REVIEW TOPICS FOR MIDTERM AND FINAL

The guidelines for the exams:

- Midterm is 90 minutes; Final exam is 180 minutes (3 hours).
- Midterm and final are worth 20% of your total grade each.
- Books, notes, communication with other people, calculators or the Internet access is not allowed. You can take one page of size A4 with any reference information you may need.
- Students who had extenuating reasons why they could not write the midterm (or final) during the regular time, are given one extra opportunity to rewrite it. The make-up exam has different questions covering similar topics at a similar level of detail.

Note: Before the midterm graph algorithms and hashing-related algorithms (Parts 6 and 7 from the list below) were covered during the lectures. Some easier questions on these topics may be asked during the midterm as well. More exercises on these topics will follow in the practice sessions after the midterm, and will be included in the final.

1.1 Question Types

[MathStatements] Mathematical statements about algorithms: Formulate your own facts about algorithms and explain why they are or aren't true. Questions are typically about algorithm complexity – asymptotic bounds for time and space, counting comparisons or other calls performed by algorithms. Mathematical reasoning is also used to prove correctness, for example, proving or disproving data structure invariants: facts that stay true as operations are performed.

[DrawDiagrams] Draw diagrams with the memory states of a data structure: Given the initial state of some data structure and the operations that are performed on it. Typical examples are diagrams showing the state of arrays (representing stacks, queues, lists or heaps), linked lists (representing stacks, queues, lists), subsequent steps for sorting algorithms, graphs or trees augmented with DFS or BFS traversal information.

[UnderstandAlgorithms] Given an algorithm, show its behavior in certain situations. Typical examples include drawing a tree of recursive calls or showing the values of local variables as some event happens. In some cases this overlaps with the (DrawDiagrams) question as it asks to show the memory state of some data structure. But in this case the operations on the data structure are not given in advance – they depend on the algorithm being used.

[CompareImplementations] Compare implementations of algorithms or data container APIs: Given a simple algorithmic task suggest an optimal implementation, write as a pseudocode and analyze its time-complexity. For example, some algorithm may need to be faster than quadratic time to get full credit. You may be asked to provide several implementations for the same abstract data type (such as a priority queue or a dictionary) and write the time complexity.

[DesignByParadigm] Write a pseudocode for an algorithm following some paradigm: Given an algorithmic task, write its solution while observing some guidelines. For example, the solution may need to contain tail-recursion. Or it has to use some paradigm of algorithm creation (paradigms are the chapter titles in *Anany Levitin. Introduction to the Design and Analysis of Algorithms 3rd Edition. Pearson, 2011.*).

[ProblemAnalysis] Given a real-world algorithmic problem, analyze it a human language: Given a problem description, list your assumptions to formalize it and suggest a method how to decompose it into simpler problems and how to solve it algorithmically.

Please note the following:

- Questions related to algorithm implementations in C++ are not part of the exam. Understanding at the pseudocode level (and hand-drawn memory structures) is typically sufficient.
- Advanced question types such as (DesignByParadigm) and (ProblemAnalysis) will be present in the final exam mostly. Still, both exams will also contain simpler, more specific questions.
- A single question may contain multiple related subtasks (A), (B), (C) that refer to each other. You can still receive partial credit, if you succeed with some subtask, but use a wrong input obtained from an earlier subtask.

1.2 List of Topics

Part 1: Asymptotic Growth Rate: Compare functions defined analytically. Compare recurrent sequences. Analyze the time complexity of algorithms from their pseudocode.

1. Analyze the growth rate of functions and sequences:
   A. Given an closed expression of a function, express its growth rate $\Theta(g(n))$ writing $g(n)$ in the simplest form possible.
   B. Compare two or more functions in terms of their asymptotic growth rate.
   C. Given a recurrent sequence, define its asymptotic growth rate.
2. Analyze simple programs and pseudocode:
   A. Express time complexity for a code snippet from the inside out.
   B. Apply assumptions on how fast are certain Python built-in data structures (lists, sets, dictionaries) to analyze simple algorithms in Python.
   C. Write a recurrence to express time complexity of an algorithm and solve it using the Master's theorem.
3. Analyze other complexity measures besides the worst-case time complexity:
   A. Evaluate the space complexity for an algorithm and its asymptotic growth rate.
   B. Evaluate the amortized time complexity, if some operation is applied many times.
   C. Evaluate the number of comparisons needed for sorting, searching or ranking algorithms.

Part 2: Lists, Stacks, Queues:

1. Typical implementations for Lists, Stacks, Queues:
   A. Given an implementation, draw the memory state at a certain moment, e.g. an array or a linked list.
   B. Create a singly linked list implementation of some ADT method.
   C. Create a doubly linked list implementation of some ADT method.
2. Implement a data structure in pseudocode, compare the implementation alternatives:
   A. Express dependent ADT operations in terms of simpler ADT operations.

   B. Given a list/stack/queue algorithm pseudocode, find its time complexity.

   C. Given a problem description, implement the algorithm at ADT Level.

3. Write algorithms using Lists, Stacks or Queues. Algorithms can call list-like data structures using their ADT functions.

   A. Write algorithms and estimate the time complexity of algorithms processing expressions.

   B. Write algorithms using stack to navigate a tree-like structure.

Part 3: Tree-like Structrues:

1. Tree concepts.

   A. Use the concepts of non-rooted trees (plain graph level), rooted trees, ordered trees.

   B. Use the concepts of binary and n-ary trees. For binary trees distinguish full, complete and perfect trees.

   C. Use the concept of binary search tree (labels/keys compare according to the in-order traversal order).

   D. Encode multiway trees with binary trees (and binary trees into multiway trees).

2. Priority Queues and Heaps.

   A. Define priority queue ADT, analyze various non-heap ways to implement it.

   B. Define a heap data structure, compute parents and children, peform insert and delete-min (or delete-max).

   C. Use priority queues to build Huffman prefix code given the alphabet of messages and their probabilities.

3. Tree traversals and Backtracking.

   A. Use BFS traversal order.

   B. Use DFS traversal (for pre-order, in-order, post-order visiting of the nodes).

   C. Solve algorithmic tasks using backtracking.

Part 4: N-ary Search Trees:

1. Regular BSTs

   A. Insert, delete and find keys in a binary search tree.

   B. Answer the questions about their properties.

   C. Perform various flavors of DFS traversals (in-order, pre-order, post-order), find in-order predecessors and successors.

   D. Reason about the expected height of a BST, if you insert keys in certain order.

2. Self-balancing Search Trees.

   A. Draw AVL Trees, answer questions about their properties (worst-case depth etc.), insert and delete keys.

   B. Insert, delete and find keys in multiway search trees.

   C. Draw 2-4 Trees, answer questions about their properties, insert and delete keys.

3. Create and Use Augmented Trees. *Extra information for any node can be computed from other attributes of the node and its children*.

   A. Consider different ways to augment trees ()

   B. Computing $\text{RANK}(v)$ – how many nodes w in the given tree satisfy the inequality $w.key \leq v.key$.

   C. Computing $\text{COUNT}(a, b)$ – how many keys are between a and b.

Part 5: Sorting:

1. Time-complexity for sorting algorithms.
    A. Use Stirling's formula to evaluate factorials and binomial coefficients.
    B. Count comparisons in a decision tree to find the lower bound of comparisons needed.
    C. Analyze some inefficient algorithms such as Bubblesort.
2. Various sorting algorithms:
    A. Criteria how to compare sorting algorithms (efficient/inefficient, stable/unstable, online/offline, in-place/outplace, behavior for random or specific inputs).
    B. Use Mergesort, draw memory states, analyze complexity, count comparisons.
    C. Use Heapsort, draw memory states, analyze complexity, count comparisons.
    D. Use Quicksort, draw memory states, analyze complexity, count comparisons.

Part 6: Graph algorithms:

1. Run graph traversal algorithms:
    A. Run Breadth-first-search (BFS) on undirected and directed graphs, classify edges as forward edges, back edges or cross edges.
    B. Run Depth-first-search (DFS) on undirected and directed graphs, add start and finish timestamps, classify edges as forward edges, back edges, cross edges or forward edges.
    C. Topologically sort vertices in a directed graph or establish that it is impossible.
    D. Find strongly connected components using Kosaraju's algorithm.
2. Run graph optimization algorithms:
    A. Run single-source shortest paths algorithms such as Dijkstra's and Bellman-Ford.
    B. Run all-pairs shortest paths algorithms such as Floyd–Warshall.
    C. Run MST algorithms such as Prim's and Kruskal's.
3. Run flow-related algorithms.
    A. Reason with augmenting paths regarding maximum flow or maximum matching problems.
    B. Run maximum flow algorithms such as Ford-Fulkerson or Edmonds–Karp.
    C. Run maximum matching algorithms such as Hopcroft-Karp algorithm.

Part 7: Sets, dictionaries and hashing:

1. Use hashing data structure:
    A. Describe and compute some typical implementations for hashing functions based on modular arithmetic.
    B. Resolve hash collisions by chaining and analyze the expected time complexity for such hashtables.
    C. Resolve hash collisions using various open addressing methods – linear probing, quadratic probing or double hashing.
2. Implement and use sets, multisets or maps.
    A. Compare hashing-based vs. tree-based implementations of sets and maps.
    B. Describe polynomial-based rolling hash algorithm, Rabin-Karp string search algorithm and its uses in checking plagiarism.
    C. Use and reason about secure hashing algorithms (such as SHA-256 or MD5), how they are used in password caching or communication algorithms.