**Сounting Sort**

**Сортировка подсчётом** (англ. *counting sort*) — алгоритм сортировки целых чисел в диапазоне от 0 до некоторой константы kk или сложных объектов, работающий за линейное время.

Исходная последовательность из n структур хранится в массиве A, а отсортированная — в массиве B того же размера. Кроме того, используется вспомогательный массив P с индексами от 0 до k−1. Идея алгоритма состоит в предварительном подсчете количества элементов с различными ключами в исходном массиве и разделении результирующего массива на части соответствующей длины (будем называть их блоками). Затем при повторном проходе исходного массива каждый его элемент копируется в специально отведенный его ключу блок, в первую свободную ячейку. Это осуществляется с помощью массива индексов P, в котором хранятся индексы начала блоков для различных ключей. P[key] — индекс в результирующем массиве, соответствующий первому элементу блока для ключа key.

Анализ **Сounting Sort.**

void SortCounting(vector <int> main, int y) { // Сортировка подсчетом

vector <int> sup;

for (int i = 0; i <= y; i++)

sup.push\_back(0);

for (int i = 0; i < main.size(); i++)

sup[main[i]]++;

}

В данном алгоритме используются дополнительный массив **sup** с размерность **y** (максимального элемента массива),в который мы сначала загружаем одни нули, а после этого подсчитываем значения.

for (int i = 0; i <= y; i++) - цикл , который заполняет по очереди каждый элемент массива нулями. Время выполнения растёт линейно - чем больше элементов, тем дольше выполнение (y — количество элементов. При y = 10 программа будет в два раза дольше выполняться, чем при y = 5). Получается, что цикл кода выше работает за линейное время (n) следовательно выполняться за O(n).

for (int i = 0; i < main.size(); i++) выполняется столько раз, сколько находится элементов в массиве **main**. С помощью этого цикла происходит подсчет элементов из исходного массива в вспомогательный массив. Так же проходится по элементам (n элементам) поэтому время выполнения тоже равно O(n). Подведя итоги, мы видим, что общее время выполнения равняется O(n). Данный алгоритм является устойчивым.

**Radix sort**

Поразрядная сортировка (*radix sort*) — алгоритм сортировки, который выполняется за линейное время. Пусть имеется n d-значных чисел, в которых каждая цифра принимает одно из k возможных значений. Тогда алгоритм RADIX-SORT позволяет выполнить корректную сортировку этих чисел за время O(d(n + k)), если устойчивая сортировка, используемая данным алгоритмом, имеет время работы O( n + k).

Корректность поразрядной сортировки доказывается методом математической индукции по сортируемым столбцам. Анализ времени работы рассматриваемого алгоритма зависит от того, какой из методов устойчивой сортировки используется в качестве промежуточного алгоритма сор­тировки. Если каждая цифра принадлежит интервалу от 0 до k - 1 (и, таким образом, может принимать k возможных значений) и k не слишком велико, то оптимальным выбором является сортировка подсчетом. Для обработки каждой из d цифр n чисел понадобится время O(n + k), а все цифры будут обработаны алгоритмом поразрядной сортировки в течение времени O(d(n + k)).

Пусть имеется n b-битовых чисел и произвольное натуральное число r <= b. Алгоритм RADIX-SORT позволяет выполнить корректную сортировку этих чисел за время O((b/r)(n + 2r)), если используемая им устойчивая сортировка имеет время работы O( n + k) для входных данных в диапазоне от 0 до k.

Для значения r <= b каждый ключ можно рассматривать как число, состоящее из d = [b/r] цифр по r бит. Все цифры представляют собой целые числа в интервале от 0 до 2r - 1, поэтому можно воспользоваться алгоритмом сортировки подсчетом, в котором k = 2r - 1 (например, 32-битовое слово можно рассматривать как число, состоящее из четырех 8-битовых цифр, так что b = 32, r = 8, k = 2r - 1 = 255, а d = b/r = 4). Каждый проход сортировки подсчетом занимает время O(n + k) = O(n + 2r), а всего выполняется d проходов, так что полное время работы алгоритма равно O(d(n + 2r)) = O((b/r)(n + 2r)).

Выберем для двух заданных значений n и b такую величину r <= b, которая бы минимизировала выражение ((b/r)(n + 2r)). Если b <|lg(n)| , то для любого значения r <= b имеем ( n + 2r) = O(n). Таким образом, выбор r = b приводит к асимптотически оптимальному времени работы (b/b)(n + 2b) = O(n). Если же b >= lg(n)(округление вниз), то наилучшее время с точностью до постоянного множителя можно получить, выбрав r = lg(n) (округление вниз). Это можно понять из следующих рассуждений. Ес­ли выбрать r = lg(n) (округление вниз), то время работы алгоритма будет равно O(bn/lg(n)). Если r увеличивается и превышает значение lg(n) (округление вниз), то член 2r в числителе возрастает быстрее, чем член r в знаменателе, так что увеличение r свыше lg(n) (округление вниз) приводит ко времени работы алгоритма, равному Ὠ( bn / lg(n) ). Если же величина r уменьшается и становится меньше lg(n) (округление вниз), то член b / r возрастает, а множитель n + 2r остается величиной порядка O(n).

void SortRadix(vector <int> main, int y) { // Поразрядная сортировка

vector <int> sup1 = main;

vector <vector<int>> sup2;

int c = 1, m = 10;

for (int i = 0; i < 10; i++)

sup2.push\_back(vector<int>());

for (int i = 0; i < 3; i++) {

for (int a = 0; a < main.size(); a++)

sup2[(sup1[a] % m) / c].push\_back(sup1[a]);

sup1.clear();

for (int a = 0; a < 10; a++)

for (int j = 0; j < sup2[a].size(); j++)

sup1.push\_back(sup2[a][j]);

for (int i = 0; i < sup2.size(); i++)

sup2[i].clear();

m \*= 10;

c \*= 10;

}

}

В ходе алгоритма, у каждого числа массива последовательно обрабатываются отдельные разряды справа-налево. Так, например, для числа 246 будут обработаны цифры 6, 4 и 2. Число будет перемещаться между пронумерованными Блоками в соответствии со значением своих разрядов. Далее будем полагать, что у нас уже есть функция Разряд(N, I), возвращающая I-тую цифру числа N справа, т.е. Разряд(246, 1) = 6, Разряд(246, 2) = 4, Разряд(246, 3) = 2, Разряд(246, 4) = 0. В последнем примере возвращается «ведущий ноль», т.е. от 0246.

for (int i = 0; i < 10; i++)

* Цикл, в котором инициализируем двумерный векторный массив

for (int i = 0; i < 3; i++) – цикл, в котором вытаскиваем каждый разряд числа. Проходится по вектору а ( n элементам ) значит время работы O(n) внутри цикла операции выполняются за O(1).

for (int a = 0; a < main.size(); a++)

Цикл, в котором сортируем выбранный разряд в свой столбец двумерного массива.

Следующие два цикла вспомогательные и используются для преобразования двумерного массива в одномерный. Последний цикл так же вспомогательный. Он очищает двумерный массив после выгрузки.

Время работы O(n). Общее время выполнения O(n). Алгоритм данной поразрядной сортировки (LSD) является неустойчивым.