Exam Imperative Programming

Tuesday 10 November 2015, 18:30-21:30h

- You can earn 90 points. You will get 10 points for free. So, you can obtain 100 points in total, and your exam grade is calculated by dividing your score by 10.
- This exam consists of 5 problems. The first two problems are multiple choice questions, and must be answered on the (separate) answer sheet. The problems 3, 4, and 5 are made using a computer.
- The problems 3, 4, and 5 are (partly) assessed by the Justitia system. If Justitia accepts a solution, then half of the points for the corresponding problem are awarded automatically. The other half of the points are awarded by manual grading after the exam. The manual grading focuses on efficiency, style, and correctness. For example, if a recursive solution is requested, then Justitia will accept a correct iterative solution while the manual assessment will refute the solution.
- This is an open book exam! You are allowed to use the reader of the course, and the prescribed ANSI C book. Any other documents are not allowed.
- Do not forget to hand in the answer sheet for the multiple choice questions. You are allowed to take the exam text home!

Problem 1: Assignments (20 points)
For each of the following annotations determine which choice fits on the empty line (.....). The variables `x`, `y` and `z` are of type `int`. Note that `X`, `Y` and `Z` (uppercase!) are specification-constants (so not program variables).

```
1.1  /* 2*y + x > 12 */
     .....
     /* x > 11 */

    (a) x = x - 2*y - 1;
    (b) y = y/2 + 1;
    (c) x = 2*y + x - 1;
```

```
1.2 /* 6*x + 3*y + z == 3*X - 2*z */
    .....
    /* z == X */

    (a) x = 2*x + y; z = z + x;
    (b) y = -2*x; z = z/3;
    (c) z = (6*x + 3*y + 2*z)/3;
```

```
1.3 /* 7*X - 3 <= x < 7*X + 4 */
    .....
    /* x == X */

    (a) x = (x - 3)/7;
    (b) x = (x + 3)/7;
    (c) x = 7*x + 3;
```

```
1.4 /* z + y > 4 */
    x = z + 1; y = x + y;
    .....

   (a) /* x + y + z > 3 */
   (b) /* x + y + z > 4 */
   (c) /* y > 5 */
```

```
1.5 /* x == X, y == Y, z == Z */
    z = x + y + z; x = z - x - y; y = z - x;
    .....

    (a) /* x == Y, y == Y - Z, z == X + Y + Z */
    (b) /* x == Z, y == X + Y, z == X + Y + Z */
    (c) /* x == X, y == Y - X, z == X + Y + Z */
```

```
1.6 /* x == 3*X - 2*Y + 1, y == X */
    y = (3*y - x + 1)/2; x = (x + 2*y - 1)/3;
    .....

    (a) /* x == X, y == Y */
    (b) /* x == Y, y == X */
    (c) /* x == X + Y, y = X - Y */
```

Problem 2: Time complexity (20 points)
In this problem the specification constant `N` is a non-zero natural number (i.e. `N>0`). Determine for each of the following program fragments the sharpest upper limit for the number of calculation steps that the fragment performs in terms of `N`. For a fragment that needs `N` steps, the correct answer is therefore $O(N)$ and not $O(N^2)$ as $O(N)$ is the sharpest upper limit.

1.
   ```
   int i = 0, s = 0;
   while (s < N) {
     s += i;
     i++;
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

2.
   ```
   int i, j, s = 0;
   for (i=0; i < N; i++) {
     for (j=N; j > i; j--) {
       s += j;
     }
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

3.
   ```
   int i = 0, j = N, s=0;
   while (j-i > 1) {
     int k = (i+j)/2;
     if (i%2 == 0) {
       i = k;
     } else {
       j = k;
     }
     s++;
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

4.
   ```
   int i, j=0, s=0;
   for (i=0; i < N; i++) {
     j += i;
   }
   while (j > 0) {
     s += j;
     j--;
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

5.
   ```
   int i=0, s=1;
   for (i=0; i < N; i+=2) {
     s = 2*s;
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

6.
   ```
   int i, j, s = 0;
   for (i=0; i < N; i++) {
     for (j = i*i; j > 0; j = j/2) {
       s++;
     }
   }
   ```
   (a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

Problem 3: bit reversal of integers (10 points)
The binary representation of the number 23 is 10111, since it is equal to $1\times2^4+0\times2^3+1\times2^2+1\times2^1+1\times2^0 = 16+4+2+1$. If we reverse the bit string 10111 we get 11101 which represents the number $29 = 1\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0 = 16+8+4+1$. Therefore, we call 29 the *bit reversal* of 23. Note that integers on a PC are represented by 32 bits, so the binary representation of 23 on a PC consists actually of 27 leading zeros followed by 10111. In this problem we omit all leading zeros. Therefore, the bit reversal of the number 1 is simply 1 itself.

The following incomplete code fragment is available in Justitia. Download it and complete the code. The body and the return type of the function `bitrev` are missing. The function should return the number that is the bit reversal of the argument of the function. You are not allowed to make changes in the `main` function.

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

.... bitrev(int n) {
  /* implement the body of this function and choose the correct return type */
}

int main() {
  int n;
  scanf ("%d", &n);  /* you may assume that n >= 0 */
  printf("bitrev(%d)=%d\n", n, bitrev(n));
  return 0;
}
```

Example 1:
input:
```
29
```
output:
```
bitrev(29)=23
```

Example 2:
input:
```
23
```
output:
```
bitrev(23)=29
```

Example 3:
input:
```
3
```
output:
```
bitrev(3)=3
```

Problem 4: job scheduling (20 points)
A couple of scientists want to run jobs on a super computer. Jobs on this super computer are executed one after another (serially). Each scientist knows in advance how long his job will run. They are all eager for their results, and therefore they do not want to wait too long for their results. The system administrator of the computer decides to run the jobs in an order that minimizes the *average waiting time*. The waiting time for a scientist is the time between supplying the job (for all jobs, this is time $t = 0$) and the time that his job finished.

As a simple example, consider the case with only two jobs. The expected execution time for job 1 is 10 hours, while the execution time of job 2 is 4 hours. If we run job 1 first, then the total waiting time is 10 + (4+10) = 24 hours (and hence the average waiting time is 12 hours). However, if we run job 2 first, then the total waiting time will be 4 + (10+4)=18 hours (and the average waiting time would be 9 hours). Therefore, the system administrator decides to run job 2 first.

Write a program that computes the order in which the jobs should be run in order to minimize the average waiting time. The first line of the input is an integer `n`, the number of jobs. You may assume that `1 < n < 100`. The rest of the input consists of `n` lines, which consist of the running times of the `n` jobs (in hours). You may assume that all running times are integer numbers less than 100. Of course, the first running time belongs to job 1, the second to job 2, and so on. The output should consist of 1 line, which is the order in which the jobs need to be run. The job numbers are separated by a comma. The output line must end with a newline (i.e the `'\n'` character). Note, that the first job has job number 1. If two (or more) jobs have the same running time, then these jobs must be ordered based on their job number (smallest number first).

Example 1:
input:
```
2
10
4
```
output:
```
2,1
```

Example 2:
input:
```
3
10
10
4
```
output:
```
3,1,2
```

Example 3:
input:
```
3
4
10
10
```
output:
```
1,2,3
```

Problem 5: ordered bit strings using recursion (20 points)
In this problem we consider bit strings, i.e. sequences consisting solely of zeros and ones. Consider the following example bit string: 010011. This string satisfies the property that *each* prefix (i.e. leading substring) of the string (including the string itself) contains at most as many ones as it contains zeros. It is easy to verify this property for all prefixes of the example string: 0, 01, 010, 0100, 01001, and 010011. An example of a string for which this property does not hold is the string 1100, since each of the prefixes 1, 11, 110 contain more ones than zeros.

The input of this problem is an integer `n`, where $0 < \mathtt{n} \leq 10$. The output must be all bit strings of length 2`n`, containing `n` zeros and `n` ones, that satisfy the above mentioned property. Moreover, the strings must be ordered in lexicographical (dictionary sorted) order. Each string must be printed on a separate line without spaces or tabs (see output examples).

The following incomplete code fragment is available in Justitia. Download it and complete the code. You are asked to implement the body of the function `showBitStrings`. This function should call a recursive helper function (with suitably chosen parameters/arguments) that solves the problem. You are not allowed to make changes in the `main` function.

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

void showBitStrings(int n) {
  /* Implement the body of this function.
   * Moreover, this function should call a recursive helper
   * function that solves the problem.
   */
}

int main() {
  int n;
  scanf ("%d", &n);   /* note: it is guaranteed that 0<n<=10 */
  showBitStrings(n);
  return 0;
}
```

Example 1:
input:
```
1
```
output:
```
01
```

Example 2:
input:
```
2
```
output:
```
0011
0101
```

Example 3:
input:
```
3
```
output:
```
000111
001011
001101
010011
010101
```