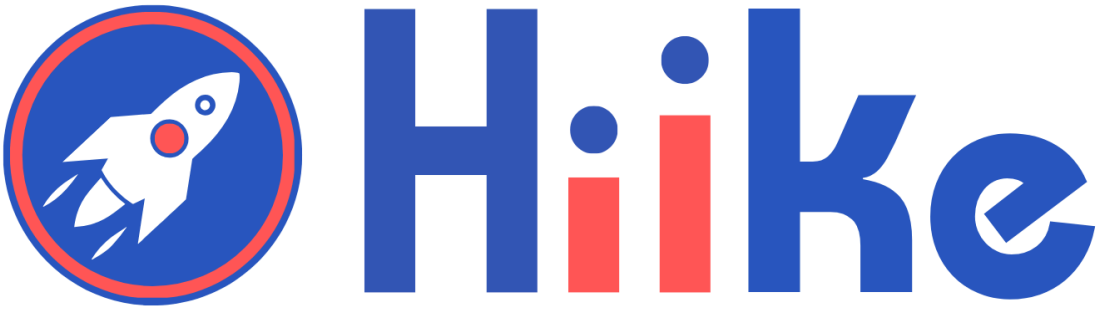
Class 7

Detailed Notes



Problem: Aggressive Cows

Link: [Aggressive Cows]

Solution:

```
class Solution {
public:
    bool isPossible(int mid, vector<int> stalls, int k) {
        int lastcow = 0; // Place the first cow in the first stall
        k--; // One cow is placed
        for (int i = 1; i < stalls.size(); i++) {
            if (stalls[i] - stalls[lastcow] >= mid) {
                k--; // Place another cow
                lastcow = i; // Update the last cow's position
                if (k <= 0) return true; // All cows placed successfully
            }
        }
        return false;
    }
    int solve(int n, int k, vector<int> &stalls) {
        sort(stalls.begin(), stalls.end()); // Sort the stalls
        int low = 0, high = stalls.back(), mid;
        while (low <= high) {
            mid = low + (high - low) / 2;
            if (isPossible(mid, stalls, k) && !isPossible(mid + 1, stalls,
k)) {
                return mid;
            }
            if (isPossible(mid, stalls, k)) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return 0;
    }
};
```

Why it works:

This algorithm uses binary search to determine the largest possible minimum distance (`mid`) that allows all cows to be placed in the stalls without any two cows being closer than `mid`. The function `isPossible` checks if it is possible to place all cows with at least `mid` distance apart.

Time Complexity: O(n log D), where n is the number of stalls and D is the range of possible distances (difference between the smallest and largest stall positions).

Space Complexity: O(1), additional space used for the binary search process.

Problem: Allocate Books

Link: [Allocate Books]

Solution:

```
bool isPossible(int mid, vector<int> &A, int B) {
    int count = 1, sum = 0; // Start with one student
    for (int i = 0; i < A.size(); i++) {
        if (A[i] > mid) return false; // A single book has more pages than
mid, not possible
        sum += A[i];
        if (sum > mid) { // Exceeds the maximum page limit for one student
            count++;
            sum = A[i]; // Start a new set with the current book
            if (count > B) return false; // More students required than
available
        }
    }
    return true;
}
int books(vector<int> &A, int B) {
    if (B > A.size()) return -1; // Not enough books for each student
    int low = *max_element(A.begin(), A.end()), high =
accumulate(A.begin(), A.end(), 0), mid;
    while (low <= high) {
        mid = low + (high - low) / 2;
        if (isPossible(mid, A, B) && !isPossible(mid - 1, A, B)) {
            return mid; // Found the minimum maximum
        }
        if (!isPossible(mid, A, B)) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return 0;
}
```

Why it works:

This solution also employs a binary search technique to find the smallest maximum number of pages that can be allocated to a student under the given constraints. The `isPossible` function checks if it is feasible to distribute all books such that no student gets more than `mid` pages.

Time Complexity: O(n \log S), where n is the number of books and S is the sum of all pages.

Space Complexity: O(1), using only a few additional variables for the search.

Problem: Painters Partition Problem

Link: [Painters Partition Problem]

Solution:

```
bool isValid(int A, vector<int> &C, int mid) {
    int count = 1, sum = 0;
    for (int i = 0; i < C.size(); i++) {
        if (C[i] > mid) return false; // A single board is too long to fit
within the time limit
        sum += C[i];
        if (sum > mid) { // Time exceeded, need a new painter
            count++;
            sum = C[i];
            if (count > A) return false; // More painters required than
available
        }
    }
    return true;
}
int paint(int A, int B, vector<int> &C) {
    int low = *max_element(C.begin(), C.end()), high =
accumulate(C.begin(), C.end(), 0), mid;
    while (low <= high) {
        mid = low + (high - low) / 2;
        if (isValid(A, C, mid) && !isValid(A, C, mid - 1)) {
            return ((long long)mid * B) % 10000003; // Calculate the
minimum time required
        }
        if (!isValid(A, C, mid)) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

Why it works:

This solution uses binary search to determine the minimum maximum time required to paint all boards. The `isValid` function checks if it's possible to paint all boards within a given time `mid` with the available painters, ensuring no painter exceeds the time limit.

Time Complexity: O(n log T), where n is the number of boards and T is the total sum of all board lengths.

Space Complexity: O(1), no extra space beyond the input is used.

Problem: Generate All Parentheses II

Link: [Generate All Parentheses II]

Solution:

```
void populate(vector<string> &str, int A, int open, int close, string
temp){
    if(close == A){
        str.push_back(temp);
        return;
    }
    if(open < A){
        populate(str, A, open + 1, close, temp + '(');
    }
    if(close < open){
        populate(str, A, open, close + 1, temp + ')');
    }
}
vector<string> Solution::generateParenthesis(int A) {
    vector<string> str;
    string temp;
    populate(str, A, 0, 0, temp);
    return str;
}
```

Explanation:

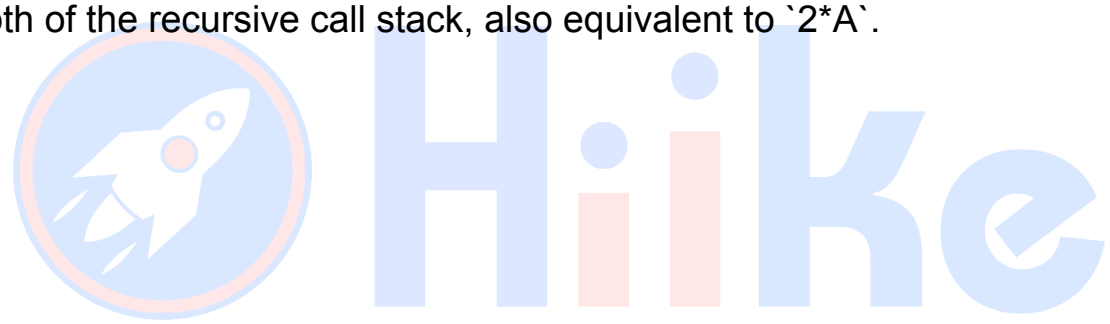
This solution employs recursive backtracking to generate all valid combinations of n pairs of parentheses by adding open or close parentheses based on count checks. The recursion ensures that at any point, the number of close parentheses does not exceed the number of open ones, thereby maintaining the validity of the sequence. This systematic approach explores all configurations where each string adheres to proper nesting rules.

Why it works:

This recursive function generates all valid combinations of parentheses. The base case for the recursion is when the number of closing parentheses matches `A`, indicating a complete valid sequence. During each recursive call, it either adds an opening parenthesis (if there are less than `A` opening parentheses already) or a closing one (if the number of closing parentheses is less than the number of opening ones), ensuring the sequence remains valid.

Time Complexity: O(4^n / \sqrt{n}) due to the nature of the branching recursive calls (each position can almost branch into two more calls).

Space Complexity: O(n) for the recursion stack, where n is the maximum depth of the recursive call stack, also equivalent to `2*A`.



Problem: Letter Phone

Link: [Letter Phone]

Solution

```
void populate(string &A, vector<string> &ans, int ind, string s, map<char,
string> &m){
    if(ind == A.length()){
        ans.push_back(s);
        return;
    }
    string temp = m[A[ind]];
    for(int i = 0; i < temp.length(); i++){
        char ch = temp[i];
        populate(A, ans, ind + 1, s + ch, m);
    }
}
vector<string> Solution::letterCombinations(string A) {
  map<char, string> m;
  m.insert({'1', "1"});
  m.insert({'0', "0"});
  m.insert({'2', "abc"});
  m.insert({'3', "def"});
  m.insert({'4', "ghi"});
  m.insert({'5', "jkl"});
  m.insert({'6', "mno"});
  m.insert({'7', "pqrs"});
  m.insert({'8', "tuv"});
  m.insert({'9', "wxyz"});
  vector<string> ans;
  string str;
  populate(A, ans, 0, str, m);
  return ans;
}
```

Explanation:

The solution for the Letter Phone problem uses recursive backtracking to convert a digit string into all possible letter combinations, according to the mapping found on a telephone keypad. For each digit, the function recurses by appending each corresponding letter, effectively building up all combinations from the digit inputs, reflecting the multi-letter possibilities of modern keypads.

Why it works:

This recursive function explores all combinations of characters that can be formed from a string of digits on a typical mobile keypad. Each recursive call processes the next character in the string, adding all possible letters it represents to the current combination string.

Time Complexity: O(4^n), where n is the length of the input string. This accounts for the worst-case where every digit corresponds to 4 letters (like digits '7' or '9').

Space Complexity: O(n) for the recursive call stack.

Problem: Subsets

Link: Subsets

Solution:

```
void populate(vector<vector<int>> &ans, vector<int> &A, int ind,
vector<int> &temp){
    if(ind == A.size()){
        ans.push_back(temp);
        return;
    }
    populate(ans, A, ind + 1, temp);
    temp.push_back(A[ind]);
    populate(ans, A, ind + 1, temp);
    temp.pop_back();
}
vector<vector<int>> Solution::subsets(vector<int> &A) {
    sort(A.begin(), A.end());
    vector<vector<int>> ans;
    vector<int> temp;
    populate(ans, A, 0, temp);
    sort(ans.begin(), ans.end());
    return ans;
}
```

Explanation:

This recursive approach generates all subsets of a given set by deciding, for each element, whether to include it in the current subset or not. The solution ensures that every possible combination of the input set's elements is considered, producing subsets ranging from the empty set to the full set, organized systematically through binary decisions at each element.

Why it works:

This recursive function generates all possible subsets of a given set of integers. Each element has two options: either it is included in the current subset or it is not. This creates a binary tree of recursive calls, effectively exploring all combinations.

Time Complexity: O(2^n * n), where 2^n is the number of subsets and n is the average time to copy these subsets into the answer list.

Space Complexity: O(n) due to the depth of the recursion and the space needed for the current subset being constructed.

Problem: Letter Case Permutation

Link: [Letter Case Permutation]

Solution Explanation:

```cpp
class Solution {
public:
    vector<string> letterCasePermutation(string S) {
        vector<string> ans;
        backtrack(S, 0, ans);
        return ans;
    }

    void backtrack(string &s, int i, vector<string> &ans) {
        if (i == s.size()) {
            ans.push_back(s);
            return;
        }
        char c = s[i];
        s[i] = islower(c) ? toupper(c) : tolower(c);
        backtrack(s, i + 1, ans);
        if (isalpha(c)) {
            s[i] = c;
            backtrack(s, i + 1, ans);
        }
    }
};
```

Explanation:

This function generates all possible strings by toggling the case of each alphabetical character in the input string using a recursive backtracking approach. It explores both the modified (case-toggled) and unmodified paths for each alphabetic character, ensuring all combinations of case changes are considered, which effectively captures all permutations of letter case within the string.

Why it works:

This function generates all possible strings by toggling the case of each alphabetical character in the input string `S`. It recursively tries both the lowercase and uppercase versions of each character if applicable.

Time Complexity: O(2^k * n), where k is the number of alphabetic characters in the string S, and n is the length of S (for creating a string copy in the worst case).

Space Complexity: O(n), mainly for the recursion stack, where n is the length of the string S.