Рекурсивный перебор и комбинаторные объекты

1 A. Без двух единиц подряд

Разбор

Обратим внимание на ограничения в этой задаче. Длина последовательностей не превышает двадцать символов. Один символ принимает максимум два значения: 0 и 1. Тогда общее число последовательностей сверху ограничено $2^{20} < 2 \times 10^6$. Значит, мы можем сделать полный перебор всех возможных последовательностей с учетом требования "без двух единиц подряд".

Опишем рекурсивный переход:

$$s_i = \begin{cases} 1, s_{i-1} \neq 1 \vee i = 0 \\ 0 \end{cases}$$

Т.е. мы добавляем к последовательности ноль **всегда** независимо от того, какой символ стоял ранее, и единицу, если ранее стоял ноль или последовательность была пустой. Для соблюдения лексикографического порядка будем сначала добавлять в последовательность ноль, а потом пытаться добавить единицу.

Асимптотика решения: $O(2^N)$

Решение

```
def solve(n, seq=[]):
    if n == 0:
        print(*seq)
        return
    solve(n - 1, seq + [0])
    if not seq or seq[-1] == 0:
        solve(n - 1, seq + [1])

n = int(input())
solve(n)
```

2 B. Сочетания-2

Разбор

В данной задаче также предполагалось решение на основе рекурсивного перебора. Разберёмся, как можно построить сочетание рекурсивно:

1. На первом шаге возьмём какое-нибудь число, которое будет началом сочетания. Какие границы будут у этого числа? Так как последовательности убывающие, то мы не можем взять число больше n и меньше $n - k$ (т.к. минимальное убывающее сочетание: $n - k, n - k - 1, ..., 1$).

2. На втором шаге мы можем взять любое число не больше предыдущего и не меньше $n - k - 1$

3. На последнем шаге мы можем взять любое число не больше предыдущего и не меньше 1

По достижении последовательностью длины n просто выведем её. Лексикографический порядок нам обеспечивает перебор чисел от меньшего к большему на каждом шаге.

Сложность решения: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Решение

```
def solve(k, n, seq, min_val):
    if min_val == 0:
        print(*seq)
        return
    max_val = n if not seq else seq[-1] - 1
    for i in range(min_val, max_val + 1):
        solve(k, n, seq + [i], min_val - 1)


k, n = map(int, input().split())
solve(k, n, [], k)
```

3 C. Генерация правильных скобочных последовательностей – 2

Разбор

В этой задаче *просто* переберём все последовательности. Возникает вопрос: как нам это сделать? Для начала рассмотрим последовательности, состоящие только из круглых скобок.

Понятно, что начать последовательность мы можем только с открывающей скобки. Теперь мы можем продлить последовательность либо ещё одной открывающей скобкой, либо закрыть скобку. Сколько закрывающих скобок мы можем поставить? Понятно, что не больше, чем число "свободных"открывающих скобок – то есть, пока баланс скобок положительный.

А сколько открывающих скобок мы можем поставить? Так как мы их ставим не обязательно подряд, переформулируем вопрос: до каких пор мы можем добавлять в последовательность открывающие скобки? Нам должно хватить длины, чтобы закрыть все открытые на текущий момент скобки. Тогда становится понятно, что нам нужно, чтобы число незакрытых открывающих скобок b было меньше числа оставшихся позиций в строке.

Как решить эту же задачу для разных типов скобок? Заведём вместо одного счётчика стек и будем смотреть, какой тип открывающих скобок лежит на вершине стека и в зависимости от этого добавлять закрывающую скобку нужного типа.

Асимптотика решения: $O(2^N)$

Решение

```
from collections import deque

def solve(n, seq: list = [], opened_pos: deque = deque(), i: int = 0):
    if len(seq) == n:
        print(''.join(seq))
        return
    balance = len(opened_pos)
    if balance + 1 < n - len(seq):
        opened_pos.append(i)
        solve(n, seq + ['('], opened_pos, i + 1)
        solve(n, seq + ['['], opened_pos, i + 1)
        opened_pos.pop()
    if balance > 0:
        last_opened_idx = opened_pos.pop()
        if seq[last_opened_idx] == '(':
            solve(n, seq + [')'], opened_pos, i + 1)
        if seq[last_opened_idx] == '[':
            solve(n, seq + [']'], opened_pos, i + 1)
        opened_pos.append(last_opened_idx)


n = int(input())
if n % 2 == 0:
    solve(n, [])
```

4 D. Задача о рюкзаке

Разбор

В этой задаче переберём все возможные комбинации включения или не-включения предметов в рюкзак. Для этого будем делать два рекурсивных перехода. Если мы можем вместить i-й предмет по весу, добавим его в рюкзак и обновим стоимость. Иначе просто пропустим его.

В задании также было требование о минимально возможном наборе предметов максимальной стоимости. Для этого в самом конце, когда мы рассмотрели все n предметов и включили или не включили их в рюкзак, попробуем обновить стоимость. Если она выше, то просто обновим ответ. Если стоимость равна уже накопленной, то выберем набор предметов наименьшей длины.

Асимптотика решения: $O(2^N)$

Решение

```
fin = open('knapsack.in', 'r')
fout = open('knapsack.out', 'w')

n, W = map(int, fin.readline().split())
items = [tuple(map(int, fin.readline().split())) for _ in range(n)]

def solve(w, c=0, ans=[], max_cost=0, best_ans=[], i=0):
    if c > max_cost or c == max_cost and len(ans) < len(best_ans):
        max_cost, best_ans = c, ans
    if i == n or w == 0:
        return max_cost, best_ans
    w_i, c_i = items[i]
    if w - w_i >= 0:
        max_cost, best_ans = solve(w - w_i, c + c_i, ans + [i + 1], max_cost, best_ans, i + 1)
    max_cost, best_ans = solve(w, c, ans, max_cost, best_ans, i + 1)
    return max_cost, best_ans

max_cost, ans = solve(W, 0)
print(len(ans), max_cost, file=fout)
print(*ans, file=fout)
```

5 E. Монетки

Разбор

В этой задаче нам нужно сделать три рекурсивных перехода. Во-первых, аналогично задаче о рюкзаке, мы можем пропустить i-ю монетку. Во-вторых, мы можем взять одну i-ю монетку, если оставшаяся сумма больше её номинала. В-третьих, мы можем взять обе i-х монетки (опять-таки, если они умещаются в остаток суммы). В конце, когда мы рассмотрели все n монеток, аналогично задаче о рюкзаке возьмём наименьший набор монеток.

Чтобы заранее определить, что мы не сможем расплатиться монетками, просто просуммируем все номиналы, умножим на два и сравним с требуемой суммой.

Асимптотика решения: $O(3^N)$

Решение

```
def solve(s, c, i=0, ans=[], best_ans=[]):
    if s == 0:
        if not best_ans:
            return ans
        return min(ans, best_ans, key=lambda x: len(x))
    if i == len(c):
        return best_ans
    c_i = c[i]
    best_ans = solve(s, c, i + 1, ans, best_ans)
    if s - c_i >= 0:
```

```
        best_ans = solve(s - c_i, c, i + 1, ans + [c_i], best_ans)
    if s - 2 * c_i >= 0:
        best_ans = solve(s - 2 * c_i, c, i + 1, ans + [c_i, c_i], best_ans)
    return best_ans

n, m = map(int, input().split())
c = list(map(int, input().split()))

if 2 * sum(c) < n:
    print(-1)
else:
    res = solve(n, c)
    print(len(res))
    print(*res)
```

6 F. Перестановка по номеру

Разбор

Сначала кажется, что мы можем сгенерировать перестановку по номеру рекурсивным перебором. Однако, посчитав, чему равен 12!, мы понимаем, что рекурсия тут нам не поможет. Надо придумать итеративный вариант решения.

Воспользуемся следующей идеей решения: если на i-м шаге мы поставили в перестановку k-е из **ещё не использованных**, то мы пропустили $k \times (n-i-1) \times (n-i-2) \times ... \times 1$ лексикографических перестановок.

Тогда будем вести подсчёт следующих величин:

- Набора использованных чисел из перестановки $1, 2, ..., n$
- Для каждого разряда перестановки - количество рассмотренных чисел o из перестановки $1, 2, ..., n$ **без учета** уже использованных

Если в какой-то момент номер перестановки лежит между $o \times (n-i-1) \times (n-i-2) \times ... \times 1$ и $(o+1) \times (n-i-1) \times (n-i-2) \times ... \times 1$, то мы можем записать текущее рассматриваемое число на i-ю позицию и вычесть из k число уже рассмотренных перестановок.

Асимптотика решения: $O(N^2)$

Решение

```
n = int(input())
k = int(input())

used = [0] * (n + 1)
f = [1] * (n + 1)
for i in range(1, n + 1):
    f[i] = f[i - 1] * i
ans = [0] * n
for pos in range(n):
    cnt = 0
    for i in range(n):
        if used[i + 1] == 0:
            if cnt * f[n - pos - 1] <= k < (cnt + 1) * f[n - pos - 1]:
                k -= cnt * f[n - pos - 1]
                cnt += 1
                used[i + 1] = 1
                ans[pos] = i + 1
                break
            else:
                cnt += 1
print(*ans)
```

7 G. Номер правильной последовательности

Разбор

Для начала научимся считать вспомогательную динамику $dp_{i,j}$, где i — длина скобочной последовательности (она "полуправильная": всякой закрывающей скобке соответствует парная открывающая, но не все открытые скобки закрыты), j — баланс (т.е. разность между количеством открывающих и закрывающих скобок), $dp_{i,j}$ — количество таких последовательностей. При подсчёте этой динамики мы считаем, что скобки бывают только одного типа.

Считать эту динамику можно следующим образом. Пусть $dp_{i,j}$ — величина, которую мы хотим посчитать. Если i=0, то ответ понятен сразу: $dp_{0,0} = 1$, все остальные $dp_{0,j} = 0$. Пусть теперь $i > 0$, тогда переберём, чему мог быть равен последний символ этой последовательности. Если он был равен '(', то до этого символа мы находились в состоянии $(i-1, j-1)$. Если он был равен ')', то предыдущим было состояние $(i-1, j+1)$. Таким образом, получаем формулу: $dp_{i,j} = dp_{i-1,j-1} + dpi - 1, j + 1$, считая, что все значения $dp_{i,j}$ при отрицательном j равны нулю. Таким образом, эту динамику мы можем посчитать за квадрат.

Рассмотрим теперь решение основной задачи. Пусть сначала допустимы только скобки одного типа. Заведём счётчик b глубины вложенности (баланса скобок) и будем двигаться по последовательности слева направо. Если текущий символ $s_i, i = 0 \dots n-1$ равен '(', то мы увеличиваем b на 1 и переходим к следующему символу. Если же текущий символ равен ')', то мы должны прибавить к ответу $dp_{n-i-1,b+1}$, тем самым учитывая, что в этой позиции мог бы стоять символ '(', который бы привёл к лексикографически меньшей последовательности, чем текущая; затем мы уменьшаем b на единицу.

Теперь разрешим два вида скобок. На шаге i мы имеем $n-i-1$ неопределённых позиций, из которых $b \pm 1$ являются скобками, закрывающими какие-то из открытых ранее, — значит, тип таких скобок мы варьировать не можем. А вот все остальные скобки (а их будет $(n-i-1-b\pm1)/2$ пар) могут быть любого из 2 типов, поэтому ответ умножается на $2^{(n-i-1-b\pm1)/2}$.

Откуда берётся ±1? Фактически в этой формуле мы просто обобщаем случай, когда мы рассматриваем открывающую либо закрывающую скобки, т.к. для разных случаев нам нужны разные состояния динамики (либо с меньшим балансом, либо с большим).

Асимптотика решения: $O(N^2)$

Решение

```
import math

with open('brackets2num2.in', 'r') as inf:
    psp_inp = inf.readline().rstrip()
n = len(psp_inp)
m = n//2 + 1
dp = [[0 for x in range(m)] for z in range(n+1)]
dp[0][0] = 1
for i in range(1, n+1):
    for j in range(m):
        if j + 1 < m:
            dp[i][j] += dp[i-1][j+1]
        if j-1 > -1:
            dp[i][j] += dp[i-1][j-1]

num = 0
bal = 0
stack = []
for i in range(n-1):
    curr = psp_inp[i]
    if curr == '(':
        stack.append('(')
        bal += 1
        continue
    elif bal + 1 < m:
        num += dp[n - i - 1][bal + 1] * int(math.pow(2, (n - i - 1 - (bal + 1)) // 2))
    if curr == ')':
        stack.pop()
        bal -= 1
```

```
            continue
        elif stack:
            if '(' == stack[-1]:
                num += dp[n - i - 1][bal - 1] * int(math.pow(2, (n - i - 1 - (bal - 1)) // 2))
        if curr == '[':
            stack.append('[')
            bal += 1
            continue
        elif bal + 1 < m:
            num += dp[n - i - 1][bal + 1] * int(math.pow(2, (n - i - 1 - (bal + 1)) // 2))
        if curr == ']':
            stack.pop()
            bal -= 1
            continue

with open('brackets2num2.out', 'w') as outf:
    outf.write(str(num))
```

8 H. Новогодняя гирлянда

Разбор

Перед тем, как решать эту задачу, научимся генерировать лексикографически следующую строку обычного алфавита. Пусть нам дана произвольная строка "azbxczz". Как мы можем её увеличить ровно на 1 шаг? Будем искать символ, который можно увеличить, начиная с конца. Почему? Потому что две строки тем ближе друг к другу в лексикографическом порядке, чем больше у них общий префикс. К примеру, для заданной выше строки строка "azbxzzz" ближе, чем строка "azbyczz". Понятно, что в исходной строке наилучший символ, который нам подходит – это 'c', стоящий на 5-й позиции (3-й с конца). Увеличим этот символ на 1, но что делать с суффиксом? Оказывается, на все позиции после изменяемой надо поставить лексикографически минимальную строку. Если мы работаем с обычными символами, то это просто "aaa...". Итак, для нашей простой строки ответом будет являться строка "azbxdaa".

Теперь научимся решать эту же задачу для наших "необычных" строк. В нашем случае лексикографически минимальной строкой будет являться последовательность "ududud...". Между символами 'd' и 'u' мы не можем вставить 'h', т.к. не может быть двух вершин, имеющих одну координату y. Когда мы должны делать замену суффикса на эту строку? Будем идти с конца строки и смотреть на баланс.

- Допустим, мы встретили символ 'd'. Лексикографически следующим символом является 'h'. При неизменной величине баланса 'u' и 'h' мы можем потратить ровно на 1 символ меньше. Поэтому осуществим подстановку, если $n - i - 1 > b_i$. Затем исправим остаток суффикса и завершим работу алгоритма.
- Допустим, мы встретили символ 'h'. Лексикографически следующим символом является 'u', поэтому продолжить мы должны его только символом 'd'. Баланс не изменился, а длина возможного суффикса уменьшилась на 1. Поэтому осуществим подстановку аналогично предыдущему случаю, исправим остаток суффикса и завершим работу алгоритма.
- Если мы встретили символ 'u', то мы можем его пропустить и перейти к следующему символу, т.к. это лексикографически максимальный символ в строке.

Асимптотика решения: $O(N)$

Решение

```
n = int(input())
s = list(input())

ctoi = { 'u': 1, 'd': -1 }
itoc = { 0: 'u', 1: 'd' }

def get_balance(s, n):
    p_balance = [0] * (n + 1)
```

```
    for i in range(n):
        p_balance[i + 1] = p_balance[i] + ctoi.get(s[i], 0)
    return p_balance

def fix_str(s, n, i):
    b = get_balance(s, n)
    b_i = b[i]
    while b_i > 0:
        s[i] = 'd'
        i, b_i = i + 1, b_i - 1
    m, rem = n - i, (n - i) % 2
    for k in range(m - rem):
        s[i + k] = itoc[k % 2]
    if rem == 1:
        s[i + m - 2], s[i + m - 1] = 'h', 'd'
    return s

def solve(s, n):
    b = get_balance(s, n)
    for i in range(n - 1, -1, -1):
        if s[i] == 'd' and b[i + 1] + 1 < n - i:
            s[i], s[i + 1] = 'h', 'd'
            return fix_str(s, n, i + 2)
        if s[i] == 'h' and b[i + 1] + 1 < n - i:
            s[i], s[i + 1] = 'u', 'd'
            return fix_str(s, n, i + 2)
    return s

x = solve(s.copy(), n)
print('No solution' if x == s else ''.join(x))
```