**UNIVERSIDAD NACIONAL DE TRUJILLO**

**Facultad de Ciencias Físicas y Matemáticas**

**Escuela Académico Profesional de Informática**



**PARADIGMA DE ALGORITMO DINÁMICO PARA RESOLVER EL PROBLEMA DE KNAPSACK**

**Espino Mostacero Richard Francisco**

**GUADALUPE- LA LIBERTAD**

**2019**

**RESUMEN**

Actualmente el campo de la programación está siendo cada vez más competitivo, entre ellos se toman en cuenta diversos puntos, tales como, la optimización que se puede dar a un algoritmo, el espacio de memoria que ocupa y sobre todo la complejidad que esta puede llegar a tener, la expectativa de esta investigación tiene como objetivo dar a conocer que existen varios caminos que pueden tomar los desarrolladores de software para realizar un determinado problema, para ello debemos tener una noción clara de lo que respecta a Paradigmas de algoritmos. Si bien es cierto cuando desarrollamos un programa a la mayoría de programadores no les importa el grado de complejidad, ya que solo se necesita resolver un problema, pero a gran escala este conllevará a una perdida alta de memoria y de eficiencia, es allí donde entra en juego el Paradigma de Algoritmos, este es un alto nivel a la hora de programar ya que estamos llevando nuestra capacidad a un nivel más alto ya que utilizamos diversas técnicas para lograr la eficiencia de dicho programa.

En esta investigación nos enfocaremos en el Paradigma de algoritmo Dinámico, ya que es uno de los más óptimos y además uno de los más fáciles de implementar, exactamente desarrollaremos el problema de la mochila, estudiando su comportamiento y su complejidad.

**ÍNDICE GENERAL**

**Resumen 2**

**1. Introducción 5**

1.1 Objetivo de la Investigación5

1.2 Importancia de la investigación 5

**2. Aplicaciones del paradigma de algoritmo Dinámico 6**

2.1 ¿Qué es el Paradigma de Algoritmos? 6

2.2 Paradigma de algoritmo Dinámico 7

**3. Problema de la mochila 8**

3.1 Formulación del Problema de la Mochila 8

3.2 ¿Cómo resolver el Problema de la Mochila? 9

3.3 Algoritmo Dinámico para el Problema de la Mochila 10

3.4 Complejidad para el Problema de la Mochila 10

**4. Optimización del algoritmo dinámico 11**

4.1 ¿Cómo optimizar nuestro algoritmo? 11

4.2 Codificación del Algoritmo Dinámico13

4.3 Solución y ejecución para el Problema de la Mochila14

**Capítulo 1**

**Introducción**

**1.1 Objetivo de la Investigación**

Debido a que muchos programadores han desarrollado sistemas que han sido a largo plazo un completo fracaso, el estudio de los algoritmos y complejidad es de carácter indispensable para el desarrollo de software. Esto conlleva a tener una mayor habilidad a la hora de programar y lograremos que nuestros algoritmos logren un mayor grado de eficiencia tanto en tiempo como en memoria. Esta investigación se basa en una de las técnicas más utilizadas por los programadores de alto nivel, y será de mayor utilidad para adentrarnos en la complejidad algorítmica.

**1.2 Importancia de la Investigación**

El Paradigma de Algoritmos y la teoría de la complejidad es muy importante para la ciencia de la computación ya que, si no se tiene un concepto muy claro de complejidad a la hora de implementar un algoritmo, a gran escala consumirá más memoria, más recursos y un posible colapso de datos dentro del sistema.

Es por ello que en ciencias de la computación saber estrictamente si un algoritmo es mejor o no va a depender de la complejidad en el contexto en el que nos encontremos, porque si queremos resolver un problema para pocos datos, lo más recomendable es utilizar un algoritmo polinomial, pero si tenemos muchos datos será muy difícil encontrar un algoritmo polinomial que pueda resolver este dilema, entonces podríamos utilizar un algoritmo para resolver un problema dentro de la clase NP que desde luego no nos dará una solución exacta solo nos informará si el problema puede ser resuelto o no.

La Programación Dinámica es un método de optimización de extraordinaria versatilidad. Si bien fue desarrollada especialmente para la resolución de problemas en Procesos de Decisión en Múltiples Pasos, diferentes investigaciones han mostrado que las mismas ideas pueden utilizarse en otro tipo de problemas de matemática aplicada, e incluso pueden ser útiles en el planteo de algunas cuestiones teóricas. Habiendo surgido en los inicios de la época de las computadoras, la Programación Dinámica fue, además, concebida con un ojo puesto en esta potente herramienta. La Ecuación Funcional que se obtiene, para cada problema, a través del uso del Principio de Optimalidad de Bellman permite, con mayor o menor esfuerzo dependiendo del caso, establecer una recurrencia que es, en sí misma, un algoritmo que resuelve el problema en cuestión.

**Capítulo 2**

**Aplicaciones del paradigma de algoritmo Dinámico**

**2.1 ¿Qué es el Paradigma de Algoritmos?**

Así como existen distintos caminos para resolver un problema en la vida, en la programación también existe diversos caminos que nos conllevan a una solución, esta dependerá del programador para lograr una mayor optimización.

Primero recordemos el paradigma de programación, que es la metodología que podemos seguir para lograr una solución entre ellas tenemos: Lógico, Imperativo, Funcional, Estructurado, etc.

Si tenemos paradigmas que nos muestran la forma en la que podemos resolver un determinado problema, el Paradigma de Algoritmos nos permite desarrollar técnicas de programación para hacer un programa más rápido y más eficiente.

**2.2 Paradigma de Algoritmo Dinámico**

*Una sub estructura óptima* significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Por ejemplo, el [camino más corto](https://es.wikipedia.org/wiki/Problema_del_camino_m%C3%A1s_corto) entre dos vértices de un [grafo](https://es.wikipedia.org/wiki/Grafo) se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

Dividir el problema en subproblemas más pequeños.

Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.

Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

Decir que un problema tiene *subproblemas superpuestos* es decir que se usa un mismo subproblema para resolver diferentes problemas mayores. Por ejemplo, en la [sucesión de Fibonacci](https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci) (F3 = F1 + F2 y F4 = F2 + F3) calcular cada término supone calcular F2. Como para calcular F5 hacen falta tanto F3 como F4, una mala implementación para calcular F5 acabará calculando F2 dos o más veces. Esto sucede siempre que haya subproblemas superpuestos: una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a problemas que ya han sido resueltos anteriormente.

Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama [*memorización*](https://es.wikipedia.org/w/index.php?title=Memoizaci%C3%B3n&action=edit&redlink=1) (no confundir con [memorización](https://es.wikipedia.org/wiki/Memorizaci%C3%B3n); en inglés es llamado memorización, véase [en](https://en.wikipedia.org/wiki/memoization#Etymology)). Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que de antemano sabemos que vamos a necesitar.

En resumen, la programación hace uso de:

-[Subproblemas superpuestos](https://es.wikipedia.org/w/index.php?title=Subproblema_superpuesto&action=edit&redlink=1)

-[Subestructuras óptimas](https://es.wikipedia.org/w/index.php?title=Estructura_%C3%B3ptima&action=edit&redlink=1)

-[Memorización](https://es.wikipedia.org/w/index.php?title=Memoizaci%C3%B3n&action=edit&redlink=1)

La programación toma normalmente uno de los dos siguientes enfoques:

[**Top-down**](https://es.wikipedia.org/wiki/Top-down): El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de [memorización](https://es.wikipedia.org/w/index.php?title=Memoizaci%C3%B3n&action=edit&redlink=1) y [recursión](https://es.wikipedia.org/wiki/Recursi%C3%B3n).

[**Bottom-up**](https://es.wikipedia.org/wiki/Bottom-up): Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Originalmente, el término de *programación dinámica* se refería a la resolución de ciertos problemas y operaciones fuera del ámbito de la [Ingeniería Informática](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_Inform%C3%A1tica), al igual que hacía la [*programación lineal*](https://es.wikipedia.org/wiki/Programaci%C3%B3n_lineal). Aquel contexto no tiene relación con la [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n) en absoluto; el nombre es una coincidencia. El término también lo usó en los [años 40](https://es.wikipedia.org/wiki/A%C3%B1os_1940) [Richard Bellman](https://es.wikipedia.org/wiki/Richard_Bellman), un matemático norteamericano, para describir el proceso de resolución de problemas donde hace falta calcular la mejor solución consecutivamente.

Algunos [lenguajes de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) [funcionales](https://es.wikipedia.org/wiki/Paradigma_funcional), sobre todo [Haskell](https://es.wikipedia.org/wiki/Haskell" \o "Haskell), pueden usar la [memorización](https://es.wikipedia.org/wiki/Memorizaci%C3%B3n) automáticamente sobre funciones con un conjunto concreto de argumentos, para acelerar su proceso de evaluación. Esto sólo es posible en funciones que no tengan [efectos secundarios](https://es.wikipedia.org/wiki/Efecto_secundario_(computaci%C3%B3n)), algo que ocurre en [Haskell](https://es.wikipedia.org/wiki/Haskell" \o "Haskell) pero no tanto en otros lenguajes.

**Capítulo 3**

**Problema de la mochila**

**3.1 Formulación del Problema de la Mochila**

Dada una mochila con una capacidad máxima **W**, y un conjunto **S** de n elementos, cada elemento **i** tiene un peso y un valor o beneficio de ( y W son enteros), pero el problema es **¿Cómo empacar la mochila para lograr un máximo valor o beneficio total de los elementos empacados?**

¿Cómo lo podemos resolver?

Una de las soluciones no tan recomendadas es hacerlo por Fuerza Bruta:

1. Dado que hay n elementos, puede haber posibles combinaciones.
2. Ir a través de todas las combinaciones y encontrar aquella cuyo peso sea menor que la capacidad W de la mochila y que maximice el valor o beneficio
3. Su complejidad será de ***O().*** Como sabemos esta complejidad estará dada por un tiempo exponencial no es un problema tratable para grandes valores de n, en este caso el algoritmo consumirá bastante tiempo y memoria.

Una de la solución más óptima seria es aplicar el Paradigma de Algoritmo Dinámico:

Tenemos que identificar la **ecuación recursiva** que asocie el problema con los subproblemas.

Dado los siguientes elementos con sus respectivos pesos y beneficion, hablar la solución más óptima.

|  |  |  |
| --- | --- | --- |
|  | Peso | Beneficio |
| Elemento |  |  |
| 1 | 2 | 3 |
| 2 | 4 | 5 |
| 3 | 5 | 8 |
| 4 | 3 | 4 |
| 5 | 9 | 10 |

* Si los elementos están etiquetados 1…n, entonces el subproblema será encontrar la solución más óptima para **= {elementos etiquetados 1, 2, 3… k}.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| = 2  =3 | = 4  =5 | = 5  =8 | = 3  =4 |  |

Peso Max: W=20

**Para :**

Peso Total: 14

Beneficio Max: 20

|  |  |  |  |
| --- | --- | --- | --- |
| = 2  =3 | = 4  =5 | = 5  =8 | = 9  =10 |

**Para S5:**

Peso Total: 20

Beneficio Max: 26 (Notamos que el beneficio crece)

* La solución para no es parte de la solución para
* Viola el principio de la optimalidad. La solución óptima del problema debe incluir la solución óptima de sus subproblemas

**3.2 ¿Cómo resolver el Problema de la Mochila?**

La forma en la que si podemos resolver este problema es agregando un nuevo parámetro: **w**, que representa el peso máximo de cada subconjunto de elementos.

El subproblema entonces será calcular **V[k,w]** para encontrar la solución óptima para = {elementos etiquetados 1,2,3…k} que entren en una mochila de tamaño **w**.

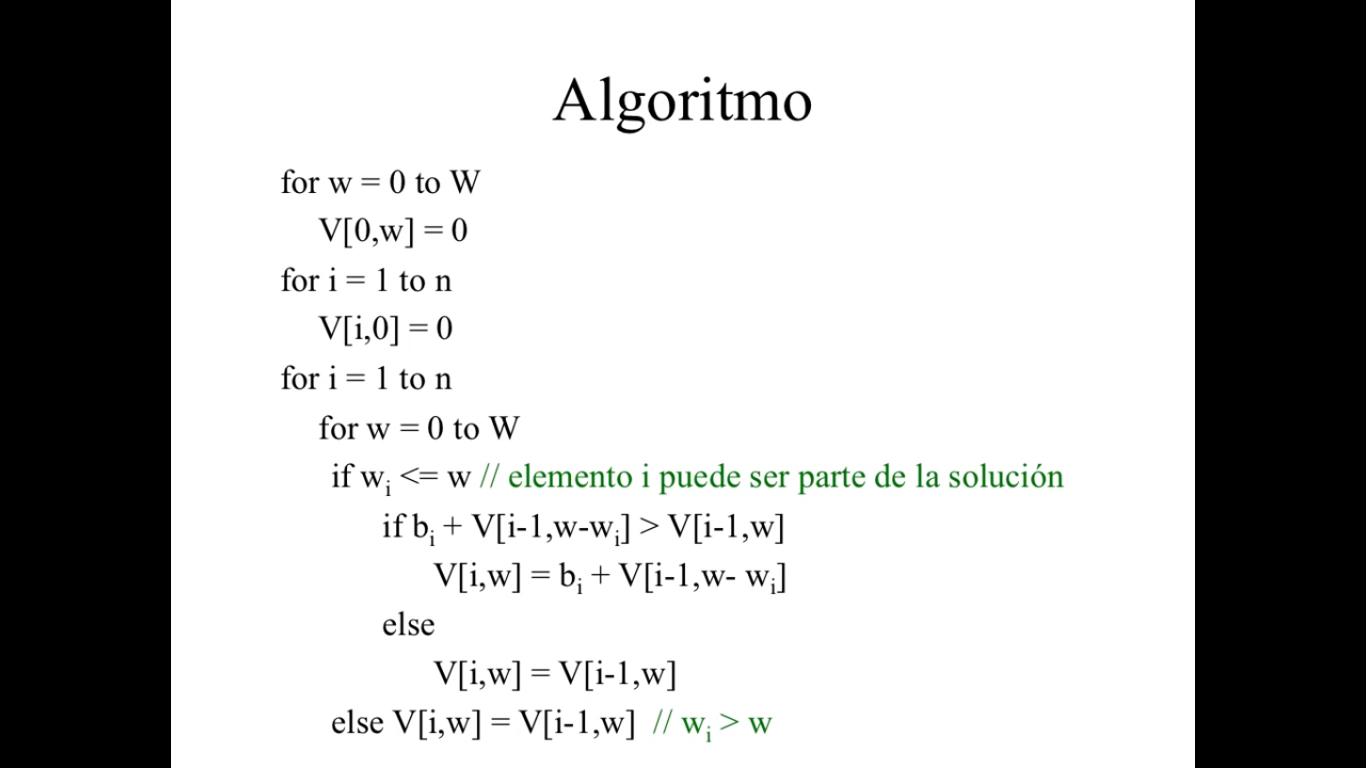
Si asumimos que conocemos **V[i,j],** donde i=0,1,2,3…k-1, j=0,1,2,3... **w**, ¿Cómo derivar **V[k,w]**?

Formula recursiva para los subproblemas:

**V[k,w] =**

1. El mejor subconjunto que tiene un peso total **<=** **W**, contiene al elemento k o no.
2. Primer caso: .El elemento k no puede ser parte de la solución, porque si lo fuera, el peso total sería **> W**, lo cual no es aceptable.
3. Segundo caso: .Entonces el elemento k puede estar en la solución, y escogemos el caso con el que maximice el valor.

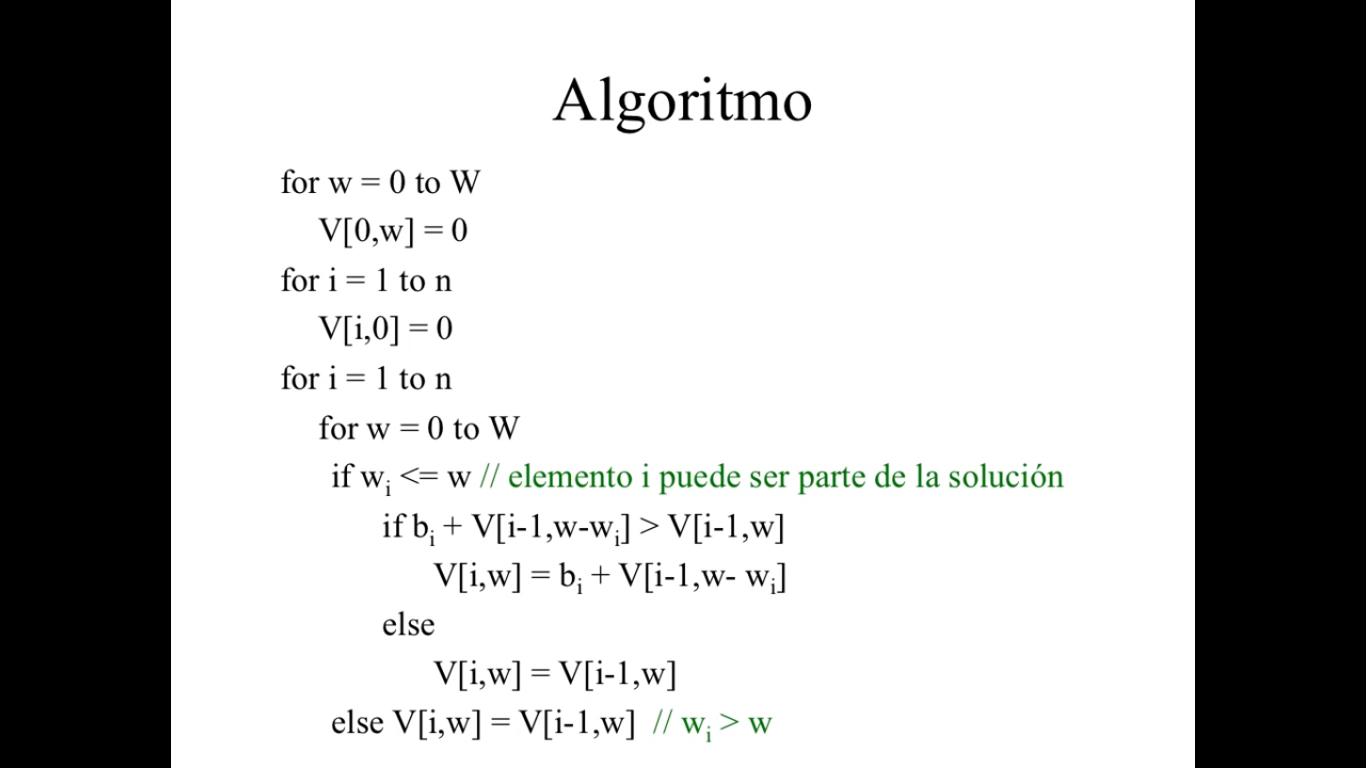
**3.3 Algoritmo Dinámico para el Problema de la Mochila**



Algorítmo dinámico

***Figura 1****:* Algoritmo del paradigma de programación dinámica.

**3.4 Complejidad para el Problema de la Mochila**



Complejidad algorítmica

O(w)

O(w)

O(n)

O(n)

O(w\*n)

O(w)

***Figura 2****:* Complejidad del algoritmo dinámico paso a paso.

Solución: O(max {w,n,w\*n}) = O(w\*n)

**Capítulo 4**

**Optimización del Algoritmo Dinámico**

**4.1 ¿Cómo optimizar nuestro algoritmo?**

Si realizamos la comparación entre la solución anteriormente planteada (Fuerza bruta) su complejidad es de ***O()*** siendo este un algoritmo de complejidad exponencial, no tan bueno para la solución que estamos buscando,pero al realizar el correcto planteamiento del paradigma de algoritmo dinámico nos damos cuenta que la complejidad ha reducido ***O(w\*n)*** que también podría expresarse como ***O()*** ya que solo es una simplemente es una multiplicación de dos variables.

Entonces hemos logrado reducir la complejidad a un tiempo polinomial (cuadrático), logrando una mayor optimización del algoritmo, esto se debe a que toda la información que vallamos sacando cada vez que actualizamos con nuevos valores nuestra tabla, algunos de ellos ya no son calculados nuevamente debido a que en el paradigma de programación dinámica se almacenan datos en una tabla, reutilizando esta información para que los cálculos futuros tomen dicha información y no realicen operaciones que ya se han realizado.

Como vemos nos convine utilizar el segundo ejemplo para un control de recursos y rapidez a la hora de resolver este problema, para ello debemos utilizar el algoritmo dinámico para este tipo de problemas. Para ello debemos tener en claro la diferencia entre problemas y algoritmos.

El problema es uno solo, pero los algoritmos son muchos, y existirán algoritmos que lo hagan mejor y otros peor, ósea esto siempre dependerá de su complejidad. Una cosa es la complejidad del algoritmo y otra cosa la complejidad del problema, y **la complejidad de un problema es la del mejor algoritmo que lo resuelva.**

Si resumimos la importancia de la complejidad en tres simples puntos, estos serían:

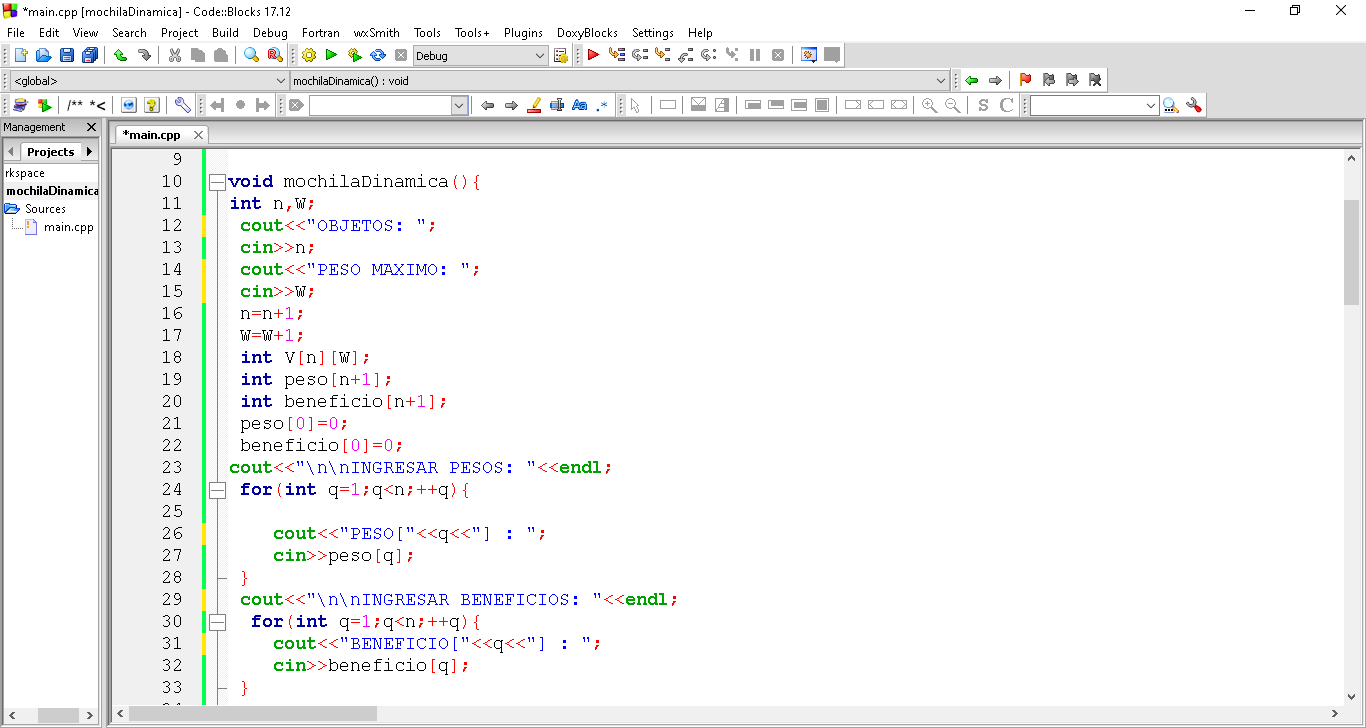
1. Podemos medir la complejidad de los problemas y de los algoritmos que los resuelvan.
2. Es una cuestión muy importante para la ciencia de la computación.
3. No es lo mismo la complejidad de un problema que la complejidad en la forma de resolverla.

Los recursos comúnmente estudiados en complejidad computacional son:

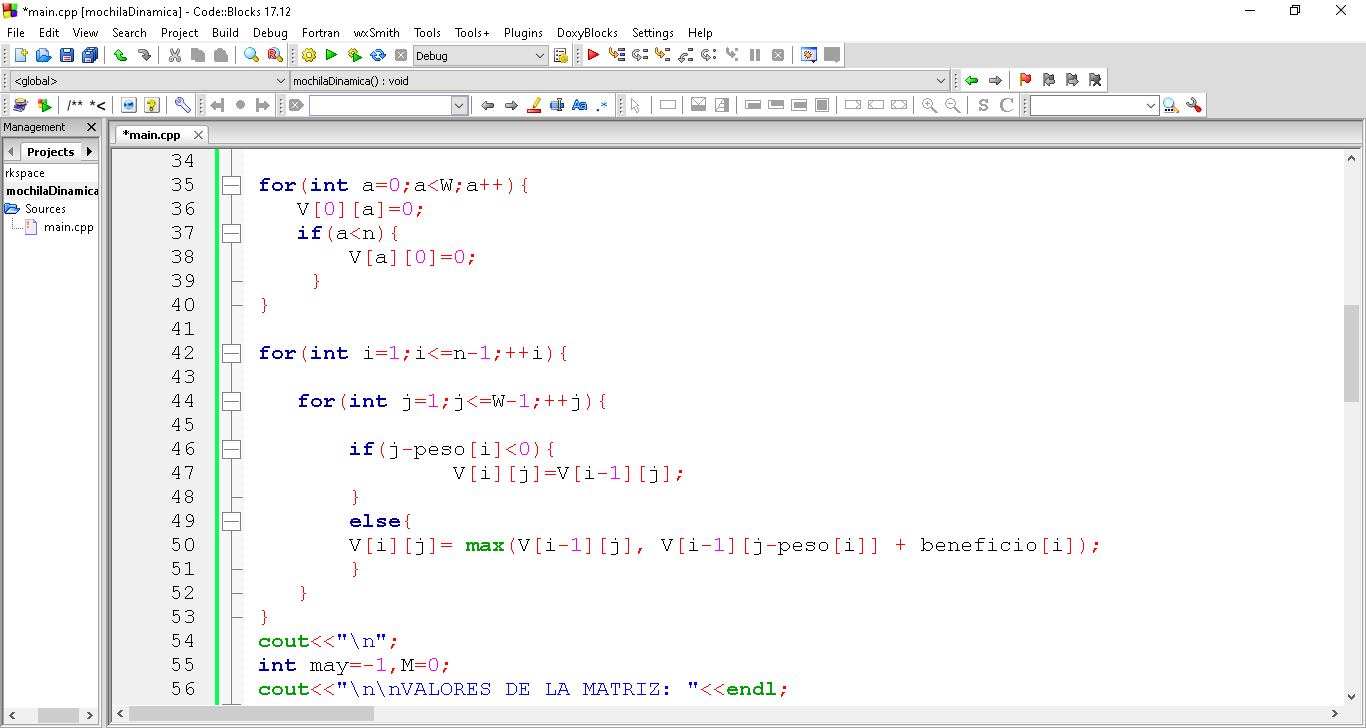
* El **tiempo**: mediante una aproximación al número de pasos de ejecución que un algoritmo emplea para resolver un problema.
* El **espacio**: mediante una aproximación a la cantidad de memoria utilizada para resolver el problema.

**4.2 Codificación del Algoritmo Dinámico**

Sabiendo todo esto es hora de utilizar un lenguaje de programación para realizar dicho problema, en este caso utilizaremos el lenguaje de programación C++, ya que se adapta mejor al algoritmo que se muestra en la ***Figura 1***, teniendo nuestro lenguaje como herramienta podemos compilar desde la terminal dependiendo del sistema operativo que usemos, pero sería recomendable utilizar un IDE para una mayor facilidad de codificación, depuración y ejecución del programa, en este caso utilizamos el IDE Codeblocks. Nuestro código quedaría de la siguiente manera:

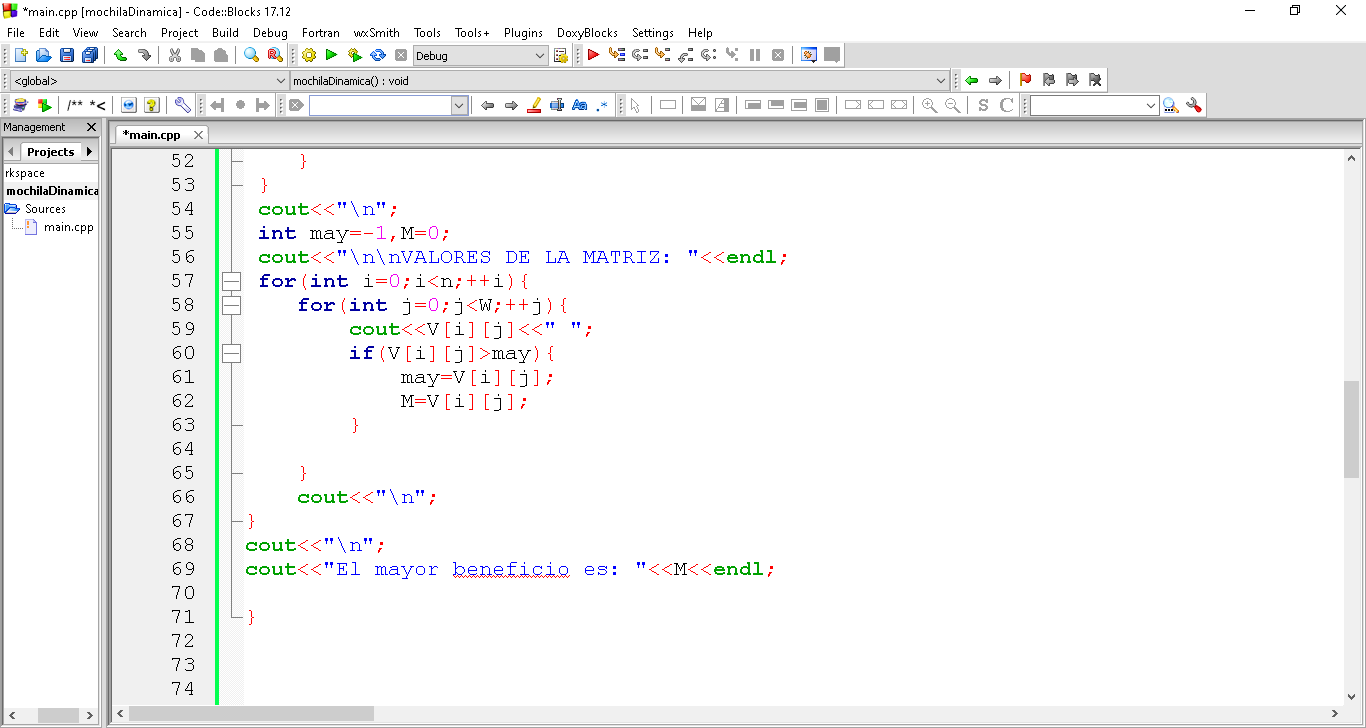


***Figura 3****:* Inicialización de nuestra tabla.



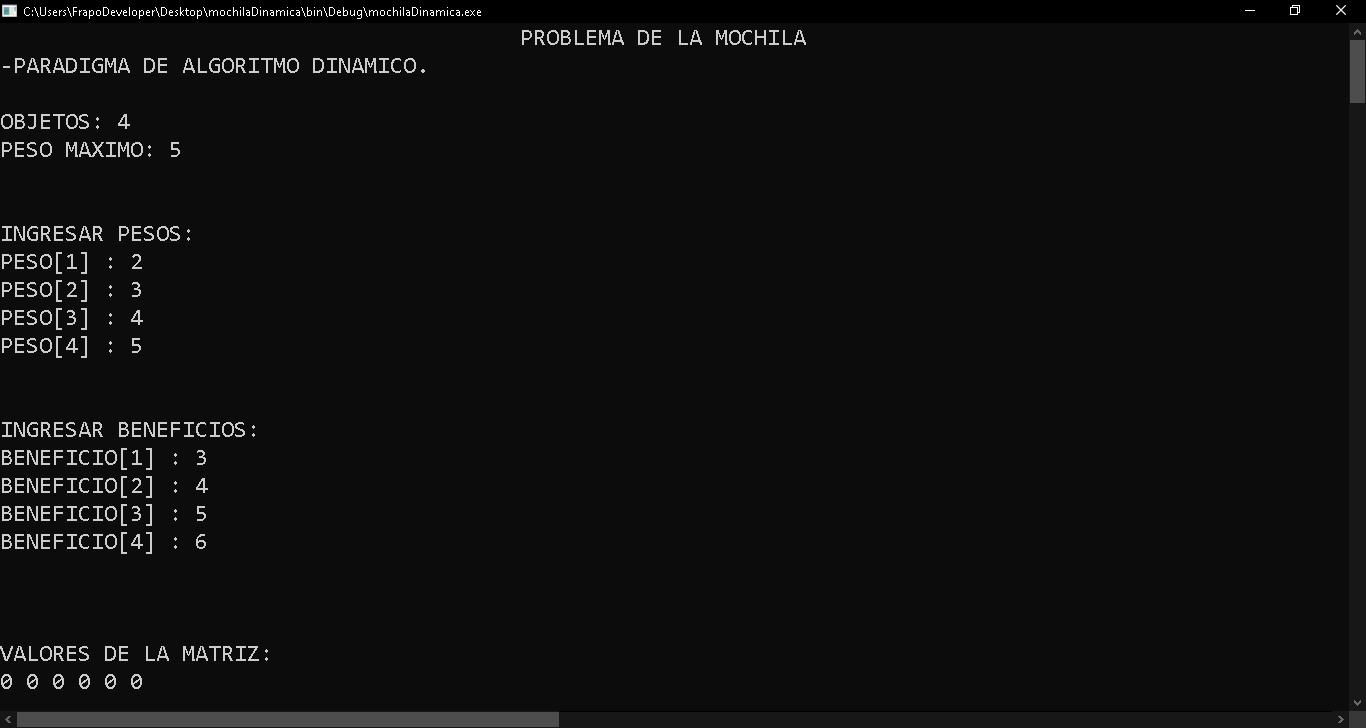
***Figura 4****:* Algoritmo recursivo del problema de la mochila.

**4.3 Solución y ejecución para el Problema de la Mochila**

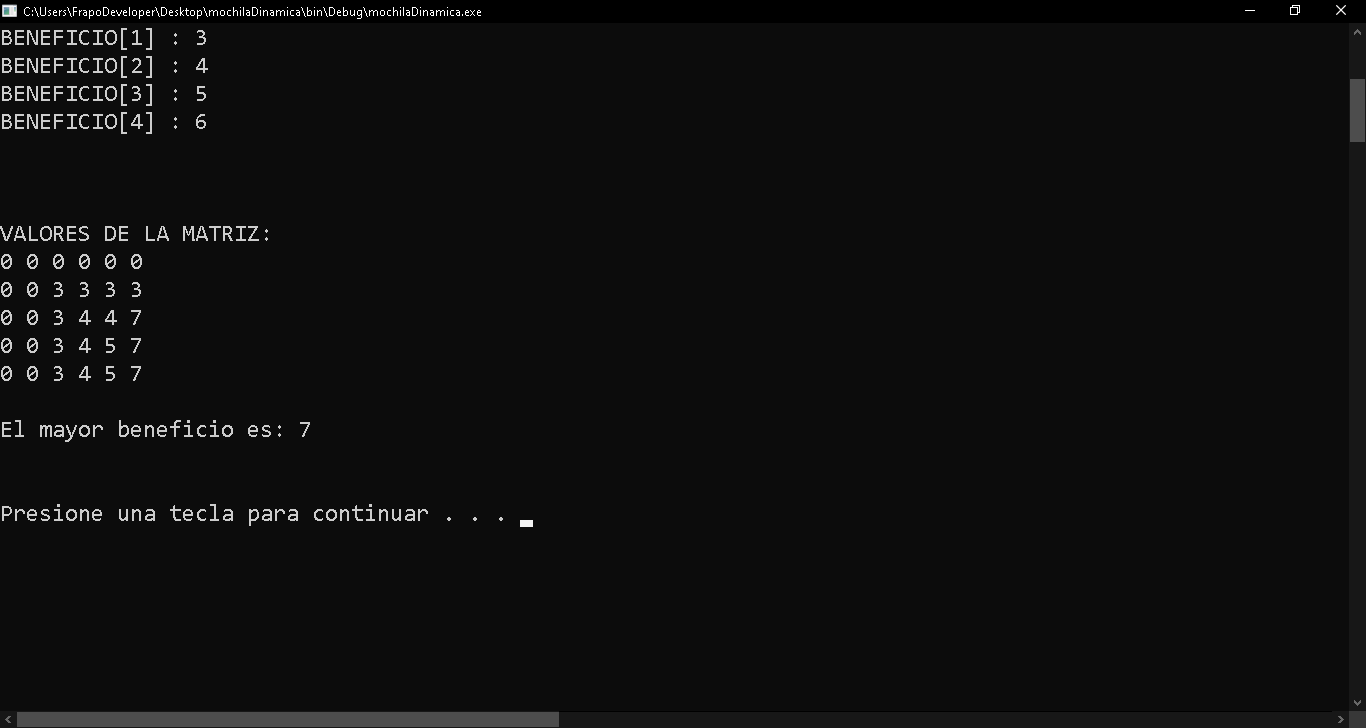


***Figura 5****:* Mostrar la tabla de todo el procedimiento realizado.

El algoritmo estaría listo, cabe resaltar que la complejidad se da respecto a la cantidad de elementos y el peso máximo que vallamos a ingresar, esto se debe a que a pesar de que la complejidad ha sido reducida a ***O(n\*m)*** *,* esto dependerá de que tan grandes puedan llegar a ser estas dos variables.



***Figura 6****:* Ejecución del programa, ingresando datos.



***Figura 7****:* Ejecución del programa, mostrando la optimización del algoritmo.

1. **Referencias**

ARNAL, J.; DEL RINCÓN, D.; LATORRE, A. (1996). Bases metodológicas de la investigación educativa. Barcelona – España. Editorial Grup92.

ROBERTO SAMPIERI & COAUTORES (1998) Metodología de la Investigación (2ª edición). México. Editorial Mc. Graw - Hill.

ANDER EGG, E. (1978). Técnicas de Investigación Social, (19ª edición), Buenos Aires – Argentina. Editorial Humanistas.

Custodio Ruiz, A. (2008, 5 agosto). Métodos y técnicas de investigación científica. Recuperado 12 octubre, 2019.

ARIAS, F.G. (1999). El Proyecto de Investigación: Guía para su elaboración. (3ª edición), Caracas – Venezuela. Editorial Episteme.

BAVARESCO DE PIETRO, A. M. (1997). Proceso Metodológico de la Investigación (Como hacer un Diseño de Investigación), Maracaibo – Venezuela. Editorial de la Universidad del Zulia.