4 Mergesort

4.1 Efficient sorting methods

First, it is customary to refer items being sought or sorted as *sort keys.* Sorting is probably the most important technique in computing. Among the popular sorting methods are

- Quicksort. Average runtime is low, but it can perform badly sometimes. Also, it needs pointer structures for splitting and concatenating lists.
- Heapsort. Provably efficient in all cases. Very clever use of array structures. A bit clunky, and *not stable* (see next).
- A stable sorting method is one which, in the presence of equal keys, preserves the initial order of equal keys.
- Binsort or lexical sort. The items to sort contain a few 'digits' on which the data is sorted. Very useful for graph algorithms, it needs pointer structures for splitting and concatenating lists.
- Mergesort: efficient and works quite naturally on arrays.

4.2 Mergesort

This is a provably efficient sorting method by repeated merging. The idea is simple. To sort an array, for block-lengths 1,2,4,... ensure that it is divided into sorted blocks of the given length and arrange it into sorted blocks of double the length by merging.

```
#include <stdio.h>
typedef int SORTABLE;

int threeway ( SORTABLE x, SORTABLE y )
{
  return x-y;
}

void merge ( int m, SORTABLE x[], int n, SORTABLE y[], SORTABLE z[] )

        // merge x[0..m-1] with y[0..n-1] to z[0..m+n-1]
        // Important: if m is zero, the routine does not
        // read any of x; likewise, n and y.

{ int i=0, j=0, k=0;
  while ( i < m || j < n )
  {
    if ( i >= m )
    { z[k] = y[j]; ++j; }
```

```
    else if ( j >= n )
    { z[k] = x[i]; ++i; }
    else if ( threeway ( x[i], y[j] ) > 0 )
    { z[k] = y[j]; ++j; }
    else
    { z[k] = x[i]; ++i; }

    ++k;
  }
}
```

4.3 Runtime of merge

Before proceeding, let us analyse the runtime of this procedure. (Its correctness is 'obvious' and it would be difficult to prove.) For the runtime, count the iterations; in each iteration either i or j increases but not both, they begin at zero and end at m and n respectively. There are $m + n$ iterations.

In simple terms the runtime is bounded by $a(m + n) + b$ where a and b are constants depending on the processor being used.

4.4 Mergesort, continued

```
void mergesort ( int n, SORTABLE a[] )
{
  SORTABLE b[n];
  SORTABLE *aa, *bb;
  int length = 1;

  for (int i=0; i<n; ++i)
    b[i] = 99; // make sure b has no garbage data

  aa = a; bb = b; // b is extra storage

  /* This is rather obscure, called 2-column mergesort.
     It goes through several 'passes' over the arrays,
     and exploiting the C pointer conventions we can
     swap the arrays at negligible cost between passes so that
     aa and bb are a and b, b and a, alternately.

     At the beginning of a pass, the array aa contains
     the items in a, but every block of (length) items
     is sorted.  At the end, the array bb contains the
     same items as a, but every block of (2*length) items
     is sorted.  The length is doubled at every pass.
  */
```

```
  int mm, nn;
  while ( length < n )
  {
    for ( int i = 0; i < n; i += 2 * length )
    {
      if ( i + 2*length <= n )
      { mm = nn = length;
        merge (mm, aa+i, nn, aa + i+mm, bb+i);
      }
      else if ( i + length <= n )
      {
        mm = length;
        nn = n - i - length;
        merge (mm, aa+i, nn, aa + i+mm, bb+i);
      }
      else
      {
        nn = 0;
        mm = n - i;
        merge (mm, aa+i, nn, aa+i, bb+i);
      }
    }
    SORTABLE * t = aa; aa = bb; bb = t;
    length *= 2;
  }

  if ( aa != a )
        // make sure a has the sorted result
        // by copying aa (which is b) to a.
    for ( int i = 0; i<n; ++i )
      a[i] = aa[i];
}

int main()
{ int a[9] = {3,1,4,1,5,9,2,6,5};
  int n = 9;

  printf("before sorting\n");
  for (int i=0; i<n; ++i) printf(" %d", a[i]);
  printf("\n");
  mergesort (n,a);
  printf("after sorting\n");
  for (int i=0; i<n; ++i) printf(" %d", a[i]);
  printf("\n");
```

```
  return 0;
}

prompt% gcc -std=gnu99 mergesort.c
prompt% a.out
before sorting
 3 1 4 1 5 9 2 6 5
after sorting
 1 1 2 3 4 5 5 6 9
```

4.5 Correctness

At any point in the computation, the array `aa` consists of sorted blocks of given length. Ultimately, `length>=n`.

4.6 Efficiency

Let $\ell = 1, 2, 4, \ldots$ be the values taken by the variable `length`. Starting the r-th pass, `aa` is in sorted blocks of length $\ell = 2^{r-1}$, and after the r-th pass, `bb` is in blocks of length 2^r.

When $2^r \geq n$ the array is sorted. Therefore there are $\leq \lceil \log_2(n+1) \rceil$ passes. (The $n+1$ avoids arguments about $\log_2 0$.)

The rounding up can be covered by adding 1. So there are at most

$$\log_2(n+1) + 1$$

passes.

In the r-th pass, blocks of length 2^r are formed by merging. Let $S = \lceil n/2^r \rceil$. There are S calls to merge, each one costing $ak + b$, where k is the number of merged items. Except for the last, $k = 2^r$. The total cost for the r-th pass is at most

$$an + bS$$

Crudely put, since $S \leq n$, the total cost of the pass is $\leq (a+b)n$. The overall cost of mergesort is

$$\leq (a+b)n(\log(n+1) + 1)$$

which is $O(n \log n)$. ∎