And that's basically it. See the reference solution for the details of how to implement all this.

StripePainter

Discuss it

Used as: Division-I, Level 2:

Value	500
Submission Rate	23 / 123 (18.70%)
Success Rate	10 / 23 (43.48%)
High Score	**bstanescu** for 390.72 points

Reference implementations: **DjinnKahn**, **ValD**

Implementation

This was an unusually hard 500p, requiring a non-straightforward dynamic programming solution (or memoization). As almost always is the case with this type of problems, the solution is easy once you see it. However, in an ordinary SRM, this would be rated as a hard problem. Why it was not a 600 pointer, I don't understand.

Lets try the standard induction approach: given a continuous part of the original strip and the current color this strip has (initially the whole strip has an undefined color), we want to break this down into smaller instances of the same problem. Define this problem like this:

```
min[left,size,color] = the minimum number of brushes required to paint position
                       left, left+1, ... , left+size-1 of the original strip,
                       assuming these position currently has color color.
```

The desired return value is then simply min[0,stripes.length(),'?'].

It requires one insight to solve this min-problem: the next stroke may always start at the leftmost position unless this position is already in the correct color. If the leftmost position is in the wrong color, we must paint it sooner or later anyway, so why not now? This reasoning only make sense for the leftmost (and rightmost) position, because there is no gain to first paint this position with another color, which may be true of interior positions.

So we first check if the leftmost position is in the wrong color. If it already is in the desired color, we simply return min[left+1,size-1,color] since we then have one less position to worry about. Otherwise we loop over all possible stroke lengths, from 1 to size:

```
loop i from 1 to size
   sum = 1 + min[left+1,i-1,stripe[left]] + min[left+i,size-i,color];
   update best solution found so far if necessary
end loop
```

The logic is as follows: we make a stroke of color stripe[left] (the color of the leftmost position) from left to left+i-1, inclusive - this accounts for the 1. We then get two subproblems, one part of the strip is in color stripe[left], the other part is in the original color. We solve these subproblems by a recursive call. On the part of the strip which we changed color, we know that the leftmost position is in the correct color, so we can skip ahead one position there (thus explaining the left+1,i-1).

What remains is a terminating case, so we don't recurse forever. That one is simple: if the length of the stripe is zero, it requires no strokes at all.

When implementing this, it's essential that we use memoization, otherwise our solution will time out pretty fast. Memoization should always be considered in a recursive function which doesn't have any side effects, such as changing global variables. A function is a pure mathematical function if it always returns the same value given the same input parameters. Such functions are the only ones memoization can be applied on: if we ever call the function with the same parameters as we've done before, we just look up what answer we got last time instead of evaluating it again. It can be implemented like this:

```
int memo[50][51][128]; // Initialize to -1 to mark not evaluated

int min(int left, int size, char col)
{
  if (memo[left][size][col]>=0) return memo[left][size][col];

  // Evaluate function and store result in best

  memo[left][size][col]=best;
  return best;
}
```

One can also use dynamic programming to solve the problem. This is basically the same thing, except that instead of the recursive calls, we evaluate the memo table in such an order that the results of recursive calls already has been calculated, like this:

```
for(int size=0;size<=stripes.length();size++)
  for(left=0;left<stripes.length()-size;left++)
    for(col='@';col<='Z';col++) {
      // Evaluate min[left][size][col] here
    }
```

Notice that in the evaluation of min[left,size,col] the subproblems always had a smaller size than the original problem, so if we evaluate the subproblems starting from size 0 and then working up, these recursive calls can instead be looked up directly from the memo table as we know they have already been evaluated.

And that's it! The hardest part in a problem like this is figuring out the mathematical function and what parameters it should use. Once this is done, it's pretty straightforward. Just remember that the function must be a mathematical one (without side effects), otherwise we can't use memoization.

For an implementation of the recursive approach, see **DjinnKahns** solution and for a dynamic programming solution, see the solution by **ValD**.

RoboCourier

Discuss it

Used as: Division-I, Level 3:

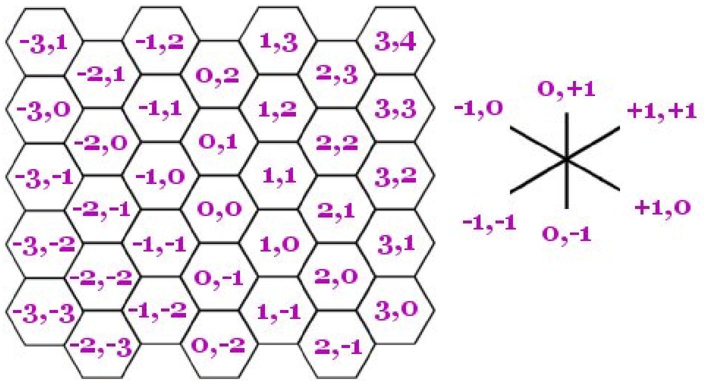
Value	1000
Submission Rate	7 / 123 (5.69%)
Success Rate	4 / 7 (57.14%)
High Score	**radeye** for 604.51 points

Reference implementation: **Yarin**

Implementation

It's apparent that this is a typical shortest-path problem, albeit a slightly trickier one due to the hexagonal coordinate system and different travelling speeds depending on if it's the first (or last) move in a direction or not.

The first thing to do is to get rid of the annoying hexagonal coordinate system. Or rather, how to translate the 6 hexagonal directions into equivalent directions in a grid world. The following picture shows how this can be done:



As can be seen, the 6 directions corresponds to movements in x & y coordinates according to the picture to the right. Or, if translated to code, the following constant arrays:

```
const int dx[6]={0,1,1,0,-1,-1};
const int dy[6]={1,1,0,-1,-1,0};
```

We can thus easily represent a position in the hexagonal grid using ordinary x,y coordinates.

The next step is to convert the input to an undirected graph. Each node in the graph should correspond to a hexagon that the scout robot has visited, and the edges in the graph should correspond to movements from one hexagon to another by the robot. One can note that each node can have at most 6 edges, one for each possible direction, and there can be at most 501 nodes (for instance, if the scout executes 500 F's). So to represent each node in the graph, we can use the following structure: (C++ code)

```
struct TNode
{
  int x,y; // x and y coordinates of the node
  int edge[6]; // Node number to reach if travelling in this direction. -1 if no edge.
};

TNode nodes[501];
int noNodes; // Number of nodes in graph
```

We create this graph by simulating the movements of the scout robot. Every time the scout moves ahead, we check which node it's at (if it's a new node, we add the information about this nodes coordinate in the structure above) and add the appropriate edge (in both directions!). It might look like this:

```
int x=0,y=0,dir=0; // Start square
for(int i=0;i<pathConcat.size();i++) {
  switch (pathConcat[i]) {
    case 'R' : dir=(dir+1)%6; break;
    case 'L' : dir=(dir+5)%6; break;
    case 'F' :
      src=lookup(x,y);
      x+=dx[dir];
      y+=dy[dir];
      dest=lookup(x,y);
```