

# ADS problems

## Содержание

Отправка домашних работ	2
<b>1. Изучение языка C</b>	<b>2</b>
1.1 Полином (polynom.c) . . . . .	3
1.2 Произведение чисел по модулю (mulmod.c) . . . . .	3
1.3 Фибоначчиева система счисления (fibsys.c) . . . . .	4
1.4 Пересечение множеств (intersect.c) . . . . .	5
1.5 Перестановка элементов массива (permut.c) . . . . .	5
1.6 Максимальная сумма подряд идущих элементов массива (maxk.c) . . . . .	6
1.7 Наибольший простой делитель (primediv.c) . . . . .	6
1.8 Седловая точка в матрице (saddlepoint.c) . . . . .	6
1.9 Функция обращения массива (revarray.c) . . . . .	7
1.10 Функция поиска максимального элемента в массиве (maxarray.c) . . . . .	7
1.11 Функция бинарного поиска в последовательности (binsearch.c) . . . . .	7
1.12 Функция поиска пика в последовательности (peak.c) . . . . .	8
1.13 Конкатенация строк (concat.c) . . . . .	8
1.14 Подсчёт слов в строке (wcount.c) . . . . .	9
1.15 Фибоначчиевы строки (fibstr.c) . . . . .	9
1.16 Функция побитового сравнения строк (strdiff.c) . . . . .	10
1.17 Рисование рамки (frame.c) . . . . .	10
1.18 Функция поиска в лисповском списке (searchlist.c) . . . . .	11
<b>2. Алгоритмы сортировки и поиска</b>	<b>12</b>
2.1 Кратчайшая суперстрока (superstr.c) . . . . .	12
2.2 Суммы, образующие степени двойки (power2.c) . . . . .	12
2.3 Функция двунаправленной пузырьковой сортировки (bubblesort.c) . . . . .	13
2.4 Функция сортировки методом Шелла (shellsort.c) . . . . .	13
2.5 Сортировка подсчётом сравнений (csort.c) . . . . .	14
2.6 Пирамидальная сортировка (hsort.c) . . . . .	15
2.7 Сортировка слиянием + вставками (mergesort.c) . . . . .	15
2.8 Быстрая сортировка + сортировка прямым выбором (quicksort.c) . . . . .	15

2.9	Сортировка букв в строке (dsort.c) . . . . .	16
2.10	Поразрядная сортировка дат (datesort.c) . . . . .	16
2.11	Поразрядная сортировка целых чисел (radixsort.c) . . . . .	17
2.12	Периодические префиксы (prefixes.c) . . . . .	17
2.13	Поиск всех вхождений подстроки в строку (Кнут–Моррис– Пратт) (kmpall.c) . . . . .	18
2.14	Слово, составленное из префиксов другого слова (pword.c) . . . . .	18
2.15	Поиск всех вхождений подстроки в строку (Бойер–Мур) (bmall.c) . . . . .	19
2.16	Расширенная эвристика стоп-символа (extstop.c) . . . . .	19
2.17	Поиск максимального элемента подпоследовательности (rangemax.c) . . . . .	20
2.18	Определение гипердромов в строке (rangehd.c) . . . . .	20
2.19	Количество пиков в подпоследовательности (rangepeak.c) . . . . .	21
2.20	Наибольший общий делитель подпоследовательности (rangegcd.c) . . . . .	21
2.21	Максимальное произведение простых дробей (maxprod.c) . . . . .	22
<b>3.</b>	<b>Применение динамических множеств</b>	<b>22</b>
3.1	Нерекурсивная быстрая сортировка (qsstack.c) . . . . .	22
3.2	Стековая машина (stackmachine.c) . . . . .	23
3.3	Кольцевой буфер (circbuf.c) . . . . .	24
3.4	Очередь с операцией Maximum (qmax.c) . . . . .	24
3.4	Слияние последовательностей (merge.c) . . . . .	25
3.5	Моделирование работы вычислительного кластера (cluster.c) . . . . .	25
3.6	Сортировка списка вставками (listisort.c) . . . . .	26
3.7	Сортировка списка пузырьком (listbsort.c) . . . . .	27
3.8	Ранги элементов в списке с пропусками (ranklist.c) . . . . .	27
3.9	Ранги вершин бинарного дерева поиска (ranktree.c) . . . . .	28
3.10	Лексический анализ (lexavl.c) . . . . .	29
3.11	Разреженный массив (disparrray.c) . . . . .	30
3.12	Количество подпоследовательностей, на которых побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ даёт ноль (zeroxor.c) . . . . .	31
3.13	Строки с общими префиксами (ptrie.c) . . . . .	31

## Отправка домашних работ

- Домашние работы следует отправлять на ящик avkonov@bmstu.ru с ящика ...@student.bmstu.ru.
- Заголовок письма должен начинаться со слова HOMEWORK.
- Файл присылается во вложении. Имя файла должно быть таким, которое указано в задании.

## 1. Изучение языка C

## 1.1 Полином (polynom.c)

Составьте программу `polynom.c`, вычисляющую значение полинома

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

и его производной в заданной точке  $x_0$ . Коэффициенты полинома и значение  $x_0$  — целые числа в диапазоне от  $-2^{63}$  до  $2^{63} - 1$ .

Программа должна считывать из стандартного потока ввода степень полинома  $n$ , значение  $x_0$  и коэффициенты полинома  $a_n, a_{n-1}, \dots, a_0$ . В стандартный поток вывода нужно вывести значения  $P_n(x_0)$  и  $P'_n(x_0)$ .

Для вычисления значения полинома нужно использовать схему Горнера:

$$P_n(x) = (\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0.$$

Например, согласно схеме Горнера

$$P_4(x) = 3x^4 + 2x^3 - 5x^2 + x - 4 = (((3x + 2)x - 5)x + 1)x - 4.$$

Для вычисления значения производной полинома необходимо очевидным образом модифицировать схему Горнера.

## 1.2 Произведение чисел по модулю (mulmod.c)

Составьте программу `mulmod.c`, вычисляющую выражение  $(a \cdot b) \bmod m$ , то есть остаток от деления произведения чисел  $a$  и  $b$  на  $m$ . Известно, что  $a$ ,  $b$  и  $m$  — натуральные числа, меньшие, чем  $2^{63}$ , причем  $m \neq 0$ .

Программа должна считывать из стандартного потока ввода числа  $a$ ,  $b$  и  $m$  и выводить результат в стандартный поток вывода.

Основная сложность этой задачи заключается в том, что произведение  $a$  на  $b$  может превышать  $2^{64}$  и, тем самым, не помещаться ни в один из целочисленных типов данных языка C. При этом представление формулы в виде

$$((a \bmod m) \cdot (b \bmod m)) \bmod m$$

тоже не решает проблемы, так как при больших значениях  $m$  произведение

$$(a \bmod m) \cdot (b \bmod m)$$

тоже может превышать  $2^{64}$ .

Решение этой задачи сводится к вычислению значения полинома по схеме Горнера. Представим число  $b$  в двоичной системе счисления:

$$b = \overline{b_{63}b_{62} \dots b_1b_0}.$$

Здесь  $b_{63}, b_{62}, \dots, b_1, b_0$  — двоичные разряды, формирующие число, то есть

$$b = b_{63} \cdot 2^{63} + b_{62} \cdot 2^{62} + \dots + b_1 \cdot 2 + b_0.$$

Тогда

$$(a \cdot b) \bmod m = [a \cdot b_{63} \cdot 2^{63} + a \cdot b_{62} \cdot 2^{62} + \dots + a \cdot b_1 \cdot 2 + a \cdot b_0] \bmod m.$$

Преобразовав это выражение по схеме Горнера, получим

$$(a \cdot b) \bmod m = [(\dots (a \cdot b_{63} \cdot 2 + a \cdot b_{62}) \cdot 2 + \dots + a \cdot b_1) \cdot 2 + a \cdot b_0] \bmod m.$$

Учитывая, что для любых  $x, y$  и  $m \neq 0$  справедливы тождества

$$\begin{aligned} (x + y) \bmod m &\equiv ((x \bmod m) + (y \bmod m)) \bmod m, \\ (x \cdot y) \bmod m &\equiv ((x \bmod m) \cdot (y \bmod m)) \bmod m, \end{aligned}$$

мы имеем право при вычислении правой части нашей формулы поступать следующим образом: если есть возможность того, что сумма двух слагаемых превзойдёт  $2^{64}$ , нужно складывать остатки от деления этих слагаемых на  $m$ ; аналогично для произведения. Этот приём даёт гарантию того, что при вычислении ни разу не произойдёт переполнение.

### 1.3 Фибоначчиева система счисления (fibsys.c)

Числа Фибоначчи — это последовательность натуральных чисел  $\{f_i\}$ , в которой

$$\begin{cases} f_0 = f_1 = 1, \\ f_i = f_{i-2} + f_{i-1}, \quad \forall i > 1. \end{cases}$$

По теореме Цекендорфа любое положительное целое число может быть единственным образом представлено в виде суммы различных чисел Фибоначчи, не включающей двух подряд идущих чисел Фибоначчи.

Например, для числа 100 суммой Цекендорфа будет  $89 + 8 + 3$ . Все остальные суммы различных чисел Фибоначчи, равные 100, содержат два подряд идущих числа Фибоначчи.

Чтобы разложить целое положительное число  $x$  в сумму Цекендорфа, нужно до тех пор, пока  $x > 0$ , выполнять следующие действия:

- находим максимальное число Фибоначчи  $f \leq x$ ;
- добавляем  $f$  в сумму Цекендорфа;
- вычитаем  $f$  из  $x$ .

Напомним, что в позиционной системе счисления вес каждой цифры зависит от её позиции в записи числа. Соответственно, базисом позиционной системы счисления называется последовательность чисел, задающих вес каждого разряда. Например, для десятичной системы базисом является последовательность 1, 10, 100, 1000, 10000, 100000, и т.д.

**Фибоначчиева система счисления** — это позиционная система счисления с двумя цифрами — 0 и 1, базисом которой являются числа Фибоначчи, начиная с  $f_1$ : 1, 2, 3, 5, 8, 13, 21, 34, и т.д.

Для того чтобы гарантировать единственность записи числа в фибоначчиевой системе счисления, запись числа не должна содержать две подряд идущие единицы (теорема Цекендорфа).

Составьте программу `fibsys.c`, выполняющую перевод целого числа  $0 \leq x < 2^{63}$  в фибоначчиеву систему счисления.

Программа должна считывать из стандартного потока ввода число  $x$  и выводить в стандартный поток вывода последовательность нулей и единиц, образующую запись числа  $x$  в фибоначчиевой системе счисления.

## 1.4 Пересечение множеств (`intersect.c`)

Пусть  $\mathbb{N}_{32}$  — множество натуральных чисел от 0 до 31. Даны два множества  $A \subseteq \mathbb{N}_{32}$  и  $B \subseteq \mathbb{N}_{32}$ . Составьте программу `intersect.c`, вычисляющую пересечение множеств  $A$  и  $B$ .

Программа должна считывать из стандартного потока ввода размер множества  $A$  и элементы множества  $A$ , а затем — размер множества  $B$  и элементы множества  $B$ . Программа должна выводить в стандартный поток вывода элементы множества  $A \cap B$ , отсортированные в порядке возрастания.

Использовать массивы для хранения множеств запрещается: каждое множество должно быть представлено 32-разрядным целым числом таким, что если его  $i$ -й бит равен 1, то число  $i$  принадлежит множеству.

## 1.5 Перестановка элементов массива (`permut.c`)

Даны два целочисленных массива размером 8 элементов. Составьте программу `permut.c`, определяющую, можно ли получить один массив перестановкой элементов другого массива.

Программа должна считывать из стандартного потока ввода элементы обоих массивов, а затем выводить в стандартный поток вывода слово «yes», если массивы совпадают с точностью до перестановки элементов, и «no» — в противном случае.

Сортировать массивы запрещается.

### 1.6 Максимальная сумма подряд идущих элементов массива (`maxk.c`)

Дан целочисленный массив, размер которого не превышает 1000000, и число  $k$ , которое меньше или равно длине массива. Составьте программу `maxk.c`, определяющую, какие  $k$  подряд идущих элементов массива имеют максимальную сумму.

Программа должна считывать из стандартного потока ввода размер массива, его элементы и число  $k$ , а затем выводить в стандартный поток вывода максимальную сумму  $k$  подряд идущих элементов массива.

В программе запрещается обращаться к одному и тому же элементу массива более двух раз, а также объявлять какие бы то ни было вспомогательные массивы.

### 1.7 Наибольший простой делитель (`primediv.c`)

Составьте программу `primediv.c`, вычисляющую наибольший простой делитель некоторого числа  $x$ . Число  $x$  вводится со стандартного потока ввода, причём известно, что  $-2^{31} \leq x < 2^{31}$ .

Программа должна использовать решето Эратосфена для построения массива простых чисел.

### 1.8 Седловая точка в матрице (`saddlepoint.c`)

Элемент двумерного массива называется седловым, если он одновременно наибольший в своей строке и наименьший в своём столбце.

Дан целочисленный двумерный массив, размер которого не превышает  $10 \times 10$ . Известно, что все элементы массива различны. Составьте программу `saddlepoint.c`, определяющую седловую точку в этом массиве.

Программа должна считывать из стандартного потока ввода количество строк и столбцов двумерного массива и его элементы. Программа должна вывести в стандартный поток координаты найденной седловой точки или слово «none», если седловой точки не существует.

В программе запрещается обращаться к одному и тому же элементу массива дважды.

## 1.9 Функция обращения массива (revarray.c)

Составьте функцию revarray, переставляющую элементы любого массива в обратном порядке. Функция должна быть объявлена как

```
void revarray(void *base, size_t nel, size_t width)
{
    ...
}
```

Здесь параметр base означает указатель на начало массива, nel — количество элементов в массиве, а width — размер каждого элемента массива в байтах.

**Примечание.** Тип size\_t определён в заголовочных файлах <stddef.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h> и <wchar.h>.

## 1.10 Функция поиска максимального элемента в массиве (maxarray.c)

Составьте функцию maxarray, возвращающую индекс максимального элемента произвольного массива. Функция должна быть объявлена как

```
int maxarray(void *base, size_t nel, size_t width,
             int (*compare)(void *a, void *b))
{
    ...
}
```

Здесь параметр base означает указатель на начало массива, nel — количество элементов в массиве, width — размер каждого элемента массива в байтах, а compare — указатель на функцию сравнения двух элементов, работающую аналогично функции сравнения для библиотечной функции qsort.

**Примечание.** Тип size\_t определён в заголовочных файлах <stddef.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h> и <wchar.h>.

## 1.11 Функция бинарного поиска в последовательности (binsearch.c)

Составьте функцию binsearch, выполняющую поиск заданного числа в последовательности чисел, отсортированной по возрастанию, методом деления пополам. Функция должна быть объявлена как

```
unsigned long binsearch(unsigned long nel, int (*compare)(unsigned long i))
{
    ...
}
```

Здесь параметр `nel` задаёт количество элементов в последовательности, а параметр `compare` — указатель на функцию сравнения, которая принимает параметр `i` и и возвращает:

- `-1`, если `i`-тое число в последовательности меньше искомого числа;
- `0`, если они равны;
- `1`, если `i`-тое число больше искомого числа.

Функция `binsearch` должна возвращать индекс найденного элемента или значение `nel`, если такого элемента не существует.

### 1.12 Функция поиска пика в последовательности (`peak.c`)

Элемент последовательности чисел, значение которого — не меньше значений его непосредственных соседей, называется **пиком**. Очевидно, что непустая последовательность размера  $n$  имеет от 1 до  $n$  пиков.

Составьте функцию `peak`, возвращающую индекс любого пика в последовательности. Функция должна быть объявлена как

```
unsigned long peak(unsigned long nel,
                   int (*less)(unsigned long i, unsigned long j))
{
    ...
}
```

Здесь параметр `nel` задаёт количество элементов в последовательности, а параметр `less` — указатель на функцию сравнения, которая принимает два параметра — `i` и `j` — и возвращает 1, если `i`-тое число в последовательности меньше `j`-того числа, и 0 — в противном случае.

### 1.13 Конкатенация строк (`concat.c`)

Составьте функцию `concat`, выполняющую конкатенацию произвольного количества строк:

```
char *concat(char **s, int n)
{
    ...
}
```

Здесь `s` — указатель на массив соединяемых строк, `n` — количество строк в массиве. Функция должна создавать в динамической памяти новую строку, размер которой равен суммарному размеру всех соединяемых строк, записывать в неё соединяемые строки друг за другом в том порядке, в котором они перечислены в массиве, и возвращать указатель на новую строку.

Программа `concat.c`, демонстрирующая работоспособность функции `concat`, должна считывать со стандартного потока ввода количество строк



и сами соединяемые строки и выводить в стандартный поток вывода результирующую строку.

В автотестах длины строк, подаваемых на стандартный ввод, не превышают 1000 символов. Однако, аргументы функции `concat()` могут иметь произвольную длину.

### 1.14 Подсчёт слов в строке (`wcount.c`)

Составьте функцию `wcount`, вычисляющую количество слов в строке. Слово — это подстрока, не содержащая пробелов. Слова разделяются произвольным количеством пробелов. Кроме того, строка может начинаться и заканчиваться произвольным количеством пробелов. Объявление функции должно выглядеть как

```
int wcount(char *s)
{
    ...
}
```

Итоговую программу, содержащую как функцию `wcount`, так и функцию `main`, демонстрирующую работоспособность функции `wcount`, нужно назвать `wcount.c`. Строка должна считываться из стандартного потока ввода с помощью функции `gets` или `fgets`, минимальный размер буфера — 1000 символов, включая концевой ноль.

**Примечание.** Функция `fgets`, в отличие от `gets`, добавляет `\n` в конец прочитанной строки.

### 1.15 Фибоначчиевы строки (`fibstr.c`)

**Строки Фибоначчи** — это последовательность строк  $\{s_i\}$ , составленных из букв  $a$  и  $b$ , в которой

$$\begin{cases} s_1 = a, \\ s_2 = b, \\ s_i = s_{i-2}s_{i-1}, \quad \forall i > 2. \end{cases}$$

Чтобы было понятнее, приведём первые пять строк Фибоначчи:  $a, b, ab, bab, abbab$ .

Составьте функцию `fibstr`, возвращающую  $n$ -ную строку Фибоначчи. Функция `fibstr` должна быть объявлена как

```
char *fibstr(int n)
{
    ...
}
```

Функция `fibstr` должна выделять в динамической памяти массив такого размера, чтобы в него как раз поместилась  $n$ -ная строка Фибоначчи. Ответственность за освобождение памяти, занятой строкой, лежит на функции, вызывающей функцию `fibstr`.

Составьте программу `fibstr.c`, демонстрирующую работоспособность составленной функции.

### 1.16 Функция побитового сравнения строк (`strdiff.c`)

Составьте функцию `strdiff`, выполняющую побитовое сравнение двух строк. Функция должна быть объявлена как

```
int strdiff(char *a, char *b)
{
    ...
}
```

Если строки равны, функция должна возвращать  $-1$ . В противном случае, она должна возвращать порядковый номер бита, до которого содержимое строк совпадает.

Например, строки `aa` и `ai` представлены следующими последовательностями битов (последовательности записаны справа налево, то есть младший бит — самый правый):

00000000	$\underbrace{01100001}_a$	$\underbrace{01100001}_a$
00000000	$\underbrace{01101001}_i$	$\underbrace{01100001}_a$

Эти строки совпадают до 11 бита (биты нумеруются, начиная с 0).

### 1.17 Рисование рамки (`frame.c`)

Составьте программу `frame.c`, выполняющую рисование рамки вокруг текстовой строки. Программа должна принимать в качестве аргументов командной строки размеры рамки и значение строки.

Например, пусть программа вызвана как

```
./frame 6 20 Abracadabra
```

Тогда в стандартный поток вывода должно быть выведено

```
*****
*           *
*  Abracadabra  *
*           *
*           *
```

```
*
*
*****
```

Текстовая строка должна быть отцентрирована как по горизонтали, так и по вертикали. В случае, если длина строки не позволяет вписать строку в рамку заданного размера, программа должна вместо рамки выводить сообщение `Error`.

Если программа вызвана с неправильным количеством аргументов командной строки, необходимо вывести подсказку

Usage: `frame <height> <width> <text>`

Из-за ограничений системы автоматического тестирования код возврата программы всегда должен быть равен 0, даже при ошибочных данных.

### 1.18 Функция поиска в лисповском списке (`searchlist.c`)

Пусть в гипотетическом интерпретаторе языка Lisp элемент списка представлен в памяти структурой `Elem`, которая объявлена в заголовочном файле `elem.h` следующим образом:

```
#ifndef ELEM_H_INCLUDED
#define ELEM_H_INCLUDED

struct Elem {
    /* «Тег», описывающий тип значения в «голове» списка */
    enum {
        INTEGER,
        FLOAT,
        LIST
    } tag;

    /* Само значение в «голове» списка */
    union {
        int i;
        float f;
        struct Elem *list;
    } value;

    /* Указатель на «хвост» списка */
    struct Elem *tail;
};

#endif
```

Таким образом, в качестве полезной нагрузки в элементе списка может храниться целое число, число с плавающей точкой или указатель на список. Причём тип хранимого значения определяется полем `tag`.

Кроме того, в элементе списка хранится указатель `tail` на «хвост» списка. Если элемент является последним в списке, этот указатель принимает значение `NULL`.

Составьте функцию `searchlist`, выполняющую поиск элемента списка, содержащего указанное целое число:

```
struct Elem *searchlist(struct Elem *list, int k)
{
    ...
}
```

Здесь параметр `list` означает указатель на первый элемент списка, `k` — искомое целое число.

Функция `searchlist` должна возвращать указатель на найденный элемент списка или `NULL`, если элемент, содержащий число `k`, не найден.

**Указание.** Добавьте в файл с решением `#include "elem.h"` (он есть в среде тестирования) или целиком определение структуры `struct Elem`, скопировав её из условия задачи. Скачать файл: `elem.h`.

## 2. Алгоритмы сортировки и поиска

### 2.1 Кратчайшая суперстрока (`superstr.c`)

Пусть дано множество из  $n$  строк, где  $0 < n \leq 10$ . Известно, что ни одна из этих строк не является подстрокой другой строки. Составьте программу `superstr.c`, вычисляющую длину кратчайшей строки, содержащей все эти строки в качестве подстрок.

Программа должна считывать из стандартного потока ввода число  $n$ , а затем  $n$  строк. Длина кратчайшей строки, содержащей все  $n$  считанных строк, должна выводиться в стандартный поток вывода.

### 2.2 Суммы, образующие степени двойки (`power2.c`)

Пусть дана последовательность из  $n$  неповторяющихся целых чисел, где  $0 \leq n \leq 24$ , и каждое целое число находится в диапазоне от  $-10^6$  до  $10^6$ .

Составьте программу `power2.c`, вычисляющую, сколько существует непустых сочетаний чисел из последовательности таких, что сумма чисел в сочетании равна степени числа 2.

Программа должна считывать из стандартного потока ввода число  $n$ , а затем  $n$  чисел, образующих последовательность. Программа должна вывести количество сочетаний в стандартный поток вывода.

## 2.3 Функция двунаправленной пузырьковой сортировки (bubblesort.c)

В классической сортировке пузырьком проход по сортируемой последовательности осуществляется всегда в одном направлении. Модифицируйте алгоритм сортировки пузырьком, чтобы в нём чередовались проходы по последовательности слева направо и справа налево.

Составьте функцию `bubblesort`, осуществляющую двунаправленную пузырьковую сортировку произвольной последовательности. Функция должна быть объявлена как

```
void bubblesort(unsigned long nel,
                int (*compare)(unsigned long i, unsigned long j),
                void (*swap)(unsigned long i, unsigned long j))
{
    ...
}
```

Параметры функции `bubblesort`:

- `nel` — количество элементов в последовательности;
- `compare` — указатель на функцию сравнения, которая возвращает  $-1$ , если  $i$ -й элемент меньше  $j$ -го,  $0$  — в случае, если  $i$ -й элемент равен  $j$ -му, и  $1$  — в случае, если  $i$ -й элемент больше  $j$ -го;
- `swap` — указатель на функцию обмена  $i$ -го и  $j$ -го элементов последовательности.

## 2.4 Функция сортировки методом Шелла (shellsort.c)

В классической сортировке вставками для вставки элемента в отсортированную часть последовательности выполняется сравнение элемента со всеми членами отсортированной части до тех пор, пока для него не будет найдено место, то есть переменная `loc` (см. алгоритм в лекциях) на каждой итерции внутреннего цикла уменьшается на единицу.

Метод Шелла является модификацией сортировки вставками, в которой переменная `loc` на каждой итерции внутреннего цикла уменьшается на некоторое число  $d \geq 1$ . При этом фактически сортировка выполняется несколько раз для всё меньших и меньших значений  $d$  до тех пор, пока  $d$  не станет равно  $1$ . Тем самым, сначала выполняется серия «грубых» сортировок, которые не дают точного ответа, но делают последовательность более упорядоченной, обеспечивая более быстрое выполнение финальной точной сортировки при  $d = 1$ .

Составьте функцию `shellsort`, выполняющую сортировку произвольной последовательности методом Шелла. Функция `shellsort` должна быть объявлена как

```

void shellsort(unsigned long nel,
               int (*compare)(unsigned long i, unsigned long j),
               void (*swap)(unsigned long i, unsigned long j))
{
    ...
}

```

Параметры функции shellsort:

- `nel` — количество элементов в последовательности;
- `compare` — указатель на функцию сравнения, которая возвращает  $-1$ , если  $i$ -й элемент меньше  $j$ -го,  $0$  — в случае, если  $i$ -й элемент равен  $j$ -му, и  $1$  — в случае, если  $i$ -й элемент больше  $j$ -го;
- `swap` — указатель на функцию обмена  $i$ -го и  $j$ -го элементов последовательности.

Значения расстояния  $d$  в ходе работы функции должны образовывать последовательность Фибоначчи (естественно, записанную задом наперёд). Первое значение в этой последовательности должно быть максимальным числом Фибоначчи, которое меньше значения параметра `nel`.

## 2.5 Сортировка подсчётом сравнений (`csort.c`)

Составьте функцию `csort`, выполняющую сортировку слов в предложении методом подсчёта сравнений. Слова в предложении разделяются произвольным количеством пробелов. Функция `csort` должна быть объявлена следующим образом:

```

void csort(char *src, char *dest)
{
    ...
}

```

В качестве параметров функция `csort` принимает указатель на исходное предложение `src` и указатель на пустой буфер `dest` подходящего размера. В результате работы функции в буфер `dest` записывается новое предложение, состоящее из слов, взятых из исходного предложения и отсортированных в порядке возрастания их длин. При этом слова в новом предложении разделяются одним пробелом.

Рассмотрим пример работы функции `csort`. Пусть исходное предложение выглядит как

```
qqq www t aa rrr bb x y zz
```

Тогда в выходной буфер должно быть записано предложение

```
t x y aa bb zz qqq www rrr
```

Итоговую программу, содержащую как функцию `csort`, так и функцию `main`, демонстрирующую работоспособность функции `csort`, нужно назвать

`csort.c`. Программа должна считывать исходное предложение со стандартного потока ввода.

Минимальный размер буфера для ввода с консоли должен быть равен 1000 символов включая `\0` в конце строки.

## 2.6 Пирамидальная сортировка (`hsort.c`)

Составьте функцию `hsort`, выполняющую пирамидальную сортировку произвольного массива. Объявление функции `hsort` должно быть выполнено по аналогии с функцией `qsort`:

```
void hsort(void *base, size_t nel, size_t width,
           int (*compare)(const void *a, const void *b))
{
    ...
}
```

В качестве параметров функция `hsort` принимает указатель на начало массива `base`, количество элементов массива `nel`, размер одного элемента `width` и указатель на функцию сравнения `compare`.

Итоговая программа `heapsort.c` должна сортировать массив строк в порядке возрастания количества букв `a` в строке. Программа должна считывать из стандартного потока ввода размер и элементы массива, и выводить в стандартный поток вывода результат сортировки.

Минимальный размер буфера для ввода с консоли должен быть равен 1000 символов включая `\0` в конце строки.

## 2.7 Сортировка слиянием + вставками (`mergesort.c`)

Составьте программу `mergesort.c`, осуществляющую сортировку массива целых чисел в порядке возрастания модуля числа.

В программе должен быть реализован алгоритм сортировки слиянием, рекурсивную функцию которого нужно модифицировать таким образом, чтобы для последовательностей длиной меньше 5 выполнялась сортировка вставками.

Размер и элементы массива должны считываться программой из стандартного потока ввода. Результат сортировки должен быть выведен в стандартный поток вывода.

## 2.8 Быстрая сортировка + сортировка прямым выбором (`quicksort.c`)

Составьте программу `quicksort.c`, осуществляющую сортировку массива целых чисел в порядке возрастания.

В программе должен быть реализован алгоритм быстрой сортировки, рекурсивную функцию которого нужно модифицировать таким образом, чтобы, во-первых, для последовательностей длиной меньше  $m$  выполнялась сортировка прямым выбором, а во-вторых, глубина стека вызовов была равна  $O(\lg n)$ , где  $n$  — размер массива.

Программа должна считывать со стандартного потока ввода размер массива  $n$ , число  $m$  и значения элементов массива. В стандартный поток вывода должны выводиться элементы отсортированного массива.

## 2.9 Сортировка букв в строке (dsort.c)

Составьте программу `dsort.c`, осуществляющую сортировку латинских букв в строке в алфавитном порядке (размер строки — до миллиона букв). В программе должен быть реализован алгоритм сортировки распределением.

Например, если введена строка

`encyclopedia`

то программа должна вывести в стандартный поток вывода

`accdeeilnopy`

Строка вводится со стандартного потока ввода, причём известно, что она содержит только маленькие латинские буквы.

## 2.10 Поразрядная сортировка дат (datesort.c)

Составьте программу `datesort.c`, осуществляющую сортировку последовательности дат по возрастанию. В программе должен быть реализован алгоритм поразрядной сортировки, адаптированный для случая, когда ключи представляются в системе счисления с основаниями, зависящими от разряда.

В программе сортируемая последовательность должна быть представлена в виде массива структур `Date`:

```
struct Date {  
    int Day, Month, Year;  
};
```

Поле `Day` может принимать значения от 1 до 31, поле `Month` — от 1 до 12, а поле `Year` — от 1970 до 2030.

Последовательность дат считывается из стандартного потока ввода. При этом в самом начале считывается общее количество дат, а каждая дата представляется тройкой чисел «`yyyy mm dd`».

Например, если введена последовательность



```
5
2005 01 12
1977 02 01
1994 03 01
2004 02 29
1977 08 01
```

то программа должна выводить в стандартный поток вывода

```
1977 02 01
1977 08 01
1994 03 01
2004 02 29
2005 01 12
```

**Указание.** Формат вывода для `printf()`: `"%04d %02d %02d"`.

### 2.11 Поразрядная сортировка целых чисел (`radixsort.c`)

Составьте программу `radixsort.c`, осуществляющую сортировку последовательности 32-разрядных целых чисел по возрастанию. В программе должен быть реализован алгоритм поразрядной сортировки.

Программа должна считывать из стандартного потока ввода размер и элементы последовательности, и записывать в стандартный поток вывода элементы отсортированной последовательности.

Например, если на вход программы подано

```
5
1000 700 -5000 2038 0
```

то программа должна выводить в стандартный поток вывода

```
-5000 0 700 1000 2038
```

В программе сортируемая последовательность должна быть представлена в виде массива объединений `Int32`:

```
union Int32 {
    int x;
    unsigned char bytes[4];
};
```

Тем самым подразумевается, что целые числа представлены в системе счисления по основанию 256. Доступ к отдельным байтам целого числа должен осуществляться через поле `bytes` объединения.

### 2.12 Периодические префиксы (`prefixes.c`)

Составьте программу `prefixes.c`, выполняющую поиск всех периодических префиксов заданной строки  $S$ . Префикс является периодическим, если его

можно представить в виде

$$\underbrace{dd \dots d}_k,$$

где  $d$  — некоторая подстрока. Для поиска префиксов программа должна строить префиксную функцию для строки  $S$ .

Программа получает строку  $S$  через аргументы командной строки, и для каждого найденного префикса выводит в стандартный поток вывода два числа: длину префикса  $n$  и количество повторений  $k$  подстроки  $d$  в префиксе.

Например, пусть программа вызвана как

```
./prefixes aabaabaabaab
```

Тогда программа должна выводить в стандартный поток вывода

```
2 2
6 2
9 3
12 4
```

**Уточнение.** Если один и тот же префикс можно разложить на одинаковые подстроки разными способами, то величина  $k$  должна быть наибольшей, и, соответственно,  $d$  — кратчайшей. Поэтому в примере выше не печатается пара чисел

```
12 2
```

### 2.13 Поиск всех вхождений подстроки в строку (Кнут–Моррис–Пратт) (kmpall.c)

Составьте программу kmpall.c, осуществляющую поиск всех вхождений подстроки  $S$  в строку  $T$ . В программе должен быть реализован алгоритм Кнута–Морриса–Пратта, изменённый таким образом, чтобы при нахождении очередного вхождения  $S$  в  $T$  алгоритм не завершался, а продолжал сканировать строку  $T$ .

Строки  $S$  и  $T$  должны передаваться в программу через аргументы командной строки. Программа должна выводить в стандартный поток вывода индексы первых символов всех вхождений  $S$  в  $T$ .

### 2.14 Слово, составленное из префиксов другого слова (pword.c)

Составьте программу pword.c, определяющую, составлена ли строка  $T$  исключительно из префиксов строки  $S$ .

Программа получает строки  $S$  и  $T$  через аргументы командной строки и выводит «yes», если  $T$  составлена из префиксов  $S$ , и «no» — в противном случае.

Например, пусть программа вызвана как

```
./pword abracadabra abrabracada
```

Тогда программа должна выводить в стандартный поток вывода «yes».

## 2.15 Поиск всех вхождений подстроки в строку (Бойер–Мура) (bmall.c)

Составьте программу bmall.c, осуществляющую поиск всех вхождений подстроки  $S$  в строку  $T$ . В программе должен быть реализован алгоритм Бойера–Мура, изменённый таким образом, чтобы при нахождении очередного вхождения  $S$  в  $T$  алгоритм не завершался, а продолжал сканировать строку  $T$ .

Строки  $S$  и  $T$  должны передаваться в программу через аргументы командной строки. Программа должна выводить в стандартный поток вывода индексы первых символов всех вхождений  $S$  в  $T$ .

Строки  $S$  и  $T$  могут состоять из произвольных печатных знаков (коды 33...126).

## 2.16 Расширенная эвристика стоп-символа (extstop.c)

Существует модификация алгоритма Бойера–Мура, в которой эвристика стоп-символа расширена следующим образом:

**Расширенная эвристика стоп-символа.** Встретив в строке  $T$  символ  $x = T[k]$  такой, что  $x \neq S[i]$ , мы можем расположить строку  $S$  относительно строки  $T$  так, что последнее вхождение  $x$  в  $S$ , расположенное левее  $S[i]$ , окажется напротив  $T[k]$ .

**Пример.** ( $T[2] = a$  — стоп-символ)

	0	1	2	3	4	5	6	7	8	9	...
$T =$	a	b	<u>a</u>	a	b	a	b	a	c	b	...
$S =$	c	a	<b>&lt;b&gt;</b>	<b>a</b>	<b>b</b>						
		c	<u>a</u>	b	a	b					
					.	.	.				

Таблица  $\delta_1$  для эффективной реализации расширенной эвристики стоп-символа должна представлять собой матрицу размера  $len(S) \times size$ , где  $size$  — размер алфавита. При неудачном сравнении символов  $S[i]$  и  $T[k]$  алгоритм Бойера–Мура должен прочесть смещение для переменной  $k$  из  $\delta_1[i, T[k]]$ .

Составьте программу `extstop.c`, осуществляющую поиск первого вхождения подстроки  $S$  в строку  $T$ . В программе должен быть реализован вариант алгоритма Бойера–Мура, в котором не используется эвристика совпавшего суффикса, а эвристика стоп-символа расширена приведённым выше способом.

Строки  $S$  и  $T$  должны передаваться в программу через аргументы командной строки. Программа должна вывести в стандартный поток вывода индекс первого символа первого вхождения  $S$  в  $T$ . Если такого вхождения нет, программа должна вывести  $len(T)$ .

Строки  $S$  и  $T$  могут состоять из произвольных печатных знаков (коды 33...126).

### 2.17 Поиск максимального элемента подпоследовательности (`rangemax.c`)

Составьте программу `rangemax.c`, выполняющую поиск максимального элемента на различных интервалах последовательности целых чисел, которая время от времени изменяется.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит размер последовательности  $n$  ( $0 < n \leq 1000000$ ). Во второй строке перечислены элементы последовательности. Каждый элемент представляет собой целое число, находящееся в диапазоне от  $-10^9$  до  $10^9$ . Элементы разделяются пробелами.

Третья строка содержит общее количество выполняемых операций  $m$  ( $0 < m \leq 20000$ ). Каждая из следующих  $m$  строк содержит описание операции.

Операция либо имеет форму `MAX  $l$   $r$`  (найти максимальный элемент подпоследовательности, начинающейся с элемента с индексом  $l$  и заканчивающийся элементом с индексом  $r$ ), либо форму `UPD  $i$   $v$`  (присвоить значение  $v$  элементу с индексом  $i$ ).

**Формат результата работы программы.** Для каждой операции `MAX` вывести в стандартный поток вывода значение максимального элемента указанной подпоследовательности.

### 2.18 Определение гипердромов в строке (`rangehd.c`)

**Гипердром** — это строка, из букв которой можно составить палиндром. Другими словами, любая буква имеет чётное количество вхождений (возможно, нулевое) в гипердром чётной длины. Если же гипердром имеет нечётную длину, то ровно одна буква имеет нечётное количество вхождений.

Составьте программу `rangehd.c`, определяющую, является ли указанная подстрока строки гипердромом. Строка время от времени может изменяться.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит строку размера  $n$  ( $0 < n \leq 1000000$ ). Строка состоит из маленьких латинских букв.

Вторая строка содержит общее количество выполняемых операций  $m$  ( $0 < m \leq 10000$ ). Каждая из следующих  $m$  строчек содержит описание операции.

Операция либо имеет форму HD  $l\ r$  (определить, является ли подстрока, начинающаяся с индекса  $l$  и заканчивающаяся индексом  $r$ , гипердромом), либо форму UPD  $i\ s$  (заменить подстроку, начинающуюся с индекса  $i$ , строкой  $s$ ).

**Формат результата работы программы.** Для каждой операции HD вывести в стандартный поток вывода слово «YES», если подстрока является гипердромом, или слово «NO» в противном случае.

## 2.19 Количество пиков в подпоследовательности (rangepcak.c)

Напомним, что элемент последовательности чисел, значение которого — не меньше значений его непосредственных соседей, называется **пиком**.

Составьте программу rangepcak.c, выполняющую вычисление количества пиков на различных интервалах последовательности целых чисел, которая время от времени изменяется.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит размер последовательности  $n$  ( $0 < n \leq 1000000$ ). Во второй строке перечислены элементы последовательности. Каждый элемент представляет собой целое число, находящееся в диапазоне от  $-10^9$  до  $10^9$ . Элементы разделяются пробелами.

Третья строка содержит общее количество выполняемых операций  $m$  ( $0 < m \leq 20000$ ). Каждая из следующих  $m$  строк содержит описание операции.

Операция либо имеет форму PEAK  $l\ r$  (вычислить количество пиков в подпоследовательности, начинающейся с элемента с индексом  $l$  и заканчивающейся элементом с индексом  $r$ ), либо форму UPD  $i\ v$  (присвоить значение  $v$  элементу с индексом  $i$ ).

**Формат результата работы программы.** Для каждой операции PEAK вывести в стандартный поток вывода количество пиков в указанной подпоследовательности.

## 2.20 Наибольший общий делитель подпоследовательности (rangegcd.c)

Составьте программу rangegcd.c, вычисляющую наибольший общий делитель на различных интервалах последовательности целых чисел.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит размер последовательности  $n$  ( $0 < n \leq 300000$ ). Во второй строке перечислены элементы последовательности. Каждый элемент представляет собой целое число, находящееся в диапазоне от  $-10^9$  до  $10^9$ . Элементы разделяются пробелами.

Третья строка содержит общее количество запросов  $m$  ( $0 < m \leq 1000000$ ). Каждая из следующих  $m$  строк содержит запрос, который представляет собой два числа  $l$  и  $r$ , задающие границы интервала, на котором нужно вычислить наименьший общий делитель ( $0 \leq l, r < n$ ).

**Формат результата работы программы.** Для каждого запроса вывести в стандартный поток вывода наименьший общий делитель указанной подпоследовательности.

## 2.21 Максимальное произведение простых дробей (maxprod.c)

Составьте программу maxprod.c, выполняющую поиск отрезка последовательности простых дробей  $\{v_i\}_0^{n-1}$ , на котором произведение дробей максимально.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит размер последовательности  $n$  ( $0 < n \leq 1000000$ ). Во второй строке перечислены элементы последовательности. Каждый элемент записывается в виде  $a / b$ , где  $a$  и  $b$  — неотрицательные целые числа ( $0 \leq a \leq 1000000, 0 < b \leq 1000000$ ). Элементы разделяются пробелами.

**Формат результата работы программы.** Программа должна вывести в стандартный поток вывода два числа  $l$  и  $r$  такие, что произведение  $\prod_{i=l}^r v_i$  — максимально. Если возможно несколько решений, следует выбрать решение с минимальным  $l$ .

## 3. Применение динамических множеств

### 3.1 Нерекурсивная быстрая сортировка (qsstack.c)

Необходимо составить программу qsstack.c, осуществляющую сортировку массива целых чисел в порядке возрастания.

В программе должен быть реализован нерекурсивный алгоритм быстрой сортировки, использующий в своей работе стек заданий. Каждое задание описывает координаты подмассива, который нужно отсортировать, и представляет собой структуру

```
struct Task {  
    int low, high;  
};
```

Программа должна считывать со стандартного потока ввода размер массива  $n$  и значения элементов массива. В стандартный поток вывода должны выводиться элементы отсортированного массива.

### 3.2 Стековая машина (`stackmachine.c`)

Пусть **стековая машина** — это устройство для выполнения арифметических операций, использующее для хранения промежуточных результатов вычислений стек целых чисел. Подразумевается, что каждая операция берёт операнды из стека и оставляет на стеке результат.

Составьте программу `stackmachine.c`, моделирующую работу стековой машины.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Стековая машина должна обеспечивать выполнение следующих операций:

- **CONST**  $x$  — кладёт в стек число  $x$  ( $-1000000000 < x < 1000000000$ );
- **ADD** — сложение (снимает со стека два операнда  $a$  и  $b$  и кладёт в стек их сумму);
- **SUB** — вычитание (снимает со стека операнд  $a$ , затем снимает со стека операнд  $b$ , кладёт в стек  $a - b$ );
- **MUL** — умножение (снимает со стека два операнда  $a$  и  $b$  и кладёт в стек их произведение);
- **DIV** — деление (снимает со стека операнд  $a$ , затем снимает со стека операнд  $b$ , кладёт в стек результат целочисленного деления  $a$  на  $b$ );
- **MAX** — максимум двух чисел (снимает со стека два операнда  $a$  и  $b$  и кладёт в стек  $\max(a, b)$ );
- **MIN** — минимум двух чисел (снимает со стека два операнда  $a$  и  $b$  и кладёт в стек  $\min(a, b)$ );
- **NEG** — меняет знак числа, находящегося на вершине стека;
- **DUP** — кладёт в стек копию числа, находящегося на вершине стека;
- **SWAP** — меняет местами два числа, находящиеся на вершине стека.

Можно считать, что последовательность операций составлена правильно, то есть перед вызовом каждой операции стек содержит нужное ей количество операндов, деление на ноль и переполнение не возникают, и, кроме того, в результате выполнения всех операций на стеке остаётся единственное число.

**Формат результата работы программы.** В стандартный поток вывода необходимо вывести число, оставшееся на вершине стека в результате выполнения последовательности операций.

### 3.3 Кольцевой буфер (circbuf.c)

Реализуйте операции `InitQueue`, `QueueEmpty`, `Enqueue` и `Dequeue` для очереди целых чисел, представленной в виде кольцевого буфера. Начальный размер буфера – 4. В случае переполнения размер буфера должен увеличиваться в два раза.

Составьте программу `circbuf.c`, демонстрирующую работоспособность реализованных операций.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму `ENQ  $x$`  (добавить число  $x$  в хвост очереди,  $-2000000000 < x < 2000000000$ ), либо форму `DEQ` (удалить головной элемент из очереди), либо форму `EMPTY` (проверить пустоту очереди).

Можно считать, что последовательность операций составлена правильно, то есть перед каждой операцией `DEQ` очередь не пуста.

**Формат результата работы программы.** Для каждой операции `DEQ` вывести в стандартный поток вывода значение удаляемого головного элемента очереди. Для каждой операции `EMPTY` вывести в стандартный поток вывода «true» или «false» в зависимости от того, пуста очередь или нет.

### 3.4 Очередь с операцией Maximum (qmax.c)

Реализуйте через двойной стек набор операций `InitQueue`, `Enqueue`, `Dequeue`, `QueueEmpty` и `Maximum` для работы с очередью целых чисел. Операция `Maximum` возвращает максимальное целое число, в данный момент времени находящееся в очереди. Операции `Enqueue`, `QueueEmpty` и `Maximum` должны работать за константное время, а операция `Dequeue` — за амортизированное константное время.

Составьте программу `qmax.c`, демонстрирующую работоспособность реализованных операций.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму `ENQ  $x$`  (добавить число  $x$  в хвост очереди,  $-2000000000 < x < 2000000000$ ), либо форму `DEQ` (удалить головной элемент из очереди), либо форму `MAX` (показать текущее максимальное число), либо форму `EMPTY` (проверить пустоту очереди).

Можно считать, что последовательность операций составлена правильно, то есть перед каждой операцией `DEQ` очередь не пуста.



**Формат результата работы программы.** Для каждой операции DEQ вывести в стандартный поток вывода значение удаляемого головного элемента очереди. Для каждой операции MAX вывести в стандартный поток вывода текущее максимальное число. Для каждой операции EMPTY вывести в стандартный поток вывода «true» или «false» в зависимости от того, пуста очередь или нет.

**Подсказка.** Как реализовать стек с операциями Push, Pop и Maximum за константное время?

### 3.4 Слияние последовательностей (merge.c)

Составьте программу merge.c, объединяющую  $k$  отсортированных по возрастанию массивов целых чисел в один отсортированный массив за время  $O(n \lg(k))$ , где  $n$  — общее количество элементов во всех входных массивах. Для слияния массивов воспользуйтесь очередью с приоритетами размера  $k$ .

Формат входных данных программы должен быть такой: первая строка, считываемая со стандартного потока ввода, содержит количество  $k$  массивов, вторая строка содержит последовательность целых чисел  $n_1, n_2, \dots, n_k$ , разделённых пробелами и задающих размеры массивов, каждая из следующих  $k$  строк описывает соответствующий массив, то есть  $i$ -тая строка содержит  $n_i$  целых чисел, отсортированных по возрастанию и разделённых пробелами.

Программа должна выводить в стандартный поток вывода отсортированную по возрастанию последовательность целых чисел, полученную путём слияния массивов. Целые числа должны разделяться пробелами или символами перевода строки.

Например, рассмотрим следующие входные данные:

```
4
3 5 1 2
10 12 20
15 16 17 19 25
20
11 12
```

Для этих данных программа должна вывести

```
10 11 12 12 15 16 17 19 20 20 25
```

### 3.5 Моделирование работы вычислительного кластера (cluster.c)

Имеется вычислительный кластер, состоящий из  $N$  однопроцессорных узлов. На кластере нужно выполнить  $M$  задач. Каждая задача описывается

парой  $\langle t_1, t_2 \rangle$ , где  $t_1$  — время в микросекундах от включения кластера, начиная с которого задачу можно посылать на выполнение (до этого времени входные данные для задачи неготовы);  $t_2$  — прогнозируемое время выполнения задачи в микросекундах.

Для выполнения каждой задачи задействуется один узел кластера, то есть задачи невозможно распараллеливать. Кроме того, нельзя менять порядок выполнения задач: если данные для задачи  $A$  оказываются подготовлены раньше, чем данные для задачи  $B$ , то задача  $A$  не может быть запущена позже задачи  $B$ .

Необходимо составить программу `cluster.c`, вычисляющую минимальное время в микросекундах от начала работы кластера, когда все задачи будут выполнены. В программе нужно использовать очередь с приоритетами для хранения времен окончания задач, запущенных на кластере.

Формат входных данных программы должен быть такой: первая строка, считываемая со стандартного потока ввода, содержит количество  $N$  узлов кластера, вторая строка содержит число  $M$  задач, каждая из следующих  $M$  строк содержит пару целых чисел  $\langle t_1, t_2 \rangle$ , описывающих задачу. Пары чисел отсортированы в порядке возрастания  $t_1$ , и, более того, в каждой паре  $t_1$  уникально.

Программа должна вывести в стандартный поток вывода целое число, выражающее время в микросекундах, прошедшее от включения кластера до момента, когда все задачи будут выполнены.

### 3.6 Сортировка списка вставками (`listisort.c`)

Составьте программу `listisort.c`, выполняющую сортировку двунаправленного кольцевого списка целых чисел по возрастанию. В программе должен быть реализован алгоритм сортировки вставками.

Элементы списка должны быть представлены структурой

```
struct Elem {
    struct Elem *prev, *next;
    int v;
};
```

Алгоритм сортировки вставками, адаптированный для списков, должен выполнять не более  $n$  обменов элементов, где  $n$  — длина списка.

Программа должна считывать со стандартного потока ввода размер списка  $n$  и значения элементов списка. В стандартный поток вывода должны выводиться элементы отсортированного списка.

### 3.7 Сортировка списка пузырьком (listbsort.c)

Составьте программу listbsort.c, выполняющую сортировку слов в предложении в порядке возрастания их длин. Слова в предложении разделены одним или несколькими пробелами. Программа должна формировать однонаправленный список слов, а затем сортировать этот список пузырьком.

Функция bsort, реализующая алгоритм сортировки, должна быть объявлена как

```
struct Elem *bsort(struct Elem *list)
{
    ...
}
```

Структура Elem, указатели на которую фигурируют в объявлении функции bsort, должна представлять элемент однонаправленного списка, содержащего одно слово из предложения:

```
struct Elem {
    struct Elem *next;
    char *word;
};
```

Исходное предложение подаётся в стандартный поток ввода программы. Программа должна вывести в стандартный поток вывода отсортированную последовательность слов, разделённых пробелами.

### 3.8 Ранги элементов в списке с пропусками (ranklist.c)

Операция Rank :  $A \times K \rightarrow \mathbb{N}$  для ассоциативного массива вычисляет порядковый номер словарной пары с ключом  $k$  в отсортированной последовательности входящих в ассоциативный массив словарных пар. Пары нумеруются, начиная с нуля.

Модифицируйте представление и реализацию списка с пропусками, чтобы операция Rank для него работала в среднем за время  $O(\lg n)$ .

Составьте программу ranklist.c, демонстрирующую работоспособность реализованной операции.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму INSERT  $k$   $v$  (добавить в список с пропусками словарную пару, в которой ключ  $k$  — целое число, значение  $v$  — строка, составленная из латинских букв;  $-1000000000 < k < 1000000000$ ,  $\text{len}(v) < 10$ ), либо форму LOOKUP  $k$  (вывести строку, связанную с ключом  $k$ ), либо форму DELETE  $k$  (удалить строку, связанную с ключом  $k$ ), либо форму RANK  $k$  (вывести порядковый номер словарной пары с ключом  $k$ ).

Можно считать, что последовательность операций составлена правильно.

**Формат результата работы программы.** Для каждой операции LOOKUP вывести в стандартный поток вывода строку, связанную с ключом  $k$ . Для каждой операции RANK вывести в стандартный поток вывода порядковый номер словарной пары с ключом  $k$ .

**Указание.** Представление списка с пропусками нужно модифицировать следующим образом: каждый элемент списка должен включать в себя массив целых чисел `span` размера  $m$ , где  $m$  — количество уровней в списке. При этом  $i$ -тый элемент массива `span` должен содержать расстояние от данного элемента до следующего элемента на  $i$ -том уровне.

### 3.9 Ранги вершин бинарного дерева поиска (`ranktree.c`)

Операция `SearchByRank` :  $A \times \mathbb{N} \rightarrow P$  для ассоциативного массива возвращает словарную пару с заданным номером в отсортированной последовательности входящих в ассоциативный массив словарных пар.

Модифицируйте представление и реализацию бинарного дерева поиска, чтобы операция `SearchByRank` для него работала за время  $O(h)$ , где  $h$  — высота дерева.

Составьте программу `ranktree.c`, демонстрирующую работоспособность реализованной операции.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму `INSERT  $k$   $v$`  (добавить в дерево словарную пару, в которой ключ  $k$  — целое число, значение  $v$  — строка, составленная из латинских букв;  $-1000000000 < k < 1000000000$ ,  $len(v) < 10$ ), либо форму `LOOKUP  $k$`  (вывести строку, связанную с ключом  $k$ ), либо форму `DELETE  $k$`  (удалить строку, связанную с ключом  $k$ ), либо форму `SEARCH  $x$`  (вывести строку, связанную с ключом, имеющим порядковый номер  $x$ ).

Можно считать, что последовательность операций составлена правильно.

**Формат результата работы программы.** Для каждой операции LOOKUP вывести в стандартный поток вывода строку, связанную с ключом  $k$ . Для каждой операции SEARCH вывести в стандартный поток вывода строку, связанную с ключом, имеющим порядковый номер  $x$ .

**Указание.** Представление бинарного дерева поиска нужно модифицировать следующим образом: в каждую вершину нужно добавить поле `count`, содержащее размер поддерева с корнем в данной вершине. Размер поддерева — это количество вершин в нём.

### 3.10 Лексический анализ (lexavl.c)

Пусть *константа* — это непустая последовательность десятичных цифр.

Пусть *специальный знак* — это один из следующих символов: +, -, \*, /, (, ).

Пусть *идентификатор* — это непустая последовательность латинских букв и десятичных цифр, начинающаяся с буквы.

Пусть *лексема* — это либо константа, либо специальный знак, либо идентификатор.

Известно, что в некоторой строке записаны лексемы и пробелы. Лексемы не обязательно разделены пробелами за исключением случая, когда непосредственно после константы идёт идентификатор. Назовём такую строку **предложением**.

**Лексический анализ** предложения заключается в выделении из него последовательности записанных в нём лексем. При этом для каждой лексемы вычисляется пара  $\langle tag, value \rangle$ , где *tag* — тип лексемы (CONST для констант, SPEC для специальных знаков и IDENT для идентификаторов), а *value* — значение лексемы.

Значение лексемы — это неотрицательное целое число, смысл которого зависит от типа лексемы.

Константу мы будем считать десятичной записью её значения.

Значением специального знака пусть будет его порядковый номер в списке +, -, \*, /, (, ) (нумерация осуществляется, начиная с нуля).

Значение идентификатора определяется следующим образом: если выписать все идентификаторы в том порядке, в каком они входят в предложение, и оставить только первые вхождения каждого идентификатора, то значением идентификатора будет являться его порядковый номер в получившейся последовательности (нумерация осуществляется, начиная с нуля).

Например, если дано предложение

alpha + x1 (beta alpha) x1 y\$

то значением идентификатора *alpha* является число 0, значением идентификатора *x1* — число 1, значением *beta* — число 2, а значением *y* — число 3.

Составьте программу lexavl.c, выполняющую лексический анализ предложения.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит размер предложения *n*. Следующая строка содержит само предложение.

**Формат результата.** Для каждой лексемы, выделенной из предложения, программа должна выводить в стандартный поток вывода её тип и значение.

**Указание.** В процессе лексического анализа необходимо использовать АВЛ-дерево, хранящее отображение идентификаторов в их значения.

### 3.11 Разреженный массив (dispararray.c)

**Разреженный массив** — это массив большого размера, большинство элементов которого равны нулю. Хранение разреженного массива в памяти целиком нецелесообразно или даже вовсе невозможно из-за его большого размера, поэтому разумным решением является хранение только ненулевых элементов за счёт некоторого снижения скорости операций над массивом.

Разреженный целочисленный массив можно представить как ассоциативный массив, в котором и ключи, и значения являются целыми числами. При этом наличие в ассоциативном массиве словарной пары  $\langle k, v \rangle$  означает, что  $k$ -й элемент разреженного массива равен  $v$ . Если же в ассоциативном массиве нет пары с ключом  $k$ , то считается, что  $k$ -тый элемент разреженного массива равен нулю.

Будем считать, что в ассоциативном массиве, представляющем разреженный массив, вообще нет словарных пар со значением ноль. Это означает, что если  $k$ -му элементу разреженного массива, содержащему ненулевое значение, присваивается ноль, то словарная пара с ключом  $k$  вообще удаляется из ассоциативного массива.

Пусть  $A$  — множество разреженных целочисленных массивов. Определим основные операции над разреженным целочисленным массивом:

- $At : A \times \mathbb{N} \rightarrow \mathbb{Z}$  — возвращает  $k$ -тый элемент массива,
- $Assign : A \times \mathbb{N} \times \mathbb{Z} \rightarrow A$  — присваивает новое значение  $k$ -тому элементу массива.

Пусть разреженный целочисленный массив представлен в виде хеш-таблицы размера  $m$ . Реализуйте для него операции  $At$  и  $Assign$ . Составьте программу `dispararray.c`, демонстрирующую работоспособность реализованных операций.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 100000$ ), а вторая — размер хеш-таблицы  $m$ . Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму `ASSIGN  $i$   $v$`  (присвоить значение  $v$  элементу разреженного массива с индексом  $i$ ;  $0 \leq i < 1000000000$ ;  $-1000000000 < v < 1000000000$ ), либо форму `AT  $i$`  (вывести значение элемента разреженного массива с индексом  $i$ ).

**Формат результата работы программы.** Для каждой операции АТ вывести в стандартный поток вывода значение элемента разреженного массива с индексом  $i$ .

**Указание.** Пусть хеш-функция вычисляется как  $h(i) = i \bmod m$ .

### 3.12 Количество подпоследовательностей, на которых побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ даёт ноль (zeroxor.c)

Пусть дана последовательность из  $n$  целых чисел, где  $0 < n \leq 100000$ , и каждое целое число находится в диапазоне от  $-2^{31}$  до  $2^{31} - 1$ . Составьте программу `zeroxor.c`, вычисляющую, сколько в последовательности существует подпоследовательностей таких, что побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ их элементов равно 0.

Например, в последовательности  $a = \{0, 14, 14, 2, 2, 0\}$  таких подпоследовательностей 10 штук:

- $a[0 : 0] = \{0\}$ ,
- $a[0 : 2] = \{0, 14, 14\}$ ,
- $a[0 : 4] = \{0, 14, 14, 2, 2\}$ ,
- $a[0 : 5] = \{0, 14, 14, 2, 2, 0\}$ ,
- $a[1 : 2] = \{14, 14\}$ ,
- $a[1 : 4] = \{14, 14, 2, 2\}$ ,
- $a[1 : 5] = \{14, 14, 2, 2, 0\}$ ,
- $a[3 : 4] = \{2, 2\}$ ,
- $a[3 : 5] = \{2, 2, 0\}$ ,
- $a[5 : 5] = \{0\}$ .

Программа должна считывать из стандартного потока ввода число  $n$ , а затем  $n$  чисел, образующих последовательность. Программа должна вывести количество подпоследовательностей в стандартный поток вывода.

**Указание.** Пусть побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ подпоследовательностей  $[0 : i]$  и  $[0 : j]$  равно  $x$ , и не существует такого  $k$ , что  $i < k < j$  и побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ подпоследовательности  $[0 : k]$  равно  $x$ .

Пусть также мы знаем количество подпоследовательностей с правой границей  $i$ , побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ которых равно 0. Очевидно, что количество подпоследовательностей с правой границей  $j$ , побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ которых равно 0, на единицу больше.

### 3.13 Строки с общими префиксами (ptrie.c)

Реализуйте структуру данных, представляющую множество строк с операциями `Insert` (добавление строки в множество), `Delete` (удаление строки из множества) и `Prefix` (подсчёт количества строк множества, имеющих

указанных префикс). Операции `Insert` и `Delete` должны работать за время  $O(\text{len}(k))$ , где  $k$  — добавляемая или удаляемая строка, а операция `Prefix` — за время  $O(\text{len}(p))$ , где  $p$  — префикс.

Составьте программу `ptrie.c`, демонстрирующую работоспособность реализованных операций.

**Формат входных данных.** Первая строка, считываемая со стандартного потока ввода, содержит общее количество выполняемых операций  $n$  ( $0 < n \leq 10000$ ). Каждая из следующих  $n$  строк содержит описание операции.

Операция либо имеет форму `INSERT  $k$`  (добавить в множество строку  $k$ ,  $0 < \text{len}(k) < 100000$ ), либо форму `DELETE  $k$`  (удалить из множества имеющуюся в нём строку  $k$ ), либо форму `PREFIX  $p$`  (вычислить количество строк в множестве, имеющих префикс  $p$ ).

Отметим, что аргументы операций — это строки, составленные из маленьких латинских букв.

Кроме того, допустим вызов операции `INSERT` для строки, уже присутствующей в множестве.

**Формат результата работы программы.** Для каждой операции `PREFIX` вывести в стандартный поток вывода количество строк в множестве, имеющих указанный префикс.