Resumen de Algoritmos y Complejidad

Apuntes para la Prueba

20 de noviembre de 2025

Índice

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

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

# define ll long long
# define ld long double
# define endl '\n'

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
  return 0;
}
```

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)

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

# define ll long long
# define ld long double
# define endl '\n'

struct Edge{
  int n1;
  int n2;
  int w;
};

bool custom_compare_min(Edge a, Edge b){
  return a.w < b.w;
}

bool custom_compare_max(Edge a, Edge b){
  return a.w > b.w;
}

struct union_find{
  vector<int> e;
  union_find(int n) {e.assign(n, -1);}

  int findSet(int x){
    return (e[x] < 0 ? x : e[x] = findSet(e[x]));
  }

  bool sameSet(int x, int y) {return findSet(x) == findSet(y);}

  int size(int x){return -e[findSet(x)];}

  bool unionSet(int x, int y){
    x = findSet(x), y = findSet(y);
    if (x == y) return 0;
    if (e[x] > e[y]) swap(x,y);
    e[x] += e[y];
    e[y] = x;
    return 1;
  }
};

pair<int, vector<pair<int,int>>> minSpanningTree(vector<Edge> v){
  int n = v.size();
  sort(v.begin(), v.end(), custom_compare_min);
  union_find sets(n);
  vector<pair<int,int>> output;

  int contador = 0;

  for (int i = 0; i < v.size(); i++){
    Edge nodo = v[i];
    if (sets.sameSet(nodo.n1, nodo.n2) == false){
      sets.unionSet(nodo.n1, nodo.n2);
      output.push_back({nodo.n1+1, nodo.n2+1});
      contador += nodo.w;
      if (output.size() == n-1) break;
    }
  }

  sort(output.begin(), output.end());

  return {contador, output};
}

pair<int, vector<pair<int,int>>> maxSpanningTree(vector<Edge> v){
  int n = v.size();
  sort(v.begin(), v.end(), custom_compare_max);
  union_find sets(n);
  vector<pair<int,int>> output;
```

```
72
73      int contador = 0;
74
75      for (int i = 0; i < v.size(); i++){
76        Edge nodo = v[i];
77        if (sets.sameSet(nodo.n1, nodo.n2) == false){
78          sets.unionSet(nodo.n1, nodo.n2);
79          output.push_back({nodo.n1+1, nodo.n2+1});
80          contador += nodo.w;
81          if (output.size() == n-1) break;
82        }
83      }
84
85      sort(output.begin(), output.end());
86
87      return {contador, output};
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: Mínima Distancia para Iluminar (Faroles)

2.0.3. 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:** $O(n)$.

2.0.4. Código C++ (Faroles)

```
1   #include <bits/stdc++.h>
2   using namespace std;
3   using 11 long long;
4
```

```cpp
int main() {
  ll n, l;
  cin >> n >> l;
  vector<double> pos(n);
  for (ll i=0; i < n; i++) {
    cin >> pos[i];
  }
  sort(pos.begin(), pos.end());
  double d=0;
  for (ll i=0; i<n - 1; i++) {
    d=max(d, (pos[i + 1] - pos[i]) / 2);
  }
  d=max(d, pos[0]);
  d=max(d, l - pos[n-1]);
  cout << fixed << setprecision (10) << d << endl;
  return 0;
}
```

Listing 3: Mínima distancia requerida para iluminar una calle

Ejercicio: Construir Número Máximo con Presupuesto (Cercas)

2.0.5. 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 ($C_{\text{mín}}$), y 2) luego, reemplazar los dígitos $D_{\text{mín}}$ 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:** $O(L)$.

2.0.6. Código C++ (Cercas)

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

int main() {
  ll v;
  cin >> v;
  vector<ll> nums (9);
  for (ll i=0; i < 9; i++) {
    cin >> nums[i];
  }
  ll min_cost = *min_element(nums.begin(), nums.end());
  ll min_digit = min_element(nums.begin(), nums.end()) - nums.begin() + 1;

  if (min_cost > v) {
    cout << -1 << endl;
    return 0;
  }

  ll max_length = v / min_cost;
  ll remaining_paint = v % min_cost;

  string result(max_length, '0' + min_digit);

  for (ll d=9; d >= min_digit; d--) {
    for (ll i=0; i < max_length; i++) {
      ll cost_diff = nums[d-1] - min_cost;
      if (remaining_paint >= cost_diff) {
        result [i] = '0' + d;
        remaining_paint -= cost_diff;
      } else {
        break;
      }
    }
```

```
    }

    cout << result << endl;
    return 0;
}
```

Listing 4: 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)

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


int main() {
  ll n, x, cont = 0, suma = 0;
  cin >> n >> x;
  deque <ll> dq;
  for (ll i=0; i < n; i++) {
    ll num;
    cin >> num;
    dq.push_back(num);
    suma += num;
    while (suma > x) {
      suma -= dq.front();
      dq.pop_front();
    }

    if (suma == x) {
      cont++;
      // Este bloque adicional maneja el caso donde solo queremos el número de subarreglos
      // y no queremos que se solapen si hubiera más de una solución.
      // Para contar todos los subarreglos, la condición de avance (pop_front) puede variar.
      suma -= dq.front();
      dq.pop_front();
    }
  }

  cout << cont << endl;
  return 0;
}
```

Listing 5: 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)

```cpp
#include <bits/stdc++.h>
using namespace std;
using 11 long long;
int main() {
  string s;
  cin >> s;
  map<char, 11> m;
  for (11 i=0; i < s.size(); i++) {
    m[s[i]]++;
  }
  11 not_pair = 0;
  char odd_char = ' ';
  for (auto &it: m) {
    if (it.second % 2 != 0) {
      not_pair++;
      odd_char = it.first;
    }
  }
  if (not_pair > 1) {
    cout << "NO SOLUTION" << endl;
    return 0;
  }
  string half = "";
  for (auto &it: m) {
    half += string(it.second / 2, it.first);
  }
  string palindrome = half;
  if (not_pair == 1) {
    palindrome += odd_char;
  }
  reverse (half.begin(), half.end());
  palindrome += half;
  cout << palindrome << endl;
  return 0;
}
```

Listing 6: 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 (Fuente Única con Pesos Negativos)

Bellman-Ford encuentra el camino más corto desde un nodo fuente único a todos los demás nodos en un grafo dirigido. Es vital para grafos que contienen **aristas con pesos negativos**.

Estrategia: El algoritmo relaja (actualiza) repetidamente todas las aristas del grafo $V - 1$ veces. Este número de iteraciones garantiza que se encuentre el camino más corto, asumiendo que no hay ciclos negativos. Una iteración número V se usa para detectar si existe un **ciclo de peso negativo**. Si en la iteración V se puede realizar alguna relajación, existe un ciclo negativo alcanzable desde el origen.

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

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

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 las distancias.

3.0.6. Código C++ (Bellman-Ford)

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

const long long INF = 1e18; // Usar un valor grande para infinito

struct Edge {
  int u, v;
  long long weight;
};

// Retorna true si no hay ciclos negativos, false si se detecta uno.
bool bellman_ford(int V, const vector<Edge>& edges, int source, vector<long long>&
  dist) {
  dist.assign(V + 1, INF);
  dist[source] = 0;

  // 1. Relajar todas las aristas V - 1 veces
  for (int i = 0; i < V - 1; ++i) {
    bool relaxed = false;
    for (const auto& edge : edges) {
      if (dist[edge.u] != INF && dist[edge.u] + edge.weight < dist[edge.v]) {
        dist[edge.v] = dist[edge.u] + edge.weight;
        relaxed = true;
      }
    }
    // Optimización: Si no se relajó ninguna arista, ya encontramos los caminos
    if (!relaxed) break;
  }

  // 2. Detectar Ciclos Negativos (V-ésima relajación)
  for (const auto& edge : edges) {
    if (dist[edge.u] != INF && dist[edge.u] + edge.weight < dist[edge.v]) {
      // Ciclo negativo detectado
      return false;
    }
  }

  return true;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  int V, E; // V = número de vértices, E = número de aristas
  int source = 1; // Suponiendo el nodo fuente 1

  // Ejemplo de inicialización (sustituir con cin >> V >> E)
  V = 5;
  E = 8;

  vector<Edge> edges = {
    {1, 2, -1}, {1, 3, 4},
    {2, 3, 3}, {2, 4, 2},
    {2, 5, 2}, {4, 3, 5},
    {4, 2, 1}, {5, 4, -3}
  };

  vector<long long> dist;

  if (bellman_ford(V, edges, source, dist)) {
    cout << "Caminos minimos encontrados desde el nodo " << source << ":" << endl;
```

```
62      for (int i = 1; i <= V; ++i) {
63        if (dist[i] == INF) {
64          cout << "Nodo " << i << ": IMPOSIBLE" << endl;
65        } else {
66          cout << "Nodo " << i << ": " << dist[i] << endl;
67        }
68      }
69    } else {
70      cout << "Ciclo de peso negativo detectado. No se pueden garantizar caminos minimos
    ." << endl;
71    }
72
73    return 0;
74  }
```

Listing 7: Implementación de Bellman-Ford con detección de ciclos negativos

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.

$$D[i][j] = \text{mín}(D[i][j], D[i][k] + D[k][j])$$

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_warshall(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      }
```

```
    }

    // 3. Detección de Ciclos Negativos
    /*
    for (int i = 1; i <= V; ++i) {
      if (D[i][i] < 0) {
        // Ciclo negativo detectado
      }
    }
    */
  }


  int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int V = 4; // Ejemplo: 4 vértices

    // Inicializar matriz D con pesos de aristas y INF para no-adyacentes
    for (int i = 1; i <= V; ++i) {
      for (int j = 1; j <= V; ++j) {
        if (i == j) D[i][j] = 0;
        else D[i][j] = INF;
      }
    }

    // Ejemplo de aristas: (u, v, w)
    D[1][2] = 5;
    D[1][4] = 10;
    D[2][3] = 3;
    D[3][4] = 1;
    D[4][1] = 7;

    floyd_warshall(V);

    // Imprimir resultado
    cout << "Matriz de distancias minimas:" << endl;
    for (int i = 1; i <= V; ++i) {
      for (int j = 1; j <= V; ++j) {
        if (D[i][j] == INF) cout << "INF\t";
        else cout << D[i][j] << "\t";
      }
      cout << endl;
    }

    return 0;
  }
```

Listing 8: 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 `digits[k] <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 `digits[k] <digits[l]`. Este elemento `digits[l]` es el más pequeño en la sub-secuencia derecha que es mayor que `digits[k]`.
3. **Intercambiar:** Se intercambian `digits[k]` y `digits[l]`.
4. **Revertir (Ordenar):** Se revierte (invierte) la sub-secuencia después del índice k (`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)

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

void solve() {
  string x;
  cin >> x;

  vector<char> digits(x.begin(), x.end());

  if (next_permutation(digits.begin(), digits.end())) {
    for(int i = 0; i < digits.size(); i++){
      cout << digits[i];
    }
    cout << endl;
  } else {
    cout << 0 << endl;
  }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  solve();
}
```

Listing 9: 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)

```cpp
#include <bits/stdc++.h>
using namespace std;
using 11 long long;

int main() {
  string cadena;
  cin >> cadena;
  vector<string> cadenas;

  // El orden inicial es crucial para next_permutation
  sort(cadena.begin(), cadena.end());

  do {
    cadenas.push_back(cadena);
  }
  while (next_permutation(cadena.begin(), cadena.end()));

  cout << cadenas.size() << endl;
  for (11 i = 0; i < cadenas.size(); i++){
    cout << cadenas [i] << endl;
  }

  return 0;
}
```

Listing 10: 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)

```cpp
/*
Este es el Knapsack 0/1, es decir, no permite la repeticion ni fragmentacion de los objetos
*/

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

# define ll long long
# define ld long double
# define endl '\n'

int knapsack(vector<int>& sizes, vector<int>& values, int C){
  int n = sizes.size();
  // Matriz bidimensional
  vector<vector<int>> A(n+1, vector<int>(C+1, 0)); // Ese ,0 inicializa la columna en 0

  for (int i = 1; i <= n; i++){
    int si = sizes[i-1]; // Esto es distinto a la teoria
    int vi = values[i-1]; // Esto tambien es distinto a la teoria
    for (int c = 0; c <= C; c++){
      if (si > c){
        A[i][c] = A[i-1][c];
      } else {
        A[i][c] = max( A[i-1][c] , A[i-1][c-si] + vi);
      }
    }
  }

  return A[n][C];
}

vector<int> knapsackReconstruction(vector<int>& sizes, vector<int>& values, int C){
  // --- Primero el Knapsack normal ---
  int n = sizes.size();
  // Matriz bidimensional
  vector<vector<int>> A(n+1, vector<int>(C+1, 0));

  // Inicializamos la fila y columna en ceros
  for (int i = 0; i < C; i++) A[0][i] = 0; // Columna en cero

  for (int i = 1; i <= n; i++){
    int si = sizes[i-1]; // Esto es distinto a la teoria
    int vi = values[i-1]; // Esto tambien es distinto a la teoria
    for (int c = 0; c <= C; c++){
      if (si > c){
        A[i][c] = A[i-1][c];
      } else {
        A[i][c] = max( A[i-1][c] , A[i-1][c-si] + vi);
      }
    }
  }

  // return A[n][C]; // Descomentar para Knapsack normal

  // Ya con la matriz A generada, vemos la reconstruccion
  vector<int> S;
  int c = C;

  for (int i = n; i >= 1; i--){
    int si = sizes[i-1]; // Esto no es parte de la teoria
    int vi = values[i-1]; // Esto no es parte de la teoria

    if (si <= c && A[i-1][c-si] + vi >= A[i-1][c]){
      S.push_back(i);
```

```
        c = c - si;
      }
    }

    return S;
  }

  void solve(){
    int C,n;
    while(cin >> C >> n){
      vector<int> values;
      vector<int> sizes;

      int v,s;

      for (int i = 0; i < n; i++){
        cin >> v >> s;
        values.push_back(v);
        sizes.push_back(s);
      }

      // cout << "Knapsack value solution: " << knapsack(sizes, values, C) << endl;

      vector<int> knapsack_solution = knapsackReconstruction(sizes, values, C);

      sort(knapsack_solution.begin(), knapsack_solution.end()); // No es necesario pero
    mejor

      cout << knapsack_solution.size() << endl; // Tamaño de la lista de elementos
    elegidos

      for (auto item: knapsack_solution){
        if (item != knapsack_solution[knapsack_solution.size()-1]){
          cout << item-1 << " ";
        } else {
          cout << item-1 << endl;
        }
      }
      cout << endl;
    }
  }

  int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    solve();

    return 0;
  }
```

Listing 11: 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.

$$DP[i] = \text{máx}(DP[i-1], A[i] + DP[i-2])$$

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)

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

vector<int> wisReconstruction(vector<int>& A){
  int n = A.size();
  if (n == 0) return {};
  if (n == 1) return {0};

  vector<int> dp(n);
  dp[0] = A[0];
  dp[1] = max(A[0], A[1]);

  for (int i = 2; i < n; i++){
    dp[i] = max(dp[i-1], A[i] + dp[i-2]);
  }

  // Reconstrucción correcta
  vector<int> S;
  int i = n - 1;

  while (i >= 1){
    if (dp[i-1] >= A[i] + (i >= 2 ? dp[i-2] : 0)){
      i = i - 1;      // no tomo i
    } else {
      S.push_back(i); // tomo i
      i = i - 2;
    }
  }

  if (i == 0){
    S.push_back(0);
  }

  // opcional: invertir para tenerlos en orden natural
  reverse(S.begin(), S.end());

  return S;
}

int mwis(vector<int>& A){
  int n = A.size();
  if (n == 0) return 0;
  if (n == 1) return A[0];
  vector<int> dp(n);

  dp[0] = A[0];
  dp[1] = max(A[0], A[1]);

  for (int i = 2; i < n; i++){
    dp[i] = max(dp[i-1], A[i] + dp[i-2]);
  }

  return dp[n-1];
}


int main(){
  vector<int> values = {1,2,4,1,7,8,3};
  cout << mwis(values) << endl;

  vector<int> res = wisReconstruction(values);
  for (auto n: res){
```

```
      cout << n << " ";
    }
  }
```

Listing 12: 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)

```cpp
#include <bits/stdc++.h>
using namespace std;
11 inf = 1e9; // Definició'n aproximada en el contexto del PDF
using 11 long long;


int main() {
  11 n, x;
  cin >> n >> x;

  vector<11> dp(x + 1, inf);
  vector<11> coins (n);

  for (ll i=0; i < n; i++) {
    cin >> coins [i];
  }

  dp [0] = 0;

  for (11 i=1; i <= x; i++) { // Corregí el <= X que faltaba en el bucle del PDF
    for (11 j=0; j<n; j++)
    if (i >= coins [j])
    {
      dp[i] = min (dp[i], dp[i - coins [j]] + 1);
    }
  }

  if (dp[x] == inf)
  {
    cout << -1 << endl; // El PDF imprime 1, pero -1 o "IMPOSSIBLE" es más estándar
  } else {
    cout << dp [x] << endl;
  }

  return 0;
}
```

Listing 13: 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)

```cpp
#include <bits/stdc++.h>
using 11 long long;
using namespace std;

#define mod 1000000007
#define endl "\n"

int main() {
  11 n, x;
  cin >> n >> x;

  11 dp [x + 1] = {0};
  11 coins [n];

  for (11 i=0; i < n; i++) {
    cin >> coins [i];
  }

  dp [0] = 1;

  for (11 i=1; i <= x; i++) {
    for (11 j=0; j<n; j++) {
      if (i >= coins[j]) {
        dp[i] = (dp[i] + dp[i - coins [j]]) % mod;
      }
    }
  }

  cout << dp[x] << endl;
  return 0;
}
```

Listing 14: 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)

```cpp
#include <bits/stdc++.h>
using 11 long long;
using namespace std;

#define endl "\n"
#define mod 1000000007

int main() {
  11 n, x;
  cin >> n >> x;

  11 dp[x + 1] = {0};
  11 coins [n];

  for (11 i=0; i < n; i++) {
    cin >> coins [i];
  }

  dp [0] = 1;

  // Bucle externo sobre las monedas (i)
  for (ll i=0; i < n; i++) {
    // Bucle interno sobre los montos (j)
    for (11 j = coins [i]; j <= x; j++) {
      dp[j] = (dp[j] + dp[j - coins [i]]) % mod;
    }
  }

  cout << dp [x] << endl;
  return 0;
}
```

Listing 15: 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 ≤ 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:

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

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)

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

vector<int> countWays(const vector<int>& prices, int C) {
  vector<int> dp(C+1, 0);
  dp[0] = 1;

  for (int i = 0; i < prices.size(); i++) {
    int p = prices[i];
    for (int x = p; x <= C; x++) {
      dp[x] += dp[x - p];
      if (dp[x] > 2) dp[x] = 2; // solo interesa 0,1,2+
    }
  }
  return dp;
}

vector<int> reconstruct(const vector<int>& prices, int C) {
  int n = prices.size();

  // dp[i] = 1 si existe EXACTAMENTE una forma de armar i
  vector<int> dp(C+1, 0);
  vector<int> last(C+1, -1); // guarda el índice del último item usado

  dp[0] = 1;

  for (int i = 0; i < n; i++) {
    int p = prices[i];
    for (int x = p; x <= C; x++) {
      if (dp[x - p] == 1 && dp[x] == 0) {
        dp[x] = 1;
        last[x] = i;
      }
    }
  }

  vector<int> sol;
  int cur = C;

  while (cur > 0) {
    int i = last[cur];
    sol.push_back(i+1);
    cur -= prices[i];
  }

  return sol;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int n;
  cin >> n;
  vector<int> prices(n);
  for (int i = 0; i < n; i++) cin >> prices[i];

  int m;
  cin >> m;
  vector<int> orders(m);
  for (int i = 0; i < m; i++) cin >> orders[i];

  for (int s : orders) {

    vector<int> ways = countWays(prices, s);

    if (ways[s] == 0) {
      cout << "Impossible\n";
      continue;
    }
    if (ways[s] >= 2) {
```

```
        cout << "Ambiguous\n";
        continue;
      }

      // reconstruccion unica
      vector<int> sol = reconstruct(prices, s);
      sort(sol.begin(), sol.end());

      for (int i = 0; i < sol.size(); i++) {
        if (i) cout << " ";
        cout << sol[i];
      }
      cout << "\n";
    }

    return 0;
  }
```

Listing 16: 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)

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

# define ll long long
# define ld long double
# define endl '\n'

int knapsack(vector<int>& sizes, vector<int>& values, int C){
  int n = sizes.size();
  vector<vector<int>> A(n+1, vector<int>(C+1,0));

  for (int i = 1; i <= n; i++){
    int si = sizes[i-1];
    int vi = values[i-1];
    for (int c = 0; c <= C; c++){
      if (si > c){
        A[i][c] = A[i-1][c];
      } else {
        A[i][c] = max(A[i-1][c], A[i-1][c-si] + vi);
      }
    }
```

```
    }

    return A[n][C];
}

int main (){
    ios_base::sync_with_stdio (false);
    cin.tie(nullptr);

    int n, C;
    cin >> n >> C;

    vector<string> nombres(n);
    vector<int> ics(n); // Indices caloricos
    vector<int> isos(n); // Indices de sabrosidad de osos

    string linea;
    getline(cin, linea); // leer el salto de línea después de cin >> n >> C

    for (int i = 0; i < n; i++) {
        // EJEMPLO DE INPUT: hamburguesa vegana con lechuga 10 3
        getline(cin, linea);    // leer línea completa
        stringstream ss(linea);

        string nombre;
        int ic, iso;

        // Leer todas las palabras del nombre menos los números
        vector<string> partes;
        string temp;
        while (ss >> temp) partes.push_back(temp);

        // Ahora partes = {'hamburguesa', 'vegana', 'con', 'lechuga', '10', '3'}
        // Los últimos dos son números
        iso = stoi(partes.back()); partes.pop_back();
        ic  = stoi(partes.back()); partes.pop_back();

        // El nombre es todo lo demás unido
        nombre = partes[0];
        for (int k = 1; k < partes.size(); k++)
        nombre += " " + partes[k];

        nombres[i] = nombre;
        ics[i] = ic;
        isos[i] = iso;
    }

    int respuesta = knapsack(ics, isos, C);

    cout << respuesta << endl;

    return 0;
}
```

Listing 17: Uso de getline y stringstream para entrada con espacios