Advanced Algorithms Study Guide - Homework 1

1. Incremental Sorting Algorithm (Bubble Sort)

Concept: Sorting Algorithms

Sorting algorithms are procedures used to arrange elements in a specific order, typically ascending or descending. They are fundamental in computer science and are used in many applications.

Problem:

Design an algorithm that sorts an array A of size n incrementally by repeatedly finding the largest unsorted element and placing it in its correct position.

Solution Approach:

This algorithm is known as Bubble Sort. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order.

Pseudocode:

```
BubbleSort(A):
    n = length(A)
    for i = 1 to n-1
        for j = 1 to n-i
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
```

Concept: Loop Invariant

A loop invariant is a condition that is true before and after each iteration of a loop. It helps in understanding and proving the correctness of an algorithm.

Loop Invariant for Bubble Sort:

"After each iteration j of the outer loop, the last j elements of the array are in their final sorted positions."

Number of Iterations:

The outer loop runs n-1 times, where n is the size of the array.

Explanation:

1. In each iteration of the outer loop, the largest unsorted element "bubbles up" to its correct position.
2. After the first iteration, the largest element is at the end of the array.
3. After the second iteration, the second-largest element is in the second-to-last position.
4. This continues until the entire array is sorted.

2. Fibonacci Sequence and Mathematical Induction

Concept: Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. It usually starts with $F_1 = 1$ and $F_2 = 1$.

Concept: Mathematical Induction

Mathematical induction is a method of mathematical proof typically used to establish that a given statement is true for all natural numbers. It consists of two steps:

1. Base case: Prove that the statement holds for the first natural number (usually 1 or 0).
2. Inductive step: Prove that if the statement holds for a number k, then it must also hold for k+1.

Problem:

Prove that $F_1^2 + F_2^2 + ... + F_n^2 = F_n \cdot F_{n+1}$, where F_k is the kth Fibonacci number.

Proof by Induction:

1. Base case (n = 2): $F_1^2 + F_2^2 = 1^2 + 1^2 = 2 = 1 \cdot 2 = F_2 \cdot F_3$

2. Inductive hypothesis: Assume $F_1^2 + F_2^2 + ... + F_k^2 = F_k \cdot F_{k+1}$ is true for all $k < n$.

3. Inductive step: We need to prove $F_1^2 + F_2^2 + ... + F_n^2 = F_n \cdot F_{n+1}$

   $F_1^2 + F_2^2 + ... + F_{n-1}^2 + F_n^2 = (F_1^2 + F_2^2 + ... + F_{n-1}^2) + F_n^2 = F_{n-1} \cdot F_n + F_n^2$ (using the inductive hypothesis) $= F_n \cdot (F_{n-1} + F_n) = F_n \cdot F_{n+1}$ (by definition of Fibonacci numbers)

Thus, the statement is proved for all natural numbers $n \geq 2$.

3. Proof by Contradiction

Concept: Proof by Contradiction

Proof by contradiction is a logical argument where we assume the opposite of what we want to prove, and then show that this assumption leads to a logical impossibility (contradiction). This implies that our original statement must be true.

Problem:

Prove that there is no greatest negative rational number.

Proof:

1. Assume the opposite: there exists a greatest negative rational number, call it x.
2. Express x as a fraction: $x = -a/b$, where a and b are positive integers.
3. Consider a new number $y = -(a-1)/b$.
4. Observe that y is also a negative rational number.
5. Compare x and y: $x = -a/b < -(a-1)/b = y < 0$
6. This shows that y is a negative rational number greater than x.
7. But this contradicts our assumption that x was the greatest negative rational number.
8. Therefore, our initial assumption must be false.

Conclusion: There is no greatest negative rational number.

4. Asymptotic Notation

Concept: Asymptotic Notation

Asymptotic notation is used to describe the running time or space requirements of algorithms as the input size approaches infinity. It provides an upper bound, lower bound, or tight bound on the growth of a function.

Types of Asymptotic Notation:

- O (Big O): Upper bound
- Ω (Omega): Lower bound
- Θ (Theta): Tight bound
- o (Little o): Upper bound that is not tight
- ω (Little omega): Lower bound that is not tight

Properties:

1. o ⇒ O and ω ⇒ Ω: If $f(n) = o(g(n))$, then $f(n) = O(g(n))$. If $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$. But the reverse is not always true.

2. O ⇒ Θ is false: Example: $n^2 = O(n^3)$, but $n^2 \neq \Theta(n^3)$

3. $f(n) = \Omega(f(n))$ for every function f: Proof: $f(n) \geq (1/2) \cdot f(n)$ for all $n \geq 1$. Choose $c = 1/2$ and $n_0 = 1$.

4. $f(n) = O(f(n))$ for every function f: Proof by contradiction: Assume $f(n) \neq O(f(n))$ for some f. But $f(n) \leq 1 \cdot f(n)$ for all $n \geq 1$, satisfying the definition of O. This contradicts our assumption, so $f(n) = O(f(n))$ must be true.

5. Analysis of Insertion Sort on Reverse-Sorted Array

Concept: Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It's much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Problem:

Analyze the number of comparisons made by Insertion Sort when sorting an array A that is initially sorted in decreasing order.

Analysis:

1. In Insertion Sort, for each element (except the first), we compare it with the previous elements until we find its correct position.
2. For an array sorted in decreasing order, each element will be compared with all previous elements.
3. Number of comparisons:
   - For 2nd element: 1 comparison
   - For 3rd element: 2 comparisons
   - For 4th element: 3 comparisons ...
   - For nth element: n-1 comparisons
4. Total comparisons: $1 + 2 + 3 + ... + (n-1) = n(n-1)/2$

Conclusion:

The number of comparisons is $\Theta(n^2)$, which is $O(n^2)$ in big O notation.

This worst-case scenario occurs because each element needs to be moved to the beginning of the array, requiring comparisons with all previously sorted elements.

Understanding these concepts and problem-solving approaches will help you tackle similar problems in advanced algorithm analysis and design. Remember to practice applying these techniques to different scenarios to strengthen your skills.

Advanced Algorithms Study Guide - Homework 2

1. Solving Recurrence Relations Using the Master Theorem

Problem Type:

Solve recurrence relations of the form $T(n) = aT(n/b) + f(n)$.

Approach:

1. Identify a, b, and f(n) in the recurrence relation.
2. Calculate $n^{\log_b(a)}$.
3. Compare f(n) with $n^{\log_b(a)}$.
4. Apply the appropriate case of the Master Theorem.

Cases of the Master Theorem:

- Case 1: If $f(n) = O(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$.
- Case 2: If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} * \log n)$.
- Case 3: If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

Example:

For $T(n) = T(8n/11) + n$:

1. $a = 1$, $b = 11/8$, $f(n) = n$
2. $n^{\log_{(11/8)}(1)} = n^0 = 1$
3. $f(n) = n$, which is larger than n^0
4. Case 3 applies
5. Verify $af(n/b) \leq cf(n)$: $1 * (8n/11) \leq cn$ implies $8/11 \leq c < 1$
6. Therefore, $T(n) = \Theta(n)$

How to Apply to Similar Problems:

1. Always start by identifying a, b, and f(n).
2. Calculate n^(log_b(a)) and compare it with f(n).
3. Be careful with the base of the logarithm in n^(log_b(a)).
4. For case 3, don't forget to verify the additional condition af(n/b) ≤ cf(n).
5. Practice with various forms of f(n) to get comfortable with the comparisons.

2. Big O Notation Proofs

Problem Type:

Prove or disprove statements involving Big O notation.

Approach:

1. Understand the definition of Big O: f(n) = O(g(n)) if there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0.
2. For disproving, use contradiction: assume the statement is true and show it leads to a contradiction.
3. For proving, find appropriate constants c and n0 that satisfy the definition.

Example:

To disprove 5^(2n) = O(5^n):

1. Assume it's true: 5^(2n) ≤ c * 5^n for some c and all large n
2. Divide both sides by $5^{n:\ 5}$n ≤ c
3. This is a contradiction because 5^n grows without bound as n increases

How to Apply to Similar Problems:

1. Always start with the definition of Big O.
2. For disproving, try to show that no constant c can satisfy the definition for all large n.
3. For proving, try to find a specific c and n0 that work.
4. Be careful with very close comparisons (like $n^{2.0000000001 \text{ vs } n}$2) - small differences can be significant.
5. Use algebra and properties of exponents to simplify expressions when necessary.

3. Algorithm Design for Array Problems

Problem Type:

Design algorithms to check properties of arrays.

Approach:

1. Clearly define the input and output.
2. Identify the key property to check.
3. Design a simple algorithm that checks this property efficiently.

Example: Checking if an array is unsorted

Input: Array A[1...n] Output: "Yes" if unsorted, "No" if sorted

Algorithm:

```
IsUnsorted(A[p, ..., q])  // Initial call p = 1 and q = n
    for i = p to q
        for j = p to q
            if A[i] > A[j]
                return "Yes"
    return "No"
```

How to Apply to Similar Problems:

1. Always start by clearly defining the input and output.
2. Think about the simplest way to check the required property.
3. Consider edge cases (e.g., empty array, single element array).
4. For efficiency, try to avoid unnecessary comparisons or iterations.
5. If the problem involves a specific subset of elements (like even-indexed), adjust your loop accordingly.

4. Divide and Conquer for Integer Multiplication

Problem Type:

Design a divide and conquer algorithm for a mathematical operation.

Approach:

1. Define the input and output clearly.
2. Identify the base case (simplest version of the problem).
3. Divide the problem into smaller subproblems.
4. Solve the subproblems recursively.
5. Combine the solutions to subproblems to solve the original problem.

Example: Multiplying two n-digit integers

1. Input: X = x1x2...xn and Y = y1y2...yn (n-digit integers)
2. Output: X * Y
3. Base case: If n = 1, return x1 * y1
4. Divide: Split X and Y into two halves each
5. Conquer: Recursively multiply the halves
6. Combine: Use the formula X*Y = X1*Y1*10^n + (X1*Y2 + X2*Y1)*10^(n/2) + X2*Y2

How to Apply to Similar Problems:

1. Always start by clearly defining the input, output, and base case.
2. Think about how to break the problem into smaller, similar subproblems.
3. Determine how to combine solutions to subproblems efficiently.
4. Write out the recurrence relation for the running time.
5. Consider the trade-offs between the number of subproblems and the complexity of combining their solutions.

5. Proving Time Complexity Using Substitution Method

Problem Type:

Prove the time complexity of a recurrence relation using the substitution method.

Approach:

1. Guess the solution (usually based on intuition or experience).
2. Assume the guess is true for smaller input sizes (inductive hypothesis).
3. Prove it's true for the current size using the recurrence and the inductive hypothesis.
4. Verify the base case(s) if necessary.

Example: Proving T(n) = 4T(n/2) + n = O(n^2)

1. Guess: T(n) ≤ cn^2 - dn for some constants c and d
2. Inductive step: $T(n) = 4T(n/2) + n \le 4(c(n/2)^{2 - d(n/2)) + n = cn}2 - 2dn + n = cn^{2 - dn - (d-1)n \le cn}2 - dn$ if $d > 1$
3. Base case: Usually skipped in these proofs, but should be verified for small n

How to Apply to Similar Problems:

1. Start with a reasonable guess based on the form of the recurrence.
2. If your first guess doesn't work, try a more specific form (like adding or subtracting a lower-order term).
3. In the inductive step, always substitute the guess for the recursive terms.

4. Be careful with algebra and simplification.
5. Make sure your final inequality matches your original guess.
6. If you can't make it work, consider a different guess or a different proof technique.

Remember, practice is key to mastering these techniques. Try to solve many similar problems to build your intuition and problem-solving skills.

Algorithms Study Guide: Chapters 1-5

1. Insertion Sort

Pseudocode:

```
INSERTION-SORT(A)
    for i = 2 to A.length
        key = A[i]
        // Insert A[i] into the sorted subarray A[1..i-1]
        j = i - 1
        while j > 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Explanation:

Insertion Sort builds the final sorted array one item at a time. It iterates through an input array and removes one element per iteration, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Time Complexity: O(n^2) for worst and average cases, O(n) for best case (already sorted)

2. Merge Sort

Pseudocode:

```
MERGE-SORT(A, p, r)
    if p < r
        q = ⌊(p + r) / 2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)

MERGE(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1] and R[1..n2] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    i = 1
    j = 1
    k = p
    while i ≤ n1 and j ≤ n2
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        k = k + 1
    while i ≤ n1
        A[k] = L[i]
        i = i + 1
        k = k + 1
    while j ≤ n2
        A[k] = R[j]
        j = j + 1
        k = k + 1
```

Explanation:

Merge Sort is a divide-and-conquer algorithm. It divides the input array into two halves, recursively sorts them, and then merges the two sorted halves. The MERGE procedure is key to the algorithm, efficiently combining two sorted subarrays into a single sorted subarray.

Time Complexity: O(n log n) for all cases

3. Strassen's Algorithm for Matrix Multiplication

Pseudocode:

```
STRASSEN(A, B)
    n = A.rows
    if n == 1
        return A[1,1] * B[1,1]
    else
        // Divide matrices into submatrices
        A11, A12, A21, A22 = divide_matrix(A)
        B11, B12, B21, B22 = divide_matrix(B)

        // Compute seven products
        P1 = STRASSEN(A11 + A22, B11 + B22)
        P2 = STRASSEN(A21 + A22, B11)
        P3 = STRASSEN(A11, B12 - B22)
        P4 = STRASSEN(A22, B21 - B11)
        P5 = STRASSEN(A11 + A12, B22)
        P6 = STRASSEN(A21 - A11, B11 + B12)
        P7 = STRASSEN(A12 - A22, B21 + B22)

        // Compute the resulting submatrices
        C11 = P1 + P4 - P5 + P7
        C12 = P3 + P5
        C21 = P2 + P4
        C22 = P1 - P2 + P3 + P6

        // Combine submatrices into result
        return combine_matrices(C11, C12, C21, C22)
```

Explanation:

Strassen's algorithm is used for matrix multiplication. It's faster than the standard matrix multiplication algorithm for large matrices. The algorithm recursively divides the matrices into submatrices and computes seven products, which are then combined to produce the final result.

Time Complexity: O(n$^{2.807)\text{, which is better than the O(n}}$3) of standard matrix multiplication

4. Randomized Quick Sort

Pseudocode:

```
RANDOMIZED-QUICKSORT(A, p, r)
    if p < r
        q = RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q - 1)
        RANDOMIZED-QUICKSORT(A, q + 1, r)

RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)

PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

Explanation:

Randomized Quick Sort is a variation of Quick Sort that randomly selects the pivot element. This randomization helps to avoid worst-case scenarios that can occur with deterministic pivot selection. The algorithm partitions the array around the pivot, then recursively sorts the two partitions.

Expected Time Complexity: $O(n \log n)$

5. Counting Sort

Pseudocode:

```
COUNTING-SORT(A, B, k)
    let C[0..k] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to i
    for i = 1 to k
        C[i] = C[i] + C[i - 1]
    // C[i] now contains the number of elements ≤ i
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Explanation:

Counting Sort is a non-comparison-based sorting algorithm. It works by counting the number of objects having distinct key values, then calculating the position of each object in the output sequence. It's efficient when the range of potential items (k) is not significantly greater than the number of items (n).

Time Complexity: O(n + k), where n is the number of elements and k is the range of input

6. Randomized Hiring Algorithm

Pseudocode:

```
RANDOMIZED-HIRE-ASSISTANT(n)
    randomly permute the list of candidates
    best = 0  // Candidate 0 is a least-qualified dummy candidate
    for i = 1 to n
        interview candidate i
        if candidate i is better than candidate best
            best = i
            hire candidate i
```

Explanation:

This algorithm is used to solve the hiring problem. Instead of interviewing candidates in a fixed order, it first randomly permutes the list of candidates. This randomization helps to improve the expected cost of the hiring process.

Expected Cost: O(ch log n), where ch is the cost of hiring and n is the number of candidates

These algorithms represent the core content from chapters 1-5 of the textbook. Understanding their pseudocode, how they work, and their time complexities will be crucial for your exam. Remember to also review the analysis techniques (like the master method for solving recurrences) and the concepts of probabilistic analysis and randomized algorithms.

Asymptotic Notation: A Comprehensive Guide

Asymptotic notation is a mathematical tool used in computer science to describe the behavior of functions as their input grows large. It's essential for analyzing algorithm efficiency and comparing algorithms. The main types of asymptotic notation are Big O, Omega, and Theta.

1. Big O Notation (O)

Definition:

O(g(n)) = { f(n) : there exist positive constants c and n0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n0$ }

Intuition:

Big O provides an upper bound on the growth rate of a function. It says that f(n) grows no faster than g(n).

Example:

If f(n) = $3n^{2 + 2n + 1, \text{ then } f(n) = O(n}$2) because for large n, the n^2 term dominates.

How to use in exams:

1. Identify the dominant term as n grows large.
2. Ignore constant factors and lower-order terms.
3. Express the result as O(dominant term).

2. Omega Notation (Ω)

Definition:

Ω(g(n)) = { f(n) : there exist positive constants c and n0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n0$ }

Intuition:

Omega provides a lower bound on the growth rate of a function. It says that f(n) grows at least as fast as g(n).

Example:

If f(n) = 3n$^{\text{2 + 2n + 1, then f(n) = Ω(n}}$2) because f(n) is always greater than some constant times n^2 for large n.

How to use in exams:

1. Identify the term that grows the slowest but still provides a valid lower bound.
2. Express the result as Ω(this term).

3. Theta Notation (Θ)

Definition:

Θ(g(n)) = { f(n) : there exist positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

Intuition:

Theta provides both an upper and lower bound on the growth rate of a function. It says that f(n) grows at the same rate as g(n).

Example:

If f(n) = 3n$^{\text{2 + 2n + 1, then f(n) = Θ(n}}$2) because it's both O(n$^{\text{2) and Ω(n}}$2).

How to use in exams:

1. Check if the function is both O(g(n)) and Ω(g(n)) for some g(n).
2. If so, it's Θ(g(n)).

4. Little o Notation (o)

Definition:

o(g(n)) = { f(n) : for any positive constant c, there exists an n0 such that 0 ≤ f(n) < cg(n) for all n ≥ n0 }

Intuition:

Little o provides a strict upper bound. It says that f(n) grows strictly slower than g(n).

Example:

n = o($n^{2)\text{ because }n/n}$2 approaches 0 as n grows large.

How to use in exams:

1. Use when you need to show that one function grows strictly slower than another.
2. Often used in more advanced proofs.

5. Little omega Notation (ω)

Definition:

ω(g(n)) = { f(n) : for any positive constant c, there exists an n0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n0$ }

Intuition:

Little omega provides a strict lower bound. It says that f(n) grows strictly faster than g(n).

Example:

$n^{2 = \omega(n)\text{ because }n}$2/n approaches infinity as n grows large.

How to use in exams:

1. Use when you need to show that one function grows strictly faster than another.
2. Often used in more advanced proofs.

Key Properties and Relationships

1. Transitivity: If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)). (Also applies to Ω and Θ)

2. Reflexivity: f(n) = O(f(n)), f(n) = Ω(f(n)), f(n) = Θ(f(n))

3. Symmetry: f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

4. Transpose symmetry: f(n) = O(g(n)) if and only if g(n) = Ω(f(n))

5. If f(n) = Θ(g(n)), then f(n) = O(g(n)) and f(n) = Ω(g(n))

6. If f(n) = o(g(n)), then f(n) = O(g(n))

7. If f(n) = ω(g(n)), then f(n) = Ω(g(n))

How to Apply in Exams

1. Analyzing Algorithms:
    - Express the running time of an algorithm in terms of the input size n.
    - Use Big O to give an upper bound on the worst-case running time.
    - Use Omega to give a lower bound on the best-case running time.
    - Use Theta when you can determine both upper and lower bounds.
2. Comparing Algorithms:
    - Use asymptotic notation to compare the efficiency of different algorithms.
    - An O(n log n) algorithm is generally more efficient than an O(n^2) algorithm for large inputs.
3. Solving Recurrences:
    - When solving recurrence relations (e.g., using the Master Theorem), express the result in asymptotic notation.
4. Proving Time Complexities:
    - Use the definitions to prove that a function belongs to a certain asymptotic class.
    - For example, to prove f(n) = O(g(n)), find constants c and n0 that satisfy the definition.
5. Simplifying Expressions:
    - When given a complex function, express it in the simplest asymptotic form.
    - For example, 3n$^{2 + 4n \log n + 2n = O(n}$$_{2}$)

Remember, asymptotic notation is about the behavior of functions as n approaches infinity. It's not concerned with small inputs or constant factors. Focus on the dominant terms and the overall growth rate of the function.

By understanding and applying these concepts, you'll be well-equipped to analyze and compare algorithm efficiencies in your exam.

Comprehensive Algorithms Study Guide: Chapters 1-5

1. Insertion Sort

Pseudocode:

```
INSERTION-SORT(A)
    for i = 2 to A.length
        key = A[i]
        // Insert A[i] into the sorted subarray A[1..i-1]
        j = i - 1
        while j > 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Tracing:

To trace Insertion Sort, follow these steps:

1. Start with i = 2 (second element)
2. Compare A[i] with previous elements
3. Shift elements greater than A[i] to the right
4. Insert A[i] in its correct position
5. Repeat for all elements

Time Complexity Analysis:

- Worst-case and average-case: O(n^2)
    - Occurs when the array is in reverse order
    - Nested loops: outer loop runs n-1 times, inner loop runs up to i-1 times
- Best-case: O(n)
    - Occurs when the array is already sorted
    - Inner while loop never executes

When to Use:

- Small input sizes
- Nearly sorted arrays
- Online algorithm (can sort as it receives input)
- When simplicity is preferred over efficiency for larger inputs

2. Merge Sort

Pseudocode:

```
MERGE-SORT(A, p, r)
    if p < r
        q = ⌊(p + r) / 2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)

MERGE(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1+1] and R[1..n2+1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
```

Tracing:

To trace Merge Sort:

1. Recursively divide the array until subarrays of size 1
2. Merge subarrays:
    - Compare elements from both subarrays
    - Place smaller element in the result array
    - Move to next element in the subarray that contributed the smaller element
    - Repeat until all elements are merged

Time Complexity Analysis:

- All cases: O(n log n)
    - Dividing: log n levels
    - Merging: O(n) work at each level

    - Total: O(n log n)

When to Use:

- Large datasets
- When stable sort is required
- When guaranteed O(n log n) performance is needed
- External sorting (when data doesn't fit in memory)

3. Heapsort

Pseudocode:

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else
        largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)

LEFT(i) = 2i
RIGHT(i) = 2i + 1
```

Tracing:

To trace Heapsort:

1. Build max heap from the input array
2. Repeatedly extract the maximum element and place it at the end of the array
3. Maintain the heap property on the reduced heap

Time Complexity Analysis:

- All cases: O(n log n)
    - BUILD-MAX-HEAP: O(n)
    - n-1 calls to MAX-HEAPIFY, each O(log n)
    - Total: O(n log n)

When to Use:

- When in-place sorting is required
- When guaranteed O(n log n) performance is needed
- When memory usage is a concern

4. Quicksort

Pseudocode:

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

Tracing:

To trace Quicksort:

1. Choose a pivot element
2. Partition the array around the pivot
3. Recursively sort the subarrays on either side of the pivot

Time Complexity Analysis:

- Worst-case: O(n^2)
    - Occurs when the pivot is always the smallest or largest element
- Average-case: O(n log n)
    - Balanced partitions on average
- Best-case: O(n log n)
    - Occurs when the pivot always divides the array in half

When to Use:

- General-purpose sorting
- When average-case performance is more important than worst-case
- When in-place sorting is desired

5. Counting Sort

Pseudocode:

```
COUNTING-SORT(A, B, k)
    let C[0..k] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to i
    for i = 1 to k
        C[i] = C[i] + C[i - 1]
    // C[i] now contains the number of elements ≤ i
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Tracing:

To trace Counting Sort:

1. Count occurrences of each element

2. Calculate cumulative counts
3. Place elements in their correct positions in the output array

Time Complexity Analysis:

- All cases: O(n + k), where n is the number of elements and k is the range of input
    - Three separate loops, each O(n) or O(k)

When to Use:

- When the range of input values (k) is not significantly larger than the number of elements (n)
- Sorting integers or elements that can be converted to integers
- When linear time sorting is required

6. Radix Sort

Pseudocode:

```
RADIX-SORT(A, d)
    for i = 1 to d
        use a stable sort to sort array A on digit i
```

Tracing:

To trace Radix Sort:

1. Start from the least significant digit
2. Sort the elements based on this digit
3. Move to the next significant digit and repeat
4. Continue until the most significant digit

Time Complexity Analysis:

- Time: O(d(n + k)), where d is the number of digits, n is the number of elements, and k is the range of each digit
    - d passes, each taking O(n + k) time if using counting sort as the stable sort

When to Use:

- Sorting integers with a fixed number of digits
- When the range of each digit is small
- When a stable sort is required

7. Bucket Sort

Pseudocode:

```
BUCKET-SORT(A)
    n = A.length
    let B[0..n-1] be a new array
    for i = 0 to n - 1
        make B[i] an empty list
    for i = 1 to n
        insert A[i] into list B[⌊nA[i]⌋]
    for i = 0 to n - 1
        sort list B[i] with insertion sort
    concatenate the lists B[0], B[1], ..., B[n-1] in order
```

Tracing:

To trace Bucket Sort:

1. Create n empty buckets
2. Distribute elements into buckets based on their value
3. Sort each non-empty bucket
4. Concatenate all sorted buckets

Time Complexity Analysis:

- Average-case: O(n) when input is uniformly distributed
  - Distribution and concatenation: O(n)
  - Sorting each bucket: O(1) on average
- Worst-case: O(n^2) when all elements fall into a single bucket

When to Use:

- When input is uniformly distributed
- When input is floating-point numbers in range [0, 1)
- When average-case linear time is acceptable

Relationships Between Algorithms

1. Comparison-Based Sorts:
   - Insertion Sort, Merge Sort, Heapsort, and Quicksort are all comparison-based.

- They have a lower bound of Ω(n log n) for worst-case time complexity.
- Insertion Sort is simplest but least efficient for large inputs.
- Merge Sort and Heapsort guarantee O(n log n) time.
- Quicksort has best average-case performance but worst-case O(n^2).

2. Linear-Time Sorts:
   - Counting Sort, Radix Sort, and Bucket Sort can achieve linear time under certain conditions.
   - They are not comparison-based and can beat the Ω(n log n) lower bound.
   - Counting Sort is the basis for Radix Sort.
   - Bucket Sort relies on the distribution of input data.
3. Divide-and-Conquer:
   - Merge Sort, Quicksort, and Strassen's algorithm use divide-and-conquer strategy.
   - They divide the problem into smaller subproblems, solve recursively, and combine results.
4. In-Place Sorting:
   - Insertion Sort, Heapsort, and Quicksort can sort in-place (O(1) extra space).
   - Merge Sort typically requires O(n) extra space.
5. Stability:
   - Insertion Sort, Merge Sort, and Counting Sort are stable (preserve relative order of equal elements).
   - Heapsort and Quicksort are not inherently stable.
6. Adaptability:
   - Insertion Sort and Quicksort adapt well to partially sorted inputs.
   - Merge Sort and Heapsort perform the same regardless of input order.

Understanding these relationships helps in choosing the right algorithm for a given situation, considering factors like input size, distribution, memory constraints, and stability requirements.