ECE 468 & 573
Problem Set 3: Common sub-expression elimination and register allocation

For the following problems, consider the following piece of three-address code:

```
1.  A = 7;
2.  B = A + 2;
3.  C = A + B;
4.  D = C + B;
5.  A = D + C;
6.  B = D + C;
7.  E = A + B;
8.  F = D + C;
9.  G = E + F;
```

1. Show the result of performing Common Subexpression Elimination (CSE) on the above code.

   Answer:

   ```
   1.  A = 7;
   2.  B = A + 2;
   3.  C = A + B;
   4.  D = C + B;
   5.  A = D + C;
   6.  B = A;
   7.  E = A + B;
   8.  F = A;
   9.  G = E + F;
   ```

2. Suppose E and C were aliased. How would that change the results of CSE?

   Answer:
   If E and C were aliased, then we would not be able to rewrite instruction 8; the assignment to E would change the value of C and so kill the expression "D + C".

3. If we were doing top-down register allocation, which variables would be put in registers? Assume a machine with 3 registers.

   Answer
   Without consider keeping registers for spilling, A is used 5 times, B is used 5 times and C is used 5 times, so they would be in registers, while the other variables would not be.

4. For each instruction, show which variables are live *immediately after the instruction.*

Answer:

Instruction	**Live after**
1	A
2	A, B
3	B, C
4	C, D
5	A, C, D
6	A, B, C, D
7	C, D, E
8	E, F
9	None

5. How many registers would be needed to perform register allocation with no spilling?

   Answer
   Four registers (to hold A, B, C & D after instruction 6).

6. Top down register allocation is inefficient for the above code, as there are some variables that could safely be assigned to the same register. What are they?

   Answer:
   Either A or B could be put in the same register as E. Either A, B, C and D could be put in the same register as F.

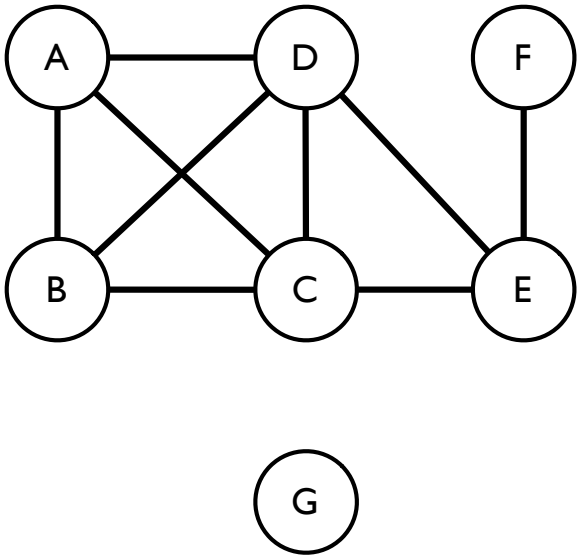
7. Perform bottom-up register allocation on the code for a machine with three registers.. Show what code would be generated for each 3AC instruction. When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the register holding a value that will be used farthest in the future (in case of a tie, choose the lowest-numbered register).

   Answer:

Instruction	R1	R2	R3	Code
A = 7;	A			`R1 = 7`
B = A + 2;	A	B		`R2 = R1 + 2`
C = A + B;	C	B		`ST R1, A` (free) `R1 = R1 + R2`
D = C + B;	C	D		`ST R2, B` (free) `R2 = R1 + R2`
A = D + C;	C	D	A	`R3 = R2 + R1`
B = D + C;	B	D	A	`ST R1, C` (spill) `R1 = R2 + R1`
E = A + B;	E	D		`ST R1, B` (free) `ST R3, A` (free) `R1 = R3 + R1`
F = D + C;	E	F		`LD C, R3` (unspill) `ST R2, D` (free) `R2 = R2 + R3` Note: no need to ST C on free because C is not dirty
G = E + F;				`ST R1, E` (free) `ST R2, F` (free) `R1 = R1 + R2` `ST R1, G` (free)

8. Draw the interference graph for the code.

Answer:



9. (ECE 573 only) Perform register allocation via graph coloring for the code. If you need to spill, use the code-rewriting approach described in the notes.

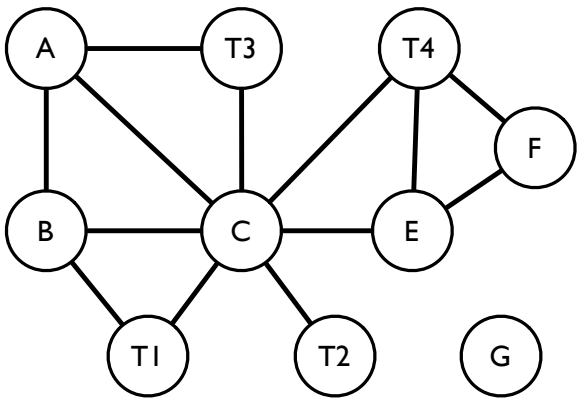
Answer:
In our first pass, we build the following stack, with starred variables marking the variables that might be spilled:

$$G, F, E, D^*, C, B, A$$

When we begin to color the stack, we realize that D will not be colorable. So we rewrite the code to use explicit moves for uses of D:

```
1.  A = 7;
2.  B = A + 2;
3.  C = A + B;
4.  T1 = C + B;
4': ST T1, D;
5': LD D, T2;
5.  A = T2 + C;
6': LD D, T3;
6.  B = T3 + C;
7.  E = A + B;
8': LD D, T4
8.  F = T4 + C;
9.  G = E + F;
```

We re-perform liveness analysis, and get the new interference graph:



When we simplify the graph, we create the following stack (there are many options):

$$G, F, T4, E, T2, T1, T3, C, A, B$$

which requires no spilling. They key, when we color this graph again, is that T1 and T3 will be given different colors (T1 will have the same color as A, and T3 will have the same color as B) and so will be assigned to different registers. Here is one possible assignment:

Register	**Variables**
R1	A, T1, T2, T4
R2	B, T3, E
R3	C, G, F

And here is the generated code:

```
R1 = 7; //A = 7
R2 = R1 + 2; //B = A + 2
R3 = R1 + R2; //C + A + B
//ST R1, A; //see Note 1
R1 = R3 + R2; //T1 = C + B
ST R1, D; //ST T1, D
LD D, R1; //LD D, T2 (a peephole optimization can eliminate this)
R1 = R1 + R3; //A = T2 + C
//ST R2, B; //See Note 1
LD D, R2; //LD D, T3
R2 = R2 + R3; //B = T3 + C
//ST R2, B; //see Note 2
R2 = R2 + R1; //E = A + B
//ST R1, A; //see Note 2
LD D, R1; //LD D, T4
//ST R3, C; //see Note 2
R3 = R1 + R3; //F = T4 + C
//ST R3, F; //see Note 2
R3 = R2 + R3; //G = E + F
```

There are some instructions in the generated code above that have been commented out. These are instructions that seem like they *should* be necessary, but actually aren't. Here is why.

Note 1: It is not necessary to emit store instructions for A and B at these places because we know that A and B will be redefined before they are used again in this basic block.

Note 2: Here, even though A, B, C and F are *not* redefined again, we do not need to emit store instructions. This is because our liveness analysis is *global*; because neither A, B, C or F are live at the places we might need to store them, we know not only

that their values won't be used later in this basic block, we know that their values won't be used anywhere later in the *program*. Thus, we do not need to store them.

Even if we eliminate Note 1-style stores from the bottom-up register allocation code generated in problem 7, we find that the global register allocation code is much shorter. This is because our global liveness data gives us very good information about whether stores are necessary or not.