

**UNIVERSIDAD CENTROAMERICANA
JOSÉ SIMEÓN CAÑAS**

DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA



ANÁLISIS DE ALGORITMOS

Ciclo 02/2024

TALLER NÚMERO 2

Estudiantes:

ALVARADO ALEGRIA, Axel Jahir, 00216022

MANZANARES SANTOS, Pedro Jose, 00230422

MONTOYA RAMIREZ,, Diego Alejandro, 00087522

Fecha de entrega: sábado 12 de octubre de 2024.

Introducción

El almacén Salem ha crecido rápidamente en eficiencia y prestigio, lo que le permite ahora distribuir bonos de fin de año a sus empleados. Sin embargo, con una nómina que incluye miles de empleados con salarios diversos, el departamento de Recursos Humanos enfrenta el desafío de ordenar estos salarios en orden descendente. Para ello, se requiere un método de ordenamiento eficiente y que consuma la menor cantidad de memoria posible, dado que los recursos de la empresa están al máximo. La elección de un algoritmo de ordenamiento rápido y confiable es crucial para garantizar que la distribución de bonos se realice de manera oportuna y precisa.

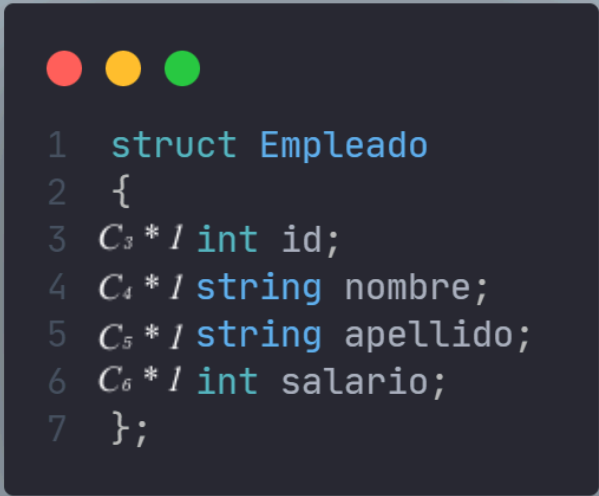
Planteamiento del Problema

La expansión de la nómina plantea un desafío logístico considerable: gestionar una extensa lista de salarios requiere un método de ordenamiento eficiente que minimice el uso de recursos de memoria.

Dado que los recursos de la empresa están siendo utilizados al máximo en otras áreas, la implementación de un algoritmo de ordenamiento adecuado es crucial para evitar retrasos en la distribución de bonos, que podrían afectar la moral y satisfacción de los empleados.

Con el objetivo de abordar esta problemática, el equipo de desarrolladores ha decidido implementar una solución utilizando el lenguaje de programación C++. Esta elección permite crear un sistema eficiente y adaptable, capaz de no solo facilitar la organización de los salarios, sino también automatizar tareas como el ordenamiento de los salarios de forma descendente.

La utilización de C++ garantizará que se cumplan los requisitos de eficiencia y rendimiento necesarios para manejar la creciente complejidad de la nómina, asegurando que Recursos Humanos pueda realizar la distribución de bonos de manera oportuna y efectiva.



```
1 struct Empleado
2 {
3     C3 * / int id;
4     C4 * / string nombre;
5     C5 * / string apellido;
6     C6 * / int salario;
7 };
```

Realizamos el análisis de la **Estructura** inicial de nuestro código, con la cual nos basamos para darle forma a los distintos campos de texto que tiene que leer el programa al momento de su ejecución.

-Línea 3: `int id`

La declaración de una variable de tipo **int** dentro de nuestra estructura, la cual consta de una complejidad constante en términos de Big O de $O(1)$, ya que la memoria interna necesaria para almacenar un entero es fija y muy pequeña.

-Línea 4 : `string nombre`

En esta línea se encuentra la declaración de la variable nombre la cual inicialmente tiene una **complejidad** en término de Big O de $O(1)$ ya que solo estamos declarando la referencia y no haciendo ninguna operación sobre nuestra cadena de texto.

-Línea 5 : `string apellido`

Esta línea es idéntica a la anterior, ya que estás declarando otra variable string para el apellido. De nuevo, la **complejidad temporal** de la declaración es $O(1)$.

-Línea 6: `int salario`

Finalmente, la declaración de la variable salario también es de tipo int, por lo que su complejidad es **constante**, $O(1)$ en términos de tiempo como de espacio. Solo se asigna un bloque fijo de memoria para almacenar un número entero.

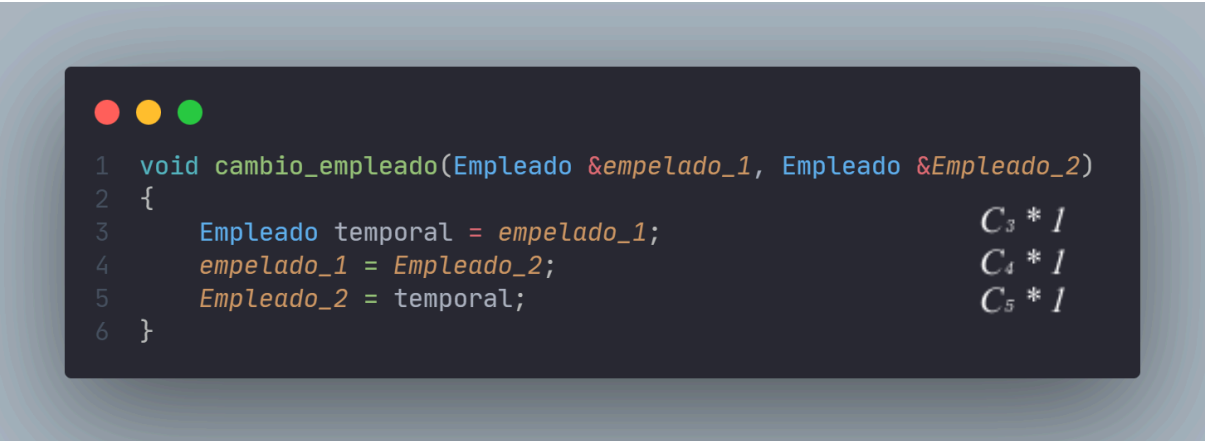
Una vez analizado el bloque de código procedemos a poder evaluar y sacar el polinomio el cual nos dará el tiempo de ejecución y magnitud de en los términos dados, ya que la **Estructura** como tal sus operaciones internas se basa en una complejidad de **$O(1)$** , obtenemos lo siguiente:

$$T(n) = (C_3 * 1) + (C_4 * 1) + (C_5 * 1) + (C_6 * 1)$$

Puesto que las C eventualmente corresponden a operaciones que son constantes y considerando las operaciones constantes al final en términos de ejecución y crecimiento a largo plazo, podemos simplificar el polinomio tal que procede a quedarnos de la siguiente manera:

$$T(n) = (1) + (1) + (1) + (1) \Rightarrow T(n) = O(1)$$

La complejidad total de la estructura es de **$O(1)$** , lo cual significa que el tiempo de ejecución de dicha estructura es de magnitud constante en la asignación de los campos del empleado.



```
1 void cambio_empleado(Empleado &empleado_1, Empleado &Empleado_2)
2 {
3     Empleado temporal = empleado_1;           C3 * 1
4     empleado_1 = Empleado_2;                 C4 * 1
5     Empleado_2 = temporal;                   C5 * 1
6 }
```

-Línea 3 : `Empleado temporal = empleado_1`

En esta línea creamos la variable temporal la cual es de tipo empleado y nos sirve para poder hacer el intercambio entre si siguiendo la lógica de nuestro algoritmo, se inicializa con el valor de **empleado_1** donde el constructor del **Empleado** copia los valores leídos de **empleado_1** en **temporal**, esto por medio de una referencia lo cual plantea que cualquier modificación que se haga a estos parámetros afectará directamente a las instancias originales, esto permite que la función no necesite

devolver un valor sino que simplemente modifica los objetos originales este proceso es de magnitud big O de **O(1)** ya que el tamaño de nuestra estructura **Empleado** es fija y no depende de ningún parámetro externo a ella.

-Línea 4 : `empleado_1 = Empleado_2`

En esta línea se asigna el valor de **empleado_2** a **empleado_1** esto implica que se copian todos los campos de empleado_2 en empleado_1 debido que esta operación es una simple asignación de valores a una variable y la estructura es determinada anteriormente como constante, tenemos que la magnitud en términos de Big O en esta línea de código es de **O(1)**.

-Línea 5: `Empleado_2 = temporal;`

El valor de temporal finalmente se asigna a **empleado_2** lo cual implica copiar todos los casos a esta nueva variable realizando de manera correcta el cambio entre si y las operaciones anteriormente mencionadas esta operación tiene una magnitud de Big O **O(1)** ya que la copia es constante y el tamaño de Empleado no cambia.

Procedemos a sacar el polinomio de tendencia de la función ya que hemos tenido cada una de las magnitudes de las líneas de código analizadas anteriormente lo cual nos da :

$$T(n) = (C_3 * 1) + (C_4 * 1) + (C_5 * 1)$$

Dado que las C corresponden a operaciones constantes, y considerando que las operaciones constantes se vuelven insignificantes podemos simplificar el polinomio de la siguiente manera: **$T(n) = O(1)$**

La complejidad total de la función es de **O(1)**, donde se visualiza que valores de ambos empleados sean intercambiados sin pérdida de datos, además de que el uso de referencias permite que el intercambio se realice en los objetos originales, evitando la necesidad de devolver un nuevo objeto o usar punteros, lo que simplifica el código.

```

1  int padre(int i)
2  {
3      return (i - 1) / 2;  $C_3 * 1$ 
4  }
5

```

-Línea 3 : `return (i - 1) / 2;`

En esta línea se declaran dos operaciones la cual es una **resta** y una **división** ambas operaciones son de tiempo constante y no dependen del tamaño del input ya que sólo están utilizando un int, por lo tanto esta operación en tiempo de magnitud Big O es de **$O(1)$** .

Quedándonos así el polinomio : $T(n) = (C_3 * 1)$ tomando la C como una constante la cual en tiempo de ejecución se vuelve irrelevante podemos decir de que la magnitud total de esta función es de $T(n) = O(1)$.

```

1  void insertar_maxHeap(Empleado empleados[], int &contador, Empleado empleado)
2  {
3      empleados[contador] = empleado;  $C_3 * 1$ 
4      int i = contador;  $C_4 * 1$ 
5      contador++;  $C_5 * 1$ 
6
7      while (i > 0 && empleados[padre(i)].salario < empleados[i].salario)  $C_7 * \log(n)$ 
8      {
9          cambio_empleado(empleados[i], empleados[padre(i)]);  $C_9 * 1 * \log(n)$ 
10         i = padre(i);  $C_{10} * 1 * \log(n)$ 
11     }
12 }

```

-Línea 3 : `empleados[contador] = empleado;`

En esta línea se asigna un nuevo **empleado** en la posición de contador del array **empleados**, La asignación de un valor en un array por índice es una operación de tiempo constante por lo tanto es de **O(1)**.

-Línea 4: `int i = contador`

En esta línea se hace la inicialización de una variable es una operación de tiempo constante **O(1)**.

-Línea 5 : `contador++`

El incremento de una variable es una operación de tiempo constante **O(1)**.

-Línea 7: `while (i > 0 && empleados[padre(i)].salario < empleados[i].salario)`

El ciclo while en este caso depende de dos comprobaciones una si $i > 0$ lo que verifica que el nodo no esté en la raíz de nuestro montículo y la otra es que verifica la propiedad del montículo se cumpla es decir que el salario analizado en momento en nuestro programa sea mayor o igual que el nodo actual

Este ciclo tiene un comportamiento logarítmico, porque en cada iteración, i se actualiza a `padre(i)`, lo que implica moverse hacia arriba en el árbol, la altura de un heap balanceado es

$O(\log n)$ y en el peor caso, el ciclo while podría recorrer la altura completa del heap.

Cuerpo del while

-Línea 9 : `cambio_empleado(empleados[i], empleados[padre(i)]);`

Esta operación intercambia dos elementos del arreglo. Dado que intercambiar dos elementos específicos por sus índices es una operación de tiempo constante, esta línea tiene una complejidad de **O(1)**, aunque esta línea está dentro de un while por lo que su tiempo de ejecución se multiplicará por el número de iteraciones totales del ciclo.

-Línea 10: `i = padre(i)`

Esta línea actualiza `i` para que apunte al índice del nodo padre en el heap el cálculo de esto en posicionamiento es de **$O(1)$** aunque Al igual que cambio del empleado, esto ocurre dentro del ciclo while, por lo que también será ejecutado $O(\log n)$ veces.

Sumando las magnitudes dadas en cada una de las líneas tenemos que el polinomio de tiempo de ejecución es el siguiente:

$$T(n) = (C_3 * 1) + (C_4 * 1) + (C_5 * 1) + (C_7 * \log(n)) + (C_9 * 1 * \log(n)) + (C_{10} * 1 * \log(n))$$

Simplificando el polinomio y agrupando términos semejantes y no tomando las constantes en el tiempo total de ejecución obtenemos :

$$T(n) = O(1) + O(1) + O(1) + O(\log(n)) + O(1 * \log(n)) + O(1 * \log(n))$$
$$T(n) = O(\log(n))$$

Está dominada por el comportamiento del ciclo que tiene una complejidad logarítmica debido a la altura del heap, las operaciones de tiempo constante no afectan significativamente el crecimiento de la función a medida que n aumenta.

```
1 void formar_monticulo(Empleado empleados[], int n, int i)
2 {
3     int mayorSalario = i;  $C_3 * 1$ 
4     int izquierda = 2 * i + 1;  $C_4 * 1$ 
5     int derecha = 2 * i + 2;  $C_5 * 1$ 
6
7     if (izquierda < n && empleados[izquierda].salario > empleados[mayorSalario].salario)  $C_7 * 1$ 
8     {
9         mayorSalario = izquierda;  $C_9 * 1$ 
10    }
11
12    if (derecha < n && empleados[derecha].salario > empleados[mayorSalario].salario)  $C_{12} * 1$ 
13    {
14        mayorSalario = derecha;  $C_{14} * 1$ 
15    }
16
17    if (mayorSalario != i)
18    {
19        cambio_empleado(empleados[i], empleados[mayorSalario]);  $C_{19} * O(1)$ 
20        formar_monticulo(empleados, n, mayorSalario);  $C_{20} * O(\log n)$ 
21    }
22 }
```


-Línea 3: `int mayorSalario = i`

La asignación de un valor a una variable es una operación de tiempo constante por lo que su magnitud es un **O(1)**.

-Línea 4: `int izquierda = 2 * i + 1`

Calcular el índice del hijo izquierdo mediante una multiplicación y una suma es una operación de tiempo constante **O(1)**.

-Línea 5: `int derecha = 2 * i + 2`

Calcular el índice del hijo de derecho de forma similar que una operación este tiempo de ejecución es constante por lo que es **O(1)**.

Primer if evaluación sobre la izquierda

-Línea 7: `if (izquierda < n && empleados[izquierda].salario > empleados[mayorSalario].salario)`

En esta línea se verifican dos condiciones de el cual es asegurar que el hijo izquierdo está dentro de los límites del arreglo esta operación es de tiempo constante **O(1)** y la comparación de salarios a su vez es igual **O(1)**.

Cuerpo del if :

-Línea 9 : `mayorSalario = izquierda`

Asignar el índice del hijo izquierdo a **mayorSalario** es una operación de tiempo constante **O(1)**.

Segundo if evaluación sobre la derecha

-Línea 12: `if (derecha < n && empleados[derecha].salario > empleados[mayorSalario].salario)`

Similar al if anterior, compara los salarios del nodo derecho y actualiza **mayorSalario** si es necesario y la complejidad sigue siendo **O(1)**.

Cuerpo del if :

-Línea 14 : `mayorSalario = derecha`

Igual que el if anterior asigna el índice evaluado a **mayorSalario** y la operación igual es **O(1)**.

-Línea 19 :

En esta línea se hace el llamado a la función de **cambio_empleados** el cual ya fue analizada anteriormente y su magnitud total en ejecución fue de **O(1)**

-Línea 20 RECURSIVIDAD : `formar_monticulo(empleados, n, mayorSalario)` ;

En esta línea se hace la recursión a la misma función **formar_monticulo** debido a que si se encuentra que uno de los hijos tiene un salario mayor que el nodo actual la llamada se hace de nuevo donde la *n* involucrada es el número de elementos en el heap y el **mayorSalario** el cual es el índice del nodo que será ajustado.

donde cada llamada recursiva efectivamente trabaja en un subárbol de tamaño reducido y el tamaño del subárbol se reduce aproximadamente a la mitad cada vez ya que un heap balanceado esto sugiere que el tamaño del problema se reduce a la mitad, lo que se traduce en un valor de $b=2$ y ya llamada recursiva es única, lo que significa que $a=1$ utilizaremos el **Teorema maestro** para resolver esta recurrencia dada

$$T(n) = T(n/2) + O(1)$$

teniendo las variables de **a = 1** y **b = 2** y **d = 0**

$$\log_b(a) \Rightarrow \log_2(1) = 0$$

Ahora comparamos $f(n)=O(1)$ con n^0 , este caso comparamos **d** la cual es 0, al ser idénticas cumple con el siguiente caso base de teorema maestro el cual nos dice que si **d = log_b(a)** entonces se aplica la magnitud de **O(n⁰log₂(n))** donde nuestra *n* elevada a la 0 da 1 lo cual nos hace tener una magnitud total de :

$$T(n) = O(\log_2(n))$$

Teniendo en cuenta las magnitudes obtenidas en cada una de las líneas de código obtenemos el polinomio : $T(n) = 4 \cdot O(1) + O(\log_2(n)) \cdot O(1)$ el cual al resolverlo nos da la magnitud total de la función de : $T(n) = O(\log_2(n))$.

```
1 void heapSort_Salarios(Empleado empleados[], int n)
2 {
3     for (int i = n / 2 - 1; i >= 0; i--)
4     {
5         formar_monticulo(empleados, n, i);
6     }
7
8     for (int i = n - 1; i > 0; i--)
9     {
10        cambio_empleado(empleados[0], empleados[i]);
11        formar_monticulo(empleados, i, 0);
12    }
13 };
```

-Línea 3: `for (int i = n / 2 - 1; i >= 0; i--)`

Este bucle recorre los elementos desde la mitad del arreglo hasta el inicio para construir el heap inicial se realiza una cantidad de iteraciones proporcional a $n/2$, por lo que este bucle es $O(n)$.

-Línea 5: `formar_monticulo(empleados, n, i);`

Esta llamada a la función `formar_monticulo` tiene una complejidad de $O(\log n)$ como se analizó anteriormente, como este ajuste ocurre $n/2$ veces, el costo del bucle será $O(n \log n)$.

-Línea 8: `for (int i = n - 1; i > 0; i--)`

Este segundo bucle también se ejecuta n veces, ya que recorre todo el montículo desde el último al primero.

-Línea 10: `cambio_empleado(empleados[0], empleados[i]);`

Esta línea realiza un intercambio simple de dos empleados, lo que tiene un valor constante $O(1)$.

-Línea 11: `formar_monticulo(empleados, i, 0)`

Se llama a **formar_monticulo**, el costo de esta operación sigue siendo $O(\log n)$, pero en este caso el bucle se ejecuta n veces por lo que el costo total de este bucle será $O(n \log n)$.

En total tenemos: $T(n) = (C1 * n \log n) + (C2 * n \log n) + (C3 * n) = O(n \log n)$

```

1 void leer_empleados(Empleado empleados[], int &contador)
2 {
3     string texto;                                OC(1)
4     ifstream archivo("Empleados.txt");          OC(1)
5
6     while (getline(archivo, texto))              OC(n)
7     {
8         Empleado empleado;                       OC(1)
9         size_t posicion;                          OC(1)
10
11         posicion = texto.find(":");               OC(1)
12         empleado.id = stoi(texto.substr(posicion + 2)); OC(1)
13         texto = texto.substr(texto.find(",") + 2); OC(1)
14
15         posicion = texto.find(' ');              OC(1)
16         empleado.nombre = texto.substr(0, posicion); OC(1)
17         texto = texto.substr(posicion + 1);      OC(1)
18
19         posicion = texto.find(',');              OC(1)
20         empleado.apellido = texto.substr(0, posicion); OC(1)
21         texto = texto.substr(posicion + 2);      OC(1)
22
23         empleado.sueldo = stoi(texto);          OC(1)
24
25         insertar_maxHeap(empleados, contador, empleado); OC(log n)
26     }
27
28     archivo.close();                             OC(1)
29 }

```

$$T_n = O(1) + O(1) + O(n) + O(1) + O(1) + O(1) + O(1)$$
$$+ O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1)$$
$$+ O(1) + O(\log n) + O(1)$$

$$T_n = O(n \log n)$$

-Línea 3: `string texto;`

Declara una variable de tipo string llamada texto. Esta variable servirá para almacenar cada línea del archivo a medida que se lea. $O(1)$

-Línea 4: `ifstream archivo("Empleados.txt");`

La apertura del archivo "Empleados.txt" tiene un costo constante, ya que es una operación que no depende del tamaño del archivo. $O(1)$

-Línea 6: `while (getline(archivo, texto))`

Este bucle se ejecuta una vez por cada línea del archivo, lo que significa que el número de iteraciones depende del número de empleados en el archivo. $O(n)$

-Línea 8: `Empleado empleado;`

Se declara la variable empleado por lo tanto es una operación de costo constante y ocurre en cada iteración del bucle. $O(1)$

-Línea 9: `size_t posicion;`

Se declara una variable de posición para almacenarlas de los separadores en cada línea. $O(1)$

-Línea 11: `posicion = texto.find(": ");`

Es una operación de tipo string y esta línea busca la posición del separador y lo convierte parte del texto en un entero por lo tanto el costo es $O(1)$

-Línea 12: `empleado.id = stoi(texto.substr(posicion + 2));`

Esta también es una operación de tipo string que depende de la longitud de la línea. $O(1)$

-Línea 13: `texto = texto.substr(texto.find(",") + 2);`

Este recorte de string es una operación que también depende de la longitud por lo que nos da un costo de $O(1)$

-Línea 15: `posicion = texto.find(' ');`

Busca el primer espacio ' ' en la cadena texto. El espacio es utilizado como separador para delimitar el nombre del empleado dentro de la línea. $O(1)$

-Línea 16: `empleado.nombre = texto.substr(0, posicion);`

Esta operación de búsqueda y extracción de string tienen un costo de $O(1)$.

-Línea 17: `texto = texto.substr(posicion + 1);`

Utiliza el método substr () para obtener un nuevo fragmento de la cadena, comenzando desde la posición inmediatamente posterior al primer espacio (posición +1) .La cadena original en texto se reemplaza por este nuevo fragmento descartando el nombre ya extraído. $O(1)$

-Línea 19: `posicion = texto.find(',');`

Busca el primer espacio ' ' en la cadena texto. El espacio es utilizado como separador para delimitar el nombre del empleado dentro de la línea. $O(1)$

-Línea 20: `empleado.apellido = texto.substr(0, posicion);`

Utiliza el método `substr()` para obtener la subcadena que contiene el apellido del empleado, desde la posición 0 hasta justo antes de la coma (especificada por posición). Esta subcadena se almacena en el campo apellido del objeto empleado $O(1)$

-Línea 21: `texto = texto.substr(posicion + 2);`

Esta línea actualiza la variable texto utilizando el método `substr()`, de manera que la nueva cadena comience justo después de la coma y un espacio. La expresión `posición + 2` asegura que saltemos dos caracteres $O(1)$

-Línea 23: `empleado.salario = stoi(texto);`

Convierte una cadena en un entero tiene un costo constante $O(1)$

-Línea 25: `insertar_maxHeap(empleados, contador, empleado);`

Inserta un objeto empleado en el max-heap. A medida que cada empleado es leído y procesado, esta función lo agrega al arreglo empleados, asegurando que el ordenamiento por salarios se mantenga mediante la estructura del heap. $O(\log n)$

-Línea 28: `archivo.close();`

El cierre del archivo tiene un costo constante $O(1)$

$T(n) = O(1) + O(n) + O(n \log n) = O(n \log n)$



```
1 void mostrar_salarios(Empleado empleados[], int contador)
2 {
3     heapSort_Salarios(empleados, contador);
4
5     for (int i = contador - 1; i >= 0; i--)
6     {
7         cout << "ID: " << empleados[i].id << ", "
8             << empleados[i].nombre << ", "
9             << empleados[i].apellido << ", Salario: $"
10            << empleados[i].salario << endl;
11     }
12 }
```

-Línea 3: `heapSort_Salarios(empleados, contador)`

La función **heapsort_salarios** tiene un costo de $O(n \log n)$ como ya analizamos.

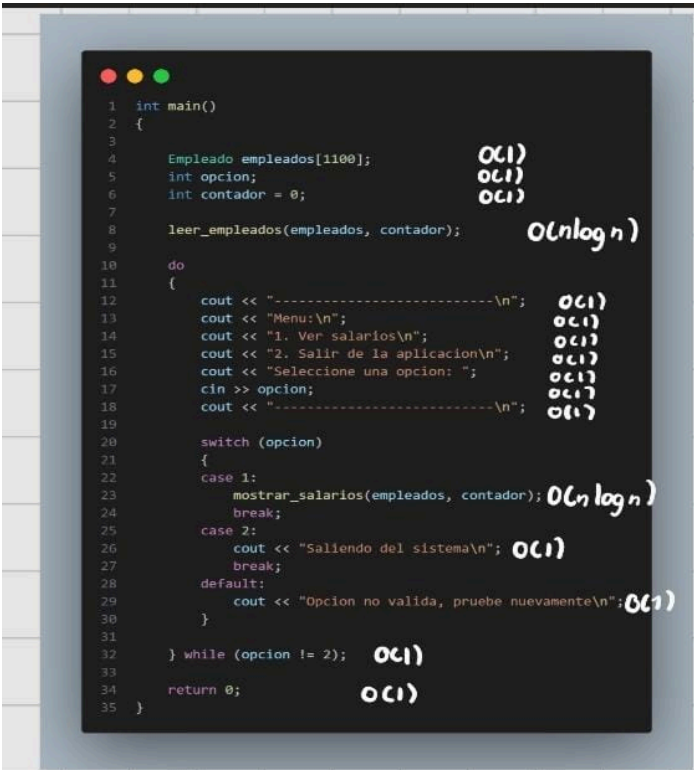
-Línea 5: `for (int i = contador - 1; i >= 0; i--)`

Este bucle for se ejecuta n veces, ya que recorre todos los empleados ordenados desde el último hasta el primero, por lo que tiene Costo: $O(n)$.

-Línea 7-10: `cout << "ID: " << empleados[i].id << ", " << empleados[i].nombre << ", " << empleados[i].apellido << ", Salario: $" << empleados[i].salario << endl;`

Cada impresión de un empleado con cout es una operación que tiene costo constante $O(1)$, pero se repite n veces dentro del bucle. Entonces, el costo total para estas operaciones es $O(n)$.

Costo total: $T(n) = O(n \log n) + O(n) = O(n \log n)$



```

1  int main()
2  {
3
4      Empleado empleados[1100];           O(1)
5      int opcion;                         O(1)
6      int contador = 0;                   O(1)
7
8      leer_empleados(empleados, contador); O(n log n)
9
10     do
11     {
12         cout << "-----\n";           O(1)
13         cout << "Menu:\n";               O(1)
14         cout << "1. Ver salarios\n";      O(1)
15         cout << "2. Salir de la aplicacion\n"; O(1)
16         cout << "Seleccione una opcion: "; O(1)
17         cin >> opcion;                   O(1)
18         cout << "-----\n";           O(1)
19
20         switch (opcion)
21         {
22             case 1:
23                 mostrar_salarios(empleados, contador); O(n log n)
24                 break;
25             case 2:
26                 cout << "Saliendo del sistema\n"; O(1)
27                 break;
28             default:
29                 cout << "Opcion no valida, pruebe nuevamente\n"; O(1)
30         }
31     } while (opcion != 2); O(1)
32
33     return 0; O(1)
34 }

```

$$T_n = O(1) + O(1) + O(1) + O(n \log n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(n \log n) + O(1) + O(1) + O(1) + O(1) + O(1)$$

$$T_n = O(1) + O(n \log n) + O(1) + O(1) + O(n \log n)$$

$$T_n = O(n \log n)$$

-Línea 5: `Empleado empleados[1100];`

Se declara un arreglo de empleados con una capacidad máxima de 1100 empleados. Esta operación es de costo constante. $O(1)$

-Línea 6: `int opcion;`

Se declara la variable opción que contendrá la selección del menú. Es una operación de costo constante. $O(1)$

-Línea 7: `int contador = 0;`

Se inicializa la variable contador que llevará la cuenta de cuántos empleados se han leído. También es una operación de costo constante. $O(1)$

-Línea 9: `leer_empleados(empleados, contador);`

Esta línea llama a la función leer_empleados, que tiene una complejidad de $O(n \log n)$ ya que depende del número de empleados que se lean desde el archivo. $O(n \log n)$

-Línea 10-19: En este segmento de líneas de código se ejecuta una serie de impresiones de texto, lo que tiene un valor constante de $O(1)$

-Líneas 20-31 (Switch - Menú de opciones):

Case 1: Si la opción seleccionada es 1, se llama a la función mostrar_salarios, que tiene una complejidad de $O(n \log n)$

Case 2: Este caso simplemente imprime un mensaje y termina el ciclo. Es de costo constante. $O(1)$

Default: Si se introduce una opción inválida, se imprime un mensaje de error. Esto también es de costo constante. $O(1)$

-Línea 35: `return 0;`

Este es el fin del programa y tiene un costo constante $O(1)$

$$T(n) = O(1) + O(n \log n) + O(n \log n) = O(n \log n)$$

Podemos apreciar la magnitud total del programa siendo $O(n \log n)$ lo que significa que el tiempo de ejecución del programa crecerá de forma eficiente respecto al número de empleados.

Reflexión acerca de los resultados obtenidos:

Por medio del análisis detallado del código implementado para el almacén Salem como equipo encontramos que su complejidad total es $O(n \log n)$. Esto se debe al uso eficiente del algoritmo (Heaps) para administrar los salarios de los empleados. Incluso para miles de empleados esto garantiza un ordenamiento de salarios rápidos. La decisión de ocupar esta estructura permite insertar a cada empleado con eficiencia manteniendo la propiedad del heap en cada inserción lo que resulta muy importante para preparar los bonos de fin de año.

Dado que la empresa ha crecido significativamente y ahora maneja una nómina amplia, era necesario implementar un método que optimice tanto el tiempo como el uso de memoria. La complejidad $O(n \log n)$ garantiza que a medida que aumenta el número de empleados, el tiempo de ejecución aumentará de manera controlada sin tener un impacto significativo en el rendimiento. Ya que el problema de la distribución de bonos también era importante resolver y se puede comprobar que se le dio una solución eficiente. Por lo que se puede ver al final del proceso permite que los salarios se ordenen en forma descendente en un tiempo eficiente.

En conclusión la implementación analizada resuelve de manera efectiva el problema planteado por el almacén Salem permitiendo gestionar de manera eficiente con una complejidad de $O(n \log n)$. Esto asegura que incluso con miles de empleados los bonos se distribuyan sin demoras cumpliendo con las expectativas y peticiones del cliente.