LEC21: REVIEW OF ANALYSIS OF ALGORITHMS
COURSE OUTCOME:

The ABET objectives for the course:

1. Ability to perform **worst-case asymptotic algorithm analysis**
2. Ability to **define and use classical combinatorial algorithms** for problems such as sorting, shortest paths and minimum spanning trees
3. Knowledge of **computational intractability** and **NP completeness**

The program objective for the course:

+ (S6) have a solid understanding of computational theory and foundational mathematics.
Objectives

Read about the CS accreditation ABET program. The ABET objectives for the course are

Provide a rigorous introduction to worst-case asymptotic algorithm analysis.

Develop classical graph and combinatorial algorithms for such problems as sorting, shortest paths and minimum spanning trees.

Introduce the concept of computational intractability and NP completeness.
WHAT IS AN ALGORITHM?

- An algorithmic problem is specified by describing the
  set of instances it must work on and
  what desired properties the output must have.

Properties of Algorithms

- Correctness: For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem.
- Efficient
Failure to find a counterexample to a given algorithm does not mean “it is obvious” that the algorithm is correct.

Mathematical **induction** is a very useful method for proving the correctness of **recursive algorithms**.

Recursion and induction are the same basic idea:

1. (1) basis case,
2. (2) general assumption,
3. (3) general case.

Ex> proving

\[ \sum_{i=1}^{n} i = n(n + 1)/2 \]
THE RAM MODEL OF COMPUTATION

Algorithms are an important and durable part of computer science because they can be studied in a machine/language independent way.

This is because we use the **RAM model of computation** for all our analysis.

- Each “*simple*” operation (+, *, -, =, if, call) takes 1 step.
- Loops and subroutines are *not* simple operations. They depend upon the size of the data and the contents of a subroutine.
  - ex> “Sort” is not a single step operation.
ASYMPTOTIC NOTATIONS: NAMES OF BOUNCING FUNCTIONS

- Big-Oh: $g(n) = O(f(n))$ means $C \cdot f(n)$ is an **upper bound** on $g(n)$.

- Big-Omega: $g(n) = \Omega(f(n))$ means $C \cdot f(n)$ is a **lower bound** on $g(n)$.

- Big-Theta: $g(n) = \Theta(f(n))$ means $C_1 \cdot f(n)$ is an **upper bound** on $g(n)$ and $C_2 \cdot f(n)$ is a **lower bound** on $g(n)$.
  (a.k.a. **tight bound**)  

  $C$, $C_1$, and $C_2$ are all constants independent of $n$. 

The definitions imply a constant $n_0$ beyond which they are satisfied. We do not care about small values of $n$. 

- **Big-Oh:** $f(n) = O(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

- **Big-Omega:** $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

- **Big-Theta:** $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that to the right of $n_0$, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive.
Faster-growing function **dominates** a slower-growing one

Common functions that appear in algorithms analysis order of increasing dominance:

\[ n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1 \]

- **Constant functions**, \( f(n) = 1 \)
- **Logarithmic functions**, \( f(n) = \log n \)
- **Linear functions**, \( f(n) = n \)
- **Superlinear functions**, \( f(n) = n \log n \)
- **Quadratic functions**, \( f(n) = n^2 \)
- **Cubic functions**, \( f(n) = n^3 \)
- **Exponential functions**, \( f(n) = c^n \) for a given constant \( c > 1 \)
- **Factorial functions**, \( f(n) = n! \)
PROGRAM COMPLEXITY ANALYSIS

- Determining time complexity analysis given a code.
  + EX> Selection Sort Worst Case Analysis
  + EX> Insertion Sort Worst Case Analysis
  + EX> String Pattern Matching Worst Case Analysis

- Properties of Logarithms
  + In relation to Trees - ex> Binary Search
  + Logarithms and Multiplication
  + The Base is not Asymptotically Important
  + Logarithms and Bits
Complexity of an algorithms may differ when using different data structures.

Types of DS:
- Contiguous vs. Linked Data Structures
- Containers: Stacks and Queues

Dictionary / Dynamic Set Operations & time analysis
- Basic Operations: \textit{Search}(S,k) \textit{Insert}(S,x) \textit{Delete}(S,x)
- Binary Search Trees: operations.
- Balanced Search Trees
- Hash Tables: Collisions, hash functions, Performance on Set Operations
- Analysis of Substring Pattern Matching using different dictionary data structures.
SORTING

- Applications of sorting
- Pragmatics of Sorting
- Selection Sort:
  + Data Structure Matters: Heapsort
- Priority Queue:
  + operations;
  + implementations;
  + time analysis of operations based on data structure used
  + Applications
- Binary Heap:
  + Constructing Heaps
  + Heap operations and time analysis: Bubble up & Bubble down
SORTING

- MergeSort & Divide-and-conquer
- Analysis of Algorithms that use Divide-and-conquer
  + EX> matrix multiplication
  + Divide-and-Conquer Recurrences
  + Application of Master Theorem
- Quicksort & Partitioning
  + Analysis – Best case, worst case, average case analysis
  + Randomized analysis
- Lower Bound Analysis on Sorting – comparison based
- Non-Comparison-Based Sorting
  + Bucketsort – time complexity
Graph data structure – characteristics & operations

- Adjacency Matrix
- Adjacency list

Graph terminology:

- Degree
- Connected & strongly connected

Types of graphs:

- Directed vs. Undirected Graphs
- Weighted vs. Unweighted Graphs
- Simple vs. Non-simple Graphs
- Sparse vs. Dense Graphs
- Cyclic vs. Acyclic Graphs
BREADTH-FIRST SEARCH

- Graph traversal: We want to visit every vertex and every edge exactly once in some well-defined order.
- Breadth-first search is appropriate if we are interested in shortest paths on unweighted graphs.
- How BFS works on graphs.
- Data Structure for BFS – using queue
- Applications of BFS
  - Shortest Paths
  - Connected Components
  - Two-Coloring Graphs – Bipartite
DEPTH-FIRST SEARCH

- BFS v.s. DFS
- How DFS works on graphs.
- Characteristics of DFS algo
  - Edge Classification for DFS: tree edges, back edges, ...
  - Finding ancestor and descendants by time intervals & applications
- Data structure for DFS: stack (recursion)
- Applications of DFS
  - Finding Cycles
  - Articulation Vertices
  - Topological Sorting
  - Strongly Connected Components
Input: Edge-weighted graphs
Characteristic of MSP

Applications
- Net Partitioning
- provides a good heuristic for traveling salesman problems

Prim’s Algorithm:
- how it works, characteristics, & time analysis

Kruskal’s Algorithm:
- how it works, characteristics, and time analysis
SHORTEST PATH

- Characteristic of shortest path problem
- Dijkstra’s Algorithm (single source shortest path)
  - how it works (Dynamic Programming), characteristics, & time analysis
  - Difference between Prim’s/Dijkstra’s
- The Floyd-Warshall Algorithm (all-pairs shortest path)
  - how it works (Dynamic Programming), characteristics, & time analysis
- Applications:
BACKTRACKING

- What is Backtracking used for?

- How to apply Backtracking
  - Modeling the solution vector
  - Recursive structure – similar to DFS
  - How to model: is_a_solution(a,k,input); process_solution(a,k,input);
    construct_candidates(a,k,input,c,&ncandidates); make_move(a,k,input);

- Applications:
  - Sudoku
  - Constructing all Subsets
  - Constructing all Permutations
  - The Eight-Queens Problem
  - Can Eight Pieces Cover a Chess Board?
HEURISTIC SEARCH

- Backtracking searches all configurations to find the best of all possible solutions.
- **Heuristic methods** provide an alternate way to approach difficult combinatorial optimization problems.
  - Solution space representation
  - Cost function
- **Heuristic search methods:**
  - Random sampling,
  - Local search strategy
    - Gradient-descent search
    - Simulated annealing
Dynamic Programming

- When DP is appropriate.
- Characteristics & Benefits of DP
  - systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing
- Three Steps to Dynamic Programming
  1. Formulate the answer as a recurrence relation
  2. Show that the number of different instances of your recurrence is bounded by a polynomial.
  3. Specify an order of evaluation for the recurrence so you always have what you need.
DP CONT.

- Examples:
  + Fibonacci Numbers
  + Binomial Coefficients - Pascal’s Triangle

- Edit Distance & applications
  + How Edit Distance works & analysis
  + Substring Matching
  + Longest Common Subsequence
  + Maximum Monotone Subsequence (Longest Increasing Sequence)
  + The Partition Problem
  + Minimum Weight Triangulation

- Comparing DP with Recurrence

- Limitations of Dynamic Programming: TSP
  + Principle of optimality
Bandersnatch(G)
+ Convert G to an instance of the Bo-billy problem Y.
+ Call the subroutine Bo-billy on Y to solve this instance.
+ Return the answer of Bo-billy(Y) as the answer to G.

Now suppose my reduction translates G to Y in $O(P(n))$:
+ 1. If my Bo-billy subroutine ran in $O(P'(n))$ I can solve the Bandersnatch problem in $O(P(n) + P'(n'))$
+ 2. If I know that $\Omega(P'(n))$ is a lower-bound to compute Bandersnatch, then $\Omega(P'(n) - P(n'))$ must be a lowerbound to compute Bo-billy.
Concepts: problem, instance, decision problem