Computer Science 52

Second Midterm: Topics and Practice Exercises

This document contains a list of topics for the midterm and sample exercises. The evercises do not comprise a complete guide for studying, but they do cover *most* of the topics. There is obviously more material here than would appear on an actual midterm. Many of the individual problems are longer as well, and some are considerably more complicated.

You are free to talk about the exercises with anyone, but they are perhaps best used in a "solo" test-like situation. A solution set will not be distributed.

Assignments 4, 5, and 6 are relevant to the exam. See also the course documents *Logic, Words, and Integers* (especially sections 1 through 6), *Induction, Recursion, and O-notation,* and the first few sections of the CS41B manual.

1. Bits and integers

- base-N representations; conversions
- unsigned and signed (two's complement) binary representations
- negation, addition, subtraction, bitwise logical operations
- error conditions and flags

1.1. Consider six-bit words under the *signed* representation.

a. Give the decimal value and the bit pattern of the largest integer that can be represented in six bits.

b. Negate the value in part a. Express the result as a decimal value and a bit pattern.

c. Give the decimal value and the bit pattern of the smallest (most negative) integer that can be represented in six bits.

d. Negate the value in part c. Express the result as a decimal value and a bit pattern.

1.2. a. Give an example of two six-bit words whose sum is correct when interpreted in both the unsigned and signed representations.

b. Give an example of two six-bit words whose sum is *incorrect* when interpreted in both the unsigned and signed representations.

c. Give an example of two six-bit words whose sum is correct when interpreted as an unsigned value but incorrect as a signed value.

d. Give an example of two six-bit words whose sum is correct when interpreted as a signed value but incorrect as an unsigned value.

2. Logic and gates

- logical connectives
- gates and circuits
- simulation of gates as Python classes

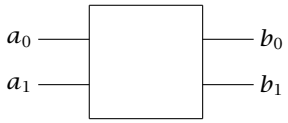
2.1. One of our examples in class was a 2^n-to-1 multiplexer. Consider the reverse idea, a 1-to-2^n multiplexer. It has one input line, n control lines, and 2^n output lines. The control lines select one output line to have the value of the input line; all the other output lines have the value 0.

a. Draw a circuit diagram for a 1-to-2 multiplexer.

b. Using the 1-to-2 multiplexer as a component, draw a diagram for a 1-to-4 multiplexer.

c. Using our Python framework for gates, create a class of 1-to-2^n multiplexers. In keeping with our convention, the class will have just one constructor, which takes an input node, a list of control nodes, and a list of output nodes.

2.2. The component pictured below has two input and two output lines; its behavior is specified by the table on the right.



inputs		outputs	
a_0	a_1	b_0	b_1
0	0	1	0
0	1	1	1
1	0	0	1
1	1	0	1

Draw a simple circuit using gates to implement the component. (On a real midterm, there would be a table of gates.)

3. Subprogram calls and stack frames

- contents of a frame
- counting the number of recursive calls that are active at a given time

3.1. Why does a subprogram push the return address? Are there situations in which it is not necessary?

3.2. The Python code below solves the `n`-disc version of the Towers of Hanoi game. We have eliminated the peg numbers to highlight the recursion.

```
def hanoi(n) :
    if 0 < n :
        hanoi(n-1)
        move one disc
        hanoi(n-1)
```

Each call to `hanoi` creates a stack frame that exists until the call has completed. In terms of `n`, what is the maximum number of stack frames that

exist at any one time during an initial call to `hanoi(n)`? Assume that `n` is non-negative; give an expression in `n` and a brief explanation. Be sure to include the initial call.

4. Recursion, induction, and *O*-notation

- computing use of a resource (number of loop iterations, number of recursive calls, number of data values, etc) as a recursive function
- showing by induction that the recursive function has a closed form
- *O*-notation and intuition

4.1. a. Prove by induction that $0^3 + 1^3 + 2^3 + \ldots + n^3 = n^2(n+1)^2/4$ for $0 \leq n$.

b. Are you surprised that $0^3 + 1^3 + 2^3 + \ldots + n^3 = (0 + 1 + 2 + \ldots + n)^2$?

4.2. As a function of n, how many and-gates are needed for your 1-to-2^n multiplexer in Problem 2.1c?

4.3. Here is one variant of the Fibonacci function.

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1, \text{ and} \\ f(n-2) + f(n-1) & \text{otherwise} \end{cases}$$

Use it to prove, for $2 \leq n$, that $f(0) + f(1) + f(2) + \ldots + f(n-1) = f(n+1) - 1$.

4.4. Here are two SML implementations of the Fibonacci function.

```
fun fibrec 0 = 1
  | fibrec 1 = 1
  | fibrec n = (fibrec (n-1)) + (fibrec (n-2));

fun fibpair 0 = (1,0)
  | fibpair 1 = (1,1)
  | fibpair n = let
                    val (u,v) = fibpair (n-1);
                in
                    (u+v, u)
                end;
```

a. Prove by induction that `fibpair n` evaluates to `(fibrec n, fibrec (n-1))` for $1 \leq$ `n`.

b. Conclude from part a that `fibrec n` evaluates to the same value as `#1(fibpair n)` for $0 \leq$ `n`. (No recursion is necessary. Recall that the function `#1` returns the first component of an ordered pair.)

c. Show that the call `fibrec n` terminates in time $O(2^n)$.

d. Show that the call `fibpair n` terminates in time $O(n)$.

4.5. Prove by list induction on `ut` that `ut @ (vt @ wt) = (ut @ vt) @ wt` for all lists `ut`, `vt`, and `wt`.

4.6. Prove by list induction on `ut` that

```
(rev ut) @ (x::yt) = (rev (x::ut)) @ yt
```

(It is easier to use the `nrev` characterization of the list-reversing function. The result of problem 4.5 may be helpful.)

4.7. Often, we must prove a stronger—or more complicated—result than the one we really want. Here is an example.

Recall the folding functions in SML.

```
fun foldl oper base nil     = base
  | foldl oper base (x::xs) = foldl oper (oper(x, base)) xs;

fun foldr oper base nil     = base
  | foldr oper base (x::xs) = oper(x, foldr oper base xs);
```

a. Prove by list induction on `ut` that the following equation holds for all lists `ut`, `vt`, and `wt`.

```
foldl oper (foldr oper base (ut @ vt)) wt
    = foldl oper (foldr oper base vt) ((rev ut) @ wt)
```

(This one is a bit complicated and will exercise your patience and skill in symbol manipulation. Feel free to use the results from previous problems.)

b. Conclude that `foldr oper base ut = foldl oper base (rev ut)` for all lists `ut`. (No induction is necessary. Start with the equation

```
foldr oper base ut = foldl oper (foldr oper base (ut @ nil)) nil
```

and apply the result from part a.)

4.8. On Assignment 6, we saw the polymorphic type `binTree`, which represented trees having nodes with at most two children. A *2-3 tree* is a tree in which each non-leaf node may have either two or three children and all subtrees of a node have the same height.

If we ignore the condition on the heights of subtrees, we can make the following SML type definition.

```
datatype 'a twoThreeTree =
  | Empty
  | Binary of 'a * 'a twoThreeTree * 'a twoThreeTree
  | Ternary of 'a * 'a twoThreeTree * 'a twoThreeTree * 'a twoThreeTree;
```

a. Write a recursive function `N` that computes the number of nodes in a 2-3 tree.

b. Write a recursive function `ht` that computes the height of a 2-3 tree. (In analogy with binary trees, make the height of the empty tree -1.)

c. Prove by tree induction that $2^{\texttt{ht(t)}+1} - 1 \leq \texttt{N(t)} \leq \frac{1}{2}\left(3^{\texttt{ht(t)}+1} - 1\right)$ for all 2-3 trees `t`.

d. What is the shortest 2-3 tree with 47 nodes? the tallest?