Курс лекций по олимпиадной информатике (C)

Лекция 1. Арифметика и теория чисел

Михаил Густокашин, 2009

Конспекты лекций подготовлены для системы дистанционной подготовки, действующей на сайте **informatics.mccme.ru**.

При нахождении ошибок или опечаток просьба сообщать по адресу

gustokashin@gmail.com

Версия C от 16.11.2009

1 Организация ввода-вывода из файла

В большинстве олимпиад по информатике входные данные вводятся из текстового файла и выводиться должны так же в файл. Очень редко возникает необходимость считывать данные из файла дважды; организаторы стараются избегать таких задач. Мы рассмотрим метод, который позволит читать и писать данные из файлов так же, как с консоли — это позволит избежать путаницы и облегчить процесс отладки.

В лекциях мы будем рассматривать программы на языке **C**. В частности, для пишущих на **C**, рекомендуем использовать библиотеку `stdio.h` для реализации ввода-вывода вместо `iostream` и других библиотек для работы с потоками ввода-вывода. Этот выбор связан со значительно более высокой производительностью библиотеки `stdio.h`. Для пользующихся **Java** рекомендуется самостоятельно изучить функции для работы с потоками ввода-вывода, а для использующих `iostream` — изучить эту библиотеку или перейти на использование `stdio`.

Итак, перейдем к рассмотрению организации ввода-вывода. Допустим, перед нами стоит задача сложить два целых числа, не превосходящих по модулю 10000. Входные данные находятся в файле input.txt, вывод должен осуществляться в файл output.txt.

Рассмотрим решение этой задачи:

```
#include <stdio.h>

int main(void)
{
  int i, j;
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
  scanf("%d%d", &i, &j);
  printf("%d", i+j);
  return 0;
}
```

Первая функция `freopen` перенаправляет стандартный поток ввода (`stdin`) на файл (`input.txt`) для чтения (`r`. Полностью аналогично в следующей строке перенаправляется стандартный поток вывода.

Стоит заметить, что при использовании этого метода нет необходимости использовать `fclose` для закрытия файлов, т.к. стандартные потоки закрываются автоматически в конце работы программы.

Для отладки достаточно закомментировать строки, перенаправляющие ввод-вывод в файл, и тогда программа будет считывать и выводить данные на консоль. Обратите внимание, что следует не забывать убрать комментарий перед отправкой решения на проверку.

2 Интуитивное понятие сложности алгоритма

Оценкой эффективности реализации решения принято считать время работы программы и количество используемой памяти. Стандартные сообщения тестирующей системы о превышении максимальных лимитов обычно носят вид **TL** (Time Limit exceed, превышен лимит по времени) и **ML** (Memory Limit exceed, превышен лимит по памяти).

Эффективность программы обычно определяется, исходя из сочетания времени работы программы и используемого объема памяти, причем важность каждого из компонентов определяется в зависимости от задачи. В случае же олимпиадного программирования следует использовать еще один решающий компонент оценки эффективности — время, затраченное на написание программы. Таким образом, наиболее эффективным решением олимпиадной задачи по информатике можно считать решение, укладывающееся в **TL** и **ML**, и написанное за кратчайшее время.

Основная часть времени работы программы состоит из произведения количества итераций цикла (т.е. сколько раз были выполнены действия) на время выполнения команд в цикле. Время выполнения условно будем называть «константой» (не зависящей от входных данных), а количество итераций — «сложностью» алгоритма. Допустим, алгоритм линейный, т.е., например, на вход дается N чисел, и следует подсчитать их сумму. Тогда сложность алгоритма будет обозначаться как $O(N)$ (O-большое от N). Константа же для этой задачи может различаться в зависимости от конкретных условий. Например, для целых чисел константа будет в несколько раз меньше, чем для вещественных (т.к. целочисленные операции выполняются в несколько раз быстрее). Для квадратичного алгоритма (например, для сортировки пузырьком) сложность записывается как $O(N^2)$. Сложность может зависеть и от нескольких параметров, если они определяют количество циклов. Тогда сложность может записываться как $O(N^2+M^3)$. Если в алгоритме нет циклов, зависящих от входных данных, то говорят, что он «работает за константу», т.е. за $O(1)$.

Обычно за O-большое обозначают сложность в худшем случае. Некоторые алгоритмы работают очень по-разному в зависимости от входных данных, в этом случае мы будем отдельно указывать «сложность в среднем».

Очень часто в процессе подсчета сложности алгоритма используется функция $\log N$ («логарифм от N»). В программировании мы обычно будем использовать двоичный логарифм. Он определяется так: $2^{\log_2 N} = N$. В дальнейшем под записью $\log N$ мы будем понимать $\log_2 N$. Интересная особенность логарифма заключается в том, что он растет очень медленно при росте N. Так при $N = 8 : \log N = 3$, $N = 65536 : \log N = 16$,

$N = 4294967296 : \log N = 32$. Скорость роста логарифма намного меньше, чем даже у квадратного корня. Поэтому при больших значениях N алгоритм со сложностью $\log N$ и большой константой будет эффективнее линейного алгоритма с маленькой константой.

На этом закончим введение в теорию сложности алгоритмов.

3 Целочисленные типы данных и их использование

На данный момент на олимпиадах используются, в основном, 32-битные системы и компиляторы. Приведем таблицу, в которой указаны максимальные и минимальные значения для основных типов переменных:

Название	**Байт**	**Минимум**	**Максимум**
unsigned char	1	0	255
char	1	−128	127
short	2	−32768	32767
unsigned short	2	0	65535
int	4	−2147483648	2147483647
unsigned int	4	0	4294967295
int64 (long long)	8	-2^{63}	$2^{63}-1$

Следует заметить, что операции с `int64` выполняются несколько медленнее, чем с другими типами целочисленных данных.

При программировании следует стараться использовать тип `int` кроме ситуаций, когда использование меньшего типа позволит уложиться в **ML** или требуется заведомо больший тип данных.

Отметим некоторые особенности использование типа `int64` в языке **C**. Во-первых, в некоторых реализациях он обозначается как `__int64`, в некоторых других (в основном в UNIX-системах): `long long`. Кроме того, на различных платформах используется своя форматная строка для ввода-вывода `int64`. Универсальный вывод программируется с помощью макрокоманд и выглядит следующим образом:

```
#ifdef WIN32
  printf("%I64d", n);
#else
  printf("%lld", n);
#endif
```

В некоторых ситуациях не хватает даже типа `int64`. Иногда, если данные не намного превышают размер `int64`, можно воспользоваться типами переменных с плавающей точкой. Так `long double` может хранить без ошибок целое число длиной до 19 десятичных знаков. Однако следует помнить, что вещественные типы обрабатываются значительно медленнее, для них не определены операции взятия остатка и деления нацело. Несмотря на это, все же существуют ситуации, когда их использование оправдано.

Классический пример на тему выбора типа переменных выглядит так: даны два числа X и Y по модулю меньшие, чем 2^{31}, следует вывести их сумму. Участник смотрит на данные, выбирает тип `int` и получает ошибку, т.к. результат сложения может не уместиться в этот тип данных.

Если же не хватает и этих переменных, то единственный выход — использование длинной арифметики, т.е. арифметики на массивах, чему и будет посвящена следующая часть лекции.

4 Длинные числа и операции над ними

При рассмотрении этой темы нам потребуются знания, полученные в начальной школе. В частности, выполнение арифметических операций «в столбик».

Идея реализации длинных чисел заключается в использовании массивов, состоящих из чисел «коротких» (они будут играть роль цифр). Будем придерживаться соглашения, что числа прижаты к «правому краю», т.е. младшие разряды стоят в ячейках с большим номером.

Во-первых, определим максимальную длину длинного числа. В общем случае — это небанальная задача, и этот параметр определяется с помощью комбинаторных формул, но можно постараться подобрать заведомо удовлетворяющее нашим потребностям значение.

Удобнее всего определить один раз максимальное значение длины (количества элементов в массиве), чтобы можно было уменьшать его для более удобной отладки. Это делается с помощью следующей макроподстановки:

```
#define MAXLEN 1000
```

Мы определили максимальную длину массива в 1000 «цифр». Кроме того, удобно определить структуру для хранения числа:

```
typedef struct
{
  int val[MAXLEN+1];
  int st;
} vlong;
```

Поле `st` будет указывать на ту позицию, с которой начинается осмысленная часть числа (т.е. начинаются отличные от нуля цифры; в дальнейшем мы увидим, что разумное использование этого поля избавляет нас от огромного количества бесполезных операций). Макроподстановка `typedef` делается для того, чтобы не писать каждый раз слово `struct`. Программирующие на **C++** и **Java** могут создать класс и дальнейшие функции записывать в виде переопределенных операторов этого класса.

Вспомним, как осуществляется сложение в столбик. Мы начинаем идти с младших разрядов, складываем цифры, стоящие на данных позициях, прибавляем то, что было «в уме», новое значение «в уме» равно целой части от деления полученного результата на 10, в итоговое число на эту позицию — остаток от деления результата на 10. Так мы повторяем до тех пор, пока оба числа не кончатся (т.е. мы не дойдем до начала большего числа), в случае же, если «в уме» не ноль — мы дописываем эту цифру в начало. В реализации длинного сложения — все то же самое, только для экономии времени и места мы можем использовать, например, не систему счисления с основанием 10, а систему с основанием в 10000 (для сложения мы можем выбрать и большее значение). Для реализации этого метода и, опять же, более удобной отладки удобно задать константу с основанием системы счисления:

```
#define SYS 10000
```

Перейдем собственно к описанию функции сложения:

```
void add(vlong *op1, vlong *op2, vlong *res)
{
  vlong *mxop, *mnop;
  int i, flag=0, st;
  mxop = op1->st>op2->st?op1:op2;
  mnop = op1->st<=op2->st?op1:op2;
  st=mnop->st;
  for(i=MAXLEN; i >= mxop->st; i--) {
    res->val[i] = mxop->val[i] + mnop->val[i] + flag;
    flag = res->val[i] / SYS;
    res->val[i] %= SYS;
  }
  for(i=mxop->st-1; i >= mnop->st; i--) {
    res->val[i] = mnop->val[i] + flag;
    flag = res->val[i] / SYS;
    res->val[i] %= SYS;
  }
  if (flag) res->val[--st] = flag;
  res->st = st;
}
```

Сделаем некоторые пояснения к программе. Первым делом она выбирает большее и меньшее по количеству знаков число. Здесь опять же полная аналогия со сложением чисел в столбик: пока у нас оба числа еще не кончились, мы складываем по всем правилам, а затем переписываем начало большего числа, не забывая про значение «в уме».

Обычно большие числа инициализируются какими-либо малыми значениями, над которыми затем производятся операции, которые и приводят к росту этого числа. Т.е. короткое число надо записывать в `x.val[MAXLEN]`, а `x.st` устанавливать в `MAXLEN`.

Если `a`, `b` и `c` - переменные типа `vlong`, то вызов функции должен выглядеть как `add(&a, &b, &c)`. Это сделано для того, чтобы передавалось не само значение структуры, а указатель на нее (и не было необходимости в копировании).

Вывести значение длинного числа можно следующим образом:

```
printf("%d", c.val[c.st]);
for (i=c.st+1; i<=MAXLEN; i++)
  printf("%.4d", c.val[i]);
```

Обратите внимание на то, что первое число выводится без ведущих нулей, а каждое следующее должно выводится с ведущими нулями. При изменении базовой системы счисления (в нашем случае это 10000) надо заменять цифру в выражении `%.4d` на количество нулей в основании этой системы. Просто выводить элементы массива с помощью `%d` нельзя! Такую ошибку достаточно легко допустить и очень сложно найти.

Сложность полученного алгоритма получилась $O(N)$, где N — количество разрядов в большем числе. Если бы мы не использовали поля `st`, то нам каждый раз пришлось бы складывать все `MAXLEN` разрядов, а с учетом того, что очень часто используется не весь массив (в процессе вычисления числа растут, или мы не можем точно определить длину заранее), то сложность программирования оправдывается производительностью.

Длинное вычитание осуществляется аналогично. Используются те же идеи, что и в вычитании в столбик.

Теперь рассмотрим умножение длинного числа на короткое (коротким числом будем называть то, квадрат которого помещается в переменную элементарного типа, т.е. меньшее `SYS`). При этом функцию умножения на короткое будем писать сразу с прицелом на использование в умножении длинного на длинное.

```
void mul(vlong *op1, int sh, int offs, vlong *res)
{
  int i, flag=0;
  int st = op1->st;
  for(i=MAXLEN-offs+1; i <= MAXLEN; i++)
    res->val[i] = 0;
  for(i=MAXLEN; i >= st; i--) {
    res->val[i-offs] = op1->val[i] * sh + flag;
    flag = res->val[i-offs] / SYS;
    res->val[i-offs] %= SYS;
  }
  if (flag) res->val[--st-offs] = flag;
  res->st = st-offs;
}
```

Сложность алгоритма умножения длинного числа на короткое составляет $O(N)$, где N — количество разрядов в длинном числе. С использованием двух предыдущих функций умножение длинного числа на длинное реализуется очень просто.

```
void mul_long(vlong *op1, vlong *op2, vlong *res)
{
  int i;
  vlong temp;
  res->st = MAXLEN;
  res->val[MAXLEN] = 0;
  for (i=MAXLEN; i>= op1->st; i--) {
    mul(op2, op1->val[i], MAXLEN-i, &temp);
    add(res, &temp, res);
  }
}
```

Вызов функции умножения на короткое: `mul(&x, sh, 0, &res)`, где `x` — это длинное число, `sh` — короткое, `res` — результат. Вызов функции умножения длинного на длинное: `mul_long(&op1, &op2, &res)`, где `op1` и `op2` — длинные числа, а `res` — их произведение.

Инициализация значений и вывод осуществляется аналогично сложению.

Сложность умножения длинного числа на длинное получилась $O(N \times M)$, где N и M — количество разрядов в операндах. Существуют более быстрые методы умножения длинных чисел (например, метод Карацубы), однако в школьных олимпиадах их применение требуется крайне редко.

Длинное деление используется гораздо реже и реализуется аналогично делению в столбик. Длинные вещественные числа также оставим без внимания (интересующиеся могут найти способы хранения вещественных чисел в памяти компьютера и реализовать нечто подобное на массивах).

На этом закончим рассмотрение длинных чисел.

5 Делимость и делители

Отвлечемся от технических вещей и перейдем к программе курса математики начальной школы.

Простым числом называется натуральное число, большее 1, которое делится нацело только на себя и 1 (имеет два натуральных делителя). Составными числами, соответственно, называются все остальные натуральные числа, большие единицы.

Простые числа имеют множество полезных применений, а также и сами по себе нередко являются сутью задачи. Наиболее известное промышленное применение простых чисел — в шифровании **RSA** с открытым ключом.

В первую очередь нам необходимо уметь проверять, является ли число N простым. Т.е. нам необходимо узнать, существуют ли такие натуральные числа x, y ($1 < x, y < N$), что $x \times y = N$. Кроме того, условимся, что $x \leq y$, тогда можно сказать, что x меньше либо равен квадратному корню из числа N (если это условие не выполнено, то произведение будет заведомо больше N).

Таким образом, можно написать следующую функцию, которая будет довольно эффективно проверять число на простоту:

```
int isprime(int n)
{
  int i, j;
  if (2 == n) return 1;
  j = (int)sqrt((double)n)+1;
  for (i=2; i<=j; i++)
    if (!(n%i)) return 0;
  return 1;
}
```

Чтобы функция `sqrt` работала, необходимо подключить библиотеку `math.h`.

Пользуясь теми же соображениями, очень просто написать функцию, находящую все разложения числа на два множителя. Для этого достаточно убрать проверку на равенство N двум и в заменить `return 0` на обработку двух найденных делителей `i` и `n/i`.

Оба этих алгоритма имеют сложность $O(\sqrt{N})$.

6 НОД и НОК. Элементы теории остатков

Наибольшим общим делителем (НОД) двух натуральных чисел называется такое максимальное натуральное число, которое является делителем и первого и второго числа. Наименьшим общим кратным (НОК) двух натуральных чисел называется такое минимальное натуральное число, которое делится нацело на оба этих числа.

Аналогично вводятся понятия НОД и НОК многих чисел. На практике НОД и НОК многих чисел считаются с помощью последовательно попарного подсчета НОД и НОК (т.е. считается НОД уже обработанной части и очередного числа).

Задачи, в которых необходимо подсчитать НОД или НОК возникают довольно часто, особенно для НОД. Так, например, в подсчете количества точек с целыми координатами на отрезке используется НОД x и y координат отрезка.

В младшей школе изучается алгоритм Евклида, который позволяет найти НОД двух чисел. В нем используются следующие соотношения:

1) НОД$(a, 0) = a$

2) НОД(a, b) =НОД$(a\%b, b)$ =НОД$(a, b\%a)$

Запишем функцию подсчета НОД (GCD — Greatest Common Divisor):

```
int gcd(int ap, int bp)
{
  int c, a = ap, b = bp;
  while (b != 0) {
    c = a % b;
    a = b;
    b = c;
  }
  return a;
}
```

Этот алгоритм очень прост, его сложность в худшем случае, равна $O(\log N)$, где N — большее из чисел.

НОК можно искать исходя из соотношения НОД$(a, b)\times$НОК$(a, b) = a \times b$.

Существует еще один способ поиска НОД: бинарный алгоритм Евклида. Этот способ основывается на соотношениях НОД$(2 \times a, 2 \times b) = 2\times$НОД$(a, b)$, НОД$(2 \times a, 2 \times b + 1)$ =НОД$(a, 2 \times b + 1)$, НОД$(2 \times a + 1, 2 \times b + 1)$ =НОД$(2 \times (a - b), 2 \times b + 1)$. Хотя при ипользовании этих соотношений время написания программы и собственно поиска НОД несколько возрастет (хотя по прежнему сложность алгоритма будет составлять $O(\log n)$), этот способ намного удобнее при поиске НОД длинных чисел. Действительно, в обычном алгоритме Евклида необходима операция взятия остатка от деления длинных чисел, а в бинарном — намного более простые операции длинного вычитания и деления на 2.

В олимпиадных задачах довольно часто требуется найти что-либо «по модулю» какого-либо числа, т.е. подсчитать остаток от деления результата на заданное число. Теория чисел достаточно подробно изучается в школьном курсе математики и мы не будем углубляться в нее, взяв лишь несколько практически важных результатов.

В частности, нам важны следующие свойства остатков: $(a+b)\%n = (a\%n + b\%n)\%n$ и $(a \times b)\%n = (a\%n \times b\%n)\%n$. Эти свойства часто бывают полезными, т.к. обычно в задачах, где требуется подсчитать остатки, возникают довольно большие числа, и

уменьшение размерности операндов в промежуточных вычислениях намного облегчает задачу.

Также довольно часто требуется найти удовлетворяющий условию элемент какой-либо последовательности по модулю данного числа. В этом случае можно составить «таблицу умножения» по модулю n или другую таблицу с правилами перехода — это может значительно облегчить решение задачи.

7 Разложение числа на простые множители

Вернемся к простым числам. Любое число можно представить в виде произведения простых чисел, причем это представление будет единственно.

Обычно, такое представление выглядит в виде одномерного массива, содержащего простые числа, и еще одного одномерного массива для непосредственного представления числа, в каждой ячейке которого содержится степень соответственного просто числа.

Например, пусть у нас есть массив простых чисел $[2, 3, 5, 7, 11, 13]$, тогда массив с представлением числа 2600 будет выглядеть как $[3, 0, 2, 0, 0, 1]$ из которого можно получить $2^3 \times 5^2 \times 13^1 = 2600$.

Такое представление неоправданно генерировать для одного числа (если это не является необходимым условием для решения задачи). Однако, если чисел много, то приведение их в такое представление и обратно может быть весьма полезным.

Например, с помощью такого представления очень просто реализовать умножение. Рассмотрим наше число 2600 и число 11858, которое представляется массивом $[1, 0, 0, 2, 2, 0]$. Чтобы получить произведение этих чисел, достаточно сложить соответствующие элементы массивов. Результатом будет массив $[4, 0, 2, 2, 2, 1]$, т.е. $2^4 \times 5^2 \times 7^2 \times 11^2 \times 13^1 = 30830800$, что совпадает с результатом умножения.

После некоторых размышлений можно также реализовать деление с остатком, но в рамках лекции мы этого делать не будем, желающие могут сами придумать алгоритм.

Из этого представления также можно получить НОД и НОК этих чисел.

Для нахождения НОД надо выбирать минимум из степеней (это логично, т.к. число X^n кратно X^k, если $n \geq k$). Для подсчета НОК надо брать максимум из степеней. Этот же метод работает для подсчета НОД и НОК произвольного количества чисел.

Для наших чисел 2600 и 11858, НОД, подсчитанный таким образом, представим массивом $[1, 0, 0, 0, 0, 0]$, т.е. равен 2, а НОК — массивом $[3, 0, 2, 2, 2, 1]$ и равен 15415400. С помощью алгоритма Евклида несложно убедиться, что результат верен.

Кроме того, у разложения числа на простые множители существуют более специфичные назначения, которые встретятся по ходу решения практических туров.

Нахождение всех простых чисел до N занимает $O(N \times \sqrt{N})$ времени, подсчет степеней для одного числа занимает около $O(N)$ времени (если степени не слишком большие), и операции умножения, нахождения НОД и НОК занимают также $O(N)$ времени. Таким образом, суммарное время составляет около $O(N \times \sqrt{N} + N \times M)$, где N — максимальное из чисел, а M — количество этих чисел.

В некоторых случаях нам необходимо разложить только одно число на простые множители. В такой ситуации наиболее эффективным методом будет модификация функции проверки числа на простоту. Действительно, если каждый раз при нахождении делителя мы будем делить разлагаемое число на него до тех пор, пока оно делится (и

запоминать степень, с которой этот делитель входит в разложение числа), то мы получим все простые делители кроме, возможно, одного. Действительно, поиск делителей необходимо осуществлять до квадратного корня из числа, а в случае, если после деления осталась не единица — этот остаток также является простым делителем, причем степень его вхождения всегда равна 1. Например, 26 представляется как $2^1 \times 13^1$, где 13 — единственный делитель, больший квадратного корня.

8 Быстрое возведение в степень

Довольно часто возникает задача быстрого возведения числа или другого объекта, для которого определена операция умножения, в какую-либо степень. Наивный алгоритм, когда мы просто нужное число раз умножаем число на само себя, имеет сложность $O(N)$, где N — показатель степени.

При возведении в степень число растет очень быстро (а значит, наверняка требуются операции с длинными числами). Поэтому научиться возводить число в степень быстрее, чем за $O(N)$ — достаточно важная задача.

Рассматривая степени некоторых чисел, можно догадаться о методе, которым следует пользоваться. Например, для возведения 3 в 4-ю степень нам нужно проделать 3 операции умножения. Однако, если изменить порядок действий на такой: $(3^2)^2$, то потребуется всего два умножения. Следует помнить, что при возведении числа в степени еще в какую-либо степень показатели степеней перемножаются. Именно на сокращении четных степеней основывается идея быстрого возведения в степень.

Приведем текст функции, которая возводит число a в степень n:

```
int pow(int a, int n)
{
  int b, c, k;
  k = n;
  b = 1;
  c = a;
  while (k)
    if (!(k%2)) {
      k /= 2;
      c *= c;
    } else {
      k--;
      b *= c;
    }
  return b;
}
```

Каждый раз, когда степень четная, мы возводим вспомогательную переменную в квадрат, а показатель степени делим на 2. Если число нечетное, то умножаем текущее вспомогательное число на результат и уменьшаем показатель степени на 1. Таким образом хотя бы один раз из двух у нас произойдет уменьшение показателя степени вдвое и, исходя из этого, мы получим сложность алгоритма $O(\log N)$.

Достижение такой сложности очень полезно, особенно при использовании медленных операций с длинными числами. Таким образом можно выбрать «универсальный» рецепт: использовать быстрое возведение степень везде, где показатель может превышать 100. С учетом того, что функция достаточно простая, можно использовать быстрое возведение в степень во всех случаях.

В англоязычной литературе алгоритм быстрого возведения в степень обзывают забавным словосочетанием “Russian peasant algorithm”, что переводится как «алгоритм русского крестьянина».

9 Матрицы и операции над ними

В рамках нашей сегодняшней лекции мы будем рассматривать только квадратные матрицы, состоящие из чисел. Матрица является, по сути, двумерным массивом, чтобы наглядно представить, что это такое, можно открыть электронную таблицу. Каждый элемент матрицы определяется именем этой матрицы и двумя числами — индексами массива.

Двумерные массивы очень часто используются в программировании для хранения данных, но сегодня мы остановимся на алгебраических свойствах матриц.

Матрица является математическим объектом, и для нее, как и для чисел, определены некоторые операции, в частности, сложение и вычитание матриц. Допустим, что A, B и C — квадратные матрицы одинакового размера. Тогда матрица C, равная сумме матриц A и B определяется поэлементно, как $C[i][j] = A[i][j] + B[i][j]$. Точно так же определяется и разность матриц. Умножение матрицы на число — это просто умножение каждого элемента матрицы на это число.

По-другому происходит умножение матриц. Допустим, $C = A \times B$, тогда $C[i][j] = \sum_{k=0}^{n-1}(A[i][k] \times B[k][j])$, k изменяется от 0 до $n-1$ (n — размер матрицы, нумерация начинается с нуля). Знак $\sum$ означает сумму, т.е. $\sum_{k=0}^{4} A[k]$ — это то же самое, что $A[0] + A[1] + A[2] + A[3] + A[4]$. Обратите внимание, что $A \times B$, вообще говоря, не равно $B \times A$.

Перед написанием функции умножения матриц определим несколько макроподстановок.

```
#define MAXN 100
```

Эта макроподстановка будет определять размер матрицы. Удобно завести ее в виде константы, чтобы иметь возможность отладить программу (на отладчике очень тяжело просматривать большие двумерные массивы).

```
#define matr(a) int a[MAXN][MAXN]
```

Это можно использовать просто для удобства написания.

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

Во многих задачах используются циклы по какой-либо переменной от нуля до другой переменной. Очень удобно оформлять их в таком виде. Т.е. команда `For(i, j)` перед компиляцией заменится на `for(i=0; i<j; i++)`. Если привыкнуть к такому стилю, то он поможет экономить время, которое очень ценно на олимпиаде. Теперь перейдем непосредственно к реализации функции умножения матриц:

```
void mulmatr(matr(a), matr(b), matr(c))
{
  int i, j, k;
  For(i, MAXN)
    For(j, MAXN) {
      c[i][j] = 0;
      For(k, MAXN)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Рассмотрим еще одно понятие, которое позволит нам применить хитрость, ускоряющее работу программы в несколько раз.

Транспонированной матрицей называется такая матрица, у которой столбцы становятся строками, а строки — столбцами. Т.е. $A^T[i][j] = A[j][i]$ (A^T - обозначение транспонированной матрицы).

В случае умножения матриц мы берем строку одной матрицы и столбец другой и осуществляем к ним последовательный доступ. В памяти компьютера многомерные массивы разворачиваются в одномерные, и доступ к строке идет довольно быстро, за счет оптимизаций компилятора при подсчете адресных выражений и кэширования (все данные лежат рядом и автоматически попадают в кэш). В случае же со столбцом адресное выражение (положение элемента массива в физической памяти компьютера) приходится вычислять заново и данные не попадают в кэш.

В нашей функции мы обращаемся к одному и тому же столбцу `MAXN` раз, за счет чего производительность резко падает. Если предварительно применить транспонирование матрицы, а потом вернуть ее обратно, то нам удастся ускорить программу в $3 - 5$ раз без использования дополнительной памяти. Измененная функция будет выглядеть так:

```
void mulmatr(matr(a), matr(b), matr(c))
{
  int i, j, k;
  For(i, MAXN)
    For(j, i) {
      k = b[i][j];
      b[i][j] = b[j][i];
      b[j][i] = k;
    }
  For(i, MAXN)
    For(j, MAXN) {
      c[i][j] = 0;
      For(k, MAXN)
        c[i][j] += a[i][k] * b[j][k];
    }
  For(i, MAXN)
    For(j, i) {
      k = b[i][j];
      b[i][j] = b[j][i];
```

```
      b[j][i] = k;
    }
}
```

В конце работы функции мы возвращаем матрицу B в ее исходное состояние. Теперь у нас есть только умножение строки на строку, которое происходит заметно быстрее. Можно было использовать и другие методы хранения столбца, например, записывая его в одномерный массив.

Кроме того, существует также операция умножения матрицы на вектор (одномерный массив). Пусть A — матрица, B - вектор из чисел, тогда в результате умножения мы получим вектор C, который будет определяться исходя из формулы $C[i] = B[i] \times \sum_{k=0}^{n-1} A[i][k]$.