Quicksort and Mergesort

Divide and Conquer Sorting

- Divide the elements to be sorted into two groups of roughly equal size
- Sort each of these smaller groups of elements
- Combine the two sorted groups into one large sorted list

Any time you see "divide and conquer", you should think recursion.

The Quicksort Algorithm

1. If the array contains only 0 or 1 element, return.
2. Select an arbitrary element P form the array (typically the first element).
3. Move all elements that are less than or equal to P to the left of the array and all elements greater than P to the right.
4. Recursively repeat this process on the left sub-array and then the right sub-array.

```
void Quicksort(int Array[], int First, int Last)
{
    if(Last - First) >= 1)
    {
        int PivotIndex;
        PivotIndex = Partition(Array, First, Last);
        Quicksort(Array, First, PivotIndex - 1); //left
        Quicksort(Array, PivotIndex + 1, Last); //right
    }
}
int Partition(int a[], int low, int high)
{
    int pi = low; //index of first element of array
    int pivot = a[low]; //first element of array
    do
    {
        while(low <= high && a[low] <= pivot)
            low++;
        //low is now equal to the index of an element greater than the first element
        while(a[high] > pivot)
            high--;
```

```
        //high is now equal to the index of an element less than or equal to the first element
        if(low < high)
            swap(a[low], a[high]); //swap the two
    } while(low < high);
    swap(a[pi], a[high]);
    pi = high;
    return pi; //return the pivot's index to the array
}
```

Big-O

- We first partition the array at a cost of **n** steps.
- Then we repeat the process the each half.
- We partition each of the 2 halves, each taking **n / 2** steps, at a total cost of **n** steps.
- Then we repeat the process each half....
- At each level, we do **n** operators, and we have **log2(n)** levels.

O(n * log2(n))
Notes: if our array is already sorte, mostly sorted, or in reverse order, then quicksort becomes very slow!
The worst case of quicksort is **O(n^2)**
Quicksort is **unstable**. It is apparently slower than mergesort when sorting linked lists.
Quicksort uses a variable amount of RAM.
Quicksort is better for arrays and can be used when average case performance matters more than worst case performance.

Mergesort
The basic merge algorithm takes two pre-sorted arrays as inputs and outputs a combined, third sorted array.

1. Initialize counter variables i1, i2 to zero
2. While there are more items to copy...
    1. If A1[i1] is less than A2[i2], copy A1[i1] to output Array B and i1++
    2. Else, copy A2[i2] to output array B and i2++
3. If either array runs out, copy the entire contents of the other array over

By always selecting and moving the smallest book from either shelf, we guarantee all of our books will end up sorted!

```
void merge(int data[], int n1, int n2)
{
    int i = 0, j = 0, k = 0;
    int *temp = new int[n1 + n2];
    int *sechalf = data + n1;
    while(i < n1 || j < n2)
    {
        if(i == n1)
```

```
            temp[k++] = sechalf[j++];
        else if(j == n2)
            temp[k++] = data[i++];
        else if(data[i] < sechalf[j])
            temp[k++] = data[i++];
        else
            temp[k++] = sechalf[j++];
    }
    for(i = 0; i < n1 + n2; i++)
        data[i] = temp[i];
    delete [] temp;
}
```

Full mergesort algorithm:

1. If array has one element, then return (it's sorted)
2. Split up the array into two equal sections
3. Recursively call Mergesort function on the left half
4. Recursively call Mergesort function on the right half
5. Merge the two halves using our `merge()` function

O(n * log2(n))
Notes: MergeSort is **stable**.

Sorting Overview

Sorting Overview

Sort Name	Stable/ Non-stable	Notes
Selection Sort	Unstable	Always $O(n^2)$, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	$O(n)$ for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	$O(n)$ for already or nearly-ordered arrays (with a good implementation). $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	$O(n^{1.25})$ approx. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.
Quick Sort	Unstable	$O(n \log_2 n)$ average, $O(n^2)$ for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up $O(n)$ slots of extra RAM (for recursion) in the worst case, $O(\log_2 n)$ avg.
Merge Sort	Stable	$O(n \log_2 n)$ always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. Downside: Requires n slots of extra memory/disk for merging – other sorts don't need xtra RAM.
Heap Sort	Unstable	$O(n \log_2 n)$ always. Sometimes used in low-RAM embedded systems because of its performance/low memory req'ts.