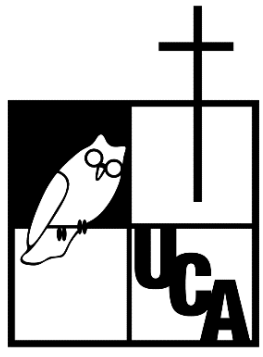
UNIVERSIDAD CENTROAMERICANA
JOSÉ SIMEÓN CAÑAS



TALLER 1

Análisis de Algoritmos

Ing. Mario Lopez

Ing. Enmanuel Araujo

Integrantes:

Maria José Campos Hernández 00010520
Gabriela Sofía Quinteros Ramírez 00060422
Fernanda Camila Vásquez Meléndez 00065221

Fecha de entrega:
viernes 6 de septiembre del 2024

```cpp
#include <iostream>
using namespace std;

// Estructura para almacenar los datos de cada producto
struct Product {
    string name;
    int stock;
};
```

$C_1 * 1 \rightarrow O(1)$ $C_2 * 1 \rightarrow O(1)$ $C_3 * 1 \rightarrow O(1)$ $C_4 * 1 \rightarrow O(1)$ $C_5 * 1 \rightarrow O(1)$

función analizada completa

1 $C_1 * (1)$
2 $C_2 * (1)$
5 $C_3 * (1)$
6 $C_4 * (1)$
7 $C_5 * (1)$

$$C_1 + C_2 + C_3 + C_4 + C_5 = O(1)$$

$\therefore O(1)$

```cpp
// Función para agregar productos al inventario
void addProduct(Product inventory[], int &count) {
    cout <<"Ingrese el nombre del producto: ";
    cin >> inventory[count].name;
    cout <<  "Ingrese la cantidad en stock: ";
    cin >> inventory[count].stock;
    count++;
}
```

$C_1 * 1 \rightarrow O(1)$ $C_2 * 1 \rightarrow O(1)$ $C_3 * 1 \rightarrow O(1)$ $C_4 * 1 \rightarrow O(1)$ $C_5 * 1 \rightarrow O(1)$ $C_6 * 1 \rightarrow O(1)$

función analizada completa

11 $C_1 * (1)$
12 $C_2 * (1)$
13 $C_3 * (1)$
14 $C_4 * (1)$
15 $C_5 * (1)$
16 $C_6 * (1)$

$$C_1 + C_2 + C_3 + C_4 + C_5 + C_6 = O(1)$$

$\therefore$ addProduct es $O(1)$

addProduct

Complejidad: O(1) Esta operación es constante ya que solo añade un nuevo producto al final del inventario, sin importar la cantidad de productos ya existentes..

```
19  // Función para eliminar un producto del inventario
20  void deleteProduct(Product inventory[], int &count) {
21      if (count == 0) { //nuevo
22          cout << "No hay productos para eliminar.\n"; //nuevo
23          return; //nuevo
24      }
25      string nameToDelete;
26      cout << "Ingrese el nombre del producto a eliminar: ";
27      cin >> nameToDelete;
28
29      bool found = false;
30      for (int i = 0; i < count; i++) {
31          if (inventory[i].name == nameToDelete) {
32              for (int j = i; j < count - 1; j++) {
33                  inventory[j] = inventory[j + 1];
34              }
35              count--;
36              found = true;
37              cout << "Producto eliminado del inventario.\n";
38              break;
39          }
40      }
41
42      if (!found) {
43          cout << "No se encontro el producto.\n";
44      }
45  }
```

$C_1 * 1 \rightarrow O(1)$

$C_2 * 1 \rightarrow O(1)$

$C_3 * max(0,1)$ $O(1)$

$C_4 * Max(0,1) \rightarrow O(1)$

$C_5 * 1 \rightarrow O(1)$

$C_6 * 1 \rightarrow O(1)$

$C_7 * 1 \rightarrow O(1)$

$C_8 * 1 \rightarrow O(1)$

$C_9 * n+1 \longrightarrow O(n)$

$C_{10} * n * 1 \rightarrow O(n)$

$C_{11} * n * max(0,1) * n-i$

$C_{12} * n * max(0,1) * (n-1)-i$

$C_{13} * n * max(0,1) \rightarrow O(n)$

$C_{14} * n * max(0,1) \longrightarrow O(n)$

$C_{15} * n * max(0,1)$ $O(n)$

$C_{16} * n * max(0,1) \longrightarrow O(n)$

$C_{17} * 1 \longrightarrow O(1)$

$C_{18} * max(0,1) \rightarrow O(1)$

20 $C_1 * 1$

21 $C_2 * 1$

22 $C_3 * max(0,1)$

23 $C_4 * max(0,1)$

$O(1)$

25 $C_5 * 1$

26 $C_6 * 1$

27 $C_7 * 1$

29 $C_8 * 1$

$O(1)$

[0, n-1]
b-a+1
(n-1)-0+1
n → cuerpo
n+1 → cabecera

[i, n-2]
b-a+1
(n-2)-i+1
(n-1)-i → cuerpo
n-i → cabecera

Sumatorias:

$\sum_{i=0}^{n-1} n-i$

$\sum_{i=0}^{n-1} (n-1)-i$

30 $C_9 * n+1$

31 $C_{10} * n * 1$ } $O(n)$

32 $C_{11} * n * max(0,1) * n-i$

33 $C_{12} * n * max(0,1) * (n-1)*i$

35 $C_{14} * n * max(0,1)$

36 $C_{15} * n * max(0,1)$

37 $C_{16} * n * max(0,1)$

38 $C_{17} * n * max(0,1)$ } $O(n)$

deleteProduct

42 $C_{18} * 1$

43 $C_{19} * max(0,1)$ } $O(1)$

Complejidad: $O(n^2)$ En el peor de los casos, se busca un producto que está al final del inventario o no existe, recorriendo todo el arreglo. Además, si se encuentra, hay que desplazar los elementos siguientes, lo que introduce otra iteración lineal, resultando en una complejidad cuadrática.

Resolviendo las sumatorias:

$C_{11} \sum_{i=0}^{n-1} n-i$

$= n + \sum_{i=1}^{n-1} (n-i)$

$= n \sum_{i=1}^{n-1} (n-i)$

$= n + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$

$= n + n(n-1) - \frac{(n-1)(n-1+1)}{2}$

$= n + n^2 - n - \frac{(n-1)(n)}{2}$

$= n^2 - \frac{n^2-n}{2}$

$\therefore C_{11}$ es $O(n^2)$

$C_{12} \sum_{i=0}^{n-1} (n-1)-i =$

$= n-1 + \sum_{i=1}^{n-1} -i-1$

$= n-1 - \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} 1$

$= n-1 - \frac{(n-1)(n-1+1)}{2} - (1)(n-1)$

$= n-1 - \frac{n^2-n}{2} - n+1$

$\therefore C_{12}$ es $O(n^2)$

$O(1) + O(11) + O(n) + O(n^2) + O(n^2) + On + O1 = O(n^2)$

$\therefore$ deleteProduct es $O(n^2)$

```
46
47  // Función para modificar la cantidad de stock de un producto
48  void updateStock(Product inventory[], int count) {
49      string nameToUpdate;
50      cout << "Ingrese el nombre del producto a modificar: ";
51      cin >> nameToUpdate;
52      bool exists = false;
53
54      for (int i = 0; i < count; i++) {
55          if (inventory[i].name == nameToUpdate) {
56              int newStock;
57              cout << "Cantidad actual de stock: " << inventory[i].stock << endl;
58              cout << "Ingrese la nueva cantidad de stock: ";
59              cin >> newStock;
60
61              // Verificar si es el mismo stock
62              if (newStock == inventory[i].stock) {
63                  cout << "El nuevo valor es el mismo que el valor actual.\n";
64                  cout << "Ingrese un valor diferente: ";
65                  cin >> newStock;
66              }
67
68              // Actualizar el stock si el valor ingresado es diferente
69              if (newStock != inventory[i].stock) {
70                  inventory[i].stock = newStock;
71                  cout << "Stock actualizado correctamente.\n";
72              }
73              exists = true;
74              break;
75          }
76      }
77      if (!exists) {
78          cout << "No se encontro el producto.\n";
79      }
80  }
81
```

$C_1 \rightarrow O(1)$

$C_2 \rightarrow O(1)$

$O(1)$

$C_4 \rightarrow O(1)$

$C_5 \rightarrow O(1)$

$C_6 \rightarrow O(n)$

$C_7 \rightarrow O(n)$

$C_8 \rightarrow O(n)$

$C_9 \rightarrow O(n)$

$C_{10} \rightarrow O(n)$

$C_{11} \rightarrow O(n)$

$C_{12} \rightarrow O(n)$

$13 \rightarrow O(n)$

$C_{14} \rightarrow O(n)$

$C_{15} \times n \rightarrow O(n)$

$C_{16} \rightarrow O(n)$

$C_{17} \rightarrow O(n)$

$C_{19} \rightarrow O(n)$

$C_{20} \rightarrow O(n)$

$C_{21} \rightarrow O(n)$

$C_{22} \rightarrow O(n)$

$C_1 * 1$

$C_2 * 1$

$C_3 * 1$

$C_4 * 1$

$C_5 * 1$

$C_1 + C_2 + C_3 + C_4 + C_5 = O(1)$

$C_6 * n + 1$

$C_7 * n * C_{23} * 1$

$C_8 * n * max(0,1)$

$C_6 + C_7 + C_8 = O(n)$

$C_9 * n * max(0,1)$
$C_{10} * n * max(0,1)$
$C_{11} * n * max(0,1)$

$C_9 + C_{10} + C_{11} = O(n)$

$C_{12} * n * max(0,1) * C_{24} * 1$
$C_{13} * n * max(0,1) * max(0,1)$
$C_{14} * n * max(0,1) * max(0,1)$
$C_{15} * n * max(0,1) * max(0,1)$

$C_{16} * n * max(0,1) * C_{25} * 1$
$C_{17} * n * max(0,1) * max(0,1)$
$C_{18} * n * max(0,1) * max(0,1)$
$C_{19} * n * max(0,1)$
$C_{20} * n * max(0,1)$

$C_{12} + C_{13} + C_{14} + C_{15} + C_{16} + C_{17} + C_{18} + C_{19} + C_{20} = O(n)$

updateStock

Complejidad: O(n) Esta operación requiere buscar un producto por su nombre. En el peor de los casos, el producto está al final o no existe, implicando un recorrido completo del inventario.

$C_{21} * C_{26} * 1$
$C_{22} * max(0,1)$
$\} O(1)$

Sumando: $O(1) + O(n) + O(n) + O(n) + O(1) = O(n)$

∴ updateStock es O(n)

```
82  // Función para ordenar los productos según el stock con BubbleSort
83  void sortByStock(Product inventory[], int count) {
84      for (int i = 0; i < count - 1; i++) {
85          for (int j = 0; j < count - i - 1; j++) {
86              if (inventory[j].stock > inventory[j + 1].stock) {
87                  Product temp = inventory[j];
88                  inventory[j] = inventory[j + 1];
89                  inventory[j + 1] = temp;
90              }
91          }
92      }
93  }
```

$C_1 \rightarrow O(1)$
$C_2 \rightarrow O(n)$
$C_3 \rightarrow O(n^2)$
$C_4 \rightarrow O(n^2)$
$C_5 \rightarrow O(n^2)$
$C_6 \rightarrow O(n^2)$
$C_7 \rightarrow O(n^2)$

```
83  void sortByStock(Product inventory[], int count) {
84      for (int i = 0; i < count - 1; i++) {
85          for (int j = 0; j < count - i - 1; j++) {
86              if (inventory[j].stock > inventory[j + 1].stock)
87                  Product temp = inventory[j];
88                  inventory[j] = inventory[j + 1];
89                  inventory[j + 1] = temp;
90              }
```

$C_1 \rightarrow O(1)$

$C_2 * n \rightarrow O(n)$

Desgloce de procedimiento: $[0, n-2]$

$b - a + 1$

$n - 2 - 0 + 1$

$n - 1 \rightarrow$ cuerpo

$n \rightarrow$ cabecera

$C_3 * (n-1) * (n-i)$

Cuerpo del for externo

Desgloce de procedimiento: $[0, n-i-2]$

$n - i - 2 - 0 + 1$

$n - i - 1 \rightarrow$ cuerpo

$n - i \rightarrow$ cabecera

$C_4 * (n-1) * (n-i-1) * C_8 * 1$

$C_5 * (n-1) * (n-i-1) * max(0,1)$

$C_6 * (n-1) * (n-i-1) * max(0,1)$

$C_7 * (n-1) * (n-i-1) * max(0,1)$

Función analizada completa

83 $C_1 * 1 \rightarrow O(1)$

84 $C_2 * n \rightarrow O(n)$

85 $C_3 * (n-1) * (n-i) \longrightarrow \sum_{i=0}^{n-2} (n-i) \rightarrow O(n^2)$

86 $C_4 * (n-1) * (n-i-1) * C_8 * 1$

87 $C_5 * (n-1) * (n-i-1) * max(0,1)$

88 $C_6 * (n-1) * (n-i-1) * max(0,1)$

89 $C_7 * (n-1) * (n-i-1) * max(0,1)$

$\sum_{i=0}^{n-2} (n-i-1) \rightarrow O(n^2)$

Resolviendo las sumatorias

$$\sum_{i=0}^{n-2} (n-i) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i$$

$$= n + \sum_{i=1}^{n-2} n - \sum_{i=1}^{n-2} i$$

$$= n + n(n-2) - \frac{(n-2)(n-1)}{2}$$

$$= O(n^2)$$

$$\sum_{i=0}^{n-2}(n-i-1) = n-1+\sum_{i=1}^{n-2}(n-i-1)$$

$$= n-1+\sum_{i=1}^{n-2}(n-i)-\sum_{i=1}^{n-2}1$$

$$= n-1+n+n(n-2)-\frac{(n-2)(n-1)}{2}-n+2$$

$$= O(n^2)$$

Entonces:

$$O(1)+O(n)+O(n)+O(n^2) = O(n^2)$$

sortByStock

$\therefore$ sortByStock es $O(n^2)$

Usa Bubble Sort para ordenar los productos por stock, con una complejidad de $O(n^2)$. Esto ocurre porque compara e intercambia elementos muchas veces, lo que es ineficiente para grandes volúmenes de datos.

```
94
95  // Función para mostrar los productos con menor stock (menor a 10 unidades)
96  void showLowStock(Product inventory[], int count, int n) {
97      if (count == 0) {
98          cout << "No hay productos.\n";
99          return;
100     }
101     sortByStock(inventory, count);
102     cout << "Productos con menor stock:\n";
103     int shown = 0;
104
105     for (int i = 0; i < count; i++) {
106         if (inventory[i].stock < 10) {
107             cout << inventory[i].name << " - Stock: " << inventory[i].stock << endl;
108             shown++;
109             if (shown == n) break;
110         }
111     }
112     if (shown == 0) {
113         cout << "No hay productos con stock menor a 10 unidades.\n";
114     }
115 }
116
```

$C_1 \rightarrow O(1)$ $C_2 \rightarrow O(1)$ $C_3 \rightarrow O(1)$ $C_4 \rightarrow O(1)$ $C_5 \rightarrow O(n^2)$ $C_6 \rightarrow O(1)$ $C_7 \rightarrow O(1)$ $C_8 \rightarrow O(n)$ $C_9 \rightarrow O(n)$ $O(n)$ C_{10} $C_{11} \rightarrow O(n)$ $C_{12} \rightarrow O(1)$ $C_{13} \rightarrow O(1)$ $C_{14} \rightarrow O(1)$

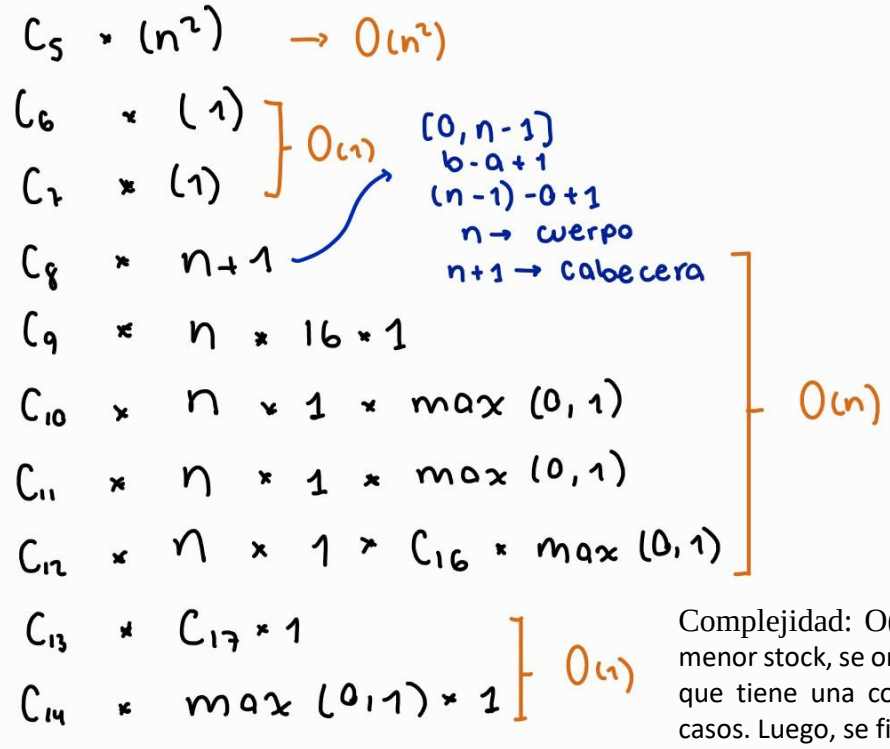
$C_1 * (1)$

$C_2 * C_{15}(1)$

$C_3 * (1) * max(0,1)$

$C_4 * (1) * max(0,1)$

$\} - O(1)$



showLowStock

Complejidad: $O(n^2)$ Antes de mostrar los productos con menor stock, se ordena el inventario utilizando Bubble Sort, que tiene una complejidad cuadrática en el peor de los casos. Luego, se filtran los productos con stock menor a 10, lo cual es lineal, pero la operación dominante es la ordenación.

$$O(1) + O(n^2) + O(1) + O(n) + O(1) = O(n^2)$$

$\therefore$ showLowStock es $O(n^2)$

```
117 int main() C1 * 1
118 {
119     Product inventory[100]; // Array para almacenar hasta 100 productos C2 * 1
120     int productCount = 0;   // Número de productos actual C3 * 1
121     int option; C4 * 1
122     do C5
123     {
124         cout << "\n=== MENU ===\n"; C6 * 1
125         cout << "1. Agregar producto\n"; C7 * 1
126         cout << "2. Eliminar producto\n"; C8 * 1
127         cout << "3. Modificar cantidad de stock\n"; C9 * 1
128         cout << "4. Mostrar productos con stock bajo\n"; C10 * 1
129         cout << "5. Salir\n"; C11 * 1
130         cout << "Seleccione una opcion: "; C12 * 1
131         cin >> option; C13 * 1
132
```

```
133         switch (option)
134         {
135         case 1:
136             addProduct(inventory, productCount);
137             break;
138         case 2:
139             deleteProduct(inventory, productCount);
140             break;
141         case 3:
142             updateStock(inventory, productCount);
143             break;
144         case 4:
145         {
146             int toShow;
147             cout << " Cuantos productos con menor stock desea ver: ";
148             cin >> toShow;
149             showLowStock(inventory, productCount, toShow);
150             break;
151         }
152         case 5:
153             cout << "Saliendo del programa.\n";
154             break;
155         default:
156             cout << "Opcion inválida, intente nuevamente.\n";
157         }
158     } while (option != 5);
159
160     return 0;
161 }
```

$C_{14} * 1$ $C_{15} * 1$ $C_{16} * O(1)$ T_1 $C_{17} * 1$ $C_{18} * 1$ $C_{19} * O(n^2)$ T_2 $C_{20} * 1$ $C_{21} * 1$ $C_{22} * O(n)$ T_3 $C_{23} * 1$ $C_{24} * 1$ $C_{25} * 1$ $C_{26} * 1$ $C_{27} * 1$ $C_{28} * O(n^2)$ T_4 $C_{29} * 1$ $C_{30} *$ $C_{31} * 1$ $O(1)$ T_5 $C_{32} * 1$ $C_{37} * 1$ $C_{34} * 1$ T_6 $O(1)$ $C_{35} * 1$ $C_{36} * 1 \longrightarrow T_6$

$C_1 * 1$
$C_2 * 1$
$C_3 * 1$
$C_4 * 1$
$C_5 * 1$
$C_6 * 1$
$C_7 * 1$
$C_8 * 1$
$C_9 * 1$

Sumatoria
$O(1)$

$C_{10} * 1$
$C_{11} * 1$
$C_{12} * 1$
$C_{13} * 1$
$C_{14} * 1$
$C_{15} * 1$
$C_{16} * O(1)$
$C_{17} * 1$
$C_{18} * 1$

Sumatoria
$O(1)$

$C_{19} * O(n^2)$
$C_{20} * 1$
$C_{21} * 1$
$C_{22} * O(n)$
$C_{23} * 1$
$C_{24} * 1$
$C_{25} * 1$
$C_{26} * 1$
$C_{27} * 1$

Sumatoria
$O(n^2)$

$C_{28} * O(n^2)$
$C_{29} * 1$
$C_{30} * 1$
$C_{31} * 1$
$C_{32} * 1$
$C_{33} * 1$
$C_{34} * 1$
$C_{35} * 1$
$C_{36} * 1$

Sumatoria
$O(n^2)$

Max $(T_1, T_2, T_3, T_4, T_5, T_6)$

max $(\underbrace{O(1)}_{T_1}, \underbrace{O(n^2)}_{T_2}, \underbrace{O(n)}_{T_3}, \underbrace{O(n^2)}_{T_4}, \underbrace{O(1)}_{T_5}, \underbrace{O(1)}_{T_6})$

$O(1)+O(n^2)+O(n)+ O(n^2)+ O(1)+ O(1)= O(n^2)$ nuestro Orden de convergencia

∴ switch $O(n^2)$

función main posee $O(n^2)$ por lo tanto, el algoritmo tiene un orden de magnitud de $O(n^2)$

∴ Algoritmo de Orden de Magnitud $O(n^2)$

Análisis Global de los resultados: En el main, estas funciones son ejecutadas dependiendo de la opción seleccionada por el usuario. Dado que las funciones de eliminación y mostrar productos con bajo stock tienen complejidades de $O(n^2)$, la complejidad total del programa también se considera $O(n^2)$ en el peor de los casos.

Conclusión sobre el resultado:

En base a los resultados del análisis de las funciones en el peor de los casos, el tiempo de ejecución del algoritmo es $O(n^2)$, lo cual significa que a medida que el tamaño del inventario crece, el tiempo necesario para completar las operaciones más costosas (como eliminar productos o mostrar aquellos con bajo stock) aumenta de manera cuadrática. Esto puede llevar a una disminución significativa del rendimiento si el inventario se hace muy grande.