Problem Tutorial: “Pawn Chess”

Task Description: Given a chessboard with black pawns, you need to plan a route for a white pawn in a way that it captures as many black pieces as possible.

Test 1

For the first test, you could iterate through all possible routes and check each one.

Test 2

For the second test, you could explore route options, eliminating most obviously bad moves.

Tests 3 and 4

For a complete solution, use dynamic programming. Dynamic programming helps avoid redundant calculations and significantly speeds up the process.

For convenience, let the cell $(1; 1)$ be in the top-left corner.

Let $dp[i][j]$ be the count of captured black pieces if we start from row i and column j (assuming that if there is a pawn at row i and column j, we can place a pawn there and capture a piece).

Dynamic Programming Base: Initialize the base of dynamic programming with values on the first row, considering the presence of black pieces. So, $dp[1][j] = 1$ when there is a black piece, and $dp[1][j] = 0$ when there is no piece at cell $(1; j)$.

Once the answer is computed for the first row, transitions for rows i where $i > 1$ are considered. Dynamics are filled in, considering the possibility of the white pawn moving upwards and diagonally. If there is a pawn at position $(i; j)$, it can move to $(i - 1; j - 1)$, $(i - 1; j)$, and $(i - 1; j + 1)$, for which answers have already been calculated. Dynamic transitions: $dp[i][j] = \max(dp[i - 1][j - 1], dp[i - 1][j], dp[i - 1][j + 1])$. If there is a black pawn at cell $(i; j)$, add 1 to $dp[i][j]$.

Simultaneously, while calculating dynamics, maintain the route.

Test Number	Answer
1	RRL
2	RUULRLL
3	LRLLUUR
4	LUURRR

Problem Tutorial: “Circus”

Problem Statement: Determine the number of children who can attend the circus for free according to the ticket distribution pattern in sectors $A, B, C, D, C, B, A, \ldots$.

Subtask 1: Modeling

For the first subtask, a solution can be written to simulate the ticket distribution process.

```
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++;
}
```

Complete Solution

For the complete solution, optimize the simulation. Notice that the sector sequence repeats cyclically (A, B, C, D, C, B). In each cycle, one ticket is given to sectors A and D, and two tickets to sectors B and C. Calculate the number of complete cycles $cycle = \min(A, D, \lfloor \frac{B}{2} \rfloor, \lfloor \frac{C}{2} \rfloor)$. Here, $\lfloor x \rfloor$ denotes rounding down the value of x (for example, $\lfloor 1.4 \rfloor = 1$, $\lfloor 2 \rfloor = 2$).

After computing the number of complete cycles, determine how many tickets will be distributed in the last incomplete cycle. Modify $A = A - cycle, B = B - 2 \cdot cycle, C = C - 2 \cdot cycle, D = D - cycle$. Then, use the algorithm from the first subtask.

```
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;
```

Problem Tutorial: "Words"

Notice that the typing time for each word does not depend on its position in the sequence, if it is typed at all. Therefore, we can immediately calculate the typing time for each word. To do this, iterate through each word $s_1, \ldots, s_l$ and add $t_{s_1,s_2} + \ldots + t_{s_{l-1},s_l}$ to the answer.

Next, we need to select m words with the minimum typing time. To achieve this, sort the words according to their required typing time, and the answer is the sum of the m smallest times.

Problem Tutorial: "Pawn-chess"

Problem Statement: Given a matrix with black pieces, the goal is to place a white pawn and determine a route for it to capture as many black pieces as possible. The white pawn cannot be placed on a cell where there is a black pawn.

Subtask 1: Naive Recursive Enumeration

For the first subtask, a naive recursive solution can be implemented. The recursive function explores all possible positions for the white pawn, checks their presence on the board, and recursively calls itself for the next step when possible. While this solution may be useful for small board sizes, it becomes inefficient for larger sizes due to repeated computations.

```
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()
```

Complete Solution

For the complete solution, a dynamic programming approach is employed for efficiency. Dynamic programming avoids redundant calculations and significantly speeds up the process.

Let cell $(1;1)$ be in the top-left corner for convenience. The dynamic programming approach involves a 2D array $dp[i][j]$ representing the count of captured black pieces if starting from row i and column j. The base of the dynamic programming is initialized with values from the first row, considering the presence of black pieces.

The transitions for rows i where $i > 1$ are determined by accounting for the upward and diagonal movement of the white pawn. The transitions are given by $dp[i][j] = \max(dp[i-1][j-1], dp[i-1][j], dp[i-1][j+1])$. If there is a black pawn at cell $(i;j)$, add 1 to $dp[i][j]$.

The answer is found in the cell $(i;j)$ where there is no black piece and the value is the maximum. Since the location of cell $(1;1)$ has changed, the output should be $(h-i+1, j)$.

To reconstruct the route, an array of ancestors is used to remember the direction from which the white pawn moved.

```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()
```

Problem Tutorial: “Playlist”

Solution Explanation:

To solve the first subtask, it is sufficient to generate all permutations of the n elements and check whether each permutation is interesting to Sparks.

Let’s examine the solution for the subgroup where all numbers are distinct. Notice that up to the maximum element, all numbers will be in ascending order, and after the maximum element, they will be in descending order. Thus, each element, except for the maximum, can be either in the left (ascending) part or in the right (descending) part. Moreover, if we fix in which part each element is located, the order is uniquely determined. This means that each way of distributing elements among the parts corresponds to a correct order, and vice versa. Therefore, the answer is the number of ways to distribute elements, which is 2^{n-1} because each element, except for the maximum, can be in the left or right part.

For a full score solution, let’s observe the following. Firstly, all occurrences of the maximum element will be contiguous in the peak part of the sequence, similar to the case with unique elements. For the other elements, some may be to the left of all occurrences of the maximum, and some may be to the right. Let $c_1, \ldots, c_{\max}$ denote the count of occurrences of the numbers $1, 2, \ldots, \max$ in the array a. Then, there are $(c_1 + 1) \cdot \ldots \cdot (c_{\max-1} + 1)$ ways to choose the partition for all numbers. However, we have not considered that the order of all occurrences of each unique number i is important. Therefore, we need to multiply by $c_1! \cdot \ldots \cdot c_{\max}!$. In total, the answer is $(c_1 + 1)! \cdot \ldots \cdot (c_{\max-1} + 1)! \cdot c_{\max}!$.