Hill Climber

The first few functions will be identical to the ones created for the Random Search. Only the neighbourhood and the main loop are new.

Constants and Imports

We will need various constants such as the target string along with imports for the packages we are going to make use of.

```
import string
import random
random.seed(42) # initialse and make repeatable
```

Generate a Random Solution

Create a random string of 1's and 0's of length defined in the parameter.

```
# Generate a random string of 1's and 0's as defined by the parameter
# Input parameters: length - the length of the string to generate
# Returns: The randomly generated string
def gen_random_string(length):
  return ''.join(random.choice('01') for _ in range(length));




# test... run this a few times to check the output is as expected
# print(gen_random_string(1))
print(gen_random_string(5))
print(gen_random_string(10))
```

```
00100
0001000000
```

Fitness Function

This defines how we score or evaluate a solution. The fitness function has an important role in guiding the search towards the solution (but not for the random search, obviously), so it is essential that we are able to identify when one solution is better than another.

There are various alternatives but the more information you provide the quicker the search is likely to be.

One important point to consider is the relationsip between the fitness function and the neighbourhood: a move in the the neighbourhood (either closer to or further away from the solution) should be reflected in the value of the fitness function. This is particularly important in the case of the hill climber as if there is no change seen in the neighbourhood then the algorithm cannot proceed and will terminate, assuming it has found the best solution.

```
# Function to calculate a candidates solution fitness - solutions closer to the goal should score higher
# Input parameters: candidate solution
# Returns: fitness of the candidate solution
def fitness(solution):
  return solution.count('1')


    # Your code goes here

# test the above (precise results will depend on your function and may vary with string length)
print(fitness("1111111111")) # output should be the highest score
print(fitness("1010110010")) # output should be a mid-range score
print(fitness("0000000000")) # output should be the lowest score
print(fitness("11111")) # output should be the highest score
print(fitness("10101")) # output should be a mid-range score
print(fitness("00000")) # output should be the lowest score
print(fitness("1")) # output should be the highest score
print(fitness("0")) # output should be the lowest score
```

```
10
5
0
5
3
0
1
0
```

Function to generate the neighbours of the solution

The current solution is input as a parameter and a list of neighours returned. There are several alternatives to defining the neighbourhood for this problem - some of which will move quite quickly towards the solution (but could be extensive and take time to evaluate), others are slightly slower (but may be cheaper and quicker to evaluate). Feel free to explore alternatives if you have time.

```
# Input parameters: The current solution (as a string)
# Returns: A list of the neighbours of the current solution
def generate_neighbours(solution):
    # Your code goes here
    neighbours = []
    current_length = len(solution)
    for i in range(current_length):
      if solution[i] == "0":
        neighbours.append(solution[:i] + "1" + solution[i+1:])
      else:
        neighbours.append(solution[:i] + "0" + solution[i+1:])
    return neighbours



# test
generate_neighbours("1")
generate_neighbours("0")
generate_neighbours("101")
generate_neighbours("100")
generate_neighbours("001")
generate_neighbours("10101")
```

```
['00101', '11101', '10001', '10111', '10100']
```

Main Hill Climber Loop

The basic loop of an algorithm is as follows:

- Generate a random solution as the current solution
- Loop
  - Identify and evaluate the neighbours of the current solution
  - Set the current solution to be a fitter solution (there are different possible ascent strategies)
- Until no fitter solution can be identified
- Return the current solution

Depending on he approach you adopt you may or may not need a random restart for this problem.

```
# Input parameters: The length of the string to be "maxed"
# Returns: A tuple with the best solution generated and its fitness value
# Note that it can be helpful to also print out the intermediate values of the solution
# and its fitness so that it is possible to see how the search is progessing
def hill_climber(length):
    current_solution = gen_random_string(length)
    current_fitness = fitness(current_solution)

    print(f"Start Solution: '{current_solution}' | Fitness: {current_fitness}")

    while True:
        neighbours = generate_neighbours(current_solution)

        best_neighbour = None
        best_neighbour_fitness = current_fitness

        for neighbour in neighbours:
            neighbour_fitness = fitness(neighbour)

            if neighbour_fitness > best_neighbour_fitness:
                best_neighbour_fitness = neighbour_fitness
                best_neighbour = neighbour

        if best_neighbour_fitness > current_fitness:
            current_solution = best_neighbour
            current_fitness = best_neighbour_fitness
            print(f"Step: {current_solution} | Fitness: {current_fitness}")
        else:
            print("Optima solution reached")
            return (current_solution, current_fitness)
```

```
# tests
hill_climber(20)
```

```
Start Solution: '01011001110010010111' | Fitness: 11
Step: 11011001110010010111 | Fitness: 12
Step: 11111001110010010111 | Fitness: 13
Step: 11111101110010010111 | Fitness: 14
Step: 11111111110010010111 | Fitness: 15
Step: 11111111111010010111 | Fitness: 16
Step: 11111111111110010111 | Fitness: 17
Step: 11111111111111010111 | Fitness: 18
Step: 11111111111111110111 | Fitness: 19
Step: 11111111111111111111 | Fitness: 20
Optima solution reached
('11111111111111111111', 20)
```