IQRA National University Peshawar

Name: Noor rahman

Reg Id# 14232

MSCS 2017-18

ADVANCED ALGORITHM ANALYSIS

Re-MID SEMESTER ASSIGNMENT

Department of Computer Science

Q1. Analyze Insertion Sort for Best Case?

Analysis of Insertion

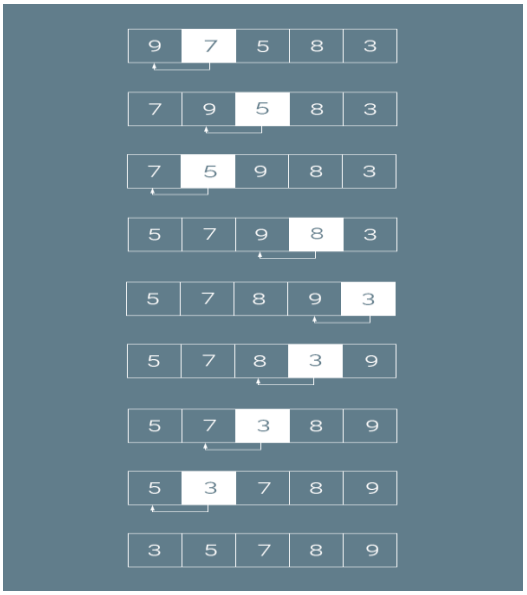
Insertion Sort is an algorithm used to sort a given list of items. It does so by iterating through the list and building the sorted output one item at a time. Upon each iteration, an item is taken from the list and inserted into the correct position by comparison with its neighbors. This process is repeated until we reach the last item and there are no more left to be sorted.

Let's begin by taking a look at **some** of it's advantages:

- It's a simple algorithm to implement
- Performance is very high when operating with **small** lists
- Even more so when the list is already mostly sorted, as fewer iterations of the sorting logic need to take place

However, the algorithm does hold some disadvantages:

- Performance suffers when large lists are used, as this could involve carrying out a lot of comparisons and shifting of array items
- The algorithm doesn't perform as well as the merge sort and quick sort algorithms, both of which we'll look at soon



Pseudo code

So now we know what the algorithm does, we should declare exactly what it does in the form of pseudo code to help aid understanding and communicate the process carried out to sort our list. For insertion sort this is fairly simple, so let's take a quick look at **one** of the ways in which we could approach it:

Pretty straightforward, eh? There's not a huge amount going on here, but it's important to understand the purpose of each line:

1. We begin by declaring the for loop for the algorithm, we're going to loop through the entire length of our input array. We start from the second item in our array as the first item has nothing to the left to compare it to.
2. We set the current item which is to be sorted (**key**) equal to the current item at our iteration position.
3. We declare the variable **i** which we use to reference the position before the current iteration, i.e. to the left of it.

	Cost
1: for $j = 2$ *to* *A.length* do	n
2: $key = A[j]$	$n-1$
3: $i = j-1$	$n-1$
4: while $i > 0$ *and* $A[i] > key$ do	$\sum_{j=2}^{n} t_j$
5: $A[i+1] = A[i]$	$\sum_{j=2}^{n} (t_j - 1)$
6: $i = i-1$	$\sum_{j=2}^{n} (t_j - 1)$
7: end while	
8: $A[i+1] = key$	$n-1$
9: end for	

4. If necessary, then this step is where we start sorting the current item. We begin by checking that our **i** index from **step 3** is at least the first item in the list (> position 0) and that the value in our A array at this index **i** is greater than our key value from **step 2**.
5. Whilst the above step holds true, the item at A[i] (the item to the left of our current iteration) becomes the value set at the current iteration A[i + 1] — which is equal to A[j]. This is because this value is greater than the value currently at A[i + 1], so it should be put in its place instead.
6. At this step in our whilst loop, we move another step to the left in our array by decrementing our **i** value.
7. We reach the end of our **whilst** statement when the conditions are no longer satisfied. This is if both the **i** value reaches zero or the value at A[i] is not greater than the key value.
8. We then finish this iteration by inserting our **key** value into it's position in the array. Whenever the whilst loop exited at **step 6,** our **i** variable was set to the next index (to the left) in our input array. If the loop exited because the value here was less than our **key** value, then the **key** value is insert to the right of that at [i + 1], yet before the item that was inserted at **step 5** before **i** was decremented. If the loop exited due to **i** reaching zero, then our **key** value is simply inserted at the beginning of our array.

Example

If you've got that, great! It'll help to run through an example so we can really nail what's going on. If you didn't follow, try going through it again and return to this section after!

So, let's start by declaring an array which we want to sort:

Now we're ready to go, we begin at the very first line of the pseudo code. Our input array contains 4 elements, so we're going to be looping:

$$A = [5, 3, 1, 9]$$

Pretty straight forward, eh? There's not a huge amount going on here, but it's important to understand the purpose of each line:

1. We begin by declaring the for loop for the algorithm, we're going to loop through the entire length of our input array. We start from the second item in our array as the first item has nothing to the left to compare it to.
2. We set the current item which is to be sorted (**key**) equal to the current item at our iteration position.
3. We declare the variable **i** which we use to reference the position before the current iteration, i.e to the left of it.

$$\text{for } j = 2 \text{ to } 4 \text{ do}$$

4. If necessary, then this step is where we start sorting the current item. We begin by checking that our **i** index from **step 3** is at least the first item in the list (> position 0) and that the value in our A array at this index **i** is greater than our key value from **step 2**.
5. Whilst the above step holds true, the item at A[i] (the item to the left of our current iteration) becomes the value set at the current iteration A[i + 1] — which is equal to A[j]. This is because this value is greater than the value currently at A[i + 1], so it should be put in it's place instead.
6. At this step in our whilst loop, we move another step to the left in our array by decrementing

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

our **i** value.

7. We reach the end of our **whilst** statement when the conditions are no longer satisfied. This is if both the **i** value reaches zero or the value at A[i] is not greater than the key value.
8. We then finish this iteration by inserting our **key** value into it's position in the array. Whenever the whilst loop exited at **step 6,** our **i** variable was set to the next index (to the left) in our input array. If the loop exited because the value here was less than our **key** value, then the **key** value is insert to the right of that at [i + 1], yet before the item that was inserted at **step 5** before **i** was decremented. If the loop exited due to **i** reaching zero, then our **key** value is simply inserted at the beginning of our array.

Time Complexity

Average Case

Now we know how our Insertion Sort algorithm works , we need to understand how efficient the algorithm is and to do this, we have to calculate it's running time. This process is done by taking a look at each step of the algorithm at calculating it's 'cost' — which is how expensive the operation is in terms of **time**.

Best Case

But what about the best case for our algorithm? This is the situation that occurs when say the input array is already sorted. The steps in the algorithm will still be executed, **but** the while loop that does the sorting will not be entered - which reduces the complexity greatly. because of this, the algorithm is simplified which is reflected in the equation below:

1: $key = 3$
2: $i = 1$
3: **while** $i > 0$ *and* $A[i] > key$ **do**
4: $\quad A[2] = 5$
5: $\quad i = i - 1$
6: **end while**
7: $A[1] = 3$

Worst Case

And finally, the worst case for our algorithm occurs when the input array is in the complete opposite order from what we want it to be for being sorted. So in our case, our example array would be equal to A= [9, 5, 3, 1] - which is in decreasing order. The equation for this would look like so:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\tfrac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\tfrac{n(n-1)}{2}\right) + c_7\left(\tfrac{n(n-1)}{2}\right) + c_8(n-1) \\
&= c_1 n + c_2 n - c_2 + c_4 n - c_4 + \tfrac{c_5 n^2}{2} + \tfrac{c_5 n}{2} - c_5 + \tfrac{c_6 n^2}{2} \\
&\quad + \tfrac{c_6 n}{2} + \tfrac{c_7 n^2}{2} + \tfrac{c_7 n}{2} + c_8 n - c_8 \\
&= \left(\tfrac{c_5}{2} + \tfrac{c_6}{2} + \tfrac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

This is because the entire algorithm, including the for loop, would need to be executed for every single iteration as each item in the array needs to be checked and sorted. We took three steps to simply this equation, with the final step being able to be expressed as:

$$an^2 + bn + c$$

We can again remove the constants from this expression, which gives use a worst case time complexity of:

$$O(n^2)$$

Q2. Explain briefly each of the following term with a supporting Graph.

1. Adjacent Edges

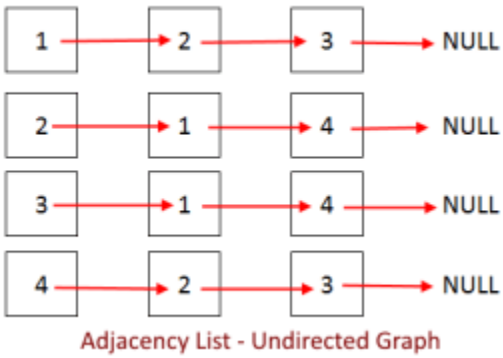
What is Graph:

G = (V,E)

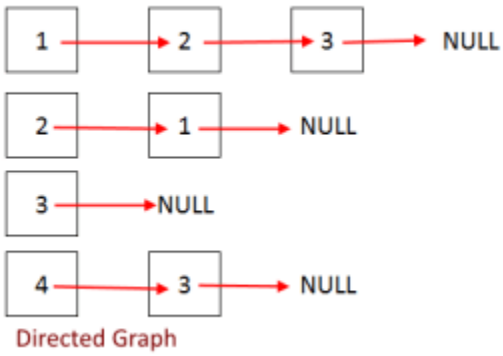
Graph is a collection of nodes or vertices (V) and edges(E) between them. We can traverse these nodes using the edges. These edges might be weighted or non-weighted.

There can be two kinds of Graphs

- Un-directed Graph – when you can traverse either direction between two nodes.
- Directed Graph – when you can traverse only in the specified direction between two nodes.
- Adjacency List is the Array[] of Linked List, where array size is same as number of Vertices in the graph. Every Vertex has a Linked List. Each Node in this Linked list represents the reference to the other vertices which share an edge with the current vertex. The weights can also be stored in the Linked List Node.



Adjacency List - Undirected Graph



Directed Graph

-
- The code below might look complex since we are implementing everything from scratch like linked list, for better understanding. Read the articles below for easier implementations (Adjacency Matrix and Adjacency List)
-

ii. Adjacent Nodes

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. The adjacency-list representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs--those for which $|E|$ is much less than $|V|2$. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. An adjacency-matrix representation may be preferred, however, when the graph is ***dense***--$|E|$ is close to $|V|2$ -- or when we need to be able to tell quickly if there is an edge connecting two given vertices. For example, two of the all-pairs shortest-paths algorithms presented in Chapter 26 assume that their input graphs are represented by adjacency matrices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains (pointers to) all the vertices v such that there is an edge $(u,v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. The vertices in each adjacency list are typically stored in an arbitrary order. Figure 23.1(b) is an adjacency-list representation of the undirected graph in Figure 23.1(a). Similarly, Figure 23.2(b) is an adjacency-list representation of the directed graph in Figure 23.2(a).

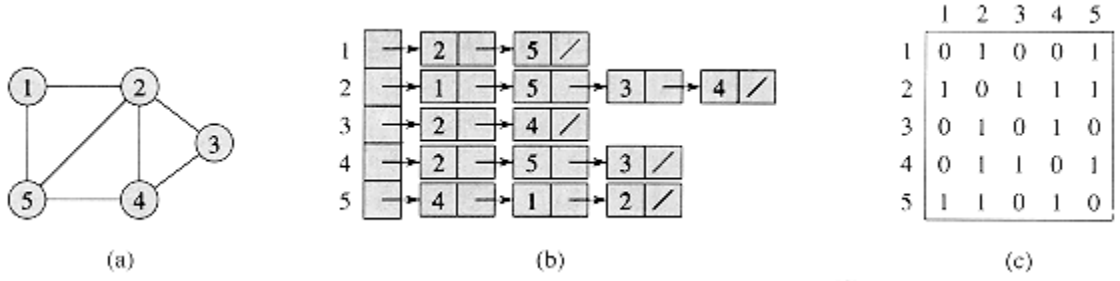


Figure .Two representations of an undirected graph. (a) An undirected graph G having five vertices and seven edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

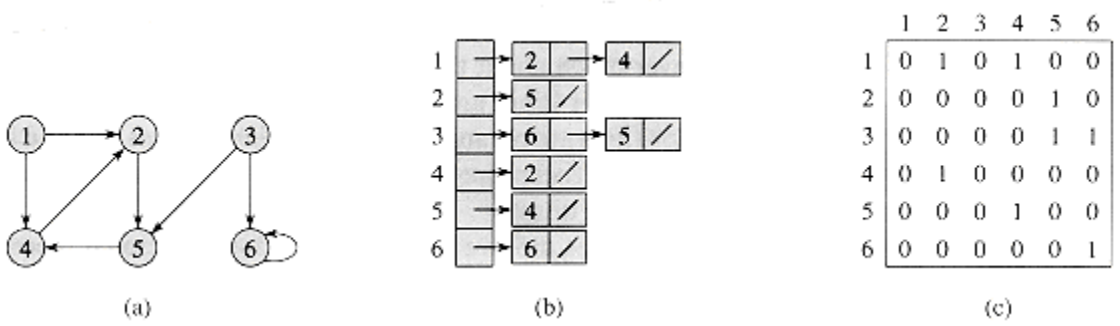


Figure Two representations of a directed graph. (a) A directed graph G having six vertices and eight edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u,v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u,v) is an undirected edge, then u appears in v's adjacency list and vice versa. Whether a graph is directed or not, the adjacency-list representation has the desirable property that the amount of memory it requires is $O(\max(V, E)) = O(V + E)$.

Adjacency lists can readily be adapted to represent ***weighted graphs***, that is, graphs for which each edge has an associated ***weight***, typically given by a ***weight function*** $w : E \rightarrow \mathbf{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. The weight $w(u,v)$ of the edge $(u,v) \in E$ is simply stored with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that it can be modified to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u,v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency-matrix re presentation of the graph, at the cost of using asymptotically more memory.

For the ***adjacency-matrix representation*** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. The adjacency-matrix representation of a graph G then consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

Figures 23.1(c) and 23.2(c) are the adjacency matrices of the undirected and directed graphs in Figures 23.1(a) and 23.2(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 23.1(c). We define the the ***transpose*** of a matrix $A = (a_{ij})$ to be the matrix $A^T = (a_{ij}^T)$ given by $a_{ij}^T = a_{ji}$. Since in an undirected graph, (u,v) and (v,u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^T$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, the weight $w(u, v)$ of the edge $(u,v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a `NIL` value can be stored as its

corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small. Moreover, if the graph is unweighted, there is an additional advantage in storage for the adjacency-matrix representation. Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

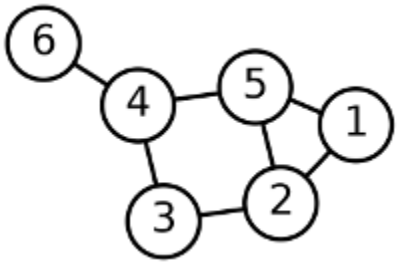
iii. Closed Graph

A **graph** *G* consists of two types of elements: **vertices** and **edges**. Each edge has two **endpoints**, which belong to the vertex set. We say that the edge **connects** (or joins) these two vertices.

The **vertex set** of *G* is denoted *V(G)*, or just *V* if there is no ambiguity.

An edge between vertices *u* and *v* is written as {*u*, *v*}. The **edge set** of *G* is denoted *E(G)*, or just *E* if there is no ambiguity.

The graph in this picture has the vertex set V = {1, 2, 3, 4, 5, 6}. The edge set E = {{1, 2}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {4, 5}, {4, 6}}.



A **self-loop** is an edge whose endpoints is a single vertex. **Multiple edges** are two or more edges that join the same two vertices.

A graph is called **simple** if it has no self-loops and no multiple edges, and a **multigraph** if it does have multiple edges.

The **degree** of a vertex *v* is the number of edges that connect to *v*.

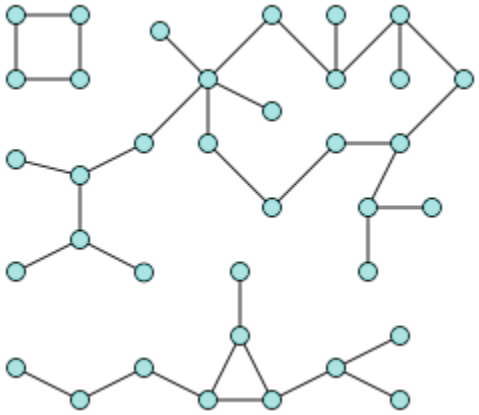
A **path** in a graph *G = (V, E)* is a sequence of vertices $v_1, v_2, \ldots, v_k$, with the property that there are edges between v_i and v_{i+1}. We say that the path goes from v_1 to v_k. The sequence 6, 4, 5, 1, 2 is a path from 6 to 2 in the graph above. A path is **simple** if its vertices are all different.

A **cycle** is a path $v_1, v_2, \ldots, v_k$ for which $k > 2$, the first $k - 1$ vertices are all different, and $v_1 = v_k$. The sequence 4, 5, 2, 3, 4 is a cycle in the graph above.

A graph is **connected** if for every pair of vertices *u* and *v*, there is a path from *u* to *v*.

If there is a path connecting *u* and *v*, the **distance** between these vertices is defined as the minimal number of edges on a path from *u* to *v*.

A **connected component** is a subgraph of maximum size, in which every pair of vertices are connected by a path. Here is a graph with three connected components.



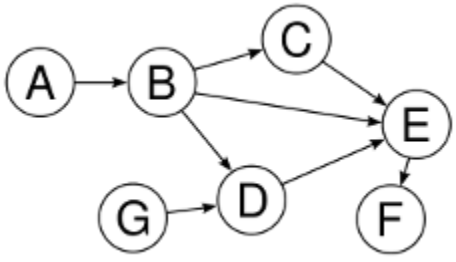
Trees

A tree is a connected simple acyclic graph. A vertex with degree 1 in a tree is called a leaf.

Directed graphs

A directed graph or digraph G = (V, E) consists of a vertex set V and an edge set of ordered pairs E of elements in the vertex set.

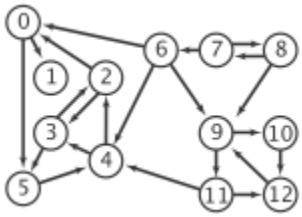
Here is a simple acyclic digraph (often called a DAG, "directed acyclic graph") with seven vertices and eight edges.



iv. Directed Graph

A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network.

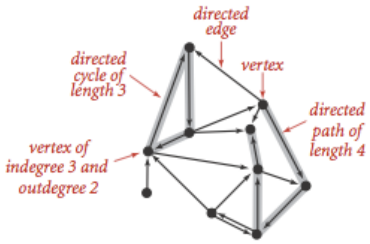
A *directed graph* (or *digraph*) is a set of *vertices* and a collection of *directed edges* that each connects an ordered pair of vertices. We say that a directed edge *points from* the first vertex in the pair and *points to* the second vertex in the pair. We use the names 0 through V-1 for the vertices in a V-vertex graph.



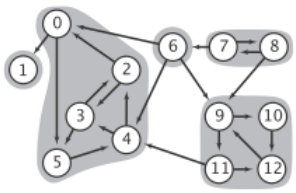
Glossary.

Here are some definitions that we use.

- A *self-loop* is an edge that connects a vertex to itself.
- Two edges are *parallel* if they connect the same ordered pair of vertices.
- The *outdegree* of a vertex is the number of edges pointing from it.
- The *indegree* of a vertex is the number of edges pointing to it.
- A *subgraph* is a subset of a digraph's edges (and associated vertices) that constitutes a digraph.
- A *directed path* in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence, with no repeated edges.
- A directed path is *simple* if it has no repeated vertices.
- A *directed cycle* is a directed path (with at least one edge) whose first and last vertices are the same.
- A directed cycle is *simple* if it has no repeated vertices (other than the requisite repetition of the first and last vertices).
- The *length* of a path or a cycle is its number of edges.
- We say that a vertex `w` is *reachable from* a vertex `v` if there exists a directed path from `v` to `w`.
- We say that two vertices `v` and `w` are *strongly connected* if they are mutually reachable: there is a directed path from `v` to `w` and a directed path from `w` to `v`.
- A digraph is *strongly connected* if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of *strongly connected components*, which are maximal strongly connected sub graphs.
- A *directed acyclic graph* (or DAG) is a digraph with no directed cycles.



Anatomy of a digraph



A digraph and its strong components

Digraph graph data type.

We implement the following digraph API.

public class	Digraph	
	Digraph(int V)	*create a V-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream in*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge v->w to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to v by edges pointing from v*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

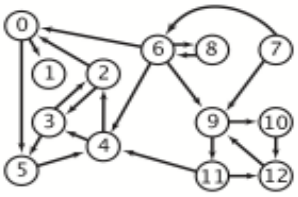
The key method `adj()` allows client code to iterate through the vertices adjacent from a given vertex.

We prepare the test data tiny DG txt using the following input file format.

tinyDG.txt

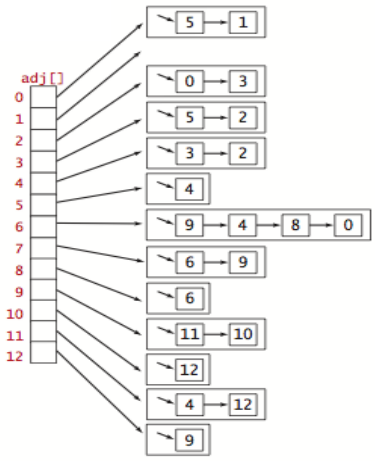
V → 13
E → 22

```
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 5  4
 0  5
 6  4
 6  9
 7  6
```



Graph representation.

We use the *adjacency-lists representation*, where we maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex.



Digraph.java implements the digraph API using the adjacency-lists representation. AdjMatrixDigraph.java implements the same API using the adjacency-matrix representation.

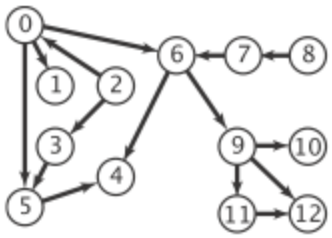
Reachability in digraphs.

Depth-first search and breadth-first search are fundamentally digraph-processing algorithms.

- *Single-source reachability:* Given a digraph and source `s`, is there a directed path from s to v? If so, find such a path. DirectedDFS.java uses depth-first search to solve this problem.
- *Multiple-source reachability:* Given a digraph and a *set* of source vertices, is there a directed path from *any* vertex in the set to v? DirectedDFS.java uses depth-first search to solve this problem.
- *Single-source directed paths:* given a digraph and source `s`, is there a directed path from s to v? If so, find such a path. DepthFirstDirectedPaths.java uses depth-first search to solve this problem.
- *Single-source shortest directed paths*: given a digraph and source `s`, is there a directed path from s to v? If so, find a shortest such path. BreadthFirstDirectedPaths.java uses breadth-first search to solve this problem.

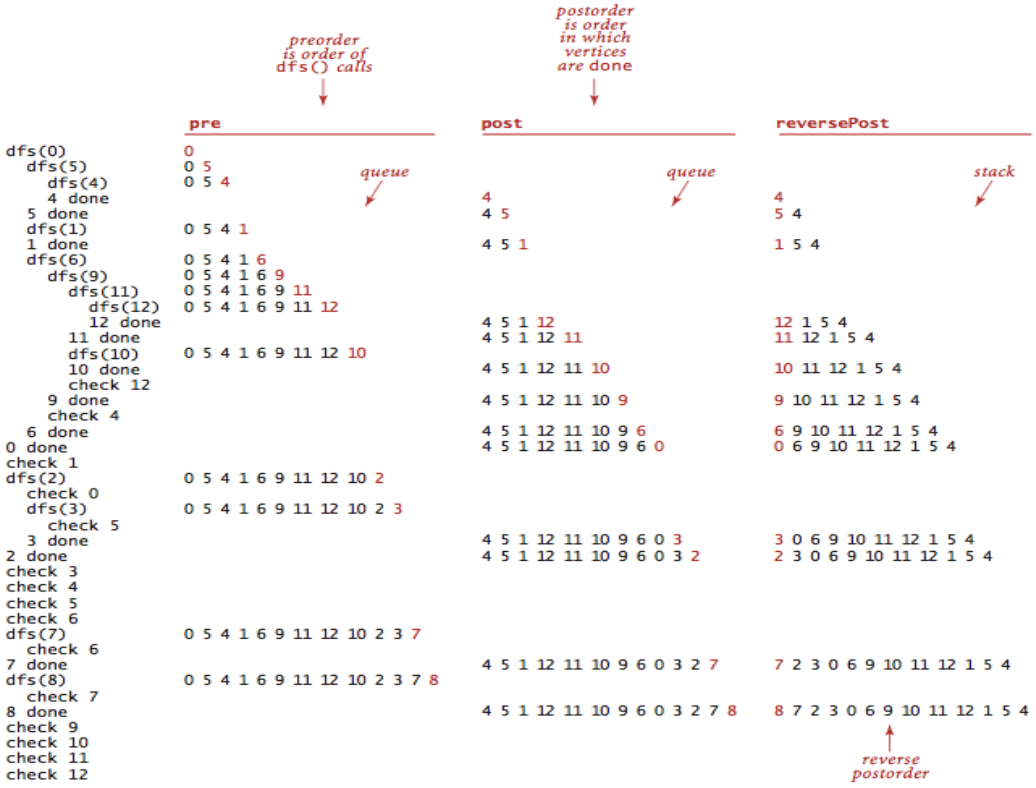
Cycles and DAGs.

Directed cycles are of particular importance in applications that involve processing digraphs. The input file tinyDAG.txt corresponds to the following DAG:



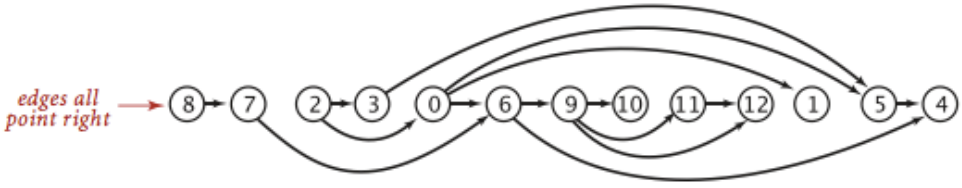
- *Directed cycle detection*: does a given digraph have a directed cycle? If so, find such a cycle. DirectedCycle.java solves this problem using depth-first search.
- *Depth-first orders*: Depth-first search search visits each vertex exactly once. Three vertex orderings are of interest in typical applications:
    - *Preorder*: Put the vertex on a queue before the recursive calls.
    - *Postorder*: Put the vertex on a queue after the recursive calls.
    - *Reverse postorder*: Put the vertex on a stack after the recursive calls.

DepthFirstOrder.java computes these orders.



- *Topological sort*: given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not

possible). Topological.java solves this problem using depth-first search. Remarkably, a reverse post order in a DAG provides a topological order.



Proposition.
A digraph has a topological order if and only if it is a DAG.

Proposition.
Reverse post order in a DAG is a topological sort.

Proposition.
With depth-first search, we can topologically sort a DAG in time proportional to V + E.

Strong connectivity.

Strong connectivity is an equivalence relation on the set of vertices:

- *Reflexive*: Every vertex v is strongly connected to itself.
- *Symmetric*: If v is strongly connected to w, then w is strongly connected to v.
- *Transitive*: If v is strongly connected to w and w is strongly connected to x, then v is also strongly connected to x.

Strong connectivity partitions the vertices into equivalence classes, which we refer to as *strong components* for short. We seek to implement the following API:

public class	SCC	
	SCC(Digraph G)	*preprocessing constructor*
boolean	stronglyConnected(int v, int w)	*are* v *and* w *strongly connected?*
int	count()	*number of strong components*
int	id(int v)	*component identifier for* v (*between* 0 *and* count()-1)

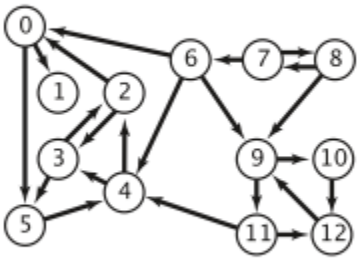
Remarkably, KosarajuSharirSCC.java implements the API with just a few lines of code added to CC.java, as follows:

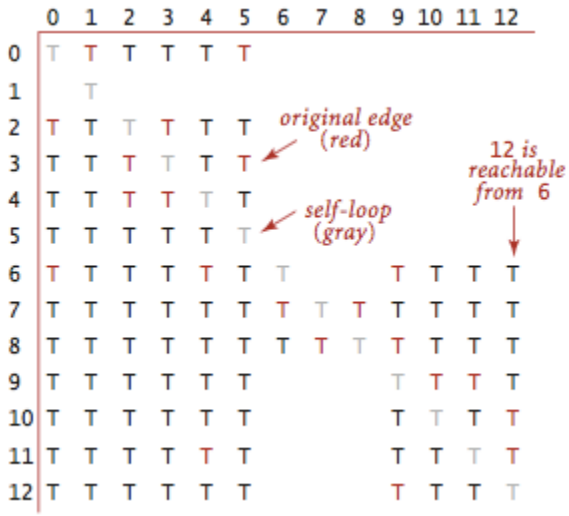
- Given a digraph G, use DepthFirstOrder.java to compute the reverse post order of its reverse, G^R.
- Run standard DFS on G, but consider the unmarked vertices in the order just computed instead of the standard numerical order.

- All vertices reached on a call to the recursive `dfs()` from the constructor are in a strong component (!), so identify them as in CC.

Proposition.
The Kosaraju-Sharir algorithm uses preprocessing time and space proportional to V + E to support constant-time strong connectivity queries in a digraph.

Transitive closure.
The *transitive closure* of a digraph G is another digraph with the same set of vertices, but with an edge from v to w if and only if w is reachable from v in G.





TransitiveClosure.java computes the transitive closure of a digraph by running depth-first search from each vertex and storing the results. This solution is ideal for small or dense digraphs, but it is not a solution for the large digraphs we might encounter in practice because the constructor uses space proportional to V^2 and time proportional to V (V + E).

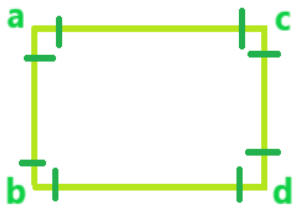
v. Cycle

Cycle: a simple path with no repeated vertices or edges other than the starting and ending vertices. A **cycle** in a directed graph is called a directed **cycle**. Multiple edges: in principle, a graph can have two or more edges connecting the same two vertices in the same direction.

Cycle Graph: In graph theory, a graph that consists of single cycle is called a cycle graph or circular graph. The cycle graph with n vertices is called Cn.

Properties of Cycle Graph:-

- It is a Connected Graph.
- A Cycle Graph or Circular Graph is a graph that consists of a single cycle.
- In a Cycle Graph number of vertices is equal to number of edges.
- A Cycle Graph is 2-edge colorable or 2-vertex colorable, if and only if it has an even number of vertices.
- A Cycle Graph is 3-edge colorable or 3-edge colorable, if and only if it has an odd number of vertices.
- In a Cycle Graph, Degree of each vertx in a graph is two.
- The degree of a Cycle graph is 2 times the number of vertices. As each edge is counted twice.



Examples: Input: Number of vertices = 4
Output: Degree is 8
Edges are 4
Explanation: The total edges are 4 and the Degree of the Graph is 8 as 2 edge incident on each of
the vertices i.e on a, b, c, and d.

```
Input: number of vertices = 5
Output: Degree is 10
        Edges are 5
```

Q3. Sort the following list using Insertion Sort.

15, 4, 11, 3, 5, 1

Insertion sort:

An **insertion sort** algorithm is one that sorts by inserting records in an existing sorted array. An example of a simple insertion sort is as follows. Assume that the keys in the first i - 1 array slots are sorted. Let x be the value of the key in the ith slot. Compare x in sequence with the key in the (i - 1) st slot, the one in the (i - 2) nd slot, etc., until a key is found that is smaller than x. Let j be the slot where that key is located. Move the keys in slots j +1 through i - 1 to slots j +2 through i, and insert x in the (j + 1)st slot. Repeat this process for i = 2 through i = n

Insertion Sort

Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n; array of keys of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

```
INSERTION-SORT(A)
   for i = 1 to n
        key ← A [i]
        j ← i – 1
         while j > = 0 and A[j] > key
                A[j+1] ← A[j]
                j ← j – 1
        End while
        A[j+1] ← key
  End for
```

How Insertion Sort Works?

The first element in the array is assumed to be sorted. Take the second element and store it separately in `key`.

Compare `key` with the first element. If the first element is greater than `key`, then *key* is placed in front of the first element.

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving form right to left.

29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64

20, 29, 34, 73, 64
20, 29, 34, 64, 73
Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:
we need 0 comparisons to insert the first element
we need 1 comparison to insert the second element
we need 2 comparisons to insert the third element
...
we need (N-1) comparisons (at most) to insert the last element
Totally,
$1 + 2 + 3 + ... + (N-1) = O(n^2)$
The worst-case runtimecomplexity is $O(n^2)$.What is the best-case runtime complexity? O(n). The advantage of insertion sort comparing it to the previous two sorting algorithm is that insertion sort runs in linear time on nearly sorted data.

Now we have the value

15, 4, 11, 3, 5, 1