Algorithm Analysis Project

Problem Description:

We are given a set of n machines, where each machine i takes k[i] seconds to produce one product. All machines operate simultaneously and independently, which means multiple products can be produced in parallel at the same time.

The main objective of this project is to determine the minimum time required to produce at least t products using these machines. Since the value of t can be very large, a naive approach that simulates production one second at a time may become impractically slow. Therefore, an efficient algorithm is needed to compute the result within a reasonable amount of time.

Additionally, the project aims to compare the efficiency of naive and optimized algorithms, highlighting the difference in performance between a straightforward simulation approach and a more advanced binary search approach. This comparison helps in understanding the importance of algorithm design and optimization when dealing with large-scale problems.

In summary, the project focuses on:

1. **Calculating the minimum time to produce a target number of products using multiple machines.**
2. **Implementing both a naive (brute force) solution and an optimized (binary search) solution.**
3. **Analyzing and comparing their performance both theoretically and empirically.**

❖ Input

- **An integer `n` — the number of machines.**
- **An integer `t` — the target number of products to produce.**
- **An array `k[]` of length `n` — where `k[i]` represents the time in seconds for machine `i` to produce one product.**

❖ **Output**

- **A single integer representing the** minimum time **required to produce at least** `t` **products.**

❖ **Examples (Input/Output):**

n	t	k[]	Output
3	10	[2,3,7]	12
1	5	[3]	15
4	20	[1,2,3,4]	8

Algorithm Analysis:

1. Naive / Brute Force Approach:

- **Description:**

  This algorithm simulates the production process by increasing the time value one unit at a time. At each time step, it computes how many products all machines can produce using time // k[i].
  Once the total production reaches at least t, the current time is returned. The method is straightforward but becomes extremely slow for large t because time grows linearly

- **Pseudocode:**

```
function bruteForce(k[], t):
    time = 0
    while true:
        time += 1
        produced = 0
        for i in 0..n-1:
            produced += time // k[i]
        if produced >= t:
            return time
```

- **Code snippet:**

```
long long brute_force(const vector<long long>&
k, long long t)
{
    long long time = 0; // O(1)
    int n = (int)k.size(); // O(1)

   while (true)
  {
    // Up to T iterations → O(n*T)
        time += 1; // O(1)
        long long produced = 0; // O(1)

      for (int i = 0; i < n; ++i)
     {
       // O(n) per iteration
       produced += time / k[i]; // O(1)
       if (produced >= t) break;// early stop
     }
        if (produced >= t) return time; // O(1)
  }
}
```

- **Complexity Analysis:**
  - **Time Complexity: O(n*T)**
  - **Space Complexity: O(n)**

---

2. Optimized Approach:

- **Description:**

  This algorithm uses binary search to find the minimum time needed to produce at least t products. Instead of checking every time unit, it tests a middle time value and determines whether the machines can produce enough. If production is sufficient, it narrows the search to smaller times; otherwise, it searches larger ones. This reduces the complexity O(n log T), making it efficient for very large inputs.

- **Pseudocode:**

```
function can(time, k[], t):
    total = 0
    for i in 0..n-1:
        total += time // k[i]
        if total >= t:
            return true
    return false

function binarySearch(k[], t):
    l = 1, r = 1e18
    while l <= r:
        mid = (l + r) // 2
        if can(mid, k, t):
            ans = mid
            r = mid - 1
        else:
            l = mid + 1
    return ans
```

- **Code snippet:**

```
bool can(long long time, const vector<int>& v,
long long t)
 {
    long long total = 0; // O(1)
    for (int i = 0; i < v.size(); ++i) { //O(n)
        total += time / v[i]; // O(1)
        if (total >= t) return true; // O(1)
    }
    return total >= t; // O(1)
}

long long binary_search_sol
(const vector<int>& v, long long t)
{
    long long l = 1, r = 1e18, mid, ans = -1;
    while (l <= r) // O(log T)
  {
        mid = l + (r - l) / 2; // O(1)
        if (can(mid, v, t)) { // O(n)
            ans = mid; // O(1)
            r = mid - 1; // O(1)
        }
        else {
            l = mid + 1; // O(1)
             }
   }
    return ans;
}
```

- **Complexity Analysis:**
    - **Time Complexity: O(n*logT)**
    - **Space Complexity: O(n)**

Empirical Analysis:

Input (n, t,k[])	Output	Brute Force Time (µs)	Binary Search Time (µs)	Notes
(3, 10,000, [2,3,7])	10,245	581	3	Brute force slower than binary search
(3, 1,000,000, [2,3,7])	1,024,392	46,257	2	Brute force time increases significantly
(3, 2,654,000, [2,3,7])	2,718,732	115,745	3	Brute force grows linearly; binary search unaffected

- Observations:
  - Brute force: performance grows linearly with t.
  - Binary search: performance almost constant, very efficient.
  - Binary search performance remains almost constant due to logarithmic search space reduction.

Results Comparison:

- **Theoretical vs Empirical:**
  - **Matches expectations:**
    - **Brute Force: O(n*T), slow for large t.**
    - **Binary Search: O(n*logT), fast even for very large t.**
  - **No significant discrepancies.**
  - **Performance difference becomes huge as t grows.**

Conclusion

- **Binary search is the optimal solution for large t.**
- **Brute force useful only for educational purposes or very small datasets.**
- **Team learned the difference between naive and optimized algorithms, and confirmed theoretical analysis with empirical data.**

Diagram of performance :

To visually illustrate the performance difference between the two algorithms, a comparison diagram based on the empirical results is shown below.

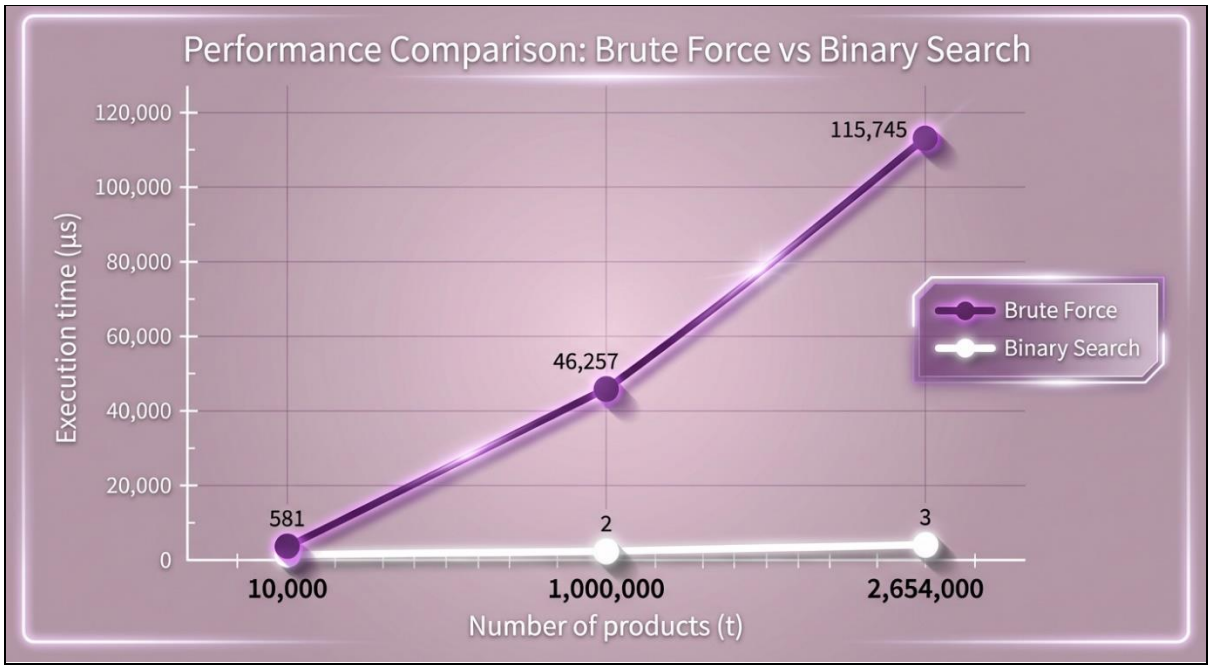

Figure 1: Performance comparison between brute force and binary search algorithms.

Source Code:

The full implementation of both algorithms is available on GitHub:

[GitHub Repository Link]