Bubble Sort

Este algoritmo funciona intercambiando repetidamente los elementos adyacentes si están en un orden incorrecto.

Ejemplo

(**5**, **1**, 4, 2, 8) → (**1**, **5**, 4, 2, 8)
(1, **5**, **4**, 2, 8) → (1, **4**, **5**, 2, 8)
(1, 4, **5**, **2**, 8) → (1, 4, **2**, **5**, 8)
(1, 4, 2, **5**, **8**) → (1, 4, 2, **5**, **8**)

(**1**, **4**, 2, 5, 8) → (**1**, **4**, 2, 5, 8)
(1, **4**, **2**, 5, 8) → (1, **2**, **4**, 5, 8)
(1, 2, **4**, **5**, 8) → (1, 2, **4**, **5**, 8)
(1, 2, 4, **5**, **8**) → (1, 2, 4, **5**, **8**)

(**1**, **2**, 4, 5, 8) → (**1**, **2**, 4, 5, 8)
(1, **2**, **4**, 5, 8) → (1, **2**, **4**, 5, 8)
(1, 2, **4**, **5**, 8) → (1, 2, **4**, **5**, 8)
(1, 2, 4, **5**, **8**) → (1, 2, 4, **5**, **8**)

Coste

Mejor: $\Omega(n)$

Peor: $O(n^2)$

Media: $\Theta(n^2)$

```
// Initial call: binarySearch(T, 0, T.size() - 1)
void bubbleSort(vector<int>& T, int l, int u) {
   // For all elements of the vector between the limits lower and upper
   for (int i = l; i <= u; i++)
      for (int j = l; j <= u - i; j++)
         if (T[j] > T[j+1])
            swap(T[j], T[j+1]);
}
```

Insertion Sort

El conjunto se divide virtualmente en una parte ordenada y otra no ordenada. Los valores de la parte no ordenada se eligen y se colocan en la posición correcta en la parte ordenada.

Ejemplo

(2, 1, 4, 3, 0) → (**1**, **2**)
(1, 2, 4, 3, 0) → (1, **2**, **4**)
(1, 2, 4, 3, 0) → (1, 2, **3**, **4**) → (1, **2**, **3**, 4)
(1, 2, 3, 4, 0) → (1, 2, 3, **0**, **4**) → (1, 2, **0**, **3**, 4) → (1, **0**, **2**, 3, 4) → (**0**, **1**, 2, 3, 4)
(0, 1, 2, 3, 4)

Coste

Mejor: $\Omega(n)$

Peor: $O(n^2)$

Media: $\Theta(n^2)$

```
// Initial call: insertionSort(T, 0, T.size() - 1);
void insertionSort(vector<int>& T, int l, int u) {
  // For all elements of the vector between the limits lower and upper
  for (int i = l; i <= u; i++) {
    int pivot = T[i]; // Set as pivot
    int j = i;
    // While the element before is higher than the actual swap it
    while (j > l and T[j-1] > pivot) {
      T[j] = T[j-1];
      --j;
    }
    T[j] = pivot; // Set the pivot to the corresponding position
  }
}
```

Selection Sort

Ordena un conjunto encontrando el elemento más pequeño y colocándolo en la posición correspondiente.

Ejemplo

(64, 25, 12, 22 , 11)
(**11**, 25, 12, 22, 64)
(11, **12**, 25, 22, 64)
(11, 12, **22**, 25, 64)
(11, 12, 22, **25**, 64)
(11, 12, 22, 25, 64)

Cost

Best: $\Omega(n^2)$

Wrost: $O(n^2)$

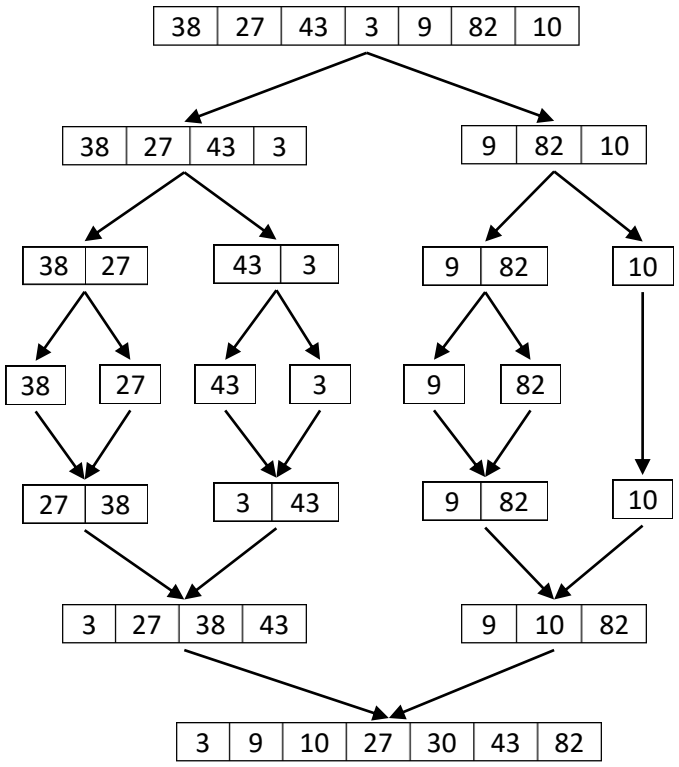
Average: $\Theta(n^2)$

```
// Initial call: selectionSort(T, 0, T.size() - 1);
void selectionSort(vector<int>& T, int l, int u) {
   // For all elements of the vector between the limits lower and upper
   for (int i = l; i <= u; ++i) {
      // Find the minimum element in unsorted array
      int min = i;
      for (int j = i+1; j <= u + 1; ++j)
         if (T[j] < T[min]) min = j;

      // Swap the found minimum element with the first element
      swap(T[min], T[i]);
   }
}
```

Merge Sort

Es un algoritmo del tipo dividir y vencer, consiste en dividir el conjunto de entrada en dos mitades, se llama a sí mismo para ir dividiéndose y fusionar las dos mitades ordenadas.

Ejemplo



Coste

Siempre: $\Theta(n \log n)$

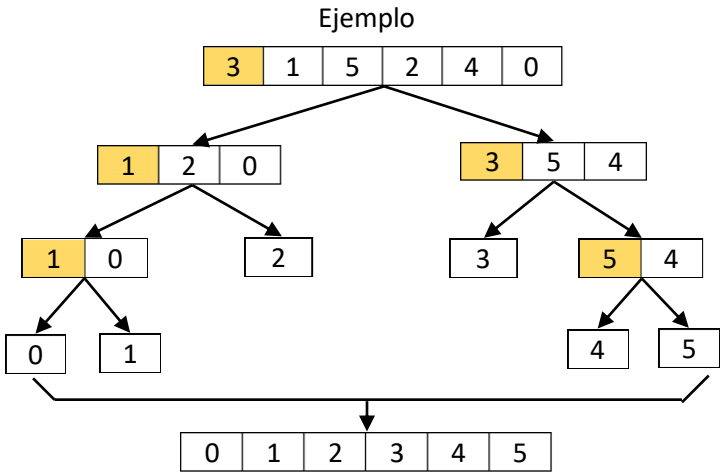
```
void merge(vector<int>& T, int l, int m, int u) {
   vector<int> B(u - l + 1);
   int i = l, j = m + 1, k = 0;
   while (i <= m and j <= u) {
      // If the element of the lower part of the secuence is smaller than the
      // upper one then add the element of the lower part to the secuence
      if (T[i] <= T[j]) B[k++] = T[i++];
      // If the element of the lower part of the secuence is greater than the
      // upper one then add the element of the upper part to the secuence
      else B[k++] = T[j++];
   }

   // If not all of the elements of the lower or upper part has been added to the
   // new secuence B then add it
   while (i <= m) B[k++] = T[i++];
   while (j <= u) B[k++] = T[j++];

   // Overwrites the part of the sequence that has been processed
   for (k = 0; k <= u - l; ++k) T[l + k] = B[k];
}

// Initial call: mergesort(T, 0, T.size() - 1);
void mergesort(vector<int>& T, int l, int u) {
   if (l < u) {
      int m = (l + u) / 2;
      // Sort the half downwards positions
      mergesort(T, l, m);
      // Sort the half upwards positions
      mergesort(T, m + 1, u);
      // Merge the two sorted sequences
      merge(T, l, m, u);
   }
}
```

Quick Sort

QuickSort es un algoritmo del tipo dividir y vencer. Este escoge un elemento como pivote y divide el conjunto dado alrededor del pivote escogido. Hay diferentes versiones, según el pivote que se elija:

1. Elegir siempre el primer o el último elemento como pivote, no muy efectivo si el conjunto está ordenado.
2. Elegir un elemento aleatorio como pivote, en media divide el problema en subproblemas parecidos, pero no siempre resulta ser el algoritmo más rápido.
3. Elegir la mediana mediana de tres elementos (porque hacer la mediana de todo sería muy costoso), hace una buena estimación, normalmente se escoge el primer, el último y el elemento del medio.

El proceso clave en quickSort es la función `partition()`. El objetivo de las particiones es, dado un conjunto y un elemento x del conjunto como pivote, poner x en su posición correcta en el array ordenado y poner todos los elementos menores (más pequeños que x) antes de x, y poner todos los elementos mayores (más grandes que x) después de x.



Coste

Mejor: $\Omega(n \log n)$

Peor: $O(n^2)$

Media: $\Theta(n \log n)$

```
int partition(vector<int>& T, int l, int u) {
   int x = T[l]; // In this case we take the first element as pivot
   int i = l - 1;
   int j = u + 1;
   while (1) {
      while (x < T[--j]);
      while (T[++i] < x);
      if (i >= j) return j;
      // If there is an element in the lower and upper part that is not in
      // the correct part, exchange them.
      swap(T[i], T[j]);
   }
   return 0;
}

// Initial call: quicksort(T, 0, T.size() - 1);
void quicksort(vector<int>& T, int l, int u) {
   if (l < u) {
      int q = partition(T, l, u);
      // Elements smaller than the pivot
      quicksort(T, l, q);
      // Elements bigger than the pivot
      quicksort(T, q + 1, u);
   }
}
```

A continuación tenemos un gráfico en el cual podemos observar cómo evolucionan los diferentes algoritmos de ordenación según el tamaño de la entrada va incrementando.