COMP251: Graphs, Probability and Binary numbers

Giulia Alberini & Jérôme Waldispühl

School of Computer Science

McGill University

Announcements

- Office hours started this week (see course website)

- Tutorials started this week:
  Mon + Tue in MC103

Outline

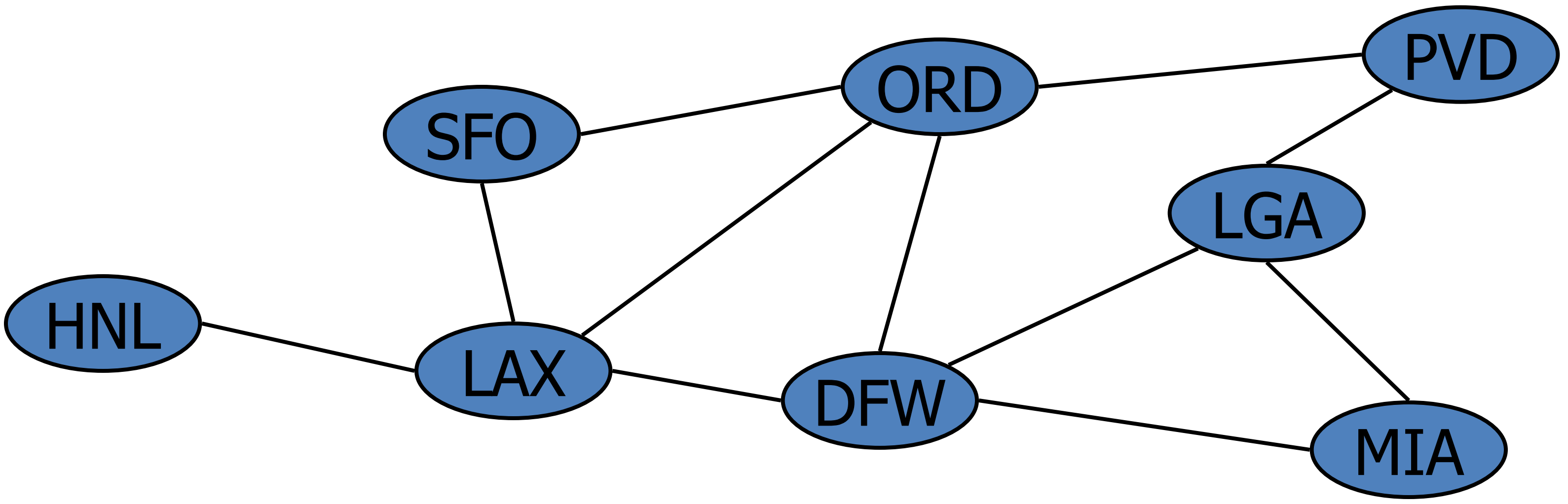
- Graphs
  - Terminology, definitions and properties
  - Graph traversal: Depth-First Search and Breadth-first search
- Binary numbers
- Probability

Background

Graphs

Graph

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called 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 $(\boldsymbol{u},\boldsymbol{v})$
  - first vertex $\boldsymbol{u}$ is the origin
  - second vertex $\boldsymbol{v}$ is the destination
  - e.g., a flight

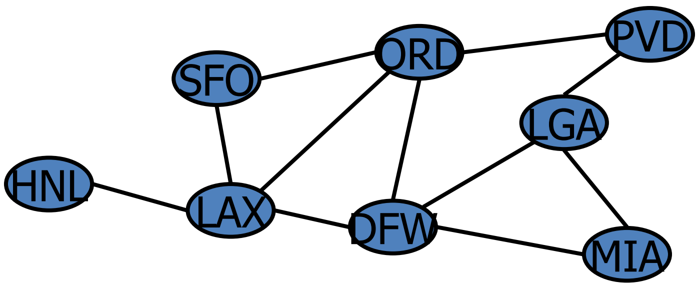
- Undirected edge
  - unordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$
  - e.g., a street

- 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

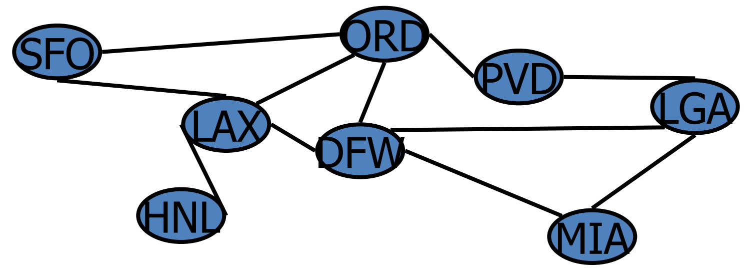
- Weighted graph: all edges have weights

Labeled graphs

- Labeled graphs: vertices have identifiers

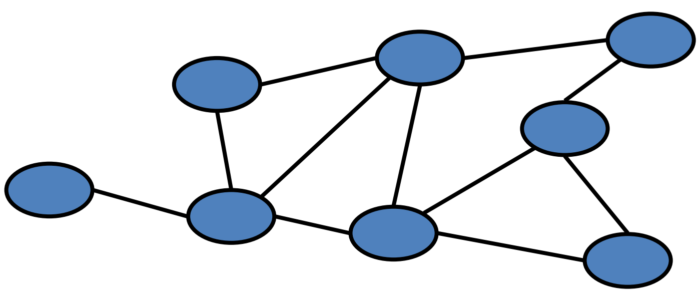


=

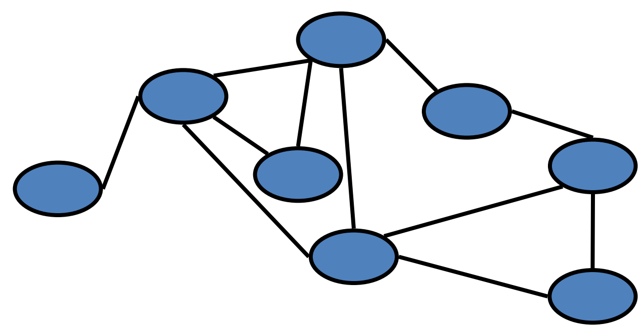


  – Note: Geometric layout doesn't matter - only connections matter

- Unlabeled graph: vertices have no identifiers

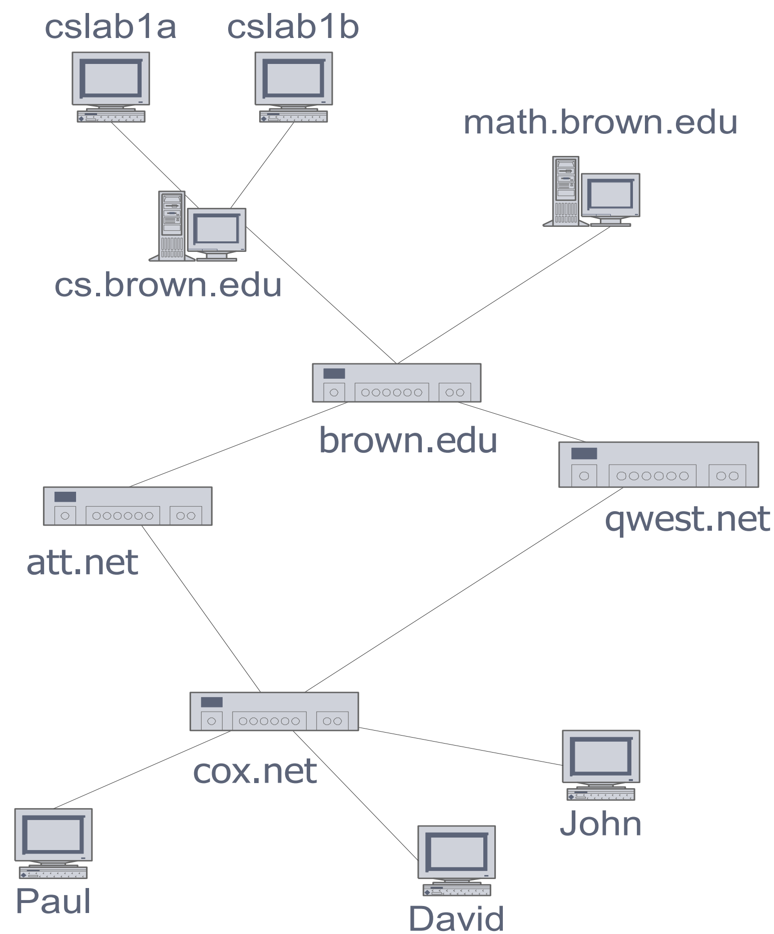


=



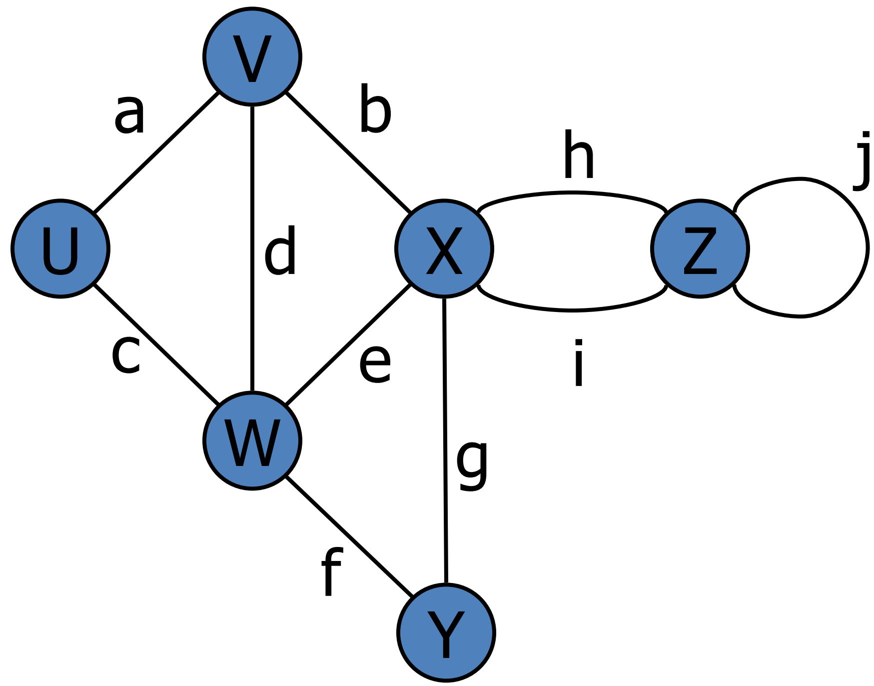
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



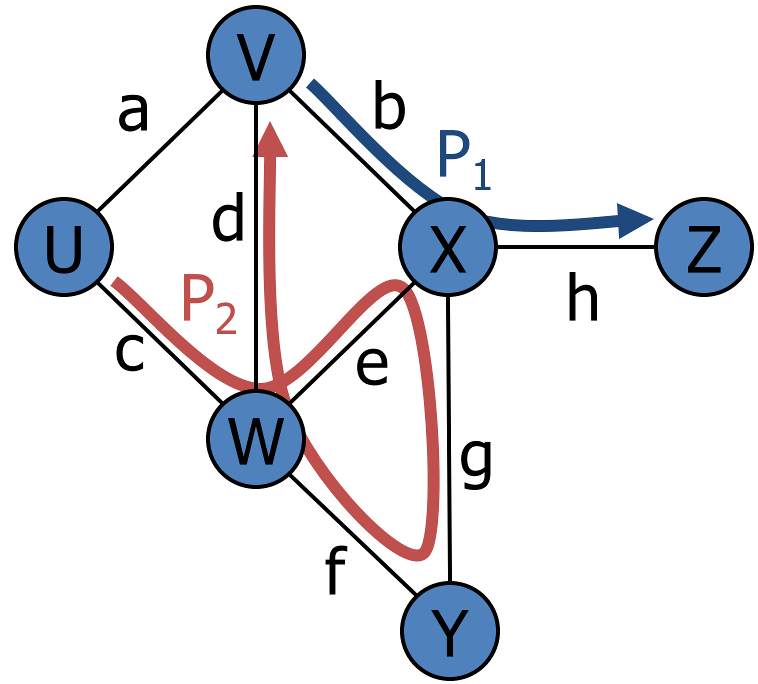
Terminology

- 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



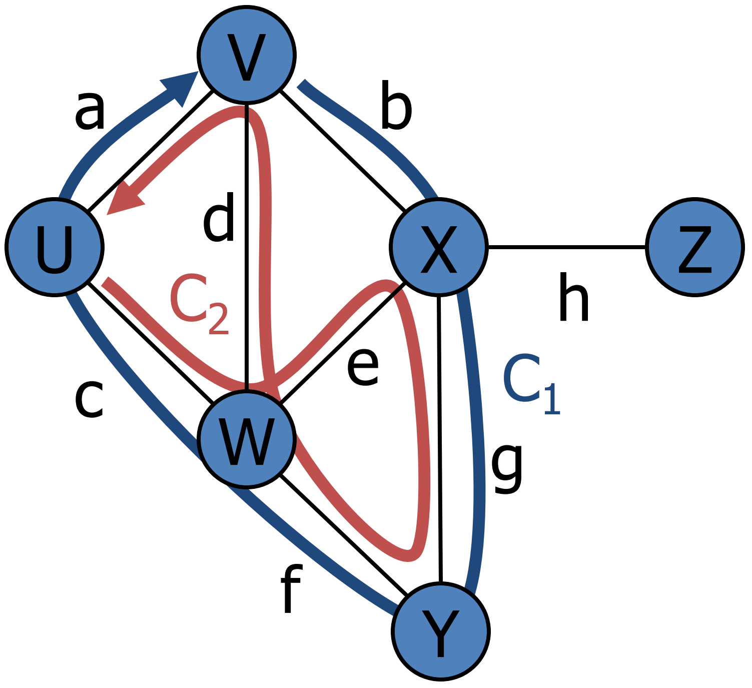
Terminology (cont.)

- 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, ↵) is a simple cycle
  - C_2=(U, W, X, Y, W, V, ↵) is a cycle that is not simple
- A tree is a connected acyclic graph



Properties

Property 1

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

Why? The sum of the degrees of all vertices is 2*number of edges

Notation

$\lvert \mathbf{V} \rvert$	number of vertices
$\lvert \mathbf{E} \rvert$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$

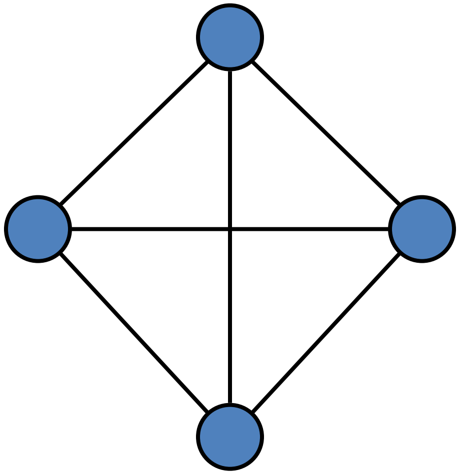
Property 2

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

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

Why?

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

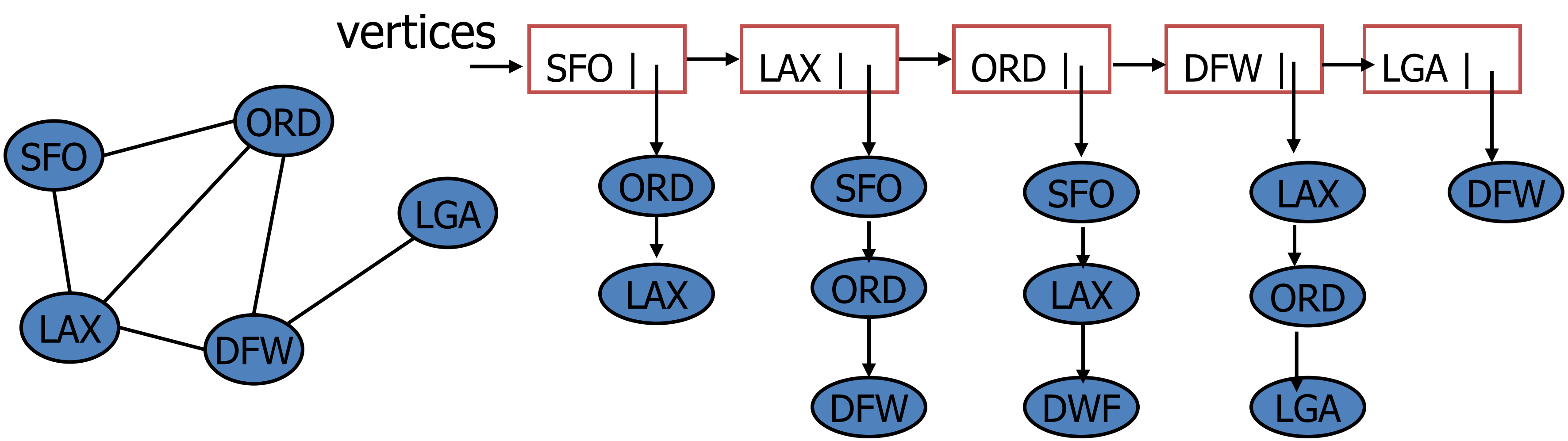


Example

- $|\mathbf{V}| = 4$
- $|\mathbf{E}| = 6$
- $\deg(\boldsymbol{v}) = 3$

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):

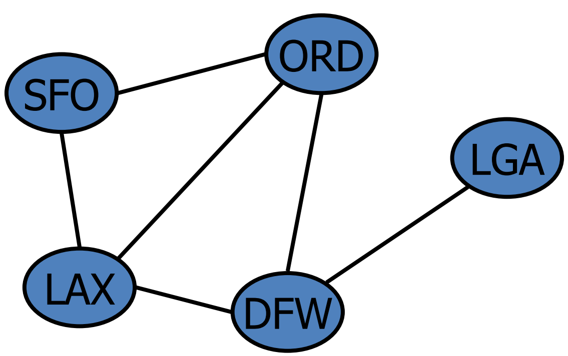
  vertices.find(k).adj.insert(l)

  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
  - $M[i][j] = \begin{cases} 1 \text{ if there is an edge between i-th and j-th vertices} \\ 0 \text{ otherwise} \end{cases}$



	DFW	LAX	LGA	ORD	SFO
DFW	0	1	1	1	0
LAX	1	0	0	1	1
LGA	1	0	0	0	0
ORD	1	1	0	0	1
SFO	0	1	0	1	0

Adjacency matrix - Operations

- addEdge(i,j): matrix[i][j] = 1
- removeEdge(i,j): matrix[i][j] = 0
- Not great for inserting/removing vertices because it requires shifting elements of matrix.
- Requires space $O(n^2)$

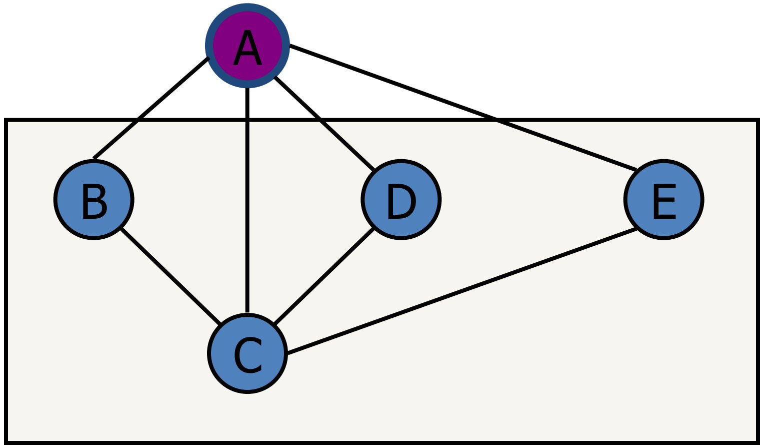
Lists vs Matrices

- 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



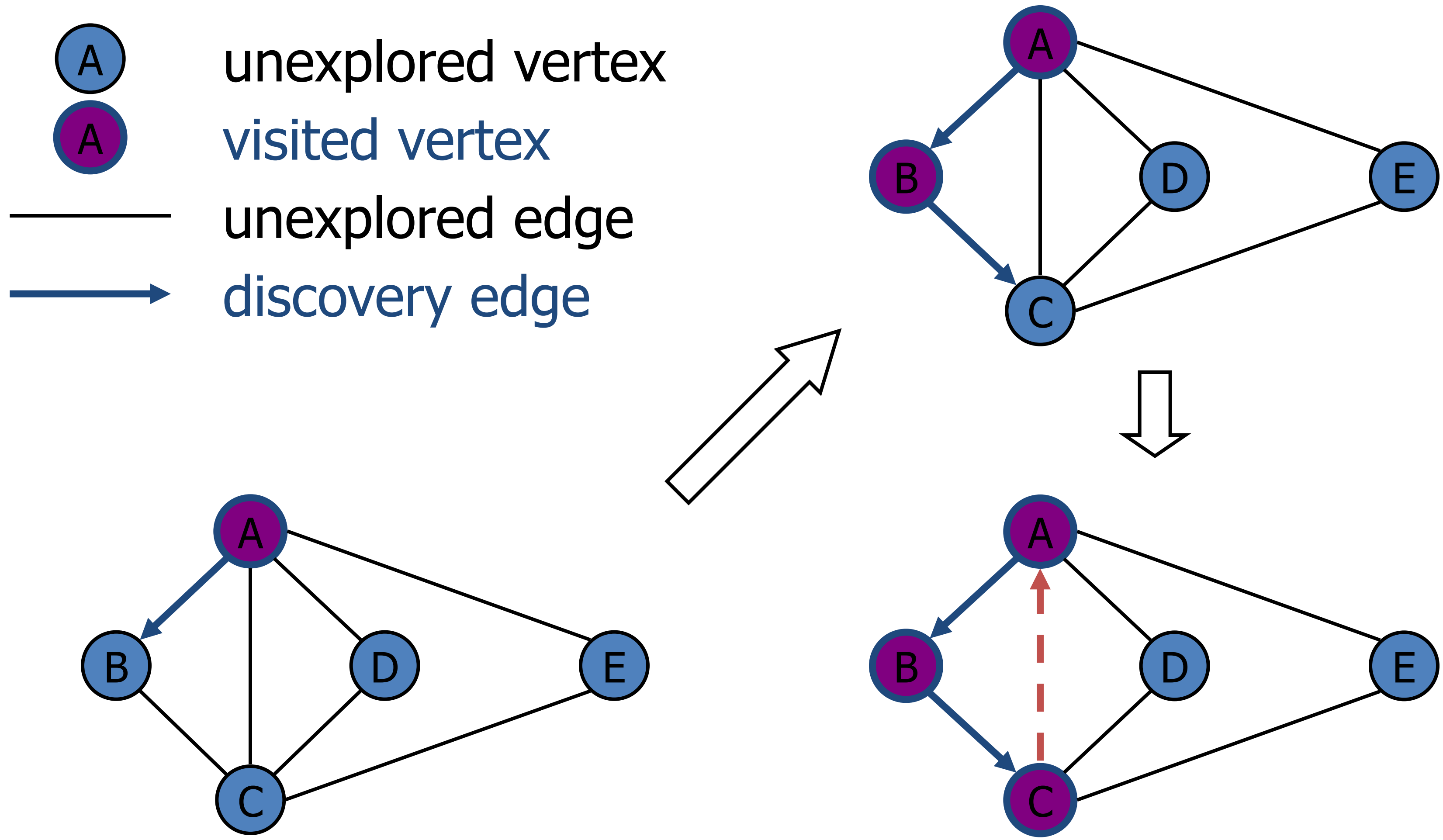
Graph traversal - Motivations

- 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.

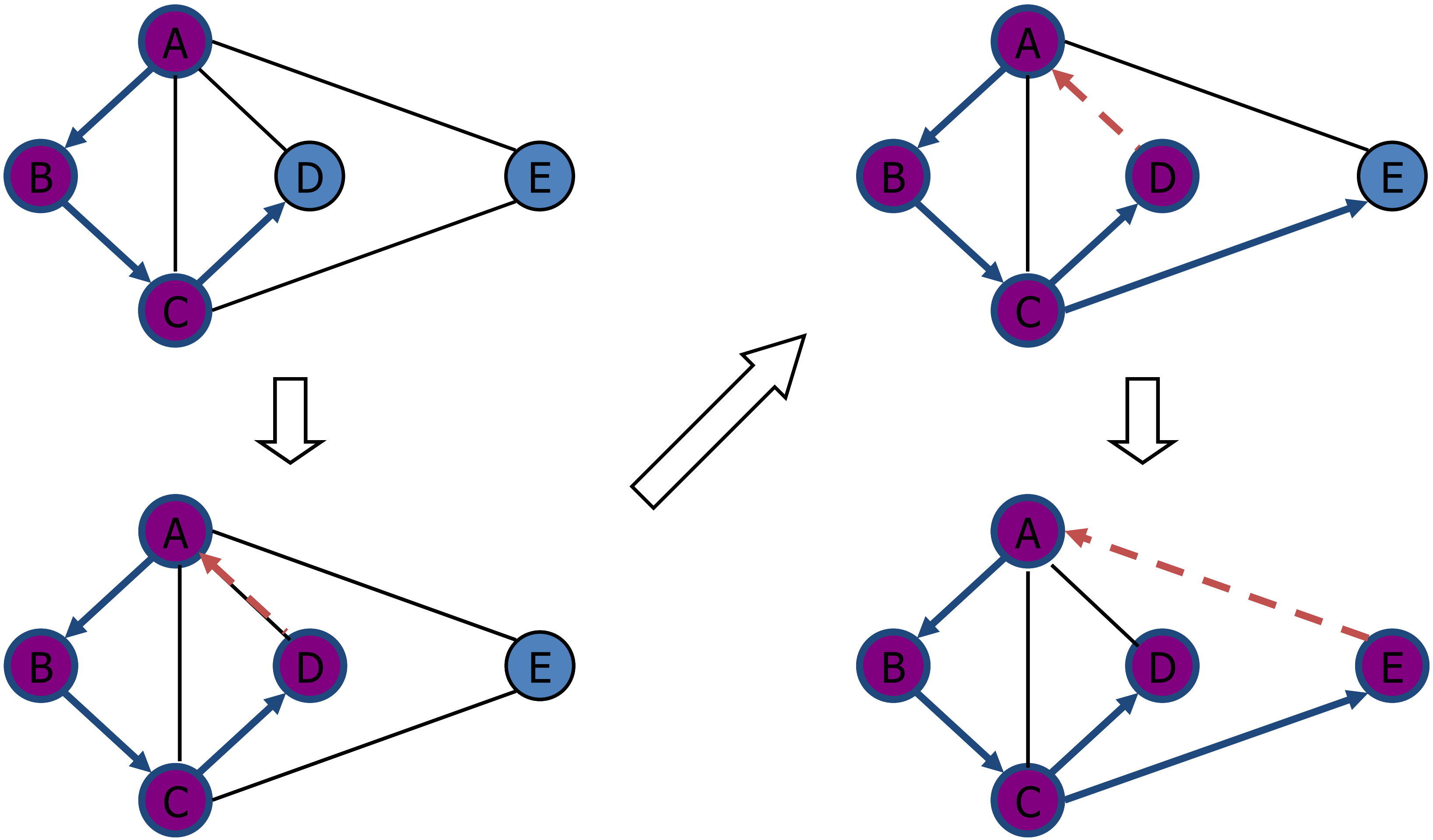
Depth-First Search

- 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


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
A
B
C
D
E
A
B
C
D
E
A
B
C
D
E

Example (cont.)



DFS Algorithm

Algorithm ***DFS***(***G, v***)

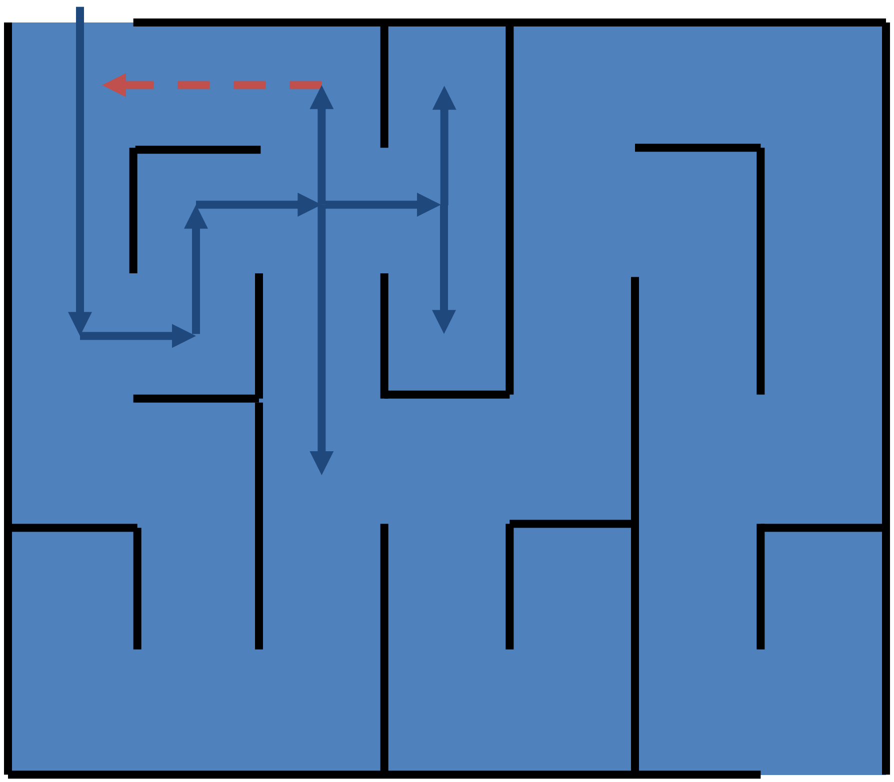
Input: graph ***G*** with no parallel edges and a start vertex ***v*** of ***G***

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

```
print v              // or do some kind of processing on v
v.setLabel(VISITED)
  for all u ∈ v.getNeighbors()
    if ( u.getLabel() != VISITED ) then DFS(G, u)
```

DFS and Maze Traversal

- 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: $\Sigma_v \deg(v) = ?$

Remember the sum of the degrees of all vertices is 2|E|

- Thus, the total running time is O(|E|)

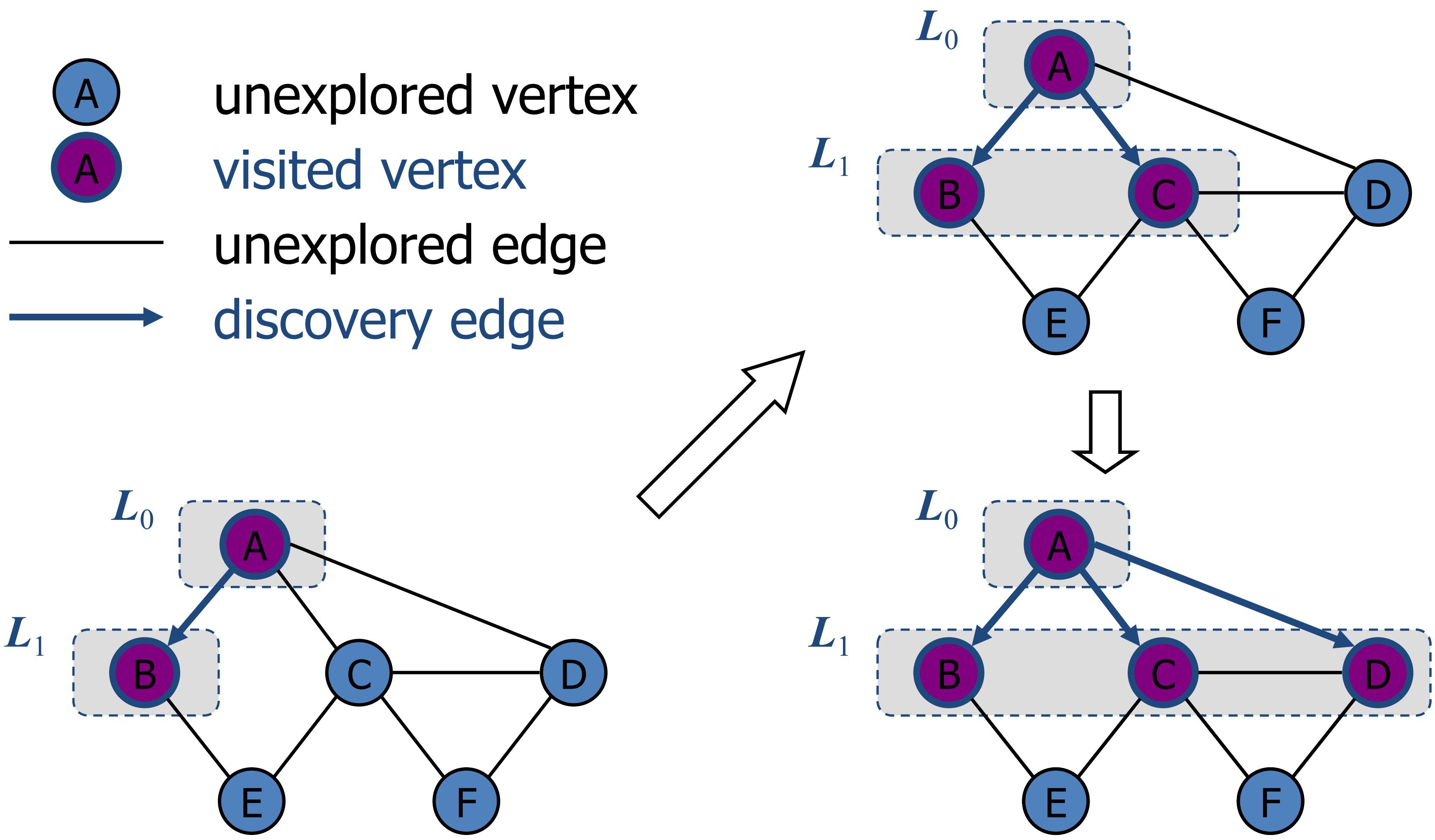
Applications of variants of DFS

- 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

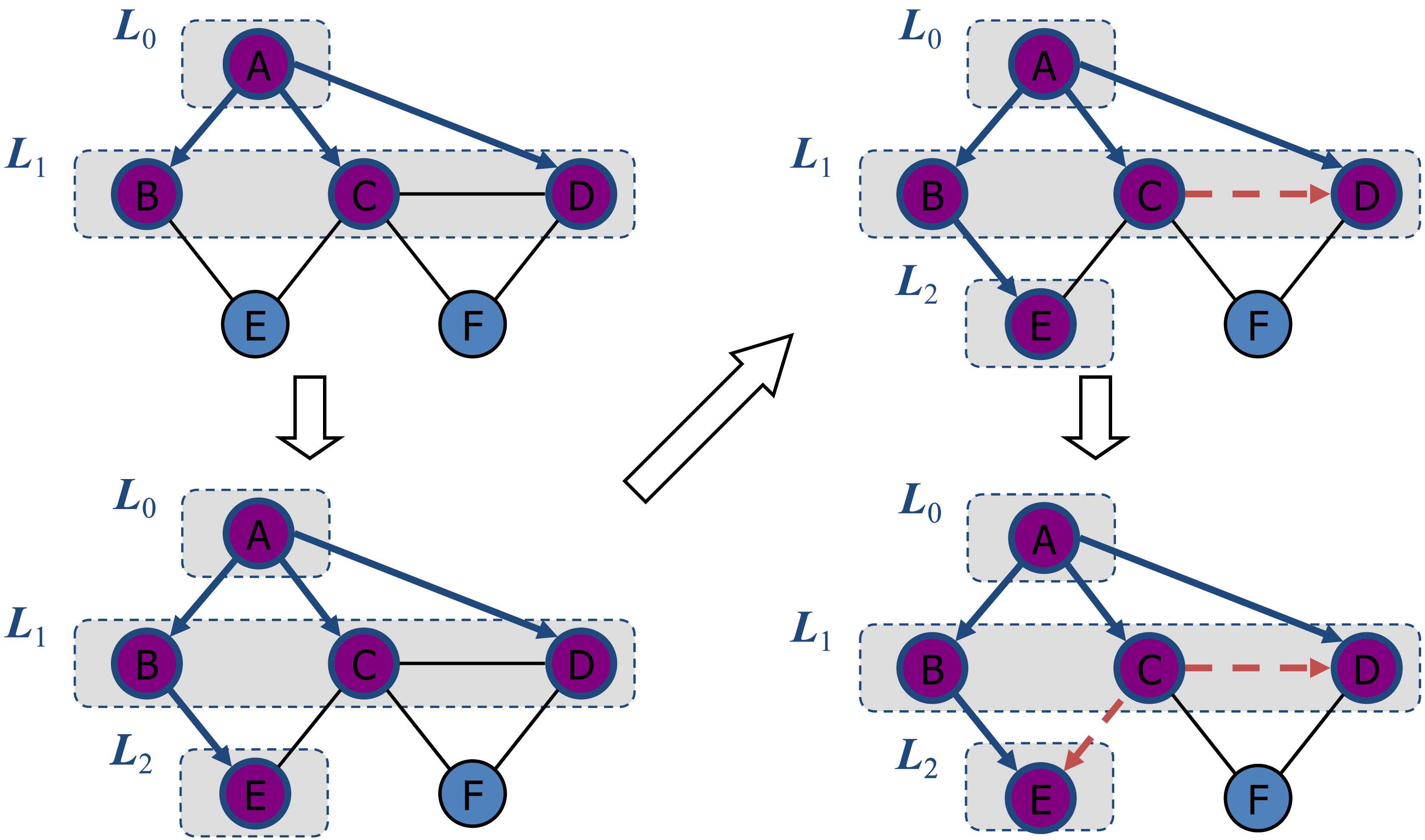
Breadth-First Search

- 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

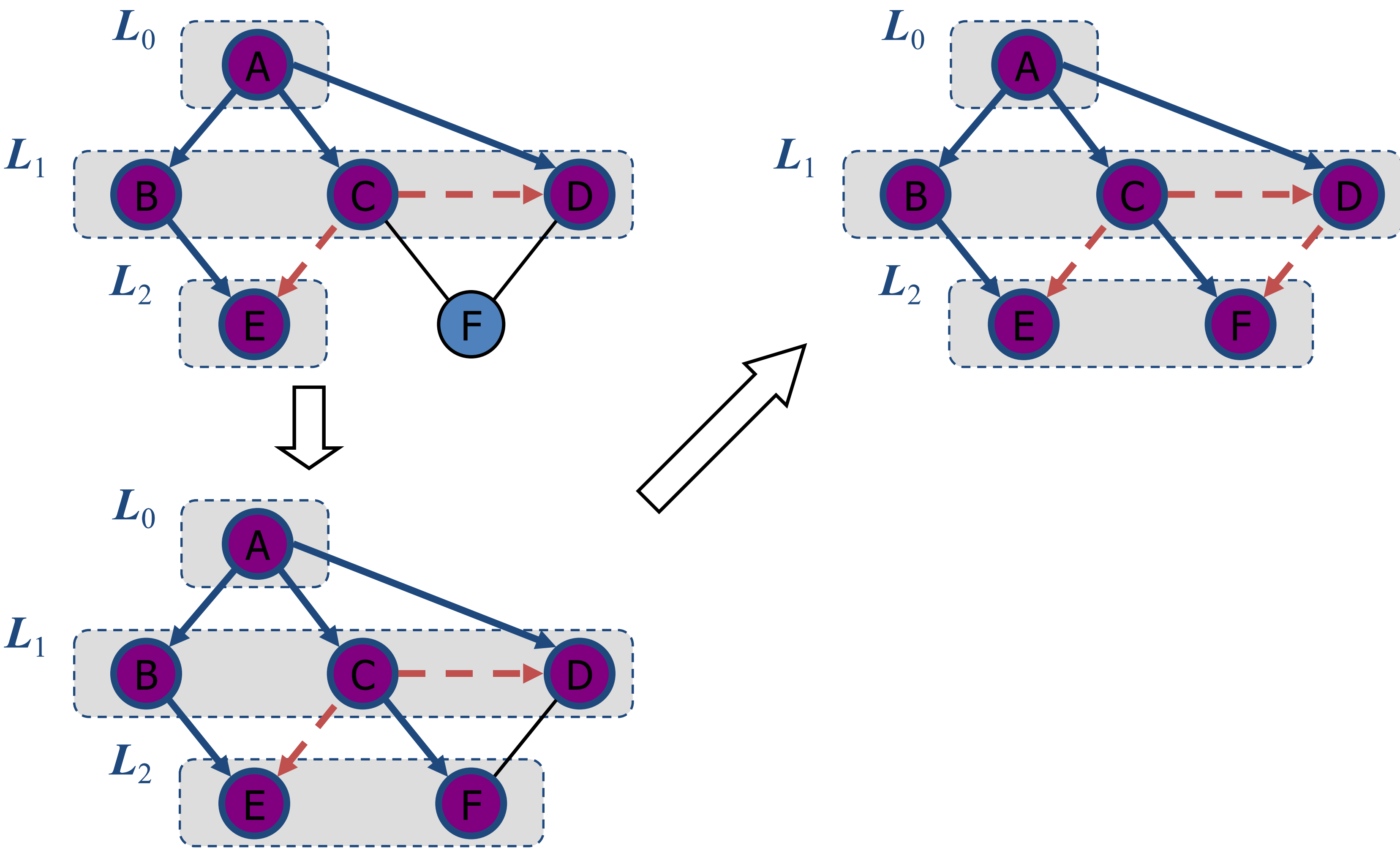
Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F

Example (cont.)



Example (cont.)



Iterative BFS

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

```
Algorithm iterativeBFS(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)
  q ← new Queue()
  v.setLabel(VISITED)
  q.enqueue(v)
  while (! q.empty()) do
    w ← s.deque()
    print w          // or do some kind of processing on w
    for all  u ∈ w.getNeighbors() do
      if  ( u.getLabel() != VISITED ) then
        u.setLabel(VISITED)
        s.enqueue(u)
```

Get the first vertex of the queue, visit it, then add all its unvisited neighbours to the queue

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 ***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)

```
s ← new Stack()
v.setLabel(VISITED)
s.push(v)
while (! s.empty()) do
    w ← s.pop()
    print w
    for all u ∈ w.getNeighbors() do
        if ( u.getLabel() != VISITED ) then
            u.setLabel(VISITED)
            s.push(u)
```

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

- The base we use every day
- It contains ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9
- Counting in base 10:
  - Start counting: 0… 1… 2… 3… 4… 5… 6… 7… 8… 9…
  - We're out of digits
  - Add a second column worth 10 times the value of the first
  - Continue counting: 10… 11… 12… and so on.

Decimal - Base 10

When we refer to a decimal (base 10) number, like 5764, we are referring to the value obtained by carrying out the following addition:

$$5000 + 700 + 60 + 4$$

That is, we add together:

- Ones: 4
- Tens: 6
- Hundreds: 7
- Thousands: 5

Decimal - Base 10

$$\mathbf{5764 = 5 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0}$$

NOTE:

- The digits of the number correspond to the coefficients of the powers of ten
- The position of the digit in the number determines to which power it is associated.

In a similar way, we can write numbers in other bases (beside 10):

- We use the digits that correspond to the coefficients on the corresponding powers (of the given base)

Given a base b

$$(m)_{10} = \sum_{i=0} d_i * b^i$$

digit in position i of m's representation in base b

NOTATION: We write

$$(n)_b$$

to denote that the number n is written in base b.

Example

What decimal number does $(132)_5$ represent ?

- First compute the corresponding powers of 5:

$$5^0 = 1$$
$$5^1 = 5$$
$$5^2 = 25$$

- Then multiply them by the corresponding digit and sum the results together:

$$1 \cdot 5^2 + 3 \cdot 5^1 + 2 \cdot 5^0 = 25 + 15 + 2 = 42$$

Binary – base 2

Exactly the same as base 10, except

- It contains only two digits: 0 and 1
- Counting in binary
  - Start counting: 0… 1…
  - We're out of digits.
  - Add a second column this time worth **2** times the value of the first
  - Continue counting: 10… 11…

Binary to Decimal

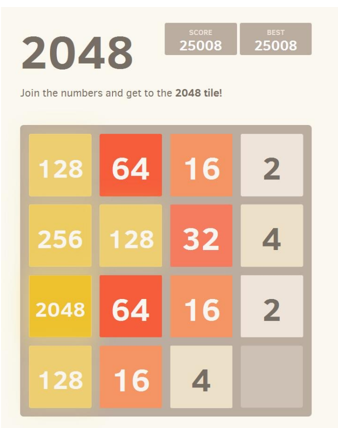
- Given the following binary representation $(a_k a_{k-1} \dots a_1 a_0)_2$ for a number, then its decimal value m is equal to

$$a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2^1 + a_0 2^0 = \sum_{i=0}^{k} a_i\, 2^i$$

- NOTE: $a_i = 0$ or 1.

  Thus only the terms with the $a_i = 1$ will be left to sum!

Binary to Decimal - Algorithm



Given a binary number, do:

- Compute the powers of 2 needed
  (as many as the binary digits in the number)

- Identify the powers associated to the digits equal to 1

- Sum them all together.

x	2^x
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048

Quotient-Remainder Theorem

Given any integer m and a positive integer d, there exist unique integers q and r such that

$$m = q \cdot d + r$$

where $0 \leq r < d$.

To compute the quotient (q) and the remainder (r) we can simply use integer division and modulo operator, respectively.

Operations on decimals

So, for any positive integer m the following holds

$$m = (m/10) \ * 10 + \ m \ \% \ 10$$

Ex: 238 = 23 * 10 + 8

- (integer) division by 10 = dropping rightmost digit

  Ex: 238/10 = 23

- Multiplication by 10 = shifting left by one digit

  Ex: 23*10 = 230

- Remainder of integer division by 10 = rightmost digit

  Ex: 238%10 = 8

Operations on binary

The same properties can be observed when using 2 as the dividend and looking at the binary representation of the number.

Recall that for any positive integer m:

$$m = (m/2) * 2 + m \,\%\, 2$$

Example:

$$\begin{aligned} m &= (1011)_2 \\ m/2 &= (0101)_2 \\ (m/2) * 2 &= (1010)_2 \\ m \,\%\, 2 &= (0001)_2 \end{aligned}$$

Decimal to Decimal

What is $(5764)_{10}$ in decimal notation (base 10)?

$$5764/10 = 576\ R\ 4$$

$$576/10 = 57\ R\ 6$$

$$57/10 = 5\ R\ 7$$

$$5/10 = 0\ R\ 5$$

Note that taking the remainders from bottom to top gives us the answer.

Decimal to Binary

What is $(13)_{10}$ in binary?

$$13/2 = 6\ R\ \mathbf{1}$$

$$6/2 = 3\ R\ \mathbf{0}$$

$$3/2 = 1\ R\ \mathbf{1}$$

$$1/2 = 0\ R\ \mathbf{1}$$

Now, the base 2 representation comes from reading off the <u>remainders</u> from bottom to top!

$$13_{10} = \mathbf{1101_2}$$

Algorithm – base conversion

The technique just shown works in every base.

In general, given a base b and a decimal number m, repeat the following until the number is 0:

- Divide m by b and prepend the remainder of the division.
- Let the new number be m divided by b, rounded down.

ALGORITHM
Constructing Base b Expansions

procedure *BaseExpansion*(n, b)
$q := n$
$k := 0$
While $q \neq 0$
    $a_k := q \bmod b$
    $q := q/b$
    $k := k + 1$
return $(a_{k-1}, \ldots, a_1, a_0)$

Why is the algorithm working?

$$m = \underbrace{\underbrace{\underbrace{m/2}_{(\ldots b[3]b[2]b[1])_2} * 2}_{(\ldots b[3]b[2]b[1]0)_2} + \underbrace{m \,\%\, 2}_{(b[0])_2}}_{(\ldots b[3]b[2]b[1]b[0])_2}$$

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.

Relationship

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Fixed size representation

Decimal	Binary
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

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

Aside: bit shift

$$23*10 = 230$$

$$\begin{aligned} m/2 &= (0101)_2 \\ (m/2) * 2 &= (1010)_2 \end{aligned}$$

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

- 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.

Additions

Decimal	**Binary**
0+1=1	0+1=1
1+1=2	1+1=10
1+2=3	1+10=11

Additions

Decimal

$$\begin{array}{rr} & 26 \\ + & 15 \\ \hline = & 41 \end{array}$$

Binary

$$\begin{array}{rr} & 11010 \\ + & 01111 \\ \hline = & ????? \end{array}$$

Addition in binary

$$\begin{array}{cccccc} & 1 & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 & 1 \\ \hline = & ? & ? & ? & ?_{\color{red}{0}} & 1 \end{array}$$

Addition in binary

$$
\begin{array}{cccccc}
 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 \\
\hline
= & ? & ? & ?_{\color{red}1} & 0_{\color{red}0} & 1
\end{array}
$$

Addition in binary

$$
\begin{array}{cccccc}
 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 \\
\hline
= & ? & ?_{\color{red}{1}} & 0_{\color{red}{1}} & 0_{\color{red}{0}} & 1
\end{array}
$$

Addition in binary

$$
\begin{array}{cccccc}
 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 1 \\
\hline
= & ?_{1} & 1_{1} & 0_{1} & 0_{0} & 1
\end{array}
$$

Addition in binary

$$\begin{array}{cccccc} & & 1 & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline = & 1 & 0_{\color{red}1} & 1_{\color{red}1} & 0_{\color{red}1} & 0_{\color{red}0} & 1 \end{array}$$

Addition in binary

$$
\begin{array}{ccccccccc}
 & & 1 & 1 & 0 & 1 & 0 & = 26 \\
+ & 0 & 0 & 1 & 1 & 1 & 1 & = 15 \\
\hline
= & 1 & 0 & 1 & 0 & 0 & 1 & = 41
\end{array}
$$

$$
\frac{2^5}{1}\ \frac{2^4}{0}\ \frac{2^3}{1}\ \frac{2^2}{0}\ \frac{2^1}{0}\ \frac{2^0}{1} = 2^5 + 2^3 + 2^0 = 32 + 8 + 1 = 41
$$

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

$$m = \sum_{i=0}^{N-1} b_i * 2^i$$

What is the relationship between m and N ?

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

Geometric Series

Recall that,

$$\sum_{i=0}^{N-1} x^i = 1 + x + x^2 + x^3 + \cdots + x^{N-1} = \frac{x^N - 1}{x - 1}$$

That is, if $x = 2$,

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1$$

Lower Bound

$$\begin{aligned} m &= \sum_{i=0}^{N-1} b_i \cdot 2^i \\ &\leq \sum_{i=0}^{N-1} 1 \cdot 2^i \\ &= 2^N - 1 \\ &< 2^N \end{aligned}$$

Thus,

$$m < 2^N$$

To solve for N, we take the log (base 2) of both sides and obtain the following equation:

$$\boxed{N > \log_2 m}$$

Lower bound

Upper Bound

Now, let's assume that $N-1$ is the index i of the leftmost bit b_i such that $b_i \ = \ 1.$

e.g. We ignore leftmost 0's of the binary representation of m, $(\ldots 00000010011)_2$

Then,

$$m \ = \sum_{i=0}^{N-1} b_i 2^i \ = 1 \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i \ \geq 2^{N-1}$$

Taking the log (base 2) of both sides,

$$\log_2 m \geq N-1 \qquad \Rightarrow \qquad \boxed{N \leq (\log_2 m) + 1}$$

Upper Bound

How many bits do we need?

We proved that,

$$\log_2 m < N \leq (\log_2 m) + 1$$

Thus, N must be equal to the largest integer less than or equal to $(\log_2 m) + 1$. We write,

$$N = floor\big((\log_2 m) + 1\big) = \lfloor(\log_2 m) + 1\rfloor$$

where $floor$ means "round down to the nearest integer".

Examples

m (decimal)	m (binary)	$N = \lfloor(\log_2 m) + 1\rfloor$
0	0	-
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4

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
  - $E[X+Y] = E[X]+E[Y]$, for all X, Y
  - $E[aX+Y] = a\,E[X] + E[Y]$, for constant a and all X, Y
- For mutually independent random variables $X_1, \ldots, X_n$
  - $E[X_1 X_2 \ldots X_n] = E[X_1] \cdot E[X_2] \cdot \ldots \cdot E[X_n]$

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 $(X_1+X_2+\ldots+X_n)/n$.
  - Since the probability of getting values 1 to 6 is (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 $(1/6)n\cdot1+(1/6)n\cdot2+\ldots+(1/6)n\cdot6$
  - The mean, hence, reduces to $(1/6)\cdot1+(1/6)\cdot2+\ldots(1/6)\cdot6$, 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:

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

Indicator Random Variable

Lemma 5.1

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

Proof:

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

Then,

$$\begin{aligned}\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\}\end{aligned}$$