1 Поиск в неупорядоченных массивах

Самым простым вариантом поиска можно считать поиск элемента в одномерном неупорядоченном массиве. Сформулируем задачу следующим образом: дан одномерный неупорядоченный массив, состоящий из целых чисел, и необходимо проверить, содержится ли данное число в этом массиве.

Пусть массив называется a и состоит из n элементов, а искомое число равно k. Тогда код, осуществляющий поиск, можно записать так:

```
int j = -1;
for (i=0; i < n; i++)
  if (a[i] ==  k) j = i;
```

В случае если число k ни разу не встречалось в массиве, j будет равно -1. Приведенная выше функция будет искать последнее вхождение числа k в массиве a. Если нам необходимо искать первое вхождение, то вместо присваивания $j = i$ следует добавить оператор `break;` (в этом случае искомый индекс будет храниться в переменной i).

И в том и в другом случае алгоритм будет иметь сложность $O(N)$.

На этом примере можно рассмотреть «барьерный» метод, который может быть полезен в очень многих задачах. Для использования барьерного метода наш массив должен иметь один дополнительный элемент (т.е. его длина должна быть не меньше, чем $n+1$ элемент). Отметим, что таким способом можно искать только первое вхождение элемента:

```
a[n+1] = k;
for (i=0; a[i] != k; i++);
```

Если элемент k встречается в массиве, то его индекс будет находиться в переменной i, если же такой элемент в массиве не встречается, то i будет равно $n + 1$.

Рассмотрим отдельно задачу поиска минимума и максимума в массиве. Так же как и при поиске вхождения элемента, будем искать не само значения минимума или максимума, а индекс минимального (максимального) элемента. Это избавит нас от многих проблем и позволит совершать меньшее количество ошибок при программировании. Поиск минимального элемента в массиве a будет выглядеть следующим образом:

```
imin = 0;
For (i, n)
  if (a[i] < a[imin]) imin = i;
```

Индекс минимального элемента будет храниться в переменной $imin$, а сам минимум равен $a[imin]$. Минимум и максимум следует обязательно искать по индексу, а не по значению. Например, если мы будем пытаться хранить непосредственно значение минимума или максимума, то можем легко ошибиться с начальной инициализацией. Например, для массива вещественных чисел определить значения, которыми изначально следует инициализировать минимум и максимум.

Теперь рассмотрим задачу поиска минимума и максимума одновременно. Можно реализовать такой поиск аналогично:

```
imin = 0;
imax = 0;
for (i=1; i<n; i++)
{
  if (a[i] < a[imin]) imin = i;
  if (a[i] > a[imax]) imax = i;
}
```

Такая реализация требует $2 \times N - 2$ сравнения. Но эту задачу можно решить и за меньшее количество сравнений. Разобьем все элементы на пары, и будем искать в каждой паре минимум и максимум ($N/2$ сравнений), затем минимум будем искать только среди минимальных элементов пар, а максимум — среди максимальных. Общее количество сравнений будет около $3 \times N/2$ (проблема возникает, когда количество элементов нечетное — один из элементов остается без пары). Точно это можно записать как $\lceil 3 \times N/2 \rceil - 2$, где $\lceil\ \rceil$ — округление до большего целого.

Рассмотрим еще один способ поиска максимума. После разбиения элементов на пары будем продолжать этот процесс, аналогично турниру «на вылет». Т.е. заново разобьем максимальные элементы из пар на пары и снова найдем максимум и т.д. Для поиска максимального элемента будет по-прежнему требовать $N - 1$ операция сравнения, но сам максимальный элемент будет участвовать только в $\log N$ сравнениях. И одно из этих сравнений обязательно будет со вторым по величине элементом. Таким образом, для поиска второго по величине элемента будет требоваться $\lceil \log N \rceil - 1$ сравнение (при условии, что все сравнения для максимального элемента проведены).

Обычно такие методы используются в особых случаях, когда это непосредственн требуется в решении задачи. Для общего случая подходят более простые методы, где количество сравнений не играет такой важной роли.

Однако и этот метод может быть полезен при поиске «порядковых статистик» массива. k-ой порядковой статистикой массива называется k-

ый по счету элемент этого массива (т.е. если массив отсортировать по неубыванию, то k-ая порядковая статистика — это элемент, стоящий на k-ой позиции).

2 Поиск порядковых статистик

Как известно, существуют методы сортировки массива за $O(N \log N)$. Для поиска k-ой порядковой статистики можно отсортировать массив и вывести k-ый элемент. Но в этом случае мы совершаем множество лишних действий, ведь с помощью сортировки мы найдем все порядковые статистики, а не только k-ую.

Приведенный ниже алгоритм работает за $O(N)$. Существует алгоритм поиска i-ой порядковой статистики за $O(N)$ в худшем случае, но он тяжел в реализации и имеет большую константу.

Схема алгоритма имеет следующий вид. Пусть k — номер искомой порядковой статистики, l (это маленькая латинская L) и r — текущие левая и правая границы области массива a, в которой мы ищем k-ую статистику. Если `l == r`, то область поиска ограничена одним элементом, т.е. k-ая порядковая статистика равна `a[r]`.

На каждом шаге будем выбирать число `s = (l + r) / 2`. Вообще говоря, мы можем выбирать произвольное число из интервала $[l, r]$, но генерация случайного числа занимает достаточно много времени, поэтому лучше использовать какое-либо фиксированное число. Расположим элементы массива интервала $[l, r]$ так, чтобы сначала шли все элементы, меньшие `a[s]`, а затем все остальные. Первый элемент второй группы обозначим за j. Тогда если $k \leq j$, то будем продолжать поиск с неизменным значением l и $r = j$ (в левой части массива), иначе будем осуществлять поиск при $l = j + 1$ и неизмененным r.

Сначала запишем функцию, осуществляющую такой поиск, а затем приведем пример и необходимые пояснения:

```
int search(int *a, int k, int l, int r)
{
  int s, m, i=l, j=r, tmp;
  if (l == r) return a[r];
  s = (l + r) / 2;
  m = a[s];
  while (i < j) {
    while (a[i] < m) i++;
    while (a[j] > m) j--;
```

```
    if (i < j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++; j--;
    }
  }
  if (k <= j) return search(a, k, l, j);
  else return search(a, k, j+1, r);
}
```

Вызывать функцию следует так: `search(a, k, 0, n-1)`, где a — массив, k — номер статистики, которую надо получить, а n — количество элементов в массиве.

Перестановку элементов мы осуществляем следующим образом: находим первый «неправильный» элемент слева, затем первый неправильный элемент «справа», а в случае, если указатели не сошлись, осуществляем обмен этих ячеек массива.

Приведем пример для массива $\{2, 7, 8, 6, 0, 4, 1, 9, 3, 5\}$ и $k = 9$:

1. $\{2, 7, 8, 6, 0, 4, 1, 9, 3, 5\}, l = 0, r = 9, m = 0$
2. $\{0, 7, 8, 6, 2, 4, 1, 9, 3, 5\}, l = 1, r = 9, m = 4$
3. $\{0, 3, 1, 4, 2, 6, 8, 9, 7, 5\}, l = 5, r = 9, m = 9$
4. $\{0, 3, 1, 4, 2, 6, 8, 5, 7, 9\}, l = 9, r = 9, m = 9$

На поиск максимального элемента нам потребовалось 4 вызова функции `search`.

Доказательство сложности $O(N)$ опирается на суммирование ряда, в котором i-ый элемент равен $N/2^i$. Методы доказательства сходимости рядов изучаются в школьном или университетском курсе математического анализа.

3 Бинарный поиск в упорядоченных массивах

Под упорядоченным массивом будем понимать массив, упорядоченный по неубыванию, т.е. $a[1] \leq a[2] \leq \ldots \leq a[N]$.

У нас имеется заданная своими границами область поиска. Мы выбираем ее середину, и, если искомый элемент меньше, чем средний, то поиск осуществляется в левой части, иначе — в правой. Действительно, если искомый элемент меньше среднего, то и меньше всех элементов, которые находятся правее

среднего, а значит, их сразу можно исключить из рассмотрения. Аналогично для случая, когда искомый элемент больше среднего.

Код, осуществляющий бинарный поиск в упорядоченном массиве выглядит так:

```
while (l<r) {
  m=(l+r)/2;
  if (a[m]<k) l=m+1;
  else r=m;
}
if (a[r]==k) printf("%d", r);
else printf("-1");
```

Перед выполнением этого кода следует присвоить переменным l и r значения 0 и $n-1$ соответственно. В случае если элемент не найдем, эта программа выводит -1.

Сложность алгоритма бинарного поиска составляет $O(\log N)$, где N — количество элементов в массиве.

4 Бинарный поиск для монотонных функций

Бинарный поиск может использоваться не только для поиска элементов в массиве, но и для поиска корней уравнений и значений монотонных (возрастающих или убывающих) функций. Напомним, что функция называется возрастающей, если $\forall x_1, x_2 : x_1 > x_2 \Rightarrow f(x_1) > f(x_2)$ (для любых x_1 и x_2, если $x_1 > x_2$, то $f(x_1)$ также больше $f(x_2)$).

Действительно, так же как и в массиве, мы можем исключить из рассмотрения половину текущей области, если нам заведомо известно, что там не существует решения. В случае же, если функция не монотонна, то воспользоваться бинарным поиском нельзя, т.к. он может выдавать неправильный ответ, либо находить не все ответы.

Для примера рассмотрим задачу поиска кубического корня. Кубическим корнем из числа x (обозначается $\sqrt[3]{x}$) называется такое число y, что $y^3 = x$.

Сформулируем задачу так: для данного вещественного числа x ($x \geq 1$) найти кубический корень с точностью не менее 5 знаков после точки.

Функция при $x \geq 1$, ограничена сверху числом x, а снизу — единицей. Таким образом, за нижнюю границу мы выбираем 1, за верхнюю — само число x. После этого делим текущий отрезок пополам, возводим середину в куб и если куб больше x, то заменяем верхнюю грань, иначе — нижнюю.

Код будет выглядеть следующим образом:

```
r = x;
l = 1;
while (fabs(l-r)>eps) {
  m=(l+r)/2;
  if (m*m*m<x) l=m;
  else r=m;
}
```

Для того чтобы пользоваться функцией `fabs`, необходимо подключить библиотеку `math.h`.

5 Бинарный поиск по ответу

Во многих задачах в качестве ответа необходимо вывести какое-либо число. При этом достаточно легко сказать, больше ли это число, чем нужно, или меньше, несмотря на то, что вычисление самого ответа может быть довольно трудоемкой операцией. В таком случае мы можем выбрать число заведомо меньшее ответа и число заведомо большее ответа, а правильное решение искать бинарным поиском.

Для примера рассмотрим решение задач нескольких прошедших олимпиад.

Очень легкая задача

Московская олимпиада по информатике 2006-2007

Сегодня утром жюри решило добавить в вариант олимпиады еще одну, Очень Легкую Задачу. Ответственный секретарь Оргкомитета напечатал ее условие в одном экземпляре, и теперь ему нужно до начала олимпиады успеть сделать еще N копий. В его распоряжении имеются два ксерокса, один из которых копирует лист за секунд, а другой — за y. (Разрешается использовать как один ксерокс, так и оба одновременно. Можно копировать не только с оригинала, но и с копии.) Помогите ему выяснить, какое минимальное время для этого потребуется.

Формат входных данных

Во входном файле записаны три натуральных числа N, x и y, разделенные пробелом ($1 \le N \le 2 \times 10^8$, $1 \le x, y \le 10$).

Формат выходных данных

Выведите одно число — минимальное время в секундах, необходимое для получения N копий.

Примеры

Входные данные	Выходные данные
4 1 1	3
5 1 2	4

Существует конструктивное решение этой задачи (формула), которую можно вывести и, при желании, доказать. Однако очень легко реализовать решение этой задачи с помощью бинарного поиска.

Первую страницу мы копируем за $min(x, y)$ секунд и, затем, рассматриваем решение уже для $N - 1$ страницы.

Пусть l - минимальное время, r - максимальное. Минимум нам необходимо потратить 0 секунд, максимум, например $(N - 1) \times x$ секунд (страницы делаются полностью на одном ксероксе). Считаем среднее значение и смотрим, сколько полных страниц можно напечатать за это время, используя оба ксерокса. Если количество страниц меньше $N - 1$, то мы меняем нижнюю границу, иначе — верхнюю.

```
int main(void)
{
  int n, x, y, i, j, l, r, now;
  double speed;
  scanf("%d%d%d", &n, &i, &j);
  x=i<j?i:j;
  y=i>j?i:j;
  l=0;
  r = (n-1)*y;
  while (l != r) {
    now = (l+r)/2;
    j = now / x + now / y;
    if (j < n-1) l = now+1;
    else r = now;
  }
  printf("%d", r+x);
  return 0;
}
```

Автобус

13 Украинская олимпиада

Служебный автобус совершает один рейс по установленному маршруту и в случае наличия свободных мест подбирает рабочих, которые ожидают на остановках, и отвозит их на завод. Автобус также может ждать на остановке рабочих, которые еще не пришли. Известно время прихода каждого рабочего на свою остановку и время проезда автобуса от каждой остановки до следующей. Автобус приходит на первую остановку в нулевой момент времени. Продолжительность посадки рабочих в автобус считается нулевой.

Задание: Написать программу, которая определит минимальное время, за которое автобус привезет максимально возможное количество рабочих.

Формат входных данных

Входной текстовый файл в первой строке содержит количество остановок N и количество мест в автобусе M. Каждая i-я строчка из последующих N строчек содержит целое число — время движения от остановки i к остановке $i+1$ ($N+1$-я остановка — завод), количество рабочих K, которые придут на i-ю остановку, и время прихода каждого рабочего на эту остановку в порядке прихода ($1 \leq M \leq 2000, 1 \leq N, K \leq 200000$).

Формат выходных данных

Единственная строка выходного текстового файла должен содержать минимальное время, необходимое для перевозки максимального количества рабочих.

Примеры

Входные данные	Выходные данные
3 5 1 2 0 1 1 1 2 1 4 0 2 3 4	4

Сначала определим, что такое максимально возможное количество рабочих. Если общее количество рабочих больше вместимости автобуса, то это — объем автобуса, если же рабочих меньше чем вместимость автобуса — то это количество всех рабочих (в этом случае вместимости автобуса уместно присвоить значение, равное количеству людей).

Когда мы считываем данные, следует определить время прихода последнего человека (т.е. то время, когда уже все люди будут на остановках) — это будет максимум в бинарном поиске. Минимум будет равен нулю. Если автобус должен задержаться перед остановкой, то он должен сделать это перед первой остановкой (действительно, если он подъедет к первой остановке, заберет людей, а потом будет ждать у второй остановки, то в это время на первую могут придти еще люди, а если ждать перед первой, то люди со второй никуда не денутся). Минимум и максимум у нас есть. Теперь берем задержку, равную $x = (min + max)/2$. С помощью процедуры, которая будет описана ниже, вычисляем, сколько людей успеет придти до момента x на первую остановку, для второй остановки будет задержка, равная $x + a[1]$, где $a[1]$ — время следования от первой остановки до второй, для третьей задержка — $x + a[1] + a[2]$ и т.д. Если количество севших в автобус на всех остановках больше либо равно вместимости автобуса, то надо заменить x на $(min + x)/2$, если остались места в автобусе то $x = (x+max)/2$. Условие выхода будет такое: если при некоторой задержке

x автобус заполнен, а при задержке $(x-1)$ автобус не полон, то ответ x.

Теперь второй бинарный поиск. Тот самый, который определяет, сколько людей успеет придти на определенную остановку до определенного момента. Здесь максимумом дихотомии будет количество людей на остановке, а минимумом — ноль. Выбираем среднего человека — если его время прихода меньше, чем задержка, то $x = (x + max)/2$, если он не успеет придти, то $x = (min+x)/2$. Здесь условие выхода такое: если человек успевает придти на остановку, а следующий за ним нет — то ответом будет номер человека. Отдельно нужно обрабатывать случай, если на автобус сядут все люди с остановки.

6 Поиск по групповому признаку

В прошлой части лекции мы рассматривали задачи, в которых для данного можно получить ответ «больше» или «меньше». Теперь рассмотрим задачи, в которых для некоторого подмножества всех элементов можно получить ответ, например, на вопрос «содержится ли искомый элемент в данном подмножестве?».

Для примера рассмотрим задачу поиска одного радиоактивного шарика среди 8 шариков. При этом мы можем измерить радиоактивность некоторой группы шариков и определить, содержится ли радиоактивный шарик в этой группе. Необходимо минимизировать количество измерений для худшего случая.

Представим номера шариков в виде двоичных чисел:

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Мы можем найти радиоактивный шарик за три измерения. При этом в первом измерении будут участвовать шарики, содержащие 1 в первом разряде, во втором — шарики с 1 во втором разряде и т.д. Результаты измерений будем записывать так: если в группе содержится радиоактивный шарик, то запишем 1, в соответствующий номеру измерения разряд, в противном случае запишем 0.

Полученное двоичное число будет однозначно определять номер радиоактивного шарика.

В общем случае для поиска 1 шарика среди N шариков необходимо $\lceil \log_2(N) \rceil$ измерений.

Похожее решение имеет следующая задача: среди 9 монет необходимо найти одну фальшивую, пользуясь чашечными весами, если известно, что

фальшивая монета весит больше настоящей. Здесь для каждого взвешивания возможно три результата: перевесила левая чашка, перевесила правая и весы уравновешены.

Закодируем номера монет в троичной системе счисления:

0	1	2	3	4	5	6	7	8
00	01	02	10	11	12	20	21	22

Задачу можно решить за два взвешивания, при этом 1 будет означать, что монету нужно положить на левую чашу весов, 2 — на правую чашу, а 0 — что монета не участвует во взвешивании.

Запишем результаты каждого взвешивания в соответствующий разряд (1 — перевесила левая чаша, 2 — правая, 0 — весы уравновешены). Полученный результат однозначно определяет номер фальшивой монеты. В общем случае для поиска ответа необходимо $\lceil \log_3(N) \rceil$ взвешиваний.

Теперь рассмотрим задачу поиска фальшивой монеты среди 12 с помощью чашечных весов, при условии, что неизвестно, тяжелее ли фальшивая монета или легче.

Первое логичное условие, которое мы опускали в предыдущих задачах, состоит в том, что количество монет на различных чашах весов должно быть одинаковым.

Интуитивное решение состоит в том, чтобы разделить монеты на 4 части — 2 части отложить и положить по одной части на каждую из чаш весов, если весы уравновешены, то фальшивая монета находится в отложенной части, иначе — среди монет, положенных на чаши. Будем продолжать данный процесс для части, содержащей фальшивую монету, и получим оценку $\lceil \log_2(N) \rceil$ (фактически, задача свелась к задаче о радиоактивном шарике). При этом мы никак не учитываем результаты предыдущих взвешиваний, которые также могут нести полезную информацию.

Введем следующую систему кодирования: 0 означает, что монета не участвует во взвешивании, 1 — что участвует (при этом, если она уже участвовала во взвешиваниях, то монета остается на той же чаше), 2 — что монета участвует во взвешивании (при этом она обязательно участвовала в одном из предыдущих взвешиваний и в текущем взвешивании лежит на противоположной чаше весов). Из этого условия следует, что ни в каком коде 2 не может предшествовать 1. Запишем все варианты, выписывая, для удобства, коды в столбик:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	1	0	1	1	1	0	1	1	1	1	1
0	0	1	0	1	0	1	1	1	0	2	1	2	2
0	1	0	0	1	1	0	1	2	2	0	2	1	2

Чтобы было возможно провести измерения, необходимо, чтобы в каждой строке было четное число элементов (иначе количество монет на чашах будет различным). Для этого вычеркнем из таблицы, например, 0 и 7 столбцы (как видно, 0 столбец не меняет четность, и задача может быть решена и для 13 монет).

В первом взвешивании возьмем 8 монет с 1 в первом разряде и положим их на чаши весов по 4. На 2 взвешивании уберем 3 монеты (у которых 0 во втором разряде), переложим 3 монеты на противоположную чашу весов и заполним оставшиеся места теми монетами, у которых 1 во втором разряде возникает впервые. Пользуясь теми же соображениями, проведем второе взвешивание.

Результаты взвешиваний будем записывать следующим образом: если чаши уравновешены, то записываем в разряд, соответствующий измерению, 0. Если одна чаша весов перевесила впервые или в ту же сторону, как и в предыдущем взвешивании (когда чаши не были уравновешены), то записываем 1. Если же весы перевесили в противоположную предыдущему неуравновешенному взвешиванию сторону, то запишем 2. В результате получим код фальшивой монеты (действительно, если фальшивая монета не участвовала во взвешивании, то весы уравновешены, если была на одной и той же чаше — получим единицы, а если на разных — двойки). В общем случае количество взвешиваний будет равно $\lceil log_3(2 \times N + 1) \rceil$, т.е. для $N \geq 9$ это решение более эффективно, чем интуитивное.

В общем случае, в решении задач, где можно определить некий признак для группы элементов или для нескольких групп, мы должны стремиться разбить элементы на как можно более похожие по количеству группы (это необходимо для минимизации количества измерений в худшем случае). Каждому элементу следует ставить в соответствие код фиксированной длины и находить метод отображения результатов измерений в соответствующий код.

7 Сортировка пузырьком

Условимся считать массив отсортированным, если элементы расположены в порядке неубывания (т.е. каждый элемент не меньше предыдущего). Все примеры будем рассматривать на типе `int`, однако он может быть заменен любым другим сравнимым типом данных.

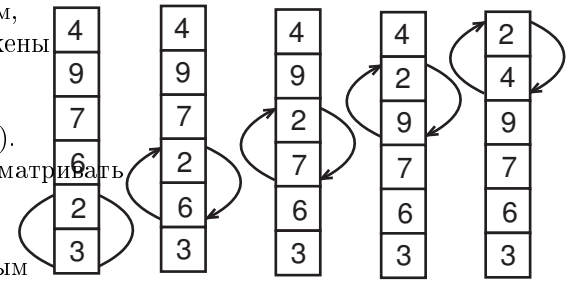

Рис. 1: Нулевой проход. Сравниваемые пары и обмены выделены

Рассмотрение методов сортировки начнем с сортировки пузырьком (BubbleSort).

Это один из простейших методов сортировки, который обычно входит в школьный курс программирования. Название метода отражает его сущность: на каждом шаге самый «легкий» элемент поднимается до своего места («всплывает»). Для этого мы просматриваем все элементы снизу вверх, берем пару соседних элементов и, в случае, если они стоят неправильно, меняем их местами.

Вместо поднятия самого «легкого» элемента можно «топить» самый «тяжелый».

Т.к. за каждый шаг на свое место встает ровно 1 элемент (самый «легкий» из оставшихся), то нам потребуется выполнить N шагов.

Текст функции сортировки можно записать так:

```
void bubble_sort(int a[], int n)
{
  int i, j, k;
  For(i, n)
  for (j=n-1; j>i; j--)
    if (a[j-1] > a[j]) {
      k = a[j-1];
      a[j-1] = a[j];
      a[j] = k;
    }
}
```

Напомним, что здесь мы использовали макроподстановку

```
#define For(a,b) for(a=0; a<b; a++)
```

которая избавляет нас от необходимости писать длинные команды и уменьшает количество возможных опечаток и ошибок.

Алгоритм не использует дополнительной памяти, т.е. все действия осуществляются на одном и том же массиве.

Сложность алгоритма сортировки пузырьком составляет $O(N2)$, количество операций сравнения: $N \times (N-1)/2$. Это очень плохая сложность, но алгоритм имеет два плюса.

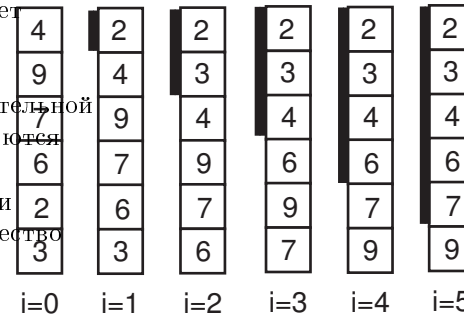

Рис. 2: Номер прохода. Отсортированная часть помечена полосой

Во-первых, он легко реализуется, а значит, может и должен применяться в тех случаях, когда требуется однократная сортировка массива. При этом размер массива не должен быть больше 10000, т.к. иначе алгоритм сортировки пузырьком не будет укладываться в отведенное время.

Во-вторых, сортировка пузырьком использует только сравнения и перестановки соседних элементов, а значит, может использоваться в тех задачах, где явно разрешен только такой обмен и для сортировки, например, списков.

Существуют разнообразные «оптимизации» сортировки пузырьком, которые усложняют (а нередко и увеличивают время работы алгоритма), но не приносят выгоды ни в плане сложности, ни в плане быстродействия.

На этом плюсы сортировки пузырьком заканчиваются. В дальнейшем мы еще более сузим область применения сортировки пузырьком.

8 Сортировка прямым выбором

Рассмотрим еще один квадратичный алгоритм, который, однако, является оптимальным по количеству присваиваний и может быть использован, когда по условию задачи необходимо явно минимизировать количество присваиваний.

Суть метода заключается в следующем: мы будем выбирать минимальный элемент в оставшейся части массива и приписывать его к уже отсортированной части. Повторив эти действия N раз, мы получим отсортированный массив.

```
void select_sort(int a[], int n)
{
  int i, j, k;
  For (i,n) {
    k=i;
    for(j=i+1; j<n; j++)
      if (a[j]<a[k]) k=j;
    j=a[k]; a[k]=a[i]; a[i]=j;
  }
}
```

Количество сравнений составляет $O(N^2)$, а количество присваиваний всего $O(N)$. В целом это плохой метод, и он должен быть использован только в случаях, когда явно необходимо минимизировать количество присваиваний

9 Пирамидальная сортировка

Начнем рассмотрение эффективных алгоритмов сортировки (работающих за $O(N \log N)$) с пирамидальной сортировки, в которой используются знакомые нам идеи кучи.

Мы будем выбирать из кучи самый большой элемент, и записывать его в начало уже отсортированной части массива (сортировка выбором в обратном порядке). Т.е. отсортированный массив будет строиться от конца к началу. Такие ухищрения необходимы, чтобы не было необходимости в дополнительной памяти и для ускорения работы алгоритма — куча будет располагаться в начале массива, а отсортированная часть будет находиться после кучи.

Напомним свойство кучи максимумов: элементы с индексами $i+1$ и $i+2$ не больше, чем элемент с индексом i (естественно, если $i+1$ и $i+2$ лежат в пределах кучи). Пусть n — размер кучи, тогда вторая половина массива (элементы от $n/2+1$ до n) удовлетворяют свойству кучи. Для остальных элементов вызовем функцию «проталкивания» по куче, начиная с $n/2$ до 0.

```
void down_heap(int a[], int k, int n)
{
  int minp, pos=0, y, temp=a[k];
  while (k*2+1 < n) {
    y=k*2+1;
    if (y < n-1 && a[y] < a[y+1]) y++;
```

```
    if (temp >= a[y]) break;
    a[k]=a[y];
    k=y;
  }
  a[k]=temp;
}
```

Эта функция получает указатель на массив, номер элемента, который необходимо протолкнуть и размер кучи. У нее есть небольшие отличия от обычных функций работы с кучей. Номер минимального предка хранится в переменной y, если необходимость в обменах закончена, то мы выходим из цикла и записываем просеянную переменную на предназначенное ей место.

Сама сортировка будет состоять из создания кучи из массива и N переносов элементов с вершины кучи с последующим восстановлением свойства кучи:

```
void heap_sort(int a[], int n) {
  int i, temp;
  for(i=n/2-1; i >= 0; i--) down_heap(a, i, n);
  for(i=n-1; i > 0; i--) {
    temp=a[i]; a[i]=a[0]; a[0]=temp;
    down_heap(a, 0, i);
  }
}
```

Всего в процессе работы алгоритма будет выполнено $3 \times N/2 - 2$ вызова функции `down_heap`, каждый из которых занимает $O(\log N)$. Таким образом, мы и получаем искомую сложность в $O(N \log N)$, не используя при этом дополнительной памяти. Количество присваиваний также составляет $O(N \log N)$.

Пирамидальную сортировку следует осуществлять, если из условия задачи понятно, что единственной разрешенной операцией является «проталкивание» элемента по куче, либо в случае отсутствия дополнительной памяти.

10 Быстрая сортировка

Мы уже рассматривали идеи, которые используются в быстрой сортировке, при поиске порядковых статистик. Точно так же, как и в том алгоритме, мы выбираем некий опорный элемент и все числа, меньшие его перемещаем в левую часть массива, а все числа большие его — в правую часть. Затем вызываем функцию сортировки для каждой из этих частей.

Таким образом, наша функция сортировки должна принимать указатель на массив и две переменные, обозначающие левую и правую границу сортируемой области.

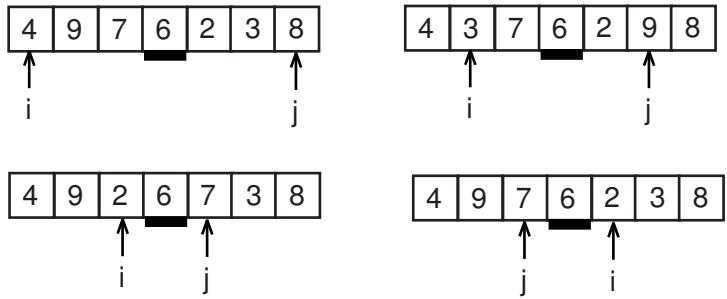

Рис. 3: Первый проход быстрой сортировки

Остановимся более подробно на выборе опорного элемента. В некоторых книгах рекомендуется выбирать случайный элемент между левой и правой границей. Хотя теоретически это красиво и правильно, но на практике следует учитывать, что функция генерации случайного числа достаточно медленная и такой метод заметно ухудшает производительность алгоритма в среднем.

Наиболее часто используется середина области, т.е. элемент с индексом $(l + r)/2$. При таком подходе используются быстрые операции сложения и деления на два, и в целом он работает достаточно неплохо. Однако в некоторых задачах, где сутью является исключительно сортировка, хитрое жюри специально подбирает тесты так, чтобы «завалить» стандартную быструю сортировку с выбором опорного элемента из середины. Стоит заметить, что это очень редкая ситуация, но все же стоит знать, что можно выбирать произвольный элемент с индексом m так, чтобы выполнялось неравенство $l \leq m \leq r$. Чтобы это условие выполнялось, достаточно выбрать произвольные два числа x и y и выбирать m исходя из следующего соотношения: $m = (x \times l + y \times r)/(x + y)$. В целом такой метод будет незначительно проигрывать выбору среднего элемента, т.к. требует двух дополнительных умножений.

Приведем текст функции быстрой сортировки с выбором среднего элемента в качестве опорного:

```
void quick_sort(int a[], int n)
{
```

```
  int i = 0, j = n-1, temp, p;
  p = a[n/2];
  do {
    while (a[i] < p) i++;
    while (a[j] > p) j--;
    if (i <= j) {
      temp = a[i]; a[i] = a[j]; a[j] = temp;
      i++; j--;
    }
  } while ( i<=j );
  if (j > 0) quick_sort(a, j);
  if (n > i) quick_sort(a+i, n-i);
}
```

Здесь мы не передаем левую границу в явном виде, а пользуемся указательной арифметикой (к указателю на начало сортируемой части массива при вызове функции для правой части просто прибавляем число — количество элементов, которое следует «пропустить»). Это делает использование функции быстрой сортировки более удобным, т.к. в нее надо просто передать массив и количество элементов в нем (как и в других видах сортировки).

Алгоритм быстрой сортировки в среднем использует $O(N \log N)$ сравнений и $O(N \log N)$ присваиваний (на практике даже меньше) и использует $O(\log N)$ дополнительной памяти (стек для вызова рекурсивных функций). В худшем случае алгоритм имеет сложность $O(N^2)$ и использует $O(N)$ дополнительной памяти, однако вероятность возникновения худшего случая крайне мала: на каждом шаге вероятность худшего случая равна $2/N$, где N — текущее количество элементов.

Рассмотрим возможные оптимизации метода быстрой сортировки.

Во-первых, при вызове рекурсивной функции возникают накладные расходы на хранение локальных переменных (которые нам не особо нужны при рекурсивных вызовах) и другой служебной информацией. Таким образом, при замене рекурсии стеком мы получим небольшой прирост производительности и небольшое снижение требуемого объема дополнительной памяти.

Во-вторых, как мы знаем, вызов функции — достаточно накладная операция, а для небольших массивов быстрая сортировка работает не очень хорошо. Поэтому, если при вызове функции сортировки в массиве находится меньше, чем K элементов, разумно использовать какой-либо нерекурсивный метод, например, сортировку вставками или выбором. Число K при этом выбирается в районе 20, конкретные значения подбираются опытным путем. Такая модификация может дать до 15% прироста производительности.

Быструю сортировку можно использовать и для двусвязных списков (т.к. в ней осуществляется только последовательный доступ с начала и с конца), но в этом случае возникают проблемы с выбором опорного элемента — его приходится брать первым или последним в сортируемой области. Эту проблема можно решить неким набором псевдослучайных перестановок элементов списка, тогда даже если данные были подобраны специально, эффект нейтрализуется.

11 Сортировка слияниями

Сортировка слияниями также основывается на идее, которая уже была нами затронута при рассмотрении алгоритма поиска двух максимальных элементов. В этом алгоритме мы сначала разобьем элементы на пары и упорядочим их внутри пары. Затем из двух пар создадим упорядоченные четверки и т.д.

3	7	8	2	4	6	1	5	Последовательности длины 1
3 7		2 8		4 6		1 5		Слияние до упорядоченных пар
2 3 7 8				1 4 5 6				Слияние пар в упорядоченные четверки
1 2 3 4 5 6 7 8								Слияние пар в упорядоченные четверки

Интерес представляет сам процесс слияния: для каждой из половинок мы устанавливаем указатели на начало, смотрим, в какой из частей элемент по указателю меньше, записываем этот элемент в новый массив и перемещаем соответствующий указатель.

Опишем функцию слияния следующим образом:

```
void merge(int a[], int b[], int c, int d, int e)
{
  int p1=c, p2=d, pres=c;
  while (p1 < d && p2 < e)
  if (a[p1] < a[p2])
    b[pres++] = a[p1++];
  else
    b[pres++] = a[p2++];
  while (p1 < d)
    b[pres++]=a[p1++];
  while (p2 < e)
    b[pres++]=a[p2++];
}
```

Здесь a — исходный массив, b — массив результата, c и d — указатели на начало первой и второй части соответственно, e — указатель на конец второй части.

Далее опишем довольно хитрую нерекурсивную функцию сортировки слиянием:

```
void merge_sort(int a[], int n)
{
  int *temp, *a2=a, *b=(int*)malloc(n*sizeof(int)), *b2;
  int c, k = 1, d, e;
  b2=b;
  while (k <= 2*n) {
    for (c=0; c<n; c+=k*2) {
      d=c+k<n?c+k:n;
      e=c+2*k<n?c+2*k:n;
      merge(a2, b, c, d, e);
    }
    temp = a2; a2 = b; b = temp;
    k *= 2;
  }
  for (c=0; c<n; c++)
  a[c] = a2[c];
  free(b2);
}
```

Рекурсивная реализация сортировки слияними несколько проще, но обладает меньшей эффективностью и требует $O(\log N)$ дополнительной памяти.

Алгоритм имеет сложность $O(N \log N)$ и требует $O(N)$ дополнительной памяти.

В оригинале этот алгоритм был придуман для сортировки данных во внешней памяти (данные были расположены в файлах) и требует только последовательного доступа. Этот алгоритм применим для сортировки односвязных списков.

12 Сортировка подсчетом

Это сортировка может использоваться только для дискретных данных. Допустим, у нас есть числа от 0 до 99, которые нам следует отсортировать. Заведем массив размером в 100 элементов, в котором будем запоминать,

сколько раз встречалось каждое число (т.е. при появлении числа будем увеличивать элемент вспомогательного массива с индексом, равным этому числу, на 1). Затем просто пройдем по всем числам от 0 до 99 и выведем каждое столько раз, сколько оно встречалось. Сортировка реализуется следующим образом:

```
for (i=0; i<MAXV; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  c[a[i]]++;
k=0;
for (i=0; i<MAXV; i++)
  for (j=0; j<c[i]; j++)
    a[k++]=i;
```

Здесь $MAXV$ — максимальное значение, которое может встречаться (т.е. все числа массива должны лежать в пределах от 0 до $MAXV - 1$).

Алгоритм использует $O(MAXV)$ дополнительной памяти и имеет сложность $O(N + MAXV)$. Его применение дает отличный результат, если $MAXV$ намного меньше, чем количество элементов в массиве.

13 Поразрядная сортировка

Алгоритм сортировки подсчетом чрезвычайно привлекателен своей высокой производительностью, но она ухудшается при возрастании $MAXV$, также резко возрастают требования к дополнительной памяти. Фактически, невозможно осуществить сортировку подсчетом для переменных типа `unsigned int` ($MAXV$ при этом равно 2^{32}).

В качестве развития идеи сортировки подсчетом рассмотрим поразрядную сортировку. Сначала отсортируем числа по последнему разряду (единиц). Затем повторим то же самое для второго и последующих разрядов, пользуюсь каким либо устойчивым алгоритмом сортировки (т.е. если числа с одинаковым значением в сортируемом разряде шли в одном порядке, то в отсортированной последовательности они будут идти в том же порядке).

Для примера приведем таблицу, в первом столбце которой расположены исходные данные, а в последующих — результат сортировки по разрядам.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	837	657	839

Для самой сортировки будем использовать сортировку подсчетом. После этого будем

переделывать полученную таблицу так, чтобы для каждого возможного значения разряда сохранялась позиция, начиная с которой идут числа с таким значением в соответствующем разряде (т.е. сколько элементов имеют меньшее значение в этом разряде). Назовем этот массив c.

После этого будем проходить по всему исходному массиву, смотреть на текущее значение разряда (i), записывать текущее число во вспомогательный массив (b) на позицию $c[i]$, а затем увеличивать $c[i]$ (чтобы новое число с таким же значением текущего разряда не легло поверх уже записанного).

Пусть количество знаков в числе равно k, а количество возможных значений равно m (система счисления, использованная при записи числа). Тогда количество присваиваний, производимое алгоритмом, будет равно $O(k \times N + k \times m)$, а количество дополнительной памяти — $O(N + k \times m)$.

Приведем эффективную реализацию поразрядной сортировки для беззнаковых 4-байтных чисел (`unsigned int`). Мы будем использовать 4 разряда, каждый из которых равен байту (система счисления с основанием 256). Эта реализация использует несколько хитростей, которые будут пояснены ниже.

```
void radix_sort(unsigned int a[], int n)
{
  unsigned int *t, *a2=a;
  unsigned int *b=(unsigned int*) malloc(n*sizeof(int));
  unsigned int *b2;
  unsigned char *bp;
  int i, j, npos, temp;
  int c[256][4];
  b2 = b;
  memset(c, 0, sizeof(int)*256*4);
  bp = (unsigned char*) a;
  For(i,n)
    For (j,4) {
      c[*bp][j]++;
      bp++;
    }
  For(j,4) {
    npos = 0;
    For(i,256) {
      temp = c[i][j];
      c[i][j] = npos;
      npos += temp;
```

```
    }
  }
  For(j,4) {
    bp = (unsigned char*) a2 + j;
    For(i,n) {
      b[c[*bp][j]++] = a2[i];
      bp += 4;
    }
    t = a2; a2 = b; b = t;
  }
  free(b2);
}
```

Функция `memset` используется для заполнения заданной области памяти нулями (обнуление массива), она находится в библиотеке `string.h`. Всю таблицу сдвигов (c) мы будем строить заранее для всех 4 разрядов. Для эффективно доступа к отдельным байтам мы будем использовать указатель *bp* типа `unsigned char *` (тип `char` как раз занимает 1 байт и может трактоваться как число). Затем мы формируем модифицированную таблицу и проводим собственно функцию расстановки чисел по всем 4 разрядам.

Внимательный читатель заметит в приведенной функции несколько мест, которые на первый взгляд кажутся ошибочными. Хотя в текстовом описании мы и говорили, что следует сортировать, начиная с последних разрядов, в реализации мы начинаем с первых байтов. Это объясняется тем, что в архитектуре **x86** числа хранятся в «перевернутом» виде — это было сделано для совместимости с младшими моделями.

Второе возможное место для ошибки — массив a не ставится в соответствие указателю отсортированного массива и не осуществляется копирование отсортированных элементов в него в конце работы функции. Это связано с четным количеством разрядов, так что результат в итоге и так оказывается в массиве a.

Вообще говоря, далеко не обязательно так сильно связывать поразрядную сортировку с аппаратными средствами. Более того, основное удобство поразрядной сортировки состоит в там, что с ее помощью можно сортировать сложные структуры, такие как даты, строки (массивы) и другие структуры со многими полями.

14 Сравнение производительности сортировок

Название	**Сравнений**	**Присваиваний**	**Память**	**Время**
Пузырьком	$O(N^2)$	$O(N^2)$	0	$---$
Выбором	$O(N^2)$	$O(N)$	0	$---$
Пирамидальная	$O(N \log N)$	$O(N \log N)$	0	16437
Быстрая	$O(N \log N)$	$O(N \log N)$	$O(\log N)$	5618
Слиянием	$O(N \log N)$	$O(N \log N)$	$O(N)$	7669
Подсчетом	0	$O(N)$	$O(MAXV)$	322
Поразрядная	0	$O(kN + km)$	$O(N + km)$	1784

Тестирование проводилось на 10^7 случайных чисел, которые не превышали 10^5. Для квадратичных алгоритмов тестирование не проводилось (поскольку это заняло бы очень большое время). При тестировании использовались приведенные выше функции. В таблице k — количество «цифр», а m — количество значений, которое может принимать каждая цифра.

Быстрая сортировка в худшем случае имеет $O(N^2)$ сравнений и присваиваний.

Как видно из таблицы, поразрядная сортировка опережает все остальные на «обычных» данных, но для практического применения больше подходит быстрая сортировка, т.к. разница уменьшается по мере уменьшения количества сортируемых элементов, а само по себе считывание 10^7 элементов занимает время намного большее, чем разница во времени между этими сортировками.

При решении той или иной задачи следует выбирать нужный тип сортировки исходя из таблицы и специфических условий применимости (минимизация количества сравнений или присваиваний, сортировка списков или последовательный доступ к данным).

Еще одна практическая рекомендация заключается в следующем: при сортировке сложных структур (например, строк) следует не производить обмены, а просто переставлять указатели (строка может состоять из нескольких тысяч символов, а указатель занимает 4 байта и процесс сортировки может ускориться во много раз).