Dynamic Programming and Fast Fourier Transform Approach for Polynomial Equation Models in Hash Collision Calculation

Fadel Pramaputra Maulana, Rully Soelaiman, *Member, IAENG,* and Yudhi Purwananto

Abstract— **The hash polynomial of a string is defined by converting each letter in the string into a numerical value according to its position in the alphabet. In some cases, the number of strings that satisfy a given hash value is determined using a specified hash polynomial function. The number of possible character combinations and the repetition of subproblems that may occur present challenges in the given hash function. In this paper, we propose a novel solution to the aforementioned problem using dynamic programming technique and Fast Fourier Transform method, which satisfy a given polynomial hash function. In our algorithm, the recurrence relation for the dynamic programming technique utilizes polynomial equation model for each state. Therefore, for each transition in the recurrence relation, the values are updated through polynomial operations. One of these operations is polynomial multiplication, where the Fast Fourier Transform (FFT) is used to speed up computation. Based on test results from a given case study, the dynamic programming approach and Fast Fourier Transform achieves an average execution time of 2.62 seconds and consumes an average of 19 MB of memory, utilizing only 1.23% of the available memory limit.**

Index Terms—**dynamic programming, fast Fourier transform, hash polynomials, set permutation**

I. INTRODUCTION

SUPPOSE a polynomial hash function $H(S)$ is defined. By converting each letter in string S into a numerical value based on its position in the alphabet, the string can be represented as a fixed value called a polynomial hash value. For example, $S = \text{abde}, B = 26, M = 36$, the string has a length of $N = 4$. Thus, $H(S) = (26^3 * 0 + 26^2 * 1 + 26^1 * 3 + 26^0 * 4)\ \%\ 36 = 2$. Since each polynomial hash value of a string is taken modulo M according to (1), different strings can have the same polynomial hash value, with the result always being less than M.

$$H(S) = \sum_{i=0}^{N-1} B^{N-i-1} * D(S_i)\ \%\ M \qquad (1)$$

Manuscript received November 11, 2024; revised March 20, 2025. This work was supported in part by Sepuluh Nopember Insitute of Technology, Surabaya, Indonesia.

Fadel Pramaputra Maulana is a graduate student at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia. (e-mail: fadelpm2002@gmail.com)

Rully Soelaiman is an associate professor at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia (e-mail: rully130270@gmail.com)

Yudhi Purwananto is an associate professor at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia (e-mail: purwananto@gmail.com)

The problem discussed in this paper is the calculation of the number of combinations of lowercase characters that can be formed such that the polynomial hash value of the string satisfies a given value. For instance, to calculate $H(S)$ with $N_{maks} = 2,\ M = 2,\ B = 26$, using (1) and Table I, while ensuring $0 \leq H(S) < M$, the numerical value of each character from "a" to "zz" are assigned as in Table II.

TABLE I
NUMERICAL VALUE FOR EACH CHARACTER

S_i	$D(S_i)$	S_i	$D(S_i)$
a	0	n	13
b	1	o	14
c	2	p	15
d	3	q	16
e	4	r	17
f	5	s	18
g	6	t	19
h	7	u	20
i	8	v	21
j	9	w	22
k	10	x	23
l	11	y	24
m	12	z	25

The number of hash values that can be generated when $N_{max} = 1$ and $N_{max} = 2$ corresponds to the number of character combinations for string lengths of 1 and 2, which totals 702. Using the method described above, if we set $N_{max} = 30000$ and $M = 30000$, the time complexity of the calculation becomes $O(10^{42449})$ and requires 10^{42440} seconds which is approximately $3.17\,x\,10^{42442}$ years. Therefore, a brute-force approach for computing the number of combinations of lowercase characters that satisfy a given hash value is not a feasible solution for real-world applications.

TABLE II
EXAMPLE CASE $N_{max} = 2$

No	$H(S)$	Hash Calculation	Hash Value
1	$H(\text{“}a\text{”})$	$(1 * 0)\ \%\ 2$	0
...	...	...	...
26	$H(\text{“}z\text{”})$	$(1 * 25)\ \%\ 2$	1
27	$H(\text{“}aa\text{”})$	$(26 * 0 + 1 * 0)\ \%\ 2$	0
...	...	...	...
701	$H(\text{“}zy\text{”})$	$(26 * 25 + 1 * 24)\ \%\ 2$	0
702	$H(\text{“}zz\text{”})$	$(26 * 25 + 1 * 25)\ \%\ 2$	1

In order for this problem to be solved efficiently, another approach is required, that is by using dynamic programming method. Dynamic programming is a problem-solving technique by decomposing the solution into a set of steps or stages in such a way that the solution of a problem can be viewed from a series of small decisions which are related to one another [1]. In defining the dynamic programming model, it is essential to determine the relationship between a given state and its previous state so that a recurrence equation can be defined [9]. A recurrence relation is an equation that defines each element of a sequence as a function of the previous elements. Therefore, the solution for each state depends on the solutions for smaller states of the same problem [3]. In the process, this approach involves computing polynomial multiplication. Thus, the Fast Fourier Transform (FFT) method is used to speed up the naïve polynomial multiplication. The Fast Fourier Transform method is a method used to calculate the Discrete Fourier Transform (DFT) efficiently. By using the FFT method, which utilizes the special properties of complex roots of unity, the DFT calculation can be done in $O(M\,log_2 M)$ time [2]. In this problem, the FFT method will be used to reduce the time complexity of calculating polynomial multiplication from $O(M^2)$ to $O(M\,log_2 M)$.

By taking previous research on the topics of hash polynomials and fast polynomial multiplication [4], [5], [6], [7], [8], [11], it can be concluded that dynamic programming and FFT-based approaches provide a feasible solution to this problem. Therefore, in this paper, we propose a solution that takes only 2.62 seconds on average to compute a given number of hash values in a time efficient manner and satisfies the problem time constraint of 10 seconds.

The rest of the paper is organized as follows: Section II presents our novel method: dynamic programming technique and Fast Fourier Transform method. Section III presents the experimental results and analysis. Finally, the conclusion is stated in section IV.

II. Methodology

Our proposed algorithm to count the number of character combinations with a given hash value utilizes dynamic programming techniques for all hashes less than M and applies the Fast Fourier Transform for polynomial multiplication. In general, the algorithm is divided into 3 parts: first, it determines whether the input is a unique case or not. If the input is a unique case, then the solution can be done with $\log_2 N$ calculations. Otherwise, if the input is not a unique case, the number of hash values for maximum string length equal to 2^k is first calculated using dynamic programming techniques. Then, the number of hash values for a maximum string length not equal to 2^k is calculated using the results from the previous part and dynamic programming techniques. In both the second and third parts, state calculations in dynamic programming involve polynomial multiplication. To improve efficiency, the naïve $O(M^2)$ polynomial multiplication is replaced with the Fast Fourier Transform (FFT), reducing the complexity to $O(M \log_2 M)$. The following discussion consists of six main sections. Section II-A explains the state model used in dynamic programming. Section II-B and II-C describe the recurrence relation used in calculating the number of hash values for a maximum string length equal to and not equal to 2^k. Section II-D explains how Fast Fourier Transform is utilized in accelerating the calculation of polynomial multiplication. Section II-E addresses the handling of unique cases, followed by Section II-F, which presents the time complexity analysis of our proposed method.

A. Dynamic Programming State Model

If the sigma form of (1) is converted to an explicit sum form, we obtain (2). To calculate the number of combinations of N characters such that $H(S)$ takes values from 0 to $M-1$, start by calculating the number of hash values when the string length consists of only one character, with each character value as in Table I. Equation (2) shows each segment of the sum is denoted as T_1 to T_N, where T_i is the set consisting of the number of hash values from 0 to $M-1$ when the string length consists of one character at the i-th character from the last character in the string. As a result, Equations (3) and (4) are derived. Equation (3) shows $T_i = \{h_0, h_1, \ldots, h_{M-1}\}$ where h_i represents the number of occurrences of hash value i. As illustrated in Fig. 1, each T in (4) produces hash values raging from 0 to $M-1$.

$$H(S) = \left(B^{N-1}D(S_0) + \cdots + B^0 D(S_{N-1})\right) \% \, M \tag{2}$$

$$T_i = B^{i-1}D(S_{N-i}) \% \, M, 1 \le i \le N \tag{3}$$

$$H(S) = (T_N + T_{N-1} + \cdots + T_2 + T_1) \% \, M \tag{4}$$

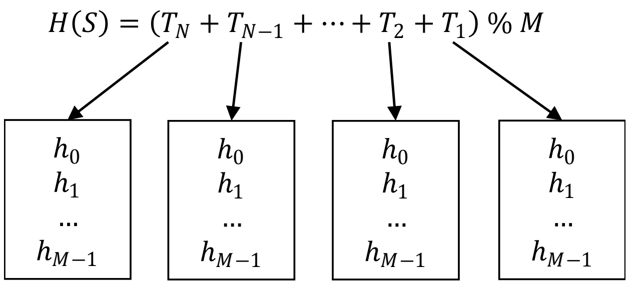


Fig. 1. The number of hash values for each T

Given T_p and T_q where each is the set number of hash values $\{h_0, h_1, \ldots, h_{M-1}\}$, these sets can be transformed into polynomial equations. Here, the values h_0 to h_{M-1} in each T are the coefficients of a polynomial, where the exponent corresponds to the hash value. Thus, T_p is defined as $h_0 x^0 + h_1 x^1 + \cdots + h_{M-1} x^{M-1}$, and similarly for T_q. To calculate the number of hash values from 0 to $M-1$ for a string of length two can be done by multiplying h_i with h_j at two T in (5). The result of the multiplication gives the number of hashes with value $i+j$, as shown in Table III. Therefore, $T_p T_q$ represents the set of hash values from 0 to $M-1$ when the string consists of two characters, spanning from the p-th character to the q-th character, starting from the end of the string.

$$T_p T_q\left(h_{i+j}\right) = T_p(h_i) * T_q\left(h_j\right), \quad 0 \le i, j \le M-1 \tag{5}$$

TABLE III
CALCULATION OF NUMBER OF HASH VALUES FOR $N = 2$

T_p	T_q	$T_p * T_q$
h_0	h_0	h_{0+0}
h_0	h_1	h_{0+1}
...	...	...
h_1	h_0	h_{1+0}
h_1	h_1	h_{1+1}
...	...	...
h_i	h_j	$h_{k=i+j}$
...	...	...
h_{M-1}	h_{M-1}	$h_{2(M-1)}$

$$h_{k\%M} = \begin{cases} h_k, & k < M \\ h_{k\,\%\,M} + h_k, & M \le k \le 2(M-1) \end{cases} \tag{6}$$

Referring to (5), the string hash polynomial function has modulo M, thus if the value of $i + j = k$ in T_pT_q, k does not exceed M. Therefore, for all h_k in T_pT_q, a formula as in (6) is applied. This formula is also used to calculate the number of hash values for strings longer than two characters, as it does not change the number of sets T consisting of h_0 to h_{M-1}. Equation (2) shows the value of each segment of the sum is differentiated by a multiplier B. T_1 and T_2 are differentiated by a multiplier B^1, T_1 and T_3 by B^2, and so on. Thus, to calculate the set on T_q, permutation can be applied to T_p using the multiplier B^{q-p} and T_p, given that $q > p$. This permutation is performed by multiplying B^{q-p} with each hash value in T_p, altering the order of the number of hash values in the T_p set so that it transforms into the T_q set, as shown in Table IV.

TABLE IV
EXAMPLE OF PERMUTATION ON T WITH MULTIPLIER B

T_1	MULTIPLIER	T_3
h_0	B^2	$h_{(0\,*\,B^2)\%M}$
h_1	B^2	$h_{(1\,*\,B^2)\%M}$
...	B^2	...
h_{M-2}	B^2	$h_{\left((M-2)\,*\,B^2\right)\%M}$
h_{M-1}	B^2	$h_{\left((M-1)\,*\,B^2\right)\%M}$

$$\begin{gathered} T_{p+r} = permut\left(T_p, B^r\right) \\ h_{(i*B^{q-p})\%M} = h_{(i*B^{q-p})\%M} + h_i, \quad 0 \le i < M \end{gathered} \tag{7}$$

In Table IV, the hash value at T_3 is taken modulo M to ensure that the result does not exceed M. Given T_p, the set at T_q can be determined by permuting the set at T_p using the multiplier B^r, as shown in (7), where $r = q - p$ and $q > p$. There are several permutation operations that are true,

1. $T_{p+r} = permut(T_p, B^r)$
2. $T_{1+r}T_{2+r} \dots T_{p+r} = permut(T_1T_2 \dots T_p, B^r)$
3. $T_{1+r} + T_{1+r}T_{2+r} + \cdots + T_{1+r} \dots T_{p+r} =$
   $permut(T_1 + T_1T_2 + \cdots + T_1 \dots T_p, B^r)$

The set $T_1T_2 \dots T_p$ consists of the number of hash values from 0 to $M - 1$, given that the string length consists of p characters, spanning from the first to the p-th character, starting from the end of the string. Whereas $T_1 + T_1T_2 + \cdots + T_1 \dots T_p$ represents the number of sets consisting of the number of hash values from 0 to $M - 1$ when the string length is less than or equal to p characters. The symbol (+) in the equation denotes summation between sets of T when T is represented as a polynomial equation. The use of this permutation function is crucial as it significantly reduce the number of calculations needed to generate a new T set from an existing one.

B. Recurrence Relation when String Length is 2^k

If there is a value $N_{\max}$ and U represents the number of hash values when the string length is less than or equal to N_{max}, then U is the union of the sets of T, as shown in (8). If $N = 2^k$ for k integers, the set U can be divided into smaller parts, as illustrated in Fig. 2.

$$U = T_1 + T_1T_2 + \cdots + T_1T_2 \dots T_{N_{maks}-1}T_{N_{maks}} \tag{8}$$

$$U = \overbrace{T_1}^{a} + \overbrace{T_1T_2}^{b} + T_1T_2(\overbrace{T_3}^{c}) + \overbrace{T_1T_2T_3T_4}^{d} + \cdots$$

Fig. 2. A set U that is divided into smaller parts

Based on Fig. 2 there are several conclusions, such that, b can be created from $a * permut(a, B^1)$. Moreover, d can be created from $b * permut(b, B^2)$. Lastly, c can be created from $permut(a, B^2)$. To determine U with a length of N_{max}, $\sqrt{N_{max}}$ iterations are performed to find the number of hash values of the character combination. If $T_{total}\left[\sqrt{N_{max}}\,\right] = U$ and U represents the number of hash values from 0 to $M - 1$ with length N_{max}, then a recurrence relation is obtained when N_{max} is 2^k with k integers, as shown in (9) and (10). The following is an explanation of the variables in (9) and (10),

- h_i = the number of hashes with a value of i.
- T_1 = the set $\{h_0, h_1, \dots, h_{M-1}\}$ when $N = 1$ at the last character of the string.
- $T_{total}[k]$ = the set $\{h_0, h_1, \dots, h_{M-1}\}$ representing string length combinations from $N = 1$ to $N = 2^k$.
- $T_{exact}[k]$ = the set $\{h_0, h_1, \dots, h_{M-1}\}$ representing string length combinations exactly at $N = 2^k$.
- $permut(A, b)$ = the permutation function applied to set $A = \{h_0, h_1, \dots, h_{M-1}\}$ with multiplier b.
- B = the base in the polynomial hash function.

$$T_{exact}[i] = \begin{cases} T_1, \; i = 0 \\ T_{exact}[i-1] * permut\left(T_{exact}[i-1], \, B^{2^{i-1}}\right), \; 2 \le 2^i \le N \end{cases} \tag{9}$$

$$T_{total}[i] = \begin{cases} T_1,\ i = 0 \\ T_{total}[i-1] + T_{exact}[i],\ i = 1 \\ T_{total}[i-1] + T_{exact}[i-1] * permut\left(T_{total}[i-1],\ B^{2^{i-1}}\right),\ 2 \le 2^i \le N \end{cases} \tag{10}$$

$$U[i] = \begin{cases} permut\begin{pmatrix} T_{total}\left[bit[i]\right], \\ B^{N-2^{bit[i]}} \end{pmatrix},\ i = 0 \\ permut\begin{pmatrix} T_{total}\left[bit[i]\right], \\ B^{N-2^{bit[0]}-\dots-2^{bit[i]}} \end{pmatrix} + permut\begin{pmatrix} T_{exact}\left[bit[i]\right], \\ B^{N-2^{bit[0]}-\dots-2^{bit[i]}} \end{pmatrix} * U[i-1],\ 1 \le i \le k \end{cases} \tag{11}$$

Based on the recurrence relation in (9) and (10), Algorithm 1 is used to calculate the number of hash values when the string length is equal to 2^k.

```
Algorithm 1 Solve recurrence relation when string length is equal to 2^k
Input: dp1, dp2, B, M, base, j
Output: dp1, dp2
  for i ← 0 to MIN(26, M)-1 do
      for k ← 0 to 25 do
          h ← (B*i+k) % M
          dp2[1][h] ← dp2[1][h] + dp1[1][i]
  dp1[1] ← ADD(dp1[1], dp2[1], M)
  for i ← 2 to j do
      pt ← PERMUT(dp2[i-1], base[i-1], M)
      dp2[i] ← MUL(dp2[i-1], pt, M)
      pt ← PERMUT(dp1[i-1], base[i-1], M)
      mul ← MUL(dp2[i-1], pt, M)
      dp1[i] ← ADD(dp1[i-1], mul, M)
  return dp1, dp2
```

C. *Recurrence Relation when String Length is Not* 2^k

To calculate the number of hash values from 0 to $M-1$ for a string length that is not a set of 2^k (where k is an integer), a method is required to determine which set of hash values at which length to use. This can be achieved by examining the bits that form the binary representation of N_{max}. For example, Fig. 3 illustrates the binary representations of 15 and 25.

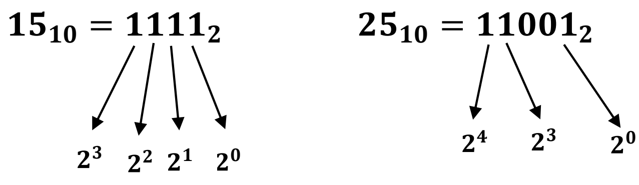


Fig. 3. Binary representation of the numbers 15 and 25

In Fig. 3, the number 15 can be formed using $2^0, 2^1, 2^2, 2^3$, while the number 25 can be formed using $2^0, 2^3, 2^4$. Similarly, to compute the set $\{h_0, h_1, \dots, h_{M-1}\}$ for a length of N_{max}, the lengths used to build the hash will correspond to the powers of two represented by the positions of the 1-bits in the binary representation of N_{max}. Suppose,

- h_i = the number of hashes with a value of i.
- $T_{total}[k] =$ the set $\{h_0, h_1, \dots, h_{M-1}\}$ representing string length combinations from $N = 1$ to $N = 2^k$.
- $T_{exact}[k] =$ the set $\{h_0, h_1, \dots, h_{M-1}\}$ representing string length combinations exactly at $N = 2^k$.
- $permut(A, b) =$ the permutation function applied to set $A = \{h_0, h_1, \dots, h_{M-1}\}$ with multiplier b.
- $B =$ the base in the polynomial hash function.
- $expo =$ the exponent of B used for multiplication in permutations.
- $bit =$ the sequence of bits in N from its binary representation, starting from the first bit (only bits with a value of 1 are considered).

To calculate the numbers of hash values when N is not equal to 2^k, the numbers of hash values for N is equal to the powers of 2 is used. The set that must be formed to solve the problem with N_{max} is $U = T_1 + T_1T_2 + T_1T_2T_3 + \dots + T_1 \dots T_{N_{max}}$. Iteration continues until all bit with a value of 1 in the binary representation N_{max} have been traversed. This results in U, the set of hash values raging from 0 to $M-1$ when the maximum string length is N_{max}. If $bit[i]$ represents the position of a bit with value 1 in the binary representation of N_{max}, where $0 \le i \le k$ and $bit[k]$ is the position of the last bit with value 1, then the recurrence relation when $N_{max} \ne 2^k$ (i.e., when N_{max} is not equal to 2^k for integers k) is shown in (11).

The total number of hash values from 0 to $M-1$ for a string length that is not 2^k is represented by $U[k]$. The addition and multiplication in the recurrence relation in (11) correspond to polynomial addition and multiplication, where $U[k][j]$ is the coefficient of the j-th polynomial term, with the limit $0 \le j < M$. The coefficient of the j-th term represents the number of hash values with a value of j. It is certain that calculating $U[i]$ only requires the value of $U[i-1]$, while storing $U[0]$ to $U[i-2]$ is unnecessary. This approach minimizes memory usage. Using this method, the number of hash values that need to be calculated is $\log_2 N$, excluding the polynomial addition and multiplication calculations, where N is the maximum string length in (11). Based on (11), Algorithm 2 is used to calculate the number of hash values when the string length is not equal to 2^k.

```
Algorithm 2 Solve recurrence relation when string length is not equal to 2^k
Input: N, M, twopowers, base, dp1, dp2
Output: ans
  i, j ← 0
  while twopowers[i] ≤ N do
      if N & twopowers[i] then
          if j = 0 then
              temp ← N - twopowers[i]
              nums[j] ← temp
          else
              temp ← nums[j-1] - twopowers[i]
```

Algorithm 2 Solve recurrence relation when string length is not equal to 2^k

```
            $nums[j] \leftarrow temp$
        $exp[j] \leftarrow i$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
for $i \leftarrow 0$ to $j - 2$ do
    $k \leftarrow exp[i+1]$
    $L \leftarrow nums[i+1]$
    $m \leftarrow$ FINDMULTIPLY($base, L, M$)
    if $i = 0$ then
        $f \leftarrow$ FINDMULL($base, nums[i], M$)
        if $L \neq 0$ then
            $pt1 \leftarrow$ PERMUT($dp2[k], m, M$)
            $pt2 \leftarrow$ PERMUT($dp1[exp[i]], f, M$)
            $ans \leftarrow$ MUL($pt1, pt2, M$)
        else
            $pt1 \leftarrow$ PERMUT($dp1[exp[i]], f, M$)
            $ans \leftarrow$ MUL($dp2[k], pt1, M$)
    else
        if $L \neq 0$ then
            $pt1 \leftarrow$ PERMUT($dp2[k], m, M$)
            $ans \leftarrow$ MUL($pt1, ans, M$)
        else
            $ans \leftarrow$ MUL($dp2[k], ans, M$)
    if $L \neq 0$ then
        $pt1 \leftarrow$ PERMUT($dp1[k], m, M$)
        $ans \leftarrow$ ADD($pt1, ans, M$)
return $ans$
```

D. *Polynomial Multiplication with Fast Fourier Transform*

It is known that multiplying two polynomials $A(x)$ and $B(x)$, with degrees a and b, requires $O\big((a+1)*(b+1)\big)$ time complexity since there are $a+1$ elements in $A(x)$ and $b+1$ elements in $B(x)$. However, there exists a multiplication method that reduces the complexity to $O(n)$, known as pointwise multiplication, when the polynomials are represented in point-value form. As mentioned in *Introduction to Algorithm* by Cormen et al page 901-902, a polynomial with a degree bound of n can be represented using at least n different points as values in the polynomial basis. For example, given $A(x) = x^2 - 1$ and $B(x) = x$, these polynomials can be represented as point-values using at least three points for $A(x)$ and two points for $B(x)$, as illustrated in Fig. 4.

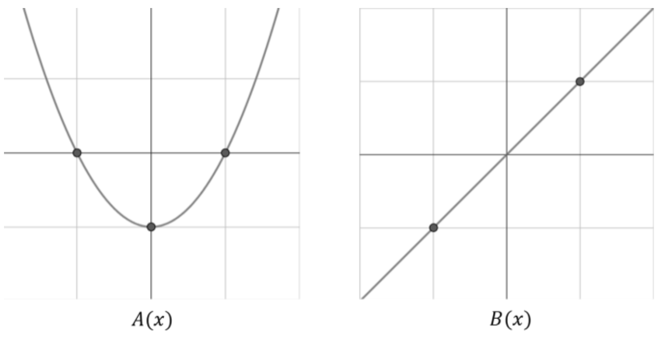


Fig. 4. Graphs of polynomials $A(x)$ and $B(x)$ on the cartesian coordinate plane

$$C(x) = \{(x_0, C(x_0)), (x_1, C(x_1)), \dots, (x_{n-1}, C(x_{n-1}))\} \qquad (12)$$

If there is a polynomial $C(x)$ with bound degree n, it can be represented using the set in (12). In polynomial multiplication, if $C(x) = A(x) * B(x)$, where $A(x)$ is a polynomial of degree a and $B(x)$ is a polynomial of degree b, then the product of polynomial $C(x)$ has a degree of $c = a + b$ or a polynomial with bounded degree of $c + 1$. Thus, at least $c + 1$ points are required to represent the polynomial $C(x)$ in point-value form. If multiplication is performed using pointwise multiplication, then $a + b + 1$ different points are required for polynomials $A(x)$ and $B(x)$. After the pointwise multiplication, $C(x)$ can be converted back from point-value representation to coefficient representation through a process known as interpolation.

Naïve polynomial evaluation at n points requires $O(n^2)$ complexity. However, in this section, the FFT (Fast Fourier Transform) is used to reduce the complexity to $O(n \log_2 n)$. The general scheme for polynomial multiplication is illustrated in Fig. 5. It is assumed that polynomial $A(x)$ has a degree bounded by a set of 2^k numbers. Therefore, Algorithm 3 is used to iteratively convert $A(x)$ between point-value representation and coefficient representation.

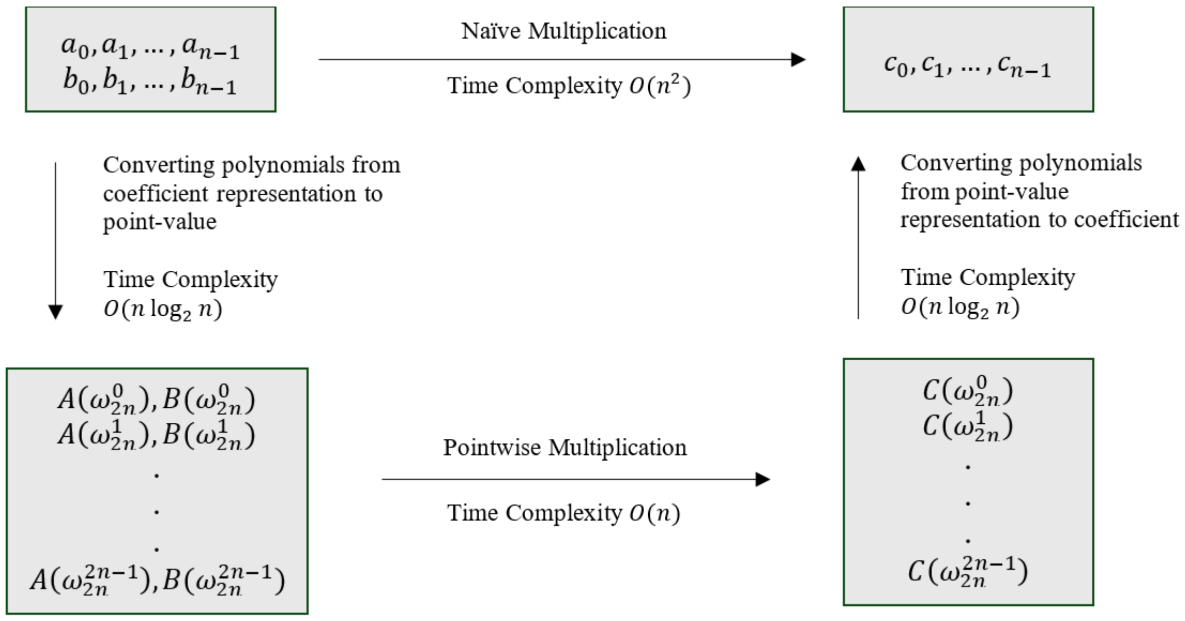
Algorithm 3 Iterative FFT

```
Input: $a, isInverse$
Output: $y$
$a \leftarrow$ BIT-REVERSAL-PERMUTATION($a$)
$n \leftarrow a.Length$
for $s \leftarrow 1$ to $\log_2 n$ do
    $m \leftarrow 2^s$
    if $isInverse = 1$ then
        $\omega_m \leftarrow e^{-2\pi i/n}$
    else
        $\omega_m \leftarrow e^{2\pi i/n}$
    for $k \leftarrow 0$ to $n$-1 by $m$ do
        $\omega \leftarrow 1$
        for $j \leftarrow 0$ to $m/2$-1 do
            $t \leftarrow \omega * a[k + j + m/2]$
            $u \leftarrow a[k + j]$
            $a[k + j] \leftarrow u + t$
            $a[k + j + m/2] \leftarrow u - t$
            $\omega \leftarrow \omega * \omega_m$
if $isInverse = 1$ then
    $a \leftarrow (a_0/n, a_1/n, \dots, a_{n-1}/n)$
return $a$
```

E. *Unique Cases*

Based on (2), if B is divisible by M and $M \geq 26$, then all the sets of hash values, which are greater than 0 in the first to the second last character, are 0. Equation (13) shows the sum of the hash values forms a geometric sequence, as illustrated in Fig. 6.

$$H(S) = (0 + 0 + \cdots + 0 + T_1) \,\%\, M \qquad (13)$$



$a_0, a_1, \ldots, a_{n-1}$
$b_0, b_1, \ldots, b_{n-1}$
Naïve Multiplication
Time Complexity $O(n^2)$
$c_0, c_1, \ldots, c_{n-1}$
Converting polynomials from coefficient representation to point-value
Time Complexity $O(n \log_2 n)$
Converting polynomials from point-value representation to coefficient
Time Complexity $O(n \log_2 n)$
$A(\omega_{2n}^0), B(\omega_{2n}^0)$
$A(\omega_{2n}^1), B(\omega_{2n}^1)$
$A(\omega_{2n}^{2n-1}), B(\omega_{2n}^{2n-1})$
Pointwise Multiplication
Time Complexity $O(n)$
$C(\omega_{2n}^0)$
$C(\omega_{2n}^1)$
$C(\omega_{2n}^{2n-1})$

Fig. 5. Efficient Polynomial Multiplication Scheme

Therefore, formula (14) is used to calculate the number of hash values for the case where $GCD(B, M) \geq 26$ and $B \geq M$. For values of i where $0 \leq i \leq 25$, the result is taken modulo $10^9 + 7$ to ensure that the output remains within the valid range. Computing 26^N can be done with $O(\log_2 N)$ time complexity using binary exponentiation. Since 26^N is large, each iteration of binary exponentiation is taken modulo 1000000007 to prevent integer overflow in certain programming languages. Therefore, the concept of inverse modular (15) is used and (14) is transformed into (16), which is used to calculate the number of hash values when $GCD(B, M) \geq 26$ and $B \geq M$, with $mult = 280000002$ and $mod = 1000000007$.

$\boldsymbol{h_0 = 1}$	$\boldsymbol{h_0 = 1 + 26^1}$	$\boldsymbol{h_0 = 1 + 26 + \cdots + 26^{N-1}}$
…	…	…
$\boldsymbol{h_{25} = 1}$	$\boldsymbol{h_{25} = 1 + 26^1}$	$\boldsymbol{h_{25} = 1 + 26 + \cdots + 26^{N-1}}$
$\boldsymbol{h_{26} = 0}$	$\boldsymbol{h_{26} = 0}$	$\boldsymbol{h_{26} = 0}$
…	…	…
$\boldsymbol{h_{M-1} = 0}$	$\boldsymbol{h_{M-1} = 0}$	$\boldsymbol{h_{M-1} = 0}$
$\boldsymbol{\lvert string \rvert = 1}$	$\boldsymbol{\lvert string \rvert = 2}$	$\boldsymbol{\lvert string \rvert = N}$

Fig. 6. Geometric sequence in the unique case where greatest common divisor $GCD(B, M) \geq 26$ and $B \geq M$

$$h_i = \begin{cases} \dfrac{26^N - 1}{25} \% \, mod, & 0 \leq i \leq 25 \\ 0, & 25 < i < M \end{cases} \tag{14}$$

$$\frac{a}{b} \% \, mod = \big((a \% \, mod) * (b^{-1} \% \, mod)\big) \% \, mod \tag{15}$$

$$h_i = \begin{cases} \big((26^N - 1) * mult\big) \% \, mod, & 0 \leq i \leq 25 \\ 0, & 25 < i < M \end{cases} \tag{16}$$

F. Time Complexity Analysis

Equation (9) shows that calculating T_{exact} involves $O(M)$ permutations and $O(M \log_2 M)$ polynomial multiplications, repeated $\log_2 N$ times. This result in a total of $(\log_2 N)(M + M \log_2 M)$ operations. Similarly, in (10), calculating T_{total} involves $O(M)$ permutations, $O(M)$ polynomial additions, and $O(M \log_2 M)$ multiplications, leading to $(\log_2 N)(2M + M \log_2 M)$ operations. Thus, the total operations for (9) can be simplified to:

- $(\log_2 N) * (M + M \log_2 M) + (\log_2 N) * (2M + M \log_2 M)$,
- $(\log_2 N) * (M + 2M + M \log_2 M + M \log_2 M)$,
- $(\log_2 N) * (3M + 2M \log_2 M)$.

Equation (11) shows that U requires two permutations $O(2M)$, $O(M)$ polynomial addition, and $O(M \log_2 M)$ multiplication over $\log_2 N$, resulting in $(\log_2 N)(3M + M \log_2 M)$ operations. Combining (9), (10), and (11) gives:

- $(\log_2 N) * (3M + 2M \log_2 M) + (\log_2 N) * (3M + M \log_2 M)$,
- $(\log_2 N) * (3M + 2M \log_2 M + 3M + M \log_2 M)$,
- $(\log_2 N) * (6M + 3M \log_2 M)$,
- $(\log_2 N) * \big(M(6 + 3 \log_2 M)\big) \approx O(\log_2 N) * (M * \log_2 M)$.

Using this method, the complexity of calculating the number of hash values is $O(\log_2 N * M * \log_2 M)$, where N is the maximum string length and M is the modulus. For the unique cases described in (16), the only non-constant operation is calculating the value of 26^N. This can be done using binary exponentiation, which has a time complexity of $O(\log_2 N)$. In conclusion, the overall complexity for unique cases is $O(\log_2 N)$.

III. Experimental Results and Analysis

The proposed algorithm was tested for both correctness and performance. The implementation code was submitted to the Sphere Online Judge (SPOJ) site to evaluate its accuracy and efficiency. To compare the performance results, the implementation code was submitted 10 times.

Section III-A described the validity check of the dynamic programming method and the FFT used to calculate the number of hash values for a given string length. In addition, Section III-B further discusses the performance check of the proposed method, both in time-wise and space-wise.

A. Validity Examination

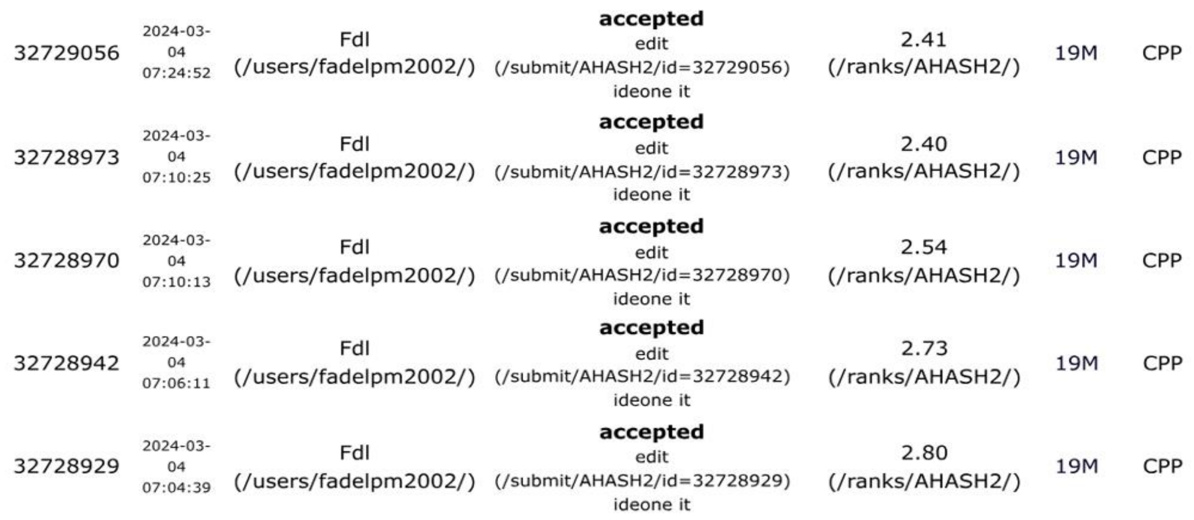
Fig. 12 shows the status on the Sphere Online Judge website of the approach used in this problem. Sphere Online Judge will provide various responses based on its judgement to the solution submitted. The **"Accepted"** status indicates that the program ran successfully and gave the correct answer, the **"Wrong Answer"** status means that the program ran successfully, but gave the wrong answer, **"Time Limit Exceeded"** indicates that the program compiled successfully, but exceeded the time limit, **"Compilation Error"** means that the program could not be compiled, and finally the **"Runtime Error"** status implies that the program compiled successfully, but exited with a runtime error or crashed.

Code testing was performed 10 times to ensure accuracy and validity during the test run. All 10 submissions received an **"Accepted"** response which proves that the proposed approach to calculate the number of hash values in a given string using dynamic programming techniques and the Fast Fourier Transform method can provide the correct answer within the time and memory limitation. The **"Accepted"** status is given only if the code passes all the test cases, proving how valid the solution proposed in this paper is. Fig. 7 displays the validity examination of our method on the problems available in Sphere Online Judge.

B. Performance Examination

There are two factors to consider in performance checks, which are program runtime and memory usage [10]. The first factor will be tested in local environment using the PC used in this research, and an external environment using the Sphere Online Judge (SPOJ) website.

Runtime: to evaluate performance, two graphs were generated based on empirical analysis, each of which has a change in value with respect to the other variable. In the local testing, the test cases used samples with $N = 30000$ and varying M values, as well as varying N values and $M = 30000$, with each case tested 10 times. There are four columns in the local test data as shown in Tables V and VI. The first column shows the test data number. The second column contains the test case, where the first row specifies the number of test cases, T. Each of these T cases is followed by a row with four numbers $B, M, N,$ and Q, representing the polynomial basis, modulus, maximum string length, and number of queries, respectively. Each query includes one value H, the hash value. The third column indicates the output, which is the count of hash values equal to H. The fourth column records the program's runtime duration. The graphs in Figs. 8, 9, and 10 show a a linear increase in time with respect to the values $\log_2 N$ and $M * \log_2 M$, indicating the time complexity of the method is $O(\log_2 N * M * \log_2 M)$. For external test, the average execution time is 2.62 seconds, which can be seen in Fig.11.



32729056	2024-03-04 07:24:52	Fdl (/users/fadelpm2002/)	**accepted** edit (/submit/AHASH2/id=32729056) ideone it	2.41 (/ranks/AHASH2/)	19M	CPP
32728973	2024-03-04 07:10:25	Fdl (/users/fadelpm2002/)	**accepted** edit (/submit/AHASH2/id=32728973) ideone it	2.40 (/ranks/AHASH2/)	19M	CPP
32728970	2024-03-04 07:10:13	Fdl (/users/fadelpm2002/)	**accepted** edit (/submit/AHASH2/id=32728970) ideone it	2.54 (/ranks/AHASH2/)	19M	CPP
32728942	2024-03-04 07:06:11	Fdl (/users/fadelpm2002/)	**accepted** edit (/submit/AHASH2/id=32728942) ideone it	2.73 (/ranks/AHASH2/)	19M	CPP
32728929	2024-03-04 07:04:39	Fdl (/users/fadelpm2002/)	**accepted** edit (/submit/AHASH2/id=32728929) ideone it	2.80 (/ranks/AHASH2/)	19M	CPP

Fig. 7. Validity examination of the program by the Sphere Online Judge site

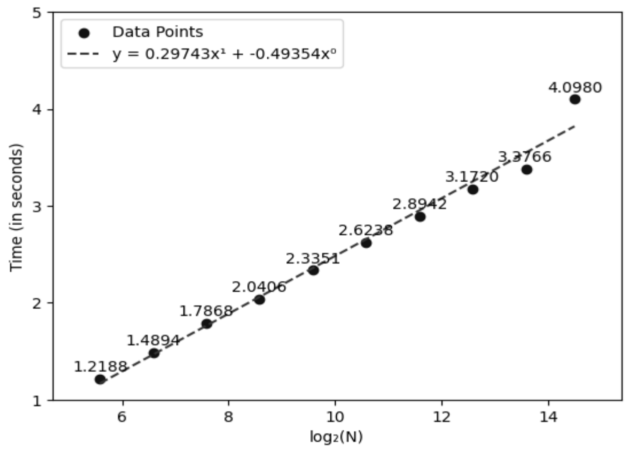


Fig. 8. Local performance time graph of the dynamic programming and Fast Fourier Transform method with respect to N for test case numbers 1 to 10 from Table V

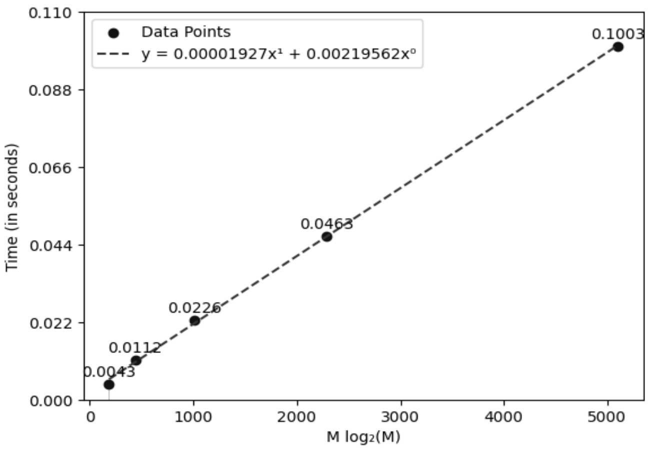


Fig. 9. Local performance time graph of the dynamic programming and Fast Fourier Transform method with respect to M for test case numbers 1 to 5 from Table VI

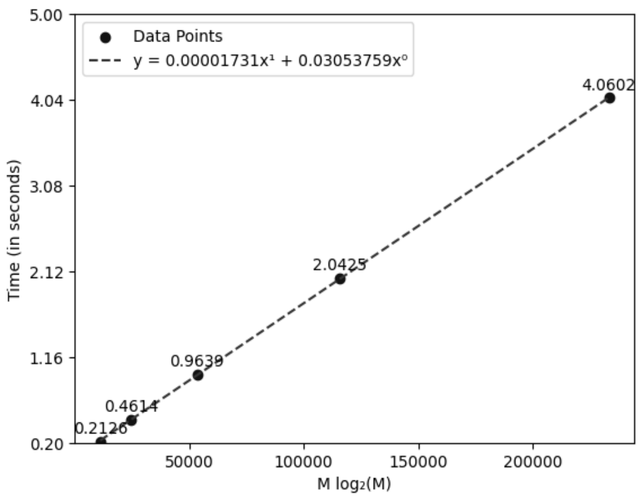


Fig. 10. Local performance time graph of the dynamic programming and Fast Fourier Transform method with respect to M for test case numbers 6 to 10 from Table VI

TABLE V
LOCAL TEST VALUES RESPECT TO N

No	Input	Output	Time (seconds)
1	1 20107 30000 48 1 0	Case 1: 673858768	1,2188
2	1 20107 30000 96 1 0	Case 1: 69750134	1,48938
3	1 20107 30000 192 1 0	Case 1: 296116833	1,78683
4	1 20107 30000 384 1 0	Case 1: 776510975	2,04062
5	1 20107 30000 768 1 0	Case 1: 908161167	2,33512
6	1 20107 30000 1536 1 0	Case 1: 609534788	2,62378
7	1 20107 30000 3072 1 0	Case 1: 248638238	2,89419
8	1 20107 30000 6144 1 0	Case 1: 608724507	3,17197
9	1 20107 30000 12288 1 0	Case 1: 631154566	3,37661
10	1 20107 30000 23192 1 0	Case 1: 4806455	4,09799

Memory Usage: as mentioned at the beginning of this section, the memory usage test utilized the Sphere Online Judge website, where the dynamic programming method and FFT used to calculate the number of hash values for a given string length only required a constant memory of 19 MB as shown in Fig. 13.

TABLE VI
LOCAL TEST VALUES RESPECT TO M

No	Input	Output	Time (seconds)
1	1 20107 48 30000 1 0	Case 1: 259407919	0,004335
2	1 20107 96 30000 1 0	Case 1: 563508849	0,011224
3	1 20107 192 30000 1 0	Case 1: 933059172	0,02264
4	1 20107 384 30000 1 0	Case 1: 401401803	0,046328
5	1 20107 768 30000 1 0	Case 1: 869182996	0,100273
6	1 20107 1536 30000 1 0	Case 1: 167383340	0,212619
7	1 20107 3072 30000 1 0	Case 1: 310041053	0,461427
8	1 20107 6144 30000 1 0	Case 1: 981530095	0,963927
9	1 20107 12288 30000 1 0	Case 1: 68348150	2,04252
10	1 20107 23192 30000 1 0	Case 1: 382152528	4,06024

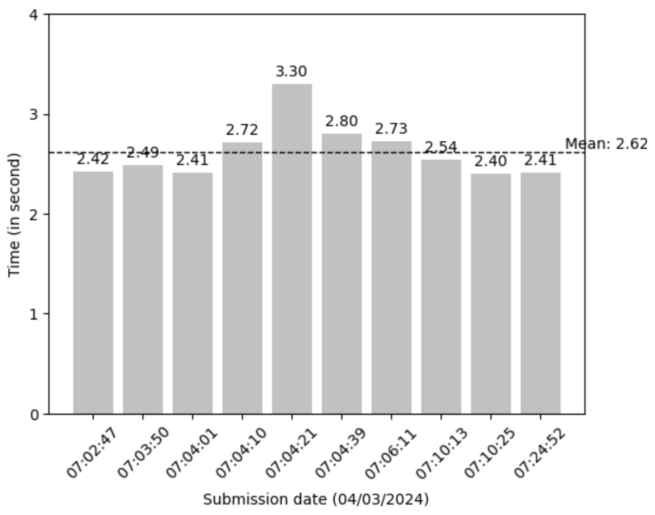
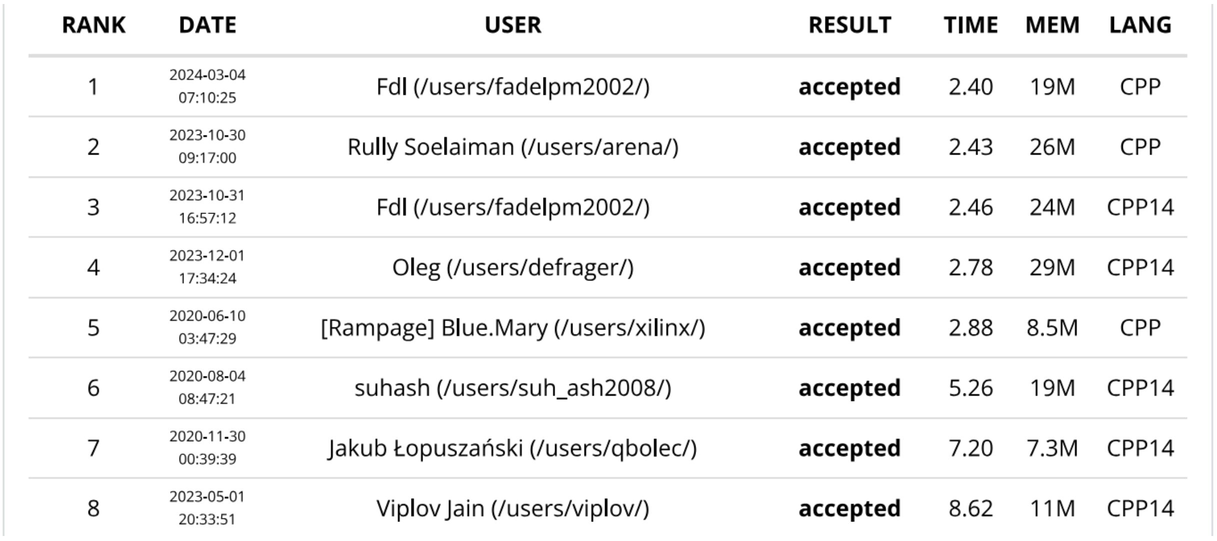


Fig. 11. External performance time graph of dynamic programming technique and Fast Fourier Transform method by Sphere Online Judge

As illustrated in Fig. 12, the dynamic programming method and Fast Fourier Transform achieves the best time complexity on the Sphere Online Judge website. This proves that the dynamic programming method and Fast Fourier Transform can successfully solve the problem with the best execution time at present.



RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2024-03-04 07:10:25	Fdl (/users/fadelpm2002/)	**accepted**	2.40	19M	CPP
2	2023-10-30 09:17:00	Rully Soelaiman (/users/arena/)	**accepted**	2.43	26M	CPP
3	2023-10-31 16:57:12	Fdl (/users/fadelpm2002/)	**accepted**	2.46	24M	CPP14
4	2023-12-01 17:34:24	Oleg (/users/defrager/)	**accepted**	2.78	29M	CPP14
5	2020-06-10 03:47:29	[Rampage] Blue.Mary (/users/xilinx/)	**accepted**	2.88	8.5M	CPP
6	2020-08-04 08:47:21	suhash (/users/suh_ash2008/)	**accepted**	5.26	19M	CPP14
7	2020-11-30 00:39:39	Jakub Łopuszański (/users/qbolec/)	**accepted**	7.20	7.3M	CPP14
8	2023-05-01 20:33:51	Viplov Jain (/users/viplov/)	**accepted**	8.62	11M	CPP14

Fig. 12. Statistics of all solutions submitted by all users received by Sphere Online Judge

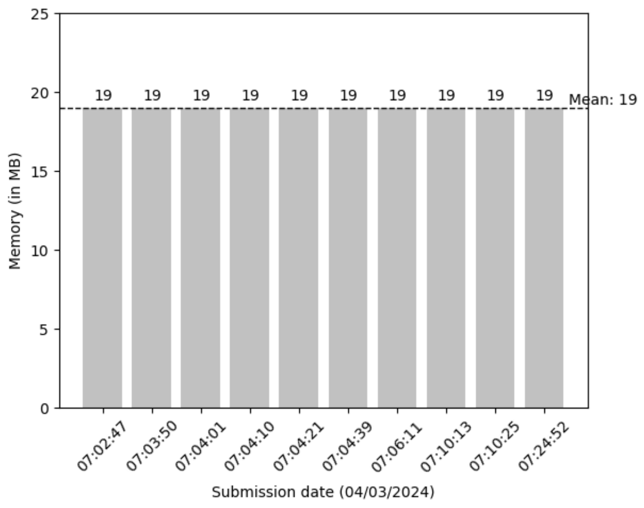


Fig. 13. External performance memory usage graph of dynamic programming method and Fast Fourier Transform by Sphere Online Judge

IV. Conclusion

In this paper, the dynamic programming method is designed and analyzed for solving combinatoric problems, which is calculating the number of hash values for a given string length. The method begins by modeling the number of hash values as a polynomial equation, followed by the use of fast multiplication on polynomials, namely the Fast Fourier Transform. A recurrence relation is derived, reducing the number of polynomial multiplication calculations to be done only in $\log_2 N_{max}$ times with N_{max} being the maximum length of the string in the test case. As a result, to calculate the number of hash values, the overall time complexity is $O(\log_2 N_{max} * M \log_2 M)$.

Experimental results for this problem have shown that the proposed approach using dynamic programming techniques and Fast Fourier Transform method can provide consistently valid answers by using efficient resources both time-wise and space-wise.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd Edition. Cambridge: The MIT Press, 2009.

[2] Flannery, B. P., Press, W. H., and Teukolsky, S. A. *Numerical Recipes in C: The Art of Scientiffic Computing*, 2nd Edition. New York: Cambridge University Press, 1992.

[3] Graham, R. L., Knuth, D. E., and Patashnik, O. *Concrete Mathematics*, 2nd Edition. Reading: Addison-Wesley, 1994.

[4] J. M. Pollard. (1971, April). *The Fast Fourier Transform in a Finite Field*. (Online). pp. 365-374 Available: https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/S0025-5718-1971-0301966-0.pdf.

[5] Moenck, Robert T. (1976). *Practical Fast Polynomial Multiplication*. (Online). pp. 136-144 Available: https://dl.acm.org/doi/pdf/10.1145/800205.806332.

[6] Pachocki, Jakub, and Jakub Radoszewski. (2013). *Where to Use and How not to Use Polynomial String Hashing*. (Online). Available: https://ioinformatics.org/journal/INFOL119.pdf.

[7] Smykalov, Vladimir. (2017). *fft: optimizations*. (Online). Available: https://neerc.ifmo.ru/trains/toulouse/2017/fft2.pdf.

[8] Weimerskirch , Andr´e, dan Christof Paa. (2006). *Generalizations of the Karatsuba Algorithm for Efficient Implementations*. (Online). Available: https://eprint.iacr.org/2006/224.pdf.

[9] Yendri, S., Soelaiman, R., Yuhana, U. L., and Yendri, S. "Dynamic Programming Approach for Solving Rectangle Partitioning Problem," *IAENG International Journal of Computer Science*, vol. 49, no.2, pp. 410-419, 2022.

[10] Yendri, S., Soelaiman, R., and Purwananto, Y., "Hybrid Algorithm to Find Minimum Expected Escape Time From a Maze," *Engineering Letters*, vol. 31, no.1, pp. 346-357, 2023

[11] Zahin, Sabit. (2021). *A collection of algorithms, data structures and templates for competitive programming*. (Online). Available: https://github.com/sgtlaugh/algovault/blob/master/code_library/fft.cpp.