



INFORME DE LABORATORIO

INFORMACIÓN BÁSICA

ASIGNATURA:	ANÁLISIS Y DISEÑO DE ALGORITMOS				
TÍTULO DE LA PRÁCTICA:	PROGRAMACIÓN DINÁMICA				
NÚMERO DE PRÁCTICA:	P2	AÑO LECTIVO:	2024	SEMESTRE:	PAR
ESTUDIANTES: 20200593, Chura Monroy Daniel Wilston					
DOCENTES: Marcela Quispe Cruz, Manuel Loaiza, Alexander J. Benavides					

RESULTADOS Y PRUEBAS

El informe se presenta con un formato de artículo.
Repositorio: <https://github.com/DanielMon/ADA/tree/master>
Revise la sección de *Resultados Experimentales*.

CONCLUSIONES

El informe se presenta con un formato de artículo.
Revise la sección de *Conclusiones*.

METODOLOGÍA DE TRABAJO

El informe se presenta con un formato de artículo.
Revise la sección de *Diseño Experimental*.

REFERENCIAS Y BIBLIOGRAFÍA

El informe se presenta con un formato de artículo.
Revise la sección de *Referencias Bibliográficas*.

Programación Dinámica – Ejemplos de Aplicación

Resumen

Este informe aborda la resolución de tres problemas clásicos de programación mediante el enfoque SRTBOT, que ayuda a definir, analizar y resolver los problemas de manera óptima. El enfoque SRTBOT incluye la Definición del Subproblema, la Recursividad, la Topología del problema, los Casos Base, el Problema Original y el Análisis Temporal, lo que permite una visión clara y precisa para cada desafío y nos brinda una estructura para ir desarrollando la solución para el problema.

Cada sección del informe detalla la aplicación de SRTBOT en la formulación y solución de los problemas, analizando la complejidad y la eficiencia de las estrategias elegidas.

1. Introducción

La resolución de problemas computacionales mediante algoritmos eficientes es un pilar fundamental en el ámbito de la informática y la optimización de recursos. Este informe analiza tres problemas de la plataforma UVA (Luggage, Collecting Beepers y Square), los cuales presentan desafíos en la manipulación de conjuntos y el cálculo de rutas óptimas, problemas comunes en la programación competitiva y la investigación operativa. El abordaje de estos problemas no solo proporciona herramientas de análisis matemático y computacional, sino que también ayuda a mejorar la toma de decisiones en situaciones de distribución de recursos y optimización de recorridos, habilidades aplicables en campos como la logística, la inteligencia artificial y la teoría de grafos.

El objetivo principal de este trabajo es aplicar el enfoque SRTBOT (Subproblema, Recursividad, Topología, Casos Base, Problema Original y Análisis Temporal) para descomponer y resolver los problemas de manera estructurada, facilitando la comprensión y optimización de las soluciones. Este enfoque permite dividir los problemas en partes manejables, definir relaciones recursivas y analizar la eficiencia temporal de los algoritmos, lo cual es crucial para resolver problemas complejos de forma efectiva.

El primer problema, Luggage, se enfoca en la distribución equitativa de pesos en dos maleteros, utilizando una técnica de partición de subconjuntos para evaluar si es posible dividir las maletas en dos grupos de igual peso. El segundo problema, Collecting Beepers, implica la búsqueda de la ruta más corta para que un robot visite varias ubicaciones y retorne a su punto de inicio, lo que se resuelve con algoritmos de optimización de caminos mínimos. Finalmente, el problema Square explora la posibilidad de formar un cuadrado a partir de palos de diferentes longitudes, utilizando combinaciones de subconjuntos para verificar si se pueden construir cuatro lados de igual longitud.

La [Sección 2](#) describe los conceptos y técnicas teóricas fundamentales, como la programación dinámica y el nemotécnico SRTBOT, para entender el problema y sus soluciones. La [Sección 3](#) presenta el Diseño Experimental, detallando el proceso seguido para seleccionar y resolver los problemas, y cómo se llevaron a cabo las actividades para lograr los objetivos del trabajo. La [Sección 4](#) muestra los Resultados, incluyendo ejemplos de problemas resueltos con análisis detallado de subproblemas, relaciones recursivas, topología y casos básicos, así como los algoritmos recursivos, con y sin memoización, y el código en C++ resultante. La [Sección 5](#) expresa los logros, dificultades y la eficacia de la metodología aplicada en este estudio.

2. Marco Teórico Conceptual

Luggage (10664), Collecting Beepers (10496) y Square (10364) son problemas que requieren el uso de técnicas de programación como la programación dinámica, la búsqueda de caminos óptimos en grafos y algoritmos que usan combinatoria.

2.1. Programación Dinámica (PD)

Técnica de optimización que descompone un problema en subproblemas más pequeños, resolviéndolos una sola vez y almacenando los resultados para reutilizarlos. Esto ayuda a evitar cálculos repetidos y mejora la eficiencia, especialmente en problemas donde subproblemas se repiten.

2.2. Memoización

Es una técnica de PD en la que se almacenan los resultados de subproblemas ya resueltos en una estructura de datos (como un arreglo o matriz). Esto permite consultar soluciones previas en lugar de volver a calcularlas, acelerando el tiempo de ejecución en problemas recursivos y evitar cálculos innecesarios.

2.3. Recursión

Método en el que una función se llama a sí misma para dividir el problema en partes más pequeñas hasta alcanzar un caso base. Es común en la resolución de problemas de PD y backtracking.

2.4. Problema de Suma de Subconjuntos

Problema clásico de PD en el que se busca determinar si existe un subconjunto de elementos que suma exactamente a un valor objetivo. Es útil en situaciones de partición y distribución, como para Luggage.

2.5. Teoría de Grafos

Área de la matemática que estudia estructuras en forma de red (grafos), compuestas por nodos y aristas. Los problemas de recorridos y optimización de caminos (como en Collecting Beepers) se modelan frecuentemente mediante grafos.

2.6. Problema del Vendedor Viajero (TSP - Traveling Salesman Problem)

Problema de optimización en grafos donde se busca encontrar el camino más corto que visita un conjunto de puntos y regresa al punto de inicio. En Collecting Beepers, este problema es análogo a encontrar la ruta óptima para visitar todos los beepers.

2.7. Distancia de Manhattan

Métrica utilizada para medir la distancia entre dos puntos en una cuadrícula, sumando las diferencias absolutas de sus coordenadas. Es adecuada para movimientos en líneas rectas a lo largo de los ejes x e y, como se emplea en Collecting Beepers.

2.8. Backtracking

Técnica de prueba y error que construye una solución progresivamente, retrocediendo (backtracking) cuando una opción no cumple las condiciones necesarias. En Square, se utiliza para explorar posibles combinaciones de palos para formar un cuadrado.

2.9. Problema de Partición de Conjuntos

Problema en el que se busca dividir un conjunto en subconjuntos con alguna condición, como tener la misma suma. En Square, esto se aplica para verificar si los palos pueden formar lados iguales para un cuadrado.

3. Diseño Experimental

3.1. Objetivos

Los objetivos de este trabajo son:

- Reforzar los conocimientos del método de programación dinámica.
- Aplicar el método de programación dinámica para resolver algunos problemas propuestos.

3.2. Actividades

El estudiante realizó las siguientes acciones.

1. Crear un usuario en <http://vjudge.net> e indicar en este paso el nombre de usuario utilizado: Daniel-Chura
2. Seleccionar aleatoriamente tres problemas de la lista disponible en <http://bit.ly/3UxdCVL>.
 - a) -UVA-10664 Luggage
 - b) -UVA-10496 Collecting Beepers
 - c) - UVA-10364 Square
3. Deberá diseñar una solución utilizando la técnica SRTBOT para cada problema.
4. Muestra el pseudocódigo recursivo resultante sin y con memoización.
5. Deberá incluir el código en C++ que resulta del modelo SRTBOT.
6. Deberá anexar el PDF del código aceptado por la plataforma <http://vjudge.net> al final del artículo.

4. Resultados

4.1. Problema 10664 – Luggage

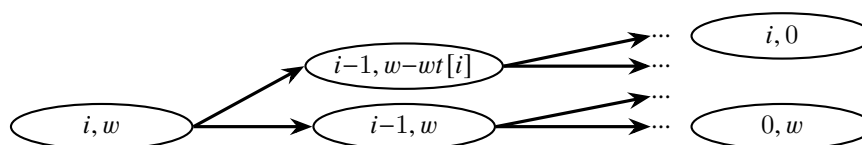
Subproblema: Determinar si es posible dividir los objetos $0 \leq i \leq n$ en dos subconjuntos con una suma total igual a la mitad del peso total W .

Para ello, definimos el subproblema: ¿Es posible alcanzar la suma $0 \leq w \leq W/2$ utilizando los primeros i objetos?

Relaciones Recursivas:

$$C(i, w) = \begin{cases} \text{true} & \text{si } w = 0 \\ \text{false} & \text{si } i < 0 \text{ o } w < 0 \\ C(i-1, w) \vee C(i-1, w - wt[i]) & \text{en otro caso.} \end{cases}$$

Topología:



Básico:

- $C(i, 0) = \text{true}$: Siempre se puede lograr una suma objetivo de 0.
- $C(-1, w) = \text{false}$: Si no hay objetos restantes, no se puede alcanzar la suma $w > 0$.

Original:**Algorithm** $\text{divide}(i, w)$ // sin memoización**Input:** objeto i , suma objetivo w **Output:** true or false

```

1: if  $i < 0$  o  $w = 0$  then
2:   return  $w = 0$ 
3: else
4:   return  $\text{divide}(i-1, w)$  or
    $\text{divide}(i-1, w-wt[i])$ 

```

Algorithm $\text{divideM}(i, w)$ // con memoización**Input:** objeto i , suma objetivo w **Output:** true or false

```

1: Crear clave  $key \leftarrow (i, w)$ 
2: if clave está en  $memo$  then
3:   return  $memo[key]$ 
4: if  $i < 0$  o  $w = 0$  then
5:    $memo[key] \leftarrow (w = 0)$ 
6: else
7:    $memo[key] \leftarrow \text{divideM}(i-1, w)$  or
    $\text{divideM}(i-1, w-wt[i])$ 
8: return  $memo[key]$ 

```

Tiempo: $\text{divideM}(i, w) \in O(i \times w)$ **Código:**

```

1 #include <iostream>
2 #include <vector>
3 #include <string>
4 #include <sstream>
5 #include <unordered_map>
6 using namespace std;
7
8 unordered_map<string, bool> memo;
9
10 string makeKey(int index, int target) {
11     return to_string(index) + "," + to_string(target);
12 }
13
14 bool canPartitionMemo(const vector<int>& nums,
15                       int index, int target) {
16     if (target == 0) return true;
17     if (index < 0 || target < 0) return false;
18
19     string key = makeKey(index, target);
20     if (memo.find(key) != memo.end()) {
21         return memo[key];
22     }
23
24     bool include = canPartitionMemo(nums, index - 1,
25                                     target - nums[index]);
26     bool exclude = canPartitionMemo(nums, index - 1,
27                                     target);
28
29     memo[key] = include || exclude;
30     return memo[key];
31 }

```

```

25 int main() {
26     int testCases;
27     cin >> testCases;
28     cin.ignore();
29
30     while (testCases-- > 0) {
31         string line;
32         getline(cin, line);
33         stringstream ss(line);
34         vector<int> nums;
35         int num, sum = 0;
36
37         while (ss >> num) {
38             nums.push_back(num);
39             sum += num;
40         }
41
42         if (sum % 2 != 0) {
43             cout << "NO" << endl;
44             continue;
45         }
46
47         int target = sum / 2;
48         memo.clear();
49         if (canPartitionMemo(nums, nums.size() - 1, target)) {
50             cout << "YES" << endl;
51         } else {
52             cout << "NO" << endl;
53         }
54     }
55     return 0;
56 }

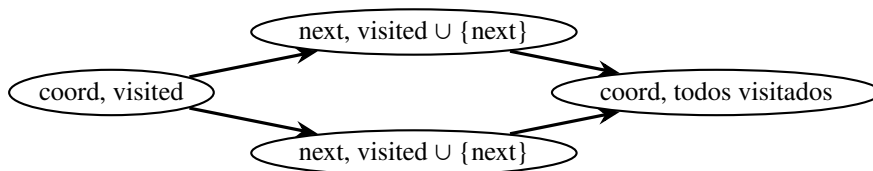
```

4.2. Problema 10496 – Collecting Beepers

Subproblema: Calcule $SP(i, v)$, donde $SP(i, v)$ representa la menor distancia posible desde el punto i al recoger todos los puntos definidos por v .

Relaciones Recursivas:

$$\text{getShortestPerimeter}(\text{coord}, \text{visited}) = \begin{cases} \text{adj}[\text{coord}][0], & \text{si todos los puntos han sido visitados} \\ \min(\text{adj}[\text{coord}][i] + \text{getShortestPerimeter}(i, \text{visited} \cup \{i\})), & \text{si } i \text{ no está visitado} \end{cases}$$
Topología:



Básico: $SP(i, 0) = adj[i][0]$

Original:

Algorithm $SP(i, v)$ // sin memoización

Input: punto i , visitados v

Output: menor distancia

```

1: if  $v = (1'(\text{beepers} + 1)) - 1$  then
2:   return  $adj[i][0]$ 
3: else
4:    $shortest = \infty$ 
5:   for all  $j \neq i$  do
6:     if  $\neg(v \& (1'j))$  then
7:        $shortest = \min(shortest, adj[i][j] + SP(j, v|(1'j)))$ 
8:   return  $shortest$ 
  
```

Algorithm $SPM(i, v)$ // con memoización

Input: punto i , visitados v

Output: menor distancia

```

1: if  $v = (1'(\text{beepers} + 1)) - 1$  then
2:   return  $adj[i][0]$ 
3: else
4:   if  $memo[i][v] = -1$  then
5:      $shortest = \infty$ 
6:     for all  $j \neq i$  do
7:       if  $\neg(v \& (1'j))$  then
8:          $shortest = \min(shortest, adj[i][j] + SPM(j, v|(1'j)))$ 
9:      $memo[i][v] = shortest$ 
10:  return  $memo[i][v]$ 
  
```

Tiempo: $SPM(i, v) \in O(2^n \cdot n^2)$

Código:

```

1 #include <bits/stdc++.h>
2 #include <cstring>
3
4 using namespace std;
5
6 int beepers;
7 int adj[11][11];
8 int memo[11][4096];
9
10 inline int getShortestPerimeter(int coordinate, int visited) {
11     if(visited == (1 << (beepers + 1)) - 1)
12         return adj[coordinate][0];
13
14     int &c = memo[coordinate][visited];
15     if(c != -1)
16         return c;
17
18     int shortest = 2000000000;
19     for(int i = 0; i <= beepers; ++i) {
20         if(i != coordinate && !(visited & (1 << i)))
21             shortest = min(shortest, adj[coordinate][i] +
22                 getShortestPerimeter(i, visited | (1 << i)));
23     }
24     return c = shortest;
25 }
  
```

```

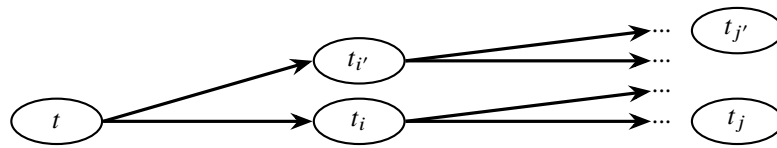
25 int main(void) {
26     ios_base::sync_with_stdio(0);
27     cin.tie(0);
28
29     int testCases;
30     pair<int, int> coordinates[11];
31     cin >> testCases;
32     while(testCases--) {
33         int width, height;
34         cin >> width >> height;
35         cin >> coordinates[0].first >> coordinates[0].second;
36         cin >> beepers;
37
38         for(int i = 1; i <= beepers; ++i)
39             cin >> coordinates[i].first >> coordinates[i].second;
40
41         for(int i = 0; i <= beepers; ++i) {
42             for(int j = i + 1; j <= beepers; ++j) {
43                 adj[i][j] = abs(coordinates[i].first -
44                     coordinates[j].first) +
45                     abs(coordinates[i].second -
46                         coordinates[j].second);
47                 adj[j][i] = adj[i][j];
48             }
49         }
50         memset(memo, -1, sizeof(memo));
51         cout << "The shortest path has length " <<
52             getShortestPerimeter(0, 1) << "\n";
53     }
54     return 0;
55 }
  
```

4.3. Problema 10364 – Square

Subproblema: Encuentre 4 subconjuntos de palos iguales $C(t) = \frac{\text{suma total de longitudes}}{4}$

Relaciones Recursivas: $C(t) = Search(C(t/4))$
 donde C busca los 4 grupos iguales de palos

Topología:



Básico: $C(t_k) = k$ con k palos usados

Original: Note que determinar el caso base requiere determinar que no hay palos posibles

Algorithm $C(t)$ // sin memoización

Input: palos t

Output: posible formar cuadrado c

```

1:  $c = \text{palos}(t)$ 
2: for each  $m \in M$  do
3:    $c' = C(t_m)$ 
4:   if  $c > c'$  then
5:      $c = c'$ 
  
```

Algorithm $CM(t)$ // con memoización

Input: palos t

Output: posible formar cuadrado c

```

1: if  $M[t]$  is undefined then
2:    $M[t] = \text{palos}(t)$ 
3:   for each  $m \in M$  do
4:      $c' = CM(t_m)$ 
5:     if  $c > c'$  then
6:        $M[t] = c'$ 
7: return  $M[t]$ 
  
```

Tiempo: $CM(t) \in O(t) \in O(M * 2^M)$ M: número de palos N: número de casos

Código:

```

1 #include <iostream>
2 #include <cstring>
3 #include <numeric>
4 using namespace std;
5
6 int M, sum, sticks[20];
7 int memo[1 << 20];
8
9 int search(int length, int bitMask)
10 {
11     if (memo[bitMask] != -1)
12         return memo[bitMask];
13
14     if (length > sum / 4)
15         return 0;
16     else if (length == sum / 4)
17     {
18         if (bitMask == (1 << M) - 1)
19             return 1;
20         length = 0;
21     }
22
23     for (int i = 0; i < M; ++i)
24         if (((bitMask & (1 << i)) == 0) && search(length + sticks[i], bitMask | (1 << i)))
25         {
26             return memo[bitMask] = 1;
27         }
28     return memo[bitMask] = 0;
29 }
30
31 int solve()
32 {
33     sum = accumulate(sticks, sticks + M, 0);
34     if (sum % 4 != 0)
35         return 0;
36
37     memset(memo, -1, sizeof(memo));
38     return search(0, 0);
39 }
40
41 int main()
42 {
43     int N;
44     cin >> N;
45     while (N--)
46     {
47         cin >> M;
48         for (int i = 0; i < M; ++i)
49             cin >> sticks[i];
50
51         cout << (solve() ? "yes" : "no") << endl;
52     }
53     return 0;
54 }

```

5. Conclusiones

En este informe hemos analizado el método SRTBOT para diseñar soluciones recursivas, enfocándonos en la implementación de programación dinámica utilizando memoización. Hemos aplicado esta técnica para resolver tres problemas complejos de programación, siendo el más complejo "Square"(10364), en el cual se verificó la viabilidad de formar un cuadrado con segmentos de diferentes longitudes.

A lo largo del proceso, hemos superado diversos desafíos en cuanto a la gestión eficiente de recursos computacionales y la implementación de soluciones optimizadas.

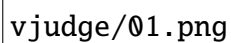
Además, hemos logrado profundizar en la integración de la técnica de memoización para mejorar el rendimiento de las soluciones recursivas, reduciendo significativamente el tiempo de ejecución en comparación con soluciones no optimizadas. Los resultados obtenidos demuestran la eficacia de la programación dinámica en la resolución de problemas complejos con restricciones específicas.

Finalmente hemos resuelto satisfactoriamente los tres problemas con la técnica de programación dinámica.

6. Anexos

En las siguientes páginas anexé el resultado de la plataforma <http://vjudge.net> al evaluar el código propuesto.

Estas impresiones fueron hechas con Chromium Browser con la impresora destino “Guardar como PDF” y con un tamaño de página “A3” para que entre en una sola.

A large rectangular box representing a screenshot of the vjudge.net evaluation results. The text 'vjudge/01.png' is located in the bottom-left corner of this box.

vjudge/01.png

vjudge/02.png

vjudge/03.png

`vjudge/perfil.pdf`