	UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS	ABET

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: 20232197, Velarde Saldaña Jhossep Fabritzio					
DOCENTES: Marcela Quispe Cruz, Manuel Loaiza, Alexander J. Benavides					

RESULTADOS Y PRUEBAS

El informe se presenta con un formato de artículo.
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 artículo aborda la programación dinámica como una técnica efectiva para resolver problemas algorítmicos complejos, optimizando el proceso mediante la división de problemas en subproblemas más simples y el uso de memoización para almacenar resultados intermedios. El objetivo fue fortalecer la comprensión teórica y práctica de esta técnica mediante la implementación de soluciones a problemas específicos en la plataforma VJudge.

En el marco teórico, se introduce el método SRTBOT, una guía que facilita el diseño de soluciones recursivas en programación dinámica, desglosada en seis etapas clave: Subproblemas, Relaciones recursivas, Topología, Bases, Original y Tiempo de ejecución.

La sección experimental detalla el proceso de selección de problemas y la aplicación de programación dinámica en su resolución, resaltando la importancia de la optimización en C++ y los beneficios de la memoización para mejorar la eficiencia de los algoritmos. Los resultados muestran cómo el uso de estas técnicas reduce significativamente la complejidad temporal en comparación con enfoques sin optimización.

Las conclusiones destacan el valor de la programación dinámica para el desarrollo de soluciones eficientes y el fortalecimiento de habilidades en el análisis y resolución de problemas algorítmicos.

1. INTRODUCCIÓN

La programación dinámica es una técnica fundamental para la resolución de problemas complejos que involucran decisiones secuenciales. Esta metodología permite descomponer un problema en subproblemas más pequeños, optimizando así la eficiencia de las soluciones mediante el almacenamiento de resultados intermedios. La importancia de la programación dinámica radica en su capacidad para resolver problemas que, de otra manera, serían computacionalmente costosos, lo que la convierte en una herramienta esencial para los programadores.

El objetivo principal de este trabajo es reforzar la comprensión del método de programación dinámica a través de la práctica. Se busca aplicar esta técnica en la resolución de problemas específicos, lo que nos permitirá familiarizarnos con los conceptos teóricos y su implementación práctica. Para lograrlo, se nos han asignado actividades concretas que involucran la selección y resolución de problemas mediante la plataforma VJudge, un sistema que facilita la evaluación automatizada de soluciones algorítmicas.

Los aportes principales de este artículo incluyen el desarrollo de soluciones utilizando la técnica SRTBOT (una metodología para abordar problemas de programación dinámica), la comparación de enfoques recursivos con y sin memoización, y la implementación de las soluciones en C++. Además, se incluirán ejemplos concretos y pseudocódigos que ilustran el uso de la programación dinámica, lo que permitirá comprender tanto la teoría como la práctica detrás de esta técnica.

En las siguientes secciones se presentará el marco teórico necesario para entender la programación dinámica, se detallará el diseño experimental y las actividades

realizadas, y se mostrarán los resultados obtenidos a partir de la resolución de los problemas seleccionados.

2. MARCO TEÓRICO CONCEPTUAL

   Propuesta por Richard Bellman en 1952, la Programación Dinámica es una técnica para abordar problemas complejos mediante la descomposición en subproblemas más simples, optimizando la solución a través del almacenamiento de resultados intermedios para evitar cálculos redundantes. La idea principal es aprovechar las propiedades de subestructura óptima y superposición de subproblemas, permitiendo que la solución de problemas grandes se construya a partir de soluciones a problemas más pequeños. Esto resulta especialmente útil en problemas donde la solución óptima puede ser compuesta a partir de soluciones parciales ya calculadas, reduciendo significativamente el tiempo de ejecución.

   La mnemotecnia SRTBOT, propuesto por Erik Demaine en 2021, es una metodología que facilita el diseño de algoritmos recursivos en el contexto de la programación dinámica. SRTBOT se divide en seis conceptos clave que permiten estructurar de manera sistemática la resolución de problemas:

   1. Subproblemas

      El primer paso consiste en dividir el problema original en subproblemas más pequeños. Para esto, es fundamental identificar la naturaleza del problema y determinar cómo puede ser fragmentado en partes que representen instancias más simples del mismo. La correcta definición de los subproblemas es crucial, ya que estos deben ser lo suficientemente pequeños para ser manejables, pero lo suficientemente significativos para contribuir a la solución del problema original.

   2. Relaciones Recursivas

      En este paso, se establecen las relaciones recursivas que permiten expresar el problema en función de sus subproblemas. Es decir, definimos cómo se puede calcular la solución de un problema en términos de soluciones a subproblemas más pequeños. Estas relaciones son el núcleo del enfoque recursivo, y la correcta formulación de las mismas determina la eficacia del algoritmo.

   3. Topología

      Dibujar la topología del problema nos ayuda a entender cómo están conectados los subproblemas entre sí. La topología puede representarse mediante un grafo o una tabla, ilustrando cómo se relacionan las diferentes soluciones parciales. Este paso es importante para visualizar la estructura de dependencias y garantizar que todos los subproblemas necesarios se resuelvan en el orden correcto.

   4. Bases

      Aquí debemos formalizar los casos base, que son las situaciones más simples del problema y que no pueden dividirse en subproblemas más pequeños. Un caso base se define como la condición que pone fin a la recursión, proporcionando una

solución directa sin necesidad de cálculos adicionales. La correcta definición de los casos base asegura que el algoritmo no caiga en bucles infinitos y tenga un punto de partida claro.

5. Original

   En este punto, debemos resolver el problema original utilizando la información de los cuatro pasos anteriores. Esto implica diseñar un algoritmo recursivo que combine las soluciones de los subproblemas siguiendo las relaciones recursivas definidas. En esta etapa, también se suele agregar memoización, que consiste en almacenar resultados de subproblemas previamente calculados para evitar la repetición de cálculos y mejorar la eficiencia del algoritmo.

6. Tiempo

   Finalmente, es crucial analizar el tiempo de ejecución de la solución propuesta. Este análisis implica evaluar la complejidad temporal del algoritmo, considerando tanto la recursividad como la memoización. El objetivo es determinar la eficiencia del algoritmo en términos de su tiempo de ejecución, identificando posibles mejoras si es necesario.

3. DISEÑO EXPERIMENTAL

En esta sección se describe el proceso seguido para seleccionar y resolver los problemas propuestos utilizando el método de programación dinámica. A lo largo de este trabajo, se emplearon técnicas específicas para diseñar algoritmos eficientes, prestando especial atención a la metodología SRTBOT para estructurar la solución de los problemas. Los resultados se evaluaron mediante la verificación automática de las soluciones en la plataforma VJudge, lo que permitió comprobar la validez y eficiencia del enfoque utilizado.

La selección de los problemas se realizó mediante una combinación de azar y la impresión inicial de que fueran interesantes a simple vista. Este enfoque permitió elegir problemas que parecían desafiantes, pero también implicó enfrentar sorpresas durante la resolución. La lógica de algunos problemas resultó más compleja de lo esperado, lo que, junto con la falta de dominio completo del lenguaje C++ utilizado, hizo que la implementación fuera un reto significativo. A pesar de estas dificultades, el esfuerzo por entender la lógica subyacente y traducirla a código efectivo permitió cumplir con los requisitos planteados en cada problema.

3.1 Objetivos

- Los objetivos de este trabajo fueron:
- Reforzar los conocimientos del método de programación dinámica a través de la resolución práctica de problemas algorítmicos.
- Aplicar el método de programación dinámica para resolver varios problemas específicos seleccionados de una lista, utilizando técnicas de optimización y análisis de eficiencia.

3.2 Actividades

En el presente trabajo se realizaron las siguientes actividades:

1. Se creó un usuario en http://vjudge.nete con el nombre de JhossepV.
2. Se seleccionaron aleatoriamente tres problemas de la lista disponible en http://bit.ly/3UxdCVL. Note que debe loguearse en la plataforma para poder ver los problemas.
3. Se diseñó una solución utilizando la técnica SRTBOT para cada problema.
4. Se mostró el pseudocódigo recursivo resultante sin y con memoización.
5. Se incluyo el código en C++ que resulta del modelo SRTBOT.
6. Se anexó el PDF del código aceptado por la plataforma http://vjudge.net al final del artículo.

4. RESULTADOS

A continuación se muestran los resultados de la resolución de los tres problemas seleccionados.

❖ **Strategic Defense Initiative**

- **Subproblema:**

  Encuentre L(i) para $0 \leq i < N$

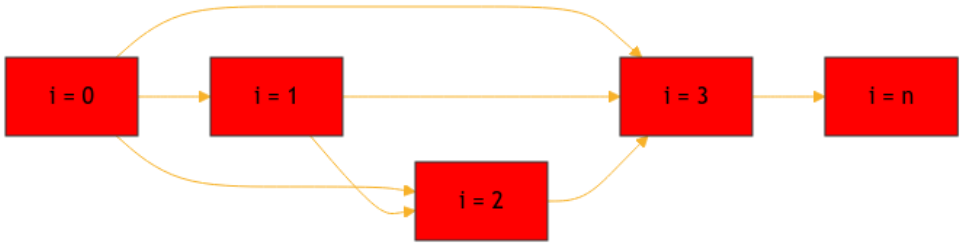
  i: posición actual del misil

  N: cantidad total de misiles

  L(i): longitud máxima de la secuencia creciente que termina en la posición i

- **Relacionar:**

  L(i) = max(L(j) + 1) para todo j < i donde altura[i] > altura[j]



- **Básico:**

  L(0) = 1 (una secuencia que sólo contiene el primer misil)

  Implementado en el código como:

```
for (int i = 0; i < cantidadMisiles; i++)
{
  longitudIntercepciones[i] = 1;
  misilAnterior[i] = -1;
}
```

Pseudocodigos:

Sin memoización:

```
función principal:
    leer numeroCasos
    ignorar la entrada de nueva línea
    leer lineaEntrada

    mientras numeroCasos > 0 hacer:
        alturaMisiles = arreglo vacío de tamaño 1000
        cantidadMisiles = 0

        mientras leer lineaEntrada hacer:
            si lineaEntrada está vacía:
                romper el bucle
            convertir lineaEntrada a entero y asignar a alturaMisiles[cantidadMisiles]
            incrementar cantidadMisiles

        maximasIntercepciones = 1
        mejorSecuencia = lista vacía

        para misilActual desde 0 hasta cantidadMisiles - 1 hacer:
            longitudActual = 1
            secuenciaActual = [alturaMisiles[misilActual]]

            para misilPrevio desde 0 hasta misilActual - 1 hacer:
                si alturaMisiles[misilActual] > alturaMisiles[misilPrevio] entonces:
                    longitudPrevio = calcular longitud máxima que termina en misilPrevio
                    si longitudPrevio + 1 > longitudActual entonces:
                        longitudActual = longitudPrevio + 1
                        secuenciaActual = [secuencia de misilPrevio] + [alturaMisiles[misilActual]]

            si longitudActual > maximasIntercepciones entonces:
                maximasIntercepciones = longitudActual
                mejorSecuencia = secuenciaActual

        imprimir "Max hits: ", maximasIntercepciones

        para cada altura en mejorSecuencia hacer:
            imprimir altura

        si numeroCasos > 1 entonces:
            imprimir línea en blanco

        decrementar numeroCasos
fin de la función
```

Con Memoización:

```
función principal:
    leer numeroCasos
    ignorar la entrada de nueva línea
    leer lineaEntrada

    mientras numeroCasos > 0 hacer:
        alturaMisiles = arreglo vacío de tamaño 1000
        cantidadMisiles = 0

        mientras leer lineaEntrada hacer:
            si lineaEntrada está vacía:
                romper el bucle
            convertir lineaEntrada a entero y asignar a alturaMisiles[cantidadMisiles]
            incrementar cantidadMisiles

        longitudIntercepciones = arreglo de 1000 elementos, todos inicializados en 1
        misilAnterior = arreglo de 1000 elementos, todos inicializados en -1
        maximasIntercepciones = 1
        ultimoMisilInterceptado = 0

        para misilActual desde 1 hasta cantidadMisiles - 1 hacer:
            para misilPrevio desde 0 hasta misilActual - 1 hacer:
                si alturaMisiles[misilActual] > alturaMisiles[misilPrevio] y
                    longitudIntercepciones[misilPrevio] + 1 > longitudIntercepciones[misilActual] entonces:

                    longitudIntercepciones[misilActual] = longitudIntercepciones[misilPrevio] + 1
                    misilAnterior[misilActual] = misilPrevio

                    si longitudIntercepciones[misilActual] > maximasIntercepciones entonces:
                        maximasIntercepciones = longitudIntercepciones[misilActual]
                        ultimoMisilInterceptado = misilActual

        imprimir "Max hits: ", maximasIntercepciones

        secuenciaIntercepciones = arreglo vacío
        totalIntercepciones = 0

        para i desde ultimoMisilInterceptado hasta -1, siguiendo misilAnterior[i] hacer:
            agregar alturaMisiles[i] a secuenciaIntercepciones
            incrementar totalIntercepciones

        para i desde totalIntercepciones - 1 hasta 0 hacer:
            imprimir secuenciaIntercepciones[i]

        si numeroCasos > 1 entonces:
            imprimir línea en blanco

        decrementar numeroCasos
fin de la función
```

- **Código:**

```cpp
#include <iostream>
using namespace std;

int main(){
  int numeroCasos;
  cin >> numeroCasos;
  cin.ignore();

  string lineaEntrada;
  getline(cin, lineaEntrada);

  // Trabajamos cada caso de prueba de misiles entrantes
  while (numeroCasos--)
  {
    int alturaMisiles[1000];
    int cantidadMisiles = 0;

    // Leemos las alturas de los misiles uno por uno
    // hasta encontrar una línea vacía que separa los casos
    while (getline(cin, lineaEntrada))
    {
      if (lineaEntrada.empty())
        break;
      alturaMisiles[cantidadMisiles++] = stoi(lineaEntrada);
    }

    // Arrays para encontrar la secuencia más larga de misiles que podemos interceptar
    int longitudIntercepciones[1000]; // Guarda cuántos misiles podemos interceptar hasta cada posición
    int misilAnterior[1000];          // Guarda la posición del misil anterior que interceptamos
    int maximasIntercepciones = 1;    // Cantidad máxima de misiles que podemos interceptar
    int ultimoMisilInterceptado = 0;  // Posición del último misil en la mejor secuencia

    // Inicializra los arrays: cada misil empieza con longitud 1
    for (int i = 0; i < cantidadMisiles; i++)
    {
      longitudIntercepciones[i] = 1;
      misilAnterior[i] = -1;
    }

    // Buscamos la secuencia más larga de misiles que podemos interceptar
    // teniendo en cuenta que cada misil interceptado debe estar a mayor altura que el anterior
    for (int misilActual = 1; misilActual < cantidadMisiles; misilActual++)
    {
      for (int misilPrevio = 0; misilPrevio < misilActual; misilPrevio++)
      {
        if (alturaMisiles[misilActual] > alturaMisiles[misilPrevio] &&
            longitudIntercepciones[misilPrevio] + 1 > longitudIntercepciones[misilActual])
        {
          longitudIntercepciones[misilActual] = longitudIntercepciones[misilPrevio] + 1;
          misilAnterior[misilActual] = misilPrevio;
          if (longitudIntercepciones[misilActual] > maximasIntercepciones)
          {
            maximasIntercepciones = longitudIntercepciones[misilActual];
            ultimoMisilInterceptado = misilActual;
          }
        }
      }
    }

    // Print del total de misiles que podemos interceptar
    cout << "Max hits: " << maximasIntercepciones << endl;

    // Reconstruimos la secuencia de misiles que vamos a interceptar
    int secuenciaIntercepciones[1000];
    int totalIntercepciones = 0;
    for (int i = ultimoMisilInterceptado; i != -1; i = misilAnterior[i])
    {
      secuenciaIntercepciones[totalIntercepciones++] = alturaMisiles[i];
    }

    // Mostramos las alturas de los misiles que interceptaremos, en orden de llegada
    for (int i = totalIntercepciones - 1; i >= 0; i--)
    {
      cout << secuenciaIntercepciones[i] << endl;
    }

    // Agregamos línea en blanco entre casos (excepto después del último caso)
    if (numeroCasos)
      cout << endl;
  }
  return 0;
}
```

- **Tiempo:**

  Sin memoización: O(2^n)

  - Cada posición genera dos llamadas recursivas

  Con memoización: O(n²)

  - Dos bucles anidados
  - El externo itera n veces
  - El interno itera i veces en cada paso i

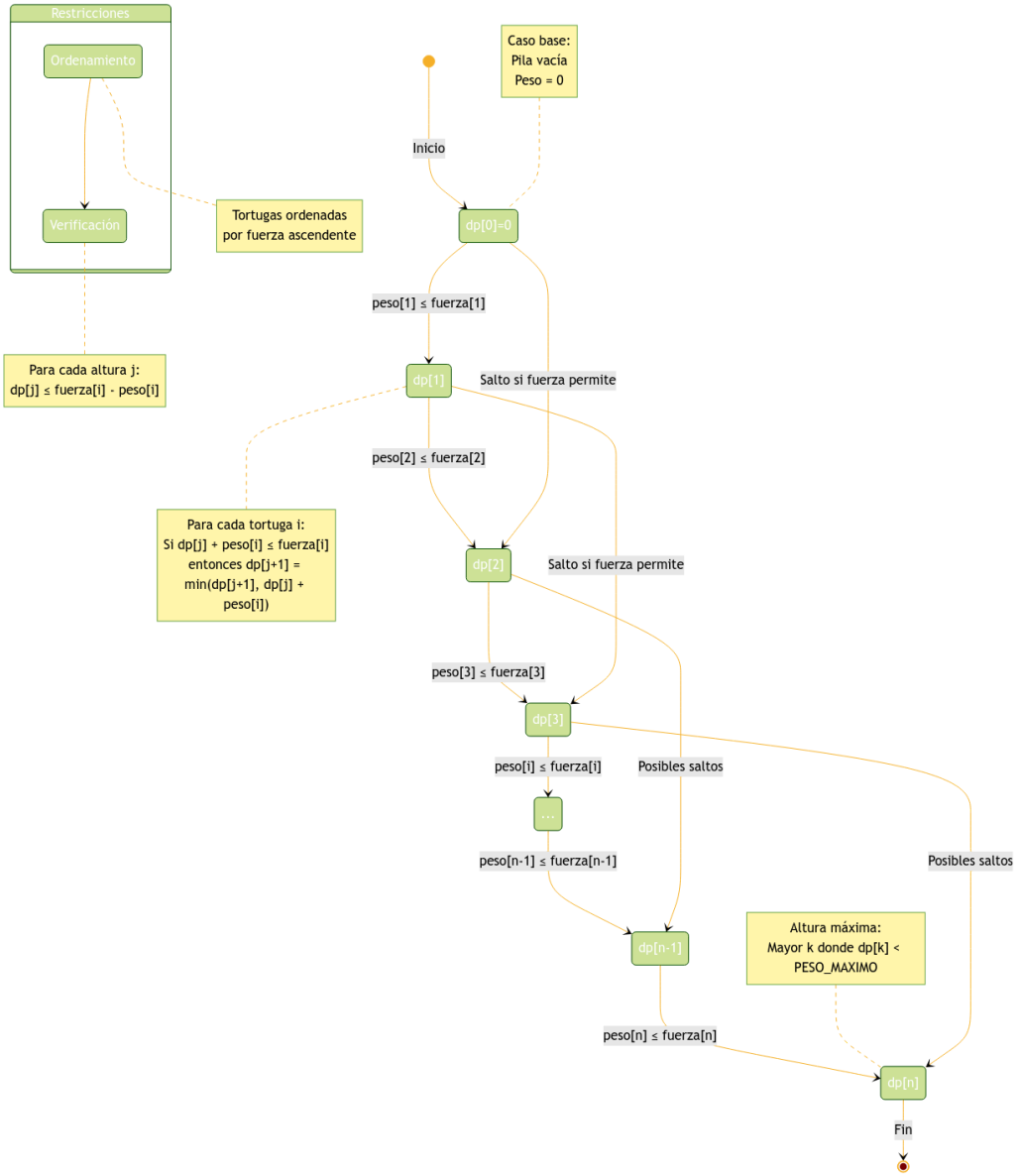
❖ **Weights and Measures**

➢ **Subproblema**:

Encontrar dp[i] para 0 ≤ i ≤ n, donde dp[i] representa el peso mínimo acumulado para una pila de altura i

➢ **Relacionar**:

dp[i] = min(dp[i], dp[i-1] + peso[j]) si dp[i-1] + peso[j] ≤ fuerza[j] donde j representa la tortuga actual siendo considerada



➢ **Básico:**

dp[0] = 0 (pila vacía)

Para toda otra altura inicial, dp[i] = PESO_MAXIMO

Pseudocodigos:

Sin Memoización:

```
función encontrarAlturaMaxima(peso, fuerza, n, altura_actual, peso_actual):
    si peso_actual > PESO_MAXIMO entonces:
        retornar altura_actual - 1

    max_altura = altura_actual

    para i desde 0 hasta n - 1 hacer:
        si peso_actual <= fuerza[i] - peso[i] entonces:
            nueva_altura = encontrarAlturaMaxima(peso, fuerza, n, altura_actual + 1, peso_actual + peso[i])
            max_altura = máximo(max_altura, nueva_altura)

    retornar max_altura
```

Con Memoización:

```
función encontrarAlturaMaximaMemo(peso, fuerza, n):
    dp = arreglo de tamaño n + 1, inicializado en PESO_MAXIMO
    dp[0] = 0
    altura_maxima = 0

    para i desde 0 hasta n - 1 hacer:
        para j desde altura_maxima hasta 0, en decremento hacer:
            si dp[j] <= fuerza[i] - peso[i] entonces:
                dp[j + 1] = mínimo(dp[j + 1], dp[j] + peso[i])
                altura_maxima = máximo(altura_maxima, j + 1)

    retornar altura_maxima
```

Código:

```cpp
#include <iostream>
using namespace std;

const int MAX_TORTUGAS = 5607;    // Máximo número de tortugas
const int PESO_MAXIMO = 10000000; // Peso muy alto para representar un límite exagerado

int peso[MAX_TORTUGAS];   // Arreglo para almacenar el peso de cada tortuga
int fuerza[MAX_TORTUGAS]; // Arreglo para almacenar la fuerza de cada tortuga
int dp[MAX_TORTUGAS + 1]; // Arreglo de DP para el peso mínimo de una pila con cierta altura

// Función para ordenar las tortugas por su fuerza usando un algoritmo de burbuja
void ordenarTortugas(int n)
{
  for (int i = 0; i < n - 1; ++i)
  {
    for (int j = 0; j < n - i - 1; ++j)
    {
      if (fuerza[j] > fuerza[j + 1])
      {
        // Intercambiar peso de las tortugas
        int tempPeso = peso[j];
        peso[j] = peso[j + 1];
        peso[j + 1] = tempPeso;

        // Intercambiar fuerza de las tortugas
        int tempFuerza = fuerza[j];
        fuerza[j] = fuerza[j + 1];
        fuerza[j + 1] = tempFuerza;
      }
    }
  }
}

int main(){
  int p, f;
  int n = 0; // Contador de tortugas

  // Leer datos de entrada de cada tortuga
  while (cin >> p >> f)
  {
    peso[n] = p;
    fuerza[n] = f;
    ++n;
  }

  // Ordenar tortugas por su fuerza para facilitar la construcción de la pila
  ordenarTortugas(n);

  // Inicializar dp con un valor grande para representar peso imposible
  for (int i = 0; i <= n; ++i)
  {
    dp[i] = PESO_MAXIMO;
  }
  dp[0] = 0; // Peso de una pila vacía es 0

  int alturaMaxima = 0; // Máxima altura de la pila de tortugas que se puede construir

  // Intentar agregar cada tortuga a una pila
  for (int i = 0; i < n; ++i)
  {
    // Recorrer dp desde el final para evitar sobrescribir valores previos
    for (int j = alturaMaxima; j >= 0; --j)
    {
      // Verificar si podemos agregar la tortuga actual sin exceder su fuerza
      if (dp[j] <= fuerza[i] - peso[i])
      {
        // Actualizar el peso mínimo para una pila de altura j+1 si esta combinación es válida
        dp[j + 1] = (dp[j + 1] < dp[j] + peso[i]) ? dp[j + 1] : (dp[j] + peso[i]);
        // Actualizar la altura máxima alcanzada
        alturaMaxima = (alturaMaxima > j + 1) ? alturaMaxima : j + 1;
      }
    }
  }

  // Imprimir la altura máxima de la pila que se puede construir sin que la tortuga ripee
  cout << alturaMaxima << endl;

  return 0;
}
```

- **Tiempo:**
  - **Sin memoización:** O(n^n) - Cada tortuga puede intentar colocarse en cada posición posible
  - **Con memoización:** O(n^2) - Para cada tortuga probamos cada altura posible una sola vez

❖ **Homer Simpson**

➔ **Subproblema:**

Encontrar dp[t] para 0 ≤ t ≤ T, donde dp[t] representa la máxima cantidad de hamburguesas que se pueden comer en t minutos.

➔ **Relacionar:**

Para cada tiempo t:

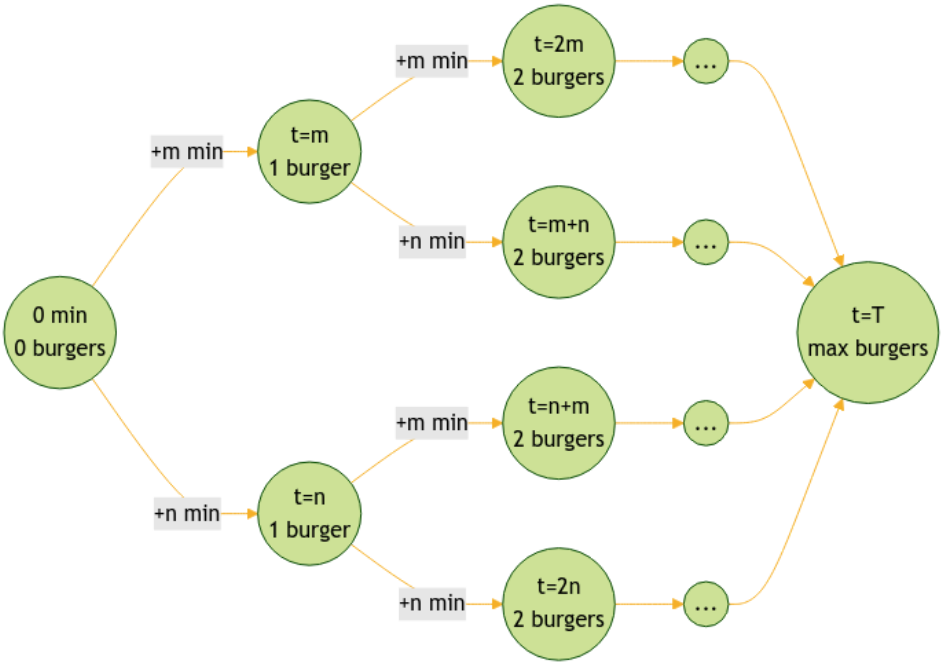
dp[t] = max(

dp[t],

dp[t-m] + 1 si t-m ≥ 0,

dp[t-n] + 1 si t-n ≥ 0

)



➔ **Básico:**

dp[0] = 0 (0 minutos = 0 hamburguesas)

dp[t] = -1 para t > 0 (estados iniciales no alcanzados)

Pseudocodigos:

Sin Memoización:

```
función maxBurgers(t, m, n):
    si t < 0 entonces:
        retornar -infinito
    si t == 0 entonces:
        retornar 0
    retornar máximo(
        maxBurgers(t - m, m, n) + 1,
        maxBurgers(t - n, m, n) + 1
    )
```

Con Memoización:

```
función maxBurgersDP(t, m, n):
    dp = arreglo de tamaño t + 1, inicializado en -1
    dp[0] = 0

    para tiempo desde 0 hasta t hacer:
        si dp[tiempo] no es -1 entonces:
            si tiempo + m <= t entonces:
                dp[tiempo + m] = máximo(dp[tiempo + m], dp[tiempo] + 1)
            si tiempo + n <= t entonces:
                dp[tiempo + n] = máximo(dp[tiempo + n], dp[tiempo] + 1)

    retornar máximo((dp[i], t - i) para i desde t hasta 0 en decremento si dp[i] no es -1)[0]
```

➔ **Código:**

```cpp
#include <iostream>
using namespace std;

int main(){
  //Try 5 -_-
  int m, n, t;

  // Leemos casos hasta EOF
  while (cin >> m >> n >> t)
  {
    // Array para programación dinámica
    int dp[10001];

    // Inicializamos el array con -1
    for (int i = 0; i <= t; i++)
    {
      dp[i] = -1;
    }
    dp[0] = 0; // Caso base: 0 minutos = 0 hamburguesas

    // Calculamos la cantidad máxima de hamburguesas para cada minuto
    for (int tiempo = 0; tiempo <= t; tiempo++)
    {
      if (dp[tiempo] != -1)
      {
        // Intentamos agregar una hamburguesa que toma m minutos
        if (tiempo + m <= t)
        {
          if (dp[tiempo + m] < dp[tiempo] + 1)
          {
            dp[tiempo + m] = dp[tiempo] + 1;
          }
        }
```

```
    // Intentamos agregar una hamburguesa que toma n minutos
    if (tiempo + n <= t)
    {
      if (dp[tiempo + n] < dp[tiempo] + 1)
      {
        dp[tiempo + n] = dp[tiempo] + 1;
      }
    }
  }
}

// Buscamos la mejor solución de atrás hacia adelante
int maxBurgers = 0;
int tiempoSobrante = 0;

for (int tiempo = t; tiempo >= 0; tiempo--)
{
  if (dp[tiempo] != -1)
  {
    maxBurgers = dp[tiempo];
    tiempoSobrante = t - tiempo;
    break;
  }
}

// Print del resultado
if (tiempoSobrante == 0)
{
  cout << maxBurgers << endl;
}
else
{
  cout << maxBurgers << " " << tiempoSobrante << endl;
}
}

return 0;
}
```

➔ **Tiempo:**

- **Sin memoización:** O(2^t) - Para cada minuto, tenemos dos opciones recursivas
- **Con memoización:** O(t) - Cada estado se calcula una sola vez

5. CONCLUSIONES

En este trabajo, se aplicó la técnica de programación dinámica para resolver problemas algorítmicos, usando el método SRTBOT como guía para estructurar cada solución. La implementación incluyó el uso de memoización, lo que optimizó el tiempo de ejecución al evitar cálculos repetidos.

Los principales logros incluyen la comparación de enfoques recursivos con y sin memoización y la implementación de algoritmos en C++ que ejemplifican los conceptos de subestructura óptima y superposición de subproblemas.

Se presentaron algunos desafíos en la formulación de las relaciones recursivas y en la traducción de la lógica a código eficiente, pero estos fueron superados, cumpliendo así

los objetivos de fortalecer el conocimiento práctico y teórico en programación dinámica.

6. REFERENCIAS BIBLIOGRÁFICAS

- Bellman, R. (1952). On the theory of dynamic programming. Proceedings of the National Academy of Sciences, 38(8), 716-719.
- Demaine, E. (2021). Dynamic Programming, Part 1: SRTBOT, Fib, DAGs, Bowling. MIT OpenCourseWare. http://youtu.be/r4-cftqTcdI

7. ANEXOS

   En esta sección se adjuntan las capturas de pantalla en formato A2 para que ingrese todo el código en una captura del formato de aceptación de Vjudge.net

#55997721 | JhossepV's solution for [UVA-497]

Status	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	2897	C++11 5.3.0	2024-11-13 14:54:08	☑	☐	29960588

```cpp
#include <iostream>
using namespace std;

int main()
{
  int numeroCasos;
  cin >> numeroCasos;
  cin.ignore();

  string lineaEntrada;
  getline(cin, lineaEntrada);

  // Trabajamos cada caso de prueba de misiles entrantes
  while (numeroCasos--)
  {
    int alturaMisiles[1000];
    int cantidadMisiles = 0;

    // Leemos las alturas de los misiles uno por uno
    // hasta encontrar una línea vacía que separa los casos
    while (getline(cin, lineaEntrada))
    {
      if (lineaEntrada.empty())
        break;
      alturaMisiles[cantidadMisiles++] = stoi(lineaEntrada);
    }

    // Arrays para encontrar la secuencia más larga de misiles que podemos interceptar
    int longitudIntercepciones[1000]; // Guarda cuántos misiles podemos interceptar hasta cada
posición
    int misilAnterior[1000];          // Guarda la posición del misil anterior que interceptamos
    int maximasIntercepciones = 1;    // Cantidad máxima de misiles que podemos interceptar
    int ultimoMisilInterceptado = 0;  // Posición del último misil en la mejor secuencia

    // Inicializra los arrays: cada misil empieza con longitud 1
    for (int i = 0; i < cantidadMisiles; i++)
    {
      longitudIntercepciones[i] = 1;
      misilAnterior[i] = -1;
    }

    // Buscamos la secuencia más larga de misiles que podemos interceptar
    // teniendo en cuenta que cada misil interceptado debe estar a mayor altura que el anterior
    for (int misilActual = 1; misilActual < cantidadMisiles; misilActual++)
    {
      for (int misilPrevio = 0; misilPrevio < misilActual; misilPrevio++)
      {
        if (alturaMisiles[misilActual] > alturaMisiles[misilPrevio] &&
            longitudIntercepciones[misilPrevio] + 1 > longitudIntercepciones[misilActual])
        {
          longitudIntercepciones[misilActual] = longitudIntercepciones[misilPrevio] + 1;
          misilAnterior[misilActual] = misilPrevio;
          if (longitudIntercepciones[misilActual] > maximasIntercepciones)
          {
            maximasIntercepciones = longitudIntercepciones[misilActual];
            ultimoMisilInterceptado = misilActual;
          }
        }
      }
    }

    // Print del total de misiles que podemos interceptar
    cout << "Max hits: " << maximasIntercepciones << endl;

    // Reconstruimos la secuencia de misiles que vamos a interceptar
    int secuenciaIntercepciones[1000];
    int totalIntercepciones = 0;
    for (int i = ultimoMisilInterceptado; i != -1; i = misilAnterior[i])
    {
      secuenciaIntercepciones[totalIntercepciones++] = alturaMisiles[i];
    }

    // Mostramos las alturas de los misiles que interceptaremos, en orden de llegada
    for (int i = totalIntercepciones - 1; i >= 0; i--)
    {
      cout << secuenciaIntercepciones[i] << endl;
    }

    // Agregamos línea en blanco entre casos (excepto después del último caso)
    if (numeroCasos)
      cout << endl;
  }
  return 0;
}
```

#55998231 | JhossepV's solution for [UVA-10465]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	120ms	1500	C++11 5.3.0	2024-11-13 16:19:09			29960657

C++

```cpp
#include <iostream>
using namespace std;

int main()
{
  //Try 5 -_-
  int m, n, t;

  // Leemos casos hasta EOF
  while (cin >> m >> n >> t)
  {
    // Array para programación dinámica
    int dp[10001];

    // Inicializamos el array con -1
    for (int i = 0; i <= t; i++)
    {
      dp[i] = -1;
    }
    dp[0] = 0; // Caso base: 0 minutos = 0 hamburguesas

    // Calculamos la cantidad máxima de hamburguesas para cada minuto
    for (int tiempo = 0; tiempo <= t; tiempo++)
    {
      if (dp[tiempo] != -1)
      {
        // Intentamos agregar una hamburguesa que toma m minutos
        if (tiempo + m <= t)
        {
          if (dp[tiempo + m] < dp[tiempo] + 1)
          {
            dp[tiempo + m] = dp[tiempo] + 1;
          }
        }

        // Intentamos agregar una hamburguesa que toma n minutos
        if (tiempo + n <= t)
        {
          if (dp[tiempo + n] < dp[tiempo] + 1)
          {
            dp[tiempo + n] = dp[tiempo] + 1;
          }
        }
      }
    }

    // Buscamos la mejor solución de atrás hacia adelante
    int maxBurgers = 0;
    int tiempoSobrante = 0;

    for (int tiempo = t; tiempo >= 0; tiempo--)
    {
      if (dp[tiempo] != -1)
      {
        maxBurgers = dp[tiempo];
        tiempoSobrante = t - tiempo;
        break;
      }
    }

    // Print del resultado
    if (tiempoSobrante == 0)
    {
      cout << maxBurgers << endl;
    }
    else
    {
      cout << maxBurgers << " " << tiempoSobrante << endl;
    }
  }

  return 0;
}
```

#55998343 | JhossepV's solution for [UVA-10154]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	30ms	2337	C++11 5.3.0	2024-11-13 16:38:40			29960670

C++

```cpp
#include <iostream>
using namespace std;

const int MAX_TORTUGAS = 5607;    // Máximo número de tortugas
const int PESO_MAXIMO = 10000000; // Peso muy alto para representar un límite exagerado

int peso[MAX_TORTUGAS];   // Arreglo para almacenar el peso de cada tortuga
int fuerza[MAX_TORTUGAS]; // Arreglo para almacenar la fuerza de cada tortuga
int dp[MAX_TORTUGAS + 1]; // Arreglo de DP para el peso mínimo de una pila con cierta altura

// Función para ordenar las tortugas por su fuerza usando un algoritmo de burbuja
void ordenarTortugas(int n)
{
  for (int i = 0; i < n - 1; ++i)
  {
    for (int j = 0; j < n - i - 1; ++j)
    {
      if (fuerza[j] > fuerza[j + 1])
      {
        // Intercambiar peso de las tortugas
        int tempPeso = peso[j];
        peso[j] = peso[j + 1];
        peso[j + 1] = tempPeso;

        // Intercambiar fuerza de las tortugas
        int tempFuerza = fuerza[j];
        fuerza[j] = fuerza[j + 1];
        fuerza[j + 1] = tempFuerza;
      }
    }
  }
}

int main()
{
  int p, f;
  int n = 0; // Contador de tortugas

  // Leer datos de entrada de cada tortuga
  while (cin >> p >> f)
  {
    peso[n] = p;
    fuerza[n] = f;
    ++n;
  }

  // Ordenar tortugas por su fuerza para facilitar la construcción de la pila
  ordenarTortugas(n);

  // Inicializar dp con un valor grande para representar peso imposible
  for (int i = 0; i <= n; ++i)
  {
    dp[i] = PESO_MAXIMO;
  }
  dp[0] = 0; // Peso de una pila vacía es 0

  int alturaMaxima = 0; // Máxima altura de la pila de tortugas que se puede construir

  // Intentar agregar cada tortuga a una pila
  for (int i = 0; i < n; ++i)
  {
    // Recorrer dp desde el final para evitar sobrescribir valores previos
    for (int j = alturaMaxima; j >= 0; --j)
    {
      // Verificar si podemos agregar la tortuga actual sin exceder su fuerza
      if (dp[j] <= fuerza[i] - peso[i])
      {
        // Actualizar el peso mínimo para una pila de altura j+1 si esta combinación es válida
        dp[j + 1] = (dp[j + 1] < dp[j] + peso[i]) ? dp[j + 1] : (dp[j] + peso[i]);
        // Actualizar la altura máxima alcanzada
        alturaMaxima = (alturaMaxima > j + 1) ? alturaMaxima : j + 1;
      }
    }
  }

  // Imprimir la altura máxima de la pila que se puede construir sin que la tortuga ripee
  cout << alturaMaxima << endl;

  return 0;
}
```