Lecture 12: Advanced Topics (String Algorithms, Tries, Dynamic Programming)

C++ Code Samples — Sedgwick Algorithms Course — lecture-12-samples.cpp

```cpp
/******************************************************************************
 * Lecture 12: Advanced Topics (String Algorithms, Tries, Dynamic Programming)
 *
 * Topics covered:
 *   1. Trie (prefix tree) insert/search/startsWith
 *   2. KMP string matching algorithm
 *   3. Longest Common Subsequence (DP)
 *   4. 0/1 Knapsack (DP)
 *   5. Fibonacci with memoization vs naive recursion timing
 *
 * Compile: g++ -std=c++17 -o lecture-12 lecture-12-samples.cpp
 * Run:     ./lecture-12
 ******************************************************************************/

#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
#include <chrono>

using namespace std;

// === SECTION: Trie (Prefix Tree) ===
// A tree where each node represents a character. Paths from root to marked
// nodes spell out stored words. Supports O(L) insert, search, and prefix
// queries where L is the length of the word/prefix.

struct TrieNode {
    unordered_map<char, TrieNode*> children;
    bool isEnd = false;

    ~TrieNode() {
        for (auto& [ch, child] : children) {
            delete child;
        }
    }
};

class Trie {
    TrieNode* root;

public:
    Trie() : root(new TrieNode()) {}
    ~Trie() { delete root; }

    // Insert a word into the trie. Time: O(L).
    void insert(const string& word) {
        TrieNode* node = root;
        for (char ch : word) {
            if (node->children.find(ch) == node->children.end()) {
                node->children[ch] = new TrieNode();
            }
            node = node->children[ch];
        }
        node->isEnd = true;
    }

    // Search for an exact word. Time: O(L).
    bool search(const string& word) const {
        TrieNode* node = findNode(word);
```

```cpp
        return node != nullptr && node->isEnd;
    }

    // Check if any word starts with the given prefix. Time: O(L).
    bool startsWith(const string& prefix) const {
        return findNode(prefix) != nullptr;
    }

private:
    TrieNode* findNode(const string& s) const {
        TrieNode* node = root;
        for (char ch : s) {
            auto it = node->children.find(ch);
            if (it == node->children.end()) return nullptr;
            node = it->second;
        }
        return node;
    }
};

// === SECTION: KMP String Matching ===
// Knuth-Morris-Pratt algorithm finds all occurrences of a pattern in a text.
// Key idea: precompute a "failure function" (longest proper prefix that is
// also a suffix) so we never re-examine text characters.
// Time: O(N + M) where N = text length, M = pattern length.

// Build the failure function (also called the "partial match table" or "lps").
vector<int> buildKMPTable(const string& pattern) {
    int m = pattern.size();
    vector<int> lps(m, 0);
    int len = 0;   // length of the previous longest prefix suffix
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];   // fall back
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}

// Find all starting indices where pattern occurs in text.
vector<int> kmpSearch(const string& text, const string& pattern) {
    vector<int> result;
    int n = text.size(), m = pattern.size();
    if (m == 0 || m > n) return result;

    vector<int> lps = buildKMPTable(pattern);
    int i = 0;   // index in text
    int j = 0;   // index in pattern

    while (i < n) {
        if (text[i] == pattern[j]) {
            i++;
            j++;
```

```
        }

        if (j == m) {
            result.push_back(i - j);  // match found at index i - j
            j = lps[j - 1];           // continue searching for next match
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return result;
}

// === SECTION: Longest Common Subsequence (DP) ===
// Find the length of the longest subsequence common to two strings.
// A subsequence does not need to be contiguous.
// DP recurrence:
//   if s1[i-1] == s2[j-1]:  dp[i][j] = dp[i-1][j-1] + 1
//   else:                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
// Time: O(N * M), Space: O(N * M).

string longestCommonSubsequence(const string& s1, const string& s2) {
    int n = s1.size(), m = s2.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    // Fill DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // Backtrack to find the actual subsequence
    string lcs;
    int i = n, j = m;
    while (i > 0 && j > 0) {
        if (s1[i - 1] == s2[j - 1]) {
            lcs += s1[i - 1];
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
    reverse(lcs.begin(), lcs.end());
    return lcs;
}

// === SECTION: 0/1 Knapsack (DP) ===
// Given N items, each with a weight and value, and a knapsack with capacity W,
// find the maximum total value that fits in the knapsack.
// Each item can be taken at most once (0/1 choice).
// DP recurrence:
//   dp[i][w] = max(dp[i-1][w],                   // skip item i
//                  dp[i-1][w-weight[i]] + val[i]) // take item i (if it fits)
```

```cpp
// Time: O(N * W), Space: O(N * W).

struct KnapsackResult {
    int maxValue;
    vector<int> selectedItems;  // indices of items chosen
};

KnapsackResult knapsack01(const vector<int>& weights, const vector<int>& values,
                          int capacity) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    // Fill DP table
    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            dp[i][w] = dp[i - 1][w];  // skip item i
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i][w],
                               dp[i - 1][w - weights[i - 1]] + values[i - 1]);
            }
        }
    }

    // Backtrack to find which items were selected
    KnapsackResult result;
    result.maxValue = dp[n][capacity];
    int w = capacity;
    for (int i = n; i >= 1; i--) {
        if (dp[i][w] != dp[i - 1][w]) {
            result.selectedItems.push_back(i - 1);  // item index (0-based)
            w -= weights[i - 1];
        }
    }
    reverse(result.selectedItems.begin(), result.selectedItems.end());
    return result;
}

// === SECTION: Fibonacci - Memoization vs Naive ===
// Classic example of how memoization transforms exponential O(2^n) into O(n).

// Naive recursive Fibonacci: O(2^n) time, O(n) stack space
long long fibNaive(int n) {
    if (n <= 1) return n;
    return fibNaive(n - 1) + fibNaive(n - 2);
}

// Memoized Fibonacci: O(n) time, O(n) space
long long fibMemo(int n, vector<long long>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    return memo[n];
}

long long fibMemoized(int n) {
    vector<long long> memo(n + 1, -1);
    return fibMemo(n, memo);
}

// === MAIN: Demos ===

int main() {
    cout << "==================================================\n";
    cout << "  Lecture 12: String Algorithms, Tries, and DP\n";
```

```cpp
    cout << "==================================================\n\n";

    // --- Trie Demo ---
    cout << "--- Trie (Prefix Tree) ---\n";
    Trie trie;
    vector<string> words = {"apple", "app", "application", "banana", "band"};
    for (const string& w : words) {
        trie.insert(w);
        cout << "  Inserted: \"" << w << "\"\n";
    }
    cout << "\n  Searches:\n";
    vector<string> queries = {"app", "apple", "ap", "ban", "banana", "bandana"};
    for (const string& q : queries) {
        cout << "    search(\"" << q << "\") = "
             << (trie.search(q) ? "FOUND" : "not found")
             << "    startsWith(\"" << q << "\") = "
             << (trie.startsWith(q) ? "YES" : "no") << "\n";
    }
    cout << "\n";

    // --- KMP Demo ---
    cout << "--- KMP String Matching ---\n";
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";
    cout << "  Text:    \"" << text << "\"\n";
    cout << "  Pattern: \"" << pattern << "\"\n";

    vector<int> lps = buildKMPTable(pattern);
    cout << "  LPS table: ";
    for (int v : lps) cout << v << " ";
    cout << "\n";

    vector<int> matches = kmpSearch(text, pattern);
    if (matches.empty()) {
        cout << "  No matches found.\n";
    } else {
        cout << "  Pattern found at indices: ";
        for (int idx : matches) cout << idx << " ";
        cout << "\n";
        // Show the match visually
        for (int idx : matches) {
            cout << "    ";
            for (int i = 0; i < idx; i++) cout << " ";
            cout << pattern << " (index " << idx << ")\n";
        }
    }
    cout << "\n";

    // --- LCS Demo ---
    cout << "--- Longest Common Subsequence (DP) ---\n";
    string s1 = "ABCBDAB";
    string s2 = "BDCAB";
    string lcs = longestCommonSubsequence(s1, s2);
    cout << "  s1 = \"" << s1 << "\"\n";
    cout << "  s2 = \"" << s2 << "\"\n";
    cout << "  LCS = \"" << lcs << "\" (length " << lcs.size() << ")\n\n";

    // --- 0/1 Knapsack Demo ---
    cout << "--- 0/1 Knapsack (DP) ---\n";
    // Items: (weight, value)
    vector<string> itemNames = {"Laptop", "Camera", "Book", "Headphones", "Tablet"};
    vector<int> weights = {3, 1, 2, 1, 4};
    vector<int> values  = {40, 15, 20, 10, 50};
    int capacity = 6;
```

```
    cout << "  Items:\n";
    for (int i = 0; i < (int)itemNames.size(); i++) {
        cout << "    " << itemNames[i] << ": weight=" << weights[i]
             << ", value=" << values[i] << "\n";
    }
    cout << "  Knapsack capacity: " << capacity << "\n";

    KnapsackResult knap = knapsack01(weights, values, capacity);
    cout << "  Maximum value: " << knap.maxValue << "\n";
    cout << "  Selected items: ";
    int totalW = 0;
    for (int idx : knap.selectedItems) {
        cout << itemNames[idx] << " ";
        totalW += weights[idx];
    }
    cout << "\n  Total weight used: " << totalW << "/" << capacity << "\n\n";

    // --- Fibonacci Timing Demo ---
    cout << "--- Fibonacci: Naive vs Memoized ---\n";
    int fibN = 40;
    cout << "  Computing fib(" << fibN << ")...\n";

    // Memoized version (fast)
    auto startMemo = chrono::high_resolution_clock::now();
    long long resultMemo = fibMemoized(fibN);
    auto endMemo = chrono::high_resolution_clock::now();
    auto durationMemo = chrono::duration_cast<chrono::microseconds>(
        endMemo - startMemo);
    cout << "  Memoized:  fib(" << fibN << ") = " << resultMemo
         << "  (" << durationMemo.count() << " microseconds)\n";

    // Naive version (slow -- O(2^n))
    auto startNaive = chrono::high_resolution_clock::now();
    long long resultNaive = fibNaive(fibN);
    auto endNaive = chrono::high_resolution_clock::now();
    auto durationNaive = chrono::duration_cast<chrono::milliseconds>(
        endNaive - startNaive);
    cout << "  Naive:     fib(" << fibN << ") = " << resultNaive
         << "  (" << durationNaive.count() << " milliseconds)\n";

    cout << "\n  Speedup: memoization reduces O(2^n) to O(n)\n";
    cout << "  The naive version makes ~2^" << fibN << " = ~"
         << (1LL << min(fibN, 40)) << " recursive calls!\n";

    return 0;
}
```