-to avoid redundancy (computing the same problems), we memoize

**Memoization:** we keep a map of solved subproblems. To solve a subproblem, we first look up the problem in the map of solved subproblems. If we find the subproblem in the map we use the corresponding answer. If not, we solve the problem and add the solution to the map.

-dynamic programming’s main idea is to keep a matrix of the subproblem answers and fill it in using an order that guarantees that a subproblem’s solution is known before you use it to fill in another place in the table

**Iterator:** can see if an element appears in the list by getting each item and seeing if it’s what we wanted, and we can test equality of two lists by marching down each of them in tandem and comparing items

**Tree Traversal**

-the basic pattern of computing something in a tree is that inner nodes recurse left and right and combine results from their children, while leaf nodes provide the base cases

**Postorder:** recurse with the left, then with the right, and then do something with those results as the

parent

**Preorder:** parent does something first, and then the two children do something

**Inorder:** do something with the left subtree, then the parent, and then the right subtree

**Eulerian Walk:** do something with the parent, then the left, then the parent, then the right, then the

parent

-recursion is essentially pushing info for the calls onto a stack (where to return & with what variables, after the recursive call), we can maintain that same info in a stack ourselves

-anonymous functions are compact ways to write a function in the middle of a piece of code, without bothering to give it a name…advantage is that everything is contained and that it has access to all of the instance variables

**Pipelines**

-proceed from a source, through a series of operations, to some final step

ex.

double len = words.stream()

.mapToInt(w->w.length()) //create a new stream of integers

.average()

.getAsDouble();

-they are lazy, in that something is computed only when it needs to be

**Graphs**

-asymmetric (directed) and symmetric (undirected)

**Edge List**

-a list of neighboring pairs

-insert is fast, but everything else is slow (O(# number of edges time))

**Adjacency List**

-the neighbors of a vertex are stored in a linked list

-makes getting neighbors fast

**Adjacency Matrix**

-matrix of 0s and 1s (no’s and yes’)

-makes discovering if there is an edge very efficient

**Depth First Search**

-what you do in a maze…keep going deeper and deeper in the maze, making choices until you hit a dead end

push the start vertex v onto the stack

repeat until we find the goal vertex or the stack is empty

pop the next vertex v from the stack

if v has not been visited

remember that v has been visited (and maybe do something while here)

for all vertices v’ that are adjacent to v

if we haven’t already visited v’

push v’ onto the stack

**Breadth First Search**

-expands uniformly in all directions, like radiating ripples; the directions here are edges out of a vertex

-instead of just keeping track of which vertices we’ve visited, it’s helpful to keep track of how we arrived

enqueue the start vertex v onto the queue and mark it

repeat until we find the goal vertex or the queue is empty

dequeue the next vertex v from the queue (maybe do something while here)

for all vertices v’ that are adjacent to v

if haven’t already added v’

remember that v’ has been added

enqueue v’ onto the queue

**Dijkstra’s Algorithm**

-greedy algorithm, at each step you take what looks like the best choice, with no look-ahead

-keep a priority queue of all of the vertices with the key for each bertex being the distance to the start vertex found so far

Initialize map of distances: dist[u]=0 and dist[v]=infinity for v!=s

Create priority queue containing all vertices, with distance as the priority

while priority queue is not empty

u=priorityqueue.removeMin()

for each edge (u,v) such that the vertex is still in the priority queue

if dist[u]+weight(u,v)<dist[v]

dist[v] =dist[u]+weight(u,v)

update vertex v in priority queue

**A\* Search**

-use an estimate of the distance remaining and add it to the distance traveled so far to get the key for the object

-DFS can lead to very long paths, BFS will guarantee the shortest path but is slow , but A\* will find the shortest path in less time

**Finite Automata**

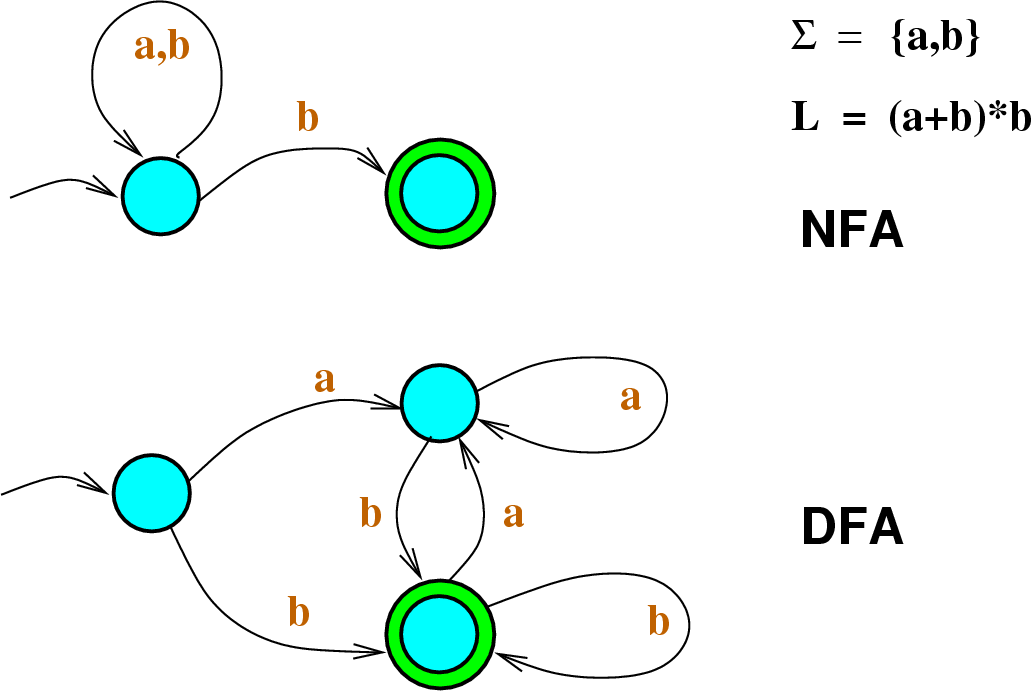
-a rule describing how to scan a sequence of input symbols, one at a time ,and decide whether to “accept” or “reject” the sequence

**Deterministic Finite Automata**

-exactly one choice for each possible symbol at each state

**Nondeterministic Finite Automata**

-may have zero, one, or more than one choice for each possible symbol at each state

**Hidden Markov Models**

-driven by transitions from state to state…which transitions are

allowed depends only on the current state, not the history of how

we got there

-emission is a property of the state that we’re in, not of the transition

**Viterbi**

-keep track of which states we could be in for each observation expanding

from the state before any observations, following all its transitions

to get to where the first emission could be made, following all their

transitions to get to where the second emissions could be made, etc.

prevStates={start}

prevScores=map{start=0}

for i=0…length(observations)-1

nextStates={}

nextScores=empty map

for each state in prevStates

for each transition from state to nextState

add nextState to nextStates

nextScore=prevScores[state]+logP(nextState|state)+logP(oservations[i]|nextState)

if nextState isn’t in nextScores or nextScore>nextScores[nextState]

set nextScores[nextState] to next Score

remember that we got to nextState for observation i from state

prevStates=nextStates

prevScores=nextScores

-XML is related to HTML in that they are both languages (sets of syntax and grammar rules) describing how to represent something

-sockets enable two different program to communicate with each other…the server publically distributes a way to make that initial connection, but since all clients will be trying to reach the server there, once it makes a connection with a new client, it shifts them over to their own connection

-must synchronize methods otherwise it can lead to deadlock, starvation, or an access error

-deadlock is when the pattern of synchronization has produced a state where no thread may proceed because each needs a resource that another is holding. Starvation is when something never gets the resource it needs

-deadlock requires mutual exclusion, hold and wait, no preemption, and a circular wait

-hashing uses an array to store data and uses a hash function to get from an object to a particular spot in the table, use chaining or open addressing to avoid collisions, chaining takes up space, open addressing complicates deleting

-a good hash function can be computed easily, spreads the universe of keys fairly evenly over the table, and allows small changes in the key to result in a different hash value

**2-3-4 Trees**

-gives up binary, allows multiple keys to be stored at each node, a node will have one more child than it has keys

**Rules:** every node has either 2, 3, or 4 children (1,2, or 3 keys) and all the leaves of the tree are on the

same level

-problems: have three different types of nodes, and keep creating new ones as you need to…have each node big enough to hold a 4-node

**Red-Black Properties**

-gives up perfectly balanced

1. Every node is either Red or Black; by convention, the empty tree is a leaf, and is always considered Black
2. The root is black. If an action colors it red, change it back to black.
3. For any node which is Red, both of its children are Black…no two nodes are red
4. Every path from any node to a descendent leaf has the same number of black nodes

-first insertion is black and then every insertion after that is red

**Skip Lists**

-the bottommost list keeps all of the elements in sorted order, with end markers –INF and INF

-use a coin flip or random number generator to see whether or not an insertion will have a tower

**Tries**

-an alternative data structure that allows efficient lookup based on the actual content of the strings in the map

-a trie is a multi-way tree…gives us linear time search

1st 10-15, 2nd 15-17, 3rd more