15-122 : Principles of Imperative Computation, Spring 2016

Written Homework D/E

Due: Monday 18th April, 2016

Name: ______________________

Andrew ID: ______________________

Section: ______________________

This written homework covers the C0VM and graphs.

The assignment is due by 1:30pm on Monday 18th April, 2016.

This assignment can be completed in one of two ways:
(A) by printing this file, handwriting your answers, and scanning it, or
(B) by editing this file and **printing it to another PDF file**

You shall then submit your solution to Gradescope.

1. **C0VM**

   Each of the following bytecode files was generated by the C0 compiler. Some comments may have been edited out, but all instructions are untouched. Write C0 programs that will generate these bytecode files.

2pts (a)

```
C0 C0 FF EE       # magic number
00 0D             # version 6, 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 00             # number of arguments = 0
00 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
```

2pts (b) (Note that the bytecode continues on the following page.)

```
C0 C0 FF EE       # magic number
00 0D             # version 6, 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 00             # number of arguments = 0
00 03             # number of local variables = 3
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>
00 02             # number of arguments = 2
00 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 (c) This question has to do with the function `f` in the bytecode given in part (b) above. When execution reaches the instruction on line 39 there are two values on the operand stack; assume they are `0x0000000A` and `0x00000009`. (It will be helpful to be aware of where these values came from.)

Write the four operand stack states after each of lines 39–42 is executed. The elements in your stack should be 32-bit hexadecimal numbers. The top of your stack should be on the right-hand side. You may not need all the provided spaces

Immediately after executing line 39: `iadd`

________________, ________________, ________________, ________________

Immediately after executing line 40: `vload 1`

________________, ________________, ________________, ________________

Immediately after executing line 41: `bipush 1`

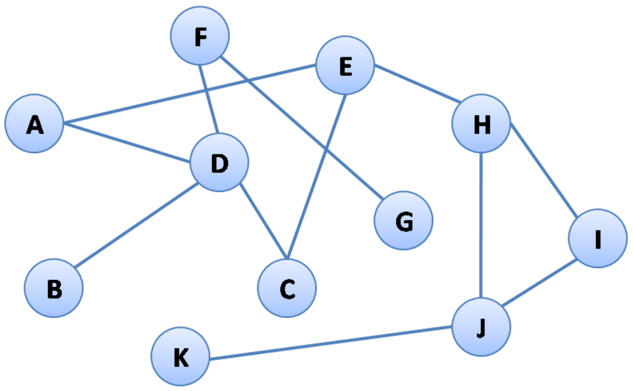
________________, ________________, ________________, ________________

Immediately after executing line 42: `isub`

________________, ________________, ________________, ________________

2. **Graphs and Graph Traversals**

4pts (a) Consider the graph:



Using a depth-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. Do not list a vertex again if you backtrack to it.

Solution:

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

Solution:

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.

Solution:

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

Solution:

2pts (b) 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.

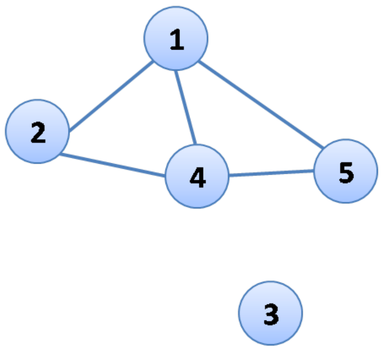
Solution:

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?

Solution:

3. **Graph representation**

1pt (a) Show the adjacency matrix that represents the graph drawn below (use the format shown in the lecture notes):



3pts (b) Recall the *adjacency list* representation of a graph from class:

```
typedef unsigned int vertex;
typedef struct graph_header* graph;
typedef struct adjlist_node adjlist;
struct adjlist_node {
   vertex vert;
   adjlist *next;
};
struct graph_header {
   unsigned int size;
   adjlist *adj[];
};
```

Extend the graph interface with a function `graph_countedges(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_countedges(graph* G, vertex v) {







 }
```

1pt (c) 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.

Solution: $O($______________________$)$

3pts (d) Recall 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 must use only the types and functions provided in `graph.h`. You may use the fact that `vertex` is an integer type, and that it is the same type returned by `graph_size`.

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









 }
```

1pt (e) 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.

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