Centro de
Investigación en
Computación

Basics of Algorithm Analysis

Conceptos básicos de Análisis de algoritmos

Tema 2

Course

Analysis and design of algorithms

Instructor

Acosta Bermejo Raúl et al.

Lecture notes

2023-A
21 de marzo del 2023

Instituto
Politécnico
Nacional

Table of contents (outline)

Tabla de contenido

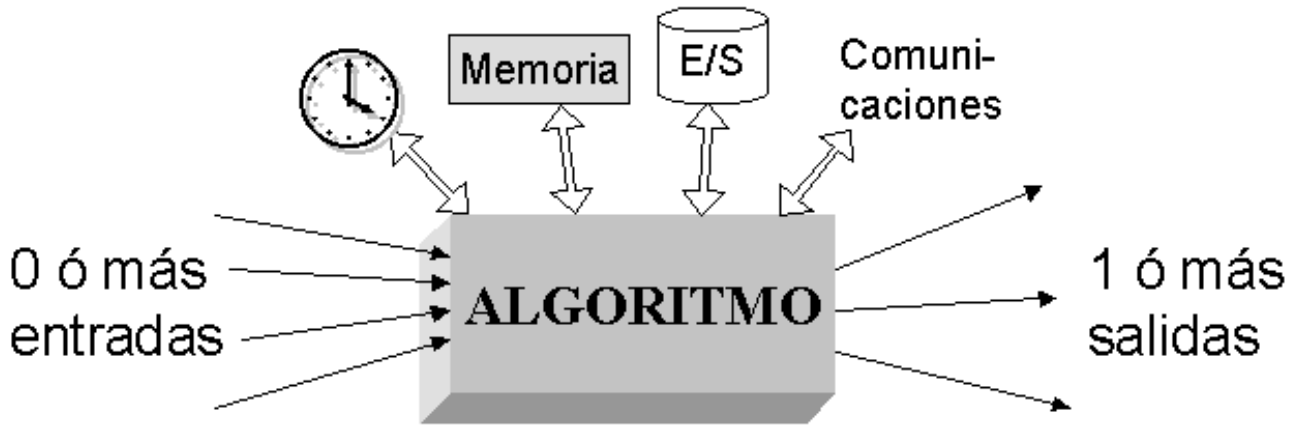
NO está en el temario pero son necesarios.

Introdution

Introducción

What is an efficient algorithm?

Our usual measure of efficiency is speed, but it dependes of computer (processor)?



Computational tractability

Estudio

In general we can analysis algorithms in the following scenarios:

1. Worst-case (Peor caso) Most used.
2. Average-Case (caso promedio)
3. Best-case (Mejor caso)

Are there other types of analysis?

2.2 Asymptotic order of growth

Crecimiento asintótico

Teoría

Ejercicios

Asymptotic order of growth

Orden asintótico de crecimiento

- Tha idea is study the behavior of algorithms when data grouths, i.e. when data (n) go to infinit.
- Little growth data is for Real time and in that case time really matters, not Growth.
- Remember to THINK BIG when working with asymptotic rates of growth.

2.2.1 Asymptotic notation

Notación asintótica

Definiciones

Bibliography

Referencias

Lista

1. https://en.wikipedia.org/wiki/Big_O_notation
2. https://en.wikipedia.org/wiki/Best,_worst_and_average_case.
3. https://en.wikipedia.org/wiki/Master_theorem

Asymptotic notation

Notación asintótica

The most common notationes are:

1. Big-Oh, O. Worst case / upper bound
2. Big-Omega, Ω lower bound
3. Big-Theta, Θ. Average case

But there are also:

1. Little-Oh, o.
2. Little-Omega, ω

And for some algorithms you use Probabilistic analysis.

Historical notes

Notas históricas

Edmund Landau
Aleman
No es Lev Landau

- Knuth traces the origin of the O-notation to a number-theory text by P. Bachmann in 1892.
- The o-notation was invented by E. Landau in 1909 for his discussion of the distribution of prime numbers.
- The Ω and Θ notations were advocated by Knuth to correct the popular, but technically sloppy, practice in the literature of using O-notation for both upper and lower bounds.
- Further discussion of the history and development of asymptotic notations can be found in Knuth and Brassard and Bratley.

Big-Oh notation, O notation

Notación O

Mathematical definition

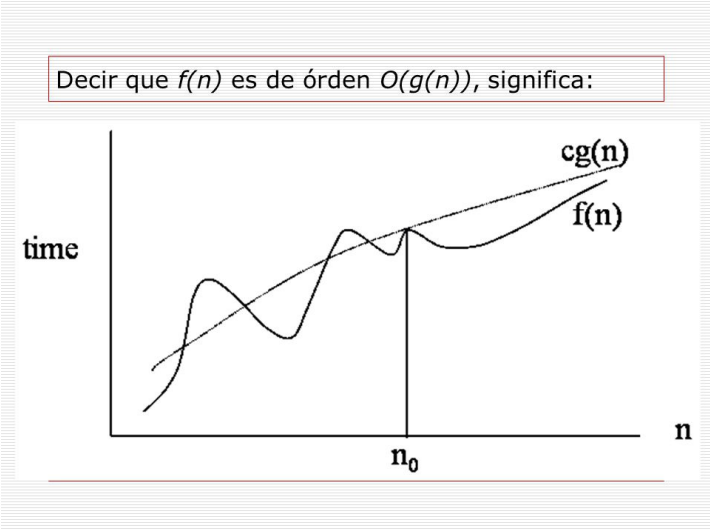
Instead of x we use n.

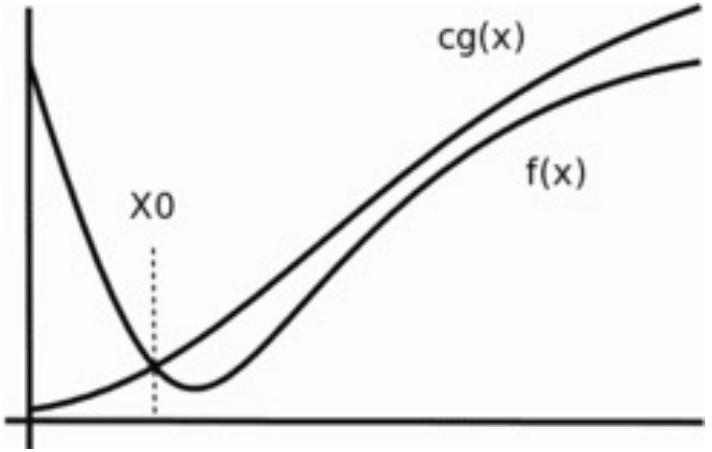
$O(g(n)) =$

{ **f(n)**: there exist positive constants c and n_0 such that /

$0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$ }

Acota por arriba





Algebraic examples

Ejemplos algebraicos

$T(n) = 1$	=> O(1) Constante
$T(n)= \log_2 n$	=> O(log n) Logarítmico
$T(n)= an + b$	=> O(n) Lineal
$T(n)= n \log_2 n$	=> O(n log n) Logarítmico
$T(n)= an^2 + bn + c$	=> $O(n^2)$ Cuadrático
$T(n)= an^3 + bn^2 + cn + d$	=> $O(n^3)$ Cúbico
$T(n)= n^m$, m=0,1,2,3...	=> $O(n^m)$ Polinomial
$T(n)= c^n$, c > 1	=> $O(n^n)$ Exponencial
T(n)= n!	=> O(n!) Factorial

Orden

Otro son $O(n^n)$ Potencial Exponencial

O(n) Sublineal, $O(n^c)$ Potencial

Practical examples

Ejercicios practicos de código

Código:

```
i=4;
printf("hello word");
```

t(n) = 0.00245 segundos?

=> O(1)

= c g(n) = 0.00245 * 1 “se dice que la” C absorve la constante

g(n) = 1

Algebraic examples

Ejemplos algebraicos

- Ejemplo 1

  $T(n) = 7n-3$ es $O(n)$

  $C=7$ $n_o=1$

  $7n-3 <= 7n$

- Ejemplo 2

  $20n^3 + 10 \log n + 5$ es $O(n^3)$

- Ejemplo 3

  $3 \log n + \log(\log(n))$ es $O(\log n)$

- Ejemplo 4

  2^{100} es $O(1)$

- Ejemplo 5

  $5/n$ es $O(1/n)$

- n^k es ?

Algebraic examples

Ejemplos prácticos

Cálculo básico (loops)

```
i=1;
while(i<n){
    i++;
}
```

Codigo equivalente

```
for(i=1; i<n; i++;){

}
```

El valor de **n** es un parámetro

Si **n** =8

i = {1,2,3,4,5,6,7}

Se ejecutan **n-1** instrucciones.

```
for(i=1; i<1; i++;){
    for(k=1; k<n; k++;);   O(n)
}
```

```
for(i=1; i<n; i++;){
    for(k=1; k<n; k++;){
        ;
    }            codigo = O(n)
}  codigo = O(n) * O(n)
           = O (n^2)
```

Algebraic examples

Ejemplos prácticos

```
i=1;                n=8
while(i<n){     1<8  2<8  4<8  8<8 => 2^0, 2^1,..
   i=i*2;           i=2   i=4    i=8
}
```

$2^0, 2^1, 2^2, 2^3, 2^x$ ... entonces en algun momento 2^x

De la condición i<n pero i toma valores 2^x

$2^x < n$

$\log_2 2^x < \log_2 n$

$x < \log_2 n$

programa es O(log n)

Algebraic examples

Ejemplos prácticos

Repaso logaritmos

$\log_b y = X \quad \leftrightarrow \quad b^x = y$

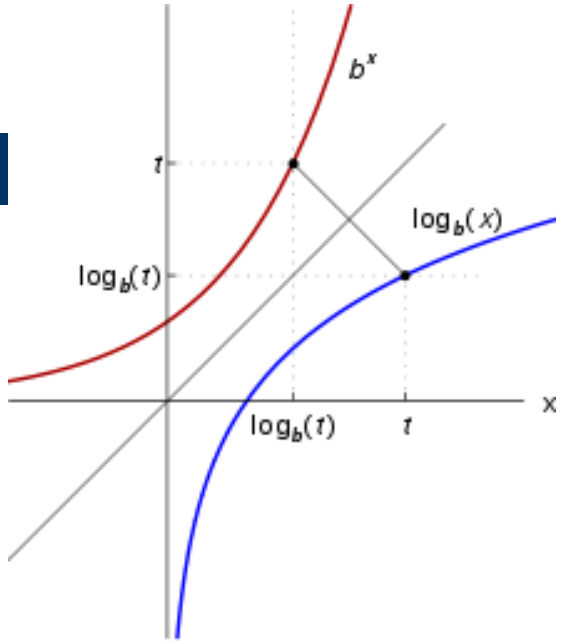
Operación inversa

Propiedades

- $\log nm = \log n + \log m$
- $\log n/m = \log n - \log m$
- $\log n^r = r \log n$
- $\log_a n = \log_b n / \log_b a$ (a y b enteros)
- $\log_b \sqrt[y]{x} = \frac{\log_b x}{y}$
- $\log_b x = \frac{1}{\log_x b}$

$e^0 = 1$
$\ln 1 = 0$
$\ln e = 1$
$\ln e^n = n$

No todas las bases y números son posibles. La base *b* tiene que ser positiva y distinta de 1.



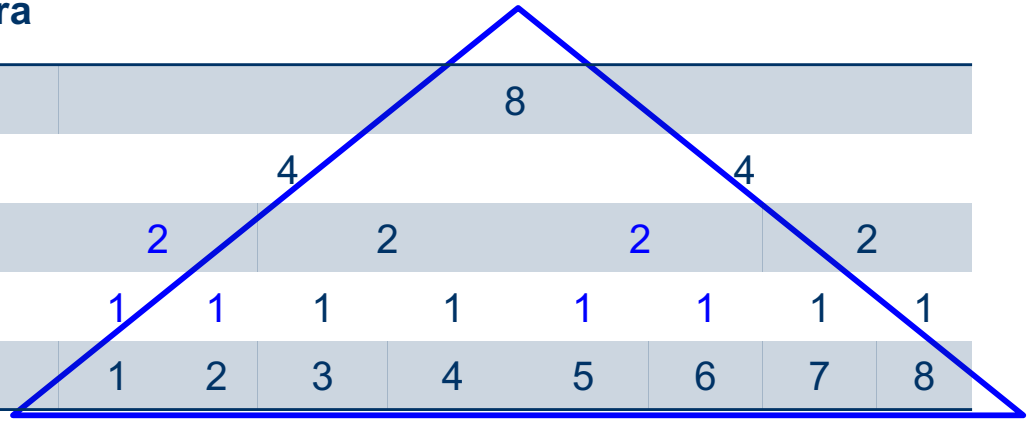
- En cualquier caso, y para todo valor apropiado de la base b, la gráfica de la función logarítmica corta al eje de las abscisas en el punto (1,0).
- El logaritmo de *n* en distintas bases, está relacionado por una constante (que es logaritmo de una base en la otra).
- Así que análisis de complejidad que se hacen en una base de logaritmos, pueden fácilmente traducirse en otra, simplemente con un factor de proporcionalidad.
- El valor de e es la base del algoritmo natural

$\ln n = \log_e n$

Algebraic examples

Ejemplos algebraicos

Nodos	Altura								
$2^3=8$	3					8			
$2^2=4$	2			4				4	
$2^1=2$	1	2		2		2		2	
$2^0=1$	0	1	1	1	1	1	1	1	1
		1	2	3	4	5	6	7	8



Niveles (iteraciones)

$$\log_2 8 + 1 = \log_2 2^3 + 1 = 3+1 = 4$$

O's algebra (some rules)

Deduciendo algunas reglas

Algebraic operations

Times

$T1(n) * T2(n) = O(f(n)*g(n))$

Particular case

$O(c * f(n)) = O(f(n))$ Por la definición de O se dice que absorve c.

Ejemplo

$O(n^2/2) = O(1/2\ n^2) = O(n^2)$

Algebraic examples

Ejemplos prácticos

```
i=n*2;
while(i>2)
{
   instrucción 1;
   for(j=0; j<n/2; j++)
      instrucción 2;
   for(k=n; k>1; k--) {
      for(m=1; m<10; m++)
         instrucción 3;
   }
   i=i/3;
}
```

Algebraic examples

Ejemplos prácticos

$T_{while} = T_{i1} + T_{fj} + T_{fk} + T_i$
$= c_1 + (c_2 n + c) + (c_3 n + c) + c_4$
$= c_5 n + C_6$

```
i=n*2;   O(1)
while(i>2)
{
   instrucción 1;  O(1)
   for(j=0; j<n/2; j++)
      instrucción 2;  O(1)
   for(k=n; k>1; k--) {
      for(m=1; m<10; m++)
         instrucción 3;  O(1)
   }
   i=i/3;  O(1)
}
```

O(1) → O(n) → O(n) → O(n) → O(n) → O(n log$_3$n)

O(n): O(n)+O(n)

O(n log$_3$n): O(n)*O(log$_3$n)

Practical examples

Estructuras de control

Paradigma de Programación: imperativo

Por default el orden en que se ejecutan las instrucciones es secuencial:

- Flujo del programa: una **secuencia** de instrucciones.
- Se puede alterar con las Estructuras de control de flujo.

instrucción-1; => $O(i_1)$, i_1 es una función

Instrucción-2; => $O(i_2)$

```
a=5;
funcion(parametros);
```

...

Instrucción-N; => $O(i_N)$

$$O_{programa} = O(i_1) + O(i_2) + \dots O(i_N)$$
$$= O(i_1 + i_2 + \dots + i_N)$$

Si los tmpos fueran contantes

$= O(0.001+0.002+...+0.003) = O(0.006) = O(C*1)$

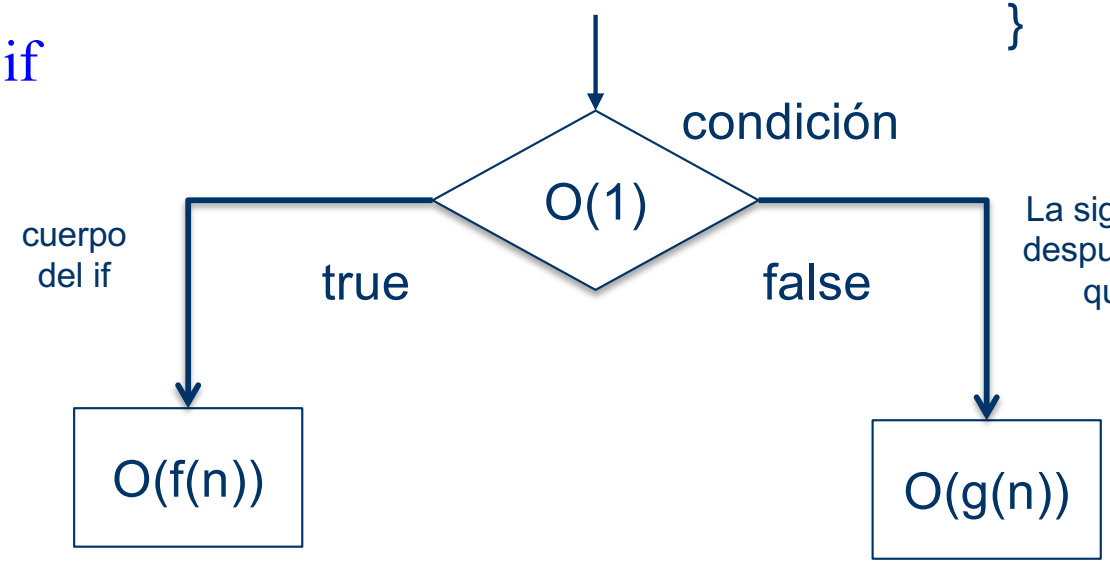
$C=3$

Practical examples

Estructuras de control

```
if(condición)
    instrucción_cuerpo

if(condición){
    instruccionES_cuerpo
}else{
}
```

Sentence if



La siguientes instrucción después del cuerpo del if que seria un else

Peor caso

$$O(\max(f(n), g(n)))$$

$f(n)=n, \quad g(n)=n^2$

$O(\max(n,n^2)) = O(n^2)$

Mejor caso

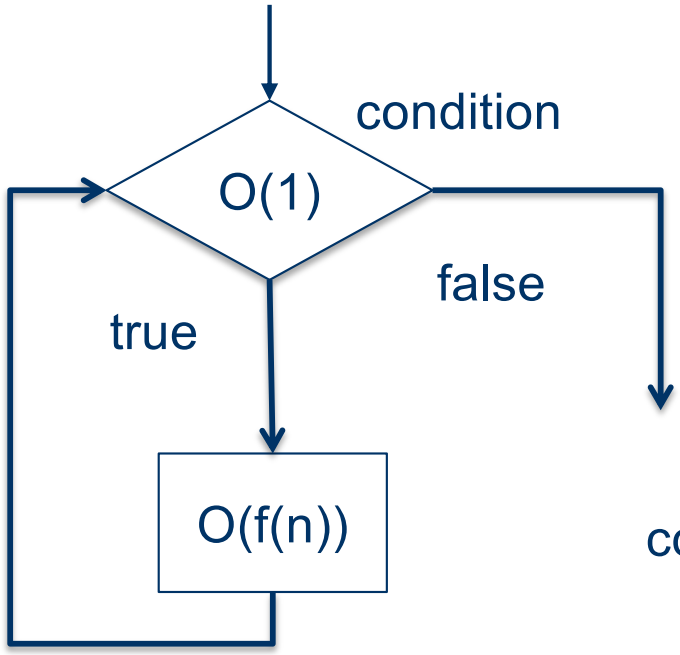
$$O(\min(f(n), g(n)))$$

Practical examples

Estructuras de control

```
while(condición){
    cuerpo
}
```

Sentence while



O(1 * g(n) * f(n))

condition
    var == valor
    var == fun(param)

O(cond(n) g(n) f(n))

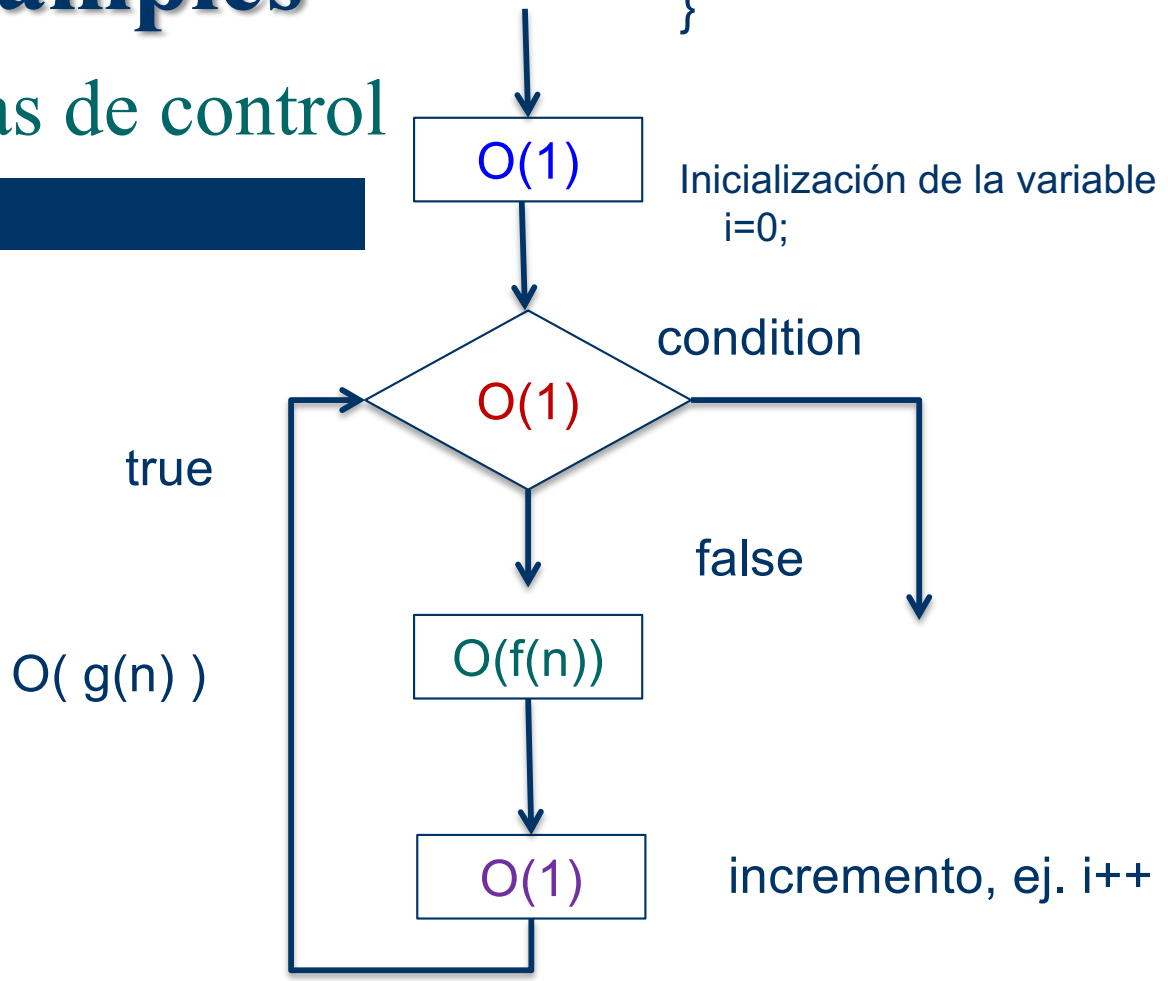
Que pasa si la condición no es O(1)?

Practical examples

Estructuras de control

Sentence for

```
for(i=1; i<=n; i++){
    cuerpo
}
```



O(g(n) f(n))

Que pasa si el incremento/decremento no es una asignación?

Practical examples

Estructuras de control

Sentence for

```
string = “cadena”;
for(cnt=1; cnt <= strlen(string); cnt++){
   printf(“%d”, cnt);
}
```

O(1)

```
scanf(“%s”, &string);
for(cnt=1; cnt <= strlen(string); cnt++){
   printf(“%d”, cnt);
}
```

O(ini) = O(1)
O(cond) = O(n)
O(cuerpo) = O(1)
O(modif_cnt)

O(f(n)) = O(cuerpo)= O(1)
O(g(n)) = O(n)

= O(1) + [O(n) O(1) O(1) O(n)]
= $O(n^2)$

```
var_lim = strlen(string);
for(cnt=1; cnt <= var_lim; cnt++){
   printf(“%d”, cnt);
}
```

O(n)

Practical examples

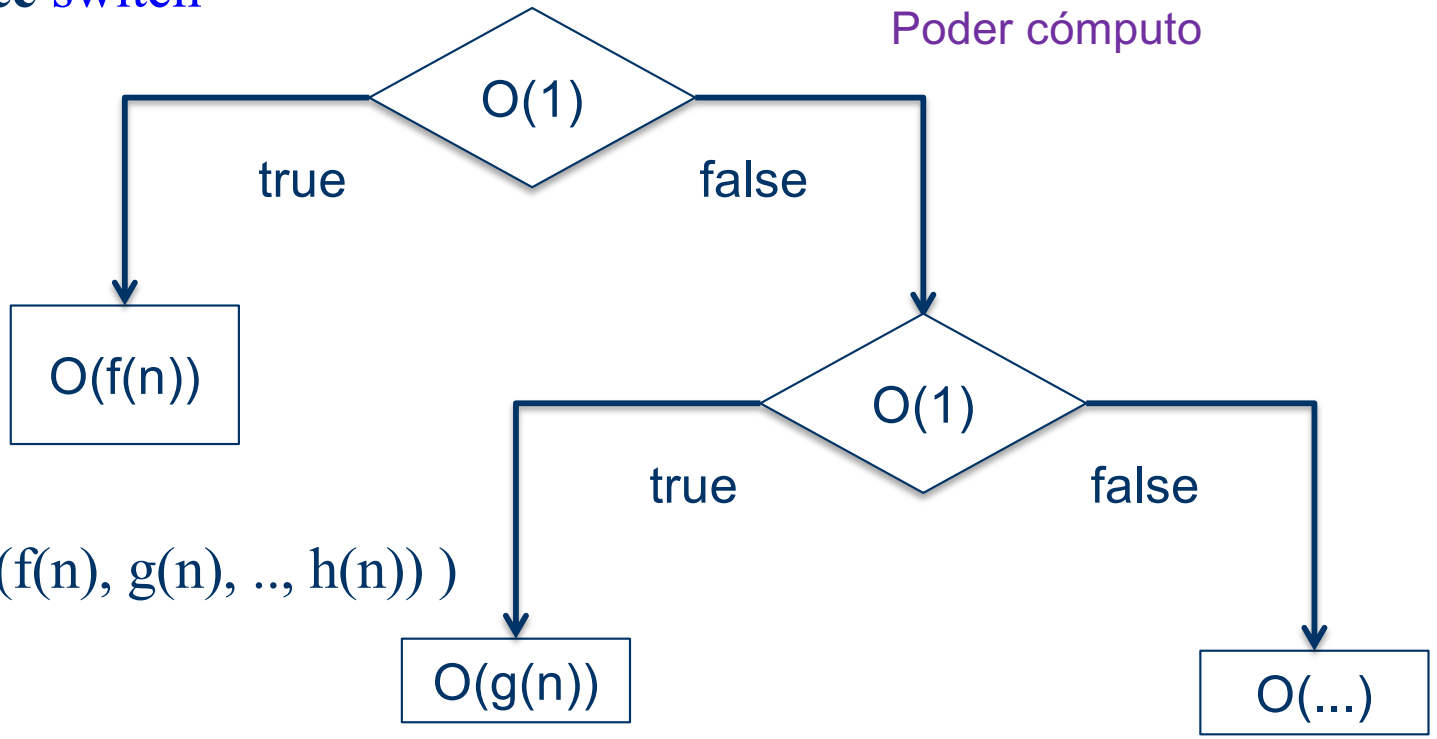
Estructuras de control

```
If( cond_1 ){

}else if( cond_2 ){

}else{
    //default
}
```

```
switch( var ){
  case valor1:
            inst1;
            break;
  case valor2:
            inst2;
            break;
```

Facilidad sintáctica
Semántica
Poder cómputo

Sentence switch



O(max(f(n), g(n), .., h(n)))

Practical examples

Estructuras de control

Ejemplo “complejo”

For

While

Do while

If else

switch

Problems notation

Problemas de notación

Is comon to write:

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

La comparación **=**
tambien es un operador con sus propiedades
la cual suele ser reflexiva pero para O ...

but we have:

función = Conjunto de funciones

$f = \{ f, g, h, \ldots \}$

that is, they are not equivalent/comparable.

The correct way is $f(n) \in O(g(n))$

So it is an abuse of the notation that in some cases gives us problems:

Investigar y dar ejemplos

Landau Notation

Notación Landau

- Is used in math & algo for symbolically expressing the asymptotic behavior of a given function.
- Definition for functions. It is what we have seen.
- Definition for sucesiones:

  No es tema del curso sin embargo ...

Tarea optativa

- Investigar la notación Landau y entregar resumen en no más de 3 cuartillas.
  - Una **sucesión** puede definirse como una función sobre el conjunto de los números naturales (o un subconjunto del mismo, y es por tanto una función discreta) y su codominio es cualquier otro conjunto, generalmente de números, figuras geométricas o funciones.
  - No confundir con una **serie** matemática, que es la suma de los términos de una sucesión.

Properties of O

Propiedades de O

Sea $E \subseteq R$, sean $f_1:E\rightarrow R$, $g_1:E\rightarrow R$, $f_2:E\rightarrow R$, $g_2:E\rightarrow R$ funciones y k un real. Entonces los siguientes enunciados son ciertos:

1. Si $f_1 = O(g_1)$ y $g_1 = O(g_2)$, entonces $f_1 = O(g_2)$
2. Si $f_1 = O(g_1)$ y $f_2=O(g_2)$, entonces $f_1 f_2 = O(g_1 g_2)$ Producto
3. $f_2\ O(g_1) = O(f_2 g_1)$ Igualdad entre conjuntos. Producto
4. Si $f_1 = O(g_1)$ y $f_2=O(g_2)$, entonces $f_1+f_2 = O(|g_1| + |g_2|)$ Suma
5. Si f_1 y g_1 son funciones positivas, $f_1+O(g1) = O(f_1 + g_1)$ Suma
6. Si $f_1= O(g_1)$, entonces k $f_1=O(g_1)$ Multiplicación
7. Si $k \neq 0$ entonces $O(kg_1) = O(g_1)$ Igualdad entre conjuntos. Mult.

Big-Omega notation, Ω notation

Notación Ω

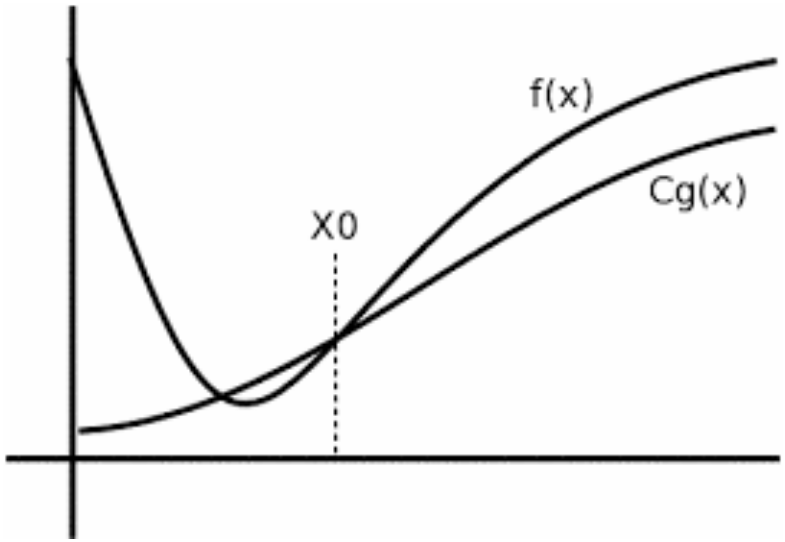
Mathematical definition

$\Omega(g(n)) =$

{ f(n): there exist positive constants c and n_0 such that

$0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }

Acota por debajo



Big-Theta notation, Θ notation

Notación Θ

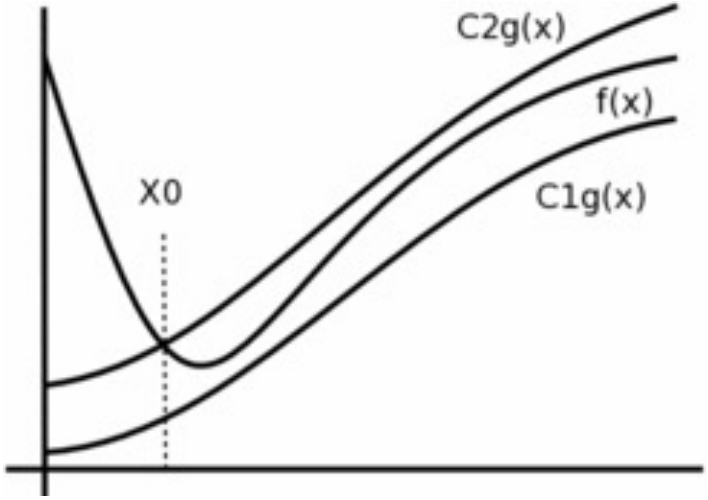
Mathematical definition

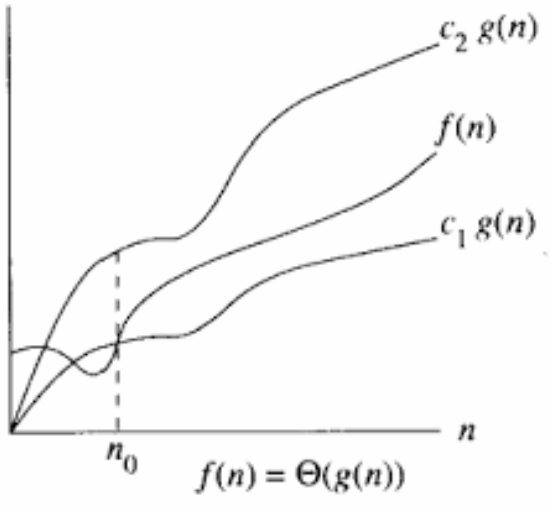
Acota por arriba y debajo
Cota ajustada

$\Theta(g(n)) =$

{ f(n) : there exist positive constants c_1, c_2, and n_0 such that

$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$ }





Big-Theta notation, Θ notation

Notación Θ

In orther words:

f(n) = Θ(g(n)) si y solo si

f(n) = O(g(n)) y

f(n) = Ω(g(n))

Properties of Θ

Propiedades de Θ

1. Dualidad

   $g(n) \in O(f(n)) \Leftrightarrow$ f(n) $\in \Omega(g(n))$

2. Clasificación

   $g(n) \in \Theta(f(n)) \Leftrightarrow$ f(n) $\in \Theta(g(n))$

3. Eliminación de términos de menor peso

   $Si\ c \geq 0, d > 0, g(n) \in O(f(n)) y\ h(n) \in \Theta(f(n))$, entonces

   $$cg(n) + dh(n) \in \Theta(f(n))$$

4. Límite

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} c \in R^+, \text{ entonces } f(n) \in \Theta(g(n)) \\ 0, \text{ entonces } f(n) \in O(g(n)), \text{ pero no a } \Theta(g(n)) \\ \infty, \text{ entonces } f(n) \in \Omega(g(n)), \text{ pero no a } \Theta(g(n)) \end{cases}$$

Little-o notation

Notación o

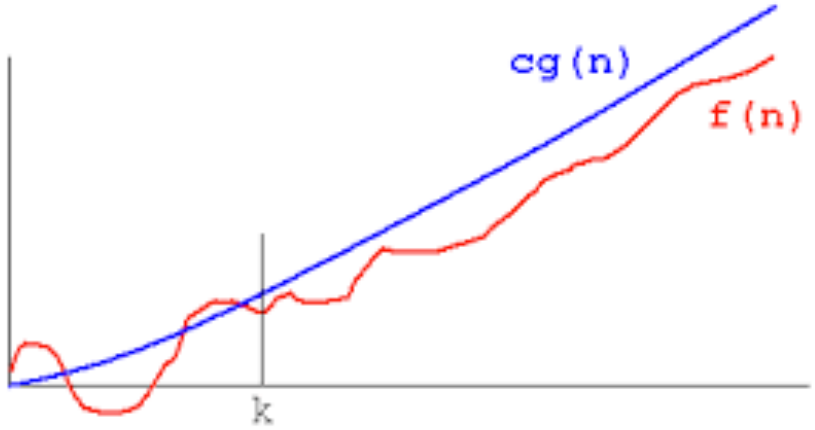
Mathematical definition

$o(g(n))$ = estrictamente menor

{ $f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that

$0 \leq f(n) < cg(n)$ for all $n \geq n_0$ }

The value of n_0 must not depend on n, but may depend on c.



Little-o notation

Notación o

Definition with limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \lim_{n\to n_0} \frac{f(n)}{g(n)} = 0$$

For some problems its useful to use this definition.

Analogy

Analogias

Notation	Definition	Analogy
$f(n) = O(g(n))$	see slides	<=
$f(n) = o(g(n))$	see slides	<
$f(n) = \Omega(g(n))$	$g(n)=O(f(n))$	>=
$f(n) = \omega(g(n))$	$g(n)=o(f(n))$	>
$f(n) = \Theta(g(n))$	$f(n)=O(g(n))$ and $g(n)=O(f(n))$	=

Properties of Asymptotic notation

Propiedades de las notaciones asintóticas

Many of the relational properties of **real numbers** apply to asymptotic comparisons as well:

- Transitivity
- Reflexivity
- Symmetry
- Transpose symmetry

But **Trichotomy** does not carry over to asymptotic notation.

Tarea optativa

- Investigar la **Tricotomia** y entregar resumen en no más de 3 cuartillas.

2.2.2

Standars notation & common functions

Notación estándar y funciones típicas

Definiciones

Standars notation & common functions

Notación estándar y funciones típicas

Monotonicity

- A function f(n) is ***monotonically increasing***

  if $m \leq n$ implies $f(m) \leq f(n)$

- Similarly, it is ***monotonically decreasing***

  if $m \leq n$ implies $f(m) \geq f(n)$

- A function f(n) is:
  - ***Strictly increasing*** if $m < n$ implies $f(m) < f(n)$
  - ***Strictly decreasing*** if $m < n$ implies $f(m) > f(n)$
- For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the ***floor*** of x"), and
- The least integer greater than or equal to x by $\lceil x \rceil$ (read "the ***ceiling*** of x"). For all real x.

Useful mathematical tools

Herramientas matemáticas utiles

For some of the calculations you might find L'Hôpital's rule helpful. Considere:

Regla que usa derivadas para ayudar a evaluar límites de funciones que estén en forma indeterminada: expresion que involucra límites de la forma 0/0, ∞/∞, etc.

$\lim_{(n \to \infty)} f(n)/ g(n) = 0$

this gives you $\Theta(f(n)) < \Theta(g(n))$

$\lim(n \to \infty) f(n)/ g(n) = c;\ c > 0$

this gives you $\Theta(f(n)) = \Theta(g(n))$

$\lim(n \to \infty) f(n)/ g(n) = \infty$

this gives you $\Theta(f(n)) > \Theta(g(n))$

2.3 A survey of common running times

Tiempos de ejecución típicos

Gráfica y
Funciones (series)

A survey of common running times

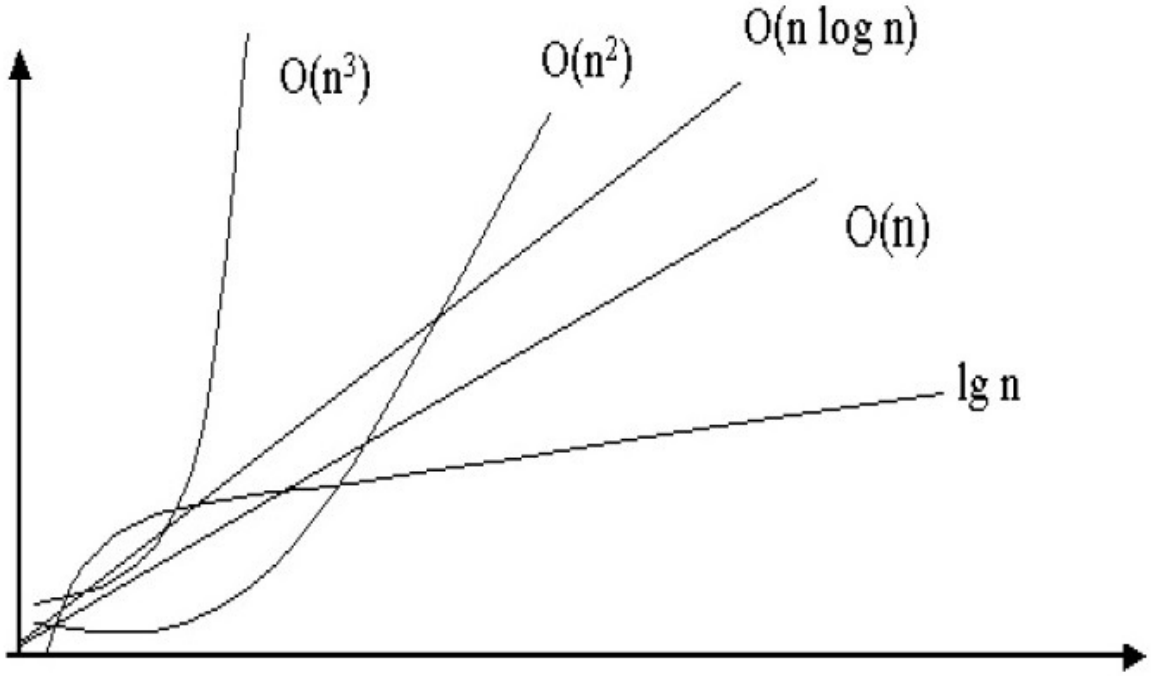
Estudio de tiempos de ejecución típicos

1. Sublinear time, O(log n)
2. Linear time, O(n).
3. Linearithmic time, O(n log n)
4. Cuadratic time, $O(n^2)$
5. Cubic time, $O(n^3)$
6. Polynomial time , $O(n^k)$
7. Exponential time, $O(2^n)$

A survey of common running times

Estudio de tiempos de ejecución típicos

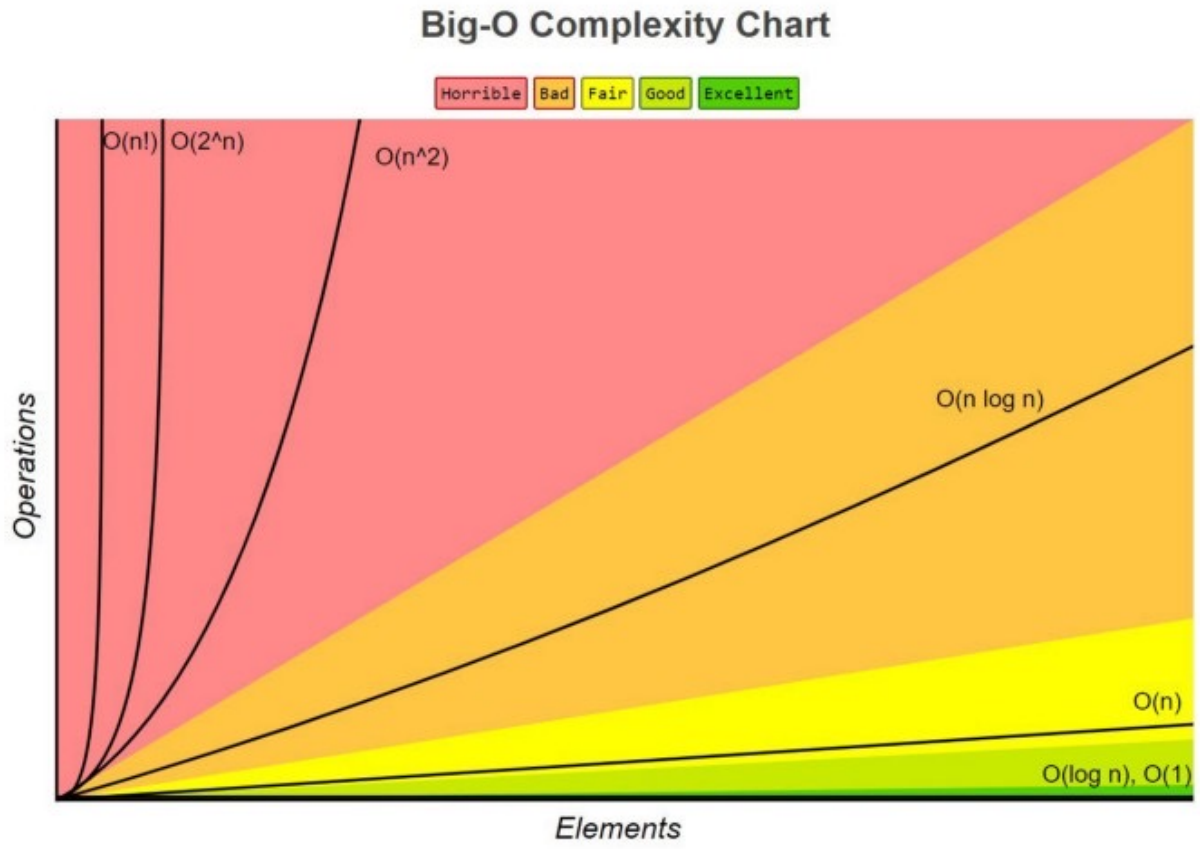
Gráficas de las complejidades típicas:



A survey of common running times

Estudio de tiempos de ejecución típicos

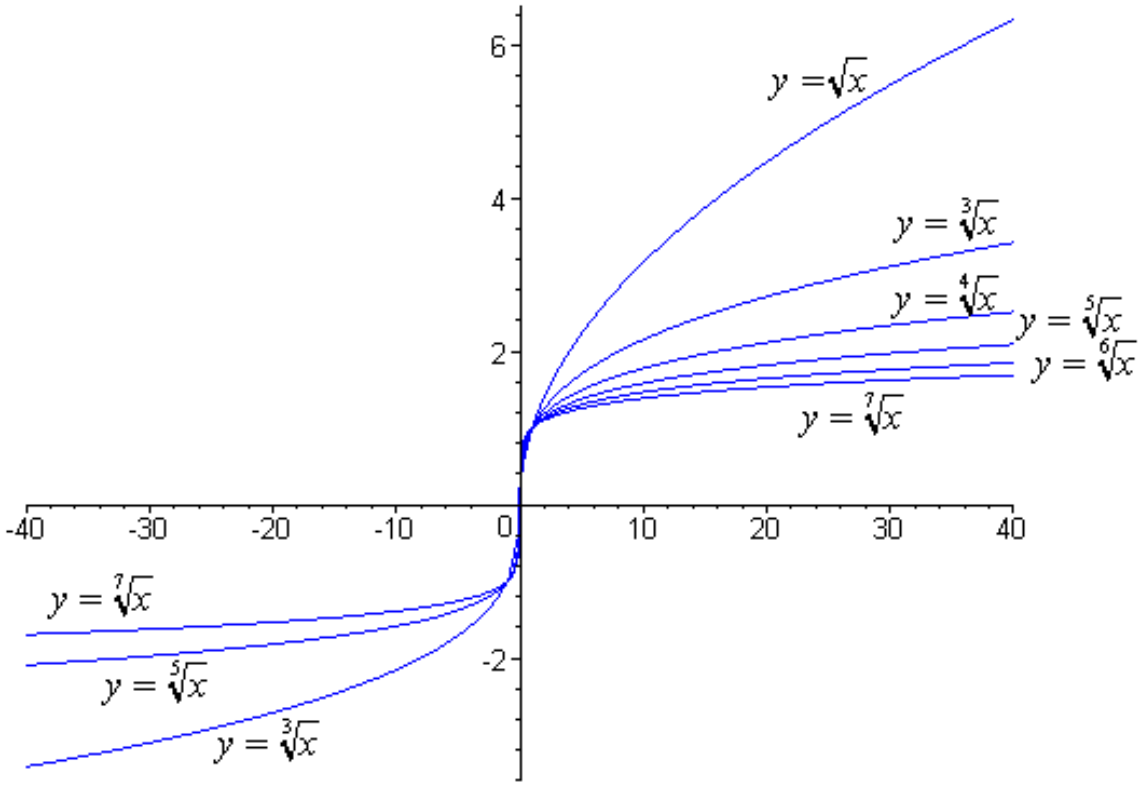
Gráficas de las complejidades típicas:



A survey of common running times

Estudio de tiempos de ejecución típicos

Otras gráficas: **Raíces**



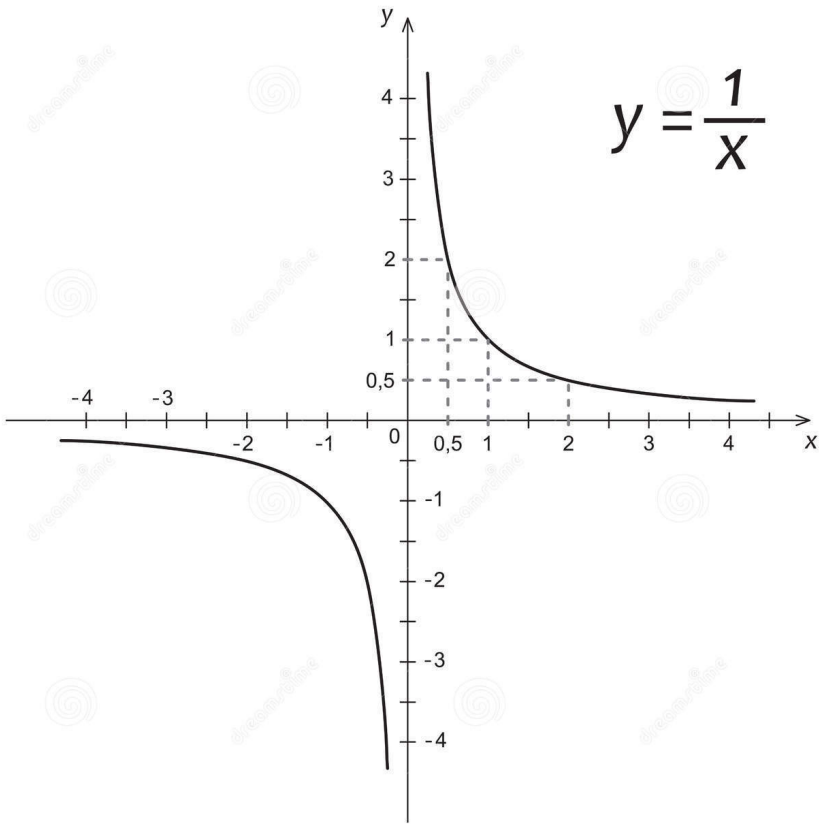
Polinomios de Taylor

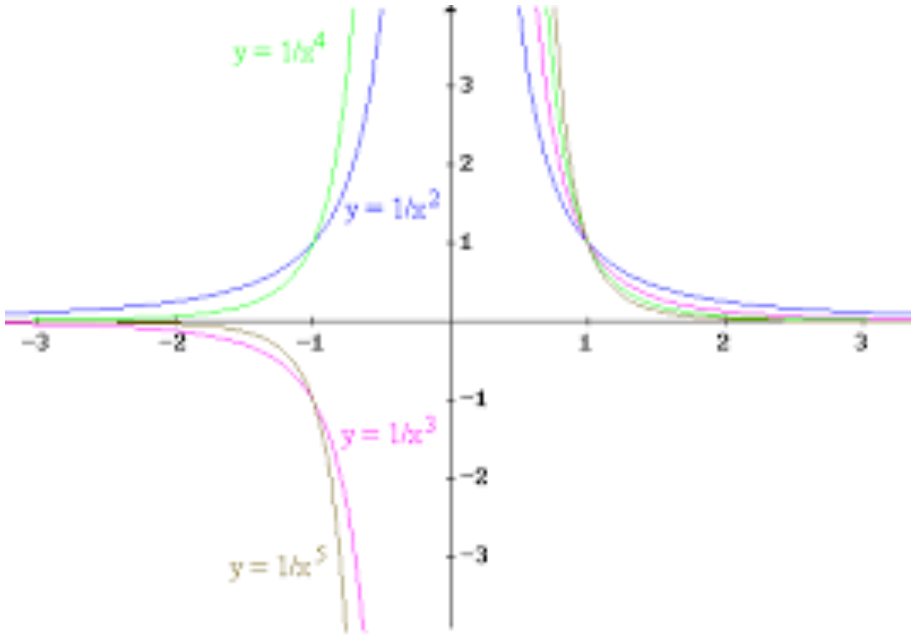
$$\sqrt{1+x} = 1 + \frac{x}{2} + \frac{x^2}{8} + \frac{x^3}{16}$$

A survey of common running times

Estudio de tiempos de ejecución típicos

Otras gráficas: **Hipérbolas**

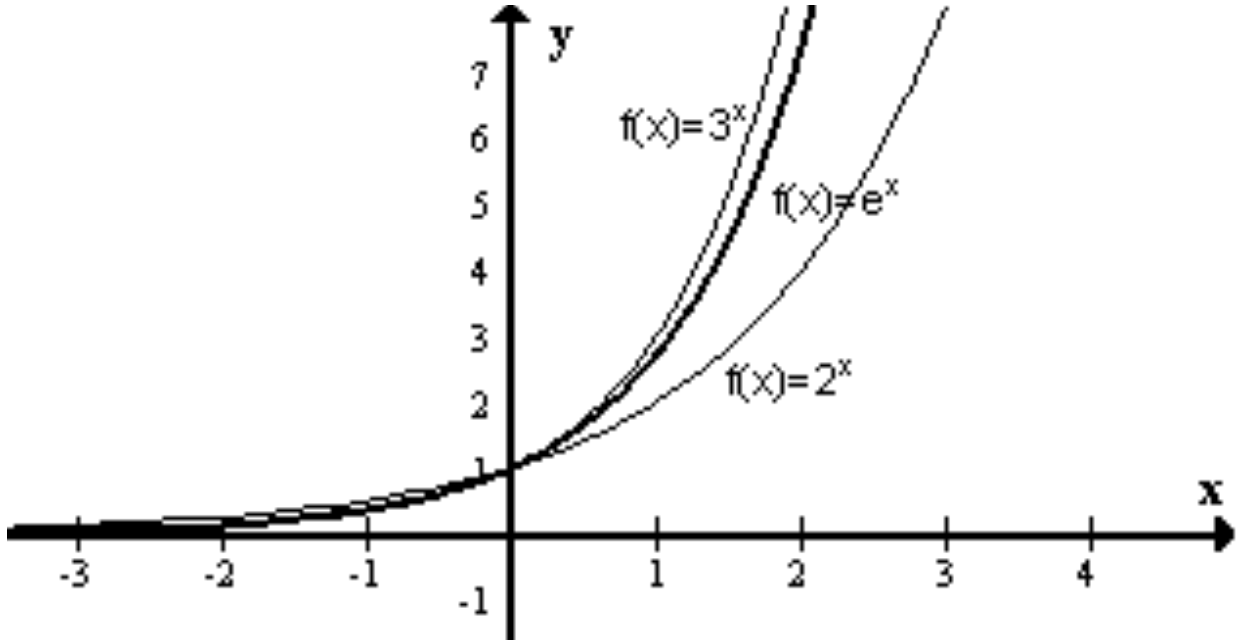




A survey of common running times

Estudio de tiempos de ejecución típicos

Otras gráficas: **Exponencial** $y = a^x$



Leyes de los exponentes

$$a)(a^x)(a^y) = a^{x+y}$$

$$b)\frac{a^x}{a^y} = a^{x-y}$$

$$c)\left(a^x\right)^y = a^{xy}$$

$$d)(ab)^x = a^x b^x$$

$$e)\left(\frac{a}{b}\right)^x = \frac{a^x}{b^x}$$

Exercises

Ejercicios

Y tareas

Homeworks

Tareas

Basic problems 2

Problemas básicos

- Order the following expressions in increasing Θ-order.
- If two functions are of the same order of growth, you should state this fact.

E1: $n \log n$, n^{-1}, $\log n$, $n^{\log n}$, $10n + n^{3/2}$, π^n, 2^n, $2^{\log n}$, $2^{2^{\log n}}$, $\log n!$

1 2 3 4 5 6 7 8 9 10

E2: 2^{2^n}, 2^{n^2}, $n^2 \log n$, n, n^{2n}

Respuesta E1

Tiempo de 15 min para responder.

Homeworks

Tareas

Basic problems 3

Problemas básicos

Demostrar

Si f(n) = O(s(n)) y g(n) = O(r(n)) entonces

f(n) + g(n) = O(s(n)+r(n))

Si f(n) = O(s(n)) y g(n) = O(r(n)) entonces

f(n) * g(n) = O(s(n)*r(n))

Usando las propiedades que definen el algebra llegar a la conclusión.

Trichotomy

Mathematics

- In mathematics, the Law of Trichotomy states that every real number is either positive, negative, or zero.

- More generally, trichotomy is the property of an order relation < on a set X that for any x and y, exactly one of the following holds:

$$x<y,\ x=y,\ \text{or}\ x>y$$

Types of Problems

Tipos de Problemas

Introducción

Types of Problems

Introducción

Una clasificación informal es:

- Problemas triviales
- Problemas de decisión
- Problemas de búsqueda
- Problemas de conteo o enumeración
- Problemas de optimización.

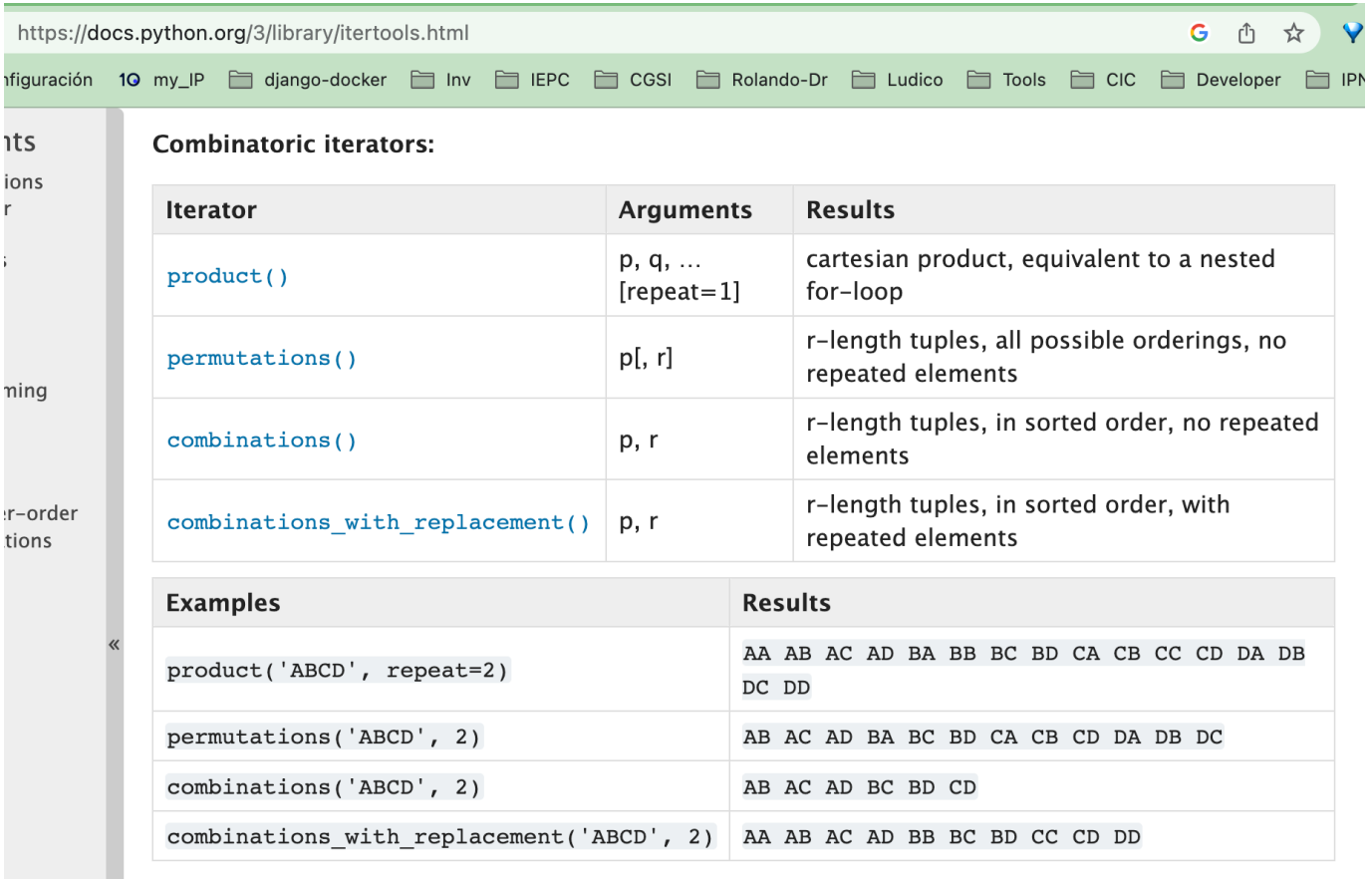
Types of Problems

Introducción

https://docs.python.org/3/library/itertools.html

Problemas de conteo o enumeración

- Permutations
- Combinations
- Power Set



https://docs.python.org/3/library/itertools.html

figuración 10 my_IP django-docker Inv IEPC CGSI Rolando-Dr Ludico Tools CIC Developer IPN

Combinatoric iterators:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

Types of Problems

Introducción

Problemas de conteo o enumeración

- Permutations
  - The number of permutations of *n* numbers is *n!*.
  - There are six possibilities for three objects: 3! = 6.
  - This is manageable, but as the number of objects increases, the number of permutations increases exponentially.

Number of Objects	Number of permut
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
20	2.432902e+18
30	2.6525286e+32
40	8.1591528e+47

Types of Problems

Introducción

Problemas de conteo o enumeración

- Combinations
- Hacer un programa que genere el C(n,r).
- Given an array of size n, generate and print all possible combinations of r elements in array.

  For example, if input array is {1, 2, 3, 4} and r is 2, then

  Output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}

$$ {}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!} $$

Types of Problems

Introducción

Problemas de conteo o enumeración

- Power Set
  - Power set P(S) of a set S is the set of all subsets of S.
  - For example S = {a, b, c} then

    P(s) = {{}, {a}, {b}, {c}, {a,b}, {a, c}, {b, c}, {a, b, c}}
  - If S has n elements in it then P(s) will have 2^n elements.
- Hacer un programa que genere el P(s).

Types of Problems

Introducción

Intento de solución

Tomaremos 10 min para que cada quien intente solucionar un problema como pueda.

- Puede ser fuerza bruta.
- Puede ser iterativo o recursivo.
- Que sean funciones con parámetros.

Luego veremos las soluciones conocidas!

Types of Problems

Introducción

Soluciones

Resumen

Sol	Permutations	Combinations	Power Set
1	Heap's algorithm 1963 *decrease and conquer*	Fix Elements and Recursion	Generating all binary numbers
2		Include and Exclude every element	Sorted by cardinality
3			Backtrack

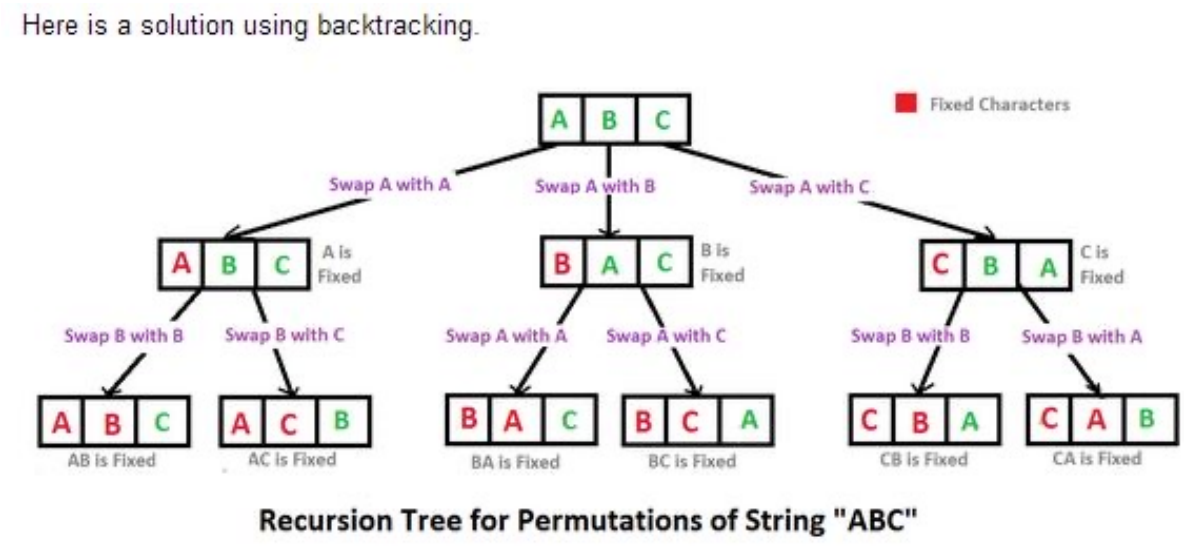
Types of Problems

Introducción

It was first proposed by B. R. Heap in 1963.
https://en.wikipedia.org/wiki/Heap's_algorithm

Soluciones

- Permutations
- **Heap's algorithm** is used to generate all permutations of n objects. The idea is to generate each permutation from the previous permutation by choosing a pair of elements to interchange, without disturbing the other n-2 elements.
- Example
  - Input: 1 2 3
  - Output: 1 2 3
    1 3 2
    2 1 3
    2 3 1
    3 1 2
    3 2 1



Recursion Tree for Permutations of String "ABC"

Types of Problems

Introducción

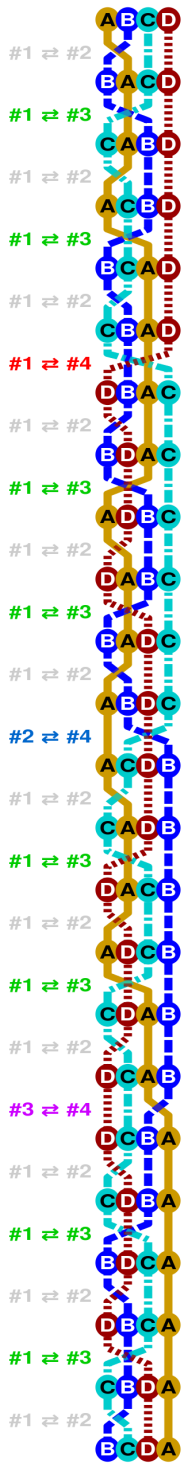
Soluciones

- **Heap's algorithm** / permutations.

```
void heapPermutation(int a[], int size, int n)
{
    if (size == 1) { printArr(a, n); return; }

    for (int i = 0; i < size; i++) {
        heapPermutation(a, size - 1, n);
        if (size % 2 == 1)
            swap(a[0], a[size - 1]);
        else
            swap(a[i], a[size - 1]);
    }
}
```

Revisar la Versión iterativa.

It was first proposed by B. R. Heap in 1963.
https://en.wikipedia.org/wiki/Heap's_algorithm



Types of Problems

Introducción

Problemas de conteo ...

- Permutations

Algorithm 3: NonRecursiveHeapsAlgorithm

```
Data:
GeneratedPermutations: generated permutations,
ElementsToPermute: elements to permute (also initial permutation),
Length: The length of the array
c[] ⟵ initArray(length, 0);
output this permutation, ElementsToPermute;
i ⟵ 0 ;
while i < Length do
    if c[i] < i then
        if i is even then
            SwapElements(0, i);
        else
            SwapElements(c[i], i);
        end
        add ElementsToPermute to GeneratedPermutations;
        c[i] ⟵ c[i] + 1;
        i ⟵ 0;
    else
        c[i] ⟵ 0;
        i ⟵ i + 1;
    end
end
```

Leer una explicación amplia:

- https://www.baeldung.com/cs/array-generate-all-permutations.
- Presenta 5 algoritmos
  - The principle of Heap's algorithm is decrease and conquer.

Types of Problems

Introducción

Soluciones

- Combinations

  There are two Methods

  1. Fix Elements and Recursion
  2. Include and Exclude every element.

  Ambos métodos no consideran elementos repetidos pero pueden ser adaptados.

Types of Problems

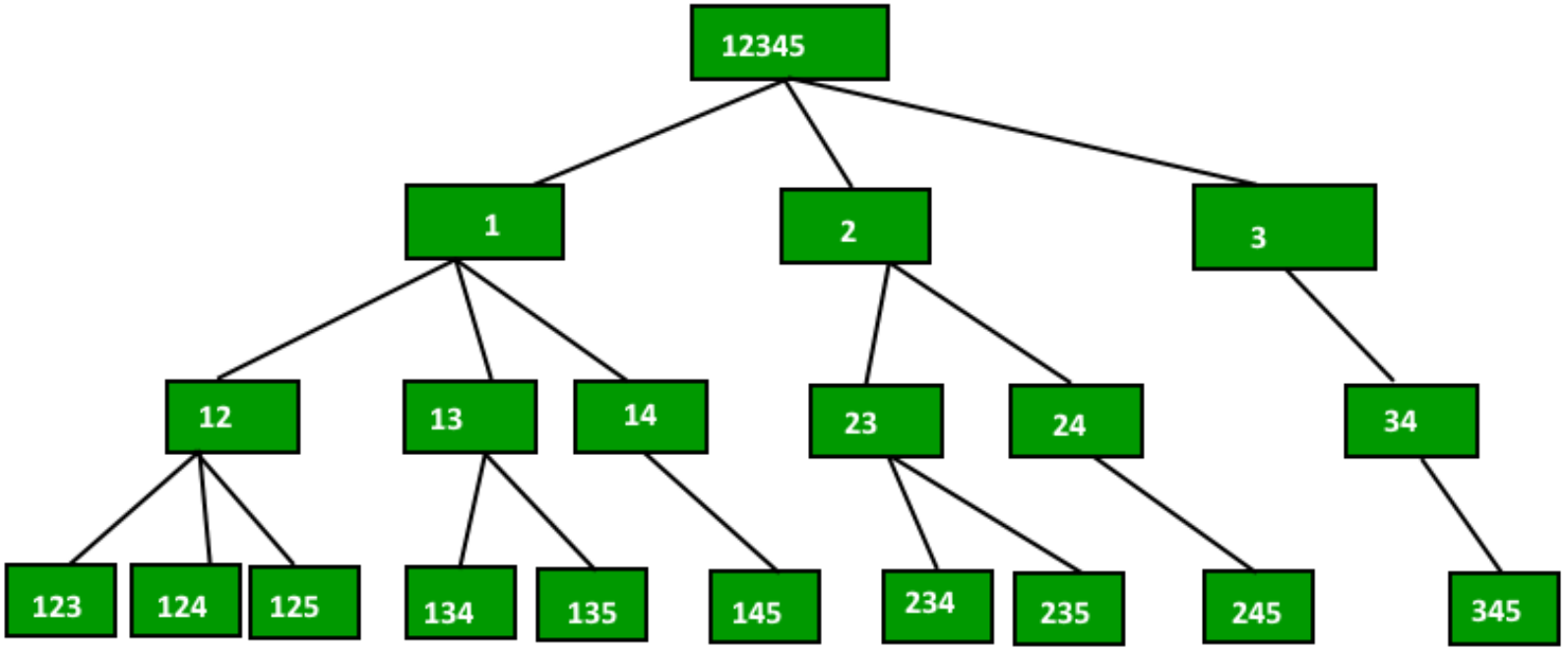
Introducción

Soluciones

- Combinations.

  Methods-1: Fix Elements and Recursion

  Time Complexity: $O(n^2)$



Types of Problems

Introducción

Soluciones

- Combinations. **Methods-1**: Fix Elements and Recursion

```
def combinationUtil(arr, data, start, end, index, r):
    if (index == r):
        for j in range(r):
            print(data[j], end = " ");
        print();
        return;

    i = start;
    while(i <= end and end - i + 1 >= r - index):
        data[index] = arr[i];
        combinationUtil(arr, data, i + 1, end, index + 1, r);
        i += 1;

arr = [1, 2, 3, 4, 5];
r = 3;
printCombination(arr, len(arr), r);
```

```
def printCombination(arr, n, r):
    data = [0]*r;
    combinationUtil(arr, data, 0, n - 1, 0, r);
```

Types of Problems

Introducción

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Soluciones

- Combinations. **Methods-2**: Include and Exclude every element.
  - This method is mainly based on *Pascal's Identity*. The idea here is similar to *Subset Sum Problem*.
  - We one by one consider every element of input array, and recur for two cases:
    1. The element is included in current combination

       We put the element in data[] and increment next available index in data[]
    2. The element is excluded in current combination

       We do not put the element and do not change index. When number of elements in data[] become equal to r (size of a combination), we print it.

Types of Problems

Introducción

Soluciones

- Combinations. **Methods-2**: Include and Exclude every element.

```
def combinationUtil(arr, n, r, index, data, i):
    if (index == r):
        for j in range(r):
            print(data[j], end = " ")
            print()
            return

    if (i >= n):
        return

    data[index] = arr[i]
    combinationUtil(arr, n, r, index + 1, data, i + 1)
    combinationUtil(arr, n, r, index, data, i + 1)

arr = [1, 2, 3, 4, 5]
r = 3
printCombination(arr, len(arr), r)
```

```
def printCombination(arr, n, r):
    data = [0]*r;
    combinationUtil(arr, n, r, 0, data, 0);
```

Types of Problems

Introducción

Soluciones

- Power Set
- Hacer un programa que genere el P(S), donde S es el conjunto de datos de entrada.
- Geek for Geeks
  1. Generating all binary numbers.
  2. Sorted by cardinality
  3. Backtrack

Types of Problems

Introducción

Soluciones

- Power Set. **Metodo-1**

  For a given set[] S, the power set can be found by:

  - Generating all binary numbers between 0 and 2^n-1, where n is the size of the set.
  - For example: for the set S {x, y, z}
    - Generate all binary numbers from 0 to 2^3-1 and
    - For each generated number, the corresponding set can be found by considering set bits in the number.

- **Time Complexity:** O(n2^n)
- **Auxiliary Space:** O(1)

Types of Problems

Introducción

Soluciones

- Power Set. **Metodo-2**

  In auxiliary array of bool set all elements to 0. That represent an empty set.

  Set first element of auxiliary array to 1 and generate all *permutations* to produce all subsets with one element.

  Then set the second element to 1 which will produce all subsets with two elements, repeat until all elements are included.

- **Time Complexity:** $O(n2^n)$
- **Auxiliary Space:** O(n)

Types of Problems

Introducción

Soluciones

- Power Set. **Metodo-3**

  Use backtrack, we have two choices:

  - First consider that element then
  - Don't consider that element.

- **Time Complexity:** $O(n2^n)$
- **Auxiliary Space:** O(n)

Algorithm design paradigm

Paradigmas de diseño de algoritmos

Introducción

Algorithm design paradigm

Introducción

Definiciones

- El término **paradigma** tiene varios significados derivados de su evolución. Nosotros usaremos: "*El conjunto de prácticas y teorías que definen algo*". La parte de teorias incluye los términos de: modelos y conceptos; la parte de prácticas incluye métodologias y procedimientos.
  - Lean: https://es.wikipedia.org/wiki/Paradigma
- Paradigmas principales
  - Graph
  - Greedy
  - Divide and Conquer
  - Dynamic Programming

Otras taxonomias

- Branch and bound.
- Branch and cut / prunnig.

Referencias

- https://en.wikipedia.org/wiki/Branch_and_bound
- https://en.wikipedia.org/wiki/Branch_and_cut

Algorithm design paradigm

Introducción

Mapa conceptual

Problemas combinatorios y de optimización
- State space search
- Backtracking => Greedy y DP => Heurística
- Constraint programming and Operations Research

Algoritmos
- Iterativos
- Recursivos

Programación
- Secuencial
- Paralela o concurrente

Programación y Algoritmia

Linear programmig
Network flow

Paradigmas de la programación
- Imperativo
- Funcional
- Lógico
- OO

Paradigmas de diseño
- Greedy
- Divide and Conquer
- Dynamic Programming

Algorithm design paradigm

Introducción

Material

Temas	**Slides**
1. Grafos	83 / 17.18%
2. Greedy	130 / 26.91%
3. Divide and Conquer	155 / 32%
	97 ejemplos + 66 math
4. Dynamic Programming	115 / 23.8%
	483

Como todos son importantes los veremos concurrentemente.

Aprox. 2 semanas y cambio de tema.

Several topics

Varios temas

Notas históricas

Problemas abiertos

Tareas

URLs

A recordar

Open problemes or Further topics

Problemas sin resolver

- Similar notatios of asintotic funcions are defined in number theory, example:

The vinogradov notation (operator <<)

- Number theory is a branch of pure mathematics devoted primarily to the study of the integers.
- It is sometimes called "The Queen of Mathematics" because of its foundational place in the discipline.
- Number theorists study **prime numbers** as well as the properties of objects made out of integers (e.g., rational numbers) or defined as generalizations of the integers (e.g., algebraic integers).
- Criptography is nowadays based on prime numbers.

To remember

A recordar

Lo más importante es:

- La forma de las curvas asintóticas.
- Saber manipular algebraicamente las funciones O, etc.
- Calcular la complejidad de algoritmos ("típicos").

Recomendable:

- Poder demostrar algunas manipulaciones algebraicas.

The end

Contacto

Raúl Acosta Bermejo

http:www.cic.ipn.mx
http://www.ciseg.cic.ipn.mx/

racostab@ipn.mx
racosta@cic.ipn.mx

57-29-60-00
Ext. 56652

Homeworks

Tareas

Basic problems 1

Problemas básicos

Calcular la complejidad temporal O(n) en el Peor caso, mejor caso, y caso promedio, para una implementación de:

1. Listas: estática y dinámica (E&D).
2. Pilas: E&D, que cambia si se implementa solo con listas.
3. Colas: E&D, que cambia si se implementa solo con listas.
4. Arboles binarios.
5. Tablas hash

Tarea optativa

- Investigar y entregar resumen en no mas de 3 cuartillas.
- Dar la complejidad para las funciones principales que caracterizan el TAD (insertar, eliminar, leer).