Diseño de Algoritmos

November 3, 2022

Resolución de problemas

- La Resolución de Problemas es una actividad diaria en la vida laboral de un programador profesional.
- Al resolver un problema lo que hacemos es buscar un método o una técnica que permita hallar una solución.
- Los problemas admiten distintas soluciones. No hay un sólo camino o forma de resolverlos.
- Muchas veces encontrar el método puede no ser trivial, o puede requerir el uso o combinación de varias técnicas.
- Gran parte del aprendizaje proviene del "hacer", del "sumergirse" en el problema para entenderlo y así hallar una solución.
- La práctica y el entrenamiento nos dará agilidad a la hora de encontrar una solución. Nos dará "mental powers".

El método Polya

Método general de resolución de problemas en 4 pasos diseñado por el matemático George Pólya.

1. Analizar el problema.
2. Construir un plan de solución.
3. Ejecutar el plan de solución.
4. Revisión y mejoramiento de la solución.

El método Pòlya para algoritmos

Al resolver problemas con la ayuda de una computadora, lo que debemos encontrar para tener una solución es un *algoritmo* que lo resuelva. Los pasos del método de Pólya se convierten en:

1. Analizar el problema.
2. Diseñar y especificar un algoritmo que solucione el problema.
3. Escribir un programa que implemente nuestro algoritmo de solución.
4. Verificación y mejoramiento de la solución.

Diseño de algoritmos

Durante la clase de hoy, nos centraremos en la etapa 2, particularmente en el **diseño** de algoritmos.

Diseño de algoritmos

Aunque en la solución de problemas sencillos parezca evidente la codificación en un lenguaje de programación concreto, es aconsejable realizar el diseño del algoritmo, a partir del cual se codifique el programa.

La soluciones a problemas más complejos pueden requerir muchos más pasos. Las estrategias seguidas usualmente a la hora de encontrar algoritmos para problemas complejos son las mismas y veremos algunas en la clase de hoy

Fuerza Bruta

Problema Un usuario guarda información importante adentro de un archivo cifrado. Por desgracia, ha olvidado la contraseña y ahora no puede acceder a sus datos. Sí recuerda que la contraseña solo contenía letras minúsculas y eran a lo sumo 8 letras.
Idea Probar combinaciones al azar hasta dar con la correcta:

- asdfg
- ertu
- fafafa
- ...

Fuerza Bruta

Idea Probar todas las combinaciones posibles, en orden:

- a
- b
- c
- ...
- z
- aa
- ab
- ...

Como no tenemos información sobre la contraseña, no nos queda otra que probar todo. Esto es un algoritmo de fuerza bruta. Sin embargo, solo tiene sentido cuando podemos enumerar las posibles soluciones en un orden lógico.

```
alfabeto = "abcdefghijklmnopqrstuvwxyz"

def es_solucion(intento):
    return intento == "abcde"

def siguiente(intento = ""):
    if intento == alfabeto[-1] * len(intento):
        if len(intento) == 8:
            return None
        return alfabeto[0] * (len(intento) + 1)
    elif intento[-1] != alfabeto[-1]:
        mantengo = intento[:-1]
        proxima =alfabeto[alfabeto.find(intento[-1]) + 1]
        return  mantengo + proxima
    else:
        return siguiente(intento[:-1]) + alfabeto[0]

intento_actual = siguiente()

while intento_actual and not es_solucion(intento_actual):
    intento_actual = siguiente(intento_actual)

print(intento_actual)
```

Fuerza Bruta

El bloque de código:

```
intento_actual = siguiente()

while intento_actual and not es_solucion(intento_actual):
    intento_actual = siguiente(intento_actual)

print(intento_actual)
```

es común a todas las resoluciones por fuerza bruta. Dado entonces un problema, para resolverlo por fuerza bruta solo necesitamos:

- Una función `es_solución` que verifique si un intento es correcto.
- Una función `siguiente` que dado un intento, nos de el próximo intento.

Fuerza Bruta

Ventajas

- Es una técnica fácil de implementar y de leer, por lo que se suele utilizar cuando se quiere tener implementaciones fáciles de probar y depurar.
- Hay problemas que admiten muchas soluciones. Cuando se usa una técnica de fuerza bruta, es fácil adaptar el programa para encontrar *todas* las soluciones del problema, o una cantidad K de soluciones, o buscar soluciones hasta haber consumido cierta cantidad de recursos (tiempo, memoria, CPU, etc.).

Desventajas

- La cantidad de posibles soluciones a explorar por lo general crece exponencialmente a medida que crece el tamaño del problema.
- Depende mucho del poder de cómputo de la máquina para resolver el problema.

Fuerza Bruta

Ejercicio 1

¿Cómo se vería un algoritmo que resuelve el problema de ordenamiento por fuerza bruta?

Ejercicio 2

Encuentre todos los divisores de un número natural n utilizando un algoritmo de fuerza bruta.

Fuerza bruta

Adaptación para que el algoritmo de fuerza bruta encuentre todas las soluciones

```
intento_actual = siguiente()
soluciones = []
while intento_actual:
    if es_solucion(intento_actual):
        soluciones.append(intent_actual)
    intento_actual = siguiente(intento_actual)

print(soluciones)
```

Solución Ejercicio 2

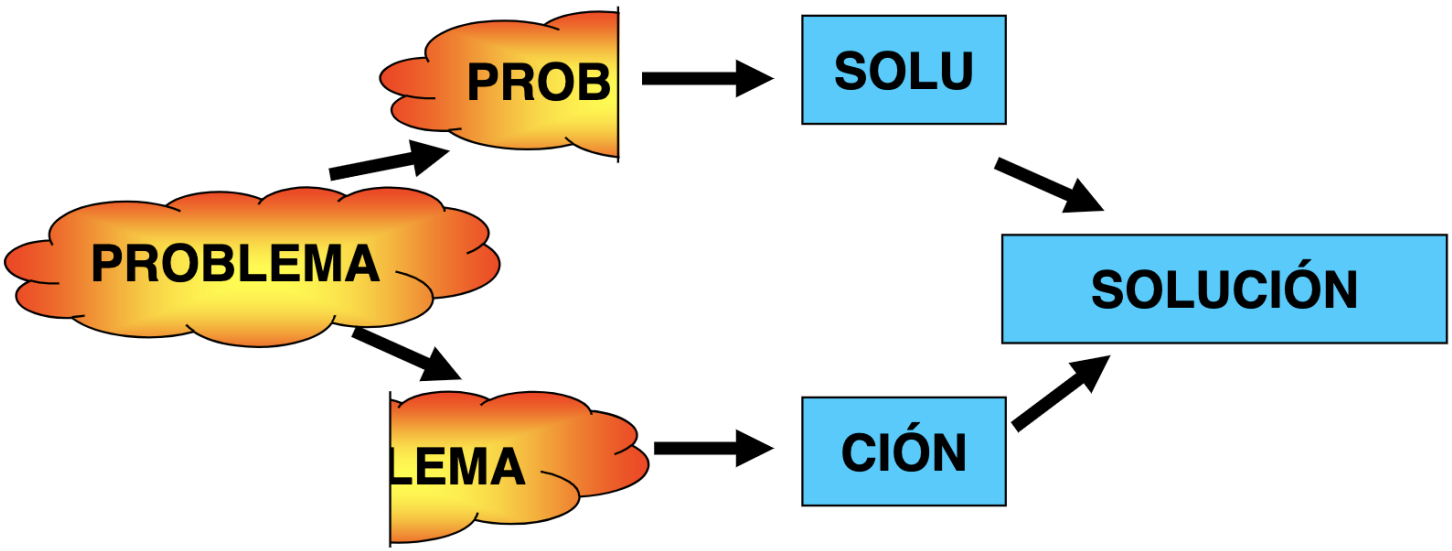
```
intento_actual = siguiente()
soluciones = []
while intento_actual:
    if es_solucion(intento_actual):
        soluciones.append(intent_actual)
    intento_actual = siguiente(intento_actual)

print(soluciones)
```

Divide y Vencerás

Muchos algoritmos útiles son recursivos, es decir, se llaman a sí mismos para resolver subproblemas más pequeños.
La técnica Divide y Vencerás permite generar algoritmos recursivos para resolver una amplia variedad de problemas. Consiste en:

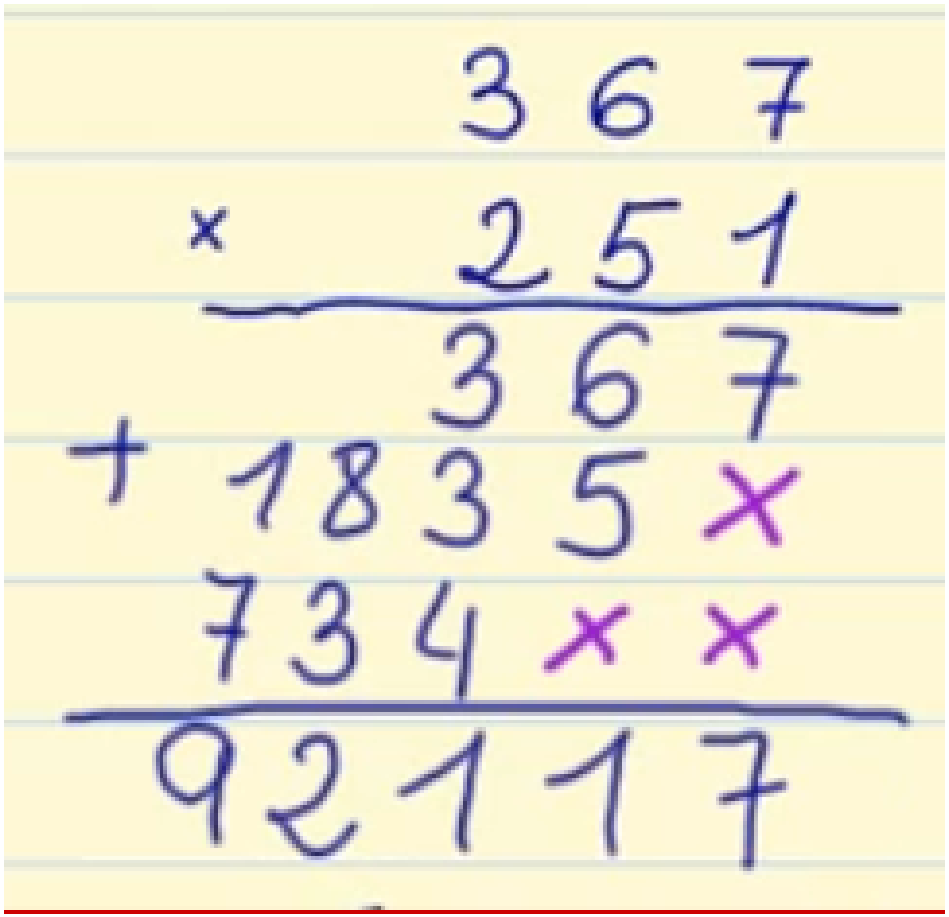
1. Descomponer un problema en un conjunto de subproblemas más pequeños.
2. Resolver los subproblemas más pequeños.
3. Combinar las soluciones de los subproblemas más pequeños para obtener la solución general.

Divide y Vencerás



Divide y Vencerás

Ejemplo Multiplicar por un número de 3 cifras



Divide y Vencerás

1. Descomponemos la multiplicación por un número de tres cifras en tres multiplicaciones por un número de una cifra.
2. Resolvemos las multiplicaciones más fáciles.
3. Combinamos los tres resultados, agregando los ceros según corresponda y sumando.

Divide y Vencerás

La estructura general de un algoritmo de este tipo es

```
def resolver(problema):
    if es_caso_base(problema):
        return resolver_caso_base(problema)

    subproblema1, subproblema2 = dividir(problema)
    solucion1 = resolver(subproblema1)
    solucion2 = resolver(subproblema2)

    return combinar(solucion1, solucion2)

print(resolver(problema))
```

Solo necesitamos identificar

- Una función `es_caso_base` que verifique si estamos en un caso base.
- Una función `resolver_caso_base` que le da solucion a los casos base.
- Una función `dividir` que nos indique como dividir el problema en varios subproblemas relacionados, pero de menor tamaño.
- Una función `combinar` que nos permita juntar las soluciones parciales del caso anterior para obtener la solución.

Notas

- Usualmente los subproblemas se resuelven recursivamente, pero no necesariamente.
- La cantidad de subproblemas en el que dividiremos el problema original puede variar.

Divide y Vencerás

Problema Calcular el enésimo número de Fibonacci.

```
def es_caso_base(p):
    return True if p <=1 else False

def dividir(p):
    return p - 1, p - 2

def combinar(s1, s2):
    return s1 + s2

def resolver(problema):
    if es_caso_base(problema):
        return resolver_caso_base(problema)
    subproblema1, subproblema2 = dividir(problema)
    solucion1 = resolver(subproblema1)
    solucion2 = resolver(subproblema2)

    return combinar(solucion1, solucion2)

print(resolver(problema))
```

Ventajas

- Por lo general, da como resultado algoritmos elegantes y eficientes.
- Como los subproblemas son independientes, si poseemos una máquina con más de un procesador podemos *paralelizar* la resolución de los subproblemas, acortando así el tiempo de ejecución de la solución.

Desventajas

- Los subproblemas en los que dividimos el problema original deben ser *independientes*, de lo contrario, intentar aplicar Divide y Vencerás estará condenado al fracaso.
- Se necesita cierta intuición para ver cuál es la mejor manera de descomponer un problema en partes. Esto solo se consigue con la práctica.
- A veces, como en el caso de Fibonacci, los subproblemas son *independientes* pero no *disjuntos*, lo que puede ocasionar trabajo extra al recomputar muchas veces el mismo valor. La técnica de **Programación Dinámica** es más apropiada en estos casos, pero no la veremos en este curso.
- Si bien los algoritmos que genera utilizar esta técnica son eficientes, demostrar tal eficiencia requiere matemáticas avanzadas.

Ejercicio 1

¿Cómo se vería un algoritmo que resuelve el problema de ordenamiento utilizando la técnica Divide y Vencerás?

Ejercicio 2

Dada una lista ordenada de números enteros, determinar eficientemente si un número se encuentra en ella o no.

Algoritmos Voraces

Decimos que un algoritmo es voraz o *greedy* cuando en cada paso toma la mejor decisión posible en ese momento hasta conseguir una solución para el problema.

Los algoritmos voraces a veces encuentran una solución óptima, y a veces no. Sin embargo, muchas veces esta pérdida de precisión es aceptable siempre y cuando el algoritmo encuentre soluciones "lo suficientemente buenas".

Algoritmos Voraces

Problema Tenemos billetes de 1000, 500, 200 100, 50, 20 y 10 pesos. Si un cliente gastó 960 pesos, pagó con 1000 pesos y suponiendo que tengo cantidad suficiente de todos los billetes, ¿cual es la mejor forma de darle vuelto, minimizando la cantidad de billetes que entrego?

Lo mas sensato, es dar dos billetes de 20 pesos. Así, minimizamos la cantidad de billetes que debemos entregar

Algoritmos Voraces

Observemos que siempre me conviene entregar un billete de denominación lo más alta posible, sin pasarme del valor que debo devolver.

Si luego de elegir esta billete sigo debiendo , vuelvo a repetir con lo que me quede.

```
total = 20
candidatos =[1000, 500, 200, 100, 50, 20, 10] * 2

def es_solucion(eleccion_actual):
    return sum(eleccion_actual) == total

def elegir_candidato():
    return max(candidatos)

def es_factible(eleccion):
    return sum(eleccion) <= total

eleccion_actual = []

while not es_solucion(eleccion_actual):
    x = elegir_candidato()
    candidatos.remove(x)
    if es_factible(eleccion_actual + [x]):
        eleccion_actual.append(x)

print(eleccion_actual)
```

Algoritmos Voraces

El fragmento

```
eleccion_actual = []

while not es_solucion(eleccion_actual):
    x = elegir_candidato()
    candidatos.remove(x)
    if es_factible(eleccion_actual + [x]):
        eleccion_actual.append(x)

print(eleccion_actual)
```

es común a todos los algoritmos voraces, también conocidos como *greedy*.

Solo debemos identificar:

- Un función ‘es_solucion‘ que decida si la solución es válida.
- Una función ‘elegir_candidato‘ que nos indique como elegir un candidato de forma adecuada.
- Una función ‘es_factible‘ que nos indique si la solución propuesta tiene sentido.

Algoritmos Voraces

Problema En el país Albariocoque la moneda oficial es el fiji. Tienen billetes de 100 fijis y monedas de 4 fijis, 3 fijis y 1 fiji. Si un cliente gastó 96 fijis, pagó con 100 fijis y suponiendo que tengo cantidad infinita de todas las monedas, ¿cual es la mejor forma de darle vuelto, minimizando la cantidad de monedas que entrego?

Algunas formas posibles de darle el vuelto, serian:

- seis monedas de un fiji.
- tres monedas de un fiji y una moneda de tres fijis.
- dos monedas de tres fijis
- una moneda de cuatro fijis y dos monedas de un fiji.

Este proceso, de buscar todas las posibilidades y ver cual me conviene utilizar, es una fuerza bruta. Podemos ver que la solución óptima es elegir entregar dos monedas de 3 fijis. Sin embargo, nuestro algoritmo greedy hubiera entregado una moneda de 4 fijis y dos monedas de 1 fiji, dando una solución subóptima.

Algoritmos Voraces

Ventajas

- Este tipo de algoritmos es útil en problemas de optimización, es decir, cuando hay una variable que maximizar o minimizar.

Desventajas

- Hay que tener cuidado. No siempre la solución que encontremos será óptima.
- Si necesitamos una solución óptima, se puede intentar demostrar matemáticamente que el algoritmo voraz es correcto, o se puede intentar con una técnica más avanzada, como la programación dinámica.

Algoritmos Voraces

Problema 1 ¿Hay algún algoritmo de ordenamiento que conozcas que trabaje de forma voraz?

Problema 2 Un algoritmo voraz recorre el siguiente árbol, comenzando desde la raíz y eligiendo en cada paso el nodo de más valor, intentando encontrar el camino desde la raíz de mayor suma. ¿Qué camino elegirá el algoritmo? ¿Es óptima la respuesta?