STM JOURNALS

Recent Trends in
Parallel Computing

https://journals.stmjournals.com/rtpc

ISSN: 2393-8749
Volume 11, Issue 3, 2024
September–December
DOI (Journal): 10.37591/RTPC

Review RTPC

Competitive Programming and Its Importance in the Present World

Himanshu Mishra[1], Usha Shukla[2,*]

Abstract

Competitive programming has become a vital skill set for both computer science students and professionals, fostering problem-solving abilities, algorithmic thinking, and time management in a highly competitive environment. This paper provides a comprehensive overview of competitive programming, beginning with its core principles and progressing through advanced strategies used by participants to solve complex problems efficiently. In addition to explaining fundamental concepts, the paper explores key algorithms and data structures frequently employed in competitive programming, emphasizing their role in optimizing both time and memory usage. Memory efficiency is discussed in detail, highlighting how careful management of resources can be the difference between success and failure in a contest setting. The paper also offers an introduction to competitive programming platforms, discussing features such as rating systems, and rating graphs, and how these metrics are used to track progress and skill development. Finally, the paper underscores the growing importance of competitive programming in today's tech-driven world, where the skills gained from participating in these competitions are increasingly sought after in both academic and professional settings. By the end, readers will have a thorough understanding of the competitive programming landscape, its challenges, and its value in modern computer science education and career development.

Keywords: Competitive programming, algorithm efficiency, memory efficiency, rating of participant, time complexity

INTRODUCTION

By name, it is easy to understand its meaning, that is, programming there is some competition. Competition to finish code first, in a more efficient time, and with less memory usage. For the reader, it may not be easy to understand these terms; as we go through, everything will be understandable. We address well-known problems in competitive programming. Problems are based on such tags, which may seem easy for the human brain to answer, but preparing a flexible code for them may cause a problem [1–6].

The main aim of this is to teach problem-solving skills to a programmer, which may not be as easy as writing code. learning competitive programming is not just the ultimate goal of a programmer; it helps produce better software that can handle large and complex real-life problems [7]. Competitive Programming is important for a programmer not only to learn question-solving skills but also to help him land in dream jobs or dream companies. A candidate with a good rating on programming platforms has a good impression in the eyes of the interviewers. Successful competitive programming can significantly enhance a candidate's profile, making it more attractive to top-tech companies. Competitive Programming promotes a culture of

***Author for Correspondence**
Usha Shukla
E-mail: usha.shukla1@gmail.com

[1]Student, Department of Computer Sciences and Engineering, Indian Institute of Information Technology, Design and Manufacturing, Jabalpur, Madhya Pradesh, India
[2]Assistant Professor, Amity School of Applied Sciences (ASAS), Amity University Lucknow, Uttar Pradesh, India

Received Date: September 13, 2024
Accepted Date: September 30, 2024
Published Date: October 08, 2024

Citation: Himanshu Mishra, Usha Shukla. Competitive Programming and Its Importance in the Present World. Recent Trends in Parallel Computing. 2024; 11(3): 39–50p.

creativity and teamwork. Participants often work in teams, honing their ability to communicate ideas and collaborate effectively [8]. This research paper deals with several types of algorithms that are used in competitive programming, and this research paper also explains various terminologies that are used in competitive programming. Several codes and references have been used for their explanations, which will help them to understand them easily. In addition, this paper discusses a rating graph that helps understand the knowledge level of a candidate. Various sources have been used for this purpose, which will help us to understand better.

UNDERSTANDING COMPETITIVE PROGRAMMING

Basics of Competitive Programming

Competitive Programming comprises a set of problems comprising certain algorithms *and* mathematical problems within specified time limits. There are various platforms, such as CodeForces, Codechef, Leetcode, and Atcoder. These platforms conduct **contests** three or more times per month. The term **contests** can be considered an exam in which a participant has to solve given problems within given constraints, which generates the global rank of a participant. This also helps generate **a rating** of a participant, which can be considered as the result of his performance during a contest, as shown in Figure 1 [7].

What is the Rating of the Participant?

In competitive programming, a rating is a numerical value assigned to participants based on their performance in contests. Ratings serve as a measure of a participant's skill level and help to rank competitors relative to one another.

How is Rating Calculated?

Ratings are typically calculated using systems similar to the Elo rating system used for chess. Some factors on which the rating depends are:

- Performance relative to other participants.
- Expected performance.
- Contest difficulty.

If we talk about the code forces platform, participants are categorized into various divisions based on their ratings, such as Pupil, Specialist, Expert, Candidate Master, Master International Master, Grandmaster, and International Grandmaster, as shown in Figure 2 [9]. Understanding the terminology of ratings is quite confusing (Figure 3).

How is Time Complexity Calculated?

In competitive programming, the time complexity is measured to predict the runtime scales of an algorithm with an increase in input size. The less time the code requires for execution, the better it is considered. During a contest, there is also a time limit for the code, which makes the problem even more challenging [10].

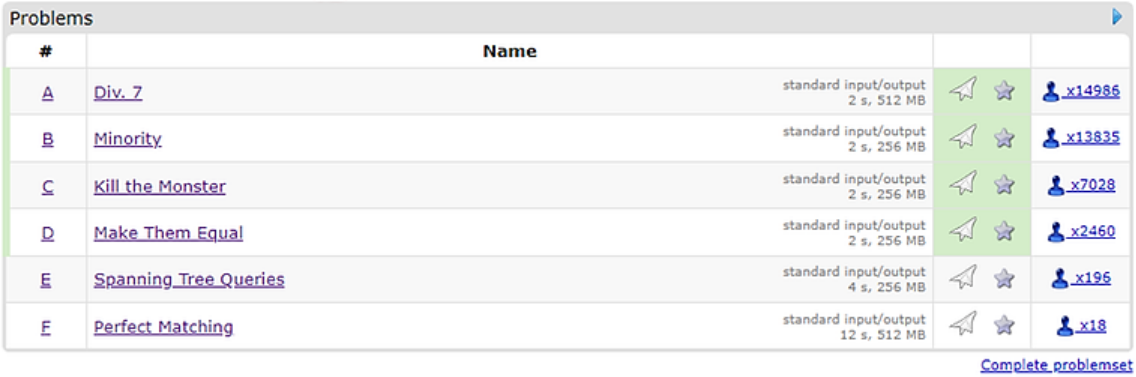


Figure 1. Problem page image of a contest on the CodeForces platform.

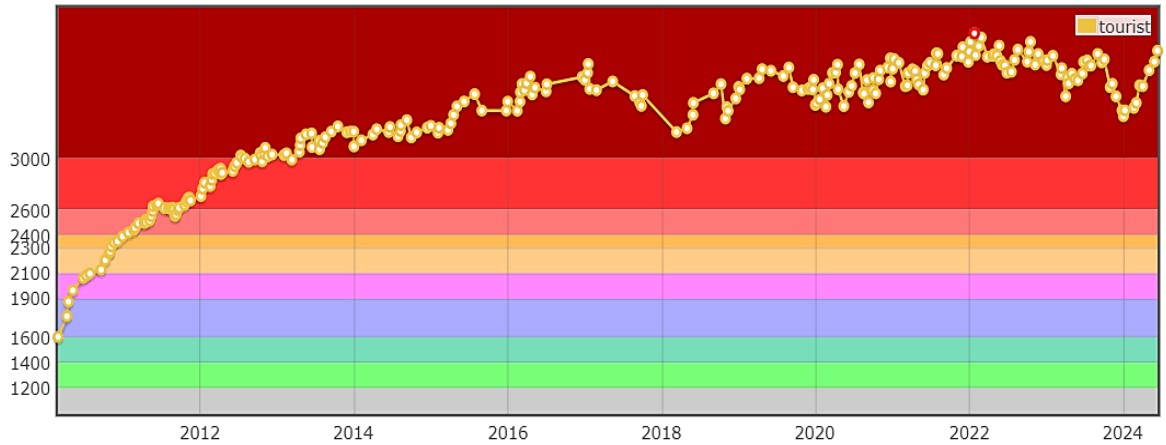


Figure 2. Rating graph of CodeForces top-rated participant.

Elo-MMR	Title	Division	Number	Percentile	CF at same rank (spread)
3000+	Legendary Grandmaster	1	8	99.99	3382+
2700-2999	International Grandmaster	1	37	99.95	3010-3329 (372)
2400-2699	Grandmaster	1	255	99.7	2565-3010 (445)
2200-2399	International Master	1	560	99.1	2317-2565 (248)
2000-2199	Master	1	2089	97	2088-2317 (229)
1800-1999	Candidate Master	2	3968	93	1804-2088 (284)
1600-1799	Expert	2	7103	86	1564-1804 (240)
1400-1599	Specialist	3	11003	75	1328-1564 (236)
1200-1399	Apprentice	3	16909	58	1104-1328 (224)
1000-1199	Pupil	4	23977	34	818-1104 (286)
Up to 999	Newbie	4	33923	0	Up to 818

Figure 3. Table explaining rating terminologies in CodeForces [1].

The time complexity of a code can be estimated easily by observing the operations that a code is following. The following steps help calculate the time complexity of a code:

- *Identifying basic operations:* Basic operations such as arithmetic operations and comparisons are those whose execution times are constant. In general, time complexity is denoted by O (1).
- *Number of times basic operations are executed:* Because basic operations have constant time complexity, if they are performed in some loops, the execution is performed n times for a given value of loop (n). Thus, the time complexity in such cases generally becomes O(n).
- *Loops:* Running loops in code is on the one hand very essential to perform operations but on the other, it affects the time complexity of code directly. Running a loop of size n generally results in a time complexity O(n). However, running a nested loop, which is a loop inside the loop, causes the time complexity to be $O(n^2)$.
- *Combine time complexities:* If an algorithm has multiple steps, the overall time complexity is often the one with the highest growth rate. For example, O(n) +O(n^2) is simplified to $O(n^2)$.
- *Recursive analysis:* Recursion is considered one of the most time-consuming algorithms with a time complexity of $O(2^n)$.

Here are some examples to understand time complexity:

Example 1: Single loop

```
for (int i = 0; i< n; i++) {
// constant time operation
}
```

Its time complexity is O(n) because it runs n times.

Example 2: Nested loop

```
for (int i = 0; i< n; i++) {
for (int j = 0; j < n; j++) {
// constant time operation
}
}
```

Time Complexity: ($O(n^2)$) because the inner loop runs (n) times for each (n) iteration of the outer loop.

Example 3: Recursive Function

```
void recursion(int n) {
if (n <= 1) return;
recursion(n - 1);
}
```

This forms the recurrence relation (T(n) = 2T(n-1) + O(1)). The time complexity is (O(2^n)).

Steps to Calculate Time Complexity

- Write down the pseudocode or actual code of the algorithm.
- Identify the loops, recursive calls, and other control structures.
- Calculate the time complexity of each component.
- The results were combined using the rules of the Big-O notation to determine the overall time complexity.

By understanding and following these steps, we can effectively calculate the time complexity of algorithms in competitive programming, helping to write more efficient and optimized codes (Figure 4).

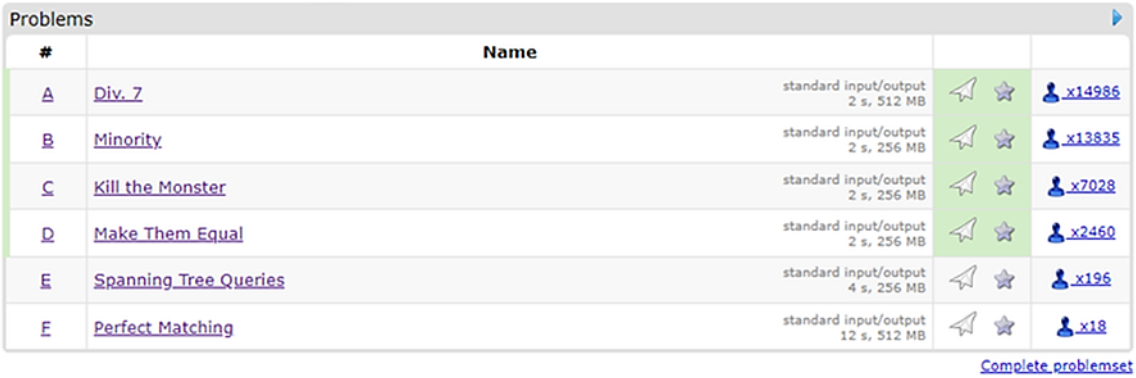
If we look below the standard input/output text, there is time with memory. This indicates the maximum time limit of the code, which is the maximum memory limit.

Understanding Memory Limits

In competitive programming, memory limits specify the maximum amount of memory a program ensures that submissions are efficient and can handle large inputs within the constraints of the problem.

KEY ASPECTS OF MEMORY LIMITS

- *Measurement units:* Memory is typically measured in bytes, but in competitive programming, it is often presented in megabytes (MB) or gigabytes (GB).
- One MB = (10^6) bytes (sometimes approximated as (2^{20}) bytes in certain contexts).
- *Common limits:* Memory limits vary from one competition to another. The common limits were 256 MB, 512 MB, and 1 GB.


Problems

#	Name			
A	Div. 7	standard input/output 2 s, 512 MB		x14986
B	Minority	standard input/output 2 s, 256 MB		x13835
C	Kill the Monster	standard input/output 2 s, 256 MB		x7028
D	Make Them Equal	standard input/output 2 s, 256 MB		x2460
E	Spanning Tree Queries	standard input/output 4 s, 256 MB		x196
F	Perfect Matching	standard input/output 12 s, 512 MB		x18

Complete problemset

Figure 4. Image showing a problem set and successful submissions of some contest.

Memory Usage Calculation

Understanding how much memory different data types and structures use is critical.

Common Sizes

- int: 4 bytes
- long: 8 bytes
- float: 4 bytes
- double: 8 bytes
- char: 1 byte
- *Arrays:* The size depends on the element type and the number of elements.
- *Objects and structures:* The memory used includes the data and references they hold.

Example Considerations

- *Array allocation:* An array of integers with (10^6) elements uses approximately (4×10^6) bytes (4 MB).
- *String handling:* A string with (10^6) characters uses 1 MB.
- *Complex data structures:* A graph with (10^5) nodes and (10^6) edges, represented with adjacency lists, can have significant memory usage owing to the storage of lists and pointers.

Until now, we have discussed various aspects of competitive programming, and we now discuss the most important topic, namely, algorithms.

ALGORITHMS

Greedy Algorithm

A greedy algorithm is a method for solving problems by choosing the best available option at each stage to achieve the optimal solution globally. It focuses on maximizing immediate benefits without considering future outcomes. Although this approach is often efficient and straightforward, it does not ensure an optimal solution for every scenario. Greedy algorithms excel in making local decisions that seem optimal at the moment but may overlook better long-term strategies that could lead to a superior overall outcome.

Examples of Greedy Algorithm

- *Dijkstra's algorithm:* This algorithm identifies the optimal path between two nodes in a graph. It operates by consistently selecting the shortest available edge, starting from the current node.
- *Kruskal's algorithm:* This algorithm determines the minimum set of edges that link all vertices in a graph without creating cycles. This was achieved by continually selecting the edge with the lowest weight that did not cause a cycle.

- *Fractional knapsack problem:* In goal of this problem is to pack a knapsack with items that offer the best value relative to their weight, given a maximum capacity. The greedy approach tackles this by prioritizing items based on their value-to-weight ratio, starting with the highest ratio and filling the knapsack until capacity is reached.
- *Coin change problem:* The greedy algorithm for making a change seeks to minimize the number of coins required to achieve a specific amount. This was achieved by continuously selecting the coin with the highest denomination, which was smaller than the remaining amount to be changed. This approach ensures that larger denominations are used first, thereby minimizing the total number of coins required.

How Does the Greedy Algorithm Work?

- It begins with the problem's initial state, which serves as the starting point for decision-making.
- Assess all potential options from their current state by considering the choices available at that moment.
- Select the option that appears most advantageous at the present moment, without considering future repercussions. This is the "greedy" approach—choosing the best available option now, even if it may not be optimal in the long term.
- Transition to a new state based on the selected option. This new state serves as the starting point for the next iteration.

Example

You are given an array of integers. Find the subsequence with the maximum product.

Suppose you have the array: [-2, 1, -3, 4, -1, 2, 1, -5, 4].

Here, we can use negative numbers to increase the maximum if the count of the negative numbers is greater than one.

If the negative numbers are odd, we can neglect the smallest one, and the remaining negative numbers contribute to the maximum product.

If negative numbers are even at times, then we will take all numbers, thereby increasing the maximum.

Of course, we take the product of all positive numbers (except for the number zero).

Binary Search

Binary search is a systematic technique for locating a specific value in a sorted list. It operates by continually dividing the search range by half until the target value is located or it is determined that the value is not present. This is achieved by comparing the target with the middle element of the current search range, thereby allowing the algorithm to efficiently narrow the possibilities with each iteration. The process continues until the target is located or the search interval is narrowed to zero.

Advantages of Binary Search

Binary search offers superior performance compared with linear search, particularly when dealing with large arrays. It outperforms alternative search algorithms such as interpolation and exponential search, which share comparable time complexities. This method excels in scenarios where datasets are extensively stored in external storage, such as cloud platforms or hard drives. Its efficiency lies in its ability to swiftly pinpoint desired values by systematically dividing the search range and swiftly honing the target value or confirming its absence.

How Does the Binary Search Algorithm Work?

Take a look at an example of deep learning:
Array arr[10] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19], and the target = 17.

How to Implement Algorithm?

Two ways to implement algorithms

- *First one:* Iterative Algorithm
- *Second one:* Recursive Algorithm

Codes are given for both:

Iterative B.S. Algorithm

In this method, a while loop is used to repeatedly compare the target value with the middle element of the current search range. This comparison allowed us to systematically split the search space into two halves (Figure 5).

Recursive B.S. Algorithm

In the recursive version of binary search, a function is defined to compare the middle element of the current search range with the target key. Based on this comparison,

- If the middle element matches the key, the function returns to its index.
- If the middle element is greater than the key, the function recursively searches for the left half of the range.

```cpp
#include <bits/stdc++.h>
using namespace std;

// An iterative binary search function.
int binarySearch(int arr[], int low, int high, int x)
{
    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if x is present at mid
        if (arr[mid] == x)
            return mid;

        // If x greater, ignore left half
        if (arr[mid] < x)
            low = mid + 1;

        // If x is smaller, ignore right half
        else
            high = mid - 1;
    }

    // If we reach here, then element was not present
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Figure 5. The following image shows the code for this approach.

- If the middle element is less than the key, then the function recursively searches for the right half of the range.

This recursive procedure continues until the key is found and its index is returned, or the search space is exhausted (Figure 6) [10, 12].

Dynamic Programming

Dynamic programming (DP) is an approach in mathematics and computer science aimed at tackling intricate problems by breaking them into smaller, manageable sub-problems. It emphasizes solving each sub-problem only once and storing its solution, thereby preventing repetitive calculations. This method optimizes the efficiency of solutions across various challenging problems by leveraging previously computed results to build the final solution effectively [2].

How Does It Work?

- Decompose the primary problem into smaller, independent sub-problems.
- Each sub-problem is solved, and its solution is recorded in an organized manner, such as in a table or array.

```cpp
#include <bits/stdc++.h>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[low..high] is present,
// otherwise -1
int binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low) {
        int mid = low + (high - low) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, high, x);
    }
}

// Driver code
int main()
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int query = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, query);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Figure 6. This image shows the code for this approach in C++.

- The precomputed solutions of these sub-problems are used to incrementally build a solution to the main problem.
- By storing the solutions, dynamic programming ensures that each sub-problem is solved only once, reducing the computational overhead and improving the efficiency of solving the problem.

Example 1: Let's understand DP by the Fibonacci sequence.

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Brute Force Approach: Using a brute force approach to determine the nth Fibonacci number involves calculating it by directly summing the (n-1) th and (n-2) th Fibonacci numbers. Although this method is straightforward, it becomes inefficient for large values of n because it requires computing all preceding Fibonacci numbers up to the nth value. Repetitive calculations can result in high time complexity, making them impractical for large values of n.

Dynamic Programming Approach
Fibonacci series using dynamic programming

Define the Problem
The Fibonacci sequence is defined as:
$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$ for $n > 1$
Our goal is to find the nth Fibonacci number, F(n).

Identify Sub-problems
In the context of dynamic programming, we recognize that calculating each Fibonacci number requires knowing its two predecessors (F(n-1) and F(n-2)). Therefore, we can break down the problem into smaller sub-problems, where each sub-problem computes a Fibonacci number of up to n.

Store Solutions
Instead of repeatedly recalculating Fibonacci numbers, we stored the results of the already computed Fibonacci numbers in an array or table. This allows us to access previously computed values directly when required.

Build Up Solutions
The solution was iteratively built starting from the base cases (F(0) and F(1)) and gradually computing each subsequent Fibonacci number up to F(n). Using the stored results from previous computations, we efficiently computed F(n) at linear time O(n). Hence, we can calculate without repeating and recomputing the sub-problems repeatedly.

When to Use Dynamic Programming?
Dynamic programming is a methodical strategy employed in problem-solving and is characterized by two key aspects.

Optimal Substructure
This principle suggests that the best solution for a larger problem can be built from the best solutions to its sub-problems. By dividing the problem into smaller manageable sections and solving each separately, we can merge these solutions to obtain the optimal solution for the entire problem.

Overlapping Sub-problems
These characteristics highlight the repetition of solving identical sub-problems across different sections of the main problem. Rather than recalculating the results for these sub-problems each time

they appear, dynamic programming aims to store and reuse previously computed solutions. This avoids redundant computations and enhances efficiency by leveraging the stored results.

Dynamic programming ensures efficiency by breaking down complex problems into simpler sub-problems and efficiently managing the reuse of solutions through systematic storage and retrieval.

Approaches of Dynamic Programming

Dynamic programming can be achieved using two approaches:

Top-Down Approach (Memoization)

In the top-down approach, also referred to as memoization, we begin by directly tackling the main problem and then recursively decompose it into smaller and more manageable sub-problems. This method guarantees that each sub-problem is addressed only once by saving its solution in a memoization table. This table acts as a cache, preventing unnecessary precomputation of the results for sub-problems that have already been solved.

Here is a simplified explanation:

1. *Starting point with final solution:* We initiated the solution process directly from the main problem. From there, we recursively break it down into smaller parts until we reach the simplest base cases or sub-problems.
2. *Storing sub-problem solutions:* To enhance efficiency, we saved the solution of each sub-problem in a memoization table as soon as it was solved. This allowed us to quickly retrieve and reuse solutions without having to recompute them, thereby saving time and computational resources.
3. *Ideal conditions for usage:* The top-down approach with memoization is particularly effective when dealing with problems that involve a large number of sub-problems, many of which overlap or are reused in the computation. By storing the solutions, we streamlined the process and avoided redundant calculations, making this approach highly suitable for complex problems with repetitive elements.

Overall, memoization enhances the efficiency of problem-solving using recursive decomposition by strategically caching solutions and minimizing unnecessary computations.

Bottom-Up Approach (Tabulation)

In the bottom-up approach, also referred to as tabulation, we first address the smallest sub-problems and systematically progress towards solving the main problem. This method involves constructing a table in which each entry represents the solution to a sub-problem. By iterative computing and storing these solutions in the table, we ensure that each sub-problem is solved sequentially and logically.

Here's a simplified breakdown:

1. *Starting with small subproblems:* The approach begins by solving the smallest and simplest instances of the problem. From there, it incrementally builds towards solving larger sub-problems until it reaches the final solution to the main problem.
2. *Using a table for solutions:* A key aspect of the bottom-up approach is the utilization of a table (or array) to store computed solutions to sub-problems. Each entry in the table represents a particular sub-problem and is populated using a methodical bottom-up approach.
3. *Ideal conditions for usage:* This approach is particularly useful for problems where the number of sub-problems is feasible, and the optimal solution can be directly obtained from the solutions to smaller sub-problems [12]. It avoids the complexities of recursion and is well-suited for scenarios in which a straightforward sequence of computations can lead to the final solution.

In essence, the bottom-up approach (tabulation) optimizes problem-solving by systematically computing and storing solutions in a structured manner, ensuring efficiency and clarity in addressing complex problems.

Some Other Mathematical Algorithms

Sieve Algorithm

The sieve of Eratosthenes is a traditional algorithm used to determine all prime numbers up to a given number T. It operates efficiently with a time complexity of (O (n × log × log (n))), which makes it suitable for large values of (T). By systematically eliminating non-prime numbers from a list, starting from the smallest prime (2), the algorithm efficiently identifies all primes less than (T). This method involves iteratively marking multiples of each prime number as non-prime, gradually reducing the list of candidates until only the prime numbers remain (Figure 7).

In the Sieve Algorithm, we only calculate all prime numbers that will be required to perform operations with a time complexity of O(n*log(log(n))). Thus, to solve any query, we do not need to calculate prime numbers repeatedly, thus reducing the execution time and risk of time complexity.

Fermat's Little Theorem

In number theory, Fermat's little theorem asserts that for any integer aa and a prime number p, the expression $ap \equiv a(\mathrm{mod}p)$ holds true. This means that raising a to the power of p and then taking the result modulo p yields a itself.

This approach leverages the efficiency of modular exponentiation, where raising a to the power of $m-2$ modulo m yields the modular inverse of an under m. This method is particularly useful in cryptographic algorithms and number-theoretic computations where modular arithmetic plays a crucial role.

```
void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++) {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true) {
            // Update all multiples of p greater than or
            // equal to the square of it numbers which are
            // multiple of p and are less than p^2 are
            // already been marked.
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p = 2; p <= n; p++)
        if (prime[p])
            cout << p << " ";
}
```

Figure 7. This image shows the code for this approach.

CONCLUSION

Competitive programming is a vital discipline that offers significant benefits to those involved. By understanding the basics, including the types of problems, competition structures, key algorithms, and data structures, as well as the platforms and skills developed, aspiring programmers can better appreciate and navigate the competitive programming (CP) world. This foundational knowledge serves as a stepping stone for further exploration and mastery in the field and contributes to personal and professional growth in computer science and software engineering.

REFERENCES

1. Thakur D, Nayyar A. Scilab Textbook Companion for Programming in ANSI C by E. Balagurusamy. 4th ed. New Delhi: Tata McGraw-Hill Education; 2008. Available from: http://scilab.in.
2. GeeksforGeeks (2024). Dynamic Programming or DP. [online] GeeksforGeeks. Available from: https://www.geeksforgeeks.org/dynamic-programming/
3. Majd A, Vahidi-Asl M, Khalilian A, Baraani-Dastjerdi A, Zamani B. Code 4Bench: A multidimensional benchmark of Codeforces data for different program analysis techniques. Comput Lang. 2019;53:38–52.
4. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. MIT Press: Cambridge, USA; 2022.
5. Skiena SS, Revilla MA. Programming challenges: The programming contest training manual. ACM SIGACT News. 2003;34:68–74. DOI: 10.1145/945526.945539.
6. Verhoeff T. The role of competitions in education. Future World: Educating for the 21st Century. pp. 1–10; 1997.
7. Laaksonen A. Guide to Competitive Programming. Springer: Cham, Germany; 2020.
8. Di Mascio T, Laura L, Temperini M. A framework for personalized competitive programming training. 2018 17th International Conference on Information Technology Based Higher Education and Training (ITHET), Olhao, Portugal. 2018. pp. 1–8. DOI: 10.1109/ITHET.2018.8424620.
9. Halim S. Competitive Programming 4: The New Lower Bound of Programming Contests in the 2020s. Olympiads in Informatics. 2020;177–80. DOI: 10.15388/ioi.2020.14.
10. Blum JJ. Competitive programming participation rates: An examination of trends in U.S. ICPC regional contests. Discov Educ. 2023;2:11. DOI: 10.1007/s44217-023-00034-1. PubMed: 36968518.
11. Wikipedia contributors. Codeforces. Wikimedia Foundation. Available from: https://en.wikipedia.org/wiki/Codeforces.
12. Vadiyala VR, Baddam PR. Exploring the symbiosis: Dynamic programming and its relationship with data structures. Asian J Appl Sci Eng. 2018;7:101–12. DOI: 10.18034/ajase.v7i1.81.

