INFORME DE LABORATORIO

INFORMACIÓN BÁSICA

ASIGNATURA:	ANÁLISIS Y DISEÑO DE ALGORITMOS				
TÍTULO DE LA PRÁCTICA:	PROGRAMACIÓN DINÁMICA				
NÚMERO DE PRÁCTICA:	P2	**AÑO LECTIVO:**	2024	**SEMESTRE:**	PAR
ESTUDIANTES: 20224230, Quispe Díaz Sara Sofía					
DOCENTES: Marcela Quispe Cruz, Manuel Loaiza, Alexander J. Benavides					

RESULTADOS Y PRUEBAS

El informe se presenta con un formato de artículo.
Revise la sección de *Resultados Experimentales*.

CONCLUSIONES

El informe se presenta con un formato de artículo.
Revise la sección de *Conclusiones*.

METODOLOGÍA DE TRABAJO

El informe se presenta con un formato de artículo.
Revise la sección de *Diseño Experimental*.

REFERENCIAS Y BIBLIOGRAFÍA

El informe se presenta con un formato de artículo.
Revise la sección de *Referencias Bibliográficas*.

Programación Dinámica – Ejemplos de Aplicación

Resumen

En este artículo, se abordan tres problemas utilizando técnicas de programación dinámica, una herramienta clave para optimizar soluciones recursivas en problemas que presentan subestructura óptima y subproblemas superpuestos. Los primeros dos problemas están relacionados con la maximización, empleando un enfoque iterativo para identificar la combinación óptima de tiempos. El tercer problema explora un enfoque de hashing simple para asignar valores numéricos a caracteres y calcular cuántas combinaciones cumplen condiciones específicas. En todos los casos, se utilizó programación dinámica, junto con estrategias como memoización y la técnica SRTBOT (Search, Recursion, and Time-based Optimization), para optimizar los resultados y mejorar la eficiencia.

1. Introducción

La programación dinámica es una técnica fundamental para la resolución eficiente de problemas complejos que involucran decisiones secuenciales y subproblemas repetitivos. En este trabajo, se aborda el uso de la programación dinámica para optimizar la solución de problemas de maximización de recursos y combinatoria. A través de su aplicación, se logra reducir el tiempo de ejecución de algoritmos y mejorar la eficiencia en la resolución de problemas que de otro modo serían computacionalmente costosos.

El principal objetivo de este trabajo fue explorar cómo las técnicas de programación dinámica, en combinación con estrategias como la memoización, pueden ser utilizadas para resolver problemas de optimización, como maximizar el consumo de recursos en un tiempo limitado y calcular combinaciones bajo restricciones específicas. A lo largo del artículo, se describen los problemas propuestos, las metodologías empleadas y los resultados obtenidos.

La estructura del artículo es la siguiente: La Sección 2 presenta el marco teórico y los conceptos clave sobre programación dinámica, así como su aplicación en la resolución de problemas similares. En la Sección 3, se detalla el diseño experimental utilizado, incluyendo la selección y resolución de los problemas, así como los métodos empleados para obtener y evaluar los resultados. La Sección 4 muestra los resultados obtenidos, incluyendo los códigos, pseudocódigos y explicaciones de las soluciones implementadas. Finalmente, la Sección 5 ofrece las conclusiones, donde se reflexiona sobre los logros alcanzados y las dificultades superadas durante la implementación de las soluciones.

2. Marco Teórico Conceptual

Programación dinámica fue propuesta por Bellman (1952), como un enfoque para resolver problemas de optimización complejos que pueden dividirse en subproblemas más pequeños. Esta técnica se utiliza para resolver problemas recursivos, donde la solución de un problema depende de las soluciones de subproblemas más pequeños. Sin embargo, en lugar de recalcular la solución de un subproblema varias veces, la PD guarda los resultados de los subproblemas resueltos previamente, lo que se conoce como memoización.

El nemotécnico SRTBOT propuesto por Demaine (2021), es una guía para diseñar algoritmos recursivos eficientes en la resolución de problemas complejos. Cada letra de SRTBOT corresponde a un concepto clave en el diseño y la optimización de algoritmos recursivos.

Subproblemas El primer paso en la aplicación de SRTBOT consiste en dividir el problema original en subproblemas más pequeños y manejables. Esto se hace buscando estructuras repetitivas dentro

del problema principal que puedan ser resueltas de forma independiente, y sus soluciones combinadas generen la solución final.

Relaciones Recursivas Una vez que el problema está dividido en subproblemas, es crucial identificar las relaciones recursivas. Estas son ecuaciones o fórmulas que describen cómo la solución de un problema depende de las soluciones de sus subproblemas. En la programación dinámica, estas relaciones suelen expresarse de manera que las soluciones anteriores puedan ser reutilizadas para resolver los subproblemas de manera más eficiente.

Topología El paso de topología implica dibujar o representar el flujo de las decisiones o subproblemas en forma de un grafo o diagrama de recursión. Este enfoque ayuda a visualizar cómo se interconectan los subproblemas, cómo fluye la información y cuál es el impacto de cada subproblema en la solución final. Entender la topología facilita la implementación del algoritmo recursivo y permite identificar oportunidades de optimización.

Bases Las bases corresponden a los casos base de la recursión, que son los subproblemas más simples que se pueden resolver directamente sin necesidad de dividirlos más. Estos casos base son fundamentales, ya que marcan el punto de inicio de la recursión y permiten que el algoritmo termine correctamente.

Original El concepto de original se refiere a la formulación del problema principal después de haber considerado los subproblemas, relaciones recursivas, topología y bases. Una vez que estos aspectos se han definido claramente, se puede diseñar el algoritmo recursivo que resuelva el problema original combinando eficientemente las soluciones de los subproblemas. Además, en esta etapa, se incorpora la técnica de memoización, que optimiza el algoritmo al almacenar los resultados de subproblemas resueltos para evitar cálculos repetidos.

Tiempo El análisis de tiempo de ejecución es una etapa crítica que evalúa la eficiencia del algoritmo diseñado. Este análisis consiste en calcular la cantidad de tiempo necesario para resolver el problema original, considerando las interacciones entre los subproblemas y el número de llamadas recursivas que se realizan. El objetivo es determinar si la solución es viable desde el punto de vista del rendimiento, y si es necesario, aplicar técnicas adicionales de optimización, como la memoización o la tabulación, para mejorar la eficiencia.

3. Diseño Experimental

El proceso para obtener los resultados experimentales comenzó con la selección de los problemas, los cuales fueron elegidos en función de su capacidad para aplicar técnicas de programación dinámica. En primer lugar, se descompuso cada problema en subproblemas más pequeños, lo que permitió la utilización de la técnica de memorización para optimizar el cálculo de soluciones parciales. Posteriormente, se implementaron algoritmos iterativos para evaluar las soluciones, siempre con el objetivo de maximizar la eficiencia en cuanto a tiempo y recursos.

Se lograron los objetivos propuestos, obteniendo soluciones correctas para cada uno de los problemas planteados. El trabajo permitió optimizar el consumo de hamburguesas en un tiempo limitado y resolver un problema de combinaciones de caracteres utilizando técnicas de hashing. Sin embargo, se presentaron algunas dificultades, principalmente al tratar de optimizar el tiempo de ejecución y al implementar la memoización, lo que requirió ajustes en la estructura del código y la depuración de errores.

3.1. Objetivos

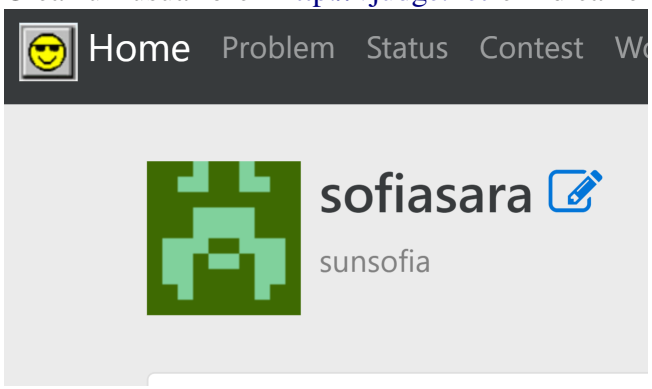
Los objetivos de este trabajo son:

- Profundizar en el uso de la programación dinámica como técnica de optimización.
- Aplicar programación dinámica en problemas de optimización de recursos y combinatoria.
- Demostrar la efectividad de la técnica SRTBOT para mejorar la eficiencia en la resolución de problemas con múltiples subproblemas interrelacionados.

3.2. Actividades

El estudiante deberá realizar las siguientes acciones.

1. Crear un usuario en http://vjudge.net e indicar en este paso el nombre de usuario utilizado.



2. Seleccionar aleatoriamente tres problemas de la lista disponible en http://bit.ly/3UxdCVL.

	Origin	Title
✔	UVA-10465	Homer Simpson
✔	UVA-10912	Simple Minded Hashing
✔	UVA-11420	Chest of Drawers

4. Resultados

A continuación se muestran como ejemplo los resultados para tres problemas seleccionados.

4.1. Problema 10465 – Homer Simpson

Subproblema: Sea $S(i)$ el máximo número de hamburguesas que Homer puede comer en exactamente i minutos. Entonces, el subproblema consiste en calcular:

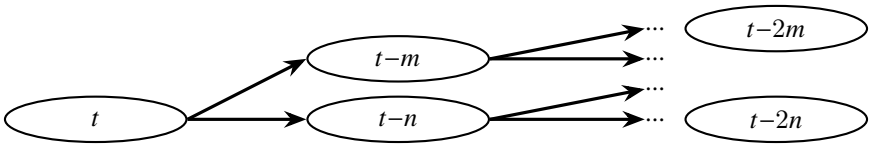
$$S(i) = \text{máx}(S(i-m)+1, S(i-n)+1)$$

Relaciones Recursivas:

$$S(i) = max(\ S(i-m)+1, S(i-n)+1)$$

$$\text{Si } i \geq m \text{ o } i \geq n$$

Topología:



Básico: $S(0) = 0$

Original:

Algorithm S(i) // sin memoización
Input: Tiempo total t, cantidad m, cantidad n
Output: Máximo número de hamburguesas y tiempo restante

function S(t)
  if $t = 0$ **then**
    return 0
  else if $t < 0$ **then**
    return $-\infty$
  else
    return $\max(S(t-m)+1, S(t-n)+1)$
$max\_burgersS(t)$
if $max\_burgers < 0$ **then**
  for $i = t$ **downto** 0 **do**
    if $S(i) \geq 0$ **then** $max\_burgersS(i)$
$tiempo\_sobrantet - i$
else$tiempo\_sobrante0$

return $(max\_burgers, tiempo\_sobrante)$

Algorithm S(i) // con memoización
Input: Tiempo total t, cantidad m, cantidad n
Output: Máximo número de hamburguesas y tiempo restante

Crear un array M de tamaño $t+1$, inicializado en -1
function SM(t)
  if $t = 0$ **then**
    return 0
  else if $t < 0$ **then**
    return $-\infty$
  else if $M[t] \neq -1$ **then**
    return $M[t]$
  else$M[t]\max(SM(t-m)+1, SM(t-n)+1)$
    return $M[t]$
$max\_burgersSM(t)$
if $max\_burgers < 0$ **then**
  for $i = t$ **downto** 0 **do**
    if $M[i] \geq 0$ **then** $max\_burgersM[i]$
$tiempo\_sobrantet - i$
else$tiempo\_sobrante0$

return $(max\_burgers, tiempo\_sobrante)$

Tiempo: $SM(t) \in O(t)$

Código:

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

int main() {
    int m, n, t;
    while (cin >> m >> n >> t) {
        int dp[10001];
        memset(dp, -1, sizeof(dp));
        dp[0] = 0;

        for (int i = 1; i <= t; ++i) {
            if (i >= m && dp[i - m] != -1) {
                dp[i] = max(dp[i], dp[i - m] + 1);
            }
            if (i >= n && dp[i - n] != -1) {
                dp[i] = max(dp[i], dp[i - n] + 1);
            }
        }

        if (dp[t] != -1) {
            cout << dp[t] << endl;
        } else {
            int max_burgers = 0, remaining_time = t;
            for (int i = t; i >= 0; --i) {
                if (dp[i] != -1) {
                    max_burgers = dp[i];
                    remaining_time = t - i;
                    break;
                }
            }
            cout << max_burgers << " " << remaining_time << endl;
        }
    }
    return 0;
}
```

4.2. Problema 11420 – Chest of Drawers

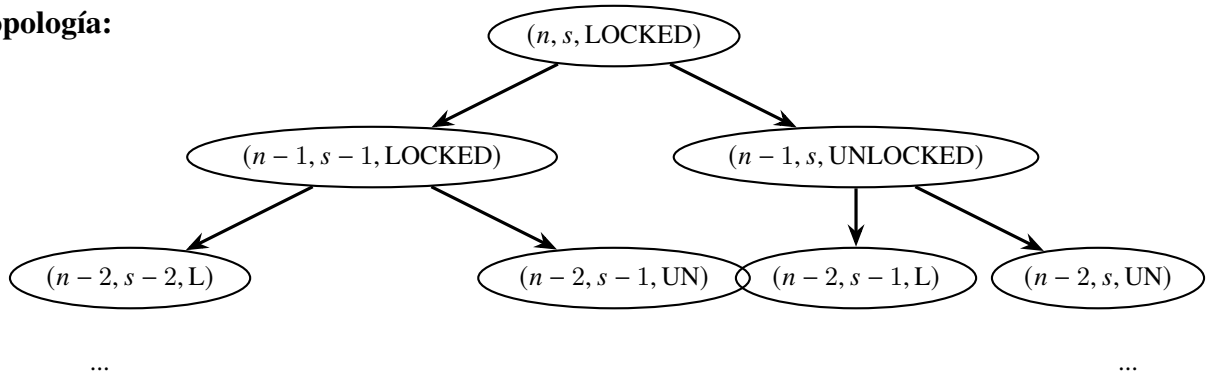
Subproblema: Encuentra $C(i, s)$, que representa el número de formas de asegurar exactamente s cajones en una cómoda de i cajones. Cada cajón puede estar bloqueado (LOCKED) o desbloqueado (UNLOCKED), y un cajón es seguro si está bloqueado y todos los cajones superiores también lo están.

Relaciones Recursivas:

$$\text{solve}(i, s, \text{LOCKED}) = \text{solve}(i-1, s-1, \text{LOCKED}) + \text{solve}(i-1, s, \text{UNLOCKED})$$

$$\text{solve}(i, s, \text{UNLOCKED}) = \text{solve}(i-1, s, \text{LOCKED}) + \text{solve}(i-1, s, \text{UNLOCKED})$$

Topología:



Básico: solve(0, s, prev) = 0 si s ≠ 0

Original:

Algorithm solve_memo(i, s, prev) // sin memoización

Input: posición i, cantidad de cajones seguros s, estado previo prev

Output: cuántas maneras de asegurar exactamente s cajones

```
1: if i < 0 or s < 0 then
2:     return 0
3: else if i = 0 and s = 0 then
4:     return 1
5: else if i = 0 then
6:     return 0
7: else
8:     if prev = LOCKED then
9:         return solve(i−1, s−1, LOCKED)+ solve(i−1, s, UNLOCKED)
10:    else
11:        return solve(i−1, s, LOCKED)+ solve(i−1, s, UNLOCKED)
```

Algorithm solve_memo(i, s, prev) // con memoización

Input: posición i, cantidad de cajones seguros s, estado previo prev

Output: cuántas maneras de asegurar exactamente s cajones

```
1: if i < 0 or s < 0 then
2:     return 0
3: else if i = 0 and s = 0 then
4:     return 1
5: else if i = 0 then
6:     return 0
7: if dp[i][s][prev] is not undefined then
8:     return dp[i][s][prev]
9: if prev = LOCKED then
10:    dp[i][s][prev] = solve_memo(i−1, s−1, LOCKED)+ solve_memo(i−1, s, UNLOCKED)
11: else
12:    dp[i][s][prev] = solve_memo(i−1, s, LOCKED)+ solve_memo(i−1, s, UNLOCKED)
13: return dp[i][s][prev]
```

Tiempo: solve$(i, s, prev) \in O(n * s * 2)$

Código:

```cpp
#include <iostream>
#include <cstdio>
#include <cstring>

#define ll long long
using namespace std;
#define LOCKED 0
#define UNLOCKED 1

ll dp[70][70][2];

ll solve(int pos, int cntSecure, int prev) {
    if (pos < 0 || cntSecure < 0) return 0;
    if (pos == 0 && cntSecure == 0) return 1;
    if (pos == 0) return 0;

    ll &ret = dp[pos][cntSecure][prev];
    if (ret != -1) return ret;

    if (prev == LOCKED) {
        ret = solve(pos - 1, cntSecure - 1, LOCKED) +
        solve(pos - 1, cntSecure, UNLOCKED);
    } else {
        ret = solve(pos - 1, cntSecure, LOCKED) +
        solve(pos - 1, cntSecure, UNLOCKED);
    }

    return ret;
}

int main() {
    int n, s;

    memset(dp, -1, sizeof(dp));

    while (scanf("%d %d", &n, &s) == 2) {
        if (n < 0 || s < 0) break;

        ll result = solve(n, s, LOCKED);

        printf("%lld\n", result);
    }

    return 0;
}
```

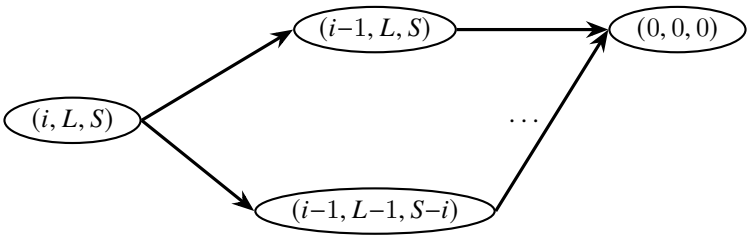
4.3. Problema 10912 – Simple Minded Hashing

Subproblema: $DP[i][j][k]$ como el número de formas de crear una cadena de longitud j con una suma de valores igual a k, utilizando las primeras i letras en orden ascendente.

Relaciones Recursivas:

$$DP[i][j][k] = DP[i-1][j][k] + DP[i-1][j-1][k-i] \quad \text{si } j > 0 \text{ y } k \geq i$$

Topología:



Básico:

$$DP[0][0][0] = 1$$

Original:

Algorithm countS(i, L, S) // sin memoización
Input: número de letras i, longitud de la cadena L, suma objetivo S
Output: número de cadenas

```
1:  function countStrings(i, L, S)
2:      if L = 0 and S = 0 then
3:
4:          return 1
5:      if i = 0 or L < 0 or S < 0 then
6:
7:          return 0
8:      result = countStrings(i − 1, L, S)
9:      if L > 0 and S ≥ i then
10:         result+ = countStrings(i − 1, L − 1, S − i)
11:
12:     return result
```

Algorithm countS(i, L, S) // con memoización
Input: número de letras i, longitud de la cadena L, suma objetivo S
Output: número de cadenas

```
1: inicializar DP como un arreglo 3D de ceros
2: DP[0][0][0] = 1
3: for i ← 1 to 26 do
4:     for j ← 0 to i do
5:         for k ← 0 to 351 do
6:             DP[i][j][k] = DP[i − 1][j][k]
7:             if j > 0 and k ≥ i then
8:                 DP[i][j][k]+ = DP[i−1][j−1][k−i]
```

Tiempo:

$$O(26 \times 26 \times 352) = O(237,952)$$

Código:

```c
#include <stdio.h>
#include <string.h>

int DP[27][27][352];

void build() {
    memset(DP, 0, sizeof(DP));
    DP[0][0][0] = 1;

    for (int i = 1; i <= 26; i++) {
        for (int j = 0; j <= i; j++) {
            for (int k = 0; k < 352; k++) {
                DP[i][j][k] = DP[i-1][j][k];
                if (j > 0 && k >= i) {
                    DP[i][j][k] += DP[i-1][j-1][k-i];
                }
            }
        }
    }
}
```

```c
int main() {
    build();
    int L, S, Case = 0;

    while (scanf("%d %d", &L, &S) == 2) {
        if (L == 0 && S == 0)
            break;

        printf("Case %d: ", ++Case);

        if (L > 26 || S > 351)
            puts("0");
        else
            printf("%d\n", DP[26][L][S]);
    }
    return 0;
}
```

5. Conclusiones

Este trabajo ha permitido explorar cómo la programación dinámica, junto con técnicas de optimización como memoización e iteración, mejora significativamente la eficiencia en la resolución de problemas complejos. En los primeros dos problemas, optimizamos el cálculo del número máximo de hamburguesas que Homer puede comer sin perder tiempo, utilizando programación dinámica para encontrar la combinación más eficiente. En el tercer problema, resolvimos una variante del hashing mediante un enfoque iterativo eficiente, utilizando PD para explorar todas las combinaciones posibles.

Además, la incorporación de la técnica SRTBOT permitió mejorar la eficiencia de las soluciones, enfocándose en la búsqueda de subproblemas relevantes, la optimización de tiempo y el uso adecuado de la recursión. Esta práctica demuestra la potencia de la programación dinámica y técnicas asociadas para resolver problemas combinatorios y de optimización de recursos de manera eficaz.

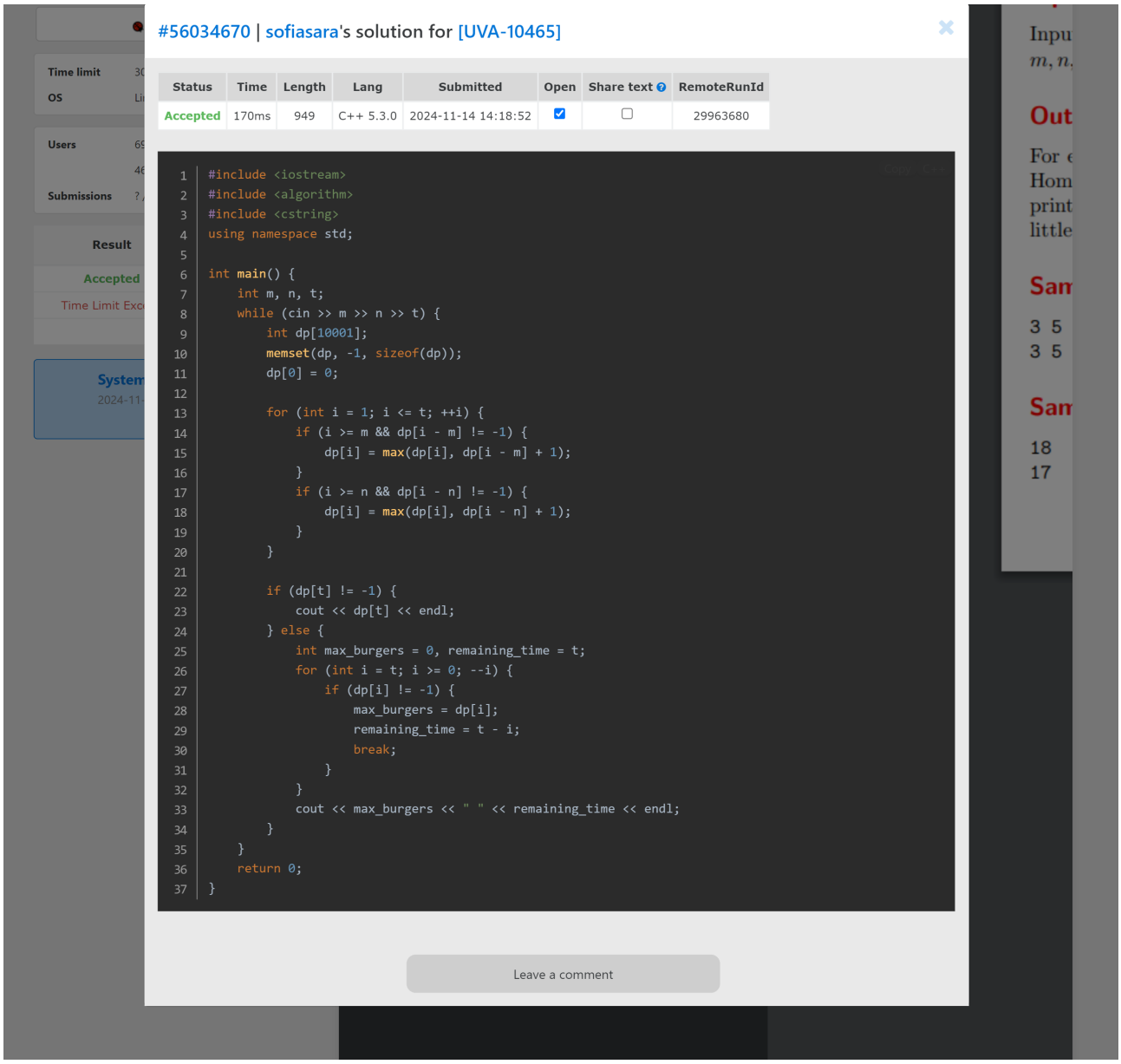
6. Referencias Bibliográficas

Bellman, R. (1952). On the theory of dynamic programming. *Proceedings of the national Academy of Sciences*, *38*(8), 716-719.

Demaine, E. (2021). *Dynamic Programming, Part 1: SRTBOT, Fib, DAGs, Bowling*. MIT OpenCourseWare. http://youtu.be/r4-cftqTcdI

7. Anexos

En las siguientes páginas anexamos el resultado de la plataforma http://vjudge.net al evaluar el código propuesto.



#56034670 | sofiasara's solution for [UVA-10465]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	170ms	949	C++ 5.3.0	2024-11-14 14:18:52	☑	☐	29963680

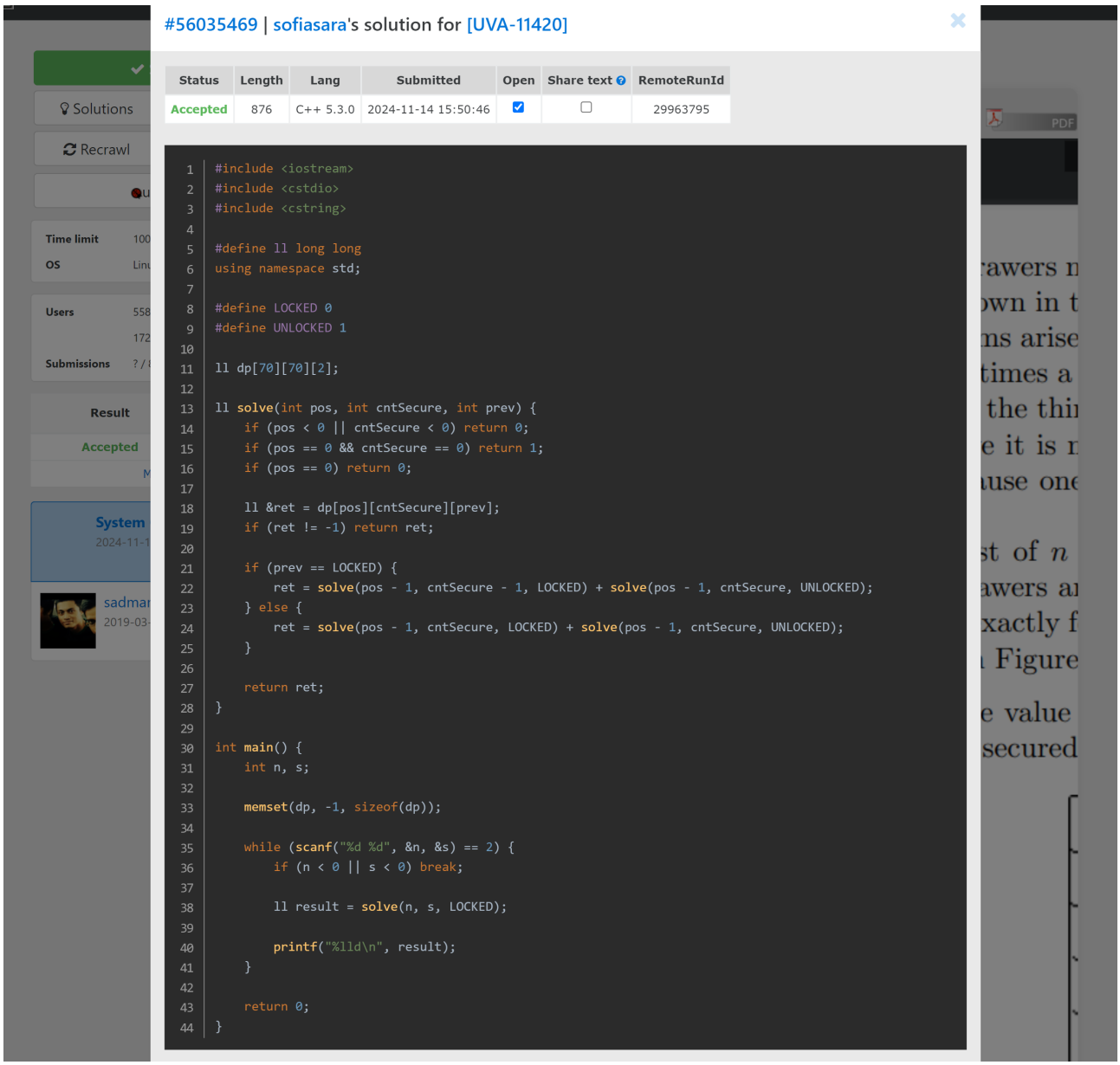
```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

int main() {
    int m, n, t;
    while (cin >> m >> n >> t) {
        int dp[10001];
        memset(dp, -1, sizeof(dp));
        dp[0] = 0;

        for (int i = 1; i <= t; ++i) {
            if (i >= m && dp[i - m] != -1) {
                dp[i] = max(dp[i], dp[i - m] + 1);
            }
            if (i >= n && dp[i - n] != -1) {
                dp[i] = max(dp[i], dp[i - n] + 1);
            }
        }

        if (dp[t] != -1) {
            cout << dp[t] << endl;
        } else {
            int max_burgers = 0, remaining_time = t;
            for (int i = t; i >= 0; --i) {
                if (dp[i] != -1) {
                    max_burgers = dp[i];
                    remaining_time = t - i;
                    break;
                }
            }
            cout << max_burgers << " " << remaining_time << endl;
        }
    }
    return 0;
}
```

Leave a comment



#56035469 | sofiasara's solution for [UVA-11420]

Status	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	876	C++ 5.3.0	2024-11-14 15:50:46	☑	☐	29963795

```cpp
#include <iostream>
#include <cstdio>
#include <cstring>

#define ll long long
using namespace std;

#define LOCKED 0
#define UNLOCKED 1

ll dp[70][70][2];

ll solve(int pos, int cntSecure, int prev) {
    if (pos < 0 || cntSecure < 0) return 0;
    if (pos == 0 && cntSecure == 0) return 1;
    if (pos == 0) return 0;

    ll &ret = dp[pos][cntSecure][prev];
    if (ret != -1) return ret;

    if (prev == LOCKED) {
        ret = solve(pos - 1, cntSecure - 1, LOCKED) + solve(pos - 1, cntSecure, UNLOCKED);
    } else {
        ret = solve(pos - 1, cntSecure, LOCKED) + solve(pos - 1, cntSecure, UNLOCKED);
    }

    return ret;
}

int main() {
    int n, s;

    memset(dp, -1, sizeof(dp));

    while (scanf("%d %d", &n, &s) == 2) {
        if (n < 0 || s < 0) break;

        ll result = solve(n, s, LOCKED);

        printf("%lld\n", result);
    }

    return 0;
}
```

#56037662 | sofiasara's solution for [UVA-10912]

Status	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	781	C++ 5.3.0	2024-11-14 21:22:04	☑	☐	29964362

Copy C++

```cpp
#include <stdio.h>
#include <string.h>

int DP[27][27][352];

void build() {
    memset(DP, 0, sizeof(DP));
    DP[0][0][0] = 1;

    for (int i = 1; i <= 26; i++) {
        for (int j = 0; j <= i; j++) {
            for (int k = 0; k < 352; k++) {
                DP[i][j][k] = DP[i-1][j][k];
                if (j > 0 && k >= i) {
                    DP[i][j][k] += DP[i-1][j-1][k-i];
                }
            }
        }
    }
}

int main() {
    build();
    int L, S, Case = 0;

    while (scanf("%d %d", &L, &S) == 2) {
        if (L == 0 && S == 0)
            break;

        printf("Case %d: ", ++Case);

        if (L > 26 || S > 351)
            puts("0");
        else
            printf("%d\n", DP[26][L][S]);
    }
    return 0;
}
```

Leave a comment