировки заключается в том, что k раз повторяется карманная сортировка. На первом шаге все ключи группируются по **младшей цифре** (разряд единиц). Для этого в каждом ключе выделяется младшая цифра и элемент помещается в соответствующий список-карман для данной цифры. Потом все списки **объединяются** и создается новый массив, в котором элементы упорядочены по младшей цифре ключа. К этому массиву опять применяется карманная сортировка, но уже по **более старшей** цифре (разряд десятков): в каждом ключе выделяется вторая справа цифра и элементы распределяются по соответствующим спискам. Потом списки объединяются в массив, где элементы будут упорядочены уже **по двум младшим** цифрам. Процесс распределения по все более старшим цифрам с последующим объединением повторяется до старшей цифры (разряд k).

Поскольку цифр всего 10, то для реализации метода необходим вспомогательный массив из 10 ячеек для хранения адресов соответствующих списков.

**Пример**. Пусть имеется исходный набор из 15-ти двухразрядных ключей (k=2):

56, 17, 83, 09, 11, 27, 33, 02, 16, 45, 08, 37, 66, 99, 90

**Первый шаг**: выделяем младшую цифру и распределяем ключи по десяти спискам:

ключ 56 в список для цифры 6, ключ 17 в список для цифры 7, …, ключ 16 опять в список для цифры 6 и т.д.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

9**0** 1**1** 0**2** 8**3** 4**5** 5**6** 1**7** 0**8** 0**9**

3**3** 1**6** 2**7** 9**9**

6**6** 3**7**

Объединяем списки: 90, 11, 02, 83, 33, 45, 56, 16, 66, 17, 27, 37, 08, 09, 99

В этом наборе ключи упорядочены по младшей цифре

**Второй шаг**: выделяем старшую цифру (десятки) и распределяем ключи по своим спискам:

ключ 90 в список для 9, ключ 11 в список для 1, ключ 02 в список для 0 и т.д.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**0**2 **1**1 **2**7 **3**3 **4**5 **5**6 **6**6 **8**3 **9**0

**0**8 **1**6 **3**7 **9**9

**0**9 **1**7

Объединение этих списков дает отсортированный набор:

02, 08, 09, 11, 16, 17, 27, 33, 37, 45, 56, 66, 83, 90, 99

Для программной реализации можно объявить десятиэлементный массив и ссылочный тип для организации списков. Удобно вместе с началом каждого списка хранить указатель на его последний элемент, что упрощает как добавление новых элементов в конец списка, так и объединение отдельных списков. Внешний цикл алгоритма повторяется k раз (разрядность ключа). Каждый раз внутри этого цикла необходимо:

* обнулить все 20 указателей
* циклом по числу элементов (m) распределить элементы по своим спискам, выделяя в ключе необходимую цифру
* циклом по 10 объединить списки в новый набор данных

Так же успешной может считаться реализация данной сортировки с использованием двойных массивов или вложенных контейнеров из стандартной библиотеки шаблонов.

В целом, поразрядная сортировка не эффективна при малых объемах входных данных, но чем больше этот объем, тем выше эффективность метода. Трудоемкость пропорциональна объему входных данных, что лучше, чем у быстрой сортировки. Платой за это являются дополнительные затраты памяти для поддержки вспомогательных структур.

Интересно отметить, что данный метод очень хорошо подходит для современных архитектур вычислительных систем с конвейерной обработкой данных и использованием нескольких процессоров.

В качестве примера приведем реализацию функции поразрядной сортировки для целых чисел любой длины. Однако максимальная длина числа должна быть известна заранее. Она передается функции сортировки в качестве аргумента.

Пример программной реализации поразрядной сортировки представлен в листинге 14.6.

**Листинг 14.6 – Поразрядная сортировка**

|  |  |
| --- | --- |
| 1 | public static void sorting(int[] arr, int range, int length) |
| 2 | { |
| 3 | ArrayList[] lists = new ArrayList[range]; |
| 4 | for(int i = 0; i < range; ++i) |
| 5 | lists[i] = new ArrayList(); |
| 6 | for(int step = 0; step < length; ++step) { |
| 7 | //распределение по спискам |
| 8 | for(int i = 0; i < arr.Length; ++i) { |
| 9 | int temp = (arr[i] % (int)Math.Pow(range, step + 1)) / (int)Math.Pow(range, step); |
| 10 | lists[temp].Add(arr[i]); |
| 11 | } |
| 12 | //сборка |
| 13 | int k = 0; |
| 14 | for(int i = 0; i < range; ++i) { |
| 15 | for(int j = 0; j < lists[i].Count; ++j) { |
| 16 | arr[k++] = (int)lists[i][j]; |
| 17 | } |
| 18 | } |
| 19 | for(int i = 0; i < range; ++i) |
| 20 | lists[i].Clear(); |
| 21 | } |
| 22 | } |

В вышеописанном методе:

* length - максимальное количество разрядов в сортируемых величинах (например, при сортировке слов необходимо знать максимальное количество букв в слове),
* range - количество возможных значений одного разряда (при сортировке слов - количество букв в алфавите).

Количество проходов равно числу length.