Tema 8. Algoritmos

January 8, 2021

1 Algoritmos

- Complejidad computacional
- Algoritmos de búsqueda y de ordenación
- Recursividad (funciones que se llaman a sí mismas)

2 Complejidad computacional

- Generalmente se asocia a un problema de computación y está referido a la complejidad del mejor algoritmo que lo resuelve
- Hay dos tipos de complejidad, la complejidad temporal y la complejidad espacial (memoria)
- Es una función de cómo varía el tiempo/memoria según varía el tamaño del problema a resolver.
    - Si tengo un algoritmo que resuelve un problema de tamaño x en un tiempo t, ¿qué tiempo necesitaré para un problema de tamaño 2*x?
    - No tiene por qué ser 2*t, podría ser mayor o menor (en los problemas interesantes puede ser de hecho mucho mayor)
- Generalmente hay un compromiso entre tiempo y memoria: si un algoritmo tiene buena complejidad temporal en general tendrá peor complejidad espacial y viceversa
- Hay clases de complejidad: conjuntos de problemas con una complejidad similar
- Las clases más conoicidas son la clase P y la clase NP, ¿son lo mismo?
- Clase P son los problemas polinomiales: el tiempo crece polinomialmente con el tamaño del problema. Ejemplos de este tipo de problemas son t = tamaño^2, t = 3 * tamaño^4 + 2 * tamaño^3. Se consideran resolubles independientemente del grado del polinomio.
- Clase NP son problemas para los que no conocemos solución polinomial pero si conocemos una solución, comprobar que realmente es una solución sí es polinomial. Ej. Factorizar números
- Suponemos que P y NP son clases distintas, pero no se ha podido demostrar
- Para expresar la complejidad de un problema usamos la notación de la O-grande (*big-O notation*)
    - O(función matemática). Se lee "orden de"
    - Los ejemplos anteriores serían $O(n^2)$ y $O(n^4)$
- Generalmente se habla de la complejidad en el peor caso, podría ser que en el caso más favorable un algoritmo resolviera un problema muy rápidamente

3 Algoritmos de búsqueda

- Formas de buscar elementos en una lista/tupla
- Dos formas: búsqueda lineal, búsqueda binaria
- Búsqueda lineal: voy elemento a elemento viendo si es el que busco. Su complejidad es lineal: $O(n)$
- Búsqueda binaria: solo se puede utilizar en listas ordenadas.
    - Miro en la mitad de la lista y si no es el elemento que busco, como la lista está ordenada puedo saber si tengo que mirar en la mitad superior o en la inferior.
    - Tomo esa nueva lista y hago lo mismo hasta que lo encuentre o descubra que no está
    - $O(ln\ n)$

4 Algoritmos de ordenación

- Dos grandes familias: ordenación interna y ordenación externa
- Externa: usa una lista auxiliar
- Interna: ordeno sobre la misma lista, intercambiando sus elementos
- Dos familias de algoritmos de ordenación interna:
    - Algoritmos simples: con complejidad $O(n^2)$ (burbuja, inserción directa y selección directa)
    - Algoritmos avanzados: con complejidad $O(n * ln\ n)$(búsqueda rápida (quicksort), búsqueda de montículo (heapsort), shellsort, mergesort

4.1 Burbuja

- Es el más sencillo de implementar
- Comparo el primer elemento con el segundo, si están en orden no hago nada y si no lo están, los intercambio. A continuación hago lo mismo con el segundo y el tercero. Y así sucesivamente. Cuando he llegado al último vuelvo a empezar.

[6,3,5,4,1,2]

Primera iteración (primera ronda de comparaciones)

3,6,5,4,1,2

3,5,6,4,1,2

3,5,4,6,1,2

3,5,4,1,6,2

3,5,4,1,2,6

Segunda iteración

3,5,4,1,2,6

3,4,5,1,2,6

3,4,1,5,2,6

3,4,1,2,5,6

Tercera

3,4,1,2,5,6

3,1,4,2,5,6

3,1,2,4,5,6

Cuarta

1,3,2,4,5,6

1,2,3,4,5,6

Quinta

1,2,3,4,5,6

El número de comparaciones que debo hacer en cada iteración es $n+n-1+n-2...+n$ equivalente a n^2 $O(n^2)$

4.2 Selección directa

- Elegimos directamente (buscamos) el elemento más pequeño de la lista lo colocamos en la primera posición intercambiándolo con el primero. A continuación buscamos el segundo más pequeño y lo colocamos en su sitio intercambiándolo con el segundo.

[6,3,5,4,1,2]

1,3,5,4,6,2

1,2,5,4,6,3

1,2,3,4,6,5

1,2,3,4,6,5

1,2,3,4,5,6

Su complejidad es también $O(n^2)$

4.3 Inserción directa

- Parto de una lista con 1 solo elemento y luego coloco el segundo en su sitio, comparándolo con el primero. Ahora tengo una lista de 2 elementos ordenados. Hago lo mismo con el tercero.

[6,3,5,4,1,2]

A la izquierda se muestran los que ya están ordenados

3,6 5,4,1,2

3,5,6 4,1,2

3,5,4,6

3,4,5,6 1,2

3,4,5,1,6

3,4,1,5,6

3,1,4,5,6

1,3,4,5,6 2

1,3,4,5,2,6

1,3,4,2,5,6

1,3,2,4,5,6

1,2,3,4,5,6

Su complejidad es también $O(n^2)$

Aunque los tres tienen la misma complejidad, inserción directa y selección directa son mucho más rápidos que la burbuja

5 Recursividad

- Funciones que se llaman a sí mismas

[ ]:
```python
def factorial(numero: int) -> int:
    """ Función que calcula el factorial usando bucles"""
    if numero < 0:
        return None
    elif numero == 0 or numero == 1:
        return 1
    else:
        resultado = 1
        for i in range(numero, 1, -1)
        resultado = resultado * i
    return resultado

def factorial_recursivo(numero: int) -> int:
    if numero == 1 or numero == 0:
        return 1
    else:
        return numero * factorial_recursivo(numero - 1)

# Si lo llamo con factorial_recursivo(5):

# 5 * factorial_recursivo(4)
# factorial_recursivo(4) = 4 * factorial_recursivo(3)
# factorial_recursivo(3) = 3 * factorial_recursivo(2)
# factorial_recursivo(2) = 2 * factorial_recursivo(1)
```

- La recursividad se basa en descomponer un problema en una versión más sencilla de sí mismo y volver a llamar a la función con esa nueva versión

- Tiene que existir uno o varios *casos base*: cuando llego al caso base (el factorial de 1 o de 0 en el ejemplo anterior) puedo parar la recursión y dar un resultado. Con ese resultado voy hacia atrás calculando el resto
- Las sucesivas llamadas a la función deben irme acercando al caso base