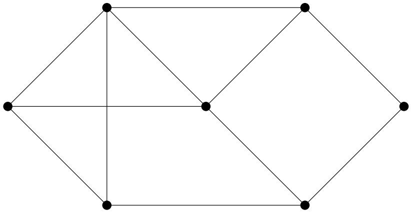
Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.

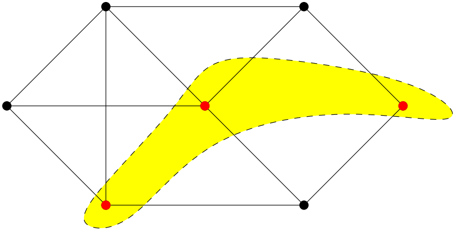
Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.



Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.



Beyond Polynomial Time: 2^n

Maximum Independent Set Problem

Input: graph $G = (V, E)$

Output: the maximum independent set of G

Beyond Polynomial Time: 2^n

Maximum Independent Set Problem

Input: graph $G = (V, E)$

Output: the maximum independent set of G

max-independent-set($G = (V, E)$)

```
R ← ∅
for every set S ⊆ V do
    b ← true
    for every u, v ∈ S do
        if (u, v) ∈ E then b ← false
    if b and |S| > |R| then R ← S
return R
```

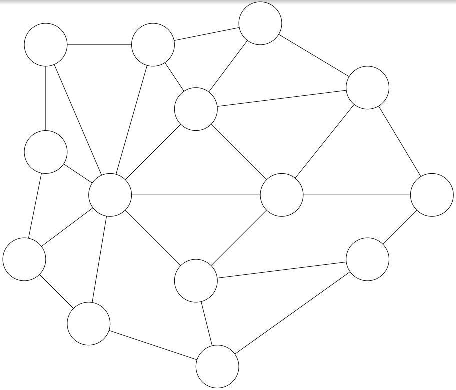
Running time $= O(2^n n^2)$.

Beyond Polynomial Time: $n!$

Hamiltonian Cycle Problem

Input: a graph with n vertices

Output: a cycle that visits each node exactly once, or say no such cycle exists

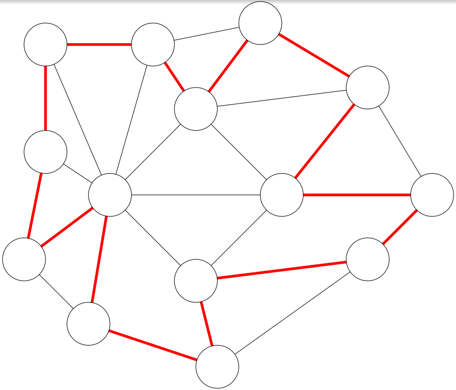


Beyond Polynomial Time: $n!$

Hamiltonian Cycle Problem

Input: a graph with n vertices

Output: a cycle that visits each node exactly once,
or say no such cycle exists



Beyond Polynomial Time: $n!$

Hamiltonian($G = (V, E)$)

```
for every permutation $(p_1, p_2, \cdots, p_n)$ of $V$ do
    $b \leftarrow$ true
    for $i \leftarrow 1$ to $n - 1$ do
        if $(p_i, p_{i+1}) \notin E$ then $b \leftarrow$ false
    if $(p_n, p_1) \notin E$ then $b \leftarrow$ false
    if $b$ then return $(p_1, p_2, \cdots, p_n)$
return "No Hamiltonian Cycle"
```

Running time $= O(n! \times n)$

$O(\log n)$ (Logarithmic) Running Time

$O(\log n)$ (Logarithmic) Running Time

- Binary search
    - Input: sorted array A of size n, an integer t;
    - Output: whether t appears in A.

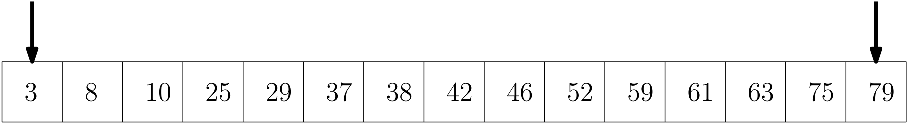
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

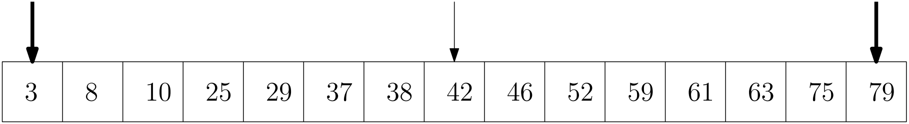
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

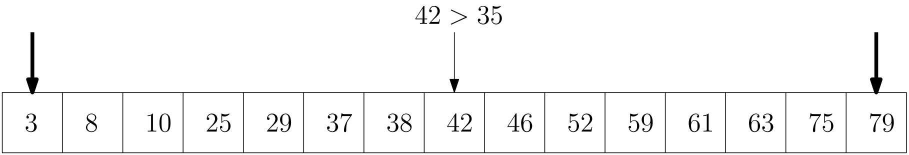
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

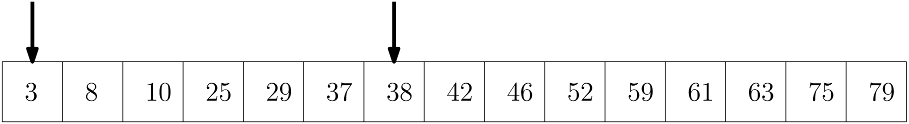
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

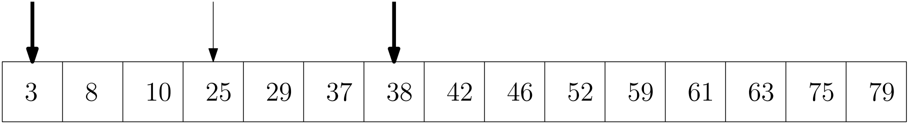
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

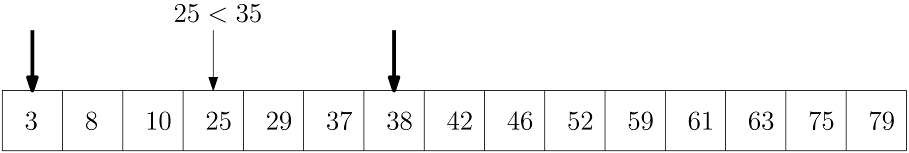
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

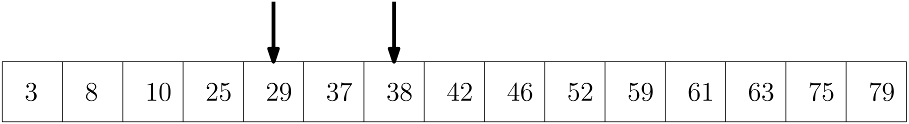
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

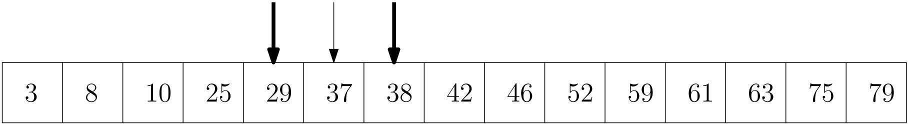
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

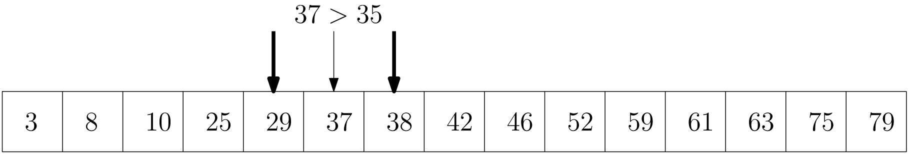
- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

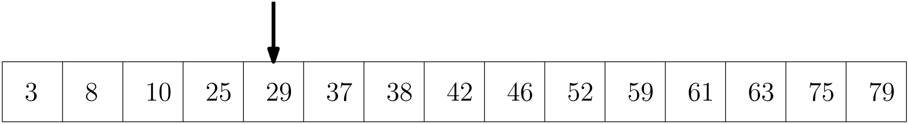
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:



3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array A of size n, an integer t;
- Output: whether t appears in A.

binary-search(A, n, t)

```
i ← 1, j ← n
while $i \le j$ do
    $k \leftarrow \lfloor (i+j)/2 \rfloor$
    if $A[k] = t$ return true
    if $t < A[k]$ then $j \leftarrow k-1$ else $i \leftarrow k+1$
return false
```

$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array A of size n, an integer t;
- Output: whether t appears in A.

binary-search(A, n, t)

```
i ← 1, j ← n
while i ≤ j do
    k ← ⌊(i + j)/2⌋
    if A[k] = t return true
    if t < A[k] then j ← k − 1 else i ← k + 1
return false
```

Running time $= O(\log n)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n \log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n,\ \ n\log n,\ \ n,\ \ n!,\ \ n^2,\ \ 2^n,\ \ e^n,\ \ n^n$
- $\log n = O(n)$
- $n = O(n^2)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \; n\log n, \; n, \; n!, \; n^2, \; 2^n, \; e^n, \; n^n$
- $\log n = O(n)$
- $n = O(n\log n)$
- $n\log n = O(n^2)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n \log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n\log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n\log n)$
- $n\log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n\log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n\log n)$
- $n\log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(e^n)$
- $e^n = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \quad n \log n, \quad n, \quad n!, \quad n^2, \quad 2^n, \quad e^n, \quad n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(e^n)$
- $e^n = O(n!)$
- $n! = O(n^n)$

Terminologies

When we talk about upper bound on running time:

- Logarithmic time: $O(\log n)$
- Linear time: $O(n)$
- Quadratic time $O(n^2)$
- Cubic time $O(n^3)$
- Polynomial time: $O(n^k)$ for some constant k
  - $O(n \log n) \subseteq O(n^{1.1})$. So, an $O(n \log n)$-time algorithm is also a polynomial time algorithm.
- Exponential time: $O(c^n)$ for some $c > 1$
- Sub-linear time: $o(n)$
- Sub-quadratic time: $o(n^2)$

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms
- Makes our life much easier! (E.g., the leading constant depends on the implementation, complier and computer architecture of computer.)

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$
- For "natural" algorithms, constants are not so big!

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$
- For "natural" algorithms, constants are not so big!
- So, for reasonably large n, algorithm with lower order running time beats algorithm with higher order running time.

CSE 431/531: Algorithm Analysis and Design (Spring 2024)

Graph Basics

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder

Examples of Graphs

Figure: Road Networks

Figure: Internet

Figure: Social Networks

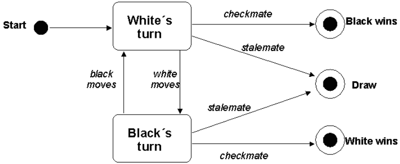
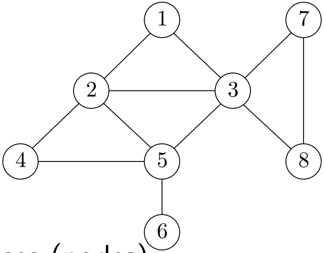


Figure: Transition Graphs

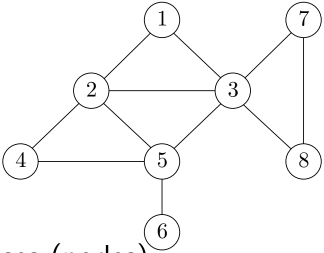
(Undirected) Graph $G = (V, E)$



- V: set of vertices (nodes);

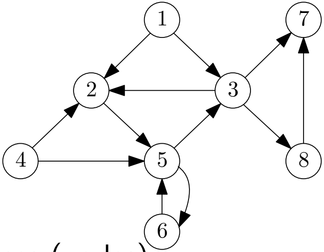
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2

(Undirected) Graph $G = (V, E)$



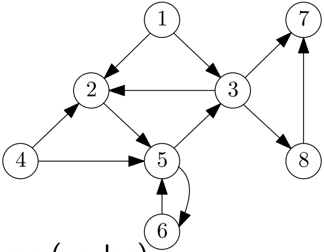
- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2
  - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$

Directed Graph $G = (V, E)$



- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - directed graphs: relationship is asymmetric, E contains ordered pairs

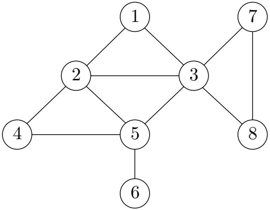
Directed Graph $G = (V, E)$



- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - directed graphs: relationship is asymmetric, E contains ordered pairs
  - $E = \{(1,2), (1,3), (3,2), (4,2), (2,5), (5,3), (3,7), (3,8), (4,5), (5,6), (6,5), (8,7)\}$

Abuse of Notations

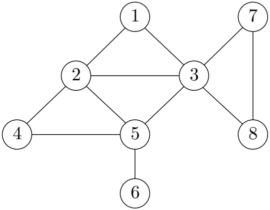
- For (undirected) graphs, we often use (i, j) to denote the set $\{i, j\}$.
- We call (i, j) an unordered pair; in this case $(i, j) = (j, i)$.



- $E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}$

- Social Network : Undirected
- Transition Graph : Directed
- Road Network : Directed or Undirected
- Internet : Directed or Undirected

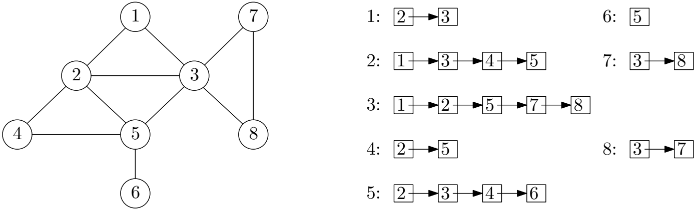
Representation of Graphs



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- Adjacency matrix
  - $n \times n$ matrix, $A[u,v] = 1$ if $(u,v) \in E$ and $A[u,v] = 0$ otherwise
  - A is symmetric if graph is undirected

Representation of Graphs



1: 2 → 3

2: 1 → 3 → 4 → 5

3: 1 → 2 → 5 → 7 → 8

4: 2 → 5

5: 2 → 3 → 4 → 6

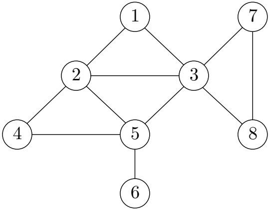
6: 5

7: 3 → 8

8: 3 → 7

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbors of v.

Representation of Graphs



1: 2 3

2: 1 3 4 5

3: 1 2 5 7 8

4: 2 5

5: 2 3 4 6

6: 5

7: 3 8

8: 3 7

$d : (2, 4, 5, 2, 4, 1, 2, 2)$

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbors of v.
  - When graph is static, can use array of variant-length arrays.

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage		
time to check $(u,v) \in E$		
time to list all neighbors of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n - 1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	
time to check $(u, v) \in E$		
time to list all neighbors of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \leq m \leq n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$		
time to list all neighbors of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	
time to list all neighbors of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v	$O(n)$	

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v	$O(n)$	$O(d_v)$

Outline

1. Graphs
2. Connectivity and Graph Traversal
    - Types of Graphs
3. Bipartite Graphs
    - Testing Bipartiteness
4. Topological Ordering
    - Applications: Word Ladder

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

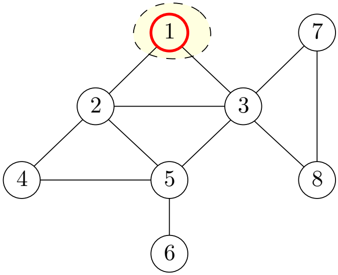
- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)

Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j

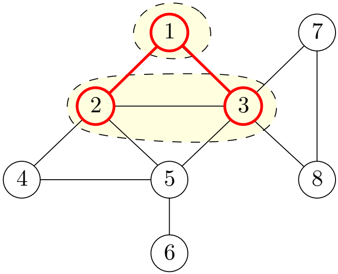
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



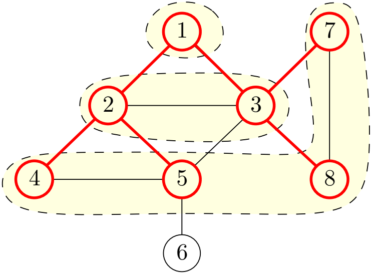
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



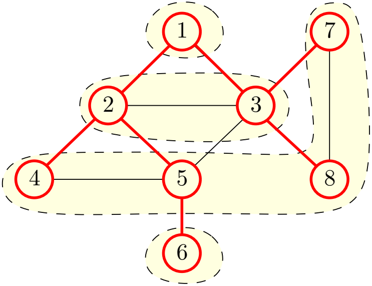
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j

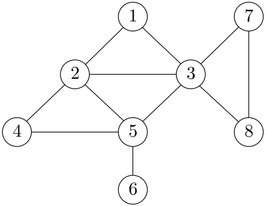


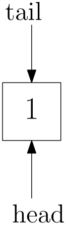
Implementing BFS using a Queue

```
BFS($s$)
$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as "visited" and all other vertices as "unvisited"
while $head \leq tail$ do
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors $u$ of $v$ do
        if $u$ is "unvisited" then
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark $u$ as "visited"
```

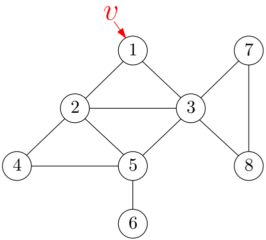
- Running time: $O(n + m)$.

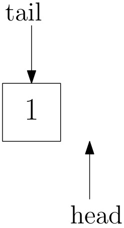
Example of BFS via Queue



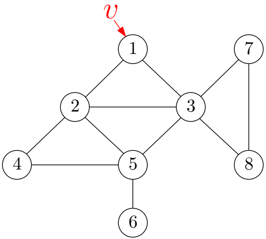


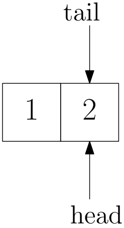
Example of BFS via Queue



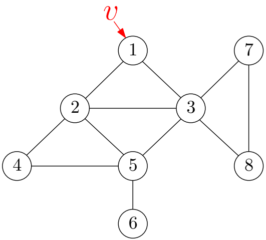


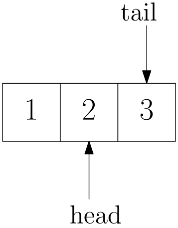
Example of BFS via Queue





Example of BFS via Queue



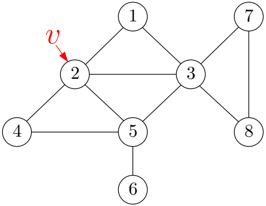


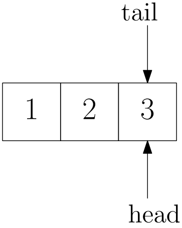
tail

1	2	3

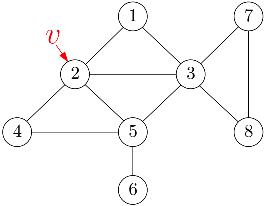
head

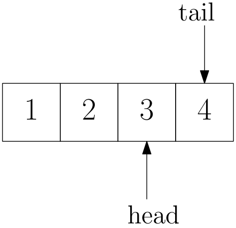
Example of BFS via Queue



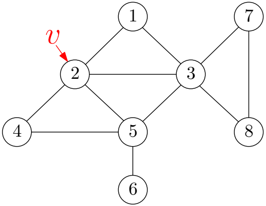


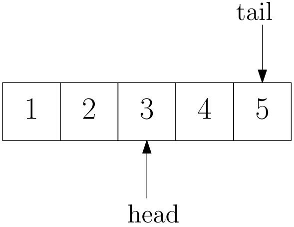
Example of BFS via Queue





Example of BFS via Queue

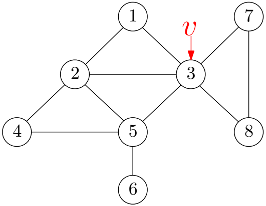


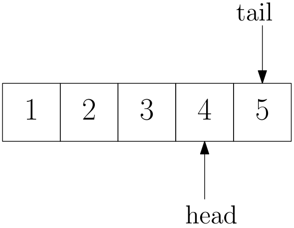

tail

1	2	3	4	5

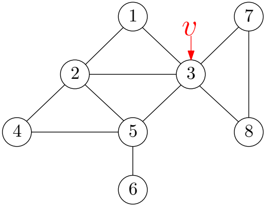
head

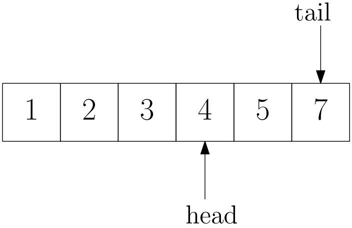
Example of BFS via Queue



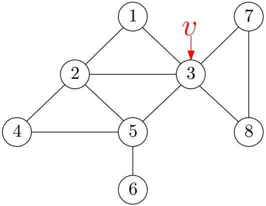


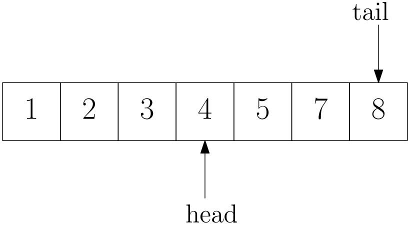
Example of BFS via Queue



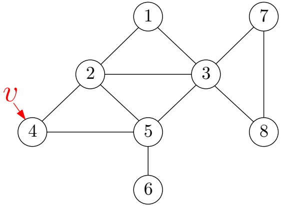


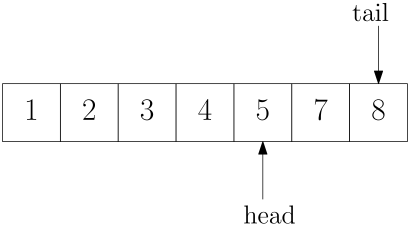
Example of BFS via Queue



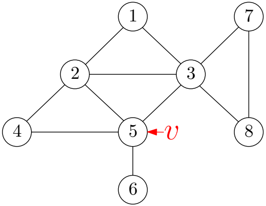


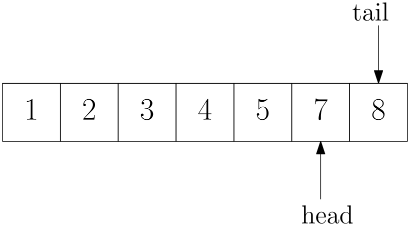
Example of BFS via Queue





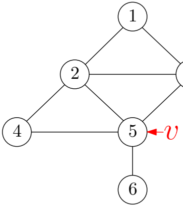
Example of BFS via Queue

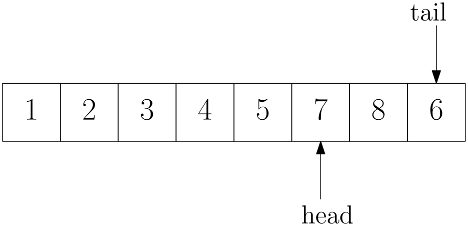




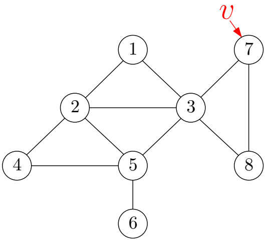
1	2	3	4	5	7	8

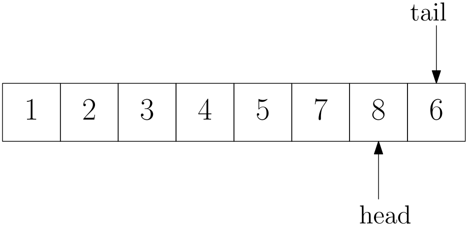
Example of BFS via Queue





Example of BFS via Queue

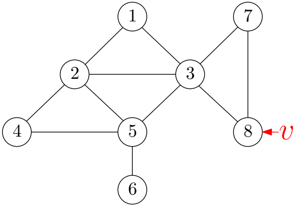


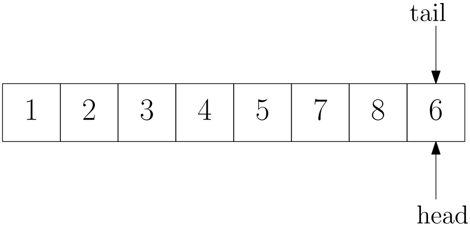

tail

1	2	3	4	5	7	8	6

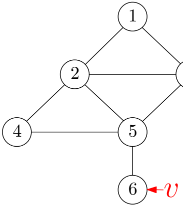
head

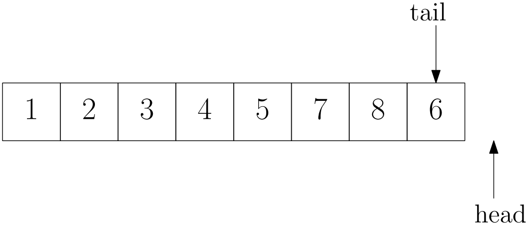
Example of BFS via Queue





Example of BFS via Queue





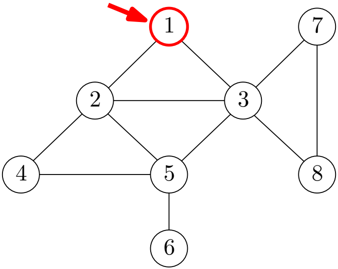
1	2	3	4	5	7	8	6

Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

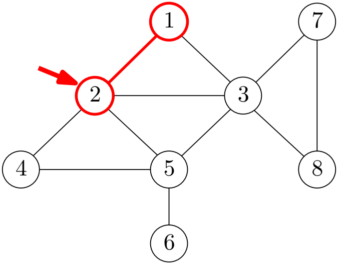
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



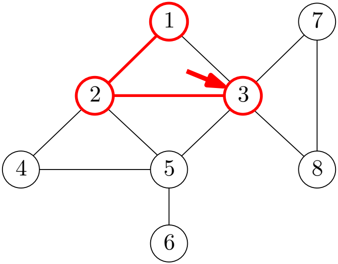
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



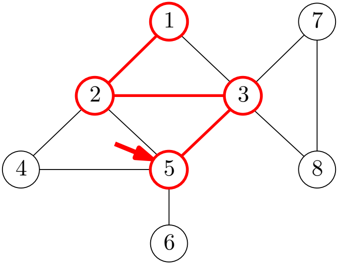
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



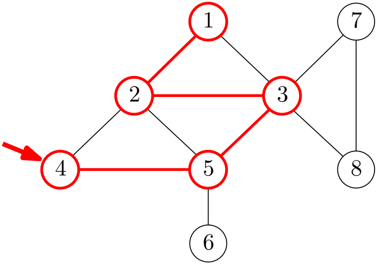
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



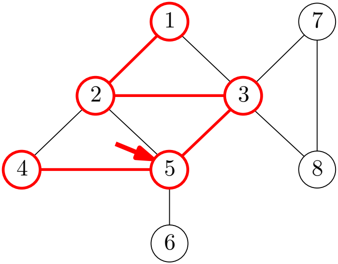
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



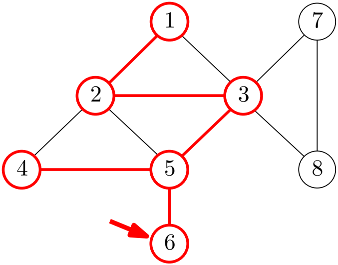
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



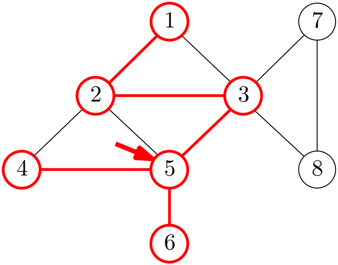
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



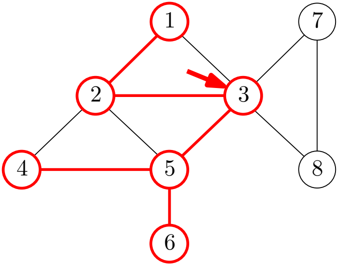
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



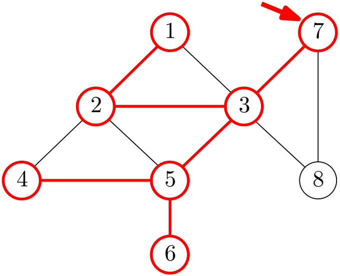
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



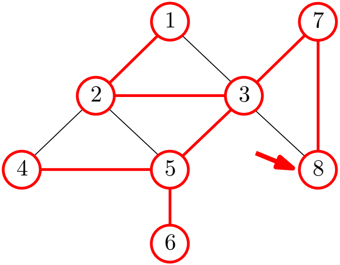
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Implementing DFS using Recurrsion

DFS(s)

```
mark all vertices as “unvisited”
recursive-DFS($s$)
```

recursive-DFS(v)

```
mark $v$ as “visited”
for all neighbors $u$ of $v$ do
    if $u$ is unvisited then recursive-DFS($u$)
```

Outline

1. Graphs

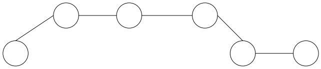
2. Connectivity and Graph Traversal
    - Types of Graphs

3. Bipartite Graphs
    - Testing Bipartiteness

4. Topological Ordering
    - Applications: Word Ladder

Path Graph (or Linear Graph)

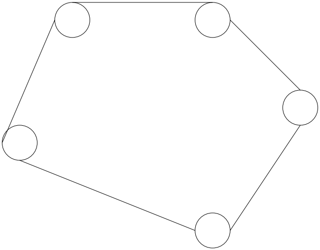
Def. An undirected graph $G = (V, E)$ is a path if the vertices can be listed in an order $\{v_1, v_2, ..., v_n\}$ such that the edges are the $\{v_i, v_{i+1}\}$ where $i = 1, 2, ..., n-1$.



- Path graphs are connected graphs.

Cycle Graph (or Circular Graph)

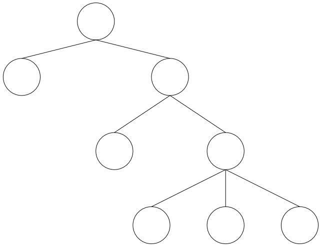
Def. An undirected graph $G = (V, E)$ is a cycle if its vertices can be listed in an order $v_1, v_2, ..., v_n$ such that the edges are the $\{v_i, v_{i+1}\}$ where $i = 1, 2, ..., n-1$, plus the edge $\{v_n, v_1\}$.



- The degree of all vertices is 2.

Tree

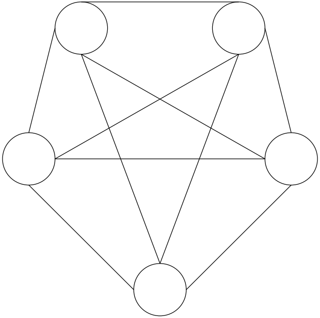
Def. An undirected graph $G = (V, E)$ is a tree if any two vertices are connected by exactly one path. Or the graph is a connected acyclic graph.



- Most important type of special graphs: most computational problems are easier to solve on trees or lines.

Complete Graph

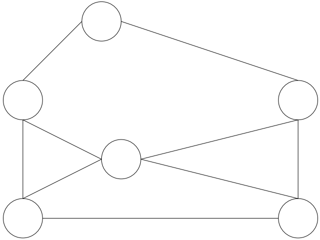
Def. An undirected graph $G = (V, E)$ is a complete graph if each pair of vertices is joined by an edge.



- A complete graph contains all possible edges.

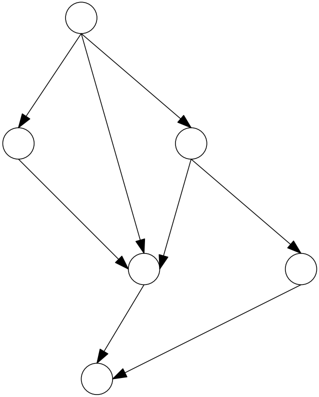
Planar Graph

Def. An undirected graph $G = (V, E)$ is a planar graph if its vertices and edges can be drawn in a plane such that no two of the edges intersect.



- Most computational problems have good solutions in a planar graph.

Directed Acyclic Graph (DAG)



Def. A directed graph $G = (V, E)$ is a directed acyclic graph if it is a directed graph with no directed cycles

- DAG is equivalent to a partial ordering of nodes.

Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.