CSI 2131 File Management — Winter 2006
Computer Science — University of Ottawa

Homework Assignment #1 (100 points, weight 10%)

Due: Friday, February 3th at 5:00 p.m. (via webCT)

1 Part 1: Handling Genome Files (90 marks)

DNA molecules can be seen as strings over a four letters alphabet, each letter correponding to a basis: A,C,G,T. The size of the strings varies from one organism to another. Typically, the genome of a bacteria consists of millions of bases (letters) while the human genome is 3-billion-letter long. DNA molecules consist of two complementary strands (strings) running in opposite directions. The rules for complementarity are that A and T are complementary, as well as C and G. Since the sequence of one strand can be deduced from the other strand, only one of the two strands needs to be physically stored.
In this assignment, you are going to write 3 programs to manipulate a file that stores a DNA strand. The file provided is simply a sequence of characters, each character being one of A,C,G,T.

1.1 Program 1: Removing low-complexity segments `project name: lowcomp` (30 marks)

Consider a string corresponding to a DNA strand. Low complexity segments of this string are mainly repetitive sequences. Certain algorithms that manipulate genome data (such as sequencing algorithms) get confused by low-complexity segments, so these segments are usually masked prior to running these algorithms. You will write a program that, given a sequence and the indices (beg, end) of low complexity segments, substitutes each characters in these segments by a wildcard character `'X'`.
Your program must prompt the user for entering the genome file name (file containing the DNA strand) and the low-complexity file name (file containing a sequence of pairs (beg, end)). Given the file names, say `genome1.txt` and `lowcomp1.txt`, your program must read the pairs (beg, end) from `lowcomp1.txt` and substitute the characters in `genome1.txt` that have byte offset *beg* up to byte offset *end* by `'X'`, for each pair (beg, end). The program must show on the screen the low-complexity segments that are substituted.

```
genome1.txt:
   ACATATATATATCTACGGTACGTGTCGTGT
lowcomp1.txt:
   3 11
   20 29
modified genome1.txt
   ACAXXXXXXXXXCTACGGTAXXXXXXXXXX
screen:
   TATATATAT
   CGTGTCGTGT
```

IMPORTANT: The genome files are typically very large, so you should assume the whole file cannot fit into an array in main memory. You must make use of the file positioning methods `seekp` and `seekg` in order to move to specific locations in the file.
When testing your program sucessively for the same input file, remeber to recopy the input file, since the program will change its contents.

1.2 Program 2: Compressing a genome file `project name: compress` (30 marks)

Consider the original file provided for program 1, containing a sequence of characters from the alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. This file format wastes a lot of space, since it uses one byte to store each of the letters in the alphabet. In this program, we will use compact notation in order to compress the file, reducing its size by a factor of 4.
Associate two bits to each character as follows: `A:00 C:01 G:10 T: 11`
Each character in the input file will be converted into two bits, and the bits corresponding to every 4 characters will be combined into a byte, that will be written to the output (compressed) file. Since the number of characters in the input file may not be a multiple of four, we need to know how many characters (1, 2, 3, or 4) are packed into the last byte converted from the original sequence; this number will be written as the last byte of the output file.
Your program must prompt the user for the name of a genome file and for the name of the compressed file to be created, and it must create the compressed file as described.

```
Example:
genome1.txt: ACTGAAAG
genome1.cmp: a binary file that contains the following 3 bytes:
             00011110 00000010 00000100
              (ACTG)   (AAAG)    (4)
genome2.txt: AATTCCGGAT
genome2.cmp: a binary file that contains the following 4 bytes:
             00001111 01011010 00110000 00000010
              (AATT)   (CCGG)   (AT--)    (2)
```

IMPORTANT: Assume your input file is too big to fit entirely into an array in main memory. Therefore, you should be reading character by character from the input file, and only need to have up to 4 characters at a time stored into main memory. Similarly, the compressed file should be assumed to be too big to fit into main memory. Therefore, your program should be writing one byte at a time into the output file. Note that file positioning methods `(seekp,seekg)` are NOT to be used in program 2 and 3, since you will process files sequentially.

1.3 Program 3: Decompressing a compressed genome file `project name: uncompress` (30 marks)

This program receives the name of the input (compressed) file, say genome2.cmp, and the name of the the output (uncompressed) file, say genome2.ucp; your program must convert the input (compressed) file into its original format, reversing the process done in program 2. You may test the correctness of the last two programs as follows:

```
run compress   with files genome1.txt genome1.cmp
run uncompress with files genome1.cmp genome1.ucp
compare the contents of genome1.txt and genome1.ucp (must be identical!)
```

1.4 Standards for your programs and submission

Testing and checking your output files:
We will provide input files for testing your program. We reserve the right of using additional input files when testing your program. When correct outputs are available for the given inputs, they will be provided to you at the webCT page.
Please, create the output files in the exact format that is specified in this handout, since it is likely that we will use some automatic way of comparing your outputs with correct ones.

Documentation and Style
Your program must be well-documented and written in good style. Part of the marks will be dedicated to documentation and style. To document your program, write comments explaining what is done in the following lines and add comments at the end of certain lines explaining what has been done on that line. For each function or method add a full explanation before it, explaining its purpose and what its parameters represent. For each class created, add an explanation before it. Good style includes good documentation, choice of clear names (for variables, classes, functions, etc) and good program organization and design (like creating methods that deal with each aspect/task within a class).

What and how to submit your assignment
Please, read the course's policy on plagiarism and collaboration. You must not include ideas or pieces of code from your colleagues or from other programs available in the web or elsewhere. By submitting this assignment you are automatically acknowledging understanding of the policy.

Create a directory/folder named `a1` containing only the following:

- folder `lowcomp` containing the program file(s) used to create the project `lowcomp`
- folder `compress` containing the program file(s) used to create the project `compress`
- folder `uncompress` containing the program file(s) used to create the project `uncompress`
- file `written.txt` containing your solution to the written questions in Section 2 (this file must be in plain text; you may use notepad, wordpad or similar editors).

EVERY FILE MUST CONTAIN A HEADER WITH Student Name, Student Number, Course and Section. **Zip the folder `a1` and submit the zipped file as your assignment#1 submission to webCT (only one file is submitted)**; this should be done in such a way that unzipping will produce a folder `a1` with the files inside.

2 Part 2: Written Assignment (10 marks)

2.1 Disks (5 marks)

Given a Western Digital Caviar AC22100 Disk System with the following characteristics:

Capacity: 2,100 MB

Average Seek Time: 12 msec

Average Rotational Delay: 6msec

Transfer rate: 12 msec/track

Bytes per Sector: 512

Sectors per track: 63

Tracks per cylinder: 16

Cylinders: 4092

Assume that you want to store a file with 350,000 80-byte records.

1. How many cylinders are necessary to hold the file given that a record can span several sectors, tracks or cylinders?
2. How many cylinders are necessary to hold the file if a record is not allowed to span more than one sector ?
3. Assume the situation in question 2, and assume that there are no interferences from other processes and that the file completely fills a cylinder before it continues into the next cylinder and that the required cylinders are dispersed randomly on the disk. How long would it take to access the entire file in sequence?
4. Under the same assumptions as question 3., how long would it take to access the entire file randomly (accessing every record, but in random order)?

2.2 Tapes (5 marks)

Assume that you want to store a file with 350,000 80-byte records on a 2400-foot reels of 1600-bpi tape with 0.5-inch interblock gaps.

1. What are the minimum and maximum blocking factors that make it necessary to use exactly three tapes?
2. What is the effective recording density when a blocking factor of 50 is used?
3. Given that the tape speed is 150 ips. What is the effective transmission rate of this configuration?