

# Resumen de Algoritmos y Complejidad

Apuntes para la Prueba

21 de noviembre de 2025

# Índice

<b>1 Base</b>	<b>4</b>
<b>Plantilla Base y Funciones Comunes</b>	
Plantilla Base C++ para Competitiva/Pruebas . . . . .	4
1.0.1 Plantilla Inicial C++ . . . . .	4
1.0.2 Código C++ Base . . . . .	4
<b>2 Greedy</b>	<b>4</b>
<b>Algoritmos Greedy</b>	
Algoritmo de Kruskal (MST) . . . . .	4
2.0.1 Algoritmo de Kruskal (MST y MaxST) . . . . .	4
2.0.2 Código C++ (Kruskal - MST y MaxST) . . . . .	5
Algoritmo de Kruskal 2 (MST) . . . . .	6
2.0.3 Algoritmo de Kruskal (Árbol de Expansión Mínima - MST) . . . . .	6
2.0.4 Código C++ (Kruskal - MST) . . . . .	7
Mínima Distancia para Iluminar (Faroles) . . . . .	8
2.0.5 Mínima Distancia para Iluminar (Faroles) . . . . .	8
2.0.6 Código C++ (Faroles) . . . . .	8
Construir Número Máximo con Presupuesto (Cercas) . . . . .	8
2.0.7 Construir Número Máximo con Presupuesto (Cercas) . . . . .	8
2.0.8 Código C++ (Cercas) . . . . .	9
<b>3 Técnicas</b>	<b>9</b>
<b>Técnicas y Estructuras Comunes</b>	
Subarreglos con Suma Objetivo (Sliding Window) . . . . .	9
3.0.1 Subarreglos con Suma Objetivo (Sliding Window) . . . . .	9
3.0.2 Código C++ (Subarreglos con Suma) . . . . .	9
Construcción de Palíndromos . . . . .	10
3.0.3 Construcción de Palíndromos . . . . .	10
3.0.4 Código C++ (Construcción de Palíndromos) . . . . .	10
Algoritmo de Bellman-Ford (Caminos Mínimos, Fuente Única) . . . . .	11
3.0.5 Algoritmo de Bellman-Ford (Caminos Mínimos con Ciclos Negativos) . . . . .	11
3.0.6 Código C++ (Bellman-Ford) . . . . .	11
Algoritmo de Floyd-Warshall (Caminos Mínimos, Todos los Pares) . . . . .	13
3.0.7 Algoritmo de Floyd-Warshall (Caminos Mínimos - Todos los Pares) . . . . .	13
3.0.8 Código C++ (Floyd-Warshall) . . . . .	13
<b>4 Backtracking</b>	<b>14</b>
<b>Algoritmos Backtracking</b>	
Siguiente Permutación Lexicográfica (Usando next_permutation) . . . . .	14
4.0.1 Resumen: Búsqueda de la Siguiente Permutación . . . . .	14
4.0.2 Código C++ (Usando next_permutation) . . . . .	15
Generación de Todas las Permutaciones . . . . .	15
4.0.3 Generación de Todas las Permutaciones . . . . .	15
4.0.4 Código C++ (Generación de Permutaciones) . . . . .	16
<b>5 DP</b>	<b>16</b>
<b>Programación Dinámica (DP)</b>	
Problema de la Mochila (Knapsack 0/1) con Reconstrucción . . . . .	16
5.0.1 Resumen: Problema de la Mochila 0/1 (Knapsack) . . . . .	16
5.0.2 Código C++ (Knapsack 0/1 con Reconstrucción) . . . . .	16
Maximum Weight Independent Set (MWIS) en Grafo de Camino . . . . .	18
5.0.3 Maximum Weight Independent Set (MWIS) en Grafo de Camino . . . . .	18

5.0.4	Código C++ (MWIS con Reconstrucción) . . . . .	18
Cantidad Mínima de Monedas (Coin Change Minimum)	. . . . .	20
5.0.5	Cantidad Mínima de Monedas (Coin Change Minimum) . . . . .	20
5.0.6	Código C++ (Cantidad Mínima de Monedas) . . . . .	20
Combinaciones de Monedas (El Orden Importa) . . . . .		20
5.0.7	Combinaciones de Monedas (El Orden Importa) . . . . .	20
5.0.8	Código C++ (Combinaciones - Orden Importa) . . . . .	21
Combinaciones de Monedas (El Orden No Importa) . . . . .		21
5.0.9	Combinaciones de Monedas (El Orden No Importa) . . . . .	21
5.0.10	Código C++ (Combinaciones - Orden No Importa) . . . . .	21
Combinaciones con Reconstrucción Única (Problema del Camarero) . . . . .		22
5.0.11	Combinaciones con Reconstrucción Única (Problema del Camarero) . . . . .	22
5.0.12	Código C++ (Reconstrucción Única) . . . . .	22
Manejo de Entrada Mixta (getline y stringstream) . . . . .		24
5.0.13	Manejo de Entrada Mixta (getline y stringstream) . . . . .	24
5.0.14	Código C++ (Knapsack con Parsing de Línea) . . . . .	24

## 1. Base

### Ejercicio: Plantilla Base C++ para Competitiva/Pruebas

#### 1.0.1. Plantilla Inicial C++

Esta plantilla incluye directivas estándar y optimizaciones para la programación competitiva (I/O rápida, alias de tipos), comunes en entornos de prueba.

**Complejidad:** \* **Tiempo:**  $O(1)$  (Inicialización). \* **Espacio:**  $O(1)$ .

#### 1.0.2. Código C++ Base

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 # define ll long long
5 # define ld long double
6 # define endl '\n'
7
8 int main() {
9     ios_base::sync_with_stdio(false);
10    cin.tie(nullptr);
11    return 0;
12 }
```

Listing 1: Plantilla C++ Rápida para Entornos de Pruebas

## 2. Greedy

### Ejercicio: Algoritmo de Kruskal (MST)

#### 2.0.1. Algoritmo de Kruskal (MST y MaxST)

El Algoritmo de Kruskal encuentra el **Árbol de Expansión Mínima (MST)** en un grafo ponderado no dirigido. El mismo algoritmo, invirtiendo el orden de las aristas (de mayor a menor peso), encuentra el **Árbol de Expansión Máxima (MaxST)**. Ambos son algoritmos **Greedy** porque en cada paso seleccionan la arista localmente óptima (la de menor o mayor peso) que no forma un ciclo.

**Estructura Clave:** El código utiliza la estructura de conjuntos disjuntos `union_find` para detectar ciclos de manera eficiente.

#### Estrategia:

1. **MST (minSpanningTree):** Ordena las aristas por peso de forma ascendente.
2. **MaxST (maxSpanningTree):** Ordena las aristas por peso de forma descendente.
3. **Selección:** En ambos casos, se seleccionan las aristas en el orden establecido siempre que conecten dos componentes previamente disjuntas (no formando ciclos).

**Complejidad:** \* **Tiempo:**  $O(E \log E + E \cdot \alpha(V))$ , donde  $E$  es el número de aristas y  $\alpha(V)$  es la función inversa de Ackermann. El término  $E \log E$  domina debido a la ordenación. \* **Espacio:**  $O(V + E)$  (para almacenar el grafo y la estructura Union-Find).

## 2.0.2. Código C++ (Kruskal - MST y MaxST)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 # define ll long long
5 # define ld long double
6 # define endl '\n'
7
8 struct Edge{
9     int n1;
10    int n2;
11    int w;
12 };
13
14 bool custom_compare_min(Edge a, Edge b){
15     return a.w < b.w;
16 }
17
18 bool custom_compare_max(Edge a, Edge b){
19     return a.w > b.w;
20 }
21
22 struct union_find{
23     vector<int> e;
24     union_find(int n) {e.assign(n, -1);}
25
26     int findSet(int x){
27         return (e[x] < 0 ? x : e[x] = findSet(e[x]));
28     }
29
30     bool sameSet(int x, int y) {return findSet(x) == findSet(y);}
31
32     int size(int x){return -e[findSet(x)];}
33
34     bool unionSet(int x, int y){
35         x = findSet(x), y = findSet(y);
36         if (x == y) return 0;
37         if (e[x] > e[y]) swap(x,y);
38         e[x] += e[y];
39         e[y] = x;
40         return 1;
41     }
42 };
43
44 pair<int, vector<pair<int,int>>> minSpanningTree(vector<Edge> v){
45     int n = v.size();
46     sort(v.begin(), v.end(), custom_compare_min);
47     union_find sets(n);
48     vector<pair<int,int>> output;
49
50     int contador = 0;
51
52     for (int i = 0; i < v.size(); i++){
53         Edge nodo = v[i];
54         if (sets.sameSet(nodo.n1, nodo.n2) == false){
55             sets.unionSet(nodo.n1, nodo.n2);
56             output.push_back({nodo.n1+1, nodo.n2+1});
57             contador += nodo.w;
58             if (output.size() == n-1) break;
59         }
60     }
61
62     sort(output.begin(), output.end());
63
64     return {contador, output};
65 }
66
67 pair<int, vector<pair<int,int>>> maxSpanningTree(vector<Edge> v){
68     int n = v.size();
69     sort(v.begin(), v.end(), custom_compare_max);
70     union_find sets(n);
71     vector<pair<int,int>> output;

```

```

72     int contador = 0;
73
74     for (int i = 0; i < v.size(); i++){
75         Edge nodo = v[i];
76         if (sets.sameSet(nodo.n1, nodo.n2) == false){
77             sets.unionSet(nodo.n1, nodo.n2);
78             output.push_back({nodo.n1+1, nodo.n2+1});
79             contador += nodo.w;
80             if (output.size() == n-1) break;
81         }
82     }
83
84     sort(output.begin(), output.end());
85
86     return {contador, output};
87 }
88
89
90 int main (){
91     ios_base::sync_with_stdio (false);
92     cin.tie(nullptr);
93
94     int n;
95     cin >> n;
96     vector<Edge> v;
97
98     for (int i = 0; i < n; i++){
99         for (int j = 0; j < n; j++){
100             int w;
101             cin >> w;
102
103             if (i < j && w != 0){ // Triangular superior sin diagonal
104                 Edge nodo;
105                 nodo.n1 = i;
106                 nodo.n2 = j;
107                 nodo.w = w;
108                 v.push_back(nodo);
109             }
110         }
111     }
112     cout << "Min ST Value: " << minSpanningTree(v).first << endl;
113     for (auto c: minSpanningTree(v).second) cout << c.first << " " << c.second << endl ;
114
115     cout << "Max ST Value: " << maxSpanningTree(v).first << endl;
116     for (auto c: maxSpanningTree(v).second) cout << c.first << " " << c.second << endl ;
117
118     return 0;
119 }
```

Listing 2: Algoritmo de Kruskal (MST y MaxST) usando Union-Find

## Ejercicio: Algoritmo de Kruskal 2 (MST)

### 2.0.3. Algoritmo de Kruskal (Árbol de Expansión Mínima - MST)

El Algoritmo de Kruskal es un método **Greedy** que encuentra un subconjunto de aristas que conecta todos los vértices de un grafo con el **menor peso total posible** sin formar ciclos.

**Implementación Clave:** Esta implementación se centra en el **MST** utilizando la estructura `union_find` para la detección eficiente de ciclos. La función `minSpanningTree` recibe la lista de aristas y el número de nodos ( $N$ ), y retorna el costo total del MST y la lista de aristas seleccionadas. Es importante notar que la entrada de nodos se convierte de 1-based a 0-based en el `main`.

**Estrategia:**

- Ordenar:** Ordenar todas las aristas por peso de forma ascendente (`custom_compare_min`).
- Seleccionar:** Iterar sobre las aristas, seleccionando solo aquellas que unen dos componentes disjuntas (no formando ciclos).

**Complejidad:** \* **Tiempo:**  $O(E \log E + E \cdot \alpha(V))$ , donde  $E$  es el número de aristas y  $\alpha(V)$  es la función inversa de Ackermann (dominada por la ordenación  $E \log E$ ). \* **Espacio:**  $O(V + E)$  (para la estructura Union-Find y las aristas).

#### 2.0.4. Código C++ (Kruskal - MST)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct Edge{
5     int n1;
6     int n2;
7     int w;
8 };
9
10 bool custom_compare_min(Edge a, Edge b){
11     return a.w < b.w;
12 }
13
14 struct union_find{
15     vector<int> e;
16     union_find(int n) {e.assign(n, -1);}
17
18     int findSet(int x){
19         return (e[x] < 0 ? x : e[x] = findSet(e[x]));
20     }
21
22     bool sameSet(int x, int y) {return findSet(x) == findSet(y);}
23
24     int size(int x){return -e[findSet(x)];}
25
26     bool unionSet(int x, int y){
27         x = findSet(x), y = findSet(y);
28         if (x == y) return 0;
29         if (e[x] > e[y]) swap(x,y);
30         e[x] += e[y];
31         e[y] = x;
32         return 1;
33     }
34 };
35
36 // N := "Cantidad de nodos" ,,, M := "Cantidad de aristas"
37 pair<int, vector<pair<int,int>>> minSpanningTree(vector<Edge> &v, int n){
38     sort(v.begin(), v.end(), custom_compare_min);
39     union_find sets(n);
40     vector<pair<int,int>> output;
41
42     int contador = 0;
43
44     for (int i = 0; i < v.size(); i++){
45         Edge nodo = v[i];
46         if (sets.sameSet(nodo.n1, nodo.n2) == false){
47             sets.unionSet(nodo.n1, nodo.n2);
48             output.push_back({nodo.n1+1, nodo.n2+1});
49             contador += nodo.w;
50             if (output.size() == n-1) break;
51         }
52     }
53
54     sort(output.begin(), output.end());
55
56     return {contador, output};
57 }
58
59 int main() {
60     int N,M;
61     cin >> N >> M;
62     vector<Edge> v;
63
64     for (int i = 0; i < M; i++){
65         int a, b, p;
66         cin >> a >> b >> p;

```

```

67
68     Edge nodo;
69     // Los nodos deben empezar en cero
70     nodo.n1 = a-1;
71     nodo.n2 = b-1;
72     nodo.w = p;
73     v.push_back(nodo);
74 }
75
76 cout << minSpanningTree(v, N).first << endl;
77
78 return 0;
79 }
```

Listing 3: Algoritmo de Kruskal para MST con manejo de entrada N/M

## Ejercicio: Mínima Distancia para Iluminar (Faroles)

### 2.0.5. Mínima Distancia para Iluminar (Faroles)

El problema busca la mínima distancia (radio  $d$ ) para que los faroles, cuyas posiciones están dadas, iluminen completamente una calle de longitud  $L$ . La solución se basa en encontrar la máxima distancia entre faroles adyacentes y las distancias a los extremos (0 y  $L$ ) después de ordenar las posiciones.

**Complejidad:** \* **Tiempo:**  $O(n \log n)$  (por la ordenación). \* **Espacio:**  $O(n)$ .

### 2.0.6. Código C++ (Faroles)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll long long;
4
5 int main() {
6     ll n, i;
7     cin >> n >> 1;
8     vector<double> pos(n);
9     for (ll i=0; i < n; i++) {
10         cin >> pos[i];
11     }
12     sort(pos.begin(), pos.end());
13     double d=0;
14     for (ll i=0; i<n - 1; i++) {
15         d=max(d, (pos[i + 1] - pos[i]) / 2);
16     }
17     d=max(d, pos[0]);
18     d=max(d, 1 - pos[n-1]);
19     cout << fixed << setprecision (10) << d << endl;
20     return 0;
21 }
```

Listing 4: Mínima distancia requerida para iluminar una calle

## Ejercicio: Construir Número Máximo con Presupuesto (Cercas)

### 2.0.7. Construir Número Máximo con Presupuesto (Cercas)

El objetivo es construir el número lexicográficamente más grande posible con el máximo número de dígitos, dado un presupuesto  $V$  y un costo  $C_d$  por cada dígito. La estrategia es doblemente greedy: 1) maximizar la longitud con el dígito de costo mínimo ( $C_{\min}$ ), y 2) luego, reemplazar los dígitos  $D_{\min}$  de izquierda a derecha por los dígitos más grandes posibles (del 9 al 1) que la pintura restante pueda pagar.

**Complejidad:** \* **Tiempo:**  $O(L)$ , donde  $L$  es la longitud máxima de la cadena. \* **Espacio:**  $O(L)$ .

### 2.0.8. Código C++ (Cercas)

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll long long;
4
5 int main() {
6     ll v;
7     cin >> v;
8     vector<ll> nums(9);
9     for (ll i=0; i < 9; i++) {
10         cin >> nums[i];
11     }
12     ll min_cost = *min_element(nums.begin(), nums.end());
13     ll min_digit = min_element(nums.begin(), nums.end()) - nums.begin() + 1;
14
15     if (min_cost > v) {
16         cout << -1 << endl;
17         return 0;
18     }
19
20     ll max_length = v / min_cost;
21     ll remaining_paint = v % min_cost;
22
23     string result(max_length, '0' + min_digit);
24
25     for (ll d=9; d >= min_digit; d--) {
26         for (ll i=0; i < max_length; i++) {
27             ll cost_diff = nums[d-1] - min_cost;
28             if (remaining_paint >= cost_diff) {
29                 result[i] = '0' + d;
30                 remaining_paint -= cost_diff;
31             } else {
32                 break;
33             }
34         }
35     }
36
37     cout << result << endl;
38     return 0;
39 }
```

Listing 5: Construir número máximo con presupuesto de pintura

## 3. Técnicas

### Ejercicio: Subarreglos con Suma Objetivo (Sliding Window)

#### 3.0.1. Subarreglos con Suma Objetivo (Sliding Window)

Algoritmo que usa la técnica de la Ventana Deslizante (Sliding Window) para encontrar el número de subarreglos contiguos en un array de enteros positivos cuya suma es igual a un valor objetivo  $X$ . Utiliza una deque para expandir y contraer la ventana de forma eficiente en  $O(1)$ .

**Complejidad:** \* **Tiempo:**  $O(N)$ , ya que cada elemento se procesa y elimina de la ventana a lo sumo una vez. \* **Espacio:**  $O(N)$ .

#### 3.0.2. Código C++ (Subarreglos con Suma)

```
1 #include <bits/stdc++.h>
2 using ll long long;
3 using namespace std;
4
5
6 int main() {
7     ll n, x, cont = 0, suma = 0;
```

```

8     cin >> n >> x;
9     deque<int> dq;
10    for (int i=0; i < n; i++) {
11        int num;
12        cin >> num;
13        dq.push_back(num);
14        suma += num;
15        while (suma > x) {
16            suma -= dq.front();
17            dq.pop_front();
18        }
19
20        if (suma == x) {
21            cont++;
22            // Este bloque adicional maneja el caso donde solo queremos el número de
23            // subarreglos
24            // y no queremos que se solapen si hubiera más de una solución.
25            // Para contar todos los subarreglos, la condición de avance (pop_front) puede
26            // variar.
27            suma -= dq.front();
28            dq.pop_front();
29        }
30
31    cout << cont << endl;
32    return 0;
}

```

Listing 6: Sliding Window para Suma Objetivo (usando deque)

## Ejercicio: Construcción de Palíndromos

### 3.0.3. Construcción de Palíndromos

El problema verifica si los caracteres de una cadena pueden reordenarse para formar un palíndromo. Esto solo es posible si, a lo sumo, un carácter tiene una frecuencia de aparición impar. La solución construye el palíndromo tomando la mitad de la frecuencia de cada carácter, poniendo el carácter impar en el centro (si existe), y luego añadiendo la mitad invertida.

**Complejidad:** \* **Tiempo:**  $O(N \log N)$  (debido al map y la construcción de la cadena). \* **Espacio:**  $O(N)$ .

### 3.0.4. Código C++ (Construcción de Palíndromos)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using int64_t long long;
4 int main() {
5     string s;
6     cin >> s;
7     map<char, int64_t> m;
8     for (int i=0; i < s.size(); i++) {
9         m[s[i]]++;
10    }
11    int not_pair = 0;
12    char odd_char = ',';
13    for (auto &it: m) {
14        if (it.second % 2 != 0) {
15            not_pair++;
16            odd_char = it.first;
17        }
18    }
19    if (not_pair > 1) {
20        cout << "NO SOLUTION" << endl;
21        return 0;
22    }
23    string half = "";

```

```

24     for (auto &it: m) {
25         half += string(it.second / 2, it.first);
26     }
27     string palindrome = half;
28     if (not_pair == 1) {
29         palindrome += odd_char;
30     }
31     reverse (half.begin(), half.end());
32     palindrome += half;
33     cout << palindrome << endl;
34     return 0;
35 }
```

Listing 7: Construir Palíndromo a partir de las frecuencias de caracteres

## Ejercicio: Algoritmo de Bellman-Ford (Caminos Mínimos, Fuente Única)

### 3.0.5. Algoritmo de Bellman-Ford (Caminos Mínimos con Ciclos Negativos)

Bellman-Ford encuentra los caminos más cortos desde un nodo fuente único ( $s$ ) a todos los demás en un grafo dirigido y ponderado. Es el único algoritmo de fuente única que funciona correctamente con **aristas de peso negativo**. Su principal fortaleza es la capacidad de **detectar y encontrar ciclos de peso negativo**.

**Contexto de Ejercicio Típico (Detección de Arbitraje):** Dado un conjunto de monedas y tasas de cambio (donde las tasas pueden representarse como logaritmos negativos de los pesos de las aristas), determine si existe una secuencia de transacciones que le permita terminar con más dinero del que empezó (un **ciclo de arbitraje o ciclo negativo**).

**Estrategia:** El algoritmo relaja todas las aristas  $V - 1$  veces. Este número de iteraciones garantiza que se encuentre el camino más corto, asumiendo que no hay ciclos negativos. Una iteración  $V$  adicional se usa para detectar la presencia de un **ciclo de peso negativo**.

**Recurrencia (Relajación):** Para cada arista  $(u, v)$  con peso  $w$ :

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w)$$

**Detección de Ciclos:** Si en la  $V$ -ésima iteración la distancia de algún nodo  $\text{dist}[v]$  se sigue actualizando, existe un ciclo negativo alcanzable. El código localiza un nodo dentro de este ciclo para reconstruirlo.

**Complejidad:** \* **Tiempo:**  $O(V \cdot E)$ , donde  $V$  es el número de vértices y  $E$  es el número de aristas. \* **Espacio:**  $O(V)$  para almacenar distancias y predecesores.

### 3.0.6. Código C++ (Bellman-Ford)

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 #define int long long // Usamos long long para distancias para prevenir overflow
5 using ll = long long;
6
7 struct BellmanFord {
8     struct Edge { int from, to, weight; };
9     int n, last_updated = -1;
10    const int INF = 1e9 + 7; // INF simplificado a int
11
12    vector<int> p;
13    vector<ll> dist; // Mantener ll para distancias/pesos acumulados
14
15    BellmanFord(vector<Edge> &G, int s, int nn) {
```

```

16     n = nn;
17     dist.assign(n+2, INF);
18     p.assign(n+2, -1);
19     dist[s] = 0;
20
21     for (int i = 1; i <= n; i++) {
22         last_updated = -1;
23         for (Edge &e : G)
24             if (dist[e.from] != INF && dist[e.from] + e.weight < dist[e.to]) {
25                 dist[e.to] = dist[e.from] + e.weight;
26                 p[e.to] = e.from;
27                 last_updated = e.to;
28             }
29     }
30 }
31
32     bool getCycle(vector<int> &cycle) {
33         if (last_updated == -1) return false;
34
35         // Moverse V veces para entrar en el ciclo
36         for (int i = 0; i < n; i++) // Cambiado a 'n' iteraciones por seguridad
37             last_updated = p[last_updated];
38
39         // Reconstruir el ciclo
40         for (int x = last_updated;; x = p[x]) {
41             cycle.push_back(x);
42             if (x == last_updated && cycle.size() > 1) break;
43         }
44         reverse(cycle.begin(), cycle.end());
45         return true;
46     }
47 };
48
49     void solve(){
50         int n, m;
51         cin >> n >> m;
52         vector<BellmanFord::Edge> G;
53         for(int i = 0; i < m; i++){
54             int u, v, w;
55             cin >> u >> v >> w;
56             G.push_back({u,v,w});
57         }
58
59         // Suponemos que el nodo fuente es 1
60         BellmanFord mish(G, 1, n);
61
62         vector<int> ans;
63         if(mish.getCycle(ans)){
64             cout << "YES " << endl;
65             for(auto x: ans){
66                 cout << x << " ";
67             }
68             cout << endl;
69         } else{
70             cout << "NO" << endl;
71         }
72     }
73
74     signed main(){
75         solve();
76         return 0;
77     }

```

Listing 8: Implementación de Bellman-Ford con detección y reconstrucción de ciclo negativo, JAIME INOSTROZA MI DIOS

## Ejercicio: Algoritmo de Floyd-Warshall (Caminos Mínimos, Todos los Pares)

### 3.0.7. Algoritmo de Floyd-Warshall (Caminos Mínimos - Todos los Pares)

Floyd-Warshall calcula las distancias más cortas entre **cada par de nodos** en un grafo ponderado. Es ideal cuando se necesita una matriz completa de distancias.

**Estrategia:** Utiliza la Programación Dinámica al iterar sobre un nodo intermedio  $k$ . Se pregunta: ¿es más corto ir del nodo  $i$  al nodo  $j$  pasando por el nodo intermedio  $k$ ?

**Fórmula DP:** Sea  $D[i][j]$  la distancia más corta de  $i$  a  $j$ .

$$D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$$

La clave es que la iteración sobre el nodo intermedio  $k$  debe ser el bucle \*\*más externo\*\*.

**Detección de Ciclos Negativos:** Si después de que el algoritmo termina, cualquier entrada  $D[i][i]$  (distancia de un nodo a sí mismo) es menor que cero, existe un ciclo negativo que involucra al nodo  $i$ .

**Complejidad:** \* **Tiempo:**  $O(V^3)$ , donde  $V$  es el número de vértices. \* **Espacio:**  $O(V^2)$  para la matriz de distancias.

### 3.0.8. Código C++ (Floyd-Warshall)

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int MAXN = 100;
5 const long long INF = 1e18;
6
7 long long D[MAXN + 1][MAXN + 1]; // Matriz de distancias
8
9 void floyd_marshall(int V) {
10    // 1. Inicialización (D[i][j] = peso de arista, D[i][i] = 0)
11    // Esto generalmente se hace al leer la entrada del grafo
12
13    // 2. Aplicación de la fórmula DP
14    for (int k = 1; k <= V; ++k) { // Vértice intermedio debe ser el bucle externo
15        for (int i = 1; i <= V; ++i) {
16            for (int j = 1; j <= V; ++j) {
17                if (D[i][k] != INF && D[k][j] != INF) {
18                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
19                }
20            }
21        }
22    }
23
24    // 3. Detección de Ciclos Negativos
25    /*
26        for (int i = 1; i <= V; ++i) {
27            if (D[i][i] < 0) {
28                // Ciclo negativo detectado
29            }
30        }
31    */
32}
33
34
35 int main() {
36     ios_base::sync_with_stdio(false);
37     cin.tie(NULL);
38
39     int V = 4; // Ejemplo: 4 vértices
40
41     // Inicializar matriz D con pesos de aristas y INF para no-adyacentes
42     for (int i = 1; i <= V; ++i) {
```

```

43     for (int j = 1; j <= V; ++j) {
44         if (i == j) D[i][j] = 0;
45         else D[i][j] = INF;
46     }
47 }
48
49 // Ejemplo de aristas: (u, v, w)
50 D[1][2] = 5;
51 D[1][4] = 10;
52 D[2][3] = 3;
53 D[3][4] = 1;
54 D[4][1] = 7;
55
56 floyd_marshall(V);
57
58 // Imprimir resultado
59 cout << "Matriz de distancias minimas:" << endl;
60 for (int i = 1; i <= V; ++i) {
61     for (int j = 1; j <= V; ++j) {
62         if (D[i][j] == INF) cout << "INF\t";
63         else cout << D[i][j] << "\t";
64     }
65     cout << endl;
66 }
67
68 return 0;
69 }
```

Listing 9: Implementación de Floyd-Warshall (Todos los pares)

## 4. Backtracking

### Ejercicio: Siguiente Permutación Lexicográfica (Usando `next_permutation`)

#### 4.0.1. Resumen: Búsqueda de la Siguiente Permutación

El problema requiere encontrar la \*\*permutación más pequeña\*\* (lexicográficamente) que sea estrictamente \*\*mayor\*\* que la secuencia de dígitos dada ( $X$ ).

Este problema se resuelve de manera eficiente en  $O(N)$  (una vez ordenado) mediante un algoritmo de búsqueda lexicográfica, que puede considerarse una forma optimizada de búsqueda que evita generar y probar todas las permutaciones, una tarea que sería  $O(N!)$ .

**Justificación Conceptual (Backtracking):** Aunque la función `std::next_permutation` es determinística y muy eficiente, el concepto general de buscar la siguiente combinación válida y probar una rama del árbol de búsqueda para modificar la secuencia de dígitos se relaciona con el \*\*Backtracking/Búsqueda Sistemática\*\*.

**Función `std::next_permutation`** La función `std::next_permutation(first, last)` de la STL de C++ reordena los elementos en el rango `[first, last]` a su \*\*siguiente permutación lexicográfica\*\* (el siguiente arreglo ordenado en un diccionario).

- **Funcionamiento Clave:** Modifica la secuencia \*\*in situ\*\* (en el lugar).
- **Valor de Retorno:**
  - Devuelve `true` si la siguiente permutación fue encontrada y el arreglo fue modificado.
  - Devuelve `false` si el arreglo ya estaba en su última permutación (ordenado de forma descendente) y lo reordena a la primera (ordenado de forma ascendente).

**Mecanismo Interno (Algoritmo  $O(N)$ ):** El algoritmo interno para encontrar la siguiente permutación es altamente eficiente, trabajando en tiempo lineal  $O(N)$  después de una posible ordenación inicial:

1. **Buscar el Punto de Quiebre ( $k$ ):** Se recorre el arreglo de derecha a izquierda para encontrar el índice más grande  $k$  tal que  $\text{digits}[k] < \text{digits}[k + 1]$ . Este es el punto donde la secuencia deja de estar en orden descendente y es el punto más a la derecha que se puede incrementar. Si no se encuentra  $k$  (la secuencia está completamente ordenada descendentemente), la permutación es la última posible, y el algoritmo retorna `false`.
2. **Buscar el Elemento de Intercambio ( $l$ ):** Se encuentra el índice más grande  $l$  tal que  $\text{digits}[k] < \text{digits}[l]$ . Este elemento  $\text{digits}[l]$  es el más pequeño en la sub-secuencia derecha que es mayor que  $\text{digits}[k]$ .
3. **Intercambiar:** Se intercambian  $\text{digits}[k]$  y  $\text{digits}[l]$ .
4. **Revertir (Ordenar):** Se revierte (invierte) la sub-secuencia después del índice  $k$  ( $\text{digits}[k + 1]$  hasta el final). Este paso garantiza que el resultado sea la **siguiente** permutación más pequeña.

**Complejidad:** \* **Tiempo:**  $O(N)$  por cada llamada a `next_permutation` (una vez que la secuencia está ordenada). \* **Pre-requisito:** Para encontrar la **\*siguiente\*** permutación **\*más pequeña\*** que la entrada, la secuencia inicial de dígitos de  $X$  debe estar **ordenada ascendentemente** antes de la primera llamada. \*\*Nota:\*\* En el código se asume que  $X$  se ingresa y se trabaja sobre la representación inicial, la función por sí misma siempre devuelve la **\*siguiente\*** en el orden lexicográfico, que es lo que pide el problema.

#### 4.0.2. Código C++ (Usando `next_permutation`)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 void solve() {
5     string x;
6     cin >> x;
7
8     vector<char> digits(x.begin(), x.end());
9
10    if (next_permutation(digits.begin(), digits.end())) {
11        for(int i = 0; i < digits.size(); i++){
12            cout << digits[i];
13        }
14        cout << endl;
15    } else {
16        cout << 0 << endl;
17    }
18 }
19
20 int main() {
21     ios::sync_with_stdio(false);
22     cin.tie(nullptr);
23
24     solve();
25 }
```

Listing 10: Uso de `next_permutation` para encontrar el siguiente número

## Ejercicio: Generación de Todas las Permutaciones

#### 4.0.3. Generación de Todas las Permutaciones

Este problema consiste en generar y listar todas las posibles permutaciones de una cadena de caracteres dada, en orden lexicográfico. Utiliza el algoritmo `std::next_permutation` dentro de un bucle `do-while` para generar todas las  $N!$  permutaciones de forma eficiente.

**Complejidad:** \* **Tiempo:**  $O(N \cdot N!)$ , ya que se generan  $N!$  permutaciones y cada una cuesta  $O(N)$ . \* **Espacio:**  $O(N \cdot N!)$ , para almacenar todas las permutaciones.

#### 4.0.4. Código C++ (Generación de Permutaciones)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll long long;
4
5 int main() {
6     string cadena;
7     cin >> cadena;
8     vector<string> cadenas;
9
10    // El orden inicial es crucial para next_permutation
11    sort(cadena.begin(), cadena.end());
12
13    do {
14        cadenas.push_back(cadena);
15    }
16    while (next_permutation(cadena.begin(), cadena.end()));
17
18    cout << cadenas.size() << endl;
19    for (ll i = 0; i < cadenas.size(); i++){
20        cout << cadenas [i] << endl;
21    }
22
23    return 0;
24 }
```

Listing 11: Generar todas las permutaciones de una cadena (STL)

## 5. DP

### Ejercicio: Problema de la Mochila (Knapsack 0/1) con Reconstrucción

#### 5.0.1. Resumen: Problema de la Mochila 0/1 (Knapsack)

El Problema de la Mochila 0/1 (Knapsack 0/1) busca maximizar el valor total de los ítems seleccionados que pueden caber en una mochila con capacidad  $C$ . Cada ítem  $i$  tiene un tamaño  $s_i$  y un valor  $v_i$ , y solo se puede tomar **una vez** (0 o 1).

**Relación de Recurrencia (Programación Dinámica):** La tabla  $A[i][c]$  almacena el valor máximo que se puede obtener considerando los primeros  $i$  ítems con capacidad  $c$ .

$$A[i][c] = \begin{cases} A[i - 1][c] & \text{si } s_i > c \text{ (No cabe)} \\ \max(A[i - 1][c], A[i - 1][c - s_i] + v_i) & \text{si } s_i \leq c \text{ (Máx. entre no tomar o tomar)} \end{cases}$$

**Complejidad:** \* **Tiempo:**  $O(N \cdot C)$ , donde  $N$  es el número de ítems y  $C$  es la capacidad de la mochila.  
\* **Espacio:**  $O(N \cdot C)$  para la matriz de DP.

**Reconstrucción:** La solución es reconstruida iterando la tabla  $A$  hacia atrás, desde  $A[n][C]$ . En el estado  $A[i][c]$ , se elige el elemento  $i$  si el valor es el resultado de haberlo incluido (comparando  $A[i][c]$  con  $A[i - 1][c]$ ).

#### 5.0.2. Código C++ (Knapsack 0/1 con Reconstrucción)

```

1 /*
2 Este es el Knapsack 0/1, es decir, no permite la repetición ni fragmentación de los
3 objetos
4 */
5 #include <bits/stdc++.h>
```

```

6   using namespace std;
7
8 # define ll long long
9 # define ld long double
10 # define endl '\n'
11
12 int knapsack(vector<int>& sizes, vector<int>& values, int C){
13     int n = sizes.size();
14     // Matriz bidimensional
15     vector<vector<int>> A(n+1, vector<int>(C+1, 0)); // Ese ,0 inicializa la columna en
16     0
17
18     for (int i = 1; i <= n; i++){
19         int si = sizes[i-1]; // Esto es distinto a la teoria
20         int vi = values[i-1]; // Esto tambien es distinto a la teoria
21         for (int c = 0; c <= C; c++){
22             if (si > c){
23                 A[i][c] = A[i-1][c];
24             } else {
25                 A[i][c] = max( A[i-1][c] , A[i-1][c-si] + vi);
26             }
27         }
28     }
29
30     return A[n][C];
31 }
32
33 vector<int> knapsackReconstruction(vector<int>& sizes, vector<int>& values, int C){
34     // --- Primero el Knapsack normal ---
35     int n = sizes.size();
36     // Matriz bidimensional
37     vector<vector<int>> A(n+1, vector<int>(C+1, 0));
38
39     // Inicializamos la fila y columna en ceros
40     for (int i = 0; i < C; i++) A[0][i] = 0; // Columna en cero
41
42     for (int i = 1; i <= n; i++){
43         int si = sizes[i-1]; // Esto es distinto a la teoria
44         int vi = values[i-1]; // Esto tambien es distinto a la teoria
45         for (int c = 0; c <= C; c++){
46             if (si > c){
47                 A[i][c] = A[i-1][c];
48             } else {
49                 A[i][c] = max( A[i-1][c] , A[i-1][c-si] + vi);
50             }
51         }
52     }
53
54     // return A[n][C]; // Descomentar para Knapsack normal
55
56     // Ya con la matriz A generada, vemos la reconstruccion
57     vector<int> S;
58     int c = C;
59
60     for (int i = n; i >= 1; i--){
61         int si = sizes[i-1]; // Esto no es parte de la teoria
62         int vi = values[i-1]; // Esto no es parte de la teoria
63
64         if (si <= c && A[i-1][c-si] + vi >= A[i-1][c]){
65             S.push_back(i);
66             c = c - si;
67         }
68     }
69
70     return S;
71 }
72
73 void solve(){
74     int C,n;
75     while(cin >> C >> n){
76         vector<int> values;
77         vector<int> sizes;

```

```

78     int v, s;
79
80     for (int i = 0; i < n; i++){
81         cin >> v >> s;
82         values.push_back(v);
83         sizes.push_back(s);
84     }
85
86     // cout << "Knapsack value solution: " << knapsack(sizes, values, C) << endl;
87
88     vector<int> knapsack_solution = knapsackReconstruction(sizes, values, C);
89
90     sort(knapsack_solution.begin(), knapsack_solution.end()); // No es necesario pero
91     mejor
92
93     cout << knapsack_solution.size() << endl; // Tamaño de la lista de elementos
94     elegidos
95
96     for (auto item: knapsack_solution){
97         if (item != knapsack_solution[knapsack_solution.size() - 1]){
98             cout << item - 1 << " ";
99         } else {
100             cout << item - 1 << endl;
101         }
102     }
103 }
104
105 int main() {
106     ios_base::sync_with_stdio(false);
107     cin.tie(nullptr);
108
109     solve();
110
111     return 0;
112 }
```

Listing 12: Knapsack 0/1: Cálculo de Valor y Reconstrucción

## Ejercicio: Maximum Weight Independent Set (MWIS) en Grafo de Camino

### 5.0.3. Maximum Weight Independent Set (MWIS) en Grafo de Camino

El problema MWIS en un \*\*grafo de camino\*\* busca encontrar un subconjunto de vértices (elementos de un array  $A$ ) tal que \*\*ningún par de vértices sea adyacente\*\* y la suma de sus pesos (valores) sea **máxima**.

**Fórmula DP:** Sea  $DP[i]$  el peso máximo del conjunto independiente usando los elementos de  $A$  hasta el índice  $i$ .

$$DP[i] = \max(DP[i - 1], A[i] + DP[i - 2])$$

**Reconstrucción:** La función `wisReconstruction` determina qué elementos fueron incluidos en el conjunto óptimo, yendo de atrás hacia adelante ( $i = n - 1$ ). En cada paso, se compara si el valor máximo  $DP[i]$  vino de **no tomar**  $A[i]$  ( $DP[i - 1]$ ) o de **tomar**  $A[i]$  ( $A[i] + DP[i - 2]$ ).

**Complejidad:** \* **Tiempo:**  $O(N)$  (un único bucle lineal para el cálculo de DP y otro para la reconstrucción). \* **Espacio:**  $O(N)$  (para el array de DP).

### 5.0.4. Código C++ (MWIS con Reconstrucción)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 # define int long long
4
5 vector<int> wisReconstruction(vector<int>& A){
6     int n = A.size();
7     if (n == 0) return {};
8     if (n == 1) return {0};
9
10    vector<int> dp(n);
11    dp[0] = A[0];
12    dp[1] = max(A[0], A[1]);
13
14    for (int i = 2; i < n; i++){
15        dp[i] = max(dp[i-1], A[i] + dp[i-2]);
16    }
17
18    // Reconstrucción correcta
19    vector<int> S;
20    int i = n - 1;
21
22    while (i >= 1){
23        if (dp[i-1] >= A[i] + (i >= 2 ? dp[i-2] : 0)){
24            i = i - 1;           // no tomo i
25        } else {
26            S.push_back(i); // tomo i
27            i = i - 2;
28        }
29    }
30
31    if (i == 0){
32        S.push_back(0);
33    }
34
35    // opcional: invertir para tenerlos en orden natural
36    reverse(S.begin(), S.end());
37
38    return S;
39}
40
41 signed mwis(vector<int>& A){
42     int n = A.size();
43     if (n == 0) return 0;
44     if (n == 1) return A[0];
45     vector<int> dp(n);
46
47     dp[0] = A[0];
48     dp[1] = max(A[0], A[1]);
49
50     for (int i = 2; i < n; i++){
51         dp[i] = max(dp[i-1], A[i] + dp[i-2]);
52     }
53
54     return dp[n-1];
55 }
56
57
58 int main(){
59     vector<int> values = {1,2,4,1,7,8,3};
60     cout << mwis(values) << endl;
61
62     vector<int> res = wisReconstruction(values);
63     for (auto n: res){
64         cout << n << " ";
65     }
66 }
```

Listing 13: Maximum Weight Independent Set (MWIS) con Reconstrucción

## Ejercicio: Cantidad Mínima de Monedas (Coin Change Minimum)

### 5.0.5. Cantidad Mínima de Monedas (Coin Change Minimum)

El problema busca encontrar el número mínimo de monedas requeridas para alcanzar un monto objetivo  $X$ , dado un conjunto de denominaciones  $C_j$  y suministro ilimitado.

Fórmula DP:

$$DP[i] = 1 + \min_j \{DP[i - C_j]\} \quad \text{si } i \geq C_j$$

Complejidad: \* Tiempo:  $O(N \cdot X)$ , donde  $N$  es el número de monedas y  $X$  el monto objetivo. \* Espacio:  $O(X)$ .

### 5.0.6. Código C++ (Cantidad Mínima de Monedas)

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 ll inf = 1e9; // Definición aproximada en el contexto del PDF
4 using ll long long;
5
6
7 int main() {
8     ll n, x;
9     cin >> n >> x;
10
11    vector<ll> dp(x + 1, inf);
12    vector<ll> coins(n);
13
14    for (ll i=0; i < n; i++) {
15        cin >> coins[i];
16    }
17
18    dp[0] = 0;
19
20    for (ll i=1; i <= x; i++) { // Corregí el <= X que faltaba en el bucle del PDF
21        for (ll j=0; j < n; j++)
22            if (i >= coins[j])
23            {
24                dp[i] = min(dp[i], dp[i - coins[j]] + 1);
25            }
26    }
27
28    if (dp[x] == inf)
29    {
30        cout << -1 << endl; // El PDF imprime 1, pero -1 o "IMPOSSIBLE" es más estándar
31    } else {
32        cout << dp[x] << endl;
33    }
34
35    return 0;
36 }
```

Listing 14: Coin Change: Mínimo número de monedas (DP)

## Ejercicio: Combinaciones de Monedas (El Orden Importa)

### 5.0.7. Combinaciones de Monedas (El Orden Importa)

Busca el número total de formas de formar el monto objetivo  $X$ , donde la secuencia de monedas usadas sí importa. La estrategia es iterar sobre los montos (exterior) y luego sobre las monedas (interior).

Fórmula DP:

$$DP[i] = \sum_j DP[i - C_j] \quad \text{si } i \geq C_j$$

**Complejidad:** \* Tiempo:  $O(N \cdot X)$ . \* Espacio:  $O(X)$ .

#### 5.0.8. Código C++ (Combinaciones - Orden Importa)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define mod 1000000007
5 #define endl "\n"
6
7 int main() {
8     int n, x;
9     cin >> n >> x;
10
11    int dp[x + 1] = {0};
12    int coins[n];
13
14    for (int i=0; i < n; i++) {
15        cin >> coins[i];
16    }
17
18    dp[0] = 1;
19
20    for (int i=1; i <= x; i++) {
21        for (int j=0; j<n; j++) {
22            if (i >= coins[j]) {
23                dp[i] = (dp[i] + dp[i - coins[j]]) % mod;
24            }
25        }
26    }
27
28    cout << dp[x] << endl;
29    return 0;
30}
31

```

Listing 15: Coin Change: Combinaciones donde el orden importa (DP)

## Ejercicio: Combinaciones de Monedas (El Orden No Importa)

#### 5.0.9. Combinaciones de Monedas (El Orden No Importa)

Busca el número total de formas de formar el monto objetivo  $X$ , donde la secuencia de monedas usadas no importa (ej., 1+2 es igual a 2+1). Esto se logra iterando sobre las monedas (exterior) y luego sobre los montos (interior), para evitar el doble conteo.

**Fórmula DP:**

$$DP[j] = DP[j] + DP[j - C_i]$$

**Complejidad:** \* Tiempo:  $O(N \cdot X)$ . \* Espacio:  $O(X)$ .

#### 5.0.10. Código C++ (Combinaciones - Orden No Importa)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define endl "\n"
5 #define mod 1000000007
6
7 int main() {
8     int n, x;
9     cin >> n >> x;
10
11    int dp[x + 1] = {0};
12
13    for (int i=1; i <= x; i++) {
14        for (int j=0; j<n; j++) {
15            if (i >= coins[j]) {
16                dp[i] = (dp[i] + dp[i - coins[j]]) % mod;
17            }
18        }
19    }
20
21    cout << dp[x] << endl;
22    return 0;
23}
24

```

```

13     11 coins [n];
14
15     for (11 i=0; i < n; i++) {
16         cin >> coins [i];
17     }
18
19     dp [0] = 1;
20
21     // Bucle externo sobre las monedas (i)
22     for (11 i=0; i < n; i++) {
23         // Bucle interno sobre los montos (j)
24         for (11 j = coins [i]; j <= x; j++) {
25             dp[j] = (dp[j] + dp[j - coins [i]]) % mod;
26         }
27     }
28
29     cout << dp [x] << endl;
30     return 0;
31 }
```

Listing 16: Coin Change: Combinaciones donde el orden no importa (DP)

## Ejercicio: Combinaciones con Reconstrucción Única (Problema del Camarero)

### 5.0.11. Combinaciones con Reconstrucción Única (Problema del Camarero)

Este problema es una variación del problema de Combinaciones de Monedas, pero con tres posibles resultados: **Impossible** (0 formas), **Ambiguous** (2 o más formas), o **\*\*Solución Única\*\*** (1 forma). La clave es usar la DP para contar las formas (limitado a  $\leq 2$ ) y, si la cuenta es 1, usar una segunda pasada de DP para almacenar los índices y reconstruir la solución.

**Estrategia de Doble DP:** 1. **Contador (countWays):** Calcula  $DP[x]$  = número de formas de sumar  $x$ , truncando a 2 (0, 1, o 2+). 2. **Reconstructor (reconstruct):** Usa una DP similar, pero solo propaga si se encuentra **exactamente una** forma de alcanzar el monto anterior, y almacena el índice del último ítem usado en `last[x]`.

**Fórmula DP (Contador):** Similar a Combinaciones (Orden No Importa), ya que el orden de los ítems en el pedido final no importa.

$$DP[x] = DP[x] + DP[x - p] \quad \text{Luego : } DP[x] = \min(DP[x], 2)$$

**Fórmula DP (Reconstructor):** Propaga solo si la única forma de alcanzar  $x$  es a través de un único camino previo:

$$\text{Si } DP[x - p] = 1 \text{ y } DP[x] = 0 \text{ entonces : } DP[x] = 1, \quad \text{last}[x] = i$$

**Complejidad:** \* **Tiempo:**  $O(N \cdot C)$ , donde  $N$  es el número de ítems del menú y  $C$  es el costo total máximo del pedido (30000). \* **Espacio:**  $O(C)$ .

### 5.0.12. Código C++ (Reconstrucción Única)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 vector<int> countWays(const vector<int>& prices, int C) {
5     vector<int> dp(C+1, 0);
6     dp[0] = 1;
7
8     for (int i = 0; i < prices.size(); i++) {
9         int p = prices[i];
10        for (int x = p; x <= C; x++) {
```

```

11     dp[x] += dp[x - p];
12     if (dp[x] > 2) dp[x] = 2; // solo interesa 0,1,2+
13   }
14 }
15 return dp;
16 }

17 vector<int> reconstruct(const vector<int>& prices, int C) {
18   int n = prices.size();
19
20   // dp[i] = 1 si existe EXACTAMENTE una forma de armar i
21   vector<int> dp(C+1, 0);
22   vector<int> last(C+1, -1); // guarda el índice del último ítem usado
23
24   dp[0] = 1;
25
26   for (int i = 0; i < n; i++) {
27     int p = prices[i];
28     for (int x = p; x <= C; x++) {
29       if (dp[x - p] == 1 && dp[x] == 0) {
30         dp[x] = 1;
31         last[x] = i;
32       }
33     }
34   }
35 }

36 vector<int> sol;
37 int cur = C;
38
39 while (cur > 0) {
40   int i = last[cur];
41   sol.push_back(i+1);
42   cur -= prices[i];
43 }
44
45 return sol;
46 }

47 }

48 int main() {
49   ios::sync_with_stdio(false);
50   cin.tie(nullptr);
51
52   int n;
53   cin >> n;
54   vector<int> prices(n);
55   for (int i = 0; i < n; i++) cin >> prices[i];
56
57   int m;
58   cin >> m;
59   vector<int> orders(m);
60   for (int i = 0; i < m; i++) cin >> orders[i];
61
62   for (int s : orders) {
63
64     vector<int> ways = countWays(prices, s);
65
66     if (ways[s] == 0) {
67       cout << "Impossible\n";
68       continue;
69     }
70     if (ways[s] >= 2) {
71       cout << "Ambiguous\n";
72       continue;
73     }
74   }
75
76   // reconstrucción única
77   vector<int> sol = reconstruct(prices, s);
78   sort(sol.begin(), sol.end());
79
80   for (int i = 0; i < sol.size(); i++) {
81     if (i) cout << " ";
82     cout << sol[i];
83   }

```

```

84         cout << "\n";
85     }
86
87     return 0;
88 }
```

Listing 17: Combinaciones y Reconstrucción Única

## Ejercicio: Manejo de Entrada Mixta (getline y stringstream)

### 5.0.13. Manejo de Entrada Mixta (getline y stringstream)

El problema central es leer datos donde el primer campo (un nombre de ítem) contiene **espacios en blanco** y es seguido por números en la misma línea, mientras que las líneas anteriores fueron leídas con cin.

**Solución Clave:** 1. **Consumir el \n:** Se utiliza una primera llamada a `getline(cin, linea)` después del último cin para consumir el salto de línea pendiente. 2. **Leer la Línea Completa:** Se usa `getline` para leer toda la línea de entrada, incluyendo el nombre con espacios. 3. **Separar (Parsing):** Se usa `stringstream` para procesar las partes de la línea. Se lee cada palabra en un vector. 4. **Extraer Números:** Se extraen los números (que deben ser las últimas palabras) del vector usando `stoi`. El resto del vector es el nombre.

La lógica principal del código resuelve un problema \*\*Knapsack 0/1\*\*, donde los índices calóricos (`ics`) son los tamaños y los índices de sabrosidad (`isos`) son los valores, con una capacidad total  $C$ .

**Complejidad:** \* **Tiempo:**  $O(N \cdot C)$  para el Knapsack (donde  $N$  es el número de ítems y  $C$  es la capacidad), más  $O(N \cdot L)$  para el parsing (donde  $L$  es la longitud promedio de la línea). \* **Espacio:**  $O(N \cdot C)$  para la DP.

### 5.0.14. Código C++ (Knapsack con Parsing de Línea)

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  # define ll long long
5  # define ld long double
6  # define endl '\n'
7
8  int knapsack(vector<int>& sizes, vector<int>& values, int C){
9      int n = sizes.size();
10     vector<vector<int>> A(n+1, vector<int>(C+1, 0));
11
12     for (int i = 1; i <= n; i++){
13         int si = sizes[i-1];
14         int vi = values[i-1];
15         for (int c = 0; c <= C; c++){
16             if (si > c){
17                 A[i][c] = A[i-1][c];
18             } else {
19                 A[i][c] = max(A[i-1][c], A[i-1][c-si] + vi);
20             }
21         }
22     }
23
24     return A[n][C];
25 }
26
27 int main (){
28     ios_base::sync_with_stdio (false);
29     cin.tie(nullptr);
30
31     int n, C;
32     cin >> n >> C;
33 }
```

```

34     vector<string> nombres(n);
35     vector<int> ics(n); // Indices caloricos
36     vector<int> isos(n); // Indices de sabrosidad de osos
37
38     string linea;
39     getline(cin, linea); // leer el salto de linea después de cin >> n >> C
40
41     for (int i = 0; i < n; i++) {
42         // EJEMPLO DE INPUT: hamburguesa vegana con lechuga 10 3
43         getline(cin, linea); // leer linea completa
44         stringstream ss(linea);
45
46         string nombre;
47         int ic, iso;
48
49         // Leer todas las palabras del nombre menos los números
50         vector<string> partes;
51         string temp;
52         while (ss >> temp) partes.push_back(temp);
53
54         // Ahora partes = {'hamburguesa', 'vegana', 'con', 'lechuga', '10', '3'}
55         // Los últimos dos son números
56         iso = stoi(partes.back()); partes.pop_back();
57         ic = stoi(partes.back()); partes.pop_back();
58
59         // El nombre es todo lo demás unido
60         nombre = partes[0];
61         for (int k = 1; k < partes.size(); k++)
62             nombre += " " + partes[k];
63
64         nombres[i] = nombre;
65         ics[i] = ic;
66         isos[i] = iso;
67     }
68
69     int respuesta = knapsack(ics, isos, C);
70
71     cout << respuesta << endl;
72
73     return 0;
74 }
```

Listing 18: Uso de getline y stringstream para entrada con espacios