



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Algoritmos y estructuras de datos
Tema 2:
ANÁLISIS Y DISEÑO DE ALGORITMOS

2 Análisis y diseño de algoritmos

Objetivo: Aplicar diversas técnicas para el análisis y el diseño de algoritmos orientados a la solución de problemas computacionales.

2 Análisis y diseño de algoritmos

2.1 Fundamentos de algorítmica.

2.2 Algorítmica básica.

2.3 Complejidad.

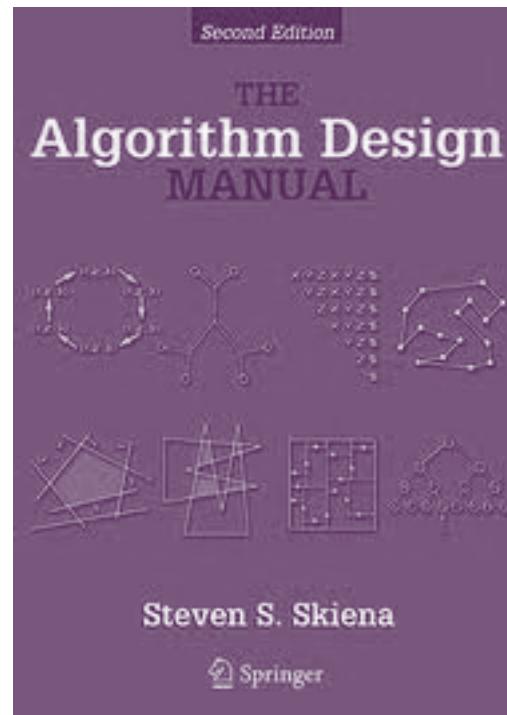
2.4 Análisis de algoritmos.

2.5 Estrategias para la construcción de algoritmos.

2.6 Definición, ejemplos, diseño, implementación, corrección, eficiencia y complejidad de algoritmos.

2.7 Análisis y diseño avanzado de algoritmos.

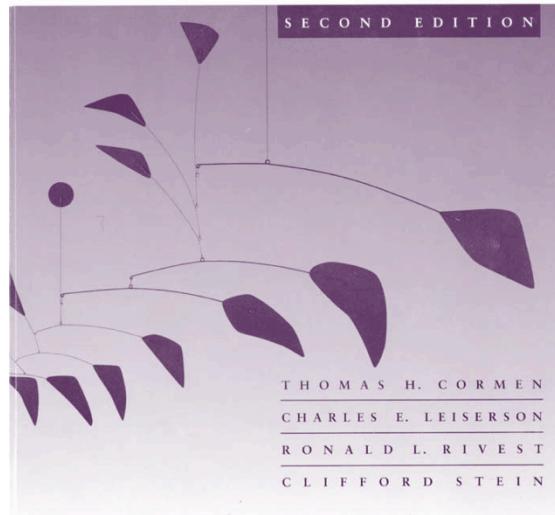
Bibliografía



The algorithm design manual.
Steven S. Skiena, Springer.

Bibliografía

INTRODUCTION TO ALGORITHMS



Introduction to Algorithms.

***Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein, McGraw-Hill.***



2.1 Fundamentos de algorítmica.

2.1 Fundamentos de algorítmica.

El diseño de algoritmos correctos, eficientes y aplicables a problemas cotidianos requiere el acceso a dos cuerpos distintos de conocimiento: las técnicas y los recursos.

Técnicas

Un buen diseñador de algoritmos conoce diversas técnicas fundamentales del diseño de algoritmo, como lo son las estructuras de datos, la programación dinámica, la búsqueda profunda, la marcha atrás y las técnicas heurísticas.

Sin embargo, la técnica más importante en el diseño de algoritmos es el modelado, que es el arte de abstraer una aplicación confusa del mundo real en un problema adaptable para atacarlo por un algoritmo.

Recursos

Un buen diseñador de algoritmos se apoya en los hombros de otros diseñadores. En lugar de reimplementar algoritmos populares, buscan implementaciones existentes para usarlas como punto de partida.

Un diseñador de algoritmos está familiarizado con diversos problemas algorítmicos clásicos, lo que les permite tener suficiente material inicial para modelar cualquier aplicación más robusta.

The greedy algorithm used to give change.

Amount owed: 41 cents.

Subtract Quarter

$$41 - 25 = 16$$



Subtract Dime

$$16 - 10 = 6$$



Subtract Nickel

$$6 - 5 = 1$$



Subtract Penny

$$1 - 1 = 0$$



2.2 Algorítmica básica.

2.2 Algorítmica básica.

Un algoritmo es un método de solución para una instancia dada de un cierto problema, expresada en un lenguaje específico, y que permite realizar una tarea en general, es decir, un conjunto de pasos, procedimientos o acciones que permiten alcanzar un resultado o resolver un problema.

Un algoritmo es la parte más importante y durable de las ciencias de la computación debido a que éste puede ser creado de manera independiente tanto del lenguaje como de las características físicas del equipo que lo va a ejecutar.

Lo que hace especial a un algoritmo es que sus reglas pueden ser aplicadas un número ilimitado de veces sobre una situación particular y siempre se obtiene un resultado acorde a la misma.

Las principales características con las que debe cumplir un algoritmo son:

- **Preciso:** llegar a la solución en el menor tiempo posible y sin caer en ambigüedades.
- **Determinista:** a partir de un conjunto de datos idénticos de entrada, debe arrojar siempre los mismos resultados a la salida.
- **Finito:** Un proceso computable implica que, en algún momento, éste va a finalizar. Un algoritmo terminar en algún momento.

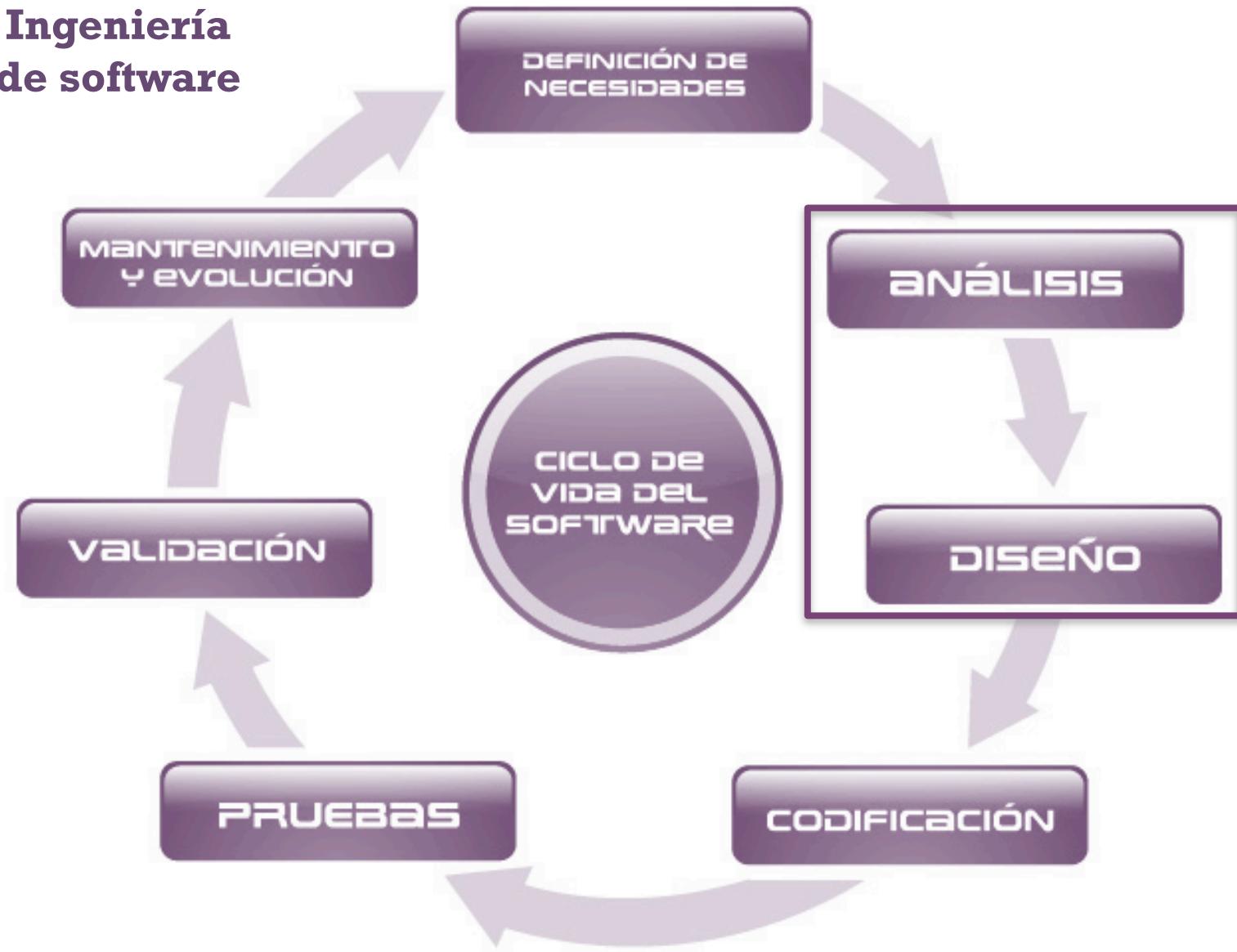
2.2.1 Algoritmos y Programas

Existen 3 propiedades deseables en un algoritmo. Un buen algoritmo debe ser correcto (cumplir con el objetivo) y eficiente (realizarlo en el menor tiempo posible), además de ser fácil de implementar.

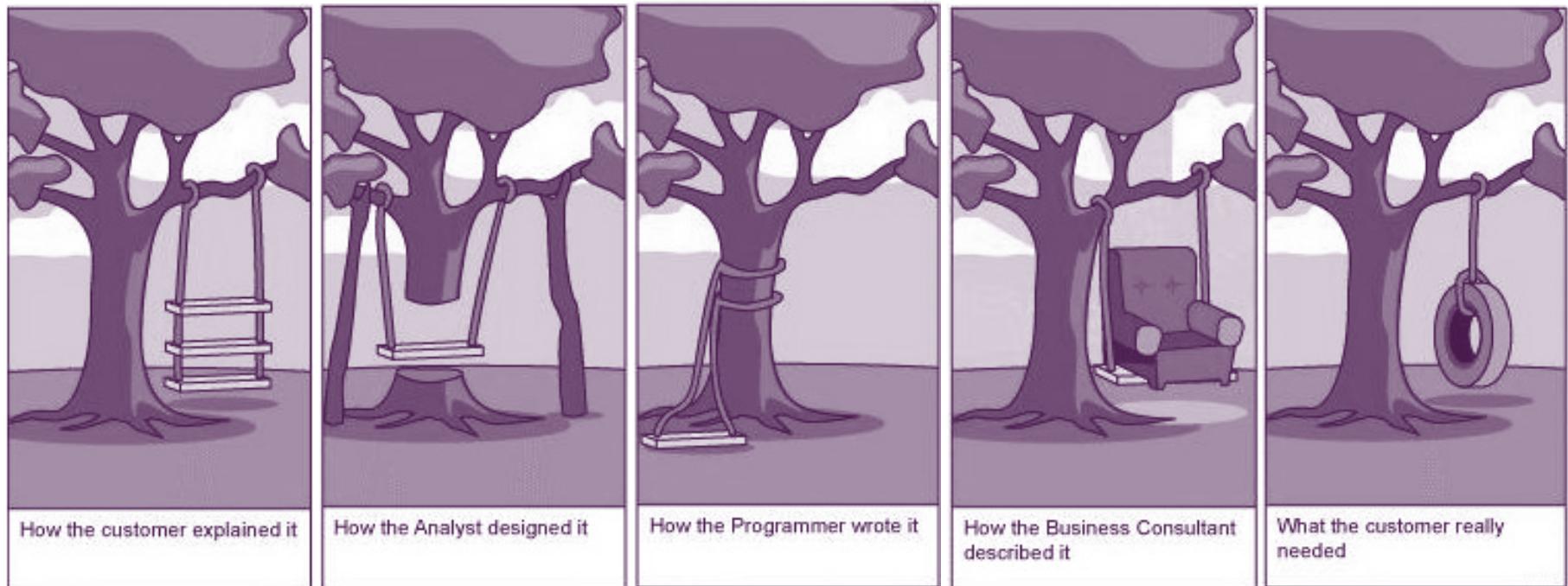
El enfoque con el que se construye un algoritmo es el método científico, que sigue los siguientes pasos:

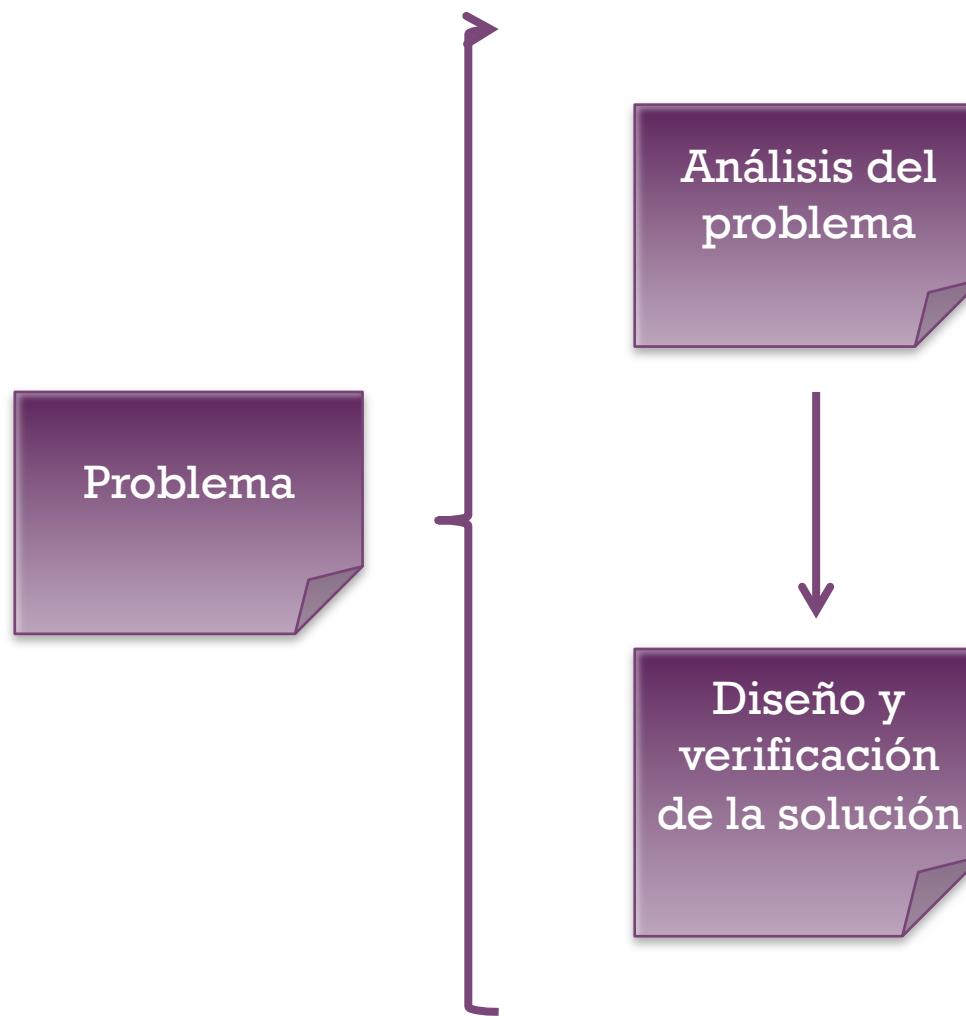
- **Observación: algún problema del real.**
- **Hipótesis: un modelo consistente con la observación.**
- **Predicción de eventos: basados en la hipótesis.**
- **Verificación: comprobar la hipótesis ahondando en las predicciones.**
- **Validación: repetir los eventos hasta que la hipótesis concuerden.**

Ingeniería de software



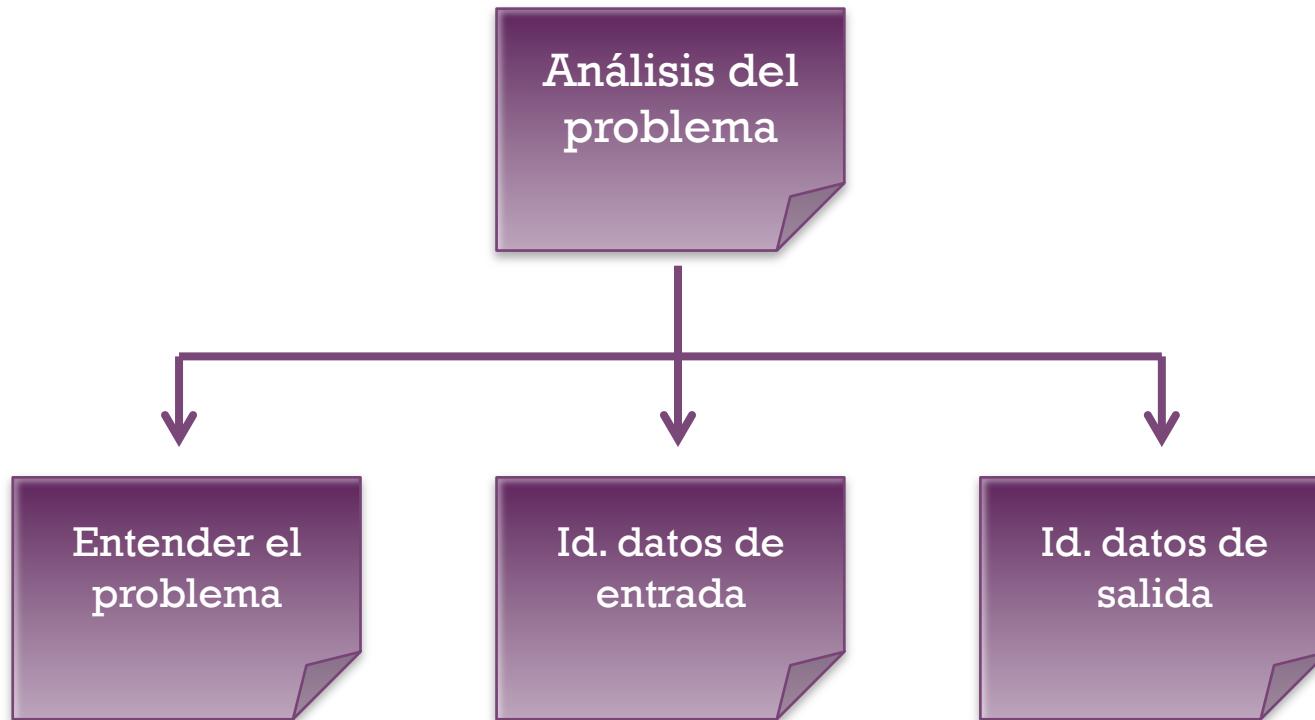
Dado un problema a resolver...



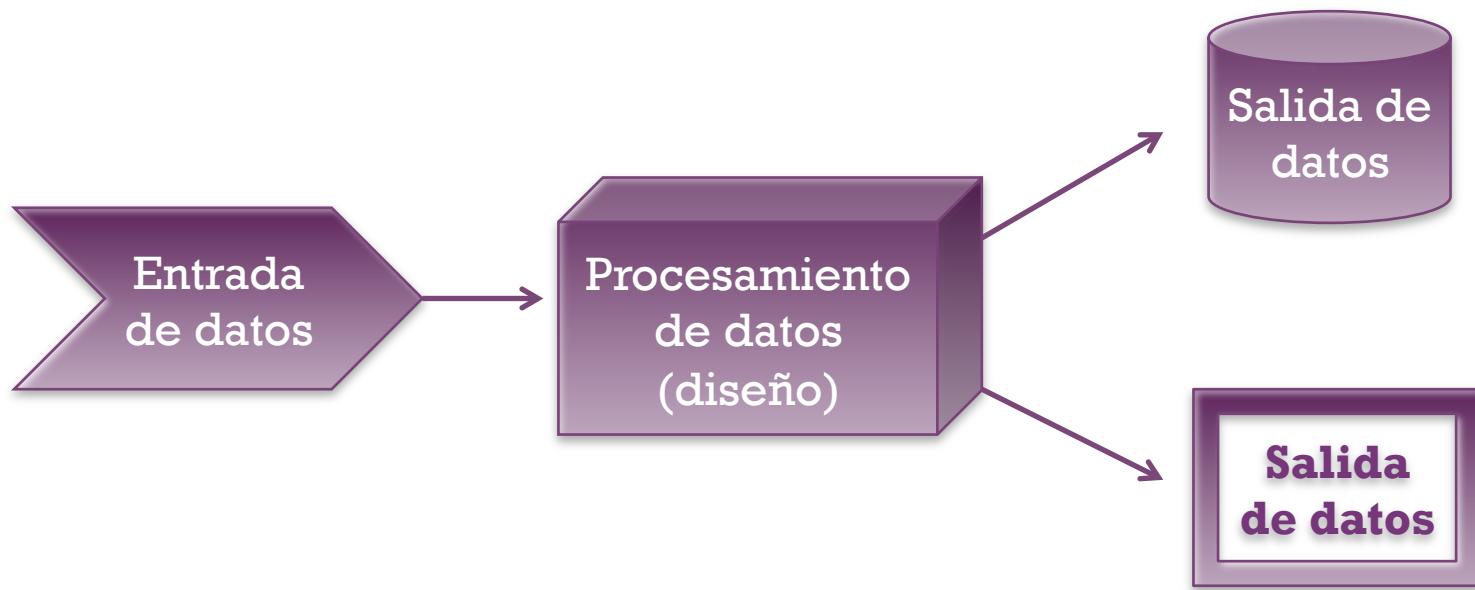


Problema

En general, se puede definir como problema al conjunto de instancias al cual corresponde un conjunto de soluciones, junto con una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío).



Durante el análisis hay que identificar 3 módulos básicos: módulo de entrada, módulo de procesamiento y módulo de salida.





Entrada
de datos

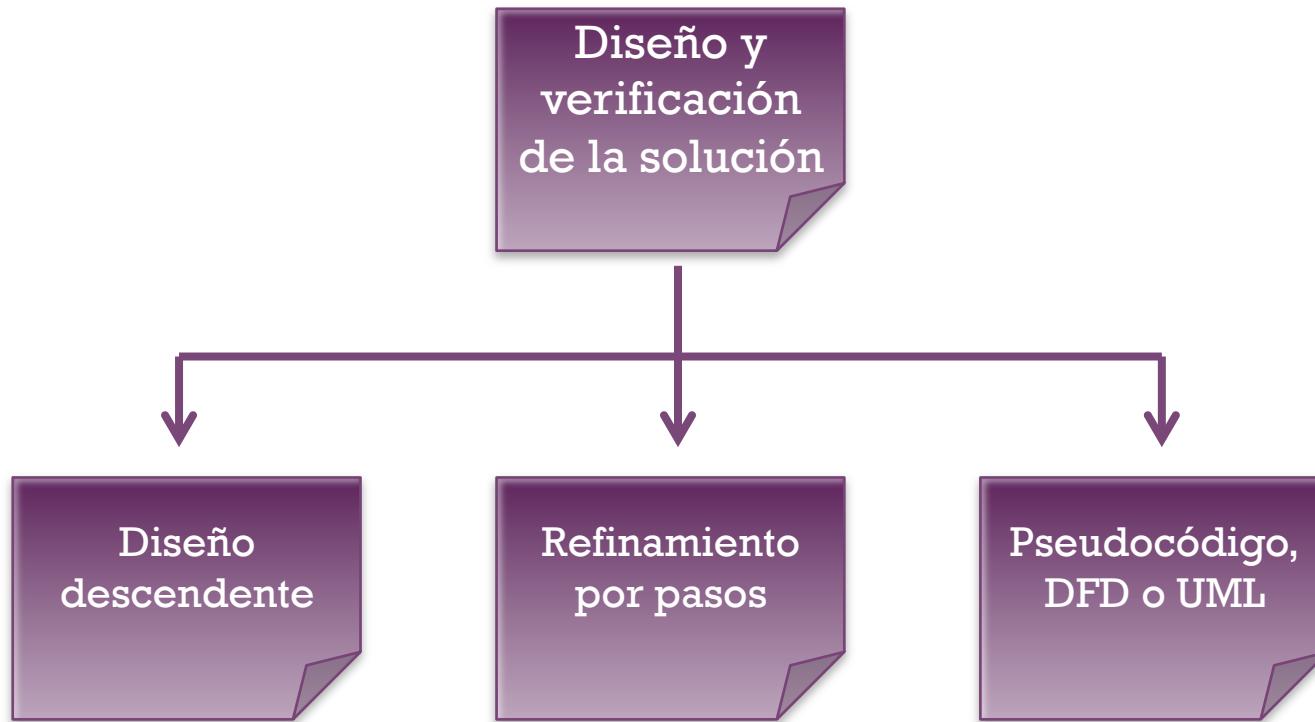
Un algoritmo se define por su conjunto E de entradas que representan las instancias del problema, es decir, el módulo de entrada de datos representa los tipos de datos alimentan al problema.

Procesamiento de datos

El módulo de procesamiento representa las operaciones necesarias para obtener un resultado a partir de los datos de entrada (diseño de la solución).



Un algoritmo posee un conjunto S de salidas que son los posibles resultados que emanen de la ejecución del algoritmo, es decir, el módulo de salida permite definir los tipos y valores que se deseean obtener al final del proceso.



El diseño descendente hace referencia a la división del problema en problemas más sencillos de tal manera que el conjunto de soluciones forme la solución general. Este diseño se conoce como divide y vencerás.

El refinamiento por pasos se refiere a la comprobación de cada uno de los módulos en los que se dividió el problema en el diseño descendente.

Algoritmo para crear un algoritmo

PROGRAMA desarrollar_algoritmo (problema)

correcto: BOOLEAN;

correcto := FALSO;

MIENTRAS (NO correcto) O (NO rápido(tiempo_ejec))

algoritmo = idear_algoritmo(problema)

correcto = analizar_exactitud(algoritmo)

tiempo_ejec= analizar_eficiencia(algoritmo)

FIN_MIENTRAS

RETURN algoritmo

FIN_PROGRAMA

Ejemplo 1

PROBLEMA: Determinar si un número dado es par o impar.

SOLUCIÓN:

1. **INICIO.**
2. **X,M: ENTERO**
3. **X := 0.**
4. **HACER**
5. **ESCRIBIR “Ingrese número entero.”**
6. **LEER X.**
7. **MIENTRAS X <> 0**
8. **M := X%2**
9. **SI M es igual a 0.**
10. **ESCRIBIR “X es par”**
11. **DE LO CONTRARIO**
12. **ESCRIBIR “X es impar”.**
11. **FIN.**

Ejemplo 1

Prueba de escritorio

iteración	X	M	Imprime
0	5	1	X es impar

iteración	X	M	Imprime
0	4	0	X es par

iteración	X	M	Imprime
0	-3	1	X es impar

iteración	X	M	Imprime
0	-10	0	X es par

Ejemplo 1**Prueba de escritorio**

iteración	X	M	Imprime
0	0	-	-
1	0	-	-
2	13	1	X es impar

Ejemplo 2

Realizar un programa que multiplique dos números (equis y ye) sin utilizar el operador multiplicación (*), es decir, no se puede especificar en el algoritmo la operación:

equis * ye

La solución se debe proporcionar en pseudocódigo. Al final se tiene que verificar el algoritmo (pruebas de escritorio).

Ejemplo 2

Solución:

1. **INICIO**
2. **X, Y, Z: ENTERO**
3. **HACER Z := 0**
4. **ESCRIBIR “Ingrese un número entero.”**
5. **LEER X**
6. **ESCRIBIR “Ingrese otro número entero.”**
7. **LEER Y**
8. **MIENTRAS X > 0**
9. **HACER Z = Z + Y**
10. **HACER X = X - 1**
11. **FIN_MIENTRAS**
12. **ESCRIBIR “El resultado es: “ Z**
13. **FIN**

Ejemplo 2**Prueba de escritorio**

iteración	X	Y	Z
0	5	7	7
1	4	7	14
2	3	7	21
3	2	7	28
4	1	7	35
5	0	7	35

El algoritmo anterior cumple con el objetivo del problema planteado, empero se puede eficientar el tiempo de ejecución.

Debido a la propiedad de la multiplicación de que “el orden de los factores no altera el producto”, una vez que ya se hayan recibido ambos valores (X y Y), se comprueba cuál es el menor y se le asigna a X . Debido a que X determina cuantas veces se va a realizar el ciclo, entre más pequeña sea X menos iteraciones se realizarán.

Ejemplo 2

Solución:

1. INICIO
2. X, Y, Z, TMP: ENTERO
3. HACER Z = 0
4. ESCRIBIR “Ingrese un número entero.”
5. LEER X
6. ESCRIBIR “Ingrese otro número entero.”
7. LEER Y
8. Si Y < X
 - 9. HACER TMP = X
 - 10. HACER X = Y
 - 11. HACER Y = TMP
12. TERMINA_SI
13. MIENTRAS X > 0
 - 14. HACER Z = Z + Y
 - 15. HACER X = X - 1
16. FIN_MIENTRAS
17. ESCRIBIR “El resultado es:” Z
18. FIN

El diseño de un algoritmo puede llegar a ser tan exquisito o refinado como se desee o experiencia se tenga.

Existe un algoritmo más eficiente para resolver una multiplicación en menos pasos.

Para entender el siguiente ejemplo es importante comprender las operaciones que realizan los operandos `>>` y `<<`, esto es, corrimientos de bits tanto a la izquierda como a la derecha.

Ejemplo 2

Solución mejorada:

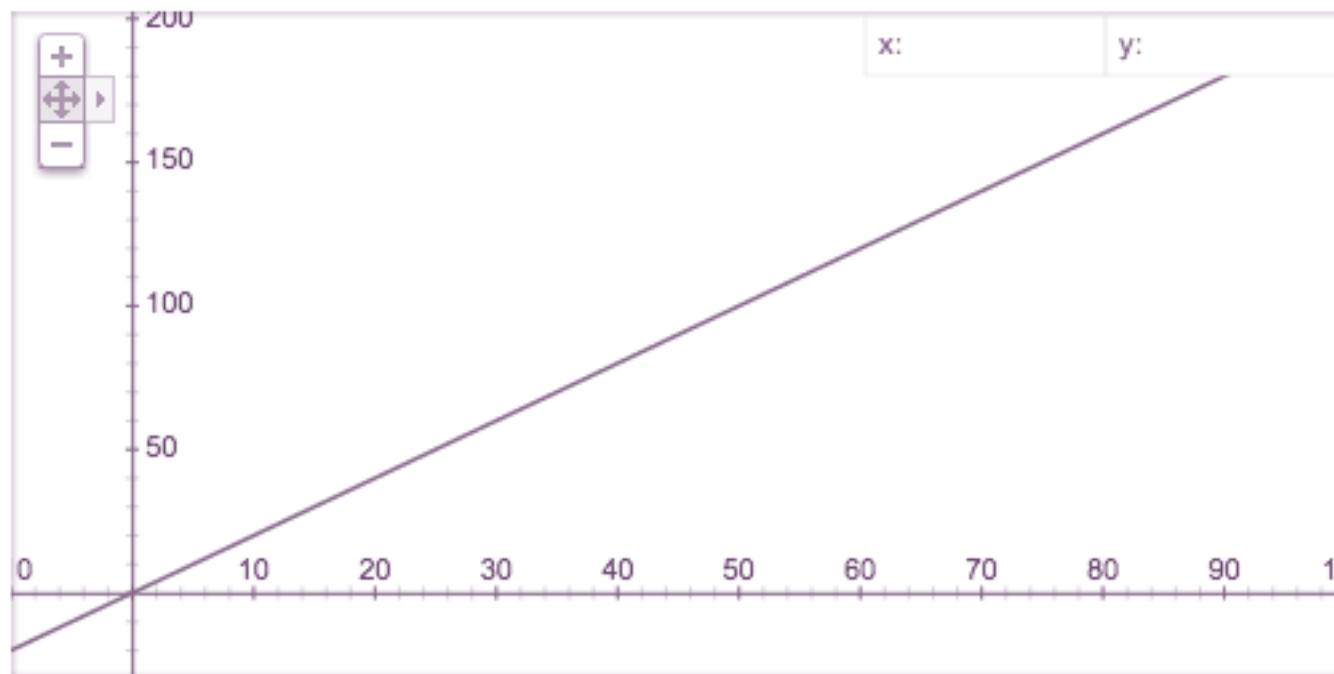
1. **INICIO**
2. **X, Y y Z: ENTERO**
3. **HACER Z = 0**
4. **ESCRIBIR “Ingrese un número entero.”**
5. **LEER X.**
6. **ESCRIBIR “Ingrese otro número entero.”**
7. **LEER Y.**
8. **MIENTRAS X > 0**
9. **Si (X % 2) = 1**
10. **HACER Z = Z + Y**
11. **HACER Y << 1**
12. **HACER X >> 1**
11. **FIN_MIENTRAS**
12. **ESCRIBIR “El resultado de la operación es: “ Z**
13. **FIN**

Ejemplo 2**Prueba de escritorio**

iteración	X	Y	Z
0	14	11	0
1	7	22	0
2	3	44	22
3	1	88	66
4	0	176	154

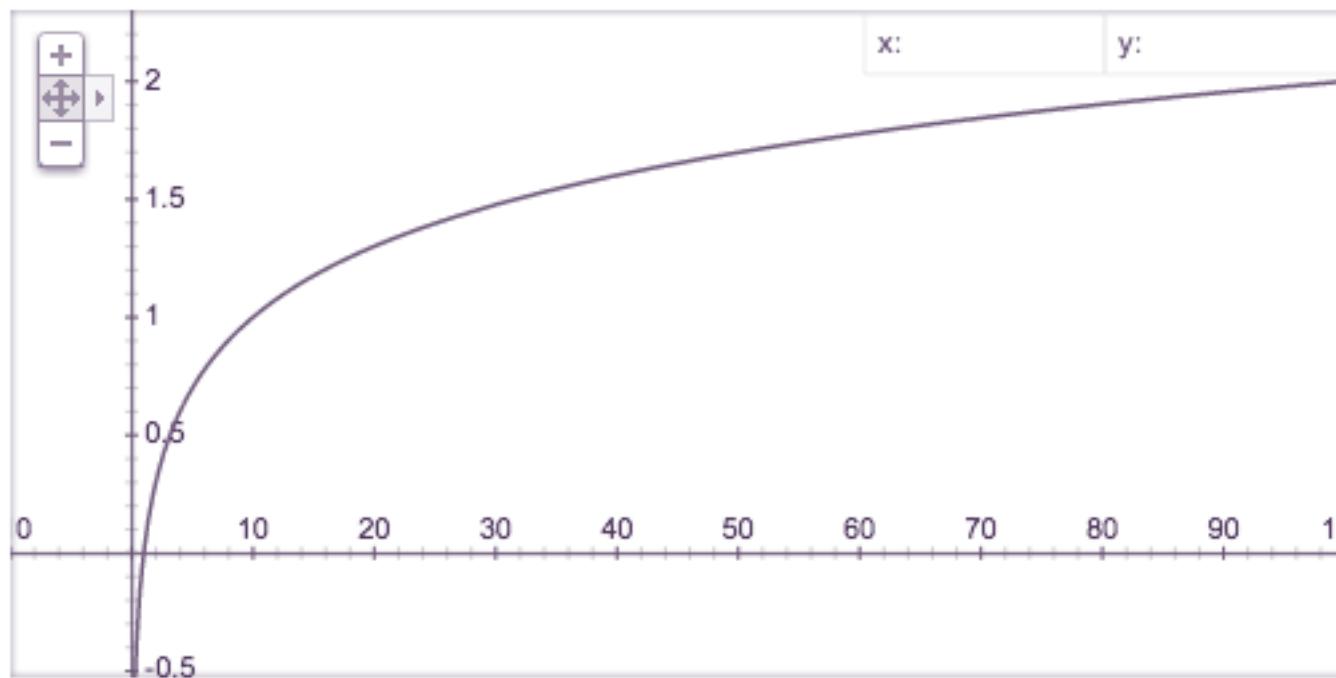
El tiempo que le toma al algoritmo de las primeras dos soluciones en realizar la multiplicación es lineal:

$$t = 2 * x$$



El tiempo que le toma al algoritmo de la solución mejorada en realizar la multiplicación es logarítmico:

$$\lceil \log_2 x \rceil + 1$$



Por lo tanto, a pesar de que los tres algoritmos cumplen con la solución del problema planteado, el último es más eficiente.

Al este último método se le conoce como algoritmo de los campesinos rusos (Russian peasant algorithm) para resolver una multiplicación.

2.2.1 Representación de algoritmos

Los algoritmos se describen como sucesiones de instrucciones que procesan la entrada $\rho \in E$ para producir el resultado $\xi \in S$.

Cada instrucción es una operación simple que produce un resultado intermedio único y es posible ejecutar con eficiencia.

La sucesión S de instrucciones tiene que ser finita y tal que para toda $\rho \in E$, el resultado del proceso será $f(\rho) \in S$. Sería altamente deseable que para todo $\rho \in E$, la ejecución terminará después de un tiempo finito.

Los algoritmos se implementan como programas en diferentes lenguajes de programación.

Existen diferentes representaciones algorítmicas, dependiendo del lenguaje de programación en el que se vaya a implementar la solución, los más utilizados son los diagramas de flujo o pseudocódigo (para lenguajes de programación estructurada) y los diagramas UML (para lenguajes de programación orientado a objetos).

Ejemplo 3

Realizar un programa que multiplique dos números (equis y ye) sin utilizar el operador multiplicación (*), es decir, no se puede especificar en el algoritmo la operación:

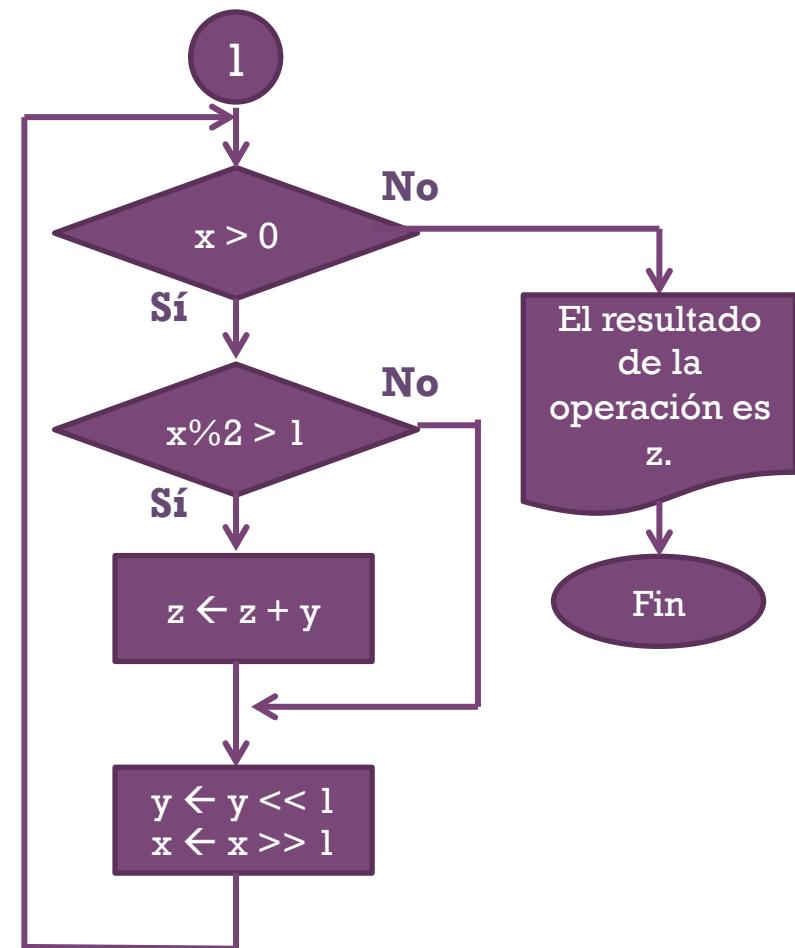
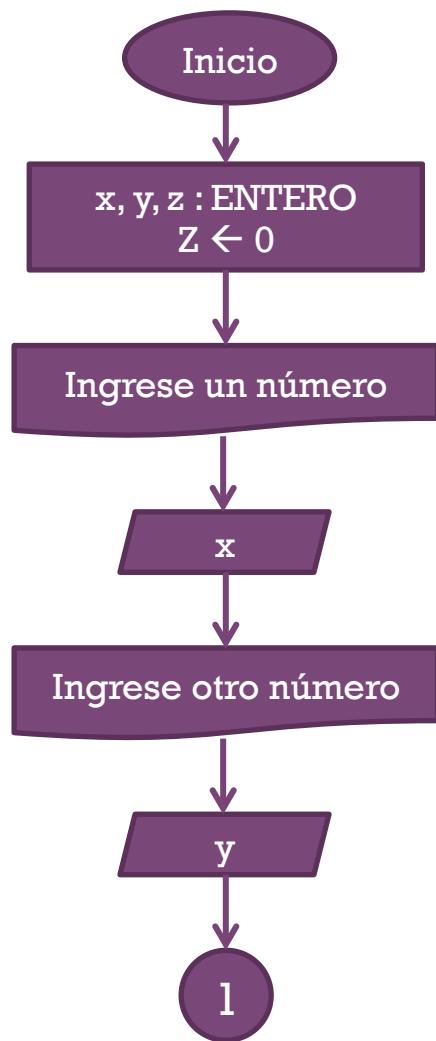
equis * ye

Ejemplo 3**Pseudocódigo**

- 1. INICIO**
- 2. X, Y, Z: ENTERO**
- 3. Z := 0**
- 4. ESCRIBIR “Ingrese un número entero.”**
- 5. LEER X.**
- 6. ESCRIBIR “Ingrese otro número entero.”**
- 7. LEER Y.**
- 8. MIENTRAS X > 0**
- 9. Si (X % 2) = 1**
- 10. Z := Z + Y**
- 11. Y << 1**
- 12. X >> 1**
- 11. FIN_MIENTRAS**
- 12. ESCRIBIR “El resultado de la operación es: “ Z**
- 13. FIN**

Ejemplo 3

Diagrama de flujo



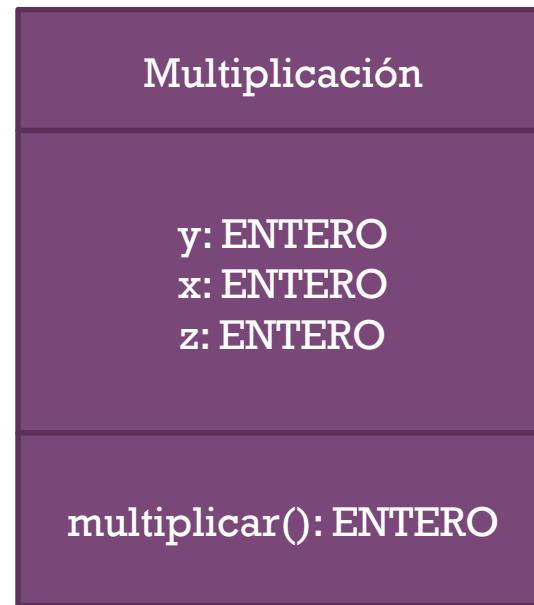
Ejemplo 3

```
#include <stdio.h>

int main () {
    int x, y, z;
    z = 0;
    printf ("Ingrese un nUmero:\n");
    scanf ("%d",&x);
    x = x<0?x*-1:x;
    printf ("Ingrese otro nUmero:\n");
    scanf ("%d",&y);
    y = y<0?y*-1:y;
    while ( x > 0 ) {
        if ((x%2) == 1)
            z+=y;
        y=y<<1;
        x=x>>1;
    }
    printf("El resultado es %d.\n",z);
}
```

Ejemplo 3

Diagrama de clase



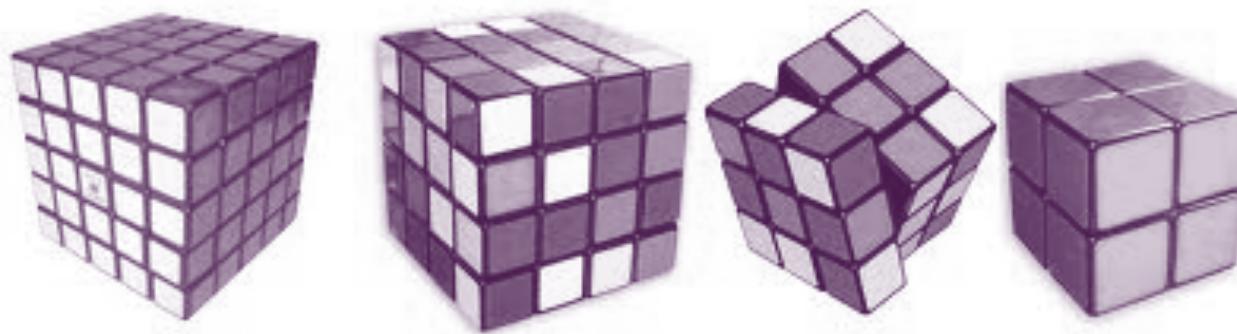
Ejemplo 3

```
public class Multiplicacion {  
    public int x, y;  
  
    public int multiplicar() {  
        if ( x < 0 )  
            x*=(-1);  
        if ( y< 0 )  
            y*=(-1);  
        int z=0;  
        while ( x > 0 ) {  
            if ((x%2) == 1)  
                z+=y;  
            y=y<<1;  
            x=x>>1;  
        }  
        return z;  
    }  
}
```

Ejemplo 3

```
import java.io.Console;

public class PruebaMultiplicacion{
    public static void main (String...str){
        Console c = System.console();
        Multiplicacion m = new Multiplicacion();
        System.out.println("Ingrese un nUmero: ");
        m.x = Integer.parseInt(c.readLine());
        System.out.println("Ingrese otro nUmero: ");
        m.y = Integer.parseInt(c.readLine());
        System.out.println("El resultado es: " + m.multiplicar());
    }
}
```



2.3 Complejidad.

2.3 Complejidad.

Las dos medidas más importantes para establecer la calidad de un algoritmo son:

- **El tiempo total de ejecución, medido por el número de operaciones de cómputo realizadas durante el proceso.**
- **La cantidad de memoria utilizada por dicho proceso.**

2.3.1 Medidas de complejidad.

Tanto la eficiencia en la ejecución como la cantidad de memoria utilizada durante un proceso son parámetros que permiten medir la complejidad de un algoritmo.

Las técnicas más utilizadas para comparar la eficiencia de algoritmos sin necesidad de implementarlos son el modelo de computación RAM (Random Acces Machine) y el análisis asintótico del peor caso de complejidad.

El modelo RAM se refiere a la representación de una computadora hipotética que permite evaluar la eficiencia del diseño de un algoritmo de manera independiente de la arquitectura (hardware) donde se implemente.

El modelo RAM asume las siguientes reglas:

- **Cada operación simple (+, -, *, /, selección o llamada) toma exactamente un paso de tiempo.**
- **Los ciclos están compuestos por operaciones simples, por lo tanto, el tiempo que toma un ciclo depende del número de iteraciones del mismo.**
- **El acceso a memoria toma exactamente un paso de tiempo. La memoria de un modelo RAM es infinita.**

Un modelo RAM permite medir el tiempo que consume la ejecución de un algoritmo, contando el número de pasos que contiene el mismo. Si se asume que un modelo RAM ejecuta un cierto número de pasos por segundo, en automático se puede obtener el tiempo de ejecución de un proceso.

La RAM es un modelo simple que intenta representar el funcionamiento de un equipo, y, por tanto, hay que tener ciertas consideraciones.

En la mayoría de los procesadores, multiplicar dos números toma más tiempo que sumarlos. Esto viola la primera regla del modelo.

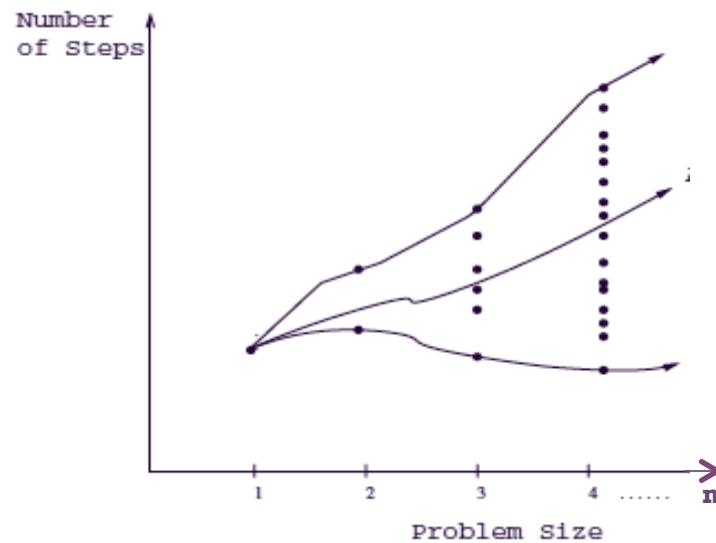
Además, los procesos paralelos o multihilos así como ciertos compiladores pueden violar la segunda regla.

Por si fuera poco, el acceso a la memoria difiere en tiempo dependiendo del lugar donde se encuentren los datos, con lo que se viola la tercera regla.

A pesar de los bemoles mencionados, el modelo RAM provee una excelente aproximación para entender cual sería el desempeño de un algoritmo en una computadora real. Es por lo anterior que es tan útil en la práctica.

2.3.2 Notación O.

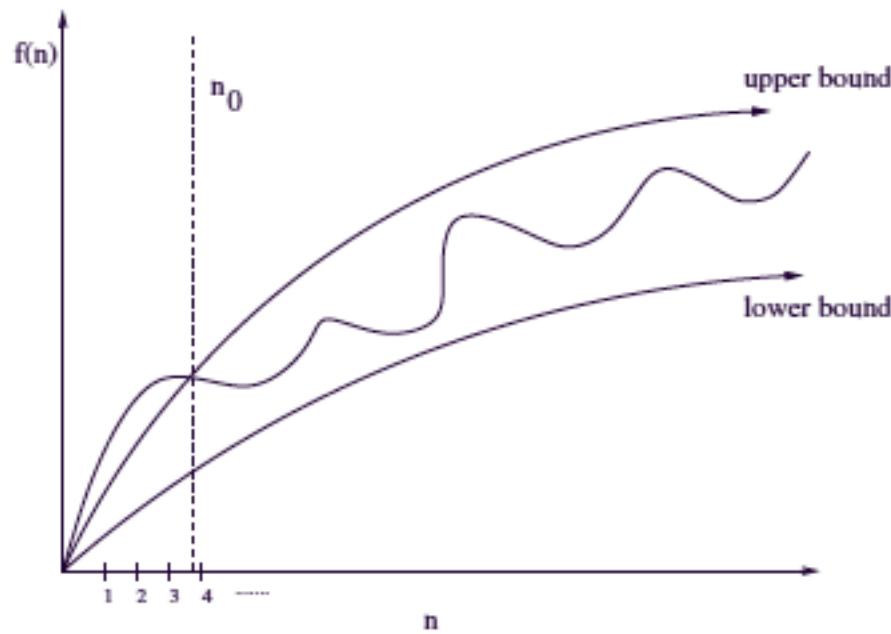
El tiempo de complejidad para un algoritmo dado es una función numérica que depende del tamaño de las instancias dadas (valores o datos de entrada).



Sin embargo, trabajar con las tres funciones o, más específicamente, con los casos extremos de las instancias para un algoritmo es bastante difícil debido a dos razones básicas:

- **Las funciones poseen muchos saltos: instancias nenes pueden requerir más tiempo que las pares, por ejemplo.**
- **Las funciones requieren mucho detalles para ser precisas: $t(n) = 12754n^2 + 4353n + 13546$**

Por tanto, es mejor trabajar con funciones de tiempo de complejidad con límites superior e inferior. Esto se logra utilizando la notación O.



La notación O simplifica el análisis, ignorando niveles de detalle que no impacta en la comparación de algoritmos.

La notación O no hace diferencia entre multiplicación de constantes, es decir, la función $f(n) = 2n$ y $g(n) = n$ son idénticas en el análisis O.

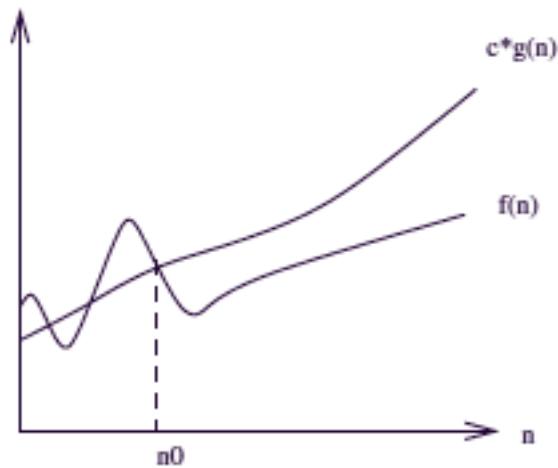
Las definiciones formales de la notación O son las siguientes:

- **$f(n) = O(g(n))$ significa que $c*g(n)$ es el límite superior en $f(n)$. Por tanto, existe una constante c tal que $f(n)$ es siempre menor o igual a $c*g(n)$ por muy grande que n sea, para $n > n_0$.**
- **$f(n) = \Omega(g(n))$ significa que $c*g(n)$ es el límite inferior en $f(n)$. Por tanto, existe una constante c tal que $f(n)$ es siempre mayor o igual a $c*g(n)$ para $n > n_0$.**

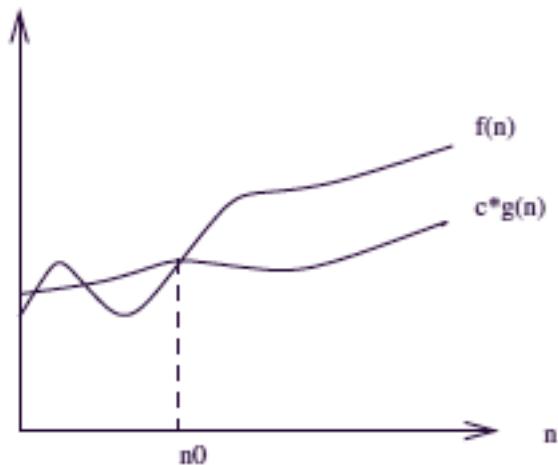
- $f(n) = \Theta(g(n))$ significa que $c_1 * g(n)$ límite superior en $f(n)$ y $c_2 * g(n)$ es un límite inferior en $f(n)$ para $n > n_0$. Por tanto, existen un par de constantes c_1 y c_2 tales que:

$$c_2 * g(n) \leq f(n) \leq c_1 * g(n)$$

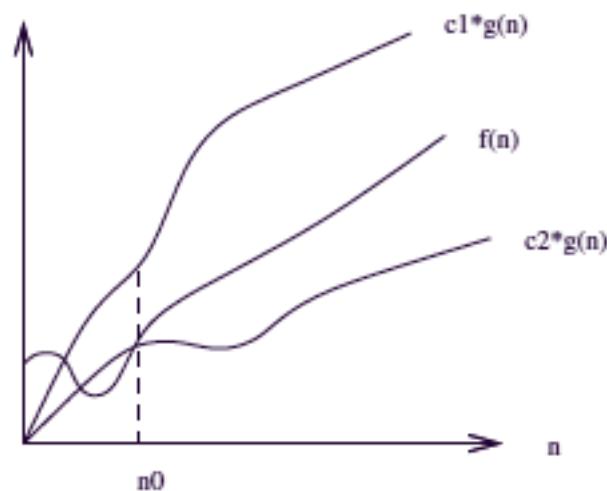
Lo que significa que $g(n)$ provee una muy estrecha y buena aproximación de $f(n)$.



$$f(n) = O(g(n))$$



$$f(n) = \Omega(g(n))$$



$$f(n) = \Theta(g(n))$$

Ejemplo 4

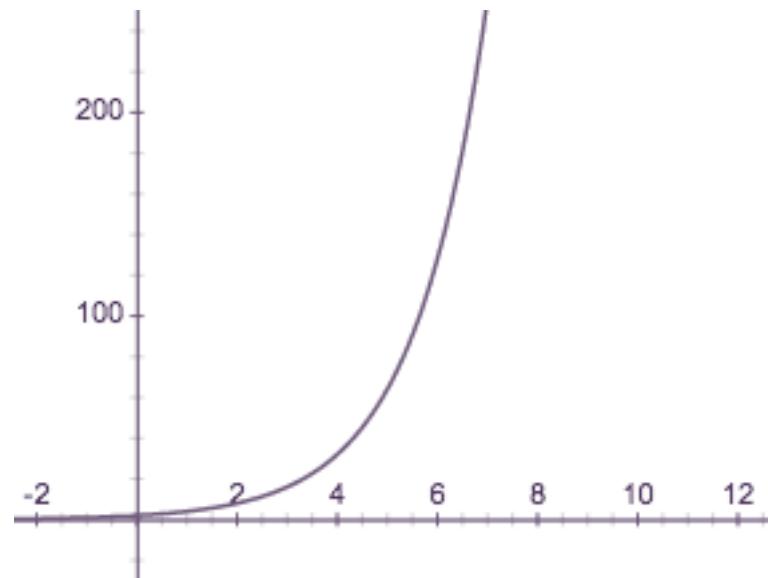
¿Se cumple la siguiente aseveración?:

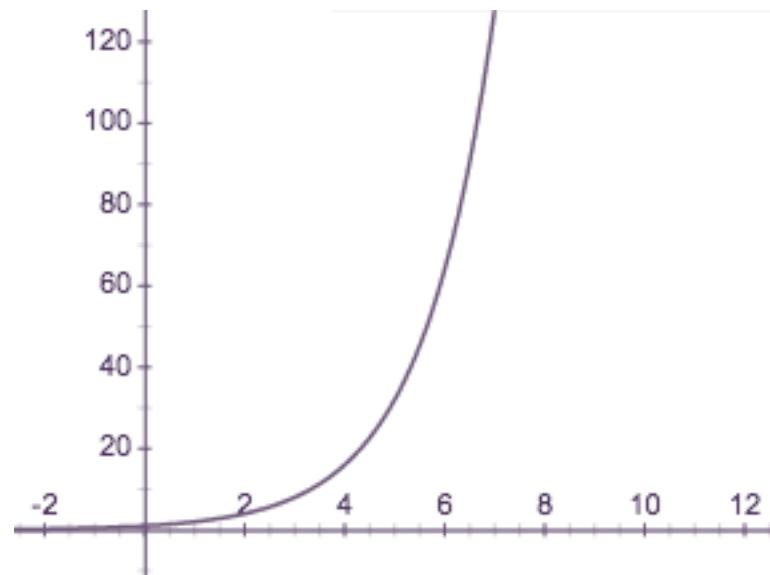
$$2^{n+1} = \Theta(2^n)$$

NOTA. En este caso, el significado del símbolo “=” se puede leer como “es una función de”.

Ejemplo 4

$$2^{n+1}$$



Ejemplo 4 2^n 

Ejemplo 4

$$\text{¿}2^{n+1} = O(2^n)?$$

Por definición, $f(n) = O(g(n))$ si y solo si existe una constante c tal que para un valor de n muy grande, se cumpla la condición $f(n) \leq c*g(n)$.

Observando la gráfica se tiene que $2^{n+1} = 2*2^n$, por lo tanto, existe una función $c*2^n$, que cumple con la función $f(n) = 2^{n+1}$ con $c \geq 2$.

Ejemplo 4

$$\text{¿}2^{n+1} = \Omega(2^n)?$$

Por definición, $f(n) = \Omega(g(n))$ si y solo si existe una constante $c > 0$ tal que para una valor de n muy grande, se cumpla la condición $f(n) \geq c*g(n)$.

Para la función 2^{n+1} la condición se satisface para los valores $0 < c \leq 2$.

Juntos los límites superior (O) e inferior (Ω), permiten validar que $2^{n+1} = \Theta(2^n)$.

Con la notación O se pueden descartar los multiplicandos constantes dentro de una función.

Así, las funciones $f(n) = 0.001n^2$ y $g(n) = 1000n^2$ son tratadas de manera idéntica, a pesar de que $g(n)$ es un millón de veces mayor que $f(n)$ para cualquier valor de n .

A continuación se muestran las tasas de crecimiento en tiempo de las funciones más comunes, en una computadora donde cada operación toma un nano segundo 10^{-9} [s]:

$n f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n! $
10	0.003 μ s	0.01 μ s	0.033 μ s	0.1 μ s	1 μ s	3.63 ms
20	0.004 μ s	0.02 μ s	0.086 μ s	0.4 μ s	1 ms	77.1 years
30	0.005 μ s	0.03 μ s	0.147 μ s	0.9 μ s	1 sec	8.4×10^{15} yrs
40	0.005 μ s	0.04 μ s	0.213 μ s	1.6 μ s	18.3 min	
50	0.006 μ s	0.05 μ s	0.282 μ s	2.5 μ s	13 days	
100	0.007 μ s	0.1 μ s	0.644 μ s	10 μ s	4×10^{13} yrs	
1,000	0.010 μ s	1.00 μ s	9.966 μ s	1 ms		
10,000	0.013 μ s	10 μ s	130 μ s	100 ms		
100,000	0.017 μ s	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μ s	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μ s	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μ s	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μ s	1 sec	29.90 sec	31.7 years		

A partir de la tabla anterior se pueden obtener las siguientes conclusiones:

- Todos los algoritmos toman aproximadamente el mismo tiempo para $n = 10$.
- El tiempo para cualquier algoritmo que ejecute $n!$ se vuelve inoperante para $n \geq 20$.
- Para algoritmos con tiempo de ejecución 2^n tienen un buen rango operativo, pero se vuelven imprácticos para $n > 40$.

- **Algoritmos cuyos tiempos de ejecución sean n^2 son rápidos mientras que $n < 10,000$, pero el tiempo se deteriora con rapidez con instancias más largas.**
- **Algoritmos lineales (n) o de la forma $n * \log(n)$ son prácticos con instancias hasta de 1000 millones.**
- **Un algoritmo de la forma $O(\log(n))$, mantienen un tiempo de ejecución bajo para cualquier valor de n .**

Como se puede observar, a pesar de ignorar los factores constantes, se obtiene una idea muy aproximada de si un algoritmo es apropiado con base en el tamaño de los elementos de entrada.

Un algoritmo cuyo tiempo de ejecución sea $f(n) = n^3$ será más rápido que un algoritmo cuyo tiempo de ejecución sea $g(n) = 1,000,000 n^2$ cuando $n < 1,000,000$.

2.3.3 Algoritmos de comportamiento asintótico.

La tasa de crecimiento del tiempo de ejecución de un algoritmo permite obtener una aproximación de la eficiencia del mismo, así como comparar el comportamiento con otros algoritmos.

Cuando se comparan valores de entrada suficientemente grandes se hace relevante el orden de crecimiento del tiempo de ejecución y, por tanto, se está estudiando la eficiencia asintótica del algoritmo.

La idea es analizar el incremento en el tiempo de ejecución de un algoritmo con respecto al tamaño de la entrada en los límites.

Usualmente, un algoritmo asintóticamente eficiente será la mejor opción para cualquier conjunto de entradas.

Los límites asintóticos están definidos por la función $\Theta(g(n))$, es decir:

$$\Theta(g(n)) = f(n)$$

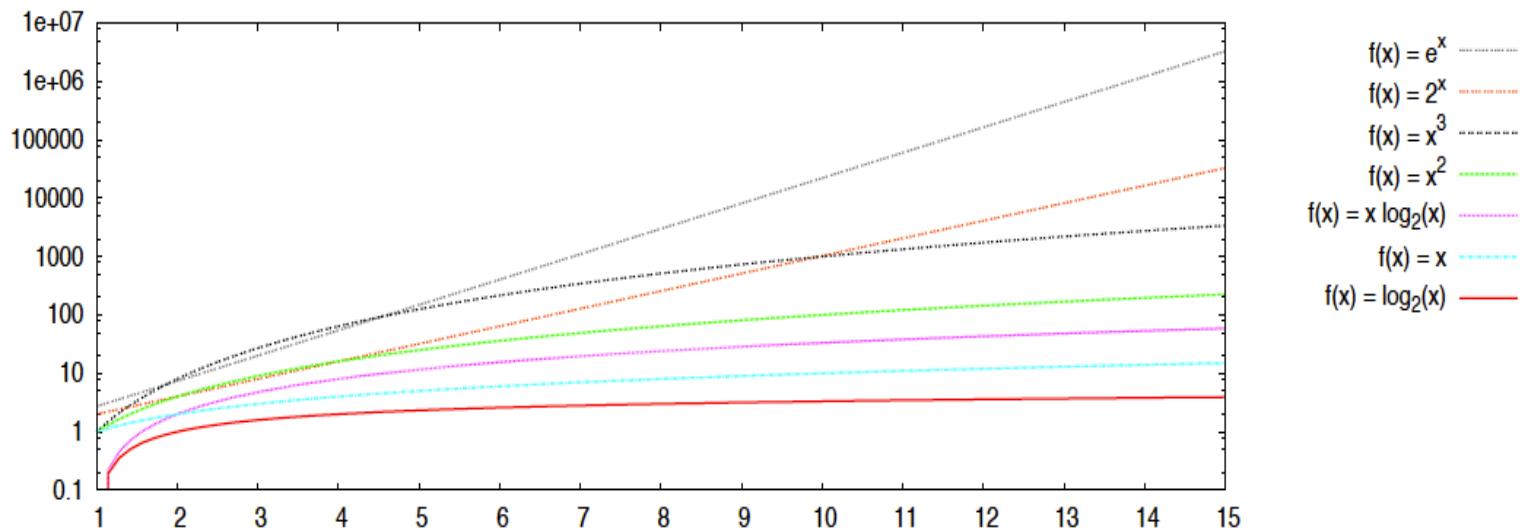
para la cual existen tres constantes positivas (c_1 , c_2 y n_0) tal que

$$0 < c_2 * g(n) < f(n) < c_1 * g(n)$$

para cualquier $n > n_0$.

Recordando que $c_1 \cdot g(n) = O(g(n))$ y $c_2 \cdot g(n) = \Omega(g(n))$, se puede afirmar que:

- **$O(g(n))$ es una cota asintótica superior.**
- **$\Omega(g(n))$ es una cota asintótica inferior.**
- **$\Theta(g(n))$ es un conjunto al que pertenece $f(n)$.**
- **$g(n)$ es un límite asintótico estrecho.**



2.3.4 Algoritmos de tiempo polinomial.

Se conoce como clase de problema P a aquellos algoritmos que son resolubles en un tiempo polinomial.

Más específicamente, los algoritmos clase P son problemas que pueden resolverse en un tiempo $O(n^k)$ para alguna constante 'k', donde 'n' es el tamaño de la entrada del problema.

Dada una función correcta de complejidad f , se tienen las siguientes clases:

tiempo

determinista $\text{TIME}(f)$
no determinista $\text{NTIME}(f)$

espacio

determinista $\text{SPACE}(f)$
no determinista $\text{NSPACE}(f)$

La clase P abarca las siguientes funciones de complejidad computacional:

Tiempo	Clase	Ejemplo
Problemas con algoritmos eficientes		
$\mathcal{O}(1)$	P	si un número es par o impar
$\mathcal{O}(n)$	P	búsqueda de un elemento entre n elementos
$\mathcal{O}(n \log n)$	P	ordenación de n elementos
$\mathcal{O}(n^3)$	P	multiplicación de matrices
$\mathcal{O}(n^k)$	P	programación lineal

2.3.5 Algoritmos factibles y no factibles.

Se parte de la afirmación de que se poseen una serie de conjuntos de números reales.

Se tienen un conjunto de n números reales ($c_1, c_2, c_3, \dots c_n$), otro conjunto de m números reales ($b_1, b_2, b_3, \dots b_n$) y un tercer conjunto $m*n$ de números reales (a_{ij} , para $i=1,2,3,\dots, m$ y $j=1,2,3,\dots, n$).

Se desean encontrar n números reales ($x_1, x_2, x_3, \dots x_n$), tales que maximicen la expresión:

$$\sum_{j=1}^n c_j * x_j \quad (1)$$

Así mismo, la expresión (1) está sujeta a:

$$\sum_{j=1}^n a_{ij} * x_j \leq b_i \quad \text{con } i=1,2,3,\dots, m \quad (2)$$

$$x_j \geq 0 \quad \text{con } j=1,2,3,\dots,n \quad (3)$$

A la expresión (1) se le llama **función objetivo** y a las desigualdades de las expresiones (2) y (3) se les llama **restricciones**. Al conjunto de n restricciones de la expresión (3) se le llama **restricciones no-negativas**.

El conjunto de expresiones anotadas anteriormente definen un **programa lineal**. A veces es conveniente expresar un programa lineal en una forma más compacta.

Si se crea una matriz A de $m \times n$ ($A = a_{ij}$), un vector b de dimensión n ($b = b_i$), un vector c de dimensión m ($c = c_j$) y un vector x de dimensión n ($x = x_j$), es posible reescribir el programa lineal definido en las expresiones (1), (2) y (3) de la siguiente manera:

$$c^T x \quad (4)$$

Donde la expresión (4) está sujeta a:

$$A x \leq b \quad (5)$$

$$x \geq 0 \quad (6)$$

Donde $c^T x$ es el producto interno de dos vectores, $A x$ es el producto de la matriz A por el vector x , tal que el vector x debe ser no-negativo.

Por tanto, se puede definir un programa lineal por una tripleta de elementos (A , b , c), donde A es una matrix de $m \times n$, b es un vector de tamaño m y c es un vector de tamaño n .

Al conjunto de variables del vector x que satisface todas las restricciones se le conoce como *solución factible*. Al conjunto de variables que no satisfacen, al menos una restricción, se les llama *solución infactible*.

Así mismo, se dice que el conjunto de soluciones x posee un valor objetivo $c^T x$.

Una solución factible x cuyo valor objetivo es el máximo de todas las soluciones factibles se le conoce como la *solución óptima*, y a ese valor objetivo $c^T x$ se le conoce como el *valor objetivo óptimo*.

Si un programa lineal no tiene soluciones factibles, se puede afirmar entonces que el programa es infactible, de otro modo se dice que es factible.

Si un programa lineal tiene algunas soluciones factibles pero no tiene un valor objetivo óptimo finito entonces se dice que el programa lineal no tiene límites (es ilimitado).

Ejemplo 5

Dado el siguiente programa lineal con dos variables, se desea maximizar la función

$$x_1 + x_2 \quad (1)$$

sujeto a las restricciones:

$$4x_1 - x_2 \leq 8 \quad (2)$$

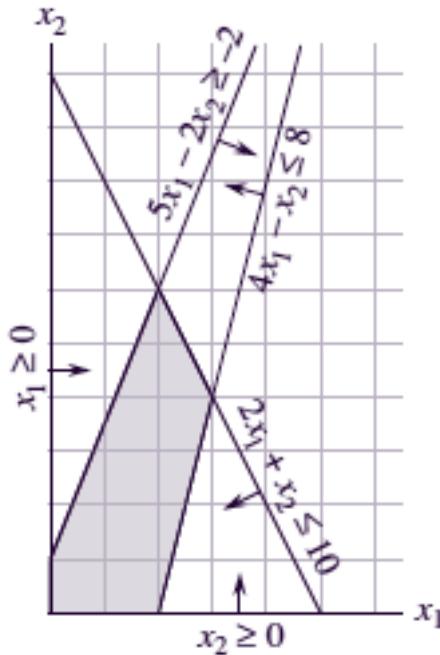
$$2x_1 + x_2 \leq 10 \quad (3)$$

$$5x_1 - 2x_2 \geq -2 \quad (4)$$

$$x_1, x_2 \geq 0 \quad (5)$$

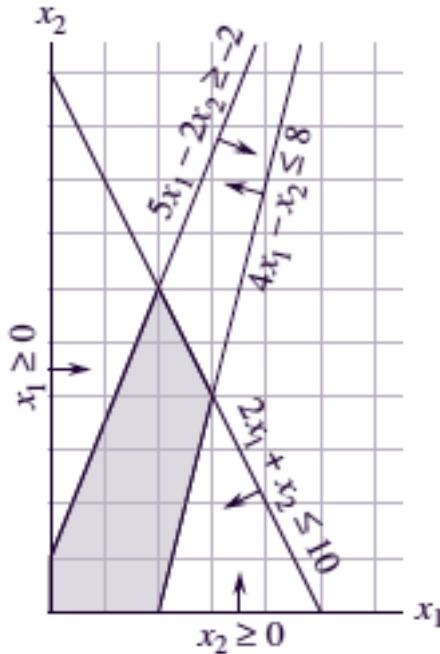
A cualquier conjunto de variables que satisface todas las restricciones se le considera una solución factible del programa lineal.

Ejemplo 5



Como se puede observar, el conjunto de soluciones factibles se ve en la parte sombreada de la gráfica (región factible).

Ejemplo 5



La función que se desea maximizar se llama función objetivo (x_1+x_2). Si se evalúa la función objetivo en cada punto de la región factible, el valor obtenido para cada punto se le conoce como valor objetivo. El valor objetivo máximo se le conoce como solución óptima.

Ejercicio 2

Para cada función en la siguiente tabla, determinar el tamaño más grande que puede tomar n para que el problema pueda ser resuelto en un tiempo t , asumiendo que el algoritmo para resolver el problema toma $f(n)$ microsegundos.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
\sqrt{n}							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

2.3.6 Cota inferior y superior.

Teorema: Para cualesquiera dos funciones $f(n)$ y $g(n)$, se tiene que $f(n) = \Theta(g(n))$ si y sólo si $f(n) = O(g(n))$ así como $f(n) = \Omega(g(n))$.

Por lo tanto, la igualdad $an^2 + bn + c = \Theta(n^2)$ para cualesquiera constantes a , b y c , con $a > 0$, implica que $an^2 + bn + c = \Omega(n^2)$ y $an^2 + bn + c = O(n^2)$.

En la práctica, el teorema anterior se utiliza para probar lo estrecho de los límites asintóticos de las cotas superior e inferior.

Debido a que la notación Ω describe una cota inferior, cuando se usa Ω para indicar el límite del mejor caso en tiempo de ejecución de un algoritmo, también se indica el tiempo de ejecución del algoritmo para cualquier entrada arbitraria.

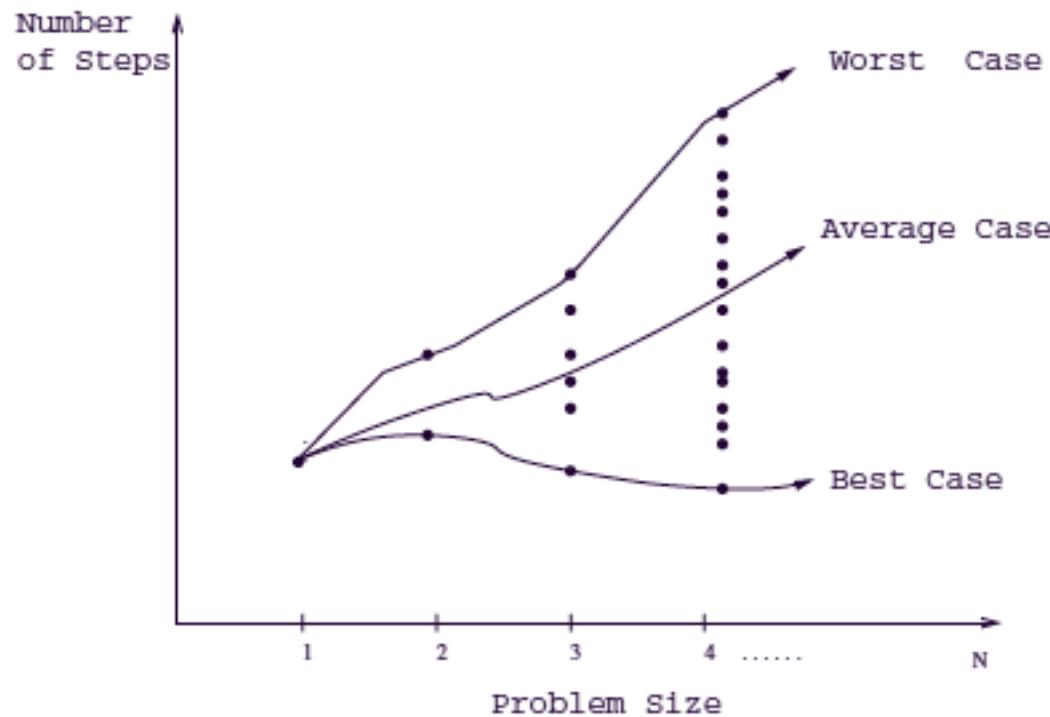
2.3.7 Valor promedio, el mejor y el peor caso.

Utilizando el modelo computacional RAM, es posible contar cuantos pasos toma un algoritmo en realizar una tarea con base en una instancia de entrada (datos de entrada).

Sin embargo, para ponderar que tan eficiente o ineficiente es un algoritmo, es necesario probar todas las instancias de entrada posibles.

Para entender el concepto de el mejor, el peor y el caso promedio de complejidad es necesario que, para un algoritmo dado, se ejecuten todas las posibles instancias de datos que puedan alimentarlo.

Por ejemplo, para un algoritmo de ordenamiento el conjunto de instancias de entrada consta de todas las posibles combinaciones de k elementos de todas las posibilidades de n .



El eje x representa el tamaño de la entrada del problema. El eje y representa el número de pasos que toma al algoritmo en cada instancia.

A partir de la gráfica anterior se pueden observar tres funciones diferentes.

1. El peor caso de complejidad de un algoritmo es una función definida por el máximo número de pasos para cada instancia de tamaño n . Esta curva representa el punto más alto para cada columna (dato de entrada).

2. El mejor caso de complejidad de un algoritmo es la función definida a partir del menor número de pasos para cada instancia de tamaño n . Esta curva representa el punto más bajo para cada columna (dato de entrada).

3. El caso promedio de complejidad de un algoritmo es una función definida por un número promedio de pasos de cada intancia de tamaño n .

2.3.8 Compromisos espacio-tiempo.

La eficiencia de un programa se puede plantear como un compromiso entre el tiempo y el espacio utilizados. Por eso, la etapa de diseño es tan importante dentro del proceso de construcción de algoritmos, ya que va a determinar, en muchos aspectos, la calidad del producto obtenido.

Por tanto, el uso eficiente de los recursos suele medirse en función del espacio (determinado por la cantidad de memoria que utiliza) y el tiempo (representado por el tiempo que tarda un algoritmo en ejecutarse).

El espacio y el tiempo representan el costo de la solución al problema planteado mediante un algoritmo. Dichos parámetros sirven para comparar varios algoritmos entre sí, para determinar el más adecuado.

El mismo concepto que se utiliza para medir la complejidad del tiempo de ejecución de un algoritmo se utiliza para medir su complejidad en espacio.

Por tanto, un programa tiene una complejidad en espacio $\Theta(n)$ significa que sus requerimientos de memoria aumentan de manera proporcional con el tamaño del problema.

Para un programa de complejidad en espacio $\Theta(n^2)$, la cantidad de memoria que se necesita para almacenar los datos crece con el cuadrado del tamaño del problema: si el problema se duplica, se requiere cuatro veces más memoria.

Al aumentar el espacio utilizado para almacenar la información se puede conseguir un mejor desempeño. Por tanto, entre más sencillas sean las estructuras de datos más lentos resultan los algoritmos.

Las estructuras de datos llevan implícitas ciertas limitaciones de eficiencia en sus operaciones básicas. Por eso la etapa de diseño es tan importante dentro del proceso de construcción de software ya que ahí se determinan muchos aspectos de la calidad del algoritmo.

2.3.9 Clases de complejidad: P, NP, NP completos.

La clase P consiste en aquel conjunto de problemas que se pueden resolver en un tiempo polinomial.

Especificamente, hay problemas que pueden ser resueltos en un tiempo $O(n^k)$, para alguna constante k , donde n es el tamaño de la entrada del problema.

La clase NP consiste de aquellos problemas que son verificables en tiempo polinomial. Esto es, si de alguna manera se obtiene un certificado de solución para un algoritmo, ese certificado se puede comprobar en tiempo polinomial en el tamaño de entrada del problema.

NP es un nombre compuesto por No-determinista y por Polinómico. Lo que significa que su solución no es polinómica porque no se puede definir una ecuación que lo resuelva, es decir, para un valor de entrada existen muchos posibles resultados.

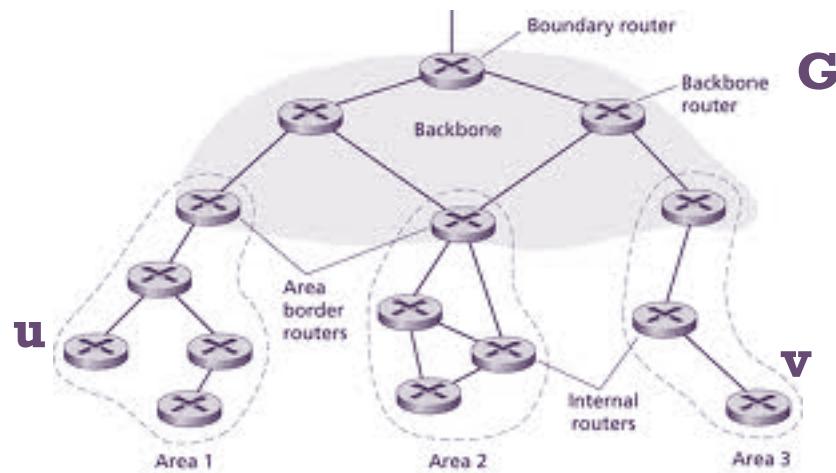
Un problema pertenece a la clase NPC (NP Completo) si está en NP y es tan difícil como cualquier problema en NP.

La mayoría de los teóricos científicos en computación creen que los problemas NP-Complejos no se pueden resolver, debido a que de todos los problemas que han estudiado hasta el momento, no hay alguno que tenga una solución en tiempo polinomial.

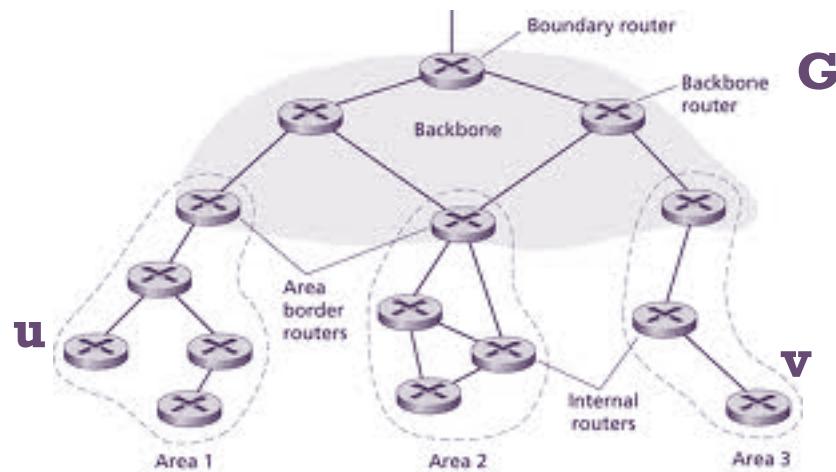
Para volverse un buen diseñador de algoritmos es necesario entender los primeros conceptos de la teoría de NP-completos.

Si se puede demostrar que un problema es NP-completo, se cuenta con pruebas suficientes de que dicho problema es intratable. Como ingeniero es mejor gastar tiempo en desarrollar un algoritmo aproximado o resolver un caso particular del problema, antes que buscar una solución exacta al problema.

En la creación de algoritmos se busca optimizar el tiempo de ejecución. Por ejemplo, al tratar de encontrar la ruta más corta, se tiene:

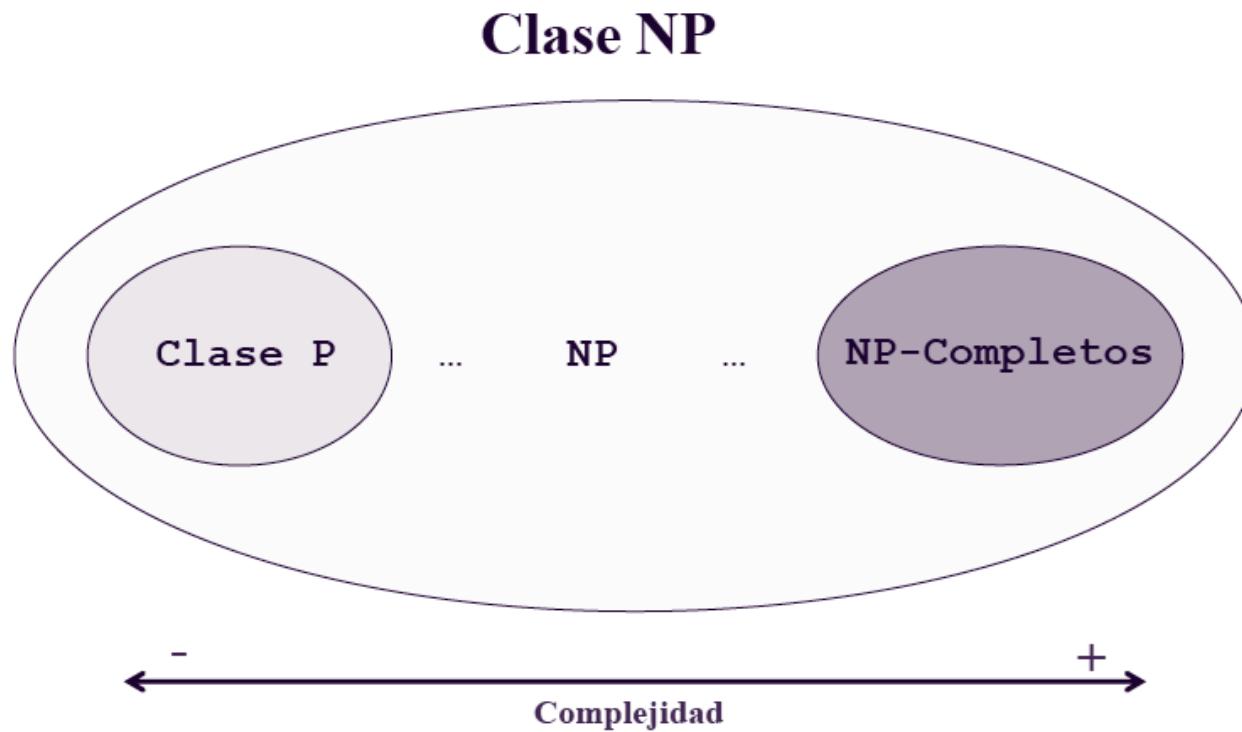


Los algoritmos NPC no ven un problema de optimización de tiempo sino un problema de decisión, donde la respuesta es muy sencilla sí (1) o no (0).



Existe una conveniente relación entre los problemas de optimización y los problemas de decisión. Usualmente, es posible moldear un problema de optimización a uno de decisión limitando los valores a ser optimizados.

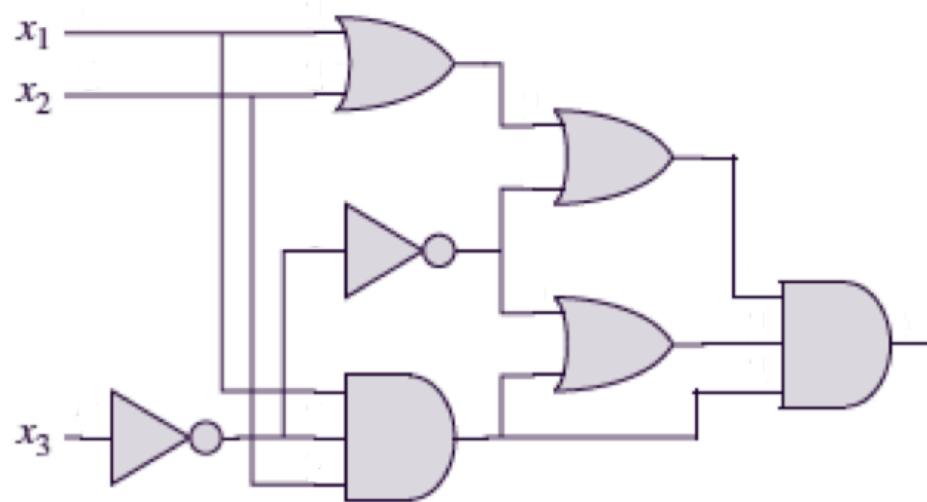
El problema de la ruta más corta, se puede moldear como un problema de decisión si, dados como datos una gráfica G , un par de vértices u y v y un entero k , existe una ruta de u a v que tenga como máximo k pasos.



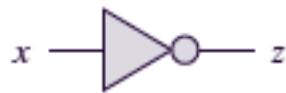
Esta figura muestra el concepto que tienen la mayoría de los científicos en computación en cuanto a la relación entre las clases P, NP y NPC.

Ejemplo 6

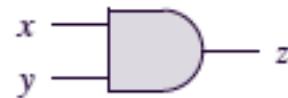
Dado el siguiente circuito booleano combinacional formado por compuertas lógicas NOT, AND y OR, ¿es posible satisfacer el circuito, es decir, ¿existe un conjunto de entradas que generen como salida 1?



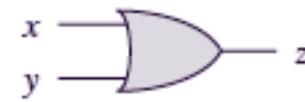
Ejemplo 6



x	$\neg x$
0	1
1	0



x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1



x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

AND

OR

Ejemplo 6

Se parte de que el circuito booleano combinacional no contiene ciclos, es decir, se posee una gráfica G , vértices con cada elemento combinacional y k vértices a la salida de los circuitos, de tal manera que existe un vértice (u,v) que conecta la salida del elemento u con la entrada del elemento v . Por lo tanto, se puede afirmar que G no es cíclica.

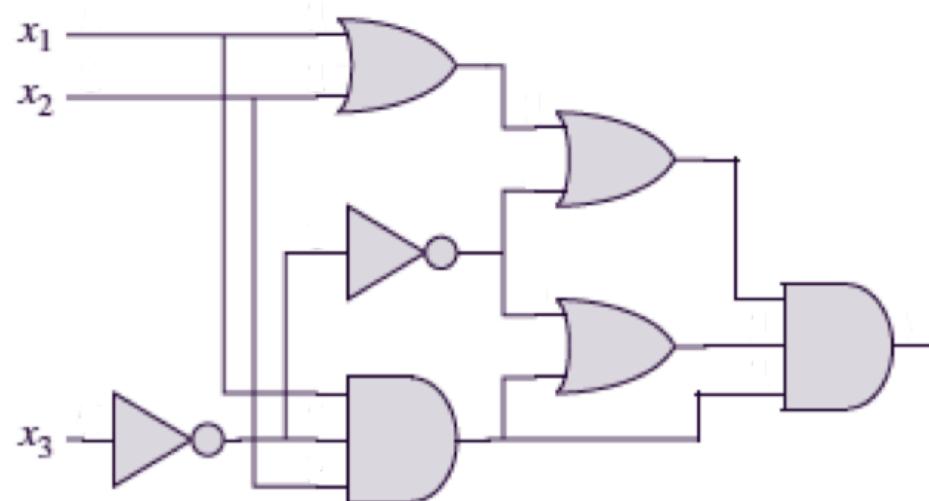
Ejemplo 6

Para un circuito booleano combinacional una asignación válida es aquel conjunto de valores de entrada booleanos.

Se dice que un circuito se puede satisfacer si existe una asignación válida (valor de entrada) que provoca que la salida del circuito sea 1.

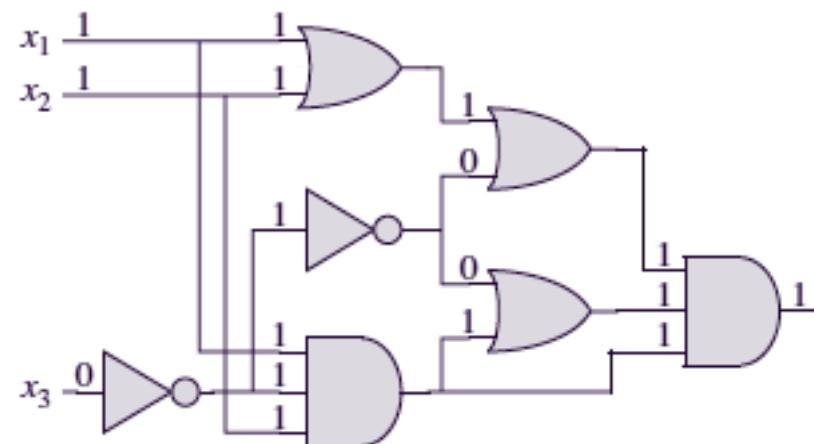
Ejemplo 6

¿Qué combinaciones de entrada satisfacen al circuito?



Ejemplo 6

¿Qué combinaciones de entrada satisfacen al circuito?



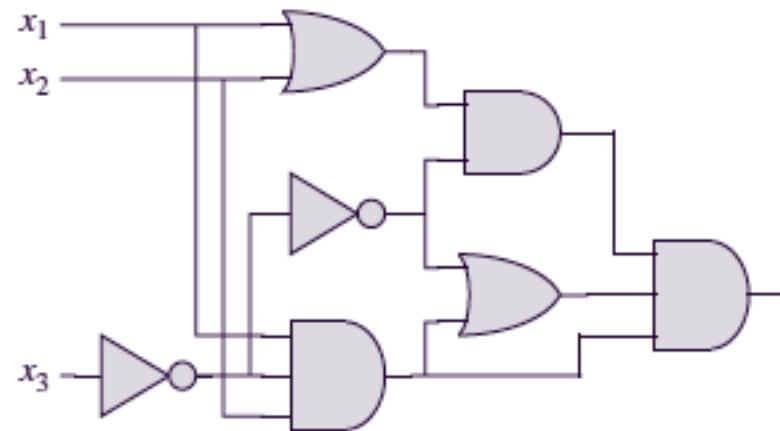
El tamaño de un circuito booleano combinacional está determinado por el número de valores booleanos de entrada y el número vértices (enlaces) del circuito.

Dado un circuito C, se puede determinar si éste se puede satisfacer únicamente revisando las posibles asignaciones de entrada.

El problema surge cuando se tienen k entradas, porque existen 2^k asignaciones posibles, lo que hace el algoritmo súper-polinomial. Con base en lo anterior, se puede afirmar que un circuito combinacional booleano pertenece a la clase NPC.

Ejercicio 3

Dado el siguiente circuito booleano combinacional formado por compuertas lógicas NOT, AND y OR, ¿es posible satisfacer el circuito, es decir, ¿existe un conjunto de entradas que generen como salida 1?



2.3.10 Métodos para encontrar soluciones aproximadas a problemas no factibles.

En la práctica, la mayoría de los problemas son de la clase NP-Completo, sin embargo, esa no es una razón suficiente para declinar la implementación de un algoritmo.

Existen, por lo menos, tres maneras diferentes para tratar de implementar algoritmos del tipo NPC.

- Dado un algoritmo con un tiempo de ejecución exponencial, si la entrada del problema es pequeña (o se puede acotar), el algoritmo se puede implementar.

n	$f(n)$	2^n
10		1 μ s
20		1 ms
30		1 sec
40		18.3 min
50		13 days
100		4×10^{13} yrs
1,000		
10,000		
100,000		
1,000,000		
10,000,000		
100,000,000		
1,000,000,000		

- Es posible acotar algunos problemas que tengan un tiempo de ejecución polinomial.

n	$f(n)$	n^2
10		0.1 μ s
20		0.4 μ s
30		0.9 μ s
40		1.6 μ s
50		2.5 μ s
100		10 μ s
1,000		1 ms
10,000		100 ms
100,000		10 sec
1,000,000		16.7 min
10,000,000		1.16 days
100,000,000		115.7 days
1,000,000,000		31.7 years

- **Partiendo de una solución en tiempo polinomial, es posible encontrar una solución óptima cercana, es decir, encontrar un algoritmo aproximado para llegar a la solución del problema.**

Suma de subconjuntos

El problema de la suma de subconjuntos es un problema de clase NP-Completo.

Se parte de un conjunto S que pertenece a N y un valor objetivo t que pertenece a N. Se desea averiguar si existe un subconjunto S' cuyos elementos sumen t.

Por ejemplo, si se posee el conjunto S y el valor objetivo t:

S = {1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993}

t = 138457

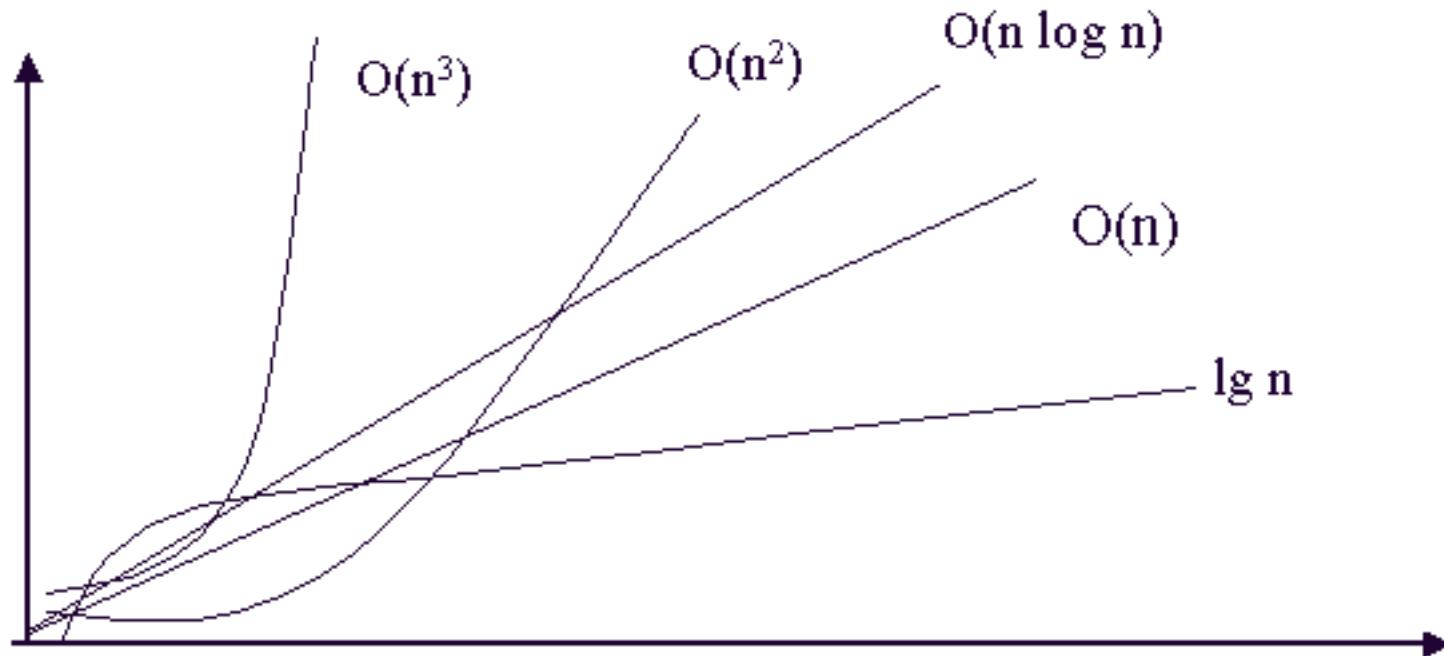
El subconjunto S' = {1, 2, 7, 98, 343, 686, 2409, 17206, 117705} es una solución al problema.

Una instancia de un problema de suma de subconjuntos está dada por los pares (S, t) , donde S es un subconjunto formado por $\{x_1, x_2, \dots, x_n\}$ de enteros positivos y t es un entero positivo.

Este es un problema de decisión que consiste en saber si existe un subconjunto S tal que sumados todos sus miembros den como resultado el valor exacto de t .

Una aplicación práctica del problema suma de subconjuntos se presenta al cargar una camioneta.

Una camioneta puede cargar no más de t kilos. Se poseen i cajas de peso x_i kilos y se desea llenar la camioneta sin exceder el límite de kilos que soporta la misma, ¿cuántas y cuáles cajas caben en la camioneta?



2.4 Análisis de algoritmos.

2.4 Análisis de algoritmos.

El análisis de algoritmos se ha convertido en una manera de predecir los recursos que se van a consumir: memoria, ancho de banda, arquitectura de la computadora, etc.

Empero, generalmente lo que se quiere medir es el tiempo computacional.

Cuando se analizan diversos algoritmos para solucionar un problema, es fácil identificar el más eficiente.

En el proceso de selección se puede tener más de un algoritmo que realice el trabajo correctamente y, a su vez, permite filtrar los que no lo son.

2.4.1 Algoritmos iterativos y recursivos

Cuando un algoritmo realiza llamadas a sí mismo se dice que éste es un algoritmo recursivo.

La recursividad se puede describir como una función en términos de un valor de entrada cada vez más pequeño.

Un algoritmo que en vez de llamarse a si mismo repite en una serie de código de manera cíclica se conoce como algoritmo iterativo.

Generalmente, un algoritmo recursivo resulta más corto que un algoritmo iterativo para solventar el mismo problema. Un algoritmo recursivo puede ser convertido a un algoritmo iterativo, aunque generalmente esto afecta la eficiencia del mismo.

Ejemplo 7

Un algoritmo iterativo se caracteriza porque utiliza estructuras de repetición:

INICIO

```
FUNC rec (x: Entero) DEV vacío
    MIENTRAS (y>0) HACER
        ESCRIBIR y%2
        y ← y/2;
    FIN_MIENTRAS
FIN_FUNC
FIN
```

Ejemplo 7

Un algoritmo recursivo se caracteriza porque utiliza estructuras de selección, además de que se manda llamar a sí mismo:

INICIO

FUNC rec (x: Entero) DEV vacío

SI (x > 0) HACER

rec (x/2);

ESCRIBIR x%2

FIN_SI

FIN_FUNC

FIN

Ejercicio 4

Realizar el pseudocódigo de un algoritmo iterativo. Así mismo, transformar el algoritmo iterativo en un algoritmo recursivo.

NOTA. No todos los algoritmos pueden ser recursivos, pensar en alguno que sí lo sea.

2.4.2 Análisis de algoritmos recursivos: ecuaciones de recurrencia.

Una ecuación de recurrencia es una expresión que define una sucesión, en la cual cada elemento se determina por otros elementos más sencillos (los anteriores), que incluyen condiciones iniciales.

Una ecuación de recurrencia sirve para medir la complejidad del algoritmo en cuestión y, con ello, estimar los recursos utilizados en tiempo de ejecución.

La sucesión de Fibonacci consiste en una serie de números que, iniciando por la unidad, cada uno de sus términos es la suma de los dos anteriores.

Esta construcción matemática aparece recurrentemente en la naturaleza: la distribución de las hojas alrededor del tallo, la reproducción de los conejos, la disposición de las semillas en numerosas flores y frutos, etc.

La sucesión de Fibonacci es la siguiente:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584...

Los números de Fibonacci tienen la característica de que el cociente de dos números consecutivos de la serie se aproxima a la denominada *razón dorada, sección áurea o divina proporción*. Este número tiene un valor de $\phi = (1+\sqrt{5})/2 = 1.61803398\dots$

Los cocientes (resultados de la división) de los números de Fibonacci consecutivos convergen rápidamente hacia el número áureo.

Los griegos y renacentistas consideraban este número el ideal de la belleza. Un objeto que tuviese una proporción (por ejemplo, entre alto y ancho) que se ajuste a la sección áurea era estéticamente más agradable que uno que no lo hiciese.



Cada término de la sucesión de Fibonacci se obtiene sumando los dos anteriores, es decir:

$$a_n = a_{n-1} + a_{n-2}, \text{ con } n > 2$$

La relación anterior define la sucesión de Fibonacci en función de los términos anteriores y, por tanto, recibe el nombre de relación de recurrencia, ecuación de recurrencia o ecuación en diferencias.

El problema de los algoritmos recursivos es que se necesita saber los términos anteriores para calcular el término n.

El objetivo de las ecuaciones de recurrencia es obtener el resultado del término enésimo sin necesidad de calcular los anteriores.

Ecuación de recurrencia lineal de primer orden

Las relaciones de recurrencia además de estar en función de uno o más términos anteriores, presentan un conjunto de condiciones iniciales para hacerse cumplir.

Sea la ecuación de recurrencia $a_n = 3a_{n-1}$ con $n \geq 1$ y el primer término de la serie $a_0 = 5$, obtener la solución para la ecuación de recurrencia descrita.

Dada $x_n = 3x_{n-1}$, con $n \geq 1$ y $x_0 = 5$:

$$a_0 = 5$$

$$a_1 = 3*a_0 = 3(5)$$

$$a_2 = 3*a_1 = 3*(3a_0) = 3*3*5 = 3^2 * 5$$

$$a_3 = 3*a_2 = 3*(3a_1) = 3*3*3*5 = 3^3 * 5$$

...

Por lo tanto, la solución de la ecuación de recurrencia es $x_n = 3^n(5)$.

En general, la solución de una ecuación de recurrencia de primer orden con $n \geq 1$ y $x_0 = A$ es:

$$a_n = A * r^n$$

Ecuación de recurrencia lineal de segundo orden

Dada la ecuación

$$c_n a_n + c_{n-1} a_{n-1} + c_{n-2} a_{n-2} = 0 \quad (1)$$

con $n \geq 2$, se busca una solución de la forma

$$a_n = A * r^n \quad (2)$$

Sustituyendo en (2) en (1) se tiene:

$$c_n A r^n + c_{n-1} A r^{n-1} + c_{n-2} A r^{n-2} = 0 \quad (3)$$

Si se factoriza (3) se obtiene:

$$Ar^{n-2} (c_n r^n + c_{n-1} r^{n-1} + c_{n-2} r^{n-2}) = 0 \quad (4)$$

$$c_n r^2 + c_{n-1} r + c_{n-2} = 0 \quad (5)$$

A la expresión (5) se le conoce como polinomio característico y sus raíces (soluciones) de ésta ecuación de recurrencia son dos raíces distintas r_1 y r_2 (reales o complejas).

Por lo tanto, la solución de la ecuación de recurrencia de segundo grado es

$$a_n = \alpha r_1^n + \beta r_2^n \quad (6)$$

Ejemplo 8

Resolver la siguiente ecuación de recurrencia con los valores iniciales dados:

$$\begin{cases} a_n + a_{n-1} - 6a_{n-2} = 0 & n \geq 2 \\ a_0 = 1, a_1 = 2 \end{cases}$$

Resolviendo el polinomio característico $x^2 + x - 6 = 0$ se tienen como resultado dos raíces distintas $x_1 = 2, x_2 = -3$.

Ejemplo 8

Se conoce que la solución general de la ecuación de recurrencia es:

$$a_n = \alpha r_1^n + \beta r_2^n$$

Sustituyendo con las raíces encontradas se tiene:

$$a_n = \alpha 2^n + \beta (-3)^n$$

Ejemplo 8

Para determinar los valores de α y β se utilizan los valores en la frontera $a_0=1$ y $a_1=2$ y con ello se tiene el sistema:

$$\alpha + \beta = 1$$

$$2\alpha - 3\beta = 2$$

Resolviendo el sistema se tiene que $\alpha = 1$ y $\beta = 0$. Con esto se puede establecer que la solución de la ecuación de recurrencia es:

$$a_n = 2^n$$

Ejercicio 5

La sucesión de Fibonacci está definida por la siguiente relación de recurrencia:

$$\begin{aligned}f_{n+2} &= f_{n+1} + f_n, \quad \text{con } n > 1 \\f_0 &= 0, f_1 = 1\end{aligned}$$

Obtener la solución general de la ecuación de recurrencia para la serie de Fibonacci.

2.4.3 Estimación de costos.

Así como las pruebas de un algoritmo pueden revelar errores de diseño, el cronometrar el tiempo de ejecución de un algoritmo puede revelar anomalías en el comportamiento del mismo en tiempo de ejecución para entradas específicas.

A partir de una función es posible estimar los recursos computacionales que necesita un equipo para llevar a cabo su rutina.

```
FUNC cuantos (lista:REAL) DEV num:ENTERO
    SI (lista=Vacía)
        return 0;
    FIN_SI
    EN_CASO CONTRARIO
        DEV cuantos(dism lista) + 1;
    FIN_CASO CONTRARIO
FIN_FUNC
```

Si, para la función anterior, se ingresa una lista de 3 elementos, el resultado debe ser 3:

lista ← ‘a’, ‘b’, ‘c’

Por tanto, si se aplica la función ‘cuantos’ a una lista más larga, se van a necesitar más pasos de recursión y viceversa.

El tiempo de ejecución de un algoritmo está dado por el número de operaciones primitivas o pasos a ejecutar. Un paso está definido por una cantidad constante de tiempo requerido para ejecutar cada línea del pseudocódigo.

Cada línea se ejecuta en una cantidad de tiempo diferente, pero se debe asumir que la ejecución de la i ésima línea toma un tiempo c_i , donde c_i es una constante.

A continuación se presenta el algoritmo de ordenamiento para un conjunto de elementos dados.

Cada línea genera un costo en tiempo de ejecución. Para cada $j = 2, 3, \dots, n$ donde $n =$ número de elementos, t_j es el número de veces que se ejecuta la condición del ciclo MIENTRAS, para cada valor de j .

Cuando existen ciclos del tipo MIENTRAS, la condición lógica se evalúa una vez más de las veces que se ejecuta el cuerpo del ciclo.

ORDENAMIENTO(A)

```

1 for j ← 2 to length[A]
2     key ← A[ j ]
3     i ← j - 1
4     while i > 0 and A[i] > key
5         A[i + 1] ← A[i ]
6         i ← i - 1
7     A[i + 1] ← key

```

costo iteraciones

c_1	n
c_2	$n - 1$
c_3	$n - 1$
c_4	$\sum_{j=2}^n t_j$
c_5	$\sum_{j=2}^n (t_j - 1)$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$n - 1$

El tiempo de ejecución de un algoritmo está dado por la suma de los tiempos de ejecución para cada línea de código.

Cada línea toma c_i pasos para ejecutarse y es ejecutada n veces, por tanto, el tiempo total de ejecución es $c_i n$.

Para computar el tiempo de ordenamiento $T(n)$ se suman los productos de costo y veces de cada línea:

$$\begin{aligned} T(n) = & c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^n t_j + \\ & + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n - 1) \end{aligned}$$

Hay que recordar que el tiempo de ejecución de un algoritmo depende del tamaño de la entrada del mismo.

Por ejemplo, para el algoritmo de ordenamiento el mejor caso ocurre cuando los elementos se dan de manera ordenada. Para cualquier valor de $j=2,3,\dots,n$, la condición del ciclo MIENTRAS siempre será $A[i] \leq key$. Por tanto, $t_j = 1$ para cualquier valor de j y, por tanto, su tiempo de ejecución es:

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

Por tanto, el tiempo de ejecución del programa se puede expresar como $an + b$, es decir, una función lineal.

Por otro lado, si el arreglo se da en forma decreciente, entonces se presenta el peor caso y, por tanto, se deben comparar todos los elementos $A[j]$ con cada elemento del subarreglo. En este caso $t_j = j$, para $j=2,3,\dots,n$, las sumatorias se expresan de la siguiente manera:

$$\sum_{j=2}^n t_j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (t_j - 1) = \frac{n(n-1)}{2}$$

Por lo tanto, para el peor caso, el tiempo de ejecución del algoritmo de ordenamiento es:

$$\begin{aligned} T(n) &= c_1n + c_2(n - 1) + c_3(n - 1) + c_4 (((n*(n+1))/2)-1) + \\ &\quad + c_5 ((n*(n-1))/2) + c_6 ((n*(n-1))/2) + c_7(n - 1) \\ &= (c_4/2 + c_5/2 + c_6/2)*n^2 + \\ &\quad + (c_1+c_2+c_3+c_4/2+c_5/2+c_6/2+c_7)*n + \\ &\quad + (c_2+c_3+c_4+c_7) \end{aligned}$$

De aquí se obtiene que el tiempo de ejecución del peor caso se puede expresar como $an^2 + bn + c$, es decir, una función cuadrática.

2.4.4 Predicción.

Como ya se había mencionado, el análisis de algoritmos se ha convertido en un sinónimo de predicción. Un buen análisis permite predecir los recursos que un algoritmo va a ocupar en tiempo de ejecución.

Determinar la complejidad temporal y/o espacial de un algoritmo puede ayudar a predecir de forma confiable el tiempo de ejecución y el espacio en memoria que se requiere para diferentes instancias sin necesidad de implementar el algoritmo.

Es por eso que se ocupa el modelo de computación RAM para ponderar los recursos utilizados durante el proceso y, por tanto, este modelo es muy utilizado para predecir el rendimiento de un equipo real.

2.4.5 Criterios de medición.

Una algoritmo eficaz no necesariamente es un algoritmo eficiente, por tanto, cuando un algoritmo logra cumplir su objetivo (resolver el problema) es necesario ponderar su rendimiento y/o comportamiento.

Los criterios para medir su eficiencia se centran en su simplicidad y en el uso eficiente de los recursos (modelo RAM).

El uso eficiente de los recursos suele medirse en función del espacio utilizado y del tiempo de ejecución.

Por lo tanto, el tiempo de ejecución de las instancias dadas (datos de entrada), el código generado por el compilador (código obj), la velocidad del procesador y, por supuesto, la complejidad del algoritmo.

```
0 FUNC Buscar(a[]:ENTERO, c:ENTERO) DEV ENTERO
1     j←1:ENTERO;
2     MIENTRAS (a[j]<c) AND (j<n) HACER
3         j←j+1
4     FIN_MIENTRAS
5     SI a[j]=c ENTONCES
6         DEV j
7     FIN_SI
8     DE_LO CONTRARIO
9         DEV 0
10    FIN_DLC
11 FIN_FUNC;
```

Para contabilizar la eficiencia del algoritmo es preciso calcular el número de operaciones elementales que éste realiza:

- La línea 1, $j:=1$, ejecuta dos operaciones elementales (declaración y asignación).
- La línea 2, MIENTRAS ($a[j] < c$) AND ($j < n$) HACER, efectúa 4 operaciones elementales (dos comparaciones, un acceso al vector y una operación lógica).

- La línea 3, $j:=j+1$, ejecuta un incremento y una asignación (2 operaciones elementales).
- La línea 5, SI $a[j]=c$ ENTONCES, valida una condición y un acceso al vector (2 operaciones elementales).
- La línea 6, DEV j , ejecuta la operación elemental si la condición se cumple.
- La línea 9 DEV 0, ejecuta la operación elemental si la condición no se cumple.

2.4.6 Instrumentos de software para efectuar mediciones.

El desempeño ideal de un algoritmo se obtiene de manera teórica obteniendo su complejidad y su consumo en la máquina RAM.

Sin embargo, cuando se implementa el algoritmo se deben tener en cuenta los atributos propios del software a utilizar.

Algunas características de software que se deben tomar en cuenta son:

- **Funcionalidad.**
- **Errores durante cierto periodo de tiempo.**
- **Capacidad de respuesta frente a errores externos.**
- **Nivel de seguridad.**
- **Errores en la codificación o diseño del sistema.**
- **Líneas de código.**

En 1991 un misil iraquí impactó en un cuartel estadounidense, matando a 28 soldados.

La investigación demostró que el misil fue detectado por el radar, pero el sistema antimisiles Patriot no entró en funcionamiento.

El sistema Patriot tenía un error de redondeo (al calcular $1/10$) que provocaba un retraso de 0.000000095 segundos cada segundo. Tras 100 horas en funcionamiento, el sistema antimisiles acumulaba un error 0.34 segundos, tiempo suficiente para que un misil Scud iraquí avanzase 600 metros y escapase de la detección.

2.4.7 Eficiencia.

La eficiencia (del latín *efficientia*) es un término que se refiere a la ausencia de recursos productivos ociosos, es decir, que los recursos se están utilizando de la mejor manera posible.

La eficiencia de un algoritmo se mide en función del uso de los recursos requeridos para su ejecución (tanto en tiempo como en espacio) con base en el tamaño de entrada, es decir, $T(n)$.



2.5 Estrategias para la construcción de algoritmos.

2.5 Estrategias para la construcción de algoritmos.

Tener una base sólida de conocimientos y técnicas algorítmicas es una característica que separa a los diseñadores expertos de los novatos.

Con las capacidades tecnológicas actuales se puede cumplir con las tareas solicitadas sin saber mucho de algoritmos, pero con buenos conocimientos de algoritmos se puede hacer mucho más.

2.5.1 Selección de métodos basados en criterios de eficiencias.

Cuando se analiza un algoritmo se puede llegar a tener varias soluciones y se debe elegir cuál es la adecuada. El criterio a tomar en cuenta debe ser la eficiencia tanto en tiempo como en espacio o, incluso, con un análisis asintótico de la función.

Ejemplo 9

Supóngase que se quiere comparar el procesamiento de una computadora rápida A corriendo un algoritmo de inserción ordenada contra una computadora más lenta B ejecutando una unión ordenada de elementos.

Cada algoritmo debe ordenar un arreglo de un millón de elementos. La computadora A ejecuta 1000 millones de instrucciones por segundo y la computadora B ejecuta 10 millones de instrucciones por segundo (A es 100 veces más rápida que B).

Ejemplo 9

Para hacer más dramático el asunto, supóngase que el programador más astuto del mundo codificó el algoritmo de inserción en lenguaje máquina para la computadora A y el código resultante para requiere $2n^2$ instrucciones para ordenar n números ($c_1 = 2$).

Por otro lado, el código de la unión ordenada fue realizado por un programador promedio usando un lenguaje de alto nivel con un compilador ineficiente y toma $50n \log n$ instrucciones ($c_2 = 50$).

Ejemplo 9

Si se ordenan 1 millon de elementos, los tiempos que tardan A y B son, respectivamente:

$$\frac{2*(10^6)^2 \text{ [instrucciones]}}{10^9 \text{ [instrucciones/s]}} = 2000 \text{ [s]}$$

$$\frac{50 * 10^6 \lg (10^6) \text{ [instrucciones]}}{10^7 \text{ [instrucciones/s]}} \approx 100 \text{ [s]}$$

Ejemplo 9

Debido a que se utilizó un algoritmo cuyo tiempo de ejecución crece poco a poco, incluso con un compilador débil, el proceso en B se ejecutó 100 veces más rápido que el proceso en A.

2.5.2 Tipos de algoritmos.

Los algoritmos se pueden clasificar con base en la manera en que optimizan el código. Algunos tipos son:

- **Algoritmos voraces o ávidos**
- **Algoritmos de programación dinámica**
- **Algoritmos heurísticos**
- **Algoritmos divide y vencerás**
- **Algoritmos de exploración de grafos**
- **Algoritmos probabilísticos**
- **Algoritmos por transformación**

2.5.2.1 Algoritmos ávidos.

Los algoritmos para optimizar problemas generalmente siguen una secuencia de pasos con un conjunto de decisiones en cada paso.

Un algoritmo ávido siempre ejecuta la decisión que parece mejor en el momento, es decir, crea una decisión local óptima con la esperanza de que esa elección contribuya a la solución óptima general.

Código Huffman

Se refieren a una técnica de compresión ampliamente usada y muy efectiva. Puede llegar a comprimir entre 20 y 90 % dependiendo de las características de los datos a comprimir.

El algoritmo ávido de Huffman utiliza una tabla de frecuencia de ocurrencia de caracteres para crear una manera óptima de representar cada carácter como una cadena binaria.

Supóngase que se tiene un archivo de datos de 1000 caracteres y se desea almacenar de manera compacta. Se puede observar que la frecuencia de caracteres en el archivo es la siguiente:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5

De la tabla anterior se puede observar que solo aparecen 6 caracteres diferentes y que el carácter más repetitivo es 'a' (45,000 veces).

Para representar el archivo de información se considera el diseño de un código binario, donde cada carácter es representado por una cadena binaria única.

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud fija.	000	001	010	011	100	101

Utilizando el código de longitud fija se observa que se necesitan 3 bits para representar los seis caracteres: a=000, b=001, c=010, d=011, e=100, f=101.

Este método requiere de 300,000 bits para codificar el archivo completo.

También se puede considerar un código de longitud variable con la siguiente distribución:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud variable.	0	101	100	111	1101	1100

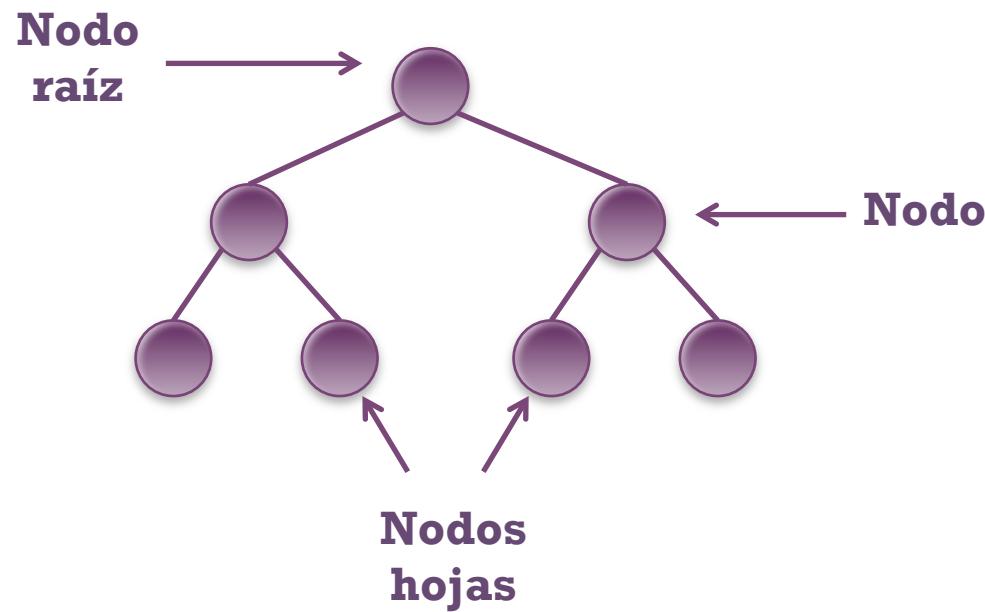
Un código de longitud variable hace un mejor trabajo que uno de longitud fija, debido a que a los caracteres frecuentes les asigna un código de palabra corto y a los caracteres que no tienen tanta frecuencia les asigna un código de palabra largo.

El código de longitud variable requiere $(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000 = 224,000$ bits para codificar el archivo, genera un ahorro de, aproximadamente, el 25%.

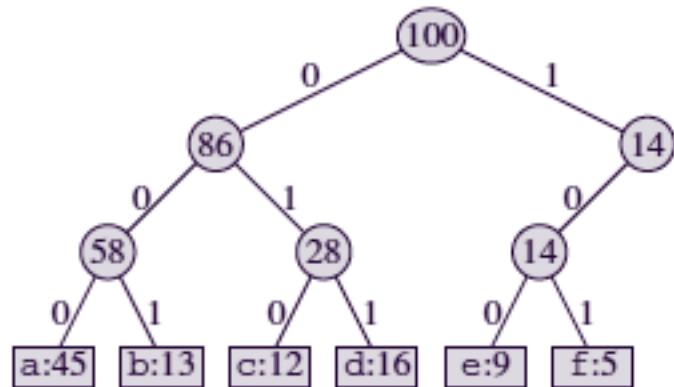
La codificación variable se considera una codificación prefija, ya que ningún código de palabra es prefijo de algún otro código. La codificación prefija simplifica el proceso de decodificación.

Por ejemplo, la cadena 001011101 se puede descodificar como 0.0.101.1101 que resulta en aabe.

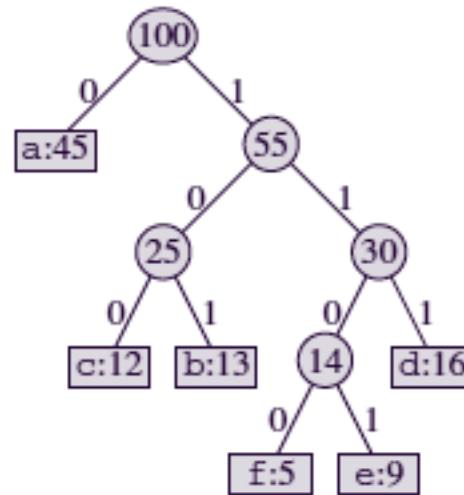
Una representación para el proceso de decodificación es mediante una estructura de árbol. Una estructura de árbol posee un **nodo raíz** (el primer nodo del árbol) y **varios nodos hojas** (los últimos nodos).



A partir de un árbol se puede interpretar el código de palabra de un carácter como la ruta que existe desde el nodo raíz hasta la hoja que contiene el carácter, donde 0 significa ir a la izquierda y 1 significa ir a la derecha.



**Longitud
fija**



**Longitud
variable**

Dado un árbol A que corresponde a un código prefijo, se puede calcular fácilmente el número de bits requeridos para codificar un archivo.

Para cada carácter c en el alfabeto C, se tiene $f(c)$ (frecuencia de aparición del carácter c en el archivo) y $p_A(c)$ (profundidad de la hoja que posee el carácter c dentro del árbol A).

El número de bits que se requieren para condificar el archivo está dado por:

$$B(A) = \sum f(c) p_A(c) \quad \text{con } c \in C$$

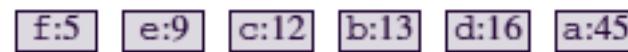
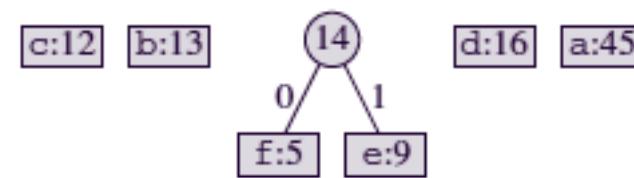
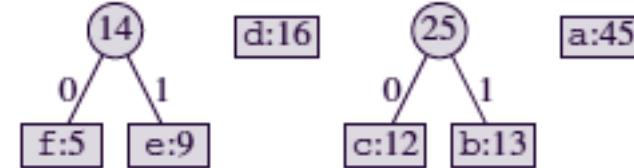
que se define como el costo del árbol A.

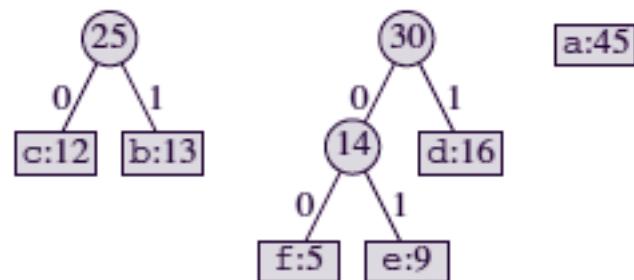
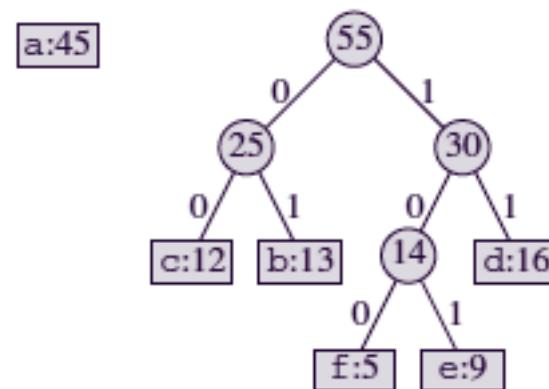
Huffman inventó un algoritmo ávido que construye un código prefijo óptimo.

A continuación se presenta el pseudocódigo del algoritmo de Huffman. Se asume que C es un conjunto de n caracteres y que cada carácter $c \in C$ es un objeto con una frecuencia $f(c)$. El algoritmo crea un árbol A que corresponde a un código óptimo de arriba hacia abajo.

Ejemplo 10

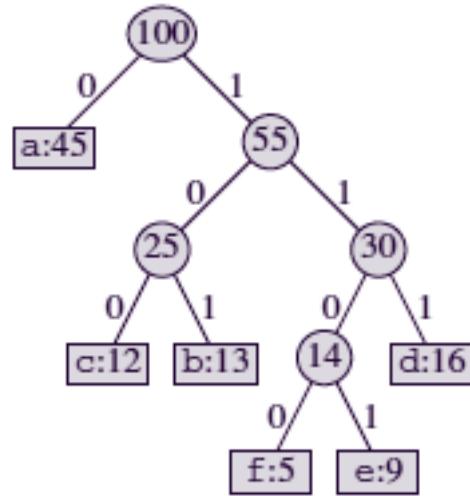
```
1 FUNC HUFFMAN(C)
2     n ← |C|          ** longitud de C
3     Q ← C            ** elementos de C
4     i ← 1             ** índice
5     MIENTRAS i < n-1 HACER
6         crearNodo(z)
7         left[z]← x ← extraerMen(Q)
8         right[z]← y ← extraerMen(Q)
9         f [z]← f [x] + f [y]
10        INSERT(Q, z)
11    FIN_MIENTRAS
12    return EXTRACT-MIN(Q)
13 FIN_FUNC
```

Ejemplo 10**1****2****3**

Ejemplo 10**4****5**

Ejemplo 10

6



Así como lo hizo el algoritmo de Huffman, un algoritmo ávido va buscando la decisión que parece mejor en el momento, esperando con ello optimizar la solución general.

Ejemplo 10

El análisis del tiempo de ejecución del algoritmo de Huffman asume que Q es una pila binaria y, por tanto, para un conjunto C de n caracteres, el ciclo se ejecuta $n-1$ veces y cada operación en la pila lleva un tiempo de $\ln(n)$.

Por tanto, el tiempo de ejecución del algoritmo ávido de Huffman para n caracteres es $O(n \ln (n))$.

2.5.2.2 Algoritmos tipo “divide y vencerás”.

En la práctica, muchos algoritmos poseen una estructura recursiva, es decir, para resolver un problema se mandan llamar ellos mismos una o más veces para resolver subproblemas cada vez más pequeños.

La mayoría de los algoritmos recursivos siguen el enfoque *divide y vencerás*, dividiendo el problema inicial en varios subproblemas similares al inicial pero con instancias de entrada menores, combinando al final las soluciones para generar una solución general.

El paradigma divide y vencerás desarrolla tres pasos en cada nivel recursivo:

- **Divide el problema en un cierto número de subproblemas.**
- **Vence los subproblemas resolviéndolos de manera recursiva. Cuando la instancia del subproblema es lo suficientemente pequeña, simplemente los resuelve de una manera sencilla.**
- **Combina la solución de los problemas hasta generar una solución general.**

Ejemplo 11

El algoritmo de unión ordenada sigue el paradigma divide y vencerás:

- **Divide la secuencia de n elementos a ser ordenados en dos subsecuencias de $n/2$ elementos cada una.**
- **Vence el reto ordenando recursivamente las dos subsecuencias de elementos utilizando unión ordenada.**
- **Combina (une) las dos subsecuencias ordenadas para producir la salida ordenada.**

Ejemplo 11

La operación clave en el algoritmo consiste en unir las dos secuencias ordenadas en un paso combinado.

Para realizar dicha unión se utiliza una función auxiliar $\text{unir}(A, p, q, r)$, donde A es un arreglo, p , q y r son índices del arreglo tales que $p \leq q < r$. La función asume que los subarreglos $A[p, q]$ y $A[q+1, r]$ están previamente ordenados. El proceso une los dos arreglos para formar un subarreglo ordenado que reemplaza a $A[p, r]$.

El procedimiento anterior toma un tiempo $\Theta(n)$ para $n = r - p + 1$.

Ejemplo 11

```
1  FUNC unir(A[]:ENT, p:ENT, q:ENT, r:ENT)
2      n1 ← q - p + 1
3      n2 ← r - q
4      create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
5      MIENTRAS i ← 1 <= n1
6          L[i ] ← A[p + i - 1]
7      MIENTRAS j ← 1 <= n2
8          R[j ]← A[q + j ]
9      L[n1 + 1]←∞
10     R[n2 + 1]←∞
11     i ← 1
12     j ← 1
13     MIENTRAS k ← p <= r
14         SI L[i ] ≤ R[ j ]
15             A[k] ← L[i ]
16             i ← i + 1
17             DE_LO CONTRARIO
18             A[k] ← R[ j ]
19             j ← j + 1
20     FIN_DLC
21 FIN_FUNC
```

Ejemplo 11

A continuación se muestran posibles transiciones para un arreglo dado:

A	8	9	10	11	12	13	14	15	16	17	\dots
	\dots	2	4	5	7	1	2	3	6	\dots	
		k									
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i										
R	1	2	3	4	5	∞					
	1	2	3	6	∞						
	j										

(a)

A	8	9	10	11	12	13	14	15	16	17	\dots
	\dots	1	4	5	7	1	2	3	6	\dots	
		k									
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i										
R	1	2	3	4	5	∞					
	1	2	3	6	∞						
	j										

(b)

Ejemplo 11

A	8	9	10	11	12	13	14	15	16	17	
	...	1	2	5	7	1	2	3	6	...	
				k							
L	1	2	3	4	5						
	2	4	5	7	∞						
						i					
R	1	2	3	4	5						
	1	2	3	6	∞						
						j					

(c)

A	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	7	1	2	3	6	...	
				k							
L	1	2	3	4	5						
	2	4	5	7	∞						
						i					
R	1	2	3	4	5						
	1	2	3	6	∞						
						j					

(d)

A	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	3	1	2	3	6	...	
				k							
L	1	2	3	4	5						
	2	4	5	7	∞						
						i					
R	1	2	3	4	5						
	1	2	3	6	∞						
						j					

(e)

A	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	3	4	2	3	6	...	
				k							
L	1	2	3	4	5						
	2	4	5	7	∞						
						i					
R	1	2	3	4	5						
	1	2	3	6	∞						
						j					

(f)

Ejemplo 11

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	3	6	...	k

L	1	2	3	4	5	∞
	2	4	5	7	∞	

(g)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	6	...	k

L	1	2	3	4	5	∞
	2	4	5	7	∞	

(h)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	7	...	k

L	1	2	3	4	5	∞
	2	4	5	7	∞	

(i)

Ejemplo 11

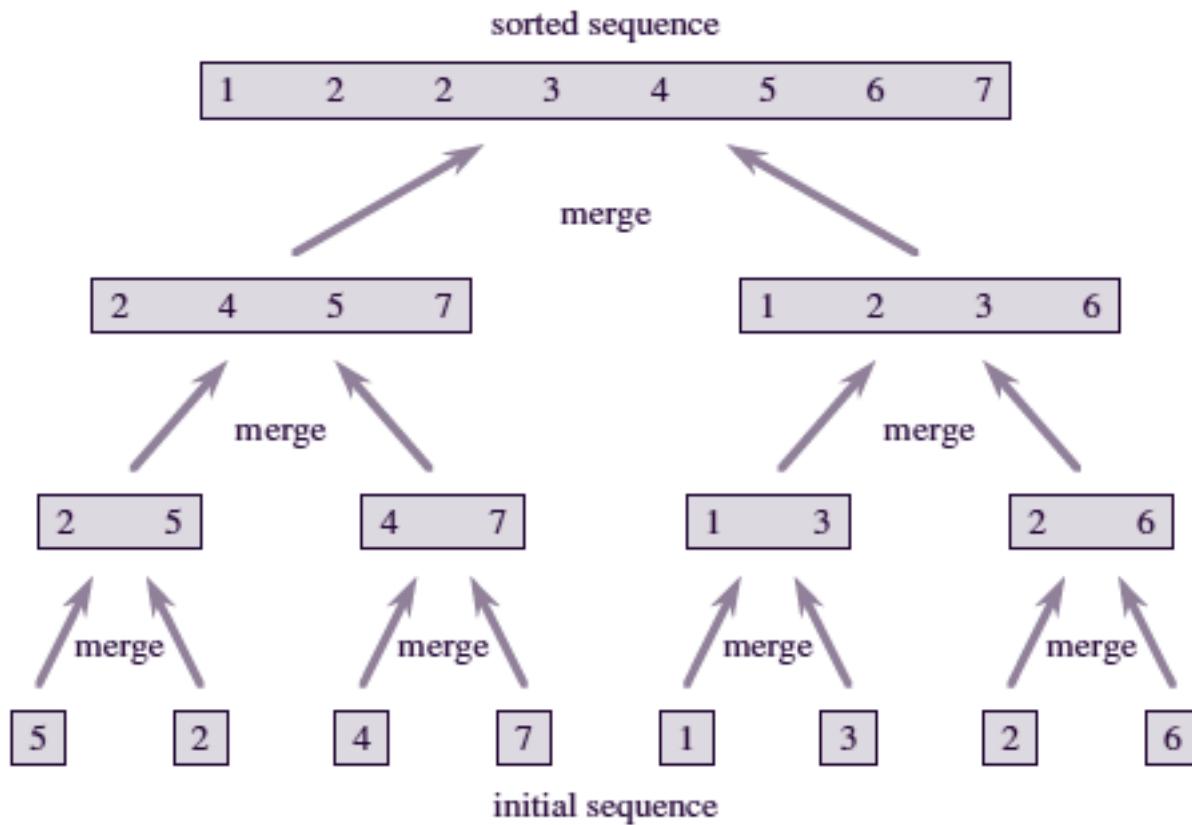
Ahora, volviendo al algoritmo de `unionOrdenada(A, p, r)`, éste recibe como parámetros un arreglo de elementos, un índice inferior y un índice superior, y acomoda los elementos en un subarreglo `A[p, r]`.

Si $p \geq r$, el arreglo tendrá como máximo un elemento y, por tanto, ya se encuentra ordenado. De lo contrario, se divide el arreglo `A[p, r]` en `A[p, q]` y `A[q+1, r]` que contienen, respectivamente, la mitad de los elementos del arreglo ($n/2$).

Ejemplo 11

El algoritmo que permite realizar la unión ordenada de elementos es el siguiente:

```
1 FUNC unionOrdenada (A, p, r)
2   SI p < r ENTONCES
3     q ← (p + r)/2
4     unionOrdenada (A, p, q)
5     unionOrdenada (A, q + 1, r)
6     unir (A, p, q, r)
7   FIN_SI
8 FIN_FUNC
```

Ejemplo 11

2.5.2.3 Algoritmos backtrack.

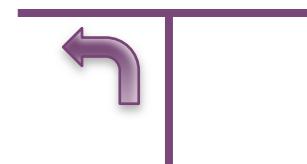
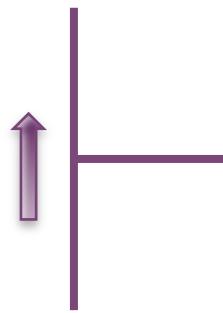
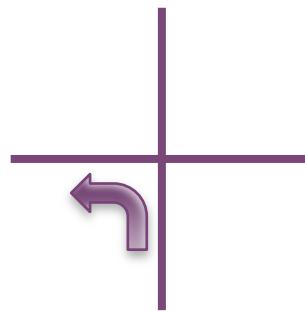
Backtracking o búsqueda hacia atrás es una técnica de programación para hacer búsquedas sistemáticas a través de todas las configuraciones posibles dentro de un espacio de búsqueda.

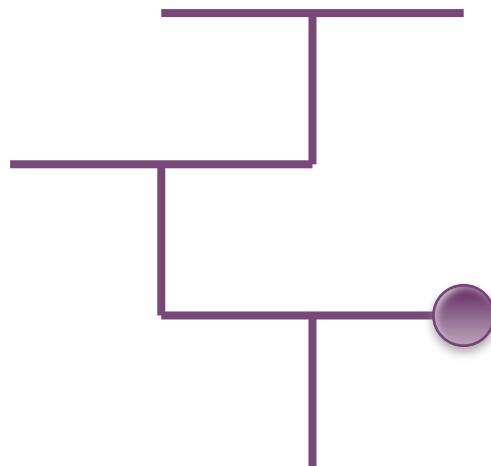
Las configuraciones pueden representar todos los posibles arreglos de objetos (permutaciones) o todos las posibles maneras para crear una colección de objetos (subconjuntos).

Ejemplo 12

Una aplicación práctica de un tipo de algoritmo backtrack se presenta al tratar de resolver un laberinto a través de un robot seguidor de línea.

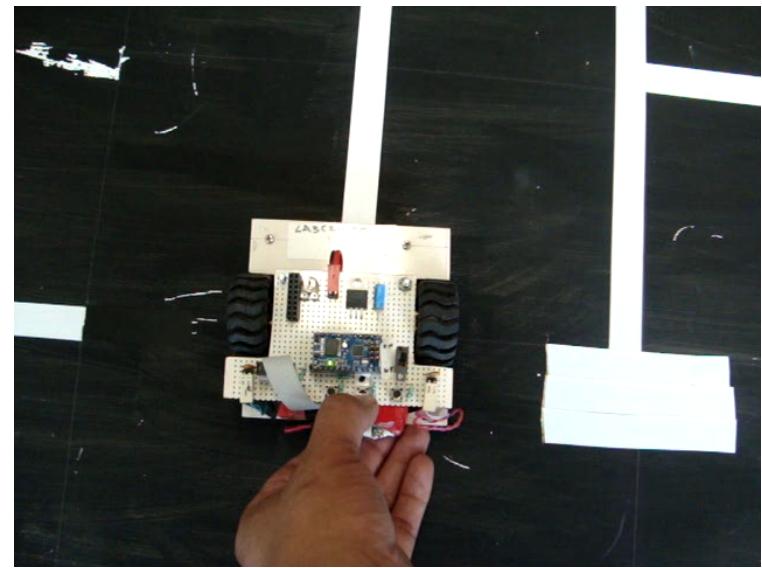
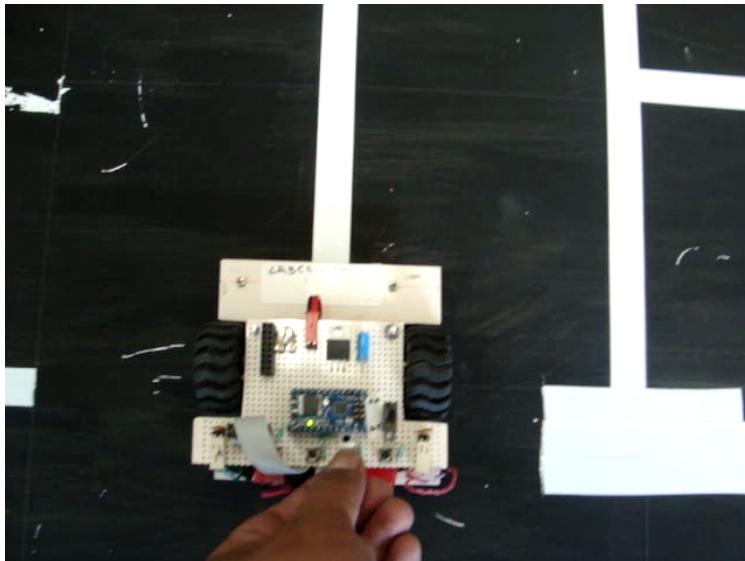
Suponiendo que se tiene un robot seguidor de línea que utiliza la lógica izquierda, esto es, en cualquier intersección que el robot encuentre va a tomar el rumbo, preferentemente, hacia la izquierda, se tienen las siguientes posibilidades:

Ejemplo 12**B****D****I****T****L****X****7**

Ejemplo 12

T
T D
T D T
T D T B
T D T B L
T D D
T D D I
T D D I T
T D D I T B
T D D I T B L
T D D I D
T D D I D B
T D D I D B 7
T D D I B
T D D I B D
T D D B
T D D B 7
T D B
T D B I
T B L
D

Ejemplo 12



2.5.2.4 Algoritmos de búsqueda local.

Un algoritmo de búsqueda local utiliza la vecindad alrededor de cada elemento en el espacio solución. Se puede pensar en cada elemento x en el espacio de soluciones como un vértice con una unión directa (x, y) , con cada candidato solución y que es vecino de x .

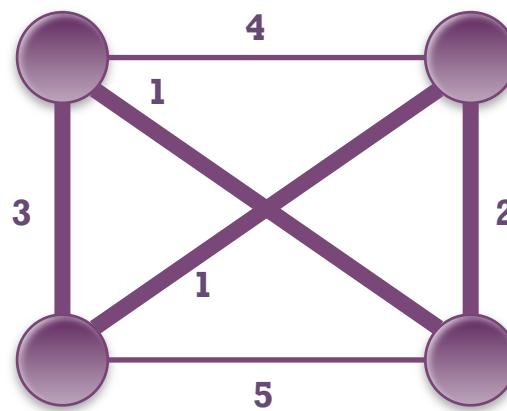
Ejemplo 13

Se tiene un vendedor ambulante que desea visitar n ciudades una sola vez y regresar a la ciudad donde partió.

Existe un costo entero $c(x, y)$ al viajar de la ciudad x a la ciudad y. El vendedor ambulante desea realizar el tour con el menor costo posible, donde el costo total está dado por la suma de todos los costos del tour.

Ejemplo 13

El tour va a depender de la gráfica dada, es decir, del número de nodos así como de las uniones entre ellos.



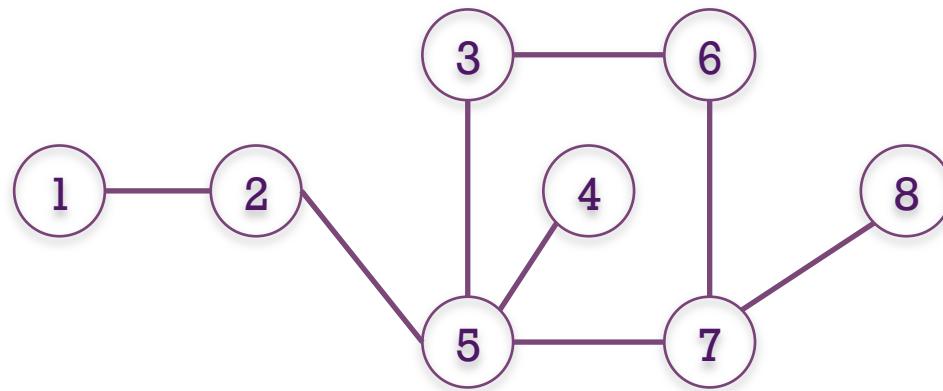
2.5.2.5 Algoritmos por transformaciones.

Un algoritmo por transformaciones permite optimizar problemas sobre espacios de búsqueda exponenciales, realizando transformaciones locales.

La idea de este tipo de algoritmos es crear una solución S, mediante heurística (estrategia sistemática para realizar de forma inmediata innovaciones positivas) y se aplica la solución a S realizando transformaciones. Este proceso se repite hasta que ninguna transformación mejore a S

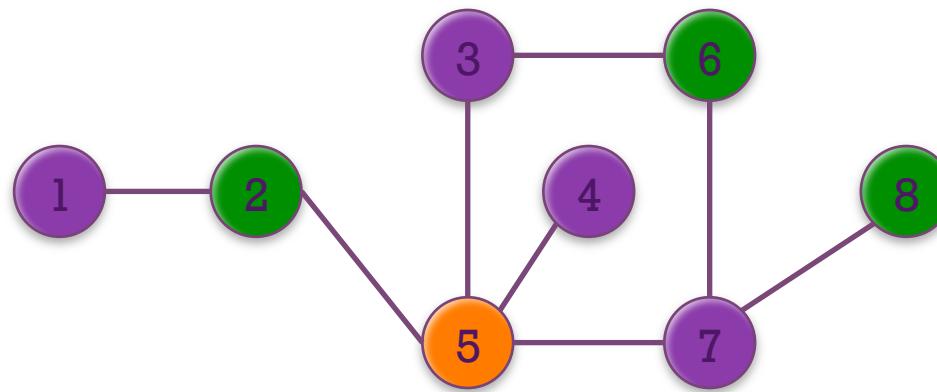
Ejemplo 14

Supóngase que se posee el siguiente grafo:



Ejemplo 14

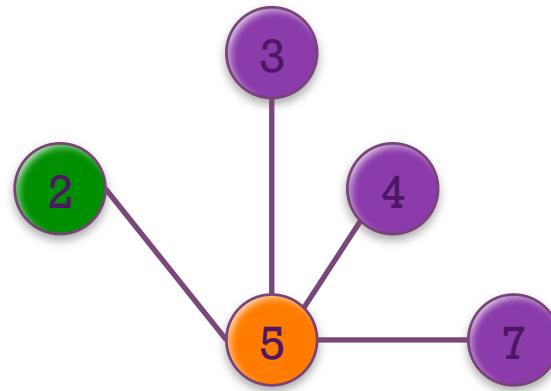
Al grafo anterior se le aplica color de manera secuencial y el resultado es el siguiente:



Plantear las posibles transformaciones adyacentes al vértice 5 para que el grafo quede únicamente con dos colores.

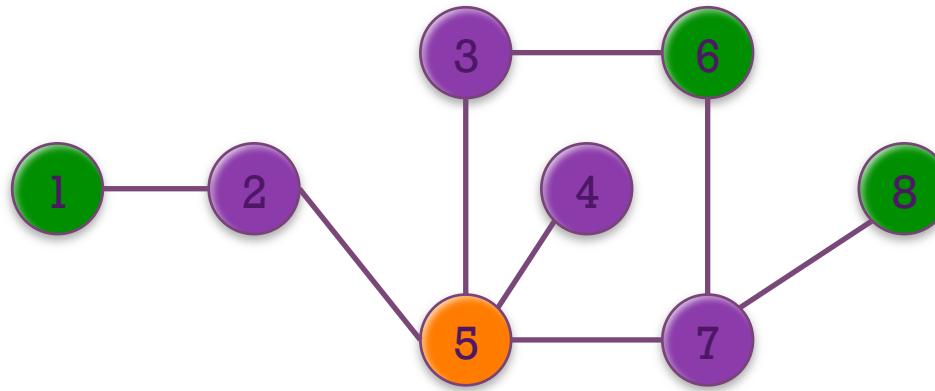
Ejemplo 14

Se deben plantear las posibles transformaciones a partir de los vértices adyacentes a 5, es decir, 3, 4 y 7 por un lado y 2 por el otro lado. Se tienen, por tanto, dos posibles transformaciones.



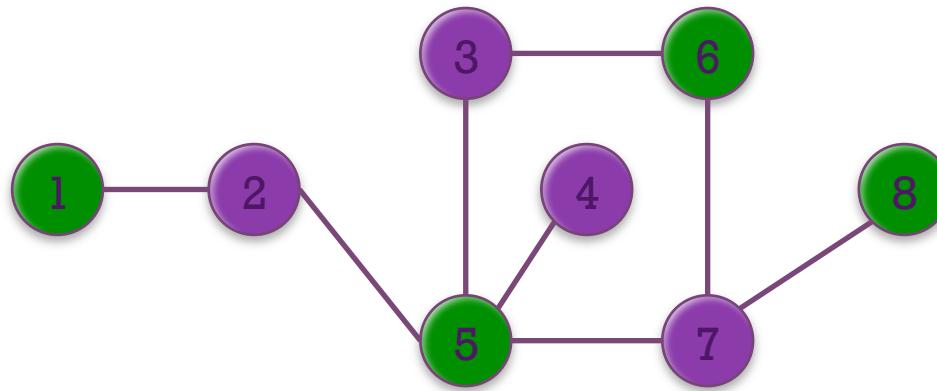
Ejemplo 14

Para la primer transformación se toma como pivote 2 y se le aplican transformaciones a sus adyacentes. Por tanto, el vértice 2 intercambia colores con el vértice 1 y viceversa.



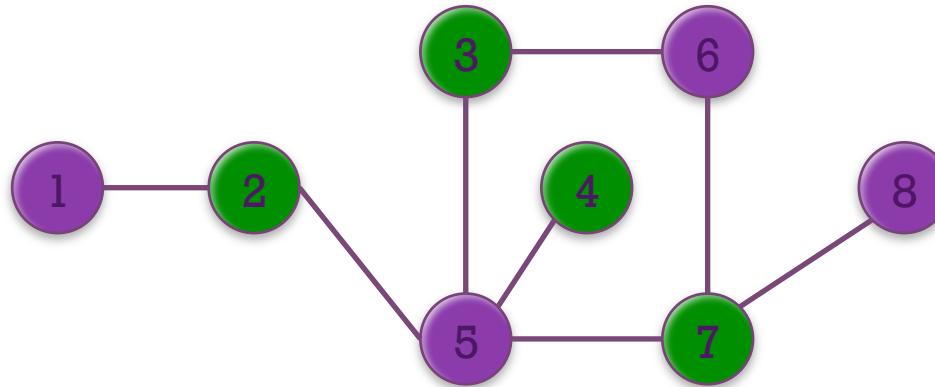
Ejemplo 14

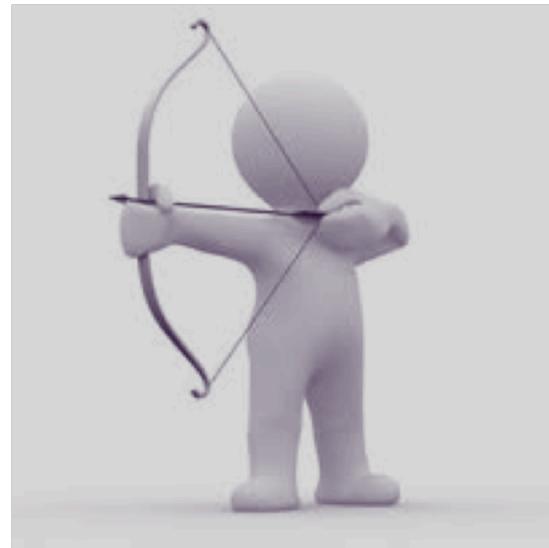
Posteriormente, si la transformación anterior es válida, se colorea el vértice 5 con el color anterior de dos y se encuentra la solución.



Ejemplo 14

La solución también se pudo haber realizado con los nodos 3, 4 y 7, coloreando éstos con 2 y, si es una transformación válida, se colorea 5 con 1.





2.6 Definición, ejemplos, diseño, implementación, corrección, eficiencia y complejidad de algoritmos.

2.6 Definición, ejemplos, diseño, implementación, corrección, eficiencia y complejidad de algoritmos.

A partir de un problema dado es posible implementar una solución (algoritmo) eficiente en tiempo y espacio mediante un análisis matemático del mismo.

Teniendo en cuenta el problema de multiplicar dos números sin usar el operador * se tiene una sucesión de

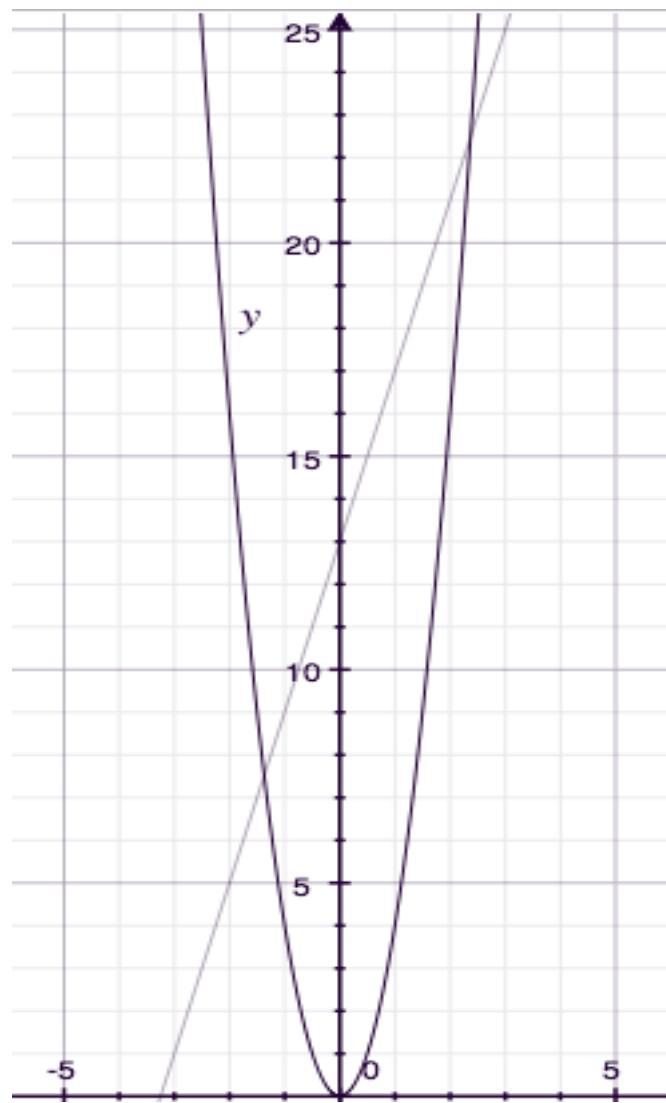
1.	INICIO	
2.	X, Y, Z, TMP: ENTERO	3
3.	HACER Z = 0	1
4.	ESCRIBIR “Ingrese un número entero.”	1
5.	LEER X	1
6.	ESCRIBIR “Ingrese otro número entero.”	1
7.	LEER Y	1
8.	Si Y < X	1
9.	HACER TMP = X	1
10.	HACER X = Y	1
11.	HACER Y = TMP	1
12.	TERMINA_SI	
13.	MIENTRAS X > 0	$\sum_{j=n}^0 t_j$
14.	HACER Z = Z + Y	2
15.	HACER X = X - 1	2
16.	FIN_MIENTRAS	
17.	ESCRIBIR “El resultado es: “ Z	1
18.	FIN	

El tiempo de ejecución del algoritmo anterior es:

$$f(n) = \sum_{j=n}^0 4*t_j + 13$$

$$f(n) = 4*n+13$$

**En general, se observa una función lineal creciente.
La función anterior se puede expresar en notación O.**

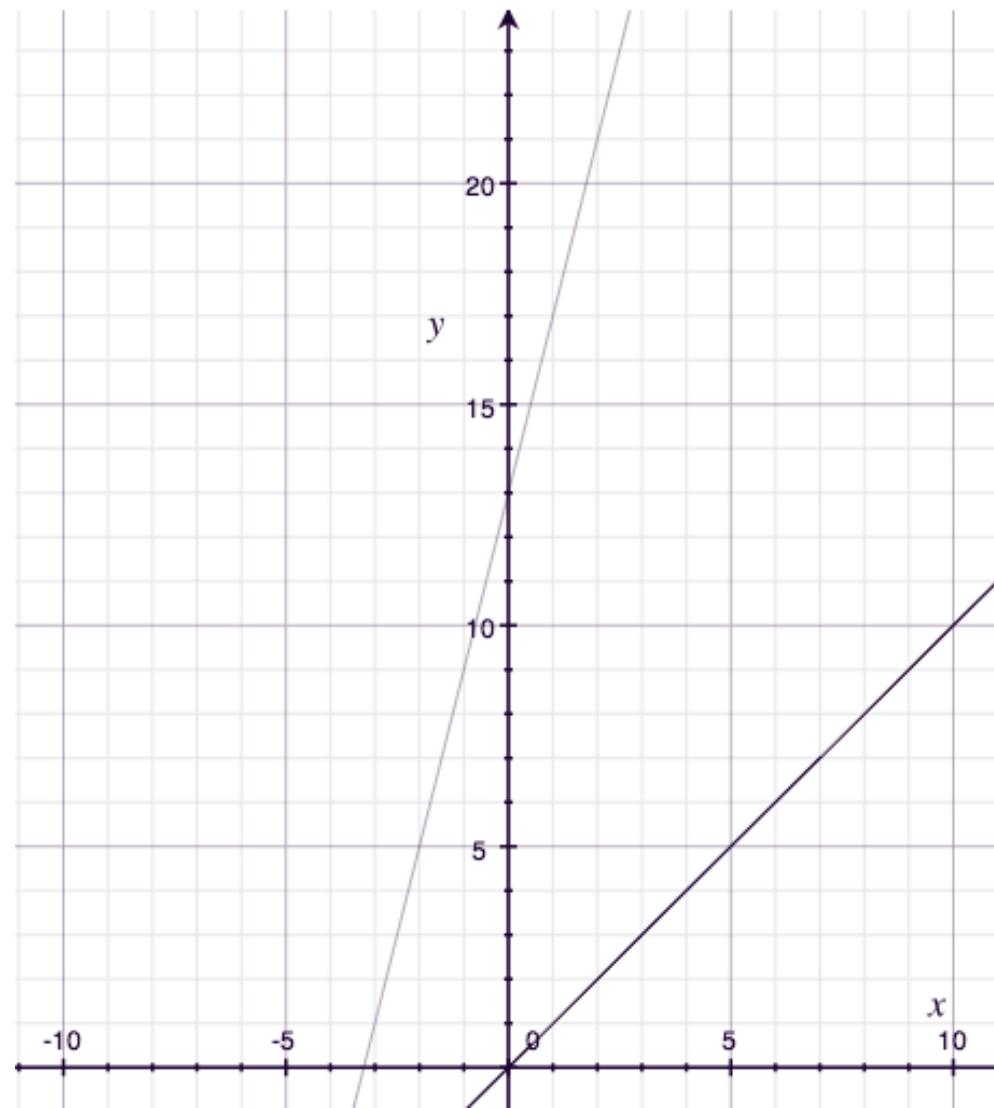


$$4n+13 = O(n^2)$$

Con $c = 4$ y $n_0 = 3$

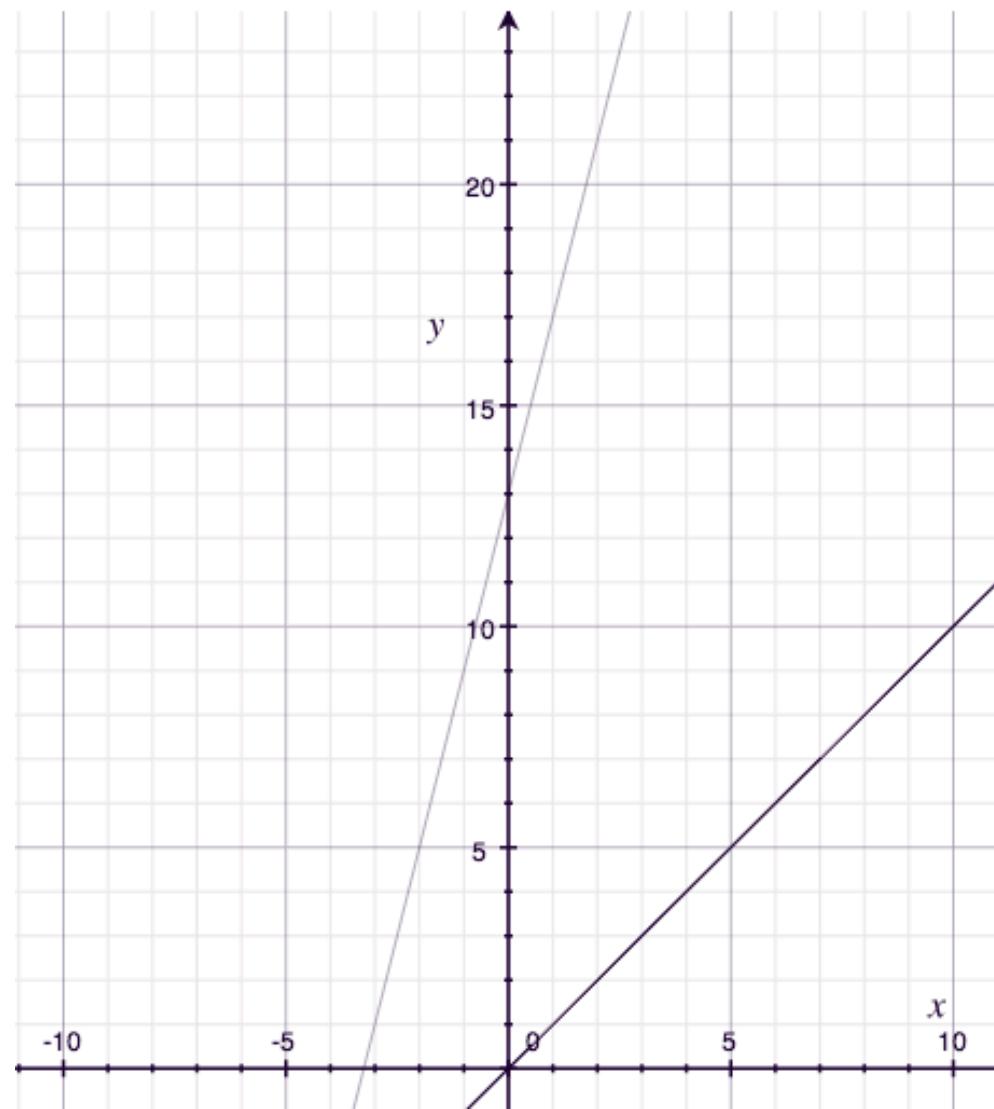
$$4n+13 = \Omega(n)$$

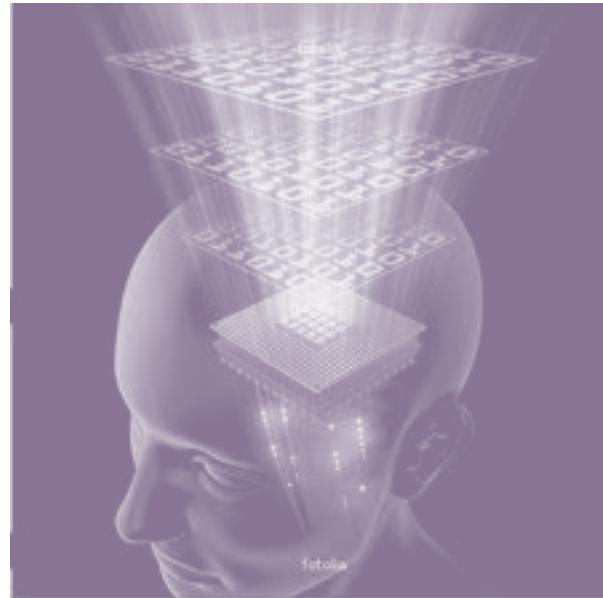
Con $0 \geq c \leq 4$ y $n_0 = 0$



$$4n+13 = \Theta(n)$$

$$\begin{aligned}c_1 &\geq 4 \\0 &\geq c_2 \leq 4\end{aligned}$$





2.7 Análisis y diseño avanzado de algoritmos.

2.7 Análisis y diseño avanzado de algoritmos.

El análisis teórico de un problema puede derivar en una compresión más profunda de su estructura y, con ello, un algoritmo más eficiente.

El conocimiento de diversos tipos de algoritmos permite observar un problema desde diversos panoramas para así encontrar la solución más adecuada.

Regresando al problema del producto de dos números $a*b$, ¿es posible mejorar su tiempo de ejecución, es decir, transformar el comportamiento $O(n)$ a uno más eficiente similar a $O(\log n)$?

Si se realiza un análisis matemático del problema se tiene que el producto de dos números está dado por la suma de uno las veces del otro, es decir:

$$a * b = \overbrace{(b + b + \dots + b)}^{a \text{ veces}}$$

$$a * b = \overbrace{(b + b + \dots + b)}^{a/2 \text{ veces}} + \overbrace{(b + b + \dots + b)}^{a/2 \text{ veces}}$$

Por lo tanto, el producto de dos números se puede representar como:

$$\overbrace{a * b = 2(b + b + \dots + b)}^{a/2 \text{ veces}}$$

La fórmula anterior aplica si el número es par. Si el número es impar, se puede omitir un elemento de b, de tal manera que las sumas se realicen un número par de veces, para al final sumar el elemento faltante, es decir:

$$a * b = \overbrace{2(b + b + \dots + b)}^{a/2 \text{ veces}} + b$$

Utilizando un algoritmo del tipo divide y vencerás es posible realizar el cálculo iterativo (o recursivo) del problema anterior. Lo último que hay que tener en cuenta es el caso base o caso final, en este caso el número de iteraciones debe ser mayor a cero.

A continuación se presenta un algoritmo más eficiente para realizar la multiplicación de manera recursiva.

```
FUNC multRec(x: ENTERO, y ENTERO) DEV ENTERO
    SI (x = 0) ENTONCES
        DEV 0;
    FIN_SI
    DE_LO CONTRARIO
        SI ((x%2) = 0) ENTONCES
            DEV 2*(multRec(x/2,y));
        FIN_SI
        DE_LO CONTRARIO
            return y+2*multRec((x-1)/2,y);
        FIN_DLC
    FIN_DLC
FIN_FUNC
```

2 Análisis y diseño de algoritmos

Objetivo: Aplicar diversas técnicas para el análisis y el diseño de algoritmos orientados a la solución de problemas computacionales.

2.1 Fundamentos de algorítmica.

2.2 Algorítmica básica.

2.3 Complejidad.

2.4 Análisis de algoritmos.

2.5 Estrategias para la construcción de algoritmos.

2.6 Definición, ejemplos, diseño, implementación, corrección, eficiencia y complejidad de algoritmos.

2.7 Análisis y diseño avanzado de algoritmos.