アルゴリズムとデータ構造

第6回　文字列探索とジェネリクス

第6回のキーワード

アルゴリズム関係

- 文字列探索（string search/matching）
- 力まかせ法（brute force）
- $O(n)$ ～ $O(nm)$
- ボイヤー・ムーア（BM）法（Boyer–Moore，Boyer–Moore-Horspool（簡易版））
- $O(n/m)$ ～ $O(nm)$
- 動的配列
- コレクション

Java関係

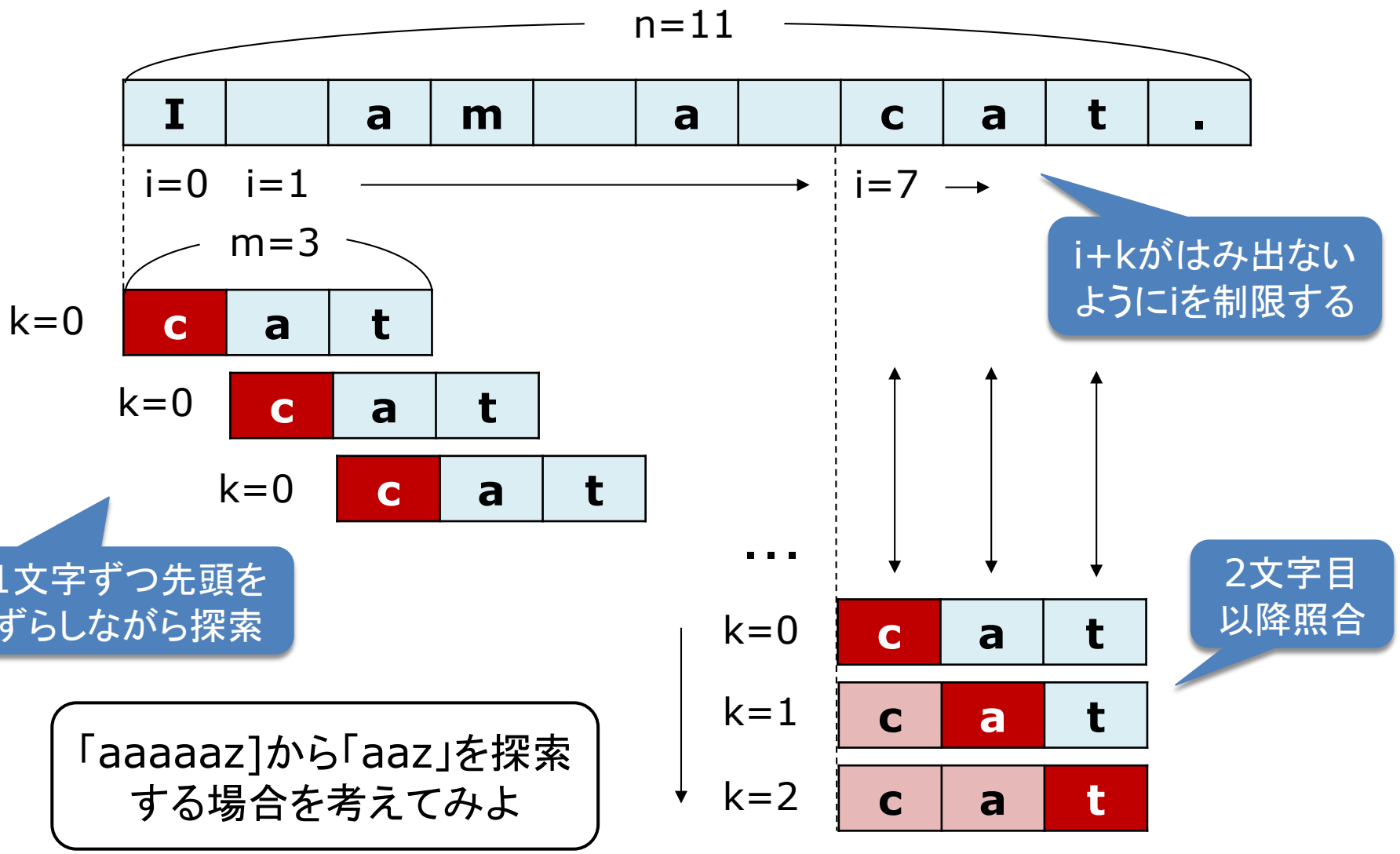
- charAt
- Arrays.sort
  Arrays.binarySearch
- ジェネリクス（総称型）
- 自然な順序
- Comparable<*E*>
- Comparator<*E*>
- ArrayList<*E*>
- ラッパークラス
- Collections.sort
  Collections.binarySearch

文字列探索

- 文字列探索とは
  - 前回までは，データ列から要素1個を探す問題を扱った
  - 今回は，文字「列」の中から，文字「列」を探す
  - 文字列だけでなく，DNA配列の探索などにも応用される

- 力まかせ法
  - 対象のテキストを n 文字，探索文字列を m 文字とする
  - まず i = 0 として，テキスト内の位置 i から位置 i + m まで，1文字ずつ順に探索文字列と照合する
  - もし，m 文字すべてが一致したら，位置 i で発見となる
  - そうでなければ，i を1だけ進めて同様の処理を繰り返す
  - ただし，テキストの残りが m 文字未満なら終了とする

文字列探索（力まかせ法）


n=11
I a m a c a t .
i=0 i=1
i=7
m=3
k=0 c a t
k=0 c a t
k=0 c a t
i+kがはみ出ないようにiを制限する
1文字ずつ先頭をずらしながら探索
…
k=0 c a t
k=1 c a t
k=2 c a t
2文字目以降照合
「aaaaaz]から「aaz」を探索する場合を考えてみよ

確認問題

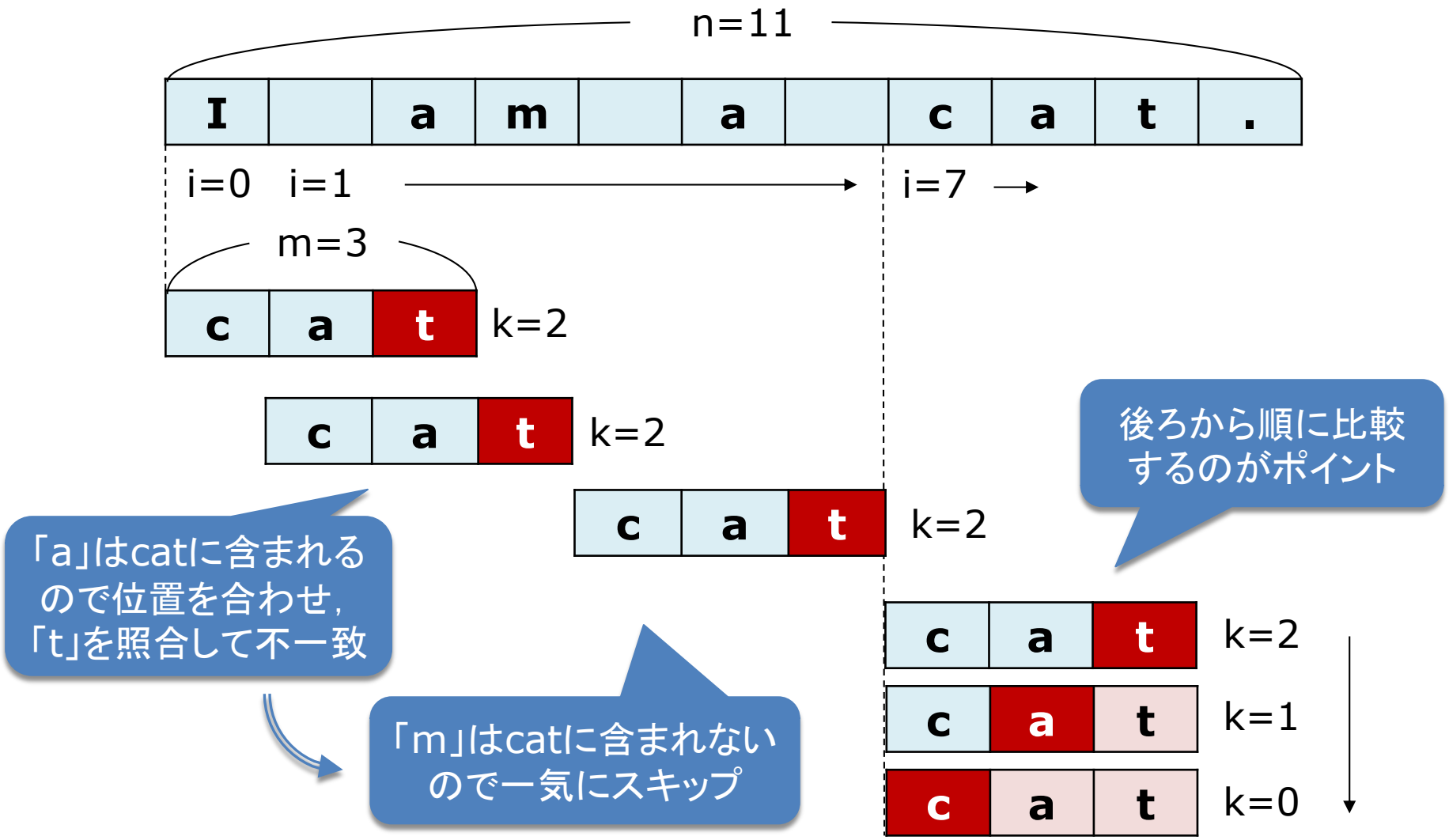
- 力まかせ法のプログラム
  - 下記のプログラムの一部の空欄を適切に埋めよ
  - 「aaaaaz]から「aaz」を探索する場合を考えてみよ

```
int n = text.length();
int m = key.length();

// textの中の位置iからkeyを探索する（残りがm文字以上なら続ける）
for (int i = 0; i <=                    ; i++) {

    // その位置から順に，keyの中の文字と比較する（kはkeyの中の位置）
    int k;
    for (k = 0; k <         ; k++) {
        if (text.charAt(          ) != key.charAt(          ))
            break;
    }
    if (k == m) return i;  // keyの全文字（m文字）が一致したら発見
}
```

ボイヤー・ムーア法（簡易版）

- 基本的なアイデア
  - 文字列の先頭から照合するよりも，末尾から照合した方が，探索位置を大きくスキップできる
- アルゴリズムの概要
  - 対象のテキストを n 文字，探索文字列を m 文字とする
  - まず i = 0 として，テキストの位置 i + m と探索文字列の末尾の位置 m から，1文字ずつ逆順に照合する
  - もし，m 文字すべてが一致したら，位置 i で発見となる
  - 一致しなかったら，テキストの位置 i + m の文字が探索文字列に含まれない場合は，i を m だけ進め，探索を続ける
  - 含まれる場合は，テキストの位置 i + m に探索文字列のその文字を合わせるように i を最小限だけ進め，探索を続ける

ボイヤー・ムーア法（簡易版）



文字列探索の計算量の概算

- 力まかせ法
  - 最善の場合：テキストの中に，探索文字列の先頭文字が1回以下しか含まれない ⇒ 線形探索と同じなので $O(n)$
  - 最悪の場合：i を進めるごとに，探索文字列の末尾直前まで照合する ⇒ 2重ループをほぼ全て回るので $O(nm)$

- ボイアー・ムーア法
  - 最善の場合：i を進めて文字を照合すると毎回不一致で，探索文字列全体を照合するのは1回だけ
    ⇒ n 文字の中で m 文字ずつスキップするので $O(n/m)$
  - 最悪の場合：i を進めるごとに，探索文字列の先頭直前まで照合する ⇒ 2重ループをほぼ全て回るので $O(nm)$

クラス型の配列(復習)

- クラス型の配列の作成
  - class Item { int code; String name; }
  - Item [] data = new Item[10];
  - for (int i = 0; i < data.length; i++) data[i] = new Item();
- 配列要素のメンバのアクセス
  - if (data[i].code == code)
- 配列要素の(位置の)交換
  - Item t; t = data[i]; data[i] = data[j]; data[j] = t;
  - Javaの配列の構造やクラス型変数の代入について再確認

確認問題

- Javaの配列とfor文
  - 下記のA)やB)のような記述はできるが，C)のような記述はできない理由を述べよ

A)
```
for (int i = 0; i < data.length; i++)
    data[i] = new Item();
```

B)
```
for (Item e : data)
    System.out.println(e.code);
```

C)
```
for (Item e : data)
    e = new Item();
```

- Javaの配列とコピー
  - クラスのインスタンスを要素とする配列 data について，下記のA)とB)の処理の違いを図解で説明せよ

A)
```
data[i] = data[j];
```

B)
```
data[i].code = data[j].code;
data[i].name = data[j].name;
```

中間試験の範囲はここまで

Javaによる探索とソート

- 配列の探索とソート
  - java.util.Arraysの静的メソッドが使用できる
  - 線形探索： Arrays.asList(*array*).indexOf(*key*)
    - ただし，これはクラスの配列でしかうまく動かない
    - intやdoubleなど基本型の配列では期待通りに動作しないので注意
  - 2分探索： Arrays.binarySearch(*array*, *key*)
  - ソート： Arrays.sort(*array*)

- Comparableインタフェース
  - 2分探索やソートでは，要素が比較できなければならない
  - そのためには，要素のクラスはComparableインタフェースを実装し，compareToメソッドを持つことが必要

確認問題

- 下記に示したクラス Item の定義を完成させよ
  - Itemのインスタンスは，品番で比較可能（整列可能）である

```
public class Item implements Comparable<Item> {
    public int code;      // 品番
    public String name;   // 品名

    public boolean equals(Object obj) {
        /* プロIIの教科書14.2.6を参考に実装 */
    }

    public int compareTo(Item item2) {
        /* codeの小さい順になるように実装 */
    }

    // equals定義時にはこれも値が等しくなるように定義が必要
    public int hashCode() { return this.code; }
}
```

動的配列とジェネリクス

- ArrayList<*E*>
  - 要素数を動的に変更できる配列（のようなクラス）
  - *E*に要素のクラス名を当てはめて使う（ジェネリクス）

    例）ArrayList<*String*> *alist* = new ArrayList<*String*>();
  - 要素の追加/取得/変更には，add/get/setメソッドを使う

    例）*alist*.add(*str*) / *alist*.get(*i*) / *alist*.set(*i*, *str*)

- ArrayListの探索とソート
  - java.util.Collectionsの静的メソッドが使用できる
  - 線形探索： *alist*.indexOf(*key*)
  - 2分探索： Collections.binarySearch(*alist*, *key*)
  - ソート： Collections.sort(*alist*)

ラッパークラス

- Javaの基本型（intやdouble）はクラスではない
  - 昔，Javaの開発時にパフォーマンスを重視したためだが…
  - Objectクラス継承していないなど，不便な場合がある
- 各基本型に対応する「ラッパークラス」が使える
  - int → Integer，double → Double，char → Character
  - 対応する基本型をフィールドに持つ（包む）特別なクラス
  - 例）Double x = new Double(1.4142); // doubleの値を包む
- 基本型と自動的に相互変換（autoboxing, unboxing）
  - 文法上，ほぼ基本型と同じように使え，四則演算等も可能
  - 例）Integer i = 10; i += 20; System.out.println(i);