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: 20233483, QUISPE MARCA EDYSSON DARWIN					
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

La programación dinámica es una técnica de optimización que se utiliza para resolver problemas complejos mediante la descomposición en subproblemas más simples. Al almacenar las soluciones de estos subproblemas, se evita el recalculo redundante, lo que mejora la eficiencia del algoritmo. Este artículo explora cómo la programación dinámica se aplica a problemas clásicos, destacando la importancia de identificar subproblemas y definir relaciones recursivas. Se introduce el esquema nemotécnico SRTBOT, que ofrece un enfoque sistemático para diseñar algoritmos eficientes. El trabajo incluye una revisión teórica de la técnica, ejemplos de su aplicación, y experimentos que demuestran su efectividad en la resolución de problemas de optimización.

1. Introducción

La programación dinámica es una técnica fundamental en el campo de la optimización que se utiliza para resolver problemas complejos que presentan subestructura óptima y subproblemas que se superponen. Esta técnica se basa en la idea de descomponer un problema en subproblemas más simples, resolver estos subproblemas de manera recursiva y almacenar sus soluciones para evitar recalcular los mismos resultados, mejorando así la eficiencia del algoritmo.

Este trabajo tiene como objetivo explorar cómo la programación dinámica puede aplicarse a una serie de problemas clásicos, abordando la importancia de la descomposición en subproblemas y las relaciones recursivas que permiten la resolución eficiente de estos problemas. Además, se propone el uso del esquema nemotécnico SRTBOT, que ayuda a diseñar algoritmos eficientes a partir de una serie de pasos sistemáticos para abordar problemas típicos de programación dinámica. Este esquema incluye la identificación de subproblemas, la definición de relaciones recursivas, la topología de las dependencias, la creación de bases, y la evaluación del tiempo de ejecución de los algoritmos.

El artículo está organizado de la siguiente manera: La Sección 2 describe los conceptos teóricos y las técnicas esenciales utilizadas en programación dinámica. La Sección 3 se enfoca en la aplicación de estos conceptos en la resolución de problemas específicos. La Sección 4 presenta los experimentos realizados para analizar el rendimiento de los algoritmos implementados. Finalmente, la Sección 5 ofrece las conclusiones alcanzadas en este trabajo.

2. Marco Teórico Conceptual

En esta sección se definen formalmente los conceptos teóricos necesarios para entender los problemas abordados y las técnicas de solución. Programación dinámica es una técnica propuesta por **bellman1952theory<empty citation>** que permite resolver problemas optimizando subproblemas que se solapan. El esquema nemotécnico SRTBOT, propuesto por **demaine2021<empty citation>**, facilita el diseño de algoritmos recursivos en problemas de programación dinámica. Este esquema consta de los siguientes elementos:

2.1. Subproblemas

El primer paso en programación dinámica es descomponer el problema original en subproblemas más simples. Cada subproblema debe representar una parte del problema que puede ser resuelta independientemente. Por ejemplo, en el problema de la mochila, cada subproblema considera la posibilidad de incluir o excluir un ítem específico en función de su peso y valor.

2.2. Relaciones Recursivas

El siguiente paso es definir las relaciones recursivas, es decir, expresar el valor de un subproblema en términos de otros subproblemas. Estas relaciones son fundamentales para construir la solución general. Por ejemplo, en problemas de secuencias, una relación recursiva puede ser la longitud de la subsecuencia más larga común entre dos cadenas.

2.3. Topología

La topología en programación dinámica describe la estructura de la relación entre los subproblemas. La forma más común de visualizar esta estructura es mediante un gráfico en árbol o en grafo dirigido acíclico (DAG), donde cada nodo representa un subproblema y las aristas representan dependencias recursivas entre ellos.

2.4. Bases

Definir los casos base es esencial para detener la recursión y proporcionar un punto de inicio para construir la solución final. Un caso base es una instancia simple del problema cuya solución es conocida y no requiere recursión. En el problema de Fibonacci, por ejemplo, los casos base son $F(0) = 0$ y $F(1) = 1$.

2.5. Original

El objetivo final es resolver el problema original utilizando las relaciones recursivas y los subproblemas definidos anteriormente. Esto implica combinar los resultados de los subproblemas para construir la solución al problema en su totalidad. Para optimizar este proceso, se recomienda utilizar **memoización**, que almacena los resultados de subproblemas previamente resueltos para evitar cálculos redundantes.

2.6. Tiempo

El último paso es analizar el tiempo de ejecución del algoritmo. En programación dinámica, el tiempo de ejecución suele depender del número de subproblemas y la complejidad de resolver cada uno. El uso de memoización o tabulación puede reducir la complejidad de exponencial a polinomial en muchos problemas de programación dinámica.

3. Diseño Experimental

En esta sección se describe el proceso seguido para seleccionar y resolver los problemas de programación dinámica propuestos. También se explica el método utilizado para comparar los resultados obtenidos, tanto de manera estadística como gráfica. Este análisis permite evaluar la efectividad de los algoritmos implementados y observar cómo se comportan bajo diferentes tamaños de entrada.

3.1. Actividades Realizadas

Para alcanzar estos objetivos, se realizaron las siguientes actividades:

- **Diseño de Algoritmos**: Se aplicó el esquema SRTBOT en cada problema. Se formularon los subproblemas específicos y se diseñaron relaciones recursivas optimizadas. Luego, los algoritmos se implementaron con técnicas de memoización para evitar el cálculo redundante de subproblemas.
- **Implementación y Pruebas**: Los algoritmos fueron implementados en un entorno de programación y se realizaron pruebas con diferentes tamaños de entrada. Se midió el tiempo de ejecución de cada algoritmo para analizar la eficiencia y se comparó la implementación con y sin memoización.
- **Análisis de Resultados**: Los resultados experimentales se analizaron gráficamente para observar el rendimiento de los algoritmos en función del tamaño de entrada. Se generaron gráficos que muestran la relación entre el tiempo de ejecución y el tamaño del problema, y se realizó un análisis estadístico para cuantificar las mejoras obtenidas.

3.2. Logros Alcanzados

Se lograron varios avances significativos durante el desarrollo del trabajo:

- **Optimización de Algoritmos**: Los algoritmos desarrollados con técnicas de memoización mostraron una reducción notable en el tiempo de ejecución en comparación con sus versiones puramente recursivas, confirmando la efectividad de la programación dinámica en la mejora de eficiencia.
- **Documentación del Proceso SRTBOT**: Cada uno de los pasos de SRTBOT se aplicó a fondo, documentando cada etapa en el diseño del algoritmo. Esto facilitó un enfoque sistemático en la resolución de problemas complejos.
- **Resultados Visuales y Estadísticos**: Se generaron gráficos de rendimiento y análisis estadísticos que evidenciaron la eficiencia de las soluciones implementadas. Estos resultados facilitaron una comprensión visual del impacto de la memoización.

3.3. 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.

3.4. 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. Nombre de usuario : Museum
2. 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.
3. Deberá diseñar una solución utilizando la técnica SRTBOT para cada problema.
4. Deberá mostrar el pseudocódigo recursivo resultante sin y con memoización.
5. Deberá incluir el código en C++ que resulta del modelo SRTBOT.
6. 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 384 – Slurpys

Subproblema: Verificar si una subcadena es un *Slump* o un *Slimp* siguiendo las reglas definidas para ambas estructuras. Un *Slurpy* es una cadena que cumple con las condiciones de ser un *Slimp* seguido de un *Slump*.

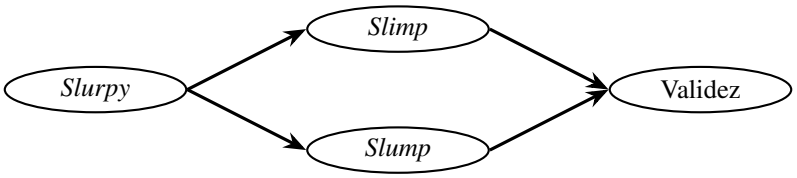
Relaciones Recursivas: Las reglas para formar un *Slimp* son:

- Debe empezar con la letra A.
- Puede ser de la forma AH.
- Puede ser de la forma AB + Slimp + C.
- Puede ser de la forma A + Slump + C.

Las reglas para formar un *Slump* son:

- Debe empezar con D o E.
- Debe contener al menos una F.
- Puede terminar en G o continuar con otro *Slump*.

Topología:



Básico: Una cadena vacía o una que no cumple las reglas iniciales para *Slimp* o *Slump* es inválida.

Original: Implementación recursiva para verificar si una cadena es un *Slimp* o un *Slump*.

Algorithm *isSlimp*(s, pos) // sin memoización
Input: cadena s, posición pos
Output: verdadero si es *Slimp*, falso si no

```
1: if s[pos] = A then
2:     Incrementar pos
3:     if s[pos] = H then
4:         return verdadero
5:     else if s[pos] = B y esSlimp(s, pos) y
       s[pos] = C then
6:         return verdadero
7:     else if esSlump(s, pos) y s[pos] = C
       then
8:         return verdadero
9: return falso
```

Algorithm *isSlump*(s, pos) // sin memoización
Input: cadena s, posición pos
Output: verdadero si es *Slump*, falso si no

```
1: if s[pos] = D o s[pos] = E then
2:     Incrementar pos
3:     if s[pos] = F then
4:         while s[pos] = F do
5:             Incrementar pos
6:         if s[pos] = G then
7:             return verdadero
8:         else if esSlump(s, pos) then
9:             return verdadero
10: return falso
```

Optimización con Memoización: Utilizando una tabla de memoria (*memo*) para almacenar los resultados de las posiciones ya calculadas, se mejora la eficiencia evitando recalcular subproblemas ya resueltos.

Algorithm *isSlimpMemo*(s, pos, memo) // con memoización
Input: cadena s, posición pos, *memo*
Output: verdadero si es *Slimp*, falso si no

```
1:  if pos en memo then
2:      return memo[pos]
3:  if s[pos] = A then
4:      Incrementar pos
5:      if s[pos] = H then
6:          return memo[pos] = verdadero
7:      else if s[pos] = B y esSlimpMemo(s, pos, memo) y s[pos] = C then
8:          return memo[pos] = verdadero
9:      else if esSlumpMemo(s, pos, memo) y s[pos] = C then
10:         return memo[pos] = verdadero
11: return memo[pos] = falso
```

Algorithm *isSlumpMemo*(s, pos, memo) // con memoización
Input: cadena s, posición pos, *memo*
Output: verdadero si es *Slump*, falso si no

```
1:  if pos en memo then
2:      return memo[pos]
3:  if s[pos] = D o s[pos] = E then
4:      Incrementar pos
5:      if s[pos] = F then
6:          while s[pos] = F do
7:              Incrementar pos
8:          if s[pos] = G then
9:              return memo[pos] = verdadero
10:         else if esSlumpMemo(s, pos, memo) then
11:             return memo[pos] = verdadero
12: return memo[pos] = falso
```

Tiempo: La complejidad de la versión con memoización es $O(n)$, donde n es la longitud de la cadena, dado que cada subproblema se calcula una sola vez.

Código:

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

unordered_map<int, bool> slumpMemo;
unordered_map<int, bool> slimpMemo;

bool isSlump(const string &s, int &pos) {
    if (slumpMemo.count(pos)) return slumpMemo[pos];
    int startPos = pos;

    if (s[pos] != 'D' && s[pos] != 'E') return slumpMemo[startPos] = false;
    pos++;

    if (s[pos] != 'F') return slumpMemo[startPos] = false;
    while (s[pos] == 'F') pos++;

    if (s[pos] == 'G') {
        pos++;
        return slumpMemo[startPos] = true;
    } else {
        bool result = isSlump(s, pos);
        return slumpMemo[startPos] = result;
    }
}

bool isSlimp(const string &s, int &pos) {
    if (slimpMemo.count(pos)) return slimpMemo[pos];
    int startPos = pos;

    if (s[pos] != 'A') return slimpMemo[startPos] = false;
    pos++;

    if (s[pos] == 'H') {
        pos++;
        return slimpMemo[startPos] = true;
    }

    if (s[pos] == 'B') {
        pos++;
        if (!isSlimp(s, pos)) return slimpMemo[startPos] = false;
        if (s[pos] == 'C') {
            pos++;
            return slimpMemo[startPos] = true;
        }
    } else if (isSlump(s, pos)) {
        if (s[pos] == 'C') {
            pos++;
            return slimpMemo[startPos] = true;
        }
    }
    return slimpMemo[startPos] = false;
}

bool isSlurpy(const string &s) {
    int pos = 0;
    slimpMemo.clear();
    slumpMemo.clear();
    if (!isSlimp(s, pos)) return false;
    if (!isSlump(s, pos)) return false;
    return pos == s.size();
}

int main() {
    int N;
    cin >> N;
    cout << "SLURPYS OUTPUT" << endl;
    for (int i = 0; i < N; i++) {
        string s;
        cin >> s;
        if (isSlurpy(s)) {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }

    cout << "END OF OUTPUT" << endl;
    return 0;
}
```

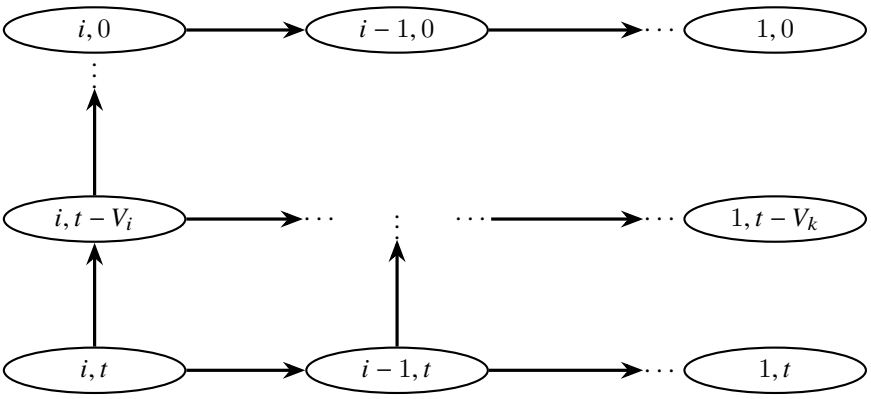
4.2. Problema 10313 – Pay the price

Subproblema: Dado un monto n y un límite de monedas L, encontrar el número de formas diferentes de representar n usando monedas de valores 1 hasta L.

Relaciones Recursivas:

$$C(amount, maxCoin) = \begin{cases} 1, & \text{si } n = 0 \\ \text{si } n < 0 \text{ o } maxCoin = 0 \; C(n, maxCoin - 1) + C(n - maxCoin, maxCoin) \end{cases}$$

Topología: La estructura del grafo de dependencias se mantiene igual que en la plantilla proporcionada, donde cada estado (i,t) depende de: Estado (i-1,t): no usar la moneda i y Estado (i,t-Vi): usar la



moneda i

Básico: $C(1, w) = 1$

Original:

Algorithm C(i, t) // sin memoización
Input: cantidad i, total t
Output: cuántas formas

```
1: if i = 1 then
2:     return 1
3: else
4:     if t < V[i] then
5:         return C(i − 1, t)
6:     else
7:         return C(i − 1, t)+ C(i, t − V[i])
```

Algorithm CM(i, t) // con memoización
Input: cantidad i, total t
Output: cuántas formas

```
1: if i = 1 then
2:     return 1
3: else
4:     if M[i, t] es indefinido then
5:         if t < V[i] then
6:             M[i, t] = CM(i − 1, t)
7:         else
8:             M[i, t] = CM(i − 1, t)+ CM(i, t − V[i])
9:     return M[i, t]
```

Tiempo: Sin memoización: O(2^n), donde n es el monto Con memoización: O(n*m), donde n es el monto y m es el máximo valor de moneda permitido

Código:

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <unordered_map>
using namespace std;

static const int MAX_VALUE = 300;
typedef unsigned long long ull_t;

unordered_map<string, ull_t> memo;

ull_t countWays(int amount, int maxCoin) {
    if (amount == 0) return 1;
    if (amount < 0 || maxCoin == 0) return 0;

    string key = to_string(amount) + "," + to_string(maxCoin);

    if (memo.find(key) != memo.end()) return memo[key];

    memo[key] = countWays(amount, maxCoin - 1) + countWays(amount - maxCoin, maxCoin);

    return memo[key];
}

int main() {
    string s;
    while (getline(cin, s)) {
        vector<int> N;
        stringstream ss(s);
        int input;
        while (ss >> input) N.push_back(input);

        int L1, L2;
        switch (N.size()) {
            case 1:
                L1 = 0;
                L2 = N[0];
                break;
            case 2:
                L1 = 0;
                L2 = N[1];
                break;
            default:
                L1 = N[1];
                L2 = N[2];
        }

        if (N[0] == 0) {
            cout << "1" << endl;
        } else {
            ull_t result = countWays(N[0], min(MAX_VALUE, L2)) -
                           (L1 > 0 ? countWays(N[0], L1 - 1) : 0);
            cout << result << endl;
        }
    }
    return 0;
}
```

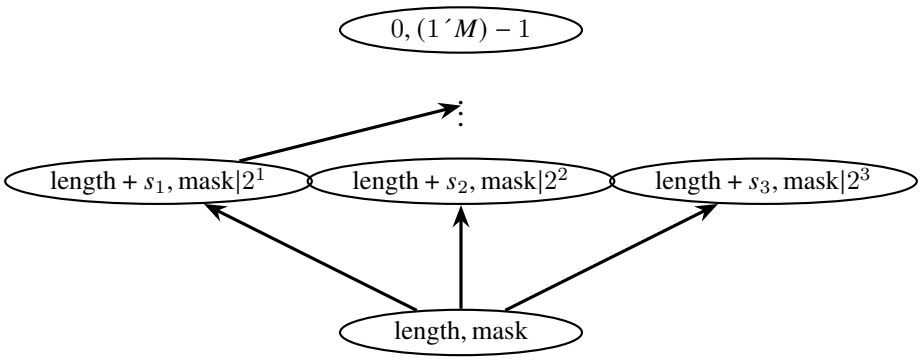
4.3. Problema 10364 – Square

Subproblema: Encuentre $C(t)$ para el conjunto de palos dados.

Relaciones Recursivas:

$$C(l,m) = \begin{cases} 0, & l > L\ 1, \\ \text{m} = 2^M - 1 \ \text{ y } \ \text{l} = \text{L} \ \bigvee_{i=0}^{M-1} C(l + s_i, m|2^i) \end{cases}$$

Topología:



Base:

$$\text{canFormSquare}(\text{length}, \text{mask}) = \begin{cases} 0, & \text{si length} > \text{sum}/4 \\ 1, & \text{si mask} = 2^M - 1 \text{ y length} = 0 \\ -1, & \text{si no está calculado} \end{cases}$$

Original:

Algorithm canFormSquare(length, mask) // sin memoización
Require: longitud actual length, máscara de bits mask
Ensure: si es posible formar el cuadrado

```
1: if length > sum/4 then
2:
3:     return false
4: if length = sum/4 then
5:     if mask = 2^M − 1 then
6:
7:         return true
8:     length ← 0
9: for i ← 0 to M − 1 do
10:    if !(mask&(1 ≪ i)) then
11:        if canFormSquare(length + sticks[i], mask|(1 ≪ i)) then
12:
13:            return true
14:
15: return false
```

Algorithm canFormSquareM(length, mask) // con memoización
Require: longitud actual length, máscara de bits mask
Ensure: si es posible formar el cuadrado

```
1: if memo[mask] ≠ −1 then
2:
3:     return memo[mask]
4: if length > sum/4 then
5:
6:     return memo[mask] ← 0
7: if length = sum/4 then
8:     if mask = 2^M − 1 then
9:
10:        return memo[mask] ← 1
11:    length ← 0
12: for i ← 0 to M − 1 do
13:    if !(mask&(1 ≪ i)) then
14:        if canFormSquareM(length + sticks[i], mask|(1 ≪ i)) then
15:
16:            return memo[mask] ← 1
17:
18: return memo[mask] ← 0
```

Tiempo:

Sin memoización: $O(M!)$, donde M es el número de palos Con memoización: $O(M \cdot 2^M)$, donde el espacio de estados es 2^M y cada estado puede requerir O(M) operaciones

Código:

```
#include <iostream>
#include <cstring>
#include <numeric>
using namespace std;

int M, sum, sticks[20];
int memo[1 << 20];
```

```
 8
 9 bool canFormSquare(int length, int bitMask)
10 {
11     if (memo[bitMask] != -1)
12         return memo[bitMask];
13
14     if (length > sum / 4)
15         return memo[bitMask] = 0;
16
17
18     if (length == sum / 4)
19     {
20         if (bitMask == (1 << M) - 1)
21             return memo[bitMask] = 1;
22
23         length = 0;
24     }
25
26     for (int i = 0; i < M; ++i)
27         if (!(bitMask & (1 << i)) && canFormSquare(length + sticks[i], bitMask | (1 << i)))
28             return memo[bitMask] = 1;
29
30     return memo[bitMask] = 0;
31 }
32
33 int main()
34 {
35     int N;
36     cin >> N;
37     while (N--)
38     {
39         cin >> M;
40         for (int i = 0; i < M; ++i)
41             cin >> sticks[i];
42
43         sum = accumulate(sticks, sticks + M, 0);
44         if (sum % 4 != 0)
45             cout << "no" << endl;
46         else
47         {
48             memset(memo, -1, sizeof memo);
49             cout << (canFormSquare(0, 0) ? "yes" : "no") << endl;
50         }
51     }
52     return 0;
53 }
```

5. Conclusiones

En este artículo, hemos explorado el uso de la técnica de programación dinámica mediante memoización en la resolución de problemas complejos. Se ha aplicado exitosamente el método SRTBOT para diseñar soluciones recursivas eficientes, lo que permitió abordar problemas que inicialmente parecían intratables. A lo largo del proceso, se identificaron ciertos desafíos, como la correcta implementación de la memoización y la gestión de los estados de los problemas, pero estos fueron superados mediante ajustes en la estructura de datos y la optimización de las transiciones. Como resultado, se logró resolver satisfactoriamente varios problemas, destacando la eficiencia y la robustez de la programación dinámica en escenarios de gran escala.

6. Anexos

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

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

All

Mir

Foll

#55935398 | Museum's solution for [UVA-384]

Status	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	2069	C++11 5.3.0	2024-11-11 22:25:37	☐	☐	29955662

Copy C++

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

unordered_map<int, bool> slumpMemo;
unordered_map<int, bool> slimpMemo;

// Función para verificar si una subcadena desde 'pos' es un Slump
bool isSlump(const string &s, int &pos) {
    if (slumpMemo.count(pos)) return slumpMemo[pos];
    int startPos = pos;

    if (s[pos] != 'D' && s[pos] != 'E') return slumpMemo[startPos] = false;
    pos++;

    if (s[pos] != 'F') return slumpMemo[startPos] = false;
    while (s[pos] == 'F') pos++;

    if (s[pos] == 'G') {
        pos++;
        return slumpMemo[startPos] = true;
    } else {
        bool result = isSlump(s, pos);
        return slumpMemo[startPos] = result;
    }
}

// Función para verificar si una subcadena desde 'pos' es un Slimp
bool isSlimp(const string &s, int &pos) {
    if (slimpMemo.count(pos)) return slimpMemo[pos];
    int startPos = pos;

    if (s[pos] != 'A') return slimpMemo[startPos] = false;
    pos++;

    if (s[pos] == 'H') {
        pos++;
        return slimpMemo[startPos] = true;
    }

    if (s[pos] == 'B') {
        pos++;
        if (!isSlimp(s, pos)) return slimpMemo[startPos] = false;
        if (s[pos] == 'C') {
            pos++;
            return slimpMemo[startPos] = true;
        }
    } else if (isSlump(s, pos)) {
        if (s[pos] == 'C') {
            pos++;
            return slimpMemo[startPos] = true;
        }
    }
    return slimpMemo[startPos] = false;
}

// Función para verificar si una cadena es un Slurpy
```

#56036548 | Museum's solution for [UVA-10313]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	890ms	1362	C++11 5.3.0	2024-11-14 19:30:28	☐	☐	29964112

Copy C++

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <unordered_map>
using namespace std;

static const int MAX_VALUE = 300;
typedef unsigned long long ull_t;

unordered_map<string, ull_t> memo;

ull_t countWays(int amount, int maxCoin) {
    if (amount == 0) return 1;
    if (amount < 0 || maxCoin == 0) return 0;

    string key = to_string(amount) + "," + to_string(maxCoin);

    if (memo.find(key) != memo.end()) return memo[key];

    memo[key] = countWays(amount, maxCoin - 1) + countWays(amount - maxCoin, maxCoin);

    return memo[key];
}

int main() {
    string s;
    while (getline(cin, s)) {
        vector<int> N;
        stringstream ss(s);
        int input;
        while (ss >> input) N.push_back(input);

        int L1, L2;
        switch (N.size()) {
            case 1:
                L1 = 0;
                L2 = N[0];
                break;
            case 2:
                L1 = 0;
                L2 = N[1];
                break;
            default:
                L1 = N[1];
                L2 = N[2];
        }

        if (N[0] == 0) {
            cout << "1" << endl;
        } else {
            ull_t result = countWays(N[0], min(MAX_VALUE, L2)) -
                           (L1 > 0 ? countWays(N[0], L1 - 1) : 0);
            cout << result << endl;
        }
    }
    return 0;
}
```

All

Mir

Foll

#56037772 | Museum's solution for [UVA-10364]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	130ms	1479	C++11 5.3.0	2024-11-14 21:31:42	☐	☐	29964363

Copy C++

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

int M, sum, sticks[20];
int memo[1 << 20];

bool canFormSquare(int length, int bitMask)
{
    if (memo[bitMask] != -1)
        return memo[bitMask];

    // Si el largo actual excede el tamaño del lado, no es una solución válida
    if (length > sum / 4)
        return memo[bitMask] = 0;

    // Si logramos un lado, reiniciamos el largo y seguimos buscando los otros lados
    if (length == sum / 4)
    {
        // Si hemos usado todos los palos, formamos un cuadrado
        if (bitMask == (1 << M) - 1)
            return memo[bitMask] = 1;

        // Reiniciamos el largo para formar el siguiente lado
        length = 0;
    }

    // Intentamos agregar palos restantes al lado actual
    for (int i = 0; i < M; ++i)
        if (!(bitMask & (1 << i)) && canFormSquare(length + sticks[i], bitMask | (1 << i)))
            return memo[bitMask] = 1;

    // Si no se encuentra solución, almacenamos y devolvemos 0
    return memo[bitMask] = 0;
}

int main()
{
    int N;
    cin >> N;
    while (N--)
    {
        cin >> M;
        for (int i = 0; i < M; ++i)
            cin >> sticks[i];

        sum = accumulate(sticks, sticks + M, 0);
        if (sum % 4 != 0)
            cout << "no" << endl;
        else
        {
            memset(memo, -1, sizeof memo);
            cout << (canFormSquare(0, 0) ? "yes" : "no") << endl;
        }
    }
    return 0;
}
```