Introduction to Computer Science

CSCI 109

Readings

St. Amant, 1-4, 8

Andrew Goodney

Fall 2019

Lecture 6: First Half Review

October 6th, 2019

Where are we?

Date	Topic		Assigned	Due	Quizzes/Midterm/Final	Slide Deck
26-Aug	Introduction	What is computing, how did computers come to be?				1
2-Sep	Labor day					
9-Sep	Computer architecture	How is a modern computer built? Basic architecture and assembly	HW1			2
16-Sep	Data structures	Why organize data? Basic structures for organizing data			**Quiz 1** on material taught in class 8/26 and 9/9	3
23-Sep	Data structures	Trees, Graphs and Traversals	HW2	HW1		4
30-Sep	More Algorithms/Data Structures	Recursion and run-time				5
7-Oct	Complexity and combinatorics	How "long" does it take to run an algorithm. P vs NP			**Quiz 2** on material taught in class 9/16 and 9/23	5
14-Oct	Algorithms and programming	Programming, languages and compilers		HW2	**Quiz 3** on material taught in class 9/30	7
21-Oct	Operating systems	What is an OS? Why do you need one?	HW3		**Quiz 4** on material taught in class 10/7	8
28-Oct	Midterm	Midterm			**Midterm** on all material taught so far.	
4-Nov	Computer networks	How are networks organized? How is the Internet organized?		HW3		9
11-Nov	Artificial intelligence	What is AI? Search, plannning and a quick introduction to machine learning			**Quiz 5** on material taught in class 9/4	10
18-Nov	The limits of computation	What can (and can't) be computed?	HW4		**Quiz 6** on material taught in class 11/11	11
25-Nov	Robotics	Robotics: background and modern systems (e.g., self-driving cars)			**Quiz 7** on material taught in class 11/18	12
2-Dec	Summary, recap, review	Summary, recap, review for final		HW4	**Quiz 8** on material taught in class 11/25	13
13-Dec	Final exam 11 am - 1 pm in SGM 123				**Final** on all material covered in the semester	

Review

- Last time we got a little ahead
- So we’ll review the first half of the semester

Lecture #1

Lecture #1

Computational Thinking

- “thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent” (Cuny, Snyder, Wing)
  - way of solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science
    - To flourish in today's world, computational thinking has to be a fundamental part of the way people think and understand the world
  - creating and making use of different levels of **abstraction**, to understand and solve problems more effectively
  - thinking **algorithmically** and with the ability to apply mathematical concepts such as **induction** to develop more efficient, fair, and secure solutions
  - understanding the consequences of **scale**, not only for reasons of efficiency but also for economic and social reasons

Humans thinking (i.e., transforming information) to devise procedures for execution by information transformers (human and/or machine)

Before Mechanical Computers

Electronic computers were preceded by mechanical computers and mechanical computers were preceded by…

Discrete Machines: State

- How does the loom behave as a function of time?
- At any given time a set of threads is raised and the rest are lowered
- Writing down the sequence of raised (and lowered) threads tells us the steps the machine went through to produce the cloth/tapestry/whatever
- The pattern of raised (and lowered) threads is called the *state* of the machine

CS Topic: State

- State is a very common CS concept
- Here we have the state of a physical machine
- In CS we talk about the "state" of an object
  - Of a database
  - Of a robot
  - Of a "state-machine" (finite, Turing, etc…)
  - Of a system (physical or virtual)
  - …
- Then we need a way to describe the state
  - Gives us the notion of an encoding

CS Topic: Discrete Machines, State and Encoding

- Choosing a state representation takes skill. The state should be
  - Parsimonious: it should be a “small” descriptor of what the machine is doing at any given time
  - Adequate: it should be “big enough” to capture everything “interesting” about the machine
- These are sometimes contradictory. They are also qualitative and depend on what behavior of the machine we want to describe
- Usually you need a vocabulary (*encoding*) to describe state. In the case of a loom, state can be expressed as a binary pattern (1 for raised, 0 for lowered)

Discrete Machines: Abstraction

- The loom is a *discrete machine*
  - *State is binary pattern – i.e. discrete*
  - *The notion of time is discrete – i.e. time is modeled as proceeding in steps or finite chunks*
- More precisely, *the loom can be usefully modeled as a discrete machine*
  - *Because of course being a physical device there is variation, nothing is exactly precise*
  - *But modeling the machine as discreet is good enough and works for this purpose*
- This is an example of an *abstraction* – a key concept in Computer Science

CS topic: Abstraction

- One of the fundamental “things” we do in CS
- Reducing or distilling a problem or concept to the essential qualities
  - Simple set of characteristics that are most relevant to the problem
- Many (most, all) of what we do in engineering and computer science involves abstractions
- Here the abstraction is modelling the loom as a simple discreet state machine
  - Makes it possible to understand
  - And makes it possible to “program” the loom

Lecture #1

- What makes a computer?
  - Lots of things can help us compute (information transformation)
  - Computers need
    - Memory
    - Control-flow
- State
- Abstraction

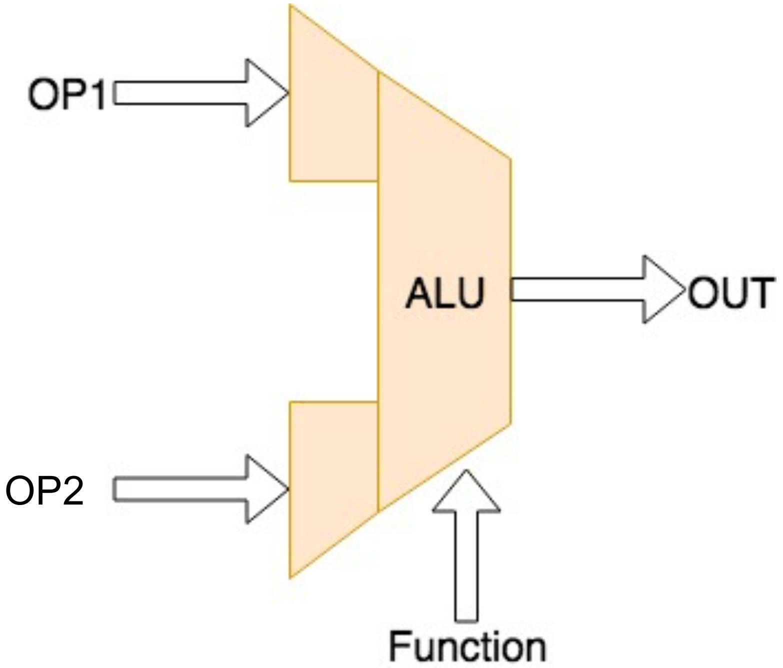
Lecture #2

Motivation

- What do computers do?
  - Math with binary numbers
- So what do we need to build a computer?
  - Place to store binary numbers
  - Way to do math

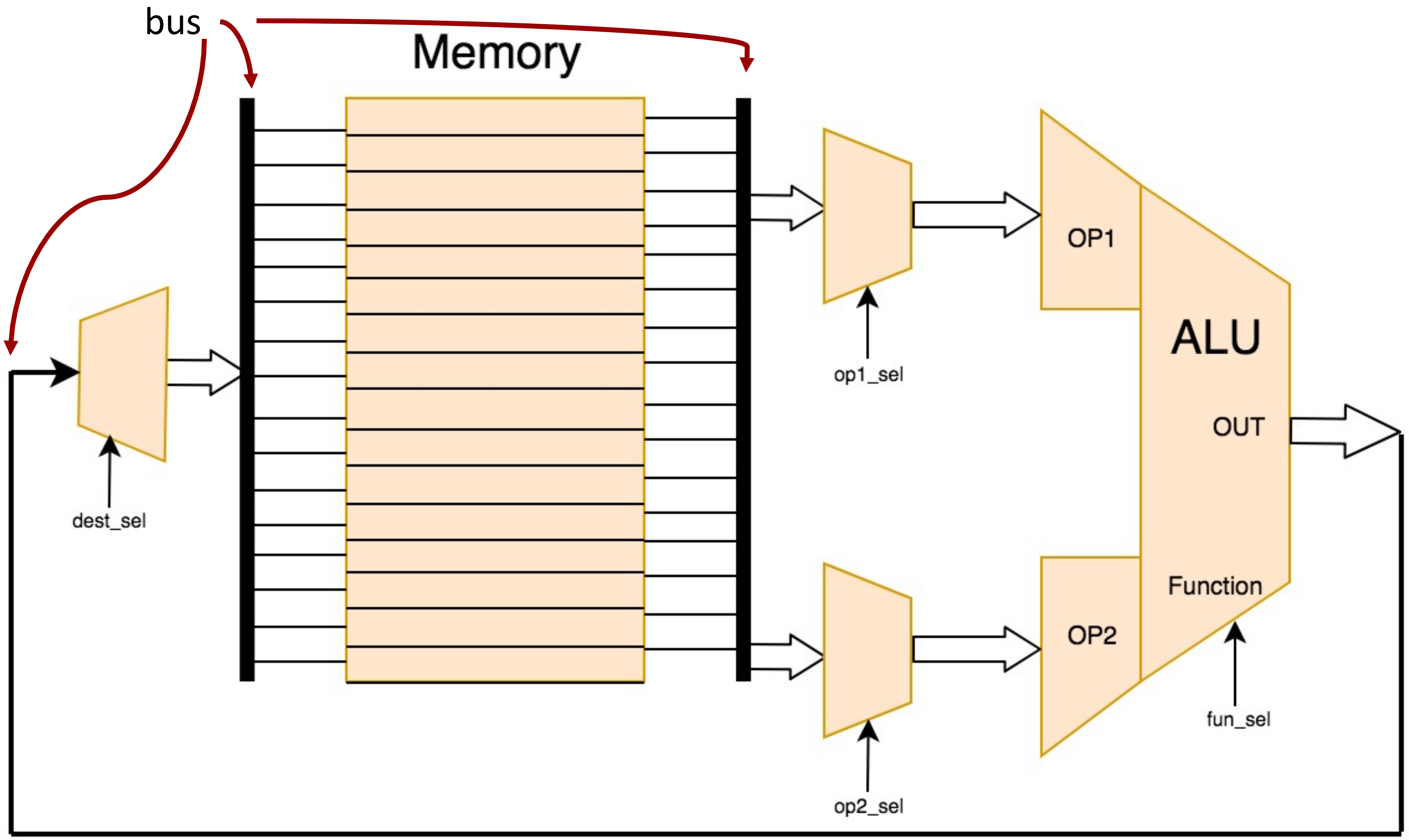
Arithmetic/Logic

- "math" we need to do with numbers in memory
  - ADD
  - SUBTRACT
  - MULTIPLY
  - DIVIDE
  - AND,OR,XOR,NOT
  - Etc…



- Assume we can build a circuit that can do this
- Takes numbers represented as digital (electrical) values, produces results as the same

Start building a circuit…



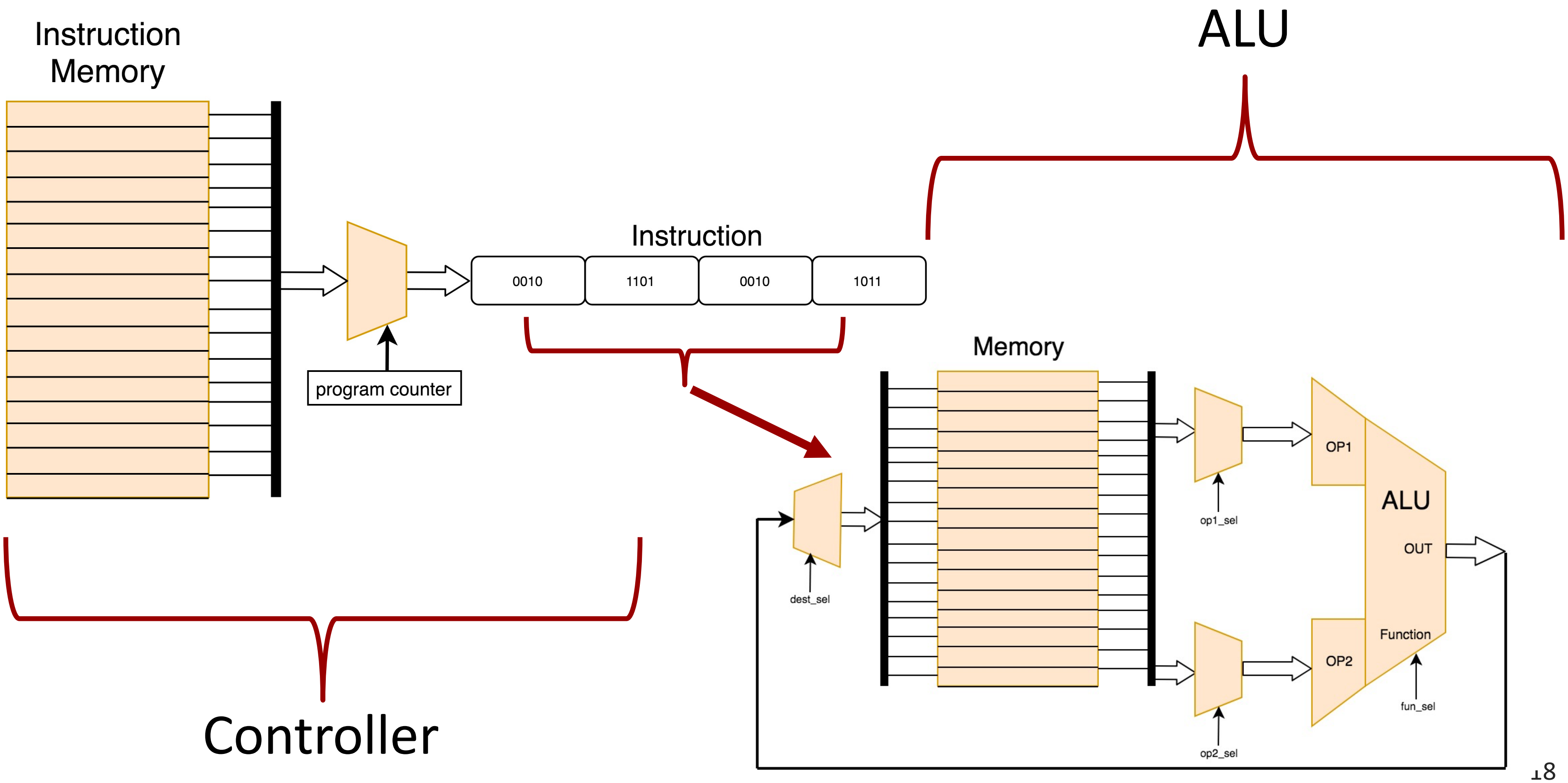
Instructing the CPU

- Now we can make instructions…
- Instructions are binary numbers that tell the circuit what to do
- Select the 1st operand, 2nd operand, destination and function
- With a series of such instructions the circuit can perform arbitrary computations

0010	1101	0010	1011
op1_sel	op2_sel	dest_sel	fun_sel

Where to get the instructions?

- Instruction Memory

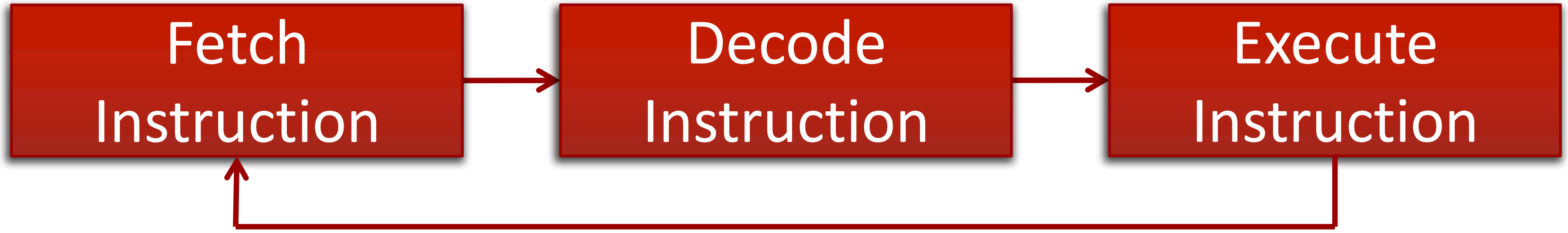


How to compute?

- Fill instruction memory with desired program
- Initialize data memory
- Run an instruction (given by program counter)
  - Then increment program counter
  - Run next instruction, increment program counter…
- Some early computers were pretty much just this

The Central Processing Unit (CPU)

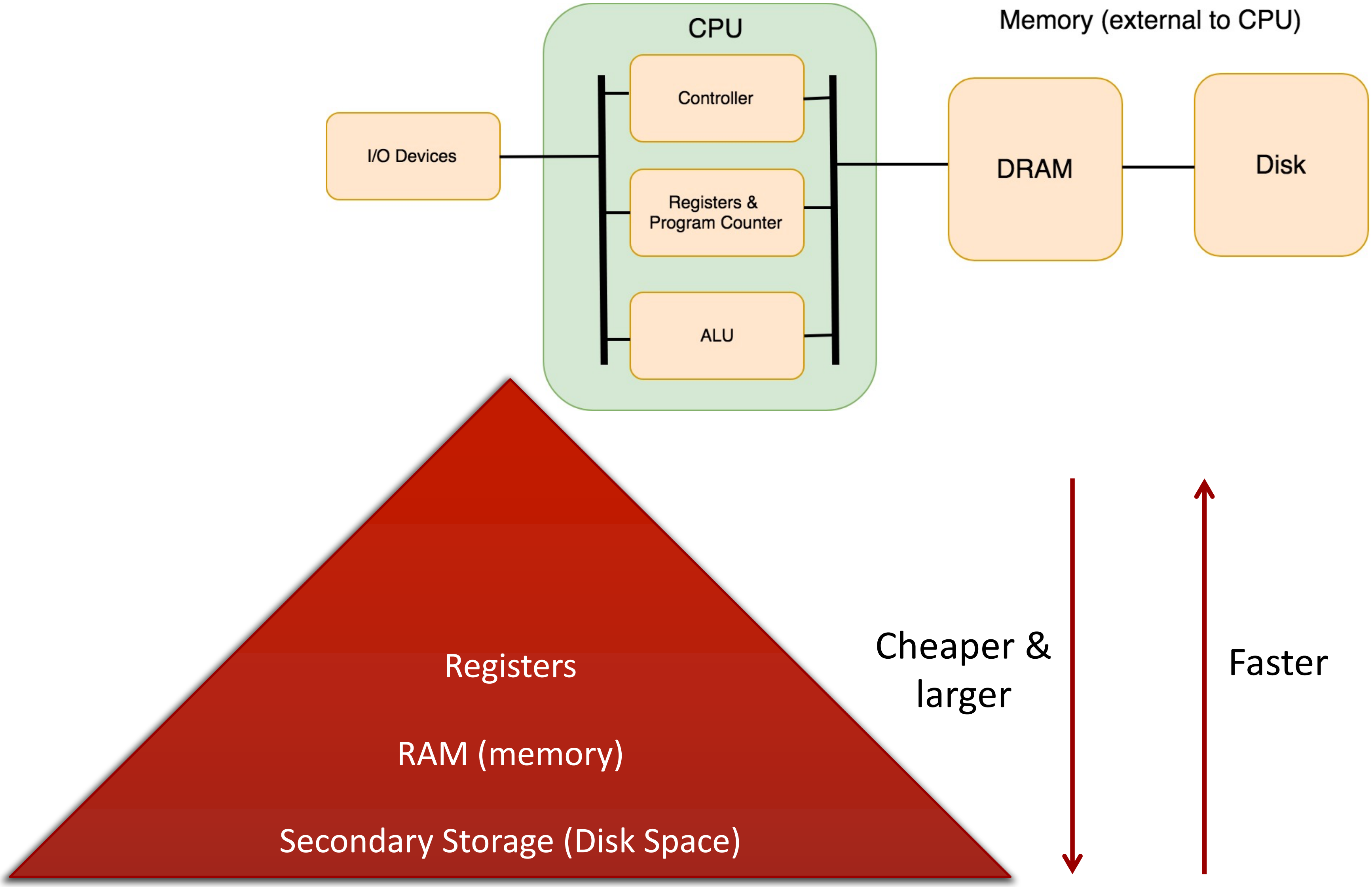
- Controller + ALU = Central Processing Unit (CPU)
- CPU has a small amount of temporary memory within it
  - Registers
  - A special register called the program counter (PC)
- CPU performs the following cycle repeatedly



Fetch-Decode-Excecute

- Fetch
  - Get the next instruction from memory
- Decode
  - Send the proper signals from the controller to the ALU and Registers
- Execute
  - Let the ALU do its work to produce a result

The Storage Hierarchy



CPU

Controller

Registers & Program Counter

ALU

I/O Devices

Memory (external to CPU)

DRAM

Disk

Registers

RAM (memory)

Secondary Storage (Disk Space)

Cheaper & larger

Faster

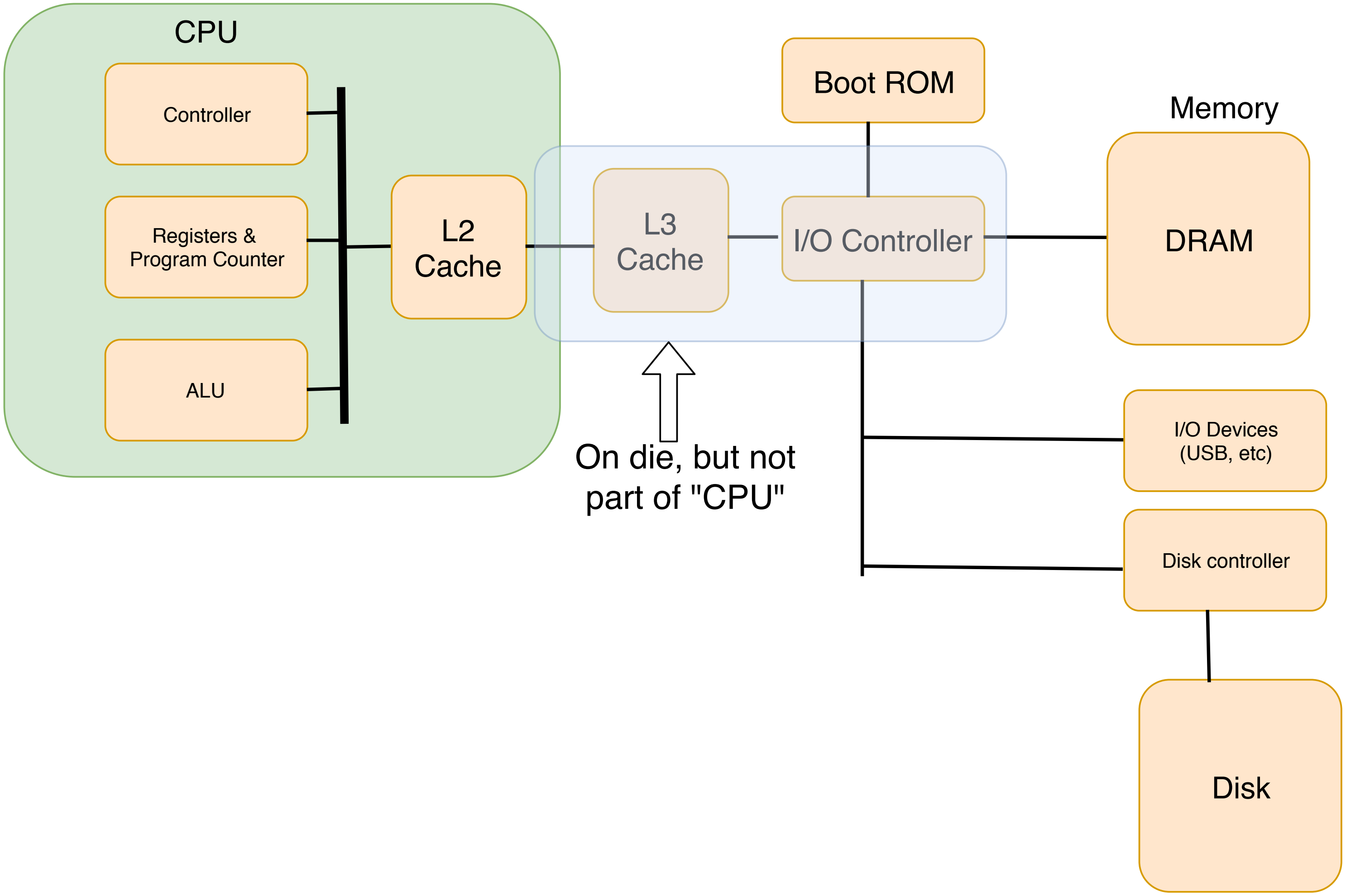
Trade-offs

- An aside…
- Identifying trade-offs is a fundamental engineering skill
- Understanding and balancing trade-offs is part of design process
- Not always easy to manage!
- Conflicting interests
- Speed vs. space is very common tradeoff in CS
  - So if you want faster execution you need more memory

Cache

- Small (but bigger than registers)
- Volatile
- Fast (not as fast as registers, but faster than RAM)
- What to keep in the cache ?
  - Things that programs are likely to need in the future
  - Locality principle:
    - Look at what items in memory are being used
    - Keep items from nearby locations (spatial locality)
    - Keep items that were recently used (temporal locality)

Modern Computer Architecture diagram



Programming a CPU

- How to compute?
- Develop a series of low-level instructions
  - Using the registers and/or main memory for storage
  - Using only low-level operations made available by the particular CPU
- ”Assembly language”
  - Or maybe even machine code (probably not, though)

Typical Operations

ADD Ri Rj Rk	Add contents of registers Ri and Rj and put result in register Rk
SUBTRACT Ri Rj Rk	Subtract register Rj from register Ri and put result in register Rk
AND Ri Rj Rk	Bitwise AND contents of registers Ri, Rj and put result in register Rk
NOT Ri	Bitwise NOT the contents of register Ri
OR Ri Rj Rk	Bitwise OR the contents of registers Ri, Rj and put result in register Rk
SET Ri value	Set register Ri to given value
SHIFT-LEFT Ri	Shift bits of register Ri left
SHIFT-RIGHT Ri	Shift bits of register Ri right
MOVE Ri Rj	Copy contents from register Ri to register Rj
LOAD Mi Ri	Copy contents of memory location Mi to register Ri
WRITE Ri Mi	Copy contents of register Ri to memory location Mi
GOTO Mi	Jump to instruction stored in memory location Mi
COND_GOTO Ri Rj Mi	If Ri > Rj, jump to instruction stored in memory location Mi

Lecture #2 Summary

- Computers do two things:
  - Binary Math
  - Move data
- So we build a state machine (CPU):
  - Controller, Registers, ALU
  - Fetch-Decode-Execute Cycle
- Memory Hierarchy and Caching
- Assembly Language Programming

Lecture #3

- “The architecture level gives us a very detailed view of what happens on a computer. But trying to understand everything a computer does at this level would be…(insert analogy about perspective). If all we can see is fine detail, it can be hard to grasp what’s happening on a larger scale.”

Problem Solving

- Architecture puts the computer under the microscope
  - Imagine solving *all* problems by thinking about the computer at the architecture level
- Early computer scientists *had* to do this
  - Luckily we don’t.

Problem Solving

- Computers are used to solve problems
- Abstraction for problems
  - How to represent a problem ?
  - How to break down a problem into smaller parts ?
  - What does a solution look like ?
- Two key building blocks
  - Abstract data types
  - Algorithms

Abstract Data Types

- Models of collections of information
  - Chosen to help solve a problem
- Typically at an abstract level
  - Don’t deal with implementation details: memory layout, pointers, etc.

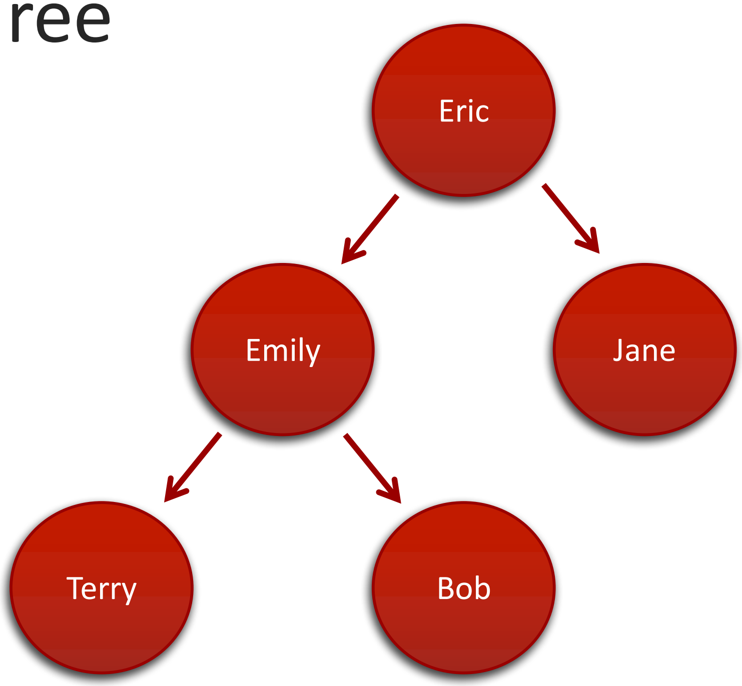
“… describes what can be done with a collection of information, without going down to the level of computer storage.” [St. Amant, pp. 53]

Motivation for Abstract Data Structures

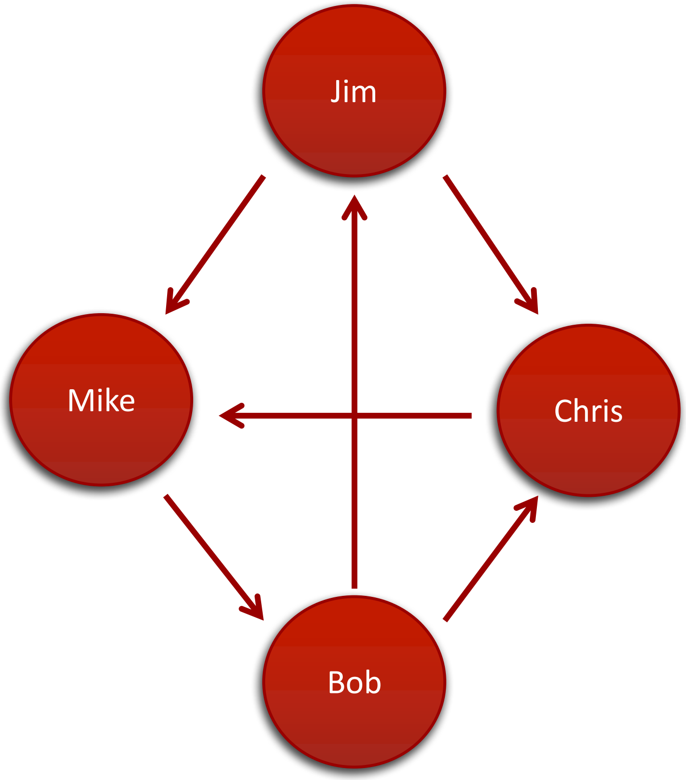
- The nature of some data, and the way we need to accesses it often requires some structure, or organization to make things efficient (or even possible)
- Data: large set of names (maybe attendance data)
- Problems: did Jelena attend on 9/9? How many lectures did Mario attend? Which students didn’t attend 8/26?

Sequences, Trees and Graphs

- Sequence: a list
  - Items are called elements
  - Item number is called the index
- Tree



- Graph



Sequences aka Lists

- Sequences are our first fundamental data structure
- Sequences hold items
  - Items = what ever we need. It’s abstract.
- Sequences have the notion of order
  - Items come one after another
- Sequences can be accessed by index, or relative
  - Find the 5th item
  - Or move to next or previous from current item
- The “how” (implementation) is not important (now)
  - Arrays (C, C++), Vectors (C++), ArrayList (Java), Lists (Python)…
  - These are all different implementations of this abstract data structure

Sequence Tasks

- Most “questions” (problems) that are solved using sequences are essentially one of two questions:
- Is item A in sequence X?
- Where in sequence Y is item B?
- Both of these are answered by searching the sequence

Sequences: Searching

- Sequential search: start at 1, proceed to next location…

brute force

- If names in the list are *sorted* (say in alphabetical order), then how to proceed?
  - Start in the ‘middle’
  - Decide if the name you’re looking for is in the first half or second
  - ‘Zoom in’ to the correct half
  - Start in the ‘middle’
  - Decide if the name you’re looking for is in the first half or second
  - ‘Zoom in’ to the correct half
  - …

divide-and-conquer

- Which is more efficient (under what conditions)?

Sorting

- If searching a sorted sequence is more efficient (per search), this implies we need a way to sort a sequence!
- Sorting algorithms are fundamental to CS
  - Used A LOT to teach various CS and programming concepts
- Computer Scientists like coming up with better more efficient ways to sort data
  - Even have contests!
- We'll look at two algorithms with very different designs
  - Selection Sort
  - Quick Sort

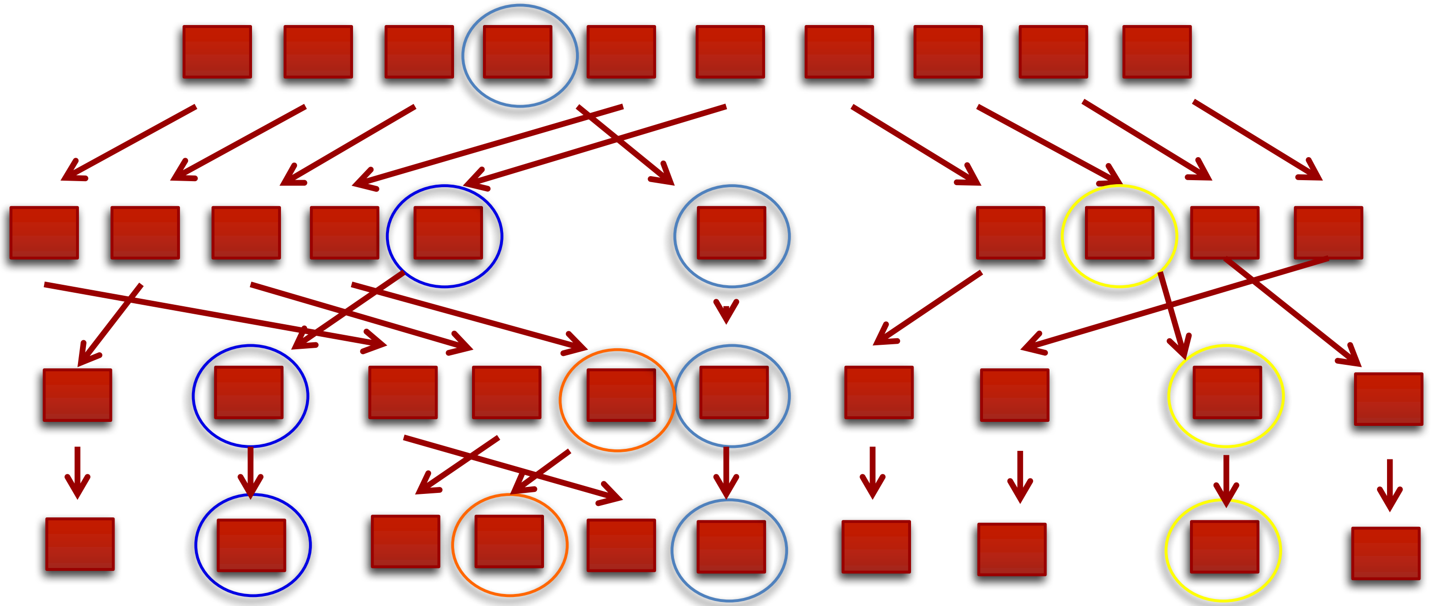
Sorting: Selection Sort

- Sorting: putting a set of items in order
- Simplest way: selection sort
  - March down the list starting at the beginning and find the smallest number
  - Exchange the smallest number with the number at location 1
  - March down the list starting at the second location and find the smallest number (overall second-smallest number)
  - Exchange the smallest number with the number at location 2
  - …

Sorting: Quicksort

- Pick a ‘middle’ element in the sequence (this is called the pivot)
- Put all elements smaller than the pivot on its left
- Put all elements larger than the pivot on the right
- Now you have two smaller sorting problems because you have an unsorted list to the left of the pivot and an unsorted list to the right of the pivot
- Sort the sequence on the left (use Quicksort!)
  - Pick a ‘middle’ element in the sequence (this is called the pivot)
  - Put all elements smaller than the pivot on its left
  - Put all elements larger than the pivot on the right
  - Now you have two smaller sorting problems because you have an unsorted list to the left of the pivot and an unsorted list to the right of the pivot
  - Sort the sequence on the left (use Quicksort!)
  - Sort the sequence on the right (use Quicksort!)
- Sort the sequence on the right (use Quicksort!)
  - Pick a ‘middle’ element in the sequence (this is called the pivot)
  - Put all elements smaller than the pivot on its left
  - Put all elements larger than the pivot on the right
  - Now you have two smaller sorting problems because you have an unsorted list to the left of the pivot and an unsorted list to the right of the pivot
  - Sort the sequence on the left (use Quicksort!)
  - Sort the sequence on the right (use Quicksort!)

Quicksort



Lecture #3 Summary

- Solving a problem with a computer usually involves:
  - A structured way to store (organize) data
  - An algorithm that accesses and modifies that data
- Algorithms have characteristics, like *brute-force* or *divide-and-conquer* that help us understand how they work
- Thinking about abstract data types and algorithms frees us from worrying about the implementation details
- Sequences are a fundamental ADT used to organize data in an ordered list.
- Sequences can be searched:
  - Linear search (brute-force)
  - Binary search (divide-and-conquer), but requires *sorted list*
- Sequences can be sorted:
  - Selection sort (brute-force)
  - Quick-sort (divide-and-conquer

Lecture #4

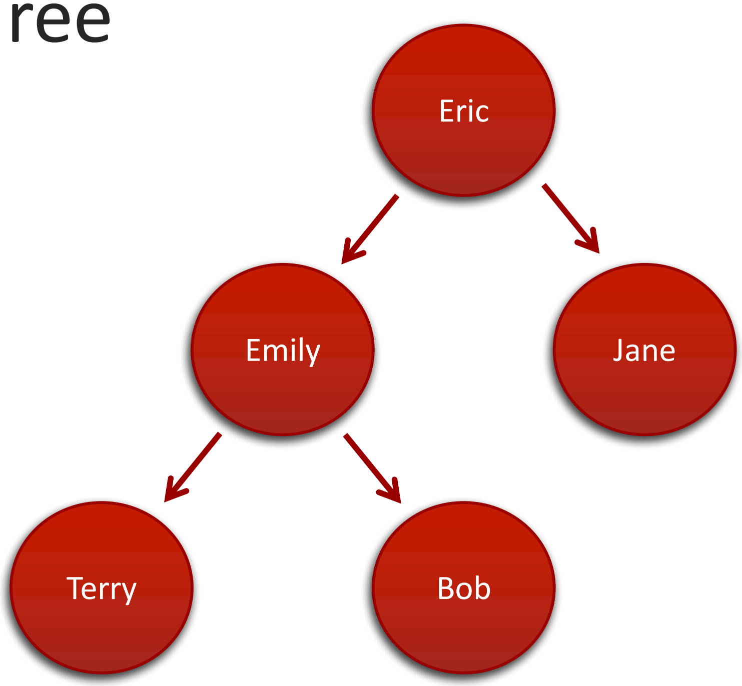
Abstract Data Types

- Models of collections of information
- Typically at an abstract level

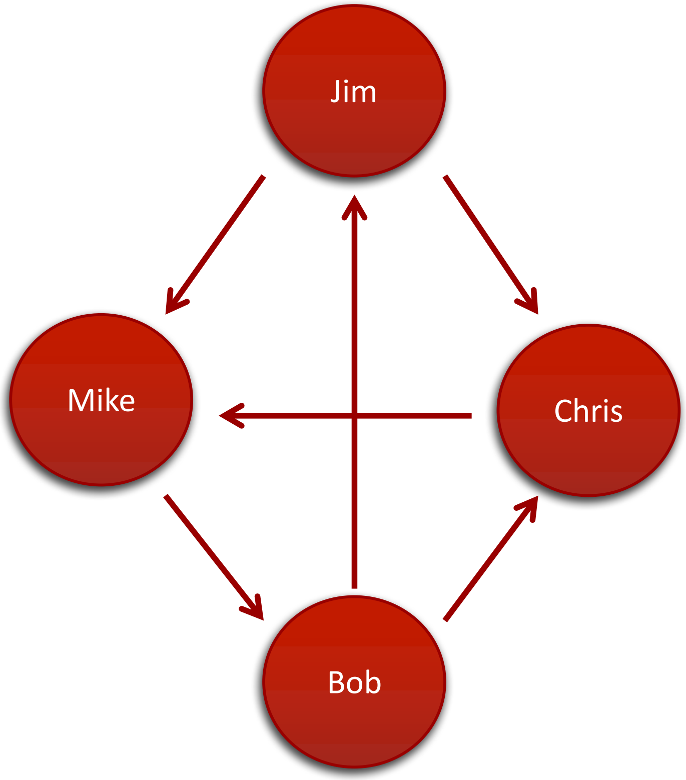
“… describes what can be done with a collection of information, without going down to the level of computer storage.” [St. Amant, pp. 53]

Sequences, Trees and Graphs

- Sequence: a list
  - Items are called elements
  - Item number is called the index
- Tree



- Graph



Motivation for Abstract Data Structures (Graphs, Trees)

- The nature of some data, and the way we need to accesses it often requires some structure, or organization to make things efficient (or even possible)
- Data: large set of people and their family relationship used for genetic research
- Problems: two people share a rare genetic trait, how closely are the related? (motivates for a tree)

Motivation for Abstract Data Structures (Graphs, Trees)

- Data set: roads and intersections.
- Problem: how to travel from A to B @5pm on a Friday? How to avoid traffic vs. prefer freeways? (motivates a weighted graph)
- Data set: freight enters country at big port (LA/Long Beach).
- Problem: How to route freight given train lines/connections?
  - Route fastest, vs. lowest cost?
- Data set: airport locations
- Problem: how to route and deliver a package to any address in the US with minimum cost? Think UPS, FedEx

Motivation for Abstract Data Structures (Graphs, Trees)

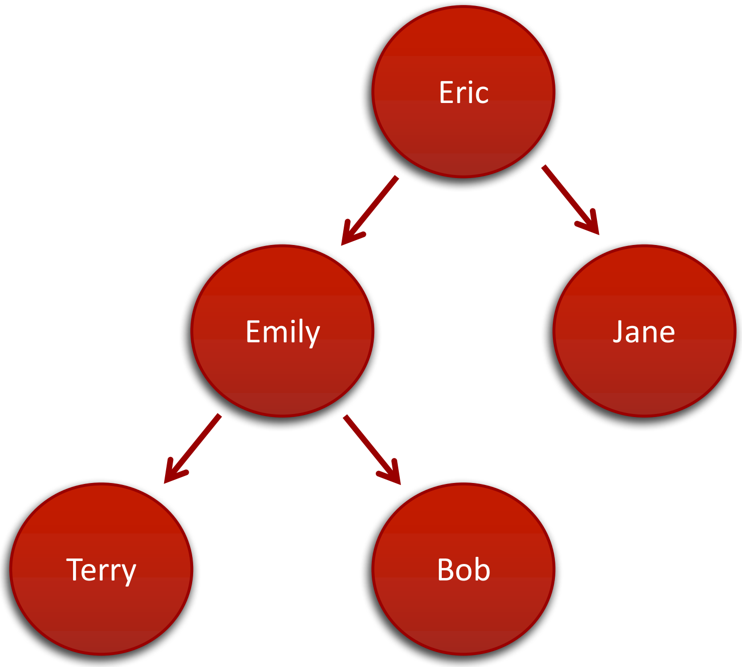
- Data set: network switches and their connectivity (network links)
- Problem: Chose a subset of network links that connect all switches without loops (networks don’t like loops). Motivates graphs, and graph -> tree algorithm

Motivation for Abstract Data Structures (Graphs, Trees)

- Data set: potential solutions to a big problem
- Problem: how to find an optimal solution to the problem, without searching every possibility (solution space too big). Motivates graphs and graph search to solve problems.
- Other data/problems that motivate graphs/trees:
  - Financial networks and money flows, social networks, rendering HTML code, compilers, 3D graphics and game engines… and more

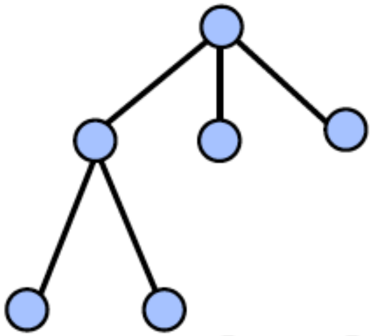
Trees

- Each node/vertex has exactly one parent node/vertex
- No loops
- Directed (links/edges point in a particular direction)
- Undirected (links/edges don't have a direction)
- Weighted (links/edges have weights)
- Unweighted (links/edges don't have weights)

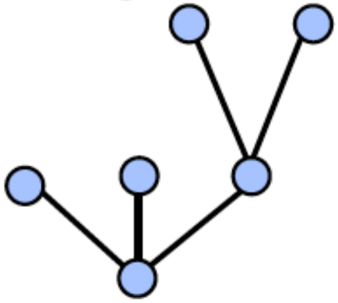


Which of these are NOT trees?

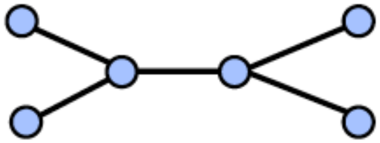
1



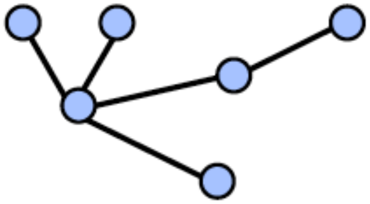
2



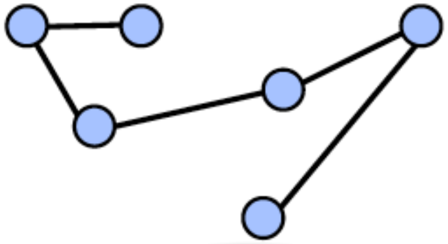
3



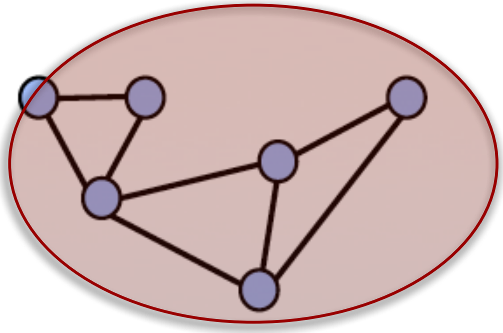
4



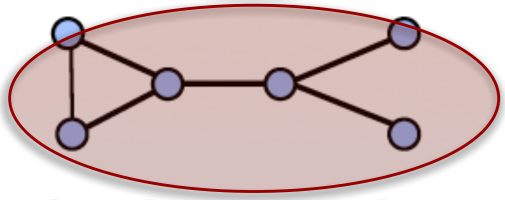
5



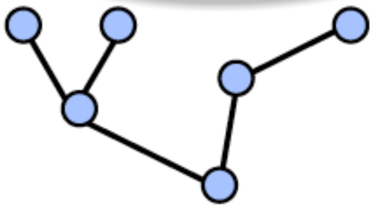
6



7



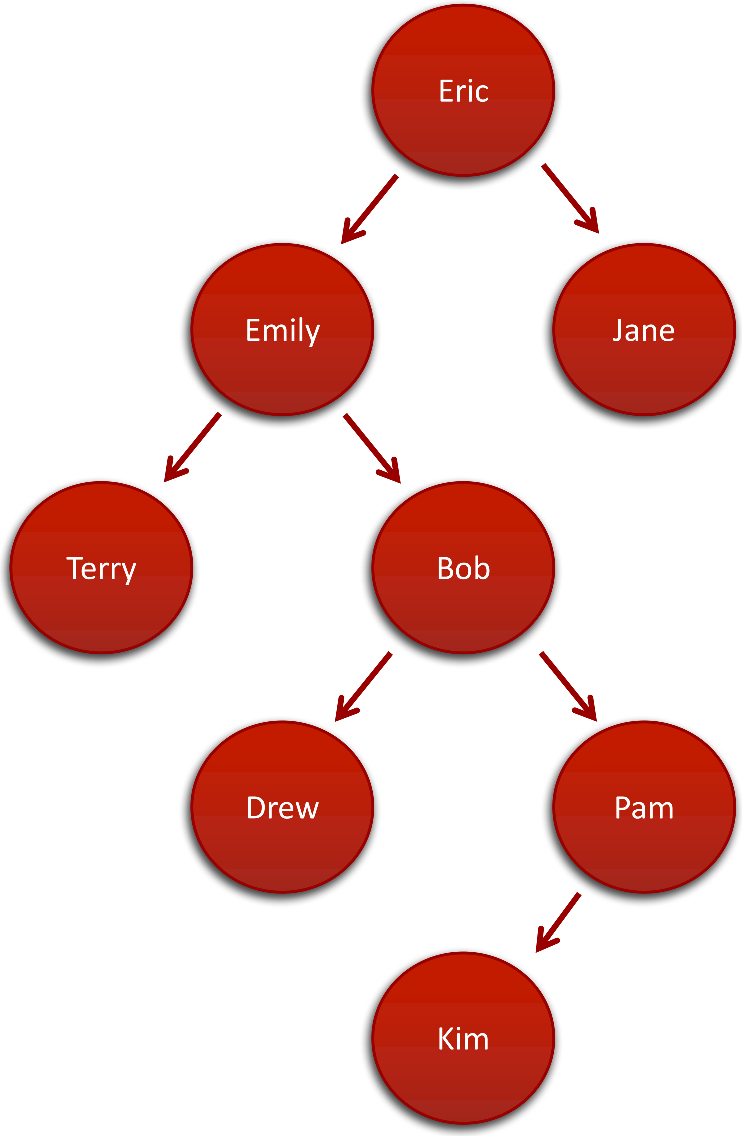
8



Graph/Tree Traversal

- Traversing a graph or a tree: “moving” and examining the nodes to enumerate the nodes or look for solutions
- Example: find all living descendants of X in our genetic database.
- For traversing a graph we pick a starting node, then two methods are obvious:
  - Depth first
    - Go as deep (far away from starting node) as possible before backtracking
  - Breadth first
    - Examine one layer at a time

Tree Traversal



- Depth first traversal

  Eric, Emily, Terry, Bob, Drew, Pam, Kim, Jane

- Breadth first traversal

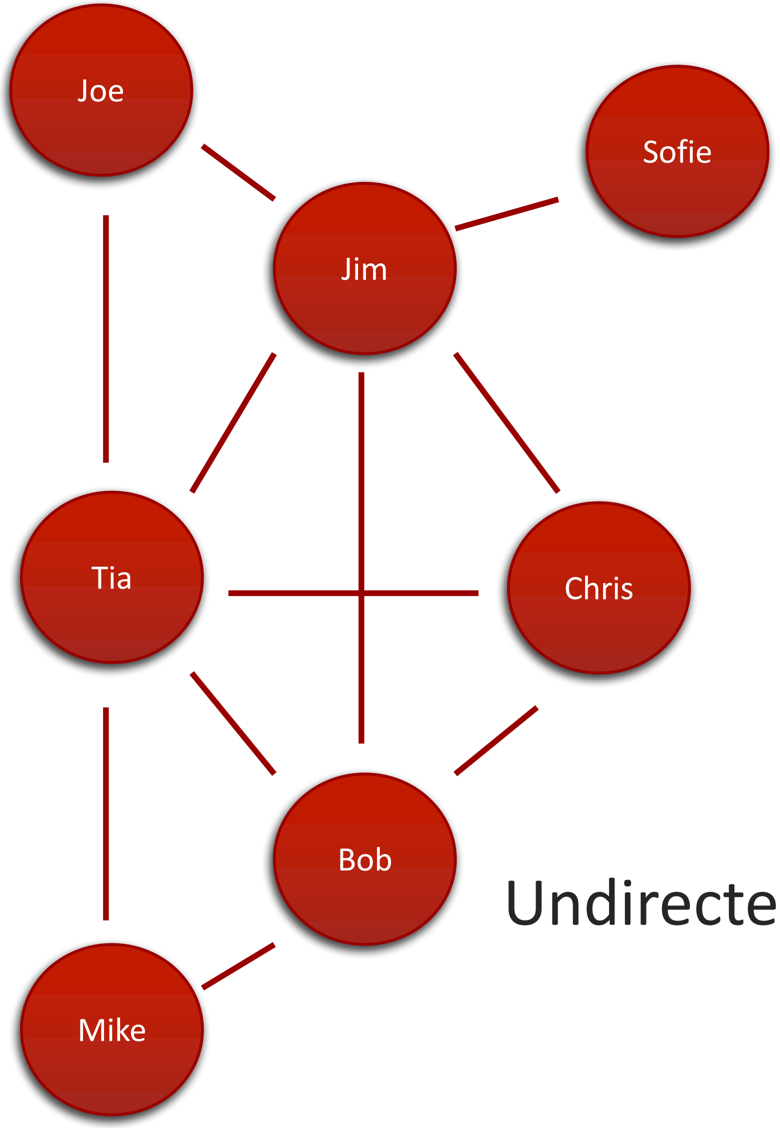
  Eric, Emily, Jane, Terry, Bob, Drew, Pam, Kim

  Eric, Jane, Emily, Bob, Terry, Pam, Drew, Kim

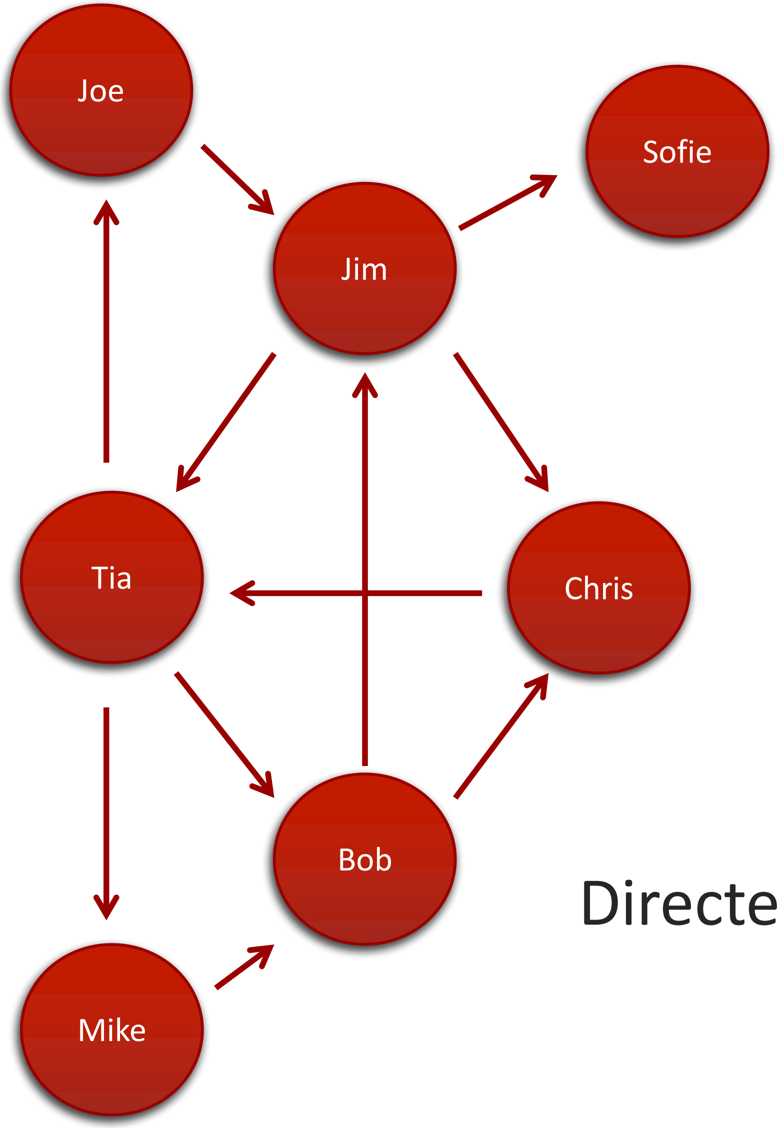
Tree Traversal

- Depth first vs. Breadth first eventually visit all nodes, but do so in a different order
- Used to answer different questions
  - Depth first: good for game trees, evaluating down a certain path
  - Breadth first: look for shortest path between two nodes (e.g for computer networks)
- Roughly:
  - Depth first: find ‘a’ solution to the problem
  - Breadth first: find ‘the’ solution to the problem

Graphs: Directed and Undirected



Undirected



Directed

Graph to Tree Conversion Algorithms

- Sometimes the question is best answered by a tree, but we have a graph
- Need to convert graph to tree (by deleting edges)
- Usually want to create a “spanning tree”

Spanning Trees

- Spanning tree: Any tree that covers all vertices
  - “Cover” = “include” in graph-speak
- Example: graph of social network connections. Want to create a “phone tree” to disseminate information in the event of an emergency
- Example: network of switches with redundant links and multiple paths between switches (there are loops aka cycles in the graph). Need to chose a set of links that connects all switches with no loops.

Minimum Spanning trees

- Spanning tree: Any tree that covers all vertices, not as common as the MST
- *Minimum* spanning tree (MST): Tree of minimal total edge cost
- If you have a graph with weighted edges, a MST is the tree where the sum of the weights of the edges is minimum
- There is at least one MST, could be more than one
- If you have unweighted edges any spanning tree is a MST

- Why compute the minimum spanning tree?
  - Minimize the cost of connections between cities (logistics/shipping)
  - Minimize of cost of wires in a layout (printed circuit, integrated circuit design)

Computing the MST

- Two greedy algorithms to compute the MST
  - Prim's algorithm: Start with any node and greedily grow the tree from there
  - Kruskal's algorithm: Order edges in ascending order of cost. Add next edge to the tree without creating a cycle.
- 'Greedy' means solution is refined at each step using the most obvious next step, with the hope that eventual solution is globally optimal

Prim’s algorithm

- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices (i.e uncovered, or disconnected), find the minimum and add it to the tree
- Keep repeating step 2 until all vertices are added to the MST

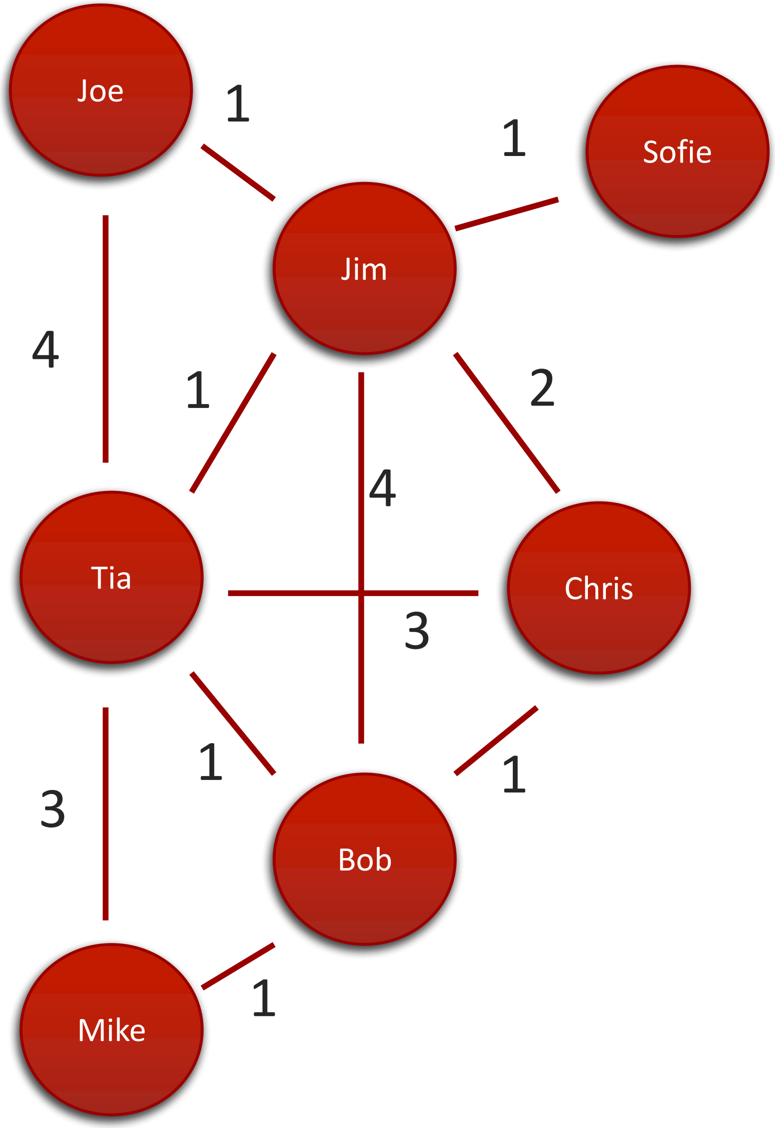
(adapted from: https://www.programiz.com/dsa)

Kruskal’s algorithm

- Sort all the edges from low weight to high
- Take the edge with the lowest weight, if adding the edge would create a cycle, then reject this edge and select the edge with the next lowest weight
- Keep adding edges until we reach all vertices.

(adapted from: https://www.programiz.com/dsa)

Shortest path



- For a given source vertex (node) in the graph, it finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex.
- Say your source vertex is Mike
  - Lowest cost path from Mike to Jim is Mike – Bob - Tia – Jim (cost 3)
  - Lowest cost path from Mike to Joe is Mike – Bob – Tia – Jim – Joe (cost 4)
- Very important for networking applications!

Dijkstra’s algorithm: Basic idea

- Fan out from the initial node
- In the beginning the distances to the neighbors of the initial node are known. All other nodes are tentatively infinite distance away.
- The algorithm improves the estimates to the other nodes step by step.
- As you fan out, perform the operation illustrated in this example: if the current node *A* is marked with a distance of 4, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 4 + 2 = 6. If B was previously marked with a distance greater than 6 then change it to 6. Otherwise, keep the current value.

Lecture 4 Summary

- Trees and Graphs
  - Sometimes need to model interactions, connections between data
  - Vertices, edges
  - Directed/undirected
  - Weighted/unweighted
- Graph Traversal
  - BFS, DFS
- Graph to Tree
  - Spanning trees, minimum spanning trees
    - Prim’s, Kruskal’s
- Shortest path: Dijkstra’s

Lecture #5

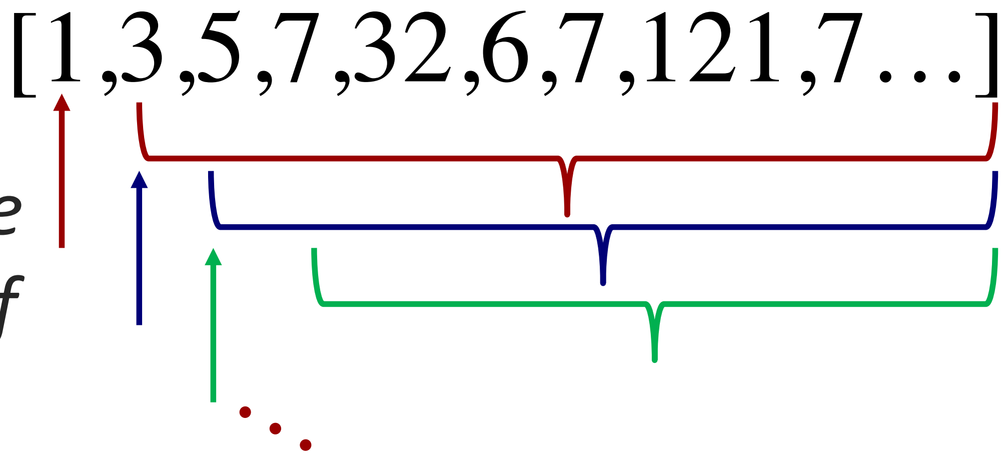
Recursion

- Recursion, recursion relations, recursive data structures, recursive algorithms
- Defining a data structure or algorithm in terms of itself
- Many problems are easier to understand (implement, solve) as recursive algorithms

Recursion: abstract data types

- Defining abstract data types in terms of themselves (e.g., trees contain trees)
- So a list is:

  The item at the front of the list, and then the rest of the list (which is, an item and then the rest of the list…)



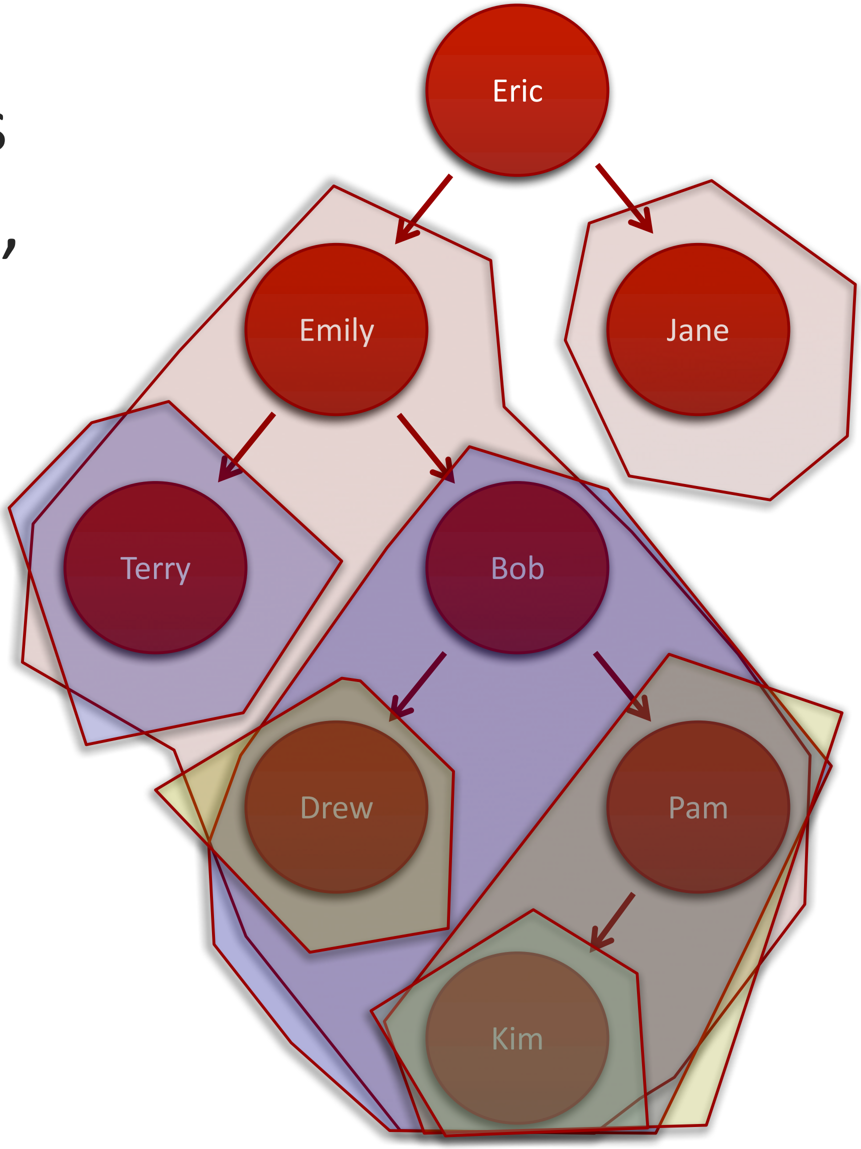
[1,3,5,7,32,6,7,121,7…]

Recursion: abstract data types

- Defining abstract data types in terms of themselves (e.g., trees contain trees)
- So a tree is

  Either a single vertex, or

  a vertex that is the parent of one or more trees



Recursion and algorithms

- Concept of recursion applies to algorithms as well
- Some algorithms are defined recursively:
  - Fibonacci numbers:
    - Fib(n) = 0 (n=0), 1 (n=1), fib(n-1) + fib(n-2)
- Some can be expressed iteratively:
  - Factorial = n*(n-1)*(n-2)*(n-3)…*1
- Or recursively:
  - Factorial = n * factorial(n-1)

Recursion and algorithms

- If an abstract data type can be thought of recursively (like a list) these often inspire recursive algorithms as well
- List sum:
  - Sum of a list = value of first item + sum of the rest of the list

Recursion: algorithms

- Defining algorithms in terms of themselves (e.g., quicksort)

  Check whether the sequence has just one element. If it does, stop

  Check whether the sequence has two elements. If it does, and they are in the right order, stop. If they are in the wrong order, swap them, stop.

  Choose a pivot element and rearrange the sequence to put lower-valued elements on one side of the pivot, higher-valued elements on the other side

  Quicksort the left sublist

  Quicksort the right sublist

Recursion: algorithms

- How do you write a selection sort recursively ?
- How do you write a breadth-first search of a tree recursively ? What about a depth-first search ?

Recursive Selection Sort

- How to do this?
- Need to think about the problem in recursive terms:
  - Think of the problem in a way that gets smaller each time you consider it…
  - Also needs to have a terminating condition (base case)
- Thinking of selection sort in this way…

Recursive selection sort

- Selection sort finds minimum element, swaps to front. Then finds next smallest, swaps to 2nd… and so on
- Observation: the front element is either:
  - Already the minimum or
  - The minimum is in the rest of the list
- Observation: once we move the minimum to the front of the list, we can call selection sort on the rest of the list

Recursive selection sort

- We actually need two recursive algorithms:
  - find_min(list): recursively find the index of the minimum item
  - selection_sort(list):
    - If the length of the list is one, stop, the list is sorted
    - call find_min() to find the minimum element, swap with the front of the list (if necessary)
    - Call selection_sort() on the rest of the list
      - Stop when ”rest of list” is one item

Recursive DFS, BFS

- Recursive DFS is pretty easy:
  - for each neighbor u of v:
    - If u is ‘unvisited’: call dfs(u)

- Recursive BFS…

Analysis of algorithms

- How long does an algorithm take to run?

  time complexity

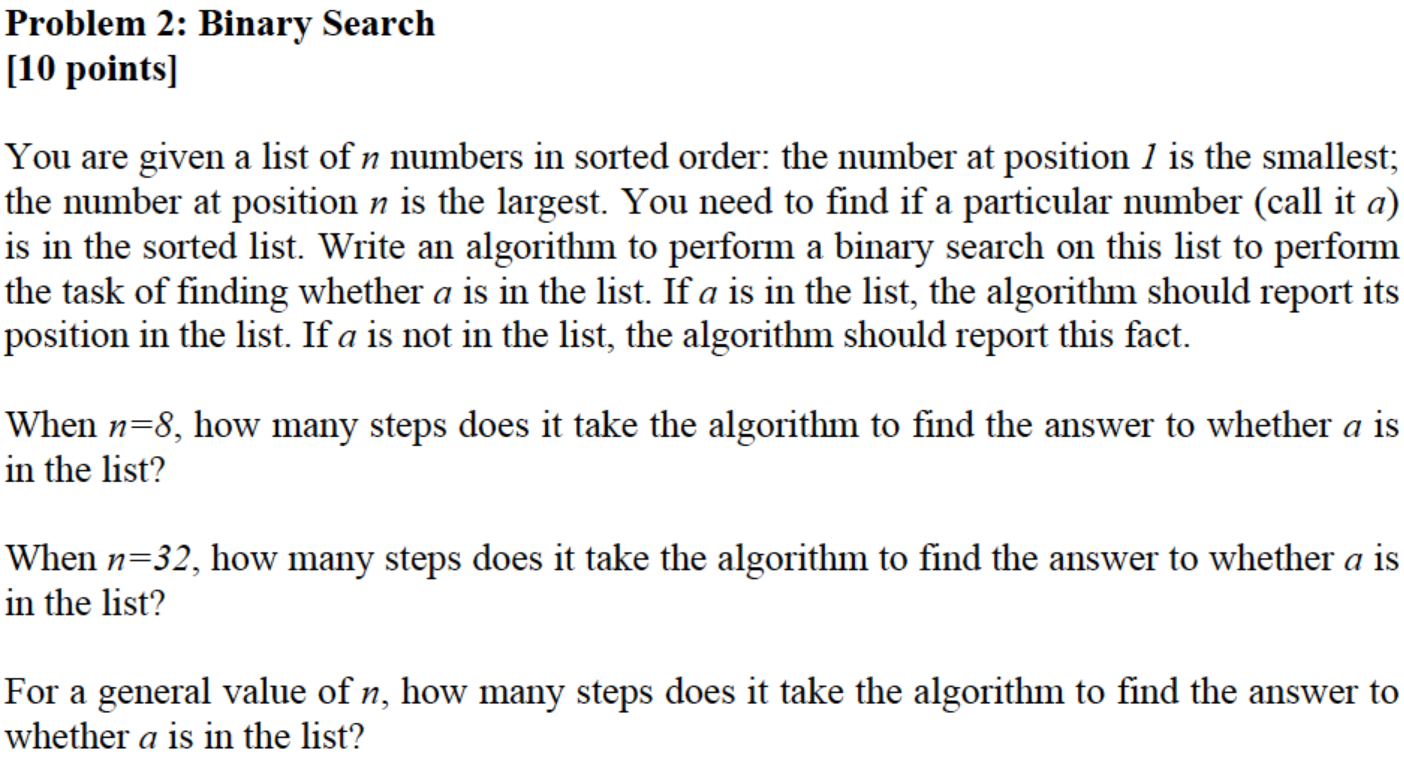
- How much memory does it need?

  space complexity

Estimating running time

- How to estimate algorithm running time?
  - Write a program that implements the algorithm, run it, and measure the time it takes
  - Analyze the algorithm (independent of programming language and type of computer) and calculate in a general way how much work it does to solve a problem of a given size
- Which is better? Why?

Analysis of binary search



Problem 2: Binary Search
[10 points]

You are given a list of *n* numbers in sorted order: the number at position *1* is the smallest; the number at position *n* is the largest. You need to find if a particular number (call it *a*) is in the sorted list. Write an algorithm to perform a binary search on this list to perform the task of finding whether *a* is in the list. If *a* is in the list, the algorithm should report its position in the list. If *a* is not in the list, the algorithm should report this fact.

When *n=8*, how many steps does it take the algorithm to find the answer to whether *a* is in the list?

When *n=32*, how many steps does it take the algorithm to find the answer to whether *a* is in the list?

For a general value of *n*, how many steps does it take the algorithm to find the answer to whether *a* is in the list?

- n = 8, the algorithm takes 3 steps
- n = 32, the algorithm takes 5 steps
- For a general n, the algorithm takes $\log_2 n$ steps

Big O notation

- Characterize functions according to how fast they grow
- The growth rate of a function is called the **order of the function**. (hence the O)
- Big O notation usually only provides an upper bound on the growth rate of the function
- Asymptotic growth

$f(x) = O(g(x))$ as $x \text{ -> } \infty$ if and only if there exists a positive number M such that $f(x) \leq M * g(x)$ for all $x > x_0$

Conventions

- $O(1)$ denotes a function that is a constant
  - $f(n) = 3$, $g(n) = 100000$, $h(n) = 4.7$ are all said to be $O(1)$

- For a function $f(n) = n^2$ it would be perfectly correct to call it $O(n^2)$ or $O(n^3)$ (or for that matter $O(n^{100})$)
- However by convention we call it by the smallest order namely $O(n^2)$
  - *Why?*

What do they have in common?

- (Binary) search of a sorted list: *$O(\log_2 n)$*
- Selection sort: *$O(n^2)$*
- Quicksort: *$O(n \log n)$*
- Breadth first traversal of a tree: O(V)
- Depth first traversal of a tree: O(V)
- Prim’s algorithm to find the MST of a graph: *$O(V^2)$*
- Kruskal’s algorithm to find the MST of a graph: *$O(E \log E)$*
- Dijkstra’s algorithm to find the shortest path from a node in a graph to all other nodes: *$O(V^2)$*

Subset sum problem

- Given a set of integers and an integer *s*, does any non-empty subset sum to *s*?
- {1, 4, 67, -1, 42, 5, 17} and *s* = 24 *No*
- {4, 3, 17, 12, 10, 20} and *s* = 19 *Yes* {4, 3, 12}

- If a set has *N* elements, it has 2^N subsets.
- Checking the sum of each subset takes a maximum of *N* operations
- To check all the subsets takes $2^N N$ operations
- Some cleverness can reduce this by a bit (2^N becomes $2^{N/2}$, but all known algorithms are exponential

Travelling salesperson problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- Given a graph where edges are labeled with distances between vertices. Start at a specified vertex, visit all other vertices exactly once and return to the start vertex in such a way that sum of the edge weights is minimized
- There are *n!* routes (a number on the order of n^n - much bigger than 2^n)
- *O(n!)*

Enumerating permutations

- List all permutations (i.e. all possible orderings) of *n* numbers
- What is the order of an algorithm that can do this?

- So we have:
  - Knapsack/Subset sum: N*2^N
  - Set permutation: n!
  - Traveling salesman: n!

Analysis of problems

- Study of algorithms illuminates the study of <u>classes</u> of problems
- If a polynomial time algorithm exists to solve a problem then the problem is called *tractable*
- If a problem cannot be solved by a polynomial time algorithm then it is called *intractable*
- This divides problems into **three** groups:
  - Problems with known polynomial time algorithms
  - Problems that are proven to have no polynomial-time algorithm
  - Problems with no known polynomial time algorithm but not yet proven to be intractable

Tractable and Intractable

- Tractable problems (**P**)
  - Sorting a list
  - Searching an unordered list
  - Finding a minimum spanning tree in a graph

- Intractable
  - Listing all permutations (all possible orderings) of n numbers

- Might be (in)tractable
  - Subset sum: given a set of numbers, is there a subset that adds up to a given number?
  - Travelling salesperson: n cities, n! routes, find the shortest route

These problems have no known polynomial time solution

However no one has been able to prove that such a solution does not exist

Tractability and Intractability

- ‘Properties of problems’ (*NOT* ‘properties of algorithms’)
- Tractable: problem can be solved by a polynomial time algorithm (or something more efficient)
- Intractable: problem cannot be solved by a polynomial time algorithm (all solutions are proven to be more inefficient than polynomial time)
- Unknown: not known if the problem is tractable or intractable (no known polynomial time solution, no proof that a polynomial time solution does not exist)

Subset sum problem

- Given a set of integers and an integer *s*, does any non-empty subset sum to *s*?
- {1, 4, 67, -1, 42, 5, 17} and $s = 24$ *No*
- {4, 3, 17, 12, 10, 20} and $s = 19$ *Yes* {4, 3, 12}

- If a set has *N* elements, it has 2^N subsets.
- Checking the sum of each subset takes a maximum of *N* operations
- To check all the subsets takes $2^N N$ operations
- Some cleverness can reduce this by a bit (2^N becomes $2^{N/2}$, but all known algorithms are exponential)

P and NP

- **P**: set of problems that can be solved in polynomial time

Easy to solve

(implies easy to check)

- Consider subset sum
  - No known polynomial time algorithm
  - However, if you give me a solution to the problem, it is easy for me to check if the solution is correct – i.e. I can write a polynomial time algorithm to check if a given solution is correct
- **NP**: set of problems for which a solution can be checked in polynomial time

Easy to check if solution is good

Easy to Solve vs. Easy to Check

- Easy to solve: sorting
  - Solve: sort the list in *O(n log n)*
  - Check: is the list sorted? *O(n)*
  - Clearly sorting is in P
- Hard to solve: sub-set sum
  - Solve: generate all subsets: $O(2^n)$
  - Check: sum-up subset. *O(n)*
- Hard to solve: integer factorization
  - Solve: check all numbers between 2 and sqrt(n) $O(2^w)$
  - Check: is one number a factor of another? Divide and check $O(n^2)$

P=NP?

- All problems in **P** are also in **NP**
- Are there any problems in **NP** that are not also in **P**?
- In other words, is

P = NP ?

- Central open question in Computer Science

P vs. NP Example

- Public key encryption uses two large prime numbers *p, q*
- If *k = p*q,* then we can send *k* in the clear need *p* and *q* to decrypt
- Why is this P vs. NP?
  - *p*q* clearly P algorithm
  - Finding *p* and *q* given just *k* is $O(2^w)$ where w = size of the number (digits or bits)
- If P = NP then public key encryption would be "broken"
- Side note: as computers have gotten faster, key size goes up, making problem exponentially harder
  - Keys are now >= 2048 bits -> 2^{2048} is a preposterously large number
  - Check 1B keys/second = 1.7 x 10^{600} years to crack

Midterm Style Questions

Based on the information presented in class and the lecture slides, which component is not part of a modern CPU:

A. Arithmetic/logic unit
B. Program Counter
C. Cache memory
D. Disk controller
E. Registers

In order to find the k-th smallest element in a list of n integers we run as many iterations of Selection Sort as necessary and then we stop. What is the complexity of this algorithm in terms of k, n?

A. O(k*log(n))
B. O(k*n*log(n))
C. O(n*log(n))
D. O(k*n)
E. Not enough information is given to determine the correct answer

Midterm Style Questions

Which of the problems described CANNOT be solved optimally with an MST (minimum spanning tree)?

A. Build the shortest-length bridge network between a set of islands.
B. Eliminate loops in a computer network.
C. Given a list of cities and the distances between each pair, find the shortest possible route that visits each city and returns to the starting city.
D. Eliminate multiple paths between any two vertices in a graph.
E. All of the above CAN be solved optimally with a MST.

Which of the following is **TRUE** about binary search?

A. Considering the input data, binary search will ALWAYS have a smaller runtime vs. sequential search on the same data.
B. Binary search can be applied to any list
C. Binary search has runtime complexity of $O(2^N)$ for an unsorted list
D. Binary search can be implemented recursively
E. None of the above is true

Midterm Style Questions

Which choice for pivot always allows optimal runtime of the quicksort algorithm?

A. Maximum element

B. Minimum element

C. Average among all elements

D. Average between maximum and minimum elements

E. None of the above

You are in a maze and a friend suggests that you put your right hand on the wall and follow the wall until you find the exit. This "right hand rule" represents an algorithm for solving the maze. Which algorithm discussed in class does the approach correspond to?

A. Breadth First Search

B. Depth First Search

C. Kruskal's Algorithm

D. Binary Search

Midterm Style Questions

The Jacquard Loom (and similar machines) are considered information transformers, but not computers. Which answer best describes why:
A. Programming these machines doesn't scale
B. Programming these machines requires punch-cards
C. Machines like these do not have memory or control flow
D. Machines like these are too old to be considered computers

The subset-sum problem has time complexity O(N*2^N). Where does the factor N come from?
A: That is how many subsets a set of size N has.
B: O(N) is the time complexity required to check each possible subset sum.
C: That is the time complexity of the algorithm that generates the subsets.
D: None of the above.