CS2040S

AY24/25 Sem 2

by ngmh

Big-O

- $T(n)$ = running time on input with size n
- Upper Bound: $T(n) = O(f(n))$ if T grows no faster than f, i.e. $\exists\, c > 0, n_0 > 0$ such that $\forall\, n > n_0, T(n) \leq cf(n)$
- Lower Bound: $T(n) = \Omega(f(n))$ if T grows no slower than f, i.e. $\exists\, c > 0, n_0 > 0$ such that $\forall\, n > n_0, T(n) \geq cf(n)$
- Tight Bound: $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n)) = \Omega(f(n))$
- $log(n!) = \Theta(n\ log\ n)$ considering $log(n/2)$ and $log(n)$
- String append is $O(n)$ in Java
- Master Theorem: For $T(n) = aT(n/b) + cn^k$, $T(1) = c$, compare a and b^k, $<: \Theta(n^k)$, $=: \Theta(n^k\ log\ n)$, $>: \Theta(n^{log_b a})$
- $T(n) = 2T(n/2) + O(n\ log\ n) = O(n\ log^2\ n)$, $T(n) = 2T(n/4) + O(1) = O(\sqrt{n})$
- Fibonacci: $O(\phi^n)$, proof by induction, $T(n) = T(n-1) + T(n-2) + O(1), T(i) = F(i) - 1$
- $log(n) < \sqrt{n}, log^k(n) < n^m, n^{log(n)} < 2^n$
- Min-Queue (T7Q6): $O(1)$ enq, deq, min

Binary Search

- Precondition: True when function begins, important for it to work correctly, e.g. array has size n and is sorted
- Postcondition: True when function ends, useful to show computation was done correctly, e.g. if element is in array, $A[begin] = key$
- Invariant: Relationship between variables which is always true, e.g. $A[begin] \leq key \leq A[end]$ and $end - begin \leq n/2^k$ for iteration k

```
int search(A, key, n)
    begin = 0; end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2
        if key <= A[mid]: end = mid
        else: begin = mid+1
    return (A[begin]==key) ? begin : -1
```

Peak Finding

- Find local maximum $A[i-1] \leq A[i] \geq A[i+1]$
- Invariant: There exists a peak in $[begin, end]$ which is also a peak in $[0, n-1]$

```
FindPeak(A, n)
    if A[n/2] is a peak then return n/2
    else if A[n/2+1] > A[n/2] then
        Search for peak in right half
    else if A[n/2-1] > A[n/2] then
        Search for peak in left half
```

- Steep Peak: $A[i-1] < A[i] > A[i+1]$
- Fails on case where both sides are equal, degenerates to $O(n)$ when recursing on both halves
- Slow 2D Peak Finding: Find global max in each columnn, find peak in array of maximums, $O(mn + log(m))$
- Fast 2D Peak Finding: Find peak in array of peaks with lazy evaluation of columns, recurse on half by comparing max of adjacent columns, $O(n\ log\ m)$

Sorting

Bubble Sort

- Loop through array n times, swapping adjacent elements that are out of order
- Best Case: Sorted, $O(n)$, Average / Worst Case: $O(n^2)$
- Invariant: After iteration j, the biggest j items are sorted in the last j positions of the array

Selection Sort

- Repeatedly find minimum element and add to sorted prefix
- Always $O(n^2)$
- Invariant: After iteration j, the smallest j items are sorted in the first j positions of the array

Insertion Sort

- Loop from front, bubbling current element downwards into correct place in prefix
- Best Case: Sorted, $O(n)$, Average / Worst Case: $O(n^2)$
- Invariant: After iteration j, the first j items in the array are sorted

Merge Sort

- Divide and conquer, sort both halves of array, then merge back together
- $T(n) = 2T(n/2) + cn = O(n\ log\ n)$
- $log\ n$ levels, each with n elements total

Quick Sort

- Unlike mergesort, partition array by a pivot before recursing
- Average Case: $O(n\ log\ n)$, Worst Case: $O(n^2)$
- Not In-Place Partition: Create new array, if current element is lower than pivot add to prefix, else add to suffix using two pointers
- In-Place Partition: Move low pointer until it is larger than pivot, move high pointer until it is not larger than pivot, then swap until both pointer cross
- Invariants: $A[high] > pivot$ after each loop, and for all $i \geq high, A[i] > pivot$ and for all $1 < j < low, A[j] < pivot$

```
swap(A[1], A[pIndex])
while (low < high)
    while (A[low] <= p) and (low < high) low++
    while (A[high] > p) and (low < high) high--
    if (low < high) swap(A[low], A[high])
swap(A[l], A[low-1])
```

- Two Pass 3-Way Partition: Regular partition, packing with duplicates after
- One Pass 3-Way Partition: Maintain four regions of array ($< pivot, = pivot, in\ progress, > pivot$)
- Compare $A[i]$ and $pivot$, $<$: swap with start of $= pivot$, $==$: increment $= pivot$ end, $>$: swap with start of $> pivot$

Paranoid QuickSort

- Partition until certain factor is met, e.g. $9/10$
- Probability of good pivot is $8/10$, so $E[Partitions] = 1/p = 10/8 < 2$, so by expectation we only have to partition twice
- $E[T(n)] = E[T(k)] + E[T(n-k)] + E[Partitions](n) \leq E[T(k)] + E[T(n-k)] + 2n = O(n\ log\ n)$
- $T(n) = T(n/10) + T(9n/10) + O(n) = O(n\ log\ n)$

Properties

- MergeSort and QuickSort are not in-place
- Stability: Preserving initial order of equal elements
- SelectionSort and QuickSort are not stable due to swaps
- Swap first and last element to differentiate between Insertion and Bubble Complexity

Quick Select

- Select k^{th} smallest element in unsorted array
- Randomly partition (with paranoia if wanted), recursing only on the correct half, $O(n)$ expected
- $E[T(n)] \leq E[T(9n/10)] + E[Partitions](n) \leq E[T(9n/10)] + 2n \leq O(n)$

Trees

- Binary Search Tree: $left\ keys < key < right\ keys$
- Height: 0 at leaf, increases by 1 for each parent after taking maximum of children
- Minimum / Maximum: Keep traversing left / right until leaf
- Search / Insertion: Keep recursing on correct child until leaf, inserting if needed
- In-Order: LSR, Pre-Order: SLR, Post-Order: LRS, Level-Order: Increasing distance from root, left to right
- Successor: Search for key, if $result > key$, return $result$, else return $successor(result)$
- $successor(key)$: If right tree exists, take its min. If not, recurse upwards until we are in a left subtree, then return parent (so that $parent > key$
- 1-Child Deletion: Remove v, connect $child(v)$ to $parent(v)$
- 2-Child Deletion: Let $x = successor(v)$. Replace v with x, then replace x with its right child, works as x cannot have a left child
- Order Statistics: Rank and Select, use subtree size information to determine where to recurse, like quick select
- Rank is $left.w + 1$, if selecting right $right.select(k - rank)$, if node is right child repeatedly add $par.left.w + 1$
- Counting Inversions: For each element, use order statistics to get inversion count of it as $tree\ size - rank(element)$
- $(a, b)-$tree: $2 \leq a \leq (b+1)/2$, $O(log\ n)$ operations, $[a, b]$ number of children
- Virus Problem: Find minimum possible number of days for every node to be visited, each day a node can visit any adjacent node, binary search and partition by each edge along path between 2 sources, solve single source case
- Tries: Store multiple strings in a tree, with letters as nodes, words are paths from root to leaf, use special mark for terminal character

AVL Trees

- Balanced: $h = O(log\ n)$
- Perfectly Balanced: For any node both subtrees have same height
- Height Balanced: $|v.left.h - v.right.h| \leq 1$, has at most $h < 2log(n)$, and $n > 2^{h/2}$
- Weight Balanced: $v.left.w, v.right.w \leq \alpha \cdot v.w$
- Scapegoat Tree: $2/3$ weight balanced, rebuild tree at highest unbalanced node if necessary
- Maximally Imbalanced: Tree with minimum possible nodes given a total height, construct recursively
- Left / Right Heavy: Child with greater height
- Rotations: Direction is where root of subtree goes, requires child opposite of rotation direction
- If middle heavy, rotate child away from middle before rotating root
  - v Left Heavy: Check Left child:
    - Balanced: Right-Rotate(v)
    - Left Heavy: Right-Rotate(v)
    - Right Heavy: Left-Rotate(v.left), Right-Rotate(v)
  - v Right Heavy, Check Right child:
    - Balanced: Left-Rotate(v)
    - Left Heavy: Right-Rotate(v.right), Left-Rotate(v)
    - Right Heavy: Left-Rotate(v)
- Insertion: Only need to fix lowest unbalanced node with at most two rotations, since insertion increases height and rotation reduces height
- Deletion: After deletion, for every ancestor of deleted node, rotate if not height-balanced, $O(log\ n)$, as rotation and deletion both reduce height
- Modification: Only store difference in height from parent

Hashing

- Direct Access Table: Table indexed by keys, but faces size problem
- Hash Function: Map large set of possible keys to smaller set of actual keys / buckets
- Collision: Two distinct keys map to same bucket
- Chaining: Each bucket is a linked list, insertion appends to front
- Insertion is $O(1)$, Search is worst case $O(n)$
- Simple Uniform Hashing Assumption: Each key is equally likely to map to every bucket and is mapped independently
- $load(table) = n/m$ where n is items and m is buckets, expected search time is $O(1) + n/m$, when $m = \Omega(n)$ e.g. $m = 2n$, expected search time is $O(1)$
- Expected maximum cost of inserting n elements is $O(log\ n)$ or $\Theta(log\ n / log\ log\ n)$
- In Java, keys should be immutable, and implement $int\ hashCode()$
- Must return same value if object hasn't changed, and two equal objects must return the same hashCode, uses $hash()$ to ensure that similar hashCodes have a bounded number of collisions
- Open Addressing: Insert items into table directly
- If hash position is occupied, increment and continue probing, need not be linear
- Tombstoning: Use marker for deleted elements, to indicate for probing to continue, replace on insertion, if too many

tombstones, rehash the table
- Insertion Resizing: Double table size when full, total cost is $O(n)$ for resizing with $O(1)$ insertion on average
- Deletion Resizing: Halve table size when quarter full
- Amortised Analysis: Inserting n elements, most will have cost $O(1)$, some have cost $O(k)$, in total will be $O(n)$ for n operations, so amortised is $O(1)$, can extend idea to deletion and resizing as well
- Linear probing results in clusters, if quarter full has clusters of size $\Omega(log\ n)$, but is still faster in practice
- Merkle Tree: Each node is the hash of the concatenation of the child nodes
- Cuckoo Hashing: Two hash tables with own keys, kick existing key to other table on collision, but fails if infinite loop, $O(1)$ expected, $O(log\ n)$ expected worst case

Binary Heap

- $O(1)$ findMax, $O(log\ n)$ extractMax, $O(log\ n)$ insertion and deletion
- Heap Ordering Property: priority of child < priority of parent
- Shape Property: The heap is a complete binary tree except the last level which is filled from left to right
- Current i, Left Child $2i$, Right Child $2i+1$, Root 1, Size of Heap 0, Parent $\lfloor i/2 \rfloor$
- Insertion: Insert at $a[a[0]+1]$, bubble up if necessary
- extractMax: Return $a[1]$
- Swap operation: Need hashtable from ID to node indices, and hashtable / array from node indices to ID
- increaseKey: Increase then bubble up
- decreaseKey: Decrease then bubble down, picking largest child
- extractMax: Cannot simply delete then replace with child as it violates shape property, instead swap root with last element, remove it, then bubble down root
- Heapify: Build heap from array in $O(n)$, repeatedly bubble down from lowest non-leaf to root, going upwards by traversing backwards through main array
- Proof of complexity is by amortisation, each node has 2 cost at the start so heapfiy is free, or solve $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$
- Heapsort: Repeatedly use extractMax, and store it by swapping with last element, in-place and $O(n\ log\ n)$, but is not stable

Graphs

- Collection of nodes and edges
- Each edge is unique and connects two nodes
- Path: Set of edges connecting two nodes, each node visited at most once
- Connected: There is a path between ever pair of nodes
- Cycle: "Path" where first and last node are the same
- Tree: Connected graph with no cycles
- Forest: Graph with no cycles
- Degree: Number of adjacent edges, for graph maximum degree of all nodes
- Diameter: Maximum distance between two nodes which is a shortest path
- Star: One central node, all edges connect it to outer nodes
- Clique: Complete graph
- Line: Nodes connected in one line, Cycle: Line which forms a loop
- Bipartite: Nodes divided into two sets with no edges connecting two nodes in the same set
- Adjacency List: Each element in array stores linked list of adjacent nodes to node at index
- Adjacency Matrix: Matrix where $A[i][j]$ indicates edge between i and j, A^x is length x paths
- Edge List: List of edges

Searching a Graph

Breadth First Search

- Explore level by level, increasing distance from source, move frontier / current level after exploring all nodes in frontier
- Finds shortest paths in unweighted graphs
- Implement with queue, $O(V+E)$

Depth First Search

- Follow path until stuck, backtrack until find new edge, recursively explore
- Implement with stack, $O(V+E)$

Directed Acyclic Graphs

- Topological Sort: Sequential total ordering of nodes, edges from original graph only point "forwards"
- Directed Acyclic Graphs have a topological ordering
- Use post-order DFS, prepend to output after visiting children (alternatively append then reverse result), $O(V+E)$
- Toposort might not be unique
- Alternative algorithm: Take set of edges with no incoming edges, add them to output, remove all edges adjacent to nodes in this set, recalculate set and repeat

Strongly Connected Components

- For every pair of nodes in a component, there is a path from each of them to the other
- Articulation Point / Bridge: Removal disconnects graph
- Tarjan's: Mark each node with time we visited it, and lowest time we can reach from its neighbours
- Low time is the minimum of its own time and low time of children that we just visited (cannot be visited by someone else)
- Only update based on nodes whose low times are not set, still in the recursion stack
- Cycle Detection: If there is a node with $low\ time < time$
- Articulation Point: Node with $low\ time \geq time$

Dijkstra

- Single Source Shortest Path on Weighted graph
- Triangle Inequality: $dist(S,C) \leq dist(S,A) + dist(A,C)$
- Maintain distance estimates from source to every node
- Relax: Lower distance estimate if there is a shorter path from source using intermediate node
- Maintain frontier, and always visit node closest to and outside frontier with smallest distance estimate
- All nodes within the frontier are visited, and have correct distance estimates
- When visiting a node, relax its neighbours
- Use a priority queue to sort nodes by distance estimates, and use decreaseKey to update their estimates
- Each node is removed from the PQ at most once, and decreaseKey is called on each node at most indegree times
- Total is $V\ O(log(V)) + E\ O(log(V)) = O(E\ log(V))$
- Or perform BFS, where traversal is slightly different due to edge weights, replace edge weight w with w new nodes
- Does not work with negative weights, as invariant is violated

Bellman-Ford

- For $V-1$ iterations, relax each edge in the graph, $O(VE)$
- After i iterations, nodes i hops away from the source in the shortest path graph have correct distance estimates
- If no distances change after a round of relaxation, we are done
- If distances change on the V^{th} iteration, we know there is a negative cycle, as it should terminate after $V-1$ iterations
- On DAG, can toposort first then traverse in order to get $O(V+E)$ complexity

Graph Modification

- Multisource: Connect all sources to a supersource with an edge of zero weight
- Path with k hops: Create k copies of graph, jump between them when taking an edge
- Product of edges: Use logarithm before applying Dijkstra's
- Maximum weight edge on any path with maximum total cost: Dijkstra from source and end, check each edge, $O(E\ log\ V)$, alternatively, binary search on maximum edge weight and try reaching with limited set of edges
- LCA: $O(V\ log\ V)$ precomputation, $O(log^2\ V)$ query

Union Find Disjoint Set

- Union: Connect two objects, Find: Check if two objects are connected
- Version 1: When connecting, set parents of all nodes in one set to the parent of other set, $O(1)$ find, $O(n)$ union
- Version 2: Use parent pointers, when connecting, traverse up to root of both nodes then set parent of one to the other, $O(n)$ for both
- Weighted Union: Link smaller tree to larger tree, $O(log\ n)$ for both as height of tree of size n is at most $log\ n$
- Path Compression: Set parent of each traversed node to the root, $O(log\ n)$ for both, proof ommitted
- Weighted Union and Path Compression: $\alpha(m,n)$ for both, sequence of m operations on n objects takes $O(n + m\alpha(m,n))$ time where α is inverse Ackermann

Minimum Spanning Trees

- Spanning Tree: Acyclic subset of edges connecting all nodes, MST: Minimum weight spanning tree
- Property 1: No cycles
- Property 2: If you cut an MST, both pieces are MSTs
- Cycle Property: For every cycle, the maximum weight edge is not in the MST (assuming distinct weights)
- Cut: Partition of nodes into two disjoint subsets
- An edge crosses a cut if both endpoints are in different sets
- Cut Property: For every partition of nodes, the minimum weight edge across the cut is in the MST, doesn't say anything about maximum edge
- For every vertex, the minimum outgoing edge is always part of the MST, nothing about maximum edge
- Generic Algorithm: Red: If C is a cycle with no red arcs, then colour the max-weight edge red, Blue: If D is a cut with no blue arcs, colour the min-weight edge blue, repeat then the blue edges form a MST
- Prim's Algorithm: Start with a single node, identify cut and find minimum weight edge on cut, then add it to visited set and repeat
- Use min PQ to keep track, use node if not visited before, add all neighbours to PQ with priority of edge weight, can use decreaseKey, $O(E\ log\ V)$
- Kruskal's: Sort edges by increasing weight, when considering an edge, take it and colour blue if no connected, if not colour red if endpoints are connected and in the same blue tree, find using UFDS, $O(E\ log\ E) = O(E\ log\ V)$ since $E = O(V^2)$
- If all same weight, use math and BFS/DFS
- If weights have fixed range, use counting sort, $O(\alpha E)$
- Fails on Directed MST
- Directed MST on DAG with one root: For every node except root, add minimum weight incoming edge

Dynamic Programming

- Optimal Sub-structure: Optimal solution can be constructed from optimal solutions to smaller sub-problems
- Overlapping sub-problems: Smaller sub-problem is used to solve multiple different bigger problems
- Toposort DAG of states, solve in reverse order / bottom-up
- LIS: $O(n^3)$ with DAG method, run $O(n^2)$ longest path algorithm n times, speedup to $O(n^2)$ total with memoisation, n subproblems with subproblem i taking $O(i)$
- Prize Collecting: Longest path on graph, limited to k edges
- Transform into DAG with k copies of each node, solve longest path, kV nodes, kE edges, takes $O(kVE)$
- Using dynamic programming, speedup to $O(kV^2)$, kV subproblems each costing $O(|v.nbrList|)$, or $O(kE)$ since there are k rows and E entries in a row
- Vertex Cover: Output set of nodes where every edge is adjacent to at least one node in this set, NP-complete on general graphs
- On tree, state is take or don't take a node, $2V$ sub-problems, $O(V)$ time to solve all sub-problems
- All Pairs Shortest Path: $O(VE\ log\ V)$ with Dijkstra's
- Floyd-Warshall: At each iteration, use the set of first k nodes to update for all i and j, relaxing through k, $O(V^3)$
- Actual path: `prev[i][j] = prev[k][j]`, backtrack
- Knapsack: Given items with weight and value, maximise total value given a total weight capacity, $O(nL)$ sub-problems, each taking $O(1)$