データ構造とアルゴリズム
（第13回）

モバイルコンピューティング研究室
柴田史久



1

1

本日の講義内容

- 文字列探索
  - 文字列に対する探索とは
  - 力まかせのアルゴリズム
  - Knuth-Morris-Prattのアルゴリズム
  - Boyer-Mooreのアルゴリズム

2

2

教科書 第18章（pp.411～434）

文字列の探索

3

3

文字列に対する探索(1)

- String
  - 文字やビットの「並んだ」データ構造
  - 文字列：character string / text string
  - ビット列：bit string
  - 一般には文字列＝string
  - 以降のアルゴリズムはビット列にも適用可能

4

4

文字列に対する探索(2)

- パターン（pattern）
  - 探し出したい文字の並び
- テキスト（text）
  - 探索される文字列
- 文字列の探索
  - テキストの中で，指定されたパターンが出現する場所を見つける操作
- パターンマッチ（pattern matching）
  - 不確定のパターンを探す操作
  - パターンの指定には正規表現（regular expression）
  - 別の問題なのでここでは扱わない

5

5

力まかせのアルゴリズム

- Brute-force algorithm
- 手順
  1. テキストの先頭にパターンを重ねる
  2. テキストとパターンが一致すれば探索成功
  3. 一致しなければパターンを1文字分だけ後ろのずらす
  4. 手順の2に戻って繰り返す
  5. テキストの最後尾に到達したら探索失敗

6

6

力まかせのアルゴリズムの実行過程

パターン　a c

テキスト　a b a c d

(1) 失敗

a c
a b a c d

(2) 失敗

a c
a b a c d

(3) 成功

a c
a b a c d

7

7

力まかせのアルゴリズムの実装

詳細は教科書 pp.414～415 List 18.1

```
public class BruteForce {
  public static int search(String text, String pattern) {
    int patLen  = pattern.length() ;
    int textLen = text.length() ;
    int i = 0 ; // 注目しているテキストの位置を表すポインタ
    int j = 0 ; // 注目しているパターンの位置を表すポインタ
    while (i < textLen && j < patLen) {
      if (text.charAt(i) == pattern.charAt(j)) {
        i++ ; j++ ; // 一文字比較して一致したらポインタを進める
      } else {
        i = i - j + 1 ; // テキストのポインタを現在注目している
                        // 先頭から1つ進める
        j = 0 ;         // パターンのポインタを先頭に戻す
      }
    }
    // 探索に成功したらパターンの位置を，失敗したら -1 を返す．
    return (j == patLen) ? (i - j) : -1 ;
  }
}
```

8

8

力まかせのアルゴリズムの計算量

- 文字列探索の計算量
  - テキスト1文字とパターン1文字の比較回数
  - テキストの長さ：n
  - パターンの長さ：m
- 最悪ケースの計算量
  - テキスト・パターンともに先頭から a が続き最後だけ b
  - テキストとパターンを重ねる位置：$n - m + 1$
  - 文字の比較回数はパターンの長さ：m
  - 比較回数：$m(n - m + 1)$
  - $n >> m$ ならば $O(mn)$

9

9

計算量の実際

- 自然言語・プログラミング言語では最悪ケースは稀
  - 文字の種類が多い　＝　先頭の数文字で比較が終了する
- 実質的計算量：$O(n)$
- 文字の種類が少ない場合は m が効いてくる

10

10

洗練されたアルゴリズム

- 力まかせのアルゴリズム
  - 最悪：O(mn)
  - 通常：O(n)
- S.A.Cookの証明（1970年）
  - 最悪の場合でも文字の比較を m + n に比例した回数だけ行う文字列探索アルゴリズムが存在する
- KMP法（Knuth-Morris-Prattのアルゴリズム）
  - D.E.Knuth，V.R.Pratt および J.H.Morris
  - 計算量：O(n)
- BM法（Boyer-Mooreのアルゴリズム）
  - 最悪：O(n)
  - 平均：O(n/m)

11

11

KMP法の原理(1)

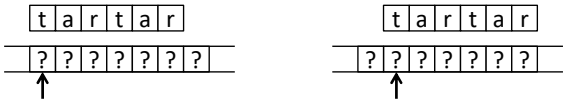
- 力まかせアルゴリズムの問題点
  - テキストとパターンが一致しなかった場合に，途中までの比較で得られた情報を捨てている
- KMP法のアイデア
  - パターンの各文字について，その文字で不一致になった際にどれだけずらせばよいかを事前に表にする
  - 探索時にこの表に基づいてパターンをずらしていく

12

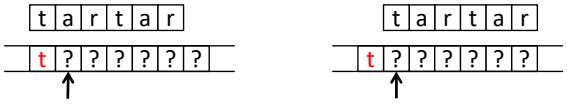
12

KMP法の原理(2)

- 1文字目で失敗
  - ポインタ+1，パターン+1，パターンの先頭から比較



- 2文字目で失敗：ポインタは既に+1
  - ポインタ±0，パターン+1，パターンの先頭から比較

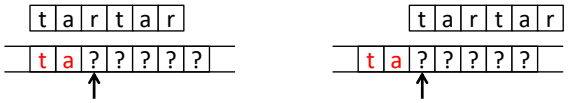


13

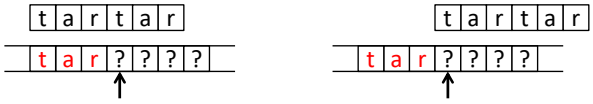
13

KMP法の原理(3)

- 3文字目で失敗：ポインタは既に+2
  - ポインタ±0，パターン+2，パターンの先頭から比較



- 4文字目で失敗：ポインタは既に+3
  - ポインタ±0，パターン+3，パターンの先頭から比較

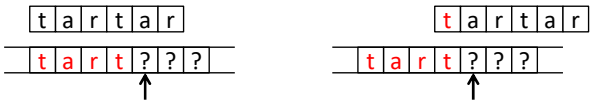


14

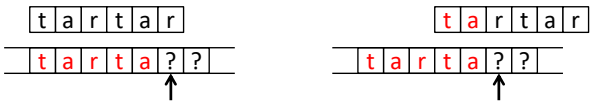
14

KMP法の原理(4)

- 5文字目で失敗：ポインタは既に+4
  - ポインタ±0，パターン+3，パターンの2文字目から比較



- 6文字目で失敗：ポインタは既に+5
  - ポインタ±0，パターン+3，パターンの3文字目から比較



15

15

KMP法の性質

- 最悪の計算量：$O(n)$
  - テキストとパターンを1回比較することで，テキストを指すポインタは必ず1つ以上進み，後戻りはない
- アルゴリズムが複雑なため定数項部分が大きい
  - 高速性の面ではKMP法を採用するメリットはあまりない
- 利点：テキストを指すポインタが後戻りしない
  - ファイルに格納されたテキストを探索する場合
- 「理論的には優れているが実戦には弱い」

16

16

BM法の原理(1)

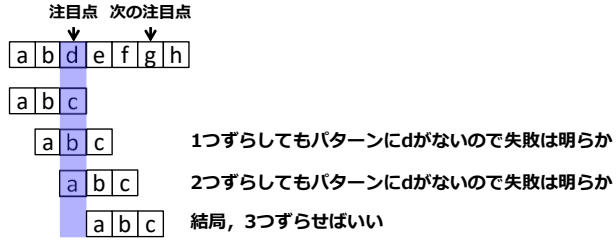
- 実用的には最も速い文字列探索アルゴリズム
- BM法のアイデア
  - パターンとテキストを重ね合わせる
  - 末尾から先頭に向かって順番に文字を比較
  - 不一致が見つかったら，不一致の原因となった文字に応じてパターンをずらす分量を決める

17

17

BM法の原理(2)

- パターンの3文字目 c と
  テキストの3文字目 d が不一致



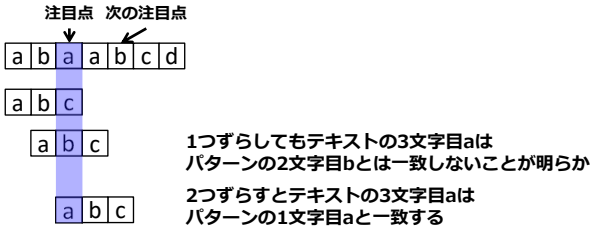
- 不一致文字がパターンに含まれない場合，
  m文字（＝パターン長）分ずらせばよい

18

18

BM法の原理(3)

- パターンの3文字目 c と
  テキストの3文字目 a が不一致



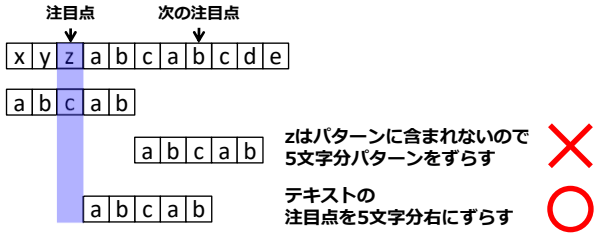
- パターンに含まれる文字で不一致になった場合，
  その文字がパターン末尾からx文字目なら
  x文字ずらせばよい

19

19

BM法の原理(4)

- パターンの3文字目 c と
  テキストの3文字目 z が不一致



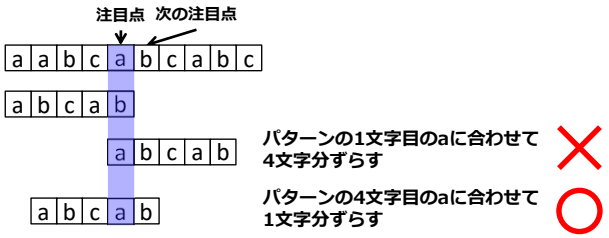
- 「パターンをx文字ずらす」のではなく
  「テキストの注目点をx文字ずらして，
  パターン末尾から比較を始める」

20

20

BM法の原理(5)

- パターンの5文字目 b と
  テキストの5文字目 a が不一致



- 最も末尾寄りに現れた文字の位置を元に
  移動量を決定する

21

21

BM法の実装(1)

詳細は教科書 pp.429～430 List 18.2

```
public class BoyerMoore {
  public static int search(String text, String pattern) {
    int patLen  = pattern.length() ;  // パターン長
    int textLen = text.length() ;     // テキスト長

    int[] skip = new int[65536] ; // Unicode分の表を準備

    Arrays.fill(skip, patLen) ;   // まずはパターン長の値で埋める
    // 先頭から表をつくる. 同一文字は最後のものが優先されることになる
    for (int x = 0 ; x < patLen - 1 ; x++) {
      skip[pattern.charAt(x)] = patLen - x - 1 ;
    }

    /*** 次ページに続く ***/
```

22

22

BM法の実装(2)

詳細は教科書 pp.429～430 List 18.2

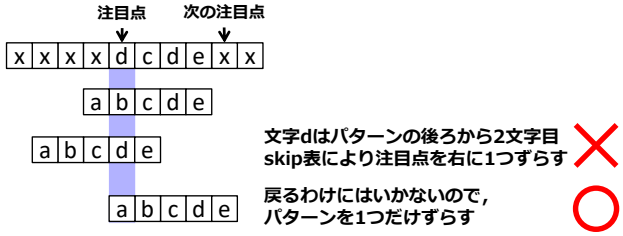
```
    /*** 前ページからの続き ***/

    int i = patLen - 1 ; // 注目点. パターン長 - 1 に初期化
    while (i < textLen) {  // テキスト最後尾まで繰り返す
      int j = patLen - 1 ; // 注目しているパターン内での位置
      while (text.charAt(i) == pattern.charAt(j)) {
        if (j == 0) {      // 最初の文字まで一致すれば探索成功
          return i ;
        }
        i-- ; j-- ;        // i, j を1文字分戻す
      }
      i = i + Math.max(skip[text.charAt(i)], patLen - j) ;
    }
    return -1 ;  // 結局, 見つからなかった
  }
}
```

23

23

BM法の原理(6)

- パターンの2文字目 b と
  テキストの d が不一致



- 不一致が発生した文字がパターンの注目点より末尾よりの場合，パターンを現在位置から1つずらす

```
i = i + Math.max(skip[text.charAt(i)], patLen - j) ;
```

24

24

BM法の性質(1)

- たいていの場合，パターンはm文字ずつ移動
  - 計算量：O(n/m)
- BM法の前提は文字の種類が十分に多いこと
  - パターンに含まれない文字が多いのでm文字移動できる
  - ビット列は文字が0か1のみなので，そのままでは難しい
  - パターンの長さが長い場合も効率が悪くなる
- 前処理：パターンをずらす量の表作成のコスト
  - Javaでは文字は2バイト（Unicode）なので65536要素の配列が必要

25

25

BM法の性質(2)

- 最悪のケース
  - テキストのすべての文字がa
  - パターンの最初の文字がbで残りがすべてa
  - この場合，O(mn)回の比較が必要
- 最悪ケースの回避
  - BM法には2つの戦略がある
  - 教科書では一方だけ紹介
  - もう一方の戦略はKMP法と同様に一致した部分の情報をもとにパターンを移動するというもの
  - これを利用すれば最悪ケースもO(n)になる
  - 実用上は効果があまりないため，教科書のもので十分

26

26

まとめ

- 文字列に対する探索
- 力まかせのアルゴリズム
- Knuth-Morris-Prattのアルゴリズム
- Boyer-Mooreのアルゴリズム

27

27

参考文献

- 定本 Javaプログラマのための
  アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ
  アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3
  アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造
  Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇
  オングス (著)・杉山 貴章・後藤 大地 (監修)

28

28