Diego Lamus
Eileen Guerrero

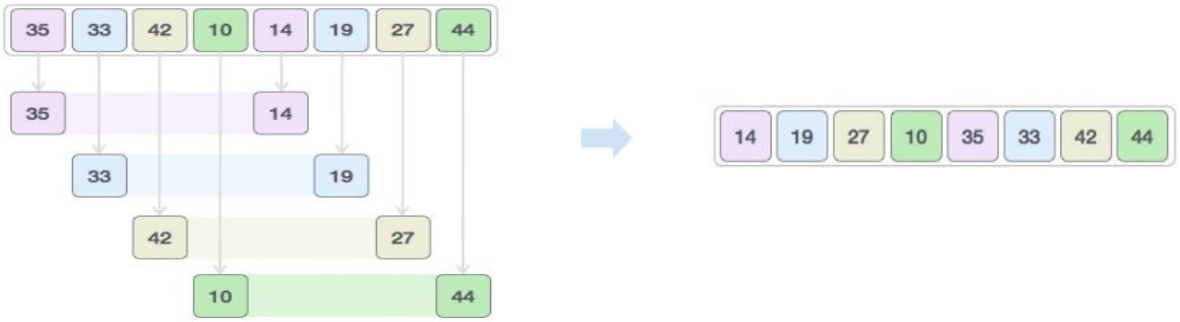
Análisis de escalabilidad

ShellSort

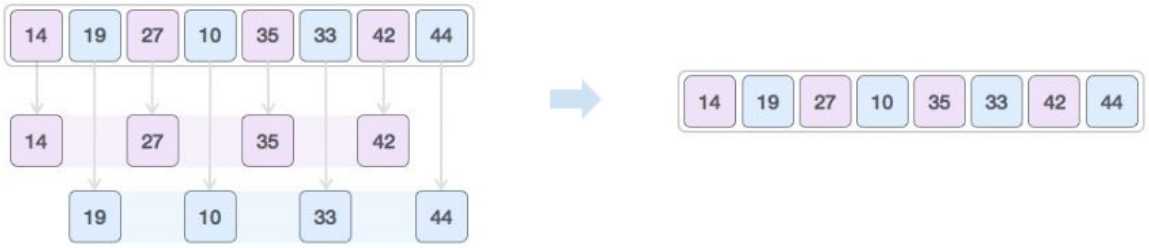
El algoritmo ShellSort ordena tomando parejas, comparándolas entre ellas, e intercambiandolo de ser necesario. El algoritmo se define en pasadas; en la primera pasada, se calcula k = n/2, donde n es el número de datos del arreglo; en la segunda pasada k = k/2, y así consecutivamente hasta llegar a 1. En cada pasada, cada dato se compara con el dato k veces adelante; si se intercambia por lo menos un elemento, se repite la pasada, hasta que ya no existan cambios.

Ejemplo

1. Primera pasada:



2. Segunda pasada:



Diego Lamus
Eileen Guerrero

3. Tercera pasada:

Dado este algoritmo, no podemos utilizar ninguna estrategia de divide y vencerás, es decir, particionando el problema en problemas más pequeños, dado que al unir las pequeñas soluciones tendríamos que recorrer nuevamente todos los datos para ordenar los datos que no se han comparado aun. De esta forma, y teniendo en cuenta que se deben comparar posiciones alejadas entre sí del arreglo, podemos concluir que hay alta dependencia de datos y que el shell sort no es distribuible.

Implementación del algoritmo:

```
public final String[] sort(final String[] cadena){

    int k, i;
    String temp;
    boolean cambios;
    for(k=cadena.length/2; k!=0; k/=2){
        cambios=true;
        while(cambios){ // Mientras se intercambie algún elemento
           cambios=false;
           for(i=salto; i< cadena.length; i++) // se da una pasada
              if(cadena[i-k].compareTo(cadena[i])>0){
                 temp=cadena[i]; // se intercambian los elementos
                 cadena[i]=cadena[i-k];
                 cadena[i-k]=temp;
                 cambios=true; // se registra un cambio para repetir la pasada
              }
        }
    }
    return cadena;
}
```

Diego Lamus
Eileen Guerrero

Gráfica de tamaño de entrada vs tiempo en milisegundos en dar una respuesta

La gráfica anterior muestra que entre más datos se ordenen, mayor tiempo tomará. 20,000 datos se usan 231 milisegundos, para 100,000 se usan 1191 milisegundos.

Teniendo en cuenta que el algoritmo no es distribuible, que los equipos se pueden escalar verticalmente hasta cierto límite, y que los tiempos en milisegundos aumentan con respecto a la carga de trabajo, es posible afirmar que el algoritmo de shell sort no es escalable. Ante cargas crecientes de trabajo el servidor no podrá incrementar su desempeño.

Como este método tiene dependencia de datos, no hay ninguna estructura de instrucciones y datos que permita la implementación en paralelo, el flujo se instrucciones y datos debe hacerse con estructura **SISD**.

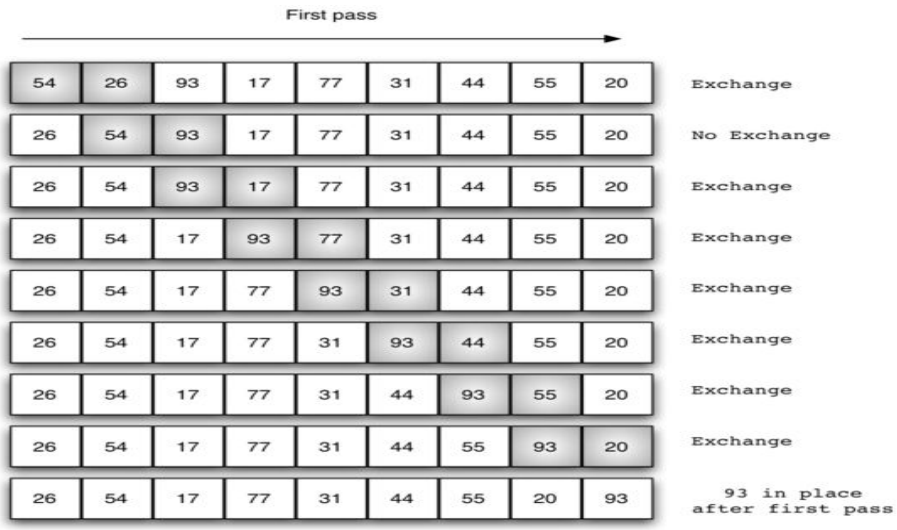
Diego Lamus
Eileen Guerrero

BubbleSort

El algoritmo de bubble sort consiste en comparar cada pareja consecutiva y intercambiarlas de ser necesario. Se recorre el arreglo n veces, y cada vez que se recorre se recorre hasta n - j veces, donde j es el número de veces que se ha recorrido iniciando en 1. Lo que se espera en cada pasada es que el elemento más grande se lleve hasta la última posición recorrida.

En el caso del algoritmo Bubble Sort existe dependencia de datos y no es imposible idear una forma de distribuirlo, pues el ordenamiento se hace recorriendo el arreglo de elementos de forma secuencial y comparando i con i-1, siendo i el índice con el cual me muevo dentro del arreglo. Para cada iteración es necesario recorrer de manera secuencial todos los campos que aún restan por ordenar del arreglo el arreglo.

Ejemplo



En la imagen anterior se observa la primera iteración del algoritmo, y como se explicó anteriormente, y se observa que es necesario recorrer el arreglo múltiples en su totalidad, al menos para la parte que aún hay que ordenar.

Si efectuamos cualquier tipo de división de los archivos para ordenar por partes, al momento de intentar unir los fragmentos, usando el método bubble sort lo que hará será recorrer nuevamente todo el vector para ordenarlo. Por tal razón, no es posible desplegar el algoritmo Bubble Sort de forma distribuida y existiría dependencia de datos.

Como este método tiene dependencia de datos, no hay ninguna estructura de instrucciones y datos que permita la implementación en paralelo, el flujo se instrucciones y datos debe hacerse con estructura **SISD**.

Diego Lamus
Eileen Guerrero

.

implementación del algoritmo

```
public final String[] sort(final String[] cadena)
{
   int n = cadena.length;
   String temp = "";
   for(int i=0; i < n; i++){
          for(int j=1; j < (n-i); j++)
                if( cadena[j-1].compareTo(cadena[j]) > 0){
                       //swap elements
                       temp = cadena[j-1];
                       cadena[j-1] = cadena[j];
                       cadena[j] = temp;
                }
   }
   return cadena;
}
```

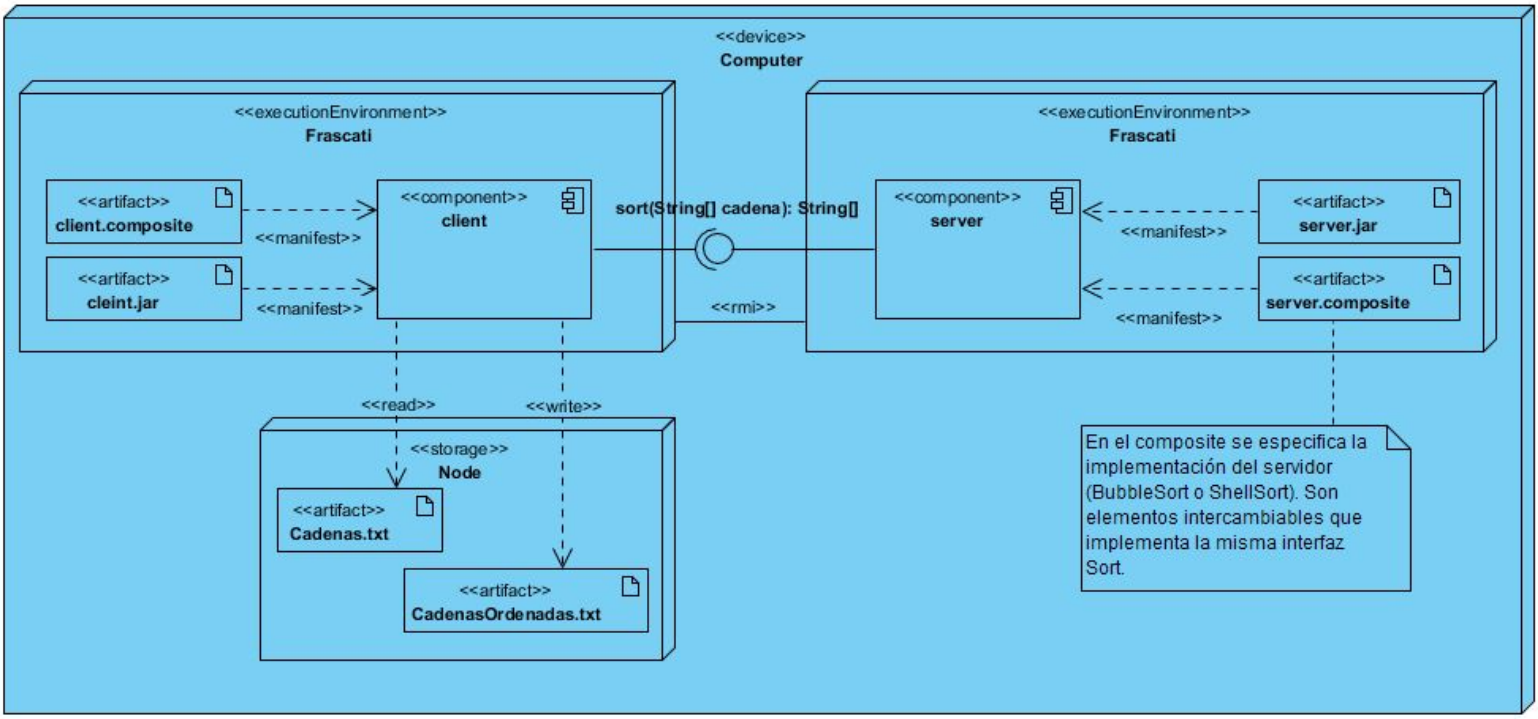
Gráfica tamaño de entrada vs tiempo en milisegundos en dar una respuesta

La gráfica anterior muestra el tiempo en milisegundo que se toma el bubble sort ordenando diferentes tamaños de arreglo. Como se observa, entre mayor es la carga de trabajo, mayor será el tiempo empleado. Para 20,000 datos se emplean 2691 milisegundos, para 20000, se emplean 127659 milisegundos.

Teniendo en cuenta que el algoritmo no es distribuible, que los equipos se pueden escalar verticalmente hasta cierto límite, y los tiempos en milisegundos aumentan con respecto a la carga de trabajo, es posible afirmar que el algoritmo de bubble sort no es escalable. Ante cargas crecientes de trabajo el servidor no podrá incrementar su desempeño.

Diego Lamus
Eileen Guerrero

Deployment de la solución



Nota: en el primer experimento se especificó BubbleSort como implementación de server en el server.composite; en el segundo, se especificó ShellSort.

Conclusiones

1. El algoritmo shell sort no es escalable horizontalmente.
2. El algoritmo Bubble sort no es escalable horizontalmente.
3. El algoritmo shell sort tiene mejor desempeño que el algoritmo bubble sort.