Analisis de Algoritmos 2024-1

Tarea 06: Ordenamientos

Profesor: Profesor(a): María de Luz Gasca Soto
Ayudantes: Rodrigo Fernando Velázquez Cruz
Teresa Becerril Torres
Alumno: Alvaro Ramirez Lopez **N° cuenta:** 316276355

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

   Considere los algoritmos de ordenamiento:

   (a) Merge Sort
   (b) Selection Sort
   (c) Quick Sort
   (d) Heap Sort
   (e) Insertion Sort
   (f) Shell Sort
   (g) Local Insertion Sort

   Pregunta: ¿Cuáles de las estrategias usadas por los algoritmos anteriores, nos ayudan a resolver el Problema de Selección, sin tener que ordenar toda la secuencia? Suponga k tal que $1 < k < n$.

   Justifique, para cada inciso, sus respuestas.

   Solución:

   1. **Merge Sort:** Este algoritmo puede modificarse para resolver el Problema de Selección. En lugar de hacer merge completo de las sublistas, podemos detener el proceso cuando queden k elementos sin mezclar en el conjunto grande, es decir que haremos recursion hasta el nivel k donde el tamaño de los subconjuntos sea 1 o 2, y luego haremos un merge pero solo con los elementos menores de esos subconjuntos, es decir que nos quedaria un subconjunto de n `div` 2 elementos, ahi las comparaciones serian a lo mas $log(n)$.
   2. **Selection Sort:** No ayuda a resolver el Problema de Selección sin ordenar toda la secuencia, ya que se basa en intercambiar el elemento mínimo sucesivamente hasta ponerlo en la posición correcta.
   3. **Quick Sort:** Al igual que Merge Sort, puede modificarse para resolver el Problema de Selección. Podemos detener el proceso cuando el pivote ocupe la posición k, retornando ese elemento. Es decir que solo haremos uso de un lado de la particion, asi descartamos los elementos mayores al pivote y seguimos haciendo recursion hasta que el tamaño del subconjunto sea 1 o 2.
   4. **Heap Sort:** Puede usarse el heap mínimo (min heap) para extraer los k elementos más pequeños en orden en O(k log n) tiempo. Esto resuelve el Problema de Selección sin ordenar toda la secuencia.
   5. **Insertion Sort:** No ayuda a resolver el Problema de Selección sin ordenar toda la secuencia, ya que ordena insertando elementos uno a uno en posiciones anteriores.
   6. **Shell Sort:** Tampoco ayuda sin ordenar toda la secuencia, ya que es una variante de Insertion Sort.
   7. **Local Insertion Sort:** Al igual que Insertion Sort, debe ordenar toda la secuencia, ademas que la referencia que se guarda sobre el ultimo elemento insertado no nos asegura que sea el elemento k-esimo mas pequeño.

2. Resolver el Problema de Selección usando el proceso Partition. Además indique el desempeño computacional de la estrategia.

Solución:

Aquí va la solución.

3. Desarrollar uno de los siguientes ejercicios:

   (A) Desarrollar un algoritmo que genere listas de datos que resulten ser el peor caso para el Shell Sort para las secuencias de Shell y para las secuencias de Hibbart.

   (B) Desarrolle un algoritmo que genere listas de datos que resulten ser el peor caso para el Quick Sort cuando el pivote es $S[n \texttt{ div } 2]$

Solución:

(A) **Desarrollo del algoritmo que genera el peor caso para el Shell Sort para las secuencias de Shell**

Para generar el peor caso para las secuencias de Shell debemos diseñar una lista de datos en orden inverso de modo que shell sort tenga que hacer muchas comparaciones e intercambios en cada paso. A continuación se muestra el algoritmo que genera el peor caso para el Shell Sort para las secuencias de Shell.

```python
import sys

def worst_case_shell_sort_shell(seq):
  n = len(seq)
  h = 1
  while h < n // 3:
    h = 3 * h + 1

  worst_case_list = list(range(n, 0, -1))
  return worst_case_list

# Ejemplo de uso
n = int(sys.argv[1])
worst_case_list = worst_case_shell_sort_shell(list(range(1, n + 1)))
print("Peor caso para Shell Sort con secuencias de Shell (n = {}):".format(n))
print(worst_case_list)
```

Desarrollo del algoritmo que genera el peor caso para Shell Sort para las secuencias de Hibbart

Para generar el peor caso para las secuencias de Hibbart debemos diseñar una lista de datos que sea la concatenación de varias secuencias ordenadas de manera inversa.

```python
import sys

def worst_case_shell_sort_hibbard(seq):
  n = len(seq)
  gaps = [2**k - 1 for k in range(int(math.log2(n)) + 1)]
  worst_case_list = []

for gap in gaps:
    sublist = list(range(n, 0, -1))[:gap]
    worst_case_list += sublist

return worst_case_list

# Ejemplo de uso
n = int(sys.argv[1])
worst_case_list = worst_case_shell_sort_hibbard(list(range(1, n + 1)))
print("Peor caso para Shell Sort con secuencias de Hibbard (n = {}):".format(n))
print(worst_case_list)
```

(B) **Desarrollo del algoritmo que genera el peor caso para el Quick Sort**

Para generar el peor caso para el algoritmo de Quick Sort cuando el pivote se elige como el elemento en la posición S[n div 2], debemos diseñar una lista de datos que esté ordenada de manera descendente o ascendente, el siguiente algoritmo crea una lista ordenada de manera descendente y luego intercambia el elemento en la posición n div 2 (la mitad de la lista) con el primer elemento. Esto garantiza que el pivote siempre divida la lista en dos sublistas desiguales, lo que resulta en el peor caso de rendimiento para Quick Sort con el pivote en S[n div 2].

```python
import sys

def worst_case_quick_sort(n):
  if n <= 0:
    return []

  # Crear una lista ordenada de manera descendente
  worst_case_list = list(range(n, 0, -1))

  # Intercambiar el elemento en la posición n div 2 con el primer elemento
  worst_case_list[0], worst_case_list[n // 2] = worst_case_list[n // 2], worst_case_list[0]

  return worst_case_list

# Ejemplo de uso
n = int(sys.argv[1])
worst_case_list = worst_case_quick_sort(n)
print("Peor caso para Quick Sort (n = {}):".format(n))
print(worst_case_list)

```

4. **Problema Opcional** Φ: Suponga que tiene n intervalos cerrados sobre la línea real: $[a(i), b(i)]$, con $1 \leq i \leq n$. Encontrar la máxima k tal que existe un punto x que es cubierto por los k intervalos.

   (a) Proporcione un algoritmo que solucione el problema Φ.

   (b) Justifique que su propuesta de algoritmo es correcta.

   (c) Calcule, con detalle, la complejidad computacional de su propuesta.

(d) Proporcione un pseudo-código del algoritmo propuesto.