Data Structures and Algorithms – Write an Algorithm

The Problem

The problem this algorithm will solve is *Goodland Electricity (hackerrank.com/challenges/pylons/problem).* Goodland is a country with cities along a line. Each city is 1 unit away from adjacent cities. Each city has a power plant; however, we want to find the fewest number of power plants required to power all the cities. The algorithm will determine this number or, if this can't be done, return *-1*.

The algorithm requires three inputs: *n*, number of cities, *k*, range power plants can provide electricity, and *arr*, city data. The number of cities and the range power plants can provide electricity must both be integers, the range being less than or equal to the number of cities, the city data must be a list consisting of only 0 and 1. This denote whether a city has a power plant or not. If a city has a value of 1 then that city has a power plant, a value of 0 means it doesn't.

The algorithm will output one integer, printing the number of power plants required or -1 meaning that there is no valid solution.

Shown below are several examples of inputs and outputs expected from this algorithm.

- *n* = 6, *k* = 2, *arr* = [0, 1, 1, 1, 1, 0], expected output = 2.
- *n* = 7, *k* = 2, *arr* = [0, 1, 0, 0, 0, 1, 0], expected output = -1.
- *n* = 10, *k* = 3, *arr* = [0, 1, 0, 0, 0, 1, 1, 1, 1, 1], expected output = 3.

How the Algorithm Works

For this, *tuple* was chosen as it is the most appropriate data structure. We are required to use an ordered data structure which will have to contain duplicates for our input *arr*. We also don't need to be able to change the data structure once it has been input into the algorithm, therefore *tuple*, unlike *list, set, dictionary,* meets all the criteria to be used as it is the only data structure which is ordered, unchangeable, and allow for duplicate members.

The algorithm will start with the input value and five extra values, *i, j, t, loc, and flag,* all initially set equal to 0. The value *i* denotes the city currently being looked at (e.g. 0 means the first city is being looked at, 2 means the third city is being looked at, etc.), *j* denotes the last covered by the range of *k* from *i*, *t* keeps track of the number of plants required, *loc* specifies the location of the plants needed to be used, and *flag* is used only where there is no valid solution to terminate the program.

The first step is adding one to the number of power plants required, *t*, and calculating *j=i+k-1*, telling us the location of the farthest city in this plant's radius. (We subtract 1 from *i+k* as the range of the power plants is not inclusive of the farthest value.)

Next the algorithm goes through checks. If *j* is less than *n*, change *j* equal to *n-1* to put it back into a valid area, if *loc* is less than or equal to *j* which is less than *n* and *arr[j]* is equal to 0, subtract one from *j*, if *j* is less than *loc*, print *-1*, as the problem cannot be solved, and change *flag* to 1 in order to terminate the algorithm, or else if none of these criteria are met, set *loc* equal to *j+1*, add *k* to *j*, set *i* equal to *j*, and then restart the while loop. This is repeated until *i* is n longer less than *n*, all cities have been included in the range of at least one plant, the algorithm will print the number of plants required.

Implementation of Algorithm

```
1 def pylons(numOfCities, towerRange, towersPerCity):
2     i,j,numTowersRequired,location,endFlag=0,0,0,0,0
3     while(i<numOfCities):
4         numTowersRequired+=1
5         j=i+towerRange-1
6         if(j>numOfCities):
7             j=numOfCities-1
8         while (location<=j<numOfCities and towersPerCity[j]==0):
9             j-=1
10        if(j<location):
11            print("-1 ")
12            endFlag=1
13            break
14        else:
15            location=j+1
16            j+=towerRange
17            i=j
18    if endFlag==0:
19        print(numTowersRequired)
```

Shown above is the completed implementation of the algorithm in Python. Following along the code it can be seen to have followed the step-by-step walkthrough written in the previous section of this document. Some variable names have been changed for the sake of clarity as follows: *n* = *numOfCities*, *k* = *towerRange*, *arr* = *towersPerCity*, *t* = *numTowersRequired*, *loc* =, and *flag* = *endFlag.*

```
21 #testcase0, expected output = 2, actual output = 2
22 pylons(6,2,(0,1,1,1,1,0))
23 #testcase1, espected output = -1, actual output = -1
24 pylons(7,2,(0,1,0,0,0,1,0))
25 #testcase 2, expected output = 3, actual output = 3
26 pylons(10,3,(0,1,0,0,0,1,1,1,1,1))
```

Above is code used to call the algorithm. These are test cases used to test the algorithm and ensure its accuracy. The first number in each line is *n* or *numOfCities*, the second number in each is *k* or *towerRange*, and the rest of the numbers contained in parentheses are the tuple *arr* or *towersPerCity*. The text above each test case includes the expected output and the actual output of the algorithm. As the actual output matches the expected output we can be assured that the algorithm functions correctly. Shown below is the console display after running the algorithm with each of these test cases as additional evidence.

```
In [2]: runfile('C:/Users/Adam/Documents/Uni/DSAA1.py', wdir='C:/Users/Adam/Documents/Uni')
2
-1
3
```

Evaluation of Algorithm Used

Overall, this is a well-made algorithm which solves the problem effectively and with appropriate parameters, as this code passes all 22 test cases on *hackerrank.com/challenges/pylons/problem* with minor changes to the premade input formatting and changing the *print* statements to *return*.