9 - merge sort

In a way quicksort pushes 'sortedness' downwards, each time it splits the array of entries it divides them into a lower and upper group so that when the process reaches the bottom, with one or two element sub-arrays, the whole array is sorted. Merge sort, sometimes called mergesort, works the other way, sorting the subarrays as it merges them together. It was invented in 1945 by John von Neumann, a towering figure in mid-C20 computer science and mathematics. It works because merging two sorted arrays into another sorted array is an $O(n)$ operation.

We will examine the merge operation first; this involves two ordered arrays, to be merged, and a extra array for putting the merged elements. There is a marker for each of the order arrays, say i and j, these both start at the low end of there arrays and the elements at i and at j are compared, the lower is added to the merged arrays and the corresponding marker is moved forward one. This keeps going until one of the markers reaches the end of its list, when that happens all the remaining elements of the other array are added to the merged array. Code to do this is given in Table 1; in this code the two sorted arrays that need to be merged are consequitive subarrays marked by `first` to `mid-1` and `mid` to `last`.

Now that that is done, it is easy to do merge sort, basically at each stage the array is split in two and merge sort is called on each half, when these return they are merged using the merge function. The recursion terminates on a single element list. A function for doing this is given in Table 2 and a rough example is given in Table 4; a Haskell version is given in Table 3.

Merge sort does the same thing no matter what order the entries are in to start with. The run times can be calculated from the recursion, ignoring the small effect from n not always begin even

$$T(n) = cn + 2T(n/2) \tag{1}$$

where the cn corresponds to the merge. This is the same recursion as for the quicksort, so merge sort is $O(n \log n)$ with the difference here that this is the run time no matter what. The algorithm, as implemented here, needs another array the same length as the original one, for doing the merges into; thus, the extra memory use is $O(n)$. This can be beaten, there are implementations with smaller memory use, even implementations that run in place, but they are more complicated. Parallel processing works well with merge sort and there are efficient sort algorithms based on a mixture of merge sort and insert sort.

```
1  void merge(int a[], int merged_a[], int first, int mid, int last)
2  {
3      int last_lower=mid-1,i=first,j=mid,merged_i=first;
4
5      while(i<=last_lower&&j<=last)
6          if(a[i]<a[j]){
7              merged_a[merged_i]=a[i];
8              i++;
9              merged_i++;
10         }
11         else{
12             merged_a[merged_i]=a[j];
13             j++;
14             merged_i++;
15         }
16
17     while(i<=last_lower){
18         merged_a[merged_i]=a[i];
19         i++;
20         merged_i++;
21     }
22
23     while(j<=last){
24         merged_a[merged_i]=a[j];
25         j++;
26         merged_i++;
27     }
28
29     for(i=first;i<=last;i++)
30         a[i]=merged_a[i];
31 }
```

Table 1: Merging. This merges the elements from first to mid-1 and mid to last under the assumption that they are already sorted, to give a merged array from first to last of merged_a, these elements are then copied back to elements first to last of a. This function is part of the full merge sort program `merge_sort.c`.

```
1  void merge_sort_r(int a[], int merged_a[],int first , int last)
2  {
3     if(last-first<=0)
4       return;
5
6     int mid=(last+first)/2+1;
7
8     merge_sort_r(a, merged_a,first ,mid-1);
9     merge_sort_r(a, merged_a,mid,last);
10
11    merge(a, merged_a, first , mid, last);
12
13 }
```

Table 2: Merge sort. This splits the array, calls itself recursively on the two parts and then merges them. It can be found as part of `merge_sort.c`, this also includes the wrapper and so on.

```
1  mergesort :: Ord a => [a] -> [a]
2  mergesort []  = []
3  mergesort [x] = [x]
4  mergesort xs  = merge (mergesort ys) (mergesort zs)
5    where
6      (ys, zs) = splitAt (length xs `div` 2) xs
7
8      merge []     ys     = ys
9      merge xs     []     = xs
10     merge (x:xs) (y:ys)
11       | x <= y     = x : merge xs (y:ys)
12       | otherwise  = y : merge (x:xs) ys
```

Table 3: Merge sort: Haskell example code from Nick W.

8	5	3	4	2	6	1	7
8	5	3	4 |	2	6	1	7
8	5 |	3	4 |	2	6 |	1	7
8 |	5 |	3 |	4 |	2 |	6 |	1 |	7
5	8 |	3	4 |	2	6 |	1	7
3	4	5	8 |	1	2	6	7
1	2	3	4	5	6	7	8

Table 4: Merge sort in action. This represents the process of merge sorting, the vertical line represents the different subarrays. This table is very schematic, it doesn't really show the order things happen, and it shows things happening to different subarrays simulataneously, when, they don't actually happen like that.