0. Задача о взвешенном независимом множестве в путевом графе

Для работы со стохастическим динамическим программированием необходимо сначала лучше понять (или вспомнить), что такое динамического программирования в общем случае. Для этого решим с его помощью сложную и конкретную вычислительную задачу.

Определение. Пусть G = (V, E) - произвольный неориентированный граф. Независимым множеством графа G называется подмножество $S \subseteq V$ взаимно несмежных вершин. К примеру, если вершины графа G обозначают людей, а рёбра соединяют тех, кто знаком друг с другом, то независимым множеством G будет любое подмножество незнакомых друг с другом людей.

Определение. Путевым графом будем называть дерево следующего вида:

Image 1

Условие задачи Пусть задан неориентированный путевой граф G = (V, E) и неотрицательные веса $w_v \ \forall v \in V$ (чаще встречается граф со взвешенными рёбрами, но в этой задаче нам интересен именно граф со взвешенными вершинами). Требуется найти независимое множество $S \subseteq V$

графа G с максимально возможной суммой $\sum_{v \in S} w_v$ вершин, входящих в S.

Число независимых множеств путевого графа растёт экспоненциально, поэтому решение перебором всех возможных независимых множеств крайне неэффективно и неосуществимо при больших n, где n - число вершин. Существуют различные способы решить данную задачу более эффективно, но ни один из них не превосходит в оптимальности линейно-временной алгоритм, который мы вскоре получим.

Идея заключается в сведении решения задачи к решению нескольких подзадач. Иначе говоря, мы хотим решить большую непонятную задачу, используя решения некоторых маленьких понятных задач. Пусть в графе G n вершин (будем называть такой граф G_n). Рёбра будем обозначать как $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ (по номерам соединяемых ими вершин). Обозначим независимое множество графа G_n с максимальным весом как $MWIS(G_n)$ (maximum-weight independent set), а его суммарный вес как W_n .

I. Пусть n = 1. Очевидно, что $MWIS(G_n) = MWIS(G_1) = \{v_1\}$. $W_1 = w_1$

II. Пусть n = 2. Две вершины графа соединены ребром, поэтому только одна из них может входить в MWIS. То есть

$$MWIS(G_n) = MWIS(G_2) = \left[\begin{array}{l} \{v_1\},\ w_1 >= w_2 \\ \{v_2\},\ \text{иначе} \end{array} \right.$$

Тогда $W_2 = \max(w_1, w_2)$

III. Пусть теперь n >= 3. Очевидно, что или $v_n \in MWIS(G_n)$, или $v_n \notin MWIS(G_n)$. Рассмотрим оба этих случая.

1. $v_n \notin S$. Тогда $MWIS(G_n) = MWIS(G_{n-1})$, где G_{n-1} - путевой граф на вершинах $v_1, v_2, \ldots, v_{n-1}$. Действительно, независимое взвешенное мн-во графа G_n, не содержащее вершины v_n, по определению является независимым взвешенным мн-вом графа G_{n-1}. Также верно, что независимое мн-во графа G_{n-1} по определению является независимым мн-вом графа G_n. И если бы в графе G_{n-1} существовало независимое множество с большим суммарным весом, чем у $MWIS(G_n)$, то следовательно, и в графе G_n было бы независимое множество с большим суммарным весом, чем у $MWIS(G_n)$, а это является противоречием определению MWIS.
2. $v_n \in S$. Тогда $MWIS(G_n) = MWIS(G_{n-2}) \cup \{v_n\}$. Действительно, $MWIS(G_{n-2}) \cup \{v_n\}$ по определению является независимым мн-вом графа G_n. Осталось проверить, что $W_n - w_n = W_{n-2}$. Пусть в G_{n-2} существует независимое мн-во $MWIS^*(G_{n-2})$ с весом $W^*_{n-2} > W_n - w_n$. Но тогда $MWIS^*(G_{n-2}) \cup v_n$ - независимое мн-во графа G_n и его вес равен $W^*_{n-2} + w_n > W_n$, т. е. $MWIS(G_n)$ не является независимым мн-вом максимального веса, что противоречит определению MWIS.

Следовательно,

$$MWIS(G_n) = \left[\begin{array}{l} MWIS(G_{n-1}) \\ MWIS(G_{n-2}) \cup \{v_n\} \end{array} \right.$$

Таким образом, существует всего 2 возможных варианта для $MWIS(G_n)$. Как следствие, получаем рекурентное соотношение для W_n при $n \geq 3$.

$$W_n = \max(W_{n-1}, W_{n-2} + w_n)$$

Или в более общем случае для i = 3, 4, ..., n:

$$W_i = \max(W_{i-1}, W_{i-2} + w_i)$$

Теперь можем написать рекурсивную функцию, которая будет вычислять W_n для любого неориентированного путевого графа G.

In [9]:
```
class RecurAlg: # чтобы не передавать каждый раз w
    def __init__(self, w):
        self.w = [0] + w # чтобы индексация велась с 1, а не с 0

    def W(self, i):
        if (i == 1):
            return self.w[1] # база
        elif (i == 2):
            return max(self.w[1], self.w[2]) # база

        return max(self.W(i - 1), self.W(i - 2) + self.w[i]) # пере
ход


n = int(input())
weights = list(map(int, input().split()))
ra = RecurAlg(weights)
print('Ответ: Wn равно', ra.W(n))
```

```
4
1 4 5 4
Ответ: Wn равно 8
```

Показан пример работы для графа:

Image 2

Тестирование на нескольких примерах показало, что ф-ия работает верно. Найдём асимптотику рекурсивного алгоритма. В каждом узле дерева рекурсии W(i) мы вызываем W(i - 1) и W(i - 2), пока не дойдём до листа W(1) или W(2). Следовательно, если корень дерева - это W(n), то высота дерева - n. Т. к. каждый уровень шире предыдущего в 2 раза, то вызов W(n) работает за $O(1 + 2 + ... + 2^n) = O(2^n)$. Весьма неэффективно, попробуем как-то оптимизировать наш алгоритм. Заметим, что мы много раз вызываем функцию от одного аргумента. Лучше будем запоминать результат работы функции и "доставать" его, когда он снова понадобится. Оптимизация сработает очень хорошо: существует всего n различных вариантов запуска ф-ии(W(1), W(2), ..., W(n)). Таким образом, понадобится всего лишь O(n) памяти. Можно было бы оставить алгоритм рекурсивным и получилась бы так называемая "рекурсия с запоминанием". Но гораздо логичнее и удобнее начать вычислять наши подзадачи не с последней, а с первой. Когда мы считаем W(i) нам достаточно знать W(i - 1) и W(i - 2). Значит, если мы пройдёмся i по циклу от 1 до n, то на каждом шаге мы без проблем сможем вычислять W(i) ((W(i - 1) и W(i - 2) к тому моменту уже будут посчитаны).

In [8]:
```
n = int(input())
w = list(map(int, input().split()))
w = [0] + w # чтобы индексация велась с 1, а не с 0
W = [0] * (n + 1)
W[1] = w[1] # база
W[2] = max(w[1], w[2]) # база
for i in range (3, n + 1):
    W[i] = max(W[i - 1], W[i - 2] + w[i]) # переход
print('Ответ: Wn равно', W[n])
```

```
4
1 4 5 4
Ответ: Wn равно 8
```

Итак, мы получили алгоритм, линейный по времени и по памяти. Подобное решение задачи и называется *динамическим программированием*.

Замечание о применимости динамического программирования

Но любой ли алгоритм, основанный на динамическом программировании правильно решает поставленную задачу? Кажется, что нет. Но хочется получить какое-то формальное требование, по которому можно будет проверять корректность построенной модели. Попробуем вывести такой принцип из примера, где он не будет выполняться.

Рассмотрим следующую задачу. Дано 2 массива a и b, содержащих N произвольных целых чисел каждый. $X = x_1 \cdot x_2 \cdot \ldots \cdot x_{N-1} \cdot x_N$, где x_i - i-ый элемент одного из массивов. Требуется найти максимально возможное значение X (обозначим его Xm). То есть для каждого i = 0, 1, ..., N - 1 (массивы нумеруются с 0) мы хотим так выбрать a_i или b_i, чтобы произведение всех выбранных чисел было максимально возможным, а ответом на задачу является это максимальное значение произведения. Например, если N = 3, a = [1, 10, 0], b = [3, 2, 5], то ответ на задачу является $\max(X) = 3 \cdot 10 \cdot 5 = 150$. Построим следующий алгоритм динамического программирования. Пусть подзадачей k называется задача, аналогичная исходной, с N = k, в которой в качестве массивов рассматриваются a[:k] и b[:k] - k-элементные срезы оригинальных массивов a и b (взятые от начала). Например, если в исходной задаче N = 3, a = [1, 10, 0], b = [3, 2, 5], то в её подзадаче 2: N = 2, a = [1, 10], b = [3, 2]. Заведём массив Xm, такой что Xm[i] - ответ на подзадачу i + 1. Очевидно, что $Xm[0] = \max(a[0], b[0])$. Возьмём кажущуюся разумной на первый взгляд рекурентную формулу $\forall i = 1, \ldots, N-1 \mapsto Xm[i] = Xm[i-1] \cdot \max(a[i], b[i])$. Итого имеем:

$$\begin{cases} Xm[0] = \max(a[0], b[0]) \\ \forall i = 1, \ldots, N-1 \mapsto Xm[i] = Xm[i-1] \cdot \max(a[i], b[i]) \end{cases}$$

Реализуем данный алгоритм на практике и получим несложную программу, представленную ниже:

In [8]:
```
def solve_task():
    N = int(input())
    a = list(map(int, input().split()))
    b = list(map(int, input().split()))
    Xm = [0] * N
    Xm[0] = max(a[0], b[0]) # база
    for i in range(1, N):
        Xm[i] = Xm[i - 1] * max(a[i], b[i]) # переход
    print(Xm[N - 1])

solve_task()
```

```
3
1 10 0
3 2 5
150
```

Но действительно ли программа работает правильно? Простой пример показывает, что нет.

In [9]:
```
solve_task()
```

```
2
1 -100
-100 -10
-10
```

Программа выдаёт -10, хотя взяв -100 и -100 мы можем получить произведение 10000. В чём же дело? Оказывается, что наша рекуррентная формула некорректна: мы не можем перейти от Xm[i - 1] к X[m]. Это происходит из-за того, что отрицательное число при умножении на отрицательное даёт положительное, и поэтому взяв в произведение чётное кол-во отрицательных множителей мы получим положительный результат, который может оказаться ответом на задачу. Наш же алгоритм всегда "предпочтёт взять" маленькое по модулю положительное число или 0, вместо большого по модулю отрицательного (а именно это большое по модулю отрицательное число, умноженное на другое отрицательное число и ставшее положительным, может быть ключом к максимальному произведению). Сформулируем причину некорректности алгоритма более формально. Пусть f – функция сопоставляющая задаче её решение, $\bigoplus$ - операция композиции. В построенной нами модели не выполняется аддитивность функции f относительно $\bigoplus$ (решение задачи может не совподать с композицией решений подзадач, на которые эта задача была разбита). Таким образом, если мы разделим динамическое программирование на базу и переход (по аналогии с математической индукцией), то понятно, что опасность заключается именно в корректности перехода (база обычно очевидна). И теперь основывясь на приведённом случае, когда динамическое программирование "ломается" и формальной причиной, почему это происходит, можем сформулировать важное условие, которое должно выполняться, чтобы выбранный алгоритм динамического программироания верно решал поставленную задачу.

Должна выполняться аддитивность функции, сопоставляющей задаче (или подзадаче) её решение, относительно операции композиции подзадач. Говоря математическим языком, пусть f – функция сопоставляющая задаче её решение, ⊕ - операция композиции. Тогда для любой задачи x (для которой определено f(x)), представимой в виде композиции конечного числа подзадач $y_1, y_2, \dots, y_n$ выполняется: $f(x) = f(y_1 \oplus y_2 \oplus \dots \oplus y_n) = f(y_1) \oplus f(y_2) \oplus \dots \oplus f(y_n)$

То есть требуется, чтобы объединение решений подзадач давало корректное решение общей задачи (иначе говоря, предъявляется требование к корректности индукционного перехода). Все предложенные в дальнейшем алгоритмы этому требованию удовлетворяют.

Модель I. Замена случайной величины математическим ожиданием

Идея первой рассматриваемой нами модели стохастического динамического программирования крайне проста. Пусть дана некоторая задача, которую теоритически можно решить динамическим программированием, но некоторые параметры заданы в ней не числами, а вероятностным распределением (такая задача называется стохастической). Тогда посчитаем математическое ожидание этих параметров и подставим их в задачу (так делается очень часто, в качестве значения случайной величины берётся её мат. ожидание). То есть мы будем использовать среднее, взвешенное по вероятностям значение этих параметров, что выглядит весьма разумно. Так стохастическая задача превратилась в детерминированную и её можно решить обычным динамическим программированием. Очевидно, что сложность алгоритма никак не изменится ни по времени, ни по памяти.

1. Стохастическая задача о рюкзаке

Применим вышеописанную модель к одной из самых известных задач на динамическое программирование - задаче о рюкзаке, представленной в стохастическом варианте. Вор грабит музей. Он взял с собой рюкзак, чтобы сложить туда краденое, но все предметы, которые он хотел украсть, не помещаются в рюкзак. Вору предстоит решить, какие предметы брать, а какие нет. Разумеется он хочет наполнить рюкзак так, чтобы суммарная цена взятых им предметов была максимально возможной. Точные цены предметов вор не знает (да и о какой точной цене может идти речь, если сбываешь товар на чёрном рынке), но для каждого предмета примерно представляет вероятностное распределение его стоимости (вор прошёл курс по рыночной аналитике от Тинькофф). Вероятностное распределение - либо дискретное, либо непрерывное равномерное на отрезке. Нужно составить оптимальную стратегию для вора, какие предметы взять, чтобы максимизировать ожидаемый доход от продажи краденого. Формализуем задачу.

Дано:

- n - кол-во предметов
- $s_1, s_2, \ldots, s_n$ - целочисленные размеры предметов
- C - максимальная вместимость рюкзака (целое число)
- Далее для каждого предмета: если распределение стоимости дискретное, то вводится $m \leq 10$ - кол-во возможных значений стоимости. Затем числа $p_1, \ldots, p_m$ и $v_1, \ldots, v_m$ - вероятности исходов и соответсующие им значения стоимости (в задаче о рюкзаке то, что мы хотим максимизировать, принято называть ценностью, отсюда v). Если же распределение непрерывное равномерное, то даны v_{min} и v_{max} - границы отрезка.

Необходимо найти подмножество $S \subseteq \{1, 2, \ldots, n\}$ предметов с максимально возможной суммой $\sum\limits_i E(v_i)$ при условии, что $\sum\limits_i s_i \leq C$ (заменили случайные величины на их математические ожидания).

Сначала надо придумать, как разбить задачу на подзадачи. Будем действовать по аналогии с задачей о графе. Пусть S(k, x) - искомое подмн-во S при n = k, C = x, а $V_{n,c} = \sum\limits_{i \in S(n,c)} E(v_i)$.

I. Рассмотрим S(n, c) при n = 1. У нас есть ровно 1 предмет, который мы можем или взять или нет. Очевидно, что мы можем взять его, только если его размер не превышает c. А т. к. $E(v_i) \geq 0$, то мы должны взять этот предмет, если есть такая возможность. Т. е.

$$S(n, c) = S(1, c) = \begin{cases} \{1\}, \text{ если } s_1 \leq c \\ \{\emptyset\}, \text{ иначе} \end{cases}$$

Значит,

$$V_{n,c} = V_{1,c} = \begin{cases} E(v_1), \text{ если } s_1 \leq c \\ 0, \text{ иначе} \end{cases}$$

II. Пусть теперь n > 1. Разобъём задачу на 2 дизъюнктивных случая - когда $n \in S(n, c)$ и когда $n \notin S(n, c)$.

1. $n \notin S(n, c)$. Тогда нетрудно показать, что $S(n, c) = S(n-1, c)$.
2. $n \in S(n, c)$. Тогда $S(n, c) = S(n-1, c-s_n) \cup n$, если $s_i <= c$ (иначе, этот случай

невозможен). Действительно, если $s_i > c$, то мы не можем использовать предмет n, чтобы набрать предметы суммарным размером не болеее, чем с. Если же $n \in S(n, c)$, то значит, среди первых n - 1 предметов мы набрали суммарную ценность $c - s_i$. ч. т. д.

Следовательно,

$$S(n, c) = \begin{cases} S(n-1, c-s_n) \cup \{n\}, \text{ если } n \in S(n, c) \\ S(n-1, c), \text{ иначе} \end{cases}$$

Тогда получаем рекурентную формулу для V_n при n > 1

$$V_{n,c} = \begin{cases} V_{n-1,c}, \text{ если } s_n > c \\ \max(V_{n-1,c}, V_{n-1,c-s_n} + Ev_n), \text{ иначе} \end{cases}$$

Или в более общем случае для каждого i = 2, ..., n

$$V_{i,c} = \begin{cases} V_{i-1,c}, \text{ если } s_i > c \\ \max(V_{i-1,c}, V_{i-1,c-s_i} + Ev_i), \text{ иначе} \end{cases}$$

Теперь можем написать программу, динамически вычисляющую V_i, c для всех возможных i и с (по базе для n = 1 и рекурентной формуле). Это возможно, потому что С и все размеры предметов - целые числа. Тогда очевидно, что $i \in \{1, 2, \ldots, n\}$, а с может принимать любые целые значения от 0 до C включительно. Значит, посчитав базу для i = 1 (для всех $c \in \{0, 1, \ldots, C\}$), можем считать все остальные возможные $V_{i,c}$ в двойном цикле (по i и по с).

In [1]:

```
# в программе для удобства нумерация ведётся с 0
n = int(input()) # число предметов
s = list(map(int, input().split())) # размеры предметов
C = int(input()) # вместительность рюкзака
Ev = [0] * n # массив математических ожиданий цен
for i in range(n):
    print('Введите данные о ', i + 1, '-ом предмете', sep='')
    # считываем информвцию о i предмете
    type = input()
    if (type == 'discrete'): # дискретное распределение
        p = list(map(float, input().split())) # вероятности возможн
ых исходов
        v = list(map(float, input().split())) # цены на i-ый предме
т при этих исходах
        m = len(p)
        for j in range(m):
            Ev[i] += p[j] * v[j]
    else: # непрерывное равномерное распределение
        v_min, v_max = map(float, input().split()) # границы отрезк
а распределения
        Ev[i] = (v_min + v_max) / 2

V = [] # двумерный массив из всех возможных V_i,c
V.append([0] * (C + 1))
for c in range(1, C + 1): # база
    if (s[0] <= c):
        V[0][c] = Ev[0]

for i in range(1, n):
    V.append([0] * (C + 1))
    for c in range(C + 1):
        if (s[i] > c): # взять предмет i чисто физически невозможно
            V[i][c] = V[i - 1][c]
        else: # взять предмет i теоритически возможно
            V[i][c] = max(V[i - 1][c], V[i - 1][c - s[i]] + Ev[i])

print('Ожидаемый доход:', V[n - 1][C])
```

```
4
2 4 1 2
6
Введите данные о 1-ом предмете
discrete
0.4 0.6
30 100
Введите данные о 2-ом предмете
segment
0 80
Введите данные о 3-ом предмете
segment
7.5 20
Введите данные о 4-ом предмете
discrete
0.6 0.2 0.2
10 20 50
Ожидаемый доход: 112.0
```

При программировании задачи мы сдвинули индексацию по i на 1 назад ($i \in \{0, 1, \ldots, n-1\}$). Это сделано для удобства, т. к. в питоне индексация массивов начинается с 0, и ни на что не влияет. Посчитаем асимптотику алгортима: ввод данных осуществляется за O(10n) = O(n) (учли максимально возможное m для каждого предмета), базу мы считаем за O(C), рекурентоне вычисление - за O(nC), и вывод ответа - за O(1). Итого, сложность программы: O(n) + O(C) + O(nC) + O(1) = O(nC). Однако на данный момент мы научились только находить $V_{n,c}$ - ожидаемый доход от продажи ценностей, но от самого этого числа проку мало, вору хотелось бы узнать, какие именно предметы ему нужно взять, чтобы достичь такого результата. Итак, задача: создать алгоритм, который будет восстанавливать S(n, C), зная все $V_{i,c}$ (такой подход приводит к самому простому решению).

Заметим, что по $V_{i,c}$ мы всегда можем понять, взяли мы i-ый предмет в S или нет. Пусть i = 1. Вспоминаем, что мы берём единственный предмет тогда и только тогда, когда его ценность не превышает c ($s_1 \leq c$). Пусть $i \neq 1$. Во-первых, если $s_i > c$, то мы чисто физически не могли взять предмет i. Если же $s_i \leq c$, то обратившись к полученной ранее рекурентной формуле понимаем, что мы взяли i, если это было выгодней, чем не брать его, т. е. если $V_{i-1,c-s_i} + Ev_i \geq V_{i-1,c}$. Теперь можем написать линейный алгоритм, восстанавливающий S(n, C).

In [5]:
```
S = []
c = C
for i in range(n - 1, -1, -1): # идём от конца в начало
    if (i != 0 and s[i] <= c and V[i - 1][c - s[i]] + Ev[i] >= V[i
- 1][c]):
        # если предмет i + 1 входит в S, добавляем его в ответ
        S.append(i + 1)
        c = c - s[i]
    elif (i == 0 and s[0] <= c):
        # отдельная проверка на включение/не включение в ответ 1-го
предмета
        S.append(i + 1)
S.reverse()
print(S)
```

```
[1, 2]
```

Сложность восстановления O(n), а значит, суммарная асимптотика алгоритма остаётся O(nC).

2. Азартная игра

Вращается колесо, по периметру которого нанесены числа 1 ... n. Вероятность того, что колесо в результате одного вращения остановится на цифре i, равна p_i. Игрок платит x долларов за возможность осуществить не более m вращений колеса. Игрок получит сумму, равную удвоенному числу, которое выпало при последнем вращении колеса. Требуется разработать оптимальную стратегию для игрока.

$f_i(j)$ - ожидаемый выигрыш при условии, что игра находится на этапе (вращении) i, исходом последнего вращения было число j, и в дальнейшем мы будем действовать оптимально.

Тогда:

$f_m(j) = 2j$

$f_i(j) = max[2j, \sum_{k=1}^{n} p_k f_{i+1}(k)]$, i = 1, 2, ..., m - 1

Иначе говоря:

$$f_i(j) = max[2j, E(f_{i+1})],$$

где $E(f_{i+1})$ - мат ожидание f_{i+1}.

Будем решать все задачи раздела 15.2 с помощью динамического программирования: перебирая этапы *с конца* (от m до 0), где i-ый этап - момент времени *после i-го вращения, но до (i + 1)-го*. Т. к. для определения $f_i(j)$ нам достаточно $\forall k$ знать вероятность выпадания k на рулетке и $f_{i+1}(k)$, то идя с конца и находя на каждом этапе i все возможные $f_i(j)$, мы найдём $f_0(0)$ - ожидаемую прибыль при оптимальной игре.

Если при этом для каждой возможной ситуации мы будем запоминать ход, который нужно сделать, чтобы прийти к наилучшему возможному результату, то будет построено *дерево оптимальной стратегии*, т. е. на каждом этапе для любого возможного события, мы будем знать, что делать дальше для достижения оптимального результата.

In [8]:
```
import numpy as np # для работы с данными
import pandas as pd # для вывода данных в виде таблиц
```

Пример 2.1

Предположим, что по периметру колеса русской рулетки расставлены числа от 1 до 5. Вероятности p_i остановки колеса на числе i соответсвенно равны следующему: $p_1 = 0.3$, $p_2 = 0.25$, $p_3 = 0.2$, $p_4 = 0.15$, $p_5 = 0.1$. Игрок платит 5 долларов за возможность сделать не более 4 вращений колеса.

Найти: оптимальную стратегию игрока и ожидаемый ваигрыш.

Дано:

n = 5, m = 4,

$$p_1 = 0.3,\ p_2 = 0.25,\ p_3 = 0.2,\ p_4 = 0.15,\ p_5 = 0.1$$

x = 5.

In [26]:
```
f = {}  # f[i, j] - ожидаемый выигрыш, если при i-м вращении выпало j

n = int(input())  # сколько чисел
m = int(input())  # сколько вращений
p = [0]
p += list(map(float, input().split()))  # вероятности
x = int(input())  # цена игры


def Ef(i):  # считаем E(f(i))
    res = 0
    for k in range(1, n + 1):
        res += p[k] * f[i, k]
    return res


for j in range(1, n + 1):  # определяем f для всех исходов последнего вращения
    f[m, j] = 2 * j

tabledata = [] # ячейки таблицы
for i in range(n):
    row = [''] * (m - 1)
    tabledata.append(row)

for i in range(m - 1, -1, -1):  # динамически определяем f для всех i и j
    if i == 0:
        f[i, 0] = Ef(i + 1)
        break

    for j in range(1, n + 1):
        f[i, j] = max(2 * j, Ef(i + 1))
        if (2 * j > Ef(i + 1)):
            tabledata[j - 1][i - 1] = 'закончить'
        else:
            tabledata[j - 1][i - 1] = 'продолжить'
prof = f[0, 0] - x  # находим прибыль, которую мы ожидаем получить при оптимальной стратегии игры
print('Ожидание прибыли:', prof)

In = [] # строки таблицы
for i in range(n):
    In.append('Если выпало ' + str(i + 1) + ', то выгодней:')
C = [] # столбцы таблицы
for i in range(m - 1):
    C.append('Вращение ' + str(i + 1))

pd.DataFrame(tabledata, index=In, columns=C) # вывод таблицы
```

```
5
4
0.3 0.25 0.2 0.15 0.1
5
Ожидание прибыли: 2.309375
```

Out[26]:

	Вращение 1	Вращение 2	Вращение 3
Если выпало 1, то выгодней:	продолжить	продолжить	продолжить
Если выпало 2, то выгодней:	продолжить	продолжить	продолжить
Если выпало 3, то выгодней:	продолжить	продолжить	закончить
Если выпало 4, то выгодней:	закончить	закончить	закончить
Если выпало 5, то выгодней:	закончить	закончить	закончить

Упражнение 2.1

Условия аналогичны предыдущему примеру. Требуется найти опримальную стратегию игрока и ожидаемый выигрыш.

Дано:

n = 8, m = 5,

$p_1 = p_2 = \ldots = p_8 = \frac{1}{8}$

x = 5.

In [27]:
```
f = {} # f[i, j] - ожидаемый выигрыш, если при i-м вращении выпало
j

n = int(input()) # сколько чисел
m = int(input()) # сколько вращений
p = [0]
p += list(map(float, input().split())) # вероятности
x = int(input()) # цена игры

def Ef(i): # считаем E(f(i))
    res = 0
    for k in range(1, n + 1) :
        res += p[k] * f[i, k]
    return res

for j in range(1, n + 1): # определяем f для всех исходов последнег
о вращения
    f[m, j] = 2 * j

tabledata = [] # ячейки таблицы
for i in range(n):
    row = [''] * (m - 1)
    tabledata.append(row)

for i in range(m - 1, -1, -1): # динамически определяем f для всех
i и j
    if i == 0:
        f[i, 0] = Ef(i + 1)
        break

    for j in range(1, n + 1):
        f[i, j] = max(2 * j, Ef(i + 1))
        if (2 * j > Ef(i + 1)):
            tabledata[j - 1][i - 1] = 'закончить'
        else:
            tabledata[j - 1][i - 1] = 'продолжить'

prof = f[0, 0] - x # находим прибыль, которую мы ожидаем получить п
ри оптимальной стратегии игры
print('Ожидание прибыли:', prof)

In = [] # строки таблицы
for i in range(n):
    In.append('Если выпало ' + str(i + 1) + ', то выгодней:')
C = [] # столбцы таблицы
for i in range(m - 1):
    C.append('Вращение ' + str(i + 1))

pd.DataFrame(tabledata, index=In, columns=C) # вывод таблицы
```

```
8
5
0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125
5
Ожидание прибыли: 8.3828125
```

Out[27]:

	Вращение 1	Вращение 2	Вращение 3	Вращение 4
Если выпало 1, то выгодней:	продолжить	продолжить	продолжить	продолжить
Если выпало 2, то выгодней:	продолжить	продолжить	продолжить	продолжить
Если выпало 3, то выгодней:	продолжить	продолжить	продолжить	продолжить
Если выпало 4, то выгодней:	продолжить	продолжить	продолжить	продолжить
Если выпало 5, то выгодней:	продолжить	продолжить	продолжить	закончить
Если выпало 6, то выгодней:	продолжить	продолжить	закончить	закончить
Если выпало 7, то выгодней:	закончить	закончить	закончить	закончить
Если выпало 8, то выгодней:	закончить	закончить	закончить	закончить

Упражнение 2.2

Человек хочет продать свой подержанный автомобиль тому, кто предложит наивысшую цену. Изучая автомобильный рынок, он пришёл к выводу, что с равными вероятостями ему за автомобиль могут предложить очень низкую цену (1050 долларов), просто низкую цену (1900 долларов), среднюю цену (2500 долларов), либо высокую цену (3000 долларов). Человек решил помещать объявление о продаже автомобиля на протяжении не более трёх дней подряд. В конце каждого дня ему нужно решить принимать, принять ли наилучшее предложение, поступившее в течение этого дня.

Найти: оптимальную стратегию поведения для продавца автомобиля.

Запишем все возможные предложения покупателей в массив w (индексация с 1): $w[j]$ - кол-во денег в предложении № j. Задача решается динамическим программированием аналогично задаче про рулетку (перебираем дни с конца и считаем $f_i(j)$ $\forall i, j$, где $f_i(j)$ - максимум ожидаемой прибыли при условии, что в i-ый день наилучшее предложение - предложение № j). Соответсвенно при вычислении $f_i(j)$ будем пользоваться ф-лой:

$$f_i(j) = max(w[j], E(f_{i+1}))$$

Дано:

m = 3, n = 4

$w[1] = 1050, w[2] = 1900, w[3] = 2500, w[4] = 3000$

$p_1 = p_2 = p_3 = p_4 = \frac{1}{4}$

In [30]:

```
f = {} # f[i, j] - ожидаемый доход, если в день i наилучшее предлож
ение - это j

m = int(input()) # кол-во дней
w = [0]
w += list(map(float, input().split())) # предложения (нумерация с 1
)
n = len(w) - 1 # кол-во предложений
p = [0.0] * (n + 1) # вер-ти предложений
for i in range(1, n + 1):
    p[i] = 1 / n

print('Вероятности:', p[1:])

def Ef(i): # считаем E(f(i))
    res = 0
    for k in range(1, n + 1) :
        res += p[k] * f[i, k]
    return res

for j in range(1, n + 1): # определяем f для всех исходов последнег
о дня
    f[m, j] = w[j]

tabledata = [] # ячейки таблицы
for i in range(n):
    row = [''] * (m - 1)
    tabledata.append(row)

for i in range(m - 1, 0, -1): # динамически определяем f для всех i
и j
    for j in range(1, n + 1):
        f[i, j] = max(w[j], Ef(i + 1))
        if (w[j] > Ef(i + 1)):
            tabledata[j - 1][i - 1] = 'продать'
        else:
            tabledata[j - 1][i - 1] = 'подождать'

f[0, 0] = Ef(1)

prof = f[0, 0] # находим прибыль, которую мы ожидаем получить при о
птимальной стратегии игры
print('Ожидание прибыли:', prof)

In = [] # строки таблицы
for i in range(1, n + 1):
    In.append('Если предложили ' + str(w[i]) + ', то выгодней:')
C = [] # столбцы таблицы
for i in range(m - 1):
    C.append('День ' + str(i + 1))

pd.DataFrame(tabledata, index=In, columns=C) # вывод таблицы
```

```
3
1050 1900 2500 3000
Вероятности: [0.25, 0.25, 0.25, 0.25]
Ожидание прибыли: 2590.625
```

Out[30]:

	День 1	День 2
Если предложили 1050.0, то выгодней:	подождать	подождать
Если предложили 1900.0, то выгодней:	подождать	подождать
Если предложили 2500.0, то выгодней:	продать	продать
Если предложили 3000.0, то выгодней:	продать	продать

3. Задача инвестирования

Некто планирует заработать деньги на фондовой бирже за последующие n лет и у него есть C тысяч долларов начального капитала. Инвестиционный план состоит в покупке акций в начале года и продаже их в конце этого же года. Накопленные деньги затем могут быть снова инвестированы (все или их часть) в начале слудующего года. Степень риска инвестиции представлена тем, что прибыль имеет вероятностный характер. Изучение рынка свидетельствует о том, что возможно m случаев (произойдёт ровно 1 из этих случаев). Случай k происходит с вероятностью p_k ($\sum_{k=1}^{m} p_k = 1$) и приводит к прибыли r_k (считается в долях).

Вопрос: как следует инвестировать C тысяч долларов для наибольшего накопления к концу n лет?

Введём некоторые обозначения

Пусть x_i - сумма денежных средств, доступных для инвестирования в начале i-го года ($x_1 = \mathrm{C}$),

y_i - сумма реальной инвестиции в начале i-го года ($y_i \le x_i$),

а $f_i(x_i)$ - полученный капитал, ожидаемый при оптимальном инвестировании к концу n-го года при условии, что в начале i-го года капитал равен x_i

В учебнике, из которого взята эта задача, приведены следующие размышления:

Для k-го условия рынка имеем следующее:

$x_{i+1} = (1 + r_k)y_i + (x_i - y_i) = r_k y_i + x_i$, k = 1, 2, .., m

Тогда рекурентное уравнение динамического программирования имеет следующий вид:

$f_i(x_i) = \max\limits_{0 \le y_i \le x_i} \left\{ \sum\limits_{k=1}^{m} p_k f_{i+1}(x_i + r_k y_i) \right\}$, i = 1, 2, ..., n,

где $f_{n+1}(x_{n+1}) = x_{n+1}$, т. к. после n-го года инвестиции нет. Отсюда следует, что

$$f_n(x_n) = \max_{0 \le y_n \le x_n} \left\{ \sum_{k=1}^{m} p_k(x_n + r_k y_n) \right\} = x_n \sum_{k=1}^{m} p_k(1 + r_k) = x_n(1 + p_1 r_1 + p_2 r_2 + \ldots + p_m r_m),$$

поскольку функция в фигурных скобках является линейной по y_n и, следовательно, достигает своего максимума при $y_n = x_n$.

Если продолжить размышления автора, можно получить интересный результат:

Во-первых, заметим, что последняя фолмула $\left(f_n(x_n) = x_n(1 + \sum\limits_{k=1}^{m} p_k r_k)\right)$ справедлива только если $\forall k\colon r_k \geq 0$, т. к. иначе мы не можем быть уверены, что функция в фигурных скобках достигает своего максимума при $y_n = x_n$. В дальнейших задачах, как и в реальной жизни, возникают ситуации, когда прибыль может быть отрицательна (неудачное инвестирование, потеря части или всех вложенных средств), поэтому рассмотрим случай, когда $\forall k\colon r_k$ может принимать любые значения из отрезка $[-1; +\infty)$.

1. Помним, что $f_{n+1}(x_{n+1}) = x_{n+1}$
2. $f_n(x_n) = \max\limits_{0 \leq y_n \leq x_n} \left\{ \sum\limits_{k=1}^{m} p_k(x_n + r_k y_n) \right\}$. Рассмотрим выражение в фигурных скобках:

$$\sum_{k=1}^{m} p_k(x_n + r_k y_n)$$

Пусть $\frac{y_n}{x_n} = c_n, 0 \leq c_n \leq 1$. Тогда $y_n = c_n x_n$

$$\sum_{k=1}^{m} p_k(x_n + r_k c_n x_n) = x_n \sum_{k=1}^{m} p_k(1 + r_k c_n) = x_n(p_1 + \ldots + p_m + p_1 r_1 c_n + \ldots p_m r_m c_n) = x_n(1 + c_n \sum_{k=1}^{m}$$

Обозначим $\sum\limits_{k=1}^{m} p_k r_k$ как $E(r)$ (ведь это ни что иное как мат ожидание прибыли) Тогда получаем:

$$f_n(x_n) = \max_{0 \leq c_n \leq 1} \{x_n(1 + c_n E(r))\} = \begin{cases} x_n(1 + E(r)),\ E(r) > 0 \\ x_n,\ E(r) \leq 0 \end{cases}$$

Т. е. если $E(r) > 0$, то нужно вкладывать все имеющиеся деньги ($y_n = x_n$), иначе не вкладывать их вообще ($y_n = 0$).

В дальнейшем будем рассматривать 2 отдельных случая: (I) когда $E(r) > 0$ и (II) когда $E(r) \leq 0$.

I.

Предложение 1. Если $E(r) > 0$, то $\forall i \in \{1, 2, \ldots, n+1\} \rightarrow f_i(x_i) = x_i(1 + E(r))^{n-i+1}$.

Докажем его методом обратной математической индукции. База уже есть.

Шаг индукции: пусть $f_j(x_j) = x_j(1 + E(r))^{n-j+1}$.

Проверим, что $f_{j-1}(x_{j-1}) = x_{j-1}(1 + E(r))^{n-j+2}$.

$$f_{j-1}(x_{j-1}) = \max_{0 \leq y_{j-1} \leq x_{j-1}} \left\{ \sum_{k=1}^{m} p_k f_j(x_{j-1} + r_k y_{j-1}) \right\}$$

Снова рассмотрим выражение в фигурных скобках, сделав замену $\frac{y_{j-1}}{x_{j-1}} = c_{j-1}$ $(0 \leq c_{j-1} \leq 1)$.

$$\sum_{k=1}^{m} p_k f_j(x_{j-1} + r_k c_{j-1} x_{j-1}) = \sum_{k=1}^{m} p_k x_{j-1}(1 + r_k c_{j-1})(1 + E(r))^{n-j+1} = x_{j-1}(1 + E(r))^{n-j+1} \sum_{k=1}^{m} p_k($$

$$= x_{j-1}(1 + E(r))^{n-j+1} \left(\sum_{k=1}^{m} p_k + c_{j-1} \sum_{k=1}^{m} p_k r_k \right) = x_{j-1}(1 + E(r))^{n-j+1}(1 + c_{j-1} E(r))$$

Т. к. $E(r) > 0$, то максимум $f_{j-1}(x_{j-1})$ достигается при $c_{j-1} = 1$. Тогда имеем:

$$f_{j-1}(x_{j-1}) = x_{j-1}(1 + E(r))^{n-j+1}(1 + E(r)) = x_{j-1}(1 + E(r))^{n-j+2}$$

ч. т. д.

$II.$

Предложение 2. Если $E(r) \leq 0$, то $\forall i \in \{1, 2, \ldots, n+1\} \to f_i(x_i) = x_i$.

Как и в прошлом пункте, докажем его методом обратной индукции. База уже есть.

Шаг индукции: пусть $f_j(x_j) = x_j$.

Проверим, что $f_{j-1}(x_{j-1}) = x_{j-1}$.

$$f_{j-1}(x_{j-1}) = \max_{0 \leq y_{j-1} \leq x_{j-1}} \left\{ \sum_{k=1}^{m} p_k f_j(x_{j-1} + r_k y_{j-1}) \right\}$$

После замены $\frac{y_{j-1}}{x_{j-1}} = c_{j-1}$ $(0 \leq c_{j-1} \leq 1)$ имеем:

$$f_{j-1}(x_{j-1}) = \max_{0 \leq c_{j-1} \leq 1} \left\{ \sum_{k=1}^{m} p_k f_j(x_{j-1} + r_k c_{j-1} x_{j-1}) \right\} = \max_{0 \leq c_{j-1} \leq 1} \left\{ \sum_{k=1}^{m} p_k (x_{j-1} + r_k c_{j-1} x_{j-1}) \right\} = {}_0$$

$$\max_{0 \leq c_{j-1} \leq 1} \left\{ x_{j-1}(1 + \sum_{k=1}^{m} p_k r_k c_{j-1}) \right\} = \max_{0 \leq c_{j-1} \leq 1} \left\{ x_{j-1}(1 + c_{j-1} E(r)) \right\}$$

Т. к. $E(r) \leq 0$, то максимум $f_{j-1}(x_{j-1})$ достигается при $c_{j-1} = 0$. Тогда:

$$f_{j-1}(x_{j-1}) = x_{j-1}$$

ч. т. д.

Таким образом,

для любого i $f_i(x_i)$ может быть найдена по формуле:

$$f_i(x_i) = \begin{cases} x_i(1 + E(r))^{n-i+1}, \ E(r) > 0 \\ x_i, \ E(r) \leq 0 \end{cases}$$

Эта формула говорит о том, что если использовать данную математическую модель в данной задаче, то оптимальная стратегия состоит либо в том, чтобы каждый год инвестировать все имеющиеся средства, либо в том, чтобы не инвестирвовать ничего, и выбор наилучшей из этих противоположных стратегий зависит только от значения $E(r)$. Причём эта зависимость согласована со здравым смыслом. Если $E(r) > 0$, то мы ожидаем положительную прибыль. Значит, чем больше денег мы вложим в начале года, тем больше получим в конце.

Следовательно, стоит инвестировать максимально возможное количесво денег, т. е. каждый год все имеющиеся деньги. Если же $E(r) \leq 0$, то мы ожидаем, что вложенная сумма денег не увеличится или даже уменьшится. Тогда логично инвестировать как можно меньше, т. е. не инвестировать вообще.

Замечание. Случай $E(r) = 0$ неслучайно отнесён к ситуации, когда не надо вкладывать деньги (могло показаться, что всё равно, к какой ситуации его отнести). Дело в том, что в реальной жизни, если от инвестирования не ожидается дохода, его не делают, потому что эти деньги можно вложить во что-то другое, что уже принесёт доход.

Таким образом, выяснилось, что иногда задача с вероятностной неопределённостью, содержащая рекурсивные вычисления, может быть решена не динамическим программированием, а нахождением формулы, сразу дающий ответ.

Пример 3.1

Решим пример к вышепредставленной математической модели. Перед началом инвестирования имеется C = 10 000 долларов. Всего на инвестироваине есть 4 года. Каждый год существует 40%-ная вероятность удвоить вложенные деньги, 20%-ная - остаться при своих деньгах и 40%-ная - потерять весь объём инвестиции.

Необходимо разработать оптимальную стратегию инвестирования.

Дано:

C = 10 000, n = 4, m = 3,

$p_1 = 0.4$, $p_2 = 0.2$, $p_3 = 0.4$,

$r_1 = 2$, $r_2 = 0$, $r_3 = -1$.

In [5]:
```
d = {} # d[i, x] - fi(x) - ожидаемый чрез n лет капитал, если в i-м году капитал равен x
# f(i, x) - ожидаемое кол-во денег в конце игры если в начале i-го года: x руб

C = int(input()) # начальный капитал
n = int(input()) # кол-во лет
m = int(input()) # кол-во условий
p = [0] # вероятности
p += list(map(float, input().split()))
r = [0] # процентные ставки, зависящие от условий
r += list(map(float, input().split()))

const = 1 + np.dot(p, r) # 1 + E(r)
cap = 0 # капитал через n лет
if const <= 1:
    print('Выгодней вообще не инвестировать')
    cap = C
else:
    print('Выгодней инвестировать каждый год все имеющиеся деньги')
    cap = C * const ** n # x * (1 + E(r)) ^ n
print('При оптимальном инвестировании капитал к концу n-го года:', round(cap, 4))

print('')

if const > 1:
    def k(i): # (1 + E(r)) ^ (n - i + 1)
        return const ** (n - i + 1)

    for i in range (1, n + 1):
        print('Коэффициент увелечения C после', i, 'года:', round(k(n - i + 1), 4))
```

```
10000
4
3
0.4 0.2 0.4
2 0 -1
Выгодней инвестировать каждый год все имеющиеся деньги
При оптимальном инвестировании капитал к концу n-го года: 38416.0

Коэффициент увелечения C после 1 года: 1.4
Коэффициент увелечения C после 2 года: 1.96
Коэффициент увелечения C после 3 года: 2.744
Коэффициент увелечения C после 4 года: 3.8416
```

Упражнение 3.1

Задача отличается от предыдущей тем, что каждый год условия, определяющие прирост капитала, меняются. Так что каждый год будем принимать решение, вкладывать все имеющиеся деньги или не вкладывать ничего, в зависимости от того, больше 1 + E(r) единицы в этом году, или нет. Корректность решения следует из математической модели предыдущей задачи (каждый год можно рассмотреть как отдельную задачу, аналогичную примеру 15.3-1, с n = 1).

Дано:

C = $10 000, n = 4, m = 3,

1-ый год: $r_1 = 2, r_2 = 1, r_3 = 0.5; p_1 = 0.1, p_2 = 0.4, p_3 = 0.5$

2-ой год: $r_1 = 1, r_2 = 0, r_3 = -1; p_1 = 0.4, p_2 = 0.4, p_3 = 0.2$

3-ий год: $r_1 = 4, r_2 = -1, r_3 = -1; p_1 = 0.2, p_2 = 0.4, p_3 = 0.4$

4-ый год: $r_1 = 0.8, r_2 = 0.4, r_3 = 0.2; p_1 = 0.6, p_2 = 0.2, p_3 = 0.2$

In [6]:

```
C = int(input()) # начальный капитал
n = int(input()) # кол-во лет
m = int(input()) # кол-во условий
p = [[0]] # вероятности
r = [[0]] # # процентные ставки, зависящие от условий
for i in range (1, n + 1):
    p.append([0])
    p[i] += list(map(float, input().split()))
    r.append([0])
    r[i] += list(map(float, input().split()))

const = [0] * (n + 1)
for i in range(1, n + 1):
    const[i] = 1 + np.dot(p[i], r[i]) # 1 + E(r_i)

fut = 1
for i in range(n, 0, -1):
    if const[i] <= 1:
        print(i, 'год: лучше не инвестировать')
    else:
        fut *= const[i] # инвестируем все деньги в те года, в котор
ые E(r) > 0
        print(i, 'год: нужно инвестировать все деньги')

print('Капитал после инвестирования:', round(fut * C, 4))
```

```
1000
4
3
0.1 0.4 0.5
2 1 0.5
0.4 0.4 0.2
1 0 -1
0.2 0.4 0.4
4 -1 -1
0.6 0.2 0.2
0.8 0.4 0.2
4 год: нужно инвестировать все деньги
3 год: лучше не инвестировать
2 год: нужно инвестировать все деньги
1 год: нужно инвестировать все деньги
Капитал после инвестирования: 3552.0
```

Входные данные:

10000

4

3

0.1 0.4 0.5

2 1 0.5

0.4 0.4 0.2

1 0 -1

0.2 0.4 0.4

4 -1 -1

0.6 0.2 0.2

0.8 0.4 0.2

Упражнение 3.2

Фирма производит квантовые суперкомпьютеры. Цена одного такого компьютера составляет 5 млн долларов. Максимально фирма может производить 3 компьютера в год. Спрос на компьютеры в каждом году определяется вероятностным распределнием ($p(D = 1) = 0.5$, $p(D = 2) = 0.3$, $p(D = 3) = 0.2$). Если компьютер был произведён, но не продан, то на его хранение и содержание тратится по 1 млн долларов каждый год. Если же количество произведённых компьютеров меньше, чем величина спроса на них в этом году, то поставки "недопроизведённых компьютеров" откладываются на следующий год, причём при переносе поставки компания несёт убытки в 2 млн долларов за каждый компьютер. Фирма планирует работать в течение 4-х лет, т. е. после 4-го года заказы приниматься не будут, но фирма будет продолжать выпускать "недопроизведённые" за прошлые года компьютеры. Требуется определить оптимальную стратегию производства, т. е. сколько компьютеров компания должна производить каждый год.

Дано:

- n = 4
- $p(D = 1) = 0.5$, $p(D = 2) = 0.3$, $p(D = 3) = 0.2$
- $c_i \in \{0, 1, 2, 3\}$ - произвели компьютеров за i-ый год
- price = 5, over_fine = 1, lack_fine = 2

Описание алгоритма (версия 1)

Пусть:

$OverCount_i$ - кол-во компьютеров, произведенных в i-ом году и нереализованных в i-ом году (избыток производства).

$LackCount_i$ - кол-во компьютеров, затребованых клиентами, но не произведенных в i-ом году (недостаток производства).

D_i - число компьютеров, которое надо произвести в i-ом году, чтобы идеально удовлетворить спрос (учитывая наши избытки и долги с прошлого года).

$\Delta D_i = \begin{cases} LackCount_i, \text{ если } D_i > c_i \\ 0, \text{ если } D_i = c_i \\ -OverCount_i, \text{ если } D_i < c_i \end{cases} = D_i - c_i$ - разница между "идеальным объёмом производста" и тем, сколько компания произвела.

$prof = \sum_{i=1}^{4} \Delta prof_i$ - итоговая прибыль компании, которую мы хотим максимизировать.

$\Delta prof_i = c_i * price - OverCount_i * \text{over\_fine} - LackCount_i * \text{lack\_fine}$ - прибыль за i-ый год.

Будем решать задачу жадно, т. е. действуя максимально эффективно на каждом шаге. Каждый год будем стараться максимизировать $\Delta prof_i$ - прибыль за этот год. Для этого нужно максимально возможно приблизить c_i к D_i. Но D_i может быть дробным числом, в то время как c_i обязано быть целым. Тогда пусть $f(D_i)$ - ф-ия, которая будет округлять D_i до целого числа, так чтобы $\Delta prof_i$ была максимальна. Также разумно считать, что $D_i = ED + \Delta D_{i-1}$, где ED - мат. ожидание спроса.

Тогда будем поддерживать следующие инварианты:

1) $D_i = ED + \Delta D_{i-1}$

2) $\text{c}_i = \begin{cases} 0, \text{ если } f(D_i) < 0 \\ f(D_i), \text{ если } 0 \le f(D_i) \le 3 \\ 3, \text{ если } f(D_i) > 3 \end{cases}$

3) $\Delta D_i = D_i - c_i$

4) $OverCount_i = max(0, -\Delta D_i)$

5) $LackCount_i = max(0, \Delta D_i)$

6) $\Delta prof_i = c_i * price - OverCount_i * \text{over\_fine} - LackCount_i * \text{lack\_fine}$

Осталось определить ф-ию $f(D_i)$:

$$f(D_i) = \begin{cases} \lfloor D_i \rfloor, \text{ если } \Delta prof_i(c_i = \lfloor D_i \rfloor) > \Delta prof_i(c_i = \lceil D_i \rceil) \\ \lceil D_i \rceil, \text{иначе} \end{cases}$$

Неравенство $\Delta prof_i(c_i = \lfloor D_i \rfloor) > \Delta prof_i(c_i = \lceil D_i \rceil)$ для простоты будем проверять простой подстановкой.

In [10]:
```
def E(p): # ED
    M = 0
    for i in range(len(p)):
        M += i * p[i]
    return round(M, 4)

p = [0, 0.5, 0.3, 0.2]
ED = E(p)
print('Мат ожидание спроса:', ED)
```

```
Мат ожидание спроса: 1.7
```

In [11]:
```
import math

n = 4 # кол-во лет
price = 5
c_min = 0 # минимально возможное число произведен. комп. в год
c_max = 3 # максимально возможное число произведен. комп. в год
over_fine = 1
lack_fine = 2

OverCount = [0] * (n + 2)
LackCount = [0] * (n + 2)
D = [0] * (n + 2)
delta_D = [0] * (n + 2)
c = [0] * (n + 2)
```

In [12]:
```
def delta_prof(i):
    return c[i] * price - OverCount[i] * over_fine - LackCount[i] *
lack_fine


def f(i): # f(D_i)
    c[i] = math.floor(D[i])
    delta_D[i] = D[i] - c[i]
    OverCount[i] = max(0, -delta_D[i])
    LackCount[i] = max(0, delta_D[i])
    Prof1 = delta_prof(i)

    c[i] = math.ceil(D[i])
    delta_D[i] = D[i] - c[i]
    OverCount[i] = max(0, -delta_D[i])
    LackCount[i] = max(0, delta_D[i])
    Prof2 = delta_prof(i)

    if Prof1 > Prof2:
        return math.floor(D[i])
    return math.ceil(D[i])
```

```
def determine_c(i): # определяем c_i по f(D_i)
    if f(i) < 0:
        return 0
    elif f(i) > 3:
        return 3
    else:
        return f(i)

def process():
    prof = 0
    for i in range(1, n + 2): # динамически вычисляем c_i Для всех
i
        if i == n + 1:
            D[i] = delta_D[i - 1]  # в последний год фирма не прини
мает новые заказы
        else:
            D[i] = ED + delta_D[i - 1]
        c[i] = determine_c(i)
        delta_D[i] = round(D[i] - c[i], 4)
        OverCount[i] = max(0, -delta_D[i])
        LackCount[i] = max(0, delta_D[i])
        prof += round(delta_prof(i), 4) # прибавляем прибыль за i-ы
й год
        print('Ожидание спроса за', i, 'год:', D[i])
        print('В', i, 'год производим:', c[i])
        print('Перепроизвели в', i, 'год:', OverCount[i])
        print('Недопроизвели в', i, 'год:', LackCount[i])
        print('Прибыль за', i, 'год:', delta_prof(i))
        print('')

    prof = round(prof, 4) # итоговая прибыль
    print('Итоговая прибыль:', prof)

process()
```

```
Ожидание спроса за 1 год: 1.7
В 1 год производим: 2
Перепроизвели в 1 год: 0.3
Недопроизвели в 1 год: 0
Прибыль за 1 год: 9.7

Ожидание спроса за 2 год: 1.4
В 2 год производим: 2
Перепроизвели в 2 год: 0.6
Недопроизвели в 2 год: 0
Прибыль за 2 год: 9.4

Ожидание спроса за 3 год: 1.1
В 3 год производим: 2
Перепроизвели в 3 год: 0.9
Недопроизвели в 3 год: 0
Прибыль за 3 год: 9.1

Ожидание спроса за 4 год: 0.7999999999999999
В 4 год производим: 1
Перепроизвели в 4 год: 0.2
Недопроизвели в 4 год: 0
Прибыль за 4 год: 4.8

Ожидание спроса за 5 год: -0.2
В 5 год производим: 0
Перепроизвели в 5 год: 0.2
Недопроизвели в 5 год: 0
Прибыль за 5 год: -0.2

Итоговая прибыль: 32.8
```

В математической модели обнаружились неточности. Во-первых, мы дали не совсем правильные определения переменным $OverCount_i$ и $LackCount_i$:

- $OverCount_i$ - кол-во компьютеров, нереализованных компанией к концу i-го года (избыток производства).
- $LackCount_i$ - кол-во компьютеров, которое компания "задолжала" клиентам к концу i-го года (недостаток производства).

Во-вторых, при рассчёте $\Delta prof_i$ при заданном c_i мы пользовались ф-лой:

$$\Delta prof_i = c_i * price - OverCount_i * \text{over\_fine} - LackCount_i * \text{lack\_fine}$$

В ней мы считаем, что продадим ровно c_i компьютеров. Но ведь это не гарантированно так: мы произвели c_i компьютеров, но купить у нас могут и меньше. На самом деле, мы можем ожидать, что у нас купят $min(D_i, c_i)$ компьютеров, т. е. число, равное минимуму из величены спроса на новые компьютеры в этот год и кол-ва произведенных в этот год компьютеров (минимумум из того, сколько новых компьютеров люди ходят купить, и того, сколько их есть у компании). Также, внесём поправку: D_i может быть только больше или равно 0 (т. к. велична спроса не может быть отрицательной).

В-третьих, мы можем найти более простой способ определять, чему равна $f(D_i)$, чем высчитывать значения нескольких параметров для двух случаев и подставлять их в формулу для $\Delta prof_i$.

Пусть $\Delta prof_i(c_i = \lfloor D_i \rfloor) = (*)$, $\Delta prof_i(c_i = \lceil D_i \rceil) = (**)$.

Пусть $m = \lfloor D_i \rfloor$, $q = D_i - \lfloor D_i \rfloor$. $D_i = m + q$

$(*) = min(m + q, m) * price - q * \text{lack\_fine} = m * price - q * \text{lack\_fine}$

$(**) = min(m + q, m + 1) * price - (1 - q) * \text{over\_fine} = (m + q) * price - \text{over\_fine} + q * \text{over\_}$

$(**) - (*) = q * price - \text{over\_fine} + q(\text{over\_fine} - \text{lack\_fine}) = q(price + \text{over\_fine} - \text{lack\_fine})$

Таким образом, чтобы определить $f(D_i)$ достаточно сравнить $q(price + \text{over\_fine} - \text{lack\_fine}) - \text{over\_fine}$ с 0.

Описание алгоритма (версия 2)

1) $D_i = max(0, ED + \Delta D_{i-1})$

2) $c_i = \begin{cases} 0, \text{ если } f(D_i) < 0 \\ f(D_i), \text{ если } 0 \le f(D_i) \le 3 \\ 3, \text{ если } f(D_i) > 3 \end{cases}$

Где $f(D_i) = \begin{cases} \lfloor D_i \rfloor, \text{ если } (D_i - \lfloor D_i \rfloor)(price + \text{over\_fine} - \text{lack\_fine}) - \text{over\_fine} \le 0 \\ \lceil D_i \rceil, \text{иначе} \end{cases}$

3) $\Delta D_i = D_i - c_i$

4) $OverCount_i = max(0, -\Delta D_i)$

5) $LackCount_i = max(0, \Delta D_i)$

6) $\Delta prof_i = min(D_i, c_i) * price - OverCount_i * \text{over\_fine} - LackCount_i * \text{lack\_fine}$

In [13]:
```
def delta_prof(i):
    return min(D[i], c[i]) * price - OverCount[i] * over_fine - LackCount[i] * lack_fine


def f(i): # f(D_i)
    q = D[i] - math.floor(D[i])
    if q * (price + over_fine - lack_fine) <= over_fine:
        return math.floor(D[i])
    return math.ceil(D[i])


def determine_c(i): # определяем c_i по f(D_i)
    if f(i) < 0:
        return 0
    elif f(i) > 3:
        return 3
    else:
        return f(i)

def process():
    prof = 0
    for i in range(1, n + 2): # динамически вычисляем c_i Для всех i
        if i == n + 1:
            D[i] = delta_D[i - 1]  # в последний год фирма не принимает новые заказы
        else:
            D[i] = max(0, ED + delta_D[i - 1])
        c[i] = determine_c(i)
        delta_D[i] = round(D[i] - c[i], 4)
        OverCount[i] = max(0, -delta_D[i])
        LackCount[i] = max(0, delta_D[i])
        prof += round(delta_prof(i), 4) # прибавляем прибыль за i-й год
        print('Ожидание спроса за', i, 'год:', D[i])
        print('В', i, 'год производим:', c[i])
        print('Перепроизвели в', i, 'год:', OverCount[i])
        print('Недопроизвели в', i, 'год:', LackCount[i])
        print('Прибыль за', i, 'год:', delta_prof(i))
        print('')

    prof = round(prof, 4) # итоговая прибыль
    print('Итоговая прибыль:', prof)

process()
```

```
Ожидание спроса за 1 год: 1.7
В 1 год производим: 2
Перепроизвели в 1 год: 0.3
Недопроизвели в 1 год: 0
Прибыль за 1 год: 8.2

Ожидание спроса за 2 год: 1.4
В 2 год производим: 2
Перепроизвели в 2 год: 0.6
Недопроизвели в 2 год: 0
Прибыль за 2 год: 6.4

Ожидание спроса за 3 год: 1.1
В 3 год производим: 1
Перепроизвели в 3 год: 0
Недопроизвели в 3 год: 0.1
Прибыль за 3 год: 4.8

Ожидание спроса за 4 год: 1.8
В 4 год производим: 2
Перепроизвели в 4 год: 0.2
Недопроизвели в 4 год: 0
Прибыль за 4 год: 8.8

Ожидание спроса за 5 год: -0.2
В 5 год производим: 0
Перепроизвели в 5 год: 0.2
Недопроизвели в 5 год: 0
Прибыль за 5 год: -1.2

Итоговая прибыль: 27.0
```

С поправками получился совсем другой результат!

4. Коварные банковские вклады

Пусть на нашем счёте в банке B лежит сумма S_0 > 1000. Через 2 года нам нужно будеть снять с этого счёта все имеющиеся к тому моменту средства. Если мы оставим деньги в банке B наш вклад увеличться на 7% после 1-го года и на 5% - после 2-го. Существует также другой банк - банк A. В начале первого, в конце первого и в конце второго годов можно перемещать **все** деньги из банка B в банк A и наоборот. При этом перевод из B в A бесплатен, а из A в B стоит 10. В конце второго года все деньги в любом случае должны оказаться в банке B. За каждый год хранения денег в банке A взимается плата в размере 20 (плата взимается после наростания процентов). В конце первого года, если деньги лежат в банке A, их сумма увеличивается на 8% или 12% (каждое из этих событий может произойти с равной вероятностью). Аналогично, в конце второго года, если деньги лежат в банке A, их сумма увеличивается на 5% или 9%. Требуется разработать оптимальную стратегию хранения и перевода денег, чтобы S_3 - сумма денег на счёте в банке B к концу второго года была максимально возможной. Условие иллюстрирует рисунок:

Image 3

Пусть 0-й этап - момент до начала первого года, 1-й этап - конец первого года и начало второго, 2-й этап - конец второго года, 3-й этап - финальная стадия, когда все деньги переведы в банк B. Пусть S_i - сумма денег на i-м этапе. Пусть $f_i(x_1, S_i, x_2)$ - значение S_3, если на i-м этапе мы имели S_i в банке x_1 и переместили все эти деньги в банк x_2, и если считать что в дальнейшем мы будем действовать максимально. Например, если $f_1(A, S_1, B) = S'$, это значит, что если на этапе 1 в банке A лежит S_1, и затем мы перемещаем эти деньги в B, то в конце инвестирования мы окажеся с $S_3 = S'$ (если в дальнейшем будем действовать наилучшим возможным образом). И последнее обозначение: максимизируя результаты всех возможных решений, находим максимально возможный итог $f_i^*(x, S_i)$. Например, $f_1^*(A, S_1) = \max\limits_{x_1 \in \{A, B\}} f_1(A, S_1, x_1)$. Пусть мы находимся на этапе 2. Значит, нам осталось только перевести все деньги в банк B, если они ещё не там. Тогда:

$$f_2^*(A, S_2) = S_2 - 10$$
$$f_2^*(B, S_2) = S_2$$

Этап 1. Пусть деньги лежат в А, их сумма равна S_1. Если мы оставим деньги в А ещё на год, то с вероятностью 0.5 получим $S_2 = S_1 \times 1.05 - 20$ или (с такой же вроятностью) $S_2 = S_1 \times 1.09 - 20$. Если же переведём их в банк В, то через год получим $S_2 = (S_1 - 10) \times 1.05$. Тогда:

$$f_1(A, S_1, A) = 0.5 f_2^*(A, S_1 \times 1.05 - 20) + 0.5 f_2^*(A, S_1 \times 1.09 - 20)$$

$$f_1(A, S_1, B) = f_2^*(B, (S_1 - 10) \times 1.05)$$

И следовательно,

$$f_1^*(A, S_1) = \max(f_1(A, S_1, A), f_1(A, S_1, B)) = \max(0.5 f_2^*(A, S_1 \times 1.05 - 20) + 0.5 f_2^*(A, S_1 \times$$

А как считать $f_2^*(A, S_2)$ мы уже знаем. Аналогично можно вычислить и $f_1^*(B, S_1)$. Заметим также, что на выходе функций f и $f*$ мы всегда будем получать линейные выражения от S_i. Утверждение легко доказывается по индукции: база - для этапа 2 мы получили, что f_2^* - имеет вид $\alpha S_2 + \beta$. Теперь переход. Нетрудно заметить, что мы получаем f_i, f_i^* из f_{i+1}, f_{i+1}^*, используя только операции сложения двух формул, сложение ф-лы с константой и умножение ф-лы на константу. При этом

$$(\alpha_1 S + \beta_1) + (\alpha_2 S + \beta_2) = (\alpha_1 + \alpha_2)S + (\beta_1 + \beta_2)$$

$$(\alpha_1 S + \beta_1) + \beta_2 = \alpha_1 S + (\beta_1 + \beta_2)$$

$$\alpha_2(\alpha_1 S + \beta_1) = \alpha_1\alpha_2 S + \alpha_2\beta_1$$

То есть все эти операции, применённые к линейным выражениям дают на выходе линейное выражение. ч. т. д. Теперь можем доказать, что замена всех стохастических процентных ставок на их мат. ожидания может быть осуществлена и корректность модели при этом сохранится. Пусть $k_1, k_2; p_1, p_2$ - возможные коэффициенты увелечения вклада и вероятности, с которыми эти события могут произойти, соответственно. Мы хотим найти $g(S) = \alpha k S + \beta$, где k задаётся вышеуказанным распределением $k_1, k_2; p_1, p_2$. В рамках модели замены случайных величин их мат. ожиданями, в которой мы сейчас работаем, следует считать (при нахождении ф-лы для $f_1(A, S_1, A)$ мы так и считали), что $g(S) = p_1(\alpha k_1 S + \beta) + p_2(\alpha k_2 S + \beta)$. Тогда:

$$p_1(\alpha k_1 S + \beta) + p_2(\alpha k_2 S + \beta) = \alpha p_1 k_1 S + p_1\beta + \alpha p_2 k_2 S + p_2\beta = \alpha(p_1 k_1 + p_2 k_2)S + (p_1 + p_2)\beta$$

Ek - мат. ожидание k. Итак, мы доказали что можем заменять стохастические процентные ставки их мат. ожиданиями.

Таким образом, двигаясь от 2-го этапа к 0-му, вычисляя по очереди f и f^* для данного этапа можем прийти к $f_0^*(B, S_0)$ - максимальной ожидаемой сумме, к которой можем придти в конце, начав с суммой S_0 в В, т. е. к ответу на задачу. Ожидается, что алгоритм будет работать за линйное время, ведь на каждом этапе мы будем максимизировать известные функции от известных величин, а значит, каждый этап будет обрабатываться за константу. Потом останется только восстановить решения, которые мы принимали, чтобы достичь найденного результата. Очевидно, что зная $f_i(x_1, S_i, x_2)$ для всех возможных i, x_1 и x_2, мы без труда сможем это сделать за линейное время (начав с 0-го этапа и каждый раз сравнивая $f_i(A, S_i, A)$ и $f_i(A, S_i, \mathrm{B})$ или $f_i(\mathrm{B}, S_i, A)$ и $f_i(\mathrm{B}, S_i, \mathrm{B})$ будем понимать, в каком банке выгодней оставить деньги на этот год). Таким образом, намечается классический сюжет стохастического динамического программирования. Конечно, т. к. кол-во лет инвестирования небольшое, задачу можно решить и аналитически, не прибегая к программированию, но мы хотим научиться решать аналогичные задачи и при больших n, поэтому запрогограммировать данную задачу очень даже релевантно.

In [ ]:
```
class Bank:
    def __init__(self, rates, p, price=0, charge_out=0):
        self.rates = [0] * len(rates)  # процентные ставки
        for i in range(len(rates)):
            for j in range(len(rates[i])):
                self.rates[i] += p[i][j] * rates[i][j]
        self.price = price  # цена за хранение денег
        self.charge_out = charge_out  # штраф за снятие денег

n = 3
A = Bank([[1.08, 1.12], [1.05, 1.09]], [[0.5, 0.5], [0.5, 0.5]], 20
, 10)
B = Bank([[1.07], [1.05]], [[1], [1]])
S0 = int(input())
f = {}
f_star = {} # f*
... # база
for i in range(n - 2, -1, -1):
    f[i, A, S, A] = f_star[i + 1, S * A.rates[i] - A.price, A]
    f[i, A, S, B]  = f_star[i + 1, (S - A.charge_out) * B.rates[i]
- B.price, B]
    f_star[i, A, S] = max(f[i, A, S, A], f[i, A, S, B])
    f[i, B, S, A] = f_star[i + 1, (S - B.charge_out) * A.rates[i] -
A.price, A]
    f[i, B, S, B] = f_star[i + 1, S * B.rates[i] - B.price, B]
    f_star[i, B, S] = max(f[i, B, A], f[i, B, B])
```

Но неожиданно мы сталкиваемся с большой проблемой! При теоритических рассуждениях, двигаясь от конца к началу, мы работали с формулами, линейными выражениями от S_i, но ведь компьютеру надо работать с конкретными числами, а подставить на i-м шаге в формулу S_i мы не можем, попросту потому что мы его ещё не знаем! Действительно, в начале работы программы мы знаем только S_0, а этапы начинаем перебирать с конца. Поразмыслив, я пришёл к выводу, что для решения этой проблемы можно применить следующий приём: создадим класс LinearFormula, в котором будет хранить α и β - коэффициенты линейного выражения от $\alpha S + \beta$. Как мы уже доказали выше, для всех возможных i f и f^* являются линейными выражениями от S_i, следовательно, с помощью класса LinearFormula мы сможем хранить все значения f и f^*. Казалось бы, проблема решена, но на самом деле самое трудное ещё впереди. Проблемы начинаются, когда мы максимизируем f^*, т. е. когда нам надо выбрать наибольшее из линейных выражений. Рассмотрим n - 2 этап: тут нам надо сравнить 2 линейных выражения и выбрать из них наибольшее.

$$\alpha_1 S + \beta_1 \vee \alpha_2 S + \beta_2$$
$$(\alpha_1 - \alpha_2) S \vee \beta_2 - \beta_1$$
$$S \vee \frac{\beta_2 - \beta_1}{\alpha_1 - \alpha_2} \text{ (если } \alpha_1 \neq \alpha_2)$$

Если $\alpha_1 = \alpha_2 = \alpha$, то искомый максимум - $\alpha S + \max(\beta_1, \beta_2)$. Если же $\alpha_1 \neq \alpha_2$, то максимумом является та или иная формула в зависимости от того, больше или меньше S ключевой точки $\frac{\beta_2 - \beta_1}{\alpha_1 - \alpha_2}$. Следовательно, нужно будет запомнить обе эти ф-лы и ключевую точку, и когда будет приходить реальное значение S, мы сможем однозначно определять искомое максимальное значение. Выглядит труднореализуемо: нам надо хранить или один объект типа LinerFormula, или 2 объекта типа LinearFormula и число. Но это только начало, ведь на следующих этапах нам надо будет выбирать наибольшую из формул, которые сами в зависимости от значения S могут иметь разные коэффициенты, и вот это уже настоящий кошмар программиста. В итоге рабочее решение будет выглядеть так:

In [6]:
```
class Bank:
    def __init__(self, rates, p, price=0, charge_out=0):
        self.rates = [0] * len(rates)  # процентные ставки
        for i in range(len(rates)):
            for j in range(len(rates[i])):
                self.rates[i] += round(p[i][j] * rates[i][j], 3)
        self.price = price  # цена за хранение денег
        self.charge_out = charge_out  # штраф за снятие денег


class LinearFormula: # линейное выражение от S
    def __init__(self, alpha=0, beta=0, account=''): # LinearFormul
a = self.alpha * S + self.beta
        self.alpha = alpha
        self.beta = beta
        self.account = account

    def add(self, const): # сложение с константой
        res = LinearFormula()
        res.alpha = self.alpha
```

```
        res.beta = round(self.beta + const, 3)
        return res

    def mul(self, const): # умножение на константу
        res = LinearFormula()
        res.alpha = round(const * self.alpha, 3)
        res.beta = round(const * self.beta, 3)
        return res

    def do(self, S): # вычисление значения выражения по заданному S
        return round(self.alpha * S + self.beta, 3)

    def make_str(self):
        alpha_coef = ''
        if (self.alpha != 1):
            alpha_coef = str(self.alpha)

        sign_and_beta = ''
        if (self.beta > 0):
            sign_and_beta = ' + ' + str(self.beta)
        elif (self.beta < 0):
            sign_and_beta = ' - ' + str(abs(self.beta))

        s = alpha_coef + 'S' + sign_and_beta
        return s

    def write(self): # вывод линейного выражения на экран
        print(self.make_str(), end=' ')


class MultiLinearFormula: # совокупность линейных выражений
    # (в зависимости от значения S коэффициенты alpha и beta приним
ают разные значения)
    def __init__(self, alpha=0, beta=0):
        self.lines = []
        self.points = []
        if (alpha != 0):
            l = LinearFormula(alpha, beta)
            self.lines.append(l)

    def write(self): # вывод мультиформулы на экран
        print('Printing MLformula')
        print(*self.points)
        for line in self.lines:
            line.write()
        print('Printing finished')


def max_of_lines(l1, l2): # находим P - точку пересечения двух прям
ых
    # и определяем max(l1, l2) при S <= P и при S > P
    if (l1.alpha == l2.alpha):
        if (l1.beta >= l2.beta):
```

```
            return (l1, 1000, l1)
        return (l2, 1000, l2)
    elif (l1.alpha > l2.alpha):
        P = round((l2.beta - l1.beta) / (l1.alpha - l2.alpha), 3)
        return (l2, P, l1)
    else:
        P = round((l2.beta - l1.beta) / (l1.alpha - l2.alpha), 3)
        return (l1, P, l2)
    # возвращаем max(l1, l2 | {S <= P}), P, max(l1, l2 | {S > P})


def momlf(ml1, ml2): # max of MultiLinearFormulas ml1, ml2
    # т. е. находим такую MultiLinearFormula, которая
    # для каждого возможного S является максимумом из ml1 и ml2
    new_ml = MultiLinearFormula()
    i = 0
    j = 0
    R = 1000
    while (True): # заполняем new_ml.lines и new_ml.points,
        # максимизирую (выбирая ml1 или ml2) на каждом из отрезков,
        # на которые точки из ml1.points и ml2.points разбивают коо
рдинатную прямую S
        L = R
        l1 = ml1.lines[i]
        l2 = ml2.lines[j]
        if (i == len(ml1.points) and j == len(ml2.points)):
            l1, P, l2 = max_of_lines(l1, l2)
            if (P <= L):
                new_ml.lines.append(l2)
            else:
                new_ml.points.append(P)
                new_ml.lines.append(l1)
                new_ml.lines.append(l2)
            break
        elif (i == len(ml1.points)):
            R = ml2.points[j]
            j += 1
        elif (j == len(ml2.points)):
            R = ml1.points[i]
            i += 1
        elif (ml1.points[i] <= ml2.points[j]):
            R = ml1.points[i]
            i += 1
        else:
            R = ml2.points[j]
            j += 1

        if (L == R):
            continue

        l1, P, l2 = max_of_lines(l1, l2)
        if (P <= L):
            new_ml.points.append(R)
```

```
                new_ml.lines.append(l2)
            elif (P >= R):
                new_ml.points.append(R)
                new_ml.lines.append(l1)
            else:
                new_ml.points.append(P)
                new_ml.lines.append(l1)
                new_ml.points.append(R)
                new_ml.lines.append(l2)

    return new_ml


def put_fotmula_in_formula(l_in, l_ex): # подставляем l_in в качест
ве S в l_ex
    return (l_in.mul(l_ex.alpha)).add(l_ex.beta)

def binary_search(points, S): # ищем минимально возможное P, больше
е S
    l = -1
    r = len(points)
    while (r - l > 1):
        m = (l + r) // 2
        if (S <= points[m]): # S <= P
            r = m
        else:
            l = r

    return r

def do_formula(ml, S): # вычислить значение ml, подставив S
    i = binary_search(ml.points, S)
    return ml.lines[i].do(S)

def find_move(ml, S): # в какой банк надо положить сумму S,
    # чтобы ml приняло оптимальное значение
    i = binary_search(ml.points, S)
    return ml.lines[i].account

def put_formula_in_multiformula(l_in, l_ex, account): # подставить
ф-лу в качестве S в
    # мультиформулу
    res = MultiLinearFormula()
    for P in l_ex.points:
        res.points.append(round((P - l_in.beta) / l_in.alpha, 3))
    for l in l_ex.lines:
        new_line = put_fotmula_in_formula(l_in, l)
        new_line.account = account
        res.lines.append(new_line)
    return res


n = 3
```

```
A = Bank([[1.08, 1.12], [1.05, 1.09]], [[0.5, 0.5], [0.5, 0.5]], 20
, 10)
B = Bank([[1.07], [1.05]], [[1], [1]])
f = {}
f_star = {} # f*
best_move = {}
f_star[n - 1, A] = MultiLinearFormula(1, -A.charge_out) # f*(n - 1,
A) = S - -A.charge_out
f_star[n - 1, B] = MultiLinearFormula(1, 0) # f*(n - 1, B) = S
for i in range(n - 2, -1, -1):
    f[i, A, A] = put_formula_in_multiformula(LinearFormula(A.rates[
i], -A.price), f_star[
        i + 1, A], 'A')  # хотим в выражение f_star[i + 1, A]
    # подставить выражение S * A.rates[i] - A.price
    # используем write для вывода промежуточных формул и точек
    f[i, A, A].write()
    f[i, A, B] = put_formula_in_multiformula(LinearFormula(B.rates[
i],
        round(-A.charge_out * B.rates[i] - B.price, 3)), f_star[i +
1, B], 'B')
    # хотим в выражение f_star[i + 1, B]
    # подставить выражение (S - A.charge_out) * B.rates[i] - B.pric
e
    f[i, A, B].write()
    f_star[i, A] = momlf(f[i, A, A], f[i, A, B])
    f_star[i, A].write()
    f[i, B, A] = put_formula_in_multiformula(LinearFormula(A.rates[
i],
        round(B.charge_out * A.rates[i] - A.price, 3)), f_star[i +
1, A], 'A')
    # хотим в выражение f_star[i + 1, A]
    # подставить выражение (S - B.charge_out) * A.rates[i] - A.pric
e
    f[i, B, A].write()
    f[i, B, B] = put_formula_in_multiformula(LinearFormula(B.rates[
i], -B.price), f_star[
        i + 1, B], 'B') # хотим в выражение f_star[i + 1, B]
    # подставить выражение  S * B.rates[i] - B.price
    f[i, B, B].write()
    f_star[i, B] = momlf(f[i, B, A], f[i, B, B])
    f_star[i, B].write()

print('')
print('Введите S0')
S = int(input()) # считываем значение S0
S3 = do_formula(f_star[0, B], S) # находим оптимальное S3
print('S3 =', S3)
print('')
acc = B
for i in range(n - 1): # восстанавливаем последовательность действи
й,
    # которые привели нас к оптимальному ответу
    move = find_move(f_star[i, acc], S)
```

```
    print('этап', i, '- кладём в банк', move)
    if (move == 'A'):
        acc = A
    else:
        acc = B
```

```
Printing MLformula

1.07S - 30 Printing finished
Printing MLformula

1.05S - 10.5 Printing finished
Printing MLformula

1.07S - 30 Printing finished
Printing MLformula

1.07S - 30.0 Printing finished
Printing MLformula

1.05S Printing finished
Printing MLformula
1500.0
1.05S 1.07S - 30.0 Printing finished
Printing MLformula

1.177S - 51.4 Printing finished
Printing MLformula
1411.869
1.124S - 11.235 1.145S - 41.449 Printing finished
Printing MLformula
1411.869
1.177S - 51.4 1.177S - 51.4 Printing finished
Printing MLformula

1.177S - 51.4 Printing finished
Printing MLformula
1401.869
1.124S 1.145S - 30.0 Printing finished
Printing MLformula
1401.869
1.177S - 51.4 1.177S - 51.4 Printing finished

Введите S0
1010
S3 = 1137.37

этап 0 - кладём в банк A
этап 1 - кладём в банк A
```

Решение получилось сложным и громоздким. Заметим, что если бы мы использовали рекурсию, а не динамическое программирование, то таких проблем бы не возникло: мы бы шли от нулевого этапа до (n - 1)-го, а не наоборот, и поэтому всегда знали бы численное значение S, с которым сейчас работаем. Таким образом, мы обнаружили недостаток динамичесого программмрования перед рекурсией. Вывод: иногда задачу рациональней решать рекурсией, а не динамическим программирвоанием.

Модель II. Максимизация вероятности достижения цели

Пришло время обратиться ко второй модели стохастического динамического программирования, рассматриваемой в моей работе - максимизации вероятности достижения цели. До этого мы заменяли вероятностную неопределённость математическим ожиданием (на каждом шаге динамического программирования вместо величины, заданной её вероятностным распределением, мы подставляли её математическое ожидание). Но алгоритмы с использованием мат. ожиданий ориентируются на средние значения, и, по большому счёту, не дают никаких гарантий. Дело в том, что мы не учитываем, наксколько и с какой вероятностью реальные значения могут отклоняться от прогнозируемых. Например, пусть X - такая случайная величина, что: $\begin{cases} P(X = 1) = 0.99 \\ P(X = 1000) = 0.01 \end{cases}$. Тогда $EX = 0.99 \cdot 1 + 0.01 \cdot 1000 = 10.99$, и именно на это значение мы будем "рассчитывать" в программах, написанных по первой модели. Но при этом в 99% случаев X = 1. Таким образом, есть потребность в методе, дающем большие гарантии.

Все рассматриваемые нами задачи сводятся к максимизации (или минимизации) значения некоторой переменной. Пусть требуется максимизировать X. Выберем X_{inf} - наименьшее удовлетворяющее нас значение X и будем стараться максимизировать $P(X \geq X_{inf})$ - вероятность того, что X не меньше, чем X_{inf}. В особых случаях (в задачах с повышенными требованиями к "надёжности" решения) можно даже потребовать, чтобы $P(X \geq X_{inf})$ была не меньше некоторого значения P_{wish}, и тогда алгоритм должен будет искать стратегию для достижения $P(X \geq X_{inf}) \geq P_{wish}$. Аналогично в задаче минимизации будем максимизировать $P(X \leq X_{sup})$, где X_{sup} - наибольшее удовлетворяющее нас значение X. Эта модель и называется ***максимизация вероятности достижения цели***. Перейдём к конкретному примеру.

5. Инвестирование продолжается

Продолжают действовать все обозначения из параграфа "2. Задача инвестирования". Пусть целью является максимизация вероятности того, что по истечении n лет инвестирования сумма накопленных денег составит S. Пусть теперь $f_i(x_i)$ - вероятность накопления суммы S к концу n-го года при условии, что в начале i-го года капитал равен x_i.

Найдём рекурентное уравнение динамического программирования

Рассмотрим $f_n(x_n)$, считая, что уже найдено оптимальное y_n. Мы знаем, что $f_n(x_n) = P(x_n + ry_n \geq S)$, где $r \in \{r_1, r_2, \ldots, r_m\}$

Пусть Ω - множество всех элементарных исходов вероятностного пр-ва. По условию задачи $\Omega = \{r = r_1\} \sqcup \{r = r_2\} \sqcup \ldots \sqcup \{r = r_m\}$ (используем $\sqcup$ для обозначения объединения непересекающихся подмножеств). Тогда можем воспользоваться формулой полной вероятности:

$$f_n(x_n) = \sum_{k=1}^{m} p_k P\left(x_n + ry_n \geq S | r = r_k\right) = \sum_{k=1}^{m} p_k P(x_n + r_k y_n \geq S)$$

Теперь вспоминаем, что y_n может принимать любые значения от 0 до x_n. Так как мы стремимся максимизировать $f_n(x_n)$, то логично взять:

$$f_n(x_n) = \max_{0 \leq y_n \leq x_n} \left\{ \sum_{k=1}^{m} p_k P(x_n + r_k y_n \geq S) \right\}$$

Заметим, что при фиксированном y_n: $f_{n-1}(x_{n-1}) = \sum_{k=1}^{m} p_k f_n(x_{n-1} + r_k y_{n-1})$ (снова исп. формулу полной вероятности, на этот раз в качестве условной вероятности выступает $f_n(x_{n-1} + ry_{n-1} | r = r_k)$). Следовательно,

$$f_{n-1}(x_{n-1}) = \max_{0 \leq y_{n-1} \leq x_{n-1}} \left\{ \sum_{k=1}^{m} p_k f_n(x_{n-1} + r_k y_{n-1}) \right\}.$$

Применяя аналогичные рассуждения (действуя по индукции) для f_{n-2}, ..., f_1, получаем рекурентное уравнение динамического программирования:

$$f_n(x_n) = \max_{0 \leq y_n \leq x_n} \left\{ \sum_{k=1}^{m} p_k P(x_n + r_k y_n \geq S) \right\},$$

$$f_i(x_i) = \max_{0 \leq y_n \leq x_n} \left\{ \sum_{k=1}^{m} p_k f_{i+1}(x_i + r_k y_i) \right\}, \text{ i = 1, 2,..., n - 1}$$

Итак, некоторая мат. модель получена. Попробуем запрограммировать простой пример.

Пример 5.1

Человек планирует инвестировать С = 2000 долларов. Инвестировать планируется в течение 3-х лет. Каждый год есть 30%-ая вероятность удвоить вложенные деньги и 70%-ая вероятность все их потерять. Акции продаются в конце года, а в начале следующего года все деньги или любая их часть снова инвестируются. Человек хочет максимизировать вероятность достижения суммы S = 4000 долларов в конце третьего года.

Дано:

- $x_1 = C = 2000$
- n = 3, m = 2
- $p_1 = 0.3, p_2 = 0.7$
- $r_1 = 1, r_2 = -1$
- S = 4000

Наивное решение:

Надо признать, что математическая модель "максимизация вероятности достижения цели" получилась, по крайней мере на первый взгляд, весьма непростой. Как всегда делается в случае, когда непонятно с чего начать, начнём с самого наивного решения. Напишем рекурсивную функцию f(i, x), которая будет вычислять $f_i(x_i)$. Внутри f(i, x) будем пробегаться по всем y из отрезка [0, x] с заданным шагом stp. Для простоты будем считать, что можно инвестировать только целое число долларов, кратное 10. Тогда stp = 10. Нетрудно показать, что ϵ - погрешность при вычислении итогового капитала не превышает $stp \cdot \max_k (1 + r_k)^n$. В нашем случае $\epsilon \leq 10 \cdot 2^3 = 80\$$. Эта погрешность мала в сравнении с x_1 и S, поэтому ей вполне можно пренебречь и наше допущение не искажает задачу.

Итак, внешним циклом будем пробегаться по y из отрезка [0, x] с шагом stp, а внутренним - по всем k из [1, m]. Для каждого k будем вычислять $f(i + 1, x + r_k y)$ (если i = n + 1, то $f(i, x) = Ind\{x \geq S\}$), тогда после внутреннего цикла сможем найти $f_i(x_i)$ для заданнаго y. Максимизировав по y (во внешнем цикле), найдём искомое $f_i(x_i)$. Тогда $f_1(x_1 = C)$ будет являться ответом на задачу.

In [7]:
```
C = 2000
n = 3
m = 2
p = [0.3, 0.7] # вероятности событий
r = [1, -1] # процентные ставки при этихсобытиях
S = 4000
```

In [9]:
```
stp = 10
flag = False # блокируем вывод промежуточных результатов
def f(i, x):
    if i == n + 1: # база
        return (x >= S) # по факту значение индикаторной величины

    max_res = -1 # наилучший результат
    best_y = -1 # y при котором он достигается
    for y in range(0, x + 1, stp): # перебор всех возможных вариант
ов
        res = 0
        for k in range(m):
            res += p[k] * f(i + 1, x + r[k] * y)
        if res > max_res:
            max_res = res
            best_y = y

    if flag:
        # вывод конкретной инструкции для конкретного года при конк
ретном значении x
        print(i, 'год: если x =', x, 'то инвестируем', best_y, 'дол
ларов')
    return max_res


max_P = f(1, C)
print('Максимальная вероятность достижения суммы S составляет', max
_P)
```

```
Максимальная вероятность достижения суммы S составляет 0.3
```

Чтобы более подробно посмотреть на работу алгоритма, возьмём stp = 1000 (можно инвестировать только суммы кратные 1000) и разрешим вывод промежуточных результатов.

In [14]:
```
stp = 1000
flag = True
max_P = f(1, C)
print('')
print('Максимальная вероятность достижения суммы S составляет', max
_P)
```

```
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 2000 то инвестируем 0 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 2000 то инвестируем 0 долларов
3 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 5000 то инвестируем 0 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 6000 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 1000 то инвестируем 1000 долларов
3 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 5000 то инвестируем 0 долларов
3 год: если x = 3000 то инвестируем 1000 долларов
3 год: если x = 6000 то инвестируем 0 долларов
3 год: если x = 2000 то инвестируем 2000 долларов
3 год: если x = 7000 то инвестируем 0 долларов
3 год: если x = 1000 то инвестируем 0 долларов
3 год: если x = 8000 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 4000 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
3 год: если x = 0 то инвестируем 0 долларов
2 год: если x = 0 то инвестируем 0 долларов
1 год: если x = 2000 то инвестируем 0 долларов

Максимальная вероятность достижения суммы S составляет 0.3
```

Составить стратегию оптимального инвестирования можно или методам внимательного взгляда на вывод программы или добавив в программу запоминание посчитанных f(i, x), и потом восстнаваливать последовательность решений, как мы делали в предыдущих задачах. Оценим теперь сложность получившегося алгоритма. Внешний цикл внутри f $\frac{x+1}{stp}$ раз вызывает внутренний. Т. к. при каждом вызове f x - разный, оценим полученное выражение свурху. Заметим, что любое значение x, возникающее в задаче, не может быть больше, чем $(1+\max_k r_k)^n C$. Следовательно, $\frac{x+1}{stp} \leq \frac{(1+\max_k r_k)^n C+1}{stp}$. Внутренний же цикл m раз рекурсивно вызывает f. Рекурсия останавливается, когда глубина (соответсвующая году инвестирования) становится равна n. Таким образом, дерево рекурсии имеет высоту n и на каждом уровне кол-во вершин в $\frac{x+1}{stp} \cdot m$ раз больше, чем на предыдущем. Тогда алгоритм имеет сложность:

$$1+\left(\frac{(1+\max_k r_k)^n C+1}{stp} \cdot m\right)+\ldots+\left(\frac{(1+\max_k r_k)^n C+1}{stp} \cdot m\right)^n = O\left(\left(\frac{(1+\max_k r_k)^n C+1}{stp} \cdot m\right)^n\right).$$

Таким образом, сложность, очевидно, превышает $O(m^n)$.