I. Introduction, Chapter 1: The role of Algorithms in Computing

A. Definition of an algorithm and difference from a program
B. Algorithms as technology
   i. A good algorithm on a slow computer beats a bad one on a fast computer
   ii. Growth of functions
   iii. Polynomial time

II. Appendices A and B

A. Discrete Math review
B. Summation review

III. Chapter 2: Getting Started

A. Insertion sort
   i. Loop invariants
      a. Initialization
      b. Maintenance
      c. Termination
   ii. Pre and Post Conditions
   iii. Program correctness
   iv. Loop correctness
B. Analyzing Algorithms
   i. “Best case” analysis
   ii. “Worst case” analysis
   iii. “Average case” analysis
      a. Mean time as average
      b. Random input as average for a sort
C. Designing Algorithms
   i. Divide and Conquer - Merge Sort
   ii. Analyzing divide and conquer - recurrence equation

IV. Chapter 3: Growth of Functions

A. Asymptotic notation
   i. “Big O”
      \[ O(g(n)) \in \{ f(n) : \text{there exist positive constants } c_2 \text{ and } n_0 \text{ such that } f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} \]
   ii. “Little o” \( o(g(n)) \)
   iii. “Big Omega”
      \[ \Omega(g(n)) \in \{ f(n) : \text{there exist positive constants } c_1 \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \text{ for all } n \geq n_0 \} \]
   iv. “Little omega” \( \omega(g(n)) \)
   v. “Big theta”
      \[ \Theta(g(n)) \in \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} \]
   vi. If the same function \( f(n) \) is both \( O(g(n)) \) and \( \Omega(g(n)) \) then \( f(n) \in \Theta(g(n)) \)
B. Shortcuts
   i. Leading constants can be ignored, i.e. \( O(2n^5) \in O(n^5) \)
ii. All logs differ by a constant, i.e. $O(n \log n) \in O(n \log_2 n)$

V. Chapter 4: Divide and Conquer

A. Recursions
   i. Must be able to divide the problem into sub-problems that are identical to the original problem
   ii. Solutions to the sub-problem must contribute to the solution of the original problem

B. $T(n)$
   i. Basically, the solution is a proof by induction
   ii. Need base case (where do we stop?)
   iii. Need inductive hypothesis
   iv. Must show I.H. $\rightarrow$ next recursion

C. Solution by recursion tree

D. Solution by substitution
   i. $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for $c_1, c_2 > 0$ and $n \geq n_0$ as $n$ grows large
   ii. guess
   iii. show your guess is correct

E. Master Theorem

Material for Exam II starts here

VI. Chapter 6: Heap Sort

A. Heaps
   i. Definition of a Heap
   ii. Uses of a Heap
      a. Heap Sort
      b. Priority Queues

B. MAX-HEAPIFY($A, i$)
   i. Pre-condition: $A[i+1, i+2, \ldots, n]$ are roots of valid heaps
   ii. Time analysis: $O(\log_2 n)$
   iii. Post condition: $A[i, i+1, i+2, \ldots, n]$ are roots of valid heaps

C. BUILD-MAX-HEAP($A$)
   i. Pre-condition: $A[i, i+1, i+2, \ldots, n] \neq \emptyset$ and $A[i, i+1, i+2, \ldots, n]$ are comparable.
   ii. Time analysis: $O(\log_2 n)$
   iii. Post condition: $A$ is a valid heap
   iv. Why does the algorithm start at $A.heapsize/2$?

D. HEAPSORT($A$)
   i. Pre-condition: $A[i, i+1, i+2, \ldots, n] \neq \emptyset$ and $A[i, i+1, i+2, \ldots, n]$ are comparable.
   ii. Time analysis: $O(n \log_2 n)$
   iii. Post condition: $A[]$ is a sorted array
      a. Ascending if using a max heap
      b. Descending if using a min heap

E. Priority queues
   i. Create a heap from an array
   ii. Can change a priority in $O(\log_2 n)$ time
   iii. Can dequeue an entry and fix the queue in $O(\log_2 n)$ time
VII. Chapter 7: Quicksort
   A. Description of Quicksort
   B. Description of PARTITION
   C. Performance
      i. Worst case:
         a. Already sorted
         b. $O(n^2)$
      ii. Best case:
         a. Partitioning is balanced
         b. $O(n \log_2 n)$
      iii. Random input:
         a. Much closer to best case than worst case
         b. $O(n \log_2 n)$
      iv. Can get best performance in the vast majority of cases with randomized PARTITION.

VIII. Chapter 15: Dynamic Programming
   A. Rod cutting
      i. Recursive top-down $O(2^n)$ (ouch!)
      ii. Top-down Memoization (dynamic programming)
      iii. Bottom-up Memoization
      iv. Re-constructing the solution
   B. Longest Common Subsequence
      i. Dynamic Programming solution
      ii. “Breadcrumbs”
   C. 0/1 Knapsack
      i. Dynamic programming
      ii. Re-constructing the solution
      iii. Remember: This solution is not polynomial time
      iv. Table can be large as $n$ and capacity of the knapsack grow large

IX. Chapter 16: Greedy Algorithms
   A. Elements of the greedy strategy
   B. Greedy choice property
   C. Optimal substructure or Property of Optimality (POO)
   D. Fractional Knapsack
   E. Huffman codes
      i. Prefix codes
      ii. Constructing the code (binary tree)
   F. Coin changing problem (fewest coins to make change)
      i. Greedy algorithm
      ii. Example where it does not work
   G. 0/1 Knapsack
      i. Greedy algorithm
      ii. Does not always work (oops!)
Material for Exam II starts here

X. Chapter 22: Elementary Graph Algorithms

A. Definitions
   i. vertex (node, point, etc.)
   ii. edge
   iii. weight
   iv. arc
   v. graph
   vi. digraph
   vii. subgraph
   viii. connected graph or digraph
   ix. tree
   x. spanning tree
   xi. path (or chain)
   xii. cycle
   xiii. \( \text{deg}(v) \)
   xiv. \( \text{indeg}(v) \), and \( \text{outdeg}(v) \)

B. Representations of graphs in the computer
   i. adjacency matrix
   ii. adjacency list
   iii. adjacency list as a vector (don’t worry, you won’t see this again in this class)
   iv. Missing edges
   v. Arc/Edge weights (cost, speed, tolls, or whatever)

C. Breadth First Search
D. Depth First Search (digraphs)
E. Topological Sort (digraph)
   i. Based upon DFS
   ii. Alternate method removing nodes and arcs

XI. Chapter 23: Minimum Spanning Trees

A. Prim’s (Greedy)
B. Kruskal’s (Greedy)

XII. Chapter 24: Single-Source Shortest Paths

A. Dijkstra
   i. Greedy algorithm
   ii. Graph
   iii. Digraph
   iv. Positive weights

B. Bellman-Ford
   i. Greedy algorithm
   ii. Graph
   iii. Digraph
   iv. Weights
      a. All positive - returns true
b. Negative - returns false

XIII. Chapter 34: NP Completeness

A. Polynomial time (on a determinate computer) problems
   i. “Easy” problems
   ii. Time complexities are polynomial in problem size, $n$
   iii. Example: $T(n) = an^2 + bn + c$

B. NP (Non-determinate computer Polynomial time) problems
   i. Many are optimization problems, but not all
   ii. Definition: CAREFUL, this is not “non-Polynomial Time”
   iii. Examples we have seen before (0/1 knapsack, TSP, etc.)
   iv. Intractable
   v. No known Polynomial time solution
   vi. Solutions typically have time complexities of $O(2^n)$ or worse
   vii. Solution can be verified in polynomial time.

C. NP Complete
   i. NP Hard
   ii. Decision problems (answer is always “yes” or “no”, 0/1)
   iii. If we can solve one in polynomial time, we can solve them all in polynomial time and then there is a good chance $P = NP$
   iv. Most CS researchers “believe” $P \neq NP$ (The Question!)

D. Examples
   i. Simple changes to an easy problem can make it NP Complete
   ii. Polynomial time solutions
      a. Known solutions for deterministic computer in polynomial time
      b. Fractional Knapsack
      c. Euler Cycle or path
      d. Single source shortest path
   iii. Non-deterministic Polynomial time solutions
      a. no known solutions for deterministic computer in polynomial time
      b. 0/1 Knapsack
      c. Hamiltonian Cycle or path
      d. Single source longest acyclic path