Лекция 4.
Сортировки (часть 1).
Порядковые статистики

Алгоритмы и
структуры данных

Мацкевич С.Е.

Быстрая сортировка = сортировка Хоара = QuickSort

1. Разделим массив на 2 части, $\{\text{элементы в левой}\} \le \{\text{элементы в правой}\}$,
2. Применим эту процедуру рекурсивно к левой части и к правой части.

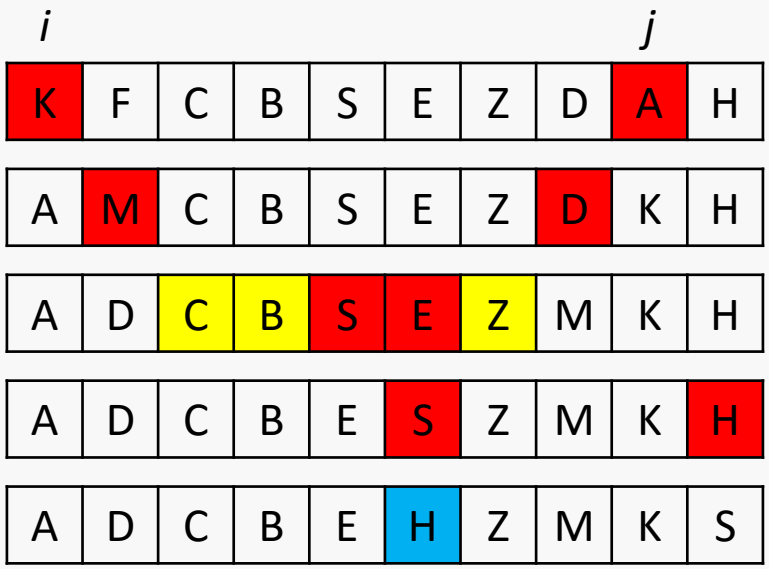
Быстрая сортировка. Partition.

Разделим массив A. Выберем разделяющий элемент – пивот.
Пусть пивот лежит в конце массива.

1. Установим 2 указателя:
   i в начало массива, j в конце перед пивотом.
2. Двигаем i вправо, пока не встретим элемент больше (или =) пивота.
3. Двигаем j влево, пока не встретим элемент меньше пивота.
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] (пивот).

Левая часть – левее пивота, правая – правее. Пивот не входит в них.

Быстрая сортировка. Partition.

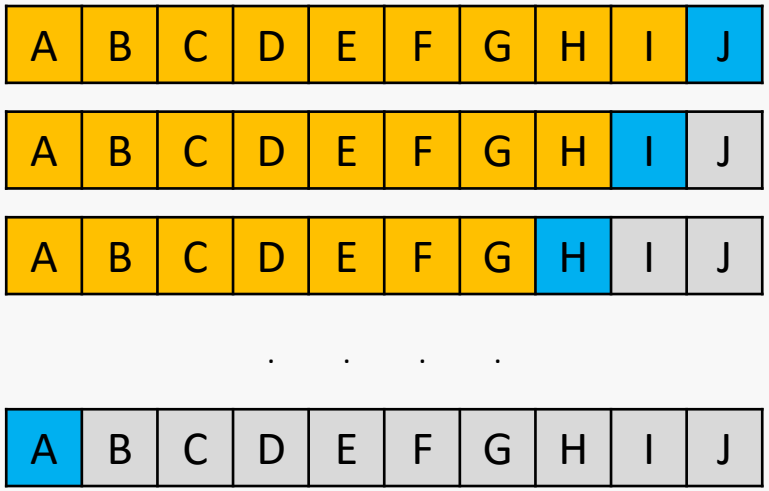


Быстрая сортировка


```
// Возвращает индекс, на который встанет пивот после разделения.
void 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
  $T(n) \le 2T\left(\frac{n}{2}\right) + cn$, следовательно, $T(n) = O(n \log n)$.
- Утверждение. (без док.)
  В среднем $T(n) = O(n \log n)$.
- Если массив упорядочен,
  пивот = $A[n-1]$,
  то массив делится
  в соотношении $n - 1 : 0$.
  $T(n) \le T(n-1) + cn \le$
  $\le T(n-2) + c(n + n - 1)$,
  $T(n) = O(n^2)$.



Быстрая сортировка. Выбор пивота.

- 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 предопределенных 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.

Порядковые статистики

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

Медиана — серединный элемент после сортировки массива.

4	7	1	3	0	6	8	5	2

Порядковые статистики

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Алгоритм 1. Поиск K-ой порядковой статистики методом «Разделяй и властвуй». KStatDC(A, N, K).

1. Выбираем пивот, вызываем Partition.
2. Пусть позиция пивота после разделения равна P.
   а) Если P == K, то пивот является K-ой порядковой статистикой.
   б) Если P > K, то K-ая порядковая статистика находится слева,
      вызываем KStatDC(A, N – 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 1. 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.

Порядковые статистики

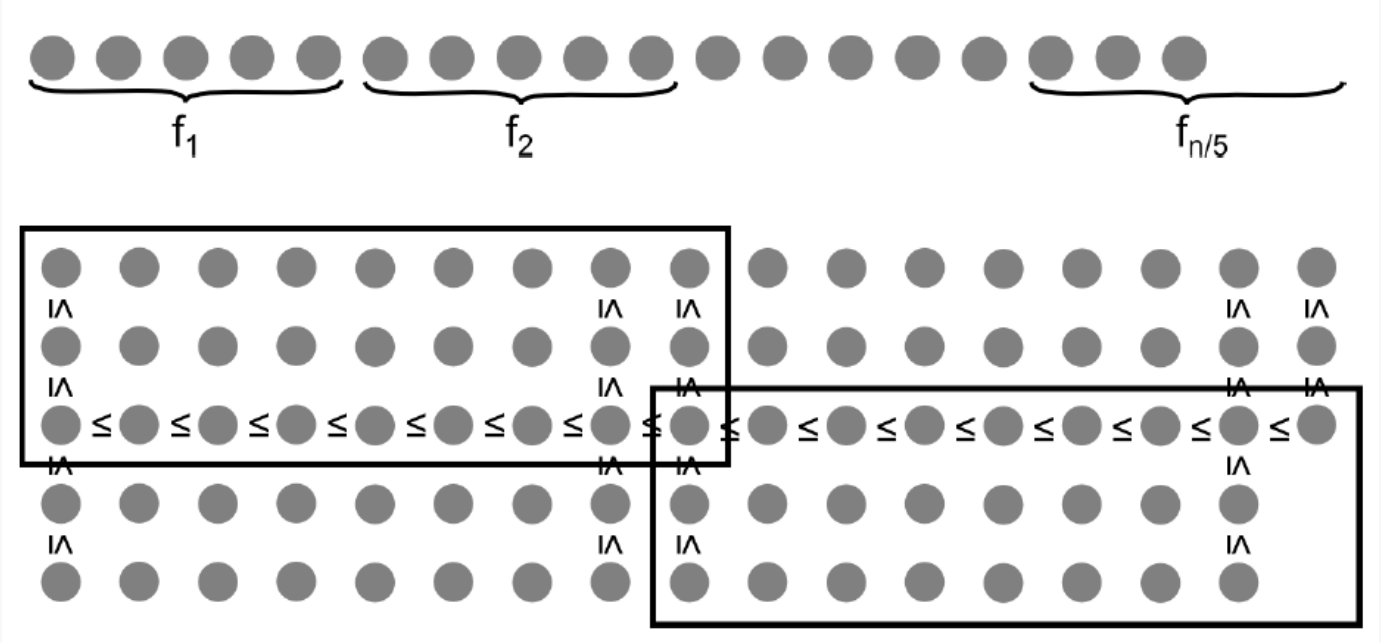
МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Алгоритм 2 (Блюма-Флойда- Пратта-Ривеста-Тарьяна). Поиск K-ой порядковой статистики за линейное время. KStatLin(A, N, K).

1. Разобьем массив на пятерки.
2. Сортируем каждую пятерку, выбираем медиану из каждой пятерки.
3. Ищем M — медиану медиан пятерок, вызвав KStatLin(Medians, N/5, N/10).
4. Разделяем по пивоту M, вызывая обычный Partition.
5. Пусть позиция пивота M после разделения равна P. ($0.3n \le P \le 0.7n$)
   а) Если P == K, то пивот является K-ой порядковой статистикой.
   б) Если P > K, то K-ая порядковая статистика находится слева, вызываем KStatLin(A, N — P, K).
   в) Если P < K, то K-ая порядковая статистика находится справа, вызываем KStatLin(A + (P + 1), N — (P + 1), K — (P + 1)).

Порядковые статистики

Алгоритм 2 (Блюма-Флойда- Пратта-Ривеста-Тарьяна). Поиск K-ой порядковой статистики за линейное время. KStatLin(A, n, K).



Порядковые статистики

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Алгоритм 2 (Блюма-Флойда- Пратта-Ривеста-Тарьяна). Поиск K-ой порядковой статистики за линейное время. KStatLin(A, n, K).
Время работы:

$$T(n) \le T\left(\frac{n}{5}\right) + cn + T(0.7n).$$

По индукции докажем, что $T(n) \le 10cn$:

$$T(n) \le \frac{10cn}{5} + cn + 7cn = 10cn.$$

Итак,

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

Типы сортировок

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

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.

Сортировка подсчетом

Как сортировать без сравнений?

Задача. Отсортировать массив `A[0..n-1]`, содержащий неотрицательные целые числа меньшие `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

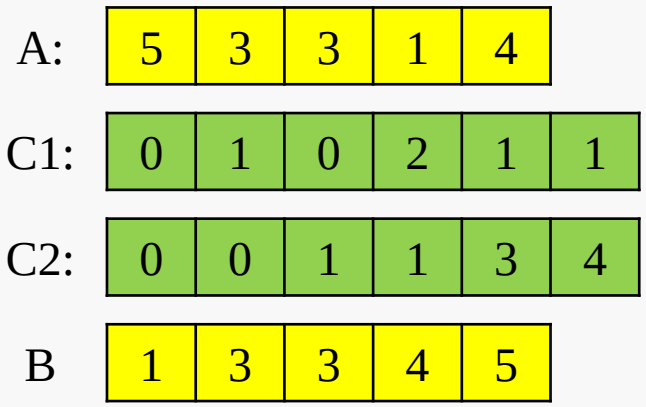
Решение 2. Не создает элементы A, а использует копирование. Полезно при сортировке структур по некоторому полю.

- Заведем массив $C[0,..,k-1]$, посчитаем в C[i] количество вхождений элемента i в массиве A.
- Вычислим границы групп элементов для каждого $i \in [0,..,k-1]$ (начальные позиции каждой группы).
- Создадим массив для результата B.
- Переберем массив A. Очередной элемент $A[i]$ разместим в B в позиции группы $C\left[A[i]\right]$. 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( b, a, n * sizeof( int ) );
}
```

Сортировка подсчетом



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]];
    int sum = 0;
    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( b, a, 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?

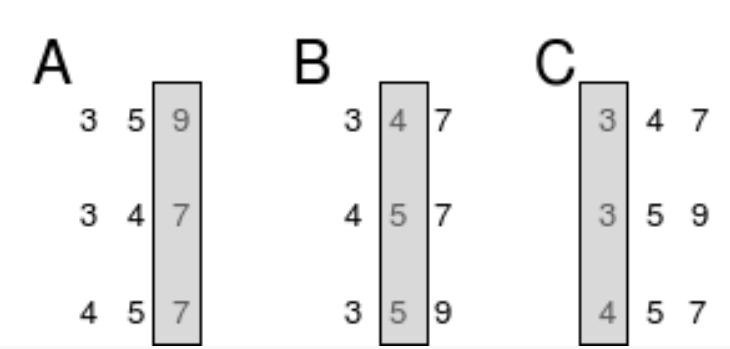
- LSD – least significant digit.
- MSD – most significant digit.

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 сортировки старших разрядов благодаря стабильности поразрядной сортировки.



Поразрядная сортировка. LSD.


```
// Поразрядная сортировка по одному конкретному байту.
void CountingSort( long long* a, int n, int byte );

void LSDSort( long long* a, int n ) {
    for( int r = 0; r < sizeof( long long ); ++r )
        CountingSort( a, n, r );
}
```

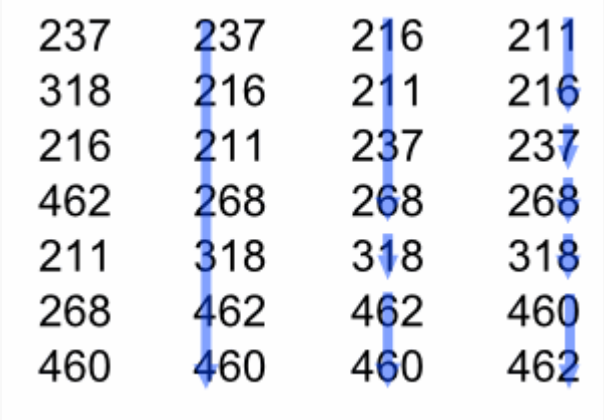
Время работы $T(n,k,r) = O\big(r \cdot (n+k)\big)$,
доп. память $M(n,k,r) = O(n+k)$,
где n – размер массива, k – размер алфавита, r – количество разрядов.

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.



Поразрядная сортировка. MSD.


```
// Поразрядная сортировка по одному конкретному байту.
// Заполняет массив c[0..k-1] – начала групп.
void CountingSort( long long* a, int n, int* c, int byte );

void MSDSort( long long* a, int n, int byte ) {
    if( n <= 1 )
        return;
    int* c = new int[k + 1];
    CountingSort( a, n, c, byte );
    if( byte > 0 ) {
        for( int i = 0; i < k; ++i )
            MSDSort( a + c[i], c[i + 1] – c[i], byte - 1 );
    }
    delete[] c;
}
```

Время работы $T(n,k,r) = O(r \cdot n \cdot k)$,
доп. память $M(n,k,r) = O(n + r \cdot k)$,
где n – размер массива, k – размер алфавита, r – количество разрядов.

Поразрядная сортировка. Ключи разной длины.

Расширим алфавит пустым символом “\0”.

+ MSD можно не вызывать для группы с текущим разрядом = “\0”.

Для массива строк различной длины такой MSD будет эффективнее.

– LSD будет обрабатывать все разряды в каждом ключе.
Время работы пропорционально длине k:

$$T(n) = O(nk)$$

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

Binary QuickSort


```
void BinaryQuickSort( int* a, int l, int r, int w ) {
    if( r <= l || w > BitsInWord )
       return;
    int i = l; j = r;
    while( i != j ) {
        while( digit( a[i], w ) == 0 && i < j ) ++i;
        while( digit( a[j], w ) == 1 && i < j ) --j;
        swap( a[i], a[j] );
    }
    if( digit( a[r], w ) == 0 )
        ++j;
    BinaryQuickSort( a, l, j - 1, w + 1 );
    BinaryQuickSort( a, j, r, w + 1 );
}
```

Binary QuickSort

Время работы $T(n,r) = O(rn)$,
доп. память $M(n,r) = O(1)$,
где n – размер массива, r – количество разрядов.

Не стабильна!
Зато локальна.

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


```
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. Вычисление run’ов, их сортировка.

Выполняем слияние соседних run’ов.

- Используем стек убывающих run’ов.
- Не сливаем, если $X_n > X_{n+1} + X_{n+2}$ и $X_{n+1} > X_{n+2}$. Иначе сливаем, X_{n+1} и меньший из соседних.
- Слияние оптимизировано меньшим промежуточным буфером.
- Слияние оптимизировано галопом.

TimSort

Визуализация:

http://www.youtube.com/watch?v=NVIjHj-lrT4

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
In-place merge sort	—	—	$n \log^2 n$	1	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	No	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

МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Спасибо за внимание!