Exercises for lesson 6: Algorithm Design, Part I

**Exercise 6.1**

1. Prove the following loop invariants for the algorithm below, given that the input n is an integer larger than or equal to 1:

I1: n, r and j are integers

I2: r = j(j+1)/2

1. Use the above to conclude that the algorithm returns .

algorithm(n):

j = 0

r = 0

while j < n:

j = j + 1

r = r + j

r = r + r – n

return r

**Exercise 6.2**

Below are two approaches (algorithm1 and algorithm2) to the traveling salesman problem – for each, indicate whether they are brute force, randomized or greedy, state the time complexity and whether they guarantee that we find the optimal solution.

Assume that:

* L = [(longitude of city 1, latitude of city 1), …] is a list of city coordinates
* total\_dist(L) calculates the total distance of the tour defined by the visiting the cities in the order they appear in L
* dist(L, i, j) calculates the distance between city i and city j
* permute(L) returns all permutations of the list.

|  |  |
| --- | --- |
| algorithm1(L):  solution = L  score = infinity  for l in permute(L):  if total\_dist(l) < score:  solution = l  score = total\_dist(l)  return (solution, score) | algorithm2(L):  solution = empty list  score = 0  step = 0  next\_city = 1  while length(solution) < length(L):  solution.append(L[next\_city])  score = score + step  step = infinity  for i from 2 to length(L):  if dist(L, next\_city, i) < step:  step = dist(L, next\_city, i)  next\_city = L[i]  return (solution, score) |
|  |  |

**Exercise 6.3**

Write the pseudocode for the greedy approach to the coin-change problem.

**Exercise 6.4**

The (foolish) sorting algorithm BogoSort goes like this:

BogoSort(L):

while L is not sorted:

shuffle(L)

What type of algorithm is this? Is it guaranteed to find the correct solution? Is it guaranteed to terminate in finite time?

**Exercise 6.5**

Suppose you have a set of jobs, each with a start time and an end time. You want to schedule these jobs on a single machine, but you can only work on one job at a time. Additionally, you can't start a new job until the previous one has finished.

Write a greedy algorithm that finds the maximum number of jobs you can complete. Your algorithm should take in a list of jobs, where each job is represented as a tuple of two integers: the start time and the end time. You can assume that the jobs are already sorted by end time.