Programming Design

Complexity and Graphs

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Complexity**
- The “big O” notation
- Terminology of graphs
- Graph algorithms

Complexity

- Given a task, we design algorithms.
  - These algorithms may all be correct.
  - One algorithm may be **better** than another one.
  - To compare algorithms, we compare their **complexity**.
- Time complexity and space complexity:
  - Time: We hope an algorithm takes a **short time** to complete the task.
  - Space: We hope an algorithm uses a **small space** to complete the task.
- Let's see some examples.

Space complexity

- Given a matrix A of $m \times n$ integers, find the row whose row sum is the largest.
- Two algorithms:
  - For each row, find the sum. Store the m row sums, scan through them, and find the target row.
  - For each row, find the sum and compare it with the currently largest row sum. Update the currently largest row sum if it is larger.

Space complexity: algorithm 1

- Let's implement algorithm 1:

```
const int MAX_COL_CNT = 3;
const int MAX_ROW_CNT = 4;

int maxRowSum(int A[][MAX_COL_CNT],
              int m, int n)
{
  // calculate row sums
  int rowSum[MAX_ROW_CNT] = {0};
  for(int i = 0; i < m; i++)
  {
    int aRowSum = 0;
    for(int j = 0; j < n; j++)
      aRowSum += A[i][j];
    rowSum[i] = aRowSum;
  }
```

```
  // find the row with the max row sum
  int maxRowSumValue = rowSum[0];
  int maxRowNumber = 1;
  for(int i = 0; i < m; i++)
  {
    if(rowSum[i] > maxRowSumValue)
    {
      maxRowSumValue = rowSum[i];
      maxRowNumber = i + 1;
    }
  }
  return maxRowNumber;
}
```

Space complexity: algorithm 2

- Let’s implement algorithm 2:

```
int maxRowSum(int A[][MAX_COL_CNT],
              int m, int n)
{
  int maxRowSumValue = 0;
  int maxRowNumber = 0;
  for(int i = 0; i < m; i++)
  {
    int aRowSum = 0;
    for(int j = 0; j < n; j++)
      aRowSum += A[i][j];

    if(aRowSum > maxRowSumValue)
    {
      maxRowSumValue = aRowSum;
      maxRowNumber = i + 1;
    }
  }
  return maxRowNumber;
}
```

Space complexity: comparison

- The two algorithms use different amounts of space:
  - Algorithm 1: Declaring an array and three integers.
  - Algorithm 2: Declaring three integers.
- Algorithm 2 has **the lower space complexity**.

Time complexity

- In general, people care more about time complexity.
  - When we say "complexity," we mean time complexity.
- Intuitively, the complexity of an algorithm can be measured by executing the algorithm and **counting the running time**.
  - Maybe you want to do this several times and calculate the average.
- However, we need to remove the impact of machine capability.
- We may count the **number of basic operations** instead.
  - Basic operations: declaration, assignment, arithmetic, comparisons, etc.

Time complexity: example

- Consider the previous example.
- Let’s count the number of basic operations algorithm 1.
- For the first part of algorithm 1, we have $5mn + 10m + 2$ basic operations.

	Decl.	Assi.	Arith.	Comp.
(1)	m	m	0	0
(2)	1	$m + 1$	m	m
(3)	m	m	0	0
(4)	m	$m(n + 1)$	mn	mn
(5)	0	mn	mn	0
(6)	0	m	0	0

```
int rowSum[MAX_ROW_CNT] = {0}; // (1)
for(int i = 0; i < m; i++) // (2)
{
  int aRowSum = 0; // (3)
  for(int j = 0; j < n; j++) // (4)
    aRowSum += A[i][j]; // (5)
  rowSum[i] = aRowSum; // (6)
}

// the remaining are skipped
```

Time complexity: principle

- Wait… this is so tedious! And there is **no need to** be that precise.
- Consider algorithm 1:
  - $5mn + 10m + 2$ is roughly $5mn$ if n is large enough.
  - The bottleneck is the first part (the second part has only one level of loop).
  - The total number of operations is roughly $5mn$.
- Moreover, that constant 5 does not mean a lot:
  - It does not change when we get more integers (m or n increases).
- As we care the complexity of an algorithm the most when **the instance size is large**, we will ignore those constants and minor (non-bottleneck) parts.
  - We only focus on how the number of operations **grow** at the **bottleneck**.

Time complexity: example

- Let's analyze algorithm 2.
- The bottleneck is the two **nested loops**.
- The complexity is roughly mn:
  - This is how the execution time would grow as the input size increases.
- To formalize the above idea, let's introduce the "big O" notation.

```
int maxRowSum(int A[][MAX_COL_CNT],
              int m, int n)
{
  int maxRowSumValue = 0;
  int maxRowNumber = 0;
  for(int i = 0; i < m; i++)
  {
    int aRowSum = 0;
    for(int j = 0; j < n; j++)
      aRowSum += A[i][j];

    if(aRowSum > maxRowSumValue)
    {
      maxRowSumValue = aRowSum;
      maxRowNumber = i + 1;
    }
  }
  return maxRowNumber;
}
```

Outline

- Complexity
- **The “big O” notation**
- Terminology of graphs
- Graph algorithms

The “big O” notation

- Mathematically, let $f(n) \geq 0$ and $g(n) \geq 0$ be two functions defined for $n \in \mathbb{N}$. We say
$$\boldsymbol{f(n) \in O(g(n))}$$
if and only if there exists a positive number c and a number N such that
$$\boldsymbol{f(n) \leq cg(n)}$$
for all $n \geq N$.
- Intuitively, that means **when n is large enough, $g(n)$ will dominate $f(n)$**.
- If $f(n)$ is the number of operations that an algorithms takes to complete a task, we say **the algorithm’s time complexity** is $g(n)$.
  - We write $f(n) \in O(g(n))$, but some people write $f(n) = O(g(n))$.

Examples

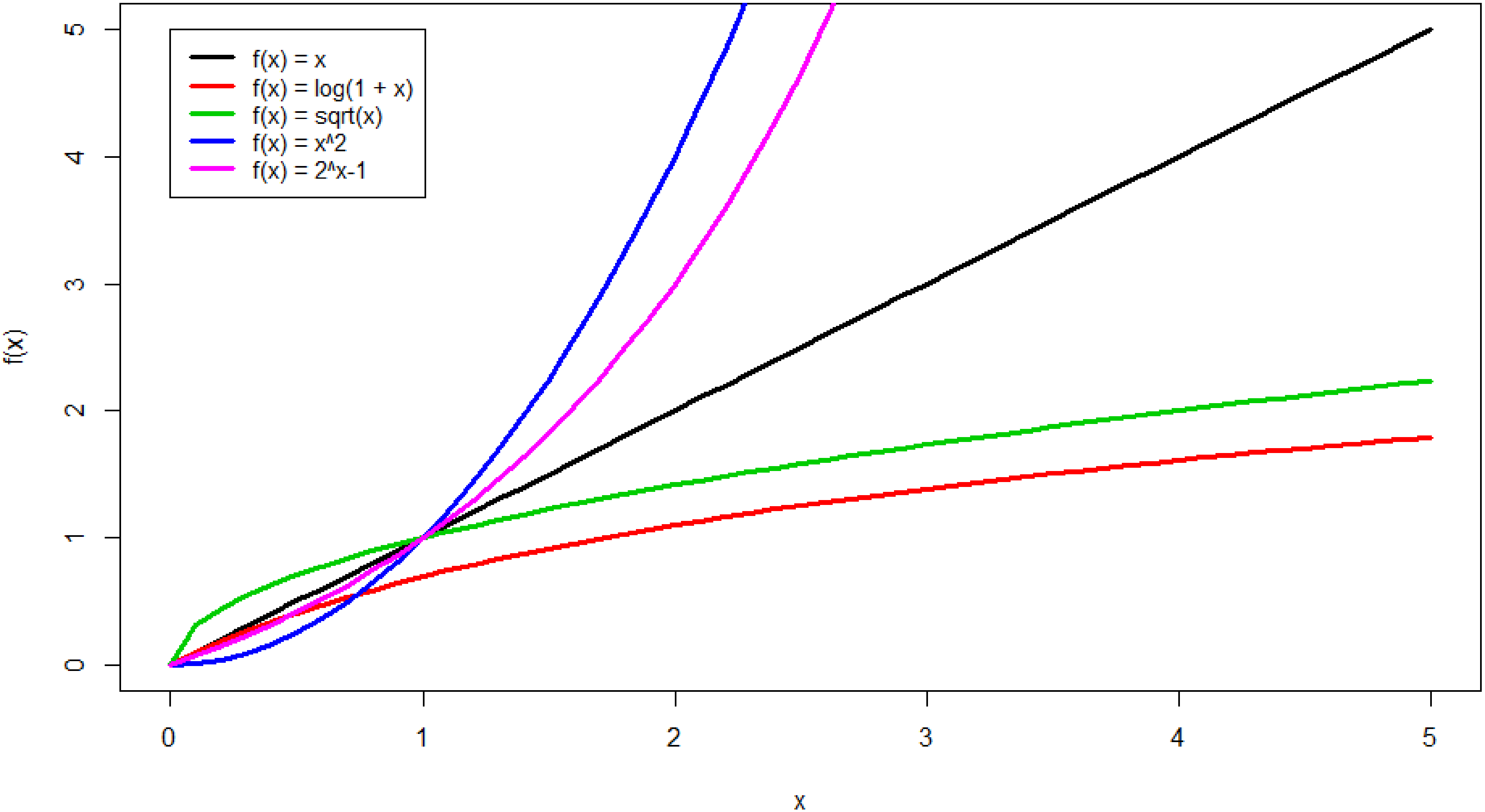
- Let $f(n) = 100n^2$, we have $g(n) = n^3$, i.e., $f(n) \in O(n^3)$.
  - We may choose $c = 100$ and $N = 1$: $100n^2 \leq \mathbf{100}n^3$ for all $n \geq \mathbf{1}$.
  - We may choose $c = 1$ and $N = 100$: $100n^2 \leq \mathbf{1}n^3$ for all $n \geq \mathbf{100}$.
- Let $f(n) = 100\sqrt{n} + 5n$, we have $g(n) = n$:
  - We may choose $c = 6$ and $N = 10$: $100\sqrt{n} + 5n \leq \mathbf{6}n$ for all $n \geq \mathbf{10000}$.
- Let $f(n) = n \log n + n^2$, we have $g(n) = n^2$.
- Let $f(n) = 10000$, we have $g(n) = 1$.
- Let $f(n) = 0.0001n^2$, we cannot have $g(n) = n$:
  - For any value of c, we have $0.0001n^2 > cn$ if $n > 10000c$.
- Let $f(n) = 2^n$, we cannot have $g(n) = n^{100}$.

Growth of functions

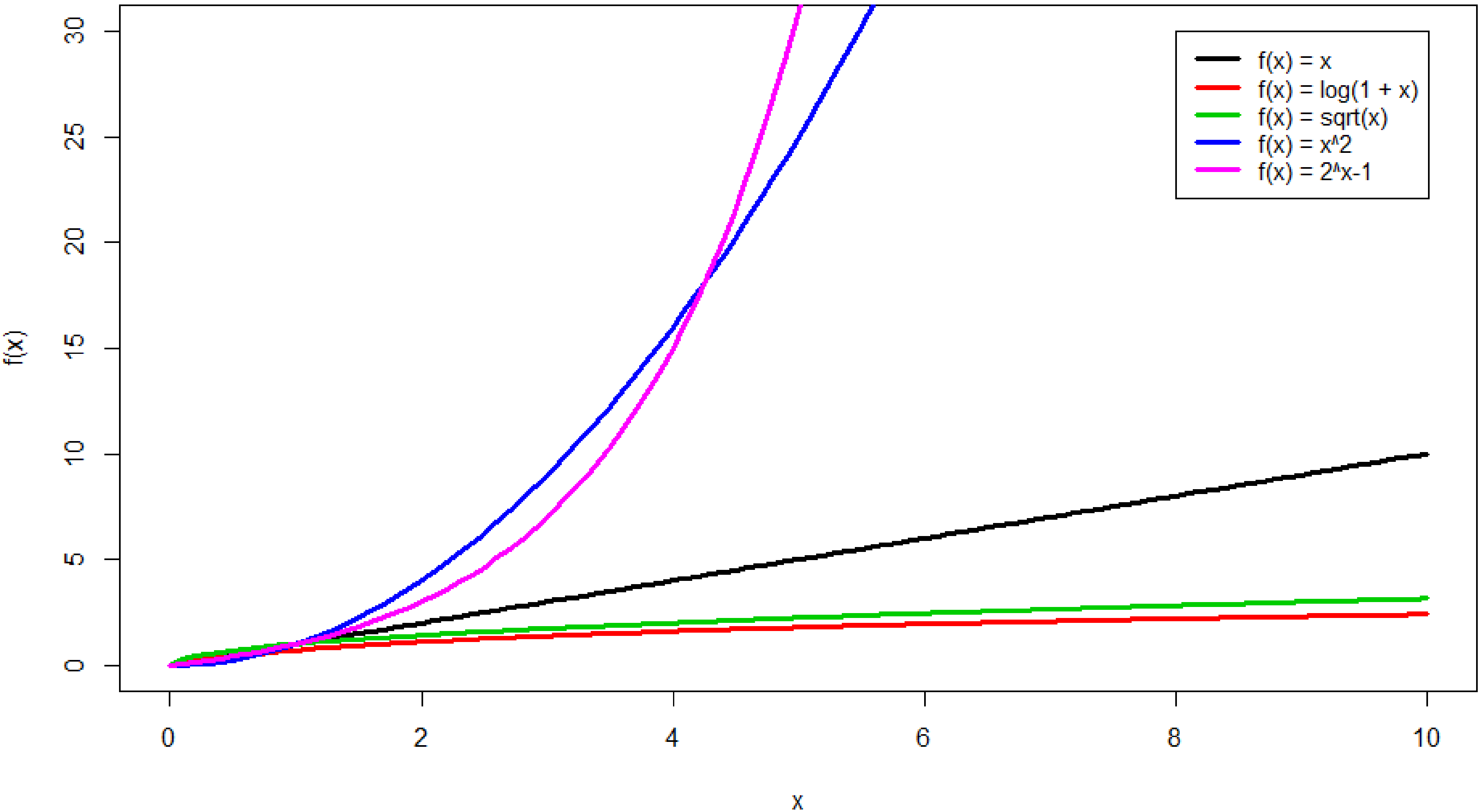
- In general, we may say that functions have **different growth speeds**.
- If a function grows faster than another one, we say the former "dominates" the latter or the former is "an upper bound" of the latter.

n	5	10	50	100	1000
$\log n$	2.32	3.32	5.64	6.64	9.97
$\sqrt{n}$	2.24	3.16	7.07	10.00	31.62
n	5	10	50	100	1000
$n \log n$	11.61	33.22	282.19	664.39	9965.78
n^2	25	100	2500	10000	1000000
2^n	32	1024	1.13×10^{15}	1.27×10^{30}	1.07×10^{301}
$n!$	120	3628800	3.04×10^{64}	9.33×10^{157}	Too big!!

Growth of functions



Growth of functions



The "big O" notation for algorithms

- For an algorithm, we use the "big O" notation to denote its complexity.
  - If the number of basic operations is $f(n)$, we first find a valid $g(n)$ such that $f(n) \in O(g(n))$.
  - We then say that **the algorithm's complexity is $O(g(n))$**, or **just $g(n)$**.
- Note that for each $f(n)$, we have many valid $g(n)$. As these $g(n)$ are all upper bounds of $f(n)$, we typically use **the smallest one** that we may find.

Example 1

- Going back to the previous example, algorithm 2's complexity is $O(mn)$.
  - The execution time is proportional to the matrix size.
  - It should be fine for the matrix to have millions of elements.

```
int maxRowSum(int A[][MAX_COL_CNT],
              int m, int n)
{
  int maxRowSumValue = 0;
  int maxRowNumber = 0;
  for(int i = 0; i < m; i++)
  {
    int aRowSum = 0;
    for(int j = 0; j < n; j++)
      aRowSum += A[i][j];

    if(aRowSum > maxRowSumValue)
    {
      maxRowSumValue = aRowSum;
      maxRowNumber = i + 1;
    }
  }
  return maxRowNumber;
}
```

Example 2

- Recall our examples for listing all prime numbers that are below n.
- What is the most naïve algorithm's complexity?
  - Consider `isPrime()` first.

```
bool isPrime(int x)
{
  for(int i = 2; i < x; i++)
    if(x % i == 0)
      return false;
  return true;
}
```

  - The number of operations **depends on the value of x**! 18 is easy but 17 is hard.

```
#include <iostream>
using namespace std;

bool isPrime(int x);
int main()
{
  int n = 0;
  cin >> n;

  for(int i = 2; i <= n; i++)
  {
    if(isPrime(i) == true)
      cout << i << " ";
  }
  return 0;
}
```

Worst-case time complexity

- In many cases, the number of operations of running an algorithm depends on not only the **number of input values** but also **contents of input values**.
- People talk about two kinds of time complexity:
  - **Average-case time complexity**: the **expected** number of operations required for a randomly drawn input. The probability distribution matters.
  - **Worst-case time complexity**: the **maximum possible** number of operations required for a randomly drawn input.
- The "big O" notation typically deals with worst-case complexity.

Example 2

- The most naïve algorithm's complexity:
  - Checking whether x is prime is $O(x)$.

```
bool isPrime(int x)
{
  for(int i = 2; i < x; i++)
    if(x % i == 0)
      return false;
  return true;
}
```

  - Checking all values below n is

$$O(1 + 2 + \cdots + n) = O(n^2).$$

- The most naïve algorithm's complexity is $O(n^2)$.

```
#include <iostream>
using namespace std;

bool isPrime(int x);
int main()
{
  int n = 0;
  cin >> n;

  for(int i = 2; i <= n; i++)
  {
    if(isPrime(i) == true)
      cout << i << " ";
  }
  return 0;
}
```

Example 3

- We have a better algorithm:

```
bool isPrime(int x)
{
  for(int i = 2; i * i <= x; i++)
    if(x % i == 0)
      return false;
  return true;
}
```

- For **isPrime()**, the complexity is $O(\sqrt{x})$.
- For the whole algorithm, the complexity is $O\left(\sum_{k=1}^{n} \sqrt{k}\right)$. How large is this?

Example 3: analysis

- Obviously, we have

$$\sum_{k=1}^{n} \sqrt{k} = \sqrt{1} + \cdots \sqrt{n} \leq \sqrt{n} + \cdots + \sqrt{n} = n\sqrt{n} = n^{3/2}.$$

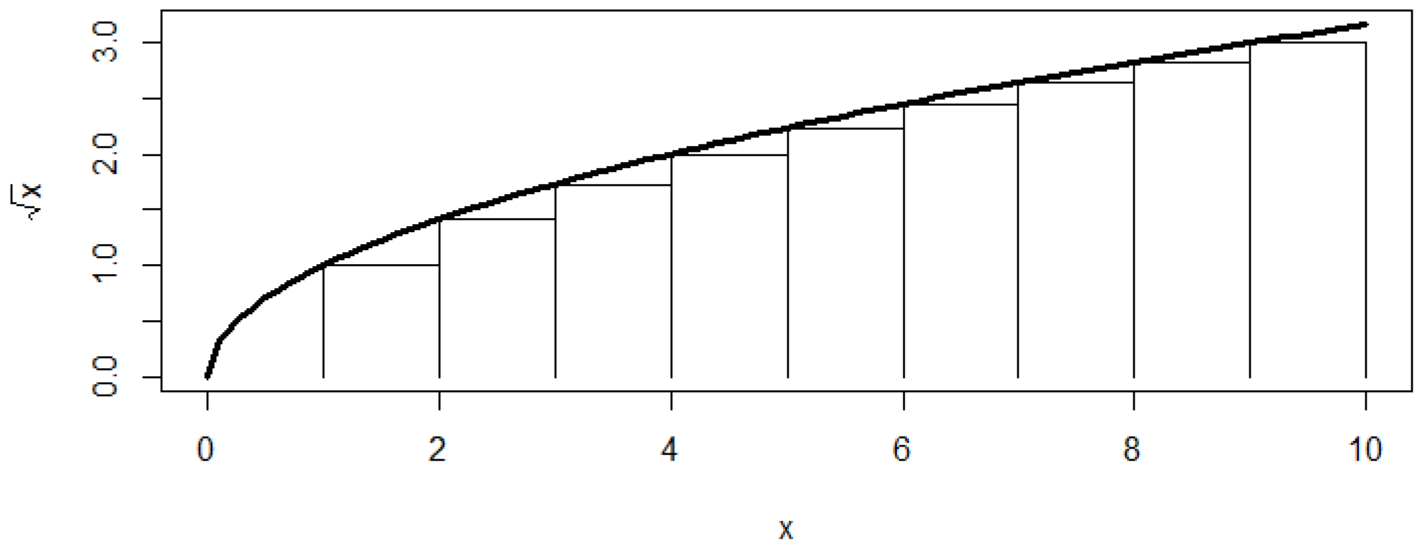
- Therefore, we have $\boldsymbol{O(n^{3/2})}$ for the better algorithm.
  - This is better than $O(n^2)$. This algorithm is indeed **theoretically** better.
  - Is it the **smallest** upper bound?

Example 3: analysis

- Thanks to calculus, we have

$$\sum_{k=1}^{n} \sqrt{k} \leq \int_{1}^{n+1} x^{1/2} dx = \frac{2}{3} x^{3/2} \Big|_{1}^{n+1} = \frac{2}{3}\left[(n+1)^{3/2} - 1\right].$$

- If $n = 9$:

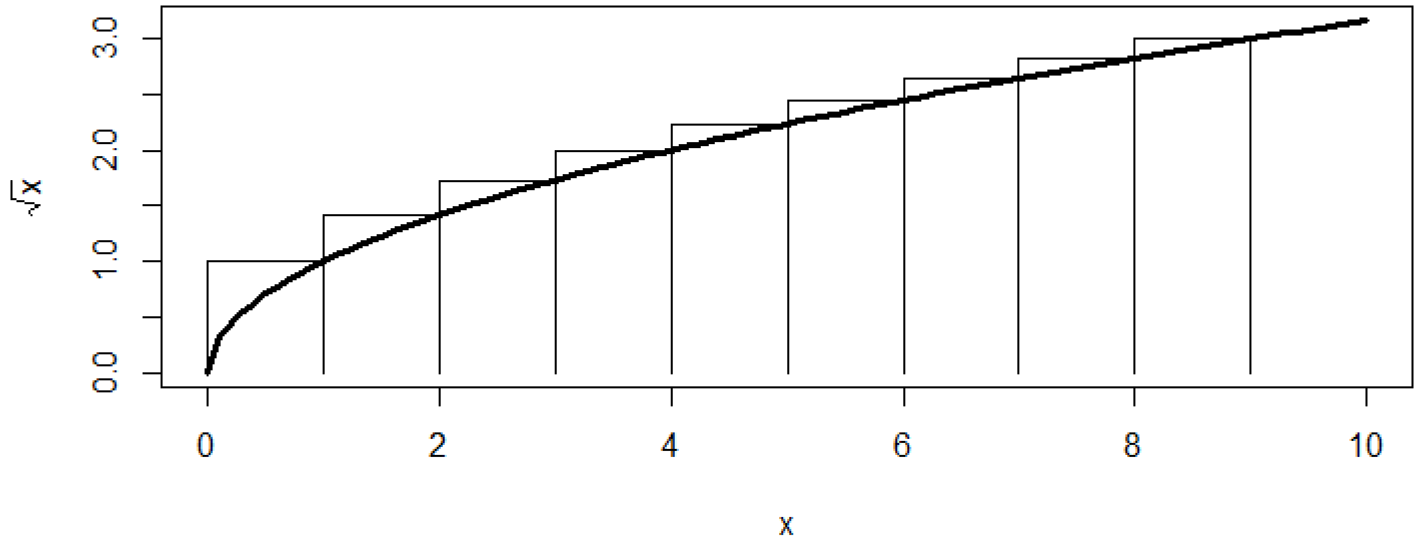


Example 3: analysis

- Thanks to calculus, we have

$$\sum_{k=1}^{n} \sqrt{k} \geq \int_{0}^{n} x^{1/2} dx = \frac{2}{3} x^{3/2} \Big|_{0}^{n} = \frac{2}{3} n^{3/2}.$$

- If $n = 9$:



Example 3: analysis

- Now we have
$$\frac{2}{3}n^{3/2} \leq \sum_{k=1}^{n} \sqrt{k} \leq \frac{2}{3}\left[(n+1)^{3/2} - 1\right],$$
- Therefore, $O\left(\sum_{k=1}^{n} \sqrt{k}\right) = O(n^{3/2})$ should be a good estimate.
- Now we know why studying calculus! XD

Example 4

- For listing all prime numbers below n, our best algorithm is:

> Given a Boolean array *A* of length *n*
> Initialize all elements in *A* to be ***true*** // assuming prime
> ***for*** *i* from 2 to *n*
>   ***if*** A_i is ***true***
>   print *i*
>   ***for*** *j* from 1 to $\lfloor n/i \rfloor$ // eliminating composite numbers
>   Set $A[i \times j]$ to ***false***

  - The outer loop has $O(n)$ iterations.
  - For the ith iteration of the outer loop, the inner loop has $O(n/i)$ iterations.
  - Let's ignore the selection statement for simplicity ("in the worst case").
- The overall complexity is $O(n/2 + n/3 + \cdots + n/n)$. How large is it?

Example 4: analysis

- We have

$$n\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right)\leq n\int_{1}^{n}\frac{1}{x}dx=n\ln n\,.$$

- Therefore, $O\left({}^{n}/_{2}+{}^{n}/_{3}+\cdots+{}^{n}/_{n}\right)=O(n\ln n)$.
  - $n\ln n<n\sqrt{n}$, good!
- In fact, the inner loop will be initiated only if we encounter a prime number.
- The true complexity is

$$O\left(\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\frac{n}{7}+\frac{n}{11}+\cdots\right).$$

  - Even smaller than $O(n\ln n)$.

Remarks

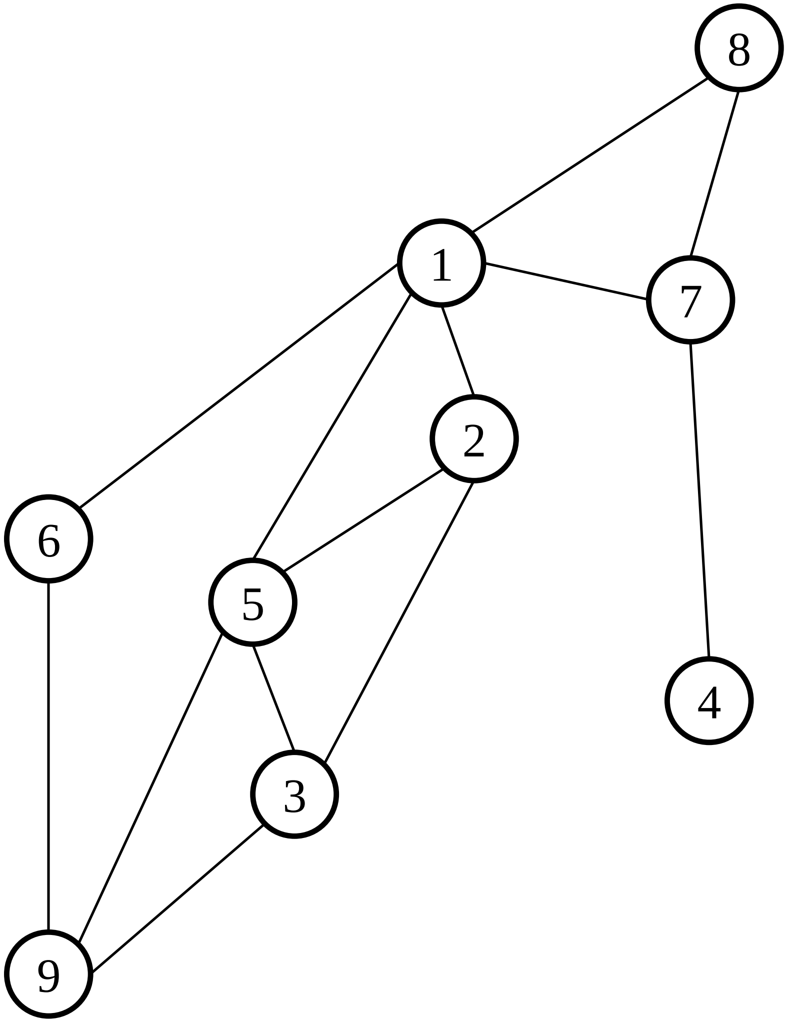
- Analyzing an algorithm’s complexity is critical in algorithm design.
  - We focus on how the number of operations grow as the input size increases.
- We use the “big O” notation:
  - We ignore tedious details, non-bottlenecks, and constants.
  - We focus on the worst case.
- There are some algorithms whose complexity cannot be easily analyzed.
  - E.g., those constructed by recursion.
- There are other measurements (small o, theta, big omega, small omega).
  - Expect them in your future courses!

Outline

- Complexity
- The “big O” notation
- **Terminology of graphs**
- Graph algorithms

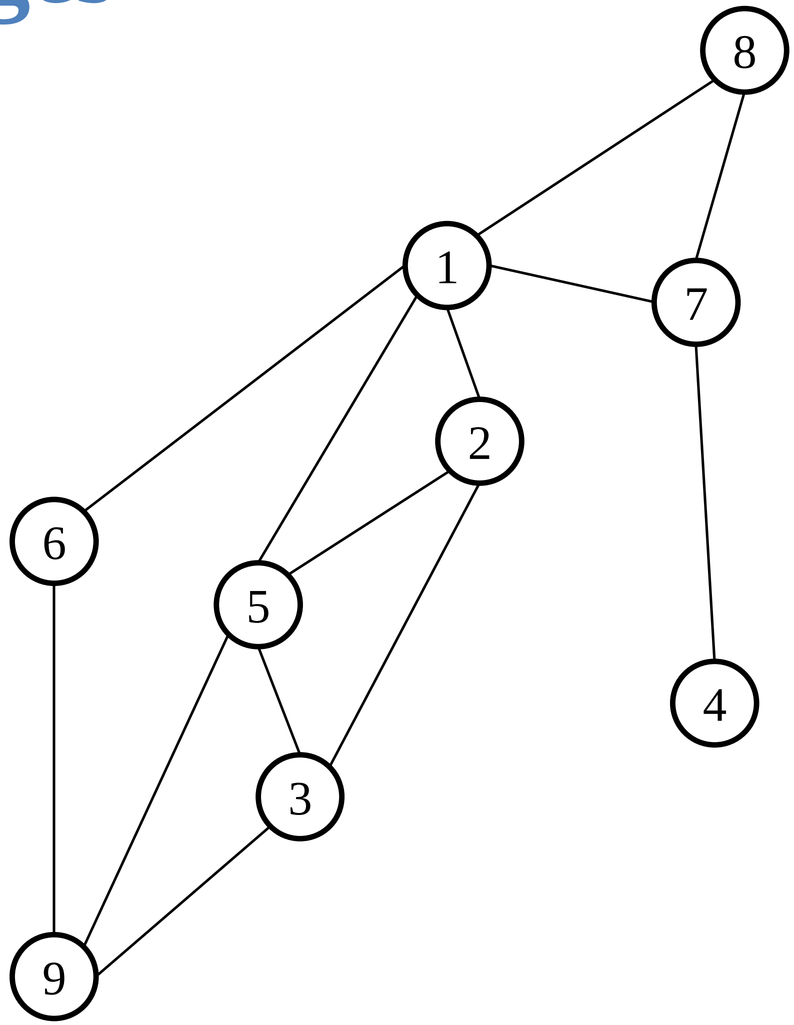
Graphs/networks

- In graph theory, we talk about **graphs/networks**.
- A graph has **nodes** (**vertices**) and **edges** (**arcs/links**).
  - A typical interpretation: Nodes are locations and arcs are roads.
- This graph has 9 nodes and 13 edges.
- Two nodes are **adjacent** if there is an edge between them.
  - We say they are **neighbors**.
  - A node's **degree** is its number of neighbors.



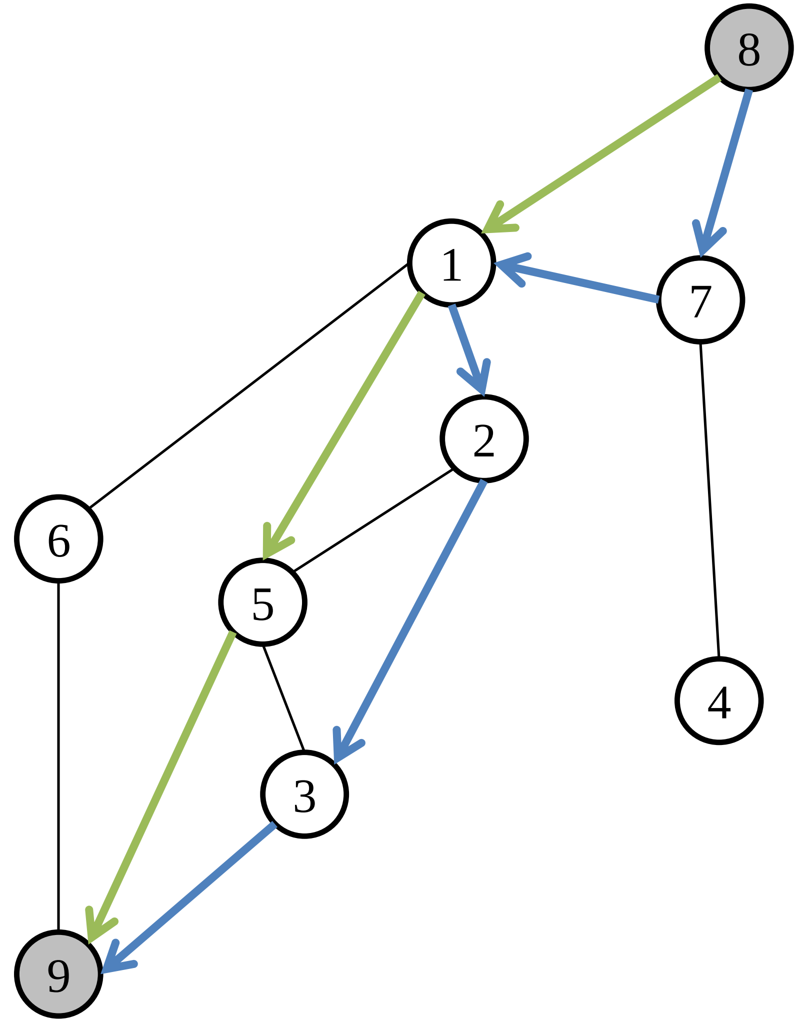
Directed/undirected edges

- Edges may be **directed** or **undirected**.
  - For an edges from u to v, we denote it as (u, v) if it is directed or $[u, v]$ if it is undirected.
  - A graph is a directed graph if its edges are directed.
- In this graph, we have edge $[1, 6]$ (or $[6, 1]$), but we do not have edge $[5, 6]$.
- This is an undirected graph.



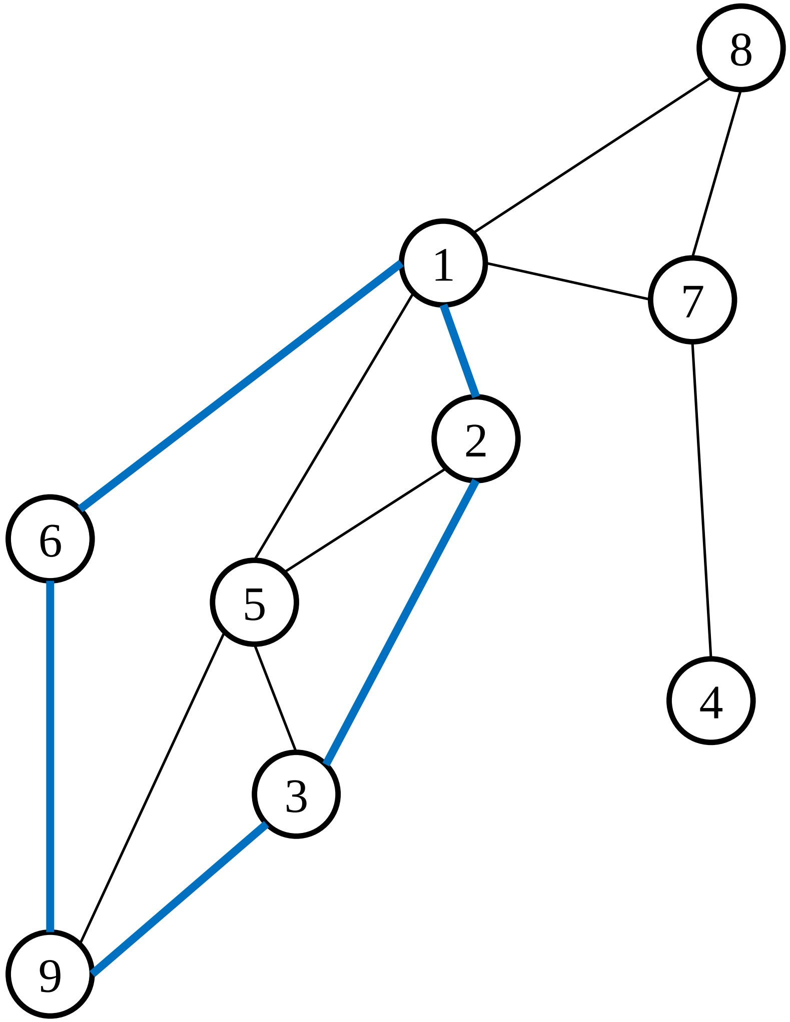
Paths

- A **path** (**route**) from node s to node t is a set of directed edges (s, v_1), (v_1, v_2), …, and (v_{k-1}, v_k), and (v_k, t) such that s and t are connected.
  - s is called the **source** and t is called the **destination** of the path.
  - Sometimes we write a path as $(s, v_1, v_2, \dots, v_k, t)$.
  - Direction matters!
- There are at least two paths from node 8 to node 9: $(8, 1, 5, 9)$ and $(8, 7, 1, 2, 3, 9)$.



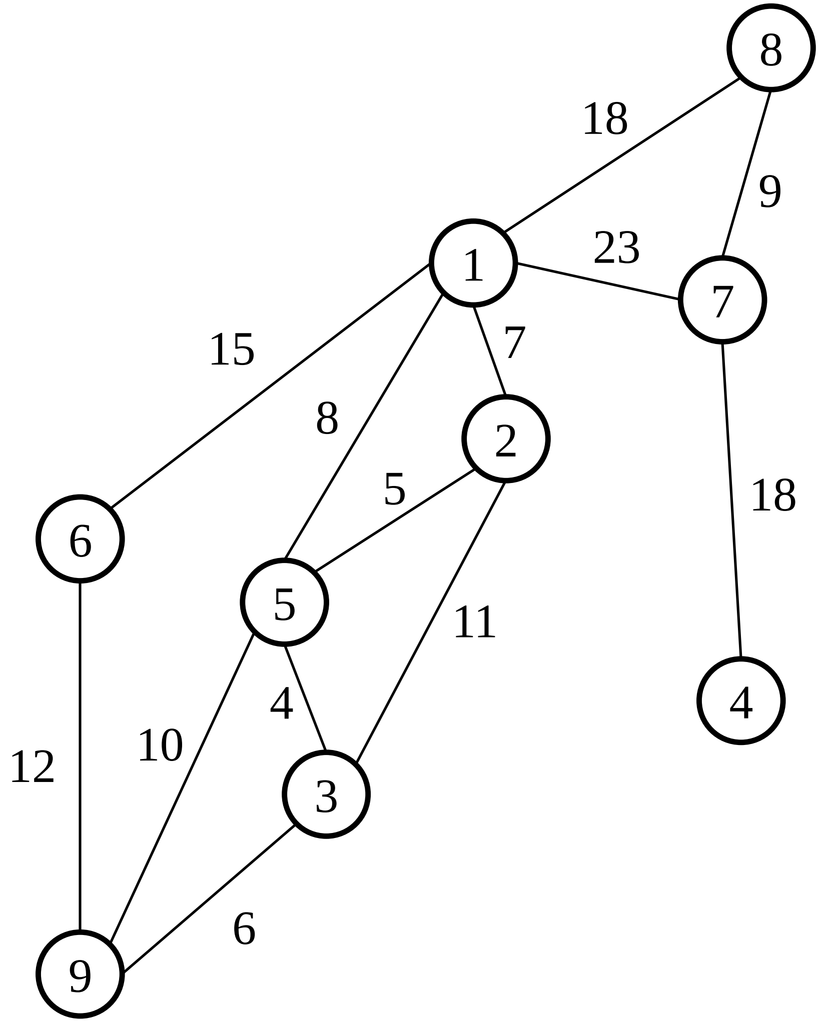
Cycles

- A **cycle** (equivalent to circuit in some textbooks) is a path whose destination node is the source node.
  - A path is a **simple path** if it is not a cycle.
  - A graph is an **acyclic graph** if it contains no cycle.
- There is a cycle $(1, 2, 3, 9, 6)$.



Weights

- An edge may have a **weight**.
  - A weight may be a distance, a cost per unit item shipped, etc.
  - A **weighted graph** is a graph whose edges are weighted.
- In this network, we may use edge weights to represent distances.
  - The distance of the path $(8, 1, 5, 9)$ is 36. That of $(8, 7, 1, 2, 3, 9)$ is 56.
- A node may also have a weight.



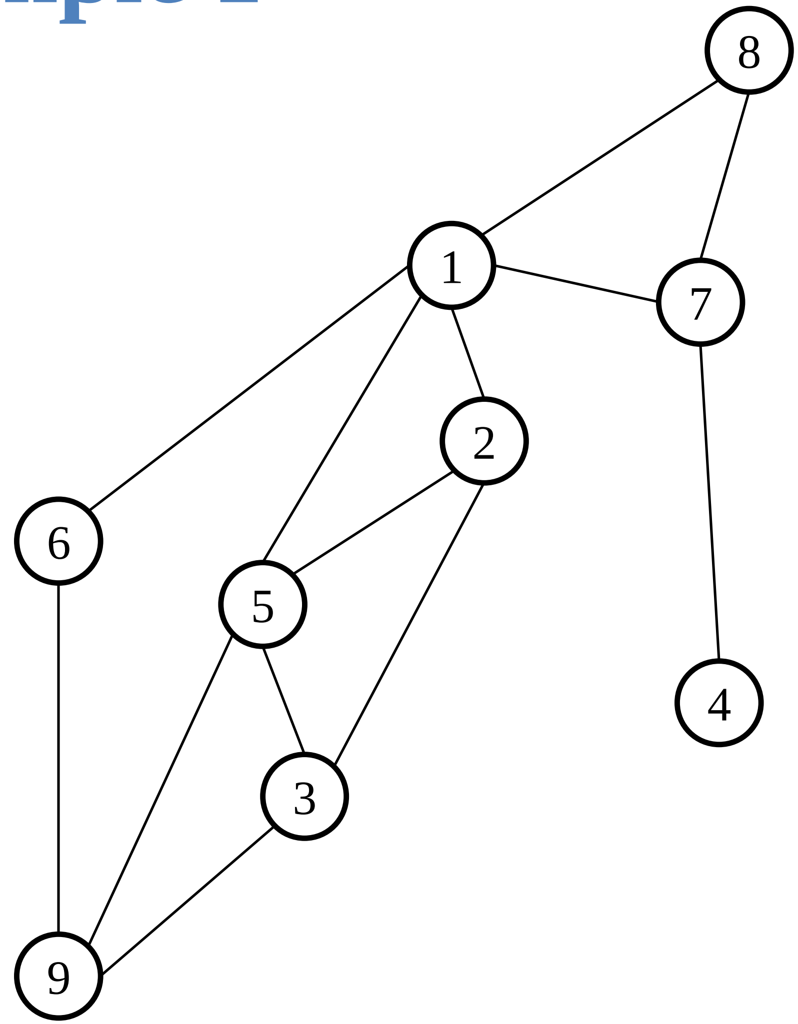
Storing a graph in an adjacency matrix

- To write a program that deals with a graph, we must have a way to store the graph in our program.
- Two typical data structures are **adjacency matrices** and **adjacency lists**.
- Adjacency matrix:
  - For a graph with n nodes, we construct an $n \times n$ array A.
  - If the graph is unweighted, make the array a Boolean array. Let $A_{ij} = 1$ if there is an edge (i, j) (or $[i, j]$ if undirected). Let $A_{ii} = 1$ for either case.
  - If the graph is weighted, make the array an integer/float/double array. Let A_{ij} be the weight of the edge (i, j) (or $[i, j]$ if undirected). Use a specially chosen value (-1, ∞, etc.) to indicate the nonexistence of edges.

Adjacency matrix: example 1

- For this unweighted graph, the adjacency matrix is

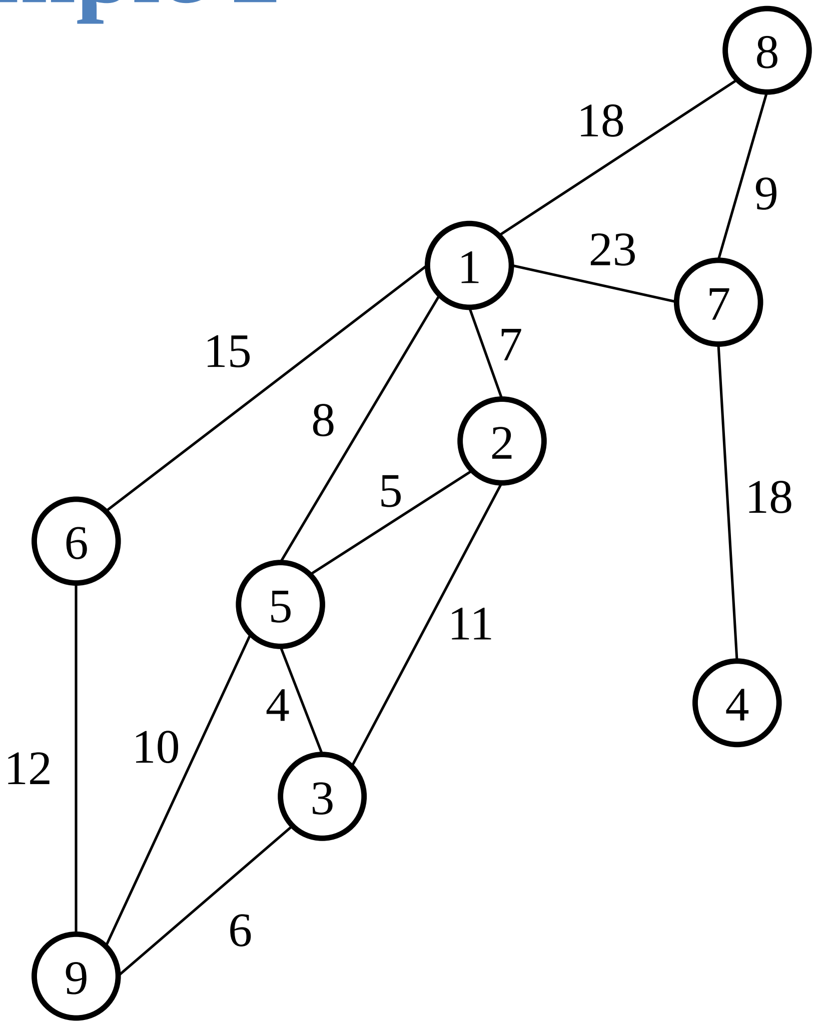
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$



Adjacency matrix: example 2

- For this weighted graph, the adjacency matrix is

$$
\begin{bmatrix}
-1 & 7 & -1 & -1 & 8 & 15 & 23 & 18 & -1 \\
7 & -1 & 11 & -1 & 5 & -1 & -1 & -1 & -1 \\
-1 & 11 & -1 & -1 & 4 & -1 & -1 & -1 & 6 \\
-1 & -1 & -1 & -1 & -1 & -1 & 18 & -1 & -1 \\
8 & 5 & 4 & -1 & -1 & -1 & -1 & -1 & 10 \\
15 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 12 \\
23 & -1 & -1 & 18 & -1 & -1 & -1 & 9 & -1 \\
18 & -1 & -1 & -1 & -1 & -1 & 9 & -1 & -1 \\
-1 & -1 & 6 & -1 & 10 & 12 & -1 & -1 & -1
\end{bmatrix}
$$



Adjacency matrix

- An adjacency matrix is simple and straightforward.
- However, it is **space inefficient** if the graph has only **few edges**.
- To remedy this, we may use an adjacency list.
  - For each node, we record its neighbors and (if weighted) distances to it neighbors.
  - We will introduce this until we introduce pointers.

Outline

- Complexity
- The "big O" notation
- Terminology of graphs
- **Graph algorithms**

Graph algorithms

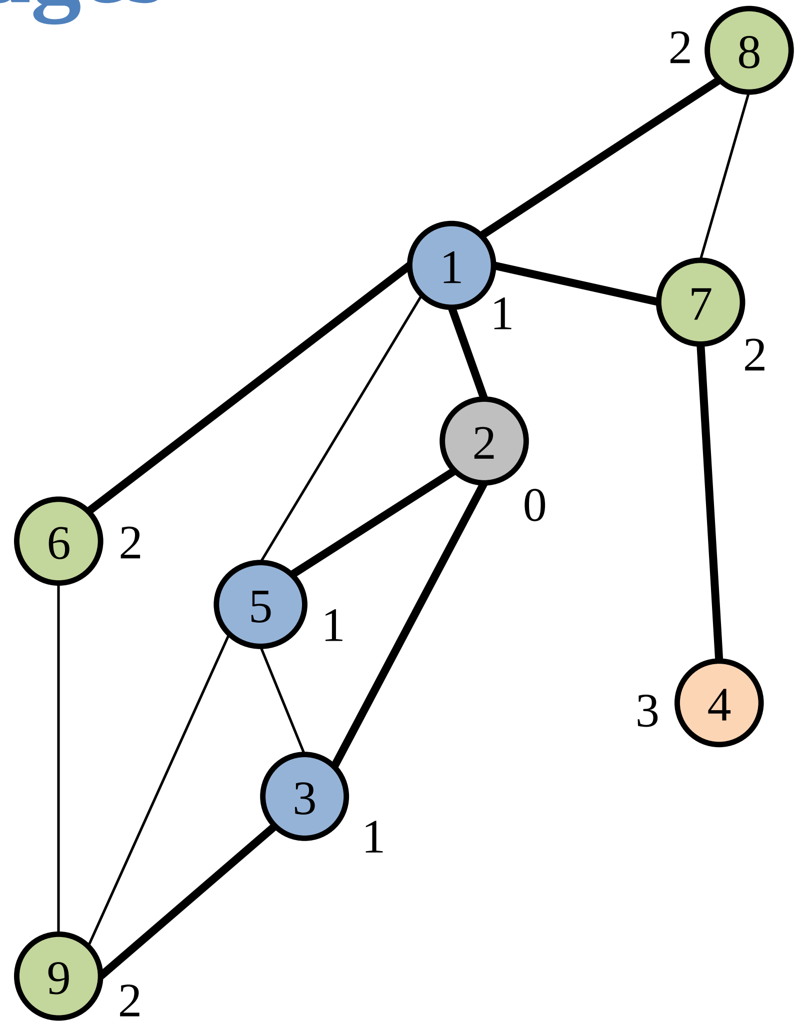
- As graphs can represent many things (logistic networks, power networks, social networks, etc.), there are many interesting issues.
  - How to find a shortest path from a node to another node?
  - How to link all nodes while minimizing the weights of selected edges?
  - How to check whether there is a cycle?
  - How to find the node with the maximum degree (number of neighbors)?
  - How to select the minimum number of nodes such that all nodes are either selected or adjacent to a selected node?
- Algorithms that solve these issues on graphs are **graph algorithms**.
- Below we give some examples demonstrating how to use an adjacency matrix.

Maximum degree

- How to find the **node** with the **maximum degree** (number of neighbors)?
- Given an adjacency matrix for an unweighted graph:
  - For each row (which means a node), find the number of 1s.
  - Compare all rows to see which row is the winner.
- This is exactly the algorithm of finding **the row with the largest row sum**!

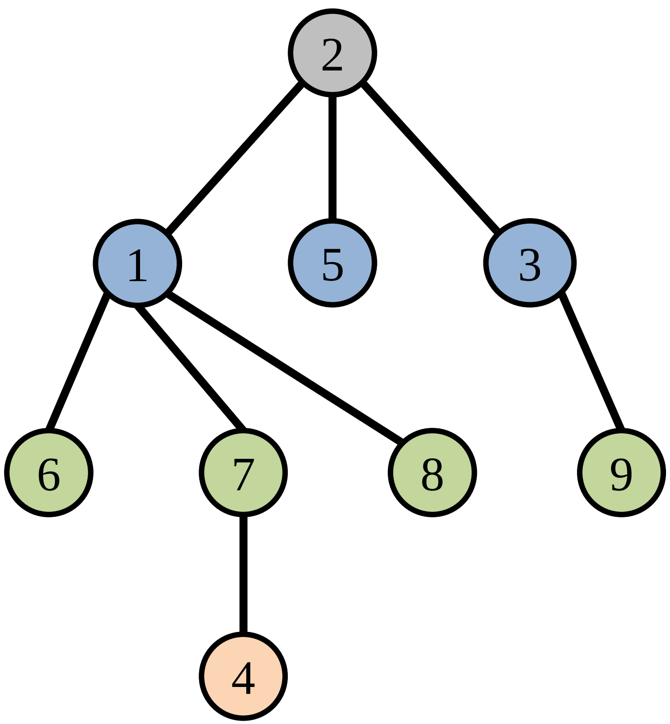
Minimum number of edges

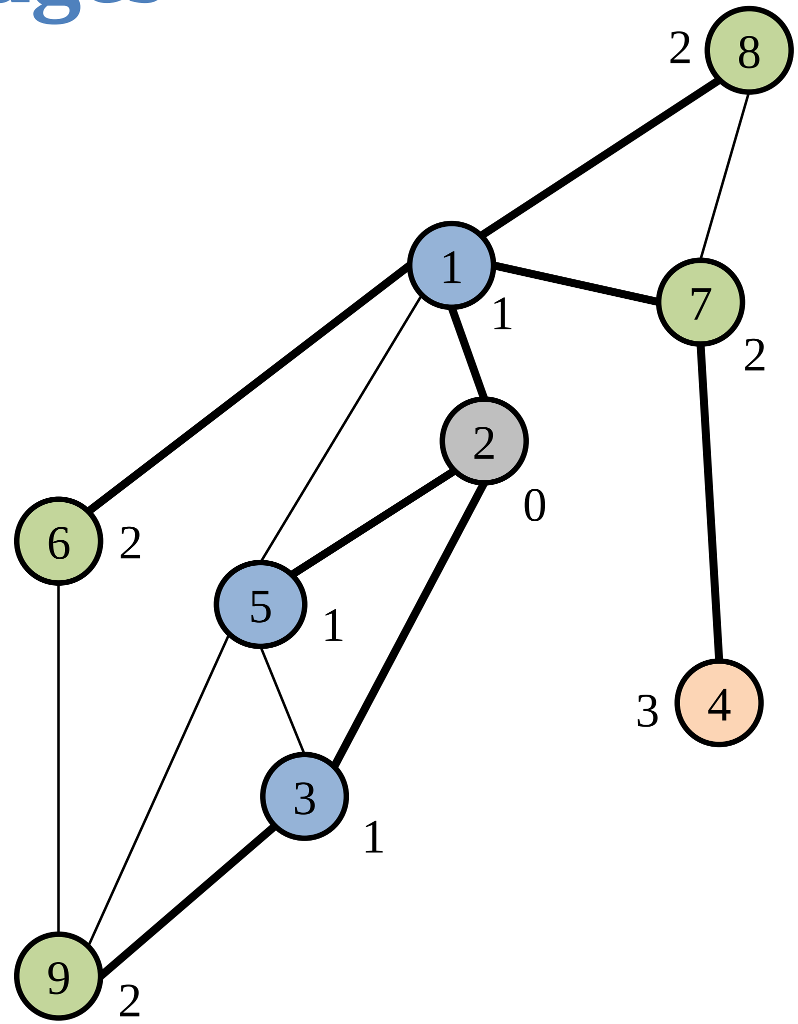
- Given an undirected unweighted graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, and a node s, please find the **minimum numbers of edges** one needs to move from s to all other nodes.
- In this graph, if $s = 2$, the value beside each node is the minimum number of edges one needs to move from node 2 to that node.



Minimum number of edges

- Those "shortest paths" (thick lines in the graph) together form a **spanning tree**.

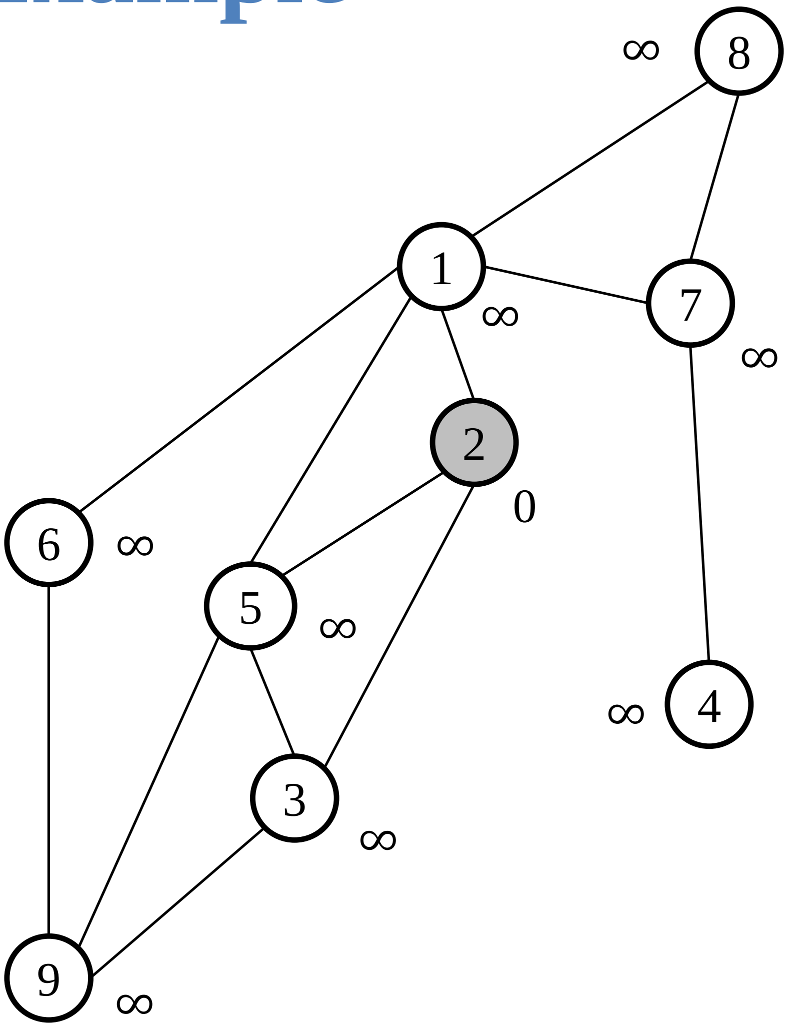


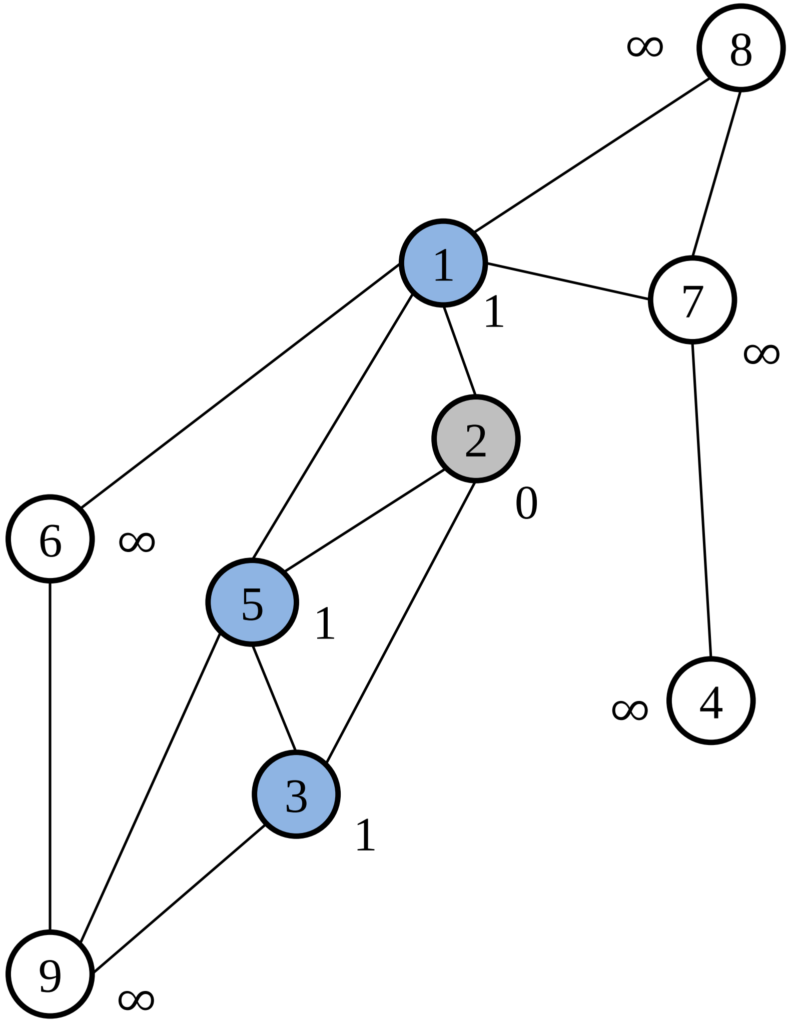


Minimum number of edges

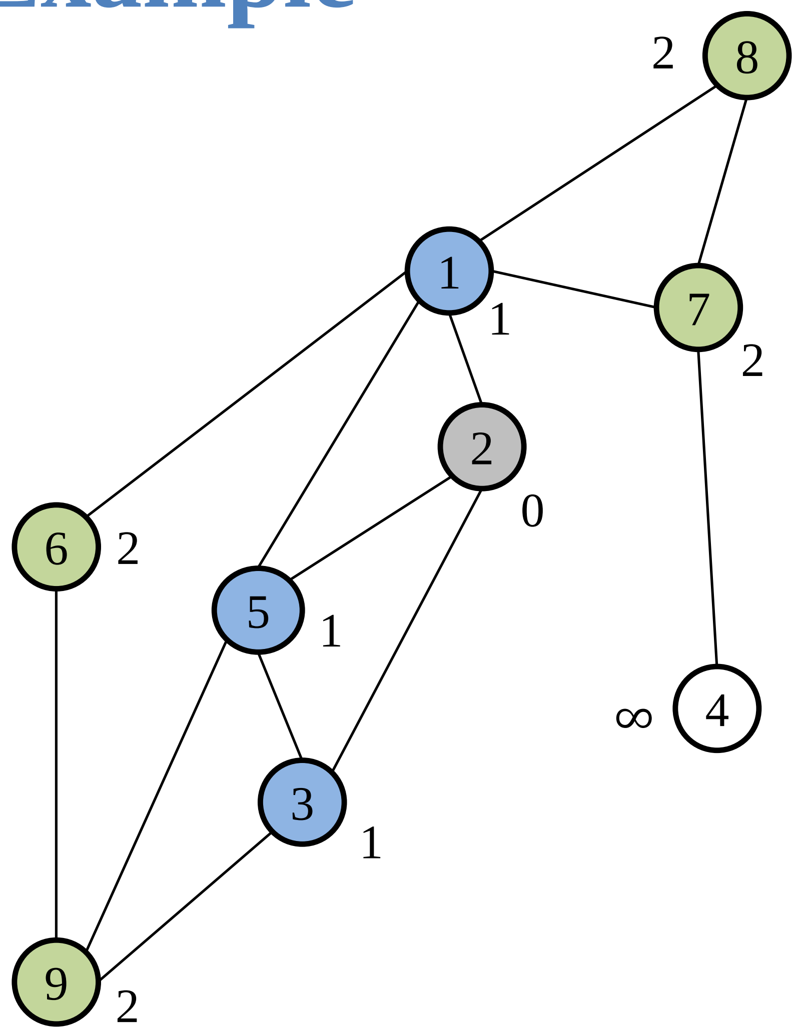
- To find the distances from s to all nodes, we use **breadth-first search** (**BFS**).
- Let all nodes have weights representing their distances from s.
  - First, we label s as 0 and all other nodes as ∞.
  - We then find the neighbors of s. Label them as 1.
  - For each node whose label is 1, find its neighbors that are currently labeled as ∞. Label them as 2.
  - Continue until all nodes are labeled.
- The graph should be **connected** (i.e., there is a path from s to any other node).

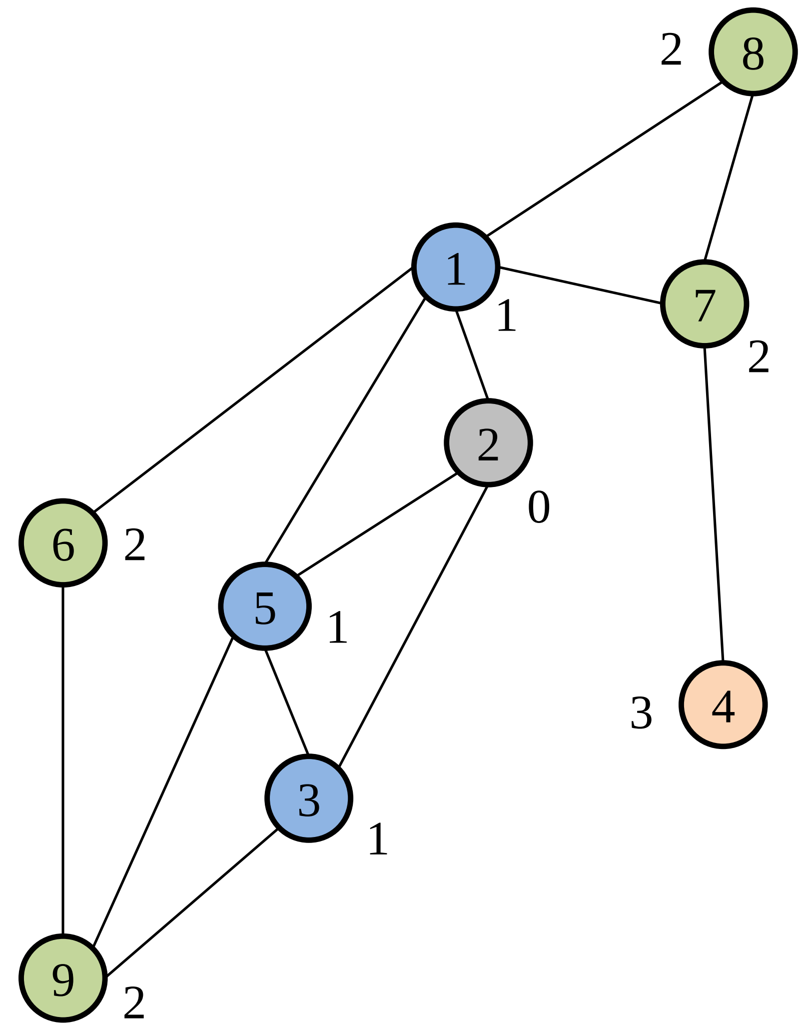
Example





Example





Implementation: function header

```
#include <iostream>
using namespace std;

const int MAX_NODE_CNT = 10;

// Input:
// - adjacent: the adjacency matrix
// - nodeCnt: number of nodes
// - source: the source node
// - dist: to store the distances from the source
// This function will find the distances from the source
// node to each node and put them in "dist"
void distFromSource(const bool adjacent[][MAX_NODE_CNT],
                    int dist[], int nodeCnt, int source);
```

Implementation: main function

```
int main()
{
  int nodeCnt = 5;
  bool adjacent[MAX_NODE_CNT][MAX_NODE_CNT]
    = {{1, 1, 0, 0, 1}, {1, 1, 1, 0, 0}, {0, 1, 1, 1, 0},
       {0, 0, 1, 1, 1}, {1, 0, 0, 1, 1}};
  int dist[MAX_NODE_CNT] = {0};
  int source = 0;

  distFromSource(adjacent, dist, nodeCnt, source);

  cout << "\nThe complete result: \n";
  for(int i = 0; i < nodeCnt; i++)
    cout << dist[i] << " ";

  return 0;
}
```

Implementation: function body

```
void distFromSource(const bool adjacent[][MAX_NODE_CNT],
                    int dist[], int nodeCnt, int source)
{
  for(int i = 0; i < nodeCnt; i++)
    dist[i] = nodeCnt; // why not infinity?

  dist[source] = 0;
  int curDist = 1;
  int complete = 1;

  // continue to the next page
```

Implementation: function body

```
  // continue from the previous page
  while(complete < nodeCnt) {
    for(int i = 0; i < nodeCnt; i++) { // one for a level
      if(dist[i] == curDist - 1) {
        for(int j = 0; j < nodeCnt; j++) { // from i to j
          if(adjacent[i][j] == true
             && dist[j] == nodeCnt) {
            dist[j] = curDist;
            complete++;
          }
        }
      }
    }
    curDist++;
  }
}
```

Complexity

- There is a three-level loop.
  - Each of the two for loops has n iterations, where n is the number of nodes.
  - In the worst case, the while loop has n iterations (if in each iteration we label only one node).
- Is the algorithm's complexity $O(n^3)$?
- Not really!
  - The most inner loop will be initiated only if its label equals **`curDist`** – 1.
  - For each node, this will be true for exactly once.
  - In the worst case, the while loop and first for loop together give $O(n^2)$.
  - The most inner loop gives another $O(n^2)$.
  - The overall complexity is $O(n^2 + n^2) = O(n^2)$.

Remarks

- The name "breadth-first search" comes from the fact that "we reach all neighbors of a node before we reach neighbors of neighbors."
  - Please search for breadth-first search and "**depth-first search**" to learn more.
- BFS can be done with a lower complexity.
  - $O(n + m)$, where m is the number of edges.
  - By using a data structure "queue."