データ構造とアルゴリズム実験レポート

＜課題 7：文字列照合＞

202213025 – 3 クラス – Can Minh Nghia

締切日：2024/1/29

提出日：2024/1/28

必須課題

注意：Makefile の内容は課題によって違う。

基本課題 1:

```
mainNaive: cmp.o naive.o readFile.o
```

基本課題 2:

```
mainCompnext: compnext.o
```

ターミナルで実行するコマンド：　./mainCompnext

基本課題 3:

```
mainKMP: cmp.o kmp.o readFile.o compnext.o
```

課題 7-1

7-1-1 実装の仕様

単純照合法に基づいて，上記のコードにおける naive()を作成してプログラムを完成させなさい． 作成したプログラムを簡単な動作例を用いて説明し，それが正しく動作することを示しなさい.

7-1-2 実装コードおよび実装コードの説明

naive.c ファイルの naive()関数を解説する。

```
16 int naive(char* text, unsigned int textlen, char* pat, unsigned int patlen) {
17     int i = 0; //i is the index of pattern
18     int j = 0; //j is the index of text
19     while (j < textlen) {
20     Ncmp++;
21       if (pat[i] != text[j]) {
22               j = j - i + 1; //j firstly moves to the POSITION right next to the recent first matching character,
   then keeps going to the next position until a matching character is found
23               i = 0; //i will remains 0 until a matching character is found
24               }      else {
25                      if (i == patlen - 1) { //all pat[i]s are the same to text[j]s, which means pattern has found
   in text
26                              return j - patlen + 1; //the position of the pattern in texts
27                      } else {
28                              i++;
29                              j++;
30                      }
31               }
32       }
33       return -1; //failed
```

Line 19: j が text を超えない限りという条件を定める。

Line 20: 発展課題に使うので今は無視する。

Line 21~23: j を次の位置に移動する。簡単な例を使って説明すると：

text: ABDABABC

pat: ABC

i=0, j=0 と i=1, j=1 の時は text[j] = pat[i]けど、i=3, j=3 の時、text[3]=D,pat[3]=C、よって違う。初めて文字が違う時に、j=j-i+1=1, i=0。その後、i がずっと 0 のままに(pat[0]と比較するために使う意味もあれば、j=j-0+1=j+1 で j が次の位置に移動する意味もある)、j は 1-0+1=2, 2-0+1=3 の順で、次の位置に一つずつ移動する。

この時、text[3] = pat[0] = A。これから text[j] = pat[i]の処理を進む。

次に text[j] != pat[i]のときは j=5, i=2、text[5]=A だけど pat[2]=C。こんな時に、j=j-i+1=5-2+1=4, i=0, text[4] != pat[0]だから、i がずっと 0 のままに、 j が引き続き j=4-0+1=5 になる。

この時、text[5] = pat[0] = A。これから text[j] = pat[i]の処理を進む。

ということで j=j-i+1 は何をするのかというと、

```
22            j = j - i + 1; //j firstly moves to the POSITION right next to the recent first matching character,
  then keeps going to the next position until a matching character is found
23            i = 0; //i will remains 0 until a matching character is found
```

このようにコメントも書いているが、合っていない文字が見つかったら、j が最後に合っている文字列の最初の文字の次の位置に移動する。その後、pat[0]に合っている文字が見つかるまで j が次の位置に一つずつ移動する。上の例を使って説明すると：

text: ABDABABC

pat: ABC

まず、text[3]!=pat[3]の時点で、最後に合っている文字列は AB(つまり text[0]text[1])である。j=j-i+1 の式で、j が最初の文字(A、つまり text[0]) の次の位置(B つまり text[1])に移動する。その後、j が text[2]、text[3]の順に移動する。text[3]が pat[0]にあっているから、これで j=j-i+1 の役割が終わる。

次に、text[5]!=pat[2]の時点で、最後に合っている文字列は AB(つまり text[3]text[4])である。j=j-i+1 の式で、j が最初の文字(A、つまり text[3]) の次の位置(B つまり text[4]) に移動する。その後、j が text[5]に

移動する。text[5]が pat[0]にあっているから、これで j=j-i+1 の役割が終わる。

Line 24: text[j]==pat[i]の処理。

Line 25: i が pat の最後まで進むことができた、つまり text の中に pat にあっている部分が見つかる。

Line 26: この時に、j-patlen+1 の式で pat の出現位置を返す。

Line 27~30: i が pat の最後まで進むことがまだできていない。それで照合を引き続き行う。

Line 33: text の全部を確認したけど pat が見つからないので、失敗という意味で-1 を返す。

7-1-3 実行結果：

正しく動作することを示すために、サンプルコードの mainNaive.c ファイルを使う。

pat が text の中に存在する場合：

```
azalea02:~ s2213025$ echo ABDABABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 9
pattern size: 3
Pattern found at 5.
```

pat が text の中に存在しない場合：

```
azalea02:~ s2213025$ echo ABE > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 9
pattern size: 3
Pattern found at -1.
```

これで、正しく動作することがわかる。

課題 7-2

7-2-1 実装の仕様

上記ソースコードに基づいて，KMP 法によって文字列照合を行うプログラムを作成する． まず，パターン pat を解析してまくらパターンのサイズを格納する配列 next を計算する関数

```
void compnext(char* pat, int* next)
```

を含むソースファイル compnext.c を作成しなさい． さらに，作成した関数が正しく動作することを，例を使って示しなさい.

7-2-2 実装コードおよび実装コードの説明

まず、compnext.c の構造を紹介する。

このファイルは max()と compnext()という 2 つの関数を含める。

max()関数

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

int max(int left, int right) {
	if (left >= right) {
		return left;
	} else {
		return right;
	}
}
```

2 つの引数を渡し、大きいほうを返す。

compnext()関数

```
12 void compnext(char* pat, int* next) {
13     for (int a = 0; a < strlen(pat); a++) {
14         next[a] = 0;
15    }
16
17        int i = 1; //i is index of pat'
18        int j = 0; //j is index of pat
19
20
21        while (i < strlen(pat)) {
22              j = 0;
23              while (j < i) j++;
24
25              while (pat[j] == pat[j - i] && j < strlen(pat)) j++;
26
27              //pat[j] != pat[j - i]
28                    printf("j is %d , i is %d \n", j, i);
29                  int k = max(0, j - i + 1);
30                  printf("k is %d \n", k);
31                  if (k >= next[j]) {
32                           next[j] = k;
33
34                    }
35              i++;
36        }
37
38 }
```

Line12~15: next 行列の初期化。

Line 21+Line 35: pat に pat'をまだ重ねていたら一つずつずらす.

i=1 というのは 1 つの文字をずらすことで、i=2 というのは 2 つの文字をずらすこと等々と意味している。

Line 22: j を pat の最初の要素に戻る。

Line 23: j を pat'の最初の要素と同じ位置に設定する。

Line 25:現在の文字があっていたら、次の文字を照合する。ちなみに、j-i は何かというと、それは pat[j]に相当する pat'のインデックスである。図にすると：

```
                    0  1  2  3  4
                pat A  B  A  B  C
i=1             pat'   A  B  A  B    //pat[j]は pat'[j-1]に相当する
i=2             pat'      A  B  A    //pat[j]は pat'[j-2] に相当する
i=3             pat'         A  B    //pat[j]は pat'[j-3] に相当する
i=4             pat'            A    //pat[j]は pat'[j-4] に相当する
```

i=5 は存在しない (line 21 によって、i はもう strlen にイコールだから while ループの条件を満たさない)。

上の図を見ると、pat に pat'を重ねる部分の中で、pat[j]の位置は pat'[j-1]の位置に相当する。例えば、i=1 のときは、pat[2]の位置は pat'[1]の位置と相当する。

Line 27: 重ねる部分の中に合わない文字があったらという処理のはじまり。読みやすいようにコメントにした。

Line 28, 30:デバッグに使う。

Line 29: next[j] = max(next[j], pat'[合わない文字の位置] + 1)であるから、まず pat'[合わない文字の位置]が負数でないことを確認する。

Line 31~32:新しい next[j]が元の next[j]より大きい場合だけ next[j]の値を更新する。

7-2-3 実行結果：

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

void compnext(char* pat, int* next);

int main(int argc, char** argv) {
    char pattern[100]; // Maximum length for the pattern string
    int next[100];     // Maximum length for the next array

    // Input the pattern string from the user
    printf("Enter the pattern string: ");
    scanf("%s", pattern);

    // Compute the next array
    compnext(pattern, next);

    // Print the next array
    printf("Next array: ");
    for (int i = 0; i < strlen(pattern); i++) {
        printf("%d ", next[i]);
   }
    printf("\n");

    return 0;
}
```

compnext.c ファイルの動作を確認するために、mainCompnext.c を書く。

Line 7~8: pat と next の長さの上限を設定する。要求に応じてその上限を変更できる。

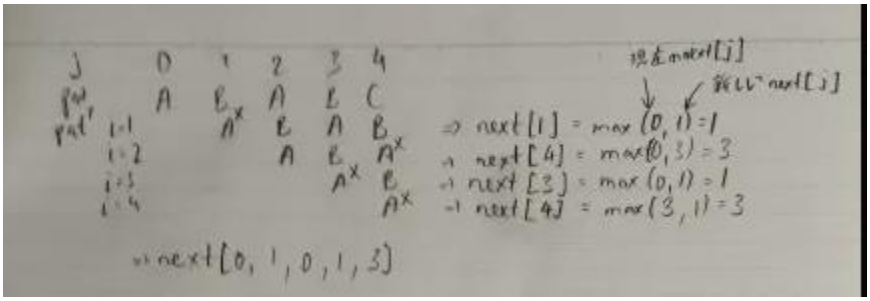
Line 11~12: scanf 関数で pat を読み取る。

Line 15: その pat を compnext 関数に渡して処理を行う。

Line 18~24: next 行列を出力する。

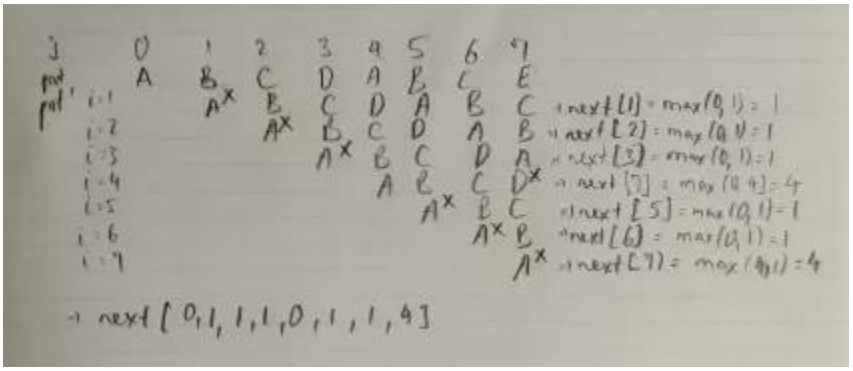
2 つの例を使う。

pat = ABABC



```
azalea02:~ s2213025$ ./mainCompnext
Enter the pattern string: ABABC
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Next array: 0 1 0 1 3
```

pat = ABCDABCE



```
azalea02:~ s2213025$ ./mainCompnext
Enter the pattern string: ABCDABCE
j is 1 , i is 1
k is 1
j is 2 , i is 2
k is 1
j is 3 , i is 3
k is 1
j is 7 , i is 4
k is 4
j is 5 , i is 5
k is 1
j is 6 , i is 6
k is 1
j is 7 , i is 7
k is 1
Next array: 0 1 1 1 0 1 1 4
```

従って、compnext 関数と compnext.c ファイルが正しく動作することがわかる。

課題 7-3

7-3-1 実装の仕様

KMP 法によって文字列照合を行うプログラム kmp.c を完成させなさい. このために修正したメイン関数は mainKMP.c とする. さらに, 作成したプログラムを簡単な動作例を用いて説明し, それが正しく動作することを示しなさい.

7-3-2 実装コードおよび実装コードの説明

kmp.c ファイル

```
1 #include <stdlib.h>
2 #include <stdbool.h>
3 #include <string.h>
4 #include <stdio.h>
5
6 extern bool isVerbose;
7 extern int Ncmp;
8
9 bool cmp(char, char);
10
11 void compnext(char* pat, int* next);
12
```

Line 6~9: サンプルコードの naive.c と同じ。

Line 11: compnext 関数を使うから呼び出す。

kmp()関数

```
20 int kmp(char* text, unsigned int textlen, char* pat, unsigned int patlen) {
21     int n = strlen(text);
22     int m = strlen(pat);
23     int i = 0;
24     int j = 0;
25     int next[m];
26     Ncmp = 0;
```

Line 23~24: i は pat のインデックスで、j は text のインデックス。

Line 25: 発展課題のために使う。今は無視する。

```
28     compnext(pat, next);
29
30       while (j < n) { //j < textlen
31             //pat[i] and text[j] are not the same
32             //while loop: find in pattern until find some pat[i] == text[j]
33             Ncmp++;
34             while (i >= 0 && pat[i] != text[j]) {
35                   i = next[i] - 1;
36          }
```

Line 28: compnext 関数を使って next 行列を計算する。

Line 30: while ループで text のすべての文字をチェックする。

Line 33: 発展課題用。今は無視する。

Line 34: i がまだ pat の範囲にある、かつ pat[i]の文字が next[j]の文字と違ったらという条件を設定する。

Line 35:　i が next[i]-1 の位置に移動する。

```
38          // pat[i] == text[j]
39                if (i != m - 1) {
40                      i++;
41                      j++;
42                } else {
43                      return j - m + 1; // success
44                }
45
46          }
47          return -1; //failed
48  }
```

Line 38: pat[i]の文字が next[j]の文字と同じの場合の処理の始まりという意味。

Line 39~41: i がまだ pat の末尾に着いていなかったら、i が pat の次の位置に移動する同様に、j も next の次の位置に移動する。その上、i=-1 の時、i が 0 に戻る、つまり pat の先頭に戻ると共に、j が text 上の次の位置に移動する意味もある。

Line 42~44: i が pat の末尾に着た、つまり text に pat が見つかったら、pat が出現する位置を返す。

Line 47: text に pat が存在しなかったら-1 を返す。

mainKMP()関数

サンプルコードの mainNaive.c と同じだが、naive だと書いてあるところを全部 kmp に変更する。

```
12 int readFile(char*, char*);
13 int kmp(char* text, unsigned int textlen, char* pat, unsigned int patlen);

43   printf("Pattern found at %d.\n", kmp(text, textlen, pat, patlen));
44   if (isVerbose)
```

簡単な動作例を用いて説明すると、こうなる：

text: ABABDABABABC

pat: ABABC

next[0 1 0 1 3] (基本課題 2 で説明した)。

まず、i=0, j=0, pat[0]=text[0], よって Line 39~41 の処理を行う。i=1,2,3 に相応する j=1,2,3 も同じ処理を行う。

text[4]!=pat[4]、よって Line 34~36 の処理を行う。i の値が 4 から next[4]-1 の値になる。つまり i=3-1=2。text[4]!=pat[2]、よって i がさらに next[2]-1 の値になる。つまり i=0-1=-1。

この時、Line 39~41 の処理を行う。i=-1+1=0, j=4+1=5。

pat[0]=text[5]、Line 39~41 の処理を行う。i=1,2,3 と共に j=5,7,8 まで Line 39~41 の処理を行う。

pat[4]!=text[9]。よって Line 34~36 の処理を行う。i の値が 4 から next[4]-1 の値になる。つまり i=3-1=2。

pat[2]=text[9]、Line 39~41 の処理を行う。i=3,4 と共に j=10,11 まで Line 39~41 の処理を行う。

この時、i=m-1=5-1=4、よって Line 42~44 の処理を行う。return j-m+1=11-5+1=7。

7-3-3 実行結果

上の例を実行したらこのような結果が出た。

```
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ echo ABABDABABABC > text
azalea02:~ s2213025$ ./mainKMP text pat
text size: 13
pattern size: 5
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Pattern found at 7.
```

pattern found at 7 という結果が得られた。上の計算と同じである。ちなみに、j is、i is、k is というのは compnext 関数にあるデバッグ用の printf 関数によるものである。

pat が存在しない場合、結果がこうなる。

```
azalea02:~ s2213025$ echo ABABDABACB > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 11
pattern size: 5
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Pattern found at -1.
```

ということで、kmp が正しく動作することがわかる。

考察と感想

naive 照合方法はわかりやすいけれど、kmp が分かりにくい。動作確認ができたけど、どうすればそんな難しい方法を思い付くことができたのかわからない。kmp の主な意図はまくらパターンがあれば、すでにあっている部分をもう照合しないことが分かった。ただ、next 行列の役割を完全に理解することがまだできていない。

発展課題

課題 7-1

7-1-1 実装の仕様

長さ n の文字列に対して，長さ m のパターンを照合することを考える． 異なる長さの文字列に対して，単純照合法と KMP 法の比較回数を調査し，どちらが高速か検討しなさい．

まず、naive()と kmp()の大きい while ループのなかに Ncmp++;を加える。

```
16 int naive(char* text, unsigned int textlen, char* pat, unsigned int patlen) {
17     int i = 0; //i is the index of pattern
18     int j = 0; //j is the index of text
19     while (j < textlen) {
20     Ncmp++;
```

```
20 int kmp(char* text, unsigned int textlen, char* pat, unsigned int patlen) {
21     int n = strlen(text);
22     int m = strlen(pat);
23     int i = 0;
24     int j = 0;
25     int next[m];
26     Ncmp = 0;
27
28     compnext(pat, next);
29
30       while (j < n) { //j < textlen
31             //pat[i] and text[j] are not the same
32             //while loop: find in pattern until find some pat[i] == text[j]
33             Ncmp++;
```

それに、mainNaive.c と mainKMP.c の中に、isVerbose の価値を false から true にする。

```
9 bool isVerbose = true;      // 比較回数表示スイッチ
```

なぜかというと、サンプルコードの下記の部分を使うからだ。

```
40   if (isVerbose) {
41     printf("text size: %d\n", textlen);
42     printf("pattern size: %d\n", patlen);
43   }
44   printf("Pattern found at %d.\n", kmp(text, textlen, pat, patlen));
45   if (isVerbose)
46     printf("# of comparisons: %d.\n", Ncmp);
```

7-1-2 実装コードおよび実装コードの説明

まくらパターンがない場合:

pat: ABC

textlen = 5:

```
azalea02:~ s2213025$ echo ABABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 6
pattern size: 3
Pattern found at 2.
# of comparisons: 7.
```

```
azalea02:~ s2213025$ echo ABABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 6
pattern size: 3
j is 1 , i is 1
k is 1
j is 2 , i is 2
k is 1
Pattern found at 2.
# of comparisons: 5.
```

textlen = 10

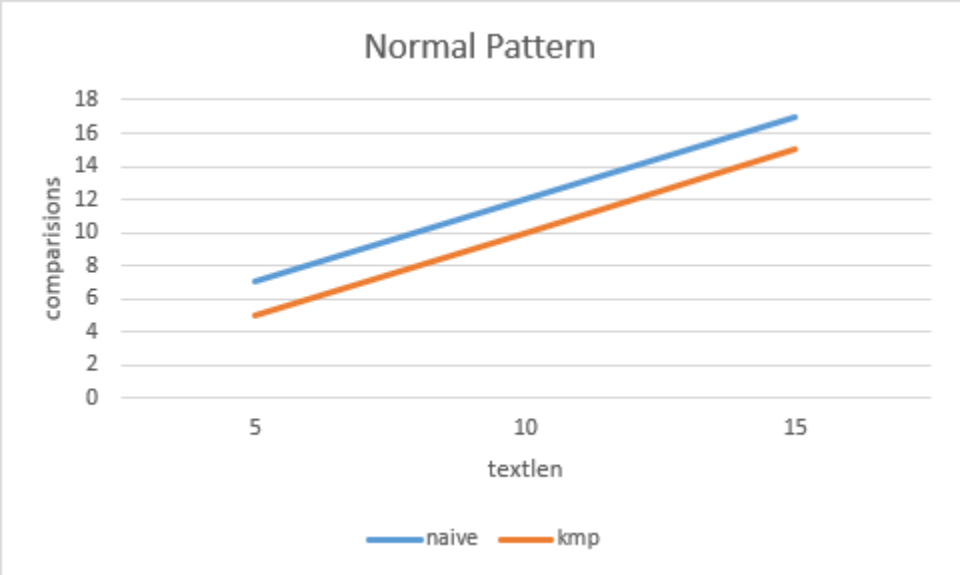
```
azalea02:~ s2213025$ echo ABDEFGHABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 11
pattern size: 3
Pattern found at 7.
# of comparisons: 12.
```

```
azalea02:~ s2213025$ echo ABDEFGHABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 11
pattern size: 3
j is 1 , i is 1
k is 1
j is 2 , i is 2
k is 1
Pattern found at 7.
# of comparisons: 10.
```

textlen = 15

```
azalea02:~ s2213025$ echo ABDEFGHIJKLMABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 16
pattern size: 3
Pattern found at 12.
# of comparisons: 17.
```

```
azalea02:~ s2213025$ echo ABDEFGHIJKLMABC > text
azalea02:~ s2213025$ echo ABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 16
pattern size: 3
j is 1 , i is 1
k is 1
j is 2 , i is 2
k is 1
Pattern found at 12.
# of comparisons: 15.
```



よって、naïve と kmp の比較回数が同様に安定して増加することがわかる。kmp の比較回数の方が少し少ないけれど、そんなに違いがないとも言える。

まくらパターンがある場合:

pat: ABABC

textlen = 10:

```
azalea02:~ s2213025$ echo ABOUGABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 11
pattern size: 5
Pattern found at 5.
# of comparisons: 12.
```

```
azalea02:~ s2213025$ echo ABOUGABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 11
pattern size: 5
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Pattern found at 5.
# of comparisons: 10.
```

textlen = 15

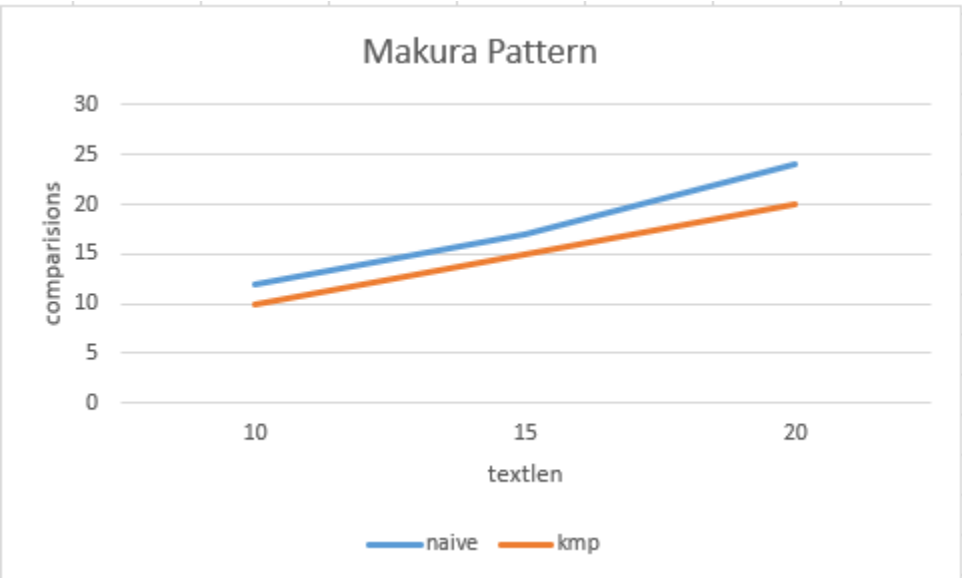
```
azalea02:~ s2213025$ echo ABOUGIOPTFABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 16
pattern size: 5
Pattern found at 10.
# of comparisons: 17.
```

```
azalea02:~ s2213025$ echo ABOUGIOPTFABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 16
pattern size: 5
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Pattern found at 10.
# of comparisons: 15.
```

textlen = 20

```
azalea02:~ s2213025$ echo ABOUGIOPTFABUYMABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 21
pattern size: 5
Pattern found at 15.
# of comparisons: 24.
```

```
azalea02:~ s2213025$ echo ABOUGIOPTFABUYMABABC > text
azalea02:~ s2213025$ echo ABABC > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 21
pattern size: 5
j is 1 , i is 1
k is 1
j is 4 , i is 2
k is 3
j is 3 , i is 3
k is 1
j is 4 , i is 4
k is 1
Pattern found at 15.
# of comparisons: 20.
```



よって、KMP のほうが Naïve より比較回数がいつも低いことがわかる。

結論として、まくらパターンがあってもなくても KMP のほうが高速である。

単純照合法（Naive string-matching algorithm）における最悪の場合は、必要なパターンが入力文字列全体で見つからない場合です。

課題 7-2

7-1-2 実装の仕様

単純照合法で最悪となる文字列およびパターンはどのようなものか．実例を挙げるとともに，計算量をオーダで示しなさい． さらに，単純照合法で最悪となる文字列およびパターンに対して KMP 法における比較回数を調査し，どちらが高速かを検討しなさい．

7-2-2 実装コードおよび実装コードの説明

単純照合法（Naive string-matching algorithm）における最悪の場合は、必要なパターンが入力文字列全体で見つからない場合である。

最悪の場合では、入力文字列全体を走査する必要がある。

pat=PQR とする

textlen=15

```
azalea02:~ s2213025$ echo ABCDEFGHIJKLMNO > text
azalea02:~ s2213025$ echo PQR > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 16
pattern size: 3
Pattern found at -1.
# of comparisons: 16.
```

textlen=30

```
azalea02:~ s2213025$ echo ABCDEFGHIJKLMNOSTUVWYZIOYTRFGH > text
azalea02:~ s2213025$ echo PQR > pat
azalea02:~ s2213025$ ./mainNaive text pat
text size: 31
pattern size: 3
Pattern found at -1.
# of comparisons: 31.
```

計算量のオーダ：O(textlen+1)

その text と pat を KMP 方法で実行したら、計算量が同じであることがわかる。

```
azalea02:~ s2213025$ echo ABCDEFGHIJKLMNOSTUVWYZIOYTRFGH > text
azalea02:~ s2213025$ echo PQR > pat
azalea02:~ s2213025$ ./mainKMP text pat
text size: 31
pattern size: 3
j is 1 , i is 1
k is 1
j is 2 , i is 2
k is 1
Pattern found at -1.
# of comparisons: 31.
```

よって、naïve の最悪ケースでは計算量が KMP でも naïve でも同じと言える。

考察と感想

最初は KMP の方が非常に naïve より性能が高いと思ったが、実はそんなに違いがないことを気づいた。今回の 2 つの方法をやったこと、単にパソコンの ctrl+F のことがそんなに複雑であることが分かった。