Лекция

## Сортировка данных

Часто приходится упорядочивать некие элементы по некоторому правилу ранжирования. Наиболее известное упорядочивание по убыванию или возрастанию значений чисел.

Пусть заданы последовательность $a_0$, $a_1$... $a_n$ и *функция сравнения*, которая на любых *двух* элементах последовательности принимает одно из трех значений: *меньше*, *больше* или *равно*. Задача сортировки состоит в перестановке членов последовательности таким образом, чтобы выполнялось условие: $a_i <= a_{i+1}$, для всех *i* от **0** до ***n***.

Возможна ситуация, когда элементы состоят из нескольких полей:


**Рис. 1**

```
struct element {
  field x;
  field y;
}
```

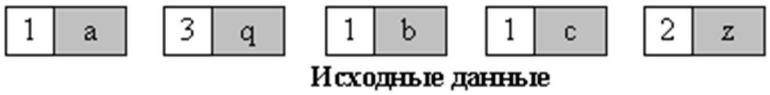
Если значение функции сравнения зависит только от поля ***x***, то ***x*** называют *ключом*, по которому производится сортировка. На практике, в качестве ***x*** часто выступает число, а поле ***y*** хранит какие-либо данные, никак не влияющие на работу алгоритма.

Проблема сортировки породила большое количество разнообразнейших решений. Существует ли некий “универсальный”, наилучший алгоритм? Вообще говоря, нет. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, работающий оптимальным образом.

Для того чтобы обоснованно сделать выбор, необходимо учесть следующие параметры, по которым и производится оценка алгоритмов.

1. *Время* сортировки — основной параметр, характеризующий быстродействие алгоритма.
2. *Память* — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.

3. *Устойчивость* — устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из нескольких полей, как на рисунке, а сортировка происходит по одному из них, например, по ***x***.




**Исходные данные**


**Пример работы устойчивой сортировки.**

Взаимное расположение равных элементов с ключом 1 и дополнительными полями "a", "b", "c" осталось прежним: элемент с полем "a", затем – с "b", затем – с "c".


**Пример работы неустойчивой сортировки.**

Взаимное расположение равных элементов с ключом 1 и дополнительными полями "a", "b", "c" изменилось.

4. *Естественность поведения* – эффективность метода при обработке уже отсортированных, или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Еще одним важным свойством алгоритма является его *сфера применения*. Здесь основных позиций две:

- *внутренние сортировки* работают с данным в оперативной памяти с произвольным доступом;
- *внешние сортировки* упорядочивают информацию, расположенную на внешних носителях. Это накладывает некоторые дополнительные ограничения на алгоритм:
  - доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим;
  - объем данных не позволяет им разместиться в ОЗУ;

Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

Данный класс алгоритмов делится на два основных подкласса:

Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к

любой ячейке. Данные обычно сортируются на том же месте, без дополнительных затрат.

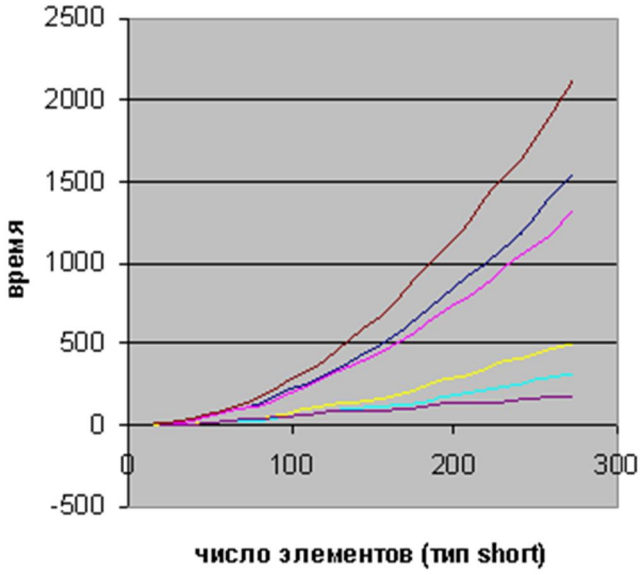
Внешняя сортировка оперирует с запоминающими устройствами большого объема, но с доступом не произвольным, а последовательным (сортировка файлов), то есть в данный момент мы “видим” только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это приводит к специальным методам сортировки, обычно использующим дополнительное дисковое пространство.

### Квадратичные и субквадратичные алгоритмы

- Сортировка выбором (SelectSort)
- Сортировка пузырьком (BubbleSort) и ее улучшения
- Сортировка простыми вставками (InsertSort)
- Сортировка Шелла (ShellSort)

### Сравнение времени сортировок

Изображенный ниже график иллюстрирует разницу в эффективности изученных алгоритмов.




- коричневая линия: сортировка пузырьком;
- синяя линия: шейкер-сортировка;
- розовая линия: сортировка выбором;
- желтая линия: сортировка вставками;
- голубая линия: сортировка вставками со сторожевым элементом;
- фиолетовая линия: сортировка Шелла.

### Логарифмические и линейные алгоритмы

- Пирамидальная сортировка (HeapSort)
- Быстрая сортировка (QuickSort)
- Поразрядная сортировка (Radix sort)

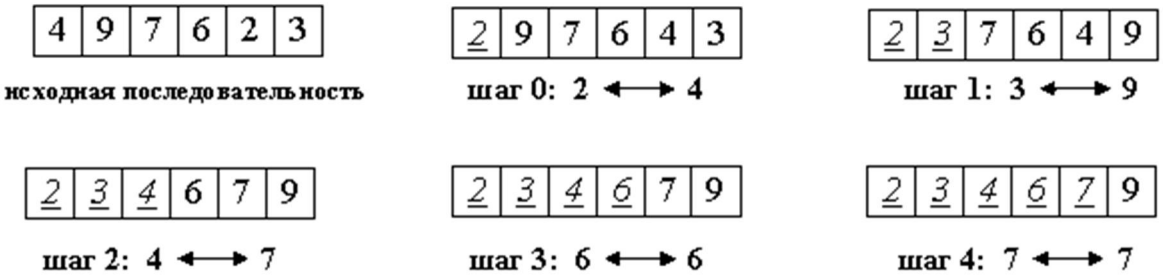
## Сортировки массивов

### *Сортировка выбором*

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая (*n*–1)-м.

На *i*-м шаге выбираем наименьший из элементов *a*[i] ... *a*[*n*] и меняем его местами с *a*[*i*]. Последовательность шагов при *n*=5 изображена на рисунке ниже.




Вне зависимости от номера текущего шага *i*, последовательность *a*[0]...*a*[*i*] (выделена курсивом) является упорядоченной. Таким образом, на (*n*–1)-м шаге вся последовательность, кроме *a*[*n*] оказывается отсортированной, а *a*[*n*] стоит на последнем месте по праву: все меньшие элементы уже ушли влево.

Для нахождения наименьшего элемента из *n*+1 рассматриваемых алгоритм совершает n сравнений. С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

$n + (n–1) + (n–2) + (n–3) + ... + 1 = 1/2 * ( n^2+n ) = O(n^2)$.

Таким образом, так как число обменов всегда будет меньше числа сравнений, время сортировки растет квадратично относительно количества элементов.

Алгоритм не использует дополнительной памяти: все операции происходят “на месте”.

Устойчив ли этот метод? Рассмотрим последовательность из трех элементов, каждый из которых имеет два поля, а сортировка идет по первому из них.


**исходная последовательность** **шаг 0: 2 ⟷ 1**

Результат ее сортировки можно увидеть уже после шага 0, так как больше обменов не будет. Порядок ключей 2a, 2b был изменен на 2b, 2a, так что метод неустойчив.

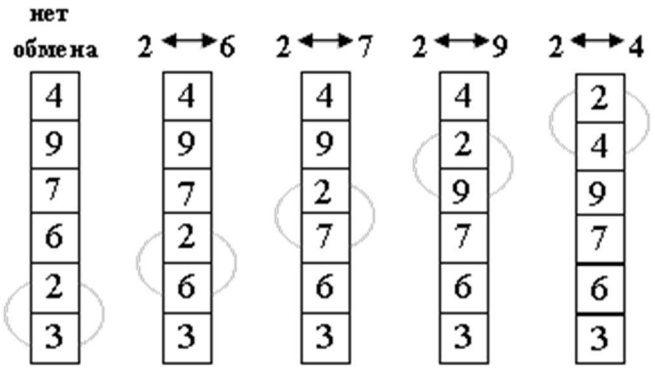
Если входная последовательность почти упорядочена, то сравнений будет столько же, значит алгоритм ведет себя *неестественно*.

*Метод пузырька*

Одним из простых, интуитивно понятных алгоритмов сортировки является алгоритм пузырька и его модификации.

Упорядочим массив по убыванию элементов. Для большей наглядности отобразим массив снизу-вверх, от нулевого элемента — к последнему.

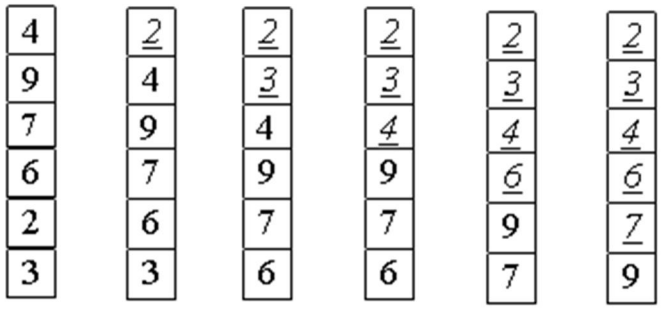
Идея метода: шаг сортировки состоит в проходе снизу-вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.




**Нулевой проход, сравниваемые пары выделены**

После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент — отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.




Среднее число сравнений и обменов имеют квадратичный порядок роста: $O(n^2)$, отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

Тем не менее, у него есть громадный плюс: он прост и его можно по-всякому улучшать, даже интуитивно.

Во-первых, рассмотрим ситуацию, когда на каком-либо из проходов *не произошло ни одного обмена*. Что это значит?

Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла (особенно, если массив был отсортирован с самого начала!).

Итак, первое улучшение алгоритма заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет — алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если *запоминать* не только сам факт обмена, но и индекс последнего обмена *k*. Действительно: все пары соседних элементов с индексами, меньшими *k*, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе *k*, вместо того чтобы двигаться до установленной заранее верхней границы *i*.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя *легкий пузырек* снизу поднимется наверх *за один проход*, *тяжелые* пузырьки опускаются с минимальной скоростью: *один шаг* за итерацию. Так что массив **2 3 4 5 6 1** будет отсортирован за *один проход*, а сортировка последовательности **6 1 2 3 4 5** потребует *пять проходов*.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда

называют "*шейкер-сортировкой*". Суть в том, что продвигаем сначала самых легкий элемент и запоминаем последний обмен. А потом двигаем самый тяжелый элемент и тоже запоминаем место последнего обмена. При каждом проходе эти индексы сближаются, а когда становятся равными, то алгоритм завершает работу.

Вопрос: насколько описанные изменения повлияли на эффективность метода? Среднее количество сравнений, хоть и уменьшилось, но остается $O(n^2)$, в то время как число обменов не поменялось вообще никак. Среднее (оно же худшее) количество операций остается квадратичным.

Дополнительная память, очевидно, не требуется. Поведение усовершенствованного (но не начального) метода довольно естественное, почти отсортированный массив будет отсортирован намного быстрее случайного. Сортировка пузырьком устойчива, однако шейкер-сортировка утрачивает это качество.

На практике метод пузырька, даже с улучшениями, работает, увы, слишком медленно. А потому — почти не применяется. Разве что для сортировки малых массивов, и если необходимо разработать подпрограмму сортировки “на ходу”.

*Сортировка вставками*

Сортировка простыми вставками в чем-то похожа на вышеизложенные методы. Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале “вырастает” отсортированная последовательность...

Однако в сортировке пузырьком или выбором можно было четко заявить, что на $i$-м шаге элементы $a[0]...a[i]$ стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: хотя последовательность $a[0]...a[i]$ упорядочена, но по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Рассмотрим действия алгоритма на $i$-м шаге. Итак, последовательность к этому моменту разделена на две части: готовую $a[0]...a[i]$ и неупорядоченную $a[i+1]...a[n]$.

На следующем, $(i+1)$-м каждом шаге алгоритма берем $a[i+1]$ и *вставляем на нужное место* в готовую часть массива.

Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.

В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо они меняются местами и процесс повторяется.

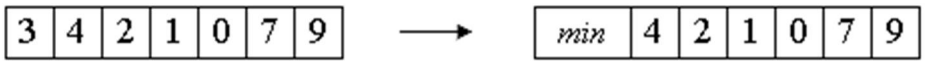

Таким образом, в процессе вставки мы "проталкиваем" элемент ***x*** к началу массива, останавливаясь в одном из случаев, когда

1. Найден элемент, меньший ***x***;
2. Достигнуто начало последовательности.

Аналогично сортировке выбором, среднее, а также худшее число сравнений и пересылок оцениваются как $O(n^2)$, дополнительная память при этом не используется.

Хорошим показателем сортировки является весьма естественное поведение: почти отсортированный массив будет ‘досортирован’ очень быстро. Это, вкупе с устойчивостью алгоритма, делает метод хорошим выбором в соответствующих ситуациях.

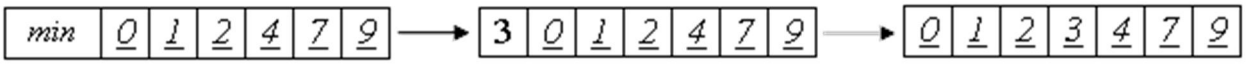
Алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла *проверяются два условия*. Можно объединить их в одно, поставив в начало массива специальный *сторожевой элемент*. Он должен быть заведомо меньше всех остальных элементов массива.




Тогда при *j*=0 будет заведомо верно *a*[0] <= *x*. Цикл остановится на нулевом элементе, что и было целью условия j>=0.

Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше. С учетом того, что

оно производилось $O(n^2)$ раз, это — реальное преимущество. Однако, отсортированный массив будет не полон, так как из него исчезло первое число. Для окончания сортировки — это число следует вернуть назад, а затем вставить в отсортированную последовательность *a*[1]...*a*[n].



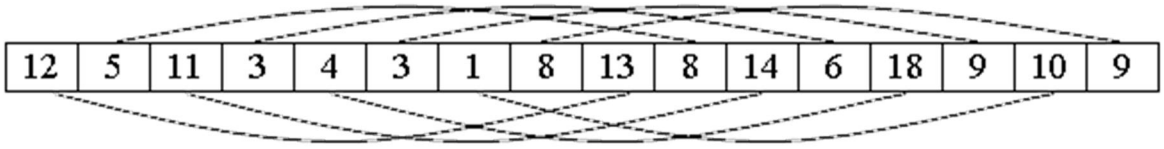

*Сортировка Шелла*

Сортировка Шелла является довольно интересной модификацией алгоритма сортировки простыми вставками.

Рассмотрим следующий алгоритм сортировки массива *a*[0]…*a*[15].

| 12 | 8 | 14 | 6 | 4 | 9 | 1 | 8 | 13 | 5 | 11 | 3 | 18 | 3 | 10 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

1. Вначале сортируем простыми вставками каждые *8 групп* из 2-х элементов (*a*[0], *a*[8[), (*a*[1], *a*[9]), ... , (*a*[7], *a*[15]).



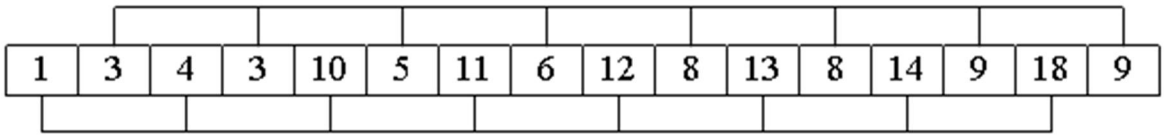

2. Потом сортируем каждую из *четырех групп* по 4 элемента (a[0], a[4], a[8], a[12]), ..., (a[3], a[7], a[11], a[15]).

*номер группы:*

| 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 1 | 3 | 12 | 5 | 10 | 6 | 13 | 8 | 11 | 8 | 18 | 9 | 14 | 9 |

В нулевой группе будут элементы 4, 12, 13, 18, в первой - 3, 5, 8, 9 и т.п.

3. Далее сортируем *2 группы* по 8 элементов, начиная с (*a*[0], *a*[2], *a*[4], *a*[6], *a*[8], *a*[10], *a*[12], *a*[14]).




4. В конце сортируем вставками все 16 элементов.

| 1 | 3 | 3 | 4 | 5 | 6 | 8 | 8 | 9 | 9 | 10 | 11 | 12 | 13 | 14 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Очевидно, лишь последняя сортировка необходима, чтобы расположить все элементы по своим местам. Так зачем нужны остальные?

На самом деле они продвигают элементы максимально близко к соответствующим позициям, так что в последней стадии число перемещений будет весьма невелико. Последовательность и так почти отсортирована. Ускорение подтверждено многочисленными исследованиями и на практике оказывается довольно существенным.

Единственной характеристикой сортировки Шелла является *приращение* — расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда равно единице — метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

Использованный в примере набор ..., 8, 4, 2, 1 — неплохой выбор, особенно, когда количество элементов — степень двойки. Однако гораздо лучший вариант предложил Р. Седжвик. Его последовательность имеет вид

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

При использовании таких приращений среднее количество операций: $O(n^{7/6})$, в худшем случае — порядка $O(n^{4/3})$.

Обратим внимание на то, что последовательность вычисляется в порядке, противоположном используемому: *inc*[0] = 1, *inc*[1] = 5, ... Формула дает сначала меньшие числа, затем все большие и большие, в то время как 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] > *size*.

Часто вместо вычисления последовательности во время каждого запуска процедуры, ее значения рассчитывают заранее и записывают в таблицу, которой пользуются, выбирая начальное приращение по тому же правилу: начинаем с *inc*[s–1], если 3**inc*[s] > *size*.

Сайты

http://algolist.manual.ru/sort/

http://algolist.manual.ru/sort/bubble_sort.php

http://algolist.manual.ru/sort/insert_sort.php

http://algolist.manual.ru/sort/shell_sort.php