

# Anagram Substring Search (Or Search for all permutations) - GeeksforGeeks

Given a text `txt[0..n-1]` and a pattern `pat[0..m-1]`, write a function `search(char pat[], char txt[])` that prints all occurrences of `pat[]` and its permutations (or anagrams) in `txt[]`. You may assume that  $n > m$ .

Expected time complexity is  $O(n)$

Examples:

```
1) Input:  txt[] = "BACDGABCD"  pat[] = "ABCD"
   Output:   Found at Index 0
             Found at Index 5
             Found at Index 6
2) Input:  txt[] = "AAABABAA"  pat[] = "AABA"
   Output:   Found at Index 0
             Found at Index 1
             Found at Index 4
```

[We strongly recommend that you click here and practice it, before moving on to the solution.](#)

This problem is slightly different from standard pattern searching problem, here we need to search for anagrams as well. Therefore, we cannot directly apply standard pattern searching algorithms like [KMP](#), [Rabin Karp](#), [Boyer Moore](#), etc.

A simple idea is to modify [Rabin Karp Algorithm](#). For example we can keep the hash value as sum of ASCII values of all characters under modulo of a big prime number. For every character of text, we can add the current character to hash value and subtract the first character of previous window. This solution looks good, but like standard Rabin Karp, the worst case time complexity of this solution is  $O(mn)$ . The worst case occurs when all hash values match and we one by one match all characters.

We can achieve  $O(n)$  time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

- 1) The first count array store frequencies of characters in pattern.
- 2) The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is  $O(1)$  as the number of elements in them are fixed (independent of pattern and text sizes). Following are steps of this algorithm.

- 1) Store counts of frequencies of pattern in first count array `countP[]`. Also store counts of frequencies of characters in first window of text in array `countTW[]`.

2) Now run a loop from  $i = M$  to  $N-1$ . Do following in loop.

.....a) If the two count arrays are identical, we found an occurrence.

.....b) Increment count of current character of text in countTW[]

.....c) Decrement count of first character in previous window in countWT[]

3) The last window is not checked by above loop, so explicitly check it.

Following is C++ implementation of above algorithm.

```
// C++ program to search all anagrams of a pattern in a
text#include<iostream>#include<cstring>#define MAX 256
using namespace std;
// This function returns true if contents of arr1[] and arr2[]// are same,
otherwise false.
bool compare(char arr1[], char arr2[])
{
    for (int i=0; i<MAX; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}// This function search for all permutations of pat[] in txt[]
void search(char *pat, char *txt)
{
    int M = strlen(pat), N = strlen(txt);
    // countP[]: Store count of all characters of pattern
    // countTW[]: Store count of current window of text
    char countP[MAX] = {0}, countTW[MAX] = {0};
    for (int i = 0; i < M; i++)
    {
        (countP[pat[i]])++;
        (countTW[txt[i]])++;
    }
    // Traverse through remaining characters of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window of text with
        // counts of pattern[]
        if (compare(countP, countTW))
            cout << "Found at Index " << (i - M) << endl;
        // Add current character to current window
        (countTW[txt[i]])++;
        // Remove the first character of previous window
        countTW[txt[i-M]]--;
```

```
}  
// Check for the last window in text  
if (compare(countP, countTW))  
    cout << "Found at Index " << (N - M) << endl;  
}/* Driver program to test above function */  
int main()  
{  
    char txt[] = "BACDGABCD";  
    char pat[] = "ABCD";  
    search(pat, txt);  
    return 0;  
}
```

Output:

```
Found at Index 0  
Found at Index 5  
Found at Index 6
```

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above