**CS 212 Notes**

*Due Date: Saturday, 5 pm each week*

*Help session: Monday/Thursday 7-9pm SB 354*

Week 1: empirical and theoretical

* Data structure, algorithm for solving problems
* Program runtimes
  + Empirical analysis: implement, gather runtime data for various input, analyze runtime vs size
    - Startup cost significant for small size. As it gets bigger not so significant because negligible
    - Straight line for polynomial
    - implementation dependent
  + theoretical analysis: high level description of algorithm not implementation
    - running time as function of input size = n
    - all possible input -> analyzing worst case
    - evaluate speed of algorithm independent
    - count ram-equivalent statement executed as function of input size
    - \*some statement not constant time
  + pseudocode= idea not syntax
  + estimating running time:
    - cpu counts number of instruction
    - each statement constant time
  + growth rate:
    - slope 1:1 = n
    - slope 2:1 = n^2
  + constant factor:
    - growth rate not affected by constant factor
    - lower order term
  + big-O notation
* Homework 1:
  + Equation -> lg(n) and lg(runtime)

Week 2: big-o, big-theta, big-omega

* Homework 2:
  + Logs and big-o
* Asymptotic analysis
  + After some point, this function bigger
  + Slope for large enough inputs -> big-o notation
  + Ignore multiplicative constant because its machine dependent = different for each machine, language, etc -> but all have the same big-o
  + Ignore lower order term
  + Upper bound on growth rate
  + summary:
    - N=problem size
    - Big-oh=upper bound over input size n
    - Constant factor -> ignore because machine dependent
    - As n grows larger
* Prefix averages
  + Keep track of sum
* Arithmetic progression
  + Lower right half is “filled”
* Sequences = blocks -> you add
* Relative of big-oh
  + Big-oh (<=)
  + Big-omega (>=)
  + Big-theta (=)
  + Little-omega (>)
  + Little-oh (<)
* Polynomial > polylogs
* Test 1
  + Online, closed book but can bring 1 paper

Week 3: data structure

* Program 1
  + Compute floor of log-log n
  + C# console app type
  + Upload zip directory
* Abstract data types = abstraction of data structure; do not specify implementation detail; interface
  + Data stored, operations on data, error condition
  + Implemented with various data structure
  + Exceptions are thrown
* Data structure:
  + Stacks (LIFO)=ADT; push, pop, growable array when filled
  + Queues (FIFO)=ADT; enqueue, dequeue
  + Linked list=singly, doubly
  + Sequences= array, max size, length
  + trees=red-black tree; nodes (child/parent) and roots, abstract model of hierarchical
    - runtime of traversal O(n)
      * inorder: left, visit, right
      * pre-order: visit, left, right
      * post-order: left, right, visit
    - binary trees: node have 2 children, balanced
      * in array: i is the index
        + parent of child(i) = i/2
        + child of parent(i) at left -> i=2i and right -> i=2i+1
    - euler tour: pre-order, post-order, inorder
  + height of trees = O(logn)
    - height = root to longest leaf
    - height X = sub tree -> node X to longest leaf
    - depth X = root to node X
* call frames = all the data you need to store to call functions

Week 4

* internal node = everything – leave nodes = every non-leaves nodes
  + external node = leaves
* root is a node
* priority queue= stores collection of item, and item is pair; insert, removeMin
  + total order relation
  + comparator ADT
  + sequence based:
    - unsorted: O(1) for insert O(n) for remove
    - sorted: O(n) for insert O(1) for remove
  + selection sort= select smallest and swap to the 0,1,2,3… index; runtime is O(n^2); insert and remove O(n)
  + insertion sort= start at left and go right at index 0,1,2,3… and swapping to the left if out of order until in right order; runtime is O(n^2); insert and remove O(n)
* heap (priority queue)= binary tree with smallest number at root and child >= parent
  + construct: array to heap = top to down, left to right, from index 0 to n-1
  + parent if bigger swaps with smaller child
  + last node = right most at lowest depth
  + insertion= next right most at lowest depth -> restore heap order
    - upheap= swapping along upward path when parent bigger than child; runtime is O(logn)
    - updating= from last node go up until root and go down on right child of root till leaf
  + removeMin= remove root; replace root with last node -> restore heap order
    - downheap= swapping along downward path with the smaller child; runtime is O(logn)
  + heap-sort priority queue (generally max heap) = insert and remove O(logn), space O(n); runtime is O(nlogn)
    - array -> build heap -> heapify -> remove min
    - max heap= parent>=child
    - min heap= parent <= child
    - build tree then switch to max heap = left to right, down to up, and at every swap check with its children
  + vector based heap= similar to binary trees in array -> left child = 2i and right child = 2i + 1
    - i is the index; start at index 1
  + build heap
    - bottom-up heap = runtime is O(n);
* question: max/min heap + bottom up
* dictionary
  + main oper: find, remove, insert
  + ordered dictionary= find, remove, insert, min, max, predecessor, successor
  + traversing= O(n)
  + array= insert O(1), delete O(1) if given location, find O(n)
  + sorted array= insert/delete O(n), find O(logn)
  + binary search tree
    - find/insert/delete O(logn)
  + logfile= unsorted array
    - insert O(1), find/remove O(n)
  + lookup table= dictionary with sorted sequence
    - find O(logn), insert/remove O(n)
  + hash function= makes a number out of the value -> use that to find
    - runtime is O(n) worst case, O(1) amortized
      * same for insert, remove, find
    - collision = in a hash when two word point to same array location
      * 1.chaining
        + insert O(1), successful search O(n/2m), unsuccessful search O(n/m)

O(n) for worst case

* + - * + sorted list: insert/unsuccessful search O(n/2m)
        + unlinked deleted
        + easiest to deploy
      * 2.linear probing
        + if full, find the next open location by ++index
        + cluster slows down
        + search is same
        + insert O(1) if table sparse, else O(n)
        + mark as deleted
        + fastest but more memory
      * clustering- probes
        + insert= (1 + 1/(1-N/M)2)/2
        + search (hit)= (1 + 1/(1-N/M))/2
        + search (miss)= insert
        + N = number of items
        + M = number of spaces
      * 3.double hashing
        + using a second hash function, if full, skip by x-number
        + better than chaining but will get full -> make space bigger
        + mark as deleted/rehash
        + least memory but takes time to compute 2nd hash function
        + insert/search (miss)= 1+1/(1-N/M)
        + search (hit)= ln(1+N/M)/(N/M)
      * dynamic hash table
        + grow when it fills, new table, new hash function, rehash, delete old
        + any number of insert < 3x
        + insert slower
    - avg= const time
    - h(k) = k mod M, M=table size (prime number)
      * 1) hash code map (key -> int)
        + Runtime is O(n)
      * 2) compression map (int -> mod)
* File tamper test
  + Cryptographic hash= a hash function where it's hard to hard to do reverse computation, but easy to check the answers
* Password verification
  + Salted hash function= hash function that adds a string to value being hashed
* Cache filenames
* Babble ~ google translate
* Test 2

Week 5

* Binary search tree
  + find/insert/delete O(logn)
  + Left child < parent < right child
  + balanced= O(logn), worst-case=O(n)
    - BST not always balanced
    - Log n -> 2log n
  + Delete root:
    - largest on left subtree of root (predecessor)
    - smallest on right subtree of root (successor)
  + multiway search tree (binary)
    - Each internal node >= 2 children, stores d-1 key element and d=number of children
    - Inorder traversal
    - Multiway searching=less, between, greater
    - (2,4) tree = lazy tree
      * Handle overflow with split
        + 4 element in 1 node -> push 3rd element up = max is 3 element in 1 node
        + O(logn)
      * All leaf same depth
      * Internal nodes at most 4 children
      * Perfectly balance
        + Grows in height, every leaves one level deeper
      * Height=Log(base x)n ~ log n
      * Insert/search/delete = worst case-O(logn)
      * Insert
        + Overflow and split
      * Deletion
        + Delete->steal->merge
    - B-tree
      * Like a (2,4) tree but with B number of keys
      * search is O(log (base b) n)
    - Red-black tree~ in a node, 1 element is black, rest of the element in the node is red; insert, search, delete, min, max, predecessor, successor
      * Root/leaf/internal nodes is black
        + Can’t have 2 red in a row
      * When inserting->rebalance by rotating if theres 2 red in a row
        + O(1)
        + Newly inserted node is red
      * Recoloring: double red on parent-child when parent red node has red sibling
        + On the newly inserted node, parent/sibling black, grandparent red
      * Reconstructing: double red on parent-child when parent red node has black sibling
        + Turn -> switch the root if have to -> Root becomes black and the children becomes red
      * Leaves have same black depth
        + But all leaves not same depth
      * Children of red is black
      * Height is O(logn) and O(2logn)
      * Number of black nodes on any path from root to leaf is the same
      * Deletion
        + O(logn)

Week 6

* Divide and conquer= general algorithm design paradigm
  + divide to subproblem -> recursive -> combine; O(logn)
  + Merge sort; divide in half -> recur (smallest to largest in each subproblem) -> combine two list into 1 big list
    - Use when there are mote items to sort than will fit in main memory
    - Double pointer at both beginning of sub problem
    - Worst case Runtime O(nlogn)
      * Merge step =O(n)
      * Levels are ‘heights’ when dividing=O(logn)
  + Quicksort (general sorting)= divide into half using a pivot and uses partition; divide into list of small and large number -> recur -> append 1 list to the other
    - Partition= pick a pivot at random; one half smaller, other half bigger
    - Double pointer at the beginning and end of sub problem
    - Randomized algorithm
    - Avg Runtime O(nlogn)
      * Height when dividing= O(logn)
      * Partition step/dividing=O(n)
      * Worst case O(n^2)
    - Recursion takes a lot of time -> create stack/memory
      * Size <=15 use insertion n sort
* Comparison based sorting= decision tree
  + Lower bound sorting
    - Height= log(n!)
      * Atleast
    - Caveat
      * it doesn’t apply to parallel algorithm
      * it doesn’t apply to sort that do more with numbers than just comparing them
  + Runtime is O(nlogn) for comparison based algorithm | comparison sequential sort
    - All comparison are parallel
* Bucket sort
  + Uses keys as indices
  + Phase 1: O(n)
  + Phase 2: O(n+N)
  + Takes O(n+N)
  + Stable sort property
* Lexicographic order
  + O(dT(n)), T(n) is running time of stable sort
    - D= number of orders

Week 7

* radix sort
  + bucket sort as stable sorting alg in each dimension
    - sort must be stable=when 2 item have same value in column, matters what order they end up in
  + runtime is O(d(n+N))
    - d= digit in input number
    - N = B =base for representing number
  + for binary numbers:
    - runtime is O(bn)
    - each element as b-tuple of integer = b-bit integer
  + non-comparative
  + LSD faster than MSD when there is fixed length
* parallel algorithms
  + data or instruction parallelism
  + exclusivity
    - EREW/CRCW…
    - elect a leader
      * exclusive write
        + O(logn)
        + Select max processor number= every memory cell only read/write by 1 processor at a time
      * Concurrent write
        + O(1)
        + Everyone write processor number to single memory location -> win is leader = multiple processor can read but only 1 write at a time
  + single input single data = SISD
  + multiple input multiple data= MIMD
* parallel max
  + tournament algorithm: compare/swap element pairwise
  + runtime is O(logn) w/ (n/2) processor
  + even number processor->compare to my left
  + biggest is index 0
  + efficiency is (1/logn)
    - ideal is P
  + speedup is (n/logn) = nlogn/(nlog^2n)=
    - ideal is 1
* parallel divide and conquer
  + lend itself to parallelism by algorithm decomposition
* odd-even merge
  + runtime is O(logn)
    - number of comparator O(n log^2 n)
  + 0-1 principle: sort 0 and 1 correctly -> any sequence correctly
  + 0=white, 1=gray
  + Comparing=constant time
* parallel merge sort
  + sort halves in parallel
    - Merge sort (divide in half and recur) -> merge odd/even -> compare/swap
  + use odd-even
    - recursive
  + runtime is O(log^2 n) on number of comparator n processor
    - log n level of merge sort each takes log n
  + speedup: n/logn = single processor/odd-even-runtime
  + efficiency: 1/logn
* sorting network
  + sort 2: 1 comparator
  + merge 2-4: two sort-2 and 1 comparator
  + sort 4: two sort-2 and a merge 2-4
  + merge 4-8: two merge 2-4s and 3 comparator
  + sort 8: two sort-4 and a merge 4-8

Week 8

* greedy method
  + general algorithm
    - configuration: different values to find
    - objective function: score assigned to configuration to maximize or minimize
  + greedy choice= global optimal solution found by series of local improvement from starting configuration
* fractional knapsack problem = hacksaw/greedy algorithm
  + greedy solution
    - calculate value=benefit/weight of each item
    - add from the highest value (once) and go down
    - fractional: benefit\*(x/weight)
  + E(b(x-1/w-1)
    - B=positive benefit/value
    - X-1=
    - W-1=weight
  + Runtime is O(nlogn)
    - Sort by value per unit weight
* Task scheduling-> start time and use few machine as possible
  + Runtime O(nlogn)
  + Start time=s-1
  + Finish time=f-1
* Divide-conquer for binary search recurrence
  + Fractal= never ending pattern
  + Recurrence equation
    - (add recursive T(N)) + (runtime of algorithm)
  + Look it up = master theorem
    - T(n)= aT(n/b) + f(n)
    - n^(log (base b) a)
    - 1) f(n) doesn’t dominate = O(n^(log (base b) a))
    - 2) same = O(f(n)\*logn)
    - 3) f(n) dominate = O(f(n))

Week 9

* Dynamic programming= problems that can be solved recursively and subproblem overlap
  + Bottom up= from 1 and up; O(v)
  + memoize= compute and store for later use; O(v)
  + Method:
    - Write definition of value to compute
    - Express solution in sub problem solutions
    - Bottom up/memoize
  + NC(V)= 1 + min(NC(V-c))
  + Making change for coin -> greedy algorithm
    - Runtime is O(v) – exponential?
  + Simple subproblem
  + Subproblem optimality
  + Subproblem overlap
* 0/1 knapsack problem= dynamic programming
  + Not allowed to take fractional amount
    - k from Sk
    - w = weight
    - Wk = weight of the k element
  + Runtime O(w\*k)
* Longest common subsequence
  + If last letter same, 1+LCS(i-1, j-1)
  + Else, max[LCS(i, j-1), LCS(i-1, j)]
  + Runtime O(n\*m)
* Graph traversal
  + Depth-first search
  + Breadth-first search

Week 10

* Graph terminology
  + Nodes = vertices
  + Pairs of vertices = edge
  + Edge types: direct/undirected edge/graph
  + Adjacent = vertices
  + Incident = edges
* Graph
  + Simple/non-simple path/cycle
  + Example= sibling relationship
  + N=vertices
  + M=edges
  + Total most edge=n(n-1)/2
    - Each vertex at most (n-1) edges
  + Adjacency list
    - Repeats
    - Space required= O(n+m)
    - Time to tell if theres an edge from v1 to v2 = O(n)
  + Adjacency matrix
    - No repeats
    - Space required = O(n^2)
    - Time if theres edge from v1 to v2= O(1)
* Depth first search = keep unrolling string, if stuck roll back
  + Just go forward, picking smallest alphabet first, then backtrack
  + Discovery edges -> spanning tree
  + First coin = visited
  + Second coin = no new vertex to visit
  + Connected graph = tree
  + Not connected = forest
  + Runtime O(n+m) for adjacency list
    - Runtime O(n^2) for adjacency matrix
  + Connected, spanning tree, if graph have cycle, path between two vertices, biconnected
  + Height of tree =
  + Find and report a path between two given vertices
  + Find a cycle in the graph
* Breadth first search = unroll string once, then twice, …
  + Visit all the adjacent vertex first with backtrack
  + Runtime O(n+m) for adjacency list
    - Runtime O(n^2) for adjacency matrix
  + Connected, spanning, paths, cycles, shortest path
  + Height of tree = from anchor to the last level
  + Find and report a path with the minimum number of edges between two given vertices
  + Find a simple cycle, if there is one

Week 11

* Directed graph
  + Example=child of relationship
  + Digraph= all edge directed
  + Directed DFS
    - Weakly and strongly connected
    - Strong connectivity: Runtime O(n+m)
      * Use reverse DFS to find if its strongly connected
  + Transitive closure
    - Floyd warshall algorithm
    - Making more edges to the vertex that can be reached
      * A->B->C = A->C
      * Starting from v1 as the midpoint and v2, v3…
    - Runtime O(n^3) = n^2 at each vertex, and there are n vertex
      * Adjacency matrix O(1)
      * O(n(n+m))
  + Topological sorting = labeling each vertex
    - Directed acyclic graph= digraph with on directed cycle
    - In degree 0 = first vertex
    - Reverse completion = second penny of DFS
    - Runtime O(n+m)
    - BFS backwards
* Weighted graph
  + Shortest path: subpath of shortest path, tree of shortest path from start to all other vertex
  + Dijkstra ~ fern
    - edge relaxation
    - adding/updating the edges at each vertex
    - rely heavily on heap
    - runtime is O(m\*logn)
    - pick which vertex to go next = the edge with lowest value
    - start at end vertex and follow parent pointer in shortest path tree
    - nonnegative edges, undirected, connected
  + bellman ford = works with negative values edges but must be directed and not a cycle
    - runtime is O(nm)
    - single source shortest path = shortest path from any to any

Week 12

* weighted graph
  + all pairs shortest path
    - Dijkstra = O((n+m)logn) = adjacency list
    - Bellman = O(n^2 m)
    - Floyd-warshall/Dynamic prog = O(n^3)
    - Min{D[i,j], D[i,k] + D[k,j]}
      * D uses from K = vertex 1, 2, 3…, where D is the intermediate vertex taken from the current vertex it is at
      * i=x
      * j=y
* spanning tree = tree that includes every vertex
  + cost of spanning = sum of weight of all edge that includes every vertex
  + minimum spanning tree = smallest sum of weight of all edge that includes every vertex
  + simple path
    - prim jarnik algorithm
      * runtime O((n+m)logn)
* AI
  + Neural nets=output, training, learning, feedback, success
  + Symbolic ai = represent problem as collection of logic statement
  + Minimax search- chess = start from bottom
    - Recursive compute search
    - Min node- min of minimax values of children
    - Max node- max of minimax values of children
    - Heuristic evaluation to see who is winning
      * Admissible=never overestimate true value
    - Not DFS cuz don’t know how deep to go
      * Runtime O(n) n=number of nodes expanded
      * Space O(d) d=depth of search
    - Not BFS cuz run out of memory
      * Runtime O(n)
      * Space O(b^d) b=branching factor
    - Staged DFS -> increase depth one by one
      * Runtime O(n)
      * Space (d)
  + Backtracking search
    - Same as DFS
  + Best-first
    - Greedy algorithm
    - DFS but pick path that gets you closest to goal first = smallest edge amount
    - O(Nlogn)
    - Most likely to find a solution
  + A\*
    - Greedy algorithm
    - DFS but shortest path = smallest weight
    - Memory bad

Week 14

* Order picking
  + Bn= bins
  + Tn = time for picking
  + Wm = workers
  + Km = cutoffs = bins
  + Bottom up/memorize
  + Time warp= time in performance foe each note in score
  + Analyze human performance and figure time that corresponds to each notes/event in projected version
  + Dynamic programming

Week 15

Big companies such as Facebook do have an obligation to design algorithms that take into account the good of society. Although trying to maximize their profit is a good goal to have, even more so in the legal way, the end does not always justify the means. In other words, a positive outcome does not always translates into a good thing if the methods used were dishonest or harmful to others. In this case, if big companies misuse and exploit their power and responsibility. I believe that spreading false and provocative or hateful items, news, and comments are morally wrong. Therefore, the federal government should, to an extent, intervene by passing restrictions on social media companies or laws to make them responsible for their action and effect on society. However, this would be impossible if the federal government is corrupted. On the other hand, if a federal government were to intervene, then I would suggest

Week 16