Part I

Exponentiation, Binary Search
Exponentiation

Input Two numbers: $a$ and integer $n \geq 0$

Goal Compute $a^n$
**Input** Two numbers: $a$ and integer $n \geq 0$

**Goal** Compute $a^n$

Obvious algorithm:

```
SlowPow(a,n):
    x = 1;
    for i = 1 to n do
        x = x*a
    Output x
```

$O(n)$ multiplications.
Fast Exponentiation

Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor} \).
Observation: $a^n = a^\lfloor n/2 \rfloor a^{n/2} = a^\lfloor n/2 \rfloor a^{n/2} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

FastPow(a,n):
   if (n = 0) return 1
   x = FastPow(a,\lfloor n/2 \rfloor)
   x = x*x
   if (n is odd)
       x = x*a
   return x
Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil-\lfloor n/2 \rfloor}$.

FastPow(a,n):
   if (n = 0) return 1
   x = FastPow(a,\lfloor n/2 \rfloor)
   x = x*x
   if (n is odd)
      x = x*a
   return x

$T(n)$: number of multiplications for $n$
Exponentiation

Binary Search

Fast Exponentiation

Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor} \).

FastPow(a,n):

if (n = 0) return 1

x = FastPow(a,\lfloor n/2 \rfloor)

x = x*x

if (n is odd)
    x = x*a

return x

\( T(n) \): number of multiplications for \( n \)

\[
T(n) = T(\lfloor n/2 \rfloor) + 2
\]

\( T(n) = \)

Chekuri CS473ug
Fast Exponentiation

**Observation:** \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor} \).

```python
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,\lfloor n/2 \rfloor)
    x = x*x
    if (n is odd)
        x = x*a
    return x
```

\( T(n) \): number of multiplications for \( n \)

\[
T(n) = T(\lfloor n/2 \rfloor) + 2
\]

\( T(n) = \Theta(\log n) \).
Complexity of Exponentiation

**Question:** Is SlowPow() a polynomial time algorithm? FastPow?
Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
**Complexity of Exponentiation**

**Question:** Is SlowPow() a polynomial time algorithm? FastPow?

Input size: \( \log a + \log n \)
Output size:
Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
Output size: $n \log a$. Not necessarily polynomial in input size!

Both SlowPow and FastPow are polynomial in output size.
Exponentiation modulo a given number

Exponentiation in applications:

**Input**  Three integers: \( a, n \geq 0, p \geq 2 \) (typically a prime)

**Goal**  Compute \( a^n \mod p \)
Exponentiation modulo a given number

Exponentiation in applications:

Input  Three integers: $a, \ n \geq 0, \ p \geq 2$ (typically a prime)
Goal  Compute $a^n \ \mod \ p$

Input size: $\log a + \log n + \log p$
Output size: $O(\log p)$ and hence polynomial in input size.
Exponentiation in applications:

**Input**  Three integers: \( a, n \geq 0, p \geq 2 \) (typically a prime)

**Goal**  Compute \( a^n \mod p \)

Input size: \( \log a + \log n + \log p \)

Output size: \( O(\log p) \) and hence polynomial in input size.

Observation: \( xy \mod p = ((x \mod p)(y \mod p)) \mod p \)
Exponentiation modulo a given number

Exponentiation in applications:

**Input** Three integers: \( a, n \geq 0, p \geq 2 \) (typically a prime)

**Goal** Compute \( a^n \mod p \)

Input size: \( \log a + \log n + \log p \)

Output size: \( O(\log p) \) and hence polynomial in input size.

Observation: \( xy \mod p = ((x \mod p)(y \mod p)) \mod p \)

**FastPowMod**(a,n,p):

1. if (n = 0) return 1
2. x = **FastPowMod**(a,\( \lfloor n/2 \rfloor \),p)
3. x = x*x \mod p
4. if (n is odd) \( x = x*a \mod p \)
5. return x

**FastPowMod** is a polynomial time algorithm. **SlowPowMod** is not.
Input: Sorted array $A$ of $n$ numbers and number $x$
Goal: Is $x$ in $A$?

Binary Search in Sorted Arrays

**Algorithm:**

$$\text{BinarySearch}(A[a..b], x):$$

- if $(b-a \leq 0)$ return NO
- \(mid = A\lfloor (a + b) / 2 \rfloor\)
- if $(x = mid)$ return YES
- else if $(x < mid)$ return BinarySearch($A[a..\lfloor (a + b) / 2 \rfloor - 1], x$)
- else return BinarySearch($A[\lfloor (a + b) / 2 \rfloor + 1..b], x$)

**Analysis:**

$$T(n) = T(\lfloor n / 2 \rfloor) + O(1).$$

$$T(n) = O(\log n).$$

**Observation:** After $k$ steps, size of array left is $n / 2^k$. 

Chekuri | CS473ug
Binary Search in Sorted Arrays

**Input**  Sorted array \( A \) of \( n \) numbers and number \( x \)

**Goal**  Is \( x \) in \( A \)?

```python
BinarySearch(A[a..b], x):
    if (b-a <= 0) return NO
    mid = A[\lfloor(a + b)/2\rfloor]
    if (x = mid) return YES
    else if (x < mid) return BinarySearch(A[a..\lfloor(a + b)/2\rfloor−1],x)
    else return BinarySearch(A[\lfloor(a + b)/2\rfloor+1..b],x)
```

Analysis: 
\[ T(n) = T(\lfloor n/2 \rfloor) + O(1). \]
\[ T(n) = O(\log n). \]

Observation: 
After \( k \) steps, size of array left is \( n/2^k \).
Binary Search in Sorted Arrays

**Input** Sorted array $A$ of $n$ numbers and number $x$

**Goal** Is $x$ in $A$?

BinarySearch($A[a..b]$, $x$):

1. if $(b-a \leq 0)$ return NO
2. $mid = A[\lfloor (a + b)/2 \rfloor]$
3. if $(x = mid)$ return YES
4. else if $(x < mid)$ return BinarySearch($A[a..\lfloor (a + b)/2 \rfloor - 1], x$)
5. else return BinarySearch($A[\lfloor (a + b)/2 \rfloor + 1..b], x$)

**Analysis:** $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

**Observation:** After $k$ steps, size of array left is $n/2^k$
Another common use of binary search

- **Optimization version:** find solution of best (say minimum) value
- **Decision version:** is there a solution of value at most a given value $v$?
Another common use of binary search

- **Optimization version**: find solution of best (say minimum) value
- **Decision version**: is there a solution of value at most a given value $v$?

Reduce optimization to decision (may be easier to think about):
- Given instance $I$ compute upper bound $U(I)$ on best value
- Compute lower bound $L(I)$ on best value
- Do binary search on interval $[L(I), U(I)]$ using decision version as black box
- $O(\log(U(I) - L(I)))$ calls to decision version if $U(I), L(I)$ are integers
Part II

Graph Basics
Why Graphs?

- Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics
Definition

An undirected (simple) graph \( G = (V, E) \) is a 2-tuple:

- \( V \) is a set of vertices (also referred to as nodes/points)
- \( E \) is a set of edges where each edge \( e \in E \) is a set of the form \( \{u, v\} \) with \( u, v \in V \) and \( u \neq v \).

Example

In figure, \( G = (V, E) \) where \( V = \{1, 2, 3, 4, 5, 6, 7, 8\} \) and \( E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}. \)
Notation and Convention

**Notation**

An edge in an undirected graphs is an *unordered* pair of nodes and hence it is a set. Conventionally we use \((u, v)\) for \(\{u, v\}\) when it is clear from the context that the graph is undirected.

- **u** and **v** are the **end points** of an edge \(\{u, v\}\)
- **Multi-graphs** allow
  - *loops* which are edges with the same node appearing as both end points
  - *multi-edges*: different edges between same pairs of nodes
- In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.
Graph Representation I

Adjacency Matrix

Represent $G = (V, E)$ with $n$ vertices and $m$ edges using a $n \times n$ adjacency matrix $A$ where

- Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
- Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$
Adjacency Lists

Represent $G = (V, E)$ with $n$ vertices and $m$ edges using adjacency lists:

- For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbours of $u$. Sometimes $\text{Adj}(u)$ is the list of edges incident to $u$.

- Advantage: space is $O(m + n)$
- Disadvantage: cannot “easily” determine in $O(1)$ time whether $\{i, j\} \in E$
  - By sorting each list, one can achieve $O(\log n)$ time
  - By hashing “appropriately”, one can achieve $O(1)$ time

**Note:** In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.
Connectivity

Given a graph \( G = (V, E) \):

- A **path** is a sequence of *distinct* vertices \( v_1, v_2, \ldots, v_k \) such that \( \{v_i, v_{i+1}\} \in E \) for \( 1 \leq i \leq k - 1 \). The length of the path is \( k - 1 \) and the path is from \( v_1 \) to \( v_k \).

- A **cycle** is a sequence of *distinct* vertices \( v_1, v_2, \ldots, v_k \) such that \( \{v_i, v_{i+1}\} \in E \) for \( 1 \leq i \leq k - 1 \) and \( \{v_1, v_k\} \in E \).

- A vertex \( u \) is **connected** to \( v \) if there is a path from \( u \) to \( v \).

- The **connected component** of \( u \), \( \text{con}(u) \) is the set of all vertices connected to \( u \).
Define a relation $C$ on $V \times V$ as $uCv$ if $u$ is connected to $v$

- In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
- Graph is connected if only one connected component
Connectivity Problems

Fudnamental Algorithmic Problems

- Given graph $G$ and nodes $u$ and $v$, is $u$ connected to $v$?
- Given $G$ and node $u$, find all nodes that are connected to $u$.
- Find all connected components of $G$. 

Can be accomplished in $O(m+n)$ time using BFS or DFS.
Connectivity Problems

Fudnamental Algorithmic Problems

- Given graph $G$ and nodes $u$ and $v$, is $u$ connected to $v$?
- Given $G$ and node $u$, find all nodes that are connected to $u$.
- Find all connected components of $G$.

Can be accomplished in $O(m + n)$ time using BFS or DFS
Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

Explore($u$):
  Initialize $S = \{u\}$
  While there is an edge $(x, y)$ with $x \in S$ and $y \not\in S$
    add $y$ to $S$

Proposition: Explore($u$) terminates with $S = con(u)$.

Running time: depends on implementation

Breadth First Search (BFS): use queue data structure

Depth First Search (DFS): use stack data structure

Review CS 225 material!
Given $G = (V, E)$ and vertex $u \in V$:

Explore($u$):
  Initialize $S = \{u\}$
  While there is an edge $(x, y)$ with $x \in S$ and $y \not\in S$
    add $y$ to $S$

Proposition

Explore($u$) terminates with $S = con(u)$.
Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

Explore($u$):
- Initialize $S = \{u\}$
- While there is an edge $(x, y)$ with $x \in S$ and $y \notin S$
  - add $y$ to $S$

**Proposition**

Explore($u$) terminates with $S = \text{con}(u)$.

Running time: depends on implementation

- Breadth First Search (BFS): use queue data structure
- Depth First Search (DFS): use stack data structure
- Review CS 225 material!
DFS is a very versatile graph exploration strategy. Hopcroft and Tarjan (Turing Award winners) demonstrated the power of DFS to understand graph structure. DFS can be used to obtain linear time ($O(m + n)$) time algorithms for

- Finding cut-edges and cut-vertices of undirected graphs
- Finding strong connected components of directed graphs
- Linear time algorithm for testing whether a graph is planar
DFS in Undirected Graphs

Recursive version.

DFS(G)
   Mark all nodes u as unvisited
   While there is an unvisited node u do
      DFS(u)

DFS(u)
   Mark u as visited
   for each edge (u,v) in Adj(u) do
      if v is not marked
         DFS(v)

Global array Mark for all recursive calls.
Example

Definition

The set of connected components of a graph is the set \( \{ \text{component} \mid u \in V \} \).

The connected components in the above graph are \( \{1, 2, 3, 4, 5, 6, 7, 8\} \) and \( \{9, 10\} \).

A graph is said to be connected when it has exactly one connected component. In other words, every pair of vertices in the graph are connected.
DFS(G)

Mark all nodes u as unvisited
T is set to ∅
While there is an unvisited node u do
    DFS(u)
Output T

DFS(u)

Mark u as visited
for each edge (u,v) in Adj(u) do
    if v is not marked
        add edge (u,v) to T
    DFS(v)
DFS Tree/Forest

DFS(G)
Mark all nodes u as unvisited
T is set to ∅
While there is an unvisited node u do
  DFS(u)
Output T

DFS(u)
Mark u as visited
for each edge (u,v) in Adj(u) do
  if v is not marked
    add edge (u,v) to T
  DFS(v)

Edges classified into two types: \((u,v) \in E\) is a
- tree edge: belongs to \(T\)
- non-tree edge: does not belong to \(T\)
Properties of DFS tree

**Proposition**

- **$T$ is a forest and connected components of $T$ are same as those of $G$.**
- **If $(u, v)$ is a non-tree edge then, in $T$, either $u$ is an ancestor of $v$ or $v$ is an ancestor of $u$.**

**Question:** Why are there no *cross-edges*?
DFS with Visit Times

Keep track of when nodes are visited.

DFS(G)

Mark all nodes u as unvisited
T is set to \( \emptyset \)
time = 0
While there is an unvisited node u do
    DFS(u)
Output T

DFS(u)

Mark u as visited
pre(u) = ++time
for each edge (u,v) in Out(u) do
    if v is not marked
        add edge (u,v) to T
    DFS(v)
post(u) = ++time
Definition

The set of connected components of a graph is the set \{\text{connected)} \mid u \in V\}.

The connected components in the above graph are \{1, 2, 3, 4, 5, 6, 7, 8\} and \{9, 10\}.

A graph is said to be connected when it has exactly one connected component. In other words, every pair of vertices in the graph are connected.
Pre and Post numbers

Node $u$ is active in time interval $[\text{pre}(u), \text{post}(u)]$

**Proposition**

For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

\[
[5, 20] \quad u \quad [10, 25] \quad v
\]
Node \( u \) is active in time interval \([\text{pre}(u), \text{post}(u)]\)

**Proposition**

For any two nodes \( u \) and \( v \), the two intervals \([\text{pre}(u), \text{post}(u)]\) and \([\text{pre}(v), \text{post}(v)]\) are disjoint or one is contained in the other.

**Proof.**
Pre and Post numbers

Node $u$ is active in time interval $[\text{pre}(u), \text{post}(u)]$

**Proposition**

For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

**Proof.**

- Suppose $\text{pre}(u) < \text{pre}(v)$. Implies $v$ visited after $u$. 
Pre and Post numbers

Node $u$ is *active* in time interval $[\text{pre}(u),\text{post}(u)]$

**Proposition**

*For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u),\text{post}(u)]$ and $[\text{pre}(v),\text{post}(v)]$ are disjoint or one is contained in the other.*

**Proof.**

- Suppose $\text{pre}(u) < \text{pre}(v)$. Implies $v$ visited after $u$.
- If DFS($v$) invoked before DFS($u$) finished, then $\text{post}(u) > \text{post}(v)$.
Pre and Post numbers

Node $u$ is active in time interval $[\text{pre}(u), \text{post}(u)]$

**Proposition**

*For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.*

**Proof.**

- Suppose $\text{pre}(u) < \text{pre}(v)$. Implies $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, then $\text{post}(u) > \text{post}(v)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, then $\text{pre}(v) > \text{post}(u)$.
Node $u$ is active in time interval $[\text{pre}(u),\text{post}(u)]$

**Proposition**

*For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u),\text{post}(u)]$ and $[\text{pre}(v),\text{post}(v)]$ are disjoint or one is contained in the other.*

**Proof.**

- Suppose $\text{pre}(u) < \text{pre}(v)$. Implies $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, then $\text{post}(u) > \text{post}(v)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, then $\text{pre}(v) > \text{post}(u)$.

Pre and post numbers useful in several applications of DFS - soon!
Part IV

Directed Graphs and Decomposition
Definition

A directed graph $G = (V, E)$ consists of
- set of vertices/nodes $V$ and
- a set of edges/arcs $E \subseteq V \times V$.

An edge is an ordered pair of vertices. $(u, v)$ different from $(v, u)$. 
Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

- Road networks with one-way streets.
- Web-link graph: vertices are web-pages and there is an edge from page $p$ to page $p'$ if $p$ has a link to $p'$. Web graphs used by Google with PageRank algorithm to rank pages.
- Dependency graphs in variety of applications: link from $x$ to $y$ if $y$ depends on $x$. Make files for compiling programs.
- Program Analysis: functions/procedures are vertices and there is an edge from $x$ to $y$ if $x$ calls $y$. 
Graph $G = (V, E)$ with $n$ vertices and $m$ edges:


**Adjacency Lists** for each node $u$, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from $u$.

Default representation is adjacency lists.
Directed Connectivity

Given a graph $G = (V, E)$:

- A (directed) path is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from $v_1$ to $v_k$.

- A cycle is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and $(v_k, v_1) \in E$.

- A vertex $u$ can reach $v$ if there is a path from $u$ to $v$. Alternatively $v$ can be reached from $u$.

- Let $rch(u)$ be the set of all vertices reachable from $u$. 

---

Chekuri
CS473ug
Asymmetricity: A can reach B but B cannot reach A
Asymmetricity: $A$ can reach $B$ but $B$ cannot reach $A$

Questions:
- Is there a notion of connected components?
- How do we understand connectivity in directed graphs?
Connectivity and Strong Connected Components

Definition

Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$. Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$. Proposition $C$ is an equivalence relation, that is reflexive, symmetric and transitive. Equivalence classes of $C$: strong connected components of $G$. They partition the vertices of $G$. $\text{SC}(u)$: strong component containing $u$. 
Connectivity and Strong Connected Components

**Definition**

Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

Proposition: $C$ is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of $C$: strong connected components of $G$.

They partition the vertices of $G$. SC($u$): strong component containing $u$. 
Connectivity and Strong Connected Components

**Definition**
Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

**Proposition**
$C$ is an equivalence relation, that is reflexive, symmetric and transitive.
Connectivity and Strong Connected Components

**Definition**

Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

**Proposition**

$C$ is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of $C$: strong connected components of $G$. They partition the vertices of $G$. 

$\text{SC}(u)$: strong component containing $u$. 

Chekuri | CS473ug
Strong Connected Components: Example

Definition

A directed graph (also called a digraph) is \( G = (V, E) \), where

- \( V \) is a set of vertices or nodes
- \( E \subseteq V \times V \) is set of ordered pairs of vertices called edges

Viswanathan CS473ug
Directed Graph Connectivity Problems

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $rch(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.
- Find the strong component containing node $u$, that is $SC(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strong components of $G$.
Directed Graph Connectivity Problems

- Given \( G \) and nodes \( u \) and \( v \), can \( u \) reach \( v \)?
- Given \( G \) and \( u \), compute \( \text{rch}(u) \).
- Given \( G \) and \( u \), compute all \( v \) that can reach \( u \), that is all \( v \) such that \( u \in \text{rch}(v) \).
- Find the strong component containing node \( u \), that is \( \text{SC}(u) \).
- Is \( G \) strongly connected (a single strong component)?
- Compute all strong components of \( G \).

First four problems can be solve in \( O(n + m) \) time by adapting BFS/DFS to directed graphs. The last one requires a clever DFS based algorithm.
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    While there is an unvisited node u do
        DFS(u)
    Output T

DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u,v) in Out(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
    post(u) = ++time
Definition

A directed graph (also called a digraph) is $G = (V, E)$, where

- $V$ is a set of vertices or nodes
- $E \subseteq V \times V$ is set of ordered pairs of vertices called edges

Viswanathan CS473ug

Example

Directed Graphs
Diagrams and Connectivity
Diagrams... where

Viswanathan CS473ug

Chekuri CS473ug
Example

Definition
A directed graph (also called a digraph) is $G = (V, E)$, where $V$ is a set of vertices or nodes and $E \subseteq V \times V$ is a set of ordered pairs of vertices called edges.

Chekuri CS473ug
Generalizing ideas from undirected graphs:

- $\text{DFS}(u)$ outputs a directed out-tree $T$ rooted at $u$
- A vertex $v$ is in $T$ if and only if $v \in \text{rch}(u)$
- For any two vertices $x, y$ the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint are one is contained in the other.
- The running time of $\text{DFS}(u)$ is $O(k)$ where $k = \sum_{v \in \text{rch}(u)} |\text{Adj}(v)|$ plus the time to initialize the Mark array.
- $\text{DFS}(G)$ takes $O(m + n)$ time. Edges in $T$ form a disjoint collection of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.
Edges of $G$ can be classified with respect to the DFS tree $T$ as:

- **Tree edges** that belong to $T$
- A **forward edge** is a non-tree edges $(x, y)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A **backward edge** is a non-tree edge $(x, y)$ such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.
- A **cross edge** is a non-tree edges $(x, y)$ such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.
Types of Edges

- A
- B
- C
- D

- Cross
- Forward
- Backward
Directed Graph Connectivity Problems

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $\text{rch}(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$.
- Find the strong component containing node $u$, that is $\text{SC}(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strong components of $G$. 
Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?

Given $G$ and $u$, compute $rch(u)$.

Use $DFS(G, u)$ to compute $rch(u)$ in $O(n + m)$ time.
Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$. 

Note: $\text{rch}(v)$ denotes the reverse reachability set of $v$. For a given $v$, $\text{rch}(v)$ contains all nodes that can be reached from $v$ by following edges in the reverse direction.

**Correctness:**
Exercise

**Running time:**
$O(n + m)$ to obtain $G_{rev}$ from $G$ and $O(n + m)$ time to compute $\text{rch}(u)$ via DFS. If both $\text{Out}(v)$ and $\text{In}(v)$ are available at each $v$ then no need to explicitly compute $G_{rev}$. Can do it DFS($u$) in $G_{rev}$ implicitly.
Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.

Given $G = (V, E)$, $G^{rev}$ is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$.
Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.

Given $G = (V, E)$, $G^{rev}$ is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $rch(u)$ in $G^{rev}$!

- **Correctness:** exercise
- **Running time:** $O(n + m)$ to obtain $G^{rev}$ from $G$ and $O(n + m)$ time to compute $rch(u)$ via DFS. If both $Out(v)$ and $In(v)$ are available at each $v$ then no need to explicitly compute $G^{rev}$. Can do it $DFS(u)$ in $G^{rev}$ implicitly.
SC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}
$SC(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

- Find the strong component containing node $u$. That is, compute $SC(G, u)$.
$SC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}$

- Find the strong component containing node $u$. That is, compute $SC(G, u)$.

$SC(G, u) = rch(G, u) \cap rch(G^{rev}, u)$
$SC(G, u) = \{v \mid u \text{ is strongly connected to } v\}$
- Find the strong component containing node $u$. That is, compute $SC(G, u)$.

$SC(G, u) = rch(G, u) \cap rch(G^{rev}, u)$

Hence, $SC(G, u)$ can be computed with two DFSes, one in $G$ and the other in $G^{rev}$. Total $O(n + m)$ time.
Is $G$ strongly connected?
Is $G$ strongly connected?

Pick arbitrary vertex $u$. Check if $SC(G, u) = V$. 
Find all strongly connected components of $G$. 
Find *all* strongly connected components of $G$.

For each vertex $u \in V$ do
find $SC(G, u)$

Running time: $O(n(n + m))$. 
Find all strongly connected components of $G$.

For each vertex $u \in V$ do
find $SC(G, u)$

Running time: $O(n(n + m))$.

Can we do it in $O(n + m)$ time?