

Final Exam

15-122 Principles of Imperative Computation

Friday 15th December, 2017

Name: _____

Andrew ID: _____

Recitation Section: _____

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 180 minutes to complete the exam.
- There are 7 problems on 28 pages (including 2 blank pages and 2 reference sheets at the end).
- Use a **dark** pen or pencil to write your answers.
- 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.

	Max	Score
Contracts [$C_0 \mapsto C$]	30	
Safety [C]	20	
Complexity	20	
Mergeable Heaps [C]	30	
Virtual Machines [C]	60	
Graph Representation [C]	50	
Shortest Path [C]	40	
Total:	250	

1 Contracts [$C_0 \mapsto C$] (30 points)

The following C0 code checks the validity of a union-find data structure implemented as an array. Recall that each element in the array maps a vertex in a graph to its canonical representative. It does not do height tracking. *You do not need to understand the details of this code.*

Recall that the array utilities `le_seg(x,A,lo,hi)` and `gt_seg(x,A,lo,hi)` implement the tests $x \leq A[lo, hi]$ and $x > A[lo, hi]$, respectively.

```

1 bool no_loops(int[] UFS, int n, int v)
2 //@requires n == \length(UFS);
3 //@requires 0 <= v && v < n;
4 //@requires le_seg(0, UFS, 0, v);
5 //@requires gt_seg(n, UFS, 0, v);
6 {
7   int w = v; // Starting point
8   int k = 0; // Counter
9   while (UFS[w] != w && k < n)
10  //@loop_invariant 0 <= k && k <= n;
11  {
12    w = UFS[w];
13    k++;
14  }
15  return k < n;
16 }
17
18 bool is_ufs(int[] UFS, int n) {
19   if (!is_array_expected_length(UFS, n))
20     return false;
21
22   for (int v = 0; v < n; v++)
23     //@loop_invariant 0 <= v && v <= n;
24     //@loop_invariant le_seg(0, UFS, 0, v);
25     //@loop_invariant gt_seg(n, UFS, 0, v);
26     {
27       if (!(0 <= UFS[v] && UFS[v] < n))
28         return false;
29
30       if (!no_loops(UFS, n, v))
31         return false;
32     }
33   return true;
34 }

```

5pts

Task 1 Prove that the loop invariant on line 25 is true initially (INIT).

To show: _____

Because

10pts

Task 2 Prove that the loop invariant on line 25 is preserved by each iteration of the loop (PRES).
You may not need all lines.

Assume: _____

To show: _____

(a) _____ by _____

(b) _____ by _____

(c) _____ by _____

(d) _____ by _____

(e) _____ by _____

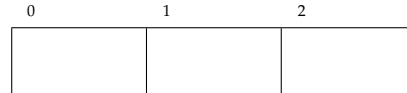
5pts

Task 3 Is the array access on line 9 safe? If you think it is, provide a proof. If you think it is not, give a 3-element array UFS and a values for v so that the call `no_loops(UFS, 3, v)` causes a memory violation.

 Safe,

because _____

 Unsafe

 Counterexample: $v ==$ _____, $UFS ==$


10pts

Task 4 Translate the function `no_loops` into C. You may assume that the array utility functions have been correctly ported to C for you. If a C0 contract cannot be expressed in C, briefly explain why in a comment. Note in particular that you will need to translate the loop invariant on line 10 into appropriately placed ASSERT statements.

2 Safety [C] (20 points)

For the C functions in this question, write (additional) preconditions that are sufficient to ensure that there will be **no undefined behavior and no memory leaks**. Your preconditions must be as permissive as possible: they should allow the function to run *whenever* undefined behavior and memory leaks would not occur. Make sure it's not possible to cause undefined behavior in the precondition itself!

If it's not possible to ensure that a function is free of undefined behavior and memory leaks, then write the precondition `false`, which indicates that the function cannot be run safely. If no preconditions are needed, write the precondition `true`.

Do not assume *any* implementation-defined behaviors except that a byte is 8 bits. Your preconditions should make your functions safe for any implementation-defined behavior.

4pts Task 1

```
unsigned short a1(unsigned short x, int z) {  
  
    REQUIRES(_____);  
  
    return x << z;  
}
```

4pts Task 2

```
void a2(long **A, size_t n) {  
  
    REQUIRES(_____);  
  
    for (size_t i = 0; i < n; i++)  
        if (A[i] != NULL) free(A[i]);  
    free(A);  
}
```

4pts Task 3

```
int a3(unsigned short n) {  
  
    REQUIRES(_____);  
  
    if (n == 0) return -1;  
  
    int x = 0;  
    int *B = xcalloc(n, sizeof(int));  
    for (size_t i = 0; i < n; i++)  
        x += B[i];  
    free(B);  
    return x;  
}
```

4pts **Task 4** For this task, assume that A is a pointer to a heap-allocated array of size n.

```
void a4(int *A, size_t n) {  
  
    REQUIRES(_____);  
  
    for (size_t i = 0; i < n; i++)  
        free(&A[i]);  
}
```

4pts **Task 5** For this task, assume that casting between signed and unsigned types of the same size works as seen in class.

```
int a5(signed char n, int m) {  
  
    REQUIRES(_____);  
  
    unsigned int x = (unsigned int)(unsigned char)n;  
    unsigned int y = (unsigned int)(signed int)n;  
  
    if (x != y) return INT_MIN + m;  
    return m;  
}
```

3 Complexity (20 points)

Give **worst-case** running time bounds for the following operations. Some of the descriptions state that multiple operations are happening: give the cost of doing the *entire sequence* of operations, not the cost of doing a single operation within the sequence.

If the worst case depends on amortized behavior, also write the word **AMORTIZED**.

Assume comparison functions and hash functions are all $O(1)$.

Always give the *simplest, tightest* big- O bound.

	Worst-case
Adding n elements to an initially empty stack (implemented as a linked list).	
Adding n elements to an unbounded array, using mergesort to re-sort the array after every single addition .	
Adding n elements to an unbounded array, using quicksort to re-sort the array once, after all n additions .	
Inserting an element into a min-heap that already has n elements in it and then removing the minimum from the min-heap.	
Adding n elements to an initially empty binary search tree.	
Adding n elements to an initially empty AVL tree.	
Inserting n elements to an initially empty non-resizing, separate-chaining hash table with table size m .	
Computing the number of chains that are empty in a separate-chaining hash table with n elements and table size m .	
Running Kruskal's algorithm on a graph with v vertices and e edges using depth-first search to verify if two vertices are already connected.	
Running Kruskal's algorithm on a graph with v vertices and e edges using union-find with height tracking to minimize tree height.	

4 Mergeable Heaps [C] (30 points)

This question explores *mergeable heaps*. Rather than growing a heap by adding an element to it, mergeable heaps are grown by merging two existing heaps. Viewed as trees, mergeable heaps maintain the same *order invariant* as the min-heaps discussed in class, but may impose *different* invariants on the heap's *shape*.

Task 1 We start out with a *naive* version of a mergeable heap, which only maintains the order invariant but does not constrain the heap's shape. Below is a summary of the relevant type definitions and C0-style prototypes. For simplicity, we record the priority as the field value rather than as we saw in class.

```
typedef struct heap_header heap;
struct heap_header {
    int value; // Node priority
    heap *left;
    heap *right;
};

bool empty(heap *H);
//@requires is_mergeable_heap(H);

heap* merge(heap *L, heap *R)
//@requires is_mergeable_heap(L);
//@requires is_mergeable_heap(R);
//@ensures is_mergeable_heap(\result);

heap* delete_min(heap *H);
//@requires is_mergeable_heap(H);
//@requires !empty(H);
//@ensures is_mergeable_heap(\result);
```

4pts

- a. Complete the specification function `is_minheap` in the box below, defining the *order invariant*, which is the same as the one for the min-heaps discussed in class. The specification function is called from within the representation invariant `is_mergeable_heap`:

```
bool is_mergeable_heap(heap *H) { // Don't worry about circularity
    if (H == NULL) return true;
    return is_minheap(H, H->value);
}
```

Hint: you will not need additional NULL-checks, nor additional space.

```
bool is_minheap(heap *H, int lower_bound) {
    if (H == NULL) return true;

    }
}
```

To maintain the order invariant, the function `merge(L, R)` merges heaps `L` and `R` by considering the priority p_L of the root r_L of `L` and the priority p_R of the root r_R of `R`:

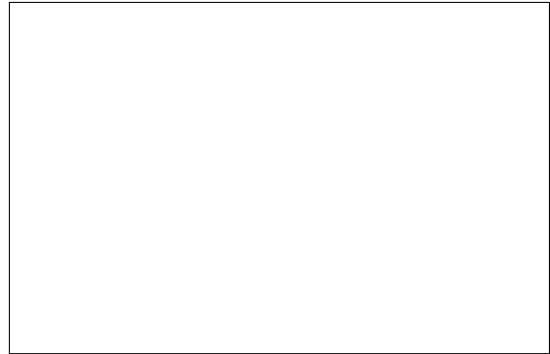
- if $p_L \leq p_R$, it chooses r_L to be the root of the resulting tree and merges the right subtree of `L` with `R`.
- if $p_L > p_R$, it chooses r_R to be the root of the resulting tree and merges the right subtree of `R` with `L`.

In summary, `merge` chooses the heap with the smaller root and merges its right subtree with the other heap. Note that `merge` leaves the left subtree of the chosen heap intact.

For each of the examples below, draw the heap resulting from calling the function merge on the two given heaps L and R in the box provided. Each node contains its priority.

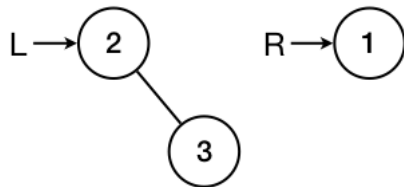
2pts

b.



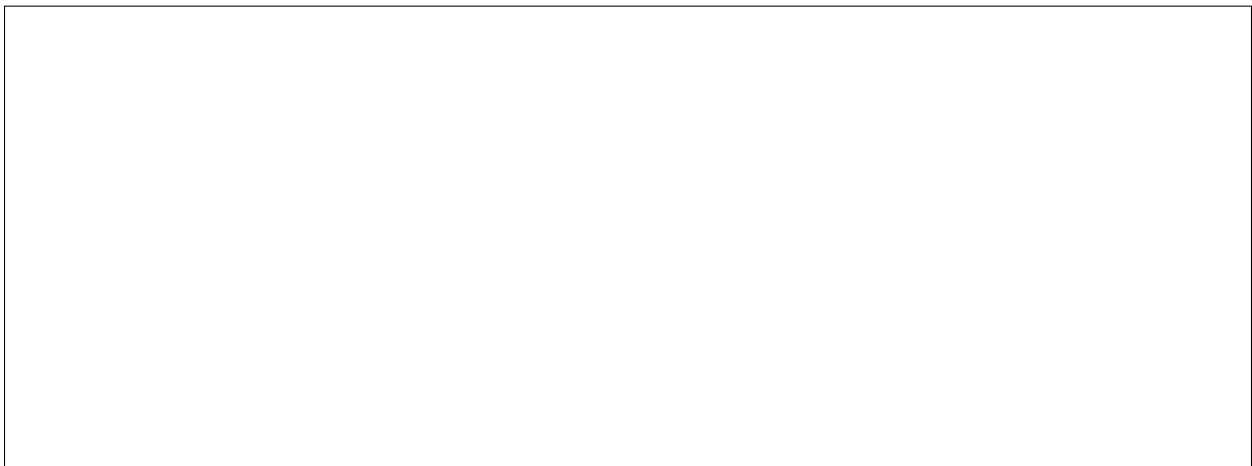
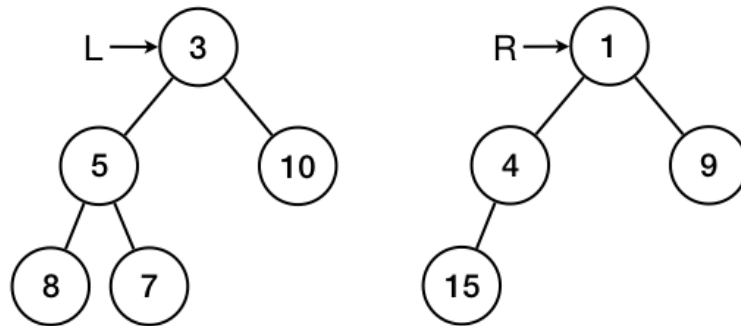
2pts

c.



3pts

d.



8pts

- e. Complete the function `merge`, which implements the specification provided in Task 1.b. You can use the type definitions and prototypes given on page 7:

```

heap* merge(heap *L, heap *R) {
    REQUIRES(is_mergeable_heap(L) && is_mergeable_heap(R));

    if (L == NULL) _____;

    if (R == NULL) _____;

    if (L->value <= R->value) {

        L->right = _____;

        ENSURES(_____);

        return _____;
    } else {
        R->right = _____;

        ENSURES(_____);

        return _____;
    }
}

```

4pts

- f. Complete function `delete_min`, which returns the heap resulting from removing the root from the heap `H`. You can use the type definitions and prototypes given on page 7:

```

heap* delete_min(heap *H) {
    REQUIRES(_____);

    _____;

    _____;

    ENSURES(_____);

    _____;

}

```

2pts

- g. What is the simplest and tightest worst-case complexity of `merge`, when invoked on heaps with n_L and n_R nodes? *Hint: observe that `merge` only traverses right subtrees.*

$O(\underline{\hspace{2cm}})$

Task 2 Next, we touch on *leftist heaps*, a more elaborate form of mergeable heaps, that maintain in addition to the order invariant also the *leftist shape invariant*. The shape invariant makes sure that the left subtree of each node is at least as deep as its right subtree.

To express this invariant, we extend the heap struct given in Task 1 with the field *rank*, which records the number of nodes along the path from the root of a heap to its rightmost leaf. More formally, the rank of a heap H is:

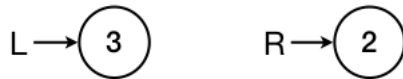
- 0 if H is empty (i.e., $H == \text{NULL}$), and otherwise
- 1 plus the rank of its right subtree.

A heap H is *leftist*, if H satisfies the order invariant and if, for every node x in H , the rank of x 's left subtree is *greater than or equal to* the rank of x 's right subtree.

To maintain the leftist shape invariant, *merge* now needs to swap the left subtree with the right subtree before returning, if the rank of the left subtree is smaller than the rank of the right subtree. For each of the examples below, draw the heap resulting from calling the new *merge* on the two given heaps L and R in the box provided:

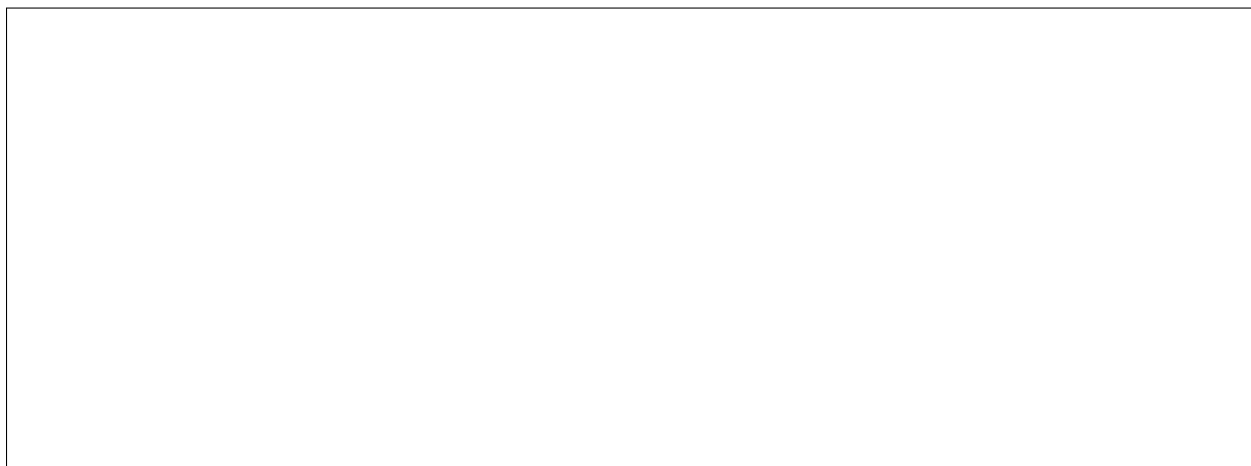
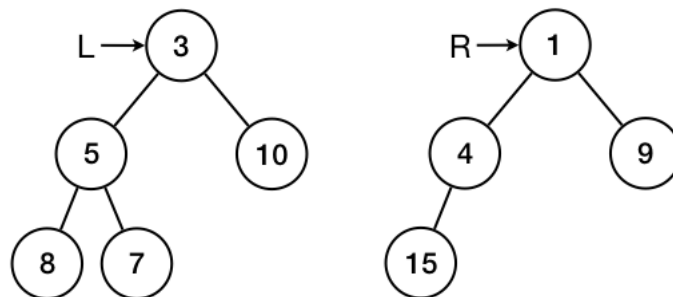
2pts

a.



3pts

b. You may find the scratch pages at the end of this exam handy.



5 Virtual Machines [C] (60 points)

This question deals with the C0VM. The specifications of a selection of C0VM instructions can be found on page 27 of this exam. You may assume the same implementation-defined C behaviors encountered in homework (e.g., 8-bit **chars** and 4-byte **ints**).

Task 1 The C0VM instruction `ildc` loads large integer constants, for example `0x15122`, onto the operand stack. In this exercise, we will pretend that this instruction does not exist, and achieve the same effect with some of the remaining instructions on page 27. We will do so in two steps.

4pts

- a. Write a C0 expression that evaluates to `0x15122` using only integers in the range `[-128, 128]`. Feel free to express these integers in either decimal or hexadecimal at your convenience. *Hint: C0's bitwise operators may come really handy.*

_____ == 0x15122

10pts

- b. Using the above as a guideline, write a C0VM bytecode fragment that loads the integer `0x15122` onto the operand stack. Write the bytecode values in the left column (e.g., `"15 0A"`). Feel free to write the corresponding instructions (here `"vload 10"`), and the contents of the stack in comments to their right.

```

# Initial stack: S
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
_____ # _____
# Final stack: S, 0x15122

```

Task 2 It is sometimes useful to provide special bytecode instructions for operations that programmers use very frequently. One such operation is the increment or decrement of a value by a small integer, e.g., $i+2$ or $x-1$.

We will do so using the new bytecode operation `iincr `, which increments the (integer) value on the top of the operand stack by ``, a signed number that fits in one byte. If `` is negative, this value is decremented. Using the notation seen in class, it is defined as follows:

```
0xd0 iincr <b>    S, x:w32 -> S, x+b:w32
```

10pts

- a. Implement the instruction `iincr` according to the above spec. Recall that `val2int` converts a `c0_value` to an integer, that `int2val` converts an integer to a `c0_value`, that `c0v_pop` pops a `c0_value` from the operand stack and that `c0v_push` pushes a `c0_value` onto the stack. You can assume that `P` is a `ubyte` array that holds the bytecode, `pc` is the current program counter, `S` is the operand stack, and `V` is the `c0_value` array containing the local variables of the current function. Recall also that this is part of a `switch` statement. Make sure to cast appropriately.

```
case IINCR: {

}

```

8pts

- b. Using `iincr`, complete the bytecode that implements the function

```
int f(int x) {
    x++;
    return x;
}
```

You will need exactly the number of lines provided. Fill all blanks.

```
<f>
00 01      # number of arguments = 1
00 01      # number of local variables = 1

_____ # code length = _____ bytes

_____ # _____
_____ # _____
_____ # _____

15 00      # vload 0          # x
B0         # return          #
```

Task 3 This simple example suggests an instruction even more useful than `incr`: an instruction that increments a *local variable* rather than the value at the top of the stack. This will allow C0 statements like `x++`; or `i -= 3`; to be implemented using a single C0VM instruction.

2pts

a. Give an abstract definition of this instruction, which we call `vincr`:

```
0xd1 vincr <b,i> _____
```

10pts

b. Complete the implementation of this instruction:

```
case VINCR: {
```

```
}
```

4pts

c. Complete the bytecode for the function `f` in Task 2.b using `vincr` rather than `incr`. Fill all blanks.

```
#<f>
00 01      # number of arguments = 1
00 01      # number of local variables = 1

_____ # code length = _____ bytes

_____ # _____

15 00      # vload 0      # x
B0         # return      #
```

Task 4 The compiler has developed a bug and is producing bytecode that may cause assertion failures or undefined behavior in your VM! (It's not too bad, though: the examples below have no funny business with the code length, and the mnemonics reported in comments faithfully represent the bytecode to the left.)

If any of these main functions execute without error or incident, write down the value they will return. If any of these main functions causes assertion failures or undefined behavior in the VM, then circle the *first* line that will cause unacceptable (e.g., assertion failures), undefined, or unpredictable behavior when executed by the COVM.

4pts

a.

```

00 00      # number of arguments = 0
00 00      # number of local variables = 0
00 08      # code length = 8 bytes
10 05      # bipush 5
10 FF      # bipush -1
59         # dup
68         # imul
78         # ishl
B0         # return

```

4pts

b.

```

00 00      # number of arguments = 0
00 00      # number of local variables = 0
00 08      # code length = 8 bytes
13 00 01   # ldc 15122 (defined in integer pool)
10 0A      # bipush 10
64         # isub
60         # iadd
B0         # return

```

4pts

c.

```

00 00      # number of arguments = 0
00 00      # number of local variables = 0
00 14      # code length = 20 bytes
10 15      # bipush 21
59         # dup
9F 00 06   # if_cmpeq +6
A7 00 0F   # goto +15
10 00      # bipush 0
10 2A      # bipush 42
4E         # imstore
A7 00 05   # goto +5
10 2A      # bipush 42
B0         # return

```

6 Graph Representation [C] (50 points)

The minimal graph interface seen in class, and reported below, did not provide a function that returns the neighbors of a vertex. As a consequence, some of the algorithms we developed, like DFS and BFS, had a higher complexity than otherwise achievable. In this exercise, we will add such a function, `get_neighbors`, together with some building blocks. The resulting extended interface is as follows (extensions are marked as `// NEW`):

```

typedef unsigned int vertex;
typedef struct graph_header* graph_t;

typedef struct adjlist_node adjlist;           // NEW
struct adjlist_node {                          // NEW
    vertex vert;                               // NEW
    adjlist *next;                             // NEW
};                                              // NEW

graph_t graph_new(unsigned int numvert);
//@ensures \result != NULL;

void graph_free(graph_t G);
//@requires G != NULL;

unsigned int graph_size(graph_t G);
//@requires G != NULL;

bool graph_hasedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);

void graph_addege(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);
//@requires v != w && !graph_hasedge(G, v, w);

adjlist* get_neighbors(graph_t G, vertex v);    // NEW
//@requires G != NULL && v < graph_size(G);    // NEW

void free_neighbors(adjlist *L);              // NEW

```

The type `adjlist`, taken identically from the adjacency list representation of graphs, is simply a NULL-terminated linked list of vertices. The function `get_neighbors(G, v)` returns the neighbors of vertex `v` in graph `G` as such a list. It should not allocate new memory unless it really has to. The function `free_neighbors(L)` disposes of a list of vertices `L` returned by `get_neighbors`, *if appropriate*.

Task 1 The following client-side code uses the new functions to print a graph.

```
1 void graph_print(graph_t G) {
2     REQUIRES(G != NULL);
3     for (vertex v = 0; v < graph_size(G); v++) {
4         printf("Vertices connected to %u: ", v);
5         for (adjlist *p = get_neighbors(G, v); p != NULL; p = p->next)
6             printf(" %u,", p->vert);
7         printf("\n");
8         free_neighbors(p);
9     }
10 }
```

3pts

- a. The above code has an error. On what line does this error occur and what is causing it?

7pts

- b. Modify the code for print_graph above in order to fix this error (you may abbreviate the print statements if you want).

```
void graph_print(graph_t G) {
    REQUIRES(G != NULL);
```

```
}
```


Task 2 We will now extend the *adjacency list* implementation of the graph interface with the added functions. Recall the definition of the internal type `graph` and the prototype of some of the specification functions:

<pre>typedef struct graph_header graph; struct graph_header { unsigned int size; adjlist **adj; };</pre>	<pre>bool is_vertex(graph *G, vertex v); bool is_graph(graph *G);</pre>
---	---

6pts

a. *Efficiently* implement the function `get_neighbors`. Include appropriate contracts.

```
adjlist* get_neighbors(graph *G, vertex v) {

}

}
```

3pts

b. What is the complexity of `get_neighbors` as a function of the number v of vertices and the number e of edges in the input graph?

$O(\text{_____})$

3pts

c. Complete the body of `free_neighbors` so that, by the time we are done using a graph, *all allocated memory has been freed and none has been freed twice*.

```
void free_neighbors(adjlist *L) {

}

}
```

3pts

d. Using this implementation of graphs, what is the simplest and tightest worst-case complexity of `graph_print`? *Hint: consider how many things get printed*.

$O(\text{_____})$

Task 3 Next, we will do the same with the *adjacency matrix* implementation of the graph interface. The internal type `graph` is now defined as follows:

```
typedef struct graph_header graph; | bool is_vertex(graph *G, vertex v);
struct graph_header {             | bool is_graph(graph *G);
    unsigned int size;
    bool *matrix; // 1-D array
};
```

In the code questions below, recall that you can use interface functions.

6pts

a. *Efficiently* implement the function `get_neighbors`. Include appropriate contracts.

```
adjlist* get_neighbors(graph *G, vertex v) {

}

}
```

3pts

b. What is the complexity of `get_neighbors` as a function of the number v of vertices and the number e of edges in the input graph?

$O(\underline{\hspace{2cm}})$

3pts

c. Complete the body of `free_neighbors` so that, by the time we are done using a graph, *all allocated memory has been freed* and *none has been freed twice*.

```
void free_neighbors(adjlist *L) {

}

}
```

3pts

d. Using this implementation of graphs, what is the simplest and tightest worst-case complexity of `graph_print`? *Hint: consider how many things get printed.*

$O(\underline{\hspace{2cm}})$

Task 4 The adjacency matrix code you worked on in lab implemented the matrix as an array of arrays (a 2D array). The above type **struct** `graph_header` uses instead a one-dimensional array, like in the images assignment. This part asks you to write some additional functions relative to one-dimensional arrays.

5pts

- a. Complete the specification function `is_graph(G)` that checks that the adjacency matrix representation of its input graph is valid. The helper function `is_valid_row` checks that the row for vertex `v` of the adjacency matrix of graph `G` is valid.

```

bool is_valid_row(graph *G, vertex v) {
    REQUIRES(G != NULL);

    size_t v_row = _____; // Start index of v's row

    if (_____ ) // No self loops
        return false;

    for (_____ ) {
        // Every outgoing edge a corresponding edge coming back to it

        if (_____ )
            return false;
    }
    return true;
}

bool is_graph(graph *G) {

    if (G == NULL) return _____;

    for (vertex v = 0; v < G->size; v++)
    {
        if (!is_valid_row(G, v)) return _____;
    }
    return _____;
}

```

5pts

- b. Complete the function `graph_new`, including the appropriate contracts.

```

graph* graph_new(unsigned int size) {

}

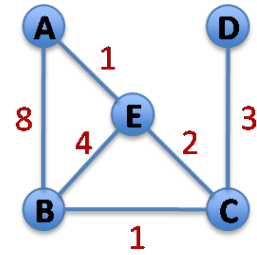
```

7 Shortest Path [C] (40 points)

In a graph with weighted edges, the *weight of a path* is the sum of the weights of its edges. For example, the path $(A-B-C-D)$ in the example graph on the right has weight 12 ($= 8 + 1 + 3$). The path $(A-B)$ has weight 8 since its one edge has weight 8. The empty path which starts and ends at vertex A has weight 0.

A *shortest path* between two vertices v and w is a path of minimum weight among all the paths between them. Its weight is called the *shortest distance* between v and w . In our example, the shortest path between A and B is $A-E-C-B$, and therefore the shortest distance between them is 4.

In this exercise, we will implement a function that computes the shortest distance between a given vertex and every other vertex in a graph. The algorithm it embodies is known as *Dijkstra's algorithm*, after the Dutch computer scientist Edsger Dijkstra.



But first, we need to update the graph interface **studied in class** to handle *weighted* graphs:

```
typedef unsigned int vertex;
typedef struct graph_header* graph_t;

graph_t graph_new(unsigned int numvert); // New graph with numvert vertices

void graph_free(graph_t G);

unsigned int graph_size(graph_t G); // Number of vertices in the graph

// Returns the weight of (v1,v2) or -1 if they are not connected
int graph_hasedge(graph_t G, vertex v1, vertex v2);
    //@requires v1 < graph_size(G) && v2 < graph_size(G);

void graph_addedge(graph_t G, vertex v1, vertex v2, int weight);
    //@requires v1 < graph_size(G) && v2 < graph_size(G);
    //@requires v1 != v2 && graph_hasedge(G, v1, v2) == -1 && weight > 0;
```

The main changes are **highlighted**:

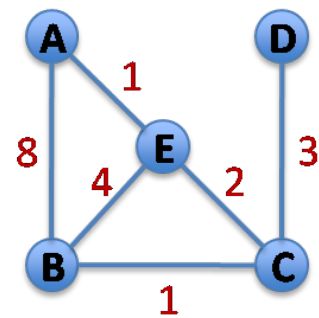
1. If graph G has an edge between vertices $v1$ and $v2$, the function `graph_hasedge(G, v1, v2)` now returns the weight of this edge. If $v1$ and $v2$ are not connected, it returns -1 .
2. Naturally, the function `graph_addedge` now takes the weight of the added edge as an additional argument. Weights are strictly positive.

Throughout this exercise, we will assume for simplicity that the weight of every path in the given graphs is less than `INT_MAX`.

Dijkstra’s algorithm takes as input a graph G and a vertex $start$. It outputs an array $dist$ that maps every vertex in G to its shortest distance to $start$. If $start$ and v are not connected, $dist[v]$ contains INT_MAX . Like DFS and BFS, Dijkstra’s algorithm uses a mark array to keep track of visited vertices. Like these algorithms, it explores the graph G beginning at $start$ and following the edges of connected vertices. It keeps track of the vertices to visit next in a *priority queue* where the priority is given by the shortest distance between $start$ and each vertex at the time of insertion (shorter paths may be discovered later). Shorter distances have higher priority. During execution $dist[v]$ contains the shortest distance between $start$ and v known *so far* (or INT_MAX if v has not yet been visited).

Dijkstra’s algorithm proceeds as follows:

1. Mark all vertices as unvisited.
2. Create $dist$ and set the distance to all vertices other than $start$ to INT_MAX . Set the distance to $start$ to 0.
3. Create a priority queue Q and insert the pair $(start, 0)$ in it.
- 4. For as long as Q is not empty
 - (a) Get the vertex v with minimum weight from Q
 - (b) If v is unmarked
 - mark it
 - for every unmarked neighbor w of v
 - i. let d be the weight of the edge (v, w) in G
 - ii. if $dist[v] + d < dist[w]$
 - update $dist[w]$ to $dist[v] + d$
 - insert $(w, dist[w])$ into Q
- ⇒ 5. Return $dist$



10pts

Task 1 Complete the table below by simulating the execution of Dijkstra’s algorithm on the above graph. The first three rows have been filled for you already. The first row shows the contents of Q and $dist$, and the marked vertices just before entering the loop (indicated with → above). The subsequent rows show these values and the selected vertex v at the end of each iteration of the loop (indicated as ⇒).

	v	Q	dist					marked
			A	B	C	D	E	
→		(A, 0)	0	∞	∞	∞	∞	
⇒	A	(B, 8), (E, 1)	0	8	∞	∞	1	A
⇒	E	(B, 8), (B, 5), (C, 3)	0	5	3	∞	1	A, E
⇒								
⇒								
⇒								
⇒								
⇒								

Since Dijkstra's algorithm relies on a priority queue, we need to define the type of its elements and the notion of priority. The interface of generic priority queues is given on page 28 of this exam. Since we are now working in C, it upgrades the interface we saw in class with functions to deallocate memory. The differences are highlighted.

The type of elements to insert in the priority queue are vertices together with their best-known distance from start at that point. This is captured by the type:

```
typedef struct vert_dist_pair vert_dist;
struct vert_dist_pair {
    vertex vert;
    int    dist; // Best known distance from start to vert
};
```

The (pre-implemented) helper function

```
vert_dist* mk_vert_dist(vertex v, int d);
```

packages vertex *v* and distance *d* into an entity of type `vert_dist` and returns a pointer to it.

6pts

Task 2 Define the function `has_shorter_distance(e1, e2)` so that it returns `true` if the distance component of *e1* is strictly less than the one of *e2*, and `false` otherwise.

```
bool has_shorter_distance(elem e1, elem e2) {
    REQUIRES(e1 != NULL && e2 != NULL);

    }
}
```

5pts

Task 3 Assuming our priority queue is not self-resizing, how big should we make it to be sure we won't run out of space no matter how many edges the graph contains? Said differently, what capacity should we specify when calling `pq_new`? Give the smallest such value relative to the number of vertices in the graph.

4pts

Task 4 Next, we will implement Dijkstra's algorithm as a single non-recursive function. What kind of arrays can the mark array, `mark`, and the shortest distance array, `dist`, be? Check all that apply.

```
mark:   Stack-allocated    Heap-allocated
dist:   Stack-allocated    Heap-allocated
```

15pts

Task 5 Based on the pseudo-code on page 21 (and mostly repeated below), complete the following client-side function that implements Dijkstra's algorithm.

```
int* dijkstra(graph_t G, vertex start) {
    REQUIRES(G != NULL && start < graph_size(G));

    // Mark all vertices as unvisited

    _____ mark _____;

    for (vertex v = 0; v < graph_size(G); v++)
        mark[v] = false;    // v has not been visited yet

    // Create dist and set the distance to all vertices other than
    // start to INT_MAX. Set the distance to start to 0.

    _____ dist _____;

    for (vertex v = 0; v < graph_size(G); v++)
        dist[v] = INT_MAX; // No known distance from start
    dist[start] = 0;

    // Create a priority queue Q and insert (start,0) in it.
    int capacity = /* Your answer from task 3 -- nothing to do */;

    pq_t Q = pq_new(capacity, _____, _____);

    pq_add(Q, _____);
}
```

(Continued on next page)

Task 6 (continued from previous page)

```

while ( _____ ) // For as long as Q is not empty
{
    // Get the vertex v with minimum weight from Q

    vert_dist *vd = _____;

    vertex v = _____;

    if ( _____ ) { // if v is unmarked
        _____; // mark it

        // for every unmarked neighbor w of v,
        // let d be the weight of the edge (v,w)
        for (vertex w = 0; w < graph_size(G); w++) {

            int d = _____;

            if ( _____ ) {

                if (dist[v] + d < dist[w]) {

                    _____; // update dist[w]

                    // insert (w, dist[w]) into Q

                    _____;

                }

            }

        }

        _____ // anything to clean up?
    }

    _____ // anything to clean up?

    _____ // anything to clean up?

    return dist;
}

```


(This page intentionally left blank.)

(This page intentionally left blank.)

REFERENCE: Selected C0VM Instructions

Stack operations

0x59 `dup` S, v -> S, v, v
0x57 `pop` S, v -> S

Arithmetic

0x60 `iadd` S, x:w32, y:w32 -> S, x+y:w32
0x7E `iand` S, x:w32, y:w32 -> S, x&y:w32
0x6C `idiv` S, x:w32, y:w32 -> S, x/y:w32
0x68 `imul` S, x:w32, y:w32 -> S, x*y:w32
0x80 `ior` S, x:w32, y:w32 -> S, x|y:w32
0x70 `irem` S, x:w32, y:w32 -> S, x%y:w32
0x78 `ishl` S, x:w32, y:w32 -> S, x<<y:w32
0x7A `ishr` S, x:w32, y:w32 -> S, x>>y:w32
0x64 `isub` S, x:w32, y:w32 -> S, x-y:w32
0x82 `ixor` S, x:w32, y:w32 -> S, x^y:w32

Local Variables

0x15 `vload <i>` S -> S, v (v = V[i])
0x36 `vstore <i>` S, v -> S (V[i] = v)

Constants

0x01 `aconst_null` S -> S, null:*
0x10 `bipush ` S -> S, x:w32 (x = (w32)b, sign extended)
0x13 `ildc <c1,c2>` S -> S, x:w32 (x = int_pool[(c1<<8)|c2])
0x14 `aldc <c1,c2>` S -> S, a:* (a = &string_pool[(c1<<8)|c2])

Control Flow

0x00 `nop` S -> S
0x9F `if_cmpeq <o1,o2>` S, v1, v2 -> S (pc = pc+(o1<<8|o2) if v1 == v2)
0xA0 `if_cmpne <o1,o2>` S, v1, v2 -> S (pc = pc+(o1<<8|o2) if v1 != v2)
0xA1 `if_icmplt <o1,o2>` S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x < y)
0xA7 `goto <o1,o2>` S -> S (pc = pc+(o1<<8|o2))
0xB0 `return` ., v -> . (return v to caller)

Memory

0xBB `new <s>` S -> S, a:* (*a is now allocated, size <s>)
0x2E `imload` S, a:* -> S, x:w32 (x = *a, a != NULL, load 4 bytes)
0x4E `imstore` S, a:*, x:w32 -> S (*a = x, a != NULL, store 4 bytes)

REFERENCE: Generic Priority Queues in C

```

/***** Client interface *****/

typedef void* elem; // Supplied by client

// f(x,y) returns true if e1 is STRICTLY higher priority than e2
typedef bool has_higher_priority_fn(elem e1, elem e2);
/*@requires e1 != NULL && e2 != NULL;

typedef void elem_free_fn(elem e);
/*@requires e != NULL;

/***** Library interface *****/

typedef struct heap_header* pq_t;

bool pq_empty(pq_t Q)
  /*@requires Q != NULL; @*/ ;

bool pq_full(pq_t Q)
  /*@requires Q != NULL; @*/ ;

pq_t pq_new(int capacity, has_higher_priority_fn* prior, elem_free_fn* fr)
  /*@requires capacity > 0 && prior != NULL && fr != NULL; @*/
  /*@ensures \result != NULL && pq_empty(\result); @*/ ;

void pq_add(pq_t Q, elem x)
  /*@requires Q != NULL && !pq_full(Q) && x != NULL; @*/ ;

elem pq_rem(pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q); @*/
  /*@ensures \result != NULL; @*/ ;

elem pq_peek(pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q); @*/
  /*@ensures \result != NULL; @*/ ;

void pq_free(pq_t Q)
  /*@requires Q != NULL; @*/ ;

```

C1-style contracts are given as documentation. Changes w.r.t. in-class version are highlighted.