UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

# Shell Sort

Macarena Oyague
macarena.oyague@utec.edu.pe

Luis Jáuregui
luis.jauregui@utec.edu.pe

April 13, 2020

Ciencia de la Computación

# Implementación

```
void ShellSort (vector<T> & vec){
    int N = vec.size();
    int size = N;
    bool somethingChange = false;
    T temp;
    while (N != 1)
    {
        N = N/2;
        do{
            somethingChange = false;
            for (int i = 0; i<size-N; i++)
            {
                if (vec[i] > vec[i+N])
                {
                    temp = vec[i];
                    vec[i] = vec[i+N];
                    vec[i+N] = temp;
                    if (!somethingChange)
                        somethingChange = true;
                }
            }
        }while (somethingChange);
    }
}
```

# Limitaciones

- **¿Cuándo usarlo?**
  - Cuando la colección de datos que queramos ordenar se encuentra parcial o moderadamente ordenada.
  - Cuando la recursión usada en otros métodos de ordenamiento hayan excedido su límite.

# Limitaciones

- **¿Cuándo usarlo?**
  - Cuando la colección de datos que queramos ordenar se encuentra parcial o moderadamente ordenada.
  - Cuando la recursión usada en otros métodos de ordenamiento hayan excedido su límite.
- **¿Dónde usarlo?**
  - En arreglos de pequeño o moderado tamaño.
  - Como es un método de ordenamiento in-place no necesita espacio extra en el stack y, por ende, no hace llamados a este segmento de datos.
  - Generalmente se usa en sistemas embebidos por su código simple. No se implementa su uso en aplicaciones grandes, debido a su alto *cache miss ratio*.

# Estabilidad

- **¿Es estable?**
  - No se le considera un método de ordenamiento estable, porque puede cambiar el orden relativo de elementos con valores iguales.
  - Además, cuando se ordenan arreglos con ciertos tamaños (i.e. 4) y gaps específicos (i.e. Hibbard), se mueven elementos sin examinar los que están en el medio.

# Complejidad

- **Complejidad Temporal**
  - Su mejor perfomance se da en $O(n \log n)$ cuando el arreglo de datos ya se encuentra ordenado, ya que serían solo bucles ejecutándose sin realizar *swaps*.
  - El caso promedio depende mucho del gap que se desee usar. En distintas pruebas realizadas se obtiene un valor entre $O(n)$ y $O(n^2)$.
  - Su peor perfomance se da en $\Omega(n^2)$ cuando el arreglo tiene todos los elementos pares mayores que el elemento medio. Las comparaciones no se dan hasta que el gap sea 1. Es por eso que se reduce a la misma complejidad del **Insertion Sort**.

# Complejidad

- **Complejidad Temporal**
  - Su mejor perfomance se da en $O(n \log n)$ cuando el arreglo de datos ya se encuentra ordenado, ya que serían solo bucles ejecutándose sin realizar *swaps*.
  - El caso promedio depende mucho del gap que se desee usar. En distintas pruebas realizadas se obtiene un valor entre $O(n)$ y $O(n^2)$.
  - Su peor perfomance se da en $\Omega(n^2)$ cuando el arreglo tiene todos los elementos pares mayores que el elemento medio. Las comparaciones no se dan hasta que el gap sea 1. Es por eso que se reduce a la misma complejidad del **Insertion Sort**.
- **Complejidad Espacial**
  - Está en el orden de $O(n)$ para el arreglo (el espacio que ocupaba originalmente) y de $O(1)$ para el temporal auxiliar que usamos para el swap de elementos.