Fuerza bruta y busqueda exhaustiva

¿Qué es un algoritmo de fuerza bruta?

Un algoritmo de fuerza bruta busca soluciones a problemas generando todas las posibles configuraciones hasta encontrar la mejor.

Pros:

- Sencillo de implementar.
- Garantiza encontrar la solución óptima.

Contras:

- Puede ser ineficiente y consumir mucho tiempo en problemas grandes.

Permutaciones

Problema: Calcular todas las permutaciones de una lista.

- Input: [1,2,3]
- Output: [1,2,3], [1,3,2] [2,1,3] [2,3,1] [3,1,2] [3,2,1]

¿Para qué?

Muchos problemas se pueden resolver encontrando “la mejor permutación” de algún determinado conjunto. Conocer como generar o recorrer las permutaciones es útil para poder hacer algoritmos de fuerza bruta.

Tamaño: Si el largo del input es **n**, la cantidad de permutaciones posibles es **n!**

Permutaciones

La STL de c++ ya implementa la función next_permutation, que permite recorrer todas las permutaciones de un contendor.

- Recorre las permutaciones en orden lexicográfico (asume que la primera permutación es la lista ordenada)
- Cada llamada reordena el contenedor y retorna:
  - True, si existe una siguiente permutación
  - False, si ya llego a la última permutación

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

void imprimir_permutaciones(std::vector<int>& arr) {
    std::sort(arr.begin(), arr.end());
    do {
        for (int num : arr) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
    } while (std::next_permutation(arr.begin(), arr.end()));
}

int main() {
    std::vector<int> arr = {1, 2, 3};
    imprimir_permutaciones(arr);
    return 0;
}
```

Problema: Vendedor viajero

Dado un conjunto de ciudades y las distancias entre ellas, encontrar la ruta más corta que permita visitar cada ciudad una vez y regresar a la ciudad de origen.

Input:

```
std::vector<int> ciudades = {0, 1, 2, 3};
std::vector<std::vector<int>> distancias = {
        {0, 40, 15, 17},
        {40, 0, 10, 20},
        {15, 10, 0, 25},
        {17, 20, 25, 0}
    };
```

Output:

```
Mejor ruta: 0 2 1 3
Costo: 62
```

Ejercicio: Modificar el código para resolver el problema

```
#include <iostream>
#include <algorithm>
#include <vector>

void imprimir_permutaciones(std::vector<int>& arr) {
    std::sort(arr.begin(), arr.end());
    do {
        for (int num : arr) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
    } while (std::next_permutation(arr.begin(), arr.end()));
}

int main() {
    std::vector<int> arr = {1, 2, 3};
    imprimir_permutaciones(arr);
    return 0;
}
```

Solución: Vendedor viajero

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

int main()
{
    std::vector<int> ciudades = {0, 1, 2, 3};
    std::vector<std::vector<int>> distancias = {
        {0, 40, 15, 17},
        {40, 0, 10, 20},
        {15, 10, 0, 25},
        {17, 20, 25, 0}};
    resolver_TSP(ciudades, distancias);
    return 0;
}

int calcular_costo_ruta(const std::vector<int> &ruta,
                        const std::vector<std::vector<int>> &distancias)
{
    int costo = 0;
    for (size_t i = 0; i < ruta.size() - 1; ++i)
    {
        costo += distancias[ruta[i]][ruta[i + 1]];
    }
    // Regreso a la ciudad de origen
    costo += distancias[ruta.back()][ruta.front()];
    return costo;
}
```

```cpp
void resolver_TSP(std::vector<int> &ciudades,
                  const std::vector<std::vector<int>> &distancias)
{
    std::sort(ciudades.begin(), ciudades.end());
    int mejor_costo = INT_MAX;
    std::vector<int> mejor_ruta;

    do
    {
        int costo = calcular_costo_ruta(ciudades, distancias);
        if (costo < mejor_costo)
        {
            mejor_costo = costo;
            mejor_ruta = ciudades;
        }
    } while (std::next_permutation(ciudades.begin(), ciudades.end()));

    std::cout << "Mejor ruta: ";
    for (int ciudad : mejor_ruta)
    {
        std::cout << ciudad << " ";
    }
    std::cout << "\nCosto: " << mejor_costo << std::endl;
}
```

Backtracking

- Es un método de búsqueda que genera soluciones incrementales y descarta aquellas que no cumplen con las condiciones requeridas.
- Se utiliza para resolver problemas donde se necesita explorar múltiples posibilidades.
- En general, un algoritmo de backtracking toma decisiones para construir la solución, cuando llega a un camino sin salida, se devuelve (*backtrack*) y sigue tomando otras decisiones. Esto se puede ver como un recorrido en un árbol (el *árbol de busqueda*).

Backtracking

- Algoritmo de backtracking para generar las permutaciones de una lista.
- **Ejercicio**. Modificarlo para resolver el problema del vendedor viajero.

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

void permutar(std::vector<int>& arr, int inicio) {
    if (inicio == arr.size() - 1) {
        for (int num : arr) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
        return;
    }
    for (int i = inicio; i < arr.size(); ++i) {
        std::swap(arr[inicio], arr[i]);
        permutar(arr, inicio + 1);
        std::swap(arr[inicio], arr[i]);
    }
}

int main() {
    std::vector<int> arr = {1, 2, 3};
    permutar(arr, 0);
    return 0;
}
```

Solución: Vendedor viajero

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits.h>
int main()
{
    std::vector<int> ruta_inicial = {0, 1, 2, 3};
    std::vector<std::vector<int>> distancias = {
        {0, 40, 15, 17},
        {40, 0, 10, 20},
        {15, 10, 0, 25},
        {17, 20, 25, 0}};

    int mejor_costo = INT_MAX;
    std::vector<int> mejor_ruta;

    resolver_TSP(ruta_inicial, distancias, 0, 0, mejor_costo, mejor_ruta);

    std::cout << "Mejor ruta: ";
    for (int ciudad : mejor_ruta)
    {
        std::cout << ciudad << " ";
    }
    std::cout << "\nCosto: " << mejor_costo << std::endl;

    return 0;
}
```

```cpp
void resolver_TSP(std::vector<int> &ruta_actual,
                  const std::vector<std::vector<int>> &distancias,
                  int nivel,
                  int costo_actual,
                  int &mejor_costo,
                  std::vector<int> &mejor_ruta)
{
    if (nivel == ruta_actual.size())
    {
        costo_actual += distancias[ruta_actual[nivel - 1]][ruta_actual[0]];
        if (costo_actual < mejor_costo)
        {
            mejor_costo = costo_actual;
            mejor_ruta = ruta_actual;
        }
        return;
    }

    for (int i = nivel; i < ruta_actual.size(); ++i)
    {
        std::swap(ruta_actual[nivel], ruta_actual[i]);

        if (nivel > 0)
        {
            int costo_nuevo = costo_actual + distancias[ruta_actual[nivel - 1]][ruta_actual[nivel]];
            resolver_TSP(ruta_actual, distancias, nivel + 1, costo_nuevo, mejor_costo, mejor_ruta);
        }
        else
        {
            resolver_TSP(ruta_actual, distancias, nivel + 1, costo_actual, mejor_costo, mejor_ruta);
        }

        std::swap(ruta_actual[nivel], ruta_actual[i]);
    }
}
```

Poda

- Es una técnica utilizada para evitar la exploración de ramas que no pueden conducir a una solución óptima.
- Mejora la eficiencia del algoritmo de backtracking al reducir el número de caminos explorados.

Solución: Vendedor viajero (con poda)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits.h>

int main()
{
    std::vector<int> ruta_inicial = {0, 1, 2, 3};
    std::vector<std::vector<int>> distancias = {
        {0, 40, 15, 17},
        {40, 0, 10, 20},
        {15, 10, 0, 25},
        {17, 20, 25, 0}};

    int mejor_costo = INT_MAX;
    std::vector<int> mejor_ruta;

    resolver_TSP_poda(ruta_inicial, distancias, 0, 0, mejor_costo, mejor_ruta);

    std::cout << "Mejor ruta: ";
    for (int ciudad : mejor_ruta)
    {
        std::cout << ciudad << " ";
    }
    std::cout << "\nCosto: " << mejor_costo << std::endl;

    return 0;
}
```

```cpp
void resolver_TSP_poda(std::vector<int> &ruta_actual,
                       const std::vector<std::vector<int>> &distancias,
                       int nivel,
                       int costo_actual,
                       int &mejor_costo,
                       std::vector<int> &mejor_ruta)
{
    if (nivel == ruta_actual.size())
    {
        costo_actual += distancias[ruta_actual[nivel - 1]][ruta_actual[0]];
        if (costo_actual < mejor_costo)
        {
            mejor_costo = costo_actual;
            mejor_ruta = ruta_actual;
        }
        return;
    }

    for (int i = nivel; i < ruta_actual.size(); ++i)
    {
        std::swap(ruta_actual[nivel], ruta_actual[i]);

        if (nivel > 0)
        {
            int costo_nuevo = costo_actual + distancias[ruta_actual[nivel - 1]][ruta_actual[nivel]];
            if (costo_nuevo < mejor_costo)
            {
                resolver_TSP_poda(ruta_actual, distancias, nivel + 1, costo_nuevo, mejor_costo,
mejor_ruta);
            }
        }
        else
        {
            resolver_TSP_poda(ruta_actual, distancias, nivel + 1, costo_actual, mejor_costo,
mejor_ruta);
        }

        std::swap(ruta_actual[nivel], ruta_actual[i]);
    }
}
```

Poda

Conjunto potencia

Otro espacio de búsqueda común es el conjunto potencia

Problema: Dado una lista de elementos, calcular todos los subconjuntos que se pueden formar a partir de los elementos

- Input: [1,2,3]
- Output: {}, {1}, {2}, {3} , {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}

Tamaño: Si el largo del input es **n**, la cantidad de subconjuntos es **2^n**

Construcción: Para cada elemento del input se toma una decisión binaria (se agrega al subconjunto o no).

Dado que cada decisión es binaria, y queremos la combinación de todas las posibles decisiones. Los subconjuntos se pueden enumerar utilizando números binarios de n dígitos.

Recorrer los subconjuntos equivale a recorrer un contador, los subconjuntos corresponden a la representación binaria del contador

Ejercicio: Algoritmo que genere el conjunto potencia utilizando los bits de un contador.

Conjunto potencia

```
vector<vector<int>> encontrarConjuntoPotencia(const vector<int> &nums)
{
    int num_elementos = nums.size();
    unsigned int num_subconjuntos = pow(2, num_elementos); // Total de subconjuntos posibles (2^n)

    vector<vector<int>> subconjuntos;           // Almacena todos los subconjuntos

    // Generar todos los subconjuntos posibles
    for (int contador = 0; contador < num_subconjuntos; ++contador)
    {
        vector<int> subconjunto_actual;

        // Construir el subconjunto actual basado en los bits de 'contador'
        for (int j = 0; j < num_elementos; ++j)
        {
            if (contador & (1 << j))
            { // Verificar si el bit j está encendido
                subconjunto_actual.push_back(nums[j]);
            }
        }

        subconjuntos.push_back(subconjunto_actual);
    }

    return subconjuntos;
}
```

Conjunto potencia

En general, el conjunto potencia es un conjunto de conjuntos y por ende no permite que haya subconjuntos repetidos. Por ejemplo

- Input: [1,2,2]
- Output: {}, {1}, {2}, {2}, {1, 2}, {1, 2}, {2, 2}, {1, 2, 2}

Aquí la salida no es la esperada, lo que se quisiera es que la salida sea

- Output: {}, {1}, {2}, {1, 2}, {2, 2}, {1, 2, 2}

Ejercicio: Arreglar el código para que no devuelva repetidos

Conjunto potencia

Solución

```cpp
vector<vector<int>> encontrarConjuntoPotencia(const vector<int> &nums)
{
    int num_elementos = nums.size();
    unsigned int num_subconjuntos = pow(2, num_elementos); // Total de subconjuntos posibles (2^n)

    // Ordenar el array para gestionar duplicados
    vector<int> elementos_ordenados = nums;
    sort(elementos_ordenados.begin(), elementos_ordenados.end());

    vector<vector<int>> subconjuntos;           // Almacena todos los subconjuntos
    unordered_set<string> subconjuntos_unicos; // Almacena representaciones únicas de subconjuntos

    // Generar todos los subconjuntos posibles
    for (int contador = 0; contador < num_subconjuntos; ++contador)
    {
        vector<int> subconjunto_actual;
        string clave_unica; // Para evitar duplicados

        // Construir el subconjunto actual basado en los bits de 'contador'
        for (int j = 0; j < num_elementos; ++j)
        {
            if (contador & (1 << j))
            { // Verificar si el bit j está encendido
                subconjunto_actual.push_back(elementos_ordenados[j]);
                clave_unica += to_string(elementos_ordenados[j]) + '$';
            }
        }

        // Si el subconjunto es único, agregarlo a la lista de subconjuntos
        if (subconjuntos_unicos.find(clave_unica) == subconjuntos_unicos.end())
        {
            subconjuntos.push_back(subconjunto_actual);
            subconjuntos_unicos.insert(clave_unica);
        }
    }

    return subconjuntos;
```