Programación Dinámica

Víctor Racsó Galván Oyola
https://github.com/racsosabe/DP

November 2016

Índice general

1. Introducción **4**

1.1. ¿A qué nos referimos con Programación Dinámica? 4

1.2. Soluciones con Programación Dinámica 4

2. Elementos de la Programación Dinámica **5**

2.1. Subestructura Óptima de la solución 5

2.1.1. Estado 5

2.2. Sutilezas 6

2.3. Subproblemas Sobrelapados 6

2.4. Reconstruyendo la Solución Óptima 6

2.5. Almacenamiento 6

3. Tipos de Programación Dinámica **7**

3.1. DP Recursivo - Top Down 7

3.2. DP Iterativo - Bottom Up 7

4. Problemas Típicos de Programación Dinámica **8**

4.1. Coeficientes de expansión Binomial 8

4.2. Números de Catalán 10

4.3. Sucesión de Fibonnaci 11

4.3.1. Versión Directa $< O(1), O(n) >$ 11

4.3.2. Versión Almacenada $< O(n), O(n) >$ 11

4.3.3. Versión Almacenada Rápida $< O(\log n), O(\log^2 n) >$ 12

4.4. Máxima Suma de Rango de 1 Dimensión (*Max 1D Range Sum*) 13

4.5. Máxima Suma de Rango de 2 Dimensiones (*Max 2D Range Sum*) 14

4.6. Corte de Barra (*Rod Cutting*) 16

4.7. Multiplicación de Cadenas de Matrices 17

4.8. Subsecuencia Común Más Larga (*Longest Common Subsequence* LCS) 19

4.9. Subsecuencia Creciente Más Larga (*Longest Increasing Subsequence* LIS) 20

4.10. Problema de la Mochila (*Knapsack Problem*) 21

4.11. Cambio de Monedas (Versión General) 22

4.12. Problema del Cartero (*Travelling Salesman Problem*) 24

4.13. Alineamiento de Cadenas (*Edit Distance*) 25

5. Problemas No Típicos de Programación Dinámica **27**

5.1. *Word Wrap Problem* 27

5.2. Subsecuencia Palíndroma Más Larga (*Longest Palindromic Subsequence*) 29

5.3. LIS con Binary Search 31

5.4. Subsecuencia Bitónica Más Larga (*Longest Bitonic Subsequence*) 32

5.5. Subsecuencia Ordenada Más Larga (Versión General) 34

5.5.1. Teorema de Dilworth 34

5.6. Particionamiento de Palíndromos (*Minimum Palindrome Partitioning*) 35

5.7. Encadenamiento Lineal (*One-Dimensional Chaining Problem*) 37

6. Técnicas Avanzadas con DP **39**

6.1. DP con *bitmask* 39

6.1.1. Forming Quiz Teams - 10911 UVa 39

6.1.2. Another Sith Tournament - 678E Codeforces 40

6.1.3. Little Pony and Harmony Chest - 453B Codeforces 42

6.2. DP sobre árboles 44

6.2.1. Minimum Vertex Cover - Árboles 44

6.2.2. Corte de Máximo Valor 46

6.3. DP sobre dígitos 47

6.3.1. Conteo de Números - UVa 11472 47

6.3.2. Conteo de Números - Cantidad de Palíndromos 50

6.3.3. Conteo en Rango - LOJ 1068 54

6.4. DP - Mínimo Lexicográfico 58

6.4.1. UVA 10419 - Sum-up the Primes 58

Capítulo 1

Introducción

1.1. ¿A qué nos referimos con Programación Dinámica?

Ya hemos escuchado sobre los algoritmos "Dividir y Conquistar", los cuales dividen un problema en subproblemas y los resuelven para luego obtener la respuesta total combinando las subrespuestas. Por su parte, la Programación Dinámica o **DP** (Dynamic Programming) se aplica cuando los subproblemas se sobreponen, es decir, los subproblemas comparten parcialmente sus respuestas. En tal caso, los algoritmos "Dividir y Conquistar" hacen mucho más trabajo que el necesario, resolviendo múltiples veces la misma porción de problema. La programación dinámica resuelve cada subproblema y guarda esa respuesta, con el fin de que ahorrarnos el tener que volver a procesar el problema en ese punto y solamente se usan los datos guardados, siendo más eficientes de esa manera.

Esto puede sonar muy confuso, pero recordemos el tema de **backtracking**. Por lo general, este se plantea en los problemas que tengan k opciones de acción en cada paso u objeto del problema, formando un árbol $k-$ario al desarrollar todas las respuestas posibles. Considerando cada uno de los subárboles de acción, es muy probable que algunos se repitan: aquí es donde aplicamos la programación dinámica.

Es común usar DP en **problemas de optimización**. Este tipo de problemas puede tener diferentes formas de obtener el valor óptimo (sea máximo o mínimo), por lo que el DP va a hallar una solución óptima con un camino de elecciones no necesariamente único.

1.2. Soluciones con Programación Dinámica

Para resolver problemas determinados con DP, se deben seguir algunos pasos necesarios y otros complementarios al plantear el algoritmo. En principio debemos **definir una estructura óptima** para poder procesar correctamente la solución. En segundo lugar, debemos **hallar una recursión** para obtener la solución óptima, para luego calcularla con nuestro programa. Un paso extra usualmente solicitado en los problemas es reconstruir una posible solución óptima o hallar una que cumpla con determinadas características para volverla única (puede ser orden lexicográfico, la que tenga mayor cantidad de estados atravesados, etc.)

Capítulo 2

Elementos de la Programación Dinámica

2.1. Subestructura Óptima de la solución

Una de las condiciones que un problema debe cumplir para que exista la posibilidad de que aplicar DP sea preferible es que la solución del problema tenga forma de subestructura; es decir, que se puedan plantear *estados* y que cada uno de estos pueda usar directa o indirectamente *estados* **anteriormente** calculados y casos base.

2.1.1. Estado

Un estado se puede definir como un conjunto de parámetros que expresan decisiones tomadas para formar una solución óptima. Los parámetros de los estados dependen de la estructura que tenga la solución y es preferible que tengan la mínima cantidad de variaciones y elementos posible (para no desperdiciar tiempo y memoria respectivamente).

La naturaleza del DP es formar un DAG (Grafo dirigido acíclico) al momento de plantear las decisiones posibles y además de ello siempre se puede expresar como una recursión, la cual dependerá de los parámetros para calcular su complejidad y sus límites de variación.

Considerando esto, se puede asumir que el DAG formado tiene pesos en sus aristas, los cuales serían asociados a cada transición entre estado y estado, y para que el problema tenga subestructura óptima se deberá cumplir que, definida la distancia óptima $\delta(x, i)$, siendo x el estado inicial, y el estado objetivo e i y j estados intermedios del camino óptimo, entonces:

$$\delta(x, j) = \delta(x, i) + w(i, j)$$

Donde $w(i, j)$ es el peso de la transición $i \to j$. Esto se debe cumplir para todo i,j que pertenezcan al camino óptimo.

2.2. Sutilezas

Es importante reconocer en qué momentos se puede aplicar una subestructura óptima y cuando no se cumple. Lo mejor es demostrar la estructura de la solución óptima y llegar a la conclusión de que la recursión usada sí dará la respuesta correcta. Este impedimento se logra evitar principalmente debido a la experiencia con este tipo de problemas usando el paradigma. Por ejemplo, el problema del camino más corto entre dos nodos de un grafo presenta subestructura óptima, pero el camino más largo no.

2.3. Subproblemas Sobrelapados

Lo que el DP aprovecha de los problemas que puede resolver es su capacidad de sobrelapado, pues al momento de desarrollar problemas pequeños y casos base, se pueden formar soluciones a otros más grandes mezclando las soluciones ya halladas. Esto no sucede estrictamente en Dividir y Conquistar debido a que en D&C siempre se crean nuevos problemas en las recursiones, además de que no todas las respuestas base se van a usar en diferentes problemas grandes.

2.4. Reconstruyendo la Solución Óptima

Al momento de intentar reconstruir la solución óptima de un problema al cual se le ha aplicado DP se debe considerar el camino que se recorrió en el árbol de las recurrencias; para ello, es fundamental guardar en una tabla la dirección de las transiciones y además plantear un estado como inicial.

La reconstrucción de la solución óptima de los problemas típicos mostrados en este capítulo estarán en el repositorio de Github asociado al tema.

2.5. Almacenamiento

Una de las formas del DP podría definirse como una “salida rápida” del problema de la ineficiencia computacional. La idea es guardar las respuestas de la natural, pero poco eficiente, recursión que se puede plantear para la solución, optimizando así el tiempo de ejecución de manera significativa, o formar los casos base del problema para construir el resto de estados.

Capítulo 3

Tipos de Programación Dinámica

Existen 2 maneras de plantear una solución usando DP, los dos se adaptan mejor para diferentes tipos de problema, pero en general ambos métodos sirven.

3.1. DP Recursivo - Top Down

El DP recursivo es también considerado *Backtracking con Almacenamiento*, debido a que lo único en lo que se diferencia del *Backtracking* es que marca los estados que ya se han visitado (y por ende, procesado) de manera que la siguiente vez que se desee su respuesta, la devuelva de manera directa. Su concepto en sí podría señalarse como "Eventualmente tendré una respuesta, por lo que no importa cuándo la obtenga mientras lo haga", haciendo analogía a su forma de atravesar el DAG que se forma con los estados (estilo DFS).

Este método se basa en colocar en la estructura en la que se almacenarán las respuestas para cada estado un valor *dummy* que signifique que el estado aún no ha sido procesado; para que al momento de sobreescribir la respuesta haya una diferencia notable y sea sencillo distinguir entre los estados visitados y los no visitados. Una alternativa a esto (cuando las respuestas son estructuras a las cuales es complicado asignar un valor *dummy*) es mantener una estructura de booleanos, idéntica a la de las respuestas, que determine si el estado fue procesado o aún no.

3.2. DP Iterativo - Bottom Up

El DP iterativo puede ser considerado como el "DP Natural", puesto que fue como el DP tuvo origen. Su planteamiento es primero determinar todos los casos base y almacenar sus respuestas, para finalmente construir el resto de estados con estos. Su concepto puede ser señalado como "Si ya tengo los cimientos, puedo construir lo que quiera", haciendo referencia a su forma de atravesar el DAG que se forma con los estados (estilo BFS multi origen).

Como el mismo nombre lo dice, se basará en iteraciones para lograr llenar la estructura que almacene las respuestas, siempre inicializando las soluciones a los casos base y casos triviales (no son exactamente base, pero sus respuestas se conocen, usualmente referido a los estados que son imposibles de obtener).

Capítulo 4

Problemas Típicos de Programación Dinámica

4.1. Coeficientes de expansión Binomial

Todos conocemos el famoso *Binomio de Newton*, el cual tiene la forma:

$$(x+y)^r = \sum_{k=0}^{\infty} \binom{r}{k} x^{r-k} y^k$$

Además, se da que

$$\binom{a}{b} = \frac{a!}{(a-b)!b!}$$

En estos momentos se nos presentarían 2 problemas particularmente:

1. La precisión al calcular los factoriales para luego dividirlos (Cuando solo deseamos $\binom{a}{b}$ como un solo elemento) sea de forma normal o modular (en cuyo caso necesitaremos hallar el inverso modular de cada número).
2. La complejidad añadida cuando queremos realizar el cálculo para hallar todos los coeficientes de la expresión del binomio.

Para cada una de las anteriores está un estilo diferente de DP, los cuales resuelven de manera eficiente estas problemáticas.
Recordemos la siguiente igualdad:

$$\binom{a}{b} = \binom{a-1}{b-1} + \binom{a-1}{b}$$

Donde se dan los casos base:

$$\binom{n}{0} = \binom{n}{n} = 1$$

Precisamente de acá sacaremos nuestras dos versiones de DP.
La versión Bottom-Up sería algo así para obtener todos los coeficientes $\binom{n}{i}$ con $i = 0, 1, \ldots, n$:

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

```
const int N = 10000+5;
8
int n;
long long memo[N][N];

void Coeficientes(int n){
        for(int i=0; i<=n; i++){
                for(int j=0; j<=i; j++){
                        if(j == 0 || j==i) memo[i][j] = 1; // Caso base
                        else memo[i][j] = memo[i-1][j-1] + memo[i-1][j]; // Recurrencia
                }
        }
}

int main(){
        cin >> n;
        Coeficientes(n);
        cout << "Coeficientes del binomio de grado " << n << ":" << endl;
        for(int i=0; i<=n; i++){
                cout << memo[n][i] << " ";
        }
        puts("");
        return 0;
}
```

Esta forma del DP es más completa que la versión Top Down dado que terminaría calculando todos los números combinatorios de la forma $\binom{a}{b}$, con $0 \leqslant a \leqslant n$ y $0 \leqslant b \leqslant a$.
Por otro lado, si lo que se desea es una complejidad más amortiguada, sería mejor usar la versión Top Down para hallar sólo un elemento. Tendrá la misma cota superior pero un mejor rendimiento en promedio.

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

const int N = 10000+5;

int a, b;
bool vis[N][N];
long long memo[N][N];

long long C(int n, int m){
        if(m == 0 || n==m) return memo[n][m] = 1;
        if(vis[n][m]) return memo[n][m];
        vis[n][m] = true; // Marcamos el estado como visitado
        return memo[n][m] = memo[n][n-m] = C(n-1,m-1) + C(n-1,m);
}

int main(){
        cin >> a >> b;
        cout << "Combinatoria de " << a << " en " << b << ": " << C(a,b) << endl;
        return 0;
}
```

Es sencillo notar que la complejidad de ambos métodos es $O(n^2)$, pero como se mencionó antes,

es cuestión de cada uno usar la versión que mejor se acople con lo que se necesita.

4.2. Números de Catalán

Los números de Catalán son una secuencia definida de la siguiente manera:

$$C_0 = 1$$

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$

Cada uno de estos números C_n son numéricamente iguales a:

- La cantidad de expresiones con n pares de paréntesis correctamente balanceadas.
- La cantidad de caminos monótonos que se forman para pasar de un punto a su opuesto diagonalmente de un cuadrado de $n \times n$ celdas tales que no crucen la diagonal que los une.

Podemos usar directamente tanto Top-Down como Bottom-Up. ¿Por qué? Pues es sencillo notar que en la sumatoria siempre se usarán elementos de posición estrictamente menor a n, por lo que si realizamos el cálculo de manera ascendente podremos realizar el DP de manera iterativa con la siguiente implementación:

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

const int N = 10000+5;

int n;
long long C[N];

void Catalan(int n){
        C[0] = 1; // Caso base
        for(int i=1; i<=n; i++){
                for(int j=0; j<=i-1; j++){
                    // Global, inicializado en 0 por lo que solamente sumamos cada aporte
                        C[i] += C[j]*C[i-1-j];
                }
        }
}

int main(){
        cin >> n;
        Catalan(n);
        cout << "Numero de Catalan (" << n << ") = " << C[n] << endl;
        return 0;
}
```

Es necesario recordar que esta serie crece de manera rápida, por lo que es común que en diferentes problemas nos pidan el resultado módulo un valor p.

4.3. Sucesión de Fibonnaci

Toda persona que haya estudiado secuencias matemáticas debería conocer la famosísima *Sucesión de Fibonacci*, que planteó Leonardo de Pisa como solución a un problema de cría de conejos. La sucesión está definida de la siguiente manera:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad 2 \leqslant i$$

Si planteamos la recursión simple, esto nos tomaría $O(2^n)$ para hallar el n-ésimo término de la serie; por ello, es que podemos usar las siguientes formas (Expresadas por $< f(n), g(n) >$ siendo la primera función la que expresa complejidad espacial y la segunda la algorítmica):

4.3.1. Versión Directa $< O(1), O(n) >$

La versión directa para hallar el n-ésimo término es usar solo 3 variables y modificar sus valores por iteración. Una implementación sería la siguiente:

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

int n;

long long F(int n){
        if(n == 1 or n == 2) return 1; // Casos base
        long long a = 1, b = 1, c; // a y b son los valores F_i-1 y F_i-2
        for(int i=3; i<=n; i++){ // Queremos hallar el n-esimo y ya sabemos los dos primeros
                c = a+b;
                a = b;
                b = c;
        }
        return c;
}

int main(){
        cin >> n;
        cout << "Numero de Fibonacci (" << n << ") = " << F(n) << endl;
        return 0;
}
```

Una de las desventajas para esta forma es que si nos piden los elementos por consultas, entonces cada consulta tomaría estrictamente $O(M)$ donde M sería el valor máximo posible que tome n en alguna de estas.

4.3.2. Versión Almacenada $< O(n), O(n) >$

La versión almacenada usa el principio de la directa para que en cada iteración se almacene el valor de la serie en un arreglo. Una implementación sería la siguiente:

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

const int N = 1000000+5;

int n;
bool vis[N];
long long F[N];

long long f(int n){
        if(n == 1 or n == 2) return 1;
        if(vis[n]) return F[n]; // Valor ya calculado, devolvemos respuesta
        vis[n] = true;
        return F[n] = f(n-1) + f(n-2);
}

int main(){
        cin >> n;
        cout << "Numero de Fibonacci (" << n << ") = " << f(n) << endl;
        return 0;
}
```

La principal ventaja a comparación del método anterior es que si nos dieran q consultas, el algoritmo finalmente las responde en un $O(n+q)$ amortizado para todas ellas.

4.3.3. Versión Almacenada Rápida $< O(\log n), O(\log^2 n) >$

La versión almacenada rápida usa la siguiente propiedad de la misma recursión:

$$F_n = (2F_{k-1} + F_k)F_k, \quad n = 2k, k \in \mathbb{Z}$$
$$F_n = F_k F_k + F_{k-1}F_{k-1}, \quad n = 2k-1, k \in \mathbb{Z}$$

Una implementación sería la siguiente:

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

const int N = 1000000+5;

int n;
unordered_map<int,bool> vis;
unordered_map<int,long long> F;

long long f(int n){
        if(n == 1 or n == 2) return 1; // Casos base
        if(vis[n]) return F[n]; // Valor ya calculado, devolvemos respuesta
        vis[n] = true;
        if(n&1){ // Si n es impar
                int k = (n+1)/2;
                return F[n] = f(k)*f(k) + f(k-1)*f(k-1);
        }
```

```
        else{
                int k = n/2;
                return F[n] = (2*f(k-1)+f(k))*f(k);
        }
}

int main(){
        cin >> n;
        cout << "Numero de Fibonacci (" << n << ") = " << f(n) << endl;
        return 0;
}
```

Es necesario recordar que esta serie crece de manera rápida, por lo que es común que en diferentes problemas nos pidan el resultado módulo un valor p. Por último, notemos que la complejidad espacial es $O(\log n)$ debido a que usamos la estructura *unordered_map*, la cual solo guardará los valores de n que se hayan visitado alguna vez y además mantiene una complejidad de acceso variable ($O(1)$ en el mejor de los casos y $O(|F|)$ en el peor, que en este caso es $O(\log n)$), por lo que la mejor complejidad temporal sería $O(\log n)$ y la peor $O(\log^2 n)$.

Otra manera para asegurar que la complejidad temporal sea $O(\log n)$ es mantener un arreglo de tamaño $O(n)$ para que el acceso sea en $O(1)$, al igual que si uno desea complejidad temporal $O(\log n \log\log n)$ y mantener el $O(\log n)$ de memoria, puede cambiar de estructura a *map* solamente.

4.4. Máxima Suma de Rango de 1 Dimensión (*Max 1D Range Sum*)

Una forma reducida del enunciado del problema 507 del UVa es la siguiente:

> Dado un arreglo A con $n \leqslant 20K = 20.10^3$ elementos diferentes de 0, determinar la máxima suma de rango de A.

Esto, en otras palabras, significa que debemos hallar el *Range Sum Query*(RSQ) entre dos índices i y j pertenecientes al intervalo $[0, n-1]$. Viéndolo:

$$Respuesta = max\left\{\sum_{k=i}^{j} A_k\right\} \quad \forall i, j = 0, 1, 2, \ldots, n-1$$

Un algoritmo de Búsqueda Completa (*Complete Search*) tomaría $O(n^2)$ para probar todos los pares $(i, j) \in [0, n-1] \times [0, n-1] \subset \mathbb{Z}^2$ junto con $O(n)$ para procesar el RSQ, con lo cual se ejecutaría con un tiempo de $O(n^3)$. Con $n \leqslant 20K$, esta solución daría TLE.

Una estrategia con DP sería pre-procesar el arreglo de forma que el valor de A'_k sea la suma de todos los valores anteriores junto con el del mismo A_k, podemos plantear el bucle siguiente:

```
A[i] += A[i-1], i=1,2,3,...,n-1
```

Ahora podremos procesar el RSQ en $O(1)$:

$$RSQ(i,j) = \begin{cases} A[j] & i = 0 \\ A[j] - A[i-1] & i > 0 \end{cases}$$

Con esto, el algoritmo de búsqueda completa procesa la respuesta en $O(n^2)$, sin embargo, con $n \leqslant 20K$ da igual TLE. A pesar de todo, esta técnica suele funcionar en otros casos. Para este problema hay incluso un algoritmo más adecuado: El algoritmo de Jay Kadane para el problema del máximo subarreglo. Este se ejecuta en $O(n)$ tomando una visión algo *greedy* con el DP, es de la forma siguiente:

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

const int N = 1000005;

int n;
int A[N];

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> A[i];
        int sum = 0, ans = INT_MIN; // Es importante la inicializacion de ans
        for(int i=0; i<n; i++){ // Procesamiento lineal
                sum += A[i]; // Tomamos, de forma greedy, la suma hasta este punto
                ans = max(ans,sum); // Maximizar el valor del RSQ
                if(sum < 0) sum = 0; // Si se torna negativo reiniciamos el valor de sum
        }
        cout << "Maximum 1D RSQ = " << ans << endl;
        return 0;
}
```

La idea principal de este algoritmo es tener una "suma de recorrido" y reiniciarla en 0 cada vez que se vuelva negativa (esto debido a que es mejor continuar una suma dejada en 0 en vez de continuarla desde un número negativo). Esta solución es adecuada para resolver este problema.

Notemos que este es un ejemplo básico de DP: En cada paso, tenemos dos opciones; podemos tomar el valor acumulado hasta antes de el actual o agregarle el actual, de todas formas se almacena el valor máximo en `ans`. De esta forma, si declaráramos la variable `dp(i)` como el `ans` perteneciente a cada i, sería el valor de la máxima RSQ de todos los posibles que terminen en A_i. Así, la respuesta sería el máximo de todos los `dp(i)` donde $i \in [0, n-1]$. Si se admite una respuesta con un subarreglo vacío (suma=0), entonces también debería considerarse como opción de respuesta.

4.5. Máxima Suma de Rango de 2 Dimensiones (*Max 2D Range Sum*)

Una forma reducida del enunciado del problema 108 del UVa es la siguiente:

> Dada una matriz A de dimensión $n \times n$ con elementos enteros dentro del rango $[-127, 127]$, se pide hallar una submatriz de A con la máxima suma de elementos si se da que $1 \leqslant n \leqslant 100$.

Si consideramos solucionar este problema con un algoritmo de búsqueda completa, nos encontraremos con la mala sorpresa de que no funcionará. El algoritmo tendría que ser esbozado con lo siguiente: Tendríamos que tomar todas los pares de filas y pares de columnas posibles para formar una submatriz, esto se daría en $O(n^4)$, además de $O(n^2)$ para sumar los elementos de la submatriz, lo que daría una ejecución en $O(n^6)$, con $n \leqslant 100$ esto da TLE.

La solución que usamos para la máxima suma de rango de 1 dimensión con preprocesamiento se puede extender a más dimensiones, siempre y cuando se aplique correctamente el principio inclusión-exclusión. Todo esto resultará en que el valor A'_{ij} almacenará la suma de los elementos de la submatriz con esquinas opuestas $(0,0)$ y (i,j), lo cual se procesa en $O(n^2)$. Podemos verlo de la siguiente manera:

$(i-1, j-1)$	$(i-1, j)$
$(i, j-1)$	(i, j)

Cuadro 4.1: Elementos de referencia con i y j

Es evidente que si consideramos los valores de A'_{ij} propuestos anteriormente, el proceso para obtener estos valores sería considerar agregar el $A'_{(i-1)j}$ y $A'_{i(j-1)}$ **si es que existen y están definidos** y restarle el $A'_{(i-1)(j-1)}$ **si es que existe y está definido** al mismo valor de A_{ij}.

Ahora, con la matriz acumulada, solo tenemos que elegir las filas (f_1, f_2); $f_1 \leqslant f_2$ y las columnas (c_1, c_2); $c_1 \leqslant c_2$ que delimiten la submatriz con la suma máxima. Para obtener los valores de f_1, f_2, c_1, c_2 haremos una búsqueda completa (entonces nos tomará $O(n^4)$) y para hallar el valor de la suma de los elementos de la matriz nos basta usar de nuevo el principio inclusión-exclusión para hallarlo en $O(1)$.

Suponiendo que queremos la suma de la submatriz con esquinas opuestas (f_1, c_1) y (f_2, c_2), entonces tomamos $A'_{f_2 c_2}$; sin embargo, en este valor están también los valores de $A'_{f_2(c_1-1)}$ y $A'_{(f_1-1)c_2}$: los restaremos si es que son valores válidos y definidos. Si volvemos a ver el valor que obtuvimos, nos daremos cuenta de que ahora falta el valor de $A'_{(f_1-1)(c_1-1)}$ que fue restado dos veces debido a lo anterior, así que lo sumamos si es un valor válido. La implementación quedaría así:

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

const int N = 105;

int n;
int A[N][N];

int main(){
        cin >> n;
        for(int i=0; i<n; i++){
                for(int j=0; j<n; j++){
                        cin >> A[i][j];
                        if(i > 0) A[i][j] += A[i-1][j]; // Agregamos el cuadro superior
                        if(j > 0) A[i][j] += A[i][j-1]; // Agregamos el cuadro del lado izquierdo
                        // Restamos el cuadro en la diagonal
                        if(i > 0 && j > 0) A[i][j] -= A[i-1][j-1];
                }
        }
        int maxSum = INT_MIN; // Inicializar con un valor contradictorio a la optimizacion
        for(int f1=0; f1<n; f1++){
```

```
                for(int f2=f1; f2<n; f2++){
                        for(int c1=0; c1<n; c1++){
                                for(int c2=c1; c2<n; c2++){
                                        int sumAct = A[f2][c2];
                                        if(f1 > 0) sumAct -= A[f1-1][c2];
                                        if(c1 > 0) sumAct -= A[f2][c1-1];
                                        if(f1 > 0 && c1 > 0) sumAct += A[f1-1][c1-1];
                                        maxSum = max(maxSum,sumAct); // Maximizar el valor
                                }
                        }
                }
        }
        cout << "Maximum 2D RSQ = " << maxSum << endl;
        return 0;
}
```

Con estos dos ejemplos anteriores (Máxima Suma de Rango de 1D y 2D) podemos ver que no todos los problemas de RSQ se resuelven necesariamente con Segment Tree o Fenwick Tree (BIT), en todo caso, cuando el query es de máximos o mínimos es favorable aplicar DP.

4.6. Corte de Barra (*Rod Cutting*)

El enunciado de este simple problema es el siguiente:

> Dada una vara de longitud l y una tabla de precios p_i que expresan el pago por una de longitud i. Halle la mayor ganancia posible luego de cortarla adecuadamente en piezas de longitud entera (no efectuar cortes en absoluto también es válido).

Un enfoque con *Complete Search* tendría que analizar todas las configuraciones posibles para cortar; como hay 2^{l-1} puntos de corte posible, su complejidad sería $O(2^{l-1})$. Como esto es muy ineficiente, plantearemos el DP:
Definamos que una solución óptima para una longitud m consta de k cortes:

$$m = i_1 + i_2 + \cdots + i_k$$

En cuyo caso, la ganancia máxima es:

$$r_m = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Lo que implica que si cortáramos la parte i_1 de m, entonces la distribución toma la siguiente forma:

$$r_m = p_{i_1} + (p_{i_2} + \cdots + p_{i_k})$$

Pero al ser la solución óptima, entonces la distribución restante también debe ser óptima al respecto, entonces

$$r_m = p_{i_1} + (p_{i_2} + \cdots + p_{i_k}) = \text{máx}\,\{p_{i_1} + r_{m-i_1}\}$$

Y claramente podemos establecer lo mismo para r_{m-i_1} y además i_1 es un valor que tomaremos como variable para evaluar cada posible distribución óptima.

$$r_l = \text{máx}\,\{p_m + r_{l-m}\}$$

Claramente, los casos base son

$$r_0 = 0 \quad r_1 = p_1$$

Y los restantes ya se construyen en función a la recursión.
Una implementación de este enfoque sería la siguiente:

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

const int N = 10005;

int n, l;
bool vis[N];
int p[N], memo[N];

int RodCut(int len){
        if(len == 0) return 0; // l = 0 -> p = 0
        if(vis[len]) return memo[len]; // Respuesta ya calculada
        int ans = 0; // Inicializar con un valor contradictorio a la optimizacion
        for(int i=1; i<=len; i++)
                ans = max(ans, p[i] + RodCut(len-i)); // Maximizar el valor
        vis[len] = true;
        return memo[len] = ans;
}

int main(){
        p[0] = 0;
        cin >> n;
        for(int i=1; i<=n; i++) cin >> p[i];
        cin >> l;
        cout << "Rod Cutting = " << RodCut(l) << endl;
        return 0;
}
```

Este algoritmo tiene complejidad $O(l^2)$, la cual es mucho más eficiente que $O(2^{l-1})$.

4.7. Multiplicación de Cadenas de Matrices

Al efectuar una multiplicación de una secuencia de matrices tenemos que la complejidad en general es de $O(mnp)$ para dos matrices de dimensiones $m \times n$ y $n \times p$. Como en general esta complejidad es en sí un problema, debemos optimizar todo lo que podamos la complejidad de toda la secuencia. Para ello aprovecharemos la propiedad asociativa del producto de matrices, entonces:

> Dada una secuencia de n matrices A_i con sus dimensiones dadas por una secuencia de tamaño $(n+1)$ que deben ser multiplicadas, halle la mínima cantidad de operaciones elementales que se pueden realizar añadiendo convenientemente paréntesis a la operación (No añadir ninguno también es válido).

Supongamos que la secuencia que almacena las dimensiones matrices A_i es p_i, en cuyo caso las dimensiones de la matriz A_i serían de $p_1 \times p_{i+1}$, entonces notamos claramente que se pueden añadir los paréntesis con respecto a la cantidad de matrices k que se tome de las n totales y formar la siguiente recurrencia:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n \leqslant 2 \end{cases}$$

Esta recurrencia es muy parecida a la de los números de Catalán, los cuales tienen un crecimiento asint

otico de $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$, lo cual es una cantidad demasiado grande como para analizar cada uno de sus casos. Veremos la resolución con DP:
En primer lugar, definimos nuestro estado como los límites de la expresión que dividiremos si es que es necesario, dado que si tenemos la secuencia

$$A_1, A_2, \ldots, A_n$$

Entonces, la solución óptima se da colocando en una posición k (entre las matrices A_{k-1} y A_k) una división y formando la solución óptima de estas dos divisiones. Mediante esa recurrencia se obtendrá la respuesta.
En general, definimos la función $mcm(i,j)$ (por *Matrix Chain Multiplication*), que expresa la cantidad de operaciones que se necesitan para multiplicar la secuencia $A_i, \ldots, A_j$. Es evidente que $mcm(i,i) = 0$, mientras que por otra parte, supongamos un valor k que determine una división de la secuencia $A_i, \ldots, A_j$, es decir:

$$(A_i, A_{i+1}, \ldots, A_{k-1}), (A_k, \ldots, A_j)$$

Entonces la cantidad de operaciones necesarias para este caso serían

$$mcm(i, k-1) + mcm(k, j) + p_i \times p_k \times p_{j+1}$$

Y si queremos la división óptima, deberíamos hallar el k tal que

$$mcm(i,k-1) + mcm(k,j) + p_i \times p_k \times p_{j+1} = \min_{x \in [i+1,j]} \{mcm(i, x-1) + mcm(x, j) + p_i \times p_x \times p_{j+1}\}$$

Formamos la recursión que usaremos de esta forma:

$$mcm(i,j) = \begin{cases} 0 & i = j \\ \min\limits_{k \in [i+1,j]} \{mcm(i,k-1) + mcm(k,j) + p_i \times p_k \times p_{j+1}\} & i \neq j \end{cases}$$

Una implementación sería la siguiente:

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

const int N = 505;
const int inf = 1<<20; // Valor "infinito" pero evitando overflow

int n;
bool vis[N][N];
int p[N], MCM[N][N];

int mcm(int i, int j){
        if(i == j) return 0; // Una sola matriz. Costo 0
        if(vis[i][j]) return MCM[i][j]; // Respuesta ya calculada
        int ans = inf; // Inicializar con un valor contradictorio a la optimizacion
        for(int k=i+1; k<=j; k++)
                ans = min(ans,mcm(i,k-1) + mcm(k,j) + p[i]*p[k]*p[j+1]); // Minimizar el valor
        vis[i][j] = true;
        return MCM[i][j] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> n;
        for(int i=0; i<=n; i++) cin >> p[i];
        cout << "Matrix Chain Multiplication = " << mcm(0,n-1) << endl;
        return 0;
}
```

Este algoritmo necesita $O(n^2)$ para cada estado posible y además en cada vez que vamos a desarrollar un estado también iteramos por un intervalo de tamaño $O(n)$. La complejidad de este algoritmo será entonces de $O(n^3)$.

4.8. Subsecuencia Común Más Larga (*Longest Common Subsequence* LCS)

Por lo general en las computadoras se suele usar la función *diff*, para comparar dos textos y verificar si son iguales o no. En este caso particular, el problema para nuestro caso es el siguiente:

> Dadas dos cadenas S_1 y S_2, hallar la máxima longitud de las subsecuencias de caracteres comunes entre ambas

Recordemos que una subsecuencia no son elementos necesariamente consecutivos.
Como ejemplo, tomemos las palabras **carro** y **arcilla**.
Particularmente, existen dos subsecuencias con la longitud más larga posible: *ar* y *ca*. Se nota claramente que Por lo tanto, la respuesta para este ejemplo sería 2.

c	**a**	r	r	o		
a	r	**c**	i	l	l	**a**

c	**a**	**r**	r	o		
a	**r**	c	i	l	l	a

El enfoque de *Complete Search* generaría todas las subsecuencias posibles de ambas y las compararia una a una, por lo que suponiendo que la cadena S_1 tiene longitud m y la S_2 tiene longitud n, entonces, la complejidad total del enfoque se calcularía mediante la suma de las complejidades de los siguientes pasos principales:

$$T(n) = T(\text{Generar subsecuencias}) + T(\text{Comparación})$$

Generar las subsecuencias de cada uno toma una complejidad de $O(2^{longitud})$, que significaría que

$$T(\text{Generar subsecuencias}) = O(2^n) + O(2^m) = O(2^{\text{máx}\{m,n\}})$$

Además la comparación sería el producto de las dos cantidades de subsecuencias, lo que daría que

$$T(\text{Comparación}) = O(n2^n) \times O(m2^m) = O(nm2^{m+n})$$

Ciertamente esto es muy ineficiente, por lo que plantearemos un DP para optimizar la complejidad del proceso.
Definamos la función $LCS(i, j)$ como el tamaño de la LCS entre las cadenas $S_1[0, 1, \ldots, i]$ y $S_2[0, 1, \ldots, j]$. En este caso, la respuesta que deseamos se daría por $LCS(m-1, n-1)$ y formamos la siguiente recurrencia.
Al ser una subsecuencia, lo importante son los caracteres que la forman y no la posición que tengan dentro de su propia cadena, por lo que si tenemos la palabra *carro*, su secuencia $[0, 1, 2]$ genera lo mismo que su secuencia $[0, 1, 3]$ y esa seguiría siendo su LCS con la cadena *carencia*. Tomando esto en cuenta, podríamos contar los caracteres en común barriendo desde un lado a otro verificando solo que los caracteres coincidan, así que notamos que

$$LCS(i,j) = \begin{cases} 1 + LCS(i-1, j-1) & S_1[i] == S_2[j] \\ \text{máx}\{LCS(i-1,j), LCS(i,j-1)\} & S_1[i] \neq S_2[j] \end{cases}$$

Este algoritmo como mucho tendría que recorrer toda la cadena S_2 por cada caracter de la cadena S_1, lo que le da una complejidad de $O(mn)$. Una implementación sería la siguiente:

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

const int N = 10005;

int n, m;
string x, y;
bool vis[N][N];
int LoCS[N][N];

int LCS(int i, int j){
        if(i < 0 || j < 0) return 0; // Comparando con cadena vacia
        if(vis[i][j]) return LoCS[i][j]; // Respuesta ya calculada
        vis[i][j] = true;
        if(x[i] == y[j]) return LoCS[i][j] = 1+LCS(i-1,j-1); // Coincidencia
        else return LoCS[i][j] = max(LCS(i-1,j),LCS(i,j-1)); // Maximizar el valor
}

int main(){
        cin >> x >> y;
        n = x.size();
        m = y.size();
        cout << "Longest Common Subsequence = " << LCS(n-1,m-1) << endl;
        return 0;
}
```

4.9. Subsecuencia Creciente Más Larga (*Longest Increasing Subsequence* LIS)

Antes que nada, definamos qué es una **LIS**:

> Dada la secuencia $A = \{A_0, A_1, \ldots, A_{n-1}\}$, determinar la máxima longitud de la subsecuencia $L = \{A_{i_1}, A_{i_2}, \ldots, A_{i_m}\}$ tal que $i_k < i_{k+1}$ y $A_{i_k} < A_{i_{k+1}}$.

Por ejemplo, si tuviéramos

$$A = \{10, 22, 9, 33, 21, 50, 41, 60, 80\}$$

Entonces la subsecuencia creciente más larga sería

$$L = \{10, 22, 33, 50, 60, 80\}$$

Y L es de longitud 6.
La solución usando *Complete Search* consistiría en generar todas las posibles subsecuencias mediante *Backtracking*, lo que tendría de complejidad $O(2^n)$. Esta solución es muy poco eficiente, por lo que aplicaremos DP:
Definamos un arreglo LIS tal que $LIS(i)$ sea la longitud de la LIS que termina en el índice i, ahora tenemos la estructura óptima definida.
Ya que planteamos la estructura, hallemos una recurrencia entre los $LIS(i)$:

1. $LIS(0) = 1$, dado que es un único elemento de referencia
2. Para hallar $LIS(i)$, debemos hallar antes una subsecuencia dentro de todos los elementos anteriores tal que su último elemento sea menor que A_i y además, en el caso de que se

añada A_i, la longitud de esta nueva subsecuencia sea la máxima posible, esto se obtiene estableciendo las siguientes condiciones:

$$LIS(i) = \max\{1 + LIS(j)\}, \forall j < i \ \wedge \ A_j < A_i$$

La respuesta será el valor de $LIS(k)$ máximo dentro del arreglo LIS. Esta solución tiene complejidad $O(n^2)$.
Una implementación que da como respuesta sólo la longitud de la LIS es la siguiente:

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

const int N = 10005;

int n;
int A[N];
int LIS[N];

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> A[i];
        LIS[0] = 1; // Trivial
        int LonLIS = 1; // Valor minimo posible
        for(int i=1; i<n; i++){
                LIS[i] = 1; // Trivial
                for(int j=0; j<i; j++){
                        if(A[j] < A[i]) // Candidato
                                LIS[i] = max(LIS[i],1+LIS[j]); // Maximizar el valor actual
                }
                LonLIS = max(LonLIS,LIS[i]); // Maximizar el valor total
        }
        cout << "Longest Increasing Subsequence = " << LonLIS << endl;
        return 0;
}
```

Además también hay una implementación de LIS con complejidad $O(nlogn)$ usando *Binary Search* pero no la veremos acá.

4.10. Problema de la Mochila (*Knapsack Problem*)

El enunciado del clásico problema de la mochila es el siguiente:

> Se tiene un conjunto de n objetos con sus respectivos valores V_i y pesos W_i. Dada una capacidad máxima S de una mochila, debemos hallar el máximo valor posible de objetos que podemos cargar simultáneamente

Matemáticamente, se plantea que

> Dado un valor L finito y un conjunto A de pares $A_i = (v_i, w_i)$, hallar valor M tal que $M =$

máx $\sum_{i=0}^{m} v_{j_i}$ sea máximo y además que $\sum_{i=0}^{m} w_{j_i} \leqslant L$

El enfoque de *Complete Search* para este problema se da con backtracking recursivo o usando bitmask si es que $n \leqslant 30$, pero su algoritmo nos da una complejidad de $O(2^n)$, por lo que no es eficiente. La solución con DP define el siguiente proceso.
Muy similar a la recursión por backtracking, definimos una función $knap(pos, cap)$, donde pos es la posición del elemento que consideraremos y cap es lo que resta antes de que la mochila esté llena, entonces notamos que la respuesta que deseamos será calculada por $knap(n, S)$ (asumiendo que el elemento de posición n de los arreglos tienen valor 0). Su comportamiento es el siguiente:

$$knap(pos, cap) = \begin{cases} knap(pos-1, cap) & w_{pos} > cap \\ \max\{knap(pos-1, cap), v_{pos} + knap(pos-1, cap - w_{pos})\} & w_{pos} \leqslant cap \end{cases}$$

Además, los casos base son cuando $pos < 0$ y cuando $w = 0$, en cuyas situaciones no se puede aumentar más la ganancia y para sus estados deberíamos devolver un 0 como posible ganancia. Como este algoritmo recorre, como mucho, todos los posibles estados, entonces toma una complejidad de $O(nS)$, que es mucho mejor que $O(2^n)$. Su implementación es la siguiente:

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

const int N = 10005;

int n, S;
bool vis[N][N];
int KS[N][N];
int v[N], w[N];

int knap(int pos, int capacidad){
        if(pos == -1 || capacidad == 0) return 0; // No hay elemento o espacio restante
        if(vis[pos][capacidad]) return KS[pos][capacidad]; // Respuesta ya calculada
        int ans;
        if(w[pos] > capacidad) ans = knap(pos-1,capacidad); // Es imposible tomar el elemento
        else ans = max(knap(pos-1,capacidad),knap(pos-1,capacidad-w[pos]) + v[pos]); // Maximizar
        vis[pos][capacidad] = true;
        return KS[pos][capacidad] = ans;
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> v[i] >> w[i];
        cin >> S;
        cout << "Knapsack Problem = " << knap(n,S) << endl;
        return 0;
}
```

4.11. Cambio de Monedas (Versión General)

El enunciado de este problema es el siguiente:

Dada una serie de n valores de moneda (un arreglo M), determinar la mínima cantidad de monedas necesarias para expresar la cantidad V

Claramente un enfoque con *Complete Search* da una complejidad presumiblemente mayor a $O(2^n)$, por lo que ni siquiera se nos debería pasar por la cabeza realizar algo así. El enfoque con DP plantea lo siguiente (muy similar al *Rod Cutting*):

1. $cambio(0) = 0$, pues se usan 0 monedas
2. $cambio(< 0) = \infty$ pues es un valor que no se puede expresar. Como queremos minimizar la cantidad, entonces devolvemos un valor muy grande para que este caso sea ignorado.
3. $cambio(i)$ se halla minimizando la cantidad de monedas total, sea su $cambio(i) = k$ con una distribución óptima de
$$i = M_{i_1} + M_{i_2} + \cdots + M_{i_k}$$
Lo que implica que
$$i - M_{i_1} = M_{i_2} + \cdots + M_{i_k}$$
Por lo que $cambio(i - M_{i_j}) = k - 1$ con $1 \leqslant j \leqslant k$, y como queremos minimizar esta cantidad, debemos probar con cada uno de los valores posibles para M_{i_j} (iterar por todos los elementos de M) para hallar la solución. Entonces, el valor mínimo de esta cantidad toma la forma de
$$cambio(i) = \text{mín}\{1 + cambio(i - M_j)\}$$
Claramente es el minimo de todos los $0 \leqslant j < n$.

Una implementación de este planteamiento es la siguiente:

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

const int N = 105;
const int K = 1000005;
const int inf = 1<<20; // Valor "infinito" pero evitando overflow

int n, V;
bool vis[K];
int v[N], memo[K];

int cambio(int valor){
        if(valor == 0) return 0; // No se usan monedas para el valor 0
        if(valor < 0) return inf; // Caso absurdo
        if(vis[valor]) return memo[valor]; // Respuesta ya calculada
        int ans = inf; // Inicializar con un valor contradictorio a la optimizacion
        for(int i=0; i<n; i++){
                ans = min(ans,1 + cambio(valor-v[i])); // Minimizar el valor
        }
        vis[valor] = true;
        return memo[valor] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> v[i];
```

```
        cin >> V;
        cout << "Coin Change = " << cambio(V) << endl;
        return 0;
}
```

La complejidad de este algoritmo es $O(nV)$ dado que se itera por los n valores desde la suma inicial V. Notemos cómo usamos una variable inf y no INT_MAX debido a que este valor puede retornarse como respuesta, por lo que generaría un overflow al sumársele 1. Para los valores "infinitos" que no operen con algún valor sí es seguro usar los valores límites.

4.12. Problema del Cartero (*Travelling Salesman Problem*)

El clásico problema del cartero presenta el siguiente enunciado:

> Dadas n ciudades y sus distancias entre sí en una matriz d de $n \times n$, hallar la distancia mínima posible de alguna ruta que pase por cada una de las otras $n-1$ ciudades **una sola vez** y termine en la ciudad en que se inició.

Notamos que nos piden hallar el ciclo con menor distancia total posible que recorra solo una vez las $(n-1)$ ciudades del interior de este; sin embargo, si fijamos como punto inicial la ciudad 1, tendríamos $(n-1)!$ posibles ciclos que cumplan con la condición. Si usáramos *Complete Search* con esta complejidad, el valor de $n = 11$ podría dar un TLE, por lo que esta solución no es lo suficientemente rápida.
El enfoque con DP se basa en la recursión que plantearíamos con *Complete Search* pero usando almacenamiento y minimización de la respuesta por cada subproblema. Es evidente que si tomáramos las ciudades

$$A - B - C - (n-3)\,\text{ciudades restantes}$$

como ciclo, entonces otro candidato sería

$$A - C - B - (n-3)\,\text{mismas ciudades}$$

Por lo que claramente hay partes de ciclo que se sobrelapan. Además para saber qué ciudades ya han sido visitadas usaremos un *bitmask*, de tal forma que si el n-ésimo bit está prendido, entonces ya se ha visitado la ciudad n.
La recursión que se usaría en *Complete Search* sería algo como

$$TSP(pos, bitmask) = \begin{cases} d[pos][0] & bitmask = 2^n - 1 \\ min(d[pos][next] + TSP(next, bitmask|(1 << next))) & bitmask \neq 2^n - 1 \end{cases}$$

Está de más señalar que la ciudad $next$ no debió haber sido visitada antes ($bitmask\&(1 << next)) = 0$) y que sea diferente de pos.
Usando esta recursión de base y una matriz de almacenamiento *memo*, realizamos el enfoque con DP de este problema. Una implementación sería la siguiente:

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

const int N = 20;

int n;
int d[N][N];
bool vis[N][(1<<N)+2];
```

```
int memo[N][(1<<N)+2];

int TSP(int pos, int bitmask){
        if(bitmask+1 == (int)(1<<n)) return d[pos][0]; // Todos los nodos fueron visitados
        if(vis[pos][bitmask]) return memo[pos][bitmask]; // Respuesta ya calculada
        int ans = INT_MAX; // Inicializar un valor contradictorio a la optimizacion
        for(int next = 0; next < n; next++){
                if(next!=pos && int(bitmask&(int)(1<<next))==0)
                        ans = min(ans,d[pos][next] + TSP(next,bitmask|(1<<next))); // Minizamos
        }
        vis[pos][bitmask] = true;
        return memo[pos][bitmask] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++)
                for(int j=0; j<n; j++) cin >> d[i][j];
        cout << "Travelling Salesman = " << TSP(0,1) << endl; // Ciudad inicial: 1
        return 0;
}
```

Este algoritmo repasa como mucho cada uno de los estados posibles del DP a partir de un punto, por la complejidad de la cantidad de estados es a lo mucho $O(n2^n)$ por punto, y en cada punto se itera a través de todas las n ciudades, dándole una complejidad de $O(n^2 2^n)$ lo cual funciona hasta el caso $n = 16$.

4.13. Alineamiento de Cadenas (*Edit Distance*)

El problema planteado por Edit Distance tiene el siguiente enunciado:

> Dadas 3 acciones posibles (insertar, reemplazar o remover un caracter), hallar la mínima cantidad de acciones requeridas para transformar la cadena S_1 a la cadena S_2.

Dado que las únicas acciones posibles para efectuar sobre la cadena S_1 son 3, entonces el enfoque con *Complete Search* generaría un *Backtracking* con complejidad $O(3^{lon(S_2)})$, lo cual no es nada eficiente. Por este motivo es que analizaremos el problema con DP:
Dadas las posiciones $S_1[i]$ y $S_2[i]$, solo hay 4 acciones posibles en total:

1. $S_1[i] = S_2[i]$, y no hacemos nada porque ya coinciden.
2. $S_1[i] \neq S_2[i]$ y reemplazamos $S_1[i]$ con $S_2[i]$.
3. Insertamos un caracter en $S_1[i]$ desplazando el resto de la cadena hacia la derecha.
4. Borramos el caracter $S_1[i]$.

Definamos ahora la estructura $editD(i, j)$, la cual nos da el valor óptimo para transformar la subcadena $S_1[0, 1, \ldots, i]$ a $S_2[0, 1, \ldots, j]$. La respuesta total sería $editD(n-1, m-1)$ asumiendo que S_1 tiene n caracteres y S_2 tiene m.
Veamos los casos base:

$$editD(i, -1) = i + 1 \quad editD(-1, j) = j + 1$$

Dado que la única forma de transformar una cadena vacía en otra no vacía sería insertar todos sus caracteres. En el caso particular de que sean ambas vacías, la respuesta sería 0, por lo que

$$editD(-1,-1) = 0$$

En el caso de que $i > 0 \wedge j > 0$, se da que

$$editD(i,j) = \begin{cases} editD(i-1,j-1) & S_1[i] = S_2[j] \\ 1 + \min\{editD(i,j-1), editD(i-1,j), editD(i-1,j-1)\} & S_1[i] \neq S_2[j] \end{cases}$$

En este caso, $editD(i,j-1)$ expresa el valor óptimo al insertar el caracter $S_2[j]$, $editD(i-1,j)$ expresa el valor óptimo al borrar el caracter $S_1[i]$ y $editD(i-1,j-1)$ expresa el valor óptimo al reemplazar el caracter $S_1[i]$ por el caracter $S_2[j]$. Una implementación sería la siguiente:

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

const int N = 10005;

int n, m;
string x, y;
bool vis[N][N];
int ED[N][N];

int min(int a, int b, int c){ return min(a,min(b,c));}

int editD(int i, int j){
        if(i == -1 && j == -1) return 0; // Dos cadenas vacias dan 0
        if(i == -1 || j == -1) return i+j+2; // Solo queda insertar lo restante
        if(vis[i][j]) return ED[i][j];
        int ans;
        if(x[i] == y[j]) ans = editD(i-1,j-1); // Son iguales, no es necesario nigun cambio
        else ans = 1 + min(editD(i,j-1),editD(i-1,j),editD(i-1,j-1)); // Minimizar el valor
        vis[i][j] = true;
        return ED[i][j] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> x >> y;
        n = x.size();
        m = y.size();
        cout << "Edit Distance = " << editD(n-1,m-1) << endl;
        return 0;
}
```

Es sencillo ver que este algoritmo tiene complejidad $O(nm)$ en el caso que se visiten todos los estados posibles. Notemos que en el caso

Capítulo 5

Problemas No Típicos de Programación Dinámica

5.1. *Word Wrap Problem*

El enunciado de este problema (muy usado por editores de texto) es el siguiente:

> Dada una anchura de línea de m caracteres y n palabras, considerando que el costo por línea es el cubo de la cantidad de espacios en blanco al final de ella (sin contar los de la última línea), hallar el mínimo costo posible al distribuir las n palabras en una cantidad arbitraria de líneas.

El planteamiento de este problema consiste en 2 partes, para lo cual se realizarán 2 preprocesamientos de apoyo y un procesamiento final para hallar la respuesta. En realidad, al analizar detalladamente el enunciado, solamente nos importa la secuencia de líneas en las que colocaremos las palabras que ya tenemos, pues no podemos cambiar su orden de aparición.

Entonces, planteamos la función $costo(i)$ que nos dará el costo óptimo de colocar las palabras en el rango $[1, i]$ en determinadas líneas. Por lo tanto, debemos verificar todos los posibles escenarios en los cuales podemos agregar la i-ésima palabra a alguna línea. Así que si tuviéramos la función $costo\_linea(i, j)$ que nos da el costo de colocar las palabras del rango $[i, j]$ en una sola línea (el cual será infinito si es que la anchura no se abastece para ponerlas), tendríamos la recursión:

$$costo(0) = 0$$
$$costo(j) = \min_{1 \leqslant i \leqslant j} \{costo(i-1) + costo\_linea(i, j)\}$$

Ahora, para obtener el valor de la función $costo\_linea(i, j)$ es sencillo notar que su cantidad de espacios libres es $\left(m - j - i - \sum_{k=i}^{j} v_k\right)$, por lo que solo basta con elevarlos al cubo si es que es una cantidad no negativa y si fuese negativa, volverla ∞.

Una implementación en la cual la primera parte halla los espacios libres, la segunda el costo por línea y la última el costo total óptimo sería la siguiente:

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

const int N = 10000+5;
const long long inf = 1LL<<42; // Representacion de infinito, 2^42

int n,m;
int v[N];
int espacios[N][N];
long long costo_linea[N][N];
long long costo[N];

long long f(int x){
        return 1LL*x*x*x; // A veces se usa el cuadrado
}


int main(){
        cin >> n >> m;
        for(int i=1; i<=n; i++) cin >> v[i];
        for(int i=1; i<=n; i++){
                espacios[i][i] = m - v[i]; // Caso base: solo esta palabra
                for(int j=i+1; j<=n; j++){
                        espacios[i][j] = espacios[i][j-1] - v[j] - 1; // Agregamos
                }
        }
        for(int i=1; i<=n; i++){
                for(int j=i; j<=n; j++){
                        if(espacios[i][j] < 0){ // Faltan espacios, asi que no es valido
                                costo_linea[i][j] = inf;
                        }
                        else if(j == n and espacios[i][j] >= 0){
                            // Es ultima linea y sobran espacios,
                            // por condicion del problema no se consideran
                                costo_linea[i][j] = 0;
                        }
                        else{
                            // Asignamos el costo de manera directa
                                costo_linea[i][j] = f(espacios[i][j]);
                        }
                }
        }
        costo[0] = 0;
        for(int j=1; j<=n; j++){
                costo[j] = inf;
                for(int i=1; i<=j; i++){
                        if(costo[i-1]!=inf and costo_linea[i][j]!=inf){ // Candidato valido
                            // Minimizamos el costo hasta la posicion actual
                                costo[j] = min(costo[j],costo[i-1]+costo_linea[i][j]);
                        }
                }
        }
        cout << "El costo minimo es " << costo[n] << endl;
        return 0;
}
```

Este algoritmo tiene una complejidad de $O(n^2)$

5.2. Subsecuencia Palíndroma Más Larga (*Longest Palindromic Subsequence*)

El enunciado de este problema es el siguiente:

> Dada una cadena de caracteres S, hallar la longitud máxima posible para alguna subsequencia de caracteres $L = S_{i_1} S_{i_2} \dots$ tal que L sea palíndroma.

Recordemos que la definición de *caracter* incluye todo tipo de símbolos, por lo que esta solución también es aplicable a secuencias de números.
Un ejemplo sería para la cadena:

$$S = \text{“}BBABCBCAB''$$

Cuya LPS sería:

$$LPS(S) = \text{“}BABCBAB''$$

Por lo que la respuesta en este caso es 7.
Es facil notar que este problema tiene un cierto parecido al de LCS; dado que si encontramos una coincidencia, aumentamos un caracter a la LPS, pero si no lo hacemos, probamos disminuyendo a un lado o disminuyendo en el otro.
Suena un poco confuso al inicio, por lo que definiremos $LPS(i, j)$ como el valor de la LPS entre S_i y S_j, ahora tenemos que formar una recurrencia:

$$LPS(i,j) = \begin{cases} 1 & i = j \\ 2 & j = i+1 \wedge S_i = S_j \\ 0 & j = i+1 \wedge S_i \neq S_j \\ 2 + LPS(i+1, j-1) & S_i = S_j \\ \max\{LPS(i+1,j), LPS(i,j-1)\} & S_i \neq S_j \end{cases}$$

Para evitar el hecho de que $j < i$, hemos definido casos base como cuando $i = j$ (un caracter solo ya es una subsecuencia palíndroma) o $j = i + 1$ (con solo dos caracteres tenemos dos opciones: que sean iguales o diferentes. En el primer caso se aumentan a la subsecuencia; si no, no se aumenta nada).
Luego los subproblemas son como señalamos anteriormente; si hay una coincidencia, se aumentan a la LPS y continuamos en lo restante, mientras que si no hay algunda, se toma el máximo entre avanzar el primer índice o retroceder el segundo. Una implementación sería la siguiente:

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

const int N = 10005;

int n;
string x;
bool vis[N][N];
int LoPS[N][N];

int LPS(int i, int j){
        if(i == j) return 1; // Caso base 1: Un solo caracter
        if(i+1 == j){
```

```
                if(x[i] == x[j]) return LoPS[i][j] = 2; // Caso base 2: Dos caracteres iguales
        }
        if(vis[i][j]) return LoPS[i][j]; // Respuesta ya calculada
        vis[i][j] = true;
        if(x[i] == x[j]) return LoPS[i][j] = 2 + LPS(i+1,j-1); // Coincidencia
        else return LoPS[i][j] = max(LPS(i+1,j),LPS(i,j-1)); // Maximizar el valor
}

int main(){
        cin >> x;
        n = x.size();
        cout << "Longest Palindromic Subsequence = " << LPS(0,n-1) << endl;
        return 0;
}
```

Este algoritmo tiene de complejidad $O(n^2)$, además existe otra forma de resolverlo en $O(n^2)$ con sólo usar LCS para la cadena S y la cadena $\overline{S}$ (cadena revertida). Una implementación es la siguiente (usando nuestra implementación anterior de LCS):

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

const int N = 10005;

int n, m;
string x, y;
bool vis[N][N];
int LoCS[N][N];

int LCS(int i, int j){
        if(i < 0 || j < 0) return 0; // Comparando con cadena vacia
        if(vis[i][j]) return LoCS[i][j]; // Respuesta ya calculada
        vis[i][j] = true;
        if(x[i] == y[j]) return LoCS[i][j] = 1 + LCS(i-1,j-1); // Coincidencia
        else return LoCS[i][j] = max(LCS(i-1,j),LCS(i,j-1)); // Maximizar el valor
}

int main(){
        cin >> x;
        n = x.size();
        y = x;
        reverse(y.begin(),y.end());
        m = y.size();
        cout << "Longest Palindromic Subsequence = " << LCS(n-1,m-1) << endl;
        return 0;
}
```

Dado que no hay exactamente alguna optimización en la complejidad, les brindamos ambas formas.

5.3. LIS con Binary Search

Ya hemos visto anteriormente que se puede hallar la LIS de una secuencia de números en $O(n^2)$; sin embargo, existe una solución más rápida que usa *Binary Search* y la misma condición de LIS para hallarla en $O(nlogn)$.
Ya sabemos el enunciado de la LIS, así que pasaremos directo al método:
Primero tenemos que notar que si tenemos la LIS de una parte de la secuencia v_i, sea LIS_{jn} la LIS desde el elemento j al n de la secuencia, entonces la LIS de toda la secuencia se da uniendo la LIS_{jn} y la LIS_{1j} tal que el elemento final de esta última sea menor que el elemento inicial de la primera. Esta condición nos lleva a pensar en comprimir lo más que se pueda los valores de la LIS_{1j} cada vez que pasamos por cada elemento. En tal caso, cuando encontremos un elemento nuevo, lo deberíamos poner en la posición que ocuparía si es que lo consideráramos necesariamente en la LIS (a pesar de que al final no sea así).

$$v = \{6, 5, 4, 3, 2, 1, 7\}$$

Nuestra idea llevada a la práctica haría algo como esto:

LIS_{11}	=	$\{6\}$	Trivial
LIS_{12}	=	$\{5\}$	Es menor, reemplaza
LIS_{13}	=	$\{4\}$	Es menor, reemplaza
LIS_{14}	=	$\{3\}$	Es menor, reemplaza
LIS_{15}	=	$\{2\}$	Es menor, reemplaza
LIS_{16}	=	$\{1\}$	Es menor, reemplaza
LIS_{17}	=	$\{1, 7\}$	Es mayor, entra al final

Lo interesante y beneficioso del método, es que prueba todas las LIS posibles reescribiendo cada una de ellas; por lo que si se guarda la longitud máxima en una variable durante cada iteración, se obtendrá el valor de la LIS total.
Ahora viene la pregunta más importante: ¿Por qué es $O(nlogn)$? La respuesta es sencilla. Dado que buscaremos el lugar ideal en el cual poner el elemento nuevo que estamos evaluando, aprovecharemos el hecho de que lo que estamos guardando en cada iteración es una LIS, por lo que podemos usar *Binary Search* para hallar el elemento más grande dentro de la LIS almacenada que sea menor que el que estamos evaluando.
Daremos un ejemplo con mejor explicación:

$$v = \{2, 5, 3, 7, 11, 8, 10, 13, 6\}$$

LIS_{11}	=	$\{2\}$	Trivial	$LIS = 1$
LIS_{12}	=	$\{2, 5\}$	Es mayor, entra al final	$LIS = 2$
LIS_{13}	=	$\{2, 3\}$	Es menor, reemplaza	$LIS = 2$
LIS_{14}	=	$\{2, 3, 7\}$	Es mayor, entra al final	$LIS = 3$
LIS_{15}	=	$\{2, 3, 7, 11\}$	Es mayor, entra al final	$LIS = 4$
LIS_{16}	=	$\{2, 3, 7, 8\}$	Es menor, reemplaza	$LIS = 4$
LIS_{17}	=	$\{2, 3, 7, 8, 10\}$	Es mayor, entra al final	$LIS = 5$
LIS_{18}	=	$\{2, 3, 7, 8, 10, 13\}$	Es mayor, entra al final	$LIS = 6$
LIS_{19}	=	$\{2, 3, 6\}$	Es menor, reemplaza	$LIS = 6$

Como solo nos interesamos en la longitud máxima posible, solo la actualizamos cuando se aumente, no cuando se reemplace.
Una implementación sería la siguiente:

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

const int N = 1000005;
```

```
int n;
int A[N];
int finalesLIS[N];

int lb(int lo, int hi, int x){
        while(lo < hi){
                int mi = lo + (hi-lo)/2; // Punto medio (Lower bound)
                if(finalesLIS[mi] < x) lo = mi+1; // Transicion de aumento
                else hi = mi; // Transicion de estabilidad
        }
        return lo; // Devolvemos el indice del elemento lower_bound
}

int LIS(){
        int len = 1; // Trivial
        finalesLIS[0] = A[0]; // Trivial
        for(int i=1; i<n; i++){
                if(A[i] < finalesLIS[0]) finalesLIS[0] = A[i]; // Reemplaza el primero
                else if(A[i] > finalesLIS[len-1]) finalesLIS[len++] = A[i]; // Adjuntamos
                else finalesLIS[lb(0,len-1,A[i])] = A[i]; // Reemplaza en el punto
        }
        return len;
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> A[i];
        cout << "Longest Increasing Subsequence = " << LIS() << endl;
        return 0;
}
```

Otro método para obtener LIS en $O(nlogn)$ es usando estructuras de datos con compresión de coordenadas, pero eso se verá en su respectivo capítulo.

5.4. Subsecuencia Bitónica Más Larga (*Longest Bitonic Subsequence*)

Una secuencia **bitónica** se define como aquella subsecuencia que al inicio es creciente y luego se vuelve decreciente. El enunciado del problema sería

> Dada una secuencia de números A de tamaño n, determinar la longitud máxima posible de alguna subsecuencia bitónica $L = A_{i_1}, A_{i_2}, \ldots$ dentro de A

Por ejemplo, en el caso de la secuencia

$$A = \{1, 11, 2, 10, 4, 5, 2, 1\}$$

Tendría como LBS:

$$LBS(A) = \{1, 2, 10, 4, 2, 1\}$$

Por lo que su LDS tiene longitud 6.

En este caso, la solución es muy simple si es que tenemos bien aprendido el concepto de LIS. Lo que necesitamos es una secuencia que sea una concatenación de una LIS y una LDS(*Longest Decreasing Subsequence*), por lo que nuestra idea sería usar el algoritmo de LIS para guardar el tamaño de la LIS hasta el elemento i en LIS_i, mientras que en el caso de la LDS, nos basta modificar ligeramente el algoritmo del primero para guardar la LDS que **comience** en el elemento i en LDS_i.
Luego, al tener los valores LIS_i y LDS_i, entonces se puede formar una subsecuencia bitónica de tamaño $LIS_i + LDS_i - 1$(se cuenta dos veces el elemento A_i) si tomáramos como punto de quiebre el elemento i. La respuesta sería el máximo valor posible de esto.

$$T(n) = T(LIS(n)) + T(LDS(i)) + O(n)$$

El primer tiempo es para formar los elementos LIS_i, el segundo es para LDS_i y el último es para recorrer y maximizar el valor $LIS_i + LDS_i - 1$. Una implementación sería la siguiente:

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

const int N = 10005;

int n;
int A[N];
int LIS[N], LDS[N];

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> A[i];
        LIS[0] = 1; // Trivial
        for(int i=1; i<n; i++){
                LIS[i] = 1; // Trivial
                for(int j=0; j<i; j++){
                        if(A[j] < A[i]) // Candidato
                                LIS[i] = max(LIS[i],LIS[j]+1); // Maximizar el valor
                }
        }
        LDS[n-1] = 1; // Trivial
        for(int i=n-2; i>=0; i--){
                LDS[i] = 1;
                for(int j=n-1; j>i; j--){
                        if(A[i] > A[j]) // Candidato
                                LDS[i] = max(LDS[i],LDS[j]+1); // Maximizar el valor

                }
        }
        int LonLBS = 1; // Trivial
        for(int i=0; i<n; i++)
                LonLBS = max(LonLBS,LIS[i]+LDS[i]-1); // Maximizar el valor
        cout << "Longest Bitonic Subsequence = " << LonLBS << endl;
        return 0;
}
```

Como se ve en el código, para el caso de la LDS se ha analizado el arreglo en orden inverso para obtener las LDS que inicien en los i-ésimos elementos. Por su parte, la complejidad de este código sería:

$$T(n) = O(n^2) + O(n^2) + O(n) = O(n^2)$$

Aunque se puede optimizar a $O(nlogn)$ usando el método con *Binary Search*. Este sería un buen ejercicio de práctica.

5.5. Subsecuencia Ordenada Más Larga (Versión General)

Para ver este tema, debemos analizar correctamente los casos en los que se puede usar este método: Para empezar deberíamos tener una secuencia de elementos A definidos en un conjunto $\mathbb{F}$, para tomar subsecuencias que cumplan con una **relación** $\mathscr{R}$ definida dentro del conjunto $\mathbb{F}$ de forma en que sea **transitiva** y **antisimétrica**.
Recordando las definiciones, denotando a la relación $\mathscr{R}$ con el símbolo $\preceq$, entonces se debería cumplir dentro de $\mathbb{F}$ que

$$Si\, x \preceq y \wedge y \preceq x \rightarrow x = y \vee F$$

$$Si\, x \preceq y \wedge y \preceq z \rightarrow x \preceq z$$

Con esta condición en la definición de la relación $\mathbb{R}$, se pueden plantear subsecuencias *ordenadas* dentro de A.
Ahora, el problema se resume a encontrar la longitud máxima posible de alguna subsecuencia de A cuyos elementos se puedan ordenar mediante la relación $\preceq$ (expresada como la función booleana *menor* en este caso). Esto puede sonar bastante complejo, pero se resume siempre a hallar candidatos y tomar sus posibilidades.
Como ya habrán notado por los ejemplos anteriores, el algoritmo para hallar la respuesta a este tipo de problemas usualmente toma esta forma:

```
SOML(A):
        Soml[A.size];
        Soml[0] = 1;
        LonSOML = 1;
        Desde i=1 hasta i=A.size - 1:
                Soml[i] = 1;
                Desde j=0 hasta j=i-1:
                        Si menor(A[j],A[i]):
                                Soml[i] = max(Soml[i],Soml[j]+1);
                        Fin_si
                Fin_desde
                LonSOML = max(LonSOML,Soml[i]);
        Fin_desde
        Devolver LonSOML;
```

Este algoritmo toma $O(n^2)O(\preceq)$, donde $O(\preceq)$ es la complejidad de comparación de la relación; en el caso de LIS o LDS $O(\preceq) = O(1)$. También está la implementación con $O(nlogn)$, pero esta implica modificar también la función *upper_bound* del *Binary Search*, por lo que sería una buena práctica para generalizar las Subsecuencias Ordenadas Más Largas. Hay otras variaciones en las que se pide maximizar no la longitud sino la suma, en la mayoría de esos casos basta con un análisis un poco más profundo y modificar el algoritmo convenientemente.

5.5.1. Teorema de Dilworth

Dado que ya hemos revisado cómo hallar la SOML dada una relación de orden en el conjunto, veamos la siguiente pregunta:

> Suponga que tiene un arreglo A con $n \leqslant 20K$ enteros, ¿Cuál es la mínima cantidad de LIS disjuntas que se deben usar para gastar todos los elementos?

Para este problema en particular, podemos usar el **Teorema de Dilworth**.

Este teorema indica que si tenemos un conjunto con una relación de orden parcial definida, entonces la mínima cantidad de secuencias ordenadas disjuntas que se deben usar para gastar todos los elementos presentes es numéricamente igual a la longitud de la *anticadena* más larga posible de este conjunto.

Una anticadena se define como una secuencia de elementos tales que para cualquier $x, y \in S$:

$$x \not\leq y \quad \wedge \quad y \not\leq x$$

Por lo tanto, también podríamos hallar la anticadena más larga usando SOML pero con el complemento de la relación de orden dada.

En el problema dado, basta con hallar la LNIS, que sería la Subsecuencia No Creciente Más Larga.

5.6. Particionamiento de Palíndromos (*Minimum Palindrome Partitioning*)

El enunciado de este problema es

> Dada una secuencia de caracteres S, hallar la mínima cantidad de cortes que se deben realizar para que todos los elementos de la partición sean palíndromos.

Es sencillo notar que este problema es parecido a MCM, dado que debemos probar el realizar cortes en diversos lugares. En este caso, el enfoque con DP vería la siguiente recursión definida como $MPP(i, j)$:

$$MPP(i,j) = \begin{cases} 0 & \text{Si } S[i,\ldots,j] \text{ es un palíndromo} \\ \min\limits_{i \leqslant k < j} \{MPP(i,k) + MPP(k+1,j) + 1\} & \text{Si } S[i,\ldots,j] \text{ no es un palíndromo} \end{cases}$$

Es claro que si la cadena de caracteres que evaluamos es un palíndromo, no es necesario realizar algún corte; al igual que si no fuese un palíndromo, deberíamos buscar minimizar la cantidad de cortes que se realizarían en las dos cadenas que se formarían al realizar un corte en alguna posición posible.
El algoritmo instintivo sería implementado de la manera siguiente:

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

const int N = 10005;
const int inf = 1<<20;

int n;
string x;
bool vis[N][N];
```

```
bool vis2[N][N];
int P[N][N]; // Se modificara luego
int memo[N][N];

bool palindromo(int i, int j){
        if(vis[i][j]) return P[i][j]; // Respuesta ya calculada
        vis[i][j] = true;
        for(int k=i; k<=int((i+j)>>1); k++){
                if(x[k]!=x[i+j-k]) return P[i][j] = 0; // Hay un fallo: Falso
        }
        return P[i][j] = 1; // Coincidencia: Verdadero
}

int MPP(int i, int j){
        if(i == j) return memo[i][j] = 0; // Un solo caracter es palindromo
        if(palindromo(i,j)) return memo[i][j] = 0; // Es palindromo
        if(vis2[i][j]) return memo[i][j]; // Respuesta ya calculada
        int ans = inf;
        for(int k=i; k<j; k++){
                ans = min(ans,MPP(i,k) + MPP(k+1,j) + 1); // Minimizar el valor
        }
        vis2[i][j] = true;
        return memo[i][j] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> x;
        n = x.size();
        cout << "Minimum Palindrome Partitioning = " << MPP(0,n-1) << endl;
        return 0;
}
```

Este código tiene una complejidad de $O(n^3)$, muy similar a la complejidad de la MCM.
Una optimización implicaría en primero almacenar en un elemento P_{ij} si la cadena $S[i,\ldots,j]$ es un palíndromo o no, para luego usar estos datos dentro del MPP original, reduciendo la complejidad a $O(n^2)$. Una implementación sería la siguiente:

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

const int N = 10005;
const int inf = 1<<20;

int n;
string x;
bool vis[N][N];
int P[N][N]; // Se modificara luego
int memo[N][N];

void init(){
        for(int i=0; i<n; i++) P[i][i] = 1; // Un solo caracter es palindromo
        for(int L=2; L<n; L++){ // Longitud de la cadena a evaluar
                for(int i=0; i<n-L+1; i++){ // Inicio de la cadena a evaluar
                        if(L == 2) P[i][i+L-1] = (x[i] == x[i+L-1]);
                        else P[i][i+L-1] = (x[i] == x[i+L-1]) && P[i+1][i+L-2];
```

```
                }
        }
}

int MPP(int i, int j){
        if(i == j) return memo[i][j] = 0; // UN solo caracter es palindromo
        if(P[i][j]) return memo[i][j] = 0; // Es palindromo
        if(vis[i][j]) return memo[i][j]; // Respuesta ya calculada
        int ans = inf;
        for(int k=i; k<j; k++){
                ans = min(ans,MPP(i,k) + MPP(k+1,j) + 1); // Minimizar el valor
        }
        vis[i][j] = true;
        return memo[i][j] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> x;
        n = x.size();
        init();
        cout << "Minimum Palindrome Partitioning = " << MPP(0,n-1) << endl;
        return 0;
}
```

Es importante (sobretodo para DP) recordar que el preprocesamiento a veces es necesario para poder tener una solución que cumpla con lo deseado, puede ser un buen salvavidas.

5.7. Encadenamiento Lineal (*One-Dimensional Chaining Problem*)

El enunciado de este problema es el siguiente:

> Dada un conjunto A de pares (l_i, r_i, w_i) con $l_i \leqslant r_i$ para $i = 1, 2, \ldots$, hallar la suma máxima posible $\left(\sum_k w_k\right)$ de algún subconjunto $L = A_{i_1}, A_{i_2}, \ldots$ tal que $L_k.r < L_{k+1}.l$ para todo $k = 1, 2, \ldots$.

Por ejemplo, dado el conjunto:

$$A = \{(0,1,1); (0,3,2); (2,4,2); (2,6,2); (5,8,2); (7,8,3)\}$$

La máxima suma posible estaría dada por el subconjunto

$$L = \{(0,1,1); (2,4,2); (7,8,3)\}$$

Y su valor es 6.
Es muy simple notar que este problema es muy similar al de SOML, la variación más importante es que nos enfocaremos ya no en la longitud máxima, sino en la suma máxima de los pesos de cada intervalo.
Nuestra relación $\preceq$ la definiremos como

$$x \preceq y \iff x.r < y.l$$

Es evidente que cumple con nuestras condiciones de ser antisimétrica y transitiva, por lo que el algoritmo sirve. Sin embargo, es fundamental para este caso que se ordenen los elementos de la secuencia para poder formar la SOML, pero guardando la suma máxima en vez de la longitud máxima para cada i. Una implementación del algoritmo $O(n^2)$ sería la siguiente:

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

const int N = 10005;

struct nodo{
        int l; // Limite inferior del intervalo
        int r; // Limite superior del intervalo
        int w; // Peso del intervalo
};

bool menor(nodo a, nodo b){
        if(a.l < b.l) return true;
        if(a.l == b.l) return a.r < b.r;
        return false;
}

int n;
nodo A[N];
int ODCP[N];

int main(){
        cin >> n;
        for(int i=0; i<n; i++) cin >> A[i].l >> A[i].r >> A[i].w;
        sort(A,A+n,menor);
        int SumODCP = -1;
        ODCP[n-1] = A[n-1].w; // Trivial
        for(int i=n-2; i>=0; i--){
                ODCP[i] = A[i].w; // Trivial
                for(int j=n-1; j>i; j--){
                        if(A[i].r < A[j].l)
                                ODCP[i] = max(ODCP[i],ODCP[j]+A[i].w);
                }
                SumODCP = max(SumODCP,ODCP[i]);
        }
        cout << "One Dimensional Chaining Problem = " << SumODCP << endl;
        return 0;
}
```

Capítulo 6

Técnicas Avanzadas con DP

6.1. DP con *bitmask*

Hay problemas en los cuales cada estado debe manejar un conjunto pequeño de booleanos para desarrollar una solución, en estas situaciones podemos usar un número para representar al conjunto (*bitmask*) y usarlo como índice dentro de la tabla de almacenamiento.

6.1.1. Forming Quiz Teams - 10911 UVa

> Dado un grupo de $2n$ personas, se quiere formar n parejas tales que la suma de las distancias entre sus viviendas sea la mínima posible. Además $1 \leqslant n \leqslant 8$ y la entrada es hasta el EOF (*End of File*) y nos darán las coordenadas de cada vivienda.

Primero, empecemos con las notaciones: S se referirá al grupo de personas en total, $G(X)$ se referirá al conjunto de parejas que se formen con X personas y $d(i, j)$ es la distancia calculada desde la casa de la persona i a la de la persona j.
Es sencillo notar que se puede dar este caso:

$$G(2n) = \{(x_{m_1}, x_{m_2}), G(2n-2)\}$$

Tomando las medidas necesarias para que los excluidos en $G(2n-2)$ sean m y n. Ahora, nuestro *costo* (valor que queremos minimizar) es la distancia, por lo que

$$Costo(G(2n)) = d(m_1, m_2) + Costo(G(2n-2))$$

Dado que las distancias son solo dependientes de la pareja. Ahora que tenemos la recursión, nos basta minimizar para obtener la respuesta que buscamos:

$$R = \text{mín}\{Costo(G(2n))\} = \underset{m_1 \neq m_2 \wedge m_1, m_2 \notin G(2n-2)}{\text{mín}} \{d(m_1, m_2) + Costo(G(2n-2))\}$$

Como ya habíamos visto en el Problema del Cartero, usaremos un bitmask para representar a las personas que ya fueron asignadas con una pareja, por lo que nuestra recursión final se vería algo así:

$$Costo(bitmask) = \underset{m_1 \neq m_2 \wedge bitmask \& (2^{m_1}|2^{m_2}) == 0}{\text{mín}} \{d_{m_1,m_2} + Costo(bitmask|2^{m_1}|2^{m_2})\}$$

Y esto es fácil de implementar:

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

const int N = 9;

int n;
int x[2*N], y[2*N];
double memo[1<<(2*N)], d[2*N][2*N];
bool vis[1<<(2*N)];

double dp(int bitmask){
        if(bitmask+1==(int)(1<<(2*n))) return 0.0; // Caso de salida
        if(vis[bitmask]) return memo[bitmask]; // Respuesta ya calculada
        int m1=0; // Primer elemento de la pareja a formar
        while(bitmask&(int)(1<<m1)) m1++;
        double ans = DBL_MAX;
        for(int m2 = m1+1; m2 < 2*n; m2++){ // Iterar para obtener el segundo elemento
                if(int(bitmask&(int)(1<<m2))==0) // Minimizamos
                        ans = min(ans,d[m1][m2] + dp(bitmask|(1<<m1)|(1<<m2)));
        }
        vis[bitmask] = true; // Estado visitado
        return memo[bitmask] = ans; // Almacenar y devolver respuesta
}

int main(){
        string name;
        int t = 1;
        while(cin >> n && n!=0){ // Leer los datos hasta el End of File
                memset(vis,0,sizeof vis);
                for(int i=0; i<2*n; i++) cin >> name >> x[i] >> y[i];
                for(int i=0; i<2*n; i++){
                        for(int j=0; j<2*n; j++){ // Calcular distancias
                                d[i][j] = hypot((x[i]-x[j]),(y[i]-y[j]));
                        }
                }
                printf("Case %d: %.2f\n",t++,dp(0));
        }
        return 0;
}
```

Las complejidades de este algoritmo serían de $O(2^m)$ por la cantidad de estados posibles y $O(m)$ para la iteración en cada estado, con $m = 2n$. Finalmente, es un algoritmo de $O(m2^m)$ con $2 \leqslant m \leqslant 16$, lo cual es suficientemente eficiente.

6.1.2. Another Sith Tournament - 678E Codeforces

> Hay un torneo con n participantes, se dan luchas consecutivas eligiendo al azar los 2 luchadores iniciales y luego uno a uno para cubrir el lugar del perdedor. El participante 1 tiene la habilidad de manipular el orden de elección de las luchas a su beneficio. Halle la probabilidad de que sobreviva, dado que nos dan la probabilidad de que el participante i le gane al j para todo $1 \leqslant i, j \leqslant 18$ y $1 \leqslant n \leqslant 18$.

Para analizar este problema, debemos darnos cuenta de que podemos manipular el orden, así

que no es probabilístico; esto implica que solamente debemos elegir un "camino" que nos de la máxima probabilidad para que el participante 1 sobreviva.

En tal caso, definimos la función $DP(last, bm)$, la cual nos dará la probabilidad máxima de que el participante 1 gane dado que bm representa por 1 a los que ya han luchado y por 0 a los que se pueden elegir como siguientes participantes y que $last$ es el vencedor de la última lucha.

Notemos además que podríamos definir una batalla 0 para el primer seleccionado para luchar, el cual siempre ganará esta batalla, lo cual nos facilita un poco la implemetación. Ahora definamos la recursión (Usaremos los índices $0 \to (n-1)$ por comodidad):

1. $last = 0$: Elegir cualquiera para asumir que gana 0.
2. $last \neq 0$: Elegir cualquiera; si es 0, $last$ debe perder, si no, probar las dos opciones posibles (gana $last$ o gana el elegido actual).

Ahora que lo vimos como predicado, veámoslo de forma matemática:

$$DP(0, bm) = \max_{bm\&2^j=0} \left\{ v_{0,j} DP(0, bm|2^j) \right\}$$

$$DP(last \neq 0, bm) = \begin{cases} \max\limits_{bm\&2^j=0} \left\{ v_{last,j} DP(last, bm|2^j) + v_{j,last} DP(j, bm|2^j) \right\} & j \neq 0 \\ v_{0,last} DP(0, bm|1) & j = 0 \end{cases}$$

Bueno, ahora que planteamos la recursión, una implementación sería la siguiente:

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

const int N = 20;

int n;
double v[N][N];
double memo[N][1<<N];
bool vis[N][1<<N];

double DP(int last, int bm){
        if(bm+1 == int(1<<n)) return 1.0; // Caso final
        if(vis[last][bm]) return memo[last][bm]; // Respuesta ya calculada
        double ans = 0.0; // Minimo posible
        if(last == 0){ // Caso 1
                for(int i=0; i<n; i++){
                        if((bm>>i)&1) continue; // Ignorar los que ya se tomaron
                        ans = max(ans,1.0*v[last][i]*DP(last,bm|int(1<<i)));
                }
        }
        else{ // Caso 2
                for(int i=0; i<n; i++){
                        if((bm>>i)&1) continue; // Ignorar los que ya se tomaron
                        if(i == 0){ // Subcaso 1
                                ans = max(ans,1.0*v[i][last]*DP(i,bm|int(1<<i)));
```

```
                        }
                        else{ // Subcaso 2
                                double carry = 1.0*v[last][i]*DP(last,bm|int(1<<i));
                                carry += 1.0*v[i][last]*DP(i,bm|int(1<<i));
                                ans = max(ans,carry);
                        }
                }
        }
        vis[last][bm] = true; // Estado visitado
        return memo[last][bm] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++){
                for(int j=0; j<n; j++){
                        cin >> v[i][j];
                }
        }
        double ans = 0.0;
        for(int i=0; i<n; i++){
                        ans = max(ans,DP(i,1<<i));
        }
        cout << setprecision(10) << fixed << ans << endl;
        return 0;
}
```

Este algoritmo tendrá complejidad $O(n^2 2^n)$

6.1.3. Little Pony and Harmony Chest - 453B Codeforces

Se tiene una secuencia a de n elementos, hallar alguna secuencia b tal que el $MCD(b_i, b_j) = 1, i \neq j$ y que $\sum_{i=1}^{n} |a_i - b_i|$ sea mínima. Además $1 \leqslant n \leqslant 100$ y $1 \leqslant a_i \leqslant 30$.

En primer lugar, este es un problema de reconstrucción de solución, así que bastará agregar algunos pasos extra luego de la recursión. Ahora fijémonos en los límites de los valores de b: El mínimo posible es 1, dando una distancia máxima de $|30 - 1| = 29$, pero también debemos considerar a aquél que tiene una distancia similar, que es $|30 - 59| = 29$, lo que implica que nuestro rango es $1 \leqslant b_i \leqslant 59$.

Ahora, notemos que para que los MCD de dos a dos en b sean 1, nos basta con llevar todos los factores primos ya usados para no volverlos a repetir en los elementos que añadiremos. Esto nos es posible debido a que hay 17 primos entre el 1 y el 59. Nuestra implementación los llevará en un bitmask por orden.

Además de ello, debemos tener para cada número del rango de b, un bitmask que represente qué factores primos tiene presente, para hacer la comparación con los que ya tenemos, esto es preprocesamiento útil para este caso.

Planteamos la función $DP(pos, bm)$, que nos dará la mínima suma de distancias posibles asumiendo que ya se colocaron los elementos hasta b_{pos}, los cuales usan todos los factores expresados en bm por 1 en su posición de la máscara.

Veamos la recursión matemática:

$$DP(pos, bm) = \min_{1 \leqslant i < MAX} \{|a_{pos} - i| + DP(pos + 1, bm|B_i)\}$$

Asumiendo que guardamos en un arreglo B los bitmask de factores primos de los números del 1 al MAX.

Para la reconstrucción de la solución, podemos guardar un arreglo de padres y elecciones, en los cuales guadaremos el bitmask tal que $DP(pos, bm)$ da su mejor respuesta tomando el camino de $DP(pos + 1, padre_{pos,bm})$, la cual es obtenida colocando en b_{pos} el valor de $eleccion_{pos,bm}$. No es muy complicado, solo es cuestión de plantear bien la dependencia.

Una implementación sería:

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

const int N = 105;
const int MAX = 60;

int n;
int a[N];
const int p = 17;
int primos[] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59};
int B[MAX];
int memo[N][(1<<p)+2];
bool vis[N][(1<<p)+2];
int padre[N][(1<<p)+2];
int eleccion[N][(1<<p)+2];

int DP(int pos, int bm){
        if(pos == n) return 0; // Caso de salida
        if(vis[pos][bm]) return memo[pos][bm]; // Respuesta ya calculada
        int ans = INT_MAX; // Maximo posible
        for(int k=1; k<MAX; k++){
                if(int(B[k]&bm)!=0) continue;
                int candidato = abs(a[pos]-k)+DP(pos+1,bm|B[k]); // Calcular
                if(candidato < ans){ // Minimizamos
                        ans = candidato; // Asignamos valor
                        padre[pos][bm] = bm|B[k]; // Asignamos padre
                        eleccion[pos][bm] = k; // Asignamos eleccion
                }
        }
        vis[pos][bm] = true; // Estado visitado
        return memo[pos][bm] = ans; // Almacenar y devolver respuesta
}

int main(){
        cin >> n;
        for(int i=0; i<n; i++){
                cin >> a[i];
        }
        for(int i=2; i<MAX; i++){ // Preprocesamiento de cada valor
                for(int k=0; k<p; k++){ // Prueba con primos
```

```
                        if(i%primos[k]==0){ // Factor presente
                                B[i] |= int(1<<k); // Prender bit
                        }
                }
        }
        int ans = DP(0,0); // Calcular respuesta y procesar
        int b_act = 0; // Valor inicial de bm
        for(int i=0; i<n; i++){ // Iteracion a traves de estados
                int nextb = padre[i][b_act]; // Siguiente bm
                int usado = eleccion[i][b_act]; // Eleccion
                printf("%d ",usado); // Imprimimos
                b_act = nextb; // Asignamos el nuevo valor
        }
        puts("");
        return 0;
}
```

Este algoritmo tendrá complejidad $O(n2^{\Pi(60)})$, donde $\Pi(x)$ es la cantidad de números primos hasta x.

6.2. DP sobre árboles

Hay veces en las que tendremos que usar DP sobre grafos, principalmente árboles dado que se puede establecer un orden para formar un DAG (Grafo acíclico y dirigido), estructura natural del DP.

La estrategia clásica para resolver estos problemas es plantear la recursión de un nodo determinado hacia sus "hijos" desde nuestra perspectiva. A veces es necesario que uno mismo determine la raiz del arbol y a partir de ahi empezar con la recursión dándole jerarquía al árbol como apoyo, mientras que en otros casos la raíz nos es brindada en el problema.

Veamos uno de los problemas más clásicos de DP sobre árboles.

6.2.1. Minimum Vertex Cover - Árboles

El enunciado de este problema es el siguiente:

> Dado un árbol $T = (V, E)$, hallar el mínimo tamaño del conjunto S de nodos que se pueden usar tales que para cada $e \in E$ se cumpla que alguno de sus extremos pertenezca a S.

Para resolver este problema tenemos 2 opciones para cada nodo, las cuales son añadirlo o no al conjunto S, esta es nuestra primera observación. En segundo lugar, ahora apelando a nuestra introducción de este subtema, si tuviéramos un nodo u con hijos $v_1, v_2, \ldots, v_m$, ¿Cuál sería la solución?

La mejor forma de verlo es así:

$$f(u) = g(v_1, v_2, \ldots, v_m)$$

Para este problema, tenemos dos opciones para cada nodo, por lo que podemos ponerlas como parámetro (representando 0 si no lo agrego y 1 si es que sí lo hago).

Notemos que si es que no consideramos el nodo u dentro de S, entonces todas las aristas formadas (u, v_i) deben estar cubiertas por v_i, por lo que llegamos a esta recursión:

$$f(u, 0) = \sum_{i=1}^{m} f(v_i, 1)$$

Ahora veamos que si es que sí tomamos u, entonces tenemos la opción de tomar o no cualquiera de los v_i, así que en tal caso se tiene que:

$$f(u, 1) = 1 + \sum_{i=1}^{m} \text{mín}\{f(v_i, 0), f(v_i, 1)\}$$

No es necesario definir casos base, pues en el caso que estemos en un nodo que sea hoja, simplemente tomará los valores de 0 o 1 como corresponda y los dará como respuesta.

Una implementación sería la siguiente:

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

const int N = 100000+5;

int n;
vector<int> G[N];
int memo[N][3];
bool vis[N][3];

int DP(int u, int tomo, int parent){
        if(vis[u][tomo]) return memo[u][tomo];
        int ans = 0;
        if(tomo){ // Tomare el nodo u
                ans += 1; // Cuento el nodo u
                for(int i=0; i<G[u].size(); i++){
                        int v = G[u][i];
                        if(v!=parent){ // Condicion para pseudo - DAG
                                ans += min(DP(v,0,u),DP(v,1,u)); // Minimizo
                        }
                }
        }
        else{
                for(int i=0; i<G[u].size(); i++){
                        int v = G[u][i];
                        if(v!=parent){ // Condicion para pseudo - DAG
                                ans += DP(v,1,u); // Unica opcion
                        }
                }
        }
        vis[u][tomo] = true;
        return memo[u][tomo] = ans;
}

int main(){
        int u,v;
        cin >> n;
        for(int i=0; i<n-1; i++){
                cin >> u >> v;
```

```
            G[u].push_back(v);
            G[v].push_back(u);
        }
        cout << "El Minimum Vertex Cover del arbol es " << min(DP(1,0,0),DP(1,1,0)) << endl;
        return 0;
}
```

La complejidad de este algoritmo es $O(V+E) = O(V)$ para revisar todo con un DFS en el árbol.

Ahora veamos un problema que no es típico pero de todas maneras es interesante:

6.2.2. Corte de Máximo Valor

> Dado un árbol binario completo (nos darán solo su altura h en *Pre orden*) y cada nodo tiene peso, hallar el subconjunto S de nodos tal que su suma de pesos sea la máxima posible, que S sea un *corte* del árbol y su tamaño no exceda k. Un corte está definido como un subconjunto de nodos tal que para todo camino de la raíz a alguna hoja solo esté presente un nodo del corte.

Para resolver este problema primero debemos notar que si es que colocamos un nodo en el corte S, entonces ya no podemos tomar algún otro nodo de su subárbol y por ende ahí acabaría el caso. Una observación extra es que dado que si para un nodo u tenemos que asignar a lo mucho k nodos para el corte, entonces podemos probar darle i de esos nodos al subárbol izquierdo y $k-i$ al subárbol derecho, siendo que ambas cantidades sean positivas.

Entonces definimos nuestra recursión como $DP(u,l)$ que nos dará el corte con suma máxima para el subárbol con raíz u tal que este tenga a lo mucho l nodos. Entonces veamos los casos base.

$$DP(u,1) = v_u$$
$$DP(u,l) = v_u \quad u \text{ es una hoja}$$
$$DP(u,l) = 0 \quad u > n$$

Además de ello, para definir la recursión tenemos que:

$$DP(u,l) = \text{máx}\left\{v_u, \underset{1 \leqslant i \leqslant l-1}{\text{máx}} \{DP(2u,i) + DP(2u+1, l-i)\}\right\}$$

Y eso es todo lo que necesitamos para nuestro DP, además implementaremos una función para leer el árbol en *Pre Orden*, así que no es necesario entender la parte de la lectura a menos que ya haya leido el tema de Grafos.

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

const int N = int(1<<20)+5;
const int K = 25;

int n, h, k;
int v[N];
```

```
long long memo[N][K];
bool vis[N][K];

long long DP(int u, int l){
        if(u > n) return 0; // No es un nodo del arbol
               if(l == 1 || 2*u > n) return v[u]; // Casos base
        if(vis[u][l]) return memo[u][l];
        long long ans = v[u]; // Caso minimo
        for(int i=1; i<=l-1; i++){
                ans = max(ans,DP(2*u,i)+DP(2*u+1,l-i)); // Maximizamos
        }
        vis[u][l] = true;
        return memo[u][l] = ans;
}

void LeerPreOrden(int pos, int nivel){
        if(nivel == h){
                cin >> v[pos];
                return;
         }
        cin >> v[pos];
        LeerPreOrden(2*pos,nivel+1);
        LeerPreOrden(2*pos+1,nivel+1);
}

int main(){
        cin >> h >> k;
        n = (1<<(h+1))-1; // Cantidad de nodos en todo el arbol
        LeerPreOrden(1,0);
        cout << "El Corte de Maximo Valor es " << DP(1,k) << endl;
        return 0;
}
```

Dado que la cantidad de estados serán $O(nl)$ y por cada estado estaremos realizando una iteración de $O(l)$, la complejidad total será de $O(nl^2)$.

6.3. DP sobre dígitos

Los problemas de DP sobre dígitos son, por lo general, de conteo. Nos son brindadas condiciones para el tipo de número que se busca (en función a relaciones entre sus dígitos) y deberemos dar como respuesta la cantidad de números que cumplen con esta condición y que además son menores o iguales a un valor n.

Este tipo de problemas tienen como parámetro fundamental la posición dentro del número, y en algunos casos también una variable booleana que nos determinará si el prefijo que ya hemos usado es menor o si es igual que el prefijo del límite superior.

6.3.1. Conteo de Números - UVa 11472

Este tipo de conteo de números es el más básico que se puede ver en DP sobre dígitos, puesto que no establece un límite de los valores que se cuentan, sino que da las características de la es-

tructura de estos.

El enunciado del problema *Beautiful Numbers* del UVa Judge se puede resumir a lo siguiente:

> Dados n y m, hallar la cantidad de números en base n que tengan como máximo m dígitos y que sean **hermosos**. $2 \leqslant n \leqslant 10$ y $0 \leqslant m \leqslant 100$. Dar la respuesta módulo $10^9 + 7$
>
> Un número en base n es **hermoso** si usa todos los dígitos del rango $[0, n-1]$ y la diferencia entre cualquier par de dígitos adyacentes es 1.

Para resolver este problema debemos notar que si $M < N$ entonces la respuesta es automáticamente 0, puesto que no habrían suficientes dígitos para ocupar todos los posibles en la base N.

Ahora, para poder formar la cantidad de números posible necesitamos saber lo siguiente:

- Qué posición de dígito vamos a asignar.
- Qué digito fue el último usado (para poder verificar los posibles valores que podemos asignar a la posición actual).
- Qué dígitos del rango a usar ya han sido asignados en el número. (Para verificar al terminar de construir el número).

Esto implica que usaremos una función que tenga 3 parámetros: posición, último dígito y algo que nos permita saber qué dígitos ya han sido usados. Para los dos primeros nos basta un entero como índice, mientras que podemos aprovechar el hecho de que N sea pequeño para poder usar un bitmask que determine lo deseado.

Plantearemos ahora nuestra función $DP(pos, last, used)$, que nos dará la cantidad de números en base N que sean hermosos dado que ya fueron asignados los dígitos hasta la posición $pos - 1$, siendo $last$ el último dígito usado y $used$ un bitmask donde 0 en el i-ésimo bit indica que el dígito i aún no ha sido usado y 1 en caso contrario.

Es sencillo notar que el conteo solo necesita probar los dos posibles valores para asignar el nuevo dígito, que serían $last - 1$ y $last + 1$ (verificar que sean válidos antes), además de que el número actual puede ya ser un número hermoso (si $used + 1 = 2^n$) y tiene claramente una cantidad de dígitos menor o igual a M (lo que significa que debemos agregarlo al conteo).

Entonces, llegamos a nuestra recursión:

$$DP(pos, last, used) = (used + 1 = 2^n) + \sum_{0 \leqslant last+k \leqslant n-1}^{k=-1,1} DP(pos + 1, last + k, used | 2^{last+k})$$

Esto considera todo lo mencionado anteriormente, la suma de valores posibles y el agregar el número actual si es que cumple con ser hermoso ($used + 1 = 2^n$).

Finalmente, nuestra respuesta sería calculada como

$$S = \sum_{i=1}^{n-1} DP(1, i, 2^i)$$

Puesto que nuestra función sirve si ya se asignó el primer dígito (en cuyo caso *last* estará definido). La razón por la que empezamos en 1 es debido a que los *leading zeros* no serán considerados, más bien de esto se encarga la condición $used + 1 = 2^n$, dado que es lo mismo analizar el número 123 que el 0000123, así que visualizaremos los números como si estuviesen alineados a la izquierda.

Una implementación sería la siguiente:

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

const int MOD = 1000000007;
const int N = 15;
const int M = 100+5;
const int B = (1<<11)+5;

// Suma modular

int add(long long a, long long b){
        return (a%MOD + MOD + b%MOD + MOD)%MOD;
}

int n,m;
int dn[] = {-1,1};
int memo[M][N][B];
bool vis[M][N][B];

int DP(int pos, int last, int used){
        if(pos >= m){
                // Condicion para que sea hermoso
                return used+1==1<<n;
        }
        if(vis[pos][last][used]) return memo[pos][last][used];
        int ans = used+1 == 1<<n; // El numero ya es hermoso
        for(int k=0; k<2; k++){
                int new_digit = last+dn[k];
                // Verificamos nuevo digito
                if(0 <= new_digit and new_digit < n){
                        // Agregamos resultado de asignarlo
                        ans = add(ans,DP(pos+1,new_digit,used|1<<new_digit));
                }
        }
        // Marcamos como visitado
        vis[pos][last][used] = true;
        // Almacenamos y devolvemos respuesta
        return memo[pos][last][used] = ans;
}

int main(){
        int t;
        scanf("%d",&t);
        while(t--){
                scanf("%d %d",&n,&m);
                // Si m < n no hay numeros posibles
                if(m < n) puts("0");
```

```
                else{
                        memset(vis,0,sizeof vis);
                        int ans = 0;
                        for(int i=1; i<n; i++){
                                ans = add(ans,DP(1,i,1<<i));
                        }
                        printf("%d\n",ans);
                }
        }
        return 0;
}
```

Dado que en cada estado realizamos una cantidad de pasos constante $O(1)$, la complejidad del algoritmo será $O(nm2^n)$ por caso de prueba.

6.3.2. Conteo de Números - Cantidad de Palíndromos

Un problema clásico es contar la cantidad de números palíndromos hay hasta un número n desde el número 0. Consideremos que $1 \leqslant n \leqslant 10^{18}$.

Para resolver este problema debemos considerar lo siguiente:

- Dado que n tenga longitud len en base 10, entonces todos los palíndromos de longitud menor que len serán contados sin restricciones.
- Para formar un palíndromo de longitud len basta con asignar los primeros $\lceil \frac{len}{2} \rceil$ dígitos.
- Si el prefijo que hemos asignado es menor que el prefijo de n, entonces el valor del palíndromo es menor que n.
- Si el prefijo que hemos asignado es igual que el prefijo de n, el palíndromo será menor si y solo si el resto de n es mayor que el prefijo revertido.
- 0 será palíndromo para este problema.

Para empezar, debido a que nuestro conteo consistirá de muchos valores, plantearemos la función $f(n)$, que nos dará la respuesta para un valor n.

Tenemos algunos casos base, que son los números con $len = 1$, cuya respuesta será $n + 1$ (incluyendo el 0 por la última consideración). Además de ello, para todo $n < 0$ la respuesta es 0.

Podemos plantear la función DP, la cual nos dará la cantidad de números palíndromos de longitud len que sean menores que n; por otra parte, definimos la cantidad Q_k que nos dará la cantidad de palíndromos de longitud k (sin el 0), con ello logramos que ($\forall n \geqslant 10$):

$$f(n) = 1 + \sum_{i=1}^{len-1} Q_i + DP + check$$

Donde *check* es una función que devuelve 1 si n es palíndromo y 0 en caso contrario. El 1 agregado inicialmente es debido a que Q no considera al 0.

Ahora que tenemos la estructura de nuestra función de respuesta, debemos determinar una manera eficiente de procesar la sumatoria de Q y definir por fin nuestro DP.

Para definir la cantidad Q_k nos podemos dar cuenta de que basta con asignar los primeros $\lfloor \frac{k}{2} \rfloor$ dígitos del prefijo, agregando un dígito más si k es impar. Notemos que todas las posiciones excepto la de mayor orden tienen 10 posibles valores (Pues no pueden empezar en 0).

Primero determinemos la cantidad de dígitos que se le asigna a un número de longitud k:

$$C(k) = \left\lfloor \frac{k}{2} \right\rfloor + [\text{k es impar}]$$

Donde $[x]$ toma 1 si x es verdadero y 0 si es falso.

Es sencillo notar que si $k = 2m$ tendrá $C(2m) = m$, pero si $k = 2m - 1$ tendrá $C(2m - 1) = (m - 1) + 1 = m$; llegando a que cada par $k = 2m, 2m - 1$ tendrá el mismo valor Q_k.

Consideremos el caso base $Q_1 = 9$, entonces $Q_2 = 9$ por lo anterior. Notemos que si deseamos agregar un número más al medio de algún palíndromo, este tendrá 10 posibilidades (pues no será el de mayor orden), entonces:

$$Q_k = 10Q_{k-2}, \forall k > 1$$

Entonces, podemos procesar los Q_i en $O(1)$ pues i puede tomar valores del 1 hasta el 18. Además de ello, para aumentar la rapidez y evitar volver a procesar, podemos definir un arreglo ac_i que será igual a:

$$ac_0 = 0$$
$$ac_i = 1 + \sum_{k=1}^{i} Q_k$$

Logrando que nuestra función se vuelva:

$$f(n) = ac_{len-1} + DP + check$$

Ahora definimos nuestra función $DP(pos, menor)$, debido a que solamente necesitamos saber qué prefijo hemos asignado ya y si es menor que el prefijo de n. El predicado de nuestra función es: $DP(pos, menor)$ nos dará la cantidad de palíndromos menores que n dado que ya asignamos un prefijo hasta $pos - 1$ tal que $menor$ determina si es menor que n o aun no (en este caso, implica que es igual).

Ahora, notemos los dos casos posibles para la transición:

- $menor$ es verdadero. Tenemos 10 opciones siempre y cuando no estemos en el mayor orden (esto lo arreglaremos en nuestra misma función).

$$DP(pos, menor) = 10 * DP(pos + 1, 1)$$

- *menor* es falso. Solo tenemos desde el 0 hasta n_{pos}, donde n_{pos} es el dígito de n de orden $len - 1 - pos$.

$$DP(pos, menor) = \sum_{k=0}^{n_{pos}} DP(pos + 1, k < n_{pos})$$

Para poder verificar los valores n_{pos} transformaremos n a un vector v donde las posiciones sean los órdenes invertidos, es decir $v_{pos} = n_{len-1-pos}$. Esto para que sea más sencilla la implementación.

Por último, verificar que el caso base es cuando $pos = \lceil \frac{k}{2} \rceil$, puesto que ya asignamos el prefijo necesario para definir el palíndromo, así que basta con contarlo si es válido o ignorarlo en caso contrario, usando las consideraciones iniciales del problema.

Una vez asignado un prefijo, este será considerado si y solamente si:

- *menor* es verdadero.
- El prefijo de n revertido es menor que el sufijo de n, valor estático que se mantendrá en una variable llamada *can*.

Al final, nuestra función tomará la expresión:

$$f(n) = ac[len - 1] + \sum_{i=1}^{v_0} DP(1, i < v_0) + check()$$

Una implementación sería la siguiente:

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

const int N = 20;

int len;
bool can;
vector<int> v;
long long ac[N];
bool vis[N][2];
long long memo[N][2];


long long DP(int pos, bool menor){
        if(pos >= (len+1)/2){
                // Caso base
                return menor or can;
        }
        if(vis[pos][menor]) return memo[pos][menor];
        long long ans = 0;
        if(menor){
                // Caso 1
                ans += 10*DP(pos+1,1);
```

```
        }
        else{
                // Caso 2
                for(int i=0; i<=v[pos]; i++){
                        ans += DP(pos+1,i<v[pos]);
                }
        }
        // Marcamos estado como visitado
        vis[pos][menor] = true;
        // Almacenamos y devolvemos respuesta
        return memo[pos][menor] = ans;
}

// Verifica que n sea palindromo
bool check(){
        for(int i=0; i<=len/2; i++){
                if(v[i]!=v[len-1-i]) return 0;
        }
        return true;
}

// Verifica si el prefijo de n
// es menor que el sufijo restante
bool PreffixLess(){
        // Iterador para prefijo
        int pos1 = len/2-1;
        // Iterador para sufijo
        int pos2= len&1?pos1+2:pos1+1;
        while(pos1 >= 0 and pos2 < len and v[pos1] == v[pos2]){
                pos1--;
                pos2++;
        }
        // Minimo lexicografico o devolver falso
        if(pos1 < 0) return false;
        return v[pos1] < v[pos2];
}

long long f(long long n){
        if(n == 0) return 0;
        if(n < 10) return n+1;
        // Transformamos n
        // a vector
        v.clear();
        while(n){
                v.push_back(n%10);
                n /= 10;
        }
        reverse(v.begin(),v.end());
        len = v.size();
        memset(vis,0,sizeof vis);
        // Verificar si el prefijo
        // Revertido es menor que el
        // Sufijo
        can = PreffixLess();
        // Ans inicializado en
        // ac[len-1] por definicion
```

```
        // de ac
        long long ans = ac[len-1];
        // Agregamos los palindromos
        // que empiecen en i
        // para todo i entre 1 y
        // v[0]
        for(int i=1; i<=v[0]; i++){
                ans += DP(1,i<v[0]);
        }
        // Agregamos a n si es palindromo
        ans += check();
        return ans;
}

void init(){
        ac[0] = 0;
        ac[1] = 9;
        ac[2] = 9;
        // Preprocesamos ac
        // Primer valor ac[i] = Q[i]
        for(int i=3; i<=18; i+=2){
                ac[i] = ac[i+1] = 10LL*ac[i-2];
        }
        // Segundo valor ac[i] = suma de Qs
        for(int i=1; i<=18; i++){
                ac[i] += ac[i-1];
        }
        // Valor final ac[i] = 1 + suma de Qs
        for(int i=1; i<=18; i++){
                ac[i]++;
        }
}

int main(){
        init();
        long long n;
        cin >> n;
        cout << f(n) << endl;
        return 0;
}
```

Dado que por cada estado realizaremos a lo mucho 10 iteraciones, consideraremos $B = 10$ como base y L como longitud de n en base B, tomando una complejidad de $O(LB)$ de manera muy vaga, puesto que el algoritmo es amortizado.

6.3.3. Conteo en Rango - LOJ 1068

El enunciado del problema del LightOJ **1068 - Investigation** se puede resumir de la siguiente manera:

> Dado un número k, hallar la cantidad de números en el rango $[a, b]$ tales que su suma sea un múltiplo de k y que el mismo número sea múltiplo de k. Además $0 < k < 10000$ y $1 \leqslant a \leqslant b < 2^{31}$. Repetir este proceso para t casos. $t \leqslant 200$

Como el mismo nombre del subtema lo dice, plantearemos un conteo sobre el rango, considerando el hecho de que deseamos una expresión como la siguiente:

$$g(a,b) = \sum_{i=a}^{b} [condicion(i)]$$

Donde $[x]$ es 1 cuando x es verdadero y 0 cuando no.

Entonces se puede ver como:

$$g(a,b) = \sum_{i=a}^{b} [condicion(i)] = \sum_{i=0}^{b} [condicion(i)] - \sum_{i=0}^{a-1} [condicion(i)]$$

Lo que implica que, si definimos $f(x) = \sum_{i=0}^{x} [condicion(i)]$:

$$g(a,b) = f(b) - f(a-1)$$

Para resolver correctamente estos problemas hay que tener una previa consideración de los casos base y los triviales ($a = 0$ o $a = 1$ según sea el caso).

Este método nos permite hallar el conteo deseado en base a una sola función uniforme para cualquier argumento, por lo que nuestro DP estará íntimamente relacionado con la función f.

Para este problema, debemos considerar algunos puntos iniciales:

a) 2^{31} tiene 10 dígitos.

b) La máxima suma de dígitos de algún número hasta 2^{31} es 81 ($10^9 - 1$). Esto implica que para cualquier valor de $k > 81$ entonces no existen números que cumplan con la condición de suma de dígitos más que el 0 (caso trivial).

c) Para el caso $k = 1$,$f(x) = x$.

Con estas consideraciones, plantearemos nuestra función DP.

Como se suele en DP sobre dígitos, supondremos que el número x del cual se obtendrá la función f estará en un arreglo v tal que el primer elemento es el dígito de mayor orden de x y el último el de las unidades. Además, asumamos que $n = v.length$.

Entonces, definamos nuestra función $DP(pos, res, sum, menor)$, que dará la cantidad de números que son menores o iguales a x tales que ya asignamos los dígitos de todas las posiciones hasta pos, los cuales aportan con res al número modulo k, además de sum como suma de dígitos modulo k y que $menor$ determina si ya es un prefijo menor que x o si es igual a x.

Esto quiere decir que asignaremos a cada posición un dígito, para que sea más sencillo de entender veamos la recursión:

El caso base es $pos = n$:

$$DP(n, res, sum, menor) = \begin{cases} 1 & res = 0 \wedge sum = 0 \\ 0 & \text{En otros casos} \end{cases}$$

La recursión es la siguiente, dividida en 2 casos:

Si $menor = 1$:

$$DP(pos, res, sum, menor) = \sum_{i=0}^{9} DP(pos+1, (res + i \times 10^{n-1-pos}) \bmod k, (sum + i) \bmod k, 1)$$

Si $menor = 0$:

$$DP(pos, res, sum, menor) = \sum_{i=0}^{v[pos]-1} DP(pos{+}1, (res{+}i{\times}10^{n-1-pos}) \bmod k, (sum{+}i) \bmod k, i < v[pos])$$

Es sencillo notar que si $menor = 1$, entonces el prefijo ya es menor que el del número x, por lo que no importa qué le agregemos a la derecha pues seguirá siendo menor que x. En cambio, si $menor = 0$, debemos cuidar de agregar solamente valores entre 0 y $v[pos]$, pues algún valor mayor sería inválido. La manera en cómo determinar si el número nuevo es menor o no es solo comparando el dígito que vamos a agregar.

Recordemos que esta recursión es posible debido a que descartamos todos los casos con $k > 81$.

Una implementación sería la siguiente:

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

const int K = 100+5;

int n;
int k;
int r[15];
vector<int> v;
int memo[10][K][82][2];
bool vis[10][K][82][2];

int DP(int pos, int res, int sum, bool menor){
        if(pos == n) return res==0 and sum == 0; // Caso base
        if(vis[pos][res][sum][menor]) return memo[pos][res][sum][menor];
        int ans = 0;
        if(menor){ // Caso menor = 1
                for(int i=0; i<10; i++){
                        int new_res = (res+i*r[n-1-pos])%k;
                        int new_sum = (sum+i)%k;
                        ans += DP(pos+1,new_res,new_sum,1);
                }

        }
        else{ // Caso menor = 0
                for(int i=0; i<=v[pos]; i++){
                        int new_res = (res+i*r[n-1-pos])%k;
                        int new_sum = (sum+i)%k;
                        bool new_menor = i<v[pos];
```

```
                        ans += DP(pos+1,new_res,new_sum,new_menor);
                }
        }
        vis[pos][res][sum][menor] = true;
        return memo[pos][res][sum][menor] = ans;
}

int f(int x){ // Funcion de conteo acumulado
        if(x == 0) return 1; // Caso trivial
        // Preprocesamiento necesario para el DP
        v.clear();
        while(x){
                v.push_back(x%10);
                x /= 10;
        }
        reverse(v.begin(),v.end());
        n = v.size();
        memset(vis,0,sizeof vis);
        // Conteo con todos los digitos posibles
        int ans = 0;
        for(int i=0; i<=v[0]; i++){
                ans += DP(1,(i*r[n-1])%k,i%k,i<v[0]);
        }
        return ans;
}

int g(int a, int b){ // Funcion de conteo en rango
        return f(b) - f(a-1);
}


int main(){
        int t;
        scanf("%d",&t);
        int caso = 1;
        int a,b;
        while(t--){
                scanf("%d %d %d",&a,&b,&k);
                if(k > 81){ // Caso descartado, no hay numeros
                        printf("Case %d: %d\n",caso++,0);
                        continue;
                }
                if(k == 1){ // Caso trivial, todos cumplen
                        printf("Case %d: %d\n",caso++,b-a+1);
                        continue;
                }
                r[0] = 1;
                for(int i=1; i<10; i++){
                        r[i] = (r[i-1]*10)%k; // Inicializamos potencias
                }
                printf("Case %d: %d\n",caso++,g(a,b));
        }
        return 0;
}
```

La complejidad total de este algoritmo sería $O(d), d = 10$ por cada estado y un total de estados

máximo aproximado de $10 \cdot 82 \cdot 82 \cdot 2 = 134480$, siendo la complejidad total de una cantidad total aproximada de 1344800 por caso de prueba.

6.4. DP - Mínimo Lexicográfico

Los problemas de DP con mínimo lexicográfico nos piden hallar, de todas las soluciones óptimas posibles, la que es mínima lexicográficamente. Para resolver esta condición, nos basta con realizar comparaciones y mantener un arreglo de elecciones (al igual que para reconstruir una respuesta), considerando lo siguiente:

La simulación de DP señala que al llegar a un estado $DP(parametros)$, lo hacemos con un prefijo P común (pues si no, no sería solución óptima), entonces:

$$DP(parametros) \xrightarrow{Transicion} DP(parametros')$$

Ahora, si planteamos nuestro DP con la condición extra de que logra el mínimo lexicográfico, entonces basta con asegurar que la transición hecha obtenga la respuesta óptima y que además sea mínima lexicográficamente, debido a que el prefijo es constante.

6.4.1. UVA 10419 - Sum-up the Primes

El enunciado de este problema se puede resumir de la siguiente manera:

> Dado un número N y un número t, debemos expresar N como la suma de t números primos, considerando las siguientes restricciones:
>
> - El número 2 solo puede ser usado como máximo 1 vez, mientras que el resto pueden ser usados máximo 2 veces.
> - Todos los primos usados deben ser menores que 300.
> - Si hay más de una solución, imprimir aquella cuya cadena sea la mínima lexicográfica.
> - Si no existe solución, imprimir "No Solution.".
> - $N \leqslant 1000$ y $t \leqslant 14$.
> - Leer la entrada hasta que se den como datos $N = 0$ y $t = 0$.

Para determinar la solución, primero notemos que necesitamos como parámetros obligatorios **el valor que vamos a armar (N)** y **la cantidad de números que nos quedan por usar (t)**. Además de ello, para mejorar nuestro tiempo de ejecución, sería bueno mantener un arreglo de los primos de los que disponemos, pero ordenados de una manera particular, sea este arreglo $primos[len]$. Notemos que si

$$N = p_1 + p_2 + p_3 + \ldots + p_t$$

Es una distribución válida para el problema, entonces el mejor candidato a respuesta (puesto que no necesariamente será óptimo, solo verificamos la forma de una posible respuesta) debe darse

que los valores p_i están ordenados lexicográficamente. Con esta consideración, la mejor manera de armar nuestra respuesta es primero ordenar los primos de manera creciente lexicográficamente. Luego de esto, podríamos asociar a cada posición la cantidad máxima de veces que puede tomarse (solamente al valor de 2 le asignamos 1, al resto 2) en un arreglo $max\_times[pos]$.

Luego de ello, podemos plantear la siguiente función $DP(pos, val, used)$, que nos dará la distribución mínima lexicográfica usando solo valores entre p_0 y p_{pos} para formar val usando $used$ de estos; ahora pensemos en la recursión:

$$DP(pos, val, used) = \min_{0 \leqslant times \leqslant max\_times[pos]} f(memo[pos-1][val - times \times v[pos]][used - times], pos, times)$$

Donde $f(s, pos, times)$ nos da la concatenación adecuada para la expresión s y $times$ veces pos. Además, nuestra función de comparación min es lexicográfica sobre cadenas (en C++ es la sobrecarga nativa del signo $<$).

Es sencillo notar que, con nuestra definición de la función DP, esta recursión es válida para distribuciones válidas, lo único que falta es agregarle una expresión que señale que la distribución que buscamos no sea posible, en cuyo caso elegimos colocar la cadena "*".

Finalmente, nuestras respuestas deseadas para los valores N y t estarán en $memo[primos\_length-1][N][t]$.

Una implementación sería la siguiente:

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

const int N = 10000+5;
const int K = 300+5;

int n,t;
int p[K];
int p_length = 0;
int max_times[K];
string primos[K];
bool compuesto[K];
int primos_length = 0;
string memo[K][N][15];

string to_s(int x){ // Funcion para convertir de entero a cadena
        string ans = "";
        while(x){
                ans = ans + char(x%10 + '0');
                x /= 10;
        }
        reverse(ans.begin(),ans.end());
        return ans;
}

int to_i(string s){ // Funcion para convertir de cadena a entero
        int x = 0;
        for(int i=0; i<s.size(); i++){
                x = (x<<1) + (x<<3) + s[i]-'0';
```

```
        }
        return x;
}

void criba(){ // Criba para obtener primos y ordenarlos
        for(int i=2; i*i<=K; i++){
                if(!compuesto[i]){
                        for(int j=i*i; j<K; j+=i) compuesto[j] = true;
                }
        }
        for(int i=2; i<=300; i++){
                if(!compuesto[i]){
                        primos[primos_length++] = to_s(i);
                }
        }
        sort(primos,primos+primos_length);
        for(int i=0; i<primos_length; i++){
                if(primos[i]=="2") max_times[i] = 1;
                else max_times[i] = 2;
                p[p_length++] = to_i(primos[i]);
        }
}

string f(string s, int pos, int times){ // Concatenacion
        if(s == ""){
                s = primos[pos];
                for(int i=0; i<times-1; i++){
                        s = s + '+' + primos[pos];
                }
        }
        else{
                for(int i=0; i<times; i++){
                        s = s + '+' + primos[pos];
                }
        }
        return s;
}

void init(){
        for(int i=0; i<N; i++){
                for(int j=0; j<15; j++) memo[0][i][j] = "*"; // Imposible por ahora
        }
        memo[0][0][0] = ""; // Caso base
        for(int i=0; i<primos_length; i++){ // Casos triviales, primera aparicion
                for(int times=1; times<=max_times[i]; times++){
                        string candidate = primos[i];
                        for(int append=0; append<times-1; append++){
                                candidate = candidate + '+' + primos[i];
                        }
                        memo[i][p[i]*times][times] = candidate;
                }
        }
        // Armamos la tabla memo para nuestro DP Iterativo
        for(int pos=1; pos<primos_length; pos++){ // Posicion
                for(int val=0; val<N; val++){ // Valor
                        for(int used=0; used<15; used++){ // Cantidad usada
```

```
                                string act = memo[pos-1][val][used];
                                for(int times=1; times<=max_times[pos]; times++){
                                        // Solo si es posible
                                        if(times <= used and val >= times*p[pos]){
                                                int lval = val - times*p[pos];
                                                int lused = used-times;
                                                string candidate = memo[pos-1][lval][lused];
                                                if(candidate != "*"){ // Si es valido
                                                        candidate = f(candidate,pos,times);
                                                        if(act == "*"){
                                                                act = candidate;
                                                        }
                                                        else if(candidate < act){
                                                                act = candidate;
                                                        }
                                                }
                                        }
                                }
                                memo[pos][val][used] = act;
                        }
                }
        }
}

int main(){
        criba(); // Inicializamos los datos
        init(); // Armamos la tabla del DP
        int caso = 1;
        while(scanf("%d %d",&n,&t) == 2 and n+t){
                string a = memo[primos_length-1][n][t]; // Consulta en O(1)
                printf("CASE %d:\n",caso++);
                if(a == "*") puts("No Solution.");
                else printf("%s\n",a.c_str());
        }
        return 0;
}
```

Bibliografía

[GeeksForGeeks, 2016] GeeksForGeeks (2016). Dp paradigm. `https://www.geeksforgeeks.org/dynamic-programming/`. Extraido: 2018-05-28.

[Halim and Halim, 2013] Halim, S. and Halim, F. (2013). *Competitive Programming 3 - The New Lower Bound of Programming Contests*.

[H.Cormen et al., 2009] H.Cormen, T., Leiserson, C. E., and Riverson, R. L. (2009). *Introduction to Algorithms*. The MIT Press.