Lecture #11

- Sorting Algorithms, part II:
  - Quicksort
  - Mergesort
- Trees
  - Introduction
  - Implementation & Basic Properties
  - Traversals: The Pre-order Traversal
- On-your-own Study
  - Full binary trees

Divide and Conquer Sorting

The last two sorts we'll learn (for now) are Quicksort and Mergesort.

These sorts 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).
3. Combine the two sorted groups into one large sorted list.

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

The Quicksort Algorithm

1. If the array contains only 0 or 1 element, return.
2. Select an arbitrary element P from the array (typically the first element in the array).
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 (this is called partitioning).

Divide

4. Recursively repeat this process on the left sub-array and then the right sub-array.

Conquer

13	1	21	30	69	40	77

The QS Partition Function

The Partition function uses the first item as the pivot value and moves less-than-or-equal items to the left 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);
}
```

And finally, return the pivot's index in the array (4) to the QuickSort function.

4	1	12	13	30	52	40	99	77	35	47	56
0	1	2	3	4	5	6	7	8	9	10	11

Big-oh of Quicksort

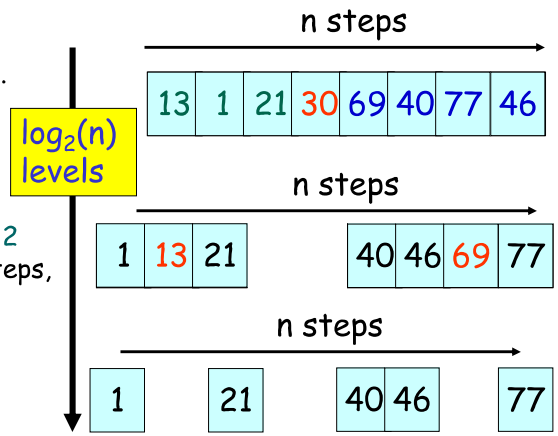
We first partition the array, at a cost of n steps.

Then we repeat the process for 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 for each half…

We partition each of the 4 halves, each taking n/4 steps, at a total cost of n steps.



So at each level, we do n operations, and we have $\log_2(n)$ levels, so we get: $n \log_2(n)$.

Other Quicksort Worst Cases?

So, as you can see, an array that's mostly in order will require an average of N^2 steps!

As you can probably guess, Quicksort also has the same problem with arrays that are in reverse order!

So if you happen to know your data will be mostly sorted (or in reverse) order, avoid Quicksort!

It's a DOG!

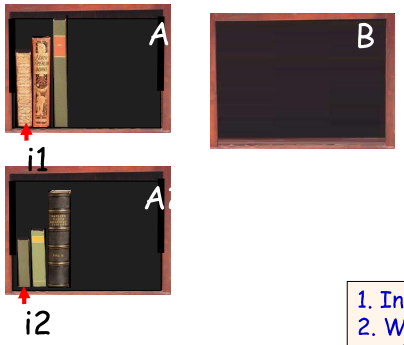
Mergesort

The Mergesort is another extremely efficient sort – yet it's pretty easy to understand.

But before we learn the Mergesort, we need to learn another algorithm called "merge".

Mergesort

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



Merge Algorithm
Consider the left-most book in both shelves
Take the smallest of the two books
Add it to the new shelf
Repeat the whole process until all books are moved

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

1. Initialize 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 Algorithm in C++

```
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;
}
```

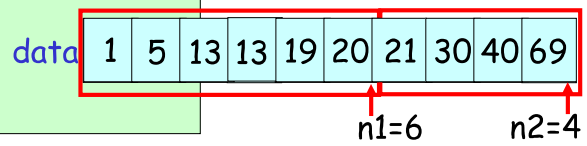
Here's the C++ version of our merge function!

Instead of passing in A1, A2 and B...

you pass in an array called data and two sizes: n1 and n2

Notice how this function uses new/delete to allocate a temporary array for merging.

data holds the merged contents at the end.



Mergesort

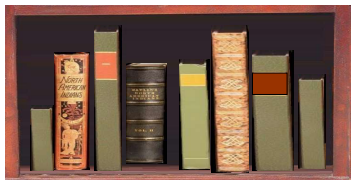
OK – so what's the full mergesort alogrithm:

Mergesort function :

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

Ok, let's see how to mergesort a shelf full of books!

Big-oh of Mergesort



n items merged

n items merged

$\log_2 n$ levels deep

Why? Because we keep dividing our piles in half...

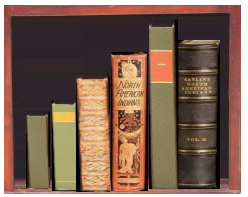
until our piles are just 1 book!

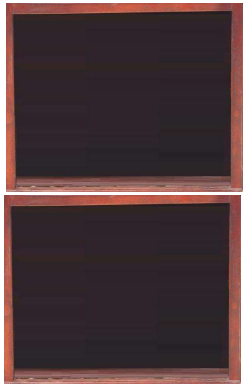
Overall, this gives us $n \cdot \log_2(n)$ steps to sort n items of data. Not bad! ☺

Mergesort – Any Problem Cases

So, are there any cases where mergesort is less efficient?

No! Mergesort works equally well regardless of the ordering of the data...





However, because the merge function needs secondary arrays to merge, this can slow things down a bit...

In contrast, quicksort doesn't need to allocate any new arrays to work.