**BIT WISE OPERTIONS:**  
// C++ program to check if n is sparse or not

// Return true if n is sparse, else false

bool checkSparse(int n)

{

// n is not sparse if there is set

// in AND of n and n/2

if (n & (n >> 1))

return false;

return true;

}

**Knuth–Morris–Pratt For Pattern Matching:**

void computeLPSArray(char\* pat, int M, int\* lps);

// Prints occurrences of pat[] in txt[]

void KMPSearch(char\* pat, char\* txt){

int M = strlen(pat);

int N = strlen(txt);

// create lps[] that will hold the longest prefix suffix

// values for pattern

int lps[M];

// Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lp

int i = 0; // index for txt[]

int j = 0; // index for pat[]

while ((N - i) >= (M - j)) {

if (pat[j] == txt[i]) {

j++;

i++;

}

if (j == M) {

printf("Found pattern at index %d ", i - j);

j = lps[j - 1];

}

// mismatch after j matches

else if (i < N && pat[j] != txt[i]) {

// Do not match lps[0..lps[j-1]] characters,

// they will match anyway

if (j != 0)

j = lps[j - 1];

else

i = i + 1;

}

}

}

// Fills lps[] for given pattern pat[0..M-1]

void computeLPSArray(char\* pat, int M, int\* lps){

// length of the previous longest prefix suffix

int len = 0;

lps[0] = 0; // lps[0] is always

// the loop calculates lps[i] for i = 1 to M-1

int i = 1;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

}

else // (pat[i] != pat[len]){

// This is tricky. Consider the example.

// AAACAAAA and i = 7. The idea is similar

// to search step.

if (len != 0) {

len = lps[len - 1];

// Also, note that we do not increment

// i here

}

else // if (len == 0){

lps[i] = 0;

i++;

}}}}

**Boyer Moore Algorithm for Pattern Searching**

# define NO\_OF\_CHARS 256

void badCharHeuristic( string str, int size, int badchar[NO\_OF\_CHARS]){

int i;

// Initializing all occurrences as -1

for (i = 0; i < NO\_OF\_CHARS; i++)

badchar[i] = -1;

// Fill the actual value of last occurrence // of a character

for (i = 0; i < size; i++)

badchar[(int) str[i]] = i;

}

void search( string txt, string pat){

int m = pat.size();

int n = txt.size();

int badchar[NO\_OF\_CHARS];

/\* Fill the bad character array by calling the preprocessing function badCharHeuristic() for given pattern \*/

badCharHeuristic(pat, m, badchar)

int s = 0;

while(s <= (n - m))

{

int j = m - 1;

/\* Keep reducing index j of pattern while characters of pattern and text are matching at this shift s \*/

while(j >= 0 && pat[j] == txt[s + j])

j--;

/\* If the pattern is present at current shift, then index j will become -1 after the above loop \*/

if (j < 0){

cout << "pattern occurs at shift = " << s << endl;

s += (s + m < n)? m-badchar[txt[s + m]] : 1;

}

else

s += max(1, j - badchar[txt[s + j]]);

}}

**Lexicographically Next Permutation of given String**

lexicographical\_compare(string1, string1+5, string2 , string2+9);

next\_permutation(arr, arr + 3) //rearrange elements in next lexicographically greater permutation

prev\_permutation(arr, arr + n)

is\_permutation ( A.begin(), A.end(), B.begin() )