Exam for Imperative Programming

November 8, 2012, 14:00-17:00

- Your name, Student ID, Study Orientation and Test-version need to be written atop each page you will be handing in.
- Your maximum score is 100 and starts of at 10. Each problem is tagged with its maximum score attainable.
- Read all details affixed to the problem before solving it.
- Maintain a legible handwriting. Don't use pencils, only pens.
- The examination lasts 3 hours. Try to double-check your answers if you have time left after finishing the test.
- Good luck!

Problem 1: Assignments (Score: 20)
Fill the gaps with the correct line. With each problem, three possible solutions are specified.
Exactly one of those solutions is the correct one. Lower-case identifiers `x`, `y` and `h` represent variables of the type `int`.
Upper-case identifiers `X`, `Y` and `Z` represent constants and are immutable.

1.1
```
/* x == X */
.....
/* x == 3*X + 1 */
```

```
(a) x = x/3 - 1;
(b) x = 3*x + 1;
(c) x = (x - 1)/3;
```

1.2
```
/* x == 3*X + 1 */
.....
/* x == X */
```

```
(a) x = x/3 - 1;
(b) x = 3*x + 1;
(c) x = (x - 1)/3;
```

1.3
```
/* x == y*y + X, y + 1 == Y */
.....
/* 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 = x + 2*y - 1; y = y - 1;
```

1.4
```
/* x == X, y == Y */
x = x + 2*y; y = x - 2*y; x = x - y;
.....
```

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

1.5
```
/* x == X + Y, y == X + Z, z == Y + Z */
y = (y + z - x)/2; z = x - z + y; x = x - z;
.....
```

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

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

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

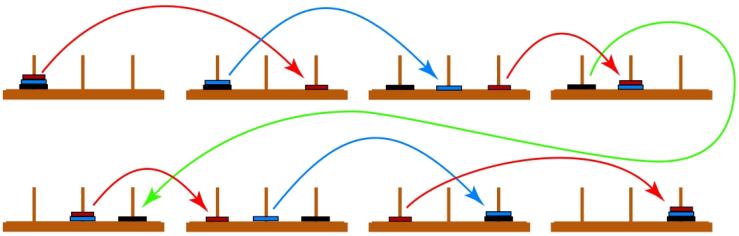
Problem 2: Find 5 errors (Score: 10)

"Towers of Hanoi" is a stacking game consisting of thee pins and differently sized disks with holes in the centre. Disks may be stacked onto each of the pins. The game starts off with all disks stacked onto one pin (Pin 1) - sorted from large to tiny starting at the bottom. The goal of the game is to move the entire stack onto another pin (Pin 3). To add a bit of challenge to this otherwise trivial action, the following rules can not be broken:

- Only one disk at a time may be moved between pins
- At no time may a disk be placed on top of a smaller disk

The program in the listing below plays the game recursively. It will output the moves required to beat the game, in order. It will conclude the output with a specification of the total amount of moves.

```
Disk on Pin 1 moves to Pin 3
Disk on Pin 1 moves to Pin 2
Disk on Pin 3 moves to Pin 2
Disk on Pin 1 moves to Pin 3
Disk on Pin 2 moves to Pin 1
Disk on Pin 2 moves to Pin 3
Disk on Pin 1 moves to Pin 3
Amount of moves: 7
```



The listed program, however, contains exactly 5 errors. Specify the line number and correction of each error.

```
 1 #include <stdio.h>
 2
 3 void showMove(int from, int to) {
 4   printf ("Disk on Pin %d moves to Pin %d\n", from, to);
 5 }
 6
 7 int hanoi(int amountofDisks, int from, int to) {
 8   int amount;
 9   if (amountofDisks == 0) {
10     /* Identity case: Do nothing if no disks need to be moved */
11   } else {
12     /* Recursion case: Disks need to be moved */
13     int via = 6 - from - to;
14     amount = hanoi(amountofDisks, from, via);
15     showMove();
16     amount = amount + hanoi(amountofDisks-1, via, to);
17   }
18   return amount;
19 }
20
21 int main (int argc, char *argv[]) {
22   amount = hanoi(3, 1, 3);
23
24   printf("Amount of moves: %d\n", amount);
25   return 0;
26 }
```

Problem 3: Complexity (Score: 20)
In this problem, `N` is a natural number to which `N>0` applies. For each of the listed code snippets below, specify the absolute maximum amount of cycles it needs using `N` as a term. An algorithm cycling N times, would be $O(N)$ and not for example $O(N^2)$ for $O(N)$ is the upper bound to an algorithm cycling N times.

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

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

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

Problem 4: Simple Algorithms (Score: 20)
(a) An array `b` is a *permutation* of an array `a` only if `b` can be obtained simply by reordering the elements of array `a`. For example, the array `b=[1, 2, 4, 3, 5]` is a permutation of the array `a=[1, 2, 3, 4, 5]`. Also, the array `b=[1, 2, 4, 3, 5]` is not a permutation of `a=[1, 2, 3, 4, 1]` since `a` does not contain `5`.
Write a function `isPermutation` that determines whether two arrays with the same length are a permutation of each other. You may assume the arrays only contains numbers (0, 1, .. 9). The function has to return 1 (true) if the lists are a permutation of each other, and 0 (false) otherwise. The prototype of the function has to be: `int isPermutation(int length, int a[], int b[])`

[Grading: You will receive 10 points for a correct algorithm with a time complexity of O(`length`), and 5 points for a correct algorithm with a worse time complexity.]

(b) De square root of 252 is equal to 6 times the square root of 7, because $\sqrt{252} = \sqrt{2^2 \cdot 3^2 \cdot 7} = 2 \cdot 3 \cdot \sqrt{7} = 6\sqrt{7}$. WWe cannot simplify $\sqrt{7}$ any further and therefore $6\sqrt{7}$ is the simplification of $\sqrt{252}$. The simplification of the square root of 504 (=$2 \cdot 252$) is thus $\sqrt{504} = \sqrt{2^3 \cdot 3^2 \cdot 7} = 2 \cdot 3 \cdot \sqrt{2 \cdot 7} = 6\sqrt{14}$.
Write a C-function `simplifySqrt(int n)` that prints the simplification of `n` to the screen. You may assume that `n` is never a negative number.
The call `simplifySqrt(252)` has to print: `sqrt(252)=6*sqrt(7).`
The call `simplifySqrt(36)` has to print: `sqrt(36)=6`.

Problem 5: Recursive algorithms (Score: 20)
(a) In this exercise we observe the following process. You start with a real number a (where $a > 0$) and may repeatedly apply one of the following operations:

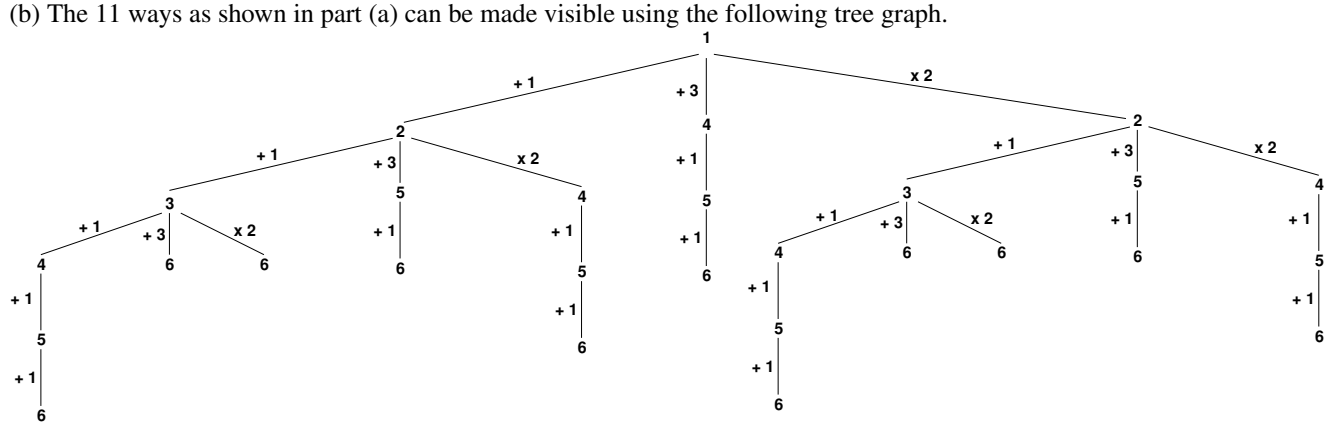
- increment a with 1
- increment a with 3
- double a

Using this process (starting with $a = 1$) we can obtain the number 6 using 11 different ways, as shown here:

$1+1+1+1+1+1$	$1+1+3+1$	$1 \times 2+1+1+1+1$	$1 \times 2+3+1$
$1+1+1+3$	$(1+1) \times 2+1+1$	$1 \times 2+1+3$	$(1 \times 2) \times 2+1+1$
$(1+1+1) \times 2$	$1+3+1+1$	$(1 \times 2+1) \times 2$	

Write a recursive function with two arguments `a` and `n`, that returns the number of possible solutions to reaching the number `n` starting from `a`. The function is allowed to have an exponential time complexity (a brute force solution is ok). A more efficient solution is asked in the (b) part of the exercise.

(b) The 11 ways as shown in part (a) can be made visible using the following tree graph.



In this tree there are quite a few branches equal to each other: the left and right branch are identical. Make use of this property to program a more efficient recursive solution to the problem described in part (a). Make use of dynamic programming.