SM2-21st Computer Lab 6

Text File, Array, String, Algorithms

20 October 2017, Friday 6:45pm

Unzip lab6.zip sent to you by email attachment. Prepare the 3 programs in advance.

Question 1:

In the C Programming Language, the **fgets** function reads characters from the input pointed to by *stream*. The **fgets** function will stop reading when *n*-1 characters are read, the first new-line character is encountered in *s*, or at the end-of-file, whichever comes first. The **fgets** function will subsequently append a null character called string terminator to the string.

The syntax for the **fgets** function is:

char *fgets(char ****s*, int** ***n*****, FILE ****stream***);**

where

s is the array where the characters that are read will be stored.
n is the size of *s*.
stream is the stream to read.

The **fgets** function returns *s* - which is the starting address of the string. If an error occurs while trying to read the stream or the end of the stream is encountered before any characters are stored, the **fgets** function will return a null pointer.

The input file **books.inf** contains the book title of not more than 18 characters, book price and the quantity in the stock as follows:

```
Learning C         15.90  400
How to Score A+    26.70   30
C Programming Tips 17.50  250
```

As each string needs a string terminator, we need to have an array size of (18+1) = 19 characters to store the book title. The program named as **book.c** making use of **fgets** function is given as follows:

```
// book.c

# include <stdio.h>

main(void)
{
  char title [19];
  float price;
  int quantity;

  FILE *indata, *outdata;

  indata  = fopen ("f:\\lab5\\books.inf","r");
  if (indata==NULL)
  {
        printf ("books.inf does not exist.");
        exit (1);
  }

  outdata = fopen ("f:\\lab5\\books.ouf","w");
  if (outdata==NULL)
  {
        printf ("books.ouf cannot be created.");
        exit (1);
  }

  while ( (fgets (title,19,indata) !=NULL) &&
        (fscanf (indata,"%f%d%*c",&price,&quantity) ==2) )
  {
     fprintf (outdata, "Book Title : %s \n",title);
     fprintf (outdata, "Price : $ %.2f \n", price);
     fprintf (outdata, "Stock : %d\n\n",quantity);
  }

  fclose (indata);
  fclose (outdata);
  return 0;
}
```

The contents of output file (**books.ouf**) produced by the program is as follows:

```
Book Title : Learning C
Price : $ 15.90
Stock : 400

Book Title : How to Score A+
Price : $ 26.70
Stock : 30

Book Title : C Programming Tips
Price : $ 17.50
Stock : 250
```

A text file named as wage.inf (unzip from lab6.zip) contains the information of all employees in a small factory. The file has the following contents:

i) Employee name of not more than 13 characters (column 1 to 13)
ii) Employee number (column 15 to 18)
iii) Gender (column 20)
iv) Hourly rate (floating-point number)
v) Number of hours worked (floating-point number)

Input File: wage.inf

```
Tan Ai Lee     3123 F   7.50  40
Wong Alice     3164 F   8.75  35.5
Chan Meng      3185 M   9.35  43.5
Tan Cheng Lee  3202 M   10.50 45
Henry Loh      3559 M   6.35  29.5
Lim Gao        3660 M   10.85 39
```

You are required to write a C program named as **<u>wage.c</u>** to compute the wages of each employee and also to sum up all the wages. The regular pay is computed by <u>Hourly rate</u> * <u>Number of hours worked</u> for up to 40 hours. For those hours greater than 40, a multiplier of 1.5 is used to compute the overtime pay. The wage for each employee is the sum of regular pay and overtime pay. For example, the wage received by `Chan Meng` is computed as 9.35*40 + 9.35*(43.5-40)*1.5 = 423.09. Print your output to a text file named as **wage.ouf** using the following format:

```
 Employee No.    Hours Worked    Hourly Rate       Wage
                     (hr)          ($/hr)           ($)
==========================================================
     3123           40.00            7.50         300.00
     3164           35.50            8.75         310.62
     3185           43.50            9.35         423.09
     3202           45.00           10.50         498.75
     3559           29.50            6.35         187.32
     3660           39.00           10.85         423.15

                                   Total wages = $2142.94
```

Question 2:

An algorithm used to sort *n* data is illustrated as follows based on 6 integers. Let the initial sequence of the data be 84, 82, 81, 86, 83, 85, and let the array have the following contents: a[0]=84, a[1]=82, a[2]=81, a[3]=86, a[4]=83, and a[5]=85. We want to sort the array in non-decreasing order.

The first iteration is to place 84 (a[0]) at a[0], but there is no change from a[1] to a[5]. This does not need you to do anything.

The second iteration is to place 82 (a[1]) at the right place, but there is no change from a[2] to a[5]. To do so, 82 is compared with the first integer 84 (a[0]). Since 82 <84, 84 will have to be pushed to the right so that 82 can be placed at a[0]. The end results are <u>82, 84,</u> 81, 86, 83, 85.

The third iteration is to place 81 (a[2]) at the right place, but there is no change from a[3] to a[5]. To do so, 81 is compared with the first integer 82 (a[0]) and continue with the next comparison and so on. Since 81 is already smaller than 82 (a[0]), both 82 (a[0]) and 84 (a[1])) will have to be pushed to the right so that 81 can be placed at a[0]. The end results are <u>81, 82, 84</u>, 86, 83, 85.

The fourth iteration is to place 86 (a[3]) at the right place, but there is no change from a[4] to a[5]. To do so, 86 is compared with the first integer 81 (a[0]). Since 86 < 81, we continue to compare 86 with 82 (a[1]). Since 86 <82 and we will subsequently reach the original position of 86 at a[3], the final position of 86 remains unchanged in a[3]. The end results are <u>81, 82, 84, 86,</u> 83, 85.

The fifth iteration is to place 83 (a[4]) at the right place, but there is no change to a[5]. To do so, 83 is compared with the integers from a[0] onwards, and the suitable position to place 83 is at a[2]. Therefore, 84 (a[2]) and 86 (a[3]) will have to be pushed right so that 83 can be placed in a[2]. The end results are <u>81, 82, 83, 84, 86</u>, 85.

The last iteration is to place 85 (a[5]) at the right place. To do so, 85 is compared with the integers from a[0] onwards, and the suitable position to place 85 is at a[4]. Therefore, 86 will have to be pushed right so that 85 can be placed at a[4]. The end results are <u>81, 82, 83, 84, 85, 86</u>,

After the 6 iterations used to sort the 6 data, the algorithm is complete.

The program sort1.c (given in Lab6.zip) contains 65300 integers in random order, i.e., *n* = 65300. Use the algorithm described above to sort the integers.

What is the time complexity of the above algorithm in the worst case? What is its time complexity in the best case?

Question 3:

Another variation of the algorithm described in question **2** is that to find the right place for a[i], it compares the value of a[i] with the previous sorted data from a[i-1] to a[0] instead of from a[0] to a[i-1]. In this variation what is the time complexity in the worst case? What is its time complexity in the best case? Program this variation in sort2.c (given in Lab6.zip).

Use debugger whenever in doubt!!

(Prepare your own 3 programs in advance. You only have 2 hours in the computer lab.)