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: 20230489, Mamani Anahua Victor Narciso					
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

Este trabajo presenta una implementación y análisis de soluciones basadas en programación dinámica para resolver problemas algorítmicos complejos. Se emplea el método SRTBOT (Subproblemas, Relaciones recursivas, Topología, Bases, Original, Tiempo) como marco metodológico para el diseño de algoritmos recursivos y su optimización mediante técnicas de memoización. El estudio incluye la resolución de tres problemas distintos que demuestran la aplicación práctica de la programación dinámica: el problema de las Torres Gemelas, el problema de Sumas Grupales Divisibles y el problema de Homer Simpson. Para cada problema, se desarrolla una solución completa que incluye el análisis de subproblemas, la definición de relaciones recursivas, la representación topológica, la identificación de casos base, y la implementación tanto recursiva como optimizada con memoización. Los resultados experimentales demuestran la eficacia de la programación dinámica para reducir la complejidad computacional en problemas que exhiben subestructura óptima y subproblemas superpuestos.

1. Introducción

La programación dinámica representa uno de los paradigmas más poderosos en el diseño de algoritmos, permitiendo resolver problemas complejos mediante la descomposición en subproblemas más pequeños y el almacenamiento de resultados intermedios.

Este enfoque es particularmente valioso cuando los problemas exhiben subestructura óptima y subproblemas superpuestos, características que permiten optimizar significativamente el tiempo de ejecución en comparación con enfoques recursivos tradicionales.

En este trabajo, nos enfocamos en la aplicación del método SRTBOT para el diseño sistemático de soluciones basadas en programación dinámica. Este método proporciona un marco estructurado para abordar problemas complejos, facilitando la identificación de subproblemas, el establecimiento de relaciones recursivas y la optimización mediante memoización.

El resto del artículo está organizado de la siguiente manera. La Sección 2 presenta el marco teórico y conceptual necesario para comprender la programación dinámica y el método SRTBOT. La Sección 3 describe el diseño experimental y la metodología empleada. La Sección 4 muestra los resultados obtenidos para tres problemas específicos. Finalmente, la Sección 5 presenta las conclusiones y observaciones finales del trabajo.

2. Marco Teórico Conceptual

Esta sección presenta formalmente los conceptos teóricos necesarios para entender cabalmente el problema y los métodos de solución. El marco conceptual esta conformado por conceptos, definiciones y métodos que forman parte del conocimiento general del problema ampliamente difundido en libros y artículos. Por ejemplo, si estudiamos programación dinámica, debe introducir los conceptos necesarios.

Programación dinámica fue propuesta por Bellman (1952). Esta es una técnica algorítmica que resuelve problemas complejos dividiéndolos en subproblemas más simples la cual se fundamenta en el principio de optimalidad y la reutilización de soluciones a subproblemas previamente calculado

- **Subestructura Óptima:** La solución óptima del problema contiene las soluciones óptimas de sus subproblemas.
- **Subproblemas Superpuestos:** El mismo subproblema aparece repetidamente al resolver el problema mayor.

El nemotécnico SRTBOT propuesto por Demaine (2021) nos ayuda a diseñar algoritmos recursivos para resolver un problema. a continuación explicamos los seis conceptos de SRTBOT.

1. Subproblemas (Subproblems):

- Identificación de la estructura mínima que puede resolver parte del problema
- Definición clara de los parámetros que caracterizan cada subproblema
- Determinación del rango de valores para cada parámetro

2. Relaciones Recursivas (Recursive Relations):

- Establecimiento de las ecuaciones que relacionan un subproblema con otros más pequeños
- Definición de las operaciones necesarias para combinar las soluciones de los subproblemas
- Garantía de que cada subproblema depende únicamente de subproblemas más pequeños

3. Topología (Topology):

- Representación gráfica de las dependencias entre subproblemas
- Verificación de la ausencia de ciclos en las dependencias
- Identificación del orden de resolución de los subproblemas

4. Base (Base Cases):

- Definición de los casos más simples que pueden resolverse directamente
- Especificación de las condiciones de borde del problema
- Garantía de que todos los casos base están correctamente identificados

5. Original (Original Problem):

- Conexión entre el problema original y los subproblemas definidos
- Implementación de la solución recursiva inicial
- Aplicación de la técnica de memoización para optimizar el rendimiento

6. Tiempo (Time Complexity):

- Análisis del número total de subproblemas distintos
- Cálculo del tiempo necesario para resolver cada subproblema
- Determinación de la complejidad temporal y espacial total del algoritmo

2.3 Memoización La memoización es una técnica de optimización que almacena los resultados de operaciones costosas para reutilizarlos cuando se necesiten los mismos cálculos:

- Utiliza una estructura de datos (generalmente un arreglo o mapa) para almacenar resultados
- Verifica si un subproblema ya ha sido resuelto antes de calcularlo
- Reduce la complejidad temporal sacrificando espacio de memoria

2.4 Implementación Eficiente Para una implementación eficiente de programación dinámica, se deben considerar los siguientes aspectos: **Estructura de Datos:**

- Selección apropiada de estructuras para almacenar los resultados intermedios
- Consideración del compromiso entre tiempo y espacio
- Manejo eficiente de la memoria

Optimización:

- Eliminación de cálculos redundantes mediante memoización
- Reutilización efectiva de resultados previamente calculados
- Minimización del espacio de almacenamiento requerido

3. Diseño Experimental

Para realizar esta actividad de programación dinámica y el método SRTBOT, se implementó un proceso que consistió para los siguientes problemas:

1. **Problema 10066 – The Twin Towers**
   Este problema ilustra cómo encontrar la subsecuencia común más larga entre dos secuencias de azulejos. Representa un caso ideal para demostrar el poder de la programación dinámica en la optimización de comparaciones repetitivas. La solución se enfoca en optimizar la estructura de comparación entre segmentos de las dos secuencias, usando una matriz para almacenar resultados parciales y evitar cálculos redundantes.
   - **Selección**: El problema fue elegido aleatoriamente de una lista de problemas algorítmicos con características adecuadas para aplicar programación dinámica, dadas las similitudes entre subsecuencias.
   - **Proceso de Resolución**:
     - Se descompuso el problema en subproblemas que analizan las coincidencias entre azulejos en dos torres, permitiendo construir la solución de forma incremental.
     - Las relaciones recursivas determinan cómo extender una subsecuencia cuando los azulejos son iguales o cómo continuar la búsqueda en caso contrario. Esto permite explorar todas las posibles subsecuencias comunes.
     - La solución se organiza en una matriz bidimensional, donde cada posición contiene la longitud de la subsecuencia común hasta ese punto, garantizando eficiencia y evitando ciclos innecesarios.
     - Los casos base se definieron para secuencias vacías, simplificando la lógica y evitando comparaciones adicionales.
     - Se implementaron versiones en C++ con y sin memoización, almacenando resultados previos para eliminar redundancias y mejorar la eficiencia.
2. **Problema 10616 – Divisible Group Sums**
   Este problema demuestra cómo aplicar programación dinámica para contar combinaciones bajo restricciones específicas de divisibilidad. Es un ejemplo de cómo las relaciones recursivas y la estructura de almacenamiento permiten evaluar grandes cantidades de combinaciones de forma eficiente, cumpliendo con condiciones específicas.
   - **Selección**: Este problema fue seleccionado aleatoriamente entre una lista de problemas que requieren el uso de programación dinámica, ideal para practicar el conteo de combinaciones que satisfacen condiciones de divisibilidad.
   - **Proceso de Resolución**:
     - Se definieron subproblemas que representan combinaciones posibles de números en un conjunto, evaluando cuáles cumplen el criterio de divisibilidad. Cada subproblema explora la inclusión o exclusión de un número en la combinación actual.
     - Las relaciones recursivas dividen el problema en dos caminos: uno que suma el número actual a la combinación y otro que lo excluye, evaluando así todas las posibles combinaciones divisibles.
     - La solución se representa en una matriz tridimensional, que organiza los datos en tres dimensiones: el índice actual del número, el tamaño de la combinación, y la suma acumulada. Esto permite una navegación eficiente y evita la repetición de cálculos.
     - Los casos base establecen que una combinación vacía es válida si su suma es divisible, lo que facilita el conteo y asegura la correcta aplicación de la condición de divisibilidad.
     - Las versiones en C++ con y sin memoización ayudan a manejar un gran número de combinaciones, permitiendo al algoritmo reutilizar resultados previos y mejorar la eficiencia en grandes conjuntos de datos.

3. **Problema 10465 – Homer Simpson**
   En este problema, la programación dinámica se utiliza para maximizar la cantidad de hamburguesas que Homer puede comer en un tiempo limitado, bajo múltiples restricciones. Es un excelente ejemplo de cómo utilizar estructuras de datos como `pair` para manejar múltiples valores de retorno, permitiendo una solución eficiente bajo condiciones específicas.
   - **Selección**: Este problema fue elegido aleatoriamente de una lista de problemas que requieren maximización bajo restricciones, demostrando cómo la programación dinámica permite optimizar soluciones complejas.
   - **Proceso de Resolución**:
     - Se descompuso el problema en subproblemas que representan combinaciones de tiempos con diferentes cantidades de hamburguesas. Cada subproblema evalúa cómo usar el tiempo disponible de manera que se maximice la cantidad de hamburguesas.
     - Las relaciones recursivas modelan el problema al reducir el tiempo total en función de las opciones de hamburguesas, sumando la cantidad resultante de hamburguesas en cada caso. Se usó la estructura `pair<int, int>` para manejar tanto la cantidad máxima de hamburguesas alcanzables como el tiempo restante, simplificando el seguimiento de ambas variables en la optimización.
     - La solución se organizó en un vector donde cada índice representa el tiempo restante al evaluar las combinaciones, respetando las restricciones temporales y evitando recalcular combinaciones ya exploradas.
     - Los casos base definen situaciones en las que el tiempo restante es insuficiente para adquirir una hamburguesa, devolviendo un valor de cero. Esto simplifica las decisiones cuando el tiempo es limitado.
     - Implementado en C++ con y sin memoización, el algoritmo explora eficientemente combinaciones complejas, optimizando el uso del tiempo disponible y maximizando el número de hamburguesas consumidas sin sobrecargar el procesamiento.

3.1. Objetivos

Los objetivos de este trabajo son:

- Reforzar los conocimientos del método de programación dinámica.
- Aplicar el método de programación dinámica para resolver algunos problemas propuestos.

3.2. Actividades

El estudiante deberá realizar las siguientes acciones.

a) Crear un usuario en http://vjudge.net e indicar en este paso el nombre de usuario utilizado , (Usuario: VictorMA).
b) Seleccionar aleatoriamente tres problemas de la lista disponible en http://bit.ly/3UxdCVL. Note que debe loguearse en la plataforma para poder ver los problemas.
c) Deberá diseñar una solución utilizando la técnica SRTBOT para cada problema.
d) Deberá mostrar el pseudocódigo recursivo resultante sin y con memoización.
e) Deberá incluir el código en C++ que resulta del modelo SRTBOT.
f) Deberá anexar el PDF del código aceptado por la plataforma http://vjudge.net al final del artículo.

4. Resultados

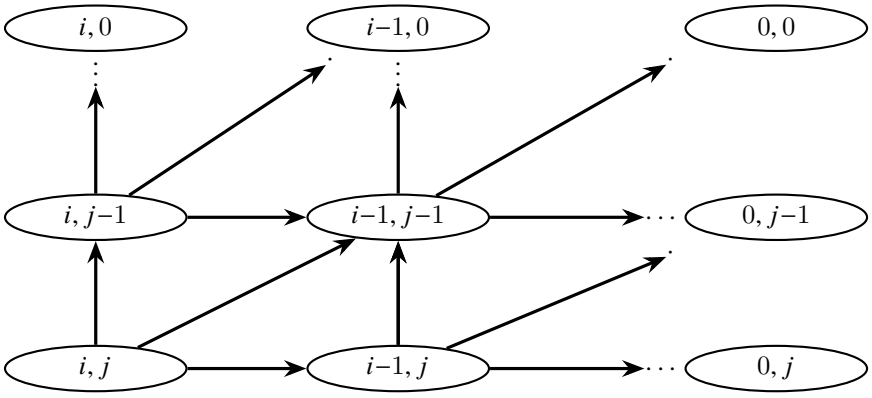
Muestra y describe los resultados del presente trabajo. A continuación se muestran como ejemplo los resultados para tres problemas desarrollados en aulas.

4.1. Problema 10066 – The Twin Towers

Subproblema: Encuentre $TG(i, j)$ para los azulejos $0 \leq i \leq N_1, 0 \leq j \leq N_2$.
Relaciones Recursivas:

$$TG(i,J) = \begin{cases} 1 + TG(i-1, j-1), & N_1[i-1] == N_2[j-2] \\ \text{máx}\ (\ TG(i-1, j),\ TG(i, j-1)\ ), & N_1[i-1]\ !\ = N_2[j-2] \end{cases}$$

Topología:



Básico: $TG(0, j) = 0\ TG(i, 0) = 0$
Original:

Algorithm TG(i, j) // sin memoización

Input: objeto i de N_1, objeto j de N_2
Output: máximo número de azulejos

```
1: if i = 0 or j = 0 then
2:     return 0
3: else
4:     if N1[i − 1] equals N2[j − 1] then
5:         return 1 + TG(i−1, j−1)
6:     else
7:         return max(TG(i−1, j),TG(i, j−1))
```

Algorithm TGM(i, w) // con memoización

Input: objeto i de N_1, objeto j de N_2
Output: máximo número de azulejos

```
1: if i = 0 or j = 0 then
2:     return 0
3: else
4:     if M[i, j] is undefined then
5:         if N1[i − 1] equals N2[j − 1] then
6:             M[i, j] = 1 + TGM(i−1, j−1)
7:         else
8:             M[i, w] = max(TGM(i−1, j),
9:                                 TGM(i, j−1))
10:    return M[i, w]
```

Tiempo: TGM$(i, j) \in O(i, j)$
Código:

```cpp
#include <bits/stdc++.h>
using namespace std;

int m, n;
int N2[100];
int N1[100];
int M[105][105];

int TG(int i, int j) {
    if (i == 0 || j == 0)
        return 0;
    else{
        if(M[i][j] == -1){
            if (N1[i-1] == N2[j-1]) {
                M[i][j] = 1 + TG(i-1, j-1);
            } else {
                M[i][j] = max(TG(i-1, j), TG(i, j-1));
            }
        }
    }
    return M[i][j];
}
```

```cpp
int main() {
    int caso = 1;
    while (cin >> m >> n) {
        if (m == 0 && n == 0)
            break;
        memset ( M, -1, sizeof ( M ) );
        for (int i = 0; i < m; i++) {
            cin >> N1[i];
        }
        for (int j = 0; j < n; j++) {
            cin >> N2[j];
        }
        int result = 0;
        result += TG(m, n);
        cout << "Twin Towers #" << caso << endl;
        cout << "Number of Tiles : " << result << endl;
        cout << "\n";
        caso++;
    }
    return 0;
}
```

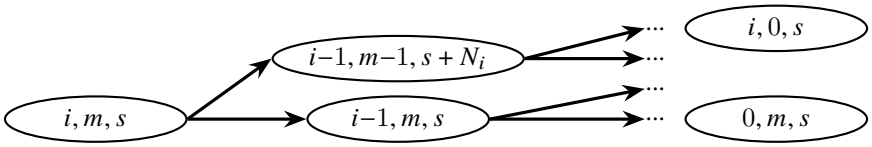
4.2. Problema 10616 – Divisible Group Sums

Subproblema: Encuentre $DGS(i, m, s)$ para los números $0 \leq i \leq 200$, la cantidad de combinaciones $0 \leq m \leq 10$ y $0 \leq s \leq$ suma máxima de una combinacion de m con i números

Relaciones Recursivas:

$$DGS(i, m, s) = \Big\{ DGS(i-1, m-1, s+N_i) + DGS(i-1, m, s) \;), \quad i > 0, m > 0$$

Topología:



Básico:

$DGS(i, 0, s) = 1$,si s es divisble por d
$DGS(i, 0, s) = 0$,si s no es divisble por d
$DGS(0, m, s) = 0$

Original:

Algorithm DGS(i, m, s) // sin memoización
Input: número i, combinaciones m, suma s
Output: cuántas maneras

```
1: if m = 0 then
2:     if s % d = 0 then
3:         return 1
4:     else
5:         return 0
6: if i = 0 then
7:     return 0
8: else
9:     return DGS(i−1, m−1, s+N_i) + DGS(i−1, m, s)
```

Algorithm DGSM(i, m, s) // sin memoización
Input: número i, combinaciones m, suma s
Output: cuántas maneras

```
1: if m = 0 then
2:     if s % d = 0 then
3:         return 1
4:     else
5:         return 0
6: if i = 0 then
7:     return 0
8: else
9:     s_p = (s %d + d) %d
10:    if M[i, m, s_p] is undefined then
11:        M[i, m, s_p] = DGSM(i−1, m−1, s+N_i) + DGSM(i−1, m, s)
12:    return M[i, m, s_p]
```

Tiempo: DGSM$(i, m, s) \in O(i, m, s)$
Código:

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, d, q, set_num;
int numeros[201];
int M[201][201][20];

int dgs(int i, int m, int sum){
    if (m == 0) {
        if(sum % d == 0){
          return 1;
        }else{
          return 0;
        }
    }
    int mod_sum = (sum % d + d) % d;
    if (i == 0) return 0;
    else{
      if(M[i][m][mod_sum] == -1){
        M[i][m][mod_sum] = dgs( i - 1, m - 1,
sum + numeros[i - 1]) + dgs( i - 1, m, sum);
      }
      return M[i][m][mod_sum];
    }
}
```

```cpp
int main() {
    set_num = 1;
    while (true) {
        cin >> n >> q;

        if (n == 0 && q == 0) break;

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

        cout << "SET␣" << set_num << ":\n";
        set_num++;

        for (int t = 0; t < q; t++) {
            cin >> d; cin >> m;
            memset ( M, -1, sizeof ( M ) );
            int result = dgs(n, m, 0);
            cout<< "QUERY␣" << (t+1) << ":␣" << result << "\n";
        }
    }
    return 0;
}
```

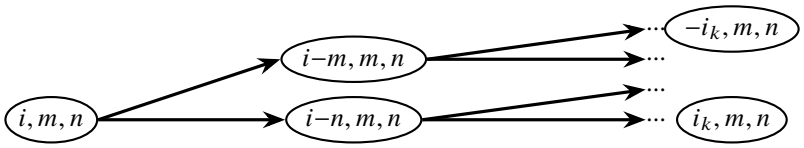
4.3. Problema 10465 – Homer Simpson

Subproblema: Encuentre $HS(i,m,n)$ para el tiempo $0 \leq i \leq 10000$, $0 \leq m \leq 10000$ y $0 \leq n \leq 10000$

Relaciones Recursivas:

$$HS(i) = \begin{cases} HS(i-m), \\ \qquad\qquad i \geq \min(m,n) \\ HS(i-n), \end{cases}$$

Topología:



Básico:

$HS(i,m,n) = \{0,i\}$, si $i < 0$

$HS(i,m,n) = \{0, Infinite\}$, si $0 \leq i < \min(m,n)$

Original:

Algorithm HS(t) // sin memoización

Input: tiempo total i, hamburguesa m y n
Output: máximo de hamburguesas sin tomar

```
1: if i < 0 then
2:     return 0, Infinite
3: if i < mín(m, n) then
4:     return 0, i
5: else
6:     H_m = hs(i − m, m, n)
7:     H_m.first++
8:     H_n = hs(i − n, m, n)
9:     H_n.first++
10:    if H_m.second != H_n.second then
11:        return H_m.second < H_n.second ? H_m.first : H_n.first, H_m.second < H_n.second ? H_m.second : H_n.second
12:    else
13:        return H_m.first > H_n.first ? H_m.first : H_n.first, H_m.second
```

Algorithm HSM(i, m, n) // con memoización

Input: tiempo total i, hamburguesa m y n
Output: máximo de hamburguesas sin tomar

```
1: if i < 0 then
2:     return 0, Infinite
3: if i < mín(m, n) then
4:     return 0, i
5: else
6:     if M[i, m, n] is undefined then
7:         Hm = hs(i − m, m, n)
8:         Hm.first++
9:         Hn = hs(i − n, m, n)
10:        Hn.first++
11:        if m.second != Hn.second then
12:            M[i] = H_m.second < H_n.second ? H_m.first : H_n.first, H_m.second < H_n.second ? H_m.second : H_n.second
13:        else
14:            M[i] = H_m.first > H_n.first ? H_m.first : H_n.first, H_m.second
15:    return M[t]
```

Tiempo: HSM$(i, m, n) \in O(i, m, n)$
Código:

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

pair<int, int> M[10001];

pair<int,int> HSM(int i,int m, int n) {
    if(i < 0) return {0, 1e7};
    if(i < min(m, n)) return {0, i};
    else{
      if(M[i].second == -1) {
        pair<int,int> Hm = HSM(i - m, m, n);
        Hm.first++;
        pair<int,int> Hn = HSM(i - n,m, n);
        Hn.first++;

        if(Hm.second != Hn.second) {
            M[i] = {Hm.second < Hn.second ?
Hm.first : Hn.first,
Hm.second < Hn.second ? Hm.second : Hn.second};
        } else {
            M[i] = {Hm.first > Hn.first ?
Hm.first : Hn.first, Hm.second};
        }
      }
      return M[i];
    }
}
```

```cpp
int main() {
    int m, n, t;
    while(cin >> m >> n >> t) {
        for (int i = 0; i <= t; i++) {
            M[i] = {0, -1};
        }

        pair<int,int> resultado = HSM(t,m,n);
        if(resultado.second == 0) {
            cout << resultado.first << endl;
        } else {
            cout << resultado.first << "␣"
<< resultado.second << endl;
        }
    }
    return 0;
}
```

5. Conclusiones

En este reporte hemos revisado el método SRTBOT para diseñar soluciones recursivas. Hemos aplicado programación dinámica usando memoización junto al método SRTBOT. Finalmente hemos resuelto satisfactoriamente tres problemas con la técnica de programación dinámica.
La implementación y análisis de soluciones basadas en programación dinámica utilizando el método SRTBOT ha demostrado ser un enfoque efectivo para resolver problemas algorítmicos complejos. Las principales conclusiones de este trabajo son:

1. La aplicación sistemática del método SRTBOT facilita significativamente el proceso de diseño de soluciones recursivas y su posterior optimización mediante memoización.
2. La programación dinámica, cuando se aplica correctamente, permite reducir la complejidad temporal de problemas que inicialmente parecen requerir soluciones exponenciales a soluciones polinomiales.
3. La implementación de la memoización resulta crucial para evitar recálculos innecesarios, mejorando sustancialmente el rendimiento de los algoritmos recursivos.
4. Los tres problemas resueltos demuestran la versatilidad de la programación dinámica para abordar diferentes tipos de desafíos algorítmicos, desde problemas de optimización hasta problemas de conteo.
5. La plataforma vjudge.net proporciona un entorno efectivo para la validación y verificación de las soluciones implementadas, permitiendo probar los algoritmos con diversos casos de prueba.

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.
Estas impresiones fueron hechas con Google Chrome con la impresora destino “Guardar como PDF” y con un tamaño de página “A3” para que entre en una sola.

All

Mine

Follov

#55937849 | VictorMA's solution for [UVA-10066]

Status	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	932	C++11 5.3.0	2024-11-11 23:48:15	☑	☐	29955877

```cpp
#include <bits/stdc++.h>
using namespace std;

int m, n;
int N2[101];
int N1[101];
int M[101][101];

int TGM(int i, int j) {
    if (i == 0 || j == 0)
        return 0;
    else{
        if(M[i][j] == -1){
            if (N1[i-1] == N2[j-1]) {
                M[i][j] = 1 + TGM(i-1, j-1);
            } else {
                M[i][j] = max(TGM(i-1, j), TGM(i, j-1));
            }
        }
    }
    return M[i][j];
}

int main() {
    int caso = 1;
    while (cin >> m >> n) {
        if (m == 0 && n == 0)
            break;
        memset ( M, -1, sizeof ( M ) );
        for (int i = 0; i < m; i++) {
            cin >> N1[i];
        }
        for (int j = 0; j < n; j++) {
            cin >> N2[j];
        }
        int result = 0;
        result += TGM(m, n);
        cout << "Twin Towers #" << caso << endl;
        cout << "Number of Tiles : " << result << endl;
        cout << "\n";
        caso++;
    }
    return 0;
}
```

Leave a comment

Loading comments...

All
Mine
Follov

#55937786 | VictorMA's solution for [UVA-10616]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	20ms	1044	C++11 5.3.0	2024-11-11 23:45:02	☑	☐	29955870

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, d, q, set_num;
int numeros[201];
int M[201][201][20];

int DGSM(int i, int m, int sum){
    if (m == 0) {
        if(sum % d == 0){
          return 1;
        }else{
          return 0;
        }
    }
    if (i == 0) return 0;
    else{
      int mod_sum = (sum % d + d) % d;
      if(M[i][m][mod_sum] == -1){
        M[i][m][mod_sum] = DGSM( i - 1, m - 1, sum + numeros[i - 1]) + DGSM( i - 1, m, sum);
      }
      return M[i][m][mod_sum];
    }
}

int main() {
    set_num = 1;
    while (true) {
        cin >> n >> q;

        if (n == 0 && q == 0) break;

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

        cout << "SET " << set_num << ":\n";
        set_num++;

        for (int t = 0; t < q; t++) {
            cin >> d; cin >> m;
            memset ( M, -1, sizeof ( M ) );
            int result = DGSM(n, m, 0);
            cout<< "QUERY " << (t+1) << ": " << result << "\n";
        }
    }
    return 0;
}
```

Leave a comment

Loading comments...

All
Mine
Follov

#55932692 | VictorMA's solution for [UVA-10465]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	440ms	1039	C++11 5.3.0	2024-11-11 20:59:18	☑	☐	29955492

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

pair<int, int> M[10001];

pair<int,int> HSM(int i,int m, int n) {
    if(i < 0) return {0, 1e7};
    if(i < min(m, n)) return {0, i};
    else{
      if(M[i].second == -1) {
        pair<int,int> Hm = HSM(i - m, m, n);
        Hm.first++;
        pair<int,int> Hn = HSM(i - n,m, n);
        Hn.first++;

        if(Hm.second != Hn.second) {
            M[i] = {Hm.second < Hn.second ? Hm.first : Hn.first, Hm.second < Hn.second ? Hm.second :
Hn.second};
        } else {
            M[i] = {Hm.first > Hn.first ? Hm.first : Hn.first, Hm.second};
        }
      }
      return M[i];
    }
}

int main() {
    int m, n, t;
    while(cin >> m >> n >> t) {
        for (int i = 0; i <= t; i++) {
            M[i] = {0, -1};
        }

        pair<int,int> resultado = HSM(t,m,n);
        if(resultado.second == 0) {
            cout << resultado.first << endl;
        } else {
            cout << resultado.first << " " << resultado.second << endl;
        }
    }
    return 0;
}
```

Leave a comment