

# Основы программирования

## Эффективные алгоритмы сортировки

# Задача сортировки элементов массива

Дан массив значений  $A_0A_1A_2 \dots A_{n-2}A_{n-1}$ .

Необходимо найти такую перестановку

$i_0i_1i_2 \dots i_{n-2}i_{n-1}$  индексов, что для последовательности  $A_{i_0}A_{i_1}A_{i_2} \dots A_{i_{n-2}}A_{i_{n-1}}$  выполняется  $A_{i_k} \leq A_{i_{k+1}} \quad \forall k = 0 \dots n - 2$ .

$i_0i_1i_2 \dots i_{n-2}i_{n-1}$  - это некоторая перестановка чисел  $0 \dots n - 1$ , поэтому общее количество потенциальных решений задачи равно числу перестановок из  $n$  элементов, т.е.  $n!$

Минимальная гарантированная трудоемкость в наихудшем для сортировки, основанной на сравнениях:  $T_{min}(n) = \lceil \log(n!) \rceil = O(n \log n)$ .

Алгоритмы с такой трудоемкостью мы будем считать **эффективными**.

# Рекуррентное слияние (снизу вверх)

Пусть  $s$  – текущая длина серий в массиве (в исходном массиве  $n$  серий и  $s = 1$ ).

**Проход** в сортировке слиянием:

- массив  $A$  содержит  $k$  текущих серий длины  $s$
- пары смежных серий сливаются в  $\lfloor k/2 \rfloor$  серий длины  $2s$  в рабочий массив  $D$
- новые серии копируются из  $D$  в  $A$ .

Проходы повторяются при  $s = 1, 2, 4, \dots$ , пока  $s < n$  (т.е.  $k > 1$ ).

Общее число проходов составляет  $\lceil \log n \rceil$ .

На каждом проходе в слиянии участвуют все  $n$  элементов, поэтому  $T(n) = O(n \log n)$

# Рекуррентное слияние (снизу вверх)

В общем случае  $n \neq 2^k$ , поэтому на любом проходе возможны варианты:

- число текущих серий нечетно
- последняя серия имеет длину  $< s$ .

Поэтому при слиянии серий необходимо учесть, что 2-я серия может быть короче 1-й или вообще пустой.

В приведенном ниже алгоритме вычисляются индексы **b**, **c** и **e**, которые задают границы сливаемых серий: **A[b...c]** и **A[c+1...e]**.

# Рекуррентный алгоритм слияния

```
void merge_sort(double *A, int n)
{
    int s, b, c, e;
    double *D = new double[n];
    for (s = 1; s < n; s *= 2) {
        for (b = 0; b < n; b += s*2) {
            c = min(b+s-1, n-1);
            e = min(c+s, n-1);
            merge_series(A, b, c, e, D);
        }
        for (b = 0; b < n; b++) A[b] = D[b];
    }
    delete [] D;
}
```

# Сортировка Шелла

Основана на сортировке вставками (или обменной):

$\forall i, i = 1 \dots n - 1$ , элемент  $i$  добавляется к уже отсортированной последовательности элементов  $0 \dots i - 1$ . Для этого  $x = A_i$  сравнивается и меняется с  $A_j, j = i - 1 \dots 0$ , пока  $A_j > x$ .



Общее число сравнений для всех  $i$ :

$$T_{best}(n) = n - 1 = O(n)$$

$$T_{worst}(n) = \frac{n(n-1)}{2} = O(n^2).$$

Чем меньше в массиве инверсий, тем быстрее он сортируется.

# Сортировка Шелла: $h$ -цепочки

Зафиксируем  $h$ ,  $1 < h < n/2$  и рассмотрим  $h$ -цепочки – последовательности элементов с индексами:

$0, h, 2h, 3h, \dots$

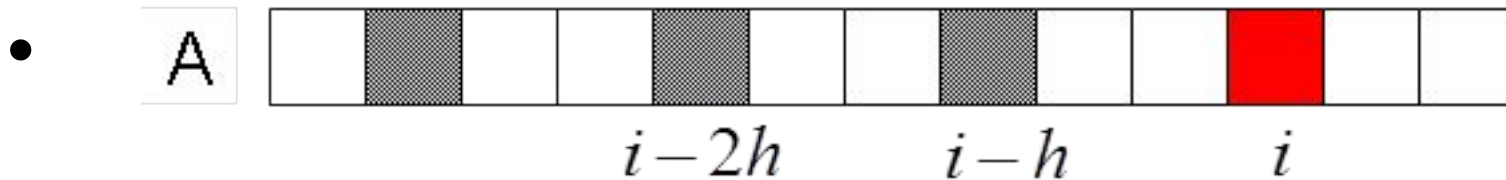
$1, h + 1, 2h + 1, 3h + 1, \dots$

$\dots$

$h - 1, 2h - 1, 3h - 1, 4h - 1, \dots$

Всего будет  $h$  цепочек длиной  $\leq \lceil n/h \rceil$ .

# Сортировка вставкам с шагом $h$



Сортировка одной цепочки:  $T_{worst}(n) = O\left(\left(\frac{n}{h}\right)^2\right)$ ,

сортировка всех цепочек:  $T_{worst}(n) = O\left(\frac{n^2}{h}\right)$ .

После упорядочения всех цепочек с шагом  $h$  массив не будет отсортирован, но число инверсий в нем уменьшится. Если повторить этот проход с шагом, меньшим  $h$ , то инверсий станет еще меньше и, соответственно, уменьшится трудоемкость выполнения следующего прохода.



# Сортировка Шелла: идея и требования

**Идея:** сортировка всех  $h$ -цепочек с последовательным уменьшением значений  $h$ :  $h_1 > h_2 > \dots > h_t = 1$ . С каждым проходом массив становится все ближе к упорядоченному, поэтому и трудоемкость проходов будет уменьшаться.

## **Требования:**

- $t$  (число проходов) должно быть небольшим
- $h_{k+1}$ -цепочки должны максимально перемешиваться с  $h_k$ -цепочками (чтобы при следующем проходе сравнивались элементы из разных цепочек предыдущего прохода).

# Сортировка Шелла: выбор шага $h$

Д. Кнут показал:

1. Выбор шага  $h_k = 2h_{k+1}$  - неудачный, т.к. при этом  $T_{worst}(n) = O(n^2)$ ,  $T_{mid}(n) = O(n\sqrt{n})$ .
2. Цепочки хорошо перемешиваются при выборе взаимно простых  $h$ .
3. Хорошие последовательности для значений  $h$ :  
 $h_{k-1} = 2h_k + 1, t = \lfloor \log_2 n \rfloor + 1$   
 $h_{k-1} = 3h_k + 1, t = \lfloor \log_3 n \rfloor + 1$ .

При выборе таких последовательностей:

$$T_{worst}(n) = O(n\sqrt{n}), \quad T_{mid}(n) = O(n \cdot \log^2 n).$$

# Сортировка Шелла: алгоритм

Используется последовательность  $h_{k-1} = 3h_k + 1$ , начальное значение  $h$  вычисляется таким образом, чтобы начальные цепочки содержали  $\geq 3$  элементов.

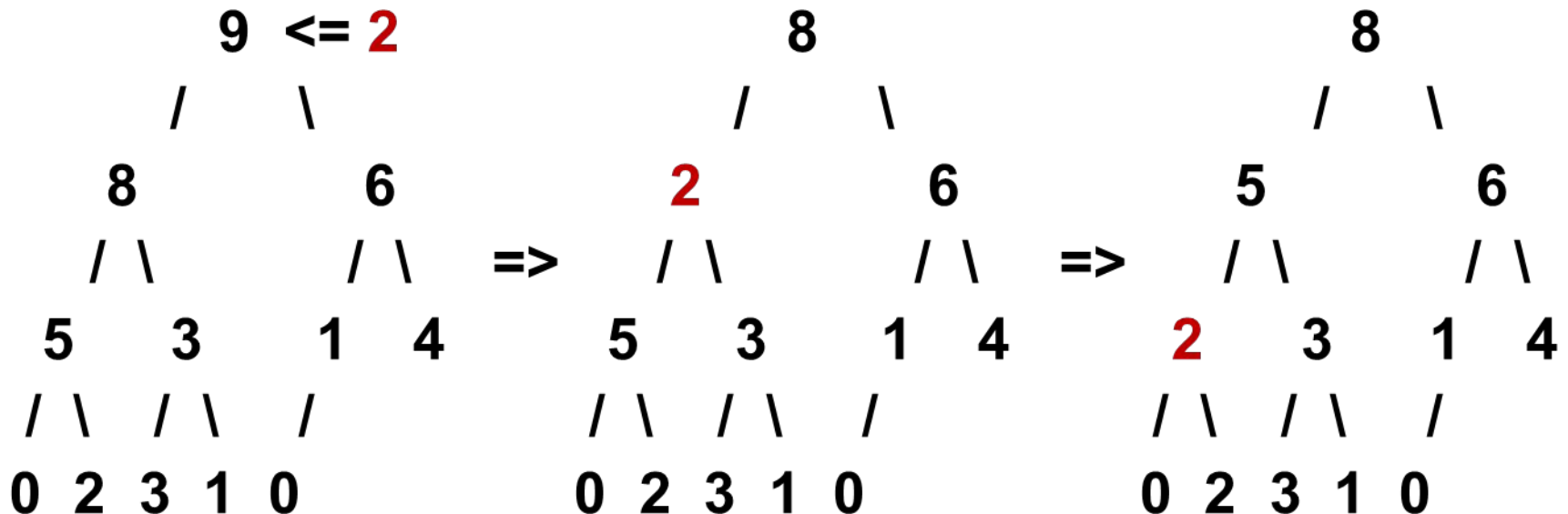
```
void shell_sort(double *A, int n)
{
    int i, j, h;
    for (h = 1; h <= n / 9; h = h * 3 + 1);
    while (h >= 1)
    {
        for (i = h; i < n; i++)
            for (j=i-h; j>=0&& A[j]>A[j+h]; j-=h)
                swap(A[j], A[j+h]);
        h = (h - 1) / 3;
    }
}
```

# Пирамидальная сортировка

В алгоритме строится **пирамида (бинарная куча)**, которую можно представить в виде бинарного дерева:

- каждой вершине соответствует элемент массива
- каждая вершина имеет  $\leq 2$  вершин-сыновей
- заполнены все уровни, кроме, возможно, последнего
- **значение-отец всегда не меньше значений-сыновей.**

**Просеивание в пирамиде** (если нарушено условие):



# Свойства пирамиды (бинарной кучи)

1. В корне пирамиды располагается максимальный элемент.
2. Если пирамида имеет  $k$  уровней и все уровни заполнены, то общее число вершин

$$n = 1 + 2^1 + 2^2 + \dots + 2^{k-1} = 2^k - 1.$$

1. Если  $2^{k-1} \leq n < 2^k$ , то  $k$ -й уровень заполнен не до конца.
2. Пирамида с  $n$  вершинами имеет  $\lceil \log n \rceil$  уровней, поэтому максимальное число сравнений при просеивании

$$T_{worst}(n) = 2\lceil \log n \rceil = O(\log n).$$

5. Пирамиду можно построить непосредственно на массиве:
  - для любого элемента-отца  $A_i$ ,  $0 \leq i \leq n/2$  сыновьями будут  $A_{2i+1}$  и  $A_{2i+2}$
  - условие пирамиды:  $\forall i, 0 \leq i \leq n/2 \quad A_i \geq A_{2i+1}, A_i \geq A_{2i+2}$

# Идея сортировки

Пусть на массиве длины  $n$  построена пирамида и номер ее последнего элемента  $m = n - 1$ .

Если поменять местами элементы с индексами 0 и  $m$ , то максимальный элемент встанет на свое (последнее) место в упорядоченном массиве.

Для 0-го элемента условие пирамиды будет нарушено. Чтобы восстановить пирамиду, этот элемент нужно просеять, не изменяя положение максимального элемента  $m$ , т.е. в пирамиде длины  $m$ .

Далее необходимо повторить эти действия для всех значений  $m$ , убывающих от  $n - 1$  до 1.

Трудоёмкость сортировки:  $T_{worst}(n) = O(n \log n)$

# Построение пирамиды

При построении пирамиды также проводится просеивание элементов. Просеивать можно только **в пирамиде**, поэтому она будет строиться **снизу вверх**:

- элементы с номерами  $\frac{n}{2} + 1 \dots n - 1$  не имеют потомков и образуют **нижний уровень** пирамиды (их просеивать не нужно)
- элементы с номерами от  $n/2$  до 0 последовательно просеиваются в уже построенных частях пирамиды.

# Трудоёмкость построения пирамиды

Пусть пирамида имеет  $k$  уровней, и все они заполнены.  
Тогда:

- на уровне  $k$  находятся  $2^{k-1} \approx n/2$  вершин, которые не надо просеивать
- на уровне  $k - 1$  находятся  $2^{k-2} \approx n/4$  вершин, которые при просеивании могут сместиться только на 1 уровень
- на уровне  $k - 2$  находятся  $2^{k-3} \approx n/8$  вершин, которые при просеивании могут сместиться только на 2 уровня

и т.д. вплоть до 1-го уровня с 1 вершиной.



# Трудоёмкость построения пирамиды

Таким образом, максимальное число уровней, которые могут пройти все вершины, составляет (при  $n \rightarrow \infty$ ):

$$\begin{aligned} \frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + 4\frac{n}{32} \dots &= \frac{n}{4} \left( 1 + 1 + \frac{3}{4} + \frac{4}{8} + \frac{5}{16} \dots \right) = \\ &= \frac{n}{4} \left[ \left( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \right) + \left( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \right) + \left( \frac{1}{4} + \frac{1}{8} + \dots \right) + \dots \right] = \\ &= \frac{n}{4} \left( 2 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \right) = n \end{aligned}$$

Следовательно, трудоёмкость построения пирамиды в наихудшем:  $T_{worst}(n) = O(n)$ .

# Функция просеивания в пирамиде

Параметры и переменные функции `sift`:

`i` – начальный номер просеиваемого элемента,

`m` – номер конечного элемента в текущей пирамиде,

`j` – текущий номер просеиваемого элемента,

`k` – номер левого или большего сына `j`.

```
void sift(double *A, int i, int m)
{
    int j = i, k = i*2+1;    // левый сын
    while (k <= m)
    {
        if (k<m && A[k]<A[k+1]) k++; // больший сын
        if (A[j] < A[k])
        { swap(A[j], A[k]); j = k; k = k*2+1; }
        else break;
    }
}
```

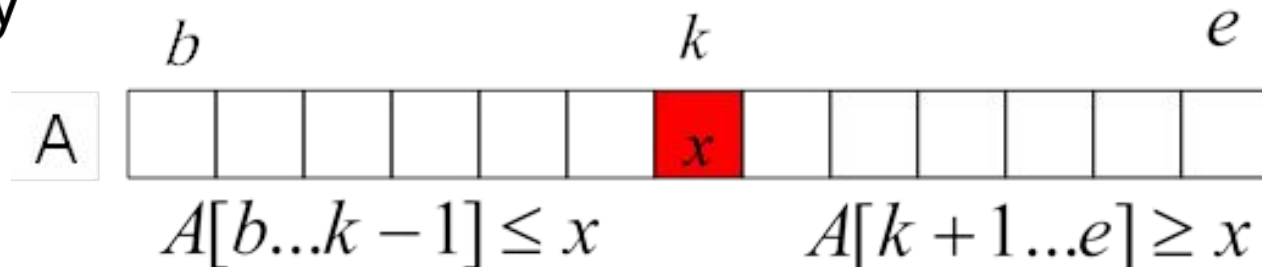
# Алгоритм пирамидальной сортировки

```
void heap_sort(double *A, int n)
{
    int i, m;
    // построение пирамиды
    for (i = n/2; i >= 0; i--)
        sift(A, i, n-1);
    // сортировка массива
    for (m = n-1; m >= 1; m--)
    {
        swap(A[0], A[m]);
        sift(A, 0, m-1);
    }
}
```

# Быстрая сортировка (Хоар)

В быстрой сортировке проводится рекурсивная обработка массива и его отдельных частей. При каждом рекурсивном вызове задаются границы текущей части. Обозначим индексы ее начального и конечного элементов  $b$  и  $e$  (при первом вызове  $b = 0, e = n - 1$ ).

Основной шаг сортировки – разделение текущей части массива опорным элементом: выбирается некоторый элемент  $x \in A[b \dots e]$  и текущая часть приводится к виду



# Быстрая сортировка

После разделения элемент  $x$  окажется на своем месте в упорядоченном массиве, а части  $A[b \dots k - 1]$  и  $A[k + 1 \dots e]$  можно сортировать рекурсивно и независимо.

Разделение массива длины  $n$  необходимо проводить за  $cn$  элементарных шагов, при этом:

- наилучшее разделение – 2 подмассива длины  $n/2$  и  
$$T_{best}(n) = cn + 2T\left(\frac{n}{2}\right) = O(n \log n)$$
- наихудшее – 1 подмассив длины  $n - 1$  (второй будет пустым) и

$$T_{worst}(n) = cn + T(n - 1) = O(n^2)$$

# Быстрая сортировка: трудоемкость в среднем

$T_{best}(n)$  и  $T_{worst}(n)$  отличаются в порядках, поэтому нужно оценить трудоемкость в среднем:

- различные случаи соответствуют различным позициям  $k = \overline{0 \dots n - 1}$
- вероятности для всех случаев равны  $1/n$
- $T(n) = cn + T(k) + T(n - k - 1)$  для любого фиксированного  $k$ .

Таким образом, **трудоемкость в среднем**:

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n - k - 1)) = cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k)$$

# Трудоёмкость в среднем: доказательство

Положим  $T(0) = T(1) = b = \text{const}$  и  $d = 2(b + c)$ .

Покажем, что  $T(n) \leq d \ln n$ ,  $\forall n \geq 2$ .

**Доказательство** (мат. индукция):

1. Базис  $n = 2$ :  $T(n) = 2c + 2b = d < d \cdot 2 \ln 2 \approx 1.4d$

2. Пусть  $\forall k = \overline{2 \dots n-1}$  выполняется  $T(k) \leq dk \ln k$ ,

$$\text{тогда } T(n) = cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \leq cn + \frac{4b}{n} + \frac{2d}{n} \sum_{k=2}^{n-1} k \ln k$$

# Трудоемкость в среднем: доказательство

- **Оценка для суммы:** 
$$\sum_{k=2}^{n-1} k \ln k \leq \int_2^n x \ln x dx = \left( \frac{x^2 \ln x}{2} - \frac{x^2}{4} \right)_2^n$$

Здесь интеграл взят по частям:  $\int v du = uv - \int u dv$ ,

и в нашем случае  $u = \frac{x^2}{2}$ ,  $v = \ln x$ .



# Трудоёмкость в среднем: доказательство

Таким образом:

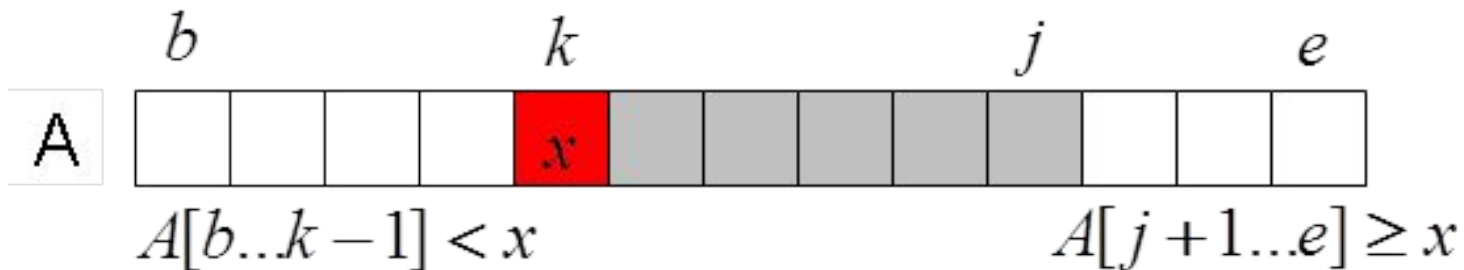
$$\begin{aligned} T(n) &= cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \leq cn + \frac{4b}{n} + \frac{2d}{n} \sum_{k=2}^{n-1} k \ln k \leq \\ &\leq cn + \frac{4b}{n} + \frac{2d}{n} \left( \frac{n^2 \ln n}{2} - 2 \ln 2 - \frac{n^2}{4} + 1 \right) = \\ &= cn + \frac{4b}{n} + dn \ln n - (b+c)n - \frac{2d(2 \ln 2 - 1)}{n} < dn \ln n = O(n \log n) \end{aligned}$$

# Разделение массива: 1-й способ

Текущая разделяемая часть массива:  $A[b \dots e]$ .

$k$  – текущий индекс опорного элемента (начальные значения  $k = b$ ,  $x = A[b]$ ).

$j$  – самый правый непроверенный элемент (начальное значение  $j = e$ ).



```
k = b; j = e; x = A[b];
```

```
while (k < j)
```

```
    if (A[j] >= x) j--;
```

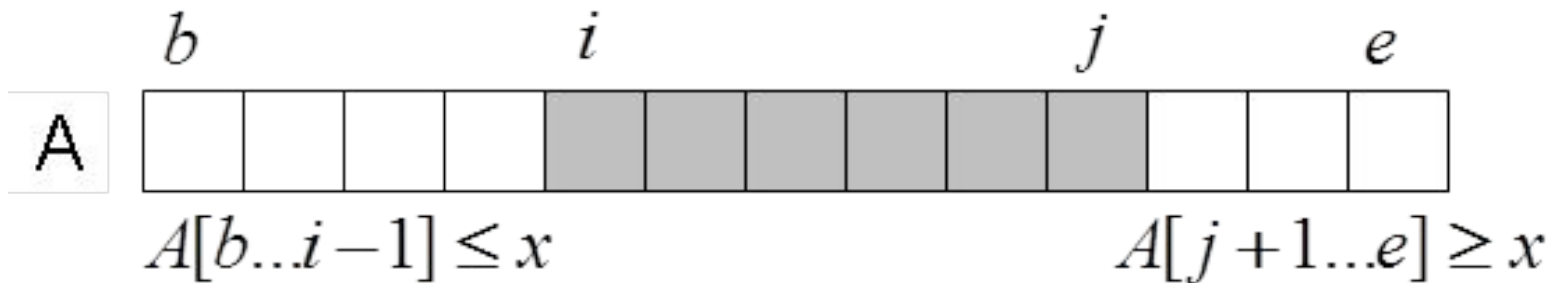
```
    else
```

```
        { A[k]=A[j]; A[j]=A[k+1]; A[k+1]=x; k++; }
```

## Разделение массива: 2-й способ

Опорный элемент  $x$  можно выбрать на любой позиции разделяемой части, его индекс не важен.

$i$  и  $j$  – левая и правая границы непроверенной части (начальные значения  $i = b$ ,  $j = e$ ).



Пока  $i < j$ :

1. Пропускаются все  $A[i] < x$  с увеличением  $i$  на 1
2. Пропускаются все  $A[j] > x$  с уменьшением  $j$  на 1
3. Если  $i \leq j$ , то  $A[i]$  и  $A[j]$  меняются местами,  $i$  увеличивается,  $j$  уменьшается на 1.

# Быстрая сортировка с 2 рекурсивными вызовами

```
void quick_sort_2(double *A, int b, int e)
{
    double x; int i, j;
    x = A[(b+e)/2]; i = b; j = e;
    while (i < j)
    {
        while (A[i] < x) i++;
        while (A[j] > x) j--;
        if (i <= j) {
            { swap(A[i], A[j]); i++; j--; }
        }
        if (b < j) quick_sort_2(A, b, j);
        if (i < e) quick_sort_2(A, i, e);
    }
}
```

# Быстрая сортировка с 1 рекурсивным вызовом

В **наихудшем случае** опорный элемент  $x$  после разделения текущей части всегда оказывается либо в позиции  $b$ , либо в позиции  $e$ , т.е. нужно рекурсивно сортировать либо  $A[b + 1 \dots e]$ , либо  $A[b \dots e - 1]$ .

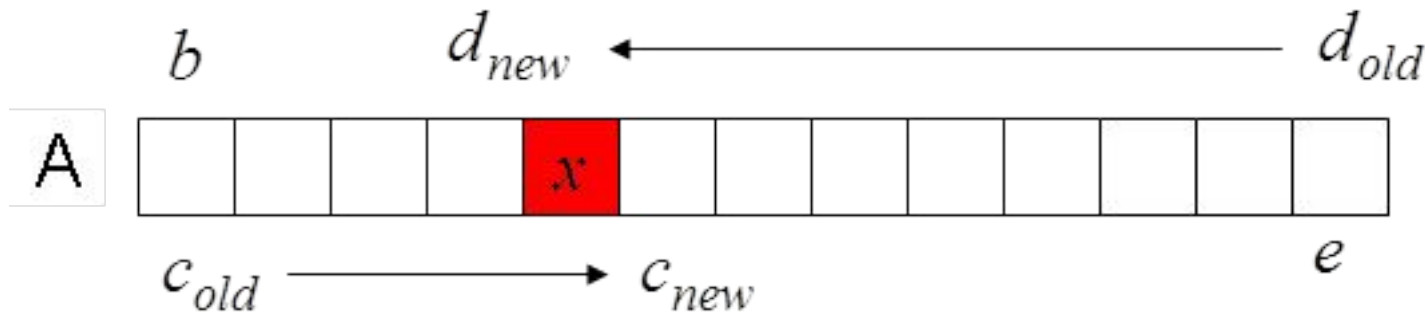
В этом случае не только  $T(n) = O(n^2)$ , но и **глубина рекурсии составит  $n$** . При каждом рекурсивном вызове в стеке выделяется память для параметров и внутренних переменных функции, и в **наихудшем случае** потребуется памяти в 5-6 раз больше, чем занимает исходный массив.

Для уменьшения глубины рекурсии нужно избавиться от рекурсивной обработки **большой из 2 частей**, полученных в результате разделения.

# Быстрая сортировка с 1 рекурсивным вызовом

Идея сортировки с 1 рекурсивным вызовом:

- устанавливаются текущие границы  $c = b$  (нижняя) и  $d = e$  (верхняя),
- текущая часть массива делится опорным элементом на 2 части,
- **меньшая часть сортируется рекурсивно,**
- большая часть становится текущей – для этого изменяется либо  $c$  (большая часть справа), либо  $d$  (большая часть слева),
- обработка продолжается в цикле, **пока  $c < d$ .**



# Быстрая сортировка с 1 рекурсивным вызовом

```
void quick_sort(double *A, int b, int e)
{ double x; int i, j, c = b, d = e;
  while (c < d) {
    x = A[(c+d)/2]; i = c; j = d;
    while (i < j) {
      while (A[i] < x) i++;
      while (A[j] > x) j--;
      if (i <= j)
        { swap(A[i], A[j]); i++; j--; }
    }
    if (j-c < d-i)
      { if (c < j) quick_sort(A, c, j); c = i; }
    else { if (i < d) quick_sort(A, i, d); d = j; }
  }
}
```

# Свойства алгоритмов сортировки

1. Сравнение алгоритмов сортировки по  $T_{mid}$ :  
быстрая < слияние < пирамидальная
2. Выбор при различных  $n$ :  
десятки – простые алгоритмы,  
сотни или несколько тысяч – алгоритм Шелла
3. Сортировка называется устойчивой, если она сохраняет порядок следования равных элементов:  
пусть в исходном массиве существуют  $A_i = A_j$ ,  $i < j$ ,  
и после сортировки  $A_i$  займет позицию  $i_s$ ,  $A_j$  –  $j_s$ .  
Если при этом  $i_s < j_s$ , то сортировка устойчива.



## Поиск $k$ -го минимального элемента

**Задача:** в массиве  $A[0 \dots n - 1]$  найти  $k$ -й по значению элемент (т.е. элемент, который стоял бы на позиции  $k$ , если бы массив был упорядочен по возрастанию).

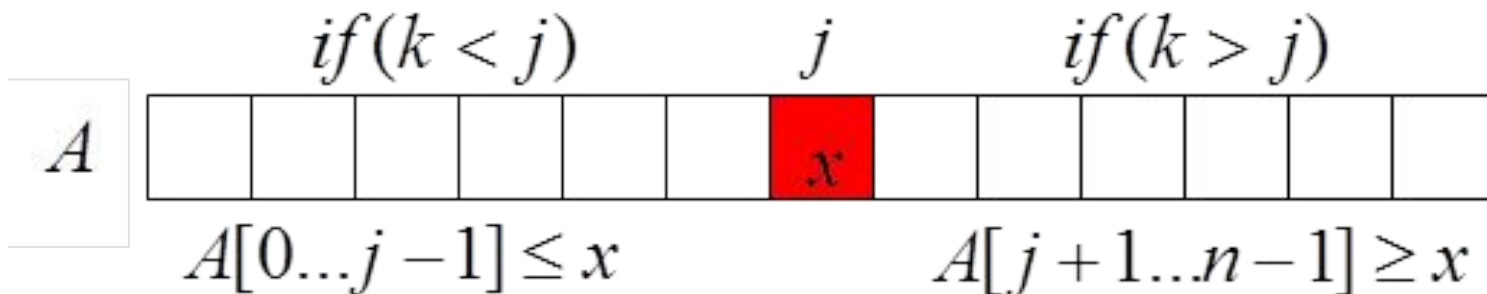
### Варианты решения:

1. Для  $k = 0|1|2|n - 3|n - 2|n - 1$  используются элементарные алгоритмы.
2. Если число запросов на поиск  $> \log n$ , то массив лучше отсортировать (прямо или косвенно).
3. Если  $k \leq \frac{n}{\log n}$  или  $k \geq n - \frac{n}{\log n}$ , то можно построить бинарную кучу и провести  $k$  шагов, как в пирамидальной сортировке.

# Поиск $k$ -го минимального элемента

## Идея для общего случая:

- разделение массива опорным элементом, как в быстрой сортировке (пусть опорный элемент  $x$  после разделения попадает на позицию  $j$ )
- рекурсивная или рекуррентная обработка той части массива, в которую попадает  $k$ -й элемент (возможны 3 варианта, в зависимости от того, какое из трех условий  $k < j$ ,  $k = j$ ,  $k > j$  выполняется).



# Алгоритм поиска $k$ -го минимального элемента

```
double med(double *A, int n, int k)
{ int b = 0, e = n-1; double x;
  while (b < e)
  {
    j = b; i = e; x = A[b];
    while (j < i)
      if (A[i] >= x) i--;
      else
        { A[j++] = A[i]; A[i] = A[j]; A[j] = x; }
    if (k < j) e = j-1;
    else if (k > j) b = j+1;
    else { b = j; break; }
  }
  return A[b];
}
```

# Цифровая сортировка

Пусть для **целочисленного массива  $A$**  выполняется:  
 $b \leq A_i \leq e$ ,  $i = \overline{0 \dots n-1}$ , где  $b$  и  $e$  – целые и  $e - b \ll n$ .

Тогда для сортировки массива достаточно сформировать  $e - b + 1$  счетчик (целочисленный массив  **$\text{count}[0 \dots e-b]$** ), подсчитать частоты всех значений и записать в  $A$  все группы одинаковых значений по возрастанию.

При этом  $\forall i$ ,  $0 \leq i \leq e - b$ , счетчик  **$\text{count}[i]$**  будет задавать общее число значений, равных  $b + i$  (или наоборот, значение некоторого элемента  **$A[j]$**  соответствует счетчику  **$\text{count}[A[j]-b]$** ).

# Простейший алгоритм цифровой сортировки

```
void rad_sort(int *A, int n, int b, int e)
{
    int i, j, k, *count;
    count = new int[e-b+1];
    for (i = 0; i <= e-b; i++)
        count[i] = 0;
    for (i = 0; i < n; i++)
        count[A[i]-b]++;
    for (k = i = 0; i <= e-b; i++)
        for (j = 0; j < count[i]; j++)
            A[k++] = b + i;
    delete [] count;
}
```

Трудоемкость данного алгоритма  $T(n) = O(n)$ .

# Косвенная цифровая сортировка

Пусть при тех же условиях массив  $A$  нужно упорядочить косвенно, т.е. сформировать массив индексов в порядке возрастания элементов  $A$ .

В этом случае нам понадобятся 3 целочисленных массива:

- формируемый массив индексов  $Ind[0..n-1]$ ,
- массив счетчиков  $count[0..e-b]$ ,
- массив  $pos[0..e-b]$  текущих позиций в  $Ind$  индексов элементов массива  $A$  (индекс  $i$  очередного выбираемого элемента  $A[i]$  будет записан в  $Ind$  на позиции  $pos[A[i]-b]$ ).

# Алгоритм косвенной цифровой сортировки

```
int* ind_rad_sort(int *A, int n, int b, int e)
{
    int i, *count, *pos, *Ind;
    count = new int[e-b+1];
    pos = new int[e-b+1]; Ind = new int[n];
    for (i = 0; i <= e-b; i++) count[i] = 0;
    for (i = 0; i < n; i++) count[A[i]-b]++;
    for (pos[0] = 0, i = 1; i <= e-b; i++)
        pos[i] = pos[i-1] + count[i-1];
    for (i = 0; i < n; i++)
        Ind[pos[A[i]-b]++] = i;
    delete [] count; delete [] pos;
    return Ind;
}
```

Трудоёмкость данного алгоритма  $T(n) = O(n)$ .

# Косвенная цифровая сортировка со списками

Если использовать списки целых (класс `List` из раздела «Линейные списки»), то можно записать более элегантный алгоритм косвенной сортировки массива `A`.

В этом случае нам понадобятся:

- формируемый список индексов `LRes` – очередь индексов в порядке возрастания значений `A`,
- массив списков `LMas[0...e-b]` (индекс `i` очередного выбираемого элемента `A[i]` будет добавлен в конец списка `LMas[A[i]-b]`).

Выходной список (очередь) `LRes` формируется с помощью последовательного объединения списков `LMas`.



# Косвенная цифровая сортировка со списками

```
List lrad_sort(int *A, int n, int b, int e)
{
    int i; List LRes, *LMas;
    LMas = new List[e-b+1];
    for (i = 0; i < n; i++)
        LMas[A[i]-b].push_back(i);
    for (i = 0; i <= e-b; i++)
        LRes.join(LMas[i]);
    delete [] LMas;
    return LRes;
}
```

Трудоемкость данного алгоритма  $T(n) = O(n)$ .

# Цифровая сортировка целых чисел

Целые 4-байтовые числа можно делить на отдельные байты и сортировать по байтам (**косвенно**), начиная с младших ( $k = 0 \dots 3$ ). При сортировке по байту  $k$  необходимо:

- выбирать числа **в порядке очереди**, полученной в результате сортировки по предыдущему байту  $k - 1$  (начальная очередь при  $k = 0$  содержит последовательность индексов от 0 до  $n - 1$ )
- разносить индексы по 256 спискам в соответствии со значением  $k$ -го байта чисел
- **сцепить полученные списки в единую очередь** (она будет определять порядок выбора чисел при сортировке по байту  $k + 1$  или окончательный порядок при  $k = 3$ ).

# Цифровая сортировка целых чисел

Отметим, что положительные и отрицательные целые числа имеют **разную внутреннюю кодировку**, поэтому их нужно сортировать отдельно. Мы рассмотрим только неотрицательные числа.

**Цифровая сортировка является устойчивой**, поэтому если  $A_i = A_j$  по байту  $k$ , то упорядоченность по байту  $k - 1$  не нарушится.

Для получения значения  $k$ -го байта числа  $A_i$  необходимо сдвинуть битовое представление числа на  $k * 8$  бит вправо и поделить по модулю 256. Результат будет в младшем байте, остальные байты – нули.

# Цифровая сортировка неотрицательных целых

```
List radix_sort(unsigned *A, int n)
{
    int i, k, j, m;
    List LRes, *LMas = new List[256];
    for (i = 0; i < n; i++) LRes.push_back(i);
    for (k = 0; k < 4; k++)
    {
        for (i = 0; i < n; i++) {
            j = LRes.pop_front(); m = (A[j] >> (k*8)) % 256;
            LMas[m].push_back(j);
        }
        for (i = 0; i < 256; i++) LRes.join(LMas[i]);
    }
    delete [] LMas;
    return LRes;
}
```

Трудоёмкость данного алгоритма  $T(n) = O(n)$ .