75.41 Algoritmos y Programación II
Análisis de Algoritmos

Dr. Mariano Méndez [1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

30 de junio de 2020

1. Introducción

Un algoritmo es una secuencia de pasos no ambiguos, que al ser ejecutados resuelven un problema. Una vez que se se ha analizado el problema, diseñado el algoritmo, es necesario poder determinar cuantos **recursos computacionales** consume. Cuando se habla de recursos se refiere a **tiempo** o **espacio** que el mismo requiere para y en su ejecución [3] esto es necesario además para poder comparar los algoritmos entre ellos.

Es necesario poder predecir qué recursos de una computadora van a ser requeridos por un determinado algoritmo que se haya desarrollado. Estos recursos pueden ser:

- utilización memoria.
- utilización del ancho de banda en las comunicaciones.
- utilización del hardware de la computadora.
- el tiempo de cómputo o *tiempo computacional*

¿Para qué? esta es una muy buena pregunta y la respuesta es la siguiente: en general no hay una sola forma de resolver un problema, por ende haciendo este tipo de análisis se puede determinar cuál es el mejor algoritmo para ser utilizado en un determinado problema, cuando se tiene mas de una única feasible solución [1].

1.1. ¿Por Qué Se estudian los Algoritmos?

La más sencilla razón del por qué analizar un algoritmo es para descubrir sus características con el fin de evaluar su idoneidad para diferentes aplicaciones y para comparar lo con otros algoritmos para el mismo fin.Las características más importantes a ser tenidas en cuenta en esta comparación son el tiempo de ejecución y el espacio, la primera es la más importante.

En otros palabras, se quiere saber cuanto tardará una implementación de un determinado algoritmo que se ejecuta en una determinada computadora, y cuanto espacio este va a ocupar, o en otras palabras cuantos recursos para este escenario se utilizaran. Generalmente hay que esforzarse para que el análisis sea independiente de cualquier implementación en particular, es importante en concentrarse en las características esenciales del algoritmo que puedan utilizarse para derivar estimaciones precisas de los verdaderos requerimientos de los recursos en diversas máquinas [2].

Por ejemplo, existen otras propiedades más allá del tiempo o el espacio qué son de interés para ser estudiados. Por ende, el foco del análisis tiene que virar de acuerdo a esas propiedades, un algoritmo construido para un dispositivo móvil debería estudiar el efecto del mismo sobre la duración de la batería. También, por otro lado, un algoritmo para resolver un problema numérico tiene que ser estudiado para determinar cuán preciso es el resultado numérico que este arroja.

El término análisis de algoritmos ha sido utilizado para describir los diferentes enfoques para encasillar el estudio de la performance (eficiencia) de los programas de computadoras sobre bases científicas. El primer enfoque, fue popularizado por Aho, Hopcroft, Ullman, Cormen,Leiserson, Rivest, y Stein entre otros. Se concentra en determinar el crecimiento del peor de todos los casos en lo que respecta a la eficiencia de un algoritmo, lo que se denomina el upper bound. Se utilizará el término teoría de algoritmos para referenciar a este tipo de análisis. Es un caso especial de la complejidad computacional, qué es un caso de estudio general de las relaciones entre problemas, algoritmos, lenguajes y máquinas.

El segundo enfoque, se tiende hacia el análisis de algoritmos, que fue popularizado por Donal Knuth, y se concentra en precisas categorizaciones del mejor, el peor y promedio en lo que respecta a la eficiencia de los algoritmos.

En la práctica, lograr la independencia entre un algoritmo y su implementación puede ser una tarea difícil de lograr. Ya que, existen varios factores que están influyendo:

- la calidad de la implementación.
- las propiedades del compilador.
- la arquitectura de la máquina.

2. Un Poco de Matemática

En realidad el análisis necesario para realizar una estimación de los recursos consumidos por un algoritmo es un tema más teórico que práctico. Por ello es necesario establecer un marco formal de trabajo. Para ello se definirán algunas herramientas, en este caso la notación Bachmann-Landau o notación asintótica. En este caso se definen como funciones cuyo dominio es el conjunto de los números naturales $\mathbb{N} = 1, 2, 3, ...$ Esta notación es comúnmente conocida como notación de **O grande (big O notation)** $\mathbf{O}()$.

La notación O-grande, es una notación matemática que describe el comportamiento de una función en el límite, es decir, cuando el argumento tiende hacia un valor particular o infinito.

La notación O-grande caracteriza a las funcione según su tasa de crecimiento, diferentes funciones con la misma tasa de crecimiento serán representadas con la misma notación O-grande. La letra O se utiliza ya que la tasa de crecimiento de la función se refiere también como el **orden de la función**. *La descripción de una función en términos de notación O-grande únicamente provee un límite superior sobre la tasa de crecimiento de la misma*.

Si un tiempo de ejecución es $O(f(n))$, entonces para un n lo suficientemente grande, el tiempo de ejecución es como mucho $k.f(n)$ para alguna constante k dada. En el gráfico se ve cómo pensar acerca de un tiempo de ejecución que es $O(f(n))$: Se dice que el tiempo de ejecución es "O grande de $f(x)$" o simplemente "O de f(x)". Sea $T(x)$ el tiempo de ejecución en función del Tamaño del problema. Esta función puede calcularse sobre el código fuente, contando las instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción:

```
1 S1;
2 for (i=0;i<n;i++)
3         S2;
```

entonces el tiempo de ejecución $T(N)$ seria:

$$T(N) = t1 + t2 * n$$

siendo t1 el tiempo requerido para que se ejecute la instrucción 1 (S1, statement 1) y t2 el tiempo requerido para que se ejecute la instrucción 2 (S2 o statement 2). Casi todo los programas de la vida real incluyen en su código instrucción condicionales, esto hace que la cantidad efectiva de instrucciones ejecutadas dependa exclusivamente del conjunto de datos. por ende :

$$T_{min}(N) \leq T(N) \leq T_{max}(N)$$

donde: $T_{min}(N)$ es conocido como el mejor de los caso, tiempo mínimo de ejecución. Y $T_{max}(N)$ es conocido como el peor de los casos, tiempo máximo de ejecución.

2.1. Big-O

Se dice que la tasa de crecimiento de un algoritmo es **big-O**:

> $\mathbf{T(N) = O(f(N))}$ **si existe una constante positiva c y un** $\mathbf{n_0}$ **tal que** $\mathbf{T(N) \leq c * f(N)}$**, donde** $\mathbf{N \geq n_0}$**.**

Esto quiere decir que la tasa de crecimiento de $T(N)$ es menor o igual a $f(N)$.Está acotada por arriba. En la figura 1 puede verse claramente esto.

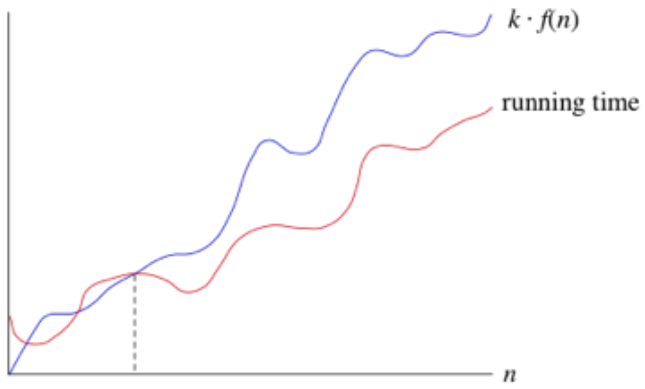

Figura 1

En otras palabras la función f pertenece a la clase de complegidad de g ($f \in O(g)$) si existe una constante positiva y un $n_0 \in \mathbb{N}$, tal que, Para todo $n > n_0$ donde $|f(n)| \leq |g(n)|$.

2.2. Big-Omega

Se dice que la tasa de crecimiento de un algoritmo es **big-Omega**:

$\mathbf{T(N) = \Omega(g(N))}$ **si existe una constante positiva c y un** $\mathbf{n_0}$ **tal que** $\mathbf{T(N) \geq cg(N)}$ **donde** $\mathbf{N \geq n_0}$**.**

Esto significa que la tasa de crecimiento de $T(N)$ es mayor o igual a $g(N)$. Está acotada por abajo. A veces se quiere decir que un algoritmo toma **por lo menos** una cierta cantidad de tiempo en ejecutarse, esto se ve el la figura 3. Esto quiere decir que big-Ω es el **límite inferior asintótico** del crecimiento del tiempo de ejecución.

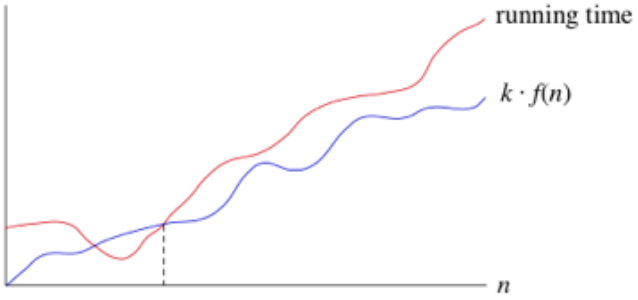

Figura 2: Notación asintótica: big-Ω

2.3. Small-o:

Se dice que la tasa de crecimiento de un algoritmo es **Small-o**:

$\mathbf{T(N) = o(p(N))}$ **si** $\mathbf{T(N) = o(p(N))}$ **y** $\mathbf{T(N)! = \Omega((p(N))}$**.**

Esto significa que la tasa de crecimiento de T(N) es estrictamente menor que la tasa de crecimiento de p(N)

2.4. Big-Theta:

Se dice que la tasa de crecimiento de un algoritmo es **Big-Theta**:

$\mathbf{T(N) = \Theta(h(N))}$ **si y solo si** $\mathbf{T(N) = O((h)N)}$ **y** $\mathbf{T(N) = \Omega((h(N))}$**.**

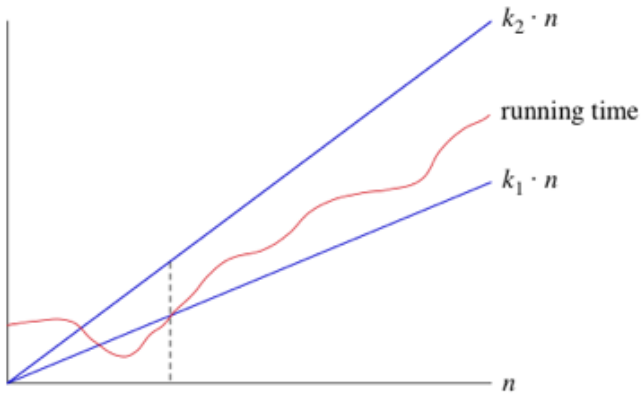

Figura 3: Notación asintótica: big-Θ

Para analizar esta notación asintótica véase el siguiente ejemplo, muy básico de una búsqueda lineal de un valor entero dentro de un arreglo de enteros, el mismo está delimitado por un tope:

```
int busqueda_lineal(int array[], int tope, int valor){

    for (var intento = 0; intento < tope; intento++) {
        if (array[intento] == valor) {
            return intento;     // encontrado! :-)
        }
    }
    return -1;                  // no lo encontre!!!  :-(
}
```

En este caso el tamaño del array es *tope* y se denota como *n*. El numero máximo de veces que la iteración puede ejecutarse es *n*, y justamente coincide con el peor de los casos, que es cuando el valor a ser hallado no se encuentra el el vector.

Cada vez que el for hace una iteración, suceden varias cosas:

- Se compara intento con tope
- se compara array[intento] con valor
- posiblemente se retorna de la función con el valor intento, que coincide con la posición en la que se encuentra el valor dentro del vector.
- se chequea, que coincida con la posición en la que se encuentra el valor dentro del vector.
- se incremente intento

cada una de estas instrucciones tarda una pequeñísima cantidad de tiempo en ser ejecutada, por ejemplo, esta instrucción, es decir, asignar a una variable un elemento de un determinado arreglo *i= array[intento]* puede tardar en una máquina moderna (intel Core i5-6330U) uno 0.34 nanosegundos.

Entonces si el for itera *n* veces, el tiempo para esa cantidad de iteraciones es $c_1 * n$, donde c es la suma de todos los tiempos de ejecución de las instrucciones ejecutadas. Algunas observaciones, no se puede decir con precisión cuál es el valor de $c_1 y aque$:

- depende de la velocidad del hardware
- del compilador
- del lenguaje de programación o interprete

entre otras cosas.

Pero hay que tener en cuenta que no sólo existen instrucciones iterativas, que pueden o no hacer que parte del código fuente se ejecute. En este caso cabe la posibilidad que el valor buscado no esté en el arreglo con lo cual la misma devolvería -1. Además, inicializar el valor de intento en 0 también necesita un tiempo para ser ejecutado. A estas instrucciones que se ejecutan o nos en un algoritmo, más allá del núcleo del mismo, se le asigna un valor c_2. Por lo tanto el tiempo total de ejecución para la búsqueda lineal es $c_1 * n + c_2$.

Teniendo en cuenta lo anterior, los valores c_1 y c_2, no aportan información de la tasa de crecimiento del tiempo de ejecución del algoritmo. Lo único que es significativo es n por ende en el peor de los casos la tasa de crecimiento en el tiempo de procesamiento es $\Theta(n)$

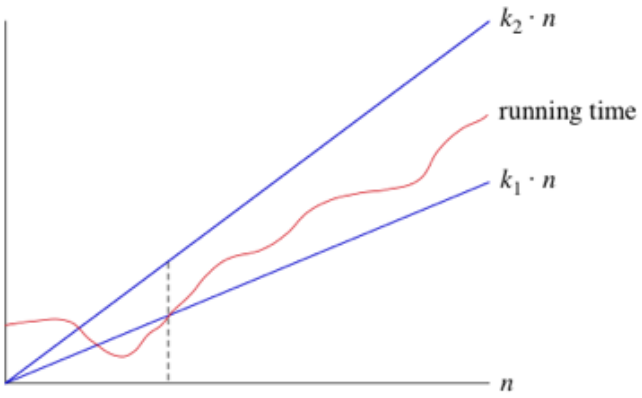

Figura 4: Notación asintótica: big-Θ

Entonces, decir que $T(N) = \Theta(g(n))$ implica que existan c_1, c_2 y n_0 tales que $0 <= c_1 * g(n) \leq T(n) \leq c_2 * g(n)$ para todo $n => n_0$. Hay que tener en cuenta que $\Theta(n)$ es un **conjunto** de funciones

3. Propiedades de la Notación Asintótica

$f = \Theta(g)$	f crece con la misma tasa que g	Existe un n_0 y dos contantes $c_1, c_2 > 0$ tales que, para todo $n > n_0, c_1 g(n) \leqslant \|f(n)\| \leqslant c_2 g(n)$.
$f = O(g)$	f no crece más rápido que g	Existe un n_0 y$c > 0$ tales que, para todo $n > n_0, \|f(n)\| \leqslant cg(n)$.
$f = \Omega(g)$	f crece a lo sumo como g	Existe un n_0 y$c > 0$ tales que, para todo $n > n_0, \|cg(n) \leqslant \|f(n)\|$.
$f = o(g)$	f crece más lentamente que g g	Existe un n_0 y$c > 0$ tales que, para todo $n > n_0, \|f(n)\| < cg(n)$.
$f \sim g$	f/g se acerca a 1	$\lim_{n\to\infty} f(n)/g(n) = 1$

Figura 5: Símbolos de la Notación Asintomática

Existen ciertas propiedades que son importantes tener en cuentas:

Regla 3.1 *Si*

$$T_1(n) = O(f(n)),\ y\ \ T_2(n) = O(g(n))$$

entonces:

a)

$$T_1(n) + T_2(n) = max(O(f(n)), O(g(n)))$$

b)

$$T_1(n) * T_2(n) = max(O(f(n) * g(n)))$$

Regla 3.2 *Si* $T(n)$ *s un polinomio de grado* k *, entonces* $T(n) = O(n^k)$.

Regla 3.3 $log^k n = O(n)$ *para cualquier constante* k. *Esto indica que el logaritmo tiene una tasa asintótica de crecimiento muy lenta.*

Si $T(n)$ s un polinomio de grado *k* , entonces $T(n) = \Theta(n^k)$.

Cuando se trabaja con notación asintótica, siempre se desprecian las constantes y las funciones de menor orden. Por ejemplo, si $T(N) = O(2n^2)$ o $T(N) = O(n^2 + n)$, en ambos casos e dice que

Por otro lado **siempre** es posible determinar la tasa de crecimiento relativa entre dos funciones, $f(N)$ y $g(N)$, aplicando la regla de L'Hopital:

Sean f y g dos funciones continuas definidas en el intervalo [a,b], derivables en (a,b) y sea c perteneciente a (a,b) tal que $f(c) = g(c) = 0$ y g'(x)≠0 si x≠c. Si existe el límite L de f'/g' en c, entonces existe el límite de f/g (en c) y es igual a L. Por lo tanto:

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)} = L$$

Entonces si:

- Si el límite es 0: implica que $f(N) = o(g(N))$.
- Si el límite es $c \neq 0$: esto significa que $f(N) = \Theta(g(N))$.
- Si el límite es infinito: esto significa que $o(N) = o(f(N))$.
- Si $f = o(g) \implies f = O(g)$
- Si $f = w(g) \implies f = \omega(g)$
- Si $f \sim w(g) \implies f = \theta(g)$

3.1. Valores

Para tener una idea de cuales son los límites asintóticos más comunes, son:

Función	Nombre
c	Constante
$logn$	Logarítmico
$log_2 n$	Logaritmo base 2
n	Lineal
$nlogn$	Casi Logaritmo
n^2	Cuadrática
n^3	Cubica
2^n	Exponencial
$loglosn)$	Log logarítmica
$n!$	Factorial

Figura 6: Valores más comunes

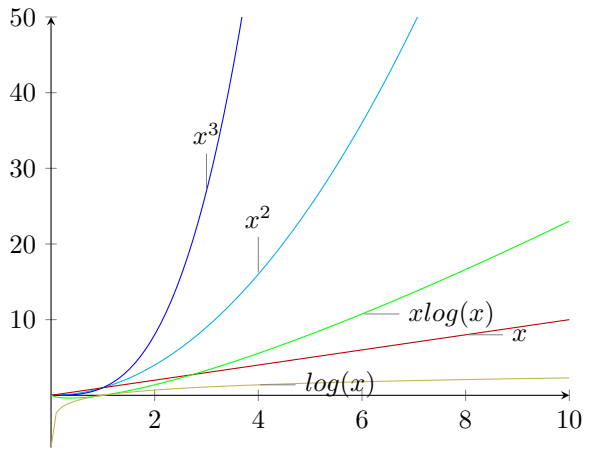

Figura 7: Comparativas de las funciones más comunes

3.1.1. Jerarquías de Complejidades

$$O(1) \subset O(logn) \subset O(n) \subset O(nlogn) \subset O(n^2) \subset O(^3) \subset O(2^n)$$

3.1.2. Efecto de Duplicaciones

- Efecto de duplicación en la cantidad de elementos

T(n)	n=100	n=200
$log\ n$	1h	1.15h
n	1h	2h
$n\ log\ n$	1h	2.3h
n^2	1h	4h
n^3	1h	8h
2^n	1h	$1{,}23 * 10^{30}$h

- Efecto de duplicación en la cantidad de tiempo

T(n)	1h	2h
$log\ n$	n=100	n=10000
n	n=100	n=200
$n\ log\ n$	n=100	n=178
n^2	n=100	n=141
n^3	n=100	n=126
2^n	n=100	n=101

4. En Otras Palabras..

4.1. $O(1)$

- $O(1) \rightarrow$ /?oh won?/
- **Complejidad Constante**
- No importa lo que se provea como input al algoritmo, este se ejecutará siempre en la misma cantidad de tiempo:
  1. 1 elemento, 1 segundo
  2. 10 elementos, 1 segundo
  3. 100 elementos, 1 segundo

4.2. $O(log\ n)$

- $O(log\ n) \rightarrow$ /?oh log en?/
- **Complejidad Logarítmica**
- El tiempo de cálculo se incrementa lentamente según la cantidad de datos de input se incrementa exponencialmente:
  1. 1 elemento, 1 segundo
  2. 10 elementos, 2 segundos
  3. 100 elementos, 3 segundos
- Ejemplo: La Búsqueda Binaria

4.3. $O(n)$

- $O(n) \rightarrow$ /?oh en?/
- **Complejidad Lineal**
- El tiempo de cálculo se incrementa al mismo ritmo de cantidad de datos de input :
  1. 1 elemento, 1 segundo
  2. 10 elementos, 10 segundos
  3. 100 elementos, 100 segundos
- Ejemplo: La Búsqueda lineal

4.4. $O(n^2)$

- $O(n^2) \rightarrow$ /?oh en squared?/
- **Complejidad Cuadrática**
- El tiempo de cálculo se incrementa al mismo ritmo de n^2 de datos de input :
  1. 1 elemento, 1 segundo
  2. 10 elementos, 100 segundos
  3. 100 elementos, 10000 segundos
- Ejemplo: El Burbujeo

4.5. $O(n!)$

- $O(n!) \rightarrow$ /oh en factorial/
- **Complejidad Factorial**
- El tiempo de cálculo se incrementa al mismo ritmo de $n!$ donde n es la cantidad de datos de input. Cuando n es pequeño no hay problemas, n=5 $n!$=120, pero rápidamente crece a valores imposibles :
  1. 1 elemento, 1 segundo
  2. 10 elementos, 3628800 segundos
  3. 100 elementos, $9{,}332621544 \times 10157$ segundos
- Ejemplo: El Problema del Viajante
- $n!$ es un mal lugar en el mundo de los algoritmos. Básicamente significa que no se puede resolver.

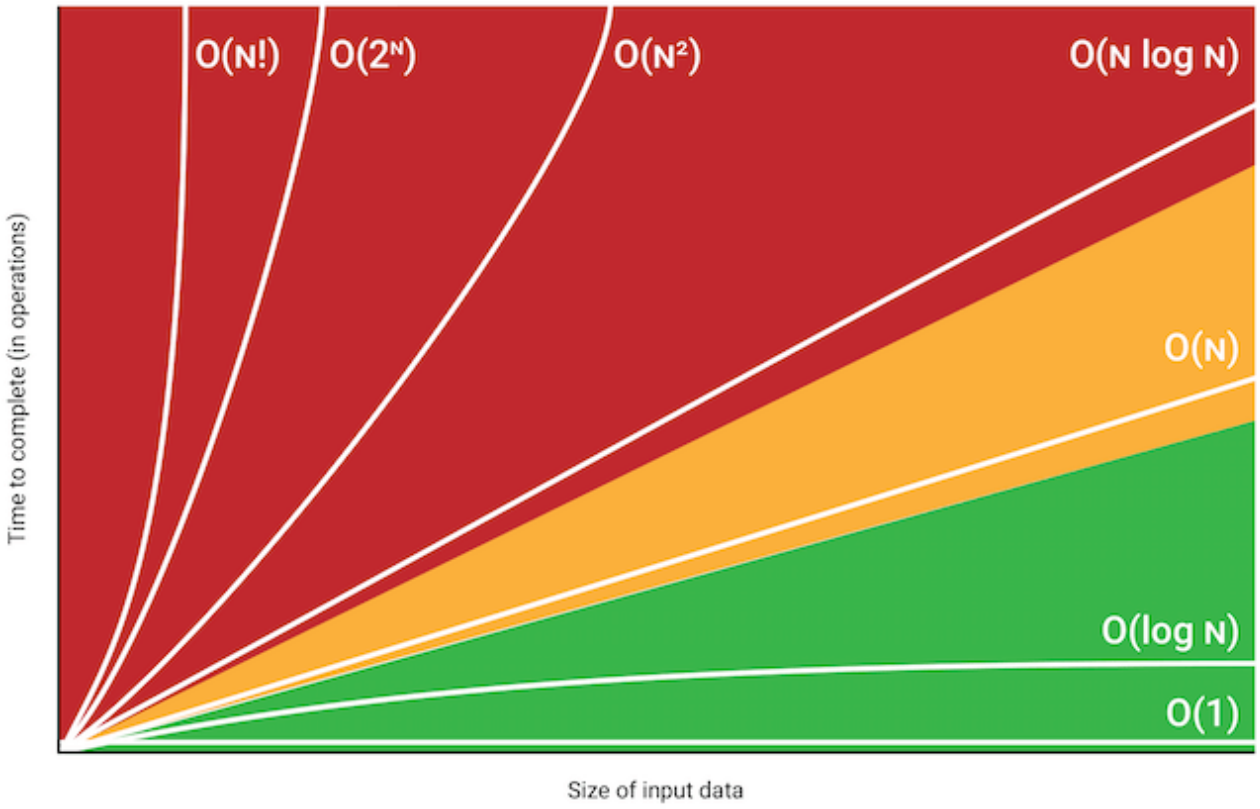

Figura 8: Notación asintótica: Comparativa

5. Ejercicios

5.1. Ejercicio 1

Demostrar que $4x_3 + 7x_2 + 12$ es $O(x_3)$ encontrar c y k.

5.2. Ejercicio 2

Demostrar que $f(x) = x + 7$ es $O(x)$ encontrar c y k.

6. El Modelo

para poder analizar un algoritmo en un marco de trabajo formal (formal framework) es necesario establecer un modelo computacional, es necesario establecer el modelo computacional. En este caso el modelo computacional seleccionado es:

- el de un estándar de simples instrucciones que se ejecutan una después de otra,
- pero para simplificar los cálculos en este modelos las instrucciones todas tardan la misma unidad de tiempo en ejecutarse (cosa que en la realidad no sucede).
- Este modelo toma números enteros de tamaño fijo de 32 bits, con memoria infinita.

6.1. ¿Qué Medir?

El recurso más importante que se mide normalmente es el tiempo. Existen muchos factores que alteran el tiempo de ejecución de un programa, pero uno de los más interesantes son:

- El Algoritmo
- El set de datos de entrada.

Normalmente el segundo siempre es el más considerado.

Definición 6.1 *Tiempo promedio T_{avg} es la utilización del tiempo utilizado por el algoritmo, promediado en todos los posibles conjuntos de entrada de datos*

Definición 6.2 *Peor Tiempo T_{worst} es aquel que que provee el límite superior de tiempo que un algoritmo puede tardar*

7. Calcular el Tiempo

Para poder calcular el tiempo exacto en el modelo computacional dado se deben determinar todas las instrucciones que han sido ejecutadas y multiplicarlas por el tiempo de ejecución de una instrucción, para ello no queda otra solución que analizar el algoritmo contabilizando las instrucciones. Para ello ademas hay que determinar cuales son instrucciones ejecutables y cuales no. Por suerte existen ciertas reglas que ayudan a realizar la contabilidad de aquellas instrucciones o conjunto de instrucciones que tienen peso en el consumo total del tiempo de ejecución.

7.1. Reglas

Regla 7.1 (Iteraciones) *El tiempo de ejecución de una iteración es a lo sumo el tiempo de ejecución de las instrucciones dentro de la iteración por el número de iteraciones.*

Regla 7.2 (Iteraciones Anidadas) *Se analizan de adentro hacia afuera. El tiempo total de una instrucción dentro dentro de un grupo de iteraciones anidadas es el tiempo de ejecución de la instrucción multiplicado por la cantidad de iteraciones de cada iteración.*

```
for (i =0; i<n; i++)
        for (j =0; j<n; j++)
                k++;
```

Regla 7.3 (Instrucciones Consecutivas) *El tiempo de ejecución de instrucciones consecutivas es el máximo de entre ambas instrucciones*

en el ejemplo se puede ver que un fragmento de código es $O(N)$ y el otro $O(N^2)$

```
for (i =0; i<n; i++)
        w++;
for (i =0; i<n; i++)
        for (j =0; j<n; j++)
                k++;
```

Regla 7.4 (Condicionales) *Para las estructuras*

```
if (Condicion)
        S1
else
        S2
```

El tiempo de ejecución de una instrucción condicional nunca será mayor al tiempo de ejecución de la condición más el más largo de los tiempos de ejecución entre S1 o S2.

7.1.1. Burbujeo

A continuación se analizará el algoritmo de ordenamiento por burbujeo:

```
int array[N];

/*1*/ for (i = 0; i <  N  ; i++) {
/*2*/     for (j = 0; j < N - i ; j++) {
/*3*/         if (array[j] > array[j+1]) /* For descending order use < */
              {
/*4*/           swap       = array[j];
/*5*/           array[j]   = array[j+1];
/*6*/           array[j+1] = swap;
              }
          }
      }
```

Utilizando las reglas descriptas anteriormente, cuando se tiene loops anidados se va desde adentro haca afuera del loop. Por ende primero se analizan las líneas 4,5,6 todas las líneas son O(1) por ende aplicando la regla de las instrucciones consecutivas es el el tiempo que se toma es el máximo para las 3 $O(max(1,1,1)) = O(1)$. Ahora el siguiente paso es determinar el tiempo del if/else, para ello se aplica la regla correspondiente, en este caso es $O(1)$. Siguiendo hacia afuera se tiene el for más interno que realiza $(n-i)$ iteraciones, por lo tanto ese ciclo for es $O(n-i)$. Ahora bien en este punto queda el ciclo for más externo que se ejecuta n-1 veces. Por ende la cantidad total de instrucciones ejecutadas es:

$$\sum_{i=1}^{n-1}(n-i) = n*(n-1)/2 = n^2/2 - n/2$$

Otra forma de realizar el análisis anterior es buscar cuantas veces se ejecuta la comparación (array[d] ¿array[d+1]) ya que este es el punto neurálgico del algoritmo. El primer ciclo for indica la cantidad de pasadas que e realizan, en el peor de los casos N pasadas. Ahora cuantas comparaciones se realizan en cada pasada:

Pasada	Comparaciones
1	n-1
2	n-2
3	n-3
...	...
...	...
n-3	3
n-2	2
n-1	1
n	0

Figura 9: Valores más comunes

Una vez que se sabe la cantidad de pasadas lo único que hace falta es sumarlas: $1+2+3+...+n-3+n-2+n-1 = (n-1)(1+n-1)/2 = (n^2-n)/2$

Teniendo en cuenta la regla numero 1, el tiempo de ejecución del burbujeo en el peor de los casos $=O(n^2)$

7.1.2. Búsqueda Binaria

Este algoritmo de búsqueda es utilizado cuando se está trabajando con un conjunto de datos ya ordenado. La búsqueda binaria se dice que es más eficiente que la búsqueda lineal. Para realizar esa afirmación se deberá hacer el análisis del algoritmo (fuente Weiss):

```
      int BinarySearch( const ElementType A[ ], ElementType X, int N ){
           int Low, Mid, High;

/* 1*/       Low = 0; High = N - 1;
```

```
5  /* 2*/       while( Low <= High )
6               {
7  /* 3*/           Mid = ( Low + High ) / 2;
8  /* 4*/           if( A[ Mid ] < X )
9  /* 5*/               Low = Mid + 1;
10                  else
11 /* 6*/           if( A[ Mid ] > X )
12 /* 7*/               High = Mid - 1;
13                  else
14 /* 8*/               return Mid;  /* Found */
15              }
16 /* 9*/       return NotFound;     /* NotFound is defined as -1 */
17          }
```

La función son tres instrucciones, una de las cuales contiene un bloque. Por ende, se aplica la regla que dice que el tiempo de ejecución es el máximo entre las tres instrucciones. En este caso la línea 1 es $O(1)$ y la línea 8 es $O(1)$. Por lo cual, el punto a analizar es la instrucción de la línea 2, que es un ciclo while, que a su vez contiene una secuencia de instrucciones, y todas son $O(1)$. Lo importante del análisis es determinar cuantas veces el ciclo itera. En cada iteración se va descartando $n/2$ elementos, hasta llegar a un arreglo de 1 elemento, eso se traduce en la siguiente ecuación:

$$1 = N/2^x$$

despejando :

$$2^x = N$$

Si se aplica logaritmo en base 2 en ambos lados:

$$log(2^x) = log(N)$$

Por propiedad de los logaritmos :

$$log(2) * x = log(N)$$

$$x = log(N)$$

Donde c es la cantidad de iteraciones del ciclo, $\mathbf{T(n) = O(log(N))}$

7.1.3. Búsqueda lineal

El algoritmo de búsqueda lineal es aplicable a cualquier conjunto de datos, el mismo no debe estar ordenado, y permite ubicar si un determinado elemento se encuentra dentro de ese conjunto.

```
1  int linear_serch( int array[], int elem, int n){
2          int i;
3
4  /*1*/    i=0;
5  /*2*/    while (i<n) && (array[i]!=elem)
6  /*3*/            i++;
7  /*4*/    if (i<n) return i
8  /*5*/    else return -1;
9  }
```

Para realizar el análisis del mismo se aplican las reglas, el algoritmo es una secuencia de instrucciones, por ende el orden del mismo es el máximo orden entre los ordenes de las instrucciones. Las instrucciones de las líneas 1,4 y 5 tiene un orden de crecimiento $(O(1)$, las líneas 2 y 3 son una única instrucción while que realiza 5 operaciones. Por ello en una iteración del ciclo se realizan 5 operaciones, 10 en dos y así sucesivamente, en n iteraciones $5n$ operaciones. Aquí podría tomarse dos caminos para determinar el orden ambos confluyen al mismo resultado.

1. El primero es tener en cuenta las reglas y determinar que el tiempo de ejecución del algotimo es $T(n) = O(n)$.
2. El segundo es totalizar las operaciones las de la iteración más las demás, entonces $T(n) = O(5n) + O(4) = O(5n) = O(n)$

7.1.4. Otro Ejemplo no Tan Trivial

Calcular el tiempo de ejecución del siguiente algoritmo en el peor de los casos:

```
void foo( int n){
     int i,k;

     i=1;
     k=1;
     while (i<n){
          if (i==k){
               printf("\nk=%i i=",k);
               k++;
               i=1;
          }
          printf("%i",i);
          i++;
     }
     printf("\n");
 }

```

7.2. Análisis Asintótico para Algoritmos Recursivos

No es fácil realizar el análisis asintótico de algoritmos recursivos para calcular su complejidad y por ende hacer una aproximación lo bastante buena de su tiempo de ejecución o del consumo de recursos que este realice.

$$n! = (n-1)!n, 1! = 0! = 1$$

```
long factorial (int n){
    if (n == 0 )
        return 1;
    else
        factorial= n * factorial(n-1);
}
```

Al intentar realizar el análisis del tiempo de ejecución de este algoritmo recursivo, puede verse que el tiempo de ejecución del algoritmo es el tiempo de ejecución del acceso a n, más el tiempo de ejecución de la función factorial para $n-1$ que justamente no sa sabe porque se quiere calcular.

Por tanto, para conocer el tiempo de ejecución de este algoritmo ¿debemos conocer previamente el tiempo de ejecución de este algoritmo?, entramos en una recurrencia.

Entonces, cuando $n = 0$ el costo del tiempo de ejecución de la función es el costo de return 1, por ende es un valor constante, por ejemplo c. Ahora para un n el tiempo de ejecución $T(n)$, es el tiempo de ejecución $T(n-1)$ más un valor constante c :

$$T(n) = \begin{cases} c & si\ n = 0 \\ \\ T(n-1) + c & si\ n > 1 \end{cases}$$

Para saber cual es el tiempo de ejecución de esta función recursiva basta con solucionar la ecuación de recurrencia definida (matemática discreta!). Para intentar solucionar esta relación, usemos sustitución:

$$T(n) = T(n-1) + c$$

tener en cuenta que $T(n-1) = T(n-2) + c$

$$\begin{aligned} &= T(n-2) + 2c \\ &= T(n-3) + 3c \\ &= T(n-4) + 4c \\ &\dots \\ &= T(4) + (n-4)c \\ &= T(3) + (n-3)c \\ &= T(2) + (n-3)c \\ &= T(1) + (n-3)c \\ &= T(0) + (n-1)c = c + (n-1)c = nc \end{aligned}$$

Por ende $T(n) = n = O(n)$ Para el algoritmo recursivo que obtiene el factorial de n.

Alguien se anima con los números de Fibonacci?

```
long int fibonacci(numero){
        if (numero < 2)
                return numero;
        return fibonacci(numero-1)+fibonacci(numero-2);
}
```

7.3. Relaciones de Recurrencia

Una relación de recurrencia define una función mediante una expresión que incluye una o más instancias (más pequeñas) de si misma. Por ejemplo:

$$n! = (n-1)!n, 1! = 0! = 1$$

Escrito de otra forma :

$$n! = \begin{cases} 1 & si\ n = 0, 1 \\ n(n-1)! & si\ n > 1 \end{cases}$$

8. El Teorema Maestro

Este es el "libro de cocina" para resolver recurrencias de la forma:

$$T(n) = aT(n/b) + f(n)$$

Donde las constantes $\mathbf{a >= 1}$ y $\mathbf{b > 1}$ y por otro lado $\mathbf{f(n)}$ es una función asintóticamente positiva. Hay que considerar:

- Este tipo de recurrencias es importante porque describe los tiempos de ejecución de algoritmos que dividen un problema de tamaño n en subproblemas de tamaño $\mathbf{a}$, cada uno de ellos de $\mathbf{n/b}$.
- Cada subproblema es resuelto en un tiempo $\mathbf{T(n/b)}$
- La función $\mathbf{f(n)}$) abarca el costo de dividir el problema y combinarlo.

En otras palabras:

- a equivale al tamaño del problema.
- a equivale a la cantidad de llamadas recursivas que realiza el algoritmo.
- b equivale a cuanto se divide el problema para resolverlo recursivamente.
- $f(n)$ es el costo en tiempo de lo que cuesta dividir y combinar.

Teorema 1 (Teorema Maestro) *Sean $a >= 1$ y $b > 1$ dos constantes, sea $f(n)$ una función asintóticamente positiva y sea $T(n)$ una relación de recurrencia de enteros no negativos:*

$$T(n) = aT(n/b) + f(n)$$

, entonces $T(n)$ posee las siguientes cotas asintóticas:

1. **Caso 1:** *Si $f(n) = O(n^{\log_b a - e})$, para alguna constante $e > 0$, entonces $T(n) = \Theta(n^{\log_b a})$*
2. **Caso 2:** *Si $f(n) = O(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a}\ lg\ n)$*
3. **Caso 3:** *Si $f(n) = \Omega(n^{\log_b a + e})$, para alguna constante $e > 0$ y si $af(n/b) <= cf(n)$ para una constante $c < 1$ y el valor de n lo suficientemente grande, entonces $T(n) = \Theta(f(n))$*

8.1. Explicación

En cada caso se está comparando la función $f(n)$ con la función $n^{log_b a}$, puede prestar confusión que en la comparación se utiliza a su vez notación asintótica. Pero si si ven como dos funciones, intuitivamente hace pensar que la mayor de las dos funciones será la que domine el tiempo de ejecución. Para poder entender esto es importante comprender como funciona el árbol de recurrencia de una función de la forma $T(n) = aT(n/b)$:

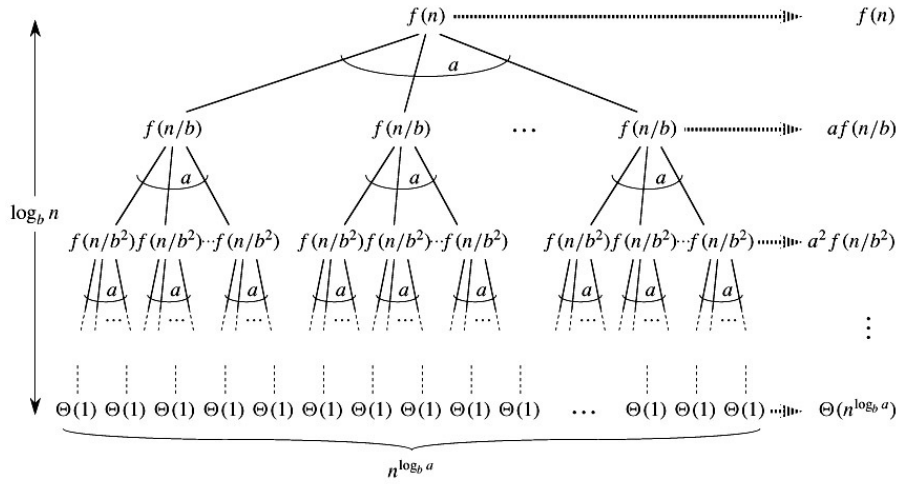

Figura 10: El árbol de recurrencia generado por la funcion $T(n) = aT(n/b) + f(n)$. Este árbol posee una $n^{log_b a}$

Esto significa que $n^{log_b a}$ es la cantidad de hojas o divisiones que se generan al final de todas las llamadas recursivas. Si se tiene en cuenta una ecuación recurrente del estilo $T(n) = aT(n/b) + f(n)$, tiene dos componentes que aportan costo computacional, entonces se pueden encontrar tres casos:

- Si $f(n)$ **es polinomicamente más pequeña** que $n^{\log_b a}$ el tiempo de ejecución que prevalece es el tiempo de división recursiva $T(n) = \Theta(n^{\log_b a})$.
- Si por el contrario $f(n)$ **es polinomicamente dominante sobre** $n^{\log_b a}$ entonces el tiempo de ejecución que prevalece es el de $f(n)$, entonces $T(n) = \Theta(f(n))$.
- Si finalmente $f(n)$ y $n^{\log_b a}$ **son asintóticamente iguales** entonces el tiempo de ejecución es $T(n) = \Theta(n^{\log_b a}\ lg\ n)$.

Existe una forma muy simple de verlo, teniendo en cuenta a $T(n) = aT(n/b) + f(n)$:

$$\begin{cases} f(n)\ es\ menor\ que\ n^{\log_b a} entonces & T(n) = \Theta(n^{\log_b a}). \\ f(n)\ es\ igual\ que\ n^{\log_b a}\ entonces & T(n) = \Theta(n^{\log_b a}\ lg\ n) \\ f(n)\ es\ polinomicamente\ mayor\ que\ n^{\log_b a},\ entonces & T(n) = \Theta(f(n)). \end{cases}$$

Existe otra forma de plantearlo que es:

1. $f(n)$ es $O(n^c)$ y $c < \log_b a$, entonces $T(n) = \Theta(n^{\log_b a})$
2. Si es verdad que para alguna constante $k?0$, que: $f(n)$ es igual que $\Theta(n^c lg^k n)$

   Si $f(n) = O(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a}\ lg\ n)$
3. $f(n)$ es polinomicamente mayor que $n^{\log_b a}$

   Si $f(n) = \Omega(n^{\log_b a + e})$, para alguna constante $e > 0$ y si $af(n/b) <= cf(n)$ para una constante $c < 1$ y el valor de n lo suficientemente grande, entonces $T(n) = \Theta(f(n))$

8.1.1. Ejemplos

Ejemplo 1 Por ejemplo, sea el tiempo de ejecución de un algoritmo dado:

$$T(n) = 9T(n/3) + n$$

la forma de resolver este tipo de ejercicio es ver si aplica el teorema maestro. Para ello identificamos los valores de a, b y $f(n)$. Entonces a=9, b=3 y f(n)=n. Ahora se calcula $n^{\log_b a}$, con estos valores de a y b $n^{\log_3 9}$, lo que arroja que $n^{\log_b a} = n^2$ por eso $f(n) < n^{\log_b a}$, entonces se está en el primer caso del teorema maestro, $T(n) = \Theta(n^2)$

Ejercicio 1

1. Calcular utilizando el **Teorema Maestro** el tiempo de ejecución de la Búsqueda Binaria recursiva.

Solución del ejercicio 1

Análisis: la búsqueda binaria recursiva se describe a continuación:

```
int busqueda_binaria(vector_t vector, int inicio, int fin, elemento_t elemento_buscado) {
    int centro;

    if(inicio<=fin){
        centro=(fin+inicio)/2;
        if(vector[centro]==elemento_buscado)
            return centro;
        else if(elementoBuscado < vector[centro])
            return busqueda_binaria(vector,inicio,centro-1,elemento_buscado);
        else
            return busqueda_binaria(vector,centro+1,fin,elemento_buscado);
    } else
        return -1;
}

```

En primer lugar, para poder aplicar el teorema maestro se debe tener una ecuación recurrente del tipo $T(n) = aT(n/b) + f(n)$ donde $f(n) = cn^k$, para ello se analiza el algoritmo teniendo en cuenta los valores de a y n/b:

$$T(n) = aT(n/b) + f(n)$$

donde a es la cantidad de llamadas recursivas dentro del algotirmo; y b es la cantidad de elemento en que se dividen los subproblemas:

$$a = 1$$

ya que en cada paso sólo se ejecuta 1 única llamada recursiva. Respecto del parámetro b, en cada llamada el conjunto de elementos de cada sub-problema es de la mitad que el anterior, por ende:

$$b = 2$$

Por último la función $f(n)$ es un valor constante, ya que según el modelo que utilizamos para análisis de algoritmos supone que todas las operaciones son constantes; entonces $f(n) = 1$. De esta forma la ecuación de recurrencia para la búsqueda binaria es:

$$T(n) = aT(n/b) + f(n)$$

$$T(n) = (n/2) + 1$$

Ya a partir de la ecuación de recurrencia se puede comenzar a aplicar el teorema maestro:

$$\begin{cases} a = 1 \\ b = 2 \\ f(n) = 1 \implies n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \end{cases}$$

caso 2 del teorema maestro, con lo cual:

$$T(n) = \Theta(n^{\log_b a}\ lg\ n)$$

sustituyendo a,b donde corresponde:

$$T(n) = \Theta(n^{\log_2 1}\ lg\ n) = \mathbf{lg\ n}$$

Ejemplo 3 Sea $T(n) = 4T(n/2) + 3n^2$ calcular el orden del tiempo de ejecución.

Ejemplo 4 Sea $T(n) = T(2n/3) + 1$ calcular el orden del tiempo de ejecución.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Pearson Education India, 2013.

[3] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 199.