INFORME DE LABORATORIO

INFORMACIÓN BÁSICA					
ASIGNATURA:	Tecnología de Objetos				
TÍTULO DE LA PRÁCTICA:	*Lenguaje de programación C++*				
NÚMERO DE PRÁCTICA:	*02*	**AÑO LECTIVO:**	*2025*	**NRO. SEMESTRE:**	*VI*
FECHA DE PRESENTACIÓN	*20/09/25*	**HORA DE PRESENTACIÓN**	*23:59*		
INTEGRANTE (s): Mamani Huarsaya, Jorge Luis Velarde Saldaña, Jhossep Fabritzio				**NOTA:**	
DOCENTE(s): Ing. Cano Mamani, Edith Giovanna					

SOLUCIÓN Y RESULTADOS

I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS

Link del repositorio: https://github.com/jorghee/teo/tree/main/labs/lab02

Vuelto. Contando el vuelto o cambio.

Escriba una función recursiva que cuente de cuántas maneras diferentes puede dar cambio para una determinada cantidad y de acuerdo a una lista de denominaciones de monedas. Por ejemplo, hay 3 maneras de dar cambio si la cantidad = 4 y tienes monedas con denominación 1 y 2:

1 + 1 + 1 + 1
1 + 1 + 2
2 + 2

Realiza este ejercicio implementando la función `countChange` en C++.
Esta función toma una cantidad a cambiar y un vector de denominaciones únicas para las monedas.

Su definición es la siguiente:

```
C/C++
int countChange(int money, const std::vector<int>& coins);
```

Puedes usar operaciones con índices, iteradores o funciones estándar de **C++ STL** para manejar el vector de monedas.

Solución

La lógica de esta solución recursiva se basa en el principio de "divide y vencerás". Para una cantidad de dinero `money` y un conjunto de monedas `coins`, tomamos una moneda (por conveniencia, la última de la lista) y nos enfrentamos a una decisión con dos caminos posibles:

- **Incluir la moneda actual:** Si usamos la moneda `c`, reducimos la cantidad de dinero que nos queda por cambiar a `money - c`. El problema ahora es encontrar de cuántas maneras podemos dar cambio para esta nueva cantidad, pero aún podemos usar el mismo conjunto de monedas (incluida la moneda `c`, ya que podemos usarla varias veces).
- **Excluir la moneda actual:** Si decidimos no usar la moneda `c`, el problema se reduce a encontrar de cuántas maneras podemos dar cambio para la misma cantidad `money`, pero con un conjunto de monedas más pequeño (todas excepto `c`).

El número total de combinaciones es la suma de las combinaciones encontradas en estos dos caminos.

Implementación

La implementación del problema del vuelto usa memoización porque evita recalcular los mismos subproblemas, lo que simplifica el algoritmo al mantener la claridad recursiva y mejora la eficiencia al reducir la complejidad de exponencial a polinómica.

```
C/C++
#include <iostream>
#include <vector>

int countChange(int money, const std::vector<int>& coins);

std::vector<std::vector<int>> memo;


int countChangeRecursive(int money, const std::vector<int>& coins, int coinIndex) {
  if (money == 0) return 1;
  if (money < 0 || coinIndex < 0) return 0;

  if (memo[coinIndex][money] != -1) {
    return memo[coinIndex][money];
```

```
  }

  int waysWithCurrentCoin = countChangeRecursive(money - coins[coinIndex], coins,
coinIndex);
  int waysWithoutCurrentCoin = countChangeRecursive(money, coins, coinIndex - 1);

  return memo[coinIndex][money] = waysWithCurrentCoin + waysWithoutCurrentCoin;
}

int countChange(int money, const std::vector<int>& coins) {
  memo.assign(coins.size(), std::vector<int>(money + 1, -1));
  return countChangeRecursive(money, coins, coins.size() - 1);
}

int main() {
  int amount = 4;
  std::vector<int> denominations = {1, 2};

  std::cout << "Calculando el numero de maneras de dar cambio para la cantidad: " <<
amount << std::endl;
  std::cout << "Usando las monedas: { ";
  for (size_t i = 0; i < denominations.size(); ++i) {
    std::cout << denominations[i] << (i == denominations.size() - 1 ? "" : ", ");
  }
  std::cout << " }" << std::endl;

  int result = countChange(amount, denominations);

  std::cout << "\nResultado esperado: 3" << std::endl;
  std::cout << "Resultado obtenido: " << result << std::endl;

  return 0;
}
```

Para evitar la creación de sub-vectores en cada llamada (lo cual sería muy ineficiente), hemos utilizado una función auxiliar `countChangeRecursive` que recibe un índice `coinIndex`. Este índice nos dice qué monedas estamos considerando (desde la 0 hasta `coinIndex`).

- **Llamada recursiva 1 (Incluir la moneda):**

  `countChangeRecursive(money - coins[coinIndex], coins, coinIndex)`

  Restamos el valor de la moneda actual (`coins[coinIndex]`) del dinero restante. Luego, mantenemos el mismo `coinIndex`, permitiendo que la misma moneda pueda ser utilizada nuevamente.

- **Llamada recursiva 2 (Excluir la moneda):**

  `countChangeRecursive(money, coins, coinIndex - 1)`

  Mantenemos la misma cantidad de dinero. Luego, decrementamos `coinIndex`, lo que efectivamente "elimina" la moneda actual de consideración para las futuras llamadas en esta rama de la recursión.

Casos Base

La recursión debe detenerse en algún punto. Estos son los casos base que terminan las llamadas:

- `money == 0`: Este es el caso de éxito. Significa que hemos encontrado una combinación exacta de monedas que suma la cantidad original. Devolvemos 1 para contar esta combinación.
- `money < 0`: Hemos "pasado" la cantidad objetivo. Esta rama no es una solución válida, por lo que devolvemos 0.
- `coinIndex < 0`: Nos hemos quedado sin monedas para considerar, pero money todavía es positivo. Esta rama tampoco es una solución válida, así que devolveremos 0.

Prueba

```
 labs/lab02  main  ×1+1?1✓
 > g++ vuelto.cpp -o vuelto.o

 labs/lab02  main  ×1+1?1✓
 > ./vuelto.o
Calculando el numero de maneras de dar cambio para la cantidad: 4
Usando las monedas: { 1, 2 }

Resultado esperado: 3
Resultado obtenido: 3

 labs/lab02  main  ×1+1?1✓
 >
```

II. SOLUCIÓN DEL CUESTIONARIO

1. **Explica el caso base de la implementación recursiva.**
   En esta implementación, existen tres condiciones que actúan como casos base para detener la recursión:

   - **Caso Base de éxito** (`money == 0`). Se ha encontrado una combinación de monedas que suma exactamente la cantidad objetivo. La función retorna 1. Este 1 se propaga hacia arriba en el árbol de llamadas recursivas, sumándose a otras soluciones encontradas

para contribuir al recuento total. Es el único caso que confirma que una ruta de decisiones (incluir/excluir monedas) ha sido exitosa.

- **Caso Base de Fracaso por Exceso** (`money < 0`). La última moneda incluida fue demasiado grande, haciendo que la suma total excediera la cantidad objetivo. La función retorna 0. Poda esta rama del árbol de recursión, ya que no puede conducir a una solución válida.
- **Caso Base de Fracaso por Agotamiento** (`coinIndex < 0`). Se han considerado y excluido todas las monedas disponibles, pero la cantidad de dinero restante (money) sigue siendo mayor que cero. La función retorna 0. Indica que con el subconjunto de monedas considerado en esta rama, es imposible alcanzar la cantidad objetivo.

Estos casos base son fundamentales porque garantizan que la recursión termine y proporcionan los valores (0 o 1) que se combinan para formar el resultado final.

2. **¿Se resuelve mejor este problema con el modelo de programación funcional? Justifica tu respuesta.**
   Sí, este problema se adapta excepcionalmente bien al modelo de programación funcional (FP). La programación funcional se centra en el uso de funciones puras, la inmutabilidad de los datos y la evitación de efectos secundarios. El problema del cambio de monedas se alinea perfectamente con estos principios:
   - **Funciones Puras**: Nuestra función countChange es una función pura. Su salida depende únicamente de sus entradas (money y coins) y no modifica ningún estado externo o global. Para la misma entrada, siempre producirá la misma salida.
   - **Inmutabilidad:** En nuestra implementación, el vector coins nunca se modifica. Las llamadas recursivas operan con valores derivados (`money - coin_value`) o subconjuntos conceptuales (manejados por el `coinIndex`), pero los datos originales permanecen intactos.
   - **Recursión como estructura de control principal:** La solución natural al problema, como hemos visto, es recursiva. La FP favorece la recursión sobre los bucles iterativos para manejar tareas repetitivas, y este problema es un ejemplo clásico de ello.
   - **Composición:** La solución se construye componiendo los resultados de funciones más simples (las llamadas recursivas a sí misma), lo que es un pilar del pensamiento funcional.

   Aunque C++ es un lenguaje multiparadigma y no puramente funcional, la solución implementada adopta un estilo funcional que resulta muy claro, elegante y matemáticamente verificable. Un lenguaje puramente funcional como Haskell o Lisp resolvería este problema de una manera muy similar, destacando la idoneidad del paradigma.

3. **¿Qué beneficios específicos se podrían obtener al usar memoización para optimizar la solución de este problema?**
   La solución recursiva pura es elegante pero muy ineficiente porque sufre de subproblemas superpuestos. Esto significa que calcula el resultado para los mismos estados (misma money y

mismo `coinIndex`) una y otra vez en diferentes ramas del árbol de recursión. La memoización (una técnica de programación dinámica) resuelve este problema almacenando en caché los resultados de los subproblemas.

Los beneficios específicos de aplicar memoización a esta solución serían:

Reducción Exponencial de la Complejidad Temporal:

- Sin memoización: La complejidad temporal es aproximadamente `O(2^(m+n))`, donde n es la cantidad de dinero y m es el número de monedas. Es una complejidad exponencial que se vuelve inviable para entradas moderadamente grandes.
- Con memoización: La complejidad temporal se reduce drásticamente a `O(money * |coins|)`. Esto se debe a que cada estado (`money`, `coinIndex`) se calcula una sola vez. Las llamadas posteriores a un estado ya calculado simplemente devuelven el valor de la caché en tiempo O(1). Esta es la diferencia entre una solución que tarda segundos para money=50 y una que podría tardar años.

Prevención de Desbordamiento de Pila (Stack Overflow):

La recursión profunda puede agotar el espacio de la pila de llamadas. Al "podar" ramas enteras del árbol de recursión cuyos resultados ya se conocen, la memoización reduce drásticamente la profundidad efectiva de la recursión, haciendo que la solución sea viable para entradas mucho más grandes sin riesgo de stack overflow.

Conservación de la Claridad del Código Recursivo:

La memoización permite mantener la estructura lógica y la elegancia de la solución recursiva de "arriba hacia abajo" (top-down). La lógica principal del problema (dividir en incluir/excluir) permanece intacta. Simplemente se le añade una capa de almacenamiento en caché (usando un `std::map` o un `std::vector<std::vector<int>>`), lo que a menudo es más intuitivo que reescribir la solución de forma iterativa "de abajo hacia arriba" (bottom-up DP).

III. CONCLUSIONES

Esta práctica fue clave para solidificar mi entendimiento de la recursión. Aprendí a descomponer un problema complejo en subproblemas más simples, destacando la importancia crítica de los casos base. Sin embargo, la lección más importante fue ver la ineficiencia de una solución recursiva pura por sus cálculos redundantes. Las preguntas teóricas me revelaron cómo la memoización, una técnica de programación dinámica, optimiza drásticamente el rendimiento, transformando una complejidad exponencial en una polinómica y demostrando la elegancia del paradigma funcional.

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA