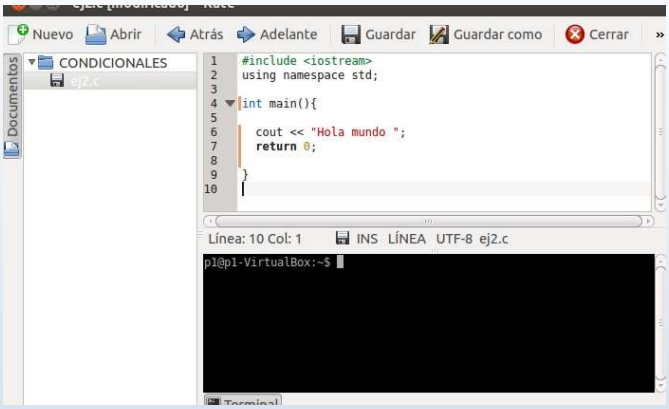


Programación 1

Tema 8. Evaluación del coste temporal de un algoritmo

Grado en Ingeniería Informática

Objetivos / Competencias

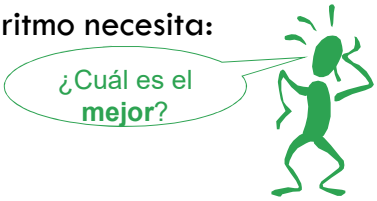
1. Conocer el concepto de eficiencia de un algoritmo
2. Saber analizar la complejidad temporal de un algoritmo
3. Aprender a utilizar la complejidad temporal como un criterio que guíe el diseño de una solución algorítmica

Índice

1. Eficiencia de un algoritmo
2. Métodos de cálculo de la eficiencia de un algoritmo
3. Coste temporal de un algoritmo
4. Análisis del coste mediante conteo de pasos
5. Ejercicios

Eficiencia de un algoritmo

- En ocasiones disponemos de varios algoritmos para resolver un determinado problema, ¿cuál utilizar?
- Un criterio para elegir un algoritmo u otro es la eficiencia de los mismos
- La eficiencia de un algoritmo tiene que ver con la cantidad de recursos que el algoritmo necesita:
  - Tiempo de ejecución
  - Espacio de almacenamiento



En esta asignatura, sólo vamos a estudiar la eficiencia de los algoritmos desde el punto de vista del tiempo de ejecución

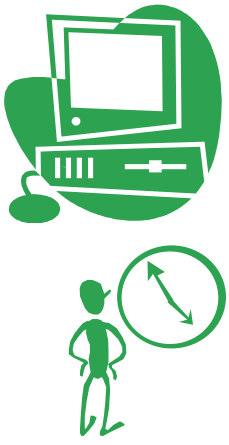
Tiempo de ejecución de un algoritmo

El tiempo de ejecución de un algoritmo depende de:

1. El tamaño de los datos a procesar
2. Velocidad del computador
3. Calidad del código generado por el compilador / intérprete

Método empírico o a posteriori

Consiste en programar los algoritmos y probarlos en un ordenador, midiendo el tiempo que consumen y el espacio que ocupan



Desventajas:

- No permite la comparación de algoritmos sobre diversos soportes
- Requiere el esfuerzo de programar cada uno de los algoritmos para determinar el mejor
- Sólo será posible comparar los tiempos para algunos tamaños del problema

Ejemplo del método empírico

Medición empírica de la eficiencia temporal de diferentes algoritmos de ordenación de *arrays*

Algoritmos de ordenación

tamaño del array

tiempo de ejecución en segundos

	10.000	50.000	100.000	500.000
Inserción	0,34	9,38	37,82	918
Selección	0,70	18,19	73,65	1740
Burbuja	0,88	22,24	98,27	2068
Shell Sort	0,01	0,04	0,06	0,32
Quick Sort	0,01	0,02	0,04	0,23
Heap Sort	0,01	0,02	0,05	0,27

Método analítico o a priori

Trata de calcular matemáticamente la cantidad de recursos que consume el algoritmo en función del tamaño del problema

Ventajas:

- Analiza el algoritmo y no el programa concreto $\Rightarrow$ resultado independiente de la máquina y del lenguaje
- Nos evitamos programar el o los algoritmos
- La expresión matemática que da el tiempo de ejecución es dependiente del tamaño del problema

Tamaño de un problema

El tamaño de un problema es cualquier parámetro en función del cual se puede expresar la complejidad del problema

- Generalmente guarda relación con el volumen de los datos de entrada a tratar

- Ejemplos:

Problema	Tamaño del problema
Ordenación de un array	**nº de elementos del array**
Búsqueda de un elemento en un array bidimensional	**nº de elementos del array (filas*columnas)**
Cálculo del factorial de un número	**el valor del número**

¿Cómo calcular la complejidad temporal de un algoritmo?

- **T(n)** : **Función del tamaño del problema** que proporciona el **tiempo de ejecución** que invierte una implementación de un algoritmo para un problema de tamaño ***n***

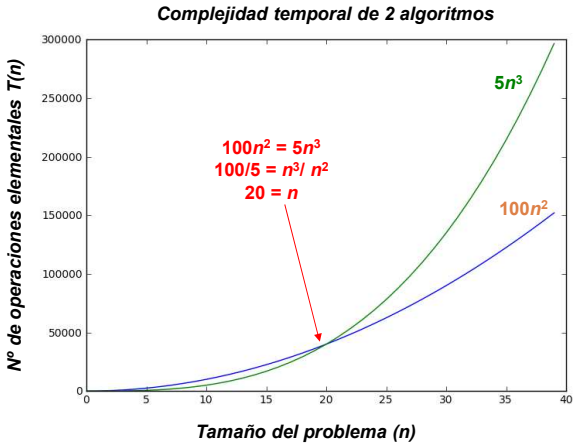
- T(n) se calculará contando el número de operaciones elementales o **pasos de programa** que realiza el algoritmo

- Por lo tanto, el resultado de T(n) es independiente de la máquina concreta utilizada

Ejemplo

¿Qué algoritmo es más eficiente para resolver un mismo problema, uno que tarda $T(n) = 100n^2$ o uno que tarda $T(n) = 5n^3$?

n	$100n^2$	$5n^3$
1	100	5
…	…	…
10	10000	5000
…	…	…
20	40000	40000
…	…	…
30	90000	135000
…	…	…
40	160000	320000
…	…	…



Análisis asintótico

- Es el estudio de la complejidad para tamaños muy grandes del problema
- Permite comparar el coste temporal de los algoritmos según el orden de magnitud de sus complejidades
- Se analiza la complejidad temporal independientemente de la velocidad del computador en donde se ejecuta una implementación del algoritmo
- Funciones de complejidad que difieren en un factor constante se consideran idénticas a efectos de medidas de coste temporal (**criterio asintótico**)

Orden de complejidad de un algoritmo

- Un algoritmo requiere un **tiempo de orden de T*(n)*,** si existe una constante positiva ***c*** y una implementación del algoritmo capaz de resolver todos los casos de tamaño ***n*** en un tiempo no superior a ***cT(n)***

- **T(n)** depende del algoritmo, ***c*** de la implementación

Ejemplo:

- Sea un algoritmo cuyo tiempo de ejecución ha resultado ser

$$T(n) = 32n^2 + 78n + 54$$

- Puesto que $(n \leq n^2)$ y $(1 \leq n^2)$ para todo $n \geq 1$, se cumple que

$$T(n) = 32n^2 + 78n + 54 \leq 32n^2 + 78n^2 + 54n^2 = 164n^2$$

- Este algoritmo tiene un tiempo de ejecución cuadrático, es decir, en el orden de n^2

Órdenes de complejidad según su eficiencia

Los órdenes más comunes ordenados de mayor a menor eficiencia son:

Orden	*T(n)*
logarítmico	**log(n)**
lineal	**n**
cuasi-lineal	**n log(n)**
cuadrático	**n^2**
polinomial (a > 2)	**n^a**
exponencial (a ≥ 2)	**a^n**
factorial	**n!**

Ejercicio

Disponemos de un tiempo de uso de 1.000 segundos para resolver un determinado problema en nuestro ordenador. Podemos ejecutar 4 algoritmos distintos cuyos tiempos de ejecución (*expresados en segundos*) son T(n)=100n, T(n)=$5n^2$, T(n) = $n^3/2$ y T(n) = 2^n

¿Cuál es el tamaño máximo del problema que pueden resolver cada uno de los algoritmos?

T(n)	n_1
100n	10
$5n^2$	14
$n^3/2$	12
2^n	10

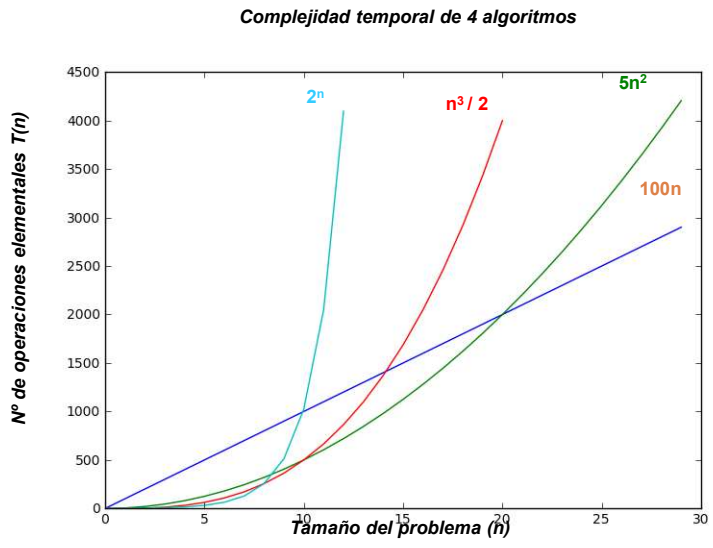
Si se incrementa la velocidad del computador 10 veces más, ¿Cuales serían los nuevos tamaños?

T(n)	n_2
100n	100
$5n^2$	45
$n^3/2$	27
2^n	12

Al aumentar la velocidad del computador ¿Cómo han aumentado los tamaños de los problemas que permitirían abordar los 4 algoritmos?

T(n)	n_2 / n_1
100n	10
$5n^2$	3,2
$n^3/2$	2,3
2^n	1,3

Coste temporal de distintos órdenes de complejidad

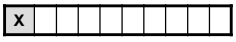


Caso peor, caso mejor y caso promedio

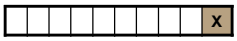
- Peor caso: se produce cuando es necesario que se efectúe el mayor número de operaciones elementales
- Mejor caso: se produce cuando es necesario que se efectúe el menor número de operaciones elementales
- Caso promedio: determina el número de operaciones elementales esperado que deben efectuarse (Requiere conocer la distribución de probabilidad de los datos de entrada)

Problema: buscar secuencialmente un elemento determinado en un array

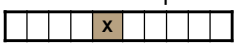
- Mejor caso: el elemento está en la primera posición (una comparación)



- Peor caso: el elemento está en la última posición o no está (n comparaciones)



- Caso medio: el elemento está en una posición intermedia (n/2 comparaciones)



Generalmente, y en particular para la asignatura de P1, en el método analítico del estudio de la complejidad de un algoritmo interesa conocer el **peor caso** y utilizar un criterio asintótico para establecer la complejidad temporal, expresándola en términos de **notación matemática O( )**

Pasos de programa

- Un **paso de programa** es una operación elemental que se lleva a cabo en un algoritmo

- Para **analizar el coste temporal** de un algoritmo, debemos calcular la función **T(n)** contando el número de pasos de programa que realiza dicho algoritmo

Operaciones elementales

- Consideraremos como operaciones elementales:
  - una sentencia de asignación 'S'

    Coste(S) = 1 paso
  - una sentencia de lectura 'S'

    Coste(S) = 1 paso
  - una sentencia de escritura 'S'

    Coste(S) = 1 paso
  - una sentencia de retorno del valor de una función

    Coste(return) = 1 paso
  - una expresión lógica (no incluida en la operaciones anteriores)

    Coste(condición) = 1 paso
  - una expresión aritmética (no incluida en la operaciones anteriores)

    Coste(expresión) = 1 paso

Pasos de programa de las sentencias de control

- secuencia de instrucciones: **{ S_1; S_2 }**

  Coste(S_1) + Coste(S_2)
- sentencia de selección: **if (condición) {S_1} else {S_2}**

  Coste(condición) + MAXIMO{Coste(S_1), Coste(S_2)}
- bucle con condición inicial: **while (condición) { S }**

  [Coste(condición) + Coste(S)] * nº de iteraciones + Coste(Condición)
- bucle con condición final: **do { S } while (condición)**

  [Coste(S) + Coste(condición)] * nº de iteraciones
- bucle controlado por contador: **for(inicializ; cond; incr) { S }**

  Coste(inic) + [Coste(cond)+Coste(S)+Coste(incr)]* nº iteraciones + Coste(cond)
- llamada a un módulo desde una operación elemental: **nombre_módulo()**

  Coste(módulo) + 1

Ejemplo 1: Calcular coste temporal

```
main() {
  int  a, n, c;

                          1
  cin >> n;               1
  a = 1;
  while (a <= n) {
          c=calcula(n);           1
          a = a+1;
  }
}

int calcula(int n) {
  int cal, i;
                          1
  cal = 1;
  for (i=1; i <= n; i++){ 1
    cal = cal * n;
  }                       1
  return(cal);
}
```

T(n) = Coste(main) = 2 + Coste(while) = 2+3n²+7n+1
=3n²+7n+3 → O(n²)

Coste(while) = [1+1+Coste(calcula)+1]*n+1 =

[1+1+3n+4+1]*n + 1 = 3n²+7n+1

Coste(calcula) = 2 + Coste(for) = 2+3n+2 = 3n+4

Coste(for) = 1+[1+1+1]*n +1 = 3n+2

Ejemplo 2 : Calcular coste temporal

```
int Busqueda_Binaria(int nom_array[], int elem)  {
  int     pos_inicio, pos_fin, pos_media;
  bool  encontrado;

  pos_inicio = 0;
  pos_fin = TAM_MAX -1;
  encontrado = false;
  while (pos_inicio <= pos_fin   &&   ! encontrado) {
       pos_media = (pos_inicio + pos_fin) / 2;
       if  (elem == nom_array[pos_media])
            encontrado = true;
       else if (elem > nom_array[pos_media] )
            pos_inicio = pos_media +1;
       else
            pos_fin = pos_media -1;
  }
  if (! encontrado)
     pos_media = -1;

  return(pos_media);
}
```

T(n) = Coste(Busqueda) = 4 + Coste(while) + Coste(if)
= 4+ 5log(n)+1 +2 = 5log(n) +7 → O(log(n))

Coste(while) = [1+(1+Coste(if-elseif)]*log(n)+1 =
[1+1+3]*log(n)+1 = 5log(n) +1

Coste(if-elseif) = 3

Coste(if) = 2

Ejercicio 1 : Calcular coste temporal

```
main() {
  int  n;

  do {
    cout << "Dame un número:";
    cin >> n;
    cout << "El resultado es " << Probando(n);
  } while (n > 0);
}

int Probando(int n) {
  int i, acu;

  acu = 0;
  i = 1;
  while (i <=n ) {
    if (n % i = = 0)
      acu = acu+1;
    i = i+2;
  }
  return(acu);
}
```

i=i+2 ➔ el bucle realiza n/2 iteraciones

Ejercicio 2: Calcular coste temporal

```
main()
{
  int fil, col, n, m;

  cout << "Introduce número de filas:";
  cin >> n;
  cout << "Introduce número de columnas:";
  cin >> m;
  for (fil=1; fil<=n; fil++) {
    for (col=0; col<m; col++) {
      if (fil = = 1  ||  fil = = n  ||  col = = 0 ||  col = = m-1)
        cout << "@";
      else
        cout << "*";
    }
    cout << endl;
  }
}
```

Ejercicio 3: Calcular coste temporal

```
void Multiplicar_Matrices(int m1[N][N], int m2[N][N], int mres[N][N])
{
  int  i, j, k;

  for (i=0; i < N; i++) {
    for (j=0; j < N; j++) {
      mres[i][j] = 0;
      for (k=0; k < N; k++) {
        mres[i][j] = mres[i][j] + (m1[i][k] * m2[k][j]);
      }
    }
  }
}
```

Ejercicio 4: Calcular coste temporal

```
main() {
  int  a, b, c;

  cin >> c;
  do {
    c = c-1;
    cin >> a;
    b = Nd(a);
    cout << "respuesta = ", b;
  } while (c > 0);
}

int Nd(int m) {
  res = 0;
  n = 0;
  while (n <=m) {
    n = n+1;
    res = res + 10;
  }
  return(res);
}
```

Ejercicio 5: Calcular nº de iteraciones del bucle

Teniendo en cuenta que n > 0

Bucle_1

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

Bucle_2

```
i = 0;
do {
    i = i+1;
} while (i < n);
```

Bucle_3

```
i = n;
do {
    i = i/2;
} while (i > 0);
```

Bucle_4

```
i = 2;
while (i <= 1) {
    i = i-1;
}
```

Bucle_5

```
i = n-1;
while (i > 1) {
    i = i-1;
}
```

Bucle_6

```
i = n;
while (i > 0) {
    i = i-2;
}
```

Bucle_7

```
for (i=0; i <= n; i++) {
    cout << endl;
}
```

Bucle_8

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

Bucle_9

```
for (i=1; i <= n; i--) {
    cout << endl;
}
```