Contents

Chapter 1

Searching for Patterns | Set 1 (Naive Pattern Searching)

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[]* in *txt[]*. You may assume that $n > m$.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

Naive Pattern Searching:
Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

C

```
// C program for Naive Pattern Searching algorithm
#include<stdio.h>
#include<string.h>

void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i+j] != pat[j])
                break;

        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
           printf("Pattern found at index %d \n", i);
    }
}

/* Driver program to test above function */
int main()
{
   char txt[] = "AABAACAADAABAAABAA";
   char pat[] = "AABA";
```

```
   search(pat, txt);
   return 0;
}
```

Python

```
# Python program for Naive Pattern Searching
def search(pat, txt):
    M = len(pat)
    N = len(txt)

    # A loop to slide pat[] one by one
    for i in xrange(N-M+1):

        # For current index i, check for pattern match
        for j in xrange(M):
            if txt[i+j] != pat[j]:
                break
        if j == M-1: # if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            print "Pattern found at index " + str(i)

# Driver program to test the above function
txt = "AABAACAADAABAAABAA"
pat = "AABA"
search (pat, txt)

# This code is contributed by Bhavya Jain
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

What is the best case?
The best case occurs when the first character of the pattern is not present in text at all.

```
   txt[]  = "AABCCAADDEE"
  pat[] = "FAA"
```

The number of comparisons in best case is O(n).

What is the worst case ?
The worst case of Naive Pattern Searching occurs in following scenarios.
1) When all characters of the text and pattern are same.

```
 txt[] = "AAAAAAAAAAAAAAAAAA"
pat[] = "AAAAA".
```

2) Worst case also occurs when only the last character is different.

```
 txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"
```

Number of comparisons in worst case is O(m*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to O(n). We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/

Chapter 2

Searching for Patterns | Set 2 (KMP Algorithm)

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[]* in *txt[]*. You may assume that $n > m$.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the previous post. The worst case complexity of Naive algorithm is O(m(n-m+1)). Time complexity of KMP algorithm is O(n) in worst case.

KMP (Knuth Morris Pratt) Pattern Searching
The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.
KMP algorithm does some preprocessing over the pattern pat[] and constructs an auxiliary array lps[] of size m (same as size of pattern). Here **name lps indicates longest proper prefix which is also suffix.**. For each sub-pattern pat[0…i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
          which is also a suffix of pat[0..i].
```

Examples:
For the pattern "AABAACAABAA", lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “ABCDE”, lps[] is [0, 0, 0, 0, 0]
For the pattern “AAAAA”, lps[] is [0, 1, 2, 3, 4]
For the pattern “AAABAAA”, lps[] is [0, 1, 2, 0, 1, 2, 3]
For the pattern “AAACAAAAAC”, lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

Searching Algorithm:
Unlike the Naive algo where we slide the pattern by one, we use a value from lps[] to decide the next sliding position. Let us see how we do that. When we compare pat[j] with txt[i] and see a mismatch, we know that characters pat[0..j-1] match with txt[i-j+1…i-1], and we also know that lps[j-1] characters of pat[0…j-1] are both proper prefix and suffix which means we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. See KMPSearch() in the below code for details.

Preprocessing Algorithm:
In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

C

```
// C program for implementation of KMP pattern searching
// algorithm
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

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

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 = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

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

```
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", 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;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // length of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
```

```
        {
          if (len != 0)
          {
            // This is tricky. Consider the example
            // AAACAAAA and i = 7.
            len = lps[len-1];

            // Also, note that we do not increment i here
          }
          else // if (len == 0)
          {
            lps[i] = 0;
            i++;
          }
        }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Python

```
# Python program for KMP Algorithm
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # create lps[] that will hold the longest prefix suffix
    # values for pattern
    lps = [0]*M
    j = 0 # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps)

    i = 0 # index for txt[]
```

```
    while i < N:
        if pat[j] == txt[i]:
            i+=1
            j+=1

        if j==M:
            print "Found pattern at index " + str(i-j)
            j = lps[j-1]

        # mismatch after j matches
        elif i < N and 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+=1

def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i]==pat[len]:
            len+=1
            lps[i] = len
            i+=1
        else:
            if len!=0:
                # This is tricky. Consier the example AAACAAAA
                # and i = 7
                len = lps[len-1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i+=1

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)

# This code is contributed by Bhavya Jain
```

Output:

```
Found pattern at index 10
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

Chapter 3

Searching for Patterns | Set 3 (Rabin-Karp Algorithm)

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[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
 txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.
Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.
Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt[s+1 .. s+m]) = d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q*

hash(txt[s .. s+m-1]) : Hash value at shift *s*.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift *s*+1)
d: Number of characters in the alphabet
q: A prime number
h: d^(m-1)

```
/* Following program is a C implementation of the Rabin Karp Algorithm
   given in the CLRS book */

#include<stdio.h>
#include<string.h>
```

```
// d is the number of characters in input alphabet
#define d 256

/*  pat  -> pattern
    txt  -> text
    q    -> A prime number
*/
void search(char *pat, char *txt, int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;  // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M-1; i++)
        h = (h*d)%q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++)
    {
        p = (d*p + pat[i])%q;
        t = (d*t + txt[i])%q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++)
    {

      // Check the hash values of current window of text and pattern
     // If the hash values match then only check for characters on by one
        if ( p == t )
        {
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
        if (j == M)  // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            {
                printf("Pattern found at index %d \n", i);
            }
        }
```

```
      // Calculate hash value for next window of text: Remove leading digit,
         // add trailing digit
         if ( i < N-M )
         {
             t = (d*(t - txt[i]*h) + txt[i+M])%q;

         // We might get negative value of t, converting it to positive
             if(t < 0)
               t = (t + q);
         }
     }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEK";
    int q = 101;  // A prime number
    search(pat, txt, q);
    getchar();
    return 0;
}
```

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = "AAA" and txt[] = "AAAAAAA".

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap34.htm

http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf

http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm

Related Posts:

Searching for Patterns | Set 1 (Naive Pattern Searching)
Searching for Patterns | Set 2 (KMP Algorithm)

Source

http://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/

Chapter 4

Searching for Patterns | Set 4 (A Naive Pattern Searching Question)

Question: We have discussed Naive String matching algorithm here. Consider a situation where all characters of pattern are different. Can we modify the original Naive String Matching algorithm so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

Solution: In the original Naive String matching algorithm, we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1. Let us see how can we do this. When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts. Following is the modified code that is optimized for the special patterns.

```
#include<stdio.h>
#include<string.h>

/* A modified Naive Pettern Searching algorithn that is optimized
   for the cases when all characters of pattern are different */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i = 0;
```

```
    while(i <= N - M)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
           printf("Pattern found at index %d \n", i);
           i = i + M;
        }
        else if (j == 0)
        {
           i = i + 1;
        }
        else
        {
           i = i + j; // slide the pattern by j
        }
    }
}

/* Driver program to test above function */
int main()
{
   char *txt = "ABCEABCDABCEABCD";
   char *pat = "ABCD";
   search(pat, txt);
   getchar();
   return 0;
}
```

Output:
Pattern found at index 4
Pattern found at index 12

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/pattern-searching-set-4-a-naive-string-matching-algo-question/

Category: Strings Tags: Pattern Searching

Chapter 5

Searching for Patterns | Set 5 (Finite Automata)

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[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
 txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

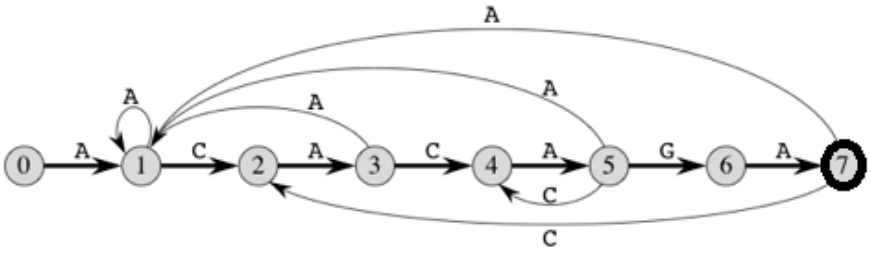
```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm

In this post, we will discuss Finite Automata (FA) based pattern searching algorithm. In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to new state. If we reach final state, then pattern is found in text. Time complexity of the search prcess is O(n).
Before we discuss FA construction, let us take a look at the following FA for pattern ACACAGA.



	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern ACACAGA.

Number of states in FA will be M+1 where M is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character x and a state k, we can get the next state by considering the string "pat[0..k-1]x" which is basically concatenation of pattern characters pat[0], pat[1] … pat[k-1] and the character x. The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of "pat[0..k-1]x". The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character 'C' in the above diagram. We need to consider the string, "pat[0..5]C" which is "ACACAC". The lenght of the longest prefix of the pattern such that the prefix is suffix of "ACACAC"is 4 ("ACAC"). So the next state (from state 5) is 4 for character 'C'.

In the following code, computeTF() constructs the FA. The time complexity of the computeTF() is O(m^3*NO_OF_CHARS) where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of "pat[0..k-1]x". There are better implementations to construct FA in O(m*NO_OF_CHARS) (Hint: we can use something like lps array construction in KMP algorithm). We have covered the better implementation in our next post on pattern searching.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character in pattern,
    // then simply increment state
    if (state < M && x == pat[state])
        return state+1;

    int ns, i;  // ns stores the result which is next state

    // ns finally contains the longest prefix which is also suffix
    // in "pat[0..state-1]c"

    // Start from the largest possible value and stop when you find
    // a prefix which is also suffix
    for (ns = state; ns > 0; ns--)
    {
        if(pat[ns-1] == x)
        {
            for(i = 0; i < ns-1; i++)
```

```
            {
                if (pat[i] != pat[state-ns+1+i])
                    break;
            }
            if (i == ns-1)
                return ns;
        }
    }

    return 0;
}

/* This function builds the TF table which represents Finite Automata for a
   given pattern  */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
           TF[state][x] = getNextState(pat, M,  state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
       state = TF[state][txt[i]];
       if (state == M)
       {
         printf ("\n patterb found at index %d", i-M+1);
       }
    }
}

// Driver program to test above function
int main()
```

```
{
   char *txt = "AABAACAADAABAAABAA";
   char *pat = "AABA";
   search(pat, txt);
   return 0;
}
```

Output:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

References:
Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

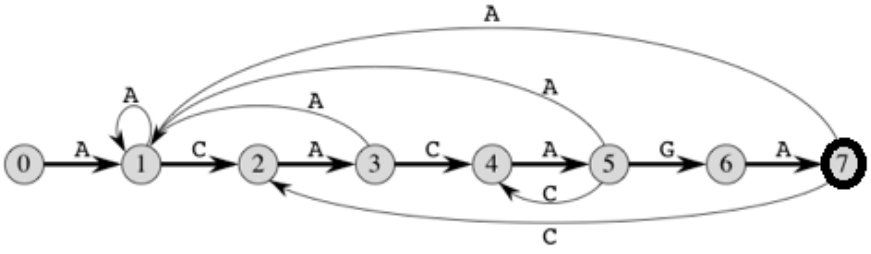
Source

http://www.geeksforgeeks.org/searching-for-patterns-set-5-finite-automata/

Chapter 6

Pattern Searching | Set 6 (Efficient Construction of Finite Automata)

In the previous post, we discussed Finite Automata based pattern searching algorithm. The FA (Finite Automata) construction method discussed in previous post takes O((mˆ3)*NO_OF_CHARS) time. FA can be constructed in O(m*NO_OF_CHARS) time. In this post, we will discuss the O(m*NO_OF_CHARS) algorithm for FA construction. The idea is similar to lps (longest prefix suffix) array construction discussed in the KMP algorithm. We use previously filled rows to fill a new row.



	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern ACACAGA.

Algorithm:
1) Fill the first row. All entries in first row are always 0 except the entry for pat[0] character. For pat[0] character, we always need to go to state 1.
2) Initialize lps as 0. lps for the first index is always 0.
3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
…..a) Copy the entries from the row at index equal to lps.
…..b) Update the entry for pat[i] character to i+1.
…..c) Update lps "lps = TF[lps][pat[i]]" where TF is the 2D array which is being constructed.

Implementation
Following is C implementation for the above algorithm.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

/* This function builds the TF table which represents Finite Automata for a
   given pattern  */
void computeTransFun(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int i, lps = 0, x;

    // Fill entries in first row
    for (x =0; x < NO_OF_CHARS; x++)
       TF[0][x] = 0;
    TF[0][pat[0]] = 1;

    // Fill entries in other rows
    for (i = 1; i<= M; i++)
    {
```

```
        // Copy values from row at index lps
        for (x = 0; x < NO_OF_CHARS; x++)
            TF[i][x] = TF[lps][x];

        // Update the entry corresponding to this character
        TF[i][pat[i]] = i + 1;

        // Update lps for next row to be filled
        if (i < M)
          lps = TF[lps][pat[i]];
    }
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTransFun(pat, M, TF);

    // process text over FA.
    int i, j=0;
    for (i = 0; i < N; i++)
    {
       j = TF[j][txt[i]];
       if (j == M)
       {
         printf ("\n pattern found at index %d", i-M+1);
       }
    }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEKS";
    search(pat, txt);
    getchar();
    return 0;
}
```

Output:

```
pattern found at index 0
pattern found at index 10
```

Time Complexity for FA construction is O(M*NO_OF_CHARS). The code for search is same as the previous post and time complexity for it is O(n).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/pattern-searching-set-5-efficient-constructtion-of-finite-automata/

Category: Strings Tags: Pattern Searching

Chapter 7

Pattern Searching | Set 7 (Boyer Moore Algorithm - Bad Character Heuristic)

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[] in txt[]. You may assume that n > m.

Examples:

```
1) Input:

  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"

Output:

Pattern found at index 10

2) Input:

  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"

Output:

   Pattern found at index 0
   Pattern found at index 9
```

```
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm

In this post, we will discuss Boyer Moore pattern searching algorithm. Like KMPand Finite Automataalgorithms, Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of following two approaches.
1) Bad Character Heuristic
2) Good Suffix Heuristic
Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

```
/* Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm */

# include <limits.h>
```

```
# include <string.h>
# include <stdio.h>

# define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize 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;
}

/* A pattern searching function that uses Bad Character Heuristic of
   Boyer Moore Algorithm */
void search( char *txt,  char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    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;  // s is shift of the pattern with respect to text
    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)
        {
            printf("\n pattern occurs at shift = %d", s);

           /* Shift the pattern so that the next character in text
               aligns with the last occurrence of it in pattern.
              The condition s+m < n is necessary for the case when
               pattern occurs at the end of text */
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;

        }

        else
            /* Shift the pattern so that the bad character in text
              aligns with the last occurrence of it in pattern. The
             max function is used to make sure that we get a positive
            shift. We may get a negative shift if the last occurrence
             of bad character in pattern is on the right side of the
                current character. */
            s += max(1, j - badchar[txt[s+j]]);
    }
}

/* Driver program to test above funtion */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}
```

Output:

```
 pattern occurs at shift = 4
```

The Bad Character Heuristic may take O(mn) time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/

Chapter 8

Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

Pattern Searching | Set 8 (Suffix Tree Introduction)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source Wiki).
A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.
Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: Wiki)

Example:

```
 Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ---------------->     1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

Naive method to build Suffix Array
A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
```

```
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return  suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
 Following is suffix array for banana
5 3 1 0 4 2
```

The time complexity of above method to build suffix array is $O(n^2Logn)$ if we consider a O(nLogn) algorithm used for sorting. The sorting step itself takes $O(n^2Logn)$ time as every comparison is a comparison of two strings and the comparison takes O(n) time.
There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array
To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, Binary Search can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```
// This code only contains search() and main. To make it a complete running
// above code or see http://ideone.com/1Io9eN

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
   int m = strlen(pat);  // get length of pattern, needed for strncmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1;  // Initilize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabtically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana";  // text
    char pat[] = "nan";   // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);
```

```
    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

```
Pattern found at index 2
```

The time complexity of the above search function is O(mLogn). There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a O(m) suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array
Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

See this for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a O(nLogn) algorithm for Suffix Array construction here. We will soon be discussing more efficient suffix array algorithms.

References:
http://www.stanford.edu/class/cs97si/suffix-array.pdf
http://en.wikipedia.org/wiki/Suffix_array

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-array-set-1-introduction/

Chapter 9

Anagram Substring Search (Or Search for all permutations)

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[] = "BACDGABCDA"  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
```

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[] = "BACDGABCDA";
    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

Source

http://www.geeksforgeeks.org/anagram-substring-search-search-permutations/

Chapter 10

Pattern Searching using a Trie of all Suffixes

Problem Statement: 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[] in txt[]. You may assume that n > m.

As discussed in the previous post, we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: KMP Algorithm, Rabin Karp Algorithm, Finite Automata, Boyer Moore Algorithm.

2) Preoprocess Text: Suffix Tree

The best possible time complexity achieved by first (preprocssing pattern) is O(n) and by second (preprocessing text) is O(m) where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in O(m) time and it is preferred when text doesn't doesn't change very frequently and there are many search queries. We have discussed Suffix Tree (A compressed Trie of all suffixes of Text).

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a Standard Trie of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a simple Trie instead of compressed Trie.

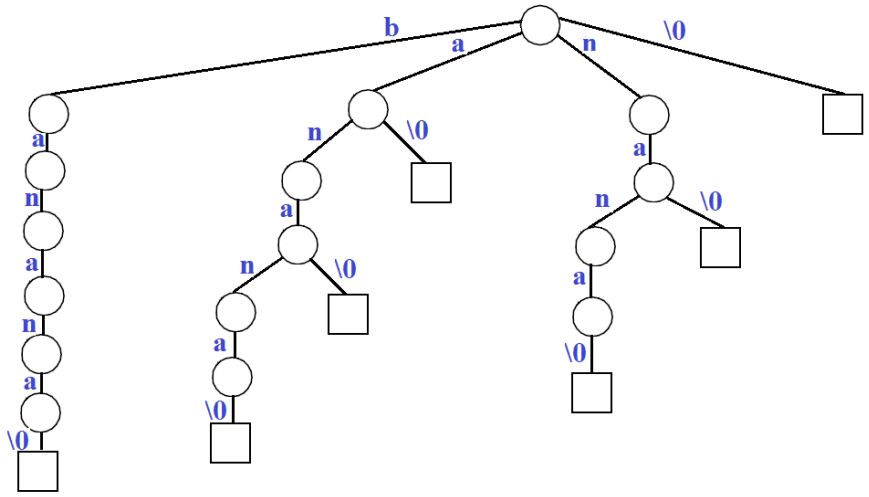
As discussed in Suffix Tree post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in O(m) time where m is pattern length.

Building a Trie of Suffixes
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a trie.

Let us consider an example text "banana\0 where '\0 is string termination character. Following are all suffixes of "banana\0

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?
Following are steps to search a pattern in the built Trie.
1) Starting from the first character of the pattern and root of the Trie, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.
…..**b)** If there is no edge, print "pattern doesn't exist in text" and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print print all indexes where pattern is

present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

```
// A simple C++ implementation of substring search using trie of suffixes
#include<iostream>
#include<list>
#define MAX_CHAR 256
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTrieNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
          children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node.The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
```

```
    SuffixTrieNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }

    // Function to searches a pattern in this suffix trie.
    void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTrieNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_front(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTrieNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;
```

```
    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)*/
void SuffixTrie::search(string pat)
{
    // Let us call recursive search function for root of Trie.
   // We get a list of all indexes (where pat is present in text) in
    // variable 'result'
    list<int> *result = root.search(pat);

    // Check if the list of indexes is empty or not
    if (result == NULL)
        cout << "Pattern not found" << endl;
    else
    {
       list<int>::iterator i;
       int patLen = pat.length();
       for (i = result->begin(); i != result->end(); ++i)
        cout << "Pattern found at position " << *i - patLen<< endl;
    }
}

// driver program to test above functions
int main()
{
    // Let us build a suffix trie for text "geeksforgeeks.org"
    string txt = "geeksforgeeks.org";
    SuffixTrie S(txt);

    cout << "Search for 'ee'" << endl;
    S.search("ee");

    cout << "\nSearch for 'geek'" << endl;
    S.search("geek");

    cout << "\nSearch for 'quiz'" << endl;
    S.search("quiz");

    cout << "\nSearch for 'forgeeks'" << endl;
```

```
    S.search("forgeeks");

    return 0;
}
```

Output:

```
 Search for 'ee'
Pattern found at position 9
Pattern found at position 1

Search for 'geek'
Pattern found at position 8
Pattern found at position 0

Search for 'quiz'
Pattern not found

Search for 'forgeeks'
Pattern found at position 5
```

Time Complexity of the above search function is O(m+k) where m is length of the pattern and k is the number of occurrences of pattern in text.

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

Source

http://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/

Chapter 11

Longest Even Length Substring such that Sum of First and Second Half is same

Given a string 'str' of digits, find length of the longest substring of 'str', such that the length of the substring is 2k digits and sum of left k digits is equal to the sum of right k digits.

Examples:

```
Input: str = "123123"
Output: 6
The complete string is of even length and sum of first and second
half digits is same

Input: str = "1538023"
Output: 4
The longest substring with same first and second half sum is "5380"
```

Simple Solution [$O(n^3)$]
A Simple Solution is to check every substring of even length. The following is C based implementation of simple approach.

```
// A simple C based program to find length of longest  even length
// substring with same sum of digits in left and right
#include<stdio.h>
#include<string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen =0;  // Initialize result

    // Choose starting point of every substring
    for (int i=0; i<n; i++)
    {
        // Choose ending point of even length substring
        for (int j =i+1; j<n; j += 2)
        {
            int length = j-i+1;//Find length of current substr

            // Calculate left & right sums for current substr
            int leftsum = 0, rightsum =0;
            for (int k =0; k<length/2; k++)
            {
                leftsum  += (str[i+k]-'0');
                rightsum += (str[i+k+length/2]-'0');
            }

            // Update result if needed
            if (leftsum == rightsum && maxlen < length)
                    maxlen = length;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "1538023";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}
```

Output:

```
 Length of the substring is 4
```

Dynamic Programming [$O(n^2)$ and $O(n^2)$ extra space]
The above solution can be optimized to work in $O(n^2)$ using **Dynamic Programming**. The idea is to build a 2D table that stores sums of substrings. The following is C based implementation of Dynamic Programming approach.

```
// A C based program that uses Dynamic Programming to find length of the
// longest even substring with same sum of digits in left and right half
#include <stdio.h>
#include <string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // A 2D table where sum[i][j] stores sum of digits
    // from str[i] to str[j].  Only filled entries are
    // the entries where j >= i
    int sum[n][n];

    // Fill the diagonal values for sunstrings of length 1
    for (int i =0; i<n; i++)
        sum[i][i] = str[i]-'0';

    // Fill entries for substrings of length 2 to n
    for (int len=2; len<=n; len++)
    {
        // Pick i and j for current substring
        for (int i=0; i<n-len+1; i++)
        {
            int j = i+len-1;
            int k = len/2;

            // Calculate value of sum[i][j]
            sum[i][j] = sum[i][j-k] + sum[j-k+1][j];

            // Update result if 'len' is even, left and right
            // sums are same and len is more than maxlen
            if (len%2 == 0 && sum[i][j-k] == sum[(j-k+1)][j]
                           && len > maxlen)
                 maxlen = len;
        }
    }
    return maxlen;
}
```

```
// Driver program to test above function
int main(void)
{
    char str[] = "153803";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}
```

Output:

```
 Length of the substring is 4
```

Time complexity of the above solution is $O(n^2)$, but it requires $O(n^2)$ extra space.

[A $O(n^2)$ and O(n) extra space solution]
The idea is to use a single dimensional array to store cumulative sum.

```
// A O(n^2) time and O(n) extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
   int sum[n+1]; // To store cumulative sum from first digit to nth digit
    sum[0] = 0;

    /* Store cumulative sum of digits from first to last digit */
    for (int i = 1; i <= n; i++)
     sum[i] = (sum[i-1] + str[i-1]  - '0'); /* convert chars to int */

    int ans = 0; // initialize result

    /* consider all even length substrings one by one */
    for (int len = 2; len <= n; len += 2)
    {
        for (int i = 0; i <= n-len; i++)
        {
            int j = i + len - 1;

          /* Sum of first and second half is same than update ans */
           if (sum[i+len/2] - sum[i] == sum[i+len] - sum[i+len/2])
                ans = max(ans, len);
```

```
        }
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
   cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}
```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

[A $O(n^2)$ time and O(1) extra space solution]
The idea is to consider all possible mid points (of even length substrings) and keep expanding on both sides to get and update optimal length as the sum of two sides become equal.

Below is C++ implementation of the above idea.

```
// A O(n^2) time and O(1) extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int ans = 0; // Initialize result

    // Consider all possible midpoints one by one
    for (int i = 0; i <= n-2; i++)
    {
        /* For current midpoint 'i', keep expanding substring on
           both sides, if sum of both sides becomes equal update
           ans */
        int l = i, r = i + 1;

        /* initialize left and right sum */
        int lsum = 0, rsum = 0;
```

```
        /* move on both sides till indexes go out of bounds */
        while (r < n && l >= 0)
        {
            lsum += str[l] - '0';
            rsum += str[r] - '0';
            if (lsum == rsum)
                ans = max(ans, r-l+1);
            l--;
            r++;
        }
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
   cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}
```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

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

Source

http://www.geeksforgeeks.org/longest-even-length-substring-sum-first-second-half/

Category: Strings Tags: Dynamic Programming

Chapter 12

Print all possible strings that can be made by placing spaces

Given a string you need to print all possible strings that can be made by placing spaces (zero or one) in between them.

```
Input:  str[] = "ABC"
Output: ABC
        AB C
        A BC
        A B C
```

Source: Amazon Interview Experience | Set 158, Round 1 ,Q 1.

We strongly recommend to minimize your browser and try this yourself first.

The idea is to use recursion and create a buffer that one by one contains all output strings having spaces. We keep updating buffer in every recursive call. If the length of given string is 'n' our updated string can have maximum length of n + (n-1) i.e. 2n-1. So we create buffer size of 2n (one extra character for string termination).
We leave 1st character as it is, starting from the 2nd character, we can either fill a space or a character. Thus one can write a recursive function like below.

```
// C++ program to print permutations of a given string with spaces.
```

```
#include <iostream>
#include <cstring>
using namespace std;

/* Function recursively prints the strings having space pattern.
   i and j are indices in 'str[]' and 'buff[]' respectively */
void printPatternUtil(char str[], char buff[], int i, int j, int n)
{
    if (i==n)
    {
        buff[j] = '\0';
        cout << buff << endl;
        return;
    }

    // Either put the character
    buff[j] = str[i];
    printPatternUtil(str, buff, i+1, j+1, n);

    // Or put a space followed by next character
    buff[j] = ' ';
    buff[j+1] = str[i];

    printPatternUtil(str, buff, i+1, j+2, n);
}

// This function creates buf[] to store individual output string and uses
// printPatternUtil() to print all permutations.
void printPattern(char *str)
{
    int n = strlen(str);

    // Buffer to hold the string containing spaces
    char buf[2*n]; // 2n-1 characters and 1 string terminator

    // Copy the first character as it is, since it will be always
    // at first position
    buf[0] = str[0];

    printPatternUtil(str, buf, 1, 1, n);
}

// Driver program to test above functions
int main()
{
    char *str = "ABCDE";
```

```
    printPattern(str);
    return 0;
}
```

Output:

```
ABCD
ABC D
AB CD
AB C D
A BCD
A BC D
A B CD
A B C D
```

Time Complexity: Since number of Gaps are n-1, there are total 2^(n-1) patters each having length ranging from n to 2n-1. Thus overall complexity would be O(n*(2^n)).

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

Source

http://www.geeksforgeeks.org/print-possible-strings-can-made-placing-spaces/

Category: Strings

Chapter 13

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 1

Given a string, find the longest substring which is palindrome.

- if the given string is "forgeeksskeegfor", the output should be "geeksskeeg"
- if the given string is "abaaba", the output should be "abaaba"
- if the given string is "abababa", the output should be "abababa"
- if the given string is "abcbabcbabcba", the output should be "abcbabcba"

We have already discussed Naïve [$O(n^3)$] and quadratic [$O(n^2)$] approaches at Set 1 and Set 2.
In this article, we will talk about Manacher's algorithm which finds Longest Palindromic Substring in linear time.
One way (Set 2) to find a palindrome is to start from the center of the string and compare characters in both directions one by one. If corresponding characters on both sides (left and right of the center) match, then they will make a palindrome.
Let's consider string "abababa".
Here center of the string is 4th character (with index 3) b. If we match characters in left and right of the center, all characters match and so string "abababa" is a palindrome.

Here center position is not only the actual string character position but it could be the position between two characters also.
Consider string “abaaba” of even length. This string is palindrome around the position between 3rd and 4th characters a and a respectively.

To find Longest Palindromic Substring of a string of length N, one way is take each possible 2*N + 1 centers (the N character positions, N-1 between two character positions and 2 positions at left and right ends), do the character match in both left and right directions at each 2*N+ 1 centers and keep track of LPS. This approach takes O(N^2) time and that's what we are doing in Set 2.

Let's consider two strings “abababa” and “abaaba” as shown below:

In these two strings, left and right side of the center positions (position 7 in 1st string and position 6 in 2nd string) are symmetric. Why? Because the whole string is palindrome around the center position.
If we need to calculate Longest Palindromic Substring at each 2*N+1 positions from left to right, then palindrome's symmetric property could help to avoid some of the unnecessary computations (i.e. character comparison). If there is a palindrome of some length L cantered at any position P, then we may not need to compare all characters in left and right side at position P+1. We already calculated LPS at positions before P and they can help to avoid some of the comparisons after position P.
This use of information from previous positions at a later point of time makes the Manacher's algorithm linear. In Set 2, there is no reuse of previous information and so that is quadratic.

Manacher's algorithm is probably considered complex to understand, so here we will discuss it in as detailed way as we can. Some of it's portions may require multiple reading to understand it properly.

Let's look at string “abababa”. In 3rd figure above, 15 center positions are shown. We need to calculate length of longest palindromic string at each of these positions.

- At position 0, there is no LPS at all (no character on left side to compare), so length of LPS will be 0.
- At position 1, LPS is a, so length of LPS will be 1.

- At position 2, there is no LPS at all (left and right characters a and b don't match), so length of LPS will be 0.
- At position 3, LPS is aba, so length of LPS will be 3.
- At position 4, there is no LPS at all (left and right characters b and a don't match), so length of LPS will be 0.
- At position 5, LPS is ababa, so length of LPS will be 5.

..... and so on

We store all these palindromic lengths in an array, say L. Then string S and LPS Length L look like below:

Similarly, LPS Length L of string "abaaba" will look like:

In LPS Array L:

- LPS length value at odd positions (the actual character positions) will be odd and greater than or equal to 1 (1 will come from the center character itself if nothing else matches in left and right side of it)
- LPS length value at even positions (the positions between two characters, extreme left and right positions) will be even and greater than or equal to 0 (0 will come when there is no match in left and right side)

Position and index for the string are two different things here. For a given string S of length N, indexes will be from 0 to N-1 (total N indexes) and positions will be from 0 to 2*N (total 2*N+1 positions).

LPS length value can be interpreted in two ways, one in terms of index and second in terms of position. LPS value d at position I (L[i] = d) tells that:

- Substring from position i-d to i+d is a palindrome of length d (in terms of position)
- Substring from index (i-d)/2 to [(i+d)/2 – 1] is a palindrome of length d (in terms of index)

e.g. in string "abaaba", L[3] = 3 means substring from position 0 (3-3) to 6 (3+3) is a palindrome which is "aba" of length 3, it also means that substring from index 0 [(3-3)/2] to 2 [(3+3)/2 – 1] is a palindrome which is "aba" of length 3.

Now the main task is to compute LPS array efficiently. Once this array is computed, LPS of string S will be centered at position with maximum LPS length value.
We will see it in Part 2.

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

Source

http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

Chapter 14

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 2

In Manacher's Algorithm – Part 1, we gone through some of the basics and LPS length array.
Here we will see how to calculate LPS length array efficiently.

To calculate LPS array efficiently, we need to understand how LPS length for any position may relate to LPS length value of any previous already calculated position.
For string "abaaba", we see following:

If we look around position 3:

- LPS length value at position 2 and position 4 are same
- LPS length value at position 1 and position 5 are same

We calculate LPS length values from left to right starting from position 0, so we can see if we already know LPS length values at positions 1, 2 and 3 already then we may not need to calculate LPS length at positions 4 and 5 because they are equal to LPS length values at corresponding positions on left side of position 3.

If we look around position 6:

- LPS length value at position 5 and position 7 are same
- LPS length value at position 4 and position 8 are same

…… and so on.
If we already know LPS length values at positions 1, 2, 3, 4, 5 and 6 already then we may not need to calculate LPS length at positions 7, 8, 9, 10 and 11 because they are equal to LPS length values at corresponding positions on left side of position 6.
For string "abababa", we see following:

If we already know LPS length values at positions 1, 2, 3, 4, 5, 6 and 7 already then we may not need to calculate LPS length at positions 8, 9, 10, 11, 12 and 13 because they are equal to LPS length values at corresponding positions on left side of position 7.

Can you see why LPS length values are symmetric around positions 3, 6, 9 in string "abaaba"? That's because there is a palindromic substring around these positions. Same is the case in string "abababa" around position 7.
Is it always true that LPS length values around at palindromic center position are always symmetric (same)?
Answer is NO.
Look at positions 3 and 11 in string "abababa". Both positions have LPS length 3. Immediate left and right positions are symmetric (with value 0), but not the next one. Positions 1 and 5 (around position 3) are not symmetric. Similarly, positions 9 and 13 (around position 11) are not symmetric.

At this point, we can see that if there is a palindrome in a string centered at some position, then LPS length values around the center position may or may not be symmetric depending on some situation. If we can identify the situation when left and right positions WILL BE SYMMETRIC around the center position, we NEED NOT calculate LPS length of the right position because it will be exactly same as LPS value of corresponding position on the left side which is already known. And this fact where we are avoiding LPS length computation at few positions makes Manacher's Algorithm linear.

In situations when left and right positions WILL NOT BE SYMMETRIC around the center position, we compare characters in left and right side to find palindrome, but here also algorithm tries to avoid certain no of comparisons. We will see all these scenarios soon.

Let's introduce few terms to proceed further:

(click to see it clearly)

- **centerPosition** – This is the position for which LPS length is calculated and let's say LPS length at centerPosition is d (i.e. L[centerPosition] = d)
- **centerRightPosition** – This is the position which is right to the centerPosition and d position away from centerPosition (i.e. **centerRightPosition = centerPosition + d**)
- **centerLeftPosition** – This is the position which is left to the centerPosition and d position away from centerPosition (i.e. **centerLeftPosition = centerPosition − d**)
- **currentRightPosition** – This is the position which is right of the centerPosition for which LPS length is not yet known and has to be calculated
- **currentLeftPosition** – This is the position on the left side of centerPosition which corresponds to the currentRightPosition
  centerPosition − currentLeftPosition = currentRightPosition − centerPosition
  currentLeftPosition = 2* centerPosition − currentRightPosition
- **i-left palindrome** – The palindrome i positions left of centerPosition, i.e. at currentLeftPosition
- **i-right palindrome** – The palindrome i positions right of centerPosition, i.e. at currentRightPosition
- **center palindrome** – The palindrome at centerPosition

When we are at centerPosition for which LPS length is known, then we also know LPS length of all positions smaller than centerPosition. Let's say LPS length at centerPosition is d, i.e.
L[centerPosition] = d

It means that substring between positions "centerPosition-d" to "centerPosition+d" is a palindrom.
Now we proceed further to calculate LPS length of positions greater than centerPosition.
Let's say we are at currentRightPosition (> centerPosition) where we need to find LPS length.
For this we look at LPS length of currentLeftPosition which is already calculated.

If LPS length of currentLeftPosition is less than "centerRightPosition – currentRightPosition", then LPS length of currentRightPosition will be equal to LPS length of currentLeftPosition. So
L[currentRightPosition] = L[currentLeftPosition] if L[currentLeftPosition] < centerRightPosition – currentRightPosition. This is **Case 1**.

Let's consider below scenario for string "abababa":

(click to see it clearly)

We have calculated LPS length up-to position 7 where L[7] = 7, if we consider position 7 as centerPosition, then centerLeftPosition will be 0 and centerRightPosition will be 14.
Now we need to calculate LPS length of other positions on the right of centerPosition.

For currentRightPosition = 8, currentLeftPosition is 6 and L[currentLeftPosition] = 0
Also centerRightPosition – currentRightPosition = 14 – 8 = 6
Case 1 applies here and so L[currentRightPosition] = L[8] = 0
Case 1 applies to positions 10 and 12, so,
L[10] = L[4] = 0
L[12] = L[2] = 0

If we look at position 9, then:
currentRightPosition = 9
currentLeftPosition = 2* centerPosition – currentRightPosition = 2*7 – 9 = 5
centerRightPosition – currentRightPosition = 14 – 9 = 5

Here L[currentLeftPosition] = centerRightPosition – currentRightPosition, so Case 1 doesn't apply here. Also note that centerRightPosition is the extreme end position of the string. That means center palindrome is suffix of input string. In that case, L[currentRightPosition] = L[currentLeftPosition]. This is **Case 2**.

Case 2 applies to positions 9, 11, 13 and 14, so:
L[9] = L[5] = 5
L[11] = L[3] = 3
L[13] = L[1] = 1
L[14] = L[0] = 0

What is really happening in Case 1 and Case 2? This is just utilizing the palindromic symmetric property and without any character match, it is finding LPS length of new positions.

When a bigger length palindrome contains a smaller length palindrome centered at left side of it's own center, then based on symmetric property, there will be another same smaller palindrome centered on the right of bigger palindrome center. If left side smaller palindrome is not prefix of bigger palindrome, then **Case 1** applies and if it is a prefix AND bigger palindrome is suffix of the input string itself, then **Case 2** applies.

*The longest palindrome i places to the right of the current center (the i-right palindrome) is as long as the longest palindrome i places to the left of the current center (the i-left palindrome) if the i-left palindrome is completely contained in the longest palindrome around the current center (the center palindrome) and the i-left palindrome is not a prefix of the center palindrome (**Case 1**) or (i.e. when i-left palindrome is a prefix of center palindrome) if the center palindrome is a suffix of the entire string (**Case 2**).*

In Case 1 and Case 2, i-right palindrome can't expand more than corresponding i-left palindrome (can you visualize why it can't expand more?), and so LPS length of i-right palindrome is exactly same as LPS length of i-left palindrome.

Here both i-left and i-right palindromes are completely contained in center palindrome (i.e. L[currentLeftPosition] <= centerRightPosition – currentRightPosition)
Now if i-left palindrome is not a prefix of center palindrome (L[currentLeftPosition] < centerRightPosition – currentRightPosition), that means that i-left palindrome was not able to expand up-to position centerLeftPosition.

If we look at following with centerPosition = 11, then

(click to see it clearly)

centerLeftPosition would be 11 – 9 = 2, and centerRightPosition would be 11 + 9 = 20
If we take currentRightPosition = 15, it's currentLeftPosition is 7. Case 1 applies here and so L[15] = 3. i-left palindrome at position 7 is "bab" which is completely contained in center palindrome at position 11 (which is "dbabcbabd"). We can see that i-right palindrome (at position 15) can't expand more than i-left palindrome (at position 7).

If there was a possibility of expansion, i-left palindrome could have expanded itself more already. But there is no such possibility as i-left palindrome is prefix of center palindrome. So due to symmetry property, i-right palindrome will be exactly same as i-left palindrome and it can't expand more. This makes L[currentRightPosition] = L[currentLeftPosition] in Case 1.

Now if we consider centerPosition = 19, then centerLeftPosition = 12 and centerRightPosition = 26
If we take currentRightPosition = 23, it's currentLeftPosition is 15. Case 2 applies here and so L[23] = 3. i-left palindrome at position 15 is "bab" which is completely contained in center palindrome at position 19 (which is "babdbab"). In Case 2, where i-left palindrome is prefix of center palindrome, i-right palindrome can't expand more than length of i-left palindrome because center palindrome is suffix of input string so there are no more character left to compare and expand. This makes L[currentRightPosition] = L[currentLeftPosition] in Case 2.

Case 1: L[currentRightPosition] = L[currentLeftPosition] applies when:

- i-left palindrome is completely contained in center palindrome
- i-left palindrome is NOT a prefix of center palindrome

Both above conditions are satisfied when
L[currentLeftPosition] < centerRightPosition – currentRightPosition

Case 2: L[currentRightPosition] = L[currentLeftPosition] applies when:

- i-left palindrome is prefix of center palindrome (means completely contained also)
- center palindrome is suffix of input string

Above conditions are satisfied when
L[currentLeftPosition] = centerRightPosition – currentRightPosition (For 1st condition) AND
centerRightPosition = 2*N where N is input string length N (For 2nd condition).

Case 3: L[currentRightPosition] > = L[currentLeftPosition] applies when:

- i-left palindrome is prefix of center palindrome (and so i-left palindrome is completely contained in center palindrome)
- center palindrome is NOT suffix of input string

Above conditions are satisfied when
L[currentLeftPosition] = centerRightPosition – currentRightPosition (For 1st condition) AND
centerRightPosition < 2*N where N is input string length N (For 2nd condition).
In this case, there is a possibility of i-right palindrome expansion and so length of i-right palindrome is at least as long as length of i-left palindrome.

Case 4: L[currentRightPosition] > = centerRightPosition – currentRightPosition applies when:

- i-left palindrome is NOT completely contained in center palindrome

Above condition is satisfied when
L[currentLeftPosition] > centerRightPosition – currentRightPosition
In this case, length of i-right palindrome is at least as long (centerRightPosition – currentRightPosition) and there is a possibility of i-right palindrome expansion.

In following figure,

(click to see it clearly)

If we take center position 7, then Case 3 applies at currentRightPosition 11 because i-left palindrome at currentLeftPosition 3 is a prefix of center palindrome and i-right palindrome is not suffix of input string, so here L[11] = 9, which is greater than i-left palindrome length L[3] = 3. In the case, it is guaranteed that L[11] will be at least 3, and so in implementation, we 1st set L[11] = 3 and then

we try to expand it by comparing characters in left and right side starting from distance 4 (As up-to distance 3, it is already known that characters will match).

If we take center position 11, then Case 4 applies at currentRightPosition 15 because L[currentLeftPosition] = L[7] = 7 > centerRightPosition – currentRightPosition = 20 – 15 = 5. In the case, it is guaranteed that L[15] will be at least 5, and so in implementation, we 1^{st} set L[15] = 5 and then we try to expand it by comparing characters in left and right side starting from distance 5 (As up-to distance 5, it is already known that characters will match).

Now one point left to discuss is, when we work at one center position and compute LPS lengths for different rightPositions, how to know that what would be next center position. We change centerPosition to currentRightPosition if palindrome centered at currentRightPosition expands beyond centerRightPosition.

Here we have seen four different cases on how LPS length of a position will depend on a previous position's LPS length.
In Part 3, we have discussed code implementation of it and also we have looked at these four cases in a different way and implement that too.

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

Source

http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-2/

Chapter 15

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 3

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.

We have seen that there are no new character comparison needed in case 1 and case 2. In case 3 and case 4, necessary new comparison are needed.
In following figure,

(click to see it clearly)

If at all we need a comparison, we will only compare actual characters, which are at "odd" positions like 1, 3, 5, 7, etc.
Even positions do not represent a character in string, so no comparison will be preformed for even positions.
If two characters at different odd positions match, then they will increase LPS length by 2.

There are many ways to implement this depending on how even and odd positions are handled. One way would be to create a new string 1^{st} where we insert some unique character (say #, $ etc) in all even positions and then run algorithm on that (to avoid different way of even and odd position handling). Other

way could be to work on given string itself but here even and odd positions should be handled appropriately.

Here we will start with given string itself. When there is a need of expansion and character comparison required, we will expand in left and right positions one by one. When odd position is found, comparison will be done and LPS Length will be incremented by ONE. When even position is found, no comparison done and LPS Length will be incremented by ONE (So overall, one odd and one even positions on both left and right side will increase LPS Length by TWO).

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int expand = -1;
    int diff = -1;
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
       //get currentLeftPosition iMirror for currentRightPosition i
        iMirror  = 2*C-i;
        //Reset expand - means no expansion required
        expand = 0;
        diff = R - i;
       //If currentRightPosition i is within centerRightPosition R
```

```
if(diff > 0)
{
    if(L[iMirror] < diff) // Case 1
        L[i] = L[iMirror];
    else if(L[iMirror] == diff && i == N-1) // Case 2
        L[i] = L[iMirror];
    else if(L[iMirror] == diff && i < N-1)  // Case 3
    {
            L[i] = L[iMirror];
            expand = 1;  // expansion required
    }
    else if(L[iMirror] > diff)  // Case 4
    {
        L[i] = diff;
        expand = 1;  // expansion required
    }
}
else
{
    L[i] = 0;
    expand = 1;  // expansion required
}

if(expand == 1)
{
//Attempt to expand palindrome centered at currentRightPosition i
    //Here for odd positions, we compare characters and
    //if match then increment LPS Length by ONE
   //If even position, we just increment LPS by ONE without
    //any character comparison
    while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
        ( ((i + L[i] + 1) % 2 == 0) ||
      (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
    {
        L[i]++;
    }
}

if(L[i] > maxLPSLength)  // Track maxLPSLength
{
    maxLPSLength = L[i];
    maxLPSCenterPosition = i;
}

// If palindrome centered at currentRightPosition i
// expand beyond centerRightPosition R,
```

```
        // adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    //printf("start: %d end: %d\n", start, end);
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}


int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();
```

```
    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

This is the implementation based on the four cases discussed in Part 2. In Part 4, we have discussed a different way to look at these four cases and few other approaches.

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

Source

http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-3-2/

Chapter 16

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 4

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. In Part 3, we implemented the same.
Here we will review the four cases again and try to see it differently and implement the same.
All four cases depends on LPS length value at currentLeftPosition (L[iMirror]) and value of (centerRightPosition – currentRightPosition), i.e. (R – i). These two information are know before which helps us to reuse previous available information and avoid unnecessary character comparison.

(click to see it clearly)

*If we look at all four cases, we will see that we 1st set minimum of **L[iMirror]** and **R-i** to **L[i]** and then we try to expand the palindrome in whichever case it can expand.*

Above observation may look more intuitive, easier to understand and implement, given that one understands LPS length array, position, index, symmetry property etc.

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
       //get currentLeftPosition iMirror for currentRightPosition i
        iMirror  = 2*C-i;
        L[i] = 0;
        diff = R - i;
       //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

     //Attempt to expand palindrome centered at currentRightPosition i
       //Here for odd positions, we compare characters and
```

```
        //if match then increment LPS Length by ONE
        //If even position, we just increment LPS by ONE without
        //any character comparison
        while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
            ( ((i + L[i] + 1) % 2 == 0) ||
            (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
        {
            L[i]++;
        }

        if(L[i] > maxLPSLength)  // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        //If palindrome centered at currentRightPosition i
        //expand beyond centerRightPosition R,
        //adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
```

```
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

Other Approaches

We have discussed two approaches here. One in Part 3 and other in current article. In both approaches, we worked on given string. Here we had to handle even and odd positions differently while comparing characters for expansion (because even positions do not represent any character in string).

To avoid this different handling of even and odd positions, we need to make even positions also to represent some character (actually all even positions should represent SAME character because they MUST match while character comparison).

One way to do this is to set some character at all even positions by modifying given string or create a new copy of given string. For example, if input string is "abcb", new string should be "#a#b#c#b#" if we add # as unique character at even positions.

The two approaches discussed already can be modified a bit to work on modified string where different handling of even and odd positions will not be needed.

We may also add two DIFFERENT characters (not yet used anywhere in string at even and odd positions) at start and end of string as sentinels to avoid bound check. With these changes string "abcb" will look like "^#a#b#c#b#$" where ^ and $ are sentinels.
This implementation may look cleaner with the cost of more memory.

We are not implementing these here as it's a simple change in given implementations.

Implementation of approach discussed in current article on a modified string can be found at Longest Palindromic Substring Part II and a Java Translation of the same by Princeton.

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

Source

http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-4/