Apuntes Algoco

Marco Repetto y Oscar Ruiz

Dios te salve, Maria,
llena eres de gracia;
el Señor es contigo.
Bendita Tu eres
entre todas las mujeres,
y bendito es el fruto de tu vientre, Jesus.
Santa Maria, Madre de Dios,
ruega por nosotros, pecadores,
ahora y en la hora de nuestra muerte. Amen.

Contents

1 Plantillas

1.1 Estructura Base para Problemas de Algoritmos

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

#define INF numeric_limits<int>::max()
#define MINF numeric_limits<int>::min()
#define pb push_back
#define pp pop_back
#define all(v) (v).begin(), (v).end()
#define fast_io ios::sync_with_stdio(false); cin.tie(nullptr);

typedef long long ll;
typedef pair<int, int> pii;
```

1.2 Plantilla para Fuerza Bruta

```cpp
// Fuerza Bruta: Generar todas las combinaciones posibles
void fuerzaBruta(const vector<int>& nums) {
    int n = nums.size();
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> subset;
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) subset.pb(nums[i]);
        }
        // Procesar el subset generado
        for (int x : subset) cout << x << " ";
        cout << endl;
    }
}
```

1.3 Plantilla para Programacion Dinamica

```cpp
// Memoizacion (Top-Down)
map<pair<int, int>, int> memo;

int DP(int i, int j) {
    if (i == 0 || j == 0) return 0; // Caso base
    if (memo.find({i, j}) != memo.end()) return memo[{i, j}];

    // Supongamos que el problema requiere comparar dos indices
    int result = max(DP(i - 1, j), DP(i, j - 1));
    memo[{i, j}] = result;
    return result;
}

// Tabla Iterativa (Bottom-Up)
int DP_iterative(const vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[n][n];
}
```

1.4 Plantilla para Algoritmos Greedy

```cpp
int greedy(vector<int>& nums) {
    sort(all(nums)); // Ordenar para facilitar decisiones
```

```
    int result = 0;
    for (int i = 0; i < nums.size(); ++i) {
        result += nums[i]; // O cualquier logica greedy
    }
    return result;
}
```

1.5 Plantilla para Backtracking

```
void backtracking(vector<int>& nums, vector<int>& solution, vector<bool>& used) {
    if (solution.size() == nums.size()) {
        // Procesar solucion valida
        for (int x : solution) cout << x << " ";
        cout << endl;
        return;
    }
    for (int i = 0; i < nums.size(); ++i) {
        if (!used[i]) {
            used[i] = true;
            solution.pb(nums[i]);
            backtracking(nums, solution, used);
            solution.pop_back();
            used[i] = false;
        }
    }
}
```

1.6 Plantilla para Dividir y Conquistar

```
int divideYConquista(const vector<int>& nums, int l, int r) {
    if (l == r) return nums[l]; // Caso base
    int mid = l + (r - l) / 2;
    int left = divideYConquista(nums, l, mid);
    int right = divideYConquista(nums, mid + 1, r);
    return max(left, right); // Combinar resultados
}
```

2 Estructuras de datos

2.1 std::array

Contenedor de tamaño fijo.

2.1.1 Formas de definirlo:

- Syntaxis genérica: `type name [size];`
- Inicializar con ceros: `int baz [5] = { };`
- Inicializar con valores: `int foo [5] = { 16, 2, 77, 40, 12071 };`

2.2 std::vector

Contenedor dinámico de tamaño variable.

2.2.1 Formas de definirlo:

- Syntaxis genérica: `std::vector<type> name;`
- Inicializar con ceros: `std::vector<int> baz (5);`
- Inicializar con valores: `std::vector<int> foo { 16, 2, 77, 40, 12071 };`

2.2.2 Funciones miembro:

- `push_back(value)`: Agrega un elemento al final [O(1)].
- `pop_back()`: Elimina el último elemento [O(1)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].
- `front()`: Retorna el primer elemento [O(1)].
- `back()`: Retorna el último elemento [O(1)].
- `insert(iterator, value)`: Inserta un elemento en la posición indicada [O(n)].

2.3 std::list

Contenedor de secuencia que permite inserciones y eliminaciones en cualquier posición en tiempo constante.

2.3.1 Formas de definirlo:

- Syntaxis genérica: `std::list<type> name;`

2.3.2 Funciones miembro:

- `push_back(value)`: Agrega un elemento al final [O(1)].
- `push_front(value)`: Agrega un elemento al principio [O(1)].
- `pop_back()`: Elimina el último elemento [O(1)].
- `pop_front()`: Elimina el primer elemento [O(1)].
- `size()`: Retorna la cantidad de elementos [O(n)].
- `empty()`: Retorna `true` si está vacío [O(1)].

- `clear()`: Elimina todos los elementos [O(n)].
- `front()`: Retorna el primer elemento [O(1)].
- `back()`: Retorna el último elemento [O(1)].
- `insert(iterator, value, r)`: Inserta un elemento en la posición indicada [O(r)].
- `erase(iterator)`: Elimina el elemento en la posición indicada [O(1)].

2.4 std::stack

Contenedor de tipo LIFO (Last In, First Out).

2.4.1 Formas de definirlo:

- Syntaxis genérica: `std::stack<type> name;`

2.4.2 Funciones miembro:

- `push(value)`: Agrega un elemento al tope [O(1)].
- `pop()`: Elimina el elemento del tope [O(1)].
- `top()`: Retorna el elemento del tope [O(1)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].

2.5 std::queue

Contenedor de tipo FIFO (First In, First Out).

2.5.1 Formas de definirlo:

- Syntaxis genérica: `std::queue<type> name;`

2.5.2 Funciones miembro:

- `push(value)`: Agrega un elemento al final [O(1)].
- `pop()`: Elimina el primer elemento [O(1)].
- `front()`: Retorna el primer elemento [O(1)].
- `back()`: Retorna el último elemento [O(1)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].

2.6 std::priority_queue

Cola de prioridad. Su funcionamiento es similar a una cola, pero los elementos son extraídos en orden de prioridad. Para definirla hay que especificar el tipo de dato y el contenedor que se usará internamente y el comparador (3.1).

2.6.1 Formas de definirlo:

- Syntaxis genérica: `std::priority_queue<type> name;`
- Syntaxis con comparador: `std::priority_queue<type, std::vector<type>, std::greater<type>> name;`

2.7 std::set

Contenedor que almacena elementos únicos ordenados.

2.7.1 Formas de definirlo:

- Syntaxis genérica: `std::set<type> name;`
- Con comparador: `std::set<type, comp> name;`

2.7.2 Funciones miembro:

- `insert(value)`: Inserta un elemento [O(log n)].
- `erase(value)`: Elimina el elemento con valor 'value' [O(log n)].
- `erase(iterator)`: Elimina el elemento en la posición indicada [O(1)].
- `erase(first, last)`: Elimina los elementos en el rango [O(n)].
- `find(value)`: Retorna un iterador al elemento con valor 'value' [O(log n)].
- `count(value)`: Retorna la cantidad de elementos con valor 'value' [O(log n)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].

2.8 std::unordered_set

Contenedor que almacena elementos únicos sin ordenar.

2.8.1 Formas de definirlo:

- Syntaxis genérica: `std::unordered_set<type> name;`
- Con comparador: `std::unordered_set<type, comp> name;`

2.8.2 Funciones miembro:

- `insert(value)`: Inserta un elemento [O(log n)].
- `erase(value)`: Elimina el elemento con valor 'value' [O(log n)].
- `erase(iterator)`: Elimina el elemento en la posición indicada [O(1)].
- `erase(first, last)`: Elimina los elementos en el rango [O(n)].
- `find(value)`: Retorna un iterador al elemento con valor 'value' [O(log n)].
- `count(value)`: Retorna la cantidad de elementos con valor 'value' [O(log n)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].

2.9 std::map

Contenedor que almacena pares clave-valor ordenados por la clave. La función del comparador es comparar las claves de los elementos, esto sirve para mantener el orden de los elementos.

2.9.1 Formas de definirlo:

- Syntaxis genérica: `std::map<key, value> name;`
- Con comparador: `std::map<key, value, comp> name;`

2.9.2 Funciones miembro:

- `operator[key]`: Retorna el valor asociado a la clave 'key' [O(log n)].
- `insert(pair)`: Inserta un par clave-valor [O(log n)].
- `erase(key)`: Elimina el elemento con clave 'key' [O(log n)].
- `erase(iterator)`: Elimina el elemento en la posición indicada [O(1)].
- `erase(first, last)`: Elimina los elementos en el rango [O(n)].
- `find(key)`: Retorna un iterador al elemento con clave 'key' [O(log n)].
- `count(key)`: Retorna la cantidad de elementos con clave 'key' [O(log n)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].

2.10 std::unordered_map

Contenedor que almacena pares clave-valor sin ordenar.

2.10.1 Formas de definirlo:

- Syntaxis genérica: `std::unordered_map<key, value> name;`
- Con comparador: `std::unordered_map<key, value, comp> name;`

2.10.2 Funciones miembro:

- `operator[key]`: Retorna el valor asociado a la clave 'key' [O(log n)].
- `insert(pair)`: Inserta un par clave-valor [O(log n)].
- `erase(key)`: Elimina el elemento con clave 'key' [O(log n)].
- `erase(iterator)`: Elimina el elemento en la posición indicada [O(1)].
- `erase(first, last)`: Elimina los elementos en el rango [O(n)].
- `find(key)`: Retorna un iterador al elemento con clave 'key' [O(log n)].
- `count(key)`: Retorna la cantidad de elementos con clave 'key' [O(log n)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].

2.11 std::deque

Contenedor de secuencia que permite inserciones y eliminaciones en cualquier posición en tiempo constante.

2.11.1 Formas de definirlo:

- Syntaxis genérica: `std::deque<type> name;`

2.11.2 Funciones miembro:

- `push_back(value)`: Agrega un elemento al final [O(1)].
- `push_front(value)`: Agrega un elemento al principio [O(1)].
- `pop_back()`: Elimina el último elemento [O(1)].
- `pop_front()`: Elimina el primer elemento [O(1)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].
- `clear()`: Elimina todos los elementos [O(n)].
- `front()`: Retorna el primer elemento [O(1)].
- `back()`: Retorna el último elemento [O(1)].
- `insert(iterator, value)`: Inserta un elemento en la posición indicada [O(n)].

2.12 Binary Search Tree

Estructura de datos que permite almacenar elementos de forma ordenada. Cada nodo tiene a lo más dos hijos, el hijo izquierdo es menor que el nodo y el hijo derecho es mayor.

2.12.1 Implementacion:

```
struct Node {
  int value;
  Node* left;
  Node* right;
  Node(int value): value(value), left(nullptr), right(nullptr) {}
};

class BST {
  Node* root;
public:
  BST(): root(nullptr) {}

  void insert(int value) {
    root = insert(root, value);
  }
  Node* insert(Node* node, int value) {
    if (node == nullptr) {
      return new Node(value);
    }
    if (value < node->value) {
      node->left = insert(node->left, value);
    } else {
      node->right = insert(node->right, value);
    }
    return node;
  }

  void erase(int value) {
    root = erase(root, value);
  }
  Node* erase(Node* node, int value) {
    if (node == nullptr) {
      return nullptr;
    }
    if (value < node->value) {
      node->left = erase(node->left, value);
    } else if (value > node->value) {
      node->right = erase(node->right, value);
    } else {
      if (node->left == nullptr) {
        Node* right = node->right;
```

```
        delete node;
        return right;
      }
      if (node->right == nullptr) {
        Node* left = node->left;
        delete node;
        return left;
      }
      Node* min = find_min(node->right);
      node->value = min->value;
      node->right = erase(node->right, min->value);
    }
    return node;
  }

  Node* find_min(Node* node) {
    while (node->left != nullptr) {
      node = node->left;
    }
    return node;
  }
  Node* find_max(Node* node) {
    while (node->right != nullptr) {
      node = node->right;
    }
    return node;
  }

  bool find(int value) {
    return find(root, value);
  }
  bool find(Node* node, int value) {
    if (node == nullptr) {
      return false;
    }
    if (value < node->value) {
      return find(node->left, value);
    }
    if (value > node->value) {
      return find(node->right, value);
    }
    return true;
  }
};

\subsection{Adjacency Matrix}
\label{subsec:adjacency_matrix}
% Explicacion
Matriz donde cada celda representa una arista entre dos nodos.

\subsubsection{Implementacion:}
% Code
\begin{lstlisting}[style=cpp]
const int V;
bool adj[V][V];
int weight[V][V];
```

2.13 Adjacency list

Lista de adyacencia donde cada nodo tiene una lista de nodos adyacentes.

2.13.1 Implementacion:

```
const int V;
std::vector<int> adj[V];
std::vector<int> weight[V];
```

2.14 Union Find

Estructura de datos que permite realizar operaciones de unión y búsqueda en tiempo constante.

2.14.1 Implementacion:

```
struct edge{
    ll from,to,weight;
};

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;
  }
};
```

2.14.2 Funciones miembro:

- `findSet(x)`: Retorna el representante del conjunto al que pertenece 'x' [O(1)].
- `sameSet(x, y)`: Retorna `true` si 'x' y 'y' pertenecen al mismo conjunto [O(1)].
- `size(x)`: Retorna el tamaño del conjunto al que pertenece 'x' [O(1)].
- `unionSet(x, y)`: Une los conjuntos a los que pertenecen 'x' y 'y' [O(1)].

2.15 Heap

Estructura de datos que permite mantener un conjunto de elementos y obtener el máximo o mínimo en tiempo constante.

2.15.1 Formas de definirlo:

- Syntaxis genérica: `std::priority_queue<type> name;`
- Para un heap mínimo: `std::priority_queue<type, std::vector<type>, std::greater<type>> name;`

2.15.2 Funciones miembro:

- `push(value)`: Agrega un elemento al heap [O(log n)].
- `pop()`: Elimina el elemento del tope [O(log n)].
- `top()`: Retorna el elemento del tope [O(1)].
- `size()`: Retorna la cantidad de elementos [O(1)].
- `empty()`: Retorna `true` si está vacío [O(1)].

3 Snippets

3.1 Comparador

Un comparador es una función que se utiliza para ordenar los elementos de un contenedor de una forma específica. En C++, los comparadores son funciones que retornan un valor booleano siendo `true` si el primer argumento debe ir antes que el segundo y `false` en caso contrario.

3.1.1 Ejemplo:

```
bool cmp(int a, int b) {
  return a > b;
}
```

3.2 std::sort

La función `std::sort` ordena los elementos de un contenedor en orden ascendente. Si se desea ordenar en orden descendente, se puede utilizar un comparador (ver 3.1).

3.2.1 Implementacion:

```
std::sort(v.begin(), v.end(), comp);
```

3.3 std::stable_sort

La función `std::stable_sort` ordena los elementos de un contenedor en orden ascendente. A diferencia de `std::sort`, esta función mantiene el orden relativo de los elementos que son iguales.

3.3.1 Implementacion:

```
std::stable_sort(v.begin(), v.end(), comp);
```

3.4 std::binary_search

La función `std::binary_search` busca un elemento en un contenedor ordenado. Retorna `true` si el elemento se encuentra en el contenedor y `false` en caso contrario.

3.4.1 Implementacion:

```
std::binary_search(v.begin(), v.end(), x);
```

3.5 std::lower_bound

La función `std::lower_bound` retorna un iterador al primer elemento en un contenedor cuyo comparador retorne `false` para el valor dado.

3.5.1 Implementacion:

```
std::lower_bound(v.begin(), v.end(), x, comp);
```

3.6 std::upper_bound

La función `std::upper_bound` retorna un iterador al primer elemento en un contenedor cuyo comparador retorne `true` para el valor dado.

3.6.1 Implementacion:

```
std::upper_bound(v.begin(), v.end(), x, comp);
```

3.7 std::reverse

La función `std::reverse` invierte el orden de los elementos de un contenedor en $O(n)$.

3.7.1 Implementacion:

```
std::reverse(v.begin(), v.end());
```

3.8 std::next_permutation

La función `std::next_permutation` reordena los elementos de un contenedor en la siguiente permutación lexicográfica. Retorna `true` si la siguiente permutación existe y `false` en caso contrario.

3.8.1 Implementacion:

```
std::next_permutation(v.begin(), v.end());
```

Para obtener todas las permutaciones de un contenedor, se puede utilizar un ciclo `do-while` asegurandose que este este ordenado en orden ascendente antes de empezar.

```
std::sort(v.begin(), v.end());

do {
  // Procesar la permutacion
} while(std::next_permutation(v.begin(), v.end()));
```

3.9 std::prev_permutation

La función `std::prev_permutation` reordena los elementos de un contenedor en la permutación lexicográfica anterior. Retorna `true` si la permutación anterior existe y `false` en caso contrario.

3.9.1 Implementacion:

```
std::prev_permutation(v.begin(), v.end());
```

Para obtener todas las permutaciones de un contenedor, se puede utilizar un ciclo `do-while` asegurandose que este este ordenado en orden descendente antes de empezar.

3.10 std::accumulate

La función `std::accumulate` calcula la suma de los elementos de un contenedor. Complejidad $O(n)$.

3.10.1 Implementacion:

```
int sum = std::accumulate(v.begin(), v.end(), 0);
```

3.11 std::partial_sum

La función `std::partial_sum` calcula la suma acumulada de los elementos de un contenedor. El resultado se almacena en otro contenedor. Complejidad $O(n)$.

3.11.1 Implementacion:

```
vector<int> acm = {1, 2, 3, 4, 5};
std::partial_sum(acm.begin(), acm.end(), acm.begin());
// acm = {1, 3, 6, 10, 15}
```

3.12 std::inner_product

La función `std::inner_product` calcula el producto punto de dos contenedores. Complejidad $O(n)$.

3.12.1 Implementacion:

```
// Recibe unicamente un end iterator porque ambos contenedores deben tener la misma long
int dot = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0);
```

3.13 std::adjacent_difference

La función `std::adjacent_difference` calcula la diferencia entre elementos consecutivos de un contenedor. El resultado se almacena en otro contenedor. Complejidad $O(n)$.

3.13.1 Implementacion:

```
vector<int> diff = {1, 2, 4, 7, 11};
// Recibe tres iteradores para el comienzo contenedor, el final del contenedor y el valo
std::adjacent_difference(diff.begin(), diff.end(), diff.begin());
// diff = {1, 1, 2, 3, 4}
```

El valor inicial es el valor que se le asigna al primer elemento del contenedor resultado.

3.14 std::iota

La función `std::iota` asigna valores consecutivos a los elementos de un contenedor. Complejidad $O(n)$.

3.14.1 Implementacion:

```
// v = {0, 0, 0, 0, 0}
std::iota(v.begin(), v.end(), 1);
// v = {1, 2, 3, 4, 5}
```

3.15 Impresion de estructuras

3.15.1 Implementacion:

```
void printVector(const vector<int>& v) {
    for (int x : v) cout << x << " ";
    cout << endl;
}
```

4 Aproximaciones genéricas

4.1 Two Pointers

Two pointers es una técnica que se basa en mantener dos punteros en un arreglo, usualmente uno al inicio y otro al final, y moverlos de acuerdo a ciertas condiciones. Es útil para resolver problemas en los que se necesita recorrer un arreglo de manera eficiente, como por ejemplo encontrar un subarreglo con una suma específica.

4.1.1 Patrones en los enunciados que indican que se puede aplicar

1. Entradas: Los problemas generalmente involucran arreglos o cadenas como estructuras de datos principales, ya sea de tamaño fijo o dinámico. A menudo, los elementos del arreglo o cadena tienen restricciones claras (por ejemplo, enteros no negativos, caracteres alfabéticos, etc.).
2. Salidas: Se busca un resultado basado en relaciones entre elementos consecutivos o intervalos en las entradas, como:
   - Encontrar un subarreglo o una subcadena con propiedades específicas (como una suma o un patrón).
   - Contar el número de subarreglos o subcadenas que cumplen una condición dada.
   - Optimizar una métrica dentro de un rango o subsección, como maximizar una suma o minimizar una distancia.
3. Condiciones: El problema involucra relaciones que pueden ser evaluadas mediante comparación de índices, como:
   - Moverse simultáneamente hacia el centro o hacia un punto de convergencia.
   - Expandir o contraer un rango (ventana) dinámicamente según una condición específica.
4. Pistas en el enunciado:
   - Se mencionan intervalos, ventanas deslizantes, o subsecciones.
   - La solución requiere un enfoque eficiente para recorrer todas las combinaciones posibles (usualmente $O(n^2)$ o más lento en un enfoque directo), lo que sugiere la necesidad de optimización.

4.1.2 Problemas específicos en los que se puede aplicar

1. Dado un arreglo de números enteros, hayar:
   - La cantidad de subarreglos con una suma específica.
   - El subarreglo con la suma más grande.
   - Bungee builder.
   - Encontrar la subcadena más larga que cumpla ciertas condiciones (ej., con un número limitado de caracteres únicos).

4.2 Dynamic Programming

La programación dinámica es una técnica que se basa en dividir un problema en subproblemas más pequeños, resolverlos y guardar los resultados para no tener que recalcularlos. Es útil para resolver problemas en los que se necesita recorrer un espacio de estados, como por ejemplo encontrar la cantidad de formas de llegar a un punto específico en un tablero de ajedrez.

4.2.1 Patrones en los enunciados que indican que se puede aplicar

1. Entradas: Los problemas generalmente incluyen estructuras de datos como arreglos, matrices, o cadenas. Las entradas suelen tener límites moderados que permiten construir soluciones parciales iterativamente (por ejemplo, tamaños de hasta unos pocos miles).
2. Salidas: Se solicita optimizar una métrica (como maximizar o minimizar un valor) o contar el número de formas de alcanzar un objetivo. Ejemplos comunes incluyen:

- Determinar el valor óptimo de una solución (máximo, mínimo, o suma específica).
- Contar combinaciones, particiones o subsecuencias que cumplen una condición.
- Resolver problemas de decisión, como verificar si es posible alcanzar un objetivo dado ciertas restricciones.

3. Subproblemas recurrentes: El problema puede dividirse en subproblemas más pequeños cuyos resultados se reutilizan para construir la solución final. Algunos patrones incluyen:
   - Dependencia entre estados definidos por índices o por el progreso de las decisiones (por ejemplo, "¿cuál es el mejor resultado hasta el índice i?").
   - Soluciones parciales que pueden combinarse para resolver el problema completo.
4. Pistas en el enunciado:
   (a) Se hace referencia a optimización (máximos, mínimos, caminos más cortos, etc.).
   (b) Se menciona explícitamente la combinación de resultados parciales o subestructuras óptimas.
   (c) Las restricciones o condiciones sugieren que algunas soluciones deben recalcularse repetidamente de forma ineficiente en un enfoque ingenuo.
   (d) Los límites del problema (n, m, etc.) hacen que un enfoque de fuerza bruta no sea viable, pero se pueden procesar con una complejidad polinómica o pseudo-polinómica (como $O(n^2)$ o $O(n \cdot m)$).

4.2.2 Formas de obtener la función de recurrencia

1. Definir claramente los subproblemas: Identificar cómo dividir el problema principal en partes más pequeñas y manejables. Cada subproblema debe depender únicamente de una o más de sus versiones anteriores.
   - Pregúntate: "¿Qué representa el estado $dp[i]$?" (por ejemplo, el mejor resultado hasta el índice i, la cantidad de maneras de llegar al estado i, etc.).
2. Establecer relaciones entre subproblemas: Analiza cómo combinar las soluciones de subproblemas para construir la solución del problema mayor. Esto usualmente se traduce en una relación matemática o lógica basada en las decisiones posibles en cada estado.
3. Examinar las opciones en cada paso: Identifica las decisiones que se pueden tomar en cada etapa del problema. Por ejemplo, ¿puedes incluir o excluir un elemento? ¿Moverte a un estado vecino? Cada decisión debe ser considerada para construir la recurrencia.
4. Asegurar la optimalidad o completitud: Confirma que cada subproblema se resuelve de manera óptima o completa antes de pasar al siguiente. Esto asegura que la solución global se construya correctamente.
5. Usar ejemplos concretos: Trabaja con casos pequeños para identificar patrones o relaciones que generalicen la recurrencia. Esto puede ayudarte a encontrar errores o confirmar que los subproblemas están correctamente definidos.

4.2.3 Problemas específicos que se resuelven con programación dinámica

1. Knapsack Problem.
2. Longest Common Subsequence.
3. Interleaving Strings.
4. Minimum Path Sum.
5. Palindrome Partitioning.
6. Coin Change.
7. Triangle Path Sum.

8. Rod Cutting.
9. Longest Increasing Subsequence.
10. Problema del comerciante viajero con restricciones.

4.3 Divide and Conquer

El enfoque de dividir y conquistar se basa en descomponer un problema grande en varios subproblemas más pequeños, resolver estos subproblemas de forma recursiva y luego combinar sus soluciones para obtener el resultado final. Este método es particularmente efectivo para problemas cuya estructura permite una partición natural en partes independientes o semi-independientes. Es utilizado en algoritmos clásicos como el ordenamiento por fusión (merge sort) y la búsqueda binaria.

4.3.1 Patrones en los enunciados que indican que se puede aplicar

1. **Entradas**: Los problemas generalmente involucran estructuras de datos que se pueden dividir fácilmente, como listas, matrices o intervalos. Suelen tener límites suficientemente grandes como para que un enfoque iterativo pueda ser ineficiente.
2. **Salidas**: Se solicita resolver el problema completo combinando soluciones de subproblemas. Ejemplos comunes incluyen:
   - Ordenar datos (como en merge sort o quick sort).
   - Encontrar elementos específicos con restricciones (como búsqueda binaria).
   - Procesar intervalos o divisiones de datos, como la multiplicación de matrices o problemas geométricos.
3. **Subproblemas independientes**: El problema puede dividirse en partes más pequeñas que son relativamente independientes. Esto significa que las soluciones de estos subproblemas no se superponen ni dependen entre sí, excepto en la etapa final de combinación. Algunos patrones comunes incluyen:
   - División de un arreglo o lista en mitades.
   - Descomposición de problemas geométricos en cuadrantes o regiones.
   - Resolución de instancias más pequeñas de forma recursiva.
4. **Pistas en el enunciado**:
   (a) Se menciona explícitamente la posibilidad de dividir el problema en partes más pequeñas.
   (b) Hay una estructura jerárquica o acumulativa que sugiere combinar soluciones parciales.
   (c) Los límites del problema son suficientemente grandes como para que un enfoque iterativo pueda ser lento, pero la recursión permite reducir significativamente el espacio de búsqueda.
   (d) El problema tiene una estructura repetitiva o recurrente que facilita la subdivisión.

4.3.2 Formas de construir la solución

1. **Dividir el problema**: Identificar cómo descomponer el problema en partes más pequeñas e independientes.
   - Pregúntate: "¿Cómo puedo particionar los datos o el espacio del problema?" (por ejemplo, dividir un arreglo en mitades, particionar un espacio geométrico, etc.).
2. **Resolver recursivamente los subproblemas**: Abordar cada subproblema de forma independiente mediante el mismo enfoque recursivo.
   - Confirma que cada subproblema es una versión más pequeña del problema original.
3. **Combinar las soluciones**: Diseñar una estrategia para integrar los resultados de los subproblemas y obtener la solución global.

- Ejemplo: Fusionar listas ordenadas en merge sort.
- Ejemplo: Comparar resultados de cuadrantes en un problema geométrico.

4. **Asegurar la corrección**: Verifica que los subproblemas sean suficientes para cubrir todo el problema inicial y que la combinación no pierda información.
5. **Optimizar la división y la combinación**: Analiza el costo de las etapas de división y combinación para asegurarte de que el enfoque es eficiente (por ejemplo, $O(\log n)$ divisiones y combinaciones en $O(n)$).

4.3.3 Problemas específicos que se resuelven con dividir y conquistar

1. Merge Sort.
2. Quick Sort.
3. Binary Search.
4. Multiplicación de matrices de Strassen.
5. Closest Pair of Points (problema geométrico).
6. Encontrar la subcadena palíndroma más larga.
7. Potenciación rápida (exponentiation by squaring).
8. Resolver el problema de las torres de Hanoi.
9. Transformadas rápidas (como FFT).
10. Problemas de máximos y mínimos en listas descompuestas.

4.4 Backtracking

El backtracking es una técnica que explora todas las posibles soluciones de un problema mediante la construcción incremental de candidatos y retrocediendo (backtrack) cuando una solución parcial no cumple con las restricciones del problema. Es útil para resolver problemas que requieren encontrar una solución exacta, como juegos, combinaciones o problemas de búsqueda en árboles o grafos.

4.4.1 Patrones en los enunciados que indican que se puede aplicar

1. **Entradas:** Los problemas suelen incluir estructuras como arreglos, listas, conjuntos o grafos. Por lo general, el espacio de soluciones es grande, pero está bien definido y puede explorarse sistemáticamente.
2. **Salidas:** Se busca una solución exacta que cumpla con restricciones específicas. Algunos casos típicos incluyen:
   - Encontrar todas las combinaciones o permutaciones posibles que satisfacen una condición.
   - Resolver problemas de decisión como verificar si existe una solución bajo restricciones específicas.
   - Identificar una solución óptima en un espacio reducido de búsqueda (por ejemplo, rutas mínimas en problemas con pocas restricciones).
3. **Características de la solución:**
   - El problema puede representarse como una serie de decisiones en un árbol de búsqueda.
   - Es posible verificar si una solución parcial es válida en cualquier punto del proceso.
   - Se puede retroceder de una solución parcial inválida sin explorar todas las decisiones derivadas de ella.
4. **Pistas en el enunciado:**

(a) Se solicita enumerar o encontrar todas las soluciones posibles.

(b) Las restricciones o condiciones son explícitas y determinan si una solución es válida en cada paso (como problemas de sudokus o laberintos).

(c) La estructura del problema sugiere un proceso exploratorio donde las decisiones previas limitan las decisiones futuras.

(d) El espacio de búsqueda es grande, pero no lo suficiente como para que sea completamente inabordable mediante enfoques sistemáticos (usualmente hasta 10^6 posibles soluciones).

4.4.2 Formas de estructurar la solución con backtracking

1. **Definir las decisiones:** Identifica qué decisiones se deben tomar en cada paso para construir una solución. Por ejemplo: ¿qué elemento agregar a la solución parcial?, ¿qué movimiento realizar en un tablero?, etc.

2. **Estructurar el árbol de búsqueda:** Representa las posibles decisiones como un árbol, donde cada nodo corresponde a una decisión parcial.

3. **Verificar la validez de las soluciones parciales:** Implementa una función que determine si una solución parcial cumple con las restricciones del problema. Esto evita explorar ramas inválidas del árbol de búsqueda.

4. **Retroceder al encontrar un callejón sin salida:** Al llegar a una solución inválida o completa, vuelve al nodo anterior y explora otras ramas. Este retroceso se implementa mediante un enfoque recursivo o explícito con pilas.

5. **Usar poda (pruning):** Implementa optimizaciones para reducir el espacio de búsqueda al eliminar decisiones innecesarias o redundantes (por ejemplo, detenerse si ya se excedió un límite o si una solución parcial no puede ser óptima).

6. **Probar con ejemplos pequeños:** Utiliza casos pequeños para validar que el proceso de exploración y retroceso funciona correctamente. Esto es clave para evitar errores en la implementación recursiva.

4.4.3 Problemas específicos que se resuelven con backtracking

1. Sudoku Solver.
2. N-Queens Problem.
3. Subset Sum Problem.
4. Permutations and Combinations.
5. Crossword Puzzle Fitting.
6. Word Search in a Grid.
7. Hamiltonian Path or Cycle.
8. Graph Coloring Problem.
9. Knight's Tour Problem.
10. Maze Solving (con restricciones específicas).

4.5 Greedy

El enfoque greedy se basa en tomar decisiones locales óptimas en cada paso con la esperanza de que estas conduzcan a una solución global óptima. Es una técnica eficiente cuando se puede garantizar que estas decisiones locales óptimas construyen la mejor solución general. Ejemplos clásicos incluyen encontrar un camino de menor costo en un grafo, seleccionar actividades que maximicen el uso de recursos, o construir códigos óptimos de compresión.

4.5.1 Patrones en los enunciados que indican que se puede aplicar

1. Entradas: Los problemas suelen involucrar estructuras de datos como listas, grafos, o conjuntos ordenables. Las entradas tienden a tener relaciones ordenables o métricas claras (por ejemplo, peso, tiempo, costo, prioridad).

2. Salidas: Se solicita optimizar una métrica bajo restricciones, como maximizar el beneficio o minimizar el costo. Los problemas típicos incluyen:

   - Seleccionar subconjuntos de elementos (por ejemplo, actividades, rutas, tareas).
   - Minimizar tiempos, costos o recursos.
   - Maximizar beneficios, utilidades o valores acumulados.

3. Decisiones locales óptimas: Los problemas pueden descomponerse en una secuencia de decisiones en las que elegir la opción "más prometedora" en cada paso conduce a la solución. Algunos patrones comunes incluyen:

   - Ordenar elementos según alguna métrica (por ejemplo, menor costo, mayor beneficio, más cercano).
   - Seleccionar el elemento "mejor" disponible para añadir al resultado parcial (ejemplo: elegir una actividad con el tiempo de finalización más temprano).
   - Evitar elementos que no cumplen con las restricciones de manera evidente.

4. Pistas en el enunciado:

   (a) Se menciona una relación directa entre decisiones locales y resultados globales (por ejemplo, "elige siempre el elemento más pequeño/más grande").

   (b) Las restricciones permiten dividir el problema en partes independientes que pueden resolverse localmente.

   (c) La solución óptima puede derivarse directamente de una estructura ordenada (lista, cola de prioridad, grafo ponderado).

   (d) Los límites del problema hacen que enfoques más complejos (como programación dinámica) sean innecesarios.

4.5.2 Formas de diseñar un algoritmo Greedy

1. **Definir una estrategia greedy:** Identificar cuál es la decisión local óptima que puede tomarse en cada paso. Pregúntate:

   - "¿Qué elemento puedo elegir ahora para acercarme más a la solución final?"
   - "¿Cómo ordeno los elementos para facilitar la selección?"

2. **Justificar la optimalidad:** Asegúrate de que la estrategia greedy lleve a la solución global óptima. Esto puede implicar probar que el problema cumple:

   - **Propiedad greedy:** Una decisión local óptima siempre forma parte de una solución global óptima.
   - **Subestructura óptima:** Un problema puede resolverse combinando soluciones óptimas de subproblemas independientes.

3. **Ordenar o priorizar elementos:** A menudo, es necesario ordenar los datos de entrada de acuerdo con alguna métrica o criterio (por ejemplo, costo ascendente, beneficio descendente). Esto facilita la selección greedy.

4. **Tomar decisiones iterativas:** Recorre los elementos ordenados y aplica la estrategia greedy. Cada decisión debe ser localmente óptima y respetar las restricciones del problema.

5. **Validar con ejemplos pequeños:** Probar la solución en casos simples ayuda a identificar si el enfoque es correcto o si hay contraejemplos que violen la optimalidad global.

4.5.3 Problemas específicos que se resuelven con un enfoque Greedy

1. Activity Selection Problem.
2. Fractional Knapsack Problem.
3. Huffman Coding.
4. Dijkstra's Shortest Path Algorithm.
5. Kruskal's Minimum Spanning Tree.
6. Prim's Minimum Spanning Tree.
7. Interval Scheduling Maximization.
8. Job Sequencing Problem.
9. Coin Change (para denominaciones estándar).
10. Problemas de cobertura mínima (Set Cover Problem, aproximado).

5 Ejercicios resueltos

5.1 Fuerza Bruta

```cpp
using namespace std;

int maxSubarraySum(vector<int>& nums) {
    int n = nums.size();
    int maxSum = INT_MIN;

    for (int i = 0; i < n; ++i) {            // O(n)
        int currentSum = 0;
        for (int j = i; j < n; ++j) {        // O(n)
            currentSum += nums[j];
            maxSum = max(maxSum, currentSum);
        }
    }
    return maxSum;
}

int main() {
    vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "Maximum Subarray Sum: " << maxSubarraySum(nums) << endl;
    return 0;
}
```

Complejidad Temporal:

- **Peor Caso:** $O(n^2)$ (dos bucles anidados).
- **Mejor caso:** $O(n)$ si encuentra rapidamente una solucion optima.

5.2 Knapsack

- **Problema:** Dado un conjunto de objetos con pesos y valores, determinar el valor maximo que se puede obtener seleccionando objetos que no excedan una capacidad W.
- **Entrada:** Un entero W $(1 \leq W \leq 10^3)$ que representa la capacidad de la mochila, seguido de n enteros que representan los pesos de los objetos, y n enteros que representan los valores de los objetos.
- **Salida:** Imprimir el valor maximo que se puede obtener seleccionando objetos que no excedan la capacidad W.
- **Paradigma:** Programacion Dinamica

```cpp
int knapsack(vector<int>& weights, vector<int>& values, int W) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {                    // O(n)
        for (int w = 1; w <= W; ++w) {                // O(W)
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
                    values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][W];
}

int main() {
    vector<int> weights = {1, 2, 3};
    vector<int> values = {6, 10, 12};
```

```
    int W = 5;
    cout << "Maximum Value: " << knapsack(weights, values, W) << endl;
    return 0;
}
```

Complejidad Temporal:

- **Peor Caso:** $O(n * W)$, donde n es el numero de objetos y W es la capacidad de la mochila.
- **Espacio:** $O(n * W)$.

5.3 Algoritmo Greedy

```
int activitySelection(vector<pair<int, int>>& activities) {
    sort(activities.begin(), activities.end(), [](pair<int, int> a, pair<int, int>
        b) {
        return a.second < b.second; // Ordenar por tiempo de finalizacion
    });

    int count = 1;
    int lastFinish = activities[0].second;

    for (int i = 1; i < activities.size(); ++i) {    // O(n)
        if (activities[i].first >= lastFinish) {
            ++count;
            lastFinish = activities[i].second;
        }
    }
    return count;
}

int main() {
    vector<pair<int, int>> activities = {{1, 3}, {2, 5}, {4, 6}, {6, 8}, {5, 7}};
    cout << "Maximum Activities: " << activitySelection(activities) << endl;
    return 0;
}
```

Complejidad Temporal:

- **Ordenar:** $O(n * logn)$, donde n es el numero de objetos y W es la capacidad de la mochila.
- **Seleccion:** $O(n)$.

5.4 Backtracking

```
bool isSafe(vector<string>& board, int row, int col, int n) {
    for (int i = 0; i < row; ++i) {
        if (board[i][col] == 'Q') return false;
        if (col - (row - i) >= 0 && board[i][col - (row - i)] == 'Q') return false;
        if (col + (row - i) < n && board[i][col + (row - i)] == 'Q') return false;
    }
    return true;
}

void solve(vector<string>& board, int row, int n, vector<vector<string>>&
    solutions) {
    if (row == n) {
        solutions.push_back(board);
        return;
    }
    for (int col = 0; col < n; ++col) {
        if (isSafe(board, row, col, n)) {
            board[row][col] = 'Q';
            solve(board, row + 1, n, solutions);
            board[row][col] = '.'; // Backtrack
        }
    }
}

vector<vector<string>> solveNQueens(int n) {
```

```
    vector<vector<string>> solutions;
    vector<string> board(n, string(n, '.'));
    solve(board, 0, n, solutions);
    return solutions;
}

int main() {
    int n = 8; // Cambiar segun el problema
    auto solutions = solveNQueens(n);
    cout << "Total Solutions: " << solutions.size() << endl;
    return 0;
}
```

Complejidad Temporal:

- **Peor Caso:** $O(N!)$, ya que explora todas las posibles configuraciones de las reinas.

5.5 Dividir y Conquistar

```
int maxCrossingSum(vector<int>& nums, int l, int m, int r) {
    int leftSum = INT_MIN, rightSum = INT_MIN;
    int sum = 0;

    for (int i = m; i >= l; --i) {
        sum += nums[i];
        leftSum = max(leftSum, sum);
    }

    sum = 0;
    for (int i = m + 1; i <= r; ++i) {
        sum += nums[i];
        rightSum = max(rightSum, sum);
    }

    return leftSum + rightSum;
}

int maxSubarraySum(vector<int>& nums, int l, int r) {
    if (l == r) return nums[l];
    int m = l + (r - l) / 2;
    return max({maxSubarraySum(nums, l, m),
                maxSubarraySum(nums, m + 1, r),
                maxCrossingSum(nums, l, m, r)});
}

int main() {
    vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "Maximum Subarray Sum: " << maxSubarraySum(nums, 0, nums.size() - 1) <<
        endl;
    return 0;
}
```

Complejidad Temporal:

- **Peor Caso:** $O(n * logn)$, debido a la recurrencia $T(n) = 2T(n/2) + O(n)$.

5.6 Monedas

- **Problema:** Dado un valor x y un conjunto de monedas, determinar la cantidad minima de monedas necesarias para formar el valor x.
- **Entrada:** Dos enteros n $(1 \leq n \leq 100)$ y x $(1 \leq x \leq 10^6)$, seguido de n enteros que representan las monedas.
- **Salida:** Imprimir la cantidad minima de monedas necesarias para formar el valor x.
- **Paradigma:** Programacion Dinamica

```cpp
int main() {
    int n, x;
    cin >> n >> x;

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

    vector<int> dp(x + 1, INF);
    dp[0] = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = coins[i]; j <= x; ++j) {
            if (dp[j - coins[i]] != INF) {
                dp[j] = min(dp[j], dp[j - coins[i]] + 1);
            }
        }
    }

    if (dp[x] == INF) {
        cout << -1 << endl;
    } else {
        cout << dp[x] << endl;
    }

    return 0;
}
```

5.7 Creando Strings

- **Problema:** Dado un string, imprimir todas las permutaciones posibles.
- **Entrada:** Una cadena de caracteres s ($1 \leq |s| \leq 8$).
- **Salida:** Imprimir todas las permutaciones posibles de la cadena s.
- **Paradigma:** Fuerza Bruta

```cpp
int main() {
    string input;
    cin >> input;

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

    vector<string> permutations;
    do {
        permutations.push_back(input);
    } while (next_permutation(input.begin(), input.end()));

    cout << permutations.size() << endl;
    for (string& s : permutations) {
        cout << s << endl;
    }

    return 0;
}
```

5.8 Subarrays

- **Problema:** Dado un arreglo de enteros, contar cuantos subarreglos suman un valor k.
- **Entrada:** Dos enteros n ($1 \leq n \leq 10^5$) y k ($1 \leq k \leq 10^9$), seguido de n enteros que representan el arreglo.
- **Salida:** Imprimir la cantidad de subarreglos que suman k.
- **Paradigma:** Dos Punteros

```
int main() {
    int largo;
    int suma;
    cin >> largo >> suma;

    vector<int> arr(largo);
    for (int i = 0; i < largo; ++i) {
        cin >> arr[i];
    }

    partial_sum(arr.begin(), arr.end(), arr.begin());

    int i = 0, j = 0;
    int count = 0;

    while (j < largo) {
        int currentSum = arr[j] - (i > 0 ? arr[i - 1] : 0);
        if (currentSum == suma) {
            ++count;
            ++j;
        } else if (currentSum < suma) {
            ++j;
        } else {
            ++i;
        }
    }

    cout << count << endl;
}
```

Container	Insert Head	Insert Tail	Insert	Remove Head	Remove Tail	Remove	Index Search	Find
vector	n/a	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(log n)
list	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	n/a	O(n)
deque	O(1)	O(1)	n/a	O(1)	O(1)	O(n)	n/a	n/a
queue	n/a	O(1)	n/a	O(1)	n/a	n/a	O(1)	O(log n)
stack	O(1)	n/a	n/a	O(1)	n/a	n/a	n/a	n/a
map	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multimap	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)
set	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multiset	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)

Figure 1: Complejidades Heinz

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA | DEPARTAMENTO DE INFORMÁTICA

Teorema maestro

El teorema maestro es una receta para resolver recurrencias algoritmicas, donde cumple la siguiente forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- a es la cantidad de llamadas recursivas
- b es factor de reducción del tamaño de la entrada
- d es el exponente en el tiempo de ejecución del "paso de combinación"

Notar que $T(n)$ es recursiva

17/30

Figure 2: Teorema Maestro

Teorema maestro

Definición formal

Theorem 4.1 (Master Method) If $T(n)$ is defined by a standard recurrence with parameteres $a \geq 1$, $b \geq 1$ and $d \geq 0$ then

$$T(n) = \begin{cases} O(n^d \log n) \text{ if } a = b^d \\ O(n^d) \text{ if } a < b^d \\ O(n^{\log_b a}) \text{ if } a > b^d \end{cases}$$

Figure 3: Teorema Maestro Parte 2