CSE200A : Competitive Programming I (Summer 2023) Reading 2

FooBar: Competitive Programming Club of IIITD

June 2023

1 Advanced Number Theory

1.1 Inverse modulo

Given two integers A and M, the modular multiplicative inverse of A under modulo M is an integer X such that:

$$AX \cong 1 \pmod{M}$$

To find the inverse modulo, you can use the extended Euclidean algorithm, which is an extension of the Euclidean algorithm for finding the greatest common divisor (GCD) of two numbers. The extended Euclidean algorithm can also find coefficients x and y such that ax + by = GCD(a, b), where a and b are integers.

To find the inverse modulo a given modulus m, you need to find the coefficients x and y such that ax + my = 1. In this case, x is the inverse of a modulo m. If x is positive, it is the inverse modulo m. If x is negative, you can add m to it to make it positive, and that will be the inverse modulo m.

Resource - Inverse Modulo

1.2 Chinese remainder theorem

The Chinese Remainder Theorem (CRT) is a fundamental result in number theory and modular arithmetic. It provides a way to solve a system of congruences and find a unique solution that satisfies each congruence simultaneously.

Suppose we have a system of congruences:

$x \equiv a_1 \pmod{m}_1$
$x \equiv a_2 \pmod{m}_2$
...
$x \equiv a_n \pmod{m}_n$

where $a_1, a_2, ..., a_n$ are integers and $m_1, m_2, ..., m_n$ are pairwise relatively prime (i.e., they have no common factors other than 1). The Chinese Remainder Theorem states that there exists a unique solution for x modulo the product of all the moduli $(m_1 * m_2 * ... * m_n)$.

Resource - Chinese remainder theorem

1.3 Permutations and combinations

Permutations: A permutation is an arrangement of objects in a specific order. The number of permutations of a set of n objects taken r at a time is denoted by $P(n, r)$ or nPr and is calculated using the formula:

$$P(n, r) = \frac{n!}{(n-r)!}$$

where $n!$ (n factorial) is the product of all positive integers from 1 to n. It represents the number of ways to arrange n distinct objects in a specific order.

For example, if you have a set of 5 different books and you want to arrange 3 of them on a shelf in a specific order, the number of permutations would be $P(5, 3) = \frac{5!}{(5-3)!} = \frac{5!}{2!} = 60$.

Combinations: A combination is a selection of objects from a set without regard to the order in which they are arranged. The number of combinations of a set of n objects taken r at a time is denoted by $C(n, r)$ or nCr and is calculated using the formula:

$$C(n, r) = \frac{n!}{r! \cdot (n-r)!}$$

Here, $n!$ represents the factorial of n, $r!$ is the factorial of r, and $(n-r)!$ is the factorial of the difference between n and r.

For example, if you have a set of 5 different books and you want to select 3 of them to take on a trip, without considering their order, the number of combinations would be $C(5, 3) = \frac{5!}{3! \cdot (5-3)!} = \frac{5!}{3! \cdot 2!} = 10$.

Permutations and combinations are widely used in various fields such as probability, statistics, combinatorial optimization, and algorithms. They help in solving problems related to arrangements, selections, probability calculations, and analyzing different possibilities.

Resource - Permutation and combinations

1.4 Practice Problems

Beautiful numbers
Sum of the kth powers)
Festival organisation
Remainders game

2 Greedy

A greedy algorithm chooses a solution based on the "best choice" at a particular instance, with no regard to future choices.

Greedy algorithms may not always yield an optimal solution, so it is important to check the correctness of your solution using proper proving methods. Greedy Algorithms may seem very intuitive but can be hard to prove.

Implementing a greedy solution for a problem during a contest without proof of it's correctness, can result in a lot of time being wasted, in case the solution isn't correct. Therefore, it is always advisable to try proving your greedy approach before you implement it at contest time. However, there can be situations when there isn't much time to prove and you need to depend on your intuition.

When you are unable to arrive at a proper proof for your algorithm, try to challenge it's correctness by coming up with possible failure cases. If you try really hard but do not succeed, there is a better chance of your solution being correct. However, without a formal proof of correctness you can never be sure.

2.1 Example of Successful Greedy Algorithm

Here, we look at a simple example of a problem where a greedy solution works. We also prove its correctness.

Problem: Given an array of n integers, select k integers such that their sum is maximum.

Solution: For k iterations, select the largest element from the array each time.

Proof: We follow a simple proof by contradiction. Suppose there exists an optimal solution that does NOT follow the above solution. This implies, there exists a ¿ b such that b is a selected integer, but a is not. We obtain a selection with a greater sum by replacing b with a. However, this means our original selectino was not optimal.

To prove the correctness of a greedy algorithm, a proof by contradiction is the most common technique. Assume that a different solution exists. Then, there must be some property not satisfied by this solution, allowing us to obtain a better solution, thus proving this solution is not optimal.

2.2 Example of Unsuccessful Greedy Algorithm

Here, we look at an example of a problem where a greedy solution does not work.

Problem: Given an array A of n positive integers, select some elements from A such that their sum is maximum, and no 2 selected elements are adjacent in A.

Proposed Solution: At every iteration, select an element in A having maximum value, that is not adjacent to any other element. Repeat until no more elements can be selected.

Counterexample: We prove the incorrectness of the above algorithm using a counterexample.

Let A = [2, 5, 6, 4, 3, 5]
Greedy Selection: 6, 5, 2. 6 + 5 + 2 = **13**.
Optimal Selection: 5, 4, 5. 5 + 4 + 5 = **14**.

This brings us to a general tip for creating greedy algorithms. When testing correctness of a greedy algorithm, try coming up with **counterexamples** that disprove the correctness of the algorithm. In case you are unable to do so, it may give you an intuition behind the correctness of your algorithm and if you can prove it.

2.3 Resources

- Basics of Greedy Algorithms
- Introduction to Greedy
- Chapter 3 Section 3.3 of cp-book
- Chapter 6 Greedy algorithms, Competitive Programmer's Handbook
- MITOpenCourseWare : Greedy Recitation

2.4 Practice Problems

- Make array increasing (Easy)
- Twins(Easy)
- Random Teams(Easy)
- Movie Festival (Medium)
- Stick Divisions (Medium)
- Tasks and Deadlines (Medium)
- Mortal Kombat Tower(Medium)
- Karen and Coffee (Medium)
- Maximum Width(Medium)
- Producing Snow(Hard)
- Let's Go Hiking(Hard)

3 Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a class of problems that have overlapping sub-problems and optimal substructure property.

The given complex problem is broken into smaller sub-problems such that the optimal solution of the complex problem can be constructed from optimal solutions of those sub-problems. To avoid repetitive computations for sub-problems, the computed results are stored for future use.

Also, these algorithms are mostly used for optimization. Before solving the in-hand sub-problem, algorithm will try to examine the results of the previously solved sub-problems. Then the solutions of sub-problems are combined in order to achieve the best solution.

3.1 Steps for dynamic programming

The following 3 basic steps are followed when creating a dynamic programming algorithm:

1. **Define a class of subproblems**: This is generally the most creative or difficult step. We define subproblems (or "states") that work on subsets of the input values.

2. **Define a recurrence of subproblems**: Define how 'larger' subproblems are dependent on 'smaller' subproblems. Additionally, we define the values of the 'smallest' subproblem(s)(base case), and identify the subproblem(s) that give us the required solution.

3. **Give an algorithm for computing the recurrence:** Define the order in which the subproblems will be solved, in order to obtain the required solution in optimal time.

3.2 Example - 1

We elaborate on the use of the above steps by applying them on the example problem below:

Problem: (Subset Sum Problem) Given an array A of n non-negative integers, and a value S, determine if there is a subset of the given set with sum equal to given S. in O(nS) time.

Solution: Subproblem Definition: Let dp[i, j] = true if there exists a subset of elements in A[0...i] with sum value j. Let dp[i, j] = false otherwise

Recurrence:

$$dp[i,j] = \begin{cases} dp[i-1,j] \quad \text{if } A[i-1] \geq j \\ dp[i-1,j] \vee dp[i-1, j - A[i-1]] \quad \text{otherwise} \end{cases}$$

Base case: $dp[i,0] = \text{true} \quad \forall\, 0 \leq i \leq n$

$dp[0,j] = \text{false} \quad \forall\, 1 \leq j \leq S$

Solution Subproblem: $dp[n, S]$

Brief Algorithm Description: First, calculate all base cases. Then, $\forall 1 \leq i \leq n$, and $\forall 1 \leq j \leq S$, calculate $dp[i, j]$. Return $dp[n, S]$.

Dynamic programming can be applied in several, often unintuitive ways. It can be applied when states are linear using one-dimensional/multidimensional arrays, as well as non-linear data structures such as on trees and graphs. Practicing various dynamic programming problems is the best way to develop the required intuition for many DP problems.

3.3 Example - 2

Problem: There is a list of n numbers and two players who move alternately. On each move, a player removes either the first or last number from the list, and their score increases by that number. Both players try to maximize their scores.

What is the maximum possible score for the first player when both players play optimally? (link to original problem)

Solution: Let $dp[i, j, k]$ denote the maximum score for both players using the subarray from index i to j. k can be either 0 or 1 for first player and second player respectively.

The idea behind this approach is that we can find the best possible score for players for a subarray i to j, if we know the scores for both players $i + 1$ to j and i to $j - 1$. Since we can trivially find the scores for subrrays of length 1, we can then proceed to find scored for arrays of any length.

Recurrence: $dp[i, j, k] = max(dp[i + 1, j, \bar{k}] + a[i], dp[i, j - 1, \bar{k}] + a[j])$

Base case: if $i = j$, $dp[i, j, 0] = a[i]$ and $dp[i, j, 1] = a[j]$.

Solution Subproblem: $dp[1, n, 0]$

You can find an example implementation here

3.4 Example - 3

Problem: Given an array of intervals, with each interval denoted as start and end value, find the minimum intervals we need to remove such that remaining intervals have the property that:

- Each interval belongs to exactly 1 pair of intervals.
- Every pair of intervals is intersecting.
- There are no 2 intervals which intersect and belong to different pairs.

You can read the complete problem description here. Also note that there is a possible greedy approach to this problem, but we will be covering the DP approach here.

Solution: First we will sort the array. Let $dp[i, s]$ represent the a DP state, where i represents the current index, and s represents if we select or leave i. We store the maximum elements we can keep in the array $a[i..n]$ (or $a[0..i]$, depending on personal preference) in $dp[i, s]$.

Once again, the idea here is to simply choose every pair, and remove all intervals which intersect with either of those pairs.

Recurrence: For any index i, we will find all such indexes j such that $a[i]$ and $a[j]$ intersect ($i < j$). Then we find the first index k, such the $a[k]$ does not intersect with either $a[i]$ or $a[j]$. Then, $dp[i, 1] = max(max(dp[k, 0], dp[k, 1]) + 2\forall k)$

On the other hand, $dp[i, 0] = max(dp[i + 1, 0], dp[i + 1, 1])$

Base case: When $i = n$, we always get 0 as we cannot find any $j > i = n$

Solution Subproblem: $max(dp[1, 0], dp[1, 1])$

3.5 Resources

- A beginner friendly introduction to DP
- DP playlist for noob
- Chapter 3 Section 3.4 of cp-book
- Fibonacci and Shortest Paths
- MITOPenCourseWare : DP Recitation
- Knapsack, Edit Distance, Parenthesization

3.6 Practice Problems

- Atcoder Problem sets (Recommended)
- Subset Sum (Easy)
- Knapsack1 (Medium)
- Knapsack2 (Medium)
- Caesar's Legions (Medium)
- Chef and Big Soccer (Medium)
- Algebra Score (Medium)
- Zuma (Hard)
- Bottles (Hard)
- Riding in a lift (Hard)
- Super Egg Drop (Hard)

- Digit Sum(Hard)
- Colin Galen's DP Mashup Contest (Beginner friendly DP mashup contest)
- CSES DP Problemset