Algorithm Design Techniques - Comprehensive Summary

A reference guide for memorizing algorithms, their structures, complexity analysis, and implementation patterns.

Table of Contents

BRUTE FORCE & BASIC OPERATIONS

1. Digit Operations

Problems: Sum of digits, reverse digits, digit set, count binary 1s

Generic Structure

```
While number > 0:
    Extract last digit using modulo (%)
    Perform operation
    Remove digit using integer division (//)
```

Complexity Analysis

Operation	Time	Space
Sum digits	Θ(log n)	Θ(1)
Reverse digits	Θ(log n)	Θ(1)
Digit set	Θ(log n)	Θ(log n)

Binary 1s count	Θ(log n)	Θ(1)

Key Insight: All operations iterate through digits, so complexity is proportional to the number of digits (log n).

DECREASE & CONQUER

2. Decrease by One: Exponentiation (Power)

Problem: Calculate x^n efficiently

Generic Structure

```
power(x, n):
    Base case: if n = 0, return 1
    Recursive case: p = power(x, n/2)
        If n is even: return p * p
        If n is odd: return p * p * x
```

Complexity Analysis

Metric	Value
Time	**Θ(log n)**
Space	**Θ(log n)**
Recurrence	T(n) = T(n/2) + 1

Key Insight: Dividing exponent by 2 each time → logarithmic complexity.

---

3. Sorting Algorithms

A) Bubble Sort

Concept: Compare adjacent elements, swap if wrong order. Larger elements bubble to end.

```
for i = 0 to n-1:
    for j = 0 to n-i-2:
        if arr[j] > arr[j+1]:
            swap(arr[j], arr[j+1])
```

Metric	Value

Time	**Θ(n²)** (all cases)
Space	**Θ(1)**
Stable	Yes
In-place	Yes

B) Selection Sort

Concept: Find minimum in unsorted part, place at front. Repeat.

```
for i = 0 to n-1:
    min_index = i
    for j = i+1 to n-1:
        if arr[j] < arr[min_index]:
            min_index = j
    swap(arr[i], arr[min_index])
```

Metric	Value
Time	**$\Theta(n^2)$** (all cases)
Space	**$\Theta(1)$**
Stable	No
In-place	Yes

C) Insertion Sort

Concept: Build sorted array incrementally by inserting each element into correct position.

```
for i = 1 to n-1:
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key:
        arr[j+1] = arr[j]
        j -= 1
    arr[j+1] = key
```

Metric	Best	Average	Worst
Time	**$\Theta(n)$**	**$\Theta(n^2)$**	**$\Theta(n^2)$**
Space	**$\Theta(1)$**	**$\Theta(1)$**	**$\Theta(1)$**
Stable	Yes		
In-place	Yes		

Invariant: After iteration i, subarray arr[0..i] is sorted.

DIVIDE & CONQUER

4. Binary Search

Problem: Find element in sorted array

Concept: Split search space in half at each step. Only one subproblem solved (reduction).

```
binary_search(arr, x):
    left = 0, right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x: return True
        elif x < arr[mid]: right = mid - 1
        else: left = mid + 1
    return False
```

Metric	Value
Time	**Θ(log n)**
Space	**Θ(1)**
Recurrence	$T(n) = T(n/2) + 1$

Invariant: If x exists, it's always in [left, right].

---

5. Merge Sort

Problem: Sort array

Concept: Split → Sort recursively → Merge sorted halves

```
merge_sort(arr):
    if len(arr) <= 1: return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)
```

```
merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:] + right[j:]
    return result
```

Metric	Value
Time	**Θ(n log n)** (all cases)
Space	**Θ(n)**
Stable	Yes
In-place	No
Recurrence	$T(n) = 2T(n/2) + n$

---

6. Quick Sort

Problem: Sort array

Concept: Choose pivot → Partition → Sort partitions recursively

```
partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j = low to high-1:
        if arr[j] <= pivot:
            i += 1
            swap(arr[i], arr[j])
    swap(arr[i+1], arr[high])
    return i + 1

quick_sort(arr, low, high):
    if low < high:
        p = partition(arr, low, high)
        quick_sort(arr, low, p-1)
        quick_sort(arr, p+1, high)
```

Metric	Best	Average	Worst
Time	**Θ(n log n)**	**Θ(n log n)**	**$\Theta(n^2)$**
Space	**Θ(log n)**	**Θ(log n)**	**Θ(n)**
Stable	No		
In-place	Yes		
Recurrence	T(n) = T(k) + T(n-k-1) + n		

Note: Worst case ($\Theta(n^2)$) occurs when pivot always picks min/max.

---

7. Triomino (Divide & Conquer)

Problem: Cover $2^n \times 2^n$ board with one missing square using L-shaped triominoes

Concept:

- Divide board into 4 quadrants
- Place one triomino at center to create "missing square" in other 3 quadrants
- Recursively solve 4 smaller boards

```
triomino(board, size, top, left, missing_r, missing_c):
    if size == 2:
        # Base case: place one triomino
        return

    half = size // 2
    # Determine which quadrant has missing square
    # Place triomino at center (covers 3 quadrants)
    # Recursively solve 4 quadrants with new missing squares
```

Metric	Value
Time	**$\Theta(n^2)$** (cover all n^2 squares)
Space	**$\Theta(n^2)$**
Recurrence	T(n) = 4T(n/2) + O(1)

BACKTRACKING

Core Pattern:

```
backtrack(position, constraints):
    if position == solution_length:
```

```
        add_solution()
        return

    for each choice in choices:
        if is_valid(choice):
            make_choice()
            backtrack(position + 1, constraints)
            undo_choice()  # CRITICAL: backtrack
```

8. Permutations

Problem: Generate all permutations of a set

```
backtrack(position, n, numbers, current, results):
    if position == n:
        results.append(current.copy())
        return

    for num in numbers:
        if num not in current:  # Check: not used yet
            current[position] = num
            backtrack(position + 1, n, numbers, current, 
results)
            current[position] = None  # Backtrack
```

Metric	Value
Time	**Θ(n!)**
Space	**Θ(n)**
Solutions	n! permutations

---

9. Subsets (Power Set)

Problem: Generate all subsets of a set

```
backtrack(start_index, n, numbers, current, results):
    results.append(current.copy())  # Every partial is valid 
subset

    for i = start_index to n-1:
        current.append(numbers[i])
        backtrack(i + 1, n, numbers, current, results)
        current.pop()  # Backtrack
```

Metric	Value
Time	**$\Theta(n \times 2^n)$** (2^n subsets, each takes O(n))
Space	**$\Theta(2^n)$** (store all subsets)
Solutions	2^n subsets

Key Insight: No validity check needed; all partial solutions are valid subsets.

---

10. N-Queens Problem

Problem: Place N queens on N×N board so none attack each other

Constraints: No two queens in same row, column, or diagonal

```
check_partial_option(board, row, col):
    for prev_row = 0 to row-1:
        prev_col = board[prev_row]
        if prev_col == col:  # Same column
            return False
        if abs(prev_row - row) == abs(prev_col - col):  # Same diagonal
            return False
    return True

backtrack(row, n, board, solutions):
    if row == n:
        solutions.append(board.copy())
        return

    for col = 0 to n-1:
        if check_partial_option(board, row, col):
            board[row] = col
            backtrack(row + 1, n, board, solutions)
            board[row] = -1  # Backtrack
```

Metric	Value
Time	**O(N!)** worst case (pruning helps)
Space	**$\Theta(N)$** (board + recursion stack)
Solutions	Number of valid N-queens configurations

---

11. Sudoku Solver

Problem: Fill 9×9 grid with digits 1-9 respecting row, column, and 3×3 box constraints

```
check(board, row, col, num):
    # Check row
    for c = 0 to 9: if board[row][c] == num: return False
    # Check column
    for r = 0 to 9: if board[r][col] == num: return False
    # Check 3x3 box
    box_r = (row // 3) * 3
    box_c = (col // 3) * 3
    for r = box_r to box_r+2:
        for c = box_c to box_c+2:
            if board[r][c] == num: return False
    return True

backtrack(board):
    empty = find_empty_cell(board)
    if empty is None: return True  # Solved

    row, col = empty
    for num = 1 to 9:
        if check(board, row, col, num):
            board[row][col] = num
            if backtrack(board): return True
            board[row][col] = 0  # Backtrack
    return False
```

Metric	Value
Time	**O(9^(empty cells))** worst case (heavy pruning)
Space	**Θ(1)** (solve in-place)
Optimizations	Check constraints before recursing

DYNAMIC PROGRAMMING

Core Pattern:

```
DP bottom-up approach:
    1. Create DP table (array/matrix)
    2. Initialize base cases
    3. Fill table iteratively using recurrence relation
    4. Reconstruct solution from DP table
```

12. Fibonacci Sequence

Problem: Calculate nth Fibonacci number

Naive Recursive (BAD)

```
fibonacci(n):
    if n <= 2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Metric	Value
Time	**$\Theta(\phi^n)$** where $\phi \approx 1.618$
Space	**$\Theta(n)$** (recursion depth)
Issue	Redundant subproblems

DP Solution (GOOD)

```
fibonacci_dp(n):
    if n <= 2: return 1
    dp = [0] * (n + 1)
    dp[1] = dp[2] = 1
    for i = 3 to n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Metric	Value
Time	**$\Theta(n)$**
Space	**$\Theta(n)$**

---

13. Coin Change (Minimum Coins)

Problem: Find minimum number of coins to form target sum

Concept: Build solutions for small sums first, use for larger sums

```
min_coins_dp(S, coins):
    best = [∞] * (S + 1)  # best[x] = min coins for sum x
    last = [-1] * (S + 1)  # track which coin was used

    best[0] = 0
```

```
    for s = 1 to S:
        for c in coins:
            if s - c >= 0 and best[s - c] + 1 < best[s]:
                best[s] = best[s - c] + 1
                last[s] = c

    # Reconstruct solution
    used = []
    s = S
    while s > 0:
        used.append(last[s])
        s -= last[s]

    return best[S], used
```

Metric	Value
Time	**Θ(S × n)** where n = # coins
Space	**Θ(S)**
Recurrence	dp[s] = min(dp[s-c] + 1) for all coins c

Key Insight: DP guarantees optimality; greedy may fail for some coin sets.

---

14. Edit Distance (Levenshtein Distance)

Problem: Minimum edits (insert, delete, replace) to transform one string to another

Concept: dp[i][j] = min edits for transforming a[0..i-1] to b[0..j-1]

```
edit_distance(a, b):
    n = len(a), m = len(b)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    # Base cases
    for i = 0 to n: dp[i][0] = i  # Delete all
    for j = 0 to m: dp[0][j] = j  # Insert all

    # Fill table
    for i = 1 to n:
        for j = 1 to m:
            if a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
```

```
                dp[i][j] = 1 + min(
                    dp[i-1][j],      # Delete
                    dp[i][j-1],      # Insert
                    dp[i-1][j-1]     # Replace
                )

    return dp[n][m]
```

Metric	Value
Time	**Θ(n × m)**
Space	**Θ(n × m)**
Recurrence	dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

GREEDY TECHNIQUE

Core Pattern:

```
greedy():
    while solution not complete:
        select locally optimal choice
        add to solution
```

15. Activity Selection

Problem: Select maximum number of non-overlapping activities

Concept: Greedily pick activities ending earliest

```
activity_selection(start, end):
    # Sort by end time
    sort_by_end_time(start, end)

    selected = []
    last_end = -1

    for i = 0 to n-1:
        if start[i] >= last_end:
            selected.append((start[i], end[i]))
            last_end = end[i]

    return selected
```

Metric	Value
Time	**Θ(n log n)** (sorting)
Space	**Θ(1)**
Optimality	YES (greedy-choice property)

Greedy Choice: Always pick activity ending earliest. Leaves room for more activities.

---

16. Fractional Knapsack

Problem: Maximize value with weight capacity (can take fractions)

Concept: Take items by value/weight ratio, highest first

```
fractional_knapsack(items, capacity):
    # items = [value, weight]
    # Compute value/weight ratio
    ratios = [[value/weight, value, weight] for value, weight
in items]

    # Sort by ratio descending
    ratios.sort(reverse=True)

    total_value = 0
    for ratio, value, weight in ratios:
        if weight <= capacity:
            total_value += value
            capacity -= weight
        else:
            total_value += value * (capacity / weight)
            break

    return total_value
```

Metric	Value
Time	**Θ(n log n)** (sorting)
Space	**Θ(n)**
Optimality	YES (can take fractions)

Key Difference from 0/1 Knapsack: Fractional version is greedy-optimal; 0/1 version requires DP.

---

17. Greedy Coin Change

Problem: Find coins to form sum (greedily)

Concept: Always pick largest coin ≤ remaining sum

```
greedy_coin_change(S, coins):
    coins.sort(reverse=True)
    used = []

    for c in coins:
        while S >= c:
            S -= c
            used.append(c)

    return used if S == 0 else None
```

Metric	Value
Time	**$\Theta(n)$** where n = # coins
Space	**$\Theta(1)$**
Optimality	NO (fails for some coin sets)

Example where it fails:

- Coins: [1, 3, 4], Sum: 6
- Greedy: 4 + 1 + 1 = 3 coins
- Optimal: 3 + 3 = 2 coins

Lesson: Greedy is fast but not always optimal. Use DP for correctness guarantee.

BIG O COMPLEXITY SUMMARY

Time Complexity Classes (from fastest to slowest)

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

```
Constant  Log    Linear  Linearithmic  Quadratic Cubic
Exponential Factorial
```

Common Algorithm Complexities

Algorithm	Time (Best)	Time (Avg)	Time (Worst)	Space
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(\log n)$
Binary Search	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Permutations	$\Theta(n!)$	$\Theta(n!)$	$\Theta(n!)$	$\Theta(n)$
Subsets	$\Theta(2^n)$	$\Theta(2^n)$	$\Theta(2^n)$	$\Theta(2^n)$
Fibonacci (DP)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Coin Change DP	$\Theta(S \cdot n)$	$\Theta(S \cdot n)$	$\Theta(S \cdot n)$	$\Theta(S)$
Edit Distance	$\Theta(n \cdot m)$	$\Theta(n \cdot m)$	$\Theta(n \cdot m)$	$\Theta(n \cdot m)$
Activity Selection	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$
Greedy Coin	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

KEY MEMORIZATION TIPS

Design Technique Selection

1. **Brute Force** → Simple problems, constraints allow enumeration
2. **Decrease & Conquer** → Reduce problem size by fixed fraction (exponentiation)
3. **Divide & Conquer** → Split equally, solve recursively, combine (sorting, search)
4. **Backtracking** → Find ALL solutions, explore with constraint pruning
5. **Dynamic Programming** → Overlapping subproblems, optimal substructure
6. **Greedy** → Locally optimal choice is globally optimal, fast but risky

Red Flags

- **Exponential Time ($\Theta(n!)$, $\Theta(2^n)$, $\Theta(\phi^n)$)**: Usually backtracking or naive recursion
- **Quadratic Time ($\Theta(n^2)$)**: Simple nested loops (bubble, selection, insertion)
- **Logarithmic Time ($\Theta(\log n)$)**: Divide by constant (binary search, power)
- **Linear Time ($\Theta(n)$)**: Single loop through data
- **N log N ($\Theta(n \log n)$)**: Efficient sorting (merge, quick, heap)

Space vs Time Trade-offs

- **DP trades space for time** (memoization: store results, avoid recomputation)
- **Backtracking uses recursion stack** (space = depth of recursion tree)

- **Divide & conquer often uses extra space** (merge sort creates new arrays)
- **In-place algorithms** preserve space (bubble, selection, insertion, quick sort)

Practice Algorithm Matrix

Category	Technique	Problems to Solve
Sorting	Decrease & Conquer	Bubble, Selection, Insertion
Searching	Divide & Conquer	Binary Search
Efficient Sorting	Divide & Conquer	Merge Sort, Quick Sort
Optimization	Greedy	Activity Selection, Fractional Knapsack
Optimization	Dynamic Programming	Coin Change, Edit Distance
Combinatorial	Backtracking	Permutations, N-Queens, Sudoku
Covering	Divide & Conquer	Triomino
Digit Ops	Brute Force	Sum, Reverse, Binary Count

Last Updated: January 19, 2026