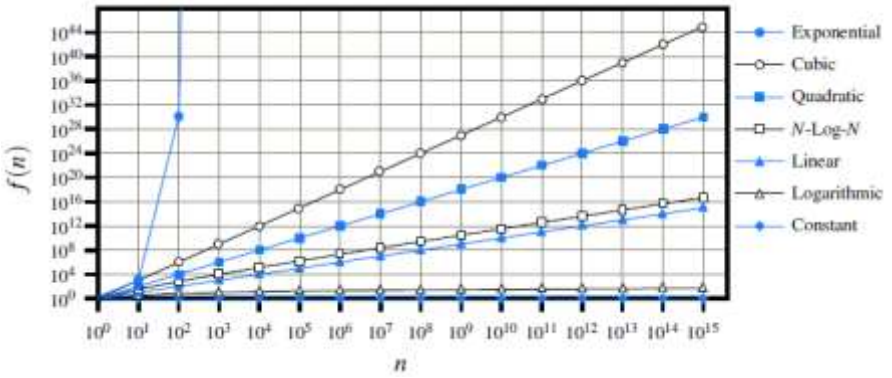
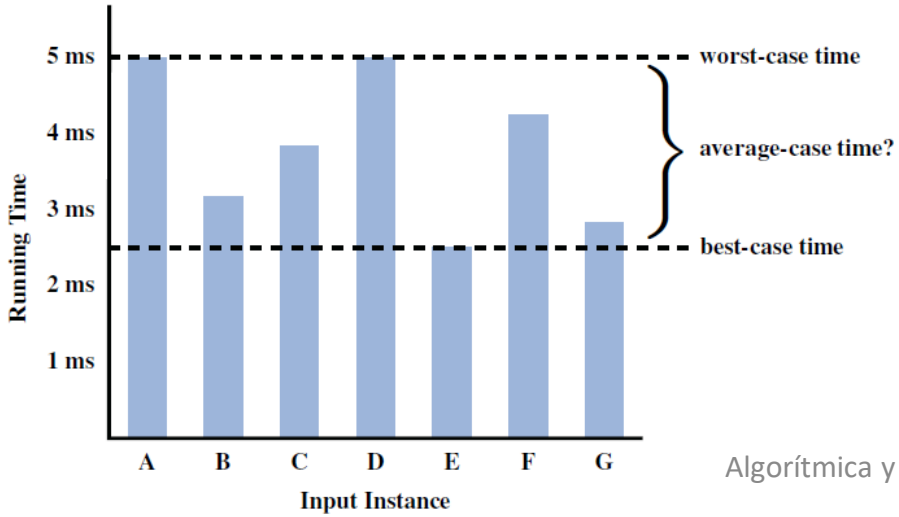


Análisis de Algorítmos

Unidad 4



Lic. Renato Mazzanti

Estudios experimentales

- Una manera de estudiar la eficiencia de un algoritmo es implementarlo y experimentar ejecutando el programa en varias entradas de prueba mientras se registra el tiempo gastado durante cada ejecución.
- Para recopilar estos tiempos de ejecución en Java se basa en el uso del método *currentTimeMillis* de la clase *System.*
- Ese método informa el número de milisegundos que han pasado desde un tiempo de referencia conocido como el *período* (1 de enero de 1970 UTC).

Estudios experimentales

- Podemos medir el tiempo transcurrido (*elapsed time*) de la ejecución de un algoritmo. ¿cómo?

Estudios experimentales

- Podemos medir el tiempo transcurrido (*elapsed time*) de la ejecución de un algoritmo. ¿cómo?

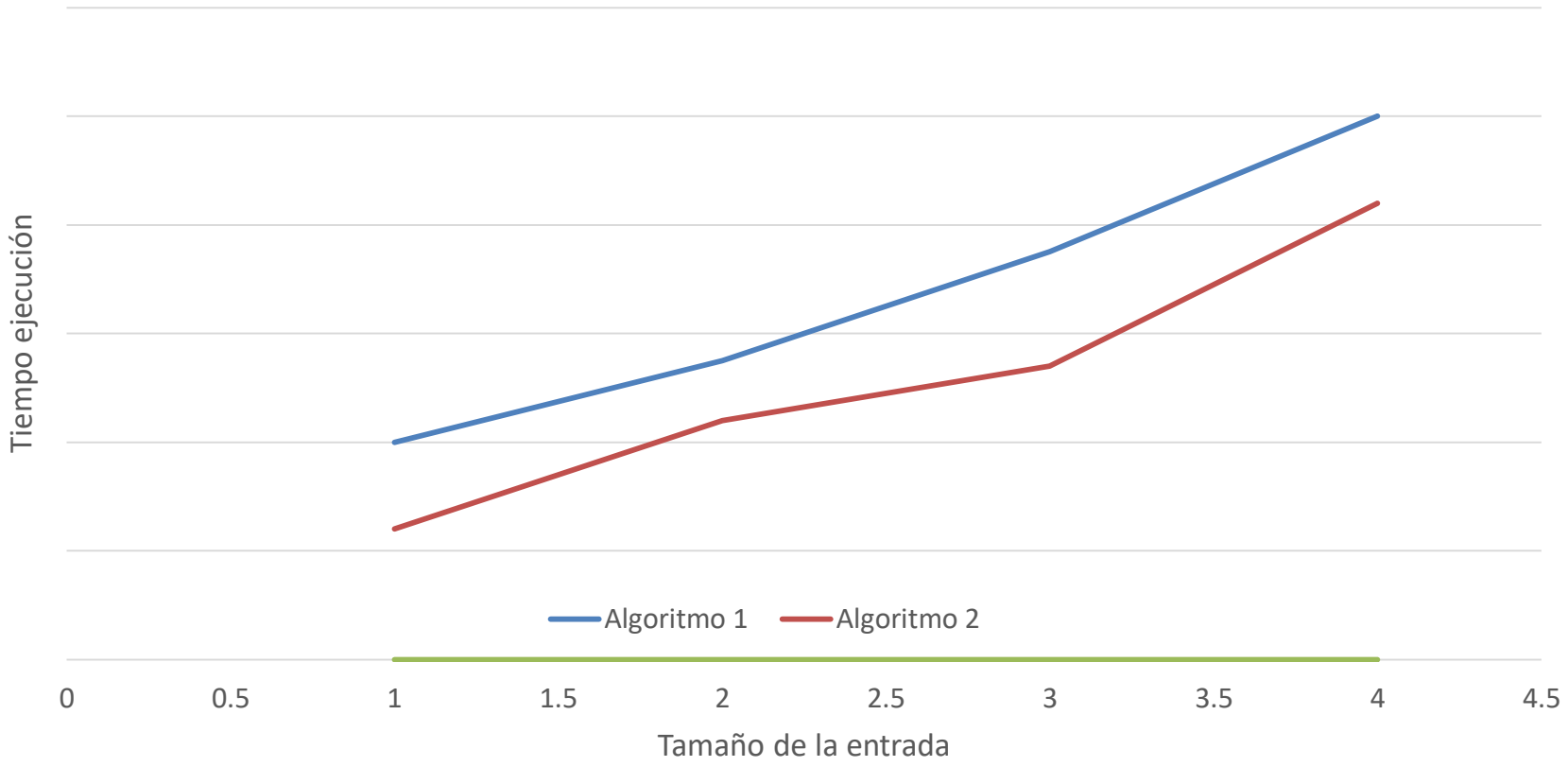
```
1  long startTime = System.currentTimeMillis();      // record the starting time
2  /* (run the algorithm) */
3  long endTime = System.currentTimeMillis();        // record the ending time
4  long elapsed = endTime − startTime;               // compute the elapsed time
```

- Para operaciones extremadamente rápidas, Java proporciona un método, *nanoTime*, que mide en nanoseconds

Estudios experimentales

- ¿Por qué estamos interesados en la dependencia general del tiempo de ejecución sobre el tamaño y la estructura de la entrada?
- Podemos graficar cada ejecución del algoritmo como un punto con coordenada *x* igual al tamaño de la entrada, *n*, y coordenada *y* igual al tiempo de ejecución, *t*. Es decir (x,y) == (n,t)
- Relación entre el **tamaño del problema** y el **tiempo de ejecución** del algoritmo.

Estudios experimentales



Estudios experimentales

- Esto puede ir seguido por un análisis estadístico que busca ajustar la mejor función del tamaño de entrada a los datos experimentales.
- Este análisis requiere que escojamos buenos datos de muestra y probemos bastantes de ellos para ser capaces de hacer afirmaciones estadísticas sobre el tiempo de ejecución del algoritmo.
- ¿Tiene algún efecto la plataforma donde se ejecuta el algorítmo sobre los resultados obtenidos de este modo?

Estudios experimentales

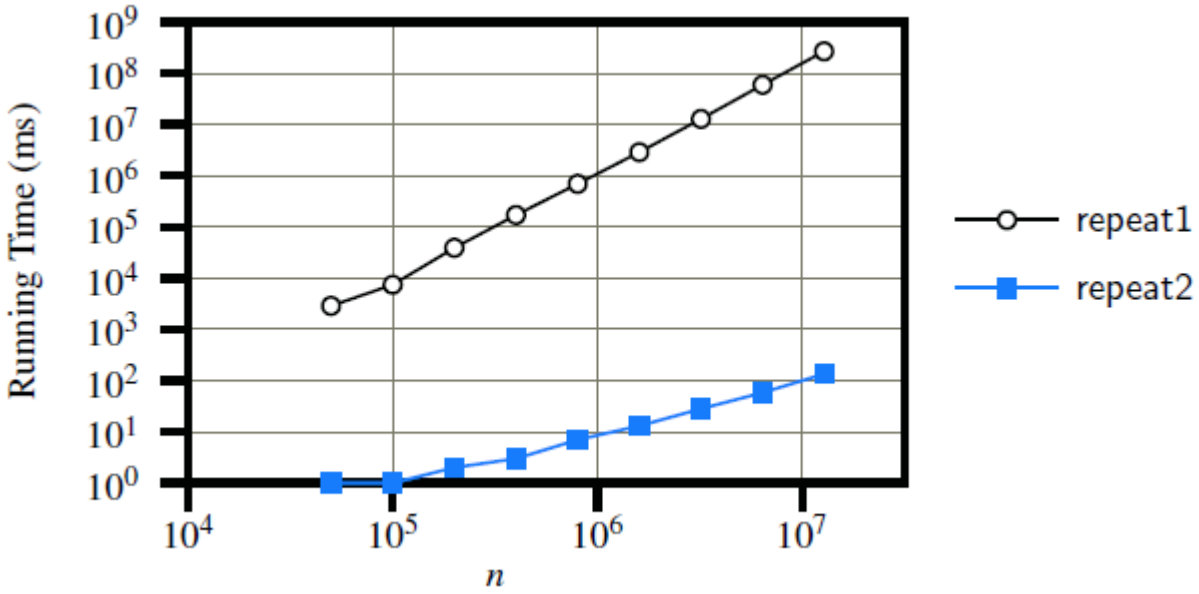
repeat('*', 40)

```
1   /** Uses repeated concatenation to compose a String with n copies of character c. */
2   public static String repeat1(char c, int n) {
3     String answer = "";
4     for (int j=0; j < n; j++)
5       answer += c;
6     return answer;
7   }
8
9   /** Uses StringBuilder to compose a String with n copies of character c. */
10  public static String repeat2(char c, int n) {
11    StringBuilder sb = new StringBuilder( );
12    for (int j=0; j < n; j++)
13      sb.append(c);
14    return sb.toString( );
15  }
```

Estudios experimentales

repeat('*', 40)

```
1   /** Uses repeated concaten
2   public static String repeat1
3     String answer = "";
4     for (int j=0; j < n; j++)
5       answer += c;
6     return answer;
7   }
8
9   /** Uses StringBuilder to c
10  public static String repeat2(char c, int n) {
11    StringBuilder sb = new StringBuilder( );
12    for (int j=0; j < n; j++)
13      sb.append(c);
14    return sb.toString( );
15  }
```



n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Estudios experimentales

- Mientras *repeat1* ya está tardando más de 3 días en componer una cadena de 12.8 millones de caracteres, *repeat2* es capaz de hacer lo mismo en una fracción de segundo.
- También vemos algunas tendencias interesantes en cómo los tiempos de ejecución de los algoritmos dependen cada uno del tamaño de *n*.
- A medida que el valor de *n* se duplica, el tiempo de ejecución de *repeat1* aumenta más de cuatro veces, mientras que el tiempo de ejecución de *repeat2* se duplica, aproximadamente.

Desafíos del análisis experimental

- Los tiempos de ejecución experimentales de dos algoritmos son difíciles de comparar directamente a menos que los experimentos se realicen en los mismos entornos de hardware y software.
- Los experimentos pueden hacerse solamente en un conjunto limitado de datos de prueba; por lo tanto, dejan fuera los tiempos de ejecución de los datos no incluidos en el experimento (y estos datos pueden ser importantes).
- Un algoritmo debe ser completamente implementado para ejecutarlo para estudiar experimentalmente su tiempo de ejecución.

Más allá del análisis experimental

- Nuestro objetivo es desarrollar un enfoque para analizar la eficiencia de los algoritmos que:
  1. Nos permita evaluar la eficiencia relativa de cualquiera de los dos algoritmos de una manera que sea independiente del entorno de hardware y software.
  2. Se realiza estudiando una descripción de alto nivel del algoritmo sin necesidad de implementación.
  3. Que tome en cuenta todas las entradas posibles.

Contando Operaciones Primitivas

- Definimos un conjunto de operaciones primitivas como las siguientes:
  - Asignación de un valor a una variable
  - “Seguir” una referencia de objeto
  - Realizar una operación aritmética (por ejemplo, sumar dos números)
  - Comparación de dos números
  - Acceso a un elemento de un arreglo con índice
  - Llamar a un método
  - Retorno de un método

Contando Operaciones Primitivas

- Idealmente, este podría considerarse como el tipo de operación básica que ejecuta el hardware, aunque estas operaciones primitivas se traducirán a un pequeño número de instrucciones.
- En lugar de intentar determinar el tiempo de ejecución específico de cada operación primitiva, simplemente contaremos cuántas operaciones primitivas se ejecutarán y usaremos este número *t* como una medida del tiempo de ejecución del algoritmo.
- El número *t* de operaciones primitivas que realiza un algoritmo será proporcional al tiempo real de ejecución de ese algoritmo.

Contando Operaciones Primitivas

constantes n=...

tipos elemento=...

vector=array [1..n] de elementos

// devuelve la posición donde se encuentra el elemento “c” en el
// vector “a”, previamente ordenado, ó 0 si “c” no se encuentra en “a”

func buscar(a:vector; c:elemento) **dev** (r:entero)

var j:entero

alg

j:=1	// 1	1 OP
mientras a[j]<y **Y** j<n	// 2	4 OP
j:=j+1	// 3	2 OP
fmientras	// 4	
si a[j]=c :	// 5	2 OP
r:=j	// 6	1 OP
|**sino:** r:=0	// 7	1 OP
fsi	// 8	

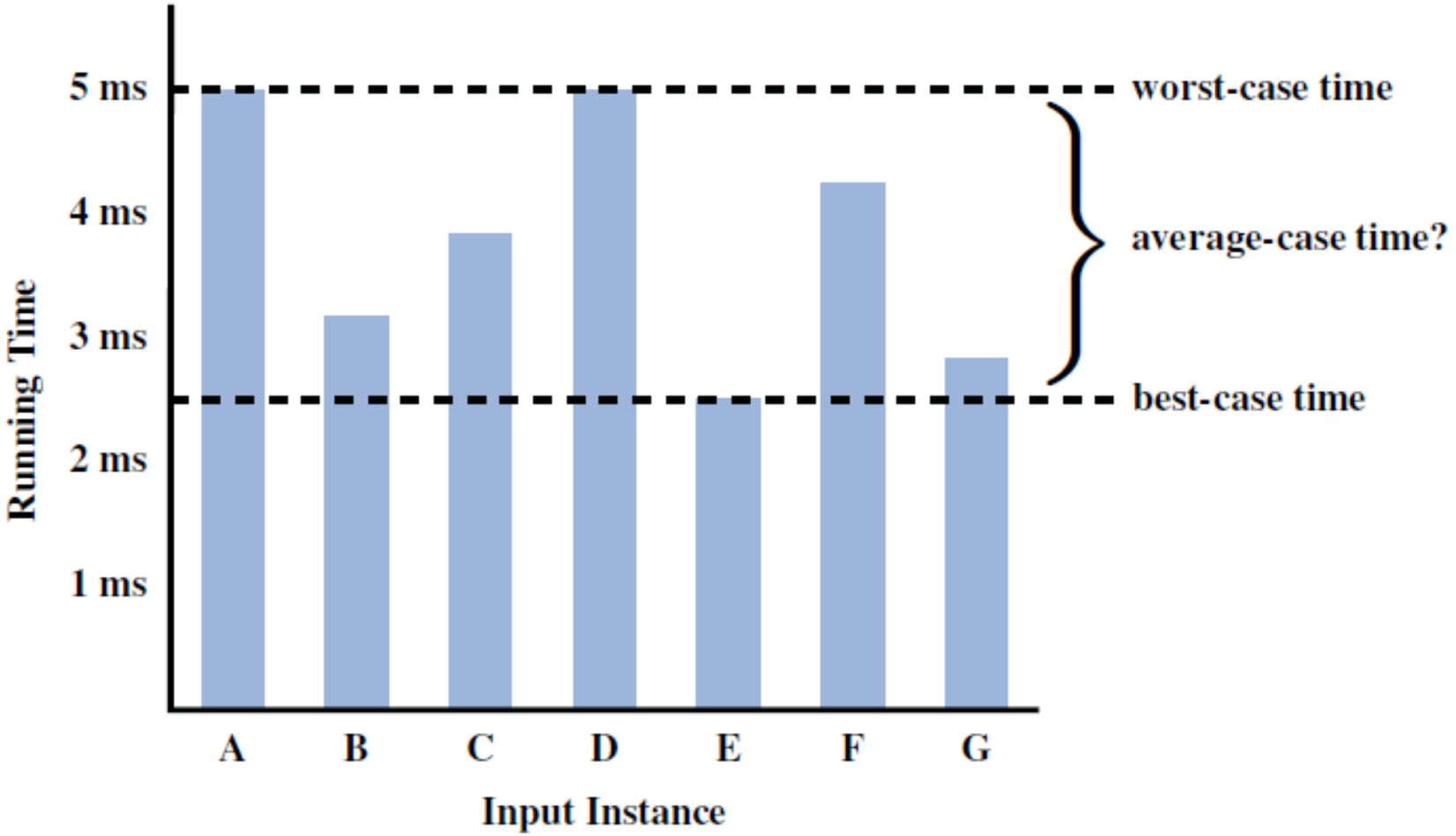
Medición de operaciones como una función del tamaño de entrada

- Para capturar el orden de crecimiento del tiempo de ejecución de un algoritmo, asociaremos, con cada algoritmo, una función *f (n)* que caracteriza el número de operaciones primitivas que se realizan en función del tamaño de entrada *n*.

El peor caso de la entrada

- Un algoritmo puede funcionar más rápido en algunas entradas que en otras del mismo tamaño.
- Por lo tanto, podemos desear expresar el tiempo de ejecución de un algoritmo como la función del tamaño de entrada obtenido tomando el promedio de todas las entradas posibles del mismo tamaño.
- Desafortunadamente, este análisis de casos promedio suele ser muy difícil (sofisticada teoría de probabilidad).
- El análisis del peor caso es mucho más fácil que el análisis del caso promedio, ya que sólo requiere la capacidad de identificar el input del peor caso, que es a menudo simple.
- El peor caso es el que mejores “músculos” le pide al algorítmo.
- Dependiendo de la distribución de entrada, el tiempo de ejecución de un algoritmo puede estar en cualquier lugar entre el tiempo del peor caso y el tiempo del mejor caso.

El peor caso de la entrada



7 funciones

- Veremos las siete funciones más importantes utilizadas en el análisis de algoritmos.
- Usaremos solamente estas siete funciones simples para casi todos los análisis que hacemos en este curso.

La función constante

- La función más simple que podemos pensar es la función constante, es decir, f (n) = c,
- Ej. $c = 5$, $c = 27$, or $c = 2^{10}$
- Independientemente del valor de *n* se asigna a *c*.
- La función constante fundamental es g (n) = 1, y esta es la función constante típica que usaremos en este curso.
- Cualquier función constante, f (n) = c, puede escribirse como una constante *c* veces *g(n)*. En este caso sería, *f (n) = c.g(n)*
- Tan simple como es la función constante es útil en el análisis de algoritmos porque caracteriza el número de pasos necesarios para realizar una operación básica en un ordenador, como sumar dos números, asignar un valor a una variable o comparar dos números.

La función logarítmo

- $f(n) = \log_b n$, con constante $b > 1$.
- Se define como la inversa de la potencia, $x = \log_b n$ si y solo si $b^x = n$.
- $\log n = \log_2 n$
- Reglas: para
- $a > 0$, $b > 1$, $c > 0$,and $d > 1$

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

- Como cuestión práctica, observamos que la regla 4 nos da una manera de calcular el logaritmo de la base-dos en una calculadora que tiene logaritmo de base 10, como: $\log_2 n$ = LOG n/LOG 2

La función lineal

- $f(n) = n$.
- Por ejemplo, comparar un número *x* con cada elemento de un arreglo de tamaño *n* requerirá *n* comparaciones.
- La función lineal también representa el tiempo de ejecución para cualquier algoritmo que procesa cada uno de los *n* objetos que no están ya en la memoria del ordenador, ya que la lectura en los *n* objetos demandará de *n* operaciones de lectura.

La función N-Log-N

- $f(n) = n \log n$
- Esta función crece un poco más rápidamente que la función lineal y mucho menos rápidamente que la función cuadrática.
- Por lo tanto, preferiríamos mucho un algoritmo con un tiempo de ejecución que sea proporcional a n log n, que uno con tiempo de ejecución cuadrático.

La función cuadrática

- $f(n) = n^2$
- La razón principal por la cual la función cuadrática aparece en el análisis de algoritmos es que hay muchos algoritmos que tienen bucles anidados, donde el bucle interno realiza un número lineal de operaciones y el bucle externo se realiza un número lineal de veces.
- Por lo que el algoritmo se ejecuta $n*n= n^2$

La función cuadrática

- En 1787, un maestro alemán decidió mantener a sus alumnos de 9 y 10 años ocupados sumando los números enteros de 1 a 100. Pero casi de inmediato uno de los niños afirmó tener la respuesta. El maestro sospechó, pues el estudiante sólo tenía la respuesta en su pizarra. Pero la respuesta, 5050, era correcta y el estudiante, Carl Gauss, llegó a ser uno de los matemáticos más grandes de su tiempo.
- Suponemos que el joven Gauss utilizó la siguiente identidad. Para n>1

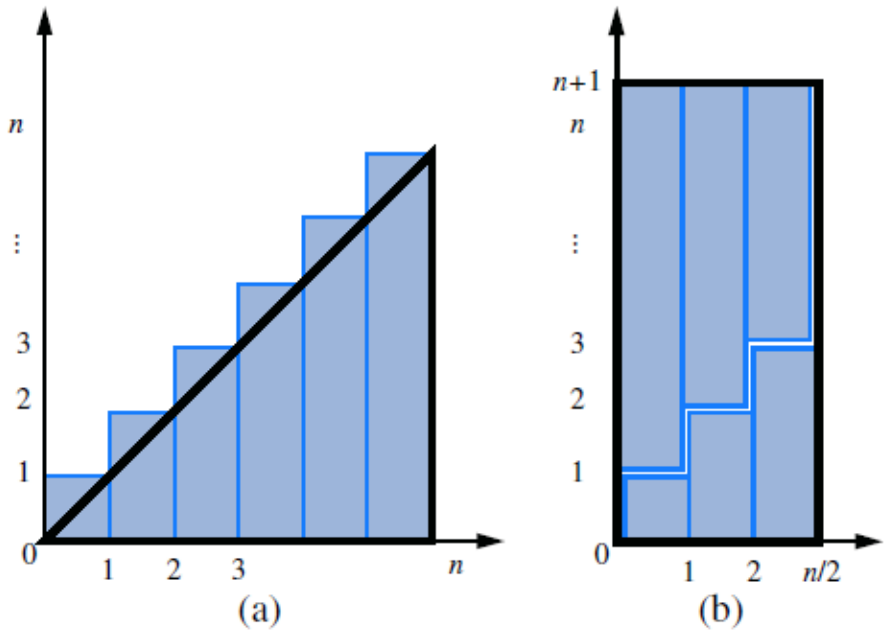
$$1+2+3+\cdots+(n-2)+(n-1)+n=\frac{n(n+1)}{2}$$

La función cuadrática

$$1+2+3+\cdots+(n-2)+(n-1)+n=\frac{n(n+1)}{2}$$

- Justificación visual

- Para ser justos, el número de operaciones es $n^2 / 2 + n / 2$, por lo que esto es poco más de la mitad del número de operaciones que un algoritmo que utiliza **n** operaciones cada vez que se realiza el bucle interno. Pero el orden de crecimiento sigue siendo cuadrático para *n*



Un ejemplo de algoritmo cuadrático

```
proc burbuja(ent/sal a:array de [n] entero)
var i,j:entero
alg
  desde i:=1 hasta n-1                          // 1
    j:=1                                        // 2
    mientras j<=n-i                             // 3
      si a[j]>a[j+1]:                           // 4
        <a[j],a[j+1]> := <a[j+1],a[j]>          // 5
      fsi                                       // 6
      j:=j+1                                    // 7
    fmientras                                   // 8
  fdesde                                        // 9
fin
```

O(1) O(1) O(1) O(n) O(n) O(n) $O(n^2)$

La función cúbica y otras polinomiales

- $f(n) = n^3$

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

La función exponencial

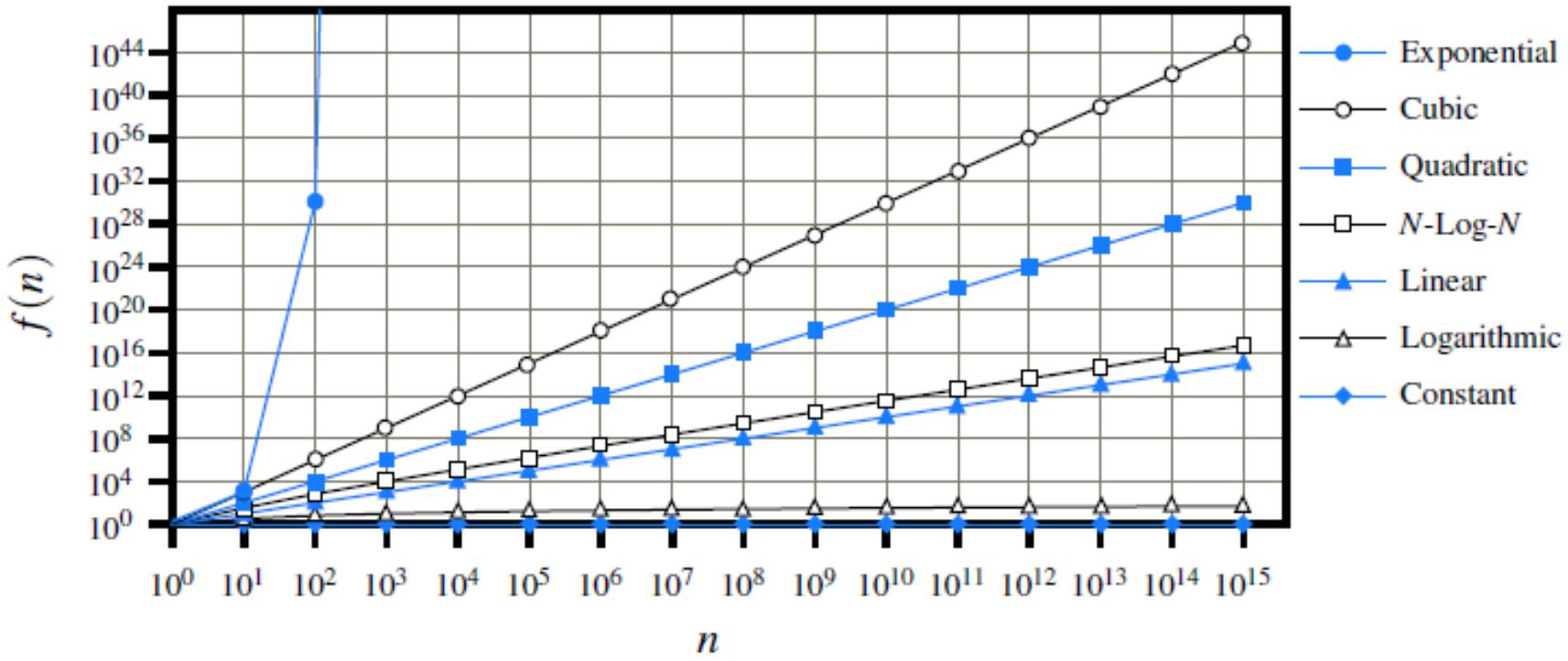
$$f(n) = b^n$$

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

- Por ejemplo, un entero que contenga n bits puede representar todos los enteros no negativos inferiores a 2^n.
- Si tenemos un bucle que comienza realizando una operación y luego duplica el número de operaciones realizadas con cada iteración, entonces el número de operaciones realizadas en la iteración n es 2^n.

Comparación de las tasas de crecimiento

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Ejemplos

- **Operaciónes de tiempo constante: O(1)**
- A.length()
- A[j] (obtención del índice)

Buscando el mayor de un arreglo

- O(n)

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

Disjunción de tres vías

- $O(n^3)$

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        for (int c : groupC)
6          if ((a == b) && (b == c))
7            return false;                    // we found a common value
8    return true;                             // if we reach this, sets are disjoint
9  }
```

- $O(n^2)$

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        if (a == b)                          // only check C when we find match from A and B
6          for (int c : groupC)
7            if (a == c)                      // and thus b == c as well
8              return false;                  // we found a common value
9    return true;                             // if we reach this, sets are disjoint
10 }
```

Análisis asintótico

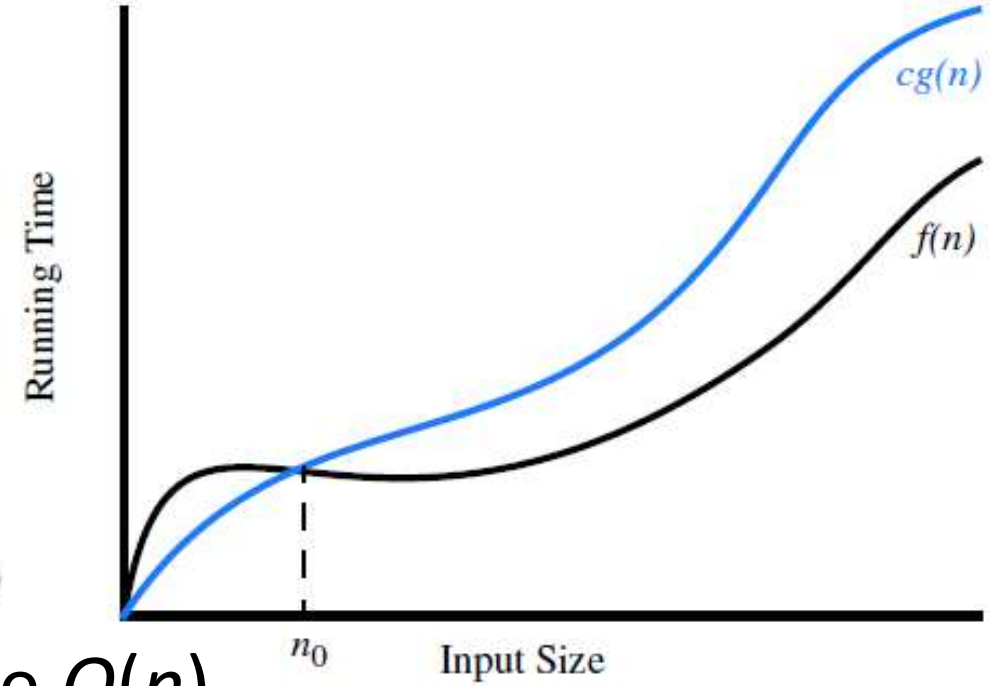
- Nos centramos en la tasa de crecimiento del tiempo de ejecución en función del tamaño de entrada *n*, tomando un enfoque de "*big-picture*".
- Por ejemplo, a menudo basta con saber que el tiempo de ejecución de un algoritmo como crece en proporción a *n*.

La notación “Big-Oh”

Sea f (n) y g (n) funciones que relacionan enteros positivos con números reales positivos. Se dice que f (n) es O (g (n)) si existe una constante real c> 0 y una constante entera $n_0 \geq 1$ tal que f (n) $\leq c \cdot g(n)$, para $n \geq n_0$.

$f(n)$ is $O(g(n))$

$f(n) \leq c \cdot g(n)$ when $n \geq n_0$



Ejemplo: 8*n*+5 es de *O*(*n*)

La notación “Big-Oh”

Ejemplo: 8*n*+5 es de *O*(*n*)

Justificación: Por la definición de gran O, necesitamos encontrar una constante real c> 0 y una constante entera $n_0 \geq 1$ tal que $8n + 5 \leq cn$ para cada entero $n \geq n_0$.

- Es fácil ver que una posible opción es c = 9 y n_0 = 5.
- De hecho, esta es una de una infinidad de opciones posibles. Por ejemplo, podríamos confiar en las constantes c = 13 y n_0 = 1.

La notación “Big-Oh”

- “ $f(n) \leq O(g(n))$”,
- El big-Oh ya denota el concepto de "menor-o-igual-a".
- Aunque es común, no es totalmente correcto decir "f (n) = O (g (n))“
- Podemos decir que "f (n) es el orden de g (n)".
- También es correcto decir “ $f(n) \in O(g(n))$”

Algunas propiedades de la notación Big O

$5n^4 + 3n^3 + 2n^2 + 4n + 1 \text{ is } O(n^4)$

$5n^2 + 3n\log n + 2n + 5 \text{ is } O(n^2)$

$20n^3 + 10n\log n + 5 \text{ is } O(n^3)$

$3\log n + 2 \text{ is } O(\log n)$

$2^{n+2} \text{ is } O(2^n)$

$2n + 100\log n \text{ is } O(n)$

Carecterizando funciones en los términos más simples

- Debemos usar la notación *big-Oh* para caracterizar a una función lo más cercana posible.
- Si bien es cierto que la función f (n) = $4n^3 + 3n^2$ es O (n^5) o incluso O (n^4), es más exacto decir que f (n) es O (n^3).
- También se considera de mal gusto incluir factores constantes y términos de orden inferior en la notación de big Oh
- Ejemplo: un algoritmo que se ejecuta en el tiempo como máximo 5n + 20log n + 4 se dirá que es de tiempo lineal O(n).

Big-Omega

- Así como la notación de Big Oh proporciona una forma asintótica de decir que una función es "menor o igual a" otra función, las siguientes notaciones proporcionan una manera asintótica de decir que una función crece a una velocidad que es "mayor o igual a“ la otra.
- Dadas las funciones $f(n)$ y $g(n)$ que van de los enteros a los reales. Decimos que $f(n)$ es $\Omega(g(n))$, y decimos “$f(n)$ es omega de $g(n)$” si $g(n)$ es $O(f(n))$, si, hay una constante $c > 0$ y una constante entero $n_0 \geq 1$ tal que $f(n) \geq cg(n)$, for $n \geq n_0$.

Big-Theta

- Hay una notación que nos permite decir que dos funciones crecen a la misma velocidad, con factores constantes.
- Decimos que f(n) es Θ(g (n)) y f(n), y decimos que “f(n) es Θ (g (n))”, si f(n) es de O(g (n)) $f(n)$ is $\Omega(g(n))$, es decir, hay constantes reales c'> 0 y c”> 0 y una constante entera $n_0 \geq 1$ tal que c‘g(n) ≤ f(n) ≤ c” g(n), para todo $n \geq n_0$.
- Ejemplo: $3n\log n+4n+5\log n$ es $\Theta(n\log n)$.

Análisis comparativo

- Podemos utilizar la notación Big-Oh para ordenar clases de funciones por la tasa de crecimiento asintótica.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Análisis comparativo

- El tamaño máximo permitido para una instancia de entrada que es procesada por un algoritmo en 1 segundo, 1 minuto y 1 hora
- Muestra la importancia de un buen diseño de algoritmos

Running Time (μs)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Análisis comparativo

- La importancia de un buen diseño de algoritmos va más allá de lo que puede resolverse eficazmente en una computadora determinada.
- Incluso si alcanzamos una aceleración enorme por hardware, no podemos superar la desventaja de un algoritmo asintóticamente lento.
- El aumento del tamaño máximo de un problema que se puede resolver en un tiempo fijo, utilizando un ordenador 256 veces más rápido que el anterior. Cada entrada es una función de m, el tamaño máximo del problema anterior.

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$
2^n	$m+8$

Algunas palabras de precaución

- En términos generales, cualquier algoritmo que se ejecuta en tiempo O (n log n) (con un factor constante razonable) debe ser considerado eficiente.
- Incluso una función $O(n^2)$-tiempo puede ser suficientemente rápida en algunos contextos, es decir, cuando *n* es pequeño.
- Pero un algoritmo cuyo tiempo de ejecución es una función exponencial, por ejemplo, $O(2^n)$, casi nunca debe ser considerado eficiente.

Tiempos de Ejecución Exponenciales

- Para ver cuán rápido crece la función 2^n, considere la famosa historia sobre el inventor del juego de ajedrez.
- Sólo le pidió a su rey que le pagara 1 grano de arroz por la primera posición del tablero, 2 granos por el segundo, 4 granos por el tercero, 8 por el cuarto, y así sucesivamente. El número de granos en la 64ava casilla sería: 2^{63} = 9,223,372,036,854,775,808, (nueve billón de billones)
- Debemos trazar una línea entre algoritmos eficientes e ineficientes, por lo tanto, es natural hacer esta distinción entre los algoritmos que funcionan en tiempo polinomial y los que funcionan en tiempo exponencial.

Bibliografía

- Data Structures and Algorithms in Java™. Sixth Edition. Michael T. Goodrich, Department of Computer Science University of California. Roberto Tamassia, Department of Computer Science Brown University. Michael H. Goldwasser, Department of Mathematics and Computer Science Saint Louis University. Wiley. 2014.