Bubble Sort

It works by repeatedly swapping the adjacent elements if they are in wrong order.

Example

(**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**)

Cost

Best: $\Omega(n)$

Worst: $O(n^2)$

Average: $\Theta(n^2)$

```
// 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

The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Example

(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)

Cost

Best: $\Omega(n)$

Worst: $O(n^2)$

Average: $\Theta(n^2)$

```
// 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

Order a set by finding the minimum element and placing it in the corresponding position.

Example

(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: $\Omega(n^2)$

Worst: $O(n^2)$

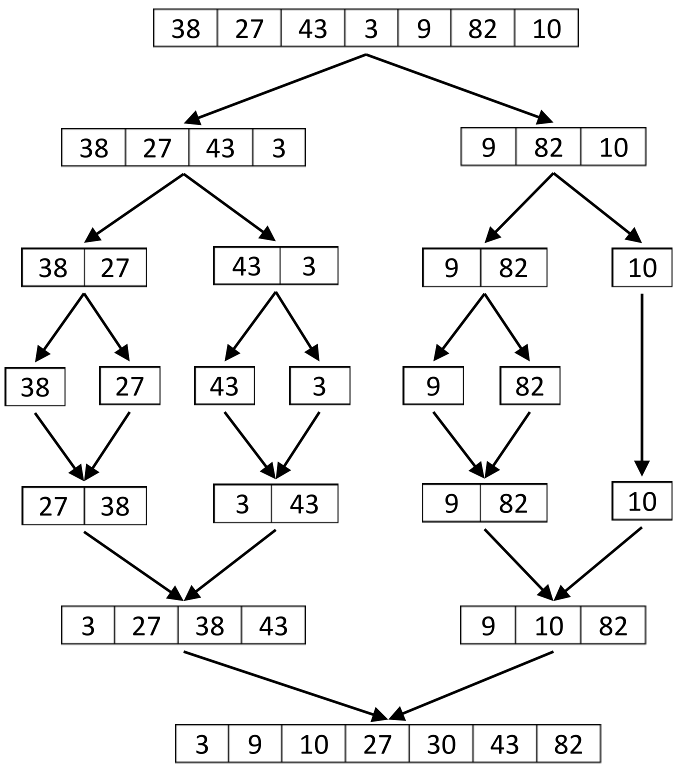
Average: $\Theta(n^2)$

```
// 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

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Example



Cost

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

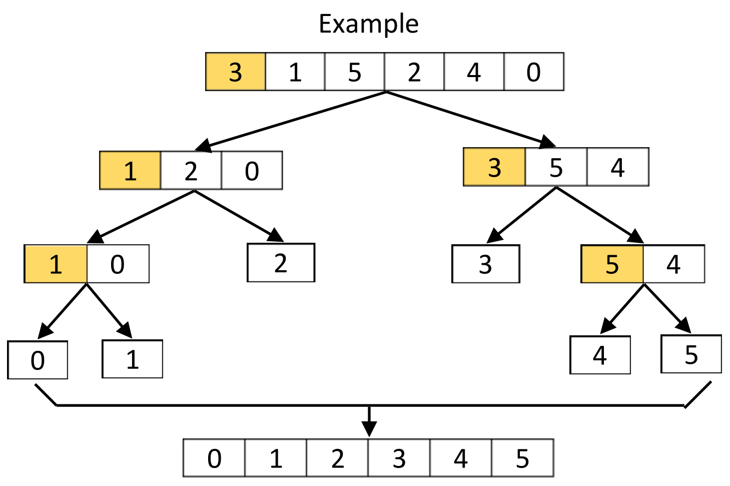
```
void merge(vector<int>& T, int l, int m, int u) {
   vector<int> B(u - l + 1);
   int i = l, j = m + 1, k = 0;
   while (i <= m and j <= u) {
      // If the element of the lower part of the secuence is smaller than the
      // upper one then add the element of the lower part to the secuence
      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 is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many versions, depending on the pivot chosen:

1. Always choose the first or last element as pivot, not very effective if the set is ordered.
2. Choosing a random element as a pivot, on average divides the problem into similar subproblems, but not always the fastest algorithm.
3. Choosing the median of three elements (because doing the median of everything would be very expensive), makes a good estimate, usually the first, the last and the middle element is chosen.

The key process in quickSort is `partition()`. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.



Cost

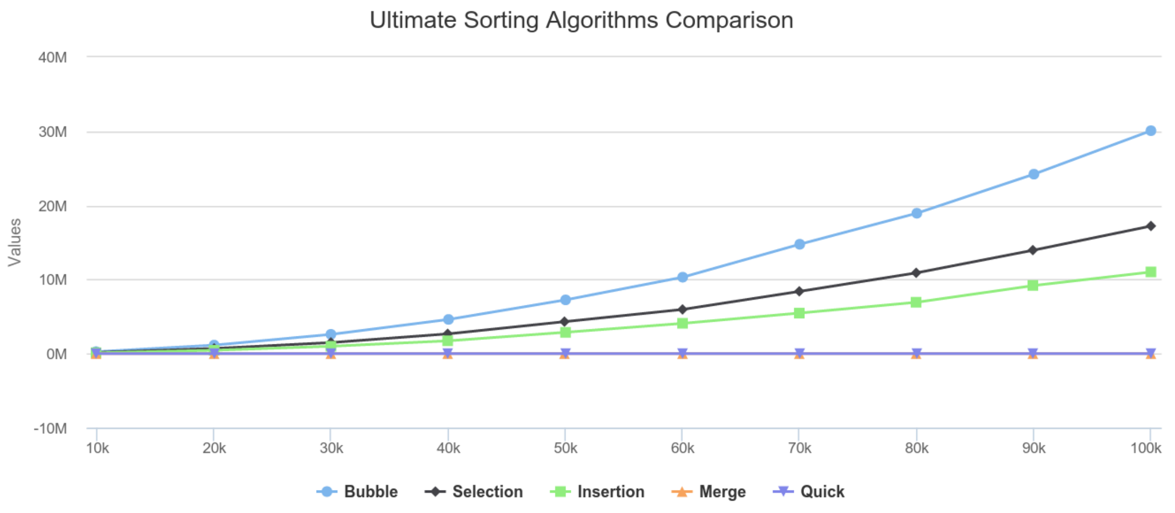
Best: $\Omega(n \log n)$

Worst: $O(n^2)$

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

```
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);
   }
}
```

Below we have a graph in which we can see how the different sorting algorithms evolve as the size of the input increases.


Ultimate Sorting Algorithms Comparison
Values
40M
30M
20M
10M
0M
-10M
10k
20k
30k
40k
50k
60k
70k
80k
90k
100k
Bubble
Selection
Insertion
Merge
Quick