Definition

The whole genome of an organism can be revealed from tissue samples by using one of several DNA sequencing technologies, each of them producing a large number of DNA fragments of various lengths that are then assembled into the DNA sequence of the molecules in either the mitochondria or the nucleus (for eukaryotes) or in the cytoplasm (for prokaryotes) of the cells.

The whole genomes of thousands of extant species have already been sequenced, including 111 archaeal genomes ranging from 1,668 to 5,751,492 nucleotides; 2,167 bacterial genomes with 846 to 13,033,779 nucleotides; 2,593 eukaryote genomes with 1,028 to 748,055,161 nucleotides; 2,651 viral genomes with 200 to 1,181,404 nucleotides; 39 viroid RNA genomes with 246 to 399 nucleotides; and 1504 plasmid genomes with 846 to 2,094,509 nucleotides.

Extant species represent only a small fraction of the genetic diversity that has ever existed, however, and whole genomes of extinct species can also be sequenced from well-conserved tissue samples.

Definition

Once the genome of a species has been sequenced, one of the first steps towards understanding it consists in the identification of genes coding for proteins.

In prokaryotic genomes, the sequence coding for a protein occurs as one contiguous open reading frame, while in eukaryotic genomes, it is often spliced into several coding exons separated by non-coding introns, and these exons can be combined in different arrangements to code for different proteins by the cellular process of alternative splicing.

Definition

Protein coding regions of a DNA sequence are first transcribed into messenger RNA and then translated into protein.

A codon of three DNA nucleotides is transcribed into a codon of three complementary RNA nucleotides, which is translated in turn into a single amino acid within a protein.

A fragment of single-stranded DNA sequence has three possible reading frames, and translation takes place in an open reading frame, a sequence of codons from a certain start codon to a certain stop codon and containing no further stop codon.

The reading frame determines the actual amino acids encoded by a gene.

A fragment of double-stranded DNA sequence, on the other hand, has six possible reading frames, three in each direction.

An open reading frame begins with the start codon ATG (methionine) in most species and ends with a stop codon TAA, TAG, or TGA.

Example

The reading frame determines the actual amino acids encoded by a gene.

For instance, the DNA sequence fragment GTCGCCATGATGGTGGTTATTATACCGTCAAGGACTGTGTGACTA can be read in the 5′ to 3′ direction in the following three frames:

```
1 GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
2  TCG CCA TGA TGG TGG TTA TTA TAC CGT CAA GGA CTG TGT GAC TA
3   CGC CAT GAT GGT GGT TAT TAT ACC GTC AAG GAC TGT GTG ACT A
```

A fragment of double-stranded DNA sequence, on the other hand, has six possible reading frames, three in each direction.

An open reading frame begins with the start codon ATG (methionine) in most species and ends with a stop codon TAA, TAG, or TGA.

Definition

The identification of genes coding for proteins in a DNA sequence is a very difficult task.

Even a simple organism such as Bacteriophage ϕ-X174, with a single-stranded DNA sequence of only 5,368 nucleotides, has a total of 117 open reading frames, only 11 of which actually code for a protein.

There are several other biological signals that help the computational biologist in the task of gene finding, but to start with, the known protein with the shortest sequence has 8 amino acids and, thus, short open reading frames, with fewer than $3 + 24 + 3 = 30$ nucleotides, cannot code for a protein.

A first algorithmic problem consists in extracting all open reading frames in the three reading frames of a DNA sequence fragment.

The problem has to be solved on the reverse complement of the sequence as well if the DNA is double stranded.

Definition

Given a fragment of DNA sequence S of n nucleotides, let $S[i]$ denote the i-th nucleotide of sequence S, for $1 \leqslant i \leqslant n$.

Let also $S[i,\ldots,j]$, where $i \leqslant j$, denote the fragment of S containing nucleotides $S[i], S[i+1], \ldots, S[j]$.

An open reading frame is a fragment $S[i,\ldots,j]$, of length $j-i+1 \geqslant 30$, such that $S[i,\ldots,i+2]$ is the start codon ATG and $S[j-2,\ldots,j]$ is one of the stop codons TAA, TAG, or TGA, and that does not contain any other stop codon.

That is, it must also fulfill the condition $S[k,\ldots,k+2] \notin \{\text{TAA}, \text{TAG}, \text{TGA}\}$ for $i+3 \leqslant k \leqslant j-6$.

Example

In the sequence $S = \text{GTCGCCATGATGGTGGTTATTATACCGTCAAGGACT GTGTGACTA}$, which has $n = 45$ nucleotides,

- $S[1] = \text{G}$, $S[2] = \text{T}$, $S[3] = \text{C}$, and $S[n] = \text{A}$.
- $S[1, \ldots, 4] = \text{GTCG}$, and $S[1, \ldots, n] = S$.
- $S[7, \ldots, 42]$ is an open reading frame, as it begins with $S[7, \ldots, 9] = \text{ATG}$, ends with $S[40, \ldots, 42] = \text{TGA}$, and has no other codon between $S[10]$ and $S[39]$ equal to TAA, TAG, or TGA.

```
GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
```

Definition

The reading frame determines a partition of the DNA sequence fragment S in codons of three consecutive nucleotides.

In reading frame 1, the first codon is $S[1,\ldots,3]$, the second codon is $S[4,\ldots,6]$, and so on.

In reading frame 2, however, the first codon is $S[2,\ldots,4]$, and the second codon is $S[5,\ldots,7]$.

The first codon in reading frame 3 is $S[3,\ldots,5]$.

In a given reading frame, the codons can be accessed by sliding a window of length three over the sequence, starting at position 1, 2, or 3, depending on the reading frame.

Definition

The sliding window is thus a kind of looking glass under which a codon of the sequence can be seen and accessed:

```
    1-3 4-6 7-9 ... ... ... ... ... ... ... ... ... ... ..n
⟶   GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
    GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
    GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
    ...
    GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
    GTC GCC ATG ATG GTG GTT ATT ATA CCG TCA AGG ACT GTG TGA CTA
```

Algorithm

Consider the problem of finding an open reading frame in a reading frame of a sequence, and let $S[k, \ldots, k+2]$ be the codon under the sliding window in the given reading frame.

Starting with an initial position k given by the reading frame, the sliding window has to be displaced by three nucleotides each time until accessing a start codon and then continue sliding by three nucleotides each time until accessing a stop codon.

This is actually not quite the case. The reading frame of the sequence fragment could contain no start codon at all, or it could contain a start codon but no stop codon, and the search for the beginning or the end of an open reading frame might go beyond the end of the sequence.

Algorithm

The first start codon in the k-th reading frame of a given DNA sequence fragment S of n nucleotides can be found by sliding a window $S[i, \ldots, i+2]$ of three nucleotides along $S[k, \ldots, n]$, until either $i+2 > n$ or $S[i, \ldots, i+2] = \text{AGT}$.

The initial position i of the candidate start codon is increased by three as long as the codon does not fall off the sequence (that is, $i+2 \leqslant n$) and is not a start codon (that is, $S[i, \ldots, i+2] \neq \text{AGT}$).

$i \leftarrow k$
while $i+2 \leqslant n$ **and** $S[i, \ldots, i+2] \neq \text{AGT}$ **do**
    $i \leftarrow i+3$
if $i+2 \leqslant n$ **then**
    output $S[i, \ldots, i+2]$

Algorithm

After having found a start codon $S[i, \ldots, i+2]$, the first stop codon can be found by sliding a window $S[j, \ldots, j+2]$ of three nucleotides, this time along $S[i+3, \ldots, n]$, until either $j+2 > n$ or $S[j, \ldots, j+2] \in \{\text{TAA}, \text{TAG}, \text{TGA}\}$.

The initial position j of the candidate stop codon is increased by three as long as the codon does not fall off the sequence (that is, $j+2 \leqslant n$) and the candidate codon is not a stop codon (that is, with $S[j, \ldots, j+2] \notin \{\text{TAA}, \text{TAG}, \text{TGA}\}$).

$j \leftarrow i+3$
while $j+2 \leqslant n$ **and** $S[j, \ldots, j+2] \notin \{\text{TAA}, \text{TAG}, \text{TGA}\}$ **do**
    $j \leftarrow j+3$
if $j+2 \leqslant n$ **then**
    output $S[j, \ldots, j+2]$

Algorithm

Now, the problem of extracting the first open reading frame in the k-th reading frame of a DNA sequence fragment S of length n can be solved by putting together the search for a start codon and the search for a stop codon.

The start codon is $S[i,\ldots,i+2]$ and the stop codon is $S[j,\ldots,j+2]$ of the sequence and, thus, the open reading frame $S[i,\ldots,j+2]$ is output.

$i \leftarrow k$
while $i+2 \leqslant n$ **and** $S[i,\ldots,i+2] \neq \mathrm{AGT}$ **do**
    $i \leftarrow i+3$
if $i+2 \leqslant n$ **then**
    $j \leftarrow i+3$
    while $j+2 \leqslant n$ **and** $S[j,\ldots,j+2] \notin \{\mathrm{TAA},\mathrm{TAG},\mathrm{TGA}\}$ **do**
        $j \leftarrow j+3$
    if $j+2 \leqslant n$ **then**
        output $S[i,\ldots,j+2]$

Algorithm

Notice that only the first start codon is found, and the first stop codon after this start codon will then signal the end of the first open reading frame.

There may be other start codons in the sequence fragment between the first start codon and the first stop codon, however, which would signal shorter open reading frames contained in the first open reading frame found.

Also, the first open reading frame might be shorter than 30 nucleotides, much too short to actually code for a protein.

Algorithm

The problem of extracting all open reading frames of at least 30 nucleotides in the k-th reading frame of a DNA sequence fragment S of length $n \geqslant 30$ can be solved by repeating the previous procedure for each start codon found in turn, checking that the open reading frames thus found have at least 30 nucleotides.

$i \leftarrow k$
while $i+2 \leqslant n$ **do**
    if $S[i,\ldots,i+2] = \mathrm{AGT}$ **then**
        $j \leftarrow i+3$
        while $j+2 \leqslant n$ **and** $S[j,\ldots,j+2] \notin \{\mathrm{TAA}, \mathrm{TAG}, \mathrm{TGA}\}$ **do**
            $j \leftarrow j+3$
        if $j+2 \leqslant n$ **then**
            if $j+2-i+1 \geqslant 30$ **then**
                output $S[i,\ldots,j+2]$
    $i \leftarrow i+3$

Algorithm

Finally, the problem of extracting all open reading frames of at least 30 nucleotides in the three reading frames of a DNA sequence fragment S of length $n \geqslant 30$ can be solved by repeating the previous procedure for each reading frame and for each start codon in turn, checking again that the open reading frames thus found have at least 30 nucleotides.

The whole algorithm is wrapped into a procedure that takes the DNA sequence fragment S as input and reports each of the open reading frames of S as output.

Algorithm

procedure extract_open_reading_frames(S)
    $n \leftarrow \text{length}(S)$
    for $i \leftarrow 1, 2, 3$ **do**
        while $i + 2 \leqslant n$ **do**
            if $S[i, \ldots, i+2] = \text{AGT}$ **then**
                $j \leftarrow i + 3$
                while $j + 2 \leqslant n$ **and** $S[j, \ldots, j+2] \notin \{\text{TAA}, \text{TAG}, \text{TGA}\}$ **do**
                    $j \leftarrow j + 3$
                if $j + 2 \leqslant n$ **then**
                    if $j + 2 - i + 1 \geqslant 30$ **then**
                        output $S[i, \ldots, j+2]$
            $i \leftarrow i + 3$

Algorithm

A related algorithmic problem consists of finding the longest open reading frame of a given DNA sequence fragment.

The longest open reading frame often determines the correct reading frame for eukaryotes, where translation usually takes place in one reading frame only.

Again, the problem has to be solved on the reverse complement of the sequence as well if the DNA is double stranded.

Algorithm

The previous algorithm for extracting all open reading frames can be extended to find the longest open reading frame, by keeping the position of the start and stop codon of the longest open reading frame found so far.

The start codon of the longest open reading frame found so far is $S[i', \ldots, i' + 2]$, and the corresponding stop codon is $S[j', \ldots, j' + 2]$.

Algorithm

function longest_open_reading_frame(S)
    $i' \leftarrow j' \leftarrow 0$
    $n \leftarrow \text{length}(S)$
    for $i \leftarrow 1, 2, 3$ **do**
        while $i + 2 \leqslant n$ **do**
            if $S[i, \ldots, i+2] = \text{AGT}$ **then**
                $j \leftarrow i + 3$
                while $j + 2 \leqslant n$ **and** $S[j, \ldots, j+2] \notin \{\text{TAA}, \text{TAG}, \text{TGA}\}$ **do**
                    $j \leftarrow j + 3$
                if $j + 2 \leqslant n$ **then**
                    if $j + 2 - i + 1 > j' + 2 - i' + 1$ **then**
                        $i' \leftarrow i$
                        $j' \leftarrow j$
            $i \leftarrow i + 3$
    return $(i', j' + 2)$

R is an interpreted scripting language.

An R program is a script containing a series of instructions, which are interpreted when the program is run instead of being compiled first into machine instructions and then assembled or linked into an executable program, thus avoiding the need for separate compilation and linking.

The R language is actually integrated in a software environment for statistical computing and graphics.

There are R distributions available for almost every computing platform, and free distributions can be downloaded from `http://www.r-project.org/`.

The actual mechanism of running a script using an R interpreter depends on the particular operating system, the common denominator being the Unix or Linux command line.

Assuming one of the R scripts shown further below was already written using a text editor and stored in a file named `sample.R` (where `R` is the standard file extension for R scripts), the following command invokes the R interpreter on the sample script:

```
$ R --vanilla < sample.R
```

The following command, on the other hand, invokes the R interpreter on the sample script from within the R software environment:

```
> source("sample.R")
```

When launching the R software environment, a new window with the name “R Console” will open and a welcome message like the following one will show in the console window.

When invoking the R interpreter from a console window, the same welcome message will show:

```
R version 2.7.1 (2008-06-23)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

>
```

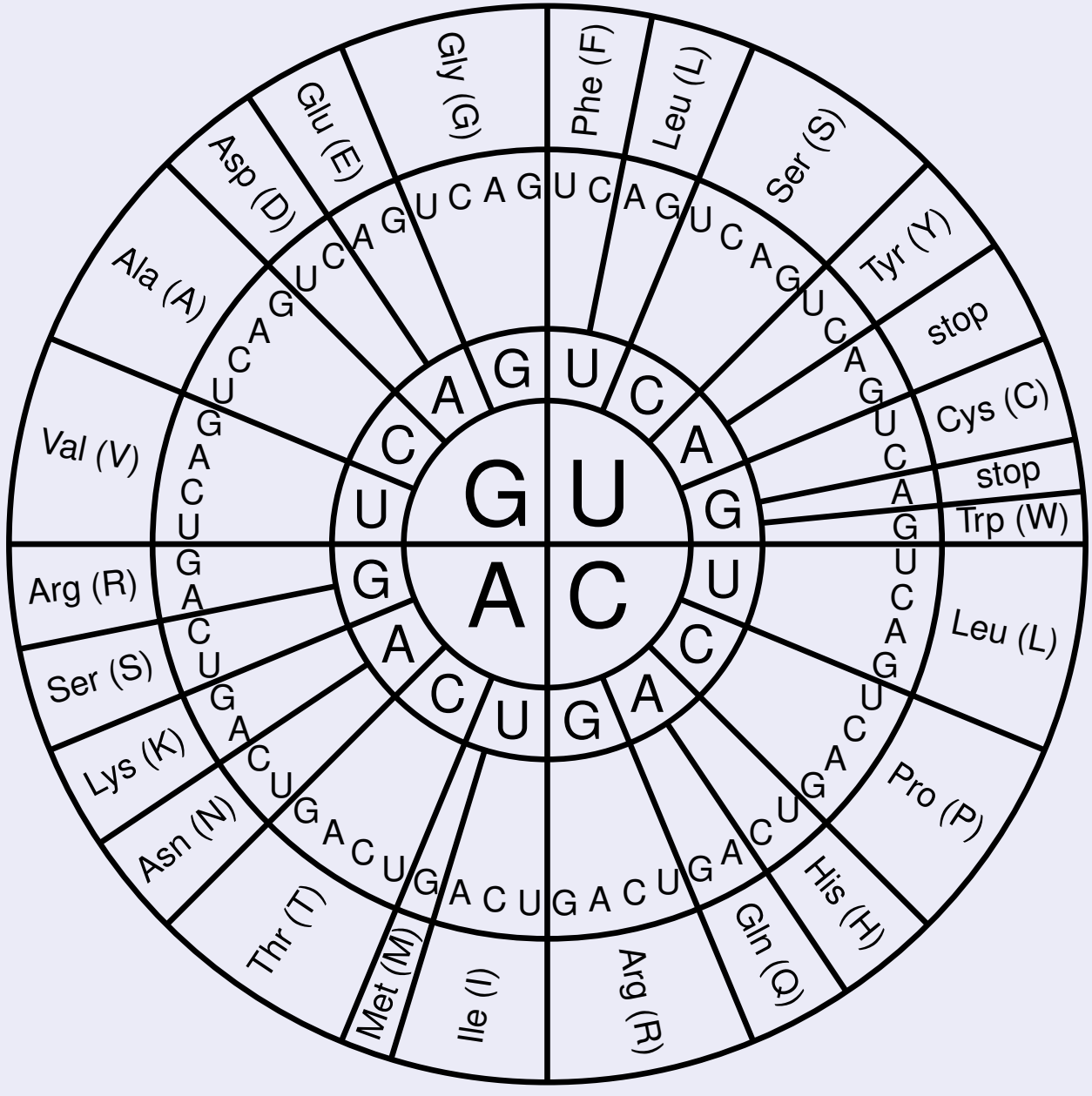
The simple computational biology problem at hand consists of translating to protein a messenger RNA sequence stored (in $5'$ to $3'$ direction) in a text file.

Recall that the primary structure of a protein can be represented as a sequence over the alphabet of amino acids A (alanine, Ala), R (arginine, Arg), N (asparagine, Asn), D (aspartate, Asp), C (cysteine, Cys), E (glutamate, Glu), Q (glutamine, Gln), G (glycine, Gly), H (histidine, His), I (isoleucine, Ile), L (leucine, Leu), K (lysine, Lys), M (methionine, Met), F (phenylalanine, Phe), P (proline, Pro), S (serine, Ser), T (threonine, Thr), W (tryptophan, Trp), Y (tyrosine, Tyr), and V (valine, Val).

A codon of three nucleotides is translated into a single amino acid within a protein, with translation beginning with a start codon (AUG) and ending with a stop codon (UAA, UAG, or UGA).

The $4^3 = 64$ different nucleotide triplets code for 20 amino acids, one translation start signal (methionine, one of these amino acids) and three translation stop signals, with some redundancies.

The genetic code defines a mapping between codons and amino acids, and despite variations in the genetic code across species, there is a standard genetic code common to most species, shown in the following circular table.


G U A C
U C A G
Phe (F)
Leu (L)
Ser (S)
Tyr (Y)
stop
Cys (C)
stop
Trp (W)
Leu (L)
Pro (P)
His (H)
Gln (Q)
Arg (R)
Ile (I)
Met (M)
Thr (T)
Asn (N)
Lys (K)
Ser (S)
Arg (R)
Val (V)
Ala (A)
Asp (D)
Glu (E)
Gly (G)

A codon is looked up in the circular genetic code table by starting with the first nucleotide at the center of the table, following with the second nucleotide at the inner circle, and finishing with the third nucleotide at the outer circle.

For instance, UAC and UAU both code for Tyr (Y).

Let us assume first the input messenger RNA sequence is already stored in a variable `$rna` and, thus, readily available for translation to protein.

The translation method will consist of skipping any nucleotides before the first start codon and then translating the rest of the sequence to protein using the standard genetic code, until the first stop codon.

As an example, translating sequence `GUCGCCAUGAUGGUGGUUAUUAUACCGUCAAGGACUGUGUGACUA` to protein involves skipping `GUCGCC` and then translating `AUG`AUGGUGGUUAUUAUACCG UCAAGGACUGUG`UGA` to `MVVIIPSRTV`, because AUG codes for Met (M), GUG and GUU code for Val (V), AUU and AUA code for Ile (I), CCG for Pro (P), UCA for Ser (S), AGG for Arg (R), and ACU codes for Thr (T).

```
rna <- "GCCAAUGACUAAGGCCUAAAGA"
protein <- rna.to.protein(rna)
cat(rna,"translates␣to",protein)
```

```
rna.to.protein <- function (rna) {
  ...
}
```

```
codon.to.amino.acid <- function (codon) {
  ...
}
```

```
rna.to.protein <- function (rna) {
  protein <- ""
  i <- 1
  while (i < nchar(rna)-1 &&
      substr(rna,i,i+2) != "AUG") { # start codon
    i <- i+1
  }
  i <- i+3 # skip the start codon
  while (i < nchar(rna)-1 &&
      substr(rna,i,i+2) != "UAA" &&
      substr(rna,i,i+2) != "UAG" &&
      substr(rna,i,i+2) != "UGA") {
    codon <- substr(rna,i,i+2)
    protein <- paste(protein,
      codon.to.amino.acid(codon),sep="")
    i <- i+3
  }
  return(protein)
}
```

```
codon.to.amino.acid <- function (codon) {
  if      (codon == "UUU") return("F") # Phe
  else if (codon == "UUC") return("F") # Phe
  else if (codon == "UUA") return("L") # Leu
  else if (codon == "UUG") return("L") # Leu
  else if (codon == "UCU") return("S") # Ser
  else if (codon == "UCC") return("S") # Ser
  ...
  else if (codon == "GAA") return("E") # Glu
  else if (codon == "GAG") return("E") # Glu
  else if (codon == "GGU") return("G") # Gly
  else if (codon == "GGC") return("G") # Gly
  else if (codon == "GGA") return("G") # Gly
  else if (codon == "GGG") return("G") # Gly
  else return("*")
}
```

```
repeat {
  rna <- scan(file="",what="character",n=1)
  if (length(rna) == 0) break
  protein <- rna.to.protein(rna)
  cat(rna,"translates to",protein)
}
```

```
codon2aa <- c(
  ...
)
```

```
rna.to.protein <- function (rna) {
  ...
}
```

```
codon2aa <- c(
  "UUU" = "F", # Phe
  "UUC" = "F", # Phe
  "UUA" = "L", # Leu
  "UUG" = "L", # Leu
  "UCU" = "S", # Ser
  "UCC" = "S", # Ser
  ...
  "GAA" = "E", # Glu
  "GAG" = "E", # Glu
  "GGU" = "G", # Gly
  "GGC" = "G", # Gly
  "GGA" = "G", # Gly
  "GGG" = "G"  # Gly
)
```

```
rna.to.protein <- function (rna) {
  protein <- ""
  i <- 1
  while (i < nchar(rna)-1 &&
      substr(rna,i,i+2) != "AUG") # start codon
  {  i <- i+1 }
  i <- i+3 # skip the start codon
  while (i < nchar(rna)-1 &&
      substr(rna,i,i+2) != "UAA" &&
      substr(rna,i,i+2) != "UAG" &&
      substr(rna,i,i+2) != "UGA") {
    codon <- substr(rna,i,i+2)
    if (is.na(codon2aa[codon])) { aa <- "*" }
    else { aa <- codon2aa[codon] }
    protein <- paste(protein,aa,sep="")
    i <- i+3
  }
  return(protein)
}
```

```
$ R --slave --vanilla --args "sample.rna" < sample.R
```

```
rna.to.protein <- function (rna) {
  ...
}
```

```
codon2aa <- c(
  ...
)
```

```
rna.file <- commandArgs(trailingOnly=TRUE)[1]
RNA <- scan(file=rna.file,what="character")
for (rna in RNA) {
  protein <- rna.to.protein(rna)
  cat(rna,"translates␣to",protein,"\n")
}
```

Vector variables hold lists of values, and these values can be accessed by position.

For instance, codon AGC is stored in position 46 of the `codon2aa` vector, and it can be extracted using the square bracket operator,

```
> codon2aa[46]
AGC
"S"
```

Vector values can also be accessed by a scalar key, using again the square bracket operator.

A list of one or more keys and values can be extracted using single square brackets,

```
> codon2aa["AGC"]
AGC
"S"
```

and a single value can be extracted using double square brackets,

```
> codon2aa[["AGC"]]
[1] "S"
```

A list of keys and values can also be extracted by position, enclosing a list of positions inside square brackets.

For instance, UCU, UCC, UCA, UCG, AGU, AGC all code for Ser (S), and these codons are stored in positions 5, 6, 7, 8, 45, 46 of the `codon2aa` vector,

```
> codon2aa[c(5:8,45,46)]
UCU UCC UCA UCG AGU AGC
"S" "S" "S" "S" "S" "S"
```

The square bracket operator also allows for the conditional selection of values in a vector, matrix, array, or list of values, by enclosing a Boolean expression inside single square brackets; for instance,

```
> codon2aa[codon2aa == "S"]
UCU UCC UCA UCG AGU AGC
"S" "S" "S" "S" "S" "S"
```

These six values are the only ones throughout the `codon2aa` vector for which the `codon2aa == "S"` expression evaluates to true:

```
> codon2aa == "S"
  UUU   UUC   UUA   UUG   UCU   UCC   UCA   UCG
FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
  UAU   UAC   UAA   UAG   UGU   UGC   UGA   UGG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  CUU   CUC   CUA   CUG   CCU   CCC   CCA   CCG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  CAU   CAC   CAA   CAG   CGU   CGC   CGA   CGG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  AUU   AUC   AUA   AUG   ACU   ACC   ACA   ACG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  AAU   AAC   AAA   AAG   AGU   AGC   AGA   AGG
FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
  GUU   GUC   GUA   GUG   GCU   GCC   GCA   GCG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  GAU   GAC   GAA   GAG   GGU   GGC   GGA   GGG
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Most of the operations on scalar values can also be applied to vectors, matrices, and arrays, as long as they are of the same length.

This is a powerful feature of R indeed, which may avoid the need for looping through a list of values.

For instance, the mean of a list of values stored in a vector, $\bar{x} = \sum x_i / n$, can be computed as their sum divided by the number of values, using basic arithmetical operations on the whole vector:

```
> x <- c(0:100)
> sum(x)
[1] 5050
> length(x)
[1] 101
> sum(x)/length(x)
[1] 50
```

The standard deviation of a list of values, $\sigma = \sqrt{\sum(x_i - \bar{x})^2/(n-1)}$, can also be computed using basic arithmetical operations on the whole vector:

```
> x.mean <- sum(x)/length(x)
> sqrt(sum((x-x.mean)^2)/(length(x)-1))
[1] 29.30017
```

Being that R is a software environment for statistical computing, these basic statistical functions are already available as `mean` and `sd` (standard deviation):

```
> mean(x)
[1] 50
> sd(x)
[1] 29.30017
```

The genetic code table can also be represented as a three-dimensional array, with one dimension for each of the three nucleotides in a codon.

The amino acids are listed with the 64 codons sorted in reverse order, from the third nucleotide back to the first nucleotide: AAA, CAA, GAA, UAA, ACA, CCA, GCA, UCA, . . . , AUU, CUU, GUU, UUU.

```
> t <- c("K","Q","E","-","T","P","A","S","R","R","G",
  "-","I","L","V","L","N","H","D","Y","T","P","A","S",
  "S","R","G","C","I","L","V","F","K","Q","E","-","T",
  "P","A","S","R","R","G","W","M","L","V","L","N","H",
  "D","Y","T","P","A","S","S","R","G","C","I","L","V",
  "F")
> acgu <- c("A","C","G","U")
> acgu <- list(acgu,acgu,acgu)
> c2aa <- array(t,dim=c(4,4,4),dimnames=acgu)
```

Array values, like vector and matrix values, can be accessed by position and also by a scalar key, using the square bracket notation:

```
> c2aa[1,3,2]
[1] "S"
> c2aa["A","G","C"]
[1] "S"
```

An empty position indicates the selection of all entries in the corresponding dimension; for instance, the amino acids coded by the four codons that start with the two nucleotides AG,

```
> c2aa["A","G",]
  A   C   G   U
"R" "S" "R" "S"
```

or the 16 codons that start with the nucleotide A,

```
> c2aa["A",,]
  A   C   G   U
A "K" "N" "K" "N"
C "T" "T" "T" "T"
G "R" "S" "R" "S"
U "I" "I" "M" "I"
```

A matrix is just a two-dimensional array, and the bracket notation can still be used for selecting individual values or whole rows or columns from a matrix.

For instance, the 16 codons that start with the nucleotide A in the `c2aa` array can also be seen as a matrix with four rows and four columns, one for each nucleotide.

```
> mat <- c2aa["A",,]
> mat["G","C"]
[1] "S"
> mat["G",]
  A   C   G   U
"R" "S" "R" "S"
```

In the previous matrix, AGU and AGC both code for Ser (S).

Their positions in the matrix can be revealed by means of the `which` function, which returns the array indices for which a given expression evaluates to true:

```
> which(mat == "S",arr.ind=TRUE)
  row col
G   3   2
G   3   4
> mat[3,2]
[1] "S"
> mat[3,4]
[1] "S"
```

Most of the operations on scalar values can also be applied to matrices, as long as they are of the same length, with one important exception to the latter rule.

Comparison operators can be applied to two matrices, but also to a scalar value and a matrix, and the result is, in both cases, a matrix.

```
> mat == "S"
      A     C     G     U
A FALSE FALSE FALSE FALSE
C FALSE FALSE FALSE FALSE
G FALSE  TRUE FALSE  TRUE
U FALSE FALSE FALSE FALSE
```

Vector, matrix, array, data frame, and list variables hold lists of values, including the special scalar constant `NA` (not available) that denotes missing values in R.

This special value is often carried through, because most operations on `NA` yield also `NA` as a result.

```
> m <- matrix(c(1,2,NA),nrow=3,ncol=4,byrow=TRUE)
> 3*m
     [,1] [,2] [,3] [,4]
[1,]    3    6   NA    3
[2,]    6   NA    3    6
[3,]   NA    3    6   NA
```

Notice that the three values 1, 2, NA are recycled by the `matrix` function in this example in order to fill in the 12 entries of the matrix.

Default values for arguments to a function can be included in the function definition by just giving the default value after each argument, separated by an equal sign; for instance,

```
sort <- function (x, decreasing = FALSE) { ... }
```

Functions written for one script can be made available to other scripts by placing them in a package, a text file with the usual extension containing R code.

For instance, any functions written in a text file named `sample.R` are readily available in any script using this package, that is, in any script containing the following header line:

```
library(sample)
```

A package in R is not only a text file containing R code, but a whole directory containing code and documentation files, including at least a `DESCRIPTION` file and a directory with at least one R code file; for instance,

```
sample/DESCRIPTION
sample/R/sample.R
```

The `DESCRIPTION` file contains basic information about the package, including at least the `Package`, `Version`, `Title`, `Author`, `Maintainer`, `Description`, and `License` information; for instance,

```
Package: sample
Version: 1.0
Title: Sample package
Author: Wilma Flintstone <wilma@hanna-barbera.com>
Maintainer: Betty Rubble <betty@hanna-barbera.com>
Description: Sample R package.
License: GPL version 3 or newer
```

The package can then be installed by the `R CMD INSTALL` command:

```
R CMD INSTALL sample
```

Now, when using several modules in an R script, the variable and function names declared in one module might clash with the names declared in another module.

Even in the same module, there might be two different `reverse_complement` functions, one for DNA and the other one for RNA nucleotides.

Such a name clash can be avoided by defining a generic function for reverse complementing a sequence, together with a `reverse.complement.dna` function for DNA and a `reverse.complement.rna` function for RNA.

The `sample.R` file in the `sample` directory would then look as follows.

```
reverse.complement <- function (seq)
  UseMethod("reverse.complement")

reverse.complement.dna <- function (seq) {
  rev <- paste(rev(unlist(strsplit(seq,split=""))),
  sep="",collapse="")
  chartr("ACGT","TGCA",rev)
}

reverse.complement.rna <- function (seq) {
  rev <- paste(rev(unlist(strsplit(seq,split=""))),
  sep="",collapse="")
  chartr("ACGU","UGCA",rev)
}
```

The two versions of the `reverse.complement` function are readily available in any script using this module, and it suffices to make the sequence of class DNA or RNA,

```
> dna <- "TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG"
> class(dna) <- "dna"
> rna <- "GGGUGCUCAGUACGAGAGGAACCGCACCC"
> class(rna) <- "rna"
```

before calling the general reverse complement function,

```
> reverse.complement(dna)
[1] "CACAAGCGTACAATGTGTAATGATCAAATAAGGTAATCAA"
> reverse.complement(rna)
[1] "GGGUGCGGUUCCUCUCGUACUGAGCACCC"
```

There are many ways in which sequences can be represented in R, let alone the simple representation of a sequence as a vector of characters and, as a matter of fact, many different R contributed packages implementing various types of sequences are available for download from CRAN, the Comprehensive R Archive Network, at `http://cran.r-project.org/`.

Among them, let us focus on the `seqinr` (Sequences in R) sequence representation, which is essentially a vector-based representation of sequences together with a collection of retrieval and analysis functions.

A sequence is represented in the R package `seqinr` as a vector of characters, and there is a function `c2s` (character vector to string) to convert a sequence to a character string, as well as a function `s2c` (string to character vector) to convert a character string to a sequence.

```
> library(seqinr)
> s <- "TGCTTCTGACTATAATAG"
> options(width=54)
> s2c(s)
 [1] "T" "G" "C" "T" "T" "C" "T" "G" "A" "C" "T" "A"
[13] "T" "A" "A" "T" "A" "G"
> c2s(s2c(s))
[1] "TGCTTCTGACTATAATAG"
```

Further, a sequence can also be obtained from the character string representing the sequence with the help of the `read.fasta` function of the R package `seqinr`, as shown in the following R script, where the sequence is stored in a file in the popular FASTA format.

```
> fas <- read.fasta(file="seq.fas",forceDNAtolower=FALSE)
> getSequence(fas[[1]])
 [1] "T" "G" "C" "T" "T" "C" "T" "G" "A" "C" "T" "A"
[13] "T" "A" "A" "T" "A" "G"
```

The R package `seqinr` can also be used to retrieve sequences from genomic databases, as shown in the following R script, where the complete genome sequence (4,639,675 nucleotides) of the bacterium Escherichia coli K-12, strain MG1655, is retrieved from the GenBank database.

```
> library(seqinr)
> choosebank("genbank")
> query("eco","AC=U00096")
> seq <- getSequence(eco$req[[1]])
> closebank()
> length(seq)
[1] 4639675
```

The representation of sequences in R package `seqinr` includes additional functions for performing various operations on sequences; for instance, to access the accession number or unique biological identifier for a sequence,

```
> getName(eco$req[[1]])
[1] "U00096"
```

to obtain the length of a sequence,

```
> getLength(eco$req[[1]])
[1] 4639675
```

to obtain the subsequence of a DNA, RNA, or protein sequence contained between an initial and a final position,

```
> getSequence(getFrag(fas[[1]],1,12))
 [1] "T" "G" "C" "T" "T" "C" "T" "G" "A" "C" "T" "A"
> getSequence(getFrag(fas[[1]],9,length(fas[[1]])))
 [1] "A" "C" "T" "A" "T" "A" "A" "T" "A" "G"
```

and to translate a fragment of DNA sequence into the corresponding protein coding sequence, according to the mapping of triplets of nucleotides (codons) to amino acids that underlies the genetic code.

```
> translate(s2c("AAAGGAGGTGGTCCA"))
[1] "K" "G" "G" "G" "P"
```