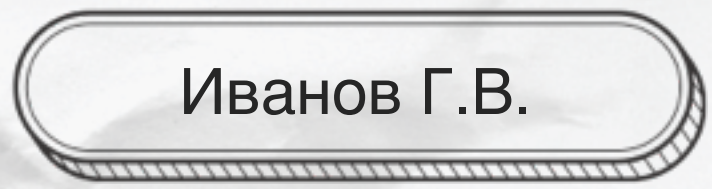



# АЛГОРИТМЫ СОРТИРОВКИ ЧАСТЬ 2

Лекция 4

A rounded rectangular box with a 3D effect, containing the author's name.

Иванов Г.В.

# Сортировки 2

---

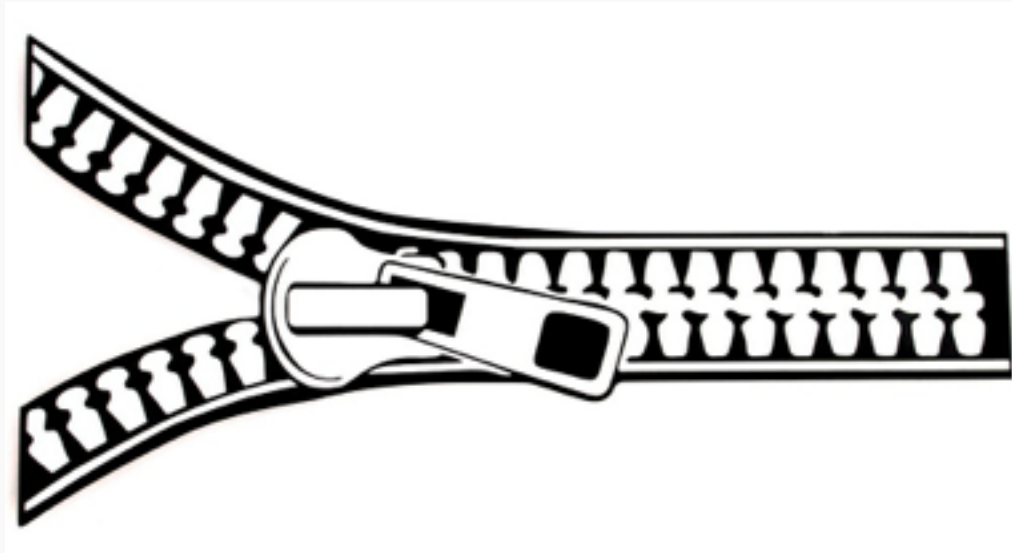


1. Сортировка слиянием
2. Быстрая сортировка
3. Порядковые статистики
4. Сортировки без сравнений

# Слияние 2х упорядоченных массивов



- ❖ Выберем массив, крайний элемент которого меньше
- ❖ Извлечём этот элемент в массив-результат
- ❖ Продолжаем, пока один из массивов не опустеет
- ❖ Копируем остаток второго массива в конец массива-результата



# Слияние 2х упорядоченных массивов



```
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]; }
    }
}
```

# Слияние 2х упорядоченных массивов

---



Сложность  $O(n+m)$

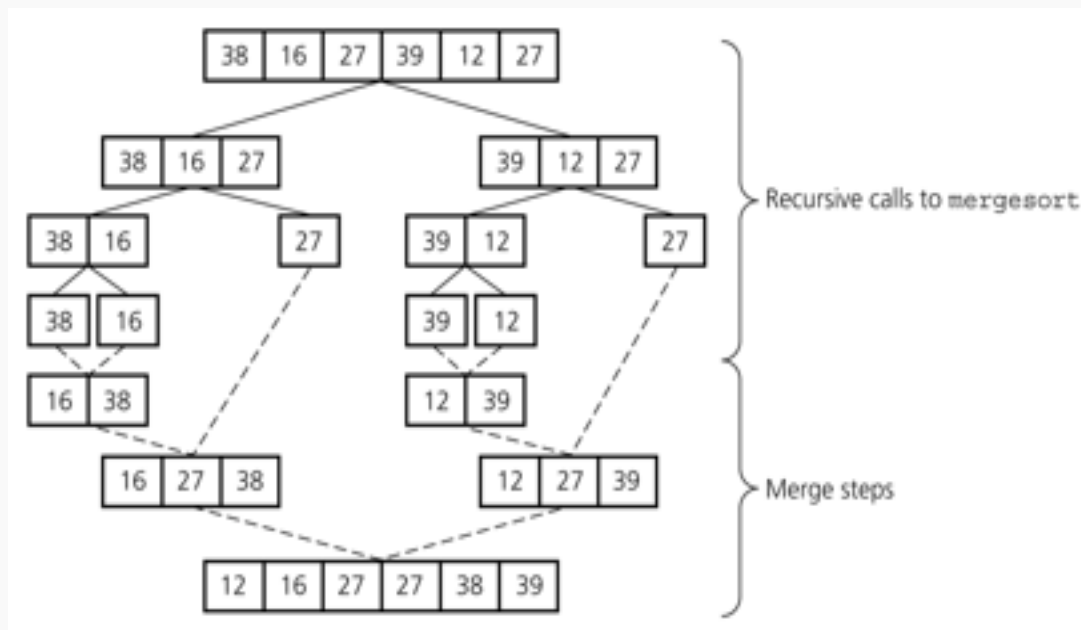
Количество сравнений

- Лучшее:  $\min(n, m)$
- Худшее  $n+m$

# 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];
    }
}
```

# 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

1	X	B	R	S	T	U	A	C	N	P	D
2	B	X	R	S	T	U	A	C	N	P	D
4	B	R	S	X	A	C	T	U	D	N	P
8	A	B	C	R	S	T	U	X	D	N	P
16	A	B	C	D	N	P	R	S	T	U	X



# 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



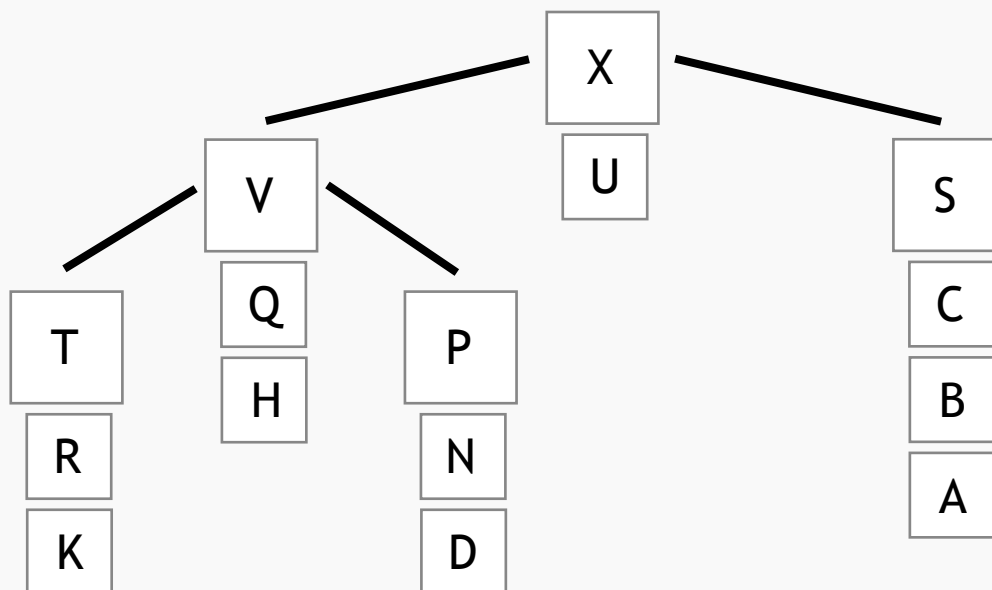
```
void merge_sort_fast(int *data, size_t size, int *buffer) {
    bool is_swapped = false;
    for(size_t chunk_size = 1; chunk_size < size; chunk_size *= 2, is_swapped = !is_swapped) {
        size_t offset = 0;
        for (; offset + chunk_size < size; offset += 2 * chunk_size) {
            size_t 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];
        }
    }
}
```

Дано  $k$  упорядоченных массивов суммарным размером  $n$  :  $A_1, A_2, \dots, A_k$

- ❖ 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:  $O(k + n \cdot \log(k))$

# К-путевое слияние



- ❖ Дано  $k$  упорядоченных массивов суммарным размером  $n$  :  $A_1, A_2, \dots, A_k$
- ❖ Построим бинарное дерево с массивами  $A_1..A_k$  в листьях
- ❖ В узловых вершинах будем хранить указатель на минимальный элемент поддеревы
- ❖ 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  $\log(k)$

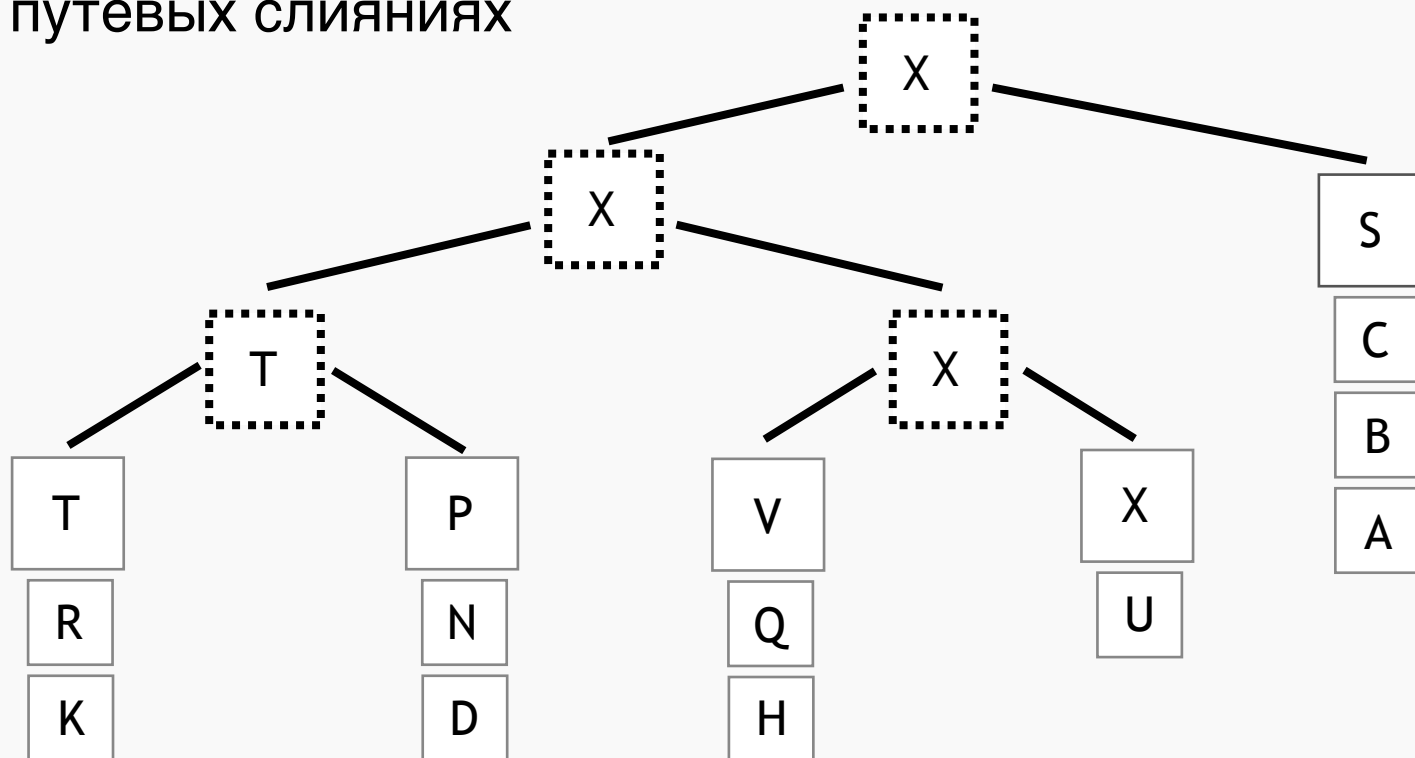
Итого  $O(n \cdot \log(k))$

# К-путевое слияние



Экономия на операциях копирования:  $n$

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



# Quick sort

---



# QuickSort: Split



- ❖ Установим **2** указателя:  $i$  в – начало массива,  $j$  – в конец
- ❖ Будем помнить под каким указателем лежит «ПИВОТ»
- ❖ Если  $a[j] > a[i]$ , поменяем элементы массива под  $i, j$
- ❖ Сместим на **1** указатель, не указывающий на «ПИВОТ»
- ❖ Продолжим пока  $i \neq j$



# QuickSort: Split



# 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: анализ



- *Предположим, что split делит массив в соотношении 1:1*

- $$T(n) \leq c_1 + c_2 n + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) =$$
$$= \sum_{k=0}^{\log_2 n} \{c_1 2^k + c_2 n\} = c_1 n + c_2 n \log(n)$$
- $T(n) = O(n \log(n))$

# Quicksort: анализ



- Упорядоченный массив делится в соотношении 1:n-1
- $T(n) = c_1 + c_2n + T(n-1) = \frac{1}{2}c_2n^2 + c_1n = O(n^2)$

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

\*\*\*\*\*

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

# Quicksort: выбор пивота

---



**1й элемент**

Серединный элемент

Медиана трёх

Случайный элемент

Медиана

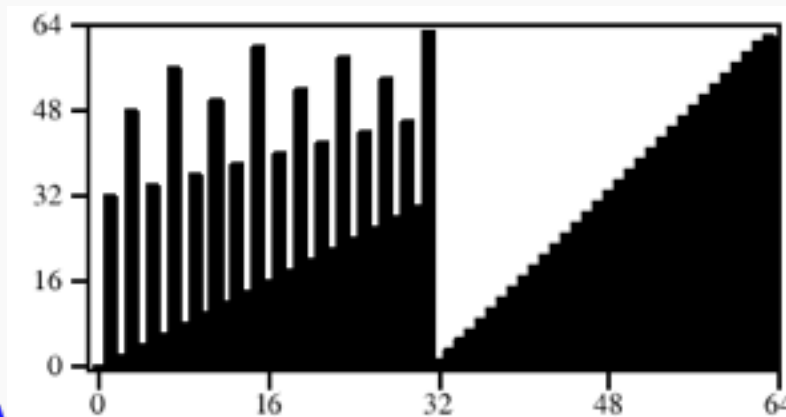
Медиана по трём случайным

...

# Quicksort-killer



- Последовательность, приводящая к времени:  $T(n) = O(n^2)$
- $[1, 2, 3, \dots, n]$   $\Leftrightarrow$  первый элемент
- Для любого predetermined порядка выбора пивота, существует **killer**-последовательность



- [http://www.cs.cmu.edu/~sreer/363/lec07/lec07\\_midmspe.pdf](http://www.cs.cmu.edu/~sreer/363/lec07/lec07_midmspe.pdf)

# Рандомизованная медиана



Медиана  $\Leftrightarrow$  *RandSelect*( $A[1, N], k$ );  $k = N/2$

Случайно выберем элемент из  $A$ :  $A[j]$

Разобьём  $A$  на 2 части: меньше/больше  $A[j]$

Пусть позиция  $A[j]$  в разделённом массиве:  $k$

$k < j$  ?

- Найдём: *RandSelect*( $A[1, j], k$ )
- Иначе: *RandSelect*( $A[j+1, N], k-j$ )

# Quicksort: медиана за линейное время



Медиана  $\Leftrightarrow$  ***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

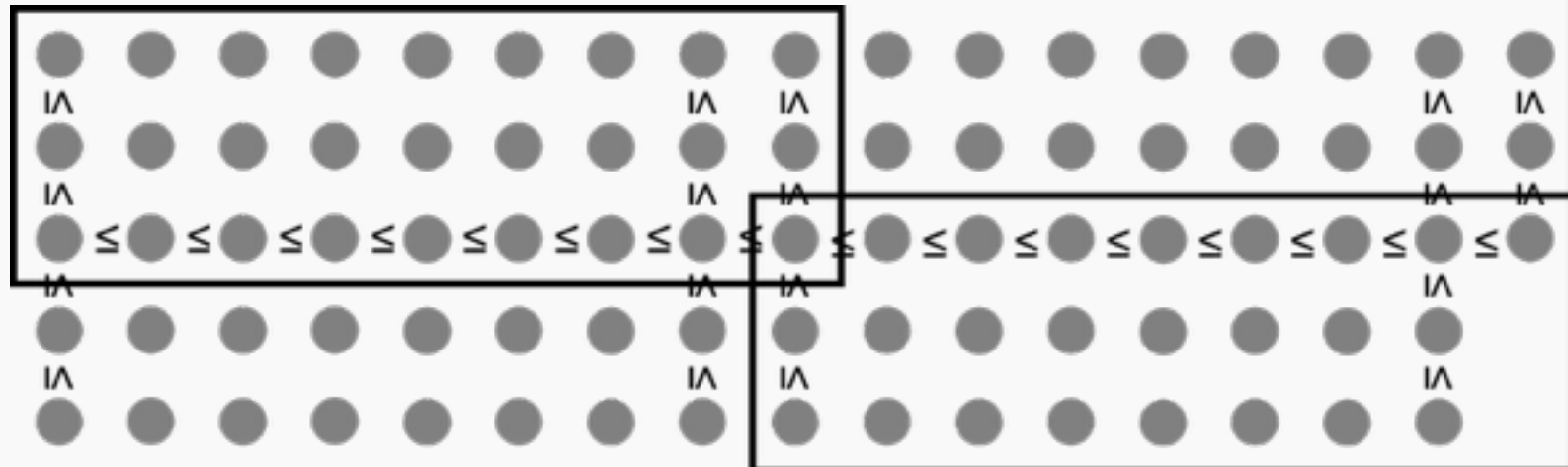
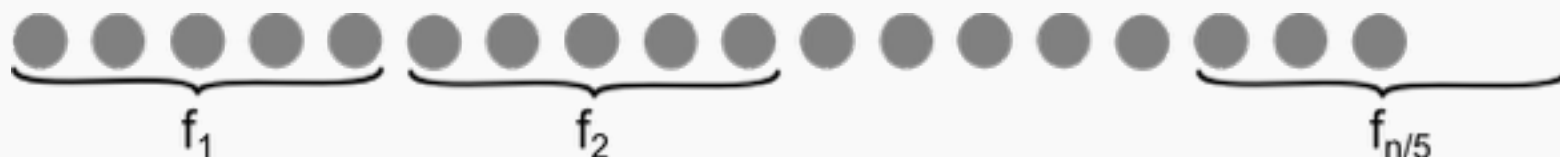
Пусть индекс медианы в массиве  $j$

***j > k ?***

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



# Медиана за линейное время



# Медиана за линейное время: анализ



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

# Сравнение сортировок



Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$ on average, worst case is $n$	typical in-place sort is not stable; stable versions exist
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends [further explanation needed], worst case is $n$	Yes
In-place merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	$n$	$n^2$	$n^2$	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	$n^2$	$n^2$	$n^2$	1	No
Timsort	$n$	$n \log n$	$n \log n$	$n$	Yes
Shell sort	$n$	$n(\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	1	No
Bubble sort	$n$	$n^2$	$n^2$	1	Yes
Binary tree sort	$n$	$n \log n$	$n \log n$	$n$	Yes
Cycle sort	—	$n^2$	$n^2$	1	No
Library sort	—	$n \log n$	$n^2$	$n$	Yes
Patience sorting	—	—	$n \log n$	$n$	No
Smoothsort	$n$	$n \log n$	$n \log n$	1	No
Strand sort	$n$	$n^2$	$n^2$	$n$	Yes
Tournament sort	—	$n \log n$	$n \log n$	$n^{[4]}$	
Cocktail sort	$n$	$n^2$	$n^2$	1	Yes
Comb sort	$n$	$n \log n$	$n^2$	1	No
Gnome sort	$n$	$n^2$	$n^2$	1	Yes
Franceschini's method <sup>[5]</sup>	—	$n \log n$	$n \log n$	1	Yes

# Сортировки без сравнений

---



1. Сортировка подсчетом
2. Поразрядная сортировка от младших к старшим
3. Поразрядная сортировка от старших к младшим
4. Поразрядная быстрая сортировка

# Сортировка подсчётом



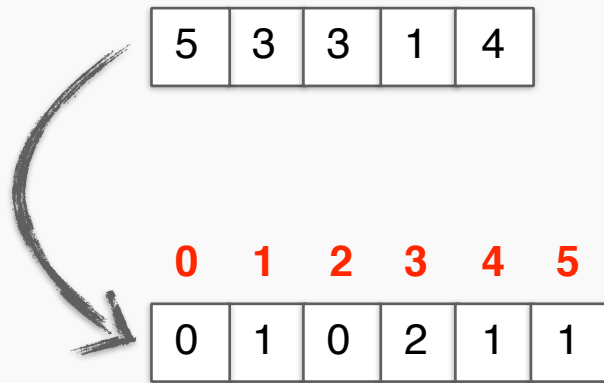
## Algorithm 3.10: CountingSort( $R$ )

Input: (Multi)set  $R = \{k_1, k_2, \dots, k_n\}$  of integers from the range  $[0..\sigma)$ .

Output:  $R$  in nondecreasing order in array  $J[0..n)$ .

```
(1) for  $i \leftarrow 0$  to  $\sigma - 1$  do  $C[i] \leftarrow 0$ 
(2) for  $i \leftarrow 1$  to  $n$  do  $C[k_i] \leftarrow C[k_i] + 1$ 
(3)  $sum \leftarrow 0$ 
(4) for  $i \leftarrow 0$  to  $\sigma - 1$  do           // cumulative sums
(5)      $tmp \leftarrow C[i]$ ;  $C[i] \leftarrow sum$ ;  $sum \leftarrow sum + tmp$ 
(6) for  $i \leftarrow 1$  to  $n$  do           // distribute
(7)      $J[C[k_i]] \leftarrow k_i$ ;  $C[k_i] \leftarrow C[k_i] + 1$ 
(8) return  $J$ 
```

# Сортировка подсчётом



# Сортировка подсчётом



```
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;
}
```

# Сортировка подсчётом

---



- +  $O(n)$  - linear time
- + stable
- + нет сравнений
- требует  $O(n)$  памяти
- $O(n)$  перемещений



# LSD raddix sort



## Algorithm 3.11: LSDRadixSort( $\mathcal{R}$ )

Input: Set  $\mathcal{R} = \{S_1, S_2, \dots, S_n\}$  of strings of length  $m$  over the alphabet  $[0..\sigma)$ .

Output:  $\mathcal{R}$  in increasing lexicographical order.

- (1) for  $\ell \leftarrow m - 1$  to 0 do CountingSort( $\mathcal{R}, \ell$ )
- (2) return  $\mathcal{R}$

# LSD raddix sort



A	00001	R	10010	T	10100	X	11000	P	10000	A	00001	A	00001
S	10011	T	10100	X	11000	P	10000	A	00001	A	00001	A	00001
O	01111	N	01110	P	10000	L	01100	I	01001	R	10010	E	00101
R	10010	X	11000	A	00001	I	01001	R	10010	S	10011	E	00101
T	10100	P	10000	E	00101	S	10011	T	10100	E	00101	G	00111
I	01001	L	01100	A	00001	L	01100	E	00101	G	00111	I	01001
N	01110	A	00001	M	01101	E	00101	M	01101	X	11000	L	01100
G	00111	S	10011	R	10010	N	01110	E	00101	I	01001	M	01101
E	00101	O	01111	S	10011	O	01111	N	01110	L	01100	N	01110
X	11000	I	01001	G	00111	G	00111	O	01111	M	01101	O	01111
A	00001	G	00111							N	01110	P	10000
M	01101	E	00101							L	01100	R	10010
P	10000	A	00001							M	01101	S	10011
L	01100	M	01101							N	01110	T	10100
E	00101	E	00101							O	01111	X	11000

# 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

---



- +  $O(n \cdot \text{key len})$  время работы
- +  $O(n)$  время работы
- + stable
- can't use unstable sort for buckets

# MSD raddix sort



## Algorithm 3.15: MSDRadixSort( $\mathcal{R}, \ell$ )

Input: Set  $\mathcal{R} = \{S_1, S_2, \dots, S_n\}$  of strings over the alphabet  $[0..\sigma)$  and the length  $\ell$  of their common prefix.

Output:  $\mathcal{R}$  in increasing lexicographical order.

- (1) if  $|\mathcal{R}| < \sigma$  then return StringQuicksort( $\mathcal{R}, \ell$ )
- (2)  $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$ ;  $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
- (3)  $(\mathcal{R}_0, \mathcal{R}_1, \dots, \mathcal{R}_{\sigma-1}) \leftarrow \text{CountingSort}(\mathcal{R}, \ell)$
- (4) for  $i \leftarrow 0$  to  $\sigma - 1$  do  $\mathcal{R}_i \leftarrow \text{MSDRadixSort}(\mathcal{R}_i, \ell + 1)$
- (5) return  $\mathcal{R}_\perp \cdot \mathcal{R}_0 \cdot \mathcal{R}_1 \cdots \mathcal{R}_{\sigma-1}$

# MSD raddix sort



0	1	0	0	0
1	0	0	0	0
0	1	1	0	0
0	0	1	1	1
0	1	1	1	0
1	0	1	0	1
1	0	0	1	0
1	0	0	0	0
0	0	1	0	1
0	1	1	1	0
1	1	0	1	1
1	1	1	0	1
0	1	1	0	1
1	0	1	1	1
0	0	0	0	1
1	0	1	0	1

0	1	0	0	0
0	0	0	0	1
0	1	1	0	0
0	0	1	1	1
0	1	1	1	0
0	1	1	0	1
0	1	1	1	0
0	0	1	0	1
1	0	0	0	0
1	0	0	1	0
1	1	0	1	1
1	0	1	0	1
1	0	1	1	1
1	0	0	0	0
1	0	1	0	1

0	0	1	0	1
0	0	0	0	1
0	0	1	1	1
0	1	1	0	0
0	1	1	1	0
0	1	1	0	1
0	1	1	1	0
0	1	0	0	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	1	0	1
1	1	0	1	1

0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	1	0
0	1	1	0	1
0	1	1	1	0
0	1	1	0	0
1	0	0	0	0
1	0	0	1	0
1	0	0	0	0
1	0	1	0	1
1	0	1	1	1
1	1	0	1	1

0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	0
1	0	0	0	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	1	1

0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	0
1	0	0	0	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	1	1

0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	0
1	0	0	0	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	1	1

# MSD raddix sort



- +  $O(n \cdot \text{key\_len})$  время работы\*
- +  $O(n)$  памяти
- + нет сортировки частей размером 1
- + could be unstable
- рекурсивная реализация

# MSD vs LSD



362	291	207	207
436	362	436	253
291	253	253	291
487	436	362	362
207	487	487	397
253	207	291	436
397	397	397	487

LSD Radix Sorting:  
Sort by the last digit, then  
by the middle and the first one

237	237	216	211
318	216	211	216
216	211	237	237
462	268	268	268
211	318	318	318
268	462	462	460
460	460	460	462

MSD Radix Sorting:  
Sort by the first digit, then sort  
each of the groups by the next digit



# Как сортировать ключи разной длины?

---



- \* расширить алфавит пустым символом
- сложность LSD ~ максимальной длине
- + MSD сортирует только непустые суффиксы

# Как сортировать длинные строки?

---



дорогое сравнение если строки отличаются  
только на конце

много итераций LSD на длинных ключах

# MSD sort vs QuickSort

---



первый бит == 1 как опорный элемент

# Binary QuickSort

---



разделим массив на две части

- с ведущим разрядом 0
- с ведущим разрядом 1ы

рекурсивно отсортируем обе части

# Binary quicksort



```
quicksortB(int a[], int l, int r, int w)
{ int i = l, j = r;
  if (r <= l || w > bitsword) return;
  while (j != i)
  {
    while (digit(a[i], w) == 0 && (i < j)) i++;
    while (digit(a[j], w) == 1 && (j > i)) j--;
    exch(a[i], a[j]);
  }
  if (digit(a[r], w) == 0) j++;
  quicksortB(a, l, j-1, w+1);
  quicksortB(a, j, r, w+1);
}
```

# 3-way split



actinian	coenobite	actinian
jeffrey	conelrad	bracteal
coenobite	actinian	coenobite
conelrad	bracteal	conelrad
secureness	secureness	cumin
cumin	dilatedly	chariness
chariness	inkblot	centesimal
bracteal	jeffrey	cankorous
displease	displease	circumflex
millwright	millwright	millwright
repertoire	repertoire	repertoire
dourness	courness	dourness
centesimal	southeast	southeast
fondler	fondler	fondler
interval	interval	interval
reversionary	reversionary	reversionary
dilatedly	cumin	secureness
inkblot	chariness	dilatedly
southeast	centesimal	inkblot
cankorous	cankorous	jeffrey
circumflex	circumflex	displease

# 3-way raddix quick sort



```
#define ch(A) digit(A, D)
void quicksortX(Item a[], int l, int r, int D)
{
    int i, j, k, p, q; int v;
    if (r-l <= M) { insertion(a, l, r); return; }
    v = ch(a[r]); i = l-1; j = r; p = l-1; q = r;
    while (i < j)
    {
        while (ch(a[++i]) < v) ;
        while (v < ch(a[--j])) if (j == l) break;
        if (i > j) break;
        exch(a[i], a[j]);
        if (ch(a[i])==v) { p++; exch(a[p], a[i]); }
        if (v==ch(a[j])) { q--; exch(a[j], a[q]); }
    }
    if (p == q)
        { if (v != '\0') quicksortX(a, l, r, D+1);
          return; }
    if (ch(a[i]) < v) i++;
    for (k = l; k <= p; k++, j--) exch(a[k], a[j]);
    for (k = r; k >= q; k--, i++) exch(a[k], a[i]);
    quicksortX(a, l, j, D);
    if ((i == r) && (ch(a[i]) == v)) i++;
    if (v != '\0') quicksortX(a, j+1, i-1, D+1);
    quicksortX(a, i, r, D);
}
```

# 3-way raddix quick sort



```

now  gig  ace  ago  a|go
for  for  bet  bet  a|ce
tip  dug  dug  and  a|nd
ilk  ilk  cab  ace  b|et
dim  dim  dim  c|ab
tag  ago  ago  c|aw
jot  and  and  c|ue
sob  fee  egg  egg
nob  cue  cue  dug
sky  caw  caw  dim
hut  hut  f|ee
ace  ace  f|or
bet  bet  f|ew
men  cab  ilk
egg  egg  gig
few  few  hut
jay  j|ay  j|a|m
owl  j|ot  j|a|y
joy  j|oy  j|o|y
rap  j|am  j|o|t
gig  owl  owl  m|en
wee  wee  now  owl
was  was  nob  nob
cab  men  men  now
wad  wad  r|ap
caw  sky  sky  sky  sky
cue  nob  was  tip  sob
fee  sob  sob  sob  t|ip  ta|r
tap  tap  tap  tap  t|ap  ta|p
ago  tag  tag  tag  t|ag  ta|q
tar  tar  tar  tar  t|ar  ti|p
dug  tip  tip  w|as
and  now  wee  w|ee
jam  rap  wad  w|ad

```



# Спасибо за внимание!

---

