JFP **16** (1): 1–12, 2006.
doi:10.1017/S095679680500571X Printed in the United Kingdom

EDUCATIONAL PEARL

Biological sequence similarity

DAVID WAKELING
Bioinformatics Group, University of Exeter, Exeter EX4 4PT, UK
(*e-mail:* D.Wakeling@exeter.ac.uk)

Abstract

Functional languages provide an excellent framework for formulating biological algorithms in a naive form and then transforming them into an efficient form. This helps biologists understand what matters about programming and brings functional programming into the realm of the practical. In this column, we present an example from our MSc course on bioinformatics and report on our experiences teaching functional programming in this context.

1 Introduction

Many bioinformatics textbooks say relatively little about programming, and so many bioinformatics students do relatively little of it (Attwood & Parry-Smith, 1999; Krane & Raymer, 2002; Lesk, 2002). Indeed, Lesk even goes as far as to quote Bismarck: "Those who love sausages or the law should not watch either being made," and to suggest that computer programs should be added to this list. This is a pity because clear programs are often the best way to explain and experiment with algorithms such as those used in bioinformatics. Here, we illustrate this principle by developing some textbook algorithms for gauging biological sequence similarity using the functional language Haskell (Peyton Jones, 2003).

This paper is organised as follows. Section 2 sets out the background and goal of our work. Section 3 describes a representation of sequences. Section 4 presents a simple algorithm for gauging global sequence similarity, and Section 5 a more sophisticated one. Section 6 outlines some laboratory exercises. Section 7 reports on our early teaching experience. Section 8 concludes.

2 Background and goal

The Exeter University Bioinformatics MSc is a joint programme between the School of Biological and Chemical Sciences, and the School of Engineering, Computer Science and Mathematics. It was established in 1999 in response to a joint initiative by the UK Biotechnology and Biological Sciences Research Council (BBSRC) and Engineering and Physical Sciences Research Council (EPSRC) identifying the need for more trained researchers in bioinformatics.

In the academic year 2003/04, there were 25 students. Of these, 14 were from countries within the European union, and 11 were from elsewhere. All have at least a 2(i) first degree, in subjects that can be categorised as either "biology" (56%), "computer science" (16%), "medicine" (8%) or "other science and engineering" (20%).

The programme of study consists of six course modules run consecutively, followed by a supervised research project. Two of the course modules last for three weeks, two for four weeks, and two for six weeks. The supervised research project lasts for six months, and can be undertaken in an academic or an industrial environment. It is also possible to take the course by distance learning over two years instead of one. Half of the available credits are awarded for the course modules, and half for the research project. From the outset, the programme has been research led – staff are encouraged to link their teaching to their research wherever possible.

Two of the course modules are *Bioinformatics: Tools and Techniques*, lasting for four weeks, and *Biological Sequence Analysis and Structural Bioinformatics*, lasting for six weeks. The first module covers the relevant aspects of molecular and cellular biology, and the basics of Java (Arnold *et al*., 2000) programming (example textbooks: Lodish *et al*. (1999), Deitel & Deitel (2001)). The second module covers the algorithms used in bioinformatics software, and the use of Perl (Wall *et al*., 2000) to connect together such software to build larger applications (example textbooks: Lesk (2002), Tisdall (2001)). Both courses follow a similar teaching pattern: classroom sessions in the morning are complemented by laboratory sessions in the afternoon.

Although this arrangement works well enough, and gets the students programming as we would like, experience has shown that there are two problems.

1. There is a large "semantic gap" between the algorithms seen in the classroom and their implementations seen later in the laboratory. Students find bridging this gap to be an uncertain and frustrating business – if the results are not as expected, has the algorithm been misunderstood, or is the implementation incorrect?
2. There is an abrupt change of both programming and execution model between modules. On the one hand, Java is an object-oriented, strongly-typed language, and programs are usually compiled before being executed by an interpreter. On the other hand, Perl is a procedural[1], weakly-typed language, and programs are usually executed directly by an interpreter. Students are confused when ideas from one language do not carry over to the other.

Observing these problems, we argued that switching languages from Java and Perl to Haskell might reduce the size of the semantic gap, and avoid the abrupt change when moving from "programming" to "scripting". As part of this argument, we prepared a pilot lecture and laboratory session on sequence similarity. The goal was to establish that a functional programming language would be a suitable vehicle for explaining and experimenting with the standard algorithms used in bioinformatics.

[1] Although object-oriented programming is possible in Perl, at heart the language is procedural.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
data AminoAcid
 = Alanine    | Arginine
 | Asparagine | Aspartate
 | Cystine    | Glutamine
 | Glutamate  | Glycine
 | Histidine  | Isoleucine
 | Leucine    | Lysine
 | Methionine | Phenylalanine
 | Proline    | Serine
 | Threonine  | Tryptophan
 | Tryosine   | Valine
 deriving Eq
```

Fig. 1. The `AminoAcid` type.

Our experience of developing and delivering this pilot lecture and laboratory session is the subject of this paper.

3 Sequence representation

Much work in bioinformatics is based on *biological sequence data*. A *deoxyribonucleic acid* (DNA) sequence is one whose elements are *nucleic acids*, whereas a *protein* sequence is one whose elements are *amino acids*. For the sake of simplicity, we consider only the representation of protein sequences here; DNA sequences can be dealt with in much the same way.

Protein sequences are traditionally written as strings in which each amino acid is written as a character according to a standard encoding (Cornish-Bowden, 1985). However, although amino acids may be *written* as characters, they should not be *represented* by a character type because then the responsibility for checking that they are used correctly falls on the programmer and the run-time system, rather than on the compiler.

The `AminoAcid` type shown in Figure 1 provides a better representation of amino acids. A protein can then be represented by a list of `AminoAcids`. Deriving `Eq` allows `AminoAcids` to be compared for equality. Deriving `Read` and `Show` as well would allow them to be input and output using their full names. However, what biologists expect is the standard encoding, and to produce this we must give explicit instance declarations. Figure 2 shows the `Read` and `Show` instance declarations that can be used to input and output single `AminoAcids` and sequences of them.

4 Simple sequence similarity

Two DNA or protein sequences are *similar* if some suitable measure, such as the percentage identity of their corresponding elements, says that they are; they are *homologous* if they share a common ancestor. Searching large databases for similar sequences often leads to the discovery of homologous ones. These homologous

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
instance Read AminoAcid where
 readsPrec p (c:cs)
  = case c of
     { 'A' -> [ (Alanine,    cs) ]; 'R' -> [ (Arginine,      cs) ]
     ; 'N' -> [ (Asparagine, cs) ]; 'D' -> [ (Aspartate,     cs) ]
     ; 'C' -> [ (Cystine,    cs) ]; 'Q' -> [ (Glutamine,     cs) ]
     ; 'E' -> [ (Glutamate,  cs) ]; 'G' -> [ (Glycine,       cs) ]
     ; 'H' -> [ (Histidine,  cs) ]; 'I' -> [ (Isoleucine,    cs) ]
     ; 'L' -> [ (Leucine,    cs) ]; 'K' -> [ (Lysine,        cs) ]
     ; 'M' -> [ (Methionine, cs) ]; 'F' -> [ (Phenylalanine, cs) ]
     ; 'P' -> [ (Proline,    cs) ]; 'S' -> [ (Serine,        cs) ]
     ; 'T' -> [ (Threonine,  cs) ]; 'W' -> [ (Tryptophan,    cs) ]
     ; 'Y' -> [ (Tryosine,   cs) ]; 'V' -> [ (Valine,        cs) ]
     }

 readList []
  = [ ([], []) ]
 readList cs
  = [ (x:xs, cs2) | (x, cs1) <- reads cs, (xs, cs2) <- reads cs1 ]

instance Show AminoAcid where
 showsPrec p a
  = case a of
     { Alanine    -> showChar 'A' ; Arginine      -> showChar 'R'
     ; Asparagine -> showChar 'N' ; Aspartate     -> showChar 'D'
     ; Cystine    -> showChar 'C' ; Glutamine     -> showChar 'Q'
     ; Glutamate  -> showChar 'E' ; Glycine       -> showChar 'G'
     ; Histidine  -> showChar 'H' ; Isoleucine    -> showChar 'I'
     ; Leucine    -> showChar 'L' ; Lysine        -> showChar 'K'
     ; Methionine -> showChar 'M' ; Phenylalanine -> showChar 'F'
     ; Proline    -> showChar 'P' ; Serine        -> showChar 'S'
     ; Threonine  -> showChar 'T' ; Tryptophan    -> showChar 'W'
     ; Tryosine   -> showChar 'Y' ; Valine        -> showChar 'V'
     }

 showList []
  = id
 showList (x:xs)
  = shows x . showList xs
```

Fig. 2. The `Read` and `Show` instances for the `AminoAcid` type.

sequences can reveal evolutionary relationships among genes, proteins or even entire species, as well as providing a basis for predicting the structure and function of proteins. All bioinformatics textbooks cover sequence similarity (Attwood & Parry-Smith, 1999; Krane & Raymer, 2002; Lesk, 2002).

One way to gauge the similarity of two sequences is to consider the series of *edit operations* required to convert from one to the other. An edit operation may either *insert* an element ($\mathscr{I}\ a$), *delete* an element ($\mathscr{D}\ a$) or *change* one element to another ($\mathscr{C}\ a\ b$). There are usually many series of edit operations that

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
data Op e = Insert e | Delete e | Change e e

edits :: Eq e => [e] -> [e] -> [[Op e]]
edits aseq@(a:as) bseq@(b:bs)
 = [ Change a b : ops | ops <- edits as bs   ] ++
   [ Delete a   : ops | ops <- edits as bseq ] ++
   [ Insert b   : ops | ops <- edits aseq bs ]
edits aseq []
 = [ [ Delete a | a <- aseq ] ]
edits [] bseq
 = [ [ Insert b | b <- bseq ] ]
```

Fig. 3. The `Op` type and `edits` function.

convert from one sequence to another. For example, five series that convert from `Q` to `PQ` are:

1. $\mathscr{C}$ `Q P`, $\mathscr{I}$ `Q`;
2. $\mathscr{D}$ `Q`, $\mathscr{I}$ `P`, $\mathscr{I}$ `Q`;
3. $\mathscr{I}$ `P`, $\mathscr{C}$ `Q Q`;
4. $\mathscr{I}$ `P`, $\mathscr{D}$ `Q`, $\mathscr{I}$ `Q`;
5. $\mathscr{I}$ `P`, $\mathscr{I}$ `Q`, $\mathscr{D}$ `Q`.

Figure 3 shows the `Op` type for edit operations and the function `edits` that computes the many series of edit operations that convert from one sequence to another.

The constructors are parameterised by the type of sequence elements. Of course, we expect sequence elements to be nucleic or amino acids. However, similar problems arise in other areas where they are either the lines of files (Miller & Myers, 1985), the characters on the screen of a display editor (Myers & Miller, 1989) or those making up a program (Wagner & Fischer, 1974). By parameterising the type and the functions that work on it, we can use them in these other areas too. One of the strengths of the Haskell type system is that it makes this so easy to do.

In general, the conversion from one sequence to another can be done in three ways: by changing the head of the first sequence to that of the second, followed by any of the possible ways of converting from the tail of the first sequence to that of the second; by deleting the head of the first sequence, followed by any of the possible ways of converting the tail of the first sequence to the second; or by inserting the head of the second sequence, followed by any of the possible ways of converting the first sequence to the tail of the second. In the case where one or other sequence is empty, insertions or deletions must be made as appropriate.

In bioinformatics, a series of edit operations is traditionally displayed as an *alignment* that should be read column-wise. The five series given above, for example, are displayed as the alignments:

```
Q-      Q--     -Q      -Q-     --Q
PQ      -PQ     PQ      P-Q     PQ-.
```

It should be clear why, in this context, insertions and deletions are known as *gaps*. Figure 4 shows the function `alignment` for displaying alignments.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
alignment :: [Op AminoAcid] -> String
alignment ops
 = concat row1s ++ "\n" ++ concat row2s
   where
   (row1s, row2s) = unzip (map column ops)

   column (Insert a)
    = ("-", show a)
   column (Delete a)
    = (show a, "-")
   column (Change a b)
    = (show a, show b)
```

Fig. 4. The `alignment` function.

```
score :: Eq e => Op e -> Int
score (Change a b)
 | a == b = 1
score other
 = -1
```

Fig. 5. The `score` function.

Clearly, some series of edit operations are better than others. In order to choose between them, we can give each a *score*. A simple model defines the score for an operation to be

$$score(op) = \begin{cases} +1, & \text{if } op = \mathscr{C}\ a\ b \text{ and } a = b \\ -1, & \text{otherwise} \end{cases}$$

and that for a series to be the sum of its operation scores. Figure 5 shows the `score` function that computes the score for an operation. Using it, the score for a series is computed by `sum . map score :: Eq e => [Op e] -> Int`.

Figure 6 shows a main program function that reads two sequences and shows the alignment with the maximum score. The auxiliary function `maxBy` picks the maximum of a list of values according to some comparison function.

5 Global sequence similarity

So far, what we have shown is a nice example of how a lazy functional program can be conveniently developed as separate pieces that are then glued together: the function `edits` produces the many series of edit operations that convert from one sequence to another; the function `maxBy` consumes them and picks the one with the maximum score; and lazy evaluation ensures that the producer and consumer behave as coroutines. Hughes develops this idea more fully in Hughes (1989). The advantage (for the programmer) is that it is easier to produce the two functions separately, thinking about just one thing at a time. The disadvantage (for the machine) is that a list is needed to glue the functions together, and even with lazy evaluation, each value in this list must still be allocated, examined and deallocated.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
main :: IO ()
main
 = do { putStr "Enter first  sequence > "
      ; aseq <- readLn
      ; putStr "Enter second sequence > "
      ; bseq <- readLn
      ; putStr (alignment (maxBy (sum . map score) (edits aseq bseq)))
      }

maxBy :: Ord b => (a -> b) -> [a] -> a
maxBy f (x:xs)
 = loop x xs
   where
   loop u []
    = u
   loop u (v:vs)
    | f u >= f v = loop u vs
    | otherwise  = loop v vs
```

Fig. 6. A `main` program function.

```
edit :: Eq e => [e] -> [e] -> [Op e]
edit aseq@(a:as) bseq@(b:bs)
 = maxBy3 (sum . map score)
    (Change a b : edit as bs  )
    (Delete a   : edit as bseq)
    (Insert b   : edit aseq bs)
edit aseq []
 = [ Delete a | a <- aseq ]
edit [] bseq
 = [ Insert b | b <- bseq ]

maxBy3 :: Ord b => (a -> b) -> a -> a -> a -> a
maxBy3 f x y z
 | fx >= fy && fx >= fz = x
 | fy >= fx && fy >= fz = y
 | otherwise            = z
 where
 fx = f x
 fy = f y
 fz = f z
```

Fig. 7. The `edit` function.

5.1 Version 1

In this case, it is possible to dispense with the intermediate list by bringing the producer and consumer together. Figure 7 shows the function `edit`. This transformation has been called *deforestation* (Wadler, 1990). The auxiliary function `maxBy3` picks the maximum of three values according to some comparison function.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
tabulate :: Eq a => [a] -> [a] -> [Op a]
tabulate aseq bseq
 = edit aseq bseq n m
   where
   n = length aseq
   m = length bseq
   table   = array ((0,0), (n,m)) entries
   entries =
    [ ((i,j), edit as bs i j)
      | as <- tails aseq, let i = length as
      , bs <- tails bseq, let j = length bs ]

   edit aseq@(a:as) bseq@(b:bs) i j
    = maxBy3 (sum . map score)
        (Change a b : table ! (i-1, j-1))
        (Delete a   : table ! (i-1, j  ))
        (Insert b   : table ! (i,   j-1))
   edit aseq [] i j
    = [ Delete a | a <- aseq ]
   edit [] bseq i j
    = [ Insert b | b <- bseq ]
```

Fig. 8. The `tabulate` function.

5.2 Version 2

Sadly, the `edit` function is still inefficient. In computing the series of operations to convert from one sequence to another, it may recompute the series of operations to convert from one subsequence to another many times. The usual way to overcome this inefficiency is by *tabulation*. A table mapping from function argument values to previously computed result values is introduced. This table is consulted before each call. If the arguments are in the table, then the call is skipped and the result is returned. Otherwise, the call is performed, the arguments and result are added to the table, and the result is returned.

A number of general strategies for the tabulation of recursive programs have been examined by Bird (1980). In this case, the one we want is *overtabulation*. The table can simply be organised as a two-dimensional array, with the lengths i and j of the (sub)sequence arguments giving the location in the table where the result is to be found. Figure 8 shows the `tabulate` function that uses tabulation. Many will recognise it as an implementation of the *dynamic programming* algorithm (Cormen *et al.*, 2001). In a traditional programming language, one has to worry about order of initialisation and computation of the values in the array. In a functional language, however, one simply describes the table entries.

5.3 Version 3

Although the `tabulate` function is considerably more efficient than version 2, it still computes the same series many times. The problem is that in computing the score for the series of operations that converts from one sequence to another, it may

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

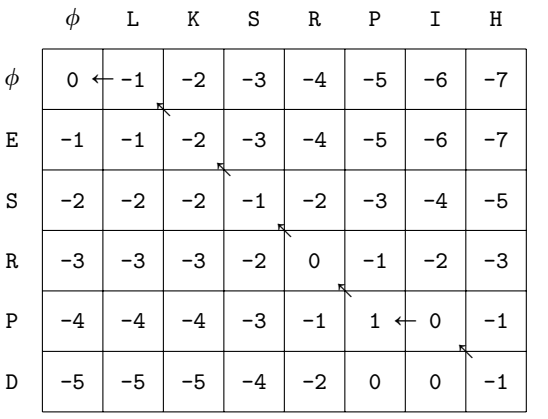

Fig. 9. Global traceback.

recompute the score for the series of operations that converts from one subsequence to another many times.

One way to overcome this inefficiency would be to make each entry in the table a *pair* of values consisting of the score for a series and the series itself. In textbook accounts, this is known as "storing a back pointer". Recomputation is avoided because a score always accompanies a series. Another way would be to make each entry in the table *only* a score, and to obtain the series by tracing back a route through the table, following maximum scores. In textbook accounts, this is known as "traceback". Recomputation is avoided because entries are scores rather than series from which they must be (re)computed.

To gauge the global similarity of two sequences, then, we shall create a table of scores and obtain a series of operations from this table by tracing back a route from the bottom-right corner of the table to the top-left corner, following maximum scores and recording operations on the way. Figure 9 shows a traceback when gauging the similarity of the sequences `DPRSE` and `HIPRSKL`. A diagonal move corresponds to a change operation, a horizontal move to an insert operation, and a vertical move to a delete operation. The global alignment is thus

```
D-PRSE-
HIPRSKL.
```

At some point in the traceback two, or even three moves/operations might be possible, each being part of a different, but equally good route/series. Notice that in textbooks, the sequences are in order and traceback produces the series of operations in reverse, whereas with our implementation, the sequences are in reverse and traceback produces the series of operations in order. Figure 10 shows the `global` and `traceback` functions that together implement this classic algorithm for gauging global sequence similarity due to (Needleman & Wunsch, 1970). The auxiliary `max3` function picks the maximum of three values.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

```
global :: Eq e => [e] -> [e] -> [Op e]
global aseq bseq
 = traceback table aseq bseq n m
   where
   n = length aseq
   m = length bseq
   table   = array ((0,0), (n,m)) entries
   entries =
    [ ((i, j), loop as bs i j)
      | as <- tails aseq, let i = length as
      , bs <- tails bseq, let j = length bs ]

   loop aseq@(a:as) bseq@(b:bs) i j
    = max3
       (score (Change a b) + table ! (i-1, j-1))
       (score (Delete a  ) + table ! (i-1, j  ))
       (score (Insert b  ) + table ! (i,   j-1))
   loop aseq [] i j
    = sum [ score (Delete a) | a <- aseq ]
   loop [] bseq i j
    = sum [ score (Insert b) | b <- bseq ]

traceback :: Eq e => Array (Int,Int) Int -> [e] -> [e] -> Int -> Int
                  -> [Op e]
traceback table aseq bseq n m
 = loop aseq bseq n m
   where
   loop aseq@(a:as) bseq@(b:bs) i j
    | v == p = Change a b : loop as   bs   (i-1) (j-1)
    | v == q = Delete a   : loop as   bseq (i-1) j
    | v == r = Insert b   : loop aseq bs   i     (j-1)
    where
    v = table ! (i, j)
    p = score (Change a b) + table ! (i-1, j-1)
    q = score (Delete a  ) + table ! (i-1, j  )
    r = score (Insert b  ) + table ! (i,   j-1)
   loop aseq [] i j
    = [ Delete a | a <- aseq ]
   loop [] bseq i j
    = [ Insert b | b <- bseq ]

max3 :: Ord a => a -> a -> a -> a
max3 x y z
 | x >= y && x >= z = x
 | y >= x && y >= z = y
 | otherwise        = z
```

Fig. 10. The `global` and `traceback` functions.

6 Exercises

The material presented here forms the foundation of a one hour classroom session followed by a one hour laboratory session. The laboratory session has three phases.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

1. *Familiarisation*. The exercises in this phase involve going through the full edit-compile-test cycle with the Hugs Haskell interpreter by trying out some small example sequences and modifying the function for scoring.
2. *Experimentation*. The exercises in this phase involve playing some bioinformatics "parlour games" inspired by examples in Lesk (2002). For example, based on the sequences of pancreatic ribonuclease from horse (*Equus Caballus*), minke whale (*Balaenoptera acutorostrada*), and red kangaroo (*Macropus rufus*), which of these species are most closely related? To answer this question, the sequences must be retrieved from an online database and input by hand; their similarity must then be gauged by inspecting alignments or comparing scores.
3. *Extension*. The exercise in this phase is to implement the textbook algorithm for *local alignment* by modifying the code for global alignment.

7 Experience

The pilot lecture and laboratory session described here have been given as part of a seminar series on research topics associated with the MSc programme. Of necessity, we took the approach of Abelson *et al.* (1985) – not formally teaching the language, but explaining as went along. This was possible because the students had already seen the algorithms in Java, adapted from a nice pedagogical implementation by Peter Sestoft of the Royal Veterinary and Agricultural University in Denmark. About half of the students managed to complete the (admittedly, highly directed) exercises in the laboratory, and all found it to be a positive experience.

Some of the staff who attended the lecture, though, were less convinced that a switch should be made from Java and Perl to Haskell. Broadly speaking, the computer scientists were concerned that students would no longer do any "real" programming; the biologists feared that they would be unable to undertake projects that involved interfacing with existing software packages; and both that there might be problems with student recruitment and subsequent employment. These, of course, are all part of the reason why no one uses functional languages (Wadler, 1998).

8 Conclusions

In this paper, we have developed some textbook algorithms for gauging biological sequence similarity in Haskell. Our early experience teaching bioinformatics students, albeit only as part of a research seminar series, has been positive. Some staff, however, have still to be convinced, and to do so we now plan to run a number of student research projects using a functional language.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1985) *Structure and Interpretation of Computer Programs*. MIT Press.

Arnold, K., Gosling, J. and Holmes, D. (2000) *The Java Programming Language (3rd ed)*. Addison-Wesley.

https://doi.org/10.1017/S095679680500571X Published online by Cambridge University Press

Attwood, T. K. and Parry-Smith, D. J. (1999) *Introduction to Bioinformatics*. Prentice Hall.

Bird, R. S. (1980) Tabulation techniques for recursive programs. *Comput. Surv.* **12**(4), 404–417.

Cormen, T. H., Clifford, S., Leiserson, C. E. and Rivest, R. L. (2001) *Introduction to Algorithms*. MIT Press.

Cornish-Bowden, A. (1985) Nomenclature for incompletely specified bases in nucleic acid sequences: Recommendations 1984. *Nucleic Acids Res.* **13**, 3021–3030.

Deitel, H. M. and Deitel, P. J. (2001) *Java: How to program*. Prentice Hall.

Hughes, J. (1989) Why functional programming matters. *Comput. J.* **32**(2), 98–107.

Krane, D. E. and Raymer, M. L. (2002) *Fundamental Concepts of Bioinformatics*. Addison Wesley.

Lesk, A. M. (2002) *Introduction to Bioinformatics*. Oxford University Press.

Lodish, H., Berk, A., Zipursky, L., Matsudaria, P., Baltimore, D. and Darnell, J. (1999) *Molecular Cell Biology (4th ed)*. W. H. Freeman.

Miller, W. and Myers, E. W. (1985) A file comparison program. *Software – Practice & Exper.* **15**(11), 1025–1040.

Myers, E. W. and Miller, W. (1989) Row replacement algorithms for screen editors. *ACM Trans. Program. Lang. & Syst.* **11**(1), 33–56.

Needleman, S. and Wunsch, C. (1970) A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Molecular Biol.* **48**, 444–453.

Peyton Jones, S. (2003) Haskell 98 language and libraries: The revised report. *J. Funct. Program.* **13**(1), 1–147.

Tisdall, J. (2001) *Beginning Perl for Bioinformatics*. O'Rielly.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* **73**, 231–248.

Wadler, P. (1998) How enterprises use functional languages, and why they don't. In: Apt, K. R., Marek, V. W., Truszczynsk, M. and Warren, D. S. (editors), *The Logic Programming Paradigm: A 25-year perspective*, pp. 209–227. Springer-Verlag.

Wagner, R. A. and Fischer, M. J. (1974) The string-to-string correction problem. *J. ACM*, **21**(1), 168–173.

Wall, L., Christiansen, T. and Orwant, J. (2000) *Programming Perl*. O'Rielly.