Diseño de Algoritmos

November 13, 2023

Contenido

Introducción

1. ¿Computadoras vs. Computación?
2. Breve historia del algoritmo.

Definición de algoritmo

El concepto intuitivo de algoritmo, lo tenemos prácticamente todos: Un algoritmo es una serie finita de pasos para resolver un problema.
Hay que hacer énfasis en dos aspectos para que un algoritmo exista:
1. El número de pasos debe ser finito. De esta manera el algoritmo debe terminar en un tiempo finito con la solución del problema,
2. El algoritmo debe ser capaz de determinar la solución del problema.
De este modo, podemos definir algoritmo como un "conjunto de reglas operacionales inherentes a un cómputo". Se trata de un método sistemático, susceptible de ser realizado mecánicamente, para resolver un problema dado.

Características de un algoritmo

1. Entrada: definir lo que necesita el algoritmo
2. Salida: definir lo que produce.
3. No ambiguo: explícito, siempre sabe qué comando ejecutar.
4. Finito: El algoritmo termina en un número finito de pasos.
5. Correcto: Hace lo que se supone que debe hacer. La solución es correcta
6. Efectividad: Cada instrucción se completa en tiempo finito. Cada instrucción debe ser lo suficientemente básica como para que en principio pueda ser ejecutada por cualquier persona usando papel y lápiz.
7. General: Debe ser lo suficientemente general como para contemplar todos los casos de entrada.

Algoritmos vs. Problemas vs. Programas

Un **algoritmo** es un método o proceso seguido para resolver un problema.
Un **problema** es una función o asociación de entradas con salidas.
Nota: Un mismo problema puede tener varios algoritmos.
Un **programa** es una instanciación de un algoritmo en un lenguaje de programación.

Resolución de problemas

La fase de resolución del problema se puede descomponer en tres etapas:
o Análisis de alternativas y selección de la solución.
o Especificación detallada del procedimiento solución.
o Adopción o utilización de una herramienta para su implementación, si es necesaria.
En el campo de las ciencias de la computación la solución de problemas se describe mediante el diseño de algoritmos, los cuales posteriormente se implementan como programas. Los programas son procedimientos que solucionan problemas y que se expresan en un lenguaje conocido por el computador.
El resto de la clase exploraremos distintas estrategias para elaborar algoritmos.

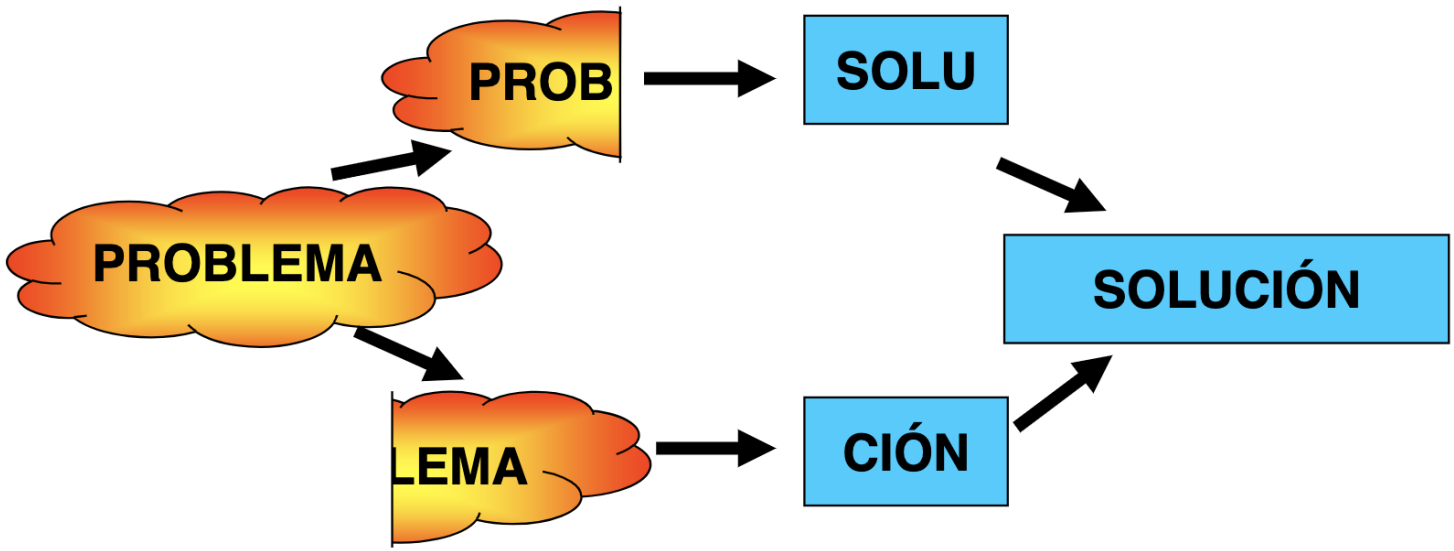
Contenido

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

Muchos algoritmos útiles tienen una estructura recursiva, de modo que para resolver un problema se llaman recursivamente a sí mismos una o más veces para solucionar subproblemas muy similares. Esta estructura obedece a una estrategia divide-y-vecerás (*divide conquer*, o simplemente DC), en que se ejecuta tres pasos en cada nivel de la recursión:

1. **Dividir** el problema en varios subproblemas similares al problema original pero de menor tamaño;
2. **Conquistar**: Resolver recursivamente los subproblemas. Si los tamaños de los subproblemas son suficientemente pequeños, entonces resuelven los subproblemas de manera directa; y luego,
3. **Combinar** estas soluciones para crear una solución al problema original.

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 solución 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

- La cantidad de subproblemas en el que dividiremos el problema original puede variar.

Divide y Vencerás

Para aplicar la estrategia Divide y Vencerás es necesario que se cumplan tres condiciones:
· Debemos elegir cuidadosamente cuales serán nuestros casos base, es decir, cuando resolveremos el problema sin necesidad de dividir. · Tiene que ser posible descomponer el caso en subcasos y recomponer las soluciones parciales de forma eficiente. · Los subcasos deben ser, en lo posible, aproximadamente del mismo tamaño.

Divide y Vencerás - Ejemplo

¿Como multiplicar números largos?

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))
```

Divide y Vencerás

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.

Divide y vencerás

Ejercicio 1

Escriba un algoritmo divide y vencerás para ordenar una lista de números enteros

Ejercicio 2

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

Contenido

Algoritmos Voraces

Los algoritmos ávidos o voraces (*Greedy Algorithms*) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.
Suelen ser bastante simples y se emplean sobre todo para resolver problemas de optimización, como por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por un computador.

Algoritmos Voraces

Habitualmente, los elementos que intervienen son:
· un conjunto o lista de **candidatos** (tareas a procesar, por ejemplo);
· un conjunto de decisiones ya tomadas.
· una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene por qué ser la óptima);
· una función que determina si un conjunto de candidatos es una solución óptima al problema

Algortimos Voraces

Para resolver el problema de optimización hay que encontrar un conjunto de candidatos que optimiza la función objetivo. Los algoritmos voraces proceden por pasos, eligiendo un candidato por vez
. Inicialmente nuestra elección de solución es vacía. A continuación, en cada paso, se intenta añadir al conjunto el mejor candidato de los aún no escogidos, utilizando la función de selección.
Si el conjunto resultante no es plausible, se rechaza el candidato y no se le vuelve a considerar en el futuro.
En caso contrario, se incorpora al conjunto de candidatos escogidos y permanece siempre en él. Tras cada incorporación se comprueba si el conjunto resultante es una solución del problema.
Un algoritmo voraz es correcto si la solución así encontrada es siempre óptima.

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 94 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 búsqueda exhaustiva. 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

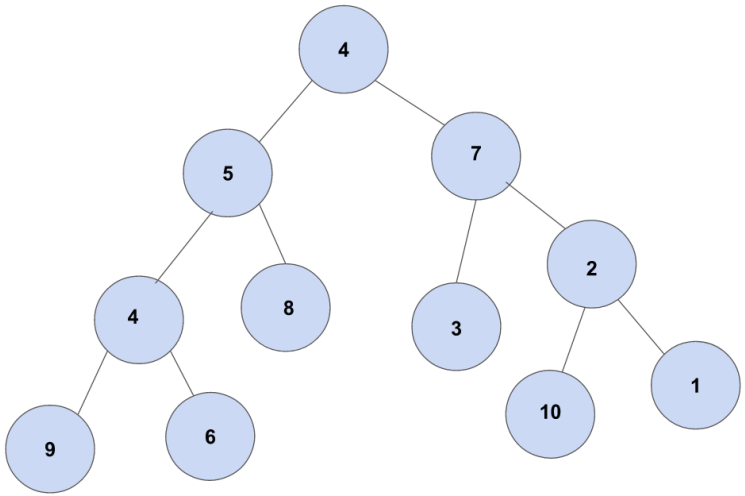
- Este tipo de algoritmos es útil en problemas de optimización, es decir, cuando hay una variable que maximizar o minimizar.

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 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?



Contenido

Búsqueda exhaustiva

La búsqueda exhaustiva, a veces llamada búsqueda por fuerza bruta es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de búsqueda exhaustiva para encontrar un divisor de un número natural n consiste en enumerar todos los enteros desde 2 hasta $n - 1$, chequeando en cada paso si divide a n sin dejar resto.

Búsqueda exhaustiva

```
N = 2047

def es_solucion(intento):
  if N % intento == 0: return True
  return False

def candidatos():
    for i in range(2, N):
      yield i

def resolver():
  for candidato in candidatos():
      if es_solucion(candidato):
        return candidato
  return -1 # Error

solucion = resolver()
print(f"{solucion} divide a {N}")
```

Para obtener un algoritmo de búsqueda exhaustiva necesitamos:

- Una función `es_solución` que verifique si un intento es correcto.
- Un generador `candidatos` que devuelva uno a uno los candidatos con los que hay que probar.

Búsqueda Exhaustiva

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 búsqueda exhaustiva, 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.

Búsqueda Exhaustiva

Pasos para resolver un problema mediante búsqueda exhaustiva:

1. Identificar el conjunto de candidatos a solución.
2. Representar formalmente cada candidato, pensar que tipo de datos tienen nuestros candidatos a solución. ¿Estamos buscando un número, una lista, una tupla de dos elementos?
3. Dar un orden lógico a los elementos de nuestro conjunto de candidatos.
4. Establecer nuestro objetivo. ¿Cuando podemos parar de buscar?
5. Basado en los puntos 2 y 4, escribir una función `es_solución` en Python.
6. Basados en los puntos 1, 2 y 3, escribir un generador que provea uno a uno los candidatos a solución en el orden establecido.
7. Iterar sobre los candidatos hasta en encontrar una solución.

Búsqueda Exhaustiva

En nuestro ejemplo:

1. El conjunto de candidatos es $\{x \in \mathbb{N} | 2 \leq x \leq n-1\}$.
2. Cada candidato a solución puede representarse en Python como un `int`.
3. Una posible forma lógica de recorrer nuestro conjunto de candidatos es recorrerlos de menor a mayor.
4. Podemos parar de buscar encontremos un candidato i tal que la división entre n e i sea exacta.
5. Programamos...
6. Programamos...
7. Programamos...

Búsqueda Exhaustiva

Ejercicio 1

Escriba un algoritmo de Búsqueda Exhaustiva que ordene una lista de números.

Búsqueda Exhaustiva

Ejercicio 1

1. El conjunto de candidatos es ... el conjunto de todas las posibles permutaciones de la lista.
2. Cada candidato puede representarse como... una lista!
3. ¿Como recorremos todas las posibles permutaciones de una lista? ...
4. ¿Cuando podemos parar de buscar? Cuando demos con una lista que esta ordenada!
5. Programar...

Búsqueda Exhaustiva

Adaptación para que el algoritmo de Búsqueda Exhaustiva encuentre todas las soluciones

Muchas veces encontramos problemas que admiten varias soluciones y estamos interesados en encontrar **todas** las soluciones del problema.

Es muy fácil adaptar el algoritmo de búsqueda exhaustiva para que encuentre todas las soluciones posibles a un problema dado - simplemente las vamos acumulando en una lista.

```
soluciones = []
for candidato in candidatos():
    if es_solucion(candidato):
        soluciones.append(candidato)

print(soluciones)
```

Búsqueda exhaustiva

Adaptación para que el algoritmo de Búsqueda Exhaustiva encuentre la mejor solución

Muchos problemas admiten varias soluciones. Sin embargo, muchas veces cada uno de estas posibles soluciones tiene un costo asociado - no da lo mismo elegir cualquiera.

En dichos casos, se puede adaptar el algoritmo de búsqueda exhaustiva para encontrar la **mejor** solución posible. Lo que hacemos es lo siguiente:

1. Agregamos una función `calcular_costo` que dado un candidato a solución, nos devuelva un número que cuantifique que tan buena es nuestra solución de alguna manera.
2. Cuando iteramos por los candidatos a solución, lo hacemos de forma que, si tenemos solución, calculamos el costo de la solución y comparamos con la mejor solución encontrada al momento.

Búsqueda exhaustiva

```
mejor_solucion = None
costo_mejor_solucion = float('inf')
for candidato in candidatos():
    if es_solucion(candidato):
        costo = medir(candidato)
        if costo < costo_mejor_solucion:
          mejor_solucion = candidato
          costo_mejor_solucion = costo

print(mejor_solucion)
```

El módulo `itertools`

Python trae incluido un módulo llamado `itertools`, el cual estandariza una serie de herramientas que se pueden utilizar para escribir generadores de forma más sencilla.

Tarea Investigar la documentación oficial de este modulo: https://docs.python.org/3/library/itertools.html