Сортировки

Алгоритмы и структуры данных

Лекция

Основные особенности сортировки

Вопросы организации сортировки относятся к наиболее часто встречающимся в задачах машинной обработки данных. В большинстве компьютерных приложений множество объектов нуждается в переразмещении в соответствии с некоторым заранее определённым порядком, что существенно упрощает дальнейшую работу с ними.

Основные особенности сортировки

Задача сортировки.

Задача сортировки обычно формулируется так: дана последовательность из n элементов a_1, a_2, …, a_n, выбранных из множества, для которого выполняется либо $a_i<a_j$, либо $a_i>a_j$, либо $a_i=a_j$. Требуется найти такую перестановку этих элементов, которая бы приводила исходную последовательность в неубывающую, то есть $a_{k1}\leq a_{k2}\leq\ldots\leq a_{kn}$.

Классификация сортировок

- По временной (вычислительной) сложности

  Быстрые (но сложные) алгоритмы сортировки требуют (при $n \to \infty$) порядка ***n log n*** сравнений, **прямые** (простые) методы - $\mathbf{n^2}$. При небольших n прямые методы работают быстрее.
- По емкостной сложности

  Некоторые алгоритмы требуют дополнительную память O(n). А некоторые нет, их сложность O(1).
- Рекурсивность
- Использование операций сравнения

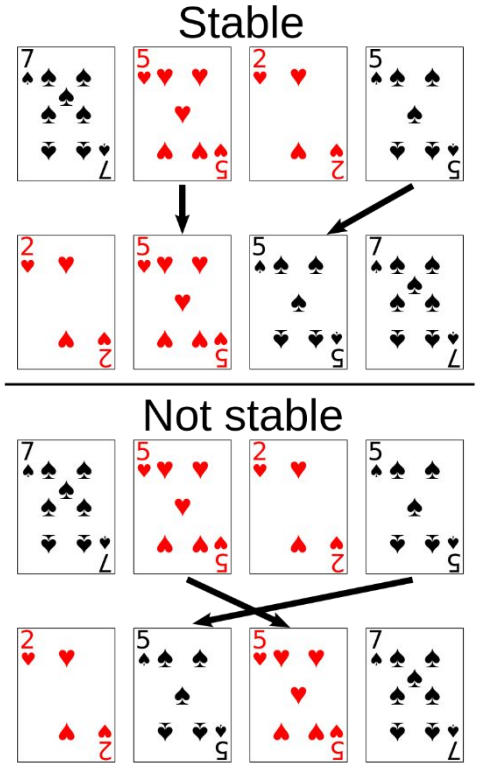
  Для алгоритмов, использующий сравнение элементов, минимальная сложность в худшем случае O(n log n).
- Естественность поведения (adaptivity)

  Эффективность алгоритма при обработке частично упорядоченных данных
- Последовательный или параллельный

Классификация сортировок

- Устойчивость (stability)

  Устойчивый алгоритм сохраняет относительный порядок элементов с одинаковыми ключами.



Классификация сортировок

- Метод
  - Вставка (включение) (insertion)
  - Обмен (exchanging)
  - Выбор (извлечение) (selection)
  - Слияние (merging)
  - Распределение (partitioning)

Идея методов

Идея **методов вставки (включения)** состоит в том, что сначала первый элемент массива рассматривается как упорядоченный массив и в этот массив включается следующий элемент исходного массива так, чтобы получился упорядоченный по неубыванию массив из двух элементов.

Затем в полученный упорядоченный массив включается третий элемент массива так, чтобы опять-таки получился упорядоченный массив. Процесс продолжается до тех пор, пока не будет включен последний элемент.

Различные алгоритмы включения отличаются способами выбора элемента для включения, способами определения места включения и методами самого включения.

Идея методов

Идея **методов обменов** состоит в следующем: в исходном массиве выбирается пара элементов, и они сравниваются между собой. Если их положение не удовлетворяет требованию упорядоченности, то элементы переставляются. Затем выбирается следующая пара элементов и так до тех пор, пока не получим упорядоченный массив.

Различные алгоритмы обменов отличаются способами выбора пары элементов для сравнения и перестановки, а также условиями окончания процесса сортировки.

Идея методов

Общая концепция **методов выбора (извлечения)** заключается в следующем: из исходного массива извлекается минимальный элемент и меняется местами с первым элементом массива, затем извлекается минимальный элемент из части массива, начиная со второго элемента, и меняется местами со вторым элементом и т. д. Последний раз минимальный элемент выбирается из двух последних элементов массива.

В результате получится массив, упорядоченный по **неубыванию**.

Различные методы извлечения отличаются объектом извлечения (минимальный или максимальный элемент) и, соответственно, объектами перестановки (первый или последний элемент), а также условием окончания процесса сортировки.

Идея методов

Метод слияния применяется в том случае, когда имеются два (или больше) упорядоченных массива и требуется соединить исходные массивы в один общий упорядоченный массив.

Идея методов

Метод распределения употребим в тех случаях, когда в исходном массиве имеется заданное, известное заранее, количество различных ключей (значений). Например, имеется список студентов с оценками по пятибалльной системе, полученными на экзамене. Нам известно заранее, что оценки могут быть 5, 4, 3 и 2.

Поэтому для упорядочения массива по невозрастанию можно сначала выбрать все записи с оценками 5, затем с оценками 4, потом с оценками 3 и, наконец, с оценками 2.

Сортировка обменами (Exchange sort)

Сравниваем 1-й элемент со всеми последующими и меняем местами, если нужно. Т.о. на 1-м месте окажется нужный элемент. Потом выполняем для 2-го элемента и т.д.

Сортировка обменами (Exchange sort)

```
Exchange-Sort(A)
    for i ← 1 to length[A]–1
        for j ← i+1 to length[A]
            if A[i] > A[j]
                Swap(A[i], A[j])
```

Сортировка обменами (Exchange sort)

В любом случае сложность $O(n^2)$.
Неустойчивая.

Плюсы:

- Простота реализации

Минусы:

- Медленная, даже в лучшем случае $O(n^2)$

Сортировка пузырьком (Bubble sort)

Метод пузырьковой сортировки представляет собой систематический обмен местами слева направо смежных элементов, не отвечающих выбранному порядку, до тех пор, пока они не оказываются на правильном месте. Большие элементы при этом как бы «всплывают пузырьками вверх» в конец списка.

Сортировка пузырьком (Bubble sort)

```
Bubble-Sort(A)
    do
        swapped ← false
        for i ← 1 to length[A]–1
            if A[i] > A[i+1]
                Swap(A[i], A[i+1])
                swapped ← true
    while swapped
```

Сортировка пузырьком (Bubble sort)

В худшем и среднем случае сложность $O(n^2)$. В лучшем (последовательность упорядочена) – $O(n)$.
Устойчивая.

Плюсы:

- Простота реализации

Минусы:

- Медленная в среднем

Сортировка перемешиванием (Shaker sort)

```
Shaker-Sort(A)
    do
        swapped ← false
        for i ← 1 to length[A]−1
            if A[i] > A[i+1]
                Swap(A[i], A[i+1])
                swapped ← true
        if not swapped
            break do-while loop
        for i ← length[A]−1 to 1
            if A[i] > A[i+1]
                Swap(A[i], A[i+1])
                swapped ← true
    while swapped
```

Сортировка расчёской (Comb sort)

Сортировка расчёской или методом прочёсывания (англ. comb sort) – это довольно упрощённый алгоритм сортировки, изначально спроектированный Влодзимежом Добосиевичем в 1980 г. Позднее он был переоткрыт и популяризован в статье Стефана Лэйси и Ричарда Бокса в журнале Byte Magazine в апреле 1991 г.

Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке.

Сортировка расчёской (Comb sort)

В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние между элементами) равен 1. **Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица**.

Сортировка расчёской (Comb sort)

Вначале выбирается последовательность расстояний $h=(h_1, h_2, h_3, \ldots, h_m)$, в которой $h_i > h_{i+1}$, например, для массива из 13 элементов, можно выбрать 8, 6, 4, 3, 2, 1.

На первом шаге при $h_1=8$ сравниваются и, в случае необходимости, переставляются местами элементы с номерами j и $j+h_1$, то есть 1-й и 9-й элементы, затем – 2-й и 10-й, 3-й и 11-й, 4-й и 12-й, 5-й и 13-й и т.д. до конца массива, то есть элементы, отстоящие друг от друга на 8 позиций.

На следующем шаге сравниваются и переставляются пары элементов с номерами j и $j+h_2$: (1, 7), (2, 8), (3, 9), (4, 10), (5,11), (6, 12), (7, 13) и т. д., то есть элементы, отстоящие друг от друга на 6 позиций.

Сортировка расчёской (Comb sort)

Далее выполняется проход по массиву для элементов, отстоящих друг от друга на 4 позиции, затем на 3 и 2 позиции.

На последнем шаге выполняется стандартная пузырьковая сортировка, которую можно рассматривать как продолжение предыдущего алгоритма для соседних элементов.

Алгоритм сортировки методом прочёсывания требует всего два цикла: один для уменьшения размера “прыжков” – расстояний между элементами, второй – для выполнения разновидности пузырьковой сортировки.

Сортировка расчёской (Comb sort)

Выбор длины прыжка

Разработчики алгоритма **эмпирическим** путем пришли к выводу, что **значение каждого следующего расстояния прыжка должно быть получено в результате деления предыдущего на 1.3**. Этот коэффициент даёт наилучшее время сортировки.

Эмпирическим путем также было установлено, что **значения расстояний 9 и 10** между сравниваемыми элементами **являются неоптимальными**, и если они присутствуют в последовательности, их лучше **заменять на 11**. В этом случае сортировка будет выполняться гораздо быстрее.

В качестве **начального расстояния** берется **целая часть от деления количества элементов n на 1.3.**

Сортировка расчёской (Comb sort)

```
Comb-Sort(A)
    gap ← length[A]
    shrink ← 1.3
    sorted ← true
    while sorted
        gap ← floor(gap / shrink)
        if gap ≤ 1
            gap ← 1
            sorted ← true

        i ← 1
        while i + gap <= length[A]
            if A[i] > A[i+gap]
                Swap(A[i], A[i+gap])
                sorted ← false
            i ← i + 1
```

В худшем случае сложность алгоритма составит $O(n^2)$.

В среднем лучше, чем пузырьковая.

Теряется устойчивость.

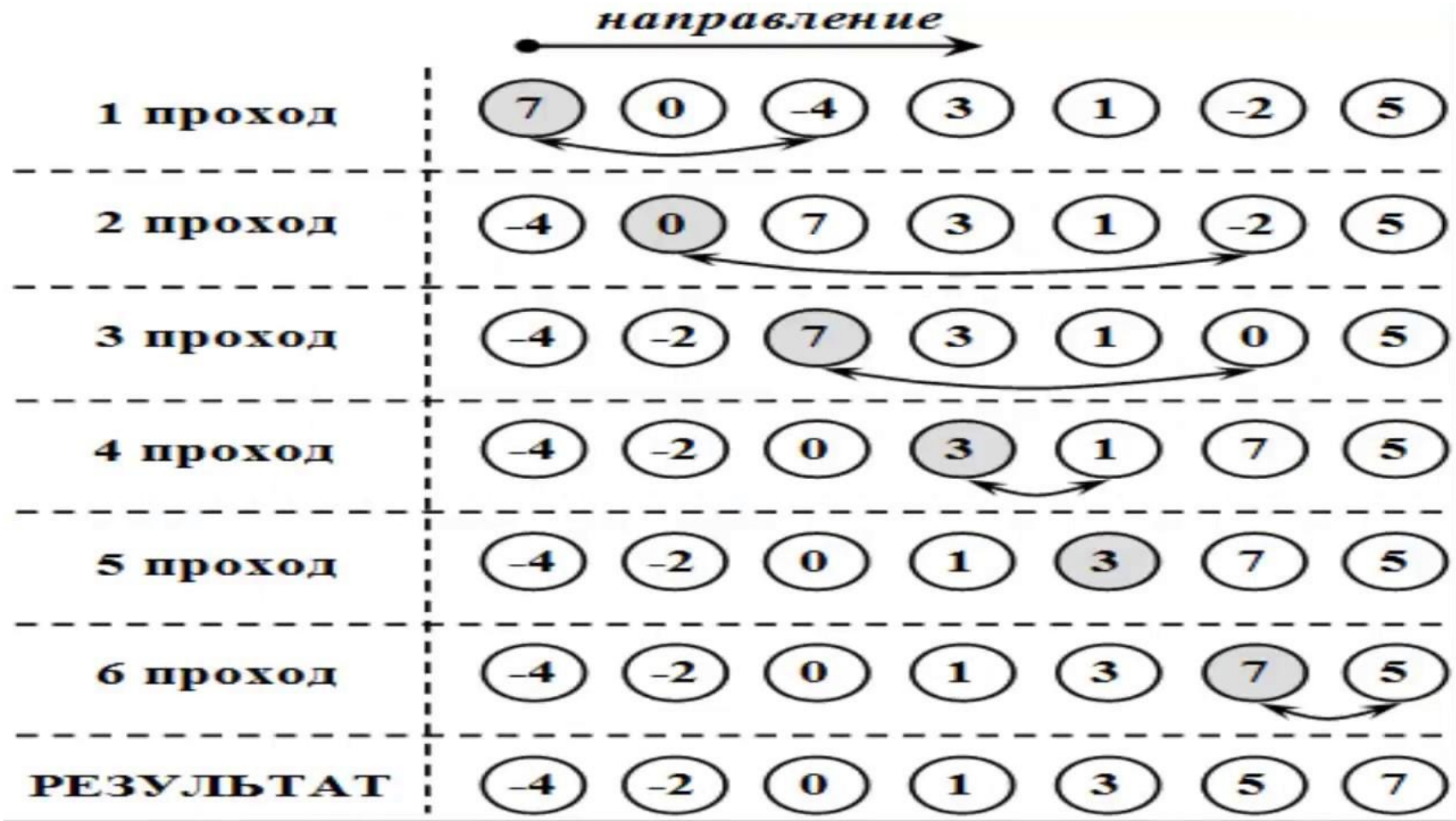
Сортировка выбором (Selection sort)

Пожалуй, самый простой алгоритм сортировок.

Судя по названию сортировки, необходимо что-то выбирать (максимальный или минимальный элементы массива). Алгоритм сортировки выбором находит в исходном массиве максимальный или минимальный элементы, в зависимости от того как необходимо сортировать массив, по возрастанию или по убыванию.

Если массив должен быть отсортирован по возрастанию, то из исходного массива необходимо выбирать минимальные элементы. Если же массив необходимо отсортировать по убыванию, то выбирать следует максимальные элементы.

Сортировка выбором (Selection sort)



Сортировка выбором (Selection sort)

```
Selection-Sort(A)
    for i ← 1 to length[A]-1
        jMin ← i
        for j ← i+1 to length[A]
            if A[j] < A[jMin]
                jMin ← j
        if jMin ≠ j
            Swap(A[i], A[jMin])
```

Сортировка выбором (Selection sort)

В любом случае сложность $O(n^2)$.
Неустойчивая.

Плюсы:

- Простота реализации

Минусы:

- Медленная, даже в лучшем случае $O(n^2)$

Пирамидальная сортировка (Heapsort)

Пирамидальная сортировка (heapsort (heap − куча)) − алгоритм сортировки, требующий при сортировке n элементов в худшем, в среднем и в лучшем случае O(n log n) операций. Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Пирамидальная сортировка сильно **улучшает базовый алгоритм (сортировку выбором**), используя структуру данных «**куча**» для ускорения нахождения и удаления максимального (минимального) элемента.

С другой стороны, пирамидальная сортировка может рассматриваться как **усовершенствованная Bubble sort**, в которой элемент всплывает (max-heap) / тонет (min-heap) по многим путям.

Сортировка вставками (Insertion sort)

Сортировка вставками — достаточно простой алгоритм.

Как в и любом другом алгоритме сортировки, с увеличением размера сортируемого массива увеличивается и время сортировки.

Основным преимуществом алгоритма сортировки вставками является возможность сортировать массив по мере его получения. То есть имея часть массива, можно начинать его сортировать.

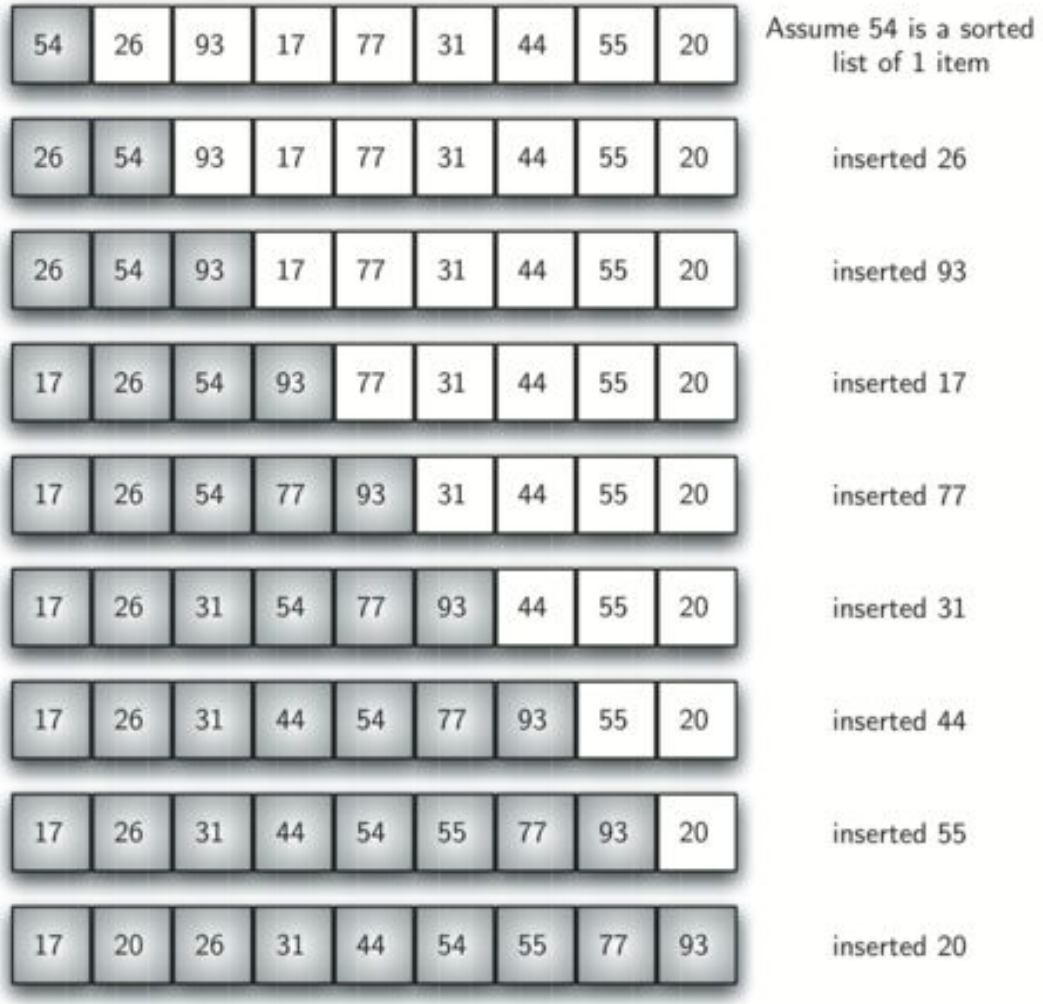
Сортировка вставками (Insertion sort)

Сортируемый массив можно разделить на две части — **отсортированная** часть и **неотсортированная**.

В начале сортировки первый элемент массива считается отсортированным, все остальные — не отсортированные. Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива.

Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная часть уменьшается.

Сортировка вставками (Insertion sort)



Сортировка вставками (Insertion sort)

```
Insertion-Sort(A)
    for i ← 2 to length[A]
        key ← A[i]
        j ← i-1
        while j > 0 and key < A[j]
           A[j+1] ← A[j]
           j ← j-1
        A[j+1] ← key
```

Сортировка вставками (Insertion sort)

В худшем и среднем случае сложность $O(n^2)$.
В лучшем случае (последовательность упорядочена) – $O(n)$
Устойчивая.

Плюсы:

- Простота реализации
- Быстро работает на частично упорядоченных последовательностях
- Одна из лучших среди сортировок сложности $O(n^2)$

Минусы:

- Медленнее сортировок со сложностью $O(n \log n)$

Сортировка Шелла (Shellsort)

Сортировка Шелла (англ. Shellsort) — разработана Дональдом Л. Шеллом в 1959 году.

Идея алгоритма состоит в сравнении элементов, стоящих не только рядом, но и на расстоянии друг от друга. Иными словами это сортировка вставками с предварительными «грубыми» проходами.

При сортировке Шелла **сначала сравниваются и сортируются между собой ключи, отстоящие один от другого на некотором расстоянии h. Полученная последовательность элементов в списке называется отсортированной по h.** После этого процедура повторяется **для некоторых меньших значений h**, а **завершается** сортировка Шелла упорядочиванием элементов **при h = 1** (то есть, обычной сортировкой вставками).

Сортировка Шелла (Shellsort)

Сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, но она имеет ряд **преимуществ**:

- отсутствие потребности в памяти под стек
- отсутствие деградации при неудачных наборах данных – в этом случае быстрая сортировка (qsort) легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

Сортировка Шелла (Shellsort)

Пусть дан список A = (32,95,16,82,24,66,35,19,75,54,40,43,93,68), в качестве значений h выбраны 5, 3, 1.

На первом шаге сортируются подсписки A, составленные из всех элементов A, различающихся на 5 позиций, то есть подсписки A5,1 = (32,66,40), A5,2 = (95,35,43), A5,3 = (16,19,93), A5,4 = (82,75,68), A5,5 = (24,54).

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

Исходный массив	32 95 16 82 24 66 35 19 75 54 40 43 93 68	
После сортировки с шагом 5	32 35 16 68 24 40 43 19 75 54 66 95 93 82	6 обменов
После сортировки с шагом 3	32 19 16 43 24 40 54 35 75 68 66 95 93 82	5 обменов
После сортировки с шагом 1	16 19 24 32 35 40 43 54 66 68 75 82 93 95	15 обменов

Сортировка Шелла (Shellsort)

СОРТИРОВКА ШЕЛЛА СО СМЕЩЕНИЯМИ 8, 4, 2, 1

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

Сортировка через 8:

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

Сортировка через 4:

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

Сортировка через 2:

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

Сортировка через 1:

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

Сортировка Шелла (Shellsort)

СОРТИРОВКА ШЕЛЛА СО СМЕЩЕНИЯМИ 7, 5, 3, 1

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

Сортировка через 7:

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

Сортировка через 5:

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

Сортировка через 3:

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

Сортировка через 1:

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

Сортировка Шелла (Shellsort)

Среднее время работы алгоритма зависит от длин промежутков h, на которых будут находиться сортируемые элементы исходного массива ёмкостью n на каждом шаге алгоритма.

Существует несколько подходов к выбору этих значений.

1. Первоначально **Шеллом** использовалась **последовательность длин промежутков 1, 2, 4, 8, 16, 32 и т.д**. В обратном порядке она вычисляется по формулам: $h_1=n/2$, $h_i=h_{i-1}/2$, $h_0=1$, **В худшем случае сложность алгоритма составит $O(n^2)$**.

Сортировка Шелла (Shellsort)

2. Гораздо **лучший вариант** предложил **Роберт Седжвик**. Его последовательность имеет вид (самая быстрая из известных на сегодня)

$$inc[s] = \begin{cases} 9*2^s - 9*2^{s/2} + 1, \text{ если } s \text{ четно} \\ 8*2^s - 6*2^{(s+1)/2} + 1, \text{ если } s \text{ нечетно} \end{cases}$$

При использовании таких приращений **среднее количество операций имеет порядок $O(n^{7/6})$, в худшем случае - порядок $O(n^{4/3})$**.

Последовательность вычисляется в порядке, противоположном используемому: inc [0] = 1, inc [1] = 5, ...19, 41, 109 и т.д. Формула дает сначала меньшие числа, затем все большие, при этом расстояние между сортируемыми элементами, наоборот, должно уменьшаться. Поэтому массив приращений inc вычисляется перед запуском собственно сортировки до максимального расстояния между элементами, которое будет первым шагом в сортировке Шелла. Потом его значения используются в обратном порядке.

При использовании формулы Р. Седжвика следует остановиться на значении inc[s-1], если 3*inc[s] > n;

Сортировка Шелла (Shellsort)

3. **Хиббардом** предложена последовательность: все значения $(2^i-1)/2<=n/2$, $i \in N$; такая последовательность шагов приводит к алгоритму со сложностью $\mathbf{O(n^{3/2})}$;

4. **Праттом** предложена последовательность: все значения $2^i \cdot 3^j <= n/2$, $i, j \in N$; в таком случае сложность алгоритма составляет $\mathbf{O(n \cdot (\log^2 n);}$

5. **Эмпирическая последовательность Марцина Циура** (последовательность A102549 в OEIS **(On-Line Encyclopedia of Integer Sequences)):** $h \in \{1, 4, 10, 23, 57, 132, 301, 701, 1750\}$; является одной из лучших, для сортировки массива ёмкостью приблизительно до 4000 элементов;

6. **Эмпирическая последовательность, основанная на числах Фибоначчи**: $h \in \{Fn\}$;

7. Для достаточно больших массивов рекомендуемой считается **последовательность, предложенная в 1969 году Дональдом Кнутом**: 1, 4, 13, 40, 121 и т.д. (каждое последующее значение на единицу больше, чем утроенное предыдущее $h_i+1 = 3h_i+1$, а $h_1 = 1$). Начинается процесс с h_m, что $h_m \geq [n/9]$. Для списков средних размеров Кнут оценил быстродействие для среднего случая как $O(n^{5/4})$, а для худшего случая как $O(n^{3/2})$.

В настоящее время не известна последовательность h_i, h_{i-1}, h_{i-2}, ...,h_1, оптимальность которой доказана.

Сортировка Шелла (Shellsort)

```
Shell-Sort(A)
    h ← length[A] / 2
    while h > 0
        for i ← h + 1 to length[A]
            key ← A[i]
            j ← i-h
            while j > 0 and key < A[j]
                A[j+h] ← A[j]
                j ← j-h
            A[j+h] = key
        h ← h / 2
```

Быстрая сортировка (Quicksort)

Быстрая сортировка (quicksort), сортировка Хоара, часто называемая qsort по имени реализации в стандартной библиотеке языков — широко известный алгоритм сортировки, разработанный английским информатиком Тони 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.

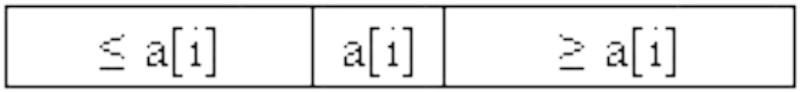
Быстрая сортировка (Quicksort)

Быстрая сортировка — рекурсивный алгоритм, который использует подход «разделяй и властвуй». Процедура быстрой сортировки разбивает список на два подсписка так, что любой элемент из 1-го подсписка не больше любого элемента из 2-го подсписка. А затем рекурсивно вызывает себя для сортировки двух подсписков.

Быстрая сортировка (Quicksort)

Схема алгоритма:

1. Из массива выбирается некоторый опорный элемент a[i]
2. Запускается процедура разделения массива, которая перемещает все ключи, меньшие либо равные a[i], влево от него, а все ключи, большие либо равные a[i] - вправо
3. Теперь массив состоит из двух подмножеств, причем левое меньше либо равно правого



4. Для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру

Быстрая сортировка (Quicksort)

На производительность алгоритма влияют выбор опорного элемента и способ разбиения на подмассивы.

Быстрая сортировка (Quicksort)

Способы выбора опорного 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 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**.

Быстрая сортировка (Quicksort)

Способы разбиения

1. **Разбиение Ломуто**.
2. **Разбиение Хоара**.

Быстрая сортировка (Quicksort)

Разбиение Ломуто

```
Quicksort(A, lo, hi)
    if lo >= 1 and hi >= 1 and lo < hi then
        p := Partition(A, lo, hi)
        Quicksort(A, lo, p – 1) // pivot is excluded
        Quicksort(A, p + 1, hi)

Partition(A, lo, hi)
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi do
        if A[j] <= pivot then
            i := i + 1
            Swap(A[i], A[j])
    return i
```

Быстрая сортировка (Quicksort)

Разбиение Хоара

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i, j по тем же правилам...
4. Повторяем шаг 3, пока i <= j.

Быстрая сортировка (Quicksort)

Разбиение Хоара

```
Quicksort(A, lo, hi)
    if lo >= 1 and hi >= 1 and lo < hi then
        p := Partition(A, lo, hi)
        Quicksort(A, lo, p) // pivot is included
        Quicksort(A, p + 1, hi)

Partition(A, lo, hi)
    pivot := A[(low + hi) / 2]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i >= j then
            return j
        Swap(A[i], A[j])
```

Быстрая сортировка (Quicksort)

Сложность

В лучшем случае временная сложность выражается рекуррентной формулой $T(n) = 2T\left(\frac{n}{2}\right) + n = O(n\,logn)$

- **Временная.** В среднем O(n log n), в худшем $O(n^2)$
- **Емкостная.** В среднем O(log n), в худшем O(n)

Быстрая сортировка (Quicksort)

Усовершенствование алгоритма

- Можно **улучшить производительность быстрой сортировки**, если прекратить рекурсию до того, как подсписки уменьшатся до нуля, и **использовать для завершения работы сортировку вставками**.
- Для того, чтобы в худшем случае емкостная сложность была O(log n) и не было переполнения стека, можно для большей подпоследовательности использовать хвостовую рекурсию или итерацию с обновлением границ

Интроспективная сортировка (Introsort)

Является гибридным алгоритмом. Использует быструю сортировку. При превышении заданного уровня рекурсии (например, log n) переключается на пирамидальную сортировку. Как и быстрая сортировка, использует сортировку вставками для небольших диапазонов.

В худшем случае имеет временную сложность O(n logn) и емкостную сложность O(log n).

Используется в качестве реализации неустойчивой сортировки многих стандартных библиотек.

Сортировка деревом (Tree sort)

Алгоритм заключается в построении бинарного дерева поиска (БДП, BST) из элементов последовательности и последующем обходе дерева.

Как и Quicksort, разделяет последовательность на 2 части по опорному элементу. Больше накладных расходов по сравнению с Quicksort.

Если использовать сбалансированное БДП, то в худшем случае временная сложность будет O(n log n). Но ещё больше накладных расходов.

Сортировка слиянием (Merge sort)

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.

Сортировка слиянием (Merge sort)

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).

Сортировка слиянием (Merge sort)

Этап слияния

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

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 вставками для завершения работы.

Сортировка слиянием (Merge sort)

Преимущества алгоритма сортировки слиянием

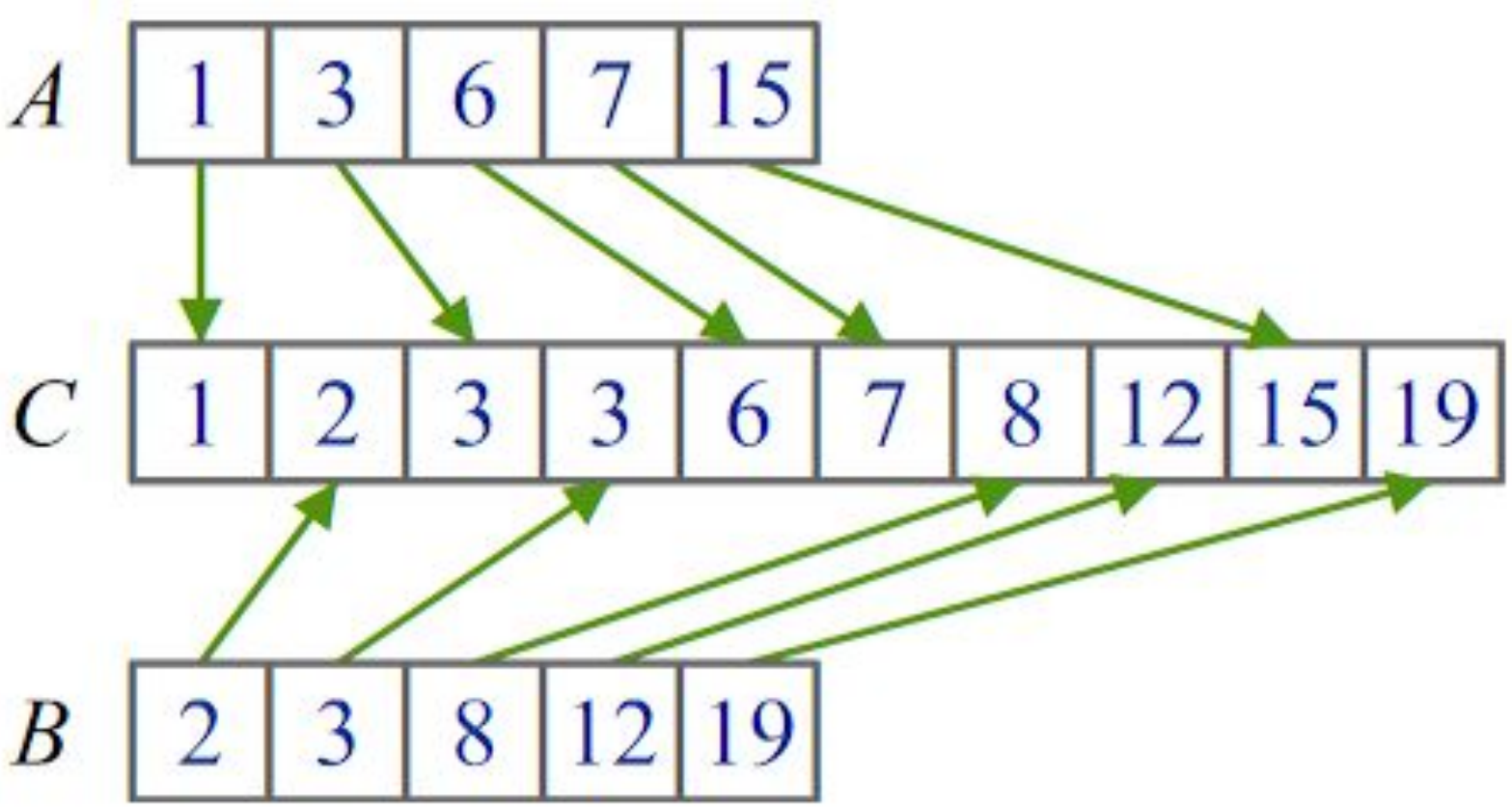
Время работы алгоритма сортировки слиянием остается одним и тем же для различных представлений данных и начального распределения. Его **можно использовать** и в случае, когда в списке имеется **много дублированных** значений.

Поскольку **сортировка слиянием** 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.

Сортировка слиянием (Merge sort)



Сортировка слиянием (Merge sort)

```
a ← 0; b ← 0;
While a < n_a and b < n_b
|   If A[a] ⩽ B[b]
|   |   C[a + b] ← A[a];
|   |   a ← a + 1;
|   Else
|   |   C[a + b] ← B[b];
|   |   b ← b + 1;
|   End;
End;
If a < n_a
|   Copy remain part of A
Else
|   Copy remain part of B
End;
```

```
a ← 0; b ← 0;
While a + b < n_a + n_b
|   If b ⩾ n_b or (a < n_a and A[a] ⩽ B[b])
|   |   C[a + b] ← A[a];
|   |   a ← a + 1;
|   Else
|   |   C[a + b] ← B[b];
|   |   b ← b + 1;
|   End;
End;
```

Сортировка слиянием (Merge sort)

Сверху вниз

```
MergeSort(A, lo, hi)
    if lo < hi then
        mid := (lo + hi) / 2
        MergeSort(A, lo, mid)
        MergeSort(A, mid + 1, hi)
        // merge fragments A[lo:mid] and A[mid+1:hi]
```

Снизу вверх

```
MergeSort(A, n)
    width := 1
    while width < n do
      i := 1
      while i <= n do
          // merge fragments A[i:i+width-1] and A[i+width:i+2*width-1]
          i := i + 2 * width
      width := 2 * width
```

Сортировка слиянием (Merge sort)

Сложность

В любом случае временная сложность выражается рекуррентной формулой $T(n) = 2T\left(\frac{n}{2}\right) + n = O(n\,logn)$

- **Временная.** В любом случае O(n log n)
- **Емкостная.** В любом случае O(n)

Сортировка слиянием (Merge sort)

Есть модификация, которой требуется O(1) вспомогательной памяти с сохранение устойчивости. Сложность слияния при этом в худшем случае получается O(n log n). Итоговая временная сложность в худшем случае $O(n \log^2 n)$.

Есть модификации с емкостной сложностью O(1) и временной сложностью O(n log n). Но они теряют свойство устойчивости.

Есть блочная сортировка слияниями (block merge sort), которая имеет сложности O(n log n) и O(1), но при этом устойчивая.

Timsort

Был разработан Тимом Питерсом в 2002 году на языке Python.

Является гибридным алгоритмом. В основе – сортировки слияниями и вставками. Отлично работает для почти упорядоченных данных.

Временная сложность: в лучшем случае O(n), в среднем и худшем O(n log n).

Емкостная сложность – в худшем случае O(n).

Используется в качестве реализации устойчивой сортировки некоторых стандартных библиотек.

Внешняя многофазная сортировка слиянием

```
CreateRuns(S)
    S – размер создаваемых отрезков

    CurrentFile ← A
    while конец входного файла не достигнут
        read S записей из входного файла
        sort S записей
        write S записей в CurrentFile
        if CurrentFile = A then
            CurrentFile ← B
        else
            CurrentFile ← A
```

Внешняя многофазная сортировка слиянием

```
PolyPhaseMerge(S)
    S – размер исходных отрезков

    Size ← S
    Input1 ← A
    Input2 ← B
    CurrentOutput ← C
    while файл Input2 непустой
        while конец Input1 не достигнут
            слить отрезок длины Size из Input1 с отрезком длины Size из Input2,
                записав результат в CurrentOutput
            if CurrentOutput = A then
                CurrentOutput ← B
            elif CurrentOutput = B
                CurrentFile ← A
            elif CurrentOutput = C
                CurrentFile ← D
            elif CurrentOutput = D
                CurrentFile ← C
        Size ← Size * 2
        if Input1 = A then
            Input1 = C
            Input2 = D
            CurrentOutput = A
        else
            Input1 = A
            Input2 = B
            CurrentOutput = C
```

Сортировка подсчетом (Counting sort)

Сортировка подсчетом (Counting sort)

Сортировка подсчетом применима, когда ключи элементов последовательности являются небольшими неотрицательными целыми числами. Либо ключи являются целыми числами и разница между минимальным и максимальным ключами небольшая.

Сортировка не использует сравнения, поэтому ограничение сложности O(n log n) для неё не действует.

Сортировка подсчетом (Counting sort)

- Считаем количество элементов для каждого ключа
- Преобразуем массив с количествами в префиксные суммы – для каждого элемента получим его номер в выходной последовательности
- С помощью префиксных сумм копируем элементы во временный массив на нужные позиции

Сортировка подсчетом (Counting sort)

Предполагаем, что ключи содержат целые числа из [0…k-1]

```
CountingSort(A, k)
    C := array of k zeroes
    B := array of same length as A

    for i := 0 to length[A] - 1 do
        j := key(A[i])
        C[j] := C[j] + 1

    for i := 1 to k - 1 do
        C[i] := C[i] + C[i - 1]

    for i := length[A] - 1 downto 0 do
        j = key(A[i])
        C[j] := C[j] - 1
        B[C[j]] := A[i]

    for i := 0 to length[A] - 1 do
        A[i] := B[i]
```

Сортировка подсчетом (Counting sort)

Сортировка является устойчивой.
Временная сложность O(n + k).
Емкостная сложность O(n + k).

Если элементами массива является числа, то можно переписать алгоритм без использования вспомогательного массива B, уменьшив емкостную сложность до O(k).

Если значение k неизвестно или в ключах есть отрицательные числа, то можно за O(n) найти минимальный и максимальный ключи и выполнить преобразование

[min…max] -> [0…max - min]

Блочная сортировка (Bucket sort)

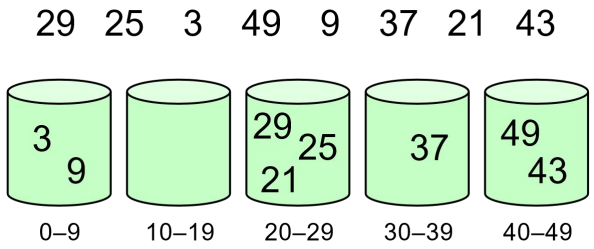
Блочная сортировка (Bucket sort)

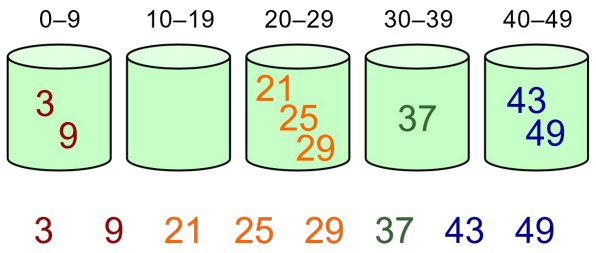
Блочная сортировка (также известная как корзинная или карманная сортировка) основана на разделении массива на блоки таким образом, что все элементы в последующем блоке не меньше, чем в предыдущем, и последующей сортировке полученных блоков.

Сложность зависит от количества блоков, алгоритма сортировки блока и насколько равномерно распределены ключи в массиве.

Блочная сортировка (Bucket sort)

- Создаем массив корзин
- Обходим исходный массив и раскладываем его элементы по корзинам
- Сортируем непустые корзины
- Обходим корзины по порядку и копируем их элементы в исходный массив





Блочная сортировка (Bucket sort)

Предполагаем, что ключи содержат числа из [0…1)

```
BucketSort(A, k)
    B := array of k empty lists

    for i := 0 to length[A] - 1 do
        insert A[i] into B[floor(key(A[i])*k)]

    for i := 0 to k - 1 do
        SomeSort(B[i])

    concatenate B[0], ... , B[k - 1] into A
```

Блочная сортировка (Bucket sort)

В худшем случае все элементы попадут в 1 корзину и временная сложность будет равняться сложности алгоритма, выбранного для сортировки корзины.

В случае равномерного распределения ключей и использовании сортировки вставками временная сложность будет $O(n^2/k + n)$. Если мы возьмем $k=O(n)$, то получим сложность $O(n)$.

Если заранее посчитать количество элементов в корзинах или использовать динамические массивы, то емкостная сложность составит $O(n)$.

Если $k=2$, то получаем Quicksort со средним значением диапазона в качестве опорного элемента.

Поразрядная сортировка (Radix sort)

Поразрядная сортировка (Radix sort)

Поразрядная (цифровая, корневая) сортировка не использует сравнения. Она распределяет элементы по корзинам в соответствии с разрядами чисел. Для элементов с более чем 1 разрядом, процесс повторяется для каждого разряда, при этом сохраняется порядок с предыдущего шага.

Можно начать сортировку как с наименее значимого разряда (LSD), так и с наиболее значимого (MSD).

Поразрядная сортировка (Radix sort)

LSD (least significant digit)

1. Начинаем с самого младшего разряда
2. Разбиваем массив на корзины по текущему разряду. Элементы внутри каждой корзины сохраняют относительный порядок, который был в массиве.
3. Собираем корзины в изначальный массив.
4. Переходим к следующему разряду и возвращаемся в (2)

Поразрядная сортировка (Radix sort)

LSD (least significant digit)

Входной массив
[170, 45, 75, 90, 2, 802, 2, 66]
После сортировки по единицам
[{170, 90}, {2, 802, 2}, {45, 75}, {66}]
После сортировки по десяткам
[{*0*2, 80*2*, *0*2}, {45}, {66}, {170, 75}, {90}]
После сортировки по сотням
[{*00*2, *00*2, *0*45, *0*66, *0*75, *0*90}, {170}, {802}]

Поразрядная сортировка (Radix sort)

LSD (least significant digit)

```
Digit(x, i, d)
    return i-й с конца разряд числа x в системе с основанием 2^d

RadixSort(A, k, d)
    dc := (k + d - 1) / d
    m := 2^d
    C := array of length m
    B := array of same length as A
    for i := 0 to dc - 1 do
        for j := 0 to m - 1 do
            C[j] := 0
        for j := 0 to length[A] - 1 do
            r := Digit(A[j], i, d)
            C[r] := C[r] + 1
        for j := 1 to m - 1 do
            C[j] := C[j] + C[j - 1]
        for j := length[A] - 1 downto 0 do
            r := Digit(A[j], i, d)
            C[r] := C[r] – 1
            B[C[r]] := A[j]
        for j := 0 to length[A] - 1 do
            A[j] := B[j]
```

Поразрядная сортировка (Radix sort)

LSD (least significant digit)

Сортировка устойчивая
Временная сложность $O(n \cdot k/d)$, где k – размер ключа в битах, d – размер разряда в битах
Емкостная сложность $O(n + 2^d)$

Поразрядная сортировка (Radix sort)

MSD (most significant digit)

- Разбиваем на корзины по старшему разряду
- Рекурсивно сортируем корзины по следующему разряду
- Собираем корзины в изначальный массив

Поразрядная сортировка (Radix sort)

MSD (most significant digit)

Входной массив
[170, 045, 075, 025, 002, 024, 802, 066]
После сортировки по сотням
[{045, 075, 025, 002, 024, 066}, {170}, {802}]
После сортировки по десяткам
[{ {002}, {025, 024}, {045}, {066}, {075} }, 170, 802]
После сортировки по единицам
[002, { {024}, {025} }, 045, 066, 075 , 170, 802]

Поразрядная сортировка (Radix sort)

MSD (most significant digit)

Сортировка устойчивая
Временная сложность $O(n \cdot k/d)$, где k – размер ключа в битах, d – размер разряда в битах
Емкостная сложность $O(n + 2^d)$

Поразрядная сортировка (Radix sort)

MSD (most significant digit)

Можно выбрать d=1, чтобы не создавать дополнительные массивы. При этом теряется устойчивость.

Рекомендации выбора алгоритма сортировки

- если нужно **быстро реализовать** алгоритм сортировки, используйте **быструю сортировку**, а затем при необходимости поменяйте алгоритм;
- если нужна устойчивая сортировка, используйте **сортировку слияниями**;
- если **более 99 процентов списка уже отсортировано или список очень мал** (100 или менее элементов), используйте **сортировку вставками**;
- если **элементы** в списке — **целые числа**, **разброс значений которых невелик** (до нескольких тысяч), используйте **сортировку подсчетом**;
- если **элементы** в списке — **целые числа**, используйте **поразрядную сортировку**.