Algoritmos y estructuras de datos

www.u-tad.com

1. Introducción a la construcción de software
2. Algoritmos no recursivos
3. Algoritmos recursivos
4. Listas, pilas y colas
5. **Ordenación** ←
6. Tablas hash
7. Árboles
8. Grafos

Introducción

- En este tema vamos a ver:
  - Algunos algoritmos sencillos de ordenación: inserción, selección y burbuja
    - Todos ellos tienen un tiempo de $O(n^2)$ para listas contiguas
  - También veremos algunos algoritmos avanzados de ordenación
    - Consiguen hacerlo (para listas contiguas) en un tiempo de $O(n \cdot \log n)$
    - Los más famosos son “MergeSort” y “QuickSort”
  - Finalmente veremos algoritmos especiales de ordenación que consiguen hacerlo en $O(n)$

Ordenación en $O(n^2)$ I

- Existen varios algoritmos de ordenación “sencillos” cuyas características son:
  - Son naturalmente iterativos
  - Son dos bucles anidados
  - Consiguen ordenar en un tiempo de $O(n^2)$ una lista contigua
    - Aunque las constantes ocultas son diferentes en cada uno de ellos
  - En el caso de una lista no contigua:
    - Sin pensar mucho, el algoritmo tardaría $O(n^3)$ pues acceder a un elemento ya nos cuesta $O(n)$
    - Podríamos conseguir $O(n^2)$ si programamos con cuidado el algoritmo, recordando los últimos accesos a determinados nodos críticos
      - Un concepto similar al de la actividad 4.4
  - Su complejidad espacial es $O(1)$ en todos los tipos de listas

Ordenación en $O(n^2)$ II

- Algoritmo de ordenación por inserción
  - Para cada elemento, buscamos el lugar en donde debería ir (posiciones anteriores a él) y lo ponemos

6 5 3 1 8 7 2 4

Ordenación en $O(n^2)$ III

- Algoritmo de ordenación por selección
  - Para cada «posicion» (desde 0 hasta n-1), buscamos el mínimo elemento entre «posicion» y el final y lo intercambiamos con «posicion»

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Ordenación en $O(n^2)$ IV

- Algoritmo de ordenación por burbuja
  - En cada iteración, se va comparando cada elemento con su siguiente, y si es mayor que dicho siguiente, ambos elementos se intercambian
  - Cada iteración simula una “burbuja” (el máximo) que asciende hasta el final del subvector de esa iteración, al igual que una burbuja asciende hasta la superficie del agua
  - Las burbujas pueden ir de izquierda a derecha (como en este ejemplo) o de derecha a izquierda (en cuyo caso la burbuja sería el mínimo y no el máximo)

6 5 3 1 8 7 2 4

MergeSort I

- La ordenación es un problema en el que los algoritmos recursivos surgen de manera natural
- Tanto en MergeSort como en QuickSort, se tiene el siguiente pseudocódigo:
  - La diferencia entre ambos es cómo se divide la lista y cómo se combinan las sublistas

```
1  algoritmo ordenar
2  entrada lista l
3  salida lista l
4  si longitud( l ) > 1 entonces
5      dividir l en dos sublistas l1 y l2
6      ordenar l1
7      ordenar l2
8      combinar l1 y l2 en l
```

MergeSort II

1. Se divide la lista original en dos sublistas de tamaño parecido
2. Se ordena cada sublista por separado (recursividad)
3. Se mezclan/combinan las dos sublistas (cada una de ellas ya está ordenada). Mezclar o combinar es:
   - Comparar el primer elemento de la primera sublista con el primer elemento de la segunda sublista.
   - El menor de los dos lo ponemos en la lista resultado
   - … y lo quitamos de la sublista en donde lo hemos encontrado
   - Repetimos los tres pasos anteriores hasta que una de las sublistas quede vacía
   - Finalmente copiamos en el resultado lo que queda en la sublista no vacía

MergeSort III

- A la hora de mezclar dos arrays hay dos opciones:
  - Espacio adicional O(n) para guardar la lista resultado en una lista temporal en cada paso, tiempo O(n)
  - Espacio adicional O(1), tiempo O(n)
    - Intentamos ahorrarnos el crear una lista resultado temporal
    - Algoritmo bastante complicado
- Si estamos aplicando el MergeSort sobre una lista enlazada (**actividad 5.1**), su mezcla tiene una complejidad espacial de O(1) y la temporal es O(n)
  - La complejidad espacial es O(1) porque cada vez que ponemos un nodo más en la lista resultado, lo hemos quitado de una de las sublistas de origen
    - Por lo tanto no necesitamos espacio adicional
  - Por lo tanto MergeSort funciona bien para listas enlazadas

MergeSort IV

- Análisis temporal de MergeSort
  - Partir la lista es simplemente calcular su punto medio: O(1) en contiguas y O(n) en enlazadas
  - Tenemos que ordenar cada parte de forma recursiva, por lo tanto tardamos 2T(n/2)
  - Hemos visto que la mezcla es O(n)
  - Por lo tanto, tenemos que MergeSort tiene una función de tiempo:
    - T(n) = 1 + 2T(n/2) + n +1 en contiguas
    - T(n) = n + 2T(n/2) + n +1 en enlazadas
  - Operando, nos queda O(n·logn) en ambas

MergeSort V

- Análisis espacial de MergeSort:
  - Partir la lista es O(1) en cualquier tipo de lista:
    - En contiguas sólo hay que recordar dónde acaba el primer trozo y donde empieza el segundo
    - En enlazadas tenemos que mover los nodos de un lugar a otro
      - Para ello, cada vez que insertemos un elemento en una sublista, lo eliminamos de la lista original
  - Hacemos dos llamadas recursivas. Como cuando hacemos la segunda ya hemos terminado la primera, la máxima memoria que necesitamos es M(n/2)
  - Combinar:
    - En listas contiguas es O(1) ó O(n), respectivamente con mezcla eficiente o no en memoria adicional
    - En listas enlazadas es siempre O(1) porque movemos los nodos de un lugar a otro
      - De nuevo, cada vez que insertemos un elemento desde una sublista a la lista total, lo eliminamos de la sublista de origen
  - Por lo tanto:
    - En listas contiguas con mezcla eficiente: $M(n)=1+M(n/2)+n \in O(n)$
    - En listas contiguas sin mezcla eficiente: $M(n)=1+M(n/2) \in O(\lg n)$
    - En listas enlazadas $M(n)=1+M(n/2) \in O(\lg n)$

QuickSort I

- Algoritmo propuesto por Hoare en 1962
- El algoritmo general es igual que en MergeSort. Recordemos que era:

```
1  algoritmo ordenar
2  entrada lista l
3  salida lista l
4  si longitud( l ) > 1 entonces
5      dividir l en dos sublistas l1 y l2
6      ordenar l1
7      ordenar l2
8      combinar l1 y l2 en l
```

QuickSort II

- La diferencia entre MergeSort y QuickSort es cómo se divide la lista y cómo se mezcla
- ¿Cómo se divide la lista en dos? (partición de la lista)
  - Se toma un elemento como pivote
    - Más adelante veremos cómo elegirlo. De momento, podemos pensar que el pivote es un elemento cualquiera (el primero, el del medio, etc.)
  - Se reparten el resto de elementos en dos listas, una con los menores que el pivote y otra con los mayores que el pivote
    - Los iguales a cualquiera de las dos… luego discutiremos sobre esto
- Por lo tanto, mezclar (una vez ordenada cada sublista) es sólo poner una sublista a continuación de otra
  - Pues todos los elementos de la primera son menores que todos los elementos de la segunda

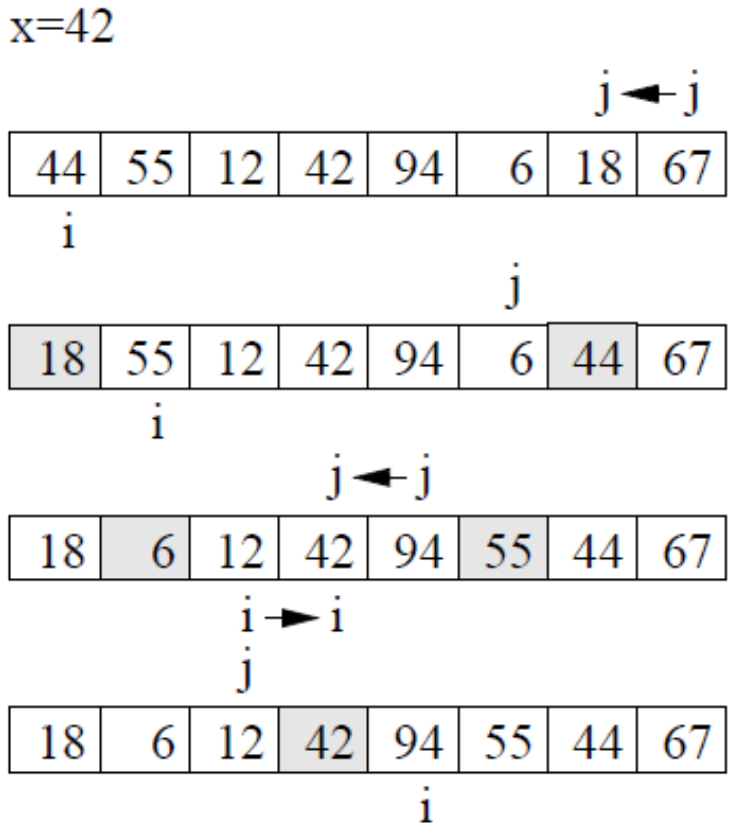
QuickSort III

- En el caso de listas contiguas:

```
1  algoritmo quick_sort
2  entrada tipo_elemento v[ inf ...sup ]
3          entero izq , der
4  salida  tipo_elemento v[ inf ...sup ]
5  local   entero ulti_izq , prim_der
6  si izq < der entonces
7     partición ( v, izq , der, ulti_izq , prim_der )
8     quick_sort ( v, izq , ulti_izq )
9     quick_sort ( v, prim_der, der )
```

QuickSort IV

- ¿Cómo hacemos la partición en listas contiguas?
  - Vamos recorriendo el vector de atrás a adelante (j) y de adelante a atrás (i)
    - Incrementamos i hasta apuntar a un elemento mayor que el pivote
    - Decrementamos j hasta apuntar a un elemento menor que el pivote
  - Intercambiamos los elementos apuntados por i y por j
  - Volvemos a incrementar/decrementar.
  - Así sucesivamente hasta que ambos índices se crucen (i>j)
- Ej: elegimos 42 como pivote por ser uno de los elementos centrales



QuickSort V

```
1  algoritmo partición
2  entrada tipo_elemento v[ inf ... sup ]
3          entero izq, der
4  salida  tipo_elemento v[ inf ... sup ]
5          entero ulti_izq , prim_der
6  local   entero i, j, k
7          tipo_elemento x
8  x ← v [ elegir_pivote ( v, izq, der ) ]
9  i ← izq
10 j ← der
11 repetir
12   mientras v[ i ] < x hacer
13     i ← i + 1
14   mientras v[ j ] > x hacer
15     j ← j − 1
16   si i ≤ j entonces
17     intercambiar( v, i, j )
18     i ← i + 1
19     j ← j − 1
20 hasta i > j
21  ulti_izq ← j
22 prim_der ← i
```

QuickSort VI

- ¿En qué sublista meter los elementos que sean iguales que el pivote? (**actividad 5.2**)
  - En general, podemos elegir meterlos en una o en otra indistintamente.
  - Nos podría pasar entonces que una de las sublistas se quedase con 0 elementos
    - Ej: elegimos meter el pivote en la misma sublista en la que metemos también los mayores que él. Tenemos la mala suerte de que el pivote es el mínimo elemento que hay en la lista. Por lo tanto la sublista que tiene los elementos estrictamente menores que el pivote quedaría vacía
  - Si una de las sublistas se queda con 0 elementos, entramos en una recursión infinita
    - Si una de las sublistas se queda con 0 elementos, la otra se queda con los n elementos… es decir, no hemos avanzado nada hacia la solución (estamos igual que al principio).
    - Cuando hagamos la llamada recursiva para la sublista que se ha quedado con los n elementos, volveremos a hacer exactamente lo mismo que antes: elegir el mismo pivote y una de sus sublistas volverá a quedar con 0 elementos, y así sucesivamente de forma infinita
    - Recordemos del tema 3 que una de las condiciones para que un programa recursivo funcione es que cada llamada recursiva se haga con un problema que sea más sencillo que el original. En este caso la llamada recursiva se hace con un problema igual que el original

QuickSort VII

- ¿Qué pivote elegimos?
  - A la hora de dividir en dos sublistas, lo peor que puede pasar es que una de ellas se quede con casi todos los elementos, y la otra con pocos
    - Luego vemos por qué
  - Si tomamos como pivote el primer (o último) elemento, y si la lista estaba ya ordenada, entonces nos encontramos con que una de las sublistas se queda con sólo 1 elemento (el pivote)
    - Es el peor caso
  - Por lo tanto, podemos:
    - Coger un pivote aleatorio
    - Coger el del medio
    - Coger la media de todos los elementos (cálculo en O(n))
    - Coger la mediana de unos cuantos (típicamente el primero, el último y el del medio)

QuickSort VIII

- Volvamos al problema de que una de las sublistas se quede con 0 elementos. ¿Cómo lo evitamos? Metiendo los elementos que sean iguales que el pivote en la sublista que quede con menos elementos al final del reparto
  - Problema: ¿qué ocurre si el primer elemento de la lista (el primero que examinamos para ver en qué sublista lo metemos) es igual que el pivote? Aún no sabemos qué sublista quedará con menos elementos al final del reparto.
    - Solución: si este fuera el caso, podríamos procesar los elementos de la lista desde el final hasta el principio… de este modo el primer elemento sería el último que procesáramos
  - Problema: ¿qué ocurre si aparece un elemento igual que el pivote en el medio de la lista? Tampoco sabemos aún qué sublista quedará con menos elementos al final del reparto
    - Solución: incrementamos un contador que nos diga el número de apariciones de elementos que sean igual que el pivote. Cuando acabemos de procesar toda la lista, miraremos ese contador y meteremos, en la sublista que haya quedado más pequeña, tantos elementos iguales que el pivote como valor haya en dicho contador
  - Problema: ¿y si calculamos el pivote como la media entre el primer y el último elemento, y ambos elementos son distintos?
    - Solución: no existe ningún problema, ya que, al ser el pivote la media, uno de los elementos es necesariamente menor que el pivote y el otro es necesariamente mayor. Por lo tanto ambos irán a sublistas distintas y por lo tanto ninguna sublista quedará con 0 elementos.
  - Problema: ¿y si calculamos el pivote como la media entre el primero y el último elemento, y ambos elementos da la casualidad de que son iguales?
    - Solución: tampoco hay ningún problema. El primer elemento lo meteremos en cualquiera de las dos sublistas (pues ambas están vacías, ya que estamos al principio del reparto), y el último elemento, cuando lleguemos a él, lo meteremos en la lista que haya quedado con menos elementos

QuickSort IX

- Si estamos en lista enlazada simple:
  - Tomaremos como pivote el primer elemento (tiempo O(1)), e iremos recorriendo la lista (O(n)), moviendo los menores a una nueva sublista y los mayores a otra
    - Si eligiéramos otra técnica para escoger el pivote (ej: el del medio), tendríamos que recorrer la lista para averiguar el valor del elemento del medio, lo cual sería O(n)

QuickSort X

- Análisis temporal:
  - Elegir el pivote: O(1) o O(n) según la técnica
    - Si sospechamos que el vector está ya ordenado o casi ordenado, debemos elegir más cuidadosamente el pivote aunque nos cueste O(n), por lo ya comentado
  - Partición: O(n)
  - Mezcla: O(1), sólo hay que poner una sublista a continuación de otra

QuickSort XI

- Análisis temporal por casos:
  - En el mejor caso: el pivote elegido siempre reparte equitativamente los elementos en dos sublistas de igual tamaño:
    - $T(n) = 2T(n/2) + n + 1$ si elegir pivote es $O(1)$
    - $T(n) = 2T(n/2) + 2n + 1$ si elegir pivote es $O(n)$
    - El resultado en ambos es $O(n\log n)$
  - En el peor caso: el pivote elegido siempre reparte mal los elementos: una sublista se queda con un elemento (y por lo tanto no hace falta ordenarla), y la otra con n-1
    - $T(n) = T(n-1) + n + 1$ si elegir pivote es $O(1)$
    - $T(n) = T(n-1) + 2n + 1$ si elegir pivote es $O(n)$
    - El resultado en ambos es $O(n^2)$
  - En el caso medio (no lo calculamos), QuickSort es $O(n\log n)$

QuickSort XII

- Análisis espacial por casos:
  - Elegir el pivote es siempre O(1)
  - Partición es O(1) siempre:
    - En listas contiguas, con el algoritmo que vimos
    - En listas enlazadas, porque movemos los nodos a una sublista o a otra
      - Cada vez que insertemos un elemento en una sublista, lo eliminamos de la lista origen
  - Las llamadas recursivas:
    - Usan M(n/2) en el mejor caso (las dos sublistas tienen el mismo tamaño), igual que en MergeSort
    - Usan M(n-1) en el peor caso (una de las sublistas tiene n-1 elementos y la otra 1)
  - La combinación es O(1)
  - Por lo tanto tenemos:
    - Mejor caso: $M(n)=1+M(n/2) \in O(\lg n)$, como en MergeSort
    - Peor caso: $M(n)=1+M(n-1) \in O(n)$
  - En el caso medio (no lo calculamos), es O(lgn)

Comparación MSort y QSort

- Tiempo en el peor caso (**actividad 5.3**):
  - MergeSort es O(nlogn)
  - QuickSort es $O(n^2)$
- Tiempo en el mejor caso:
  - MergeSort es O(nlogn)
  - QuickSort es O(nlogn), con constantes ocultas mejores que las del MergeSort (no lo calculamos) si la elección del pivote es O(1) y si n es muy grande
  - Cuando n es pequeño, QuickSort es peor que MergeSort e incluso peor que los algoritmos básicos de ordenación (inserción, selección, burbuja, etc.)
- Tiempo en el caso medio:
  - MergeSort es O(nlogn)
  - QuickSort es O(nlogn), con constantes ocultas mejores que las del MergeSort (no lo calculamos) si la elección del pivote es O(1) y si n es muy grande
  - Cuando n es pequeño, QuickSort es peor que MergeSort e incluso peor que los algoritmos básicos de ordenación (inserción, selección, burbuja, etc.)
- Por lo tanto, en cuestión de eficiencia temporal, QuickSort es el mejor algoritmo general de ordenación de todos los que existen, para el caso medio y cuando n es grande

Ordenación paralela

- Si usamos varios procesadores trabajando en paralelo podemos ordenar mucho más rápido, pues cada sublista puede ser ordenada en un procesador distinto
  - De esta manera, ambas llamadas recursivas se ejecutan a la vez
    - En MergeSort sobre listas contiguas tendríamos ahora T(n)=1+n+T(n/2) en vez de T(n)=1+n+2T(n/2)
  - El “paralelismo” es la disciplina que intenta dividir un algoritmo en partes que se puedan ejecutar en paralelo

Ordenación con rango I

- Si tenemos acotado el rango de los valores que toman los elementos que hay que ordenar, podemos desarrollar un algoritmo con complejidad temporal O(n) a costa de desperdiciar mucha memoria
  - Se trata de contar las veces que cada número aparece en la lista que queremos ordenar
    - Lo contamos en un array que tenga tantas posiciones como números posibles en el rango establecido
  - Una vez contadas las apariciones de cada número, podemos reconstruir la lista original, pero esta vez ya ordenada
  - Aunque desperdiciemos mucha memoria, la complejidad espacial sigue siendo O(1)

Ordenación con rango II

```
// Ordena n enteros que sabemos que pertenecen al rango [1,10000]
procedimiento ordenarPorRango (T[1..n])

    contador[1..10000] <- 0 // Es O(1)

    // Contamos las veces que aparece cada numero. O(n)
    para i <- 1 hasta n hacer
        contador[T[i]] ++

    // Ordenamos segun el contador anterior
    // Entramos en el bucle mientras n veces, por lo tanto O(n)
    j <- 0
    para i <- 1 hasta 10000
        mientras contador[i] > 0
            T[j] <- i
            contador[i] --
            j++
```

Ordenación con rango III

- Ej: ordenar los 10 elementos de la lista (1, 3, 5, 4, 1, 3, 3, 1, 5, 3), sabiendo que cada uno de sus elementos estarán en el rango [0,5]
  - Creamos el array “contadores”, con tamaño 6 (amplitud del rango)
    - Tiempo O(1), espacio O(n)
  - Inicializamos cada posición de “contadores” a 0. Nos queda contadores = (0, 0, 0, 0, 0, 0)
    - Tiempo O(1) porque el array siempre mide 5, independientemente de lo grande que sea n
    - Espacio O(1)
  - Contamos las veces que cada número aparece en la lista y lo anotamos en “contadores”. Para ello recorremos la lista, y cada vez que, por ejemplo, veamos un “3”, incrementaremos en una unidad la posición 3 de “contadores”. Por tanto, “contadores” queda con (0, 3, 0, 4, 1, 2)
    - Tiempo O(n), porque estamos recorriendo la lista y haciendo operaciones elementales por cada elemento
    - Espacio O(1)
  - Recorremos “contadores” y vamos poniendo en la lista original tantos números iguales como nos diga cada casilla de contadores. Por lo tanto, la lista queda con (1, 1, 1, 3, 3, 3, 3, 4, 5, 5)
    - Tiempo O(n), porque recorremos la lista para modificar cada elemento de la misma según lo que nos vaya diciendo “contadores”
    - Espacio O(1)

Ordenación con rango IV

- ¿Cuándo podemos usar este tipo de ordenación y por lo tanto conseguir la estupenda complejidad temporal de O(n)? Cuando se cumplan todos los requisitos siguientes:
  - Cada componente de la lista a ordenar está comprendido en un rango definido
  - Además, el rango tiene que ser discreto. Es decir, la cantidad de posibles números dentro del rango tiene que ser finita
    - Ej: no puede ser un rango de números reales
  - Cuando nos sobran grandes porciones de memoria contigua, para poder reservar el array “contadores”

Listas ordenadas I

- Podemos decidir mantener siempre ordenada una lista (**actividad 5.4**)
  - Buscar:
    - Como la lista ya está ordenada, la búsqueda es O(logn) si es contigua (con búsqueda binaria) o O(n) si es enlazada (con búsqueda secuencial)
      - Si la búsqueda es secuencial, en cuanto encontremos el elemento ya paramos, y así reducimos las constantes ocultas
  - Insertar:
    - Como la lista ya está ordenada, para insertar sólo hace falta buscar su lugar apropiado (con el método buscar visto) y meterlo ahí para que la lista permanezca ordenada
      - La búsqueda será binaria o secuencial según el tipo de lista
  - Eliminar:
    - Como la lista ya está ordenada, buscamos el elemento a eliminar y, al quitarlo, la lista permanecerá ordenada
      - La búsqueda también será binaria o secuencial según el tipo de lista

Listas ordenadas II

- ¿Para qué nos sirve mantener siempre ordenada una lista?
  - Es útil para cuando recorremos muchas veces la lista (para imprimirla, por ejemplo) y queremos que el recorrido sea en orden
  - Es útil para cuando necesitamos obtener el menor o el mayor muy rápidamente (están al principio y/o al final).
    - Ej: colas de prioridad
  - Es útil cuando necesitamos hacer muchas búsquedas y pocas inserciones/eliminaciones, pues la búsqueda sobre una lista contigua ordenada es más eficiente que en una desordenada
    - Si la lista es contigua, la búsqueda es O(logn) en una lista ordenada, pero O(n) si está desordenada
    - Si la lista es enlazada, la búsqueda es O(n) tanto si está ordenada como si no

Listas ordenadas III

- A cambio de tener la lista siempre ordenada, hacemos más trabajo para insertar y para eliminar
  - Insertar en una lista contigua desordenada:
    - Insertamos el elemento en la posición especificada como parámetro: O(n), pues recordemos que es posible que haya que desplazar a la derecha los elementos siguientes
    - Total: O(n)
  - Insertar en una lista contigua ordenada:
    - Buscamos (con búsqueda binaria) la posición en donde insertar el elemento: O(logn)
    - Insertamos en la posición encontrada: O(n)
    - Total: O(n), pero las constantes ocultas son mayores que si la lista está desordenada, pues aquí trabajamos más (tenemos que buscar la posición primero)
  - Insertar en una lista enlazada desordenada:
    - Buscamos el nodo de la posición especificada como parámetro: O(n), pues recordemos que hay que recorrer la lista
    - Insertamos el elemento entre ese nodo y el anterior: O(1), pues sólo hay que cambiar punteros
    - Total: O(n)
  - Insertar en una lista enlazada ordenada:
    - Buscamos (con búsqueda secuencial) la posición en donde insertar el elemento: O(n)
    - Metemos el elemento ahí: O(1), pues sólo hay que cambiar punteros
    - Total: O(n), con constantes ocultas iguales que si la lista enlazara estuviera desordenada
  - Asintóticamente, por lo tanto, tenemos O(n) tanto si la lista se mantiene ordenada como si no, pero observamos que hacemos más trabajo en las listas contiguas en el caso de querer mantenerla ordenada (las constantes ocultas son mayores)
  - Casos similares para eliminar, con similares soluciones