Sorting

Sorting is the process of ordering a bunch of items on one more more rules, subject to one or more constaints

Items: what are we sorting, and how many are there?

- Strings, numbers, objects
- Thousands, millions

Rules: how do we order them?

- Ascending, descending
- Based on what criteria?

Constaints

- Is data in an array or a linked lsit?
- Are the items in RAM or on disk?

2 Rules of Sorting

- Don't choose a sorting algorithm until you understand the requirement of your problem
- Always choose the simplest sorting algorithm possible that meets your requirements

Inefficient Sorting Algorithms

Selection sort, insertion sort, bubble sort, etc.

Generally require $O(N^2)$ steps to order N values

They are slow because they compare every items to every other item, swapping out-of-order items

Stable Sort

An unstable sorting algorithm re-orders the items without taking into account their initial ordering

A stable sorting algorithm does take into account the initial ordering when sorting, maintaining the order of similar-valued items

Selection Sort

Unstable sort

Steps

- Look at all N books, select the shortest book

    - Swap this with the first book
- Look at the remaining N-1 books, and select the shortest
    - Swap this with the second book
- Look at the remaining N-2 books, so on...

Efficiency: **$O(N^2)$**

- Each look at N books is roughly N, swap is 1 step
    - We have to look N times

> Selection sort takes just as many steps when input data is already sorted or not sorted

Code

```
void selectionSort(int A[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (A[j] < A[minIndex])
                minIndex = j;
        }
        swap(A[i], A[minIndex]);
    }
}
```

- For each of the n array elements
    - Locate the smallest item in the array between the ith slot and slot n - 1
    - Swap the smallest item found with slot A[i]

Insertion Sort

Algorithm

- Start with set size s = 2
- While there are still books to sort
    - Focus on the first s books
    - If the last book in this set is in the wrong order
        - Remove it from the shelf
        - Shift the books before it to the right as necessary

        - Insert book into proper slot
    - s = s + 1

Efficiency: **$O(N^2)$**

> If all books are in proper order already, then Insertion Sort will be very quick

- No need to do any shifting
- A perfeclty mis-ordered set of books is the worst case
    - Every round requires the maximum shifts

Code that sorts an array in ascending order

```
void insertionSort(int A[], int n)
{
    for (int s = 2; s <= n; s++)
    {
        int sortMe = A[s - 1];

        int i = s - 2;
        while (i >= 0 && sortMe < A[i])
        {
            A[i + 1] = A[i];
            --i;
        }
        A[i+1] = sortMe;
    }
}
```

- Focus on successively larger prefixes of the array
    - Start with the first s = 2 elements
- Make a copy of the last val in the current set, this opens up a slot in the array for us to shift items
- Shift the values in the focus region right until we find the proper slot for sortMe
- Store the sortMe value into the vacated slot

Bubble Sort

Algorithm

- Start at the top element of your array
- Compare the first two elements: A[0] and A[1]
    - If they are out of order, then swap them
- Then advance one element in your array

- Compare these two elements: A[1] and A[2]
    - If they are out of order, swap them
- Repeat until you hit the end of the array
- When you hit the end, if you make at least one swap, repeat the whole process again

Efficiency: **$O(N^2)$**

- Each pass through the array is N steps
- Worst case, there is swap every pass, repeat the process N times

> But really efficient on pre-sorted arrays and linked lists

Code

```
void bubbleSort(int Arr[], int n)
{
    bool atLeastOneSwap;

    do
    {
        atLeastOneSwap = false;
        for (int j = 0; j < (n-1); j++)
        {
            if (Arr[j] > Arr[j+1])
            {
                Swap(Arr[j], Arr[j+1]);
                atLeastOneSwap = true;
            }
        }
    }
    while (atLeastOneSwap == true);
}
```

- Start by assuming that we won't do any swaps
- Compare each element with its neighbor and swap them if they're out of order
- Don't forget we swapped
- If swapped at least once, then start back at the top and repeat whole process

---

Divide and Conquer Sorting Algorithms

Quicksort and Mergesort are efficient "divide and conquer" sorting algorithms

Generally work as follows:

1. Divide the elements to be sorted into two groups of roughly equal size

2. Sort each of these smaller groups of elements (conquer) using recursion
3. Combine the two sorted groups into one large sorted group

Generally **$O(Nlog2(N))$**

Quicksort

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

Code

```
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
    }
}
```

- `First` specifies the starting element of the array to sort (index)
- `Last` specifies the last element of the array to sort (index)
- `if(Last - First >= 1)`: We only sort arrays of at least two elements
- Partition
    - Move <= items left
    - Move > items right
    - The `Partition` function uses the first item as the pivot value and moves less-than-or-equal items to the elft and larger ones to the right

```
int Partition(int a[], int low, int high)
{
```

```
    int pi = low;
    int pivot = a[low];
    do
    {
        while(low <= high && a[low] <= pivot)
            low++;
        while (a[high] > pivot)
            high--;
        if (low < high)
            swap(a[low], a[high]);
    }
    while (low < high);
    swap(a[pi], a[high]);
    pi = high;
    return (pi);
}
```

- Select the first item as our pivot value `int pi = low`
- doWhile loop, while `low < high`
  - Find the first value > than the pivot
  - Find the first value <= than the pivot
  - Swap the larger with the smaller
  - Find the next value > than the pivot
  - Find the next value <= than the pivot
  - Swap larger with the smaller
- Swap pivot to proper position in the array
- Finally, return the pivot's index in the array to the QuickSort function

Efficiency: **$O(nlog2(n))$**

- We first partition the array at a cost of n steps
  - Partition function moves smaller values to the left of the array, larger values to the right
- Repeat the process for each half
  - Partition each of the 2 halves, each taking n/2 steps, at a total cost of n steps
- At each level, we do n operations, and we have log2(n) levels (due to splitting in half)

> Is it always fast?

- If array is already sorted or mostly sorted, then quicksort becomes **very slow**

Worst Case Big-O of Quicksort: When array mostly in order or in reverse order

$O(N^2)$

- First partition the array at a cost of n steps
- Our partition function moves smaller values of pivot to the left and larger to the right
    - But since array already sorted, the pivot value (first value) IS THE SMALLEST value
- We repeat process for the left & right groups
    - But the left group is empty, and the right group still has nearly n items
- Each time we partition, we remove **only one item** off the left side -> then partitioning has to happen n times to process the entire array
- If partition algorithm requires n steps at each levels, and there are n levels -> n^2

Mergesort

The merge algorithm takes two-presorted arrays as inputs and outputs a combined, third sorted array

Merge Algorithm

1. Initialize the counter variables i1, i2 to zero
2. While there are more items to copy
    - If A1[i1] is less than A2[i2] Copy A1[i1] to output array B and i1++
    - 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

Merge Code

```
void merge(int data[], int n1, int n2, int temp[])
{
    int i1 = 0, i2 = 0, k = 0;
    int *A1 = data, *A2 = data + n1;

    while (i1 < n1 || i2 < n2)
    {
        if (i1 == n1)
            temp[k++] = A2[i2++];
        else if (i2 == n2)
            temp[k++] = A1[i1++];
        else if (data[i1] <= A2[i2])
            temp[k++] = A1[i1++];
        else
            temp[k++] = A2[i2++];
    }
    for (int i = 0; i < n1 + n2; i++)
        data[i] = temp[i];
}
```

- Pass in an input array called `data` and the sizes of the two parts of it to merge: `n1` and `n2`

```
[1, 13, 21, 4, 11, 25, 30]
          n1=3          n2=4
A1          A2
```

- `temp` is a temporary array of size `n1+n2` that holds the merged resuts as we loop
- Finally, we copy our merged results back to the `data` array

Mergesort Algorithm

1. If the array has one element, then return (already 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

Efficiency: **$O(nlog2(n))$**

- There are log2n levels deep, because we keep dividing our piles in half
- Each merge goes through n items, a merge happens at each level, so nlog2n

Problem Cases for Mergesort: NO

- Mergesort works equally well regardless of the ordering of the data
- But mergesort needs secondary arrays to merge-> can slow things down a bit
    - Quicksort does not allocate any new arrays

 picture 1