Задача A. Ферзя в угол

Вспомним как мы определеляем выигрыши и проигрыши в симметричной игре: состояние выигрышно, если их него есть ход в проигрышное, состояние проигрышно, если все ходы из него выигрышны.

Введем $d_{i,j}$, соответствующее выигрышности клетки (i, j). Изначально $d_{1,1} = LOSE$. Тогда мы сразу знаем, что $d_{1,*} = d_{*,1} = d_{i,i} = WIN$, потому что из них есть переход в $(1, 1)$. Для общего правила пересчета такой динамики определим переходы более четко.

Из $d_{i,j}$ есть переходы в $d_{k,j}(k < i), d_{i,k}(k < j), d_{i-k,j-k}(k < i, k < j)$.

Пересчитывать динамику можно как назад (посчитать $d_{i,j}$ через описанные выше переходы), так и вперед (для каждого найденного проигрышного состояния добавлять новые выигрышные состояния, из которых был переход в проигрышное).

```
for i in range(m - 1, -1, -1):
    for j in range(n - 1, -1, -1):
        if(Table[i][j] == LOSE):
            for Id in range(j):
                Table[i][Id] = WIN
            for Id in range(i):
                Table[Id][j] = WIN
            for Id in range(1, min(i, j) + 1):
                Table[i - Id][j - Id] = WIN
```

Задача B. Монетки

Давайте переберем все подмножества и проверим, что находится подмножество с нужной суммой. Чтобы перебрать все подмножества, переберем маски от 0 до $2^n - 1$, найдем сумму чисел для каждого подмножества, и выведем найденный ответ.

```
for (int mask = 0; mask < (1 << m); mask++) {
    int sum = 0;
    for (int i = 0; i < m; i++) {
        bool subset_has_i = (mask >> i) & 1;
        if (subset_has_i) {
            sum += a[i];
        }
    }
    if (sum == n) {
        // print answer
        break;
    }
}
```

Задача C. Разложение Фибоначчи

Давайте определим $f(n)$ как количество способов представить n как произведение чисел Фибоначчи.

Сначала приведем интуитивное, но неправильное решение задачи, а потом превратим его в правильное.

Как пересчитывать $f(n)$ рекурсивно? Нужно перебрать все делители в виде чисел Фибоначчи и решить меньшую задачу. Числа Фибоначчи можно предподсчитать заранее, их достаточно мало (100-е число уже точно больше 10^{18}).

$$f(n) = \sum_k f(\frac{n}{F_k})$$

Как выглядит перебор для функции выше? Спускаясь по стеку рекурсии мы каждый раз выписываем какой-то делитель, и когда перебор закончился в числе 1 — мы нашли какое-то разложение, а наши рекурсивные переходы как раз и соответствуют этому разложению.

В таком решении корректность оказывается нарушена, потому что наш перебор учитывает порядок пересчета: $2 \cdot 5 \cdot 8$ и $2 \cdot 8 \cdot 5$ соответствуют разным порядкам пересчета, и мы в рекурсии прибавим оба, хотя это одно и то же множество и должно быть учтено один раз.

Для того, чтобы задать порядок, мы добавим еще один аргумент в $f(n)$ — $f(n, p)$ будет соответсвовать тому, что мы раскладываем число n на множители, и на последнем рекурсивном переходе мы делили на F_p (по умолчанию $p = 0$). Тогда формула трансформируется в:

$$f(n, p) = \sum_{k \geqslant p} f(\frac{n}{F_k}, k)$$

Такой перебор каждое множество учитывает ровно один раз - в отсортированном порядке.

Для большей красоты и некоторой оптимизации, поскольку мы уже ввели параметр p, можно проверять делимость только на F_p, а иначе рекурсивно проверять $p + 1$.

```
ll f(ll n, int p) {
    if (n == 1) return 1;
    if (p >= fib.size()) return 0;
    if (n < fib[p]) return 0;

    ll res = f(n, p + 1);
    if (n % fib[p] == 0) {
        res += f(n / fib[p], p);
    }
    return res;
}
```

Задача D. Разбиения на слагаемые

Поскольку нам требуется разбиение на невозрастающие слагаемые, в переборе можно поддерживать прошлое число в разбиении и число, которое сейчас нужно доразложить на слагаемые. При переборе мы пробуем все слагаемые, которые меньше чем наше ограничение с прошлых шагов рекурсии.

```
def part(index, mx, num):
    if (num == 0):
        for i in range(index):
            print(x[i], end=' ')
        print()
    else:
        for i in range(1, min(mx, num) + 1):
            x[index] = i
            part(index + 1, i, num - i)
```

Задача E. Альфа Дерево

Эта задача требовала сделать $\alpha - \beta$-отсечение. Деревом перебора в этой игре выступало бинарное дерево, описанное в условие. Соответственно, перебор предполагает, что после обхода одного из сыновей, можно по условию $\alpha < \beta$ отсекать вторую ветку перебора.

Чтобы не строить дерево в явном виде (что не влезет в ограничения по памяти), можно воспользоваться каким-нибудь неявным способом задать бинарное дерево. Например, можно в аргументах обхода хранить самого левого и самого правого листа в поддереве.

```
private fun abPruning(
    left: Long,
    right: Long,
```

```
    alpha: Long,
    beta: Long,
    player: Player
): Long {
    if (left == right) return f(left)
    val middle = (left + right) / 2
    if (player == Player.SECOND) {
        beta = minOf(beta, abPruning(left, middle, alpha, beta, Player.FIRST))
        if (beta <= alpha) return beta
        beta = minOf(beta, abPruning(middle + 1, right, alpha, beta, Player.FIRST))
        return beta
    } else {
        alpha = maxOf(alpha, abPruning(left, middle, alpha, beta, Player.SECOND))
        if (beta <= alpha) return alpha
        alpha = maxOf(alpha, abPruning(middle + 1, right, alpha, beta, Player.SECOND))
        return alpha
    }
}
```