Final Exam

15-122 Principles of Imperative Computation
Frank Pfenning

May 3, 2012

Name: Andrew ID: Section:

Instructions

- This exam is closed-book with one double-sided sheet of notes permitted.
- You have 3 hours to complete the exam.
- There are 6 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- Consider writing out programs or diagrams on scratch paper first.

	Integers	Big-O	BDDs	Interfaces	Union-Find	Spanning Trees	
	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	45	35	40	35	60	35	250
Grader							

1. Integers (45 pts)

Task 1 (15 pts). For each of the following expressions in C, indicate whether they are always true. If not, give a counterexample and state whether the counterexample is guaranteed to be false or undefined in C. You should not assume any particular size for ints and state your counterexamples in terms of `INT_MAX`, `INT_MIN`, and `UINT_MAX`. Assume we are in the scope of declarations

```
int x = ...;
unsigned int u = ...;
```

so x and u are properly initialized to arbitrary values. We have filled in one row for you.

C expression	Always true?	If no, counterexample	False or undefined
`x + 1 == 1 + x`	**no**	`x = INT_MAX`	**undefined**
`x <= 0 || -x < 0`			
`x >= 0 || -x > 0`			
`-u - 1 == u ^ -1`			
`u - 1 < u`			
`u - 1 == -(1 - u)`			

Task 2 (10 pts). Write a C function `safe_add` which aborts if the addition would raise an overflow and otherwise returns the sum. You should not assume any particular size for integers, but use `INT_MAX` and `INT_MIN` as needed.

```
int safe_add(int x, int y) {













}
```

Task 3 (10 pts). Write a C function `mod_add` which implements two's complement modular addition on its arguments. As in Task 1, you should not assume any particular size for integers. On the other hand, you may assume values of type `int` and `unsigned int` have the same size, and that values of type `int` are in two's complement representation.

```
int mod_add(int x, int y) {







}
```

Task 4 (10 pts). Write a C function `extend_add` which adds two ints and returns a result of type `long long int`. Since a `long long int` must be large enough to hold the result of adding two ints, the result should always be defined.

```
long long int extend_add(int x, int y) {







}
```

2. Big-O (35 pts)

Task 1 (15 pts). Define the big-O notation

$$O(f(n)) = \{g(n) \mid \rule{8cm}{0.4pt}\}$$

and briefly state the two key ideas behind this definition in two sentences:

Task 2 (10 pts). For each of the following, indicate if the statement is true or false.

1. $O(n^2 + 1024n + 32) = O(31n^2 - 34)$
2. $O(n * \log(n)) \subset O(n)$
3. $O(n) \subset O(n * \log(n))$
4. $O(32) = O(2^{32})$
5. $O(2^n) = O(2^{2^n})$

Task 3 (10 pts). You observe the following timings when executing an implementation of sorting on randomly chosen arrays of size n. Form a conjecture about the asymptotic running time of each implementation.

A

n	time (in secs)
2^{15}	10.23
2^{16}	20.51
2^{17}	41.99
2^{18}	85.27

B

n	time (in secs)
2^{15}	22.36
2^{16}	90.55
2^{17}	368.97
2^{18}	1723.03

C

n	time (in secs)
2^{15}	2.01
2^{16}	5.03
2^{17}	12.77
2^{18}	29.93

$O(\rule{3cm}{0.4pt})$ $O(\rule{3cm}{0.4pt})$ $O(\rule{3cm}{0.4pt})$

Which would be preferable for inputs of about 1 million elements? Circle one:

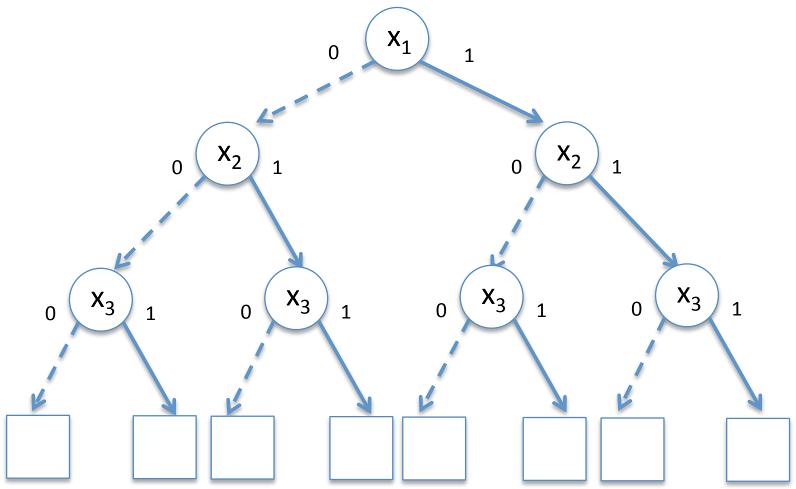
A B C

3. Binary Decision Diagrams (40 pts)

An important operation in the construction of digital circuits is the *nand*-gate, which inputs booleans x_1 and x_2 and outputs $\neg(x_1 \wedge x_2)$. We write $\mathrm{nand}(x_1, x_2)$. In C, this might be defined as

```
bool nand(bool x1, bool x2) {
  return !(x1 && x2);
}
```

Task 1 (15 pts). Construct a ordered binary decision tree (without sharing) for $\mathrm{nand}(x_1, \mathrm{nand}(x_2, x_3))$ where the variables are tested in the order x_1, x_2, x_3.



Task 2 (10 pts). Ordered binary decision diagrams (OBDD) differ from binary decision trees in that they exploit sharing. State the two conditions on an OBDD that are required for a *reduced* ordered binary decision diagram (ROBDD).

1.

2.

Task 3 (15 pts). Convert the binary decision tree from Task 1 into an ROBDD, where the variables are tested in the same order. You only need to show the final ROBDD, but if you show intermediate steps we may more easily assign partial credit.

4. Interfaces (35 pts)

We now consider the C implementation of ROBDDs with hash tables. A node in an OBDD is implemented with the following struct.

```
struct node {
  int var;                /* testing variable x_i */
  struct node* lo;        /* successor for x_i = 0 */
  struct node* hi;        /* successor for x_i = 1 */
  int result;             /* result = 0 or 1 */
  int refcount;           /* reference count */
};
typedef struct node* node;
```

We have the following basic invariants:

1. For an interior node u, we have `u->lo != NULL` and `u->hi != NULL`.
2. For a leaf node u we have `u->lo == NULL` and `u->hi == NULL`, and `u->result == 0` or `u->result == 1`.
3. Since the variables must be tested in order, the low and high successor of an interior node u must have a higher variable number than u.

All interior nodes are created with a function

```
node make_node(int var, node low, node high);
```

In order to make sure that the OBDDs we construct are *reduced*, we use a hash table. If an interior node testing the same variable with the same low and high successors has already been constructed before, we return the one stored in the hash table.

We use the following hash table interface

```
typedef void* ht_key;
typedef void* ht_elem;

typedef struct ht* ht;
ht ht_new (int init_size,
           ht_key (*elem_key)(ht_elem e),
           bool (*key_equal)(ht_key k1, ht_key k2),
           int (*key_hash)(ht_key k, int m));
void ht_insert(ht H, ht_elem e);
ht_elem ht_search(ht H, ht_key k);
void ht_free(ht H, void (*elem_free)(ht_elem e));
```

Task 1 (5 pts). Define an appropriate client-side type `key` in C.

Task 2 (5 pts). Define an appropriate client-side type `elem` in C.

Task 3 (10 pts). Define an appropriate client-side function `elem_key` in C.

```
__________ elem_key(__________ u) {





}
```

Task 4 (10 pts). Define an appropriate client-side function `key_equal` in C.

```
bool key_equal(__________ u, __________ v) {





}
```

Task 5 (5 pts). Assume you have also defined an appropriate function `key_hash`. Show the function call the client needs to construct an appropriate (empty) hash table of initial size 1024.

```
ht_new(1024, ________________ , _________________ , __________________ )
```

5. Union-Find (60 pts)

In this problem we reconsider the *union-find* data structure used to maintain equivalence classes of integers. We say that the representation of an equivalence class has *depth* d if the longest path from an element in the class to the canonical representative is d. For example, any singleton equivalence class has depth 0, and an equivalence class with two elements must have depth 1.

Task 1 (5 pts). Initialize the union-find structure to singleton sets of 8 elements (0–7).

0	1	2	3	4	5	6	7

Task 2 (10 pts). Show the intermediate states of the union-find structure after applying a sequence of seven union operations. Recall that the representative of a union should be the representative of the class with the greater depth. If there is a tie, make the smaller index the new representative. Do not apply any path compression.

call	0	1	2	3	4	5	6	7
`union(0,1)`								
`union(2,3)`								
`union(4,5)`								
`union(6,7)`								
`union(0,3)`								
`union(4,7)`								
`union(0,7)`								

Task 3 (5 pts). If we do not track the depth of the equivalence classes, but always select the representative with the smaller index as the representative of the union, we may generate a class of depth d after d union operations. Give a sequence of union operations which demonstrates this, using the union-find structure from Task 1 with 8 singleton sets as the starting point.

Task 4 (15 pts). The C0 code below checks whether a given union-find structure is valid. As compared to the code in lecture, it also checks that there are no infinite paths. We use the declaration

```
struct ufs {
  int size;
  int[] A;                      /* \length(A) == size */
};
typedef struct ufs* ufs;
```

In this task we ask you to analyze why some array accesses are safe (that is, in bounds) and why some loop invariants hold. Instead of giving an explicit proof, however, we just ask you to indicate the lines in the code you need to conclude that an access is safe or a loop invariant holds.

Your analysis must be precise: only list the lines upon which the safety of an array access or the loop invariant depends. We will deduct points if you have too many or two few lines. We have given you some examples to get you started.

```
00 bool is_ufs (ufs eqs) {
01   if (eqs == NULL) return false;
02   int[] A = eqs->A;                         /* eqs->A safe by line 01 */
03   int size = eqs->size;                     /* eqs->size safe by line 01 */
04   //@assert size == \length(A);
05   int i = 0;
06   while (i < size)
07     //@loop_invariant 0 <= i && i <= size;  /* holds initially by lines 04, 05 */
                                               /* is preserved by lines 06, 07, 19 */
08     { int k = i;
09       int d = 0;
10       while (A[k] != k && d < size)         /* A[k] safe by line(s) ______________ */

11         //@loop_invariant 0 <= k && k < size; /* holds initially by line(s) ____________ */

                                               /* is preserved by line(s) ____________ */
12         {
13           if (!(0 <= A[k] && A[k] < size))  /* A[k] safe by line(s) ______________ */
14             return false;
15           k = A[k];                         /* A[k] safe by line(s) ______________ */
16           d++;
17         }
18       if (d >= size) return false;
19       i++;
20     }
21   return true;
22 }
```

Task 5 (15 pts). Write a *recursive* version of `find` in C0, without path compression. Carefully state pre- and post-conditions. The only function you should use (in addition to the usual arithmetical and logical operators) is `is_ufs` from above.

```
int find(ufs eqs, int i)

//@requires ______________________________________________ ;

//@requires ______________________________________________ ;

//@ensures _______________________________________________ ;

//@ensures _______________________________________________ ;
{






}
```

Task 6 (10 pts). *Path compression* is a refinement of the basic union-find algorithm. Whenever we apply `find` to compute the canonical representative of an element, we short-circuit its path to go directly to the canonical representative. Add path compression to the `find` function from Task 5. You do not need to repeat pre- or post-conditions unless they have changed.

```
int find(ufs eqs, int i)





{











}
```

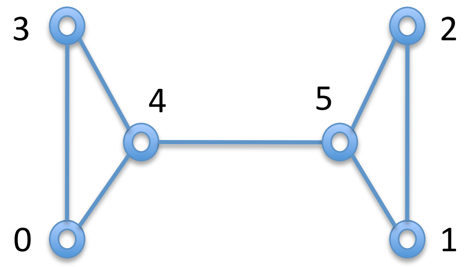
6. Spanning Trees (35 pts)

In this problem we implement an alternative way to compute spanning trees for undirected graphs. This algorithm is *vertex-centric*.

- At the start of each iteration of the algorithm we have two sets of nodes: those that are in the spanning tree already, and those that are not. Among those already in the tree we further distinguish the *frontier*, consisting of the nodes whose neighbors have not been considered.
- One iteration of the algorithm removes a node from the frontier and adds to the spanning tree the edges to those among its neighbors that are not already in the spanning tree. These neighbors are then also added to the frontier.
- We start the algorithm by initializing the frontier to an arbitrary single element.
- The algorithm finishes when the frontier has become empty.

There are different ways we can choose an element from the frontier. Here, we use a stack since it is simple, and we have used the stack interface multiple times in this course.

Task 1 (15 pts). Simulate this algorithm on the given graph, starting at node 0, always considering the neighbors in ascending order. We have filled in the first line for you, displaying the stack with its top on the right.



Node considered	Edges added	New frontier stack
0	**0-3, 0-4**	**3, 4**

For reference, here is the C interface to stacks.

```
typedef struct stack* stack;
bool stack_empty(stack S);      /* O(1) */
stack stack_new();              /* O(1) */
void push(stack S, void* e);    /* O(1) */
void* pop(stack S);             /* O(1) */
void stack_free(stack S, void (*data_free)(void* e));
```

Our C implementation of graphs is abstract with respect to the vertices, which are therefore of type `void*`. For example, vertices could be pointers to structs containing strings (like `"New York"`, `"Chicago"`, etc.). However, vertices must be numbered, so the client must provide the total number n of vertices in the graph as well as a function pointer $vindex$ to extract an index in the range from 0 to $n-1$ from a vertex.

```
struct graph {
  int num_vertices;             /* number n of vertices */
  bool* edges;                  /* n*n array of edges */
  void** vertices;              /* array of n vertices */
  int (*vindex)(void* u);       /* index of vertex, in range [0..n)  */
};
typedef struct graph* graph;
```

We have the invariant `(*vindex)(vertices[i]) == i` for any index $0 \leq i < n$, so the index of the ith vertex is indeed i.

The edge array `edges` holds $n * n$ booleans, where `edges[j*n+i] == true` if there is an edge from vertex j to i. Because we work with undirected graphs, we maintain the invariant that `edges[j*n+i] == edges[i*n+j]`.

We assume there is a function

```
bool is_graph(graph G);
```

which checks the expected invariants explained above on the graph to the extent that this it possible in C. For example, it verifies that G is not null, that function pointer `vindex` is not null, that the edge matrix is symmetric, etc.

We use `xcalloc` which is like `calloc`, initializing the elements to zeroes, but never returns `NULL`, aborting instead if it is out of memory. Recall that in C, 0 is identified with the boolean `false`.

Task 2 (20 pts). Complete the following implementation of `spanning_tree` in C. This function takes two arguments, a graph G representing the original graph, and a graph T representing the spanning tree for G. For simplicity, we assume that the *caller* has already allocated an appropriate graph structure for the spanning tree and passes it as the second argument. We only have to modify its edge array which is initialized with all entries `false`.

```
void spanning_tree (graph G, graph T) {
  REQUIRES(is_graph(G) && is_graph(T));
  REQUIRES(G->num_vertices == T->num_vertices);
  int n = G->num_vertices;
  bool* E = G->edges;
  void** V = G->vertices;

  bool* intree = xcalloc(n, ___________________________); /* initializes to false */

  /* pick starting node and initialize frontier */

  stack S = ______________________________________________________ ;

  __________________________________________________________ ;

  __________________________________________________________ ;

  /* build the spanning tree */
  while (!stack_empty(S)) {

    int j = ______________________________________________ ;
    for (int i = 0; i < n; i++)

      if (_____________________________________________________) {
        T->edges[j*n+i] = true;  /* add edge to spanning tree */
        T->edges[i*n+j] = true;  /* for undirected graph */
        /* update frontier */

        ___________________________________________ ;

        ___________________________________________ ;
      }
  }
  /* free temporary storage */

  ___________________________________ ;

  ___________________________________ ;
  return;
}
```