Merge Sort

- Uses divide-and –conquer methodology.

- Main idea: Divide the array into roughly equal sized arrays and sort them separately (it is much easier to sort short lists than long ones). Then merge the two sorted arrays in order to get one sorted array.

- Trace with:

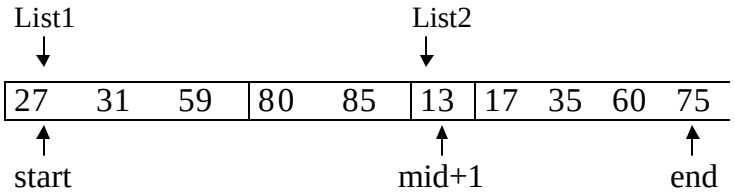
59	27 80 35 13	75

In C:

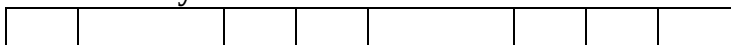
```
void MergeSort(int List[], int start, int end)
{
    int mid;

    if (start < end) {
        mid = (start+end)/2;
        MergeSort(List, start, mid);
        MergeSort(List, mid+1, end);
        Merge(List, start, mid+1, end);
    }
}
```

Merging Two Sorted Lists

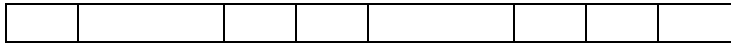
Illustration of merging two sorted sublists in one array:



Local Array



Sorted Array




```
void Merge(int List[], int start1,int start2,
           int end2)
{
    int hold[maxN];
    int index, length, count1, count2, total;

    length = start2 – start1;

    /* copy values in first half into local array */
    for (index = 0; index < length; index++)
        hold[index] = List[start1 + index];

    /* Counters keep track of the current elements
     * in each of the two sublists to merge
     */
    count1 = 0;
    count2 = start2;
    total = start1;

    /* Loop until all values in one of the sublists
     * have been merged into the sorted part.
     */
    while(count1 < length && count2 <= end2){
        if (hold[count1] < List[count2]){
            List[total] = hold[count1];
            count1 = count1 + 1;
        }
        else {
            List[total] = List[count2];
            count2 = count2 + 1;
        }
        total = total + 1;
    }
    while (count1 < length){
        List[total] = hold[count1];
        count1 = count1 + 1;
        total = total + 1;
    }
    return;
}
```

Class Exercise

```
void MergeSort(int List[], int start, int end)
{   int mid;
    if (start < end) {
        mid = (start+end)/2;
        MergeSort(List, start, mid);
        MergeSort(List, mid+1, end);
        Merge(List, start, mid+1, end);
    }
}
```

Suppose we are going to sort the following array of numbers using mergesort algorithm.

60	12	90	30	64	8	6

a) How many recursive calls to the MergeSort function are made to sort this array? (Do not count the original call.)

b) How many calls to the Merge function are made in total?

Quick-Sort

- Like merge sort, Quicksort is also based on the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works by partitioning an array into two parts, then sorting the parts independently, and finally combining the sorted subsequences by a simple concatenation.

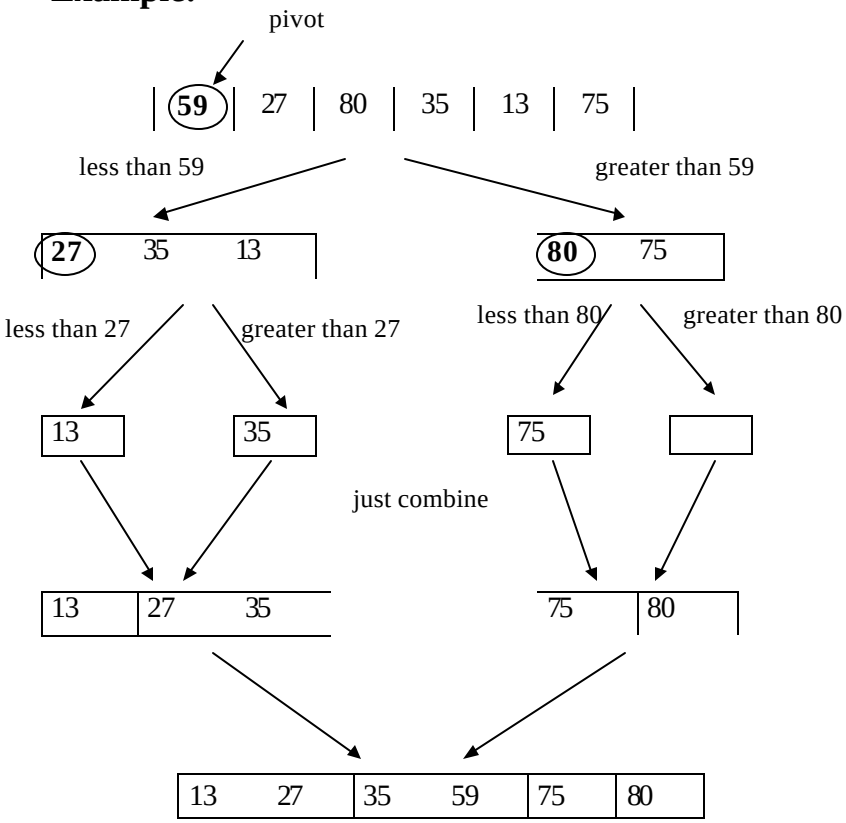
In particular, the quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.

2. ***Recur***: Recursively sort the sublists separately.

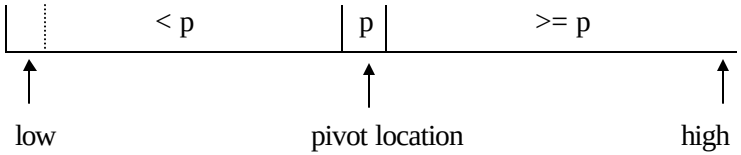
3. ***Conquer***: Put the sorted sublists together.

Example:

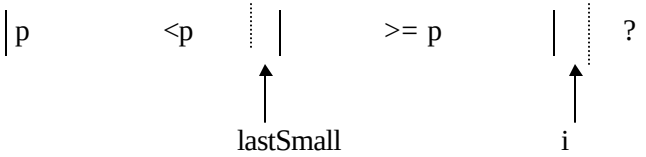


Partitioning the List

Goal:



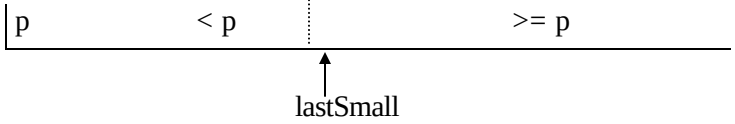
In the middle of the loop:



At each iteration check the value in entry i:

- if value in entry i >= p then just increment i.
- if value in entry i < p then increment lastSmall and swap the contents of entry lastSmall with entry i.

At the end of the loop:



Swap pivot with entry lastSmall

Implementation

```
/* If the array has one or fewer elements
 * do nothing. Otherwise the array is processed
 * by a partition function which puts L[pivotLoc]
 * into position and rearranges the
 * other elements such that recursive calls
 * properly finish the sort
*/

void quickSort(int L[], int low, int high)
{
   int pivotLoc;

   if (low < high){
      pivotLoc = partition(L, low, high);
      quickSort(L, low, pivotLoc – 1);
      quickSort(L, pivotLoc + 1, high);
   }

}
```



```
/*
 * Center element is chosen as pivot
 * /

int partition(int L[], int low, int high)
{
   int pivot;
   int i, lastSmall;

   swap(L,low,(low+high)/2);
   pivot = L[low];
   lastSmall = low;

   for (i = low+1; i<=high; i++){
      if (L[i] < pivot) {
         lastSmall = lastSmall + 1;
         swap(L, lastSmall, i);
      }
   }

   swap(L, low, lastSmall);
   pivotLoc = lastSmall;
   return pivotLoc;
}

void swap(int L[], int i, int j)
{
   int temp;
   temp = L[i];
   L[i] = L[j];
   L[j] = temp;
}
```