**Tail Recursion**: working from the beginning towards the end.

```plaintext
# X       list of integers to be summed
# Start   start summing at this index . . .
# Stop    . . . and stop summing at this index
# Pre: X is a list of integers,
        Start & Stop are valid list indexes

algorithm SumArray takes list number X, number Start, number Stop

    if (Start = Stop)  # base case
        return X[Stop]
    else  # recursion
        return (X[Start] + SumArray(X, Start + 1, Stop))
    endif
```
Recursive Array Summation Trace

The invocation:
List number x
x := [37, 14, 22, 42, 19]
display SumArray( X, 1, 5)

would result in the recursive trace:

```
# return values:
SumArray(X, 1, 5)                        # 134
   return(X[1]+SumArray(X,2,5))          # 37 + 97
      return(X[2]+SumArray(X,3,5))       # 14 + 83
         return(X[3]+SumArray(X,4,5))   # 22 + 61
            return(X[4]+SumArray(X,5,5)) # 42 + 19
               return X[5]               # 19
```
**Head Recursion**: working from the end towards the front.

```plaintext
# X       list of integers to be summed
# Start   stop summing at this index . . .
# Stop    . . . and start summing at this index
# Pre: X is a list of integers,
       Start & Stop are valid list indexes

algorithm SumArray2 takes list number X, number Start, number Stop

    if (Start = Stop)             # base case
        return X[Stop]
    else                           # recursion
        return (X[Stop] + SumArray(X, Start, Stop-1))
    endif
```
Recursive Array Summation2 Trace

The invocation:

List number x

x := [37, 14, 22, 42, 19]
display SumArray2(X, 1, 5)

would result in the recursive trace:

```
SumArray2(X, 1, 5)                        # return values:  
  return(X[5]+SumArray2(X,1,4))          # 19 + 115
    return(X[4]+SumArray2(X,1,3))       # 42 + 73
      return(X[3]+SumArray2(X,1,2))    # 22 + 51
        return(X[2]+SumArray2(X,1,1)) # 14 + 37
          return X[1]                # 37
```
Middle Recursion: working from middle towards both ends.

```plaintext
# X       list of integers to be searched
# Find    integer to be located
# Start   start searching at this index . . .
# Stop    . . . and stop searching at this index
# Pre: X is an ascending ordered list of integers,
#      Find is an integer, Start & Stop are valid list indexes

algorithm BinarySearch takes list number X, number Find, number Start, number Stop
    if (Start > Stop)   # base case, value not found
        return -1
    endif

    number mid := trunc( (Start + Stop) / 2 )
    if (Find = list[mid])    # base case
        return mid
    endif
    if (Find < list[mid])    # search lower half
        return BinarySearch(X, Find, Start, mid-1)
    else                    # search upper half
        return BinarySearch(X, Find, mid+1, Stop)
    endif
```
**Edges & Center Recursion:** working from both ends towards the middle.

**Problem:**
- sort a subset, \((m:n)\), of an array of integers (ascending order)

**Solution:**
- Find the smallest and largest values in the subset of the array \((m:n)\) and swap the smallest with the \(m^{th}\) element and swap the largest with the \(n^{th}\) element, (i.e. order the edges).
- Sort the center of the array \((m+1: n-1)\)

**Solution Trace:**

<table>
<thead>
<tr>
<th>Unsorted List</th>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted List</strong></td>
<td>56</td>
<td>23</td>
</tr>
<tr>
<td><strong>After call#1</strong></td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td><strong>Variation of the “selection” sort algorithm</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>After call#3</strong></td>
<td>17</td>
<td>23</td>
</tr>
</tbody>
</table>
# ray     list of integers to be sorted  
# Start   start sorting at this index . . .  
# Stop    . . . and stop sorting at this index  
# Pre: ray is a list of integers,  
#       Start & Stop are valid list indexes

algorithm DuplexSelection takes list number ray,  
        number Start, number Stop

    if (Start < Stop)  # start=stop -> only 1 elem to sort  
        number mini := FindMinNumIndex(ray, Start, Stop)  
        number maxi := FindMaxNumIndex(ray, Start, Stop)  
        SwapEdges( ray, Start, Stop, mini, maxi)  
        DuplexSelection( ray, start+1, stop-1 )  
    endif

Alternatively, the calls to the Find functions can be replaced by a single loop through the list to locate the minimum and maximum indexes.
Recursive Sorting; SwapEdges

# ray     list of integers
# Start   left element index
# Stop    right element index
# mini    index for left swapping
# maxi    index for rightswapping
# Pre: ray is a list of integers,
#       Start, Stop mini, maxi are valid list indexes

algorithm SwapEdges takes list number ray,
      number Start, number Stop, number mini, number maxi
      
      # check for double swap interference
      if ( (mini=Stop) and (maxi=Start) )  # double interference
          Swap( ray, Start, Stop )
      else if (maxi=Start)  # low 1/2 interference
          Swap( ray, maxi, Stop )
          Swap( ray, mini, Start )
      else  # (mini=Stop) or no interference
          Swap( ray, mini, Start )
          Swap( ray, maxi, Stop )
      endif
  endif