Introduction to Number Theory and Algorithms

Christian J. Rudder

August 2024

Contents

This page is left intentionally blank.

Prerequisites

Theorem 0.1: Common Derivatives

Power Rule: For $n \neq 0$	$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$. E.g., $\frac{d}{dx}(x^2) = 2x$
Derivative of a Constant:	$\frac{d}{dx}(c) = 0$. E.g., $\frac{d}{dx}(5) = 0$
Derivative of ln:	$\frac{d}{dx}(\ln x) = \frac{1}{x}$
Derivative of $\log_a$:	$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
Derivative of $\sqrt{x}$:	$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$
Derivative of function $f(x)$:	$\frac{d}{dx}(x) = 1$. E.g., $\frac{d}{dx}(5x) = 5$
Derivative of the Exponential Function:	$\frac{d}{dx}(e^x) = e^x$

Theorem 0.2: L'Hopital's Rule

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

Theorem 0.3: Exponents Rules

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

Note: The $:=$ symbol is short for "is defined as." For example, $x := y$ means x is defined as y.

Definition 0.1: Logarithm

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm x base a is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \iff a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

Tip: To remember the order $log_a(x) = a^y$, think, "base a," as a is the base of our *log* and y.

Theorem 0.4: Logarithm Rules

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b \log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

Definition 0.2: Permutations

Let n be a positive integer. Then the number of distinct ways to arrange n objects in order is it's *permutation*. Denoted:

$$n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

Definition 0.3: Combinations

Let n and k be positive integers. Where order doesn't matter, the number of distinct ways to choose k objects from n objects is it's *combination*. Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "n choose k.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient*.

Theorem 0.5: Binomial Theorem

Let a and b be real numbers, and n a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n-1} a b^{n-1} + \binom{n}{n} b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \leq k \leq n$.

Theorem 0.6: Binomial Expansion of 2^n

For any non-negative integer n, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

Definition 0.4: Well-Ordering Principle

Every non-empty set of positive integers has a least element.

Definition 0.5: "Without Loss of Generality"

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

Theorem 0.7: Pigeon Hole Principle

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute m pigeons into n pigeonholes, there must be at least one pigeonhole with more than one pigeon.

Theorem 0.8: Growth Rate Comparisons

Let n be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Introduction to Runtime Complexity

1.1 Asymptotic Notation

Asymptotic analysis is a method for describing the limiting behavior of functions as inputs grow infinitely.

Definition 1.1: Asymptotic

Let $f(n)$ and $g(n)$ be two functions. As n grows, if $f(n)$ grows closer to $g(n)$ never reaching, we say that "$f(n)$ is **asymptotic** to $g(n)$."

We call the point where $f(n)$ starts behaving similarly to $g(n)$ the **threshold** n_0. After this point n_0, $f(n)$ follows the same general path as $g(n)$.

Definition 1.2: Big-O: (Upper Bound)

Let f and g be functions. $f(n)$ our function of interest, and $g(n)$ our function of comparison.

Then we say $f(n) = O(g(n))$, "$f(n)$ **is big-O of** $g(n)$," if $f(n)$ grows no faster than $g(n)$, up to a constant factor. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq f(n) \leq c \cdot g(n)$$

Represented as the ratio $\dfrac{f(n)}{g(n)} \leq c$ for all $n \geq n_0$. Analytically we write,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Meaning, as we chase infinity, our numerator grows slower than the denominator, bounded, never reaching infinity.

Examples:

(i.) $3n^2 + 2n + 1 = O(n^2)$

(ii.) $n^{100} = O(2^n)$

(iii.) $\log n = O(\sqrt{n})$

Proof 1.1: $\log n = O(\sqrt{n})$

We setup our ratio:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$$

Since $\log n$ and $\sqrt{n}$ grow infinitely without bound, they are of indeterminate form $\frac{\infty}{\infty}$. We apply L'Hopital's Rule, which states that taking derivatives of the numerator and denominator will yield an evaluateable limit:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{d}{dn}\log n}{\frac{d}{dn}\sqrt{n}}$$

Yielding derivatives, $\log n = \frac{1}{n}$ and $\sqrt{n} = \frac{1}{2\sqrt{n}}$. We substitute these back into our limit:

$$\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

Our limit approaches 0, as we have a constant factor in the numerator, and a growing denominator. Thus, $\log n = O(\sqrt{n})$, as $0 < \infty$. ■

Definition 1.3: Big-Ω: (Lower Bound)

The symbol Ω reads "Omega." Let f and g be functions. Then $f(n) = \Omega(g(n))$ if $f(n)$ grows no slower than $g(n)$, up to a constant factor. I.e., lower bounded by $g(n)$. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq c \cdot g(n) \leq f(n)$$

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Meaning, as we chase infinity, our numerator grows faster than the denominator, approaching 0 asymptotically.

Examples: $n! = \Omega(2^n)$; $\frac{n}{100} = \Omega(n)$; $n^{3/2} = \Omega(\sqrt{n})$; $\sqrt{n} = \Omega(\log n)$

Definition 1.4: Big Θ: (Tight Bound)

The symbol Θ reads "Theta." Let f and g be functions. Then $f(n) = \Theta(g(n))$ if $f(n)$ grows at the same rate as $g(n)$, up to a constant factor. I.e., $f(n)$ is both upper and lower bounded by $g(n)$. Let n_0 be our asymptotic threshold, and $c_1 > 0, c_2 > 0$ be some constants. Then, for all $n \geq n_0$,

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

Meaning, as we chase infinity, our numerator grows at the same rate as the denominator.

Examples: $n^2 = \Theta(n^2)$; $2n^3 + 2n = \Theta(n^3)$; $\log n + \sqrt{n} = \Theta(\sqrt{n})$.

Tip: To review:

- **Big-O:** $f(n) < g(n)$ (Upper Bound); $f(n)$ grows no faster than $g(n)$.
- **Big-Ω:** $f(n) > g(n)$ (Lower Bound); $f(n)$ grows no slower than $g(n)$.
- **Big-Θ:** $f(n) = g(n)$ (Tight Bound); $f(n)$ grows at the same rate as $g(n)$.

Theorem 1.1: Types of Asymptotic Behavior

The following are common relationships between different types of functions and their asymptotic growth rates:

- **Polynomials.** Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.
  E.e., $3n^2 + 2n + 1$ is $\Theta(n^2)$.
- **Logarithms.** $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. That is, logarithmic functions in different bases have the same growth rate.
  E.g., $\log_2 n$ is $\Theta(\log_3 n)$.
- **Logarithms and Polynomials.** For every $d > 0$, $\log n$ is $O(n^d)$. This indicates that logarithms grow slower than any polynomial.
  E.g., $\log n$ is $O(n^2)$.
- **Exponentials and Polynomials.** For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$. This means that exponentials grow faster than any polynomial.
  E.e., n^2 is $O(2^n)$.

1.2 Evaluating Algorithms

When analyzing algorithms, we are interested in two primary factors: time and space complexity.

Definition 2.1: Time Complexity

The **time complexity** of an algorithm is the amount of time it takes to run as a function of the input size. We use asymptotic notation to describe the time complexity of an algorithm.

Definition 2.2: Space Complexity

The **space complexity** of an algorithm is the amount of memory it uses to store inputs and subsequent variables during the algorithm's execution. We use asymptotic notation to describe the space complexity of an algorithm.

Below is an example of a function and its time and space complexity analysis.

Function 2.1: Arithmetic Series - `Fun1`(A)

Computes a result based on a length-n array of integers:

Input: A length-n array of integers.
Output: An integer p computed from the array elements.

```
1 Function Fun1(A):
2 |  p ← 0;
3 |  for i ← 1 to n − 1 do
4 |  |  for j ← i + 1 to n do
5 |  |  |  p ← p + A[i] · A[j];
6 |  |  end
7 |  end
8 return p
```

Time Complexity: For $f(n) :=$ `Fun1(`A`)`, $f(n) = \frac{n^2}{2} = O(n^2)$. This is because the function has a nested loop structure, where the inner for-loop runs $n - i$ times, and the outer for-loop runs $n - 1$ times. Thus, the total number of iterations is $\sum_{i=1}^{n-1} n - i = \frac{n^2}{2}$.

Space Complexity: We yield $O(n)$ for storing an array of length n. The variable p is $O(1)$ (constant), as it is a single integer. Hence, $f(n) = n + 1 = O(n)$.

Additional Example: Let $f(n, m) = n^2m + m^3 + nm^3$. Then, $f(n, m) = O(n^2m + m^3)$. This is because both n and m must be accounted for. Our largest n term is n^2m, and our largest m term is m^3 both dominate the expression. Thus, $f(n, m) = O(n^2m + m^3)$.

— 2 —

Proving Algorithms

2.1 Stable Matchings

In proving the correctness of algorithms we introduce the stable matching problem. A combinatorial optimization problem that seeks to find the best possible matching between two sets of elements. When we say "best possible matching," we mean that the matching is stable, and that there is no other matching that is better.

Definition 1.1: Stable Matching

A matching is **stable** if there is no pair of elements that prefer each other over their current match.

Definition 1.2: Unstable Matching

A matching is **unstable** if there is a pair of elements that prefer each other over their current match.

I.e., in verifying a stable matching, if any one pair of elements switch partners, the matching is unstable. If no pairs swap, the matching is stable.

Scenario: *Lunch Time*

Imagine it's lunch time at elementary school, and a group of kids E ={Ena, Eda} swap lunches with A ={Ava, Adi}. They each have a list of preferences from favorite to least favorite. We visualize the following preferences:

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Ava	Adi

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

Observe the following matchings:

(1.) Pairs, Ena-Ava, Eda-Adi swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Ava	Adi

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

This matching is **stable**. Ena and Ava prefer each other's lunches. Eda will ask Ava to trade, and Ava will refuse because she prefers Ena's lunch. Adi does the same with Ena, but they also refuse.

Tip: If it's hard to keep track who is who, here's a possible order to read in: Ena got Ava, and Ena is their 1st choice. Eda got Adi, and Eda is their 2nd choice.

Changing the preference tables,

(2.) Pairs, Ena-Adi, Eda-Ava swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Adi	Ava

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

This matching is **unstable**. Ena and Ava would rather eat each other's lunches.

Definition 1.3: Unique Stable Matching

A matching is **uniquely stable** if between two sets of elements, there is only one possible stable matching.

Example: If everyone uniquely prefers each other, there is only one stable matching.

(3.) Pairs, Ena-Ava, Eda-Adi swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Adi	Ava

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Eda	Ena

This matching is a **unique stable matching**. If rather Ena-Adi and Eda-Ava (2nd-choice parings), then both pairs would end up swapping to their 1st-choice.

2.2 Gale-Shapley Algorithm

We will now introduce the Gale-Shapley algorithm, for which we will prove its correctness, time complexity, and space complexity.

Theorem 2.1: Gale-Shapley Algorithm

The **Gale-Shapley algorithm** is a method for finding a stable matching between two sets of elements. It is also known as the **Deferred Acceptance Algorithm**.

Algorithm: Given sets $E = e_1, \ldots, e_n$ and $A = a_1, \ldots, a_n$. Then find a stable matching:

(i.) Each $e_i \in E$ proposes to their most preferred a_j.

(ii.) For each $a_j \in A$:

(a.) If a_j is free, they accept the proposal.

(b.) If a_j is already matched, a_j either accepts or rejects. If a_j accepts, the previous match is broken.

Each e_i continues to propose to their next most preferred a_j until all e_i are matched.

Claims:

1. At least one stable matching is guaranteed.
2. Unless the table is unique, the proposing will always get their best choice unless it conflicts with another proposer.

First we will prove the correctness, then implement the algorithm and analyze its time and space complexity.

Proof 2.1: Gale-Shapley Algorithm Correctness

Claim 1: Suppose, for sake of contradiction, that some $a_j \in A$ is not matched upon termination of the algorithm. Then some $e_i \in E$ is also not matched assuming $|E| = |A|$. Then e_i must have not proposed to a_j, contradicting that e_i proposed to all elements of A. Thus, the program only terminates when all e_i are matched.

Claim 2: Suppose E proposes to A with unique first choices. Then all $a_i \in A$ must accept their first proposal. Now suppose $e_i, e_j \in E$ have a conflicting choice a_i. Then a_i gets their preference only in that case.

■

Function 2.1: Gale-Shapley Algorithm - `GS(E,A)`

Finds a stable matching between two sets of elements:

Input: Two sets, E and A, of equal size.
Output: A stable matching between E and A.

```
1  Function GS(E,A):
2      M ← ∅;
3      while there is some unmatched element in E do
4          e ← next unmatched element in E;
5          a ← next avalaible preferred choice of e;
6          if a is not yet matched then
7              match e and a;
8              add the pair (e, a) to M;
9          end
10         else
11             if a prefers e over their current match then
12                 match e and a, replacing the current match;
13                 update M accordingly;
14             end
15         end
16     end
17     return M;
18 return Matching M
```

Time Complexity: $O(n^2)$ time, where n is the number of elements in E and A. Worst-case, each element in E proposes to each element in A, i.e., $n \cdot n$ combinations to check.

Space Complexity: $O(n^2)$ space, where we store $|E| \cdot |A| = n \cdot n$ pairs.

— 3 —

Graphs and Trees

3.1 Paths and Connectivity

Graphs are similar to train networks or airline routes. They connect one location to another.

Definition 1.1: Graph

A **graph** is a collection of points, called **vertices** or **nodes**, connected by lines, called **edges**. Similarly to how a polygon has vertices connected by edges.

Definition 1.2: Undirected Graph

An **undirected graph** is a graph where the edges have no particular direction going both ways between nodes. A **degree** of a node is the number of edges connected to it.

Example: Figure (3.1) shows an undirected graph:

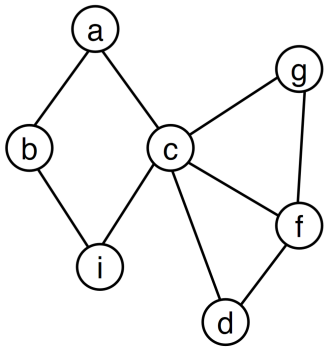


Figure 3.1: An undirected graph with 7 vertices and 9 edges.

Node a has a degree of 3, and node c has a degree of 4.

Definition 1.3: Directed Graph

A **directed graph** is where the edges have a specific direction from one node to another.

- The **indegree** of a node is the number of edges that point to it.
- The **outdegree** of a node is the number of edges that point from it.

Example: Figure (3.2) shows a directed graph:

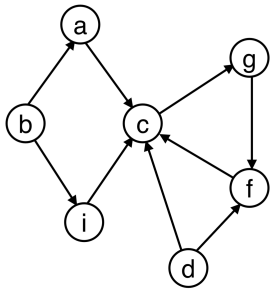


Figure 3.2: A directed graph with 7 vertices and 9 edges.

Node b has an outdegree of 2 and an indegree of 0. c has an indegree of 4 and an outdegree of 0.

Definition 1.4: Weighted Graph

A **weighted graph** is a graph where each edge has a numerical value assigned to it.

Example: Figure (3.3) shows a weighted graph:

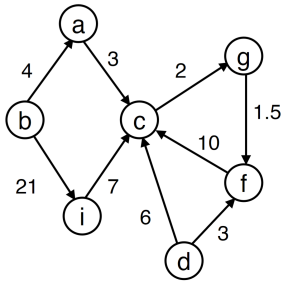


Figure 3.3: A weighted graph with 7 vertices and 9 edges.

Definition 1.5: Path

A **path** is a sequence of edges that connect a sequence of vertices. A path is **simple** if all nodes are distinct.

Example: In Figure (3.4), a simple path $h \leftrightarrow b \leftrightarrow i \leftrightarrow c \leftrightarrow d$ is shown:

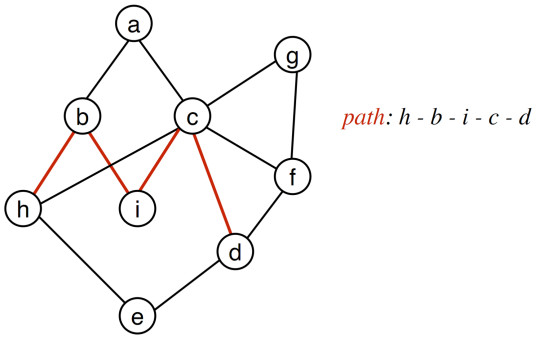


Figure 3.4: A graph with a simple path from h to d.

Definition 1.6: Connectivity

A graph is **connected** if there is a path between every pair of vertices.
A graph is **disconnected** if there are two vertices with no path between them.
Connected graphs of n nodes have at least $n - 1$ edges.

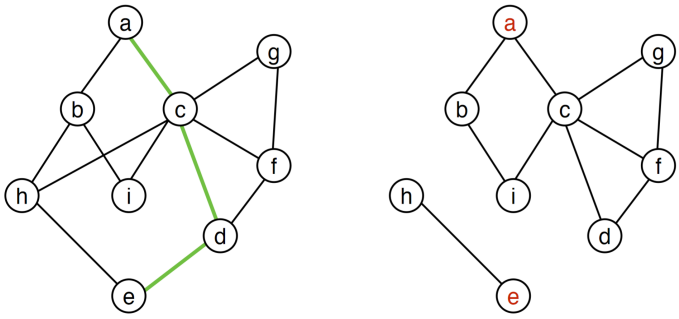


Figure 3.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

Definition 1.7: Adjacency Matrix

An **adjacency matrix** is an $n \times n$ matrix where such that $A[i][j] = 1$ if there is an edge between nodes i and j.

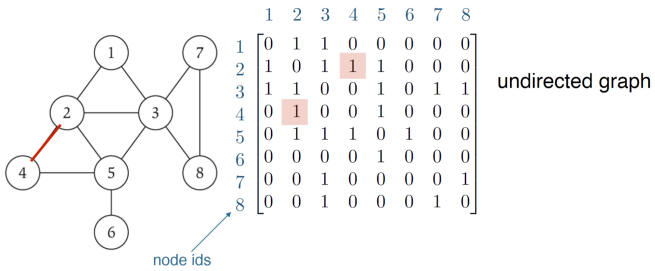


Figure 3.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

Theorem 1.1: Properties of Adjacency Matrix

The following properties hold for adjacency matrices:

- An undirected graph is symmetric about the diagonal.
- A directed graph is not symmetric about.
- A weighted graph has the weight of the edge instead of binary.

Space Complexity: $\Theta(n^2)$; **Time Complexities:**
Check edges $A[i][j]$**:** $\Theta(1)$; **List neighbors** $A[i]$: $\Theta(n)$; **List all edges:** $\Theta(n^2)$.

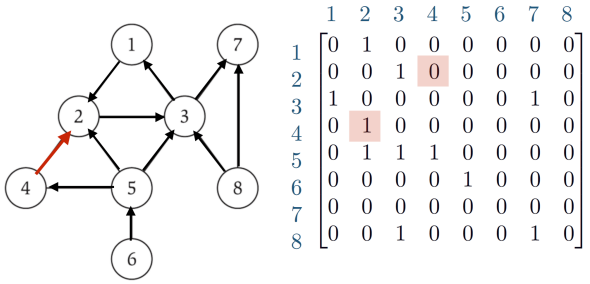


Figure 3.7: An adjacency matrix for a directed graph with path $4 \leftrightarrow 2$ highlighted ($A[2][4]$ and $A[4][2]$).

Definition 1.8: Adjacency List

An **adjacency list** is a list of keys where each key has a list of neighbors.

Often taking form as a dictionary or hash-table:
Space Complexity: $\Theta(n+m)$ for n nodes and m total edges; **Time Complexities:** **Check key:** $\Theta(1)$; **List neighbors:** $\Theta(|\text{outdegrees}|$ of key); **List all edges:** $\Theta(n+m)$; **Insert edge:** $\Theta(1)$.

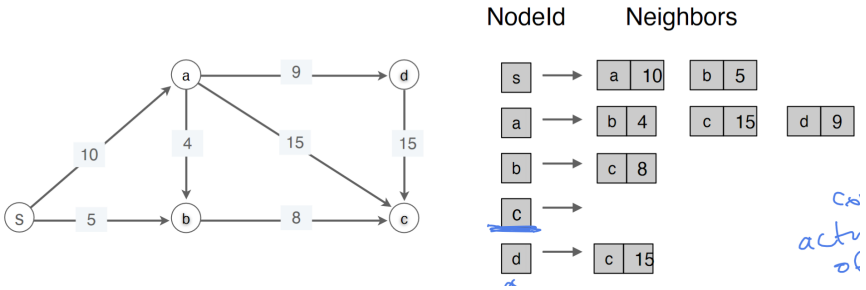


Figure 3.8: An adjacency list of a directed graph, c highlighted with no outdegrees.

3.2 Breath-First and Depth-First Search

Two general methods for traversing a graph are, **breadth-first search** and **depth-first search**.

Definition 2.1: Cycle

A **cycle** is a path that starts and ends at the same node.

Example: In the above Figure (3.7), $1 \to 2 \to 3 \to 1$ form a cycle.

Definition 2.2: Tree

A **tree** is a connected graph with no cycles. A **leaf-nodes** is the outer-most nodes of a tree. A **branch** is a path from the root to a leaf.

Theorem 2.1: Tree Identity

Let G be an undirected graph of n nodes. Then any two statements imply the third:

(i.) G is connected.

(ii.) G has $n-1$ edges.

(iii.) G has no cycles.

Definition 2.3: Rooted Trees

Let T be a tree with a designated root node r. Then each degree is considered an outdegree, called a **child**, and its indegree its **parent**.

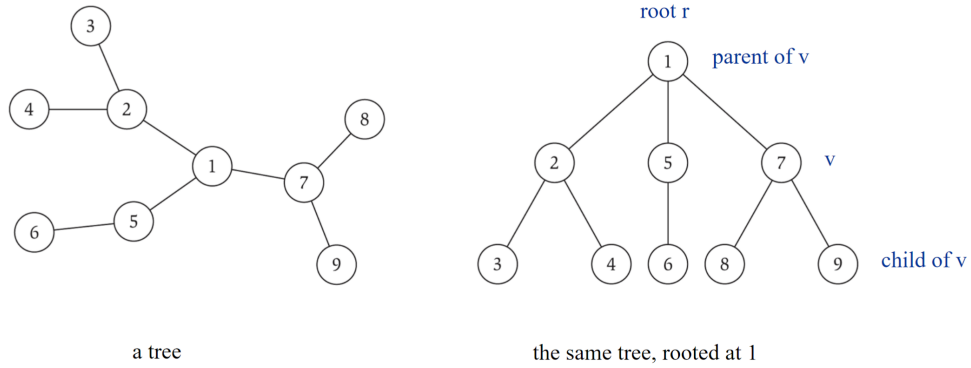


Figure 3.9: A rooted tree with root 1 and children $\{2, 5, 7\}$

Definition 2.4: Levels and Heights

The **level** of a node is the number of edges from the root. The **height** of a tree is the maximum level of any node.

Example: In Figure (3.9)'s rooted tree , node 5 has a level of 1, and the height of the tree is 2.

Definition 2.5: Breadth-First Search (BFS)

In a **breadth-first search**, we start at a node's children first before moving onto their children's children in level order.

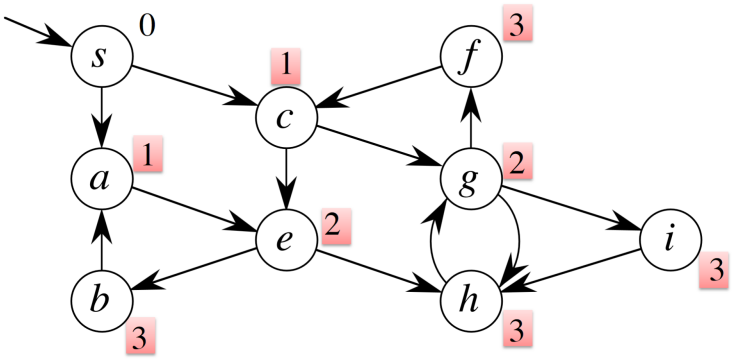


Figure 3.10: A BFS tree traversal preformed on a graph with each level enumerated

Theorem 2.2: Properties of BFS

BFS run on any graph T produces a tree T' with the following properties:

(i.) T' is a tree.

(ii.) T' is a rooted tree with the starting node as the root.

(iii.) The height of T' is the shortest path from the root to any node.

(iv.) Any sub-paths of T' are also shortest paths.

Proof 2.1: Proof of BFS

(i.) and (ii.) follow from the definition of a tree. (iii.) and (iv.) follow that since a tree contains a direct path to any given node in our parent child relationship, that path must be the shortest. ■

Tip: In a family tree, there is only one path from each ancestor to each descendant.

We create a BFS algorithm from what we know, though not the best implementation:

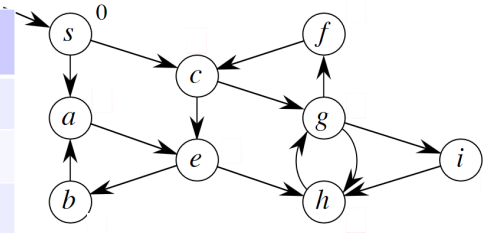
Function 2.1: BFS Algorithm - BFS(s)

Breadth-First Search starting from node s.

Input: Graph $G = (V, E)$ and starting node s.
Output: Levels of each vertex from s.

```
1  Function BFS(s):
2      for each v ∈ V do
3          Level[v] ← ∞;
4      end
5      Level[s] ← 0;
6      Add s to Q;
7      while Q not empty do
8          u ← Q.Dequeue();
9          for each v ∈ G[u] do
10             if Level[v] = ∞ then
11                 Add edge (u, v) to tree T (parent[v] = u);
12                 Add v to Q;
13                 Level[v] ← Level[u] + 1;
14             end
15         end
16     end
```

u	Queue contents before exploring u	… after exploring u
s	(empty)	*[a,c]*
a	*[c]*	*[c,e]*
c	*[e]*	*[e,g]*
e	*[g]*	*[g,b,h]*
g	*[b,h]*	*[b,h,f,i]*
b	*[h,f,i]*	*[h,f,i]*
h	*[f,i]*	*[f,i]*
f	*[i]*	*[i]*
i	(empty)	(empty)



Loop invariant: If the first node in the queue has level i, then the queue consists of nodes of level i possibly followed by nodes of level $i + 1$.

Consequence: Nodes are explored in increasing order of level.

Figure 3.11: A table showing the queue at each level of iteration

We analyze the time and space complexity in the below Figure (3.12):

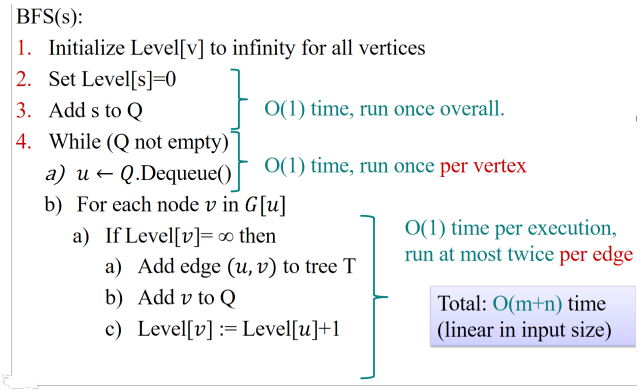


Figure 3.12: An analysis showing $O(m+n)$ for both time and space complexity

Proof 2.2: Claim 1 for BFS

Let s be the root of the BFS tree, then: **Proof:** Induction on the distance from s to u.

Base case ($u = s$): The code sets Level$[s] = 0$, and there is no path to find since the path has length 0.

Induction hypothesis: For every node u at distance $\leq i$, Claim 1 holds.

Induction step:

- Let v be a node at distance exactly $i+1$ from s. Let u be its parent in the BFS tree.
- The code sets Level$[v]$ = Level$[u]$ + 1.
- Let x be the last node before v on a shortest path from s to v. Since v is at distance $i+1$, then x must be at distance i, and so Level$[x] = i$ (by induction hypothesis).
- If $u = x$, we are done!
- If $u \neq x$, then it must be that u was explored before x, since otherwise x would be the parent of u.
- Since we explore nodes in order of level, Level$[u] \leq$ Level$[x] = i$.
- If Level$[u] = i$, then we are done.
- If Level$[u] < i$, then the path $s \sim u \rightarrow v$ has length at most i, which contradicts the assumption that the distance of v is $i+1$.

We conclude that Level$[u] = i$, Level$[v] = i+1$, and the path in the BFS tree that goes from s to u to v has length $i+1$. QED. ■

Definition 2.6: Tyes of Edges

In graphs we have three types of edges:

1. **Tree-edges:** An Edge present in a BFS tree.
2. **Forward-edges:** Edges from an ancestor to a descendant.
3. **Back-edges:** Edges from a descendant to an ancestor.
4. **Cross-edges:** Edges between which connect nodes that have no ancestor-descendant relationship.

Example: In Figure (3.10), $\{(s,c),(a,c),...\}$ are forward and tree edges, $\{(g,h)\}$ are cross edges.

The following arises from the above definitions:

Theorem 2.3: BFS does not Contain Back-edges

In a BFS tree, there are no back-edges, as they would create a cycle.

Note: In Figure (3.10), $b \to a$ is not a tree-edge, a is on level 1 and b on level 3. If this connection were to occur, it would create a cycle, breaking our tree.

Definition 2.7: Depth-First Search (DFS)

In a **depth-first search**, we recursively explore each an entire branch before moving onto the next.

Function 2.2: DFS Algorithm - `DFS`(G)

Depth-First Search on graph G (recursive).

Input: Graph $G = (V, E)$.
Output: Discovery and finishing times for each vertex.

```
1  Function DFS(G):
2      for each u ∈ G do
3          u.state ← unvisited;
4      end
5      time ← 0;
6      for each u ∈ G do
7          if u.state == unvisited then
8              DFS-Visit(u);
9          end
10     end
11 Function DFS-Visit(u):
12     time ← time + 1;
13     u.d ← time ;
14     // record discovery time;
15     u.state ← discovered;
16     for each v ∈ G[u] do
17         if v.state == unvisited then
18             DFS-Visit(v);
19         end
20     end
21     u.state ← finished;
22     time ← time + 1;
23     u.f ← time;
24     // record finishing time;
```

Time and Space Complexity: $O(n + m)$ where n is the number of vertices and m the number of edges.

Proof 2.3: DFS Correctness

Case 1: When s is discovered, there is a path of unvisited vertices from s to y. We use induction on the length L of the unvisited path from x to y.

- **Base case:** There is an edge (x, y), and y is unvisited.
- **Induction hypothesis:** Assume that the claim is true for all nodes reachable via k unvisited nodes.
- **Induction step:**
  - Consider u reachable via $k + 1$ unvisited nodes.
  - Let z be the last node on the path before y, and z is discovered from x (by I.H.).
  - The edge (z, y) will be explored from x, ensuring that y is eventually visited.

Thus, DFS-Visit(x) explores all nodes reachable from x through a path of unvisited nodes, as required. ■

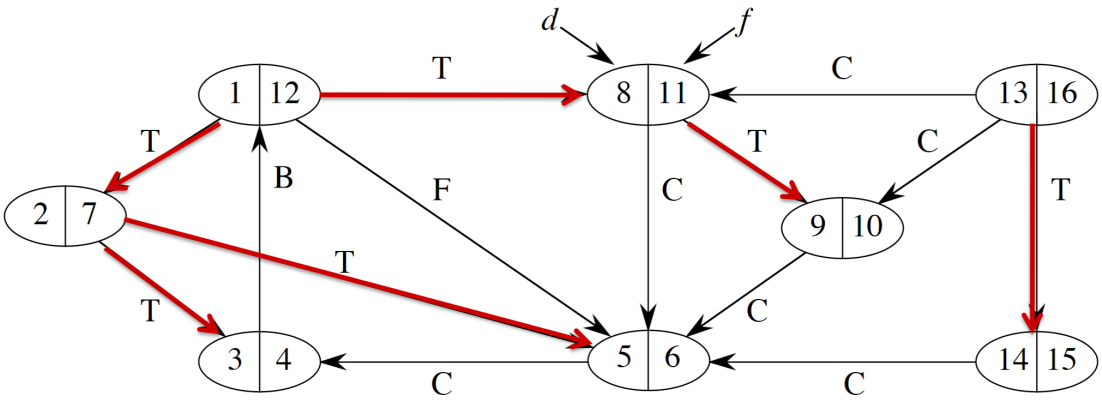


Figure 3.13: A graph showing our d discovered and f finished times, denoted in pairs (f, d).

Theorem 2.4: DFS and Cycles

Let DFS run on graph G, then:

$$(G \text{ has a cycle}) \iff (\text{DFS run reveals back edges})$$

Proof 2.4: Proof of Cycles and Back Edges

Proving (G has a cycle) $\Longleftarrow$ (DFS run reveals back edges): Every back edge creates a cycle.
Proving (G has a cycle) $\Longrightarrow$ (DFS run reveals back edges) Suppose G has a cycle:

- Let u_1 be the first discovered vertex in the cycle, and let u_k be its predecessor in the cycle.
- u_k will be discovered while exploring u_1.
- The edge (u_k, u_1) will be a back edge.

■

3.3 Directed-Acyclic Graphs & Topological Ordering

Graphs may represent a variety of relationships, such as dependencies between tasks or the flow of information.

Definition 3.1: Directed-Acyclic Graph (DAG)

directed-acyclic graph is a directed graph with no cycles.

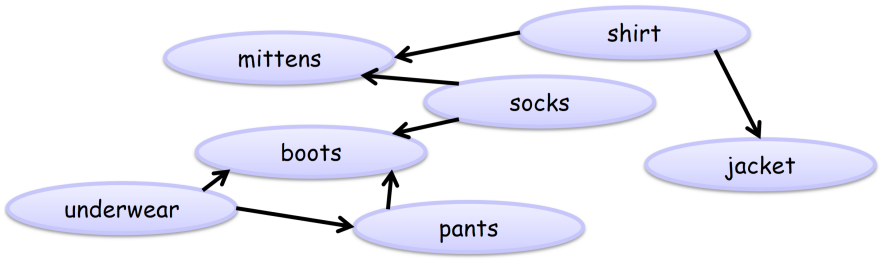


Figure 3.14: A DAG depicted by getting dressed for winter.

For each node to be processed its **dependencies** or parents must be processed first.

Definition 3.2: Topological Ordering

Given a graph, a **topological ordering** is a linear ordering of nodes such that for every edge (u, v), u comes before v.

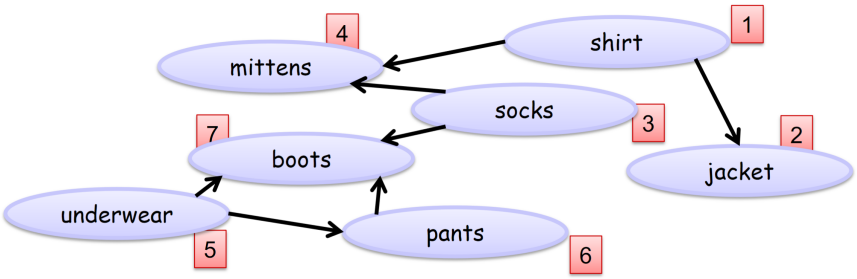


Figure 3.15: A topological ordering of the DAG in Figure (3.14) enumerated in red.

Another possible ordering of $[1, 2, 3, 4, 5, 6, 7]$ is $[5, 6, 1, 2, 3, 4, 7]$, as $5 \to 6$ is independent.

Theorem 3.1: Topological Sort

iven a DAG, a topological ordering can be found.

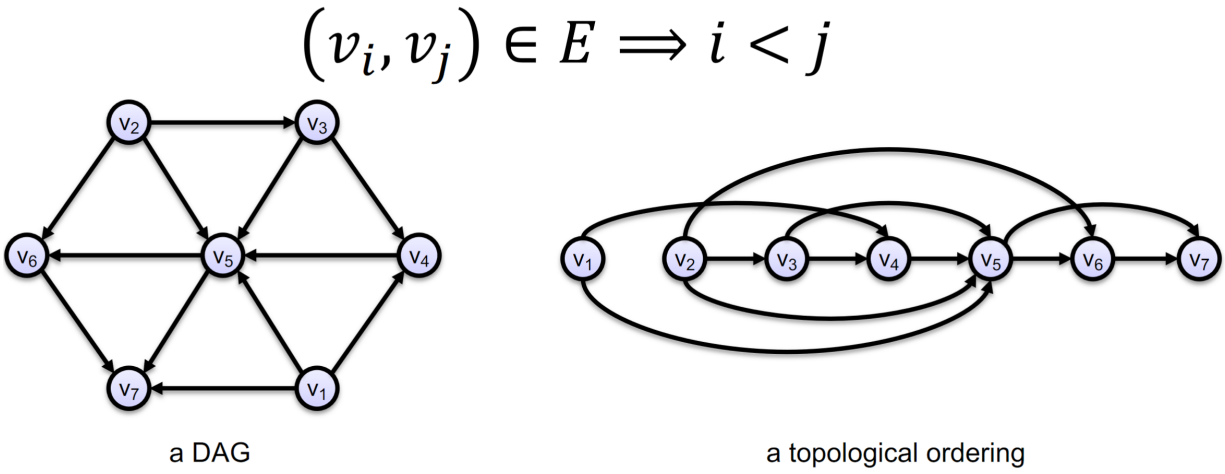


Figure 3.16: A topological sorting of a DAG E and v nodes

Proof 3.1: Topological Sort via DFS

Lemma: In a directed graph G, if (note necessarily acyclic):

- (u, v) is an edge, and
- v is not reachable from u,

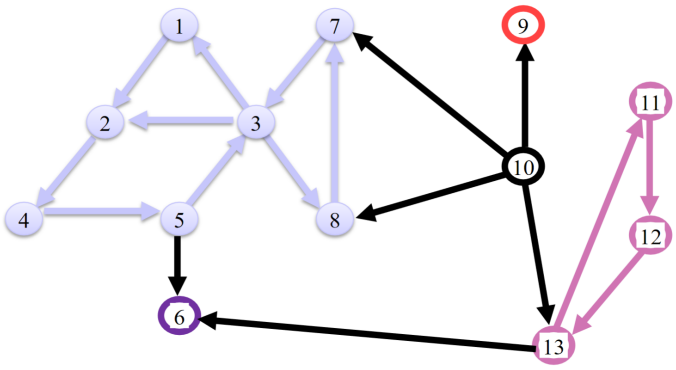
Then in every run of DFS, $u.f > v.f$.
Proof:

- If v is started before u, then the DFS-Visit(v) will terminate without reaching u (because there is no path to u).
- If u is started before v, then the edge (u, v) will be explored before u is finished.

Therefore, in all cases, $u.f > v.f$. ■

Definition 3.3: Strongly Connected Components

A **strongly connected component** is a subgraph where every node is reachable from every other node. Then we say $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are **mutually reachable**.



- Observation. Two SCCs are either disjoint or equal.
- If we contract the SC components in one node we get an *acyclic* graph.

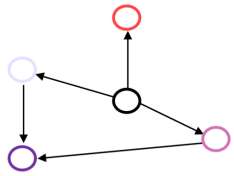


Figure 3.17: A graph with 5 strongly connected components.