CSOR 4231 Note (lec 1-8)

$\Theta\big(g(n)\big) = \{f(n) \mid \exists c_1, c_2 > 0,\ n_0 \colon \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)\}$, $c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$

$O\big(g(n)\big) = \{f(n) \mid \exists c > 0, n_0 \colon\ \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$

$\Omega\big(g(n)\big) = \{f(n) \mid \exists c > 0, n_0 \colon\ \forall n \geq n_0, cg(n) \leq f(n)\}$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$

$o\big(g(n)\big) = \{f(n) \mid \forall c > 0 \colon \exists n_0\ s.t. \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$\omega\big(g(n)\big) = \{f(n) \mid \forall c > 0 \colon \exists n_0\ s.t. \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

$\log(n) = o(n^c)$ for all $c > 0$. [$\lim_{n\to\infty} \frac{\ln n}{n^c} = \lim \frac{1/n}{cn^{c-1}} = \lim \frac{1}{cn^c} = 0$ by L]

MERGESORT: $\mathrm{T(n)} = 2T\left(\frac{n}{2}\right) + O(n) = \mathrm{O(n \log n)}$

Master's Method to solve recurrences: $\mathrm{T(n)} = \mathrm{aT}\left(\frac{\mathrm{n}}{\mathrm{b}}\right) + \mathrm{f(n)}$, for some constant $\epsilon > 0$

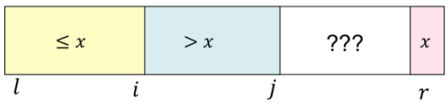
- $\mathrm{f(n)} = \mathrm{O}\big(\mathrm{n}^{\log_b a - \epsilon}\big) \to T(n) = \Theta(\mathrm{n}^{\log_b a})$
- $\mathrm{f(n)} = \Theta\big(\mathrm{n}^{\log_b a}\big) \to T(n) = \Theta(\mathrm{n}^{\log_b a} \log n)$
  $\mathrm{f(n)} = \Theta\big(\mathrm{n}^{\log_b a} \log^k n\big) \to T(n) = \Theta(\mathrm{n}^{\log_b a} \log^{k+1} n)$
- $\mathrm{f(n)} = \Omega\big(\mathrm{n}^{\log_b a + \epsilon}\big) \wedge a\, f\left(\frac{n}{b}\right) \leq cf(n),\ c < 1 \to T(n) = \Theta(\mathrm{f(n)})$

Priority Queue: insert/check_max/extract_max
Heap: binary tree with shape property (all levels are full, except last level partially filled from left) and order property (key(i) ≤ key(parent(i)) for max heap)

- HEAPIFY(A, i): fix order property violation between a node and its children, $\mathrm{O}(\log n)$
  A[j]: child with max key; if A[j] > A[i]: switch A[i], A[j] and call HEAPIFY(A, j)
- EXTRACT_MAX: replace root with last element in array, and HEAPIFY(A, root), $\mathrm{O}(\log n)$
- BUILD_HEAP: for $\mathrm{i} = \mathrm{n}/2$ to 1, HEAPIFY(A, i)
  runtime: $\sum_{\mathrm{j}=1}^{\mathrm{h}} j \cdot 2^{h-j} = 2^{\mathrm{h}} \sum_{\mathrm{j}=1}^{\mathrm{h}} \frac{j}{2^j} \leq 2^{\mathrm{h}} \sum_{\mathrm{j}=1}^{\infty} \frac{j}{2^j} = 2^{h+1} = O(n)$

Heap Sort: BUILD_HEAP, then EXTRACT_MAX for n times. $\mathrm{O(n \log n)}$
Top k largest elements in online stream: use MIN_HEAP to store the k largest elements so far, replace root with any larger element and run HEAPIFY.

PARTITION(A, l, r): in-place partition the subarray A[l:r] to three sections: <= A[r], >A[r], return new index of pivot A[r], $\Theta(\mathrm{n})$



SELECTION(A, l, r, i): select the i^{th} element from A[l:r]
If l == r: return A[l]
q = PARTITION(A, l, r), k = q – l + 1 (k elements are less than or equal to A[q])
if i == k: return A[q]; if i < k: return SELECT(A, l, q – 1, i); if l > k: return SELECT(A, q + 1, r, i – k)
Worst Case: $\mathrm{T(n)} = T(n-1) + \Theta(n)$, Best Case: $\mathrm{T(n)} = \mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + \Theta(n)$, Typical Case: $\Theta(\mathrm{n})$

Pivot-selecting strategy: Divide array into $\lceil n/5 \rceil$ groups of 5, L = medians from each group, pivot = SELECT(L)
Half of $\lceil n/5 \rceil$ groups has median ≤ pivot, and 3 numbers are no greater than median. Vice Versa.
SELECTION with strategy runtime: $\mathrm{T(n)} = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + \mathrm{T}\left(\frac{7\mathrm{n}}{10} + 2\right) + \Theta(n)$, $\mathrm{T(n)} = \Theta(\mathrm{n})$ prove by substitution

Harmonic Series: $\sum_{i=1}^{n} 1/i = H_n \approx \ln n$
QUICKSORT: PARTITION the array (similar to in SELECTION), and quick-sort the two sub-array
runtime: $T(n) = T(|left|) + T(|right|) + \Theta(n)$. Worst case: $O(n^2)$. Best case: $O(n \log n)$
RAND-QUICKSORT: for each partition, randomly choose an element as pivot

- two elements z_i and z_j are only compared if i or j is chosen as pivot for the set $\{z_i, z_{i+1}, \dots, z_j\}$
- expected number of comparisons: $\sum_{i=1}^{n-1} \sum_{j=1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$

Markov bound: for positive random variable X, $\Pr[X > b] \leq E[X]/b$
Quicksort with duplicated items: three-way partition($<, =, >$) and recurse on non-equal subarrays.

RAND-SELECT: randomly select k, split array to $|k|$ and $|n - k - 1|$ for $0 \leq k \leq n - 1$, recurse on larger part.
$T(n) \leq \sum_{k=0}^{n-1} \frac{1}{n} T(max(k, n-k-1)) + \Theta(n) \leq \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} T(k) + \Theta(n)$, then prove to be $\Theta(n)$ by substitution

Comparison sorts: only allow comparison, must make at least $\Omega(n \log n)$ comparisons in worst case.
Proof: construct a decision tree for sorting, number of leaves = all possible permutations (n!), then number of comparisons needed = length of a path = height of tree. Note $\log n! \geq \log(n/2)^{n/2}$.
log(# of possible answers) is a lower bound for any problem.

SIMPLE_COUNTING_SORT: use a count array to save count of elements, and output them in order.
restricted domain: $D = \{1, \dots, k\}$, runtime: $O(n + k)$
BUCKET_SORT: construct a "bucket" for each key, and dump buckets from small to large. $\Theta(k + n|record|)$

COUNTING_SORT: count prefix besides simple occurrences
for j from n to 1: { OUTPUT[PREFIX[INPUT[j].key]] = INPUT[j]; j--; }, $\Theta(k + n|record|)$

Counting sort and Bucket sort are stable: preserve order among equal input elements

RADIX_SORT: for d-digit number, each digit takes k possible values. Runtime: $\Theta(d(n + k))$
sort the numbers from least significant digit to most significant digit. Correctness proven by induction.
RADIX_SORT for b-bit integers: divide all bits into blocks of r bits, then $d = b/r$, $k = 2^r$
When $n = 2^r$, runtime$= \Theta(\frac{bn}{\log n})$

CSOR 4231 Note (lec 9-14)

Dynamic Programming:

Bottom-Up version: iterative, solution tabulated from small to large

Top-down version: before recursive call, check whether solution was computed already

KNAPSACK: n valued items with weight capacity W, maximize total values

$W[b,i] = \max(W[b,i-1], W[b-w_i, i-1] + v_i)$ for each capacity b and first i items, runtime $O(nW)$

LONGEST-COMMON-SUBSEQUENCE: longest common subsequence of two sequences, gap allowed

$C[i,j] = C[i-1,j-1] + 1$ if $x_i = y_j$, $\max(C[i,j-1], C[i-1,j])$ if $x_i \neq y_j$, runtime $O(nm)$

Recovering the solution: Record the choice at each step

SPARSE-LCS: improvement from LCS if few symbol repetitions. (only process matches since match -> increase)

runtime: $O((m+n+p)\log(m+n))$ where $p =$ number of matches

(1) Compute all matching indices (2) sort matching indices in increasing i and decreasing j (3) S = empty set (4) For each (i, j) in M, if j not in S -> {Insert j to S and delete j's successor in S}

MATRIX_CHAIN_MULTIPLICATION: grouping matrix chain multiplication to minimize total cost

$M[i,j] = \min_{i \le k < j}(M[i,k] + M[k+1,j] + p_{i-1}p_k p_j)$, record the optimal k to recover solution, runtime $\Omega(n^3)$

Greedy Algorithms: 1) There exists some optimal solution containing first greedy choice (by exchange argument) 2) Given the first greedy choice, the optimal solution for the remainder of problem leads to overall optimal solution

ACTIVITY SELECTION: choose the activity that finishes first

JOB SCHEDULING: each job has a processing time and deadline, earliest deadline first.

HUFFMAN'S ALGORITHM: assign codes to a set C of characters that minimizes $\sum_c frequency[c] \cdot |code(c)|$

(prefix-free property) no codeword is a prefix of another word; if internal node has one child -> suboptimal

construct code tree: merge sibling characters and reduce problem size

Greedy Choice: choose 2 lowest-frequency characters (there exists an optimal tree where x, y are neighbors)

Tree construction: when merging two chars x and y, add a node "xy" to tree, with edge 0, 1 pointing to x and y

Runtime: use min-priority queue to store nodes and frequencies, $O(n \log n)$

FRACTIONAL KNAPSACK: pack best-value (v_i/w_i) item to backpack until full (last item possibly fractionally)

Amortized Analysis: average cost per operation

Aggregate Analysis: Compute total time of all operations

Banker's Method: each operation has real cost c_i, amortized cost $\widehat{c_i}$, amortized costs are valid if $\forall l \ \sum_{i \in l} \widehat{c_i} \geq \sum_{i \in l} c_i$; if $\sum_{i \in l} \widehat{c_i} \leq X$, total runtime is at most X

Potential Function: Let D_i be the state of system after operation i. Choose potential function such that $\Phi(D_0) = 0$ and $\Phi(D_n) \geq 0$. The amortized cost of each operation is $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$

MULTIPOP: choose potential function to be # of elements in the stack

BINARY_COUNTER: independent of number of bits in the counter

(banker) pay \$2 for increment, and future flipping back to 0;

(potential function) let f_{01} and f_{10} be flips from 0 to 1, and flips from 1 to 0, $\Phi(D_i)$ is the number of 1, then $c_i = f_{01} + f_{10}$, $\Phi(D_i) - \Phi(D_{i-1}) = f_{01} - f_{10}$, and f_{01} is always 1.

TABLE_INSERT: double capacity of table and copy all elements when table is full

(potential function) $\Phi(T_i) = 2num(T_i) - size(T_i)$

Graph:

Path in a graph: a sequence of edges with no repeated node

Walk in a graph: a sequence of vertices/edges of graph (can be repeated)

Simple graph: no self-loops or multi-edges (number of edges is between 0 and n^2)

Forest/Tree: undirected acyclic graph, DAG: directed acyclic graph

	Space	Edge(i, j)?	List edge(i)
List of edges	e	e	e
Adjacency Matrix	n^2	1	n
Adjacency List	n+e	degree(i)	degree(i)

CSOR 4231 Note (lec 15-24)

DEPTH-FIRST-SEARCH (DFS): Implemented with a stack (Last-In-First-Out), runtime: $O(n+e)$
Connected Components: set of mutually reachable (exists connection path) vertices in undirected graph
Can compute with DFS (for each search mark the number of CC it's on) -> produces a spanning forest
Graph acyclic ↔ no other edge besides spanning forest
Any tree with n vertices has exactly $n-1$ edges; DFS to check acyclic (return when meet visited vertex) $O(n)$
DFS with start/finish time: increment time when discover/finish a node, initial time = 0
Pre-order: discovery time of nodes, Post-order: finish time of nodes, interval of nodes: disjoint or nested

Edge classification (u,v): tree edge (in search tree), forward/back edge (u is ancestor/descendant of v), cross-edge (unrelated); in undirected graph: only tree edge and forward/back edges, in directed graph: cross edge (u,v) iff $d[v] < f[v] < d[u] < f[u]$
White Path Theorem: u is ancestor of v ↔ at time d[u], there's a white path from u to v

Directed Acyclic Graph (DAG): Directed graph with no cycles
Topological Sort: linear ordering of nodes so all edges go left to right. (Graph Acyclic ↔ Topological sort)
Reversed finishing time of DFS is a topological sort; runtime $O(n+e)$

Strongly Connected Components (SCC): all nodes mutually reachable
Testing for Strongly connected: pick any source node, Search(G, s), then search again on reverse graph G_r
Cycle is contained in some SCC, structure of DiGraph can be shrink to DAG of SCC's.
Algorithm: (1) run DFS to compute finish time (2) In reverse graph G_r, call DFS in decreasing finish time (3) grouping of DFS trees in second DFS calls -> SCC's. (show that u, v mutually reachable = in same DFS tree in G_r)

Minimum Spanning Tree: [input]undirected graph with weighted edge [output]minimum weight spanning tree
Exchange argument: out-of-tree edge (u,v) has weight greater than max weight edge on u->v path in tree
Partition Theorem: every MST contains some min-weight edge across vertex partition, and every min-weight edge across partition belongs to some MST
Distinct edge weights -> unique MST
PRIM'S ALGORITHM: choose edge from R to N\R with smallest weight (implement with priority queue of nodes, priority is weight of edge to R), runtime: n EXTRACT_MIN and e DECREASE_KEY operations. (heap: $O(e \log n)$, Fibonacci heap: $O(e + n\log n)$)
KRUSKAL'S ALGORITHM: process edge in non-decreasing order of weight: select an edge if it joins different connected components defined by selected edges
UNION-FIND data structure: (1) Linked List (with pointer to the head) amortized runtime: $O(e + nlogn)$
(2) Forest Data: find (find root of tree), union (hang one tree under another); $O(e + \alpha(n))$
-union by rank: (rank r -> at least 2^r nodes) hang smaller rank set from larger
-path compression: when traverse the tree, make all nodes on path from root to x be direct children of x

DIJKSTRA'S ALGORITHM: find shortest path; [implementation] Priority queue with tentative distance. While queue is not empty {u = extract min, update v ∈ Adj[u] to *dist[v] = min(dist[v], dist[u] + w(u, v))*}; Runtime: n EXTRACT-MIN and e DECREASE-KEY

	EXTRACT-MIN	DECREASE-KEY
Array	O(n)	O(1)
Heap	O(log n)	O(log n)
Fibonacci Heap (amortized)	O(log n)	O(1)

General Weighted Graphs: no negative cycle -> well-defined distances -> shortest paths are simple

Non-circular recurrence for dynamic programming: sub-problem is solved when its answer is needed.
DP for shortest paths: process nodes in topological order: d[v] = min(d[v], d[u] + weight(u, v)). $O(n+e)$
DP on DAGs: compute value from parent (in topo. order); compute from children (in reverse topo. order)
Problem on directed graph: decompose to SCC, process in topo. order
LONGEST-PATH in DAG: (in reverse topo. order) $d[v_i] = \max\{0,\ w(v_i, v_{i+1}) \mid v_{i+1} \in Adj[v_i]\}$ time: $O(n+e)$

BELLMAN-FORD: for n-1 times { for each edge (u, v): { if d[v] > d[u] + w(u, v) -> d[v] = d[u] + w(u, v), p[v] = u } }; for each edge (u, v): if d[v] > d[u] + w[u, v] -> return False; return True. Runtime: $O(n \cdot e)$
Detect any negative cycles: (1) run Bellman-Ford on every SCC (2) add a source connects to all nodes, run BF
ALL-PAIR-SHORTEST-DISTANCE: use (2) to detect neg. cycles, and use computed distance as potentials to reduce to non-negative weight: {w'(u, v) = w(u, v) + d[u] – d[v]}, then run DIJKSTRA on all nodes. -> (path between x, y -> w'(p) = w(p) + d[x] – d[y])
FLOYD-WARSHALL algorithm: DP algorithm for all-pair shortest paths, good for dense graphs, runtime $O(n^3)$
Use adjacency matrix representation for graph, DP: compute d_{ij}^k for distance between i, j with first k nodes
$d_{ij}^k = w(i,j)$ if k = 0; $\min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$ else;

Edge flow constraints: (1) flow non-negativity (2) capacity constraints: f(u, v) <= c(u, v) (3) flow conservation constraints, for all non-source, non-sink vertices
FLOW-DECOMPOSITION THEOREM: every edge flow -> a set of s-t paths and a set of cycles.
Augmenting Path (rerouting): undirected path from s to t, such that forward edge not saturated, backward edge carries positive flow. (the augmented flow δ must be less than residual capacity $c(u,v) - f(u,v)$ in forward edges, and less than $f(u,v)$ in backward edges)
Residual network: residual capacity of edge (u, v) -> additional flow can be pushed through (u, v) -> $c(u,v) - f(u,v) \mid (u,v) \in E$, or $f(u,v) \mid (v,u) \in E$; residual capacity of path: min of residual capacities among edges
If g is a feasible flow in G_f, then $f+g$ is a feasible flow in G (add flow for forward edge, subtract for reverse)

FORD-FULKERSON ALGORITHM: keep finding augmenting path, and update the residual network
Integrality property: integer capacities -> integer max flow; every iteration increases total flow by at least 1
Finding an augmenting path ($O(e)$) -> total runtime $O(e \cdot |f^*|)$
EDMONDS-KARP ALGORITHM: use shortest augmenting path -> number of iterations $\leq ne$ -> $O(ne^2)$

s-t cut: partition nodes to S, T such that $s \in S \wedge t \in T$; capacity of cut: $\sum_{u \in S, v \in T} c[u,v]$; question: compute minimum cut.
[Theorem] min-cut(s, t) = max-flow(s, t): (1) maxflow <= mincut (any flow <= any cut): $|f| = f(S,T) - f(T,S)$
(2) mincut <= maxflow: let f be a max flow, let S be the set of nodes that s can reach in G_f, then no edge should be connecting S and T, forward edges must be saturated, backward edge must be empty
[Example] Node capacity: capacity are on nodes not edges -> split node to two connected by constrained edge
[Example] Maximum Bipartite Matching (set of disjoint edges): max cardinality matching -> reduce to max flow and solve with Ford-Fulkerson in $O(ne)$; since n outgoing edge from s, max flow is bounded by n

Linear Programming: maximize/minimize a (linear) objective subject to some linear constraints
[Max flow as LP] Maximize $\sum_{s,v \in E} f(s,v) - \sum_{v,s \in E} f(v,s)$. Subject to $\forall (u,v) 0 \leq f(u,v) \leq c[u,v]$; $\forall u \neq s,t$ $\sum f(v,u) - \sum f(u,w) = 0$
[Min Cost flow problem] Each edge has a cost, minimize cost given a flow demand. Minimize $\sum a(u,v) f(u,v)$, subject to $\sum f(u,v) - \sum f(v,u) = 0 \mid v \neq s,t$, $\sum f(s,v) - \sum f(v,s) = 0$, $0 \leq f(u,v) \leq c(u,v)$
[Shortest path]: min cost flow with all capacity = 1, demand = 1 (due to integrality property)
LP has 3 possible solution: infeasible, unbounded, finite optimum (can still be unbounded feasible solutions)
Linear Program feasible region: polyhedron bounded by constraints (optimal solution: a vertex)

P = problems solvable in polynomial time (quantitative notion of tractability)
Closed under +, x: poly(poly(n)) = poly(n) [Psuedo-polynomial]: polynomial under unary representation
Decision Problem: outputs yes/no; optimization problem: return max/min optimal value
Solving optimization problem immediately gives decision problem answer: optimization as hard as decision

Decision Problem=Languages over {0, 1}*: Input a binary string, output whether it's in L ;encode of yes instances
P: languages that can be decided in polynomial time (closed under union, intersection, complement, difference)
NP: languages that can be verified efficiently that $x \in L$ given a short certificate(witness/proof) y
$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^*, |y| \leq |x|^c, V(x,y) = 1\}$ (for language L, given a polynomial algorithm V and c)

Reductions: $L \leq_p L'$ polynomial time reduction from L to L'. Reduce decision problem A to decision problem B: match A's yes/no instances to B's yes/no instances (if B is P, then A is P; if A is not P, then B is not P; if B is NP, then A is NP; if A is not NP, then B is not NP)

NP-hard: L is NP-hard if every NP problem reduces to L; NP-complete: if both NP and NP-hard
If A is NP-hard and A $\leq_p$B, then B is NP-hard
CURCUIT-SAT is NP-COMPLETE (given a circuit, determine whether there's a Boolean assignment to let circuit output 1); To prove A is NP-hard/complete: CIRCUIT-SAT $\leq_p$ A
Other NP-COMPLETE problems: 3-SAT, max clique, 3-colorability, 0-1 knapsack, graph hamiltoniancity
Prove problem B is NP-COMPLETE: formulate B as decision problem (1) prove B is NP (2) find function f that maps instance of NP-hard A to B (3) Proves yes of A ↔ yes of B (4) prove f is P

3SAT: satisfaction of CNF with 3 literals in each clause
Independent Set: max independent set (pairwise non-adjacent nodes) in a graph
Vertex-Cover: subset of vertices that every edge has at least one endpoint in the subset
Set-Cover: given a set U, m subsets $S_1 \ldots S_m$ of U, if there exists at most k subsets whose union equal to U