Разбор задачи «Прогулка по Бруклину»

Решение 1. Перебор. Заметим, что траектория движения движения Майлза однозначно задается местами, в которых он пересекал очередную горизонталь города. Таким образом, существует не более $(n+1)^n$ различных траекторий, и при небольших ограничениях на n мы можем перебрать их все, проверив для каждой, что она не пересекает дома, и выбрать лучшую из подходящих.

Основное решение. Динамическое программирование. Для того, чтобы решить задачу более эффективно решать задачу, применим динамическое программирование. Состоянием будет являться узел решетки, в котором сейчас находится Майлз, направление, в котором он сейчас двигается, а также разность между площадями частей города. Заметим, что эта разность считается только по части города, которую Майлз уже прошел, и она будет меняться в процессе перехода между состояниями динамики. Значением динамики будет булево значение — существует ли такая траектория, которая заканчивается в данном узле, в которой последний шаг был сделан в данном направлении, а текущая разность площадей частей равна данному значению. Чтобы пересчитывать динамику, нам нужно перебрать, в каком направлении сделать очередной шаг, при этом, если необходимо, пересчитать значение разности площадей. Таким образом, каждый переход выполняется за константное время, а общее время работы алгоритмы составит $\mathcal{O}(n^4)$, так как существует $\mathcal{O}(n^2)$ узлов решетки и $\mathcal{O}(n^2)$ значений разности площадей.

Полное решение. Битовое сжатие Заметим, что значения нашей динамики — это двоичные биты, а пересчет этой динамики по одному измерению происходит так: набор подряд идущих значений динамики сдвигается на определенное значение, а после этого применяется операция ИЛИ в каждом бите к определенному диапазону значений. В этом случае можно применить такую оптимизацию динамики, как битовое сжатие: хранить диапазоны значений динамики блоками по 64 бита в целочисленном типе данных. После этого изменения операцию побитового или с диапазоном значений динамики можно производить в 64 раза быстрее. Также, чтобы не реализовывать битовое сжатие, можно применить структуру `bitset`. Таким образом, с помощью этой оптимизации можно было ускорить решение в 64 раза и получить полное решение.