Introduction to Computer Science

CSCI 109

Readings

St. Amant, Ch. 4, Ch. 8

“An **algorithm** (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850.”

Andrew Goodney

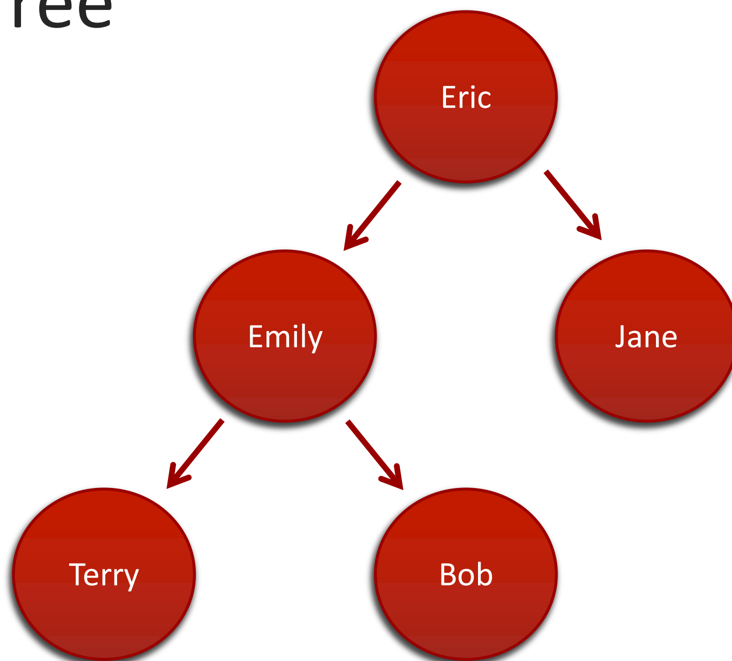
Fall 2019

Where are we?

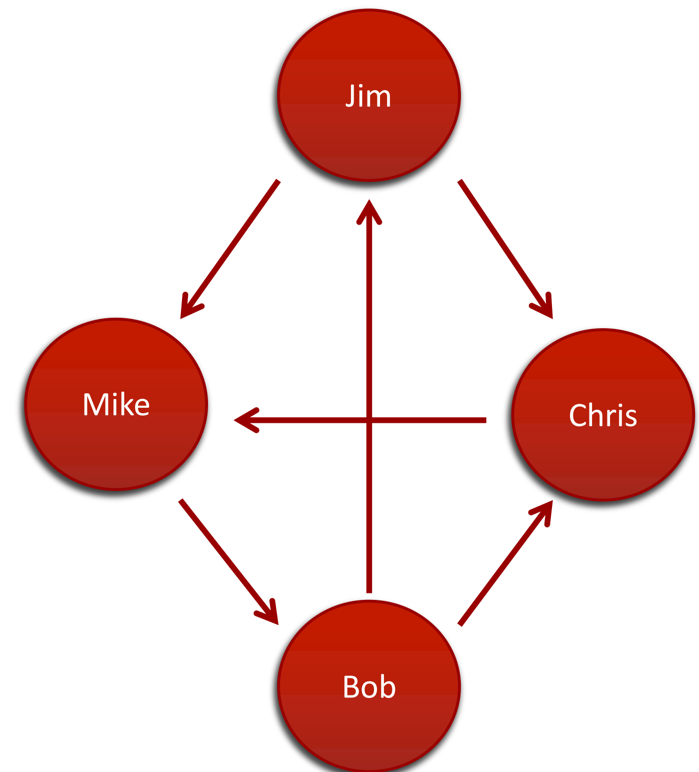
Date	Topic		Assigned	Due	Quizzes/Midterm/Final	Slide Deck
26-Aug	Introduction	What is computing, how did computers come to be?				1
2-Sep	Labor day					
9-Sep	Computer architecture	How is a modern computer built? Basic architecture and assembly	HW1			2
16-Sep	Data structures	Why organize data? Basic 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	

Sequences, Trees and Graphs

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



- Graph



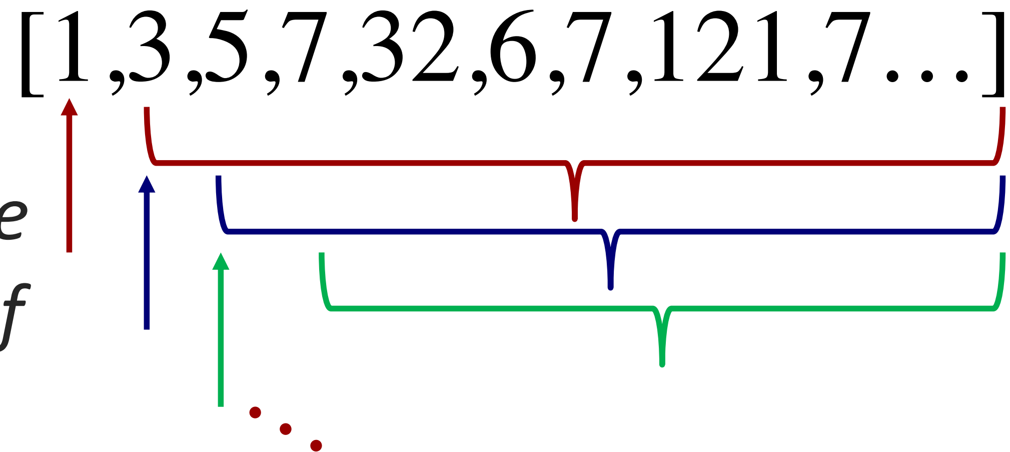
Recursion

- 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…)

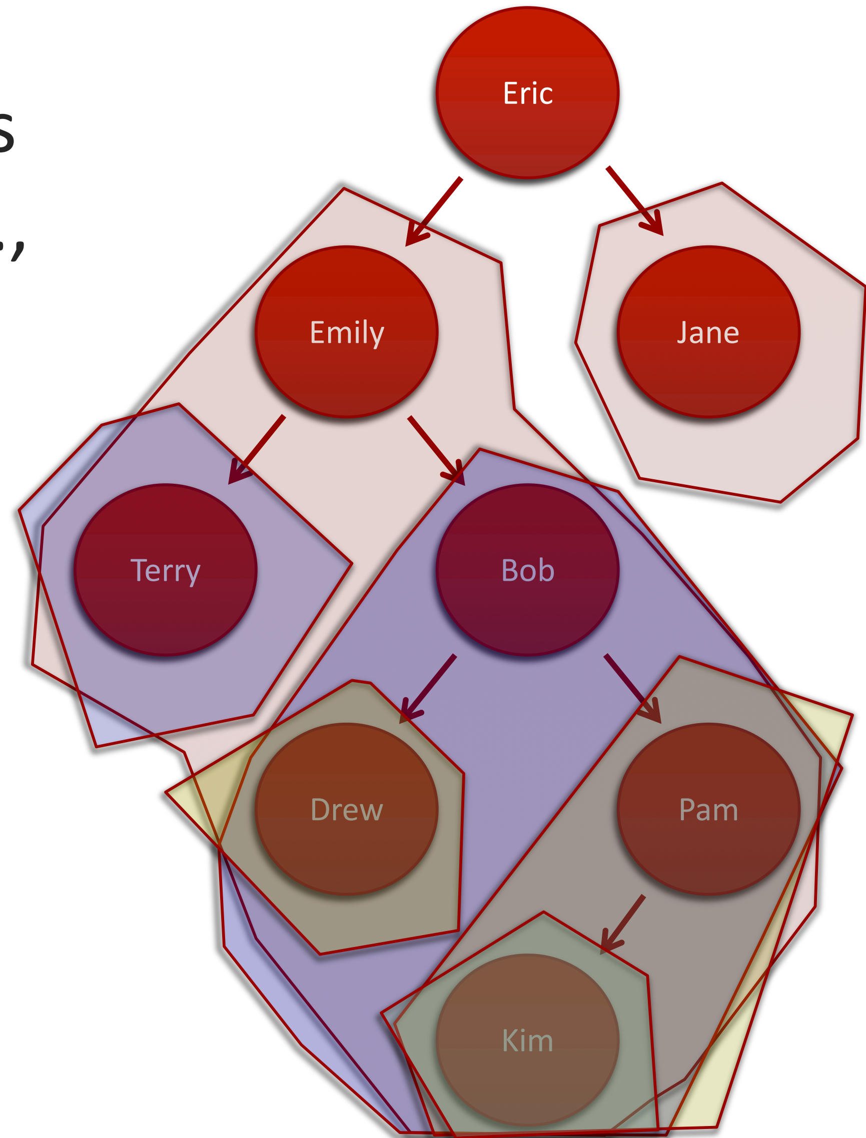


Recursion: abstract data types

- 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?

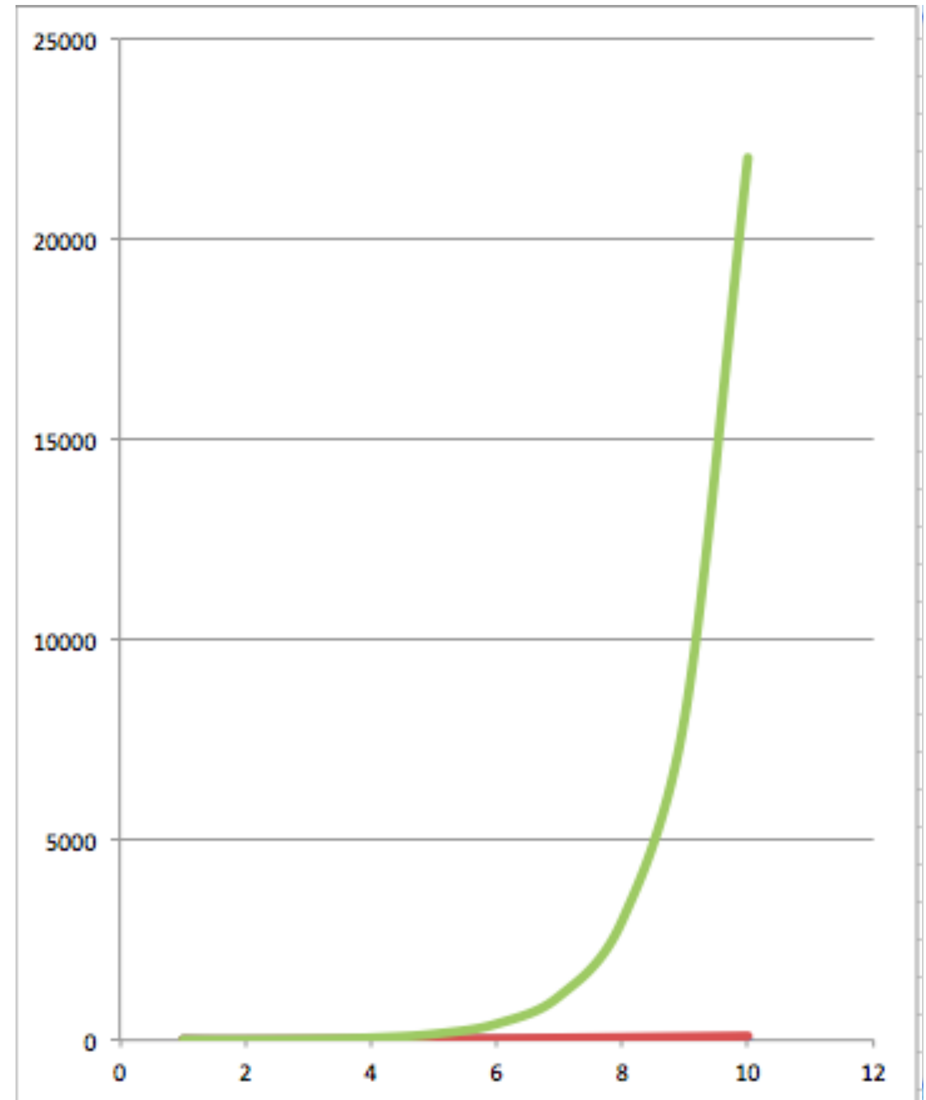
When *n=32*, how many steps does it take the algorithm to find the answer to whether *a* is in the list?

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

Growth rates of functions

- Linear
- Quadratic
- Exponential



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 $x \to \infty$ if and only if there exists a positive number M such that $f(x) \leq M * g(x)$ for all $x > x_0$

Examples

- $f(n) = 3n^2 + 70$
  - We can write $f(n) = O(n^2)$
  - What is a value for M?
- $f(n) = 100n^2 + 70$
  - We can write $f(n) = O(n^2)$
  - Why?
- $f(n) = 5n + 3n^5$
  - We can write $f(n) = O(n^5)$
  - Why?
- $f(n) = n \log n$
  - We can write $f(n) = O(n \log n)$
  - Why?
- $f(n) = \pi n^n$
  - We can write $f(n) = O(n^n)$
  - Why?
- $f(n) = (\log n)^5 + n^5$
  - We can write $f(n) = O(n^5)$
  - Why?

Examples

- $f(n) = log_a n$ and $g(n) = log_b n$ are both asymptotically $O(log\ n)$
  - The base doesn't matter because $log_a n = log_b n / log_b a$, $M = 1/log_b a$
- $f(n) = log_a n$ and $g(n) = log_a(n^c)$ are both asymptotically $O(log\ n)$
  - Why?
- $f(n) = log_a n$ and $g(n) = log_b(n^c)$ are both asymptotically $O(log\ n)$
  - Why?
- What about $f(n) = 2^n$ and $g(n) = 3^n$?
  - Are they both of the same order?

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?*

Complexity

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

Selection sort

- Putting the smallest element in place requires scanning all n elements in the list (and $n-1$ comparisons)
- Putting the second smallest element in place requires scanning $n-1$ elements in the list (and $n-2$ comparisons)
- …
- Total number of comparisons is
  - $(n-1) + (n-2) + (n-3) + \dots + 1$
  - $n(n-1)/2$
  - $O(n^2)$
- There is no difference between the best case, worst case and average case

Quicksort

- Best case:
  - Assume an ideal pivot
  - The average depth is $O(\log n)$
  - Each level of processes at most n elements (compare to pivot, move)
  - The total amount of work done on average is the product, $O(n \log n)$
  - Why is ideal pivot important? What breaks/changes in above if pivot is “bad”?
- Worst case:
  - Accidentally (or on purpose) chose max (or min)
  - Each time the pivot splits the list into one element and the rest
  - Each level processes at most n elements… but
  - How many levels? n levels * n/level = $O(n^2)$
- Average case:
  - $O(n \log n)$ [but proving it is a bit beyond CS 109]

BF and DF traversals of a tree

- A breadth first traversal visits the vertices of a tree level by level
- A depth first traversal visit the vertices of a tree by going deep down one branch and exhausting it before popping up to visit another branch
- What do they have in common?

BF and DF traversals of a tree

- A breadth first traversal visits the vertices of a tree level by level
- A depth first traversal visit the vertices of a tree by going deep down one branch and exhausting it before popping up to visit another branch
- What do they have in common?
- Both visit **all** the vertices of a tree
- If a tree has V vertices, then both BF and DF are $O(V)$

Prim's algorithm

- Initialize a tree with a single vertex, chosen arbitrarily from the graph
- Grow the tree by adding one vertex. Do this by adding the minimum-weight edge chosen from the edges that connect the tree to vertices not yet in the tree
- Repeat until all vertices are in the tree
- How fast it goes depends on how you store the vertices of the graph
- If you don't keep the vertices of the graph in some readily sorted order then the complexity is $O(V^2)$ where the graph has V vertices
  - Intuition: at each vertex search O(V) for minimum to add = $V*O(V) = O(V^2)$
  - *Can do better with some fancy data structures*

Kruskal's algorithm

- Initialize a tree with a single edge of lowest weight
- Add edges in increasing order of weight
- If an edge causes a cycle, skip it and move on to the next highest weight edge
- Repeat until all edges have been considered
- Complexity
- $|E|$ = number of edges, $|V|$ = number of vertices
  - We need to sort the edges = $O(|E| \log |E|)$
  - Then add in increasing order of weight, one per vertex
    - 'disjoint data set' $O(|V| \log |V|)$
- Total
  - $O(|E| \log |E|) + O(|V| \log |V|) = O(|E| \log |E|)$

Dijkstra's algorithm

- At each iteration we refine the distance estimate through a new vertex we're currently considering
- So for each of *V* vertices, we update O(*V-1)* paths
- In a graph with *V* vertices, a loose bound is $O(V^2)$

Recap

- (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: *$O(V^2)$*

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: $O(V^2)$

A knapsack problem

- You have a knapsack that can carry 20 lbs
- You have books of various weights
- Is there a collection of books whose weight adds up to exactly 20 lbs?
- Can you enumerate all collections of books that are 20 lbs

Book	Weight
Book 1	2
Book 2	3
Book 3	13
Book 4	7
Book 5	10
Book 6	6

A knapsack problem

- You have a knapsack that can carry 20 lbs
- You have books of various weights
- Is there a collection of books whose weight adds up to exactly 20 lbs?
- Can you enumerate all collections of books that are 20 lbs

Book	Weight
Book 1	2
Book 2	3
Book 3	13
Book 4	7
Book 5	10
Book 6	6

A knapsack problem

- You have a knapsack that can carry 20 lbs
- You have books of various weights
- Is there a collection of books whose weight adds up to exactly 20 lbs?
- Can you enumerate all collections of books that are 20 lbs

Book	Weight
Book 1	2
Book 2	3
Book 3	13
Book 4	7
Book 5	10
Book 6	6

How many combinations are there?

# of books	Combinations	Combinations
0	{}	1
1	{2} {3} {13} {7} {10} {6}	6
2	{2,3} {2,13} {2,7} {2,10} {2,6} {3,13} {3,7} {3,10} {3,6} {13,7} {13,10} {13,6} {7,10} {7,6} {10,6}	15
3	{2,3,13} {2,13,7} {2,7,10} {2,10,6} {2,3,7} {2,3,10} {2,3,6} {2,13,10} {2,13,6} {2,7,6} {3,13,7} {3,13,10} {3,13,6} {3,7,10} {3,7,6} {3,10,6} {13,7,10} {13,10,6} {13,7,6} {7,10,6}	20
4	{2,3,13,7} {2,3,13,10} {2,3,13,6} {2,3,7,10} {2,3,7,6} {2,3,10,6} {2,13,7,10} {2,13,10,6} {2,13,7,6} {2,7,10,6} {3,13,7,10} {3,13,10,6} {3,13,7,6} {3,7,10,6} {13,7,10,6}	15
5	{2,3,13,7,10} {3,13,7,10,6} {13,7,10,6,2} {7,10,6,2,3} {10,6,2,3,13} {6,2,3,13,7}	6
6	{2,3,13,7,10,6}	1
	TOTAL	64

How many combinations are there?

# of books	Combinations	Combinations
0	{}	1
1	{2} {3} {13} {7} {10} {6}	6
2	{2,3} {2,13} {2,7} {2,10} {2,6} {3,13} {3,7} {3,10} {3,6} {13,7} {13,10} {13,6} {7,10} {7,6} {10,6}	15
3	{2,3,13} {2,13,7} {2,7,10} {2,10,6} {2,3,7} {2,3,10} {2,3,6} {2,13,10} {2,13,6} {2,7,6} {3,13,7} {3,13,10} {3,13,6} {3,7,10} {3,7,6} {3,10,6} {13,7,10} {13,10,6} {13,7,6} {7,10,6}	20
4	{2,3,13,7} {2,3,13,10} {2,3,13,6} {2,3,7,10} {2,3,7,6} {2,3,10,6} {2,13,7,10} {2,13,10,6} {2,13,7,6} {2,7,10,6} {3,13,7,10} {3,13,10,6} {3,13,7,6} {3,7,10,6} {13,7,10,6}	15
5	{2,3,13,7,10} {3,13,7,10,6} {13,7,10,6,2} {7,10,6,2,3} {10,6,2,3,13} {6,2,3,13,7}	6
6	{2,3,13,7,10,6}	1
	TOTAL	64

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 – i.e. $O(2^N N)$

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 $x \to \infty$ if and only if there exists a positive number M such that $f(x) \leq M * g(x)$ for all $x > x_0$

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?

Enumerating permutations

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

- 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 classes 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 #?   groups:

Analysis of problems

- Study of algorithms illuminates the study of classes 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)

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)

Take away

- Some simple problems seem to be very hard to solve because of exponential or factorial run-time
- Not so simple in practice:

Problem	Naïve Solution(s)	Best?
Knapsack	$N*2^N$	$N*2^{N/2}$, pseudopolynomial
Subset-sum	$N*2^N$	$N*2^{N/2}$, pseudopolynomial
Travelling Salesman	N!	$N^2 2^N$

Pseudopolynomial?

- Sometimes we have to be careful about choosing ‘n’ a.k.a the size of the problem
- There are dynamic programming solutions to the subset-sum (and a lot of other similar problems) that appear to be polynomial time
- But on further inspection, if you chose n to be the size of the numbers in the problem (bits or digits) then the solution is exponential time

P and NP

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

(implies easy to check)

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=NP?

- Why do we care?
- "Aside from being an important problem in computational theory, a proof either way would have profound implications for mathematics, cryptography, algorithm research, artificial intelligence, game theory, multimedia processing, philosophy, economics and many other fields."

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