Reporte de Soluciones de Programación

Alumno: Héctor Javier Salazar Álvarez

División de Ciencias Naturales y Exactas
Universidad de Guanajuato

1 de junio de 2025

1. La Bolsa del Ladrón

1.1. Descripción del Problema

Dada una mochila con capacidad máxima W y N objetos con pesos y valores asociados, seleccionar los objetos que maximicen el valor total sin exceder la capacidad.

1.2. Solución Implementada

Se utilizó programación dinámica con un arreglo unidimensional que almacena los valores máximos posibles para cada capacidad.

1.3. Intuición del Algoritmo

- La solución construye gradualmente la respuesta considerando cada objeto
- Para cada capacidad posible, decide si incluir o no el objeto actual
- El enfoque "de derecha a izquierda.evita contar el mismo objeto múltiples veces

1.4. Código Implementado

Listing 1: Solución al Problema de la Mochila

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int N, W;
    cin >> N >> W;

    vector<int> pesos(N);
    vector<int> valores(N);

    for (int i = 0; i < N; ++i) {
        cin >> pesos[i] >> valores[i];
    }

    vector<int> dp(W + 1, 0);

    for (int i = 0; i < N; ++i) {
        int peso = pesos[i];
        int valor = valores[i];

        for (int w = W; w >= peso; --w) {
            if (dp[w - peso] + valor > dp[w]) {
                dp[w] = dp[w - peso] + valor;
            }
```

```
            }
        }

        cout << dp[W] << endl;

        return 0;
}
```

1.5. Pruebas Realizadas

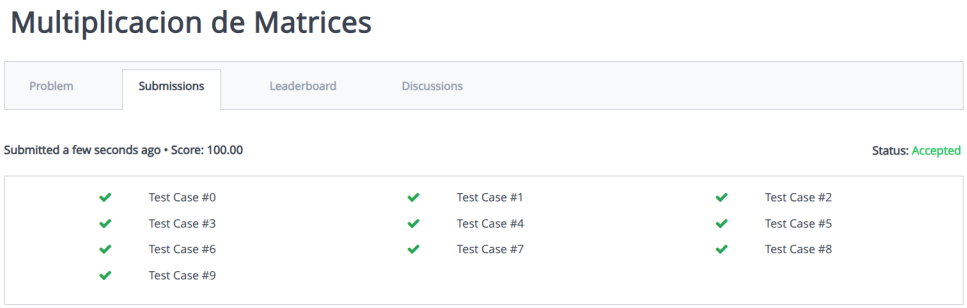

Figura 1: Resultados de las pruebas para el problema de la mochila

2. Multiplicación de Matrices en Cadena

2.1. Descripción del Problema

Dada una secuencia de matrices, encontrar el orden de multiplicación que minimice el número de operaciones escalares.

2.2. Solución Implementada

Programación dinámica con tabla bidimensional que almacena costos mínimos para cada subcadena de matrices.

2.3. Intuición del Algoritmo

- Divide el problema en subproblemas más pequeños (subcadenas de matrices)
- Calcula el costo para todas las posibles particiones de cada subcadena
- Combina soluciones de subproblemas para construir la solución global

2.4. Código Implementado

Listing 2: Solución para Multiplicación de Matrices en Cadena

```
#include <iostream>
#include <vector>
#include <climits>
```

```
using namespace std;

int main() {
    int n;
    cin >> n;

    vector<int> p(n + 1);
    for (int i = 0; i <= n; ++i) {
        cin >> p[i];
    }

    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int length = 2; length <= n; ++length) {
        for (int i = 0; i <= n - length; ++i) {
            int j = i + length - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; k < j; ++k) {
                int cost = dp[i][k] + dp[k+1][j] + p[i] * p[k+1] * p[j+1];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }

    cout << dp[0][n-1] << endl;

    return 0;
}
```

2.5. Pruebas Realizadas

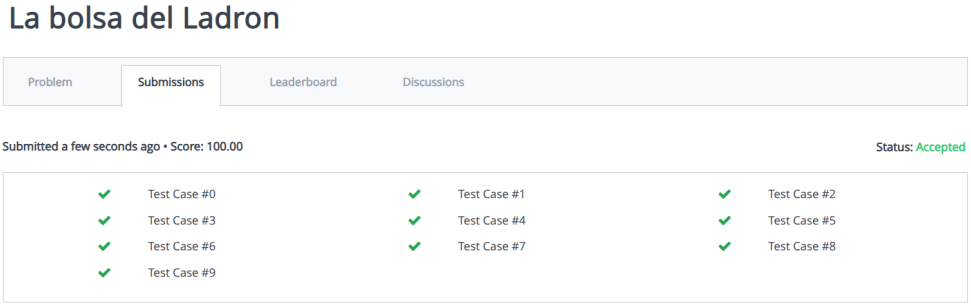

Figura 2: Resultados de las pruebas para multiplicación de matrices

3. Subsecuencia Común Más Larga (LCS)

3.1. Descripción del Problema

Encontrar la subsecuencia más larga presente en dos strings dados, donde una subsecuencia mantiene el orden relativo pero no necesariamente la continuidad.

3.2. Solución Implementada

Programación dinámica con tabla que registra longitudes de LCS para todos los prefijos de ambos strings.

3.3. Intuición del Algoritmo

- Compara caracteres uno por uno de ambos strings
- Cuando hay coincidencia, extiende la LCS encontrada hasta ese punto
- Cuando no hay coincidencia, toma el máximo de las soluciones anteriores

3.4. Código Implementado

Listing 3: Solución para Subsecuencia Común Más Larga

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int main() {
    string s, p;
    cin >> s >> p;

    int m = s.size();
    int n = p.size();

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (s[i-1] == p[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    cout << dp[m][n] << endl;
```

```
    return 0;
}
```

3.5. Pruebas Realizadas

Figura 3: Resultados de las pruebas para LCS