COL100 – Minor exam 2 solutions

March 24, 2018

1 Exam begins here

1. (**1 point**) Suppose we define the following *recursive* data type to represent *natural numbers*.

```
type nat =
  | Zero
  | Succ of nat
```

Give an expression to construct the number 4 (that is, `let four = ...`).

```
let four = Succ (Succ (Succ (Succ (Zero))))
```

2. (**5 points**) Write a function `addnat: nat -> nat -> nat`, that takes in two natural numbers (as defined in the previous question) and returns their sum. You may define additional functions if required.

```
let rec getnum a =
match a with
| Zero -> 0
| Succ y -> getnum y + 1;;

let rec constructnum a =
match a with
| 0 -> Zero
| k -> Succ(constructnum (k-1));;

let addnat a b =
constructnum (getnum a + getnum b);;
```

3. (**2 points**) State the steps involved in proving a statement $P(n)$ using mathematical induction.

There are three steps that we need to show.
First, the *basis step*, shows that $P(0)$ is true
Second, we assume the *induction hypothesis* that for some $k > 0$, $P(k)$. (Also acceptable: assume that for all $0 < k < m$, $P(k)$ holds)
Third, we show the induction step. Based on the induction hypothesis, $P(k+1)$ also holds.
Therefore, we conclude that $P(n)$ holds.

4. (**2 points**) Precisely define $\Theta(g(n))$. What is the role of $\Theta(g(n))$ in the asymptotic analysis of an algorithm (state in just 1 or 2 sentences *only*).

$\Theta(g(n)) = \{f(n) : \text{there exist } c_1 > 0, c_2 > 0 \text{ and } n_0 \text{ s.t.} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$
$\Theta(g(n))$ bounds the runtime $f(n)$ of the given algorithm from both above and below.

5. **(3 marks)** Prove that the following function `sum` correctly returns the sum of 2 positive integers. If it does not return the sum, state and correct the mistake and then provide a correctness proof.

```
let rec sum a b =
  if a = 0 then b
     else (sum (a-1) b) + 1;;
```

To prove: The function $sum(a, b)$ returns the sum of two positive integers a and b.
Basis: Consider $a = 0$. $sum(a, b) = b$ as required.
Induction hypothesis: For some $k = a - 1$, $k \geq 0$, we have that $\forall b$, $sum(k, b) = k + b = a - 1 + b$.
Induction step:
Consider $sum(a, b)$

$$\begin{aligned} sum(a,b) &= (sum(a-1), b) + 1 \text{ (from the function)} \\ &= (a-1) + b + 1 \text{ (by induction hypothesis)} \\ &= a + b \end{aligned}$$

Therefore, the statement is proved.

6. (**5 marks**) Prove that the following function `mult` correctly returns the product of 2 positive integers. If it does not return the product, state and correct the mistake and then provide a correctness proof.

```
let rec mult a b =
  if b = 0 then 0
  else if b = 2 * b/2 then 2 * mult a (b/2)
  else a + mult a (b-1);;
```

Mistake in the code: the condition `b = 2 * b/2` should be `b = 2 * (b/2)`. However, you will *not* be penalised for this. Instead, if you pointed this out, you will get bonus marks of 0.5.
To prove: The function $mult(a, b)$ returns the product of two positive integers a and b.
Basis: If $b = 0$, then $mult(a, b) = 0$.
Induction hypothesis: For some $k > 0$, we have $\forall a$, $mult(a, k) = ka$
Induction step:
Case 1: Consider $mult(a, k + 1)$ when k is odd:

$$\begin{aligned} mult(a,k+1) &= 2 * mult(a,(k+1)/2), \text{ if } k \text{ is odd} \\ &= 2 * a * (k+1)/2 \text{(by the induction hypothesis)} \\ &= a * (k+1) \end{aligned}$$

Case 2: Consider $mult(a, k + 1)$ when k is even:

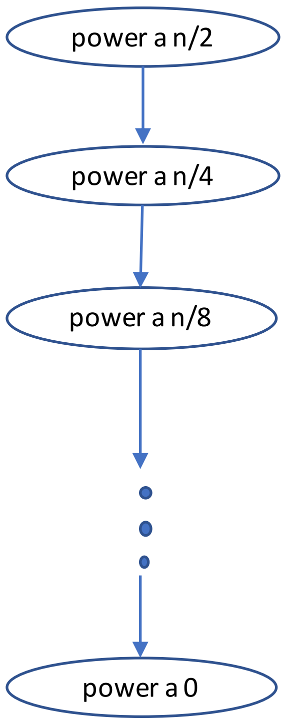
$$\begin{aligned} mult(a,k+1) &= a + mult(a,k), \text{ if } k \text{ is even} \\ &= a + ka, \text{ by induction hypothesis} \\ &= a * (k+1) \end{aligned}$$

Therefore, the statement is proved.

7. **(5 marks)** Derive an expression in terms of the Θ notation for the *runtime complexity* of the following function `power` that computes a positive power of a positive integer. Use the recursion tree method. Show *at least* the first three levels of the tree (including the root).

```
let rec power a n =
   if n = 0 then 1
   else
     let half_pow = power a (n/2) in
     if n mod 2 == 0 then half_pow * half_pow
     else a * half_pow * half_pow;;
```

The key point to note is that there is a recursive call `power a (n/2)` that will determine the complexity of the algorithm, since all the other statements can be executed in *constant* time. Therefore, the recursion tree will look as follows:



The number of levels in this tree is $log\, n$. To see this, note the pattern: in level 0, we call *power* with second parameter $n/2^{(0+1)}$, in level 1, the second parameter is $n/2^{(1+1)}$. In general, if we denote level by l, the second parameter will be $n/2^{l+1}$. But, we need to do this until the second parameter equals 0. That happens when $l = log_2\, n - 1$ (that is $n/2^{(log_2\, n-1)+1} = 1$). And the next iteration, the $log_2\, n^{th}$ iteration, results in the parameter 0 (because in this iteration $n = 1$ and $1/2 = 0$).
It is straightforward to show that $log\, n = \Theta(log\, n)$.