(Exercise 10) This week you will revise the Ford-Fulkerson algorithm and at the same time think about implementing it on a computer. It is not a simple algorithm to write so I don't expect you to complete it. Various useful bits of code and elements of performing the algorithm are broken up for this week's exercise. You can choose which parts to tackle in whatever order you like. Although some parts are advised before others.

This week it would be great to see you use a variety of the functions you've been using so far this term. Here's a reminder of some of the clever things I've seen used this term which are useful for the exercises this week:

- For a random vector v <−**sample**(1:9), what does v > 4 look like?
- What does the **as.integer** function do?
- Type ?**union** for a the help file relating to a number of popular *set functions*.
- In R you can perform a **for** loop not only over a range like 1:10 but also over a set, via syntax like this:

```
W <- c(1, 2, 7, 8)
X <- c(3, 4)
for (i in W) {
    for (j in X) {
        print(1000 * i + j)
    }
}
```

- For vectors, and even matrices you can use the **all**() function to find out if every element satisfies some property.
- What does **matrix**(0, **nrow**=7, **ncol**=7) create?
- The **which** function, when applied to matrices, doesn't return something entirely helpful. There is a parameter called arr.ind you can set to TRUE which returns answers in terms of co-ordinates. e.g. try the following

```
M <- matrix(sample(1:16), nrow = 4)
M
which(M > 8)
which(M > 8, arr.ind = TRUE)
```

- If you desire the 1st, 3rd and 7th rows from a matrix M, then if B <−**c**(1, 3, 7) then M[B,] is what you want. Similarly, M[, 2] is the second column of M. You can even combine these ideas with conditions. Have a look at these outputs:

```
M <- matrix(1:24, ncol = 2)
M[, 1] %% 3
M[, 1] %% 3 == 0
M[M[, 1] %% 3 == 0, ]
M[M[, 1] %% 3 == 0, ]
```

Exercise 10 (Hunayn)**.** The Ford-Fulkerson algorithm

For all tasks below:

- You may set n=7, (if you like) or keep n unknown if you are more adventurous.
- You can also assume all flows are from vertex 1 to vertex 7 (or n).
- Code should not, however, exploit any specific known values of other parameters/variables, and so should work for general M, S, c_{ij}, and flows – not just the for M, S, etc. . . with which you are working.
- The idea behind writing general code is that it will also work when someone comes along with a different starting graph $G = (V, E)$ without any changes to the code.

You should all attempt these three parts: Code up the problem from the start of the chapter (with the proposed flow) as follows:

- Create the capacity matrix M, where M[i, j]=c_{ij}, (0 if no edge exists). e.g M[1, 2]=12, M[1, 3]=10, etc. . . . (Hint: Start with a matrix full of zeroes!)
- Create a flow matrix representing a particular flow. By hand type the entries of the flow drawn during lectures onto the diagram in the notes.
- Create an *adjacency matrix*, A: entry (i, j) is 1 if the edge (i, j) edge exists in the graph, 0 if not. You should use the variable M as a starting point. (Not by hand! There are short solutions to this.)

You should select *some* of the below exercises:

(A) Create a list of all the edges in the network, using the capacity or adjacency matrix. (Not by hand! Hints in accompanying text.)

(B) Write a function to check a flow is feasible (meets all the constraints). This code should take M, and a flow matrix as the starting point.

(C) (i) Given n, and a given subset S of V, create the complement S^c.

(ii) Create code to list all the edges between a given set S, and the complement of S. (You may assume you have a capacity matrix or adjacency matrix, and are given S)

(D) For a given list of edges, and a given flow matrix, identify if there are any edges along which flow could be increased.

You should all comment briefly how the parts you attempted might be useful in a full version of the Ford-Fulkerson algorithm. Describe also the elements of the algorithm that haven't been discussed.