

1141 NTOUCSE Programming 1C Final Exam

Name: _____ Major: _____ Student Id: _____ 114/12/23 (Tue.)

Exam Time: **13:20 – 16:00**

- Exam rules :
1. **Close book, close** everything including quizzes, homeworks, assignments, reference materials, etc.
 2. You can answer the questions in **English** or in **Chinese**, in this **problem sheet**, the **answer sheet**, or **both**.
 3. You can use language features not taught in class if you feel necessary, but **strictly** limited in **C**.
 4. **No mobile phone, pad, computer** or **calculator** is allowed. (Electronic) English dictionary is OK
 4. No peeping around! No discussion! No exchange of any material; **raise your hand if you have any question about the exam problems**
 5. If you turn in the paper earlier than the specified time, **leave the classroom immediately and quietly**
 6. Against any of the above rules will be treated as cheating in the exam and handled by school regulations.
 7. **Turn in BOTH the problem sheet with your name and id and answer sheet with your name and id.**

1. In this course, we explained that the two primary repetitive structures in programming are loops and recursion. As you progress through the field of computer science, a recurring question will inevitably arise: which is better, recursion or loops? This is not an easy question to answer; it is deeply tied to your programming experience and involves other paradigms, such as Functional Programming. Through various exercises in this course, we aim to help you find the answer. Please use the examples provided in our curriculum to compare the applications of recursion versus loops:
 - (a) [3] Explain in terms of current mainstream processor architectures.
 - (b) [3] In terms of program complexity (e.g., number of variables and lines of code).
 - (c) [3] Consider the automatic utilization of multi-core, multi-threaded, and multi-machine network computing environments.
 - (d) [3] When the problem structure itself possesses a recursion formula, for example:
 - (e) [3] When the "solution for a problem of size n" can be directly expressed using the "solutions for problems of size 1 to n-1," for example:
 - (f) [3] When using a brute-force Depth First Search (DFS) method to traverse a decision tree, for example:
2. [12] When using nested loops to enumerate all possibilities from 0 to 9 as shown below, the program structure is straightforward, but what is the biggest problem with this approach? Please write a recursive function to achieve the same functionality to solve the problem.

```
1 int i1, i2, i3, i4;
2 for (i1=0; i1<=9; i1++)
3     for (i2=0; i2<=9; i2++)
4         for (i3=0; i3<=9; i3++)
5             for (i4=0; i4<=9; i4++)
6                 printf("%d %d %d %d\n", i1, i2, i3, i4);
```

3. (a) [8] How can you use pointer syntax in a C program to make a one-dimensional integer array `int array1D[12]` behave simultaneously as a two-dimensional integer array (identical in usage to `int array2D[3][4]`)? This allows the same segment of memory to be accessed via two different methods. The loop on line 07 will print: 0 1 2 3 10 11 12 13 20 21 22 23. A similar concept is applied when we want to examine data byte-by-byte; we can use pointer syntax to treat this `array1D` as an array of 48

bytes, such that the loop on line 09 will print 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 0a 00 00 00 0b 00 00 00 0c 00 00 00 0d 00 00 00 14 00 00 00 15 00 00 00 16 00 00 00 17 00 00 00

```

01 int i, array1D[12];
02 _____ array2D _____ = _____ array1D;
03 _____ cbuf = _____ array1D; // or array2D equivalently
04 for (i=0; i<3; i++)
05     for (j=0; j<4; j++)
06         array2D[i][j] = i*10+j;
07 for (i=0; i<12; i++)
08     printf("%d ", array1D[i]);
09 for (i=0; i<12*sizeof(int); i++)
10     printf("%02x ", cbuf[i]);

```

- (b) [6] Many people find the type definition for two-dimensional arrays, such as `int x[3][4]`, or the type definition for the pointer `x` to be counterintuitive because the syntax splits the type around the variable name. In C, the `typedef` keyword provides a way to create aliases for types. Please define an alias `I4ARY` for the type `int[4]`, then use `I4ARY` to define the aforementioned 2D array `x`. Furthermore, define a constant pointer `y` as an equivalent name for the array `x`, so that the following two loops can operate on the same array using `x` and `y` respectively.

```

04 for (i=0; i<3; i++)
05     for (j=0; j<4; j++)
06         x[i][j] = i*10+j;
07 for (i=0; i<3; i++) {
08     for (j=0; j<4; j++)
09         printf("%d ", y[i][j]);
10     printf("\n");

```

- (c) [6] Please provide the other three common equivalent parameter declaration methods for passing a two-dimensional array `int x[3][4]` into a function, in addition to `int func(int y[3][4])`.
- (d) [6] Regardless of which of these four methods is used, the way you access array elements inside the function remains exactly the same: `y[i][j]`. What will be printed when the following program executes? Please explain why (assuming a 64-bit memory address environment).

```

01 #include <stdio.h>
02 void func(int y[3][4])
03 {
04     printf("%d %d", sizeof(y), sizeof(*y));
05 }
06 int main()
07 {
08     int x[3][4];
09     printf("%d ", sizeof(x));
10     func(x);
11     return 0;
12 }

```

4. (a) [5] What is the prerequisite for using binary search to find a specific value within the array `int`

data[1000]?

- (b) [5] Please complete the following loop to search for the specified data in the array and return the index of that element. If the data is not found, return -1.

```
01 int binarySearch(int target, int data[], int left, int right)
02 {
03     int mid;
04     while (left <= right)
05     {
06         mid = (left+right)/2;
07         if (target == data[mid])
08             return _____;
09         else if (target > data[mid])
10             _____;
11         else
12             _____;
13     }
14     return -1;
15 }
```

- (c) [5] Please complete the following lower_bound() function to search for the specified data in the array and return the index of its first occurrence. If the data is not found, return the index of the last occurrence of the largest element that is smaller than the target value. Essentially, this function returns the 'index of the array element that is less than or equal to the specified element.'

For example, given the following data array:

0	1	2	3	4	5	6	7	8	9
1	2	2	3	3	3	5	6	6	8

lower_bound(3,data, 0, 9) returns 3 and lower_bound(4,data, 0, 9) returns 6

```
01 int lower_bound(int target, int data[], int left, int right)
02 {
03     int mid, result = _____;
04     while (left <= right)
05         if (data[mid=(left+right)/2] _____ target)
06             _____;
07         else
08             _____, result = _____;
09     return result;
10 }
```

- (d) [3] How can you use the lower_bound() function from question (c) to obtain the index of the first element greater than a specified target value? Using the data array mentioned above as an example, when the target is 3, the result should be 6; when the target is 4, the result should also be 6.
- (e) [3] How can you utilize the lower_bound() function from question (c) to obtain the index of the last occurrence of an element less than or equal to a specified value? Using the aforementioned data array as an example, when the target is 3, the result should be 5; when the target is 4, the result should also be 5.

- (f) [3] How can you utilize the `lower_bound()` function from question (c) to obtain the index of the last occurrence of an element strictly less than a specified value? Using the aforementioned data array as an example, when the target is 3, the result should be 2; when the target is 4, the result should be 5.
5. The following bubble sort algorithm can be viewed as consisting of $n-1$ passes. In each pass, the smallest element within the range $[i, n-1]$ is moved to its correct position through a series of swaps.

```

01 for (i=0; i<n-1; i++)
02     for (j=n-2; j>=i; j--)
03         if (x[j]>x[j+1])
04             swap(x, j, j+1);

```

During this process, larger elements simply drift slowly toward the back without being specifically addressed. Consequently, one might wonder if algorithm efficiency could be improved by moving the smallest element in the range forward in one pass and the largest element backward in the next. Please complete the following code:

- (a) [5] In the aforementioned loop, the data range for each pass covers the i -th to the $(n-1)$ -th elements of the array. However, if we alternate between forward and backward directions, the data range should be $[(i+1)/2, n-1-i/2]$. Additionally, the direction of the inner loop must change according to the pass number.

```

01 for (i=0; i<n-1; i++)
02     if (i%2==0)
03     {
04         for (j_____ ; j_____ ; j--)
05             if ( x[j]>x[j+1])
06                 swap(x, j, j+1);
07     }
08     else
09     {
10         for (j_____ ; j_____ ; j++)
11             if (x[j]>x[j+1])
12                 swap(x, j, j+1);
13     }

```

- (b) [5] In the implementation of question (a), we can see recurring blocks of code. Although the data-swapping logic has been encapsulated within a `swap()` function—making the duplication less obvious—redundancy still leads to difficulties in debugging and maintenance. We need to continuously practice abstracting common logic and using variables to represent the parts that change. Please complete the following consolidated code:

```

01 for (i=0; i<n-1; i++)
02 {
03     start = i%2 ? _____ : _____;
04     end = i%2 ? _____ : _____;
05     dir = i%2 ? _____ : _____;
06     for (j=_____ ; j_____ ; j_____ )
07         if (x[j]>x[j+1])
08             swap(x, j, j+1);

```

09 }

- (c) [5] In the implementation of question (b), if the three variables *start*, *end*, and *dir* are omitted, the program can be written as:

```
01 for (i=0; i<n-1; i++)
02     for (j=_____; j_____; j_____)
03         if (x[j]>x[j+1])
04             swap(x, j, j+1);
```

- (d) [5] If we add a variable to record the number of swaps, what is the relationship between this figure and the number of swaps recorded in the original bubble sort algorithm?