Cheatsheet 01 — Алгоритмы сортировки

Пререквизиты

> **Info**
>
> **Инвариант** — состояние данных, неизменное на каждом шаге алгоритма.

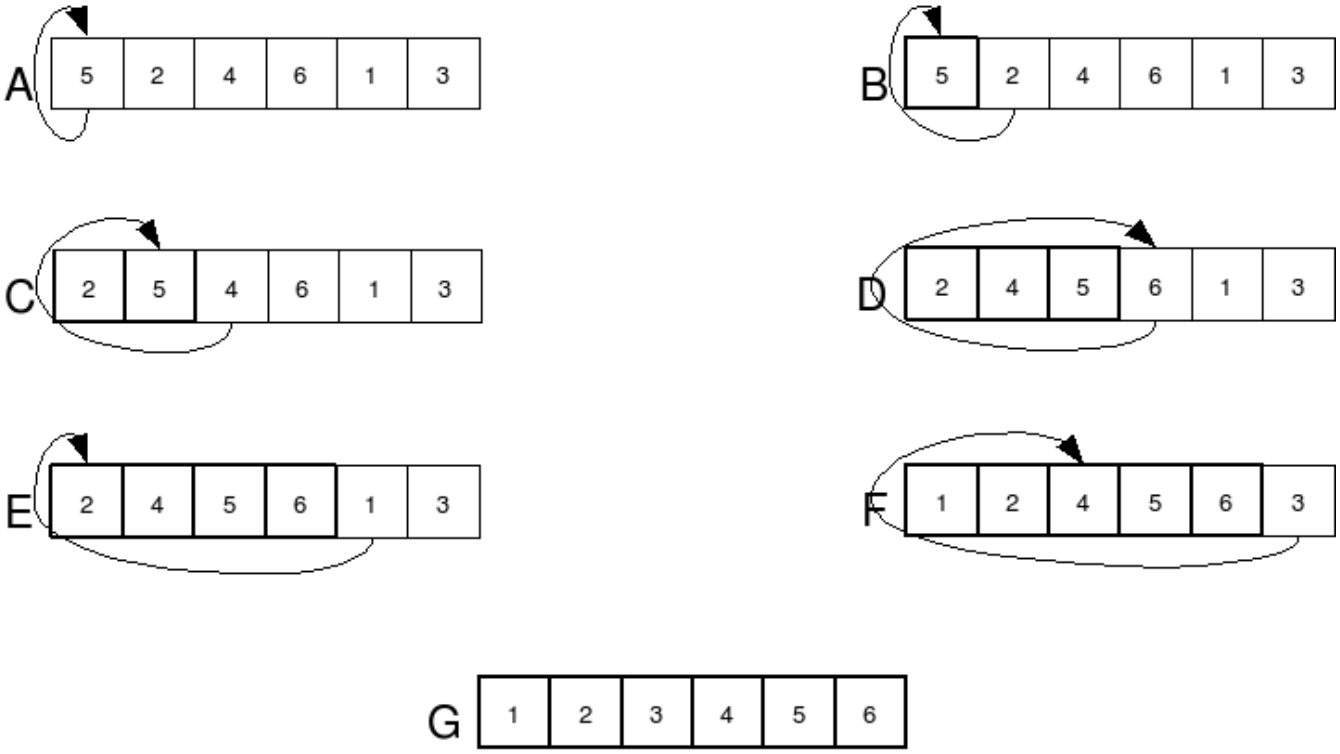
> **Info**
>
> Сортировка называется **устойчивой**, если она сохраняет исходный порядок элементов при равенстве значений.

Сортировка вставками

Теория

Инвариант: к концу i-й итерации алгоритма первые i элементов отсортированы.

Идея: на каждой итерации мы хотим увеличивать длину отсортированного префикса ровно на один элемент. Этот элемент может как быть, так и не быть отсортирован относительно предыдущих. В случае, если он не отсортирован, нам необходимо найти такую позицию на текущем префиксе, которую должен занимать этот элемент, и "вставить" его на это место. Этот процесс реализуется так: будем менять местами текущий элемент с предыдущим до тех пор, пока не выполняется условие отсортированности. Когда оно впервые выполнится, новый элемент встанет на нужное место.



В некоторых ситуациях имеет смысл воспользоваться *бинарным поиском* для определения нужной позиции для следующего элемента. Это не скажется на асимптотике, покольку в процессе вставки мы всё ещё вынуждены менять элементы местами, однако уменьшение количества сравнений до $O(\log n)$ за итерацию может существенно уменьшить реальное время работы алгоритма, если эта операция является достаточно ресурсоёмкой (например, для больших строк или каких-то более сложных объектов).

Реализация

```
void InsertionSort(int* array, int array_size) {
// Передаём массив и его размер
    for (int i = 0; i < array_size; ++i) {
    // Перебираем вставляемый элемент
        for (int j = i; j > 0 && array[j - 1] > array[j]; --j) {
        // Пока элемент не на своём месте, меняем его местами с предыдущим
            std::swap(array[j - 1], array[j]);
        }
    }
}
```

Характеристика

Временная сложность — $O(n^2)$
Доказательство: пусть $k_i = |i - j|$, где i — позиция элемента в исходном массиве, j —

позиция элемента в отсортированном массиве. Тогда алгоритм совершит порядка $\sum_{i=0}^{n} k_i$ действий, поскольку сдвиг элемента — единственная операция, которую мы выполняем. Асимптотическая оценка этой величины — $O(n \max k)$, что эквивалентно $O(n^2)$.

Сложность по памяти — $O(1)$. Дополнительная память не выделяется.

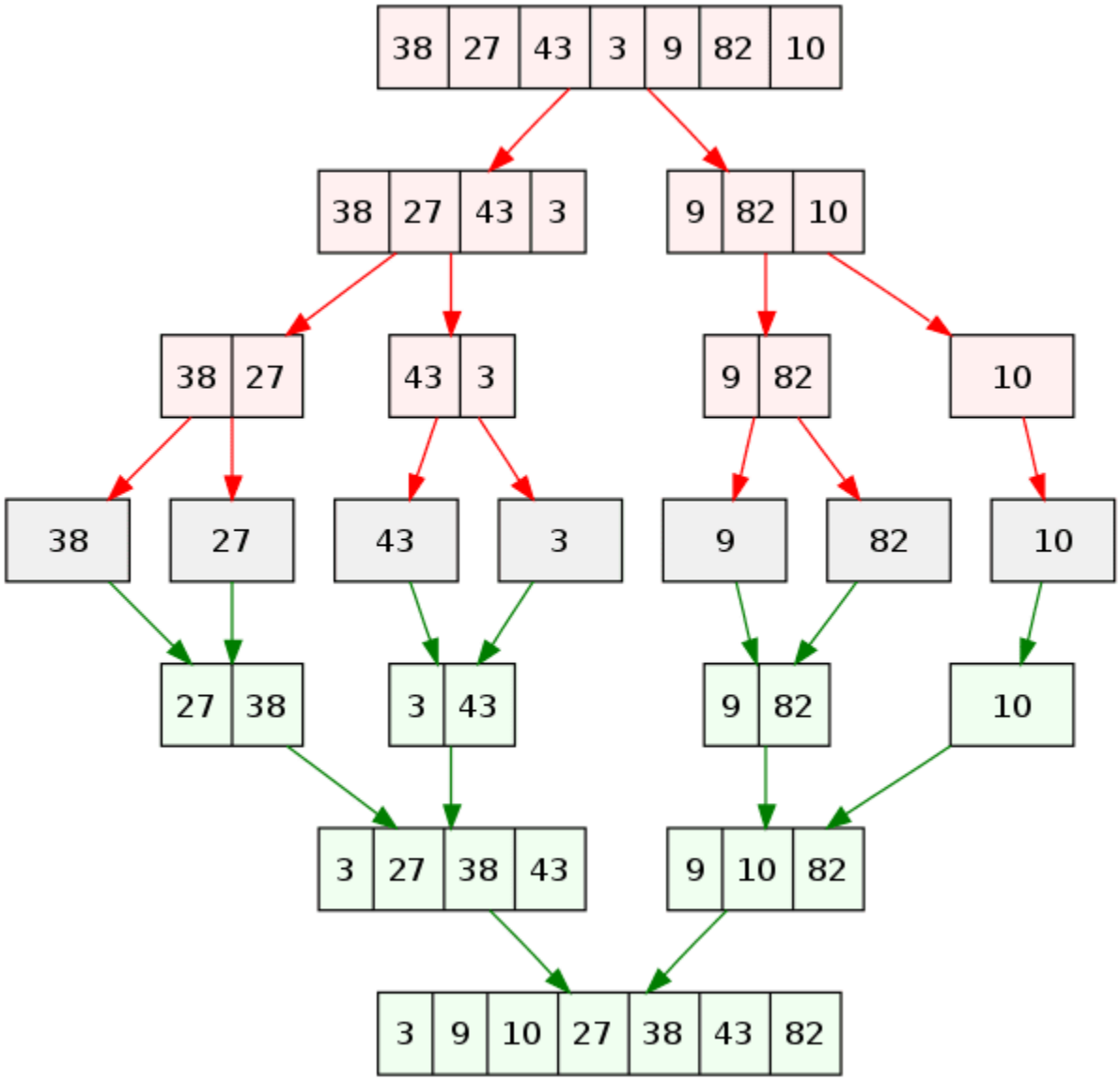
Устойчивость: можно написать как устойчивую, так и неустойчивую реализацию. Они будут отличаться только строгостью сравнения элементов.

Сортировка слиянием

Теория

Инвариант: к моменту вызова `merge` левая и правая половины подмассива отсортированы.

Идея: будем рекурсивно разбивать массив пополам до тех пор, пока не получим подмассив из одного элемента. Теперь мы хотим обратно "слить" воедино пары подмассивов, начиная с самых маленьких, но так, чтобы в процессе поддерживать нужный порядок элементов:



Заметим, что массив из одного элемента уже отсортирован. Воспользуемся этим фактом и определим функцию `merge`, которая будет "сливать" два отсортированных массива в один, также отсортированный.

Пусть мы хотим слить два массива a и b в третий массив c. Допустим также, что мы уже корректно заполнили первые $l + r - 2$ элементов c, причём из них ровно $l - 1$ принадлежат a и ровно $r - 1$ принадлежат b.
На примере a покажем, что это будут первые элементы обоих массивов.

Если это не так, то есть среди первых $l + r - 2$ элементов есть какой-то $a_k, k \geq l$, то далее в массиве c обязан встретиться такой $a_i, 1 \leq i < l$, что $a_i \leq a_k$ (поскольку массив a отсортирован). Если $a_i = a_k$, то это просто нарушит устойчивость сортировки, однако если $a_i < a_k$, то это нарушит и порядок, а поскольку мы предполагаем, что алгоритм корректен, то исходное предположение выполняться не может.

Теперь мы хотим понять, каким будет $l + r - 1$-й элемент массива c. Здесь возникают 4 варианта:

1. Если $l \leq |a|, r \leq |b|, a_l < b_r$, то:
   - $\forall i < l : a_i \leq a_l$, поскольку массив a отсортирован;
   - $\forall i < r : b_i \leq a_l$, поскольку если это не так, то в дальнейшем нарушится порядок сортировки (аналогично док. выше);
   - $\forall i > l : a_i \geq a_l$, поскольку массив a отсортирован;
   - $\forall i > r : b_i \geq b_r > a_l$, поскольку массив b отсортирован.

   Таким образом, a_l не меньше всех элементов из $[a_0 \dots a_l)$ и из $[b_0 \dots b_r)$, т.е. не меньше всех тех элементов, что лежат в c до c_{l+r-1}, и не больше всех элементов из $(a_l \dots a_{|a|}]$ и из $(b_r \dots b_{|b|}]$, т.е. не больше всех тех элементов, что будут лежать в c позже, чем c_{l+r-1}. Нетрудно заметить, что в таком случае $c_{l+r-1} = a_l$ по определению.
2. Если $l \leq |a|, r \leq |b|, b_r < a_l$, то аналогичными пункту (1) рассуждениями получаем, что $c_{l+r-1} = b_r$.
   Примечание: случай $a_l = b_r$ валидно отнести к любому из описанных выше.
3. Если $l = |a|$, то все элементы из a уже стоят на своих местах в массиве c. Заметим, что тогда:
   - $\forall x \in a : b_r \geq x$, поскольку если $\exists a_k > b_r$, то мы бы не взяли его в c раньше, чем b_r (см. пункт 2);
   - $\forall i < r : b_i \leq b_r$, поскольку массив b отсортирован;
   - $\forall i > r : b_i \geq b_r$, поскольку массив b отсортирован.

   Таким образом, b_r больше всех элементов из a и из $[b_0 \dots b_r)$, т.е. больше всех тех элементов, что лежат в c до c_{l+r-1}, и меньше всех элементов из $(b_r \dots b_{|b|}]$, т.е. меньше всех тех элементов, что будут лежать в c позже, чем c_{l+r-1}. Нетрудно заметить, что в таком случае $c_{l+r-1} = b_r$ по определению.
4. Если $r = |b|$, то аналогичными пункту (3) рассуждениями получаем, что $c_{l+r-1} = a_l$.

Таким образом, реализация функции `merge` сводится к рассмотрению всех этих случаев и к выбору оптимального варианта на каждой итерации.

Реализация

Важное замечание: мы рассматриваем блоки, на которые разбиваем массив, не как отдельные объекты, а как подмассивы, ограниченные какими-то индексами внутри исходного массива, причём эти индексы образуют не отрезок, а *полуинтервал*. Это необходимо как для эффективности алгоритма, так и для удобства реализации.

```
int merged_array[kMaxArraySize];
// массив под слияние. объявляем глобально и переиспользуем, чтобы
сэкономить память

void Merge(int* array, int left_index_first, int right_index_first, int
left_index_second, int right_index_second) {
```

```
/* Передаём сам массив и границы подмассивов, которые будем сливать
- [left_index_first, right_index_first) — левый подмассив
- [left_index_second, right_index_second) — правый подмассив
- [left_index_first, right_index_second) — подмассив, от которого мы вызвали
MergeSort() и куда мы запишем результат слияния */
    int array_begin = left_index_first;
    /* Запоминаем, откуда потом будем начинать записывать слитый массив. Это
нужно сделать по той причине, что после сортировки значение  переменной, в
которой лежит левая граница, изменится */
    for (int i = 0; i < right_index_second; ++i) {
        if (left_index_first == right_index_first) {
        // Если левый подмассив кончился, то берем элемент из правого и
двигаем границу r
            merged_array[i] = array[left_index_second++];
        } else if (left_index_second == right_index_second) {
        // Если правый подмассив кончился, то берем элемент из левого и
двигаем границу l
            merged_array[i] = array[left_index_first++];
        } else if (array[left_index_first] <= array[left_index_second]) {
        // Если эл-т левого подмассива меньше или равен эл-ту правого, то
берем его и двигаем границу l
            merged_array[i] = array[left_index_first++];
        } else {
        // Если эл-т правого массива меньше эл-та левого, то берем его и
двигаем границу r
            merged_array[i] = array[left_index_second++];
        }
    }
    for (int i = 0; i < right_index_second; ++i) {
    // Записываем результат слияния в исходный подмассив
        array[i + array_begin] = merged_array[i];
    }
}

void MergeSort(int* array, int left_index, int right_index) {
    if (right_index - left_index == 1) {
    // Массив из одного элемента уже отсортирован, поэтому мы выходим из
рекурсии
        return;
    }
    int mid_index = (right_index + left_index) / 2;
    MergeSort(array, left_index, mid_index); // Рекурсивно сортируем левую
половину
    MergeSort(array, mid_index, right_index); // Рекурсивно сортируем правую
половину
    Merge(array, left_index, mid_index, mid_index, right_index); // Сливаем
```

```
воедино
}
```

Характеристика

Временная сложность — $O(n \log n)$
Доказательство: поскольку после каждого рекурсивного запуска размер подмассива уменьшается вдвое, то глубина рекурсии не будет превышать $\log_2 n$. На каждом её уровне `merge` рассматривает каждый из n элементов исходного массива, причем ровно единожды, так как сливаемые подмассивы не пересекаются.
Таким образом, суммарно совершается порядка $n \log_2 n$ или, асимптотически, $O(n \log n)$ действий.
Сложность по памяти — $O(n)$, поскольку мы дополнительно объявляем только один массив размера n.
Устойчивость: можно написать как устойчивую, так и неустойчивую реализацию. Мы в праве самостоятельно задать механизм обработки равных значений.

Быстрая сортировка

Теория

Инвариант: после каждого разбиения все элементы слева от опорного не больше него, справа — не меньше него.

Идея: выберем произвольный элемент массива (назовём его *опорным* или *pivot*), после чего разделим массив на два подмассива так, чтобы в первом все элементы были не больше опорного, а во втором — не меньше. Если мы сможем это сделать, то опорный элемент по определению займёт свою конечную позицию, а слева и справа от него образуются два независимых подмассива, каким-либо образом отсортировав которые, мы отсортируем и массив целиком.

Описанный выше процесс называется *разбиением* и является ключевой частью алгоритма.

Как уже было сказано, после того, как прошло разбиение, мы хотим отсортировать два получившихся подмассива. Предлагается делать это рекурсивно с помощью всё того же алгоритма, повторяя процедуру до тех пор, пока мы не получим подмассив из одного элемента.

Рассмотрим несколько основных способов реализовать разбиение.

1. **Разбиение Ломуто**[1]
   В качестве опорного элемента выберем последний элемент массива.
   Заведём два указателя $i = j = 0$, где i просматривает массив, а j отвечает за размер первого подмассива, который мы будем расширять по ходу итерации. Сразу заметим, что по такому определению на каждой итерации $i \geq j$, поскольку мы не можем добавить в подмассив больше элементов, чем мы просмотрели.
   Начнём итерироваться по массиву. Для любого a_i возможны два варианта:

   1. $a_i > pivot$ — тогда поскольку $i \geq j$, то a_i уже находится в своём подмассиве, и ничего делать не нужно;
   2. $a_i < pivot$ — тогда поскольку $i \geq j$, то a_i находится не в своём подмассиве. Возьмём первый элемент второго подмассива, т.е. a_j, и поменяем его местами с a_i. Тогда a_j всё ещё будет принадлежать второму подмассиву, а a_i перейдёт в первый, размер которого увеличится на один элемент, т.е. мы должны также увеличить значения указателя j на один.

   Теперь для поддержания инварианта нам необходимо также поместить опорный элемент на границе двух подмассивов, то есть поменять его местами с a_j.
2. **Разбиение Хоара** [2]
   В качестве опорного элемента выберем произвольный (обычно центральный) элемент массива.
   Заведём два указателя $i = l$ и $j = r$ — левую границу второго и правую границу первого подмассивов соответственно. Будем сдвигать их к центру до тех пор, пока не найдем такую пару элементов $a_i \geq pivot, b_j \leq pivot$, перемена местами которых либо не скажется на выполнении инварианта, либо является обязательной для этого.
   Если при этом разбиение не завершено, т.е. $[l, j] \cap [i, r] \neq \emptyset$, то мы обязаны поменять их местами и, поскольку данные элементы уже стоят на своих местах, сдвинуть соответствующие индексы.
   Будем повторять эту процедуру, пока разбиение не завершится, т.е. пока $i \leq j$.
3. **Бельгийское разбиение**
   Изучение данной концепции выходит за рамки нашего курса, однако читателям предлагается самостоятельно ознакомиться с ней по материалам всемирной сети Интернет.

Реализация

Важное замечание: в отличие от большинства рекурсивых алгоритмов (например, сортировки слиянием), быстрая сортировка традиционно работает не с полуинтервалами, а с отрезками. Это важно учесть при реализации.

Разбиение Ломуто

```
void QuickSortLomuto(int* array, int left_index, int right_index) {
// Передаём подмассив и его границы [left_index; right_index]
    if (left_index >= right_index) return;
    // Если left_index > right_index, то это некорректный набор входных
данных
    // Если left_index == right_index, то это массив из одного элемента, с
которым нам ничего не надо делать
    int pivot = array[right_index];
    // Выбираем опорный элемент
    int j = left_index;
    // Задаём указатель j на границу правого подмассива
    for (int i = left_index; i < right_index; ++i) {
    // Перебираем все элементы массива
        if (array[i] > pivot) continue;
        // Если текущий элемент больше опорного, то нам ничего не надо
делать
        std::swap(array[i], array[j]);
        j++;
        // Если текущий элемент меньше опорного, то мы добавляем его в левый
подмассив и увеличиваем размер этого подмассива
    }
    std::swap(array[right_index], array[j]);
    // Ставим опорный элемент на место
    QuickSortLomuto(array, left_index, j - 1);
    QuickSortLomuto(array, j + 1, right_index);
    // Рекурсивно сортируем подмассивы
}
```

Разбиение Хоара

```
void QuickSortHoare(int* array, int left_index, int right_index) {
// Передаём подмассив и его границы [left_index; right_index]
    if (left_index >= right_index) return;
    // Если left_index > right_index, то это некорректный набор входных
данных
    // Если left_index == right_index, то это массив из одного элемента, с
которым нам ничего не надо делать
    int pivot = a[(left_index + right_index) / 2];
    // Выбираем опорный элемент
    int i = left_index, j = right_index;
    // Объявляем указатели на границы подмассивов
    while (i <= j) {
    // Пока подмассивы пересекаются
        while (array[i] < pivot) i++;
        // Пока array[i] не является кандидатом на swap, сдвигаем i к центру
```

```
        while (array[j] > pivot) j--;
        // Пока array[j] не является кандидатом на swap, сдвигаем j к центру
        if (i <= j) {
        // Если подмассивы пересекаются
            std::swap(array[i], array[j]);
            i++;
            j--;
            // Меняем эл-ты местами и сдвигаем соответствующие указатели
        }
    }
    QuickSortHoare(array, left_index, j);
    QuickSortHoare(array, i, right_index);
    // Рекурсивно сортируем подмассивы
}
```

Характеристика

Временная сложность — в среднем $O(n \log n)$
Доказательство этого факта оставим в качестве упражнения читателю.
Стоит также заметить, что разбиение Хоара в среднем кратно быстрее разбиения Ломуто. Существуют модификации разбиения Ломуто[3], для которых это утверждение неверно, но их изучение выходит за рамки нашего курса.

Сложность по памяти — в среднем $O(\log n)$
Доказательство этого факта оставим в качестве упражнения читателю.

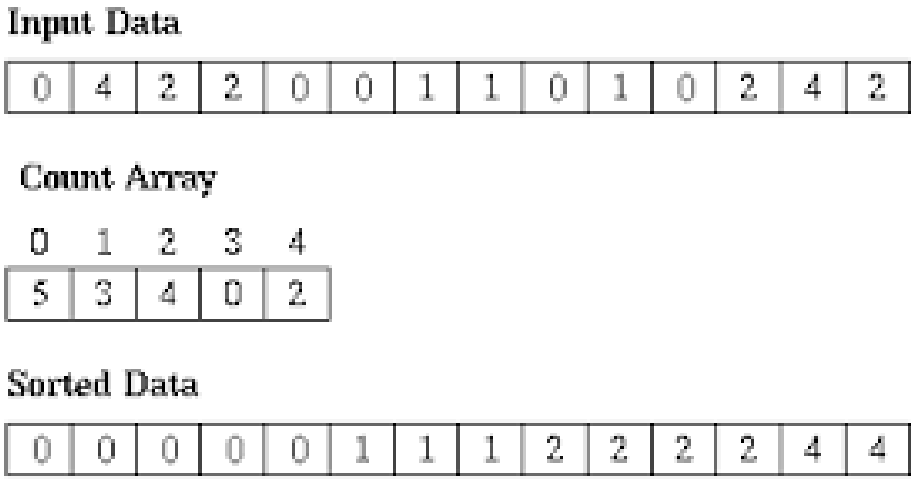
Устойчивость: сортировка не устойчива ни в какой из реализаций — равные элементы всегда будут переставляться произвольным образом. Мы не можем гарантировать на них никакого порядка.

Сортировка подсчётом

Теория

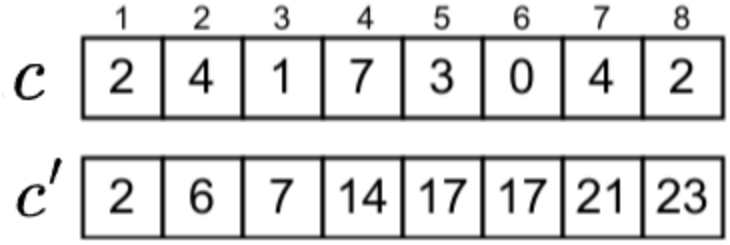
Идея: заметим удивительный факт: последовательность из индексов массива — это отсортированная последовательность.
Заведём массив c, где значение c_i равняется количеству вхождений числа i в исходный массив. Заведём ещё один массив, куда каждое число i будем записывать c_i раз. Очевидно, что если перебирать i в порядке возрастания, то все числа в новом массиве будут записаны в порядке неубывания (не возрастания, ибо если были равные, то мы запишем и их). Очевидно также, что если мы корректно построили c, то каждое число из

исходного массива обязательно встретится в новом.
Заметим, что мы только что отсортировали массив.



У этой идеи две проблемы.
Во-первых, таким образом мы не можем сортировать ничего, кроме целых чисел, поскольку более сложные объекты нельзя так просто перезаписать или использовать в качестве индекса.
Во-вторых, она не является устойчивой, поскольку мы буквально создаём новые объекты, порядок которых никак не связан с порядком исходных.

Чтобы исправить это, мы хотим как-то использовать c_i для того, чтобы определять, на каком месте в новом массиве будет лежать каждый элемент из исходного.
Построим *префиксные суммы* на массиве c, т.е. такой массив c', что $c'_i = c'_{i-1} + c_i$:



Тогда c'_i можно воспринимать как последнюю свободную позицию, которую может занять элемент i в итоговом массиве. Каждый раз будем ставить элемент a_i на позицию c'_{a_i} и уменьшать этот индекс на 1, поскольку в процессе мы занимаем одно свободное место.

Если теперь перебирать i в порядке возрастания, то порядок равных элементов в итоговом массиве инвертируется, поскольку первый обработанный элемент всегда занимает последнюю свободную позицию. Если же перебирать i в порядке убывания, то последний из равных элементов встанет на последнюю позицию, предпоследний — на предпоследнюю и так далее, т.е. мы сохраним исходный порядок.

Реализация

```
void CountingSort(int *array, int array_size) {
// Передаём массив и размер
    int count[kMaxNumber];
    // Заводим массив подсчёта
    for (int i = 0; i < kMaxNumber; ++i) count[i] = 0;
    // Зануляем массив подсчёта
    for (int i = 0; i < array_size; ++i) count[array[i]]++;
    // Считаем элементы сортируемого массива
    for (int i = 1; i < kMaxNumber; ++i) count[i] = count[i] + count[i - 1];
    // Строим префиксные суммы (не создаём новый массив, а перезаписываем
старый)
    int sorted_array[kArraySize];
    // Создаём массив для отсортированных элементов
    for (int i = array_size - 1; i >= 0; --i) {
    // Проходимся по всем элементам массива, начиная с конца
        sorted_array[count[array[i]] - 1] = array[i];
        // Ставим текущий элемент на нужное место и не забываем перевести
индексы в 0-нумерацию (в массиве с они в 1-нумерации)
        count[array[i]]--;
        // Переходим к следующей свободной позиции
    }
    for (int i = 0; i < array_size; ++i) array[i] = sorted_array[i];
    // Записываем отсортированный массив в исходный
}
```

Характеристика

Временная сложность — $O(n + k)$

Сложность по памяти — $O(n + k)$

Оба этих факта легко заметить из реализации, поскольку алгоритм не совершает практически никаких действий, кроме итерирования по цилку и объявления массивов.

Устойчивость: приведенная реализация является устойчивой. Это было доказано выше.

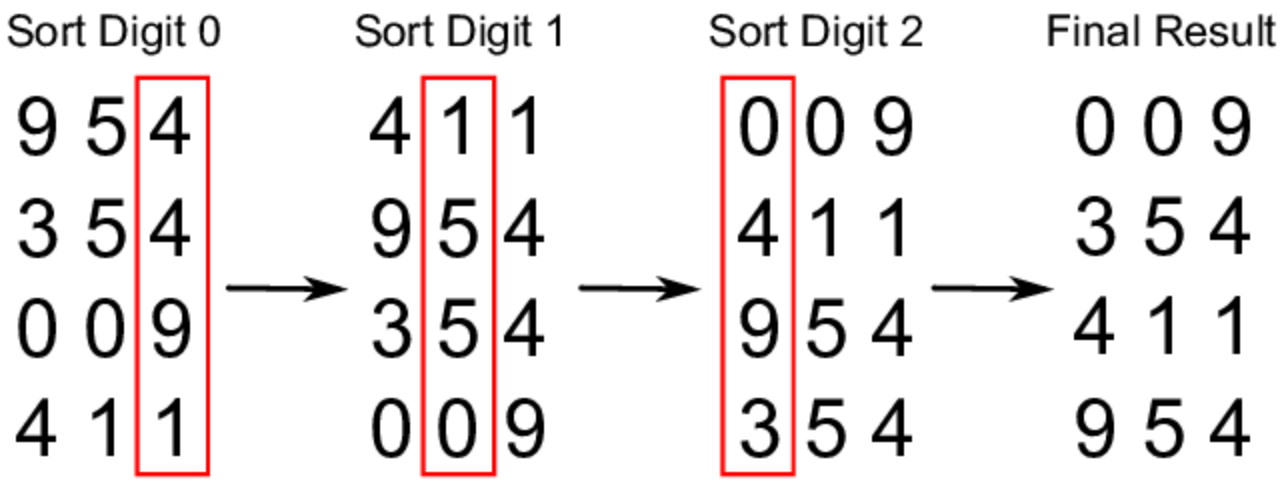
Поразрядная сортировка

Пусть мы хотим отсортировать массив последовательностей одинаковой длины, состоящих из сравнимых элементов (будем называть их *разрядами*), в

лексикографическом порядке. Рассмотрим два подхода, которые позволяют решить эту задачу.

LSD-сортировка (*Less Significant Digit*)

Будем рассматривать разряды в порядке от младшего к старшему и сортировать последовательности по значениям текущего разряда какой-либо устойчивой сортировкой. Условимся, что все последовательности в массиве имеют одинаковую длину k (если нет, то можно записать в начало меньших последовательностей разряды с нулевым значением):

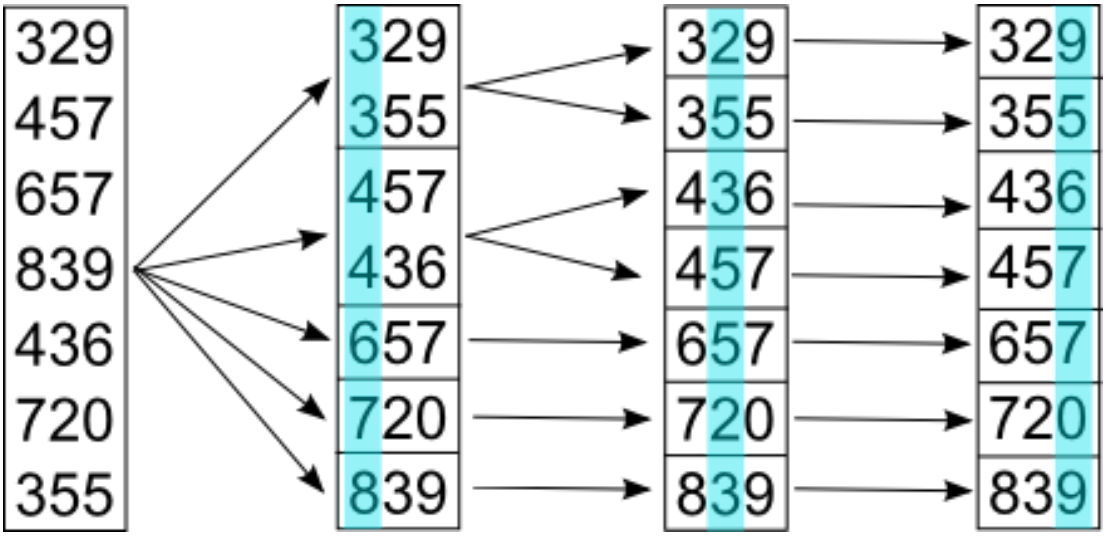


Инвариант. в конце i-й итерации алгоритма суффиксы длины i всех последовательностей отсортированы друг относительно друга.

Покажем, что инвариант действительно выполняется.
При $i = 1$ это очевидно: последний разряд — единственный, который мы отсортировали, причем мы сделали это по его же значениям.
Пусть инвариант выполняется для $i = n$. Отсортируем последовательности по $n + 1$-му разряду и посмотрим, в каком порядке они располагаются.
Если значения $n + 1$-го разряда какой-то пары последовательностей различаются, то соответствующие суффиксы будут располагаться в правильном порядке просто потому, что мы отсортировали их по наибольшему разряду.
Если значения $n + 1$-го разряда какой-то пары последовательностей совпадают, то для того, чтобы соответствующие суффиксы были расположены в правильном относительном порядке, нам необходимо, чтобы в этом же порядке располагались суффиксы этих последовательностей длины n. Заметим, что это гарантируется двумя вещами: предположением индукции и устойчивостью выбранной сортировки, которая позволяет сохранить порядок элементов, полученный в результате предыдущей итерации, т.е. не разрушить предположение индукции в ходе работы алгоритма.
Таким образом, при $i = n + 1$ инвариант также выполняется, что и требовалось доказать.

MSD-сортировка (*Most Significant Digit*)

Будем рассматривать разряды в порядке от старшего к младшему и сортировать последовательности по значениям текущего разряда какой-либо устойчивой сортировкой. Для этого нам необходимо, чтобы все последовательности в массиве имели одинаковую длину (если нет, то нужно записать в начало меньших последовательностей разряды с нулевым значением):



Инвариант: в конце i-й итерации алгоритма префиксы длины i всех последовательностей отсортированы друг относительно друга.
Корректность этой сортировки можно доказать, например, аналогичным приведенному выше способом.

Реализация

В качестве устойчивой сортировки выберем *сортировку подсчётом*. Тогда, например, сортировку чисел при помощи LSD можно реализовать следующим образом:

```
int count[kBucketSize];
int sorted_array[kArraySize];
// Объявляем массивы глобально для экономии памяти

void PrecalcDigitMaximums() {
// Для получения значений каждого разряда нам нужно знать значение base^k.
Поскольку оно одинаковое для всех чисел, предподсчитаем его
    powers[0] = 1;
    for (int i = 1; i < kMaxDigitsAmount; ++i) {
        powers[i] = powers[i - 1] * kBucketSize;
    }
}
```

```
int GetDigit(int x, int digit) {
// Получаем digit-й разряд числа x по основанию kBucketSize
    return (x / powers[digit]) % kBucketSize;
}

void CountingSortDigits(int *array, int array_size, int digit) {
// Сортируем числа из array по значению digit-того разряда
    for (int i = 0; i < kBucketSize; ++i) {
    // Обнуляем массив подсчёта
        count[i] = 0;
    }
    for (int i = 0; i < array_size; ++i) {
    // Считаем количество соответствующих разрядов в исходном массиве
        count[GetDigit(array[i], digit)]++;
    }
    for (int i = 1; i < kBucketSize; ++i) {
    // Строим префиксные суммы
        count[i] = count[i - 1] + count[i];
    }
    for (int i = array_size - 1; i >= 0; --i) {
        int d = GetDigit(array[i], digit);
        // Запоминаем значение нужного разряда, по которому будем сравнивать
        sorted_array[count[d] - 1] = array[i];
        // Ставим текущий элемент на нужное место и не забываем перевести
индексы в 0-нумерацию (в массиве с они в 1-нумерации)
        count[d] -= 1;
        // Переходим к следующей свободной позиции
    }
    for (int i = 0; i < array_size; ++i) array[i] = sorted_array[i];
    // Записываем отсортированный массив в исходный
}

void RadixSortLSD(int *array, int array_size) {
    int max_element = array[0];
    // Инициализируем переменную под поиск максимума (нам нужно знать макс.
количество разрядов)
    for (int i = 0; i < array_size; ++i) {
    // Определяем максимальный элемент в массиве
        max_element = std::max(max_element, array[i]);
    }
    int i = 1, max_digits_count = 0;
    // i – вспомогательная переменная для определения числа разрядов
max_digits_count
    while (i < max_element) {
    // Пока наибольший элемент превосходит kBucketSize^max_digits_count,
увеличиваем счётчик разрядов
```

```
        i *= kBucketSize;
        max_digits_count++;
    }
    for (i = 0; i < max_digits_count; ++i) {
    // Сортируем элементы по разрядам, начиная с младшего
        CountingSortDigits(array, array_size, i);
    }
}
```

Реализация MSD реализуется аналогично с точностью до порядка выбора разрядов для сортировки.

Характеристика

Временная сложность — $O(k(n+m))$
Сложность по памяти — $O(n+m)$, где:

- k — количество разрядов
- n — количество элементов
- m — максимальное значение одного разряда

Оба этих факта легко заметить из реализации аналогично тому, как это было сделано в сортировке подсчётом.

Устойчивость: поразрядная сортировка устойчива. Поскольку для сортировки каждого из разрядов мы также используем устойчивую сортировку, то ни на одном из шагов алгоритма равные элементы не поменяются местами.

---

1. Для визуализации разбиения Ломуто см. https://habrastorage.org/webt/dh/9t/e1/dh9te1tgswnrpegjr_pwao3vlpi.gif ↩
2. Для визуализации разбиения Хоара см. https://upload.wikimedia.org/wikipedia/commons/9/9c/Quicksort-example.gif ↩
3. Статья на тему эффективности разбиения Ломуто: https://habr.com/ru/articles/512106/ ↩