UNIVERSIDAD DE GUANAJUATO

Conceptos fundamentales de algoritmos

Dr. Mario Alberto Ibarra Manzano
Departamento de Ingeniería Electrónica
División de Ingeniería del Campus Irapuato-Salamanca

Conceptos fundamentales de algoritmos

- Conceptos y propiedades de los algoritmos
- Solución de problemas usando algoritmos
- Estrategias para construir algoritmos

❖Algoritmos de búsqueda exhaustiva y fuerza bruta

❖Top-Down y Botton-up

❖Divide y vencerás

Definición de algoritmo

Un algoritmo es un conjunto finito de instrucciones o pasos que se siguen para resolver un problema o realizar una tarea específica. Los algoritmos son fundamentales en la informática, ya que proporcionan un método sistemático para abordar problemas complejos y encontrar soluciones eficaces.

En el contexto de la ingeniería de datos y la inteligencia artificial (IA), un algoritmo es una secuencia finita y bien definida de instrucciones o reglas, diseñadas para realizar una tarea específica de procesamiento de datos, análisis o toma de decisiones de manera eficiente y automática. Estos algoritmos son fundamentales para convertir datos en conocimiento útil, facilitar el aprendizaje automático y automatizar procesos complejos.

Características de un algoritmo

Finitud: Un algoritmo debe tener un número finito de pasos.

Definición: Cada paso del algoritmo debe estar claramente definido y ser inequívoco.

Entrada: Un algoritmo debe tener una o más entradas, que son valores dados antes de que el algoritmo comience.

Salida: Un algoritmo debe producir una o más salidas, que son resultados esperados después de completar el proceso.

Efectividad: Los pasos de un algoritmo deben ser suficientemente básicos para ser llevados a cabo, en principio, por una persona usando papel y lápiz.

Características de un algoritmo (Ingeniería de Datos e Inteligencia Artificial)

Automatización: Los algoritmos permiten la automatización de tareas repetitivas y complejas, reduciendo la intervención humana.

Eficiencia: Deben estar optimizados para manejar grandes volúmenes de datos (Big Data) y realizar operaciones en tiempo real o casi real.

Adaptabilidad: Muchos algoritmos en IA, como los de aprendizaje automático, son adaptativos, lo que significa que pueden mejorar su desempeño a medida que procesan más datos.

Robustez: Deben ser capaces de manejar datos incompletos, ruidosos o inconsistentes, y producir resultados fiables.

Escalabilidad: Es crucial que los algoritmos puedan escalar para trabajar con cantidades crecientes de datos y requerimientos de procesamiento.

Importancia de los algoritmos

Solución Sistemática de Problemas: Los algoritmos proporcionan un método claro y sistemático para abordar problemas, lo que facilita su solución.

Eficiencia: Los algoritmos bien diseñados son eficientes en términos de tiempo y espacio.

Reusabilidad: Los algoritmos pueden ser reutilizados para resolver problemas similares.

Mantenimiento: Los algoritmos claros y bien documentados son más fáciles de entender y mantener.

Clasificación de algoritmos

1. Por su diseño:
a) Algoritmos recursivos.
b) Algoritmos iterativos.
2. Por su propósito:
a) Algoritmos de búsqueda.
b) Algoritmos de ordenamiento.
c) Algoritmos de grafos.
3. Por su complejidad:
a) Algoritmos de complejidad constante.
b) Algoritmos de complejidad lineal.
c) Algoritmos de complejidad polinómica.
d) Algoritmos de complejidad exponencial.

Propiedades de los algoritmos

Los algoritmos tienen varias propiedades esenciales que los caracterizan y aseguran que sean útiles para resolver problemas de manera efectiva. Estas propiedades son las siguientes:

1. Finitud
2. Claridad y Precisión
3. Entrada (Input)
4. Salida (Output)
5. Efectividad
6. Determinismo
7. Generalidad

Propiedades de los algoritmos

Finitud

Un algoritmo debe terminar después de un número finito de pasos. No debe ejecutarse indefinidamente; debe llegar a una conclusión en un tiempo razonable.

Claridad y Precisión

Cada paso del algoritmo debe estar claramente definido y ser inequívoco. Las instrucciones deben ser precisas y no deben dar lugar a ninguna ambigüedad.

Entrada (Input)

Un algoritmo debe tener cero o más entradas. Estas entradas son valores suministrados al inicio del algoritmo, sobre los cuales se realizan las operaciones.

Propiedades de los algoritmos

Salida (Output)
Un algoritmo debe tener una o más salidas. Las salidas son los resultados que se obtienen después de que el algoritmo haya completado su proceso. Estas salidas deben estar relacionadas con el objetivo del algoritmo.

Efectividad
Las operaciones descritas en el algoritmo deben ser lo suficientemente básicas como para ser realizadas, en principio, por una persona usando papel y lápiz en un tiempo finito. Esto significa que cada operación debe ser realizable y ejecutable.

Propiedades de los algoritmos

Determinismo
Dado un conjunto de entradas específicas, el algoritmo siempre debe producir las mismas salidas. Esto implica que no hay lugar para la aleatoriedad o la indeterminación en el algoritmo. Cada paso debe llevar de manera determinística al siguiente.

Generalidad
Un algoritmo debe ser aplicable a una clase de problemas en lugar de estar limitado a una instancia específica. Debe ser lo suficientemente general para resolver todos los problemas de una categoría particular.

Solución de problemas usando algoritmos

Cálculo de la función seno.

La serie de Taylor para la función $sin(x)$, donde x está expresado en radianes, se puede expresar como una serie infinita de potencias centrada en $x = 0$. La fórmula general de la serie de Taylor para $sin(x)$ es:

$$\sin(x) = \sum_{i=0}^{n\to\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Es decir,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

Esta serie converge para todos los valores de x.

Solución de problemas usando algoritmos

Cálculo de la función seno.

Supongamos los valores de $x = 2.3$ para 10 términos:

$$\sin(x) = \sum_{i=0}^{9} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \frac{x^{17}}{17!} - \frac{x^{19}}{19!}$$

i	$(-1)^i$	$2i+1$	x^{2i+1}	$(2i+1)!$	$(-1)^i \frac{x^{2i+1}}{(2i+1)!}$	$\sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$
0	+1	1	2.300000	1	2.300000	2.300000
1	-1	3	12.167000	6	-2.027833	0.272167
2	+1	5	64.363430	120	0.536362	0.808529
3	-1	7	340.482545	5040	-0.067556	0.740973
4	+1	9	1801.152661	362880	0.004963	0.745936
5	-1	11	9528.097579	39916800	-0.000239	0.745697
6	+1	13	50403.636194	6227020800	0.000008	0.745705
7	-1	15	266635.235464	1307674368000	-0.000000	0.745705
8	+1	17	1410500.395607	355687428096000	0.000000	0.745705
9	-1	19	7461547.092759	121645100408832000	-0.000000	0.745705

Solución de problemas usando algoritmos

Cálculo de la función seno.

Redefiniendo la serie es posible optimizar el cálculo:

$$\sin(x) = \sum_{i=0}^{9} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \frac{x^{17}}{17!} - \frac{x^{19}}{19!}$$

$$(-1)^0 = +1 = 1 - 2(0\%2) = 2\big((0+1)\%2\big) - 1$$

$$(-1)^1 = -1 = 1 - 2(1\%2) = 2\big((1+1)\%2\big) - 1$$

$$(-1)^2 = +1 = 1 - 2(2\%2) = 2\big((2+1)\%2\big) - 1$$

$$(-1)^3 = +1 = 1 - 2(3\%2) = 2\big((3+1)\%2\big) - 1$$

$$(-1)^i = 1 - 2(i\%2) = 2\big((i+1)\%2\big) - 1$$

Solución de problemas usando algoritmos

Cálculo de la función seno.

Redefiniendo la serie es posible optimizar el cálculo:

$$\sin(x) = \sum_{i=0}^{9} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \frac{x^{17}}{17!} - \frac{x^{19}}{19!}$$

$$\frac{x^3}{3!} = x \times \frac{x^2}{2 \times 3} = x \times \frac{x}{2} \times \frac{x}{3}$$

$$\frac{x^5}{5!} = \frac{x^3}{3!} \times \frac{x^2}{4 \times 5} = \frac{x^3}{3!} \times \frac{x}{4} \times \frac{x}{5}$$

$$\frac{x^7}{7!} = \frac{x^5}{5!} \times \frac{x^2}{6 \times 7} = \frac{x^5}{5!} \times \frac{x}{6} \times \frac{x}{7}$$

$$\frac{x^7}{7!} = \frac{x^5}{5!} \times \frac{x^2}{6 \times 7} = \frac{x^5}{5!} \times \frac{x}{6} \times \frac{x}{7}$$

$$\frac{x^{2i+1}}{(2i+1)!} = \frac{x^{2i-1}}{(2i-1)!} \times \frac{x^2}{(2i) \times (2i+1)} = \frac{x^{2i-1}}{(2i-1)!} \times \frac{x}{2i} \times \frac{x}{2i+1}$$

Solución de problemas usando algoritmos

Cálculo de la función seno.

Supongamos los valores de $x = 2.3$ para 10 términos:

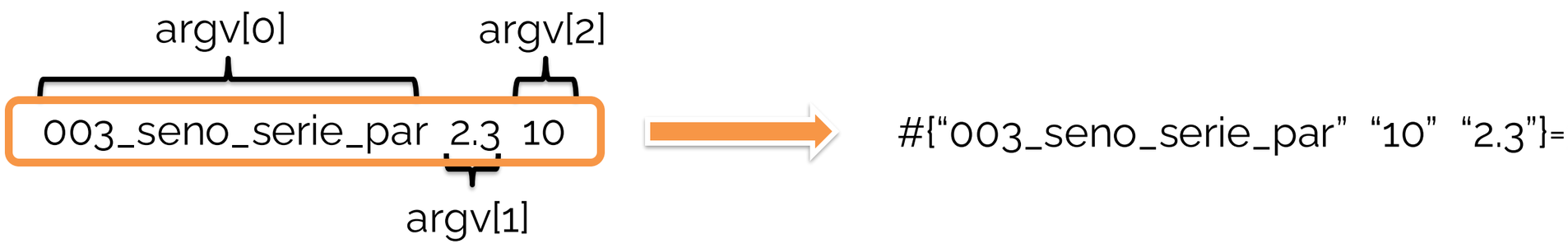
$$\sin(x) = \sum_{i=0}^{9} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \frac{x^{17}}{17!} - \frac{x^{19}}{19!}$$

i	$2((i+1)\%2)-1$	$2i$	$2i+1$	$\frac{x}{2i+2}$	$\frac{x}{2i+3}$	$\frac{x^{2i+1}}{(2i+1)!}$	$\sum_{i=0}^{n}(-1)^i\frac{x^{2i+1}}{(2i+1)!}$
0	+1	0	1	1.150000	0.766667	2.027833	2.300000
1	-1	2	3	0.575000	0.460000	0.536362	0.272167
2	+1	4	5	0.383333	0.328571	0.067556	0.808529
3	-1	6	7	0.287500	0.255556	0.004963	0.740973
4	+1	8	9	0.230000	0.209091	0.000239	0.745936
5	-1	10	11	0.191667	0.176923	0.000008	0.745697
6	+1	12	13	0.164286	0.153333	0.000000	0.745705
7	-1	14	15	0.143750	0.135294	0.000000	0.745705
8	+1	16	17	0.127778	0.121053	0.000000	0.745705
9	-1	18	19	0.115000	0.109524	0.000000	0.745705

Solución de problemas usando algoritmos

Cálculo de la función seno.

Supongamos los valores de $x = 2.3$ para 10 términos, la función *int main(int argc, char *argv[])*



argc = 3 // Número de argumentos incluyendo el nombre del programa, cada argumento es considerado por el espacio que existe entre ellos.

argv[0] = “003_seno_serie_par” // Los argumentos son cadenas de texto y siempre inician en el índice 0, cada argumento es un elemento en el arreglo argv

argv[1] = “2.3”

argv[2] = “10”

Solución de problemas usando algoritmos

Cálculo de la función seno.

Conversión de cadena a entero

Librería: stdlib.h (Librería de funciones estándar)

Prototipo: int atoi(const char *str)

Ejemplo:
```
int valor;
const char* str = “10”;
valor = atoi(str);
```

Prototipo: long int strtol(const char *nptr, char **endptr, int base)

Ejemplo:
```
long int valor;
const char* str = “10”;
char *finptr;
valor = strtol(str, &finptr, 10);
```

Solución de problemas usando algoritmos

Cálculo de la función seno.

Conversión de cadena a entero

str[]

"4826" → $\text{“0”} \leq \text{str[i]} \leq \text{”9”}$ → $\text{str}[0] \neq \text{” – ”} \therefore valor$

$\text{valor} = \text{str}[0] - \text{”0”} = \text{“4”} - \text{”0”} = 4$

$\text{valor} = 10 \times \text{valor} + (\text{str}[1] - \text{”0”}) = 10 \times 4 + (\text{“8”} - \text{”0”}) = 48$

$\text{valor} = 10 \times \text{valor} + (\text{str}[2] - \text{”0”}) = 10 \times 48 + (\text{“2”} - \text{”0”}) = 482$

$\text{valor} = 10 \times \text{valor} + (\text{str}[3] - \text{”0”}) = 10 \times 482 + (\text{“6”} - \text{”0”}) = 4826$

Solución de problemas usando algoritmos

Cálculo de la función seno.

Conversión de cadena a punto flotante

Librería: stdlib.h (Librería de funciones estándar)

Prototipo: double atof(const char *str):

Ejemplo:
```
double valor;
const char* str = “2.3”;
valor = atof(str);
```

Prototipo: long double strtold(const char *nptr, char **endptr)

Ejemplo:
```
long double valor;
const char* str = “2.3”;
char *finptr;
valor = strtold(str, &finptr);
```

Solución de problemas usando algoritmos

Cálculo de la función seno.

Conversión de cadena a punto flotante

str[]

"-12.4" → $\text{“0”} \leq \text{str[i]} \leq \text{”9”}$ → $\text{str}[0] = \text{”} - \text{”} \therefore -valor$

$\text{valor} = \text{str}[1] - \text{”0”} = \text{“1”} - \text{”0”} = 1$

$\text{valor} = 10 \times \text{valor} + (\text{str}[2] - \text{”2”}) = 10 \times 1 + (\text{“2”} - \text{”0”}) = 12$

$\text{str}[3] \equiv '.'$

$$\text{valor} = \text{valor} + \frac{\text{str}[4] - \text{”0”}}{10} = 12 + \frac{\text{”4”} - \text{”0”}}{10} = 12.4 \rightarrow -\text{valor} = -12.4$$

Solución de problemas usando algoritmos

Banknotes (Billetes).

Tienes una cantidad ilimitada de billetes por valor de A y B dólares ($A \neq B$). Sí desea pagar S dólares usando exactamente N billetes. Implementé una función que regresé la cantidad de billetes por valor de A necesarios. Si no tiene solución regresé el valor de -1.

Entrada estándar.
La función recibe cuatro números enteros A, B, S y N.

Salida estándar.
La función regresa un número entero con la cantidad de billetes por valor de A

Restricciones y notas
- $1 \leq A, B, S, N \leq 10^9$
- $A \neq B$

Entrada	Salida
1 2 7 5	3
2 1 7 5	2
2 1 4 5	-1
2 3 20 8	4

Solución de problemas usando algoritmos

Banknotes (Billetes).

Supongamos que queremos encontrar X cantidad de billetes de valor A y Y cantidad de billetes de valor B

$$X + Y = N$$
$$XA + YB = S$$

Despegando Y de la primera ecuación

$$Y = N - X$$

Sustituyendo en la segunda ecuación

$$XA + (N - X)B = S$$

Ordenando respecto de X

$$X(A - B) = S - NB$$

Despejando X

$$X = \frac{S - NB}{A - B}$$

Es necesario tomar en consideración las siguientes restricciones

$$\frac{S - NB}{A - B} \equiv \left\lfloor \frac{S - NB}{A - B} \right\rfloor$$

$$0 \leq X \leq N$$

Solución de problemas usando algoritmos

Banknotes (Billetes).

Usando la formula y considerando las restricciones

$$X = \frac{S - NB}{A - B}$$

Entrada	Salida
1 2 7 5	3
2 1 7 5	2
2 1 4 5	-1
2 3 20 8	4

$$A = 1, B = 2, S = 7, N = 5, X = \frac{7 - 5 \times 2}{1 - 2} = \frac{-3}{-1} = 3$$

$$A = 2, B = 1, S = 7, N = 5, X = \frac{7 - 5 \times 1}{2 - 1} = \frac{2}{1} = 2$$

$$A = 2, B = 1, S = 4, N = 5, X = \frac{4 - 5 \times 1}{2 - 1} = \frac{-1}{1} = -1$$

$$A = 2, B = 3, S = 20, N = 8, X = \frac{20 - 8 \times 3}{2 - 3} = \frac{-4}{-1} = 4$$

Solución de problemas usando algoritmos

Cuatro X-tremes.

Implementé una función que tomé como argumentos cuatro enteros. Devuelva la diferencia máxima entre dos números enteros cualesquiera.

Entrada estándar.
La función debe recibir en la primera línea cuatro números enteros.

Salida estándar.
La función regresa un número que corresponde a la máxima diferencia entre los cuatro números.

Restricciones y notas
- El resultado es un entero con signo de 32 bits.

Entrada	Salida
2 1 5 3	4
-10 4 -9 -5	14
100 10 10 10	90
-509916144 -285442181 688434324 -869739739	1558174063

Solución de problemas usando algoritmos

Cuatro X-tremes.

Intercambio de variables enteras (swap)

Algoritmo de suma-resta

instrucción	a	b
a=40;	40	
b=-12;	40	-12
a+=b;	28	-12
b=a-b;	28	40
a-=b;	-12	40

```
#define SWAP(a, b) {\
a+=b;\
b=a-b;\
a-=b;\
}
```

Algoritmo de XOR

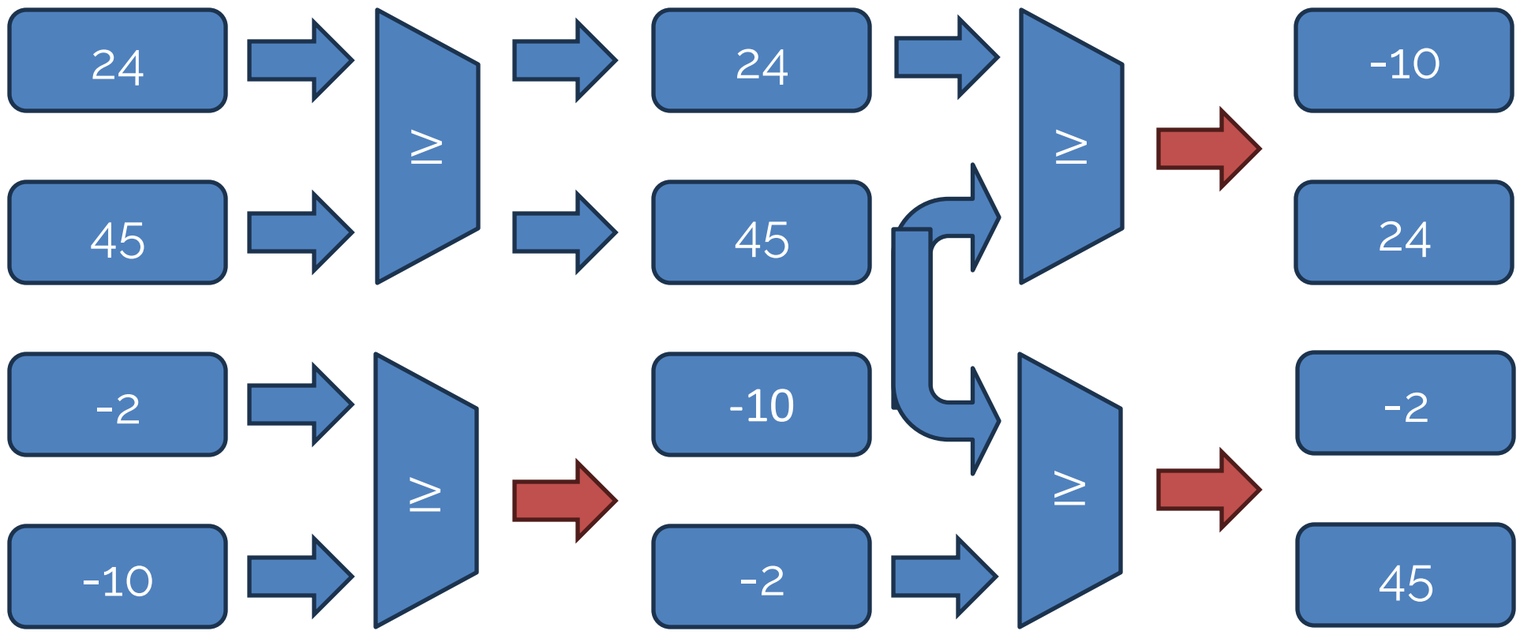
instrucción	a	b
a=40;	40_{10}=0010 1000_2	
b=12;	40_{10}=0010 1000_2	-12_{10}=1111 0100_2
a^=b;	-36_{10}=1101 1100_2	-12_{10}=1111 0100_2
b^=a;	-36_{10}=1101 1100_2	40_{10}=0010 1000_2
a^=b;	-12_{10}=1111 0100_2	40_{10}=0010 1000_2

```
#define SWAP(a, b) {\
a^=b;\
b^=a;\
a^=b;\
}
```

Solución de problemas usando algoritmos

Cuatro X-tremes.

Comparación




```
long int a, b, c, d;

if(a>b)
        SWAP(a,b);
if(c>d)
        SWAP(c,d);
if(a>c)
        SWAP(a,c);
if(b>d)
        SWAP(b,d);
return d-a;
```

Solución de problemas usando algoritmos

Cuatro X-tremes.

Instrucción	a	b	c	d
	2	1	5	3
if(a>b) SWAP(a,b);	1	2	5	3
if(c>d) SWAP(c,d);	1	2	3	5
if(a>c) SWAP(a,c);	1	2	3	5
if(b>d) SWAP(b,d);	1	2	3	5
return d-a;	5-1=4			

Instrucción	a	b	c	d
	-10	4	-9	-5
if(a>b) SWAP(a,b);	-10	4	-9	-5
if(c>d) SWAP(c,d);	-10	4	-9	-5
if(a>c) SWAP(a,c);	-10	4	-9	-5
if(b>d) SWAP(b,d);	-10	-5	-9	4
return d-a;	4-(-10)=14			

Solución de problemas usando algoritmos

Cuatro X-tremes.

Instrucción	a	b	c	d
	100	10	10	10
if(a>b) SWAP(a,b);	10	100	10	10
if(c>d) SWAP(c,d);	10	100	10	10
if(a>c) SWAP(a,c);	10	100	10	10
if(b>d) SWAP(b,d);	10	10	10	100
return d-a;	100-10=90			

Solución de problemas usando algoritmos

Cuatro X-tremes.

Instrucción	a	b	c	d
	-509,916,144	-285,442,181	688,434,324	-869,739,739
if(a>b) SWAP(a,b);	-509,916,144	-285,442,181	688,434,324	-869,739,739
if(c>d) SWAP(c,d);	-509,916,144	-285,442,181	-869,739,739	688,434,324
if(a>c) SWAP(a,c);	-869,739,739	-285,442,181	-509,916,144	688,434,324
if(b>d) SWAP(b,d);	-869,739,739	-285,442,181	-509,916,144	688,434,324
return d-a;	688,434,324-(-869,739,739)=1,558,174,063			

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema del cruce del puente.

Cuatro personas deben cruzar un puente de noche y está demasiado oscuro para cruzar sin una linterna. Solo hay una linterna disponible. El puente solo admite a dos personas a la vez. Cada persona tarda un tiempo diferente en cruzar: 1 minuto, 2 minutos, 7 minutos y 10 minutos. ¿Cuál es el tiempo más corto posible para que las cuatro personas crucen el puente?

A: 1 minuto B: 2 minutos C: 7 minutos D: 10 minutos

1. A y B cruzan juntos: 2 minutos
2. A regresa: 1 minuto
3. C y D cruzan juntos: 10 minutos
4. B regresa: 2 minutos
5. A y B cruzan juntos: 2 minutos

Tiempo total: 2+1+10+2+2 = 17 minutos = A+3B+D = 1+3(2)+10

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema del cruce del puente.

Objetivo: Cuatro personas necesitan cruzar un puente de noche con una linterna. Solo una linterna está disponible y el puente solo puede soportar a dos personas al mismo tiempo. Cada persona tiene una velocidad diferente de cruce y dos personas que cruzan juntas deben ir a la velocidad de la más lenta.

Condiciones:

- Solo una linterna.
- El puente solo soporta dos personas a la vez.
- Diferentes tiempos de cruce para cada persona.

Meta: Determinar la mínima cantidad de tiempo para que todos crucen el puente.

Estrategia Óptima

La solución del problema implica enviar a las personas más rápidas de ida y vuelta para minimizar el tiempo total en el que las personas más lentas cruzan el puente.

1. Las dos personas más rápidas cruzan primero.
2. La persona más rápida regresa con la linterna.
3. Las dos personas más lentas cruzan juntas.
4. La persona más rápida que queda en el lado inicial regresa con la linterna.
5. Las dos personas más rápidas cruzan juntas de nuevo.

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

Descripción

Dado un conjunto de elementos, cada uno con un peso y un valor, determinar la cantidad de cada elemento que se puede incluir en una colección de manera que el peso total sea menor o igual a una capacidad dada y el valor total sea lo más grande posible.

Formulación

- Entrada
  - Una lista de n elementos, donde cada elemento i tiene un peso w_i y un valor v_i.
  - Una capacidad máxima de la mochila W.
- Salida
  - Un subconjunto de los elementos que maximiza el valor total sin exceder la capacidad W.

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

Supongamos que tienes los siguientes elementos:

Elemento	Peso (kg)	Valor ($)
1	2	3
2	3	4
3	4	5
4	5	6

La capacidad máxima de la mochila es 5 kg. Determine el valor máximo posible para la mochila.

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

Supongamos que tienes los siguientes elementos:

Pesos = [2, 3, 4, 5]
Valores = [3, 4, 5, 6]
Capacidad = 5

La estrategia de búsqueda exhaustiva implica evaluar todas las combinaciones posibles de los elementos para encontrar la que maximiza el valor total sin exceder la capacidad de la mochila.

Algoritmo

1. Generar todas las combinaciones posibles de elementos.
2. Para cada combinación, calcular el peso total y el valor total.
3. Si el peso total de una combinación es menor o igual a la capacidad de la mochila y su valor total es mayor que el valor máximo encontrado hasta ahora, actualizar el valor máximo.
4. La combinación con el valor máximo es la solución óptima.

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

Supongamos que tienes los siguientes elementos:

Pesos = [2, 3, 4, 5]

Valores = [3, 4, 5, 6]

Capacidad = 5

n = #{Pesos} = 4

Número de combinaciones = 2^n=16

Combinación		Peso	Valor	Combinación		Peso	Valor
[]	[]	0	0	[5]	[6]	5	6
[2]	[3]	2	3	[2, 5]	[3, 6]	7	9
[3]	[4]	3	4	[3, 5]	[4, 6]	8	10
[2, 3]	[3, 4]	5	7	[2, 3, 5]	[3, 4, 6]	10	13
[4]	[5]	4	5	[4, 5]	[5, 6]	9	11
[2, 4]	[3, 5]	6	8	[2, 4, 5]	[3, 5, 6]	11	14
[3, 4]	[4, 5]	7	9	[3, 4, 5]	[4, 5, 6]	12	15
[2, 3, 4]	[3, 4, 5]	9	12	[2, 3, 4, 5]	[3, 4, 5, 6]	14	18

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

```
function mochila(pesos, valores, capacidad):
    n = length(pesos)
    max_valor = 0
    mejor_comb = []

    for i from 0 to 2^n - 1:  # 2^n combinaciones
    comb = []
    peso_total = 0
    valor_total = 0

    for j from 0 to n - 1:
        if i & (1 << j) != 0:  # si jth bit en i está en alto
            combination.append(j)
            peso_total += pesos[j]
            valor_total += valores[j]

    if peso_total <= capacidad and valor_total > max_valor:
        max_valor = valor_total
        mejor_comb = comb

return max_valor, mejor_comb
```

Estrategias para construir algoritmos

Algoritmos de búsqueda exhaustiva y fuerza bruta

Problema de la mochila.

Banco de pruebas:

Prueba	Pesos (Kg)	Valores ($)	Capacidad (Kg)	Valor máximo ($)
1	[2, 3, 4, 5]	[3, 4, 5, 6]	5	7
2	[1, 2, 3, 8, 7, 4]	[20, 5, 10, 40, 15, 25]	10	60
3	[4, 2, 3, 1]	[10, 4, 7, 2]	5	11
4	[5, 3, 4, 2]	[6, 4, 5, 3]	6	8
5	[10, 20, 30]	[60, 100, 120]	50	220

Estrategias para construir algoritmos

Estrategia Top-Down

Concepto
La estrategia Top-Down, también conocida como descomposición descendente o memorización en programación dinámica, es una técnica de diseño de algoritmos que descompone un problema principal en subproblemas más pequeños y manejables. El enfoque es recursivo y se resuelven los subproblemas de manera que sus resultados se combinan para resolver el problema original.

Características

Recursión: La solución se construye de manera recursiva, descomponiendo el problema en subproblemas más pequeños.

Memorización: Para optimizar los recursos de cómputo se reutilizan los resultados de subproblemas ya resueltos, sus resultados se almacenan en una estructura de datos (como una tabla o matriz). Esto mejora la eficiencia del algoritmo al reducir el número de cálculos redundantes.

Descomposición de Problemas: Cada subproblema es similar al problema original, pero de menor tamaño, y se resuelve de la misma manera.

Estrategias para construir algoritmos

Estrategia Top-Down

Ventajas

- *Claridad y Simplicidad:* La recursión a menudo hace que el algoritmo sea más fácil de entender y escribir.

- *Evita Cálculos Redundantes:* Almacena los resultados de subproblemas ya resueltos, optimizando los recursos computacionales y evitar la redundancia en calculo.

Desventajas

- *Consumo de Memoria:* La memorización puede requerir una cantidad significativa de memoria para almacenar los resultados de los subproblemas.

- *Profundidad de Recursión:* En algunos lenguajes, una gran cantidad de llamadas recursivas puede causar un desbordamiento de la pila.

Estrategias para construir algoritmos

Estrategia Bottom-Up

Concepto
La estrategia Bottom-Up, también conocida como tabulación en programación dinámica, es una técnica de diseño de algoritmos que resuelve primero los subproblemas más pequeños y utiliza sus resultados para resolver subproblemas más grandes. El enfoque es iterativo y comienza resolviendo los casos más simples, construyendo gradualmente la solución del problema principal.

Características

Iteración: En lugar de recursión, se utiliza una estructura iterativa para resolver los subproblemas en un orden específico.

Tabulación: Los resultados de los subproblemas se almacenan en una tabla (como una matriz) y se utilizan para resolver subproblemas más grandes.

Construcción de Soluciones: Se comienza con los casos base y se construyen soluciones a problemas más grandes utilizando los resultados almacenados en la tabla.

Estrategias para construir algoritmos

Fibonacci

La serie de Fibonacci es una secuencia infinita de números enteros en la que cada número después de los dos primeros es la suma de los dos números anteriores. Es una de las secuencias más famosas en matemáticas y aparece en muchos contextos, desde la teoría de números hasta fenómenos naturales como la disposición de las hojas en plantas o la reproducción de conejos.

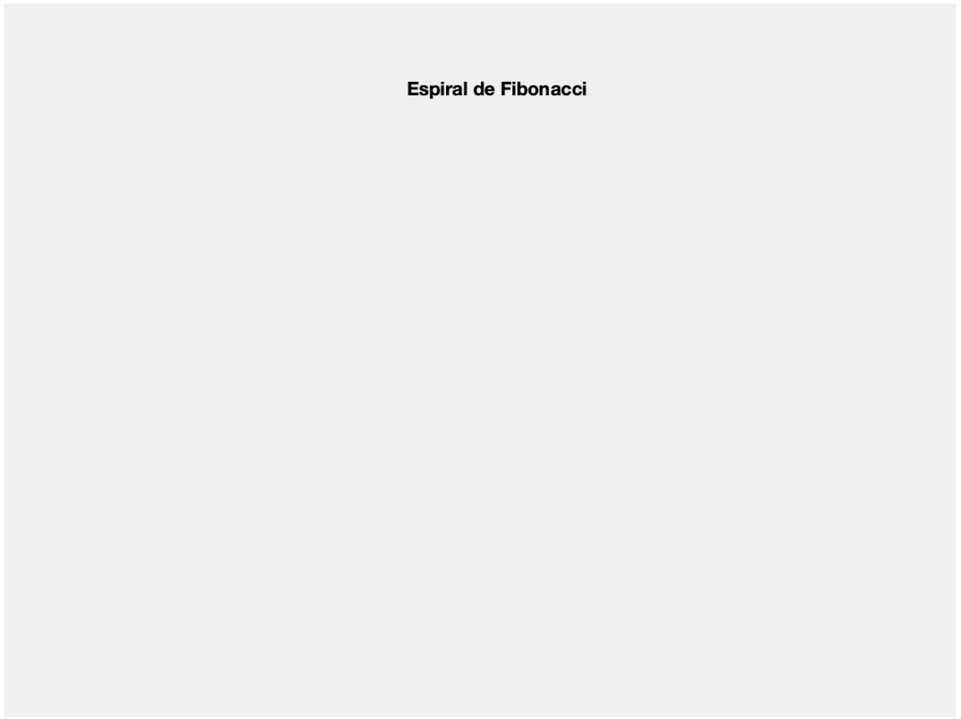
Definición Matemática
La serie de Fibonacci se define de la siguiente manera:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2) \qquad para\ n \geq 2$

Ejemplo de los Primeros Números de la Serie de Fibonacci
A continuación, se muestran los primeros números de la serie:

$$0,1,1,2,3,5,8,13,21,34,55,89,144,233,377$$

- $F(0) = 0$
- $F(1) = 1$
- $F(2) = F(1) + F(0) = 1 + 0 = 1$
- $F(3) = F(2) + F(1) = 1 + 1 = 2$
- $F(4) = F(3) + F(2) = 2 + 1 = 3$
- $F(5) = F(4) + F(3) = 3 + 2 = 5$
- $\vdots$



Estrategias para construir algoritmos

Estrategia Top-Down: Fibonacci

En la estrategia Top-Down, descomponemos el problema en subproblemas más pequeños y almacenamos los resultados de estos subproblemas para evitar recalcularlos.

```
function Fibonacci_TD(n, memo)
    if n <= 1 then
        return n
    end if

    if memo[n] is not defined then
        memo[n] = Fibonacci_TD (n-1, memo) + Fibonacci_TD (n-2, memo)
    end if

    return memo[n]
end function

function Main
    n = 10  // Ejemplo: Calcular el 10° número de Fibonacci
    memo = array of size (n+1) initialized with undefined values
    result = Fibonacci_TD (n, memo)
    print "Fibonacci(", n, ") = ", result
end function
```

Estrategias para construir algoritmos

Estrategia Bottom-Up: Fibonacci

En la estrategia Bottom-Up, resolvemos los subproblemas más pequeños primero y utilizamos sus resultados para construir la solución de subproblemas más grandes de manera iterativa.

```
function Fibonacci_BU(n)
    if n <= 1 then
        return n
    end if
    // Crear una tabla para almacenar los valores de Fibonacci
    dp = array of size (n+1)
    // Inicializar los primeros valores
    dp[0] = 0
    dp[1] = 1
    // Llenar la tabla usando los valores anteriores
    for i = 2 to n do
        dp[i] = dp[i-1] + dp[i-2]
    end for

    return dp[n]
end function

function Main
    n = 10  // Ejemplo: Calcular el 10° número de Fibonacci
    result = Fibonacci_BU (n)
    print "Fibonacci(", n, ") = ", result
end function
```

Estrategias para construir algoritmos

Conclusión

Top-Down

- Divide el problema principal en subproblemas recursivamente.
- Usa memorización para almacenar resultados de subproblemas ya resueltos.
- Facilita la comprensión y diseño del algoritmo, pero puede consumir más memoria y ser susceptible a desbordamientos de pila..

Bottom-Up

- Resuelve los subproblemas más pequeños primero de manera iterativa.
- Usa tabulación para almacenar resultados de subproblemas.
- Es más eficiente en memoria y evita problemas de recursión profunda, pero puede ser más complejo de implementar..