15-122: Principles of Imperative Computation, Spring 2024

Written Homework 12

Due on Gradescope: Monday 15th April, 2024 by 9pm

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

This written homework provides practice with C features such as strings and casting, and with the C0VM.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- **Kami**, **Adobe Acrobat Online**, or ***DocHub***, some web-based PDF editors that work from anywhere.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

Please do not add, remove or reorder pages.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	4	Total
Points:	4.5	5	2	3.5	15
Score:					

1. **C0VM**

Consider the following C0 code that populates a structure with values.

```
typedef struct gap_buffer gapbuf;
struct gap_buffer {
    char[] buffer; /* \length(buffer) == limit */
    int limit;     /* limit > 0 */
    int gap_start; /* 0 <= gap_start */
    int gap_end;   /* gap_start <= gap_end <= limit */
};

int main() {
  gapbuf* gb = alloc(gapbuf);
  gb->buffer = alloc_array(char, 65536);
  gb->limit = 65536;
  gb->gap_start = 48112;
  gb->gap_end = gb->gap_start;
  return 1;
}
```

0.5pts

1.1 Fill in the missing instructions (on the next page) in the following bytecode that corresponds to the above C0 code. Fill in **both** the hex opcodes **and** the instruction name and its argument(s) in decimal. Be careful, the answers may or may not match the bytecode output generated by compiling the C0 code directly.

```
1  C0 C0 FF EE       # magic number
2  00 17             # version 11, arch = 1 (64 bits)
3
4  00 02             # int pool count
5  # int pool
6  00 01 00 00
7  00 00 BB F0
8
9  00 00             # string pool total size
10 # string pool
11
12 00 01             # function count
13 # function_pool
14
```

(bytecode continues on next page)

© Carnegie Mellon University 2024

(continued from previous page)

```
15 #<main>
16 00                   # number of arguments = 0
17 01                   # number of local variables = 1
18 00 2B                # code length = 43 bytes
19 BB 18     # new 24         # alloc(gapbuf)
20 ________  # ____________   # gb = alloc(gapbuf);
21 15 00     # vload 0        # gb
22 62 00     # aaddf 0        # &gb->buffer
23 ________  # ____________   # 65536
24 ________  # ____________   # alloc_array(char, 65536)
25 4F        # amstore        # gb->buffer = alloc_array(char, 65536);
26 15 00     # vload 0        # gb
27 62 08     # aaddf 8        # &gb->limit
28 ________  # ____________   # 65536
29 4E        # imstore        # gb->limit = 65536;
30 15 00     # vload 0        # gb
31 62 0C     # aaddf 12       # &gb->gap_start
32 13 00 01  # ildc 1         # 48112
33 4E        # imstore        # gb->gap_start = 48112;
34 15 00     # vload 0        # gb
35 62 10     # aaddf 16       # &gb->gap_end
36 15 00     # vload 0        # gb
37 ________  # ____________   # &gb->gap_start
38 2E        # imload         # gb->gap_start
39 4E        # imstore        # gb->gap_end = gb->gap_start;
40 10 01     # bipush 1       # 1
41 B0        # return         #
42
43 00 00                # native count
44 # native pool
```

1pt

1.2 After execution reaches line 21 in the byte code, there is only one value on the operand stack; assume it is the pointer `0xC0`.

You will now trace the execution forward and determine the four operand stack states after each of lines 22–25 is executed. Write pointers in hexadecimal and numbers in decimal. The stack grows upward. Assume that **`alloc_array`** returns `0x80`. Lines of code you had to fill are drawn as "_______".

(You may not need all the provided spaces.)

Fill in the contents of the stack immediately before and after each of the following lines is executed:

	line 22		line 23		line 24		line 25	
	`aaddf 0`		_______		_______		`amstore`	
		____		____		____		____
		____		____		____		____
`0xC0`		____		____		____		____
(rest of the stack)		(rest of the stack)		(rest of the stack)		(rest of the stack)		(rest of the stack)

© Carnegie Mellon University 2024

2pts

1.3 The following bytecode file was generated by the C0 compiler. Some comments may have been blanked out or deleted, but all instructions are untouched. Write a C0 program that will generate this bytecode file. You do not have to fill in the blanked out comments, but feel free to do so if you find it useful.

(Note that the bytecode continues on the following page.)

```
1  C0 C0 FF EE       # magic number
2  00 15             # version 10, arch = 1 (64 bits)
3
4  00 00             # int pool count
5  # int pool
6
7  00 11             # string pool total size
8  # string pool
9  48 61 70 70 79 20 43 61 72 6E 69 76 61 6C 21 0A 00
10
11 00 02             # function count
12 # function_pool
13
14 #<main>
15 00                # number of arguments = 0
16 00                # number of local variables = 0
17 00 11             # code length = 17 bytes
18 14 00 00 # aldc 0          # ____________________
19 B7 00 00 # invokenative 0  # ____________________
20 57       # pop             # (ignore result)
21 10 00    # bipush 0        # ____________________
22 10 01    # bipush 1        # ____________________
23 10 0A    # bipush 10       # ____________________
24 B8 00 01 # invokestatic 1  # ____________________
25 B0       # return          #
26
27 #<g>
28 03                # number of arguments = 3
29 03                # number of local variables = 3
30 00 30             # code length = 48 bytes
31 15 02    # vload 2         # ____________________
32 10 00    # bipush 0        # ____________________
33 9F 00 06 # if_cmpeq +6     # ____________________
34 A7 00 09 # goto +9         # ____________________
35 15 00    # vload 0         # ____________________
36 B0       # return          #
37 A7 00 03 # goto +3         # ____________________
38 15 02    # vload 2         # ____________________
39 10 01    # bipush 1        # ____________________
40 9F 00 06 # if_cmpeq +6     # ____________________
41 A7 00 09 # goto +9         # ____________________
```

© Carnegie Mellon University 2024

```
42 15 01     # vload 1          # ________________________________
43 B0        # return           #
44 A7 00 03  # goto +3          # ________________________________
45 15 01     # vload 1          # ________________________________
46 15 00     # vload 0          # ________________________________
47 15 01     # vload 1          # ________________________________
48 60        # iadd             # ________________________________
49 15 02     # vload 2          # ________________________________
50 10 01     # bipush 1         # ________________________________
51 64        # isub             # ________________________________
52 B8 00 01  # invokestatic 1   # ________________________________
53 B0        # return           #
54
55 00 01                # native count
56 # native pool
57 00 01 00 06          # print
```

© Carnegie Mellon University 2024

1pt

1.4 This question has to do with the function `g` in the bytecode given in part (3) above.

When execution reaches the instruction on line 52, there are three values on the operand stack, listed below with the topmost being at the top of the stack.

You will now trace the execution backward and determine the four operand stack states before each of lines 48–51 is executed. Write your numbers in hexadecimal. The stack grows upward. *(You may not need all the provided spaces.)*

Fill in the contents of the stack immediately before and after each of the following lines is executed:

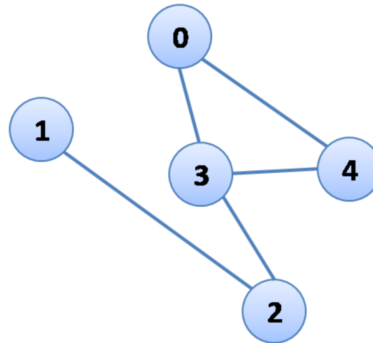
	line 48		line 49		line 50		line 51	
	`iadd`		`vload 2`		`bipush 1`		`isub`	
____		____		____		____		
____		____		____		____		`0x9`
____		____		____		____		`0x15`
____		____		____		____		`0xD`
(rest of the stack)		(rest of the stack)		(rest of the stack)		(rest of the stack)		(rest of the stack)

© Carnegie Mellon University 2024

2. **Graph Representation**

1pt

2.1 Show the adjacency matrix that represents the graph drawn below. Indicate the presence of an edge with '**X**'; leave the cell blank when there is no edge.



© Carnegie Mellon University 2024

1.5pts

2.2 Recall the *adjacency list* representation of a graph from class:

```
typedef unsigned int vertex;
typedef struct adjlist_node adjlist;
struct adjlist_node {
    vertex vert;
    adjlist *next;
};
typedef struct graph_header graph;
typedef struct neighbor_header neighbors;
```

```
struct graph_header {
    unsigned int size;
    adjlist **adj;
};
struct neighbor_header {
  adjlist *next_neighbor;
};
```

Extend the graph interface with a *library* function `graph_countneighbors(G,v)` that returns the number of edges at vertex v of graph G. Be sure to include appropriate `REQUIRES` and `ENSURES` contracts. You may call any functions given in the code in class posted on our website for the lecture on representing graphs. Your solution should be as efficient as possible, without making any changes to the definition of any data structure used in the graph representation.

```
unsigned int graph_countneighbors(graph* G, vertex v) {

}
```

0.5pts

2.3 Give the worst-case asymptotic complexity of your function for a graph of v vertices and e edges, as a function of v and e.

$O($______________________________$)$

© Carnegie Mellon University 2024

1.5pts

2.4 Here is a *subset* of the interface to the graph library in `graph.h` (some contracts have been omitted):

```
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

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

void graph_addedge(graph_t G, vertex v, vertex w); // Edge can't be in graph!
  //@requires v < graph_size(G) && w < graph_size(G);
  //@requires v != w && !graph_hasedge(G, v, w);
```

Write another function to count the edges at a vertex. This must be a *client* function, that is, it may *only* use the types and functions listed above — in particular the function `graph_get_neighbors` is **not** available. You may use the fact that `vertex` is an integer type, and that it is the same type returned by `graph_size`. Be sure to include appropriate `REQUIRES` and `ENSURES` contracts.

```
unsigned int countneighbors(graph_t G, vertex v) {

}
```

0.5pts

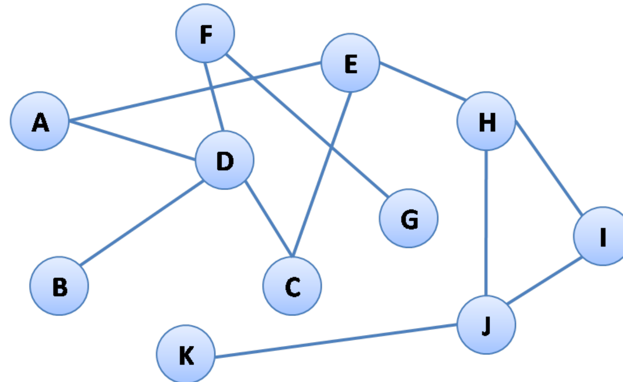
2.5 Give the worst-case asymptotic complexity of your function for a graph of v vertices and e edges, as a function of v and e. For this calculation, you may assume the adjacency list implementation.

$O($______________________________$)$

© Carnegie Mellon University 2024

3. **Graph Traversals**

1pt **3.1** Consider the graph:



Using **recursive** depth-first traversal, list the vertices in the order they are visited as we search from vertex J to vertex G. When we visit a vertex, we explore its outgoing edges in alphabetical order. Do not list a vertex again if you backtrack to it.

J,

List the vertices of the path found from J to G by the search.

Using a breadth-first traversal, list the vertices in the order that they are visited as we search from vertex J to vertex G. When we visit a vertex, we explore its outgoing edges in alphabetical order.

J,

List the vertices of the path found from J to G by the search.

1pt **3.2** In an undirected graph with v vertices, what is the maximum possible number of edges? (This kind of graph is called a *complete graph*). Express your answer in closed form as a function of v.

A path in a graph is called a *simple cycle* if it lets you go from a vertex to itself without repeating an edge or any intermediate vertex. What is the maximum possible number of edges in a graph with v vertices that contains no simple cycles?

© Carnegie Mellon University 2024

4. **Checking Paths**

We can represent an n-vertex path in a graph as a stack of n vertices where each vertex is connected by an edge to the vertex below it in the stack. For example, the 5-vertex path 1—3—5—7—3 will be represented as a 5-element stack, with 1 at the top, then 3, then 5, then 7 and finally 3. Thus, a path can have zero or more vertices, and cycles are permitted.

Here's the C interface for **generic** stack:

```
typedef void *elem;                     // stack element
typedef void elem_free_fn(elem x);      // function that frees an element

typedef struct stack_header *stack_t;   // Generic stacks

bool stack_empty(stack_t S)                                    // O(1)
/*@requires S != NULL; @*/ ;

stack_t stack_new()                                            // O(1)
/*@ensures \result != NULL && stack_empty(\result); @*/ ;

void push(stack_t S, elem x)                                   // O(1)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)                                            // O(1)
/*@requires S != NULL && !stack_empty(S); @*/ ;

void stack_free(stack_t S, elem_free_fn* elem_free)            // O(n)
/*@requires S != NULL; @*/
/* if elem_free is NULL, then elements will not be freed */ ;
```

You may **only** use the following subset of the graph interface:

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

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 && v < graph_size(G) && w < graph_size(G); @*/ ;

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

© Carnegie Mellon University 2024

2.5pts

4.1 Complete the code for the **client-side** function `check_path(G,S)` that return

- `true` if the stack `S` represents a path that is present in the graph `G`, and
- `false` if `S` does not represent a valid path in `G`.

You may use the specification function `stack_of_valid_vertices(G,S)` that returns `false` if any element in `S` is not a valid vertex for graph `G`, and `true` otherwise.

The stack `S` and its elements should be freed upon returning and your code should not leak memory. Your code should be provably safe. Recall that the stack library is generic. *You may write code in any blank space.*

```
bool check_path(graph_t G, stack_t S) {

  REQUIRES(____________________ && ____________________);

  REQUIRES(__________________________________________);

  if (______________________________________________) {


    return true;
  }
                                        // Get first vertex



  while (___________________________________________) {
                                        // Get next vertex


    if (____________________________________________) {


      return ________________________________________;
    }



  }


  return ____________________________________________;
}
```

© Carnegie Mellon University 2024

1pt

4.2 Consider a graph `G` with v vertices and e edges, and a stack `S` contains s elements. What is the worst-case asymptotic complexity of the call `check_path(G,S)` assuming an adjacenty list representation? What if we assume an adjacency matrix representation instead?

Adjacency list representation: $O($______________________$)$

Adjacency matrix representation: $O($______________________$)$

© Carnegie Mellon University 2024