**Чулюков В.А.**

# МЕТОДЫ РАЗРАБОТКИ ПРОГРАММ (АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ)

## ЧАСТЬ 3

# СОРТИРОВКИ


**Воронеж - 2015**

## Лекция № 1.

## ТЕМА: **Понятия сортировки. Простые методы сортировки массивов.**

### Основные вопросы, рассматриваемые на лекции:

1. Понятия и цели сортировки.
2. Сортировки массивов и сортировки файлов.
3. Требования к методам сортировки массивов.
4. Меры эффективности.
5. Сортировка простыми включениями.
6. Сортировка бинарными включениями.
7. Сортировка простым выбором.
8. Метод «пузырька».
9. Шейкер-сортировка.

### Краткое содержание лекционного материала

Рассмотрим и проанализируем простые методы сортировки, которые являются основными, именно на них основаны все известные методы упорядочивания данных в зависимости от условия задачи (по убыванию, по возрастанию, по алфавиту). Сортировка служит хорошим примером того, что одна и та же цель может достигаться с помощью различных алгоритмов, причем каждый из них имеет свои определенные преимущества и недостатки, которые нужно оценить с точки зрения конкретной ситуации.

Под сортировкой обычно понимают процесс перестановки объектов данного множества в определенном порядке. Цель сортировки – облегчить последующий поиск элементов в отсортированном множестве. В этом смысле элементы сортировки присутствуют почти во всех задачах. Упорядоченные объекты содержаться в телефонных книгах, в ведомостях подоходных налогов, в оглавлениях, в библиотеках, в словарях, на складах, да и почти всюду, где их нужно разыскивать.

Следовательно, методы сортировки очень важны, особенно при обработке данных. Казалось бы, что легче рассортировать, чем набор данных? Однако с сортировкой связаны многие фундаментальные приемы построения алгоритмов, которые и будут нас интересовать в первую очередь. Почти все такие приемы встречаются в связи с алгоритмами сортировки. В частности, сортировка является идеальным примером огромного разнообразия алгоритмов, выполняющих одну и ту же задачу, многие из которых в некотором смысле являются оптимальными, а большинство имеет какие-либо преимущества по сравнению с остальными. Поэтому на примере сортировки мы убеждаемся в необходимости сравнительного анализа алгоритмов. Кроме того, здесь мы видим, как при помощи усложнения эффективности по сравнению с более простыми и очевидными методами.

Зависимость выбора алгоритмов от структуры данных – настолько сильна, что методы сортировки обычно разделяют на две категории: сортировка массивов и

сортировка (последовательных) файлов. Эти два класса часто называют внутренней и внешней сортировкой, так как массивы располагаются во «внутренней» (оперативной) памяти ЭВМ; для этой памяти характерен быстрый произвольный доступ, а файлы хранятся в более медленной, но более вместительной «внешней» памяти, т.е. на запоминающихся устройствах с механическим передвижением (дисках и лентах). Это существенное различие можно наглядно показать на примере сортировки пронумерованных карточек.

Рис.1.1. Сортировка массива

Представление карточек в виде массива соответствует тому, что все они располагаются перед сортирующим так, что каждая карточка видна и доступна (см. рис. 1). Представление карточек в виде файла предполагает, что видна только верхняя карточка из каждой стопки (см. рис. 2). Очевидно, что такое ограничение приведет к существенному изменению методов сортировки, но оно неизбежно, если карточек так много, что их число на столе не уменьшается.

Рис.1.2**.** Сортировка файла

Прежде всего, мы введем некоторую терминологию и систему обозначений, которые будем использовать. Нам даны элементы

$$a_1, a_2, ..., a_n.$$

Сортировка означает перестановку этих элементов в таком порядке:

$$a_{k_1}, a_{k_2}, ..., a_{k_n}.$$

что при заданной функции упорядочения f справедливо решение

$$f(a_{k_1}) \le f(a_{k_2}) \le ... \le f(a_{k_n}) \qquad (1)$$

Обычно функция упорядочения не вычисляется по какому-то специальному правилу, а содержится в каждом элементе в виде явной компоненты (поля). Ее значение называется ключом элемента. Следовательно, для представления элемента ai особенно хорошо подходит структура записи. Поэтому мы определяем тип item (элемент), который будет использоваться в последующих алгоритмах сортировки следующим образом:

```
type item = record key: integer;
                   { описание других компонентов }        (2)
            end
```

« Прочие компоненты» - это все существенные данные об элементе; поле key - ключ служит лишь для идентификации элементов. Однако, когда мы говорим об алгоритмах сортировки, ключ для нас – единственная существенная компонента, и нет необходимости как-то определять остальные. Выбор в качестве типа ключа целого типа достаточно произволен; ясно, что точно так же можно использовать и любой тип, на котором задано отношение всеобщего порядка.

Метод сортировки называется устойчивым, если относительный порядок элементов с одинаковыми ключами не меняется при сортировке. Устойчивость сортировки часто бывает правильна, если элементы упорядочены (рассортированы) по каким-то вторичным ключам, т. е. по свойствам, не отраженным в первичном ключе.

*Сортировка массивов.*

Основное требование к методам сортировки массивов – экономное использование памяти. Это означает, что переупорядочение элементов нужно выполнять *in situ* (на том же месте) и что методы, которые пересылают элементы из массива *a* в массив *b*, не представляет для нас интереса. Таким образом, выбирая метод сортировки, руководствуясь критерием экономии памяти, классификацию алгоритмов мы проводим в соответствии сих эффективностью, т. е. экономией времени или быстродействием. Удобная *мера эффективности* получается при подсчете числа C – необходимых *сравнений ключей* и M – *пересылок элементов*. Эти числа определяются некоторыми функциями от числа n сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n \bullet \log n$ сравнений, мы обсудим несколько несложных и очевидных способов сортировки, называемых *простыми методами*, которые требуют порядка $n^2$ сравнений ключей. Рассмотрим простые методы по следующим трем причинам:

1. Простые методы особенно хорошо подходят для разъяснения свойств большинства принципов сортировки.

2. Программы, основанные на этих методах, легки для понимания и коротки. Следует помнить, что программы также занимают память!
3. Хотя сложные методы требуют меньшего числа операций, эти операции более сложны; поэтому при достаточно малых *n* простые методы работают быстрее, но их не следует использовать при больших *n*.

Методы, сортирующие элементы in situ, можно разбить на три основных класса в зависимости от лежащего в их основе приема:

Сортировка включениями.

Сортировка выбором.

Сортировка обменом.

Рассмотрим и сравним эти три принципа. Программы работают с переменной-массивом *a*, который нужно рассортировать in situ. В этих программах используются типы данных item (2) и index , определенные так:

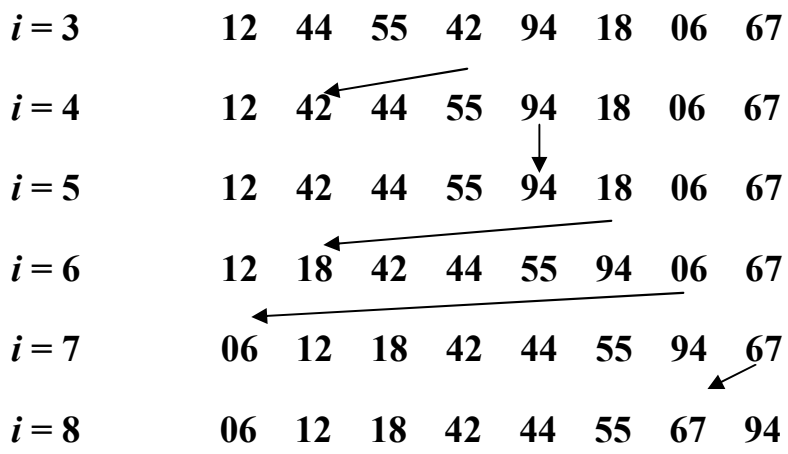
```
type index = 0 . . n;
var a: array [1 . . n]  of item                    (3)
```

## *Сортировка простыми включениями*

Этот метод обычно используют игроки в карты. Элементы (карты) условно разделяются на готовую последовательность $a_1,...,a_{i-1}$ и входную последовательность $a_i,...,a_n$. На каждом шаге, начиная с i = 2 и увеличивая i на единицу, берут i–й элемент входной последовательности и передают в готовую последовательность, вставляя его на подходящее место.

Таблица 1.1. Пример сортировки простыми включениями

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ***Начальные ключи*** | **44** | **55** | **12** | **42** | **94** | **18** | **06** | **67** |
| ***i* = 2** | **44** | **55** | **12** | **42** | **94** | **18** | **06** | **67** |




Процесс сортировки включениями показан на примере восьми случайно взятых чисел (см. табл. 1). Алгоритм сортировки простыми включениями выглядит следующим образом:

for ***i*** **:= 1** to **n** do

begin ***x*** **:= a[*i*];**

**«вставить *x* на подходящее место в** $a_1 ... a_i$ **»**

end

При поиске подходящего места удобно чередовать сравнения и пересылки, т. е. как бы «просеивать» x, сравнивая с очередным элементом $a_j$ и либо вставляя x, либо передается вправо и продвигаясь налево. Заметим, что «процесс просеивания » закончится при двух различных условиях:

1. Найден элемент $a_j$ с ключом меньшим, чем ключ у $x$.
2. Достигнут левый конец.

Такой повтор процесса с двумя условиями позволяет нам воспользоваться хорошо известным приемом фиктивного элемента («барьера»). Его можно легко приметить в этом случае. Установив барьер $a_0 = x$. (Заметим, что для этого нужно расширить диапазон индексов в описании a до 0, …, $n$.) Окончательно алгоритм представлен в виде программы 1.

```
procedure straightinsertion;
    var i, j: index; x: item;
begin
    for i:=2 to n do
```

```
begin x := a[i]; a[0] := x; j := i-1;
    while x.key < a[j].key do
        begin a[j+1] := a[j]; j := j-1;
        end;
    a[j+1]:= x
end;
end;
```

Программа 1.1. Сортировка простыми включениями

*Анализ сортировки простыми включениями*. Общее число сравнений C и пересылок M есть

$$C_{\min} = n-1 \qquad M_{\min} = 2(n-1)$$

$$C_{cp.} = \frac{1}{4}(n^2+n-2) \qquad M_{cp.} = \frac{1}{4}(n^2+9n-10) \qquad (4)$$

$$C_{\max} = \frac{1}{2}(n^2+n)-1 \qquad M_{\max} = \frac{1}{2}(n^2+3n-4)$$

Наименьшие числа появляются, если элементы с самого начала упорядочены, а наихудший случай встречается, если элементы расположены в обратном порядке. В этом смысле сортировка включениями демонстрирует вполне *естественное поведение*. Ясно также, что данный алгоритм описывает *устойчивую* сортировку: он оставляет неизменным порядок элементов с одинаковыми ключами.

## *Сортировка бинарными включениями*

Алгоритм сортировки простыми включениями легко можно улучшить, пользуясь тем, что готовая последовательность $a_1,...,a_{i-1}$, в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно быстрее. Очевидно, что здесь можно применить бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сортировки называется *сортировкой бинарными включениями*, он показан в программе 2.

```
procedure binaryinsertion;
    var i, j, l, r, m: index; x: item;
```

```
begin
    for i := 2 to n do
    begin x := a[i]; l := 1; r := i;
        while l < r do
        begin m := (l + r) div 2;
            if x.key >= a[m].key then l := m + 1 else r := m
        end;
        for j := i downto r+1 do a[j] := a[j-1];
        a[r] := x;
    end;
end;
```

Программа 1.2. Сортировка бинарными включениями

*Анализ сортировки бинарными включениями.*

Количество сравнений не зависит от исходного порядка элементов. Но из-за округления при делении интервала поиска пополам действительное число сравнений для i элементов может быть на 1 больше ожидаемого. Природа этого «перекоса» такова, что в результате места включения в нижней части находятся в среднем несколько быстрее, чем в верхней части. Это дает преимущество в тех случаях, когда элементы изначально далеки от правильного порядка. На самом же деле минимальное число сравнений требуется, если элементы вначале расположены в обратном порядке, а максимальное – если они уже упорядочены. Следовательно, это случай *неестественного* поведения алгоритма сортировки:

К сожалению, улучшение, которое мы получаем, используя метод бинарного поиска, касается только числа сравнений, а не числа необходимых пересылок. В действительности поскольку пересылка элементов, т. е. ключей и сопутствующей информации, обычно требует значительно больше времени, чем сравнение двух ключей, то это улучшение ни в коей мере не является решающим: важный показатель M по-прежнему остается порядка $n^2$ . И в самом деле, пересортировка уже рассортированного массива занимает больше времени, чем при сортировке простыми включениями с последовательным поиском! Этот пример показывает, что «очевидное улучшение» часто оказывается намного менее существенным, чем кажется в начале, и в некоторых случаях (которые *действительно* встречаются) может на самом деле

оказаться ухудшением. В конечном счете сортировка включениями оказывается не очень подходящим методом для цифровых вычислительных машин: включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна. Лучших результатов можно ожидать от метода, при котором пересылки элементов выполняются только для отдельных элементов и на большие расстояния. Эта мысль приводит к сортировке выбором.

### *Сортировка простым выбором*

Этот метод основан на следующем правиле:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом $a_1$.

Эти операции затем повторяются с оставшимися $n$ - 1 элементами, затем с $n$ - 2 элементами, пока не останется только один элемент – наибольший. Этот метод продемонстрирован на тех же восьми ключах в табл. 2.

Таблица 1.2. Пример сортировки простым выбором

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *Начальные ключи* | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| | 06 | 55 | 12 | 42 | 94 | 18 | 44 | 67 |
| | 06 | 12 | 55 | 42 | 94 | 18 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 67 | 94 |

Программу можно представить следующим образом:

**for** *i* := 1 **to** *n*-1 **do**

**begin** «присвоить *k* индекс наименьшего элемента из *a*[*i*]. . . *a*[*n*] »;

«поменять местами $a_i$ и $a_k$»

**end**

Этот метод, называемый *сортировкой простым выбором*, в некотором смысле противоположен сортировке простыми включениями; при сортировке простыми включениями на каждом шаге рассматривается только *один* очередной элемент *входной последовательности* и *все* элементы *готового массива* для нахождения места

включения; при сортировке простым выбором рассматриваются *все* элементы *входного массива* для нахождения элемента с наименьшим ключом, и этот *один* очередной элемент отправляется в *готовую последовательность*. Весь алгоритм сортировки простым выбором представлен в виде программы 3.

```
procedure straightselection;
  var i, j, k: index; x: item;
  begin for  i :=1 to n - 1 do
    begin k := i;  x :=  a[i];
                for j :=  i+ 1 to n do
              if a[j].key < x.key then
            begin k := j; x := a[j]
            end;
        a[k] := a[i];  a[i] := x;
    end
end
```

Программа 1.3. Сортировка простым выбором

*Анализ сортировки простым выбором*. Очевидно, что число C сравнений ключей не зависит от начального порядка ключей. В этом смысле можно сказать, что сортировка простым выбором ведет себя менее естественно, чем сортировка простыми включениями. Мы получаем

$$C = \frac{1}{2}(n^2 - n). \tag{5}$$

Минимальное число пересылок равно

$$M_{\min} = 3(n-1) \tag{6}$$

в случае изначально упорядоченных ключей и принимает наибольшее значение:

$$M_{\max} = trunc\left(\frac{n^2}{4}\right) + 3(n-1), \tag{7}$$

если вначале ключи расположены в обратном порядке. Среднее $M_{ср.}$ трудно определить, несмотря на простоту алгоритма. В некоторых источниках получено приближенное значение

$$M_{ср.} = n(\ln n + \gamma) \tag{8}$$

где $\gamma = 0.577216...$ – эйлерова константа.

Мы можем сделать вывод, что обычно алгоритм сортировки простым выбором предпочтительней алгоритма сортировки простыми включениями, хотя в случае, когда ключи заранее рассортированы или почти рассортированы, сортировка простыми включениями все же работает несколько быстрее.

## *Сортировка простым обменом*

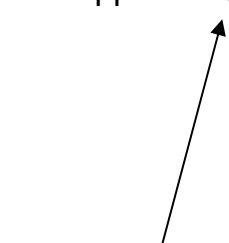
Классификация методов сортировки не всегда четко определена. Оба представленных ранее метода можно рассматривать как сортировку обменом. Однако в этой части мы остановимся на методе, в котором обмен двух элементов является основной характеристикой процесса. Приведенный ниже алгоритм сортировки простым обменом основан на принципе *сравнения и обмена* пары соседних элементов до тех пор, пока не будут рассортированы все элементы.

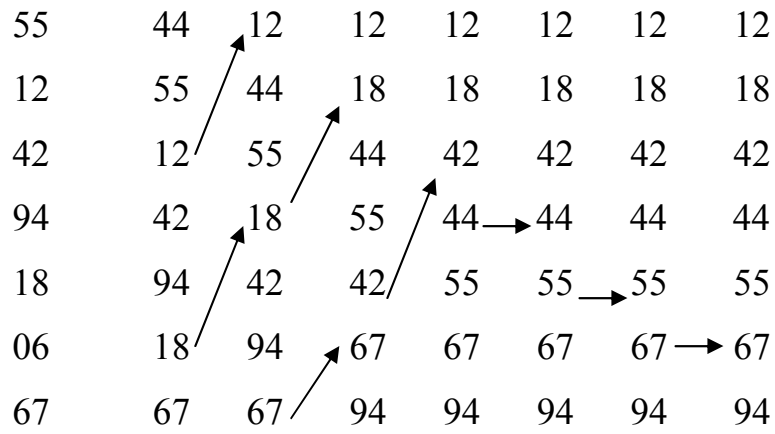
Как и в предыдущих методах простого выбора, мы совершаем повторные проходы по массиву, каждый раз просеивая наименьший элемент оставшегося множества, двигаясь к левому концу массива. Если, для разнообразия, мы будем рассматривать массив, расположенный вертикально, а не горизонтально и – при помощи некоторого воображения - представим себе элементы пузырьками в резервуаре с водой, обладающими «весами», соответствующими их ключам, то каждый проход по массиву приводит к «всплыванию» пузырька на соответствующий его весу уровень (см. табл. 3). Этот метод широко известен как *сортировка методом пузырька*. Его простейший вариант приведен в программе 4.

Таблица 1.3. Пример сортировки методом пузырька

| *Начальные ключи* | *i*=2 | *i*=3 | *i*=4 | *i*=5 | *i*=6 | *i*=7 | *i*=8 |
|---|---|---|---|---|---|---|---|
| 44 | 06 | 06 | 06 | 06 | 06 | 06 | 06 |








```
procedure bubblesort;
    var i, j: index; x: item;
 begin for i := 2 to n do
        begin for j := n downto i do
            if a[j-1].key > a[j].key then
            begin x := a[j-1]; a[j-1] := a[j]; a[j] := x
            end
        end
 end {bubblesort}
```

Программа 1.4. Сортировка методом пузырька

Этот алгоритм легко оптимизировать. Пример в табл. 3 показывает, что три последних прохода никак не влияют на порядок элементов, поскольку те уже рассортированы. Очевидный способ улучшить данный алгоритм – это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то это означает, что алгоритм можно продолжить, если запоминать не только сам факт обмена, но и место (индекс) последнего обмена. Ведь ясно, что все пары соседних элементов с индексами, меньшими этого индекса *k*, уже расположены в нужном порядке. Поэтому следующие проходы можно заканчивать на этом индексе, вместо того чтобы двигаться до установленной заранее нижней границы *i*. Однако внимательный программист заметит здесь странную асимметрию: один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывает на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только на один шаг на каждом проходе. Например, массив

12 18 42 44 55 67 94 06

будет рассортирован при помощи метода пузырька за один проход, а сортировка массива

94 06 12 18 42 44 55 67

потребует семи проходов. Эта неестественная асимметрия подсказывает третье улучшение: менять направление следующих один за другим проходов.

*Анализ сортировки методом пузырька.*

Число сравнений в алгоритме простого обмена равно

$$C = \frac{1}{2}(n^2 - n), \qquad (9)$$

минимальное, среднее и максимальное количества пересылок (присваиваний элементов) равны

$$M_{\min} = 0,\ M_{ср.} = \frac{3}{4}(n^2 - n),\ M_{\max} = \frac{3}{2}(n^2 - n). \qquad (10)$$

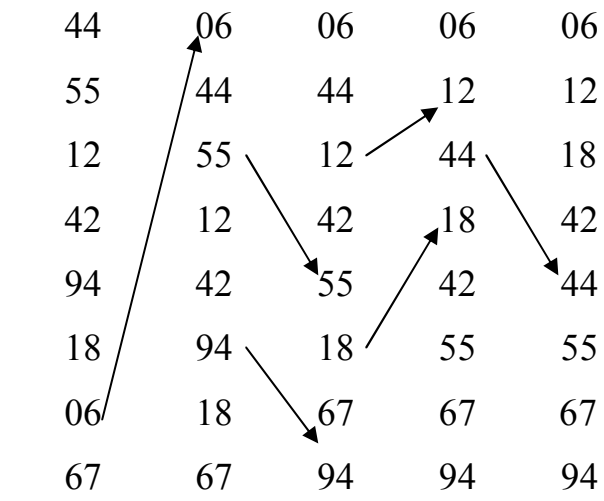
## *Шейкер-сортировка*

Если в методе пузырька менять направление следующих один за другим проходов, то полученный в результате алгоритм называют *шейкер-сортировкой*. Его работа показана в табл. 4 на тех же восьми ключах, которые использовались в табл. 3.

Анализ сортировки методом пузырька и шейкер-сортировки.

Таблица 1.4. Пример шейкер-сортировки

| | | | | |
|---|---|---|---|---|
| I = 2 | 3 | 3 | 4 | 4 |
| R = 8 | 8 | 7 | 7 | 4 |
| ↑ | ↓ | ↑ | ↓ | ↑ |




```
procedure shakersort;
    var j, k, l, r : index; x: item;
begin l := 2; r := n; k := n;
    repeat
        for j := r downto l do
            if a[j – 1].key > a[j].key then
            begin x := a[j – 1]; a[j – 1] := a[j]; a[j] := x; k := j
            end;
        l := k+1;
        for j := l to r do
            if a[j – 1].key > a[j].key then
            begin x := a[j – 1]; a[j – 1] := a[j]; a[j] := x; k := j
            end;
        r := k – 1;
    until l > r
end {shakersort};
```

Программа 1.5. Шейкер-сортировка

Анализ улучшенных методов, особенно метода шейкер-сортировки, довольно сложен. Наименьшее число сравнений есть $C_{\min} = n - 1$. Для усовершенствованного метода пузырька Кнут получил, что среднее число проходов пропорционально $n - k_1\sqrt{n}$ и среднее число сравнений пропорционально $\frac{1}{2}[n^2 - n(k_2 + \ln n)]$. Но мы замечаем, что все предложенные выше усовершенствования никоим образом не влияют на число обменов; они лишь уменьшают число избыточных повторных проверок. К сожалению, обмен двух элементов – обычно намного более дорогостоящая операция,

чем сравнения ключей, поэтому все наши усовершенствования дают значительно меньший эффект, чем можно было бы ожидать.

Анализ показывает, что сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором, и действительно, сортировка методом пузырька вряд ли имеет какие-то преимущества, кроме своего легко запоминающегося названия. Алгоритм шейкер-сортировки выгодно использовать в тех случаях, когда известно, что элементы уже почти упорядочены – редкий случай на практике.

Можно показать, что среднее расстояние, на которое должен переместиться каждый из $n$ элементов во время сортировки, - это $n/3$ мест. Это число дает ключ к поиску усовершенствованных, т. е. более эффективных, методов сортировки. Все простые методы в принципе перемещают каждый элемент на одну позицию на каждом элементарном шаге. Поэтому они требуют порядка $n^2$ таких шагов. Любое улучшение должно основываться на принципе пересылки элементов за один цикл на большее расстояние.