

академия
больших
данных

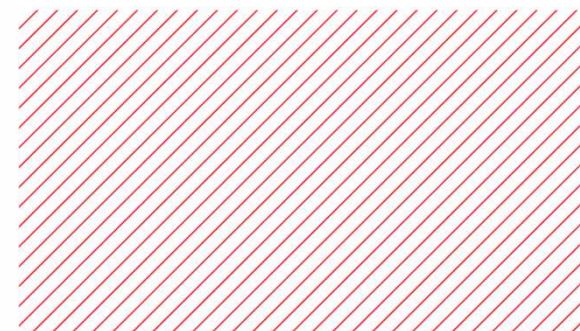
mail.ru
group

made

Сортировки

Корепанов Дмитрий

Алгоритмы и структуры данных





План лекции

Сортировки сравнением:

- SelectionSort - сортировка выбором
- InsertionSort - сортировка вставками
- HeapSort - сортировка кучей
- MergeSort - сортировка слиянием
- QuickSort - быстрая сортировка
- TimSort – сортировка Тима Петерса

К-я порядковая статистика

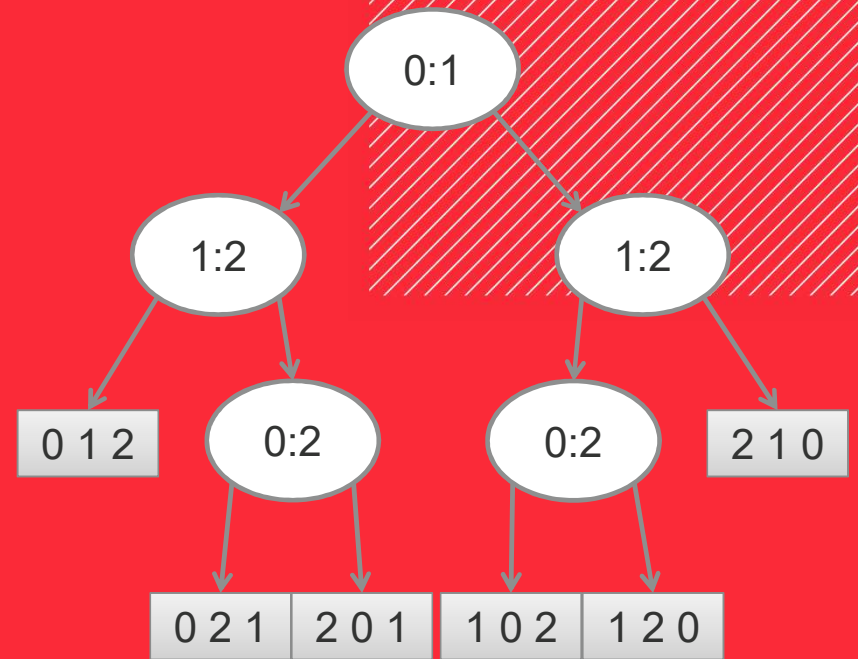
Поразрядные сортировки:

- CountingSort - сортировка подсчётом
- LSD – поразрядная от младших разрядов
- MSD – поразрядная от старших разрядов

Сортировка – процесс упорядочивания элементов массива.

Пример. Сортировка трех:

```
void Sort3( int* a ) {  
    if( a[0] < a[1] ) {  
        if( a[1] < a[2] ) {  
            // 0 1 2  
        } else {  
            if( a[0] < a[2] )  
                // 0 2 1  
            else  
                // 2 0 1  
        }  
    } else {  
        if( a[1] < a[2] ) {  
            if( a[0] < a[2] )  
                // 1 0 2  
            else  
                // 1 2 0  
        } else {  
            // 2 1 0  
        }  
    }  
}
```



**Сортировки
сравнением**



Типы сортировок

Определение. **Стабильная** сортировка – та, которая сохраняет порядок следования равных элементов.

Пример. Сортировка чисел по старшему разряду.

10	25	30	31	24	21	36	32	11
----	----	----	----	----	----	----	----	----

10	11	25	24	21	30	31	36	32
----	----	----	----	----	----	----	----	----



Типы сортировок

Определение. **Локальная** сортировка – та, которая не требует дополнительной памяти.

Примеры:

HeapSort – локальная.

MergeSort – нелокальная.



Типы сортировок

Квадратичные сортировки:

- Сортировка выбором,
- Сортировка вставками,
- Пузырьковая сортировка (не рассматриваем 😊).

Сортировки за $N \cdot \log N$:

- Сортировка кучей
- Сортировка слиянием
- прочие



Сортировка выбором

Во время работы алгоритма:

Массив разделен на 2 части: левая – отсортированная, правая – нет.

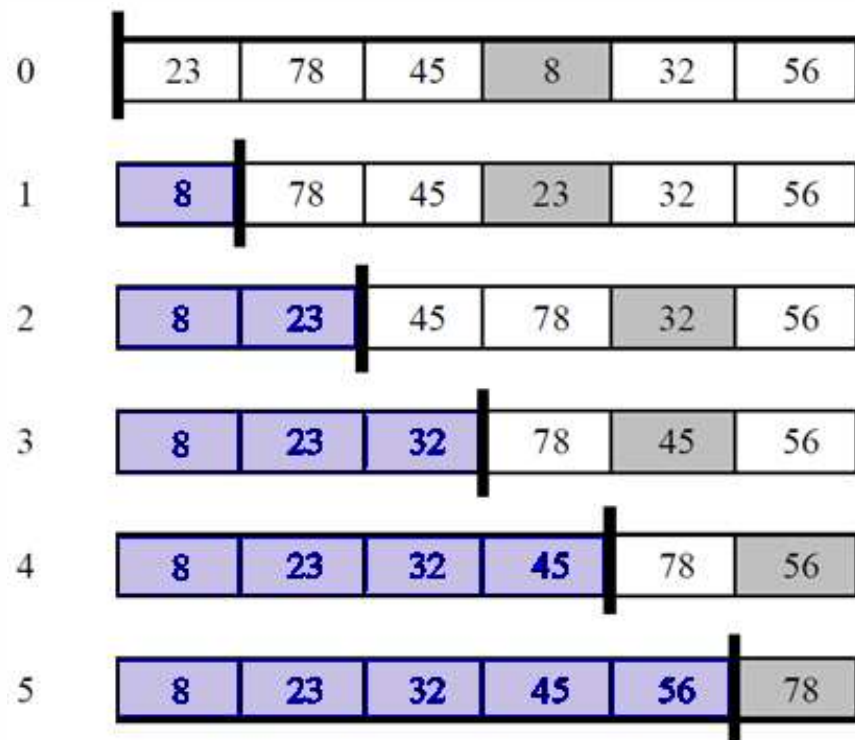
На одном шаге:

- 1) ищем минимум в правой части,
- 2) меняем его с первым элементом правой части,
- 3) сдвигаем границу разделения на 1 вправо.

Свойства:

- Локальная.
- Нестабильная.

Сортировка выбором



Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=0s

Сортировка выбором

```
void SelectionSort( int* a, int n ) {  
    for( int i = 0; i < n - 1; ++i ) {  
        // i - индекс начала правой части.  
        int minIndex = i;  
        for( int j = i + 1; j < n; ++j ) {  
            if( a[j] < a[minIndex] )  
                minIndex = j;  
        }  
        swap( a[i], a[minIndex] );  
    }  
}
```

$\frac{n(n-1)}{2}$ сравнений, $3(n-1)$ перемещений. $T(n) = \Theta(n^2)$.



Сортировка вставками

Простой алгоритм, часто применяемый на малых объемах.

Массив разделен на 2 части:
левая – упорядочена, правая – нет.

На одном шаге:
1) берем первый элемент правой части,
2) вставляем его на подходящее место в левой части.

Свойства:

- Локальная.
- Стабильная.

Сортировка вставками

23	78	45	8	32	56
23	78	45	8	32	56
23	45	78	8	32	56
8	23	45	78	32	56
8	23	32	45	78	56
8	23	32	45	56	78

Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=10s



Сортировка вставками

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



Сортировка вставками. Анализ времени работы.

- Лучший случай: $O(n)$
 - Массив упорядочен по возрастанию.
 - $2 \cdot (n - 1)$ копирований,
 - $(n - 1)$ сравнений.
- Худший случай: $O(n^2)$
 - Массив упорядочен по убыванию.
 - $2 \cdot (n - 1) + \frac{n(n-1)}{2}$ копирований,
 - $\frac{n(n-1)}{2}$ сравнений.
- В среднем: $O(n^2)$



Сортировка вставками. Оптимизации.

- Используем бинарный поиск места вставки в левой части,
- Используем memmove, чтобы эффективно сдвинуть часть элементов левой части вправо на 1 позицию.

$O(n \log n)$ сравнений,

$O(n^2)$ для копирования элементов (с маленькой константой).

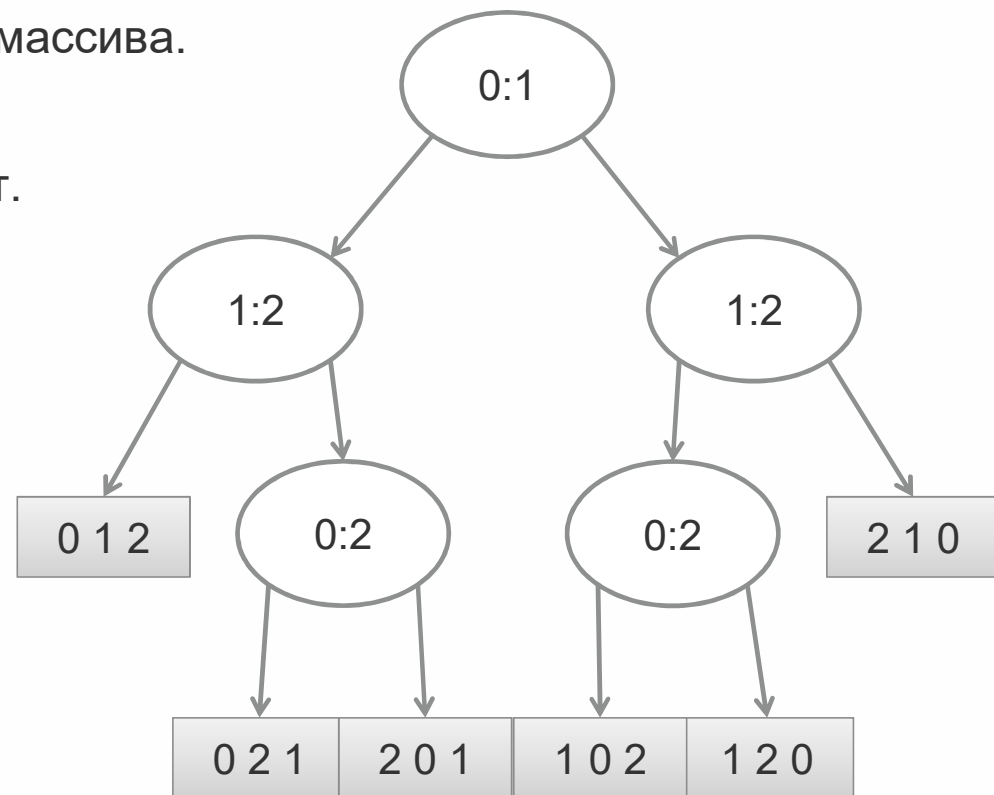
Оценка сложности снизу

В процессе работы алгоритма
сравниваются элементы исходного массива.

Ветвление = дерево.

Окончание работы алгоритма – лист.

Лист = перестановка.





Оценка сложности снизу

Утверждение. Время работы любого алгоритма сортировки, использующего сравнение, $\Omega(N \log N)$.

Доказательство.

Всего листьев в дереве решения не меньше $N!$

Высота дерева не меньше $\log(N!) \cong CN \log N$.

Следовательно, существует перестановка, на которой алгоритм делает не менее $CN \log N$ сравнений.



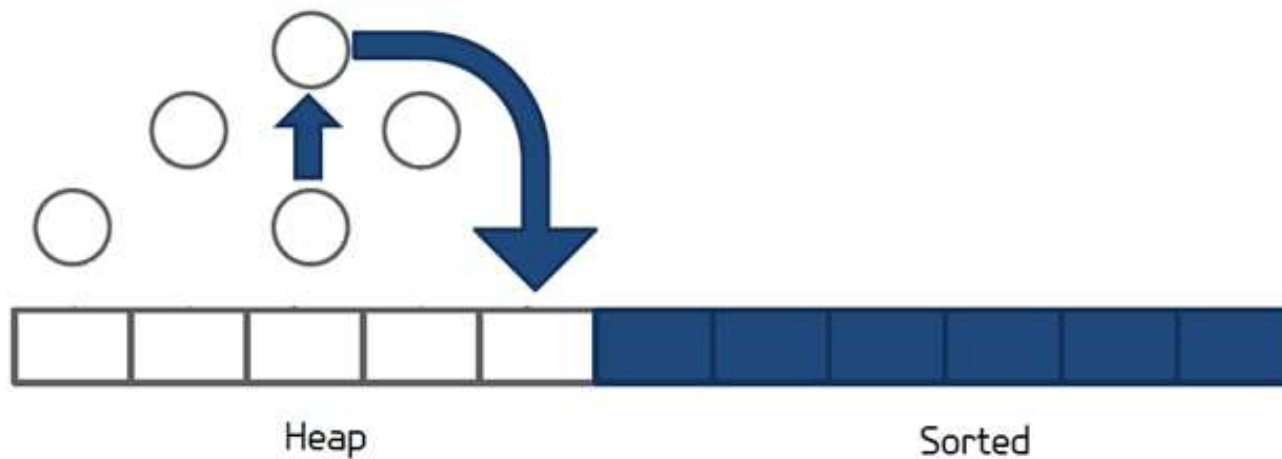
Сортировки за $N \cdot \log N$

- Пирамидальная сортировка – Heap Sort.
- Сортировка слиянием – Merge Sort.
- Быстрая сортировка (сортировка Хоара) – Quick Sort.
- Сортировка Тима Петерса – TimSort.

Пирамидальная сортировка

Аналогия с сортировкой выбором:

Берем максимум из левой части, кладем в конец левой части.



Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=89s



Пирамидальная сортировка

```
void HeapSort( int* a, int n ) {  
    int heapSize = n;  
    BuildHeap( a, heapSize );  
    while( heapSize > 1 ) {  
        // Немного переписанный ExtractMax.  
        swap( a[0], a[heapSize - 1] );  
        --heapSize;  
        SiftDown( a, heapSize, 0 );  
    }  
}
```

$$T(n) = O(n \log n).$$



Сортировка слиянием

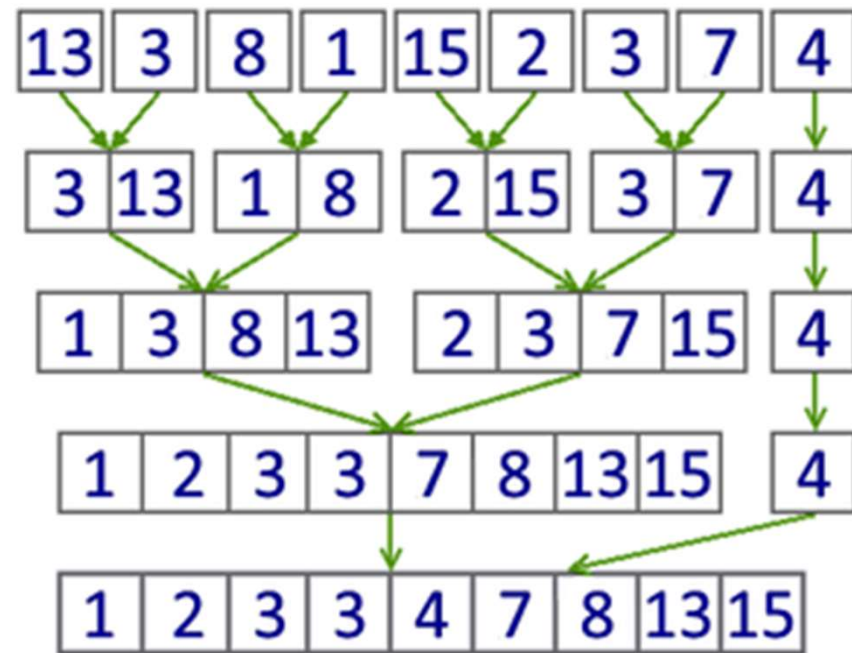
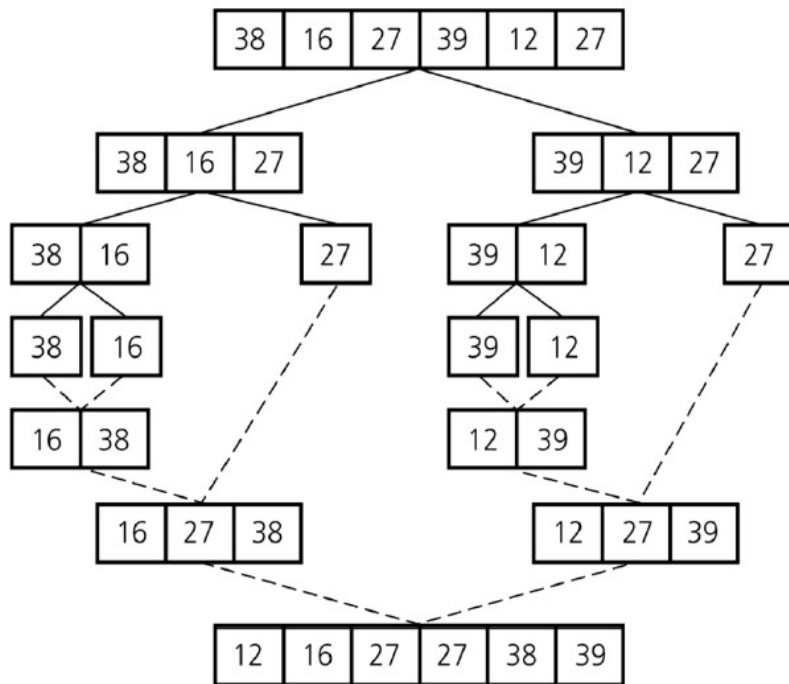
Алгоритм:

1. Разбить массив на два.
2. Отсортировать каждый (рекурсивно).
3. Слить отсортированные в один.

Вариант без рекурсии:

1. Разбить на 2^k подмассива, $2^k < n$.
2. Отсортировать каждый.
3.
 - Слить 1 и 2, 3 и 4, 5 и 6, ..., $2^k - 1$ и 2^k ,
 - Слить 12 и 34, 56 и 78, ...,
 - ...
 - Слить $123 \dots 2^{k-1}$ и $2^{k-1} + 1 \dots 2^k$.

Сортировка слиянием



Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=66s

Слияние двух отсортированных массивов

Слияние двух отсортированных массивов:

- Выберем массив, крайний элемент которого меньше,
- Извлечем этот элемент в массив-результат,
- Продолжим, пока один из массивов не опустеет,
- Копируем остаток второго массива в конец массива-результата.

5	7	15	20
---	---	----	----

9	30	45	90
---	----	----	----

5	7	9				
---	---	---	--	--	--	--



Слияние двух отсортированных массивов

Сложность: $T(n, m) = O(n+m)$.

Количество сравнений:

- В лучшем случае $\min(n, m)$.
- В худшем случае $n+m-1$.



Сортировка слиянием

```
void MergeSort( int* a, int aLen ) {  
    if( aLen <= 1 ) {  
        return;  
    }  
    int firstLen = aLen / 2;  
    int secondLen = aLen - firstLen;  
    MergeSort( a, firstLen );  
    MergeSort( a + firstLen, secondLen );  
    int* c = new int[aLen];  
    Merge( a, firstLen, a + firstLen, secondLen, c );  
    memcpy( a, c, sizeof( int ) * aLen );  
    delete[] c;  
}
```

Свойства:

- Нелокальная.
- Стабильная.



Сортировка слиянием

- Утверждение. Время работы сортировки слиянием $= O(n \log n)$.
- Доказательство.
- Рекуррентное соотношение
- $T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n$,
- разложим дальше
- $T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n \leq 4T\left(\frac{n}{4}\right) + 2c \cdot n \leq \dots \leq 2^k T(1) + k \cdot c \cdot n$.
- $k = \log n$, следовательно,
- $T(n) = O(n \log n)$.

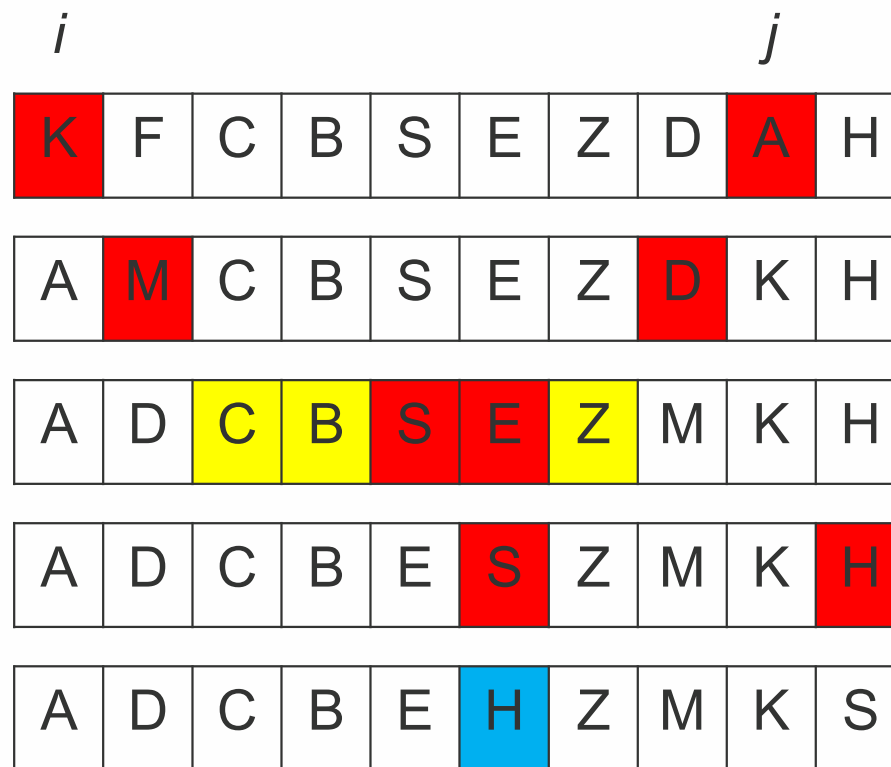
- Используется доп. память $M(n) = O(n)$.



Быстрая сортировка = сортировка Хоара = QuickSort

1. Разделим массив на 2 части,
$$\left\{ \begin{array}{c} \text{элементы} \\ \text{в левой} \end{array} \right\} < \left\{ \begin{array}{c} \text{опорный} \\ \text{элемент} \end{array} \right\} \leq \left\{ \begin{array}{c} \text{элементы} \\ \text{в правой} \end{array} \right\},$$
2. Применим эту процедуру
рекурсивно к левой части и
к правой части.

Быстрая сортировка. Partition.



Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=39s



Быстрая сортировка. Partition.

Разделим массив A . Выберем опорный элемент – pivot .
Пусть опорный элемент лежит в конце массива.

1. Установим 2 указателя:
 i в начало массива, j в конце перед опорным элементом.
2. Двигаем i вправо, пока не встретим элемент больше (или $=$) опорного элемента.
3. Двигаем j влево, пока не встретим элемент меньше опорного элемента.
4. Меняем $A[i]$ и $A[j]$, если $i < j$.
5. Повторяем 2, 3, 4, пока $i < j$.
6. Меняем $A[i]$ и $A[n-1]$ (опорный элемент).

Левая часть – левее опорного элемента, правая – правее. Опорный элемент не входит в них.

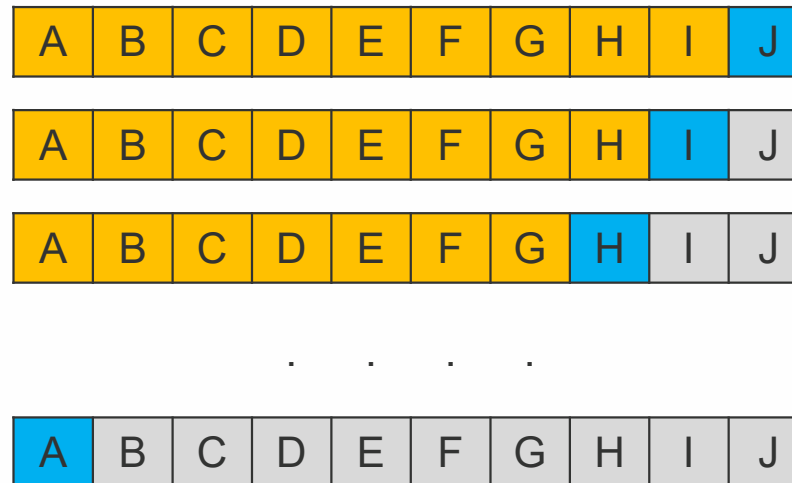
Быстрая сортировка

```
// Возвращает индекс, на который встанет пивот после разделения.
int Partition( int* a, int n ) {
    if( n <= 1 ) {
        return 0;
    }
    const int& pivot = a[n - 1];
    int i = 0; j = n - 2;
    while( i <= j ) {
        // Не проверяем, что i < n - 1, т.к. a[n - 1] == pivot.
        for( ; a[i] < pivot; ++i ) {}
        for( ; j >= 0 && !( a[j] < pivot ); --j ) {}
        if( i < j ) {
            swap( a[i++], a[j--] );
        }
    }
    swap( a[i], a[n - 1] );
    return i;
}

void QuickSort( int* a, int n ) {
    int part = Partition( a, n );
    if( part > 0 ) QuickSort( a, part );
    if( part + 1 < n ) QuickSort( a + part + 1, n - ( part + 1 ) );
}
```

Быстрая сортировка. Анализ.

- Если Partition всегда пополам, то
 $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$, следовательно, $T(n) = O(n \log n)$.
- Утверждение. (без док.)
В среднем $T(n) = O(n \log n)$.
- Если массив упорядочен,
 $\text{pivot} = A[n - 1]$,
то массив делится
в соотношении $n - 1 : 0$.
 $T(n) \leq T(n - 1) + cn \leq$
 $\leq T(n - 2) + c(n + n - 1),$
 $T(n) = O(n^2)$.





Быстрая сортировка. Выбор опорного элемента.

- Последний,
- Первый,
- Серединный,
- Случайный,
- Медиана из первого, последнего и серединного,
- Медиана случайных трех,
- Медиана, вычисленная за $O(n)$,
- ...



Быстрая сортировка. Killer sequence.

Killer-последовательность — последовательность, приводящая к времени $T(n) = O(n^2)$.

Для многих predetermined порядков выбора пивота существует **killer**-последовательность.

- Последний, первый. 1, 2, 3, 4, 5, 6, 7.
- Серединный. x, x, x, 1, x, x, x.
- Медиана трех (первого, последнего и серединного). Массив будем делить в отношении $1 : n - 2$.



Быстрая сортировка

Свойства:

- Локальная.
- Нестабильная. Partition может менять местами равные элементы.



TimSort

Гибридная сортировка Тима Петерса – TimSort (2002г)

Реальные данные часто бывают частично отсортированы.

Используется в Java 7, Python как стандартный алгоритм.

1. Вычисление minRun.
2. Сортировка вставками каждого run.
3. Слияние соседних run (отсортированных).



TimSort. Вычисление minRun

```
• // Вычисление длины стандартного (минимального) run'а.
• // Это число от 32 до 64, которым хорошо укладывается n.
• // n / minRun ~ степень двойки.
• // Например, при n = 96, minRun = 48.
int GetMinrun( int n )
{
    // Станет 1, если среди сдвинутых битов будет хотя бы 1 ненулевой.
    int r = 0;
    while( n >= 64 ) {
        r |= n & 1;
        n >>= 1;
    }
    return n + r;
}
```



TimSort. Вычисление run'ов, их сортировка.

Собираем run:

- Ищем максимально отсортированный подмассив, начиная с текущей позиции.
- Разворачиваем его, если он отсортирован по убыванию.
- Дополняем отсортированный подмассив до minRun элементов.

Сортируем вставками каждый run.

- Отсортированную часть run'а заново не сортируем, только новые элементы вставляем на свои места.

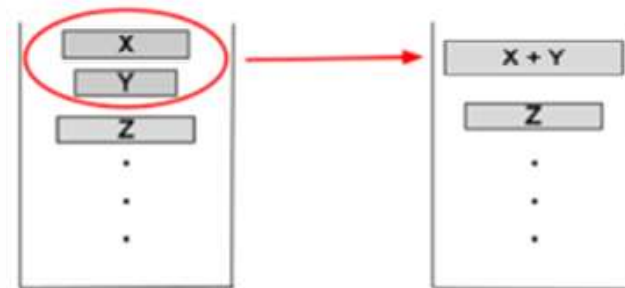
TimSort. Слияние run'ов.

Выполняем слияние соседних run'ов.


- Создается пустой стек пар <индекс начала подмассива>-<размер подмассива>. Берётся первый упорядоченный подмассив.
- В стек добавляется пара данных <индекс начала>-<размер> для текущего подмассива.
- Определяется, нужно ли выполнять процедуру слияния текущего подмассива с предыдущими. Для этого проверяется выполнение двух правил (пусть X , Y и Z — размеры трёх верхних в стеке подмассивов):

$$X > Y + Z$$

$$Y > Z$$



- Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z . Повторяется до выполнения обоих правил или полного упорядочивания данных.
- Слияние оптимизировано галопом.



**К-я порядковая
статистика**



Порядковые статистики

Определение. **К-ой порядковой статистикой** называется элемент, который окажется на К-ой позиции после сортировки массива.

Частный случай - Медиана – срединный элемент после сортировки массива.

4	7	1	3	0	6	8	5	2
---	---	---	---	---	---	---	---	---



Порядковые статистики

Алгоритм. Поиск K -ой порядковой статистики методом «Разделяй и властвуй». $KStatDC(A, n, K)$.

1. Выбираем опорный элемент, вызываем Partition.
2. Пусть позиция опорного элемента после разделения равна P .
 - а) Если $P == K$, то опорный элемент является K -ой порядковой статистикой.
 - б) Если $P > K$, то K -ая порядковая статистика находится слева, вызываем
 $KStatDC(A, P, K)$.
 - в) Если $P < K$, то K -ая порядковая статистика находится справа, вызываем
 $KStatDC(A + (P + 1), n - (P + 1), K - (P + 1))$.



Порядковые статистики

Алгоритм. Поиск K -ой порядковой статистики методом «Разделяй и властвуй». $KStatDC(A, n, K)$.

Время работы

- $T(n) = O(n)$ в лучшем,
- $T(n) = O(n)$ в среднем (без доказательства),
- $T(n) = O(n^2)$ в худшем.



Поразрядные
сортировки



Сортировка подсчетом

Как сортировать без сравнений?

Задача. Отсортировать массив $A[0..n-1]$, содержащий неотрицательные целые числа меньше k .

Решение 1.

- Заведем массив $C[0..k-1]$, посчитаем в $C[i]$ количество вхождений элемента i в массиве A .
- Выведем все элементы C по $C[i]$ раз.



Сортировка подсчетом

```
void CountingSort1( int* a, int n ) {  
    int* c = new int[k];  
    for( int i = 0; i < k; ++i )  
        c[i] = 0;  
    for( int i = 0; i < n; ++i )  
        ++c[a[i]];  
    int pos = 0;  
    for( int i = 0; i < k; ++i ) {  
        for( int j = 0; j < c[i]; ++j ) {  
            a[pos++] = i;  
        }  
    }  
    delete[] c;  
}
```

Сортировка подсчетом

A:

5	3	3	1	4
---	---	---	---	---

C1:

0	1	0	2	1	1
---	---	---	---	---	---

C2:

0	0	1	1	3	4
---	---	---	---	---	---

B

1	3	3	4	5
---	---	---	---	---



Сортировка подсчетом

Решение 2. Не создает элементы A , а использует копирование. Полезно при сортировке структур по некоторому полю.

- Заведем массив $C[0, \dots, k - 1]$, посчитаем в $C[i]$ количество вхождений элемента i в массиве A .
- Вычислим границы групп элементов для каждого $i \in [0, \dots, k - 1]$ (начальные позиции каждой группы).
- Создадим массив для результата B .
- Переберем массив A . Очередной элемент $A[i]$ разместим в B в позиции группы $C[A[i]]$. Сдвинем текущую позицию группы.
- Скопируем B в A .



Сортировка подсчетом

```
void CountingSort2( int* a, int n ) {
    int* c = new int[k];
    for( int i = 0; i < k; ++i )
        c[i] = 0;
    for( int i = 0; i < n; ++i )
        ++c[a[i]];
    int sum = 0;
    for( int i = 0; i < k; ++i ) {
        int tmp = c[i];
        c[i] = sum; // Начала групп.
        sum += tmp;
    }
    int* b = new int[n];
    for( int i = 0; i < n; ++i ) {
        b[c[a[i]]++] = a[i];
    }
    delete[] c;
    memcpy( a, b, n * sizeof( int ) );
    delete[] b;
}
```



Сортировка подсчетом

Сортировка подсчетом – стабильная, но не локальная.

Время работы $T(n, k) = O(n + k)$.

Доп. память $M(n, k) = O(n + k)$.



Сортировка подсчетом

```
void CountingSort2( int* a, int n ) {
    int* c = new int[k];
    for( int i = 0; i < k; ++i )
        c[i] = 0;
    for( int i = 0; i < n; ++i )
        ++c[a[i]];
    for( int i = 1; i < k; ++i ) {
        c[i] += c[i - 1]; // Концы групп.
    }
    int* b = new int[n];
    for( int i = n - 1; i >= 0; --i ) { // Проход с конца.
        b[--c[a[i]]] = a[i];
    }
    delete[] c;
    memcpy( a, b, n * sizeof( int ) );
}
```



Поразрядная сортировка = Radix sort

Если диапазон значений велик – сортировка подсчетом не годится.

Строки, целые числа можно разложить на разряды. Диапазон значений разряда не велик.

Можно выполнять сортировку массива по одному разряду, используя сортировку подсчетом.

С какого разряда начать сортировку?

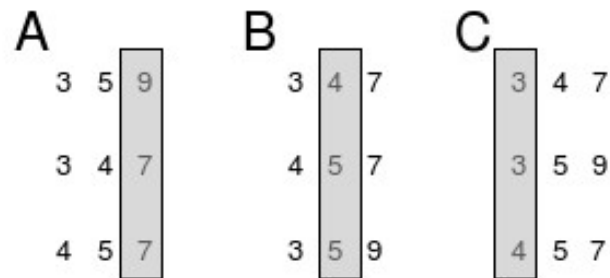
- LSD – least significant digit.
- MSD – most significant digit.

Поразрядная сортировка. LSD.

Least Significant Digit.

Сначала сортируем подсчетом по младшим разрядам, затем по старшим.

Ключи с различными младшими разрядами, но одинаковыми старшими не будут перемешаны при сортировке старших разрядов благодаря стабильности поразрядной сортировки.



Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=115s



Поразрядная сортировка. LSD.

Время работы $T(n, k, r) = O(r \cdot (n + k))$,

доп. память $M(n, k, r) = O(n + k)$,

где n — размер массива, k — размер алфавита, r — количество разрядов.

Поразрядная сортировка. MSD.

Most Significant Digit.

Сначала сортируем подсчетом по старшим разрядам, затем по младшим.

Чтобы не перемешать отсортированные старшие разряды, сортируем по младшим только группы чисел с одинаковыми старшими разрядами отдельно друг от друга.

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

Визуализация: https://www.youtube.com/watch?v=Gnp8G1_kO3I&t=131s



Поразрядная сортировка. MSD.

Время работы в лучшем случае $T(n, k, r) = O(n \cdot \log_k n)$,

Время работы в худшем случае $T(n, k, r) = O(r \cdot n \cdot k)$,

доп. память $M(n, k, r) = O(n + r \cdot k)$,

где n – размер массива, k – размер алфавита, r – количество разрядов.



Binary QuickSort

Похожа на MSD по битам.

1. Сортируем по старшему биту.
Это Partition с фиктивным пивотом 10000..0.
2. Рекурсивно вызываем
от левой части = 0xxxxxxx,
от правой части = 1xxxxxxx.

Binary QuickSort

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



Binary QuickSort

Время работы $T(n, r) = O(rn)$,

доп. память $M(n, r) = O(1)$,

где n — размер массива, r — количество разрядов.

Нестабильна!

Зато локальна.

Сравнение сортировок. Итог.

Алгоритм	В лучшем	В среднем	В худшем	Память	Стабильность	Метод
Quicksort	$n \log n$	$n \log n$	n^2	1	No	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Selection sort	n^2	n^2	n^2	1	Yes	Selection
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging
LSD	$r(k + n)$	$r(k + n)$	$r(k + n)$	$k + n$	Yes	Radix
MSD	$n \log n$	$n \log n$	rnk	$rk + n$	Yes	Radix
Binary QuickSort	$n \log n$	$n \log n$	rn	1	No	Radix



Вопросы?
