9. INTRODUCCIÓN AL ESTUDIO DE ALGORITMOS Y SU COMPLEJIDAD

9.1. DEFINICIÓN DE COMPLEJIDAD Y SU MEDIDA 1
9.1.1. Introducción *1*
9.1.2. Concepto de talla de un problema *2*
9.2. RECUPERACIÓN DE INFORMACIÓN 3
9.2.1. Búsqueda secuencial *3*
9.2.2. Búsqueda dicotómica *6*
9.3. EL PROBLEMA DE LA ORDENACIÓN. MÉTODOS DE ORDENACIÓN INTERNA 7
9.3.1. Inserción *7*
9.3.2. Selección *9*
9.3.3. Intercambio (o burbuja) *10*
9.3.4. Quick-sort *11*
*Cuadro resumen** *12*
BIBLIOGRAFÍA ESPECÍFICA 12

9.1. Definición de complejidad y su medida

9.1.1. Introducción

Objetivo: Determinar qué algoritmo es mejor dentro de una familia de algoritmos que resuelven el mismo problema.

Se define coste o complejidad temporal de un algoritmo al tiempo empleado por éste para ejecutarse y a partir de unos datos de entrada obtener unos resultados.

Se define coste o complejidad espacial de un algoritmo al espacio ocupado en memoria (suma total del espacio que ocupan las variables del algoritmo) antes, durante y después de ejecutarlo.

A partir de la definición -> Problemas a la hora de evaluar la eficiencia de una manera objetiva.

Dependencias en la medida de estos costes con: el lenguaje de programación, la máquina en donde se ejecute, el compilador utilizado, etc. Además de depender de los datos de entrada (número de datos, valor de las variables iniciales, ...), la forma de realizar llamadas a otras funciones de librería, variables auxiliares (del propio lenguaje) ...

Intentaremos evitar estos problemas midiendo (o previendo) sobre el algoritmo (y las instrucciones contenidas en él) el tiempo que tardará en terminar la tarea. De manera que lo que vamos a medir no será un valor temporal exacto (en segundos) sino un valor estimado en unidades de tiempo que dependerán de la maquina, la implementación o el compilador, pero que para una misma máquina siempre serán las mismas.

Así y todo este coste será difícil de evaluar debido a que diferentes operaciones cuestan un tiempo diferente (i*i / i^2), existen llamadas a funciones de librería de las que a priori no conocemos el tiempo de ejecución, diferencias de tiempo para la misma operación con diferentes tipos de datos, acceso a perifericos, ...

Lo que intentaremos hacer será una estimación aproximada de los costes agrupando los tiempos de ejecución en grupos

→ Operaciones aritméticas: $\mathbf{t_o}$

→ Asignaciones: $\mathbf{t_a}$

→ Comparaciones: $\mathbf{t_c}$

Ejemplo: Realizar un algoritmo que calcule

$$y = \sum_{i=1}^{100} x$$

a.- Si realizamos el algoritmo, basándonos exclusivamente en el enunciado del problema, tendríamos:

```
y ← 0
i ← 1
Mientras (i ≤ 100) Hacer
    y ← y + x
    i ← i + 1
Fin_mientras
```

Si realizamos el estudio del número de asignaciones, operaciones y comparaciones que se realizan tendríamos:

y ← 0	$\mathbf{t_a}$		
i ← 1	$\mathbf{t_a}$		
Mientras (i ≤ 100) Hacer	$\mathbf{t_c}$		
y ← y + x	$\mathbf{t_o + t_a}$	$*100$	$\mathbf{= 100 * (t_c + 2t_o + 2t_a)}$
i ← i + 1	$\mathbf{t_o + t_a}$		
Fin_mientras	$\mathbf{t_c}$	*(Comparación para salir del bucle)*	

Que si sumamos nos dará:

$$\mathbf{Tiempo_1 = 2t_a + 100 * (t_c + 2t_o + 2t_a) + t_c = 202\ t_a + 202\ t_o + 101\ t_c}$$

b.- Si estudiamos el problema, podemos ver que sumar cien veces la 'x' es similar a multiplicar 'x' por 100, de manera que:

$$y = \sum_{i=1}^{100} x = 100 \cdot x$$

Y el algoritmo de resolución quedaría como sigue:

```
y ← 100 * x
```

Y el análisis nos daría:

$$\mathbf{Tiempo_2 = t_o + t_a}$$

Que es mejor tiempo que el obtenido para el primer algoritmo.

A partir de los valores obtenidos en el ejemplo, podemos determinar que el segundo algoritmo es 'mejor' que el primero, independientemente de la máquina que utilicemos o del compilador que tengamos.

9.1.2. Concepto de talla de un problema

A parte de la problemática debida al hecho de medir el tiempo de ejecución del algoritmo en función de las operaciones elementales, también existe el hecho de que el coste del algoritmo puede

depender de los datos de entrada del algoritmo. Diferentes datos de entrada pueden llevar a tiempos de ejecución distintos.

Ejemplo: Realizar un algoritmo que calcule

$$y = \sum_{i=1}^{n} i$$

En este caso el algoritmo tendrá como entrada el valor de 'n' y como salida el valor de 'y'

y ← 0	t_a		
i ← 1	t_a		
Mientras (i ≤ n) Hacer	t_c		
y ← y + i	$t_o + t_a$	$* n$	$= n * (t_c + 2t_o + 2t_a)$
i ← i + 1	$t_o + t_a$		
Fin_mientras	t_c	*(Comparación para salir del bucle)*	

Si al igual que antes sumamos los tiempos de las operaciones elementales, obtendremos:

$$Tiempo_1 = 2t_a + n * (t_c + 2t_o + 2t_a) + t_c$$

Evidentemente, el tiempo total que consumirá el algoritmo para obtener el resultado va a depender de el valor 'n'

Llamaremos ***talla*** de un problema al valor o conjunto de valores asociados a la entrada del problema y que representa una medida del tamaño del problema respecto de otras entradas posibles.

9.2. Recuperación de información

Uno de los principales problemas con los que nos tenemos que enfrentar cuando tenemos una gran cantidad de información es la búsqueda de un elemento concreto en el conjunto de datos.

Un problema típico en algoritmia es la búsqueda. Existen dos métodos básicos de búsqueda en un conjunto: La búsqueda secuencial y la búsqueda dicotómica.

9.2.1. Búsqueda secuencial

La búsqueda secuencial se aplica cuando no existe ningún conocimiento previo sobre la ordenación de los elementos del conjunto en donde se va a realizar la búsqueda.

La búsqueda secuencial se basa en ir recorriendo uno a uno los elementos del conjunto en busca del elemento deseado.

La idea general del algoritmo sería la siguiente:

1. Suponemos que no hemos encontrado el elemento.
2. Desde el primer elemento hasta el último elemento del conjunto

   2.a. Comprobamos si el elemento que buscamos es el que estamos comprobando del conjunto

   Si lo es la suposición inicial es falsa y si que hemos encontrado el elemento.

Basado en este algoritmo podemos escribir una función en C++ que nos devuelva si un elemento 'x' está o no entre un conjunto de valores guardados en un vector 'v'.

```
/*
 * La constante la fijamos correctamente en nuestro problema
 */
const int TAM = .??.;

typedef int Vector [TAM];

bool BusquedaSecuencial (Vector v, int n, int x)
{
   int i;
   bool enc = false;

   for (i = 0; i < n; i++)
      if (v[i] == x)
         enc = true;

   return enc;
}
```

O lo que sería lo mismo cambiando el bucle '**for**' por un bucle '**while**':

```
const int TAM = .??.;

typedef int Vector [TAM];

bool BusquedaSecuencial (Vector v, int n, int x)
{
   int i;
   bool enc = false;

   i = 0;
   while (i < n)
   {
      if (v[i] == x)
         enc = true;

      i = i + 1;
   }

   return enc;
}
```

Si estudiamos el algoritmo (sólo la parte esencial, es decir la parte que realiza la búsqueda):

`enc = false;`	t_a	
`i = 0;`	t_a	
`while (i < n) {`	t_c	$\}*n$
`  if (v[i] == x)`	t_c	
`  enc = true`	t_a	
`  i = i + 1;`	$t_o + t_a$	
`}`	t_c	*(Comparación para salir del bucle)*

Podemos ver que dependiendo de los valores de 'x' y de los elementos contenidos dentro del vector, puede que se realicen más o menos pasos en la ejecución del algoritmo (entraremos en la asignación contenida en el '**if**' o no entraremos), y que el tiempo total de ejecución no depende sólo de la talla del problema o de la implementación del algoritmo, sino que depende de los parámetros con que se llame a la función.

En este punto es donde aparece el concepto de mejor caso, peor caso y caso medio. Diremos que estamos en el mejor caso, cuando los parámetros del problema nos lleven a realizar el menor número de pasos posibles, y al tiempo que tarda el algoritmo en resolver el problema lo representaremos por T^m. Diremos que estamos en el peor de los casos cuando los parámetros del problema nos lleven a realizar el máximo número de pasos posibles, y al tiempo que tarda el algoritmo en resolver el problema lo representaremos por T^p. El caso medio vendrá expresado por la media de pasos en función de todos los casos posibles (básicamente se realizará una estadística de los posibles casos y se hará la media de todos estos casos estadísticos). El tiempo medio lo representaremos por T^m.

En el caso que nos ocupa, el mejor de los casos será cuando en ningún momento entremos en el '**if**' es decir, el elemento 'x' no se encuentra en el vector. En ese caso tendremos, sumando los valores de cada una de las líneas, excepto la correspondiente a la asignación:

$$T^m = 2t_a + n * (2t_c + t_o + t_a) + t_c$$

Agrupando términos en función de la dependencia que tienen en 'n' obtendríamos:

$$T^m = (2t_c + t_o + t_a) * n + (t_c + 2t_a) = A * n + B$$

Siendo **A** y **B** constantes independientes de la talla.

El peor de los casos sería el vector que hiciese que siempre pasasemos por la asignación (vector compuesto totalmente por el valor 'x' buscado.) En esta situación el valor obtenido sería:

$$T^p = 2t_a + n * (2t_c + t_o + 2t_a) + t_c$$

Al igual que antes, agrupando términos en función de la dependencia que tienen en 'n', obtendremos:

$$T^p = (2t_c + t_o + 2t_a) * n + (t_c + 2t_a) = A' * n + B$$

La diferencia básica entre un caso y otro, estriva en las constantes **A** y **A'** (**A'** > **A**) aunque el tiempo sigue siendo función lineal de la talla del problema.

Búsqueda secuencial con parada

Este algoritmo es susceptible de ser mejorado: Si en un momento dado hemos encontrado el elemento buscado, ya no tiene sentido seguir la búsqueda en el resto del vector. Con esta modificación, al salir del bucle deberemos comprobar por qué motivo hemos salido del bucle: Si porque no hemos encontrado el elemento o porque hemos llegado al final del vector. Así, la parte del la función que realiza la búsqueda quedaría como sigue:

```
i = 0;
while ( (i < n) && (v[i] != x) )
   i = i + 1;

if (i == n)
   enc = false;
else
   enc = true;
```

Búsqueda secuencial con centinela

En este algoritmo vemos que una de las condiciones que hay que comprobar es que el índice permanezca en el rango adecuado (es decir, entre **0** y **n-1**) y la búsqueda no siga fuera de este rango. Esta comprobación no sería necesaria si estuviesemos seguros de encontrar el elemento.

Podemos estar seguros de encontrar el elemento, si en un momento determinado lo ponemos en una posición del vector, por ejemplo, en la posición '**n**'.

Con ello, la condición de que el índice permanezca en el vector no es necesaria, y el algoritmo tendría una comparación menos cada vez que se pasa por el bucle.

```
v[n] = x;
i = 0;
while (v[i] != x)
   i = i + 1;

if (i == n)
   enc = false;
else
   enc = true;
```

Búsqueda secuencial con vector ordenado

Si tenemos información adicional sobre los elementos contenidos en el vector, podemos aprovechar esta información para mejorar el algoritmo de búsqueda.

Si sabemos que el vector está ordenado, si empezamos la búsqueda desde el primer elemento, una vez superemos un elemento mayor que el buscado, ya no tiene sentido seguir buscando por el resto del vector. Si además añadimos el centinela, la búsqueda quedaría como sigue:

```
v[n] = x;
i = 0;
while (v[i] < x)
   i = i + 1;

if ( (i == n) || (v[i] != x) )
   enc = false;
else
   enc = true;
```

9.2.2. Búsqueda dicotómica

El hecho de que el vector esté ordenado, se puede aprovechar mejor.

Si en vez de empezar la búsqueda desde el primer elemento, comenzamos la búsqueda desde el centro del vector, podemos determinar, si no hemos encontrado el elemento, en que mitad del vector se encuentra el elemento, descartando de la búsqueda la otra mitad. Aplicando sucesivamente a las mitades del vector donde es posible encontrar el elemento, llegaremos o a un subvector de un solo elemento o al elemento buscado.

El algoritmo de búsqueda quedaría como sigue:

```
/*
 * La constante la fijamos correctamente en nuestro problema
 */
const int TAM = .??.;

typedef int Vector [TAM];

bool BusquedaDicotomica (Vector v, int n, int x)
{
   int izq, der, cen;
   bool enc = false;

   izq = 0;
   der = n – 1;
   cen = (izq + der) / 2;
```

```
   /*
    * Mientras no encontremos el elemento en el centro del subvector
    * y ademas, existan mas de dos elementos en el subvector continuamos
    * la búsqueda
    */
   while ( (izq <= der) && (v[cen] != x) )
   {
      if (x < v[cen])
         der = cen - 1;
      else
         izq = cen + 1;
      cen = (izq + der) / 2;
   }
   /*
    * Cuando salimos del bucle, podemos salir o por haber encontrado el
    * elemento o por haber llegado a un subvector de un solo elemento
    */
   if (izq < der)
      enc = false;
   else
      enc = true;

   return enc;
}
```

En este algoritmo en el mejor de los casos el elemento a buscar estaría situado en el centro del vector y el coste no dependería de la cantidad de elementos contenidos en el vector. Tendríamos para el mejor de los casos un coste constante.

En el peor de los casos tendríamos una división sucesiva del vector en mitades, hasta llegar a una mitad de un solo elemento. El número de veces que es posible dividir por dos un valor 'n' es $\mathbf{lg_2 n}$. En informática siempre que se hable de logaritmos, hablaremos de logaritmos en base dos, de manera que, en el peor de los casos, hablaremos de un coste logarítmico.

En el caso medio el cote también es logarítmico, pero la constante que afecta al termino es la mitad que en el caso anterior.

9.3. El problema de la ordenación. Métodos de ordenación interna

9.3.1. Inserción

El algoritmo de inserción se basa en la idea de ir insertando en la parte ordenada del vector, uno a uno, los elementos de la parte desordenada. De esta manera el algoritmo general que nos ordenaría un vector 'vec' de 'n_ele' elementos sería el siguiente:

```
Desde i ← 1 Hasta n Hacer
  InsertarEnLaParteOrdenada (v[i])
Fin_desde
```

Y la inserción, considerando que la parte ordenada esta formada por los '**i-1**' primeros elementos, sería básicamente:

```
aux ← vec[i]
k ← (i - 1)
Mientras (k ≥ 0) && (aux < vec[k]) Hacer
      vec[k + 1] ← vec[k]
      k ← k - 1
Fin_mientras
vec[k + 1] ← aux
```

Con estas consideraciones la función en C++ quedaría, considerando, al igual que en la búsqueda, que sólo una parte del vector contiene información 'relevante':

```
void OrdenarInsercion (Vector v, int n)
{
   int i, j;
   int i_aux;

   for (i = 1; i < n; i++)
   {
      i_aux = v[i];
      j = i - 1;
      while ( (j >= 0) && (v[j] > i_aux) )
      {
         v[j + 1] = v[j];
         j = j - 1;
      }
      v[j + 1] = i_aux;
   }
}
```

Ordenación por inserción con centinela

La idea es la misma que en la búsqueda secuencial con centinela. Si seguro que encontramos una posición dentro del vector para el nuevo elemento, no es necesario comprobar que el índice no se sale del rango válido. En este caso, el elemento hay que situarlo al inicio del vector, y la comprobación '**k** ³ **0**' no es necesaria:

```
Desde i ← 1 Hasta n Hacer
     vec[-1] ← aux
     k ← (i - 1)
     Mientras (aux < vec[k]) Hacer
           vec[k + 1] ← vec[k]
           k ← k - 1
     Fin_mientras
     vec[k + 1] ← vec[-1]
Fin_desde
```

Esto, en C++ es incorrecto. No podemos utilizar la posición '-1'.

La forma correcta de utilizar el centinela, sería desplazando los elementos una posición a la izquierda para poder 'abrir' un hueco para el centinela, y, una vez ordenado el vector, desplazar la información a su lugar original.

```
void OrdenarInsercionCentinela (Vector v, int n)
{
   int i, j;
   int i_aux;

   for (i = 0; i < n – 1; i++)
      v[i + 1] = v[i];

   for (i = 1; i < n; i++)
   {
      j = i - 1;
      /* recordar que la utilización de la posición '-1' es incorrecta */
      v[0] = v[i];
```

```
      while (v[j] > v[-1])
      {
         v[j + 1] = v[j];
         j = j - 1;
      }
      v[j + 1] = v[0];
   }

   for (i = 0; i < n – 1; i++)
      v[i] = v[i + 1];
}
```

Este trabajo adicional es lineal, mientras que la ordenación, en el caso general, es cuadrática, de manera que el coste del algoritmo aumenta, pero el orden de complejidad sigue siendo cuadrático.

De todas formas estos desplazamientos son fácilmente evitables si la ordenación la realizamos suponiendo la parte ordenada a la derecha de la parte desordenada (ordenamos desde arriba hacia abajo el vector). Con este cambio, el centinela quedaría en la posición 'n', y no tendríamos que desplazar innecesariamente los valores para insertar el centinela.

Con esta modificación el algoritmo resultante será el siguiente:

```
void OrdenarInsercion (Vector v, int n)
{
      int i, j;

      for (i = n-2; i>=0; i--)
      {
            v[n] = v[i]; //centinela
            j = i+1;
            while ( v[j] < v[n] )
            {
                  v[j-1] = v[j];
                  j++;
            }
            v[j-1] = v[n];
      }
}
```

9.3.2. Selección

El algoritmo de selección se basa en la idea de ir seleccionando para cada una de las posiciones del vector el elemento que debería estar en esa posición (en la primera posición el menor, en la segunda el menor de los restantes y así sucesivamente hasta el último que ya está en su lugar.)

```
Desde i ← 0 Hasta n – 1 Hacer
      pos_min ← BuscarElMinimoDeLosDesordenados
      Si (pos_min ≠ i) Entonces
            v[i] ↔ v[pos_min]
      Fin_si
Fin_Desde
```

La búsqueda del mínimo es fácil:

```
pos_min ← i
Desde j ← i + 1 Hasta n Hacer
      Si (v[j] < v[pos_min]) Entonces
            pos_min ← j
      Fin_si
Fin_desde
```

La función en C++ que realizaría esta ordenación sería:

```
void OrdenarSeleccion (Vector v, int n)
{
   int i, j, pos_min;
   int i_aux;

   for (i = 0; i < n - 1; i++)
   {
      pos_min = i;
      for (j = i + 1; j < n; j++)
         if (v[j] < v[pos_min])
            pos_min = j;

      if (pos_min != i)
      {
         i_aux = v[i];
         v[i] = v[pos_min];
         v[pos_min] = i_aux;
      }
   }
}
```

9.3.3. Intercambio (o burbuja)

La idea básica de este algoritmo es ir comparando dos a dos los elementos y situarlos en el orden correcto dentro de la secuencia. Si imaginamos el vector en posición vertical, el método consiste en llevar los elementos más ligeros (los que contienen claves de ordenación más pequeñas) hacia arriba, o al contrario llevar los elementos más pesados hacia abajo. Todo depende de la dirección en que se hacen las comparaciones. En cualquier caso, en cada pasada un elemento se compara con el siguiente, situandolos en el orden adecuado que deberían tener según sus pesos. Al finalizar la pasada i-esima el elemento 'i' quedará en la posición adecuada según su peso.

Si hacemos las comparaciones en forma ascendente tendremos el algoritmo que va subiendo los elementos más ligeros:

```
void OrdenarBurbuja (Vector v, int n)
{
   int i, j;
   int i_aux;

   for (i = 1; i < n - 1; i++)
   {
      for (j = n - 1; j > i - 1; j--)
      {
         if (v[j - 1] > v[j])
         {
            i_aux = v[j];
            v[j] = v[j + 1];
            v[j + 1] = i_aux;
         }
      }
   }
}
```

Si, por el contrario, hacemos las comparaciones en forma descendente tendremos el algoritmo que va bajando los elementos más pesados:

```
void OrdenarBurbuja (Vector v, int n)
{
   int i, j;
   int i_aux;

   for (i = 1; i < n - 1; i++)
   {
      for (j = 0; j < n - i - 1; j++)
      {
         if (v[j] > v[j + 1])
         {
            i_aux = v[j];
            v[j] = v[j + 1];
            v[j + 1] = i_aux;
         }
      }
   }
}
```

9.3.4. Quick-sort

En este algoritmo se trata de ir agrupando los elementos en dos subgrupos: Un grupo de elementos 'pequeños' y otro grupo de elementos 'grandes' respecto de una referencia, que llamaremos pivote, y que será un elemento cualquiera del vector (generalmente el elemento central del vector). Una vez tenemos los dos subconjuntos, volvemos a aplicar la misma idea a cada uno de los subconjuntos por separado, hasta obtener subconjuntos de un solo elemento.

```
void OrdenarQuickSort (Vector v, int n)
{
   QuickSortRec (v, 0, n - 1);
}

void QuickSortRec (Vector v, int izq, int der)
{
   int i, j, i_aux, piv;

   piv = v[(izq + der) / 2];
   i = izq;
   j = der;

   while (i <= j)
   {
      while (v[i] < piv)
         i++;
      while (v[j] > piv)
         j--;

      if (i < j)
      {
         i_aux = v[i];
         v[i] = v[j];
         v[j] = i_aux;
         i++;
         j--;
      }
      else
         if (i == j)
         {
            i++;
            j++;
         }
```

```
        }

        if (izq < j)
           QuickSortRec (v, izq, j);
        if (i < der)
           QuickSortRec (v, i, der);
    }
```

*Cuadro resumen**

	Asignaciones			comparaciones		
	Mejor	Medio	peor	mejor	medio	peor
Inserción						
Selección						
Burbuja						
quick-sort						

**Rellenar con lo visto en clase. Recordar que se pueden tener en cuenta todas las operaciones involucradas en el algoritmo, o sólo las operaciones que involucren elementos del vector.*

Bibliografía específica

- "*Introducció a l'anàlisi i disseny d'algorismes*"
  Francesc J. Ferri, Jesús V. Albert, Gregorio Martín. Ed. Universitat de València. 1998.
- "*Estructuras de datos y algoritmos*"
  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Ed. Addison Wesley Iberoamericana. 1988.