Good Programming Guidelines

1. Programs should be modular.
   If you design nice modules the code is reusable for other programs such as Assignment #2 or a program that implements another graph algorithm (finding a minimum cut set, finding a maximum independent set). Do NOT put everything in the main.
2. A routine that has a purpose should do the whole task. Examples: read_graph should read both n and the graph, find_min_dom_set (Assignment #2) should initialize its own data structures (at level 0).
3. A routine with a purpose should do ONLY its task.
   For example: read_graph should not print a graph or read in a dominating set. check_dom_set should not read or print a dominating set.

4. Use meaningful variable names.

For example, check_graph is better than check for the function that checks the graph.
Integer loop indices are usually i, j, k (not a,b).
But you could use u, v, w to index through vertices.

5. Avoid global variables.

If everything for a module is in the module it is easier to argue that the module is correct than if what happens depends partly on changes to global variables for which you have to search the whole entire program to see what is happening.
Verbose is not too bad as a global variable with the way we run this program but maybe in future you might want to use verbose printing sometimes but not other times. Then it should be a parameter.

6. Do not initialize variables to irrelevant values that are not used in the program.

```
int i = 0;  // Initialization not needed.
int n = 0; // Initialization not needed.

// Value of n is read in.

if (scanf(“%d”, &n)!=1) exit(0);

// Value of i is initialized in for loop.
for (i=0; i < n; i++)
{
}
```

It makes is harder for someone to check your code.

7. When you are done, re-read the specs to make sure have followed them.

   Use standard input and output.
   Do not use malloc.

8. Do not use provided types if they make the code less efficient (e.g. vectors). This is a course about algorithm analysis and we are trying to implement tasks efficiently.

   I would prefer it if you keep things simple so that we can write code that is as fast as possible and because it is more likely you understand the consequences of what you are doing.

9. If you are trying to add complete error checking, never ignore the return value of scanf.

On my computer:

```
int n, ok;
ok= scanf(“%d”, &n);
```

ok is equal to:

1 if n is read in successfully

0 if read fails (maybe because there is some text in the input stream that cannot be parsed as an integer)

-1 if we have reached the end of the data

Note: any non-zero value including -1 evaluates to true in C/C++ [e.g. if (ok) { … }].

10. Do not use complicated code to parse strings
    when you want to read in one integer
    when you can write much more elegant code that
    just uses scanf.

11. Make sure you test your programs extensively
    in.txt was just a sample file that did not contain
    instances of everything that could go wrong.

12. If you are given sample output files, make sure
    your output is the same (or matches within the
    guidelines for the specifications).

13. Use lots of comments!

At top of program:

How do I type in input for the program?
What does it compute for me?
What are the data structures?

Adjacency matrix or adjacency list?

Is the dominating set {1, 2, 5} stored as
dom[0]= 1, dom[1]=2, dom[2]= 5 and dom_size =3

or dom[i]= 0 except for dom[1]=1 dom[2]=1 dom[5]=1?

12. Every function should have comments.

What are the inputs to the function?
What does the function compute?
If there is a return value, what does it mean?
What are the data structures?

13. Don't store items with different meanings in the same array.

For example:
G[u][0]= degree of vertex u = d
G[1 … d] = neighbours of G
We used to do things like that when we programmed in assembly language sometimes.
Use:
int degree[NMAX]; int G[NMAX][NMAX];
(allows room for loops for dominating set alg.)

13. If you borrow code from somewhere (e.g. my slides) and do not acknowledge this then it is a serious academic offense (plagiarism).

To be safe:
Acknowledge at the top of the program.
Mention it again at the top of the function (read_graph or print_graph for example) you are copying.

14. Use meaningful loop constructs.
If your task is to read a graph until you run out of graphs in the input then your main should have:
while (read_graph(…))
not
while (true) or
other types of loops.

15. Your main should give a high-level outline of what your program does.

```
while we can read a graph
    check the graph
    if the graph is bad then terminate
    read a dominating set
    check the dominating set
    print results (terse or verbose)
end while
```

15. Some errors should abort the read process:

n or k > NMAX
If we try to continue then the program will crash because we do not have enough space to store the graph/dominating set.

Invalid values for the graph:
n, degree or neighbour number out of range.
Program specs tell us to terminate.

Others can be tested for separately:

This application asks for simple graphs as input.
If we separate out checking for loops, multiple edges, ensuring the graph is symmetric (if G has (u,v) then it also has (v,u)) then read_graph can be reused when our requirements change.

16. If you want to use C++ or java objects think carefully about what these should be.

A graph might have:

n= number of vertices
m= number of edges (optional)
degree of each vertex (optional)
adjacency matrix or adjacency list
or other way to represent edges
(for example a list of edges)

An array:

is_dominated[0 .. (n-1)] is specific to the dominating set problem so it does not belong in a graph object.

This allows you to reuse the graph class/object for other problems.