ГЛАВА 9

СОРТИРОВКА И ПОИСК ДАННЫХ

1. Алгоритмы сортировки

Концепция упорядоченного множества элементов является одной из тех концепций, которые имеют существенное влияние на многие стороны нашей жизни.

В общем *сортировку* следует понимать как процесс перегруппировки заданного множества объектов в некотором определённом порядке. Цель сортировки - облегчить поиск элементов в таком упорядоченном множестве.

Уточним задачу сортировки.

Пусть надо упорядочить n элементов

$$R_1, R_2, \ldots, R_n,$$

которые назовём *записями*. Каждая запись R_j имеет свой ключ K_j, который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную "сопутствующую информацию", которая не влияет на сортировку, но всегда остаётся в этой записи.

Задача сортировки - найти такую перестановку записей $p(1)p(2)\ldots p(n)$, после которой ключи расположились бы в неубывающем порядке:

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

При сортировке перемещаются либо сами записи, либо создаётся вспомогательная таблица, которая описывает перестановку и обеспечивает доступ к записям в соответствии с порядком их ключей.

Обычно сортировку подразделяют на два класса: *внутреннюю*, когда все записи хранятся в оперативной памяти, и *внешнюю*, когда они там все не помещаются. Мы будем рассматривать только внутреннюю сортировку.

Основное требование к методам сортировки - экономное использование времени процессора и памяти. Хорошие алгоритмы затрачивают на сортировку n записей время порядка $n \log n$.

Существующие методы сортировки обычно разбивают на три класса, в зависимости от лежащего в их основе приема:

а) сортировка выбором,

б) сортировка обменами,

в) сортировка включениями (вставками).

Рассмотрим основные алгоритмы сортировки на примере сортировки целочисленного массива.

Линейный выбор

Метод предполагает использование рабочего массива, в который помещается отсортированный массив. Количество просмотров определяется количеством элементов массива. Сортировка посредством линейного выбора сводится к следующему:

1) Найти наименьший элемент, переслать его в рабочий массив и заменить его в исходном массиве величиной, которая больше любого реального элемента.

2) Повторить шаг (1). На этот раз будет выбран наименьший из оставшихся элементов.

3) Повторять шаг (1) до тех пор, пока не будут выбраны все *n* элементов.

Функция, реализующая алгоритм линейного выбора, имеет следующий вид:

```
#include <limits.h>
void lin_wib(int *a, int n)
{
 int i, j, imin, amin, *p;
 p=(int *)malloc(n*sizeof(int));  // выделяем память под рабочиий массив
 for(j=0; j<n; j++)
   {
    for(amin=INT_MAX, i=0; i<n; i++)
       if(a[i]<amin) { imin=i; amin=a[i]; }  // ищем минимальный элемент
    p[j]=amin; a[imin]=INT_MAX;
   }
 for(j=0; j<n; j++)   //отсортированный массив --> на место исходного
  a[j]=p[j];
 free(p);
}
```

Линейный выбор с обменом

В этом методе рабочий массив не используется. Общая схема алгоритма следующая.

Просмотрим элементы $a_1, a_2, \ldots, a_n$, найдем среди них минимальный элемент и переставим его на первое место. Теперь просмотрим элементы $a_2, \ldots, a_n$, найдем среди них минимальный элемент и поставим его на второе место. И так до тех пор, пока не останется один элемент.

Стандартный обмен (метод "пузырька")

Просматриваем элементы $a_1, \ldots, a_n$ и попутно меняем местами те соседние элементы, для которых выполнено неравенство $a_i > a_{i+1}$. В результате первого просмотра максимальный элемент станет последним ("он вниз - пузыри вверх"). На следующем просмотре аналогичную процедуру проведем над

элементами $a_1, \ldots, a_{n-1}$ и т.д. Сортировку необходимо закончить, если будет выполнено одно из двух условий:

- после очередного прохода не сделано ни одного обмена;
- сделан проход для элементов a_1, a_2.

Челночная сортировка

Челночная сортировка, называемая также "сортировкой просеиванием" или "линейной вставкой с обменом" является наиболее эффективной из всех рассмотренных выше методов и отличается от сортировки обменом тем, что не сохраняет фиксированной последовательности сравнений.

Алгоритм челночной сортировки действует точно так же, как стандартный обмен до тех пор, пока не возникает необходимость перестановки элементов исходного массива. Сравниваемый меньший элемент поднимается, насколько это возможно, вверх. Этот элемент сравнивается в обратном порядке со своими предшественниками по направлению к первой позиции. Если он меньше предшественника, то выполняется обмен и начинается очередное сравнение. Когда элемент, передвигаемый вверх, встречает меньший элемент, этот процесс прекращается и нисходящее сравнение возобновляется с той позиции, с которой выполнялся первый обмен.

Сортировка заканчивается, когда нисходящее сравнение выходит на границу массива.

Процесс челночной сортировки можно проследить на следующем примере:

Исходный массив: 2 7 9 5 4

Нисходящие сравнения: 2 7; 7 9; ***9 5;***

После перестановки: 2 7 ***5 9*** 4

Восходящие сравнения и обмен : ***7 5*** -> ***5 7***; 2 5 - конец восходящего сравнения; получен массив : 2 5 7 9 4

Нисходящие сравнения: ***9 4***

После перестановки: 2 5 7 ***4 9***

Восходящие сравнения и обмен : ***7 4*** -> ***4 7***; ***5 4*** -> ***4 5***; 2 4 - конец восходящего сравнения; получен массив : 2 4 5 7 9 .

Сортировка Шелла

Все рассмотренные выше методы обмена были не столь эффективны при реализации на ЭВМ из-за сравнений и возможных обменов только соседних элементов. Лучших результатов следует ожидать от метода, в котором пересылки выполняются на большие расстояния. Один из таких методов предложил D.L.Shell.

Сортировка Шелла, называемая также "слиянием с обменом", является расширением челночной сортировки.

Идея метода заключается в том, что выбирается интервал h между сравниваемыми элементами, который уменьшается после каждого прохода. Для последовательности интервалов выполняются условия:

$$h_p = 1, \quad h_{i+1} < h_i, \quad i = 1,2,\ldots, p-1.$$

Таким образом, вначале сравниваются (и, если нужно, переставляются) далеко стоящие элементы, а на последнем проходе - соседние элементы.

Выигрыш получается за счет того, что на каждом этапе сортировки либо участвует сравнительно мало элементов, либо эти элементы уже довольно хорошо упорядочены, и требуется небольшое количество перестановок. Последний просмотр сортировки Шелла выполняется по тому же алгоритму, что и в челночной сортировке. Предыдущие просмотры подобны просмотрам челночной обработки, но в них сравниваются не соседние элементы, а элементы, отстоящие на заданное расстояние h. Большой шаг на начальных этапах сортировки позволяет уменьшить число вторичных сравнений на более поздних этапах.

При анализе данного алгоритма возникают достаточно сложные математические задачи, многие из которых еще не решены. Например, неизвестно, какая последовательность расстояний дает лучшие результаты, хотя выяснено, что расстояния h_i не должны быть множителями один другого. Можно рекомендовать следующие последовательности (они записаны в обратном порядке):

а) 1, 4, 13, 40, 121, ... , где $h_{k-1} = 3h_k + 1, \quad h_p = 1$ и $p = \left[\log_2 n\right] - 1$;

б) 1, 3, 7, 15, 31, ... , где $h_{k-1} = 2h_k + 1, \quad h_p = 1$ и $p = \left[\log_2 n\right] - 1$.

Линейная вставка

Линейная (*простая*) *вставка* основана на последовательной вставке элементов в упорядоченный рабочий массив. Линейная вставка чаще всего используется тогда, когда процесс, внешний к данной сортировке, динамически вносит изменения в массив, все элементы которого известны и который все время должен быть упорядоченным.

Алгоритм линейной вставки следующий. Первый элемент исходного массива помещается в первую позицию рабочего массива. Следующий элемент исходного массива сравнивается с первым. Если этот элемент больше, он помещается во вторую позицию рабочего массива. Если этот элемент меньше, то первый элемент сдвигается на вторую позицию, а новый элемент помещается на первую. Далее все выбираемые из исходного массива элементы последовательно сравниваются с элементами рабочего массива, начиная с первого, до тех пор, пока не встретится больший. Этот больший элемент и все последующие элементы рабочего массива перемещаются на одну позицию вниз, освобождая место для нового элемента.

Центрированная и двоичная вставки

Введение "структуры" в упорядочиваемый рабочий массив в общем случае сокращает количество сравнений, выполняемых при поиске позиции элемента в упорядочиваемом массиве. При использовании "структуры" в сортировке вставкой обычно применяют либо центрированную, либо двоичную вставки. Эти алгоритмы просты, но обладают особенностями реализации, характерными для нелинейных алгоритмов сортировки

Центрированная вставка

Представим рабочий массив состоящим как бы из двух ветвей - нисходящей (левой) и восходящей (правой). Центральный элемент этого массива называется медианой.

Приведём описание алгоритма центрированной вставки.

В позицию, расположенную в середине рабочего массива, помещается первый элемент (он и будет медианой). Нисходящая и восходящая ветви имеют указатели, которые показывают на ближайшие к началу и концу занятые позиции. После загрузки первого элемента в центральную позицию оба указателя совпадают и показывают на него. Следующий злемент исходного массива сравнивается с медианой. Если новый элемент меньше, то он размещается на нисходящей ветви, в противном случае - на восходящей ветви. Кроме того, соответствующий концевой указатель продвигается на единицу вниз (нисходящая ветвь) или единицу вверх (восходящая ветвь).

Каждый последующий элемент исходного массива сравнивается вначале с медианой, а затем с элементами на соответствующей ветви до тех пор, пока не займёт нужную позицию. Если область памяти, выделенная для одной ветви, будет исчерпана, то все элементы рабочего массива сдвигаются в противоположном направлении. Величина сдвига может варьироваться.

Двоичная вставка

Двоичная (*бинарная*) вставка в отличии от линейной использует для отыскания места вставки алгоритм двоичного (бинарного) поиска, т.е. при вставке *j-го* элемента он сравнивается вначале с элементом [*j*/2], затем, если он меньше, сравнивается с элеменом [j/4], а если больше - с [j/2]+[j/4] и т.д. до тех пор, пока он не найдёт своё место. Все элементы рабочего массива, начиная с позиции вставки и ниже, сдвигаются на одну позицию вниз, освобождая место для *j-го* элемента.

Быстрая сортировка (метод Хоара)

Метод "пузырька" является не самым эффективным из рассмотренных алгоритмов, поскольку требует большого количества сравнений и обменов. Однако оказалось, что сортировку, основанную на принципе обмена, можно усовершенствовать таким образом, что получится самый хороший из известных на сегодняшний день методов.

Алгоритм *быстрой сортировки*, предложенный Хоаром, опять-таки использует тот факт, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях.

Суть метода заключается в следующем. Найдем такой элемент массива, который разобьет весь массив на два подмножества: те элементы, которые меньше делящего элемента, и те, которые по величине не меньше его (т.е. как бы "отсортируем" один элемент, определив его окончательное местоположение).

Далее применим эту же процедуру к более коротким левому и правому подмножествам.

Таким образом, надо реализовать рекурсивный алгоритм, который сортирует элементы множества, начиная с элемента с индексом *left* и завершая элементом с индексом *right*. Условие окончания данного алгоритма - совпадение левой и правой границ подмножества (т.е. когда в подмножестве остается один элемент).

Точку деления массива может быть определена следующим образом.

Вводятся два указателя i и j, причём вначале $i = left$, а $j = right$. Сравниваются *i-й* и *j-й* элементы и, если обмен не требуется, то $j = j - 1$ и этот процесс повторяется. После первого обмена i увеличивается на единицу $(i = i + 1)$ и сравнения продолжаются с увеличением i до тех пор, пока не произойдёт ещё один обмен. Тогда опять уменьшим j и т.д. пока не станет $i = j$. К моменту, когда $i = j$, элемент a_i займёт свою окончательную позицию, так как слева от него не будет больших элементов, а справа - меньших. Таким образом, поставленная задача решена.

Для повышения эффективности быстрой сортировки можно использовать следующий приём: не применяя рекурсивной процедуры к ставшему коротким подмножеству, для его сортировки перейти к другому методу, например, челночной сортировке. То есть рекурсивная процедура применяется только для массивов, длина которых не менее определённого размера.

2. Алгоритмы поиска

Алгоритмы поиска, как и алгоритмы сортировки, являются основными алгоритмами обработки данных как системных, так и прикладных задач.

Дан аргумент поиска K. Задача поиска состоит в отыскании записи, имеющей K своим ключом. Существуют две возможности окончания поиска: либо поиск оказался удачным, т.е. позволил определить местонахождение записи, содержащей ключ K, либо он оказался неудачным, т.е. показал, что ни одна из записей не содержит ключ K.

Рассмотрим основные алгоритмы поиска. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, то мы будем считать, что ключ и запись совпадают и имеют тип *int*.

Последовательный поиск

Для массива, данные в котором не упорядочены путём сортировки или каким-либо другим способом, единственный путь для поиска заданного элемента состоит в сравнении каждого элемента массива с заданным. При совпадении некоторого элемента массива с заданным его позиция в массиве фиксируется. Этот алгоритм называется *последовательным* или *линейным поиском*. Эфективность этого метода очень низкая, так как для отыскания одного элемента в массиве размерности N в среднем нужно сделать $N/2$ сравнений.

Бинпрный (двоичный) поиск

Бинарный или *двоичный* поиск может быть использован в качестве алгоритма поика в упорядоченном массиве: $K_1 \le K_2 \le \ldots \le K_n$.

Схема алгоритма следующая. Сначала сравнивают K со средним элементом массива. Результат сравнения позволяет определить, в какой половине массива следует продолжать поиск, применяя к ней ту же процедуру, и т.д. После не более чем $\lfloor \log_2 n \rfloor + 1$ сравнений ключ либо будет найден, либо будет установлено его отсутствие.

Интерполяционный поиск

По-видимому, бинарный поиск не является самым лучшим методом поиска, доказательством чему служит то обстоятельство, что в повседневной жизни им мало пользуются. Гораздо чаще используется *интерполяционный поиск*. Его схема следующая, Если известно, что K находится между K_l и K_r, то номер очередного элемента для сравнения определяется формулой

$$m = l + (r - l) * (K - K_l) / (K_r - K_l).$$

Последующая процедура аналогична процедуре, используемой в бинарном поиске.

Интерполяционный поиск асимптотически предпочтительнее бинарного; по существу один шаг бинарного поиска уменьшает количество "подозреваемых" записей от n до $\frac{n}{2}$, а один шаг интерполяционного - от n до $\sqrt{n}$. В среднем интерполяционнный поиск требует $\log_2 \log_2 n$ шагов.

ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ

Задача А

Написать и протестировать функцию сортировки целочисленного массива методом стандартного обмена.

Определить время сортировки массива объемом 1000, 5000 и 10000 элементов. Для контроля за сортировкой организовать выдачу 10 элементов из начала, середины и конца исходного и отсортированного массивов.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

// функция st_obmen() - сортировка методом "пузырька"
void st_obmen(int *a, int n)
{
 int i, pp, buf;

 do
   { n--;
    for(pp=i=0; i<n; i++)
       if(a[i]>a[i+1])
        { buf=a[i]; a[i]=a[i+1]; a[i+1=buf; pp=1; }
   }
 while(pp);
}

// функция printm() - печать 10 первых, средних и последних элементов массива
void printm(char *str, int *a, int n)
{
 int i, j, in;

 printf("\n\n\t\t\t %s \n", str);
 for(j=0; j<3; j++,printf("\n"))
   {
    switch(j)
       {
        case 0 : in=0; printf("  начало : "); break;
        case 1 : in=n/2-5; printf("середина : "); break;
        case 2 : in=n-10; printf("   конец : ");
       }
    for(i=in; i<in+10; i++)
       printf("%6d", a[i]);
   }
}
```

```
void main()
{
 int i, j, x[10000], n[3]={ 1000, 5000, 10000 };
 time_t t1, t2, t;
 double dt[3];
 clrscr();
 srand(2213);
 for(j=0; j<3; j++)
  {
   for(i=0; i<n[j]; i++)
     *(x+i)=rand()%n[j];
   if(!j) printm(" Исходный массив (1000 эл.)", x, n[j]);
   t1=time(&t);
   st_obmen(x, n[j]);
   t2=time(&t);
   dt[j]=difftime(t2, t1);
   if(!j) printm("  Отсортированный массив", x, n[j]);
  }
 printf("\n\n \t Время сортировки ");
 for(j=0; j<3; j++)
   printf("\n %5d элементов --> %4.1lf c ", n[j], dt[j]);
}
```

Задача B

Написать и протестировать функцию поиска ключа в целочисленном массиве
методом бинарного поиска.

Тест: поиск *m* ключей в целочисленном массиве объема *n*.

Элементы массива - случайные числа из интервала $(0, n-1)$; ключи для поиска - случайные числа из интервала $(0, n+m-1)$.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <alloc.h>
// функция bin_poisk() - бинарный поиск в целочисленном массиве
int bin_poisk(int *a, int n, int v)
{
 int l=0, k=n-1, m;
 while(l<=k)
   {
```

```
    m=(l+k)/2;
    if(v<a[m]) k=m-1;
     else if(v>a[m]) l=m+1;
            else return m;  // нашли!
   }
  return -1; // не нашли...
}
// функция st_obmen() - сортировка методом "пузырька"
void st_obmen(int *a, int n)
{ int i, pp, buf;
  do
   { n--;
    for(pp=i=0; i<n; i++)
       if(a[i]>a[i+1])
        { buf=a[i]; a[i]=a[i+1]; a[i+1=buf; pp=1; }
   }
  while(pp);
}
void main()
{ int i, n, m, *a, k, pk;
  srand(2213);
 m1:;
  clrscr();
  printf(" Введите размер массива : ");
  scanf("%d", &n);
  a=(int *)malloc(n*sizeof(int));
  printf(" Введите число разыскиваемых ключей : ");
  scanf("%d", &m);
  for(i=0; i<n; i++)
    *(a+i)=rand()%n;
  st_obmen(a, n);
  printf("\n\t Аргумент \t Позиция ");
  for(i=0; i<m; i++)
    {
     k=rand()%(n+m);
     pk=bin_poisk(a, n, k);
     printf("\n\t %6d", k);
     if(pk==-1) printf("\t\t не найден ");
      else printf("\t\t %5d", pk);
    }
  printf("\n\n Продолжим? Да - введи 7 : ");
  scanf("%d", &i);
```

```
  if(i==7) goto m1;
}
```

ЗАДАНИЕ НА ПРОГРАММИРОВАНИЕ

Задача 1

Провести сравнительный анализ эффективности методов сортировки вставками: линейной, двоичной, центрированной.

Предлагаемый тест: сортировка целочисленного массива размера n, элементы которого - случайные величины, распределённые в интервале $(0, \quad n-1)$.

Задача 2

Исследовать эффективность методов поиска: последовательного, бинарного, интерполяционного.

Предлагаемый тест: поиск m элементов в целочисленном массиве длины n.

Элементы массива - случайные величины, распределённые в интервале $(0, \quad n-1)$. Ключи поиска - случайные величины, распределённые в интервале $(0, \quad n+m-1)$.

Задача 3

Провести сравнительный анализ эффективности следующих методов сортировки:

1) линейный выбор с обменом, челночная сортировка, двоичная вставка;
2) сортировка Шелла, центрированная вставка;
3) стандартный обмен, быстрая сортировка, линейная вставка.

Предлагаемый тест: сортировка целочисленного массива размера n, элементы которого - случайные величины, распределённые в интервале $(0, \quad n-1)$.

Задача 4

Написать и протестировать функции сортировки целочисленных массивов и поиска ключей в них по следующим мктодам:

1) челночная сортировка, бинарный поиск;
2) сортировка Шелла, бинарный поиск (рекурсивная функция);
3) быстрая сортировка, бинарный поиск;
4) центрированная вставка, интерполяционный поиск.

Тест сортировки: сортировка целочисленного массива размера n, элементы которого - случайные величины, распределённые в интервале $(0,\ n-1)$.

Тест поиска: поиск m элементов в отсортированном массиве.

Задача 5

Написать и протестировать функции сортировки записей и поиска их по ключам для следующих методов:

1) линейный выбор с обменом, бинарный поиск;
2) челночная сортировка, бинарный поиск;
3) сортировка Шелла, бинарный поиск;
4) быстрая сортировка, бинарный поиск;
5) стандартный обмен, интерполяционный поиск;
6) линейная вставка, интерполяционный поиск;
7) центрированная вставка, бинарный поиск.

Запись имеет три поля, например, фамилия, имя, номер телефона. Иметь не менее 30 записей. Поиск - по любому ключу, задаваемому из меню.

Задача 6

Провести сравнительный анализ эффективности не менее двух вариантов быстрой сортировки целочисленных массивов большого размера $(n > 5000)$.

Задача 7

Написать и протестировать функцию челночной сортировки целочисленного массива с управляемым направлением (возрастание/убывание) сортировки по признаку.

Заголовок функции должен иметь вид:

*void shatl(int *a, int n, char *pr) ,*

где *a* - сортируемый массив;
n - размер массива;
pr - признак, управляющий направлением сортировки:
pr="*incr*" - сортировка по возрастанию;
pr="*decr*" - сортировка по убыванию.

Задача 8

Исследовать возможности адаптации различных методов сортировки к структуре исходного массива. С этой целью определить время сортировки целочисленного массива объёма n для следуюших вариантов представления исходшого массива:

- неупорядоченный,
- почти упорядоченный,
- упорядоченный в противоположном направлении.

Методы, подлежащие исследованию:

1) линейный выбор с обменом, центрированная вставка;
2) челночная сортировка, линейная вставка;
3) сортировка Шелла, линейная вставка;
4) быстрая сортировка, бинарная вставка;
5) стандартный обмен, бинарная вставка.

ЛИТЕРАТУРА

1. Кнут Д. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. -М.: Мир, 1978.
2. Вирт Н. Алгоритмы и структуры данных. -М.: Мир, 1989.
3. Лэнгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. -М.: Мир, 1989.
4. Сибуя М., Ямамото Т. Алгоритмы обработки данных. -М.: Мир, 1986.