15-122: Principles of Imperative Computation, Spring 2023

Written Homework 12

Due on Gradescope: Monday 17$^{\text{th}}$ April, 2023 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:

- ***PDFescape*** or ***DocHub***, two 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.*

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**

Each of the following bytecode files was generated by the C0 compiler. Some comments may have been blanked out or deleted, but all instructions are untouched. Write C0 programs that will generate these bytecode files. You do not have to fill in the blanked out comments, but feel free to do so if you find it useful.

1pt

1.1

```
C0 C0 FF EE       # magic number
00 15             # version 10, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 00             # string pool total size
# string pool

00 01             # function count
# function_pool

#<main>
00                # number of arguments = 0
02                # number of local variables = 2
00 26             # code length = 38 bytes
10 00    # bipush 0        # ____________________
36 00    # vstore 0        # ____________________
10 00    # bipush 0        # ____________________
36 01    # vstore 1        # ____________________
15 00    # vload 0         # ____________________
10 0A    # bipush 10       # ____________________
A1 00 06 # if_icmplt +6    # ____________________
A7 00 14 # goto +20        # ____________________
15 00    # vload 0         # ____________________
10 01    # bipush 1        # ____________________
60       # iadd            # ____________________
36 00    # vstore 0        # ____________________
15 01    # vload 1         # ____________________
15 00    # vload 0         # ____________________
60       # iadd            # ____________________
36 01    # vstore 1        # ____________________
A7 FF E8 # goto -24        # ____________________
15 01    # vload 1         # ____________________
B0       # return          #

00 00             # native count
# native pool
```

2.5pts

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

```
C0 C0 FF EE       # magic number
00 15             # version 10, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 15             # string pool total size
# string pool
48 61 70 70 79 20 54 68 61 6E 6B 73 67 69 76 69 6E 67 21 0A 00

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 0F             # code length = 15 bytes
14 00 00 # aldc 0         # ______________________
B7 00 00 # invokenative 0 # ______________________
57       # pop            # (ignore result)
10 00    # bipush 0       # ______________________
10 0A    # bipush 10      # ______________________
B8 00 01 # invokestatic 1 # ______________________
B0       # return         #

#<f>
02                # number of arguments = 2
03                # number of local variables = 3
00 23             # code length = 35 bytes
15 01    # vload 1        # ______________________
10 00    # bipush 0       # ______________________
9F 00 06 # if_cmpeq +6    # ______________________
A7 00 0A # goto +10       # ______________________
15 00    # vload 0        # ______________________
36 02    # vstore 2       # ______________________
A7 00 12 # goto +18       # ______________________
15 00    # vload 0        # ______________________
15 01    # vload 1        # ______________________
60       # iadd           # ______________________
15 01    # vload 1        # ______________________
10 01    # bipush 1       # ______________________
64       # isub           # ______________________
B8 00 01 # invokestatic 1 # ______________________
36 02    # vstore 2       # ______________________
```

```
15 02     # vload 2         # ________________________________
B0        # return          #

00 01              # native count
# native pool
00 01 00 10        # print
```

1pt

1.3 This question has to do with the function `f` in the bytecode given in part (2) above.

When execution reaches the instruction on line 39, there are two values on the operand stack, listed below with the topmost being at the top of the stack. (It will be helpful to be aware of where these values came from.)

You will now trace the execution forward and determine the four operand stack states after each of lines 39–42 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:

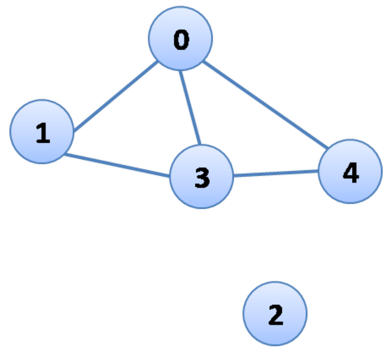
	line 39	line 40	line 41	line 42
	`iadd`	`vload 1`	`bipush 1`	`isub`

`0x9`				
`0xA`				
(rest of the stack)	(rest of the stack)	(rest of the stack)	(rest of the stack)	(rest of the stack)

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.



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.

$O($______________________________$)$

1.5pts

2.4 Here is a *subset* of the interface to the graph library in `graph.h`:

```
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

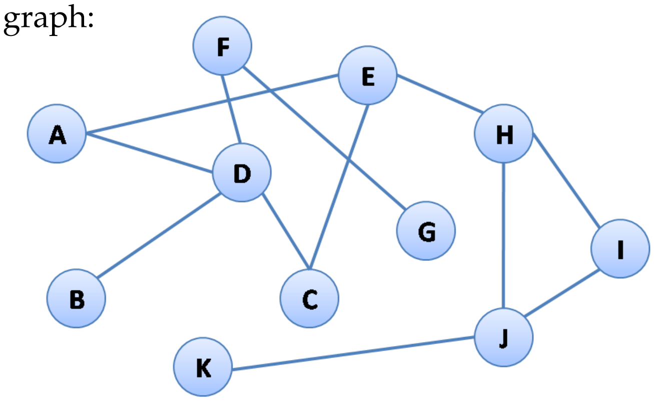
2.5 Give the worst-case asymptotic complexity of your function for a graph of v vertices and e edges, as a function of v and e. For this calculation, you may assume the adjacency list implementation.

$O(\underline{\hspace{6cm}})$

3. **Graph Traversals**

1pt

3.1 Consider the following 11-vertex graph:



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

A,

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

A,

1pt

3.2 For an undirected graph with n vertices, what is the maximum number of edges this graph can have? (This is called a *complete graph*). Express your answer in closed form as a function of n.

A politician must plan a trip to visit n cities and give a speech once at each city and then return home, starting and ending in the politician's home city (which is one of the n cities). The politician can fly directly from any city to any other city. The politician does not want to visit each city more than once and wants to return back to the home city at the end of the trip. The politician's staff wants to figure out all of the possible trips that the politician can make to determine which will have the maximum impact on voters.

Express the number of possible unique trips to visit the n cities in big-O notation as a function of n in its simplest, tightest form. (HINT: Think about this as a complete graph.)

$O(\underline{\hspace{6cm}})$

4. **Checking Paths**

We can represent a path in a graph as a stack of vertices where each vertex is connected by an edge to the vertex below it in the stack. For example, the 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 is either empty or has at least one node, 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);                     @*/ ;
```

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 _________________________________________________________;
}
```

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: $O($__________________________$)$