Algoritmos de Ordenamiento

Por Ariel Parra

[ΓΩ5]

Algoritmos de ordenamiento cuadraticos

Burbuja (Bubble Sort): O(n * (n - 1)) ≈ O(n²)

El algoritmo de Burbuja compara e intercambia elementos adyacentes si están en el orden incorrecto. Cada pasada a través del arreglo realiza aproximadamente N - 1 comparaciones, y en cada iteración, el rango efectivo de comparación disminuye.

```
for (int i = 0; i < n - 1; ++i)
    for (int j = 0; j < n - i - 1; ++j)
        if (vec[j] > vec[j + 1])
            swap(vec[j], vec[j + 1]);
```

Selección O(n * (n - 1) / 2) ≈ O(n²)

El algoritmo de Selección busca el elemento más pequeño en la lista no ordenada y lo coloca en su posición correcta al final de cada iteración.

```
for (int i = 0; i < n - 1; ++i) {
    int minIndex = i;
    for (int j = i + 1; j < n; ++j) {
        if (vec[j] < vec[minIndex]) minIndex = j;
        swap(vec[i], vec[minIndex]);
    }
}
```

Inserción O(n * (n - 1) / 2) ≈ O(n²)

El algoritmo de Inserción construye una sublista ordenada y coloca los elementos restantes en su lugar, uno por uno, en su posición correcta dentro de la sublista.

```
for (int i = 1; i < n; ++i) {
    int key = vec[i];
    int j = i - 1;

    while (j >= 0 && vec[j] > key) {
        vec[j + 1] = vec[j];
        j--;
    }
    vec[j + 1] = key;
}
```

Algoritmos de ordenamiento logarítmicos

Los algoritmos de ordenamiento logarítmicos usan el paradigma de **Divide y Vencerás (Divide and Conquer)**. Esto implica dividir el problema en subproblemas más pequeños, resolverlos de manera recursiva y luego combinar las soluciones.

Merge Sort O(n * log n)

Merge Sort divide el arreglo en dos mitades, las ordena de manera recursiva y luego combina las dos mitades ordenadas.

```
void mergeSort(std::vector<int>& vec, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);
        merge(vec, left, mid, right);
    }
}
```

Función auxiliar

```
void merge(vector<int>& vec, int left, int mid, int right) {
    int n1 = mid - left + 1; n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = vec[left + i];
    for (int j = 0; j < n2; j++) R[j] = vec[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
              vec[k] = L[i]; i++;
        } else {
            vec[k] = R[j]; j++;
        }
        k++;
    }
    while (i < n1) { vec[k] = L[i]; i++; k++; }
    while (j < n2) { vec[k] = R[j]; j++; k++; }
}
```

Quick Sort O(n * log n) en promedio, O(n²) en el peor caso

Quick Sort elige un elemento llamado *pivote*, coloca todos los elementos menores a su izquierda y los mayores a su derecha, y luego ordena recursivamente las sublistas.

```
void quickSort(std::vector<int>& vec, int low, int high) {
    if (low < high) {
        int pi = partition(vec, low, high);

        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}
```

Función auxiliar

```
int partition(std::vector<int>& vec, int low, int high) {
    int pivot = vec[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (vec[j] < pivot) {
            i++;
            std::swap(vec[i], vec[j]);
        }
    }
    std::swap(vec[i + 1], vec[high]);
    return (i + 1);
}
```

Heap Sort O(n * log n)

Heap Sort utiliza un **Heap (Montículo)** para ordenar un arreglo. Un heap es una estructura de datos en forma de árbol donde el nodo padre es mayor (para un max-heap) o menor (para un min-heap) que sus hijos.

```
void heapSort(vector<int>& vec, int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(vec, n, i);
    }

    for (int i = n - 1; i >= 0; i--) {
        std::swap(vec[0], vec[i]);
        heapify(vec, i, 0);
    }
}
```

Función auxiliar

```
void heapify(vector<int>& vec, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && vec[left] > vec[largest])
        largest = left;
    if (right < n && vec[right] > vec[largest])
        largest = right;
    if (largest != i) {
        swap(vec[i], vec[largest]);
        heapify(vec, n, largest);
    }
}
```

Comparación de algoritmos de ordenación

Sorting Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(\log N)$
Radix Sort	$\Omega(N k)$	$\Theta(N k)$	$O(N k)$	$O(N + k)$
Count Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$	$O(k)$
Bucket Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$	$O(N)$

Estabilidad en algoritmos de ordenamiento

> Se dice que un algoritmo de ordenamiento o clasificación es estable si dos objetos con claves iguales aparecen en el mismo orden en la salida ordenada que aparecen en el conjunto de datos de entrada.

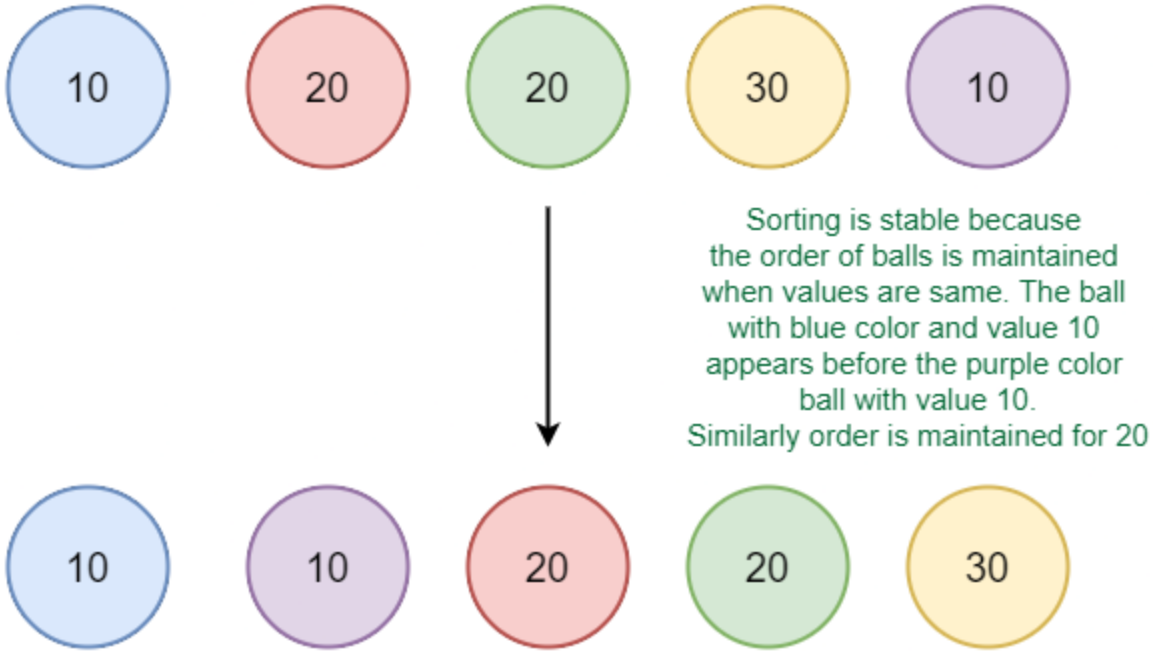

Tabla de Estabilidad

Sorting Algorithm	Best Case	Worst Case	Passes	Sort Stability
Bubble Sort	$\Omega(n^2)$	$O(n^2)$	n-1	Stable
Selection Sort	$\Omega(n^2)$	$O(n^2)$	n-1	Unstable (can be stable using Linked List)
Insertion Sort	$\Omega(n)$	$O(n^2)$	n-1	Stable
Quick Sort	$\Omega(n \log n)$	$O(n^2)$	log n (best), n-1 (worst)	Unstable
Merge Sort	$\Omega(n \log n)$	$O(n \log n)$	log n	Stable
Shell Sort	$\Omega(n)$	$O(n^2)$	log n	Unstable
Radix Sort	$\Omega(n)$	$O(n)$	Number of digits in the largest number	Stable

Implementación en C++

En C++, la función `std::sort()` se implementa utilizando el algoritmo **Intro Sort**. Este algoritmo combina tres algoritmos de ordenamiento estándar: **insertion sort**, **quick sort** y **heap sort**. Está diseñado para elegir el mejor algoritmo que se ajuste al caso dado. En resumen:

- Para conjuntos de datos pequeños, utiliza **insertion sort**.
- Para conjuntos de datos grandes, utiliza **quick sort**.
- Si la profundidad de recursión de quicksort supera un límite especificado, cambia a **heap sort**.

`std::sort()` **no es estable**, es decir, puede cambiar el orden relativo de elementos que sean iguales. Si necesitas un algoritmo estable, usa `std::stable_sort()`, el cual generalmente utiliza **merge sort**. Por lo tanto, `std::stable_sort()` preserva el orden físico de valores semánticamente equivalentes y está garantizado con una complejidad de $\mathbf{O(n \log^2 n)}$.

No podemos utilizar `std::sort()` con contenedores que no tengan acceso aleatorio, ya que quicksort y heapsort requieren acceso aleatorio a los elementos del contenedor. Sin embargo, la STL proporciona métodos especializados para ordenar contenedores sin acceso aleatorio, como es el caso de **list**. Para estos, existe el método `list::sort()`.

Usos de std::sort

Orden de ascendiencia

```
sort(vec.begin(), vec.end());//ascending

sort(vec.begin(), vec.end(), greater<int>());//non ascending (descending)
```

Orden propio con lambda

```
auto myOrder = [](pair<int, int> p1, pair<int, int> p2) { return p1.first < p2.first; };
vector<pair<int, int>> vec_pair = { {3, 1}, {2, 5}, {1, 4} };
sort(vec_pair.begin(), vec_pair.end(), myOrder);
```

Algoritmo más simple para validar anagramas

```
bool isAnagram(string& s1, string& s2) { // O(m*log(m) + n * log(n))
    if (s1.size() != s2.size())
        return false;

    sort(s1.begin(), s1.end());
    sort(s2.begin(), s2.end());

    return s1 == s2;
}
```

Problemas

- **339A** Helpful Maths ⮭
- **863B** Kayaking ⮭
- **405A** Gravity Flip ⮭

Referencias

- SW Hosting. (s.f.). *Algoritmos de ordenación con ejemplos en C++*. Recuperado de https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c
- UnexpectedValue. (2022). *A Time Complexity Guide*. Recuperado de https://codeforces.com/blog/entry/104888
- Tuteja, S. (2024). *Analysis of different sorting techniques*. Recuperado de https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/
- Manwani, C. (2023). *Stable and Unstable Sorting Algorithms*. Recuperado de https://www.geeksforgeeks.org/stable-and-unstable-sorting-algorithms/
- Agrawal, S. (2024). *std::sort() in C++ STL*. Recuperado de https://www.geeksforgeeks.org/sort-c-stl/