Dynamic Programming

Raveesh Gupta

February 9, 2022

Chapter 1

Longest Increasing Sub-sequence

The Longest Increasing Sub-sequence (LIS) problem is to find the length of the longest sub-sequence of a given sequence such that all elements of the sub-sequence are sorted in increasing order.

1.1 Numerical Analysis

This is a classical problem in *dynamic programming* to find the longest increasing sub-sequence in a sequence of integers $arr_{1...n}$. First step is to define dp_i.

dp_i := length of longest possible increasing sub-sequence in $arr_{1...n}$

Second step is transition . Definition of a *recurrence relation* is defined using dynamic technology.

$$dp_i = MAX_{arr_j > arr_i}(1 + dp_j), \forall j \epsilon [i+1, n]$$

Final step is to define answer hence $Ans\ Definition$.

$$MAX_{\forall i \epsilon [1,n]}(dp_i)$$

1.2 Non-Numerical Analysis

Define $forw_i$ that will store j ie index of the the next element in the sub-sequence $arr_{1...n}$, then assume k is the index s.t dp_k is the solution for the given problem. Longest Increasing Sub-sequence is,

$$arr[k], forw[arr[k]], forw[forw[arr[k]]]]...tothelengthofdp[k]$$

Chapter 2

Nails on the board

Consider a marked wooden board, at different points n nails are hammered such that ith nail is on position x_i. Given limitless string, you are required to calculate the minimum total length of string such that.

$$\forall i \epsilon [1, n]$$

is connected to some j.

2.1 Observations

1. It is optimal to connect neighbouring nails to avoid overlapping. $i.e$ j will always equal to $i+1$.

2. When $n > 4$ sometimes it is optimal to use 2 strings instead of 3 strings to satisfy the condition.

2.2 Analysis

First step is to define a DP definition.

dp_i := Minimum Possible Length of string to connect first i nails.

Second step is to define a transitive function. DP Recursive Definition case 1.

$$dp_i = MIN(dp_{i-1} + x_i - x_{i-1}, dp_{i-2} + x_i - x_{i-1}),$$

case 2.

$$dp_3 = x_3 - x_1, i = 3$$

case 3.

$$dp_2 = x_2 - x_1, i = 2$$

Final step is answer definition: dp_n.

Chapter 3

Avoid-two-neighbouring-ones problem

Given positive integer n; how many sequences of n zeroes and ones such that no any two ones occur in neighbouring positions?

3.1 Observations

Let $S_{1..n}$ be a good sequence then if S_n is 0 then S_{n-1} can be 1 or 0 thus reducing our problem to solving $S_{1..n-1}$, otherwise if S_n is 1 then S_{n-1} can only be 0 hence reducing our problem to solving $S_{1...n-2}$.

Trivial cases $n = 1$ or $n = 2$ can be seen easily *i.e* $dp_1 = 2$ and $dp_2 = 3$.

3.2 Analysis

First step is to define a DP definition.

Let dp_i := Number of good sequences with length i.

Second step is to define a transitive function. DP Recursive Definition.

$$dp_i = dp_{i-1} + dp_{i-2}, \forall i >= 3$$

Final step is answer definition: dp_n.

Chapter 4

Largest-Common-Sub-sequence LCS

Its a classical problem in which given two sequences

$$A = (A_1, A_2, ..., A_n)$$

and

$$B = (B_1, B_2, ..., B_m)$$

, we are to find such two sequences

$$1 <= i_1 < i_2 < ... < i_k <= n$$

,

$$1 <= j_i < j_2 < ... < j_k <= m$$

for maximal possible k, such that

$$A_{i_1} = B_{j_1}, A_{i_2} = B_{j_2}, ..., A_{i_k} = B_{j_k}.$$

4.1 Observations

1. Its obvious that $dp_{i,j} = 0$ if $i = 0$ or $j = 0$.

2.If $A_i = B_j$ then there is only 1 solution $dp_{i-1,j-1} + 1$ and A_i becomes part of our ans, otherwise if $A_i! = B_j$ then there are 2 cases $dp_{i-1,j}$ or $dp_{i,j-1}$ (these two statements can be true at the same time, but at least one must be true definitely).

4.2 Analysis

First step is to define a DP definition.

Let $dp_{i,j}$:= Length of the largest common sub-sequence in $A_{1...i}$ and $B_{1...j}$.

Second step is to define a transitive function. DP Recursive Definition.
case 1. if $A_i = B_j$, then

$$dp_{i,j} = dp_{i-1,j-1} + 1, \forall i, j > 0$$

case 2. if $A_i ! = B_j$, then

$$dp_{i,j} = MAX(dp_{i-1,j}, dp_{i,j-1}), \forall i, j > 0$$

case 3. if $i = 0$ or $j = 0$, then

$$dp_{i,j} = 0$$

Final step is answer definition: $dp_{n,m}$.

Chapter 5

Pie Progress (FHC)

Some pies are sweet, full of fruit and jam and sugar

Some pies are savory, full of meat and potatoes and spices.

Some pies are in fact not pies at all but tarts or galettes. This probably won't stop you from eating them.

Every single day for N days, you're determined to eat a pie for dinner. Every morning, you'll take a trip to your local pie shop, and buy 0 or more of their pies. Every night, you'll eat one pie that you've bought. Pies never go bad, so you don't need to eat a pie on the same day that you bought it. You may instead eat one that you purchased on an earlier day.

On the i^{th} day, the shop has M pies for sale, with the j^{th} of these pies costing $C_{i,j}$ dollars. You can choose to buy any (possibly empty) subset of them. However, this shop has measures in place to protect itself against crazy pie fanatics buying out its products too quickly. In particular, if you buy p pies on a single day, you must pay an additional tax of p^2 dollars.

5.1 Analysis

First rule of solving is to express the problem in a mathematical expression and try to fit in some existing computational model.

Let val_i and k_i be the total money spent on ith day and no of pies bought on ith day respectively. We don't know val_i and k_i but we know its properties,

$MIN(\sum_{i=0}^{N-1} val_i) \mid \forall j \in [0, N) \sum_{i=0}^{j} k_i >= j + 1 \wedge \sum_{i=0}^{N-1} k_i = N$
Dynamic Programming can solve this problem as on each step we have to choose something to compute some minimum while maintaining some property.

Observation: On each day its optimal to choose the first k_i pies in an non-decreasing order.

Lets define $dp(i, j)$: Minimum cost of choosing j pies in days[0..i]

Let's define the transition:
$dp(i, j) = \forall k \in [0, M] MIN(dp(i-1, j-k) + cost)(j >= i)$

Base cases:
$dp(-1, > 0) = \infty$
$dp(i > -1, < 0) = \infty$
$dp(-1, 0) = 0$

Ans: $dp(n-1, n)$
Complexity: $O(n^2 * m)$

5.2 Implementation

```
int n, m, mat[MAX][MAX], dp[MAX][MAX];

void init(){
    FOR(i, 0, n) FOR(j, 0, n) dp[i][j] = -1;
}

ll DP(int i, int j){
        if((i == -1 && j > 0) || ( i > -1 && j < 0)) return LINF;
        if(i == -1 && j == 0) return 0;
        if(dp[i][j] != -1 ) return dp[i][j];

        dp[i][j] = LINF;
        ll prefix_sm = 0;
        FOR(k, 0, m){
                prefix_sm += mat[i][k];
            dp[i][j] = min(dp[i][j], DP(i - 1, j - k) + prefix_sm);
        }
    return dp[i][j];
}
```

5.3 Another Approach