String Algorithms

(1) Naive Algo:

- Pattern = p { To find }
- Text = t { To search in }

m = length(Pattern)

n = length(text)

{ Assume n > m }

Algo:

```
for (int i=0; i < n-m; i++)
{
    int count = 0;
    for (int j=0; j < m; j++)
    {
        if (p[j] == text[i+j])
            count++;
    }
    if (count == m)
        cout << " found at index " << i << endl;
}
return false.
```

(2) ~~Optimised~~ Time Complexity

Best case	Worst case
O(n)	O(m * (n-m+1))
- When not found or other first character is not present	- When only last character is diff rest all occurs - T = AAAAAAAB, P = AAB

② Optimised Naive

→ In this if the characters are diff in the pattern to search then instead of iterating 1 by 1 we can directly jump 'M' [length of pattern] times.

Also:

P = Pattern
T = Text
M = length (P)
N = length (T)

Ex:
T = ABCE ABCD ABCEABCD
P = ABCD
M = 4
N = 12

[Assume: n > m]

→ while (i <= N-M) {

int j;

for (j = 0; j < M; j++)

if (T[i+j] != P[j])

break;

if (j == M)

cout << "pattern at index:" << i << endl;

else if (j == 0)

i = i + 1;

else { i = i + j; } // jump by j

}

* Complexity → Worst case: same.

* Rolling hash : Gives you the ability to calculate the hash value without rehashing whole string.

~~Ex: hash(1234) = hash(123~~

Page No.
Date

(3) Rabin Karp Algo. : It is better than Naive as it first calculates the hash value of the substring of length 'm' (length of pattern) & only if it matches then ~~calculat~~ compares the string.

Time - Complexity

(a) Avg. Case : O(n+m)
(b) Worst Case : O(nm) : When hash value matches.
↳ P = AAA
↳ T = AAAAAAA

Based on rolling - hash technique

↳ string = ~~abcd~~. ~~1234~~ abc

~~hash~~ =

→ We need rolling hash so that we can fastly & efficiently calculate hash.

Page No.
Date

→ Ex: String = abcd. (Text)
Pattern = abcd (pattern)

so we need to create all possible strings of length 3 (length of pattern) & calculate their hash value.

#define ll long long int

```
ll createHashValue(string str, int n)
{
    ll result = 0;
    for (int i=0; i<n; i++)
        result += (ll)str[i] * (ll)pow(prime, i);
    return result;
}
// Prime no: say: 3,

// Rolling hash code down.
ll recalculateHash(string str, int oldIndex, int newIndex,
                   ll oldHash, int patLength)
{
    ll newHash = oldHash - str[oldIndex];
    newHash = newHash / prime;
    newHash += (ll)(str[newIndex]) * (ll)(pow(
                prime, patLength-1));
    return newHash;
}
```

O(m) complexity

O(1) complexity

ROLLING HASH

In rolling hash we simply ~~[illegible]~~

Example: patt length = 3, string = 23456

hash value for first window '234' = 234.

value of next window '345' = (234 − 2×100)*10 + 5

→ The algo we have used works as.

string = abcd , prime = 5, length = 3

hash (abc) = 97 * 5^0 + 98 * 5^1 + 99 * 5^2

↓

hash (bcd) = (hash (abc) − 97 * 5^0) / 5 + 100 * 5^2

① fixed; ASCII of d; ② pattern length − 1; 5 → prime

EXPLANATION

```
bool checkEqual (string text, string pattern,
                 int s1, int e1, int s2, int e2)
{
    if (e1-s1 != e2-s2)
        return false;          // length mis-match

    while (s1 <= e1 && s2 <= e2) {
        if (text[s1] != patt[s2])
            return false;

        s1 ++;
        s2 ++;
    }

    return true;
}
```

Actual string comparison

Date

// DRIVER CODE

```
int main ()
{
    string str = "abgbcabdab";
    string patt = "abd";

    ll pathash = createHashValue( patt, patt.length());
    ll strhash = createHashValue( str, patt.length());

    for (int i=0; i<= str.length() - patt.length(); i++)
    {
        if (pathhash == strhash &&
            checkEqual(str, patt, i, i + patt.length()-1,
                       0, patt.length()-1))
            cout << "Match at : " << i << endl;
            return 0;

        // only for run-time checks on boundaries
        of length
        if (i < str.length() - patt.length())
        {
            strhash = recalculateHash(str, i, i+patt.length()
                      strhash, patt.length());
        }
    }
    return 0;
}
```

Page No.
Date

③ Knuth Morris Pratt (KMP) algo.

→ Worst case time complexity : O(n) [O(m+n)] (for creating Reset table from pattern + searching)

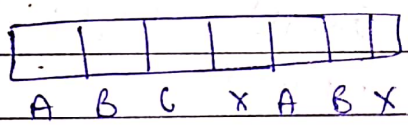
→ KMP algo acts smart whenever we detect a mismatch i.e. when a mismatch occurs it knows already some of the characters in text window & it tries to skip them.

→ It tries to iterate over & retry on some character that were already matched; for that we use a Reset table.

* In the reset table we will store the value of length of longest suffix present till now which is also present as a prefix in path.

Ex: T - ABAABCXABCXABXA

P - ABCXABX



— Reset table

CRUX

→ In this what we basically do is that till any point that we have matched in the string (Pattern) does there exist any suffix which is also a prefix, if yes then we start to compare from character after the prefix match.

* It helps us do reduce the total number of comparisons by considering re-occuring sub-patterns.

NOTE: Creating the Reset table is the imp part of KMP as it only helps do reduce the complexity.

Algo - Reset Table creation.

```
void createResetTable(int *arr, int M, char *pat)
// length of "arr" is M i.e. length of pattern.
{
    int i=1, j=0;    // Initialise
    arr[0] = 0;      // Always, bcoz nothing before it.
    while (i < M)
    {
        if (pat[i] == pat[j])
        {   arr[i] = ++j;
            i++;
        }
        else // When the characters don't match
        {
            // When j is still equal to 0.
            if (j == 0)
            {
                arr[i] = 0;
                i++;
            }
```