TECNICATURA EN

PROGRAMACIÓN

184 – ALGORTIMOS Y ESTRUCTURAS DE DATOS

JTP: ANGEL LEONARDO BIANCO

ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN

Los algoritmos de búsqueda son aquellos que permiten buscar uno o varios elementos dentro de una lista, matriz o arreglo. Dichos algoritmos se utilizan en una amplia variedad de aplicaciones informáticas, como la búsqueda de valores en una base de datos, la búsqueda de archivos en un sistema de archivos o la búsqueda de palabras en un procesador de texto.

Existen varios tipos de algoritmos de búsqueda, pero los más conocidos y utilizados son la búsqueda lineal y la búsqueda binaria.

BÚSQUEDA LINEAL

La búsqueda lineal, también conocida como búsqueda secuencial, es uno de los algoritmos de búsqueda más simples y populares. En este tipo de búsqueda, se recorre la lista de elementos uno por uno hasta encontrar el valor buscado. Si el valor no se encuentra en la lista, la búsqueda finaliza.

Por ejemplo, si tenemos una lista de números [2, 5, 8, 12, 19, 27, 35] y queremos buscar el número 19, el algoritmo de búsqueda lineal recorrería la lista de la siguiente manera:

1. Comenzamos por el primer elemento de la lista, que es 2.
2. Como 2 no es igual a 19, avanzamos al siguiente elemento, que es 5.
3. Continuamos por la lista hasta llegar a 19, que es el valor buscado.

Si el valor no se encuentra en la lista, el algoritmo simplemente devuelve un valor nulo o una indicación de que el valor no existe en la lista.

El algoritmo de búsqueda lineal se puede implementar en cualquier lenguaje de programación, como Python, utilizando un bucle for o while para recorrer la lista y una declaración if para comparar el valor buscado con cada elemento de la lista. A continuación se muestra un ejemplo de código Python para implementar la búsqueda lineal:

```
```
def busqueda_lineal(lista, valor):
      for i in range(len(lista)):
            if lista[i] == valor:
                  return i
      return -1
```
```

En este ejemplo, la función "busqueda_lineal" toma una lista y un valor como argumentos y recorre la lista utilizando un bucle for. Dentro del bucle, se compara cada elemento de la lista con el valor buscado utilizando una declaración if. Si el valor se encuentra en la lista, se devuelve su índice. Si no se encuentra el valor, devuelve -1.

TIEMPO Y COMPLEJIDAD

El algoritmo de búsqueda lineal en Python recorre una lista de elementos uno por uno hasta encontrar el valor buscado o llegar al final de la lista. La complejidad temporal de este algoritmo es O(n), donde n es el número de elementos en la lista.

Para calcular el tiempo de ejecución del algoritmo, es necesario conocer el tamaño de la lista y el tiempo que tarda en comparar cada elemento con el valor buscado. Si la lista tiene n elementos y el tiempo de comparación es constante, entonces el tiempo de ejecución será proporcional a n.En cuanto a la complejidad espacial, el algoritmo utiliza una cantidad constante de memoria para almacenar la lista y el valor buscado, por lo que su complejidad espacial es O(1).En resumen, la complejidad temporal del algoritmo de búsqueda lineal en Python es O(n), mientras que su complejidad espacial es O(1).

BÚSQUEDA BINARIA

La búsqueda binaria es un algoritmo de búsqueda más eficiente que la búsqueda lineal para listas ordenadas. En este algoritmo, se divide la lista en dos partes y se compara el valor buscado con el valor del elemento central de la lista. Si el valor buscado es menor que el valor del elemento central, la búsqueda continúa en la mitad izquierda de la lista. Si el valor buscado es mayor que el valor del elemento central, la búsqueda continúa en la mitad derecha de la lista. Este proceso se repite hasta que se encuentra el valor buscado o se determina que no existe en la lista.

Por ejemplo, si tenemos la misma lista [2, 5, 8, 12, 19, 27, 35] y queremos buscar el número 19, el algoritmo de búsqueda binaria adivinaría la mitad de la lista (12), lo compararía con 19, y determinaría que 19 estaría en la mitad derecha de la lista. A continuación, dividiría la mitad derecha de la lista en dos y repetiría el proceso hasta encontrar 19.

El algoritmo de búsqueda binaria se puede implementar en Python de la siguiente manera:

```
```
def busqueda_binaria(lista, valor):
      izq = 0
      der = len(lista) - 1
      while izq <= der:
```
```

```
            medio = (izq + der) // 2
            if lista[medio] == valor:
                  return medio
            elif lista[medio] < valor:
                  izq = medio + 1
            else:
                  der = medio - 1
      return -1
```

En este ejemplo, la función "busqueda_binaria" toma una lista y un valor como argumentos y utiliza un bucle while para buscar el valor. La variable "izq" se inicializa al principio de la lista y la variable "der" se inicializa al final de la lista. En cada iteración del bucle, se encuentra la mitad de la lista utilizando la fórmula (izq + der) // 2. Si el valor buscado es igual al valor en el elemento medio de la lista, se devuelve el índice de ese elemento. Si el valor buscado es menor que el valor en el elemento medio, la búsqueda continúa en la mitad izquierda de la lista. Si el valor buscado es mayor que el valor en el elemento medio, la búsqueda continúa en la mitad derecha de la lista. Si el valor no se encuentra en la lista, se devuelve -1.

En conclusión, los algoritmos de búsqueda son herramientas útiles para buscar elementos dentro de una lista o matriz en un programa. La búsqueda lineal es útil para listas desordenadas, mientras que la búsqueda binaria es más rápida y eficiente para listas ordenadas.

TIEMPO Y COMPLEJIDAD

El algoritmo de búsqueda binaria en Python busca un valor en una lista ordenada dividiendo repetidamente la lista en mitades y descartando la mitad en la que no se encuentra el valor buscado. El algoritmo comienza definiendo dos variables, izq y der, que representan el índice del primer y último elemento de la lista, respectivamente. Luego, mientras el índice izquierdo sea menor o igual al índice derecho, se calcula el índice medio de la lista y se compara el valor en ese índice con el valor buscado. Si el valor es igual al valor buscado, se devuelve el índice medio. Si el valor es menor que el valor buscado, se actualiza el índice izquierdo para que sea el índice medio más uno. Si el valor es mayor que el valor buscado, se actualiza el índice derecho para que sea el índice medio menos uno. Si el valor no se encuentra en la lista, se devuelve -1.La complejidad temporal de la búsqueda binaria en el peor de los casos es O(log n), donde n es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en mitades, reduciendo a la mitad el número de elementos que se deben buscar en cada iteración. La complejidad espacial es O(1), ya que el algoritmo utiliza una cantidad constante de memoria para almacenar las variables izq, der y medio.En resumen, la complejidad temporal de la búsqueda binaria en Python es O(log n), mientras que su complejidad espacial es O(1).

ALGORITMOS DE ORDENACIÓN

Los algoritmos de ordenación son técnicas utilizadas para ordenar un conjunto de elementos de acuerdo a un criterio específico. Existen diversos algoritmos de ordenación, cada uno con su propia complejidad, velocidad y eficiencia. A continuación, se explicarán algunos de los algoritmos más comunes de ordenación, así como su funcionamiento y ejemplos en el lenguaje de programación Python.

ALGORITMO DE ORDENACIÓN POR SELECCIÓN

El algoritmo de ordenación por selección es uno de los más simples pero menos eficientes. La idea principal es elegir el elemento más pequeño del conjunto y colocarlo en primer lugar. Luego, se busca el segundo elemento más pequeño y se coloca en segundo lugar, y así sucesivamente. Este proceso se repite hasta que todos los elementos estén en su posición correcta. A este método se le llama "ordenación por selección" porque en cada paso se selecciona el elemento más pequeño del conjunto.

Aquí se presenta un ejemplo de cómo se implementaría el algoritmo de selección en Python:

```
```
def selection_sort(nums):
    n = len(nums)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if nums[j] < nums[min_index]:
                min_index = j
        nums[i], nums[min_index] = nums[min_index], nums[i]
    return nums
```
```

TIEMPO Y COMPLEJIDAD

El algoritmo de ordenamiento por selección en Python recorre una lista de elementos y en cada iteración encuentra el elemento más pequeño y lo coloca en la posición correcta. La complejidad temporal de este algoritmo es O(n^2), ya que en cada iteración se recorre la lista para encontrar el elemento más pequeño y luego se realiza un intercambio. En el peor de los casos, se deben realizar n-1 iteraciones, y en cada iteración se deben realizar n-i comparaciones, lo que da un total de (n-1)*(n/2) comparaciones. La complejidad espacial es O(1), ya que el algoritmo utiliza una cantidad constante de memoria para almacenar las variables min_index, i, j y los

elementos de la lista que se intercambian.En resumen, la complejidad temporal del algoritmo de ordenamiento por selección en Python es O(n^2), mientras que su complejidad espacial es O(1).

ALGORITMO DE ORDENACIÓN POR INSERCIÓN

El algoritmo de ordenación por inserción es otro algoritmo simple pero eficiente que funciona comparando cada elemento con los elementos que lo preceden y colocándolo en su lugar correcto dentro del conjunto. Este proceso se realiza iterativamente para cada elemento del conjunto. En este algoritmo, se asume que la parte izquierda del conjunto siempre está ordenada y se insertan los elementos de la parte derecha en su posición correcta dentro de la parte ordenada.

A continuación, se presenta un ejemplo de implementación del algoritmo de ordenación por inserción en Python:

```
def insertion_sort(nums):
    n = len(nums)
    for i in range(1, n):
        j = i-1
        key = nums[i]
        while j >= 0 and nums[j] > key:
            nums[j+1] = nums[j]
            j -= 1
        nums[j+1] = key
    return nums
```

TIEMPO Y COMPLEJIDAD

El algoritmo de ordenamiento por inserción en Python recorre una lista de elementos y en cada iteración inserta el elemento actual en la posición correcta en la sublista ordenada que se ha generado hasta ese momento. La complejidad temporal de este algoritmo es O(n^2), ya que en el peor de los casos se deben realizar n-1 iteraciones, y en cada iteración se deben realizar hasta i comparaciones para encontrar la posición correcta del elemento actual. La complejidad espacial es O(1), ya que el algoritmo utiliza una cantidad constante de memoria para almacenar las variables i, j, key y los elementos de la lista que se intercambian.En resumen, la complejidad temporal del algoritmo de ordenamiento por inserción en Python es O(n^2), mientras que su complejidad espacial es O(1).

ALGORITMO DE ORDENACIÓN MERGE SORT

El algoritmo de ordenación Merge Sort es un método divide y conquista en el que el conjunto se divide en dos mitades y se ordenan de manera recursiva. Luego, se mezclan las dos mitades para obtener el conjunto ordenado final. Este algoritmo es mucho más eficiente que los anteriores algoritmos de ordenación y es muy útil para ordenar grandes conjuntos de datos.

Aquí se presenta un ejemplo de la implementación del algoritmo Merge Sort en Python:

```
def merge_sort(nums):
    if len(nums) > 1:
        mid = len(nums) // 2
        left_half = nums[:mid]
        right_half = nums[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                nums[k] = left_half[i]
                i += 1
            else:
                nums[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            nums[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            nums[k] = right_half[j]
            j += 1
            k += 1
```

TIEMPO Y COMPLEJIDAD

El algoritmo de ordenamiento por mezcla en Python utiliza la técnica divide y vencerás para ordenar una lista de elementos. En cada iteración, el algoritmo divide la lista en dos mitades y ordena cada mitad recursivamente hasta que se llega a listas de tamaño 1 o 0. Luego, combina las dos mitades ordenadas en una sola lista ordenada. La complejidad temporal de este algoritmo es O(n log n), ya que en cada iteración se divide la lista en dos mitades y se ordenan recursivamente, y luego se combinan las dos mitades ordenadas en una sola lista ordenada. La complejidad espacial es O(n), ya que el algoritmo utiliza memoria adicional para almacenar las dos mitades de la lista y la lista ordenada resultante.En resumen, la complejidad temporal del algoritmo de ordenamiento por mezcla en Python es O(n log n), mientras que su complejidad espacial es O(n).

ALGORITMO DE ORDENACIÓN QUICK SORT

El algoritmo de ordenación Quick Sort es otro algoritmo divide y conquista que funciona seleccionando un elemento pivot y dividiendo el conjunto en dos partes: una con elementos más pequeños que el pivot y otra con elementos más grandes. Luego, se ordenan de manera recursiva las partes izquierda y derecha del conjunto. Si la parte izquierda del conjunto tiene elementos más pequeños que la parte derecha, el pivot se sitúa en su posición correcta en el medio del conjunto.

A continuación, se presenta un ejemplo de implementación del algoritmo Quick Sort en Python:

```
```
def quick_sort(nums):
    if len(nums) <= 1:
        return nums
    else:
        pivot = nums[0]
        less = [x for x in nums[1:] if x <= pivot]
        greater = [x for x in nums[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)
```
```

TIEMPO Y COMPLEJIDAD

El algoritmo de ordenamiento QuickSort en Python utiliza la técnica de divide y vencerás para ordenar una lista de elementos. En cada iteración, el algoritmo elige un elemento de la lista como pivote y divide la lista en dos sub-listas: una con los elementos menores o iguales al pivote y otra con los elementos mayores al pivote. Luego, ordena recursivamente cada sub-lista y las combina en una sola lista ordenada. La complejidad temporal de este algoritmo es O(n log n) en el mejor y promedio de los casos, y O(n^2) en el peor caso, cuando la lista está ordenada en orden inverso. La complejidad espacial es O(log n), ya que el algoritmo utiliza memoria adicional para almacenar las llamadas recursivas a la función.En resumen, la complejidad temporal del algoritmo de ordenamiento QuickSort en Python es O(n log n) en el mejor y promedio de los casos, y O(n^2) en el peor caso, mientras que su complejidad espacial es O(log n).

ALGORITMO DE ORDENACIÓN BUBBLE SORT

El algoritmo de ordenación Bubble Sort es otro algoritmo simple que funciona comparando parejas de elementos adyacentes y cambiándolos si están en el orden

incorrecto. Este proceso se repite varias veces hasta que no se produzcan más cambios. Este algoritmo no es muy eficiente y no se recomienda su uso para grandes conjuntos de datos.

Aquí se presenta un ejemplo de la implementación del algoritmo Bubble Sort en Python:

```
```
def bubble_sort(nums):
    n = len(nums)
    for i in range(n):
        for j in range(0, n-i-1):
            if nums[j] > nums[j+1]:
                nums[j], nums[j+1] = nums[j+1], nums[j]
    return nums
```
```

En conclusión, existen diversos algoritmos de ordenación con diferentes complejidades, velocidades y eficiencias. La elección del algoritmo de ordenación apropiado dependerá del conjunto de datos que se desee ordenar y de la eficiencia requerida.

TIEMPO Y COMPLEJIDAD

El algoritmo de ordenamiento Bubble Sort en Python recorre la lista varias veces, comparando elementos adyacentes y realizando intercambios si es necesario. En cada iteración, el algoritmo compara los elementos adyacentes y los intercambia si están en el orden incorrecto. La complejidad temporal de este algoritmo es O(n^2), ya que en el peor caso, el algoritmo debe recorrer la lista n veces, y en cada iteración, debe comparar n-i-1 elementos. La complejidad espacial es O(1), ya que el algoritmo no utiliza memoria adicional para ordenar la lista.En resumen, la complejidad temporal del algoritmo de ordenamiento Bubble Sort en Python es O(n^2), mientras que su complejidad espacial es O(1).

ENLACES

Implementaciones de algoritmos de búsqueda en Python (geekflare.com)

6.6 Complejidad de algoritmos - Programación en Python (unsam.edu.ar)

https://programacionpython.ecyt.unsam.edu.ar/material/12_Ordenamiento/02_Ordenamiento_sencillo/

https://diegoamorin.com/ordenamiento-seleccion-python/

https://elblogpython.com/tutoriales-python/que-es-la-ordenacion-por-seleccion-en-python/

https://uniwebsidad.com/libros/algoritmos-python/capitulo-19/ordenamiento-por-insercion

https://www.geeksforgeeks.org/sorting-algorithms/