Математичка, ты довольна?
Часть 2.

Сапожников Денис

Contents

1 Алгоритм Евклида **2**
1.1 Определения . . . 2
1.2 Медленный алгоритм . . . 2
1.3 Быстрый алгоритм . . . 3
1.4 Время работы . . . 3
1.5 Упражнения . . . 3

2 Решето Эратосфера **4**
2.1 Задача . . . 4
2.2 Самый простой подход за $O(n\sqrt{n})$ . . . 4
2.3 Решение за $O(n \log n)$ . . . 4
2.4 Решение за $O(n \log \log n)$ . . . 4
2.5 Решение за $O(n)$* . . . 5

3 Диофантовы уравнения и расширенный алгоритм Евклида **6**
3.1 Одно решение $ax + by = gcd(a, b)$ . . . 6
3.2 Одно решение $ax + by = c$ . . . 6
3.3 Все решения $ax + by = c$ . . . 7
3.4 Возвращаемся к обратным элементам по модулю . . . 7

4 Китайская теорема об остатках (КТО)* **8**

5 Комбинаторика **9**
5.1 Комбинаторные объекты и подсчёт их количества . . . 9
5.2 Сочетания . . . 10
5.3 Задачи . . . 11
5.4 Генерация комбинаторных объектов . . . 11
5.4.1 Перестановки . . . 11
5.4.2 Сочетания . . . 12
5.4.3 Разбиения . . . 13
5.5 Литература . . . 13

1 Алгоритм Евклида

1.1 Определения

Определение (НОД)**.** Наибольшим общим делителем (НОДом) двух чисел a и b называется такое число g, что $\begin{cases} a \vdots g \\ b \vdots g \\ g \ - \ \textit{максимальное из возможных} \end{cases}$

Обозначения: НОД(a, b), или просто (a, b).

Лемма 1. *Пусть $(a, b) = g$. Пусть d – какой-то (не обязательно наибольший) общий делитель a и b. Тогда $g \vdots d$.*

Proof. Через разложение на простые множители по ОТА. □

Определение (НОК)**.** Наименьшим общим кратным (НОКом) двух чисел a и b называется такое число l, что $\begin{cases} l \vdots a \\ l \vdots b \\ l \ - \ \textit{минимальное из возможных} \end{cases}$

Обозначения: НОК(a, b), или просто $[a, b]$.

Лемма 2. $(a, b) \cdot [a, b] = a \cdot b$

Proof. Через разложение на простые множители по ОТА. □

Данная лемма позволяет находит НОК по НОДу, таким образом надо научиться искать лишь НОД.

1.2 Медленный алгоритм

Лемма 3. *Пусть $a \geq b$, тогда $(a, b) = (a - b, b)$.*

Proof. Пусть $(a, b) = g_1, (a - b, b) = g_2$.

Докажем, что $g_2 \vdots g_1$. Если $(a, b) = g_1$, то по определению НОДа: $a \vdots g_1, b \vdots g_1$. Значит и $a - b \vdots g_1$. То есть получили, что $a - b$ и $b \vdots g_1$. По лемме 1 $g_2 \vdots g_1$.

Теперь докажем, что $g_1 \vdots g_2$. Если $(a - b, b) = g_2$, то b и $a - b \vdots g_2$. Значит и $a \vdots g_2$. Получили: a *и* $b \vdots g_2$. Значит по лемме 1 $g_1 \vdots g_2$.

Итого: $g_1 \vdots g_2$ *и* $g_2 \vdots g_1$. Значит, очевидно, $g_1 = g_2$ что и т.д. □

Лемма 3 позволяет легко находить НОД двух чисел без разложения на простые множители.

```
int gcd(int a, int b) {
  if (a == 0 || b == 0)
    return a + b;
  if (a >= b)
    return gcd(a - b, b);
  else
    return gcd(a, b - a);
}
```

По сути, мы уже доказали, что алгоритм корректный, но, увы, он долго работает, например, на тесте $(10^9, 1)$. На данном тесте будет выполняться 10^9 преобразований $(a, 1) = (a - 1, 1) = \ldots (1, 1) = (0, 1) = 1$.

1.3 Быстрый алгоритм

Идея по ускорению: заметим, что из $(a, b) = (a - b, b)$ следует, что $(a, b) = (a\%b, b)$, так как операция взятия по модулю эквивалента большому количеству вычитаний b.

Теперь алгоритм имеет вид

```
int gcd(int a, int b) {
  if (a == 0 || b == 0)
    return a + b;
  return gcd(b, a % b);
}
```

Более того, начиная с C++17 этот алгоритм есть в стандартной бибилоитеке numeric и называется gcd.

1.4 Время работы

Сейчас будет трюк. Следите за руками.

Пусть $a \geq b$. Тогда $0 \leq a\%b < b$. Что эквивалентно тому, что остатки от деления a на b лежат в интервале $\left[-\frac{b}{2}; \frac{b}{2}\right]$. Тогда после каждой итерации НОДа модулю одного из чисел уменьшается хотя бы в 2 раза.

Таким образом время работы алгоритма будет $O(\log n)$.

1.5 Упражнения

Задача 4. $(a_1, a_2, \ldots, a_n) = (a_1, (a_2, a_3, \ldots a_n))$

Задача 5. Докажите, что простых чисел бесконечно много.

Решение. Возможно, стоит посмотреть на числа вида $n! - 1$.

Задача 6. Пусть $x = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}$. Найдите количество делителей числа.

Решение. $\alpha(x) = (\alpha_1 + 1)(\alpha_2 + 1)\ldots(\alpha_k + 1)$

Задача 7. Докажите, что число $\underbrace{111\ldots1}_{n}$ для какого-то n делится на 123456789.

Решение. Посмотрим на остатки от деления первых 123456789+1 чисел вида $\underbrace{111\ldots1}_{n}$. Различных остатков не больше 123456789. Значит по принципу Дирихле найдутся два одинаковых остатка, пусть это будут $\underbrace{111\ldots1}_{k}$ и $\underbrace{111\ldots1}_{m}$, где $k > m$.

Посмотрим на число $\underbrace{111\ldots1}_{k} - \underbrace{111\ldots1}_{m}$. Оно делится на 123456789 и имеет вид $\underbrace{111\ldots1}_{k-m}\underbrace{000\ldots0}_{m}$. Поделим его на 10^m, результат будет иметь вид $\underbrace{111\ldots1}_{k-m}$ и всё ещё делиться на 123456789, так как 10 и 123456789 были взаимнопросты.

Задача 8. Научитесь вычислять $\lceil \frac{a}{b} \rceil$ без if.

Решение.

```
(a + b - 1) / b;
```

2 Диофантовы уравнения и расширенный алгоритм Евклида

2.1 Одно решение $ax + by = gcd(a, b)$

Задача. Найти одно решение уравнения $ax + by = g$, где g — это НОД(a, b).

Пусть (x_0, y_0) – это решение уравнения $(b\%a)x + ay = g$. Научимся находить решение исходного уравнения.

По определению остатка от деления: $b \bmod a = b - \lfloor \frac{b}{a} \rfloor a$

Значит: $g = (b \bmod a)x_1 + ay_1 = (b - \lfloor \frac{b}{a} \rfloor a)x_1 + ay_1 = bx_1 + a(y_1 - \lfloor \frac{b}{a} \rfloor x_1)$

В итоге формулы для пересчёта следующие:

$$\begin{cases} x_0 = y_1 - \lfloor \frac{b}{a} \rfloor x_1 \\ y_0 = x_1 \end{cases}$$

```
int gcd(int a, int b, int &x, int &y) {
  if (a == 0) {
    x = 0, y = 1;
    return a + b;
  }
  int x1, y1;
  int g = gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return g;
}
```

2.2 Одно решение $ax + by = c$

А теперь давайте решим уравнение $ax + by = c$ для произвольного c.

Если $c \vdots gcd(a, b)$, то всё просто: знаем решение уравнения

$$ax_0 + by_0 = g$$

Домножим обе части на $\frac{c}{g}$

$$a\left(\frac{c}{g}x_0\right) + b\left(\frac{c}{g}y_0\right) = c$$

То есть

$$\begin{cases} x = \frac{c}{g}x_0 \\ y = \frac{c}{g}y_0 \end{cases}$$

Если же $c \not\vdots gcd(a, b)$, то решений, очевидно, нет, так как

2.3 Все решения $ax + by = c$

Лемма. *Пусть (x_0, y_0) — это решение уравнения $ax + by = c$. Тогда все решения имею вид:*

$$\begin{cases} x = x_0 + \dfrac{b}{g}t \\ y = y_0 - \dfrac{a}{g}t \end{cases}, t \in \mathbb{Z}$$

Proof. Рассмотрим ещё одно решение (x_1, y_1) уравнения $ax + by = c$ и приравняем правые части:

$$\begin{aligned} ax_0 + by_0 &= ax_1 + by_1 \\ a(x_0 - x_1) &= b(y_1 - y_0) \\ \left(\frac{a}{g}\right)(x_0 - x_1) &= \left(\frac{b}{g}\right)(y_1 - y_0) \\ a'(x_0 - x_1) &= b'(y_1 - y_0) \end{aligned}$$

Левая часть делится на a', значит и правая часть тоже должна делиться на a'. Так как мы сократили обе части на (a, b), то $(a', b') = 1$, значит $(y_1 - y_0) \vdots a'$. То есть $y_1 = y_0 + a' \cdot t = y_0 - \dfrac{a}{g} \cdot t$ для $t \in \mathbb{Z}$. В пару к такому y_1 подходит единственное решение $x_1 = x_0 + \dfrac{b}{g} \cdot t$. □

2.4 Возвращаемся к обратным элементам по модулю

$$\begin{aligned} ab &\equiv 1 \pmod{n} \\ ab + kn &= 1 \end{aligned}$$

А это просто диофантово уравнение, решив которое мы найдем искомое a.

Итоговая сложность алгоритма $O(\log n)$

При этом мы решили для всех случаев, когда $\gcd(a, n) = 1$, а для других доказали, что решений нет.

3 Решето Эратосфера

3.1 Задача

Хотим найти все простые числа от 2 до n.

3.2 Самый простой подход за $O(n\sqrt{n})$

Заметим, что все имеет смысл перебирать делители лишь до $\sqrt{x}$, так как если есть делитель больше корня - d, то будет и делитель меньше корня: $\frac{x}{d}$. Поэтому код будет такой:

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i <= n; i++)
3   for (int j = 2; j * j <= i; j++)
4     if (i % j == 0)
5       is_prime[i] = false;
```

3.3 Решение за $O(n \log n)$

Пусть если число простое, то переберём все числа, которые делятся на него, то есть $2p, 3p, 4p, \ldots$ и пометим их как составные. Ну и действительно так мы пометим все числа составными, когда в качестве p мы возьмем любой простой делитель составного числа.

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i <= n; i++)
3   if (is_prime[i])
4     for (int j = 2 * i; j <= n; j += i)
5       is_prime[j] = false;
```

Время работы. Для каждого простого числа p внутренний фор сделает $\frac{n}{p}$ итераций. Значит всего алгоритм сделает не больше чем $\sum_{i=1}^{n} \frac{n}{i} = O(n \log n)$ [1], что и т.д.

3.4 Решение за $O(n \log \log n)$

Заметим факт, что нам необходимо помечать все простые делители, начиная не с $2i$, а с i^2 по идее из пункта 1. Таким образом нужно лишь заменить строку 4 и поменять условия для строки 2, чтобы не возникало переполнений:

```
1 vector<bool> is_prime(n, true);
2 for (int i = 2; i * i <= n; i++)
3   if (is_prime[i])
4     for (int j = i * i; j <= n; j += i)
5       is_prime[j] = false;
```

Почему это работает за $O(n \log \log n)$? Для доказательства используется факт, что простых чисел от 1 до n порядка $\frac{n}{\ln n}$ и они распределены примерно равномерно, а далее считается интеграл, более подробно можно прочитать на e-maxx.

[1] В целом, это известный факт, но доказывать мы его, конечно же, не будем

3.5 Решение за $O(n)$*

Вместо того, чтобы считать только пометку, простое ли было число, но и минимальное простое, на которое оно делится - p_i. Ещё будем поддерживать массив простых чисел от 2 до текущего i.

Пусть у нас есть текущее i. Обновим массив p с помощью i: пройдём по всем числам вида $x_j = p_j \cdot i$ (пока $p_j \leq p_i$) и отметим для них минимальное простое как p_j.

Итого код получается очень простым и лаконичным.

```
vector<int> pr;
vector<int> p(n, 0);
for (int i = 2; i < n; i++) {
  if (p[i] == 0) {
    p[i] = i;
    pr.push_back(i);
  }
  for (int j = 0; j < pr.size() && pr[j] <= pr[i] && i * pr[j] < n; ++j)
    pr[i * pr[j]] = pr[j];
}
```

Рассмотрим произвольное число x. Из того, что оно единственным образом представляется в виде $x = p(x) \cdot y$, где $p(x)$ - минимальное простое число, на которое делится x следует, что алгоритм посетит каждое число ровно один раз. Значит время работы $O(n)$

4 Китайская теорема об остатках (КТО)*

Пусть у нас есть система сравнений:

$$\begin{cases} a \equiv x_1 (\mathrm{mod} p_1) \\ a \equiv x_2 (\mathrm{mod} p_2) \\ \dots \\ a \equiv x_n (\mathrm{mod} p_n) \end{cases}$$

И ещё все модули попарно взаимнопросты, то есть $\forall i, j : gcd(p_i, p_j) = 1$
Утверждения про такую систему:

- Каждому числу из отрезка $[0; p_1 p_2 \dots p_n - 1]$ соответствует единственный набор переменных $(x_1, x_2, \dots, x_n)$, удовлетворяющий системе уравнений.

  Будем обозначать это как $a \Leftrightarrow (x_1, x_2, \dots, x_n)$

- Любой из операций $+, -, *$ соответствует ровно такая же над набором чисел $\{x_i\}$, то есть, если $a \Leftrightarrow (x_1, x_2, \cdots x_n)$ и $b \Leftrightarrow (y_1, y_2, \cdots y_n)$, тогда

  $a + b \Leftrightarrow (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n)$

  $ab \Leftrightarrow (x_1 y_1, x_2 y_2, \dots, x_n y_n)$

Понятно, как, зная число, получить набор $\{x_i\}$. Но как это сделать в обратную сторону?
Обозначим за $r_{i,j}$ обратный элемент к p_i по модулю p_j
Будем находить наше число a в таком виде:

$$a = y_1 + y_2 p_1 + y_3 p_1 p_2 + \dots + y_n p_1 \dots p_n$$

Тогда из 1-го уравнения знаем, что

$$y_1 = x_1$$

Из второго:

$$x_2 = y_1 + y_2 p_1$$
$$x_2 - y_1 = y_2 p_1$$
$$(a_2 - x_1) r_{1,2} = x_2$$

Ровно так же можно посчитать любое x_k алгоритм, очень похожим на схему Горнера за $O(k)$, значит восстановление всех x_i и искомого a - $O(n^2)$.

```
1  int a = 0, curp = 1;
2  for (int i = 0; i < n; ++i) {
3    y[i] = x[i];
4    for (int j = 0; j < i; ++j) {
5      y[i] = r[j][i] * (y[i] - y[j]);
6
7      y[i] = y[i] % p[i];
8      if (y[i] < 0)
9        y[i] += p[i];
10   }
11
12   a += curp * y[i];
13   curp *= p[i];
14 }
```

Но проблема в том, что встроенные типы очень быстро переполнятся, поэтому обычно придётся написать длинную арифметику.

5 Комбинаторика

Комбинаторика – это раздел математики, который занимается подсчётом количества способов что-то выбрать.

Задача 9. Есть коробка шаров, в которой 5 красных и 3 белых шара. Сколько способов выбрать из коробки 2 белых и 1 красный шар?

Решение довольно простое: нам нужно выбрать *независимо* 1 красный шар и 2 белых. Один красный из 5 шаров можно выбрать 5 способами, два белых из трёх – тремя. Итого получается $5 \cdot 3 = 15$ способов.

5.1 Комбинаторные объекты и подсчёт их количества

Размещением из n элементов по k с повторением называется всякий набор из k элементов n-элементного множества. Легко понять, что таких наборов всего

$$\bar{A}_n^k = n^k$$

Пример 10. Сколько существует чисел в семеричной системе счисления длины не больше 5. Ответ: 7^5.

Перестановкой из n элементов (например чисел $1, 2, \ldots, n$) называется всякий упорядоченный набор из этих элементов. Тут тоже все достаточно просто: на первое место мы можем поставить любой из n элементов. На второе *независимо* любой из $n-1$ оставшихся и т.д. То есть получим $n(n-1) \times \ldots \times 1 = n!$ способов.

Размещением из n элементов по k без повторений (далее это будет по умолчанию, если не оговорено иное) называется **упорядоченный** набор из k различных элементов некоторого n-элементного множества. Как посчитать A_n^k — количество размещений из n по k? Выпишем все перестановки лексикографическом, то есть в «алфавитном» порядке:

$$\begin{array}{cccccc}
1 & \ldots & n-2 & n-1 & n \\
1 & \ldots & n-2 & n & n-1 \\
1 & \ldots & n-1 & n-2 & n \\
1 & \ldots & n-1 & n & n-2 \\
1 & \ldots & n & n-2 & n-1 \\
\vdots & & & & \vdots \\
2 & \ldots & n-2 & n-1 & n \\
\vdots & & & & \vdots \\
n & \ldots & 3 & 2 & 1
\end{array}$$

Мы хотим посчитать, а сколько есть различных префиксов длины k? Заметим, что у первых $(n-k)!$ строк одинаковые префиксы длины k (так как происходят изменения только в последних $n-k$ элементах). Аналогично для следующих $(n-k)!$ строк. Получается, что всего $n!$ строк и уникальных среди них можно вычислить по формуле:

$$A_n^k = \frac{n!}{(n-k)!}$$

Пример 11. Есть 10 различных домиков и 7 одинаковых котят. Сколько существует способов расселения котят по домикам? Ответ: A_{10}^7.

Сочетанием C_n^k из n по k называется набор k элементов, выбранных из данных n элементов. Наборы, отличающиеся только порядком следования элементов (но не составом), считаются одинаковыми, этим сочетания отличаются от размещений.

По аналогии с предыдущим пунктом мы можем выписать все перестановки, но теперь нам нужно ещё, чтобы первые k элементов тоже были различны. Легко убедиться, что получается следующая формула:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Пример 12. Теперь мы можем легко решить задачу 1 уже в терминах сочетаний (и обобщить): нам нужно выбрать 2 элемента из 3 и 1 из 5 независимо друг от друга, получим $C_3^2 \cdot C_5^1 = 15$.

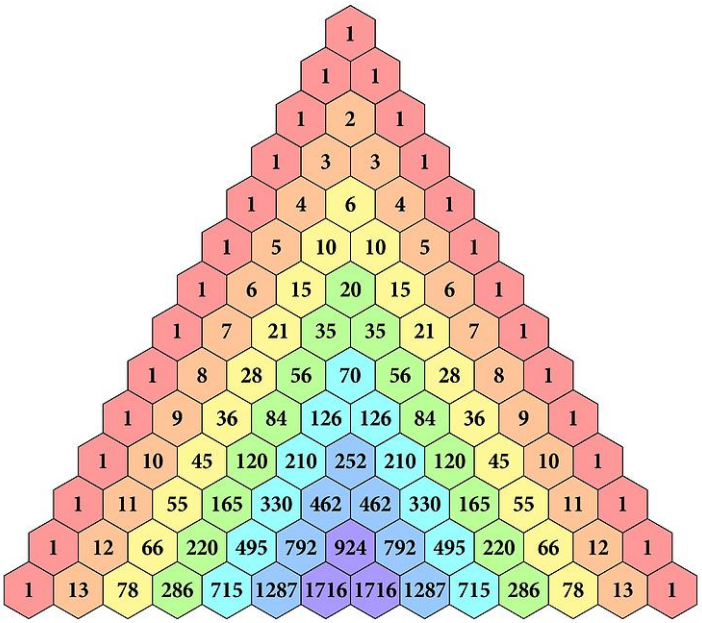
Сочетанием с повторением из n различных типов. Сколькими способами можно сделать из них комбинацию из k элементов, если не принимать во внимание порядок элементов внутри комбинации, а элементы одного типа могут повторяться.

Тут уже придётся применить комбинаторную идею: сделаем множество из n шаров и $k-1$ перегородки. Мы можем расставить из C_{n+k-1}^k способами, при этом на выходе будем получать k множеств и все способы будут различны. Таким образом, получаем формулу

$$\bar{C}_n^k = C_{n+k-1}^k$$

5.2 Сочетания

Сочетания чаще всего встречаются в задачах, потому на них стоит остановиться подольше.
Прямой проверкой можно показать формулу:
$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$
На основании этой формулы можно составить так называемый треугольник Паскаля:



k-й элемент в n-й строке — это C_n^k, а сам элемент равен сумме двух соседних элементов, стоящих строкой выше.

Но более того, давайте рассмотрим сумму $(a+b)^n$ — это называется бином Ньютона. Мы можем раскрыть скобки и получим сумму $(a+b)^n = (a+b)(a+b)\dots(a+b) = \sum_{k=0}^{n} C_n^k a^k b^{n-k}$.

Ещё интересный факт: сумма элементов в строке n равна 2^n:

$$C_n^0 + C_n^1 + \dots + C_n^n = 2^n$$

Это легко доказать, используя комбинаторное рассуждение. Мы считаем количество способов выбрать из n элементов $0, 1, 2, \dots, n$, то есть все возможные подмножества, получая таким образом все возможные варианты выбрать любле подмножество из n-элементного множества, а всего подмножеств 2^n.

Ещё пара полезных формул:
$C_n^k = C_n^{n-k}$ – прямая проверка.
$C_n^k + C_{n-1}^k + \dots + C_k^k = C_{n+1}^{k+1}$ – подумайте над комбинаторным смыслом.

5.3 Задачи

1. Сколькими способами можно расставить n единиц и m нулей так, чтобы никакие 2 единицы не стояли рядом?

   Выпишем m нулей в ряд. На любое из мест между нулями, а так же по краям можно поставить единицу, таким образом получили C_{m+1}^n способов.

2. На окружности отмечены n точек. Сколько существует многоугольников, вершинами которых является подмножество отмеченных точек? Сколько выпуклых из них?

   Каждый m-угольник определяется выбором m точек из n, взятых в определённом порядке, при чём циклическая перестановка и изменения обхода не меняет многоугольник, потому различных m-угольников $\frac{1}{2m} A_n^m$, а всего их $\sum_{m=3}^{n} \frac{1}{2m} A_n^m$. Число выпуклых многоугольников равно $\sum_{m=3}^{n} \frac{1}{2m} C_n^m$.

5.4 Генерация комбинаторных объектов

Мы поговорили о комбинаторных объектах, посчитали их, даже задачи порешали, осталось научиться их генерировать.

5.4.1 Перестановки

Генерировать все перестановки с ходу кажется сложным. Вместо этого научимся по перестановке генерировать следующую перестановку. В C++ уже есть встроенная функция `std::next_permutation(itbegin, itend)`, лежащая в `<algorithm>`, которая возвращает `false`, если следующей перестановки нет, а иначе получает следующую перестановку.

Чтобы понять, как получается следующая перестановка, напишем какой-нибудь пример: $(1, 5, 3, 6, 4, 2)$ и $(1, 5, 4, 2, 3, 6)$. На самом деле мы уже видим, как работает функция: мы берём первый элемент после возрастающей с конца последовательности, находим первый элемент, который меньше него в последовательности, свопаем и ревёрсим.

```
1 bool next_permutation(vector<int> &p) {
2   int n = p.size();
3   int pos = n - 2;
4   for (; pos >= 0; pos--)
```

```
    if (a[pos] < a[pos + 1])
      break;
  if (i == -1)
    return false;
  for (int i = n - 1; i >= 0; i--)
    if (a[pos] > a[i]) {
      swap(a[pos], a[i]);
      break;
    }
  reverse(p.begin() + pos, p.end());
  return true;
}
```

5.4.2 Сочетания

У нас была хорошая формула $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$, которая подсказывает, как можно генерировать все сочетания. Она говорит, что в одном случае давайте возьмём элемент n и тогда решим задачу меньше, а в другом случае – не возьмём и опять перейдём к задаче меньше.

```
int p[N];
void combination(int n, int k) {
  if (n == 0) {
    print();
    return;
  }
  if (k > n || k < 0)
    return;
  p[k] = n;
  combination(n - 1, k - 1);
  combination(n - 1, k);
}
```

А ещё иногда в задачах просят научиться генерировать сочетания, которые отличаются ровно в одном элементе. Мы на самом деле почти это сделали, нужно лишь поставить вызовы рекурсии в нужном порядке и отреверсить одну часть. В общем то, формула у нас будет такая: $\{C_n^k\} = \{C_{n-1}^k\} \cup \{C_{n-1}^{k-1} \cup n\}^R$

Можно либо написать это и ревёрсить ответ, либо изощратся в порядке, приведём код второго способа.

```
bool used[N];

void gen(int n, int k, int l, int r, bool rev, int old_n) {
  if (k > n || k < 0)
    return;
  if (n == 0) {
    for (int i = 0; i < old_n; ++i)
      if (used[i])
        cout << ans[i] << ' ';
    cout << '\n';
    return;
  }
  used[rev?r:l] = false;
  gen(n-1, k, !rev?l+1:l, !rev?r:r-1, rev, old_n);
  used[rev?r:l] = true;
```

```
16    gen(n-1, k-1, !rev?l+1:l, !rev?r:r-1, !rev, old_n);
17  }
18
19  gen(n, k, 0, n-1, false, n);
```

5.4.3 Разбиения

Есть ещё один комбинаторный объект, о котором можно поговорить — это разбиение числа, то есть разложение его на сумму нескольких строго положительных слагаемых, обозначение: n_λ. Генерировать все разбиения тоже очень просто: будем генерировать их в лексикографически убывающем порядке.

```
1  int top = 0;
2  int p[N];
3
4  void gen(int n, int last = n) {
5    if (n == 0) {
6      print();
7      return;
8    }
9    top++;
10   for (int i = last; i >= 1; i--) {
11     p[top - 1] = i;
12     gen(n - i, i);
13   }
14 }
```

5.5 Литература

1. «Комбинаторика для программистов» В. Липсицкий
2. «Комбинаторика» Виленкин
3. http://e-maxx.ru/algo/
4. Раздел «комбинаторика» на http://neerc.ifmo.ru/wiki/