Class 9

Detailed Notes

Problem: Matchsticks to Square

Link: Matchsticks to Square

Solution:

```
bool recur(vector<int> &matchsticks, vector<int> &arr, int &ind, long long
int &sidelen) {
    if (ind >= matchsticks.size()) {
        return arr[0] == arr[1] && arr[2] == arr[3] && arr[0] == arr[2];
    }
    for (int i = 0; i < 4; i++) {
        if (arr[i] + matchsticks[ind] > sidelen) continue;
        arr[i] += matchsticks[ind];
        ind++;
        if (recur(matchsticks, arr, ind, sidelen)) return true;
        ind--;
        arr[i] -= matchsticks[ind];
    }
    return false;
}
```

Explanation:

This problem is solved by attempting to form a square using matchsticks, where each side of the square has an equal length. The solution uses a backtracking approach to distribute matchsticks among the four sides of the square:
- It recursively tries to place each matchstick in one of the four sides if adding it does not exceed the target side length (`sidelen`).
- The recursion terminates successfully if all matchsticks are used such that all four sides are equal.

Why it works:

The recursive function efficiently explores all possible ways to distribute matchsticks among the four sides while ensuring that no side exceeds the predetermined side length. This is done by early pruning of branches that can't possibly lead to a solution, thereby optimizing the search.

Time Complexity: O(4^N), as each matchstick has four possible places to go.

Space Complexity: O(N) for recursion depth.

Problem: Partition to K Equal Sum Subsets

Link: [Partition to K Equal Sum Subsets]

Solution:

```
bool recur(vector<bool>& vec, vector<int>& nums, int total, int ind, int
k, int s) {
    if (k == 1) return true;  // Only one group left, no need to check
further
    if (ind == nums.size()) return false;
    if (total == s) return recur(vec, nums, 0, 0, k-1, s);  // One group
complete, continue with the next group
    for (int j = ind; j < nums.size(); j++) {
        if (vec[j] || total + nums[j] > s) continue;
        vec[j] = true;
        if (recur(vec, nums, total + nums[j], j + 1, k, s)) return true;
        vec[j] = false;  // Backtrack
    }
    return false;
}
```

Explanation:

The solution involves partitioning an array into `k` subsets where the sum of each subset equals a target sum. It utilizes backtracking to try assigning each element to one of the subsets:
- If adding an element to a subset causes the subset sum to exceed the target, that path is abandoned (backtracked).
- The recursion continues until all elements are either successfully assigned to a subset or all combinations are exhausted.

Why it works:

This approach ensures that each number is tried in each subset, checking all possible combinations through backtracking. Early termination happens when an invalid or complete path is found, optimizing the exploration.

Time Complexity: $O(k \times 2^N)$ assuming the worst case where each number is considered for each subset.

Space Complexity: $O(N)$ for the recursion stack and the use of an auxiliary array for tracking used elements.

Problem: Closest Dessert Cost

Link: Closest Dessert Cost

Solution:

```
void calculate(int curr_cost, int ind, vector<int>& toppingCosts, int
&target, int &ans) {
    if (curr_cost > ans && (abs(curr_cost - target) >= abs(ans - target)))
return;
    if ((abs(curr_cost - target) == abs(ans -

 target)) && curr_cost < ans) {
        ans = curr_cost;
    } else if (abs(curr_cost - target) < abs(ans - target)) {
        ans = curr_cost;
    }
    if (ind >= toppingCosts.size()) return;
    calculate(curr_cost, ind + 1, toppingCosts, target, ans);
    calculate(curr_cost + toppingCosts[ind], ind + 1, toppingCosts,
target, ans);
    calculate(curr_cost + (toppingCosts[ind] * 2), ind + 1, toppingCosts,
target, ans);
}
```

Explanation:

This problem involves finding the dessert cost closest to a target by choosing a base dessert and adding zero or more of each type of topping (up to two of each type). The solution uses a recursive function that:
- Adds each possible topping in amounts of 0, 1, or 2 and checks if the new total cost is closer to the target than the previous closest cost.
- The recursion stops when all toppings are considered.

Why it works:

The function exhaustively explores all possible combinations of topping additions, updating the closest cost when a better one is found. It optimizes by stopping further exploration if the current path cannot possibly result in a closer cost.

Time Complexity: O(3^T), where T is the number of topping types, as each topping has three possible quantities to be added.

Space Complexity: O(T) for the recursion stack depth, equal to the number of topping types.

Problem: Construct the Lexicographically Largest Valid Sequence

Link: Construct the Lexicographically Largest Valid Sequence

Solution:

```
void fill(int placed, int ind, int n, vector<int> &vec, vector<int> &ans,
vector<int> &vec2) {
    if (placed == n) {
        for (int i = 0; i < vec.size(); i++) {
            ans[i] = vec[i];
        }
        return;
    }
    if (vec[ind] != 0) {
        fill(placed, ind + 1, n, vec, ans, vec2);
        return;
    }
    for (int i = n; i >= 1; i--) {
        if (vec2[i] == 0 && (ind + i < vec.size()) && vec[ind] == 0 &&
vec[ind + i] == 0) {
            vec2[i] = 1;
            vec[ind] = i;
            vec[ind + i] = i;
            fill(placed + 1, ind + 1, n, vec, ans, vec2);
            vec2[i] = 0;
            vec[ind] = 0;
            vec[ind + i] = 0;
        }
    }
}
```

Explanation

The task is to create the lexicographically largest valid sequence using numbers from 1 to `n` where each number `i` appears twice, and the second occurrence of each number appears `i` positions after the first. The solution uses a backtracking approach:
- It tries to place each number in the sequence, respecting the distance constraint.
- If placing a number leads to a valid sequence, the recursion proceeds; otherwise, it backtracks.

Why it works:

This method ensures that each number is placed according to the sequence rules, generating the largest possible sequence by trying higher numbers first. The use of backtracking allows it to abandon unfruitful paths quickly.

Time Complexity: Potentially exponential due to the branching factor of placing each number. Precise complexity is difficult to state without more detailed analysis.

Space Complexity: O(n) for the recursion stack and additional space for sequence storage.