СОРТИРОВКА

Сортировка – это процесс упорядочивания информации в соответствии определенным правилом (критерием).

Одним из важных свойств алгоритмов сортировки является *устойчивость* - сортировка не меняет взаимного расположения равных элементов.

(Жираф, 800), (Слон, 1500), (Лиса, 40), (Волк, 90), (Кит, 3000), (Барсук, 40)

Вариант сортировки 1:

(Барсук, 40), (Лиса, 40), (Волк, 90), (Жираф, 800), (Слон, 1500), (Кит, 3000) – порядок не соблюден

Вариант сортировки 2:

(Лиса, 40), (Барсук, 40), (Волк, 90), (Жираф, 800), (Слон, 1500), (Кит, 3000) – порядок соблюден

Сортировка

Случаи сортировки:

1. При упорядочивании данные меняют свое местоположение
2. При упорядочивании некие идентификаторы данных меняют свое местоположение

Направления сортировки (для линейных структур данных):

1. По возрастанию (в алфавитном порядке)
2. По убыванию (в обратном алфавитному порядку).

Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях.

В противоположность им есть алгоритмы, которые используют структуру ключей для сортировки объектов.

Группы алгоритмов сортировки

1. Элементарные алгоритмы сортировки
2. Алгоритмы «быстрой» сортировки
3. Алгоритмы поразрядной сортировки
4. Алгоритмы слияния и сортировки слиянием
5. Алгоритмы пирамидальной сортировки

Элементарные алгоритмы сортировки

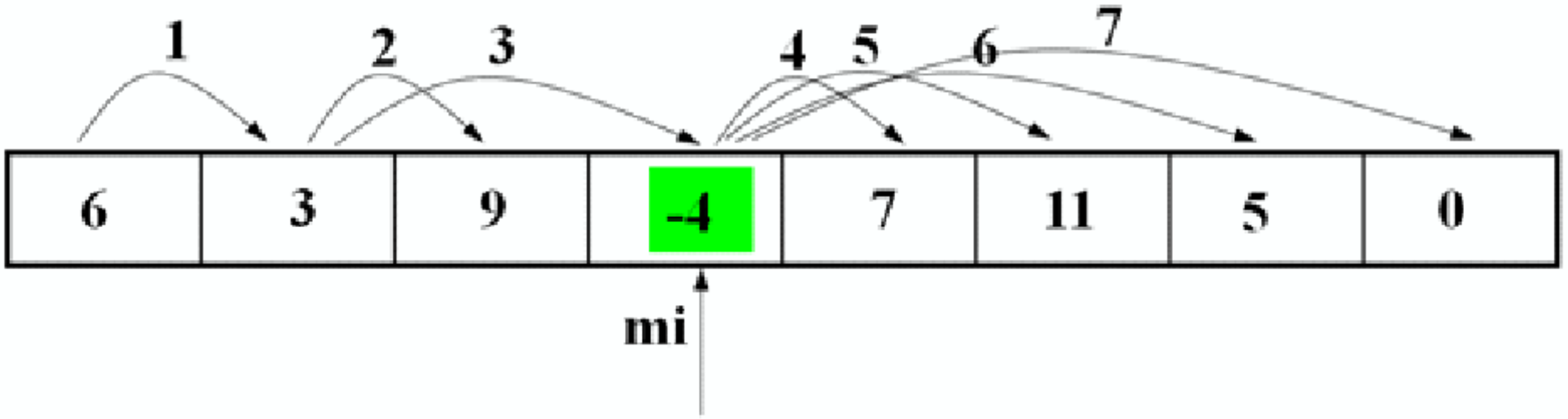
1. Алгоритм выборки
2. Алгоритм вставки
3. Алгоритм пузырька
4. Алгоритм Шелла
5. Алгоритм сортировки по индексам
6. Алгоритм сортировки по указателям (ссылке)
7. Алгоритм распределяющего подсчета
8. Алгоритм карманной сортировки

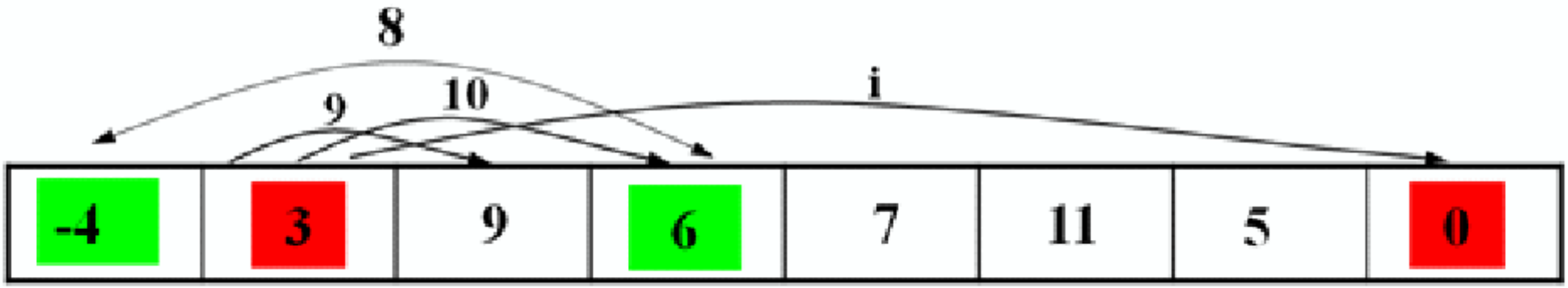
Алгоритм выборки

1) в наборе данных отыскивается наименьший (наибольший) элемент;
2) найденный элемент меняется местами с первым (крайний левый);
3) в оставшейся неупорядоченной части набора данных отыскивается следующий наименьший (наибольший) элемент;
4) найденный элемент меняется местами с крайним левым;
5) п.3 и п4 повторяются до тех пор, пока данные не будут упорядочены.

Недостаток –степень упорядоченности данных до сортировки оказывает очень малое влияния на время выполнения.

Сортировка





Алгоритм сортировки выборкою (по убыванию)

```
Selection_Sort(Array)
    for  k ← 0  to length(Array)   do
            maxPos  ←  k
            for  i  ←  k+1  to  length(Array) do
                    if  Array[i]  > Array[maxPos]  then
                            maxPos  ←  i
            temp  ←  Array[maxPos]
            Array[maxPos]  ←  Array[k]
            Array[k]  ←  temp
```

Сортировка

```
Enter value elements -> 9.9
-8.8
7.7
-1.1
2.2
-3.3
4.4
-5.5
6.6
0.0
before sort
9.9     -8.8    7.7     -1.1    2.2     -3.3    4.4     -5.5    6.6     0.0
after sort
9.9     7.7     6.6     4.4     2.2     0.0     -1.1    -3.3    -5.5    -8.8
```

Алгоритм вставки

1) Набор данных делится на две части: упорядоченную и неупорядоченную;
2) каждый элемент набора данных из неупорядоченной части анализируется и определяется место его размещения уже среди отсортированных данных;
3) все данные, которые должны стоять после него, смещаются вправо;
4) данный элемент помещается в освободившуюся позицию.

В отличие от сортировки алгоритмом выборки время выполнения зависит от исходного порядка упорядоченности данных.

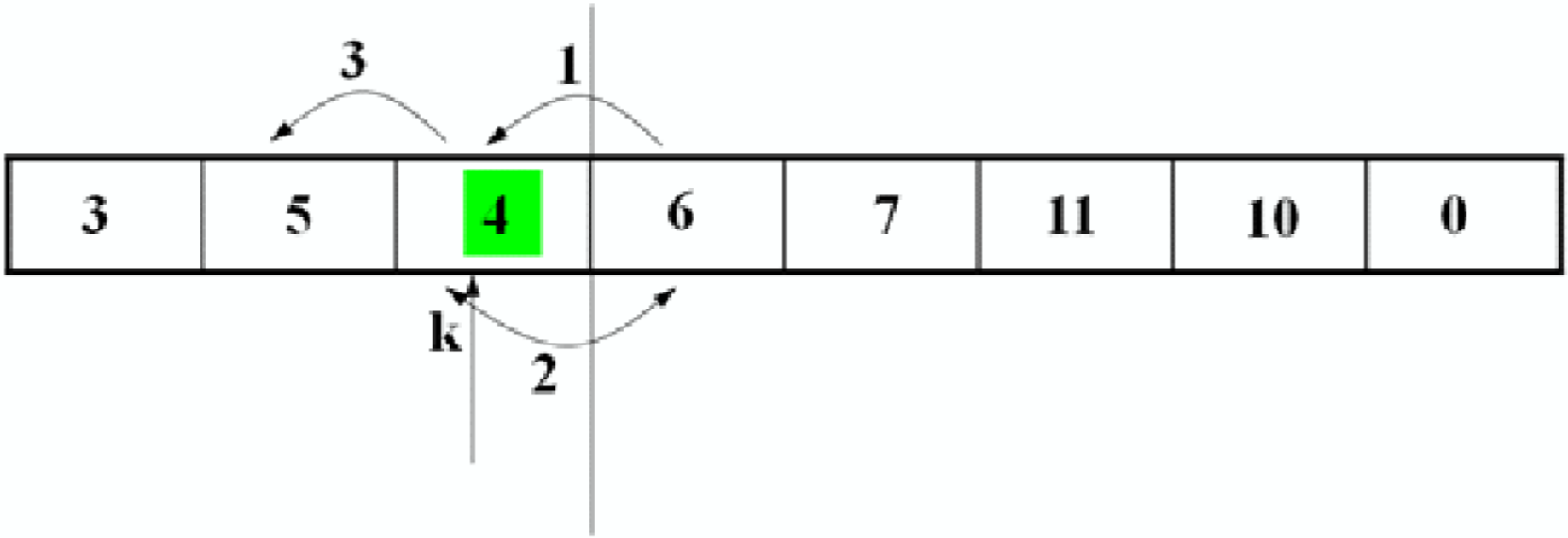
Алгоритм сортировки вставкою (по возрастанию)

Первый вариант

```
Insertion_Sort_1(Array)
    for k ← length(Array)-1 to 0 by -1 do
        if Array[k] < Array[k-1] then
            Array[k] ↔ Array[k-1]
    for i ← 2 to length(Array) do
        j ← i
        temp ← Array[i]
        while temp < Array[j-1] do
            Array[j] ← Array[j-1]
            j ← j-1
        Array[j] ← temp
```

```
8
before sort
6.0     4.0     3.0     7.0     9.0     1.0     2.0     0.0     5.0     8.0
after sort
0.0     1.0     2.0     3.0     4.0     5.0     6.0     7.0     8.0     9.0
```

Второй вариант реализации сортировки вставками:



Второй вариант

Insertion_Sort_2(*Array*)

for $k \leftarrow 1$ **to** *length*(Array) **do**

for $i \leftarrow k$ **to** 0 **by** -1 **do**

if *Array*[i] < *Array*[i-1] **then**

$temp \leftarrow Array[i]$

$Array[i] \leftarrow Array[i-1]$

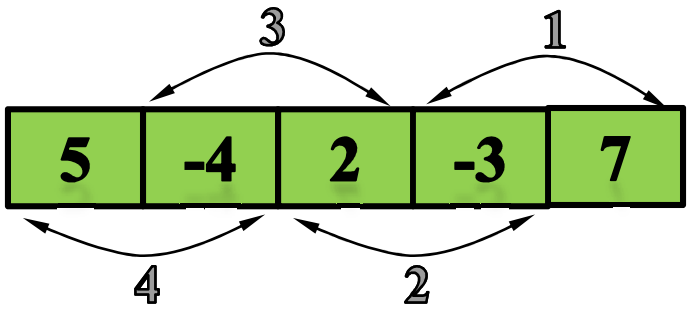
$Array[i-1] \leftarrow temp$

```
Enter value elememts -> -0.9
0.7
-.8
0.5
-.6
0.3
-0.2
0.0
-0.1
0.4
before sort
-0.9    0.7     -0.8    0.5     -0.6    0.3     -0.2    0.0     -0.1    0.4
after sort
-0.9    -0.8    -0.6    -0.2    -0.1    0.0     0.3     0.4     0.5     0.7
```

Алгоритм пузырька

- Просмотреть набор данных, начиная с последнего и двигаясь к первому с выполнением обмена соседних элементов, которые нарушают требуемый порядок;
- Повторить просмотр с обменом столько раз, сколько элементов в наборе данных.

Первый проход



Сортировка

Алгоритм сортировки пузырьком (по возрастанию)

Классический

Bubble_Sort(*Array*)

for $k \leftarrow 0$ **to** *length*(Array) **do**

for $i \leftarrow$ *length*(Array)-1 **to** 0 **by** -1 **do**

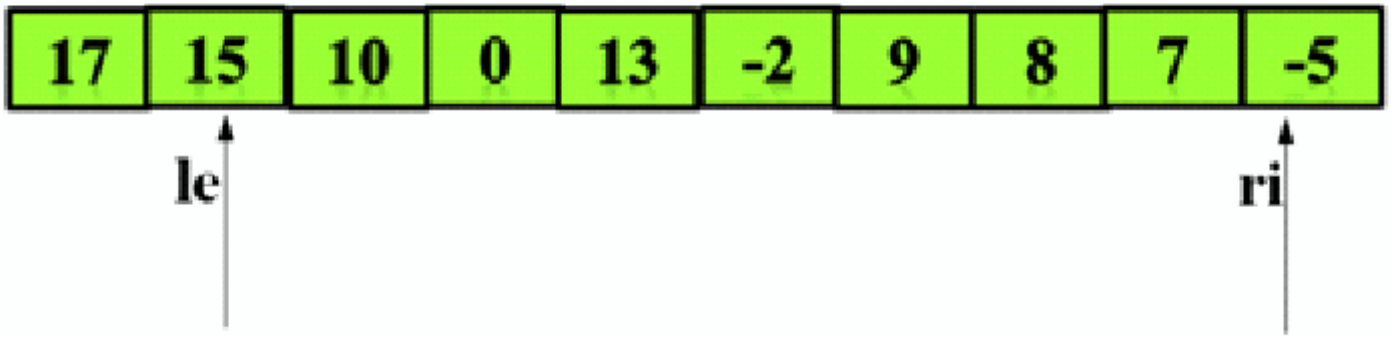
if *Array*[i] < *Array*[i-1] **then**

Array[i] $\leftrightarrow$ *Array*[i-1]

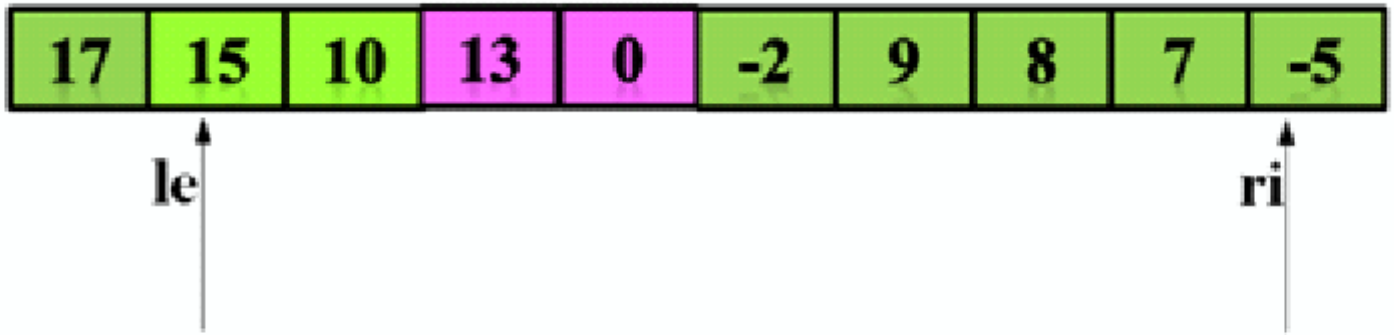
```
before sort
7.0    -9.0    0.0    -1.0    2.0    -3.0    4.0    -5.0    6.0    -7.0
after sort
-9.0   -7.0    -5.0    -3.0    -1.0    0.0    2.0    4.0    6.0    7.0
```

Алгоритм пузырька шейкерный

Первый проход (по убыванию)



Второй проход



```
Bubble_Shaker_Sort(Array)
        left ← 0
        right ← length(Array)-1
        while  left < right   do
              for  i ← right   to  left  by  -1  do
                    if   Array[i]  > Array[i-1]   then
                          Array[i]  ↔ Array[i-1]
             left ← left +1
             for  j ← left   to  right   do
                    if   Array[j]  < Array[j+1]   then
                          Array[j]  ↔ Array[j+1]
             right ← right - 1
```

Алгоритм Шелла

Суть – расширенный алгоритм сортировки вставкой, в котором осуществляется сравнение и обмен не столько соседних данных, а и находящихся на некотором расстоянии друг от друга.

Набор данных разбивается на независимые поднаборы, которые являются пересекающимися, и в каждом поднаборе используется сортировка алгоритмом вставки.

Сортировка

4-сортировка

1 поднабор: 5, 4 (индексы 0, 4);

2 поднабор: 6, 8 (индексы 1, 5);

3 поднабор: 3, 7 (индексы 2, 6);

4 поднабор: 0, 10 (индексы 3, 7);

2-сортировка

1 поднабор: 4, 3, 5, 7 (индексы 0, 2, 4, 6);

2 поднабор: 6, 0, 8, 10 (индексы 1, 3, 5, 7);

На практике используются убывающие последовательности шагов близкие к геометрической прогрессии (т.е. число шагов находится в логарифмической зависимости от размера набора данных).

Кнут предложил следующую последовательность шагов:

1 4 13 40 121 364 1093 3280

(h = h*3+1)

Сортировка

Алгоритм сортировки Шелла (по возрастанию)

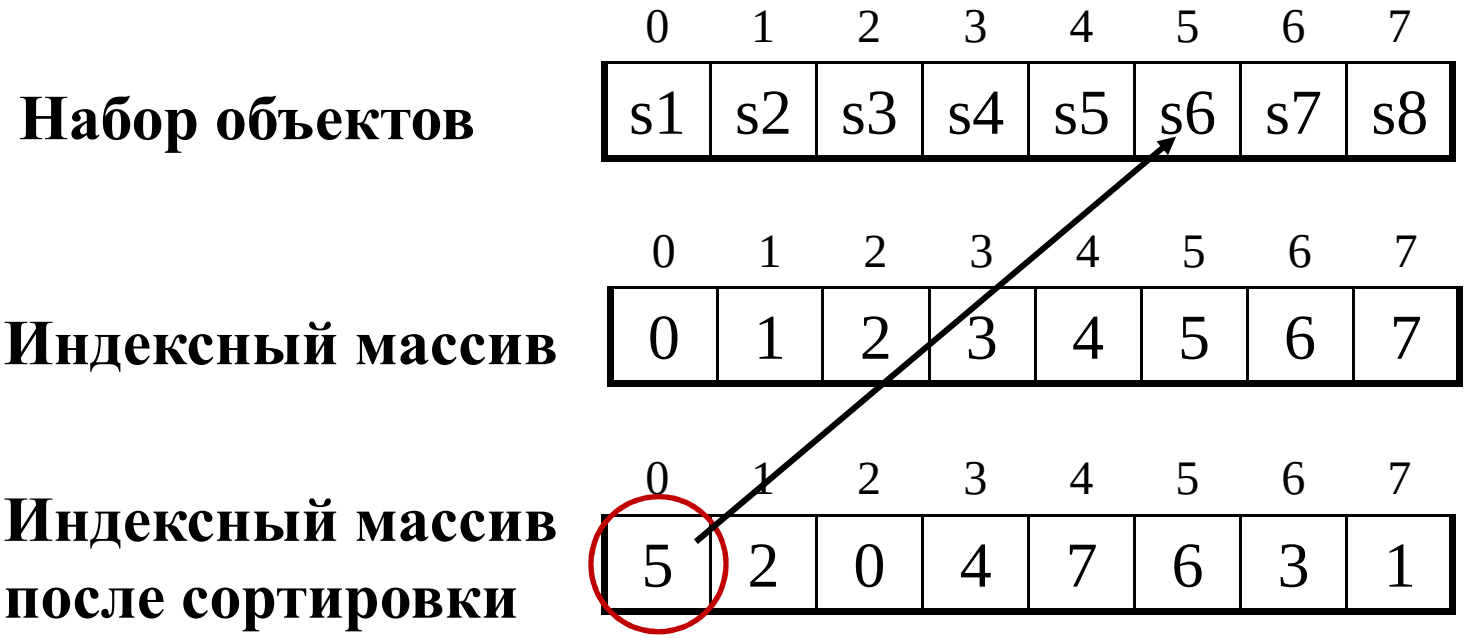
```
Shell_Sort(Array)
    h ← 1
    while h ≤ length(Array) / 9   do
            h ← (3 * h + 1)
   while  h > 0  do
            for  i ← h  to  length(Array)   do
                    key ← Array[i]
                    j ← i
                    while  j ≥ h  and   key < Array[j - h]   do
                            Array[j] ← Array[j - h]
                            j ← j − h
                    Array[j] ← key
            h ← h / 3
```

```
before sort
3.0     5.0     8.0     4.0     1.0     3.0     9.0     0.0     5.0     1.0
after sort
0.0     1.0     1.0     3.0     3.0     4.0     5.0     5.0     8.0     9.0
```

Алгоритм сортировки по индексам

Применяется для сложных и больших типов данных.

Суть – сортировке подлежит не сам набор данных, а индексный массив (элементы – это индексы элементов набора данных).



Сортировка

```
class Ob {
        String name;
        void prin() {...}
        ……….          }
public class Sort_5 {
   public static void sort(Ob m[], int a[])  {          int h;
      for (int k=0; k<a.length; k++)
         for(int i=k+1; i<a.length; i++)
            if ((m[a[k]].name).compareTo(m[a[i]].name) > 0)  {
               h = a[k];      a[k] = a[i];         a[i]=h;     }
    }
    public static void main(String [] args) throws IOException {
        Ob mas[] = new Ob [7];        int ind[] = new int [7];
        .....
         sort(mas, ind);
         for(int i=0; i<ind.length; i++)
            mas[ind[i]].prin();          }   }
```

Сортировка

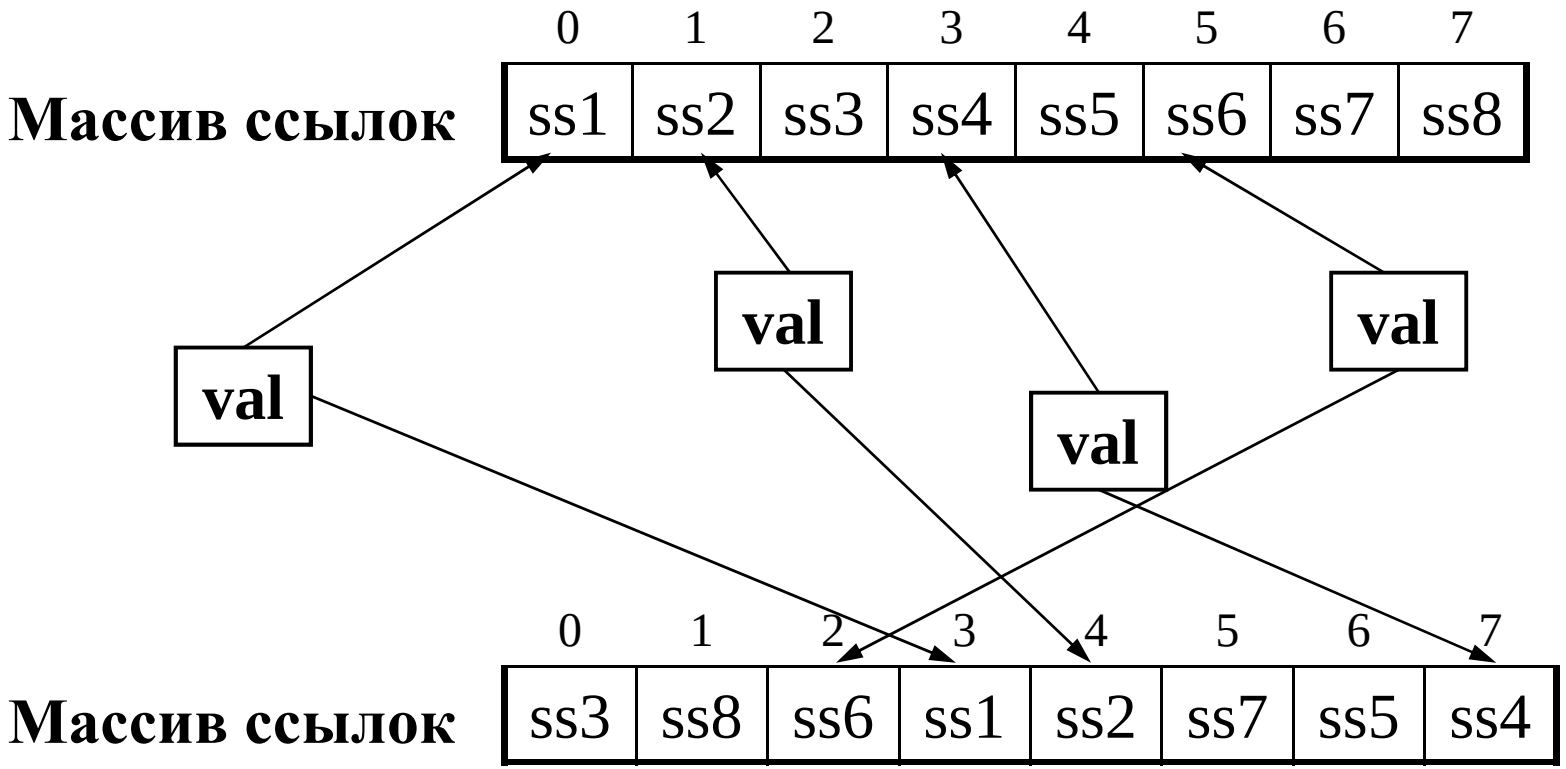
```
before sort
name=aaaaa      nomer=1 ball=85.0;
name=qqqq       nomer=2 ball=86.0;
name=rrrr       nomer=3 ball=87.0;
name=ccccc      nomer=4 ball=88.0;
name=hhhhh      nomer=5 ball=89.0;
name=nnnn       nomer=6 ball=90.0;
name=ooooo      nomer=7 ball=91.0;

after sort
name=aaaaa      nomer=1 ball=85.0;
name=ccccc      nomer=4 ball=88.0;
name=hhhhh      nomer=5 ball=89.0;
name=nnnn       nomer=6 ball=90.0;
name=ooooo      nomer=7 ball=91.0;
name=qqqq       nomer=2 ball=86.0;
name=rrrr       nomer=3 ball=87.0;

C:\Program Files\Java\jdk1.5.0\bin>_
```

Алгоритм сортировки по указателям (ссылке)

Суть – сортировке подлежит не сам набор данных, а массив ссылок (указателей на элементы набора данных)



Сортировка

```
class Std {
    private String name;
    private int groupe;
    private double mark;
   Std(String mm) {
       name = mm;
       groupe = 101 + (int)(Math.random()*8);
       mark = 60 + Math.random()*40;
   }
   public String toString() {
         return (name + "\t" + String.valueOf(groupe) + "\t" +
                                    String.valueOf(mark)); }
   public double getMark() {
       return mark;
    }
}
```

```
public static void sort51(Std mm[]) {
        Std temp;
        int p;
        for(int j=0; j<=mm.length-1; j++) {
             p = j;
            for(int k=j+1; k<=mm.length-1; k++)
                  if (mm[k].getMark() < mm[p].getMark())
                        p = k;
            temp = mm[j];
            mm[j] = mm[p];
            mm[p] = temp;
           }
    }
```

Massiv object:

aaaaaaaa	108	89.59
wwwwwwww	105	65.60
hhhhhhhh	101	81.78
vvvvvvvv	101	66.69

Massiv sort object:

wwwwwwww	105	65.60
vvvvvvvv	101	66.69
hhhhhhhh	101	81.78
aaaaaaaa	108	89.59

Алгоритм распределяющего подсчета

Применяется для наборов данных, элементы которых идентифицируются ключом.

Ключ – это целое положительное значение из диапазона [0; M].

Количество элементов в наборе данных больше чем количество возможных значений ключей, т.е. ключ не уникален.

Суть

1. Подсчитать количество элементов (объектов) для каждого значения ключа;
2. Вычислить частичные суммы (для каждого значения ключа подсчитывается количество объектов со значениями ключа меньшими данного);
3. Переписать в дополнительный массив объекты в соответствии с вычисленными частичными суммами;
4. Перезаписать исходный массив объектов, используя дополнительный массив.

Сортировка

Индексный массив, содержит ключи в диапазоне [0; 3]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3	3	0	1	1	0	3	0	2	0	1	1	2	0

	0	1	2	3
Количество ключей по каждому значению	**6**	**4**	**2**	**3**
Частичные суммы ключей для каждого значения	**0**	**6**	**10**	**12**

Частичные суммы представляют собой позицию (индекс) в массиве ключей, с которого располагается ключ с соответствующим значением

0	1	2	3	4	5	**6**	7	8	9	**10**	11	**12**	13	14
0	0	0	0	0	0	1	1	1	1	2	2	3	3	3

Count_Sort(*Array*, *m*)

► описать массив ключей *key[]* размерностью *m*

► описать дополнительный массив *dop[]* размерностью *Array[]*

for *i* ← 0 **to** *length*(Array) **do**

if **not** *nomer*(Array[i]) = m-1 **then**

key[*nomer*(Array[i]) + 1] ← *key*[*nomer*(Array[i]) + 1] + 1

for *j* ← 1 **to** *length*(key) **do**

key[j] ← *key*[j] + *key*[j-1]

for *i* ← 0 **to** *length*(Array) **do**

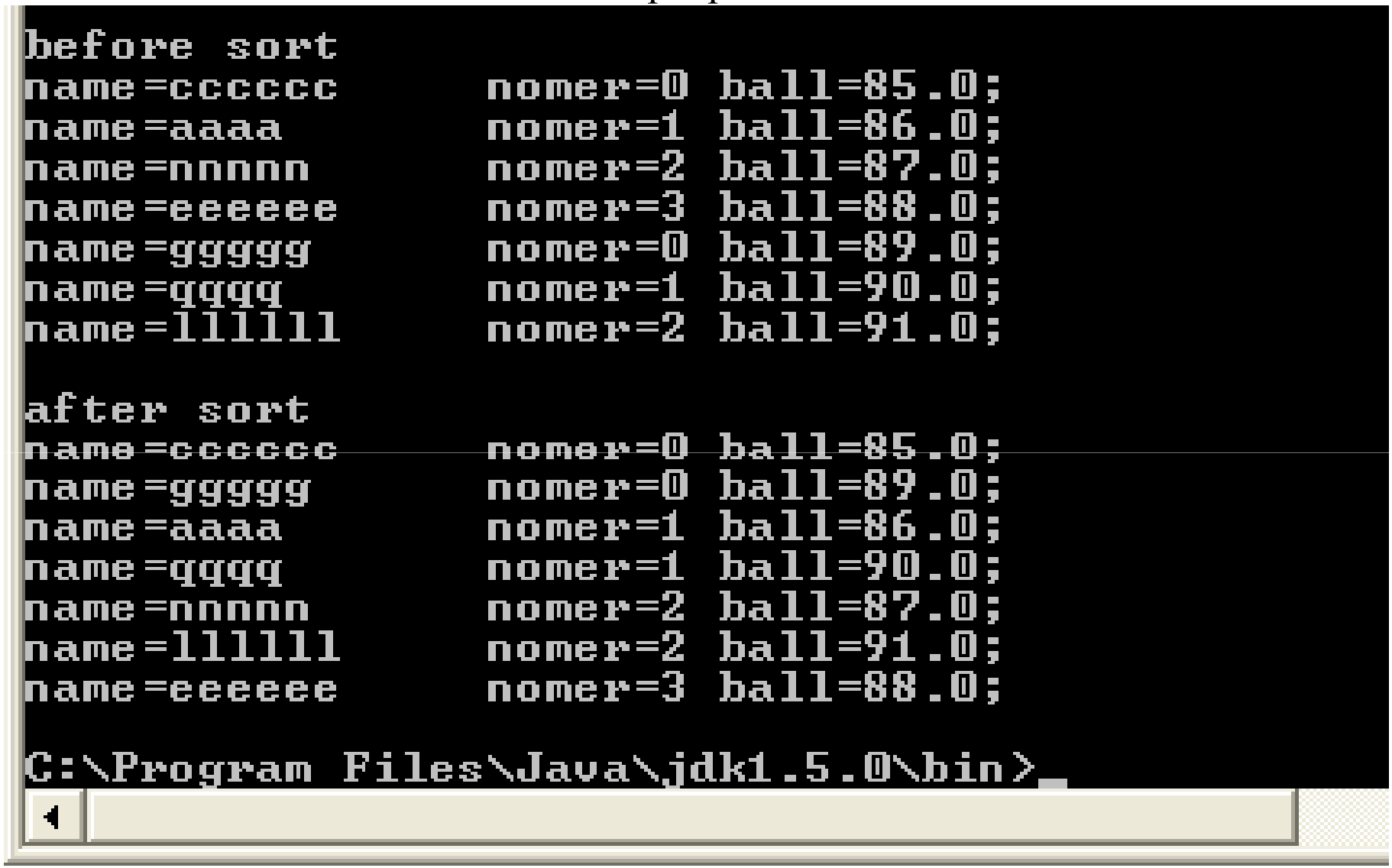
dop[*key*[*nomer*(Array[i])]] ← *Array*[i]

key[*nomer*(Array[i])] ← *key*[*nomer*(Array[i])] + 1

for *i* ← 0 **to** *length*(Array) **do**

Array[i] ← *dop*[i]

Сортировка


before sort
name=cccccc nomer=0 ball=85.0;
name=aaaa nomer=1 ball=86.0;
name=nnnnn nomer=2 ball=87.0;
name=eeeeee nomer=3 ball=88.0;
name=ggggg nomer=0 ball=89.0;
name=qqqq nomer=1 ball=90.0;
name=llllll nomer=2 ball=91.0;
after sort
name=cccccc nomer=0 ball=85.0;
name=ggggg nomer=0 ball=89.0;
name=aaaa nomer=1 ball=86.0;
name=qqqq nomer=1 ball=90.0;
name=nnnnn nomer=2 ball=87.0;
name=llllll nomer=2 ball=91.0;
name=eeeeee nomer=3 ball=88.0;
C:\Program Files\Java\jdk1.5.0\bin>

Алгоритм карманной сортировки

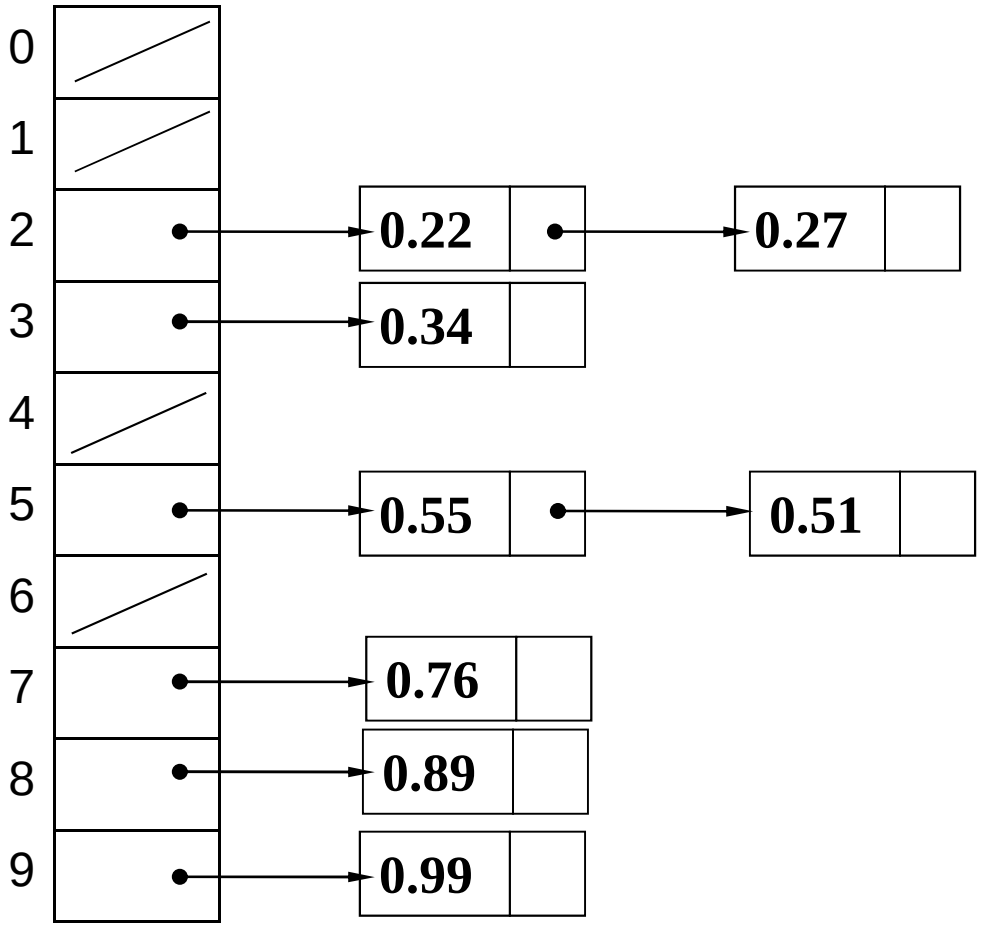
Чаще всего применяется для значений, находящихся в диапазоне [0, 1].

Суть – диапазон значений набора данных делится на одинаковые интервалы (части). Каждый интервал представляет собой карман (яму), в который будут вкладываться в виде однонаправленного списка элементы набора данных.

1. Диапазон значений разбивается на интервалы и по их количеству создается одномерный массив.
2. Элементы набора данных распределяются по интервалам в виде списка.
3. Каждый карман сортируется любым алгоритмом.
4. Последовательно все элементы вынимаются и переписываются в набор данных

Сортировка

0.22	0.34	0.27	0.89	0.76	0.55	0.99	0.51



Bucket_Sort(*Array*)

$n \leftarrow$ *length*(Array)

$k \leftarrow$ *interval*

for $i \leftarrow 0$ **to** n **do**

- Вставка *Array*[i] в список *B*[p(Array[i])]

for $i \leftarrow 0$ **to** k **do**

- Сортировка списка *B[i]* любым алгоритмом

- Сбор списков *B[0], B[1], . . B[k-1]* в один.

Пример классической карманной сортировки:

```
public class Bucket {
    private double mm[];

        class Share {
                    double dd;
                    Share next;
            Share(double val) {
                    dd = val;
            }
        }
```

```
public void sort() {
        Share kar[] = new Share [10],     newel;
        int p;                              // индекс кармана
        for(int i=0; i<mm.length; i++) {
            p = (int)(mm[i]*10);
            newel = new Share(mm[i]);
            kar[p] = add(kar[p], newel);
        }
        for(int j=0; j<kar.length; j++)
            if (kar[j] != null)
                kar[j] = sortSp(kar[j]);
        move(kar);
    }
```

Сортировка

```
private Share add(Share cur, Share obj) {
        obj.next = cur;
        return obj;
    }
private Share sortSp(Share cur) {
        Share nn = cur;
        int count = 0;
        while (nn != null) {
            count++;
            nn = nn.next;
        }
        if (count == 1)   return cur;
        for(int r=0; r < count; r++)
            cur = srt(cur);
        return cur;
    }
```

```
private Share srt(Share obj) {
        if (obj.next != null)     obj.next = srt(obj.next);
        else     return obj;
        if (obj.dd > obj.next.dd) {
            Share temp = obj.next;
            obj.next = obj.next.next;
            temp.next = obj;
            return temp;
        }
        else
            return obj;
}
```

```
private void move(Share k[], double m[]) {
        int p = 0;
        Share curr;
        for(int i=0; i<k.length; i++) {
            curr = k[i];
            while (curr != null) {
                m[p++] = curr.dd;
                curr = curr.next;
            }
        }
}
}
```

Результат:

Enter size massiv -> 8

Enter value elememt 1 -> 0.23

Enter value elememt 2 -> 0.12

Enter value elememt 3 -> 0.07

Enter value elememt 4 -> 0.21

Enter value elememt 5 -> 0.67

Enter value elememt 6 -> 0.29

Enter value elememt 7 -> 0.62

Enter value elememt 8 -> 0.05

Massiv before sort:

0.23 0.12 0.07 0.21 0.67 0.29 0.62 0.05

Massiv after sort:

0.05 0.07 0.12 0.21 0.23 0.29 0.62 0.67

Пример карманной сортировки, в которой распределение элементов по карманам осуществляется упорядочено:

```
public void sort() {
    Dop kar[] = new Dop [10], newel;
    int p;
    for(int i=0; i<mm.length; i++) {
        p = (int)(mm[i]*10);
        newel = new Dop(mm[i]);
        if(kar[p] == null)
            kar[p] = newel;
```

```
else  {     Dop cur = kar[p];
            if (cur.dd > newel.dd) {
                newel.next = cur;
                kar[p] = newel;
             } else {
                   while(cur.next != null)
                      if (cur.next.dd > newel.dd) {
                         newel.next = cur.next;
                         cur.next = newel;
                         break;
                      } else   cur = cur.next;
                   if (cur.next == null)
                      cur.next = newel;
                   }        }        }
```

```
        p = 0;
        for(int i=0; i<kar.length; i++)
            if (kar[i] != null) {
                Dop curr = kar[i];
                while (curr != null) {
                    mm[p++] = curr.dd;
                    curr = curr.next;
                }
            }
}
```

Enter value elememt -> 0.22
Enter value elememt -> 0.45
Enter value elememt -> 0.39
Enter value elememt -> 0.12
Enter value elememt -> 0.41
Enter value elememt -> 0.47
Enter value elememt -> 0.88

Massiv before sort:
0.22 0.45 0.39 0.12 0.41 0.47 0.88

Massiv after sort:
0.12 0.22 0.39 0.41 0.45 0.47 0.88