# Разбор задачи «Пеш-хматы»

Постановка задачи: Дано шахматное поле стоят черными пешками, нужно составить маршрут для белой пешки так, чтобы она съела как можно больше черных фигур.

### Тест 1

Для первого теста можно было перебрать все варианты маршрутов и проверить каждый.

### Тест 2

Для второго теста можно было перебрать варианты маршрута, отсеивая большинство заведомо плохих ходов.

### Тесты 3 и 4

Для полного решения задачи используем метод динамического программирования. Динамическое программирование позволяет избежать повторных вычислений и значительно ускорить процесс.

Для удобства пусть клетка $(1;1)$ находится в левом верхнем углу.

Пусть $dp[i][j]$ — количество съеденных черных фигур, если мы стартуем с $i$ строчки и $j$ столбца (допускаем, что если на $i$ строчке и $j$ столбце стоит пешка то мы можем поставить туда пешку и съесть фигуру).

База динамики: Инициализируем базу динамики значениями на первой строке, учитывая наличие черных фигур. То есть $dp[1][j] = 1$ при наличии черной фигуры и $dp[1][j] = 0$ при отсутствии в $(1;j)$ клетке.

Так как для первой строчки ответ посчитан, то переходы для строк $i$, где $i > 1$. Динамика заполняется, учитывая возможность движения белой пешки вверх и по диагонали. Если пешка стоит на $(i;j)$ позиции, то она может пойти в $(i-1;j-1)$, $(i-1;j)$ и $(i-1;j+1)$, для которых ответы уже посчитаны. Переходы динамики: $dp[i][j] = \max(dp[i-1][j-1], dp[i-1][j], dp[i-1][j+1])$. Если $(i;j)$ клетке стоит черная пешка, то прибавим 1 к $dp[i][j]$.

Параллельно вычисления динамики поддерживаем маршрут.

| Номер теста | Ответ |
|---|---|
| 1 | RRL |
| 2 | RUULRLL |
| 3 | LRLLUUR |
| 4 | LUURRR |

# Разбор задачи «Цирк»

Постановка задачи: определить количество детей, которые могут посетить цирк бесплатно в соответствии с порядком раздачи билетов в сектора $A, B, C, D, C, B, A, \ldots$.

### Подзадача 1

Моделирование

Для первой подзадачи можно написать решение, которое будет моделировать процесс раздачи билетов.

```
vector<long long> ticket = {a, b, c, d};
vector<long long> tickets = {0, 1, 2, 3, 2, 1};
long long ans = 0;
while(ticket[tickets[it % 6]] != 0){
  ticket[tickets[it % 6]]--;
  ans++;
}
```

### Полное решение

Для полного решения задачи оптимизируем моделирование. Заметим, что последовательность секторов повторяется циклически $(A, B, C, D, C, B)$. В цикле в сектора $A$ и $D$ было отдано по одному билету, в сектора $B$ и $C$ — по два билета. Посчитаем сколько полных циклов могло быть сделано $cycle = \min(A, D, \lfloor \frac{B}{2} \rfloor, \lfloor \frac{C}{2} \rfloor)$. Запись $\lfloor x \rfloor$ обозначает округление вниз значения $x$ (например, $\lfloor 1.4 \rfloor = 1$, $\lfloor 2 \rfloor = 2$).

После того как мы вычислили количество полных циклов осталось посчитать сколько будем отдано билетов в последнем не полном цикле. Для этого изменим $A = A - cycle, B = B - 2 \cdot cycle, C = C - 2 \cdot cycle, D = D - cycle$. После этого можно использовать алгоритм из первой подзадачи.

```
long long cycle = min(
    min(a, d),
    min((long long)b/2, (long long)c/2)
  );
long long ans = cycle * 6LL;

a -= cycle;
b -= 2LL*cycle;
c -= 2LL*cycle;
d -= cycle;

vector<long long> ticket = {a, b, c, d};
vector<long long> tickets = {0, 1, 2, 3, 2, 1};
long long ans = 0;
while(ticket[tickets[ans % 6]] != 0){
  ticket[tickets[ans % 6]]--;
  ans++;
}
cout << ans + cycle * 6LL;
```

# Разбор задачи «Слова»

Заметим, что время набора каждого слова не зависит от того, каким по счету мы его будем набирать, если вообще будем. Поэтому для каждого слова можно сразу посчитать это время. Чтобы это сделать, пройдемся по каждому слову $s_1, \ldots, s_l$ и добавим к ответу $t_{s_1,s_2} + \ldots + t_{s_{l-1},s_l}$.

Далее нужно выбрать $m$ слов с минимальным временем набора. Для этого отсортируем слова в соответствии с необходимым для набора временем, после чего ответ - это сумма $m$ минимальных временем.

# Разбор задачи «Пеш-хматы»

Постановка задачи: Дана матрица на которой стоят черные фигуры, нужно поставить пешку и составить маршрут для белой пешки так, чтобы она съела как можно больше черных фигур. Белую пешку нельзя ставить на поле, где есть черная пешка.

### Подзадача 1

Наивный рекурсивный перебор

Для первой подзадачи можно написать наивное рекурсивное решение. Рекурсивная функция будет рассматривать все возможные позиции белой пешки, проверять их наличие на поле и, при возможности, рекурсивно вызывать саму себя для следующего шага. Такое решение может быть полезным для небольших размеров поля, но оно неэффективно для больших размеров из-за повторных вычислений.

```
def rec(x, y):
    global ans, pathAns, curPath, cur, a, w
    if x == 0:
        if cur > ans:
```

```
            ans = cur
            pathAns = curPath
        return

    if y - 1 >= 0:
        curPath += "L"
        cur += (a[x - 1][y - 1] == 'B')
        rec(x - 1, y - 1)
        cur -= (a[x - 1][y - 1] == 'B')
        curPath = curPath[:-1]
    if y + 1 < w:
        curPath += "R"
        cur += (a[x - 1][y + 1] == 'B')
        rec(x - 1, y + 1)
        cur -= (a[x - 1][y + 1] == 'B')
        curPath = curPath[:-1]

    curPath += "U"
    cur += (a[x - 1][y] == 'B')
    rec(x - 1, y)
    cur -= (a[x - 1][y] == 'B')
    curPath = curPath[:-1]
    return

def solve():
    global ans, pathAns, curPath, cur, a, w, h, ansX, ansY
    h, w = map(int, input().split())

    a = []
    for _ in range(h):
        a.append(input())

    last = -1
    for j in range(1, h):
        for i in range(w):
            if a[j][i] == '*':
                last = ans
                rec(j, i)
                if ans > last:
                    ansX, ansY = j, i
    print(h-ansX, ansY + 1)
    print(ans)
    print(pathAns)
    return

ans = 0
cur = 0
pathAns = ""
curPath = ""
ansX = 0
ansY = 0
w = 0
h = 0
a = []

if __name__ == "__main__":
    solve()
```

## Полное решение

Для удобства пусть клетка $(1; 1)$ находится в левом верхнем углу.

Для полного решения задачи используем метод динамического программирования. Динамическое программирование позволяет избежать повторных вычислений и значительно ускорить про-

цесс.

Пусть $dp[i][j]$ — количество съеденных черных фигур, если мы стартуем с $i$ строчки и $j$ столбца (допускаем, что если на $i$ строчке и $j$ столбце стоит пешка то мы можем поставить туда пешку и съесть фигуру).

База динамики: Инициализируем базу динамики значениями на первой строке, учитывая наличие черных фигур. То есть $dp[1][j] = 1$ при наличии черной фигуры и $dp[1][j] = 0$ при отсутствии в $(1; j)$ клетке.

Так как для первой строчки ответ посчитан, то переходы для строк $i$, где $i > 1$. Динамика заполняется, учитывая возможность движения белой пешки вверх и по диагонали. Если пешка стоит на $(i; j)$ позиции, то она может пойти в $(i-1; j-1)$, $(i-1; j)$ и $(i-1; j+1)$, для которых ответы уже посчитаны. Переходы динамики: $dp[i][j] = \max(dp[i-1][j-1], dp[i-1][j], dp[i-1][j+1])$. Если $(i; j)$ клетке стоит черная пешка, то прибавим 1 к $dp[i][j]$.

Ответ находится в $(i; j)$ клетке, где не стоит черная фигура и наибольшее значение. Так как мы поменяли местоположение клетки $(1; 1)$, то выводить надо $(h - i + 1, j)$.

Для восстановления маршрута воспользуемся массивом предков, то есть запомним куда мы пошли из клетки $(i; j)$.

```python
def solve():
    h, w = map(int, input().split())

    a = [input() for _ in range(h)]

    dp = [[0] * w for _ in range(h)]
    parent = [[-1] * w for _ in range(h)]
    for i in range(w):
        dp[0][i] = int(a[0][i] == 'B')
    for i in range(1, h):
        dp[i][0] = int(a[i][0] == 'B') + dp[i - 1][0]
        parent[i][0] = 0
        if 1 < w and int(a[i][0] == 'B') + dp[i - 1][1] > dp[i][0]:
            dp[i][0] = int(a[i][0] == 'B') + dp[i - 1][1]
            parent[i][0] = 1

        dp[i][w - 1] = int(a[i][w - 1] == 'B') + dp[i - 1][w - 1]
        parent[i][w - 1] = w - 1
        if w - 2 >= 0 and int(a[i][w - 1] == 'B') + dp[i - 1][w - 2] > dp[i][w - 1]:
            dp[i][w - 1] = int(a[i][w - 1] == 'B') + dp[i - 1][w - 2]
            parent[i][w - 1] = w - 2

        for j in range(1, w - 1):
            if dp[i - 1][j - 1] > dp[i][j]:
                dp[i][j] = dp[i - 1][j - 1]
                parent[i][j] = j - 1

            if dp[i - 1][j] > dp[i][j]:
                dp[i][j] = dp[i - 1][j]
                parent[i][j] = j
            if dp[i - 1][j + 1] > dp[i][j]:
                dp[i][j] = dp[i - 1][j + 1]
                parent[i][j] = j + 1
            dp[i][j] += int(a[i][j] == 'B')

    ans = (-1, -1)
    for i in range(h):
        for j in range(w):
            if a[i][j] == '*' and (ans[0] == -1 or dp[ans[0]][ans[1]] < dp[i][j]):
                ans = (i, j)

    print(h - ans[0], ans[1] + 1)
    print(dp[ans[0]][ans[1]] - int(a[ans[0]][ans[1]] == 'B'))
    s = ""
```

```
    while parent[ans[0]][ans[1]] != -1:
        if parent[ans[0]][ans[1]] - ans[1] == -1:
            s += "L"
        elif parent[ans[0]][ans[1]] - ans[1] == 0:
            s += "U"
        elif parent[ans[0]][ans[1]] - ans[1] == 1:
            s += "R"
        ans = (ans[0] - 1, parent[ans[0]][ans[1]])
    print(s)

if __name__ == "__main__":
    solve()
```

# Разбор задачи «Плейлист»

Для решения первой подзадачи достаточно перебрать все перестановки из $n$ элементов и для каждой проверить, будет ли она интересна Спарксу.

Рассмотрим решение подгруппы, где все числа различны. Заметим, что до максимального элемента все числа будут возрастать, а после него - убывать. Таким образом, каждый элемент, кроме максимума, может находиться либо в левой (возрастающей) части, либо в правой (убывающей). Кроме того, если мы зафиксировали, в какой части находится каждый элемент, то ответ порядок определяется единственным образом. Это значит, что каждый способ распределить элементы по частям соответствует одному корректному порядку, и наоборот. Поэтому ответ — это количество способов так распределить элементы. Это число равно $2^{n-1}$, потому что каждый элемент, кроме максимума, может быть либо в левой части, либо в правой.

Для решения на полный балл заметим следующее. Во-первых, все вхождения максимального элемента будут идти подряд в пиковой части последовательности, по аналогии со случаем с уникальными элементами. Для остальных элементов сколько-то может находиться слева от всех максимумом, сколько-то — справа. Обозначим за $c_1, \ldots, c_{max}$ количество вхождений в массив $a$ чисел 1, 2, …, $max$. Тогда есть $(c_1+1)\cdot\ldots\cdot(c_{max-1}+1)$ способов выбрать разбиение всех чисел. Однако, мы не учли то, что для нас порядок всех вхождений каждого уникального числа $i$ — важен. Поэтому нужно еще домножить на $c_1!\cdot\ldots\cdot c_{max}!$. Итого ответ равен $(c_1+1)!\cdot\ldots(c_{max-1}+1)!\cdot c_{max}!$.