inzva Algorithm Programme 2018-2019

Bundle 9

DP - 2

Editor
Salih Furkan Ceyhan

Reviewers
Caner Demirer

Contents

1 Bitmask DP

1.1 What is Bitmask

The "mask" in bitmask means hiding something. Bitmask is a binary number that represents something. We use a bitmask to represent a subset of a set with bits.
For example, let A = {1, 2, 3, 4, 5}, we can represent B = {1, 2,4} with 11010 bitmask.

1.2 Bitmask operations

- `Set the` i_{th} `bit:`
  mask = mask | (1 << i)
- `Unset the` i_{th} `bit:`
  mask = mask & $\sim$ (1 << i)
- `Check if` i_{th} `bit is set:`
  mask & (1 << i)
- `Toggle` i_{th} `bit:`
  mask = mask ^ (1 << i)
- `Count the number of 1's:`
  `__builtin_popcount(mask)`

1.3 Bitmask DP Problem Example

There are N people and N tasks and each task is going to be allocated to a single person. We are also given a matrix *cost* of size $N \times N$, where $cost[i][j]$ denotes how much a person is going to charge for a task. Now we need to assign each task to a person in such a way that the total cost is minimum. Note that each task is to be allocated to a single person, and each person will be allocated only one task.

Naive Approach: Try $N!$ possible assignment. Time complexity: $O(N!)$

DP Approach: For every possible subset find new subsets can be generated from this subset and update DP array. Time complexity: $O(2^N * N)$

Solution Code:

```
for (int mask = 0; mask < (1<<n); ++mask)
{
    for (int j = 0; j < n; ++j)
    {
        //check(mask , j) returns jth bit of the mask
        if(check(mask,j) == 0) // jth task not assigned
        {
            //count(mask) returns number of 1's in binary representation of mask
            dp[mask | (1 << j)] =
                min( dp[mask | (1 << j)] , dp[mask] + cost[count(mask)][j]);
        }
    }
}
//after this operation our answer stored in dp[ (1<<N) - 1 ]
```

2 DP on Rooted Trees

We define functions for nodes of the trees which we calculate recursively based on children of a node. One of the states in our DP is usually a $node_i$, denoting that we are solving it for the sub-tree of $node_i$.

2.1 Problem Example

Given a tree T of N (1-indexed) nodes, where each $node_i$ has C_i coins attached to it. You have to choose a subset of nodes such that no two adjacent nodes (nodes connected directly by an edge) are chosen and the sum of coins attached to nodes in the chosen subset is maximized.

We define $dp1(V)$ and $dp2(V)$ as the optimal solution for when we are choosing nodes from sub-tree of node V and if we include node V in our answer or not, respectively. Our final answer is the maximum of two cases, $max(dp1(V), dp2(V))$.

We can see that $dp1(V) = C_V + \sum_{i=1}^{n} dp2(v_i)$, where n is the number of children of node V and v_i is the ith child of the V. Similarly, $dp2(V) = \sum_{i=1}^{n} max(dp1(v_i), dp2(V))$.

Complexity: $O(N)$.
Solution Code:

```
//pV is parent of V
void dfs(int V, int pV){
    //base case:
    //when dfs reaches a leaf it finds dp1 and dp2 and does not branch again.

    //for storing sums of dp1 and max(dp1, dp2) for all children of V
    int sum1=0, sum2=0;

    //traverse over all children
    for(auto v: adj[V]){
        if(v == pV) continue;
        dfs(v, V);
        sum1 += dp2[v];
        sum2 += max(dp1[v], dp2[v]);
    }

    dp1[V] = C[V] + sum1;
    dp2[V] = sum2;
}
//Nodes are 1-indexed, therefore our answer stored in dp1[1] and dp2[1]
//for the answer we take max(dp1[1],dp2[1]) after calling dfs(1,0).
```

3 DP on DAGs

As we know, nodes of a DAG can be sorted topologically, and DP can be implemented efficiently through this sorting.

First, we can find topological order with topological sort in $O(N)$ complexity. Then we can find $dp(V)$ values in topological order, where V is a node in the DAG and $dp(V)$ is the answer for node V. Answer and implementation will differ from problem to problem.

3.1 Conversion of a Problem to a DAG

Most of DP problems can be converted into a DAG. We are going to see why that is the case.

It is obvious that while doing DP and processing a state we evaluate a state by looking possible previous states; and to be able to this, all of the possible previous states must be processed before the current state. From this point of view, we can see that some states depend on other states to be able to get processed, which gives us a DAG formation.

Note that some DP problems cannot be reduced to a DAG and may require hyper-graphs.

Let's see a conversion on an example.

Problem:

There are N stones numbered 1,2,...,N. For each i ($1 \leq\ i\ \leq N$), the height of i_{th} stone is h_i. There is a frog who is initially on the stone 1. He will repeat the following action some number of times to reach stone N:
If the frog is currently on $stone_i$, it can jump to $stone_{i+1}$ or $stone_{i+2}$. Here, the cost of a jump from i to j is $\mid h_i - h_j \mid$. Find the minimum possible cost to reach $stone_N$.

Solution:

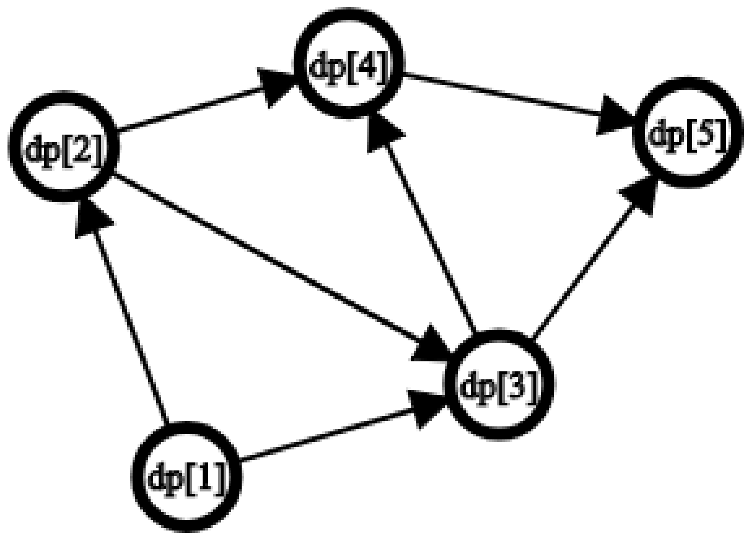
DP approach: We define $dp[i]$ as the minimum cost of the first i index. In the end, our answer will be $dp[N]$. We can denote $dp[i]$ in terms of $dp[i-1]$ and $dp[i-2]$:

$dp[i] = min(dp[i-1] + abs(h_i - h_{i-1}),\ dp[i-2] + abs(h_i - h_{i-2}))$

For N=5, we can see that in order to calculate $dp[5]$ we need to calculate $dp[4]$ and $dp[3]$ first and this rule applies to $dp[4]$ and $dp[3]$ as well.

Similarly,
$dp[4]$ needs $dp[3]$ and $dp[2]$ to be calculated,
$dp[3]$ needs $dp[2]$ and $dp[1]$ to be calculated,
$dp[2]$ needs $dp[1]$ to be calculated,

From these dependencies we can construct a DAG:



3.2 DP on DAG Problem Example

For a given DAG with N nodes and M weighted edges, find the longest path in the DAG.

Complexity: $O(N + M)$
Solution Code:

```
// topological sort is not written here so we will take tp as it is already sorted
// note that tp is reverse topologically sorted
// vector <int> tp
// n , m and vector <pair<int,int>> adj is given.Pair denotes {node,weight}.
// flag[] denotes whether a node is processed or not.Initially all zero.
// dp[] is DP array.Initially all zero.

for (int i = 0; i < (int)tp.size(); ++i)//processing in order
{
  int curNode = tp[i];

  for (auto v : adj[curNode])//iterate through all neighbours
      if(flag[v.first])//if a neighbour is already processed
          dp[curNode] = max(dp[curNode] , dp[v.first] + v.second);

  flag[curNode] = 1;
}
//answer is max(dp[1..n])
```

4 Digit DP

Problems that require the calculation of how many numbers there are between two values (say, A and B) satisfying a particular property, can be solved using digit dynamic programming.

4.1 How to Work on Digits

While constructing our numbers recursively, we need a way to check if our number is still smaller than the given boundary number. For that, while branching, we keep a variable named “strict” or “tight” which limits our ability to select numbers that are bigger than the boundary number.

Let’s suppose that the boundary number is A. We start filling the number from the left (most significant digit) and set strict to $true$, meaning that we cannot take any number that is higher than A’s corresponding digit. While branching, the values less than that digit are now going to be not strict (strict $= false$) because after that point we guarantee that the number is going to be smaller than A. For the value equal to A’s corresponding digit, strictness will continue to be $true$.

4.2 Problem Example

How many numbers x are there in the range A to B, where the digit d occurs exactly k times in x? Constraints: $A, B < 10^{18}$

Brute force: $O(N \log_{10}(N))$, this complexity is too big, hence we need a better approach.

Recursive approach: $O(N)$, actually $O(3^{\log_{10} N}))$ but can be reduced to N. Still not enough because N is too big.

Recursive with memoization: $O((\log_{10} N)^2)$. We can use $dp[25][2][25]$ array and we calculate every value maximum one time, therefore worst case is $25*2*25$ which equals $(\log_{10} N)*([0..1])*(\log_{10} N))$

Solution Code:

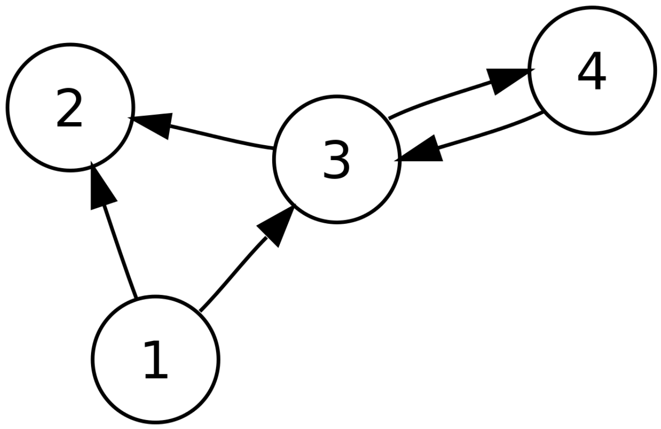
```cpp
#include <bits/stdc++.h>
using namespace std;
#define ll long long
ll A,B,d,k,dg;// dg: digit count
vector <ll> v;// digit vector
ll dp[25][2][25];
void setup(ll a)
{
    memset(dp,0,sizeof dp);
    v.clear();
    ll tmp = a;
    while(tmp)
    {
        v.push_back(tmp%10);
        tmp/=10;
    }
    dg = (ll)v.size();
    reverse(v.begin(), v.end());
}
ll rec(int idx, bool strict, int count)
{
    if(dp[idx][strict][count]) return dp[idx][strict][count];
    if(idx == dg or count > k) return (count == k);
    ll sum = 0;
    if(strict)
    {
        // all <v[idx] if d is included -1
        sum += rec(idx+1, 0, count) * (v[idx] - (d<v[idx]));
        // v[idx], if d==v[idx] send count+1
        sum += rec(idx+1, 1, count + (v[idx]==d) );
        if(d < v[idx])
            sum += rec(idx+1, 0, count+1); // d
    }
    else
    {
        sum += rec(idx+1, 0, count) * (9); // other than d (10 - 1)
        sum += rec(idx+1, 0, count+1); // d
    }
    return dp[idx][strict][count] = sum;
}
int main()
{
    cin >> A >> B >> d >> k;
    setup(B);
    ll countB = rec(0,1,0);//countB is answer of [0..B]
    setup(A-1);
    ll countA = rec(0,1,0);//countA is answer of [0..A-1]
    cout << fixed << countB - countA << endl;//difference gives us [A..B]
}
```

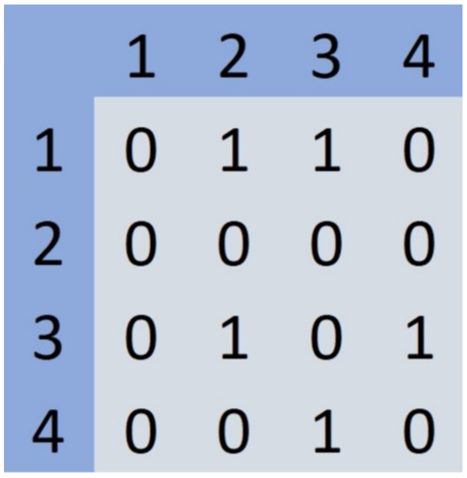
5 Walk Counting using Matrix Exponentiation

This method helps to count *walks* with desired *length*.

Let l be the desired length and let A and B be a node in graph G. If D is the adjacency matrix of G. Then $D^l[A][B]$ is the number of *walks* from A to B with length l.

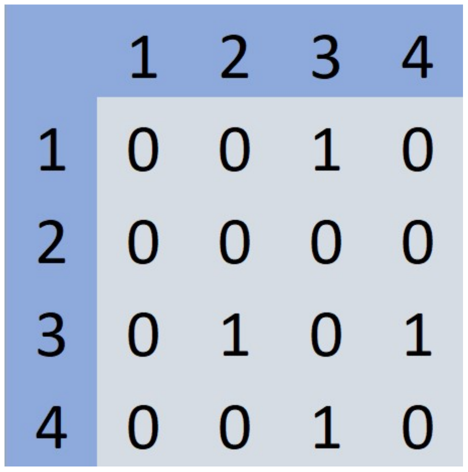
5.1 Example





	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	1
4	0	0	1	0

Figure 1: D, adjacency matrix of G



	1	2	3	4
1	0	0	1	0
2	0	0	0	0
3	0	1	0	1
4	0	0	1	0

Figure 2: D^3

From the D^3 we can see that there is 4 total *walks* with 3 length.
Let S be the set of *walks* and let w be a *walk* where $w = \{n_1, n_2, ..n_k\}$ and $n_i = i_{th}$ node of the *walk*. Then,
$S = \{\{1, 3, 4, 3\}, \{3, 4, 3, 2\}, \{3, 4, 3, 4\}, \{4, 3.4.3\}\}$ and $|S| = 4$.

Using fast exponentiation on adjacency matrix we can find number of *walks* with length k in $O(N^3 \log k)$ time, where N is the number of nodes in the graph.

6 Tree Child-Sibling Notation

In this method, we change the structure of the tree. In the standard tree, each parent node is connected to all of its children. Here, instead of having each node store pointers to all of its children, a node will store a pointer to just one of its children. Apart from this, the node will also store a pointer to its immediate right sibling.

In this notation we can see that every node at most has 2 children (left(first-child), right(first-sibling)), therefore this notation represents a binary tree.

This notation is also called LCRS (left child-right sibling).

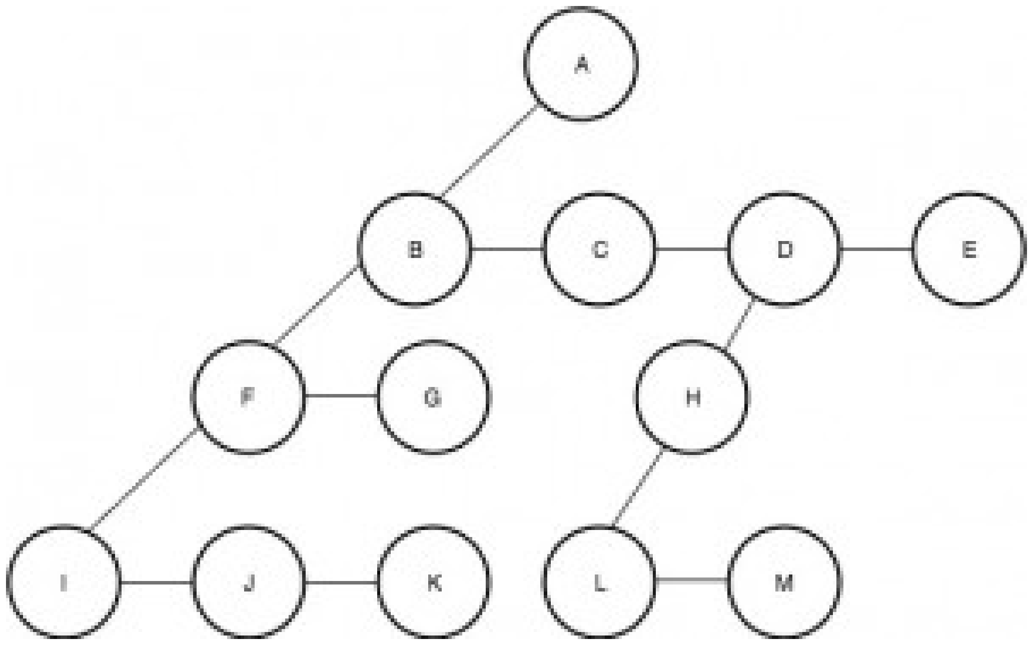


Figure 3: a tree notated with child-sibling notation

6.1 Why You Would Use LCRS

The main reason for using LCRS is to save memory. In LCRS notation we do use less memory than the standard notation.

When you might use LCRS:

- `Memory is extremely scarce`
- `Random access to a node's children is not required`

Possible cases:

1. `If you needed to store a staggeringly huge multi-way tree in main memory.`

   For example, phylogenetic tree.

2. `In specialized data structures in which the tree structure is being used in very specific ways.`

   For example, heap data structure. The main operations used on Heap data structure are:

   - `Remove the root of a tree and process each of its children,`
   - `Join two trees together by making one tree a child of the other.`

   These two operations can be done efficiently on an LCRS structure. Therefore, using LCRS is convenient while working on a heap data structure.

References

[1] https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree

[2] https://contribute.geeksforgeeks.org/wp-content/uploads/new.jpeg

[3] https://en.wikipedia.org/wiki/Phylogenetic_tree

[4] https://www.hackerearth.com/practice/algorithms/dynamic-programming/bit-masking/tutorial/

[5] http://www.aclweb.org/anthology/C08-5001

[6] https://noi.ph/training/weekly/week5.pdf

[7] https://courses.csail.mit.edu/6.006/fall11/rec/rec19.pdf

[8] https://stackoverflow.com/questions/14015525/what-is-the-left-child-right-sibling-representation-of-a-tree-why-would-you-us

[9] Bitmasks Codeforces

[10] DP on Tree Codeforces

[11] Digit DP Codeforces

[12] Digit DP HackerRank

[13] Graph Drawing/Editing

[14] Walk Counting