Índice general

Capítulo 1

Fundamentos de los Algoritmos

1.1. Introducción al Análisis de algoritmos

Un algoritmo es una secuencia de instrucciones que transforma una entrada (*input*) en una salida. Un ejemplo computacional básico es el *problema del ordenamiento*. Dado un conjunto de N elementos, se pueden ordenar en un orden dado, esto es aplicable a todo tipo de situaciones.

Hay distintos tipos de algoritmos que se puede implementar, tales como bubblesort, heapsort, mergesort, quicksort, radixsort, etc que veremos en el transcurso del curso.

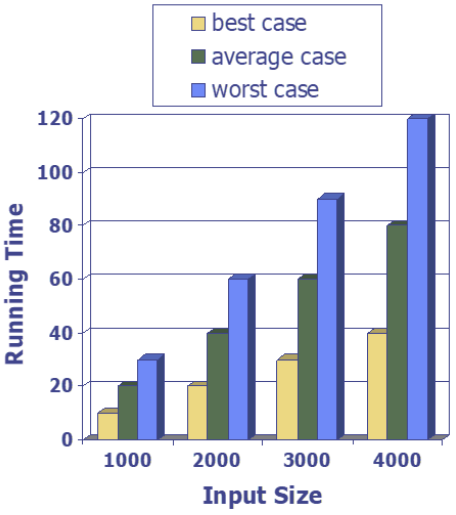
Es importante que el algoritmo sea correcto en el sentido en que ordene lo que le pido como yo quiero. Además, debe ser **rápido** y **ocupar el menos espacio posible**. Para ello, aprenderemos a medir estas variables.

1.1.1. *Tiempo de ejecución*

El tiempo de ejecución depende del **input**[1], de modo que cuanto más grande sea el input mayor es el tiempo del algoritmo.

¿Qué input debemos considerar?
Hay tres input que podemos considerar: El mejor caso es cuando la entrada ya está ordenada (por ej. alfabéticamente). Un caso promedio puede ser que los elementos estén dispuestos de forma aleatoria, mientras que el peor caso es cuando los elementos están ordenados de forma contraria al deseado.

Aquí cabe considerar que el mejor caso es poco realista, el caso promedio es difícil de caracterizar matemáticamente. Por lo que consideramos el peor caso para estudiar el tiempo de ejecución, puesto que es más fácil de analizar y porque nos da garantías de centrarnos en el peor caso posible (ser pesimistas) y preparar al algoritmo para este.



[1] conjunto de datos que se introduce en un sistema

1.1.2. *Memoria*

Es importante conocer cuánto espacio está usando el algoritmo y optimizarlo. El espacio a medir será el espacio nuevo, por ejemplo, cada vez que se llama a una función ésta ocupa un espacio en la *"pila de ejecución"* que puede ser llenada mediante funciones recursivas.

Usualmente se da un tradeoff entre tiempo y espacio, es decir, podemos hacer que nuestros algoritmos sean más rápidos a cambio de ocupar más espacio, o bien si se quiere reducir espacio los algoritmos serán más lentos.

Ahora bien, las variables de tiempo y espacio se miden mediante **estudios experimentales** y **análisis teóricos** de algoritmos.

1.1.3. *Estudios experimentales*

La idea se basa en escribir un programa implementando el algoritmo, ejecutarlo con inputs de diferente tamaño y composición, se graba el tempo que toma esas ejecuciones y se realiza una gráfica para distintos tamaños de input para visualizar cuánto tiempo toma el algoritmo.
Veamos como ejemplo la comparación de tres algoritmos que organizan la información sobre razas de perro y se quiere ordenar de forma alfabética

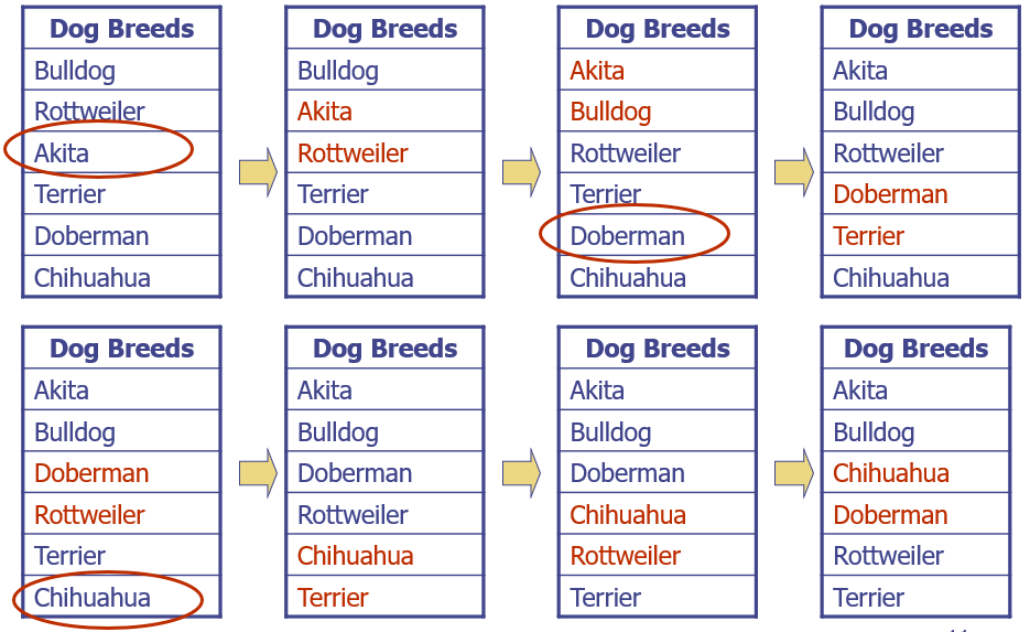


Figura 1.1: Funcionamiento de **Insertion Sort**

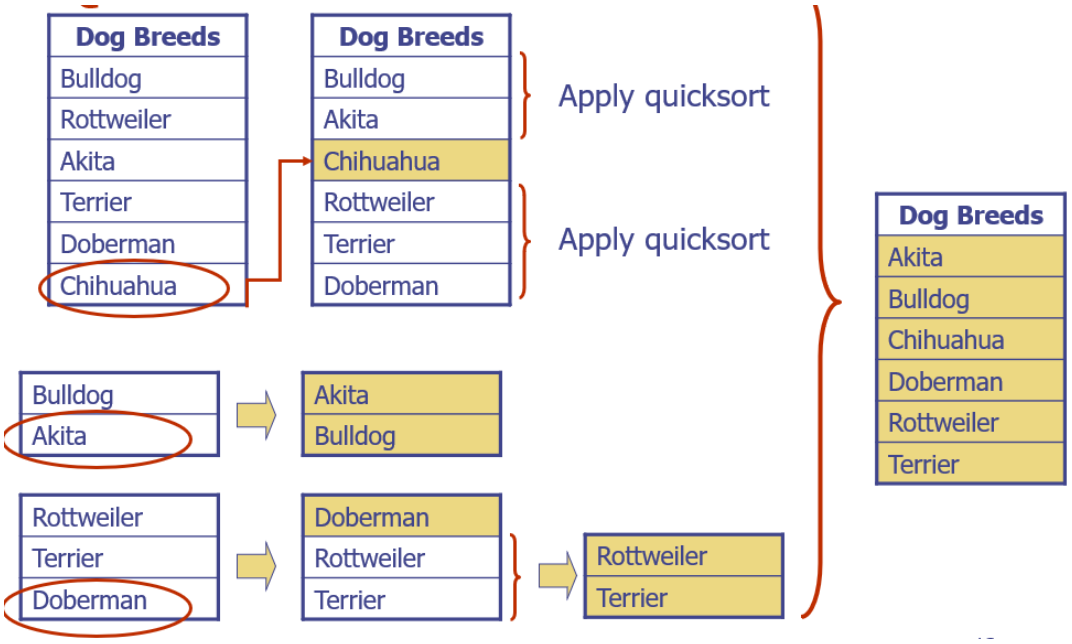


Figura 1.2: Funcionamiento de **Quick Sort**

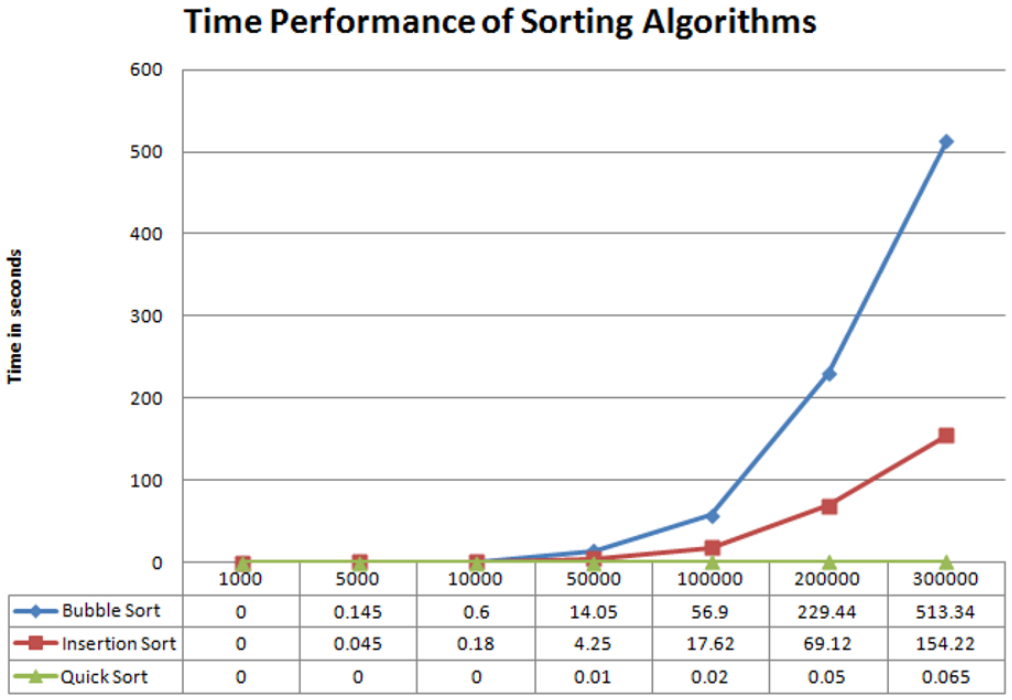


Figura 1.3: Comparación de los algoritmos Bubble Sort, Insertion Sort y Quick Sort para la peor entrada

Podemos apreciar que *quicksort* es el mejor algoritmo de los tres, dado que es el tiene menor tiempo de ejecución.

LIMITACIONES DE LOS EXPERIMENTOS

Sin embargo, no siempre es recomendable hacer estudios experimentales. Primero; dependen de la máquina en la que estamos ejecutando el programa, como también del sistema operativo de ésta y del compilador que se use para ejecutar el algoritmo, por

lo que no es un buen método para comparar la velocidad del algoritmo.
Además, no se puede ejecutar el algoritmo con todos las posibles entradas, puesto que tomaría demasiado tiempo. En ese sentido, es complicado encontrar inputs que sean representativos de todo lo que se puede encontrar en el algoritmo. Por último, la limitación principal es que **toma mucho tiempo** realizar estos estudios.

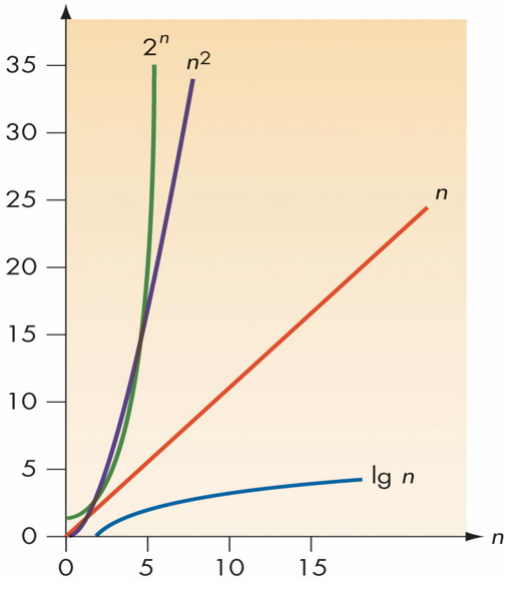
Dadas estas limitaciones es válido preguntarnos qué se puede hacer mejor. Queremos una metodología para analizar el tiempo de ejecución de un algoritmo que

- considere todos los posibles inputs
- permita comparar dos algoritmos de una forma que sea independiente de la máquina y de quien programó el algoritmo (*software environment*)
- se pueda realizar estudiando una descripción de alto nivel del algoritmo sin implementarlo realmente.

1.1.4. *Análisis teóricos*

Nos centraremos en caracterizar el tiempo o espacio que toma un algoritmo en función del número de elementos y ver cómo crece estas variables, es decir, caracterizar el tiempo que toma el algoritmo o bien, el espacio que ocupa una estructura de datos como una función, que en función del tamaño del input nos dice cuánto tiempo o espacio ocupa.
Para ello, necesitaremos siete funciones relevantes

- Constante ≈ 1
- Logarítmica $\approx \log n$
- Lineal $\approx n$
- N-log-N $\approx n \log n$
- Cuadrática $\approx n^2$
- Cúbica $\approx n^3$
- Exponencial $\approx 2^n$

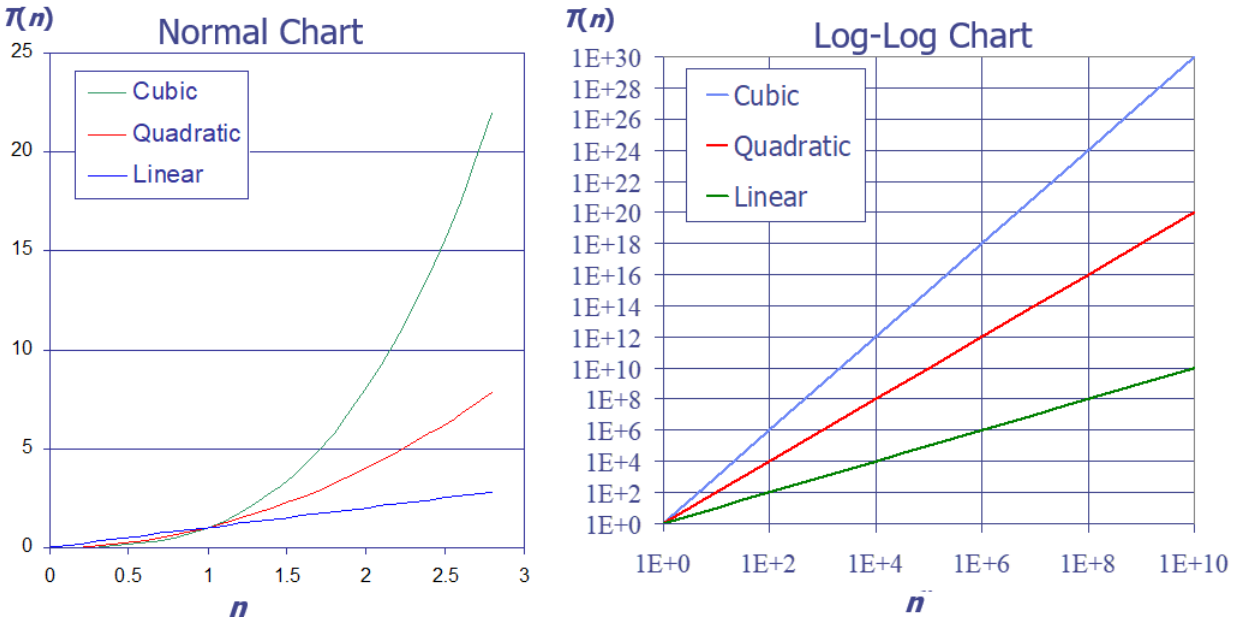


Estas están dispuestas de modo que a medida que crece el input, nuestro algoritmo tarda más tiempo en ejecutarse. Por ello, lo ideal es que el algoritmo tome tiempo constante, es decir, sea independiente de cuánto crezca nuestro input el tiempo de ejecución será siempre el mismo, o bien que tome tiempo logarítmico o lineal.
Por el contrario, no queremos que nuestros algoritmos sean exponenciales, porque tomaría mucho tiempo en ejecutarse.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3{,}3 \times 10$	10^2	10^3	10^3	$3{,}6 \times 10^6$
10^2	6.6	10^2	$6{,}6 \times 10^2$	10^4	10^6	$1{,}3 \times 10^{30}$	$9{,}3 \times 10^{157}$
10^3	10	10^3	10×10^3	10^6	10^9		
10^4	13	10^4	13×10^4	10^8	10^{12}		
10^5	17	10^5	17×10^5	10^{10}	10^{15}		
10^6	20	10^6	20×10^6	10^{12}	10^{18}		

Cuadro 1.1: Tasa de crecimiento

En la figura se aprecia que mientras mayor sea el exponente mayor será la pendiente en el gráfico log-log, lo cual es lo opuesto a lo que buscamos en nuestros algoritmos, puesto que queremos que la pendiente sea lo menor posible.



Por otra parte, recordemos la relación que hay entre el logaritmo y ...

$$
\begin{aligned}
y &= a \cdot x^b \\
\log y &= \log\left(a \cdot x^b\right) \\
\log y &= \log a + \log x^b \\
\log y &= \log a + b \log x \\
\Rightarrow y' &= m \cdot x' + n
\end{aligned}
$$

En un gráfico log-log, la pendiente de la recta corresponde a la tasa de crecimiento de la función.

Es necesario recordar y tener presentes las propiedades de las siguientes funciones

LOGARITMO Y EXPONENTES

Propiedad de Logaritmo

$$\begin{aligned}\log_b a = c &\Rightarrow a = b^c\\ \log_b(xy) &= \log_b x + \log_b y\\ \log_b(x/y) &= \log_b x - \log_b y\\ \log_b x^a &= a\log_b x\\ \log_b a &= \frac{\log_x a}{\log_x b}\end{aligned}$$

Propiedad de la Exponencial

$$\begin{aligned}a^{(b+c)} &= a^b a^c\\ a^{bc} &= \left(a^b\right)^c\\ \frac{a^b}{a^c} &= a^{(b-c)}\\ b &= a^{\log_a b}\\ b^c &= a^{c\log_a b}\end{aligned}$$

SUMATORIAS Y SERIRES

Definición

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

Series aritméticas

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

Series geométricas, con $a > 0$

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + ... + a^n = \frac{1 - a^{n+1}}{1 - a} \tag{1.1}$$

Recordemos que buscamos un método para analizar el tiempo de ejecución de un algoritmo que

- considere todos los posibles inputs
- permita comparar dos algoritmos de una forma que sea independiente de la máquina y de quien programó el algoritmo (*software environment*)
- se pueda realizar estudiando una descripción de alto nivel del algoritmo sin implementarlo realmente.

1.1.5. *Pseudocódigos*

Si bien no queremos programar el algoritmo (por el tiempo que tarda), necesitamos .ªterrizarlo"de alguno forma y lo haremos por medio de lo que llamaremos pseudocódigo, el cual utiliza una descripción de alto nivel de un algoritmo en vez de implementarlo (es decir, programarlo).

Un pseudocódigo caracteriza nuestras soluciones (tiempo de ejecución) como una función del tamaño de nuestra entrada, que usualmente denotamos como n. Además,

considera todas las entradas posibles y nos permite evaluar la velocidad de un algoritmo independiente del entorno de hardware/software.

Es un nivel intermedio entre un lenguaje de programación y el mundo de las ideas, es decir, es más flexible que un lenguaje de programación (en cuanto a sintaxis), pero es mucho más estructurado que una descripción en lenguaje natural.

Antes de definir nuestros algoritmos debemos representarlos en pseudocódigos, es importante recalcar que **el pseudocódigo se abstrae de los detalles de programación.**

Ejemplo: Algoritmo que encuentra el mayor elemento dentro de un arreglo

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax ← A[0]
  for i ← 1 to n − 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```

DETALLES DEL PSEUDOCÓDIGO

- La sangría reemplaza las llaves
- Declaración de método
  `Algorithm method(arg [,arg...])`

  `Input...`

  `Output...`
- Valor retornado
  `return expression`
- Expresiones
  ← Asignación (como = en Java)
  = Prueba de igualdad (como == en Java)
- Método de llamada
  `var.method(arg[, arg...])`
  `var` se puede omitir si es´ta claro qué objeto llama al método
- Control flow
  if condition **then** true-action **[else** false-condition **]**
  while condition **do** actions

  ```
  while count ⇐ 10 do
    M ← M × count
  ```

```
    Add 1 to count
```

repeat actions **until** condition

```
repeat
  Display count_value
  Add 1 to count_value;
until count_value >10
```

for iteration-bounds **do** actions

```
for count = 1 to 10 do
  Display count + count
for each month of the year do
  Display month
```

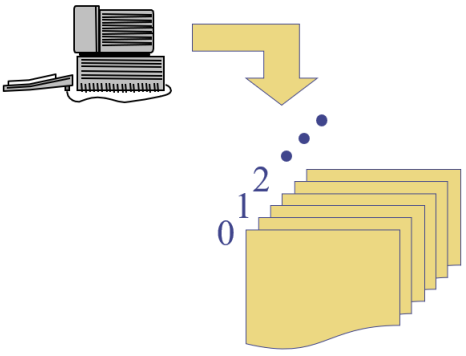
Siguiendo estas propiedades o reglas de un pseudocódigo, cualquier programador será capaz de comprenderlo, ya sea para analizar o para programar el algoritmo.

1.1.6. *Modelos Teóricos*

Una vez que se tiene nuestras soluciones expresadas en pseudocódigos, podemos analizarlas sobre un modelo. Un modelo simplifica ciertas características del algoritmo y durante el curso asumiremos como ciertas estas características (no captura toda la realidad, es una simplificación). Hay muchos modelos, pero utilizaremos principalmente el modelo RAM.

THE RANDOM ACCESS MACHINE (RAM) MODEL

Este modelo se basa en que un computador es básicamente una CPU (unidad de procesamiento) y una memoria. La memoria está compuesta por una serie de celdas, en éstas se almacenan datos y con estos datos podemos hacer una serie de operaciones. Por ejemplo, escribir dentro de una celda, leer de una celda, sumar varios valores dentro de una celda, etc. Las celdas de memoria están numeradas y el acceso a cualquier celda de la memoria lleva un tiempo unitario.



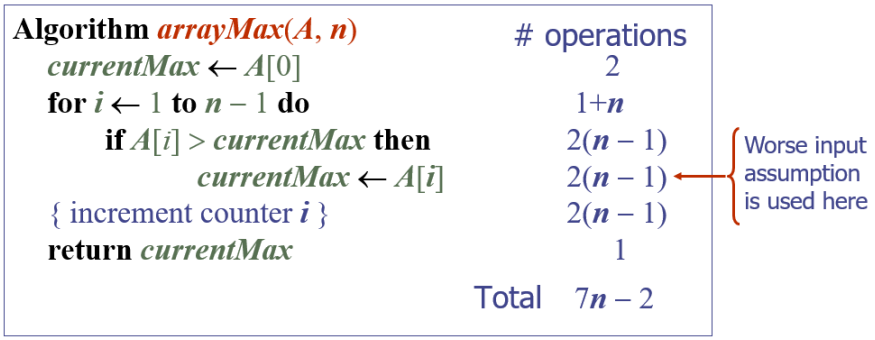
OPERACIONES PRIMITIVAS

Las operaciones primitivas son cálculos básicos realizadas por un algoritmo, son identificable en el pseudocódigo y, en gran parte, son independientes del lenguaje de programación. La definición exacta **no** es importante y se asume que **todas** las operaciones toma una misma cantidad de tiempo en el modelo RAM.
Ejemplos:

- Evaluar una expresión: $N - 1$
- Asignar un valor a una variable: Max $\leftarrow 1$
- Indexación en una matriz: A[2]
- Llamar a un método: Max(L)
- Retornando de un método: `return 3`

Contando operaciones primitivas

Al inspeccionar el pseudocódigo podemos determinar el número máximo de operaciones primitivas para un algoritmo en función del tamaño de la entrada n



Cabe notar que en el ciclo `if` estamos asumiendo que siempre se evalúa verdadero, es decir, siempre ingresamos a la linea línea 5 del pseudocódigo, de modo que aquí se refleja que estamos haciendo análisis de peor caso. Cuando hay un condicional (`if`) siempre se evalúa la condición que hace el mayor número de operaciones.

ESTIMANDO EL TIEMPO DE EJECUCIÓN

El algoritmo `arrayMax` ejecuta $7n - 2$ operaciones primitivas en el peor de los casos. Sea

a: tiempo empleado por la operación primitiva más rápida
b: tiempo empleado por la operación primitiva más lenta
$T(n)$ el tiempo del peor caso de `arrayMax`. Entonces,

$$a(7n - 2) \leq T(n) \leq b(7n - 2)$$

Por tanto, el tiempo de ejecución está limitado por dos funciones lineales.

TASA DE CRECIMIENTO DEL TIEMPO DE EJECUCIÓN

Aunque no podemos determinar la función de tiempo exacta, sabemos que tiene un límite superior dado por $T(n) \leq b(7n - 2)$.
Ahora bien, ¿cuál es la tasa de crecimiento de $b(7n - 2)$? Es lineal, por ende las constantes b, 7 y 2 no son relevantes.

En una gráfica log-log, la pendiente de la recta $b(7n-2)$ es igual a la pendiente de la recta n. Lo que nos interesa conocer es cómo crece el tiempo de ejecución de nuestros algoritmos, entonces decimos que $b(7n-2)$ crece igual que la función n (a la misma velocidad). Además, el gráfico logarítmico muestra que la tasa de crecimiento no se ve afectada (es 1 en ambos casos).

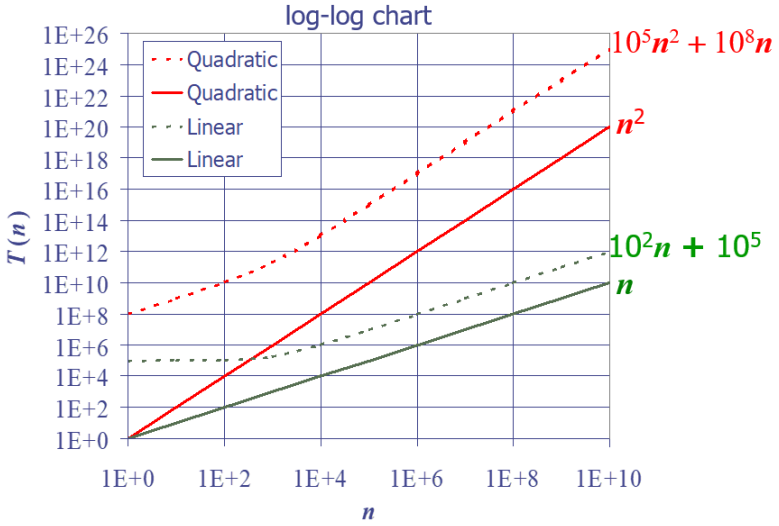
Figura 1.4: tasa de crecimiento de las funciones n y $b(7n-2)$

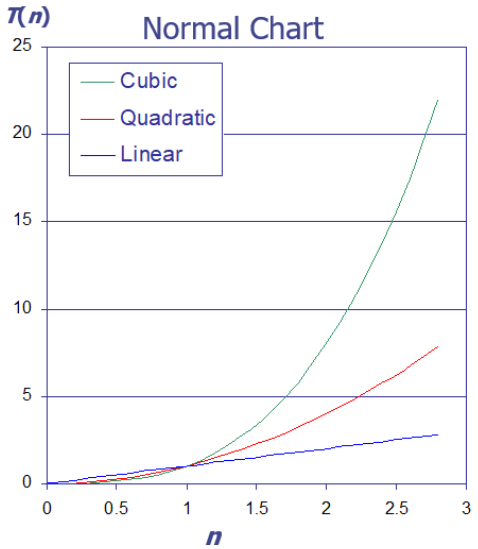
Qué pasa si tenemos

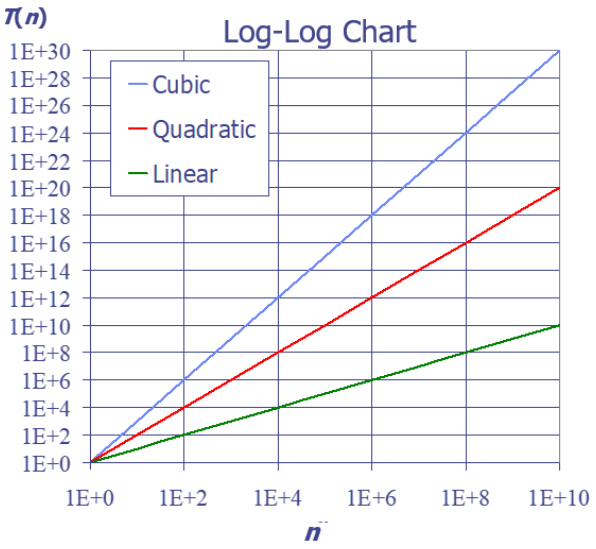
$$T(n) = n^2 + 10^8 n + 3001$$

tal como mencionamos las constantes son despreciables, por lo que tanto 3001 como 10^8 lo son. Lo que desprendemos de esta función es que el algoritmo está modelado por una función cuadrática.

En resumen, lo que nos importa es la función modela el crecimiento del tiempo de ejecución, no las constantes que lo acompaña. La tasa de crecimiento no es afectada por factores constantes o términos de orden inferior.







Lo que haremos es centrarnos en el conteo de operaciones que nos dará una estimación del tiempo de ejecución, que no está influida ni por cambios de máquina ni quién lo programó, etc.

1.1.7. *Notación Asintótica*

Dadas las funciones $f(n)$ y $g(n)$, decimos que $f(n)$ es $O(g(n))$ si existe $c > 0$ y $n_0 \geq 1$, tal que

$$f(n) \leq c \cdot g(n); \qquad \text{para } n \geq n_0$$

Ejemplo 1. $2n + 10$ es $O(n)$

$$2n + 10 \leq c \cdot n$$
$$(c - 2)n \geq 10$$
$$n \geq \frac{10}{c - 2}$$

de aquí, tenemos dos opciones:
si $c = 3$ entonces $n_0 = 10$
si $c = 12$ entonces $n_0 = 1$

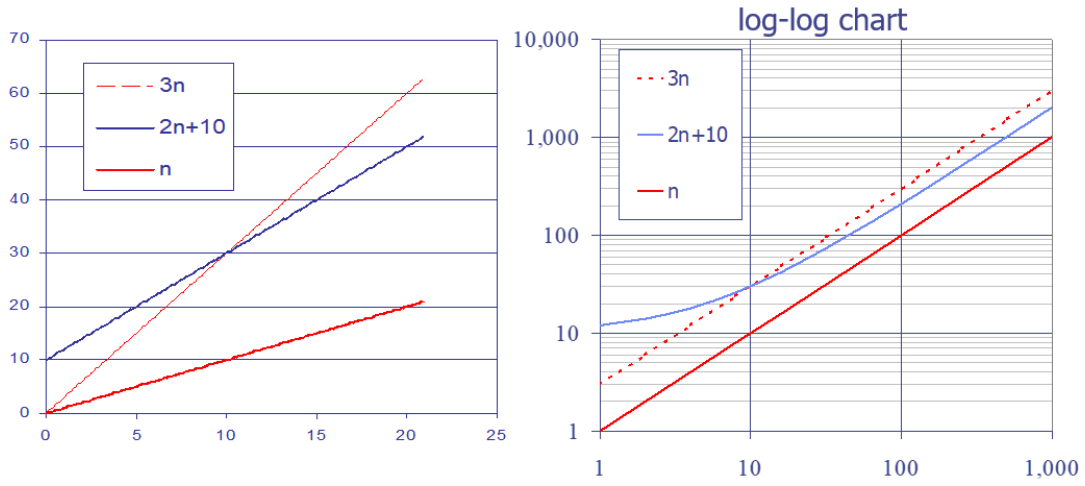
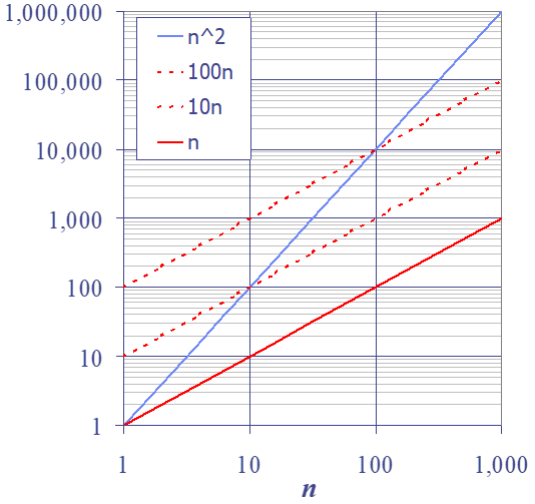


Figura 1.5: c es el factor que desplaza la función n hacia arriba, mientras que n_0 es el punto a partir del cual siempre se cumple la desigualdad y, a su vez, el que se encuentra en la intersección de ambas funciones.

Ejemplo 2. La función n^2 no es $O(n)$

$$n^2 \leq c \cdot n$$
$$n \leq c$$

La desigualdad no se puede satisfacer debido a c debe ser constante. Notamos, además que n^2 no es $O(n)$ porque n^2 crece más rápido que $O(n)$.
La gráfica nos muestra, independiente del valor que tome c, siempre habrá un n_0 que haga que la función n^2 esté por encima de la función n.



La notación asintótica da un límite superior en la tasa de crecimiento de una función. La declaración "$f(n)$ es $O(g(n))$"significa que **la tasa de crecimiento de $f(n)$ no es mayor que la tasa de $g(n)$**.

Podemos usar la notación asintótica para clasificar funciones de acuerdo con su tasa de crecimiento

	$f(n)$ es $O(g(n))$	$g(n)$ es $O(f(n))$
$g(n)$ crece más	✓	×
$f(n)$ crece más	×	✓
Mismo crecimiento	✓	✓

Cuadro 1.2: A partir de un punto, la función $f(n)$ está dominada por la función $g(n)$.

REGLAS DE LA NOTACIÓN ASINTÓTICA

Si $f(n)$ es un polinomio de grado d, entonces $f(n)$ es $O(n^d)$, es decir,

- despreciamos los términos de menor orden
- despreciamos las constantes
- decimos que "$2n^2 + 3n + 3$ es $O(n^2)$" en vez de "$2n^2 + 3n + 3$ es $O(2n^2 + 3n)$"

Utilizar la clase de funciones más pequeña posible

- "$2n$ es $O(n)$" en vez de "$2n$ es $O(n)$"
- notar que $2n$ **es** $O(n)$, pero el análisis no es ajustado, por ende no es correcto.

Usar la expresión más simple de la clase

- "$3n + 5$ es $O(n)$" en vez "$3n + 5$ es $O(3n)$"
- al igual que en el caso anterior "$3n + 5$ **es** $O(n)$", pero el análisis no es ajustado

ANÁLISIS ASINTÓTICOS DE ALGORITMOS

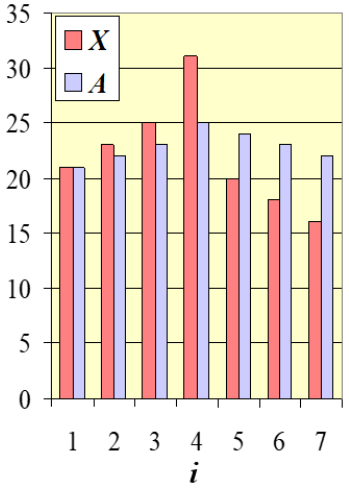
El análisis asintótico de un algoritmo determina el tiempo de ejecución en notación asintótica.
Para realizar el análisis asintótico

- Encontramos el número de operaciones primitivas ejecutadas e el pero de los casos en función del tamaño de entrada (considerar el mayor número de operaciones posibles.)
- Expresamos esta función en notación asintótica

Ejemplo. Determinamos que el algoritmo `arrayMax` n-ejecuta como máximo $7n - 2$ operaciones primitivas. Decimos que `arrayMax` se *"ejecuta en $O(n)$ tiempo*.
Dado que los factores constantes y los términos de orden inferior eventualmente se eliminan, podemos ignorarlos al contar operaciones primitivas, tal como se ve a continuación

Algorithm *arrayMax*(A, n)	primitive # operations	asymptotic analysis
currentMax ← $A[0]$	2	1
for $i \leftarrow 1$ **to** $n - 1$ **do**	$n + 1$	n
if $A[i] >$ *currentMax* **then**	$2(n - 1)$	n
currentMax ← $A[i]$	$2(n - 1)$	n
{ increment counter i }	$2(n - 1)$	n
return *currentMax*	1	1
Total	$7n - 2$	$O(n)$

Ejemplo La siguiente figura ilustra el análisis asintótico con dos algoritmos para promedios de prefijos.
El arreglo X corresponde al input, el cual posee 7 posiciones, mientras la salida corresponde al arreglo A, donde para cada posición se considera el promedio de los i valores anteriores (todos los valores hasta esa posición), esto es



$$A[i] = \frac{(X[0] + X[1] + \ldots + X[i])}{(i+1)}$$

Algoritmo 1. El siguiente algoritmo calcula los promedios de prefijos en tiempo cuadrático aplicando la definición

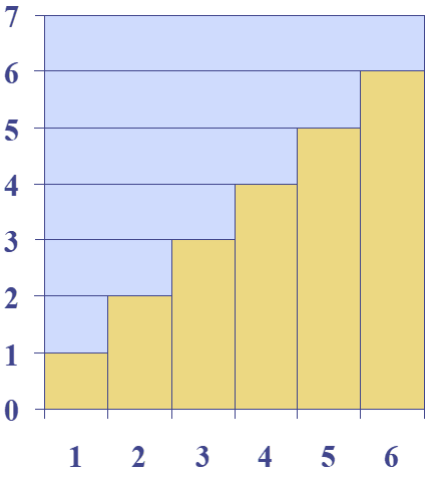
Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	*#operations*
A ← new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow 0$	n
for $j \leftarrow 0$ **to** i **do**	1+2+…+(n-1)
$s \leftarrow s + X[j]$	1+2+…+(n-1)
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Recordemos cuál es el orden de una progresión aritmética

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$$

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

de modo que, el tiempo de ejecución de `prefixAverage1` es $O(n^2)$

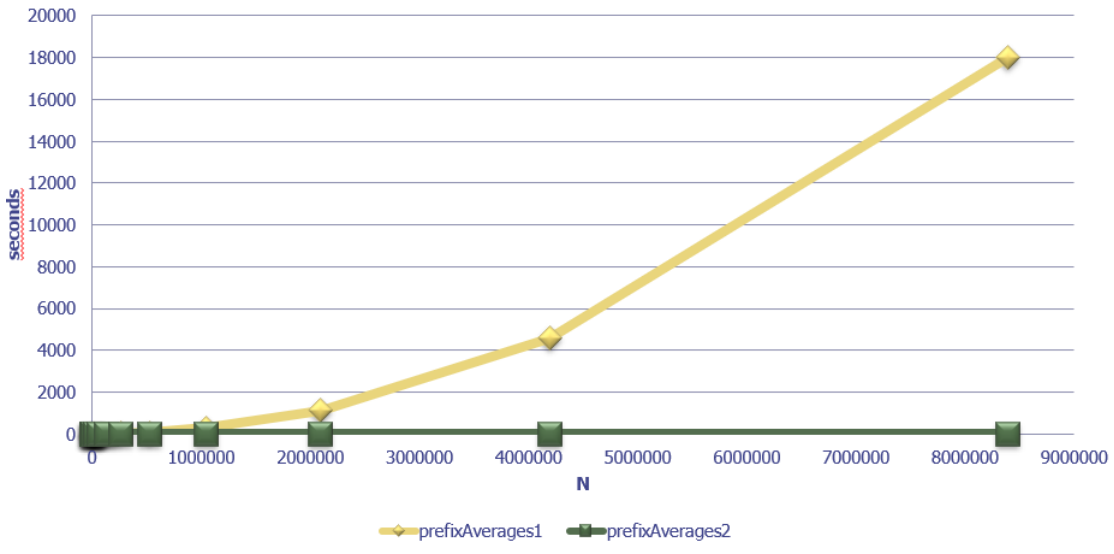


Algoritmo 2. El siguiente algoritmo calcula los promedios de prefijos en tiempo lineal manteniendo una suma acumulada.

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers                    #operations
    Output array A of prefix averages of X
    A ← new array of n integers                    n
    s ← 0                                          1
    for i ← 0 to n − 1 do                          n
        s ← s + X[i]                               n
        A[i] ← s / (i + 1)                         n
    return A                                       1
```

de modo que `prefixAverage2` es ejecutado en $O(n)$ tiempo.

Como era de esperar, si ejecutamos ambos algoritmos obtenemos lo siguiente:



VARIACIONES DE LA NOTACIÓN ASINTÓTICA

Big-Omega

$f(n)$ es $\Omega(g(n))$ si existe una constante $c > 0$ y un entero $n_0 \geq 1$, tal que

$$f(n) \geq c \cdot g; \qquad \text{para } n \geq n_0$$

Big-Theta

$f(n)$ es $\Theta(g(n))$ si existen constantes $c' > 0$, $c'' > 0$ y un entero $n_0 \geq 1$, tal que

$$c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n); \qquad \text{para } n \geq n_0$$

INTUICIÓN PARA LA NOTACIÓN ASINTÓTICA

Big-Oh: $f(n)$ es $O(g(n))$ si $f(n)$ es asintóticamente menor o igual que $g(n)$.

Big-Omega: $f(n)$ es $\Omega(g(n))$ si $f(n)$ es asintóticamente mayor o igual que $g(n)$.

Big-Theta: $f(n)$ es $\Theta(g(n))$ si $f(n)$ es asintóticamente igual a $g(n)$.

Ejemplos.
• $5n^2$ es $\Omega(n^2)$
$f(n)$ es $\Omega(g(n))$ si hay una constante $c > 0$ y $n_0 \geq 1$, tal que

$$\begin{aligned} 5n^2 &\geq c \cdot n^2 \\ 5 &\geq c \end{aligned}$$

de aquí tenemos que $c = 5$ y $n_0 = 1$ satisfacen la proposición.

• $5n^2$ es $\Omega(n)$
$f(n)$ es $\Omega(g(n))$ si hay una constante $c > 0$ y $n_0 \geq 1$, tal que

$$\begin{aligned} 5n^2 &\geq c \cdot n \\ 5n &\geq c \\ 5n &\geq 1; \qquad c = 1 \\ n &\geq 0{,}2 \end{aligned}$$

de aquí, vemos que $c = 1$ y $n_0 = 1$, recordando que n_0 debe ser entero.

• $5n^2$ es $\Theta(n^2)$
$f(n)$ es $\Theta(g(n))$ si es $\Omega(n^2)$ y $O(n^)$. Ya vimos el primer caso, así que ahora veremos el siguiente, $f(n)$ es $O(n^2)$ si hay una constante $c > 0$ y $n_0 \geq 1$, tal que

$$\begin{aligned} 5n^2 &\geq c \cdot n^2 \\ 5 &\geq c \end{aligned}$$

así, vemos que para $c = 5$ y $n_0 = 1$ se cumple lo pedido, de modo que para estas constantes $f(n)$ es $\Theta(g(n))$.

RESUMEN

Variables a considerar

- Tiempo de ejecución
- Espacio (uso de memoria)

¿Cómo los calculamos o estimamos?

- Estudios experimentales
- Análisis teórico

Teórico vs Experimental

El análisis teórico tiene muchas propiedades excelentes:

- Utiliza una descripción de alto nivel del algoritmo en lugar de una implementación.
- Caracteriza el tiempo de ejecución en función del tamaño de entrada, n.
- Tiene en cuenta todas las entradas posibles
- Independiente de hardware/software.

Entonces... ¿cuándo son útiles los experimentos? Los experimentos son útiles para:

- Comparar algoritmos con los mismos tiempos de ejecución asintóticos.
- Comparar algoritmos con tiempo de ejecución exponencial.
- Comparar el rendimiento del algoritmo en tipos específicos de instancias, incluso si los algoritmos han conocido diferentes tiempos de ejecución asintóticos.
- Demostrar o confirmar las diferencias conocidas en los tiempos de funcionamiento asintóticos.

1.2. Corrección de Algoritmos

1.2.1. *Técnicas de justificación*

Debemos justificar que nuestras afirmaciones sobre la exactitud y el tiempo de ejecución de nuestros algoritmos, por medio de diversas técnicas, tales como:

El contraataque: Contrapositivo y contradicción
Inducción matemática
Ciclo invariante

Contrajemplo:

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

Si $n = 3$, entonces

$$\sum_{i=1}^{3} i = 1+2+3 = 6$$
$$\frac{n(n-1)}{2} = \frac{3\times 2}{2} = 3$$

EL CONTRAATAQUE

Contrapositivo: En lugar de probar *"Si p es verdadero, entonces q es verdadero"*, demostramos **"Si q es falso, entonces p es falso"**

$$p \to q = \neg q \to \neg p$$

Ejemplo.

'Si el cuadrado de un número es impar, entonces el número es impar"

"Si un número no es impar, entonces su cuadrado no es impar"

Contradicción: $p \iff \neg p$ conduce a la contradicción.
Ejemplo. $\sqrt{2}$ es irracional.

Por contracción asumiremos que es racional, Luego existe p y q tal que $\sqrt{2} = p/q$

$$2 = \frac{p^2}{q^2} \Rightarrow 2q^2 = p^2$$

Entonces, p^2 es par y, por lo tanto, p es par y se puede escribir como $2k$:

$$2q^2 = p^2 = (2k)^2 = 4k^2 \Rightarrow q^2 = 2p^2$$

Por tanto, q^2 es par, por consiguiente q es par. Esto significa que tanto p y q son divisibles por 2 y esto contradice el hecho de que no tienen factores comunes. Por tanto, $\sqrt{2}$ es irracional.

Sea $a \times b$ impar, entonces a es impar y b es impar. Trate de demostrar que para $a \times b$ impar, a o b son pares y llegan a una contradicción.
Existen, al menos, tres formas de probar lo pedido

1. $a \times b$ impar, a par (se puede escribir como $2s$), b impar. Entonces $a \times b = 2s \times b$, lo cual es par, de modo que se produce una contradicción

2. $a \times b$ impar, a impar y b par (se puede escribir como $2t$). Entonces $a \times b = a \times 2t$, lo cual es par, de modo que se produce una contradicción

3. $a \times b$ impar, a par ($2s$) y b par ($2t$). Entonces $a \times b = 2s \times 2t$, lo cual es par, de modo que se produce una contradicción

INDUCCIÓN

Una técnica para demostrar que la declaración $P(n)$ es verdadero para todos los enteros positivos $n > n_0$. n_0 es el caso base.

Pasos a seguir:
Caso base: demuestre que $P(n)$ es cierto para $n = n_0$.
Caso inductivo: demuestre que si $P(n)$ es cierto para $n = n_0, \ldots, k$, entonces $P(n)$ también es cierto para $n = k + 1$.

Ejemplo.

$$\sum_{i=0}^{n} 3^i = \frac{3^{n+1} - 1}{2}$$

Caso base $i = 0$, $(i = n_0)$

$$\sum_{i=0}^{1} 3^i = 3^0 = 1; \qquad \frac{3^{0+1} - 1}{2} = 1$$

Caso inductivo

$$\text{Asumir: } \sum_{i=0}^{k} 3^i = \frac{3^{k+1} - 1}{2} \qquad \text{Probar: } \sum_{i=0}^{k+1} 3^i = \frac{3^{(k+1)+1} - 1}{2}$$

$$\begin{aligned} \sum_{i=0}^{k+1} 3^i = \sum_{i=0}^{k} 3^i + 3^{k+1} &= \frac{3^{k+1} - 1}{2} + 3^{k+1} \\ &= \frac{3^{k+1} - 1 + 2 \cdot 3^{k+1}}{2} \\ &= \frac{3 \cdot 3^{k+1} - 1}{2} \\ &= \frac{3^{(k+1)+1} - 1}{2} \end{aligned}$$

CICLO INVARIANTE

Esta técnica se usa generalmente para probar la corrección de un algoritmo, especialmente para aquellos que usan bucles (ciclo for, ciclo while).

Tenemos que establecer una declaración relacionada con el ciclo (el ciclo invariante) y demostrar que el enunciado es verdadero al principio y/o al final de cada ciclo.

La técnica que usamos es muy similar a la inducción matemática. Después de establecer el ciclo invariante, probamos que es cierto antes de entrar en el ciclo.
Entonces,

Suponemos que la afirmación es verdadera al principio y/o al final del k-ésimo ciclo.

Probamos que también es cierto para el principio y/o el final del ciclo $(k+1)$ o la declaración para el siguiente ciclo no existe (porque el ciclo termina).

Como resultado, llegamos a la conclusión de que el ciclo invariante es verdadero y el algoritmo es correcto.

	primitive # operations	asymptotic analysis
Algorithm ***arrayMax***(A, n)		
currentMax $\leftarrow A[0]$	2	1
for $i \leftarrow 1$ **to** $n-1$ **do**	$n+1$	n
if $A[i] >$ ***currentMax*** **then**	$2(n-1)$	n
currentMax $\leftarrow A[i]$	$2(n-1)$	n
{ increment counter i }	$2(n-1)$	n
return ***currentMax***	1	1
Total	$7n-2$	$O(n)$

Ciclo invariante

•S_i: x no es igual a ninguno de los primeros i elementos de A.

Caso base: la declaración es verdadera al comienzo de los bucles:

•S_0: x no es igual a ninguno de los primeros 0 elementos de A.

Suponemos que el enunciado es verdadero hasta S_k al comienzo del ciclo $(k+1)^{th}$:

•S_k: x no es igual a ninguno de los primeros k elementos de A.

•Durante el ciclo $(k+1)^{th}$, pueden suceder dos cosas:

- si $x = A[k]$, devolvemos k, por lo que no habrá $S_k + 1$.
- si $x \neq A[k]$, continuamos con el siguiente ciclo. En este caso, sabemos por S_k que x no es igual a ninguno de los primeros k elementos de A, pero también tenemos que x no es igual al $(k+1)$ ésimo elemento de A. Por lo tanto, sabemos $S_k + 1$ también es cierto:
  - $S_k + 1$: x no es igual a ninguno de los primeros $k+1$ elementos de A

1.3. Patrón de recursividad

1.3.1. *Diseño e Implementación de Algoritmos*

Objetivos de diseño

- Exactitud
- Eficiencia
- Robustez
- Adaptabilidad
- Reutilización

Criterios de diseño

- Abstracción
- Encapsulamiento
- Modularidad

Técnicas de diseño

- Interfaces y mecanografía
- Herencia y polimorfismo
- Clases y objetos

Metas de implementación

- Patrones de diseño

PATRONES DE RECURSIVIDAD

Un patrón de diseño describe una solución a un problema de diseño de software "típico". Describe los elementos principales de una solución de una manera abstracta que se puede especializar para el problema específico en cuestión.
Ejemplos. **Recursividad, Amortización, Divide y conquistarás, Podar y buscar, Fuerza bruta, Método codicioso.**

Recursión: cuando un método se llama a sí mismo. Ejemplo clásico: la función factorial: $n! = 1 \cdot 2 \cdot 3 \ldots \cdot (n-1) \cdot n$

Definición recursiva:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

Código

```
// recursive factorial function
int recursiveFactorial(int n {
    if (n == 0) return 1;     // base case
    else return n * recursiveFactorial(n-1);     // recursive case
```

A tener consideración

Debe haber al menos un caso (el caso base), para un valor pequeño de n, que se pueda resolver directamente.

Si un problema de un tamaño n dado (lo suficiente complejo), se puede dividir en una o más versiones más pequeñas del mismo problema (caso recursivo). En este caso, lo que se debe hacer es

- Reconocer el caso base y brindarle una solución.
- Diseñar una estrategia para dividir el problema en versiones más pequeñas de sí mismo mientras avanza hacia el caso base.
- Combinar las soluciones de los problemas más pequeños de tal manera que se resuelva el problema más grande.

Ciclos Infinitos

La recursividad es una alternativa al bucle. Al igual que con los bucles, la recursividad puede hacer que el programa se repita para siempre. Esto se evita con el **caso base**, que actúa como condición de parada.

RECURSIVIDAD LINEAL

Prueba para casos base. Comenzar probando un conjunto de casos base (*debe haber al menos uno*). Toda posible cadena de llamadas recursivas **debe** llegar finalmente a un caso base.

Recurrir una vez (lineal). Realizar una sola llamada recursiva – este paso puede implicar una prueba que decida cuál de varias posibles llamadas recursivas realizar – La llamada recursiva debería avanzar hacia un caso base.

Ejemplo. Suma de elementos en un Arreglo

Escribiremos, en primer lugar, un algoritmo iterativo para sumar los elementos en un arreglo A de tamaño n: El siguiente algoritmo debería retornar 20 en la salida (4+3+6+2+5=20)

¿Cuál es su rendimiento? ¿Cómo se puede transformar esto en un algoritmo recursivo?

1. Algoritmo SumaLineal iterativo

Veamos el siguiente pseudocódigo correspondiente a un algoritmo iterativo de suma lineal

```
Algorithm IterativeSum(A):
Input: An integer array A
Output: The sum of the integers in A
n ← A.length
sum ← 0
for i ← 0 to n – 1 do
  sum ← sum +A[i]
return sum
```

¿Cuál es el rendimiento de IterativeSum?

¿Espacio? La complejidad espacial, o bien, el espacio que ocupa nuestro algoritmo está dado por el espacio que ocupa nuestro input. En este caso, tenemos que ver cuántas variables utilizamos, que sería un número constante de variables, las cuales a su vez, ocupan un número constante de bits. Por tanto, el espacio que ocupa nuestro algoritmo es constante, $O(1)$ (el puntero en el arreglo y las variables n y suma).

¿Tiempo de ejecución? La complejidad temporal de este algoritmo es lineal, $O(n)$

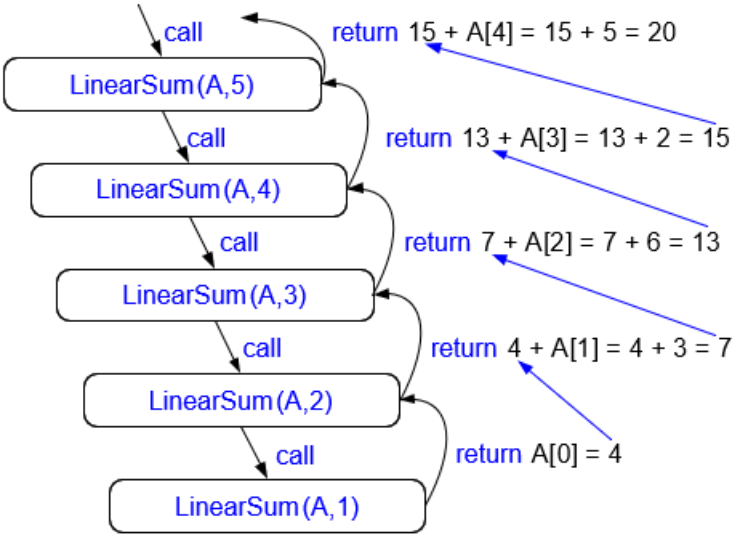
1. Algoritmo SumaLineal recursivo

Otra forma de implementar el algoritmo anterior es por medio de la recursividad, en donde podemos implementar el siguiente algoritmo y a su vez, cómo funciona éste

```
Algorithm LinearSum(A, n):
Input:
  A integer array A and an integer n = 1,
    such that A has at least n elements
Output:
  The sum of the first n integers in A
if n = 1 then
  return A[0]
else
  return LinearSum(A, n - 1) + A[n - 1]
```

Podemos notar que este algoritmo no toma tiempo constante, puesto que a medida que va creciendo, éste irá tomando cada vez más tiempo. Por tanto, la metodología que utilizábamos para analizar algoritmos iterativos no nos sirve para analizar nuestros algoritmos recursivos.

Para analizar nuestros algoritmos recursivos vamos a contar los contextos (las llamadas a funciones que metemos en nuestro stack), esto lo haremos tanto para analizar el tiempo como para analizar el espacio



Para analizar el tiempo, lo que hay que contar es cuántas veces se llama a la función recursiva, es decir, cuántos contextos nuevos se introducen en el stack. En este caso, se hace una llamada con n, luego con $n-1$ hasta llegar a $n=1$, por ende metemos n contextos en el stack o hacemos n llamadas recursivas. Por tanto, el tiempo de ejecución de este algoritmo recursivo va a ser **lineal** .

Ahora bien, el espacio es lo máximo que llega a crecer el stack, en este caso lo máximo que llega a crecer es el número de llamadas recursivas. Por ende, tanto el tiempo como el espacio (**en este caso**) son lineales

Definición de argumentos a favor de la recursividad

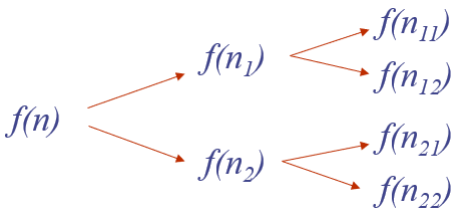
Al crear métodos recursivos, es importante definir los métodos de manera que faciliten la recursividad, esto a veces requiere que definamos parámetros adicionales que se pasan al método. Por ejemplo, definimos el método de inversión de matriz como `ReverseArray(A, i, j)` y no `ReverseArray(A)`.

RECURSIVIDAD BINARIA Y MÚLTIPLE

Caso Base. Comience probando un conjunto de casos base
Caso no base.

Caso binario: dos llamadas recursivas.

Caso múltiple: n llamadas recursivas.

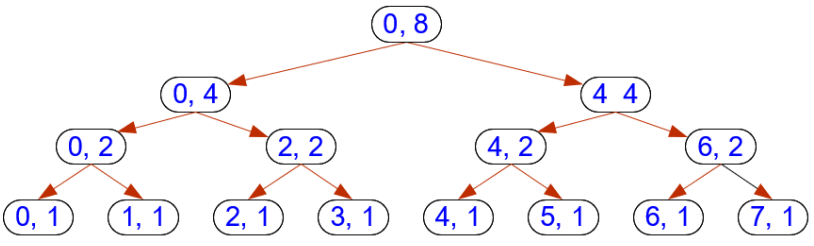


Este tipo de recursividad nos permite descomponer los problemas de diferentes formas, ya sea lineal, binaria o múltiple.

Ejemplo. Suma de todos los números en un arreglo A

```
Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
    if n = 1 then
        return A[i]
    return BinarySum(A, i, n/2) + BinarySum(A, i + n/2, n-n/2)
```

En recursividad binaria, tenemos un problema de Tenemos un problema de tamaño n, en recursividad binaria lo que haremos es descomponerlo en dos problemas cada uno con tamaño $n/2$, de modo que sumaremos los primeros $n/2$ elementos del arreglo y luego sumar los últimos $n/2$, combinando la solución final por medio de una suma. En la siguiente figura vemos el recorrido que hace este algoritmo



Ahora, cómo analizamos este árbol de recursividad para hallar el tiempo de ejecución y el espacio que ocupa nuestro algoritmo. Para ello, debemos contar cuántas llamadas recursivas hacemos, esto es

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{n} = n \times \sum_{i=0}^{\log(n)} \frac{1}{2^i}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \qquad c \neq 1$$

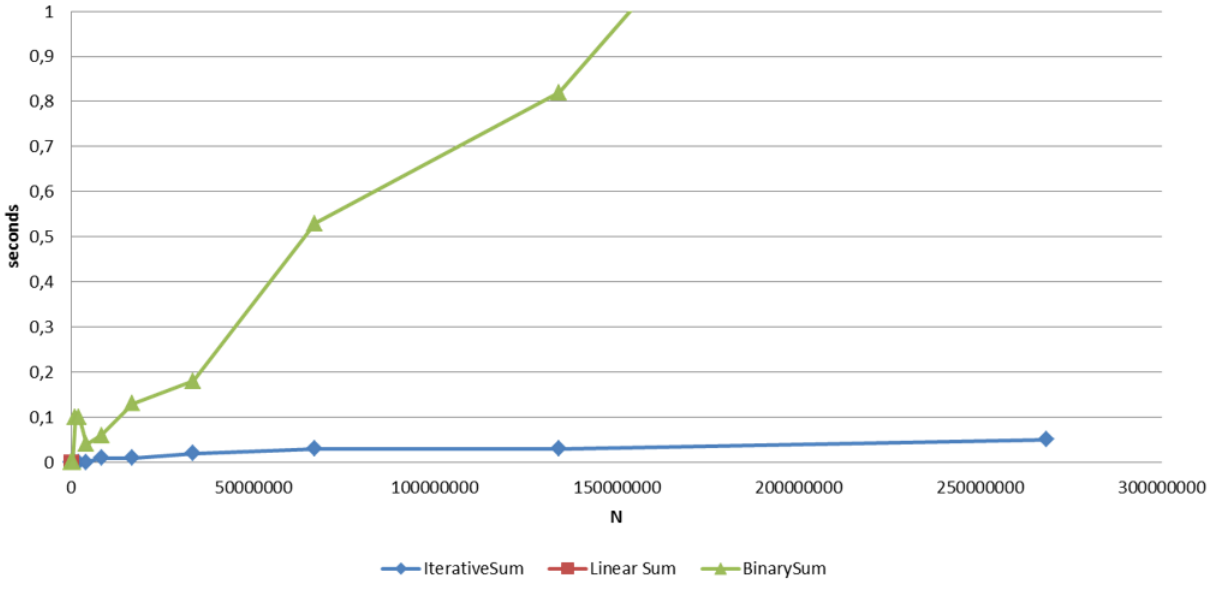
$$n \times \frac{(1/2)^{\log(n)+1} - 1}{1/2 - 1} = 2n - 1$$

Por tanto, el árbol de recursividad posee $2n - 1$ nodos, lo cual en notación asintótica corresponde a un número lineal de llamadas recursivas, por ende la complejidad temporal es lineal $O(n)$.

Tal como mencionamos anteriormente, a veces el número de llamadas recursivas es lo mismo al máximo que puede llegar a crecer el stack. Sin embargo, en este ejemplo no ocurre eso, dado cómo se ejecuta el algoritmo, puesto que nunca llegamos a tener todos los nodos simultáneamente en el stack, sino que se van liberando a medida que se van llamando.

Lo máximo que se puede tener simultáneamente es una rama completa del árbol, el cual posee una altura máxima $\log(n)$, por ende la complejidad espacial es logarítmica $O(\log n)$.

Si comparamos los tres algoritmos analizados (iterativa, lineal y binaria), notamos que de las tres la que es más eficiente es el algoritmo iterativo, puesto que su complejidad temporal es lineal y su complejidad espacial es constante. En la siguiente figura podemos apreciar un estudio experimental de las tres soluciones, donde la solución lineal apenas se ve dado que se queda sin memoria rápidamente, precisamente en $N = 16384$



1.4. Tipos Abstractos de Datos

1.4.1. *Tipos de Datos*

Escribimos datos, los clasificamos en varias categorías, como int, bool, string, float. Un data type representa un conjunto de valores posibles, como {. . ., -2, -1, 0, 1, 2,. . . } o {true, false}.
Al escribir nuestras variables, permitimos que la computadora encuentre algunos de nuestros errores.

Los tipos de datos son caracterizados por:

- un conjunto de valores
- una representación de datos, que es común a todos estos valores
- un conjunto de operaciones, que se pueden aplicar de manera uniforme a todos estos valores.

TIPOS PRIMITIVOS

Tipos primitivos comunes:

- bool
- char, byte, short, int, long
- float, double

No hay nada que el programador pueda hacer para cambiar nada sobre ellos, solo se puede trabajar con ellos sabiendo qué valores entran, cómo se representan y qué operaciones están permitidas.

Tipo	Valores	Representación	Operaciones
boolean	Verdadero, falso	Un solo byte/bit	&&, ||, !
char, byte, short, int, long	Enteros de diferentes tamaños	Complemento a dos	$+, -, \times, /$, otros
float, double	Números de decimales de diferentes tamaños y precisiones	Complemento a dos con exponente y mantisa	$+, -, \times, /$, otros

Cuadro 1.3: Valores, representación y operaciones permitidas de los Datos Primitivos.

CLASES EN C++/JAVA

Una clase es un tipo de dato. Dentro del contexto de programación orientado, dado un problema que queremos modelar lo primero que se hace es identificar **cuáles son los objetos que modelan ese dominio**. Las clases poseen propiedades/atributos y al interior de estas se crean **objetos** que nos entrega la representación de esas clases.
Los posibles **valores** de una clase se denominan objetos.

La **representación de datos** es una referencia (puntero) a un bloque de almacenamiento. La estructura de este bloque está definida por los campos/atributos (tanto heredados como inmediatos) de la clase.
Las **operaciones** sobre los objetos se denominan métodos.

Ejemplo. Clase Lista de la compra

¿Cuáles son los métodos que necesitamos?

- Constructor. Método que permite crear instancias de un objeto
- Insertar elemento. A medida que compramos nuevos objetos, los introducimos en la lista de la compra.
- Quitar elemento. Si nos equivocamos con un objeto o al momento de pagar nos arrepentimos, quitamos un elemento de la compra.
- ¿Esta vacío?. Es necesario conocer si la lista de compras está vacía o no.

Inserción en la lista de compras

Hay muchas formas de insertar un nuevo elemento en una lista:

- Como el nuevo primer elemento
- Como el nuevo último elemento
- Antes de un nodo dado
- Después de un nodo dado
- Antes de un valor dado
- Después de un valor dado
- Antes del enésimo elemento
- Después del enésimo elemento
- Antes del enésimo desde el final
- Después del enésimo desde el final
- En la ubicación correcta para mantener la lista en orden

¿Es necesaria toda esta funcionalidad para una lista de compras? ¿Es importante saber dónde y cómo se almacenan los artículos?
A la hora de diseñar las clases, debemos tener en cuenta que éstas serán utilizadas por un cliente, por ende es necesario preguntarnos *¿qué necesita saber ese cliente?* Dependiendo de la necesidad que requiera es la cantidad de información que verá y probablemente no sea todo, generalmente se le oculta varios detalles de la implementación.

Eficiencia

Una lista es solo una secuencia de valores; podría implementarse mediante una **lista enlazada** (la memoria está porcionada y se va enlazando) o mediante un **arreglo** (toda la memoria consecutiva). Dependiendo qué implementemos se tendrá diferente eficiecia.

- Insertar como un nuevo primer elemento es eficiente para una representación de lista enlazada, pero ineficiente para un arreglo.

- Acceder al enésimo elemento es eficiente para un arreglo, pero ineficiente para una lista enlazada (porque se tendrá que recorrer desde el principio).

- Insertar en la enésima posición no es eficiente para ninguno.

Según como decidamos implementar la lista de las compras, la eficiencia será mejor para algunas operaciones y peor para otras.
Se pueden distinguir dos niveles de abstracción:
¿Qué es y qué hace la lista?
¿Cómo se implementa?

El cliente de una clase:
Necesita saber qué es la lista y qué hace
No necesita conocer los detalles de la implementación
A lo más, necesita saber que la clase se implementa de manera eficiente.

Esto es lo va a definir el tipo abstracto de datos, es un nivel de abstracción superior con el que trabajarán los clientes

1.4.2. *Tipo de datos abstractos (ADT)*

Por lo general, es mucho más importante saber cómo se comporta una estructura de datos que cómo se representa o se implementa.

Un tipo definido en términos de su comportamiento en lugar de su representación se denomina tipo de datos abstractos (ADT). Un tipo de dato abstracto (ADT) es:

- Un conjunto de valores
- Un conjunto de operaciones, que se pueden aplicar de manera uniforme a todos estos valores

Abstraer es omitir información, manteniendo las partes más importantes. ¿Qué parte de un tipo de datos omite un ADT? Omitimos cómo está representado internamente el tipo abstracto de datos, cómo se implementa.

CONTRATOS ADT

Cada ADT debe tener un contrato (o especificación) que:

- Especifica el conjunto de valores válidos del ADT
- Especifica, para cada operación del ADT:
  - Su nombre
  - Sus tipos de parámetros
  - Su tipo de resultado, si lo hay
  - Su comportamiento observable
- No especifica:
  - La representación de datos, ni los algoritmos que implementen una determinada operación.

Los ADT están definidos por una interfaz

Simple. Ocultar la representación interna del cliente significa que hay menos detalles para que el cliente los comprenda.

Flexible. Dado que el ADT se define en términos de su comportamiento, el programador que implementa la biblioteca es libre de cambiar su representación subyacente.

Segura. El límite de la interfaz actúa como un muro que protege la implementación y al cliente entre sí.
Si un programa cliente tiene acceso a la representación, puede cambiar los valores en la estructura de datos subyacente de formas inesperadas.

Ejemplo. Lista de compras ADT

Una lista de compras permite: Agregar, eliminarlos y encontrar elementos.

```
/*
*Esta es una lista de compras donde puede agregar elementos y eliminarlos a
medida que compra (en JAVA)
*/
class ShoppingList {
   public:
       /** Inserta un elemento */
       void add(string s) ;
       /** Devuelve el elemento en el índice i, sin eliminarlo. */
       string remove(string s);
       /** Devuelve verdadero si s está en la lista. Falso de lo contrario*/
        bool find(string s);
       /** Devuelve el número de elementos de esta lista. */
       int size();
       /** Devuelve si la lista está vacía. */
       bool isEmpty();
};
```

Lista de compras contrato ADT (especificación alternativa)

Esta es una lista de compras donde puede agregar elementos y eliminarlos a medida que compra.

ADT asociados: `bool, string, int`
Operaciones: (en notación de orientación de objetos con L una lista de compras)
Constructores: `empty` → `ShopList`
Operaciones principales:

`L.add: string` →	`L.isEmpty` → `bool`
`L.remove: string` → `string`	`L.size` → `int`
`L.find: string` → `bool`	`L.print` →

Requisitos para cada lista L y cadena s:

`(L.add(s)).find(s) = true`
`(L.remove(s)).find(s) = false`
`(L.add(s)).remove(s)` contiene los mismos elementos que L.
`L←empty();`
`L.isEmpty() = true`
`L.size() = 0`
`(L.add(s)).isEmpty() = false`
`L.size() + 1 = (L.add(s)).size()`
Excepciones:
`L←empty(); L.remove(s)` no es válido.

Importancia del Contrato

Un contrato es un acuerdo entre dos partes; en este caso
• El implementador del ADT, quien se preocupa por que las operaciones sean correctas y eficientes.
• El programador de aplicaciones, que solo quiere usar el ADT.
No importa si uno desempeña las dos partes; el contrato sigue siendo esencial para un buen código, dadas las ventajas que hay *sencillez, flexibilidad y seguridad.*

AL PROGRAMAR

Idealmente, el programa debería estar separado en:
Interfaz: define el ADT
Implementación: que especifica el tipo de datos definido en el ADT
Cliente: un programa que usa el tipo de datos

En C

avg.c

```
#include <stdliO.h>
#include <math.h>
typedef int numType;
numType randNum(){  return rand(); }
int main(int argc, char *argv[])
{
  int i, N = atoi(argv[1]);
  float m1 = 0.0, m2 = 0.0;
  numType x;
  for (i = 0; i < N; i++)
  {
    x = randNum();
    m1 += ((float) x)/N;
    m2 += ((float) x*x)/N;
  }
  printf(" Average: %f\n", m1);
  printf("Std. deviation: %f\n", sqrt(m2-
m1*m1));
  return (0);
}
```

Interface

Num.h

```
typedef int Number;
Number randNum( );
```

Implementation

Num.c

```
#include <stdlib.h>
#include "Num.h"
Number randNum() { return rand();}
```

Client

avg.c

```
#include <stdio.h>
#include <math.h>
#include "Num.h"
int main(int argc, char *argv[])
{
  int i, N = atoi(argv[1]);
  float m1 = 0.0, m2 = 0.0;
  Number x;
  for (i = 0; i < N; i++)
  {
    x = randNum();
    m1 += ((float) x)/N;
    m2 += ((float) x*x)/N;
  }
  printf(" Average: %f\n", m1);
  printf("Std. deviation: %f\n", sqrt(m2-
m1*m1));
  return (0);
}
```

Interface

```
public interface Stack<E> {
          public E pop();
       public void push(E element);
}
```

Implementation

```
public class ArrayStack<E> implements Stack<E> {
     protected E S[];
     protected int top=-1;
     protected int capacity;
     public ArrayStack(int cap){
          capacity=cap;
          S=(E[])new Object[cap];
     }
     public E pop() {
          E element;
          if(top>=0){
               S[top--]=null;
               return element;}
     }
      public void push(E element) {S[++top]=element;}
}
```

Client

```
public static void main(String[] args){
     Object o;
     Stack<Integer> A = new
ArrayStack<Integer>(30);
     A.push(7);
     A.push(8);
     o=A.pop();
}
```

En Java

Interface

```
class stack {
public:
     virtual int pop() = 0;
     virtual void push(int element) = 0;
};
```

Implementation

```
#include "stack.h"
class array_stack : stack {
     public:
          array_stack();
          ~array_stack();
          int pop();
          void push(int element);
     private:
          int* container;
          int top;
          int len;
};
…
int array_stack::pop() {
     if(top>=0) return container[top--];
     return -1;
}
```

Client

```
int main() {
     stack s = array_stack();
     s.push(1);
     s.push(2);
     cout << s.pop() << endl;
     cout << s.pop() << endl;
}
```

En C++

Esta es una pila de entradas, las plantillas se pueden usar para generalizar esta implementación.

Ejemplo. Punto ADT con coordenadas x e y

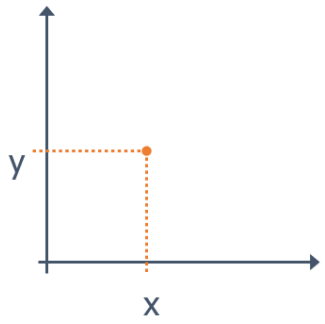
```
public interface 2DPoint {
// Return the x-coordinate of this point.
public double getX();
// Modifies the x-coordinate of this point.
public void setX();
// Return the y-coordinate of this point.
public double getY();
// Modifies the y-coordinate of this point.
public void setY();
// Print string of form (x,y)
public void print(); }
```

¿Cómo implementamos el Punto ADT? Para ello necesitamos

- **Una representación de datos**. La cual debe poder representar todos los valores posibles del ADT y debe ser privado.
- **Un algoritmo para cada una de las posibles operaciones**. Este debe ser coherente con la representación elegida, todas las operaciones auxiliares (ayudantes) que no están en el contrato deben ser privadas.

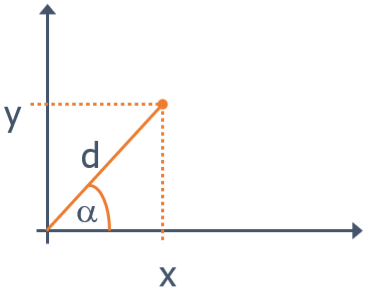
Implementaciones

Cartesianas




```
class PointCoords implements 2DPoint{
    private double x, y;
    public PointCoords(double x, double y){this.x = x; this.y = y; }
    public double getX() { return x; }
    public setX(double x) { this.x = x; }
    public double getY() { return y; }
    public setY(double y) { this.y = y; }
    public void print() {
        System.out.println("("+this.getX()+","+this.getY()+")");}
```

Polares




```
class PointAngle implements 2DPoint {
   private double angle, d;
   public PointAngle(double x, double y){this.convert(x,y);}
   public double getX() { return d*Math.cos(angle); }
   public setX(double x) { this.convert(x,this.getY());}
   public double getY() { return d*Math.sin(angle); }
   public setY(double y) { this.convert(this.getX(),y)}
   void print() {
      System.out.println ("("+this.getX()+","+this.getY()+")");}
   private void convert(double x, double y){
      this.angle = Math.atan(y/x);
      this.d = Math.sqrt(Math.pow(x,2)+Math.pow(y,2)); }
}
```

Clientes de Punto ADT

```
public interface 2DPoint {
  // Return the x-coordinate of this point.
  public double getX();
  // Modifies the x-coordinate of this point.
  public void setX();
  // Return the y-coordinate of this point.
  public double getY();
  // Modifies the y-coordinate of this point.
  public void setY();
  // Print string of form (x,y)
  public void print();
  }
```

Cliente 2

```
public myFigure{
  …
  public void foo(){
    Point2D p0= new PointCoords(0,0);
    p0.print();
    p0.setX(1);
    p0.print();
    p0.setY(2);
     System.out.println("x:"+p0.getX());
  }
```

Cliente 1

```
public geometryComparator {
  …
  // Returns true if the two given points are the same
  public boolean comparePoint(2DPoint p1, 2DPoint p2){
     return (p1.getX()==p2.getX() && p1.getY()==p2.getY())}
  …
  }
```

¿Qué pasa si la representación de los datos es pública?

```
class PointCoords implements 2DPoint{
    private public double x, y;
    public PointCoords(double x, double y){this.x = x; this.y = y; }
    public double getX() { return x; }
    public setX(double x) { this.x = x; }
    public double getY() { return y; }
    public setY(double y) { this.y = y; }
    public void print() {System.out.println("("+this.getX()+","+this.getY()+")");}
}
```

Client 2

```
public myFigure{
  …
  public void foo(){
    Point2D p0= new PointCoords(0,0);
  p0.print();              p0.setX(1);
  p0.print(); p0.setY(2);
     System.out.println("x:"+p0.getX());
  }
```

Client 3

```
public myFigure{
  …
  public void bar(){
    Point2D p0= new PointCoords(0,0);
  p0.print();
    p0.x=1; p0.print(); p0.y=2;
     System.out.println("x:"+p0.getX());
  }
```

Los métodos `foo()` y `bar()` hacen lo mismo.
¿Qué sucede si el implementador de la clase Point se da cuenta de que la implementación se puede mejorar cambiando la representación? `doble ángulo, d`

Hay que ser estrictos como se aplica el diseño por contrato. Las propiedades siempre deben ser privadas.

CONTRATO E IMPLEMENTACIÓN DE ADT

Crea una interfaz y una clase
- El contrato ADT es la interfaz
- La implementación es la clase

Cree una interfaz bien documentada para que la documentación también se pueda utilizar como contrato de ADT.

La documentación debe describir **campos públicos y encabezados de métodos**

La clase también necesita documentación para complementar las de la interfaz: **Rubros de constructores**

Responsabilidades del Implementador de la Clase

Una clase es responsable de sus propios valores. Debe protegerlos de usuarios descuidados o malintencionados.

Idealmente, una clase debe estar escrita para que sea útil en general.
- El objetivo es hacer que la clase sea reutilizable.
- La clase no debe ser responsable de nada específico de la aplicación en la que se usa.

En la práctica, la mayoría de las clases son específicas de la aplicación.

Las clases de Java/STL están, en general, muy bien diseñadas.
- No se escribieron específicamente para su programa.
- Esfuércese por hacer que sus clases se parezcan más a las de Java/STL.

RESUMEN DE ADT

Un tipo de datos abstracto describe valores y operaciones, pero **no representaciones**.

Un ADT debe proteger sus datos y mantenerlos válidos.
Todos, o casi todos, los datos deben ser privados.
El acceso a los datos debe realizarse a través de getters y setters.

Un ADT debe proporcionar:
Un contrato
Un conjunto de operaciones necesario y suficiente

1.4.3. *Implementaciones*

ARREGLO

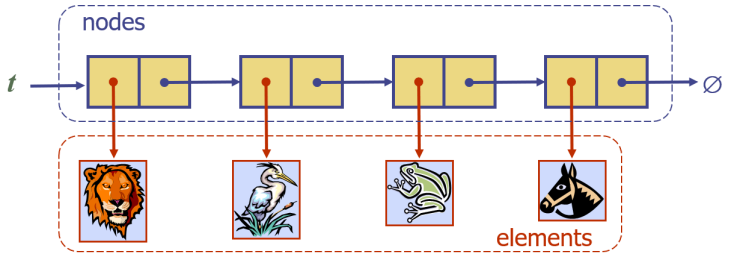
Representación de datos s y el tamaño de éste.

Dado un arreglo con n elementos, ¿cuál es el costo de
Insertar/quitar el primer elemento? $O(n)$
Insertar/quitar el último? $O(1)$
Insertar/quitar en el medio? $O(n)$
Manteniendo el orden de la matriz

LISTA VINCULADA

Representación de datos *t* y el tamaño de éste.
Se puede representar mediante un puntero al primer elemento y un número entero que almacena su tamaño

```
node head;
int size;
```




```
public class Node   {
    // Instance variables:
    private  Object element;
    private  Node next;
    /** Creates a node with null references to
  its element and next node. */
    public  Node() {
       this(null,  null);
    }
    /** Creates a node with the given element
  and next node. */
    public  Node(Object e,  Node n)  {
         element  =  e;
         next  =  n;
    }
```

```
// Accessor methods:
    public  Object getElement(){
       return  element; }
    public  Node getNext()  {
       return  next;  }
// Modifier methods:
    public void  setElement(Object newElem)
    {
         element  =  newElem;
    }
    public void  setNext(Node newNext)
    {
         next  =  newNext;
    }
}
```

La estructura del nodo (C)

```
typedef struct node
{
      int data;      // almacenerá información
      node *next;    // referencia al siguiente nodo
};
```

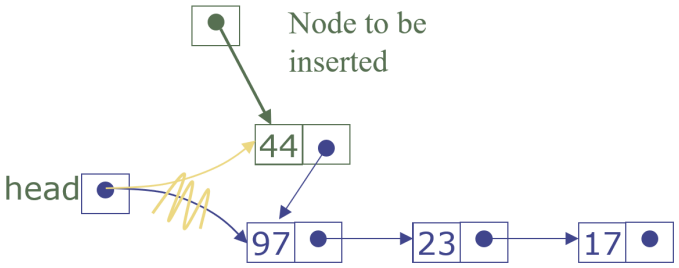
La clase nodo (C++)

```
class node {
public:
       node();
       node* getNext();
       int getElement();
private:
      int data;      // almacenerá información
      node *next;    // referencia al siguiente nodo
};
```

Dada una lista vinculada con n elementos, ¿cuál es el costo de
Insertar/quitar la cabeza,
Insertar/quitar la cola,
Insertar/eliminar después del nodo,
Etc.

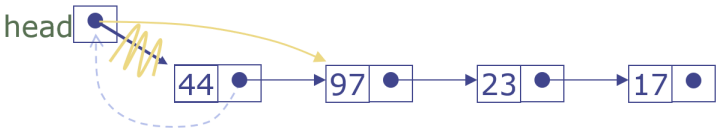
Insertar elementos en la cabeza

Para insertar un nodo:
- Cambiar el nodo que se insertará para que apunte al primer nodo.
- Cambiar el primero para apuntar al nuevo nodo



Quitar elementos en la cabeza

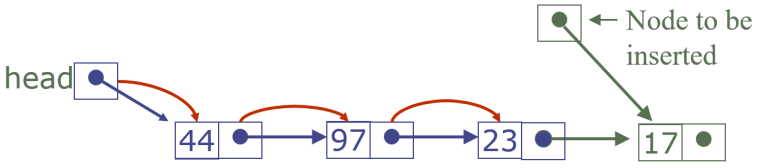
Para quitar un nodo:
- Copiar el puntero del primer nodo en el encabezado.



Insertar elementos en la cola

Para insertar un nodo:
- Encontrar el último nodo actual.
- Cambiarlo para que apunte al nuevo último nodo.

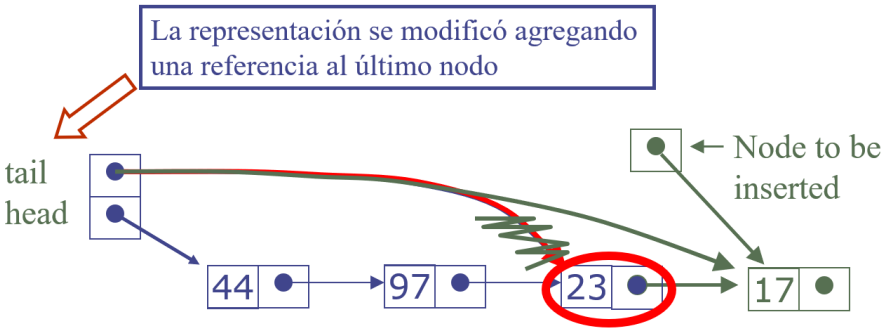
Corre en $O(n)$
Podemos hacerlo mejor que esto modificando la representación de la lista vinculada agregando un enlace al último elemento de la lista.



Inserción en la cola $O(1)$

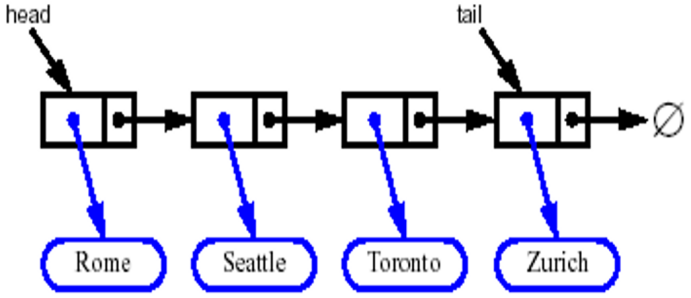
Para insertar un nodo:

- Encuentra el último nodo actual.
- Cámbielo para que apunte al nuevo último nodo.
- Cambiar el puntero de cola en el encabezado de la lista



Quitar elementos en la cola

¡Eliminar al final de una lista enlazada individualmente no es eficiente!
No hay una forma de tiempo constante para actualizar la cola para apuntar al nodo anterior.



1.5. Stacks

1.5.1. *El Stack (pila) ADT*

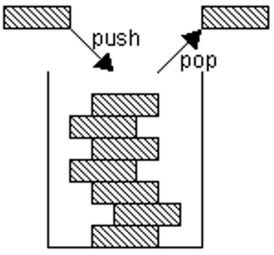
El Stack ADT almacena objetos arbitrarios. Las inserciones y eliminaciones siguen el esquema de *último en entrar, primero en salir* (LIFO).

Operaciones de la pila principal:

- push(object): inserta un elemento.
- object pop(): elimina y devuelve el último elemento insertado.

Operaciones auxiliares de stack:

- object top(): devuelve el último elemento insertado sin eliminarlo.
- integer size(): devuelve el número de elementos almacenados.
- boolean isEmpty(): indica si no se almacenan elementos.



ESPECIFICACIÓN ALGEBRAICA

Structure Stack (Element)

declare

```
new() → Stack
push(Stack; Element) → Stack
pop(Stack) → Element
top(Stack) → Element
isEmpty (Stack) → boolean
size(Stack) → integer
```

for all STACK s and Element e let

```
isEmpty (new()) ::= true
isEmpty (push(s; e)) ::= false
size(new()) ::= 0
size(push(s;e)) ::= size(s)+1
size(pop(s)) ::= size(s)-1
top(new()) ::= error
top(push(s;e)) ::= e
pop(new()) ::= new()
pop(push(s; e)) ::= s
```

C++ STL Stack

```
stack<int> myStack;
myStack.push(5);
cout << myStack.top() << endl;
cout << myStack.size() << endl;
myStack.pop();
cout << myStack.size() << endl;
```

INTERFAZ DE STACK EN JAVA

La interfaz en Java correspondiente a nuestro Stack ADT requiere la definición de la clase `EmptyStackException` diferente de la clase Java incorporada `java.util.Stack`

```
public interface Stack {

  public int size();

  public boolean isEmpty();

  public Object top()
      throws EmptyStackException;

  public void push(Object o);

  public Object pop()
       throws EmptyStackException;
}
```

Excepciones

Intentar la ejecución de una operación de ADT a veces puede causar una condición de error, llamada excepción. Se dice que las excepciones son "lanzadas"por una operación que no se puede ejecutar.

En Stack ADT, las operaciones pop y top no se pueden realizar si la pila está vacía. Intentar la ejecución de pop o top en una pila vacía arroja una `EmptyStackException`.

INTERFAZ DE STACK EN C++

```
class stack {
public:
    virtual void push(int val) = 0;
    virtual int pop() = 0;
    virtual bool isEmpty() = 0;
};
```

Se pueden usar plantillas para crear una pila genérica

PILA IMPLEMENTADA CON ARREGLO

Una forma sencilla de implementar Stack ADT es por medio de un arreglo.
- Agregamos elementos de izquierda a derecha.
- Una variable realiza un seguimiento del índice del elemento superior.

```
public class ArrayStackimplements Stack {
      Object S[ ];
      int t = -1;
      …
}
```


```
Algorithm size()
 return t + 1

Algorithm pop()
 if isEmpty() then
  throw EmptyStackException
  else
  t ← t – 1
  return S[t + 1]
```

```
Algorithm isEmpty()
 return (t = -1)
```

La matriz que almacena los elementos de la pila puede llenarse.
Una operación push arrojará un `FullStackException`, por ende hay una limitación de la implementación basada en arreglos.
¡No es intrínseco al Stack ADT!

```
Algorithm push(o)
 if t = S.length – 1 then
  throw FullStackException
  else
  t ← t + 1
  S[t] ← o
```

Rendimiento

- Sea n el número de elementos en la pila y N el tamaño del arreglo. El espacio utilizado es $O(N)$
- Cada operación se ejecuta en tiempo $O(1)$.

Limitaciones

- El tamaño máximo de la pila debe definirse a priori y no se puede cambiar.
- Intentar insertar un nuevo elemento en una pila completa provoca una excepción específica de la implementación.

ArrayStack en Java

```
public class ArrayStack
   implements Stack {

 // contiene los elementos de la pila
 private Object S[ ];

 // índice al element superior
 private int top = -1;

 // constructor
 public ArrayStack(int capacity) {
    S = new Object[capacity]);
  }
```

```
 public Object pop()
    throws EmptyStackException {
  if isEmpty()
    throw new EmptyStackException
      ("Empty stack: cannot pop");
  Object temp = S[top];
  // facilita la recolección de basura
  S[top] = null;
  top = top - 1;
  return temp;
 }
```

Nota: Casting al usar Objetos

Una ventaja de las pilas es que almacenan objetos genéricos.
Cuando hacemos pop() obtenemos un objeto que es una instancia de la clase objeto.
¿Es correcta esta afirmación?

```
Integer b = A.pop();
```

¡No! Necesitamos usar casting

```
Integer b = (Integer) (A.pop());
```

Tienes que estar seguro que en el tiempo de ejecución esto será cierto.

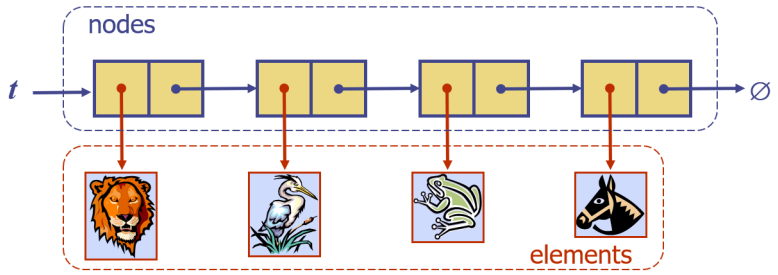
ArrayStack en C++

```
class array_stack : public stack {
public:
   array_stack(int inital);
  ~ array_stack();
  void push(int val);
  int pop();
  bool isEmpty();
private:
  int* container;
  int last;
};
```

```
array_stack:: array_stack(int i) {
       container = new int[i];
       last = -1;
}

pila_array::~pila_array() {
       delete [] container;
}

…

bool array_stack::isEmpty() {
       return last == -1;
}
```

APILAR CON UNA LISTA ENLAZADA INDIVIDUALMENTE

Podemos implementar una pila con una lista enlazada individualmente. El elemento superior se almacena en el primer nodo de la lista.
El espacio utilizado es $O(n)$ y cada operación del ADT en la pila toma tiempo $O(1)$.



Con una representación de lista vinculada, no se producirá una `FullStackException`. Puede obtener una excepción si agota la memoria, que es otro tipo de problema.

Cuando un nodo se elimina de una lista y el nodo hace referencia a un objeto, la referencia (el puntero en el nodo) no necesita establecerse en `null`. A diferencia de una implementación en un arreglo, realmente se elimina: ya no se puede acceder a ella desde la lista vinculada.

En Java, la recolección de basura puede ocurrir según corresponda. Sin embargo, en C++ necesitamos destruir el objeto

APLICACIONES DE STACK

Aplicaciones directas
- Historial de páginas visitadas en un navegador web
  - Deshacer secuencia en un editor de texto
  - Cadena de llamadas a métodos en la pila

Aplicaciones indirectas
- Estructura de datos auxiliares para algoritmos
  - Componente de otras estructuras de datos

Ejemplo 1. MÉTODO STACK EN JVM

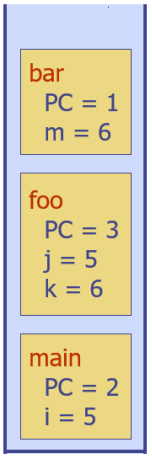
La máquina virtual Java (JVM) realiza un seguimiento de la cadena de métodos activos con una pila (stack). Cuando se llama a un método, la JVM empuja en la pila un marco que contiene:
- Variables locales y valor de retorno
- Contador de programa, seguimiento de la declaración que se está ejecutando.

Cuando un método finaliza, su marco se extrae de la pila y el control se pasa al método en la parte superior de la pila. Además, permite la **recursividad**.

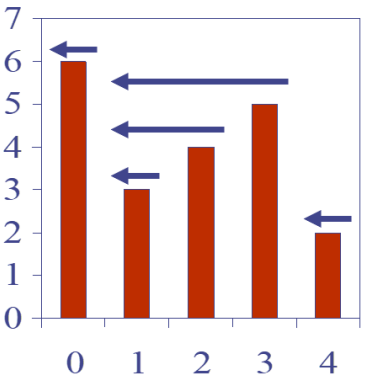
Cada "(", "{" o "[" debe estar emparejado con un ")", "}" o "]" que coincida

- correct: ()(()){([()])}
- incorrect: (((()(()){([()])}
- incorrect:)(()){([()])}
- incorrect: ({[])}
- incorrect: (

```
main() {
  int i = 5;
  foo(i);
  }
foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }
bar(int m) {
  …
  }
```



Ejemplo 2. COINCIDENCIA DE PARÉNTESIS



X	6	3	4	5	2
S	1	1	2	3	1