DESIGN AND ANALYSIS OF ALGORITHMS – CSCI 612

NAME: AADITHYA GOPALAKRISHNA BHARADWAJ

ZID: Z1862641

ASSIGNEMENT NUMBER, SEMESTER: Assignment 6, SPRING 2019

==

1. (5 points) Give an O(n^2) time algorithm to find the longest monotonically increasing sub-sequence of a sequence of n integers.

Answer.)

Step 1:

Consider the input sequence be Se[1], Se[2], Se[3], Se[4], … , Se[n]. Let us assume that there is a value Le[j] (such that the values of j is 1 to n) to be the longest monotonically increasing sub-sequence of the first j letters Se[1], Se[2], Se[3], Se[4], … , Se[j].

It is assumed that the last letter of the sub-sequence is Se[j].

Assume that the position of the last letter before Se[j] in the longest sub-sequence of the first j letters be Po[j] such that Se[j] is the last letter.

R is the resulting sub-sequence.

Step 2:

The recurrence function for computing Le[j] is provided as below:

Le[j] =

The algorithm provided is as follows:

// This loop will run for n times:

for j = 1 to n do

Le[j] = 1;

Po[j] = 0;

// This inner loop will execute for $O(n^2)$ times.

for k = 1 to j - 1 do

if Se[k] < Se[j] && Le[k] ≥ Le[j] then

Le[j] = Le[k] + 1;

Po[j] = k;

end if

end for ; end for

// The largest value of Le[m] in Le[1] … Le[n] should be found.

// Backtracking we get the following

j = 1; k = m do

R[j] = Se[k];

j++;

k = Po[k];

Until k = 0

Output Le[m] and the reverse of R.

Hence, it is evident that the time complexity is $O(n^2)$

==

2. (5 points) Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Answer.)

The array of n points on the real line is A[1..n]: Points A[1] < A[2] < … < A[n]

The array of intervals B[1 … m]: Intervals [B[j], B[j] +1]

Algorithm smallest-set (n)

Step1: m = 1;

Step 2: B[1] = A[1];

Step 3: for j = 1 to n do

Step 4: if (A[j] > B[m] +1)

Step 5: m = m+1; // Adds a new interval

Step 6: B[m] = A[j];
else

Step 7: A[j] is in [B[m], B[m] +1]

Step 8: return m;

Proof of the algorithm:

Theorem:
smallest-set returns the minimal number of unit intervals that cover
A[1], A[2], …, A[n].

Lemma: None of the intervals obtained by smallest-set can move right by any distance, otherwise some points won't be covered.
Proof: Induction on m

Base case: m = 1 and **B[1] = A[1]. If [B[1], B[1]+1]** moves right, then **A[1]**cannot be covered.

Inductive hypothesis: Suppose none of the first m intervals can move right. When **A[j] > B[m] +1**, the last interval **[B[m], B[m]+1]** cannot cover **A[j]** (and it cannot move right). So need a new interval and let it be **[A[j], A[j] +1]**, which cannot move right; otherwise **A[j]** cannot be covered.
==

3.) **Suppose you want to drive from San Francisco to New York on I-80. Your car holds C gallons of gas and gets m miles to the gallon. You are handed a list of the n gas stations that are on I-80 and the price that they sell gas. Let di be the distance of the i-th gas station from SF, and ci be the cost of gasoline at the i-th gas station. Furthermore, you can assume that for any two stations i and j, the distance di-dj between these two stations is a multiple of m. You start out with an empty tank at station 1. Your final destination is gas station n. You need to end at station n with at least 0 gallons of gas.**

Find a polynomial-time dynamic programming algorithm to output the minimum gas bill to cross the country. Analyze the running time of your algorithm in terms of n and C. You do not need to find the most efficient algorithm as long as your solution's running time is polynomial in n and C.

Remember that your car cannot run if your tank ever holds less than 0 gallons of gas. Also, if you decide to get gasoline at a particular station, you needn't fill up the tank; for example, you might decide to purchase only 7 gallons of gas at one station.

Answer.)

Let Mi (g) be the minimum gas bill to reach gas station i with g gallons of gas in the tank (after potentially purchasing gas at station i). The range of the indices is $1 \le i \le n$ and $0 \le g \le C$.

The recursive equation will be written in terms of the number of gallons of gas in the car when leaving station i − 1. Call this number h. Clearly $(di - di{-}1)/m \le h \le C$ otherwise the car cannot reach station i. Also $h \le (di - di{-}1)/m + g$ because we cannot purchase a negative number of gallons at station i.

The recursive equation is

$$M^i(g) = \min_h[M^{i-1}(h) + (g + (di - di-1)/m - h)c_i]$$

where, h runs from (di − di−1)/m to min(C,(di − di−1)/m + g).

The base case is $M^1(g) = c1g$ where $0 \le g \le C$.

The answer will be given by $\min^c_{g=0}(M^n(g))$. One can argue that the cheapest solution will involve arriving at gas station n with 0 gallons in the tank, so the answer is also simply the entry $M^n(0)$.

We choose to evaluate the matrix in increasing order of i. Note that to compute M^i we only need Mi−1 , so the space can be reused. This is demonstrated in the pseudo-code, which uses only two arrays M and N.

GasolineRefilling(n, d[], c[])

```
{
    for g from 0 to C M[g] = c[1]*g
    // base case for i from 2 to n
    {
        for g from 0 to C
        {
            N[g] = infinity // N is a temporary array
            for h from (d[i] - d[i-1])/m to min(C, (d[i] - d[i-1])/m + g)
            {
                 cost = M[h] + (g + (d[i] - d[i-1])/m – h)*c[i]
                if (cost < N[g]) N[g] = cost;
            }
        }
        for g from 0 to C
        M[g] = N[g] // copy entries from the temporary array
    }
    return M[0]
}
```