Today's contents

- Macros
  - Object-like macros and Function-like macros
- Pre-processor

- Command line arguments
  - "argc" and "argv"

- Measuring execution time
  - "time" command
  - "<time.h>" and "clock()"
- Getting the file size

- Character string search by Boyer-Moore method

今日の内容

- マクロ
  - オブジェクト形式マクロと関数形式マクロ
- プリプロセッサによる処理

- プログラムの実行時に引数を渡す
  - argcとargv

- 実行時間を計る
  - timeコマンド
  - <time.h>とclock()
- ファイルのサイズを知る

- BM法による文字列検索

Object-like macros (1)

- A program to process students' achievements
  - The number of students was five (5).
  - The number of subjects was five (5).

  - Students increased from five (5) to eight (8) !
  - Suppose you want to change "five" to "eight".
  - But, you must not change the number of subjects (it's also five (5)). So you cannot use a replace function on an editor.
    - You must check the whole of those numbers by your eyes.
    - It's very hard if the program has thousands of lines.

オブジェクト形式マクロ（1）

- 学生の成績を処理するプログラムを作った
  - 作ったときの学生の数は5人
  - 科目数も5科目

  - 学生の人数が増えた！8人になった！
  - 「5」を「8」に変更したい！
  - でも、科目数の”5”は”8”に変えてはいけないから、エディタの一括置換は使えない
    - 全部目で見てチェックしなきゃいけない！
    - プログラムが何千行もあったら、すごく大変だ！

Object-like macros (2)

- Let's use macros !

```
#define STUDENTNUMBER 5

int tensu[STUDENTNUMBER];
```

  - Define a macro by "#define"
    - Notation beginning with "#" are commands to the compiler ("#include" is one of them).
  - THE CAPITAL LETTERS are used (it's a custom).
    - It becomes a kind of variables which can be used in a program.
  - Benefits
    - You can discriminate roles of the numbers easily !
    - You can change the program easily !
    - You can decrease coding mistakes in the program !

オブジェクト形式マクロ(2)

- こんなときにマクロを使おう！

```
#define STUDENTNUMBER 5

int tensu[STUDENTNUMBER];
```

  - “#define”で定義する
    - ”#”で始まるものは全部コンパイラへの命令(”#include”も)
  - 大文字で定義する(慣習)
    - プログラムの中で使える変数みたいなもの
  - 利点
    - 何の数値か分かりやすくなる！
    - 変更が容易になる！
    - プログラムの間違いが減る！

Function-like macros (1)

- You can write a macro like as a function.

```
#define  sqr(x)   ((x) * (x))   /* calculate square */

...

  printf( “Square of the number is %d.¥n”, sqr(nx));
```

  - You have not to mind the type of arguments and the return value.
    - CAUTION: Automatic type conversion will be applied !
  - The macros are expanded at compiling.
    - Generally, program size ⇒ large, running speed ⇒ fast

関数形式マクロ（1）

- 関数っぽくマクロを書くことができます

```
#define  sqr(x)   ((x) * (x))   /* 二乗を計算する関数形式マクロ */
...
  printf(“その数の二乗は%dです。¥n”, sqr(nx));
```

  - 引数・返却値の型を気にしなくてもよい
    - ただし、自動的に型が拡張されるので注意！
  - マクロなので、コンパイル時に展開されます
    - 通常、プログラムサイズ⇒大、速度⇒速

Function-like macros (2)

- WARNING: Side effects
  - sqr(a++) ⇒ (a++)*(a++)
  - It causes unintended results.
- Put the arguments and the whole of macro in parentheses !
  - You might get a strange expression when expanded

```
#define  add(x,y)  x + y

...

z = add(a, b) * add(c, d); ⇒ z = a + b * c + d; Unintended result !
```

関数形式マクロ（2）

- 副作用に注意
  - sqr(a++) ⇒ (a++)*(a++)
  - 意図しない結果に！
- 引数・マクロ全体を ()でくくろう！
  - 式を入れるとおかしなことに

```
#define  add(x,y)  x + y

…

z = add(a, b) * add(c, d); ⇒ z = a + b * c + d; 意図しない結果に！
```

Comma operator

- Combining expressions by “,”, they become one expression.
  - evaluated from left to right
  - You can embed plural expressions into a function-like macro by using this operator.
  - A macro that warns, then display the strings.

```
#define  putsa(str)  ( putchar( '¥a' ), puts(str) )
```

コンマ演算子

- “,”で式をつなぐと、一つの式になる
  - 左から順に評価される
  - これを利用して、関数形式マクロに
    複数の式(複文)を埋め込むことができる

  - 警報を発して文字列を表示するマクロ

```
#define  putsa(str)  ( putchar( '¥a' ), puts(str) )
```

Looking the expansion of macros

- Give “-E” option to gcc when compiling
  - like “gcc –E your_source_code.c”
  - Only pre-processing will be done (object files are not generated)
    - Only the lines beginning with “#” in the C source files are processed.

- You can find that the processes for “#include” or “#define” have done.
  - File inclusion (#include)
  - Replacing macro strings (#define)
  - Conditional source selection (#if, #ifdef)
  - NOTICE: Only replacement of macro strings has done before compiling for function-like macros as well as object-like macros.

マクロの展開のされ方を見る

- コンパイルするときに、gccにオプションとして”-E”を与える
  - “gcc –E your_source_code.c” のようにする
  - コンパイルされず、プリプロセス (preprocess) だけが行われる
    - C言語ソースファイル中の”#”で始まる行の処理だけが行われる

- #includeや#defineの処理が行われていることが分かる
  - ファイルの挿入（#include）
  - マクロの文字の置き換え（#define）
  - 条件によるソースの選択（#if, #ifdef）
  - 関数形式マクロもあくまでマクロなので「コンパイル前に文字の置き換えが行われるだけ」なことに注意せよ

Pre-processor

- Commands to the compiler beginning with “#”
  - #include
    - You can include a file there.
    - You can include not only include files but also any files.
  - #define
    - definition of macros (as explained so far)
  - #if (x) A（#else B）#endif
    - If x is true, the code “A” will be utilized (if false, the code “B” will be utilized).
    - Like as “#if 0 ～ #endif”, you can comment out plural lines.
      A nest of comments is okay by this (A nest of /* ～ */ is not allowed).
  - #ifdef (x) A（#else B）#endif
    - If x is “defined”, the code “A” will be utilized (if undefined, the code “B” will be utilized).

プリプロセッサ

- 「#」で始まるコンパイラへの命令群（の中でよく使うもの）
  - #include
    - 指定したファイルをそこに読み込む
    - ヘッダファイルでなくても読み込める
  - #define
    - マクロ定義（前述の通り）
  - #if (x) A（#else B）#endif
    - x が「真」なら A のコードが生きる（「偽」なら B のコードが生きる）
    - #if 0 ～ #endif で複数行をコメントアウトできる。
      コメントの入れ子も可能に！（/* ～ */だと入れ子にできない）
  - #ifdef (x) A（#else B）#endif
    - x が「定義済み」なら A のコードが生きる（「未定義なら」 B のコードが生きる）

inline

- Honestly, I don't recommend function-like macros because complex function-like macros cause unpredictable side effects.
- Use "inline" instead of function-like macros
  - But, only after C99

```
inline int sqr(int x)
{
  return x * x;
}
```

  - This function will be EMBEDDED into the place where it will be used when compiling.
    - The compiler won't generate code as an individual function.
  - You can write "sqr(a++)" without any side effects !

inline

- 正直な話、複雑なマクロを使うと、予測できない副作用が起きることが多いので、関数形式マクロはお勧めできない
- 代わりに inline を使いましょう
  - ただし、C99以降

```
inline int sqr(int x)
{
  return x * x;
}
```

  - コンパイル時に使用場所に埋め込まれる
    - 関数として独立にコード生成されない
  - sqr(a++)と書いても大丈夫

Command line arguments

- Suppose that you want to give parameters when running the code.
  - Control of options, etc.

- Giving the parameters to the program as arguments.
  - "-l" in "$ ls -l"
  - after "gcc" in "$ gcc –o exp_6_2 exp_6_2.c"

実行時引数

- プログラムの実行時に何らかのパラメータを与えたい
  - オプションのコントロール等
- プログラムの引数としてそのパラメータを渡したい
  - “$ ls -l”の”-l”とか
  - “$ gcc –o exp_6_2 exp_6_2.c”のgccから後ろの部分とか

“argc” and “argv”

- Variables which are set when starting the code.

- int argc
  - The number of arguments
  - If no arguments (only the program name), argc will be 1.
  - If one arguments, argc will be 2.

- char *argv[]
  - An array of argument strings
  - argv[0] has a string of the program name.
  - argv[1] has a string of the first argument.

argcとargv

- プログラムの実行時に設定される変数

- int argc
  - 引数の数
  - プログラム名のみなら、argcは1になる
  - 引数を1つ付けて実行すると、argcは2になる

- char *argv[]
  - 引数の文字列の配列
  - プログラム名が、argv[0]に入っている
  - 1つ目の引数が、argv[1]に入っている

Simple example

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;
        printf("argc = %d¥n", argc);
        for (i = 0; i < argc; i++) {
                printf("argv[%d] = %s¥n", i, argv[i]);
        }
        return 0;
}
```

```
$ ./a.out
argc = 1
argv[0] = ./a.out
```

```
$ ./a.out 10 6
argc = 3
argv[0] = ./a.out
argv[1] = 10
argv[2] = 6
```

簡単な例

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;
        printf(“argc = %d¥n”, argc);
        for (i = 0; i < argc; i++) {
                printf(“argv[%d] = %s¥n”, i, argv[i]);
        }
        return 0;
}
```

```
$ ./a.out
argc = 1
argv[0] = ./a.out
```

```
$ ./a.out 10 6
argc = 3
argv[0] = ./a.out
argv[1] = 10
argv[2] = 6
```

Measuring execution time

- We often want to know the execution time of the program.
- Measuring the whole execution time of the program
  - “time” command
- We also want to know a part of the execution time in the program (e.g. a specified function).

実行時間を計る

- 実行時間を計りたい場面は多い
- プログラム全体の実行時間を計る
  - timeコマンドを使う
- プログラム全体ではなく、プログラムの中でどこに時間が掛かっているのか知りたいことも多い

clock() function

- clock_t clock()
  - a function to get the processor time
  - By calling the function both of before and after the process which you want to measure the time and taking difference of the times, you get the processor time taken for the process.
    - NOTICE: It's the processor time, not real time.
    - It does not include loss time caused by multi task processing.
    - If you use a multi-core CPU, all core time are summarized.
  - Return value type : clock_t
    - Dividing by CLOCKS_PER_SEC, the unit of the value will be second.
  - #include <time.h>

clock()関数

- clock_t clock()
  - プロセッサ時刻を得る関数
  - 時間を計りたい処理の前後で呼び出し、差を見ることで、その処理を行うためにプロセッサが使用した時間が分かる
    - 実時間ではなく、プロセッサ時間であることに注意
    - マルチタスクでロスした時間は入らない
    - マルチコアCPUでは全てのコアの時間が積算される
  - 返り値はclock_t型
    - CLOCKS_PER_SECで割ると、単位が秒になる
  - #include <time.h>

Getting file size

- Suppose that you want to read a whole file and place it on the memory.
  → You want to know allocate the memory matching with the file size.
  → You want to know the file size before reading the file.

- for UNIX OS
  - Use the structure “stat” and the function “stat()”

```
struct stat buf;  // definition of the structure variable
stat(“filename”, &buf);  // a file named “filename”
filesize = buf.st_size;  // the file size is in the variable “st_size”.
```

  - #include <sys/stat.h>

- A method with standard functions
  - Use the function “fseek()” and the function “fgetpos()”
  - A file has already opened by the function “fopen()”

```
fpos_t sz;  // a variable for the file size
fseek(fp, 0, SEEK_END);  // move the reading point to the end of the file
fgetpos(fp, &sz);  // the file size is in the variable “sz”.
```

  - #include <stdio.h>

ファイルのサイズを知る

- ファイルを全部読み込んでメモリ上に置きたい
  → ファイルのサイズに合わせて動的にメモリを確保したい
  → ファイルを読む前にファイルのサイズを知りたい

- UNIXの場合
  - stat構造体とstat()関数を使う

```
struct stat buf;  // 構造体定義
stat("filename", &buf);  // filenameというファイル
filesize = buf.st_size;  // st_sizeというメンバーがサイズ
```

  - #include <sys/stat.h>

- 標準関数を用いる手法
  - fseek()関数とfgetpos()関数を使う
  - ファイルはfopen()関数で既に開かれているとする

```
fpos_t sz;  // サイズを入れる変数
fseek(fp, 0, SEEK_END);  // ファイルの最後まで移動
fgetpos(fp, &sz);  // szにファイルサイズが入る
```

  - #include <stdio.h>

String search by Boyer-Moore method (1)

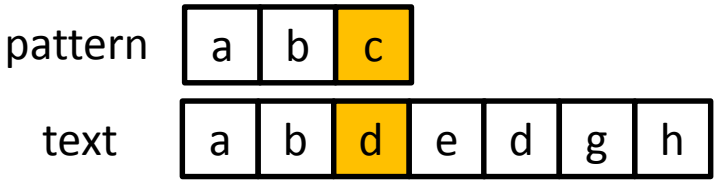
- Boyer-Moore (BM) method
  - Effective character string search method
  - To slide the search pattern effectively, the information “how long the search pattern can slide at once” is calculated in advance.
  - characterized by searching backward for the pattern

Boyer-Moore法による文字列検索（1）

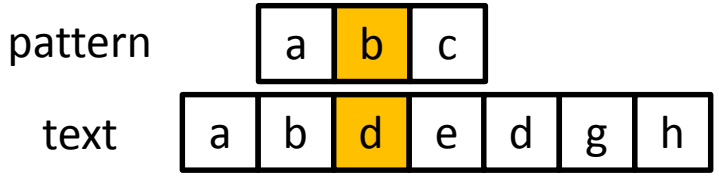
- Boyer-Moore法（BM法）
  - 効率の良い文字列検索法
  - 検索パターンをどこまでずらせるか？という情報を予め作っておいて、効率よく検索パターンをずらしていく
  - パターンの後ろから照合するのが特徴

String search by Boyer-Moore method (2)

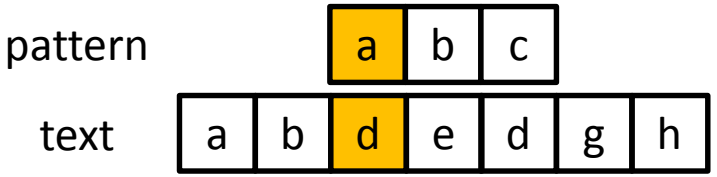
(1) The 3rd character of the pattern “c” doesn’t match with the 3rd character of the text “d”.



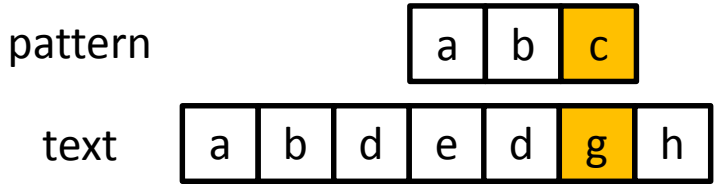
(2) If the pattern would be slid by one character, next matching must fail because the 3rd character of the text “d” is not included in the patter.



(3) Similarly, If the pattern would be slid by two characters, next matching must fail because the 3rd character of the text “d” is not included in the patter.

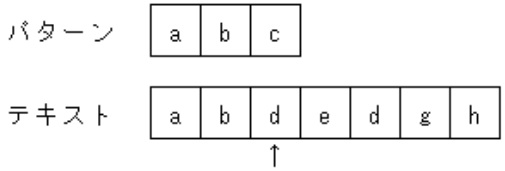


(4) Now you know that the pattern can be slid by three characters.

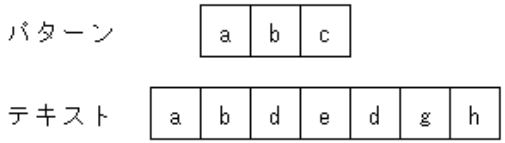


Boyer-Moore法による文字列検索(2)

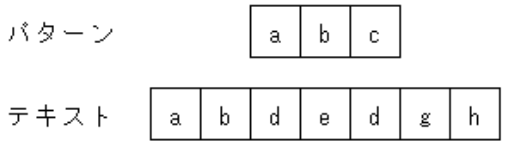
(1) パターンの3文字目cとテキストの3文字目dが一致しなかった



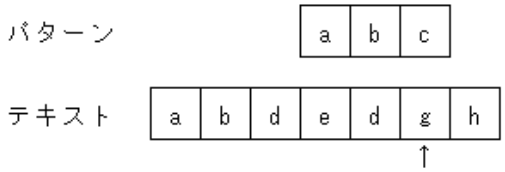
(2) もしパターンを1つずらしたとしても，テキストの3文字目のdは
パターンに含まれないので失敗するのは明らかである



(3) 同様に，パターンを2つずらしたとしても，テキストの3文字目のdは
パターンに含まれないので失敗する



(4) 結局(2)(3)のようにずらしたとしても失敗するのは明らかなので，
パターンを3つずらせばよいことになる



String search by Boyer-Moore method (3)

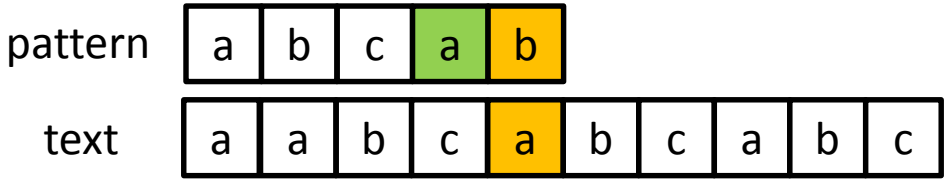
- How long can the pattern slide?
  - What number of characters from the tail of the pattern is a text character, when a pattern character wouldn't match the text character ? Keep the information as a table "skip".
    - If the text character appears in the pattern twice or more, the latest place will be taken.
  - Slide the pattern by the pattern length if the text character is not included in the pattern.

Boyer-Moore法による文字列検索（3）

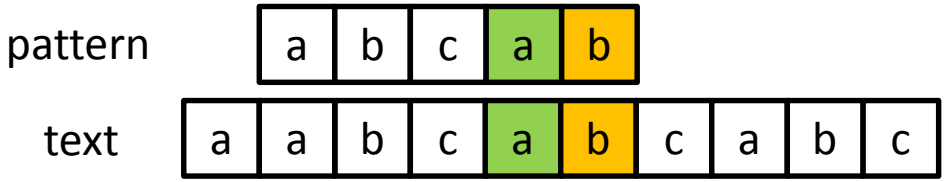
- ずらせる量の計り方
  - パターンとテキスト（検索対象）が一致しなかったとき、テキスト側の文字がパターンの（後ろから数えて）何文字目にあるか、という情報をテーブル（配列「スキップ」）で持っておく
    - パターン中に同じ文字が複数回出現する場合は、後ろ側の文字の位置とする
  - パターン中にその文字がない場合は、パターンの長さ分ずらせばよい

String search by Boyer-Moore method (4)

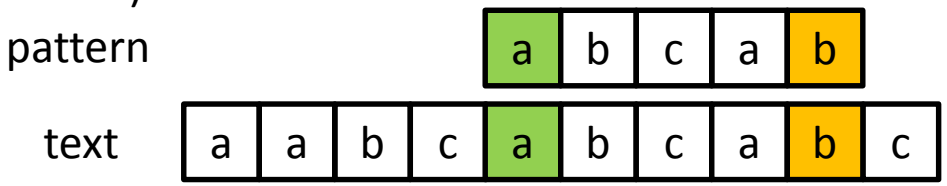
(1) The last character of the pattern does not match with the 5th character of the text.



(2) If the pattern would be slid by one character to match with the 4th character of the pattern “a”, the pattern “abcab” from the 2nd character of the text can be found. (Correct behavior)

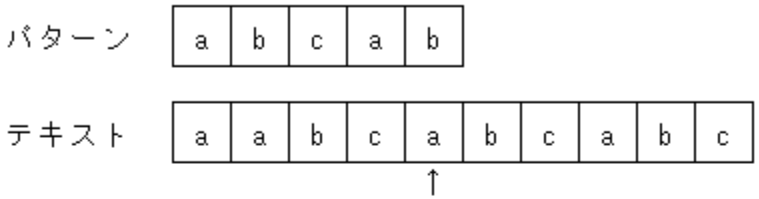


(3) If the pattern would be slid by four characters to match with the 1st character of the pattern “a”, the pattern “abcab” from the 2nd character of the text would be missed and the pattern “abcab” from the 5th character of the text would be found. (Wrong behavior)

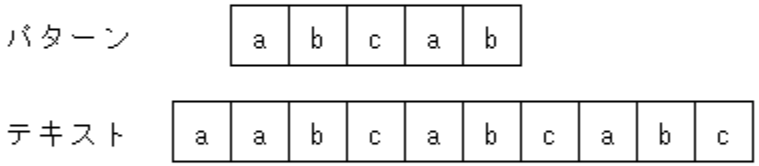


Boyer-Moore法による文字列検索（4）

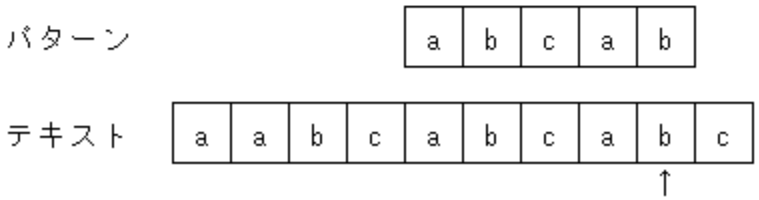
(1) パターンの最後の文字bとテキストの5文字目のaが一致しない



(2) パターンの4文字目のaに合うように，パターンを1文字ずらすと，
テキストの2文字目から始まるabcabが見つかる（正しい動作）

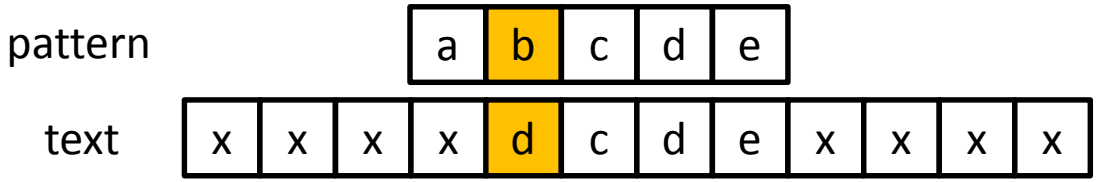


(3) パターンの1文字目のaに合うように，パターンを4文字ずらすと，
テキストの2文字目から始まるabcabを見逃して、代わりに5文字目
から始まるabcabを見つけてしまう（誤った操作）

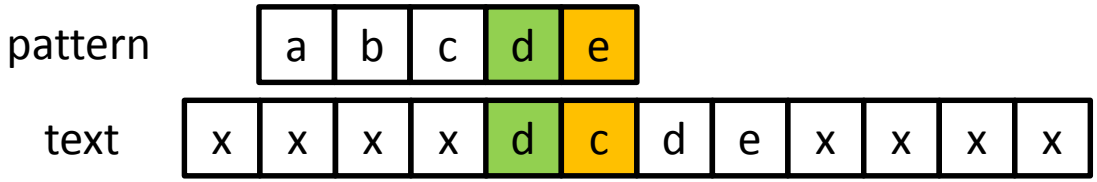


String search by Boyer-Moore method (5)

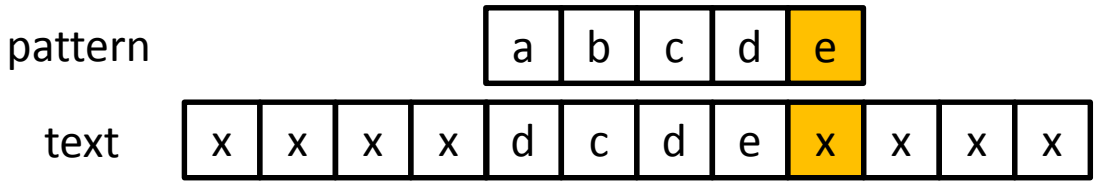
(1) The 2nd character of the pattern doesn't match with the character in the text "d".



(2) "d" is the 2nd character from the tail of the pattern. The focusing point to the text moves the next to match with the pattern. The pattern was rewound !

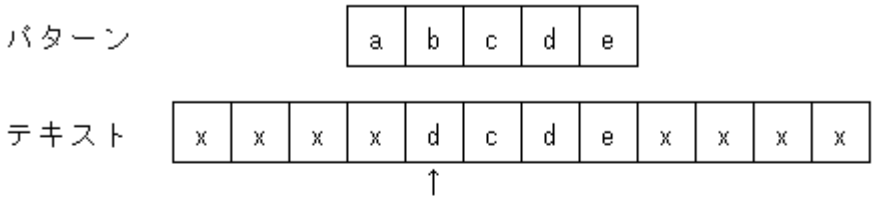


(3) If the pattern would rewind, the pattern will be slid by one character.

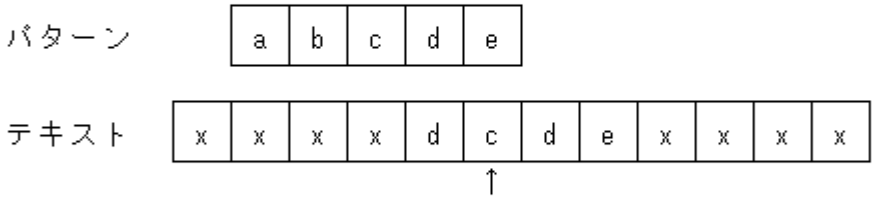


Boyer-Moore法による文字列検索(5)

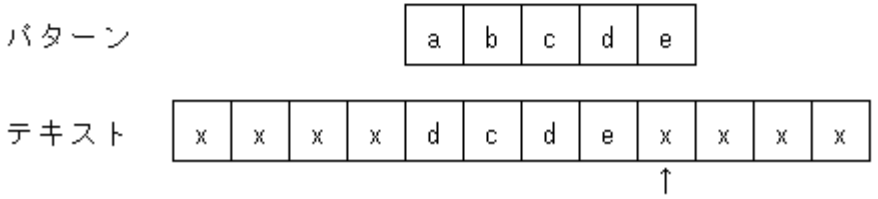
(1) パターンの2文字目のbがテキストのdと一致しない



(2) 文字dはパターンの後ろから2文字目に当たるので，テキストの注目点を1つ右へずらして，パターンを重ねる．何とパターンが戻ってしまう！



(3) もし，配列skipに従ってパターンを移動すると前に戻ってしまうなら，仕方がないのでパターンを1つだけずらす



String search by Boyer-Moore method (6)

- Rough procedure

```
while ( i < text_len ) {   // “i” means an attention point of the text
    j ← Tail index of the pattern (initial attention point of the pattern)
    while ( Compare characters of the text and pattern. Match them ? ) {
        When all characters matched, count up !
        Move back “j” by one character.
        Move back “i” by one character.
    }
    Move forward “i” according to the mismatched character.
    If rewinding, move forward “i” by one character.
}
```

Boyer-Moore法による文字列検索（６）

- おおまかな手順

```
while ( i < text_len ) {   // i はテキストの注目位置
    j ← パターンの末尾（最初の注目位置）
    while ( テキストとパターンを１文字比較。一致？ ) {
        すべての文字が一致したら、カウントを増やす
        テキストの注目位置 i を1文字戻す
        パターンの注目位置 j を1文字戻す
    }
    テキストの注目位置 i を、不一致になった文字に応じた文字数だけスライド
    巻き戻しが起きているなら、１文字だけ i を強制スライド
}
```