Tecnológico de Monterrey.

Campus Querétaro.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

1

Técnicas de diseño de algoritmos

Antecedentes

- ¿Qué son las técnicas de diseño de algoritmos?
- ¿Por qué vale la pena el estudio de algoritmos?
- ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en la solución de problemas y su rendimiento?

Algoritmos (Es un procedimiento para resolver un problema particular en un número finito de pasos para una entrada definida)

- Método de implementación
  - Recursión o iteración
  - Exacto o Aproximado
  - Algoritmos en serie o en paralelo o distribuidos
- Método de diseño
  - Divide y vencerás
  - Programación dinámica
  - Algoritmos avaros
  - Programación lineal
  - Reducción (Transformar y Conquistar)
- Otras clasificaciones
  - Algoritmos aleatorios
  - Clasificación por complejidad
  - Clasificación por área de investigación
  - Backtracking
  - Ramificación y poda

Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-techniques/

2

Técnicas de diseño de algoritmos

Antecedentes

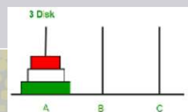
Método de implementación
- Recursión o iteración
- Exacto o Aproximado
- Algoritmos en serie o en paralelo o distribuidos

Recursivo / Iteración

Recursivo
Se llama a sí mismo una y otra vez hasta que se logra una condición base.

Iterativos
Usan bucles y/o estructuras de datos como pilas, colas para resolver cualquier problema.

Cada solución recursiva se puede implementar como una solución iterativa y viceversa.



Scheme **Recursividad *vs* iteratividad**

Ejemplo: ➢ **Contar el número de letras en una palabra**

```
(define (cuenta pal)
  (if (cade_vacia? pal)
    0
   (1+ (cuenta (resto_cade pal))) ))
```

Proceso:

```
(cuenta 'abcdef)
(1+ (cuenta 'bcdef))
(1+ (1+ (cuenta 'cdef)))
(1+ (1+ (1+ (cuenta 'def))))
(1+ (1+ (1+ (1+ (cuenta 'ef)))))
(1+ (1+ (1+ (1+ (1+ (cuenta 'f))))))
(1+ (1+ (1+ (1+ (1+ (1+ (cuenta "")))))))
(1+ (1+ (1+ (1+ (1+ (1+ 0))))))
(1+ (1+ (1+ (1+ (1+ 1)))))
(1+ (1+ (1+ (1+ 2))))
(1+ (1+ (1+ 3)))
(1+ (1+ 4))
(1+ 5)
6
```

recursivo

```
(define (cuenta-cuantas pal resultado)
(if (cade_vacia? pal)
  resultado
  (cuenta-cuantas (resto_cade pal) (1+ res
```

```
(define (cuenta pal)
 (cuenta-cuantas pal 0) )
```

Proceso:

```
(cuenta 'abcdef)
(cuenta-cuantas 'abcdef 0)
(cuenta-cuantas 'bcdef 1)
(cuenta-cuantas 'cdef 2)
(cuenta-cuantas 'def 3)
(cuenta-cuantas 'ef 4)
(cuenta-cuantas 'f 5)
(cuenta-cuantas "" 6)
6
```

iterativo

Algorithms Design Techniques,

Material tomado del curso de
Lenguajes de programación 20

3

Técnicas de diseño de algoritmos

Antecedentes

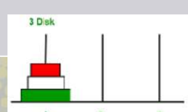
Método de implementación
- Recursión o iteración
- Exacto o Aproximado
- Algoritmos en serie o en paralelo o distribuidos

Recursivo / Iteración

Recursivo
Se llama a sí mismo una y otra vez hasta que se logra una condición base.

Iterativos
Usan bucles y/o estructuras de datos como pilas, colas para resolver cualquier problema.

Cada solución recursiva se puede implementar como una solución iterativa y viceversa.

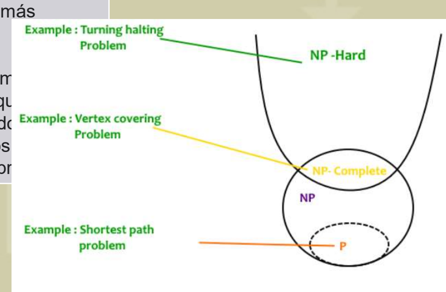


Exacto / Aproximado

Exacto
Algoritmos capaces de encontrar una **solución óptima** para cualquier problema.

Aproximación
Se utilizan para todos aquellos problemas, donde no es posible encontrar la solución más optimizada.

Los algoritmos aproxim
el tipo de algoritmos qu
encuentran el resultado
resultado medio de los
subresultados de un pr



Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-tech

4

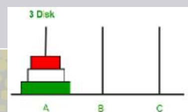
Técnicas de diseño de algoritmos

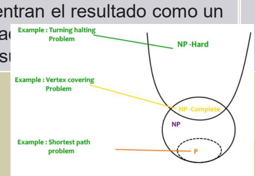
Antecedentes

Método de implementación

- Recursión o iteración
- Exacto o Aproximado
- Algoritmos en serie o en paralelo o distribuidos

Recursivo / Iteración	Exacto / Aproximado	Serie, paralelo o distribuidos
Recursivo Se llama a sí mismo una y otra vez hasta que se logra una condición base. Iterativos Usan bucles y/o estructuras de datos como pilas, colas para resolver cualquier problema. Cada solución recursiva se puede implementar como una solución iterativa y viceversa.	Exacto Algoritmos capaces de encontrar una **solución óptima** para cualquier problema. Aproximación Se utilizan para todos aquellos problemas, donde no es posible encontrar la solución más optimizada. Los algoritmos aproximados son el tipo de algoritmos que encuentran el resultado como un resulta subresu	Algoritmos en serie Se ejecuta una instrucción a la vez Algoritmos en paralelo Se divide el problema en subproblemas y se ejecutan en diferentes procesadores. Si los algoritmos paralelos se distribuyen en diferentes máquinas, se conocen como algoritmos distribuidos.





Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-techniques/

5

Técnicas de diseño de algoritmos

Antecedentes

Método de implementación

- Recursión o iteración
- Exacto o Aproximado
- Algoritmos en serie o en paralelo o distribuidos

Serie, paralelo o distribuidos
Algoritmos en serie Se ejecuta una instrucción a la vez Algoritmos en paralelo Se divide el problema en subproblemas y se ejecutan en diferentes procesadores. Si los algoritmos paralelos se distribuyen en diferentes máquinas, se conocen como algoritmos distribuidos.

```
bool esPrimo(int n) {
  if(n < 2)   {
    return false;
  }
  for(int i = 2; i <= sqrt(n); i++)   {
    if(n % i == 0)     {
      return false;
    }
  }
  return true;
}
```

Ejemplo:

La suma de los números primos menores a 10 es: 2 + 3 + 5 + 7 = 17.

Elige un lenguaje de programación y escribe dos versiones de un programa que calcule **la suma de todos los números primos** menores a 5,000,000 (*cinco millones*):

- La primera versión debe ser una implementación convencional que realice el cómputo de manera secuencial.
- La segunda versión debe realizar el cómputo de manera eficiente, puede ser otro algoritmo o puede ser en forma paralela a través de los mecanismos provistos por el lenguaje que elegiste (por ejemplo *places* o la función *pmap*), en este caso, debes procurar paralelizar el código aprovechando todos los núcleos disponibles en tu sistema.

Ambas versiones del programa deben dar 838,596,693,108 como resultado.

Con el fin de que el proceso de cómputo sea más intenso para el CPU, utiliza el siguiente algoritmo:

Algoritmo para determinar si ***n*** es un número primo. Devuelve verdadero o falso.

1. Si *n* es menor que 2, el algoritmo termina devolviendo falso.
2. Para i desde 2 hasta $\lceil \sqrt{n} \rceil$, realiza lo siguiente:
    - El algoritmo termina devolviendo falso si *n* es divisible entre i de manera exacta, de otra se repite el ciclo con el siguiente valor de i.
3. El algoritmo termina devolviendo verdadero si el ciclo del punto anterior concluyó de manera normal.

Mide el tiempo que tarda en ejecutar cada versión y calcula el *speedup* obtenido usando la siguiente fórmula:

Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-techniques/

6

Técnicas de dis

Antecedentes

Método de implementación

Serie, paralelo o distribuidos

Algoritmos en serie
Se ejecuta una instrucción a la vez

Algoritmos en paralelo
Se divide el problema en subproblemas y se ejecutan en diferentes procesadores.

Si los algoritmos paralelos se distribuyen en diferentes máquinas, se conocen como algoritmos distribuidos.

```
bool esPrimo(int n) {
  if(n < 2)   {
    return false;
  }
  for(int i = 2; i <= sqrt(n); i++)   {
    if(n % i == 0)    {
      return false;
    }
  }
  return true;
}
```

Ejemplo:

La suma de

Elige un leng
calcule **la su**
millones):

- La primera
cómputo c
- La segund
otro algori
provistos
en este ca
núcleos di

Ambas versi

Con el fin de
siguiente alg

Algoritmo pa

1. Si *n* es
2. Para i d
   - E
   r
3. El algor
concluy

Mide el tiemp
usando la si

main.py

```python
    for i in range(2, int(math.sqrt(n)) + 1):
      if n % i == 0:
        return False
    return True
  else:
    return False


# print("Number of processors:", mp.cpu_count())
# Tenemos 8 procesadores asi que nuestro pool maximo es 8
#Calcular de manera computacional
start2 = time.time()
with Pool(8) as p:
  p.map(NumPrimConvencional, [100])
end2 = time.time()
tiempoParalelo = (end2 - start2)
print (tiempoParalelo)

#Calcular de manera convencional
start = time.time()
NumPrimConvencional(100)
end = time.time()
tiempoSecuencial = (end-start)
print (tiempoSecuencial)

speedup = (tiempoSecuencial/tiempoParalelo)

print("El speedup fue de: ", speedup)
```

Console Shell

```
1060
0.039505958557128906
1060
0.0003948211669921875
El speedup fue de:  0.0099939649969825
>
```


7

Técnicas de diseño de algoritmos

Antecedentes

Método de diseño

- Divide y vencerás
- Programación dinámica
- Algoritmos avaros o codiciosos
- Programación lineal
- Reducción (Transformar y Conquistar)

Divide y vencerás

Es un enfoque de arriba hacia abajo. Los pasos son:

1) Divide el problema original en un conjunto de subproblemas.

2) Resuelve cada subproblema individualmente, de forma recursiva.

3) Combina la solución de los subproblemas (nivel superior) en una solución de todo el problema original.

Ejemplo: Ordenación.

Programación dinámica

Es un enfoque de abajo hacia arriba, resolvemos todos los pequeños problemas posibles y luego los combinamos para obtener soluciones para problemas mayores.

Reducción (Transformar y Conquistar)

En este método, resolvemos un problema difícil transformándolo en un problema conocido para el cual tenemos una solución óptima.

Ejemplo: algoritmo de selección para encontrar la mediana en una lista implica primero ordenar la lista y luego encontrar el elemento del medio en la lista ordenada.

Algoritmos codiciosos

En cada paso, se toma la decisión de elegir el óptimo local, sin pensar en las consecuencias futuras.

El algoritmo codicioso no siempre garantiza la solución óptima, sin embargo, generalmente produce una solución que tiene un valor muy cercano al óptimo.

Ejemplo: Selección de actividades.

Programación lineal

En la programación lineal, existen desigualdades en términos de entradas y maximización o minimización de algunas funciones lineales de entradas.

Ejemplo: flujo máximo del grafo dirigido.

8

Técnicas de diseño de algoritmos

Antecedentes

Otras clasificaciones

- Algoritmos aleatorios
- Clasificación por complejidad
- Clasificación por área de investigación
- Backtracking
- Ramificación y poda

Algoritmos aleatorios	Complejidad	Por área de investigación
Son algoritmos que toman decisiones aleatorias para soluciones más rápidas. Ejemplo: algoritmo de ordenación rápida aleatorio.	Algoritmos que se clasifican en función del tiempo necesario para obtener una solución a cualquier problema por tamaño de entrada.	En CS cada campo tiene sus propios problemas y necesita algoritmos eficientes. Ejemplo: algoritmo de clasificación, algoritmo de búsqueda, aprendizaje automático, etc.

Backtracking	Ramificación y poda
El algoritmo de retroceso intenta cada posibilidad hasta que encuentran la correcta. Es una búsqueda en profundidad del conjunto de posibles soluciones. Durante la búsqueda, si una alternativa no funciona, retroceda al punto de elección, el lugar que presentó diferentes alternativas, e intente la siguiente alternativa.	En este algoritmo, un subproblema dado, que no puede acotarse, debe dividirse en al menos dos nuevos subproblemas restringidos.

Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-techniques/

9

Técnicas de diseño de algoritmos

Antecedentes

- ¿Qué son las técnicas de diseño de algoritmos?
- ¿Por qué vale la pena el estudio de algoritmos?
- ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en la solución de problemas y su rendimiento?

Algoritmos

(Es un procedimiento para resolver un problema particular en un número finito de pasos para una entrada definida)

- Método de implementación
  - Recursión o iteración
  - Exacto o Aproximado
  - Algoritmos en serie o en paralelo o distribuidos
- Método de diseño
  - Divide y vencerás
  - Programación dinámica
  - Algoritmos avaros
  - Programación lineal
  - Reducción (Transformar y Conquistar)
- Otras clasificaciones
  - Algoritmos aleatorios
  - Clasificación por complejidad
  - Clasificación por área de investigación
  - Backtracking
  - Ramificación y poda

Algorithms Design Techniques, https://www.geeksforgeeks.org/algorithms-design-techniques/

10

Técnicas de diseño de algoritmos

Antecedentes

- ¿Qué son las técnicas de diseño de algoritmos?
- ¿Por qué vale la pena el estudio de algoritmos?
- ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en la solución de problemas y su rendimiento?

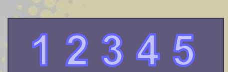
11

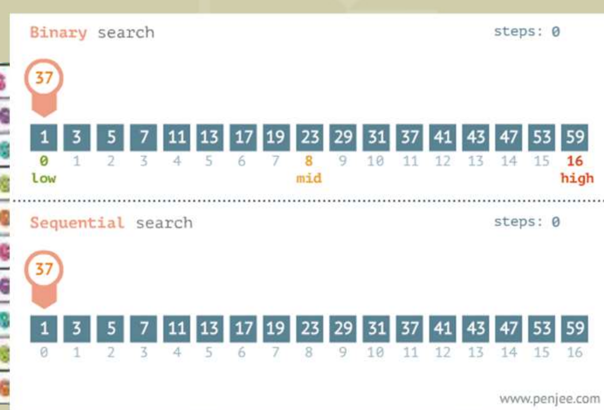
Técnicas de diseño de algoritmos

Antecedentes

- ¿Qué son las técnicas de diseño de algoritmos?
- ¿Por qué vale la pena el estudio de algoritmos?
- ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en la solución de problemas y su rendimiento?

Ejercicio: Adivina, adivinador…





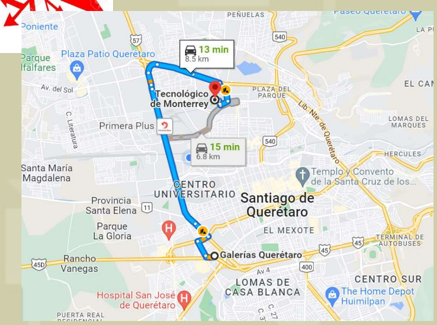
12

Técnicas de diseño de algoritmos

Antecedentes

- ¿Qué son las técnicas de diseño de algoritmos?
- ¿Por qué vale la pena el estudio de algoritmos?
- ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en la solución de problemas y su rendimiento?

Ejercicio: Encontrar un camino…



13

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos
  - Tuplas
  - Relaciones
  - Funciones
  - Sucesiones

Análisis de complejidad

- Notaciones
  - Notación Big Theta - Θ
  - Notación Big O - O
  - Notación Big Omega - Ω
  - Jerarquía de los algoritmos
- Reglas prácticas para el cálculo de la complejidad
  - Sentencias simples
  - Condicionales
  - Ciclos
  - Algoritmos recursivos

14

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

Conjuntos

(Colección bien definida de objetos denominados **elementos** o miembros del conjunto)

- Conceptos
  - Nomenclatura A= { }
  - Elemento (∈) $a \in S$ $a \notin S$
  - Universo (U)
  - Cardinalidad |A|, Card (A), #A
  - Subconjunto $A \subset B$, $A \subseteq B$
  - Conjunto vacío $\varnothing = \{\ \}$
  - Conjuntos ajenos o disjuntos
  - Conjunto potencia $P(A) = 2^n$
    - $C(n,k) = nCk = \binom{n}{k} = \frac{n!}{k!(n-k)!}$
- Numerabilidad
  - Descripción
    - Enumeración
    - Comprensión
  - Tipo de elementos
    - Homogéneo
    - Heterogéneo
  - Orden
    - Ordenable
    - No ordenable
  - Tamaño
    - Finito
    - Infinito
- Representación
  - Diagramas de Venn

15

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

Conjuntos

(Colección bien definida de objetos denominados **elementos** o miembros del conjunto)

- Conceptos
  - Nomenclatura A= { }
  - Elemento (∈) $a \in S$ $a \notin S$
  - Universo (U)
  - Cardinalidad |A|, Card (A), #A
  - Subconjunto $A \subset B$, $A \subseteq B$
  - Conjunto vacío $\varnothing = \{\ \}$
  - Conjuntos ajenos o disjuntos
  - Conjunto potencia $P(A) = 2^n$
    - $C(n,k) = nCk = \binom{n}{k} = \frac{n!}{k!(n-k)!}$
- Numerabilidad
  - Descripción
    - Enumeración
    - Comprensión

$N = \{0, 1, 2, 3, \ldots\}$, el conjunto de *números naturales*

$Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, el conjunto de *enteros*

$Z^+ = \{1, 2, 3, \ldots\}$, el conjunto de *enteros positivos*

$Q = \left\{\frac{p}{q} \mid p \in z, q \in z, q \neq 0\right\}$, el conjunto de *números racionales*

R = el conjunto de *números reales*.

- Representación
  - Diagramas de Venn

16

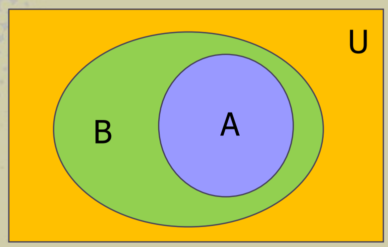
Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

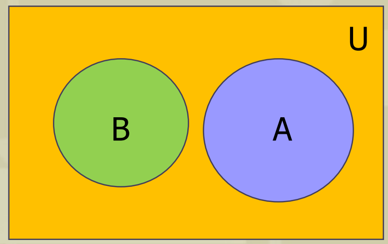
Representación { Diagramas de Venn

- Herramientas gráficas para representar conjuntos.



A ⊂ B

A y B son ajenos



Curso en Blackboard

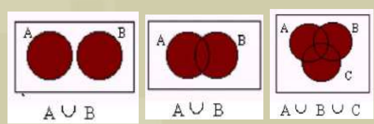
17

Técnicas de diseño de algoritmos

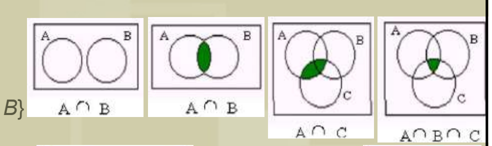
Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

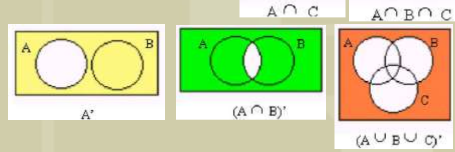
Operaciones con conjuntos



- **Unión:** $A \cup B = \{x \,/\, x \in A \vee x \in B\}$



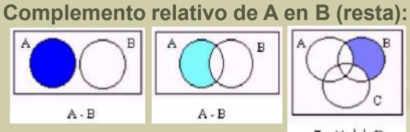
- **Intersección**: $A \cap B = \{x \,/\, x \in A \wedge x \in B\}$



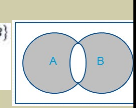
- **Complemento:** $A^c = \{x \,/\, x \notin A\}$

- **Complemento relativo de A en B (resta):** $A - B$

$A - B = \{x \,/\, x \in A \wedge x \notin B\}$ $= A \cap \overline{B}$



$$A \,\Delta\, B = \{\, x \,/\, (x \in A \vee x \in B) \wedge x \notin A \cap B \}$$
$$= \{\, x \in A \cup B \;\wedge\; x \notin A \cap B \}$$
$$= (A \cup B) - (A \cap B)$$



- **Diferencia sintética:** A \ B = (A ∪ B) - (A ∩ B)
- **Diferencia simétrica:** A ⊕ B = (A ∪ B) - (A ∩ B) = (A - B) U (B - A)

Curso en Blackboard

18

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

Ejercicios

- **Unión:** $A \cup B = \{x / x \in A \lor x \in B\}$
- **Intersección:** $A \cap B = \{x / x \in A \land x \in B\}$
- **Complemento:** $A^c = \{x / x \notin A\}$
- **Complemento relativo de A en B (resta):** $A - B$
  $$A - B = \{x / x \in A \land x \notin B\} = A \cap \overline{B}$$
- **Diferencia sintética:** A \ B = (A ∪ B) - (A ∩ B)
- **Diferencia simétrica:** A ⊕ B = (A ∪ B) - (A ∩ B)
  $$A \,\Delta\, B = \{ x / (x \in A \lor x \in B) \land x \notin A \cap B\} = \{ x \in A \cup B \land x \notin A \cap B\} = (A \cup B) - (A \cap B)$$

1. **Sea U = {1, 2, …, 9}** y

A = {1, 2, 3, 4, 5} **D = {1, 3, 5, 7, 9}**
B = {4, 5, 6, 7} **E = {2, 4, 6, 8}**
C = {5, 6, 7, 8, 9} **F = {1, 5, 9}**

Encontrar el resultado de las siguientes operaciones:

No	Operación	Resultado
a)	A ∪ C	
b)	A ∩ C	
c)	D ∪ F	
d)	D ∩ F	
e)	A^C	
f)	D^C	
g)	E^C	
h)	A - B	
i)	B ⊕ A	
j)	F - D	
k)	E \ F	
l)	B \ C	
m)	$(F \cup E)^C$	

Curso en Blackboard

19

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

Ejercicios

2. Resuelve los siguientes problemas utilizando conjuntos:

a) La compañía de "Desarrollo de sistemas, S.A." necesita contratar 18 personas que programen en Python y 12 personas que programen en Java. De estos programadores se considera que 10 personas saben programar tanto Python como en Java. ¿Cuántos programadores deberá contratar la compañía?

Solución:

Resultado

b) De una muestra de 42 estudiantes de la carrera de Informática se obtuvo el siguiente número de reprobados por materia:

28 Matemáticas para computación
26 Fundamentos de programación
17 Administración
16 Matemáticas para computación y fundamentos de programación
12 Fundamentos de programación y Administración
8 Matemáticas para computación y Administración
4 Matemáticas para computación Fundamentos de programación y Administración

Contesta:

1) ¿Cuántos estudiantes no reprobaron ninguna materia de las antes mencionadas?
2) ¿Cuántos estudiantes reprobaron solamente fundamentos de programación?
3) ¿Cuántos estudiantes reprobaron solamente alguna de las tres materias?
4) ¿Cuántos reprobaron matemáticas para computación y fundamentos para programación, pero no administración?

Solución:

No	Resultado
1)	
2)	
3)	
4)	

Curso en Blackboard

20

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos

Ejercicios

2. Resuelve los siguientes problemas utilizando conjuntos:

   c) En una encuesta aplicada a 120 personas se encontró que:

   65 leen libros impresos
   45 leen libros electrónicos
   42 leen periódicos y revistas
   20 leen tanto libros impresos como electrónicos
   25 leen tanto libros impresos como periódicos y revistas
   15 leen tanto libros electrónicos como periódicos y revistas
   8 leen de cualquier tipo de material y en cualquier formato

   Con diagramas de Venn determinar:
   a) Encuentre el número de personas que leen por lo menos una de las tres publicaciones.
   b) Encuentre el número de personas que leen exactamente una publicación.

   d) En la universidad *Nueva Realidad* cada estudiante debe acreditar un curso básico sobre Lógica y uno en particular sobre Design Thinking. En una muestra de 140 estudiantes de segundo año se observó lo siguiente:

   60 acreditaron el curso sobre Design Thinking,
   45 acreditaron el curso sobre Lógica
   20 acreditaron ambos cursos.

   Utiliza los diagramas de Venn para determinar el número de estudiantes que acreditaron:

   a) Por lo menos uno de los cursos
   b) Exactamente uno de Design Thinking o uno de Lógica
   c) Ninguno de los cursos.

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Tuplas
    - Una tupla es una sucesión finita de elementos, donde el orden sí importa.
    - Se representa por los elementos colocados entre paréntesis y separados por comas.
      - Ejemplo, la tupla (a, b, c) es diferente de la tupla (b, c, a).
    - Si la tupla tiene dos elementos se le conoce como "par", si tiene tres, se le conoce como "tercia", pero si tiene más elementos se complican los nombres, por lo que, en general, a una tupla de *k* elementos se le conoce como una *k-tupla*.
    - El orden en los elementos de una tupla sí importa.
      - Ejemplo, los puntos en un plano cartesiano se pueden representar por un par ordenado o 2-tupla, llamada coordenadas del punto.

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Tuplas
    - Los elementos de una tupla pueden ser de diferentes tipos.
      - En algunas aplicaciones de Ciencias Computacionales como las Bases de Datos o la Ciencia de Datos, los datos se pueden representar por medio de una tupla, en donde el primero elemento corresponde al valor de una variable o característica (*feature*) en ciencia de datos, el segundo al valor de otra variable y así sucesivamente.
    - Las tuplas nos permiten definir una operación más entre dos conjuntos, llamada producto cruz y representada con el símbolo "x".
      - El producto cruz de dos conjuntos A y B, representado por A x B, está formado por todas las parejas ordenadas en las que el primer elemento pertenece a A y el segundo pertenece a B.
      - Ejemplo,
        - si A = {a, b, c} y B = {1, 2}, el producto cruz A x B es:

A x B = {(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)}

23

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Tuplas
    - Producto cartesiano

Para el producto cartesiano de dos conjuntos A y B: $A \times B$ (en este orden), es el conjunto de todos los posibles pares ordenados, tales que la primer componente del par ordenado es un elemento de A y el segundo componente es un elemento de B.

La expresión $A \times B$ se le "A cruz B" y se expresa por descripción:

$A \times B = \{ (x, y)\ x \in A\ \wedge y \in B \}$

que se lee: el producto A cruz B es el conjunto de parejas ordenadas (x, y) tal que x pertenece a A y y pertenece a B.

De tal forma que para: $A = \{1,2,3\}$ y $B = \{a,b,c,d\}$

$$A \times B = \{(1,a),(1,b),(1,c),(1,d),(2,a),(2,b),(2,c),(2,d),(3,a),(3,b),(3,c),(3,d)\}$$

en donde:

en la pareja $(1,a)$ *1*, es la primera componente y *a* es la segunda componente.

Si los elementos de los conjuntos A y B son números reales, se acostumbra llamar a las componentes (x, y) como (abscisa y ordenada).

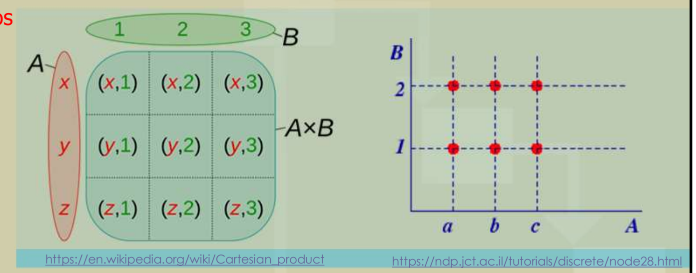
24

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Tuplas
    - Producto cartesiano

Ejemplos



Ejercicios

- Sea A = {a, b, c}, B = {x} y C = {0, 1}

  a) Encuentra A x B
  b) Encuentra C x B
  c) Encuentra B x B
  d) Encuentra A x B x C
  e) Encuentra C x A x B

Curso en Blackboard

25

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones
    - El concepto de relación surge de manera natural en el análisis de un sistema.

      Ejemplo: en los números Naturales se establece la relación "… es menor que …".

      Bajo esta relación **R** el número 2 se relaciona con el 3: 2 es menor que 3, pero no así al contrario (3 no es menor que 2).

    - Una relación es cualquier subconjunto de un producto cruz, es decir, es un conjunto donde sus elementos son tuplas.

      R⊆AxB

      Así, una relación *R* es un conjunto cuyos elementos son parejas ordenadas.

      Formalmente, si $R = A \times B$ entonces $R = \{(a,b) | \quad a \in A, \quad b \in B\}$

      - Un par ordenado (*también llamado pareja ordenada*) consta de dos elementos: (a, b) en donde el orden en que aparece (primero a, después b) indica la relación: *aRb* de a con b

      Si $(a,b) \in R$, $\rightarrow aRb$

      Si $(a,b) \notin R$, $\rightarrow a\not{R}b$

    - Una relación asocia un elemento de un conjunto A con un elemento de otro conjunto B o con un elemento del mismo conjunto A.

    - Una relación es binaria cuando se establece entre dos objetos.

      Ejemplo: $R : x < y$.
      (dominio, rango)

Curso en Blackboard

26

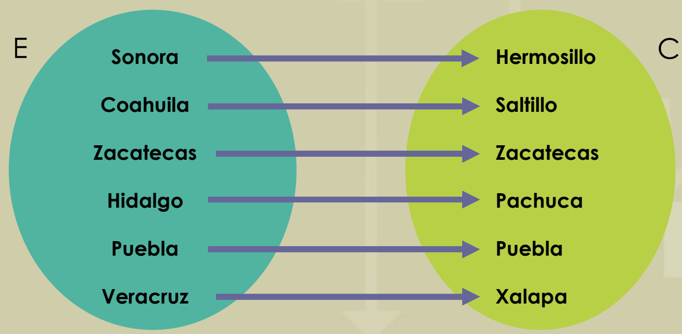
Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

Sea una relación R la que existe entre cada Estado de la República y sus capitales.



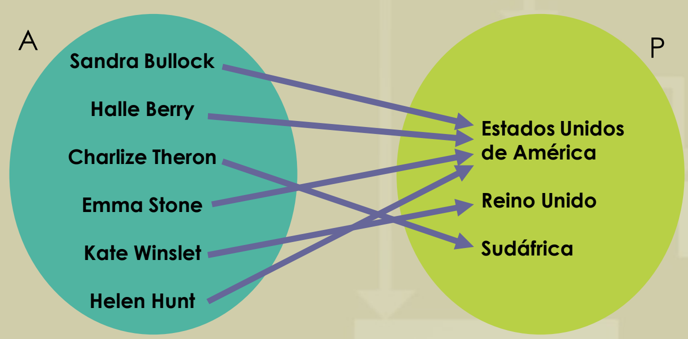
Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

Sea una relación R la que existe entre actrices que han ganado el Oscar y sus países de nacimiento.



Técnicas de diseño de algoritmos

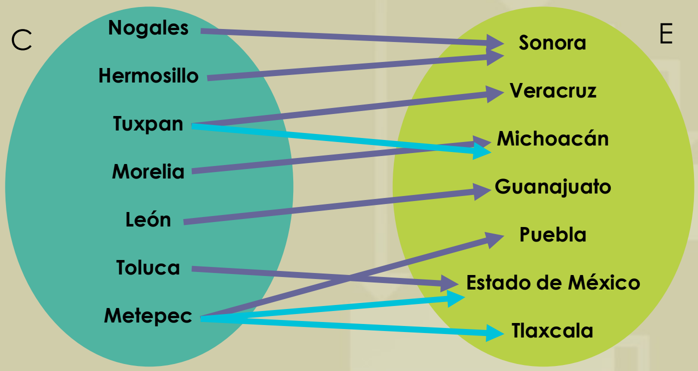
Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

Ciudades de la República Mexicana con su Estado

Nótese que pueden existir 2,3 ciudades/poblaciones con el mismo nombre en diferentes estados.



29

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

- Para A= {a, b, c}, A1= {a, b, c}

  R_1= {(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c)}

  $\Rightarrow R = A \times A1$ Producto cartesiano

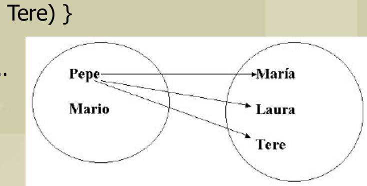
  (dominio, rango)

- Para A = {España, Inglaterra, Italia}, B= {París, Roma, Madrid}

  R_2: { (España, París), (Inglaterra, Roma), (Italia, Madrid) }

- R_3: { (Pepe, María), (Pepe, Laura), (Pepe, Tere) }

Esta relación puede ser: ... hermano de...



30

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

- A = {Familia Rodríguez}
  - R_1: … es papá de … (A, C) (A, D) (A, E)
  - R_2: … es más alto que …
  - R_3: … es más grande en edad que …

Miembro	Edad	Peso	Estatura
Papá Alfonso	(A) 42	77	1.80
Mamá Beatriz	(B) 40	57	1.68
Hijo 1: Carlos	(C) 19	61	1.88
Hijo 2: David	(D) 17	66	1.63
Hijo 3: Elena	(E) 15	48	1.53

- Sea A = {0, 1, 2} y B = {a, b}.

  Si se define: {(0, a), (0, b), (1,a), (2, b)} como relación de A a B.

  Esto indica que: 0 R a, pero que 1 $\not R$ b.

Curso en Blackboard

31

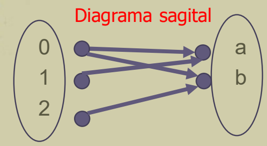
Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Las relaciones también pueden ser expresadas gráficamente:

Diagrama sagital



Matriz de la relación

R	a	b
0	1	1
1	1	0
2	0	1

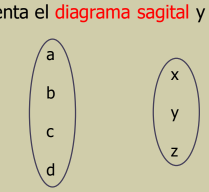
Renglones son los elementos de A, y las columnas son los elementos de B. En cada posición se coloca un 1 (*si se encuentra en la relación*) o 0 (*si no existe relación*) según se describa en la relación.

Ejemplos:

- Sean los conjuntos A = { a, b, c, d } y B= { x, y, z }

  y sea la relación R: A → B tal que: R= { (a,y), (a,z), (b,x), (c,y), (d,y) }

  Representa el diagrama sagital y la matriz de esa relación:



M_R =

R	x	y	z
a			
b			
c			
d			

Curso en Blackboard

32

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

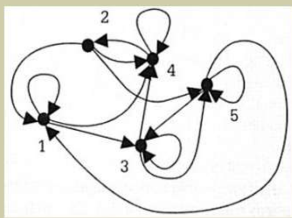
Ejemplos

- Sea los conjuntos: A = B = {1, 2, 3, 4, 5 }
- y la relación

  R = {(1, 1), (1, 3), (1, 4), (2, 1), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5), (4, 2), (4, 4), (5, 1), (5, 3), (5, 5) }

- Elabora el grafo dirigido y la representación matricial:

Grafo de R



Matriz de R

M_R =

R	1	2	3	4	5
1					
2					
3					
4					
5					

Curso en Blackboard

33

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos:

Sea los conjuntos:

A = {1, 2, 3, 4, 5}, B = {1, 2, 3, 4, 5, 6, 7}

* Dominio

* Rango

y sea la relación R: A → B tal que:

R = {(1, 2), (1, 3), (2, 2), (2, 5), (3, 2), (3, 7), (4, 2), (4, 5), (5, 6)}

Esta relación se puede representar en forma de matriz como sigue:

M_R =

R	1	2	3	4	5	6	7
1							
2							
3							
4							
5							

Los elementos del conjunto A se representan como filas y los del conjunto B como columnas.

Se coloca un 1 si el par ordenado se encuentra en la relación y un 0 en caso contrario.

Curso en Blackboard

34

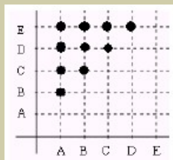
Técnicas de diseño de algoritmos

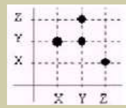
Herramientas básicas

- Herramientas matemáticas
  - Relaciones

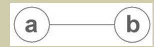
Representaciones gráficas

Gráfica de relaciones no numéricas





Grafo NO dirigido

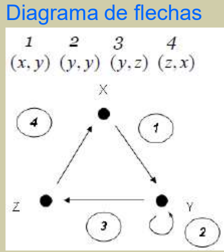


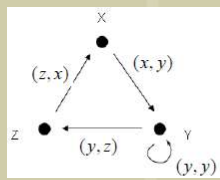
Grafo dirigido

Gráfica dirigidas

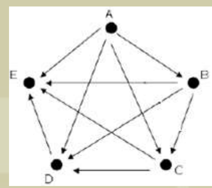
1) Se escriben los elementos del conjunto.
2) Se traza una flecha desde cada elemento *x* hasta cada elemento *y*. *Siempre que x este relacionada con y.*

Diagrama de flechas





Relación: ...es más grande que...



35

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Ejemplos

Grafos dirigidos

1) Se escriben los elementos del conjunto.
2) Se traza una flecha desde cada elemento *x* hasta cada elemento *y*. *Siempre que x este relacionada con y.*

Ejemplo:

Elabora el grafo dirigido de la relación R sobre el conjunto A = B = {1, 2, 3, 4}:

R = {(1, 2), (2, 2), (2, 4), (3, 2), (3, 4), (4, 1), (4, 3)}.

Realiza el grafo dirigido

1 2

3 4

Realiza la representación matricial:

M_R =

R	1	2	3	4
1				
2				
3				
4				

36

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Relaciones

Propiedades

$$M_R = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 \end{array} \qquad M_R = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

1) Reflexividad:

Una relación R es reflexiva si para cada elemento $a \in A$, existe un par ordenado $(a, a) \in R$.

Es decir: $_xR_x : \forall x \in S \Rightarrow {}_xR_x$ (x está relacionada con x)

Diremos que R es **reflexiva** si $\forall x \in A, \quad x R x$ — Tiene unos en su DIAGONAL PRINCIPAL

Diremos que R es irreflexiva si $\forall x \in A, \quad x \not R x$ — Ningún elemento está relacionado consigo mismo.

2) Simétrica:

Una relación R es **simétrica** si para cada par ordenado $(a, b) \in R$, existe un par ordenado $(b, a) \in R$.

Es decir: $\forall x, y \in S$. Si $_xR_y \Rightarrow {}_yR_x$

$$M_R = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

Antisimétrica:

Una relación R es **antisimétrica** si para cada par $(a, b) \in R$, no existe un solo par ordenado tal que $(b, a) \in R$. Es decir: $(a, b) \not\Rightarrow (b, a)$

3) Transitiva:

Una relación R es transitiva si incluye pares tales que $(a, b) \in R$ y $(b, c) \in R$, y además existe un par ordenado tal que $(a, c) \in R$.

Es decir: $\forall x, y, z \in S$. Si $_xR_y$ y $_yR_z \Rightarrow {}_xR_z$

```
String Transitiva(){
    if(a==b and b==c){
      return "a es igual a c";
    }
}
```

37

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Funciones

Intuitivamente una función es una regla que asocia elementos de un conjunto A con elementos de un conjunto B de modo que el elemento del conjunto A se asocia con uno y sólo un elemento del segundo conjunto.

En otras palabras, una función es una máquina que transforma elementos en otros elementos y cada elemento puede transformarse en un único elemento, *no en dos o tres.*

modelo matemático

a → Función → b

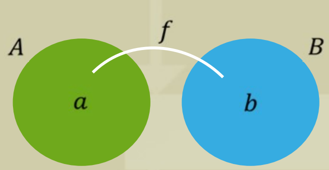
$y = f(x)$

Definición:

Decimos que la variable y está en función de la variable x, si se cumple que cada valor de x se relaciona con un único valor de y. La forma de denotar esta relación funcional es: $y = f(x)$.

Notación:

$$f : A \to B$$
$$a \mapsto b$$
$$f(a) = b$$



Curso en Blackboard

38

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Funciones como relaciones
    - Una función puede ser expresada *(y considerada)* como una relación,
      - El primer elemento *a* del par ordenado (*a, b*) pertenece al dominio de la función y el segundo elemento *b* se encuentra en el codominio (*imagen*) de la función.
    - La relación de una función **nunca** repetirá el primer elemento en los pares ordenados.

Ejemplo:

A= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = {conjunto de calificaciones en base a 10}

B= {NA, S, B, MB} = {conjunto de símbolos que representan un rendimiento escolar }

A × *B* son todas las posibles relaciones

A × *B* = 44 parejas

$$A \times B = \begin{Bmatrix} (0, NA)(1, NA) & \ldots & (10, NA) \\ (0, S)(1, S) & & (10, S) \\ (0, B)(1, B) & & (10, B) \\ (0, MB)(1, MB) & & (10, MB) \end{Bmatrix}$$

R **:**

Si NA = no acreditada ⇒ calificación 0 - 5

Si S = suficiente ⇒ calificación 6 - 7

Si B = bien ⇒ calificación 8 - 9

Si MB = muy bien ⇒ calificación 10

⇒ es una función porque a cada elemento de A corresponde solo uno de B a la relación se le llama regla de correspondencia *f*, entonces, b = f(a) un elemento del conjunto B está en función de un elemento del conjunto A.

Nomenclatura: $y = f(x)$

Curso en Blackboard

39

Técnicas de diseño de algoritmos

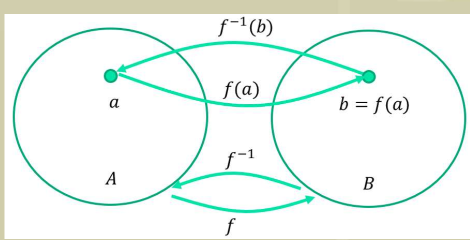
Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Inversa

- Consideremos una función biyectiva f desde el conjunto A al conjunto B.
- f es una función uno a uno: B es la imagen de un elemento único de A.
- Como consecuencia, podemos definir una nueva función de B a A que revierta la correspondencia dada por f.
- La función inversa se denota por: f^{-1}. Así: $f^{-1}(b) = a$ cuando $f(a) = b$
- Es decir, si f es una función uno a uno, entonces la inversa de f, denotada por f^{-1} es:

$$f^{-1} = \{(y, x) \,/ (x, y) \text{ está en } f\}$$



Curso en Blackboard

40

Técnicas de diseño de algoritmos

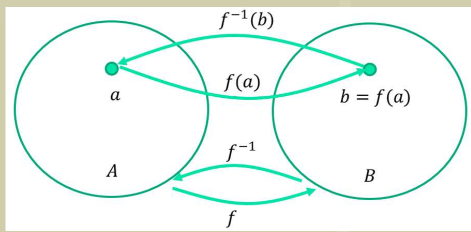
Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Inversa

- Consideremos una función biyectiva f desde el conjunto A al conjunto B.
- f es una función uno a uno: B es la imagen de un elemento único de A.
- Como consecuencia, podemos definir una nueva función de B a A que revierta la correspondencia dada por f.
- La función inversa se denota por: f^{-1}. Así: $f^{-1}(b) = a$ cuando $f(a) = b$
- Es decir, si f es una función uno a uno, entonces la inversa de f, denotada por f^{-1} es:

$f^{-1} = \{(y, x) \, / (x, y) \text{ está en } f\}$



Curso en Blackboard

41

Técnicas de diseño de algoritmos

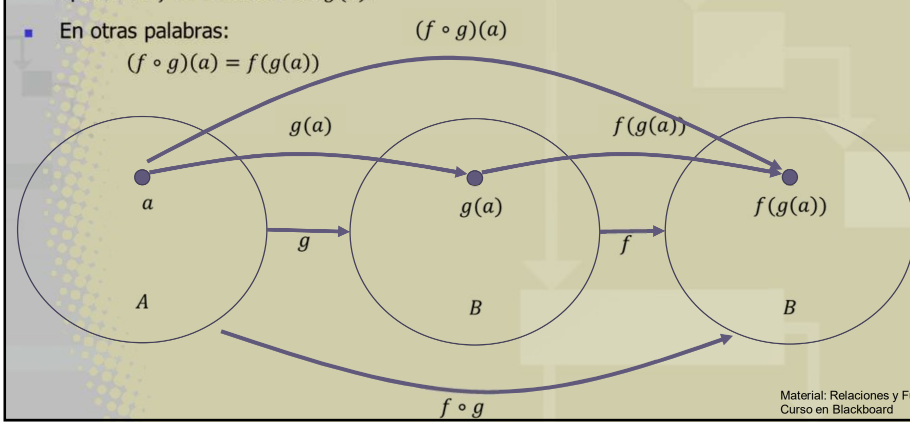
Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Función compuesta: fog

- $f \circ g$ es la función que asigna al elemento a de A el elemento asignado por f a $g(a)$.
- Para encontrar $(f \circ g)(a)$, aplicamos la función g a a para obtener $g(a)$ y entonces aplicamos f al resultado de $g(a)$.
- En otras palabras:

$(f \circ g)(a) = f(g(a))$



Material: Relaciones y Funciones.
Curso en Blackboard

42

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Función compuesta: fog

Ejemplo:

```
int main()
{   const int responseSize = 99;
    int f[ 10 ] = { 0 };
    response[ responseSize ] = { 1,1,2,3,4,5,5,5,6,7,8,9,9 }

    QUEHACE( f, response, responseSize );

    system("pause");
    return 0;
}
```

```
void QUEHACE( int fr[], int answer[], int size )
{
  int  rating, largest = 0, mValue = 0;

  for ( rating = 1; rating <= 9; rating++ )
      fr[ rating ] = 0;

   for ( int j = 0; j < size; j++ )
      fr[ answer[ j ] ]++;

   cout << "Respuesta ";

   for ( rating = 1; rating <= 9; rating++ )  {
           cout << rating << "    "<< fr[ rating ] << "      ";

            if ( fr[ rating ] > largest )   {
                  largest = fr[ rating ];
                  mValue = rating;
                  }

            cout << '\n';
    }
   cout << " El valor  " << mValue  <<  "  es igual a  " << largest;
}
```


Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Función recursiva

- Una función recursiva es aquella que depende de valores precedentes (anteriores).
- Debe contener:
  - Condiciones iniciales
  - Procedimiento
  - Condición de término

Ejemplos:

Subrutina *fibonacci* (L_1, L_2, n, L)
Definición de variables
Si $n > 2$ entonces $L = L_1 + L_2$
$L_1 = L_2$
$L_2 = L$
$n = n - 1$
llamar *fibonacci* (L_1, L_2, n, L)
si no
regresar

Función factorial

- La función de factorial es aquella que depende de valores precedentes (anteriores).

Ejemplos:

Función factorial (n, fac)
Definición de variables
Si $n = 0$ entonces $fac = 1$
regresar fac;
Si no
si $n > 1$ entonces
$fac = fac * n$
$n = n - 1$
$fac =$ función factorial (n, fac)
regresar fac;

Curso en Blackboard

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Cotas superiores

Función Entero Mayor

- Cota superior, mejor conocida como función ceil.
- La función superior, redondea x hacia arriba, es decir al número entero más cercano mayor o igual a x.
- $\lceil x \rceil$: (función techo) redondea hacia el siguiente entero.

Se denota por:

- Valor de la función ceil en x: $\lceil x \rceil$

- Ejemplos:

$y = \lceil 3.01 \rceil = 4 \qquad y = \lceil 3.51 \rceil = 4 \qquad y = \lceil 3.91 \rceil = 4$

$y = \lceil -3.01 \rceil = -3 \qquad y = \lceil -3.51 \rceil = -3 \qquad y = \lceil -3.91 \rceil = -3$

Material: Relaciones y Funciones. Curso en Blackboard

45

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Cotas inferiores

Función Entero Menor

- Cota inferior, mejor conocida como función floor.
- Dado x, que es un número real, la función inferior redondea x hacia abajo, es decir al número entero más cercano menor o igual a x.
- $\lfloor x \rfloor$: (función suelo) redondea hacia el entero.

Se denota por:

- Valor de la función floor en x: $\lfloor x \rfloor$

- Ejemplos:

$y = \lfloor 3.01 \rfloor = 3 \qquad y = \lfloor 3.51 \rfloor = 3 \qquad y = \lfloor 3.91 \rfloor = 3$

$y = \lfloor -3.01 \rfloor = -4 \qquad y = \lfloor -3.51 \rfloor = -4 \qquad y = \lfloor -3.91 \rfloor = -4$

Material: Relaciones y Funciones. Curso en Blackboard

46

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Algunas otras funciones

Función truncar

- Función Truncar
- *TRUNC*(x) : da como resultado la parte entera.
- Ejemplos:

$y = trunc(3.01) = 3$	$y = trunc(3.51) = 3$	$y = trunc(3.91) = 3$
$y = trunc(-3.01) = -3$	$y = trunc(-3.51) = -3$	$y = trunc(.3.91) = -3$

Logaritmos

Logaritmos

- $\log_b a$ es una función estrictamente creciente.
- $\log_b 1 = 0$.
- $\log_b b^a = a$.
- $\log_b(XY) = \log_b X + \log_b Y$
- $\log_b X^a = a \log_b X$
- $X^{\log_b Y} = Y^{\log_b X}$
- $\log_c X = \frac{\log_b x}{\log_b c}$

Material: Relaciones y Funciones. Curso en Blackboard

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Sucesiones

- Las sucesiones son un tipo especial de función que tiene por dominio un conjunto de enteros consecutivos, los cuales indican las posiciones de los elementos dentro de la posición, por lo que se les conoce como índices.

0	6	12	18	24	30	…

1	3	5	7	9	…

- El *n-ésimo* término de una sucesión S se denomina S*n* o con notación de función S(*n*). $\{a_n\}$, describe una sucesión,

  con a_n identificamos al n-ésimo elemento de la sucesión.

- Los elementos de una sucesión pueden ser identificados por la posición que guardan dentro de la sucesión, donde, el primer índice puede ser cualquier entero y los que le siguen serán consecutivos.

  $a_m, a_{k+1}, a_{k+2}, \dots, a_n$

  - Cada elemento a_k (*a sub k*) se llama término.
  - La letra *k* en a_k se conoce como subíndice o índice.
  - m es el subíndice del término inicial.
  - n es el subíndice del término final.

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Sucesiones

$N = \{0, 1, 2, 3, \ldots\}$, el conjunto de *números naturales*
$Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, el conjunto de *enteros*
$Z^+ = \{1, 2, 3, \ldots\}$, el conjunto de *enteros positivos*
$Q = \left\{\frac{p}{q} \mid p \in z, q \in z, q \neq 0\right\}$, el conjunto de *números racionales*
R = el conjunto de *números reales*.

Ejemplos

1) $\{a_n\} = 0, 1, 4, 9, 16, 25, \ldots$ es equivalente a… $\forall_n \in \boldsymbol{N}, a_n = n^2$
2) Se tiene la siguiente lista de términos: 2, 4, 8, 16, 32…

   Esta lista se obtiene a partir de la sucesión $\{a_n\}$ donde $a_n = 2^n$

   La lista de términos de esta secuencia es: $a_1, a_2, a_3, \ldots$

   Donde: $a_1 = 2$, $a_2 = 4$, $a_3 = 8$, … y $a_n = 2^n$.

Ejercicios

1) Determine los 5 primeros términos de la sucesión definida por:

$$a_n = 3\left[\frac{n}{3}\right], \text{ para } n \geq 4$$

2) Considere la secuencia $\{b_n\}$, donde $b_n = (-1)^n$.

   ¿Cuál es la lista de términos de la secuencia, iniciando desde: b_0?

3) ¿Qué "regla" permite producir la siguiente lista de términos de la secuencia?

   5, 11, 17, 23, 29, 35, 41, 47, 53, 59…

https://www.tiger-algebra.com/es/soluci%C3%B3n/progresi%C3%B3n-aritm%C3%A9tica/5,11,17,23,29,35/

Curso TC1003b.FIT

49

Técnicas de diseño de algoritmos

Herramientas básicas

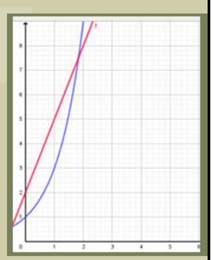
- Herramientas matemáticas
  - Sucesiones

Progresión aritmética

Es una sucesión de la forma: $a,\ a + d,\ a + 2d,\ a + 3d,\ \ldots,\ a + nd$

Donde: el término inicial a y la diferencia d son números reales.
La diferencia entre cualquier par de números sucesivos es constante.

Ejemplo:
para la sucesión 3, 6, 9, 12, …
Es una progresión aritmética de diferencia 3.



Gráficamente la progresión aritmética es una recta y la geométrica una función exponencial.

Progresión geométrica

Se tiene la siguiente lista de términos: *4, 8, 16, 32…*

Se puede expresar como: $4, 4*2^1, 4*2^2, 4*2^3, \ldots$

Una progresión geométrica es una sucesión de la forma:

$$a,\ ar,\ ar^2,\ \ldots,\ ar^n$$

Donde: el término inicial a y la razón r son números reales.

Curso TC1003b.FIT

50

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Sucesiones

Progresión aritmética

Es una sucesión de la forma: $a, \; a + d, \; a + 2d, \; a + 3d, \; \dots, \; a + nd$

Progresión geométrica

Una progresión geométrica es una sucesión de la forma: $a, \; ar, \; ar^2, \; \dots, \; ar^n$

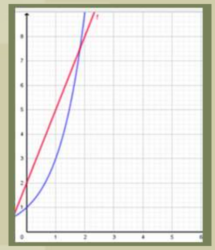
Ejercicios

1) Dadas las siguientes listas de términos, indique cuáles corresponden a progresiones aritméticas o geométricas:

a) 2, 4, 6, 8, 10, 12, …

b) 2, 4, 8, 16, 32, 64, …

c) 3, 1, -1, -3, -5, -7, …



Gráficamente la progresión aritmética es una recta y la geométrica una función exponencial.

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Sumatorias
    - La notación:

$$\sum_{k=m}^{n} a_k$$

Representa la suma desarrollada

$$a_m + a_{m+1} + a_{m+2} + \dots + a_n$$

Notación introducida en 1772 por el matemático francés J.L. Lagrange. En la notación de sumatoria, k se llama índice, m se llama el índice inferior de la suma, n se llama el índice superior de la suma.

Dada una serie $\{a_n\}$, una cota inferior entera (o *límite*) $j \geq 0$, y una cota superior entera $k \geq j$, entonces la sumatoria de $\{a_n\}$ *de j a k* se define y denota por:

$$\sum_{i=j}^{k} a_i = a_j + a_{j+1} + \dots + a_k$$

i se denomina *índice de la sumatoria*

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Sumatorias

Ejemplos

Series de Taylor / Leibniz

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

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad , \forall x$$

The Gregory Leibniz Series

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \cdots$$

Algunas otras expresiones

$$\sum_{k=0}^{n} ar^k = a(r^{n+1} - 1)/(r-1), r \neq 1$$

$$\sum_{k=1}^{n} k = n(n+1)/2$$

$$\sum_{k=1}^{n} k^2 = n(n+1)(2n+1)/6$$

$$\sum_{k=1}^{n} k^3 = n^2(n+1)^2/4$$

FÓRMULAS DE SUMATORIAS

1	$\sum_{i=1}^{n} 1 = n$	Suma de ***n*** veces 1 . Si n ≠ 1 $\sum_{i=1}^{n} c = c * n$
2	$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	suma de los números naturales desde *1* hasta ***n***.
3	$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$	suma de los cuadrados de los números naturales desde *1* hasta ***n*** .
4	$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$	suma de los cubos de los números naturales desde *1* hasta ***n*** .

Técnicas de diseño de algoritmos

Herramientas básicas

- Herramientas matemáticas
  - Conjuntos
  - Tuplas
  - Relaciones
  - Funciones
  - Sucesiones

Análisis de complejidad

- Notaciones
  - Notación Big Theta - Θ
  - Notación Big O - O
  - Notación Big Omega - Ω
  - Jerarquía de los algoritmos
- Reglas prácticas para el cálculo de la complejidad
  - Sentencias simples
  - Condicionales
  - Ciclos
  - Algoritmos recursivos

Técnicas de diseño de algoritmos

Análisis de complejidad

- Antecedentes
  - ***Algoritmos:*** el intelecto detrás del hardware que impulsa la industria de la computación.
  - ***Algoritmos eficientes:*** soluciones de calidad para problemas complejos o problemas que involucran grandes cantidades de información (*cada vez más abundantes en el mundo moderno*).
  - ***Comparar algoritmos:*** decidir (*con sustento*) que algoritmo es mejor que otro entre varias opciones que resuelven el mismo problema.
- ¿Por qué es importante?
  - ¿De qué depende la calidad de una solución?
    - ¿De la habilidad del programador?
    - ¿De la calidad del algoritmo utilizado?
    - ¿Del lenguaje de programación?

Para problema sencillos puede bastar un programador hábil

Problemas complejos demandan un "buen algoritmo"

Generalmente, el lenguaje de programación no es relevante

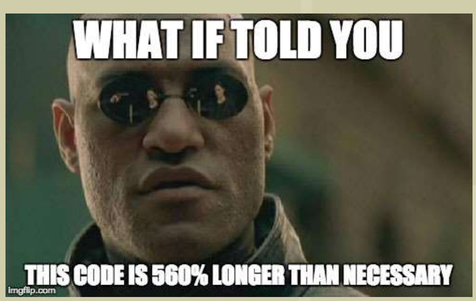
Técnicas de diseño de algoritmos

Análisis de complejidad

- ¿Cómo encontrar un "buen algoritmo"?
  - Se debe analizar el problema
  - Pensar en una solución viable
  - Analizar las soluciones existentes
  - Pensar en la eficacia y eficiencia de la solución.

El uso de la intuición no basta

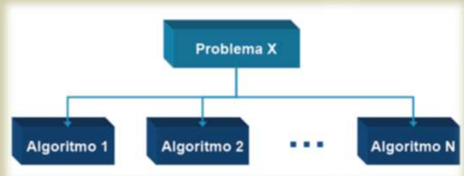
Encontrar o diseñar "buen algoritmo" es un proceso formal que no debe ser tomado a la ligera.



Técnicas de diseño de algoritmos

Análisis de complejidad

- ¿Cómo sé que mi algoritmo es eficiente?



- ¿Cuál es el mejor?
  - Criterios:
    - Tiempo
    - Facilidad de implementación
    - Memoria
    - Almacenamiento
    - Complejidad

Término clave

- No se pueden utilizar *unidades relativas* para medir la eficiencia de un algoritmo
  - Memoria
  - Tiempo
  - Almacenamiento
- La complejidad se basa en el número de operaciones elementales que deben realizarse para resolver un problema.

57

Técnicas de diseño de algoritmos

Análisis de complejidad

- Algoritmos
  - Se dice que cuando un problema tiene solución:
    - existe un algoritmo susceptible de implantarse en una computadora, **capaz de producir** la respuesta correcta para cualquier instancia del problema en cuestión.
  - En muchas ocasiones habrá más de un algoritmo disponible.
  - Clases de criterios para decidir:
    1. Orientados a minimizar **el costo de desarrollo** (claridad, sencillez, facilidad de implantación, depuración y mantenimiento).

       Para programas con un tiempo de vida corto
    2. Orientados a disminuir el **costo de ejecución**: tiempo de procesador y cantidad de memoria.

       Para programas que se utilizarán frecuentemente

58

Técnicas de diseño de algoritmos

Análisis de complejidad

- Análisis de Algoritmos
  - Los recursos que consumen los algoritmos pueden estimarse mediante herramientas teóricas (análisis de algoritmos).
  - Esto constituye una base confiable para la elección de un algoritmo.
- Tamaño del problema
  - Se asume que un problema tiene solución algorítmica si además de que el algoritmo existe, su tiempo de ejecución es razonablemente corto.
  - *Para fines prácticos*: si un problema tiene una solución que toma muchísimo tiempo en computar (***años***) dicha solución no existe.
    - Ejemplo:
      - Algoritmo para ganar en Ajedrez (https://www.chess.com/es/article/view/una-estrategia-de-ajedrez-para-ganar-un-millon-de-partidas).
  - En general, la cantidad de recursos que consume un algoritmo se incrementará conforme crece el tamaño del problema.
    - Ejemplo:
      - Algoritmo para ordenar una secuencia de números

IMTA, Dr. Alberto González, albertogonzalez@itesm.mx

59

Técnicas de diseño de algoritmos

Análisis de complejidad

- Tamaño del problema
  - Es necesario elegir uno o varios parámetros de un problema para definir el tamaño del problema.

Problema	Tamaño
Buscar *x* en un arreglo	Número de elementos del arreglo
Multiplicar dos matrices	Dimensión de las matrices
Recorrer un árbol	Número de nodos
Resolver un sistema de ecuaciones	Número de ecuaciones y/o incógnitas
Ordenar un conjunto de valores	Número de elementos en el conjunto

- Proporciona un parámetro para medir el desempeño de los algoritmos.
- En general, nos interesa comparar algoritmos diferentes que resuelven el mismo problema.
- Implementaciones distintas del mismo algoritmo tendrán un desempeño similar, afectados únicamente por una constante multiplicativa (***principio de invarianza***).
- Para comparar algoritmos utilizamos su función de complejidad.

IMTA, Dr. Alberto González, albertogonzalez@itesm.mx

60

Técnicas de diseño de algoritmos

Análisis de complejidad

- Instancia del problema
  - Además del tamaño, otro parámetro que afecta el desempeño de un algoritmo es la instancia.
  - La instancia son los valores concretos que toman los elementos variables involucrados en un problema, por ejemplo:

Problema	Tamaño	Instancia
Buscar *x* en un arreglo	Número de elementos del arreglo	Los valores concretos que tienen los elementos del arreglo
Multiplicar dos matrices	Dimensión de las matrices	Los valores de los elementos de la matriz
Recorrer un árbol	Número de nodos	La forma del árbol (balanceado, no balanceado)
Resolver un sistema de ecuaciones	Número de ecuaciones y/o incógnitas	Los valores de los coeficientes de los polinomios
Ordenar un conjunto de valores	Número de elementos en el conjunto	Orden de los valores originales del arreglo.

IMTA Dr. Alberto González

61

Técnicas de diseño de algoritmos

Análisis de complejidad

- Complejidad de algoritmos
  - La función complejidad ***f(n)*** (donde ***n*** es el tamaño del problema), busca dar una medida de la cantidad de recursos que un algoritmo necesitará al implantarse y ejecutarse en una computadora.
  - La cantidad de recursos que consume un algoritmo crece conforme el tamaño del problema se incrementa:

$$f(n) > f(m) \leftrightarrow n > m$$

  - Hay dos clases de función de complejidad
    - Función de complejidad espacial
    - Función de complejidad temporal

IMTA Dr. Alberto González

62

Técnicas de diseño de algoritmos

Análisis de complejidad

- Complejidad de algoritmos
  - Complejidad espacial
    - La cantidad de memoria que utiliza un algoritmo depende de la implantación, no obstante, se puede obtener una aproximación a partir de la inspección del algoritmo.
    - Es necesario sumar todas las celdas de memoria que utiliza:
      - Celdas estáticas (análogas a las variables globales).
      - Celdas dinámicas (análogas a las variables locales, el uso del *stack* y el uso de memoria dinámica).
    - Las variables de tipo simple ocuparán una celda de memoria, mientras que a las compuestas se le asignan tantas celdas como requieran sus elementos simples.

IMTA Dr. Alberto González

63

Técnicas de diseño de algoritmos

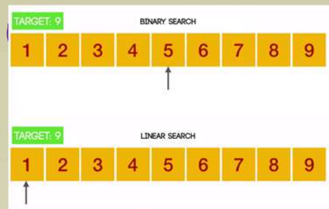
Análisis de complejidad

- Complejidad de algoritmos
  - Complejidad temporal
    - Generalmente, es más relevante que la complejidad espacial.
    - Refleja la cantidad de trabajo realizado al dar una medida del tiempo que requerirá la ejecución de un algoritmo para resolver un problema.
    - Se puede determinar de forma experimental:
      - El algoritmo ***A*** se codifica en el lenguaje ***L***, se compila con el compilador ***C*** y se ejecuta en la máquina ***M***.
    - El inconveniente es que los resultados dependerán de:
      - Las entradas proporcionadas
      - La calidad del código generado por el compilador utilizado
      - La máquina en la que se hagan las corridas
    - **¿Cómo lo comparo contra otro algoritmo?**
      - se necesitaría que el otro algoritmo se codificara por el mismo programador (*utilizando técnicas similares*), en el mismo compilador, y se ejecute en la misma máquina (*en las mismas condiciones*).
    - **¿Cómo evitar dicha dependencia?**

IMTA Dr. Alberto González

64

Técnicas de diseño de algoritmos



Análisis de complejidad

- Complejidad de algoritmos
  - Clasificación de funciones

Tenemos lo siguiente:

- Búsqueda lineal en arreglo desordenado: $f_1(n) = n$
- Búsqueda lineal en arreglo ordenado con condición de salida: $f_2(n) = n$
- Búsqueda binaria sobre arreglo ordenado: $f_3(n) = \log_2 n+1$

¿Qué tan buenas son estas medidas de complejidad?

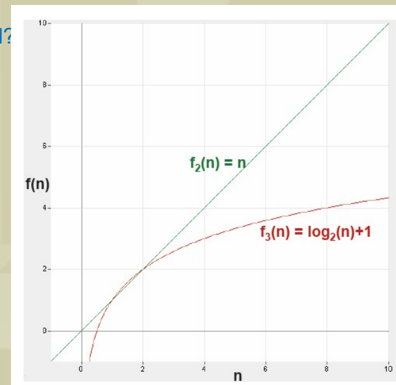
¿Cómo las comparo?

¿Qué es mejor?

$f_2(n) = n$ ó $f_3(n) = \log_2 n + 1$

Analizando con una gráfica en el rango [0,10] para *n*:

Si probamos los algoritmos con entradas pequeñas no veremos una gran diferencia en su eficiencia.



IMTA Dr. Alberto González

Notación Asintótica https://www.youtube.com/watch?v=-Wg0RVE37UM

65

Técnicas de diseño de algoritmos

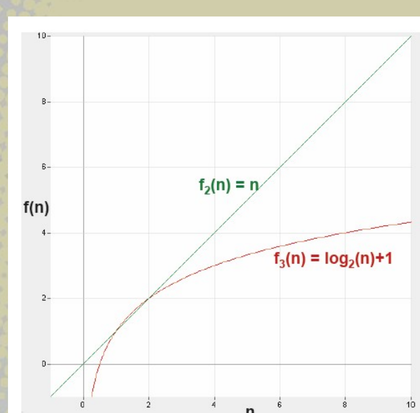
Análisis de complejidad

- Notación Asintótica
  - Tasa de crecimiento

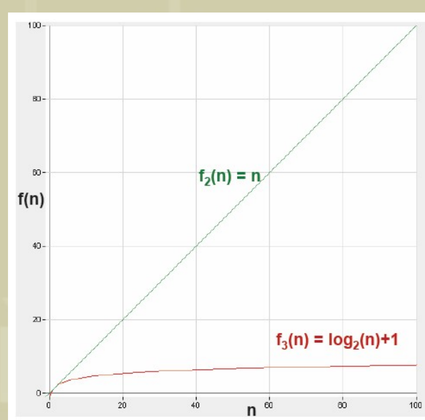
Estimación que nos indica la cantidad de recursos que requiere un algoritmo conforme crece el problema.

Conforme el problema es más grande:

$f_2(n) = n$ ❯ $f_3(n) = \log_2 n + 1$



Rango [0,10]



Rango [0,100]

IMTA Dr. Alberto González

66

Técnicas de diseño de algoritmos

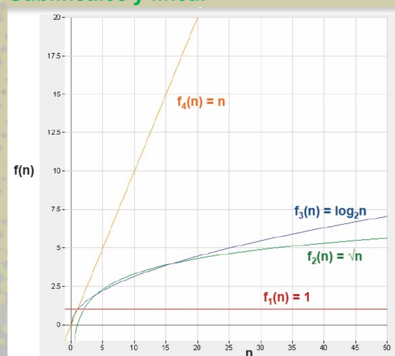
Análisis de complejidad

- Notación Asintótica
  - Tasa de crecimiento
    - Categorías

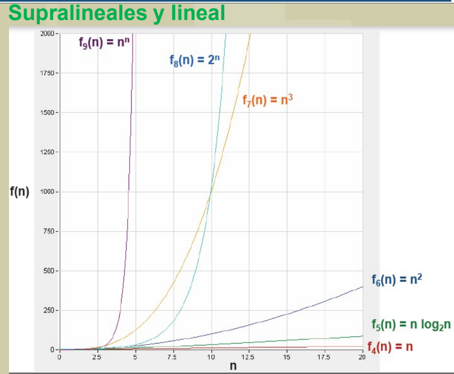
Sublineales		$F_1(n) = c$	(constante)
		$F_2(n) = c \cdot \log_2 n$	(logarítmica)
		$F_3(n) = c \cdot \sqrt{n}$	
Lineales		$F_4(n) = c \cdot n$	
Supralineales		$F_5(n) = c \cdot n \log_2 n$	
	Polinómicas	$F_6(n) = c \cdot n^2$	(cuadrática)
		$F_7(n) = c \cdot n^3$	(cúbica)
	Exponenciales	$F_8(n) = c \cdot 2^n$	
		$F_9(n) = c \cdot n^n$	

Orden de complejidad

Sublineales y lineal



Supralineales y lineal



IMTA Dr. Alberto González

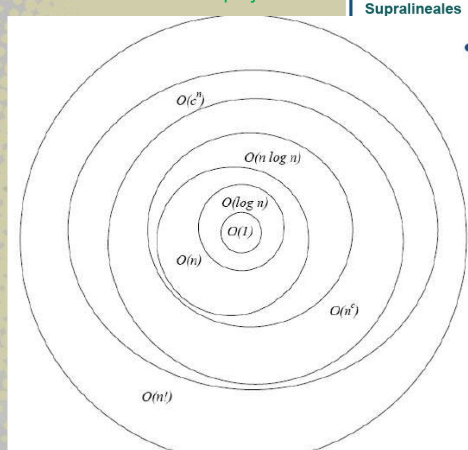
67

Técnicas de diseño de algoritmos

Análisis de complejidad

- Notación Asintótica
  - Tasa de crecimiento
    - Categorías
    - Orden de complejidad

Sublineales		$F_1(n) = c$	(constante)
		$F_2(n) = c \cdot \log_2 n$	(logarítmica)
		$F_3(n) = c \cdot \sqrt{n}$	
Lineales		$F_4(n) = c \cdot n$	
Supralineales		$F_5(n) = c \cdot n \log_2 n$	
	Polinómicas	$F_6(n) = c \cdot n^2$	(cuadrática)
		$F_7(n) = c \cdot n^3$	(cúbica)
	Exponenciales	$F_8(n) = c \cdot 2^n$	
		$F_9(n) = c \cdot n^n$	



Entre menor sea el orden de complejidad mejor será el algoritmo.

IMTA Dr. Alberto González

68

Técnicas de diseño de algoritmos

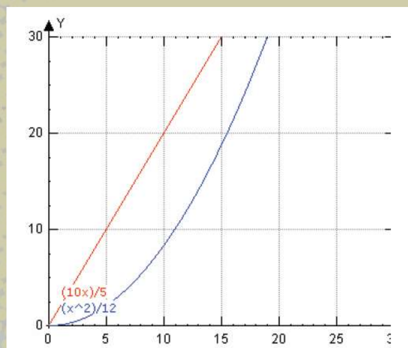
Análisis de complejidad

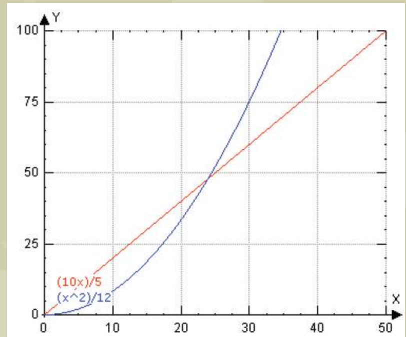
- Notación Asintótica
  - Tasa de crecimiento

    Categorías de tasa de crecimiento

    Constantes

    Para problemas con entradas muy grandes el valor de las constantes no modifica la tasa de crecimiento.





Técnicas de diseño de algoritmos

Análisis de complejidad

- Notación Asintótica
  - Asignación de orden de complejidad

    Si la complejidad de un algoritmo posee una expresión compuesta entonces se toma el término de mayor complejidad.

$4n^2 + 2n + 2$	es de orden n^2
$5n + 7$	es de orden n
$n\left(1-\frac{1}{2}q\right)+\frac{1}{2}q$	es de orden n

  - Notación asintótica (de la gran "O", Big O)

    Dada una función ***f(n)*** que representa la complejidad de un algoritmo:

    – **Θ(*f(n)*)**, se lee "theta de f" y denota el conjunto de funciones que crecen con la misma rapidez que f.

    – **O(*f(n)*)**, se lee "O de f" y denota el conjunto de funciones que no crecen más rápido que f.

    – **Ω(*f(n)*)**, se lee "omega de f" y denota al conjunto de funciones que crece al menos tan rápido como f.

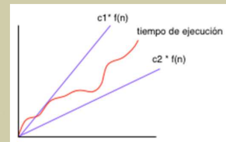
O(*f(n)*) y **Ω(*f(n)*)** es el conjunto de funciones que, asintóticamente, podemos acotar superiormente (**O**) e inferiormente (**Ω**) con una función proporcional a ***f(n)*** ➔

Video: https://www.youtube.com/watch?v=-Wg0RVE37UM

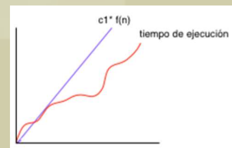
Técnicas de diseño de algoritmos

Análisis de complejidad

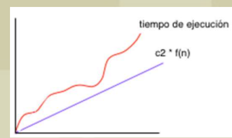
- Notaciones
  - Notación Big Theta - Θ
    - En general cuando se tiene una función de tiempo de ejecución *f(n)*, cuando *n* es suficientemente grande, el tiempo de ejecución estará entre c1 ∗ *f(n)* y c2 ∗ *f(n)*.
    - Mientras existan contantes c1 y c2 que delimiten *f(n)* para *n* muy grandes decimos que el tiempo de ejecución del algoritmo es Θ(n).



  - Notación Big O - *O*
    - Usaremos la notación Big-O para las cotas superiores asintóticas para entradas muy grandes y se dice que el tiempo de ejecución es “Big O de f(n)”, “O de f(n)” o simplemente “Orden de f(n)”.
    - Entonces podemos decir que la búsqueda secuencial *O(n)*.



  - Notación Big Omega - Ω
    - La Notación Big Ω es cuando se desee decir que un algoritmo toma por lo menos cierta cantidad de tiempo, sin querer ofrecer la cota superior.
    - Ya que la notación Big Ω es para mostrar los límites asintóticos inferiores.



71

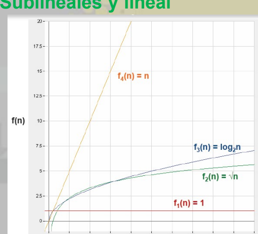
Técnicas de diseño de algoritmos

Análisis de complejidad

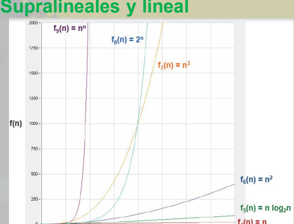
- Notación Asintótica
  - Tasa de crecimiento

Orden de complejidad

Sublineales y lineal



Supralineales y lineal



Categorías

Sublineales		$F_1(n) = c$	(constante)
		$F_2(n) = c \cdot \log_2 n$	(logarítmica)
		$F_3(n) = c \cdot \sqrt{n}$	
Lineales		$F_4(n) = c \cdot n$	
Supralineales		$F_5(n) = c \cdot n \log_2 n$	
	Polinómicas	$F_6(n) = c \cdot n^2$	(cuadrática)
		$F_7(n) = c \cdot n^3$	(cúbica)
	Exponenciales	$F_8(n) = c \cdot 2^n$	
		$F_9(n) = c \cdot n^n$	

O(*f(n)*)

Ejemplos:

Pertenece f(n)	O(n)	Resultado
$t(n) = 3n + 2$	$O(n)$	√
$t(n) = 100n + 6$	$O(n)$	√
$t(n) = 10n^2 + 4n + 2$	$O(n)$	×
$t(n) = 6 * 2^n + n^2$	$O(2^n)$	√
$t(n) = 3$	$O(1)$	√

Video: https://www.youtube.com/watch?v=-Wg0RVE37UM

72

Técnicas de diseño de algoritmos

Análisis de complejidad

- Jerarquía de los algoritmos
  - Complejidades usuales

Notación	Nombre
$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

O(1): complejidad constante. Ocurre cuando instrucciones son ejecutadas una y una vez solamente, independientemente del tamaño del problema.

O(log n): complejidad logarítmica, que crece ligeramente con n. Problemas de dicotomía, típicamente, donde el problema es compartido en varias instancias y solo una está procesada.

O(n): complejidad lineal. Problemas con ciclos sobre los datos con procesamiento de duración constante.

O(nlog(n)): complejidad n-logarítmica. Problemas donde en cada iteración se divide en sub-problemas, donde el procesamiento es lineal (*mergeSort*, *quickSort*).

$O(n^2)$: complejidad cuadrática. Caso de todos los algoritmos que implica dos ciclos, donde el ciclo interno se hace sobre todos los datos o aun sobre un número de datos lineal en el *index* del primer ciclo.

$O(n^3)$: complejidad cúbica, implica generalmente tres ciclos imbricados.

$O(2^n)$: complejidad exponencial, eso corresponde a algoritmos ingenuos, que a partir de ciertos tamaños no muy altos son completamente inutilizables.

https://medium.com/@joseguillermo_/qu%C3%A9-es-la-complejidad- http://aplicaciones.cimat.mx/Personal/sites/default/files/jbhayet/files/clase

73

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Complejidad temporal
  - El cálculo de ***f(n)*** se hará con base al algoritmo escrito en pseudocódigo.
  - La complejidad temporal se expresará en términos de la cantidad de operaciones que realiza, ya que cada operación requiere de una cantidad constante de tiempo para ser ejecutada.
- Elementos de un pseudocódigo
  - Será común el uso de los siguientes elementos:
    - Estructuras condicionales (*si-entonces/si-entonces-otro caso*)
    - Estructuras repetitivas (*repetir, mientras, para*)
    - Funciones, procedimientos
    - Arreglos, matrices
    - Objetos

IMTA, Dr. Alberto González, albertogonzalez@itesm.mx

74

Técnicas de diseño de algoritmos

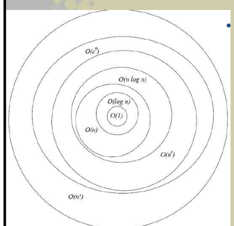
Reglas prácticas para el cálculo de la complejidad

- Sentencias simples

Las sentencias simples son aquellas que ejecutan operaciones básicas, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño está relacionado con el tamaño del problema. La inmensa mayoría de las sentencias simples requieren un tiempo constante de ejecución y su complejidad es *O(1)*.

Ejemplos:

```
x ← 1
y ← z + x + w
print x
read x
```



Algoritmos TC2038,

75

Técnicas de diseño de algoritmos

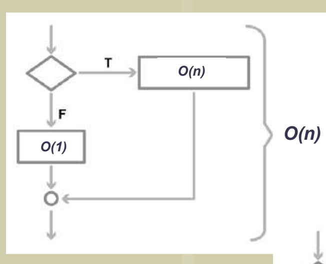
Reglas prácticas para el cálculo de la complejidad

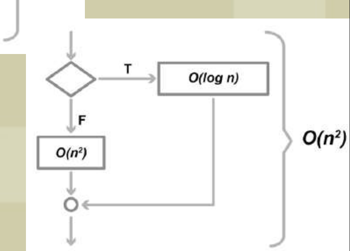
- Condicionales

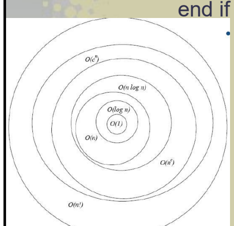
Los condicionales suelen ser *O(1)*, *a menos que* involucren un llamado a un procedimiento, y siempre se debe tomar la peor complejidad posible de las alternativas del condicional, bien en la rama afirmativa o bien en la rama alternativa. En decisiones múltiples *(switch)* se tomará la peor de todas las ramas.

Ejemplo:

```
if a > b then
  for i ← 1 to n do
    sum ← sum + 1
  end for
else
  sum ← 0
end if
```







Algoritmos TC2038,

76

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Ciclos (while, for, repeat-until)

En los ciclos con un contador explícito se distinguen dos casos: que el tamaño ***n* forme parte de los límites del ciclo**, con una complejidad basada en *n*, *O(n)*, o que dependa de la forma como avanza el ciclo hacia su terminación.

Si el ciclo se realiza un **número constante de veces**, independientemente de *n*, entonces la repetición solo introduce una constante multiplicativa que puede absorberse, lo cual da como resultado *O(1)*.

Ejemplo:

```
for i ← 1 to k do
  sentencias simples O(1)
end for
```

Si el tamaño *n* aparece como límite de las iteraciones, entonces la complejidad será: $n * O(1) \rightarrow O(n)$.

Ejemplo:

```
for i ← 1 to 5 do
  sentencias simples O(1)
end for.
```

En este caso, la complejidad será: $O(1) \rightarrow O(1)$.

77

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Ciclos (while, for, repeat-until)

Si los ciclos son anidados...

Ejemplo:

```
for i ← 1 to n do
  for j ← 1 to n do
    sentencias simples O(1)
  end for
end for
```

En este caso, la complejidad sería: $n * n * O(1) \rightarrow O(n^2)$

Para ciclos anidados pero con variables independientes:

Ejemplo:

```
for i ← 1 to n do
  for j ← 1 to i do
    sentencias simples O(1)
  end for
end for
```

La complejidad será

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

78

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Ciclos (while, for, repeat-until)

A veces aparecen **ciclos multiplicativos**, donde la evolución de la variable de control no es lineal:

Ejemplo:

```
c ← 1
while c < n do
  c ← c * 2
end while
```

La complejidad será

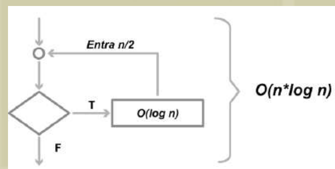
El valor inicial de la variable *c* es *1*, y llega a 2^n al cabo de *n* iteraciones → $log_2\ n$.

Y la combinación de los anteriores:

Ejemplo:

```
for i ← 1 to n do
  c ← n
  while c > 0 do
    c ← c/2
  end while
end for
```

→ $log_2\ n$.



La complejidad será

Se tiene un ciclo interno de orden $O(log_2\ n)$ que se ejecuta *n* veces en el ciclo externo; por lo que, el ejemplo es de orden $O(n\ log_2\ n)$.

79

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos
    - Se basan en la ejecución de ciclos.
    - Las funciones y estructuras repetitivas siguen una secuencia continua.
    - Ejemplo:

```
for(int i = 1; i <= 10; i++) {
        printf("¡Esta es la vez %d que hago esto!\n", i); }
```

  - Algoritmos recursivos
    - Se basan en la ejecución de rutinas que se "llaman" a sí mismas.
    - Implementan procesos que están especificados basados en su propia definición.
    - Ejemplo:

```
void imprime(int n) {  // requiere llamarse como imprime(10)
   if (n==0)
          return;
   else  printf("¡Esta es la vez %d que hago esto!\n", 11-n);
   imprime(n--);
 }
```


80

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos recursivos

    Para poder analizar la eficiencia de los algoritmos recursivos, se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva.

    Normalmente se comportan de una de las siguientes formas:

    - *$O(n)$* - Cuando se tiene *una sola llamada* recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.
    - *$O(\log_b n)$* - Cuando se tiene *una sola llamada* recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor *b* constante.
    - *$O(C^n)$* - Cuando se tienen *c llamadas* recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.
    - *$O(n^{\log_b c})$* - Cuando se tienen *c llamadas* recursivas en ejecución y su parámetro de control se divide o se multiplica por un valor *b* constante.

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

Ejemplo:

Contabilizar el número de instrucciones que ejecutan cada uno de los siguientes programas. Encontrar la forma de su función de complejidad temporal.

Producto.C

```c
int main(void)
{
  int m, n;
  scanf("%d", &n);
  m = n * n;
  printf("%d\n", m);
  return 0;
}
```

Suma.C

```c
int main(void)
{
  int m, n, i;

  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    m = m + n;
  printf("%d\n", m);
  return 0;
}
```

Incremento.C

```c
int main(void)
{
  int m, n, i, j;

  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      m++;
  printf("%d\n", m);
  return 0;
}
```

Operaciones a contabilizar:

- Productos (*multiplicaciones*)
- Sumas
- Incrementos
- Asignaciones
- Comparaciones

Todos los programas obtienen el cuadrado de un número, pero ¿cuál se ejecuta más rápido?

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

Ejemplo: Contabilizar el número de instrucciones que ejecutan cada uno de los siguientes programas. Encontrar la forma de su función de complejidad temporal.

Producto.C

```
int main(void)
{
  int m, n;
  scanf("%d", &n);
  m = n * n;
  printf("%d\n", m);
  return 0;
}
```

Suma.C

```
int main(void)
{
  int m, n, i;

  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    m = m + n;
  printf("%d\n", m);
  return 0;
}
```

Incremento.C

```
int main(void)
{
  int m, n, i, j;

  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      m++;
  printf("%d\n", m);
  return 0;
}
```

Evaluando:

Número de operaciones: 2

Número de operaciones: 4n + 3

Número de operaciones: $3n^2 + 4n + 3$

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones
producto.c	1			1	
suma.c		n		$n+2$	$n+1$
incremento.c			$2n^2+n$	$n+2$	n^2+2n+1

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

Ejemplo: Contabilizar el número de instrucciones que ejecutan cada uno de los siguientes programas. Encontrar la forma de su función de complejidad temporal.

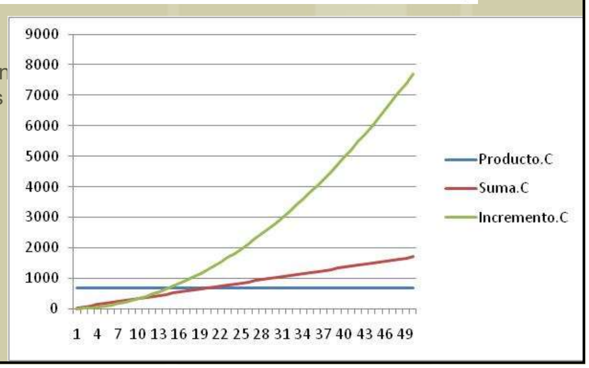
Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones	
producto.c	1			1		2
suma.c		n	n	$n+2$	$n+1$	4n + 3
incremento.c			$2n^2+n$	$n+2$	n^2+2n+1	$3n^2 + 4n + 3$

Evaluando:

Para evaluar el comportamien
los siguientes costos para las

- Producto: 342 μs
- Suma: 31μs
- El resto: 1 μs

¿A partir de dónde es más rápido producto.c?



Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

Ejemplo: Contabilizar el número de instrucciones que ejecutan cada uno de los siguientes programas. Encontrar la forma de su función de complejidad temporal.

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones	
producto.c	1			1		2
suma.c		n	n	$n+2$	$n+1$	4n + 3
incremento.c			$2n^2+n$	$n+2$	n^2+2n+1	$3n^2 + 4n + 3$

Evaluando:

Y si los costos fueran:

- Producto: 100 µs
- Suma: 10µs
- El resto: 1 µs

Por tal razón, en la práctica se llega a despreciar la diferencia del costo de operaciones elementales.

- Si lo has notado, el comportamiento de forma asintótica no cambia.
- Las gráficas son "proporcionalmente similares".

85

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

Ejemplo:

```
void busquedaLineal(valor,int[] A) {
    i=1;
    while (i<=A.Length && A[i]!=valor){
        i=i+1;
    }
    return i;
}
```

- ***k*** *sumas (una por cada iteración).*
- ***k*** *+ 2 asignaciones (las del ciclo y las realizadas fuera del ciclo).*
- ***k*** *+ 1 operaciones lógicas (la condición se debe probar* ***k*** *+ 1 veces)*
- ***k*** *+ 1 comparaciones con el índice.*
- ***k*** *+ 1 comparaciones con elementos de* ***A****.*
- ***k*** *+ 1 accesos a elementos de* ***A****:*

6k + 6 operaciones en total

¿Qué se observa en el ejemplo?

- El número de veces que se ejecutan algunas operaciones presentan un modelo de crecimiento similar al que tiene el número total de operaciones que ejecuta el algoritmo.
- No es necesario contar todas las operaciones, se puede elegir una operación en particular (denominada ***operación básica***) que será proporcional al tiempo total de ejecución.
- Ésta debe cumplir con los siguientes criterios:
  - Estar relacionada con el tipo de problema que resuelve
  - Ejecutarse un número de veces cuyo modelo de crecimiento sea similar al del número total de operaciones que efectúa el algoritmo.

86

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Tipos de Algoritmos
  - Algoritmos iterativos

    Ejemplo:

Problema	Operación básica
Búsqueda de un elemento en un conjunto.	Comparación entre el valor y los elementos del conjunto.
Multiplicar dos matrices.	Producto de los elementos de las matrices.
Recorrer un árbol.	Visitar un nodo.
Resolver un sistema de ecuaciones lineales.	Suma.
Ordenar un conjunto de valores.	Comparación entre valores.

Regresando al problema 1:

Producto.C

Número de operaciones:
2
f(n) = 2

Suma.C

Número de operaciones:
4n + 3
f(n) = 4n + 3

Incremento.C

Número de operaciones:
$3n^2 + 4n + 3$
f(n) = $3n^2 + 4n + 3$

Técnicas de diseño de algoritmos

Reglas prácticas para el cálculo de la complejidad

- Actividad

  Obtener la función de complejidad temporal de los siguientes segmentos de código:

(1)
```
mult = 1;
for (i = 0; i <= n; i++)
  for ( j = 0; j < i; j++)
    mult=i*j;
```

En este caso, la complejidad sería:
*n * n * O(1)* → $O(n^2)$

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

$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

(2)
```
mult = 1;
for (i = 1; i <= n; i++)
  for (j = 1; j <= i; j*=2)
    mult=i*j;
```

A veces aparecen **ciclos multiplicativos**, donde la evolución de la variable de control no es lineal:

La complejidad sería:

Se tiene un ciclo interno de orden $O(\log_2 n)$ que se ejecuta *n* veces en el ciclo externo; por lo que, el ejemplo es de orden $O(n \log_2 n)$.