動的計画法と配列アラインメント

コンピューター・サイエンスを分子生物学に役立てる

Paul D. Reiners
Staff Software Engineer
IBM

2008年 3月 11日

分子生物学では、コンピューター・サイエンスのアルゴリズムを研究用のツールとしてますます利用するようになってきています。この記事では、コンピューターを使って生物学上の問題を解決する、バイオインフォマティクス (bioinformatics: 生命情報科学) を紹介します。そして多くのプログラミング・プロジェクトで役立つ高度なアルゴリズム手法である、動的計画法の基本を学びます。

遺伝学では、非常に大量の生データをデータベースに保持しています。ヒトゲノムだけでも約 30 億の DNA 塩基対を持っています。こうしたデータすべてを検索し、その中から意味のある関係を見つけるために、分子生物学者はコンピューター・サイエンスの効率的なストリング・アルゴリズムに次第に大きく依存するようになってきています。この記事では、そうしたアルゴリズムのうちの 3 つを紹介しますが、この 3 つのアルゴリズムはどれも動的計画法を使用しています。動的計画法は、副問題に対する最適解を発見しながらボトムアップの方法で最適化問題を解決する、高度なアルゴリズム手法です。この記事ではこれらのアルゴリズムを Java™ で実装し、また生物学のデータを処理するためのオープンソースの Java フレームワークについて学びます。

遺伝学とストリング・アルゴリズム

遺伝物質である DNA と RNA の鎖は、ヌクレオチド (nucleotide) と呼ばれる小さな単位の配列 (sequence) からなっています。研究上の重要な疑問に答えるという目的において、遺伝のストリングはコンピューター・サイエンスのストリングと等価です。つまり物理的、化学的な性質を無視すれば、遺伝のストリングは単なる文字の配列と考えることができます。(ただし厳密に言えば、遺伝のストリングの化学的性質も、通常はこの記事で説明するストリング・アルゴリズムのパラメーターとしてコード化されています。)

この記事の例では DNA を使いますが、DNA はアデニン (adenine: A)、シトシン (cytosine: C)、チミン (thymine: T)、そしてグアニン (guanine: G) というヌクレオチドの 2 本の鎖から構成されています。DNA の 2 本の逆向きの鎖は互いに相補関係になっています。A と T、そして C と G がそれぞれ相補的塩基です。つまり、1 本の鎖の A はもう 1 本の鎖の T と対になり (またその逆同士が対になり)、そして1 本の鎖の C はもう 1 本の鎖の G と対に (またその逆同士が対に) なっています。そ

のため、1 本の鎖の A、C、T、G の配列がわかると、もう 1 本の鎖の配列も導き出すことができます。従って DNA の鎖は単なる A、C、G、T という文字のストリングと考えることができます。

動的計画法

動的計画法は配列解析で一般的に使われるアルゴリズム手法です。動的計画法が使われるケースは、再帰が使えるものの、再帰では同じ副問題を繰り返し解決することになるため非効率になってしまうケースです。例えばフィボナッチ数列 (0, 1, 1, 2, 3, 5, 8, 13, …) を考えてみてください。1 番目と 2 番目のフィボナッチ数は、それぞれ 0 と 1 に定義されています。n 番目のフィボナッチ数は、この数の前 2 つのフィボナッチ数の和として定義されます。従って n 番目のフィボナッチ数は再帰関数を使って計算することができます (リスト 1)。

リスト 1. n 番目のフィボナッチ数を計算する再帰関数

```
public int fibonacci1(int n) {
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return fibonacci1(n - 1) + fibonacci1(n - 2);
   }
}
```

しかしリスト 1 のコードは、いくつかの同じ再帰副問題を繰り返し解くことになるため、効率的ではありません。例えば `fibonacci1(5)` の計算を考えてみてください (図 1)。

図 1. フィボナッチ数の再帰的な計算

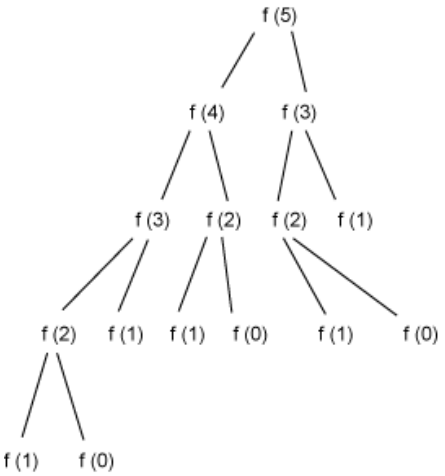


図 1 を見るとわかるように、例えば `fibonacci1(2)` は 3 回計算されています。フィボナッチ数はトップダウンで計算するよりも、(リスト 2 のように) ボトムアップで計算した方が、はるかに効率的です。

リスト 2. ボトムアップによるフィボナッチ数の計算

```
public int fibonacci2(int n) {
   int[] table = new int[n + 1];
   for (int i = 0; i < table.length; i++) {
      if (i == 0) {
         table[i] = 0;
      } else if (i == 1) {
         table[i] = 1;
      } else {
         table[i] = table[i - 2] + table[i - 1];
      }
   }

   return table[n];
}
```

リスト 2 は、中間結果を破棄して何度も計算し直すのではなく、再利用できるように中間結果をテーブルに保存します。テーブルに保存する方法は (一度にテーブルの 2 つのエントリーしか使用しないため) メモリーの利用に関して非効率であることは事実ですが、今はそれを無視することにします。この方法で行う理由は、これと似たテーブルを、後の方で説明するもっと複雑な例でも使用するからです (ただしそこで使用するテーブルは 2 次元のテーブルです)。重要な点は、リスト 2 の実装はリスト 1 の実装よりも時間効率がずっと高いということです。リスト 2 の実装は O(n) 時間で実行されます。ここでは証明しませんが、リスト 1 の初歩的な再帰による実装では実行時間が n のべき乗で増加します。

これは動的計画法の動作そのものです。つまりトップダウンで再帰的に解くことができる問題を、ボトムアップで繰り返しによって解決するのです。中間結果を後で使えるようにテーブルに保存します。そうしないと中間結果を繰り返し計算する羽目になり、効率の悪いアルゴリズムになります。しかし動的計画法は、この記事の他の例で説明するように、フィボナッチ数の問題などよりも最適化の問題に適用されるのが普通です。次の例は、計算生物学 (computational biology) で一般的に使われるようなストリング・アルゴリズムです。

最長共通部分列の問題

まず、動的計画法を使って 2 つの DNA 配列の LCS (longest common subsequence: 最長共通部分列) を求める方法を調べます。新しい遺伝子配列を発見する生物学者は通常、その配列に最も似た他の配列は何かを知ろうとします。LCS を求める作業は、2 つの配列がどれほど似ているかを計算することでもあります。つまり LCS が長ければ長いほど 2 つの配列は似ていることになります。

部分列の中の文字列は、サブストリングの中の文字列とは異なり、配列の中で連続している文字列である必要はありません。例えば ACE は ABCDE の部分列ですが、ABCDE のサブストリングではありません。次の 2 つの DNA 配列を考えてみてください。

- S1 = `GCCCTAGCG`
- S2 = `GCGCAATG`

この 2 つの配列の LCS は GCCAG であることがわかります。(これが唯一の LCS ではなく、LCS の 1 つであることに注意してください。同じ長さの共通部分列は他にも存在するかもしれないからです。この最適化問題や、この先で調べる他の最適化問題には解が複数ある可能性があります。)

LCS のアルゴリズム

まず、LCS を再帰的に計算する方法を考えます。次のように仮定しましょう。

- C1 を S1 の右端の文字とします。
- C2 を S2 の右端の文字とします。
- S1' を、C1 を「取り去った」 S1 とします。
- S2' を、C2 を「取り去った」 S2 とします。

再帰的な副問題は次のように 3 つあります。

- L1 = LCS(S1', S2)
- L2 = LCS(S1, S2')
- L3 = LCS(S1', S2')

ここでは証明しませんが、元々の問題に対する解は下記のうち最も長いものであることが示されています (またそれを理解するのは難しくはありません)。

- L1
- L2
- C1 と C2 が等しい場合には L3 に C1 を付加したもの、また C1 と C2 が等しくない場合には L3

(基本ケースは S1 または S2 が長さゼロのストリングの場合です。この場合では、S1 と S2 の LCS は明らかに長さゼロのストリングです。)

しかし再帰によってフィボナッチ数を計算する手順と同様、この再帰的なソリューションでは同じ副問題を複数回計算しなければならず、実行時間が指数関数的に増加することが示されます。対照的に、動的計画法によるこの問題に対するソリューションの実行時間は Θ(mn) です (m と n は 2 つの配列の長さです)。

動的計画法を使って効率的に LCS を計算するためには、まず中間結果を作成するためのテーブルを作成します。一方の配列が第 1 行、もう一方の配列が第 1 列、というテーブルを作ります (図 2)。

図 2. 最初の LCS テーブル

		G	C	C	C	T	A	G	C	G
G										
C										
G										
C										
A										
A										
T										
G										

考え方としては、このテーブルに、上から下、左から右へと数字を入力していき、各セルに入る数字が 2 つのストリングの先頭からそのセルの行と列までで構成されるサブストリングの LCS の長さを表すようにします。つまり、各セルは元々の問題の副問題に対する解を含むことになります。例えば、6 行目で 7 列目のセルを考えてみてください。このセルは GCGCAATG の 2 番目の C の右、また GCCCTAGCG の T の下にあります。このセルには、最終的に GCGC と GCCCT の LCS の長さを表す数字が入ることになります。

まず、このテーブルの 2 行目のエントリーは何かを考えます。これらのエントリーは、左にある配列 GCGCAATG の先頭部分の長さゼロのサブストリングと、上にある配列 GCCCTAGCG の先頭からエントリーのあるセルの列までのサブストリングとの LCS の長さを表しています。明らかに、これらの LCS の値はすべて 0 です。2 列目の値も、同様にすべて 0 です。これは再帰的なソリューションの基本ケースに相当します。ここまでで、テーブルは図 3 のようになります。

図 3. 基本ケースが入力された LCS テーブル

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0									
C	0									
G	0									
C	0									
A	0									
A	0									
T	0									
G	0									

次に、再帰アルゴリズムの再帰的なサブケースに対応するものを実装しますが、既に入力された値を使います。図 4 では約半数のセルに数字が入力されています。

図 4. セルの半数に数字が入力された LCS テーブル

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3						
A	0									
A	0									
T	0									
G	0									

セルに数字を入力する際には、下記のセルを考慮します。

- そのセルのすぐ左側のセル
- そのセルのすぐ上のセル
- そのセルの左上のセル

下記の 3 つの値はそれぞれ、先ほど示した 3 つの再帰的副問題によって返される値に対応します。

- V1 = 左側にあるセルの値
- V2 = 上にあるセルの値
- V3 = 左上にあるセルの値

次の 3 つのなかで最も大きな数字を空のセルに入力します。

- V1
- V2
- C1 と C2 が等しい場合には V3 + 1、C1 と C2 が等しくない場合には V3 (C1 は現在のセルの上にある文字であり、C2 は現在のセルの左にある文字です)。

これらの 3 つのセルのうち、どれを使って現在のセルの値が得られたのかわかるように、矢印を追加してあることに注目してください。このテーブルは、(単に LCS の長さを求めるためだけに使うのではなく) 後ほど「トレースバック」の際にこれらの矢印を利用して、実際に LCS を作成するために使うことになります。

今度は図 4 の中の、次の空白セル (GCCCTAGCG の 3 番目の C の下、GCGCAATG の 2 番目の C の右にある空白セル) に数字を入力します。このセルの上には 2 があり、左には 3 があり、左上に

は 2 があります。このセルの上にある文字とこのセルの左側にある文字は同じ (どちらも C) なので、2、3、3 (左上のセルの 2 に 1 を加えた数) のうち最大の数字を選択する必要があります。従ってこのセルの値は 3 になります。この新しい数字を得る元となったセルがわかるように矢印を描きます。この場合は新しい数字の元となりうるセルが 1 つ以上あるため、その中から任意の 1 つ、例えば左上のセルを選びます。

演習として、このテーブルの残り部分に値を入力してみるとよいかもしれません。もし同順位のセルがある場合には必ず左上のセルを上のセルより優先し、また上のセルを左のセルより優先することにすると、図 5 のテーブルが得られます。(もし同順位の場合に他の方法で選択を行うと、当然ながら矢印のパターンは異なりますが、数字は同じになります。)

図 5. データが入力された LCS テーブル

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

どのセルの数字も、上と左にあるストリングの先頭からそのセルの行と列までで構成されるサブストリングの LCS の長さであることを思い出してください。従って、最も右下のセルの数字は、2 つのストリング S1 と S2 (この場合は GCCCTAGCG と GCGCAATG) の LCS の長さを表します。つまりこの 2 つの配列の LCS の長さは 5 です。

ここで説明したことは、こうした動的計画法のアルゴリズムの重要なポイントとして覚えておく必要があります。つまりこのテーブルの各セルは、上と左にある配列の、先頭からそのセルの行と列までで構成される配列に対する問題の解を含んでいるのです。

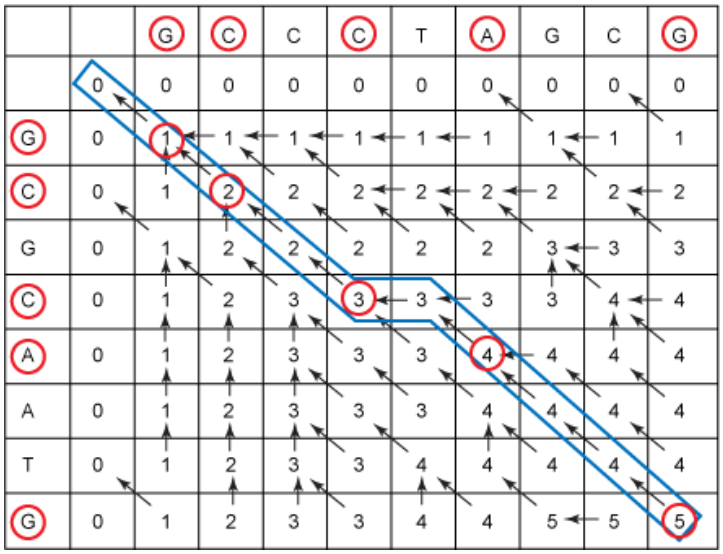
実際の LCS をトレースバックで求める

次に行うのは、実際の LCS を求めることです。これは、先ほど描いたセル・ポインター (矢印) を使うトレースバック・ステップとして行います。テーブルを作成する際、左上のセルへのポインターがあり、また現在のセルの値が左上のセルの値よりも 1 大きい場合、左側の文字と上側の文字が等しかったことを思い出してください。このケースは、LCS の作成においては、この等しい文字を LCS に追加することに対応します。そのため LCS を作成する方法としては、右下のコーナーのセルから開始し、そこからポインターの矢印をさかのぼっていきます。左上の対角線上にあるセルへのポインターをたどる場合、その矢印の先にあるセルの値が現在のセルの値よりも 1 少なければ、対応する共通の文字を、作成中の LCS の先頭に追加します。先頭に追加する理由は

LCS の最後から開始しているからであることに注意してください。(図 5 の場合、右下のセルにある 5 は追加された 5 番目の文字に対応します。)

では LCS を作成することにしましょう。右下のコーナーにあるセルから開始すると、セル・ポインターが左上を指しており、また現在のセルの値 (5) が左上のセルの値 (4) よりも 1 大きいことがわかります。そこで、最初の、長さゼロのストリングに文字 G を追加します。その 4 の値のセルからの矢印も左上を指していますが、その左上のセルの値は変わらず 4 のままです。さらにその変わらず 4 の値を持つセルからの矢印も左上を指していますが、その左上のセルもやはり値は 4 と変わりません。そしてついに、その 4 の値を持つセルからの矢印が指す左上のセルで、値が 4 から 3 に変わっています。これは、そのセルの行と列に共通の文字 (A) を LCS に追加したことを意味します。そのため、ここまでの段階で LCS は AG です。ここから、ポインターを左にたどり (これは上にある T をスキップすることに対応します)、隣の 3 に進みます。すると、そこには対角線方向にある 2 を指すポインターがあります。従って現在の行と列に共通の文字 (C) を追加するので、LCS は CAG となります。このようにして、最終的に 0 に達するまで続けます。図 6 はこのトレースバックの全体を示したものです。

図 6. データが入力された LCS テーブルとトレースバック



このトレースバックから、ある 1 つの LCS として GCCAG が得られます。

Java 言語による動的計画法の実装

今度は Java 言語を使って動的計画法のアルゴリズムを実装します。まず LCS アルゴリズムを使って、その少し後で別の 2 つのアルゴリズムを使って、配列アラインメント (sequence alignment) を実行します。それぞれの例では、ある方法で 2 つの配列を比較し、また 2 次元のテーブルを使って副問題の解を保存します。ここでは抽象クラス `DynamicProgramming` を定義します。このクラスはすべてのアルゴリズムに共通なコードを含んでいます。この記事のサンプル・コードはすべて「ダウンロード」セクションから入手することができます。

まず、テーブルのセルを表すクラスが必要です (リスト 3)。

リスト 3. Cell クラス (リストの一部)

```
public class Cell {
   private Cell prevCell;
   private int score;
   private int row;
   private int col;
}
```

どのアルゴリズムの場合も、最初のステップはテーブルの中のスコアを初期化することです。また場合によるとテーブルの中のポインターも初期化する必要があります。最初のスコアとポインターをどう設定するかはアルゴリズムによって異なります。そのため、`DynamicProgramming` クラスは 2 つの抽象メソッドを定義しています (リスト 4)。

リスト 4. DynamicProgramming の初期化コード

```
protected void initialize() {
   for (int i = 0; i < scoreTable.length; i++) {
      for (int j = 0; j < scoreTable[i].length; j++) {
         scoreTable[i][j] = new Cell(i, j);
      }
   }
   initializeScores();
   initializePointers();

   isInitialized = true;
}

protected void initializeScores() {
   for (int i = 0; i < scoreTable.length; i++) {
      for (int j = 0; j < scoreTable[i].length; j++) {
         scoreTable[i][j].setScore(getInitialScore(i, j));
      }
   }
}

protected void initializePointers() {
   for (int i = 0; i < scoreTable.length; i++) {
      for (int j = 0; j < scoreTable[i].length; j++) {
         scoreTable[i][j].setPrevCell(getInitialPointer(i, j));
      }
   }
}

protected abstract int getInitialScore(int row, int col);

protected abstract Cell getInitialPointer(int row, int col);
```

次に、テーブルの各セルにスコアとポインターを入力します。入力の方法もアルゴリズムごとに異なるため、抽象メソッド `fillInCell(Cell, Cell, Cell, Cell)` を使います。リスト 5 はテーブルにデータを入力するための `DynamicProgramming` のメソッド群を示しています。

リスト 5. テーブルにデータを入力するための DynamicProgramming のコード

```
protected void fillIn() {
   for (int row = 1; row < scoreTable.length; row++) {
      for (int col = 1; col < scoreTable[row].length; col++) {
         Cell currentCell = scoreTable[row][col];
         Cell cellAbove = scoreTable[row - 1][col];
         Cell cellToLeft = scoreTable[row][col - 1];
         Cell cellAboveLeft = scoreTable[row - 1][col - 1];
         fillInCell(currentCell, cellAbove, cellToLeft, cellAboveLeft);
      }
   }
}

protected abstract void fillInCell(Cell currentCell, Cell cellAbove,
      Cell cellToLeft, Cell cellAboveLeft);
```

最後はトレースバックです。トレースバックの方法はアルゴリズムによって異なります。リスト 6 は `DynamicProgramming.getTraceback()` メソッドを示しています。

リスト 6. DynamicProgramming.getTraceback() メソッド

```
abstract protected Object getTraceback();
```

Java による LCS の実装

これで、LCS アルゴリズムを Java で実装するためのコーディングの準備ができました。

セルのスコアを初期化する動作は簡単です。リスト 7 に示すように、すべてのセルを最初は 0 に設定するだけです (後ほど一部のセルをリセットします)。

リスト 7. LCS の初期化メソッド

```
protected int getInitialScore(int row, int col) {
   return 0;
}
```

リスト 8 はテーブルの個々のセルにスコアとポインターを入力するためのコードを示しています。

リスト 8. セルにスコアとポインターを入力するための LCS メソッド

```
protected void fillInCell(Cell currentCell, Cell cellAbove, Cell cellToLeft,
      Cell cellAboveLeft) {
   int aboveScore = cellAbove.getScore();
   int leftScore = cellToLeft.getScore();
   int matchScore;
   if (sequence1.charAt(currentCell.getCol() - 1) == sequence2
         .charAt(currentCell.getRow() - 1)) {
      matchScore = cellAboveLeft.getScore() + 1;
   } else {
      matchScore = cellAboveLeft.getScore();
   }
   int cellScore;
   Cell cellPointer;
   if (matchScore >= aboveScore) {
```

```
      if (matchScore >= leftScore) {
         // matchScore >= aboveScore and matchScore >= leftScore
         cellScore = matchScore;
         cellPointer = cellAboveLeft;
      } else {
         // leftScore > matchScore >= aboveScore
         cellScore = leftScore;
         cellPointer = cellToLeft;
      }
   } else {
      if (aboveScore >= leftScore) {
         // aboveScore > matchScore and aboveScore >= leftScore
         cellScore = aboveScore;
         cellPointer = cellAbove;
      } else {
         // leftScore > aboveScore > matchScore
         cellScore = leftScore;
         cellPointer = cellToLeft;
      }
   }
   currentCell.setScore(cellScore);
   currentCell.setPrevCell(cellPointer);
}
```

最後に、トレースバックを使って実際の LCS を作成します。

リスト 9. LCS のトレースバック・コード

```
protected Object getTraceback() {
   StringBuffer lCSBuf = new StringBuffer();
   Cell currentCell = scoreTable[scoreTable.length - 1][scoreTable[0].length - 1];
   while (currentCell.getScore() > 0) {
      Cell prevCell = currentCell.getPrevCell();
      if ((currentCell.getRow() - prevCell.getRow() == 1 && currentCell
            .getCol()
            - prevCell.getCol() == 1)
            && currentCell.getScore() == prevCell.getScore() + 1) {
         lCSBuf.insert(0, sequence1.charAt(currentCell.getCol() - 1));
      }
      currentCell = prevCell;
   }

   return lCSBuf.toString();
}
```

このアルゴリズムによる計算が Θ(mn) 時間 (そしてスペース) を要することは容易にわかります (m と n は 2 つの配列の長さです)。各セルにデータを入力するための時間は一定 (単に一定数の加算と比較) であり、データの入力が必要なセルの数は mn です。またトレースバックは O(m + n) 時間、実行されます。

Java による次の 2 つの例は、配列アラインメント用のアルゴリズムである、Needleman-Wunsch アルゴリズムと Smith-Waterman アルゴリズムを実装しています。

配列アラインメント

ホモロジー

ホモロジー (homology) は生物学上の重要な概念です。2 つの種が進化の上で共通の祖先を共有している場合、その 2 つの種は相同 (homologous) と見なされます。相同な種の DNA には

共通な部分が多数あります。逆に、2 つの種の DNA のサブストリングが似ている場合、この類似の DNA は共通の祖先に由来している、と推論できることが多いものです。そうした類似の DNA のサブストリングを発見するために配列アラインメント・アルゴリズムを使用することができます。

ゲノミクス (genomics) の主要テーマは、DNA 配列を比較し、2 つの配列の共通部分の整列を試みることです。2 つの DNA 配列が (偶然起きたと思える以上に) 類似の部分列を共通に持っている場合、それらの配列は相同である可能性が高いと言えます (囲み記事「ホモロジー」を参照)。2 つの配列を整列させる際には、完全に一致している文字について検討するだけでなく、一方の配列の中の空白やギャップ (あるいは逆に、もう一方の配列への挿入)、また不一致も考慮します。空白やギャップ、不一致は、どれも変異に対応します。配列アラインメントでは、最適なアラインメントを発見しようとします。これを大まかに言えば、一致するものの数を最大にし、空白と不一致の数を最小にするということです。もっと正式に言えば、可能なアラインメントのそれぞれに対して、文字が一致する場合はポイントを加算し、空白および不一致の場合はポイントを減算することで、スコアを決めるということです。

グローバル配列アラインメントとローカル配列アラインメント

グローバル配列アラインメントでは、S1 という配列全体と S2 という別の配列全体との間で、最高のアラインメントを見つけようとします。次のような 2 つの DNA 配列を考えてみてください。

- S1 = `GCCCTAGCG`
- S2 = `GCGCAATG`

一致には 1 ポイントを与え、空白はマイナス 2 ポイント、不一致はマイナス 1 ポイントとすると、下記が最適のグローバル・アラインメントです。

- S1' = `GCCCTAGCG`
- S2' = `GCGC-AATG`

ダッシュ (-) は空白を表します。一致するものが 5 つあり、S2' に 1 つの空白があり (または逆に、S1' に 1 つの挿入があり)、そして 3 つの不一致があります。すると得られるスコアは (5 * 1) + (1 * -2) + (3 * -1) = 0 であり、これが最善の結果です。

ローカル配列アラインメントでは、両方の配列全体を整列するという制限はなく、単に各配列の一部を使って最大のスコアを得ます。先ほどと同じ S1 と S2 という配列を使い、またスコアの仕組みも同じとすると、最適なローカル・アラインメント、S1" と S2" として、下記が得られます。

- `S1 = GCCCTAGCG`
- `S1'' = GCG`
- `S2'' = GCG`
- `S2 = GCGCAATG`

このローカル・アラインメントには偶然にも不一致や空白がありませんが、通常はローカル・アラインメントには不一致や空白があります。このローカル・アラインメントのスコアは (3 * 1) + (0 * -2) + (0 * -1) = 3 です (最高のローカル・アラインメントのスコアは最高のグローバル・アラインメントのスコアと同じか、またはそれ以上です。これはグローバル・アラインメントがローカル・アラインメントでもあるためです。)

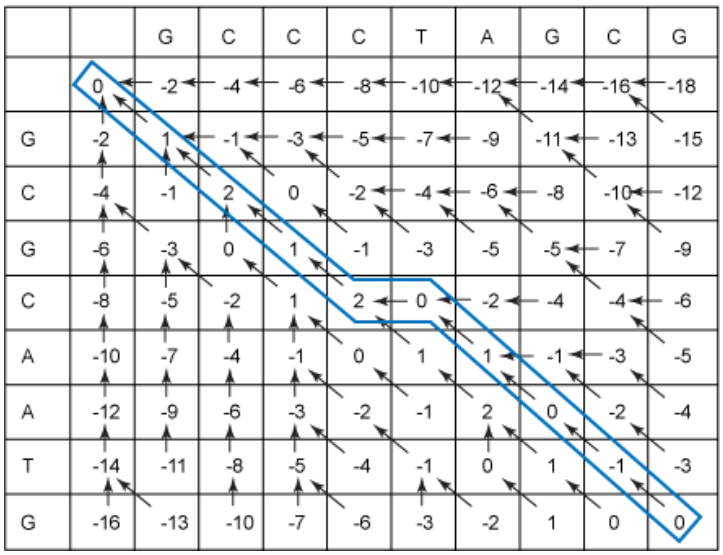
Needleman-Wunsch アルゴリズム

Needleman-Wunsch アルゴリズムは、グローバル・アラインメントを計算するために使われます。Needleman-Wunsch アルゴリズムの考え方は LCS アルゴリズムと似ています。この場合も 2 次元のテーブルを使い、1 つの配列を第 1 行目に、もう 1 つの配列を第 1 列目に置きます。そしてこの場合も、各セルに到達する方法は次の 3 つのうちのどれか 1 つです。

- 上のセルから (これは空白を使って左にある文字を整列することに対応します)
- 左側のセルから (これは空白を使って上にある文字を整列することに対応します)
- 対角線上の左上のセルから (これは左と上にある文字を整列することに対応しますが、この 2 つの文字は一致することも一致しないこともあります)

最初にテーブル全体を示します (図 7)。これからこのテーブルにデータを入力する方法を説明しますので、それを読みながらこのテーブルを参照してください。

図 7. データが入力された Needleman-Wunsch テーブルとトレースバック



まず、テーブルを初期化する必要があります。これは、2 行目と 2 列目にスコアとポインターを入力するということです。2 行目を右に移動することは、最上部にある最初の配列の中の文字を使うこと、また空白を使うことに対応し、左側にある配列の最初の文字を使うことに対応するわけではありません。空白による減点は -2 なので、この手順を行うごとに、前のセルに -2 を加算します。この、前のセルというのは、左側にあるセルです。2 行目の 0, -2, -4, -6, … というシーケンスは、このようにして得られています。同様に、2 列目のスコアとポインターも得られます。リスト 10 は Needleman-Wunsch アルゴリズムの初期化コードを示しています。

リスト 10. Needleman-Wunsch の初期化コード

```
protected Cell getInitialPointer(int row, int col) {
   if (row == 0 && col != 0) {
      return scoreTable[row][col - 1];
   } else if (col == 0 && row != 0) {
      return scoreTable[row - 1][col];
   } else {
      return null;
   }
}

protected int getInitialScore(int row, int col) {
   if (row == 0 && col != 0) {
      return col * space;
   } else if (col == 0 && row != 0) {
      return row * space;
   } else {
      return 0;
   }
}
```

次に、残りのセルにデータを入力する必要があります。LCS アルゴリズムの場合と同じく、各セルには 3 つの選択肢があり、そのなかから最大のスコアとなるものを選択します。各セルに到達する方向には、上から、左から、あるいは左上から、という 3 つの方向があります。ここで、整列を行う対象の文字列を S1 と S2、整列を行った結果の文字列を S1' と S2' としましょう。あるセルに上から来るということは、左にある S2 の文字を S2' に追加することと同じであり、またこの場合は上にある S1 の文字をスキップし、そして S1' には空白を挿入します。空白のスコアは -2 なので、現在のセルのスコアは上のセルから 2 を引くことで得られます。同様に、左から空白セルに入る場合には左のセルのスコアから 2 を引きます。最後に、上側にある文字を S1' に、また左側にある文字を S2' に追加することもできますが、これは左上から空白セルに入ることに相当します。この 2 つの文字が一致する場合には、新しいスコアは左上のセルのスコアに 1 を加えた値であり、2 つの文字が一致しない場合には、新しいスコアは左上のセルのスコアから 1 を引いた値です。このようにして、3 つの選択肢の中から最大のスコアを得られるものを選択します (最大のスコアとなるセルが複数ある場合は、そのなかから任意のセルを選択します)。図 7 のポインターには、こうした 3 つの選択肢の例がすべて見つかります。

リスト 11 は空白セルを埋めるためのコードを示しています。

リスト 11. テーブルを埋めるための Needleman-Wunsch コード

```
protected void fillInCell(Cell currentCell, Cell cellAbove, Cell cellToLeft,
      Cell cellAboveLeft) {
   int rowSpaceScore = cellAbove.getScore() + space;
   int colSpaceScore = cellToLeft.getScore() + space;
   int matchOrMismatchScore = cellAboveLeft.getScore();
   if (sequence2.charAt(currentCell.getRow() - 1) == sequence1
         .charAt(currentCell.getCol() - 1)) {
      matchOrMismatchScore += match;
   } else {
      matchOrMismatchScore += mismatch;
   }
   if (rowSpaceScore >= colSpaceScore) {
      if (matchOrMismatchScore >= rowSpaceScore) {
         currentCell.setScore(matchOrMismatchScore);
         currentCell.setPrevCell(cellAboveLeft);
      } else {
         currentCell.setScore(rowSpaceScore);
         currentCell.setPrevCell(cellAbove);
```

```
      }
   } else {
      if (matchOrMismatchScore >= colSpaceScore) {
         currentCell.setScore(matchOrMismatchScore);
         currentCell.setPrevCell(cellAboveLeft);
      } else {
         currentCell.setScore(colSpaceScore);
         currentCell.setPrevCell(cellToLeft);
      }
   }
}
```

次に、実際にアラインメントを行った結果のストリング (S1' と S2') とアラインメントのスコアを得る必要があります。一番右下のセルのスコアは S1 と S2 に対するアラインメントの最高スコアです (これは LCS アルゴリズムで一番右下のセルが LCS の長さを含むのと同じです)。また LCS アルゴリズムと同様、S1' と S2' を得るためには、この一番右下のセルからポインターに従って逆方向にトレースバックし、S1' と S2' を作成します。テーブルの作成方法からわかるように、下に進むことは、左にある S2 の文字を S2' に追加し、空白を S1' に追加することに対応します。右に進むことは、上にある S1 の文字を S1' に追加し、空白を S2' に追加することに対応します。また右下に進むことは、S1 と S2 の文字をそれぞれ S1' と S2' に追加することに対応します。

この結果、Needleman-Wunsch に使用するトレースバック・コードは、Smith-Waterman でローカル・アラインメントに使用するトレースバック・コードと同じになりますが、トレースバックをどのセルから開始し、どこでトレースバックを終了するべきなのかを判断する方法が異なります。リスト 12 は、この 2 つのアルゴリズムに共通するコードを示しています。

リスト 12. Needleman-Wunsch と Smith-Waterman の両方に使われるトレースバック・コード

```
protected Object getTraceback() {
   StringBuffer align1Buf = new StringBuffer();
   StringBuffer align2Buf = new StringBuffer();
   Cell currentCell = getTracebackStartingCell();
   while (traceBackIsNotDone(currentCell)) {
      if (currentCell.getRow() - currentCell.getPrevCell().getRow() == 1) {
         align2Buf.insert(0, sequence2.charAt(currentCell.getRow() - 1));
      } else {
         align2Buf.insert(0, '-');
      }
      if (currentCell.getCol() - currentCell.getPrevCell().getCol() == 1) {
         align1Buf.insert(0, sequence1.charAt(currentCell.getCol() - 1));
      } else {
         align1Buf.insert(0, '-');
      }
      currentCell = currentCell.getPrevCell();
   }

   String[] alignments = new String[] { align1Buf.toString(),
         align2Buf.toString() };

   return alignments;
}

protected abstract boolean traceBackIsNotDone(Cell currentCell);

protected abstract Cell getTracebackStartingCell();
```

リスト 13 は Needleman-Wunsch に特有のトレースバック・コードを示しています。

リスト 13. Needleman-Wunsch のトレースバック・コード

```
protected boolean traceBackIsNotDone(Cell currentCell) {
   return currentCell.getPrevCell() != null;
}

protected Cell getTracebackStartingCell() {
   return scoreTable[scoreTable.length - 1][scoreTable[0].length - 1];
}
```

当初の Needleman-Wunsch アルゴリズム

厳密に言うと、私が示しているものは Needleman-Wunsch アルゴリズムではありません。Needleman-Wunsch が発表した当初のアルゴリズムは 3 次元時間で実行されるものであり、もはや使われていません。それでもなお、この記事で説明している 2 次元のアルゴリズムは、一般に Needleman-Wunsch アルゴリズムと呼ばれています。

トレースバックを行うと、このセクションの最初で説明した最適グローバル・アラインメントが次のように得られます。

- S1' = `GCCCTAGCG`
- S2' = `GCGC-AATG`

このアルゴリズムの実行には、明らかに O(mn) 時間かかります。

Smith-Waterman アルゴリズム

Smith-Waterman アルゴリズムでは、配列全体を整列するという制約がありません。この事実と、長さゼロの 2 つのストリングはスコア 0 のローカル・アラインメントであるという事実から、Smith-Waterman アルゴリズムの場合、ローカル・アラインメントを作成する際に、マイナスのスコアとなることがありません。したがって、さらに整列を進めることによって、長さゼロの 2 つのストリングで「リセットする」ことで得られるスコアよりも低いスコアが得られます。また、いずれかの配列の最後でローカル・アラインメントが終わらなければならないということもありません。そのためトレースバックを右下のコーナーから開始する必要はなく、最高のスコアを持つセルから開始することができます。

このことから、Smith-Waterman アルゴリズムは Needleman-Wunsch アルゴリズムと 3 つの点で異なることになります。第 1 に、Smith-Waterman アルゴリズムでは、初期化の段階で 1 行目と 1 列目にはすべて 0 が入力されます (また 1 行目と 1 列目のポインターはすべてヌルです)。リスト 14 は Smith-Waterman の初期化コードを示しています。

リスト 14. Smith-Waterman の初期化コード

```
protected int getInitialScore(int row, int col) {
   return 0;
}

protected Cell getInitialPointer(int row, int col) {
   return null;
}
```

第 2 に、Smith-Waterman アルゴリズムではテーブルにデータを入力する際、スコアがマイナスになったらマイナスの代わりに 0 を入力し、また発生元を示すポインターを、スコアがプラスのセルに対してのみ追加します。リスト 15 では、どのセルのスコアが高いかも追跡していることに注目してください (これはトレースバックのために必要です)。

リスト 15. セルに入力するための Smith-Waterman のコード

```
protected void fillInCell(Cell currentCell, Cell cellAbove, Cell cellToLeft,
      Cell cellAboveLeft) {
   int rowSpaceScore = cellAbove.getScore() + space;
   int colSpaceScore = cellToLeft.getScore() + space;
   int matchOrMismatchScore = cellAboveLeft.getScore();
   if (sequence2.charAt(currentCell.getRow() - 1) == sequence1
         .charAt(currentCell.getCol() - 1)) {
      matchOrMismatchScore += match;
   } else {
      matchOrMismatchScore += mismatch;
   }
   if (rowSpaceScore >= colSpaceScore) {
      if (matchOrMismatchScore >= rowSpaceScore) {
         if (matchOrMismatchScore > 0) {
            currentCell.setScore(matchOrMismatchScore);
            currentCell.setPrevCell(cellAboveLeft);
         }
      } else {
         if (rowSpaceScore > 0) {
            currentCell.setScore(rowSpaceScore);
            currentCell.setPrevCell(cellAbove);
         }
      }
   } else {
      if (matchOrMismatchScore >= colSpaceScore) {
         if (matchOrMismatchScore > 0) {
            currentCell.setScore(matchOrMismatchScore);
            currentCell.setPrevCell(cellAboveLeft);
         }
      } else {
         if (colSpaceScore > 0) {
            currentCell.setScore(colSpaceScore);
            currentCell.setPrevCell(cellToLeft);
         }
      }
   }
   if (currentCell.getScore() > highScoreCell.getScore()) {
      highScoreCell = currentCell;
   }
}
```

最後に、Smith-Waterman アルゴリズムではトレースバックを行う際、最高のスコアを持つセルから開始し、スコア 0 のセルに到達するまでトレースバックを行います。それ以外の点では、トレースバックは Needleman-Wunsch アルゴリズムとまったく同じように動作します。リスト 16 は Smith-Waterman のトレースバック・コードを示しています。

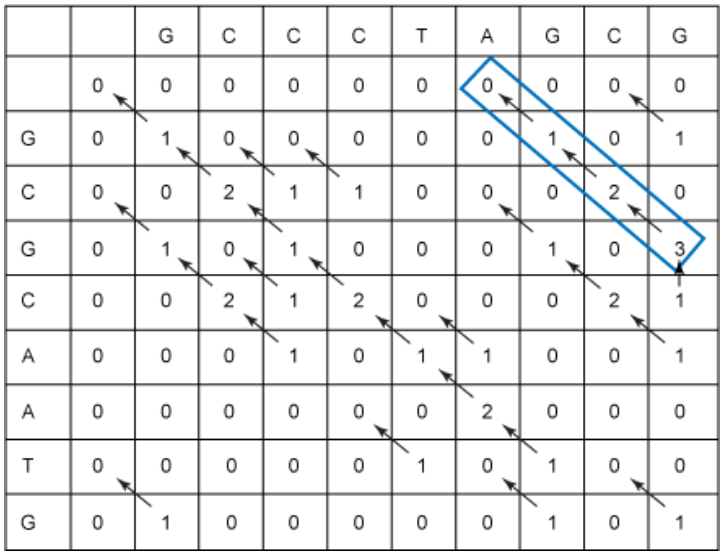
リスト 16. Smith-Waterman のトレースバック・コード

```
protected boolean traceBackIsNotDone(Cell currentCell) {
   return currentCell.getScore() != 0;
}

protected Cell getTracebackStartingCell() {
   return highScoreCell;
}
```

図 8 は、この記事を通して使用している配列 S1 と S2 に対して Smith-Waterman アルゴリズムを実行した様子を示しています。

図 8. データが入力された Smith-Waterman テーブルとトレースバック



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	1	0	1
C	0	0	2	1	1	0	0	0	2	0
G	0	1	0	1	0	0	0	1	0	3
C	0	0	2	1	2	0	0	0	2	1
A	0	0	0	1	0	1	1	0	0	1
A	0	0	0	0	0	0	2	0	0	0
T	0	0	0	0	0	1	0	1	0	0
G	0	1	0	0	0	0	0	1	0	1

Needleman-Wunsch アルゴリズムの場合と同様、Smith-Waterman コードを実行したことにより得られる (あるいは図 8 を読み取ることで得られる) 最適なローカル・アラインメントは次のようになります。

- `S1 = GCCCTAGCG`
- `S1'' = GCG`
- `S2'' = GCG`
- `S2 = GCGCAATG`

ALIGN、FASTA、そして BLAST

この記事で示しているのは、最適化を行わずにグローバル・アラインメントとローカル・アラインメントを O(mn) 時間で発見するための、Needleman-Wunsch アルゴリズムと Smith-Waterman アルゴリズムの基本実装です。通常、実際の研究者は、2 つの配列を比較するのではなく、ある特定の配列に類似のすべての配列を見つけようとします。もし見つかった類似の配列の 1 つが既知の生物学的機能を持っている場合には、その元となった配列が類似の機能を持っている可能性が高いことになります。類似の配列は類似の機能を持つ可能性が高いからです。ALIGN と FASTA、そして BLAST (Basic Local Alignment Search Tool) は産業向けのアプリケーションであり、ALIGN はグローバル・アラインメントを求めるために、また FASTA と BLAST はローカル・アラインメントを求めるために使われます。BLAST はユーザーが入力する配列に類似した (そして場合によっては相同な) 配列に対する大規模な配列データベースを検索し、類似性に従ってランク付けを行います (「参考文献」を参照)。BLAST は Smith-Waterman を直接使用するわけではありません。これは、たとえ 2 次元実行時間であっても、遺伝子配列の非常に大規模なデータベースの各配列に対して 1 つの配列を比較したのでは遅すぎるからです (各遺伝子配列は 30 億 (あるいはそれ以上) もの塩基対で構成されている可能性があります)。BLAST はその代わり、まずシーディング (seeding) と呼ばれるプロセスを使って、シードを見つけます (シードは、一致する可能性のあるもの、つまりヒットとなりうるものの先頭部分です)。次に BLAST は動的計画法のアルゴリズムを使うことで、(見つかった) ヒットとなりうるものを、入力配列による実際のローカル・アラインメントにまで拡張します。そして最後に、どの一致が統計的に重要かを判断し、それらの一致をランク付けします。この、部分的にヒューリスティックなプロセスは Smith-Waterman ほど精度が良く (正確では) ありませんが、ずっと高速です。

BioJava

BioJava は生物学のデータを処理するための Java フレームワークを開発する、オープンソースのプロジェクトです。このプロジェクトには、生物学上の配列を操作するためのオブジェクトや、配

列解析を行うための GUI、そして動的計画法のツールキットを含む解析や統計のルーチンなどが含まれています (「参考文献」を参照)。

リスト 17 は、この記事の例で使用してきたものと同じ配列そして同じスコア方式で Needleman-Wunsch アルゴリズムと Smith-Waterman アルゴリズムの BioJava 実装を実行する方法を示しています。

リスト 17. BioJava による配列アラインメント用のコード (Andreas Dräger による BioJava のサンプル・コードをベースにしています)

```
// The alphabet of the sequences. For this example DNA is chosen.
FiniteAlphabet alphabet = (FiniteAlphabet) AlphabetManager
      .alphabetForName("DNA");
// Use a substitution matrix with equal scores for every match and every
// replace.
int match = 1;
int replace = -1;
SubstitutionMatrix matrix = new SubstitutionMatrix(alphabet, match,
      replace);
// Firstly, define the expenses (penalties) for every single operation.
int insert = 2;
int delete = 2;
int gapExtend = 2;
// Global alignment.
SequenceAlignment aligner = new NeedlemanWunsch(match, replace, insert,
      delete, gapExtend, matrix);
Sequence query = DNATools.createDNASequence("GCCCTAGCG", "query");
Sequence target = DNATools.createDNASequence("GCGCAATG", "target");
// Perform an alignment and save the results.
aligner.pairwiseAlignment(query, // first sequence
      target // second one
      );
// Print the alignment to the screen
System.out.println("Global alignment with Needleman-Wunsch:\n"
      + aligner.getAlignmentString());

// Perform a local alignment from the sequences with Smith-Waterman.
aligner = new SmithWaterman(match, replace, insert, delete, gapExtend,
      matrix);
// Perform the local alignment.
aligner.pairwiseAlignment(query, target);
System.out.println("\nLocal alignment with Smith-Waterman:\n"
      + aligner.getAlignmentString());
```

BioJava による方法は、これらの例に対して、もう少し汎用性があります。第 1 に、`SubstitutionMatrix` が使われていることに注目してください。これまでに示した例では単純に、DNA 塩基間の不一致に対する減点は同じと見なしていました (例えば G が A に変異する可能性は C が A に変異する可能性と同程度、など)。しかし実際の生物学の配列では、特に蛋白質のアミノ酸では、そんなことはありません。そこで、あり得ない不一致の減点を、あり得る不一致の減点よりも大きくする必要があります。置換行列を使うと、シンボルの対それぞれに対して一致スコアを個別に割り当てることができます。ある意味で、置換行列は化学的な性質をコード化します。例えば BLAST 検索では蛋白質用の BLOSUM (BLOcks SUbstitution Matrix) 行列がよく使われますが、BLOSUM 行列の中の値は経験的に決定されています。

次に、単に 1 つの空白によるスコアではなく `insert` と `delete` というスコアが使われていることに注目してください。先ほど触れたように、空白については、空白のない配列での挿入 (`insert`)、あるいは空白のある配列での欠失 (`delete`) と考えることもできます。一般的に、2 つ

の配列を比較する方法には、互いに補完し合う 2 つの方法があります。これまで見てきた方法では、2 つの配列を「静的に」見ることで両者の違いを調べました。しかし 2 つの配列を比較する方法としては、一方の配列をもう一方の配列に変換するために個々のシンボルの挿入、欠失、置換を最低限何回行う必要があるかを求める方法もあります。この最小変更回数は編集距離 (edit distance) と呼ばれます。編集距離を計算する際には、挿入と欠失に異なる値を割り当てる必要があるかもしれません。例えば、挿入はより一般的なので、欠失よりも減点を少なくする必要があるかもしれません。

バイオインフォマティクスで Perl を使う

大規模なサーバーによるバイオインフォマティクス・ソフトウェアの大部分は C または C++ で作成されています。BLAST は元々 C で作成されていましたが、現在は C++ バージョンがあります。しかし研究者が作成する簡単なアプリケーションの多くは Perl で作成されています (研究者は多くの場合、生物学者としてはプロかもしれませんが、プログラマーとしての技術はそこまでは高くないものです)。おそらく、その理由は、オープンソースのバイオインフォマティクス・ライブラリーとして最大の Bioperl が Perl で作成されているためです。バイオインフォマティクスのプログラミングの仕事を得たい人は、どこかの時点でおそらく Perl と Bioperl を学ぶ必要があります。

今度は `gapExtend` 変数に注目してください。本来、ギャップとは、空白が連続している部分を表します。しかし一部の文献では、実際には 1 つの空白を意味する場合にギャップという用語を使っています。これまでは、たとえ空白が大きなギャップの一部である場合であっても、すべての空白に同じスコアを与えてきました。しかし自然界では、いったんギャップが開始されたら別の空白でそのギャップが拡張される可能性の方が、ギャップが新たに開始される可能性よりも高いものです。従って、意味のある結果を得るためには、ギャップに後から追加される空白への減点を、そのギャップの最初の空白への減点よりも少なくする必要があります。`gapExtend` 変数はそのためにあります。アルゴリズムの観点から言えば、こうしたスコアの仕組みはやや無原則であることを頭に入れておく必要があります。しかし当然ながら、計算対象であるストリングの編集距離を可能な限り自然界の進化距離に近くなるように合わせる必要があります。(さまざまな状況に対して適切なスコアの仕組みを考え出すことは、それ自体が研究の副分野として非常に興味深く、また複雑です。)

最後に、変数 `insert`、`delete`、`gapExtend` は、マイナスのもの (コスト、あるいは罰金) として定義されるため、これまでに使用した負の値ではなく、正の値を取ります。

リスト 17 のコードを実行すると、次のような出力が得られます。

リスト 18. 出力

```
Global alignment with Needleman-Wunsch:

  Length:9
  Score:-0.0
  Query:query,Length:9
  Target:target,Length:8

Query:  1 gccctagcg 9
          || | |  |
Target: 1 gcgc-aatg 8


Local alignment with Smith-Waterman:
```

```
  Length:3
  Score:3.0
  Query:query,Length:9
  Target:target,Length:8

Query:  7 gcg 9
          |||
Target: 1 gcg 3
```

ローカル・アラインメントとグローバル・アラインメントの両方に対して、これまで得られたスコアと同じスコアが得られます。この方法による Smith-Waterman の実装では、先ほど得られたものと同じローカル・アラインメントが得られます、この方法による Needleman-Wunsch の実装で得られるグローバル・アラインメントは先ほど得られたものと異なりますが、スコアは同じです。しかしどちらのグローバル・アラインメントも最大グローバル・アラインメントです。テーブルにデータを入力する際、そのセルの前の複数のセルから最大スコアが得られる場合が時々あることを思い出してください。そのうちのどれを選択してセルにデータを入力するかによって、異なる (ただしどれも同じスコアを持つ) アラインメントの結果が得られます。

まとめ

この記事では、動的計画法を使うことで解決できる、3 つの問題の例を調べました。この 3 つの問題は、どれも次のような同じ特徴を持っています。

- それぞれの問題に対する解は、再帰的な関係を使って表現することができます。
- この再帰的な関係に対して再帰的な方法による初歩的な実装を行うと、副問題を何度も計算する非効率的な解になる可能性があります。
- 問題に対する最適解は、元々の問題の副問題に対する最適解から構成することができます。

動的計画法は、複数の行列の乗算や、組立ラインのスケジューリング、コンピューターによるチェスなどのプログラムにも使われます。プログラミング・コンテストの複雑な問題にも動的計画法が必要になることがよくあります。興味のある方のために、どのような場合に動的計画法を応用できるか、また動的計画法アルゴリズムの正しさは通常どのように証明されるのかの詳細が、『アルゴリズムイントロダクション』(「参考文献」を参照) に説明されています。

動的計画法は生物学におけるコンピューター・サイエンスの用途として最も重要かもしれませんが、もちろん動的計画法のみが重要なわけではありません。バイオインフォマティクスと計算生物学は学際的な分野であり、この分野自体が急速に専用の学術プログラムを持った専門分野になりつつあります。現在の多くの分子生物学者はプログラミングについてほんの少ししか知らず、少し生物学を勉強する気のあるプログラマーならば行えるような興味深く重要な作業が数多くあります。詳しく知りたい方のために、役立ちそうな資料を「参考文献」に挙げてあります。

謝辞

私が計算生物学に関心を持つきっかけを作ってくださった Sonna Bristle 氏と、この記事をレビューしてくださり、有益な助言をいただいた IBM の Carlos P. Sosa 氏に感謝いたします。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-seqalign-code.zip	12KB

著者について

Paul D. Reiners

Paul Reiners は Sun 認定の Java プログラマーであり Java 開発者です。彼は Automatous Monk や Twisted Life、Leipzig などいくつかのオープン・ソース・プログラムの開発者です。彼は University of Illinois at Urbana-Champaign で応用数学 (計算理論) で修士号を取得しています。彼はミネソタに住み、時間がある時には電子ベース・ギターを練習し、ペットのネズミである Fred と Mortimer と遊び、また TopCoder で競いあっています。