**Problem 1.1 Solutions**

**Group 0**

1. f2 = n (linear growth)
2. f3 = n log n (linearithmic growth, faster than linear)
3. f1 = n^2 (quadratic growth, faster than linearithmic)
4. f5 = (log n^2)^2 = (2 log n)^2 = O((log n)^2) (very slow growth, but still faster than quadratic)
5. f4 = 2^n (exponential growth, fastest in this group)

**Explanation:** The growth rates are arranged from slowest to fastest. Linear growth (f2) is the slowest, followed by linearithmic (f3), then quadratic (f1). The logarithmic function (f5) grows very slowly, but still faster than quadratic. Finally, exponential growth (f4) is the fastest.

**Group 1**

1. f5 = log 3n^n^3 (very slow growth)
2. f1 = log ((log n)^3) (slow growth)
3. f6 = (log log n)^3 (slow growth, but faster than f1)
4. f2 = (log n)^(3(log 3n)) (faster than logarithmic, but slower than exponential)
5. f3 = 3^(log n) (exponential growth, faster than f2)
6. f4 = n^(3^(log n)) (very fast growth, fastest in this group)

**Explanation:** The growth rates are arranged from slowest to fastest. Logarithmic functions (f5, f1, f6) grow slowly, followed by a mix of logarithmic and exponential growth (f2). Exponential growth (f3) is faster, and the fastest growth is exhibited by f4.

**Group 3**

1. f1 = 4^(3n) (exponential growth)
2. f5 = 2^(5n) (faster exponential growth)
3. f3 = 2^(3n+1) (faster exponential growth)
4. f2 = 2^(n^4) (very fast growth, but slower than f4)
5. f4 = 2^(3^n) (extremely fast growth, fastest in this group)

**Explanation:** The growth rates are arranged from slowest to fastest. Exponential growth (f1) is the slowest, followed by faster exponential growth rates (f5, f3). The growth rate of f2 is very fast, but slower than f4, which exhibits extremely fast growth.

**Problem 1.2 Solutions**

**(a) True**

If t(n) ∈ O(g(n)), then there exist positive constants c and n0 such that:

t(n) ≤ c \* g(n) for all n ≥ n0

By definition of Ω notation, this implies:

g(n) ≥ (1/c) \* t(n) for all n ≥ n0

So, g(n) ∈ Ω(t(n)).

**(b) True**

Θ(αg(n)) = {f(n) | there exist positive constants c1, c2, and n0 such that c1 \* αg(n) ≤ f(n) ≤ c2 \* αg(n) for all n ≥ n0}

= {f(n) | there exist positive constants c1/α, c2/α, and n0 such that c1/α \* g(n) ≤ f(n) ≤ c2/α \* g(n) for all n ≥ n0}

= Θ(g(n))

**(c) True**

Θ(g(n)) = {f(n) | there exist positive constants c1, c2, and n0 such that c1 \* g(n) ≤ f(n) ≤ c2 \* g(n) for all n ≥ n0}

= O(g(n)) ∩ Ω(g(n))

**(d) False**

Counterexample:

t(n) = n^2 sin(n), g(n) = n^2

t(n) is not O(g(n)) because the coefficient of n^2 in t(n) oscillates between -1 and 1, so it cannot be bounded by a constant multiple of g(n).

t(n) is not Ω(g(n)) for the same reason.

However, neither t(n) ∈ O(g(n)) nor t(n) ∈ Ω(g(n)) holds, but not both.

**Problem 2.1 Solutions**

**(a)**

T(n) = aT(n/2) + Θ(n^2)

Substituting T(n) = Θ(n^2 log n):

a(n/2)^2 log(n/2) + Θ(n^2) = n^2 log n

Simplifying:

a(n^2/4) log n - a(n^2/4) log 2 + Θ(n^2) = n^2 log n

Comparing coefficients:

a/4 = 1 => a = 4

**(b)**

T(n) = aT(n/3) + Θ(n)

Substituting T(n) = Θ(n^2):

a(n/3)^2 + Θ(n) = n^2

Simplifying:

a(n^2/9) + Θ(n) = n^2

Comparing coefficients:

a/9 = 1 => a = 9

**(c)**

T(n) = 4T(n/b) + Θ(n^2)

Substituting T(n) = Θ(n^2):

4(n/b)^2 + Θ(n^2) = n^2

Simplifying:

4(n^2/b^2) + Θ(n^2) = n^2

Comparing coefficients:

4/b^2 = 1 => b^2 = 4 => b = 2

**(d)**

T(n) = 5T(n/b) + Θ(n^5)

Substituting T(n) = Θ(n^6.006):

5(n/b)^6.006 + Θ(n^5) = n^6.006

Simplifying:

5(n^6.006/b^6.006) + Θ(n^5) = n^6.006

Comparing coefficients:

5/b^6.006 = 1 => b^6.006 = 5 => b ≈ 1.098

**(e)**

T(n) = 6T(n/6) + f(n)

Substituting T(n) = Θ(n^2):

6(n/6)^2 + f(n) = n^2

Simplifying:

n^2/6 + f(n) = n^2

f(n) = n^2 - n^2/6 = 5n^2/6

So, one possible function f is f(n) = 5n^2/6.

**Problem 2.2 Solutions**

**(a)**

T(n) = T(n/2) + T(n/4) + n^2

Recursion Tree:

* Level 1: T(n) = n^2
* Level 2: T(n/2) + T(n/4) = (n/2)^2 + (n/4)^2
* Level 3: T(n/4) + T(n/8) + T(n/8) + T(n/16) = (n/4)^2 + 2(n/8)^2 + (n/16)^2

Notice the pattern:

* Each level has a total of n^2 work
* The tree has log4(n) levels (since the input size decreases by a factor of 4 at each level)

Total work = n^2 \* log4(n) = Θ(n^2 log n)

**(b)**

T(n) = 4T(n/2) + n/lg n

Recursion Tree:

* Level 1: T(n) = n/lg n
* Level 2: 4T(n/2) = 4(n/2)/lg(n/2) = 2n/lg(n/2)
* Level 3: 16T(n/4) = 16(n/4)/lg(n/4) = 4n/lg(n/4)

Notice the pattern:

* Each level has a total of n/lg(n/2^k) work
* The tree has log2(n) levels (since the input size decreases by a factor of 2 at each level)

Total work = ∑[k=0 to log2(n)] n/lg(n/2^k) = Θ(n lg lg n)

Note: The recursion tree method helps visualize the recursive calls and their contributions to the total work, making it easier to derive the complexity class.

Here are the steps to sort the array using quicksort:

**Iteration 1**

* Choose a pivot: E (first element)
* Partition the array:
  + Left: [A, E, E, L] (elements less than E)
  + Pivot: E
  + Right: [X, M, P, S] (elements greater than E)
* Recursively sort left and right subarrays

**Iteration 2 (Left subarray)**

* Choose a pivot: A
* Partition the array:
  + Left: [E, E] (elements less than A)
  + Pivot: A
  + Right: [L] (elements greater than A)
* Recursively sort left and right subarrays

**Iteration 3 (Left subarray of Left subarray)**

* Choose a pivot: E
* Partition the array:
  + Left: [] (elements less than E)
  + Pivot: E
  + Right: [E] (elements greater than E)
* No more recursion needed (array is sorted)

**Iteration 4 (Right subarray of Left subarray)**

* Choose a pivot: L
* Partition the array:
  + Left: [] (elements less than L)
  + Pivot: L
  + Right: [] (elements greater than L)
* No more recursion needed (array is sorted)

**Iteration 5 (Right subarray)**

* Choose a pivot: X
* Partition the array:
  + Left: [M, P, S] (elements less than X)
  + Pivot: X
  + Right: [] (elements greater than X)
* Recursively sort left subarray

**Iteration 6 (Left subarray of Right subarray)**

* Choose a pivot: M
* Partition the array:
  + Left: [P, S] (elements less than M)
  + Pivot: M
  + Right: [] (elements greater than M)
* Recursively sort left subarray

**Iteration 7 (Left subarray of Left subarray of Right subarray)**

* Choose a pivot: P
* Partition the array:
  + Left: [S] (elements less than P)
  + Pivot: P
  + Right: [] (elements greater than P)
* No more recursion needed (array is sorted)

**Final sorted array**

[A, E, E, L, M, P, S, X]

Note: The pivot selection and partitioning steps may vary depending on the specific quicksort implementation. This is just one possible way to sort the array using quicksort.

**Problem 3.1 Solutions**

**(a)**

Pattern: 0 0 0 0 1

* The brute force algorithm will compare the pattern with every substring of the binary tree.
* Since the tree has 1000 zeros, there are 996 substrings of length 5 (the length of the pattern).
* For each substring, the algorithm will make 5 comparisons (one for each character in the pattern).
* Therefore, the total number of comparisons is: 996 substrings × 5 comparisons/substring = 4980 comparisons

**(b)**

Pattern: 0 1 0 1 0

* The brute force algorithm will compare the pattern with every substring of the binary tree.
* Since the tree has 1000 zeros, there are 996 substrings of length 5 (the length of the pattern).
* For each substring, the algorithm will make 5 comparisons (one for each character in the pattern).
* However, since the pattern contains a '1', the algorithm will stop comparing as soon as it finds a mismatch (i.e., a '0' in the tree).
* Therefore, the average number of comparisons per substring is less than 5.
* Assuming an average of 3 comparisons per substring (a rough estimate), the total number of comparisons is: 996 substrings × 3 comparisons/substring ≈ 2988 comparisons

Note: The actual number of comparisons may vary depending on the specific implementation of the brute force algorithm.

**Problem 3.2 Solutions**

**Brute Force Analysis:**

Brute force analysis involves analyzing an algorithm by exhaustively enumerating all possible inputs and computing the exact number of operations (e.g., comparisons, additions) performed for each input. This approach provides an exact count of the algorithm's time complexity.

**Strengths:**

1. **Exact count:** Brute force analysis provides an exact count of the algorithm's time complexity, which is useful for small inputs.
2. **Simple to implement:** Brute force analysis is straightforward to implement, as it involves simply running the algorithm on all possible inputs.

**Weaknesses:**

1. **Inefficient for large inputs:** Brute force analysis becomes impractical for large inputs, as the number of possible inputs grows exponentially.
2. **Limited scalability:** Brute force analysis is not scalable, as it requires re-computing the exact count for each input size.

**(a)**

**Minimum number of character comparisons:**

To find the minimum number of character comparisons, we can use the brute force algorithm to search for the pattern BRANDING in the text.

1. Compare the first character of the pattern (B) with the first character of the text (T). Mismatch, move to the next character in the text.
2. Compare the first character of the pattern (B) with the second character of the text (H). Mismatch, move to the next character in the text.  
   ...
3. Compare the last character of the pattern (G) with the last character of the text (D). Mismatch.

The minimum number of character comparisons is equal to the length of the text (47) plus the length of the pattern minus one (8-1=7), since we can stop comparing as soon as we find a mismatch.

Minimum number of character comparisons = 47 + 7 = 54

Note: This is a simplified analysis, as it assumes the algorithm stops comparing as soon as it finds a mismatch. In practice, the algorithm may continue comparing until it finds a match or reaches the end of the text.

**Problem 3.3 Solutions**

**(i) Construct a heap:**

1. Start with an empty heap: []
2. Insert 1: [1]
3. Insert 8: [8, 1] (swap 8 and 1 since 8 is larger)
4. Insert 6: [8, 1, 6]
5. Insert 5: [8, 5, 6, 1] (swap 5 and 1 since 5 is larger)
6. Insert 3: [8, 5, 6, 3, 1]
7. Insert 7: [8, 7, 6, 3, 1, 5] (swap 7 and 5 since 7 is larger)
8. Insert 4: [8, 7, 6, 4, 1, 5, 3] (swap 4 and 3 since 4 is larger)
9. Insert 2: [8, 7, 6, 4, 2, 5, 3, 1]

The resulting heap is: [8, 7, 6, 4, 2, 5, 3, 1]

**(ii) Sort the list using Heapsort:**

1. Build the heap (already done): [8, 7, 6, 4, 2, 5, 3, 1]
2. Swap the root (8) with the last element (1): [1, 7, 6, 4, 2, 5, 3, 8]
3. Heapify the reduced heap (excluding the last element): [7, 5, 6, 4, 2, 3, 1]
4. Swap the root (7) with the last element (1): [1, 5, 6, 4, 2, 3, 7]
5. Heapify the reduced heap (excluding the last element): [5, 4, 6, 3, 2]
6. Swap the root (5) with the last element (2): [2, 4, 6, 3]
7. Heapify the reduced heap (excluding the last element): [4, 3, 6]
8. Swap the root (4) with the last element (3): [3, 4]
9. Heapify the reduced heap (excluding the last element): [4]

The sorted list is: [1, 2, 3, 4, 5, 6, 7, 8]

**Problem 3.4 Solutions**

**(a) Basic operations:**

* Comparisons (A[j] ≤ A[j+1])
* Swaps (SWAP(A[j], A[j+1]))

**(b) Complexity class:**

The algorithm has a nested loop structure:

* Outer loop runs n-1 times
* Inner loop runs up to j+1 times (worst-case scenario)

Total number of comparisons and swaps: ∑(j+1) from i=1 to n-1 = n\*(n-1)/2

Complexity class: O(n^2)

**(c) In-place algorithm:**

Yes, the algorithm is in-place because it only uses a single array and swaps elements within the array without using any additional storage.

**(d) Stability:**

No, the algorithm is not stable because it uses a swap operation that can change the relative order of equal elements.

Modification to make it stable:

Replace the swap operation with a "move-to-the-right" operation, where the larger element is moved to the right of the smaller element without swapping them. This preserves the relative order of equal elements.

Example:

Instead of swapping A[j] and A[j+1], move A[j+1] to the right of A[j] by shifting elements to the right.

This modification maintains the correctness of the algorithm while making it stable.

**Problem 4.1 Solutions**

**(a) Exhaustive Search:**

1. Generate all possible subsets of the items:
   * {} (empty set)
   * {1}
   * {2}
   * {3}
   * {4}
   * {1,2}
   * {1,3}
   * {1,4}
   * {2,3}
   * {2,4}
   * {3,4}
   * {1,2,3}
   * {1,2,4}
   * {1,3,4}
   * {2,3,4}
   * {1,2,3,4}
2. Calculate the total weight and value of each subset:
   * {} : weight=0, value=$0
   * {1} : weight=2, value=$15
   * {2} : weight=3, value=$20
   * {3} : weight=1, value=$10
   * {4} : weight=2, value=$12
   * {1,2} : weight=5, value=$35
   * {1,3} : weight=3, value=$25
   * {1,4} : weight=4, value=$27
   * {2,3} : weight=4, value=$30
   * {2,4} : weight=5, value=$32
   * {3,4} : weight=3, value=$22
   * {1,2,3} : weight=6, value=$45 (exceeds knapsack capacity)
   * {1,2,4} : weight=7, value=$47 (exceeds knapsack capacity)
   * {1,3,4} : weight=5, value=$37
   * {2,3,4} : weight=6, value=$42 (exceeds knapsack capacity)
   * {1,2,3,4} : weight=8, value=$57 (exceeds knapsack capacity)
3. Select the subset with the maximum value that fits within the knapsack capacity (5 units):
   * {1,2} : weight=5, value=$35

**(b) Dynamic Programming:**

1. Create a 2D table dp[n+1][W+1] where n is the number of items and W is the knapsack capacity:
   * dp[i][w] represents the maximum value that can be obtained using the first i items and a knapsack capacity of w
2. Initialize the table:
   * dp[0][w] = 0 for all w (no items)
   * dp[i][0] = 0 for all i (no capacity)
3. Fill the table using the recurrence relation:
   * dp[i][w] = max(dp[i-1][w], dp[i-1][w-wi] + vi) if wi ≤ w
   * dp[i][w] = dp[i-1][w] otherwise
4. The maximum value that fits within the knapsack capacity is stored in dp[n][W]:
   * dp[5][5] = $37

The most valuable subset of items that fit into the knapsack using dynamic programming is {1,3,4} with a total value of $37.

**Problem 4.2 Solutions**

**(a) Recurrence relation:**

Let X = x1x2...xm and Y = y1y2...yn be two sequences. Let LCS(X, Y) be the length of their longest common subsequence.

* If xm = yn, then LCS(X, Y) = LCS(x1x2...xm-1, y1y2...yn-1) + 1
* If xm ≠ yn, then LCS(X, Y) = max(LCS(x1x2...xm-1, Y), LCS(X, y1y2...yn-1))

**(b) Algorithm to compute LCS length:**

1. Create a 2D table dp[m+1][n+1] where m and n are the lengths of X and Y respectively.
2. Initialize the table:
   * dp[0][j] = 0 for all j
   * dp[i][0] = 0 for all i
3. Fill the table using the recurrence relation:
   * If xm = yn, then dp[i][j] = dp[i-1][j-1] + 1
   * If xm ≠ yn, then dp[i][j] = max(dp[i-1][j], dp[i][j-1])
4. The length of the LCS is stored in dp[m][n].

**(c) Dynamic programming example:**

X = KADUNA, Y = KAKNO

|  | **0** | **A** | **K** | **N** | **O** |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| K | 0 | 0 | 1 | 1 | 1 |
| A | 0 | 1 | 1 | 1 | 1 |
| K | 0 | 1 | 2 | 2 | 2 |
| N | 0 | 1 | 2 | 3 | 3 |
| O | 0 | 1 | 2 | 3 | 3 |

The length of the LCS is 3, which corresponds to the subsequence "KAN".

**Problem 4.3 Solutions**

**(a) Algorithm:**

1. Sort the books using a stable sorting algorithm (e.g., Merge Sort) based on their catalogue numbers.
2. Initialize an empty list to store the unique books.
3. Iterate through the sorted list of books:
   * If the current book's catalogue number is different from the previous book's, add it to the unique list.
   * If the current book's catalogue number is the same as the previous book's, skip it (return it to the catalogue section).
4. Return the unique list of books.

**(b) In-place algorithm:**

No, the proposed algorithm is not in-place because it uses an additional list to store the unique books.

**(c) Performance evaluation:**

* Time complexity: O(n log n) due to the stable sorting algorithm.
* Space complexity: O(n) for the additional list to store unique books.

**(d) Alternative solution:**

1. Sort the books using an efficient sorting technique (e.g., Merge Sort) with complexity Θ(n log n).
2. Iterate through the sorted list of books, checking only consecutive books:
   * If two or more books have the same catalogue number, they must be next to each other.
   * Remove duplicates by skipping books with the same catalogue number as the previous book.

Complexity analysis:

* Sorting: Θ(n log n)
* Iterating through the sorted list: O(n)
* Total complexity: Θ(n log n) + O(n) = Θ(n log n)

This alternative solution has the same time complexity as the original algorithm but uses less extra space (only a small amount for the sorting algorithm's recursion stack).

**Problem 4.4 Solutions**

**(a) Adjacency Matrix:**

|  | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| c | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| d | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| e | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| f | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| h | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**(b) Depth-First Traversal (from vertex a):**

1. a
2. b
3. e
4. f
5. g
6. h
7. c
8. d

**(c) Breadth-First Traversal (from vertex a):**

1. a
2. b
3. d
4. e
5. c
6. h
7. f
8. g

**(d) Topological Ordering (using Source Removal Algorithm):**

1. a
2. d
3. b
4. h
5. c
6. e
7. f
8. g

Note: The topological ordering is not unique, but this is one possible ordering.