Tema 14

Introducción al análisis de algoritmos

> *Sin embargo, cuando ya llevaban corriendo una media hora o así, y estaban completamente secos otra vez, el Dodo dijo de repente en voz alta: «¡La carrera ha terminado!», y se agruparon todos a su alrededor, jadeando y preguntado: «Pero, ¿quién ha ganado?».*
> *El Dodo no podía contestar a esta pregunta sin meditarlo mucho antes, y permaneció largo rato con un dedo apretado en la frente (en la postura que normalmente veis a Shakespeare en los retratos), mientras el resto esperaba en silencio.*
>
> LEWIS CARROLL, *Alicia en el País de las Maravillas*.

Si tuvieses que escoger un programa entre varios que resuelven un mismo problema, ¿en función de qué escogerías?: ¿de su elegancia?, ¿de la legibilidad?, ¿del interfaz de usuario?, ¿de su velocidad de ejecución?, ¿de la memoria que consume? No cabe duda de que todos los factores influyen. Nosotros consideraremos aquí criterios basados en la *eficiencia*, es decir, en el mejor aprovechamiento de los recursos computacionales. Nuestro objeto de estudio serán los métodos de resolución de problemas, es decir, los *algoritmos*, y no los *programas*, o sea, sus implementaciones concretas usando diferentes lenguajes de programación.

Estudiaremos dos factores:

- El *coste* o *complejidad espacial*, es decir, la cantidad de memoria que consume.
- El *coste* o *complejidad temporal*, o sea, el tiempo que necesita para resolver un problema.

Ambos determinan el *coste* o *complejidad computacional*. No siempre coincidirán consumo espacial óptimo con mínimo coste temporal; es más, ambos factores entrarán, normalmente, en competencia. En ocasiones estaremos obligados, pues, a adoptar compromisos razonables entre ambos costes.

Centraremos el estudio, principalmente, en el análisis de la complejidad temporal de los algoritmos. Empezaremos proponiendo una aproximación *empírica* consistente en implementar diferentes programas que resuelven un mismo problema (con el mismo o diferentes algoritmos) y medir y comparar los tiempos de ejecución. Veremos que, en principio, resulta más determinante una correcta elección del *algoritmo* que los detalles de implementación o, incluso, que la elección de un lenguaje de programación frente a otro. La aproximación empírica supone un notable esfuerzo en tanto que obliga a implementar diferentes programas, prepararlos para hacer factible la medida de tiempos y realizar diferentes experimentos que nos permitan extraer conclusiones. Presentaremos una aproximación más *teórica* que permitirá arrojar luz sobre la eficiencia de los algoritmos sin necesidad de implementarlos y realizar experimentos.

14.1. Complejidad temporal: una aproximación empírica

La aproximación empírica se basa en la realización de estudios comparativos basados en la realización de experimentos de medición de tiempos de ejecución. Ilustraremos el problema de la medición de

tiempos con un ejemplo concreto. Mediremos tiempo de ejecución con varios programas que resuelven un mismo problema: la ordenación de un vector de enteros. Naturalmente, el tiempo de ejecución dependerá del tamaño del vector a ordenar, así que nuestro estudio considerará la dependencia del tiempo en función de dicho tamaño.

Empecemos presentando un programa Python que resuelve el problema mediante el método de la burbuja:

ordena_burbuja.py

```
from random import randrange

def rellena(talla, rango):
  valores = [0] * talla
  for i in range(talla):
    valores[i] = randrange(0, rango)
  return valores

def burbuja(valores):
  nuevo = valores[:]
  for i in range(len(nuevo)):
    for j in range(len(nuevo)-1-i):
      if nuevo[j] > nuevo[j+1]:
        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]
  return nuevo

# Programa principal
talla = 1000
rango = 100000

vector = rellena(talla, rango)

print "Vector desordenado:", vector
ordenado = burbuja(vector)
print "Vector ordenado:", ordenado
```

El programa define dos funciones: `rellena` y `burbuja`. La primera crea un vector con `talla` elementos enteros de valor aleatorio (la llamada `randrange(0, rango)` selecciona un entero al azar entre 0 y `rango` menos uno). La segunda función ordena y devuelve una copia del vector que se le suministra como parámetro.

El programa principal solicita la creación de un vector con 1000 enteros (con valores aleatorios entre 0 y 99999), muestra por pantalla el resultado, ordena el vector y vuelve a mostrar el resultado en pantalla. Aquí tienes un ejemplo (resumido) de la salida por pantalla que produce una ejecución del programa:

```
$ python ordena_burbuja.py
Vector desordenado: [41689, 56468, 40098, 49718, 72475, ..., 97318, 10805, 55123]
Vector ordenado: [9, 14, 37, 38, 44, 48, 71, 95, 56782, ..., 99957, 99976, 99983]
```

¿Cómo podemos medir el tiempo de ejecución de la función `burbuja`? Una posibilidad es usar un cronómetro de mano, pulsar el botón de inicio justo en el instante en que pulsamos la tecla de retorno de carro y pulsar el botón de parada cuando se imprime el resultado de ordenar el vector. Mmmm. Pero así estaríamos midiendo el tiempo *total* de ejecución del programa. ¿Cómo medir únicamente el tiempo de `burbuja`? ¡Pulsando los botones de arranque y parada del cronómetro justo después de que salga el mensaje que empieza con «`Vector desordenado`» y en el preciso instante en que aparece el texto «`Vector ordenado`». ¿Podemos hacerlo? Ten en cuenta que el tiempo de ejecución puede rondar las ¡milésimas o centésimas de segundo! Tranquilo, no hace falta tener unos reflejos prodigiosos para medir el tiempo de ejecución de una función.

14.1.1. Medida de tiempo de ejecución en Python: la función `clock`

El módulo `time` ofrece una función para la medición de tiempos. Su nombre es `clock` y devuelve el número de segundos que ha dedicado la CPU a la ejecución programa hasta el punto en que se efectúa la llamada a la función. El valor devuelto es un flotante. Aquí tienes una versión del programa `ordena_burbuja.py` que mide únicamente el tiempo de ejecución de la ordenación por burbuja.

ordena_burbuja.py

```
from random import randrange
from time import clock

def rellena(talla, rango):
  valores = [0] * talla
  for i in range(talla):
    valores[i] = randrange(0, rango)
  return valores

def burbuja(valores):
  nuevo = valores[:]
  for i in range(len(nuevo)):
    for j in range(len(nuevo)-1-i):
      if nuevo[j] > nuevo[j+1]:
        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]

# Programa principal
talla = 1000
rango = 100000

vector = rellena(talla, rango)

print "Vector desordenado:", vector
tiempo_inicial = clock()
ordenado = burbuja(vector)
tiempo_final = clock()
print "Vector ordenado:", ordenado

print "Tiempo de ejecución de burbuja: %f" % (tiempo_final - tiempo_inicial)
```

Aquí tienes el resultado de ejecutar el programa:

```
$ python ordena_burbuja.py
Antes de ordenar: [41689, 56468, 40098, 49718, 72475, ..., 97318, 10805, 55123]
Después de ordenar: [9, 14, 37, 38, 44, 48, 71, 95, ..., 99957, 99976, 99983]
Tiempo de ejecución de burbuja: 0.660000
```

La ejecución del método de la burbuja ha requerido 66 centésimas[1]. La resolución del reloj que consulta `clock` es de una centésima, así que no cabe esperar obtener más de dos decimales en nuestras medidas.

Debes tener en cuenta que la medida de `clock` se puede ver afectada por cierto grado de error. Si volvemos a ejecutar el programa, obtenemos una medida ligeramente diferente:

```
$ python ordena_burbuja.py
Antes de ordenar: [41689, 56468, 40098, 49718, 72475, ..., 97318, 10805, 55123]
```

[1] Esta medida se ha realizado ejecutando el programa en un ordenador Pentium IV Xeon a 2.0 GHz, dual, con 2 gigabytes de memoria RAM, corriendo bajo el sistema operativo Linux con núcleo 2.4.19, con la versión 2.2.1 del intérprete Python compilado con la versión 3.2 de `gcc`... Es posible que obtengas resultados muy diferentes si haces el experimento en tu ordenador y/o con un sistema operativo diferente y/o otra versión del intérprete Python... O sea, que el valor de tiempo obtenido significa, por sí mismo, más bien poco. Tendrá interés, en todo caso, cuando deseemos comparar la eficiencia de dos programas diferentes para efectuar un estudio comparativo.

```
Después de ordenar: [9, 14, 37, 38, 44, 48, 71, 95, ..., 99957, 99976, 99983]
Tiempo de ejecución de burbuja: 0.690000
```

Aquí tienes las medidas obtenidas tras 10 ejecuciones:

0.66 0.69 0.69 0.70 0.67 0.70 0.70 0.68 0.69 0.68

Tendremos que recurrir a la estadística descriptiva para proporcionar información acerca del tiempo de ejecución. La media del tiempo de ejecución es de 0.686 segundos (y su desviación estándar, 0.0135).

14.1.2. Tiempo medio de ejecución

Mmmm. Como estamos programando, podemos automatizar el proceso de medición y obtención de estadísticas (eliminamos en ésta y posteriores versiones la impresión por pantalla de los vector desordenado y ordenado, pues no influye en la medición de tiempos y llena la pantalla de información que no nos resulta útil):

ordena_burbuja.py

```
from random import randrange
from time import clock

def rellena(talla, rango):
  valores = [0] * talla
  for i in range(talla):
    valores[i] = randrange(0, rango)
  return valores

def burbuja(valores):
  nuevo = valores[:]
  for i in range(len(nuevo)):
    for j in range(len(nuevo)-1-i):
      if nuevo[j] > nuevo[j+1]:
        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]
  return nuevo

# Programa principal
talla = 1000
rango = 100000

repeticiones = 10

vector = rellena(talla, rango)

tiempo_inicial = clock()
for i in range(repeticiones):
  ordenado = burbuja(vector)
tiempo_final = clock()

print "Tiempo medio de ejecución de burbuja: %f" % \
  ((tiempo_final - tiempo_inicial) / repeticiones)
```

He aquí el resultado de ejecutar el programa:

```
$ python ordena_burbuja.py
Tiempo medio de ejecución de burbuja: 0.683000
```

Este método de efectuar medidas de tiempo repetidas adolece de un problema: ¿Cuántas veces hemos de repetir la ejecución para obtener una medida fiable? Diez repeticiones parecen suficientes para medir el tiempo necesario para ordenar 1000 enteros. ¿Cuántas hacen falta para medir el tiempo necesario para ordenar diez valores? ¿También diez?

ordena_burbuja.py

```
from random import randrange
from time import clock

def rellena(talla, rango):
  valores = [0] * talla
  for i in range(talla):
    valores[i] = randrange(0, rango)
  return valores

def burbuja(valores):
  nuevo = valores[:]
  for i in range(len(nuevo)):
    for j in range(len(nuevo)-1-i):
      if nuevo[j] > nuevo[j+1]:
        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]
  return nuevo

# Programa principal
talla = 10
rango = 100000

repeticiones = 10

vector = rellena(talla, rango)

tiempo_inicial = clock()
for i in range(repeticiones):
  ordenado = burbuja(vector)
tiempo_final = clock()

print "Tiempo medio de ejecución de burbuja: %f" % \
  ((tiempo_final - tiempo_inicial) / repeticiones)
```

Ejecutemos el programa:

```
$ python ordena_burbuja.py
Tiempo medio de ejecución de burbuja: 0.000000
```

¿Cero segundos? No es posible. Claro, la velocidad de ejecución es tan grande para vectores tan pequeños que diez repeticiones no son suficientes. Probemos con `repeticiones` igual a 100:

```
$ python ordena_burbuja.py
Tiempo medio de ejecución de burbuja: 0.000100
```

Ya se aprecia algo. Probemos con 1000 repeticiones:

```
$ python ordena_burbuja.py
Tiempo medio de ejecución de burbuja: 0.000090
```

Mejor, probemos con 10000:

```
$ python ordena_burbuja.py
Tiempo medio de ejecución de burbuja: 0.000087
```

Ahora tenemos un par de cifras significativas. Parece que ya empezamos a obtener medidas con una precisión suficiente.

El problema de fijar a priori el número de repeticiones estriba en que no es fácil determinar cuántas conviene efectuar para obtener una medida fiable para cada talla del vector. Si fijamos un número de repeticiones muy alto, nos curamos en salud y podemos medir razonablemente el tiempo de ejecución requerido para ordenar listas pequeñas; pero entonces puede que efectuar experimentos con listas grandes nos lleve (innecesariamente) una eternidad. Hay una aproximación alternativa que permite superar este problema: repetir la ejecución hasta que haya transcurrido una cantidad mínima de tiempo, por ejemplo, diez segundos. Este programa implementa esta idea:

ordena_burbuja.py

```
from random import randrange
from time import clock

def rellena(talla, rango):
  valores = [0] * talla
  for i in range(talla):
    valores[i] = randrange(0, rango)
  return valores

def burbuja(valores):
  nuevo = valores[:]
  for i in range(len(nuevo)):
    for j in range(len(nuevo)-1-i):
      if nuevo[j] > nuevo[j+1]:
        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]
  return nuevo

# Programa principal
talla = 10
rango = 100000

tiempo_minimo = 10
repeticiones = 0

vector = rellena(talla, rango)

tiempo_inicial = tiempo_final = clock()
while tiempo_final - tiempo_inicial < tiempo_minimo:
  ordenado = burbuja(vector)
  repeticiones += 1
  tiempo_final = clock()

print "Tiempo de ejecución de burbuja: %f" % \
  ((tiempo_final - tiempo_inicial) / repeticiones)
```

Este es el resultado de ejecutar el programa:

```
$ gcc ordena_burbuja.c -o ordena_burbuja
$ ./ordena_burbuja
Tiempo medio de ejecucion de burbuja: 0.000088
```

Ejercicios

▶ **331** El programa que acabamos de presentar efectúa muchas repeticiones del fragmento de programa que estamos estudiando cuando la talla del vector es muy pequeña. No ocurre lo mismo para vectores de talla grande, pues una sola repetición puede tardar más que el tiempo mínimo. El resultado es que obtenemos una única medida de tiempo, no un promedio de varias repeticiones.

Modifica el programa anterior para que, además de esperar un tiempo mínimo, efectúe siempre, al menos, un determinado número de repeticiones.

Podemos usar el mismo método para medir tiempos de ejecución con cualquier talla de vector. Para finalizar este apartado, te mostramos el resultado de efectuar la medida con este método para un valor de `talla` de 1000:

```
$ gcc ordena_burbuja.c -o ordena_burbuja
$ ./ordena_burbuja
Tiempo medio de ejecucion de burbuja: 0.675333
```

14.1.3. Medida de tiempo en C: la función `clock`

Disponemos en C de una herramienta homóloga a la función `clock` de Python. Es también una función y se llama también `clock` (disponible al incluir `time.h`). Este es su perfil:

```
int clock(void);
```

Su uso es ligeramente diferente al de la función `clock` de Python: La función `clock` de C no mide el tiempo en segundos, sino en unas unidades de tiempo propias. Un segundo tiene `CLOCKS_PER_SEC` unidades de tiempo, así que hemos de dividir por esta cantidad el resultado devuelto por `clock`.

Aquí tienes una versión C del programa Python `ordena_burbuja.py` que ordena un vector con 1000 elementos:

ordena_burbuja.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int * rellena(int talla, int rango)
{
  int * valores, i;

  valores = malloc(talla * sizeof(int));
  for (i = 0; i < talla; i++)
    valores[i] = rand() % rango;

  return valores;
}

int * burbuja(int valores[], int talla)
{
  int * nuevo, i, j, aux;

  nuevo = malloc(talla * sizeof(int));
  for (i = 0; i < talla; i++)
    nuevo[i] = valores[i];

  for (i = 0; i < talla; i++)
    for (j = 0; j < talla-1-i; j++)
      if (nuevo[j] > nuevo[j+1]) {
        aux = nuevo[j];
        nuevo[j] = nuevo[j+1];
        nuevo[j+1] = aux;
      }

  return nuevo;
}

```

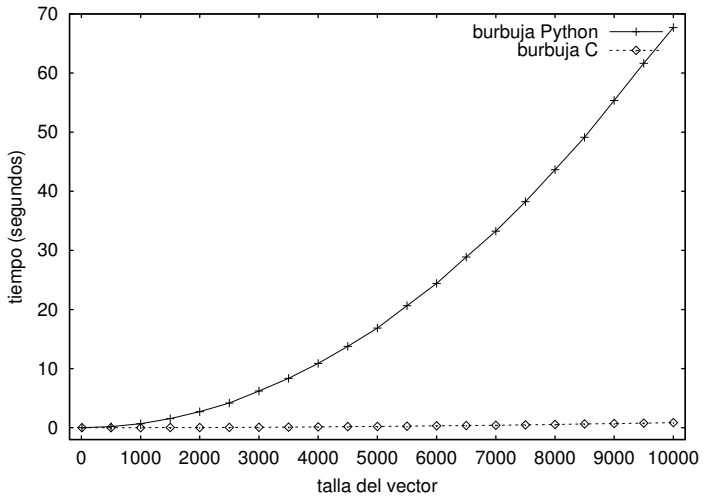
```
35 int main(void)
36 {
37   int * vector, * ordenado, talla, rango;
38   int tiempo_minimo, tiempo_inicial, tiempo_final;
39   int repeticiones;
40
41   talla = 1000;
42   rango = 100000;
43
44   tiempo_minimo = 10 * CLOCKS_PER_SEC;
45   repeticiones = 0;
46
47   vector = rellena(talla, rango);
48
49   tiempo_inicial = tiempo_final = clock();
50   while (tiempo_final - tiempo_inicial < tiempo_minimo) {
51       ordenado = burbuja(vector, talla);
52       free(ordenado);
53       repeticiones++;
54       tiempo_final = clock();
55   }
56   free(vector);
57   printf("Tiempo medio de ejecucion de burbuja: %f\n",
58          ((tiempo_final - tiempo_inicial) / (double)CLOCKS_PER_SEC / repeticiones));
59 }
```

Compilemos y ejecutemos el programa.

```
$ gcc ordena_burbuja.c -o ordena_burbuja
$ ./ordena_burbuja
Tiempo medio de ejecucion de burbuja: 0.008224
```

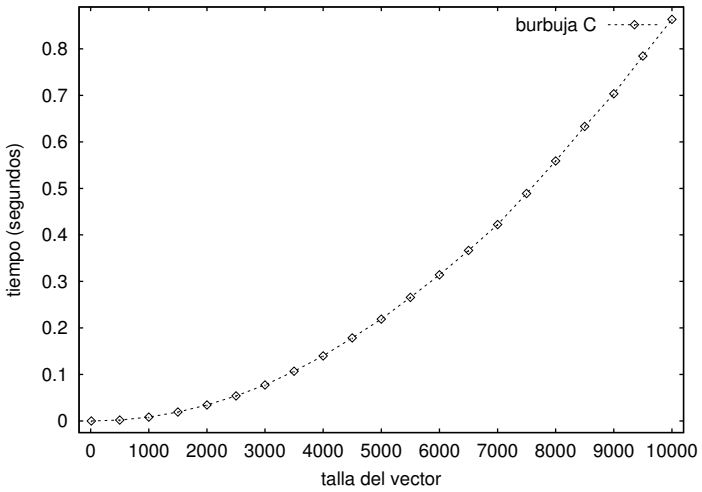
14.1.4. Python versus C

Comparemos los 0.008224 segundos que tarda el programa C con los 0.675333 del programa Python equivalente: ¡el programa C es unas 80 veces más rápido! Veamos, gráficamente, la diferencia de velocidad entre ambos programas midiendo tiempos de ejecución para varias tallas del vector de entrada:



Llama la atención que el tiempo de ejecución del programa Python no crezca *linealmente* con la longitud del vector a ordenar. La curva de crecimiento se asemeja a una parábola, no a una línea

recta. El tiempo medio de ejecución del programa C es tan pequeño en comparación con el del programa Python equivalente que parece una línea recta, pero no lo es. Mostremos únicamente la curva correspondiente al programa C para ver si su tendencia de crecimiento es similar a la del programa Python:



Efectivamente. Ambos programas requieren más tiempo para vectores mayores, pero no una cantidad de tiempo *proporcional* al tamaño del vector: parece que el tiempo necesario crece *cuadráticamente*.[2]

La conclusión de nuestro estudio es obvia, si nos preocupa la velocidad, Python está descartado. ¿Seguro? Fíjate en este otro programa que también ordena un vector de enteros, pero siguiendo una aproximación diferente: *mergesort*.

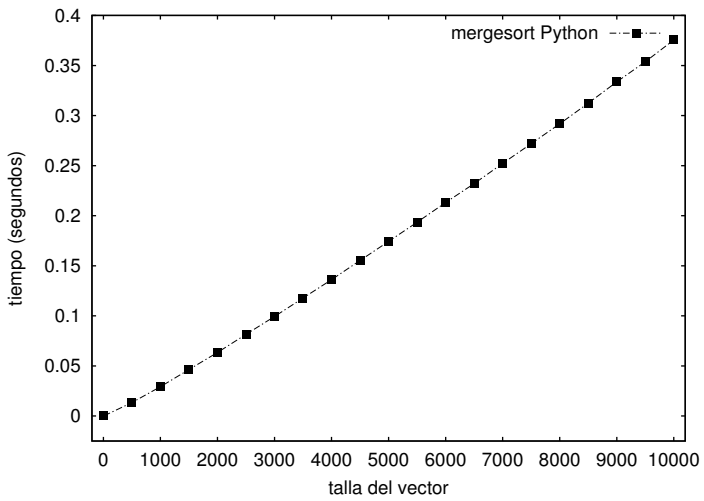
ordena_mergesort.py

```
1  from random import randrange
2  from time import clock
3  import sys
4
5  def rellena(talla, rango):
6    valores = [0] * talla
7    for i in range(talla):
8      valores[i] = randrange(0, rango)
9    return valores
10
11 def copia_mergesort(v):
12   copia = v[:]
13   mergesort(copia, 0, len(copia))
14   return copia
15
16 def mergesort(v, inicio, final): # ordena v[inicio:final]
17   global _aux
18
19   centro = (inicio+final)/2
20   if centro-inicio > 1:
21     mergesort(v, inicio, centro)
22   if final-centro > 1:
23     mergesort(v, centro, final)
24
25   # merge
26   for k in range(inicio, final):
27     _aux[k] = v[k]
28   i = k = inicio
29   j = centro
```

[2] Más adelante comprobaremos que esta impresión es cierta.

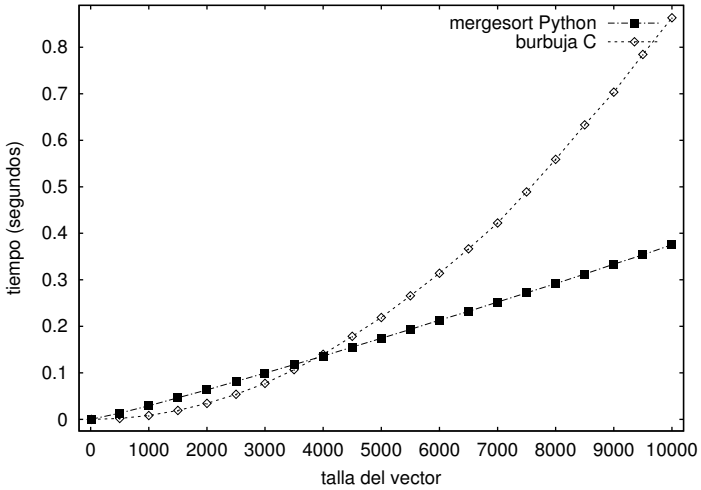
```
30
31    while i < centro and j < final:
32      if _aux[i] < _aux[j]:
33        v[k] = _aux[i]
34        i += 1
35      else:
36        v[k] = _aux[j]
37        j += 1
38      k += 1
39
40    while i < centro:
41      v[k] = _aux[i]
42      i += 1
43      k += 1
44
45    while j < final:
46      v[k] = _aux[j]
47      j += 1
48      k += 1
49
50
51  # Programa principal
52  talla = 1000
53  _aux = [0] * talla
54  rango = 100000
55
56  tiempo_minimo = 10
57  repeticiones = 0
58
59  vector = rellena(talla, rango)
60
61  tiempo_inicial = tiempo_final = clock()
62  while tiempo_final - tiempo_inicial < tiempo_minimo:
63    ordenado = copia_mergesort(vector)
64    repeticiones += 1
65    tiempo_final = clock()
66
67  print "Tiempo medio de ejecución de burbuja: %.12f" % \
68    ((tiempo_final - tiempo_inicial) / repeticiones)
```

Estudiemos la evolución del tiempo medio de ejecución de este programa. He aquí la gráfica que muestra el tiempo medio en función de la talla del vector:



Una primera observación que cabe hacer es que la curva se asemeja más a una línea recta que a una parábola[3]. La tendencia de crecimiento del tiempo de *mergesort* parece, pues, de naturaleza distinta a la que observamos al ordenar por el método de la burbuja (tanto en C como en Python).

Comparemos ahora la curva de tiempo de *mergesort* implementado en Python con la correspondiente curva del método de la burbuja implementado en C:



¡Sorpresa! El programa Python es más lento que el programa C para vectores de talla pequeña (hasta 4000 elementos), pero *mucho* más rápido que el programa C para vectores de talla grande. Es más, parece que conforme trabajemos con vectores más y más grandes, mayor será la diferencia de tiempos de ejecución a favor del programa Python.

Benchmarking: hecha la ley...

Estudiar la eficiencia temporal midiendo tiempos de ejecución es una técnica clásica para comparar las prestaciones de diferentes *ordenadores*. Existen conjuntos de programas destinados a facilitar la comparativa entre ordenadores: los llamados *benchmarks*. Los benchmarks tratan de caracterizar los «programas tipo» de ámbitos de aplicación concretos. Si, por ejemplo, queremos elegir el mejor ordenador para un centro de cálculo científico, mediremos tiempos con un benchmark centrado en la multiplicación de matrices, la resolución numérica de ecuaciones diferenciales, etc., pues es previsible que el ordenador se explote con ese tipo de programas. Si el ordenador se va a destinar a servir páginas web dinámicas, nuestro benchmark se centrará en peticiones de página, accesos a una base de datos, velocidad de acceso a información en el disco duro, etc. Y si se va a dedicar a aplicaciones lúdicas, los programas del benchmark generarán imágenes tridimensionales con millones de objetos, moverán grandes bloques gráficos por pantalla, etc.

Para facilitarnos el trabajo, algunas organizaciones y revistas publican periódicamente tablas con los tiempos de ejecución de un mismo benchmark sobre diferentes máquinas. Son, indudablemente, de gran ayuda a la hora de elegir la máquina adecuada. ¿Indudablemente? Los fabricantes de hardware tienen acceso a los benchmarks y mucho interés en demostrar que su ordenador es el más rápido... así que algunos optimizan el diseño de sus computadores para que batan las marcas de benchmark ¡analizando los programas que los forman! O sea, son capaces de asegurarse un excelente tiempo de ejecución para cierto programa de un benchmark que se supone mide la velocidad de cálculo para el producto de dos matrices aunque, en la práctica, el computador vaya bastante peor cuando se enfrenta a un problema similar o igual en condiciones ligeramente diferentes de las propias del benchmark.

Podemos extraer algunas conclusiones de nuestro estudio:

- No sólo el lenguaje de implementación influye en la mayor velocidad de un programa frente a otro cuando resolvemos un mismo problema: el *algoritmo* (en nuestro caso, el método de la burbuja frente a *mergesort*) es determinante.

[3] Aunque no es exactamente una línea recta. Más detalles dentro de poco.

- Parece que dos implementaciones de un mismo algoritmo presentan una tendencia similar, aunque las diferencias de velocidad sean notables. Las dos versiones del método de la burbuja, parecen crecer cuadráticamente, aunque estén implementadas con lenguajes muy distintos.
- No tiene mucho sentido decir si un algoritmo es más rápido que otro estudiando únicamente un problema de una tamaño determinado. Resulta más interesante estudiar el comportamiento del tiempo de ejecución como función del tamaño del vector.
- Las medidas de tiempo se toman sobre un ordenador concreto, controlado por un sistema operativo concreto, con un programa implementado por un programador concreto... Las medidas de tiempos no son muy informativas si no van acompañadas de una descripción completa del entorno de trabajo. Sirven, a lo sumo, para comparar dos o más programas ejecutados en las mismas condiciones.
- Para poder medir tiempo de ejecución, hemos de disponer de una implementación de cada programa y modificarla para que incorpore las acciones de medición de tiempo e impresión de resultados obtenidos. Es una obviedad, pero hemos de tenerla presente. Si queremos elegir entre 20 algoritmos diferentes, hemos de implementarlas todas y cada una de ellas, medir tiempos y efectuar el estudio comparativo. Es un esfuerzo notable.

¡Ah! Antes de acabar esta sección, hemos de dejar una cosa clara: más rápido que una implementación de *mergesort* en Python resultará, con toda probabilidad, una implementación de *mergesort* en C (muy mal programador hay que ser para que no sea así). El siguiente ejercicio te sugiere que lo compruebes tú mismo.

Ejercicios

▶ **332** Implementa el método *mergesort* en C. Mide el tiempo medio de ejecución para diferentes tallas del vector a ordenar y compara las medidas efectuadas con las del mismo método implementado en Python.

14.2. Complejidad temporal: hacia una aproximación teórica

Hemos visto, de momento, cómo efectuar experimentos para medir el tiempo. ¿Podemos prescindir de estos experimentos y seguir obteniendo resultados que nos permitan comparar dos (o más) programas o, mejor, dos (o más) algoritmos? Es decir, ¿es posible «calcular» o «estimar» el tiempo de ejecución a partir del código fuente, sin necesidad de implementar y ejecutar los programas?

14.2.1. Dependencia del tiempo con el coste de las operaciones elementales

Vamos a estudiar esta posibilidad considerando tres formas diferentes de solucionar un mismo problema en C: el cálculo del cuadrado del número 10 (aunque, ciertamente, no parece necesario programar un computador para ello). El primer programa sigue una aproximación directa y usa la operación producto para resolver el problema:

producto.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int m;
6
7    m = 10 * 10;
8    printf("%d\n", m);
9    return 0;
10 }
```

El segundo programa suma 10 veces el valor 10:

suma.c

```
#include <stdio.h>
int main(void)
{
  int m, i;

  m = 0;
  for (i=0; i<10; i++)
    m = m + 10;
  printf("%d\n", m);
  return 0;
}
```

Y el tercero repite 10 veces 10 incrementos unitarios de un contador:

incremento.c

```
#include <stdio.h>

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

  m = 0;
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      m++;
  printf("%d\n", m);
  return 0;
}
```

¿Cuál de los tres programas se ejecutará más rápidamente? A simple vista diríamos que el primero, pues ocupa menos líneas y resuelve «directamente» el problema. Pero hemos de pensar un poco la respuesta: eso será cierto si cuesta menos tiempo efectuar una multiplicación que 10 sumas o 100 incrementos. Normalmente, el producto es una operación más lenta que la suma, y ésta es más costosa que el incremento. Supón que *en nuestro ordenador* cada operación básica tarda lo que se indica en esta tabla ($1\,\mu s$ es una millonésima de segundo):[4]

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	$342\,\mu$s	$31\,\mu$s	$1\,\mu$s	$1\,\mu$s	$1\,\mu$s

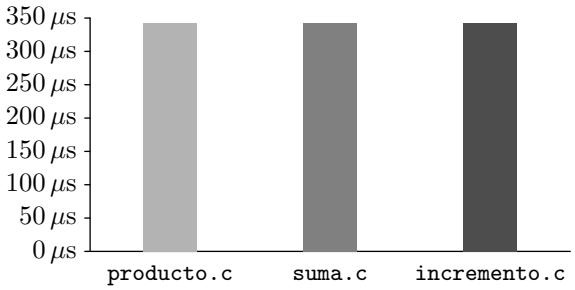
(No tendremos en cuenta el tiempo de impresión de resultados en aras de simplificar la exposición.) Cada programa efectúa un número diferente de productos, sumas e incrementos:

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones	Tiempo
`producto.c`	1			1		$343\,\mu$s
`suma.c`		10	10	12	11	$343\,\mu$s
`incremento.c`			210	12	121	$343\,\mu$s

Mmmm. Quizá necesitemos justificar algunos de estos valores. El número de incrementos en `sumas.c`, por ejemplo, es de 10 por que el bucle `for` hace que `i` pase de 0 a 10 con el operador de postincremento. Las 12 asignaciones del mismo programa corresponde a las sentencias «`m = 0`» y «`m = m + 10`» (esta última se ejecuta 10 veces) y a la inicialización del bucle `for`. El número de comparaciones es de 11 porque el bucle `for` efectúa una comparación para cada valor de `i` entre 0 y 10.

Nos precipitamos al juzgar como mejor a uno de ellos. Los tres son igual de rápidos.

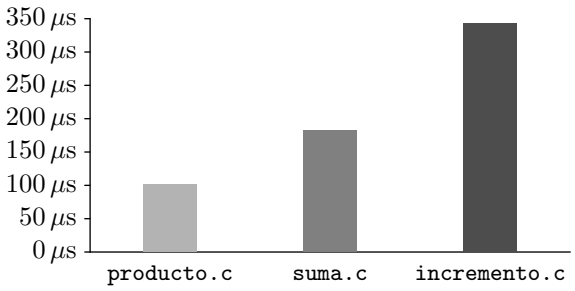
[4] No tenemos en cuenta el coste de `printf` o `return` porque sólo estamos interesados en estudiar el coste temporal del método de cálculo del producto de 10 por 10.



Bien. ¿Y si el coste de cada operación fuera diferente? Por ejemplo, el que se indica en esta tabla:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	100 μs	15 μs	1 μs	1 μs	1 μs

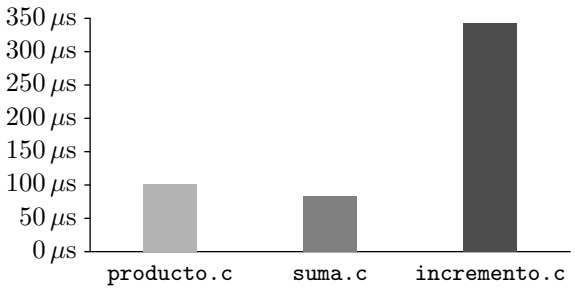
¿Qué programa sería más rápido en este otro escenario? En ese caso, `producto.c` sería el más rápido e `incremento.c` el más lento.



¿Y si sumar fuese más rápido? Pongamos por caso que los tiempos de ejecución de cada operación fueran éstos:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	100 μs	5 μs	1 μs	1 μs	1 μs

Entonces resultaría «vencedor» el programa que calcula 10^2 sumando (tardaría 83 μs):



No es tan fácil, pues, decidir qué programa es más rápido, al menos no si queremos tener en cuenta el coste de cada instrucción ya que éste depende del ordenador.

Los programas estudiados presentan una aplicación muy pobre: se limitan a resolver el problema del cálculo de 10 al cuadrado. Generalicémoslos para resolver el problema de calcular n^2, siendo n un entero cualquiera:

producto.c

```
#include <stdio.h>

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

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

suma.c

```
#include <stdio.h>

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

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

incremento.c

```
#include <stdio.h>

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

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

El cálculo de 10^2 es un caso particular del cálculo de n^2. Decimos que el caso $n = 10$ es una *instancia* del problema de calcular n^2. En principio, cuanto mayor es el valor de n, más costoso es resolver el problema. Por ejemplo, la instancia $n = 100$ es más difícil de resolver que la instancia $n = 10$. En este caso decimos, además, que n es el tamaño o *talla* del problema.

Podemos expresar el tiempo de ejecución de los tres programas como una función del valor de n, es decir, de la talla. Anotaremos al margen el número de operaciones que implica la ejecución de cada línea del programa:

producto.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4   int m, n;
5   scanf("%d", &n);
6   m = n * n;
7   printf("%d\n", m);
8   return 0;
9 }
```

operaciones
1 suma y 1 asignación

suma.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int m, n, i;
5    scanf("%d", &n);
6    m = 0;
7    for (i=0; i<n; i++)
8      m = m + n;
9    printf("%d\n", m);
10   return 0;
11 }
```

operaciones	veces
1 asignación	
1 asignación, $n+1$ comparaciones y n incrementos	
1 suma y 1 asignación	n veces

(Observa que detallamos por una parte el número de operaciones que aporta al coste total cada línea por sí misma y el número de veces que ésta se ejecuta.)

incremento.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int m, n, i, j;
5    scanf("%d", &n);
6    m = 0;
7    for (i=0; i<n; i++)
8      for (j=0; j<n; j++)
9        m++;
10   printf("%d\n", m);
11   return 0;
12 }
```

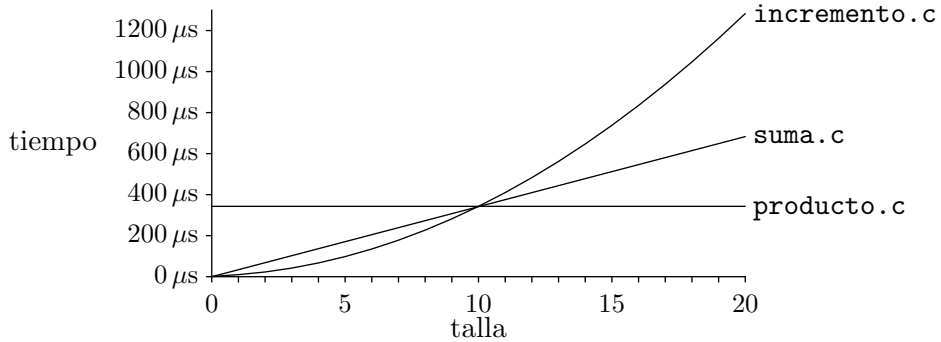
operaciones	veces
1 asignación	
1 asignación, $n+1$ comparaciones y n incrementos	
1 asignación, $n+1$ comparaciones y n incrementos	n veces
1 incremento	n^2 veces

Resumiendo:

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

Si representamos gráficamente la *evolución del coste temporal en función de la talla del problema*, nos haremos una idea de cuan bueno o malo es un algoritmo conforme se enfrenta a problemas más y más grandes.

En el supuesto de que un producto cueste $342\,\mu$s, una suma $31\mu s$ y $1\,\mu$s el resto de operaciones, la evolución del coste con la talla se puede representar gráficamente así:

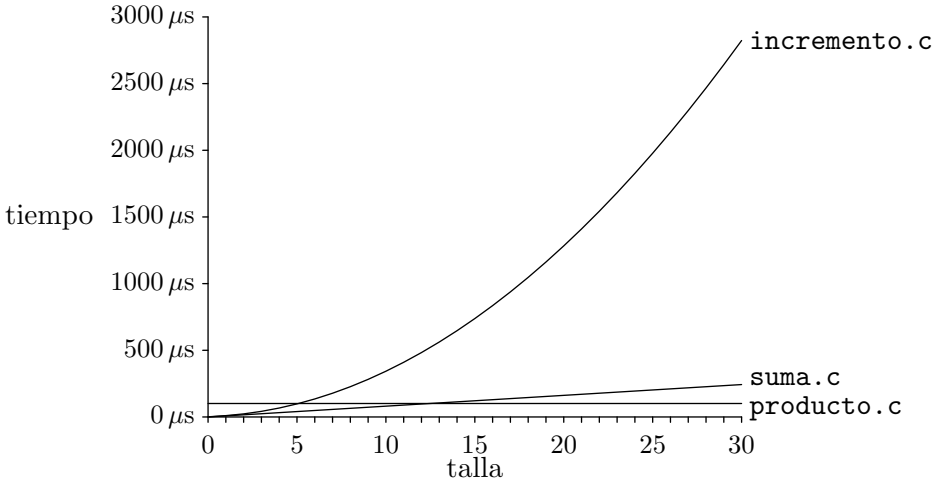


Queda patente que *para cada talla hay un programa más rápido, pero no siempre el mismo.* Hasta $n = 10$, el mejor es `incremento.c`, en $n = 10$, los tres son igual de rápidos, y a partir de $n = 10$ siempre es más rápido `producto.c`. Es más, cuanto mayor es n, mejor resulta `producto.c` y peor `incremento.c`.

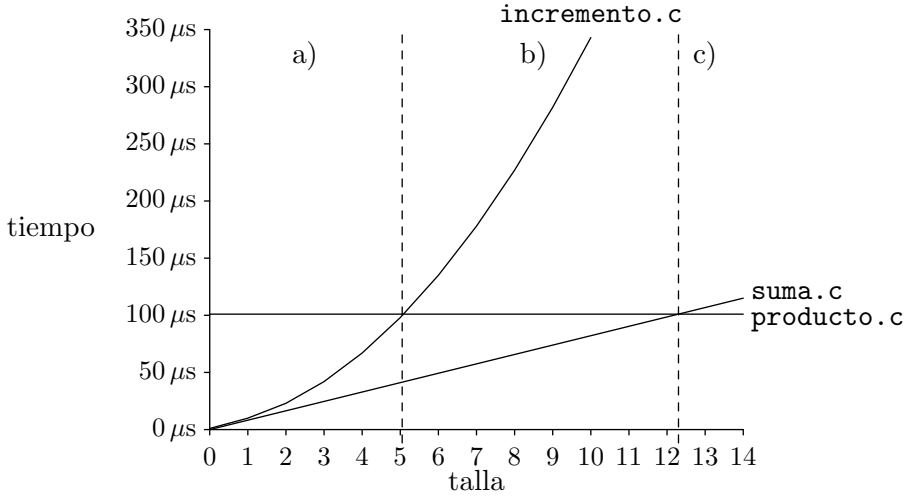
Vimos que había «configuraciones» de los tiempos de cada operación que hacían que `suma.c` fuera más rápido cuando `n` toma el valor 10. Consideremos, por ejemplo, la siguiente combinación:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	$100\,\mu$s	$5\,\mu$s	$1\,\mu$s	$1\,\mu$s	$1\,\mu$s

La gráfica de evolución del coste es ahora esta otra:



Ahora vemos que no es cierto que `suma.c` sea mejor que `producto.c` Al menos no siempre. De hecho, en diferentes tramos resultan «ganadores» diferentes programas:



a) `suma.c` mejor que `incremento.c` e `incremento.c` mejor que `producto.c`.

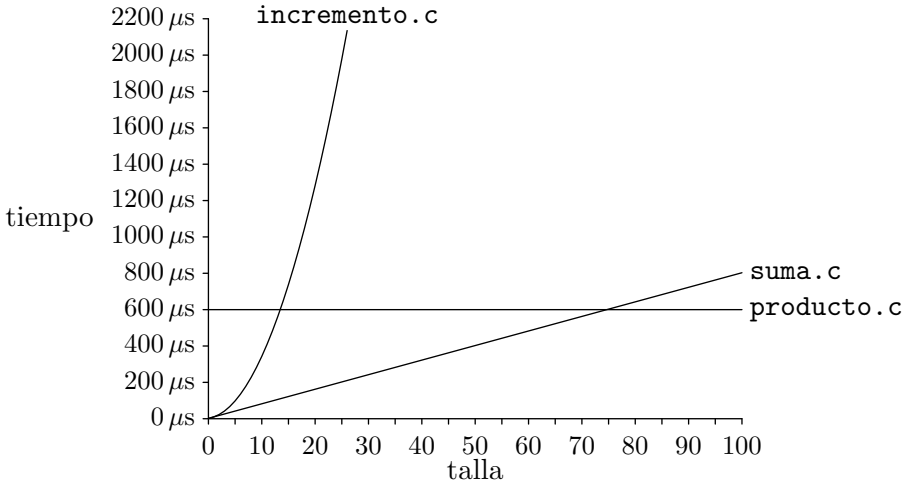
b) `suma.c` mejor que `producto.c` y `producto.c` mejor que `incremento.c`.

c) `producto.c` mejor que `suma.c` y `suma.c` mejor que `incremento.c`.

Fíjate en que a partir de $n = 13$, siempre resulta mejor `producto.c`.

Aunque el producto sea mucho más caro del que hemos supuesto, siempre habrá un valor de n a partir del cual `producto.c` es el programa más rápido: su «curva» es plana, no crece, mientras que las otras dos crecen (una más rápidamente que la otra, pero ambas crecen) así que siempre habrá un valor de n a partir del cual `producto.c` sea más rápido.

Pongamos que el producto es casi el doble de «caro», que cuesta, digamos, $600\,\mu$s:



Podemos decir algo similar al comparar `suma.c` con *incremento.c*: no importa cuan costosa sea la operación de suma, siempre habrá un valor de n a partir del cual `suma.c` es mejor que `incremento.c`.

O sea, *independientemente del coste de cada operación básica*, `producto.c` siempre acaba siendo mejor que los otros dos programas. Un método que tarda un *tiempo constante* siempre acaba siendo mejor que uno cuyo tiempo depende *linealmente* de la talla del problema. Y un método cuyo tiempo depende *linealmente* de la talla del problema, siempre llega un punto para el que es mejor que otro método cuyo tiempo de ejecución crece *cuadráticamente* con la talla del problema.

Decimos que el primer método es *asintóticamente* más eficiente que los otros, y que el segundo método es *asintóticamente* más eficiente que el tercero.

Ejercicios

► **333** Implementa los tres programas y mide tiempos de ejecución en un ordenador usando las técnicas de medida de tiempos que has aprendido. ¿Es cierto que, a la larga, siempre es mejor `producto.c`? ¿En qué rango de valores de n es más rápido cada uno de los programas en ese ordenador?

Una conclusión es, pues, que si queremos ver cómo evoluciona el coste con la talla, podemos hacer estudios asintóticos independientes del coste de cada operación.

Ya, pero el coste asintótico sólo nos dice qué programa es mejor cuando el problema es suficientemente grande. ¿Nos basta con eso? En principio, sí, por estas razones:

- Los programas son especialmente útiles para resolver problemas de gran talla.
- El estudio de costes asintóticos nos permite efectuar ciertas simplificaciones en los análisis que facilitan enormemente la comparación de métodos. La primera de dichas simplificaciones, que acabamos de justificar, es que no nos importa la velocidad con que se ejecuta cada operación elemental. Y hay más.

14.2.2. Independencia del coste de cada operación elemental: el concepto de paso

No nos interesa que nuestros estudios dependan del coste concreto de cada operación elemental. Bastará con que sepamos cuántas operaciones elementales ejecuta un programa y cómo depende ese número de la talla del problema a resolver. ¿Y qué será para nosotros una operación elemental, ahora que nos da igual lo que tarda cada operación? Responder a esta pregunta nos lleva a introducir el concepto de *paso*. Un paso es un segmento de código cuyo tiempo de proceso no depende de la talla del problema considerado y está acotado por alguna constante. Una suma es una operación elemental, y un producto, y un incrementos... Calcular la longitud de una cadena *no es una operación elemental*. ¿Por qué? Porque cuesta tanto más cuanto más larga es la cadena (recuerda que la longitud se determina recorriendo la cadena hasta encontrar un carácter `\0`).

Aquí tienes algunas operaciones que siempre consideramos pasos:

- operaciones aritméticas,
- operaciones lógicas,
- comparaciones entre escalares,
- accesos a variables escalares,
- accesos a elementos de vectores o matrices,
- asignaciones de valores a escalares,
- asignaciones de valores a elementos de vectores o matrices,
- lectura de un valor escalar,
- escritura de un valor escalar,
- ...

El coste de las operaciones elementales que no son pasos se expresará en función del número de pasos con el que podrían efectuarse. El coste de efectuar un corte de n elementos de una lista Python, por ejemplo, se expresará como un número de pasos proporcional n, pues podríamos efectuar esa operación con operaciones elementales copiando en una nueva lista cada uno de los elementos del corte. La copia se efectuaría, pues, con un número de operaciones elementales proporcional a n.

Definimos el *coste computacional temporal* de un programa como el número de pasos expresado en función de la talla del problema.

Contemos los pasos de cada programa. Empecemos por `producto.c`.

producto.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int m, n;
5    scanf("%d", &n);
6    m = n * n;                    1 paso
7    printf("%d\n", m);
8    return 0;
9  }
```

El número total de pasos es 1.

Vamos a analizar ahora `suma.c`.

suma.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int m, n, i;
5    scanf("%d", &n);
6    m = 0;                        1 paso
7    for (i=0; i<n; i++)           2n + 2 pasos
8      m = m + n;                  2 pasos,        n veces
9    printf("%d\n", m);
10   return 0;
11 }
```

O sea, un total de $4n + 3$ pasos.

Nos queda por analizar `incremento.c`

incremento.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int m, n, i, j;
5    scanf("%d", &n);
6    m = 0;                          1 paso
7    for (i=0; i<n; i++)             2n + 2 pasos
8      for (j=0; j<n; j++)           2n + 2 pasos,   n veces
9        m++;                        1 paso,         n^2 veces
10   printf("%d\n", m);
11   return 0;
12 }
```

Sumando tenemos un coste de $2n^2 + 4n + 4$ pasos.

En resumen:

	producto.c	suma.c	incremento.c
pasos	1	$4n + 3$	$2n^2 + 4n + 4$

El valor concreto de los factores de cada término en estas expresiones del número de pasos no importa. Fíjate en que podríamos haber decidido que, en nuestro ordenador, la sentencia `s += n` cuente como dos pasos (suma por un lado y asignación por otro) o como uno (suma y asignación). O sea, con la introducción del concepto de paso da igual decir que el coste de un algoritmo es $2n+3$ que $c_1 \cdot n + c_2$, siendo c_1 y c_2 constantes arbitrarias.

Cualquier secuencia de pasos cuya longitud no depende de la talla del problema cuenta como una cantidad constante de pasos.

Esta nueva simplificación ayuda enormemente a efectuar conteos de pasos. Podemos reescribir la tabla anterior así:

	producto.c	suma.c	incremento.c
pasos	c_0	$c_1 \cdot n + c_2$	$c_3 \cdot n^2 + c_4 \cdot n + c_5$

14.2.3. Independencia del lenguaje de programación y de los detalles de implementación

Una ventaja del tipo de simplificaciones que estamos adoptando es que realmente independizan nuestros cálculos del lenguaje de programación en el que se ha escrito un programa y de los detalles de implementación. Un ejemplo te ayudará a entenderlo. Vamos a analizar ahora el número de pasos que deben ejecutar dos programas equivalentes codificados en C y otro también equivalente pero escrito en Python.

Esta función C calcula el sumatorio de los n primeros números naturales:

```
int sumatorio(int n)
{
  int s, i;

  s = 0;
  for (i=1; i<=n; i++)
    s = s + i;
  return s;
}
```

¿Cuál es su coste temporal? En primer lugar hemos de plantearnos qué parámetro determina la talla del problema. Parece evidente: la talla es n. Desglosemos el análisis en las diferentes operaciones:

sumatorio1.c

```
1  int sumatorio(int n)
2  {
3    int s, i;
4    s = 0;                          1 paso
5    for (i=1; i<=n; i++)            2n + 2 pasos
6      s = s + i;                    2 pasos,        n veces
7    return s;                       1 paso
8  }
```

El total es de $4n + 4$ pasos, pero se simplifica con la expresión $c_0 \cdot n + c_1$.

Esta otra versión C del programa cuesta fundamentalmente lo mismo:

sumatorio2.c

```
1  int sumatorio(int n)
2  {
3    int s, i;
4    s = 0;                          1 paso
5    i = 1;                          1 paso
6    while (i<=n) {                  1 paso,    n + 1 veces
7      s = s + i;                    1 paso,    n veces
8      i = i + 1;                    1 paso,    n veces
9    }
10   return s;                       1 paso
11 }
```

Un total de $3n + 4$ pasos, o sea, $c_2 \cdot n + c_3$ pasos. Es un programa diferente del anterior y ejecuta un número diferente de pasos, pero estarás de acuerdo en que sustituir un bucle `for` por un bucle `while` es un cambio menor, un detalle de implementación. Lo sustancial es que se repite n veces la acción de acumular en `s` el valor de un índice que va tomando los valores entre 1 y n y por eso, porque el procedimiento es básicamente el mismo, el coste obtenido es fundamentalmente idéntico.

Vamos ahora con una versión Python del mismo programa:

sumatorio.py

```
1  def sumatorio(n):
2    s = 0                           1 paso
3    for i in range(1, n+1):         ??? pasos
4      s = s + i                     1 paso,        n veces
5    return s                        1 paso
```

Parece que no podemos efectuar el análisis si no conocemos bien qué hace la función `range`, pues no constituye una operación elemental, es decir, puede que cuente como más de un paso. La llamada `range(1, n+1)` construye una lista (un vector) de n elementos y asigna a cada elemento un valor. Esta acción requiere, evidentemente, n pasos. Una vez construida la lista, se debe asignar a `i` el valor de cada uno de sus elementos. Esta acción requerirá otros n pasos.

sumatorio.py

```
1  def sumatorio(n):
2    s = 0                           1 paso
3    for i in range(1, n+1):         2n pasos
4      s = s + i                     1 paso,        n veces
5    return s                        1 paso
```

El coste total del programa Python también puede expresarse, pues, como $c_4 \cdot n + c_5$ pasos. O sea, es un coste de la misma naturaleza que el que presentan los dos programas C:

Programa	Coste
`sumatorio1.c`	$c_0 \cdot n + c_1$
`sumatorio2.c`	$c_2 \cdot n + c_3$
`sumatorio.py`	$c_4 \cdot n + c_5$

En la práctica, el programa Python es más lento que los programas C, ya que el tiempo de ejecución de cada uno de sus pasos es mayor que el de las sentencias C equivalente (Python es un lenguaje interpretado y C es un lenguaje compilado). O sea, el valor de c_4 y c_5 es mayor que el de las restantes constantes. Por la misma razón, es probable que un programa equivalente cuidadosamente diseñado en ensamblador fuera aún más rápido que un programa C. Pero ten cuenta que la velocidad de ejecución no es el único criterio a tener en cuenta cuando se elige un lenguaje de programación. Por regla general, programar en ensamblador es mucho más complicado que programar en C, y programa en C es mucho más complicado que programar en Python. Pero no nos desviemos: queremos llamarte la atención sobre el hecho de que el coste temporal crece, en los tres programas, en *proporción directa* a n, la talla del problema. Y eso es lo que, en el fondo, nos interesa estudiar con el enfoque que estamos adoptando: cómo evoluciona el coste *de un algoritmo* con la talla, independientemente del lenguaje de programación o de los detalles de implementación. ¿Es el coste constante?, ¿crece linealmente con n? ¿o crece con el cuadrado de n? Ese es el tipo de preguntas acerca de un algoritmo que nos interesa responder.

Recuerda que nuestra tesis es que más importante que el programa resulta ser el algoritmo. El *principio de invarianza* dice que dos implementaciones diferentes del mismo algoritmo difieren en eficiencia en una constante multiplicativa, pero no más.

Mira estos otros programas Python y C que resuelven el mismo problema con un algoritmo diferente:

`sumatorio3.c`

```c
int sumatorio(int n)
{
  return n * (n + 1) / 2;
}
```

1 paso

`sumatorio3.py`

```python
def sumatorio(n):
  return n * (n + 1) / 2
```

1 paso

El número de pasos es constante (¡uno!), independientemente del valor de n.

14.3. Mejor caso, peor caso, caso promedio

14.3.1. Talla e instancia

Algunos de los ejemplos que hemos estudiado hasta el momento «confunden» los conceptos de talla e instancia porque en ellos coinciden ambos. Por ejemplo, al calcular n^2 la *talla* era el mismo valor n, que describe a la vez el problema concreto a resolver, es decir, determina la *instancia* a resolver. Cuando calculábamos el valor del sumatorio de los n primeros números, la instancia venía determinada por n, al igual que la talla. Pero hay problemas en los que, para una misma talla, hay muchas instancias diferentes. Por ejemplo, al ordenar un vector de enteros podemos distinguir claramente talla e instancia: no hay un único vector de talla 10. Cada vector diferente de talla 10 es una instancia diferente del problema de ordenación. Consideremos algunos ejemplos más.

Esta función calcula, dados dos enteros a y b tales que $a \leq b$, la suma de todos los enteros comprendidos entre ellos (incluidos ambos):

Pseudocódigo

En muchos libros de texto encontrarás los algoritmos expresados en una notación que no corresponde a la de ningún lenguaje de programación, pero que expresa perfectamente un método resolutivo para un problema; es la denominada *notación algorítmica* o *pseudocódigo*.

El algoritmo que subyace al cálculo del sumatorio de los n primeros números mediante bucles puede expresarse con dicha notación así:

algoritmo sumatorio
  Entrada: $n \in \mathbb{N}$
  Salida: $s \in \mathbb{N}$
método
  $s \leftarrow 0$
  para i **de** 1 **a** n **hacer**
    $s \leftarrow s + i$
  fin para
fin método

```
1  def sumatorio(a, b):
2    s = 0
3    for i in range(a, b+1):
4      s += i
5    return s
```

La instancia viene determinada por los valores concretos de a y b, pero la talla parece estar relacionada con $n = b - a + 1$: cuanto mayor es la diferencia entre b y a, más «grande» es el problema. El coste temporal es función de n:

1	`def sumatorio(a, b):`	
2	`  s = 0`	1 paso
3	`  for i in range(a, b+1):`	$n = b - a + 1$ pasos
4	`    s += i`	1 paso, n veces
5	`  return s`	1 paso

Para un valor dado de la talla n hay una infinidad de valores posibles de a y b, es decir, de instancias posibles. Por ejemplo, $a = 1$ y $b = 10$ es una instancia con la misma talla ($n = 10$) que la instancia $a = 101$ y $b = 110$.

Analicemos otro programa: un programa que calcula el sumatorio de números de una lista. Aquí tienes el código:

```
1  def sumatorio(lista):
2    s = 0
3    for i in lista:
4      s += i
5    return s
```

Antes de empezar el análisis hemos de tener claro qué parámetro determina la talla del problema. Ahora no tenemos uno o dos valores escalares como datos, sino una lista o vector. En este caso, la talla del problema es la longitud de `lista`. A más elementos en la lista, más «grande» es el problema y más cuesta calcular el sumatorio. Si denotamos con n a la talla de la lista, es fácil comprobar que el coste es de $c_1 \cdot n + c_0$ pasos:

1	`def sumatorio(lista):`		
2	`  s = 0`	1 paso	
3	`  for i in lista:`	n pasos (n es `len(lista)`)	
4	`    s += i`	1 paso,	n veces
5	`  return s`	1 paso	

Para un valor dado de n hay una infinidad de listas diferentes. Por ejemplo, `[1, 3, 5]`, `[-4, 2, 109]` y `[0, 0, 2]` son tres instancias de talla $n = 3$.

Hemos visto, de momento, que el coste depende de la talla del problema, y no de la instancia. No siempre es así.

14.3.2. Coste dependiente de la instancia

Vamos a analizar otro programa: una función Python que recibe una lista y un valor y dice si el elemento pertenece o no a la lista (sin hacer uso del operador `in`).

```
1  def pertenece(lista, buscado):
2    for i in lista:
3      if i == buscado:
4        return 1
5    return 0
```

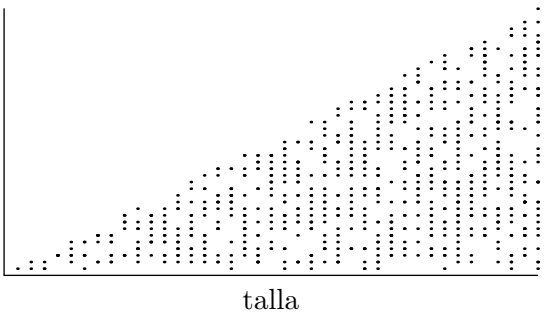
Empecemos nuestro análisis. ¿Cuál es la talla del problema? Está claro que, a más elementos en `lista`, más cuesta resolver el problema. Que el valor de `buscado` sea mayor o menor no influye en la talla del problema. La talla, será, pues la longitud de la lista, a la que denotaremos con n. Nuevamente hay varias instancias posibles *para cada valor de* n, pues hay infinidad de listas de n elementos e infinidad de valores que podemos buscar en la lista.

¿Cuántos pasos, en función de n, se ejecutan? Hay veces que muy pocos (cuando el elemento buscado ocupa las primeras posiciones de `lista`) y hay veces que muchos (cuando el elemento buscado está hacia el final de `lista` o, sencillamente, no está en `lista`).

1	`def pertenece(lista, buscado):`		
2	`  for elemento in lista:`	entre 1 y n pasos	
3	`    if elemento == buscado:`	1 paso,	entre 1 y n veces
4	`      return 1`	1 paso,	entre 0 y 1 veces
5	`  return 0`	1 paso	

Este análisis es más complicado que los realizados hasta el momento. ¡Una línea supone la ejecución de un número indeterminado de pasos, hay una sentencia que se ejecuta entre una y n veces y otra que se ejecuta a lo sumo una vez!

Sigamos una aproximación empírica. Ejecutemos el programa con diferentes tallas y diferentes instancias para cada talla y veamos cuántos pasos cuesta en cada caso. Esta gráfica te muestra, en función de n, el número de pasos para diferentes instancias del problema:



No obtenemos, como hasta ahora, una curva, sino una serie de puntos con una gran variabilidad. Cada vez que ejecutamos el programa con una instancia diferente tarda un número de pasos diferente, incluso para un valor de n fijo.[5] ¿Cómo podemos caracterizar este comportamiento?

Centrémonos en las dos situaciones extremas del algoritmo, en su comportamiento para una situación ideal y en su comportamiento para la peor situación imaginable: analicemos el *mejor* y el *peor* de los casos.

Cuando hablamos del mejor de los casos nos referimos a las instancias que, para cada valor particular de n, se resuelven más rápidamente con el algoritmo estudiado. En nuestro caso, esa instancia es siempre aquella en la que el elemento buscado ocupa la primera posición de la lista.

El peor de los casos lo determinan aquellan instancias que, para cada valor de n, hacen que el algoritmo se ejecute con el mayor número posible de pasos. En nuestro programa, el peor caso es aquel en el que el elemento buscado no está en la lista, pues entonces hemos de recorrerla completamente y ejecutar el `return 0` de la última línea.

Un error demasiado frecuente

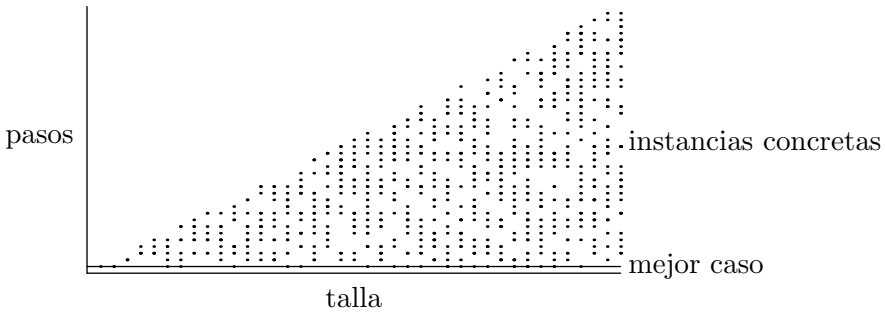
Ojo con el concepto de «mejor de los casos», pues muchos estudiantes lo confunden con «la instancia más pequeña» y responden que el mejor de los casos es que n valga 0, pues entonces ¡el problema es trivial! Es un error de concepto muy serio. No hay *un* mejor caso, sino un mejor caso para cada valor de n. El mejor caso *para cada valor de* n es aquel que hace que el algoritmo se ejecute con el menor número posible de pasos.

Del mismo modo, no debes confundir el «peor de lo casos» con el que presenta la talla más grande (¿existe, siquiera, una talla máxima?). Se trata de ver, para cada valor de n, qué características tienen las instancias del problema que hacen que el algoritmo haya de ejecutar el mayor número posible de pasos.

En el *mejor de los casos*, la función ejecuta un número constante de pasos (tres). Si el elemento buscado está en la primera posición de la lista, la función finaliza enseguida:

- se asigna a `i` el valor del primer elemento,
- se compara `i` con `buscado` y ambos valores coinciden,
- y se devuelve el valor 1.

No importa cuán grande es la lista: el mejor caso («el elemento buscado es el primero de la lista») siempre se resuelve en 3 pasos. En el mejor de los casos, pues, el coste *no depende de* n y es una simple constante, digamos c_0.



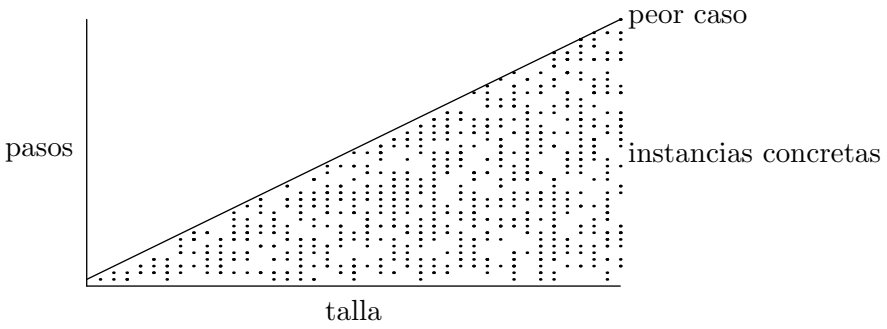
El coste en el *peor de los casos* («el elemento `buscado` no está en la lista») sí depende de n:

- se asigna a `i` el valor de cada uno de los n elementos,
- se compara `i` con `buscado` n veces sin que coincida ninguna de las veces,

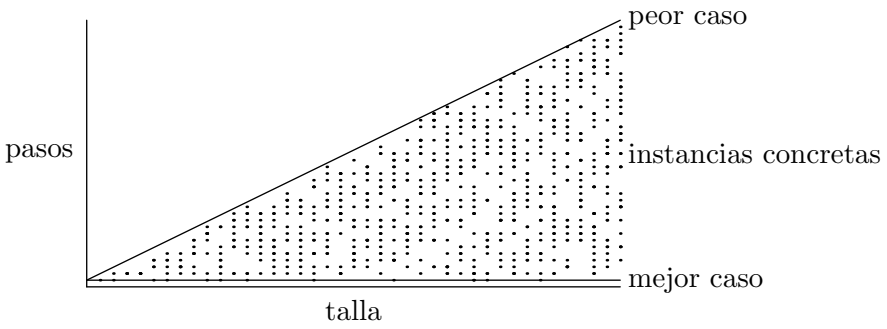
[5] Algo parecido nos hubiera ocurrido con los algoritmos de ordenación si hubiésemos reparado en estas consideraciones en su momento. ¿Ves por qué?

- se devuelve el valor 0 desde el `return` que hay fuera del bucle.

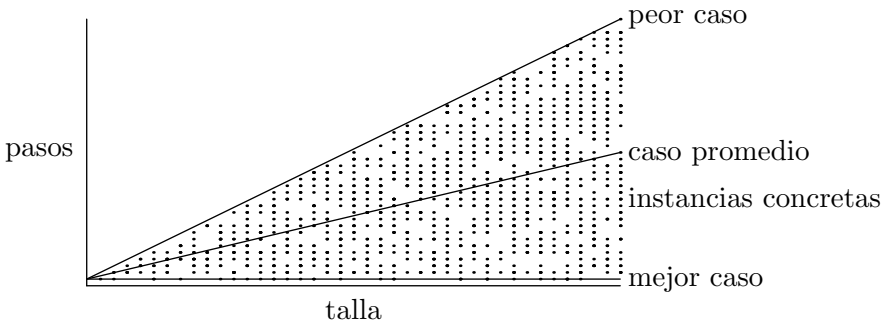
El coste total para el peor caso es de $c_1 \cdot n + c_2$ pasos. O sea, el coste de resolución de cada instancia está siempre por debajo del que determina una línea recta:



En conclusión, el tiempo invertido en resolver cada instancia está *acotado inferiormente* por una curva de la forma c_0 y *acotado superiormente* por una curva de la forma $c_1 \cdot n + c_2$:



Podríamos estudiar el coste del *caso promedio* empírica o analíticamente. Promediando el número de pasos de la ejecución de toda instancia para cada valor de n tendríamos una nueva curva en la gráfica con su propio comportamiento asintótico:



Lo único que sabemos a ciencia cierta sobre la curva de coste promedio es que siempre está comprendida entre las otras dos curvas.

14.3.3. ¿Qué tipo de coste es más interesante cuando comparamos dos algoritmos?

Tenemos, pues, tres conceptos distintos de coste asintótico: coste en el mejor de los casos, coste en el peor de los casos y coste promedio. Los tres caracterizan el comportamiento asintótico del coste temporal del algoritmo.

¿Y cuál de ellos resulta más informativo? Pues depende. Parece evidente que el coste en el caso promedio es muy interesante, pero en según qué aplicación resulta más relevante el coste en el peor de los casos. Imagina que un programa se comporta de cierta forma en promedio, pero que en ciertos casos presenta un coste prohibitivo. ¿Elegirías ese algoritmo frente a otro que, en promedio es algo peor, pero nunca dispara su coste?

Un ejemplo te ayudará a decidir. Supón que usas una determinada rutina de cómputo para controlar el proceso de fisión en un reactor nuclear y que dicho proceso puede controlarse siempre que el tiempo de respuesta del programa sea inferior a la décima de segundo. ¿Te conformarías con un programa que en promedio tarda 1 milisegundo en dar la solución pero que, de vez en cuando, para instancias raras pero posibles, tarda dos días en responder. Seguro que no. Preferirías un algoritmo cuyo coste *en el peor de los casos* entre dentro de los márgenes de seguridad, aunque en promedio tarde más de 1 centésima de segundo, es decir, diez veces más que el otro.

Nosotros centraremos la atención en el análisis del coste en el peor y el mejor de los casos por las siguientes razones:

- Por el momento, las herramientas matemáticas a nuestro alcance hacen relativamente sencillo el análisis de estos costes. El coste promedio es, normalmente, mucho más complicado de estimar y requiere destreza con ciertas técnicas matemáticas avanzadas.

- Un estudio apropiado del coste promedio implica efectuar asunciones acerca de la distribución estadística de las instancias, lo cual no siempre resulta inmediato.

- A fin de cuentas, el coste promedio siempre está acotado superiormente por el coste en el peor de los casos e inferiormente por el coste en el mejor de los casos. Estimar, pues, estos últimos costes proporciona implícitamente cierta información sobre el coste en el caso promedio. En no pocas ocasiones, cuando el mejor y el peor de los casos coinciden, conocemos implícitamente el coste en el caso promedio.

14.4. Notación asintótica

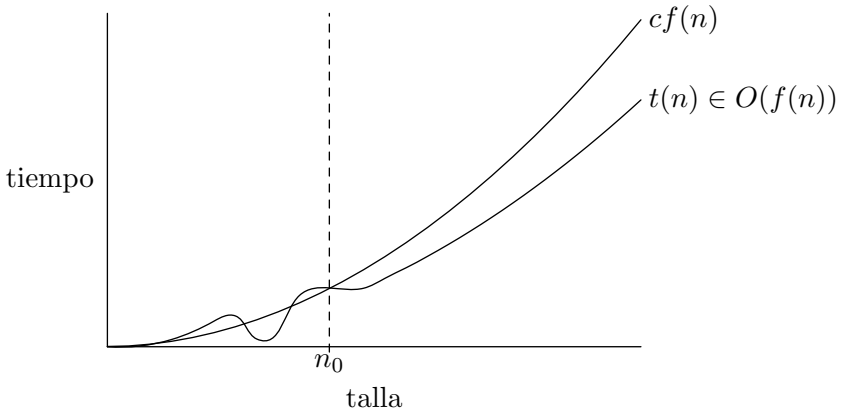
Antes de seguir hemos de detenernos a estudiar ciertas herramientas matemáticas fundamentales que simplifican notablemente los análisis de costes y permiten expresar de forma muy concisa los resultados. Aprenderemos a caracterizar el coste mediante funciones *simples* que acoten superior e inferiormente el coste de toda instancia para tallas suficientemente grandes. Para ello necesitamos definir familias de cotas.

14.4.1. Orden y omega

Sea $f : \mathbb{N} \to \mathbb{R}^+$ una función de los naturales en los reales positivos. Definimos las siguientes familias de funciones:

$$\begin{aligned} O(f(n)) &\doteq \{t : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad t(n) \leq cf(n)\}, \\ \Omega(f(n)) &\doteq \{t : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad t(n) \geq cf(n)\}. \end{aligned}$$

Analicemos qué es cada familia de funciones. $O(f(n))$, que se lee «orden de $f(n)$», es el conjunto de funciones cuyo valor es siempre menor que un cierto número constante c de veces $f(n)$ para valores de n suficientemente grandes. Es decir, es el conjunto de funciones que, asintóticamente, podemos acotar superiormente con una función proporcional a $f(n)$. Este gráfico ilustra qué significa que una función $t(n)$ sea $O(f(n))$:



Fíjate: hay un punto, un valor de n al denominamos n_0, a partir del cual es seguro que $t(n)$ es menor que $cf(n)$ para algún valor constante de c, aunque en el tramo inicial $t(n)$ pueda ser mayor que $cf(n)$.

La función $t(n) = n + 1$, por ejemplo, pertenece a $O(n)$, pues siempre hay un valor n_0 y un valor c para los que $cn \geq n + 1$ si $n \geq n_0$. Considera, por ejemplo, $n_0 = 1$ y $c = 10$. Por cierto, la expresión $t(n) \in O(f(n))$ se lee tanto «$t(n)$ *pertenece* a $O(f(n))$» como «$t(n)$ *es* $O(f(n))$».

Estudiemos algunos ejemplo adicionales de pertenencia de funciones a familias de funciones $O(\cdot)$:

- ¿Pertenece $t(n) = 3n + 2$ a $O(n)$?

  La respuesta es sí. Para todo n mayor o igual que 2, $t(n)$ es menor que $c \cdot n$ cuando c es, por ejemplo, 4:

  $$t(n) = 3n + 2 \leq 4n, \quad \forall n \geq 2. \longrightarrow t(n) \in O(n).$$

- ¿Pertenece $t(n) = 100n + 6$ a $O(n)$?

  Sí:

  $$t(n) = 100n + 6 \leq 101n, \quad \forall n \geq 6 \qquad \longrightarrow t(n) \in O(n).$$

- ¿Pertenece $t(n) = 10n^2 + 4n + 2$ a $O(n^2)$?

  Sí:

  $$t(n) = 10n^2 + 4n + 2 \leq 11n^2, \quad \forall n \geq 5 \qquad \longrightarrow t(n) \in O(n^2)$$

- ¿Pertenece $t(n) = 10n^2 + 4n + 2$ a $O(n)$?

  No. No existen valores de c y n_0 tales que $t(n) \leq c \cdot n$ para todo n mayor o igual que n_0.

- ¿Pertenece $t(n) = 6 \cdot 2^n + n^2$ a $O(2^n)$?

  Sí:

  $$t(n) = 6 \cdot 2^n + n^2 \leq 7 \cdot 2^n, \quad \forall n \geq 4 \qquad \longrightarrow t(n) \in O(2^n)$$

- ¿Pertenece $t(n) = 6 \cdot 2^n + n^2$ a $O(n^{100})$?

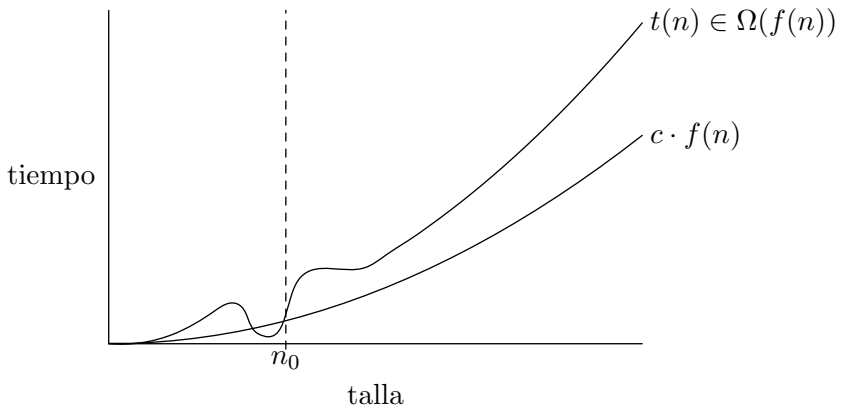
  No. No existen valores de c y n_0 tales que $t(n) \leq c \cdot n^{100}$ para todo n mayor o igual que n_0.

- ¿Pertenece $t(n) = 3$ a $O(1)$?

  Sí.

  $$t(n) = 3 \leq 3, \quad \forall n \geq 0 \qquad \longrightarrow t(n) \in O(1)$$

Cuando una función $t(n)$ es $\Omega(f(n))$ (decimos que pertenece a omega de $f(n)$ o que es omega de $f(n)$) está acotada inferiormente por $c \cdot f(n)$ para valores de n suficientemente grandes y algún valor c, es decir, hay un valor n_0 tal que, para todo n mayor, la función $t(n)$ es mayor que $c \cdot f(n)$.



Vemos un ejemplo: ¿pertenece $t(n) = 10n^2 + 4n + 2$ a $\Omega(n^2)$? La respuesta es sí. Para $n \geq 5$, el valor de $t(n)$ es siempre menor que $11n^2$.

Ejercicios

▶ **334** Determina el orden y omega de estas funciones:

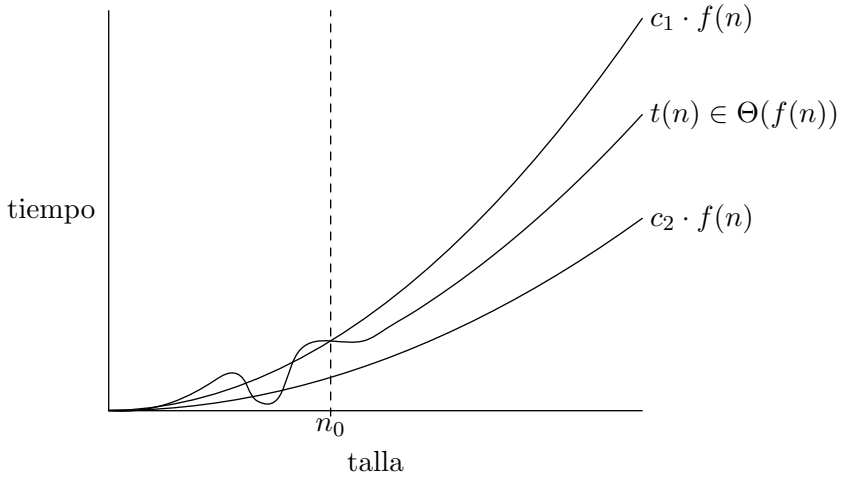
a) $t(n) = 4$.
b) $t(n) = 0.1$.
c) $t(n) = 108374$.
d) $t(n) = n$.
e) $t(n) = 10n$.
f) $t(n) = 0.028764n$.
g) $t(n) = \pi n$.
h) $t(n) = n + 3$.
i) $t(n) = 2n + 1093842$.
j) $t(n) = n^2$.
k) $t(n) = 4n^2$.
l) $t(n) = 8n^2 + n$.
m) $t(n) = 0.1n^2 + 10n + 1$.
n) $t(n) = n^6 + n^3 + 10n$.
ñ) $t(n) = n^{100} + n^{99}$.
o) $t(n) = 2^n$.
p) $t(n) = 2^n + n^{100}$.
q) $t(n) = 3^n + 2^n$.
r) $t(n) = \log_2(n)$.
s) $t(n) = \log_{10}(n)$.
t) $t(n) = n + \log_2(n)$.
u) $t(n) = n \log_{10}(n)$.
v) $t(n) = n^2 + n \log_{10}(n)$.
w) $t(n) = n^2 \log_2(n)$.
x) $t(n) = n^2 + n^2 \log_e(n)$.
y) $t(n) = \log_e(n^2)$.
z) $t(n) = n \log_2(n^3)$.

14.4.2. Zeta

Hay un elemento de la notación asintótica que aún no hemos presentado. Cuando una función $t(n)$ es a la vez $O(f(n))$ y $\Omega(f(n))$, decimos que es $\Theta(f(n))$ (que se lee «zeta de $f(n)$»).

$$\Theta(f(n)) \doteq O(f(n)) \cap \Omega(f(n)).$$

Una función $t(n)$ que pertenece a $\Theta(f(n))$ está acotada superior e inferiormente por sendas funciones proporcionales a $f(n)$ a partir de un valor determinado de n:



Por ejemplo, la función $t_1(n) = 3n + 2$ es $\Theta(n)$: $t(n)$ es menor que $4n$ para todo n mayor o igual que 2, así que es $O(n)$; por otra parte, $t(n)$ es mayor o igual que $2n$ para todo n mayor o igual que uno, así que es $\Omega(n)$.

Esta otra función, sin embargo, no es $O(\cdot)$ y $\Omega(\cdot)$ de un algún polinomio o función simple:

$$t_2(n) = \begin{cases} n^2 & \text{si } n \text{ par;} \\ n & \text{si } n \text{ impar.} \end{cases}$$

$t_2(n)$ es $O(n^2)$ y $\Omega(n)$, así que no es $\Theta(\cdot)$ de de un polinomio.

Ejercicios

▶ **335** Determina el orden/omega (y zeta, si procede) de las siguientes funciones:

a) $t(n) = 4$.

b) $t(n) = n^2 + n$.

c) $t(n) = 20n^3 - n^2 + 1000n$.

d) $t(n) = n \log n$.

e) $t(n) = n \log n^3$.

f) $t(n) = \begin{cases} 3n + 2, & \text{si } n \text{ par;} \\ 8n^2 + n, & \text{si } n \text{ impar.} \end{cases}$

g) $t(n) = \begin{cases} 3n + 2, & \text{si } n \text{ par;} \\ 8n + n, & \text{si } n \text{ impar.} \end{cases}$

h) $t(n) = \begin{cases} 3n + 2, & \text{si } n < 10; \\ n^2 + n, & \text{si } n \geq 10. \end{cases}$

i) $t(n) = |n \sin(n)|$.

j) $t(n) = n^2 + n \log n$.

14.4.3. Jerarquía de cotas

Hay una relación de inclusión entre diferentes órdenes:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n).$$

Y también entre diferentes omegas:

$$\Omega(1) \supset \Omega(\log n) \supset \Omega(\sqrt{n}) \supset \Omega(n) \supset \Omega(n \log n) \supset \Omega(n^2) \supset \Omega(n^3) \supset \Omega(2^n) \supset \Omega(n^n).$$

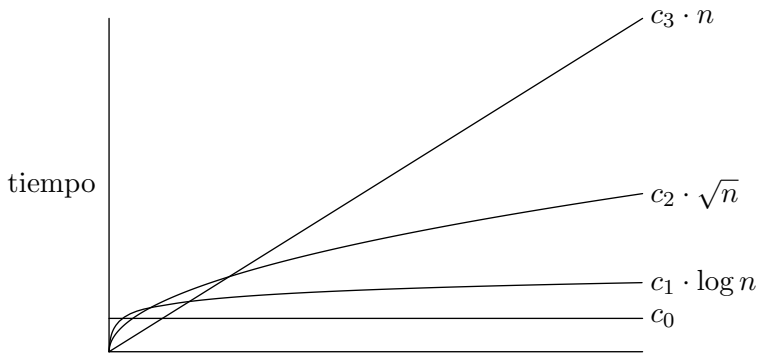
Fíjate que estas relaciones entre costes determinan que si una función como $3n + 2$ pertenece a $O(n)$, pertenece también a $O(n^2)$ y a $O(2^n)$, pero *siempre proporcionaremos como cota el orden que más se ajusta a la función.*

Las funciones que pertenecen a cada orden tienen un adjetivo que las identifican:

Sublineales		Constantes	$O(1)$
		Logarítmicas	$O(\log n)$
			$O(\sqrt{n})$
Lineales			$O(n)$
Superlineales			$O(n \log n)$
	Polinómicas	Cuadráticas	$O(n^2)$
		Cúbicas	$O(n^3)$
	Exponenciales		$O(2^n)$
			$O(n^n)$

Así, decimos que *el coste* (temporal) de un algoritmo es lineal cuando es $O(n)$ y cúbico cuando es $O(n^3)$. De hecho, abusando del lenguaje decimos que el algoritmo *es* lineal o cúbico, respectivamente.

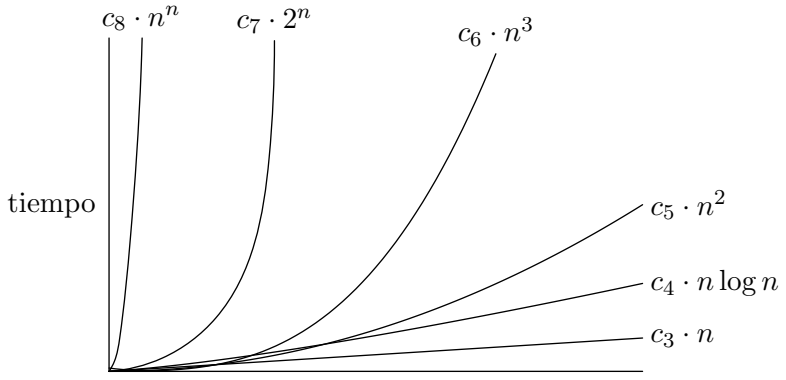
¿Qué implica, a efectos prácticos, que un coste sea logarítmico, lineal, exponencial, etc.? Lo mejor será que comparemos algunas gráficas de crecimiento. Empecemos por los crecimientos lineal y sublineales.



A la vista de las gráficas, reflexionemos sobre el crecimiento de algunas funciones de coste y las implicaciones que tienen al evaluar la eficiencia de un algoritmo:

- Un algoritmo de coste constante ejecuta un número constante de instrucciones o acotado por una constante independiente de la talla del problema. Un algoritmo que soluciona un problema en tiempo constante es lo ideal: a la larga es mejor que cualquier algoritmo de coste no constante.
- El coste de un algoritmo logarítmico crece muy lentamente conforme n crece. Por ejemplo, si resolver un problema de talla $n = 10$ tarda $1\,\mu$s, puede tardar $2\,\mu$s en resolver un problema 10 veces más grande ($n = 100$) y sólo $3\,\mu$s en resolver uno 100 veces mayor ($n = 1000$). Cada vez que el problema es a veces más grande, se incrementa en una unidad el tiempo necesario.
- Un algoritmo cuyo coste es $\Theta(\sqrt{n})$ crece a un ritmo superior que otro que es $\Theta(\log n)$, pero no llega a presentar un crecimiento lineal. Cuando la talla se multiplica por cuatro, el coste se multiplica por dos.

Analicemos ahora los crecimientos lineal y superlineales.



- Un algoritmo $\Theta(n \log n)$ presenta un crecimiento del coste ligeramente superior al de un algoritmo lineal. Por ejemplo, si tardamos $10\mu s$ en resolver un problema de talla 1000, puede que tardemos $22\,\mu$s, poco más del doble, en resolver un problema de talla 2000.
- Un algoritmo cuadrático empieza a dejar de ser útil para tallas medias o grandes, pues pasar a tratar con un problema *el doble* de grande requiere *cuatro veces más* tiempo.
- Un algoritmo cúbico sólo es útil para problemas pequeños: *duplicar* el tamaño del problema hace que se tarde *ocho veces más* tiempo.
- Un algoritmo exponencial raramente es útil. Si resolver un problema de talla 10 requiere una cantidad de tiempo determinada con un algoritmo $\Theta(2^N)$, tratar con uno de talla 20 requiere ¡unas 1000 veces más tiempo!

La siguiente tabla también te ayudará a ir tomando conciencia de las tasas de crecimiento. Supón que las instancias de un problema de talla $n = 1$ se resuelven con varios algoritmos constantes, lineales, etc. en 1 μs. En esta tabla tienes el tiempo aproximado que cuesta resolver problemas con cada uno de ellos:

Coste	$n = 1$	$n = 5$	$n = 10$	$n = 50$	$n = 100$
Constante	1 μs	1 μs	1 μs	1 μs	1 μs
Logarítmico	1 μs	1.7 μs	2 μs	2.7 μs	3 μs
Lineal	1 μs	5 μs	10 μs	50 μs	100 μs
$n \log n$	1 μs	4.5 μs	11 μs	86 μs	201 μs
Cuadrático	1 μs	25 μs	100 μs	2.5 ms	10 ms
Cúbico	1 μs	125 μs	1 ms	125 ms	1s
Exponencial (2^n)	1 μs	32 μs	1 ms	1 año y 2 meses	40 millones de eones

El último número es bárbaro. Ten en cuenta que un eón son mil millones de años y que los científicos estiman actualmente que la edad del universo es de entre 13 y 14 eones.

No tiene sentido seguir estudiando cómo crece el coste de un algoritmo exponencial. Probemos con valores de n mayores para el resto de costes:

Coste	$n = 1000$	$n = 10000$	$n = 100000$
Constante	1 μs	1 μs	1 μs
Logarítmico	4 μs	5 μs	6 μs
Lineal	1 ms	10 ms	100 ms
$n \log n$	3 ms	40 ms	500 ms
Cuadrático	1 s	100 s	16 minutos y medio
Cúbico	16 minutos y medio	1 día y medio	casi 32 años

14.4.4. Algunas propiedades de las cotas

Una propiedad evidente es que $f(x)$ es $O(f(x))$. Sí, pero ¿es $O(n)$ cualquier función $O(3n)$? La respuesta es sí: si $t(n) \in O(n)$, existe un valor c tal que, para n suficientemente grande, $t(n)$ es menor o igual que $c \cdot n$. Para ser $O(3n)$ debe existir una constante c' tal que, para n suficientemente grande, $c' \cdot n \geq t(n)$. ¿Existe con seguridad esa c'? Sí: $c' = c/3$. O sea, podemos simplificar $O(3n)$ como $O(n)$: son la misma familia de funciones.

En general,

$$O(c \cdot f(n)) = O(f(n)).$$

Ejercicios

▶ **336** ¿Es $O(\log_a n) = O(\log_b n)$ siendo a y b constantes?

Es más, si una función es $O(n^2+n+1)$, ¿es $O(n^2)$? Fíjate en que $10n^2+4n+2$ crece ligeramente más rápido que $10n^2$, pues tiene un término que depende linealmente de n. Pero ese término no domina el crecimiento del coste: lo domina el término cuadrático. En el apartado anterior demostramos que, efectivamente, $10n^2 + 4n + 2$ es $O(n^2)$.

La notación $O(\cdot)$ simplifica mucho la expresión de los costes, pues permite su reducción a su *término dominante*, eliminando todas las constantes de proporcionalidad y términos adicionales.

Podemos expresar esa propiedad así:

$$\begin{aligned} t_1(n) \in O(t_2(n)) &\rightarrow (t_1(n) + t_2(n)) \in O(t_2(n)), \\ t_1(n) \in \Omega(t_2(n)) &\rightarrow (t_1(n) + t_2(n)) \in \Omega(t_1(n)). \end{aligned}$$

Una conclusión de esta propiedad es que cualquier polinomio de grado k es $O(n^k)$ y $\Omega(n^k)$ (o sea, es $\Theta(n^k)$).

Algunas cotas precalculadas

Esta tabla resume algunas relaciones entre funciones de coste y sus cotas superiores e inferiores:

$t(n)$	Orden	Omega	Zeta	
c	$O(1)$	$\Omega(1)$	$\Theta(1)$	$\forall c > 0$
$\sum_{i=0}^{k} c_i n^i$	$O(n^k)$	$\Omega(n^k)$	$\Theta(n^k)$	$\forall c_k \in \mathbb{R}^+$ y $c_i \in \mathbb{R}$, $1 \leq i < k$
$\sum_{i=1}^{n} i^k$	$O(n^{k+1})$	$\Omega(n^{k+1})$	$\Theta(n^{k+1})$	$\forall k \in \mathbb{N}^+$
$\sum_{i=1}^{n} (n-i)^k$	$O(n^{k+1})$	$\Omega(n^{k+1})$	$\Theta(n^{k+1})$	$\forall k \in \mathbb{N}^+$
$n!$	$O(n^n)$	$\Omega(2^n)$	—	
$\log(n!)$	$O(n \log n)$	$\Omega(n \log n)$	$\Theta(n \log n)$	
$\sum_{i=1}^{n} r^i$	$O(r^n)$	$\Omega(r^n)$	$\Theta(r^n)$	$\forall r \in \mathbb{R}^{>1}$
$\sum_{i=1}^{n} \frac{1}{i}$	$O(\log n)$	$\Omega(\log n)$	$\Theta(\log n)$	
$\sum_{i=1}^{n} \frac{i}{r^i}$	$O(1)$	$\Omega(1)$	$\Theta(1)$	$\forall r \in \mathbb{R}^{>1}$

Otra propiedad interesante porque permite simplificar el análisis de algoritmos atañe al producto de funciones:

$$t_1(n) \in \Theta(f_1(n)), t_2(n) \in \Theta(f_2(n)) \quad \rightarrow \quad t_1(n) \cdot t_2(n) \in \Theta(f_1(n) \cdot f_2(n)).$$

Podemos manipular algebraicamente las cotas expresadas en notación asintótica y efectuar así simplificaciones notables con gran facilidad. Nos tomaremos ciertas libertades con la notación para expresar operaciones como la suma o el producto de cotas y, basándonos en propiedades como las expresadas en el apartado anterior, concluir que:

- $c \cdot O(f(n)) = O(f(n))$.
- $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$.
- $O(O(f(n))) = O(f(n))$.
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- $f(n) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- $O(f(n)) + O(g(n)) = O(f(n))$ si $O(g(n)) \subseteq O(f(n))$.

Encuentra tú mismo relaciones similares para Ω y Θ.

Aquí hacemos uso de estas propiedades para determinar el orden de una función relativamente

complicada:

$$\begin{aligned}\left(8+\log n+\frac{1}{n}\right)\cdot\left(n+\sqrt{n}\right) &= 8n+n\log n+1+8\sqrt{n}+\sqrt{n}\log n+\frac{\sqrt{n}}{n}\\ &\in O\left(8n+n\log n+1+8\sqrt{n}+\sqrt{n}\log n+\frac{\sqrt{n}}{n}\right)\\ &= O(8n)+O(n\log n)+O(1)+O(8\sqrt{n})+O(\sqrt{n}\log n)+O\left(\frac{\sqrt{n}}{n}\right)\\ &= O(n)+O(n\log n)+O(1)+O(\sqrt{n})+O(\sqrt{n}\log n)+O\left(\frac{\sqrt{n}}{n}\right)\\ &= O(n\log n)\end{aligned}$$

Ejercicios

▶ **337** Determina el orden de las siguientes funciones. Utiliza una expresión sencilla para la cota.

a) $f(n)=(n+1)(n-1)$.

b) $f(n)=10^2 n\log n^{(n+1)}$.

c) $f(n)=7e^n+12/n$.

14.5. Ejemplos de análisis de coste temporal

Ya disponemos de las herramientas necesarias para analizar asintóticamente el coste temporal de los algoritmos y presentar su coste como relación de pertenencia a orden, omega y zeta. Vamos a presentar unos cuantos análisis de complejidad temporal de algoritmos.

14.5.1. Sumatorio de una serie aritmética

Esta rutina C calcula $\sum_{i=1}^{n} a_i$, donde a_n es una serie aritmética, es decir, dados a_1 y d, el valor de a_i es $a_{i-1}+d$ para todo $i>1$.

```
1  double suma_aritmetica(double a1,
   double d, int n)
2  {
3    double s, an;
4    int i;
5    an = a1;                          1 paso
6    s = a1;                           1 paso
7    for (i=2; i<=n; i++) {            2(n − 1) + 2 pasos
8      an += d;                        1 paso,          n − 1 veces
9      s += an;                        1 paso,          n − 1 veces
10   }
11   return s;                         1 paso
12 }
```

El número de pasos es función de n:

$$t(n)=5+4(n-1)=4n+1.$$

Tenemos, pues, que $t(n) \in \Theta(n)$. El coste es lineal. A mayor valor de n aumenta proporcionalmente el tiempo de cálculo.

No es necesario efectuar un análisis tan detallado. Podemos limitarnos a estudiar el número de veces que se ejecuta cada línea y acotarlo convenientemente:

```
1  double suma_aritmetica(double a1,
   double d, int n)
2  {
3    double s, an;
4    int i;
5    an = a1;                          Θ(1)
6    s = a1;                           Θ(1)
7    for (i=2; i<=n; i++) {            Θ(n)
8      an += d;                        Θ(1),   Θ(n) veces
9      s += an;                        Θ(1),   Θ(n) veces
10   }
11   return s;                         Θ(1)
12 }
```

El coste es $5\Theta(1) + \Theta(n) + 2\Theta(1)\Theta(n)$. Simplificando, el coste queda en $\Theta(n)$.

Aquí tienes un programa equivalente para calcular el valor del sumatorio de una serie aritmética que sigue una estrategia distinta:

```
1  double suma_aritmetica(double a1,
   double d, int n)
2  {
3    double an = a1 + (n-1) * d;       Θ(1)

4    return (n * (a1 + an)) / 2.0;     Θ(1)

5  }
```

El coste del algoritmo que implementa este programa es constante, es decir, $\Theta(1)$. Siempre tarda lo mismo.

14.5.2. Pertenencia de un elemento a una lista

Completemos el análisis de la función `pertenece` que dejamos en suspenso para introducir los conceptos de coste en el mejor y peor de los casos (n representa a `len(lista)`):

```
1  def pertenece(lista, buscado):
2    for i in lista:                   O(n)
3      if i == buscado:                Θ(1),   O(n) veces
4        return 1                      Θ(1),   O(1) veces
5    return 0                          Θ(1)
```

El coste es $O(n) + \Theta(1)O(n) + \Theta(1)O(1) + \Theta(1)$. ¿Qué es $\Theta(1)O(n)$ Evidentemente, $O(n)$: hacer a lo sumo n veces algo que es constante, es $O(n)$. ¿Y $\Theta(1)O(1)$? Naturalmente, $O(1)$. El coste es, pues, $O(n) + O(n) + O(1) = O(n)$.

Por cierto, sabemos escribir de forma más concisa la determinación de la pertenencia a una lista:

```
def pertenece(lista, buscado):
  return buscado in lista
```

Sólo consta de una línea. ¿Es esta versión $\Theta(1)$? No. Sigue siendo $O(n)$. Python no hace magia: determina si un elemento pertenece o no a una lista recorriendo sus elementos, igual que hicimos en el programa anterior. Recuerda que no nos interesa contar el *número de líneas de un programa*, sino el *número de pasos de un algoritmo*. Si el algoritmo efectúa una operación que es $O(n)$, poco importa que nuestro lenguaje permita expresarla con una sola línea o con un simple operador: en el peor caso sigue requiriendo recorrer una lista de n elementos.

Ejercicios

▶ **338** Describe con notación asintótica el coste temporal de esta rutina Python:

```
def productorio(n):
  p = 1
  for i in range(2, n+1):
    p *= i
  return p
```

▶ **339** Describe con notación asintótica el coste temporal de esta rutina Python:

```
def minimo(lista):
  if len(lista) == 0:
    return None
  m = lista[0]
  for i in range(1, len(lista)):
    if m < lista[i]:
      m = lista[i]
  return m
```

(Nota: `len` no es una operación elemental. Debes saber, pues, que el coste de calcular `len(lista)` es, en Python, $\Theta(1)$.)

A la hora de efctuar los cálculos, supón primero que acceder a un elemento de una lista Python es $\Theta(1)$. A continuación, supón que esa misma operación es $O(n)$.

▶ **340** Describe con notación asintótica el coste temporal de esta rutina Python:

```
def ultimo(lista):
  if len(lista) > 0:
    return lista[-1]
  else:
    return None
```

Supón que acceder a un elemento de una lista Python es $\Theta(1)$.

▶ **341** Describe con notación asintótica el coste temporal de esta rutina C:

```
char ultimo_caracter(char cadena[])
{
  return cadena[strlen(cadena)-1];
}
```

Nota: `strlen` no es una operación elemental. Recuerda qué hace exactamente `strlen` y cómo funciona antes de responder.

▶ **342** Describe con notación asintótica el coste temporal de esta rutina C:

```
char busca_caracter(char cadena[], char caracter)
{
  int i;
```

```
  for (i=0; i<strlen(cadena); i++)
    if (cadena[i] == caracter)
      return 1;
  return 0;
}
```

▶ **343** Describe con notación asintótica el coste temporal de esta rutina C:

```
char busca_caracter(char cadena[], char caracter)
{
  int i, talla;

  talla = strlen(cadena);
  for (i=0; i<talla; i++)
    if (cadena[i] == caracter)
      return 1;
  return 0;
}
```

14.5.3. Búsqueda de elementos en un vector ordenado

Supongamos ahora que el vector en el que buscamos un elemento está ordenado de menor a mayor. Podemos modificar el algoritmo del apartado anterior para que finalice la búsqueda tan pronto estemos seguros de que el elemento buscado no está en el vector:

1	`def pertenece(vector, buscado):`	
2	`  for i in vector:`	$O(n)$
3	`    if i == buscado:`	$\Theta(1)$, $O(n)$ veces
4	`      return 1`	$\Theta(1)$, $O(1)$ veces
5	`    elif i > buscada:`	$\Theta(1)$, $O(n)$ veces
6	`      return 0`	$\Theta(1)$, $O(1)$ veces
7	`  return 0`	$\Theta(1)$, $O(1)$ veces

Esta nueva versión permite que muchas de las veces en que el elemento buscado no está en el vector, la rutina no se vea obligada a recorrer el vector entero para concluir que no está.

Es seguro que, en la práctica, mejora el coste promedio, pero ¿mejora también el coste en el peor de los casos? Estudiémoslo. Lo peor que puede ocurrir es que `buscado` no se encuentre en `vector` y que su valor sea mayor que el mayor elemento del vector, pues en tal caso tendremos que recorrerlo todo para concluir que el elemento no pertenece.

O sea, el coste en el peor de los casos es proporcional a n, así que el coste temporal de este método sigue siendo $O(n)$ (efectúa los cálculos tú mismo y comprobarás que es así).

En el mejor de los casos, el algoritmo de «búsqueda lineal», que así es como se le conoce, es $\Omega(1)$: el mejor caso es aquel en el que el elemento buscado ocupa la primera posición del vector y se resuelve ejecutando 3 pasos.

Estudiamos ahora un procedimiento de búsqueda dicotómica:

1	`def pertenece(vector, buscado):`		
2	`  desde = 0`	$\Theta(1)$	
3	`  hasta = len(vector) - 1`	$\Theta(1)$	
4	`  while desde <hasta:`	$O(???)$	
5	`    i = (hasta + desde) / 2`	$\Theta(1)$,	$O(???)$ veces
6	`    if buscado == vector[i]:`	$\Theta(1)$,	$O(???)$ veces
7	`      return 1`	$\Theta(1)$,	$O(1)$ vez
8	`    elif buscado <vector[i]:`	$\Theta(1)$,	$O(???)$ veces
9	`      hasta = i - 1`	$\Theta(1)$,	$O(???)$ veces
10	`    else:`	$\Theta(1)$,	$O(???)$ veces
11	`      desde = i + 1`	$\Theta(1)$,	$O(???)$ veces
12	`  return 0`	$\Theta(1)$	

Parece claro que el peor de los casos para este otro método también consiste en buscar un elemento que no está en `vector`. En ese supuesto, ¿cuántas veces se ejecutará el bucle? La clave está en entender que con cada iteración infructuosa del bucle se reduce el espacio de búsqueda a (aproximadamente) la mitad. Es decir, si empezamos buscando un valor en un vector de n elementos, la primera iteración reduce el espacio de búsqueda a un vector virtual de a lo sumo $n/2$ elementos (el número de elementos entre `desde` y `hasta`), y después de la segunda, a a lo sumo $n/4$ elementos.

Elementos iniciales:	n
Elementos tras la primera iteración:	menos de $\frac{n}{2}$
Elementos tras la segunda iteración:	menos de $\frac{n}{4}$
Elementos tras la tercera iteración:	menos de $\frac{n}{8}$
$\vdots$	$\vdots$
Elementos tras la k-ésima iteración:	menos de $\frac{n}{2^k}$

La última iteración tiene lugar cuando sólo queda un elemento en el vector. ¿Cuándo será $n/2^k = 1$? Tomando el logaritmo de ambas partes de la igualdad solucionaremos fácilmente el problema:

$$\frac{n}{2^k} = 1 \quad \rightarrow \quad \log_2 \frac{n}{2^k} = \log_2 1 = 0.$$

Ahora tenemos,

$$0 = \log_2 \frac{n}{2^k} = \log_2 n - \log_2 2^k = \log_2 n - k \quad \rightarrow \quad k = \log_2 n.$$

El número de iteraciones está acotado superiormente por k, que es el logaritmo en base 2 de n:

1	`def pertenece(vector, buscado):`		
2	`  desde = 0`	$\Theta(1)$	
3	`  hasta = len(vector) - 1`	$\Theta(1)$	
4	`  while desde <hasta:`	$O(\log n)$	
5	`    i = (hasta + desde) / 2`	$\Theta(1)$,	$O(\log n)$ veces
6	`    if buscado == vector[i]:`	$\Theta(1)$,	$O(\log n)$ veces
7	`      return 1`	$\Theta(1)$,	$O(1)$ vez
8	`    elif buscado <vector[i]:`	$\Theta(1)$,	$O(\log n)$ veces
9	`      hasta = i - 1`	$\Theta(1)$,	$O(\log n)$ veces
10	`    else:`	$\Theta(1)$,	$O(\log n)$ veces
11	`      desde = i + 1`	$\Theta(1)$,	$O(\log n)$ veces
12	`  return 0`	$\Theta(1)$	

Hacer $O(\log n)$ veces algo que $\Theta(1)$ supone un coste total $O(\log n)$. Así pues, el coste del algoritmo es $O(\log n)$.

Ya tenemos el coste para el peor de los casos. ¿Y cuál es el mejor caso? Aquel en el que el elemento buscado ocupa la posición central del vector, pues lo encontramos en tiempo constante. Así pues, el coste de la búsqueda dicotómica es $\Omega(1)$.

El algoritmo de búsqueda dicotómica es asintóticamente mejor que el de búsqueda directa ($O(\log n)$ frente a $O(n)$), así que es el algoritmo a elegir cuando buscas un elemento en un vector ordenado.

14.5.4. Trasposición de una matriz

Vamos a analizar otro programa. Esta vez, un programa C que traspone una matriz de $n \times n$ elementos. ¿Cuál es el tamaño del problema? Es n.

```
1  #define n 10
2  void trasponer(double m[n][n])
3  {
4    int i, j;
5    double aux;
6    for (i=0; i<n-1; i++)                    Θ(n)
7      for (j=i+1; j<n; j++) {                Θ(n), Θ(n − i) veces
8        aux = m[i][j];                       Θ(1), Θ(n − i) · Θ(n) veces
9        m[i][j] = m[j][i];                   Θ(1), Θ(n − i) · Θ(n) veces
10       m[j][i] = aux;                       Θ(1), Θ(n − i) · Θ(n) veces
11     }
12 }
```

Si nos damos cuenta de que $\Theta(n-i)$ para i entre 0 y $n-2$ es $O(n)$, podemos efectuar un análisis para el peor de los casos:

```
1  #define n 10
2  void trasponer(double m[n][n])
3  {
4    int i, j;
5    double aux;
6    for (i=0; i<n-1; i++)                    Θ(n)
7      for (j=i+1; j<n; j++) {                Θ(n), O(n) veces
8        aux = m[i][j];                       Θ(1), O(n) · Θ(n) = O(n²) veces
9        m[i][j] = m[j][i];                   Θ(1), O(n) · Θ(n) = O(n²) veces
10       m[j][i] = aux;                       Θ(1), O(n) · Θ(n) = O(n²) veces
11     }
12 }
```

El coste del algoritmo es $O(n^2)$.

¿Y si queremos un análisis más preciso? Nuestro problema es que, a simple vista, no sabemos cuántas veces se repetirán el bucle interior y las tres sentencias de asignación que incluye. Pero podemos expresar el número de pasos así:

$$t(n) = n + \sum_{i=0}^{n-2} \left(n - i + \sum_{j=i+1}^{n-1} 3 \right).$$

Calculemos el valor del sumatorio:

$$\begin{aligned}
t(n) &= n + \sum_{i=0}^{n-2} \left(n - i + \sum_{j=i+1}^{n-1} 3 \right) \\
&= n + \sum_{i=0}^{n-2} n - i + 3 \cdot (n - 1 - (i+1) + 1) \\
&= n + \sum_{i=0}^{n-2} 4n - 4i - 3 \\
&= n + \left(\sum_{i=0}^{n-2} 4n \right) - \left(\sum_{i=0}^{n-2} 4i \right) - \left(\sum_{i=0}^{n-2} 3 \right) \\
&= n + (4n^2 - 4n) - (2n^2 - 6n + 4) - (3n - 3) \\
&= 2n^2 - 1.
\end{aligned}$$

Ya está claro. El coste $t(n)$ es $\Theta(n^2)$. No sólo es cuadrático para el peor de los casos: lo es siempre.

Sumatorios

El conteo de pasos en el cálculo de costes para algoritmos iterativos comporta frecuentemente el cálculo de sumatorios. Te presentamos algunos de los que encontrarás con cierta frecuencia:

$$\begin{aligned}
\sum_{i=1}^{n} 1 &= n \\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \\
\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\
\sum_{i=1}^{n} i^3 &= \frac{n^2(n+1)^2}{4}
\end{aligned}$$

El sumatorio de términos una progresión aritmética $a_n = a_1 + (n-1)d$ es

$$\sum_{i=1}^{n} a_n = \frac{n}{2}(a_1 + a_n)$$

y el de una progresión geométrica, $a_n = a_1 r^{n-1}$

$$\sum_{i=1}^{n} a_n = a_1 \frac{r^n - 1}{r - 1}.$$

Las siguientes propiedades de los sumatorios también te resultarán útiles:

$$\begin{aligned}
\sum_{i=1}^{n} c \cdot f(n) &= c \cdot \sum_{i=1}^{n} f(n) \\
\sum_{i=1}^{n} (f(n) + g(n)) &= \left(\sum_{i=1}^{n} f(n) \right) + \left(\sum_{i=1}^{n} g(n) \right)
\end{aligned}$$

14.5.5. Suma de matrices

Este programa C suma dos matrices de $n \times m$ y deja el resultado en una tercera matriz:

```
1  #define M 10
2  #define N 20
3  void suma_matrices(double a[M][N],
   double b[M][N], double c[M][N])
4  {
5    int i, j;
6    for (i=0; i<N; i++)                    Θ(N)
7      for (j=0; j<M; j++)                  Θ(M), Θ(N) veces
8        c[i][j] = a[i][j] + b[i][j];       Θ(1), Θ(NM) veces
9  }
```

Este programa es diferente de los demás a efectos del análisis. La talla del problema no viene determinada por un único parámetro, sino por dos: el número de filas, N, y el de columnas, M. El algoritmo es $O(NM)$. No hay problema en expresar el coste de un algoritmo como función de dos parámetros, pues la talla es también función de dos parámetros.

Ejercicios

▶ **344** Esta función C nos dice si una cadena de talla m (`patron`) aparece o no en otra cadena de talla n (`texto`). ¿Qué coste temporal tiene?

```
int incluye_cadena(char patron[], char texto[])
{
  int i, j;
  int coincidencias;
  int talla_patron, talla_texto;

  talla_patron = strlen(patron);
  talla_texto = strlen(texto);
  for (i=0; i<talla_texto - talla_patron + 1; i++) {
    coincidencias = 0;
    for (j=0; j<talla_patron; j++)
      if (patron[j] == texto[i+j])
        coincidencias++;
      else
        break;
    if (coincidencias == talla_patron)
      return 1;
  }
  return 0;
}
```

▶ **345** Hay algoritmos capaces de decidir si una cadena de talla n aparece en otra de talla m en tiempo $O(n+m)$. Encuentra uno en libros de programación o algorítmica y estúdialo.

14.5.6. Producto de matrices

Este programa multiplica dos matrices cuadradas de $n \times n$ y deja el resultado en una tercera matriz de $n \times n$:

```
1  #define N 10
2  void producto_matrices(double
   a[N][N], double b[N][N], double
   c[N][N])
3  {
4    int i, j, k;
5    for (i=0; i<N; i++)                    Θ(N)
6      for (j=0; j<N; j++) {                Θ(N),    Θ(N) veces
7        c[i][j] = 0.0;                     Θ(1),    Θ(N^2) veces
8        for (k=0; k<N; k++)                Θ(N),    Θ(N^2) veces
9          c[i][j] += a[i][k] * b[k][j];    Θ(1),    Θ(N^3) veces
10     }
11 }
```

El coste temporal de esta técnica para el cálculo del producto matricial es $\Theta(N^3)$.

Ejercicios

► **346** ¿Qué coste presenta un algoritmo que calcula el producto de una matriz de talla $p \times q$ por una matriz de talla $q \times r$ con un procedimiento similar al implementado en `producto_matrices`?

14.5.7. Cálculo de la moda

Este programa C calcula la moda de una serie de valores enteros entre 0 y 100:

```
char moda(char valores[], unsigned int talla)
{
  unsigned int i, j, contador, maximo=0;
  char candidato=-1;

  for (i=0; i<talla; i++) {
    contador = 0;
    for (j=0; j<talla; j++)
      if (valores[i]==valores[j]) contador++;
    if (contador > maximo) {
      maximo = contador;
      candidato = valores[i];
    }
  }
  return candidato;
}
```

Si n es la talla de la lista, el coste de este método es $\Theta(n^2)$. (Completa tú los detalles del análisis.)

Hay una forma alternativa de calcular la moda y que resulta asintóticamente más eficiente:

```
def moda(valores):
  contador = [0] * 101
  for valor in valores:
    contador[valor] = contador[valor] + 1
  maximo = -1
  candidato = -1
  for i in range(101):
    if contador[i] > maximo:
      maximo = contador[i]
      candidato = i
  return candidato
```

Este método es $\Theta(n)$. (Nuevamente, completa tú los detalles del análisis.)

El primer programa está escrito en C y el segundo en Python. Pero el primero es cuadrático y el segundo lineal frente a cuadrático. ¿Cuál funcionará más rápidamente para valores de n suficientemente grandes?

14.5.8. Operaciones sobre listas enlazadas

Recuerda lo estudiado en el tema anterior sobre listas enlazadas. Esta tabla resume los costes de las diferentes operaciones que presentamos sobre listas enlazadas.

	simple	**simple c.c.**	**doble**	**doble c.c.**
Insertar por cabeza	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Borrar cabeza	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insertar por cola	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Borrar cola	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Insertar en una posición	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$
Buscar un elemento	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$	$\Omega(1)$ y $O(n)$

Te sugerimos que analices los programas que aparecen en dicho tema y compruebes si los costes de la tabla son correctos.

Fíjate en que, al trabajar con listas enlazadas, no podemos efectuar una búsqueda dicotómica para decidir si un elemento está o no en la lista, pues la búsqueda dicotómica se basa en la capacidad de indexar los vectores en tiempo $\Theta(1)$. En las listas enlazadas, acceder a un elemento cualquiera es $O(n)$.

Ejercicios

▶ **347** Deseamos utilizar listas enlazadas para implementar una pila. Las operaciones de una pila son `push` (inserción por cabeza), `pop` (borrado de la cabeza) y `top` (consulta del valor de cabeza). ¿Qué tipo de lista es más adecuado? Ten en cuenta el consumo de memoria a la hora de tomar una decisión.

▶ **348** Deseamos utilizar listas enlazadas para implementar una cola. Las operaciones de una pila son `encola` (inserción por el final) y `extrae` (extracción de la cabeza). ¿Qué tipo de lista es más adecuado? Ten en cuenta el consumo de memoria a la hora de tomar una decisión.

▶ **349** Determina, para cada uno de los cuatro tipos de lista enlazadas, el coste asintótico de cada una de las siguientes operaciones:

- Generar una lista que resulte de concatenar los elementos de dos listas de tamaño n y m, respectivamente.
- Añadir a una lista de talla n una copia de otra lista de talla m.
- Añadir a una lista de talla n otra lista de talla m (ojo: no una copia de la lista, sino la propia lista).
- Borrar de una lista de enteros todos los elementos de valor mayor que un valor dado.
- Intercambio de los dos últimos elementos de la lista.
- Invertir la lista (el último elemento pasa a la primera posición, el penúltimo a la segunda, etc.).

▶ **350** Describe con notación asintótica el coste temporal de esta rutina en tres supuestos diferentes:

```
def repetidos(lista):
  repes = []
  for i in range(lista):
    for j in range(lista):
      if lista[i] == lista[j] and i != j:
        repes.append(lista[i])
  return lista
```

Los supuestos son:

a) El método `append` es $\Theta(1)$.

b) El método `append` es $\Theta(n)$.

c) El método `append` es $\Omega(1)$ y $O(n)$.

14.6. Análisis del coste temporal en algoritmos recursivos

Los algoritmos recursivos son un poco más difíciles de analizar. Tomemos por caso un programa de cálculo recursivo del factorial:

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

¿Cómo calcular su coste? Sea $t(n)$ el coste del algoritmo. Podemos expresar la función de coste así:

$$t(n) = \begin{cases} 1, & \text{si } n = 0; \\ 1 + t(n-1), & \text{si } n > 0. \end{cases}$$

Es lo que denominamos una *ecuación recursiva*. La ecuación es similar al programa en tanto que presenta recursión en los mismos puntos en los que el programa efectúa llamadas recursivas. Pero $t(n)$ permite calcular el número de pasos ejecutados para calcular `factorial(n)` y `factorial(n)` es una función que calcula el factorial de n.

Hay un técnica, conocida por *desplegado*, que permite obtener una expresión no recursiva para $t(n)$. Consiste en ir expandiendo la expresión recursiva sustituyendo cualquier término de la forma $t(i)$ por su parte derecha:

$$\begin{aligned} t(n) &= 1 + t(n-1) \\ &= 1 + 1 + t(n-2) \\ &= 1 + 1 + 1 + t(n-3) \\ &= \cdots \\ &= \overbrace{1 + 1 + 1 + \cdots + 1}^{n \text{ veces}} + t(0) \\ &= \overbrace{1 + 1 + 1 + \cdots + 1}^{n+1 \text{ veces}} \\ &= n + 1 \end{aligned}$$

Como ves, el desplegado finaliza al llegar a un caso base. La única dificultad del método estriba en determinar cuántas expansiones hemos de aplicar para llegar a ese caso base.

La técnica del desplegado no constituye, por sí misma, una demostración de que $t(n)$ sea igual a $n + 1$, pues hemos efectuado un «salto» en la serie de igualdades sin más justificación que la que proporciona la intuición. Lo que hace la técnica es proporcionar una expresión para el coste que podemos demostrar ahora por inducción.

Ejercicios

► **351** Demuestra por inducción que $t(n) = n$. No te vendrá mal repasar los apuntes de Matemática Discreta.

Concluimos, pues, que $t(n) \in O(n)$.

14.6.1. Cálculo de a^n

Este programa C calcula a^n iterativamente para n entero:

```
double elevado(double a, int n)
{
  int i;
  double r;

  r = 1;
  for (i=0; i<n; i++)
    r *= a;
  return r;
}
```

El coste es, evidentemente, $\Theta(n)$.

Una translación directa de esta función iterativa a otra recursivo resulta en esta función:

```
double elevado(double a, int n)
{
  if (n==0)
    return 1;
  else
    return a * elevador(a, n-1);
}
```

El coste de este algoritmo recursivo es $\Theta(n)$. ¿Sabrías analizar tú mismo el coste?

Hay una idea potente que permite diseñar un algoritmo recursivo muy eficiente. Analicemos este otro programa, que efectúa el mismo cálculo recursivamente:

```
double elevado(double a, int n)
{
  int i;
  double r;

  if (n==0)
    return 1;
  if (n==1)
    return a;
  r = elevado(a, n/2);
  if (n % 2 == 0) {
    return r * r;
  else
    return a * r * r;
}
```

Fíjate en lo que hace. Cuando le piedes que solucione el problema de calcular a^n y n es par, lo divide en dos problemas más fáciles: a^n es $a^{n/2}$ por $a^{n/2}$; pero como sólo es necesario «perder tiempo» calculando $a^{n/2}$ una sola vez, entra en recursión para calcular $a^{n/2}$ y multiplica el resultado por sí mismo. Si n es impar, a^n es a por $a^{(n-1)/2}$ por $a^{(n-1)/2}$, así que calcula $a^{(n-1)/2}$recursivamente y multiplica el resultado por sí mismo y por a. Ya está. Es una estrtaegia inteligente: en lugar de solucionar un problema de talla n «reduciéndolo» a otro de talla $n-1$, lo «reduce» a otro que es la mitad de difícil. *Divide y vencerás*, así es como se conoce a esa estrategia de solución recursiva de problemas.

El coste temporal de la nueva función es

$$t(n) = \begin{cases} 4 + t(n/2), & \text{si } n > 1; \\ 3 & \text{si } n = 1; \\ 2 & \text{si } n = 0. \end{cases}$$

Apliquemos la técnica de desplegado. Supondremos que n es una potencia de 2 para que las sucesivas divisiones por 2 proporcionen resultados exactos:

$$\begin{aligned} t(n) &= 4 + t(n/2) \\ &= 4 + 4 + t(n/4) \\ &= 4 + 4 + 4 + t(n/8) \\ &= \cdots \\ &= \overbrace{4 + 4 + 4 + \cdots + 3}^{(\log_2 n) \text{ veces}} + t(n/2^{\log_2 n}) \\ &= \overbrace{4 + 4 + 4 + \cdots + 3}^{\log_2 n \text{ veces}} \\ &= 4 \log_2 n + 3. \end{aligned}$$

Este método es, pues, $\Theta(\log_2 n)$.

14.6.2. Números de Fibonacci

El procedimiento recursivo de cálculo de los números de Fibonacci se puede codificar así en Python:

```
def fibonacci(n):
  if n < 2:
    return 1
  else:
    return fibonacci(n-2) + fibonacci(n-1)
```

El coste $t(n)$ tiene una curiosa expresión recursiva:

$$t(n) = \begin{cases} 2 + t(n-1) + t(n-2), & \text{si } n \geq 2; \\ 2, & \text{si } n < 2. \end{cases}$$

El coste de calcular el n-ésimo número de Fibonacci está relacionado con el valor del n-ésimo número de Fibonacci. La fórmula de $t(n)$ es muy parecida a la propia fórmula de cálculo del n-ésimo número de Fibonacci:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & \text{si } n \geq 2; \\ 1, & \text{si } n < 2. \end{cases}$$

Más que resolver la ecuación recursiva de $t(n)$, vamos a acotarla superiormente:

$$\begin{aligned} t(n) &= 2 + t(n-1) + t(n-2) \\ &\leq 2 + 2 \cdot t(n-1) \\ &= 2 + 2 \cdot (2 + t(n-2) + t(n-3)) \\ &\leq 2 + 2 \cdot (2 + 2 \cdot t(n-2)) \\ &= 2 + 4 + 4 \cdot t(n-2) \\ &\leq \cdots \\ &\leq 2^1 + 2^2 + 2^3 + \cdots + 2^{n-1} + 2^{n-1} \cdot t(n-(n-1)) \\ &= \left(\sum_{i=1}^{n-1} 2^i \right) + 2^{n-1} \cdot t(1) \\ &= \left(\sum_{i=1}^{n-1} 2^i \right) + 2^{n-1} \cdot 2 \\ &= 2^n - 2 + 2^n \\ &= 2^{n+1} - 2. \end{aligned}$$

Así pues, calcular recursivamente los números de Fibonacci es $O(2^n)$. ¡Una cantidad de tiempo exponencial!

Esta otra función calcula iterativamente los números de Fibonacci[6]:

```
def fibonacci_iterativo(n):
  a, b = 1, 1
  for i in range(2, n):
    a, b = b, a + b
  return b
```

Su coste es $\Theta(n)$. Lineal frente a exponencial. Elige.

14.6.3. Comparación de algoritmos de ordenación

Acabamos el estudio de la complejidad temporal con el mismo problema con el que empezamos el tema: el problema de la ordenación.

Empecemos por analizar la complejidad temporal del método de la burbuja.

1	`def burbuja(valores):`	
2	`  nuevo = valores[:]`	$\Theta(n)$
3	`  for i in range(len(nuevo)):`	$\Theta(n)$
4	`    for j in range(len(nuevo)-1-i):`	$O(n)$, $\Theta(n)$ veces
5	`      if nuevo[j] > nuevo[j+1]:`	$\Theta(1)$, $O(n^2)$ veces
6	`        nuevo[j], nuevo[j+1] = nuevo[j+1], nuevo[j]`	$\Theta(1)$, $O(n^2)$ veces
7	`  return nuevo`	$\Theta(1)$

Evidentemente, el método de la burbuja es $O(n^2)$. ¿Puedes demostrar que es $\Theta(n^2)$?

El método *mergesort* es recursivo, así que presenta un análisis algo más complicado.

[6] Recuerda que en Python la asignación `a, b = 1, 2` asigna a `a` el valor `1` y a `b` el valor `2`.

1	`def mergesort(v, inicio, final):`		
2	`  global _aux`		
3	`  centro = (inicio+final)/2`	$\Theta(1)$	
4	`  if centro-inicio > 1:`	$\Theta(1)$	
5	`    mergesort(v, inicio, centro)`	$t(n/2)$	
6	`  if final-centro > 1:`	$\Theta(1)$	
7	`    mergesort(v, centro, final)`	$t(n/2)$	
8	`  # merge`		
9	`  for k in range(inicio, final):`	$\Theta(n)$	
10	`    _aux[k] = v[k]`	$\Theta(1)$,	$\Theta(n)$ veces
11	`  i = k = inicio`	$\Theta(1)$	
12	`  j = centro`	$\Theta(1)$	
13	`  while i <centro and j <final:`	$\Theta(1)$,	$O(n)$ veces
14	`    if _aux[i] <_aux[j]:`	$\Theta(1)$,	$O(n)$ veces
15	`      v[k] = _aux[i]`	$\Theta(1)$,	$O(n)$ veces
16	`      i += 1`	$\Theta(1)$,	$O(n)$ veces
17	`    else:`		
18	`      v[k] = _aux[j]`	$\Theta(1)$,	$O(n)$ veces
19	`      j += 1`	$\Theta(1)$,	$O(n)$ veces
20	`    k += 1`	$\Theta(1)$,	$O(n)$ veces
21	`  while i <centro:`	$\Theta(1)$,	$O(n)$ veces
22	`    v[k] = _aux[i]`	$\Theta(1)$,	$O(n)$ veces
23	`    i += 1`	$\Theta(1)$,	$O(n)$ veces
24	`    k += 1`	$\Theta(1)$,	$O(n)$ veces
25	`  while j <final:`	$\Theta(1)$,	$O(n)$ veces
26	`    v[k] = _aux[j]`	$\Theta(1)$,	$O(n)$ veces
27	`    j += 1`	$\Theta(1)$,	$O(n)$ veces
28	`    k += 1`	$\Theta(1)$,	$O(n)$ veces

Tenemos:

$$t(n) = t(n/2) + t(n/2) + \Theta(n)$$

El coste de `merge` es Θ de la suma de las tallas de las listas que funde, es decir, es $\Theta(n)$.

$$\begin{aligned} t(n) &= t(n/2) + t(n/2) + \Theta(n) \\ &= 2 \cdot t(n/2) + \Theta(n) \\ &= 2 \cdot (t(n/4) + t(n/4) + \Theta(n/2)) + \Theta(n) \\ &= 4 \cdot t(n/4) + 2 \cdot \Theta(n) \\ &= 8 \cdot t(n/8) + 3 \cdot \Theta(n) \\ &= \cdots \\ &= 2^k t(n/2^k) + (k-1) \cdot \Theta(n) \end{aligned}$$

El valor de k para el que $t(n/2^k)$ llega al caso base de la recursión es $k = \log_2 n$. Tenemos entonces:

$$\begin{aligned} t(n) &= n \cdot t(1) + (\log_2 n/2) \cdot \Theta(n) \\ &= \Theta(1) + (\log_2 n/2) \cdot \Theta(n) \\ &= (\log_2 n) \cdot \Theta(n). \end{aligned}$$

Podemos concluir que `mergesort` es $\Theta(n \log n)$, así que presenta un coste asintóticamente mejor el de la burbuja. Pudimos conjeturar que había diferencias de comportamiento asintótico entre el coste

temporal del algoritmo de la burbuja y el coste temporal de *mergesort* cuando efectuamos el estudio experimental.

Ejercicios

▶ **352** El método de selección busca el menor elemento de la lista y lo ubica en primer lugar. A continuación, busca el segundo menor elemento y lo ubica en el segundo lugar, y así sucesivamente. Esta función implementa en Python ese algoritmo de ordenación:

```
def ordena_seleccion(lista):
  for i in range(lista):
    min = lista[i]
    posmin = i
    for j in range(i+1, lista):
      if lista[j] < min:
        min = lista[j]
        posmin = j
    lista[posmin], lista[i] = lista[i], lista[posmin]
```

¿Qué coste temporal tiene? ¿Escogerías este algoritmo o `mergesort`?

Quicksort

Parece desprenderse del estudio comparativo que `mergesort` es el algoritmo a usar cuando deseamos ordenar un vector. Pero en la práctica se usa mucho más otro: Quicksort. ¿Es Quicksort $\Theta(n \log n)$, como `mergesort`? No. ¡Quicksort es $O(n^2)$! Aunque Quicksort presenta un coste cuadrático para el peor de los casos, en promedio es $O(n \log n)$ y, en la práctica, sensiblemente más rápido que *mergesort*.

Te esbozamos aquí una implementación del algoritmo Quicksort (buscando más legibilidad que una implementación óptima):

```
def quicksort(lista):
  if len(lista) <= 1:
    return lista
  menores = []
  mayores = []

  for i in lista[1:]:
    if i < lista[0]:
      menores.append(i)
    else:
      mayores.append(i)

  return quicksort(menores) + [lista[0]] + quicksort(mayores)
```

Quicksort es, quizá, el algoritmo sobre el que más trabajos se han realizado y publicado. Los estudios concluyen que resulta crítico, por ejemplo, escoger el elemento que se usa para dividir la lista en `menores` y `mayores` (en la implementación que te damos, `lista[0]`), pues el algoritmo se comporta de forma óptima cuando `len(menores)` es igual a `len(mayores)`. Una optimización frecuente y que tiene un fuerte impacto sobre la velocidad de ejecución consiste en usar ordenación por selección u otro algoritmo de ordenación cuando `len(lista)` es menor que, digamos, 10, pues se evita el sobrecoste de efectuar llamadas recursivas para listas muy pequeñas.

14.7. Complejidad espacial

La complejidad espacial es el estudio de la eficiencia de los algoritmos en lo relativo a su consumo de memoria. Un razonamiento similar al seguido con el coste temporal nos lleva a considerar únicamente

la evolución del consumo de memoria con la talla del problema. En el estudio asintótico no nos preocupa que un programa utilice la mitad o el doble de memoria que otro, pero sí que utilice una cantidad de memoria que crece con el cuadrado de n cuando otro requiere sólo una cantidad de memoria constante, por ejemplo. El concepto de «paso» utilizado en los análisis de coste temporal tiene su análogo en el estudio del coste espacial con el concepto de «celda de memoria». No importa si una variable ocupa 2, 4 u 8 bytes. Sólo importa si su tamaño es o no es función (y de qué orden) de n, la talla del problema.

Un análisis de la complejidad espacial, pues, obtiene cotas superiores e inferiores para el consumo de memoria.

El estudio de los algoritmos no recursivos es relativamente sencillo. Por ejemplo, si un algoritmo únicamente utiliza variables escalares, presenta un coste espacial constante. Si necesita vectores cuyo tamaño es proporcional a n, la talla del problema, el coste espacial es $\Theta(n)$.

Los algoritmos recursivos deben tener en cuenta, además, el espacio de pila consumido durante el cálculo. Aunque una rutina recursiva sólo utilice variables escalares, puede requerir espacio $\Theta(n)$ si efectúa del orden de n llamadas recursivas para solucionar un problema de talla n.