Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Sorting

- Putting items in any order
- Items need to be "comparable"
- Need to know how to "compare"
- Results of compare need to be "definitive" (YES/NO/EQUAL)

Complexity Measures

- Number of **Comparisons** made
- Number of **Data Movements** made

Inefficient Sorting Algorithms

Selection Sort

- Repeatedly **select** the next smallest item and place it in right location
- **Invariant**: After k iterations first k smallest items are in right location
- In iteration k, find smallest item in locations k .. n and swap it with item in location k
- Time complexity of iteration k is $O(n-k)$
- Total time complexity = $(n-1) + (n-2) + \ldots + 1 = O(n^2)$

Insertion Sort

- Repeatedly **insert** next item into "growing" sorted list
- **Invariant**: After k iterations the first k locations are in sorted order
- In iteration k, insert item in location k into sorted sublist in locations $1 .. k-1$
- Time complexity of iteration k is $O(k)$
- Total time complexity = $1 + 2 + \ldots + (n-2) + (n-1) = O(n^2)$

Figure 8.3
Basic action of insertion sort (the shaded part is sorted)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

Figure 8.4
A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

Insertion Sort

```
public static <AnyType extends Comparable<? super AnyType>>
   void insertionSort( AnyType [ ] a )
   {
       int j;

       for( int p = 1; p < a.length; p++ )
       {
           AnyType tmp = a[ p ];
           for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
               a[ j ] = a[ j - 1 ];
           a[ j ] = tmp;
       }
   }
```

Bubble Sort

- Repeatedly **bubble** smaller items “upward”
- **Invariant**: After k iterations the first k locations are in sorted order
- In iteration k, scan entire list from end comparing adjacent items along the way and swapping if they are out of order
- Time complexity of iteration k is O(n)
- Total time complexity = n (n-1) = $O(n^2)$

ShellSort: Sophisticated Insertion Sort

Idea: Make "sublists" and sort them using insertion sort

Figure 8.5
Shellsort after each pass if the increment sequence is {1, 3, 5}

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

ShellSort

```
public static <AnyType extends Comparable<? super AnyType>>
  void shellsort( AnyType [ ] a )
  {
     int j;

     for( int gap = a.length / 2; gap > 0; gap /= 2 )
        for( int i = gap; i < a.length; i++ )
        {
           AnyType tmp = a[ i ];
           for( j = i; j >= gap &&
                    tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
              a[ j ] = a[ j - gap ];
           a[ j ] = tmp;
        }
  }
```

Improved Sorting Algorithms

Divide and Conquer

- Divide the work into smaller subproblems by partitioning
- Sort each partition separately
- Merge sorted sublists

Merge Sort

```
public static <AnyType extends Comparable<? super AnyType>>
  void mergeSort( AnyType [ ] a ) {
     AnyType [ ] tmpArray = (AnyType[]) new Comparable[ a.length ];
     mergeSort( a, tmpArray, 0, a.length - 1 );
  }

private static <AnyType extends Comparable<? super AnyType>>
  void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
           int left, int right )
  {
     if( left < right )
     {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
     }
  }
```

Majority of work happens here in merge.

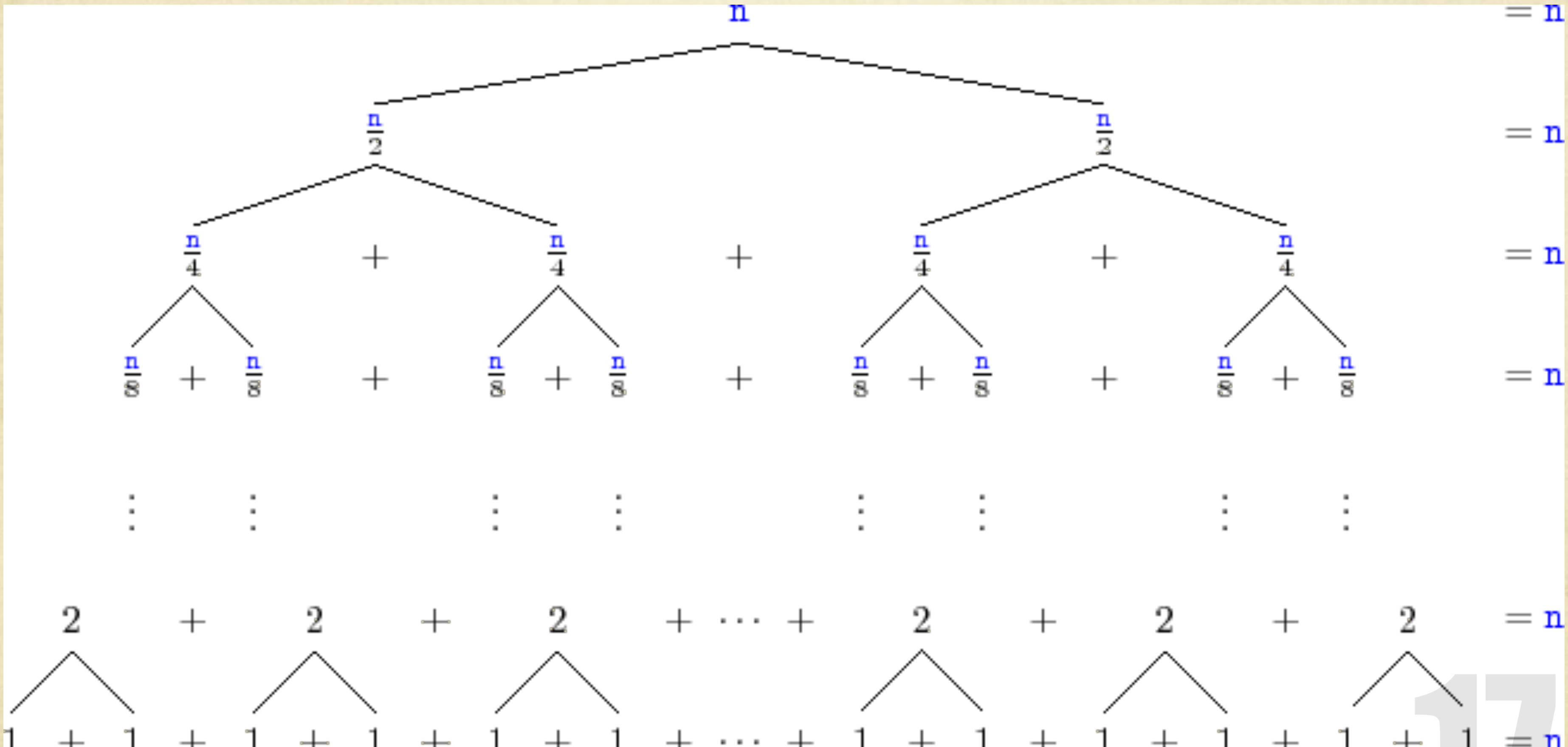
Merge in Merge Sort

```
private static <AnyType extends Comparable<? super AnyType>>
void merge( AnyType[ ] a, AnyType[ ] tmpArray, int leftPos, int rightPos, int rightEnd )
  {
     int leftEnd = rightPos - 1;
     int tmpPos = leftPos;
     int numElements = rightEnd - leftPos + 1;
     while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ].compareTo( a[ rightPos ] ) < 0 )
                 tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
                  tmpArray[ tmpPos++ ] = a[ rightPos++ ];
     while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];
     while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
  }
```

Time Complexity:
$O(L_a + L_b)$

Analysis: Recursion Tree



http://opendatastructures.org/versions/edition-0.1c/ods-java/img1231.png

Alternative Analysis

- Let T(n) = time complexity of MergeSort on list with n elements
- We know that time complexity of merge operation on two sorted lists of total length n is O(n)
- We can write a recurrence relationship as follows:
  - $T(n) = T(n/2) + T(n/2) + O(n)$

$T(n) = 2T(n/2) + O(n)$

- Expansion Method:

$T(n) = 2T(n/2) + O(n)$

$T(n) \le 2T(n/2) + cn$

$T(n) \le 2(2T(n/4) + cn/2) + cn$

$T(n) \le 4T(n/4) + 2cn$

$T(n) \le 8T(n/8) + 3cn$

$T(n) \le 4(2T(n/8) + cn/4) + 2cn$

$T(n) \le 8T(n/8) + 3cn$

…

$T(n) \le nT(1) + (\log n)\, cn$

$T(n) = O(n \log n)$

Use "Guestimations"

- Guess that $T(n) = O(n) \leq cn$
  - Then we know that $T(n/2) = cn/2$
  - Right side $= 2(cn/2) + c_1n = cn + c_1n$
  - Since both c and $c_1 > 0$, we cannot make right side $\leq cn$
  - Failure!
- Guess that $T(n) = O(n^2) \leq cn^2$
  - Then we know that $T(n/2) \leq c(n/2)^2$
  - Right side $= 2(cn^2/4) + c_1n = cn^2/2 + c_1n$
  - We can make right side $\leq cn^2$ by choosing c large enough
  - But was our choice too liberal?

Use “Guestimations” … 2

- Guess that $T(n) = O(n \log n) \le c\,(n \log n)$
  - Then we know that $T(n/2) \le c(n/2) \log (n/2)$
  - Right side
    - $= 2(c(n/2) \log (n/2)) + c_1 n$
    - $= cn (\log n - 1) + c_1 n$
    - $= c\,(n \log n) + (c_1 - c)n$
  - We can make right side $\le c\,(n \log n)$ by choosing $c > c_1$
- Thus $T(n) = O(n \log n)$ is the best solution among three choices

More general recurrences

- $T(n) = a\,T(n/b) + f(n)$
  - $T(n) = O(n \log n)$, if $a = b$ and $f(n) = \Theta(n)$
  - $T(n) = \Theta(n^{\{\log_b a\}})$, if $f(n) = O(n^{\{\log_b a - \varepsilon\}})$

Important Case

HeapSort

- Discussed earlier
- Time Complexity = $O(n \log n)$

QuickSort

- Carefully divide, then conquer
- First partition into **Small** and **Large** sets, then call recursively on each of the sets and concatenate two sorted sublists
- Since we partition first, “merge” is not necessary; only need to concatenate two sorted lists

Figure 8.10 Quicksort

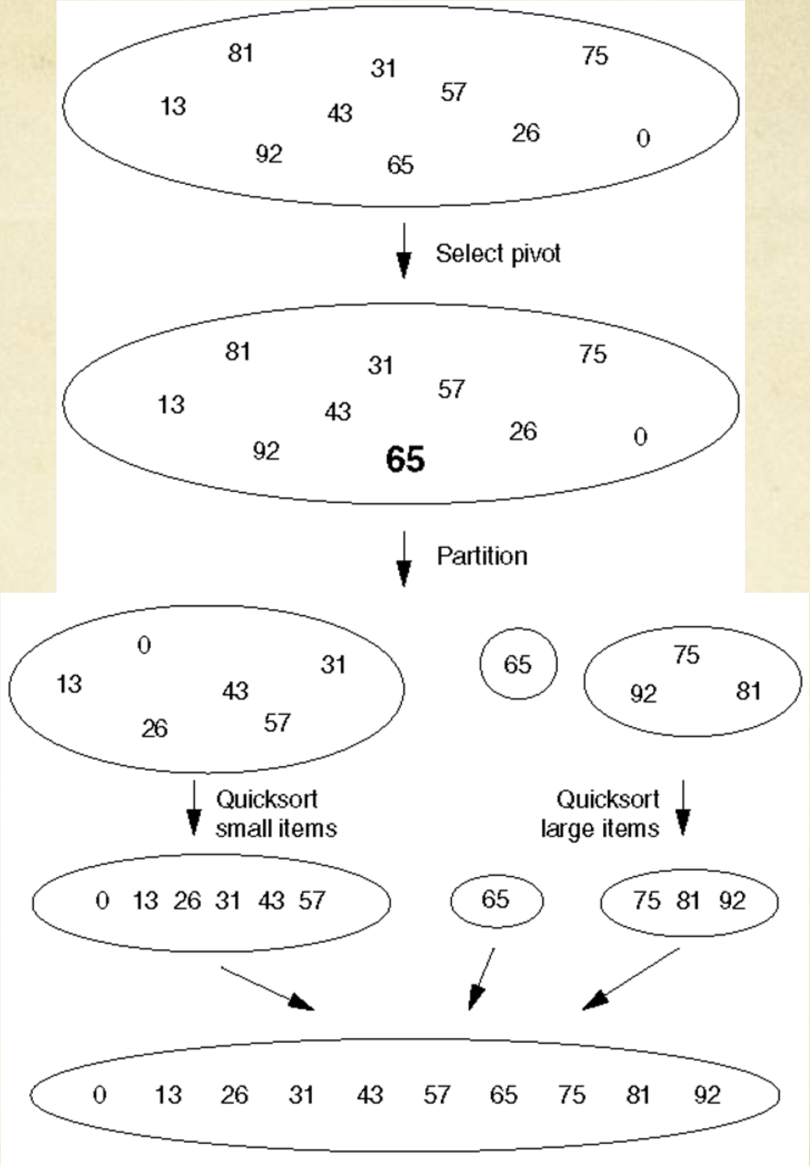


Figure 8.11 Partitioning algorithm: Pivot element 6 is placed at the end.

8	1	4	9	0	3	5	2	7	6

Figure 8.12 Partitioning algorithm: i stops at large element 8; j stops at small element 2.

8	1	4	9	0	3	5	2	7	6

Figure 8.13 Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

2	1	4	9	0	3	5	8	7	6

Figure 8.14 Partitioning algorithm: i stops at large element 9; j stops at small element 5.

2	1	4	9	0	3	5	8	7	6

Figure 8.15 Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

2	1	4	5	0	3	9	8	7	6

Figure 8.16 Partitioning algorithm: i stops at large element 9; j stops at small element 3.

2	1	4	5	0	3	9	8	7	6

Figure 8.17 Partitioning algorithm: Swap pivot and element in position i.

2	1	4	5	0	3	6	8	7	9

Figure 8.18 Original array

8	1	4	9	6	3	5	2	7	0

Figure 8.19 Result of sorting three elements (first, middle, and last)

0	1	4	9	6	3	5	2	7	8

Figure 8.20 Result of swapping the pivot with the next-to-last element

0	1	4	9	7	3	5	2	6	8

Quicksort

```
public static <AnyType extends Comparable<? super AnyType>> void quicksort(AnyType [ ] a
    {  quicksort( a, 0, a.length - 1 );  }


private static <AnyType extends Comparable<? super AnyType>>
void quicksort(AnyType [ ] a, int left, int right ) {
    if( left + CUTOFF <= right ) {
        AnyType pivot = median3( a, left, right );

        // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; ) {
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j )
                swapReferences( a, i, j );
            else
                break;
        }
        swapReferences( a, i, right - 1 );   // Restore pivot
        quicksort( a, left, i - 1 );    // Sort small elements
        quicksort( a, i + 1, right );   // Sort large elements
    }
    else  // Do an insertion sort on the subarray
        insertionSort( a, left, right );
  }
```

Upper and Lower Bounds

- Upper bound on time complexity of sorting is $O(n \log n)$, because there exists at least one algorithm that runs in time $O(n \log n)$ in the worst case.
- But is this the best possible?
- Lower bound on the time complexity of a problem is $T(n)$ if $\forall$ algorithms that solve the problem, their time complexity is $\Omega(T(n))$.
- It can be mathematically proved that lower bound for sorting is $\Omega(n \log n)$.
- Thus Merge Sort and Heap Sort are optimal.

Special Sorting Algorithms

Bucket Sort

- N **integer** values in the range [a..a+m-1]
- For e.g., sort a list of 50 scores in the range [0..9].
- Algorithm
  - Make m buckets [a..a+m-1]
  - As you read elements throw into appropriate bucket
  - Output contents of buckets [0..m] in that order
- Time O(N+m)
- **Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

Stable Sort

- A sorting algorithm is stable if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!

Radix Sort

3	5	9
3	5	7
3	5	1
7	3	9
3	3	6
7	2	0
3	5	5

→

3	5	9
3	5	7
3	5	1
3	3	6
3	5	5
7	3	9
7	2	0

→

3	3	6
3	5	9
3	5	7
3	5	1
3	5	5
7	2	0
7	3	9

→

3	3	6
3	5	1
3	5	5
3	5	7
3	5	9
7	2	0
8	3	9

Algorithm

for i = 1 **to** d **do**

  sort array A on digit i using any sorting algorithm

Time Complexity: $O((N+m) + (N+m^2) + \ldots + (N+m^d))$

Space Complexity: $O(m^d)$

Improved Radix Sort

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Algorithm

for i = d **to** 1 **do**

  sort array A on digit i using a stable sort algorithm

Time Complexity: O((n+m)d)

•**Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

Counting Sort

Initial Array

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

Counts

0	1	2	3	4	5
2	0	2	3	0	1

Cumulative Counts

0	1	2	3	4	5
2	2	4	7	7	8

Time Complexity: O(n+C)

•**Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

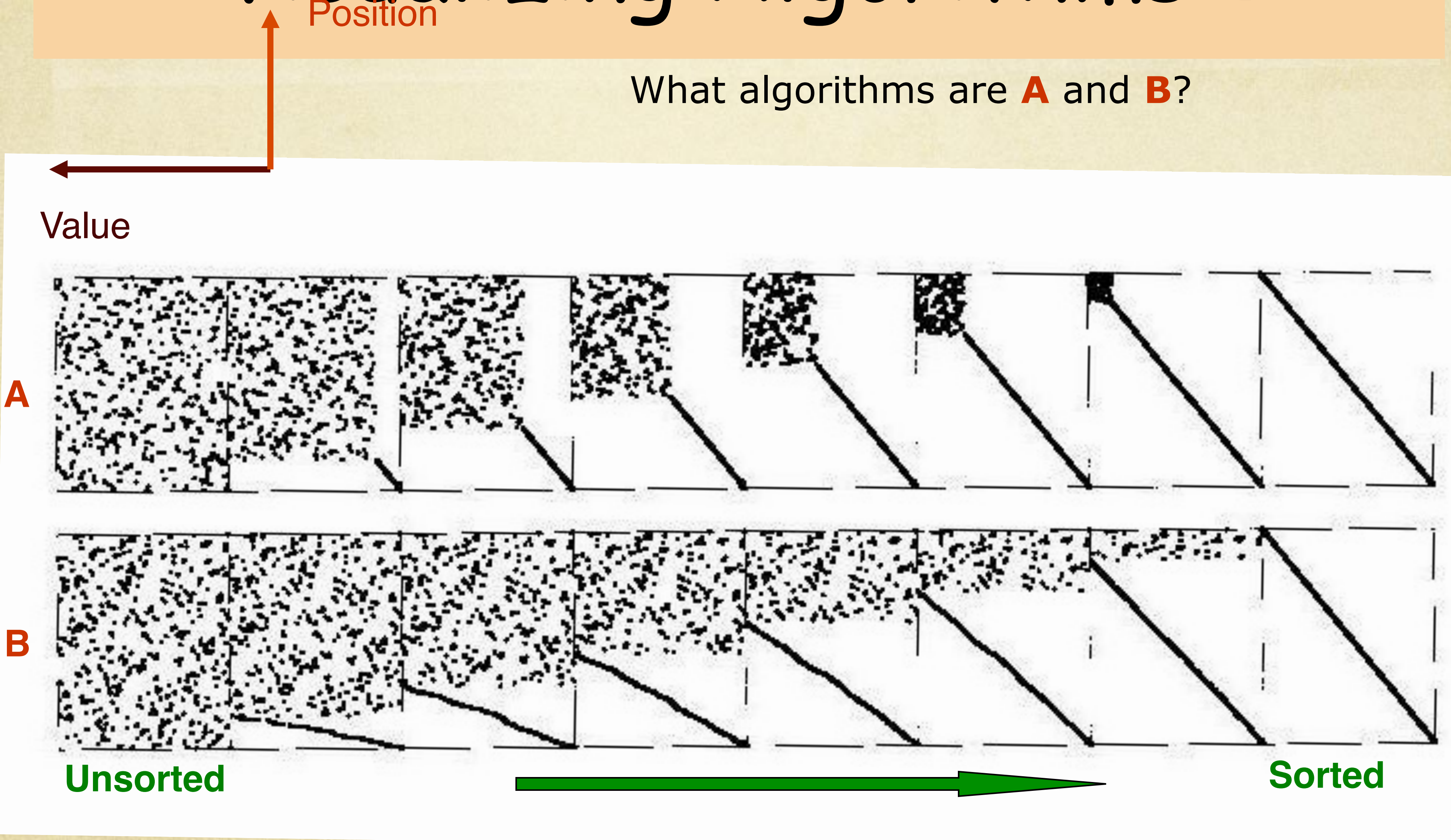
Sorting Algorithms Summary

- $O(n^2)$ sorting algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort & Shaker Sort
- $O(n^2)$ sorting algorithms, but $O(n \log n)$ on average
  - Quick Sort
- $O(n^{2-})$ sorting algorithms
  - Shell Sort
- $O(n \log n)$ sorting algorithms
  - Merge Sort
  - Heap Sort
- $O(n)$ specialized sorting algorithms
  - Bucket and Radix Sort
  - Counting Sort

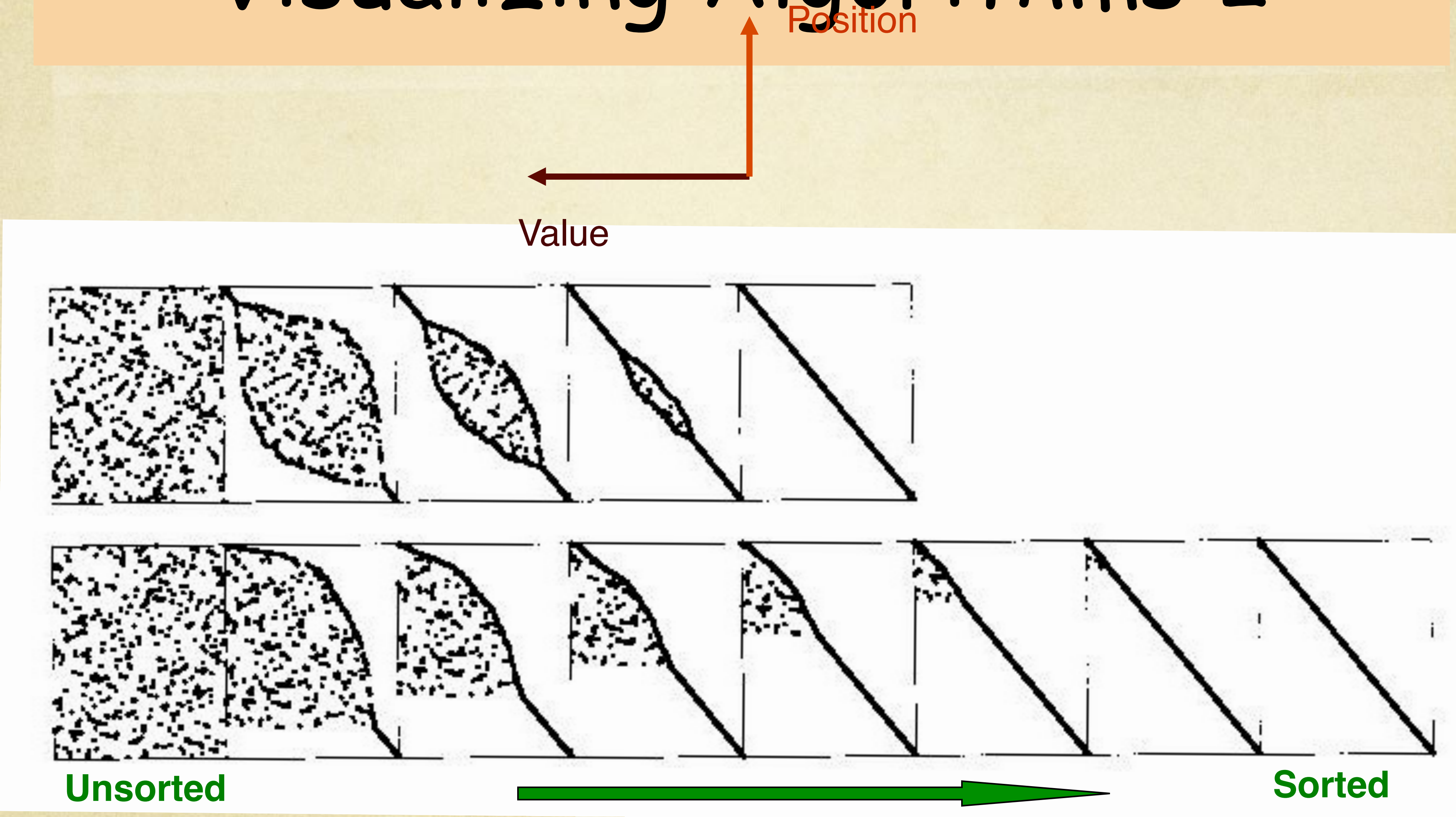
Visualizing Sorting

Visualizing Algorithms 1

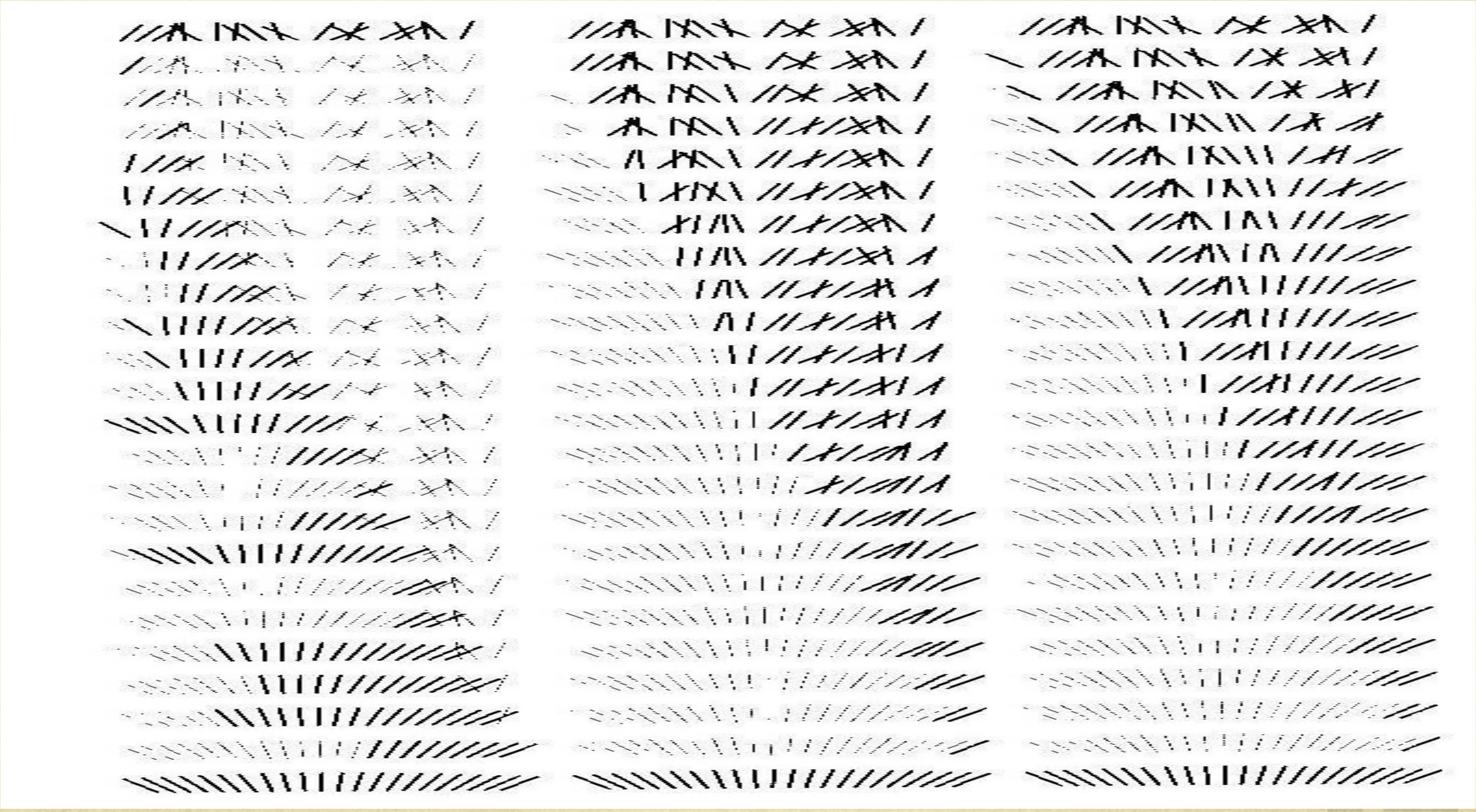
What algorithms are **A** and **B**?



Visualizing Algorithms 2



Visualizing Comparisons 3



Animations (Not sure which work)

- http://cg.scs.carleton.ca/~morin/misc/sortalg/
- http://home.westman.wave.ca/~rhenry/sort/
- time complexities on best, worst and average case
- http://vision.bc.edu/~dmartin/teaching/sorting/anim-html/quick3.html
- runs on almost sorted, reverse, random, and unique inputs; shows code with invariants
- http://www.brian-borowski.com/Sorting/
- comparisons, movements & stepwise animations with user data
- http://maven.smith.edu/~thiebaut/java/sort/demo.html
- comparisons & data movements and step by step execution

Optional Topics

- Lower Bound for Sorting
  - Needs to decide which of **n!** perms is the right answer
  - Each comparison can only separate two subsets and the whole process requires $\Omega(\log(n!))$
  - $\Omega(n \log n)$
- External Sorting Algorithms

External Sorting Methods

- Assumptions:
  - data is too large to be held in main memory;
  - data is read or written in blocks;
  - 1 or more external devices available for sorting
- Sorting in main memory is cheap or free
- Read/write costs are the dominant cost
- Wide variety of storage types and costs
- No single strategy works for all cases

External Merge Sort

- Initial distribution pass
- Several multi-way merging passes

ASORTINGANDMERGINGEXAMPLEWITHFORTYFIVERECORDS.$

AOS.DMN.AEX.FHT.ERV.$
IRT.EGR.LMP.ORT.CEO.$
AGN.GIN.EIW.FIY.DRS.$

AAGINORST.FFHIORTTY.$
DEGGIMNNR.CDEEORRSV.$
AEEILMPWX.$

AAADEEEGGGIIILMMNNNOPRRSTWX.$
CDEEFFHIOORRRSTTVY.$

AAACDDEEEEEEFFGGGHIIIILMMNNNOO

With 2P external devices
Space for M records in main memory
Sorting N records needs
$1 + \log_P(N/M)$ passes