CS416

Introduction to Computer Science II

Spring 2018

11 Sorting and Searching

Previously in 416

- Regular expressions

Preview

- *Algorithm analysis* issues
- Sorting algorithms
- Searching algorithms

Based on Sanders and van Dam, Chapter 16

Algorithm Analysis

- How do we measure the <u>efficiency</u> of an algorithm?
  - Run time?
    - depends on computer, language, compiler, programmer skill, the data -- too many variables to be meaningful
  - Define an *abstraction* of the algorithm in terms of basic <u>*logical operations*</u> that characterize the algorithm
- Also need an *abstraction* for input and decide what measure is most useful; for example
  - average case -- probably most useful, often harder
  - worst case -- usually easier to do, gives upper bound
  - best case -- seldom useful

Logical Operations

- For each class of algorithm ask
  - What are the most important (most often done) operations?
    - Need to identify the operations required to process each "unit" of input; these are called the "steps" of the executing program
    - For example, for <u>searching</u> we asked: how many item <u>comparisons</u> does it take to find the item we seek?
  - What is the nature of the <u>data</u> processed by the algorithm?
    - What are its units? Is it ordered?

Worst Case Analysis

- Given <u>all</u> possible inputs of size N, what is the maximum execution time (in terms of operations)?
- Call this T(N)
- Probably not as useful as computing an *average* time, but average is usually much harder.

Time Analysis

- Once (if) we can figure out how many operations an algorithm takes for N inputs, we then want to compare that to other algorithms
  - Is there a <u>real</u> difference between 4N and 5N?
    - we abstracted a lot detail to define and count “operations”, so we can’t put any faith in differences of “constants”
  - What about 8N versus $4N^2$?
    - we care especially about very large N
    - 8 * 100,000 << 4 *(100,000 * 100,000) = 4 * 100,000,000
  - Need an <u>abstraction</u> for the time analysis

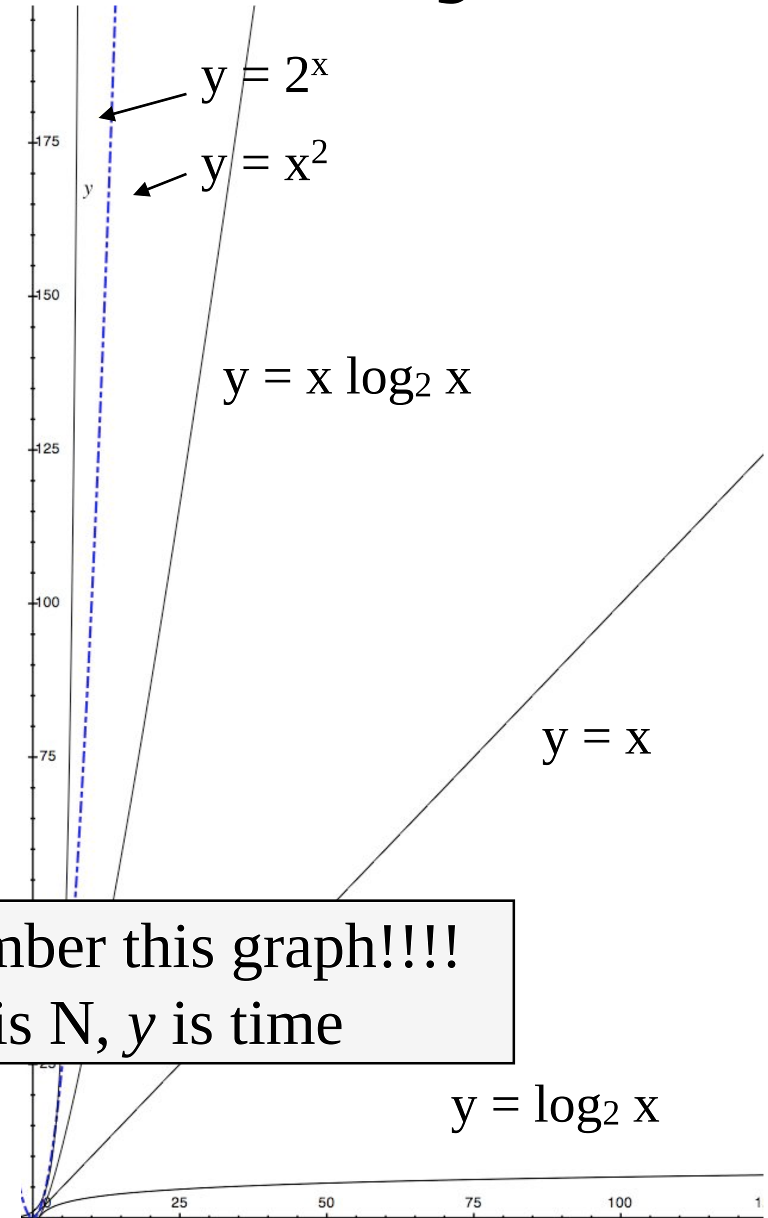
Big O Notation

time "cost"	Big O
10000	$O(1)$
4	$O(1)$
$20N + 22$	$O(N)$
$N - 30000$	$O(N)$
$100N^2$	$O(N^2)$
$100N^2 + 30N$	$O(N^2)$
$2^N + 4N^2 + 2N$	$O(2^N)$

- Big O notation: abstraction for time measurements based on abstract operations given N input units
- I.e., time does not depend on N $O(1)$ is called *constant time*
- An $O(N)$ algorithm is said to be "on the order of N"
- Categorize only by fastest growing term; it dominates all others

Growth Rates are Key

- Graphs show how dramatic the time growth can be with exponent-based time (x^2 and 2^x)
- That's why other terms don't matter
- x^2 and 2^x don't look much different, but they **<u>are</u>** for big enough x:
  - $30^2 = 900$, but
  - $2^{30} = 1{,}073{,}741{,}824$
- $x \log_2 x$ grows much more slowly
  - x = 30, $x \log_2 x = 147$
  - x = 100, $x \log_2 x = 767$
- $\log_2 x$ is nearly flat!
  - x = 32, $\log_2 x = 5$
  - x = 1024, $\log_2 x = 10$



Remember this graph!!!!
x is N, *y* is time

Basic Sorting

- Given N items, sort them in alphabetic (or numeric) order.
- We'll look at 4 algorithms:
    - Bubblesort
    - Insertion sort
    - Selection sort
    - Merge sort
- We'll describe all as if the data is in an array, but there are variations that work for lists as well

Bubble Sort I

- Compare neighbors in the array; if they are not in order, swap them, and move 1 step to right and do it again.
- At end of array go back to the start and do it again -- until you make a pass without swapping anything.

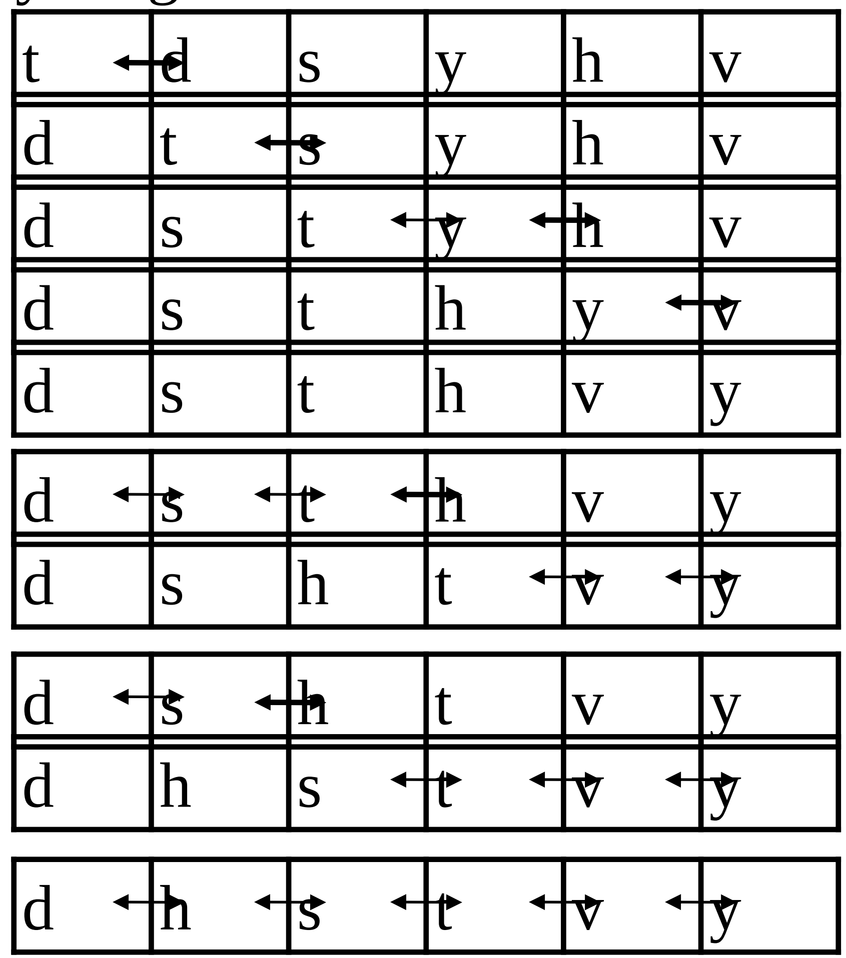
```
bubblesort( int data[]):
int n = data.size;
sorted = false;
while ( !sorted && n > 1 )
{
  sorted = true;
  for ( i=1; i<n; i++ )
  {
    if ( data[i-1] > data[i] )
       swap( data, i-1, i );
       sorted = false;
  }
}
```

Start at 1, not 0, since comparing i-1 and i-th

This test can be omitted, (but it helps analysis)

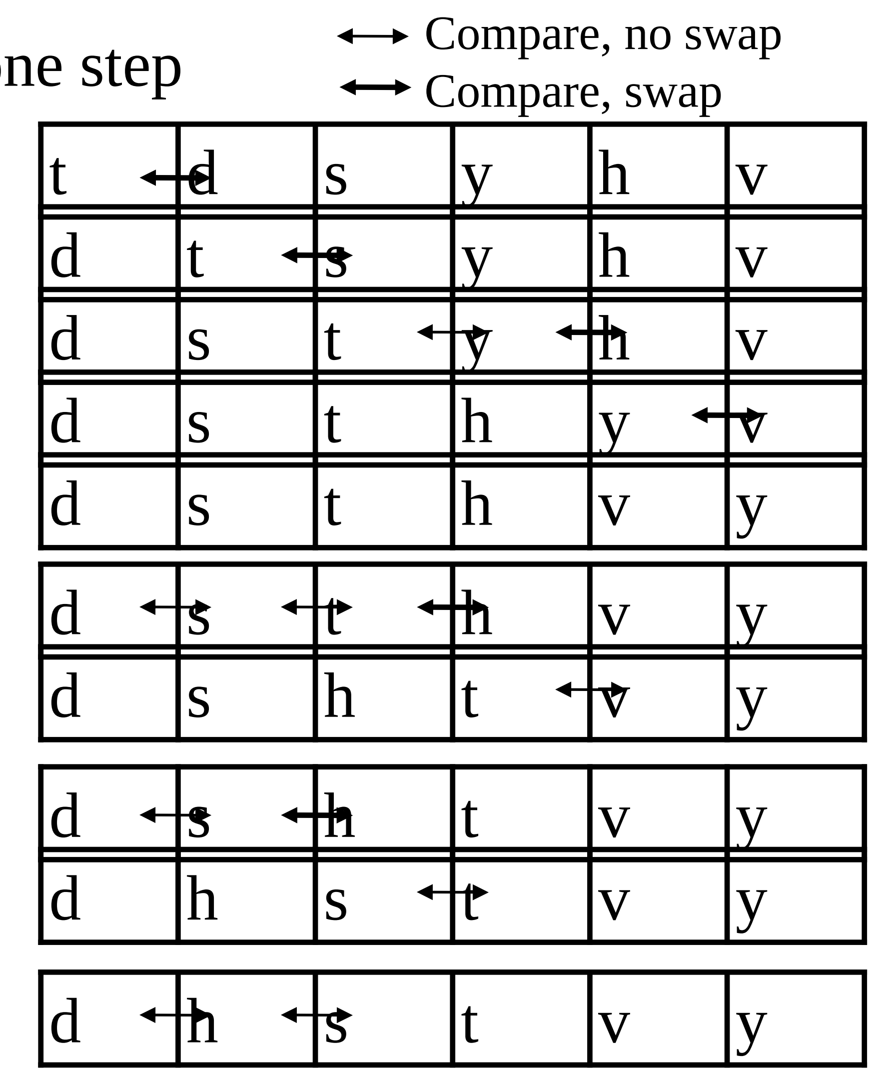
Compare, no swap

Compare, swap



Bubble Sort II

- Notice that after the first pass through the array, the "highest" item must be at the end position: during that pass, it will keep being swapped until the end
- On the next pass, we can stop comparing one step earlier; this is true for every pass




```
bubblesort( int data[]):
int n = data.size;
sorted = false;
while ( !sorted && n > 1 )
{
  sorted = true;
  for ( i=1; i<n; i++ )
  {
    if ( data[i-1] > data[i] )
        swap( data, i-1, i );
        sorted = false;
  }
  n--;
}
```

Drop last entry of subarray used in this step.

Bubblesort Analysis

```
bubblesort abstraction:
n = N
while ( n > 1 )
  for ( i=1; i<n; i++ )
    if ( comparison )
      swap;
  n--;
```

- What are important operations?
  - Comparisons and swaps
- What is worst case?
  - Need a swap and a comparison at every test
  - How many times is the **while** body be executed? N - 1
  - How many times is **for** body executed for each **while** body? n - 1
    - which is N-1 then N-2 then N-3 ...then 1: this sums to N(N-1)/2
  - So, have $(N^2-N)/2$ comparisons and sorts: $O(N^2)$
- What is best case?
  - No swaps; but still must <u>always</u> do $(N^2-N)/2$ comparisons!
    - Note: this version does not have early exit test if no swap

Insertion Sort

- Card player's sort
  - Assume you have a pile of unordered objects (could be cards on a table or in an array) and you have another array into which you will sort these objects

while still objects left

pick one and <u>insert</u> it into its correct location in the sorted array

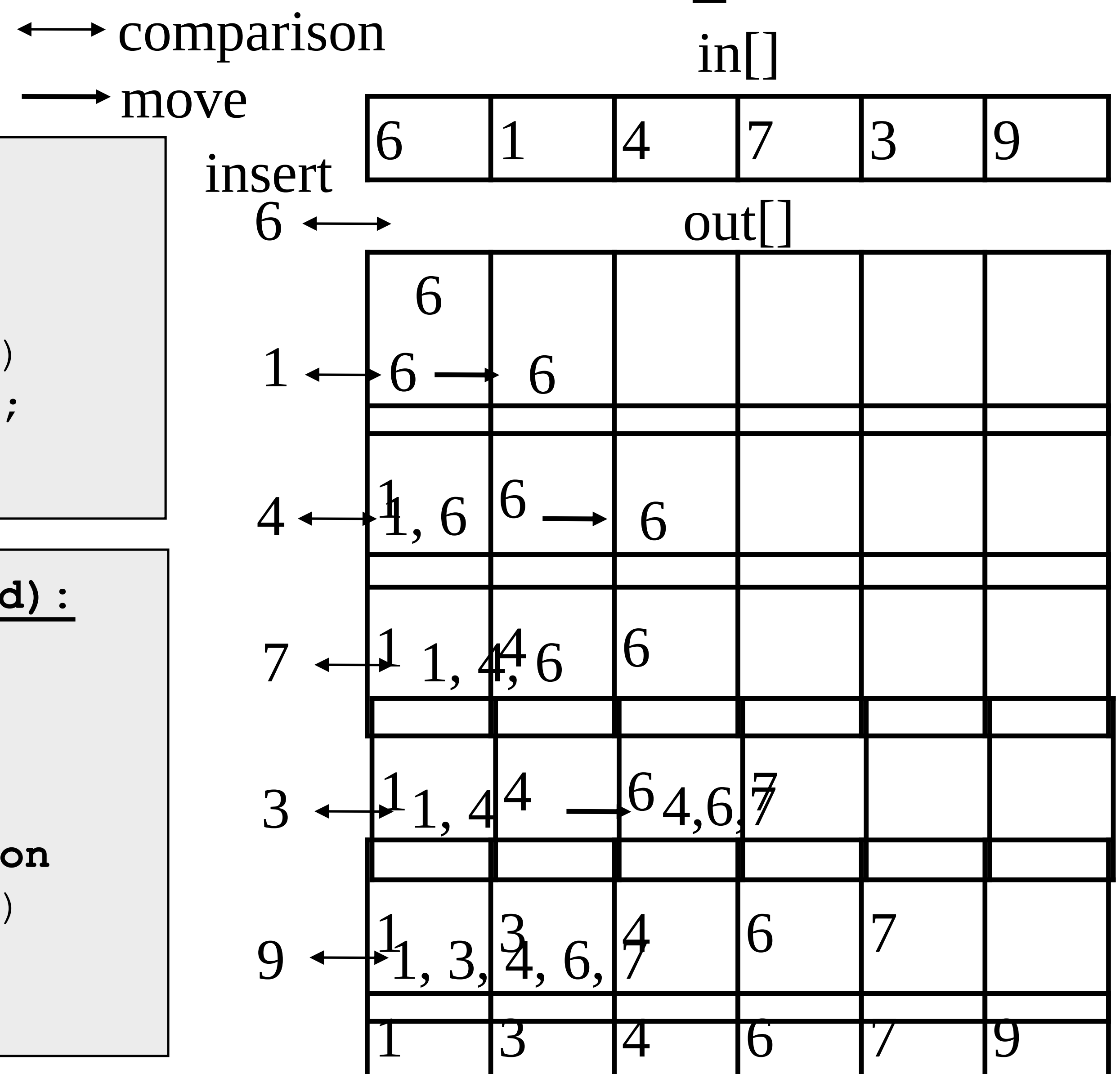
```
insertionSort( int[] in ):
int n = in.length;
int[] out = new int[ n ];
for ( int i = 0; i < n; i++ )
   insert( out, i, in[ i ] );
return out;
```

```
insert( int[] a, int k, int d):
int i = 0;
// find insertion point
while ( i < k && a[i] < d )
   i++;
// make room for the insertion
for ( int j = k; j > i; j-- )
   a[ j ] = a[ j - 1 ];
a[ i ] = d;   // insert it
```

Insertion Sort Example

```
insertionSort( int[] in ):
int n = in.length;
int[] out = new int[ n ];
for ( int i = 0; i < n; i++ )
   insert( out, i, in[ i ] );
return out;
```

```
insert( int[] a, int k, int d):
int i = 0;
// find insertion point
while ( i < k && a[i] < d )
   i++;
// make room for the insertion
for ( int j = k; j > i; j-- )
   a[ j ] = a[ j - 1 ];
a[ i ] = d;    // insert it
```



Insertion Sort In Place

- Why use 2 arrays? Can sort in place.

Can use same framework; don't need return

```
insertionSort( int[] arr ):
int n = arr.length;
for ( int k = 0; k < n; k++ )
   insert( arr, k, arr[ k ] );
```

```
insert( int[] a, int k, int d):
int i = 0;
// find insertion point
while ( i < k && a[i] < d )
   i++;
// make room for the insertion
for ( int j = k; j > i; j-- )
   a[ j ] = a[ j - 1 ];
a[ i ] = d;   // insert it
```

Note: start at 1

Or, merge into 1 method

```
insertSort( int[] arr ):
for ( int k=1; k < arr.length; k++ )
{
   int d = arr[k];
   int i = 0;
   // find insertion point
   while ( i < k && arr[ i ] < d )
      i++;
   // make room for the insertion
   for ( int j = k ; j > i; j-- )
     arr[ j ] = arr[ j - 1 ];
   arr[i] = d;   // insert it
}
```

Insertion Sort in Place 2

- Can it be simpler?
- Can we merge the inner *while* and *for* loops?
- Together they go through the array once: the *while* goes up to the insertion point, doing comparisons and the *for* goes *down* to the insertion point, copying elements
- If we searched <u>backwards</u> through the sorted portion of the array initially, we could <u>move as we compare</u> and do it in one loop.

```
for . . .
   int i = 0;
   while ( i < k && arr[ i ] < d )
      i++;
   for ( int j = k ; j > i; j-- )
     arr[ j ] = arr[ j - 1 ];
   arr[ i ] = d;
```

```
insertionSort( int[] arr ):
for( int k = 1; k < arr.length; k++ )
{
  int d = arr[ k ]; // next unsorted value
  int i;
  for( i=k; i>0 && arr[ i-1 ] >= d ), i--)
     arr[ i ] = arr[ i-1 ]; // move
  arr[ i ] = d;   // insert
}
```

Insertion Sort Analysis

- The outer *for* always executes *N-1* times
- In the *worst case*, the array is inversely sorted and the comparison test of the inner *for* loop is never true
- this means that the inner loop executes 1 time for k=1, then 2 for k = 2, etc. upto N-1 for k = N-1 or: *sum(k) for k = 1 .. N-1*
- This is N*(N-1)/2

```
insertionSort( int[] arr ):
for k = 1 to arr.length - 1 // N-1
{
  d = arr[k];
  for i=k downto 1 until arr[i] >= d
     move
  insert
}
```

k	1	2	...	N-2	N-1
inner loop	1	2	...	N-2	N-1

Total moves = 1 + 2 + … + N-1 = N*(N-1)/2

Algorithm is $O(N^2)$

Selection Sort

- Given a pile of objects and an array into which you will sort them:
  - Pick the smallest object, put in 1st position
  - Pick the smallest remaining object, put in 2nd position
  - Continue until done
- Assume initial objects are stored in an array and we'll sort in place.
  - Note: don't need to do a test if only 1 object left unsorted; it's the max and is at the end

```
selectionSort( int[] arr ):
int n = arr.length;
for ( int i = 0; i < n - 1; i++ )
{
  int p = minPos( arr, i );
  int temp = arr[ p ]; //swap
  arr[ p ] = arr[ i ];
  arr[ i ] = temp;
}
```

```
int minPos( int[] arr, int k):
int n = arr.length;
int minP = k;  // let k be min
for (int i = k + 1; i < n; i++)
{
  if ( arr[ i ] < arr[ minP ] )
     minP = i;
}
return minP;
```

Selection Sort Example

```
selectionSort( int[] arr ):
int n = arr.length;
for ( int i = 0; i < n - 1; i++ )
{
  int p = minPos( arr, i );
  int temp = arr[ p ]; //swap
  arr[ p ] = arr[ i ];
  arr[ i ] = temp;
}
```

```
int minPos( int[] arr, int k):
int n = arr.length;
int minP = k;  // let k be min
for ( int i = k + 1; i < n; i++ )
{
  if ( arr[ i ] < arr[ minP ] )
     minP = i;
}
return minP;
```

6	1	4	7	3	9

minLoc(arr, 0) ⇒ 1@1; swap with 6@0

1	6	4	7	3	9

minLoc(arr, 1) ⇒ 3@4; swap with 6@1

1	3	4	7	6	9

minLoc(arr, 2) ⇒ 4@2; swap with 4@2

1	3	4	7	6	9

minLoc(arr, 3) ⇒ 6@4; swap with 7@3

1	3	4	6	7	9

minLoc(arr, 4) ⇒ 7@4; swap with 7@4

1	3	4	6	7	9

Selection Sort Analysis

- Outer loop body always executes N-1 times
- Inner loop body executes N-1 times then N-2, then N-3 etc.
- N-1 + N-2 ...
  = sum(k) for k = 1..N-1
  which is N(N-1)/2
- Selection sort is O(N^2)

```
selectionSort( int[] arr ):
for k = 0 to N-2
{
  for j = k+1 to N-1
     if ( … )
}
```

k	0	1	...	N-3	N-2
inner loop	N-1	N-2	...	2	1

Total compares = N-1 + N-2 + … + 2 + 1 = N*(N-1)/2

Algorithm is O(N^2)

A Better Sort

- Bubble, insertion and selection sort are all $O(N^2)$
- There must be a better way!
  - It's called *merge sort* (and an even better variant, *quicksort*)
- Suppose we have 2 <u>sorted</u> arrays
  - Can <u>merge</u> these into a new array in O(N) time
  - OK, but how do we get the 2 sorted arrays in the first place?
    - *divide and conquer!*

```
Merge( int[] a, int[] b ):
int[] c = new int[aSize + bSize]
int ia = 0, ib = 0, ic = 0;
while ia < aSize && ib < bSize
  if a[ ia ] <= b[ ib ]
    c[ ic++ ] = a[ ia++ ]
  else
    c[ ic++ ] = b[ ib++ ];
move rest of a to c
move rest of b to c
```

Merge analysis

N data copies

every item copied to new array

N comparisons (<u>at most</u>)

need 1 comparison for each move

until one array becomes empty; rest can move without a comparison

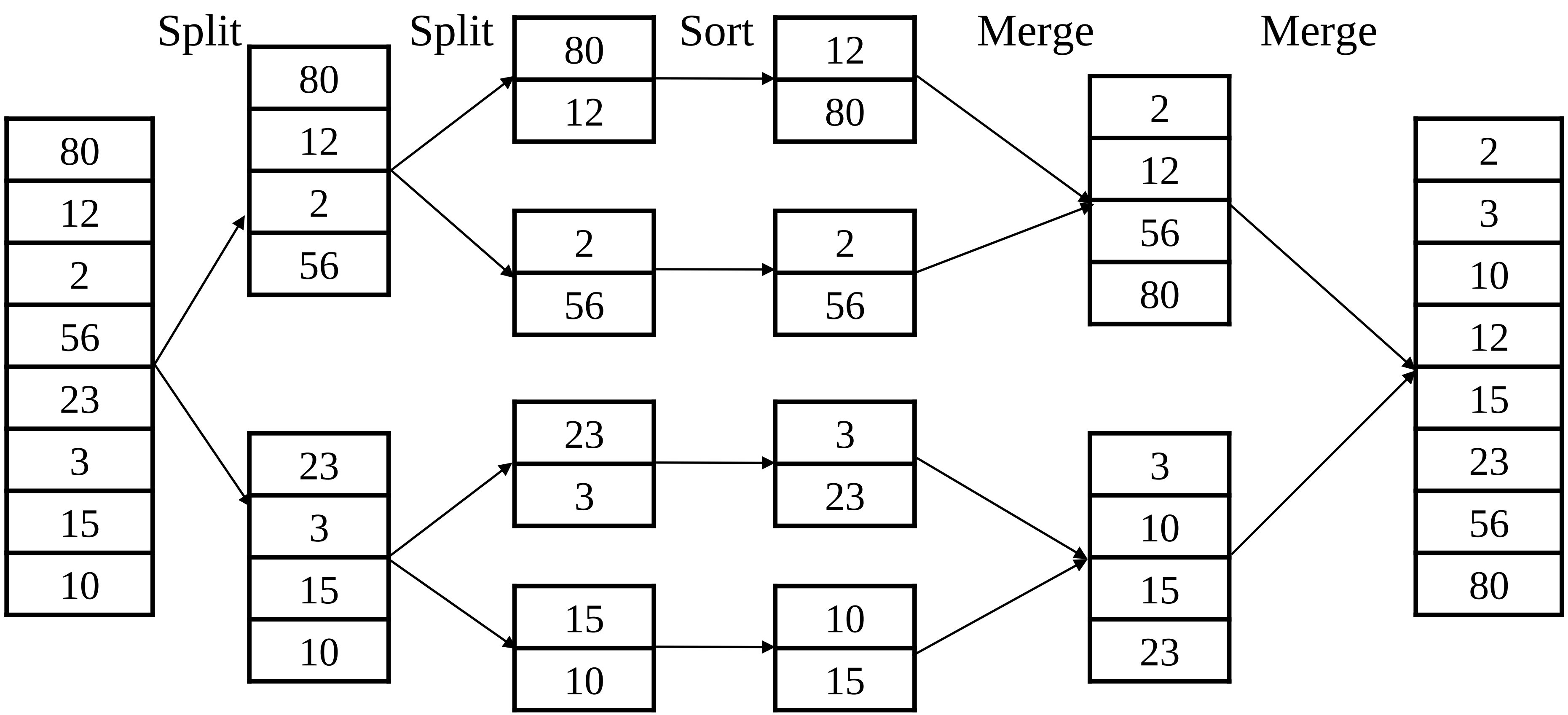
Merge Sort

- Given N items
  - split into 2 groups of N/2 items
  - sort each group
  - merge the 2 groups
- Split is O(N), merge is O(N)
- But what about the *sort* part?
  - Apply *mergeSort* to each group
    - get 4 groups of N/4
  - Recurse again
    - get 16 groups of N/16
  - Recurse until each group has only 1 or 2 elements
    - if 1 element, it is sorted
    - if 2 elements, takes 1 comparison to sort

```
mergeSort( int a[] ):
if a.size == 2
   sort the 2 elements
else if a.size > 2
   a1 = a[0 .. N/2]
   a2 = a[N/2+1 .. N-1]
   mergeSort( a1 )
   mergeSort( a2 )
   int[] c = merge( a1, a2 )
   copy c into a
```

Merge Sort Example

- Three phases: split, sort, merge



Merge Sort Analysis

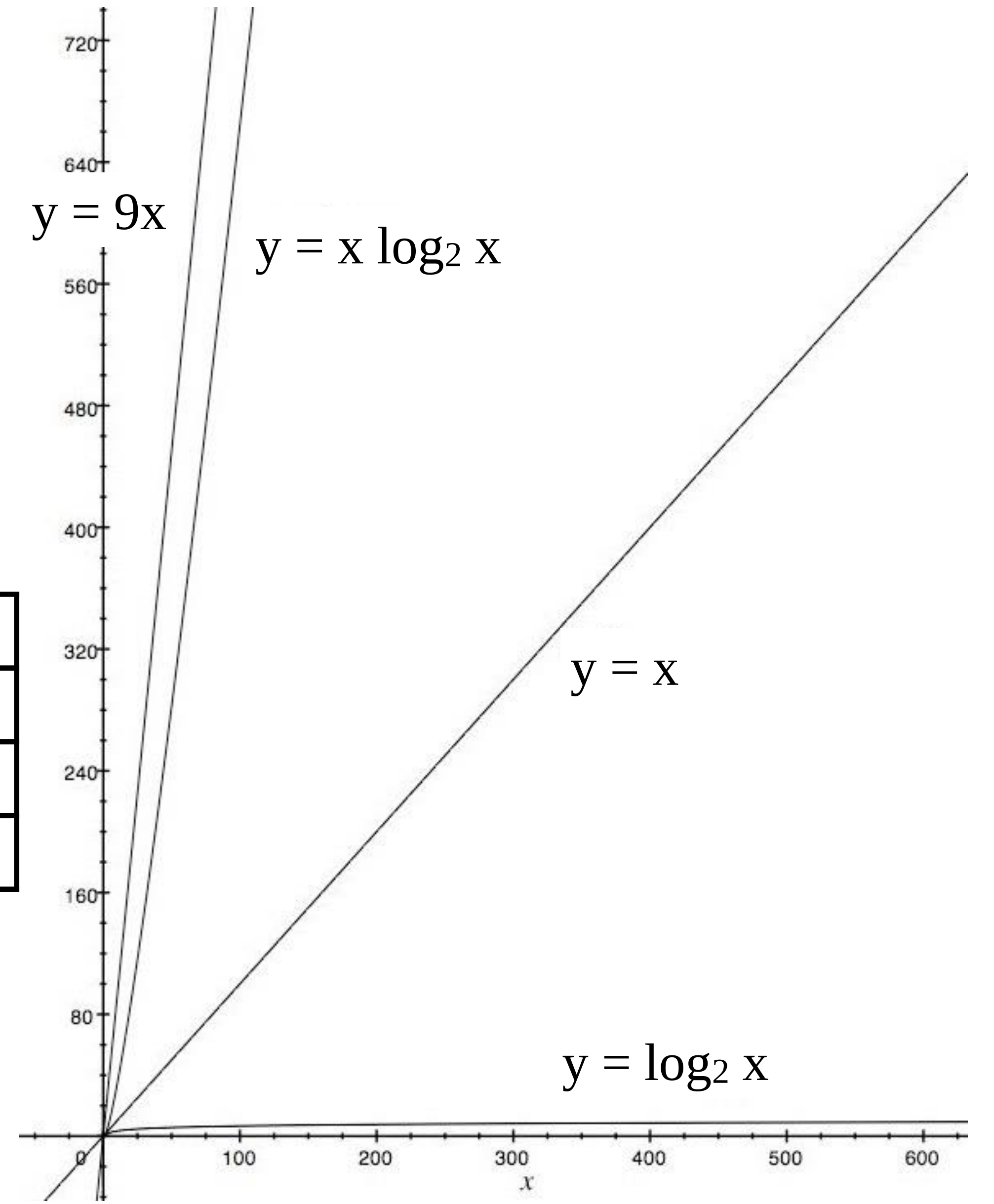
- Split steps: takes $\log_2 N$ steps to get to pairs: $O(\log_2 N)$; each step is $O(N)$, so split complexity is $O(N \log_2 N)$
- Sort steps: there are N/2 pairs, so N/2 comparisons and at most N/2 swaps: $O(N)$
- Merge steps: there are $\log_2 N$ merge steps and each merge step is $O(N)$: $O(N \log_2 N)$
- Overall complexity is: $O(N \log_2 N)$

What is O(N $\log_2$N)?

- $x \log_2 x$ seems to be a pretty steep curve!
  - but it's still less than $y = 9x$ for $x < 512$
- Consider some sample values:

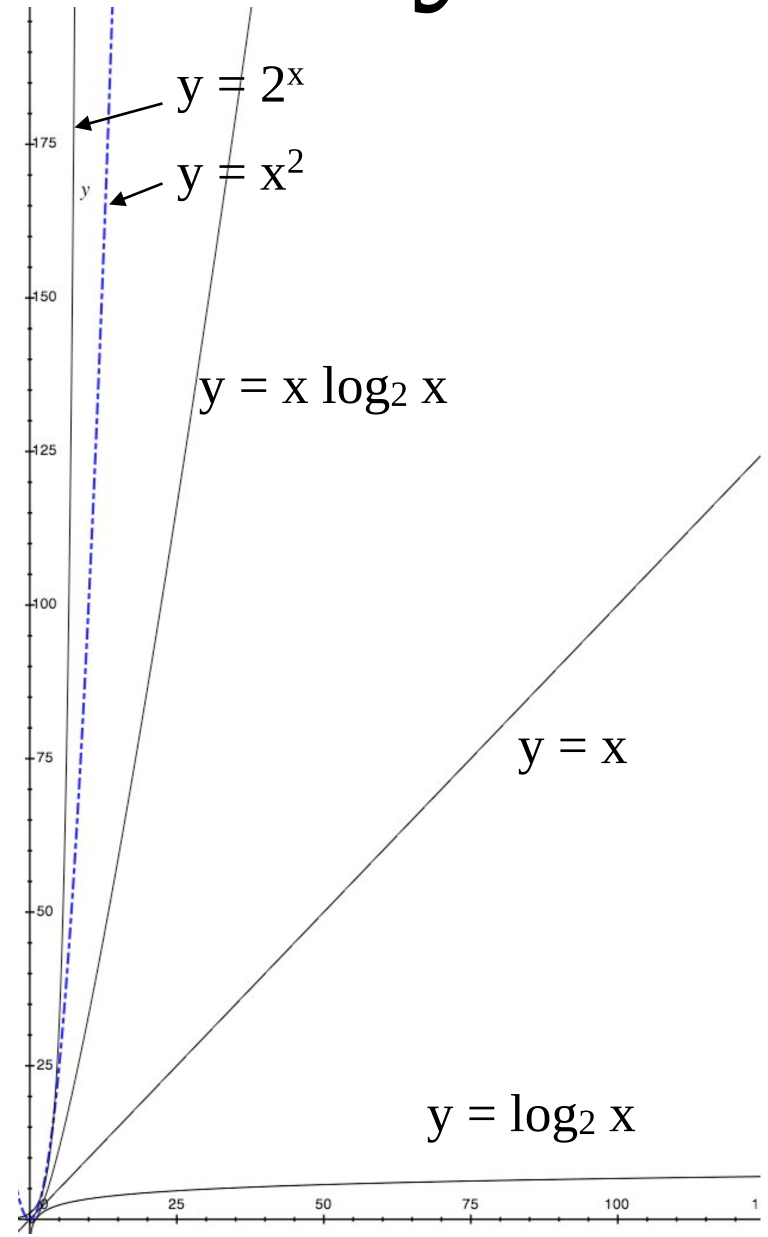
x	256	512	1024	4096	16384
$\log_2 x$	8	9	10	12	14
$x \log_2 x$	2048	4608	10K	49K	0.229M
x^2	65536	262144	1048K	16000K	268M

Clearly, $x \log_2 x$ grows <u>much</u> more slowly than x^2



Growth Rates are Key

- Graphs show how dramatic the time growth can be with exponent-based time (x^2 and 2^x)
  - That's why other terms don't matter
  - x^2 and 2^x don't look much different, but they **are** for big enough x:
    $30^2 = 900$, but
    $2^{30} = 1{,}073{,}741{,}824$
- x $\log_2$ x
  - x = 30, x $\log_2$ x = 147
  - x = 100, x $\log_2$ x = 767
- Remember this graph!



Merge Sort Algorithm

```
sort( int[] arr ):
mergeSort( arr, 0, arr.length - 1 );

mergeSort(int[] a, int lo, int hi):
if ( lo == hi - 1 ) // a pair yet?
  orderPair( a, lo );
else if ( lo != hi ) // recurse
{
  int mid = ( lo + hi ) / 2;
  mergeSort( a, lo, mid );
  mergeSort( a, mid + 1, hi );
  merge( a, lo, hi );
}
orderPair( int[] arr, int lo ):
if ( a[ lo ] > a[ lo + 1 ] )
{
  int tmp = a[ lo ];
  a[ num ] = a[ lo + 1 ];
  a[ lo + 1 ] = temp;
}
```

Just 2 in array, order them

More than 2 in array, split and recurse

Find the midpoint

Split array into 2 parts, sort each part and put them back together

merge Method

```
merge( int[] a, int lo, int hi):
int[] tmp = new int[ hi - lo + 1 ];
int mid = ( lo + hi )/ 2;
int k1 = lo;      // start of one split
int k2 = mid + 1; // start of other
int t = 0;
while ( k1 <= mid && k2 <= hi )
{
  if ( a[ k1 ] < a[ k2 ] )
    tmp[ t++ ] = a[ k1++ ];
  else
    tmp[ t++ ] = a[ k2++ ];
}
while ( k1 <= mid )
  tmp[ t++ ] = a[ k1++ ];
while ( k2 <= hi )
  tmp[ t++ ] = a[ k2++ ];
// copy tmp array back
for ( int i = 0, k = lo; k <= hi; i++, k++ )
  a[ k ] = tmp[ i ];
```

get a temp array to merge into

Set up regions to be merged

quit loop as soon as <u>either</u> subpart runs out

look at “first” of each group and copy smaller to temp

copy any thing left in first partition

copy any thing left in second partition

copy tmp back into the array

Review

- Introduced 3 sorting algorithms
  - bubble sort
  - insertion sort
  - selection sort
- Analyzed their complexity: $O(N^2)$
- Introduced merge sort algorithm
  - Analyzed its complexity $O(N \log_2 N)$

Next

- Programming design and style review
  - Assignment/lab solutions
  - Coding style pointers
- CS 515 Data structures and algorithms
  - More data structures
    - AVL trees, B-trees, and more
  - More algorithms and algorithm complexity
  - C++

Next

- Programming design and style review
  - Assignment/lab solutions
  - Coding style pointers
- CS 515 Data structures and algorithms
  - More data structures
    - AVL trees, B-trees, and more
  - More algorithms and algorithm complexity
  - C++