

The QuickSort Routine

QuickSortAnArray

ArrayToBeSorted

duplicate

inputArray

sizeof(inputArray)
// Visually, this box has been "rolled-up" so that only the function name shows

size

inputArray

arraySize

```
pairingCode ( isValidSet when (inputArray.tag == arraySize.tag) );
inputTypes ( inputArray: int[], arraySize: int )
prepareForSort( inputArray, arraySize )
{
  newQSortStruc
    ut new QSortStruc ; // "ut" means transfer the association
  newQSortStruc.array
    ud inputArray ; // "ud" duplicate the assoc. (enable side-effect)
  newQSortStruc.upperBound
    uc arraySize;
  newQSortStruc.upperOuterBound
    uc newQSortStruc.upperOuterBound;
  newQSortStruc.lowerOuterBound
    ut 0;
  newQSortStruc.lowerBound
    ut 0;
  newQSortStruc.pivot
    uc newQSortStruc.array[ 0 ] ; // "uc" means copy
  newQSortStruc.lowerScan
    uc newQSortStruc.lowerBound;
  newQSortStruc.upperScan
    uc newQSortStruc.upperBound;
  return newQSortStruc to QSortStrucOut;
}
```

QSortStrucOut

BeforeRecursiveCall

inputArray

inputTypes (S: QSortStruc)

```
// Given an input structure which contains the array to be sorted, the indexes
// of the lower and upper bounds for this partial sort.
// The variables used for pivot keeping during the scan.
// One var to indicate position of empty slot after upper scan done.
// One var to indicate position of the median value (pivot pos when scan done).
// and the pivot to use in the partial sort.
// Before first part of the partial sort, upper down, then send to lower part
// If scan points hit each other then send to complete
subpartition( S )
{
  if ( S.array[ S.upperScan ] >= S.pivot ) then
  {
    S.lowerScan := S;
    if ( S.upperScan == S.lowerBound + 1 ) then
    {
      S.array[ lowerBound ] set S.pivot withOrdering
      { thisLevel := this when otherWith( other.tag.level == this.tag.level );
        higherLevel := otherWith( other.tag.level > this.tag.level );
        subID( thisLevel ) before subID( higherLevel );
      }
    }
    S.splitAt mid S.lowerBound;
    return S to subComplete; // no "with" statement so tag stays same
  }
  subID;
}
else
{
  return S to meet with tag.scan := S;
}
}
else // failed test, so move the too small element then return so lower scan can run
{
  S.array[ lowerBound ] set S.array[ upperScan ];
  S.upperSplit := mid S.upperScan;
  return S to upperDown; // tag stays the same
}
}
```

subComplete

upperDown

inputArray

```
// Given an input structure which contains the array to be sorted, and the indexes
// of the lower and upper bounds for this iteration of the scan.
// The variables used for pivot keeping during the scan.
// One var to indicate position of empty slot after upper scan done.
// One var to indicate position to split at when recursive upper pos when scan done.
// and the pivot to use in the partial sort.
// Before second part of the partial sort, lower up, then send to complete
subLower( S )
{
  if ( S.array[ S.lowerScan ] < S.pivot ) then
  {
    S.lowerScan := S;
    if ( S.lowerScan == S.upperSplit ) then
    {
      S.array[ S.upperSplit ] set S.pivot withOrdering
      { thisLevel := this when otherWith( other.tag.level == this.tag.level );
        higherLevel := otherWith( other.tag.level > this.tag.level );
        subID( thisLevel ) before subID( higherLevel );
      }
    }
    S.splitAt mid S.upperSplit; // replace with transferred value
    return S to subComplete; //tag same
  }
  subID;
}
else
{
  return S to meet with tag.scan := S;
}
}
else // failed test, so move failed to upper empty slot, leave newly tested empty
// then put S into form needed for next iteration of upper and lower scan
{
  S.array[ S.upperSplit ] set S.array[ S.lowerBound ];
  // original array bounds preserved in upperOuterBound and lowerOuterBound
  // however, for next iteration of low and upper, need to set bounds
  // and need to set scan start points
  S.lowerBound := S.lowerScan;
  S.upperBound := S.upperSplit;
  S.lowerScan := S;
  S.upperScan := S;
  return S to lowerDown; //tag same
}
}
```

subComplete

lowerDown

Q*

```
inputTypes ( S: QSortStruc )
// Given an input structure which contains the array to be sorted,
// indexes of outermost bounds for this partial sort.
// Final position of the pivot, at which to split the array.
// -- make new structure to carry recursion on lower half of array
// then recurse
// If, after splitting, the size == to base case, then return to base
splitAndRecurse( S )
{
  newLeft := new QSortStruc ; // first use, so nothing on left
  newLeft.lowerOuterBound := S.lowerOuterBound; // "uc" means copy
  newLeft.upperOuterBound := S.upperOuterBound;
  newLeft.lowerBound := newLeft.lowerOuterBound;
  newLeft.upperBound := newLeft.upperOuterBound;
  newLeft.array := S.array; // "ud" means enable side effects
  newLeft.pivot := newLeft.array[newLeft.lowerBound];
  newLeft.lowerScan := newLeft.lowerBound;
  newLeft.upperScan := newLeft.upperBound;
  // now, check whether new pivot is small enough to be base case
  if ( newLeft.upperOuterBound - newLeft.lowerOuterBound == 1 ) then
  {
    return newLeft to baseCase;
  }
  subID; // not strictly needed, just to be safe..
}
else return newLeft to recurse withTag
{
  tag := S.tag; // copy tag then modify
  tag.self.level := S;
  tag.self.type := left; // "u" means symbol
  tag.parent := S.tag.self;
}
}
```

recurse

baseCase

```
inputTypes ( S: QSortStruc )
// Given an input structure which contains the array to be sorted,
// the indexes of the outermost lower and upper bounds for this partial sort,
// the final position of the pivot, at which to split the array,
// make two new structures, one for the part below the split, one for the part above
// then recurse on them
// after splitting, if size == to base case, then return to base case
splitAndRecurse( S )
{
  newRight := new QSortStruc ;
  newRight.lowerOuterBound := S.lowerOuterBound;
  newRight.upperOuterBound := S.upperOuterBound;
  newRight.lowerBound := newRight.lowerOuterBound;
  newRight.upperBound := newRight.upperOuterBound;
  newRight.array := S.array;
  newRight.pivot := newRight.array[newRight.lowerBound];
  newRight.lowerScan := newRight.lowerBound + 1;
  newRight.upperScan := newRight.upperBound;
  if ( newRight.upperOuterBound - newRight.lowerOuterBound == 1 ) then
  {
    return newRight to baseCase;
  }
  subID;
}
return newRight to rightRecurse withTag
{
  tag := S.tag;
  tag.self.level := S;
  tag.self.type := right;
  tag.parent := S.tag.self;
}
}
```

baseCase

recurse

Q*

AfterRecursiveCall

done

new

```
inputTypes ( done, new: QSortStruc )
pairingCode
{
  parent := oneFrom (new);
  leftChild := oneFrom (done) suchThat leftChild.self.type == L;
  rightChild := oneFrom (done) suchThat rightChild.self.type == R;
  isValidSet when
  {
    parent.self == leftChild.parent && parent.self == rightChild.parent;
  }
  // Box inputs are treated as data pools. Once an input data item
  // appears in a set, it is consumed and will not appear in others
  presentSetAs ( parent, leftChild, rightChild );
  afterRecursionDo( parent, leftChild, rightChild )
  {
    if parent.type == root then
    {
      return parent.array to finished;
    }
    else
    {
      return parent to newlyDone;
    }
  }
}
```

newlyDone

finished