7 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.

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 practise, 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.

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, lets say it has value p. The list is then split into two piles, elements less than p and elements greater than, or equal to, 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.

So, lets 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 $a[j] \geq p$ 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 is decreased. If $a[i] \geq p$ and $a[j] < p$ then both are in the wrong pile and they are swapped, i is then increased and j decreased. If $a[i] \geq p$ but $a[j] \geq p$ as well, then $a[i]$ needs to be swapped, but can't be because $a[j]$ needs to stay put. In this case j is decreased by one; in the converse case where $a[i]$ and $a[j]$ are both less than p, i is increased by one. This process of swapping finishes when $i \geq j$. In practise that means i is increased until $a[i] \geq p$ and j is decreased until $a[j] < p$, then, if $i < j$ the two entries are swapped. This part of quicksort is illustrated in Table 3.

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..

The algorithm is then applied recursively to each of the pile. If a pile has only one entry it is already sorted, if it has two it can be sorted with a single comparison and, if needed, a single swap. We still haven't dealt with the choice of the partition point. Obviously the best thing is to chose a value that splits the pile more-or-less in half each time; if you are very unlucky and choose the lowest value then the lower pile would be empty. The normal approach is called median of threes, three entries are chosen and the middle one is used as the partition point. This means that the lower pile will have at least one element so the algorithm is guaranteed to converge and, by looking at three it makes it likely that it will give a reasonably even split.

Before finally looking at the code for quicksort, we need to think a little bit about being careful about keeping track of where the split between the two piles. Furthermore, it is worthwhile not wasting what we know about the partition value: its value lies between the two piles. Thus, at the end we can swap it to the point where the two piles join and then not include it in either since it is already in the right place. A collection of functions for performing quicksort are given in Table 4 and Table 5. Table 6 illustrates the part of the precise algorithm that divides the data into two piles that is used in the code.

The run time for this algorithm needs to be worked out recursively. If there are n entries $T(n)$ requires the approximately n comparisons and swaps needed to partition the algorithm. It also calls quicksort on the two piles afterwards. If these piles are roughly equal then

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

where a more exact version might look something like $T(n) = c_1 + c_2 n + T(n_1) + T(n_2)$ where $n_1 + n_2 = n$, but the approximate version is good enough to find the algorithmic complexity provided we assume $n_1 \approx n_2 \approx n/2$.

We will have a look at how we might solve this recursion relation, but we are in the happy position of not needing to do this, we can just apply the master theorem. The master theorem

```
1  void swap(int a[],int i, int j)
2  {
3      int temp=a[i];
4      a[i]=a[j];
5      a[j]=temp;
6  }
7
8
9  void bubble(int a[], int n)
10 {
11     int i,unfinished=1;
12
13     while(unfinished){
14         unfinished=0;
15         for(i=0;i<n-1;i++)
16             if(a[i]>a[i+1]){
17                 unfinished=1;
18                 swap(a,i,i+1);
19             }
20     }
21 }
```

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`.

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

Table 3: 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 $5 \geq 3$ and $1 < 3$ these values are swapped and i and j moved. 6 needs to be moved, but $4 \geq 3$ so it can't be, instead j is moved, next $3 \geq 3$ so j is decreased again, now, in line 3, $6 \geq 3$ 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.

is used to calculate the algorithmic behavior of solutions to the recursion

$$T(n) \approx f(n) + aT(n/b) \tag{2}$$

which is our situation with $a = b = 2$ and $f(n) = cn$. This means $f(n) \in \Theta(n)$ which is of the form $\Theta(n^c)$ with $c = 1$, now we also have $\log_b a = \log_2 2 = 1$ so this is the marginal case where $T(n) \in \Theta(n^c \log n)$, with $c = 1$ this means $T(n) \in \Theta(n \log n)$, with the warning that this applies to the $T(n)$ that satisfies the recursion relation, which isn't always all the run times, in particular, here, the run time only satisfies the recursion is the pivot divides the piles in half.

Since we were able to use the master theorem there is really no need to think further about how to solve the recursion relation. We will do so anyway for nosing around reasons. Now to solve the recursion relation consider

$$T(n) = An \log n \tag{3}$$

so

$$2T(n/2) = 2A\frac{n}{2} \log \frac{n}{2} = An \log n - An \tag{4}$$

and so this solves the approximate equation when $A = c$. In fact by substitution you can see that

$$T(n) = An \log n + Cn \tag{5}$$

solves the equation for any value of C, if we were solving this equation exactly then the initial value would be used to fix C; here though we are only interested in the large n behavior, so this isn't important.

Of course, the problem here is that we solved the equation by roughly knowing the answer and checking our guess was correct. You can see that you might arrive at this guess by staring at the equation and trying lots of things. Another, more straight forward approach, would be to do a few steps of the iteration

$$\begin{aligned} T(n) &= cn + 2T(n/2) && (6)\\ &= cn + 2c(n/2) + 4T(n/4) && (7)\\ &= 2cn + 4c(n/4) + 8T(n/8) && (8)\\ &= 3cn + 8c(n/8) + 16T(n/16) && (9)\\ &= \dots && (10)\\ &= rcn + 2^r T(n/2^r) && (11) \end{aligned}$$

which carries on until $n/2^r = 1$, or $r = \log n$. Substituting that into the equation we get

$$T(n) = cn \log n + 2^{\log n} T(1) = cn \log n + nT(1) \tag{12}$$

or $T(n) \in O(n \log n)$ as before.

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 $O(n^2)$, as bad as insertion sort since each run through

```
1  int median(int a[],int i, int j, int k)
2  {
3
4    if(a[i]>a[j]&&a[i]>a[k])
5      return (a[j]>a[k]) ? j : k;
6    if(a[i]<a[j]&&a[i]<a[k])
7      return (a[j]>a[k]) ? k : j;
8
9    return i;
10 }
11
12 void swap(int a[],int i, int j)
13 {
14   int temp=a[i];
15   a[i]=a[j];
16   a[j]=temp;
17 }
```

Table 4: Some functions for quicksort. These are two functions needed for quicksort, it has been split into two parts to help it fit nicer, the other part contains the actual algorithm, this part contains two of the functions it needs, the swap basically swaps the values at a[i] and a[j] and, if you unpack all the ternary operators, median returns i, j or k depending on which of a[i], a[j] and a[k] has the value in the middle when they are put in order.

```
1  void quick_r(int a[],int first , int last)
2  {
3     if(first>=last)
4        return;
5     if(last==first+1)
6        if(a[first]<a[last])
7           return;
8        else{
9          swap(a,first ,last);
10         return;
11       }
12
13    int i=first ,j=last -1;
14
15    swap(a,median(a,first ,first+1,last),last);
16
17    while(i<j){
18       while(a[j]>=a[last]&&j>first)
19          j--;
20       while(a[i]<a[last])
21          i++;
22       if(i<j)
23          swap(a,i,j);
24    }
25    swap(a,last ,i);
26
27    quick_r(a,first ,i-1);
28    quick_r(a,i+1,last);
29 }
30
31 void quick(int a[], int n)
32 {
33    quick_r(a,0,n-1);
34 }
```

Table 5: 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 6.

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`.

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.