Quick Sort

Quick sort was discovered by Tony Hoare in 1960. In general it has much better performance than brute force sorting methods like bubble sort or selection sort - $O(n \log_2 n)$ versus $O(n^2)$. It works by first of all by partitioning the array around a pivot value and then dealing with the 2 smaller partitions separately.

Partitioning is the most complex part of quick sort. The simplest thing is to use the first value in the array, a[l] (or a[0] as l = 0 to begin with) as the pivot. After the partitioning, all values to the left of the pivot are <= pivot and all values to the right are > pivot.

For example, consider

	l											r
k	0	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	**8**	2	5	13	4	19	12	6	3	11	10	7	9

where the value of a[l], namely 8, is chosen as pivot. Then the partition function moves along the array from the lhs until it finds a value > pivot. Next it moves from the rhs, passing values > pivot and stops when it finds a value <= pivot. This is done by the following piece of code:

```
i = l; j = r+1;
do ++i;
while( a[i] <= pivot && i <= r );

do --j;
while( a[j] > pivot );
```

Then if the lhs value is to the left of the rhs value, they are swapped. Variable i keeps track of the current position on moving from left to right and j does the same for moving from right to left. You then get

	l		i								j	r
a[k]	**8**	2	5	13	4	19	12	6	3	11	10	7	9

a[i] and a[j] are swapped to give

	l		i								j	r
a[k]	**8**	2	5	7	4	19	12	6	3	11	10	13	9

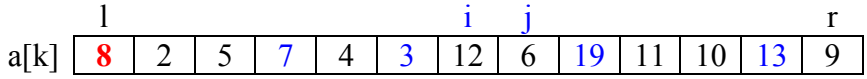
This process is repeated with i and j to get

	l				i			j				r
a[k]	**8**	2	5	7	4	19	12	6	3	11	10	13	9

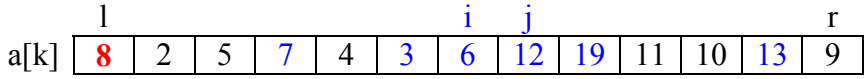
and do a swap to get

	l				i			j				r
a[k]	**8**	2	5	7	4	3	12	6	19	11	10	13	9

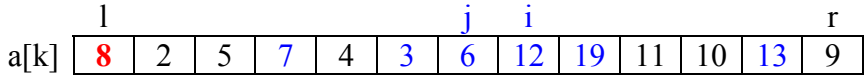
Move i and j again to get



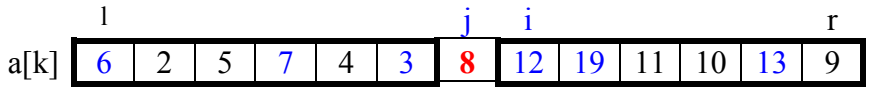
and swap again



Move i and j again.



When j passes i, the partitioning is finished. At this stage the partition code breaks out of the main while loop. All a[k] <= pivot where k <= j. All that we do next is swap the pivot with a[j] to get

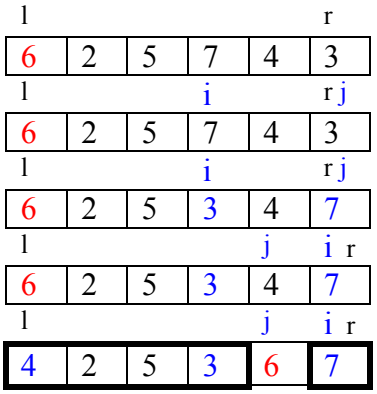


Then quickSort is called again separately on the two smaller arrays (here a[0] to a[5] and a[7] to a[12]). The pivot is left alone as it is in the correct position. So quickSort divides the problem into two smaller ones and recursively calls itself on each of then as in:

```
quickSort( a, l, j-1) or  quickSort( a, 0, 5)
quickSort( a, j+1, r) or  quickSort( a, 7, 12)
```

The smaller arrays are again partitioned in the same way as described above and so on. Eventually quickSort will arrive at arrays of size 1 or 2 or 0. Arrays of size 0 or 1 are already so to say sorted, while an array of size 2 can be sorted with an if statement. It is a waste of computation power to partition an array of size 2 and call quickSort twice more.

The lhs array is similarly partitioned from



Next do

```
quickSort( a, 0, 3)
quickSort( a, 5, 5)
```
– no more partition in rhs subarray.

Next **`quickSort( a, 0, 3)`** is tackled which partitions subarray as follows:

4	2	5	3
4	2	5	3
4	2	3	5
3	2	4	5

Next do

`quickSort( a, 0, 1)`

`quickSort( a, 3, 3)` – no more partition in rhs subarray.

The lhs subarray is again partitioned and a swap done

3	2
2	3

Next do

`quickSort( a, 0, 0)` – no more partition in lhs subarray.

Next we return to the original lhs subarray from the first partition

`quickSort( a, 7, 12)`

which partitions subarray as follows:

12	19	11	10	13	9
12	19	11	10	13	9
12	9	11	10	13	19
10	9	11	12	13	19

and calls **`quickSort( a, 7, 9)`** and **`quickSort( a, 11, 12)`**.

Next **`quickSort( a, 7, 9)`** is tackled which partitions subarray as follows:

10	9	11
9	10	11

No further partitioning occurs on these two subarrays. Next **`quickSort( a, 11, 12)`** is tackled which partitions subarray as follows:

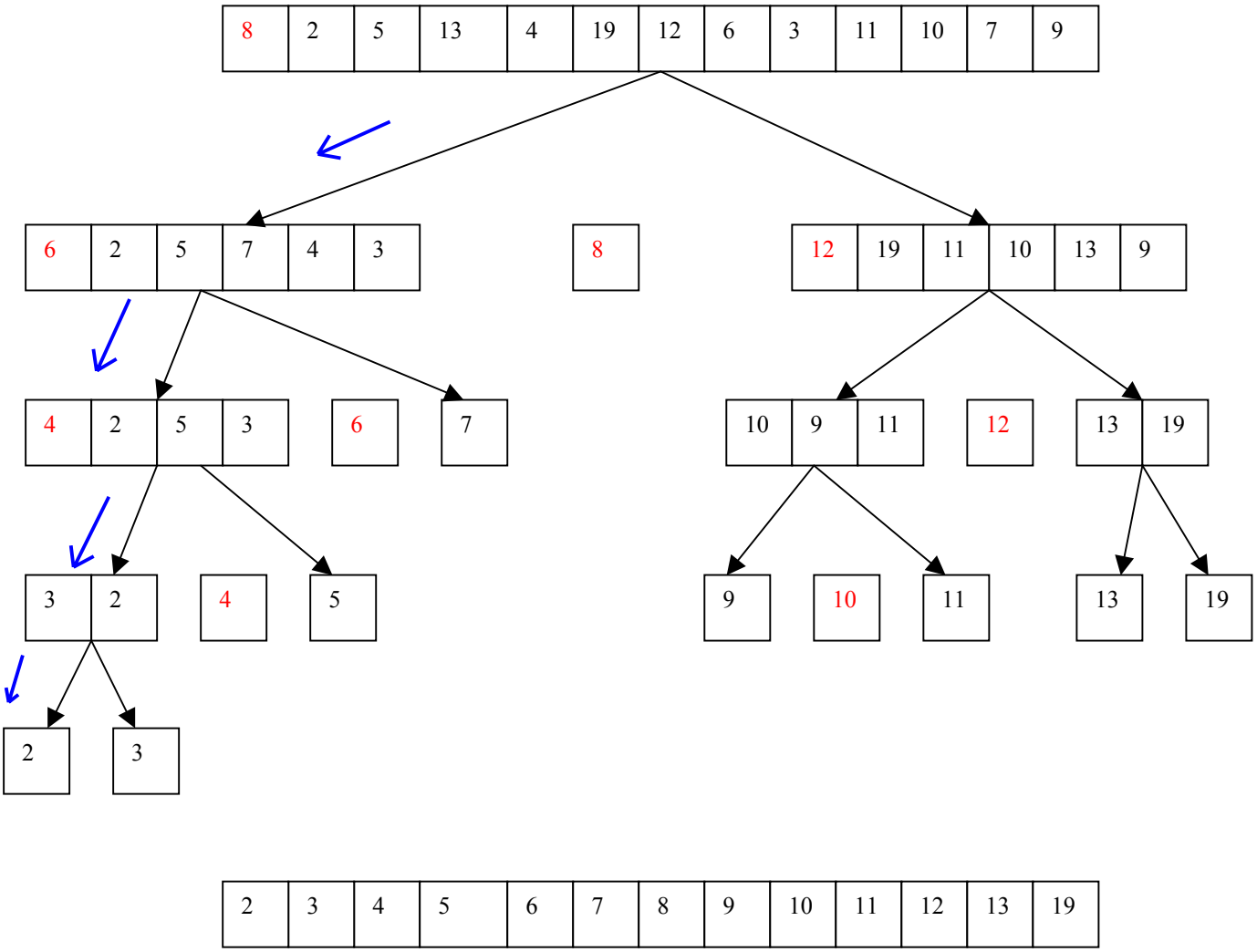
13	19
13	19

All the partitioning in now over with the result that in the meantime the array has been sorted.

Number of computations:

I counted: comparisons = 40 swaps = 13

Overview



Note: Quick sort works its way down through partitions, creating smaller ones all the time before it deals with all partitions on a given level. This is referred to as a ***depth-first*** approach.

quickSort.c

```
// quickSort.cpp
#include <stdio.h>

void quickSort( int[], int, int);
int partition( int[], int, int);

void main()
{
     int a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};

     int i;
     printf("\n\nUnsorted array is:  ");
     for(i = 0; i < 9; ++i)
          printf(" %d ", a[i]);

     quickSort( a, 0, 8);

     printf("\n\nSorted array is:  ");
     for(i = 0; i < 9; ++i)
          printf(" %d ", a[i]);

}



void quickSort( int a[], int l, int r)
{
     int j;

     if( l < r )
     {
        // divide and conquer
        j = partition( a, l, r);
        quickSort( a, l, j-1);
        quickSort( a, j+1, r);
   }

}
```

```
int partition( int a[], int l, int r) {
   int pivot, i, j, temp;
   pivot = a[l];
   i = l; j = r+1;

   while( 1)
   {
          do ++i; while( a[i] <= pivot && i <= r );
          do --j; while( a[j] > pivot );
          if( i >= j ) break;
          temp = a[i];  a[i] = a[j]);  a[j] = temp;
   }

   temp = a[l];  a[l] = a[j]);  a[j] = temp;
   return j;
}
```

Version from Sedgewick

```
// quickSort_Sedgew.cpp
#include <stdio.h>

void quickSort( int[], int, int);

void main()
{
     int a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};

     int i;
     printf("\n\nUnsorted array is:  ");
     for(i = 0; i < 9; ++i)
          printf(" %d ", a[i]);

     quickSort( a, 0, 8);

     printf("\n\nSorted array is:  ");
     for(i = 0; i < 9; ++i)
          printf(" %d ", a[i]);

}


void quickSort( int a[], int l, int r)
{
     int pivot, i, j, t;

     if( l < r )
     {
          pivot = a[l];
          i = l; j = r+1;

          while( 1)
          {
               do ++i; while( a[i] <= pivot && i <= r );
               do --j; while( a[j] > pivot );
               if( i >= j ) break;
               t = a[i]; a[i] = a[j]; a[j] = t;
          }
          t = a[l]; a[l] = a[j]; a[j] = t;

          quickSort( a, l, j-1);
          quickSort( a, j+1, r);
     }
}
```

quickSort_Ver2.cpp

This version is more efficient when dealing with small subarrays.

```
// quickSort_Ver2.cpp
// more efficient near end of sorting

// main() etc left out



void quickSort( int a[], int l, int r)
{
     int pivot, i, j, t;

   // sub array of size 1 or 0 - already sorted
   if(l >= r )
      return;

   // array of size 2, sort directly with if
   if( l+1 == r) {
      if(a[l] > a[r]) {
         t = a[l]; a[l] = a[r]; a[r] = t;
      }
      return;
   }

     pivot = a[l];
     i = l; j = r+1;

     while( 1)
     {
          do ++i; while( a[i] <= pivot && i <= r );
          do --j; while( a[j] > pivot );
          if( i >= j ) break;
          t = a[i]; a[i] = a[j]; a[j] = t;
     }
     t = a[l]; a[l] = a[j]; a[j] = t;

     quickSort( a, l, j-1);
     quickSort( a, j+1, r);
}
```