

Universidad del Valle De Guatemala

Facultad de Ingeniería

Análisis y Diseño de Algoritmos



Proyecto 1 - Knapsack Problem

Javier Mombiela 20066

Pablo González 20362

Jose Hernández 20053

Guatemala, 27 de abril 2023

Definición y enunciado del problema

Enunciado

El problema que elegimos como grupo fue el “Knapsack Problem” o el Problema de la mochila es un problema de optimización combinatorial clásico en el que se busca determinar la combinación de objetos de una lista dada que proporciona el valor máximo total sin exceder una capacidad dada. El problema se puede formular de la siguiente manera: hay un ladrón que ha entrado a una casa y tiene una mochila que puede llevar una cantidad limitada de peso. En la casa hay varios objetos, cada uno con su propio peso y valor. El objetivo del ladrón, es elegir qué objetos llevar para maximizar el valor total de los objetos robados mientras se asegura de que el peso total de los objetos no supere la capacidad de su mochila. Para esto el ladrón tiene que ver cuál es la combinación de objetos que sumen el mayor valor y que la suma de los pesos no exceda el peso de su mochila.

Variantes

Para este problema existen dos variantes; fraccional y 0-1. La variante fraccional permite que el ladrón pueda partir objetos para que el peso disminuya y de esta manera poder meter más objetos a la mochila. La variante 0-1 no permite que se dividan los objetos, por lo cual en esta variante el ladrón puede elegir meter un objeto entero a la mochila o puede elegir no meterlo.

Este tipo de problemas se pueden resolver utilizando diferentes algoritmos, como el algoritmo de fuerza bruta, el algoritmo voraz, el algoritmo de divide and conquer, o el algoritmo de programación dinámica. Cada uno de estos algoritmos tiene sus propias ventajas y desventajas, y la elección del algoritmo adecuado depende del tamaño y complejidad del problema.

Además, el problema de la mochila puede ser extendido a situaciones en las que los objetos tienen restricciones adicionales, como límites de cantidad o de espacio. En estos casos, se pueden utilizar técnicas avanzadas como la programación lineal o la optimización combinatoria.

Por lo tanto, podemos concluir que, el problema de la mochila es un ejemplo común de un problema de optimización que se puede resolver utilizando diferentes algoritmos y técnicas avanzadas. Para este proyecto, estaremos utilizando la variante 0-1, y también estaremos tomando en cuenta que un objeto no se puede meter más de una vez en la mochila. Estaremos utilizando la variante de 0-1, ya que esta variante aplica más a los algoritmos divide and conquer y programación dinámica. La variable fraccional aplica más a un algoritmo greedy, por lo que no nos es de mucha utilidad para este proyecto.

Ejemplo



Para poder resolver el problema de la mochila, podemos identificar las siguientes variables:

- La capacidad de la mochila, en este caso, sería de 15 kg.
- La cantidad de objetos, que en este caso serían 5 objetos.
- Los valores y los pesos de cada objeto. Estos los podemos representar como un arreglo de números enteros.
 - Valor = [4, 2, 10, 2, 1]
 - Peso = [12, 1, 4, 2, 1]

Ya que se definen todas las variables del problema, se puede proceder a resolverlo. Para este ejemplo, podemos ver que el valor máximo posible con los objetos dados sería de \$15. ¿Pero cómo se llegó a esta respuesta? Bueno, lo que se hizo fue encontrar la combinación de objetos que nos dieron el mayor valor sin excedernos de 15 kg. Por lo tanto, de primero se excluye el primer objeto, ya que este es el que más peso tenía, pero su valor no era tan alto, por lo que la combinación con mayor valor posible con este objeto sería de \$8.

Entonces pasamos al objeto con mayor valor, que es el tercer objeto, y vemos que su peso no es tan alto, por lo cual lo podríamos combinar con varios objetos. De hecho, lo podemos meter en la mochila junto con todos los otros objetos (excluyendo el primero). Al hacer la suma del valor esto nos da un valor de \$15 y un peso total de 8 kg. Podemos ver que todavía tenemos 7 kg libres, pero solo nos queda un objeto, que sería el primero y este pesa más de 7 kg, y como estamos usando la variante 0-1, esta no nos permite solo agarrar una fracción del objeto, por lo que lo tenemos que dejar el objeto, y con esto resolvemos el ejemplo.

Explicación de algoritmos

Algoritmo DaC

Código y explicación

```
1 def knapSack(W, wt, val, n):
2
3     # Base Case
4     if n == 0 or W == 0:
5         return 0
6
7     # If weight of the nth item is
8     # more than Knapsack of capacity W,
9     # then this item cannot be included
10    # in the optimal solution
11    if (wt[n-1] > W):
12
13        return knapSack(W, wt, val, n-1)
14
15    # return the maximum of two cases:
16    # (1) nth item included
17    # (2) not included
18    else:
19        return max(
20            val[n-1] + knapSack(
21                W-wt[n-1], wt, val, n-1),
22            knapSack(W, wt, val, n-1))
```

[GeeksforGeeks](#). (24 de febrero, 2023).

1. Def KnapSack(W,wt,val,n): Esta es la definición de la función que se puede mencionar que toma 4 parametros los parametros son W que es la capacidad de la mochila, una lista de pesos wt, una lista de valores correspondientes que se define como val y por último la cantidad de elementos que es n.
2. “If n == 0 or W ==0”: Este if se hace con la intención de que si no quedan elementos o la capacidad de la mochila es de 0, se retorna 0 ya que es imposible añadir algún elemento.
3. “if (wt[n-1] > W):”: Si el peso del último elemento es mayor que la capacidad de la mochila, este elemento se puede mencionar que no se puede añadir, por lo tanto se llama a la función de forma recursiva con elemento n-1.

4. “return knapSack(W, wt, val, n-1)”: Se retorna el resultado de la llamada recursiva para el caso donde el elemento no está incluido.
5. “Else”: Si el peso del elemento se puede incluir en la mochila se consideran dos casos: ‘val[n-1] + KnapSack(W-wt[n-1],wt,val,n-1)’ El valor del elemento se añade al valor total y se llama a la función KnapSack recursivamente con la capacidad de la mochila actualizada para excluir el elemento y la lista elementos actualizada sin el último elemento.
6. “knapSack(W, wt, val, n-1)”: Se llama a la función recursivamente sin incluir el último elemento.
7. “return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1), knapSack(W, wt, val, n-1))”: Se devuelve el valor máximo de los casos planteados con anterioridad.

Justificación

En el caso de Knapsack, se utiliza la técnica divide and conquer a la hora que se divide el problema en dos partes: incluir el n -ésimo elemento o no incluirlo. Luego, se resuelve cada subproblema por separado y se elige la mejor opción. Este enfoque es adecuado para problemas grandes, ya que permite una reducción significativa en el tiempo de ejecución. Sin embargo, tiene un alto costo de memoria debido a la necesidad de almacenar las soluciones intermedias.

Divide: El problema se divide en subproblemas recursivamente, considerando si incluir o no el n -ésimo elemento en la mochila. La función knapSack se llama recursivamente con $n-1$ como argumento, lo que significa que estamos dividiendo el problema en un subconjunto de elementos más pequeños.

Conquer: La función knapSack se encarga de resolver los subproblemas en su nivel de recursión. Se analiza si el n -ésimo elemento puede ser incluido en la mochila comparando su peso ($wt[n-1]$) con el peso restante (W). Si el peso del n -ésimo elemento es mayor que el peso restante, no puede ser incluido, por lo que se resuelve el subproblema sin incluirlo. En caso contrario, se exploran dos opciones: incluir el elemento y no incluirlo.

Combine: Las soluciones a los subproblemas se combinan utilizando la función max, que toma el máximo entre dos valores: el valor del n -ésimo elemento sumado al resultado de la llamada recursiva que incluye el n -ésimo elemento, y el resultado de la llamada recursiva que no lo incluye.

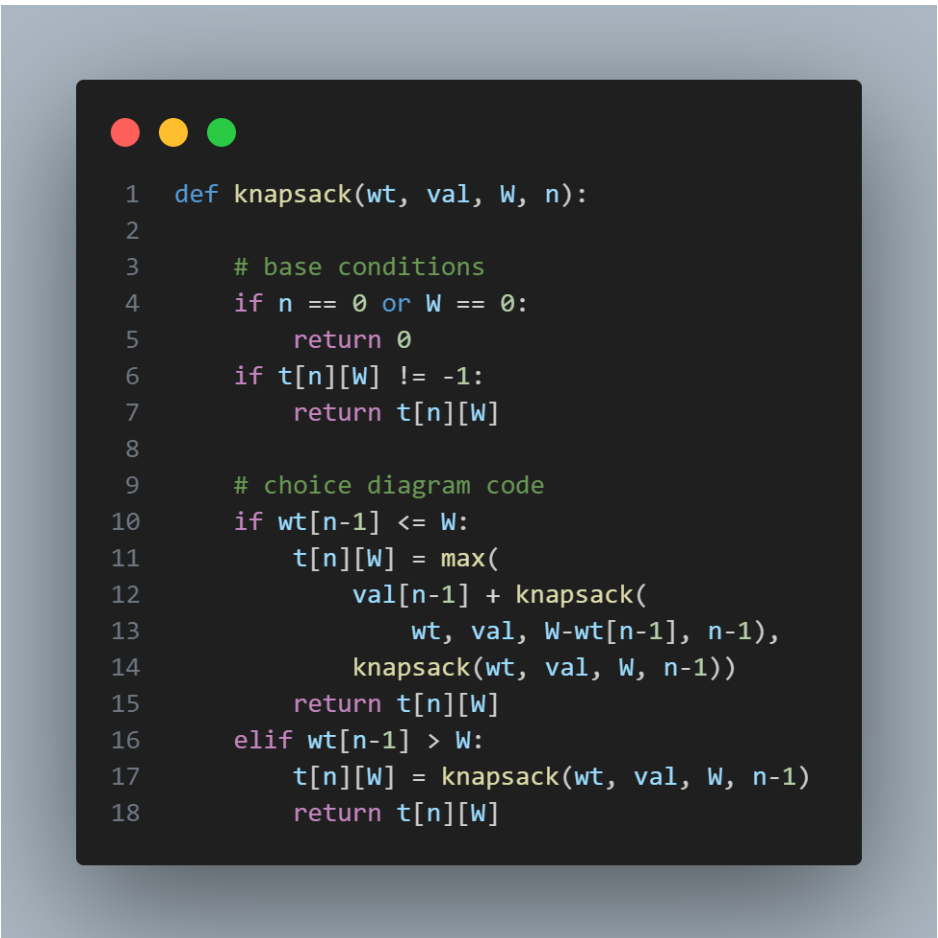
Aplicabilidad de "Knapsack Problem" utilizando divide and conquer:

Una posible aplicación de la técnica de divide y conquista para resolver el problema de la mochila es en la optimización de carteras de inversión. En este caso, se busca maximizar el retorno de una cartera de inversión, donde cada activo tiene un retorno esperado y un riesgo

asociado. El problema se puede formular como una instancia de "Knapsack Problem", donde los activos son los objetos a elegir, y el retorno esperado y el riesgo son el valor y el peso de cada objeto, respectivamente. El objetivo es encontrar la combinación óptima de activos que maximice el retorno esperado, sujeto a una restricción de riesgo total que no puede ser excedida. La técnica de divide y conquista se puede aplicar para buscar soluciones óptimas para portafolios de inversión de gran tamaño, al dividir el problema en subproblemas más pequeños y luego combinar las soluciones.

Algoritmo DP

Código y explicación



```
1  def knapsack(wt, val, W, n):
2
3      # base conditions
4      if n == 0 or W == 0:
5          return 0
6      if t[n][W] != -1:
7          return t[n][W]
8
9      # choice diagram code
10     if wt[n-1] <= W:
11         t[n][W] = max(
12             val[n-1] + knapsack(
13                 wt, val, W-wt[n-1], n-1),
14             knapsack(wt, val, W, n-1))
15         return t[n][W]
16     elif wt[n-1] > W:
17         t[n][W] = knapsack(wt, val, W, n-1)
18         return t[n][W]
```

[GeeksforGeeks](#). (24 de febrero, 2023).

1. “def knapsack(wt, val, W, n)” : Esta es la definición de la función que se puede mencionar que toma 4 parametros los parametros son W que es la capacidad de la mochila, una lista de pesos wt, una lista de valores correspondientes que se define como val y por último la cantidad de elementos que es n.

2. “if $n == 0$ or $W == 0$:

 return 0

if $t[n][W] \neq -1$:

 return $t[n][W]$ ”

En estas dos líneas de código se establecen las condiciones de la recursión siendo en donde si N es cero o W es cero, entonces la función devolverá como valor 0 ya que no hay elementos que agregar o no hay capacidad en la mochila. Además también se define la condición $\text{if } t[n][W] \neq -1$ qué es que si el valor ya ha sido calculado y almacenado en la tabla de memoización, la función simplemente devuelve ese valor en lugar de volver a calcularlo.

3. “if $wt[n-1] \leq W$:

$t[n][W] = \max($

$val[n-1] + \text{knapsack}($

$wt, val, W - wt[n-1], n-1),$

$\text{knapsack}(wt, val, W, n-1))$

 return $t[n][W]$

elif $wt[n-1] > W$:

$t[n][W] = \text{knapsack}(wt, val, W, n-1)$

 return $t[n][W]$

”

En estas líneas se puede mencionar que se implementa el diagrama de decisión del problema de la mochila en donde si el peso del elemento actual definido por $wt[n-1]$ es menor a la capacidad disponible W , entonces se tiene la opción de agregar el elemento a la mochila por medio de la línea “ $val[n-1] + \text{knapsack}(wt, val, W - wt[n-1], n-1)$ ” o no agregarlo que se define por medio de la línea “ $\text{knapsack}(wt, val, W, n-1)$ ” y se toma la opción de generar un valor máximo de esta función. Si el peso del elemento actual es mayor que la capacidad disponible, entonces no se puede agregar el elemento a la mochila, por lo que se pasa al siguiente elemento ($\text{knapsack}(wt, val, W, n-1)$). En ambos casos, se almacena el valor obtenido en la tabla de memorización para tener este como referencia y devolver dicho valor.

Justificación

En el caso de Knapsack, se utiliza la técnica de programación dinámica a la hora de utilizar una matriz (t) para almacenar las soluciones de subproblemas ya resueltos. Cada entrada de la matriz se inicializa con un valor negativo para indicar que aún no se ha calculado. Cuando se necesita el resultado de un subproblema, se consulta la matriz. Si el valor ya está presente, se devuelve; de lo contrario, se resuelve el subproblema y se almacena en la matriz para su reutilización posterior. Este enfoque tiene una complejidad temporal y espacial óptima, lo que lo hace adecuado para problemas de tamaño moderado.

Memoización: La tabla t se utiliza para almacenar los resultados de los subproblemas que ya han sido resueltos. La función `knapsack` primero verifica si el resultado para el subproblema

actual (representado por n y W) ya ha sido calculado previamente y almacenado en $t[n][W]$. Si es así, se devuelve el resultado almacenado en lugar de calcularlo de nuevo.

Subestructura óptima: El problema de la mochila 0-1 presenta una subestructura óptima, lo que significa que la solución óptima para un problema dado se puede construir a partir de las soluciones óptimas de sus subproblemas. En este algoritmo, se toma la decisión de incluir o no incluir el n -ésimo elemento en la mochila de la misma manera que en el algoritmo de divide y vencerás, pero se aprovecha la tabla t para almacenar y reutilizar los resultados de los subproblemas.

Solución de subproblemas superpuestos: La programación dinámica es especialmente útil para resolver problemas en los que los subproblemas se solapan, es decir, se repiten varias veces con los mismos argumentos. En el problema de la mochila 0-1, los subproblemas con los mismos valores de n y W se resuelven múltiples veces en el algoritmo de divide y vencerás, pero en este enfoque de programación dinámica, esos resultados se almacenan en la tabla t y se reutilizan para evitar cálculos redundantes.

Aplicabilidad de "Knapsack Problem" utilizando programación dinámica:

Otra posible aplicación de la técnica de programación dinámica para resolver el problema de la mochila es en la planificación de proyectos. En este caso, se busca maximizar la eficiencia de un proyecto, donde cada tarea tiene un costo y una duración asociados. El problema se puede formular como una instancia de "Knapsack Problem", donde las tareas son los objetos a elegir, y el costo y la duración son el valor y el peso de cada objeto, respectivamente. El objetivo es encontrar la combinación óptima de tareas que maximice la eficiencia total del proyecto, sujeto a una restricción de duración total que no puede ser excedida. La técnica de programación dinámica se puede aplicar para buscar soluciones óptimas para proyectos de gran tamaño, al construir una tabla de valores óptimos para subconjuntos cada vez mayores de tareas y utilizar esta información para determinar la solución óptima para el proyecto completo.

Analisis Teorico

Monday, April 24, 2023 3:02 PM

```
1 def knapSack(W, wt, val, n):
2
3     # Base Case  $O(1)$ 
4     if n == 0 or W == 0:
5         return 0
6
7     # If weight of the nth item is
8     # more than Knapsack of capacity W,
9     # then this item cannot be included
10    # in the optimal solution
11    if (wt[n-1] > W):  $O(1)$ 
12
13        return knapSack(W, wt, val, n-1)  $O(T-1)$ 
14
15    # return the maximum of two cases:
16    # (1) nth item included
17    # (2) not included
18    else:
19        return max(
20            val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
21            knapSack(W, wt, val, n-1))
22     $T(n-1)$ 
```

La relación de recurrencia sería:

$$T(n) = 2T(n-1) + O(1)$$

Para calcular la tasa de crecimiento utilizando sustitución:

Caso base $n=1$:

$$T(1) \leq C \cdot 2^1 = 2C$$

Supongamos que $T(k) \leq C \cdot 2^k$ para todo $k < n$

Entonces, para $n \geq 1$:

$$T(n) = 2T(n-1) + O(1)$$

$$T(n) \leq 2(C \cdot 2^{n-1}) + O(1)$$

$$T(n) \leq C \cdot 2^n + O(1)$$

$$T(n) \leq C \cdot 2^n$$

Si elegimos un C' lo suficientemente grande para que $O(1)$ sea $\leq C' \cdot 2^n$

\therefore demostramos la inducción para todo $n \geq 1$.

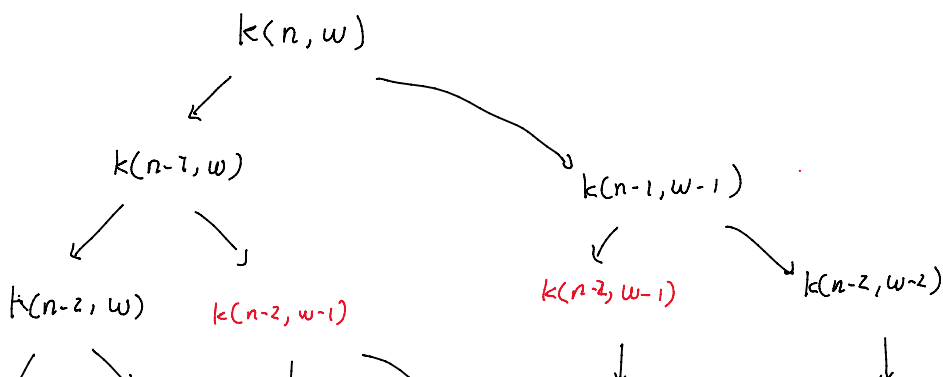
Tiempo de ejecución: $O(2^n)$

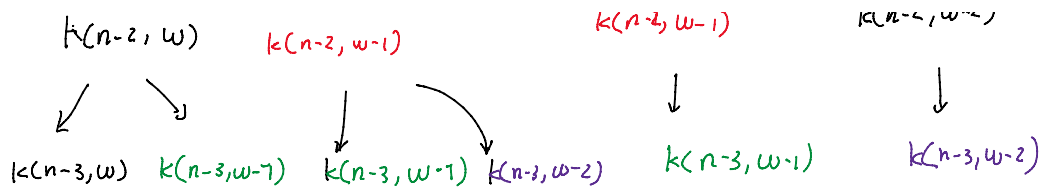
```

1  def knapsack(wt, val, W, n):
2
3      # base conditions
4      if n == 0 or W == 0:
5          return 0
6      if t[n][W] != -1:
7          return t[n][W]
8
9      # choice diagram code
10     if wt[n-1] <= W:
11         t[n][W] = max(
12             val[n-1] + knapsack(
13                 wt, val, W-wt[n-1], n-1),
14             knapsack(wt, val, W, n-1))
15         return t[n][W]
16     elif wt[n-1] > W:
17         t[n][W] = knapsack(wt, val, W, n-1)
18         return t[n][W]

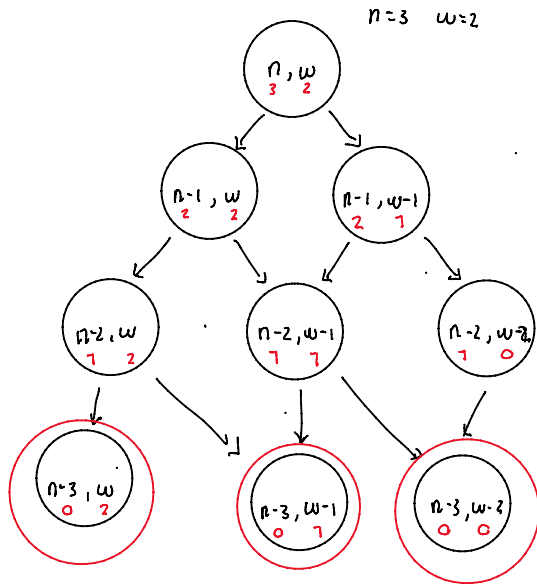
```

- Árbol de recursión





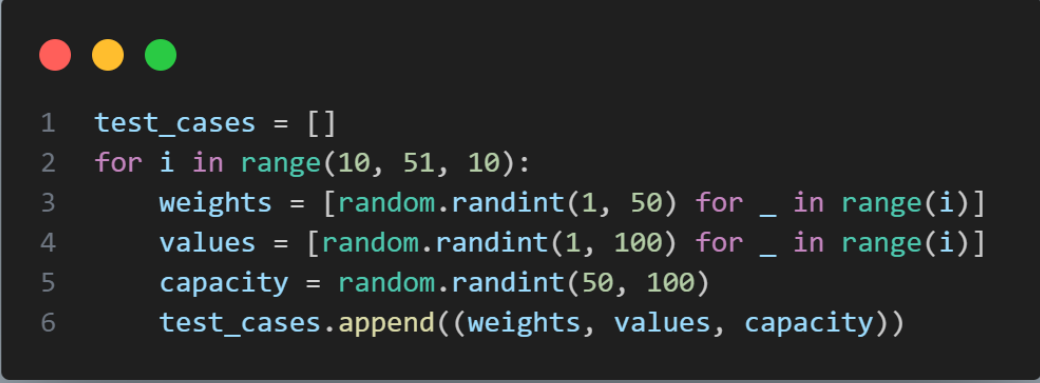
Aplicando memorización nos queda el siguiente grafo de subproblemas



Análisis Empírico

Listado de entradas de prueba utilizadas

El primer bucle for se utiliza para generar las entradas de prueba:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains a Python code snippet for generating test cases. The code is as follows:

```
1 test_cases = []
2 for i in range(10, 51, 10):
3     weights = [random.randint(1, 50) for _ in range(i)]
4     values = [random.randint(1, 100) for _ in range(i)]
5     capacity = random.randint(50, 100)
6     test_cases.append((weights, values, capacity))
```

- Este bucle for itera sobre un rango de números desde 10 hasta 50 (sin incluir 51) en incrementos de 10. Estas son las 'n' que serán utilizadas en ambos algoritmos; es decir, ambos algoritmos van a correr varias veces con los valores de $n = [10, 20, 30, 40, 50]$.
- En cada iteración, se generan los pesos y valores de cada uno de los objetos, la cantidad de objetos depende del valor de n que se está iterando.
- También se genera un valor aleatorio para la capacidad del problema de la mochila en el rango de 50 a 100. Este es el peso máximo que se puede meter a la mochila.
- Luego, se crea una tupla con las listas de pesos, valores y la capacidad, y se agrega a la lista `test_cases`. Al final del bucle, `test_cases` contendrá varias tuplas con diferentes conjuntos de pesos, valores y capacidades que se utilizarán para probar los algoritmos. Cabe mencionar que ambos algoritmos siempre tendrán los mismos parámetros.

El segundo bucle for se utiliza para medir los tiempos de ejecución de cada algoritmo en función de las entradas de prueba:

```
1  times_dc = []
2  times_dp = []
3
4  for wt, val, W in test_cases:
5      n = len(wt)
6
7      # Tiempo de ejecución de Divide and Conquer
8      start_time = time.time()
9      knapSack(W, wt, val, n)
10     end_time = time.time()
11     times_dc.append(end_time - start_time)
12
13     # Tiempo de ejecución de Programación Dinámica
14     t = [[-1 for _ in range(W + 1)] for _ in range(n + 1)]
15     start_time = time.time()
16     knapsack(wt, val, W, n, t)
17     end_time = time.time()
18     times_dp.append(end_time - start_time)
```

- Este bucle for itera sobre las tuplas en la lista test_cases que contiene las entradas de prueba generadas previamente.
- En cada iteración, se extraen los pesos (wt), los valores (val) y la capacidad (W) de la tupla actual.
- Luego, se calcula la longitud de la lista de pesos (n) que representa la cantidad de elementos en el problema de la mochila.
- Se mide el tiempo de ejecución del algoritmo Divide and Conquer utilizando la función knapSack. Se guarda el tiempo de inicio (start_time) antes de ejecutar la función y el tiempo de finalización (end_time) después de ejecutarla. La diferencia entre estos dos tiempos es el tiempo de ejecución, que se agrega a la lista times_dc.
- Se hace lo mismo para medir el tiempo de ejecución del algoritmo de Programación Dinámica utilizando la función knapsack. Antes de ejecutar esta función, se crea una matriz t de tamaño (n+1) x (W+1) inicializada con -1, que se utiliza para almacenar los resultados intermedios del algoritmo (memorización). Luego, se guarda el tiempo

de inicio (`start_time`) antes de ejecutar la función y el tiempo de finalización (`end_time`) después de ejecutarla. La diferencia entre estos dos tiempos es el tiempo de ejecución, que se agrega a la lista `times_dp`.

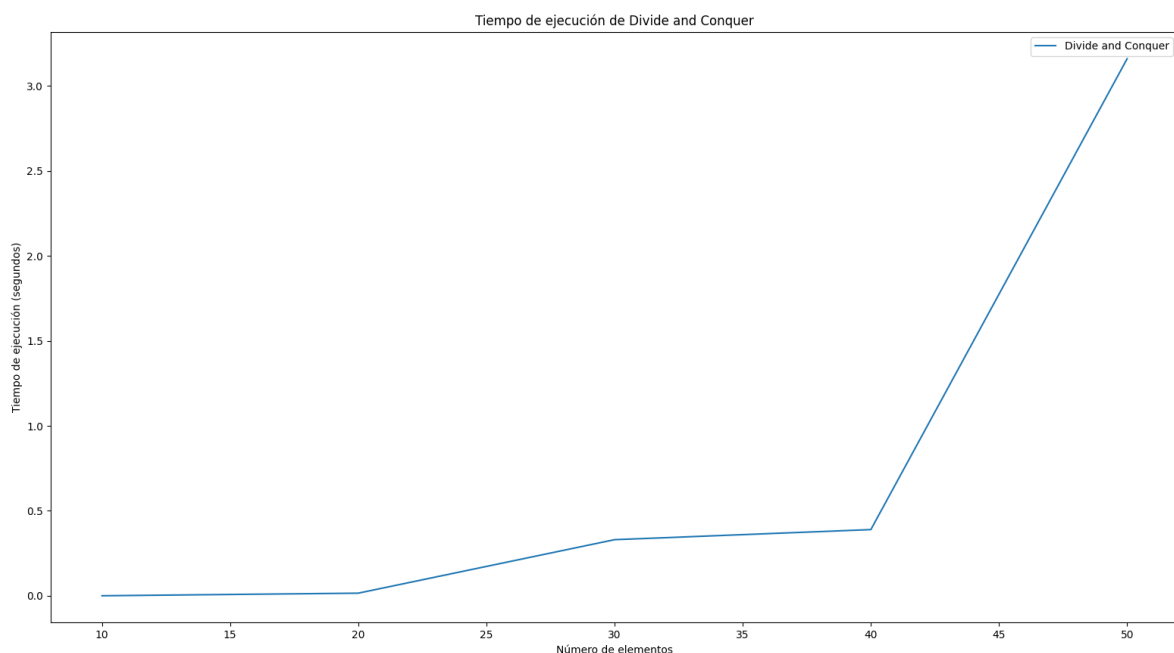
Después de completar el segundo bucle `for`, las listas `times_dc` y `times_dp` contendrán los tiempos de ejecución de los algoritmos Divide and Conquer y Programación Dinámica, respectivamente, para cada conjunto de entradas de prueba. Estos tiempos de ejecución se utilizan para trazar las gráficas de rendimiento de cada algoritmo y la gráfica combinada que compara ambos algoritmos.

En resumen, el listado de entradas de prueba se basa en la cantidad de objetos por cada iteración, y corremos cada algoritmo 5 veces, con $n = 10, 20, 30, 40, 50$. Los valores y los pesos de cada objeto en cada iteración son valores generados aleatoriamente, pero serán los mismos valores para ambos algoritmos en cada iteración.

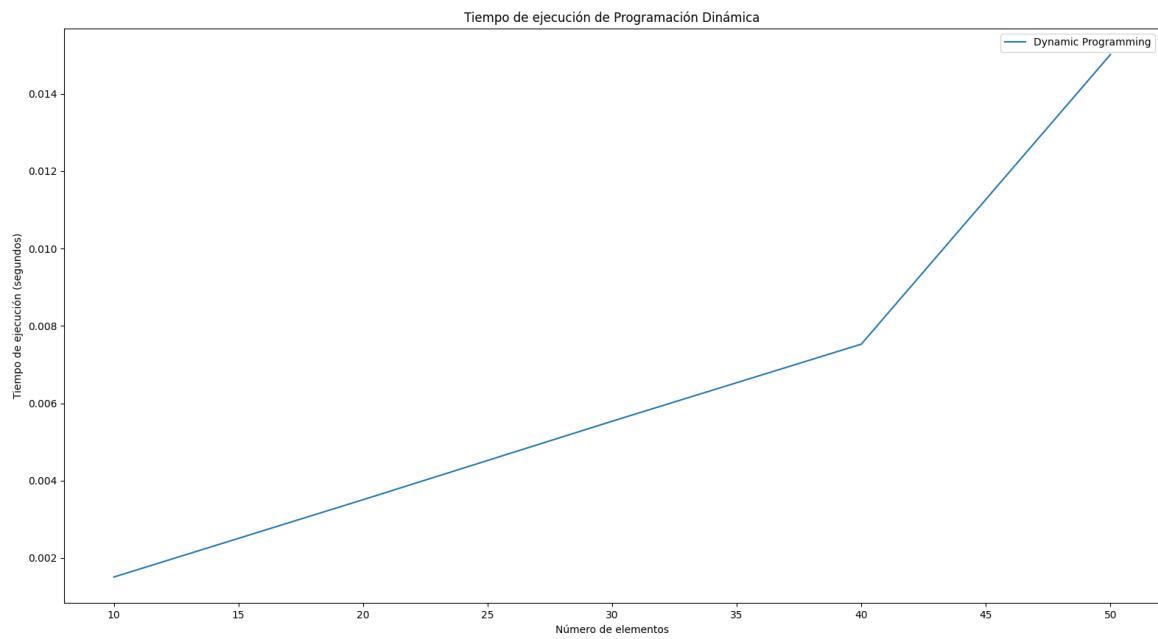
A continuación, mostramos las gráficas de ambos algoritmos a la hora de correr el programa tres diferentes veces; para poder identificar los comportamientos de cada uno de los algoritmos con los datos de prueba que se especificaron en el programa. Se harán tres diferentes iteraciones para poder comprobar que los comportamientos de cada algoritmo sean el mismo o similar en cada iteración.

Primera Iteración

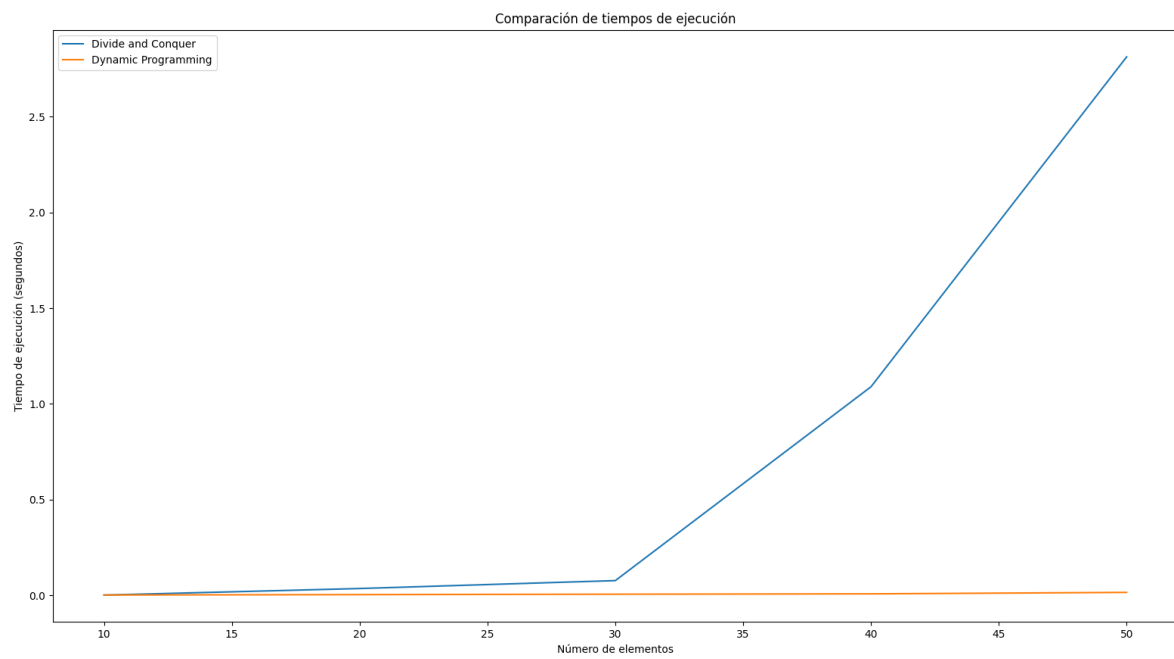
Gráfica 1: Tiempo de ejecución de DaC iteración 1



Gráfica 2: Tiempo de ejecución de DP iteración 1

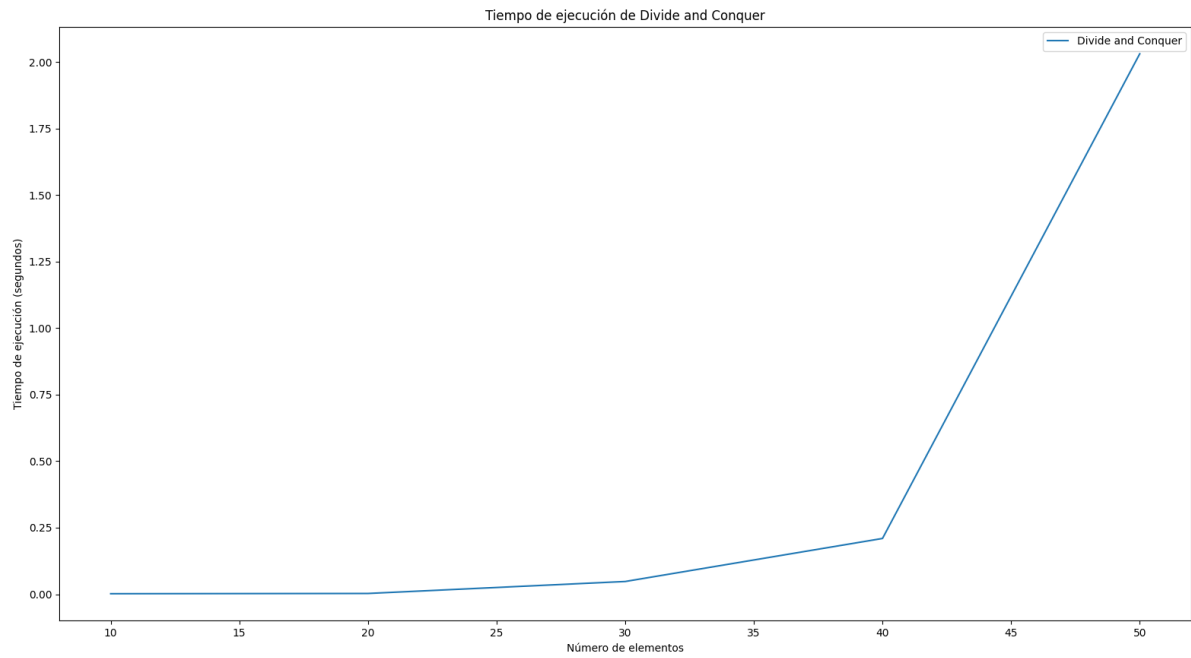


Gráfica 3: Comparación de tiempos de ejecución iteración 1

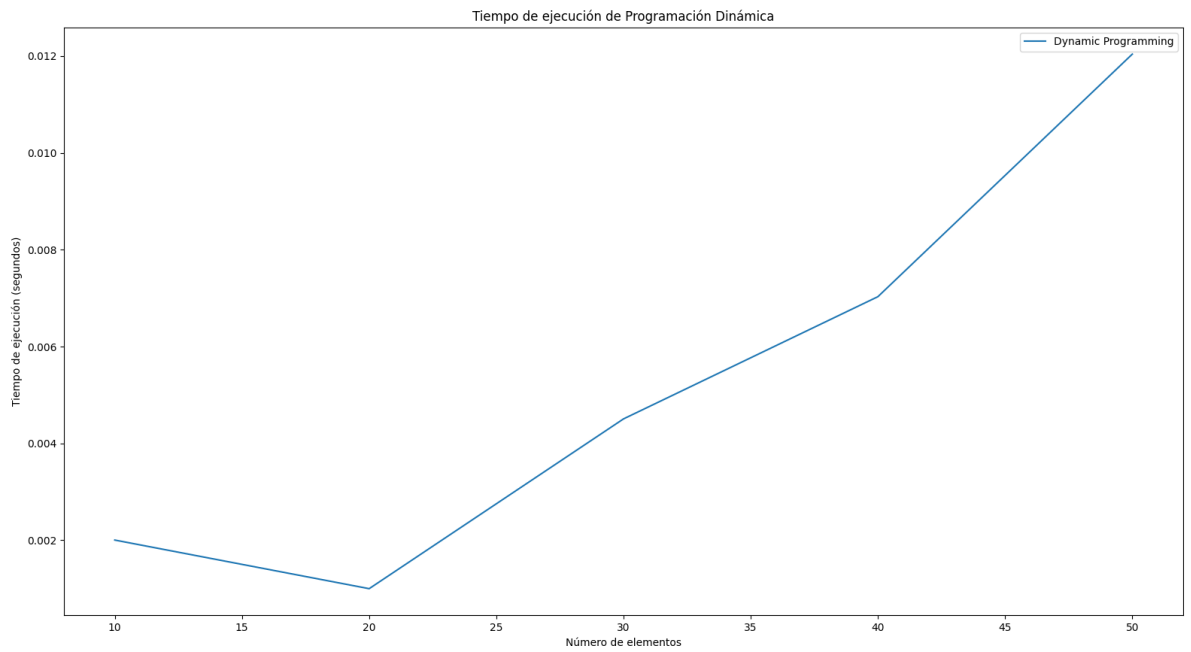


Segunda Iteración

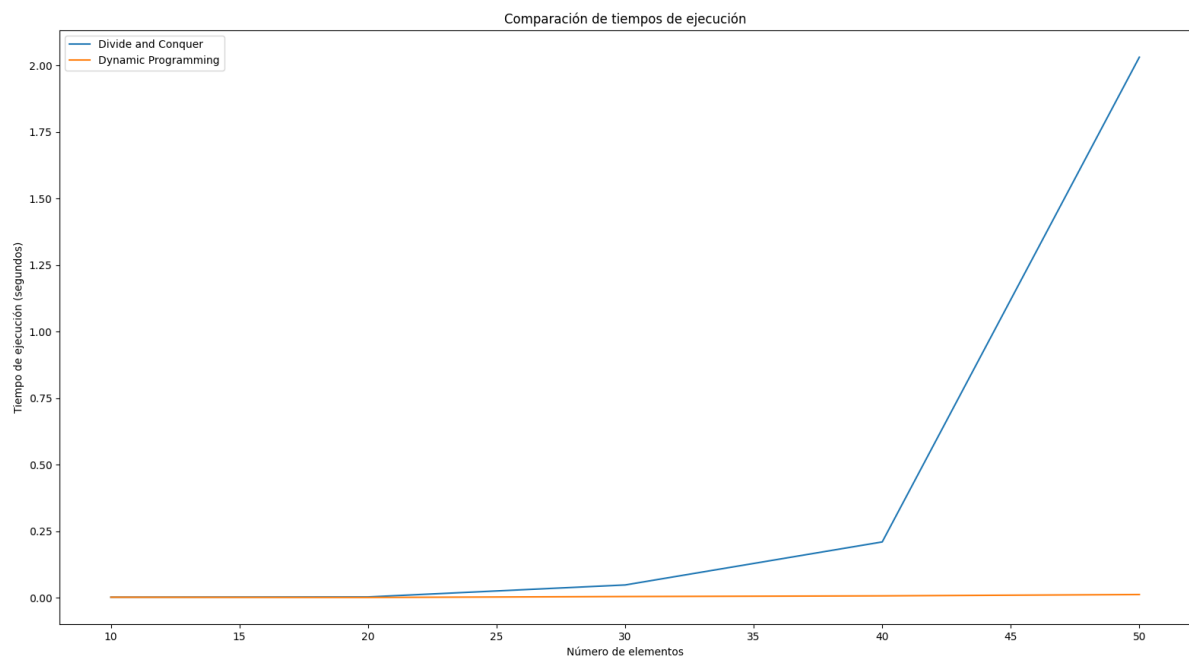
Gráfica 4: Tiempo de ejecución de DaC iteración 2



Gráfica 5: Tiempo de ejecución de DP iteración 2

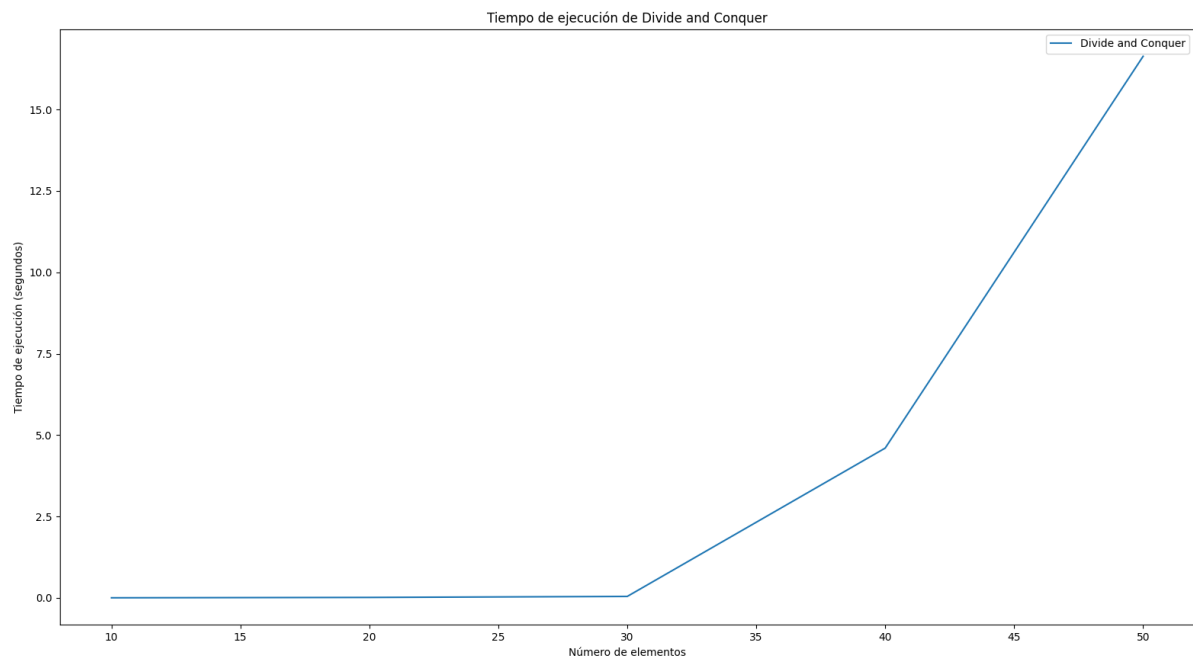


Gráfica 6: Comparación de tiempos de ejecución iteración 2

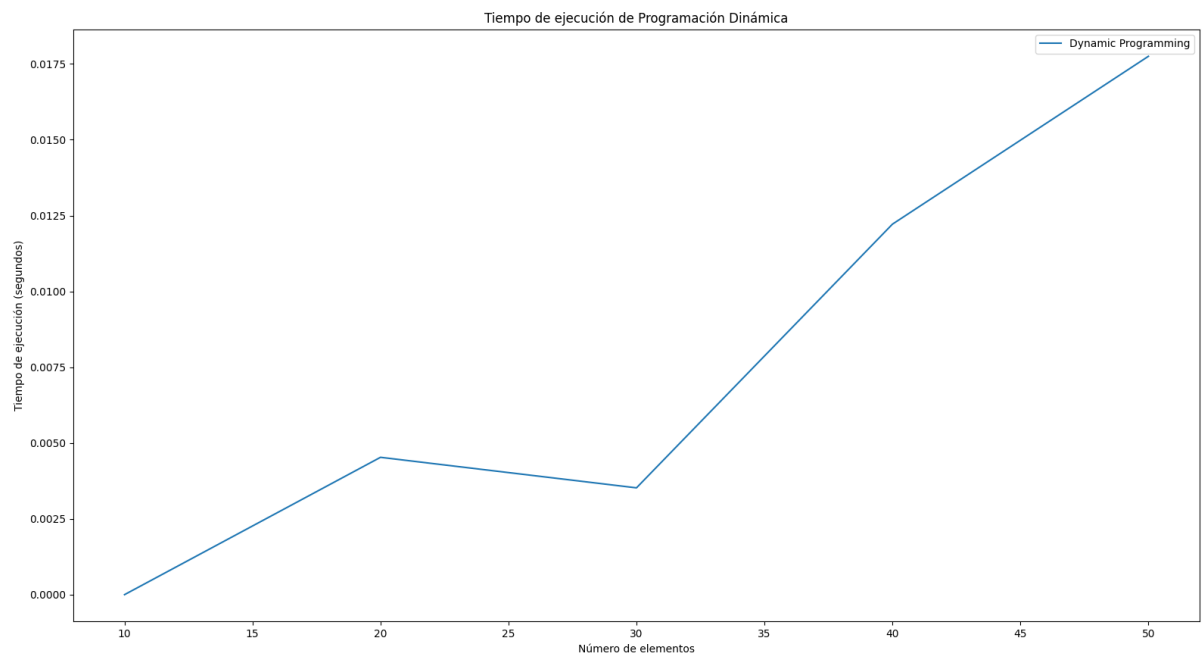


Tercera Iteración

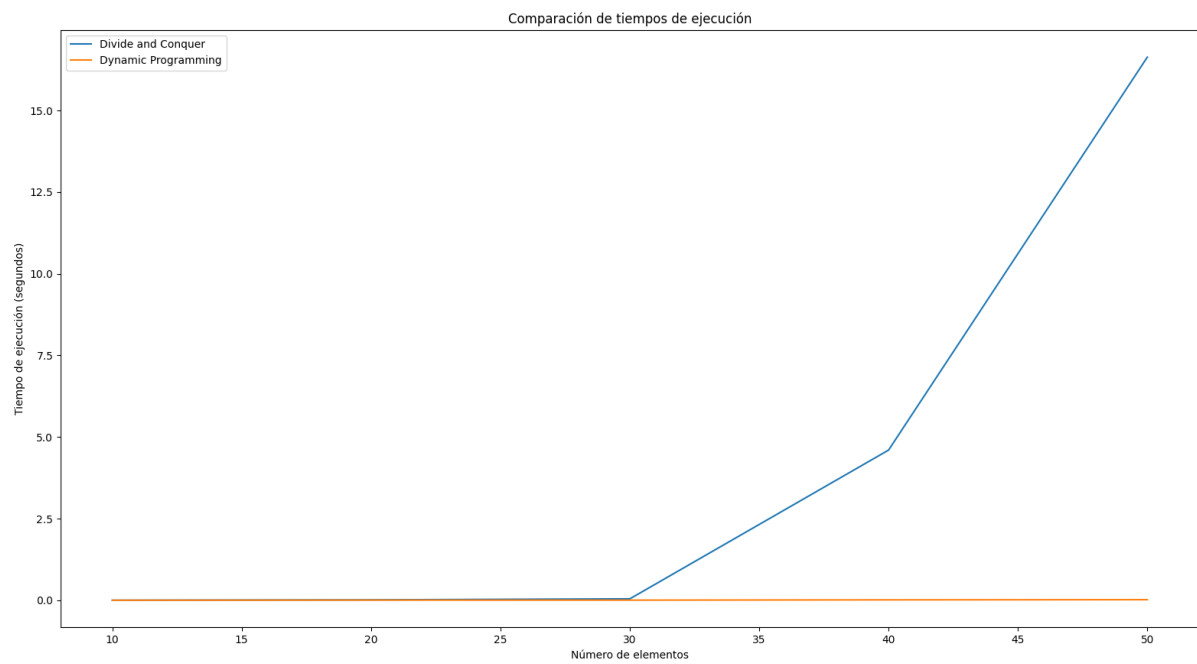
Gráfica 7: Tiempo de ejecución de DaC iteración 3



Gráfica 8: Tiempo de ejecución de DP iteración 3



Gráfica 9: Comparación de tiempos de ejecución iteración 3



Comentarios de comparación

Comparando los tiempos de ejecución, podemos notar lo siguiente:

- El algoritmo de Divide and Conquer tiene tiempos de ejecución mucho mayores que el algoritmo de Programación Dinámica, especialmente en casos con un mayor número de elementos. Por ejemplo, en el último caso, Divide and Conquer tarda aproximadamente 362 segundos en ejecutarse, mientras que el algoritmo de Programación Dinámica tarda solo 0.0152 segundos. Esto indica que el algoritmo de Programación Dinámica es más eficiente y puede manejar casos más grandes en comparación con el algoritmo de Divide and Conquer.
- El algoritmo de Programación Dinámica utiliza técnicas de memoización, lo que le permite recordar resultados de sub problemas previamente resueltos y evitar volver a calcularlos. Esto reduce significativamente el tiempo de ejecución en comparación con el algoritmo de Divide and Conquer, que no utiliza ninguna técnica de optimización y recalcula los subproblemas varias veces.
- Aunque el algoritmo de Divide and Conquer es más fácil de entender e implementar, su rendimiento es mucho peor que el algoritmo de Programación Dinámica en términos de tiempo de ejecución. Por lo tanto, en situaciones donde el rendimiento y la eficiencia son críticos, especialmente cuando se trata de casos de mayor tamaño, se recomienda utilizar el algoritmo de Programación Dinámica en lugar del algoritmo de Divide and Conquer.

Como resultado, podemos concluir que, el algoritmo de Programación Dinámica es una opción mucho mejor en términos de rendimiento y eficiencia en comparación con el algoritmo de Divide and Conquer para resolver el problema de la mochila. A pesar de ser un poco más complejo de implementar, sus ventajas en términos de tiempo de ejecución lo convierten en una opción más adecuada para casos de mayor tamaño y situaciones en las que el rendimiento es crítico.

Bibliografía

GeeksforGeeks. (24 de febrero, 2023). Recupeardo de: 0/1 Knapsack Problem.
<https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>

Google OR-Tools. (18 de enero, 2023). The Knapsack Problem. Recuperado de:
<https://developers.google.com/optimization/pack/knapsack>

Terh, F. (28 de marzo, 2019). How to solve the Knapsack Problem with Dynamic Programming. Recuperado de:
<https://medium.com/@fabianterh/how-to-solve-the-knapsack-problem-with-dynamic-programming-eb88c706d3cf>

Thelin, R. (8 de octubre, 2020). Demystifying the 0-1 knapsack problem: top solutions explained. Recuperado de:
<https://www.educative.io/blog/0-1-knapsack-problem-dynamic-solution>