1 RM and Binary Digit Sums (30)

Background
Recall the register machine from class that computes the (binary) digit sum of a given number.

// binary digitsum of X --> Z
0: dec X 1 4
1: dec X 2 3
2: inc Y 0
3: inc Z 4
4: dec Y 5 8
5: inc Y 6
6: dec Y 7 0
7: inc X 6
8: halt

In this problem you are supposed to determine the running time of this particular register machine program. Also, we will give one application of binary digit sums. For simplicity write $\sigma(n)$ for the binary digit sum of $n$, so $\sigma(10) = 2$ and $\sigma(255) = 8$ (of course, the argument is written in decimal).

We write $\overline{x}$ for the complement of bit $x$ and likewise for bit-sequences. Now consider the following bit-sequence $(t_i)_{i \geq 0}$:

\[
\begin{align*}
  t_0 &= 0 \\
  t_{2n} &= t_n \\
  t_{2n+1} &= \overline{t_{2n}}
\end{align*}
\]

Let’s write $T_n$ for the binary word consisting of the first $n$ bits of this sequence. For example,

\[
T_{64} = 011010001100101101000100100110011001101100100110010110110011010011011001
\]

Recall the shuffle operation which interleaves the elements of two sequences of equal length in a strictly alternating fashion:

\[
a \parallel b = a_1b_1a_2b_2\ldots a_kb_k
\]

Given shuffle we can define a sequence of binary words as follows.

\[
S_0 = 0 \\
S_n = S_{n-1} \parallel \overline{S_{n-1}}
\]

As it turns out, $\sigma(n) \mod 2$, $T$ and $S$ all describe the same infinite sequence.
Task

A. Determine the time complexity of the digit sum register machine.

B. Show that $t_n = \sigma(n) \pmod{2}$.

C. Show that $T_{2k} = S_k$.

Comment

For the running time first try to find an approximate answer, and then refine your solution to obtain a precise answer. Elegance counts. It is probably a good idea to take a close look at the flowgraph of the machine.

2 The Busy Beaver Function (RM) (30)

Background

The Busy Beaver function $\beta$ is a famous example of a non-computable function. For our purposes, let’s define $\beta(n)$ as follows. Consider all register machines $P$ with $n$ instructions and no input (so all registers are initially 0). Executing such a machine will either produce a diverging computation or some output $x_P$ in register $R_0$. Define $\beta(n)$ to be the maximum of all $x_P$ as $P$ ranges over $n$-instruction programs that converge.

It is intuitively clear that $\beta$ is not computable: we have no way to eliminate the non-halting programs from the competition. Alas, it’s not so easy to come up with a clean proof. One line of reasoning is to show that $\beta$ grows faster than any (register) computable function.

Task

A. Show that for any natural number $m$ there is a register machine without input using $O(\log m)$ instructions that outputs $m$.

B. Assume $f : \mathbb{N} \to \mathbb{N}$ is a strictly increasing computable function. Show that for some sufficiently large $x$ we must have $f(x) < \beta(x)$.

C. Conclude that $\beta$ is not computable.

D. Prof. Dr. Blasius Wurzelbrunft has a startup that sells a device called HaltingBlackBox™: according to the press release, the gizmo solves the Halting Problem for register machines. Explain how Wurzelbrunft’s gizmo could be used to compute $\beta$.

Comment

The bound in part (A) is far from tight in special cases: some number $m$ have much shorter programs. But in general $\log m$ is impossible to beat (Kolmogorov-Chaitin program-size complexity). Part (D) says that $\beta$ is $K$-computable.
3 Primitive Recursive Loops (40)

Background
Consider a small programming language LOOP with the following syntax:

- **constants**: $0 \in \mathbb{N}$
- **variables**: $x, y, z, \ldots$ ranging over $\mathbb{N}$
- **operations**: increment $x++$
- **assignments**: $x = 0$ and $x = y$
- **sequential composition**: $P; Q$
- **control**: do $x : P$ od

The semantics are obvious, except for the loop construct: do $x : P$ od is intended to mean: “Let $n$ be the value of $x$ before the loop is entered; then execute $P$ exactly $n$ times.” Thus, the loop terminates after $n$ rounds even if $P$ changes the value of $x$. For example, the following LOOP program computes addition:

```loop
1 // add : x, y --> z
2 z = x;
3 do y :
4 z = z++;
5 od
```

Here $x$ and $y$ are input variables, and the result is in $z$. We assume that all non-input variables are initialized to 0. So, we have a notion of a LOOP-computable function.

Task

A. Show how to implement multiplication and the predecessor function as LOOP-programs.

B. What function does the following loop program compute?

```loop
1 // mystery : x --> x
2 do x:
3 do x: x++ od
4 od
```

C. Show that every primitive recursive function is LOOP-computable.

D. Show that every LOOP-computable function is primitive recursive.

Comment
Part (D) is a bit tricky: the LOOP-program contains variables, so you have to keep track of their values during execution.