Разбор задачи «Big Money»

В первой подзадаче можно перебрать, сколько кладем по первому предложению, а сколько по второму:

```
for i : 0 .. r_1
  if i = 0 or i >= l_1
    for j : 0 .. r_2
      if (j = 0 or j >= l_2) and i + j <= m
        res = max(res, i * (1 + p_1) + j * (1 + p_2))
print(res)
```

Во второй подзадаче можно заметить, что если мы еще можем положить на какой-то из двух вкладов, то при наличии денег это надо сделать, поэтому перебрав, сколько положим в первый вклад, можно понять, сколько положим во второй:

```
for i : 0 .. r_1
  if i = 0 or i >= l_1
    j = min(m - i, r_2)
    if j < l_2
      j = 0
    res = max(res, i * (1 + p_1) + j * (1 + p_2))
print(res)
```

В третьей подзадаче нет ограничения снизу на размер вклада, поэтому надо положить сначала максимум денег на вклад с большим процентом, и если остались еще деньги, то положить их на другой из вкладов.

В четвертой подзадаче можно разобрать несколько случаев, на каждый вклад можно положить фиксированное число денег: либо положить в оба вклада, либо ровно в один, либо ни в какой.

Полное решение: если мы положили x денег в первый вклад и y денег во второй, и первый вклад выгоднее (то есть у него больше процент), то при возможности (если мы не нарушим условия вкладов), можно d денег переложить из второго вклада в первый, получив $x + d$ в первом, и $y - d$ во втором. Из этого можем показать, что для одного из вкладов верно следующее: в одном из оптимальных ответов мы в него положим либо 0 денег, либо l_i денег, либо r_i денег. Поэтому мы попробуем решить дважды задачу, выбрав какой из вкладов первый. Попытаемся в него положить либо 0, либо l_i, либо r_i, вычислим как в решении второй подзадачи сколько денег положим во второй из вкладов, и возьмем среди всех шести вариантов наилучший ответ.

```
solve(l_1, r_1, l_2, r_2, m)
  for i : [0, l_1, r_1]
    j = min(m - i, r_2)
    if j < l_2
      j = 0
    res = max(res, i * (1 + p_1) + j * (1 + p_2))

solve(l_1, r_1, l_2, r_2, m)
solve(l_2, r_2, l_1, r_1, m)
```

Разбор задачи «Cake Tasting»

Нам заданы размеры объединений множеств, нужно вывести сами множества. Если мы все множества S_i заменим на их дополнения $T_i = \overline{S_i}$. Тогда $b_A = N - a_A$ — размер пересечения тех T_i, где

$i \in A$. (При этом N — сколько всего элементов мы рассматриваем, можно считать, что $N = a_{2^n-1}$). Тут действует закон $\overline{\cup S_i} = \cap\overline{S_i} = \cap T_i$.

То есть мы свелись к такой же задаче, но про пересечения множеств. Рассмотрим теперь некоторый элемент α. Он входит в какие-то множества $A_\alpha = \{i_1, i_2, \ldots, i_k\}$. Получается, что для любого множества $B \subset A_\alpha$, в b_B учитывается элемент α, а для любого множества $B \not\subset A_\alpha$, в b_B не учитывается элемент α.

Назовем c_A — сколько элементов входят в множества с индексами из A. Тогда $b_A = \sum\limits_{A \subset B} c_B$. Тогда легко вычислить как $b \to c$, так и $c \to b$. Вычислить $c \to b$ легко по определению:

```
for i : 0 .. 2^n - 1
  for j : i .. 2^n - 1
    if (i and j) = i
      b[i] += c[j]
```

А можно и сделать прямо из массива c:

```
b = clone(c)
for i : 0 .. 2^n - 1
  for j : i + 1 .. 2^n - 1
    if (i and j) = i
      b[i] += b[j]
```

Тогда можно и обратно:

```
c = clone(b)
for i : 2^n - 1 .. 0
  for j : i + 1 .. 2^n - 1
    if (i and j) = i
      c[i] -= c[j]
```

Что доказывает, что если все c неотрицательны, то существуют множества и они ровно так и задаются. Для каждого A нужно создать c_A элементов, которые лежат только в множествах с индексами из A.

Это работает за $\mathcal{O}(2^{2n})$, так можно решить первые три подзадачи.

Чтобы решить четвертую, нужно заметить, что внутренний цикл должен перебирать не все множества, а только те, которые являются надмножеством i. Решение будет работать за $\mathcal{O}(3^n)$, более подробно про этот метод можно почитать по ссылке `http://e-maxx.ru/algo/all_submasks`.

Чтобы решить последнюю подзадачу, нужно эту операцию выполнить за $\mathcal{O}(2^n \cdot n)$. Для этого надо посмотреть на операцию $(c \to b)$ $b_A = \sum\limits_{A \subset B} c_B$ внимательнее. Изначально был массив c в конце получился массив b, в массиве b сумма по всем надмножествам, то есть каждый бит в A, в котором стоит 0, мог быть в B как 0, так и 1. А в массиве c никакого выбора нет, можно представить его как тривиальную сумму $c_A = \sum\limits_{A=B} c_B$. Давайте обобщим, представим, что по первым k битам по правилам массива b, а по другим битам по правилам массива c: $b_{k,A} = \sum\limits_{\substack{A \subset B \\ B \setminus A \subset \{0,1,\ldots,k-1\}}} c_A$.

Например, $b_{0,A} = c_A$, а $b_{n,A} = b_A$.

Научимся из $b_{k,A}$ получать $b_{k+1,A}$. Если $k \in A$, то $b_{k+1,A} = b_{k,A}$, а иначе $b_{k+1,A} = b_{k,A} + b_{k,A\cup\{k\}}$.

```
b = copy(c)
for k : 0 .. n - 1
  for i : 0 .. 2^n - 1
    if (i and (1 << k)) = 0
      b[i] += b[i or (1 << k)]
```

Ну и обратная операция:

```
c = copy(b)
for k : n - 1 .. 0
  for i : 0 .. 2^n - 1
```

```
if (i and (1 << k)) = 0
  c[i] -= c[i or (1 << k)]
```

Это работает за $\mathcal{O}(2^n \cdot n)$.

Разбор задачи «Well, Just You Wait!»

В задаче был задан выпуклый многоугольни и одна точка (Петя посадил дерево), нужно было найти самую далекую точку (Вася), которая не отделяется от заданной диагональю многоугольника.

Можно было заметить следующие факты: диагональ разделяет область на две части, и та часть, в которой дерево и будет содержать точку Васи. Поэтому задача сводится к такой: нужно из каждой дигонали сделать полуплоскость, пересечь все полуплоскости, получится выпуклый многоугольник. И в многоугольники найти самую далекую точку от заданной.

Давайте начнем с самой далекой точки: она лежит в вершине многоугольника: если она не на границе многоугольника, можно сдвинуть в направлении вектора, соединяющего две эти точки, а если она на границе, но не вершине, сдвиг в одну из сторон не ухудшит ответ.

Поэтому теперь осталось найти пересечение полуплоскостей: выпуклый многоугольник. Пересекать полуплоскости можно за $\mathcal{O}(k^2)$ или $\mathcal{O}(k \log k)$, где k — число полуплоскостей. Первое — пересекать многоугольник с прямой каждый раз за $\mathcal{O}(k)$, в второе похоже на алгоритм Грэхема: `https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%` или `http://algolist.manual.ru/maths/geom/convhull/graham.php`.

Более быстрый алгоритм поможет набрать больше баллов.

Всего полуплоскостей $\mathcal{O}(n^2)$. Если даже мы решим задачу для одной точки за $\mathcal{O}(n^2 \log n)$, то не уложимся во время. На самом деле из этих n^2 есть только не более n, которые образуют границу этого пересечения. Для каждой вершины i, рассмотрим все диагонали, которые исходят из i и при этом дерево лежит в левой полуплоскости этой диагонали. Нас интересует только та полуплоскость, которая меньше, она ровно одна. Чтобы ее найти, нужно сделать $j = i + 1$, и увеличивать j, пока дерево лежит слева от отрезка (p_i, p_j).

Если применить этот алгоритм построения полуплоскостей, найти пересечение этих полуплоскостей за $\mathcal{O}(n \log n)$, а на самом деле даже за $\mathcal{O}(n)$ (потому что фаза алгоритма пересечения — сортировка, уже выполнилась автоматически, а остальная часть линейна). То получите полный балл.

Разбор задачи «Super Non-massive Black Hole»

Назовем множества точек A и B.

Мы для каждой точки из B хотим найти, на сколько надо подвинуть A, чтобы эта точка доминировалась хотя бы одной точкой из A. Тогда максимум по этим величинам — ответ.

Для фиксированной точки $b \in B$ все точки из A условно разобьём на три класса. Рассмотрим три значения: $b_x - a_x$, $b_y - a_y$ и $b_z - a_z$. Если первое максимально среди трех, то a входит в первый класс, иначе если второе — то во второй, и иначе в третий. Например, в первом множестве могут быть точки $a \in A$, для которых $b_y - a_y \leqslant b_x - a_x$ и $b_z - a_z \leqslant b_x - a_x$. Для таких точек $b_x - a_x$ — это то, на сколько надо сдвинуть (потому что в общем случае надо двигать на $\max\{b_x - a_x, b_y - a_y, b_z - a_z\}$.

Осталось заметить, что условие $b_y - a_y \leqslant b_x - a_x$ эквивалентно условию $b_y - b_x \leqslant a_y - a_x$, аналогично имеем $b_z - b_x \leqslant a_z - a_x$.

Решаем задачу для каждого из трех классов отдельно. Сортируем все точки по убыванию $y - x$, храним дерево отрезков, где индекс — это значение $z - x$, а значение это x. Тогда нужно на отрезке вычислять максимальное значение x.

Тогда для каждой точки B мы найдём точку из A, которую по x надо двигать меньше всего, но при этом по y и z ещё меньше. Повторим это для двух остальных координат. Вычислим максимум среди всех точек.

Время работы $\mathcal{O}(n \log n)$.

Разбор задачи «Yet Another Tree Problem»

Задача решается с помощью методов динамического программирования на дереве.

Наивное решение предполагает посчитать значения (v, i) — наибольший путь, который начинается в v и посещает i помеченных вершин, за линейное время.

За это можно получить 14 баллов.

Когда есть запросы на изменения применим метод разделяй-и-властвуй, воспользуемся тем, что запросы нам заданы заранее. У нас будет рекурсивная функция от отрезка запросов $[L, R)$. Эта функция отвечает на эти запросы на этом отрезке. Поддерживается инвариант для дерева, что дерево сжато так, что в нем есть вершины, которые участвуют в запросах $[L, R)$ (вершина участвует в запросе, если она меняет свою помеченность, или если она конец измененного ребра, или она наименьший общий предок двух помеченных). Чтобы разделиться, нам нужно взять $M = (L+R)/2$, и перейти к $[L, M)$ и $[M, R)$. Чтобы перейти к $[L, M)$, нужно сжать дерево и пересчитать значения функций динамического программирования.

Итого решение за $\mathcal{O}(n \log n)$, где n — число вершин + число запросов.