# 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:** 20232193 - VALDIVIA SEGOVIA RYAN FABIAN | | | | | |
| **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 trabajo se explora el uso de programación dinámica, combinada con el enfoque SRTBOT, para resolver problemas recursivos complejos. El objetivo fue aplicar la técnica de memoización a un algoritmo recursivo y demostrar sus ventajas en términos de eficiencia, al evitar cálculos repetidos. Se resolvieron tres problemas utilizando esta técnica, con el fin de optimizar las soluciones y comparar los resultados con y sin memoización. A lo largo del reporte, se explicó el uso del método SRTBOT, detallando cada una de sus etapas para diseñar algoritmos recursivos eficientes. Se concluye que la memoización mejora significativamente la eficiencia de los algoritmos, especialmente al abordar problemas con subproblemas recurrentes.


## 1. Introducción

El presente trabajo aborda el diseño de soluciones recursivas utilizando el enfoque SRTBOT, una metodología para la creación de algoritmos recursivos eficientes. Este enfoque, combinado con la programación dinámica y la técnica de memoización, permite optimizar problemas complejos que involucran subproblemas recurrentes. En este estudio, se exploró la aplicación de la memoización en problemas de combinaciones y optimización, destacando sus beneficios en cuanto a reducción de la complejidad temporal.

El objetivo principal de este trabajo fue demostrar cómo el uso de la programación dinámica y la memoización puede mejorar el rendimiento de los algoritmos recursivos. Para ello, se resolvieron tres problemas aplicando estos enfoques y comparando los resultados obtenidos con y sin memoización. Los resultados experimentales muestran una mejora significativa en la eficiencia del algoritmo, confirmando la efectividad de la memoización.

El resto del artículo está organizado de la siguiente manera: la Sección 2 describe los conceptos teóricos fundamentales de la programación dinámica y el enfoque SRTBOT; la Sección 3 presenta el proceso seguido para resolver los problemas propuestos; la Sección 4 muestra los resultados obtenidos y su comparación; y la Sección 5 concluye con las observaciones finales y recomendaciones.

## 2. Marco Teórico Conceptual

Esta sección presenta los conceptos teóricos necesarios para entender el problema y los métodos de solución aplicados en este trabajo. El marco conceptual está formado por los conceptos de programación dinámica, memoización y el enfoque SRTBOT, todos los cuales son fundamentales para el desarrollo de soluciones eficientes a problemas recursivos.

**Programación dinámica** fue propuesta por Bellman (1952) como una técnica para resolver problemas que pueden ser divididos en subproblemas más pequeños y cuya solución puede ser reutilizada. Esta técnica es especialmente útil para problemas donde los subproblemas se repiten, como en el caso de la optimización combinatoria.

El nemotécnico SRTBOT, propuesto por Demaine (2021), es una estrategia para diseñar algoritmos recursivos que resuelvan problemas complejos. Este enfoque se basa en seis pasos:

**Subproblemas:** El primer paso consiste en dividir el problema original en subproblemas más pequeños, los cuales son más fáciles de resolver individualmente.

**Relaciones Recursivas:** Este paso implica identificar cómo los subproblemas se relacionan entre sí y cómo pueden combinarse para resolver el problema original.

**Topología:** La topología de los subproblemas nos ayuda a visualizar la estructura de la solución, indicando cómo se pueden agrupar y resolver los subproblemas.

**Bases:** Los casos base son fundamentales para terminar la recursión. Un caso base es una condición que no puede ser dividida más y que tiene una solución trivial.

**Original:** El siguiente paso es usar los cuatro puntos anteriores para diseñar un algoritmo recursivo que resuelva el problema original. En este punto, la **memoización** juega un papel clave, ya que permite almacenar las soluciones de los subproblemas para evitar recalcularlas repetidamente.

**Tiempo:** Finalmente, se realiza un análisis de la complejidad temporal del algoritmo, asegurándose de que la solución sea eficiente y escalable.

# 3. Diseño Experimental

En esta sección se describe el proceso seguido para resolver los problemas propuestos utilizando el enfoque SRTBOT y la técnica de memoización. Los problemas seleccionados implicaban optimización y combinaciones, donde la solución recursiva sin memoización presentaba una alta complejidad temporal debido a la repetición de cálculos. El objetivo fue optimizar estos algoritmos mediante la implementación de memoización.

Primero, se analizó cada problema para identificar los subproblemas y establecer las relaciones recursivas entre ellos. A continuación, se aplicaron los pasos de SRTBOT para formular la solución recursiva, incorporando la memoización para evitar la recalculación de subproblemas. Para validar la eficiencia de los algoritmos, se compararon los resultados obtenidos con y sin memoización. Se observó que la memoización redujo significativamente el tiempo de ejecución, permitiendo resolver los problemas en un tiempo mucho menor.

Durante el proceso, se encontraron ciertos desafíos relacionados con la gestión de la memoria y la estructura de los datos, pero estos fueron superados mediante la implementación cuidadosa de las estructuras adecuadas y la optimización de los estados en la memoización.

## 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.
- Analizar y aplicar el método SRTBOT para resolver problemas propuestos.

## 3.2. Actividades

Para la resolución de este laboratorio, se realizaron las siguientes acciones:

1. Se creó un usuario en la plataforma Virtual Judge http://vjudge.net, con el nombre de `rvaldiviase`
2. Se seleccionó tres ejercicios aleatorios de la lista proporcionada por el docente http://bit.ly/3UxdCVL.
3. Para cada problema, se diseñó una solución utilizando la técnica SRTBOT.
4. Se realizó el pseudocódigo resultante del modelo SRTBOT
5. Se diseño el código en C++ que resulta del modelo SRTBOT, con y sin memoización.
6. Se anexaron imagenes de las soluciones aceptadas en la plataformahttp://vjudge.net al final del artículo.

# 4. Resultados

A continuación, se muestran los resultados obtenidos de cada uno de los problemas desarollados

## 4.1. Problema 1213 – Sum of Different Primes

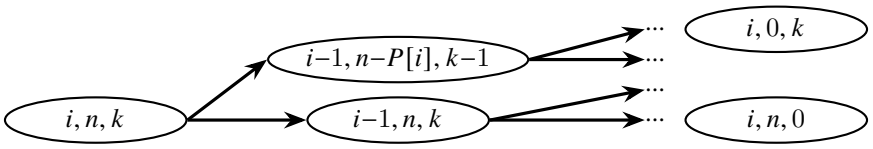
**Problema: Expresión de un Número como Suma de Primos**

**Subproblema:** Contar el número de maneras de expresar un número entero positivo $n$ como la suma de $k$ números primos diferentes.

**Relaciones Recursivas:**

$$C(i,n,k) = \begin{cases} 0, & \text{si } n < 0 \text{ o } k < 0 \\ 1, & \text{si } n = 0 \text{ y } k = 0 \\ C(i-1,n,k), & \text{si } P[i] > n \\ C(i-1,n,k) + C(i-1,n-P[i],k-1), & \text{si } P[i] \leq n \end{cases}$$

**Topología:**




**Básico:**

$$C(i,n,k) = 0 \quad \text{si } n < 0 \text{ o } k < 0$$
$$C(i,0,k) = 1 \quad \text{si } k = 0 \text{ y } n = 0$$
$$C(0,n,k) = 0 \quad \text{si } n > 0 \text{ o } k > 0$$

**Original:**

**Algorithm** C($n, k, P, index$) // without memoization

**Input:** target number $n$, number of primes $k$, primes list $P$ with all the prime numbers less than $n$, current prime index $index$

**Output:** number of ways to express $n$ using $k$ primes

```
1: if k = 0 then
2:     if n = 0 then
3:         return 1
4:     else
5:         return 0
6: if n ≤ 0 or index ≥ length of P then
7:     return 0
8: include = C(n − P[index], k − 1, P, index + 1)
9: exclude = C(n, k, P, index + 1)
10: return include + exclude
```

**Algorithm** CM($n, k, P, memo$) // with memoization

**Input:** target number $n$, number of primes $k$, primes list $P$ with all the prime numbers less than $n$, current prime index $index$, memoization table $memo$

**Output:** number of ways to express $n$ using $k$ primes

```
1: if k = 0 then
2:     if n = 0 then
3:         return 1
4:     else
5:         return 0
6: if n ≤ 0 or index ≥ length of P then
7:     return 0
8: if memo[n][k][index] ≠ −1 then
9:     return memo[n][k][index]
10: include = CM(n − P[index], k − 1, P, memo, index + 1)
11: exclude = CM(n, k, P, memo, index + 1)
12: memo[n][k][index] = include + exclude
13: return memo[n][k][index]
```

**Tiempo de ejecución:** $CM(i,n,k) \in O(i \times n \times k)$

**Código:**

Listing 1: Generación de primos y conteo de formas

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

std::vector<int> generatePrimes (int n);

int countWays (int n, int k, std::vector<int>& primes);

int countWays (int n, int k, std::vector<int>& primes, int index, std::vector<std::vector<std::vector<int>>>& memo);

std::vector<int> generatePrimes (int n) {
    std::vector<int> primes;
    std::vector<bool> isPrime(n + 1, true);

    for (int p = 2; p <= n; ++p) {
        if (isPrime[p]) {
            primes.push_back(p);
            for (int i = p * p; i <= n; i += p) {
                isPrime[i] = false;
            }
        }
    }
    return primes;
}

int countWays (int n, int k, std::vector<int> &primes, std::vector<std::vector<std::vector<int>>>& memo) {
    return countWays(n, k, primes, 0, memo);
}

int countWays (int n, int k, std::vector<int>& primes, int index, std::vector<std::vector<std::vector<int>>>& memo) {
    if (k == 0) {
        return n == 0 ? 1 : 0;
    }
    if (n <= 0 || index >= primes.size()) {
        return 0;
    }

    if (memo[n][k][index] != -1) {
        return memo[n][k][index];
    }

    int include = countWays(n - primes[index], k - 1, primes, index + 1, memo);
    int exclude = countWays(n, k, primes, index + 1, memo);

    int result = include + exclude;
    memo[n][k][index] = result;
    return result;
}
```

Listing 2: Función principal del programa

```cpp
int main () {
    std::vector<int> primes = generatePrimes(1120);
    int maxN = 1120;
    int maxK = 14;
    int maxIndex = primes.size();
    std::vector<std::vector<std::vector<int>>> memo(maxN + 1, std::vector<std::vector<int>>
    (maxK + 1, std::vector<int>(maxIndex, -1)));

    while (true) {
        int n, k;
        std::cin >> n >> k;

        if (n == 0 && k == 0) {
            break;
        }

        std::cout << countWays(n, k, primes, memo) << std::endl;
    }

    return 0;
}
```

## 4.2. Problema 10912 – Simple minded hashing

**Problema: Número de Cadenas con Hashing Simple**

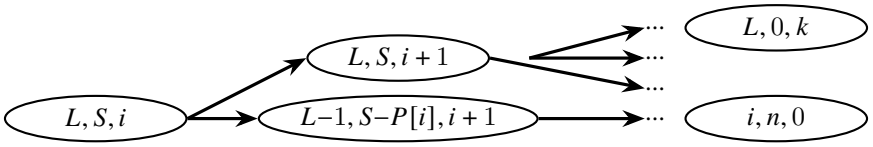
**Subproblema:** Contar el número de cadenas de longitud $L$ que se mapean a un valor de suma $S$, usando el esquema de hashing simple descrito, con la restricción de que las letras de la cadena deben estar en orden estrictamente ascendente.

**Relaciones Recursivas:**

$$C(L,S,i) = \begin{cases} 0, & \text{si } S < 0 \text{ o } L \leq 0 \\ 1, & \text{si } S = 0 \text{ y } L = 0 \\ C(L-1, S-P[i], i+1) + C(L,S,i+1), & \text{si } P[i] \leq S \end{cases}$$

Donde $P[i]$ es el valor numérico asignado a la letra $i$ (es decir, $P[i] = i + 1$, con $i$ representando el índice de la letra en el alfabeto).

**Topología:**




**Básico:**

$$C(L,S,i) = 0 \quad \text{si } L \leq 0 \text{ o } S < 0$$
$$C(0,S,i) = 1 \quad \text{si } L = 0 \text{ y } S = 0$$

**Original:**

**Algorithm** C(sum, length, last) // without memoization

**Input:** target hashcode *sum*, target string length *length*, last used value *last*

**Output:** number of ways to express sum using length numbers in strictly increasing order

```
1:  if length = 0 then
2:      if sum = 0 then
3:          return 1
4:      else
5:          return 0
6:  if sum < 0 or length < 0 or last×length > sum then
7:      return 0
8:  count = 0
9:  for i = last + 1 to 26 do
10:     count += C(sum − i, length − 1, i)
11: return count
```

**Algorithm** CM(sum, length, last, memo) // with memoization

**Input:** target hashcode *sum*, target string length *length*, last used value *last*, memoization table *memo*

**Output:** number of ways to express sum using length numbers in strictly increasing order

```
1:  if length = 0 then
2:      if sum = 0 then
3:          return 1
4:      else
5:          return 0
6:  if sum < 0 or length < 0 or last × length > sum then
7:      return 0
8:  if memo[sum][length][last] ≠ −1 then
9:      return memo[sum][length][last]
10: count = 0
11: for i = last + 1 to 26 do
12:     count += CM(sum − i, length − 1, i, memo)
13: memo[sum][length][last] = count
14: return count
```

**Tiempo de ejecución:**

$$\mathrm{CM}(i, \text{sum}, \text{length}) \in O(i \times \text{sum} \times \text{length})$$

**Código:**

Listing 3: Función recursiva con memoización para contar las combinaciones

```cpp
#include <iostream>

int memo[352][27][27];

int collisions(int sum, int length, int last) {

    if (sum < 0 || length < 0 || last * length > sum) {
        return 0;
    }

    if (length == 0 && sum == 0) {
        return memo[sum][length][last] = 1;
    }

    if (memo[sum][length][last] != -1) {
        return memo[sum][length][last];
    }

    unsigned int count = 0;

    for (int i = last + 1; i < 27; i++) {
        count += collisions(sum - i, length - 1, i);
    }

    memo[sum][length][last] = count;
    return count;
}
```

Listing 4: Función principal del programa con entrada y salida de resultados

```cpp
int main() {
    for (int i = 0; i < 352; i++)
        for (int j = 0; j < 27; j++)
            for (int k = 0; k < 27; k++)
                memo[i][j][k] = -1;

    int i = 1;
    while (true) {
        int length, sum;
        std::cin >> length >> sum;

        if (length == 0 && sum == 0) {
            break;
        }

        if (length > 26 || sum > 351) {
            std::cout << "Case␣" << i << ":␣" << 0 << std::endl;
        } else {
            std::cout << "Case␣" << i << ":␣" << collisions(sum, length, 0) << std::endl;
        }

        i++;
    }

    return 0;
}
```

## 4.3. Problema 11218 – KTV

**Problema: Dividir a 9 personas en 3 grupos de 3 para maximizar la puntuación.**

**Subproblema:** Encontrar la manera de dividir 9 personas en 3 grupos de 3 para maximizar la puntuación dada para cada combinación de 3 personas.

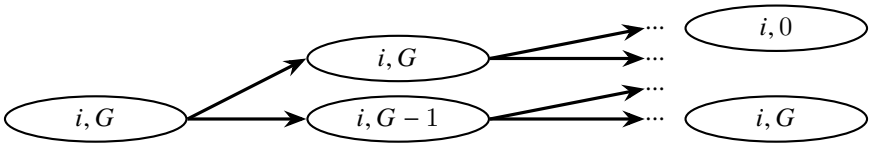
**Relaciones Recursivas:**

$$S(i,G) = \begin{cases} -1, & \text{si no hay más combinaciones posibles para el grupo } G \\ S(i,G-1) + \text{puntuación}(i), & \text{si el grupo } G \text{ es válido para agregar } i \end{cases}$$

Donde:

- $i$ es el índice de la combinación.
- $G$ es el número de grupos que quedan por formar.
- **Puntuación** se refiere a la puntuación dada para una combinación de 3 personas en el problema original.

**Topología:**




**Básico:**

$$S(i,3) = 0 \quad \text{si } G = 0 \text{ (sin más combinaciones)}$$

$$S(i,G) = -1 \quad \text{si no es posible encontrar combinaciones válidas}$$

$$S(i,3) = \max(S(i,G-1) + \text{puntuación}(i)) \quad \text{si el grupo es válido}$$

**Original:**

**Algorithm** maxScoreRec($taken$, $combinations$, $groups$) // without memoization

**Input:** vector of booleans $taken$, list of combinations $combinations$, number of groups formed so far $groups$

**Output:** maximum score achievable by forming 3 groups

```
1: if groups = 3 then
2:     return 0
3: maxScore = -1
4: for each combination in combinations do
5:     a, b, c, score ← comb
6:     if not taken[a−1] and not taken[b−1] and not taken[c − 1] then
7:         mark a, b, c as taken
8:         newScore = maxScoreRec(taken, combinations, groups + 1)
9:         if newScore != -1 then
10:            maxScore = max(maxScore, newScore + score)
11:        unmark a, b, c as taken
12: return maxScore
```

**Algorithm** maxScoreRec($taken$, $combinations$, $groups$, $memo$) // with memoization

**Input:** vector of booleans $taken$, list of combinations $combinations$, number of groups formed so far $groups$, memoization table $memo$

**Output:** maximum score achievable by forming 3 groups

```
1: if groups = 3 then
2:     return 0
3: if memo[groups] ≠ −1 then
4:     return memo[groups]
5: maxScore = -1
6: for each combination in combinations do
7:     a, b, c, score ← comb
8:     if not taken[a − 1] and not taken[b − 1] and not taken[c − 1] then
9:         mark a, b, c as taken
10:        newScore = maxScoreRec(taken, combinations, groups + 1, memo)
11:        if newScore != -1 then
12:            maxScore = max(maxScore, newScore + score)
13:        unmark a, b, c as taken
14: memo[groups] ← maxScore
15: return maxScore
```

**Tiempo de ejecución:** El tiempo de ejecución con memoización es $O(n \times 2^n)$, donde $n$ es el número de combinaciones.

**Código:**

Listing 5: Función recursiva con memoización para contar las combinaciones

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

int maxScore (const std::vector<std::vector<int>>& combinations, std::unordered_map<std::string, int>& memo);

int maxScoreRec (std::vector<bool>& taken, const std::vector<std::vector<int>>& combinations, int groups,
std::unordered_map<std::string, int>& memo);

int maxScore (const std::vector<std::vector<int>>& combinations,
std::unordered_map<std::string, int>& memo) {
    std::vector<bool> taken(9, false);
    return maxScoreRec(taken, combinations, 0, memo);
}

int maxScoreRec (std::vector<bool>& taken, const std::vector<std::vector<int>>& combinations,
int groups, std::unordered_map<std::string, int>& memo) {
    if (groups == 3) {
        return 0;
    }

    std::string state = "";
    for (bool taken : taken) {
        state += taken ? "1" : "0";
    }

    if (memo.find(state) != memo.end()) {
        return memo[state];
    }

    int maxScore = -1;

    for (auto& comb : combinations) {
        int a = comb[0], b = comb[1], c = comb[2], score = comb[3];

        if (!taken[a - 1] && !taken[b - 1] && !taken[c - 1]) {
            taken[a - 1] = taken[b - 1] = taken[c - 1] = true;

            int newScore = maxScoreRec(taken, combinations, groups + 1, memo);

            if (newScore != -1) {
                maxScore = std::max(maxScore, newScore + score);
            }

            taken[a - 1] = taken[b - 1] = taken[c - 1] = false;
        }
    }

    memo[state] = maxScore;
    return maxScore;
}
```

Listing 6: Función principal del programa con entrada y salida de resultados

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

int main () {
    std::unordered_map<std::string, int> memo;

    for (int i = 0; i < 1000; i++) {
        int n;
        std::cin >> n;

        if (n == 0) {
            break;
        }

        std::vector<std::vector<int>> combinations;

        for (int j = 0; j < n; j++) {
            int a, b, c, s;
            std::cin >> a >> b >> c >> s;

            combinations.push_back(std::vector<int>({a, b, c, s}));
        }
        std::cout << "Case␣" << i + 1 << ":␣" << maxScore(combinations, memo) << std::endl;
    }
    return 0;
}
```

# 5. Conclusiones

En este reporte se exploraron métodos de diseño de soluciones recursivas, destacando el enfoque SRTBOT, que combina la recursividad con técnicas de programación dinámica, como la memoización. La aplicación de la memoización demostró ser fundamental para mejorar el rendimiento de los algoritmos, evitando cálculos repetidos y reduciendo la complejidad temporal. Al comparar los algoritmos con y sin memoización, se evidenció una mejora significativa en la eficiencia, lo que permitió resolver problemas complejos de combinaciones y optimización de manera más rápida y escalable.

A pesar de los desafíos en la adaptación de estructuras de datos y la gestión de estados dentro del marco SRTBOT, los resultados obtenidos confirmaron la efectividad de esta técnica al abordar problemas con subproblemas recurrentes. Este trabajo resalta la importancia de optimizar soluciones recursivas a través de la programación dinámica, utilizando el enfoque SRTBOT como una herramienta potente para enfrentar problemas de optimización y combinaciones, logrando soluciones más eficientes y robustas.

# 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.

## #56033899 | rvaldiviase's solution for [UVA-10912]

| Status | Time | Length | Lang | Submitted | Open | Share text | RemoteRunId |
|---|---|---|---|---|---|---|---|
| Accepted | 10ms | 1141 | C++11 5.3.0 | 2024-11-14 13:10:30 | ☐ | ☐ | 29963550 |

```cpp
#include <iostream>
#include <vector>

int memo[352][27][27];

int collisions(int sum, int length, int last) {

    if (sum < 0 || length < 0 || last * length > sum) {
        return 0;
    }

    if (length == 0 && sum == 0) {
        return memo[sum][length][last] = 1;
    }

    if (memo[sum][length][last] != -1) {
        return memo[sum][length][last];
    }

    unsigned int count = 0;

    for (int i = last + 1; i < 27; i++) {
        count += collisions(sum - i, length - 1, i);
    }

    memo[sum][length][last] = count;
    return count;
}

int main() {
    for (int i = 0; i < 352; i++)
        for (int j = 0; j < 27; j++)
            for (int k = 0; k < 27; k++)
                memo[i][j][k] = -1;

    int i = 1;
    while (true) {
        int length, sum;
        std::cin >> length >> sum;

        if (length == 0 && sum == 0) {
            break;
        }

        if (length > 26 || sum > 351) {
            std::cout << "Case " << i << ": " << 0 << std::endl;
        } else {
            std::cout << "Case " << i << ": " << collisions(sum, length, 0) << std::endl;
        }

        i++;
    }

    return 0;
}
```

## #55929114 | rvaldiviase's solution for [UVA-1213]

| Status | Time | Length | Lang | Submitted | Open | Share text | RemoteRunId |
|---|---|---|---|---|---|---|---|
| Accepted | 60ms | 1828 | C++11 5.3.0 | 2024-11-11 13:00:59 | ☐ | ☐ | 29954733 |

Copy C++

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

std::vector<int> generatePrimes (int n);

int countWays (int n, int k, std::vector<int>& primes);

int countWays (int n, int k, std::vector<int>& primes, int index,
std::vector<std::vector<std::vector<int>>>& memo);

std::vector<int> generatePrimes (int n) {
    std::vector<int> primes;
    std::vector<bool> isPrime(n + 1, true);

    for (int p = 2; p <= n; ++p) {
        if (isPrime[p]) {
            primes.push_back(p);
            for (int i = p * p; i <= n; i += p) {
                isPrime[i] = false;
            }
        }
    }
    return primes;
}

int countWays (int n, int k, std::vector<int> &primes,
std::vector<std::vector<std::vector<int>>>& memo) {
    return countWays(n, k, primes, 0, memo);
}


int countWays (int n, int k, std::vector<int>& primes, int index,
std::vector<std::vector<std::vector<int>>>& memo) {
    if (k == 0) {
        return n == 0 ? 1 : 0;
    }
    if (n <= 0 || index >= primes.size()) {
        return 0;
    }

    if (memo[n][k][index] != -1) {
        return memo[n][k][index];
    }

    int include = countWays(n - primes[index], k - 1, primes, index + 1, memo);
    int exclude = countWays(n, k, primes, index + 1, memo);

    int result = include + exclude;
    memo[n][k][index] = result;
    return result;
}

```

## #55929169 | rvaldiviase's solution for [UVA-11218]

| Status | Time | Length | Lang | Submitted | Open | Share text | RemoteRunId |
|---|---|---|---|---|---|---|---|
| Accepted | 750ms | 2039 | C++11 5.3.0 | 2024-11-11 13:04:30 | ☐ | ☐ | 29954747 |

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

int maxScore (const std::vector<std::vector<int>>& combinations);

int maxScoreRec (std::vector<bool>& taken, const std::vector<std::vector<int>>& combinations, int groups, std::unordered_map<std::string, int> memo);

int maxScore (const std::vector<std::vector<int>>& combinations, std::unordered_map<std::string, int>& memo) {
    std::vector<bool> taken(9, false);
    return maxScoreRec(taken, combinations, 0, memo);
}

int maxScoreRec (std::vector<bool>& taken, const std::vector<std::vector<int>>& combinations, int groups, std::unordered_map<std::string, int> memo) {
    if (groups == 3) {
        return 0;
    }

    std::string state = "";
    for (bool taken : taken) {
        state += taken ? "1" : "0";
    }

    if (memo.find(state) != memo.end()) {
        return memo[state];
    }

    int maxScore = -1;

    for (auto& comb : combinations) {
        int a = comb[0], b = comb[1], c = comb[2], score = comb[3];

        if (!taken[a - 1] && !taken[b - 1] && !taken[c - 1]) {
            taken[a - 1] = taken[b - 1] = taken[c - 1] = true;

            int newScore = maxScoreRec(taken, combinations, groups + 1, memo);

            if (newScore != -1) {
                maxScore = std::max(maxScore, newScore + score);
            }

            taken[a - 1] = taken[b - 1] = taken[c - 1] = false;
        }
    }

    memo[state] = maxScore;
    return maxScore;
}



int main () {
    std::unordered_map<std::string, int> memo;

```