Tarea 6

Juárez Torres Carlos Alberto

April 18, 2023

1 Problema de Selección

El Problema de Selección consiste en encontrar el k-ésimo elemento más pequeño de un conjunto de n datos, $1 \leq k \leq n$. Considere los algoritmos de ordenamiento:

- Bubble Sort
- Insertion Sort
- Shell Sort
- Selection Sort
- Local Insertion Sort
- Heap Sort
- Merge Sort

Suponiendo que k es tal que $1 < k < n$. ¿Cuáles de las estrategias usadas por los algoritmos anteriores, nos ayudan a resolver el Problema de Selección sin tener que ordenar toda la secuencia? Justifica, para cada inciso, tus respuestas.

- **Selection Sort** Este algoritmo selecciona el elemento más pequeño en cada iteración y lo coloca en la posición correspondiente en el arreglo. Para resolver el Problema de Selección, podemos modificar este algoritmo para que en lugar de ordenar todo el arreglo, pare cuando encuentre el k-ésimo elemento más pequeño. Esto se logra mediante la implementación de una condición adicional que detenga el algoritmo cuando se encuentre el k-ésimo elemento.
- **Heap Sort** Este algoritmo utiliza una estructura de datos llamada heap para ordenar los elementos del arreglo. Para resolver el Problema de Selección, podemos construir un heap de tamaño k con los primeros k elementos del arreglo. Luego, recorremos los elementos restantes del arreglo y los vamos insertando en el heap si son más pequeños que el elemento más grande del heap. De esta manera, al final del recorrido, el elemento en la raíz del heap es el k-ésimo elemento más pequeño.
- **Merge Sort** Este algoritmo divide el arreglo en dos mitades, las ordena por separado y luego las combina en un arreglo ordenado. Para resolver el Problema de Selección, podemos modificar este algoritmo para que solo ordene la mitad que contiene el k-ésimo elemento. Si el k-ésimo elemento está en la primera mitad, se aplica el algoritmo de ordenamiento recursivamente a la primera mitad; si está en la segunda mitad, se aplica recursivamente a la segunda mitad. Si el k-ésimo elemento está en la intersección de ambas mitades, se devuelve ese elemento.

2 Indicar cuáles de los algoritmos de ordenamiento mencionados en el son estables y justificar por qué.

- **Insertion Sort** Este algoritmo es estable, ya que inserta cada elemento en la posición correcta en la sublista ordenada. Si hay varios elementos con la misma clave, se colocan en el orden en que aparecen en la lista original.
- **Merge Sort** ste algoritmo es estable, ya que en la fase de combinación siempre se toman los elementos de las sublistas en orden, es decir, primero se toma el elemento de la sublista izquierda y luego el de la sublista derecha. De esta forma, si hay varios elementos con la misma clave, el que aparece primero en la lista original será el primero en ser tomado y por lo tanto aparecerá antes en la lista ordenada.

3 Realiza la siguiente modificación al algoritmo Insertion Sort: Para buscar la posición del nuevo elemento, el que está en revisión, usar Búsqueda Binaria, en vez de hacerlo secuencialmente.

```
def Busqueda(arr, val, start, end):

    if start == end:
        if arr[start] > val:
            return start
        else:
            return start + 1

    if start > end:
        return start

    mid = (start + end) // 2

    if arr[mid] < val:
        return Busqueda(arr, val, mid + 1, end)
    elif arr[mid] > val:
        return Busqueda(arr, val, start, mid - 1)
    else:
        return mid + 1

def insertion_sort_b(arr):

    for i in range(1, len(arr)):
        val = arr[i]
        pos = Busqueda(arr, val, 0, i - 1)
        arr = arr[:pos] + [val] + arr[pos:i] + arr[i+1:]
    return arr
```

La función insertion_sort_b toma como entrada un arreglo arr y utiliza el algoritmo de inserción con búsqueda binaria para ordenar el arreglo de manera ascendente.

La función Busqueda toma como entrada el arreglo arr, los índices left y right que delimitan el rango en el que se buscará la posición del elemento key , y el valor key que se está buscando. La función devuelve la posición donde se debería insertar el valor key en el arreglo arr para que este siga ordenado.

En la función insertion_sort_b, el bucle principal itera sobre los índices i del arreglo desde el segundo elemento hasta el último. En cada iteración, se almacena el valor del elemento en la posición i en la variable key y se busca su posición adecuada en el arreglo utilizando la función Busqueda. A continuación, se desplazan los elementos

1. **Complejidad del algoritmo** La operación más costosa en este algoritmo es la búsqueda binaria, que tiene un tiempo de ejecución de $O(logn)$, donde "n" es el número de elementos en el arreglo. Dentro del bucle principal, la búsqueda binaria se realiza para cada elemento del arreglo, por lo que el tiempo total de ejecución del algoritmo es de $O(nlogn)$.

En el peor de los casos, cuando el arreglo está completamente desordenado, la búsqueda binaria tendrá que buscar la posición adecuada del elemento en todo el arreglo, por lo que su tiempo de ejecución será $\log_2(n)$ iteraciones en la búsqueda binaria. Así, el tiempo total de ejecución del algoritmo sería $O(nlogn)$, ya que la búsqueda binaria se realiza para cada uno de los n elementos del arreglo.

En resumen, el desempeño computacional del algoritmo Insertion Sort modificado con búsqueda binaria es $O(nlogn)$, lo que lo hace más eficiente que la versión original del algoritmo de inserción, que tiene un desempeño de $O(n^2)$ en el peor de los casos.

2. **¿Mejora el desempeño computacional del proceso?** Si, La versión original del algoritmo de inserción tiene un tiempo de ejecución de $O(n^2)$, lo que significa que el tiempo de ejecución aumenta cuadráticamente con el número de elementos en el arreglo. Por lo tanto, cuando el tamaño del arreglo aumenta, el tiempo de ejecución se vuelve muy grande y el algoritmo se vuelve menos eficiente.

   Por otro lado, el algoritmo Insertion Sort modificado con búsqueda binaria tiene un tiempo de ejecución de $O(nlogn)$, lo que significa que el tiempo de ejecución aumenta linealmente con el tamaño del arreglo, pero no tan rápido como en la versión original del algoritmo de inserción. Por lo tanto, el algoritmo modificado es más eficiente en términos de tiempo de ejecución y se puede manejar fácilmente un mayor número de elementos en el arreglo.

4

5 Dados n puntos en el plan, encontrar un poligono que tenga como vértices los n puntos dados

1. **Diseño de un algoritmo** Se puede utilizar un enfoque de dividir y conquistar. Primero, se selecciona un punto extremo en el plano como el primer vértice del polígono, por ejemplo, el punto con la coordenada x más baja. A continuación, se ordenan los puntos restantes por su ángulo polar con respecto a este punto extremo. Luego, se construye el polígono agregando puntos en orden, siempre y cuando el ángulo formado por el nuevo punto y los dos puntos más recientes del polígono sea menor que 180 grados. Si no es menor que 180 grados, se elimina el punto más reciente del polígono y se vuelve a verificar el ángulo con el nuevo punto. Este proceso se repite hasta que se hayan agregado todos los puntos.

2. **Justificación de la propuesta** La estrategia de dividir y conquistar para construir el polígono garantiza que se pueda encontrar un polígono que tenga como vértices los n puntos dados. Al seleccionar un punto extremo como primer vértice, el algoritmo asegura que este punto estará en el polígono resultante y que no habrá puntos adicionales dentro del polígono. El ordenamiento de los puntos restantes por su ángulo polar con respecto al punto extremo también garantiza que los puntos se agreguen al polígono en orden, siguiendo una trayectoria continua alrededor del borde del polígono.

3. **Complejidad** El algoritmo tiene una complejidad de tiempo de $O(n \log n)$ para ordenar los puntos por ángulo polar utilizando, por ejemplo, un algoritmo de ordenamiento quicksort o mergesort. Luego, la construcción del polígono se realiza en tiempo lineal $O(n)$, ya que cada punto se agrega al polígono una vez. Por lo tanto, la complejidad total del algoritmo es $O(n \log n)$.

4. **Pseudo-código**

```
def poligono_convexo(puntos):

    extremo = min(puntos, key=lambda p: p[0])

    puntos_ordenados = sorted(puntos,
                              key=lambda p: math.atan2(p[1]-extremo[1], p[0]-extremo[0])

    poligono = [extremo, puntos_ordenados[0]]

    for i in range(1, len(puntos_ordenados)):
        while len(poligono) >= 2 and \
              (poligono[-1][0]-poligono[-2][0])*
                (puntos_ordenados[i][1]-poligono[-2][1]) - \
              (poligono[-1][1]-poligono[-2][1])*
                (puntos_ordenados[i][0]-poligono[-2][0]) < 0:

            poligono.pop()

        poligono.append(puntos_ordenados[i])
    return poligono
```

6 Suponga que tiene n intevalos cerrados sobre la recta real [a(i), b(i)], con $i \leq i \leq n$. Encontrar la máxima k tal que existe un punto x que es cubierto por los k intervalos.

- **Diseño del algoritmo** Podemos utilizar la estrategia de búsqueda binaria. Primero, ordenamos los intervalos de manera creciente por sus puntos finales $b(i)$. Luego, utilizamos una búsqueda binaria para encontrar el valor k máximo para el cual existe un punto x que es cubierto por los k intervalos. En cada iteración de la búsqueda binaria, consideramos el punto medio entre el valor actual de k y el valor máximo posible de k (que es n, ya que hay n intervalos). Luego, verificamos si este valor medio es válido, es decir, si hay un punto x que es cubierto por los k intervalos. Si el valor medio es válido, actualizamos el valor de k mínimo encontrado hasta ahora y continuamos buscando en la mitad inferior del rango de búsqueda. Si el valor medio no es válido, buscamos en la mitad superior del rango de búsqueda.

- **Justificación** Esta estrategia es correcta porque aprovecha el hecho de que los intervalos están ordenados por sus puntos finales $b(i)$, lo que nos permite realizar una búsqueda binaria para encontrar el valor máximo de k en lugar de recorrer todos los posibles subconjuntos de intervalos. Además, si un punto x está cubierto por k intervalos, entonces también estará cubierto por $k-1$ intervalos. Por lo tanto, el valor de k máximo es monótonamente decreciente con respecto al punto x. Esto significa que podemos utilizar una búsqueda binaria para encontrar el valor máximo de k de manera eficiente.

- **Complejidad** La complejidad computacional de este algoritmo es $O(n \log n)$, ya que primero se debe ordenar la lista de intervalos (lo que toma $O(n \log n)$ tiempo) y luego se realiza una búsqueda binaria (lo que toma $O(\log n)$ tiempo por iteración). En total, se realizan $O(\log n)$ iteraciones de la búsqueda binaria, por lo que la complejidad total del algoritmo es $O(nlogn)$.

- **Pseudo-código**

```
def max_intervalos_cubren_punto(intervalos):
  intervalos_ordenados = sorted(intervalos, key=lambda intervalo: intervalo[1])
  k_min = 0
  k_max = len(intervalos_ordenados)
  while k_min < k_max:
      k_medio = (k_min + k_max + 1) // 2
          k_min = k_medio
      else:
          k_max = k_medio - 1
  return k_min

def existe_punto_cubierto_por_k(intervalos, k):
  contador_puntos = {}
  for intervalo in intervalos[:k]:
      for i in range(intervalo[0], intervalo[1]+1):
          contador_puntos[i] = contador_puntos.get(i, 0) + 1
  for punto, conteo in contador_puntos.items():
      if conteo >= k:
          return True
```

```
return False
```