8 Sorting - bubble sort and quicksort

We have already looked at one sorting algorithm, insert sort and found that it was $O(n^2)$. We also saw that it has only a $O(1)$ extra memory requirement, that is, not including the memory to store the array itself, it only required a fixed amount of memory to run. Here we are going to look at other sorting algorithms, both because sorting algorithms are important and useful and because they serve as a good set of example algorithms for considering different aspects of efficiency.

Bubble sort

In bubble sort the list is sorted by going through it element by element and swapping each element with the one next to it if they are in the wrong order. This is repeated until the list is sorted. Thus, the small numbers 'rise to the top', that is end up near the start of the list, because they are swapped repeatedly with larger numbers that start off above above them. This terminology can get confusing since it can be hard to remember what you mean by 'top' and 'bottom', but the analogy is useful, the elements move up or down the list with each successive run through. Table 1 gives an example of bubble sort in action and code for bubble sort is given in Table 2 and Table **??**.

Bubble sort is $O(n^2)$, for the worst case where the list starts off in reverse order it takes n passes through the list to order it, with each pass involving $n-1$ comparisons. This is because each pass basically moves the largest unsorted element to its correct position. Bubble sort is $O(1)$ in extra memory. However, although it appears in these characteristics to match insert sort, it is regarded as being inferior because, in practice, it is slower, particularly for lists that are close to being sorted. There are improvements, like cocktail sort, which alternates sorting big elements up and small ones down, and comb sort which initially moves elements by more than one step at a time. Really though, the appeal of bubble sort is its simplicity and an intuitive appeal, since it is worse than insert sort, in most cases, it isn't actually useful.

We have been assuming that the group is divided roughly in two at each step. In fact, this depends on a good choice of the partition value. If the partition value is chosen to be the lowest value in the list, the pile of entries with value lower than the partition value will be empty. If we have adapted the strategy of excluding the partition value from both piles, then this run through will have reduced the number of elements by one. If this were to happen every time the run time for quicksort would be $O(n^2)$, as bad as insertion sort since each run through involves n comparisons and reduces the number of elements by only one. In other words, the worst case run time for quicksort is $O(n^2)$. If a median of threes approach is used the partition value can't be the lowest value, but it could be the second lowest and if the second lowest value is chosen each time, the algorithm is still $O(n^2)$. However, this is hugely unlikely, for most initial data the chance of getting $O(n^2)$ behavior is extremely small. An exception might be, for example, a set where there are lots of equal small entries with a few large entries mixed in.

Quicksort is probably the most used sorting algorithm. Although its worst case behavior is $O(n^2)$ it usually runs at $O(n \log n)$ and, in most applications, it usually runs faster than other $O(n \log n)$ algorithms. Furthermore, it is done in place, it doesn't require substantial amounts of extra memory.

0	5	6	2	3	4	1
1	5	6	2	3	4	1
2	5	2	6	3	1	4
3	5	2	3	6	1	4
4	5	2	3	1	6	4
5	5	2	3	1	4	6
6	2	5	3	1	4	6
7	2	3	5	1	4	6
8	2	3	1	5	4	6
9	2	3	1	4	5	6
10	2	3	1	4	5	6
11	2	1	3	4	5	6
12	2	1	3	4	5	6
13	1	2	3	4	5	6
14	1	2	3	4	5	6
15	1	2	3	4	5	6

Table 1: Bubble sort. Line zero is the original list and the horizontal lines seperate different runs. Each line represents a comparison and, when a swap is appropriate, a swap. It avoids comparisons at the end where the elements are already sorted..

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


void bubble(int a[], int n)
{
    int i,unfinished=1;

    while(unfinished){
        unfinished=0;
        for(i=0;i<n-1;i++)
            if(a[i]>a[i+1]){
                unfinished=1;
                swap(a,i,i+1);
            }
    }
}
```

Table 2: A bubble sort. The int unfinished is used to stop the while loop, at the start of each while loop it is set to zero, if any pairs need to be swapped it is changed to one; zero counts as false when evaluated as a boolean value, anything else casts to true. This pair of functions is part of `bubble_sort.c`. One easy way to improve this version is to avoid the unneeded comparisons with the elements that have already been sorted, this improved version is also $O(n^2)$ but is quicker, it can be seen at `bubble_sort_better.c`.

```
bubblesort :: Ord a => [a] -> [a]
bubblesort xs
  | ys == xs   = ys
  | otherwise = bubblesort ys
  where ys = bubble xs

bubble :: Ord a => [a] -> [a]
bubble (x:y:zs)
  | x > y      = y : bubble (x:zs)
  | otherwise = x : bubble (y:zs)
bubble xs = xs
\caption{A Haskell bubble sort from Nick W. \label{haskell_bubble}}
\end{table}

\subsection*{Quicksort}

Quicksort, usually written like that, as one word, is probably the
most popular sorting algorithm. It was discovered in 1960 by Tony
Hoare. It works by choosing an element, possibly at random, called the
partition value or pivot, let's␣say␣it␣has␣value␣$p$.␣The␣list␣is␣then
split␣into␣two␣piles,␣elements␣less␣than␣$p$␣and␣elements␣greater
than␣$p$.␣This␣is␣then␣repeated␣recursively␣on␣each␣of
the␣two␣piles␣with␣the␣recursion␣terminating␣if␣a␣pile␣has␣only␣one
element.␣Crucially,␣with␣a␣bit␣of␣care,␣the␣process␣of␣splitting␣the
list␣into␣two␣piles␣can␣be␣done␣in␣place,␣so␣no␣substantial␣extra
memory␣is␣needed.␣In␣reading␣the␣algorithm,␣most␣of␣the␣complexity
comes␣from␣the␣desire␣to␣sort␣in␣place,␣so␣keep␣in␣mind␣the␣central
idea,␣choose␣an␣element␣and␣make␣two␣piles,␣one␣for␣elements␣smaller
than␣the␣chosen␣value,␣one␣for␣elements␣larger,␣and␣then␣recurse␣onto
those␣two␣piles.

\begin{table}
\begin{tabular}{c|cccccc}
0&\underline{5}&6&2&3&4&\underline{1}\\
1&1&\underline{6}&2&3&\underline{4}&5\\
2&1&\underline{6}&2&\underline{3}&4&5\\
3&1&\underline{6}&\underline{2}&3&4&5\\
4&1&2&6&3&4&5\\
\end{tabular}
\caption{Dividing␣the␣piles␣in␣quicksort.␣Here␣the␣partition␣value␣is
␣␣3␣and␣the␣underlines␣mark␣the␣position␣of␣$i$␣on␣the␣left␣and␣$j$␣on
␣␣the␣right.␣In␣line␣0␣the␣$i$␣and␣$j$␣are␣at␣the␣start␣and␣the␣end,
␣␣since␣$5\ge␣3$␣and␣$1<3$␣these␣values␣are␣swapped␣and␣$i$␣and␣$j$
␣␣moved.␣6␣needs␣to␣be␣moved,␣but␣$4\ge␣3$␣so␣it␣can't be, instead $j$
  is moved, next $3\ge 3$ so $j$ is decreased again, now, in line 3,
  $6\ge 3$ and $2<3$ so these entries are swapped. Increasing $i$ and
  decreasing $j$ would leave them in the wrong order, so the procedure
  is finished. The value of $i$ and $j$ in line 3 mark out the
  division between the low pile and the high
  pile.\label{table_quick_divide}}
\end{table}

So, let's␣examine␣quicksort␣in␣practise␣and␣ignore␣for␣now␣how␣the
partition␣value␣is␣chosen␣$p$.␣A␣marker␣$i$␣is␣placed␣at␣the␣first
entry␣of␣the␣list,␣and␣another␣$j$␣at␣the␣last␣entry.␣If␣$a[i]<p$␣and
$a[j]\ge␣p$␣then␣both␣$a[i]$␣and␣$a[j]$␣are␣in␣the␣correct␣pile␣and
nothing needs to be done. In this case $i$ is increased by one and $j$
```

```
void quick_r(int a[],int first, int last)
{
  if(first>=last)
    return;
  if(last==first+1)
    if(a[first]<a[last])
      return;
    else{
     swap(a,first,last);
     return;
    }

  int i=first,j=last-1;

  swap(a,median(a,first,first+1,last),last);

  while(i<j){
    while(a[j]>=a[last]&&j>first)
      j--;
    while(a[i]<a[last])
      i++;
    if(i<j)
      swap(a,i,j);
  }
  swap(a,last,i);

  quick_r(a,first,i-1);
  quick_r(a,i+1,last);
}

void quick(int a[], int n)
{
  quick_r(a,0,n-1);
}
```

Table 4: Quicksort. This is the business part of the quicksort algorithm, see Table 4 for some of the functions used. Notice how j is decreased first and is made to stop if it reaches first, i is then increased and stops if a[i]<a[last], this means that, at the end, when i≥j, i gives the first entry of the upper pile. Since the partition value has been placed for safe keeping at the end of the upper pile, it can be swapped for this value. This is illustrated in Table 7.

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
    where ys = [ y | y <- xs , y <= x ]
            zs = [ z | z <- xs , x <  z ]
```

Table 5: Quicksort. This is a Haskell programme from Nick W implementing quicksort, it always uses the first element as the pivot.

0	6	3	2	5	4	1
1	6	1	2	5	4	3
2	6_i	1	2	5	4_j	3
3	6_i	1	2	5_j	4	3
4	6_i	1	2_j	5	4	3
5	2_i	1	6_j	5	4	3
6	2_i	1_j	6	5	4	3
7	2	1_j	6_i	5	4	3
8	2	1	3	5	4	6

Table 6: The division method for our implementation of quicksort. This shows the location of `i` and `j` as quicksort is running with a partition value of 3. In line 7 `i>j` so the algorithm stops and in line 8 the partition value is swapped back to the place marked by `i`.