В заключение мы разберём ещё одну классическую задачу.

Паркет
Имеется прямоугольное поле $n \times m$. Нужно посчитать число способов замостить его доминошками размера 1×2. Доминошки могут быть повёрнуты горизонтально или вертикально, но они не должны перекрываться или выходить за пределы поля.

На рис. 1 изображён пример замощения поля 5×6.

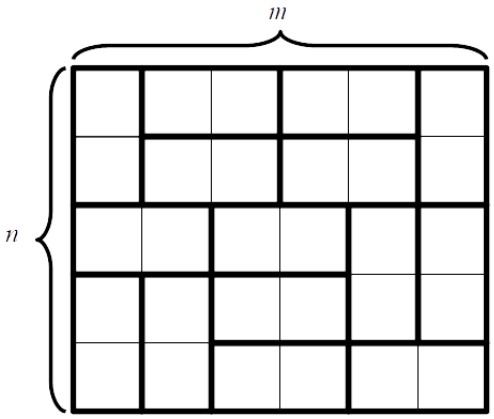

Рис. 1. Пример к задаче «Паркет»

Эта задача очень похожа на задачу замощения доминошками полосы $2 \times n$, с которой мы начинали изучать динамическое программирование (см. рис. 2). Вспомним, как мы ее решали. При решении мы опирались на тот факт, что одна из размерностей полосы — 2, а значит, в конце полосы возможны два варианта. Либо две доминошки лежат горизонтально, либо одна вертикально, и мы можем перейти к полосе меньшего размера.

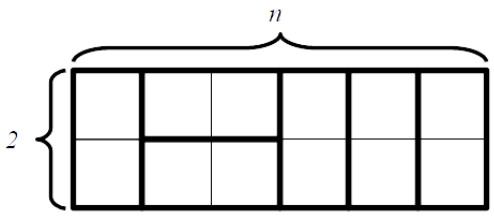

Рис. 2. Пример к задаче о замощении полосы доминошками

Если высота поля — произвольная, вариантов будет не два, а больше. Но мы убедимся, что можно применить аналогичную идею, используя динамику по профилю.

Представим, что мы полностью замостили первые несколько столбцов, следующий столбец замощён неполностью, как показано на рис. 3. Этот частично замощённый столбец и называется *профилем*. Мы можем закодировать его при помощи битовой маски, и решать задачу динамическим программированием.

Рис. 3. Профиль

Состоянием динамики будет пара (i, mask), где i — количество полностью замощённых столбцов, mask — маска профиля. Единицы в маске есть в тех позициях, которые соответствуют замощённным клеткам. d[i][mask] — число различных замощений для такого состояния.

Программный код приведён на рис. 4.

```
d[0][0] = 1;
for (int i = 0; i < m; i++)
for (int mask = 0; mask < (1 << n); mask++)
{
    for (int new_mask = 0; new_mask < (1 << n); new_mask++)
        if (can(mask, new_mask))
            d[i + 1][new_mask] += d[i][mask];
}
cout << d[m][0] << endl;
```

Рис. 4. Решение задачи «Паркет»

Сначала инициализируем начальное состояние, когда ноль столбцов заполнено и маска равна нулю, то есть профиль тоже не заполнен. Это соответствует ситуации, когда на поле нет доминошек, и такая ситуация у нас единственная:

d[0][0] = 1;

Мы считаем, что остальные элементы массива d изначально заполнены нулями. Далее мы делаем динамику «вперёд». Перебираем состояния: i идёт по столбцам от 0 до $(m-1)$, маска перебирается от 0 до (2^n-1):

for (**int** i = 0; i < m; i++)

for (**int** mask = 0; mask < (1 << n); mask++)

Затем мы перебираем новую маску new_mask. Это профиль, который будет в следующем $(i+1)$-м столбце (см. рис. 5):

for (**int** new_mask = 0; new_mask < (1 << n); new_mask++)

Функция can проверяет, возможен ли переход от маски mask к new_mask:

if (can(mask, new_mask))

Если он возможен, мы к d от нового состояния, в котором $(i+1)$ столбец замощён полностью и на профиле маска new_mask, прибавляем количество способов замощения для старого состояния — d[i][mask]:

d[i + 1][new_mask] += d[i][mask];

Конечное состояние — $(m, 0)$. Выводим результат:

cout << d[m][0] << endl;

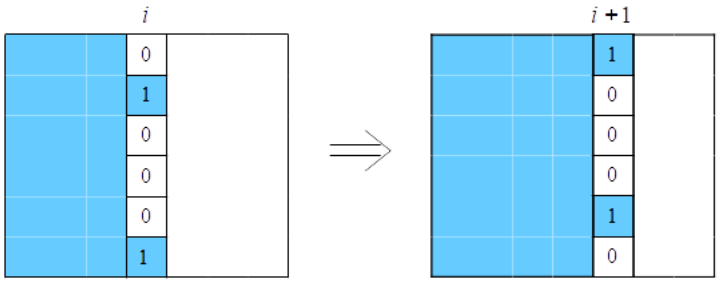

Рис. 5

Осталось решить самый принципиальный вопрос — как работает функция can. Реализация этой функции остаётся в качестве упражнения. Приведём только основные идеи.

Нужно будет для двух масок определять, можно ли перейти от старого профиля к новому путём добавления доминошек. Условимся, что вертикальные доминошки мы будем добавлять только в столбец, соответствующий старому профилю. Тогда переход между профилями будет единственным. Иначе, если мы будем их добавлять в оба столбца, одни и те же замощения посчитаются несколько раз.

Все заполненные клетки в новом профиле соответствуют горизонтальным доминошкам и незаполненным клеткам в старом профиле (см. рис. 6). Иначе говоря, всем единицам в новой маске должны соответствовать нули в старой маске. Проверить, что в масках нет единиц в одинаковых позициях, можно при помощи битовой операции: (mask & new_mask) == 0.

Кроме того, нужно проверить, что все оставшиеся клетки в старом профиле можно заполнить вертикальными доминошками. Добавим в старую маску единички, соответствующие горизонтальным доминошкам. Получится маска (mask | new_mask). Нужно, чтобы нули в ней шли парами. Можно заранее определить для всех масок, удовлетворяют они этому условию или нет. Тогда функция can будет выполняться за $O(1)$. Можно предпосчитать значения

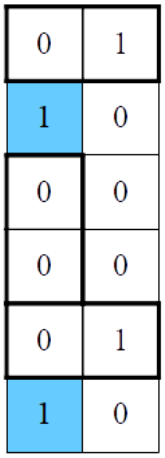

Рис. 6

функции can для всех пар масок, чтобы не вызывать ее каждый раз при выполнении динамики.

Упражнение 4.5.1
Определите асимптотику времени работы программы на рис. 4 при условии наиболее оптимальной реализации функции can.

Приведённое решение будет работать только при достаточно небольшом n, например, до 10. Если $n > m$, то поле можно перевернуть. Но если обе размерности поля достаточно большие (даже 15×15) — решение будет работать слишком долго.

На примере задачи «Паркет» мы разобрали идею динамики по профилю и использование для неё битовых масок. Отметим, что описанный способ решения задачи — не самый оптимальный. Кроме того, у этой задачи существует много разных обобщений.

Подведём итоги модуля. Мы разобрали основные битовые операции с целыми числами. Это AND, OR, XOR, битовые сдвиги. Мы научились смотреть на числа как на битовые маски, которые кодируют подмножества некоторого множества. Битовые маски удобно использовать для перебора подмножеств. Также их можно использовать как состояния динамики. Таким образом, битовые маски — удобный инструмент для решения олимпиадных задач.