**Radix Sort**

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).

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).

Рассмотрим

Сортировка подсчетом превосходит нижнюю границу Ὠ ( n\*lg(n)), поскольку не является сортировкой сравнением. Фактически нигде в коде не производится сравнение входных элементов.

1.vector<int> RadixSort(vector <int> A)//поразрядная сортировка

2.{

3. vector <vector<int>> sup;//вектор

4. int c = 1, m = 10;//

5.

6. for (int i = 0; i < 10; i++)//делаем копию массива

7. sup.push\_back(vector<int>());//заносим в массив

8.

9. for (int i = 0; i < 2; i++)//тут ходим по разрядам ( 1 и 2 разряд) тк у нас рандом 10. до 100 то есть 1-99 то смотрим первый и второй разряд

11. {

12. for (int j = 0; j < A.size(); j++)

13. sup[(A[j] % m) / c].push\_back(A[j]);//заносим

14.

15. A.clear();//очищаем

16.

17.

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

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

20. A.push\_back(sup[a][j]);

21.

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

23. sup[i].clear();

24.

25. m \*= 10;

26. c \*= 10;

27. }

28. return A;//возвращаемся

29. }

В строке 6 используется цикл который делает всего 10 проходов с операцией сложения, поэтому время его работы учитывать не будем. Далее на 9 строке идет цикл с 2 проходами, его мы тоже не учитываем. Так же внутри него есть цикл на 12 строке, который идет по разрядам, а количество разрядов ничтожно мало при большом значении n, поэтому общее время выполнения будет составлять O(n).На 13 строке мы заносим в копию массива наши числа, затем очищаем массив для новых чисел. Рассмотрим цикл на 18 строке, здесь он делает 10 проходов, его время мы не учитываем. После проверки всех разрядов мы дольжны вывести все наши числа в один массив.

**Сounting Sort**

В данной сортировке, в сортировке подсчетом алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих Предположим, что входной массив состоит из {n} целых чисел в диапазоне от {0} до { k-1}, где { k -> {N} }. Далее алгоритм будет обобщён для произвольного целочисленного диапазона. Существует несколько модификаций сортировки подсчётом, ниже рассмотрены три линейных и одна квадратичная, которая использует другой подход, но имеет то же название.

Сортировка подсчетом превосходит нижнюю границу Ὠ ( n\*lg(n)), поскольку не является сортировкой сравнением. Фактически нигде в коде не производится сравнение входных элементов. Вместо этого непосредственно используются их значения, с помощью которых элементам сопоставляются конкретные индексы. Нижняя граница Ὠ(n\*lg(n)) для сортировки сравнением неприменима при отказе от модели сортировки, использующей сравнения.

Это простейший вариант алгоритма. Создать вспомогательный массив C[0..k - 1], состоящий из нулей, затем последовательно прочитать элементы входного массива A, для каждого A[i] найти максимальный элемент и заполнить массив данными элементами.

1. vector<int> CountingSort(vector <int> A)//сортировка подсчетом

2. {

3. vector <int> j;//вектора

4. int ma = A[0];//

5.

6. for (int i = 1; i < A.size(); i++)//делаем копию

7. if (ma < A[i])//ищим макс элемент

8. ma = A[i];//окей это максимальный элемент

9.

10. for (int i = 0; i < ma + 1; i++)

11.

12. j.push\_back(0);//заполняем

13.

14. for (int i = 0; i < A.size(); i++)//ходим бродим

15. j[A[i]]++;//прибавляем + 1

16.

17. A.clear();//очищаем

18. for (int i = 0; i < j.size(); i++)//создаем второй массив двумерный так скажем ибо 19. то был одномерный а это вот такой

20. if (j[i] != 0)//если он не = 0 то

21. for (int a = 0; a < j[i]; a++)//заполняем

22. A.push\_back(i);//заносим элементы в наш массив

23. return A;//возвращаем

24. }

Давайте пошагово рассмотрим наш код. Для начала мы создаем наш вектор размерностью n. Для удобства делаем копию нашего массива, в которой будем хранить временные числа. Цикл на 7 строке выполняется за О(n), так как он бежит по всем элементам массива. Далее, если элемент такой имеется, то он автоматически становится максимальным элементом. Этот цикл проходит до тех пор, пока в нашем массиве не кончатся наши элементы. На 10 строке мы создаем массив размерностью нашего максимального элемента а заполняем его соответственно. На 18 строке мы делаем всю ту же процедуру для двумерного массива. После чего мы делаем проверку на то, является ли элемент равным 0 как показано на 20 строке. Если все отлично, то мы заполняем наш массив размерностью A.

Важное свойство алгоритма сортировки подсчетом заключается в его устой­чивости (stable): элементы с одним и тем же значением находятся в выходном массиве в том же порядке, что и во входном. Обычно свойство устойчивости важно только в ситуации, когда вместе сортируемые элементы имеют сопутствующие данные.