

Greedy & Programación Dinámica

26 de mayo

¿Por qué nos importan estos métodos?



¿Por qué nos importan estos métodos?

- Visualicemos los siguientes problemas

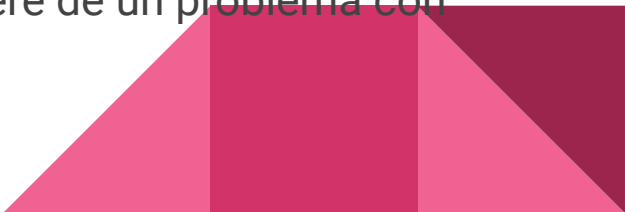
Queremos encontrar la ruta más corta entre ciudad A y ciudad B utilizando carreteras interconectadas

Tenemos monedas con valores de 3, 5 y 2. Queremos entregar un vuelto de exactamente **C** utilizando la mínima cantidad de monedas

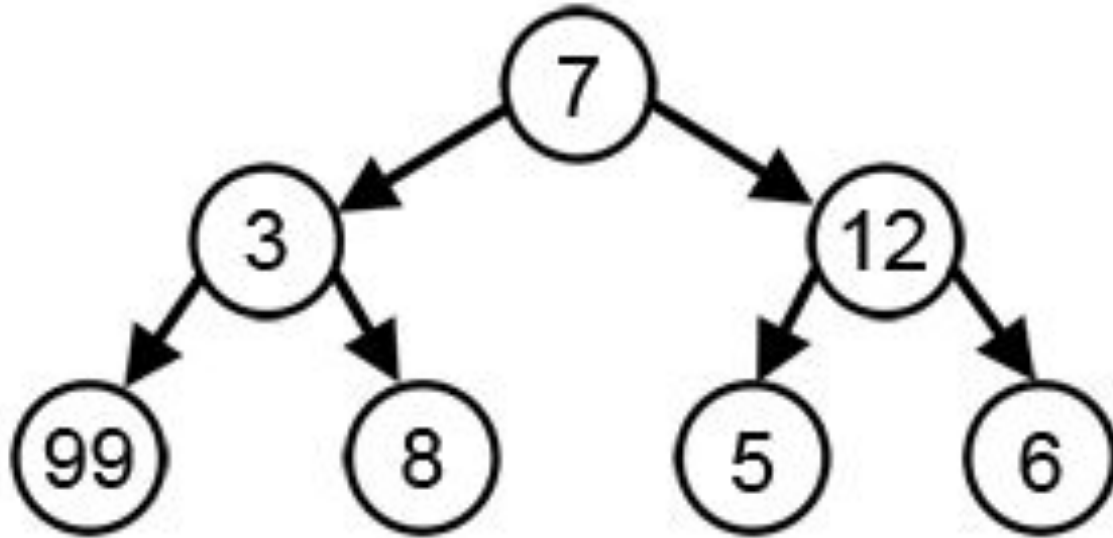
¿Cómo resolvemos este problema, asegurando que nuestra solución es óptima?



Greedy

- Un algoritmo greedy es un enfoque de resolución de problemas que se basa en tomar decisiones locales óptimas en cada paso con la esperanza de alcanzar una solución global óptima.
 - Comienza con una solución vacía y agrega gradualmente elementos o toma decisiones que maximicen una métrica o función objetivo en cada paso.
 - No considera ni revisa las decisiones tomadas anteriormente, solo se enfoca en la decisión actual basada en la información disponible en ese momento.
 - No garantiza una solución óptima global y puede quedar atrapado en mínimos locales o soluciones parcialmente satisfactorias. (Requiere de un problema con subestructura optima)
- 

Greedy



Problema corto

Estamos a final de semestre y quieres alcanzar a realizar la mayor cantidad de actividades antes de que el semestre cierre.

Supongamos que tienes un conjunto de actividades **A**, cada una con un tiempo de inicio (t_i) y un tiempo de finalización (t_f).

El objetivo es seleccionar el conjunto máximo de actividades compatibles de manera que ninguna actividad se solape entre sí.



Problema corto

- ¿Por qué este problema es greedy?
- ¿Cómo lo soluciono de forma codiciosa?



Problema corto

Actividad 1: [Inicio: 1, Fin: 4]

Actividad 2: [Inicio: 3, Fin: 5]

Actividad 3: [Inicio: 0, Fin: 6]

Actividad 4: [Inicio: 5, Fin: 7]

Actividad 5: [Inicio: 3, Fin: 8]

Actividad 6: [Inicio: 5, Fin: 9]

Paso 1: Seleccionar la actividad 3, ya que es la que **finaliza primero**.

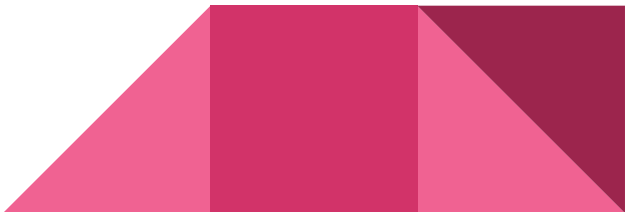
Paso 2: Descartar la actividad 1, ya que se solapa con la actividad 3.

Paso 3: Seleccionar la actividad 2, ya que es la que **finaliza primero después de la actividad 3**.

Paso 4: Descartar la actividad 4, ya que se solapa con la actividad 2.

Paso 5: Descartar la actividad 5, ya que se solapa con la actividad 2.


Paso 6: Seleccionar la actividad 6, ya que es la que finaliza primero después de la actividad 2.



Problema corto

```
ActivitySelection(A):  
    Sort activities in non-decreasing order of finish time  
    selectedActivities = [A[0]]  
    lastSelectedActivity = A[0]  
  
    for i = 1 to length(A) - 1:  
        if A[i].start >= lastSelectedActivity.finish:  
            selectedActivities.append(A[i])  
            lastSelectedActivity = A[i]  
  
    return selectedActivities
```

DP

- La programación dinámica es un enfoque de resolución de problemas que involucra dividir un problema en subproblemas más pequeños y resolverlos de manera independiente.
 - Se caracteriza por almacenar y reutilizar los resultados de los subproblemas para evitar recalcularlos, lo que mejora la eficiencia del algoritmo.
 - La programación dinámica busca encontrar la solución óptima global combinando las soluciones óptimas de los subproblemas. Es especialmente útil para problemas con superposición de subproblemas, donde los mismos subproblemas se resuelven repetidamente.
- 

DP

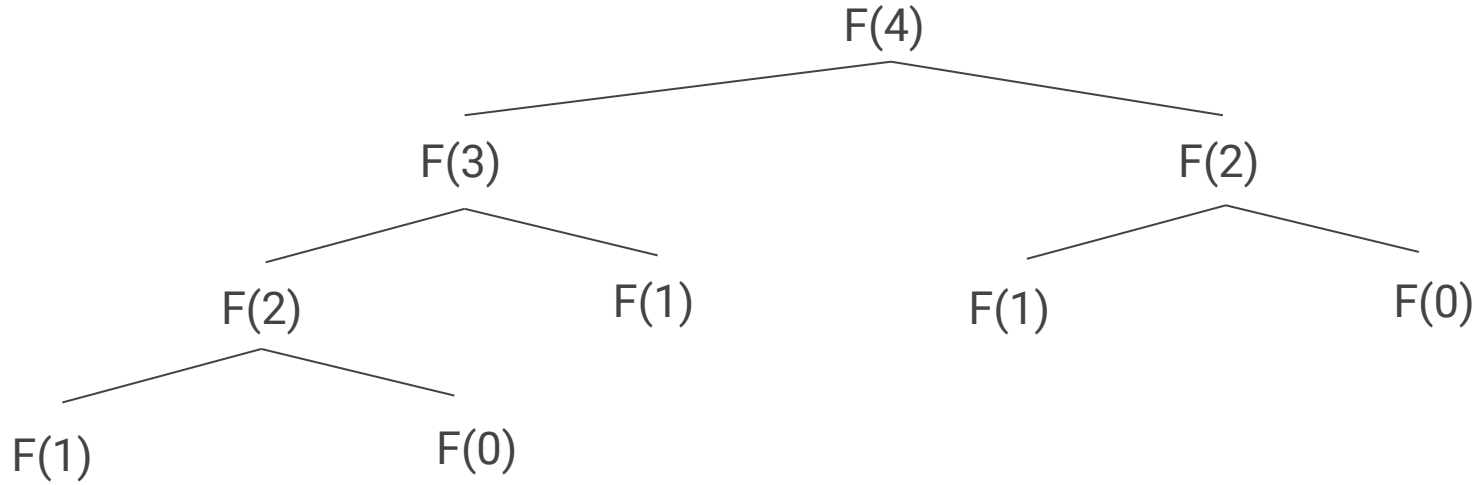
Se basa en la propiedad de la subestructura óptima, que establece que una solución óptima al problema puede ser construida a partir de soluciones óptimas de subproblemas más pequeños.

DP

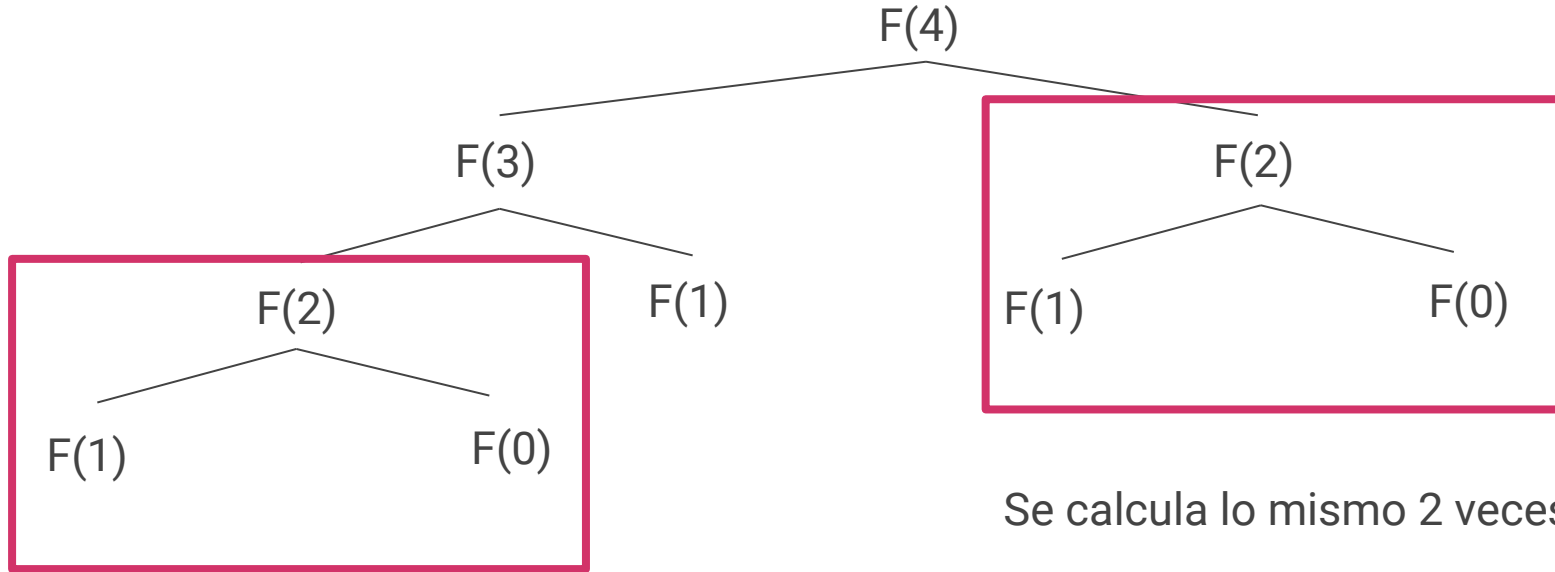
Problema: Calcular el n -ésimo número de Fibonacci.

1. La subestructura óptima en este problema radica en que el n -ésimo número de Fibonacci se puede calcular utilizando los resultados de los dos números de Fibonacci anteriores ($n-1$ y $n-2$).
2. Por ejemplo, para calcular el quinto número de Fibonacci ($n = 5$), necesitamos conocer los resultados de los números de Fibonacci anteriores ($n = 4$ y $n = 3$).
3. Estos a su vez se calculan utilizando los resultados de los números de Fibonacci aún más anteriores ($n = 3$, $n = 2$, $n = 2$ y $n = 1$).
4. La solución óptima para calcular el quinto número de Fibonacci es combinar las soluciones óptimas para calcular el cuarto y tercer número de Fibonacci.

DP



DP



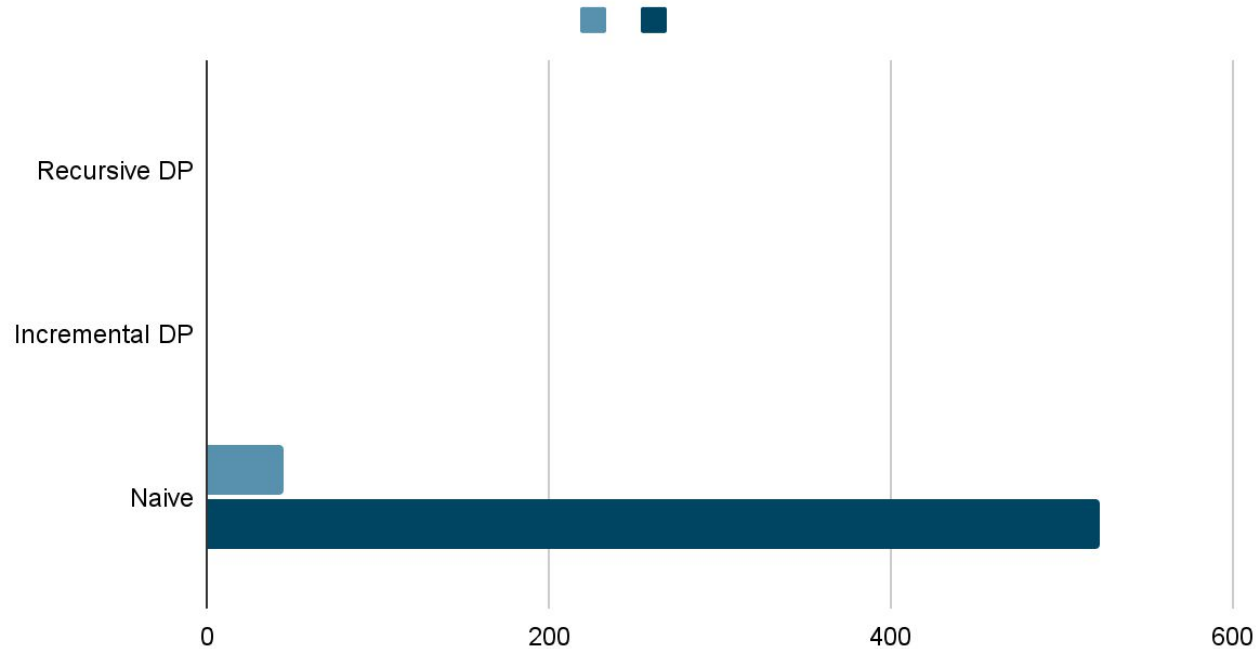
Se calcula lo mismo 2 veces!

DP

```
Fibonacci(n):  
    if n <= 1:  
        return n  
  
    fib = [0, 1]  
  
    for i = 2 to n:  
        fib.append(fib[i-1] + fib[i-2])  
  
    return fib[n]
```

DP (Tiempo de ejecucion)

Tiempo de ejecucion(Segundos)



Problema

Dado un conjunto de monedas con diferentes valores y un monto objetivo a devolver como vuelto, ¿cómo se puede encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo utilizando programación dinámica?

Objetivo:

Encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo de vuelto utilizando las monedas disponibles.

Restricciones:

- Pueden utilizarse múltiples monedas del mismo valor.
- El monto devuelto debe ser exacto
- No hay límite en la cantidad de monedas utilizadas.

Pregunta

¿Es subestructura optima? ¿Por qué?

Pregunta

¿Es subestructura óptima? ¿Por qué?

El problema del vuelto es de subestructura óptima porque la solución óptima para calcular la cantidad mínima de monedas necesarias para un monto objetivo se puede obtener combinando las soluciones óptimas para montos más pequeños, es decir, se puede construir a partir de las soluciones óptimas de subproblemas.

Pregunta

¿Es subestructura óptima? ¿Por qué?

El problema del vuelto es de subestructura óptima porque la solución óptima para calcular la cantidad mínima de monedas necesarias para un monto objetivo se puede obtener combinando las soluciones óptimas para montos más pequeños, es decir, se puede construir a partir de las soluciones óptimas de subproblemas.

Pregunta

Algoritmo propuesto:

1. Inicializar una matriz dp de tamaño $(monto + 1)$ con valores infinitos, excepto $dp[0] = 0$, ya que no se necesita devolver ningún vuelto.
2. Para cada monto desde 1 hasta $monto$, realizar los siguientes pasos:
 - Para cada moneda c en el conjunto de monedas:
 - Si c es menor o igual al monto actual y $dp[monto - c] + 1$ es menor que el valor actual en $dp[monto]$, actualizar $dp[monto]$ con $dp[monto - c] + 1$.
3. El valor en $dp[monto]$ representa la cantidad mínima de monedas necesarias para alcanzar el monto objetivo de vuelto.

Este enfoque de programación dinámica permite encontrar la cantidad mínima de monedas necesarias para devolver un vuelto específico. El algoritmo utiliza una matriz dp para almacenar y reutilizar los resultados de subproblemas más pequeños, mejorando así la eficiencia del cálculo del vuelto mínimo.

Pregunta

```
MinimumChange(coins, amount):  
    n = length(coins)  
    dp = [infinity] * (amount + 1)  
    dp[0] = 0  
  
    for i = 1 to amount:  
        for j = 0 to n-1:  
            if coins[j] <= i:  
                dp[i] = min(dp[i], 1 + dp[i - coins[j]])  
  
    return dp[amount]
```

¿Que otras aplicaciones hay para estas dos estrategias?

