Algoritmos de ordenación:

Insert
Select
Bubble

Merge
Quick
n log(n)

$l \neq$ nullptr

$l \rightarrow$ next

```
template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
    for( int p = 1; p < a.size( ); ++p )
    {
        Comparable tmp = std::move( a[ p ] );

        int j;
        for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
            a[ j ] = std::move( a[ j - 1 ] );
        a[ j ] = std::move( tmp );
    }
}
```

6 5 3 1 8 7 2 4

https://en.wikipedia.org/wiki/Insertion_sort

Selection sort

```
void selectionSort(int array[], int size) {
    for (int step = 0; step < size - 1; step++) {
        int min_idx = step;
        for (int i = step + 1; i < size; i++) {

            // To sort in descending order, change > to < in this line.
            // Select the minimum element in each loop.
            if (array[i] < array[min_idx])
                min_idx = i;
        }

        // put min at the correct position
        swap(&array[min_idx], &array[step]);
    }
}
```

Bubble sort

```
void bubbleSort(int array[], int size) {

    // loop to access each array element
    for (int step = 0; step < size; ++step) {

        // loop to compare array elements
        for (int i = 0; i < size - step; ++i) {

            // compare two adjacent elements
            // change > to < to sort in descending order
            if (array[i] > array[i + 1]) {

                // swapping elements if elements
                // are not in the intended order
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}
```

n log (n)

$$T(1) = 1$$
$$T(N) = 2T(N/2) + N$$

This is a standard recurrence relation, which can be solved several ways. We will show two methods. The first idea is to divide the recurrence relation through by N. The reason for doing this will become apparent soon. This yields

$T(N) = 2T(N/2) + N$

n log(n)

$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

This equation is valid for any N that is a power of 2, so we may also write

$n = 2^a$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

and

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$+$

$\frac{T(N)}{N} = 1 + \log(N)$

$T(N) = n + n \cdot \log(N)$

$T(n) = n \log(N)$

Merge sort

If the number of items to sort is 0 or 1, return.
Recursively sort the first and second halves separately.
Merge the two sorted halves into a sorted group.

Merge sort

```
template <typename Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size( ) );

    mergeSort( a, tmpArray, 0, a.size( ) - 1 );
}
```

```
/**
 * Internal method that makes recursive calls.
 * a is an array of Comparable items.
 * tmpArray is an array to place the merged result.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */
template <typename Comparable>
void mergeSort( vector<Comparable> & a,
                vector<Comparable> & tmpArray, int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```

(1, 2, 3), , 0,

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

* (1, 2, 3, , 2, 2

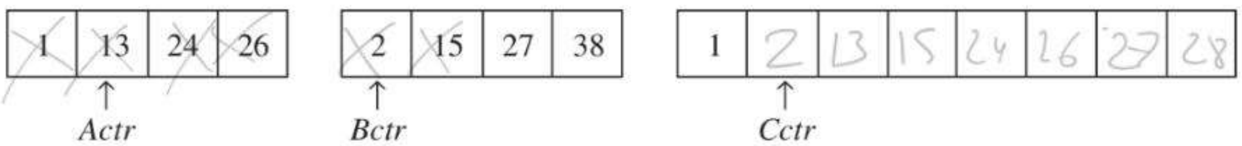
(2) * (1, 2, 3), 0

* (1, 1, 1), , 1,

```
template <typename Comparable>
void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
            int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ] <= a[ rightPos ] )
            tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );
        else
            tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );

    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );

    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );

    // Copy tmpArray back
    for( int i = 0; i < numElements; ++i, --rightEnd )
        a[ rightEnd ] = std::move( tmpArray[ rightEnd ] );
}
```

( ) ( ) ( )( )

(1, 13)(24, 26)



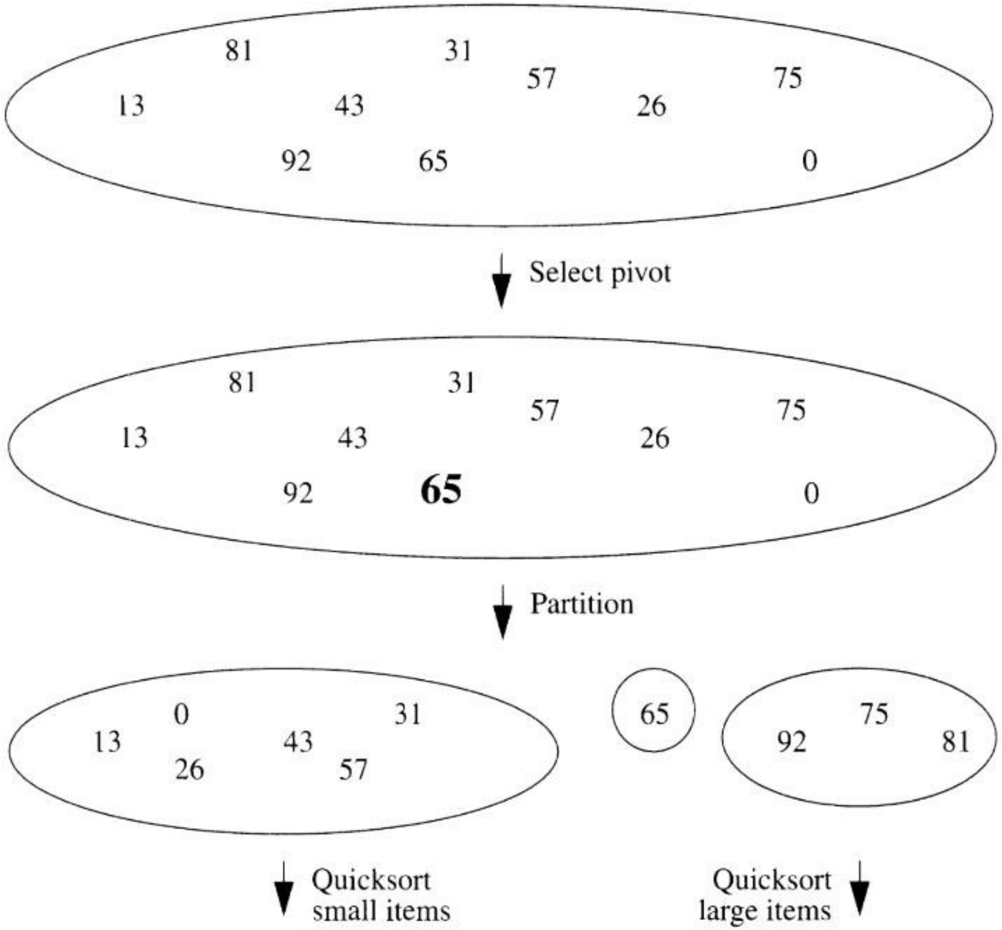
Quick sort

If the number of elements in S is 0 or 1, then return.

Pick *any* element v in S. It is called the *pivot*.

Partition $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $L = \{x \in S - \{v\} | x \le v\}$ and $R = \{x \in S - \{v\} | x \ge v\}$.

Return the result of *Quicksort*(L) followed by v followed by *Quicksort*(R).


81 31 57 75 13 43 26 92 65 0
Select pivot
81 31 57 75 13 43 26 92 65 0
Partition
0 31 13 43 26 57
65
75 92 81
Quicksort
small items
Quicksort
large items

```
// Internal quicksort method that makes recursive calls.
// Uses median-of-three partitioning and a cutoff.
template <class Comparable>
void quicksort( vector<Comparable> & a, int low, int high )
{
    if( low + CUTOFF > high )
        insertionSort( a, low, high );
    else
    {
          // Sort low, middle, high
        int middle = ( low + high ) / 2;
        if( a[ middle ] < a[ low ] )
            swap( a[ low ], a[ middle ] );
        if( a[ high ] < a[ low ] )
            swap( a[ low ], a[ high ] );
        if( a[ high ] < a[ middle ] )
            swap( a[ middle ], a[ high ] );

          // Place pivot at position high - 1
        Comparable pivot = a[ middle ];
        swap( a[ middle ], a[ high - 1 ] );
```

```
        // Begin partitioning
        int i, j;
        for( i = low, j = high - 1; ; )
        {
            while( a[ ++i ] < pivot ) { }
            while( pivot < a[ --j ] ) { }
            if( i < j )
                swap( a[ i ], a[ j ] );
            else
                break;
        }
        swap( a[ i ], a[ high - 1 ] );   // Restore pivot

        quicksort( a, low, i - 1 );      // Sort small elements
        quicksort( a, i + 1, high );     // Sort large elements
    }
}
```

1 7 4 8 15 17

1 3 9 8 4 9 3 14

1 3 4 2 3

1 3 2 4 8 9