Final Exam Solutions
15-122 Principles of Imperative Computation
Tom Cortina and Rob Simmons
May 4, 2015

Name: Harry Q. Bovik
Andrew ID: bovik

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 180 minutes to complete the exam.
- There are 6 problems on 23 pages.
- 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.
- You can assume the presence of `#use <util>` for the first problem in C0, and any standard C libraries discussed in class (including `limits.h`, `xalloc.h`, and `contracts.h`) throughout the exam.
1 Union Find [C0] (40 points)
In this question, we consider a C0 implementation of the union find data structure from class. This is the more efficient version that stores tree heights, but without path compression.

typedef struct ufs_header* ufs;
struct ufs_header {
  int size;
  int[] data;
};

bool is_ufs(ufs U) {
  return U != NULL && is_arr_expected_length(U->data, U->size);
}

Task 1 The is_ufs data structure invariant above is not sufficient to ensure the correctness of ufs_find.
/* 1 */ int ufs_find(ufs U, int x)
/* 2 */ //@requires is_ufs(U);
/* 3 */ //@requires 0 <= x && x < U->size;
/* 4 */ {
/* 5 */   int i = x;
/* 6 */   while (U->data[i] >= 0) {
/* 7 */     i = U->data[i];
/* 8 */   }
/* 9 */   return i;
/* 10 */ }

If U contains the address of the data structure above, ufs_find(U, 4) will return

Solution: 0

Show a U such that is_ufs(U) holds but ufs_find(U, 4) will cause a memory error:

Show a U such that is_ufs(U) holds but ufs_find(U, 4) will never terminate:
5 Task 2 In order to ensure that `ufs_find` can never cause a memory error, we need to extend the data structure invariant to check that every integer in the array `U->data` is...

**Solution:** ...less than `U->size` (or, equivalently, less than `\length(U->data)`)

10 Task 3 Give loop invariant(s) for `ufs_find` that would ensure the array accesses on lines 6 and 7 are safe. Using the data structure invariant improvement described in Task 2, you should be able to reason that this loop invariant is true initially and preserved by an arbitrary iteration of the loop. You don’t necessarily have to use all the lines.

```
//@loop_invariant 0 <= i && i < U->size;
```

10 Task 4 The `union` operation of union-find uses `ufs_find`:

```c
/* 1 */ void ufs_union(ufs U, int x, int y)
/* 2 */ //@requires is_ufs(U);
/* 3 */ //@ensures is_ufs(U);
/* 4 */ {
/* 5 */ int[] A = U->data;
/* 6 */ int i = ufs_find(U, x);
/* 7 */ int j = ufs_find(U, y);
/* 8 */ //@assert A[i] < 0 && A[j] < 0;
/* 9 */
/* 10 */ if (i == j) return;
/* 12 */ else if (A[i] < A[j]) { A[j] = i; }
/* 13 */ else { A[i] = j; }
/* 14 */ }
```

What postconditions does `ufs_find` need in order for us to reason that the assertion on line 8 always returns true – even if we know nothing about the implementation of `ufs_find`? You don’t necessarily have to use all lines.

```
//@ensures 0 <= \result && \result < U->size && U->data[\result] < 0;
```

5 Task 5 In this implementation, if `U->data[0] == -9`, then that means `U->size` must be at least...

**Solution:** $256 = 2^{(9-1)}$
2 Grab Bag [C] (50 points)

Task 1 The following run times were obtained when using two different algorithms on a data set of size $n$. You are asked to determine asymptotic complexity of the algorithms based on this time data. Determine the asymptotic complexity of each algorithm as a function of $n$. Use big-O notation in its tightest, simplest form.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.564 milliseconds</td>
</tr>
<tr>
<td>2000</td>
<td>2.271 milliseconds</td>
</tr>
<tr>
<td>4000</td>
<td>8.992 milliseconds</td>
</tr>
<tr>
<td>8000</td>
<td>36.150 milliseconds</td>
</tr>
</tbody>
</table>

$O(n^2)$

<table>
<thead>
<tr>
<th>$n$</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.042 milliseconds</td>
</tr>
<tr>
<td>1000000</td>
<td>0.042 milliseconds</td>
</tr>
<tr>
<td>1000000000</td>
<td>0.042 milliseconds</td>
</tr>
</tbody>
</table>

$O(1)$

Task 2 Represent the min-heap pictured above as an array by the method discussed in class:

If the number 5 is added to this heap structure, what will the array look like afterwards?

If, after adding the number 5, we then remove the minimum element from the heap, what will the final contents of the array be?
Consider the following implementation of a bounded queue of integers in C:

```c
typedef struct queue_header queue;
struct queue_header {
    int capacity; // maximum size of queue (overflow not allowed)
    int front;   // index of front element of queue
    int rear;    // index of rear element of queue
    int *data;   // queue data, length of the array is capacity
};
```

An empty queue is always represented by front = -1 and rear = -1. Otherwise, front is the index of the first element of the queue and rear is the index of the last element of the queue. If an element is enqueued on to an empty queue, it is always stored at index 0 of the array. Elements of the queue are stored in contiguous cells of the array. Note that front index can be greater than rear index. In this case, the queue wraps around from the end of the array back to the beginning, as shown in the example below.

```
Q = queue_new(5)  front = -1, rear = -1, capacity = 5
enq(Q, 4);       front = 0,  rear = 0,  capacity = 5
enq(Q, 7);       front = 0,  rear = 1,  capacity = 5
enq(Q, 8);       front = 0,  rear = 2,  capacity = 5
deq(Q);          front = 1,  rear = 2,  capacity = 5
deq(Q);          front = 2,  rear = 2,  capacity = 5
enq(Q, 9);       front = 2,  rear = 3,  capacity = 5
enq(Q, 6);       front = 2,  rear = 4,  capacity = 5
enq(Q, 3);       front = 2,  rear = 0,  capacity = 5
deq(Q);          front = 3,  rear = 0,  capacity = 5
deq(Q);          front = 4,  rear = 0,  capacity = 5
enq(Q, 10);      front = 4,  rear = 1,  capacity = 5
deq(Q);          front = 0,  rear = 1,  capacity = 5
deq(Q);          front = 1,  rear = 1,  capacity = 5
enq(Q, 42);      front = 1,  rear = 2,  capacity = 5
deq(Q);          front = 2,  rear = 2,  capacity = 5
deq(Q);          front = -1, rear = -1, capacity = 5
enq(Q, 2);       front = 0,  rear = 0,  capacity = 5
```
Assume there is a function is_queue that tests the data structure invariant for a queue* as described and illustrated above. Complete the following functions to implement the queue described above. All functions should run in constant time.

```c
queue *queue_new(int n) {
    REQUIRES(n > 0);

    queue *Q = xmalloc(sizeof(struct queue_header));

    Q->data = xalloc(n, sizeof(int));
    Q->front = -1;
    Q->rear = -1;
    Q->capacity = n;
    ENSURES(is_queue(Q) && queue_empty(Q));
    return Q;
}

void queue_free(queue *Q) {
    REQUIRES(is_queue(Q));

    free(Q->data);
    free(Q);
}

bool queue_empty(queue *Q) {
    REQUIRES(is_queue(Q));
    return Q->front == -1 && Q->rear == -1;
}

bool queue_full(queue *Q) {
    REQUIRES(is_queue(Q));

    return (Q->rear + 1) % Q->capacity == Q->front;
}

void enq(queue *Q, int element) {
    REQUIRES(is_queue(Q) && !queue_full(Q));

    Q->rear = (Q->rear + 1) % Q->capacity;
    Q->data[Q->rear] = element;
    if (Q->front == -1) Q->front = 0;
    ENSURES(is_queue(Q));
}
```

CONTINUED ON NEXT PAGE...
Task 4 continued...

```c
int deq(queue *Q) {
    REQUIRES(is_queue(Q) && !queue_empty(Q));

    int element = Q->data[Q->front];

    if (Q->front == Q->rear) {
        Q->front = -1;
        Q->rear = -1;
    } else {
        Q->front = (Q->front + 1) % Q->capacity;
    }

    ENSURES(is_queue(Q));
    return element;
}
```

8 Task 4 The ternary search trie (TST) pictured below stores a set of numerical strings. List all numerical strings stored in the trie.

![Ternary Search Trie Diagram]

Solution: 1, 15, 18, 20, 36, 4, 755, 791
Consider the following C function that computes the integer square root of \( n, n > 0 \). The integer square root \( i \) of a positive integer \( n \) is the value of \( i \) such that \( i^2 \leq n \) and \((i + 1)^2 > n\). For example, \( \text{isqrt}(15) = 3 \) and \( \text{isqrt}(16) = 4 \).

```c
/* 1 */ int isqrt(int n) {
    /* 2 */ REQUIRES(n < 46000);
    /* 3 */
    /* 4 */ if (n == 1) return 1;
    /* 5 */ int lower = 0;
    /* 6 */ int upper = 1 + n/2;
    /* 7 */
    /* 8 */ while(lower + 1 < upper) {
        /* 9 */ printf(" lower = %d, upper = %d\n", lower, upper);
        /* 10 */ int mid = lower + (upper - lower)/2;
        /* 11 */ int square = mid * mid;
        /* 12 */ if (square == n) {
            /* 13 */ lower = mid;
            /* 14 */ break; // break out of loop, go straight to line 21
        /* 15 */ } else if (square < n) {
            /* 16 */ lower = mid;
        /* 17 */ } else {
            /* 18 */ upper = mid;
        /* 19 */ }
        /* 20 */ }
    /* 21 */ printf(" lower = %d, upper = %d\n", lower, upper);
    /* 22 */ return lower;
    /* 23 */ }
```

Trace the function for the following values of \( n \), showing the values printed for \( \text{lower} \) and \( \text{upper} \) at the start of each iteration, along with their values after the loop terminates. You may not need to use all the available space provided. The first values output for \( \text{lower} \) and \( \text{upper} \) are given for you.

<table>
<thead>
<tr>
<th>( n = 25 )</th>
<th>( \text{lower} )</th>
<th>( \text{upper} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n = 42 )</th>
<th>( \text{lower} )</th>
<th>( \text{upper} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>11</td>
</tr>
</tbody>
</table>

What is the worst case runtime complexity of the function above in terms of \( n \) using big O notation in its simplest, tightest form?

\[ O(\log n) \]
3 Strings [C] (40 points)

20 Task 1 String buffers are useful as a data structure because concatenating strings naively can get really expensive. Concatenating a string of length $x$ and a string of length $y$ with the library function `strcat` requires $x + y$ constant-time operations.

(In your answers below, both $n$ and $k$ can vary: don’t treat $k$ as a constant.)

If we are concatenating $n$ strings with C’s `strcat`, where each string has length $k$, by joining them one at a time (so we have $n$ strings of length $k$, then $n - 2$ strings of length $k$ and 1 string of length $2k$, and then we have $n - 3$ strings of length $k$ and one string of length $3k$...), the total running time of the process taking $n$ length-$k$ strings and returning a single string of length $nk$ will be in

$$O(kn^2)$$

If we instead only concatenate pairs of strings with equal length (so we have $n$ strings of length $k$, then $n/2$ strings of length $2k$, and then $n/4$ strings of length $4k$), the total running time of the process taking $n$ length-$k$ strings and returning a single string of length $nk$ will be in

$$O(kn \log n)$$

If we use a single correctly-implemented string buffer to add the $n$ strings of length $k$ to the buffer one at a time, and then we use `strbuf_str` to make a copy of the final string with size $nk$, the running time of this whole process will be in

$$O(kn)$$

During the process described immediately above, the worst-case running time of a single call to the function `strbuf_addstr` could be in

$$O(kn)$$

Despite this, our amortized analysis of the problem ensures most calls to `strbuf_addstr` will have a running time that is in

$$O(k)$$
Task 2

For these questions, imagine that we have placed $n$ Andrew IDs, represented as strings of 2-8 characters, into a separate-chaining hashtable with a capacity (table size) of $m$, where $n$ and $m$ are both very large. (Don’t assume anything about their relationship, though: $n$ could be much larger than $m$ or vice-versa.)

If our hash function takes $s$ and returns $\text{strlen}(s) \times 1664525 + 1013904223$, where $\text{strlen}$ gives us the length of the Andrew ID we expect that a single lookup or insertion to take time in $O(n)$.

If our hash function always returns 4, we expect that a single lookup or insertion to take time in $O(n)$.

If we know the $n$ Andrew IDs in advance, then the best possible hash function for those Andrew IDs would ensure that a single lookup or insertion takes time in $O(n/m)$.

Is the pseudorandom number generator hash_lcg discussed in lab and lecture, which applies a linear congruential generator to every character in the string, always going to ensure this best possible performance? Briefly justify your answer (a sentence or two at most).

(Hint: The specifics of hash_lcg aren’t important here, it’s just an example of a good hash function.)

Solution: No: it’s possible to create collisions with any hash function.

Task 3

If a hashtable using open addressing (specifically linear probing) contains $k$ elements and has a table with size $k$, then looking up any key that is not in the table will take time in $O(k)$. 
4 Clac Revisited [C] (40 points)

One of the jobs of a parser is to take an infix expression like $7 + 7 \times 1 < 9$ and figure out that it is supposed to be understood as $(7 + (7 \times 1)) < 9$. This unambiguous representation of the structure of an expression can be represented as a tree structure:

Clac gave us another way to unambiguously represent expressions without using parentheses. We can represent the expression given as a tree above in Clac by writing $7 \ 7 \ 1 \ * \ + \ 9 \ <$.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Queue</td>
</tr>
<tr>
<td>$S$</td>
<td>$n, Q$</td>
</tr>
<tr>
<td>$S, x, y$</td>
<td>$+, Q$</td>
</tr>
<tr>
<td>$S, x, y$</td>
<td>$-, Q$</td>
</tr>
<tr>
<td>$S, x, y$</td>
<td>$\ast, Q$</td>
</tr>
<tr>
<td>$S, x, y$</td>
<td>$&lt;, Q$</td>
</tr>
<tr>
<td>$S, x, y$</td>
<td>$&lt;, Q$</td>
</tr>
</tbody>
</table>

Task 1 Draw the tree corresponding to the C0 expression $(17 - 8 \times 2 + 3)$, which evaluates to 4.

Solution:

Task 2 Draw the tree corresponding to the Clac expression $2 - 2 \ 17 \ 1 - \ast \ <$.

Solution:
Task 3  Expression trees can be written in C as elements of the type exp*. The specification function is_binop checks that a char* is a well-formed C string representing a binary operator: either "<", "*", "+", or "-". The specification function is_int checks that a char* is a well-formed C string that can be parsed as a 32-bit signed integer.

```c
typedef struct exp_node exp;
struct exp_node {
    char *data;
    exp *left;
    exp *right;
};
bool is_exp(exp *E) {
    if (E == NULL) return false;
    if (E->left == NULL && E->right == NULL && is_int(E->data)) return true;
    return is_exp(E->left) && is_exp(E->right) && is_binop(E->data);
}
```

Clac programs are represented by queues of strings: we represent the Clac program 2 -2 17 1 * < as a queue with "2" is at the front of the queue and "<" at the back of the queue. Strings are pointers to char in C, so generic queues can easily hold strings.

```c
bool queue_empty(queue_t Q);
void enq(queue_t Q, void *x);
void *deq(queue_t Q); // Requires !queue_empty(Q);
```

Write a (simple and efficient!) recursive procedure for converting a exp* expression into a Clac program and adding that program to the end of a queue. Don’t modify the existing tree E, don’t call the specification functions is_int and is_binop, and don’t explicitly allocate or free memory (calling enq and deq is fine, though). Explicitly write a cast whenever you convert pointers.

```c
void convert(exp *E, queue_t Q) {
    REQUIRE(is_exp(E));

    if (E->left != NULL) {
        convert(E->left, Q);
        convert(E->right, Q);
        enq(Q, (void*)E->data);
    } else {
        enq(Q, (void*)E->data);
    }
}
```
Clac and the C0VM behave in essentially the same way, except that one is based on queues and the other is based on a program counter that can move around more freely. Here are some bytecode instructions you’ll use in this question (you don’t need to use them all):

0x60 iadd \( S, x:w_{32}, y:w_{32} \rightarrow S, x+y:w_{32} \)
0x7E iand \( S, x:w_{32}, y:w_{32} \rightarrow S, x\&y:w_{32} \)
0x6C idiv \( S, x:w_{32}, y:w_{32} \rightarrow S, x/y:w_{32} \)
0x68 imul \( S, x:w_{32}, y:w_{32} \rightarrow S, x*y:w_{32} \)
0x64 isub \( S, x:w_{32}, y:w_{32} \rightarrow S, x-y:w_{32} \)
0x10 bipush <b> \( S \rightarrow S, x:w_{32} (x = (w_{32})b, \text{sign extended}) \)
0x00 nop \( S \rightarrow S \)
0xA1 if_icmplt <o1,o2> \( S, x:w_{32}, y:w_{32} \rightarrow S \quad (pc = pc+(o1<<8\mid o2) \text{ if } x < y) \)
0xA7 goto <o1,o2> \( S \rightarrow S \quad (pc = pc+(o1<<8\mid o2)) \)
0xB0 return ., v \( \rightarrow . \quad (\text{return } v \text{ to caller}) \)

Write C0VM bytecode that behaves, as much as possible, the same way as the Clac code 2 -2 17 1 - * <. Your code should return either 0 or 1.

You only have to write the direct bytecode (“15 02”), but you can also write the mnemonic forms (“vload 2”). You don’t have to use every line.

```
# C0 bytecode that mimics the Clac code 2 -2 17 1 - * <
10 02  # bipush 2
10 FE  # bipush -2
10 11  # bipush 17
10 01  # bipush 1
64    # isub
68    # imul
A1 00 08 # if_icmplt 8
10 00  # bipush 0
A7 00 05 # goto 5
10 01  # bipush 1
B0    # return
```
5 Spanning Trees [C] (40 points)
The C interface for a weighted undirected graph with positive integer weights is given below:

typedef unsigned int vertex;

void graph_free(graph G);
unsigned int graph_size(graph G); // Number of vertices in the graph
graph graph_new(unsigned int n);
  // New graph with n vertices and no edges, requires n > 0
bool graph_hasedge(graph G, vertex v, vertex w);
  // Requires v < graph_size(G) && w < graph_size(G);
void graph_addedge(graph G, vertex v, vertex w, int weight);
  // Requires v < graph_size(G) && w < graph_size(G);
  // Requires !graph_hasedge(G, v, w);
  // Requires weight > 0;
int graph_getweight(graph G, vertex v, vertex w);
  // Requires v < graph_size(G) && w < graph_size(G);
  // Requires graph_hasedge(G, v, w);

Task 1
10 Assume an adjacency list implementation of graphs, as indicated below.

typedef struct graph_header *graph;
typedef struct adjlist_node adjlist;
struct adjlist_node {
  vertex vert;
  int weight; // weight on edge to vertex vert
  adjlist *next;
};
struct graph_header {
  unsigned int size;
  adjlist **adj; // An array of adjacency lists
};

Write the graph function graph_getweight, assuming the existence of is_graph.

int graph_getweight(graph G, vertex v, vertex w) {
  REQUIRES(is_graph(G) && v < graph_size(G) && w < graph_size(G));
  REQUIRES(graph_hasedge(G, v, w));
  adjlist* p = G->adj[v];
  while (p != NULL) {
    if (p->vert == w) return p->weight;
    p = p->next;
  }
  return -1;
}
**Prim’s algorithm** is another way to compute a minimum spanning tree for a graph. In this implementation of Prim’s algorithm, we need an array of booleans to mark when each vertex is added to the minimum spanning tree. We also need a priority queue to hold edges under consideration, ordered based on weight: the edge with the highest priority in the priority queue is the edge with minimum weight.

Assume the following C interface for priority queues.

```c
typedef void *elem;
typedef bool higher_priority_fn(elem e1, elem e2);
// (*prior)(e1,e2) returns true if e1 is *strictly* higher priority than e2
pq_t pq_new(size_t capacity, /* > 0 */
    higher_priority_fn *prior); /* != NULL */
bool pq_empty(pq_t P);
void pq_add(pq_t P, elem e);
elem pq_rem(pq_t P); /* Must not be empty */
void pq_free(pq_t P); /* Must be empty */
```

In the next two tasks, you will complete a client program to implement Prim’s algorithm. Note that for these tasks, you should respect the interfaces for graphs and priority queues. From the client’s perspective, you do not know how these data structures are implemented. Start with the following client code in prim.c:

```c
#include <stdlib.h>
#include <stdbool.h>
#include "lib/graph.h"
#include "lib/pq.h"
#include "lib/xalloc.h"
#include "lib/contracts.h"

struct edge_header { // edge from v to w
    vertex v;
    vertex w;
    int weight;
};
typedef struct edge_header* edge;
```

**Task 2** First, write a client function `edgepriority` of type `higher_priority_fn` that can be used with this priority queue. This function will be stored in `prim.c` also.

```c
bool edgepriority(elem e1, elem e2) {
    REQUIRE(e1 != NULL && e2 != NULL);
    return (((edge)e1)->weight < ((edge)e2)->weight;
}
```
Task 3 We will now write a function `prim` (in the same file `prim.c`) that will compute and return a new graph representing the minimum spanning tree of a graph $G$ with at most 1000 vertices. Complete the missing parts.

```c
graph prim(graph G) {
    REQUIRES(graph_size(G) <= 1000);
    // Requires that the graph G is connected

    unsigned int n = graph_size(G);

    // Create a new priority queue whose capacity should be large enough
    // to hold every edge of the graph.
    pq_t PQ = pq_new(n*n, &edgepriority);

    // Create the array of bool to mark each vertex that is
    // added to the minimum spanning tree, all set to false.
    bool *mark = xcalloc(n, sizeof(bool));

    // Create a new graph for the minimum spanning tree
    // with the same number of vertices as G but no edges.
    graph T = graph_new(n);

    // Start with vertex 0 as the first vertex added to tree T
    vertex v = 0;
    mark[v] = true;
    unsigned int numv = 1; // number of vertices in spanning tree

    // While the spanning tree is not complete...
    while (numv != n) {

        // For each neighbor w of v, if w is not in the spanning tree,
        // add the edge to the neighbor to the priority queue.
        for (vertex w = 0; w < n; w++) {
            if (graph_hasedge(G,v,w) && !mark[w]) {
                edge e = xmalloc(sizeof(struct edge_header));
                e->v = v;
                e->w = w;
                e->weight = graph_getweight(G,v,w);
                pq_add(PQ, e);
            }
        }
    }
}
```

THE BODY OF THE WHILE LOOP IS CONTINUED ON NEXT PAGE...
Task 3 continued...

```c
    // Retrieve the minimum weight edge from priority queue
    // until you find one that leads to an unmarked vertex.
    edge minedge = pq_rem(PQ);
    while (mark[minedge->w]) {
        free(minedge);
        minedge = pq_rem(PQ);
    }

    // Add this edge to the minimum spanning tree
    graph_addedge(T, minedge->v, minedge->w, minedge->weight);
    numv++;

    // Now consider edges from vertex w in the next iteration.
    v = minedge->w;
    mark[v] = true;
    free(minedge);
}

    // Free up remaining dynamically-allocated memory.
    free(mark);
    while (!pq_empty(PQ)) {
        free(pq_rem(PQ));
    }
    pq_free(PQ);

    // Return answer.
    return T;
```

Task 4

If our graph has $v$ vertices and $e$ edges, what is the worst case runtime complexity of any single call to `pq_rem` in the algorithm above if the priority queue is implemented using a heap? Express your answer using big O notation in its simplest, tightest form as a function of $v$ and/or $e$.

$O(\log e)$
6 I can C clearly now [C] (40 points)
For all the parts of this question, you can assume standard implementation-defined behavior: 8-bit bytes, 2-byte shorts, and so on. In every case, assume we are calling gcc with the command “gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic example.c”

Task 1 For each of the five following code examples, complete the given assignments in such a way that undefined behavior is triggered on the specified line (and not before). Only give values that are in range of the specified type.
If there is no way to trigger undefined behavior, check the box instead.

POSSIBLE SOLUTIONS: i = 252, 253, 254, or 255
unsigned char i = ____________________________________________________;
char *S = xcalloc(63, sizeof(int));

// Cause undefined behavior on the next line:
char c = S[i] + 5;
Check this box if there is no way to trigger undefined behavior:  

unsigned int x = _____________________________________________________;
signed short y = _____________________________________________________;
unsigned int z = (unsigned int)(signed int)y;

// Cause undefined behavior on the next line:
if (x + z < x) y = 0xbad;
Check this box if there is no way to trigger undefined behavior: X

POSSIBLE SOLUTIONS: x = 0, causes the array index to overflows to SIZE_T_MAX
int x = __________;
assert(0 <= x);
int *A = xcalloc(100, sizeof(int));

// Cause undefined behavior in this for loop
for (size_t i = (size_t)(unsigned int)x; i < 100; i++)
Check this box if there is no way to trigger undefined behavior:  

POSSIBLE SOLUTIONS: Anything where i+j overflows, where k is in (0, 100000)

```c
int i = ______________________________________________________________;
int j = ______________________________________________________________;
int k = ______________________________________________________________;

assert(0 < k && k < 100000);
void **A = xmalloc(sizeof(void*) * k);

// Cause undefined behavior on the next line:
if (0 <= i + j && i + j < k) A[i + j] = NULL;

Check this box if there is no way to trigger undefined behavior: [ ]
```

```c
size_t x = ______________________________________________________________;

assert (x <= strlen("Hello"));
char *str = "Hello";

// Cause undefined behavior in this for loop
for (size_t i = 0; i < x; i++) {
    str = str + 1;
}
printf("The string is \"\%s\"\n", str)

Check this box if there is no way to trigger undefined behavior: [X]
```

**Task 2** Which of the following code snippets would cause undefined behavior? Circle Error if there is undefined behavior, otherwise circle OK.

<table>
<thead>
<tr>
<th>Code Snippet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void* x = xmalloc(sizeof(int)); int i = *(int*)x;</code></td>
<td>Uninitialized read</td>
</tr>
<tr>
<td><code>void* x = xcalloc(1, sizeof(int)); int i = *(int*)x;</code></td>
<td>4 byte allocation</td>
</tr>
<tr>
<td><code>void* x = xcalloc(1, sizeof(short)); int i = *(int*)x;</code></td>
<td>2 byte allocation</td>
</tr>
<tr>
<td><code>void* x = xcalloc(2, 3); int i = *(int*)x;</code></td>
<td>6 byte allocation</td>
</tr>
<tr>
<td><code>void* x = xcalloc(sizeof(void*), 1); int i = *(int*)x;</code></td>
<td>8(4?) byte allocation</td>
</tr>
</tbody>
</table>
3 Task 3 For each example, either say what \( y \) is at the end of the code snippet as an 8-hex-digit constant or write \texttt{undefined} if any undefined behavior occurs.

\begin{enumerate}
\item \begin{verbatim}
short w = -1;
unsigned short x = (unsigned short)w;
x = x << 4;
unsigned int y = (unsigned int)x;
\end{verbatim}
\( y \) is \texttt{0x0000FFF0}
\item \begin{verbatim}
signed char x = -2;
unsigned int y = (unsigned int)(int)x;
y = y << 8;
\end{verbatim}
\( y \) is \texttt{0xFFFFFFFF}
\item \begin{verbatim}
unsigned char x = 3;
int y = (int)(signed char)x;
y = y << 32;
\end{verbatim}
\( y \) is \texttt{undefined}
\item \begin{verbatim}
unsigned char x = 255;
x = x >> 1;
int y = (int)(signed char)x;
\end{verbatim}
\( y \) is \texttt{0x0000007F}
\end{enumerate}

3 Task 4 The following function is intended to do a C0VM-like operation: treating an array of 3 bytes like a signed integer. Given the bytes \{80, 00, 00\}, for instance, this function returns the signed quantity \texttt{0xFF800000} (or -8388608), as it should. Reveal the bug in this function by giving a test case, the desired outcome, and the actual result.

\textbf{POSSIBLE SOLUTIONS:} Anything that causes sign extension in a low-order byte and has something besides FF in a higher-order byte. Example: 00 FF FF

\begin{verbatim}
// Treats three bytes as a signed integer
int32_t g(uint8_t *P) {
    int32_t x = (int32_t)(int8_t)P[0];
    int32_t y = (int32_t)(int8_t)P[1];
    int32_t z = (int32_t)(int8_t)P[2];
    int32_t r = (x << 16) | (y << 8) | z;
    return r;
}
\end{verbatim}
(This page intentionally left blank.)