Алгоритмы и анализ сложности

Эффективные алгоритмы сортировки

Задача сортировки элементов массива

Дан массив значений $A_0 A_1 A_2 \dots A_{n-2} A_{n-1}$.

Необходимо найти такую перестановку $i_0 i_1 i_2 \dots i_{n-2} i_{n-1}$ индексов , что для последовательности $A_{i_0} A_{i_1} A_{i_2} \dots A_{i_{n-2}} A_{i_{n-1}}$ выполняется $A_{i_k} \le A_{i_{k+1}} \ \forall\, k = \overline{0 \dots n-2}$.

$i_0 i_1 i_2 \dots i_{n-2} i_{n-1}$ - 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 $\lceil k/2 \rceil$ серий длины $2s$ в рабочий массив *D*
- новые серии копируются из *D* в *A*.

Проходы повторяются при $s = 1, 2, 4, \ldots,$ 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):

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

$0,\ h,\ 2h,\ 3h, \ldots$

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

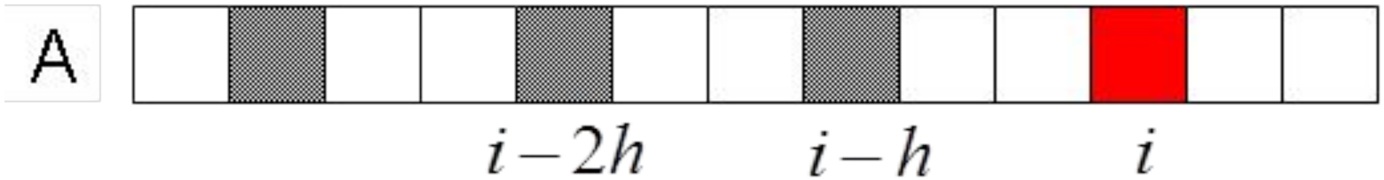
$\ldots$

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

Всего будет h цепочек длиной $\leq [n/h]$.

Упорядочим элементы в цепочках, используя сортировку обменом (вставками).

Сортировка вставкам с шагом *h*

- 

Сортировка одной цепочки: $T_{worst}(n) = 0(\left(\frac{n}{h}\right)^2)$,

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

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: $h_1 > h_2 > \cdots > h_t = 1$. 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 $T_{worst}(n) = O(n^2), \quad T_{mid}(n) = O(n\sqrt{n})$.
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 ≥ 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 ≤ 2 вершин-сыновей
- заполнены все уровни, кроме, возможно, последнего
- **значение-отец всегда не меньше значений-сыновей.**

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

```
        9  <= 2                  8                    8
      /     \                 /      \             /     \
   8           6            2           6         5          6
  / \         / \    =>    / \         / \   =>  / \        / \
 5    3     1    4        5    3     1    4     2    3     1    4
/ \  / \   /             / \  / \   /          / \  / \   /
0  2  3  1  0            0  2  3  1  0         0  2  3  1  0
```

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

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

3. Если $2^{k-1} \leq n < 2^k$, то k-й уровень заполнен не до конца.

4. 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: $\forall i,\ 0 \leq i \leq n/2\ \ A_i \geq A_{2i+1},\ A_i \geq A_{2i+2}$

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.

Трудоемкость 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.

Трудоемкость 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.

Трудоемкость построения пирамиды

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

$$\frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + 4\frac{n}{32} \ldots = \frac{n}{4}\left(1 + 1 + \frac{3}{4} + \frac{4}{8} + \frac{5}{16} \ldots\right) =$$

$$= \frac{n}{4}\left[\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right) + \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right) + \left(\frac{1}{4} + \frac{1}{8} + \ldots\right) + \ldots\right] =$$

$$= \frac{n}{4}\left(2 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right) = n$$

Следовательно, трудоемкость 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;
  }
}
```

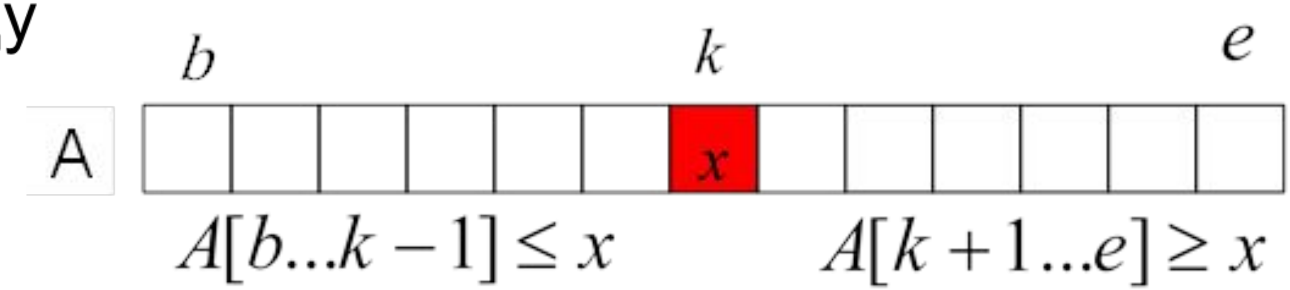
Алгоритм пирамидальной сортировки

```
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 $x \in A[b \ldots e]$ и текущая часть приводится к виду



Быстрая сортировка

- После разделения элемент x окажется на своем месте k 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 $k = \overline{0 \ldots n-1}$
- 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:

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

Трудоемкость в среднем: доказательство

- Положим $T(0) = T(1) = b = const$ и $d = 2(b + c)$.

Покажем, что $T(n) \le d\,n \ln n\,,\ \forall\, n \ge 2$.

Доказательство (мат. индукция):

1. Базис $n = 2$: $T(n) = 2c + 2b = d < d \cdot 2 \ln 2 \approx 1.4d$
2. Пусть $\forall\, k = \overline{2 \ldots n - 1}$ выполняется $T(k) \le dk \ln k$,

   тогда $T(n) = cn + \dfrac{2}{n} \sum\limits_{k=0}^{n-1} T(k) \le cn + \dfrac{4b}{n} + \dfrac{2d}{n} \sum\limits_{k=2}^{n-1} k \ln k$

Трудоемкость в среднем: доказательство

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

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

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

Трудоемкость в среднем: доказательство

Таким образом:

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

$$\le cn + \frac{4b}{n} + \frac{2d}{n}\left(\frac{n^2 \ln n}{2} - 2\ln 2 - \frac{n^2}{4} + 1\right) =$$

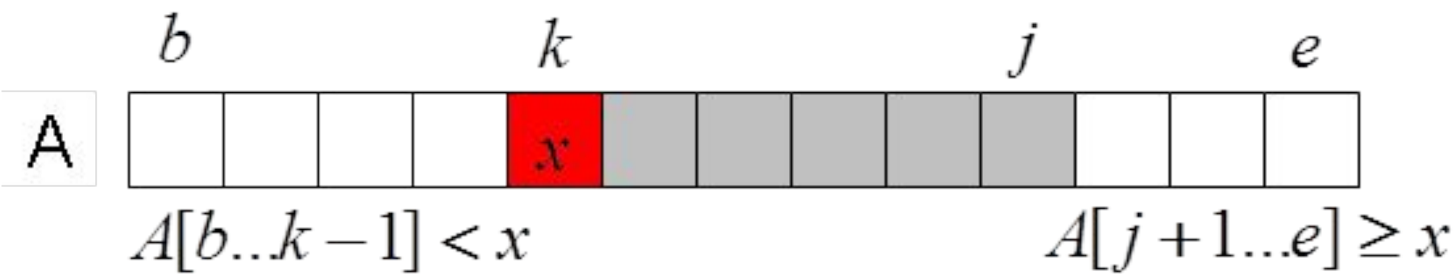
$$= cn + \frac{4b}{n} + dn\ln n - (b+c)n - \frac{2d(2\ln 2 - 1)}{n} < dn\ln n = O(n\log n)$$

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 $i = b,\ j = e$).

$A[b..i-1] \le x$ $\qquad$ $A[j+1..e] \ge x$

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 $i \le j$, 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);
}
```

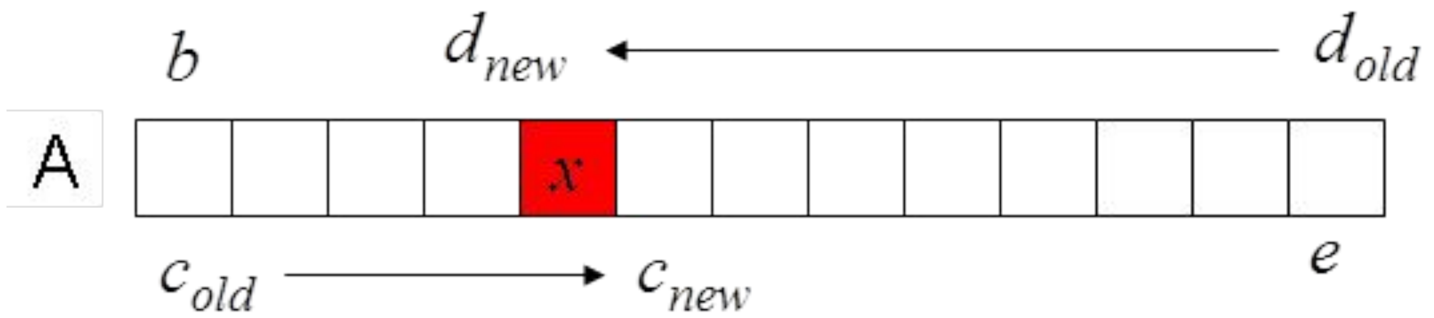
Быстрая сортировка с 1 рекурсивным вызовом

- 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 $A[b+1 \ldots e]$, либо $A[b \ldots e-1]$.
- 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 большей 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).

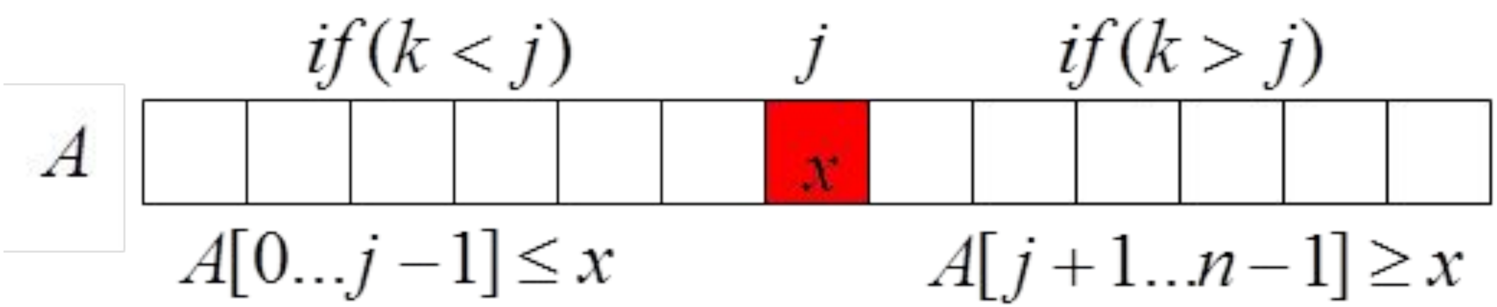
Варианты решения:

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];
}
```