**MergeSort**

**Mergesort Algorithm**

|  |
| --- |
| Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted)  Repeatedly [merge](http://en.wikipedia.org/wiki/Merge_algorithm) sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list. |

**MergeSort Efficiency**

|  |
| --- |
| The mergesort time is **O(N\*logN)**.  **Disadvantage**: the requirement of an additional array in memory, equal in size to the one being sorted. |

**The mergeSort Method**

|  |
| --- |
| **public** **void** mergeSort() // called by main()  { // provides workspace  **long**[] workSpace = **new** **long**[nElems];  **recMergeSort**(workSpace, 0, nElems-1);  } |

**The recMergeSort Method**

|  |
| --- |
| **private** **void** **recMergeSort**(**long**[] workSpace, **int** lowerBound, **int** upperBound)  {  **if**(lowerBound == upperBound) // if range is 1,  **return**; // no use sorting  **else** // find midpoint  **int** mid = (lowerBound+upperBound) / 2; // sort low half  **recMergeSort**(workSpace, lowerBound, mid); // sort high half  **recMergeSort**(workSpace, mid+1, upperBound); // merge them  merge(workSpace, lowerBound, mid+1, upperBound);  }  } |

**The merge Method**

|  |
| --- |
| **private** **void** **merge**(**long**[] workSpace, **int** lowPtr, **int** highPtr, **int** upperBound)  {  **int** j = 0; // workspace index  **int** lowerBound = lowPtr;  **int** mid = highPtr-1;  **int** n = upperBound-lowerBound+1; // # of items  **while**(lowPtr <= mid && highPtr <= upperBound)  **if**( theArray[lowPtr] < theArray[highPtr] )  workSpace[j++] = theArray[lowPtr++];  **else**  workSpace[j++] = theArray[highPtr++];  **while**(lowPtr <= mid)  workSpace[j++] = theArray[lowPtr++];  **while**(highPtr <= upperBound)  workSpace[j++] = theArray[highPtr++];  **for**(j=0; j<n; j++)  theArray[lowerBound+j] = workSpace[j];  } |