1131 NTOUCSE 程式設計 1C 期末考

姓名：________________ 系級：________________ 學號：_________________ **113/12/24** (Tue.)

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. A basic string in C is a char array terminated with special ASCII value '\0'. Please answer the following questions:

(a) [3] An array defined as char buf[10]; is capable of storing upto 10 English or multi-byte characters encoded as ASCII/Big5/Utf-8/Utf-16. Why do we put a terminating character at the end of the storing character sequence? What is this terminating character?

(b) [4] The following definitions of *str*1 and *str*2 at line 01 and line 02 are two ways to define character strings to be used in the program. Is there any difference between the strings printed out on the screen by the statements at line 03 and line 04? Please show me an example that makes a difference for these two kinds of string definitions.

```
01 const char *str1="Hello World!";
02 char str2[]="Hello World!";
03 printf("%s", str1);
04 printf("%s", str2);
```

(c) [4] What is the data type of a string constant "Hello World!" to the compiler except the usage at line 02 of part (b)? Please explain from the grammar aspect for the output ‘o’ in executing the statement printf("%c", *("Hello World!"+4));

(d) [8] The comments at lines 05,06,07 are the corresponding outputs of these three lines at a particular run. Please explain why these values are observed and their numerical differences in particular. What will be the output result if the statement at line 07 were modified as printf("%p\n",buf+10);? Where in the memory will the inputs from the keyboard be written for line 08? How many characters at most can be input from the keyboard without showing any memory access error for line 08? Where in the memory will the inputs be written for the execution of line 09? Will there be any error at execution?

```
01 #include <stdio.h>
02 int main()
03 {
04         char buf[]="SweetHome";
05        printf("%p\n", buf);       // 0000000000CFF938
06        printf("%p\n", buf+1);    // 0000000000CFF939
07        printf("%p\n", &buf+1); // 0000000000CFF942
08        scanf("%s", buf+1);
09        scanf("%s", &buf+1);
10        return 0;
11 }
```

2. Please complete the following two versions of C(n,k) enumeration. Version 1 is iterative and version 2 is recursive.

   (a) [6] iterative version: Basically this program counts in an integer array *x*[] that holds *k* digits. For example, all the C(5,3)=10 combinations of 3 elements chosen from the set {1,2,3,4,5} are shown in order in the figure on the righthand side. Please complete the rules for incrementing a digit and calculating its carry operation at line 07-13 of the following program. Line 07 compares the *i*-th digit with its upper limit, lines 10 and 11 set up the initial values for the digits from (*i*+1)-th place to (*k*-1)-th place after the increment of the *i*-th digit.

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

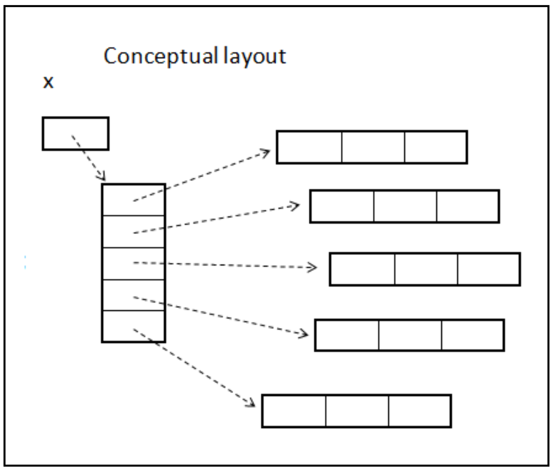
```
01 #include <stdio.h>
02
03 int next(int x[], int n, int k)
04 {
05        int i, j;
06        for (i=k-1; i>=0; i--)
07             if (x[i] < ________________)
08             {
09                  x[i]++;
10                  for (j=____________; j<____________; j++)
11                       x[j] = ______________
12                  return 1;
13             }
14        return 0;
15 }
16
17 int main()
18 {
19        int i, x[10], n, k;
20        scanf("%d%d", &n, &k);
21        for (i=0; i<k; i++)
22             x[i] = i+1;
23        do
24             for (i=0; i<k; i++)
25                  printf("%d%c", x[i], "\n "[i<k-1]);
26        while (next(x, n, k));
27        return 0;
28 }
```

   (b) [10] recursive version: It is shorter than the iterative version in part (a) and is logically more clear to implement a variable layer of for loops to enumerate all possibilities. This version implements the basic Depth First Search (DFS) mechanism. You need to have the decision tree in mind at the design phase and know the recursive template contains three important parts: the state defined by the parameters of the recursive function, the check of terminating condition, and the enumeration of multiple possibilities. In the following program, the recursion state is the *p*-th digit to be filled.

Please complete the definition at line 02. The terminating condition is checked at line 04. The enumeration of all possibilities is at line 08 and the recursive function call to the next level of the decision tree is at line 09. What is the grammatical effect of the third parameter *x*+1 at line 15 and what is its logical reason?

```
01 #include <stdio.h>
02 void comb(int n, int k, int x[], ____________)
03 {
04       if (___________)
05             for (int i=0; i<k; i++)
06                   printf("%d%c", x[i], "\n "[i<k-1]);
07       else
08             for (x[p]=____________; x[p]<___________; x[p]++)
09                   comb(n, k, x, _________);
10 }
11 int main()
12 {
13       int n, k, d[10]={0};
14       scanf("%d%d", &n, &k);
15       comb(n,k,d+1,0);
16       return 0;
17 }
```

3. (a) [9] Please complete the following code segment to dynamically allocate a two-dimensional integer array *x* shown in the right figure for storing the *m* row by *n* column integer matrix presented to the program in the following input stream. (The first line contains integers *m* and *n*, following by *m* lines, each line have *n* integers) The memory allocated for this array would be released after this array is passed into the function release() at line 28. Please define variable *x* at line 14 and use malloc() function in stdlib.h to allocate memory for the pointer array as shown in the center part of the figure at line 18. Please use the compile time macro sizeof() to calculate the number of required bytes. At line 20, please use malloc() again to allocate the five independent memory segments shown in the right part of the figure. At line 24, please use scanf() to read the data from the input stream into the allocated array *x*. At line 28, the function release(*x*, *m*) is called to release all the allocated memory. Please define the parameter *x* for the release() function at line 04 and use free() function in stdlib.h to release the allocated memory completely at line 07 and line 08.

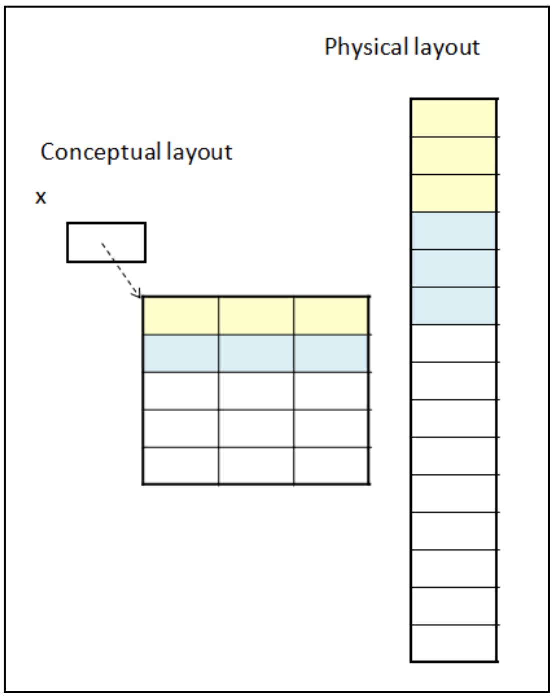


```
5 3
1 3 2
7 1 2
6 5 3
2 3 7
6 0 4
```

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
```

```
void release(int ______________, int m)
{
        for (int i=0; i<m; i++)
                _______________
        ________________
}

int main()
{
        int i, j, m, n;
        int ________________;

        scanf("%d%d", &m, &n);

        x = ____________ malloc(_________________*m);
        for (i=0; i<m; i++)
            x[i] = ________________ malloc(_________________*n);

        for (i=0; i<m; i++)
                for (j=0; j<n; j++)
                        scanf("%d", _________________);

        // some processing on data stored in array x

        release(x, m);
        return 0;
}
```

(b) [10] Please compete the following code segment according to the functional descriptions in part (a) to dynamically allocate a two-dimensional array *x* using the memory layout shown in the righthand side figure. This array is again used to store the *m* by *n* integer matrix in the input stream. Please finish the code of the function release(*x*) to release the allocated memory for the array *x* being passed in.



```
#include <stdio.h>
#include <stdlib.h>

void release(int __________)
{
        ______________;
}

int main()
{
        int i, j, m, n;
        int ______________;

        scanf("%d%d", &m, &n);

        x = (___________) malloc(______________*m);

        for (i=0; i<m; i++)
                for (j=0; j<n; j++)
                        scanf("%d", _________________);
21
22        // some processing on data stored in array x
23
24        release(x);
25        return 0;
26 }
```

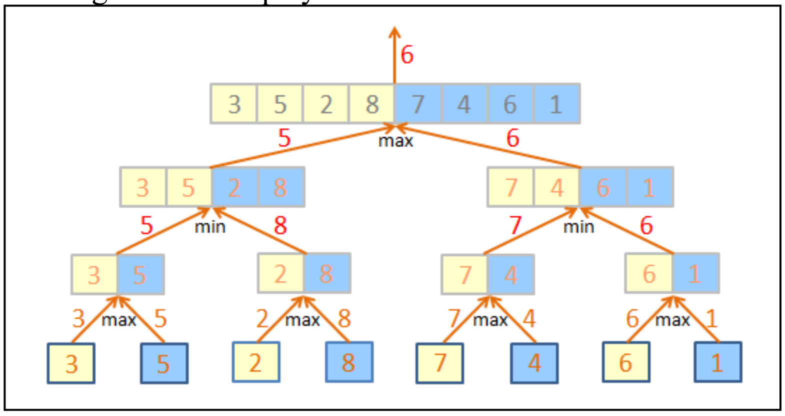
4. (a) [12] We have cover the Divide and Conquer strategy in presenting the design of recursive function. We have also practice the in-place quicksort algorithm in week 13. In some applications, we only need to find the *k*-th element of a sequence without putting the whole sequence in order. At this time we can implement the quick select algorithm, which use the same divide and conquer strategy like quick sort, with a recursive function. Please complete the following quickSelect() recursive function to find the *k*-th element in an unordered sequence. (In the following code segment, it is assumed that we have a function ‘int partition(int d[], int start, int end)’ that splits the data in an array into two halves according to the first element, the pivot, in the array. The pivot element is put into its right place as if the whole array is put in ascending order. The first half consists of elements less than the pivot and the second half consists of all elements equal or larger than the pivot. The function partition() returns the index of the result array which holds the pivot.)

```
01 int quickSelect(int k, int d[], int start, int end) //data is stored in d[start], d[start+1], …, d[end]
02 {
03        int t, pivot;
04        if (end-start > 1)
05        {
06             pivot = ____________________
07             if (pivot>k-1)
08                  return _______________________
09             else if (pivot<k-1)
10                  return __________________________
11        }
12        else if (end-start == 1)
13        {
14             if (d[start]>d[end])
15                  t=d[end], d[end]=d[start], d[start]=t;
16        }
17        return k-1;
18 }
19
20 int main()
21 {
22        int k=8, ans, data[]={-8, 20, -25, -15, 18, -49, 29, 15, 13, 9}, ndata=10;
23        ans = quickSelect(k, data, 0, ndata-1);
24        printf("the %d-th element is %d\n", k, data[ans]);
25        return 0;
26 }
```

(b) [4] Upon execution of the above quickSelect() function, we observe the output ‘**the 8-th element is 18**’. Many elements in the data array are modified by this function. Which elements of the array at lease must be in their correct places after the execution? ____________________ (please explain your reasons)

(c) [4] It appears that the purpose of line 12 to line 16 only put the sequence into correct order, how does this help in finding the *k*-th element of the sequence?

5. There are $n=2^k$ players in an online game. Each player has its own score. The winner of this game is determined by the comparison of these scores. However, the procedure to find out the winner is not simply finding maximum or minimum. Instead, as shown in the following figure, n players are first divided into $n/2$ players on the left side and $n/2$ players on the right side and the winners for both side are found out independently. These two winners then determine the winner of the overall game. Now the $n/2$ players on the left(right) side are splitted again into $n/4$ players and their winners determined separately to contribute to the winner of these $n/2$ players. This procedure of winner determination terminates until there is only one player in the subgroup which is clearly the winner of that subgroup. Now the winner determination criteria is maximum for the top level, minimum for the second level, keeps alternating till the level next to bottom.



For example there are $n=2^3$ players in the above figure. At the top level, two groups{3,5,2,8}, {7,4,6,1} are divided. The subgroups are divided again and again until only one player is in the subgroup. Now at the bottom level, 3 is compared with 5 for the maximum and 5 wins, 2 is compared with 8 for the maximum and 8 wins, 7 is compared with 4 for the maximum and 7 wins, and 6 is compared with 1 for the maximum and 6 wins. For the next level upward, 5 is compared with 8 for the minimum and 5 wins, 7 is compared with 6 for the minimum and 6 wins. For the 2nd level, 5 is now compared with 6 for the maximum and 6 wins. The score for the final winner of this game is 6. Because the winner of n players can be divided into two sub-problems of n/2 players and the final winner can be determined from these two sub-problems, this problem is suitable to be solved by a recursive function. Please complete the following design of a recursive function findScore() as follows:

(a) [2] Please define the recursive function findScore() at line 06 to calculate the score for the winner of a group of players, its parameter includes the number of players *n*, the integer array *x*[] to store all scores of the players, and an integer *type* to store the way of determination for the topmost level (1 denotes maximum and 0 denotes minimum), this function returns the score of the winner

(b) [2] Line 09 is the check for the terminating condition of the recursive function

(c) [2] Line 10 and line 11 recursively call findScore() and pass suitable parameters for calculating the score of the winner of the left side and the score of the winner of the right side.

(d) [2] Line 13 and line 15 calculates the score of the winner according to the criteria specified by the *type* parameter

(e) [2] Line 25 is the initial call for findScore() function in the function main()

```
01 #include <stdio.h>
02
03 int max(int x, int y) { return x>y ? x : y; }
```

```
04 int min(int x, int y) { return x<y ? x : y; }
05
06 __________ findScore(__________, __________, __________)
07 {
08         int left, right;
09         if (_____________) return ______________;
10         left = findScore(______, _______, ______);
11         right = findScore(______, _______, ______);
12         if (type)
13                 return _______________
14         else
15                 return _______________
16 }
17
18 int main()
19 {
20         int i, n, x[1024];
21         while (1==scanf("%d", &n))
22         {
23                 for (i=0; i<n; i++)
24                         scanf("%d", x+i);
25                 printf("%d\n", findScore(______, _______, ______));
26         }
27         return 0;
28 }
```

(f) [5] Although the above recursive function is correct, this program runs a little bit slow for a large number of players at real test. If you take a closer look at the recursion for its depth of recursive function calls, you might find that the depth is $k+1$ for $n=2^k$. If we can make the depth to k, the number of recursive function calls would reduce by a half. Since there are no complex calculations in this function, the speedup for the new function would be almost two times the original function. Please modify the check of terminating condition for the recursive function to realize this speedup.

(g) [3] Please define a two-element array *fn*[] of function pointers and initialize them to the functions min() max().

(h) [3] Please use the array of function pointers *fn* defined in part (g) to replace the codes from line 12 to line 15 as follows:

```
return ___________(left, right);
```