CSCI 104
Sorting Algorithms

Mark Redekopp
David Kempe
Sandra Batista

Algorithm Efficiency

SORTING

Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Sorting provides a "classical" study of algorithm analysis because there are many implementations with different pros and cons

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

Original

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

Sorted

Applications of Sorting

- Find the set_intersection of the 2 lists to the right
  - How long does it take?

A

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

B

9	3	4	2	7	8	11
0	1	2	3	4	5	6

Unsorted

- Try again now that the lists are sorted
  - How long does it take?

A

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

B

2	3	4	7	8	9	11
0	1	2	3	4	5	6

Sorted

Sorting Stability

- A sort is stable if the order of equal items in the original list is maintained in the sorted list
  - Good for searching with multiple criteria
  - Example: Spreadsheet search
    - List of students in alphabetical order first
    - Then sort based on test score
    - I'd want student's with the same test score to appear in alphabetical order still
- As we introduce you to certain sort algorithms consider if they are stable or not

List	7,a	3,b	5,e	8,c	5,d
index	0	1	2	3	4

Original

List	3,b	5,e	5,d	7,a	8,c
index	0	1	2	3	4

Stable Sorting

List	3,b	5,d	5,e	7,a	8,c
index	0	1	2	3	4

Unstable Sorting

Bubble Sorting

- Main Idea: Keep comparing neighbors, moving larger item up and smaller item down until largest item is at the top. Repeat on list of size n-1
- Have one loop to count each pass, (a.k.a. i) to identify which index we need to stop at
- Have an inner loop start at the lowest index and count up to the stopping location comparing neighboring elements and advancing the larger of the neighbors

List	7	3	8	6	5	1	Original
List	3	7	6	5	1	8	After Pass 1
List	3	6	5	1	7	8	After Pass 2
List	3	5	1	6	7	8	After Pass 3
List	3	1	5	6	7	8	After Pass 4
List	1	3	5	6	7	8	After Pass 5

Bubble Sort Algorithm

```
void bsort(vector<int>& mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
          swap(mylist[j], mylist[j+1]);
  }  }  }
}
```

Pass 1

7	3	8	6	5	1	
j					i	
3	7	8	6	5	1	swap
	j				i	
3	7	8	6	5	1	no swap
		j			i	
3	7	6	8	5	1	swap
			j		i	
3	7	6	5	8	1	swap
				j	i	
3	7	6	5	1	8	swap

Pass 2

3	7	6	5	1	8	
j				i		
3	7	6	5	1	8	no swap
	j			i		
3	6	7	5	1	8	swap
		j		i		
3	6	5	7	1	8	swap
			j	i		
3	6	5	1	7	8	swap

…

Pass n-2

3	1	5	6	7	8	
j	i					
1	3	5	6	7	8	swap

Bubble Sort

Courtesy of wikipedia.org

Bubble Sort Analysis

- Best Case Complexity:
  - When already ________________ but still have to ________________
  - O(____)
- Worst Case Complexity:
  - When ___________________________
  - O(____)

```
void bsort(vector<int>& mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
          swap(mylist[j], mylist[j+1]);
  }  }  }
}
```

Bubble Sort Analysis

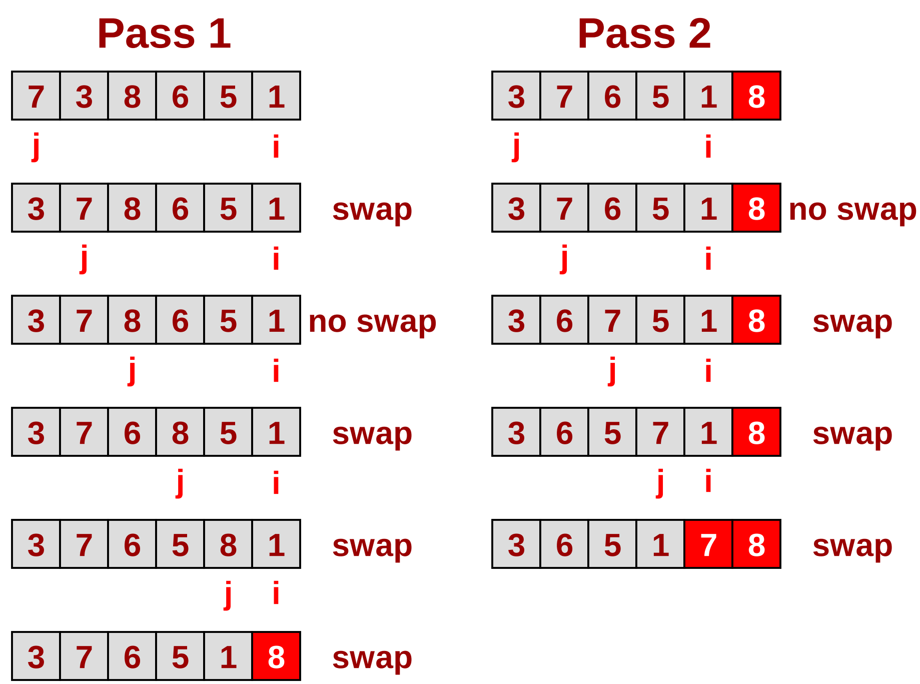
- Best Case Complexity:
  - When already sorted (no swaps) but still have to do all compares
  - $O(n^2)$
- Worst Case Complexity:
  - When sorted in descending order
  - $O(n^2)$

```
void bsort(vector<int>& mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(mylist[j], mylist[j+1]);
  }  }  }
}
```

Loop Invariants

- Loop invariant is a statement about what is true either before an iteration begins or after one ends
- Consider bubble sort and look at the data after each iteration (pass)
  - What can we say about the patterns of data after the k-th iteration?

```
void bsort(vector<int>& mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(mylist[j], mylist[j+1]);
  }  }  }
}
```



Loop Invariants

- What is true after the k-th iteration?
- All data at indices n-k and above ____________
  - $\forall i, i \geq n - k$:
- All data at indices below n-k are ________________ ______________________
  - $\forall i, i < n - k$:

```
void bsort(vector<int>& mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(mylist[j], mylist[j+1]);
  }  }  }
}
```

Pass 1

7	3	8	6	5	1	
j					i	
3	7	8	6	5	1	swap
	j				i	
3	7	8	6	5	1	no swap
		j			i	
3	7	6	8	5	1	swap
			j		i	
3	7	6	5	8	1	swap
				j	i	
3	7	6	5	1	8	swap

Pass 2

3	7	6	5	1	8	
j				i		
3	7	6	5	1	8	no swap
	j			i		
3	6	7	5	1	8	swap
		j		i		
3	6	5	7	1	8	swap
			j	i		
3	6	5	1	7	8	swap

Loop Invariants

- What is true after the k-th iteration?
- All data at indices n-k and above are sorted
  - $\forall i, i \geq n-k$: $a[i] \leq a[i+1]$
- All data at indices below n-k are less than the value at n-k
  - $\forall i, i < n-k$: $a[i] \leq a[n-k]$

```
void bsort(vector<int>& mylist)
{
   int i ;
   for(i=mylist.size()-1; i > 0; i--){
      for(j=0; j < i; j++){
         if(mylist[j] > mylist[j+1]) {
            swap(mylist[j], mylist[j+1]);
   }  }  }
}
```

Pass 1

7	3	8	6	5	1	
j					i	
3	7	8	6	5	1	swap
	j				i	
3	7	8	6	5	1	no swap
		j			i	
3	7	6	8	5	1	swap
			j		i	
3	7	6	5	8	1	swap
				j	i	
3	7	6	5	1	8	swap

Pass 2

3	7	6	5	1	8	
j				i		
3	7	6	5	1	8	no swap
	j			i		
3	6	7	5	1	8	swap
		j		i		
3	6	5	7	1	8	swap
			j	i		
3	6	5	1	7	8	swap

Selection Sort

- Selection sort does away with the many swaps and just records where the min or max value is and performs one swap at the end
- The list/array can again be thought of in two parts
  - Sorted
  - Unsorted
- The problem starts with the whole array unsorted and slowly the sorted portion grows
- We could find the max and put it at the end of the list or we could find the min and put it at the start of the list
  - Just for variation let's choose the min approach

Selection Sort Algorithm

```
void ssort(vector<int>& mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j;
     }  }
     swap(mylist[i], mylist[min]);
}
```

Note: One can choose to find the min and place at the start of the list or max and place at the end.

Pass 1 min=0

7	3	8	6	5	1	min=1
i	j					
7	3	8	6	5	1	min=1
i		j				
7	3	8	6	5	1	min=1
i			j			
7	3	8	6	5	1	min=1
i				j		
7	3	8	6	5	1	min=5
i					j	
1	3	8	6	5	7	swap

Pass 2 min=1

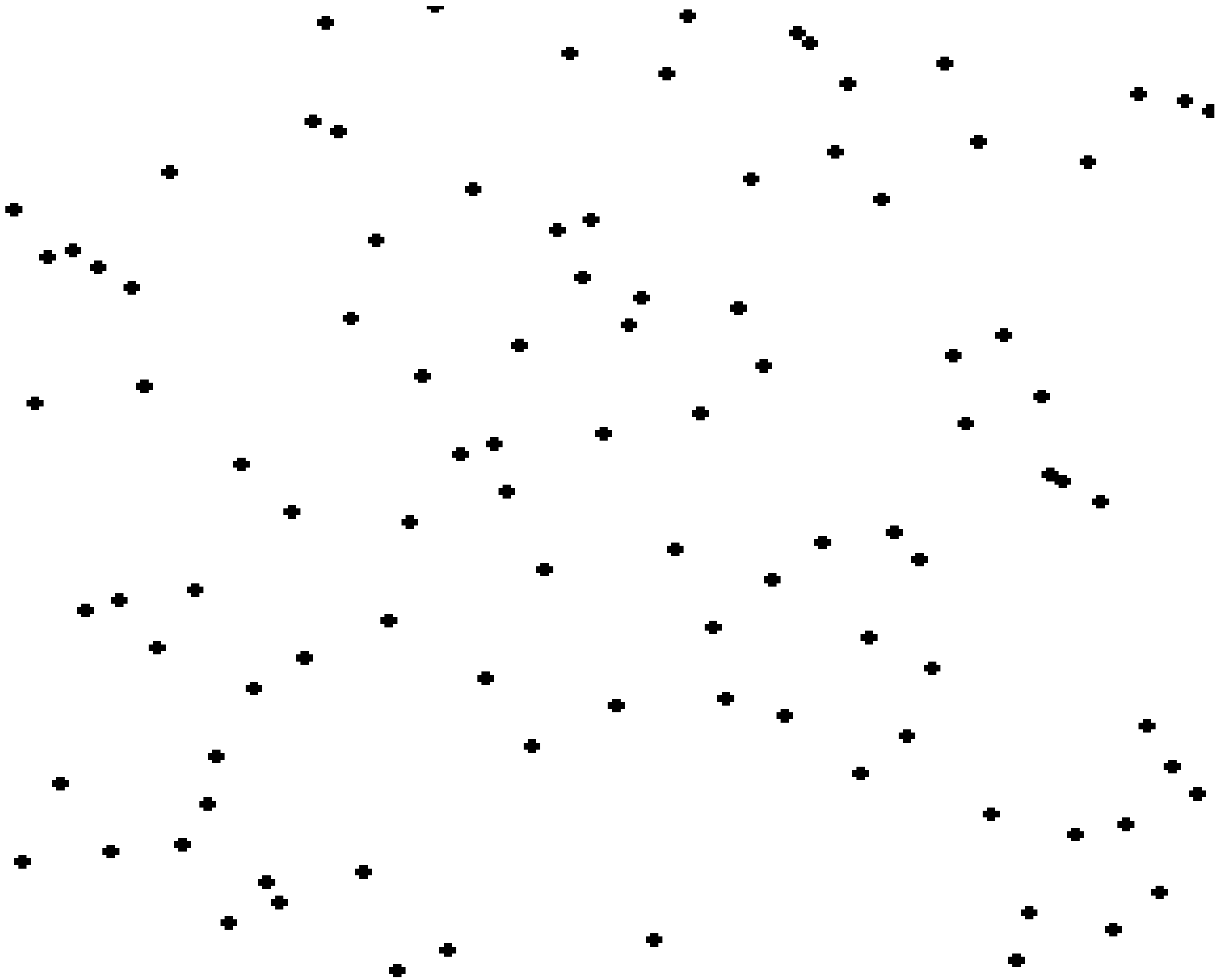
1	3	8	6	5	7	min=1
	i	j				
1	3	8	6	5	7	min=1
	i		j			
1	3	8	6	5	7	min=1
	i			j		
1	3	8	6	5	7	min=1
	i				j	
1	3	8	6	5	7	swap

Pass n-2 min=4

1	3	5	6	7	8	min=4
				i	j	

Selection Sort

Value



List Index

Courtesy of wikipedia.org

Selection Sort Analysis

- Best Case Complexity:
  - ____________________
  - O(___)
- Worst Case Complexity:
  - ______________________
  - O(___)

```
void ssort(vector<int>& mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j;
     }  }
     swap(mylist[i], mylist[min]);
}
```

Selection Sort Analysis

- Best Case Complexity:
  - Sorted already
  - $O(n^2)$
- Worst Case Complexity:
  - When sorted in descending order
  - $O(n^2)$

Note: a+b+c = c+b+a (the order you sum doesn't matter so we can perform reordering or substitutions for our summations)

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \theta(1) = \sum_{i=0}^{n-2} \theta(n-i-1)$$

When i=0, we sum Θ(n-1). When i=n-2, we sum Θ(1). Thus,

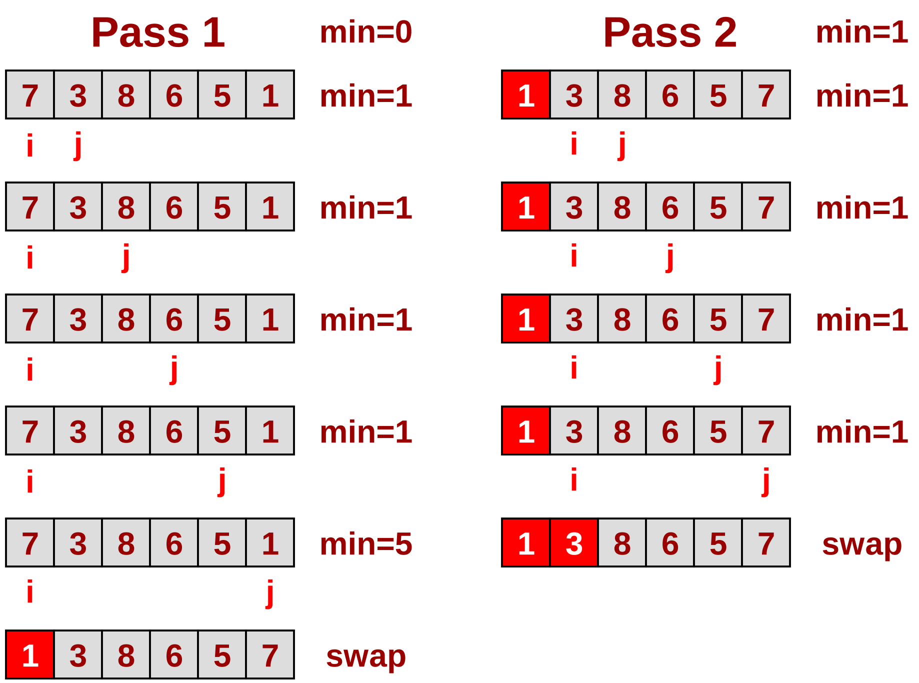
$$= \sum_{k=1}^{n-1} \theta(k) = \theta(n^2)$$

```
void ssort(vector<int>& mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j;
     }  }
     swap(mylist[i], mylist[min]);
}
```

Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than k are ________________
  - $\forall i, i < k$:
- All data at indices k and above are ______________ ______________________
  - $\forall i, i \geq k$:

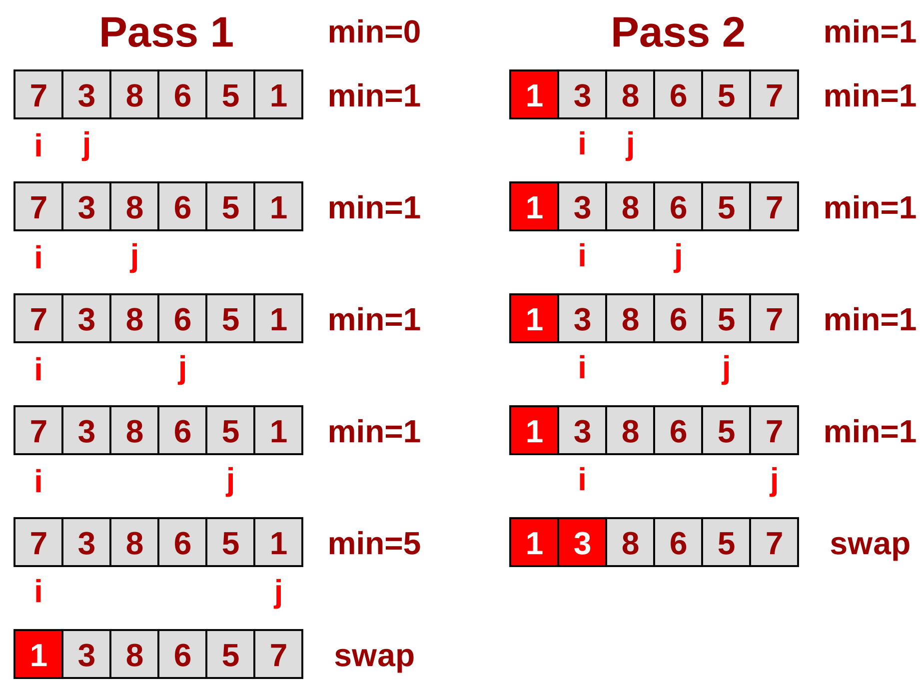
```
void ssort(vector<int>& mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j;
     }  }
     swap(mylist[i], mylist[min]);
}
```



Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than k are sorted
  - $\forall i, i < k$: $a[i] \leq a[i+1]$
- All data at indices k and above are greater than the value at k-1
  - $\forall i, i \geq k$: $a[k-1] \leq a[i]$

```
void ssort(vector<int>& mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j
     }  }
     swap(mylist[i], mylist[min])
}
```



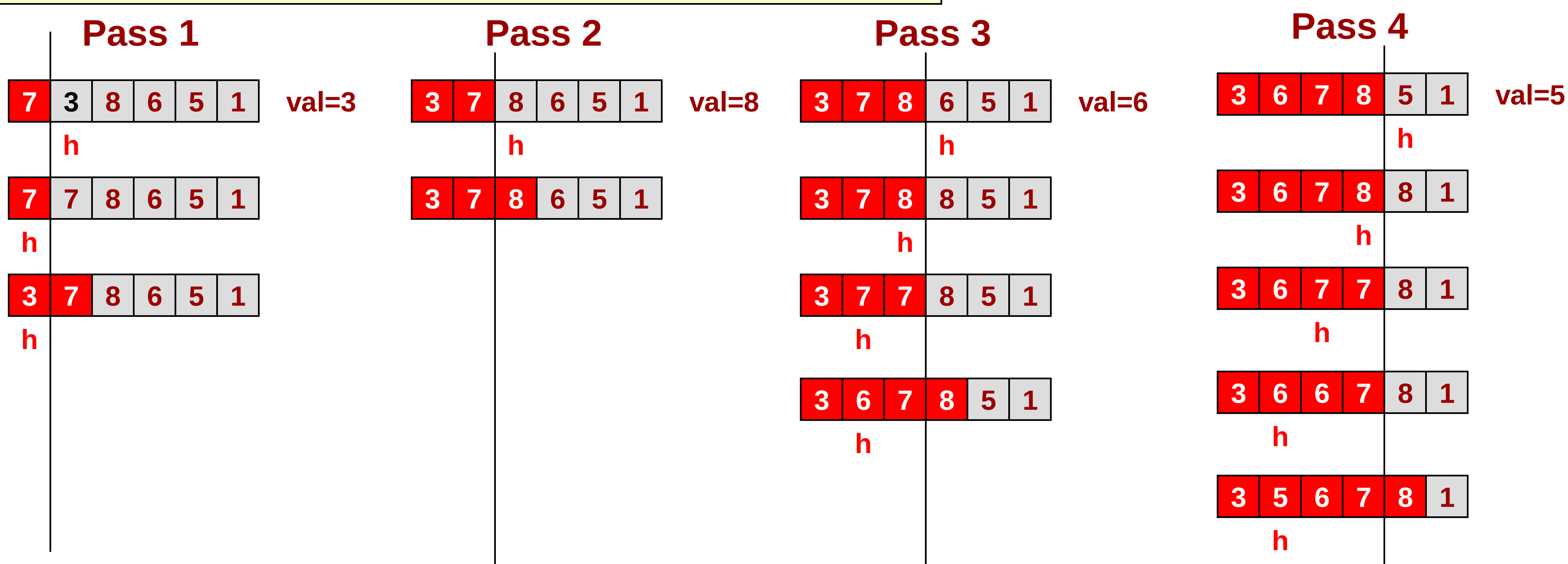
Insertion Sort Algorithm

- Imagine we pick up one element of the array at a time and then just insert it into the right position
- Similar to how you sort a hand of cards in a card game
  - You pick up the first (it is by nature sorted)
  - You pick up the second and insert it at the right position, etc.

Start	1st Card	2nd Card	3rd Card	4th Card	5th Card
? ? ? ? ?	7 ? ? ? ?	7 3 ? ? ?	3 7 8 ? ?	3 7 8 6 ?	3 6 7 8 5
		3 7 ? ? ?	3 7 8 ? ?	3 6 7 8 ?	3 5 6 7 8

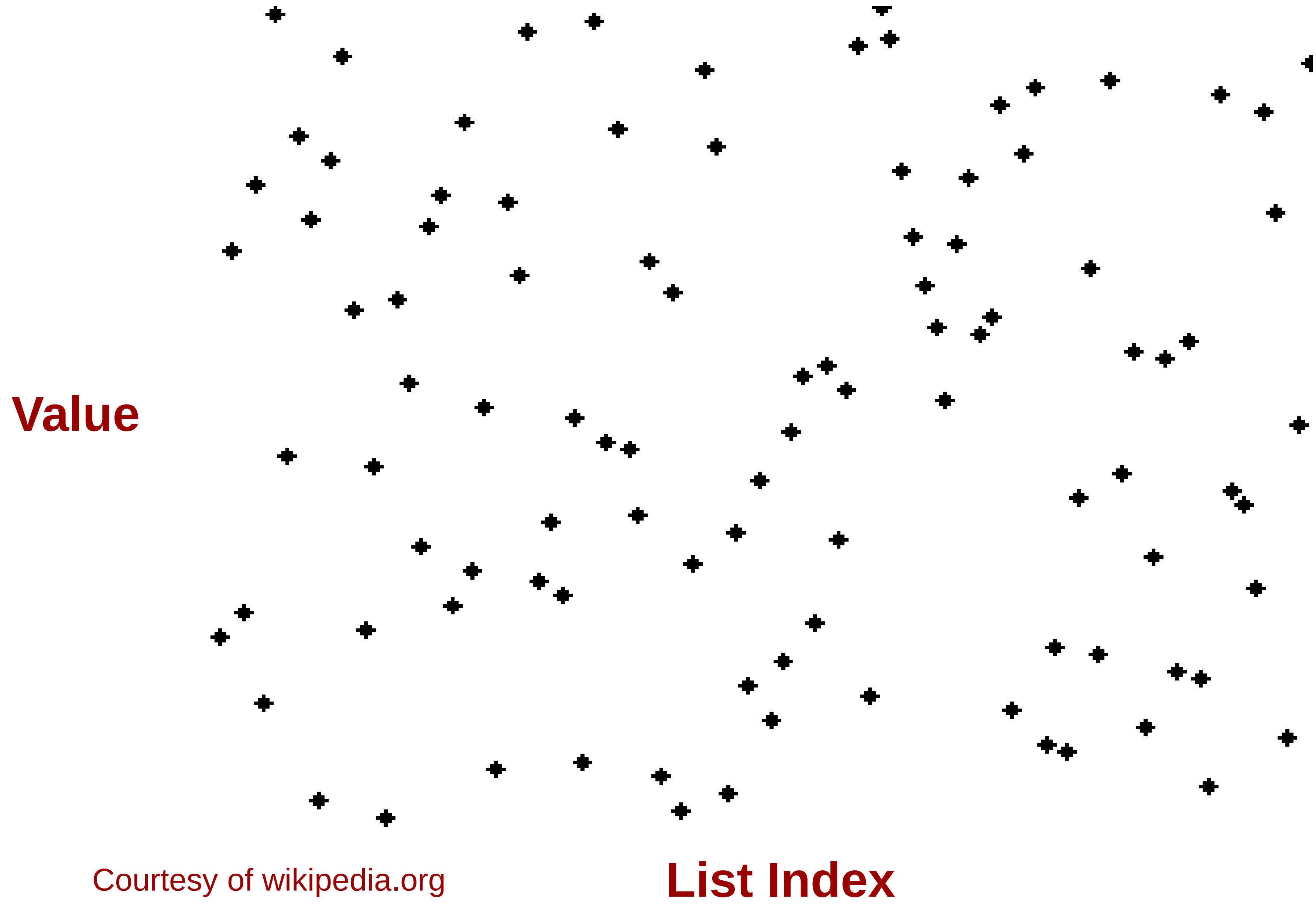
Insertion Sort Algorithm

```
void isort(vector<int>& mylist)
{  for(int i=1; i < mylist.size(); i++){
      int val = mylist[i];
      hole = i;
      while(hole > 0 && val < mylist[hole-1]){
         mylist[hole] = mylist[hole-1];
         hole--;
      }
      mylist[hole] = val;
}  }
```



h = hole

Insertion Sort



Courtesy of wikipedia.org

Insertion Sort Analysis

- Best Case Complexity:
  - Sorted already
  - ___________________
- Worst Case Complexity:
  - When sorted in descending order
  - ___________________

```
void isort(vector<int>& mylist)
{  for(int i=1; i < mylist.size()-1; i++){
      int val = mylist[i];
      hole = i;
      while(hole > 0 && val < mylist[hole-1]){
         mylist[hole] = mylist[hole-1];
         hole--;
      }
      mylist[hole] = val;
}  }
```

Insertion Sort Analysis

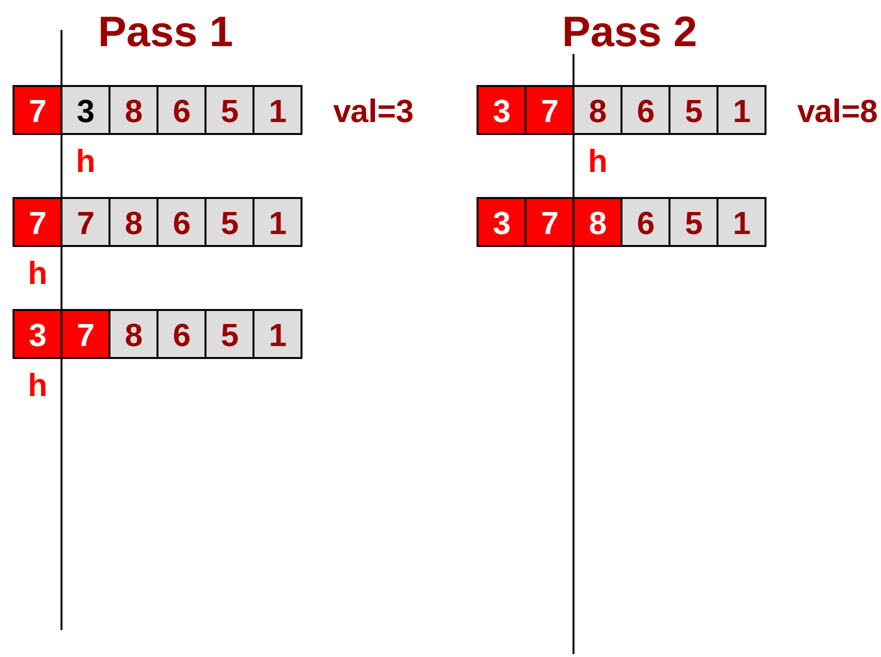
- Best Case Complexity:
  - Sorted already
  - O(n)
- Worst Case Complexity:
  - When sorted in descending order
  - O(n^2)

```
void isort(vector<int>& mylist)
{  for(int i=1; i < mylist.size()-1; i++){
      int val = mylist[i];
      hole = i;
      while(hole > 0 && val < mylist[hole-1]){
         mylist[hole] = mylist[hole-1];
         hole--;
      }
      mylist[hole] = val;
}  }
```

Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than ________________
  - $\forall i,$
- Can we make a claim about data at k+1 and beyond?

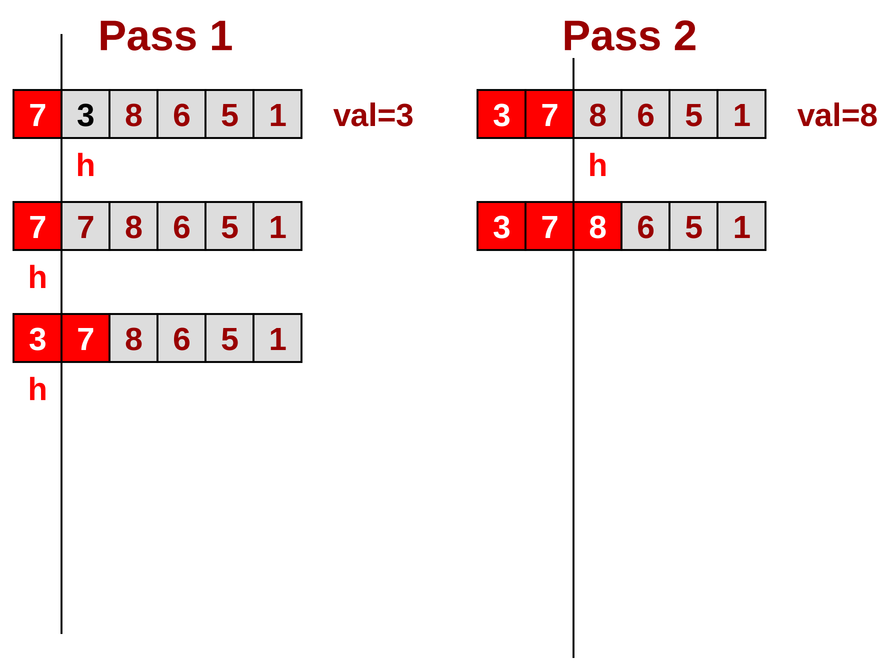
```
void isort(vector<int>& mylist)
{  for(int i=1; i < mylist.size()-1; i++){
      int val = mylist[i];
      hole = i;
      while(hole > 0 && val < mylist[hole-1]){
         mylist[hole] = mylist[hole-1];
         hole--;
      }
      mylist[hole] = val;
}  }
```



Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than k+1 are sorted
  - $\forall i, i < k: a[i] \leq a[i+1]$
- Can we make a claim about data at k+1 and beyond?
  - No, it's not guaranteed to be smaller or larger than what is in the sorted list

```
void isort(vector<int>& mylist)
{  for(int i=1; i < mylist.size()-1; i++){
      int val = mylist[i];
      hole = i;
      while(hole > 0 && val < mylist[hole-1]){
         mylist[hole] = mylist[hole-1];
         hole--;
      }
      mylist[hole] = val;
}  }
```



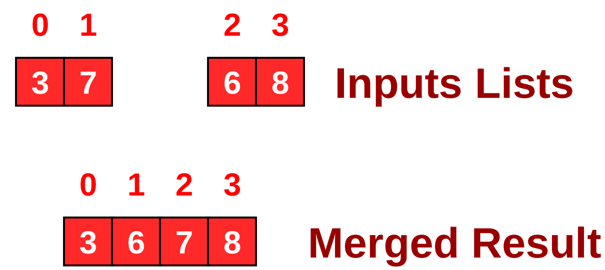
MERGESORT

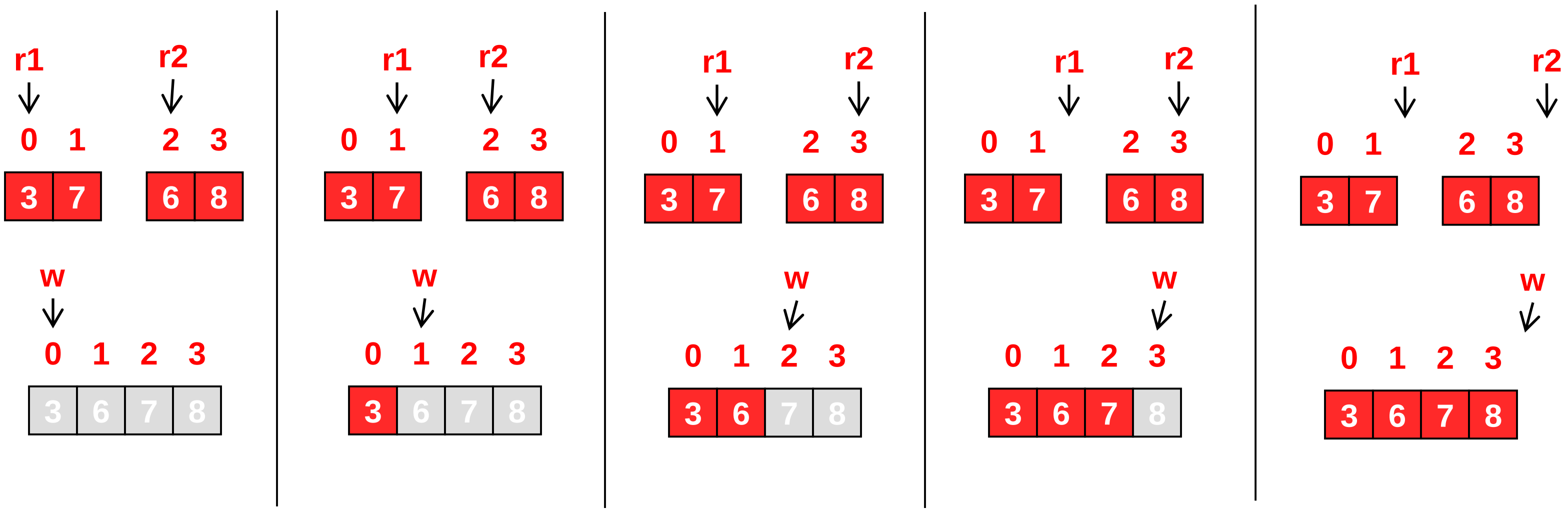
Exercise

- In-class exercise:
  - merge

Merge Two Sorted Lists

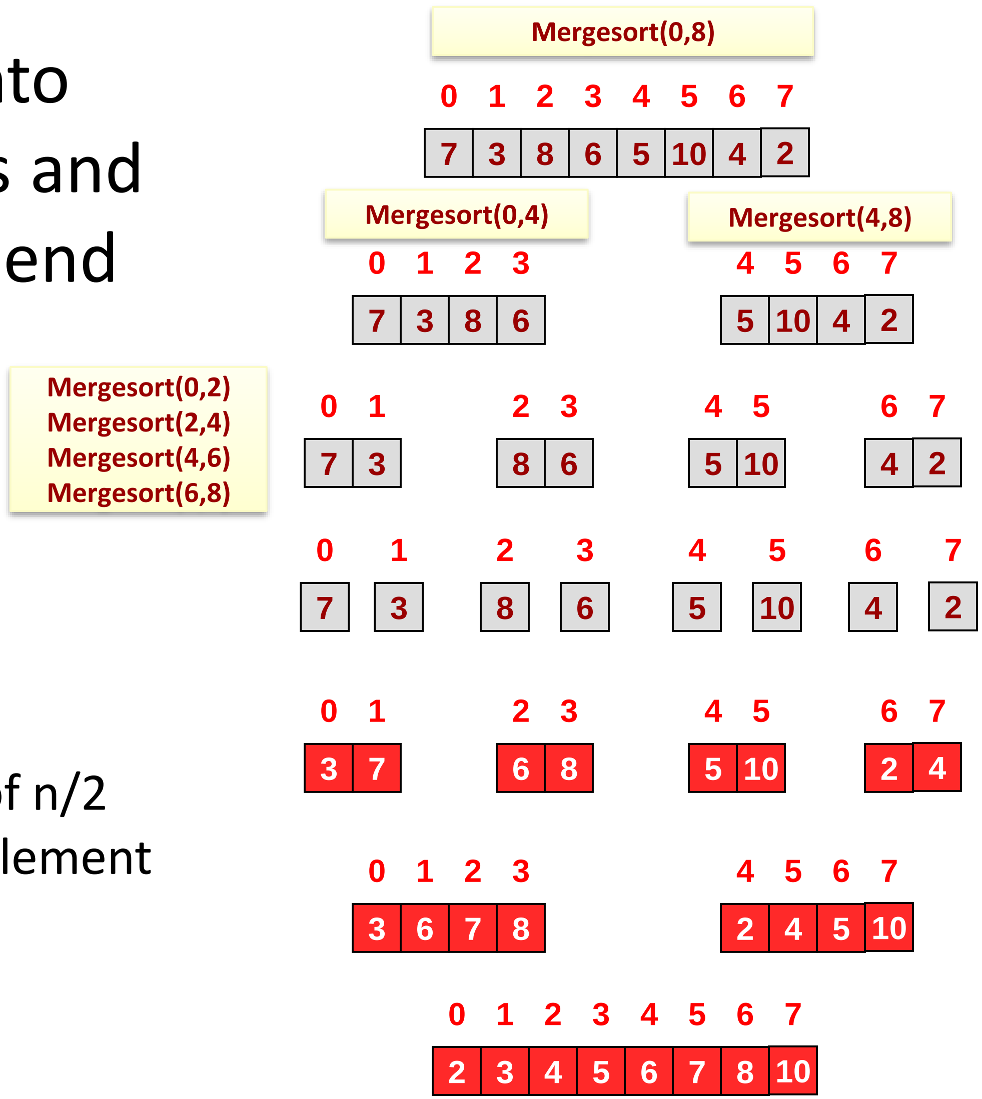
- Consider the problem of merging two sorted lists into a new combined sorted list
- Can be done in O(n)
- Can we merge in place or need an output array?





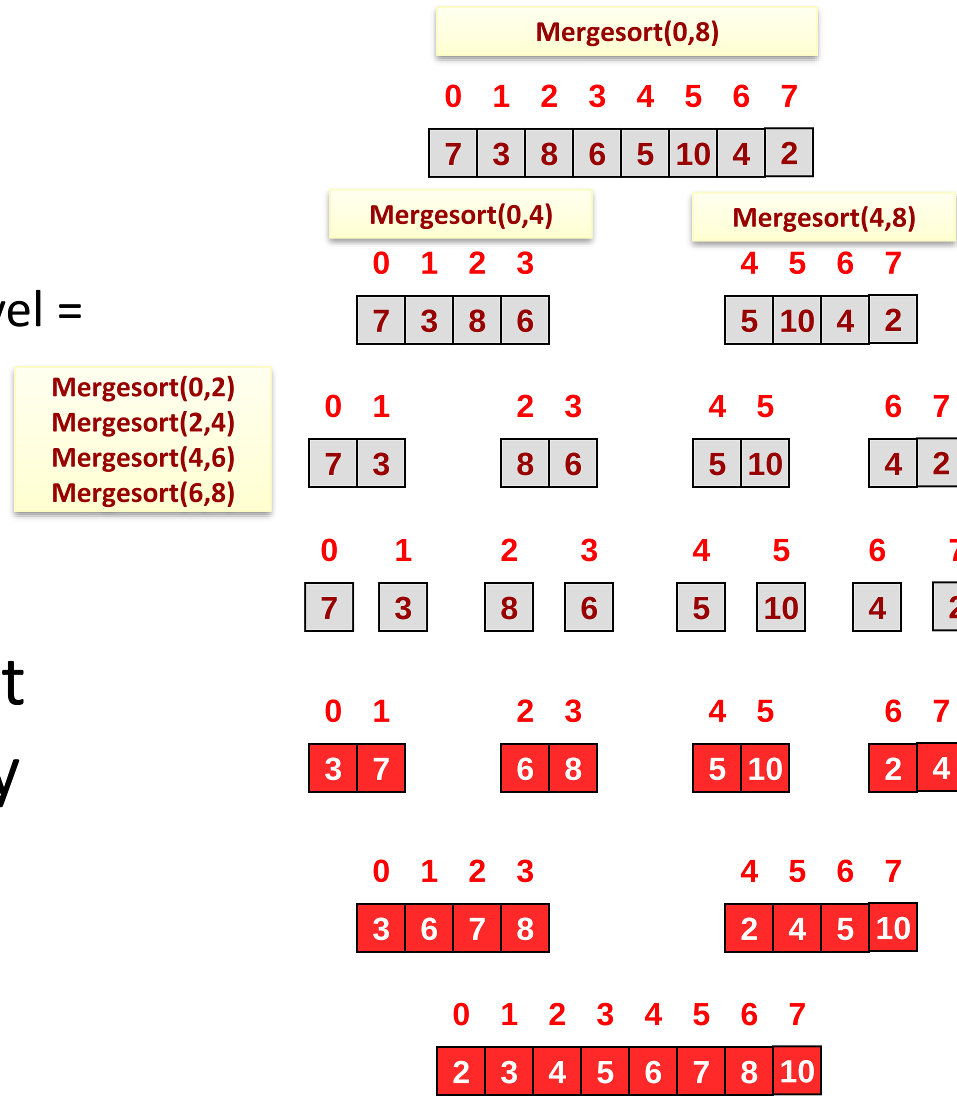
Recursive Sort (MergeSort)

- Break sorting problem into smaller sorting problems and merge the results at the end
- Mergesort(0..n)
  - If list is size 1, return
  - Else
    - Mergesort(0..n/2 - 1)
    - Mergesort(n/2 .. n)
    - Combine each sorted list of n/2 elements into a sorted n-element list



Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels =
    - $\log_2(n)$
  - Total operations to merge each level =
    - n operations total to merge two lists over all recursive calls at a particular level
- Mergesort = $O(n * \log_2(n))$
  - Usually has high constant factors due to extra array needed for merge



MergeSort Run Time

- Let's prove this more formally:
- $T(1) = \Theta(1)$
- $T(n) =$

MergeSort Run Time

- Let's prove this more formally:
- $T(1) = \Theta(1)$
- $T(n) = 2*T(n/2) + \Theta(n)$

k=1 $\quad T(n) = 2*T(n/2) + \Theta(n) \qquad T(n/2) = 2*T(n/4) + \Theta(n/2)$

k=2 $\quad = 2*2*T(n/4) + 2*\Theta(n)$

k=3 $\quad = 8*T(n/8) + 3*\Theta(n)$

$= 2^k*T(n/2^k) + k*\Theta(n)$

Stop @ T(1)
[i.e. $n = 2^k$]
$k=\log_2 n$

$= 2^k*T(n/2^k) + k*\Theta(n) = 2^{\log2(n)}*\Theta(1) + \log_2*\Theta(n) = n+\log_2*\Theta(n)$

$= \Theta(n*\log_2 n)$

Merge Sort

Courtesy of wikipedia.org

Recursive Sort (MergeSort)

```
void mergesort(vector<int>& mylist)
{
   vector<int> other(mylist);  // copy of array
   // use other as the source array, mylist as the output array
   msort(other, mylist, 0, mylist.size() );

}
void msort(vector<int>& mylist,
           vector<int>& output,
           int start,  int end)
{
   // base case
   if(start >= end) return;
   // recursive calls
   int mid = (start+end)/2;
   msort(mylist, output, start, mid);
   msort(mylist, output, mid,   end);
   // merge
   merge(mylist, output, start, mid, mid, end);
   // necessary bookkeeping to update mylist
}
void merge(vector<int>& mylist, vector<int>& output
           int s1, int e1, int s2, int e2)
{
...
}
```

Divide & Conquer Strategy

- Mergesort is a good example of a strategy known as "divide and conquer"
- 3 Steps:
  - Divide
    - Split problem into smaller versions (usually partition the data somehow)
  - Recurse
    - Solve each of the smaller problems
  - Combine
    - Put solutions of smaller problems together to form larger solution
- Another example of Divide and Conquer?
  - Binary Search

QUICKSORT

Partition & QuickSort

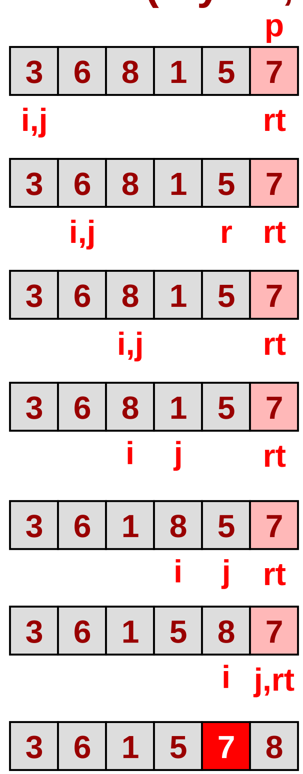
- Partition algorithm picks one (potentially arbitrary) number as the 'pivot' and puts it into the 'correct' location


```
int partition(int a[], int lt, int rt)
{
  int i = lt; // i will finish as the final location for pivot
  int pivot = a [rt]; // set pivot

  // iterate to find all elements smaller than pivot
  for ( int j = lt ; j < rt ; j++) {
    // move a smaller value to the left
    if ( a[j] <= pivot ) {
      swap ( a[i] , a[j] ) ;
      i++;
    }
  }
  swap ( a[i], a[rt] ); // place the pivot
  // return index where pivot is in correct position
  return i;
}
```

Note: lt and rt is inclusive in this example

Partition(mylist,0,5)



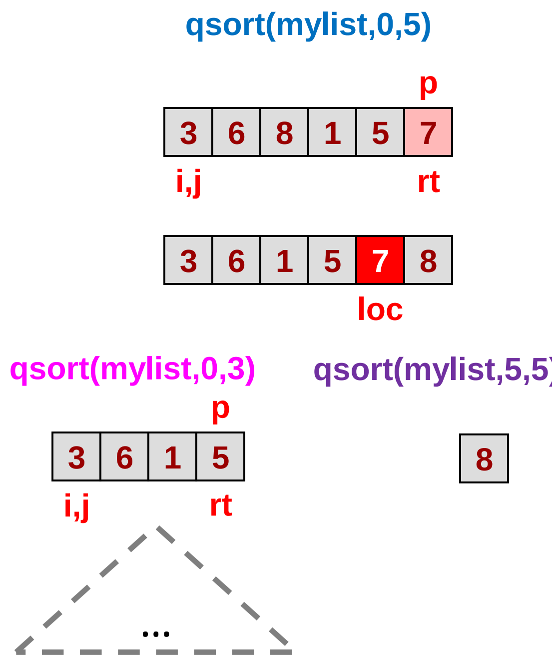
QuickSort

- Use the partition algorithm as the basis of a sort algorithm
- Partition on some element and the recursively call on both sides

< pivot	p	> pivot

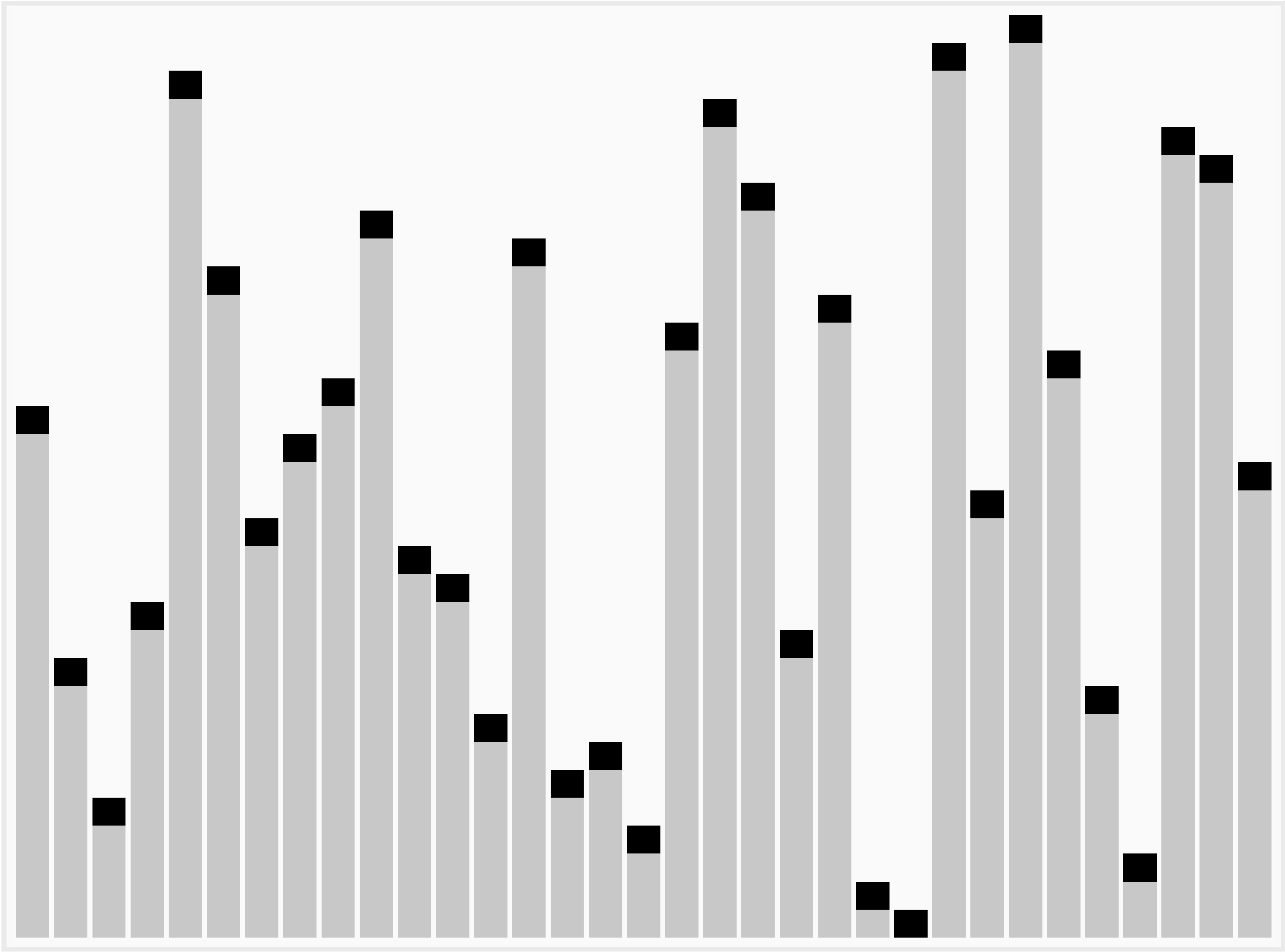
```
// range is [start,end] where end is inclusive
void qsort(vector<int>& mylist, int start, int end)
{
   // base case: list has 1 or less items
   if(start >= end) return;

   // pick a random pivot location [start..end]
   int p = start + rand() % (end-start+1);
   // partition
   int loc = partition(mylist,start,end,p)
   // recurse on both sides
   qsort(mylist,start,loc-1);
   qsort(mylist,loc+1,end);
}
```



Quick Sort

Value



List Index

Courtesy of wikipedia.org

QuickSort Analysis

- Worst Case Complexity:
  - When pivot chosen ends up being _______________________
  - Runtime:

3	6	8	1	5	7
3	6	1	5	7	8

3	6	8	1	5	7
3	1	5	6	8	7

- Best Case Complexity:
  - Pivot point chosen ends up being the _____________________
  - Runtime:

QuickSort Analysis

- Worst Case Complexity:
  - When pivot chosen ends up being min or max item
  - Runtime:
    - T(n) = Θ(n) + T(n-1)

3	6	8	1	5	7
3	6	1	5	7	8

3	6	8	1	5	7
3	1	5	6	8	7

- Best Case Complexity:
  - Pivot point chosen ends up being the median item
  - Runtime:
    - Similar to MergeSort
    - T(n) = 2T(n/2) + Θ(n)

QuickSort Analysis

- Average Case Complexity: O(n*log(n))
  - _____________ choose a pivot

3	6	8	1	5	7

QuickSort Analysis

- Worst Case Complexity:
  - When pivot chosen ends up being max or min of each list
  - $O(n^2)$
- Best Case Complexity:
  - Pivot point chosen ends up being the middle item
  - O(n*log(n))
- Average Case Complexity: O(n*log(n))
  - Randomly choose a pivot
- Pivot and quicksort can be slower on small lists than something like insertion sort
  - Many quicksort algorithms use pivot and quicksort recursively until lists reach a certain size and then use insertion sort on the small pieces
- Is quicksort stable? No (depends on pivot selection)

Comparison Sorts

- Big O of comparison sorts
  - It is mathematically provable that comparison-based sorts can never perform better than O(n*log(n))

- So can we ever have a sorting algorithm that performs better than O(n*log(n))?

- Yes, but only if we can make some meaningful assumptions about the input

OTHER SORTS

Sorting in Linear Time

- Radix Sort
  – Sort numbers one digit at a time starting with the least significant digit to the most.
- Bucket Sort
  – Assume the input is generated by a random process that distributes elements uniformly over the interval [0, 1)
- Counting Sort
  – Assume the input consists of an array of size N with integers in a small range from 0 to k.

Applications of Sorting

- Find the set_intersection of the 2 lists to the right
  – How long does it take?

A

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

B

	9	3	4	2	7	8	11
	0	1	2	3	4	5	6

Unsorted

- Try again now that the lists are sorted
  – How long does it take?

A

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

B

	2	3	4	7	8	9	11
	0	1	2	3	4	5	6

Sorted

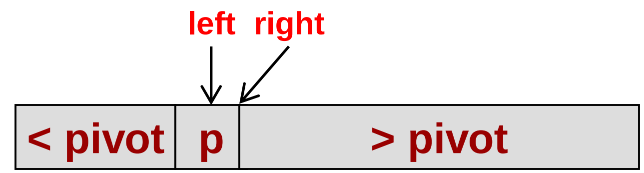
Other Resources

- http://www.youtube.com/watch?v=vxENKlcs2Tw
  - http://flowingdata.com/2010/09/01/what-different-sorting-algorithms-sound-like/
  - http://www.math.ucla.edu/~rcompton/musical_sorting_algorithms/musical_sorting_algorithms.html
- http://sorting.at/
- Awesome musical accompaniment: https://www.youtube.com/watch?v=ejpFmtYM8Cw

USC Viterbi School of Engineering

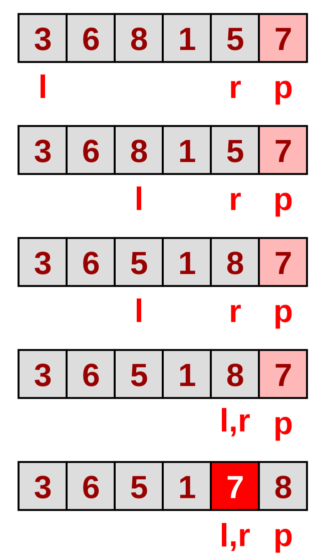
An Alternate Partition Implementation

- Partition algorithm picks one (potentially arbitrary) number as the 'pivot' and puts it into the 'correct' location




```
int partition(vector<int>& mylist, int start, int end, int p)
{  int pivot = mylist[p];
   swap(mylist[p], mylist[end]);  // move pivot out of the
                                  //way for now
   int left = start; int right = end-1;
   while(left < right){
     while(mylist[left] <= pivot && left < right)
          left++;  // find one from left that should be on right
     while(mylist[right] >= pivot && left < right)
        right--;  // find one from right that should be on left
     if(left < right)
        swap(mylist[left], mylist[right]);  // now swap them
   }
   if(mylist[right] > mylist[end]) {    // put pivot in
      swap(mylist[right], mylist[end]); // correct place
      return right;
   }
   else { return end; }
}
```

Partition(mylist,0,5,5)



Note: end is inclusive in this example