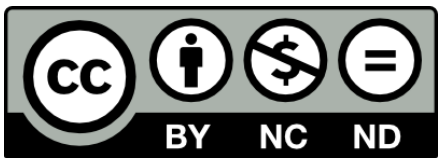
Resumen de CLRS

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Resumen de CLRS

Los Del DGIIM, losdeldgiim.github.io

José Juan Urrutia Milán

Granada, 2025

Índice general

1. Introducción

1.1. Comenzando

1.1.1. Insertion Sort

Insertion Sort resuelve el problema típico de ordenación:

Dado un vector de n elementos: $(a_1, \ldots, a_n)$, buscamos generar una permutación del mismo: $(b_1, \ldots, b_n)$ de forma que:

$$b_1 \leqslant b_2 \leqslant \ldots \leqslant b_{n-1} \leqslant b_n$$

El algoritmo sigue la filosofía que seguimos los humanos para ordenar una mano de cartas que vamos cogiendo de una mesa; es decir, imaginemos que tenemos un mazo de cartas boca abajo en la mesa, y que tenemos que ir cogiendo cartas de una en una que vamos ordenando en nuestra mano, que inicialmente está vacía.

Inicialmente no tenemos cartas en la mano y cogemos una del mazo. Ahora, nuestra mano de cartas está ordenada, ya que solo contiene una carta; seguimos cogiendo otra carta, que procedemos a colocar antes o después de la nuestra en función del número de la carta. Nuestra mano vuelve a estar ordenada. Repetiremos este proceso hasta que todo el mazo que hay sobre la mesa esté ya ordenado.

Podemos programar este procedimiento en `C++`, pensando que recibimos como parámetros:

- `v`, vector de enteros.
- `n`, el tamaño del vector.

```
1  void insertion_sort(int* v, int n){
       for(int i = 1; i < n; i++){
           int key = v[i];

5          // Inserta key en la posición adecuada
           int h = i-1;
           while(h >= 0 && v[h] > key){    // Mientras que el anterior sea mayor
               v[h+1] = v[h];  // Lo rota a la derecha
               h--;
10         }
           v[h+1] = key;       // Coloca la clave en el sitio correcto
       }
   }
```

Análisis

Como podemos ver en el código, el bucle de la línea 2 se ejecuta un total de $n-1$ veces, por lo que el tiempo de ejecución del algoritmo tendrá un factor n. Por otra parte, el bucle `while` de la línea 7 no tiene un número de iteraciones fijado, sino que depende de la entrada del algoritmo: del vector `v`. Veamos varios ejemplos:

- Si el vector `v` está previamente ordenado antes de la entrada al algoritmo, entonces la condición `v[h] > key` será falsa tras la inicialización `h = i-1`, por lo que el bucle `while` no se ejecutará, con lo que el tiempo de ejecución del algoritmo es de la forma:
$$an + b$$
para ciertas constantes a, b, c. Este es el *mejor caso* de ejecución del algoritmo, puesto que es el que menos iteraciones del bucle `while` provoca, dando un menor tiempo de ejecución.

- Si el vector `v` está ordenado en orden decreciente, entonces el bucle `while` se ejecuta el mayor número de veces posible, es decir, un total de `i` veces en la iteración `i`-ésima del bucle `for` exterior, de donde:
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$
Por lo que la ejecución del bucle `while` anidado en el `for` hace que el algoritmo se ejecute en un tiempo de ejecución de la forma:
$$an^2 + bn + c$$
Este caso es el *peor caso* de ejecución del algoritmo, puesto que es el que más iteraciones del bucle `while` provoca, maximizando el tiempo de ejecución.

- Si el vector `v` es uno cualquiera. Para calcular el tiempo de ejecución, podemos pensar que, de media, la mitad de las veces se cumplirá que la mitad de los elementos de la parte ya ordenada del vector serán mayores que el elemento `key` y que la otra mitad serán menores que el elemento `key`, por lo que podemos pensar que el bucle `while` realizará $\mathtt{i}/2$ iteraciones en la iteración `i`-ésima del bucle `for`. Si realizamos las cuentas obtendremos un tiempo muy similar al del peor caso:
$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{4}$$
Obteniendo un tiempo de la forma
$$an^2 + bn + c$$
Llamaremos a este caso *caso promedio*.

En la mayoría de los casos no calcularemos estos tres tipos de tiempos, sino que simplemente calcularemos el tiempo del peor caso de ejecución, ya que:

- Este tiempo acota cualquier ejecución del programa, por lo que nos da una estimación máxima del tiempo de ejecución que este requiere.
- El peor tiempo de ejecución se da en algunos casos de manera muy frecuente. Por ejemplo, a la hora de buscar un dato en una base de datos, el peor tiempo de ejecución se da cuando el dato no está presente en la base de datos, ya que obliga a comprobar todos los datos de la base de datos; y que un dato no se encuentre en una base de datos puede resultar muy frecuente en algunos casos.
- En muchas ocasiones el tiempo de ejecución del caso promedio es muy similar (o del mismo orden) que el tiempo de ejecución del peor caso, tal y como sucede con *Insertion Sort*.

Aunque anteriormente dijimos que el tiempo de ejecución de *Insertion Sort* es de la forma

$$an^2 + bn + c$$

en la mayoría de los casos nos quedaremos con el sumando de dicha expresión que más rápido crezca conforme n crece (en este caso, an^2), puesto que su aportación a la suma es el que más importa para tamaños de n suficientemente grandes. Además, nos olvidaremos de las constantes que acompañan a dicho sumando (es decir, pensaremos simplemente en n^2), puesto que el valor de dichas constantes es de menor relevancia en comparación con el "orden" de creciemiento.

Tomando estas consideraciones, diremos que el tiempo de ejecución de *Insertion Sort* en el peor caso es del orden de n^2.

1.1.2. Selection Sort

Este algoritmo también resuelve el problema típico de ordenación, pero de otra forma distinta a como lo hacía *Insertion Sort*. Para ello, lo que hace el algoritmo es, dado un vector de elementos a ordenar, busca el mínimo del vector y lo intercambia con el elemento que ocupa la primera posición. Una vez hecho esto, busca el segundo elemento mínimo del vector y lo intercambia con el elemento que ocupa la segunda posición. El procedimiento se repite hasta buscar el segundo mayor elemento del vector, que se intercambia con el que ocupa la penúltima posición del vector.

Bajo las mismas precondiciones que el algoritmo para *Insertion Sort*, una posible implementación de *Selection Sort* es:

```
// Devuelve el indice del primer menor elemento
// comprendido en [start, start + end)
int minimo(int* v, int start, int end){
    int i = start;
    for(int j = start+1; j < start + end; j++){
        if(v[j] < v[i]){
            i = j;
        }
    }
    return i;
}

void selection_sort(int* v, int n){
```

```
    for(int i = 0; i < n-1; i++){
15      // Busca el minimo
        int h = minimo(v, i, n-i);
        // Lo intercambia
        swap(v[i], v[h]);
    }
20  }
```

Análisis

El bucle `for` de la línea 14 realiza $n-1$ iteraciones independientemente de la entrada `v`. Además, cada llamada al procedimiento `minimo(v,i,n-i)` realiza unas `n-i-1` número de iteraciones, por lo que el tiempo de ejecución del algoritmo será:

$$\sum_{i=0}^{n-2} n-i-1 = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 = n(n-1) - \frac{(n-1)(n-2)}{2} - (n-1)$$

Que podemos expresarlo en la forma

$$an^2 + bn + c$$

para ciertas constantes a, b, c, por lo que el tiempo de ejecución de *Selection Sort* es del orden n^2, tanto en el mejor como en el peor caso (no hemos hecho ninguna asunción sobre la entrada, puesto que el número de iteraciones de los bucles no depende de dicha entrada, sino que son fijos).

Si comparamos *Insertion Sort* con *Selection Sort* vemos que el orden de ejecución de los algoritmos es el mismo en los casos peor y promedio, por lo que el tiempo de ejecución de ambos será parecido para valores grandes de n y la diferencia entre ellos dependerá generalmente de los valores de las constantes que estamos despreciando al analizar los órdenes.

Por otra parte, en el mejor caso de ejecución, *Insertion Sort* tiene un tiempo de ejecución lineal, por lo que en dicho caso es preferible frente a *Selection Sort*. Por esta última razón, puede ser preferible usar *Insertion Sort* frente a *Selection Sort*, puesto que su tiempo de ejecución puede reducirse en relación a la distribución de los valores de entrada.

Comentamos finalmente que aunque sobre la teoría esto se vea muy claro, en la práctica habría que medir la eficacia de *Insertion Sort* frente a *Selection Sort*, puesto que sus tiempos promedios dependen del ámbito de uso de los mismos.

1.1.3. Diseñando algoritmos: Divide y vencerás y Merge Sort

Algunos problemas tienen una estructura recursiva, que nos permite aplicar la estructura divide y vencerás. Esta estructura de diseño de algoritmos se basa en 3 pasos que se realizan en cada nivel de recursión:

- **Dividir** el problema en otros varios de la misma naturaleza pero de menor tamaño.

- **Vencer** el problema de forma recursiva. Si el tamaño del problema es suficientemente pequeño, resolverlo directamente por el primer algoritmo que se nos ocurra.
- **Combinar** la soluciones de los distintos subproblemas para obtener la solución general al problema..

El algoritmo *Merge Sort* sigue esta forma de estructura de un algoritmo:

- **Divide** la secuencia de n datos a ordenar en dos subsecuencias de tamaño $n/2$ cada una.
- **Vence** cada uno de los subproblemas de forma recursiva.
- **Combina** las dos subsecuencias ya ordenadas en una secuencia final ordenada.

La recursión se acaba cuando la secuencia a ordenar es de tamaño 1, en cuyo caso ya tenemos una secuencia ordenada. Comenzamos describiendo la parte del **combinado** de las soluciones, que puede llevarse a cabo por la siguiente función:

```
// Siendo p <= q < r:
// Supuesto que v[p,q] y v[q+1,r] son dos subvectores ordenados
// Genera la combinacion de ellos en el vector v[p,r], ordenado
void merge(int* v, int p, int q, int r){
    long long int MAX = 9999999999999;

    int n_1 = q-p+1;   //nº de elementos en v[p,q]
    int n_2 = r-q; //nº de elementos en v[q+1,r]

    // Se crean copias de v[p,q] y v[q+1,r]
    int left[n_1+1];
    int right[n_2+1];
    for(int i = 0; i < n_1; i++){
        left[i] = v[p+i];
    }
    left[n_1] = MAX;
    for(int j = 0; j < n_2; j++){
        right[j] = v[q+1+j];
    }
    right[n_2] = MAX;

    // Se juntan
    int i = 0;
    int j = 0;
    for(int k = p; k <= r; k++){
        if(left[i] <= right[j]){
            v[k] = left[i];
            i++;
        }else{
            v[k] = right[j];
            j++;
        }
    }
}
```

Este algoritmo tarda un tiempo de orden de n, donde $n = r - p + 1$, puesto que los bucles de copia en `left` y `right` son despreciables respecto al último bucle `for`.

Una vez discutido este proceso, estamos en condiciones de crear el proceso `merge_sort`, que dado un vector y dos posiciones `p` y `r` con $p \leqslant r$, ordenará dicho vector. El proceso será el siguiente: Si $p \geqslant r$, entonces la secuencia ya está ordenada por contener un elemento ($p = r$). En caso contrario, se calcula un índice `q` de forma que al final tengamos:

- Una secuencia `v[p,q]` conteniendo $\lceil n/2 \rceil$ elementos.
- Una secuencia `v[q+1,r]` conteniendo $\lfloor n/2 \rfloor$ elementos.

De esta forma, el procedimiento tendrá el siguiente código:

```
// Siendo p <= r, ordena el vector v[p,r]
// Si p >= r, el vector ya está ordenado
void merge_sort(int* v, int p, int r){
    if(p < r){
        int q = (p+r)/2;    // se coge el suelo
        merge_sort(v, p, q);
        merge_sort(v, q+1, r);
        merge(v, p, q, r);
    }
}
```

El tiempo de ejecución de un algoritmo del tipo divide y vencerás suele ser de la siguiente forma, donde $T(n)$ indica el tiempo que tarda el algoritmo para resolver un problema de tamaño n:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leqslant c \\ aT(n/b) + D(n) + C(n) & \text{si } n > c \end{cases}$$

Donde `a` es el número de subproblemas que se consiguen en la parte de "dividir", `b` determina el tamaño de los subproblemas, `c` determina a partir de qué tamaño del problema se aplica el algoritmo de resolución del caso base. $D(n)$ y $C(n)$ determinan los tiempos de dividir y el problema y de combinar las distintas soluciones para un problema de tamaño n.

Análisis de Merge Sort

Aunque el algoritmo funciona perfectamente para cualquier número n, supondremos para simplificar el análisis del tiempo de ejecución que n es una potencia de 2. De esta forma, el tiempo de ejecución de *Merge Sort* puede calcularse por la recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Ya que en cada paso dividimos el problema en 2 subproblemas de tamaño $n/2$, el caso base es en $n = 1$, dividir el problema requiere un tiempo constante (se desprecia) y combinar las soluciones requiere un tiempo del orden de $\Theta(n)$.

Próximamente veremos que este tiempo de ejecución es $\Theta(n \log n)$. Aunque no sepamos calcularlo de forma analítica, sí que podemos intuir el por qué de esta

solución, ya que lo que hacemos es en cada nivel de recursión realizar un trabajo de tamaño n y realizar aproximadamente $\log n$ niveles de recursión, por lo que acabaremos con un tiempo del orden $n \log n$.

1.1.4. Búsqueda binaria

Suponiendo que tenemos un vector `v` ordenado, la búsqueda binaria devuelve el índice en dicho vector de la primera aparición de dicho elemento. Si el elemento no se encuentra en el vector `v`, devuelve la posición `-1`. El código es el siguiente:

```
// Devuelve el índice del valor k dentro del vector v[p,q], con p <= q
// Supone que v está ordenado de menor a mayor
// Si k no se encuentra en v, devuelve -1
int busqueda_binaria(int* v, int p, int q, int k){
    if(p >= q){ // Tamaño 1
        return (v[p] == k)? p : -1 ;
    }else{
        // Dividimos el vector en dos
        int r = (p+q)/2;

        if(v[r] == k)
            return r;
        else if(v[r] > k)
            return busqueda_binaria(v, p, r-1, k);
        else
            return busqueda_binaria(v, r+1, q, k);
    }
}
```

Este tiene un orden de eficiencia $\log n$ donde $n = q - p + 1$, ya que en el peor de los casos tenemos que dividir el vector todas las veces posibles, y un vector de tamaño n solo puede ir dividiéndose en mitades hasta alcanzar un tamaño 1 en $\log n$ pasos.

Ejercicio 1.1.1. Determina un algoritmo con orden de eficiencia $n \log n$ que dados un conjunto S de n enteros y otro entero x, determine si existen dos elementos en S cuya suma sea exactamente x.

Este ejercicio podemos resolverlo aplicando un proceso similar a la búsqueda binaria: Dado un vector `v`, primero lo ordenamos por *Merge Sort* en tiempo $n \log n$ y posteriormente aplicamos el siguiente proceso: por cada elemento del conjunto, buscamos por búsqueda binaria un segundo elemento del conjunto que sumado con este primer elemento sume x. De esta forma, este procedimiento tiene un coste $n \log n$, que sumado al coste del *Merge Sort* nos da un coste total $n \log n$.

```
// Busca un indice en el vector S[p,q] de forma que S[k] + S[indice] = x
int busca_sumar(int *S, int p, int q, int k, int x){
    if(p >= q){ // Tamaño 1
        return (S[p] + S[k] == x)? p : -1 ;
    }else{
        int r = (p+q)/2;

        if(S[r] + S[k] == x)
```

```
            return r;
        else if(S[r] + S[k] > x)
            return busca_sumar(S, p, r-1, k, x);
        else
            return busca_sumar(S, r+1, q, k, x);
    }
}

// Busca dos indices en el vector S[p,q] que sumen x
pair<int, int> busca_suma(int *S, int n, int x){
    // Ordenamos el vector
    merge_sort(S, 0, n-1);

    int i = 0;
    while(i < n){
        int k = busca_sumar(S, 0, n-1, i, x);
        if(k != -1)
            return pair<int, int>(S[i], S[k]);
        i++;
    }
    return pair<int, int>(-1, -1);
}
```

1.1.5. Bubble Sort

Bubble Sort es un algoritmo popular pero ineficiente de ordenación, que funciona intercambiando repetidamente las posiciones de los elementos que están fuera de orden:

```
// Ordena el vector v
void bubble_sort(int* v, int n){
    for(int i = 0; i < n; i++){
        for(int j = n-1; j > i; j--){
            if(v[j] < v[j-1]){
                // Intercambio
                swap(v[j], v[j-1]);
            }
        }
    }
}
```

1.1.6. Algoritmo de Horner

Dado un polinomio P de grado n:

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

Si tratamos de evaluzarlo en un punto b, tenemos que realizar $\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ multiplicaciones y $n-1$ sumas. En definitiva, un algoritmo del orden de n^2 operaciones,

siendo n el grado del polinomio P. Sin embargo, podemos ver la evaluación del polinomio como:

$$P(a) = \sum_{k=0}^{n} a_k b^k = a_0 + a_1 \cdot b + a_2 \cdot b^2 + \ldots + a_n \cdot b^n$$

Si sacamos b factor común, reducimos un cierto número de operaciones, y podemos sacar b como factor común un cierto número de veces:

$$\begin{aligned} P(a) = \sum_{k=0}^{n} a_k b^k &= a_0 + a_1 \cdot b + a_2 \cdot b^2 + \ldots + a_n \cdot b^n \\ &= a_0 + b(a_1 + a_2 \cdot b + \ldots + a_n \cdot b^{n-1}) \\ &= a_0 + b(a_1 + b(a_2 + \ldots + a_n \cdot b^{n-2})) \\ &\vdots \\ &= a_0 + b(a_1 + b(a_2 + \ldots + b(a_{n-1} + ba_n))) \end{aligned}$$

En definitiva, este procedimiento nos permite realizar la misma cuenta pero mediante n sumas y n productos, con lo que la eficiencia final del algoritmo es del orden de n. El código que reproduce este comportamiento es el llamado Algoritmo de Horner, que viene dado por:

```
// Evalúa el polinomio de grado n y coeficientes a[0], a[1], ..., a[n] en b
int evaluar_horner(int* a, int n, int b){
   int y = 0;
   for(int i = n; i >= 0; i--){
      y = a[i] + b * y;
   }
   return y;
}
```