Rope Bridge

Problem Description

The full description can be found in this site's repository.

You are going to write an algorithm which opens a file `input.txt` and follows a series of motions for the head of a rope. The rope is made up of a series of knots that move in accordance with the rules of the rope. You will track the movement of the tail of the rope throughout the motions.

Consider a rop with a knot at each end; these knots mark the head and the tail of the rope. If the head moves far enough away from the tail, the tail is pulled toward the head. You are able to model the positions of the knots on a two-dimensional grid. Following a series of motions for the head, you will determine how the tail will move.

- If the head is ever two steps up, down, left, or right from the tail, the tail must also move in one step that direction so it's close enough.
- Otherwise, if the head and tail aren't touching and aren't in the same row or column, the tail always moves one step diagonally to keep up.

Your goal is to simulate a rope consisting of *ten* knots. One knot is the head of the rop and moves according to the series of motions. Each knot further down the rope follows the knot in front of it using the same rules above. You need to keep track of the positions that the new tail, `9`, visits.

You are going to compute the *number of positions that the tail of the rope visits at least once*. Encapsulate your answer in flag braces, i.e. `flag{36}`. *Hint*: The start point is arbitrary.

Walkthrough

This is an interesting challenge. There are many ways to solve this one, and I don't think that my solution is the best. I want to use this challenge to demonstrate my thought process in solving challenges. Although the most efficient solution may not come out the first time, that doesn't discount it as a working solution. If you want to practice writing more space efficient solutions, I challenge you to track with a linked list, and check for unique points before you enter them into the list. Be sure that you don't inadvertently write an `O(n^2 log n)` solution; mine runs in `O(n log n)` time.

> **Remember that you have limited time in the screener.** If you spend all your time searching for a more optimal solution, you won't have time to solve other problems. It's better to have a working solution that is prone to bad input or isn't the most efficient than to lose points in other areas.
>
> If you have spare time at the end, this is the time to polish solutions. I submitted some weird solutions on game-day that worked, but were far less efficient. I brushed up on my skills between screener and interview time and was able to produce better quality code during the interview.

Our goal is to track the number of unique points that are in the graph. A *static array* does this for us just fine.

- We can make a `struct` to represent a point on the graph.

- We can make a `struct` to represent a vector between two points. This will help us track the distance between each knot to see if we must move any knot to catch up.

> The vector `struct` is not an intuitive idea. I didn't think about making one until most of the way through my solution. I'll mention when I decided to add this to my solution. It is perfectly valid to solve it without one, but it made sense in my head at the time of writing.

The current plan of action is to read each line, perform the instruction, and then add the tail's position to the list. Then, we can sort the list of tail positions and count the number of unique points. *If you're comfortable, try to solve this by checking for unique points before adding to the list, but watch for exponential time complexity.*

Reading the File

We'll begin with the basics. We need to open the file and initialize the data.

```
FILE* fp = fopen("input.txt", "r");
if (fp == NULL)
    exit(EXIT_FAILURE);

char* line = NULL;
size_t len = 0;
```

In this stage, we also need to define our `struct Point` and `struct Vector`.

```
struct Point {
    int x;
    int y;
} Point_default = {0, 0};

struct Vector {
    int x;
    int y;
} Vector_default = {0, 0};
```

This syntax allows us to set a default value. Note that `_default` is not required, but I use this to distinguish that this object is the default for each `struct`. If I want to initialize a `struct Point` with the default, I can do the following:

```
struct Point point = Point_default;
```

Now, we need to initialize the chain. The chain contains 10 points. We can initialize them all to the default value.

```

```

```
struct Point* chain = malloc(10 * sizeof(struct Point));
for (size_t i = 0; i < 10; ++i)
    chain[i] = Point_default;
```

We also need to hold the list of points that the tail visits. As stated previously, we are using a static array to do this.

```
struct Point* tail_visits = malloc(20000 * sizeof(struct Point));
```

Parsing the Data

We can use the same `getline` function that we used in the previous challenge to read each line. Since we know the length of each line and it doesn't matter to us, we don't need to store `read`.

```
while (getline(&line, &len, fp) != -1) { ... }
```

Each line has two items: a direction and a number of steps. We can use `strtok` to parse each line.

```
size_t count = 0;
while (getline(&line, *&len, fp) != -1) {
    char dir = strtok(line, " ")[0];
    int times = atoi(strtok(NULL, " "));

    /* process the instruction */

    ++count;
}
```

We need to process each instruction `times` times. Each instruction, we should determine which direction we travel and move the head of the chain. We then need to adjust the rest of the chain to follow the head. Finally, we need to record the tail's position in the list.

```
// read the directional changes
size_t count = 0;
while (getline(&line, &len, fp) != -1) {
    char dir = strtok(line, " ")[0];
    int times = atoi(strtok(NULL, " "));
    while (times-- > 0) {
        // adjust the head coordinate
        if (dir == 'U')
            ++chain[0].y;
        else if (dir == 'D')
            --chain[0].y;
```

```
            else if (dir == 'L')
                --chain[0].x;
            else if (dir == 'R')
                ++chain[0].x;
            else {
                printf("Bad\n");
                exit(EXIT_FAILURE);
            }

            // adjust the links of the chain
            for (size_t i = 0; i < 9; ++i)
                chain[i + 1] = adjust(chain[i], chain[i + 1]);

            // record tail location in list
            tail_visits[count++] = chain[9];
        }
    }
```

We make a call to `adjust` to move each link of the chain. I chose to make this a free function because we must do it for each chain link and it makes the code a lot cleaner. This method must determine if the provided chain links are two far, then move the tail closer to the head.

Let's think about how we might do this. Below is a diagram of all the cases that you need to move the tail:

```
.TTT.
T...T
T.H.T
T...T
.TTT.
```

I chose to break this down by side. The end state is that the tail is in one four positions:

```
.....
..T..
.THT.
..T..
.....
```

Given each side, we are guaranteed one move for the tail: moving inward. Then, based on the perpendicular direction, we see if we need to move laterally to reach this position. We use a `struct Vector` to represent the distance between `head` and `tail`. We use this difference to determine how we must move.

```
struct Point adjust(struct Point head, struct Point tail)
{
    struct Vector distance = build_vector(head, tail);

    // adjust tail location based on vector
```

```
    if (distance.x == 2) {
        ++tail.x;
        if (distance.y == 1)
            ++tail.y;
        if (distance.y == -1)
            --tail.y;
    }
    if (distance.x == -2) {
        --tail.x;
        if (distance.y == 1)
            ++tail.y;
        if (distance.y == -1)
            --tail.y;
    }
    if (distance.y == 2) {
        ++tail.y;
        if (distance.x == 1)
            ++tail.x;
        if (distance.x == -1)
            --tail.x;
    }
    if (distance.y == -2) {
        --tail.y;
        if (distance.x == 1)
            ++tail.x;
        if (distance.x == -1)
            --tail.x;
    }

    return tail;
}
```

This completes the processing of the instructions.

Printing the Flag

Now that our data is properly complete and processed, we must compute the flag. The flag is the number of unique positions that the tail visited. We can do this by sorting the list of points the tail visited. If duplicates are in the list, they will be adjacent. As we iterate across the list, we'll check for unique lists.

First, let's sort the data.

```
qsort(tail_visits, count, sizeof(struct Point), compare);
```

We need to define the comparison function. We'll arbitrarily choose to compare the `x` values first, then the `y` values.

```
int compare(const void* a, const void* b)
{
```

```
    struct Point* first = (struct Point*)a;
    struct Point* second = (struct Point*)b;

    if (first->x < second->x)
        return -1;
    else if (first->x > second->x)
        return 1;
    else {
        if (first->y < second->y)
            return -1;
        if (first->y > second->y)
            return 1;
        else
            return 0;
    }
}
```

Once our data is sorted, we can iterate and count the number of unique points. We'll define a function called `equal` that returns `1` if two points are equal and `0` otherwise.

```
int equal(struct Point a, struct Point b)
{
    return (a.x == b.x && a.y == b.y);
}
```

Then, we check if adjacent points are equal and count the points that are unique.

```
size_t visited = 1;
for (size_t i = 1; i < count; ++i) {
    if (!equal(tail_visits[i], tail_visits[i - 1])) {
        ++visited;
    }
}
```

We can use this to print the flag! Don't forget to free your memory.

```
printf("flag{%zd}\n", visited);

free(chain);
free(tail_visits);
fclose(fp);

exit(EXIT_SUCCESS);
```

Full Solution

The full solution is below:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Point {
    int x;
    int y;
} Point_default = {0, 0};

struct Vector {
    int x;
    int y;
} Vector_default = {0, 0};

struct Vector build_vector(struct Point head, struct Point tail)
{
    struct Vector* ret = malloc(sizeof(struct Vector));
    ret->x = head.x - tail.x;
    ret->y = head.y - tail.y;
    return *ret;
}

int compare(const void* a, const void* b)
{
    struct Point* first = (struct Point*)a;
    struct Point* second = (struct Point*)b;

    if (first->x < second->x)
        return -1;
    else if (first->x > second->x)
        return 1;
    else {
        if (first->y < second->y)
            return -1;
        if (first->y > second->y)
            return 1;
        else
            return 0;
    }
}

int equal(struct Point a, struct Point b)
{
    return (a.x == b.x && a.y == b.y);
}

struct Point adjust(struct Point head, struct Point tail)
{
    struct Vector distance = build_vector(head, tail);

    // adjust tail location based on vector
```

```
    if (distance.x == 2) {
        ++tail.x;
        if (distance.y == 1)
            ++tail.y;
        if (distance.y == -1)
            --tail.y;
    }
    if (distance.x == -2) {
        --tail.x;
        if (distance.y == 1)
            ++tail.y;
        if (distance.y == -1)
            --tail.y;
    }
    if (distance.y == 2) {
        ++tail.y;
        if (distance.x == 1)
            ++tail.x;
        if (distance.x == -1)
            --tail.x;
    }
    if (distance.y == -2) {
        --tail.y;
        if (distance.x == 1)
            ++tail.x;
        if (distance.x == -1)
            --tail.x;
    }

    return tail;
}

int main(void)
{
    FILE* fp = fopen("input.txt", "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    char* line = NULL;
    size_t len = 0;

    // build the chain
    struct Point* chain = malloc(10 * sizeof(struct Point));
    for (size_t i = 0; i < 10; ++i)
        chain[i] = Point_default;

    // list of points
    struct Point* tail_visits = malloc(20000 * sizeof(struct Point));

    // read the directional changes
    size_t count = 0;
    while (getline(&line, &len, fp) != -1) {
        char dir = strtok(line, " ")[0];
        int times = atoi(strtok(NULL, " "));
```

```
        while (times-- > 0) {
            // adjust the head coordinate
            if (dir == 'U')
                ++chain[0].y;
            else if (dir == 'D')
                --chain[0].y;
            else if (dir == 'L')
                --chain[0].x;
            else if (dir == 'R')
                ++chain[0].x;
            else {
                printf("Bad\n");
                exit(EXIT_FAILURE);
            }

            // adjust the links of the chain
            for (size_t i = 0; i < 9; ++i)
                chain[i + 1] = adjust(chain[i], chain[i + 1]);

            // record tail location in list
            tail_visits[count++] = chain[9];
        }
    }

    // sort the list of points
    qsort(tail_visits, count, sizeof(struct Point), compare);

    // count non-duplicates
    size_t visited = 1;
    for (size_t i = 1; i < count; ++i) {
        if (!equal(tail_visits[i], tail_visits[i - 1])) {
            ++visited;
        }
    }

    printf("flag{%zd}\n", visited);

    exit(EXIT_SUCCESS);
}
```