Data Structures:
We shall regard integers, real numbers, bits as well as more complicated objects such as lists and sets as primitive data structures. Recall that a list is just a sequence of arbitrary elements.
List $q := [x_1, x_2, \ldots, x_n]$.
x_1 is called the head of the list.
x_n is called the tail of the list.
$n = |q|$ is the size of the list.
We denote by $\circ$ the concatenation operation. Thus $q \circ r$ is the list that results from concatenating the list q with the list r.
The operations on lists that are especially important for our purposes are:

- head(q). return(x_1).
- push(x, q). $q := [x] \circ q$.
- pop(q). $q := [x_2, \ldots, x_n]$, return(x_1).
- inject(x, q). $q := q\&[x]$.
- eject(q). $q := [x_1, x_2, \ldots, x_{n-1}]$, return($x_n$).

A *stack* is a list that supports operations head, push, pop.
A *queue* is a list that supports operations head, inject, pop.
A *deque* supports all these operations.
Note that we can implement lists either by arrays or using pointers as the usual linked lists. In either case, each of the above operations can be implemented in *a constant number of steps.* In analysing algorithms using list operations, we shall therefore consider each operation as a single step.
For the rest of the lecture, we will review the procedure mergesort. The input is a non-empty list of n numbers, and the output is a list of the given numbers sorted in increasing order. The main data structure used by the algorithm will be a queue. We will assume that each queue operation takes 1 step, and that each comparison (is $x > y$?) takes 1 step. We will show that mergesort takes $O(n \log n)$ steps to sort a sequence of n numbers.
The procedure mergesort relies on a function merge which takes as input two *sorted* (in increasing order) lists of numbers and outputs a single sorted list containing all the given numbers (with repitition).

```
function merge (s,t)
    list s , t
    if s = [ ] then return t
      else if t = [ ] then return s
      else if s(1) ≤ t(1) then return push(pop(s),merge(s, t))
                          else return push(pop(t),merge(s, t))
end merge
```

The correctness of the function merge follows from the following fact: the smallest number in the input is either $s(1)$ or $t(1)$, and must be the first number in the output list. The rest of the output list is just the list obtained by merging s and t *after* deleting that smallest number.

The number of steps for each invocation of function merge is $O(1)$ steps. Since each recursive invocation of merge removes an element from either s or t, it follows that function merge halts in $O(|s| + |t|)$ steps.

```
function mergesort (s)
    list s, q
    q = [ ]
    for x ∈ s do:
        {q := inject(q, [x])}
    while |q| ≥ 2 do
        {inject(merge(pop(q),pop(q)), q)}
    return q(1)
end mergesort
```

The correctness of the algorithm follows easily from the fact that we start with sorted lists (of length 1 each), and merge them in pairs to get longer and longer sorted lists, until only one list remains. To analyze the running time of this algorithm, consider the first element to be sorted, $s(1)$. During the execution of mergesort, $s(1)$ will be part of various elements of the queue q. Whenever the element of q that currently contains $s(1)$ is popped, we say that a new *phase* of the execution of mergesort begins. Then we claim that the total time per phase is $O(n)$. This is because each phase just consists of pairwise merges of disjoint lists in the queue. Each such merge takes time proportional to the sum of the lengths of the lists, and the sum of the lengths of all the lists in q is n. On the other hand, the number of lists is halved in each phase, and therefore the number of phases is at most $\log n$. Therefore the total running time of mergesort is $O(n \log n)$.

1 Graphs

Coming up with a simple, precise formulation of a computational problem is often a prerequisite to writing a computer program for solving the problem. Many computational problems are best stated in terms of *graphs:* A directed graph $G(V, E)$ consists of a finite set of vertices V and a set of (directed) edges or arcs E. An arc is an ordered pair of *distinct* vertices (v, w) and is usually indicated by drawing a line between v and w, with an arrow pointing towards w. Stated in mathematical terms, a directed graph $G(V, E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V. Thus, there can be no multiple arcs from a node to another —but there can very well be two arcs, one going from node a to node b, and another from node b to node a. An *undirected graph* is a special kind of directed graph, such that, whenever $(u, v) \in E$, it must be the case that $u \neq v$, and $(v, u) \in E$. Thus, since the directions of the edges are unimportant in undirected graphs, an undirected graph $G(V, E)$ consists of a finite set of vertices V, and a set of edges E, each of which is an unordered pair of vertices $\{u, v\}$, denoted by a line joining u with v.

Graphs are useful for modeling a diverse number of situations. For example, the vertices of a graph might represent cities, and edges might represent highways that connect them. In this case, each edge might also have an associated length. Alternatively, an edge might represent a flight from one city to another, and each edge might have a weight which represents the cost of the flight. The typical problem in this context is computing shortest paths: given that you wish to travel from city X to city Y, what is the shortest path (or the cheapest flight schedule). There are very efficient algorithms for solving these problems. A different kind of problem —

the traveling salesman problem — is very hard to solve. Suppose a traveling salesman wishes to visit each city exactly once and return to his starting point, in which order should he visit the cities to minimize the total distance travelled? This is an example of an NP-complete problem, and one we will study towards the end of this course.

A different context in which graphs play a critical modeling role is in networks of pipes or communication links. These can, in general, be modeled by directed graphs with capacities on the edges. A directed edge from u to v with capacity c might represent a pipeline that can carry a flow of at most c units of oil per unit time from u to v. A typical problem in this context is the max-flow problem: given a network of pipes modeled by a directed graph with capacities on the edges, and two special vertices — a source s and a sink t — what is the maximum rate at which oil can be transported from s to t over this network of pipes? There are ingenious techniques for solving these types of flow problems, and we shall see some of them later in this class.

In all the cases mentioned above, the vertices and edges of the graph represented something quite concrete such as cities and highways. Often, graphs will be used to represent more abstract relationships. For example, the vertices of a graph might represent tasks, and the edges might represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. There is a very fast algorithm for solving this problem, that we will see shortly.

We usually denote the number of nodes of a graph by n, and its number of edges by e. e is always less than or equal to n^2, since this is the number of all ordered pairs of n nodes —for undirected graphs, this upper bound is $\frac{n(n-1)}{2}$. On the other hand, it is reasonable to assume that e is not much smaller than n; for example, if the graph has no *isolated nodes*, that is, if each node has at least one edge into or out of it, then $n \geq \frac{n}{2}$. Hence, e ranges roughly between n and n^2. Graphs that have about n^2 edges —that is, nearly as many as they could possibly have— are called *dense*. Graphs with far fewer than n^2 edges are called *sparse*. Planar graphs (graphs that can be drawn on a sheet of paper without crossings of edges) are always sparse, having $O(n)$ edges.

2 Representing graphs on the computer

One common representation for a graph $G(V, E)$ is the *adjacency matrix*. Suppose $V = \{1, \cdots, n\}$. The adjacency matrix for $G(V, E)$ is an $n \times n$ matrix A, where $a_{i,j} = 1$ if $(i, j) \in E$ and $a_{i,j} = 0$ otherwise. The advantage of the adjacency matrix representation is that it takes constant time (just one memory access) to determine whether or not there is an edge between any two given vertices. In the case that each edge has an associated length or a weight, the adjacency matrix representation can be appropriately modified so entry $a_{i,j}$ contains that length or weight instead of just a 1. The most important disadvantage of the adjacency matrix representation is that it requires n^2 storage, even if the graph is very sparse, having as few as $O(n)$ edges. Moreover, just examining all the entries of the matrix would require n^2 steps, thus precluding the possibility of linear-time algorithms for sparse graphs.

The *adjacency list* representation avoids these disadvantages. The adjacency list for a vertex i is just a list of all the vertices adjacent to i (in any order). In the adjacency list representation, we have an array of size n to represent the vertices of the graph, and the i^{th} element of the array points to the adjacency list of the i^{th} vertex. The total storage used by an adjacency list representation of a graph with n vertices and m edges is $O(n + m)$. We will use this representation for all our graph algorithms that take linear or near linear time. Using the adjacency lists representation, it is easy to iterate through all edges going out of a particular

node v —and this is a very useful kind of iteration in graph algorithms, see for example the procedure `explore` in the next section; with an adjacency matrix, this would take n steps, even if there were only a few edges going out of v. One potential disadvantage of adjacency lists is that determining whether there is an edge from vertex i to vertex j may take as many as n steps, since there is no systematic shortcut to scanning the adjacency list of vertex i.
So far we have discussed how to represent directed graphs on the computer. To represent undirected graphs, just remember that an undirected graph is just a special case of directed graph: one that is symmetric and has no loops. The adjacency matrix of an undirected graph has the additional proprties that $a_{ii} = 0$, and $a_{ij} = a_{ji}$ for all nodes i and j.

3 Depth-first search

We will start by studying two fundamental algorithms for searching a graph: *depth-first search* and *breadth-first search.* To better understand the need for these algorithms, let us imagine the computer's view of a graph that has been input into it: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as "visited." This corresponds to exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from the current room, and you are also allowed to use the chalk to mark the current room as having been "visited." Clearly, it would not be easy to find your way around without the use of any additional data structures.
Mythology tells us that the right data structure for exploring a maze is a ball of string. Depth-first search is technique for exploring a graph using as a basic data structure a *stack* —the cyberanalog of a ball of string. It is not hard to visualize why a stack is the right way to implement a ball of string in a computer. A ball of string allows two primitive steps: *unwind* to get into a new room (the stack analog is *push the new room*) and *rewind* to return to the previous room —the stack analog is *pop*.
Actually, we shall present depth-first search not as an algorithm explicitly manipulating a stack, but as a *recursive* procedure, using the stack of activation records provided by the programming language. We start by defining a recursive procedure `explore`. When `explore` is invoked on a vertex v, and explores all previously unexplored vertices that are reachable from v.

```
1.  procedure explore(v:  vertex)
2.  previsit(v)
3.  for each edge (v,w) out of v do
4.    {if not visited(w) then explore(w)}
5.  postvisit(v)
6.  algorithm dfs(G = (V,E): graph)
7.  for each v̄ in V do {visited(v) := false}
8.  for each v̄ in V do
9.    {if not visited(v) then explore(v)}}
```

`previsit(v)` and `postvisit(v)` are two routines that perform a constant number of operations on the node v. As we shall see, by varying these routines depth-first search can be tuned to accomplish a wide range of important and sophisticated tasks. Minimally, `previsit(v)` should set `visited(v)` to `true` —effectively "chalkmarking" v.

It is easy to see that depth-first search takes $O(n + e)$ steps on a graph with n vertices and e edges, because the algorithm performs *a finite number of steps per each node and edge of the graph.* To see this clearly, let us go through the exercise of "charging" each operation of the

algorithm to a particular node or edge of the graph. The procedure `explore` (lines 1 through 5) is called at most once —in fact, *exactly* once— for each node; the boolean variable `visited` makes sure each node is visited only once. The finite amount of work required to perform the call `explore(v)` (such as adding a new activation record on the stack and popping it in the end) is charged to the node v. Similarly, the finite amount of work in the routines `previsit(v)` and `postvisit(v)` (lines 2 and 5) is charged to the node v. Finally, the work done for each outgoing edge (v, w) (looking it up in the adjacency list of v and checking whether v has been visited, lines 3 and 4) is charged to that edge. Each edge (v, w) is processed only once, because its tail v is visited only once —in the case of an undirected graph, each edge will be visited twice, once in each direction. Hence, every element (node or edge) of the graph will end up being charged a small finite number of elementary operations. Since this accounts for all the work performed by the algorithm, it follows that depth-first search takes $O(n + e)$ work —it is a *linear-time algorithm.*

This is the fastest possible running time of any algorithm solving a nontrivial problem, since any decent problem will require that each data item be inspected at least once.[1] It means that a graph can be searched in time comparable to the time it takes to read it into the memory of a computer!

By modifying the procedures `previsit` and `postvisit`, we can use depth-first search to compute and store useful information and solve a number of important problems. The simplest such modification is to record the "time" each node is first seen and last seen by the algorithm. We could keep a counter (or *clock*), and assign to each vertex the "time" `previsit` was executed, and the time `postvisit` was executed. This would correspond to the following code:

```
procedure previsit(v:  vertex)
visited(v) := true
pre(v) := clock++
procedure postvisit(v:  vertex)
post(v) := clock++
```

Naturally, `clock` will have to be initialized to zero in the beginning of the main algorithm. If we think of depth-first search as using an explicit stack, then `pre(v)` is the time node v is first placed on the stack, and `post(v)` is the time when v is removed from the stack.

Let us run depth-first search on the undirected graph shown below. (We use the convention that the nodes of the graph are always processed in loops such as those in line 3 and line 8 in increasing lexicographic order of their names.) Next to each node v we show the numbers `pre(v)/post(v)`. Depth-first search defines a tree in a natural way: each time a new vertex,

[1] There is an exception to this, namely *database queries*, which can often be answered without examining the whole database (for example, by binary search). Such procedures, however, require data that have been preprocessed and structured in a particular way, and so cannot be fairly compared with algorithms, such as depth-first search, which must work on unpreprocessed inputs.

say w, is discovered, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v, w). The contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. Thus, the tree summarizes the history of the stack during the algorithm. If the graph is disconnected, and thus depth-first search has to be restarted, a separate tree is created for each restart. The edges of the depth-first search tree(s) are called *tree edges;* they are the edges through which the control of the algorithm flows. The remaining edges, shown as broken lines, go from a node of the tree to an *ancestor* of the node, and are therefore called *back edges.*

Notice the following important property of the numbers `pre[v]` and `post[v]`: Since `[pre[v],post[v]]` is essentially the time interval during which v stayed on the stack, it is always the case that

two intervals `[pre[u],post[u]]` *and* `[pre[v],post[v]]` *are either disjoint, or one contains another.*

In other words, any two such intervals cannot partially overlap. We shall see many more useful properties of these numbers.

One of the simplest feats that can accomplished by appropriately modifying depth-first search is to subdivide an undirected graph to its *connected components*, defined next. A *path* in a graph $G = (V, E)$ is a sequence $(v_0, v_1, \ldots, v_n)$ of nodes, such that $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, n$. An undirected graph is said to be *connected* if there is a path between any two nodes.[2] If an undirected graph is disconnected, then its nodes can be partitioned into *connected components;* for example, the undirected graph above has three connected components. We can use depth-first search to tell if a graph is connected, and, if not, to assign to each node v an integer, `ccnum[v]`, identifying the connected component of the graph to which v belongs. This can be accomplished by adding a line to `previsit`.

```
procedure previsit(v:  vertex)
visited(v) := true
ccnum(v) := cc
```

Here `cc` is an integer used to identify the various connected components of the graph; it is initialized to zero at line 7 of dfs, and increased by one just before each call of `explore` at line 9 of dfs.

[2] As we shall soon see, connectivity in directed graphs is a much more subtle concept.