COMP251: Graphs, Probability and Binary numbers

Jérôme Waldispühl & Roman Sarrazin-Gendron
School of Computer Science
McGill University
Outline

• Graphs
  o Terminology, definitions and properties
  o Graph traversal: Depth-First Search and Breadth-first search

• Probability

• Binary numbers
Background

Graphs
A graph is a pair \((V, E)\), where

- \(V\) is a set of nodes, called \textit{vertices}
- \(E\) is a collection of pairs of vertices, called \textit{edges}

Example:
- A vertex represents an airport and stores the airport code
- An edge represents a flight route between two airports
Edge Types

- Directed edge
  - ordered pair of vertices \((u,v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices \((u,v)\)
  - e.g., a street
- Directed graph: all edges are directed
- Weighted edge: has a real number associated to it
  - e.g. distance between cities
  - e.g. bandwidth between internet routers
- Weighted graph: all edges have weights

ORD PVD

ORD PVD

ORD PVD

849 miles
Labeled graphs

• Labeled graphs: vertices have identifiers

  - Note: Geometric layout doesn’t matter - only connections matter

• Unlabeled graph: vertices have no identifiers
Applications

- **Electronic circuits**
  - Printed circuit board
  - Integrated circuit

- **Transportation networks**
  - Highway network
  - Flight network

- **Computer networks**
  - Local area network
  - Internet
  - Web

- **Databases**
  - Entity-relationship diagram
Terminology

• Endpoints of an edge
  – U and V are the endpoints of a
• Edges incident on a vertex
  – a, b, and d are incident on V
• Adjacent vertices
  – Connected by an edge
  – U and V are adjacent
• Degree of a vertex
  – Number of incident edges
  – X has degree 5
• Parallel edges
  – h and i are parallel edges
• Self-loop
  – j is a self-loop
Terminology (cont.)

- **Path**
  - sequence of adjacent vertices

- **Simple path**
  - path such that all its vertices are distinct

- **Examples**
  - $P_1=(V, X, Z)$ is a simple path
  - $P_2=(U, W, X, Y, W, V)$ is a path that is not simple

- **Graph is connected iff**
  - For all pair of vertices $u$ and $v$, there is a path between $u$ and $v
Terminology (cont.)

- **Cycle**
  - path that starts and ends at the same vertex
- **Simple cycle**
  - cycle where each vertex is distinct
- **Examples**
  - $C_1 = (V, X, Y, W, U, \ldots)$ is a simple cycle
  - $C_2 = (U, W, X, Y, W, V, \ldots)$ is a cycle that is not simple
- **A tree is a connected acyclic graph**
Properties

Property 1

$\sum_{v \in V} \deg(v) = 2|E|

Why? The sum of the degrees of all vertices is $2 \times$ number of edges

Property 2

In an undirected graph with no self-loops and no multiple edges

$|E| \leq |V| (|V| - 1)/2$

Why?

Each edge links exactly two vertices. Eventually you run out of vertices to draw edges

Notation

| $V$ | number of vertices |
| $|E|$ | number of edges |
| $\deg(v)$ | degree of vertex $v$ |

Example

- $|V| = 4$
- $|E| = 6$
- $\deg(v) = 3$
Data structure for graphs - Adjacency lists

• Graph can be stored as
  – A dictionary of pairs (key, info) where
  – key = vertex identifier
  – info contains a list (called adj) of adjacent vertices
• Example: if the dictionary is implemented as a linked-list
Adjacency lists - Operations

- addVertex(key k): vertices.insert(k, emptyList)
- addEdge(key k, key l):
  vertices.find(k).adj.insert(l)
  vertices.find(l).adj.insert(k)
- areAdjacent(key k, key l):
  return vertices.find(k).adj.find(l)
Data structure for graphs - Adjacency matrix

- Define some order on the vertices, for example: DFW, LAX, LGA, ORD, SFO
- Graph with n vertices is stored as
  - n x n array M of boolean, where
    - \( M[i][j] = \begin{cases} 
    1 & \text{if there is an edge between i-th and j-th vertices} \\
    0 & \text{otherwise} 
    \end{cases} \)
Adjacency matrix - Operations

- **addEdge**(i,j): \[ \text{matrix}[i][j] = 1 \]
- **removeEdge**(i,j): \[ \text{matrix}[i][j] = 0 \]
- Not great for inserting/removing vertices because it requires shifting elements of matrix.
- Requires space \( O(n^2) \)
Lists vs Matrices

• Adjacency lists are better if:
  – You frequently need to add/remove vertices
  – The graph has few edges
  – Need to traverse the graph

• Adjacency matrices are better if
  – you frequently need to
    • add/remove edges, but NOT vertices
    • Check for the presence/absence of an edge between i,j
  – matrix is small enough to fit in memory

In computer science we often compare different solutions to the same problem
Graph traversal - Idea

• Problem:
  – you visit each node in a graph, but all you have to start with is:
    • One vertex A
    • A method getNeighbors(vertex v) that returns the set of vertices adjacent to v
Graph traversal - Motivations

• Applications
  – Exploration of graph not known in advance, or too big to be stored:
    • Web crawling
    • Exploration of a maze
  – Graph may be computed as you go. Example: game strategy:
    • Vertices = set of all configurations of a Rubik's cube
    • Edges connect pairs of configuration that are one rotation away.
Depth-First Search

• Idea: Go Deep!
  – **Intuition**: Adventurous web browsing: always click the first unvisited link available. Click "back" when you hit a dead end.
  – Start at some vertex v
  – Let w be the first neighbor of v that is not yet visited. Move to w.
  – If no such **unvisited** neighbor exists, move back to the vertex that lead to v
Example

unexplored vertex
visited vertex
unexplored edge
discovery edge
Example (cont.)
Algorithm DFS$(G, v)$

**Input:** graph $G$ with no parallel edges and a start vertex $v$ of $G$

**Output:** Visits each vertex once (as long as $G$ is connected)

```plaintext
print $v$  // or do some kind of processing on $v$
$v$.setLabel(VISITED)

for all $u \in v$.getNeighbors()
    if ($u$.getLabel() \neq VISITED) then DFS$(G, u)$
```
The DFS algorithm is similar to a classic strategy for exploring a maze:

- We mark each intersection, corner and dead end (vertex) visited
- We mark each corridor (edge) traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)
DFS and Rubik’s cube

• Rubik’s cube game can be represented as a graph:
  – Vertices: Set of all possible configurations of the cube
  – Edges: Connect configurations that are just one rotation away from each other

• Given a starting configuration $S$, find a path to the “perfect” configuration $P$

• Depth-first search could in principle be used:
  – start at $S$ and making rotations until $P$ is reached, avoiding configurations already visited

• Problem: The graph is huge: $43,252,003,274,489,856,000$ vertices
Running time of DFS

- DFS(G, v) is called once for every vertex v (if G is connected)
- When visiting node v, the number of iterations of the for loop is deg(v).
- Conclusion: The total number of iterations of all for loops is: \( \sum_v \text{deg}(v) = ? \)
  
  Remember the sum of the degrees of all vertices is 2|E|

- Thus, the total running time is \( O(|E|) \)
Applications of variants of DFS

• DFS can be used to:
  – Determine if a graph is connected
  – Determine if a graph contains cycles
  – Solve games single-player games like Rubik’s cube
Breadth-First Search

Idea:
- Explore graph layers by layers
- Start at some vertex \( v \)
- Then explore all the neighbors of \( v \)
- Then explore all the unvisited neighbors of the neighbors of \( v \)
- Then explore all the unvisited neighbors of the neighbors of the neighbors of \( v \)
- until no more unvisited vertices remain
Example

- A: unexplored vertex
- B: visited vertex
- unexplored edge
- discovery edge

V0

V1

L0

L1

L0

L1
Example (cont.)
Example (cont.)

Depth-First Search
Iterative BFS

- Idea: use a queue to remember the set of vertices on the frontier

Algorithm `iterativeBFS(G, v)`

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>graph $G$ with no parallel edges and a start vertex $v$ of $G$</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Visits each vertex once (as long as $G$ is connected)</td>
<td></td>
</tr>
<tr>
<td>$q \leftarrow$ new Queue()</td>
<td>Get the first vertex of the queue, visit it, then add all its unvisited neighbours to the queue</td>
<td></td>
</tr>
<tr>
<td>$v.setLabel(VISITED)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q.enqueue(v)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (! $q.empty()$) do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w \leftarrow s.dequeue$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print $w$ // or do some kind of processing on $w$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for all $u \in w.getNeighbors() do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if ($u.getLabel() \neq VISITED$) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u.setLabel(VISITED)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s.enqueue(u)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Running time and applications

- Running time of BFS: Same as DFS, $O(|E|)$
- BFS can be used to:
  - Find a shortest path between two vertices
    - Rubik’s cube’s fastest solution
  - Determine if a graph is connected
  - Determine if a graph contains cycles
  - Get out of an infinite maze...
Iterative DFS

- Use a stack to remember your path so far

Algorithm \textit{iterativeDFS}(G, v)

Input graph \(G\) with no parallel edges and a start vertex \(v\) of \(G\)

Output Visits each vertex once (as long as \(G\) is connected)

\[
\begin{align*}
&\text{s }\leftarrow \text{new Stack()} \\
&v.\text{setLabel}(\text{VISITED}) \\
&s.\text{push}(v) \\
&\text{while } (! \text{s.\text{empty}()} ) \text{ do} \\
&\quad w \leftarrow \text{s.\text{pop}()} \\
&\quad \text{print } w \\
&\quad \text{for all } u \in w.\text{getNeighbors()} \text{ do} \\
&\quad \quad \text{if } ( \text{u.\text{setLabel}()} \neq \text{VISITED} ) \text{ then} \\
&\quad \quad \quad u.\text{setLabel}(\text{VISITED}) \\
&\quad \quad \quad s.\text{push}(u)
\end{align*}
\]

Note: Code is identical to BFS, but with a stack instead of a queue!

Instead of visiting all of a vertex’s neighbours first, we visit the first neighbour’s neighbours, etc.
Background

Binary numbers
Decimal (base 10)

Digits = \{ 0, 1, 2, 3, 4, 5, 7, 8, 9 \}

Example of numerals: 11, 923, 5548, etc.

\begin{align*}
11 &= 1 \cdot 10^1 + 1 \cdot 10^0 \\
923 &= 9 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 \\
5548 &= 5 \cdot 10^3 + 5 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0
\end{align*}

\[ m = \sum_{i=0}^{\text{digit}} d[i] \cdot 10^i \]
Binary (base 2)

Bits = \{ 0, 1 \}

Example of numerals: 11, 101, 1010, etc.

\[
11 = 1 \cdot 2^1 + 1 \cdot 2^0 \\
101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
1010 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0
\]

\[
m = \sum_{i=0}^{\infty} b[i] \cdot 2^i
\]

Storing one of two digits requires one “slot” of memory (bit)
## Relationship

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
</tbody>
</table>
Fixed size representation

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
</tbody>
</table>

Fixed number of bits (typically 8, 16, 32, 64...).

8 bits is called “byte”.

It makes sense to assign a fixed space in memory to each number.
Conversion

• How to convert from binary to decimal?

\[(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0\]
\[= 16 + 8 + 0 + 2 + 0\]
\[= 26\]

Note: You need to know powers of 2...

• How to convert from decimal to binary?

\[(241)_{10} = ???\]
Operations on decimals

Use this property of any positive integer \( m \):

\[
m = \left( \frac{m}{10} \right) \times 10 + m \mod 10
\]

Ex: \( 238 = 23 \times 10 + 8 \)

• (integer) division by 10 = dropping rightmost digit

Ex: \( 238/10 = 23 \)

• Multiplication by 10 = shifting left by one digit

Ex: \( 23 \times 10 = 230 \)

• Remainder of integer division by 10 = rightmost digit

Ex: \( 238 \mod 10 = 8 \)
Operations on binary

Same property holds for binary:

\[ m = m \% 2 + (m/2) \times 2 \]

Example:

\[
\begin{align*}
m & = (1011)_2 \\
m/2 & = (0101)_2 \\
(m/2) \times 2 & = (1010)_2 \\
m \% 2 & = (0001)_2
\end{align*}
\]
Aside: bit shift

- Sometimes we need to move bits from left to right/right to left

\[
23 \times 10 = 230
\]
\[
m/2 = (0101)_2
\]
\[
(m/2) \times 2 = (1010)_2
\]

- We can do it arithmetically by multiplying or integer dividing by the base (10 in decimal, 2 in binary, etc)

- We may want to shift the bits of the binary representation of a number (in memory, all numbers are in binary).

- A left-shift is represented by the operator \(<<\), and a right shift is represented by the operator \(>>\). The operator is typically followed by the number of bits to shift by.
Aside: bit shift

• Example: 00001110 << 3 = 01110000

• Bit shifts can lead to loss of information if you reach the “end” of the number’s allocated memory.
• Example: 00001110 >> 3 = 00000001

• In practice, things are a bit more complicated because we have to deal with sign bits, so there is a difference between logical shift and arithmetic shift.
• For the purpose of this class, let’s only consider bit shifting in the context of positive integers. In this context, the two types of shifting are equivalent.
Decimal$\rightarrow$Binary (Algorithm)

**Algorithm** `decimal2binary(m)`

**Input:** a decimal $m$

**Output:** a binary $b$

i$\leftarrow$0

**while** $m>0$ **do**

- $b[i] \leftarrow m\%2$
- $m \leftarrow m/2$
- $i \leftarrow i+1$

We can fill the whole array of bits by computing the “remainder” for each bit
Decimal $\rightarrow$ Binary (Example)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$m/2$</th>
<th>$m%2$ ($b[i]$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Answer:**

$b[i] = \ldots011110001$
Why is the algorithm working?

\[ m = m/2 \times 2 + m \% 2 \]

Remember what we did with the decimal number

Divide by 10, shift left, add the remainder.

This is the EXACT same thing, but because we are not used to it, it feels like magic.
## Additions

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+1=1</td>
<td>0+1=1</td>
</tr>
<tr>
<td>1+1=2</td>
<td>1+1=10</td>
</tr>
<tr>
<td>1+2=3</td>
<td>1+10=11</td>
</tr>
</tbody>
</table>
Additions

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>11010</td>
</tr>
<tr>
<td>+ 15</td>
<td>+ 01111</td>
</tr>
</tbody>
</table>
| = 41    | = ?????
Addition in binary

\[
\begin{array}{cccccc}
1 & 1 & 0 & 1 & 0 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 & 1 \\
\hline
? & ? & ? & ? & \color{red}0 & 1 \\
\end{array}
\]
Addition in binary

\[
\begin{array}{cccccc}
1 & 1 & 0 & 1 & 0 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 & 1 \\
\hline
= & ? & ? & ? & 1 & 0 & 0 & 1 \\
\end{array}
\]
Addition in binary

\[
\begin{array}{ccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 & 1 \\
\hline
= & ? & ? & 1 & 0 & 0 & 1 \\
\end{array}
\]
Addition in binary

\[
\begin{array}{cccccc}
  & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 \\
\hline
= & ? & 1 & 1 & 0 & 0 & 1
\end{array}
\]
Addition in binary

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\hline \\
= & 1 & 0 & 1 & 1 & 0 & 0 & 1
\end{array}
\]
Addition in binary

\[
\begin{array}{c}
\begin{array}{cccccc}
1 & 1 & 0 & 1 & 0 \\
+ & 0 & 0 & 1 & 1 & 1 \\
\hline
= & 1 & 0 & 1 & 0 & 0 & 1
\end{array}
\end{array}
= 26
\]

\[
= 15
\]

\[
= 41
\]

\[
\begin{array}{cccccc}
2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
1 & 0 & 1 & 0 & 0 & 1
\end{array}
= 2^5 + 2^3 + 2^0 = 32 + 8 + 1 = 41
\]
Operation in binary

Recall grade-school algorithm for addition, subtraction, multiplication, and division.

There is nothing special about base 10.

*These algorithms work for binary (base 2), and work for other bases too!*
Representation size

\[ m = \sum_{i=0}^{N-1} b[i] \times 2^i \]

What is the relationship between \( m \) and \( N \)?

(How many bits \( N \) do we need to represent a positive integer \( m \)?)
Lower bound

\[ \sum_{i=0}^{N-1} 2^i = 1 + 2 + 4 + \cdots + 2^{N-1} = 2^N - 1 \]

(This is a special case of \( \sum_{i=0}^{N-1} x^i = \frac{x^{N-1}}{x-1} \) where \( x = 2 \))

\[ m = \sum_{i=0}^{N-1} b[i] \cdot 2^i \leq \sum_{i=0}^{N-1} 1 \cdot 2^i \]
\[ = 2^N - 1 \]
\[ < 2^N \]

Remember: whenever we split a problem in two at each step, we get something that grows with \( \log N \)

\[ \log_2 m < N \] (apply log on both sides)
Upper bound

We can assume that $N - 1$ is the index $i$ of the leftmost bit $b[i]$ such that $b[i] = 1$ (we ignore leftmost 0’s: 00001101).

A binary number $\geq$ its leftmost bit.

$$m = \sum_{i=0}^{N-1} b[i] \cdot 2^i \geq 2^{N-1}$$

$$\log_2 m \geq N - 1$$

$$\log_2 m + 1 \geq N$$
How many bit do we need?

\[ \log_2 m < N \leq (\log_2 m) + 1 \]

**Answer:** The largest integer less than or equal to \((\log_2 m) + 1\).

We write it as \(N = \lfloor(\log_2 m) + 1\rfloor\) (a.k.a. ”floor” that means “round down”)
## Examples

<table>
<thead>
<tr>
<th>$m$ (decimal)</th>
<th>$m$ (binary)</th>
<th>$N = \lceil \log_2 m \rceil + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>4</td>
</tr>
</tbody>
</table>
To think about...

• How are negative integers represented?

• How many bits are used to represent int, short, long in a computer?

• How are non-integers (fractional numbers) represented?

• How are characters represented?
Background

Expectation & Indicators
Expectation

• **Average or mean**

• The expected value of a discrete random variable $X$ is $E[X] = \sum_x x \Pr\{X=x\}$

• **Linearity of Expectation**
  
  
  – $E[aX+Y] = a \ E[X] + E[Y]$, for constant $a$ and all $X, Y$

• For **mutually independent random variables** $X_1,\ldots, X_n$
  
  – $E[X_1X_2 \ldots X_n] = E[X_1] \cdot E[X_2] \cdot \ldots \cdot E[X_n]$
Expectation – Example

- Let $X$ be the RV denoting the value obtained when a fair die is thrown. What will be the mean of $X$, when the die is thrown $n$ times.
  - Let $X_1, X_2, ..., X_n$ denote the values obtained during the $n$ throws.
  - The mean of the values is $\frac{X_1+X_2+...+X_n}{n}$.
  - Since the probability of getting values 1 to 6 is $\frac{1}{6}$ in average, we can expect each of the 6 values to show up $(1/6)n$ times.
  - So, the numerator in the expression for mean can be written as $(1/6)n\cdot1+(1/6)n\cdot2+...+(1/6)n\cdot6$
  - The mean, hence, reduces to $(1/6)\cdot1+(1/6)\cdot2+...(1/6)\cdot6$, which is what we get if we apply the definition of expectation.
Indicator Random Variables

- A simple yet powerful technique for computing the expected value of a random variable.
- Convenient method for converting between probabilities and expectations.
- Helpful in situations in which there may be dependence.
- Takes only 2 values, 1 and 0.
- **Indicator Random Variable for an event** $A$ of a sample space is defined as:

$$I\{A\} = \begin{cases} 
1 & \text{if } A \text{ occurs,} \\
0 & \text{if } A \text{ does not occur.}
\end{cases}$$
**Lemma 5.1**

Given a sample space $S$ and an event $A$ in the sample space $S$, let $X_A = I\{A\}$. Then $E[X_A] = Pr\{A\}$.

**Proof:**
Let $\bar{A} = S - A$ (Complement of $A$)

Then,

$$E[X_A] = E[I\{A\}]$$

$$= 1 \cdot Pr\{A\} + 0 \cdot Pr\{\bar{A}\}$$

$$= Pr\{A\}$$
Important dates

• First assignment will come out in late September
• Assignments due roughly every two weeks in October and November.
• Office hours start next week (see course website)
• You will get a break for the midterm (late October)
• Tutorials will come out late September