РК2 по Алгоритмам

Ссылочки))

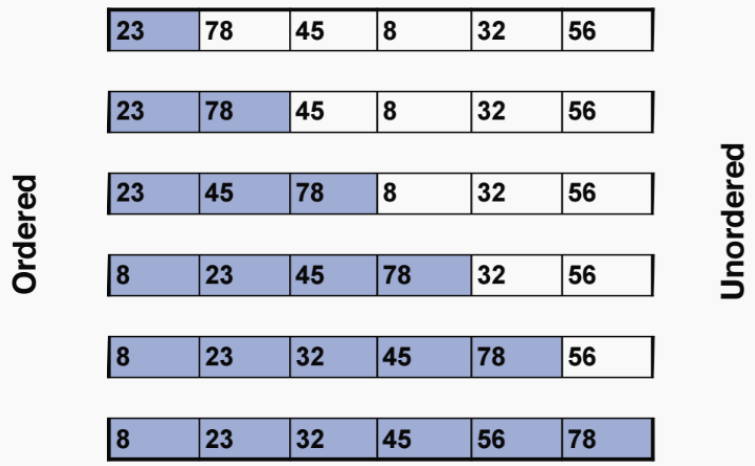
Powered by Говязин и Набережный, 2016

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$ variation is n	$n \log n$	n^2	$\log n$ on average, worst case space complexity is n; Sedgewick variation is $\log n$ worst case.	Typical in-place sort is not stable; stable versions exist.	Partitioning	Quicksort is usually done in-place with O(log *n*) stack space.[2][3]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n A hybrid block merge sort is O(1) mem.	Yes	Merging	Highly parallelizable (up to O(log *n*) using the Three Hungarians' Algorithm[4] or, more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$ See above, for hybrid, that is $n \log n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[5]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(*n* + *d*), in the worst case over sequences that have *d* inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists.[6]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes *n* comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes *n* comparisons when the data is already sorted or reverse sorted.
Shell sort	$n \log n$	$n \log^2 n$ or $n^{5/4}$	Depends on gap sequence; best known is $n \log^2 n$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications. Best case n log n and worst case n log² n cannot be achieved together. With best case n log n, best worst case is $n^4/^3$.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	n	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in O(*n* log *n*).
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[7]	No	Selection	Variation of Heap Sort.
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
UnShuffle Sort[8]	n	kn	kn	In-place for linked lists. n × sizeof(link) for array.	No	Distribution and Merge	No exchanges are performed. The parameter *k* is proportional to the entropy in the input. *k* = *1* for ordered or reverse ordered input.
Franceschini's method[9]	—	$n \log n$	$n \log n$	1	Yes	?	
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based O(n) in-place merge algorithm[10] with a bottom-up merge sort.
Odd-even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

1) Сортировка вставками

Сортировка вставками

- Сортировка вставками – простой алгоритм часто применяемый на малых объёмах данных
- Самый популярный метод сортировки у игроков в покер
- Массив делится на две части, упорядоченную - левую и неупорядоченную - правую
- На каждой итерации выбираем элемент из правой части и вставляем его на подходящее место в левой части
- Массив из ***n*** элементов требует ***n-1*** итерацию

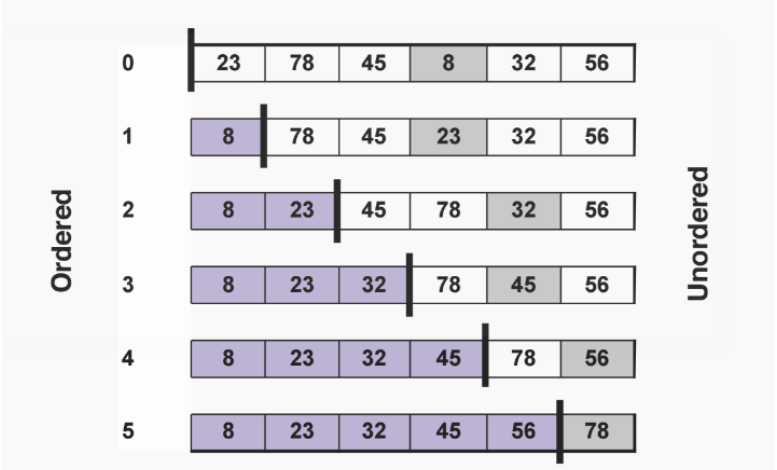



```
void insertion_sort(int *a, int n) {
  for (int i = 1; i < n; ++i) {
    int tmp = a[i];
    for (int j = i; j > 0 && tmp < a[j-1]; --j) {
      a[j] = a[j-1];
    }
    a[j] = tmp;
   }
}
```

2) Сортировка выбором

Сортировка выбором

- Разделим массив на две части: левую - упорядоченную и правую – неупорядоченную
- Будем гарантировать, что элементы в правой части больше чем в левой
- Выберем наименьший элемент в правой части и переставим его в её начало
- После каждого выбора и перестановки, будем смещать границу между частями массива на *1* вправо
- Выбор каждого нового элемента требует прохода по правой части
- Для сортировки массива из *N* элементов требуется *N-1* проход

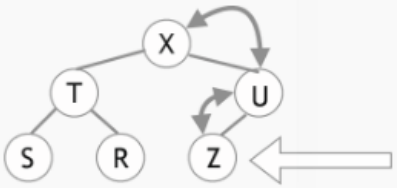


Сортировка выбором

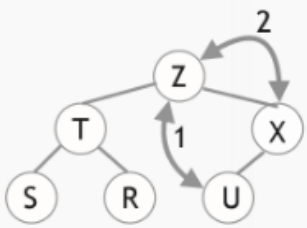
```
void selection_sort(int *a, int n) {
  for (int i = 0; i < n - 1; ++i) {
    int min_index = i;
    for (int j = i + 1; j < n; ++j) {
       if (a[j] < a[min_index]) min_index = j;
    }
    swap(a[i], a[min_index]);
  }
}

void swap(int &a, int &b ) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

3) Двоичная куча. Методы вставки и извлечения максимума

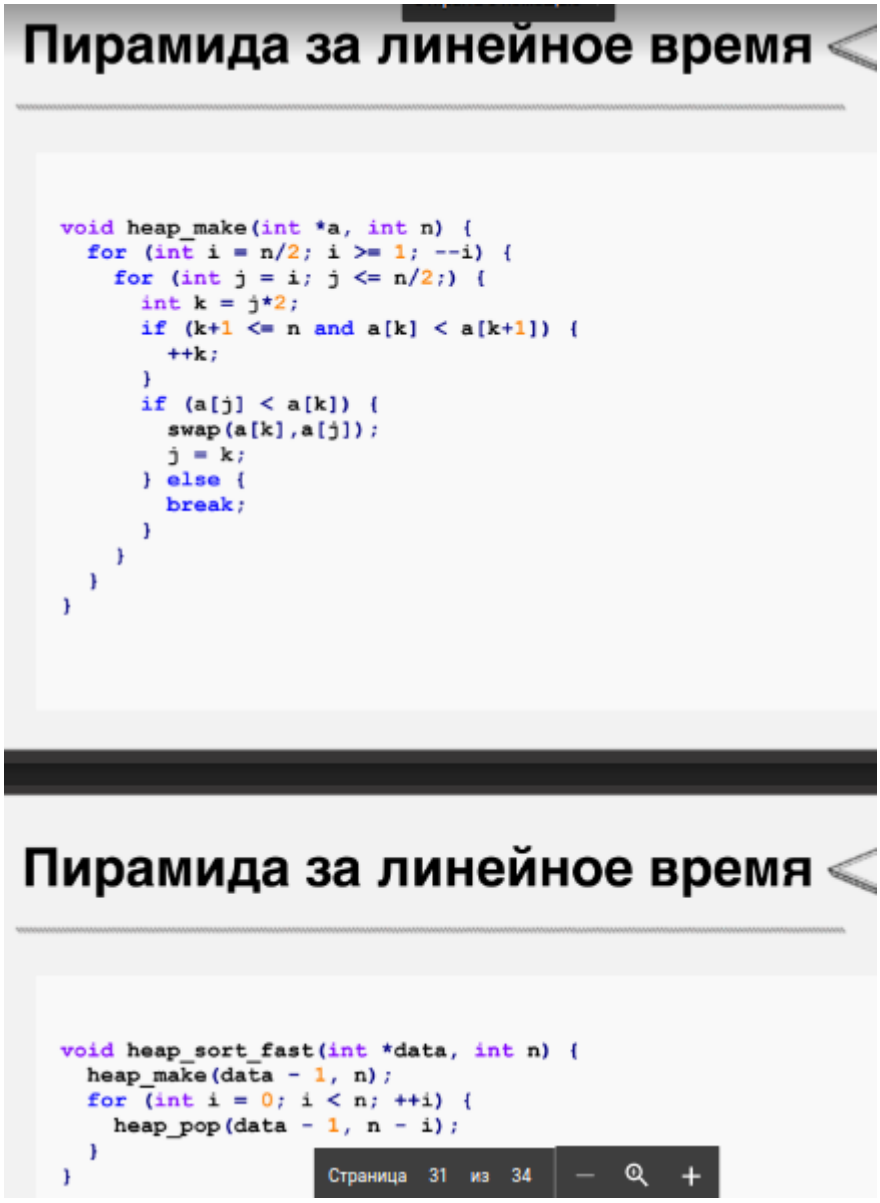
```
void heap_insert(int *a, int n, int x)
{
  a[n+1] = x;
  for (int i = n+1; i > 1;) {
    if (a[i] > a[i/2]) {
      swap(a[i], a[i/2]);
      i = i/2;
    } else {
      break;
    }
  }
}
```




```
void heap_pop(int *a, int n) {
  swap(a[n],a[1]);

  for (int i = 1; 2*i < n;) {
    i *= 2;
    if (i+1 < n && a[i] < a[i+1]) {
      i += 1;
    }
    if (a[i/2] < a[i]) {
      swap(a[i/2], a[i]);
    }
  }
}
```



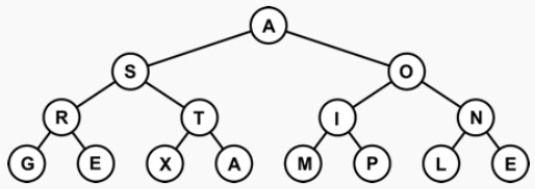
4) Построение кучи за линейное время



5) Пирамидальная сортировка

```
void heap_sort(int *data, int n) {
  int *buff = new int[n+1];
  for (int i = 0; i < n; ++i) {
    heap_insert(buff, i, data[i]);
  }
  for (int i = 0; i < n; ++i) {
    data[n-1-i] = buff[1];
    heap_pop(buff, n - i);
  }
  delete [] buff;
}
```

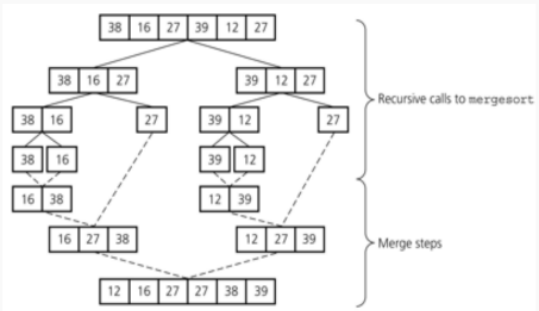
N вставок в кучу: ***N · O(log(N))***
N Извлечение минимума из кучи: ***N · O(log(n))***
Построение кучи из ***N*** элементов: ***O(N·log(N))***



6) Сортировка слиянием

Нисходящая сортировка слиянием

- Разбить массив на 2 части
- Отсортировать каждую часть рекурсивно
- Объединить 1 и 2 части



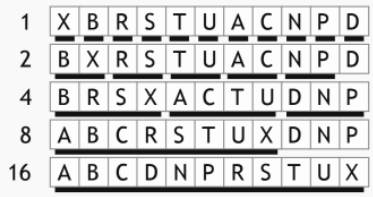
Нисходящая сортировка слиянием

```
void merge_sort(int *data, int size, int *buffer) {
  if (size < 2) return;
  merge_sort(data, size / 2, buffer);
  merge_sort(&data[size / 2], size - size / 2, buffer);

  merge(&data[0], size / 2, &data[size/2], size - size / 2, buffer);

  for (size_t pos = 0; pos < size; ++ pos) {
    data[pos] = buffer[pos];
  }
}
```

```
void merge(int *a, int a_len, int *b, int b_len, int *c) {
  int i=0; int j=0;
  for (;i < a_len and j < b_len;) {
    if (a[i] < b[j]) {
      c[i+j] = a[i];
      ++i;
    } else {
      c[i+j] = b[j];
      ++j;
    }
  }
  if (i==a_len) {
    for (;j < b_len;++j) { c[i+j] = b[j]; }
  } else {
    for (;i < a_len;++i) { c[i+j] = a[i]; }
  }
}
```

Восходящая сортировка слиянием

- Разбить массив на 2^k частей размером не больше m.
- Отсортировать каждую часть другим алгоритмом
- Объединить 1 и 2, 3 и 4, … n-1 и n части.
- Повторить шаг 3, пока не останется одна часть



Восходящая сортировка слиянием

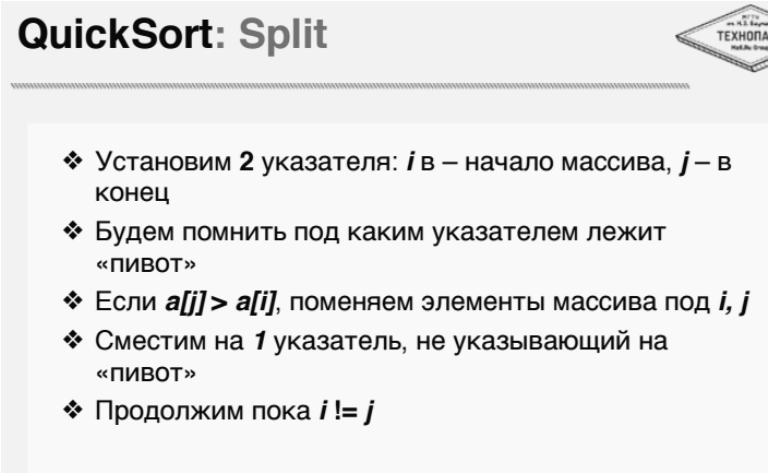
```
void merge_sort(int *data, size_t size, int *buffer) {
  for(size_t chunk_size = 1; chunk_size < size; chunk_size *= 2) {
    size_t offset = 0;
    for (; offset + chunk_size < size; offset += 2 * chunk_size) {
      size_t right_size = chunk_size;
      if (offset + chunk_size + right_size > size) {
        right_size = size - offset - chunk_size;
      }
      merge(
        &data[offset], chunk_size,
        &data[offset + chunk_size], right_size,
        &buffer[offset]);
    }
    for(size_t pos = 0; pos < size; ++pos) {
      data[pos] = buffer[pos];
    }
  }
}
```

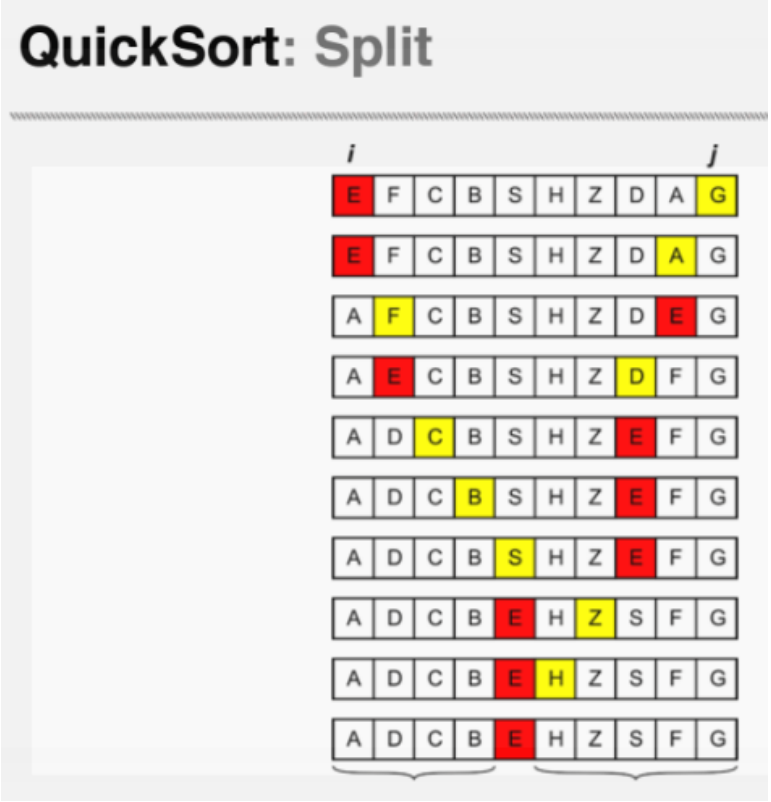
Восходящая сортировка слиянием

```
void merge_sort_fast(int *data, size_t size, int *buffer) {
  bool is_swapped = false;
  for(size_t chunk_size = 1; chunk_size < size; chunk_size *= 2, is_swapped = !is_swapped) {
    size_t offset = 0;
    for (; offset + chunk_size < size; offset += 2 * chunk_size) {
      size_t right_size = chunk_size;
      if (offset + chunk_size + right_size > size) {
        right_size = size - offset - chunk_size;
      }
      merge(
        &data[offset], chunk_size,
        &data[offset + chunk_size], right_size,
        &buffer[offset]);
    }
    for (size_t pos = offset; pos < size; ++pos) {
      buffer[pos] = data[pos];
    }
    std::swap(data, buffer);
  }
  if (is_swapped) {
    std::swap(data, buffer);
    for(size_t pos = 0; pos < size; ++pos) {
      data[pos] = buffer[pos];
    }
  }
}
```

7) Быстрая сортировка (Сортировка Хоара)

В массиве выбирается случайный элемент x, и выполняется просмотр массива слева, пока не найдётся элемент a[i]>x, затем выполняется просмотр справа, пока не будет найден элемент a[j]<x. Как только два таких элемента найдены, выполняется их обмен, и просмотр продолжается до тех пор, пока индексы i, j не станут равны где-то в середине массива. В результате получается массив, левая часть которого содержит элементы <=x, а правая часть содержит элементы >=x. Описанная процедура применяется рекурсивно для левой и правой части и продолжается до тех пор, пока не будет получен полностью отсортированный массив.





Сортировка Хоара

```
void quick_sort(int *a, int n) {
  int i = 0;
  int j = n - 1;
  bool side = 0;
  while (i != j) {
    if (a[i] > a[j]) {
      swap(a[i], a[j]);
      side = !side;
    }
    if (side) {
      ++i;
    } else{
      --j;
    }
  }
  if (i > 1) quick_sort(a, i);
  if (n > i+1) quick_sort(a + (i+1), n - (i+1));
}
```

Quicksort: выбор пивота

1й элемент

Серединный элемент

Медиана трёх

Случайный элемент

Медиана

Медиана по трём случайным

...

8) Алгоритм поиска k-ой порядковой статистики

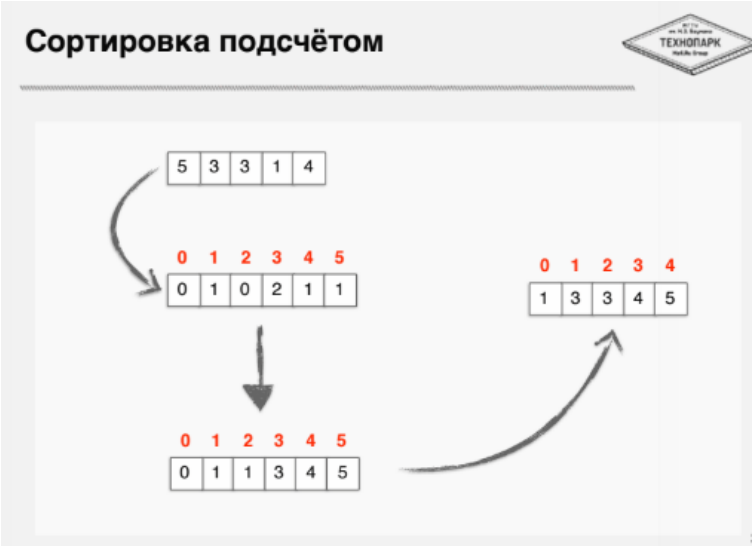
k-ой порядковой статистикой набора элементов линейно упорядоченного множества называется такой его элемент, который является k-ым элементом набора в порядке сортировки.

ВНИМАНИЕ!!! ТУПО КОПИПАСТА!!!

k-ой порядковой статистикой массива называется значение, которое будет стоять на k-ом месте после сортировки.

Тривиальный алгоритм, основанный на сортировке и выводе k-го элемента, работает за O(nlog(n)). Однако существует метод, использующий модифицированный qsort, чтобы найти k-ую порядковую статистку в среднем за O(n).
Алгоритм нахождения k-го элемента состоит в выборе опорного элемента x и разделении массива на три части аналогично функции qsort. Далее, в зависимости от того, в каком из блоков находится k-ый элемент, нужно продолжать поиск именно на этом блоке, игнорируя остальные. Заметим, что если k-ый попал в блок элементов, равных x, то уже можно завершить выполнение алгоритма и в качестве ответа вывести x.
В среднем, на i-ом по глубине вызове функции нахождения порядковой статистики, проверяется в среднем **n/(2^i)** элементов, где n — размерность исходного массива. А так как $\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n,$, то количество действий в среднем в алгоритме не превосходит 2n. Следовательно, в среднем алгоритм работает за O(n).

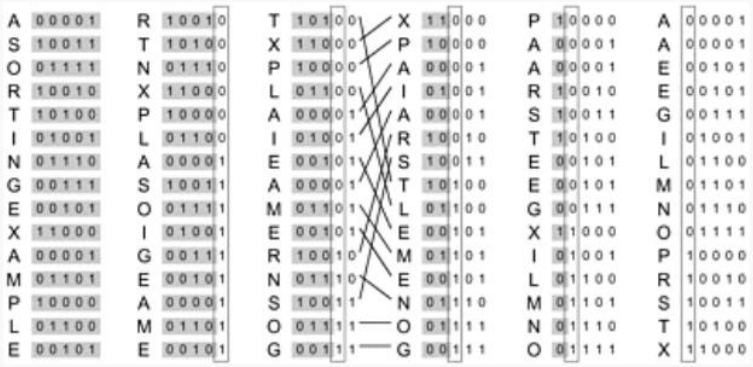
9) Сортировка подсчетом




```
void count_sort(int *data, int size, int range) {
  int *count = new int[range];
  int *aux = new int[size];

  for (int i = 1; i < range; ++i) {
    count[i] = 0;
  }
  for (int i = 0; i < size; ++i) {
    ++count[data[i] + 1];
  }
  for (int i = 1; i < range; ++i) {
    count[i] += count[i - 1];
  }
  for (int i = 0; i < size; ++i) {
    aux[count[data[i]]++] = data[i];
  }
  for (int i = 0; i < size; i++) {
    data[i] = aux[i];
  }

  delete [] aux;
  delete [] count;
}
```

10) Поразрядные сортировки (LSD и MSD)

LSD raddix sort



LSD raddix sort

```
 3
 4 void jsw_radix_pass ( int a[], int aux[], int n, int radix )
 5 {
 6   int i;
 7   int count[RANGE] = {0};
 8
 9   for ( i = 0; i < n; i++ )
10     ++count[digit ( a[i], radix ) + 1];
11
12   for ( i = 1; i < RANGE; i++ )
13     count[i] += count[i - 1];
14
15   for ( i = 0; i < n; i++ )
16     aux[count[digit ( a[i], radix )]++] = a[i];
17
18   for ( i = 0; i < n; i++ )
19     a[i] = aux[i];
20 }
21
```

LSD raddix sort

+ O(n*key len) время работы
+ O(n) время работы
+ stable
- can’t use unstable sort for buckets

MSD raddix sort

MSD raddix sort

+ O(n*key_len) время работы*
+ O(n) памяти
+ нет сортировки частей размером 1
+ could be unstable
- рекурсивная реализация

11) Частичная сортировка, поиск k -максимальных элементов

Частичная сортировка

Ранее мы изучали проблему полного упорядочения *множества* имен, не имея *a priori* информации об абстрактном порядке имен. Имеется два очевидных уточнения этой проблемы. Вместо полного упорядочения требуется только определить k -е наибольшее имя (то есть k -е имя в порядке убывания) (выбор) или, вместо того чтобы начинать процесс, не располагая информацией о порядке, начинать с двух отсортированных *подтаблиц* (слияние). Мы рассматриваем обе проблемы частичной сортировки.

Частичная сортировка (выбор)

Как при данных именах $x_1, x_2, \ldots, x_n$ можно найти k -е из наибольших в порядке убывания? Задача, очевидно, симметрична: отыскание $(n-k+1)$ -го наибольшего (k -го наименьшего) имени можно осуществить, используя *алгоритм* отыскания k -го наибольшего, но меняя местами действия, предпринимаемые при результатах < и >сравнения имен. Таким образом, отыскание наибольшего имени $(k=1)$ эквивалентно отысканию наименьшего имени $(k=n)$; отыскание второго наибольшего имени $(k=2)$ эквивалентно отысканию второго наименьшего $(k=n-1)$ и т.д.

Конечно, все перечисленные варианты задачи выбора можно решить, используя любой из методов полной сортировки имен и затем тривиально обращаясь к k -му наибольшему. Такой подход потребует порядка $n \log n$ сравнений имен независимо от значений k .

При использовании алгоритма сортировки для выбора наиболее подходящим будет один из алгоритмов, основанных на выборе: либо простая *сортировка* выбором (*алгоритм* 15.1) либо *пирамидальная сортировка* (*алгоритм* 15.3). В каждом случае мы можем остановиться после того, как выполнены первые k шагов. Для простой *сортировки выбором* это означает использование

$$(n-1)+(n-2)+\ldots+(n-k)=kn-\frac{k(k+1)}{2}$$

сравнений имен, а для пирамидальной — использование $n+k\lg n$ сравнений имен. В обоих случаях мы получаем больше информации, чем нужно, потому что мы полностью определяем порядок k наибольших имен.

Частичная сортировка (слияние)

Вторым направлением исследования частичной сортировки является задача слияния двух отсортированных таблиц $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$ и $y_1 \leqslant y_2 \ldots \leqslant y_m$ в одну отсортированную таблицу $z_1 \leqslant z_2 \leqslant \ldots \leqslant z_{n+m}$. Существует очевидный способ это сделать: таблицы, подлежащие слиянию, просматривать параллельно, выбирая на каждом шаге меньшее из двух имен и помещая его в окончательную таблицу. Этот процесс немного упрощается добавлением имен-сторожей $x_{n+1}=y_{m+1}=\infty$, как в алгоритме 15.5. В этом алгоритме i и j указывают, соответственно, на последние имена в двух входных таблицах, которые еще не были помещены в окончательную таблицу.

$x_{n+1} \leftarrow y_{m+1} \leftarrow \infty$
$i \leftarrow j \leftarrow 1$

$$\textit{for } k=1 \textit{ to } n+m \textit{ do} \begin{cases} \textit{if } x_i < y_j \textit{ then} \begin{cases} z_k \leftarrow x_i \\ i \leftarrow i+1 \end{cases} \\ \textit{else} \begin{cases} z_k \leftarrow y_j \\ j \leftarrow j+1 \end{cases} \end{cases}$$

ТУПО КОПИПАСТА!! ВНИМАНИЕ!! Хотя, она с хабра...

Процедура разделения, используемая в быстрой сортировке, даёт потенциальную возможность находить искомый (k-ый) элемент гораздо быстрее.

Этот алгоритм работает следующим образом. На первом шаге вызывается процедура разделения с L=1 и R=N (т.е. разделение

выполняется для всего массива), причём в качестве разделяющего значения x выбирается a[k]. После разделения получаются значения индексов i, j такие, что

a[h]<x для всех h<i
a[h]>x для всех h>j
i>j

Здесь возможны три случая:

•Разделяющее значение x оказалось слишком мало. В результате граница между двумя частями меньше нужного значения k. Тогда операцию разделения нужно повторить с элементами a[i]…a[R].

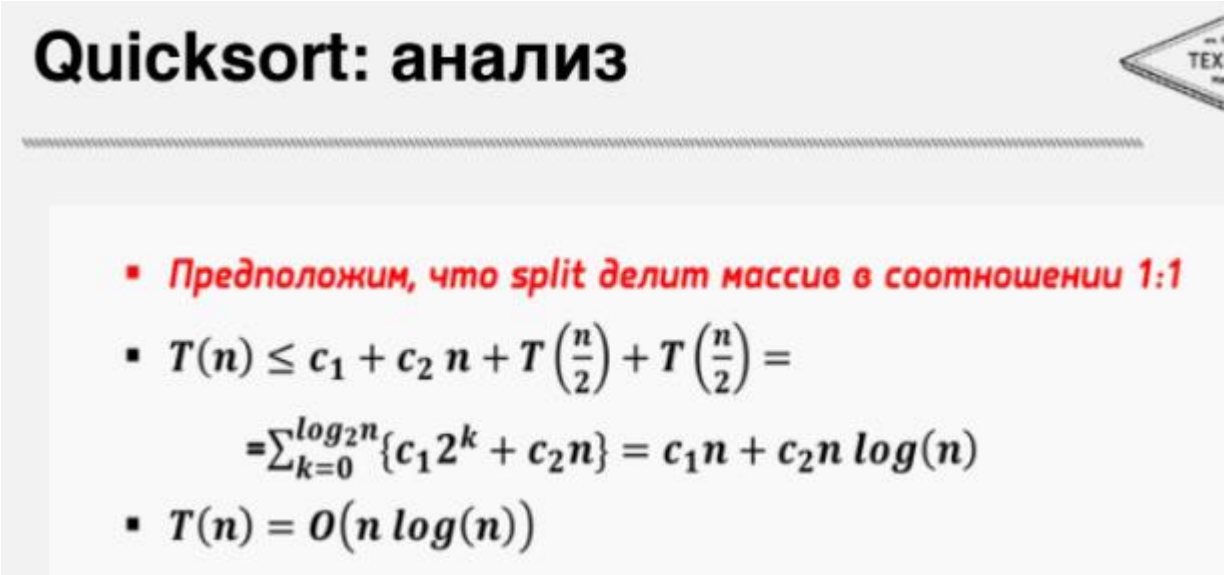
•Выбранное значение x оказалось слишком велико. Тогда операцию разделения нужно повторить с элементами a[L]…a[j].

•Элемент a[k] разбивает массив на две части в нужной пропорции и поэтому является искомым значением.

Операцию разделения нужно повторять, пока не реализуется случай 3.

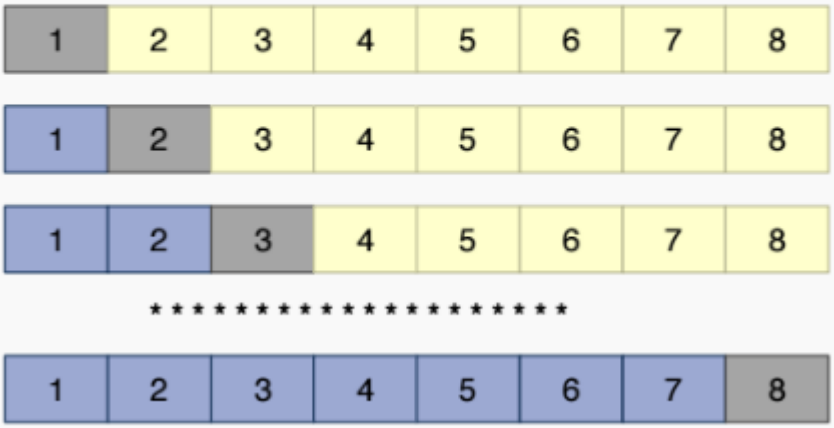
Если предположить, что в среднем каждое разбиение делит пополам размер части массива, в которой находится искомое значение, то необходимое число сравнений будет N+N/2+N/4+…+1=2N. Это объясняет эффективность приведённой процедуры для поиска медиан и прочих величин, а также объясняет её превосходство над простым методом, состоящем в предварительной сортировке всего массива с последующим выбором k-ого элемента (где наилучшее поведение имеет порядок N*log(N)).

12) Быстрая сортировка со сложностью O(n log n) в худшем случае



13) Нижний предел сложности сортировки

- *Упорядоченный массив делится в соотношении 1:n-1*
- $T(n) = c_1 + c_2 n + T(n-1) = \frac{1}{2} c_2 n^2 + c_1 n = O(n^2)$



14) Стабильность алгоритмов сортировки

Во всех вышеприведённых алгоритмах при разбиении исходного массива на части рекурсивное разбиение можно остановить, если размер разбиваемого массива станет достаточно маленьким. Тогда можно применить какой-либо из простых алгоритмов сортировки (например, сортировка вставками), которые, как известно, работают быстрее, чем сложные, если размер входного массива невелик. Фактически данный приём применим не только для представленных здесь алгоритмов, но и для любого другого алгоритма, где применяется рекурсивное разбиение исходного массива (например, обычная нестабильная быстрая сортировка). Конкретное число входных элементов, при котором надо переходить на простой алгоритм сортировки, зависит от используемой вычислительной машины.

15) Медиана за линейное время

Quicksort: медиана за линейное время

Медиана ⇔ ***SELECT(A[1,N], k)***; ***k = N/2***

Разобьём массив на пятёрки

Отсортируем каждую пятёрку

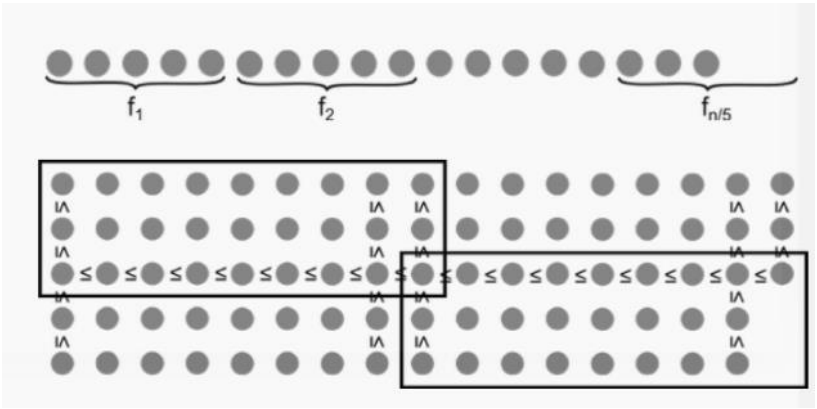
Найдём медиану середин пятёрок

Разобьём массив на 2 группы: меньше/больше медианы пятёрок

Пусть индекс медианы в массиве ***j***

j > k ?

- найдём: ***SECLECT(A[1, j], k)***
- иначе: ***SELECT(A[j+1, N], k-j)***



- Разобьём массив на пятёрки
- Отсортируем каждую пятёрку $c_1 N$
- Найдём медиану середин пятёрок $T(N/5)$
- Разобьём массив на 2 группы <> медианы медиан: $c_2 N$
  - найдём: ***SECLECT(A[1, j], k)*** => $T(j)$
  - иначе: ***SELECT(A[j+1, N], k-j)*** => $T(N-j)$; $0.3N \le j \le 0.7N$;
- $T(N) \le T\left(\frac{N}{5}\right) + cN + T(0.7N)$; => $T(N) = O(N)$;

16) Многопутевое слияние (k-путевое)

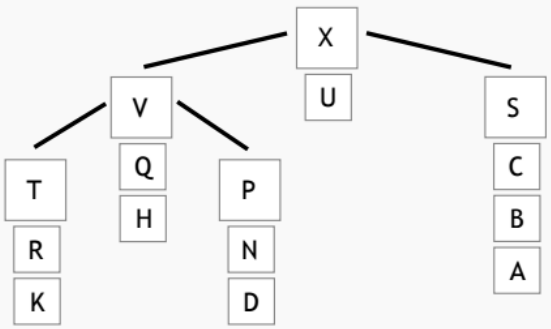
K-путевое слияние

Дано *k* упорядоченных массивов суммарным размером *n* : A1, A2, .. Ak

- Построить кучу из *k* массивов *O(k)*
- Перенести первый элемент из вершины кучи в результат *O(1)*
- Если массив на вершине кучи пуст – извлечь элемент из кучи
- Восстановить порядок массивов в куче *O(log(k))*
- Повторить пока куча не пуста

Суммарная сложность: *O(k + n*log(k))*

K-путевое слияние



K-путевое **слияние**

- Дано *k* упорядоченных массивов суммарным размером *n* : A1, A2, .. Ak
- Построим бинарное дерево с массивами A1..Ak в листьях
- В узловых вершинах будем хранить указатель на минимальный элемент поддерева
- Изъятие элемента из узловой вершины – изъятие из минимального из дочерних узлов
- При изъятии элемента из списка, его размер уменьшается на 1
- Если список пуст – его сосед перемещается на место родительской узловой вершины
- При изъятии происходит одно сравнение на каждом уровне дерева

Высота дерева *log(k)*

Итого *O(n*log(k))*

K-путевое **слияние**

Экономия на операциях копирования: *n*

Количество сравнений такое же как и при log(k) 2х путевых слияниях