Topic 1: Combination & Pruning

Q1: Explain how generating all combinations of n items can be modeled as generating an n-bit binary number. Discuss how the recursive function `comb(i)` constructs this state space.

Answer: Generating combinations of n items can be modeled as an n-bit binary number because each item represents a binary choice: it is either "selected" (1) or "not selected" (0). Consequently, there are 2^n possible combinations. +4

The recursive function `comb(i)` constructs this state space by iterating through the index i, which represents the current item being processed (from 0 to $n-1$). +2

- **Assignment:** At each step, the function assigns a value (0 or 1) to the i-th position in a global list. +1
- **Recursion:** It then recursively calls `comb` for the next index ($i+1$) to determine the rest of the combination.
- **Termination:** Once the index reaches n (implicit in the logic), a complete combination has been assigned and is printed. This process explores the full binary tree of depth n, effectively visiting all 2^n leaf nodes. +1

Q2: In the context of generating combinations with exactly k ones, analyze the concept of "pruning." Explain the three conditions under which recursion terminates early.

Answer: "Pruning" is the technique of terminating a recursion early when it is determined that no further recursive calls will result in a feasible answer. For the problem of finding combinations with exactly k ones (where s is the current sum of ones so far), the recursion can be pruned in three specific cases: +1

1. **Target Met ($s = k$):** If the current sum of ones equals k, the recursion terminates immediately. The only valid assignment for the remaining items is 0, so no further branching is needed.
2. **Forced Selection ($n - i + s = k$):** If the number of remaining items ($n - i$) plus the current sum (s) exactly equals k, the recursion terminates. To reach k, all remaining items *must* be 1s; no other variation is possible.
3. **Impossible Target ($n - i + s < k$):** If the remaining items plus the current sum is less than k, it is impossible to reach the target even if all subsequent items are selected. The recursion terminates as the branch is invalid.

Q3: Compare the brute-force method of generating combinations with the optimized approach for selecting k items. Why is keeping k as a global variable preferred over passing it as a recursive argument?

Answer: The brute-force method generates all possible states (all 2^n combinations) and then tests each complete combination to see if it meets the criteria (e.g., having exactly k ones). This is inefficient because it explores paths that are guaranteed to fail. The optimized approach uses pruning to detect these failure conditions early, stopping the recursion before it generates invalid complete states. +3

Keeping k as a global variable is preferred because its value remains constant throughout the program execution. Passing it as an argument to every recursive call consumes additional stack memory and processing overhead without providing any benefit, as the recursive copies do not need to modify or uniquely track k.

Topic 2: Balance Split

Q4: The "Balance Split" problem requires splitting goods into two groups with minimal value difference. Describe how this problem can be structurally mapped to the "Combination" problem.

Answer: The Balance Split problem involves dividing n goods with values $v_1, \ldots, v_n$ into two groups. This maps directly to the Combination problem because the assignment of each good can be represented as a binary choice: a good belongs either to Group 0 or Group 1. +1

Ideally, an n-bit binary number represents a single unique split. If the i-th bit is 0, the good v_i is assigned to the first group; if it is 1, it is assigned to the second group. The goal is to generate these binary states and find the one that minimizes the difference between the total values of the two groups. +2

Q5: Analyze the time complexity of the Balance Split solution. Explain why passing the sum of values in group 1 as a recursive argument reduces the running time from $O(n \cdot 2^n)$ to $O(2^n)$.

Answer: The brute-force recursive search explores 2^n possible splits. +1

- **Unoptimized ($O(n \cdot 2^n)$):** In the standard approach, the algorithm generates a complete split (reaching a leaf node) and then iterates through the list of n items to calculate the sums of the groups. This summation takes $O(n)$ time per leaf, leading to a total complexity of $O(n \cdot 2^n)$.
- **Optimized ($O(2^n)$):** By passing the current sum of Group 1 as an additional argument in the recursive call, the sum is maintained incrementally. When the recursion reaches a leaf node, the group total is already computed. This eliminates the need for the final $O(n)$ summation loop, reducing the running time to $O(2^n)$. +1

Q6: Discuss the limitations of the brute-force recursive search for the Balance Split problem. Given that the input size is limited to $n \leq 10$, how would performance degrade as n increases?

Answer: The primary limitation is the exponential growth of the search space (2^n). While $n \leq 10$ results in a manageable 1024 operations (2^{10}), the number of operations doubles with every additional item. As n increases, the performance degrades rapidly; for example, $n = 20$ requires over a million operations. This indicates that brute-force recursion is not scalable for larger datasets without optimization techniques like dynamic programming or pruning. +2

Topic 3: Edit Distance

Q7: Define the three core edit operations used to calculate the Levenshtein distance. Explain the recursive state transitions required when characters $A[i]$ and $B[j]$ do not match.

Answer: The three edit operations are Insertion, Deletion, and Substitution (Change). When transforming string A to string B, if $A[i]$ does not match $B[j]$, the recursive solution considers three scenarios, each adding 1 to the edit distance: +1

1. **Insertion:** Insert $B[j]$ in front of $A[i]$. The next state is `(i, j+1)` because $B[j]$ is now matched/inserted, but we still need to handle $A[i]$. +1
2. **Deletion:** Delete $A[i]$. The next state is `(i+1, j)` because $A[i]$ is removed, and we move to the next character in A. +1
3. **Substitution:** Change $A[i]$ to $B[j]$. The next state is `(i+1, j+1)` because both current characters are now processed and matched. +1

Q8: In the Edit Distance problem, what are the base cases for the recursion? Specifically, describe the calculation required when one string is exhausted.

Answer: The base cases handle the scenario where one string has been fully processed (exhausted).

- **A runs out ($i == len(A)$):** If A is exhausted but B is not, the remaining characters in B must be inserted. The additional cost is the number of remaining characters in B, calculated as $len(B) - j$.
- **B runs out ($j == len(B)$):** If B is exhausted but A is not, the remaining characters in A must be deleted. The additional cost is the number of remaining characters in A, calculated as $len(A) - i$.

Q9: The worksheet suggests improving the brute-force Edit Distance solution to handle strings up to 1000 letters. Discuss why a simple recursive solution is insufficient and how Dynamic Programming helps.

Answer: A simple recursive solution is insufficient because it repeatedly solves the same subproblems (overlapping recursive calls), leading to exponential time complexity. For strings of length 1000, this would take practically infinite time.

Dynamic Programming (specifically memoization) addresses this by storing the result of each subproblem (state `mm[v]`) after it is computed. When the function is called, it first checks if the value for the current state exists; if so, it returns the stored value immediately, avoiding redundant calculations. This reduces the complexity to polynomial time, allowing the program to finish within the 2.5-second constraint. +3

Topic 4: Comparative & Synthesis

Q10: Compare the optimization techniques used in "Counting Combinations" (pruning) and "Minimum Coin Change" (memoization).

Answer:

- **Pruning (Combinations):** This technique focuses on avoiding **invalid paths**. The recursion stops early (e.g., when $s = k$) because logic dictates that no valid solution can exist further down that branch. +1
- **Memoization (Coin Change):** This technique focuses on avoiding **repeated calculations**. It does not necessarily stop a "wrong" path but prevents re-calculating the optimal result for a state (e.g., a specific remaining value v) that has already been solved by a previous branch. +1

Q11: Contrast the branching factor of the Balance Split/Combination problems with the Edit Distance problem.

Answer:

- **Balance Split / Combination:** These problems rely on a **binary choice** model (select/not select, Group 0/Group 1), resulting in a branching factor of 2. The complexity grows as $O(2^n)$. +1
- **Edit Distance:** This problem relies on a **ternary choice** model (Insert, Delete, Substitution) whenever characters do not match. This results in a branching factor of 3, causing the brute-force search space to grow as $O(3^n)$ (or similar based on string length), which is significantly faster and more computationally expensive than the binary model. +1

Topic 1: Minimum Coin Change

Question: Describe the core recursive step for determining the minimum number of coins for a given change value v. specifically, how do you determine the optimal first coin to pick?
Answer: To determine `mincoin(v)`, the algorithm considers every possible coin denomination (c) available in the input list.

- For every valid coin choice (where $c \leq v$), the remaining value becomes $v - c$.
- The problem then recursively solves for the minimum coins needed for this remainder, which is `mincoin(v-c)`.
- The value of `mincoin(v)` is calculated as $1 + \min(\text{mincoin}(v - c))$ across all valid coin options. This effectively tests every coin as the "first" coin and selects the path resulting in the lowest total count.

Question: Explain why a brute-force recursive solution for the Minimum Coin Change problem is inefficient for large inputs, such as the second example case ($V = 3377$). **Answer:**

- The brute-force recursive solution recalculates the optimal change for the same remaining values repeatedly. +1
- The worksheet notes that for the second test case (Change = 3377), a brute-force program "cannot handle it," implying the computation time is excessive due to the sheer volume of redundant calculations.
- By adding a counter to the recursive function, one can observe that the number of recursive calls grows rapidly (likely exponentially) as the change value increases. +2

Question: What are the base cases (terminating conditions) for the `mincoin(v)` recursive function? **Answer:** The worksheet asks to identify values of change for which the minimum number of coins is "obvious".

- The primary base case is when the change value v is 0, which requires 0 coins.
- Another implicit base case is when v exactly matches a coin denominator, requiring exactly 1 coin (though the general recursive step $1 + \text{mincoin}(0)$ covers this).

Topic 2: Rod Cutting

Question: In the Rod Cutting problem, how is the decision to cut a rod of length L modeled recursively? **Answer:**

- The problem assumes we know the price p_i for a rod of length i.
- To find the maximum revenue for length L, the algorithm considers all possible "first cuts" at length i (where $1 \leq i \leq L$).
- For each cut at length i, the immediate revenue is p_i, and the remaining rod has length $L - i$. The algorithm then recursively finds the maximum revenue for the remainder, `maxRev(L-i)`.
- The total revenue for that specific cut is $p_i + \text{maxRev}(L - i)$. The function `maxRev(L)` returns the maximum of these values across all possible splits.

Question: Compare the growth of recursive calls in the Rod Cutting problem as the rod length L increases. **Answer:** Similar to the coin change problem, the worksheet instructs to add a global variable to count function calls. Testing with incremental lengths of rod L reveals that the number of recursive calls grows significantly because the same subproblems (shorter rod lengths) are solved repeatedly for different cut combinations.

Question: How does the Rod Cutting problem differ from a simple greedy approach where one might just pick the highest price per inch? **Answer:** The problem requires determining the *maximum* revenue by considering all combinations of cuts, not just the highest unit price. For example, a rod of length 4 could be sold as one piece or split into smaller pieces like $2 + 2$ or $1 + 3$. A greedy approach might pick the single highest density piece first, potentially missing a better combination. The recursive solution `maxRev(L)` systematically checks all splits ($p_1 + \text{maxRev}(L - 1)$, etc.) to ensure the global maximum is found.

Topic 3: Comparative Analysis

Question: Both the Minimum Coin Change and Rod Cutting problems in Worksheet 3 are optimization problems. What is the common "philosophical" structure shared by their recursive solutions? **Answer:** Both problems rely on the principle of **optimal substructure**. The optimal solution for a larger problem (value v or length L) is constructed from the optimal solutions of smaller subproblems.

- In Coin Change, `mincoin(v)` depends on finding the minimum of $1 + \text{mincoin}(v - c)$.
- In Rod Cutting, `maxRev(L)` depends on finding the maximum of $p_i + \text{maxRev}(L - i)$.
- Both worksheets emphasize observing the explosion of recursive calls, highlighting the inefficiency of pure recursion without memoization.

Topic 1: Maximum Contiguous Subsequence

Q1: Compare the computational efficiency of the "Straightforward Solution" ($O(n^3)$) versus the "Accumulation Technique" ($O(n^2)$). Specifically, explain how the pre-computation of an accumulated list allows the sum of any subsequence to be calculated in constant time. Answer: The "Straightforward Solution" iterates through all possible start indices i and end indices j, and for each pair, it runs a third loop to calculate the sum from $x[i]$ to $x[j]$. This triple nesting results in a complexity of $O(n^3)$.

The "Accumulation Technique" improves this by transforming the list so that index i stores the sum of all numbers from index 0 to i. Once this pre-computation is done (in linear time), the sum of any subsequence from i to j can be calculated directly using the accumulated values without a loop, reducing the sum operation to constant time . This removes the innermost loop, reducing the overall complexity to $O(n^2)$.

Q2: Explain the logic behind Kadane's Algorithm for finding the maximum contiguous sum. How does the algorithm decide at each step whether to extend the current subsequence or restart it from the current position? Answer: Kadane's algorithm iterates through the sequence maintaining a running sum. At each step, it decides whether to extend the current subsequence or reset by comparing the current accumulated sum against zero. Specifically, the logic is $max(current\_sum + x, 0)$. If adding the current number x results in a negative sum, the algorithm resets the current sum to 0 (effectively restarting the subsequence at the next element), because a negative prefix would only decrease the sum of any subsequent sequence. The global maximum is updated whenever the current running sum exceeds the recorded maximum . +1

Q3: Discuss why large input sizes ($n \leq 100,000$) make brute-force solutions ($O(n^3)$ or $O(n^2)$) infeasible for the Maximum Contiguous Subsequence problem. Answer: The problem input specifies $n \leq 100,000$. An $O(n^3)$ or even $O(n^2)$ algorithm would require significantly more operations than a standard processor can handle within a reasonable time limit (typically 1 second). The worksheet notes that only small-sized test cases will finish within 1 second using the straightforward approach. Therefore, a linear time solution like Kadane's Algorithm ($O(n)$) is necessary to handle the maximum input size efficiently. +1

Topic 2: 0-1 Knapsack Problem

Q4: Analyze the transition from a simple recursive combination approach ("Version 1") to a state-based recursive approach ("Version 2") for the Knapsack problem. Why is defining the state as `(index, capacity)` essential for enabling memoization? Answer: In "Version 1," the recursion only tracks the item index i. This is problematic because when deciding on item i, the algorithm does not know the remaining capacity resulting from previous choices. "Version 2" introduces a second state variable, C (capacity), making the state `(i, C)` . This is essential for memoization because the value returned for a specific item index i with a specific remaining capacity C is always unique and consistent. By defining the state this way, we can cache the result of `maxVal(i, C)` and return it immediately if the state repeats, minimizing redundant recursive calls . +3

Q5: Describe the recursive decision-making process in the 0-1 Knapsack problem. How does the algorithm determine the value of `maxVal(i, C)`? Answer: The function `maxVal(i, C)` determines the optimal value by comparing two choices:

1. **Skip:** Do not include item i. The value is the result of the recursive call `maxVal(i+1, C)`.
2. **Take:** Include item i (if $w[i] \leq C$). The value is $v[i] + \text{maxVal}(i+1, C-w[i])$. The function returns the maximum of these two options (`max(skip, take)`), thereby finding the optimal set of items for that specific state.

Q6: Explain the concept of Dynamic Programming in the context of the Knapsack problem as described in Worksheet 6. Answer: Dynamic Programming in this context involves converting the recursive logic into a non-recursive, nested loop structure. Instead of making top-down recursive calls, the algorithm builds a table of answers bottom-up. It iterates through the indices (i) and capacities (C) in a specific direction (e.g., i decreasing, C increasing) to ensure that when `maxVal(i, C)` is computed, the necessary sub-values `maxVal(i+1, C)` and `maxVal(i+1, C-w[i])` are already pre-computed and stored.

Topic 3: Ultimate Greyness

Q7: Explain why the "Ultimate Greyness" problem is structurally similar to generating all subsets and why a brute-force search is acceptable. Answer: The problem asks to mix colors, which implies selecting a subset of the available N colors. This is structurally identical to generating all combinations (subsets) of items. Since the input size is small, with $N \leq 10$, the total number of combinations is $2^{10} = 1024$ (excluding the empty set). This small state space allows a brute-force search to check every possible mixture and find the minimum difference without exceeding time limits.

Q8: Describe how the recursive state must track accumulating values in the "Ultimate Greyness" problem. Answer: Unlike simple counting problems, this problem requires tracking two distinct properties for any given combination:

1. **Vividness:** The product of the vividness of all selected colors.
2. **Dullness:** The sum of the dullness of all selected colors. The algorithm must calculate these two distinct values for every generated subset to compute the final absolute difference.

Topic 4: Minimum Energy Stair Climbing

Q9: Formulate the optimal substructure for the "Minimum Energy Stair Climbing" problem. Answer: The robot can reach step i from either step $i-1$ (climbing 1 step) or step $i-2$ (climbing 2 steps). Therefore, the minimum energy to reach step i is the cost of step i plus the minimum of the energy required to reach step $i-1$ or step $i-2$. This defines the recurrence relation and allows the problem to be solved by building upon optimal solutions to previous steps. +1

Q10: Explain how the option to climb 2 steps allows the algorithm to "skip" expensive steps. Answer: The ability to climb 2 steps provides a strategic advantage to bypass steps with high energy costs. In the provided example, steps with a cost of 100 are avoided by jumping over them from the preceding step (e.g., moving from index 0 to 2, or 2 to 4) . This "skipping" capability is crucial for minimization; the algorithm will always prefer paying the cost of a single step further ahead if it avoids a significantly larger cost at the immediate next step.

Topic 5: Longest Common Subsequence (LCS)

Q11: Based on the principles established in the Knapsack problem, why does the LCS problem typically require a 2-dimensional table to solve efficiently? Answer: Worksheet 6 identifies LCS as a problem solvable via Dynamic Programming, similar to the Knapsack problem. Just as the Knapsack problem requires a state defined by two variables (item index and capacity) to handle constraints, the LCS problem involves comparing two strings. The state is typically defined by the current index in the first string and the current index in the second string. To avoid the exponential time cost of re-calculating the LCS for the same string suffixes recursively, a 2-dimensional table is used to memoize or iteratively build the solutions for these subproblems. +4