GENETIC ALGORITHM:

```
import random

POPULATION_SIZE = 6
GENES = [str(i) for i in range(10)]
TARGET = 30
MAX_ITERATIONS = 50

class Chromosome:
  def __init__(self, genes):
     self.genes = genes
     self.fitness = self.calculate_fitness()

  def calculate_fitness(self):
     a, b, c, d = map(int, self.genes)
     return abs(a + 2*b + 3*c + 4*d - TARGET)


def selection(population):
  sorted_population = sorted(population, key=lambda x: x.fitness)
  return sorted_population[:2]


def crossover(parent1, parent2):
  # Two-point crossover
  crossover_points = sorted([random.randint(1, len(parent1.genes) - 1) for _ in range(2)])
  child_genes = (
        parent1.genes[:crossover_points[0]] +
        parent2.genes[crossover_points[0]:crossover_points[1]] +
        parent1.genes[crossover_points[1]:]
  )
  return Chromosome(child_genes)


def mutation(child):
  # Randomly mutate one or two genes
  mutated_gene_indices = random.sample(range(len(child.genes)), random.randint(1, 2))
  for index in mutated_gene_indices:
     child.genes[index] = random.choice(GENES)
  return child


def main():
```

```
    # Initialize population with specific chromosomes
    population = [
        Chromosome(['1', '2', '3', '4']),
        Chromosome(['5', '6', '7', '8']),
        Chromosome(['9', '0', '1', '2']),
        Chromosome(['3', '4', '5', '6']),
        Chromosome(['7', '8', '9', '0']),
        Chromosome(['1', '2', '3', '4'])
    ]

    iteration = 0
    best_solution = None

    while iteration < MAX_ITERATIONS:
        parent1, parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutation(child)

        population.remove(max(population, key=lambda x: x.fitness))
        population.append(child)

        best_chromosome = min(population, key=lambda x: x.fitness)
        if best_solution is None or best_chromosome.fitness < best_solution.fitness:
            best_solution = best_chromosome

        iteration += 1

    print("Initial Chromosomes:")
    for i, chromosome in enumerate(population, 1):
        print(f"Chromosome {i}: {chromosome.genes}")

    print("\nFinal Chromosomes:")
    for i, chromosome in enumerate(population, 1):
        print(f"Chromosome {i}: {chromosome.genes}")

    print(f"Best Chromosome: {best_solution.genes}")
    print(f"Best Solution: {sum((i + 1) * int(gene) for i, gene in enumerate(best_solution.genes))}")
    print(f"Values of ['a', 'b', 'c', 'd']: {tuple(int(gene) for gene in best_solution.genes)}")
    print(f"Number of Iterations: {iteration}")


if __name__ == "__main__":
    main()
```

OUTPUT:

```
/usr/local/bin/python3.11 /Users/yash/PycharmProjects/tsec/temp.py
Initial Chromosomes:
Chromosome 1: ['1', '2', '3', '0']
Chromosome 2: ['8', '2', '3', '1']
Chromosome 3: ['5', '2', '3', '4']
Chromosome 4: ['9', '2', '3', '4']
Chromosome 5: ['1', '2', '9', '4']
Chromosome 6: ['1', '8', '8', '0']

Final Chromosomes:
Chromosome 1: ['1', '2', '3', '0']
Chromosome 2: ['8', '2', '3', '1']
Chromosome 3: ['5', '2', '3', '4']
Chromosome 4: ['9', '2', '3', '4']
Chromosome 5: ['1', '2', '9', '4']
Chromosome 6: ['1', '8', '8', '0']
Best Chromosome: ['1', '2', '3', '4']
Best Solution: 30
Values of ['a', 'b', 'c', 'd']: (1, 2, 3, 4)
Number of Iterations: 50

Process finished with exit code 0
```

HILL CLIMBING:

```
import random
import math

# Objective function to be maximized
def objective_function(x):
  return math.sin(x)
# Generate initial solution randomly
def generate_initial_solution():
  return random.uniform(-math.pi, math.pi)
# Generate neighbour solutions
def generate_neighbours(solution):
  neighbours = []
  for delta in [-0.1, 0.1]:
     neighbours.append(solution + delta)
  return neighbours
# Get highest quality neighbour of current solution
def get_best_neighbour(neighbours):
  best_neighbour = neighbours[0]
  best_quality = objective_function(best_neighbour)
  for neighbour in neighbours[1:]:
     neighbour_quality = objective_function(neighbour)
     if neighbour_quality > best_quality:
        best_quality = neighbour_quality
        best_neighbour = neighbour
  return best_neighbour
# Hill climbing algorithm
def hill_climbing():
  current_solution = generate_initial_solution()
  while True:
     neighbours = generate_neighbours(current_solution)
     best_neighbour = get_best_neighbour(neighbours)
     if objective_function(best_neighbour) <= objective_function(current_solution):
        return current_solution
     current_solution = best_neighbour

# Main
def main():
  best_solution = hill_climbing()
  print("Best solution found:", best_solution)
  print("Objective function value:", objective_function(best_solution))

if __name__ == "__main__":
  main()
```

OUTPUT:

```
/usr/local/bin/python3.11 /Users/yash/PycharmProjects/tsec/temp.py
Best solution found: 1.5522719605474555
Objective function value: 0.9998284288339007

Process finished with exit code 0
```