Exam Imperative Programming

7 November 2013, 14:00-17:00h

- Write on each sheet of paper your name, student number and study (discipline). Number each sheet, and write on the first sheet the total number of sheets.
- You can earn 90 points. You will get 10 points for free.
- Write neatly and carefully with a pen (do not use a pencil).

Exercise 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  /* x == X */
     .....
     /* x == 42*X + 21 */

    (a) x = x/42 - 21;
    (b) x = 42*x + 21;
    (c) x = (x - 21)/42;
```

```
1.2 /* x == 42*X + 21 */
    .....
    /* x == X */

    (a) x = x / 42 - 21;
    (b) x = 42*x + 21;
    (c) x = (x - 21)/42;
```

```
1.3 /* x == y*y */
    .....
    /* x == y*y */

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

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

    (a) /* x + y + z > 5 */
    (b) /* y + z > 6 */
    (c) /* y > 5 */
```

```
1.5 /* x == X, y == Y */
    x = x + y; y = 2*x - y; x = y - 2*x; y = x + y;
    .....

    (a) /* x == Y, y == X */
    (b) /* x == -Y, y == 2*X */
    (c) /* x == 2*Y, y == -X */
```

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

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

Exercise 2: Find the 5 errors (10 points)

The following program fragment (it is not a complete program) is supposed to implement the quicksort algorithm. There are, however, five errors in this implementation. Find them, and give of each error the line number and a correction.

```
 1 void swap(int i, int j) {
 2   int h;
 3   h = arr[i];
 4   arr[i] = arr[j];
 5   arr[j] = h;
 6 }
 7
 8 int partition(int length, int arr[]) {
 9   int left, right, pivot;
10   left = 0;
11   right = length;
12   pivot = arr[0];
13   while (left < right) {
14     while ((left < right) && (arr[left] <= pivot)) {
15       left++;
16     }
17     while ((left < right) && (pivot <= arr[right-1])) {
18       right--;
19     }
20     if (left < right) {
21       /* (arr[left] > pivot) && (arr[right-1] <= pivot) : swap */
22       right--;
23       swap(left, right, arr);
24       left++;
25     }
26   }
27   /* put pivot in right location: swap(0,left-1,arr) */
28   left--;
29   arr[0] = arr[left];
30   return left;
31 }
32
33 void quickSort(int length, int arr[]) {
34   if (length <= 1) {
35     /* empty or singleton array: nothing to sort */
36     return;
37   }
38   boundary = partition(length, arr);
39   /* recursively sort the two partitions */
40   quickSort(boundary, arr);
41   quickSort(length - boundary, &arr[boundary+1]);
42 }
```

Exercise 3: Time complexity (20 points)

In this exercise 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 an algorithm 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, j = N;
while (i < j) {
  i++;
  j/=2;
}
```
(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=i; j < N-i; j++) {
    s += i + 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 = 0;
while (i < N) {
  i += 2*j + 1;
  j++;
}
```
(a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

4.
```
int i = 1, j = N;
while (i < j) {
  i += i;
  j--;
}
```
(a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

5.
```
int i, j, s = 0;
for (i=1; i < N; i*=3) {
  for (j=1; j < i; j++) {
    s += j;
  }
}
```
(a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

6.
```
int i, j, s = 0, t = 0;
for (i=1; i < N; i++) {
  s += i;
  for (j=1; j < s; j*=2) {
    t += j;
  }
}
```
(a) $O(\log N)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(N \log N)$ (e) $O(N^2)$

Exercise 4: Iterative algorithms (10+10 points)

(a) The integer 125874, and its double, 251748, contain exactly the same digits, but in a different order. There exists a non-zero positive integer `x` such that `2*x`, `3*x`, `4*x`, `5*x` and `6*x` all contain the same digits as `x`. Write a program that computes this number `x`. [Note: The number is less than the maximum integer/6, so you do not need to worry about integer overflow]

(b) Given is the declaration of some square 2-dimensional array: **int** `square[N][N];`
You may assume that the array is already filled with data. Write a program fragment that determines whether the array is a *Latin square*. This means that each row and each column must contain the values `1, 2, ..., N` with no repeats. The time complexity of your solution must not exceed $O(N^2)$.

Exercise 5: Recursive algorithms (5+15 points)

(a) Write a recursive function `mul` with the following prototype: `int mul(int a, int b);`
The function call `mul(a, b)` should return the value `a*b`. You are not allowed to use loops (only recursion), and you are only allowed to use addition and subtraction (multiplication or division is not allowed). Note that `a` or `b` might be a negative number.

(b) Consider the integer sequence `s=[1, 4, 2, 6, 7, 3, 7, 8, 3, 9, 0]`. The sequence `t=[1,6,7,9]` can be obtained from the sequence `t` by crossing out elements from the sequence `s`:

[1, ~~4~~, ~~2~~, 6, 7, ~~3~~, ~~7~~, ~~8~~, ~~3~~, 9, ~~0~~]

In fact, there is another way:

[1, ~~4~~, ~~2~~, 6, ~~7~~, ~~3~~, 7, ~~8~~, ~~3~~, 9, ~~0~~]

These are the only two ways that we can obtain `t` from `s`. Note that the sequence `[1,2,4]` can not be obtained from `s` by crossing out elements, since the elements occur in `s` in the wrong order.
Write a recursive function that determines the number of ways that one sequence (an **int** array) can be obtained from another sequence by crossing out elements.