Mountain Climbers

Problem Description

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

You are going to write a program which opens the file `input.txt` and finds the shortest path from the starting position to the end position. The path is constrained by the elevation of the surrounding area.

- The contour map is broken into a grid; the elevation of each square on the grid is given by a single lowercase letter, where `a` is the lowest elevation and `z` is the highest elevation.
- Also included in the heightmap are your current position (`S`) and the target position for the best view (`E`). Your current position has elevation `a` and the target position has elevation `z`.
- You would like to reach `E`, but to save energy for the rest of the trip home, want to do it in as few steps as possible. During each step, you can move exactly one square up, down, left, or right. Because a certain MIDN forgot their climbing gear at home, the elevation of the destination square can be ***at most one higher*** than the elevation of the current square. Meaning, if you are at an elevation of height `m`, you can step to `n` but not to `o`. This also means that the elevation of the destination square can be much lower than the elevation of your current square.

You have two parts to this challenge:

1. Find the fewest number of steps to move from starting position `S` to the target location.
2. Find the fewest number of steps to move from any base position `a` to the target location.

Concatenate these answers in flag braces separated by a semicolon (`flag{part1:part2}`).

Walkthrough

This is a challenging problem that is aptly worth the points it is worth. This challenge tests your ability to implementing a BFS algorithm with proper backtracking to find the shortest path.

We are using a DFS to solve this problem. Let's talk data structures:

- I chose to simply store the current length as an `int`. Realistically, I could have used an unsigned integer, but with backtracking algorithms I like to have a sanity check that the number won't underflow.
- We need to store the grid map. I chose a 2D static array for this since the size is unchanging. I ended up making the variable global so any function can access it.
- At each state, we need to store the current queue of positions. I used a linked list to hold this.
- We also need to store the visited positions. I used a linked list for this as well.

For the linked list, we need to know the position and the least number of steps to get to that position. I made a struct to hold this information.

```
typedef struct node {
    int x, y, steps;
```

```
    struct node *next;
} node;
```

We'll need to keep a global linked list containing the queue of nodes plus the visited nodes.

```
node *queue, *visited;
```

We also want to know the height and width of the board. We'll store this globally too so every function can see it.

```
int height, width;
```

With this out of the way, we can start opening the file and initializing data. We'll open the file as normal.

```
// Open file for reading
FILE *file = fopen("input.txt", "r");
if (file == NULL) {
    printf("Could not open file.\n");
    exit(EXIT_FAILURE);
}
```

Reading the File

Once we have the file opened, we need to initialize the data. This simply involves reading the array into the `map` we created. This time, I chose to use `fscanf()` to get the string, but `getline()` works too.

```
// Read data for file
int index = 0;
map[index] = malloc(100);
while (fscanf(file, "%s", map[index]) != EOF) {
    map[++index] = malloc(100);
}
```

Once this is done, we must initialize the height and width of the map.

```
// Declare height and width of the heightmap
height = index;
width = strlen(map[0]);
```

Once this is initialized, the processing can begin. We do almost all this in sub-functions, so we'll discuss the looping mechanism in this part and then the processing in the next.

We must parse through all the data for both parts. In each part, we check for something different:

- Part 1: We check for the starting position `S`.
- Part 2: We check for any `a` position.

For the BFS algorithm, we simply check to ensure that any result that finishes is less than the current best case. If it is, we update the best case.

```
// Find Starting coordinates and calculate shortest path
int part1 = INT_MAX;
int part2 = INT_MAX;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        // part 1 - provided start point
        if (map[y][x] == 'S') {
            if (BFS(x, y) < part1)
                part1 = BFS(x, y);
        }
        // part 2 - any 'a' start point
        if (map[y][x] == 'a') {
            if (BFS(x, y) < part2)
                part2 = BFS(x, y);
        }
    }
}
```

From here, we can parse the data and perform the BFS algorithm.

Parsing the Data

Here comes the challenge. We need to define a function `BFS()` that takes a coordinate and returns the shortest path to the end.

The first thing we must do is free `queue` and `visited` because we are starting a new BFS algorithm. This ensures the lists are empty.

```
// Free memory of the queue and visited
free_memory(queue);
free_memory(visited);
queue = visited = NULL;
```

Next, we need to make a new node for the starting position. We'll build nodes often enough that it's worthy of its own function. This function should take the three values we need to make a node, allocate a new one, and return a pointer to the node.

```
node *buildNode(int x, int y, int steps)
{
    node *new_node = malloc(sizeof(node));
```

```
    new_node->x = x;
    new_node->y = y;
    new_node->steps = steps;
    new_node->next = NULL;
    return new_node;
}
```

We'll use this to make a new node for the starting position. At the same time, we'll start our queue with this node.

```
// Create new node
node *start = buildNode(x, y, 0);
// Add starting node to the queue
queue = start;
```

We are going to run the BFS algorithm until the queue is empty. When the queue is empty, we have no more nodes to check. Since BFS doesn't require backtracking, we can make this a simple `while` loop.

```
while (queue != NULL)
{
    /* BFS algorithm */
}
```

We need to get the data from the current node. We'll store this in variables for ease of use.

```
// Get data from current node
int x = queue->x;
int y = queue->y;
int value = getValue(x, y);
int steps = queue->steps;
```

We introduce a new function `getValue` that finds the value of that position in the map.

```
int getValue(int x, int y)
{
    char value = map[y][x];
    switch (value) {
        case 'S':
            return 0;
        case 'E':
            return 25;
        default:
            return value - 'a';
    }
}
```

> *Why do we define 'S' as 0 and 'E' as 25?* In this case, these represent the start and end positions. We want to ensure that the start position is always the lowest value and the end position is always the highest value. This ensures that we don't backtrack to a position we've already been to.

We need to remove the current node from the queue. We should also add the current node to the visited list.

```
// Remove node from the queue
node *current = queue;
queue = queue->next;
current->next = NULL;

// Add current node to the visited queue
current->next = visited;
visited = current;
```

We need to iterate through the neighbors of the current node. We can define `dx` and `dy` to represent the directional arrays for the neighbors.

```
// Neighbors coordinates
int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};
```

At each iteration, we need to do the following checks:

- Get the coordinates of the neighbor
- Check if the neighbor is in the map. If so:
    - Check if the node is the goal node (`E`). If so, return the number of steps.
    - Check if the node is movable. If so:
        - Build a new node for the neighbor.
        - Check if the node is already in the queue. If not, add it to the queue.
        - If it is, free the node.

```
// Find neighbors
for (int i = 0; i < 4; i++) {
    // Get neighbor coordinates
    int nx = x + dx[i];
    int ny = y + dy[i];

    // Check if neighbor is in the map
    if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
        // Check if node is the goal
        if (value >= 24 && map[ny][nx] == 'E')
            return steps + 1;
```

```
            // check if neighbor is movable
            int neighbor_value = getValue(nx, ny);
            if (neighbor_value <= value + 1) {
                node *n = buildNode(nx, ny, steps + 1);

                // Check if node is already in the queue
                if (!inQueue(n))
                    // Add node to the queue
                    enqueue(&queue, n);
                else
                    free(n);
            }
        }
    }
```

We made two new functions here.

- The first function is `inQueue()` that checks if the node is in the queue.

```
int inQueue(node *curr)
{
    // Check if node is in the queue
    node *ptr = queue;
    while (ptr != NULL) {
        if (curr->x == ptr->x && curr->y == ptr->y && curr->steps >=
ptr->steps)
            return 1;
        ptr = ptr->next;
    }

    // Check if node is in the visited queue
    ptr = visited;
    while (ptr != NULL) {
        if (curr->x == ptr->x && curr->y == ptr->y && curr->steps >=
ptr->steps)
            return 1;

        ptr = ptr->next;
    }
    return 0;
}
```

- The second function is `enqueue()` that adds a node to the back of the queue.

```
void enqueue(node **list, node *n)
{
    if (*list == NULL) { // If list is empty
        *list = n;
    } else { // else add node to the end of the list
```

```
            node *current = *list;
            while (current->next != NULL)
                current = current->next;

            current->next = n;
        }
    }
```

If we reach the end of the queue and did not return a value, this means there was no path to the end. In this case, we should return `INT_MAX` to indicate that there is no path.

This is everything we need to execute a BFS algorithm. We can use this to print the flag.

Printing the Flag

To print the flag, we simply take the answers from the two parts and combine them.

```
printf("flag{%d:%d}\n", part1, part2);
```

Once we're done, we free the memory. `map` is a 2D array, so we need to free each row individually. We also need to free our linked lists and close the file.

```
// Free memory
for (int i = 0; i < index + 1; i++)
    free(map[i]);

free_memory(visited);
free_memory(queue);
fclose(file);
```

We define `free_memory()` to free a Linked List.

```
void free_memory(node *list)
{
    // Free memory of the list
    while (list) {
        node *temp = list;
        list = list->next;
        free(temp);
    }
}
```

Running this method will return the flag! Note that this method will take some time to run depending on the machine. This method has to run a BFS algorithm for every `a` in the map. BFS is an `O(V + E)` algorithm, so the overall time is `O(V + E * a)`. This is a very large number, so it will take some time to run.

Full Solution

The full solution is below:

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

typedef struct node {
    int x, y, steps;
    struct node *next;
} node;

// Global variable declaration
char *map[50];
node *queue, *visited;
int height, width;

// Functions declarations
int BFS(int x, int y);
int inQueue(node *n);
void free_memory(node *list);
int getValue(int x, int y);
node *buildNode(int x, int y, int steps);
void enqueue(node **list, node *n);

int BFS(int x, int y)
{
    // Free memory of the queue and visited
    free_memory(queue);
    free_memory(visited);
    queue = visited = NULL;

    // Create new node
    node *start = buildNode(x, y, 0);

    // Neighbors coordinates
    int dx[] = {1, 0, -1, 0};
    int dy[] = {0, 1, 0, -1};

    // Add starting node to the queue
    queue = start;

    while (queue != NULL)
    {
        // Get data from current node
        int x = queue->x;
        int y = queue->y;
        int value = getValue(x, y);
        int steps = queue->steps;
```

```
            // Remove node from the queue
            node *current = queue;
            queue = queue->next;
            current->next = NULL;

            // Add current node to the visited queue
            current->next = visited;
            visited = current;

            // Find neighbors
            for (int i = 0; i < 4; i++) {
                // Get neighbor coordinates
                int nx = x + dx[i];
                int ny = y + dy[i];

                // Check if neighbor is in the map
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    // Check if node is the goal
                    if (value >= 24 && map[ny][nx] == 'E')
                        return steps + 1;

                    // check if neighbor is movable
                    int neighbor_value = getValue(nx, ny);
                    if (neighbor_value <= value + 1) {
                        node *n = buildNode(nx, ny, steps + 1);

                        // Check if node is already in the queue
                        if (!inQueue(n))
                            // Add node to the queue
                            enqueue(&queue, n);
                        else
                            free(n);
                    }
                }
            }
        }
        return INT_MAX;
    }

    int inQueue(node *curr)
    {
        // Check if node is in the queue
        node *ptr = queue;
        while (ptr != NULL) {
            if (curr->x == ptr->x && curr->y == ptr->y && curr->steps >= ptr-
    >steps)
                return 1;
            ptr = ptr->next;
        }

        // Check if node is in the visited queue
        ptr = visited;
        while (ptr != NULL) {
            if (curr->x == ptr->x && curr->y == ptr->y && curr->steps >= ptr-
```

```
>steps)
            return 1;

        ptr = ptr->next;
    }
    return 0;
}

void free_memory(node *list)
{
    // Free memory of the list
    while (list) {
        node *temp = list;
        list = list->next;
        free(temp);
    }
}

int getValue(int x, int y)
{
    char value = map[y][x];
    switch (value) {
        case 'S':
            return 0;
        case 'E':
            return 25;
        default:
            return value - 'a';
    }
}

node *buildNode(int x, int y, int steps)
{
    node *new_node = malloc(sizeof(node));
    new_node->x = x;
    new_node->y = y;
    new_node->steps = steps;
    new_node->next = NULL;
    return new_node;
}

void enqueue(node **list, node *n)
{
    if (*list == NULL) { // If list is empty
        *list = n;
    } else { // else add node to the end of the list
        node *current = *list;
        while (current->next != NULL)
            current = current->next;

        current->next = n;
    }
}
```

```c
int main(void)
{
    // Open file for reading
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Could not open file.\n");
        exit(EXIT_FAILURE);
    }

    // Read data for file
    int index = 0;
    map[index] = malloc(100);
    while (fscanf(file, "%s", map[index]) != EOF) {
        map[++index] = malloc(100);
    }

    // Declare height and width of the heightmap
    height = index;
    width = strlen(map[0]);

    // Find Starting coordinates and calculate shortest path
    int part1 = INT_MAX;
    int part2 = INT_MAX;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            // part 1 - provided start point
            if (map[y][x] == 'S') {
                if (BFS(x, y) < part1)
                    part1 = BFS(x, y);
            }
            // part 2 - any 'a' start point
            if (map[y][x] == 'a') {
                if (BFS(x, y) < part2)
                    part2 = BFS(x, y);
            }
        }
    }

    // Print results to console
    printf("flag{%d:%d}\n", part1, part2);

    // Free memory
    for (int i = 0; i < index + 1; i++)
        free(map[i]);

    free_memory(visited);
    free_memory(queue);
    fclose(file);
}
```