Análisis y resolución de algoritmos

1) **Imprimir los primeros "n" términos de la sucesión de fibonacci. El "n" lo determina el usuario.**

Dato: La sucesión de fibonacci es una secuencia de números en la cual a excepción de los 2 primeros términos, cada término es la suma de los 2 los anteriores.

0, 1, 1, 2, 3, 5, 8, 13, 21...

Modelo recursivo

```
def fibo(n):
    if n < 2:
        return n
    return fibo(n-2) + fibo(n-1)

x = int(input("Ingrese un numero positivo n: "))
print("Se imprimira los n primeros terminos de la sucesion de fibnacci: ")

for i in range(x):
    print(fibo(i), end=' ')
```

Notar que la función fibo retorna de manera recursiva el enésimo término de la sucesión tomando el primer término como n = 0.

Esa función tal como está trabaja de manera muy ineficiente, si se emplea este algoritmo para imprimir los primeros 50 términos, jamás lo logrará.

Sucede que se repiten cálculos ya realizados anteriormente de forma innecesaria.

Ejemplo:

fibo(6)
= fibo(4) + fibo(5)
= fibo(2) + fibo(3) + fibo(3) + fibo(4)
= fibo(0) + fibo(1) + fibo(1) + fibo(2) + fibo(1) + fibo(2) + fibo(2) + fibo(3)
= 0 + 1 + 1 +fibo(0) + fibo(1) + 1 + fibo(0) + fibo(1) + fibo(0) + fibo(1) + fibo(1) + fibo(2)
...

Todo esto y más para calcular el sexto término fibo(6) = 8

Al resolver un problema de manera recursiva se emplea la estrategia "Divide y vencerás", ya que se subdivide el problema original en pequeños subproblemas los cuales se van resolviendo secuencialmente. Pero a veces, se crean demasiados

subproblemas y puede que en distintos ambientes recursivos se repitan los mismos, afectando la complejidad computacional del algoritmo.

La programación dinámica, es una estrategia de programación que permite optimizar en gran manera el tiempo de ejecución de algoritmos recursivos, mediante el uso inteligente de estructuras de datos, para poder "memorizar" tareas realizadas anteriormente.

Modelo recursivo con memorización

```
memoria = {0: 0, 1: 1}

def fibo_mem(n):

    if not n in memoria:
        memoria[n]= fibo_mem(n-1) + fibo_mem(n-2)
    return memoria[n]

x = int(input("Ingrese un numero positivo n: "))
print("Se imprimira los n primeros terminos de la sucesion de fibnacci: ")


for i in range(x):
    print(fibo_mem(i), end=' ')
```

Es intuitivo, voy anotando los cálculos que realizó y cuando me manden a realizar otro, revisar primero si ya lo he hecho. En este caso "se anota" los cálculos en un Diccionario. La clave sería la posición del término (desde 0), inicialmente tengo los dos primeros de la sucesión. De esta manera no se realizan cálculos innecesarios.

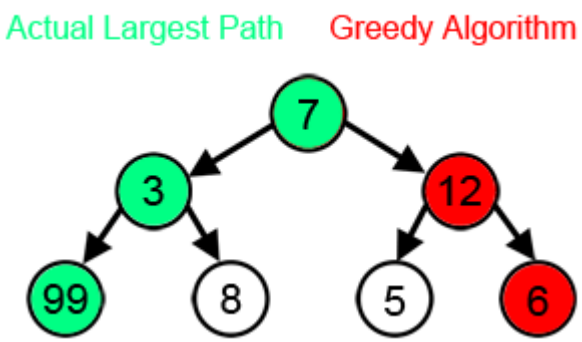
Anexado a esta píldora está un código donde puede comparar con mediciones de tiempo la rapidez de ambos modelos.

Ahora sí sería realista mandar a imprimir los primeros 50 términos de la sucesión.

Algoritmos voraces (Greedy algorithms)

Son algoritmos en el cual en cada estado, de tener varias posibles caminos van a escoger el que mejor le parezca entre todos, sin importar que ese camino a sea sólamente a simple vista el correcto. No siempre arrojan la solución óptima al problema, pero reducen el tiempo de búsqueda. Es una estrategia de programación que es bastante usada para problemas de optimización.

En la siguiente imagen se muestra cómo se comportaría un algoritmo voraz. En el siguiente árbol se quiere seleccionar el camino más costoso, desde la raíz hasta alguna de sus hojas. En cada nodo se muestra su valor.



nodo actual: 7
hijos: 3 y 12 elegimos 12 puesto que 3 < 12
nodo actual: 12
hijos: 5 y 6 elegimos 6 puesto que 5 < 6

pero 7 + 12 + 6 = 25
el camino más costoso es 7 + 3 + 99 = 109

Por lo que hemos visto, los algoritmos voraces pueden parecer poco útiles, pero con ligeras modificaciones lograremos además de encontrar la solución correcta, un óptimo desempeño en tiempo.

Típicos ejemplos de algoritmos voraces son:

- ## Algoritmo de Dijkstra

  Es un algoritmo que encuentra las rutas más baratas partiendo un vértice origen al resto de vértices en un grafo conexo ponderado. Un grafo ponderado tiene la característica de que sus aristas tienen un valor asociado. Un grafo ponderado fácilmente puede ser asociado con una red de computadoras y valor de las aristas a algún costo que le pueda tomar en viajar la información, este es apenas una de las tantas analogías que se le puede dar a un grafo ponderado.

  Edsger Dijkstra, científico de la computación neerlandés inventó este algoritmo en 1959.

  Algoritmo original: (distancias = valor de las aristas = costo)

- **Dado un conjunto de vértices y aristas ponderadas, tener una estructura de datos "D" donde se guardarán las distancias menos costosas desde el vértice inicial "x" hasta los demás.**

- **Establecer todas las distancia en "D" en infinito , pues son desconocidas (para el programa) a excepción de x la cual se coloca en 0, pues claramente la distancia de x a x es 0.**

- **x se toma como vértice actual.**

- **Se consideran las distancias de los vértices adyacentes (vecino_i) al actual (ignorando los visitados, luego se entenderá).**

  Se procede con la siguientes instrucciones:

  p_vecino = D[vertice_acutal] + distancia(x, vecino_i).
  si p_vecino < D[vecino] entonces D[vecino] = p_vecino

  Es decir, se compara las distancias y si es menor a la que está almacenada en en la estructura D, se actualiza. El vértice actual se marca como visitado. Ahora se toma como vértice actual el que tenga menor distancia en D y que no haya sido visitado y este paso se realizará hasta que no quede vértices sin visitar se ejecuta este mismo paso.

Notar que es voraz porque en cada estado se toma arbitrariamente se toma el vértice al que por simple inspección es menos costoso, pero eso no significa que pertenezca a la ruta más barata, Entonces gracias a que se actualiza la estructura D en cada estado no caemos en el error que puede producirse por tomar decisiones por simple inspección.

2) Realizar un programa que aplique el algoritmo de Dijkstra, para encontrar la ruta más barata desde cualquier vértice de un grafo conexo ponderado hasta cualquier otro.

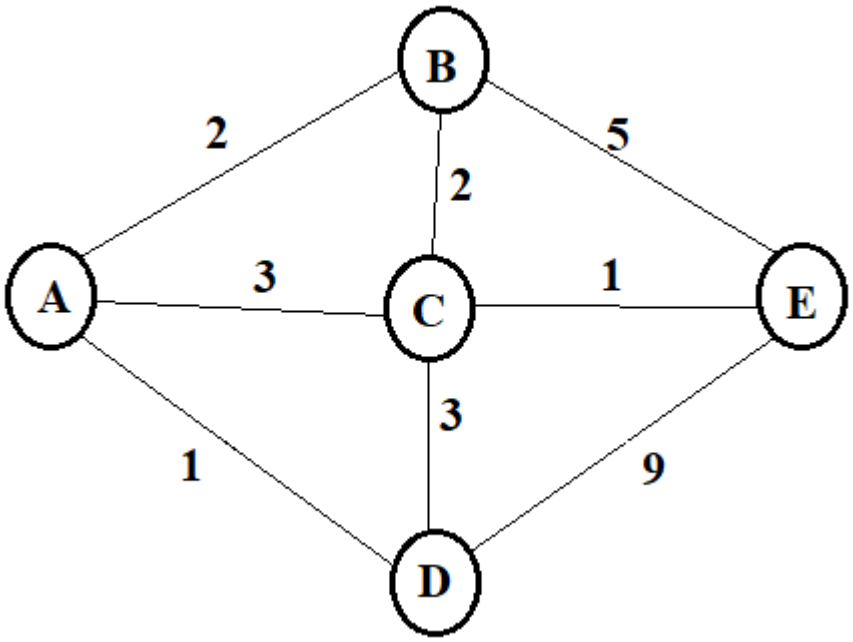
Se anexa a la píldora el archivo Dijkstra.py donde se resuelve.

El programa lee un archivo "entrada.txt" que debe estar situado en el mismo directorio que el código fuente, este archivo contiene la información pertinente a los vértices y valores de las aristas del grafo con el que va a trabajar.

El formato del archivo de entrada sería el siguiente:

 # de vértices del grafo
nombre de un vértice i
nombre de un vértice adyacente j, distancia al vértice adyacente j …. (pueden haber varios)

Es decir, en una línea se coloca el nombre de un vértice y en la siguiente, los adyacentes y sus respectivas distancias separados por un espacio. Se dejará un archivo "entrada.txt" ya creado, Dijkstra.py está listo para trabajar con el siguiente grafo:



En el archivo los nombres de los vér

- ## Problema del cambio de monedas (método voraz)

Este otro problema famoso en el campo de análisis de algoritmos, su solución óptima se consigue mediante programación dinámica , pero es posible (no siempre) conseguirla mediante un muy sencillo algoritmo voraz.

Sea N un número positivo, y S un conjunto finito de denominaciones distintas de monedas. Se desea calcular la cantidad mínima de monedas tales que entre ellas sumen N. Se asume que la cantidad de monedas para cada denominación es infinita.

Ejemplo:

N = 70
S = { 1, 5, 20, 35, 50}

Solución: una moneda de 20 y otra de 50

El algoritmo voraz sería el siguiente:

- **Conociendo N, se crea una estructura de datos "D" vacía, donde se guardarán las monedas de la solución.**

- **Se selecciona una moneda de la denominación más alta y que a su vez no sea mayor a N, llamaremos a esa moneda "P" y se agrega a D.**

- **N = N - P y se repite el paso anterior hasta que N sea 0**

En el ejemplo dado, aplicando este algoritmo sería así, el valor de N actual se pintó de rojo:

70 - 50 = 20 20 - 20 = 0 por eso la solución es 50 y 20.

En sistemas canónicos de monedas como el de Estados Unidos y el de otros países, este algoritmo funciona. Pero para sistemas arbitrarios no. Si las denominaciones de las monedas fueran 1, 3 y 4, entonces para obtener 6, el algoritmo voraz retornaría solución con tres monedas (4,1,1) mientras que la solución óptima es dos monedas (3,3).

Anexado a esta píldora se deja un código "Moneda.py" donde con programación dinámica se resuelve este problema.