```
using PlutoUI   , SparseArrays
```

```
parse_line (generic function with 1 method)
```

```
parse_file (generic function with 1 method)
```

```
@enum CaveType LargeCave SmallCave
```

```
struct Cave
    symbol::String
    cave_type::CaveType
    connections::Vector{Cave}
end
```

```
struct CaveGraph
    graph::Dict{String, Cave}
end
```

```
get_cave_type (generic function with 1 method)
```

```
get_cave_type(symbol) = match(r"^[A-Z]+$", symbol) != nothing ? LargeCave : SmallCave
```

```
create_graph (generic function with 1 method)
```

```
function create_graph(connections)::CaveGraph
    graph = CaveGraph(Dict{String, Cave}())

    for (lhs, rhs) in connections
        lhs_cave = get(graph.graph, lhs, Cave(lhs, get_cave_type(lhs), []))
        rhs_cave = get(graph.graph, rhs, Cave(rhs, get_cave_type(rhs), []))

        push!(lhs_cave.connections, rhs_cave)
        push!(rhs_cave.connections, lhs_cave)

        graph.graph[lhs] = lhs_cave
        graph.graph[rhs] = rhs_cave
    end

    return graph
end
```

```
("start", "end")
```

Problem 1

```
calculate_unique_paths (generic function with 1 method)
```

```
function calculate_unique_paths(cave::Cave, caves_explored::Dict{String, Bool})
    if cave.symbol == END_SYMBOL
        return 1
    end

    caves_explored[cave.symbol] = true

    unique_path = 0
    for connected_cave in cave.connections
        if connected_cave.cave_type == SmallCave &&
get(caves_explored,connected_cave.symbol, false) == true
            # Skip small caves that have already been explored
            continue
        end
        unique_path += calculate_unique_paths(connected_cave, caves_explored)
    end

    caves_explored[cave.symbol] = false
    return unique_path
end
```

```
calculate_unique_paths (generic function with 2 methods)
```

```
function calculate_unique_paths(graph::CaveGraph)
    caves_explored = Dict{String, Bool}()
    start_cave = graph.graph[START_SYMBOL]
    caves_explored[START_SYMBOL] = true

    return calculate_unique_paths(start_cave, caves_explored)
end
```

```
4754
```

```
  0.000035 seconds (270 allocations: 19.891 KiB)
  0.001535 seconds (7 allocations: 1.422 KiB)
```

```
with_terminal() do
    open("./Day12/prob_input.txt") do io
        connections = parse_file(io)
        @time graph = create_graph(connections)
        @time calculate_unique_paths(graph)
    end
end
```

Problem 2

```
fib (generic function with 1 method)
```

```
fib(n) = n <= 1 ? n : fib(n-1) + fib(n-2)
```

In this problem we have the option of visting **only one** small cave twice but all other small caves in the path should be visted once. The only exceptions are `"start"` and `"end"` state.

What we do is that we we discover path we maintain a state in that path context which checks if we already visited a small cave in previous part of path more than once. If we have then don't try going to another small cave twice.

In the rescursive function that state is represented as `used_extra_hop` .

We also change our exploration dictionary from `Bool` to `Int` as in case of `Bool` if we visit that node twice then it can be set to false when we get out of it once, even though its still in the previous path so that can lead to an infinite recursion

```
calculate_unique_paths_prob2 (generic function with 2 methods)
```

```
function calculate_unique_paths_prob2(cave::Cave, caves_explored::Dict{String, Int},
used_extra_hop::Bool)
    if cave.symbol == END_SYMBOL
        # If we got to end then stop further path finding as
        # it's the terminal state
        return 1
    end

    caves_explored[cave.symbol] += 1

    unique_path = 0
    for connected_cave in cave.connections
        if connected_cave.symbol == START_SYMBOL
            # Skip going to start as it can only be visited once
            continue
        end

        added_extra_hop = used_extra_hop
        # 1. If the cave is `LargeCave` it will fail all these conditions
        #    and we can continue with exploring path.
        # 2. If its a `SmallCave` then
        #    a. If we have visited this node once or more AND we have previously
        #       explored a `SmallCave` twice then skip. (BTW if it has been
        #       visited twice then `added_extra_hop` will be `true` but if its
        #       visited only once then it could be either `true` or `false)
        #    b. If the small cave has only been visited once AND we have previously
        #       not explored a `SmallCave` twice then explore it.
        #    c. If it doesnt meet **a** and **b** then it means its either `LargeCave`
        #       or a `SmallCave` we have not explored before or if we have explored
        #       before then we can't explore twice because this or some other cave
        #       was explored twice before.
        if (connected_cave.cave_type == SmallCave &&
            get(caves_explored,connected_cave.symbol, 0) >= 1 &&
            added_extra_hop == true)
            continue
        elseif (connected_cave.cave_type == SmallCave &&
            get(caves_explored,connected_cave.symbol, 0) == 1 &&
            added_extra_hop == false)
            added_extra_hop = true
        end

        unique_path += calculate_unique_paths_prob2(
            connected_cave, caves_explored, added_extra_hop)
    end

    caves_explored[cave.symbol] -= 1
    return unique_path
end
```

```
calculate_unique_paths_prob2 (generic function with 3 methods)
```

```
function calculate_unique_paths_prob2(graph::CaveGraph)
    caves_explored = Dict{String, Int}()
    map(sym -> caves_explored[sym] = false, collect(keys(graph.graph)))

    start_cave = graph.graph[START_SYMBOL]

    return calculate_unique_paths_prob2(start_cave, caves_explored, false)
end
```

```
143562
```

```
  0.000034 seconds (270 allocations: 19.891 KiB)
  0.066241 seconds (9 allocations: 2.250 KiB)
```

```
with_terminal() do
    open("./Day12/prob_input.txt") do io
        connections = parse_file(io)
        @time graph = create_graph(connections)
        @time calculate_unique_paths_prob2(graph)
    end
end
```