Эффективные алгоритмы сортировки 2

**Лексикографическое упорядочение строк**

Эффективные алгоритмы сортировки со сравнениями не годятся, т.к. сравнение строк – не элементарный шаг.

Выход – цифровая сортировка от конца строк.

Для строк одинаковой длины  сортировка практически не отличается от сортировки целых чисел (вместо цикла по 4 байтам выполняется цикл по  символам от конца строк). Трудоемкость = .

Обычно длины строк  существенно различаются, и . Дополнять все строки пробелами до длины  невыгодно ни по времени, ни по памяти.

=>

Сортировать по символам в позиции  нужно только строки длины , причем в очереди на просмотр должны сначала идти все строки длины , а потом все более длинные строки в порядке их сортировки по символу .

Если при сортировке по символу  для выделения строк длины  нужно просматривать весь массив из  строк, то только этот просмотр потребует  ЭШ.

=>

Необходимо заранее сформировать списки строк по длинам, тогда при сортировке по символу  можно за 1 ЭШ добавить список строк длины  к началу текущей очереди строк, отсортированных по символу .

Все списки должны хранить по 2 указателя (ссылки) – на начальный и конечный элементы.

В приведенном ниже алгоритме используются переменные:

**A[n]** – исходный массив строк (указателей),

**lmax** – максимальная длина строки,

**LLen[lmax]** – массив списков строк по длинам,

**LSym[256]** – массив списков строк по символам.

**List sort\_string(char \*\*A, int n)**

**{**

**int i, k, lmax = 0;**

**List LRes, \*LSym, \*LLen;**

**LSym = new List[256];**

**for (i = 0; i < n; i++)**

**if (strlen(A[i]) > lmax) lmax = strlen(A[i]);**

**LLen = new List[lmax];**

**for (i = 0; i < n; i++)**

**LLen[strlen(A[i])-1].push\_back(i);**

**for (k = lmax-1; k >= 0; k--)**

**{**

**LLen[k].join(LRes);**

**while (1) {**

**i = LLen[k].pop\_front();**

**if (i < 0) break;**

**LSym[A[i][k]].push\_back(i);**

**}**

**for (i = 0; i < 256; i++) LRes.join(LSym[i]);**

**}**

**delete [] LLen; delete [] LSym;**

**return LRes;**

**}**

Трудоемкость отдельных частей алгоритма.

В практических случаях , поэтому общая трудоемкость – линейная относительно .

Модификация алгоритма при больших  (Ахо).

**Поиск *k*-го минимального элемента**

Идея для общего случая:

* разделение массива опорным элементом, как в быстрой сортировке (пусть опорный элемент после разделения попадает на позицию )
* рекурсивная или рекуррентная обработка той части массива, в которую попадает -й элемент (возможны 3 варианта, в зависимости от того, какое из трех условий выполняется).

Трудоемкость:















* в наилучшем ,
* при разделении массива пополам ,
* в наихудшем .

Трудоемкость в среднем:

* различные случаи соответствуют различным ,
* вероятности для всех случаев равны ,
* , если ,
* , если .

Рекуррентное соотношение для трудоемкости в среднем:





 максимальна при , т.е. при поиске медианы

Покажем, что , при  (по мат. индукции).

Доказательство:

1. базис: 

2. пусть неравенство справедливо для , тогда



Алгоритм имеет линейную трудоемкость в среднем, а наихудшие случаи связаны с неудачным выбором опорного элемента и, как следствие, несбалансированным разделением массива на каждом шаге.

**Алгоритм с гарантированной линейной трудоемкостью**

Идея: исключить худшие случаи разделения, используя неслучайный выбор опорного (разделяющего) элемента.

Пусть массив разбит на группы по 5 элементов, значения в каждой группе упорядочены, а затем все пятерки отсортированы по значению их медианы:

Пусть  – медиана медиан пятерок элементов.



 эл-тов 

 эл-тов 

по медианам

При  в  будет не менее  эл-тов как , так и 

=>  выгодно использовать как разделяющий элемент

Для получения  совершенно не нужно сортировать массив медиан. Ниже приводится рекурсивная функция поиска -го минимального элемента, где  вычисляется рекурсивно – как медиана отдельного массива медиан.

Если длина обрабатываемой части массива , то используется простой алгоритм.

Вызовы функций:

**d = median(A, b, e, k)** – получение -го минимального элемента массива ,

**S = get\_medians(A, b, e, L)** – формирование нового массива **S** длины **L** из медиан пятерок эл-тов массива .

**double median(double \*A, int b, int e, int k)**

**{**

**int L, j; double d;**

**if (e – b < 50)**

**return med(A, b, e, k);** // простой алгоритм

**S = get\_medians(A, b, e, L);**

**d = median(S, 0, L-1, L/2);**

**j = separation\_by\_element(A, b, e, d);**

**if (k < j) return median(A, b, j-1, k);**

**else if (k > j) return median(A, j+1, e, k);**

**else return A[k];**

**}**



Для групп по 7 или 9 элементов разделение будет еще лучше, но требуется много сравнений при расчете медиан:

7 сравнений для 5 элементов, 13 – для 7, 19 – для 9.

Для групп по 3 элемента:

.

Выбор простого алгоритма при :

 элементов, меньших\больших ,

величина  при .

Емкостная сложность: 

**Внешняя сортировка**

Особенности ввода\вывода для внешних устройств

=>

необходим последовательный доступ и В\В блоками.

Непригодность алгоритмов внутренней сортировки.

Основные требования для внешней сортировки слиянием:

1. Сливаемые серии должны располагаться в разных файлах, новые серии должны записываться в отдельный файл (или файлы).

2. Для уменьшения числа проходов необходимо формировать начальные серии максимальной длины.

3. Общее число файлов и порядок размещения начальных серий на входных файлах должны обеспечивать выполнение всех проходов алгоритма без дополнительного копирования.

**Двухпутевое сбалансированное слияние**

В данной сортировке используется 2 входных и 2 выходных файла (ленты). Пусть эти файлы имеют номера 0, 1, 2 и 3.

Прежде всего, необходимо сформировать начальные серии:

1. Длина начальных серий  выбирается таким образом, чтобы серию можно было целиком поместить в ОП (чем больше величина , тем лучше).

2. Элементы исходного файла считываются в ОП группами длины  (последняя может оказаться короче).

3. Элементы в группах упорядочиваются с помощью какой-либо внутренней сортировки.

4. Полученные серии поочередно записываются в файлы 0 и 1 (на 1-м проходе файлы 0 и 1 будут входными, а файлы 2 и 3 – выходными).

На каждом проходе алгоритма последовательно сливаются все пары серий из входных файлов, а получаемые серии удвоенной длины поочередно записываются в выходные файлы.

На следующем проходе входные и выходные файлы меняются ролями.

На последнем проходе входные файлы должны содержать по одной серии – в результате их слияния будет получена одна серия (отсортированный массив) – либо в файле 2 (при нечетном числе проходов), либо в файле 0 (при четном).

Пример с 7 нач. сериями (входные файлы на всех проходах):

Если общее число сортируемых элементов , а длина начальных серий , то число проходов составляет .

7

5

3

1

2

4

6

6

5

2

1

4

3

2

1

5

6

7

3

4

7

0

1

0

3

2

1

На каждом проходе все  элементов считываются из входных файлов и записываются в выходные, следовательно, общее число чтений\записей элементов составляет .

Варианты выделения отдельных входных серий:

1. Проверка по длине текущих серий (, , ). Последняя серия может оказаться короче, поэтому при чтении необходимо проверять, не достигнут ли конец файла.

2. Запись после **каждой** серии (начальной или полученной в результате слияния) специального значения , которое должно быть больше любого исходного значения.

В приведенном ниже алгоритме используются следующие переменные и значения:

**f[4]** – массив файловых переменных,

**r** – номер 1-го входного файла (2-й имеет номер **r+1**),

**w** – номер файла, в который пишется текущая выходная серия,

**k** – текущее число серий,

**x** – текущее значение из 1-го входного файла,

**y** – текущее значение из 2-го входного файла,

**MAX** – разделительное значение для входных серий.

Функция **get\_elem(F)** возвращает очередное число из файла **F** или значение **MAX** при **eof**,

**put\_elem(F, z)** пишет число **z** в файл **F**.

Предполагается, что начальные серии уже записаны в файлы **f[0]** и **f[1]**.

**r = 0; w = 2; k = число\_начальных\_серий;**

**while (k > 1) { k = 0;**

**открыть\_для\_чтения(f[r], f[r+1]);**

**открыть\_для\_записи(f[w], f[w+1]);**

**while (!eof(f[r]) || !eof(f[r+1])) {**

**x = get\_elem(f[r]); y = get\_elem(f[r+1]);**

**while (x < MAX || y < MAX) {**

**if (x <= y)**

**{ put\_elem(f[w],x); x=get\_elem(f[r]); }**

**else**

**{put\_elem(f[w],y); y=get\_elem(f[r+1]);}**

**}**

**put\_elem(f[w], MAX); k++;**

**if (w%2 == 1) w--; else w++;**

**} закрыть\_все\_файлы(); r = 2–r; w = 2–r;**

**}**

Реализация функций **get\_elem** и **put\_elem** при блоковом вводе-выводе.

Разные физические устройства для файлов.

**Многопутевое слияние** ( входных и выходных файлов)

На каждом проходе алгоритма последовательно сливаются по  серий из входных файлов. Получаемые серии поочередно записываются в выходные файлы.

На следующем проходе входные и выходные файлы меняются ролями.

На каждом проходе длина серий увеличивается в  раз, при  исходных эл-тах и длине начальных серий  требуется  проходов => всего  чтений\записей

**Многофазное слияние**

Используется  входных () и 1 выходной файл (-й). Начальные серии распределяются по входным файлам по определенному правилу, при этом чем больше номер файла, тем меньше в нем начальных серий

На каждом проходе последовательно сливается по  серий с файлов  на -й до тех пор, пока не будет достигнут конец файла . После этого закрываются только файлы  и , для остальных чтение будет продолжено на следующем проходе. Перед следующим проходом все файлы перенумеровываются циклическим сдвигом номера на 1: 

Как всегда при слиянии, перед последним проходом входные ленты должны содержать по одной серии

**Случай  входных лент**

Числа входных серий  и коэффициенты увеличения длин серий  для входных файлов на каждом проходе распределяются по значениям двух соседних чисел Фибоначчи ранга 2

, 

,  => 

Распределение по входным файлам чисел и длин серий (отн-но длин начальных серий) при  начальных сериях

1

2

3

4

5

6

7

8

1

2

3

4

5

6

7

8

1

2

7

3

4

8

6

5

1

2

5

6

7

3

4

8



1

1

2

1

3

2

3

5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Проход | 1 | 2 | 3 | … |  |
| Отн.длины серий |  |  |  | … |  |
| Число серий |  |  |  | … |  |
| Отн.число сливаемых |  |  |  | … |  |

Известно, что  (золотое сечение) достигается очень быстро => для двух послед-ных проходов отношение числа сливаемых 

=> доля сливаемых значений для всех проходов (кроме последнего) примерно одинакова и составляет , т.е. на каждом из  проходов переписывается  значений.

Известно, что ,

поэтому 

Если число начальных серий , то необходимо записать на входные файлы  пустых серий (просто некоторое значение MAX), которые желательно распределять по файлам равномерно.

**Случай  входных лент**

Числа входных серий  и коэффициенты увеличения длин серий  на каждом проходе распределяются по значениям обобщенных чисел Фибоначчи ранга .





При :  0, 0, 1, 1, 2, 4, 7, 13, 24,…

 1, 1, 1, 3, 5, 9, 17, 31, 57,…

Распределение по файлам  начальных серий ():



Файлы: 0 1 m-2 m-1Число проходов .

При равном общем числе файлов  многофазное слияние выгоднее сбалансированного, если  (т.е. ).

Приведенный ниже алгоритм многофазной сортировки с  входными файлами во многом аналогичен алгоритму двухпутевого сбалансированного слияния.

Основные отличия:

* начальные серии дополнены необходимым числом пустых,
* **r1** и **r2** – текущие номера входных файлов, **w** – текущий номер выходного файла в массиве **f[3]**,
* на каждом проходе открываются и закрываются только 2 файла,
* используются стандартные функции ввода\вывода.

Полагаем, что  исходных (сортируемых) **целых** значений уже записаны в файл **f[2]**.

Для формирования начальных серий длины  исходные значения будут считываться из **f[2]** группами по  элементов и сортироваться в группах (с помощью некоторой внутренней сортировки **inner\_sort**), а получаемые в результате начальные серии будут записываться в **f[0]** и **f[1]**.

При расчете общего числа начальных серий вычисляется величина **steps** – число проходов последующей сортировки.

При подготовке начальных серий используются массивы:

**series[2]** – число непустых серий во входных файлах,

**empty[2]** – число пустых серий во входных файлах,

**buf[L+1]** – массив целых для сортировки начальных серий.

**Расчет числа начальных серий для входных файлов**:

**ns = ceil((double)N / L);** // всего непустых серий

**for (steps = -1, s0 = 0, s1 = 1; s1 < ns;) {**

**s2 = s1 + s0; s0 = s1; s1 = s2;**

**steps++;**

**}**

// s1 – всего начальных серий, s0 – начальных серий в f[0]

**ne = s1 – ns;** // всего пустых серий

**empty[0] = ne / 2;** // пустых серий в f[0]

**empty[1] = ne – empty[0];** // пустых в f[1]

**series[0] = s0 – empty[0];** // непустых в f[0]

// непустых серий в f[1]

**series[1] = s1 – s0 – empty[1];**

Далее предполагается, что имена файлов хранятся в массиве строк **fname[3]**, исходный файл имеет имя **"source.bin",** а **MAX** – это переменная.

Для ускорения обработки сортируемые значения хранятся в файлах во внутреннем формате (4 байта на каждое целое число, прямой или дополнительный двоичный код). Поэтому в программе используются операторы read и write, которые работают с последовательностями байт без преобразования чисел в символьный формат (не текстовые, а бинарные файлы).

**FILE \*f[3];**

**char \*fname[3] = {"f0.bin", "f1.bin", "f2.bin"};**

**int MAX = MAX\_INT;**

**Запись начальных серий на входные файлы f[0] и f[1]**:

**f[2] = fopen("source.bin", "rb");**

**for (i = 0; i < 2; i++) {** // цикл по входным файлам

**f[i] = fopen(fname[i], "wb");**

**for (j = 0; j < series[i]; j++) {**// непустые

**k = fread(buf, sizeof(int), L, f[2]);**

**inner\_sort(buf, 0, k-1);** // серия

**buf[k] = MAX;** // конец серии

// запись серии

**fwrite(buf, sizeof(int), k+1, f[i]);**

**}**

**for (j = 0; j < empty[i]; j++)** //пустые

**fwrite(&MAX, sizeof(int), 1, f[i]);**

**fclose(f[i]);**

**}**

**fclose(f[2]);**

**Многофазная сортировка с 2 входными файлами**:

**r1 = 0; r2 = 1; w = 2;**

**f[r2] = fopen(fname[r2], "rb");**

// цикл по проходам

**for (k = 0; k < steps; k++) {**

**f[r1] = fopen(fname[r1], "rb");**

**f[w] = fopen(fname[w], "wb");**

// цикл по входным сериям файла f[r2]

**while (fread(&y,sizeof(int),1,f[r2]) == 1) {**

**fread(&x, sizeof(int), 1, f[r1]);**

// слияние пары серий

**while (x < MAX || y < MAX)**

**if (x <= y) {**

**fwrite(&x, sizeof(int), 1, f[w]);**

**fread(&x, sizeof(int), 1, f[r1])**

**}**

**else {**

**fwrite(&y, sizeof(int), 1, f[w]);**

**fread(&y, sizeof(int), 1, f[r2])**

**}**

// конец полученной серии

**fwrite(&MAX, sizeof(int), 1, f[w]);**

**}**

**fclose(f[r2]); fclose(f[w]);**

**j = w; w = r2; r2 = r1; r1 = j;**

**}**

**fclose(f[r2]);**

// результат – в файле fname[r1]