**How our Rolling Hash Multiple Patterns Matching Program Works**

**Ryun Kim and Angela Yang**

1. **Why use Rolling Hash for string match?**

In the Brute Force algorithm, at each position, the substring in the test window is compared with pattern, character by character. It is very slow and the BigO is O(n x m ).

Abce**abcde**dgcggggedtsststst ←- string

**hdgk ←** pattern

**????**

Using the hash code of the substring and pattern can compare them in one step

2345 ( hashCode of **abcd** )

Abce**abcde**dgcggggedtsststst ←- string

**wxyz ←** pattern

4568 (hashCode **of wxyz**)

2345 != 4568 so it is not a match which means it needs to shift to the right by one character to continue the next comparison.

**This hashCode comparison avoids the character by character comparison.**

Since the current position hashcode comparison did not match, we need to shift to the right

Abce**abcde**ecggggedtsststst ← string

**hdgk ←** pattern

We know the hashCode for our pattern, **hdgk** ( 4568 ) so no need to calculate it again.

We still have to calculate the hashCode for **bcde** before we can compare.

However, calculation of hashCodes takes time. Is there a way to avoid calculation of hashCode for **bcde**?

Robin Karp has figured out a way.

His idea is that hashCode of **bcde** can be derived from **abcd** (our previous hashcode)

HashCode(**bcde) =** HashCode(**abcd**) – a + e

= 2345 – 1 + 5 ( assume a’s ascii is 1 and e is 5)

By simply reusing **abcd**’s hashCode and doing some simple math, we get the hash code of **bcde.**

We repeat the process as we move through the entire string to be searched. This is what “rolling the hash” means.

1. **How to implement a multiple-pattern match rolling hash**

On the internet, there are several code samples of the implementation of the Rolling Hash algorithm.

However, those implementations are “single” pattern match, not multiple patterns match.

Rolling Hash has an advantage over Boyer and KMP when it is used in multiple patterns matches.

It is inferior in single pattern match as it moves through to test a string one character at a time like the Brute Force algorithm

Wikipedia has proposed a multiple pattern match pseudo code. After careful reading of it, we have found that it only works for the patterns that all have the **same length**, which is very limited in real use.

For example:

This would work:

We love to see **movie** on the weekend.

**Work** (pattern’s length is 4)

**Sing** (pattern’s length is 4)

**Jump** (pattern’s length is 4)

But this would not work due to different length of each pattern:

We love to see **movie** on the weekend.

**Work** (pattern’s length is 4)

**Swimming** (pattern’s length is 8)

**Run** (pattern’s length is 3)

We need to know the length of the pattern so we know how many characters to include in our calculations in order to calculate the hashCode for the substring we want to test. When patterns lengths change, we don’t have a fixed length value to use.

After some experiments we have found a solution – we use the shortest length of all patterns to calculate hashCode for substring and patterns and use them as initial match. If the shortened patterns signal a match, we then compare the substring and the pattern character by character to see if it is a true match.

2345

**swim**ming

**swim** ←--- this indicates potential match

2345

in the pattern hashTable - ( 2345 , swimming)

Now we know we need to expand the substring **swim** with 4 more characters. ( That’s how many more characters that are left in the original pattern. After expansion, the substring becomes 8 characters, it results in “swimming” which matches the pattern “swimming” exactly. Now we flag this location (begining of the substring ) as a match location (for example 56 ) and store (56, swimming ) in results hashTable. The results hashtable is used to display the final search results.

Another example:

We love to go **Swimming** on the weekend.

**Work** (pattern’s length is 4)

**Swimming** (pattern’s length is 8)

**Run** (pattern’s length is 3)

The shortest pattern is “Run” which is 3 character. We use 3 as length to cut each pattern shorter.

**Wor**

**Swi**

**Run**

We then calculate the hashCode for these shortened patterns:

**Wor --> 234**

**Swi -- > 345**

**Run -- > 679**

And we store this set of hash code is a hash table along with their true patterns:

**(234, work)**

**(345, Swimming)**

**(679, Run)**

So when we run the rolling hash algorithm and we are on the character “swi”, its hashCode (345) matches the “345” in our hashTable. This signals a potential match.

345

We love to see **swimming** on the weekend.

**Wor --> 234**

**Swi -- > 345**

**Run -- > 679**

We then invoke a deep check method to find out if there is a true match.

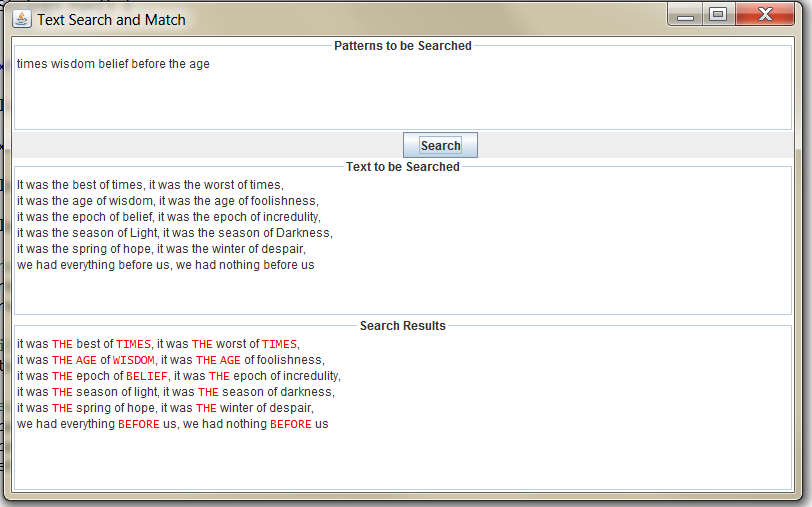
Since we know 345 in our hashTable points at “swimming” and it is 8 characters long, we need to read in 8 characters from the current position of the text to be tested. We have “swimming” which matches our pattern. Now we have a match.

This is how we have resolved the challenge of multiple patterns with various lengths.

The worst case for this approach is that when the shortest pattern is very short like 2,3 characters, we will have some false alarms which causes us to invoke the deep comparison method. That can slow the process down.

Other than that, our algorithm works well. We put together a simple GUI to demonstrate it.

You can copy and paste and paragraph of string into the “Text to be searched” middle text box and enter several words you want to test the match in the “Patterns to be searched” text area. Press the “Search” button and the results will be displayed below. The matched words are in read font and capitalized.



1. **A mathematical explanation of how the rolling hash works**
2. **How is hashCode calculated?**

If we have a substring “**abc**”, how do we calculate its hashCode?

We can use polynomial to represent hashCode:

hashCode(abc) = a\* base^**2** + b \* base ^ **1** + c \* base ^ **0** ( polynomial format )

(base = 256 or other prime number)

a, b, c is the value of each character’s ascii code. a = 1, b = 2, c = 3 for example.

Base is a prime number. We use 256 in our implementation.

^2, ^1, ^0 are the power of base. The highest power if the length of substring – 1

Since length of abc is 3, so the highest power is ^2,

Since ^ 0 is 1, so we can simplify the formula a bit :

a \* base^2 + b \* base ^ 1 + c \* base

So how to get the hashCode for “bcd” from “abc”?

It turns out we can do: hashCode(bcd) = hashCode(abc) – a + d.

That means we can get hashCode of bcd by subtracting leading character’s ascii value and adding new character’s asscii value from abc’s hashCode. This avoids calculating hashCode of bcd directly and reuse the hardcode of abc.

1. **Proof:**

hashCode for “bcd” would be :

b \* base ^2 + c \* base ^ 1 + d

we know hashCode of “abc” is :

hashCode(abc) = a \* base^2 + b \* base ^ 1 + c

we multiple base on both side of equation:

hashCode(abc) \* base = a \* base^3 + b\* base ^2 + c \* base

we add ‘d’ to both side:

hashCode(abc) \* base + d = a \* base^3 + b\* base ^2 + c \* base + d

Notice that the highlight items are hashCode for ‘bcd’, so we move rest of items to left

hashCode(abc) \* base + d – a \*base ^ 3 = b\* base ^2 + c \* base + d

to organize it a bit :

b\* base ^2 + c \* base + d = hashCode(abc) \* base + d – a \*base ^ 3

= base [( hashCode(abc) - a ] + d

The left side is the hashCode of ‘bcd’ we are looking for:

hashCode(bcd) = base[(hashCode(abc) – a ] + d

so we indeed can get hashCode of bcd by reusing the hash(Code(abc)

1. **How is hashCode of patterns are stored:**

We use a HashMap<Integer, String> to store the shortened strings’ hashCode.

Integer is the hashCode, String is the original pattern

Work

Swimming

Run

Will be shortened to shortest pattern length 3:

Wor --> 234

Swi -- > 345

Run -- > 679

They are stored as:

**(234, work)**

**(345, Swimming)**

**(679, Run)**

1. **How are the match result stored?**

We also use HashMap to store the match results:

HashMap<int, String>

Int – points to the location of the match in the string.

String – indicates which pattern is matched at that location.

1. **How are match results visualized?**

We have two ways of visualizing the matched results:

1. In the core RollingHash class, we have a method “printMatchResult”.

In this method we loop thru the string and check each position against the hashmap of Matched Results. If it founds a location that match occurs, it CAPITALIZEs next several characters to indicate a match visually.

1. In the GUI version, we use JTextPane to re-display the string to be searched. We loop thru entire string and check it against match results hashTable. For matched location, we turn the font into red to visualized the matching.