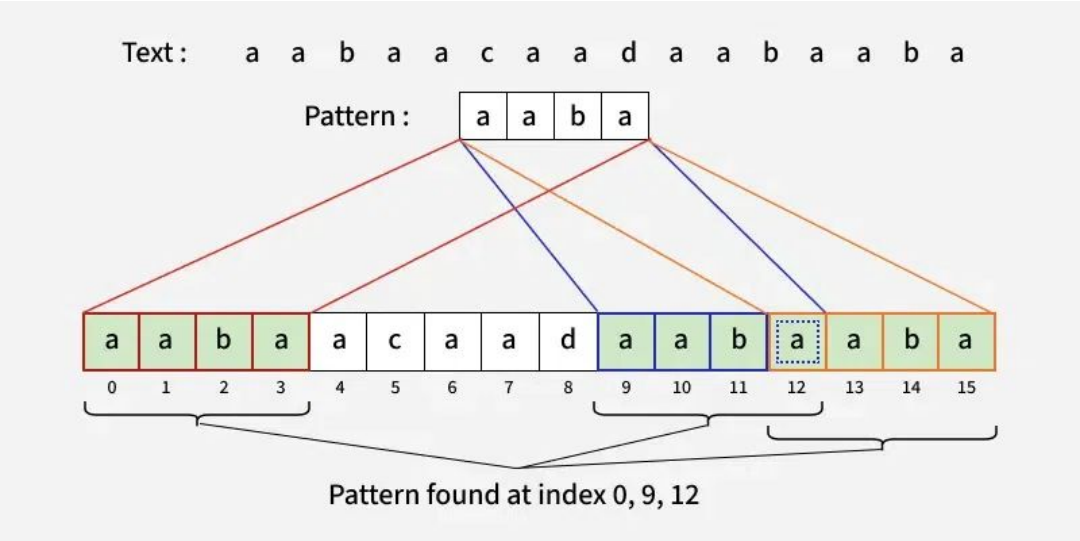
String Matching
Rabin Karp Algorithm

Problem Statement

Given two strings - a pattern s and a text t, determine if the pattern appears in the text and if it does, enumerate all its occurrences.



Approach 1: Naive

```cpp
class Solution {
public:
    //returns the list of starting indices of s where p matches with the substring of s
    vector<int> search(string s, string p) {
        vector<int> occurrences;

        int sLen = s.length();
        int pLen = p.length();

        //i denotes the starting point of the substring in s
        //slide the p[] one by one
        for (int i = 0; i <= sLen - pLen; i++) {
            int j = 0;

            //For current index i, check for pattern match
            //Iterate over the pattern j and the substring (i...pLen)
            for (j = 0; j < pLen; j++) {
                if (s[i + j] != p[j])
                    break;
            }

            //if the pattern and substring matched then j will be at pLen
            if (j == pLen)
                occurrences.push_back(i);
        }

        return occurrences;
    }
};
```

Time Complexity Analysis:

n = size of pattern

m = size of string

The outer loop runs (m-n+1) times.

The inner loop runs n times

Overall time complexity - O((m-n+1) * n)

Approach 2: Rabin Karp Algorithm

The main idea behind Rabin-Karp is to convert strings into numeric hashes so we can compare numbers instead of characters. This makes the process much faster.

To do this efficiently, we use a **rolling hash**, which allows us to compute the hash of the next substring in constant time by updating the previous hash. So instead of rechecking every character, we just slide the window and adjust the hash.

We first compute the hash of the pattern, then compare it with the hashes of all substrings of the text. If the hashes match, we do a quick character-by-character check to confirm (to avoid errors due to hash collisions).

Calculation of nextHash

currentHash = prevHash - val(skippedChar) * 2^{10}

currentHash = currentHash * 2

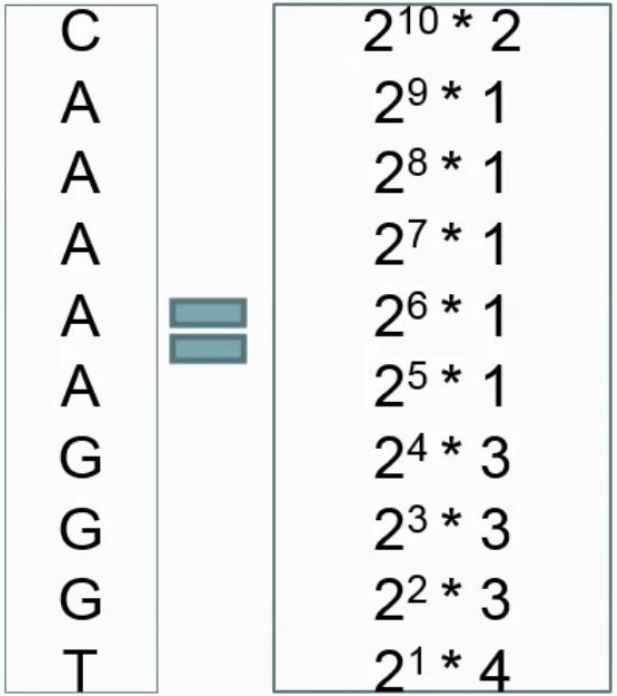
currentHash = currentHash + 2 * val(newChar)

Here once we compute the hash value of current window, we want to compute the hash value of the next window. Current window:

Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

Rolling Hash makes use of already computed hash value of previous sequence to compute hash value for next sequence.

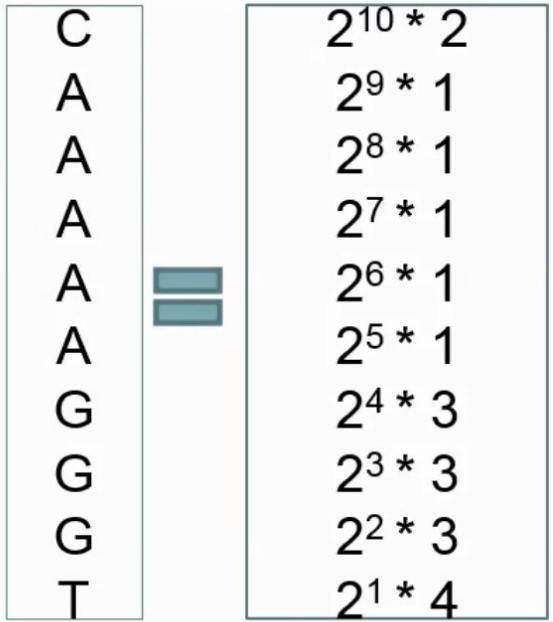


Next window for which we want to compute the hash.

Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

Rolling Hash makes use of already computed hash value of previous sequence to compute hash value for next sequence.

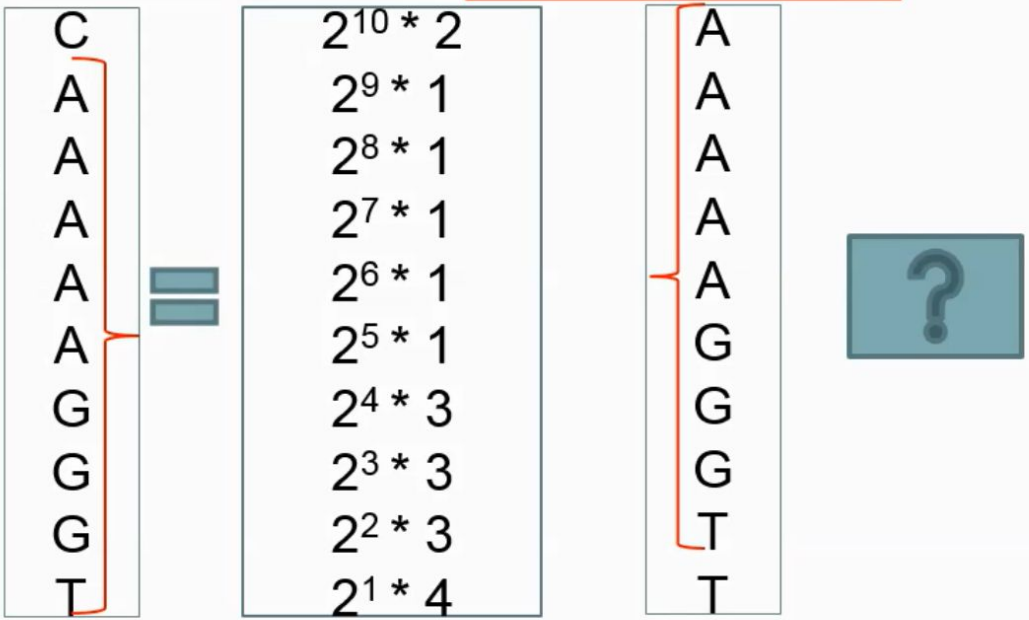


How can we make use of the previously compute hash value to compute the hash value of new sequence? Since we shifted the 10 letter pattern by 1 character, there is only difference of 1 character b/w old and new sequence. T is added and C is removed. Rest 9 letters are same.

Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

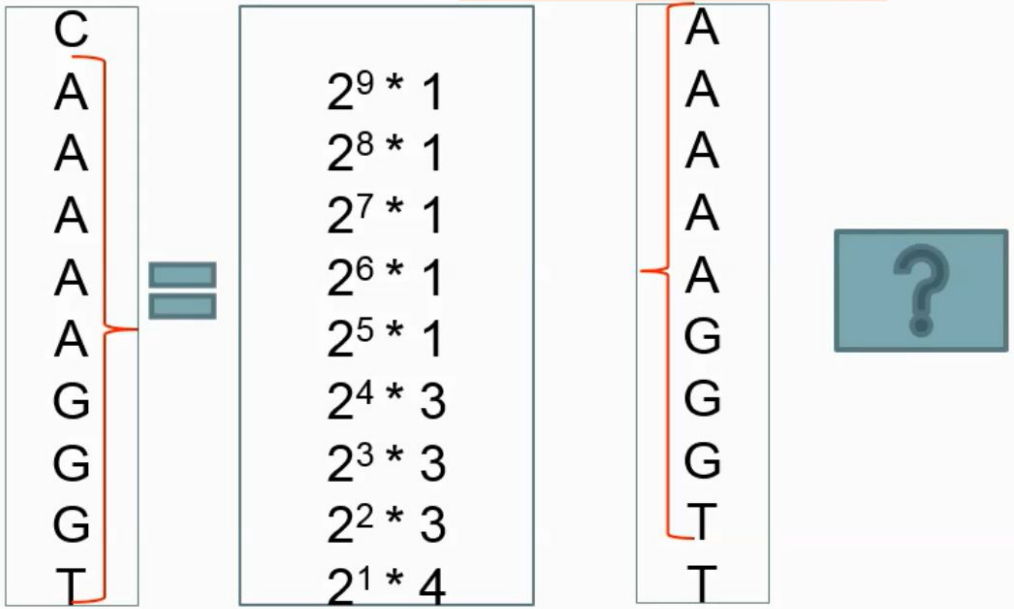
Rolling Hash makes use of already computed hash value of previous sequence to compute hash value for next sequence.



Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

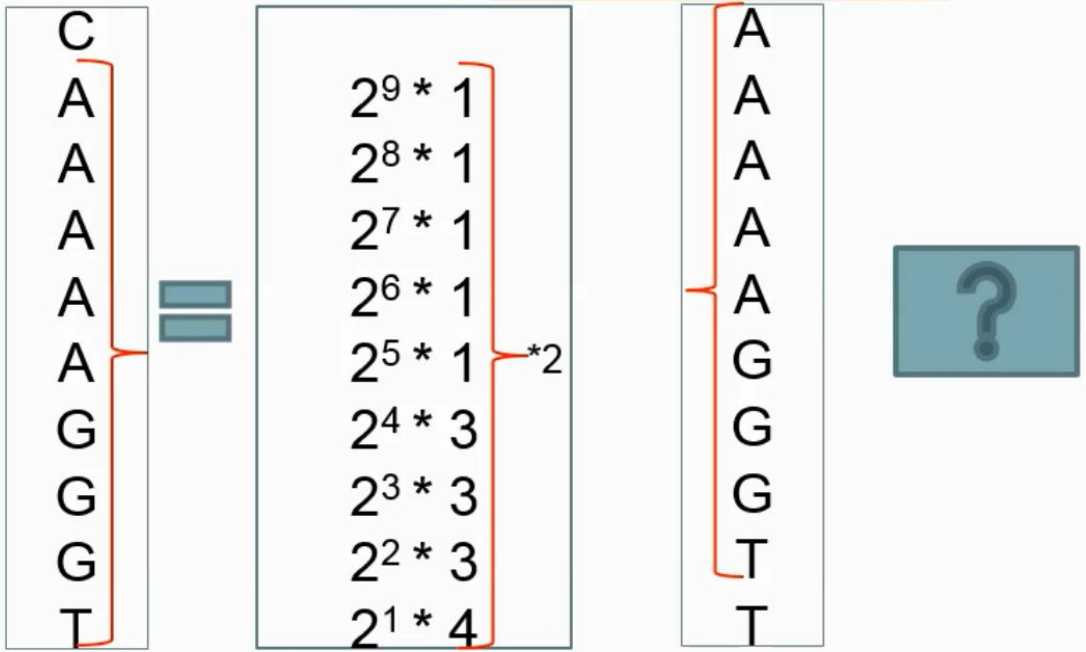
currHash = prevHash – val(skippedChar)*2^{10}



Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

currHash = prevHash – val(skippedChar)*2^{10}

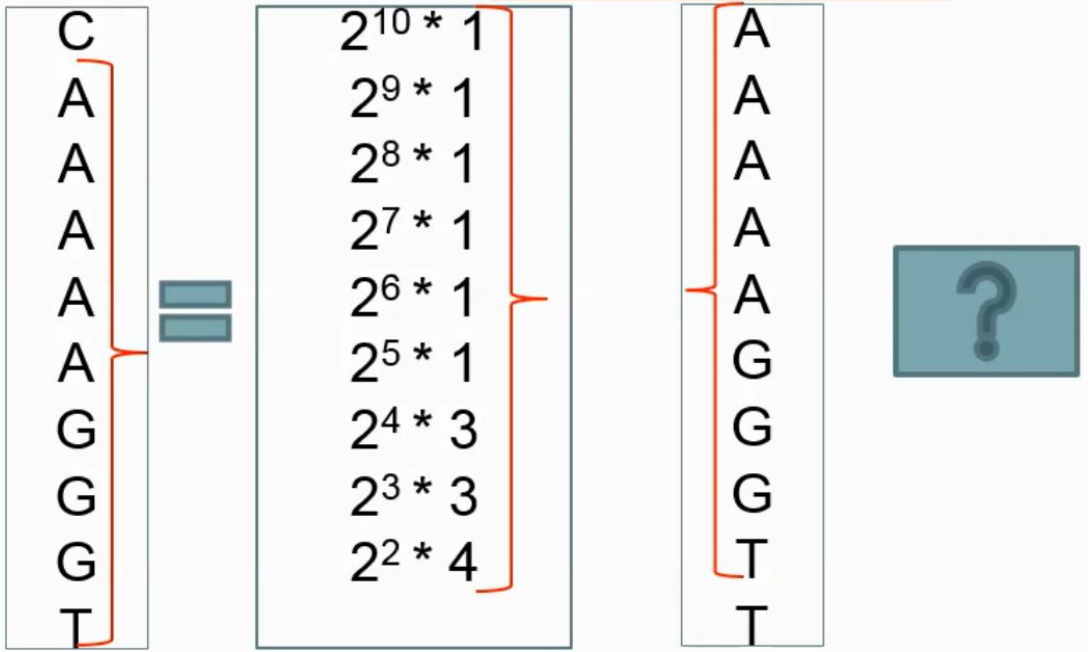


Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

currHash = prevHash – val(skippedChar)*2^{10}

currHash = currHash * 2



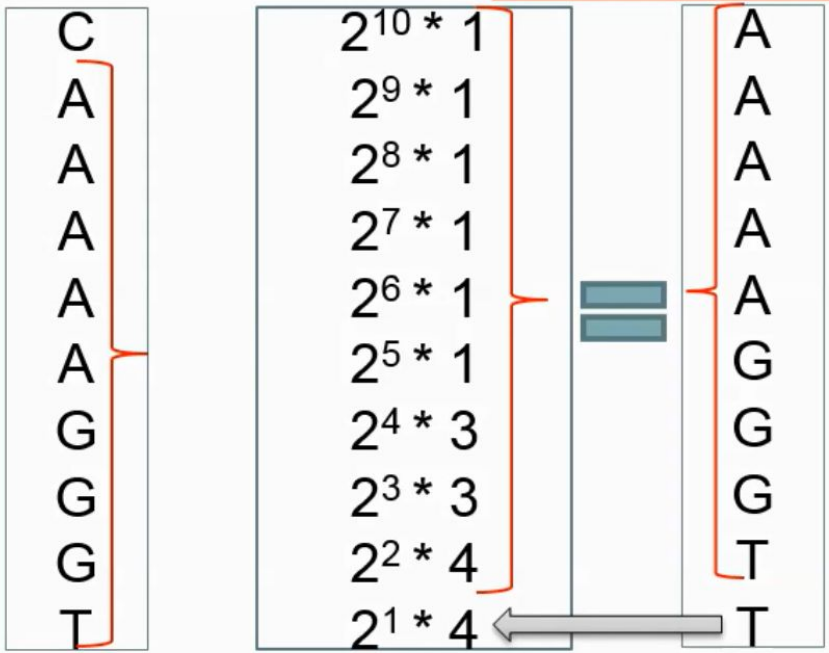
Input: “AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT“

A: 1, C:2, G:3, T:4

currHash = prevHash – val(skippedChar)*2^{10}

currHash = currHash * 2

currHash = currHash + 2*val(newChar)



The hash function

So we understood the rolling part. But how the hash function is built? We use a polynomial hash function.

We treat the string as a number in a specific base (usually a prime number roughly equal to the number of characters in the alphabet, e.g., 256 or 31).

For a string S of length M:

$$H(S) = (S[0] \cdot b^{M-1} + S[1] \cdot b^{M-2} + ... + S[M-1] \cdot b^0) \pmod q$$

- b: Base (e.g., 256 for ASCII).
- q: A large prime number (to minimize collisions).

The Rolling Update Formula

When sliding the window from index i to $i+1$:

1. **Remove** the leading character ($T[i]$).
2. **Shift** the remaining value (multiply by b).
3. **Add** the new trailing character ($T[i+M]$).

$$H_{new} = ((H_{old} - T[i] \cdot h) \cdot b + T[i+M]) \pmod q$$

Where $h = b^{M-1} \pmod q$ *(the value of the highest place).*

A simple implementation using pow() and 2 as the base

```
vector<int> search(string s, string p) {
    vector<int> occurrences;

    int sLen = s.length();
    int pLen = p.length();

    int pHash = 0;
    int sHash = 0;

    //calculate the hash of p and the first window in s
    for (int i = 0; i < pLen; i++) {
        pHash = pHash + (p[i] * pow(2, pLen-i-1));
        sHash = sHash + (s[i] * pow(2, pLen-i-1));
    }

    for (int i = pLen; i < sLen; i++) {
        //check if the previous window matches or not
        if (pHash == sHash && s.substr(i - pLen, pLen) == p)
            occurrences.push_back(i - pLen);

        //subtract the first character from the hash
        sHash = sHash - (s[i-pLen] * pow(2, pLen-1));

        //adjust the powers of each element
        sHash = sHash * 2;

        //add the new element at the end
        sHash = sHash + s[i];
    }

    //for the last window
    if (pHash == sHash)
        occurrences.push_back(sLen - pLen);

    return occurrences;
}
```

When we subtract the first character of the window from the sHash, we compute pow(2, pLen-1).

This is costly!
O(log(pLen-1)) called sLen-pLen times totalling to O((sLen-pLen) * (log(pLen-1)).

We can reduce this to O(sLen-pLen) by precomputing it.

```
vector<int> searchOptimized1(string s, string p) {
    vector<int> occurrences;

    int sLen = s.length();
    int pLen = p.length();

    int pHash = 0;
    int sHash = 0;

    //When we subtract the first character of the window from the sHash,
    //we compute pow(2, pLen-1). This is costly! O(log(pLen-1)) called sLen-pLen times
    //totalling to O((sLen-pLen) * (log(pLen-1)). We can reduce this to O(sLen-pLen)
    //by precomputing it.
    int power = 1;
    for (int i = 1; i <= pLen-1; i++) {
        power = (power * 2);
    }


    for (int i = 0; i < pLen; i++) {
        pHash = pHash + (p[i] * pow(2, pLen - i - 1));
        sHash = sHash + (s[i] * pow(2, pLen - i - 1));
    }

    for (int i = pLen; i < sLen; i++) {
        //check if the previous window matches or not
        if (pHash == sHash && s.substr(i - pLen, pLen) == p)
            occurrences.push_back(i - pLen);

        //subtract the first character from the hash
        sHash = sHash - (s[i - pLen] * power);

        //adjust the powers of each element
        sHash = sHash * 2;

        //add the new element at the end
        sHash = sHash + s[i];
    }

    //for the last window
    if (pHash == sHash)
        occurrences.push_back(sLen - pLen);

    return occurrences;
}
```

Pow() is costly operation. Instead of using pow(), we can use the **Hornor's method** to evaluate the polynomial.

Let's look at the polynomial from your comment:

$$P(x) = 10x^3 + 20x^2 + 2x + 5$$

Using the **Naive Method** (First code block), you calculate each term separately and sum them:

1. Calculate $10 \cdot x^3$
2. Calculate $20 \cdot x^2$
3. Calculate $2 \cdot x$
4. Add them all up.

Using **Horner's Method** (Second code block), you factor out x repeatedly.

$$10x^3 + 20x^2 + 2x + 5$$

Factor x from the first three terms:

$$x \cdot (10x^2 + 20x + 2) + 5$$

Factor x again from the inner terms:

$$x \cdot (x \cdot (10x + 20) + 2) + 5$$

Factor x one last time:

$$x \cdot (x \cdot (x \cdot (10) + 20) + 2) + 5$$

Notice the pattern? You start with the highest degree coefficient (`10`), multiply by x, add the next coefficient (`20`), multiply by x, add the next (`2`), and so on.

Your Horner's Formula:

$$x \cdot (x \cdot (x \cdot (10) + 20) + 2) + 5$$

Our Code Expansion (where x=2):

$$2 \cdot (2 \cdot (2 \cdot (10) + 20) + 2) + 5$$

They are identical.

```
vector<int> searchOptimized2(string s, string p) {
    vector<int> occurrences;

    int sLen = s.length();
    int pLen = p.length();

    int pHash = 0;
    int sHash = 0;

    int power = 1;
    for (int i = 1; i <= pLen-1; i++) {
        power = (power * 2);
    }

    //Instead of evaluating the polynomial in the following way,
    //lets use Hornor's method.
    //10.x^3 + 20.x^2 + 2.x + 5
    //x.(10.x^2 + 20.x + 2) + 5
    //x.(x.(10x + 20) + 2 ) + 5
    //x.(x.(x.(10) + 20) + 2 ) + 5
    /*
    for (int i = 0; i < pLen; i++) {
        pHash = pHash + (p[i] * pow(2, pLen - i - 1));
        sHash = sHash + (s[i] * pow(2, pLen - i - 1));
    }
    */
    pHash = 0, sHash = 0;

    for (int i = 0; i < pLen; i++) {
        pHash = (pHash * 2) + p[i];
        sHash = (sHash * 2) + s[i];
    }

    for (int i = pLen; i < sLen; i++) {
        //check if the previous window matches or not
        if (pHash == sHash && s.substr(i - pLen, pLen) == p)
            occurrences.push_back(i - pLen);

        //subtract the first character from the hash
        sHash = sHash - (s[i - pLen] * power);

        //adjust the powers of each element
        sHash = sHash * 2;

        //add the new element at the end
        sHash = sHash + s[i];
    }

    //for the last window
    if (pHash == sHash)
        occurrences.push_back(sLen - pLen);

    return occurrences;
}
```

2. Code Mapping

In your code, the "variable" x is **2**. The coefficients are the characters (or numbers) in your array `p[]`.

The Naive Code

C++

```
// This is slow because pow() is expensive
pHash = pHash + (p[i] * pow(2, pLen - i - 1));
```

This is doing: $p[0] \cdot 2^3 + p[1] \cdot 2^2 + \dots$

The Horner's Code

C++

```
// This is fast: only multiplication and addition
pHash = ((pHash * 2) + p[i]);
```

This directly implements the recursive bracket logic: `(PreviousValue * x) + CurrentCoefficient`.

3. Dry Run Example

Let's trace `p = [3, 1, 4]` (Just three numbers for simplicity). We want to calculate: $3 \cdot 2^2 + 1 \cdot 2^1 + 4 \cdot 2^0 = 12 + 2 + 4 = 18$.

Setup:

- `pHash = 0`
- `x = 2` (The multiplier in your loop)

Iteration 1 (`i = 0`, value `3`):

- `pHash = (0 * 2) + 3`
- `pHash` becomes **3**

Iteration 2 (`i = 1`, value `1`):

- `pHash = (3 * 2) + 1`
- `pHash` becomes **7**
- *Math check: This represents* $(3 \cdot 2) + 1$

Iteration 3 (`i = 2`, value `4`):

- `pHash = (7 * 2) + 4`
- `pHash` becomes **18**
- *Math check: This represents* $((3 \cdot 2 + 1) \cdot 2) + 4$
- *Expand it:* $(3 \cdot 2^2) + (1 \cdot 2) + 4$. **It matches!**

Reducing collisions

The modulo operator q in a hash function (like `hash % q`) determines the **range of possible output values**. You can think of q as the **number of buckets** available to store your data.

The probability of collision is directly tied to two factors controlled by the modulo: the **Size** (how large q is) and the **Quality** (whether q is prime).

1. The Size Factor (Number of Buckets)

This is the most direct relationship. If you are mapping random integers into q buckets, the probability that any two specific integers map to the same bucket is roughly $\frac{1}{q}$.

- **Small Modulo ($q = 10$):** You only have 10 slots (0-9). If you insert 11 items, you are **guaranteed** a collision (Pigeonhole Principle). Even with fewer items, the chance of overlap is high.
- **Large Modulo ($q = 1,000,000$):** You have 1 million slots. If you insert 100 items, the chance they land in the same slot is very low.

The Rule: A larger modulo decreases collision probability by spreading data out over a wider range.

```cpp
vector<int> searchOptimized3(string s, string p) {
    vector<int> occurrences;

    int sLen = s.length();
    int pLen = p.length();

    int pHash = 0;
    int sHash = 0;
    long long int MOD = 10e9 + 7;

    int power = 1;
    for (int i = 1; i <= pLen - 1; i++) {
        power = (power * 2) % MOD;
    }

    for (int i = 0; i < pLen; i++) {
        pHash = ((pHash * 2) + p[i]) % MOD;
        sHash = ((sHash * 2) + s[i]) % MOD;
    }

    for (int i = pLen; i < sLen; i++) {
        //check if the previous window matches or not
        if (pHash == sHash && s.substr(i - pLen, pLen) == p)
            occurrences.push_back(i - pLen);

        //subtract the first character from the hash
        sHash = (sHash - (s[i - pLen] * power) % MOD ) % MOD;
        if (sHash < 0) {
            sHash = sHash + MOD;
        }

        //adjust the powers of each element
        sHash = (sHash * 2) % MOD;

        //add the new element at the end
        sHash = (sHash + s[i]) % MOD;
    }

    //for the last window
    if (pHash == sHash)
        occurrences.push_back(sLen - pLen);

    return occurrences;
}
```

2. The Prime Factor (Pattern Breaking)

This is the subtle but critical part. In real-world data, inputs are rarely random. They often have patterns (e.g., memory addresses are multiples of 4 or 8; IDs might end in 0 or 5).

If your modulo q matches these patterns, collisions skyrocket.

Scenario: Input data are multiples of 10 (10, 20, 30, 40...)

Case A: Bad Modulo (Non-Prime) Let $q = 100$.

- $10\%100 = 10$
- $20\%100 = 20$
- $110\%100 = 10$ (**Collision** with 10)
- $120\%100 = 20$ (**Collision** with 20)
- **Result:** We are only using indices that end in 0 (0, 10, 20...). 90% of our buckets (1, 2, 3...) are wasted.

Case B: Good Modulo (Prime) Let $q = 101$ (A prime number).

- $10\%101 = 10$
- $20\%101 = 20$
- $110\%101 = 9$ (**No Collision** - landed in a new spot!)
- $120\%101 = 19$ (**No Collision**)
- **Result:** The prime number "scatters" the patterned data across *all* available buckets, utilizing the entire table space.

3. The "Negative Number" Trap (C++ Specific)

This is the most common bug source. Math and C++ handle negative modulo differently.

- **Math / Python:** Result always has the same sign as the divisor (usually positive).
  - $-5 \mod 3 = 1$
- **C++ / Java:** Result has the same sign as the dividend (the left number).
  - `-5 % 3 =` `-2` (This acts as a pure remainder, not a mathematical modulus).

The Fix in C++: To get the positive, mathematical result (needed for cyclic arrays or Rabin-Karp), use this formula:

C++

```
int a = -5;
int n = 3;
// Wrong: a % n -> -2
// Correct:
int result = ((a % n) + n) % n; // Result is 1
```

Time Complexity

To understand the Time Complexity of the Rabin-Karp algorithm, we must separate the **hashing efficiency** (fast) from the **verification cost** (potentially slow).

Let:

- N = Length of the Text.
- M = Length of the Pattern

2. Average (and Best) Case: $O(N + M)$

In the average scenario, the hash function distributes values well.

1. **Preprocessing:** Takes $O(M)$.
2. **Rolling Hash Updates:** We perform N updates. Each update is simple math (multiply, add, mod), taking $O(1)$.
3. **Hash Mismatches:** For the vast majority of windows, the `WindowHash` will **not** equal the `PatternHash`. We skip the verification loop entirely.
4. **Matches:** If the hashes *do* match, it is almost always a real string match. We verify it in $O(M)$, but since real matches are rare, this doesn't add up to much.

Total Time: $O(M) + O(N) \approx O(N + M)$

3. Worst Case: $O(N \cdot M)$

The worst case occurs when we encounter a **Spurious Hit** (Hash Collision) at every single step. This forces the algorithm to run the slow character verification loop N times.

When does this happen?

1. **Bad Hash Function:** If you use a weak hash (e.g., `sum(characters)`), many different strings produce the same hash.
   - Hash("AD") = $65 + 68 = 133$
   - Hash("BC") = $66 + 67 = 133$
   - *Result:* You verify characters constantly even when strings don't match.
2. **Pathological Input:** Even with a good hash, specific inputs can trigger worst-case behavior.
   - **Text:** `"AAAAAAA..."`
   - **Pattern:** `"AAAA"`
   - Here, the hash matches at *every* window. The algorithm must check M characters for every one of the N positions.

Total Time: $N \text{ windows} \times M \text{ comparisons/window} = O(N \cdot M)$

References

https://www.youtube.com/watch?v=BfUejqd07yo

https://www.youtube.com/watch?v=97_vofsFauU

https://www.youtube.com/watch?v=Sj2seJl9F_4

https://www.youtube.com/watch?v=Vg8c9uTXDeY

https://www.youtube.com/watch?v=S-LXeuHGF98