Министерство образования Российской Федерации

Тюменский Государственный Университет

Факультет Математики и Компьютерных Наук

# Распределяющая поразрядная сортировка Змановского.

Конкурсная работа выполнена:

студентом 384 группы

Змановским П. П.

Научный руководитель:

Деревнина А. Ю.

Тюмень

2001

## *Содержание*

## ***Введение.***

В программировании очень часто возникает вопрос переразмещения элементов в порядке возрастания или убывания. От порядка, в котором хранятся элементы в памяти компьютера, во многом зависит скорость выполнения и простота алгоритмов, предназначенных для их обработки.

Хотя в словарях слово "сортировка" определяется как процесс разделения объектов по виду или сорту, программисты традиционно используют его в гораздо более узком смысле, обозначая им такую перестановку предметов, при которой они располагаются в порядке возрастания или убывания. Такой процесс следовало бы называть не сортировкой, а упорядочением, но использование этого слова привело бы к путанице из-за перегруженности значениями слова "порядок". Поэтому слово "сортировка" используется в качестве обозначения для процесса упорядочения.

По оценкам производителей компьютеров в среднем более четверти машинного времени тратится на сортировку. Во многих вычислительных системах на нее уходит больше половины машинного времени. Исходя из этих статистических данных, можно заключить, что либо сортировка имеет много важных применений, либо ею часто пользуются без нужды, либо применяются в основном неэффективные алгоритмы сортировки. По-видимому, каждое из приведенных предположений содержит долю истины. В любом случае ясно, что сортировка заслуживает серьезного изучения с точки зрения ее практического использования.

Вот некоторые из наиболее важных областей применения сортировки:

1. *Решение задачи группирования*, когда нужно собрать вместе все элементы с одинаковыми значениями некоторого признака. Допустим, имеется 10 000 элементов, расположенных в

случайном порядке, причем значения многих из них равны и нужно переупорядочить массив так, чтобы элементы с равными значениями занимали соседние позиции в массиве. Это, по существу, тоже задача "сортировки", но в более широком смысле, и она легко может быть решена путем сортировки массива в указанном выше узком смысле, а именно - в результате расположения элементов в порядке неубывания $v_1 \le v_2 \le \ldots \le v_{10000}$. Эффект, который может быть достигнут после выполнения этой процедуры, и объясняет изменение первоначального смысла слова "сортировка".

2. *Поиск общих элементов в двух или более массивах.* Если два или более массивов рассортировать в одном и том же порядке, то можно отыскать в них все общие элементы за один последовательный просмотр всех массивов без возвратов. Таким образом, сортировка позволяет использовать

последовательный доступ к большим массивам в качестве приемлемой замены прямой адресации, что имеет очень большое значение при работе с внешними источниками данных, скорость чтения информации которых значительно уступает скорости чтения информации из оперативной памяти.

3. *Поиск информации по значениям ключей.* Сортировка также используется при поиске какого-либо рода данных в массивах по значениям ключей. Если массив отсортирован, то, применяя специальные методы (например, алгоритм Бинарного поиска), можно значительно уменьшить время нахождения необходимой информации.

Алгоритмы сортировки представляют собой интересный *частный пример* того, как следует подходить к решению проблем программирования вообще. Обобщение частных методов позволяет в значительной степени овладеть теми подходами, которые помогают

создавать качественные алгоритмы для решения других проблем, связанных с использованием компьютеров.

Методы сортировки служат великолепной иллюстрацией базовых концепций *анализа алгоритмов*, т.е. оценки качества алгоритмов, что, в свою очередь, позволяет разумно делать выбор среди, казалось бы, равноценных методов.

## *Задача сортировки массива*

Для того, чтобы описывать различные алгоритмы сортировки, необходимо более четко сформулировать задачу и ввести соответствующую терминологию. Пусть надо упорядочить N элементов

$$R_1, R_2, \ldots, R_N .$$

Назовем их *записями*, а всю совокупность N записей назовем *файлом*. Каждая запись $R_j$ имеет *ключ*, $K_j$, который и управляет процессом сортировки. Помимо ключа, запись может содержать

дополнительную "сопутствующую информацию", которая не влияет на сортировку, но всегда остается в этой записи.

Отношение порядка "<" на множестве ключей вводится таким образом, чтобы для любых трех значений ключей a, b, c выполнялись следующие условия:

справедливо одно и только одно из соотношений a < b, a = b, b < a (закон трихотомии);

если a < b и b < c, то a < c (закон транзитивности).

Эти два свойства определяют математическое понятие *линейного упорядочения*, называемого также *совершенным упорядочением.* Любое множество с отношением "<", удовлетворяющим обоим этим свойствам, поддается сортировке большинством известных методов, хотя некоторые предназначены только для числовых и буквенных ключей с общепринятым отношением порядка.

Задача сортировки - найти такую перестановку записей p(1) p(2) … p(N) с индексами {1, 2, …, N}, после которой ключи расположились бы в порядке неубывания:

$$K_{p(1)} \le K_{p(2)} \le \ldots \le K_{p(N)}.$$

Сортировка называется *устойчивой*, если она удовлетворяет такому дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке, т.е., другими словами,

$p(i) < p(j)$ для любых $K_{p(i)} = K_{p(j)}$ и $i < j$.

В одних случаях приходится физически перемещать записи в памяти так, чтобы их ключи были упорядочены; в других случаях достаточно создать вспомогательную таблицу, которая некоторым образом описывает перестановку и обеспечивает доступ к записям в соответствии с порядком их ключей.

Некоторые методы сортировки предполагают существование величин "$\infty$" и "$-\infty$" или одной из них. Величина "$\infty$" считается больше, а величина "$-\infty$" меньше любого ключа:

$-\infty < K_j < \infty$ для $1 \le j \le N$.

Эти величины используются в качестве искусственных ключей, а также граничных признаков. Равенство ключей с одним из граничных элементов, вообще говоря, исключено. Если же оно, тем не менее, допускается, алгоритмы можно модифицировать так, чтобы они все-таки работали, хотя нередко при этом их изящество и эффективность отчасти утрачиваются.

Обычно методы сортировки подразделяют на два класса: внутренние, когда все записи хранятся в быстрой оперативной памяти, и внешние, когда все записи в ней не помещаются. Методы внутренней сортировки обеспечивают большую гибкость при построении структур

данных и доступа к ним, внешние же методы обеспечивают достижение нужного результата в условиях ограниченных ресурсов.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время пропорционально N log N; при этом требуется около log N "проходов" по данным. (На самом деле, когда N неограниченно возрастает, время сортировки растет как

N (log N)^2, если все ключи различны, поскольку размеры ключей также увеличиваются с ростом N).

С другой стороны, если известно, что ключи являются случайными величинами с некоторым непрерывным распределением, то сортировка может быть выполнена в среднем за O(N) шагов.

Было изобретено множество различных методов сортировки. Каждый метод имеет свои преимущества и недостатки, поэтому он оказывается эффективнее других при некоторых конфигурациях данных и аппаратуры. К сожалению, неизвестен наилучший способ

сортировки (если он вообще существует); имеется *много* наилучших методов, но только в случаях, когда известно, что сортируется, на каком компьютере и с какой целью. Поэтому полезно изучить характеристики каждого метода, чтобы для конкретного случая можно было сделать разумный выбор.

Большой интерес для изучения представляет *внутренняя сортировка*, когда число записей, подлежащих сортировке, достаточно мало, так что весь процесс можно провести в оперативной памяти компьютера, обладающей высоким быстродействием.

## ***Анализ алгоритмов внутренней сортировки.***

Существует огромное количество разнообразных алгоритмов внутренней сортировки. Большинство из них можно разделить на пять основных групп:

1. Сортировка путем вставок (элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов),
2. Обменная сортировка (если два элемента расположены не по порядку, то они меняются местами. Этот процесс повторяется до тех пор, пока элементы не будут упорядочены),
3. Сортировка посредством выбора (сначала выделяется наименьший (или наибольший) элемент и каким-либо образом отделяется от остальных, затем выбирается наименьший (наибольший) из оставшихся и т. д.),
4. Сортировка методом слияния (объединение двух или более упорядоченных массивов в один упорядоченный массив),
5. Сортировка методом распределения (основана на предварительном подсчете частот повторения ключей и

последующем перемещении записей в зависимости от этих частот).

Несмотря на то, что практически каждый из известных алгоритмов обладает определенными достоинствами, существуют методы сортировки, которые являются наиболее быстрыми и наиболее универсальными в большинстве случаев применения. Среди таких алгоритмов самыми прекрасными характеристиками обладают Быстрая сортировка (QuickSort),изобретенная Ч. Э. Р. Хоаром в 1962 году, а также распределяющая поразрядная сортировка Х. Х. Сьюворда (Radix2), изобретенная в 1954 году.

Распределяющая поразрядная сортировка Х. Сьюворда является одной из самых быстрых сортировок. Она предназначена для сортировки положительных чисел путем предварительного подсчета частоты повторения выбранной последовательности битов в числе. Начиная с младших битов, элементы массива перемещаются в новый

массив-буфер согласно ранее подсчитанному количеству элементов с такой же последовательностью битов. После такого перемещения числа во втором массиве расставлены в порядке не убывания данной последовательности битов. Далее массив сортируется по следующей цепочке битов такой же длины. Числа перемещаются обратно в исходный массив. После этого они находятся в порядке не убывания данной последовательности битов и всех предыдущих последовательностей. Так сортировка проводится по всем битовым цепочкам числа и в конце массив становится отсортированным по возрастанию.

Этот алгоритм сортировки примерно в три раза обгоняет Быструю Сортировку (QuickSort) по скорости на случайных равномерно распределенных массивах. Важным преимуществом Radix2 с точки зрения программирования является также отсутствие рекурсии. Выигрыш в скорости достигается путем значительного уменьшения

количества перемещения данных. Каждый элемент массива перемещается одинаковое число раз - в зависимости от длины битовых цепочек. Если сортировать по цепочкам длины 8 бит, то при сортировке 16-битных чисел каждое число перемещается ровно два раза. Даже при работе с массивами чисел, это приводит к колоссальному выигрышу в скорости. При сортировке записей меньшее количество перемещений элементов дает еще больший выигрыш. Сравнения элементов в поразрядной сортировке вообще отсутствуют.

Однако, такое улучшение в скорости приводит к некоторым недостаткам сортировки. Radix2 использует дополнительный массив-буфер, который занимает столько же места, как и исходный массив. Кроме того, заводится массив для подсчета частот повторения битовых цепочек. При сортировке массивов небольшого размера (до 1000 элементов) эти недостатки не играют особой роли, но в реальных приложениях размеры массивов нередко достигают нескольких сотен

тысяч элементов. Все это приводит к тому, что Radix2 значительно проигрывает QuickSort по размеру используемой памяти. В некоторых областях применения это может иметь очень большое значение и таким образом склонить выбор в сторону Быстрой сортировки. То, что Radix2 сортирует массивы только по численным ключам, используя их двоичное представление, заставляет применять его только в специальных областях. Поэтому QuickSort является более универсальной.

Для улучшения характеристик алгоритма поразрядной сортировки Сьюворда, необходимо уменьшить количество используемой ею памяти. Наличие массивов, используемых для подсчета частот битовых цепочек, обусловлено принадлежностью алгоритма к классу распределяющих сортировок, поэтому единственная возможность улучшить сортировку в данном направлении - попытаться избавиться от дополнительного массива-буфера.

В процессе работы над улучшением характеристик распределяющей поразрядной сортировки был предложен новый алгоритм, который не использует дополнительный массив и также как и Radix2 основан на подсчете частот повторений битовых цепочек определенной длины.

### *Распределяющая поразрядная сортировка Змановского.*

Поскольку новый алгоритм разрабатывался путем усовершенствования сортировки Сьюворда, очевидно, что основной ее принцип - подсчет частот повторений битовых цепочек должен был остаться и в новом алгоритме. Иначе терялась бы всякая логическая связь с самой быстрой сортировкой, и возможное преимущество в скорости терялось бы тоже. На самом деле, не было очевидно, что такая сортировка обгоняла бы QuickSort. На это можно было только надеяться, ведь алгоритм, вероятнее всего, был бы принципиально

новым. Никаких сомнений не было только по поводу того, что новая сортировка будет проигрывать сортировке Сьюворда по времени, ведь Radix2 после подсчета частот повторений битовых цепочек обладает "идеальными" условиями для дальнейшей работы (огромное количество используемой памяти, никаких дальнейших сравнений элементов, никакой обработки информации, практически без вычислений элементы просто перемещаются на нужные места). В новой же сортировке, скорее всего, пришлось бы производить долгие вычисления.

После долгих поисков, был придуман алгоритм, который не требует использования дополнительного массива, но работает только для сортировки по самим числам, а не по их битовым цепочкам. Далее приведена его примерная схема:

***Алгоритм математической сортировки Змановского.***

Этот алгоритм сортирует записи R1, R2,…,Rn по ключам K1, K2, …, Kn, используя вспомогательную таблицу COUNT[1], COUNT[2], … , COUNT[N] для подсчета числа ключей, меньших данного. После завершения алгоритма величина COUNT[j] + 1 определяет окончательное положение записи Rj.

Все элементы массива "делятся" на уже поставленные в нужное место и не поставленные. В самом начале - все элементы считаются стоящими не на своем месте.

1. Выбираем (берем в буфер buf1) любой не поставленный элемент, например последний элемент массива. (buf1 := a[n]).
2. Через массив Count (в котором мы запоминали частоты повторений битовых цепочек), также как и в Radix2 находим место в массиве (с индексом p), куда мы должны поставить этот элемент (buf1). (p:=Count[buf1]).

3. Пересылаем в буфер (buf2) элемент, который стоит на том месте, куда мы должны поставить ранее выбранный (buf1). (buf2:=a[p]).
4. Ставим выбранный элемент (buf1) на положенное место и уменьшаем соответствующий элемент массива Count. (a[p]:=buf1; Count[buf1] := Count[buf1] - 1 ).
5. Пересылаем данные из buf2 в buf1, теперь мы будем искать, куда поставить этот элемент. (buf1 := buf2).
6. Делаем это до тех пор, пока "цепочка" не замкнется, то есть пока на место элемента, выбранного в п.1 не поставят другой (на самом деле сюда могут поставить и его самого).
7. Если цепочка не замкнулась, то возвращаемся к п.2. Если замкнулась, то идем далее.
8. Массив Count2 на некотором этапе полностью равен массиву Count (Count2:=Count). При этом элементы обоих массивов

равны индексам концов цепочек массива с нужным значением последовательности битов, по которым массив сортируется. Массив Count2 остается неизменным и служит для того, чтобы заново не высчитывать индексы концов цепочек. Элемент же массива Count уменьшается, как только соответствующий элемент исходного массива будет поставлен на свое место. Совершенно очевидно, что когда массив отсортируется по нужной последовательности битов, все элементы массивов Count и Count2 будет связывать такое соотношение:

Count2[q-1]=Count[q] , для q=1..255. (То есть элементы массива Count теперь будут показывать индексы концов цепочек с предыдущим значением последовательности битов, так как все элементы уже поставлены на место). Значит, после того, как какая-то цепочка замкнулась в п.7, мы должны найти

следующий "не поставленный" элемент, а если такого не найдется, то завершить цикл.

```
.          .          .
  var bbuf,q,p,i:integer;
     buf1,buf2:word;
.          .          .
   Count2:=Count;Count2[-1]:=-1; {Граничный элемент}
   q:=255;
   repeat
     i:=Count[q];buf1:=a[i];                    {1}
     repeat
       p:=Count[buf1];                          {2}
       buf2:=a[p];                              {3}
       a[p]:=buf1;dec(Count[bbuf]);             {4}
       buf1:=buf2;                              {5}
     until p=i;                                 {6,7}
     while Count2[q-1]=Count[q]do dec(q);       {8}
    until q<1;
.          .          .
```

Таким образом данный массив сортируется без использования дополнительного массива-буфера.

Данный алгоритм не подходит для распределяющей поразрядной сортировки Сьюворда. В Radix2 при перемещениях данных из исходного массива в массив-буфер, а также при перемещениях в обратную сторону, соблюдается одно главное правило: порядок элементов, отсортированных на предыдущих этапах не нарушается. В

этом и состоит главная идея алгоритма Radix2: числа в начале сортируются по младшим битам, затем по старшим, не нарушая порядок среди чисел с одинаковыми битовыми цепочками. В конце массив будет упорядочен по старшим битам, а так как при перемещениях относительный порядок чисел с одинаковыми текущими цепочками не нарушался, то каждая группа с одинаковой последовательностью старших битов также упорядочена и по всем остальным битам. Таким образом, весь массив полностью отсортирован. Если бы в Radix2 сортировка начиналась со старших битов, то, очевидно, что она бы не работала.

При использовании нового алгоритма для сортировки массива по битовым цепочкам, обеспечивать взаимный порядок чисел, отсортированных на предыдущих этапах, не представляется возможным. Поэтому его не удалось использовать для усовершенствования Radix2.

Однако, решение проблемы все-таки удалось найти.

В отличии от Radix2, массив нужно сортировать начиная со старших битовых цепочек. Естественно, что после этого по младшим битовым цепочкам весь массив сортировать уже не получится. Но это и не требуется. Достаточно запомнить границы областей с одинаковыми старшими последовательностями битов (они все равно вычисляются на предыдущем этапе), а затем отсортировать каждую область по младшим цепочкам. При этом увеличение памяти происходит незначительное - приходится заводить дополнительный массив для того, чтобы помнить границы областей.

## *Характеристики ZmRadix.*

Описанный алгоритм был реализован на практике. Сортировались случайные равномерно распределенные массивы положительных чисел размером два байта. Длины битовых цепочек равнялись восьми, то есть

в начале сортировка проводилась по старшему байту, а затем по младшему. Этот выбор является оптимальным во всех отношениях: для запоминания частот используются три массива длиной 256 элементов каждый и доступ к отдельным байтам слова происходит намного быстрее чем к битам.

Было проведено тестирование алгоритма на случайных массивах разных длин и с разными границами равномерного распределения случайных величин. Для сравнения также тестировались Быстрая сортировка и Распределяющая поразрядная сортировка Сьюворда. Результаты оказались очень неожиданными: для случайных массивов с равномерным распределением, числа которого находятся в отрезке [0..N] (где N - длина массива) ZmRadix обгоняет по скорости не только QuickSort, но и Radix2. И это при том, что в новом алгоритме не используется дополнительный массив, как в Radix2, а также нет рекурсии, как в QuickSort. При сортировке массивов с распределением

из отрезка [0..L], и увеличении значения L по сравнению с длиной массива N, алгоритм ZmRadix начинает вести себя нестабильно и проигрывает в скорости Быстрой сортировке при L>5*N (в зависимости от N) и Поразрядной сортировке при L>2*N .При дальнейшем увеличении верхней границы распределения ZmRadix начинает проигрывать в скорости еще сильнее. Анализ этого явления будет проведен ниже. Однако необходимо отметить, что "классический" ZmRadix легко модифицируется и этот недостаток полностью устраняется. Способ усовершенствования алгоритма также будет рассмотрен ниже.

## *Сравнительные таблицы сортировок Radix2, ZmRadix и QuickSort.*

Сортировки тестировались на компьютере с процессором Pentium 166 MMX.

Опции компилятора Borland Pascal 7.0:

{$A+,B-,D+,E+,F-,G-,I-,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+,Y+}

{$M 16384,0,655360}

(Отключен контроль за переполнением стека, выходом за границы массива и переполнением при выполнении арифметических операций, так как это значительно увеличивает время сортировки).

Число элементов массива - 30.000, сортировка запускается 300 раз подряд (иначе, тяжело отслеживать время выполнения сортировки - она происходит слишком быстро), верхняя граница распределения чисел в массиве L = N. Массив генерируется случайно при помощи стандартной процедуры. Время сортировки при этом всегда одинаковое (в пределах одной миллисекунды). Это объясняется тем, что массивы имеют

большую длину, равномерно распределены и сортируются сразу много раз (из теории вероятностей известно, что при количестве испытаний, стремящемся к бесконечности, эмпирически полученная вероятность приближается к теоретической). Чистое время выполнения алгоритма сортировки - время выполнения минус время генерации случайных массивов.

Время в минутах: секундах: сотых долях.

N = 30 000;  K = 300;  L = N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:12:91 | 140 966 | 00:07:26 |
| Radix2 | 00:14:34 | 60 000 | 00:08:69 |
| QuickSort | 00:29:44 | 351 920 | 00:23:79 |

Теперь проверим, что будет происходить со временем работы алгоритмов при увеличении L:

N = 30 000;  K = 300;  L = 2*N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:15:98 | 141 758 | 00:10:33 |
| Radix2 | 00:14:34 | 60 000 | 00:08:69 |
| QuickSort | 00:29:76 | 359 443 | 00:24:11 |

При L = 2*N ZmRadix по-прежнему значительно обгоняет по скорости Быструю сортировку, но уступает Поразрядной сортировке Сьюворда.

N = 10 000;  K = 500;  L = 6*N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:14:45 | 45 255 | 00:11:21 |
| Radix2 | 00:08:13 | 20 000 | 00:04:89 |
| QuickSort | 00:13:18 | 110 295 | 00:09:94 |

Как видно из таблицы, при L = 6*N ZmRadix уступает по скорости уже и Быстрой сортировке, но пока незначительно.

При дальнейшем увеличении L по сравнению с N скорость ZmRadix будет по-прежнему падать.

## *Анализ распределяющей поразрядной сортировки Змановского.*

Распределяющая поразрядная сортировка Змановского оказалась самым быстрым алгоритмом сортировки случайных равномерно распределенных массивов для L = N (где N - длина массива, L - верхняя граница области распределения чисел). Он обгоняет не только QuickSort, но и Radix2. Эмпирически подсчитано, что при разных N и L ZmRadix всегда производит в два раза больше перемещений данных, чем Radix. И несмотря на это, он, хотя и не много, но быстрее его по скорости. Это связано с тем, что, в связи с особенностями алгоритма ZmRadix, значительная часть перемещений производится при использовании локальных переменных, работа с которыми происходит намного быстрее, чем с элементами массива. В связи с особенностями распределяющих сортировок, сравнений в Radix и ZmRadix не производится совсем, а подсчет частот повторения битовых цепочек занимает одинаковое время.

Выигрывая в скорости, ZmRadix также прекрасно выглядит в аспекте размера используемой памяти. То, что Radix2 хуже по этому показателю – очевидно. QuickSort в худших вариантах массива приводит к использованию больших объемов памяти. Конечно, речь идет о сортировке больших массивов (n > 700), так как если n мало, то вполне можно завести дополнительный массив и использовать Radix2).

Также ZmRadix показывает себя великолепно с точки зрения количества "перемещений" элементов массива. Перемещений в ZmRadix примерно в 3 раза меньше, чем в QuickSort (эти данные получены эмпирически). При сортировке записей - это очень важное преимущество, так как перемещение данных большей длины занимает больше времени. Как уже было сказано, ZmRadix уступает Radix2'у по этому показателю (ровно в 2 раза), но Radix2 использует дополнительный массив, поэтому все элементы пересылаются напрямую, без использования буферов.

Замедление скорости работы алгоритма ZmRadix при увеличении L происходит потому, что при этом увеличивается число отрезков, которые нужно будет сортировать на втором этапе (по младшему байту). При этом, основная проблема заключается не конкретно в увеличении количества отрезков, а в уменьшении их средних длин. Очевидно, что для сортировки массивов малой длины выгоднее использовать простые методы (так, например, известно, что массивы длиной менее 8 элементов метод Пузырька сортирует значительно быстрее, чем QuickSort). ZmRadix же даже для такого массива подсчитывает частоты распределений младших байтов ключей, поэтому на небольших массивах проигрывает более простым алгоритмам.

Средняя длина подмассива на втором этапе сортировки равна x = N/d, где d - наибольшее значение старшего байта среди всех ключей массива, очевидно, d = (L + 255) div 256 (div - операция целочисленного деления).

Можно подсчитать средние длины подмассивов при тестировании сортировок, которое описывалось выше:

при N = 30 000 ; L = N; x = 254,237;

при N = 30 000 ; L = 2*N; x = 127,660;

при N = 10 000 ; L = 6*N; x = 42,553;

Из полученных значений видим, почему, например, при L=6*N ZmRadix проигрывает остальным сортировкам - средний размер подмассива на втором проходе сортировки (по младшим байтам) всего 42 элемента.

Очевидно, что решить проблему замедления скорости работы алгоритма ZmRadix при увеличении L очень просто. Поскольку данная сортировка не очень хорошо показывает себя при сортировке массивов, меньших определенного предельного значения, то на втором этапе нужно сравнивать длину подмассива с имеющимся предельным значением, которое можно вывести эмпирически, сравнивая скорость

работы сортировки с каким-либо другим методом (тестирование показало, что такое значение приблизительно равно 100). Если длина подмассива меньше 100, то его нужно сортировать какой-нибудь другой сортировкой (либо сортировками), например Быстрой.

Проведем снова три теста с такими же значениями L, N и K, как и прежде. Только теперь в сортировке ZmRadix подмассивы, длина которых меньше 100 элементов будем сортировать Быстрой сортировкой, а те, длина которых меньше 8 элементов будем сортировать методом Пузырька (значение предельной длины подмассива на разных компьютерах, вообще говоря, может отличаться, но не значительно).

N = 30 000;  K = 300;  L = N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:12:91 | 140 976 | 00:07:26 |
| Radix2 | 00:14:34 | 60 000 | 00:08:69 |
| QuickSort | 00:29:44 | 351 920 | 00:23:79 |

Время в минутах: секундах: сотых долях.

Видим, что время выполнения ZmRadix осталось прежним, значит, в большинстве случаев длины подмассивов превышали выбранное нами предельное значение.

N = 30 000;  K = 300;  L = 2*N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:15:98 | 142 744 | 00:10:33 |
| Radix2 | 00:14:34 | 60 000 | 00:08:69 |
| QuickSort | 00:29:76 | 359 443 | 00:24:11 |

Здесь также время выполнения ZmRadix осталось прежним, значит, в большинстве случаев длины подмассивов превышали выбранное нами предельное значение.

N = 10 000;  K = 500;  L = 6*N;

| Сортировка | Время выполнения | Среднее число перемещений | Чистое время выполнения |
|---|---|---|---|
| ZmRadix | 00:09:72 | 67 123 | 00:06:48 |
| Radix2 | 00:08:13 | 20 000 | 00:04:89 |
| QuickSort | 00:13:18 | 110 295 | 00:09:94 |

Здесь же произошло увеличение скорости работы алгоритма примерно в два раза. Это объясняется тем, что средняя величина подмассивов (теоретически она равна 42,553) на втором этапе сортировки меньше выбранного нами предельного значения (100), поэтому большинство подмассивов сортировались Быстрой сортировкой.

Такое усовершенствование алгоритма ZmRadix делает сортировку устойчивой к изменению значений ключей массива, изменению верхней границы области равномерного случайного распределения. При этом, все же, ZmRadix немного уступает Radix2 по скорости, так как подмассивы сортируются Быстрой сортировкой, но зато обладает более хорошими характеристиками в отношении используемой памяти.

## *Заключение.*

Распределяющая поразрядная сортировка Змановского является новым алгоритмом сортировки массивов записей по их ключам.

Скорость выполнения алгоритма на случайных равномерно распределенных массивах при L = N (где N - длина массива, L - верхняя граница области распределения чисел) относит сортировку к наиболее быстрым алгоритмам, таким как Быстрая сортировка (QuickSort) и Распределяющая поразрядная сортировка Сьюворда (Radix2). Тестирование показывает, что Распределяющая поразрядная сортировка Змановского (ZmRadix) обгоняет оба алгоритма при вышеуказанном случайном распределении.

Важным преимуществом нового алгоритма является использование малых объемов памяти. Radix2 всегда заводит дополнительный массив в связи с особенностями алгоритма, а QuickSort и в рекурсивном, и в не рекурсивном вариантах в самых

лучших случаях требует в стеке (либо в мнимом стеке при не рекурсивном алгоритме) объем памяти, пропорциональный двоичному логарифму из N. В худших случаях QuickSort использует колоссальный объем памяти.

Сравнения в ZmRadix, как и в Radix2, отсутствуют. Перемещений же в новой сортировке примерно в 3 раза меньше, чем в QuickSort, и в 2 раза больше, чем в Radix2, но при этом, в связи с преимуществами алгоритма, использование локальных переменных при перемещении элементов позволяет сортировке Змановского обгонять сортировку Сьюворда по времени работы.

Задача сортировки случайных равномерно распределенных массивов при L = N встречается очень часто. Наиболее распространенный пример - разработка баз данных, в которых записи в подавляющем большинстве случаев однозначно определяются по уникальным числовым ключам. Уникальность ключей удовлетворяет

необходимому условию распределения массива (L = N). Поэтому при сортировке таких данных ZmRadix будет обгонять все остальные методы сортировки, а также значительно уменьшит объем используемой памяти.

Таким образом, изобретенный алгоритм имеет огромное практическое значение - его можно эффективно использовать в наиболее развивающейся в настоящее время области применения компьютерных технологий - разработке баз данных.

Распределяющая поразрядная сортировка Змановского - это принципиально новый алгоритм, который встал на одну ступень с такими сортировками, как

Распределяющая поразрядная сортировка (Х. Сьюворд) 1954 г.,

Метод Шелла (Д. Л. Шелл) 1959 г.,

Обменная поразрядная сортировка (Г. Исбитц) 1959 г.,

Быстрая сортировка (Ч. Э. Р. Хоар) 1962 г.,

Метод Бэтчера (К. Э. Бэтчер) 1964 г.,

Пирамидальная сортировка (Дж. У. Дж. Уильямс) 1964 г.,

Распределяющая поразрядная сортировка Змановского

(Змановский П.П.) 2001 г.

## *Список использованной литературы.*

1. Кнут Д. Искусство программирования. М.: Вильямс, 2000. Т.3: Сортировка и поиск.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение вычислительных алгоритмов. М.: Мир, 1979.
3. Гэри М., Дэжонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
4. Липский В. Комбинаторика для программистов. М.: Мир, 1988.
5. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1985.