1.4 Поиск и сортировка

Процедуры поиска и сортировки традиционно относятся к классическим задачам программирования. Всюду в этом разделе для определенности мы будем рассматривать сортировку по возрастанию, т.е. упорядочивание элементов массива $a[]$ так, чтобы для любого допустимого i было выполнено $a[i] \le a[i+1]$. При реализации программ поиска и сортировки можно следовать двум подходам: 1) частные программы, ориентированные на конкретный тип данных массивов; 2) универсальные программы, предназначенные для сортировки любых массивов. В первом случае для сравнения элементов между собой используются стандартные операции `<,<=,>,>=,==`. При втором подходе программа сортировки приобретает дополнительный параметр — указатель на функцию сравнения, которую можно определять отдельно для каждого требуемого типа данных.

1.142. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива считая, что массив имеет достаточную длину. Местоположение нового элемента следует определить последовательным поиском.
Решение. Пусть для определенности массив имеет тип `double`. Построенная функция будет возвращать новую длину массива.

```
int   insert (double *mas, int n, double x){
  int  i=0,j=0;
   /* ищем место для x */
   for (i=0; i<n; ){
     if(x>mas[i])i++;
     else break;
   }
/* раздвигаем массив */
   for (j=n; j>i; j--){
      mas[j]=mas[j-1];
   }
/* вставляем элемент */
   mas[i] = x;
   return n+1;
}
```

1.143. Пусть элементы массива не убывают. Требуется бинарным поиском (методом деления пополам) определить принадлежит ли массиву заданный элемент x.
Решение. Построим функцию, возвращающую 1, если элемент x присутствует в массиве и 0 в противном случае. Приведем вариант решения для массива элементов типа `int`.

```
int   search (int *mas, int n, int x){
   int  left,right,mid;
   if ( x<mas[0] || x>mas[n-1] ) return 0;
   left = 0;  right = n-1;
   if(x==mas[0]) return 1;
```

```
   while ( right-left> 1 ){
      mid = (left+right)/2;
      if ( x==mas[mid] ) return 1;
      if ( x>mas[mid] ){
         left = mid;
      }
      else{
         right = mid;
      }
   }
   return  (x==mas[right]) ? 1:0;
}
```

1.144. Измените решение предыдущей задачи так, чтобы функция `search` дополнительно определяла индекс найденного элемента в массиве либо позицию, в которую можно поставить искомый элемент, если он не присутствует в массиве.

1.145. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива. Местоположение нового элемента определить бинарным поиском.

1.146. Даны два неубывающих массива. Реализуйте функцию, строящую третий неубывающий массив, который является их объединением (т.е. содержит все элементы двух исходных массивов). Оформите решение в виде функции с прототипом

```
int   merge (double *c, double *a, double *b,  int na, int nb);
```

получающей необходимые адресса и длины na, nb исходных массивов и возвращающей длину полученного массива `c`, т.е. $na + nb$. Оцените сложность алгоритма (количество необходимых действий).

Указание. На первом шаге из двух значений $a[0], b[0]$ выбираем наименьшее и помещаем в $c[0]$. Если оказалось, что $b[0] < a[0]$, то на втором шаге процедура повторяется для $a[0]$ и $b[1]$, иначе — для $a[1]$ и $b[0]$. При этом найденный на втором шаге минимальный элемент укладывается в $c[1]$. Если на очередном шаге получили, что в одном из массивов (в $a[]$ или $b[]$) все элементы закончились, то оставшийся "хвост" второго массива просто копируется в $c[]$. Сложность алгоритма $O(na + nb)$.

В следующих задачах необходимо реализовать функцию сортировки массива вещественных чисел по возрастанию с заголовком

```
void  sort (double *a, int n);
```

1.147. "Пузырьковая" сортировка. *Идея алгоритма:* за первый проход обеспечим "подъем" наибольшего элемента из $a[0], \ldots, a[n-1]$ до позиции $a[n-1]$; далее процедуру будем повторять для подмассивов $a[0] \ldots a[n-2]$; $\cdots$; $a[0], a[1]$. *Формальное описание.* Последовательно сравниваем два соседних элемента a_i и a_{i+1} и, если оказывается, что $a_i > a_{i+1}$, то меняем

их местами. Такое сравнение и, возможно, перестановку выполняем для $i = 0, 1, \ldots, n-2$. Данную процедуру назовем подъемом до $(n-1)$-й позиции. Далее последовательно выполняем подъем до позиции $(n-1), \ldots, 1$.
Решение.

```
for(k=0;k<n-1;k++){
 for(i=0;i<n-1-k;i++){
    if(a[i]>a[i+1]){
      tmp=a[i+1]; a[i+1]=a[i];a[i]=tmp;
    }
  }
}
```

Поясните, почему верхняя граница второго цикла после каждого прохода уменьшается на единицу. Если при очередном подъеме не будет выполнено ни одной перестановки, то массив уже является упорядоченным и процесс сортировки можно прекратить. Внесите соответствующее дополнение в код программы. Итоговая сложность алгоритма $O(n^2)$ действий. Отметим, если в упорядоченном массиве наибольший элемент переставить в начало, то сортировка потебует $n-1$ сравнение и столько же перестановок. Если же в упорядоченном массиве наименьший элемент переставить в конец, то для сортировки потребуется выполнить $O(n^2)$ сравнений.

1.148. Метод перестановки максимального элемента. *Идея алгоритма:* за первый просмотр найдем максимальный элемент из $a[0], \ldots, a[n-1]$ и переставим его в $a[n-1]$; далее процедура повторяется для подмассивов $a[0], \ldots, a[n-2]$; $\cdots$; $a[0], a[1]$. *Формальное описание.* Пусть дано некоторое число k, $(0 < k < n)$. Находим максимальный элемент среди чисел $a_0, \ldots, a_k$. Пусть этим максимумом является некоторый элемент a_j. Обмениваем значения элементов a_j и a_k. Указанную процедуру последовательно выполняем для $k = n-1, n-2, \ldots, 1$. Сложность алгоритма $O(n^2)$ действий.

1.149. Сортировка вставками с последовательным поиском. *Идея алгоритма:* массив из одного элемента $a[0]$ упорядочен; добавим к нему элемент $a[1]$, сохранив общую упорядоченность; затем добавим к результату $a[2]$ и т.д.; положение добавляемого элемента определяется последовательным перебором. *Формальное описание.* Предположим, что первые k элементов массива (т.е. $a_0, \ldots, a_{k-1}$) уже упорядочены нужным образом. Тогда, выполнив упорядоченную вставку элемента a_k в этот массив (задача 1.142), мы получим упорядоченную совокупность из $k+1$ элемента. Последовательно выполняем эту процедуру для $k = 1, 2, \ldots, n-1$. Сложность алгоритма $O(n^2)$ действий.

1.150. Сортировка вставками с бинарным поиском. Алгоритм решения аналогичен предыдущему, но место для вставки нового элемента в упорядоченную часть массива отыскивается с помощью бинарного поиска (задача 1.143). Сложность алгоритма $O(n \log_2 n)$ действий.

1.151. Сортировка просеиванием (шейкерная сортировка, shakesort). *Идея алгоритма:* в пузырьковой сортировке легкий (самый большой элемент) всплывает за один проход, но тяжелые (малые элементы) за один

проход опускаются только на одну позицию; добавим процедуры быстрого спуска тяжелых элементов. *Формальное описание.* Метод является модификацией пузырьковой сортировки и состоит из двух этапов — подъема и спуска. При подъеме последовательно сравниваются соседние элементы a_i и a_{i+1} $(i = 0, 1, \ldots)$ до тех пор, пока не будет сделана первая перестановка. Пусть эта перестановка затронула элементы a_k и a_{k+1}. Следующим этапом является спуск. Новый элемент a_k сравниваются с a_{k-1} и если $a_k < a_{k-1}$, то выполняется перестановка. Сравнение продолжается в нисходящем направлении (т.е. для a_{k-1} и a_{k-2}, a_{k-2} и a_{k-3} и т.д.) до тех пор, пока *выполняются* перестановки либо достигается начало массива. После этого возобновляется подъем с позиции $i = k + 1$. Таким образом, сортировка состоит из сменяющих друг друга процессов подъема (до первой перестановки) и спуска (до первого отсутствия перестановки) до тех пор, пока при подъеме не будет затронут последний элемент массива a_{n-1} (при этом спуск также должен быть выполнен). Сложность алгоритма $O(n^2)$ действий.

1.152. Быстрая сортировка Хоара (quicksort). *Идея реализации:* в шейкерной сортировке обмены происходят только между соседними ячейками; для ускорения модифицируем алгоритм так, что станут возможны и дальние перестановки; при этом за первый проход массив станет частично упорядоченным — будет состоять из двух неупорядоченных частей так, что элементы первой части строго меньше некоторого *опорного* числа, а элементы второй — больше либо равны опорному. Опорное число подбирается (по возможности) так, чтобы длины полученных массивов были близки. Далее процедуру повторим для каждой из неупорядоченных частей, пытаясь в каждом случае оптимальным образом выбрать свой опорный элемент. *Формальное описание.* Пусть мы имеем неупорядоченный массив $a_0, \ldots, a_{n-1}$ и некоторое число m, которое удовлетворяет условию $\min\limits_i a_i \le m \le \max\limits_i a_i$. Перестроим массив так, чтобы при некотором $0 \le s \le n-1$ было выполнено $a_i \le m$ при $i \le s$ и $a_i \ge m$ при $i \ge s$. Последовательно сравнивая элементы массива a_j, $j = 0, 1, \ldots$ с m найдем первое такое значение j, что $a_j \ge m$. Теперь последовательно сравнивая элементы массива a_k, $k = n-1, n-2, \ldots$ (т.е. от конца к началу) с m мы либо найдем такое значение k, что $a_k \le m$, $k > j$, либо получим $k = j$. Если $j < k$, то обмениваем местами элементы a_j и a_k и повторяем описанную выше процедуру для элементов массива $a_{j+1}, \ldots, a_{k-1}$. Если $k = j$, то мы получили искомое разбиение массива на две требуемых части при $s = k$. Теперь рекурсивно применим описанную выше процедуру для каждой из полученных частей массива (естественно, с соответствующим новым значением m). После завершения всех рекурсивных вызовов массив будет упорядочен.

Быстродействие алгоритма сильно зависит от удачного выбора числа m, поскольку именно это число определяет где массив будет разделен на две части. Если это разделение каждый раз происходит примерно посередине, то трудоемкость сортировки составляет $O(n \log_2 n)$ операций (n — количество элементов массива), однако, если разделение каждый раз "отщепляет" только один элемент, то трудоемкость составит $O(n^2)$ операций. Неплохие

результаты получаются, когда в качестве m берется среднее арифметическое нескольких (двух – трех) элементов рассматриваемой части массива. Заметим также, что при рекурсии первым следует обрабатывать часть массива, имеющую меньшую длину так как это гарантирует, что глубина стека, хранящего рекурсивные вызовы функции не превысит $\log_2 n$. При этом не стоит рекурсивно обрабатывать массивы из одного элемента и пустые массивы.

1.153. Пирамидальная сортировка (сортировка двоичной кучей, heapsort). *Идея алгоритма:* Будем считать, что в массиве хранится двоичное дерево (см. раздел 3.3): a_0 — родительская (корневая) вершина, a_1 и a_2 — ее потомки, для родителя a_1 потомками являются a_3 и a_4, для a_2 потомки это a_5 и a_6, и.т.д.; т.е. для родителя a_i потомками являются a_{2i+1} и a_{2i+2} при всех $i = 0, \ldots, n-1$. Данное дерево называется пирамидой (сортирующим деревом, двоичной кучей), если каждый родитель не меньше, чем его потомки. В вершине такой пирамиды находится наибольший элемент массива. Исходный неотсортированный массив можно превратить в пирамиду, если двигаясь от вершины, последовательно сравнивать родителя с потомками и при необходимости менять их местами. Если для некоторой тройки "родитель — потомки" произошел обмен, то в соответствующей вершине спуск вниз следует приостановить, а процедуру проверок заново провести, поднявшись к ее родителю (и так, возможно, далее до корня дерева). Затем продолжается спуск вниз от вершины останова. В построенной таким образом пирамиде первый элемент a_0 меняется с последним a_{n-1}. В результате наибольший элемент попадает на свое место, но условие упорядоченности нарушается. Поэтому процедура перестройки повторяется для массива на единицу меньшей длины. И т.д. *Формальное описание.* Пусть мы имеем массив a_i, $i = 0, \ldots, n-1$. Будем говорить, что i-й треугольник пирамиды выстроен, если элемент a_i массива удовлетворяет условию $a_i \geq a_{2i+1}$ и $a_i \geq a_{2i+2}$ (если одно или оба значения $2i+1$ и $2i+2$ выходят за границы массива, то соответствующее неравенство считается выполненным). Процедура выстраивания i-го треугольника состоит в проверке неравенств $a_i \geq a_{2i+1}$ и $a_i \geq a_{2i+2}$ и выполнении обмена элемента a_i с максимальным из a_{2i+1}, a_{2i+2} в том случае, когда указанные неравенства нарушены.

Шаг 1. Сборка пирамиды. Выстроим 0-й треугольник. Далее для каждого $k = 3, \ldots, n-1$ будем выстраивать s-й треугольник, где s последовательно вычисляется по рекуррентной формуле $s = [k-1]/2$ и далее $s = [s-1]/2$ до $s = 0$ ([ ] — целая часть). Заметим, что если при некотором s выстраивание треугольника не приводит к перестановке, то обработку текущего значения k можно не продолжать. В результате все i-е, $i = 0, \ldots, n-1$, треугольники пирамиды будут выстроены. Заметим, что при этом элемент a_0 окажется максимальным в массиве.

Шаг 2. Разборка пирамиды. Пусть $k = n$. Обменяем местами элементы a_0 и a_{k-1}. Таким образом, максимальный элемент занял свое место в будущем упорядоченном массиве. Будем теперь считать, что длина массива равна $k-1$, и выстроим 0-й треугольник. Если при этом произошла пере-

становка, то выстроим тот треугольник, вершина которого участвовала в обмене (т.е. каждый раз выстраиваем тот треугольник, куда переместился бывший элемент a_0), и т.д. до тех пор, пока при выстраивании треугольников происходят перестановки. Напомним, что элемент a_{n-1} уже занял свое место и в выстраиваниях не участвует. Далее последовательно выполняем шаг 2 для $k = n-1, n-2, \ldots, 1$. В результате на позиции $n-2, n-3, \ldots, 0$ последовательно поступают требуемые элементы.

Нетрудно видеть, что алгоритм имеет гарантированную трудоемкость $O(n \log_2 n)$ и не требует дополнительной памяти.

Следующие алгоритмы, применяющие идею слияния учитывают, что два упорядоченных массива длины n можно объединить в один упорядоченный за n сравнений (см. задачу 1.146).

1.154. Сортировка простым слиянием. На первом шаге весь массив рассматривается как совокупность упорядоченных групп по одному элементу в каждой. Слиянием (объединением) соседних групп во вспомогательный массив мы получаем упорядоченные группы, каждая из которых содержит два элемента (кроме, может быть, последней группы, которой не нашлось парной). Далее упорядоченные группы укрупняются в исходный массив и т.д. Данный алгоритм может быть реализован с использованием рекурсивной процедуры. Разбиваем исходный массив на две половины. Если каждая из частей является упорядоченной, то после их слияния мы получим отсортированный исходный массив. Иначе, рекурсивно применяем указанную процедуру к неупорядоченным частям. По завершении всех рекурсивных вызовов мы получим упорядоченный массив. В общем случае промежуточные проверки на упорядоченность можно опустить, считая, что упорядоченными будут только подпоследовательности, состоящие из одного элемента. Сложность алгоритма $O(n \log_2 n)$ действий.

В дальнейшем предполагается, что исходный массив не помещается в оперативную память, и мы вынуждены считывать информацию по частям из некоторого файла a.dat. При необходимости мы можем разбить исходные данные на несколько частей и записать их в файлы a1.dat, a2.dat, ..., ak.dat, либо, реально работая с одним файлом, программно поддерживать многоканальное чтение. При анализе алгоритмов такого рода особое внимание стоит уделять количеству обращений к файлу.

1.155. $2k$-ленточное слияние. Пусть необходимо упорядочить массив длины n. Будем считать, что мы можем поддерживать k различных каналов на чтение и столько же на запись. На первом шаге считаем, что входные последовательности состоят из упорядоченных серий единичной длины. Считываем по одной серии из каждого входного канала, сливаем их в одну упорядоченную серию длины k и записываем в первый выходной канал. Следующую упорядоченную серию длины k записываем во второй канал, и так далее, циклически меняя номера выходных каналов. Процедура повторяется до тех пор, пока не исчерпаются все входные последовательности. Далее входные и выходные каналы меняются местами и алгоритм по-

вторяется. За каждый такой шаг мы получаем, что длина упорядоченных последовательностей увеличивается в k раз. Если общее число элементов равно $n = k^p$, то за $\log_k n$ шагов мы получим упорядоченный массив. Число пересылок при этом $O(n \log_k n)$. При произвольном n входные последовательности будут исчерпываться с различной скоростью и на каждом шаге мы будем реально сливать $k_i \leq k$ последовательностей различной длины.

1.156. Естественное слияние. В случае прямого слияния мы не получаем выигрыша, если исходный массив был частично упорядочен. Естественным обобщением является алгоритм, когда объединяем максимальные упорядоченные серии, а не последовательности фиксированной длины.

1.157. Многофазная сортировка. Откажемся от идеи, что у нас есть k входных и столько же выходных последовательностей одновременно. Будем считать, что в каждый момент имеется только один выходной канал и работать так, что если какой-то из $2k - 1$ входных массивов опустел, то он становится выходным. Данный алгоритм полезен, когда входные последовательности имеют существенно разные длины. В лучшем случае число пересылок порядка $O(n \log_{2k-1} n)$.

1.158. Отсортировать массив целых чисел по младшей цифре посредством библиотечной функции

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

из стандартной библиотеки `stdlib.h`, Здесь первый аргумент `base` указывает на начало массива, второй — содержит его длину, третий — размер в байтах одного элемента массива, четвертый является указателем на функцию, позволяющую сравнивать элементы массива. Функция сравнения должна возвращать 0 для равных элементов, произвольное целое отрицательное число, если первый элемент меньше второго, и произвольное положительное число, если первый элемент больше второго. Если два элемента массива равны (в смысле функции сравнения), то их порядок в отсортированном массиве неопределен.

Решение.

```
#include <stdlib.h>
int cmp(const void * a, const void * b);
int main(){
  int  * mas = {123, -101, 1000};
  qsort((void*)mas, 3, sizeof(int), cmp);
 return 0;
}
int cmp(const void * a, const void * b) {
    int ia = *(int *)a;
    int ib = *(int *)b;
    return ia%10 - ib%10;
    }
```

В результате получим 1000, -101, 123.

1.159. Отсортировать массив целых чисел с помощью библиотечной функции

```
void * bsearch( const void * search,
             const void * base, size_t nmeber, size_t size,
            int (*compar)(const void *,const void *));
```

из стандартной библиотеки `stdlib.h`, выполняющей бинарный поиск в массиве.

1.5 Задачи на работу с символьными переменными

При работе с символами на ЭВМ необходимо помнить, что в памяти компьютера хранится не изображение символа, а его целочисленный код. Как именно происходит восстановление внешнего вида символа по его коду зависит от типа кодирования. Наиболее простым является однобайтовое кодирование символов на основе табицы ASCII (American Standart Code Information Interchange). Символы с нулевого по 127 ASCII-таблицы фиксированы. Это связано с тем, что ранее стандартом была семибитная таблица ASCII-7, использовавшаяся для кодирования телефонограмм — код символа пробивался на бумажной ленте. При этом символы с 0-го по 31-й являются управляющими непечатными символами, например, 07 - звонок, 08 - шаг назад (BackSpace), 10 - перевод строки, 13 - возврат каретки. Вторая часть таблицы, т.е. начиная с 32-го символа (пробел) по 127-й символ (т.н. del-забой — в двоичном представлении семь единиц, что пробивалось на ленте как семь отверстий), содержит знаки препинания, цифры, заглавные и строчные буквы английского алфавита. При этом цифры '0',. . . ,'9' имеют коды с 48 по 57, буквы 'A', . . . , 'Z' — с 65 по 90, буквы 'a', . . . , 'z' — с 97 по 122. Отметим, что после '0' идет '1','2' и т.д., аналогично для заглавных 'A','B', . . . и строчных 'a','b', . . . букв. Третья часть, т.е. с 128-го символа по 255-й, зависит от конкретной реализации таблицы — типа выбранной кодовой страницы. К наиболее известным кириллическим таблицам относятся cp1251 (Windows-кодировка), koi-8, koi-8r, koi-8u (Linux-кодировки) и cp866 (DOS-кодировка). Файл, содержащий кириллицу, при неправильно указанной кодировке будет отображаться "кроказябрами". Для решения подобных проблем, актуальных для большинства национальных алфавитов, на данный момент установлен стандарт двухбайтового кодирования UNICODE, описание формата которого можно найти в соответствующей литературе. Большую популярность на данный момент имеет кодировка UTF-8 (Unicode Transformation Format, 8-bit), основанная на UNICODE–таблице, но использующая по необходимости для хранения символа от одного до 4-х байт. Какое именно количество байт кодирует текущий символ определяется по старшим битами, например, байты вида (0*******) кодируют символы с кодом от 0 до 127 из первой части таблицы ASCII. Отметим, что текстовый