Advanced Algorithms: The 20/80 Summary

COMS3005A - University of the Witwatersrand

2024

Introduction

This comprehensive summary distills the core principles (20%) that enable deep understanding of all algorithmic concepts (80%) from the Advanced Analysis of Algorithms course. The 20% represents the fundamental mental models and ways of thinking that allow you to analyze, design, and reason about any algorithm, while the 80% represents the specific implementations and applications of these principles. Master these fundamentals to develop true algorithmic intuition.

1 The 20%: Foundational Principles

1.1 The Trade-Off Triad: The Fundamental Constraint

- **Time vs Space vs Optimality**: Every algorithmic decision represents a point in this three-dimensional trade-off space. Understanding where your solution lies in this space is crucial for making informed design decisions.
- **Time Complexity**: How the runtime scales with input size. This is typically the primary concern, but not always. Some systems prioritize predictable performance over absolute speed.
- **Space Complexity**: How memory usage scales with input size. In memory-constrained environments (embedded systems, mobile devices), this can be the dominant constraint.
- **Optimality**: Whether the solution is guaranteed to be the best possible. In practice, we often accept suboptimal solutions that are "good enough" but much faster to compute.
- **Real-world Example**: Merge Sort provides optimal $O(n \log n)$ time but requires $O(n)$ extra space. Quick Sort provides $O(n \log n)$ average time with only $O(\log n)$ extra space but has $O(n^2)$ worst case. The choice depends on which trade-offs are acceptable for your specific context.

1.2 Asymptotic Analysis: The Language of Scalability

- **Big-O (O)**: Describes the *upper bound* of growth. When we say an algorithm is $O(n^2)$, we mean it will *not grow faster than* quadratic time. This is about *worst-case* guarantees.
- **Omega (Ω)**: Describes the *lower bound* of growth. $\Omega(n \log n)$ means the algorithm requires *at least* linearithmic time. This captures *best-case* requirements.
- **Theta (Θ)**: Describes a *tight bound* where the algorithm grows *exactly* at this rate (both upper and lower bounds match). This is the most informative notation when applicable.
- **Practical Interpretation**:
  - $O(1)$: Instant, regardless of input size (hash table lookup)
  - $O(\log n)$: Incredibly fast even for huge inputs (binary search)

- $O(n)$: Proportional time (linear scan)
- $O(n \log n)$: The "bar" for efficient algorithms (optimal sorting)
- $O(n^2)$: Becomes slow for moderate inputs (naïve sorting)
- $O(2^n)$: Impractical except for tiny inputs (brute force)

- **Why Constants Don't Matter**: For large n, the growth rate dominates any constant factors. $1000n$ is eventually better than $2n^2$ because linear growth will always beat quadratic growth for sufficiently large n.

1.3 Problem-Solving Paradigms: The Algorithm Designer's Toolkit

- **Brute Force**: The simplest approach - enumerate all possibilities. While often impractical ($O(n!)$, $O(2^n)$), it serves as:
  - A baseline for comparing more sophisticated algorithms
  - A solution for very small input sizes
  - A starting point for understanding the problem structure
- **Decrease & Conquer**: Solve a smaller instance of the *same* problem, then extend the solution. Examples:
  - **Insertion Sort**: Sort first $n-1$ elements, then insert the n^{th} element
  - **Binary Search**: Eliminate half the search space at each step
- **Divide & Conquer**: The most powerful paradigm:
  - **Divide**: Split problem into independent subproblems
  - **Conquer**: Solve subproblems recursively
  - **Combine**: Merge solutions to form final answer
  - **Examples**: Merge Sort, Quick Sort, Closest Pair
- **Transform & Conquer**: Convert the problem into a different representation where it becomes easier:
  - Sort data first to enable efficient searching
  - Use hash functions to transform complex keys into simple indices
  - Represent graphs as adjacency matrices vs lists based on operations needed
- **Greedy**: Make the locally optimal choice at each step. The key insight is recognizing when local optimality guarantees global optimality.
- **Dynamic Programming**: The art of solving each subproblem exactly once by storing solutions. The essence is overlapping subproblems + optimal substructure.

1.4 Data Structures as Algorithm Enablers

- **Critical Insight**: Algorithms don't exist in isolation - they're built on data structures that enable specific operations efficiently.
- **Array**:
  - **Strengths**: $O(1)$ random access, cache-friendly, simple

- **Weaknesses**: Fixed size, slow insertion/deletion ($O(n)$)
- **Best for**: Sequential processing, known size datasets

- **Linked List**:
  - **Strengths**: $O(1)$ insertion/deletion at known positions, dynamic size
  - **Weaknesses**: $O(n)$ random access, poor cache performance
  - **Best for**: Frequent insertions/deletions, unknown size data
- **Hash Table**:
  - **Strengths**: $O(1)$ average case operations, flexible keys
  - **Weaknesses**: $O(n)$ worst case, no ordering, hash function dependent
  - **Best for**: Dictionary operations, membership testing
- **Binary Search Trees**:
  - **Strengths**: $O(\log n)$ operations, maintains ordering, range queries
  - **Weaknesses**: $O(n)$ worst case if unbalanced, more complex
  - **Best for**: Ordered data, range queries, sequential access

1.5 Correctness & Optimality: Mathematical Rigor

- **Proof Techniques**:
  - **Induction**: Perfect for recursive algorithms and loop invariants
  - **Contradiction**: Assume algorithm is wrong and derive contradiction
  - **Adversary Arguments**: Construct worst-case inputs that force minimum work
  - **Loop Invariants**: Properties that hold before/during/after loop execution
- **Lower Bounds**: The theoretical minimum work required to solve a problem:
  - **Comparison Sorting**: $\Omega(n \log n)$ from decision tree height
  - **Searching Unsorted Data**: $\Omega(n)$ from adversary argument
  - **Importance**: Tells us when we can stop looking for better algorithms
- **Optimality**: An algorithm is optimal if its complexity matches the problem's lower bound. Merge Sort is optimal for comparison sorting. Binary Search is optimal for searching sorted arrays.

2 Searching Algorithms

2.1 Linear Search: The Baseline

- **Fundamental Operation**: Sequential comparison of each element with the key until found or end reached.
- **Complexity Analysis**:
  - **Best Case**: $\Theta(1)$ - key is first element
  - **Worst Case**: $\Theta(n)$ - key is last element or not present

  - **Average Case**: $\Theta(n)$ - key is equally likely in any position
- **Optimality Proof**: For unordered data, any correct algorithm must examine every element in the worst case. An adversary could always place the key in the last position examined.
- **Variants**:
  - **Sentinel Technique**: Place key at end to eliminate bound checking
  - **Probability-based**: Search more likely positions first
- **Real-world Use**: Small datasets, unsorted data, when simplicity is more important than speed.

2.2 Binary Search: Divide and Conquer in Action

- **Prerequisite**: Data must be sorted - this is the "transform" step that enables the efficient "conquer" step.
- **Algorithm Intuition**: At each step, the search space is halved. This exponential reduction is what gives logarithmic complexity.
- **Complexity Analysis**:
  - **Recurrence Relation**: $T(n) = T(n/2) + O(1)$
  - **Solving**: $T(n) = T(n/4) + 2 = T(n/8) + 3 = \cdots = T(1) + \log n = O(\log n)$
  - This represents the height of the decision tree
- **Implementation Details**:
  - **Midpoint Calculation**: Use $\lfloor low + (high - low)/2 \rfloor$ to avoid overflow
  - **Termination Condition**: $low \leq high$ vs $low < high$ affects edge cases
  - **Element Not Found**: Return insertion point for complete specification
- **Variants**:
  - **Lower/Upper Bound**: Find first/last occurrence in duplicates
  - **Exponential Search**: Unknown array size
  - **Interpolation Search**: $O(\log \log n)$ for uniform distributions
- **Optimality**: The decision tree argument shows that any comparison-based search on sorted data requires $\Omega(\log n)$ comparisons.

3 Sorting Algorithms

3.1 Comparison Sort Lower Bound: A Fundamental Limit

- **The Core Argument**:
  - There are $n!$ possible permutations of n elements
  - Each comparison gives at most 1 bit of information (true/false)
  - To distinguish among $n!$ permutations, we need $\log(n!)$ bits
  - $\log(n!) \approx n \log n - n \log e + \Theta(\log n)$ by Stirling's approximation
  - Therefore, $\Omega(n \log n)$ comparisons are required

- **Decision Tree Model**: Any comparison-based sort corresponds to a decision tree where:
  - Internal nodes represent comparisons
  - Leaves represent sorted permutations
  - The height of the tree is the number of comparisons needed
- **Implications**:
  - Merge Sort, Heap Sort are optimal (they achieve $O(n \log n)$)
  - Quick Sort is optimal on average ($O(n \log n)$ expected)
  - No comparison-based sort can do better than $O(n \log n)$
- **Non-comparison Sorts**: Bucket Sort, Radix Sort can achieve $O(n)$ by exploiting additional information about the data.

3.2 Quadratic Sorts: When Simple is Better

- **Selection Sort**:
  - **Mechanism**: Repeatedly find minimum element and swap to front
  - **Complexity**: Always $\Theta(n^2)$ comparisons, $O(n)$ swaps
  - **Advantage**: Minimal data movement - good when writes are expensive
  - **Stability**: Not stable due to swapping distant elements
- **Bubble Sort**:
  - **Mechanism**: Repeatedly swap adjacent inverted elements
  - **Complexity**: $O(n^2)$ worst/average, $O(n)$ best (already sorted)
  - **Optimization**: Stop if no swaps in a pass (detects sorted array)
  - **Advantage**: Simple to implement, stable, detects sorted input
  - **Real Use**: Educational purposes, tiny datasets
- **Insertion Sort**:
  - **Mechanism**: Build sorted array one element at a time by insertion
  - **Complexity**: $O(n^2)$ worst/average, $O(n)$ best (already sorted)
  - **Advantages**:
    * Excellent for small n ($n \leq 50$)
    * Adaptive: very fast on nearly sorted data
    * Stable: preserves order of equal elements
    * In-place: $O(1)$ extra space
  - **Real Use**: Small arrays, as the base case for hybrid sorts like Timsort
- **When to Use Quadratic Sorts**:
  - Small datasets where n^2 is acceptable
  - Nearly sorted data (Insertion Sort shines here)

- Simple implementation is more important than absolute speed
- Educational contexts to understand sorting fundamentals

3.3 Merge Sort: The Optimal Workhorse

- **Divide and Conquer Structure**:
  - **Divide**: Split array into two equal halves
  - **Conquer**: Recursively sort both halves
  - **Combine**: Merge the two sorted halves
- **Merge Operation**: The key to efficiency:
  - Compare elements from front of both subarrays
  - Copy smaller element to result
  - Continue until one subarray is exhausted
  - Copy remaining elements
  - Time: $\Theta(n)$ for merging two subarrays of total size n
- **Complexity Analysis**:
  - **Recurrence**: $T(n) = 2T(n/2) + \Theta(n)$
  - **Solving**:

$$\begin{aligned}T(n) &= 2T(n/2) + cn \\ &= 2[2T(n/4) + c(n/2)] + cn = 4T(n/4) + 2cn \\ &= 4[2T(n/8) + c(n/4)] + 2cn = 8T(n/8) + 3cn \\ &\vdots \\ &= 2^k T(n/2^k) + kcn\end{aligned}$$

  - When $n/2^k = 1$, $k = \log n$, so $T(n) = nT(1) + cn \log n = \Theta(n \log n)$
- **Properties**:
  - **Stable**: Yes (if merge prefers left element on ties)
  - **Parallelizable**: Easy to parallelize the recursive calls
  - **External Sorting**: Can sort data too large for memory
  - **Disadvantage**: $O(n)$ extra space requirement
- **Variants**:
  - **Natural Merge Sort**: Exploits existing sorted runs
  - **Bottom-up Merge Sort**: Iterative version avoids recursion
  - **In-place Merge Sort**: Complex but reduces space to $O(1)$

3.4 Quick Sort: The Practical Champion

- **Algorithm Structure**:
  - **Choose Pivot**: Select an element to partition around

- **Partition**: Rearrange so elements < pivot come before, elements > pivot come after
- **Recurse**: Sort left and right partitions

- **Partition Strategies**:
  - **Lomuto**: Simpler but less efficient, multiple swaps
  - **Hoare**: More efficient, fewer swaps, complex invariants
  - **Dutch National Flag**: Handles duplicates efficiently
- **Pivot Selection Critical**:
  - **Worst Case**: Already sorted data with first/last pivot ($O(n^2)$)
  - **Best Case**: Median element as pivot ($O(n \log n)$)
  - **Good Strategies**:
    * **Median-of-Three**: First, middle, last - choose median
    * **Random**: Provides probabilistic guarantees
    * **Introselect**: Hybrid for guaranteed $O(n \log n)$
- **Complexity Analysis**:
  - **Worst Case**: $O(n^2)$ when pivot is always min/max
  - **Best Case**: $O(n \log n)$ when pivot is always median
  - **Average Case**: $O(n \log n)$ with constant factor $\approx 1.39n \log n$
  - **Expected Case**: $O(n \log n)$ with randomized pivot
- **Why Quick Sort Wins in Practice**:
  - Excellent cache performance (sequential access pattern)
  - Small constant factors
  - In-place ($O(\log n)$ stack space for recursion)
  - Hardware-friendly access patterns
- **Optimizations**:
  - **Hybrid Approach**: Switch to Insertion Sort for small subarrays
  - **Tail Recursion**: Eliminate one recursive call
  - **Three-way Partitioning**: Handle duplicates efficiently

4 Informed Search & Heuristics

4.1 A* Search: The Perfect Balance

- **The Insight**: Best-first search that considers both:
  - $g(n)$: Actual cost from start to node n (like Uniform Cost Search)
  - $h(n)$: Estimated cost from node n to goal (like Greedy Search)
- **Evaluation Function**: $f(n) = g(n) + h(n)$ represents estimated total cost through node n

- **Optimality Conditions**:
  - **Admissible Heuristic**: $h(n) \leq h^*(n)$ for all n (never overestimates)
  - **Consistent Heuristic**: $h(n) \leq c(n, n') + h(n')$ for all n, n' (triangle inequality)
  - **Tree Search**: Optimal with admissible heuristic
  - **Graph Search**: Optimal with consistent heuristic
- **Why Consistency Matters**: Ensures $f(n)$ is non-decreasing along any path, so we never need to reconsider nodes
- **Complexity**:
  - **Time**: $O(b^{\epsilon d})$ where ϵ is heuristic error
  - **Space**: $O(b^d)$ - stores all generated nodes
  - **Optimal Efficiency**: No other optimal algorithm expands fewer nodes
- **Implementation**:
  - **Priority Queue**: Ordered by $f(n) = g(n) + h(n)$
  - **Closed Set**: Track visited nodes for graph search
  - **Path Reconstruction**: Store parent pointers

4.2 Heuristic Design: The Art of Estimation

- **Admissible Heuristics for Pathfinding**:
  - **Manhattan Distance**: $|x_1 - x_2| + |y_1 - y_2|$ for grid movement
  - **Euclidean Distance**: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ for direct movement
  - **Chebyshev Distance**: $\max(|x_1 - x_2|, |y_1 - y_2|)$ for 8-direction movement
- **Relaxation Method**: Create admissible heuristics by solving a relaxed version of the problem:
  - **8-puzzle**: Remove wall constraints = Manhattan distance
  - **8-puzzle**: Remove tile interaction = misplaced tiles count
  - The more constraints removed, the easier to compute but less informative
- **Pattern Databases**: Precompute solutions to subproblems:
  - Store optimal costs for pattern configurations
  - Use as lookup during search
  - Very effective for puzzles like 15-puzzle, Rubik's cube
- **Learning Heuristics**:
  - Machine learning to predict costs
  - Neural networks for complex state evaluation
  - Requires training data but can discover non-obvious patterns
- **Heuristic Quality**: Measured by:

- **Effective Branching Factor**: How much the heuristic reduces search
- **Informedness**: How close $h(n)$ is to $h^*(n)$ without overestimating
- The perfect heuristic $h(n) = h^*(n)$ would solve the problem instantly

5 Greedy Algorithms

5.1 The Greedy Choice Property

- **Core Idea**: A series of locally optimal choices leads to a globally optimal solution.
- **When It Works**:
  - **Matroid Structure**: Problems that can be represented as matroids
  - **Exchange Argument**: Can transform any solution to the greedy solution without making it worse
  - **Greedy Stays Ahead**: The greedy solution is never worse than any partial solution
- **Optimal Substructure**: An optimal solution contains optimal solutions to subproblems. This is shared with Dynamic Programming, but greedy makes irrevocable choices.
- **Proof Techniques**:
  - **Greedy Stays Ahead**: Show greedy is always at least as good as any other solution at each step
  - **Exchange Argument**: Transform any optimal solution into the greedy solution
  - **Matroid Theory**: Formal mathematical framework for greedy algorithms

5.2 Classic Greedy Problems

- **Optimal Service (Scheduling)**:
  - **Problem**: Minimize total waiting time for n customers
  - **Greedy Strategy**: Serve shortest job first
  - **Proof**: Exchange argument - swapping two jobs shows optimality
  - **Complexity**: $O(n \log n)$ for sorting
- **Change Making**:
  - **Problem**: Make change with fewest coins
  - **Greedy Strategy**: Use largest denomination possible
  - **When Optimal**: For "canonical" coin systems (1, 5, 10, 25, ...)
  - **Counterexample**: Coins 1, 3, 4, target 6: greedy gives 4+1+1=3 coins, optimal is 3+3=2 coins
  - **Characterization**: Coin system is canonical if greedy works for all amounts
- **Huffman Coding**:
  - **Problem**: Optimal prefix-free compression
  - **Greedy Strategy**: Merge least frequent symbols
  - **Optimality**: Exchange argument shows no better code exists

  - **Complexity**: $O(n \log n)$ with priority queue

- **Interval Scheduling**:
  - **Problem**: Schedule maximum number of non-overlapping intervals
  - **Greedy Strategy**: Choose interval with earliest finish time
  - **Proof**: Greedy stays ahead - always leaves maximum remaining capacity

- **Minimum Spanning Tree**:
  - **Problem**: Connect all vertices with minimum total edge weight
  - **Greedy Strategies**:
    * **Kruskal's**: Add smallest edge that doesn't create cycle
    * **Prim's**: Grow tree from vertex adding cheapest connecting edge
  - **Optimality**: Cut property guarantees optimality

5.3 When Greedy Fails

- **Typical Failure Modes**:
  - Local optimum doesn't lead to global optimum
  - Decisions are not reversible
  - Problem lacks optimal substructure

- **Examples of Failure**:
  - **0-1 Knapsack**: Greedy by value/weight ratio fails
  - **Traveling Salesman**: Nearest neighbor heuristic can be arbitrarily bad
  - **Non-canonical Coin Systems**: Greedy change making suboptimal

- **Testing Greedy Approaches**:
  - Try to construct counterexamples
  - Check if exchange argument works
  - Verify greedy choice property holds

6 Dynamic Programming

6.1 Recognizing DP Problems

- **Overlapping Subproblems**: The same subproblem is solved multiple times in a naive recursive solution.
- **Optimal Substructure**: An optimal solution can be constructed from optimal solutions of subproblems.
- **Common Patterns**:
  - "Find the minimum/maximum cost/path"
  - "Count the number of ways to..."
  - "Decide if possible to achieve..."

  - Sequences, strings, grids, trees, graphs
- **DP vs Divide & Conquer**:
  - **Divide & Conquer**: Subproblems are independent
  - **Dynamic Programming**: Subproblems overlap
- **DP vs Greedy**:
  - **Greedy**: Make choice and solve one subproblem
  - **Dynamic Programming**: Try all choices and combine results

6.2 DP Implementation Strategies

- **Top-Down with Memoization**:
  - Write natural recursive solution
  - Add cache to store computed results
  - Check cache before computing
  - Store result in cache after computing
  - **Advantages**: Natural, computes only needed subproblems
  - **Disadvantages**: Recursion overhead, harder to optimize space
- **Bottom-Up with Tabulation**:
  - Identify dependency order of subproblems
  - Solve smallest subproblems first
  - Build up to original problem
  - **Advantages**: No recursion overhead, easier space optimization
  - **Disadvantages**: May compute unnecessary subproblems, less intuitive
- **State Definition**: The art of DP is defining the state:
  - What parameters define a subproblem?
  - What is the recurrence relation between states?
  - What are the base cases?
  - How do we reconstruct the solution?

6.3 Classic DP Problems and Patterns

- **Fibonacci Sequence**:
  - **Naive**: $O(2^n)$ - exponential blowup
  - **DP**: $O(n)$ time, $O(n)$ space
  - **Optimized**: $O(n)$ time, $O(1)$ space - only need last two values
  - **Pattern**: Caching overlapping computations
- **Change Making**:

- **Problem**: Minimum coins to make amount with given denominations
- **State**: $dp[i]$ = min coins for amount i
- **Recurrence**: $dp[i] = \min(dp[i - c_j] + 1)$ for all coins $c_j \leq i$
- **Complexity**: $O(n \cdot k)$ for amount n, k coin types
- **Advantage**: Handles non-canonical coin systems

- **Longest Common Subsequence**:
  - **State**: $dp[i][j]$ = LCS of first i chars of A and first j chars of B
  - **Recurrence**:
    * If $A[i] = B[j]$: $dp[i][j] = dp[i-1][j-1] + 1$
    * Else: $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$
  - **Complexity**: $O(mn)$ for strings of length m, n
  - **Reconstruction**: Trace back through DP table
- **Matrix Chain Multiplication**:
  - **Problem**: Optimal parenthesization of matrix multiplication
  - **State**: $dp[i][j]$ = min cost to multiply matrices i through j
  - **Recurrence**: $dp[i][j] = \min(dp[i][k] + dp[k+1][j] + cost)$ for $i \leq k < j$
  - **Complexity**: $O(n^3)$ vs brute force $O(2^n)$
- **Knapsack Problems**:
  - **0-1 Knapsack**: Each item take or leave
  - **Unbounded Knapsack**: Unlimited copies of each item
  - **State**: $dp[i][w]$ = max value with first i items and weight w
  - **Complexity**: $O(nW)$ pseudo-polynomial
- **Edit Distance**:
  - **Problem**: Minimum operations to transform string A to B
  - **Operations**: Insert, delete, replace
  - **State**: $dp[i][j]$ = edit distance of first i chars of A and first j chars of B
  - **Recurrence**: Consider all three operations

6.4 DP Optimization Techniques

- **Space Optimization**:
  - Reuse arrays when only previous row/column needed
  - Fibonacci: $O(1)$ space instead of $O(n)$
  - Knapsack: $O(W)$ space instead of $O(nW)$
- **State Reduction**:
  - Find more compact state representation

- Use bitmasking for small sets
- Exploit symmetries in the problem

- **Decision Optimization**:
  - Monotonic queue/stack for certain recurrences
  - Convex hull trick for specific cost functions
  - Divide and conquer optimization

7 Computational Geometry

7.1 Closest Pair Problem: Divide and Conquer Mastery

- **Problem Statement**: Given n points in the plane, find the pair with smallest Euclidean distance.
- **Brute Force**: Check all $\binom{n}{2}$ pairs - $O(n^2)$
- **Divide and Conquer Approach**:
  1. **Sort by x-coordinate**: $O(n \log n)$ preprocessing
  2. **Divide**: Split points into left and right halves by x-coordinate
  3. **Conquer**: Recursively find closest pairs in left and right
  4. **Combine**: Check pairs that cross the dividing line
- **The Key Insight - The Strip**:
  - Let $\delta = \min(\delta_{left}, \delta_{right})$
  - Only need to consider points within δ of the dividing line
  - For each point in the strip, only check next 7 points by y-coordinate
- **Why 7 Points? Geometric Proof**:
  - Consider a $\delta \times 2\delta$ rectangle centered on the dividing line
  - Divide into 8 $\delta/2 \times \delta/2$ squares
  - Each square can contain at most 1 point (otherwise δ would be smaller)
  - Therefore, at most 8 points in the rectangle
  - For any point, only need to check the other 7
- **Complexity Analysis**:
  - **Recurrence**: $T(n) = 2T(n/2) + O(n)$
  - **Solving**: By master theorem, $T(n) = O(n \log n)$
  - **Total**: $O(n \log n)$ for sort + $O(n \log n)$ for algorithm = $O(n \log n)$
- **Implementation Details**:
  - **Sorting**: Pre-sort by x and maintain y-order during recursion
  - **Merge by y**: $O(n)$ merge of two sorted y-lists
  - **Strip Processing**: $O(n)$ for processing all points in strip

- **Generalization**:
  - **Higher Dimensions**: Becomes $O(n \log^{d-1} n)$ for d dimensions
  - **Other Metrics**: Works for any L_p metric
  - **Approximate Versions**: Can find $(1+\epsilon)$-approximation faster

7.2 Other Geometric Algorithms

- **Convex Hull**:
  - **Graham Scan**: $O(n \log n)$ - sort by angle and scan
  - **Jarvis March**: $O(nh)$ - output-sensitive, good for small hulls
  - **QuickHull**: $O(n \log n)$ expected - divide and conquer
- **Line Segment Intersection**:
  - **Sweep Line Algorithm**: $O(n \log n)$
  - **Sweep vertical line**, maintain active segments in balanced BST
  - Detect intersections when sweep line reaches endpoints
- **Point Location**:
  - **Preprocessing**: Build data structure for point-in-polygon queries
  - **Trapezoidal Decomposition**: $O(n \log n)$ preprocessing, $O(\log n)$ query
- **Voronoi Diagrams & Delaunay Triangulation**:
  - **Fortune's Algorithm**: $O(n \log n)$ sweep line
  - **Applications**: Nearest neighbor, mesh generation, terrain modeling

8 Data Structures

8.1 Hash Tables: The Power of Direct Access

- **Core Idea**: Use a hash function to map keys to array indices, enabling $O(1)$ average-case operations.
- **Hash Function Design**:
  - **Requirements**: Fast to compute, uniform distribution, deterministic
  - **Good Hash Functions**: MurmurHash, CityHash, cryptographic hashes
  - **Universal Hashing**: Family of hash functions for worst-case guarantees
- **Collision Resolution Strategies**:
  - **Separate Chaining**:
    - * Each bucket is a linked list
    - * Simple, handles arbitrary load factors
    - * Cache-unfriendly, pointer overhead
  - **Open Addressing**:

    * Store all entries in the table itself
    * **Linear Probing**: $h(k, i) = (h(k) + i) \mod m$
    * **Quadratic Probing**: $h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m$
    * **Double Hashing**: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$

- **Load Factor and Performance**:
  - $\alpha = n/m$ where n = entries, m = buckets
  - **Separate Chaining**: Expected chain length = α
  - **Open Addressing**: Performance degrades as $\alpha \to 1$
  - **Rehashing**: Double table size when α exceeds threshold
- **Complexity Analysis**:
  - **Average Case**: $O(1)$ for all operations with good hash function
  - **Worst Case**: $O(n)$ if all keys hash to same bucket
  - **Amortized**: $O(1)$ considering periodic rehashing
- **Advanced Variants**:
  - **Cuckoo Hashing**: Multiple hash functions, guaranteed $O(1)$ worst-case lookup
  - **Perfect Hashing**: $O(1)$ worst-case for static sets
  - **Bloom Filters**: Space-efficient probabilistic membership testing

8.2 Balanced BSTs: Red-Black Trees

- **Motivation**: Maintain $O(\log n)$ operations while preserving order, unlike hash tables.
- **Red-Black Properties**:
  1. Every node is either red or black
  2. The root is always black
  3. All leaves (NIL) are black
  4. If a node is red, then both its children are black (no two consecutive reds)
  5. Every path from a node to any of its descendant NIL nodes contains the same number of black nodes
- **Consequences of Properties**:
  - **Height Bound**: $h \leq 2\log(n + 1)$ - approximately balanced
  - **Black Height**: Same for all paths, ensures balance
  - **Longest Path**: At most twice the shortest path
- **Rotation Operations**:
  - **Left Rotation**: Make right child the new root
  - **Right Rotation**: Make left child the new root
  - Preserve BST property while changing tree structure

- **Insertion Cases**:
  - **Case 1**: Uncle is red - recolor parent, uncle, grandparent
  - **Case 2**: Uncle is black, node is right child - rotate to make left child
  - **Case 3**: Uncle is black, node is left child - rotate grandparent
  - At most 2 rotations needed for insertion
- **Deletion Cases**:
  - More complex due to ”double black” nodes
  - 4 main cases with symmetric variants
  - May require rotations and recoloring up to the root
- **Comparison with Other Balanced BSTs**:
  - **AVL Trees**: Stricter balance, faster lookups, slower insert/delete
  - **B-Trees**: Better for disk-based systems, higher branching factor
  - **Splay Trees**: Amortized bounds, no balance guarantees, good locality
- **Real-world Use**:
  - **Java**: TreeMap, TreeSet
  - **C++**: std::map, std::set (typically red-black)
  - **Linux Kernel**: Completely fair scheduler uses red-black trees

9 Adversarial Search

9.1 Minimax Algorithm: Optimal Play

- **Problem Setting**: Two-player zero-sum games with perfect information.
- **Key Assumptions**:
  - Both players play optimally
  - Players have opposite objectives (zero-sum)
  - All information is visible to both players
- **Minimax Values**:
  - $V(s)$: Utility of state s for MAX player
  - If s is terminal: $V(s) =$ known utility
  - If MAX to move: $V(s) = \max_{s' \in \text{successors}(s)} V(s')$
  - If MIN to move: $V(s) = \min_{s' \in \text{successors}(s)} V(s')$
- **Algorithm**:
  - Recursively compute minimax values from leaves up
  - MAX chooses move that maximizes minimax value
  - MIN chooses move that minimizes minimax value

- **Complexity**:
  - **Time**: $O(b^m)$ where b = branching factor, m = maximum depth
  - **Space**: $O(bm)$ for DFS implementation
  - **Example**: Chess has $b \approx 35$, $m \approx 100$, making 35^{100} infeasible
- **Evaluation Functions**:
  - Estimate utility of non-terminal states
  - Should correlate with actual probability of winning
  - Often weighted linear functions of features
  - Example: Chess - material advantage, piece activity, king safety
- **Depth-Limited Search**:
  - Search to fixed depth rather than leaves
  - Use evaluation function at depth limit
  - Must handle quiescence - avoid evaluating volatile positions

9.2 Alpha-Beta Pruning: Optimizing Minimax

- **Core Idea**: Prune branches that cannot affect the final decision.
- **Alpha (α)**: Best value MAX can guarantee so far (lower bound)
- **Beta (β)**: Best value MIN can guarantee so far (upper bound)
- **Pruning Conditions**:
  - **MAX node**: Prune if value $\geq \beta$ (MIN won't allow this path)
  - **MIN node**: Prune if value $\leq \alpha$ (MAX has better option)
- **Algorithm**:
  - Initialize: $\alpha = -\infty$, $\beta = +\infty$
  - MAX nodes: Update α, prune if value $\geq \beta$
  - MIN nodes: Update β, prune if value $\leq \alpha$
- **Effectiveness**:
  - **Best Case**: $O(b^{m/2})$ - effectively doubles search depth
  - **Worst Case**: $O(b^m)$ - no pruning occurs
  - **Average Case**: $O(b^{3m/4})$ for reasonable move ordering
- **Move Ordering**:
  - Critical for effective pruning
  - Try best moves first (captures, threats, good positional moves)
  - Use iterative deepening to inform move ordering
  - Killer heuristic: moves that were good in similar positions
- **Enhancements**:

  - **Iterative Deepening**: Search to depth 1, 2, 3, ... using previous results
  - **Transposition Tables**: Cache previously computed positions
  - **Null Move Pruning**: Assume passing is worse than any real move
  - **Quiescence Search**: Extend search until position is stable

- **Real-world Performance**:
  - Modern chess engines search 15-20 plies deep
  - Alpha-beta crucial for making deep search feasible
  - Combined with sophisticated evaluation functions and opening books

10 P vs NP Complexity

10.1 Complexity Classes Overview

- **P (Polynomial Time)**: Decision problems solvable in polynomial time by deterministic Turing machines.
  - Examples: Sorting, shortest path, minimum spanning tree
  - Considered "efficiently solvable" in practice
- **NP (Nondeterministic Polynomial Time)**: Decision problems where "yes" answers can be verified in polynomial time.
  - Examples: SAT, traveling salesman, graph coloring
  - Can be solved in polynomial time by nondeterministic Turing machines
- **NP-complete**: The hardest problems in NP. If any NP-complete problem is in P, then P = NP.
  - Examples: SAT, 3-SAT, vertex cover, Hamiltonian path
  - All NP-complete problems are polynomially reducible to each other
- **NP-hard**: Problems at least as hard as NP-complete problems, but not necessarily in NP.
  - Examples: Halting problem, chess optimal play
  - May be harder than NP-complete
- **The P vs NP Question**: Is every problem that can be verified quickly also solvable quickly?
  - One of the seven Millennium Prize Problems
  - Most experts believe P $\neq$ NP
  - Would have profound implications for cryptography, optimization, AI

10.2 Practical Implications

- **Algorithm Design Strategy**:
  - First try to find polynomial-time algorithm
  - If problem appears hard, check if it's NP-complete
  - For NP-complete problems, consider:

    * Approximation algorithms
    * Heuristics and local search
    * Parameterized algorithms
    * Exact algorithms for small instances

- **Approximation Algorithms**:
  - Polynomial-time algorithms with guaranteed solution quality
  - Example: 2-approximation for vertex cover
  - Some problems have polynomial-time approximation schemes (PTAS)
- **Parameterized Complexity**:
  - Analyze complexity in terms of input size and additional parameter
  - Example: $O(2^k n)$ for vertex cover with parameter k = solution size
  - Fixed-parameter tractable (FPT) if $O(f(k) \cdot n^c)$
- **Heuristics and Metaheuristics**:
  - No guarantees but work well in practice
  - Genetic algorithms, simulated annealing, tabu search
  - Often the only practical approach for large instances

Algorithm Selection Guide

Decision Framework

- **Understand the Problem Constraints**:
  - Input size: Small ($n < 50$), Medium ($50 \leq n \leq 10^4$), Large ($n > 10^4$)
  - Time constraints: Real-time, interactive, batch processing
  - Space constraints: Memory-limited vs compute-limited
  - Accuracy requirements: Exact vs approximate solutions
- **Analyze the Data Characteristics**:
  - Sorted vs unsorted
  - Random access vs sequential access
  - Static vs dynamic (frequent updates)
  - Distribution: Uniform, skewed, clustered
- **Consider the Operations Needed**:
  - Search-heavy vs insert/delete-heavy
  - Point queries vs range queries
  - Need ordering or not
  - Concurrent access requirements

Quick Reference Table

Scenario	Recommended Approach	Time Complexity	Key Considerations
Small dataset	Quadratic sorts	$O(n^2)$	Simple, low constant factors
Large dataset sorting	Merge Sort / Quick Sort	$O(n \log n)$	Merge Sort stable, Quick Sort faster
Fast search, no ordering	Hash Table	$O(1)$ avg	No ordering, worst case $O(n)$
Ordered data, range queries	Balanced BST	$O(\log n)$	Maintains order, range operations
Optimization with optimal substructure	Dynamic Programming	Problem-dependent	Identify overlapping subproblems
Optimization with greedy choice	Greedy Algorithm	Usually $O(n \log n)$	Verify greedy choice property
Game playing	Minimax + Alpha-Beta	$O(b^{m/2})$	Evaluation function critical
Pathfinding with estimates	A* Search	$O(b^{\epsilon d})$	Need admissible heuristic
NP-complete problems	Approximation + Heuristics	Varies	Trade optimality for tractability
Geometric problems	Divide and Conquer	Often $O(n \log n)$	Exploit spatial properties

Hybrid Approaches

- **Timsort**: Merge Sort + Insertion Sort - Python's built-in sort
- **Introsort**: Quick Sort + Heap Sort - C++ std::sort
- **B-trees**: BST + arrays - database indices, file systems
- **Bloom filters**: Hashing + probability - network routers, databases

Final Principles: Developing Algorithmic Intuition

The Mindset of an Algorithm Designer

- **Think in Trade-offs**: Every design decision has consequences. Understand what you're optimizing for and what you're sacrificing.
- **Embrace Asymptotic Thinking**: Focus on how algorithms scale. A "slow" $O(n \log n)$ algorithm will eventually beat a "fast" $O(n^2)$ algorithm.
- **Look for Patterns**: Most new problems resemble classic ones. Learn to recognize when to apply divide-and-conquer, dynamic programming, or greedy strategies.
- **Understand the Data**: The right data structure can make an intractable problem tractable. Choose based on the operations you need, not just what's familiar.
- **Prove Your Solutions**: Don't rely on intuition alone. Use mathematical reasoning to verify correctness and optimality.
- **Consider the Constants**: While asymptotic analysis is primary, constant factors matter in

practice. Profile and optimize hot paths.

- **Know the Limits**: Understand computational complexity theory. Recognize when you're facing an NP-hard problem and adjust your expectations accordingly.
- **Iterate and Refine**: Start with a simple solution, then optimize. Premature optimization is the root of much evil, but thoughtful optimization is the essence of good engineering.

Continuous Learning

- **Study Classic Algorithms**: Understand why they work and their historical context
- **Analyze Real-world Systems**: Look at how algorithms are used in databases, operating systems, compilers
- **Practice Problem Solving**: Regular practice develops intuition and pattern recognition
- **Read Research Papers**: Stay current with new developments in algorithms
- **Implement and Experiment**: Theory informs practice, but practice deepens theoretical understanding