DP, Greedy and Number Theory

Programming Club

(2016-17)

Science and Technology Council

IIT Kanpur

Introduction

In this lecture, we will be getting you acquainted with some methods of solving specific problems. Though there may be multiple methods of solving certain problems it helps to know which class of problems does the problem at hand belong to. Once we identify this, we can get onto refining our approach,etc.

In this lecture we will start with Elementary Number Theory ,the knowledge of which is an essential tool to solving problems apart from competitive programming also.

Next we move onto 2 popular Problem Solving Paradigms: Greedy Strategies and Dynamic Programming.

Topics to be covered:

1. Number Theory
    a. Congruences
    b. Fermat’s Little Theorem
    c. Chinese Remainder Theorem
    d. Euler’s Totient Function
2. Greedy Strategies
    a. Representative Examples
3. Dynamic Programming
    a. Representative Examples
4. Identifying Greedy / DP problems

Congruences

Definition: Let m be a positive integer. We say that a is congruent to b modulo m if $m \mid (a - b)$ where a and b are integers, i.e. if $a = b + km$ where $k \in Z$. If a is congruent to b modulo m, we write $a \equiv b \pmod{m}$.

Eg. $19 \equiv 5 \pmod{7}$. Similarly $2k + 1 \equiv 1 \pmod{2}$ which means every odd number is congruent to 1 modulo 2.

Properties of Congruences

Let a, b, c and d denote integers. Let m be a positive integers. Then:

1. If $a \equiv b \pmod{m}$, then $b \equiv a \pmod{m}$.
2. If $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$.
3. If $a \equiv b \pmod{m}$, then $a + c \equiv b + c \pmod{m}$.
4. If $a \equiv b \pmod{m}$, then $a - c \equiv b - c \pmod{m}$.
5. If $a \equiv b \pmod{m}$, then $ac \equiv bc \pmod{m}$.
6. If $a \equiv b \pmod{m}$, then $ac \equiv bc \pmod{mc}$, for $c > 0$.
7. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $a + c \equiv (b + d) \pmod{m}$.
8. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $a - c \equiv (b - d) \pmod{m}$.
9. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $ac \equiv bd \pmod{m}$.

All of these can be proved using the original definition of congruences. Try it out!

Euler’s Totient Function

Definition: The Euler φ-function of a positive integer n, denoted by φ(n) counts the number of positive integers less than n that are relatively prime to n.

Example . Since 1 and 3 are the only two integers that are relatively prime to 4 and less than 4, then φ(4) = 2. Also, 1,2,...,6 are the integers that are relatively prime to 7 that are less than 7, thus φ(7) = 6.

In the following slides we will see some properties of the Totient Function.

Properties of Totient Function

1. If p is prime, then $\varphi(p) = p - 1$. Conversely, if p is an integer such that $\varphi(p) = p - 1$, then p is prime.

2.Let p be a prime and m a positive integer, then $\varphi(p^m) = p^m - p^{m-1}$

3.Let m and n be two relatively prime positive integers. Then $\varphi(mn) = \varphi(m)\varphi(n)$.

4.Let $n=p_1^{a1}p_2^{a2}....p_k^{ak}$ .Then $\varphi(n)=n(1-p_1^{-1})(1-p_2^{-1})....(1-p_k^{-1})$.

Fermat’s and Euler’s Theorems

Fermat’s Theorem:If p is a prime and a is a positive integer with (p,a)=1 , then
$a^{p-1} \equiv 1 (\text{mod } p)$.

Fermat’s Little Theorem is a special case of Euler’s Theorem which states that ,
If m is a positive integer and a is an integer such that (a, m) = 1, then
$a^{\varphi(m)} \equiv 1 (\text{mod } m)$ where $\varphi(m)$ is Euler’s Totient Function.

Fermat’s theorem, among other uses helps us in finding modular inverses of numbers which has wide application in many kinds of problems.

Modular Inverse

For a given modulo m , the modular inverse of a with respect to m is the integer b such that $ab \equiv 1 \bmod m$.

Fermat’s little theorem states that $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$. Modifying this , we get $a.a^{p-2} \equiv 1 (\mathrm{mod}\ p)$. Thus a^{p-2} is the modular inverse of a with respect to a prime p.

Luckily for us we can find a^{p-2} in $O(\log p)$ time using binary exponentiation(also known as repeated squaring).

Finding the modular inverse efficiently has many applications.One is to calculate inverse factorials of a number with respect to a given prime,which helps in combinatorics (calculating ${}^{n}C_{r}$ for a large n and r).

Chinese Remainder Theorem

Theorem 27. The system of congruences

$x \equiv b_1 \pmod{n_1}$,

$x \equiv b_2 \pmod{n_2}$,

.

.

.

$x \equiv b_t \pmod{n_t}$,

has a unique solution modulo $N = n_1 n_2 \ldots n_t$ if $n_1, n_2, \ldots, n_t$ are pairwise relatively prime positive integers.

The solution is given by: $x=b_1N_1y_1 + b_2N_2y_2 \ldots + b_tN_ty_t$ Where $N_i=N/n_i$ and y_i is the inverse of N_i with respect to n_i .

Greedy Strategies

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. In some cases, greedy works—the solution is short and runs efficiently. For many others, however, it does not. A problem must exhibit these two properties in order for a greedy algorithm to work:

1. It has optimal sub-structures. Optimal solution to the problem contains optimal solutions to the sub-problems.

2. It has the greedy property .If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We will never have to reconsider our previous choices.

The main tough part in applying greedy strategies is rarely the implementation but proving whether the greedy strategy applied to solve the problem is correct or not.

Some Representative Examples

Question:Given a set of N presents ,with the ith present having a weight of a_i kgs , and you have a bag with capacity of W kgs , what is the maximum number of presents that you can carry with yourself?

Solution. Here we will use a simple and quite intuitive greedy approach. Here is the approach:

Let us arrange the presents in the increasing order of their weights.Now while we have not exceeded W kgs, let us start taking presents in our bag from the smallest to the largest. The number of presents that we can take in our bag in such a fashion is the maximum possible.

How do we prove that this is the maximum possible? Let us say that in our solution we have taken weights a_1 ,a_2...,a_k and $a_1+a_2+...+a_k+a_{k+1}>W$. Let us assume that the optimal solution chooses $b_1,b_2,b_3.....b_j$ and $b_1+b_2+...+b_j<=W$ such that j>k. Let us choose the smallest k elements in the optimal solution. It is seen that $b_1+b_2+....+b_k>=a_1+a_2+...a_k$. Also bk+1>=ak+1.But since $b_1+b_2+...b_k+b_{k+1}<=W$ and $a_1+a_2+...+a_k+a_{k+1}>W$, we have a contradiction.Hence our solution finds the maximal possible solution.

Another Representative Example

Question: Interval Scheduling:Given N intervals of the form [s_i,e_i], find a maximal set of intervals so that for any two intervals in the set are disjoint. Eg: The intervals are [1,2] [2,4] [4,7] [5,8] [4,8] [7,9] and [3,6]. The maximal cardinality of the set is 3. One possible set is {[1,2],[3,6],[7,9]}.Note that this example has only one possible solution set but in the general set there may be more than one solution set.

Solution: Sort the intervals in ascending order of endpoints and then by ascending order of starting points.The order in the example will be [1,2] [2,4] [3,6] [4,7] [4,8] [5,8] [7,9]. Now choose the earliest not-chosen interval. Remove all those intervals that have a non-zero intersection with that particular interval. Repeat this process till there are no more intervals left to choose. The set chosen is a maximal set.

Proof:Consider the set containing the smallest not-chosen interval and all those intervals that overlap with this interval.All the intervals in this set overlap with each other and thus we can choose only one interval from this set.So it makes sense to choose the one with the smallest end-point as choosing one with a larger end point may lead to more overlaps and a smaller solution set.

Conclusion

These were just 2 examples of quite simple greedy algorithms which may be used in real life too.There are various other “standard” greedy algorithms , in other areas like graph theory,etc.

However, greedy algorithms vary from problem to problem and thus it is necessary to come with a particular greedy strategy and prove it.

Not proving a greedy strategy can lead to a number of wrong submissions.Thus it helps to know different methods of proving greedy strategies like Contradiction,Exchange Argument,Greedy stays ahead, etc. Links to examples of such proofs are provided.

Dynamic Programming

What is Dynamic Programming?

We all know recursion (courtesy ESC101). So what does it have to do with Dynamic Programming?Dynamic Programming(DP) is a kind of *intelligent* recursion. In DP , we remember the states we have visited in the recursion and store them in a table.So when we reach that state again , we simply look at the value from the table.

Dynamic Programming is used for counting and optimization problems.

Counting problems are those in which we have find the number of arrangements/subsets,etc. That satisfy a given property.

Optimization problems are those in which we have to minimize or maximize a given value/property subject to some constraints. We’ll be illustrating both of these.

Fibonacci:First look at DP

All of us know about Fibonacci numbers.They satisfy the relation F(n)=F(n-1)+F(n-2).
How would we code Fibonacci recursively?

```
int f(int n){
if(n==0)return 0;
if(n==1)return 1;
return f(n-1)+f(n-2);
}
```

This takes exponential time (It can be proved.Try to prove F(n)>$2^{n/2}$ for large n by induction).
However we can optimise this to O(n).If one notices the recursive calls carefully we can see that we are calculating the same value again and again.For example if we calculate F(5) we call F(4) and F(3).In the calculation of F(4) we call F(3) and F(2). We see that F(3) is called again. Similarly F(2) , F(1) and F(0) are called again and again leading to exponential time.

Continued..

We can see that we can avoid making repetitive recursive calls by storing the values of F(n) in an array. The code will be amended as follows:

```
int Fibonacci[50];
//Set Fibonacci[i]=-1 for all i initially and Fibonacci[0]=0 and Fibonacci[1]=1
int f(int n){
if(Fibonacci[n]!=-1)
return Fibonacci[n];//We have already calculated F(n) and hence don’t need to do so again
Fibonacci[n]=f(n-1)+f(n-2);//Appropriate recursive calls
return Fibonacci[n] ;
}
```

This code works in O(n) time.

This is the core idea of Dynamic Programming.We trade on space to improve the efficiency of the algorithm. Here we made an array of N integers to and amended our recursion so that we come down from exponential complexity to linear complexity.

Top-Down and Bottom-Up DP

The code in the last slide was an example of top-down DP.Why is it called Top-Down?This is because we started the recursion from a higher(larger) state ,calculated the smaller values it was built up from and then combined them.

There is another mode of DP called bottom-up.Here we calculate the lower states first and then move on to the higher states.This way the values that a particular state are dependent on are already calculated before we reach that state.Bottom-up DP is usually done iteratively.The code for Bottom-Up fibonacci is:

```
int Fibonacci[50];
Fibonacci[0]=0;Fibonacci[1]=1;
for(int i=2;i<=n;i++)
Fibonacci[i]=Fibonacci[i-1]+Fibonacci[i-2];
```

This is also O(n) in time and space.

Counting DP:An easy example

Question:You are given a staircase having N stairs.If you are on the i^{th} stair you can go to either the $i+1^{th}$ stair ot the $i+2^{th}$ stair.You are now on the floor(or stair number 0).Print the number of ways to reach the N^{th} stair.

Solution:This is a counting problem solvable by Dynamic Programming.

Let DP(i) denote the number of ways to reach the ith stair from the floor.We can either jump onto the ith stair from the i-1th stair or from the i-2th stair . So DP(i) depends on the values of DP at i-1 and i-2.The recursive formula is : $DP(i)=DP(i-1)+DP(i-2)$.

But this precisely the same as Fibonacci numbers (except with a different base case).Here $DP(0)=1$ and $DP(1)=1$. So we can solve this problem in $O(n)$ in either a recursive fashion or an iterative one.

Optimization DP:An easy example

Question:Again we have a staircase with N steps.Again we have to reach the top of the staircase.This time we can jump from the ith stair to the i+1th stair or the i+3th stair.There is an additional condition that some stairs are blocked and can't be used.Now we are interested in finding the minimum number of jumps to reach the Nth stair.

Solution:This is an optimization problem.Here we want to minimize the number of jumps to reach the Nth staircase.

Let DP(i) denote the minimum number of jumps we need to reach the ith stair.The recurrence is

If ith stair is not blocked :DP(i)=min(DP(i-1) , DP(i-3)) + 1

If ith stair is blocked:DP(i)=infinity

This is because we can come either from the i-1th step or the i-3rd step and we need 1 jump to reach the ith step and hence the +1.This solution is O(n).

An introduction to 2D DP

Question:You are given a NxN grid. The cell at the intersection of the ith row and the jth column has A[i,j] gold coins. You start at the top left corner of the grid and can only move right or down.You're final position is the bottom right corner of the grid.On your way through the cells you pick up all the gold coins lying in the cell.What is the maximum number of gold coins that you can pick up on your way to the bottom right corner?

Solution:The number of valid paths in this case is ${}^{2N}C_N$ .For large N we cannot enumerate all paths.We must do better.Here DP comes into the picture. Let F[i,j] denote the maximum number of gold coins we can pick up from on our way from [1,1] to [i,j]. The recurrence is easy enough:

$$F[\,i\,,j\,]=\max\,(\,F[\,i-1\,,j\,]\,,F[\,i\,,j-1\,])+A[\,i\,,j\,]$$

The answer is F[N , N].

Follow up Question:What if some cells are blocked in the grid and you cannot pass through them?Then what is the maximum number of coins we can collect if we can reach [N,N].

Further Follow Up: What if we have a limited number of steps that we can take in a particular direction?Say that we can move maximum D steps right or D steps down after which we are forced to change our current direction. (Hint:Think 3D DP -Not easy).

In Conclusion

Dynamic Programming is not the easiest of topics to learn let alone master . This is because of the sheer number of different problems that can be made using DP.

However once one begins to grasp the key idea of DP , i.e, identifying states and then identifying recurrences, they can move onto more variations.

The only thing that will ensure progress is practice.One must practice relentlessly to become good at DP.

When To be Greedy

Often when one sees an optimization problem one thinks of a greedy approach. It is a natural tendency to try to obtain some kind of strategy to solve the optimization problem when there exists none.

What helps to construct a whole lot of test cases before you code. In particular you should try to make test cases with the intention of breaking your greedy strategy.What really helps if you are able to prove the strategy but if you cannot do so this is the second best thing.

When greedy fails,we must turn to DP. One should remember that DP is a kind of intelligent brute force.It is *always correct*. Whereas greedy isn't. If one can apply a DP solution respecting the constraints of the problem then one should go ahead.

However, we can get some kind of idea of the solution from the constraints too.For eg.If N is 10^5 and the only possible DP is $O(n^2)$ then it most probably is a greedy solution.However, figuring out the algorithm from the constraints is a bad idea.

Further reading

1. Must read tutorial on Dynamic Programming:
   https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/
   https://www.hackerearth.com/notes/dynamic-programming-i-1/
2. DP:Classical Problems:Reading the solutions of these is essential.These are easily googlable:
   a. Longest Common Subsequence of two string
   b. Maximum Subarray Sum
   c. Knapsack/Subset Sum
   d. Edit Distance
   e. Longest Increasing Subsequence of an array (O(n^2) solution)
   f. Travelling Salesman Problem(Bit advanced)
3. Number Theory : All the theorems given can be found on Wikipedia with proof.
4. Greedy:
   http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/120%20Guide%20to%20Greedy%20Algorithms.pdf
   Topcoder: https://www.topcoder.com/community/data-science/data-science-tutorials/greedy-is-good/