1. (20 points) Consider the following interference graph:

![Interference Graph](image.png)

a. Give a machine with 3 registers, is it possible to find an allocation for the graph above without spilling? You may answer one of yes, no, “I don’t know”. If yes, give a coloring.

   Yes.
   A = 2
   B = 3
   C = 1
   D = 3
   E = 2
   F = 2
   G = 1
   H = 3

b. How does the “improved Chaitin’s algorithm” behave when given this input? Recall that the improved Chaitin’s algorithm keeps variables that may need to be spilled on the “stack”, and tries to assign a register to them in the assignment phase.

   It is possible that it does not find a coloring - it depends on which node it picks to spill.
   After eliminating E, H and G it is left with ABCDF. If it picks C you can assign 123 to ADDF and fail to assign C.
2. (20 points) Consider the following control flow graph:

And here is its dominator tree.

What are the natural loops in this flow graph? (Show intermediate steps for partial credit.)

There are 3 retreating edges, only 1 is also a back-edge: K->M
6. (20 points) What is the best software pipelined schedule that you can create for the following precedence graph. This machine has two resources, R0, R1. The edges are labeled by the \(<\text{iteration difference}, \text{latency}>\); each node is represented by its respective resource reservation table. If an instruction uses resource R_j in cycle i after it has been issued, \((i = 0, 1)\), it is indicated by a black square in row i and column j in the resource reservation table. For example, node 4 uses R0 in cycle 0, and R1 in cycle 1 after the node is issued.

![Graph Image]

a. What is the bound of the initiation interval?

- Resource constraints: 3
- Precedence constraints: 3

b. Find the best software pipelined schedule. What is the initiation interval of your schedule? Show the schedule of one iteration.

1: 1 5
2: - 3
3: 2 -
4: 4 2
4. (20 points) Show the points-to result obtained with a flow-insensitive, context-insensitive inclusion-based points-to analysis. The allocation sites are named $a_1, a_2, \ldots, a_6$.

```java
main ()
{
    List list1 = new List();  (a1)
    List list2 = new List();  (a2)
    List list3 = new List();  (a3)
    C elem1 = new C();        (a4)
    D elem2 = new D();        (a5)
    list1.add (elem1);        (a6)
    list3.add (elem2);        (a7)
    list2.add (list3);        (a8)
}
```

```java
public class Node {
    public Node next;
    public Object elem;
}
```

```java
public class List {
    private Node head;
    void add (Object item) {
        Node t = new Node();  (a6)
        t.next = head;
        t.elem = item;
        this.head = t;
    }
```

Name the objects pointed to by:

a. $a_1$.head
   
   $a_6$

b. $a_2$.head
   
   $a_6$

c. $a_3$.head
   
   $a_6$

d. $a_6$.next
   
   $a_6$

e. $a_6$.elem
   
   $a_3, a_4, a_5$
2. (20 points) Briefly give one important advantage and disadvantage for each of the following styles of garbage collection.

a. Reference counting
   Advantage:
   incremental, immediate
   Disadvantage:
   can't handle cycles, must instrument code

b. Copying garbage collection
   Advantage:
   compression, sweep amount proportional to amount of stuff on heap
   Disadvantage:
   shadow space

c. Generational garbage collection
   Advantage:
   things die young
   Disadvantage:
   heavy intergenerational pointer structure to support collection

d. Incremental garbage collection
   Advantage:
   don't stop the world.
   Disadvantage:
   any given point you'll have garbage you don't know about that was previously marked as clean
5. Assignments in a Java program have implicit type cast operations. For assignment statement

```
A x = y;
```

object y needs to be checked dynamically if it may be type-cast to A. More specifically, the compiled Java bytecode for the above assignment will include a bytecode called `CHECKCAST` that performs the following operation:

```
if (! type(y) is subclass of type(x)) {
    raise exception
}
```

a. We can optimize a `CHECKCAST` bytecode away at compile time if we know that it cannot raise an exception. For the following program, identify all the assignment statements that need no `CHECKCAST` bytecodes. Assume that class A is a super class of B and B is a super class of C.

```
A a1 = x;               Keep
A a2 = x;               Eliminate
C c1 = x;               Keep
B b1 = x;               Eliminate
```
b. For the following control flow graph, find all assignments whose `CHECKCAST` bytecode can be optimized away. The class hierarchy is the same as the one above.

```
entry

x = createObj();

A a1 = x;
A a2 = x;
B b1 = x;
C c = x;
B b3 = x;
A a3 = x;
```

exit

c. Design a data flow analysis algorithm to remove unnecessary type cast checks. Fill in a table like the one in Problem 4. State clearly which assignments need no `CHECKCAST` bytecodes.
5. [10 points] a. [1 point] Following two assignments do not need CHECKCAST

   \[
   \begin{align*}
   A \quad & a2 = x; \\
   B \quad & b1 = x;
   \end{align*}
   \]

b. [1 point] Two basic blocks containing following assignments do not need CHECKCAST

   \[
   \begin{align*}
   A \quad & a3 = x; \\
   B \quad & b3 = x;
   \end{align*}
   \]

c. [8 points] Our dataflow analysis is a forward analysis. Similar to Q4, we define semi-lattice per variable, and the entire semi-lattice is the product of them. The values in semi-lattice for a variable are types for which the CHECKCAST is applied. At the top of the semi-lattice is UNDEF and at the bottom is \texttt{java.lang.Object}. Right below the UNDEF are the most specific classes (or most subclasses) and below them are the superclasses of the classes. For example, following left is the semi-lattice for a variable \(x\), and following right is the class hierarchy of all the types that the variable \(x\) can have.

\[
\begin{array}{c}
\text{UNDEF} \\
C & \\
& Y \\
& B \\
& A \\
& \text{java.lang.Object}
\end{array}
\quad
\begin{array}{c}
\text{java.lang.Object} \\
& A \\
& B \\
& C \\
& \text{X} \\
& \text{Y} \\
& \text{java.lang.Object}
\end{array}
\]

As the semi-lattice indicates, the meet operator is finding the first common superclass. Let \(IN_x[B]\) be the semi-lattice value associated with the beginning of block \(B\) and variable \(x\). Then the transfer functions with respect to block \(B\) and variable \(x\) are defined as follows. (for the sake of simplicity we assume that basic blocks consist of single statement)

\[
\begin{align*}
OUT_x[B] &= IN_x[B] & \text{if there is no assignment containing } x \text{ in } B \\
OUT_x[B] &= \text{return type of foo()} & \text{if } x = \text{foo()} \text{ in } B \\
OUT_x[B] &= IN_y[B] & \text{if } x = y \text{ in } B \\
OUT_x[B] &= IN_x[B] & \text{if } Cc = (C)x \text{ in } B \text{ and } IN_x[B] \text{ is a subclass of } C \\
OUT_x[B] &= C & \text{if } Cc = (C)x \text{ in } B \text{ and } IN_x[B] \text{ is not a subclass of } C
\end{align*}
\]
This dataflow framework is monotone and distributive. This converges since it is monotone and has a finite height. Once the analysis is finished, CHECKCAST of variable $x$ into a class $C$ in a basic block $B$ may be removed if $IN_x[B]$ is subclass of the $C$.

Grading: Basic dataflow design (semi-lattice and transfer functions) has 4 points and the analysis of it (e.g., monotonicity) has 4 points.

Basic dataflow design: When the main part of dataflow analysis (semi-lattice and transfer functions) is not reasonably close to what the problem asked, 0 point is given. When the description of transfer functions is too vague or has many errors, 0 point is given. When there is a minor mistake in transfer functions or semi-lattice, 1 point is taken out.

Analysis of the dataflow: 1 point is taken out for incorrect description of boundary condition, initial interior values, monotonicity, distributivity or convergence. For incorrect description of which CHECKCAST bytecode can be safely removed, 1 point is taken out.