Here is Part 2 to the sorting algorithms

**Quicksort**

The quicksort divides the array into two groups and sorts the either group (conquering). It divides the array based on a random element from the array.

1. Check if array has at least 2 elements

2. Select the first element in the array (the pivot)

3. Move all elements that are less than or equal to the selected element to the left the array and move all the elements greater to the right of it (do this with the partition function)

     i. Set a variable low as the first index and a variable high as the last index.

     ii. Keep on incrementing low until its value is greater than the pivot value.

     iii. Keep on decrementing high until its value is less than or equal to the pivot value.

     iv. If low's value is less than high's value, swap the elements.

     v. Repeat steps ii-iv until high is less than or equal to low.

     vi. When high is less than or equal to low, swap the pivot element with the high element.

4. Repeat 1-3 each sub-array.

Let's try this with an array of numbers.

2 6 5 1 4 3

1. Select a Moses (pivot).

2. Partition the array based on Moses.

2 6 5 1 4 3

moses = 2

low = 0 (2)     index (value)

high = 5 (3)

low = 1 (6)     greater than moses

high = 4 (4)

high = 3 (1)     less than moses

Low (1) < high (3), so swap 6 and 1

2 1 5 6 4 3

low = 1 (1)

low = 2 (5)     greater than moses

high = 3 (6)

high = 2 (5)

high = 1 (1)    less than moses

Low (2) not < high (1), break out of loop

Swap moses with high

1 2 5 6 4 3

3. Repeat on left sub-array.

1

Since it has one element, return.

4. Repeat on right sub-array

5 6 4 3

moses = 5

low = 0 (5)

high = 3 (3)

low = 1 (6)     greater than moses

high = 3 (3)    less than moses

Low (1) < high (3), so swap 6 and 3

5 3 4 6

low = 1 (3)

low = 2 (4)

low = 3 (6)     greater than moses

high = 3 (6)

high = 2 (4)     less than moses

Low (3) not < high (2), break out of loop

Swap moses with high

4 3 5 6

5. Repeat on this sub-array's left sub-array

4 3

moses = 4

low = 0 (4)

high = 1 (3)

low = 1 (3)

low = 2 (?)     index greater than high

high = 1 (3)    less than moses

Low (2) not < high (1), break out of loop

Swap moses and high

3 4

5. Repeat on sub-array's right sub-array

6

Since it has one element, return.

foo = 3 4 5 6

Return

foo = 1 2 3 4 5 6

Return

Done!

Here is the code:

void quick(int foo[], int first, int last)     // Indexes of first and last elements you wish to sort

{

     if (last - first >= 1)     // If there are at least two elements

     {

          int moses;     // Index of partitioner

          moses = partition(foo, first, last);

          quick(foo, first, moses - 1);     // left sub-array

          quick(foo, moses + 1, last);     // right sub-array

     }

}

int partition(int sea[], int low, int high)

{

     int pivotIndex = low;

     int pivotValue = sea[low];     // Set pivotValue to first element

     do

     {

          while (low <= high && sea[low] <= pivotValue)     // Find first value greater than pivotValue

               low++;

          while (sea[high] > pivotValue)     // Find the first value less than or equal to the pivotValue

               high--;

          if (low < high)

               swap(sea[low], sea[high);

     }

     while (low < high);     // Continue looping until high has a lesser value than low

     swap(sea[pivotIndex], sea[high]);     // Swap the pivotValue with this one because we know that high points to the correct place for the pivot

     pivotIndex = high;     // Get the index of the high because that's the pivot's index now

     return pivotIndex;

}

What is the big-o? You exponentially split the array up (O(log(n))) and loop through all the elements for at most O(n) times. On average it is O(n \* log(n)).

void quick(int foo[], int first, int last)

{

     if (last - first >= 1)

     {

          int moses;     // Index of partitioner

          moses = partition(foo, first, last);

          quick(foo, first, moses - 1);     // O(log(n) / 2)

          quick(foo, moses + 1, last);     // O(log(n) / 2)

     }

}     // O(n \* log(n) / 2 + n \* log(n) / 2) = O(n \* log(n))

int partition(int sea[], int low, int high)

{

     int pivotIndex = low;

     int pivotValue = sea[low];

     do

     {

          while (low <= high && sea[low] <= pivotValue)

               low++;

          while (sea[high] > pivotValue)

               high--;

          if (low < high)

               swap(sea[low], sea[high);

     }

     while (low < high);     // O(n), at most n steps

     swap(sea[pivotIndex], sea[high]);

     pivotIndex = high;

     return pivotIndex;

}     // O(n)

In the worst case it is O(n^2). This is when you have an already ordered array, because you will split up the array n times instead of log(n).

               1 2 3 4 5 6

            1     2 3 4 5 6

                2     3 4 5 6

                    3     4 5 6

                         4     5 6

                              5   6

The quicksort is unstable.

2(a) 2(b) 1

moses = 2(a)

low = 0 (2(a))

high = 2 (1)

low = 1 (2(b))

low = 2 (1)

low = 3 (?)     greater than high

high = 2 (1)     less than high

Low (3) is not < high (2), break out of loop

Swap moses and high

1 2(b) 2(a)

[after evaluating left sub-array]

Done!

Notice that the 2s have not retained their original order.

**Merge Sort**

Merge sort breaks the array into halves and merges them together.

1. Check that the array has at least two elements

2. Split up the array in halves

3. Recursively repeat step 1 - 2 until you have at least two elements

4. Merge the two halves.

2 6 5 1 4 3

Half 1: 2 6 5

Half 1.1: 2

Half 1.2: 6 5

Half 1.2.1: 6

Half 1.2.2: 5

Merge half 1.2.1 and 1.2.2

5 6

Merge half 1.1 and 1.2

2 5 6

Half 2: 1 4 3

Half 2.1: 1

Half 2.2: 4 3

Half 2.2.1: 4

Half 2.2.2: 3

Merge half 2.2.1 and 2.2.2

3 4

Merge half 2.1 and 2.2

1 3 4

Merge half 1 and 2

1 2 3 4 5 6

void mergeSort(int foo[], int n)

{

     if (n <= 1)

          return;

     mergeSort(foo, n / 2);

     mergeSort(foo + n / 2, n);

     merge(foo, n / 2, n);     // foo now sorted

}

void merge(int arr[], int n1, int n2)

{

     int i = 0, j = 0, k = 0;     // i is used for first half of inputted array, j for second half, k for temp array

     int \*temp = new int[n1 + n2];     // temporary array

     int \*first = array;          // Pointer to beginning of first half

     int \*second = first + n1;     // Pointer to beginning of second half

     while (i < n1 || j < n2)

     {

          if (i == n1)     // first reached the end of its half

               temp[k++] = second[j++];

          else if (j == n2)     // second reached the end of its half

               temp[k++] = first[i++];

          else if (first[i] < second[j])

               temp[k++] = first[i++];     // Copy the value of the smaller number into temp, then increment the variables

          else // The second number must be smaller or equal to the first number

               temp[k++] = second[j++];

     }

     for (i = 0; i < n1 + n2; i++)

          arr[i] = temp[i];     // Override the contents of the original array with temp

     delete [] temp;

}

The big-o is always O(nlog(n)). Why? The mergeSort function always breaks the array in half log(n) times, and the merge function merges n items every time.

void mergeSort(int foo[], int n)

{

     if (n <= 1)

          return;

     mergeSort(foo, n / 2);     // O(log(n))

     mergeSort(foo + n / 2, n);     // O(log(n))

     merge(foo, n / 2, n);     // O(n)

}     // O(nlog(n))

void merge(int arr[], int n1, int n2)

{

     int i = 0, j = 0, k = 0;

     int \*temp = new int[n1 + n2];

     int \*first = array;

     int \*second = first + n1;

     while (i < n1 || j < n2)     O(n)

     {

          if (i == n1)

               temp[k++] = second[j++];

          else if (j == n2)

               temp[k++] = first[i++];

          else if (first[i] < second[j])     // \*\* Should this be <= instead?

               temp[k++] = first[i++];

          else

               temp[k++] = second[j++];

     }

     for (i = 0; i < n1 + n2; i++)     O(n)

          arr[i] = temp[i];

     delete [] temp;

}     O(n) (f(n) = n + n = 2n)

The merge sort is stable.

2(a) 2(b) 1

Half 1: 2(a)

Half 2: 2(b) 1

Half 2.1: 2(b)

Half 2.2: 1

Merge 2.1 and 2.2

1 2(b)

Merge 1 and 2

2(a) 1 2(b)

n1 = 1

n2 = 2

i = 0 (2(a))     index(value)

j = 0 (1)

first[0] > second[0], copy 1

temp = 1

j = 1 (2(b))

first[0] == second[1], copy 2(b)     \*\* Is this a mistake? Wikipedia says that mergesort is stable most of the time, is this an exception or is there a mistake?

temp = 1 2(b)

j = 2 (?)

j == n2, copy 2(a)

temp = 1 2(b) 2(a)

arr = 1 2(b) 2(a)