Announcements

- Assignment 3 posted. Deadline: March 1st at 23:55
- Midterm: March 3rd, at 6:00pm.
- Students taking BIOL 303 or BIOC 458 send me an e-mail
- WebCT disscusion board

Prime numbers

- Problem: determine whether a given positive integer is prime or not

Prime numbers

- Analysis:
  - Input: an integer n
  - Output: a boolean: true if n is prime, false otherwise
  - Definitions:
    - A *prime* number is a number which is divisible only by 1 and itself
    - An integer *a* is *divisible* by *b* if there is an integer k such that $a = kb$
  - Assumptions: *n* is positive

Prime numbers

- Basic idea: try to find a factor of n (i.e. a number that divides n), between 1 and n. If such number exists. then n is not prime, otherwise it is prime.

1. Set *is_prime* to true

2. Set i to be 2

3. While $i < n$, repeat:

   (a) if i divides n, then set *is_prime* to false
   (b) increment i by 1

4. Return the value of *is_prime*

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n)
{
  if (n % i == 0)
  {
    is_prime = false;
  }
  i++;
}
```

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n)
{
  if (n % i == 0)
  {
    is_prime = false;
    i = n;
  }
  i++;
}
```

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n)
{
  if (n % i == 0)
  {
    is_prime = false;
    break;
  }
  i++;
}
```

Nested Loops

```
while (condition)
{
  statements;
}
```

But loops are statments, so it is possible to “nest” them:

```
while (condition)
{
  statements1;
  while (condition2)
  {
    statements2;
  }
  statements3;
}
```

Nested Loops: example

```
int row, column;
final int MAX_ROW = 6;
final int MAX_COLUMN = 19;

row = 1;
while (row <= MAX_ROW)
{
    column = 1;
    while (column <= MAX_COLUMN)
    {
        System.out.print(‘‘*’’);
        column++;
    }
    System.out.println();
    row++;
}
```

Nested Loops: example

```
*******************
*******************
*******************
*******************
*******************
*******************
```

Nested Loops: example

```
int row, column;
final int MAX_ROW = 6;
final int MAX_COLUMN = 19;

row = 1;
while (row <= MAX_ROW)
{
    column = 1;
    while (column <= MAX_COLUMN)
    {
        System.out.print(“*”);
        column++;
    }
    System.out.println();
    row++;
}
```

Nested Loops: example

```
int row, column;
final int MAX_ROW = 6;
final int MAX_COLUMN = 19;

row = 1;
while (row <= MAX_ROW)
{
    column = 1;
    while (column <= row)
    {
        System.out.print(‘‘*’’);
        column++;
    }
    System.out.println();
    row++;
}
```

McGill

Nested Loops: example

```
*
**
***
****
*****
******
```

Nested Loops: example

```
******
*****
****
***
**
*
```

Nested Loops: example

```
int row, column;
final int MAX_ROW = 6;
final int MAX_COLUMN = 19;

row = MAX_ROW;
while (row >= 1)
{
    column = 1;
    while (column <= row)
    {
        System.out.print(‘‘*’’);
        column++;
    }
    System.out.println();
    row--;
}
```

Printing primes

- Problem: print the first *n* prime numbers
- Analysis:
  - Input: *n*, a positive integer
  - Output: a list of the first *n* prime numbers

Printing primes

- Design:
  - General idea: check each natural number in order, starting with 2, to see if it is prime or not. If it is prime, print it, and increment a counter by 1. Once the counter is over n, stop.
  - Algorithm:

```
1. Set counter to 0
2. Set number to 2
3. While counter < n, repeat:
   (a) If number is prime,
        i. Print number
       ii. increment counter by 1
   (b) Increment number by 1
```

Printing primes

- Algorithm refinement (make an algorithm more precise):

```
1. Set $counter$ to 0
2. Set $number$ to 2
3. While $counter < n$, repeat:
   (a) Set $is\_prime$ to true
   (b) Set $i$ to be 2
   (c) While $i < number$, repeat:
       i. if $i$ divides $number$, then set $is\_prime$ to false
       ii. increment i by 1
   (d) If $is\_prime$ is true,
       i. Print $number$
       ii. increment $counter$ by 1
   (e) Increment $number$ by 1
```

Printing primes

```
int n, counter, number, i;
boolean is_prime;
n = scanner.nextInt();

counter = 0;
number = 2;
while (counter < n) {
  is_prime = true;
  i = 2;
  while (i < number) {
    if (n % i == 0) { is_prime = false; break; }
    i = i + 1;
  }
  if (is_prime) {
    System.out.print(“ ” + number);
    counter = counter + 1;
  }
  number = number + 1;
}
```

Bioinformatics and Computational Biology

- The use of computational techniques to solve problems in Biology such as
  - Small scale Biology:
    - Analyzing DNA,
    - Analyzing the structure of proteins,
  - Large scale Biology:
    - Simulating eco-systems,
  - ...etc

DNA

- DNA is a large molecule encoding information about the structure and functions of organisms.
- DNA is made of two long chains or strings of molecules called *nucleotides*, which are twisted so it has an helix shape.
- There are four types of nucleotides, called Adenine, Cytosine, Guanine and Thymine.
- The two chains are complementary in the sense that
  - if there is Adenine in one chain, in the opposite chain there is Thymine in the same position, and viceversa, and
  - if there is Guanine in one chain, there is Cytosine in the opposite, and viceversa
  - For example

    ```
    AGGTAC
    TCCATG
    ```

Problem

Given a host DNA sequence and a gene sequence, find out if the gene occurs in the host or not, and if so, then say in which position.

For example, given a the host

```
AGGTACGCC
```

and the gene

```
ACG
```

we say that the gene does occur in the host at position 4 (counting from 0.)

But the gene

```
ATCA
```

does not occur in the host.

Analysis

- Input: two strings: the host, and the gene
- Output: a boolean which is true if the gene occurs in the host. If so, then we also produce a natural number which is the starting position of the gene in the host.
- Data representation: Abstraction
  - The internal chemical structure of A, T, G and C is irrelevant, so we *abstract* it.
  - We do not need to represent both chains, because they are redundant.
  - The shape of the molecules is also irrelevant for this problem, so we abstract it too.
  - The gene and the host are represented as strings,
  - The gene and the host are made up of only four characters: A, T, G, and C.
- If the gene is larger than the host, then obviously it cannot occur in it.

Design: General algorithm

Compare the gene with the host from left to right:

1. Compare the gene with the host starting at the first position of the host
2. If it matches then stop,
3. Otherwise, compare it, starting at the second position of the host
4. If it matches then stop,
5. Otherwise, compare it, starting at the third position of the host
6. ... etc.

Design: General algorithm (contd.)

```
0
AGGTACGCTAGGCA
TAGG
```

No match, so we move on...

```
01
AGGTACGCTAGGCA
 TAGG
```

No match, so we move on...

```
012
AGGTACGCTAGGCA
  TAGG
...
012345678
AGGTACGCTAGGCA
        TAGG
```

Match!

Design: General algorithm (contd.)

```
          1111
01234567890123
AGGTACGCTATGCA
          TAGG
```

No match, so the gene doesn't occur in the host.

Design: A bit more precise...

1. Set *position* to 0
2. While *position* <= length of the *host* - length of the *gene*, repeat:
   - (a) Compare the gene with the host starting from *position* in the host
   - (b) If the gene matches, then we found it, so stop looking
   - (c) Otherwise, increment *position* by 1 and continue

Design: Comparing the gene with the host

1. Compare the character *position* of the *host* with the first of the *gene*
2. If they are different then the gene doesn't match, so stop
3. Otherwise, compare the character *position+1* of the *host* with the second of the *gene*
4. If they are different then the gene doesn't match, so stop
5. Otherwise, compare the character *position+2* of the *host* with the third of the *gene*
6. ... etc
7. If we reach the end of the gene, then it matches

Design: Comparing the gene and the host (cont.)

```
0123
AGGTACGCTAGGCA
   TAGG
```

We compare the first character of the gene...

```
   3
AGGTACGCTAGGCA
   TAGG
   0
```

They match so we continue...

```
   34
AGGTACGCTAGGCA
   TAGG
   01
```

They match so we continue...

```
   345
AGGTACGCTAGGCA
   TAGG
   012
```

They don't match so we stop the comparison and continue were we left...

```
01234
AGGTACGCTAGGCA
    TAGG
```

Design: A bit more precise...

```
1. Set host_index to position

2. Set gene_index to 0

3. Set occurs to true

4. While host_index < the length of the host and
   gene_index < the length of the gene, repeat:

   (a) If the host nucleotide at host_index is different than
       the gene nucleotide at gene_index, then:
        i. Set occurs to false, and
       ii. stop testing this position
   (b) Increment the host_index by 1 and the gene_index
       by 1
```

Design: Putting it all toghether

```
1. Set occurs to false
2. Set position to 0
3. While position < the length of the host - length of the gene, repeat:
   (a) Set host_index to position
   (b) Set gene_index to 0
   (c) Set occurs to true
   (d) While host_index < the length of the host and gene_index < the length of the gene, repeat:
       i. If the host nucleotide at host_index is different than the gene nucleotide at gene_index, then:
          A. Set occurs to false, and
          B. stop testing this position
       ii. Increment the host_index by 1 and the gene_index by 1
   (e) If occurs is true then stop the main loop
   (f) otherwise, increment the position by 1 and continue
```

Implementation

```
public class GeneFinder
{
  public static void main(String[] args)
  {
    String host, gene;
    boolean occurs;
    int position, host_index, gene_index;
    char host_nucleotide, gene_nucleotide;

    System.out.print("Enter a host DNA seq: ");
    host = scanner.nextLine();

    System.out.print("Enter a gene DNA seq: ");
    gene = scanner.nextLine();

    // Continues below...
```

Implementation (cont.)

```
occurs = false;
position = 0;
while (position <= host.length()
                  - gene.length()) {
  host_index = position;
  gene_index = 0;
  occurs = true;
  while (host_index < host.length()
           && gene_index < gene.length()) {
    host_nucleotide = host.charAt(host_index);
    gene_nucleotide = gene.charAt(gene_index);
    if (gene_nucleotide != host_nucleotide) {
       occurs = false;
       break;
    }
    host_index++;
    gene_index++;
  } // End of inner while
  if (occurs) { break; }
  position++;
} // End of outer while
```

```
      if (occurs) {
        System.out.println(“The gene ” + gene
          + “ occurs at ” + position
          + “ in the host ” + host);
      }
      else {
        System.out.println(“The gene ” + gene
          + “ does not occur in the host ” +host);
      }
    } // End of main
  } // End of class GeneFinder
```

Alternative forms

- There are two alternative syntactic forms for loops:
  - The "do-while" loop:

```
do {
  list_of_statements;
} while (boolean_expression);
```

  - The "for" loop:

```
for (stmt1; boolean_expression; stmt2) {
  list_of_statements;
}
```

Alternative forms

- A loop of the form

```
do {
  S;
} while (C);
```

- is equivalent to

```
S;
while (C) {
  S;
}
```

where `S` is any list of statements and `C` is any boolean expression

Alternative forms

```
int n, i;
i = 1;
n = 1;
while (i < n)
{
  System.out.println(i);
  i++;
}
```

Alternative forms

```
int n, i;
i = 1;
n = 1;
do
{
  System.out.println(i);
  i++;
} while (i < n);
```

Alternative forms

- A loop of the form

```
for (I; C; A) {
  S;
}
```

- is equivalent to

```
I;
while (C) {
  S;
  A;
}
```

where `I` is a statement, called the “initializer”, `C` is a boolean expression, `A` is a statement, called the “advance”, and `S` is any list of statements.

Alternative forms

```
boolean x = true;
int i, m;
m = 101;
for (i = m - 1; i > 1 && x; i--) {
  if (m % i == 0) x = !x;
}
```

Alternative forms

```
boolean x = true;
int i, m;
m = 101;
i = m - 1;
while (i > 1 && x) {
  if (m % i == 0) {
    x = !x;
  }
  i--;
}
```

Alternative forms

```
while(C)
{
  S;
}
```

is equivalent to

```
do {
  if (!C) { break; }
  S;
} while (true);
```

and also equivalent to

```
for ( ; C ; ) {
  S;
}
```

The end