When working with graphs we often need to explore or search the graph, starting from a specified vertex. This may occur in the context of finding shortest routes in a network, or determining structural properties of the graph.

There are two standard methods for exploring a graph: breadth-first search and depth-first search. We will examine both of these, and discuss how the data structures we choose can affect the efficiency of these algorithms.

**Breadth-First Search**

The idea of BFS is to explore the graph in levels. The first level consists of the start vertex, then all of its neighbours are on the next level, then all of their neighbours, etc. We keep track of which vertices we have seen so that we don’t go around in circles (or cycles, ha ha).

Using the graph shown above, if we start at vertex E, the set of vertices on the first level is just \{E\}. Its neighbours are G, A and F – we do not specify an order for these vertices, and the set of vertices on the next level is \{F,G,A\}. To get the next level, we take one of these vertices … say F … and add all of its neighbours that we have not already seen to the set for the next level. This gives us \{B\}. Then we take another vertex from the current level … say G … and add its neighbours (again, only the ones we haven’t seen) to the new set, which is now \{B,D\}. And then we take the next (and in this case last) vertex from the current set, which is A. A has no new neighbours to contribute, so the next level consists of \{B,D\}. The final level of the exploration consists of just \{C\}.
We often keep track of the "neighbour" relations that we are using – this lets us build a BFS tree for the graph. The exploration above would give this tree:

![Diagram of BFS tree]

Note that the edge weights from the graph are not used in this algorithm.

It is important to see that this will always generate a tree – the rule that we only go to each vertex once guarantees this. It is also important to see that the tree is not unique. To see this for yourself, draw the tree that results from starting at E (as above) but then choosing A as the first vertex on the next level to explore further.

Even though we can get several different trees by applying BFS and starting at E, you will see that they have something in common: the sets of vertices at each level are always the same. Make sure you can explain why this is true.

We should also recognize that the number of levels in the BFS tree can vary depending on the vertex where we start. To convince yourself of this, find a vertex in the graph shown above which gives BFS trees with a different number of levels than the BFS trees that start at E.
Let’s look at pseudo-code for the BFS algorithm. The first thing we need to do is decide how to keep track of the vertices at the “current” level while we build the set of vertices at the “next” level. Fortunately this is very easy – we can just use a queue. Each time we find “new” neighbours for a vertex at the current level, we add them to the end of the queue. This ensures that we will work through all vertices at the current level before we start on the next level.

```python
def BFS(v):
    Q = new queue
    mark v “visited”  # this can be done with a Boolean attribute
                      # attached to the vertex
    Q.append(v)
    while Q is not empty:
        x = Q.remove_first()
        for each neighbour y of x:
            if y is not “visited”:
                mark y “visited”
                Q.append(y)
```

You should work through this for the graph in our examples to see how it works. You will observe that each vertex is added to Q exactly once and removed from Q exactly once.

If we want to build the tree, every time we mark a neighbour of x as visited, we need to include the edge from x to y in our tree. We can represent the tree by the list of edges we have used.

The tree that we generate through BFS has a useful property: the graph is connected if and only if a BFS tree of it contains all n vertices. This gives us a very practical method to test a graph for connectivity. When we look at other graph algorithms we often start with “Assume G is connected”. We can make that assumption because now we have a simple way to test it – and if the graph is not connected we can either deal with each piece separately, or add some more edges until it is connected.

Now we can address the crucial question: what is the complexity of this algorithm? First, we can assume that all queue operations are in O(1). Second, we can see that (assuming the graph is connected), all n vertices go onto Q and come off Q again. Thus the while loop executes n times. The tricky part of the analysis is the for loop. We need to find or build a list of the neighbours of vertex x. The time required to do this depends on the graph representation we are using.

If we are using an adjacency matrix, this will take O(n) time, even if vertex x has very few neighbours … because we have to look at an entire row of the matrix A. This gives our BFS algorithm O(n²) complexity.
But if we are using adjacency lists, we don’t need to do any work to obtain the neighbours of x. The structure simply *gives us* the list of neighbours, and all we need to do is iterate through it.

This means that the inner loop (the *for* loop) will potentially execute a different number of times for each vertex – this may make it seem impossible to compute the complexity. But here is one idea (not a very good one as it turns out): each vertex has at most \( n-1 \) neighbours, so this inner loop will execute at most \( n-1 \) times for each vertex … this gives us \( O(n^2) \) complexity again.

But a bit more thought lets us get a more precise bound. Once we have the list of neighbours of x, the “if y is not visited …” test executes once for each neighbour of x. And since each vertex gets to take the role of x exactly once, the total number of times this test is executed is exactly the sum of the lengths of all the adjacency lists.

And we know what that is! The length of each vertex’s adjacency list is the degree of the vertex, so we just need to sum the degrees. In CISC-203 we learned that for any graph,

\[
\sum_{v \in V} d(v) = 2m
\]

where \( d(v) \) is the degree of vertex \( v \) and \( m \) is the number of edges in the graph.

So this test (which is the most frequently executed line in the algorithm) executes \( O(m) \) times.

In summary, when we use adjacency lists, the complexity of BFS is \( O(m) \).

(Some authors write this as \( O(n+m) \) and this is technically correct … but I am assuming the graph is at least potentially connected, which means \( m \geq n-1 \). This makes it unnecessary to include the \( n \) in the complexity for our analysis.)

Now if the graph is sparse, we know \( m \) is in \( O(n) \) … which means that for sparse graphs represented by adjacency lists, BFS runs in \( O(n) \) time. Remember that when we use an adjacency matrix BFS runs in \( O(n^2) \) time regardless of whether the graph is sparse or not.

Note that on dense graphs, where \( m \in \Omega(n^2) \), both representations of the graph will give BFS the same complexity: \( O(n^2) \)

The bottom line is that for this algorithm, adjacency lists are never worse than the adjacency matrix, and they are sometimes much better.