Разбор задачи «F. Симпатичные прямоугольники»

Автор задачи и разбора — М. Тихомиров

Решение на 30 баллов

Будем проверять каждый подпрямоугольник на симпатичность и считать количество подходящих. Для этого переберём все возможные части четырьмя вложенными циклами по координатам границ подпрямоугольника. Непосредственным сравнением символов, симметричных относительно центра текущей части, проверим её на симпатичность.

Посчитаем общее количество подпрямоугольников в картине. Каждый из них задается четырьмя числами — координатами верхней и нижней (от 1 до N), а также левой и правой границы (от 1 до M). Поскольку координата верхней границы не должна превышать координаты нижней, возможных комбинаций этих двух чисел $N(N-1)/2 = O(N^2)$. Аналогично для левой и правой границы — $O(M^2)$. Таким образом, всего возможных кусков $O((NM)^2)$.

Количество операций, необходимых для проверки одного куска, равно половине количества символов в нём, что равно $O(NM)$. Перемножая, получаем общую сложность алгоритма — $O((NM)^3)$.

Решение на 60 баллов

В предыдущем решении мы проверяли части на симпатичность независимо друг от друга. Оптимизируем алгоритм засчёт использования уже полученных результатов.

Пусть нам необходимо проверить некоторую часть A, ширина которой больше двух. Рассмотрим прямоугольник B, получаемый из A отрезанием левого и правого крайних столбцов. Несложно заметить, что центры A и B совпадают. Поэтому симпатичность B является необходимым условием симпатичности A. Чтобы завершить проверку, остаётся проверить симметричные буквы из крайних столбцов прямоугольника A.

Итак, два внешних цикла будут по верхней и нижней границам проверяемых кусков. Внутри них мы можем, постепенно увеличивая ширину, проверять каждый новый кусок за $O(N)$. Итоговая сложность — $O(N^3M^2)$.

Код решения (на C++):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  freopen("f.in", "rt", stdin);
  freopen("f.out", "wt", stdout);
  int n, m;
  cin >> n >> m;
  string a[100];
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int ans = 0;
```

```
  for (int x1 = 0; x1 < n; ++x1)
    for (int x2 = x1; x2 < n; ++x2)
    {
      for (int y = 0; y < m; ++y)
      {
        int f = 1;
        for (int d = 0; (y - d >= 0 && y + d < m); ++d)
        {
          for (int x = x1; x <= x2; ++x)
            f &= (a[x][y - d] == a[x2 - (x - x1)][y + d]);
          if (!f) break;
          ans++;
        }
      }
      for (int y = 1; y < m; ++y)
      {
        int f = 1;
        for (int d = 1; (y - d >= 0 && y + d - 1 < m); ++d)
        {
          for (int x = x1; x <= x2; ++x)
            f &= (a[x][y - d] == a[x2 - (x - x1)][y + d - 1]);
          if (!f) break;
          ans++;
        }
      }
    }
  cout << ans << endl;
  return 0;
}
```

Правильное решение

В этом пункте мы научимся проверять часть A на симпатичность за $O(1)$. Для этого рассмотрим два ее подпрямоугольника. Первый, B, как и до этого получается из A отрезанием левого и правого крайних столбцов. Второй, C, получается отрезанием верхней и нижней крайних строк. Опять же, для симпатичности A необходимо и достаточно выполнение двух условий:

- B и C симпатичные,
- противоположные угловые клетки A равны.

Эта идея позволяет проверять каждый кусок за $O(1)$, что дает итоговую сложность $O((NM)^2)$. Основная трудность реализации заключается в том, что в каждый момент времени надо иметь информацию о симпатичности частей B и C. При этом хранить каждый прямоугольник в памяти явно невозможно: 100^4 байт не укладывается в граничения задачи). Проблема решается несколькими способами:

- Независимые вычисления для каждого центра (в этом случае расход памяти квадратичный).

- Поскольку общее количество подпрямоугольников не может быть больше $(100 * 99/2)^2$, то эффективной их нумерацией можно добиться уменьшения размера массива в 4 раза.
- Использование битовых массивов.

Релизация третьего подхода:

```
#include <iostream>
using namespace std;

char b[13][100][100][100] = {{{{0}}}};

void place1(int x1, int x2, int y1, int y2)
{
  b[x1>>3][x2][y1][y2] += 1<<(x1&7);
  return;
}

bool get(int x1, int x2, int y1, int y2)
{
  if (x1 > x2 || y1 > y2) return 1;
  return (b[x1>>3][x2][y1][y2] & (1 << (x1 & 7)));
}

int main()
{
  int n, m;
  freopen("f.in", "rt", stdin);
  freopen("f.out", "wt", stdout);
  cin >> n >> m;
  string a[100];
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int ans = 0;
  for (int dx = 1; dx <= n; ++dx)
    for (int dy = 1; dy <= m; ++dy)
      for (int x = 0; x + dx <= n; ++x)
        for (int y = 0; y + dy <= m; ++y)
          if (get(x+1, x+dx-2, y, y+dy-1) && get(x, x+dx-1, y+1, y+dy-2) &&
              a[x][y] == a[x+dx-1][y+dy-1] && a[x][y+dy-1] == a[x+dx-1][y])
          {
            place1(x, x+dx-1, y, y+dy-1);
            ans++;
          }
  cout << ans << endl;
  return 0;
}
```

Решение на $+\infty$

Для дальнейшей оптимизации решения используется полиномиальное хэширование строк. За $O(NM)$ насчитаем хэш для всех префиксов всех строк картины, а также для префиксов инвертированных строк. Теперь мы за $O(1)$ можем получить хэш подстроки любой строки и любой инвертированной строки. Про полиномиальное хеширование можно прочитать, например, здесь:

`http://e-maxx.ru/algo/string_hashes`

.

Внешний цикл будет по координатам центра проверяемых частей. Рассмотрим отдельно случаи чётной и нечётной высоты кусков (без ограничения общности — нечётной). Найдем максимальную строку с данным центром, являющуюся симпатичной частью (то есть палиндромом). Увеличивая высоту на 2 (прибавляя по строке сверху и снизу), каждый раз будем определять максимальную длину части с такой высотой, являющейся симпатичной. Это число полностью зависит от добавленных строк. Сравнивая хэши прямой верхней и инвертированной нижней строк, будем уменьшать длину на 2 до тех пор, пока они не станут равными. Количество симпатичных частей с данным центром и данной высотой будет равно $(k + 1)/2$, где k — найденная длина максимальной части, а $/$ — целочисленное деление. Прибавим эту величину к ответу.

Для чётной высоты следует начинать с частей высоты 2.

Поскольку при увеличении высоты максимальная длина не увеличивается, для обработки каждого центра потребуется $O(N + M)$ операций. Общая сложность алгоритма равна $O(NM(N + M))$.

Код решения (автор — Сергей Копелиович):

```
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

typedef long long ll;

#define forn(i, n) for (int i = 0; i < (int)(n); i++)
#define P 239017
#define maxn 100

int h, w, eq[2][maxn][maxn];
char s[maxn][maxn + 1];
ll h1[maxn][maxn + 1];
ll h2[maxn][maxn + 1];
ll p[maxn + 1];

inline ll H1( int i, int l, int r )
{
  return h1[i][r + 1] - h1[i][l] * p[r - l + 1];
}
```

```
inline ll H2( int i, int l, int r )
{
  return h2[i][l] - h2[i][r + 1] * p[r - l + 1];
}

int main()
{
  p[0] = 1;
  forn(i, maxn)
    p[i + 1] = p[i] * P;

  freopen("f.in", "r", stdin);
  freopen("f.out", "w", stdout);

  scanf("%d%d", &h, &w);
  forn(i, h)
    scanf("%s", s[i]);

  memset(h1, 0, sizeof(h1));
  forn(i, h)
    forn(j, w)
      h1[i][j + 1] = h1[i][j] * P + s[i][j];
  memset(h2, 0, sizeof(h2));
  forn(i, h)
    for (int j = w - 1; j >= 0; j--)
      h2[i][j] = h2[i][j + 1] * P + s[i][j];

  ll sum = 0;
  forn(t1, 2)
    forn(t2, 2)
      forn(i, h)
        forn(j, w)
        {
          int k = w;
          while (!(0 <= j - k && j + k + t2 < w))
            k--;
          for (int l = 0; 0 <= i - l && i + l + t1 < h; l++)
          {
            while (k >= 0 &&
                   H1(i - l, j - k, j + k + t2) != H2(i + l + t1, j - k, j + k + t2))
              k--;
            sum += k + 1;
          }
        }
  cout << sum << endl;
  return 0;
}
```