SORT

Marco histórico

Cuando nos referimos a ordenamiento en la computación, lo hacemos en la forma más vulgar, ya que nuestra misión en esta ocasión será tomar un conjunto de elementos y acomodarlos bajo la regla que convengamos. A simple vista esto podría parecer una tarea trivial pero no es así, ya que ha tomado mucho tiempo y mentes para ser mejorada con el paso del tiempo.

Como se pudo ver en el video del tema, el problema no es cosa fácil ya que en este se busca la o las permutaciones eficientes para ordenar la lista de menor a mayor, lo cual representa un costo considerable durante el tiempo de ejecución, llegando a complejidades de **$O(n^2)$**, como ejemplo de estas tenemos:

Simple Sort

1. **Insertion Sort:** Este algoritmo va tomando números de 1 en 1 y los coloca en el lugar que les corresponde de la lista ordenada. Es útil para listas de datos muy pequeñas.
2. **Selection Sort:** Este algoritmo toma el mínimo del conjunto y lo coloca al inicio, luego toma el segundo mínimo distinto de él y lo coloca detrás del primero y así consecutivamente.
3. **Bubble Sort:** Toma dos elementos del conjunto, lo compara y si uno es menor que otro lo invierte de posición y así hasta que concluye de ordenar todos.

Por otra parte, los mismo años de estudio que han tenido este tema nos permite contar con algoritmos de mayor eficiencia que logran complejidades de **$O(n \log n)$**, tales son los casos de:

Ordenamientos Eficientes

1. **Merge Sort:** Este es un algoritmo de "divide y vencerás" en donde divide la lista en n sublistas, si estas tienen tamaño 1 entonces están ordenadas, en otro caso divide estas aproximadamente a la mitad y luego seguir ordenando con el mismo algoritmo hasta que la última lista se la lista original, completamente ordenada.
2. **Quicksort:** Este es un algoritmo de "divide y vencerás" a diferencia de su competidor directo este toma el primer elemento como pivote y busca dentro de los demás valores si es que existe uno menos, si esto pasa entonces lo coloca de

un lado y deja a los demás del otro, después toma un nuevo pivote en las dos sublistas y repite el proceso.

Sorting funcional

Cuando se está en una competencia, lo que importa es qué tan rápido implementemos un problema, por tanto, aunque este es un problema muy bonito se necesita de la implementación más eficiente en cuanto a tiempo, por lo que haremos uso de las funciones de biblioteca. La función **sort()** pertenece a la biblioteca **#include <algorithm>**, esta te permite ordenar un conjunto de elementos **[first, last)** de forma ascendente usando el algoritmos **Quicksort**. Con **first** y l**ast**, nos referimos al principio y final de nuestro conjunto de elementos.

Implementación

sort(inicio del conjunto, final del conjunto)

Como se mencionó en la parte de arriba, al usar esta función estaremos ordenando de menor a mayor, por default, sin embargo si deseamos hacer nuestro propio parámetro de ordenamiento podremos hacer uso de función ó clase, con fines de continuidad recomendamos usar una función para llevar a cabo esto.

Implementación

función parámetro:

```
bool mayor_que(int i, int j) {
     return (i>j);

}
```

función Sort:

sort(inicio del conjunto, final del conjunto, mayor_que)

Notemos que nuestra función parámetro es del tipo bool, ya que nos dirá si es que los elementos comparados por el algoritmo **Quicksort** cumplen la condición deseada. Por parte de la función **sort()** propia de las bibliotecas, sólo hace falta mandar un tercer parámetro, correspondiente a nuestra función parámetro.