Week 15 - Wednesday

COMP 2230

Last time

- Reviewed second third of the course

Questions?

Assignment 6

Logical warmup

- A customer wants change for a dollar
- The cashier cannot make change for a dollar
- The customer asks for change for a half dollar
- The cashier still can't
- The (increasingly annoyed) customer asks for change for a quarter, a dime, and a nickel, but the cashier still can't
- However, the cashier has $1.15 in coins
- Which coins are in the cash register?

Final exam

- **Final exam:**
  - **Wednesday, April 29, 2026**
  - **8:00 to 10:00 a.m.**
  - **50% longer than previous exams, but you'll have 100% more time**
- There will be short answer, diagrams, and proofs
- It will be comprehensive but weighted toward the last quarter of the course

Graphs and Trees

Graphs

- A **graph** G is made up of two finite sets
  - **Vertices**: $V(G)$
  - **Edges**: $E(G)$
- Each edge is connected to either one or two vertices called its **endpoints**
- An edge with a single endpoint is called a **loop**
- Two edges with the same sets of endpoints are called **parallel**
- Edges are said to **connect** their endpoints
- Two vertices that share an edge are said to be **adjacent**
- A graph with no edges is called **empty**

Special graphs

- A **simple graph** does not have any loops or parallel edges
- Let n be a positive integer
- A **complete graph** on n vertices, written K_n, is a simple graph with n vertices such that every pair of vertices is connected by an edge
- A **bipartite graph** is a simple graph whose vertices can be partitioned into sets X and Y such that there are no edges from any vertex in X to another vertex in X and there are no edges from any vertex in Y to another vertex in Y
- A **complete bipartite graph on** (m, n) **vertices**, written $K_{m,n}$ is a simple graph with a set of m vertices and a disjoint set of n vertices such that:
  - There is an edge from each of the m vertices to each of the n vertices
  - There are no edges among the set of m vertices
  - There are no edges among the set of n vertices
- A **subgraph** is a graph whose vertices and edges are a subset of another graph

Degree

- The **degree of a vertex** is the number of edges that are incident on the vertex
- The **total degree of a graph** G is the sum of the degrees of all of its vertices
- What's the relationship between the degree of a graph and the number of edges it has?
- What's the degree of a complete graph with n vertices?
- Note that the number of vertices with odd degree must be even

Example

- What is the maximum number of edges a simple disconnected graph with n vertices can have?
- Prove it.

Definitions

- A **walk from v to w** is a finite alternating sequence of adjacent vertices and edges of G, starting at vertex v and ending at vertex w
  - A walk must begin and end at a vertex
- A **path from v to w** is a walk that does not contain a repeated edge
- A **simple path from v to w** is a path that does not contain a repeated vertex
- A **closed walk** is a walk that starts and ends at the same vertex
- A **circuit** is a closed walk that does not contain a repeated edge
- A **simple circuit** is a circuit that does not have a repeated vertex other than the first and last

Circuits

- If a graph is connected, non-empty, and every node in the graph has even degree, the graph has an **Euler circuit**
- Algorithm to find one:
  1. Pick an arbitrary starting vertex
  2. Move to an adjacent vertex and remove the edge you cross from the graph
     - Whenever you choose such a vertex, pick an edge that will not disconnected the graph
  3. If there are still uncrossed edges, go back to Step 2
- A **Hamiltonian circuit** must visit every vertex of a graph exactly once (except for the first and the last)
- If a graph G has a Hamiltonian circuit, then G has a subgraph H with the following properties:
  - H contains every vertex of G
  - H is connected
  - H has the same number of edges as vertices
  - Every vertex of H has degree 2

Matrix representations of graphs

- To represent a graph as an **adjacency matrix**, make an $n \times n$ matrix a, where n is the number of vertices
- Let the nonnegative integer at a_{ij} give the number of edges from vertex i to vertex j
- A simple graph will always have either 1 or 0 for every location
- We can find the number of walks of length k that connect two vertices in a graph by raising the adjacency matrix of the graph to the kth power

Isomorphism

- We call two graphs isomorphic if we can relabel one to be the other
- Graph isomorphisms define equivalence classes
  - **Reflexive:** Any graph is clearly isomorphic to itself
  - **Symmetric:** If x is isomorphic to y, then y must be isomorphic to x
  - **Transitive:** If x is isomorphic to y and y is isomorphic to z, it must be the case that x is isomorphic to z

Isomorphism invariants

- A property is called an **isomorphism invariant** if its truth or falsehood does not change when considering a different (but isomorphic) graph
- These **cannot** be used to prove isomorphism, but they can be used to disprove isomorphism
- 10 common isomorphism invariants:
  1. Has n vertices
  2. Has m edges
  3. Has a vertex of degree k
  4. Has m vertices of degree k
  5. Has a circuit of length k
  6. Has a simple circuit of length k
  7. Has m simple circuits of length k
  8. Is connected
  9. Has an Euler circuit
  10. Has a Hamiltonian circuit

Trees

- A **tree** is a graph that is **circuit-free** and **connected**
- Any tree with more than one vertex has at least two vertices of degree 1
- If a vertex in a tree has degree 1 it is called a **terminal vertex** (or **leaf**)
- All vertices of degree greater than 1 in a tree are called **internal vertices** (or **branch vertices**)
- For any positive integer n, a tree with n vertices must have $n - 1$ edges

Rooted trees

- In a **rooted tree**, one vertex is distinguished and called the **root**
- The **level** of a vertex is the number of edges along the unique path between it and the root
- The **height** of a rooted tree is the maximum level of any vertex of the tree
- The **children** of any vertex v in a rooted tree are all those nodes that are adjacent to v and one level further away from the root than v
- Two distinct vertices that are children of the same parent are called **siblings**
- If w is a child of v, then v is the parent of w
- If v is on the unique path from w to the root, then v is an ancestor of w and w is a descendant of v

Binary trees

- A **binary tree** is a rooted tree in which every parent has at most two children
- Each child is designated either the **left child** or the **right child**
- In a **full binary tree**, each parent has exactly two children
- Given a parent v in a binary tree, the **left subtree** of v is the binary tree whose root is the left child of v
- Ditto for **right subtree**

Spanning Trees

Spanning trees

- A **spanning tree** for a graph G is a subgraph of G that contains every vertex of G and is a tree
- Some properties:
  - Every connected graph has a spanning tree
    - Why?
  - Any two spanning trees for a graph have the same number of edges
    - Why?

Weighted graphs

- In computer science, we often talk about **weighted graphs** when tackling practical applications
- A weighted graph is a graph for which each edge has a real number **weight**
- The sum of the weights of all the edges is the **total weight** of the graph
- Notation: If e is an edge in graph G, then $w(e)$ is the weight of e and $w(G)$ is the total weight of G
- A **minimum spanning tree** (**MST**) is a spanning tree of lowest possible total weight

Finding a minimum spanning tree

- Kruskal's algorithm gives an easy to follow technique for finding an MST on a weighted, connected graph
- Informally, go through the edges, adding the smallest one, unless it forms a circuit
- **Algorithm:**
  - Input: Graph G with n vertices
  - Create a subgraph T with all the vertices of G (but no edges)
  - Let E be the set of all edges in G
  - Set $m = 0$
  - While $m < n - 1$
    - Find an edge e in E of least weight
    - Delete e from E
    - If adding e to T doesn't make a circuit
      - Add e to T
      - Set $m = m + 1$
  - Output: T

Prim's algorithm

- Prim's algorithm gives another way to find an MST
- Informally, start at a vertex and add the next closest node not already in the MST
- **Algorithm:**
  - Input: Graph G with n vertices
  - Let subgraph T contain a single vertex v from G
  - Let V be the set of all vertices in G except for v
  - For i from 1 to $n - 1$
    - Find an edge e in G such that:
      - e connects T to one of the vertices in V
      - e has the lowest weight of all such edges
    - Let w be the endpoint of e in V
    - Add e and w to T
    - Delete w from V
  - Output: T

Growth of Functions

Growth of functions

- Power functions:
  - $p_a(x) = x^a$ where $a, x \geq 0$
- Increasing and decreasing functions
- Let f and g be real-valued functions defined on the same set of nonnegative real numbers
  - f **is of order at least** g, written $f(x)$ is $\Omega(g(x))$, iff there is a positive $A \in \mathbb{R}$ and a nonnegative $a \in \mathbb{R}$ such that
    - $A|g(x)| \leq |f(x)|$ for all $x > a$
  - f **is of order at most** g, written $f(x)$ is $O(g(x))$, iff there is a positive $B \in \mathbb{R}$ and a nonnegative $b \in \mathbb{R}$ such that
    - $|f(x)| \leq B|g(x)|$ for all $x > b$
  - f **is of order** g, written $f(x)$ is $\Theta(g(x))$, iff there are positive $A, B \in \mathbb{R}$ and a nonnegative $k \in \mathbb{R}$ such that
    - $A|g(x)| \leq |f(x)| \leq B|g(x)|$ for all $x > k$

Polynomials

- Let $f(x)$ be a polynomial with degree n
  - $f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \ldots + a_1 x + a_0$
- By extension from the previous results, if a_n is a positive real, then
  - $f(x)$ is $O(x^s)$ for all integers $s \geq n$
  - $f(x)$ is $\Omega(x^r)$ for all integers $r \leq n$
  - $f(x)$ is $\Theta(x^n)$

Running time for algorithms

- First, assume that the number of operations performed by A on input size n is dependent only on n, not the values of the data
  - If $f(n)$ is $\Theta(g(n))$, we say that A **is** $\Theta(g(n))$ or that A **is of order** $g(n)$
- If the number of operations depends not only on n but also on the values of the data
  - Let $b(n)$ be the **minimum** number of operations where $b(n)$ is $\Theta(g(n))$, then we say that **in the best case**, A **is** $\Theta(g(n))$ or that A **has a best case order of** $g(n)$
  - Let $w(n)$ be the **maximum** number of operations where $w(n)$ is $\Theta(g(n))$, then we say that **in the worst case**, A **is** $\Theta(g(n))$ or that **A has a worst case order of** $g(n)$

Computing running time

- With a single `for` (or other) loop, we simply count the number of operations that must be performed
- When loops do not depend on each other, we can simply multiply their iterations (and asymptotic bounds)
- When loops depend on each other, we should analyze the loops like a series
- Repeated divisions by a constant k **often** lead to a $\log_k n$ term

Formal Languages

Rules

- We say that a **language** is a set of strings
- A **string** is an ordered n-tuple of elements of an alphabet Σ or the empty string ε (which has no characters)
- An alphabet Σ is a finite set of characters

Notation

- Let Σ be some alphabet
- For any nonnegative integer n, let
  - Σ^n be the set of all strings over Σ that have length n
  - Σ^+ be the set of all strings over Σ that have length at least 1
  - Σ^* be the set of all strings over Σ
- Σ^* is called the **Kleene closure of** Σ and the $*$ operator is often called the **Kleene star**

Languages defined by a regular expression

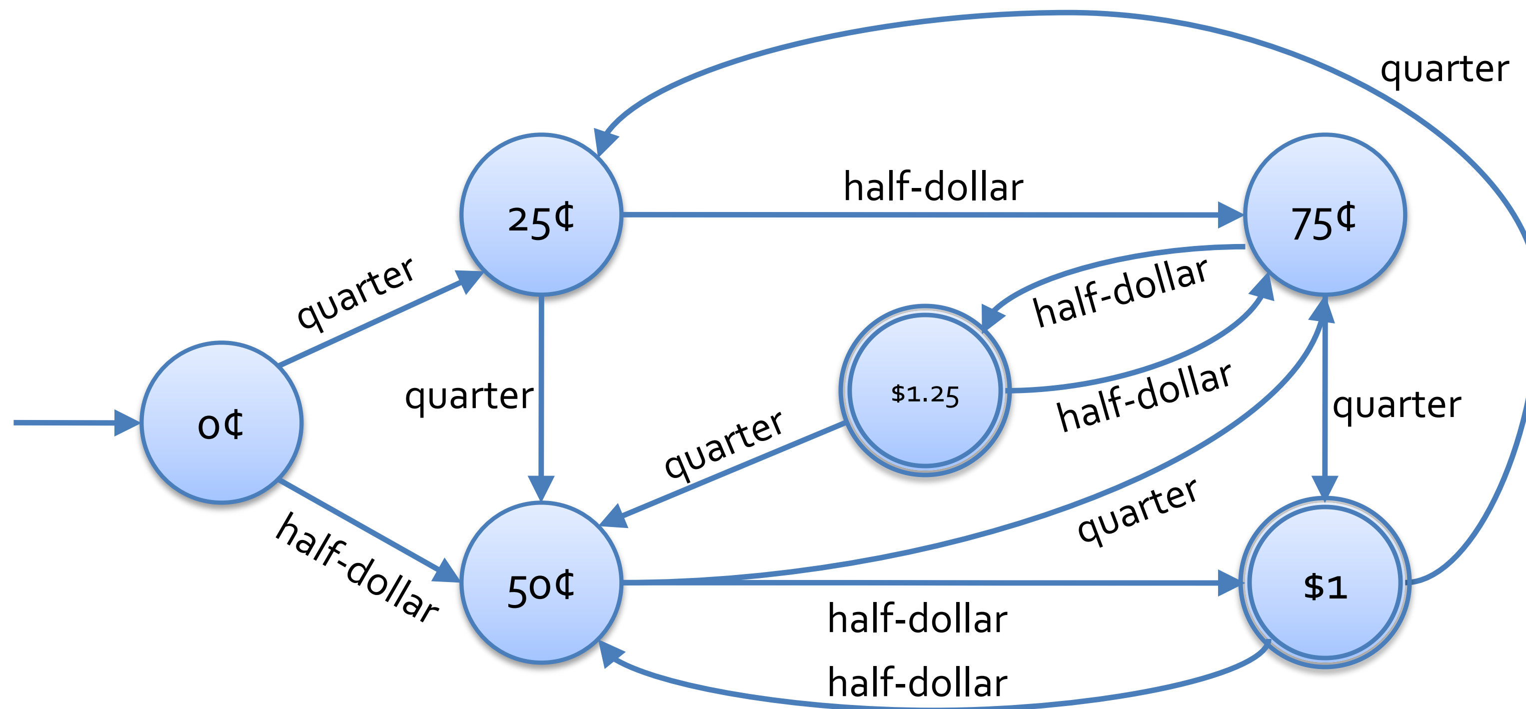
- For a finite alphabet Σ, the language $L(r)$ defined by a regular expression r is as follows
- **Base**: $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$ for every $a \in \Sigma$
- **Recursion**: If $L(r)$ and $L(r')$ are the languages defined by the regular expressions r and r' over Σ, then
  - $L(r\,r') = L(r)L(r')$
  - $L(r \mid r') = L(r) \cup L(r')$
  - $L(r^*) = \left(L(r)\right)^*$

Finite-state automaton

- A finite-state automaton is an idealized machine composed of five objects:
  1. A finite set I, called the **input alphabet**, of input symbols
  2. A set S of **states** the automaton can be in
  3. A designated state s_0 called the **initial state**
  4. A designed set of states called the set of **accepting states**
  5. A **next-state function** N: $S \times I \rightarrow S$ that maps a current state with current input to the next state

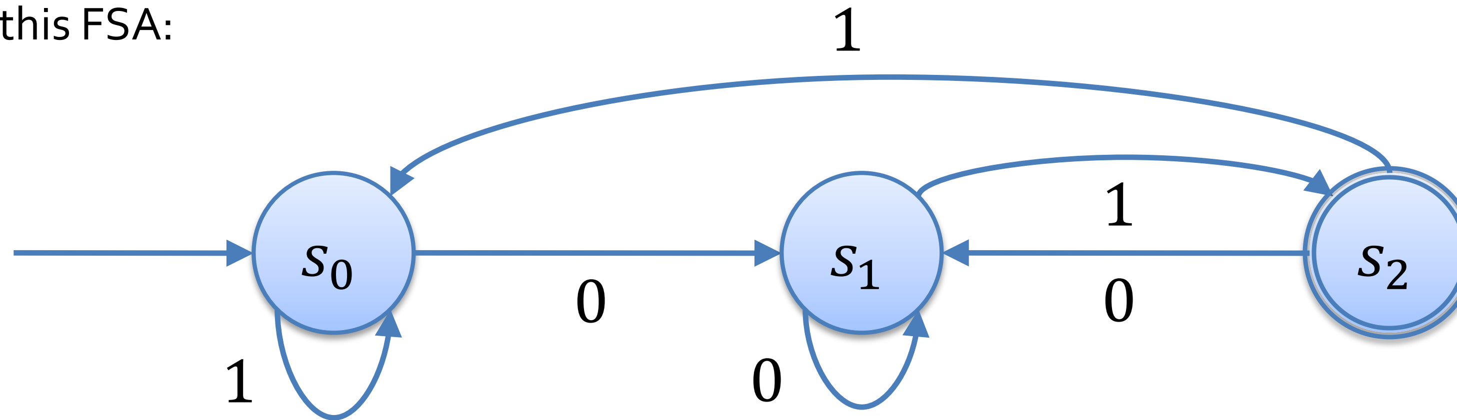
Transition diagram

- FSA's are often described with a state transition diagram
  - The starting state has an arrow
  - The accepting states are marked with circles
  - Each rule is represented by a labeled transition arrow
- The following FSA represents a vending machine



Annotated next-state tables

- Consider this FSA:



- We can also describe an FSA using an **annotated next-state table**
- A next-state table gives shows what the transition is for each state for all possible input
- An annotated next-state table also marks the initial state and accepting states

		0	1
→	s_0	s_1	s_0
	s_1	s_1	s_2
•	s_2	s_1	s_0

*-equivalence

- Two states of a finite-state automaton are $*$**-equivalent** if any string accepted by the automaton when it starts from one state is accepted when starting from the other
- Given an automaton A with eventual-state function N^*, we can formally say:
  - States s and t in A are $*$-equivalent iff $N^*(s, w)$ and $N^*(t, w)$ are both accepting states or both not
- It turns out that $*$-equivalence defines an equivalence relation

k-equivalence

- $*$-equivalence is hard to demonstrate directly
- Instead, we'll focus on equivalence after k or fewer inputs
- Given an automaton A with eventual-state function N^*, we can formally say:
  - States s and t in A are k-equivalent iff $N^*(s, w)$ and $N^*(t, w)$ are both accepting states or both not, for all strings w of length k or less

Finding the $*$-equivalence classes

- Keep finding k-equivalence classes for larger and larger values of k
- If you ever find that the set of k-equivalence classes is equal to the set of $(k + 1)$-equivalence classes, that is the set of $*$-equivalence classes
- This is known as a **fixed point** in mathematics
- Using the $*$-equivalence classes, we can build the **quotient automaton** that is the smallest FSA that accepts a given language
- Two equivalent FSA's must have the same quotient automaton (except for labels)

Context free languages

- A context free language is one that can be described by a context free grammar
- Every regular language is context free, but there are context free languages that are not regular
- Classic examples:
  - Strings of k 0's followed by k 1's
  - Palindromes made up of a's and b's
  - Legally nested parentheses
- All of these involve counting arbitrary numbers of characters
  - Regular expressions can't count

Context free grammars

- Instead of using regular expressions, a context free language is often described with a **grammar**
- A grammar is a formal system of rewriting rules consisting of
  - Terminals: symbols of the alphabet
  - Non-terminals: symbols that produce other sequences of terminals and non-terminals
- Grammars often start with the non-terminal starting symbol S
- Any string that can be derived from S through some sequence of rule rewrites is a string in the language

Chomsky hierarchy

- A grammar consists of terminals (alphabet symbols), non-terminals, production rules, and a start symbol
- Chomsky noted that grammars can be divided into four levels in terms of expressiveness:
  - Type-0 (Unrestricted grammars)
  - Type-1 (Context sensitive grammars)
  - Type-2 (Context free grammars)
  - Type-3 (Regular grammars)

Rules for grammars

- Each grammar has rules for what is a legal production rule
- Let α, β, and γ be any combinations of terminals and non-terminals, where γ is non-empty

1. Unrestricted grammars
   - $\alpha \rightarrow \beta$ (anything to anything)
2. Context-sensitive grammars
   - $\alpha A \beta \rightarrow \alpha \gamma \beta$ (non-terminal in a particular context to anything)
3. Context-free grammars
   - $A \rightarrow \gamma$ (a single non-terminal to anything)
4. Regular grammars
   - $A \rightarrow a$ and $A \rightarrow aB$ (a single non-terminal to a single terminal or a terminal and a single non-terminal on the right side)

Languages broken down

- Every kind of language has a particular kind of machine associated with it

Grammar	Languages	Automaton	Production Rules	Examples
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$	All languages, any computable functions
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	Most natural human languages
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$	Most programming languages
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$	Regular expressions

Formal language practice

- Write a regular expression that accepts strings that contain an even numbers of b's, an odd numbers of a's, and at least one c
- Draw an FSA that accepts this language
- Create a context-free grammar that defines this language

Upcoming

Next time…

- There is no next time!

Reminders

- Finish Assignment 6
  - **Due tonight!**
- **Final exam:**
  - **Wednesday, April 29, 2026**
  - **8:00 to 10:00 a.m.**