Задача 1.
Дана матрица. На некоторые клетки можно наступать, на некоторые нет. Сколько существует путей из левой нижней клетки в правую верхнюю, если нельзя ходить вниз и налево?

```
good[i][j] - можно ли наступать
dp[i][j] - ответ для клетки (i,j)

1.
if (good[i][j]) {
	dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1] + dp[i][j - 1];
}

2.
if (i + 1 < n && good[i + 1][j])
	dp[i + 1][j] += dp[i][j];
if (i + 1 < n && j + 1 < m && good[i + 1][j + 1])
	dp[i + 1][j + 1] += dp[i][j];
if (j + 1 < m && good[i][j + 1])
	dp[i][j + 1] += dp[i][j];
```

Задача 2.
Дан ориентированный граф без циклов. Найти самый длинный путь в нем из А в Б.

```
1.
Сортим вершины в порядке топологической сортировки.
Затем:
for (x : [1,n]) {
	for (prev : (prev,x) - ребро) {
		dp[x] = max(dp[x], dp[prev] + 1);
	}
}

2.
for (x : [1,n]) {
	for (next : (x,next) - ребро) {
		dp[next] = max(dp[next], dp[x] + 1);
	}
}

Как восстановить ответ?
if (dp[x] < dp[prev] + 1) {
	dp[x] = dp[prev] + 1;
	parent[x] = prev;
}
```

3а. Есть отрезок длины n. Можно прыгать на a[1], a[2], …, a[k] вперед. Сколько всего вариантов попасть из 0 в n?

```
1.
dp[0] = 1;
for (int x = 1; x <= n; x++) {
      for (int i = 0; i < k; i++) {
            if (x - a[i] >= 0) {
                  dp[x] += dp[x - a[i]];
            }
      }
}

2.
dp[0] = 1;
for (int x = 0; x < n; x++) {
      for (int i = 0; i < k; i++) {
            if (x + a[i] <= n) {
                  dp[x + a[i]] += dp[x];
            }
      }
}
```

3б.
То же самое, но n <= 10^18, k <= 50, a[i] <= 50. Естественно, по модулю.
Для простоты такой пример:
dp[x] = 2*dp[x - 1] + dp[x - 3] + dp[x - 4]

<table>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td rowspan="4">*</td><td>dp[x]</td><td rowspan="4">=</td><td>dp[x + 1]</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>dp[x + 1]</td><td>dp[x + 2]</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>dp[x + 2]</td><td>dp[x + 3]</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>2</td><td>dp[x + 3]</td><td>dp[x + 4]</td></tr>
</table>

A * (dp[0], dp[1], dp[2], dp[3]) = (dp[1], dp[2], dp[3], dp[4])
A * A * (dp[0], dp[1], dp[2], dp[3]) = (dp[2], dp[3], dp[4], dp[5])
…..
A^(n-3) * (dp[0], dp[1], dp[2], dp[3]) = (dp[n-3], dp[n-2], dp[n-1], dp[n])

Осталось научиться возводить матрицы в степень.

Как умножать матрицы (за n^3, где n - размер матрицы):

```
for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++) {
                  res[i][j] += a[i][k] * a[k][j];
            }
```

Как возводить в степень.
Научимся сначала просто числа возводить в степень.
Как бы вы считали 3^100?
3^100 = 3^50 * 3^50. (тут важно не посчитать 3^50 два раза. Надо один раз посчитать и запомнить.)
Если посчитать 3^50, можно возвести его в квадрат. А размерность задачи при этом падает в 2 раза.
Как посчитать 3^25?
Это просто, 3^25 = 3 * 3^24 = 3 * 3^12 * 3^12.

Это работает за log(p), где p - степень, в которую возводим число.

4.
Дано дерево. В каждой вершине записано число v[i].
Надо выбрать такое множество вершин, чтобы среди выбранных никакие две не были соединены ребром, а сумма v[i] в выбранных вершинах была бы максимальна. Вывести сумму.

Будем считать две величины:
a[i] - это максимальная сумма v[i] в поддереве вершины i, причем вершина i выбрана
b[i] - это максимальная сумма v[i] в поддереве вершины i, причем вершина i не выбрана

Как получить результат для вершины x, если известны результаты для всех ее потомков?
Если вершина x выбрана, то ни один ее потомок не может быть выбран. Поэтому a[x] = v[x] + sum(b[child])
Если же вершина x не выбрана, то ее потомки могут быть какими угодно. Поэтому b[x] = sum(max(a[child], b[child]))

Ответ будет равен max(a[root], b[root]).

```
void dfs(int x, int p) {
      a[x] = v[x];
      b[x] = 0;
      for (int y : g[x]) {
            if (y != p) {
                  dfs(y, x);
                  a[x] += b[y];
                  b[x] += max(a[y], b[y]);
            }
      }
}
```

5. http://contest.samara.ru/ru/problemset/5658/

Будем считать одну величину:
dp[x] = минимальное время, за которое можно обучить поддерево вершины x, при условии, что вершина x уже обучена (т.е. если начать в момент времени 0, самый последний чувак в поддереве обучится в момент времени dp[x]).
(если у вершины нет потомков, ответ для нее 0)

Допустим, что мы посчитали dp[i] для всех потомков вершины x. Как теперь получить dp[x]?

Понятно, что в каждый момент времени выгодно учить того потомка, для которого dp[i] максимально. Поэтому посортим все dp[i] по убыванию (пусть это будет v[0], v[1], …, v[k-1]).

Теперь рассмотрим максимальное dp[i]. (оно хранится в v[0]). Сколько его надо обучать? На обучение корня этого поддерева потратится 1, а на обучение всего поддерева потратится v[0]. Значит, на обучение максимального потомка потребуется v[0] + 1 времени.

А сколько надо времени, чтоб обучать второго по величине потомка? v[1] + 2. Для третьего v[2] + 3. И т.д.
Максимальное из этих чисел и даст нам dp[x].

Ответ хранится в dp[0].

```
void dfs(int x, int p) {
      List<Integer> v = new ArrayList<>();
      for (int y : g[x]) {
            if (y != p) {
                  dfs(y, x);
                  v.add(dp[y]);
            }
      }
      Collections.sort(v); // каждая вершина выступает потомком ровно 1 раз
      Collections.reverse(v); // поэтому суммарный размер посорченных вершин -
n
      dp[x] = 0;
      for (int i = 0; i < v.size(); i++) {
            dp[x] = Math.max(dp[x], v.get(i) + i + 1);
      }
}
```

7.
i = [0, n], j = [0, m]
dp[i][j] = dp[i - 1][j - 2] + dp[i - 1][j - 1] + dp[i - 1][j];
Посчитать за O(m) памяти.

В каждый момент времени нужны только две строки: i-1-ая и i-ая. Когда посчитали i-ую, i-1 можно забыть.
Например, можно делать так: dp[i][j] = dp[1 - i][j - 2] + dp[1 - i][j - 1] + dp[1 - i][j];
(1 - i) дает 0, если i=1, и 1, если i=0. Так строки будут чередоваться.
Потом, в зависимости от четности, надо взять ответ либо из первой, либо из второй строки.

8.
i = [0, n], j = [0, m]
dp[i][j] = dp[i - 1][j - k] + ... + dp[i - 1][j];
Посчитать за O(n*m) времени.

Втупую, за O(n*m*m), было бы так:

```
for (int i = 1; i <= n; i++) {
      for (int j = 0; j <= m; j++) {
            for (int z = j - k; z <= j; z++) {
                  dp[i][j] += dp[i - 1][z];
            }
      }
}
```

Введем префикс суммы:
```
for (int i = 1; i <= n; i++) {
      // считаем префикс суммы предыдущего ряда
      pref[0] = dp[i - 1][0];
      for (int j = 1; j <= m; j++) {
            pref[j] = pref[j - 1] + dp[i - 1][j];
      }
      for (int j = 0; j <= m; j++) {
            dp[i][j] = pref[j];
            if (j - k - 1 >= 0) dp[i][j] -= pref[j - k - 1];
            // теперь dp[i][j] = dp[i - 1][j - k] + … + dp[i][j]
      }
}
```

9. Рюкзак.
Есть n предметов. i-ый предмет стоит c[i] и весит w[i]. Рюкзак выдерживает вес maxW. Надо запихать в него предметы, чтоб продать их подороже.

dp[i][j] - максимальная стоимость предметов в рюкзаке, если просмотрено было первых i предметов, а вес взятых предметов равен j. При этом, какие именно из первых i предметов мы взяли, а какие нет - неважно.

Пусть для i предметов мы все посчитали. Рассматриваем (i+1)-ый предмет. Переберем, сколько щас весит рюкзак. Пусть это j.

Тогда либо не берем (i+1)-ый предмет, и это значит, dp[i + 1][j] = max(dp[i + 1][j], dp[i][j]) (т.е. предмет рассмотрели, и поэтому i увеличилось на 1, но предмет не взяли, т.е. вес рюкзака не изменился)
Либо берем, и тогда вес рюкзака станет уже не j, а j + w[i]:
dp[i + 1][j + w[i]] = max(dp[i + 1][j + w[i]], dp[i][j] + c[i]).

Для динамики назад это будет выглядеть так:
dp[i][j] = max(dp[i][j], dp[i - 1][j]);
dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + c[i]);

Почему всегда максимум? Потому что одна и та же комбинация (i, j) (т.е. количество просмотренных предметов + вес) могла получиться разными способами, и какой-то из них выгоднее. Поэтому и максимум.

Ответ хранится в одном из dp[n][j] (надо взять максимум)