アルゴリズム入門　＃13

地引　昌弘

2019.01.10

はじめに

今回は、アルゴリズム入門の最終回として、まずは、これまで学んで来たループ処理/再帰処理/動的計画法といったアルゴリズム技法が、問題をどのように見通し良くするのかを整理してみます。そして最後に、様々な技法を駆使しても効率的な解決が難しい例を取り上げ、現在のアルゴリズム分野が直面している難問を見ながら、この講義を終わることにします。

1　前回の演習問題の解説

1.1　演習 12-1: — 部屋割り問題の拡張

最初の問題は、「13 人部屋 33,000 円」「17 人部屋 40,000 円」という選択肢を追加して、最適な部屋割りおよび宿泊費を求めるというものでした。Python のプログラムだけ示しておきます:

```
import i2a

def room_2(n):
  room_price = i2a.array.make1d(n+18)    # 配列のサイズを変更することを忘れないように
  for i in range(len(room_price)): room_price[i] = 0
  room_sel = i2a.array.make1d(n+1)
  for i in range(len(room_sel)): room_sel[i] = 0

  # 過去の計算結果を利用し、人数が増える度にどの部屋割りが最安かを調べて行く
  for i in range(1, n+1):
    min = room_price[i-1] + 5000; s = 1
    if min > room_price[i-3] + 12000: min = room_price[i-3] + 12000; s = 3
    if min > room_price[i-7] + 20000: min = room_price[i-7] + 20000; s = 7
    if min > room_price[i-13] + 33000: min = room_price[i-13] + 33000; s = 13  # 13 人部屋
    if min > room_price[i-17] + 40000: min = room_price[i-17] + 40000; s = 17  # 17 人部屋
    room_price[i] = min
    room_sel[i] = s        # i 人が宿泊する時に、最後に選択した部屋 (の人数) を記録して行く

  # 最安の宿泊費および選択した部屋を逆向きに辿り、配列に入れて行く (整形した表示にするため)
  a = [room_price[n]]      # 初項は最安の宿泊費
  i = len(room_sel) - 1
  while i > 0:
    a.insert(len(a), room_sel[i])
    i = i - room_sel[i]    # トレースバック
  print(a)
```

1.2 演習 12-2: — 釣り銭問題

この問題は、部屋割り問題とほぼ同様の方法で解けます。但し、お釣りを「多め」に渡したら損してしまうので、選択肢を考える際は「残額がそのコインの額面以上」という条件を考慮する (要は追加する) 必要があります (部屋割り問題では「宿泊者は部屋の収容人数以下」でした/その違いに注意して下さい)。Python のプログラムは、下記の通り:

```
def tno_coins(n):
  coins = i2a.array.make1d(n+1)     # 部屋割り問題との違いに注意
  for i in range(len(coins)): coins[i] = 0
  c_used = i2a.array.make1d(n+1)
  for i in range(len(c_used)): c_used[i] = 0

  # 過去の計算結果を利用し、金額が増える度にどんなコイン選択が最小枚数かを調べて行く
  for i in range(1, n+1):
    min = coins[i-1]+1; s = 1
    if (i >= 5) and (coins[i-5]+1 < min): min = coins[i-5]+1; s = 5
    if (i >= 10) and (coins[i-10]+1 < min): min = coins[i-10]+1; s = 10
    if (i >= 25) and (coins[i-25]+1 < min): min = coins[i-25]+1; s = 25
    coins[i] = min
    c_used[i] = s         # i¢を返す時に、最後に選択したコイン (の金額) を記録して行く

  # コインの最小枚数および選択したコインを逆向きに辿り、配列に入れて行く (整形した表示にするため)
  a = [coins[n]]          # 初項は最安の宿泊費
  i = len(c_used) - 1
  while i > 0:
    a.insert(len(a), c_used[i])
    i = i - c_used[i]     # トレースバック (コインの額面分だけ、選んだコインのリストを巻き戻す)
  print(a)
```

配列 coins は、釣り銭として返すコインの枚数を保持しています (i¢になるコインの枚数が coins[i] に入ります)。配列 c_used には、トレース バック用に選んだコインのリストが記録されます (部屋割り問題のトレース バックと同様、金額が i¢になった時、どの額面のコインを追加したかが c_used[i] に記録されます)。例えば、c_used[i] に 25 が記録されていた場合は、25¢コインが追加された、つまり 25¢コインを 1 枚使うことを意味します。これにより、総額 i¢に対してまずは 25¢コインを 1 枚使うことが分かったので、次は $(i-25)$¢ではどの額面のコインが追加されたかを調べます。これが c_used[i - 25] に記録されています。もし、c_used[i - 25] = 1 であれば、1¢コインを 1 枚使うことを示しているので、その次は c_used[i - (25 + 1)] = c_used[i - 26] を調べます。以下順次、その金額を構成するために新たに追加されたコインを辿って行くわけです。ところで、トレース バックとは、記録したリストを逆順に辿りながら行なう処理の総称なので、全てのトレース バックが部屋割り問題や釣り銭問題と同じような処理をするわけではないことに留意して下さい (本日取り上げる LCS の計算と比較してみましょう)。

最後に、上でも述べましたが、釣り銭問題ではお釣りを「多め」に渡さないように、選択肢 (条件分岐) の条件として「残額がそのコインの額面以上」という条件を追加しています (条件分岐にある and の左側です)。これにより、i が小さい時でも、coins[-6] や coins[-14] などを参照するようなことはありません。配列 coins のサイズに注意して下さい。一応確認のため、実行例を挙げておきます:

```
>>> tno_coins(80)
[4, 5, 25, 25, 25]
>>> tno_coins(115)
[6, 5, 10, 25, 25, 25, 25]
>>> tno_coins(40)
[3, 5, 10, 25]
```

1.3 演習 12-3: — 経路数算出問題の拡張

これは、左上端 (S) から、「右/下」の 2 方向へ移動しながら右下端 (G) へ至る経路数を求める問題に対し、移動方向に「斜め右下」を追加して拡張せよ、という問題でした。障害のない場合およびある場合のそれぞれにつき、Python のプログラムだけ示しておきます:

```
import pprint

# 障害がない場合
def paths1a():
  size_i = 6; size_j = 8
  a = i2a.array.make2d(size_i, size_j)
  for i in range(size_i): a[i][0] = 1
  for j in range(size_j): a[0][j] = 1

  # 上セルおよび左セルに至る経路数を用いて、各セルに至る経路数を調べて行く
  for i in range(1, size_i):
    for j in range(1, size_j):
      a[i][j] = a[i-1][j] + a[i][j-1] + a[i-1][j-1]
  pprint.pprint(a, width=50)
  return(a[size_i-1][size_j-1])

# 障害がある場合
def paths2a():
  size_i = 6; size_j = 8
  a = [[0 for j in range(size_j)] for i in range(size_i)]
  a[0][4] = a[2][2] = a[3][5] = a[4][1] = -1      # まず最初に障害物のあるセルへ -1 を入れる
  for i in range(size_i):
    if a[i][0] < 0: break                         # 障害物があったら、それより下へは行けない
    else: a[i][0] = 1
  for j in range(size_j):
    if a[0][j] < 0: break                         # 障害物があったら、それより右へは行けない
    else: a[0][j] = 1

  # 上/左/左上セルに至る経路数を用いて、各セルに至る経路数を調べて行く
  for i in range(1, size_i):
    for j in range(1, size_j):
      if a[i][j] < 0: continue                    # 障害物のセルは何もしない
      if a[i-1][j] > 0: a[i][j] += a[i-1][j]      # 上セルが障害物かを調べてから経路数を追加
      if a[i][j-1] > 0: a[i][j] += a[i][j-1]      # 左セルが障害物かを調べてから経路数を追加
      if a[i-1][j-1] > 0: a[i][j] += a[i-1][j-1]  # 左上セルが障害物かを調べてから経路数を追加
  pprint.pprint(a, width=50)
  return(a[size_i-1][size_j-1])
```

どちらのプログラムも、斜め右下へ移動する経路数 (`a[i-1][j-1]`) を追加して行くだけですね。

2　パターン認識

以下では、パターン認識の問題を題材に、ループ処理・再帰処理・動的計画法の適用について、改めて整理してみましょう。

2.1　パターン認識とは

パターンという単語は様々な状況/場面で使われますが、ここで取り上げるパターンは、規則性や体系などを意味しています。**パターン認識** (pattern recognition) とは、様々なデータの中にあるパターンを同定することを言い、次のものが代表的です[1]:

- 音声認識 — 「音」から「何を喋っているか」を抽出
- 画像認識 — 「画像」から「誰の写真か」「どこに何があるか」などを抽出
- 文字認識 — 「文字の画像」や「書いている様子のデータ」から「何が書かれているか」を抽出[2]

特に、パターン認識でいうところのパターンは、一つのデータではなく、似通ったデータの集まりという意味を含んでいます。従って、パターンへの「当てはまり」も、YES/NO ではなく、「どれくらい似ているか」を判断することが普通です。パターン認識が難しい理由として、次のようなものが挙げられます:

- あるパターンに属するデータに大きな多様性がある。例えば、「あ」という文字でも、実に様々な書き方が存在する。
- パターン認識に用いるデータにはノイズが含まれていることがある。例えば、紙に汚れが付いた状態で文字を読み取る場合など。
- パターンをどのような構造 (or 関係) で捉えたら良いかの判断が難しい。例えば、音声認識であれば「音素」→「音節」→「単語」→「文」のような構造があり、各段階の認識では別の段階の認識が参照されます。「今日の天気は晴れですね。」という音声においてその一部が欠落し、「きょうの・てんきは・□れ・ですね。」という音声を認識することになったとしても、話題の範囲が「天気」だと認識されていれば、「□れ」は「はれ」という音節だと認識出来るわけです。

人間がパターン認識を得意とするのは、“長期間+様々な事象” の**学習** (learning) により、多数の神経細胞が繋がった複雑な回路を構成しているからとされています[3]。そこで、これを真似して、人間の神経回路のような構造をプログラム上のデータ構造として構築し、それに学習を行わせてパターンを認識させる**ニューラル ネットワーク** (neural networks) などの研究が多く行われています。今回は学習などではなく、アルゴリズムによる類似度の計算に取り組んでみます。

2.2　文字列の対応付け

「どれくらい似ているか」を調べる例として、二つの文字列を対応付ける問題を考えてみましょう。例えば、「isasaka」「sassa」「wakasa」の 3 文字列のうち、互いに一番似ているのはどの二つだと思いますか。「似ている」の定義にもよりますが、今回は、「二つの文字列において “同じ文字が対応する” 組が多い」方が似ていると考えることにします (図 1: 次ページ)。但し、ここで言う “同じ文字が対応する” とは、(文字が同じだけではなく) 対応付けの線がクロスしてはいけないとします。この定義より、「isasaka」「sassa」の組が一番多く (4 箇所で) 対応付け出来ると分かります[4]。この問題は、「**最長共通部分列** (longest common subsequence, LCS)」と呼ばれています。これは、両方の文字列から順序を変えずに (間は飛ばしてもよい) 一部を抜き出した二つの部分列に対し、互いに一致する最長のものを求めているからです。

---

[1] パターン認識の中には、人間にとっては簡単なものの、プログラムに認識させることは難しい対象が多数あります。これらはその代表例でもあります。

[2] 「書いている様子のデータ」のことを**ストローク データ** (stroke data) と呼びます。これは書き順や書く速さなども含んだデータなので、単なる書かれた画像よりも多くの情報が含まれています。

[3] 回路の様々な構造が、それぞれ情報を蓄積していると見ることも出来ます

[4] 「sassa」「wakasa」の対応付けは、図 1 以外にも考えることが出来ますが、組の数は変わりません。「isasaka」と「wakasa」も同様です。

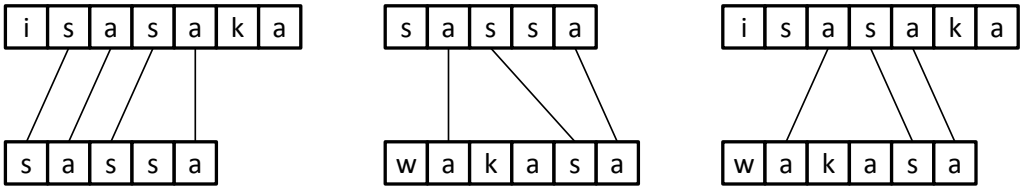

図 1: 最長共通部分列

では、最長共通部分列をどのように求めたらよいでしょうか。まずは、図 2 左のように、比較する二つの文字列それぞれに「カーソル」を対応させてみましょう。カーソルは、二つの文字列において対応付ける文字の位置を表します。上段 (or 下段) の文字列で対応付ける文字を一つ右へずらす場合は、カーソル a (or カーソル b) を一つ右へ進めます。

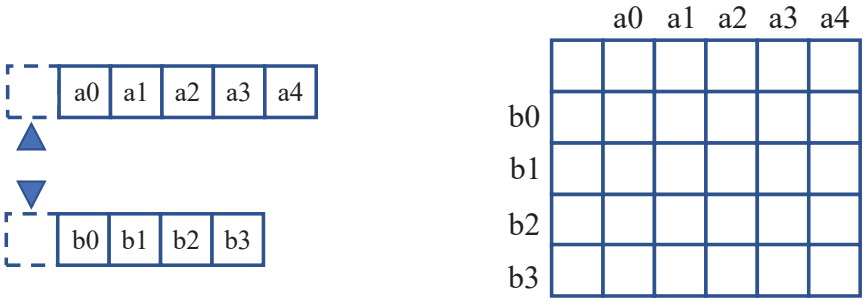

図 2: 文字列の対応付けの考え方

以下では、この二つのカーソルを用いて、LCS の値を計算するアルゴリズムを考えてみます。まずは、上段の文字列より 1 文字を左から順に取り出し、下段の文字列と比較して行く (考え方としては素直な) ループ処理を試してみましょう。その疑似コードは以下のようになります:

- lcs1(s, t) — 文字列 `s, t` の LCS を求める
- 共通部分列の最大長を記録する変数 `c-max` を用意 (初期値は 0)
- 文字列 `s` の対応付けを始める場所を `i` とし、`i` が文字列 `s` 内にある間、以下を繰り返し
- 文字列 `t` の対応付けを始める場所の一つ前を `lm` (初期値は -1) とする
- カーソル a の位置を示す変数 `j` (初期値は `i`)、共通部分列の長さを示す変数 `c` (初期値は 0) を用意
- カーソル a が文字列 `s` 内にある or `lm` が文字列 `t` の最後にない間、以下を繰り返し
- カーソル b の位置を示す変数 `k` (初期値は `lm+1`) を用意
- カーソル b が文字列 `t` 内にある間、以下を繰り返し
- もし、カーソル a の文字とカーソル b の文字が等しければ、
- `lm` (文字列 `t` の対応付けを始める場所の一つ前) を `k` に変える
- 文字列 `s,t` のカーソル a, b を一つずつ進め、共通部分列の長さを 1 増やす
- カーソル a が文字列 `s` の外に出たら、繰り返しを抜ける
- そうでなければ、
- 文字列 `t` のカーソル b を一つ進める
- (枝分かれ終わり)
- (繰り返し終わり)
- 文字列 `s` のカーソル a を一つ進める
- (繰り返し終わり)
- `c` が過去の最大値 `c-max` より大きければ、こちらを記録
- (繰り返し終わり)　　# 文字列 `s` の対応付けを始める場所を一つずらす
- `c-max` を表示

対応する Python のプログラムは、こんな感じ:

```
def lcs1(s, t):
  c_max = 0
  for i in range(len(s)):
    c = 0; lm = -1; j = i

    # この while ループは、上段の文字列 s のカーソル a を進めて行く処理
    while (j <= len(s)-1) and (lm <= len(t)-2):
      k = lm + 1

      # この while ループは、下段の文字列 t のカーソル b を進めて行く処理
      while k <= len(t)-1:
        if s[j] == t[k]:
          lm = k
          j += 1; k += 1; c += 1;
          if j >= len(s):
            break
        else:
          k += 1
      j += 1
    if c > c_max:
      c_max = c
  return(c_max)
```

カーソルを用いた LCS の計算では、片方の文字列のカーソルが右端に到達した後は、残った文字列のカーソルだけを進める (比較して行く) という処理が必要になります。常に上段 (or 下段) の文字列の方が先に右端に到達すると決まっていれば、計数ループを用いて簡潔に扱えるのですが、どちらが先に右端に到達するかは動かしてみないと分かりません (渡された文字列に応じて決まります)。よって、上のコードのように複数の `while` 文を用意し、カーソルの進み具合に応じた終了条件を、それぞれで個別に制御する必要があります。

このように、複数の `while` 文を組み合わせることで、一見 LCS をうまく計算出来ているように見えますが、残念ながらカーソル法は、二つの文字列を対称に扱えているわけではありません。この例で言えば、上段の文字列 s のカーソル a を優先して進めています。例えば “上段: powercomhumble”, “下段: powerhumblecom” を渡された場合、上段では com の部分が先に調べられ、その後 humble の部分を調べる時には、下段にある humble の部分が既に調査済みとして扱われ (要はカーソルが進んでしまっている)、調べられることはありません。実際に試してみると:

```
>>> lcs1("powercomhumble", "powerhumblecom")
8
>>>
```

となり、正しい結果は得られていません。上段と下段を入れ替えて再実行すれば、この 2 回の実行を通して二つの文字列を対称に扱えたと言えますが、あまり嬉しくありません。一応、上段と下段を入れ替えた実行結果もお見せしておきます (これが正しい結果になります):

```
>>> lcs1("powerhumblecom", "powercomhumble")
11
>>>
```

カーソル法は、考え方としては素直でしたが、プログラムは複雑になってしまい、また完成度の高いアルゴリズムとは言えない面がありました。他に、より見通しの良い考え方はないでしょうか。

これまで見てきたように、もし再帰的な手続きを定義することが出来れば、アルゴリズムをシンプルに記述することが出来ます。まずは、二つの文字列を “$a : a_0 \cdots a_{i-1}a_i$” および “$b : b_0 \cdots b_{j-1}b_j$” とします。この時、LCS の計算を再帰的に考えると、下記の値のうち最も大きいものが全体の LCS 値になります (部屋割り問題の考え方に少し似ていますね):

- 二つの文字列 a, b において、最後の文字が等しい場合は、その両方を削除した文字列同士の LCS 値より 1 だけ多い値
- 等しくない場合は、a から最後の 1 文字 a_i を除いた場合の LCS 値
- 同、b から最後の 1 文字 b_j を除いた場合の LCS 値

形式的な再帰的定義は下記のようになります:

$$lcs(a_0 \cdots a_i,\ b_0 \cdots b_j) = max \begin{cases} lcs(a_0 \cdots a_{i-1},\ b_0 \cdots b_{j-1}) + 1 & (a_i == b_j) \\ lcs(a_0 \cdots a_i,\ b_0 \cdots b_{j-1}) & (a_i \neq b_j) \\ lcs(a_0 \cdots a_{i-1},\ b_0 \cdots b_j) & (a_i \neq b_j) \end{cases}$$

この再帰的定義をそのままコードにすることで、簡潔にプログラムを作成出来ます。では、その計算量はどうでしょうか。上の再帰的定義を見る限り、毎回三つの再帰呼び出しが発生する一方で、各再帰呼び出しにおける処理が減っているようには見えません (毎回、3 回の比較と状況に応じて +1 の計算)。

しかし、再帰的な手続きを定義することが出来れば、動的計画法による効率的なプログラムを作ることが出来ます。まずは、上段の文字列を横軸とし、下段の文字列を縦軸とした図 2 右のような表を考えます[5]。上段の文字列において比較する文字を一つ進めることは、この表で右セルへの移動と対応し、下段の文字列を進めることは下セルへの移動と対応します。また、比較する文字が一致し、両者で文字を一つ進めることは、斜め右下への移動に対応します。ここで、セル $(i,\ j)$ の値を文字列 “$a_0 \cdots a_{i-1}a_i$” と “$b_0 \cdots b_{j-1}b_j$” との LCS 値とすれば、上で定義した再帰的手続きを、図 3 左のようなセルの位置関係と対応した動的計画に落とし込むことが出来るわけです。

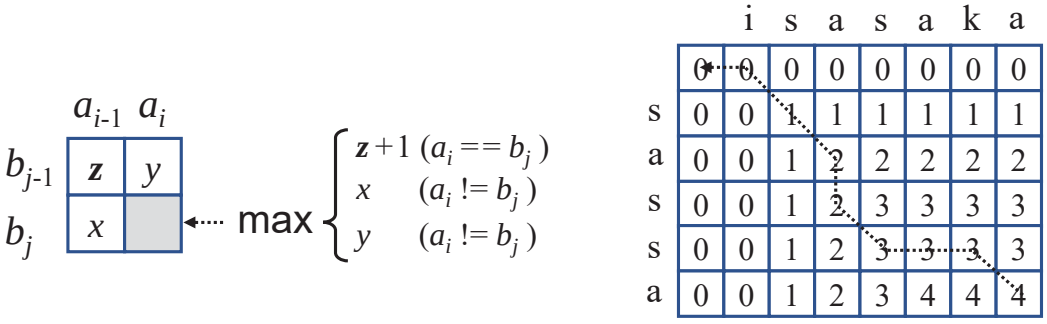

図 3: 動的計画法による LCS の計算

では、これを Python のプログラムにしたものを次に示します。毎度のことですが、2 次元配列 `a[i][j]` は、i 行/j 列のセルに対応することに注意しましょう (座標系とは違います/混乱し易いので、以降のプログラムでもしっかりと確認して下さい)。x, y, z+1 の最大値を保持する max は、図 3 左の x, y, z+1 に想定される最低値より小さい値を初期値としているので、三つの条件分岐により、必ず x, y, z+1 の最大値が入ります。

[5] この表について少し補足をしておきます。アルゴリズムの説明としては、両文字列 $(a_x,\ b_y)$ を上詰め/左詰めした方が分かりやすいですよね。しかし、配列を用いた動的計画法のプログラムでは、既に見てきたように添字番号が負になる場合があります。以後の図/プログラムでは、これを避けるために最上行および最左列を空けてあるのです。

```
def lcs2(s, t):
  a = [[0 for j in range(len(s)+1)] for i in range(len(t)+1)]
  for i in range(1, len(t)+1):
    for j in range(1, len(s)+1):
      max = -1
      if s[j-1] == t[i-1]: max = a[i-1][j-1] + 1
      if a[i][j-1] > max: max = a[i][j-1]
      if a[i-1][j] > max: max = a[i-1][j]
      a[i][j] = max
  pprint.pprint(a, width=50)
  return(a[len(t)][len(s)])
```

実行例は下記の通り。各セルの値 (LCS 値) は、上辺/左辺にある文字列の (先頭文字を含む) 部分列同士を比較した LCS 値に該当します。例えば、“isasaka” のうち “isas” までと、“sassa” のうち “sass” までを比較した LCS 値は 3 になります。図 3 右 (前ページ) にある 3 行 4 列のセルが、それに該当します (行と列は 0 から始まることに注意)。また、上端/左端のセルは、動的計画の手順として右下方向への移動によりそのセルへ至ることはないので、全て 0 になります (文字列比較における意味については、10 ページを参照)。下記の結果と、図 3 右を見比べてみて下さい (同図にある点線矢印は、LCS 値が最大となる対応付けを求めるトレース バックを示しています/詳細は後で):

```
>>> lcs2("isasaka", "sassa")
[[0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 1, 1, 1, 1, 1],
 [0, 0, 1, 2, 2, 2, 2, 2],
 [0, 0, 1, 2, 3, 3, 3, 3],
 [0, 0, 1, 2, 3, 3, 3, 3],
 [0, 0, 1, 2, 3, 4, 4, 4]]
4
>>>
```

ところで、上で説明した LCS 値を計算するプログラムでは、具体的な最長部分文字列を求めていません。これを求めるには、部屋割り問題や釣り銭問題と同様、トレース バックを行なう必要があります。部屋割り問題や釣り銭問題では、動的計画を進める際、トレース バックに必要な情報をその都度記録して行きましたが、LCS の計算では、各セルに LCS 値が記録されているので、これを利用出来ます。部屋割り問題では選択した部屋に応じて、また釣り銭問題では選択したコインに応じて、トレース バックをしました。LCS の計算では、各セルの LCS 値を計算する際に、図 3 左にあるどの式が適用されたかを調べて、トレース バックをします。トレース バックを行なう Python プログラムは以下の通り (各コードの意味については、プログラム内のコメントを参考にして下さい):

```
def lcstrace(s, t)

  # まずは、動的計画法により LCS 値の表を作成する (この部分は lcs2 関数と全く同じ)。
  a = [[0 for j in range(len(s)+1)] for i in range(len(t)+1)]
  for i in range(1, len(t)+1):
    for j in range(1, len(s)+1):
      max = -1
      if s[j-1] == t[i-1]: max = a[i-1][j-1] + 1
      if a[i][j-1] > max: max = a[i][j-1]
      if a[i-1][j] > max: max = a[i-1][j]
      a[i][j] = max
```

```
  # 作成したLCS値の表の右下端から、
  # 各セル毎にどの規則を適用したかを調べながら「逆向きに」辿って行く。
  # 終点セルより逆向きに辿ってc番目となるセルの、
  # 行番号iはord[c]に、列番号jはabs[c]に記録する。
  ord = [0 for k in range(len(t)*len(s))]; abs = [0 for k in range(len(t)*len(s))]
  i = len(t); j = len(s); c = 0
  ord[c] = i; abs[c] = j; c += 1
  while i >= 1 or j >= 1:    # 0行0列に辿り着くまでトレースバックを繰り返す。
    if (j-1 >= 0) and (i-1 >= 0) and (s[j-1] == t[i-1]):  # 添字番号の範囲に注意
      ord[c] = i-1; abs[c] = j-1; c += 1
      i -= 1; j -= 1
    elif (j-1 >= 0) and (a[i][j] == a[i][j-1]):           # 添字番号の範囲に注意
      ord[c] = i; abs[c] = j-1; c += 1
      j -= 1
    elif (i-1 >= 0) and (a[i][j] == a[i-1][j]):           # 添字番号の範囲に注意
      ord[c] = i-1; abs[c] = j; c += 1
      i -= 1

  # トレースバックした結果(辿ったセルの位置)は、"ord[]行abs[]列"として表せる。
  # 表示の詳細な意味は、本文を参照のこと。
  for k in range(c-1):
    print("[%d][%d] -> " % ((ord[k]), (abs[k])), end="")
  print("[%d][%d]" % ((ord[c-1]), (abs[c-1])))

  # 以下は参考: 最長部分文字列を見易い形に整形して表示する。
  ns = [' ' for k in range(len(s))]    # ns=文字列sで対応する文字の番号を記録
  nt = [' ' for k in range(len(t))]    # nt=文字列tで対応する文字の番号を記録
  d = 0                                # 対応する文字のカウンタ
  for k in range(c-1, 0, -1):
    if (ord[k]-ord[k-1] == -1) and (abs[k]-abs[k-1] == -1):
      d += 1
      ns[abs[k]] = str(d); nt[ord[k]] = str(d)
  ns = "".join(ns); nt = "".join(nt)   # 文字の配列を文字列に変更
  print(s); print(ns); print(nt); print(t)
```

トレース バックをする際は、作成したLCS値の表を突き抜けて辿らないように、添字番号の範囲を、1セル辿る毎に調べることを忘れないようにしましょう。上のプログラムでは、空白文字列ns, ntに対し、トレース バックにより判明した経路 “ord[]行abs[]列” から、両文字列s, tで等しい文字の位置と同じ位置に、それが何番目の対応であるかを入れた表示を作っています(上下の文字列で同じ数字の文字が対応しています)。実行結果を下記に示します。

```
>>> lcstrace("isasaka", "sassa")
[5][7] -> [4][6] -> [4][5] -> [4][4] -> [3][3] -> [2][3] -> [1][2] -> [0][1] -> [0][0]
isasaka
 123  4
12 34
sassa
>>>
```

トレース バックによる表示の意味は次の通りです。

- [i] [j] とは、文字列 s の i 文字目と文字列 t の j 文字目を比較することを意味する。
- [i'] [j'] → [i] [j] とは、上で比較した結果に応じて、次は文字列 s の i' 文字目と文字列 t の j' 文字目を比較することを意味する。
- $i = 0$ ($j \neq 0$) の場合、文字列 s は先頭から j 文字を飛ばして (つまり j+1 文字目から) 比較を始めたことを意味する[6]($j = 0$ の場合は、文字列 t の先頭を飛ばす)。

また、最長部分文字列は、この限りではない (つまり、トレース バックの結果も、図 3 右 (7 ページ) に示す点線矢印だけではない) ことに注意して下さい。その理由は、図 3 左の規則において、例えば $a_i \neq b_i$ が成り立ちかつ x == y であれば、“b_j行 a_{i-1}列のセル” あるいは “b_{j-1}行 a_i列のセル” のどちらを選んでもよいからです。今回示したトレース バックのプログラムは、図 3 左の規則において、毎回 y に該当するセルを選択していますが、x に該当するセルを選択することも当然正しい動作であり、このような場合はトレース バックの結果も異なります。同様な例として、図 3 左の z+1 と x (あるいは y) の値が偶然一致する場合が存在します (図 3 右にある 5 行 6 列のセルが該当します)。この場合、前者であれば「対応する文字が等しいためにそのセルへ至った」となり、後者であれば「文字が異なるので片方を 1 文字飛ばしてそのセルに至った」となるので、当然両者の経路は異なりますが、どちらも正しい経路です。今回示したトレース バックのプログラムは、常に前者 (a_i == b_i の場合) を最初に調べていますので、後者のような経路は出現しません。

2.3 参考: 遺伝子のアラインメント

皆さんは、人間を始めとする地球上の生物が持つ遺伝子情報は、DNA 中にある 4 種類の塩基「A/T/G/C」の並びで表現されることを知っていますね。遺伝子に関する研究では、「二つの遺伝子がどれくらい似ているか」や「どことどこが対応付けられるか」を調べる必要があります。このような対応付けのことを**アラインメント** (alignment) と呼びます。アラインメントの計算では、先に述べた LCS の計算と少し異なり、点数に基づいた対応を計算します。今回は例として、次のような点数化の規則を考えてみましょう。この合計が対応付けの点数になります。

- 対応する塩基が一致している場合: +2 点
- 対応する塩基が不一致の場合: −1 点
- 対応する塩基が片方欠落している場合: −2 点
- 対応する塩基の片方が入れ替わっている場合[7]: +1 点

ここで、LCS とアライメントの違いについて、少し整理しておきます。LCS では、文字列の端から順に文字を比較し、両者が異なる場合は不一致と判断して、片方の文字列だけ 1 文字ずらしました。アラインメントでは、両者の塩基が異なる場合は常に不一致と判断するのではなく、片方の遺伝子にその塩基が欠落している or 片方の遺伝子に塩基の入れ替わりがある可能性も考慮します。片方の遺伝子にその塩基が欠落している場合は、不一致部分に空白 (ギャップ) を挿入したという扱いで、片方の塩基だけ一つずらします (これは LCS と同じ処理になりますね)。図 4 に、「GTACGACG」と「GATCCAG」を対応付けた例を示します。ここでは、入れ替えが 1 回発生し、さらに後者に一つの空白を挿入して点数を計算しています。

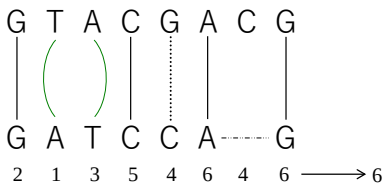

図 4: 遺伝子のアラインメントの点数計算

[6] これは、文字列 s の先頭から j 文字目までが文字列 t の先頭文字と等しくないことを表しています。
[7] 例えば、遺伝子 AT と遺伝子 TA では、そのままだと該当する塩基同士は不一致ですが、片方を入れ替えることでどちらも AT (あるいは TA) となり、一致するということです。

アライメントの計算では、様々な場所に空白を挿入しながら、あるいは前後を入れ替えながら、最終的に点数が最大となる対応付けを求めます。比較する塩基を示すカーソルを用意し、これをずらしながら 1 塩基ずつ比較するという考え方は素直ですが、LCS の計算と同様な理由から、そのプログラムは複雑そうな予感がします。そこで、これも同様に、“再帰的定義 → 動的計画法” の考え方に沿ってプログラムを作成してみましょう。まずは、二つの遺伝子を “$a: a_0 \cdots a_{i-1}a_i$” および “$b: b_0 \cdots b_{j-1}b_j$” とします。この時、アライメントの計算を再帰的に考えると、下記の値のうち最も大きいものが最良のアライメントになります:

- 二つの遺伝子 a, b において最後の塩基が等しい場合は、その両方を削除した遺伝子同士のアライメントより 2 大きい値
- 等しくない場合は、“a から最後の 1 塩基 a_i を除く” + “b から最後の 1 塩基 b_j を除く” 場合より 1 小さい値
- 同、“a から最後の 1 塩基 a_i を除く” + “b はそのまま (ギャップを入れるという意味)” 場合より 2 小さい値
- 同、“a はそのまま (ギャップを入れるという意味)” + “b から最後の 1 塩基 b_j を除く” 場合より 2 小さい値
- 同、その手前の塩基と交換したら等しくなる場合は、“a から最後の 2 塩基 a_ia_{i-1} を除く” + “b から最後の 2 塩基 b_jb_{j-1} を除く” 場合より 1 大きい値

形式的な再帰的定義は下記のようになります:

$$aln(a_0 \cdots a_i,\ b_0 \cdots b_j) = max \begin{cases} aln(a_0 \cdots a_{i-1}, b_0 \cdots b_{j-1}) + 2 & (a_i == b_j) \\ aln(a_0 \cdots a_{i-1}, b_0 \cdots b_{j-1}) - 1 & (a_i \neq b_j) \\ aln(a_0 \cdots a_{i-1}, b_0 \cdots b_j) - 2 & (a_i \neq b_j) \\ aln(a_0 \cdots a_i, b_0 \cdots b_{j-1}) - 2 & (a_i \neq b_j) \\ aln(a_0 \cdots a_{i-2}, b_0 \cdots b_{j-2}) + 1 & (a_i \neq b_j,\ a_i == b_{j-1},\ a_{i-1} == b_j) \end{cases}$$

図 3 左と同様なマスの位置関係に対応したアライメント計算の動的計画は、図 5 左のようになります。

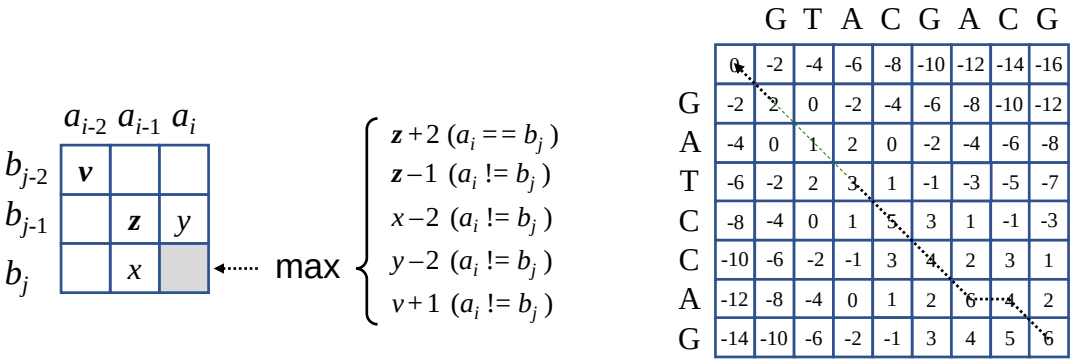

図 5: 遺伝子のアラインメントの動的計画法 (トレース バックの緑線は入れ替えが生じている部分)

トレース バックを行なう Python プログラムを次ページに示します (各コードの意味については、プログラム内のコメントを参考にして下さい)。特に、入れ替えが発生しているかどうかを調べる条件式の作り方に注意しましょう (アライアンス表の存在しないセルを参照しないように)。Python では、条件式が長いため途中で改行したい場合は、“\” (バックスラッシュ) を入れてから改行します。また、トレース バックでは、入れ替えが発生した場合にセルを一つ飛ばすことになるので、塩基の対応を見易い形に整形 (ギャップの挿入+入れ替え) して表示する場合は、特別な対応が必要になります:

```
def alntrace(s, t):

  # まずは、動的計画法によりアライアンスの表を作成する。
  a = [[-2*(i+j) for j in range(len(s)+1)] for i in range(len(t)+1)]
  for i in range(1, len(t)+1):
    for j in range(1, len(s)+1):
      max = a[i-1][j-1] + 2;
      if t[i-1] != s[j-1]: max -= 3
      if (i > 1 and j > 1) and (t[i-2] == s[j-1] and t[i-1] == s[j-2]) \
         and (a[i-2][j-2] + 1 > max):    # 入れ替えの有無を調べる（条件式の作り方に注意）
        max = a[i-2][j-2] + 1
      if a[i][j-1] - 2 > max: max = a[i][j-1] - 2
      if a[i-1][j] - 2 > max: max = a[i-1][j] - 2
      a[i][j] = max

  # 作成したアライアンスの表の右下端から、
  # 各セル毎にどの規則を適用したかを調べながら「逆向きに」辿って行く。
  # 終点セルより逆向きに辿って c 番目となるセルの、
  # 行番号 i は ord[c] に、列番号 j は abs[c] に記録する。
  # 入れ替えが起きた場合は、セルを一つ飛ばして記録するので、
  # その回数を d に記録しておく（後で、塩基の対応を整形して表示する際に使用する）。
  ord = [0 for k in range(len(t)*len(s))]; abs = [0 for k in range(len(t)*len(s))]
  i = len(t); j = len(s); c = 0; d = 0
  ord[c] = i; abs[c] = j; c += 1
  while i >= 1 or j >= 1:
    if (j-1 >= 0) and (i-1 >= 0) and (s[j-1] == t[i-1]):
      ord[c] = i-1; abs[c] = j-1; c += 1
      i -= 1; j -= 1
    elif (j-1 >= 0) and (i-1 >= 0) and (a[i][j] == a[i-1][j-1]-1):
      ord[c] = i-1; abs[c] = j-1; c += 1
      i -= 1; j -= 1
    elif (j-1 >= 0) and (a[i][j] == a[i][j-1]-2):
      ord[c] = i; abs[c] = j-1; c += 1
      j -= 1
    elif (i-1 >= 0) and (a[i][j] == a[i-1][j]- 2):
      ord[c] = i-1; abs[c] = j; c += 1
      i -= 1
    elif (j-2 >= 0 and i-2 >= 0) and (t[i-2] == s[j-1] and t[i-1] == s[j-2]) \
         and (a[i][j] == a[i-2][j-2] + 1):
      ord[c] = i-2; abs[c] = j-2; c += 1; d +=1    # 入れ替えが起きた回数を記録
      i -= 2; j -= 2
# pprint.pprint(a, width=100)                      # アライアンス表の表示
  print(a[len(t)][len(s)])

  # トレースバックした結果（辿ったセルの位置）は（ord[] 行, abs[] 列）として表せる。
  # 結果の意味: [i][j] では、文字列 s の i+1 文字目と文字列 t の j+1 文字目を比較し、
  for k in range(c-1):
    print("[%d][%d] -> " % ((ord[k]), (abs[k])), end="")
  print("[%d][%d]" % ((ord[c-1]), (abs[c-1])))
```

```
  # 塩基の対応を見易い形に整形 (ギャップの挿入+入れ替え) して表示する。
  if len(s) > len(t): l = len(s)
  else: l = len(t)
  ns = [' ' for k in range(2*l)]  # ns: 整形した遺伝子 s
  nt = [' ' for k in range(2*l)]  # nt: 整形した遺伝子 t

  # 入れ替えが起きた場合は、セルを一つ飛ばして記録するので、
  #   整形した遺伝子の塩基数 (b) = トレース バックで辿ったセル数 + 入れ替え数
  # となる。
  b = (c-1) + d; h = b
  for k in range(c-1, 0, -1):
    if (ord[k]-ord[k-1] == -1) and (abs[k]-abs[k-1] == -1):
      ns[h] = s[abs[k]]; nt[h] = t[ord[k]]
    elif (ord[k]-ord[k-1] == -2) and (abs[k]-abs[k-1] == -2):
      ns[h] = s[abs[k]]; nt[h] = t[ord[k]]
      ns[h-1] = s[abs[k]+1]; nt[h-1] = t[ord[k]+1]
      h = h - 1
    elif (ord[k]-ord[k-1] == 0) and (abs[k]-abs[k-1] == -1):
      ns[h] = s[abs[k]]; nt[h] = ' '
    else:
      ns[h] = ' '; nt[h] = t[ord[k]]
    h = h - 1

  # 最初の段に、ギャップを入れて整形した遺伝子 ns を表示
  for k in range(b, 0, -1): print("%c" % (ns[k]), end="")
  print("")

  # 次の段に、(整形済み) 両遺伝子の対応する塩基に応じた記号を表示 (一度、配列 r にしまってから表示)
  r = []
  for k in range(b, 0, -1):
    if ns[k] == nt[k]: r.insert(len(r), '|')
    elif (ns[k] == ' ') or (nt[k] == ' '): r.insert(len(r), ' ')
    else:
      if (ns[k] == nt[k+1]) and (ns[k+1] == nt[k]):
        r[len(r)-1] = '('; r.insert(len(r), ')')
      else:
        r.insert(len(r), ':')
  r = "".join(r); print(r)

  # 最後の段に、ギャップを入れて整形した遺伝子 nt を表示
  for k in range(b, 0, -1): print("%c" % (nt[k]), end="")
```

実行結果は、下記の通り:

```
>>> alntrace("GTACGACG", "GATCCAG")
6
[7][8] -> [6][7] -> [6][6] -> [5][5] -> [4][4] -> [3][3] -> [1][1] -> [0][0]
GTACGACG
|()|:| |
GATCCA G
>>>
```

さて、最後に動的計画法の制約について少し述べておきます。動的計画法には様々な手法があり、色々なややこしい問題を解くことが出来ます。その一方で、動的計画法は、まず問題をより小さな問題の組み合わせで解く形に定式化出来る必要があります。またその際は、それぞれの小さな問題が互いに独立し、影響を与え合わない必要があります。次に、各小問題の情報を扱い易い形 (例えば配列など) として表現出来る必要があり、なおかつこれらの情報 (or 配列) が大きくなり過ぎてはいけません。今回は題材として、動的計画法を利用することで効率的に解ける問題を選んで紹介しました。次の章では、(動的計画法を含め) これまでのアルゴリズム技法ではうまく扱えない問題を取り上げます。

3 クラスタリング

これまで、逐次細分最適化法/ランダム アルゴリズム/ループ処理/再帰処理/動的計画法といった様々なアルゴリズム技法について学んで来ました。これに対し、ある問題が与えられた場合、「どのような手順に直せば効率的に問題が解けるか」が不明であれば、これらアルゴリズム技法を駆使しても良い解には至らないことを (前回の最後で) 説明しました。以下で取り上げるクラスタリングの問題は、このような問題のグループに分類され、まだ効率的な解法 (手順) が見つかっていません。最後のテーマとして、この問題が難しい理由を説明し、本講義を終わることにします。

3.1 クラスタリングとは

第 8 回では、未知のデータ集合 (数値の組の列) が与えられた時、そこから何らかの規則を見い出す手法について取り上げました。その際は、議論を簡潔にするため、各データには取りあえず関係する事項が二つ (X と Y) あるとして (つまり、データは (X, Y) という組だとして)、X と Y が示す関係 (相関係数) や各 (X, Y) の組が示す関係 (回帰分析) について、数値解析を行ないました。これらと同様に、**クラスタリング** (clustering) も、未知のデータ集合から何らかの傾向を見い出す手法の一つです。具体的には、各データ組に対して、「似ているもの」あるいは「近そうなもの」を集めて自動的にグループ分け (分類) して行く手法です。また、分類されたグループのことを、**クラスタ** (cluster) と呼びます[8]。

クラスタリングについて説明する際に、よく使われる例として、商品の売り上げデータの分類があります。例えば、事前に各商品が、服/玩具/音楽/電化製品/食料品/日用品/園芸品…などに分類されていれば、商品が一つ売れる度に各分野の売り上げを一つ計上しておくことで、どの分野の商品が一番売れたかが分かります。しかし、事前の分類が出来ない場合、言い換えれば、どのような分野があるのか明確に定義出来ない場合は、何らかの基準を決め、それに従い売れた商品同士の類似性を調べて各商品を分類し、その後、新たに定義した分野毎の商品売り上げを計上することになります。

以下では、クラスタリングを行なうプログラムについて考えたいので、未知のデータを (第 8 回と同様) 2 次元数値データ (X, Y) とします。また、各数値データが似ているかどうかは、数値データ同士の距離が近いかどうかで判断することとします[9]。

3.2 k-means 法

数値データを距離に応じて分類する方法のうち簡単なものの一つは、ある距離 d を基準として、d 以内にあるデータ同士を結び付けて行く方法です。しかし、この方法には一つ問題があります。それは、そもそも各データの散らばり具合が不明なので、d 以内だけではなく、$d + \Delta d$ の辺りにも相当なデータが存在している場合をうまく分類出来ないからです。もちろん、d の値を変えながら、生成されるクラスタ数の変化を調べれば、最適な d の値を決められますが、これは実用的ではありません。

これに対し、生成されるクラスタ数を事前に決めておき、各データを距離の一番近いクラスタと結び付けて行く方法があります。もちろん、クラスタはデータの集合なので、データの集合とデータの距離を直接比較するこ

[8] 本来、クラスタは、葡萄などの房や動物などの群れを意味しています

[9] 例えば、商品データであれば、二つの商品が似ているかどうかは、材質/大きさ/購買層/用途など、各項目毎に概念的な距離を定義し、これらを総合して判断することになります。

とは出来ません。そこで、まずはクラスタに属する全データの中心点を定め (以後、これを重心と呼ぶことにします)、その重心からの距離をクラスタからの距離として扱うことにします。この方法は、あるデータが、クラスタ A の重心から $d + \Delta d$ の距離にあっても、他のクラスタ重心から $d + \Delta d$ 以上の距離にあれば、必ずクラスタ A に分類されるので、距離を基準とした分類に比べて、より「似ているもの」あるいは「近そうなもの」を集めることが出来そうです。

以下では、図 6 のようなデータが与えられ、これを二つのクラスタに分ける場合を考えてみましょう。

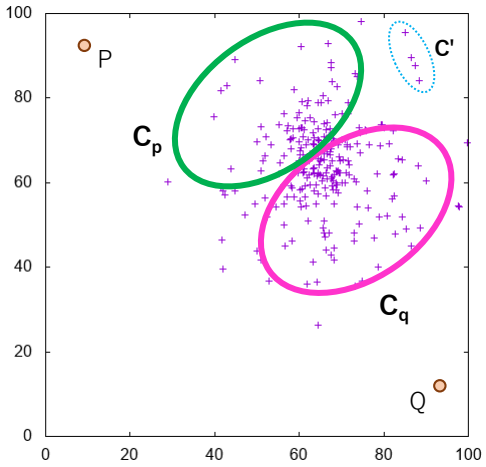

図 6: あるデータ分布を二つのクラスタに分ける例

クラスタの重心は、計算しなければ分からないので、取りあえず最初は適当な位置にある P, Q にしておきます。続いて、各データ毎に P, Q との距離を計算することで、P に近いクラスタ C_p と Q に近いクラスタ C_q に分けることが出来ます。これらは適当な重心に基いてクラスタ分けをしただけなので、最適な分類ではありませんが、“多少は” 近そうなデータになっていると言えます。そこで次は、クラスタ C_p に属するデータを用いて C_p の重心 P’ を計算することにします (クラスタ C_q も同じ)。P’ は P よりクラスタ C_p に近いので (Q’ も同じ)、C’ 周辺のデータは当初より正確に分類されると期待出来ます。

この処理を繰り返して行き、クラスタ間で移動するデータが存在しなくなれば (つまり、誤分類がなくなれば)、別の言い方をするとクラスタ重心が変化しなくなれば、最適な分類が完成したと言えます。但し、各副分類では重心の位置が毎回微妙に変わるので、その影響で永遠にクラスタ間を移動し続けるようなデータが存在し、この手順も永遠に終了しないことがありそうに思えます。しかし、次のように考えることで、このような場合は存在しないことが分かります。

略証:

この手順 (副分類) を、次の二つの Step の繰り返しと考えます。

Step.1: 各データ毎に、一番近い重心 $o_i^{(t)}$ を選出し、所属するクラスタを決定する。

Step.2: $o_i^{(t)}$ を重心とするクラスタに属する各データを対象に、最適な重心 $o_i^{(t+1)}$ を計算する。

また、“各データと自身のクラスタ重心との距離” を、(クラスタの区別なく) 全てのデータに対して集積したものを「総距離」と呼ぶことにします。

まずは、Step.1 から Step.2 への遷移および Step.2 から Step.1 への遷移における総距離の変化を考えます。Step.1 から Step.2 への遷移では、各クラスタ毎に最適な重心の位置を計算するので、(データの中には距離が増えてしまうものも幾つか生じますが) 全体として総距離が減ることはあっても増えないことは明らかです。さらに、Step.2 から Step.1 への遷移においても、以下の理由により総距離が増えることはありません。

Step.2 において、あるデータ a がクラスタ C_v に所属し、その最適なクラスタ重心を $o_v^{(t+1)}$ とする。Step.1 では重心の計算をしないので、$o_v^{(t+1)}$ は、Step.2 から Step.1 への遷移後も重心の選出候補として存在する。この時、Step.1 では、以下のどちらかが必ず成り立つ。

1) 新たに $|a - o_w^{(t+1)}| < |a - o_v^{(t+1)}|$ を満たす重心 $o_w^{(t+1)}$ が存在すれば、a の重心は $o_w^{(t+1)}$ に変わる。

2) 同、存在しないならば、a の重心は $o_v^{(t+1)}$ のまま変わらない。

どちらの場合も、データ a とその重心との距離は増えない。

これより、Step.2 から Step.1 への遷移においても、全体として総距離が減ることはあっても増えません。つまり、各副分類において、総距離は必ず減少して行くわけです。

次に、重心が変化する回数について考えます。重心の変化は、クラスタに属するデータの組み合わせが変わることにより生じます。クラスタ数およびデータ数は有限なので、その組み合わせも有限です。よって、重心が変化するパターンも有限になります。

以上より、有限回の重心変化で総距離は必ず最小となり[10]、以後変化することはありません。

最後に、総距離の変化とクラスタ間を移動するデータとの関係を考えます。総距離が変化しない場合とは、まずは Step.1 から Step.2 への遷移において、全ての重心 $o_i^{(t)}$ と $o_i^{(t+1)}$ が等しい場合だと言えます。この状態で Step.2 から Step.1 へ遷移した場合、データ a の最適重心も $o_v^{(t+1)}$ のまま変化することはありません。つまり、上記 2) しか発生しません。データがクラスタ間を移動する場合とは、上記 1) が発生する場合ですが (データ a は $\mathrm{C}_v \to \mathrm{C}_w$ に移動)、総距離が変化しなくなると、これは絶対に発生しないわけです。

よって、有限回の重心変化により、データはクラスタ間を移動しなくなります (データが永遠にクラスタ間を移動することはありません)。

この考え方に従ってグループ分けを行なう手順の概要を、疑似コードとして以下に示します (Python プログラムは 19 ページを参照)。このコードでは、データ移動の有無ではなく、重心移動の有無により、分類が完了したかどうかを調べています:

- calc_cl(fname, k) — ファイル `fname` 内のデータを `k` 個のクラスタに分ける
- 重心 `cset` の初期値を適当に決める
- クラスタ間を移動するデータが存在する間 (フラグ `changed` を調べる)、以下を繰り返し
- 現時点における各重心 `cset` からの距離に応じて、データを k 個のクラスタに分類
- k 個のクラスタ毎に理想的な重心を計算し、`i_cset` に記録
- これまでの重心 `cset` と今回計算した重心 `i_cset` との違いを調べる。
- 違いがあれば、フラグ `changed` に記録
- 記録した重心を入れ替える (`i_cset` → `cset`)
- (繰り返し終わり)

この手順を用いて図 7 のデータを分類した結果を、図 8 に示します。さらに、同じ手順によりもう一度最初から分類した結果を、図 9 に示します。両者を比べてみると、左上および右下辺りのデータに対するクラスタ分けに違いが見えます。両者の違いをもう少し詳しく調べるために総距離を見てみると、図 8 の総距離は 917.8、図 9 の総距離は 789.5 でした。結果としては、図 9 の方が、より良いクラスタに分類されているようです。しかし、図 8 においても、クラスタ間で移動するデータが存在しなくなるまで (誤分類がなくなるまで) 処理を進めたはずです。15 ページでの議論より、疑似コードで示した手順に何か矛盾や誤りがあるようには見えません。両者の違いは、初期値として適当に決めた重心の位置しかないので、初期値により結果が異なっているというわけです。これは、次のように考えることが出来ます。データの位置 $dset$ と重心の位置 $cset$ から、総距離 D を計算する関数を $D = F(dset, cset)$ とします。ここで、D の最小値を数値計算により求めたいのですが、関数 F は高次元多項式関数 (例えば、$y = ax^4 + bx^3 + cx^2 + dx + e$ など) のように凸凹しているので、$dset$ や $cset$ の初期値に応じて様々な凹に陥ってしまう (つまり、これを最小値だと計算してしまう) というわけです。これより、図 9 も最良のクラスタ分けとは言い切れません。

[10] 少し補足しておきます。重心の変化が有限回だったとしても、副分類により総距離が増えることがあれば、振動が発生して最小に至らない場合があるわけです。

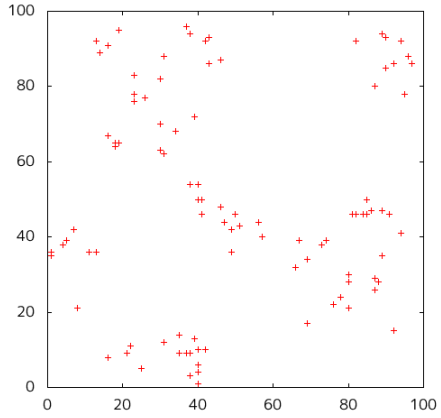

図 7: クラスタ分けする前のデータ分布

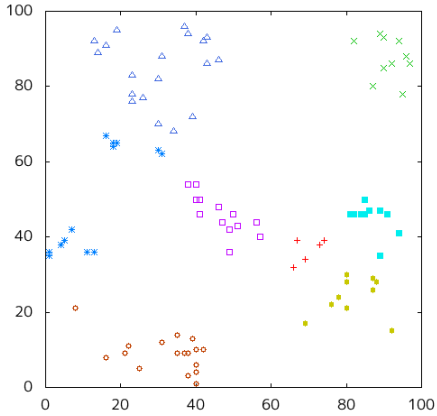

図 8: クラスタ分けの例 1

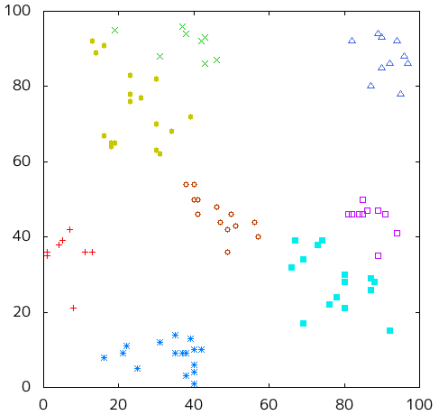

図 9: クラスタ分けの例 2

このような初期値の違いに応じて様々に異なる最適解を、**局所的最適解** (local optimal solution) と呼びます。これに対し、全ての局所最適解の中で最適な解を**大域的最適解** (global optimal solution) と呼びます。k-means 法において、動的計画法のような効率的な方法で大域的最適解が求められれば良いのですが、残念ながら現時点ではそのような方法は存在しません。動的計画法では、与えられた問題に対し、より小さな問題の組み合わせを用いることで解いて行きますが、14 ページで述べたように、小さな問題は大きな問題から独立しており、小さな問題を組み合わせて行く過程で、より小さな問題に影響を与えないという前提があります。もし、より小さな問題に影響を与えてしまうと、その解を利用することが出来なくなるので、結局のところ毎回最初から計算し直しとなってしまうわけです。k-means 法では、例えば “データを追加しながら…” という処理を考えた場合、データを追加する度に重心が変わるため、それ以前に追加したデータの重心にも影響が出てしまいます。

これより、大域的最適解を求める場合は、全てのデータを対象に最初から計算するしかありません。データ数を n とすると、まずはこれを k 個のクラスタに分類する方法は、

$$\sum_{i_1,\, i_2,\, \ldots,\, i_k \in I} {}_nC_{i_1} \cdot {}_{n-i_1}C_{i_2} \cdots {}_{n-(i_1+\ldots+i_{k-1})}C_{i_k} \tag{1}$$

但し、I は $i_1 + i_2 + \ldots + i_k \leq n,\ i_1 \geq i_2 \geq \ldots \geq i_k \geq 1$ を満たす全ての組み合わせ集合[11]

通りになります (詳細は、#8 の 2.2 節および式 (2) を参照)。そして、各分類毎にクラスタ重心を計算し、総距離が最小となる場合を求めるわけです。式 (1) には階乗が入っているので、計算量は最悪レベルと言えます。

[11] この組み合わせ集合について、少し補足しておきます。#8 と異なり、今回は k 個のクラスタに分類したいので、例えば $n = 9$, $k = 4$ の場合における $(i_1, i_2, \ldots, i_k)$ の組合わせ集合は、{(6,1,1,1), (5,2,1,1), (4,3,1,1), (4,2,2,1), (3,3,2,1), (3,2,2,2)} となります。参考までに、n をこのような k 個の数字に分けるプログラムを 22 ページに示しておきます。このプログラムでは、n 個のブロックを場所 a_i $(0 \leq i \leq k-1)$ に配置する問題として考えています。まずは、各 a_i に 1 ブロックを置き、残りの $n-k$ 個のブロックを a_0 に積みます。その後、“a_i のブロック数” > “a_j のブロック数” $(i < j)$ を満たしながら (つまり、常に勾配が降順になるように)、各 a_i にあるブロックを a_j へ移動させ、その回数を数えます。これに対し、#8 の組み合わせ集合は、同じく $n = 9$ の場合、{(9), (8), (7,2), (7), …, (5), (4,4), (4,3,2), (4,3), …, (2,2,2,2), (2,2,2), (2,2), (2)} という具合に、各組み合わせに出現する全ての数字を足しても 9 になる必要がないため、これを求めるプログラムは、より複雑となります (仮に 22 ページのプログラムに従って考えるならば、ブロックの移動だけではなく、ブロックの消失を追加する必要があります)。

最後に

3 章の冒頭では、ある問題が与えられた場合、「どのような手順に直せば効率的に問題が解けるか」が不明であれば、アルゴリズム技法を駆使しても良い解には至らない、と述べました。そして、クラスタリングの問題は、このような問題のグループに分類されることも述べました。最後に、解決困難な問題に関連する話題を紹介して、本講義を終えることにします。

ある問題が与えられ、それが多項式時間の計算量で解ける場合、その解法 (手順) は実用的だと考えられます。例えば、組み合わせ数 ${}_nC_r$ をループにより計算するアルゴリズムなどが該当します (再帰処理にしてしまうと $O(2^n)$ になってしまいますが、ループ処理だと $O(n)$ でしたね)。このような問題が属するグループを **P (polynomial time) クラス**と呼びます。この講義で取り上げたアルゴリズムのほとんどは、P クラスに属する問題でした。

続いて、次のような問題を考えてみましょう。

「ある数 x が与えられた時、x が合成数か素数かを判定せよ。」

この解を得るためには、x を 2 以上の数で割り算して行く必要があります。#6, #7 では、2 以上の数を効率的に選ぶ方法を紹介しましたが、残念ながら x が数千桁の数になると (x = 数千 ではないですよ)、合成数か素数かを判定するための計算量はやはり膨大になります。しかし、もし x が p で割り切れるというヒントがあれば、実用的な計算量 (即ち多項式時間の計算量) で判定することが出来ます。この一連の手順を、もう少し一般的な言い方に改めると、これは、“与えられたヒントが正しいかどうかを検証している” ことだと言えます。つまり、合成数/素数判定問題は、解を直接求めることは困難だが、与えられたヒント (これを “証拠” と呼びます) を多項式時間の計算量で検証出来るというわけです。このような問題が属するグループを **NP (Non-deterministic Polynomial time) クラス**と呼びます。

ここで、P クラスと NP クラスの関係について、少しだけ説明しておきます。多項式時間で問題が解ける場合、多項式時間で証拠 (つまり解) の検証が既に出来ているとも言えるので、P $\subseteq$ NP であることは明らかです。では、P $\subset$ NP と P = NP のどちらが成り立つのでしょうか。これが、世界の研究者達が現在取り組んでいる未解決な有名難問「P $\neq$ NP 予想」です。仮に、P = NP が証明された場合、多項式時間で証拠の検証が出来る問題は、全て多項式時間で解けることになります。これは、上で挙げた合成数/素数判定問題のような難問に対し、多項式時間で解を得られる “未発見な素晴らしい手順” が存在することを意味します。P クラスと NP クラスの正確な関係については、まだ証明されていませんが、長年に渡り、世界中の研究者が様々な難問に対して効率的な手順を探求して来たものの、現時点で一つも見つかっていないことから、P $\neq$ NP (つまり P $\subset$ NP) だと予想されています。

さて最後に、3.2 節で取り上げた k-means 法について考えてみましょう。証拠として、“最良のクラスタ分け: C_{best}” が示されたとします。しかし、C_{best} が本当に最良であるかどうかは、どのように検証したらよいでしょうか。最終的に C_{best} が最良であるかどうかを判断するには、結局のところ全てのクラスタ分け C_{other} を求め、これら一つ一つと比べる必要があります。つまり、k-means 法を用いて「最良」なクラスタ分けをする問題は、証拠の検証さえも多項式時間の計算量で終了出来ないわけです。このような問題が属するグループを、最も難しい問題という意味で **NP 困難 (NP-hard) クラス**と呼びます。NP 困難な問題は、最適解を得ることがほぼ不可能なため、例えば 3.2 節で考えたような様々な近似アルゴリズムが試みられています。

演習 13-1 本資料に掲載されている各プログラムのうち、適当なものを入力し、その動作を確認せよ (k-means 法のテスト データは、Web ページより取得できます)。

演習 13-2 過去の資料に掲載されている各プログラム (特に#5 以降) について、その意味/動作を再確認せよ。

演習 13-3 コンピュータ上での計算において生じる誤差の種類とその理由、および誤差を削減する工夫について再確認せよ。

参考: k-means 法の Python プログラム

参考として、Python による k-means 法のプログラムを示しておきます。基本的な手順については、16 ページの疑似コードで示してあります (詳細な動作については、プログラム内のコメントを参照)。このプログラムは、例えば、データの入っているファイルが cls.data で、分類するクラスタ数が 8 個の場合、calc_cl ("cls.data", 8) と実行します。データ ファイルには、各データを「x 座標 y 座標」という形式で、1 データを 1 行に入れています (100 データあれば 100 行)。クラスタ分けした結果も同じく、各データが「x 座標 y 座標」という形式で、1 データ毎に 1 行ずつ出力しますが、クラスタの境界を #### で表しています。図 8, 9 は、出力されたデータを各クラスタ毎のファイルに切り分け、gnuplot により重ね合わせて表示したものです:

```
import i2a
import random
import math

MAX_INT = 1000000000    # Python では整数の上限が適宜拡張されるので、取りあえず 10 億にする。


# 評価用データの読み込み + 重心の初期値設定
def init_data(fname, dset, cset):

  # ファイルから 1 行読み込み、line に格納する。
  i = 0
  for line in open(fname, "r"):

    # 配列の要素を一つ増やす (初期値: x/y 座標 = 0, 所属するグループ = -1)。
    dset.insert(len(dset), [0, 0, -1])

    # まずは、読み込んだ行を " " 毎に区切る。
    items = line.split(" ")

    # 区切られた部分には、まだ ",", "(", ")", "\n" (改行) が残っているので、
    # それらを削除した後、数値データとして配列へ格納する。
    # dset における x 座標, y 座標の扱いについては、コメント (*1) を参照
    dset[i][0] = int(items[0].strip("(),\n"))
    dset[i][1] = int(items[1].strip("(),\n"))
    i = i + 1

  # 重心の初期値として適当なものを設定
  # ("重心 ≠ 評価用データのどれか " だが、ここでは取りあえず、
  #  ランダムに k 個選んだ評価用データの座標を重心にしておく)
  for j in range(len(cset)):
    while True:    # 面倒だが、重複しない値を選択するまで繰り返し
      k = int(len(dset)*random.random())
      if dset[k][2] == -1:
        cset[j][0] = dset[k][0]; cset[j][1] = dset[k][1]; dset[k][2] = j
        break

  # 各データが所属するグループの初期値として適当なものを設定
  for j in range(len(dset)):
    if dset[j][2] == -1:
      dset[j][2] = int(len(cset)*random.random())
```

```
# 重心からの距離に応じて評価用データを分類
def classify_dset(dset, cset):

  # 評価用データ毎に最も近い重心を設定
  for i in range(len(dset)):

    # 重心からの距離を記録する必要はないので、math.sqrt 関数は省略
    min_d = MAX_INT
    min_c = -1
    for j in range(len(cset)):
      if cset[j][0] == -1 and cset[j][1] == -1:    # 意味は (*1) を参照
        continue
      dx = cset[j][0] - dset[i][0]; dy = cset[j][1] - dset[i][1]
      if (min_d >= dx**2 + dy**2):
        min_d = dx**2 + dy**2; min_c = j
    dset[i][2] = min_c


# 各グループ毎に理想的な重心を計算
def calc_cset(dset, i_cset):

  # 各評価用データが属するグループの理想的な重心を算出
  for k in range(len(i_cset)):

    # 簡略化のため、評価用データの x, y 座標は正数に限定
    mx = 0; my = 0; cnt = 0
    for i in range(len(dset)):
      if dset[i][2] == k:
        mx = mx + dset[i][0]; my = my + dset[i][1]
        cnt = cnt + 1

    # *1) cnt == 0 とは、データを k グループに分けられなかったことを意味する。
    #      強制的に分割するのは少々面倒なので、以後は素直に重心数は k 個未満
    #      として扱う (目印として -1 を入れておく)。
    if cnt != 0:
      i_cset[k][0] = mx/cnt; i_cset[k][1] = my/cnt
    else:
      i_cset[k][0] = -1; i_cset[k][1] = -1


# 新旧重心群の相違を確認
def is_changed(cset, i_cset):

  for i in range(len(cset)):
    changed = True
    for j in range(len(i_cset)):

      # 新旧重心群に同じものがあった → この重心は変化しなかった
      if (cset[i][0] == i_cset[j][0]) and (cset[i][1] == i_cset[j][1]):
        changed = False
        break
    # 新旧重心群に同じものがなかった → この重心は変化した
    if changed == True:
      break

  return(changed)
```

```
# 評価用データを k 個のグループに分類
def calc_cl(fname, k):

  # 評価用データを格納する 3 次元配列を用意
  # 3 次元成分の意味 = (x 座標, y 座標, 所属するグループ)
  # 注意!!: 多次元配列は、"1 次元配列の各要素が 1 次元配列になる" という
  #          構造なので、x 座標 = dset[i][0], y 座標 = dset[i][1],
  #          所属するグループ = dset[i][2] となる (*1)。
  dset = []

  # k 個のグループの重心を記録しておく 2 次元配列を用意
  # 2 次元成分の意味 = (x 座標, y 座標)
  # 重心は、評価用データのどれかではなく、計算で得られる理想上の点
  cset = i2a.array.make2d(k, 2)

  # 評価用データを dset に読み込み、適当に決めた k 個の重心を cset に記録
  init_data(fname, dset, cset)

  # メイン ループ (評価用データを分類)
  changed = True
  while changed:

    # 現重心からの距離に応じて、評価用データを k 個のグループに分類
    classify_dset(dset, cset)

    # k 個のグループ毎に理想的な重心を計算し、i_cset に記録
    i_cset = i2a.array.make2d(k, 2)
    calc_cset(dset, i_cset)

    # 前回記録した重心群と今回計算した重心群とに違いがあるか?
    changed = is_changed(cset, i_cset)
    cset = i_cset    # 次回に備えて、取りあえず入れ替えておく。

  # 出力 (各グループ間は #### で区切っている)
  for k in range(len(cset)):
    for i in range(len(dset)):
      if dset[i][2] == k:
        print("%g %g" % (dset[i][0], dset[i][1]))
    print("####")

  # 全評価用データ～重心間の距離を計算
  d = 0
  for i in range(len(dset)):
    dx = dset[i][0] - cset[dset[i][2]][0]
    dy = dset[i][1] - cset[dset[i][2]][1]
    d = d + math.sqrt(dx**2 + dy**2)
  print(d)
```

参考: 組み合わせ集合を求める Python プログラム

```
import i2a

def calc_cs(n, k):

  # 全ての場所にブロックを配置出来ないので排除
  if n < k: return(None)

  # 各場所に n 個のブロックを初期配置
  a = i2a.array.make1d(k)
  for i in range(len(a)): a[i] = 1
  a[0] += n-k; print(a)
  cnt = 1    # 初期配置の時点で、一つの組み合わせが得られたという意味

  # a[p] > a[q] (p < q) を満たしながら a[p] → a[q] へブロックを移動
  done = False
  while not done:
    done = True

    # まずは、隣のブロック数と 2 個以上の段差がある場合、これを移動する。
    # 段差: 2x → x 回移動, 2x+1 → x 回移動 (この調整を int 関数で実施)
    for i in range(len(a)-1):
      if a[i] > a[i+1] + 1:
        d = int((a[i] - a[i+1])/2); cnt += d
        for j in range(d): a[i] -= 1; a[i+1] += 1; print(a)
        done = False
        break

    # 隣のブロック数と 2 個以上の段差がある場所が存在しなかった場合は、
    # 間を開けて 2 個以上の段差がある場所を探す。
    # 但し、移動後に a[p] > a[q] (p < q) の条件を破らないようにするため、
    # 移動元の候補は a[i] = a[i+1] + 1 となっている必要がある (*1)。
    # 例えば、[3, 3, 2, 1] であれば、a[1] = 3 → a[3] = 1 に移動させる。
    # 安易に a[0] = 3 を選ぶと、[2, 3, 2, 2] となり、条件を満たさない。
    if done == True:
      for i in range(len(a)-1):
        if a[i] == a[i+1]: continue    # この処理の意味は、上記 (*1) を参照
        for j in range(i, len(a)-1):
          if a[i] > a[j+1] + 1:
            a[i] -= 1; a[j+1] += 1; print(a); cnt += 1
            done = False
            break
        if done == False: break

  # 移動回数を表示して、プログラムを終了
  print(cnt)
  return(None)
```