ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Под сортировкой обычно понимают процесс перестановки объектов данного множества в определенном порядке. Цель сортировки – облегчить последующий поиск элементов в отсортированном множестве. В этом смысле элементы сортировки присутствуют почти во всех задачах. Упорядоченные объекты содержатся в телефонных книгах, в ведомостях подоходных налогов, в оглавлениях, в библиотеках, в словарях, на складах, да и почти всюду, где их нужно разыскивать. Даже маленьких детей приучают приводить вещи «в порядок», и они сталкиваются с некоторым видом сортировки задолго до того, как узнают что-либо об арифметике.

Сортировка служит хорошим примером того, что одна и та же цель может достигаться с помощью различных алгоритмов, причем каждый из них имеет свои определенные преимущества и недостатки, которые нужно оценить с точки зрения конкретной ситуации. Многие из алгоритмов в некотором смысле являются оптимальными. Поэтому на примере сортировки мы убеждаемся в необходимости сравнительного анализа алгоритмов. Кроме того, здесь мы увидим, как при помощи усложнения алгоритмов можно добиться значительного увеличения эффективности по сравнению с более простыми и очевидными методами.

Зависимость выбора алгоритмов от структуры данных – явление довольно частое, и в случае сортировки она настолько сильна, что методы сортировки обычно разделяют на две категории: сортировка массивов и сортировка (последовательных) файлов. Эти два класса часто называют внутренней и внешней сортировкой, так как массивы располагаются во «внутренней» (оперативной) памяти ЭВМ; для этой памяти характерен быстрый произвольный доступ, а файлы хранятся в более медленной, но более вместительной «внешней» памяти, т. е. на запоминающих устройствах с механическим передвижением (дисках и лентах). Это существенное различие можно наглядно показать на примере сортировки пронумерованных карточек [1]. Представление карточек в виде массива соответствует тому, что все они располагаются перед сортирующим так, что каждая карточка видна и доступна. Представление карточек в виде файла предполагает, что видна только верхняя карточка из каждой стопки. Очевидно, что такое ограничение приведет к существенному изменению методов сортировки, но оно неизбежно, если карточек так много, что их число на столе не уменьшается.

Прежде всего, мы введем некоторую терминологию и систему обозначений, которые будем использовать. Нам даны элементы:

$$a_1,\ a_2,\ a_3,\ \ldots a_n;$$

Сортировка означает перестановку этих элементов в таком порядке:

$$a_{k1},\ a_{k2},\ a_{k3},\ \ldots\ a_{kn};$$

что при заданной функции упорядочения f справедливо отношение:

$$f(a_{k1}) \leq f(a_{k2}) \leq f(a_{k3}) \leq \ldots \leq f(a_{kn});$$

Обычно функция упорядочения не вычисляется по какому-то специальному правилу, а содержится в каждом элементе в виде явной компоненты (поля). Ее значение называется ключом элемента. Для представления элемента a, особенно хорошо подходит представление данных – структура. В данном пособии для простоты будем считать функцией упорядочения тождественную функцию, т.е. ключом для сортировки будут являться сами элементы, а порядок сортировки – по возрастанию.

Тип ключа (а в данном случае тип элемента) может быть произвольным, но обязательно у такого типа должно быть задано отношение всеобщего порядка.

Метод сортировки называется устойчивым, если относительный порядок элементов с одинаковыми ключами не меняется при сортировке. Устойчивость сортировки часто бывает желательна, если элементы упорядочены (рассортированы) по каким-то вторичным ключам, т. е. по свойствам, не отраженным в первичном ключе.

СОРТИРОВКА МАССИВОВ

Основное требование к методам сортировки массивов – экономное использование памяти. Это означает, что переупорядочение элементов нужно выполнять на том же месте. Методы, которые пересылают элементы из массива A в массив B, не представляют для нас интереса. Таким образом, выбирая метод сортировки, руководствуясь критерием экономии памяти, классификацию алгоритмов проводят в соответствии с их эффективностью, т. е. экономией времени или быстродействием. Удобная мера эффективности получается при подсчете числа C – необходимых сравнений ключей и M – пересылок элементов. Эти числа определяются некоторыми функциями от числа N сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n*ln(n)$ сравнений, рассмотрим несколько несложных и очевидных способов сортировки, называемых простыми методами, которые требуют порядка n^2 сравнений ключей.

Простые методы особенно хорошо подходят для разъяснения свойств большинства принципов сортировки. Программы, основанные на этих методах, легки для понимания и коротки. Хотя сложные методы требуют меньшего числа операций, эти операции более сложные, поэтому, при достаточно малых n простые методы работают быстрее, но их не следует использовать при больших n.

Методы сортировки массивов можно разбить на три основных класса в зависимости от лежащего в их основе приема:

Сортировка включениями (вставкой).

Сортировка выбором.

Сортировка обменом.

Рассмотрим и сравним эти три принципа. На рисунке *Программа 1* показан пример основной программы для тестирования методов сортировки и вспомогательные функции заполнения и отображения массивов. В программе объявляется переменная - одномерный массив *a[n]*, где размерность задается константой *n*. Все основные действия реализованы с помощью функций. Передача массива в функции выполняется по адресу двумя способами *int a[]* и *int *a*, размерность массива передается через параметр *int size* . Выбор способа заполнения массива, и метода сортировки осуществляется в функции *main*. Тестирование различных методов сортировки полезно проводить на массивах с различным способом заполнения, а именно: на случайно заполненных, изначально отсортированных, отсортированных в обратном порядке. Для этого реализованы соответствующие функции заполнения.

Программа 1. Функция *main* и вспомогательные функции для тестирования методов сортировки

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

////////////////////////////////////////////////////////////////////////////////////////////////
//прототипы функций заполнения и вывода массива
// первый параметр int* указатель для передачи начального адреса массива
// второй параметр int целая переменная для передачи значения размерности массива

// процедура заполнение массива значениями по возрастанию
void Mas_Init_Up(int*, int);

// процедура заполнение массива значениями по убыванию
void Mas_Init_Down(int*, int);

// процедура заполнение массива псевдослучайными значениями
void Mas_Init_Rand(int*, int);

//процедура вывода массива на экран
void Mas_Print(int*, int);

////////////////////////////////////////////////////////////////////////////////////////////////
//прототипы функций сортировки массива
// первый параметр int[] указатель для передачи начального адреса массива
// второй параметр int целая переменная для передачи значения размерности массива

void SortInsert (int[], int); //сортировка вставкой (включениями)
void SortInsertBin (int[], int);//сортировка бинарными включениями
void SortSelect (int[], int);// сортировка простым выбором
void SortBubble(int[], int);// сортировка простым обменом
void ShakerSort(int[], int);// шейкер сортировка обменом
void QuickSort(int[], int, int); //сортировка разделением (быстрая сортировка)
int BinarySearch (int*, int, int);// функция бинарного поиска

// функция сравнения элементов массива для процедуры быстрой сортировки
qsort стандартной библиотеки C++ stdlib
int compare( const void *arg1, const void *arg2 );
int compareAlt( const void *arg1, const void *arg2 );
```

```
//объявление глобальной константы для задания размерности массива

const int n = 100;

////////////////////////////////////////// m a i n//////////////////////////////////////////
void main()
{
   //объявление одномерного массива a из целых чисел на 100 элементов
   int array[n];
   // объявление служебной переменной для выбора режимов заполнения и
сортировки
   int selection = 0;
   // вывод меню пользователя
   printf("\nSelect array filling mode\n1 - Up\n2 - Down\n3 - Random\n\n");
   //считывание значения выбора без отображения с условием равенства 1 - 3
   while (selection != '1'&& selection != '2'&&selection != '3') selection =_getch();
   //выбор процедуры заполнения
   switch(selection)
   {
      case '1': Mas_Init_Up(array,n); break;
      case '2': Mas_Init_Down(array,n); break;
      case '3': Mas_Init_Rand(array,n); break;
   }
   //вывод исходного массива на экран
   printf("Initial array:\n");
   Mas_Print(array,n);
   // вывод меню пользователя
   printf("\nSelect array sorting mode\n1 - Insert Sort\n2 - Binary Insert Sort\n3 -
Selection Sort\n");
   printf("4 - Bubble Sort\n5 - Shaker Sort\n6 - Quick Sort\n7 - C++ QSort\n\n");
   //считывание значения выбора с условием равенства от 1 до 7
   selection = 0;
   while (selection < '1' || selection > '7') selection =_getch();
   //выбор процедуры сортировки
   switch(selection)
   {
      case '1': SortInsert (array,n); break;
      case '2': SortInsertBin (array,n); break;
      case '3': SortSelect (array,n); break;
      case '4': SortBubble (array,n); break;
      case '5': ShakerSort (array,n); break;
      case '6': { printf("Quick Sort Mode:\n"); QuickSort(array, 0,n -1);} break;
      case '7':
                {
                   printf("C++ Quick Sort Mode :\n");
                   qsort(array,n,sizeof(int),compare);
                } break;
   }
```

```
    printf("Sorted array:\n"); //вывод отсортированного массива на экран
    Mas_Print(array,n);

    // демострация алгоритма бинарного поиска
    printf("Binary Search test\n Enter number N = ");
    int N_to_Find = 0; //объявление и считывание переменной ключа поиска
    scanf("%d",&N_to_Find);
    // вызов процедуры бинарного поиска
    int res = BinarySearch(&array[0],n,N_to_Find);
    // вывод результата поиска на экран
    if (res!=-1) printf("index %d = %d\n",N_to_Find, res);
    else printf("%d not in mas\n", N_to_Find);
} // конец main

////////////////////////////////////////////////////////////////////////////////////////////////
// процедура заполнение массива значениями по возрастанию
void Mas_Init_Up(int *a, int size)
{
    for (int i=0;i<size;i++) a[i] = i; // элементу присваиваем его индекс
    printf("Array filled up\n\n");// вывод комментария
}
////////////////////////////////////////////////////////////////////////////////////////////////
// процедура заполнение массива значениями по убыванию
void Mas_Init_Down(int *a, int size)
{
    // элементу присваиваем размерность - индекс
    for (int i=0;i<size;i++) a[i] = size - i;
    printf("Array filled down\n\n");
}
////////////////////////////////////////////////////////////////////////////////////////////////
// процедура заполнение массива псевдослучайными значениями
void Mas_Init_Rand(int *a, int size)
{
    // инициализация случайного счетчика из системных часов
    srand( (unsigned)time( NULL ));
    // a[i] = случайное целое число в диапазоне от 0 до 99
    for (int i=0;i<size;i++) a[i]= rand()%100;
    printf("Array filled random\n\n");
}
////////////////////////////////////////////////////////////////////////////////////////////////
//процедура вывода массива на экран
void Mas_Print(int *a, int size)
{
    //последовательный вывод элементов массива на экран в формате
    // 4 символа на элемент + выравнивание по левому краю
    for (int i=0;i<size;i++)  printf("%-4d",a[i]);
    printf("\n");
}
```

СОРТИРОВКА ПРОСТЫМИ ВКЛЮЧЕНИЯМИ.

Этот метод обычно используют игроки в карты. Процесс сортировки включениями показан на примере восьми случайно взятых чисел (см. рис. 1).

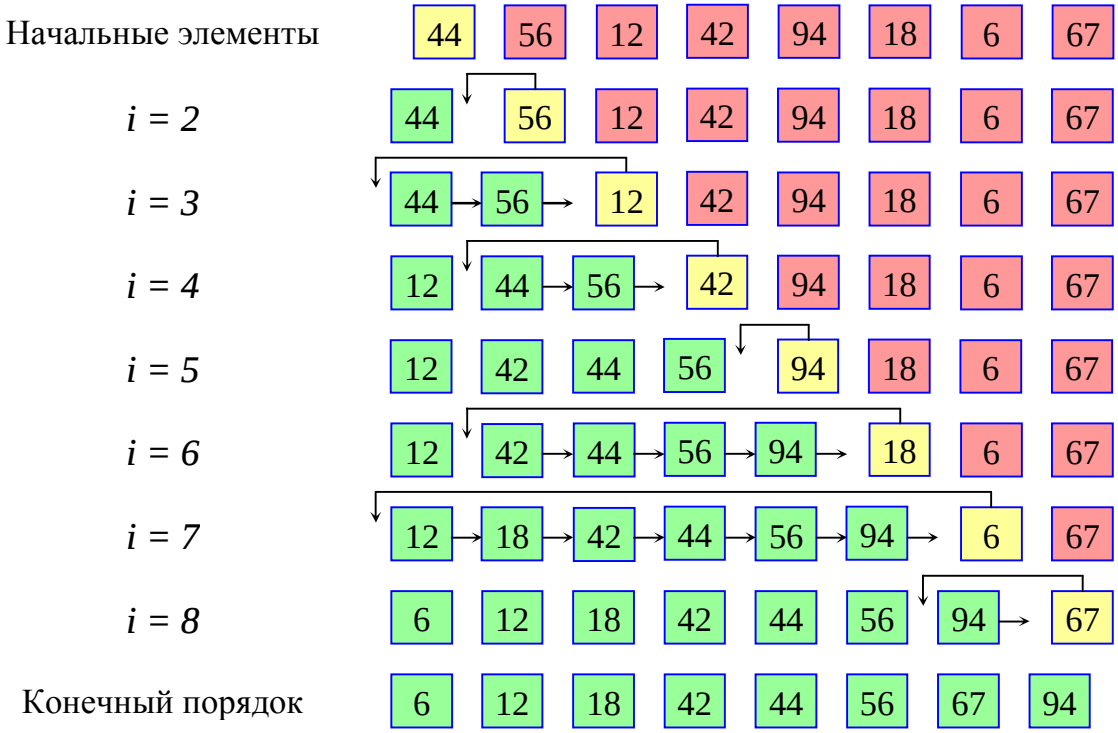

Рис. 1. Пример сортировки простыми включениями: – отсортированная часть, – входная часть, - текущий элемент.

Элементы условно разделяются на готовую (отсортированную) последовательность $\{a_1 \dots a_{i-1}\}$ и входную последовательность $\{a_i \dots a_n\}$. На каждом шаге, начиная с $i = 2$ и увеличивая i на единицу, берут i-й элемент входной последовательности и передают в готовую последовательность, вставляя его на подходящее место. Алгоритм сортировки массива простыми включениями выглядит следующим образом (см. рис. программа 2):

Программа 2. Алгоритм сортировки простыми включениями на псевдокоде.

1. Для всех **i** от **1** до **n** выполнить:
- **1.1.** Взять очередной **i**-й не отсортированный элемент и сохранить его в рабочей переменной **x;**
- **1.2.** Найти позицию **j** в отсортированной **(0…i-1)** части массива, в которой присутствие взятого элемента не нарушит упорядоченности элементов
- **1.3.** Сдвиг элементов массива от **i-1** до **j-1** вправо, чтобы освободить найденную позицию вставки
- **1.4.** Вставка взятого элемента **x** в найденную **j**-ю позицию.

Данный алгоритм можно оптимизировать, скомбинировав шаги 1.2 и 1.3, т.е. при поиске подходящего места вставки чередовать сравнения и пересылки как бы «просеивая» x, сравнивая его с очередным элементом a, и затем либо вставляя x, либо пересылая a_j направо и продвигаясь налево. Просеивание может закончиться при двух различных условиях:

1. Найден элемент a с ключом меньшим, чем ключ x.
2. Достигнут левый конец готовой последовательности.

Процедура сортировки простыми включениями представлена на рисунке программа 3.

Программа 3. Процедура сортировки простыми включениями.

```
void InsertSort (int a[],int size)
{
//последовательный перебор не отсортированных элементов массива.
   for (int i=1;i< size;i++)
   {
         int x = a[i]; //взятие очередного элемента
         int j = i-1;
         while (x<a[j]) // повторять пока место вставки не найдено
         {// сдвиг текущего j- го элемента на 1 позицию вправо
               a[j+1] = a[j];
               j--;
               if (j<0) break; //условие выхода при достижении левой границы
         }
         a[j+1] = x; // вставка взятого элемента
   }
}
```

Анализ сортировки простыми включениями. Число C_i сравнений ключей при i-м просеивании составляет самое большее $i - 1$, самое меньшее 1 и, если предположить, что все перестановки n ключей равновероятны, в среднем равно $i/2$. Число M_i пересылок (присваиваний) равно $C_i + 1$. Общее число сравнений и пересылок есть:

$$
\begin{aligned}
C_{\min} &= n-1; & M_{\min} &= 2(n-1);\\
C_{cp} &= \frac{1}{4}(n^2+n-2); & M_{cp} &= \frac{1}{4}(n^2+9n-10);\\
C_{\max} &= \frac{1}{2}(n^2+n)-1; & M_{\max} &= \frac{1}{2}(n^2+3n-4);
\end{aligned}
\qquad (1)
$$

Наименьшие числа появляются, если элементы с самого начала упорядочены, а наихудший случай встречается, если элементы расположены в обратном порядке. В этом смысле сортировка включениями демонстрирует вполне естественное поведение. Ясно также, что данный алгоритм описывает устойчивую сортировку: он оставляет неизменным порядок элементов с одинаковыми ключами.

Алгоритм сортировки простыми включениями легко можно улучшить, пользуясь тем, что готовая последовательность a_1, ..., a_{i-1} в которую нужно включить новый элемент, уже упорядочена. Место включения можно найти значительно быстрее, применив бинарный или двоичный поиск.

Здесь стоит отвлечься от сортировки и рассмотреть алгоритм двоичного поиска. При двоичном алгоритме поиска после каждого сравнения исключается половина элементов массива, в котором проводится поиск. Алгоритм находит средний элемент массива и сравнивает его с ключом поиска. Если они равны, ключ поиска считается найденным и возвращается индекс этого элемента. Если они не равны, задача упрощается до поиска в одной из половин массива, в зависимости от отношения значений ключа поиска и среднего элемента.

Если ключ поиска меньше среднего элемента массива, поиск производится в первой половине массива, в противном случае поиск производится во второй половине. Поиск проводится таким же образом, как и в первоначальном массиве, т.е. выделением среднего элемента и сравнением с ним ключа поиска. Если ключ поиска в указанной части первоначального массива не найден, алгоритм повторяется для четверти массива, далее для восьмой части, шестнадцатой и т.д. Поиск продолжается до тех пор, пока ключ не окажется равен среднему элементу подмассива или пока подмассив не будет состоять только из одного элемента, не равного ключу. Это означает, что ключ поиска не найден.

В наихудшем случае двоичный поиск в массиве из 1024 элементов потребует только 10 сравнений. Последовательное деление 1024 на 2 дает значения 512, 256, 128, 64, 32, 16, 8, 4, 2 и 1. Число 1024 (2^{10}) до получения значения, равного 1, делится на 2 только десять раз. Деление на 2 соответствует одному сравнению в алгоритме двоичного поиска. Чтобы найти ключ в массиве из 1048576 (2^{20}) элементов, потребуется максимум 20 сравнений. В массиве из миллиарда элементов потребуется максимум 30 сравнений. Это является огромным увеличением эффективности в сравнении с последовательным перебором, который требует в среднем сравнения ключа поиска с половиной элементов массива. Для массива из миллиарда элементов разница между 500 миллионами сравнений (в среднем) и максимум 30 сравнениями очевидна! Максимальное число сравнений для любого массива можно определить путем нахождения первой степени 2, превосходящей число элементов в массиве.

Программа 4. Алгоритм бинарного поиска в массиве размера n

1. Задать ключ поиска **key;**
2. Положить **left=0, right=n-1;**
3. Повторять пока **left < right**
   - 3.1. Найти индекс среднего элемента в массиве
     mid = (left+right)/2

3.2. Если значение ключа поиска > элемента массива с индексом **mid**,
положить **right = mid-1**,
иначе если значение ключа < элемента массива с индексом **mid**,
положить **left = mid+1**,
иначе выйти из цикла (элемент равный ключу найден, его индекс = **mid**);

4. Если значение ключа = элементу с индексом **mid**, поиск успешен, иначе – искомого элемента в массиве нет.

На рисунке Программа 5 представлена итеративная версия функции *BinarySearch*. Функция принимает три параметра: массив через указатель на целое – *Mas*, размерность этого массива – *size*, целочисленную переменную *key* – ключ поиска, Функция возвращает целочисленный индекс найденного элемента или значение *–1* если заданный ключ поиска не обнаружен. Задаем переменные *left* и *right* для обозначения текущих границ подмассива для поиска. Далее организуется цикл *while* до тех пор, пока подмассив не сузится до одного элемента или не найдется элемент равный ключу. Если ключ поиска не соответствует среднему элементу подмассива, один из индексов *left* или *right* изменяется таким образом, чтобы поиск можно было проводить в меньшем подмассиве. Если ключ меньше среднего элемента, индексу *right* присваивается значение *mid - 1* и поиск продолжается для элементов от *left* до *mid-1*. Если ключ больше среднего элемента, индексу *left* присваивается значение *mid+1* и поиск продолжается для элементов от *mid+1* до *right*.

Программа 5. Реализация процедуры бинарного поиска.

```
int BinarySearch (int *Mas, int size, int key)
{
   int left = 0, right = size, mid=0; //первоначальные границы - весь массив
   while (left <= right) // повторять пока подмассив не сузится до 1 элемента
   {
      mid = (left+right)/2; // расчет индекса среднего элмента
      if (key < Mas[mid]) right = mid-1; //выбор нужного подмассива
      else if (key > Mas[mid]) left = mid+1;
            else break; // элемент найден
   }
   if (Mas[mid] == key) return mid; // возвращение результата
   else return -1;
}
```

Теперь стоит вернуться назад к сортировке. Модифицированная программная реализация алгоритма сортировки включениями с бинарными поиском места вставки текущего элемента показана на рисунке Программа 6 ниже.

Программа 6. Процедура сортировки массива бинарной вставкой.

```
void SortInsertBin(int a[],int size)
{
   for (int i=1;i< size;i++) //перебор всех элементов начиная со второго
   {
      int x = a[i]; // присвоение переменной x текущего элемента
      int left = 0; //установка текущих границы для поиска
      int right = i-1;

      while (left<=right) //поиск места вставки бинарным поиском
      {
         int mid = (left+right)/2;
         if (x< a[mid]) right = mid - 1;
         else left = mid + 1;
      }
      // сдвиг элементов после найденного места на 1 позицию вправо
      for (int j = i-1; j>=left; j--) a[j+1] = a[j];
      a[left] = x; //вставка текущего элемента в найденную позицию
   }
}
```

Анализ сортировки бинарными включениями. Место включения найдено, если $a[left] < x < a[right]$. Интервал поиска в конце должен быть равен 1. Это означает, что интервал из i ключей делится пополам $log_2 i$ раз. Таким образом, количество сравнений C:

$$C = \sum_{i=1}^{n} \log_2 i \qquad (2)$$

Аппроксимировав эту сумму с помощью интеграла получаем:

$$\int_1^n \log_2 x dx = x(\log_2 x - c)\Big|_1^n = n(\log_2 n - c) + c, \quad где\ c = \log_2 e = 1/\ln 2 = 1.44269 \qquad (3)$$

Количество сравнений не зависит от исходного порядка элементов. Но из-за округления при делении интервала поиска пополам действительное число сравнений для i элементов может быть на 1 больше ожидаемого. Природа этого «перекоса» такова, что в результате места включения в нижней части находятся в среднем несколько быстрее, чем в верхней части. Это дает преимущество в тех случаях, когда элементы изначально далеки от правильного порядка. На самом же деле минимальное число сравнений требуется, если элементы вначале расположены в обратном порядке, а максимальное – если они уже упорядочены. Следовательно, это случай неестественного поведения алгоритма сортировки:

$$C = n(\log_2 n - \log_2 e) \pm 0.5 \qquad (4)$$

К сожалению, улучшение, которое получается, используя метод бинарного поиска, касается только числа сравнений, а не числа необходимых пересылок. В действительности, поскольку пересылка элементов, т. е. ключей и сопутствующей информации, обычно требует значительно больше времени, чем сравнение двух ключей, то это улучшение ни в коей мере не является решающим: важный показатель M по-прежнему остается порядка n^2. И в самом деле, пересортировка уже рассортированного массива занимает больше времени, чем: при сортировке простыми включениями с последовательным поиском! Этот пример показывает, что «очевидное улучшение» часто оказывается намного менее существенным, чем кажется вначале, и в некоторых случаях (которые действительно встречаются) может на самом деле оказаться ухудшением. В конечном счете, сортировка включениями оказывается не очень подходящим методом для вычислительных машин: включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна. Лучших результатов можно ожидать от метода, при котором пересылки элементов выполняются только для отдельных элементов и на большие расстояния. Эта мысль приводит к сортировке выбором.

Некоторое усовершенствование сортировки простыми включениями было предложено Д. Л. Шеллом в 1959 г. На первом проходе отдельно группируются и сортируются все элементы, отстоящие друг от друга на четыре позиции. Этот процесс называется 4-сортировкой. После этого элементы вновь объединяются в группы с элементами, отстоящими друг от друга на две позиции, и сортируются заново. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Сначала может показаться, что необходимость нескольких проходов сортировки, в каждом из которых участвуют все элементы, больше работы потребует, чем сэкономит. Однако на каждом шаге сортировки либо участвует сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок.

Очевидно, что этот метод в результате дает упорядоченный массив, и также совершенно ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая i-сортировка объединяет две группы, рассортированные предыдущей $2i$-сортировкой. Приемлема любая последовательность приращений, лишь бы последнее было равно 1, так как в худшем случае вся работа будет выполняться на последнем проходе. Однако менее очевидно, что метод убывающего приращения дает даже лучшие результаты, когда приращения не являются степенями двойки.

СОРТИРОВКА ПРОСТЫМ ВЫБОРОМ

Этот метод основан на следующем правиле;

Из всего массива *[0..n-1]* выбирается элемент с наименьшим ключом (в нашем случае минимальный). Он меняется местами с первым элементом массива *a[0]*. Далее выбирается минимальный элемент из подмассива *[1..n-1]* и меняется местами с *a[1]*, далее минимальный из *[2..n-1]* и меняется местами с *a[2]*, и т.д. Эти операции повторяются до тех пор, пока не останется только один элемент – наибольший. Этот метод продемонстрирован на рисунке 2.

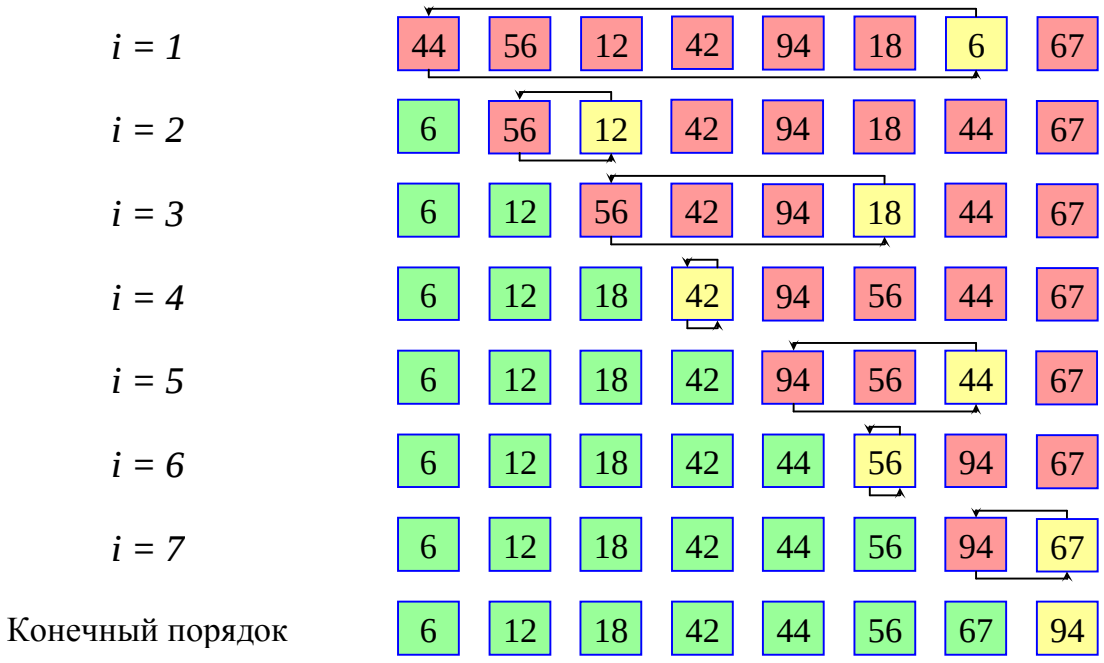

Рис. 2. Пример сортировки выбором: – отсортированная часть, – входная часть, – текущий минимум входной (не отсортированной) части

Этот метод, называемый сортировкой простым выбором, в некотором смысле противоположен сортировке простыми включениями; при сортировке простыми включениями на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готового массива для нахождения места включения; при сортировке простым выбором рассматриваются все элементы входного массива для нахождения элемента с наименьшим ключом, и этот один очередной элемент отправляется в готовую последовательность. Алгоритм и процедура сортировки массива *a[n]* простым выбором представлены на рисунке Программа 7 ниже:

Программа 7. Алгоритм сортировки выбором массива [0..n-1] на псевдокоде.

1. Для всех **i** от **0** до **n-2** выполнить**:**

1.1. Взять очередной **i**-й не отсортированный элемент и сохранить его в рабочей переменной **min**

1.2. Присвоить переменной **min_pos** значение **i;**

1.3. Найти минимум в части массива от **a[i+1]** до **a[n-1]** и запомнить его позицию в переменной **min_pos;**

1.4. Поменять местами элементы **a[i]** и **a [min_pos];**

Процедура сортировки простым выбором.

```
void SortSelect (int a[],int size)
{
   printf("Selection Sort Mode:\n");
// последовательный перебор всех элементов кроме последнего
for (int i=0;i<size-1;i++)
   {
      int min = a[i]; //присвоение переменной минимум текущего элемента
      int index_min = i; // запоминаем индекс текущего элемента
      // поиск минимума в части массива от i+1 до конца
      for (int j=i+1;j< size;j++)
      if (a[j] < min)
      {
         min = a[j]; // запоминаем текущий найденный минимум
         index_min = j; // запоминаем его индекс
      }
      //обмен местами текущего элемента и найденного минимального
      a[index_min] = a[i];
      a[i] = min;
   }
}
```

Анализ сортировки простым выбором. Очевидно, что число C сравнений ключей не зависит от начального порядка ключей. В этом смысле можно сказать, что сортировка простым выбором ведет себя менее естественно, чем сортировка простыми включениями. Мы получаем:

$$C = \frac{1}{2}(n^2 - n) \qquad (5)$$

Минимальное число пересылок M равно:

$$M_{\min} = 3(n-1) \qquad (6)$$

в случае изначально упорядоченных ключей и принимает наибольшее значение:

$$M_{\max} = \left[\frac{n^2}{4}\right] + 3(n-1) \qquad (7)$$

Если вначале ключи расположены в обратном порядке. Среднее $M_{ср}$. трудно определить, несмотря на простоту алгоритма. Оно зависит от того, сколько раз определяется, что k_j меньше всех предшествующих величин $k_1, ..., k_{j-1}$ при просмотре последовательности чисел $k_1, ..., k_n$. Это значение, взятое в среднем для всех перестановок n ключей, число которых равно $n!$, есть

$$H_n - 1$$

где H_n – n-е гармоническое число

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \mathbf{K} + \frac{1}{n} \qquad (8)$$

Число H_n можно выразить как

$$H_n = \ln n + g + \frac{1}{2n} - \frac{1}{12n^2} + \mathbf{K} \qquad (9)$$

где $\gamma = 0.577216...$ – эйлерова константа. Для достаточно больших n мы можем опустить дробные слагаемые и, таким образом, аппроксимировать среднее число присваиваний на i-м проходе следующим образом:

$$F_i = \ln i + g + 1 \qquad (10)$$

Тогда среднее число пересылок $M_{ср.}$ при сортировке выбором есть сумма F_i, где i принимает значения от 1 до n:

$$M_{ср} = \sum_{i=1}^{n} F_i = n(g+1) + \sum_{i=1}^{n} \ln i \qquad (11)$$

Аппроксимируя далее сумму отдельных слагаемых с помощью интеграла

$$\int_1^n \ln x dx = x(\ln x - 1)\Big|_1^n = n \ln n - n + 1$$

получаем приближенное значение

$$M_{ср} = n(\ln n + g) \qquad (12)$$

Можно сделать вывод, что обычно алгоритм сортировки простым выбором предпочтительней алгоритма сортировки простыми включениями, хотя в случае, когда ключи заранее рассортированы или почти рассортированы, сортировка простыми включениями все же работает несколько быстрее.

СОРТИРОВКА ПРОСТЫМ ОБМЕНОМ

Классификация методов сортировки не всегда четко определена. Оба представленных ранее метода можно рассматривать как сортировку обменом. Однако в этом разделе мы остановимся на методе, в котором обмен двух элементов является основной характеристикой процесса. Приведенный ниже алгоритм сортировки простым обменом основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут рассортированы все элементы.

Как и в предыдущих методах простого выбора, мы совершаем повторные проходы по массиву, каждый раз просеивая наименьший элемент оставшегося множества, двигаясь к левому концу массива. Если мы будем рассматривать массив, расположенный вертикально, а не горизонтально и представим себе элементы пузырьками в емкости с водой, обладающими «весами», соответствующими их ключам, то каждый проход по массиву приводит к «всплыванию» пузырька на соответствующий его весу уровень (см. рис. 3).

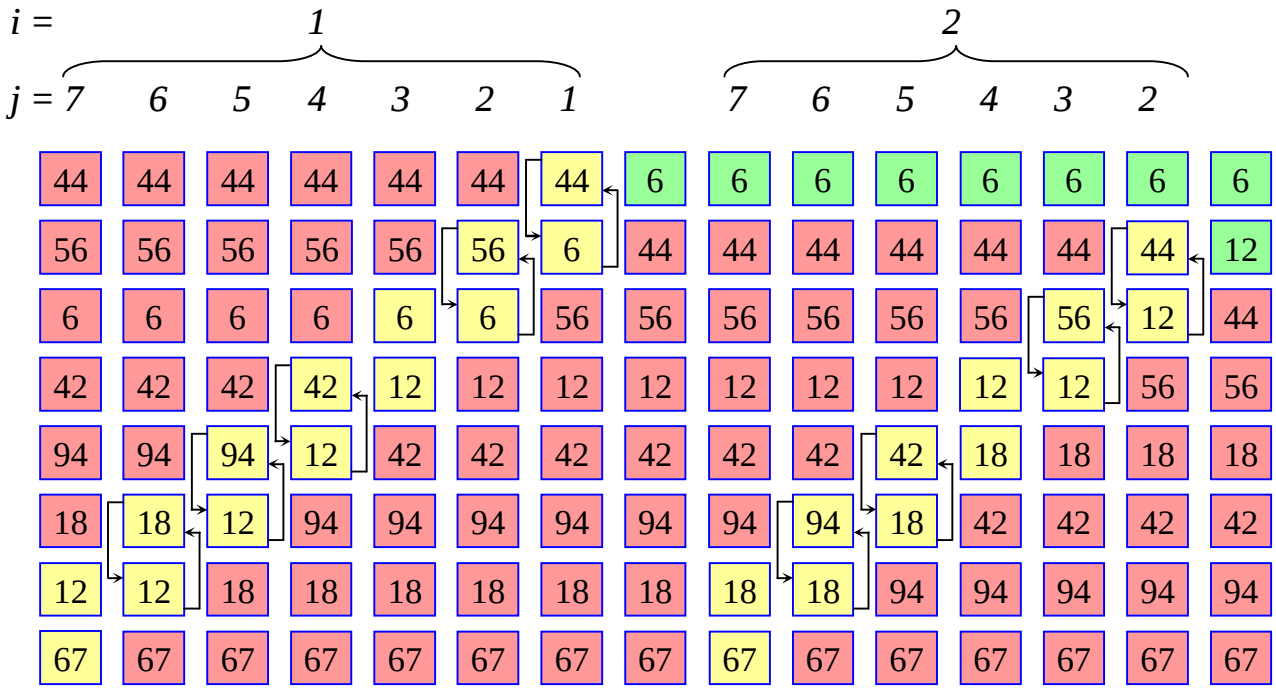

Рис. 3. Пример двух проходов сортировки методом пузырька: ■ – отсортированная часть, ■ – входная часть, ■ – пара сравниваемых элементов.

Этот метод широко известен как сортировка методом пузырька. Его простейший алгоритм и реализация приведены в Программе 8.

Программа 8. Сортировка обменом или методом пузырька.

1. Для всех **i** от **1** до **n-1** выполнять :

1.1. Слева направо поочередно сравнивать два соседних элемента и если их взаиморасположение не соответствует заданному условию упорядоченности, то менять их местами.

```
void SortBubble(int a[], int size)
{
   printf("Bubble Sort Mode:\n");
   for(int i=1;i< size;i++) //повторять проходы по массиву n-1 раз
   {
      for (int j= size -1; j >= i; j--) //проход с n – 1-го элемента вверх до i-го
         if (a[j-1]>a[j])
         {
            int x = a[j-1]; // обмен элементов в случае неправильного порядка
            a[j-1] = a[j];
            a[j] = x;
         }
   }
}
```

Этот алгоритм легко оптимизировать. В случае, когда последние элементы упорядочены, можно заметить что соответствующие проходы никак не влияют на их порядок. Очевидный способ улучшить данный алгоритм – это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то это означает, что алгоритм может закончить работу. Этот процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и место (индекс) последнего обмена. Ведь ясно, что все пары соседних элементов с индексами, меньшими этого индекса k, уже расположены в нужном порядке. Поэтому следующие проходы можно заканчивать на этом индексе, вместо того чтобы двигаться до установленной заранее нижней границы i. Однако внимательный программист заметит здесь странную асимметрию: один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только на один шаг на каждом проходе. Например, массив {12 18 42 44 55 67 94 06} будет рассортирован при помощи метода пузырька за один проход, а сортировка массива {94 06 12 18 42 44 55 67} потребует семи проходов. Эта неестественная асимметрия подсказывает третье улучшение: менять направление следующих один за другим проходов. Полученный в результате алгоритм называют шейкер-сортировкой. Процедура шейкер-сортировки показана на рисунке Программа 9.

Программа 9. Процедура шейкер-сортировки.

```
void ShakerSort(int a[], int size)
{
      printf("Shaker Sort Mode:\n");
// объявление переменных левой и правой границ для текущего участка
массива
// объявление переменной last для хранения индекса последнего обмена
```

```
        int left = 1, right = size-1, last = right;
        do // повторять до тех пор пока границы не сомкнутся
        {
           for (int j = right; j >= left; j--) // движение справа налево
              if (a[j-1] > a[j]) // проверка условия обмена пары
              {      // обмен элементов в случае неправильного порядка
                 int temp = a[j-1];
                 a[j-1] = a[j];
                 a[j] = temp;
                 last = j; // запоминание индекса последнего обмена
              }
           left = last + 1; //расчет новой левой границы
           for (j = left; j < right+1; j++) // движение слева направо
              if (a[j-1] > a[j]) // проверка условия обмена пары
              {      // обмен элементов в случае неправильного порядка
                 int temp = a[j-1];
                 a[j-1] = a[j];
                 a[j] = temp;
                 last = j; // запоминание индекса последнего обмена
              }
           right = last - 1; //расчет новой правой границы
        } while(left < right); // повторять до тех пор пока границы не сомкнутся
}
```

Число сравнений в алгоритме простого обмена равно:

$$C = \frac{1}{2}(n^2 - n) \qquad (13)$$

Минимальное, среднее и максимальное количества присваиваний равны:

$$M_{\min} = 0, \quad M_{cp} = \frac{1}{4}(n^2 - n), \quad M_{\max} = \frac{1}{2}(n^2 - n) \qquad (14)$$

Анализ улучшенных методов, особенно метода шейкер-сортировки, довольно сложен. Наименьшее число сравнений есть $C_{min} = n - 1$. Среднее число сравнений пропорционально $1/2[n^2 - n(k_2 + \ln n)]$. Но оказывается, все предложенные выше усовершенствования никоим образом не влияют на число обменов, они лишь уменьшают число избыточных повторных проверок. К сожалению, обмен двух элементов – обычно намного более дорогостоящая операция, чем сравнения ключей поэтому эти усовершенствования дают значительно меньший эффект, чем можно было бы ожидать.

Сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором, и действительно, сортировка методом пузырька вряд ли имеет какие-то преимущества, кроме своего легко запоминающегося названия. Алгоритм шейкер-сортировки выгодно использовать в тех случаях, когда известно, что элементы уже почти упорядочены – редкий случай на практике.

СОРТИРОВКА С РАЗДЕЛЕНИЕМ (БЫСТРАЯ СОРТИРОВКА)

Сортировка с разделением – усовершенствованный метод сортировки, основанный на принципе обмена. Учитывая, что сортировка методом пузырька в среднем была наименее эффективной из трех алгоритмов простой сортировки, мы должны требовать значительного улучшения. Однако, неожиданно оказывается, что усовершенствование сортировки, основанной на обмене, дает вообще лучший, из известных до сего времени, метод сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель К. Хоар окрестил его быстрой сортировкой.

Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях. Предположим, что даны n элементов с ключами, расположенными в обратном порядке. Их можно рассортировать, выполнив всего $n/2$ обменов, если сначала поменять местами самый левый и самый правый элементы и так постепенно продвигаться с двух концов к середине. Разумеется, это возможно, только если мы знаем, что элементы расположены строго в обратном порядке. Но все же этот пример наводит на некоторые мысли.

Рассмотрим следующий алгоритм (см. рис.4): выберем случайным образом какой-то элемент (назовем его x), просмотрим массив, двигаясь слева направо, пока не найдем элемент $a_i > x$, а затем просмотрим его справа налево, пока не найдем элемент $a_i < x$. Теперь поменяем местами эти два элемента и продолжим процесс «просмотра с обменом», пока два просмотра не встретятся где-то в середине массива. В результате массив разделится на две части: левую – с ключами меньшими, чем x, и правую – с ключами большими x

Если, например, в качестве x выбрать средний ключ, равный 42, из массива ключей {44 55 12 42 94 06 18 67}, то для того, чтобы разделить массив, потребуются два обмена:

конечные значения индексов $i = 5$ и $j = 3$. Ключи $a_1, \ldots, a_{i-1}$ меньше или равны ключу $x = 42$, ключи $a_{j+1}, \ldots, a_n$ больше или равны x.

$$a[k] \le x, \text{ при } k = 1 \ldots i-1;$$
$$a[k] \ge x, \text{ при } k = j+1 \ldots n;$$
$$a[k] = x, \text{ при } k = j+1, i-1$$

Этот алгоритм очень прост и эффективен. В его правильности можно убедиться на основании того, что два первых утверждения являются вариантами оператора цикла с постусловием. Вначале, при $i = 1$ и $j = n$, они, очевидно, истинны, а при выходе с $i > j$ они предполагают, что получен нужный результат.

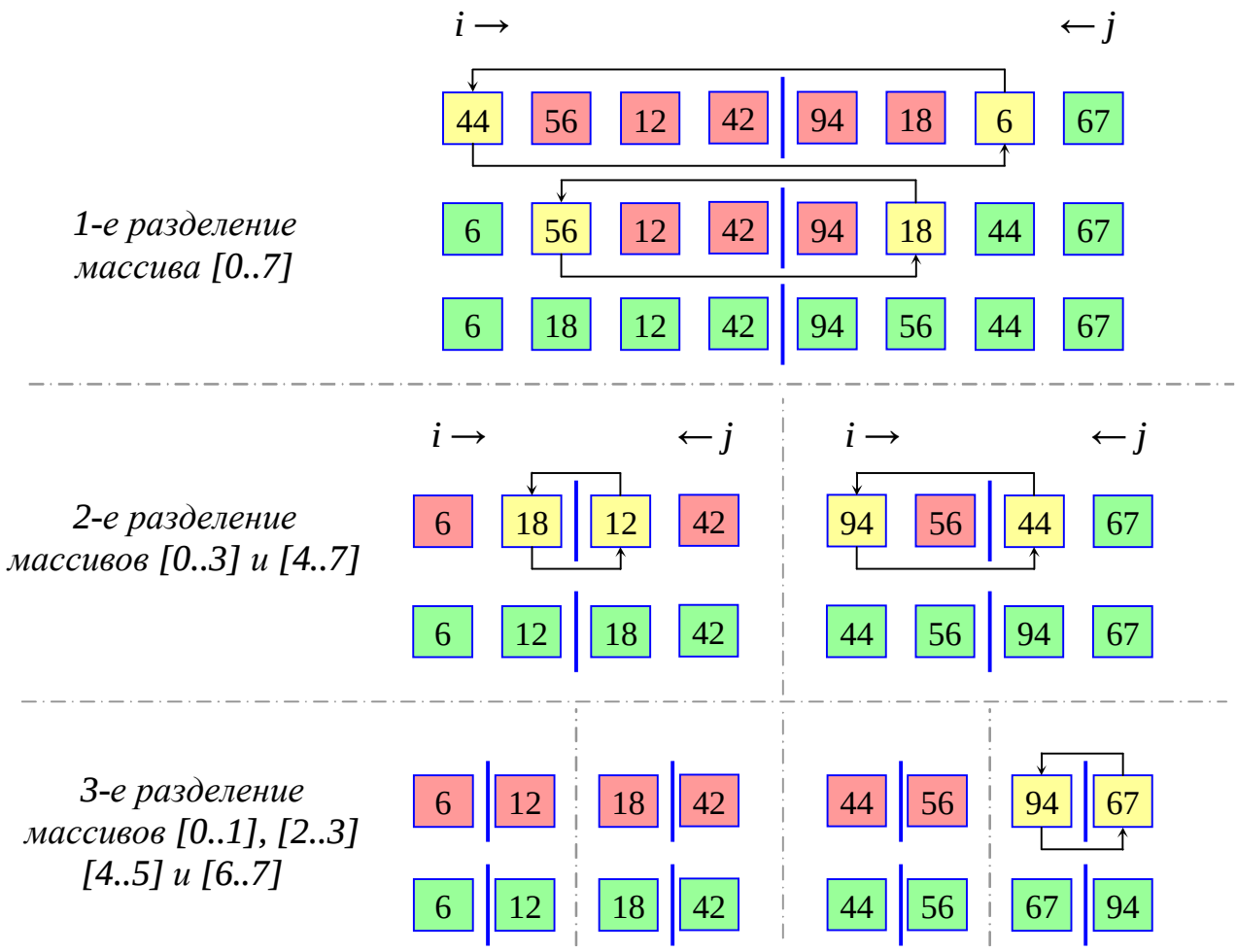

Рис. 4. Пример сортировки разделением: ■ – текущая отсортированная часть, ■ – входная часть, ■ – текущая пара обмениваемых элементов.

Теперь пора вспомнить, что наша цель – не только разделить исходный массив элементов на большие и меньшие, но также рассортировать его. Однако от разделения до сортировки всего лишь один небольшой шаг: разделив массив, нужно сделать то же самое с обеими полученными частями, затем с частями этих частей и т. д. пока каждая часть не будет содержать только один элемент. Этот метод представлен на рисунке Программа 10.

Процедура *QuickSort* рекурсивно вызывает сама себя каждый раз для новой части исходного массива, границы которой определяется парой индексов *left* и *right*. Такое использование рекурсии в алгоритмах – очень мощное средство.

Программа 10. Сортировка разделением (быстрая сортировка).

```
void QuickSort(int a[], int left, int right)
   {
   //присвоение параметрам счетчиков текущих левой и правой границ части
   //массива
      int i = left, j = right;
      // сохранение в mid значения среднего элемента
      int mid = a[(left +right)/2;

      do // повторять пока проходы слева и справа не перехлестнутся
      {
         //проход слева до первого попавшегося элемента больше среднего
         while (a[i] < mid) i++;
         //проход справа до первого попавшегося элемента меньше среднего
         while (a[j] > mid) j--;
         if (i <= j) // обмен местами найденных элементов
         {
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++; j--;// переход к следующим для каждого прохода значениям
         }
      } while(i < j); // повторять пока проходы не перехлестнутся

   // рекурсивное повторение алгоритма для разделенных частей массива
      if ( left < j ) QuickSort(a,left,j);
      if (i < right) QuickSort(a, i,right);
   }
```

Выразить этот же алгоритм можно в виде нерекурсивной процедуры, но это потребует более сложной реализации, т.к. необходимы некоторые дополнительные операции для хранения и обработки информации о границах подмассива.

Основа итеративного решения – ведение списка запросов на разделения, которые еще предстоит выполнить. После каждого шага нужно произвести два очередных разделения, и лишь одно из них можно выполнить непосредственно при следующей итерации, запрос на другое заносится в список. Важно, что запросы из списка выполняются в обратной последовательности. Это предполагает, что первый, занесенный в список запрос, выполняется последним и наоборот; список ведет себя как пульсирующий стек. В нерекурсивной версии быстрой сортировки каждый запрос представлен левым и правым индексами, определяющими границы части, которую впоследствии нужно будет разделить.

Анализ быстрой сортировки. Для того чтобы проанализировать свойства быстрой сортировки, мы должны сначала изучить поведение процесса разбиения.

После выбора границы x процессу разбиения подвергается весь массив. Таким образом, выполняется ровно n сравнений. Число обменов можно оценить при помощи следующего вероятностного рассуждения.

Предположим, что множество данных, которые нужно разделить, состоит из n ключей $1, \ldots, n$, и мы выбрали x в качестве границы. После разделения x будет занимать в массиве позицию x. Число требующихся обменов равно числу элементов в левой части $x-1$, умноженному на вероятность того, что ключ нужно обменять. Ключ обменивается, если он не меньше чем x. Вероятность этого равна $(n - x + 1)/n$. Ожидаемое число обменов вычисляется при помощи суммирования всех возможных вариантов выбора границы и деления этой суммы на n:

$$M = \frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}(n-x+1) = \frac{n}{6} - \frac{1}{6n} \qquad (15)$$

Следовательно, ожидаемое число обменов равно приблизительно $n/6$.

Если предположить, что нам очень везет, и мы всегда выбираем в качестве границы медиану, то каждое разделение разбивает массив на две равные части и число проходов, необходимых для сортировки, равно $\log n$. Тогда общее число сравнений составит $n \log n$, а общее число обменов – $(n/6) \log n$. Разумеется, нельзя ожидать, что мы все время будем попадать на медиану. На самом деле, вероятность этого равна всего лишь $1/n$. Но к удивлению, если граница выбирается случайным образом, эффективность быстрой сортировки в среднем хуже оптимальной лишь в $2 \ln 2$ раз.

Однако быстрая сортировка все же имеет свои «подводные камни». Прежде всего, при небольших значениях n ее эффективность невелика, как и у всех усовершенствованных методов. Ее преимущество по сравнению с другими усовершенствованными методами заключается в том, что для сортировки уже разделенных небольших подмассивов легко можно применить какой-либо простой метод. Это особенно ценно, если говорить о рекурсивной версии программы.

Тем не менее, остается проблема наихудшего случая. Как тогда ведет себя быстрая сортировка? Ответ, к сожалению, разочаровывает. Здесь проявляется слабость быстрой сортировки, которая в таких случаях становится «медленной сортировкой». Рассмотрим, например, неблагоприятный случай, когда каждый раз в качестве x выбирается наибольшее значение в подмассиве. Тогда каждый шаг разбивает сегмент из n элементов на левую часть из $n - 1$ элементов и правую часть, состоящую из одного элемента. В результате вместо $\log n$ необходимо n разбиений, и скорость работы в наихудшем случае оказывается порядка n^2.

Очевидно, что эффективность алгоритма быстрой сортировки определяется выбором элемента x. В нашем примере программа выбирает в качестве x элемент,

расположенный посередине. Заметим, что почти с тем же успехом можно было бы выбрать либо первый, либо последний элемент: a [left] или a [right]. Но при таком выборе наихудший вариант встретится, когда массив уже предварительно рассортирован; быстрая сортировка в этом случае проявляет определенную «неприязнь» к тривиальной работе и предпочитает беспорядочные массивы. При выборе среднего элемента в качестве границы это странное свойство быстрой сортировки менее заметно, так как уже рассортированный массив становится оптимальным случаем! Действительно, средняя скорость работы здесь оказывается несколько выше, если выбирается средний элемент. Хоар считает, что выбор *x* должен быть «случайным», или в качестве *x* нужно выбирать медиану из небольшого числа ключей (скажем, из 3-ех). Такой осмотрительный выбор почти не влияет на среднюю скорость быстрой сортировки, но значительно улучшает скорость в худшем случае. Как мы видим, быстрая сортировка напоминает азартную игру, где следует заранее рассчитать, сколько можно позволить себе проиграть в случае невезения.

Тем не менее, алгоритм быстрой сортировки завоевал высокую популярность у программистов чем говорит его реализация в виде функции стандартной библиотеки языка C++ *stdlib*. Рассмотрим теперь ее применение. Прототип функции выглядит следующим образом:

```
void qsort( void *base, size_t num, size_t width, int (__cdecl *compare )(const
void *elem1, const void *elem2));
```

где:

**base* – бестиповый указатель на начало сортируемого массива;

num – количество сортируемых элементов массива, переменная типа *size_t*;

width – размер в байтах одного элемента массива, переменная типа *size_t*;

*compare – указатель на функцию пользователя реализующую сравнение элементов массива;

Тип *size_t* определен в библиотеке *stdlib* как

```
typedef unsigned int size_t;
```

т.е. параметры *num* и *width* беззнаковые целые числа

Прототип функции сравнения представлен ниже:

```
int compare (const void *elem1, const void *elem2);
```

где:

**elem1, *elem2* – бестиповые указатели константы на сравниваемые элементы.

Процедура *qsort* использует функцию *compare* каждый раз, когда сравнивает элементы, при этом при вызове этой функции в нее передаются указатели непосредственно на сравниваемые элементы. Функция реализуется программистом произвольно в зависимости от поставленной задачи, но она обязательно должна вернуть целочисленное значение по следующему правилу:

меньше нуля (<0) – если *elem1* меньше чем *elem2* (*elem1 < elem2*);

равно нулю (=0) – если *elem1* равен *elem2* (*elem1 = elem2*);

больше нуля (>0) – если *elem1* больше чем *elem2* (*elem1 > elem2*);

Процедура *qsort* сортирует исходные данные всегда в возрастающем порядке или по возрастанию. Для того чтобы изменить порядок сортировки нужно изменить реализацию функции *compare* на логически противоположную, что в большинстве случаев делается заменой операций отношения на противоположные (больше на меньше и наоборот) или обменом возвращаемых значений. Ниже на рисунке Программа 11 представлены вызов функции *qsort и* реализация функции *compare* для нашего рассматриваемого примера сортировки простого массива из целых чисел.

Программа 11. Вызов функции *qsort и* реализация функции *compare*

```
#include <stdlib.h>
< ... > // см. Стр. 5
case '7':
      {
         printf("C++ Quick Sort Mode :\n");
         qsort(array,n,sizeof(int),compare);} break;
      }
< ... >
// функция сравнения элементов массива для процедуры qsort

int compare( const void *arg1, const void *arg2 )
   {
      // arg1 и arg2 - безтиповые указатели константы
      // на сравниваемые элементы
      // *(int*) arg1 - приведение безтипового указателя arg1 к типу
      // указатель на int и его разыменование (взятие значения)

      if ( *(int*) arg1 > *(int*) arg2) return 1;
      //если [1] > [2] вернуть значение >0
      else
         if ( *(int*) arg1 < *(int*) arg2) return -1; //если [1] > [2] вернуть <0
         else return 0; // если [1] > [2] вернуть значение =0
   }
```

Более оптимальный вариант функции *compare* для нашего случая :

```
int compare( const void *arg1, const void *arg2 )
   {
      return *(int*) arg1 - *(int*) arg2;
      // если [1]>[2] то разность >0, иначе равна или меньше нуля
   }
```

ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы + структуры данных = программы. Москва, Мир, 1985, 406 с.

2. Дейтел Х.М., Дейтел П.Дж. Как программировать на C++. М., Бином-пресс, 2007, 800 с.