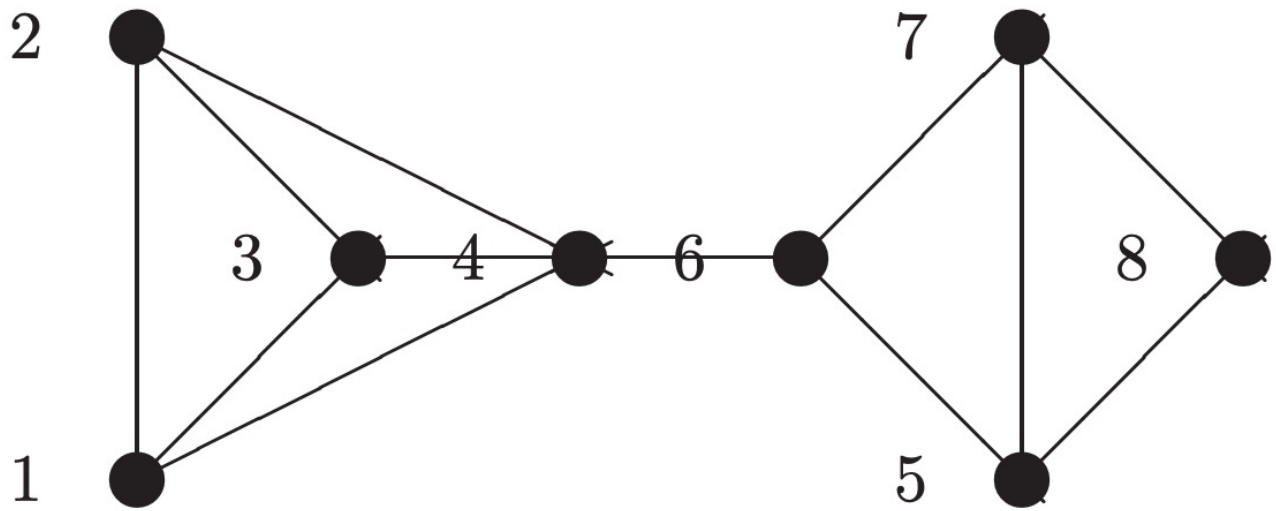
R-13.5 Hint: Consider the way that edges are represented.

Solution: Inserting an edges runs in $O(1)$ time since it is simply inserting an element for $e = (i, j)$ into the appropriate two locations, $A[i, j]$ and $A[j, i]$. To insert a vertex, on the other hand, we must construct a new version of the entire array, A. Clearly this will take $O(n^2)$ time.

R-13.6 Solution:

a. A drawing of graph G is given below:



b. A DFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.

c. A BFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.

R-13.7 Hint: Review the combinatorial properties for graphs.

Solution:

a. The adjacency list structure is preferable. Indeed, the adjacency matrix structure wastes a lot of space. It allocates entries for 100,000,000 edges while the graph has only 20,000 edges.
b. In general, both structures work well in this case. Regarding the space requirement, there is no clear winner. Note that the exact space usage of the two structures depends on the implementation details. The adjacency matrix structure is much better for operation areAdjacent, while the adjacency list structure is much better for operations insertVertex and removeVertex.
c. The adjacency matrix structure is preferable. Indeed, it supports operation areAdjacent in $O(1)$ time, irrespectively of the number of vertices or edges.

C-13.15 Hint: There has to be a good traversal that does this.

Solution: Perform a DFS on G. G will have at least two cycles if and only if this DFS encounters at least two back edges.

C-13.16 Hint: Start with a DFS.

Solution: Perform a DFS on G. G will have at least two cycles if and only if this DFS encounters at least two back edges. If there are more than two back edges, then G has more than two cycles. So, for the first back edge, mark all the edges in the cycle determined by this edge and the DFS tree as "used." Then, for the second back edge, examine each of the edges in the cycle determined by this edge. If non of them is marked "used," then these two cycles are disjoint. If one of them is "used," then there are no two disjoint cycles.

Oppgave 2 (Eulerkrets) I forelesningen definerte vi en Eulerkrets som en vei som starter og slutter i samme node og er innom alle kantene i G nøyaktig en gang. Vi så på problemet *Eulerkrets*:

Eulerkrets	
Instans:	En graf G
Spørsmål:	Har G en Eulerkrets?

Lag en algoritme som løser Eulerkrets. Output skal være *true/false*. Kjører algoritmen i polynomiell tid? Begrunn svaret.

Hint: Det finnes en egenskap som har med graden til nodene i G å gjøre, som nøyaktig bestemmer om G har en Eulerkrets eller ikke.

Løsningsforslag Sjekk om G er sammenhengende. Hvis G inneholder minst to komponenter som inneholder kanter, kan det umulig finnes en Eulerkrets. Sjekk graden til alle nodene i G. Hvis graden til hver eneste node er et partall innholder G en Eulerkrets. Hvis én eller flere noder har odde grad inneholder G ikke en Eulerkrets.

Algoritmen bruker polynomiell tid, siden å sjekke om G er sammenhengende kan gjøres med én traversering og fordi det å sjekke graden til hver node kun krever å gå gjennom nabo-lista til hver node.

Oppgave 3 (Hamiltonsykel) Vi så også på problemet *Hamiltonsykel*.

Hamiltonsykel	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsykel?

En Hamiltonsykel i en en graf G er en sykel som inneholder hver node nøyaktig en gang.

Lag en algoritme som løser Hamiltonsykel. Output skal være *true*/*false*.

Hint: Enn så lenge har ingen klart å finne en algoritme som løser dette problemet i polynomiell tid, så ikke ta det så tungt om også din algoritme bruker eksponensiell tid.

Løsningsforslag Anta at G har n noder. Generer alle $n!$ permutasjoner av nodene. For hver permutasjon: Sjekk at det går en kant mellom nodene som kommer etter hverandre i permutasjonen Sjekk at det er en kant mellom siste og første node.