UNIVERSIDAD CENTROAMERICANA

JOSÉ SIMEÓN CAÑAS

DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA

ANÁLISIS DE ALGORITMOS

TALLER 2

ANÁLISIS FORMAL DEL CÓDIGO

Estudiantes:

Argueta Flores, Alison Aracely - 00076422

Jacobo Cornejo, Diego Armando - 00043719

Octubre 2024

Antiguo Cuscatlán, San Salvador, CA

```cpp
#include <iostream> C1 * O(1)
#include <fstream> C2 * O(1)
#include <iomanip> C3 * O(1)

using namespace std; C4 * O(1)

// Algoritmo utilizado max-heap
void heapify(double arr[], int n, int i) { C5 * O(1)
    int mayor = i; C6 * O(1)
    int izq = 2 * i + 1;C7 * O(1)
    int der = 2 * i + 2; C8 * O(1)

    if (izq < n && arr[izq] > arr[mayor]) C9 * O(1)
        mayor = izq; C10 * O(1)

    if (der < n && arr[der] > arr[mayor]) C11 * O(1)
        mayor = der; C12 * O(1)

    if (mayor != i) { C13 * O(1)
        swap(arr[i], arr[mayor]); C14 * O(1)

        // Recursivamente hacer heapify en el subárbol afectado
        heapify(arr, n, mayor); C15 * O(lgn)
    }
}
```

En la línea 23 del código se produce recursividad debido a que la función heapify se vuelve a llamar dentro de ella misma.
En este caso podría tomar hasta 2/3 de los nodos en el árbol original si no está balanceado, y solo realiza un llamado:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Aplicando el teorema maestro se tienen los siguientes valores:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

donde $a = 1$, $b = \frac{3}{2}$ ^ $d = 0$

$\frac{n}{b} = \frac{2n}{3} \rightarrow n = \frac{2nb}{3}$

$3n = 2nb \rightarrow \frac{3n}{2n} = b$

$b = \frac{3}{2}$

Resolvemos:

$\log_b a = \log_{\frac{3}{2}} 1$ lo que es igual a cero.

Ya que $d = 0$ entonces $d = \log_a b$, según el caso 2 del teorema maestro el orden de la recursividad es:

$$O(n^d \lg n) = O(n^0 \lg n) = O(\lg n)_{//}$$

Ahora que todas las líneas poseen su respectivo orden estas se relacionan entre sí:

$C1 * O(1) + C2 * O(1) + C3 * O(1) + C4 * O(1)$

Al ser todas las líneas de comportamiento $O(1)$ entonces se concluye que el uso de librerías es constante: $O(1)$

Seguidamente se analiza el comportamiento de la función heapify relacionando cada línea:

$C5 * O(1) + C6 * O(1) + C7 * O(1) + C8 * O(1) + C9 * O(1)$
$+ C10 * O(1) + C11 * O(1) + C12 * O(1) + C13 * O(1)$
$+ C14 * O(1) + C15 * O(\lg n)$

las líneas de C5 a C14 todas poseen el mismo orden $O(1)$ por lo cual se identificarán con la variable "A"; por último se realiza la unión de Ambas: $A + (C15 * O(\lg n))$

$\therefore O(1) + O(\lg n) = O(\lg n)$ la función heapify posee comportamiento $O(\lg n)$

```
26
27  // Heap Sort en orden ascendente
28  void heapSort(double arr[], int n) {  C1 * O(1)
29      for (int i = n / 2 - 1; i >= 0; i--) C2 * O(n)
30          heapify(arr, n, i); C3 * O(lgn) * O(n)        } O(nlgn)
31
32      // Extraer uno por uno los elementos del heap
33      for (int i = n - 1; i > 0; i--) {  C4 * O(n)
34          swap(arr[0], arr[i]); C5 * O(1) * O(n)        } O(nlgn)
35          heapify(arr, i, 0); C6 * O(lgn) * O(n)
36      }
37
38      // Orden descendente
39      for (int i = 0; i < n / 2; i++) {  C7 * O(n)
40          swap(arr[i], arr[n - i - 1]); C8 * O(1) * O(n)   } O(n)
41      }
42  }
```

Para conocer los límites del primer for en la línea c2 se realiza lo siguiente:

$[0, n/2 - 1] = \frac{n}{2} - 1$ y al ser cabecera se ejecuta una vez más; definiendo su comportamiento como $n/2 - 1 + 1 = n/2$

Al obtener $\frac{n}{2}$ se puede representar como $\frac{1}{2}n$ lo que sería cn ∴ la línea 29 tiene comportamiento O(n)

Para los for en las líneas C4 ∧ C7 se hará un análisis similar para obtener sus límites y siempre sumando su ejecución extra al ser cabeceras:

$[0, n-1) = n-1-0 = n-1+1 = n$ ∴ la línea C4 posee comportamiento lineal $O(n)$

$[0, n/2) = n/2 - 0 = \frac{n}{2} + 1 = \frac{n+2}{2}$ ∴ la línea C7 posee comportamiento lineal $O(n)$

las líneas C3, C5, C6 ∧ C8 poseen su propio comportamiento, sin embargo al estar dentro de un for su ejecución es dada por $n/2-1, n-1, n-1$ ∧ $n/2$ respectivamente, las cuales son de comportamiento lineal $O(n)$

Con estos resultados se obtiene:

$$C1 * O(1) + C2 * O(n) + C3 * O(\lg n) * O(n) + C4 * O(n) + C5 * O(1) * O(n) + C6 * O(\lg n) * O(n) + C7 * O(n) + C8 * O(1) * O(n)$$

los comportamientos obtenidos en las líneas C1 es $O(1)$; en las líneas C2, C4, C5, C7 ∧ C8 es lineal $O(n)$; y en las líneas C3 ∧ C6 son $O(\lg n) * O(n)$ resultando en $O(n \lg n)$

Ahora se operan los 3 órdenes presentes:

∴ $O(1) + O(n) + O(n \lg n) = O(n \lg n)$ heapSort posee dicho comportamiento

```
43
44  //Mostrar el simbolo de dolar
45  void printArray(double arr[], int n) { C1 * O(1)
46      for (int i = 0; i < n; ++i) {  C2 * O(n)
47          cout << i + 1 << ". $" << fixed << setprecision(2) << arr[i] << "\n"; C3 * O(1) * O(n)
48      }
49  }
50
```

Obtiendo límites del for en la línea C2:

$[0, n) = n - 0 = n$ y al ser cabecera es $n+1$, lo cual es un comportamiento lineal $O(n)$

la línea C3 es multiplicada a su vez por el comportamiento en el for al ser parte del mismo, es decir, por $O(n)$.

Comparando toda la función:

$C1 * O(1) + C2 * O(n) + C3 * O(1) * O(n)$

$\therefore O(1) + O(n) + O(n) =$ O(n) printArray posee comportamiento lineal

```
50
51  //Obtiene la informacion del archivo donde se encuentran los datos
52  double* leerSalariosDesdeArchivo(const string& nombreArchivo, int& n) { C1 * O(1)
53      ifstream archivo(nombreArchivo); C2 * O(1)
54      string linea; C3 * O(1)
55      n = 0; C4 * O(1)
56
57      // Enumera cada salario existente, lee linea por linea
58      while (getline(archivo, linea)) { C5 * O(n)
59          n++; C6 * O(1) * O(n)                    } O(n)
60      }
61
62      archivo.clear(); C7 * O(1)
63      archivo.seekg(0); C8 * O(1)
64
65      // Arreglo nuevo para guardar los nuevos salarios ingresados
66      double* salarios = new double[n]; C9 * O(n)
67
68      int index=0; C10 * O(1)
69      double salario; C11 * O(1)
70      while (archivo >> salario) { C12 * O(n)
71      salarios [index++] = salario; C13 * O(1) * O(n)   } O(n)
72      }
73
74      //Cierra el archivo
75      archivo.close(); C14 * O(1)
76      return salarios; C15 * O(1)
77  }
78
```

Se realiza el análisis relacionando cada línea:

$$C1 * O(1) + C2 * O(1) + C3 * O(1) + C4 * O(1) + C5 * O(n) + C6 * O(n) + C7 * O(1) + C8 * O(1) + C9 * O(n) + C10 * O(1) + C11 * O(1) + C12 * O(n) + C13 * O(n) + C14 * O(1) + C15 * O(1)$$

Por lo que los valores de $C1, C2, C3, C4, C7, C8, C10, C11, C14$ ^ $C15$ son constantes $O(1)$
y $C5, C6, C9, C12$ ^ $C13$ son de orden constante $O(n)$

Con ambos resultados se tiene:

$\therefore O(1) + O(n) = O(n)$ leerSalariosDesdeArchivo es de comportamiento lineal

```
78
79  // Funcion agregar nuevo salario
80  double* agregarSalario(double* arr, int& n, double nuevoSalario) { C1 * O(1)
81      double* nuevoArr = new double[n + 1]; C2 * O(n)
82      for (int i = 0; i < n; i++) { C3 * O(n)
83          nuevoArr[i] = arr[i]; C4 * O(1) * O(n) } O(n)
84      }
85      nuevoArr[n] = nuevoSalario; C5 * O(1)
86      n++; C6 * O(1)
87      delete[] arr; C7 * O(1)
88      return nuevoArr; C8 * O(1)
89  }
90
```

Para obtener el orden del for se necesita conocer sus límites:

$[0, n) = n - 0 = n$ y al ser cabecera esto se suma más 1, quedando $n+1$, lo que es de comportamiento $O(n)$ y las líneas dentro del for también poseerán orden $O(n)$

Relacionando líneas con comportamiento constante al que se le asignará como variable A:

$$C1 * O(1) + C5 * O(1) + C6 * O(1) + C7 * O(1) + C8 * O(1)$$

y líneas de comportamiento lineal como B:

$$C2 * O(n) + C3 * O(n) + C4 * O(1) * O(n)$$

Ahora se operan ambos comportamientos:

$\therefore A + B = O(1) + O(n) =$ $O(n)$ agregar Salario es de comportamiento lineal

```
91  // Función para eliminar un salario
92  double* eliminarSalario(double* arr, int& n, int index) {
93      if (index < 0 || index >= n) {
94          cout << "Índice inválido.\n";
95          return arr;
96      }
97
98      double* nuevoArr = new double[n - 1];
99      for (int i = 0, j = 0; i < n; i++) {
100         if (i != index) {
101             nuevoArr[j++] = arr[i];
102         }
103     }
104     n--;
105     delete[] arr;
106     return nuevoArr;
107 }
```

C1 * O(1)
C2 * O(1)
C3 * O(1)
C4 * O(1)
C5 * O(n)
C6 * O(n)
C7 * O(1) * O(n)
C8 * O(1) * O(n)
} O(n)
C9 * O(1)
C10 * O(1)
C11 * O(1)

Primero se necesita conocer los valores del límite for para conocer su comportamiento:

$[0, n) = n - 0 = n$, al ser cabecera es n+1 lo que es de comportamiento lineal O(n)

las líneas dentro del for poseen también el comportamiento de los límites de su cabecera: n, siendo toda la complejidad del bucle O(n)

Colocando todas las líneas con orden semejante:

Variable "A" con O(1):

$C1 * O(1) + C2 * O(1) + C3 * O(1) + C4 * O(1) + C7 * O(1) + C8 * O(1) + C9 * O(1) + C10 * O(1) + C11 * O(1)$

Variable "B" con O(n):

$C5 * O(n) + C6 * O(n) + C7 * O(n) + C8 * O(n)$

Operando ambas variables A ∧ B:

$\therefore O(1) + O(n) =$ O(n) eliminar Salario posee complejidad lineal

```
109
110  // Funcion para buscar un salario
111  void buscarSalario(double*arr, int n, int index){ C1 * O(1)
112      if (index < 0 || index >= n) { C2 * O(1)
113          cout << "Índice inválido.\n"; C3 * O(1)
114      } else{ C4 * O(1)
115          cout << "El salario " << (index + 1) << " es: $" << fixed << setprecision(2) << arr[index] << ".\n"; C5*O(1)
116      }
117  }
118
```

Para obtener la complejidad de la función se opera cada línea entre sí:

$C1 * O(1) + C2 * O(1) + C3 * O(1) + C4 * O(1) + C5 * O(1)$

Al ser su orden el mismo entonces:

la complejidad de buscarSalario es constante O(1)

```cpp
int main() { C1 * O(1)
    int n = 0; C2 * O(1)
    double* salarios = nullptr; C3 * O(1)

    // Llama al archivo donde se encuentran los salarios
    salarios = leerSalariosDesdeArchivo("salarios.txt", n); C3 * O(n)

    // Ordenar los salarios de forma descendente
    heapSort(salarios, n); C4 * O(nlgn)

    int opcion; C5 * O(1)
    do { C6 * O(1)
        cout << "\n** Registro de Salarios **\n"; C7 * O(1)
        cout << "1. Agregar un salario\n"; C8 * O(1)
        cout << "2. Eliminar un salario\n"; C9 * O(1)
        cout << "3. Buscar un salario\n"; C10 * O(1)
        cout << "4. Ver salarios\n"; C11 * O(1)
        cout << "5. Salir\n"; C12 * O(1)
        cout << "Elija una opción: "; C13 * O(1)
        cin >> opcion; C14 * O(1)

        switch (opcion) { C15 * O(1)
            case 1: { C16 * O(1)
                double nuevoSalario; C17 * O(1)
                cout << "Ingrese el nuevo salario: "; C18 * O(1)
                cin >> nuevoSalario; C19 * O(1)
                salarios = agregarSalario(salarios, n, nuevoSalario); C20 * O(n)
                heapSort(salarios, n); C21 * O(nlgn)
                cout << "Salario agregado.\n"; C22 * O(1)
                break; C23 * O(1)
            }
            case 2: { C24 * O(1)
                int indice; C25 * O(1)
                cout << "Ingrese el índice del salario a eliminar (1 a " << n << "): "; C26 * O(1)
                cin >> indice; C27 * O(1)
                salarios = eliminarSalario(salarios, n, indice - 1); C28 * O(n)
                heapSort(salarios, n); C29 * O(nlgn)
                cout << "Salario eliminado. \n"; C30 * O(1)
                break; C31 * O(1)
            }
            case 3: { C32 * O(1)
                int indice; C33 * O(1)
                cout << "Ingrese el índice del salario a buscar (1 a " << n << "): "; C34 * O(1)
                cin >> indice; C35 * O(1)
                buscarSalario(salarios, n, indice- 1); C36 * O(1)
                break; C37 * O(1)
            }
```

```
166             }
167             case 4: { C38 * O(1)
168                 if (n > 0) { C39 * O(1)
169                     cout << "** Salarios de los empleados ** \n"; C40 * O(1)
170                     printArray(salarios, n); C41 * O(n)
171                 } else { C42 * O(1)
172                     cout << "No hay salarios para mostrar.\n"; C43 * O(1)
173                 }
174                 break; C44 * O(1)
175             }
176             case 5: { C45 * O(1)
177                 cout << "Cerrar\n"; C46 * O(1)
178                 break; C47 * O(1)
179             }
180             default: C48 * O(1)
181                 cout << "Opción no válida.\n"; C49 * O(1)
182         }
183     } while (opcion != 5); C50 * O(1)
184
185     // Libera la memoria
186     delete[] salarios; C51 * O(1)
187
188     return 0; C52 * O(1)
189 }
190
```

Primero se separan las líneas con operaciones primitivas, es decir, de igual orden constante:

$$C1 * O(1) + C2 * O(1) + C3 * O(1) + C5 * O(1) * C6 * O(1) + C7 * O(1) + C8 * O(1) + C9 * O(1) + C10 * O(1) + C11 * O(1) + C12 * O(1) + C13 * O(1) + C14 * O(1) + C15 * O(1) + C16 * O(1) + C17 * O(1) + C18 * O(1) + C19 * O(1) + C22 * O(1) + C23 * O(1) + C24 * O(1) + C25 * O(1) + C26 * O(1) + C27 * O(1) + C30 * O(1) + C32 * O(1) + C33 * O(1) + C34 * O(1) + C35 * O(1) + C36 * O(1) + C37 * O(1) + C38 * O(1) + C39 * O(1) + C40 * O(1) + C42 * O(1) + C43 * O(1) + C44 * O(1) + C45 * O(1) + C46 * O(1) + C47 * O(1) + C48 * O(1) + C49 * O(1) + C50 * O(1) + C51 * O(1) + C52 * O(1)$$

Como el comportamiento es el mismo $O(1)$ se identificará en una sola variable A

Seguidamente se identifican lineas de comportamiento lineal $O(n)$ que se identificará como B:

$$C3 * O(n) + C20 * O(n) + C28 * O(n) + C41 * O(n)$$

y la variable C se tomará con orden $O(n \lg n)$:

$$C21 * O(n \lg n) + C29 * O(n \lg n)$$

Por último se operan todas las variables:

$$A + B + C$$

$\therefore O(1) + O(n) + O(n \lg n) = O(n \lg n)$ main posee dicho comportamiento

CONCLUSIÓN

En conclusión, según los comportamientos obtenidos en cada una de las partes del código, se puede mencionar que el programa presentado es considerado bastante eficiente, debido a que el peor orden de magnitud presentado es O(nlgn), el cual se encuentra entre el rango aceptado de operaciones de ordenamiento, permitiendo manejar múltiples datos en poco tiempo.

Por otro lado, se encontraban operaciones dentro del programa que no cuentan con una respuesta fija, como en el caso del bucle do-while, ya que la cantidad de operaciones depende del usuario que hace uso del programa, por lo tanto, no se tiene previsto dentro del análisis de complejidad al no estar predefinida.

En definitiva, el programa es eficiente y completamente funcional, permitiendo realizar las operaciones solicitadas, esto debido a la implementación de heapify y heapSort, demostrando un impacto favorable la utilización de la estructura heap.