Selection sort

- Search through the list and find the smallest element
- swap the smallest element with the first element
- repeat starting at second element and find the second smallest element

```
public static void selectionSort(int[] list)
    {       int min;
            int temp;
            for(int i = 0; i < list.length - 1; i++) {
            min = i;
            for(int j = i + 1; j < list.length; j++)
                        if( list[j] < list[min] )
                                min = j;
                temp = list[i];
                list[i] = list[min];
                list[min] = temp;
             }
    }
```

Insertion sort

- Another of the O(N^2) sorts
- The first item is sorted
- Compare the second item to the first
  - if smaller swap
- Third item, compare to item next to it
  - need to swap
  - after swap compare again
- And so forth…

```
public void insertionSort(int[] list)
{       int temp, j;
        for(int i = 1; i < list.length; i++)
        {       temp = list[i];
                j = i;
                while( j > 0 && temp < list[j - 1])
                {       // swap elements
                        list[j] = list[j - 1];
                        list[j - 1] = temp;
                        j--;
                }
        }
}
```

Shell sort

- **Created by Donald Shell in 1959**
- **Wanted to stop moving data small distances (in the case of insertion sort and bubble sort) and stop making swaps that are not helpful (in the case of selection sort)**
- **Start with sub arrays created by looking at data that is far apart and then reduce the gap size**

46 2 83 41 102 5 17 31 64 49 18
Gap of five. Sort sub array with 46, 5, and 18
5 2 83 41 102 18 17 31 64 49 46
Gap still five. Sort sub array with 2 and 17
5 2 83 41 102 18 17 31 64 49 46
Gap still five. Sort sub array with 83 and 31
5 2 31 41 102 18 17 83 64 49 46
Gap still five Sort sub array with 41 and 64
5 2 31 41 102 18 17 83 64 49 46
Gap still five. Sort sub array with 102 and 49
5 2 31 41 49 18 17 83 64 102 46
Gap now 2: Sort sub array with 5 31 49 17 64 46
5 2 17 41 31 18 46 83 49 102 64
Gap still 2: Sort sub array with 2 41 18 83 102
5 2 17 18 31 41 46 83 49 102 64
Gap of 1 (Insertion sort)
2 5 17 18 31 41 46 49 64 83 102

```
public static void shellsort(Comparable[] list)
{  Comparable temp; boolean swap;
  for(int gap = list.length / 2; gap > 0; gap /= 2)
    for(int i = gap; i < list.length; i++)
    {  Comparable tmp = list[i];
       int j = i;
       for( ; j >= gap &&
                              tmp.compareTo( list[j - gap] ) <
0;
                              j -= gap )
                list[ j ] = list[ j - gap ];
       list[ j ] = tmp;
     }
}
```

Quick sort

- A divide and conquer approach that uses recursion

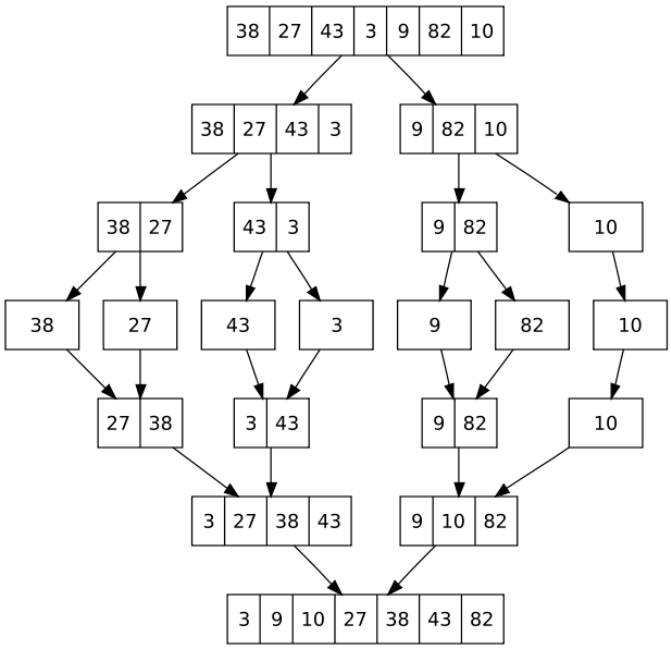
1. If the list has 0 or 1 elements it is sorted
2. otherwise, pick any element p in the list. This is called the pivot value
3. Partition the list minus the pivot into two sub lists according to values less than or greater than the pivot. (equal values go to either)
4. return the quicksort of the first list followed by the quicksort of the second list

```
public static void swapReferences( Object[] a, int
index1, int index2 )
  {  Object tmp = a[index1];
     a[index1] = a[index2];
     a[index2] = tmp;
  }
   public void quicksort( Comparable[] list, int start, int
stop )
   {   if(start >= stop)
         return; //base case list of 0 or 1 elements
          int pivotIndex = (start + stop) / 2;
          // Place pivot at start position
       swapReferences(list, pivotIndex, start);
       Comparable pivot = list[start];
       // Begin partitioning
       int i, j = start;
          // from first to j are elements less than or equal
to pivot
          // from j to i are elements greater than pivot
          // elements beyond i have not been checked
yet
          for(i = start + 1; i <= stop; i++ )
          {  //is current element less than or equal to
pivot
             if(list[i].compareTo(pivot) <= 0)
             {  // if so move it to the less than or equal
portion
                j++;
                swapReferences(list, i, j);
             }
          }

          //restore pivot to correct spot
          swapReferences(list, start, j);
          quicksort( list, start, j - 1 );    // Sort small
elements
          quicksort( list, j + 1, stop );   // Sort large
elements
    }
```

Merge Sort

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together




```
public static void mergeSort(Comparable[] c)
{       Comparable[] temp = new Comparable[ c.length
];
        sort(c, temp, 0, c.length - 1);
}
private static void sort(Comparable[] list, Comparable[]
temp,                          int low, int high)
{       if( low < high){
                int center = (low + high) / 2;
                sort(list, temp, low, center);
                sort(list, temp, center + 1, high);
                merge(list, temp, low, center + 1, high);
        }
}
```

```
private static void merge( Comparable[] list,
Comparable[] temp,
                           int leftPos, int rightPos, int
rightEnd){
        int leftEnd = rightPos - 1;
        int tempPos = leftPos;
        int numElements = rightEnd - leftPos + 1;
        //main loop
        while( leftPos <= leftEnd && rightPos <=
rightEnd){
                if( list[ leftPos
].compareTo(list[rightPos]) <= 0){
                        temp[ tempPos ] = list[ leftPos ];
                        leftPos++;
                }
                else{
                        temp[ tempPos ] = list[ rightPos
];
                        rightPos++;
                }
                tempPos++;
        }
        //copy rest of left half
        while( leftPos <= leftEnd){
                temp[ tempPos ] = list[ leftPos ];
                tempPos++;
                leftPos++;
        }
        //copy rest of right half
        while( rightPos <= rightEnd){
                temp[ tempPos ] = list[ rightPos ];
                tempPos++;
                rightPos++;
        }
        //Copy temp back into list
        for(int i = 0; i < numElements; i++, rightEnd--)
                list[ rightEnd ] = temp[ rightEnd ];
}
```

Bubble sort

- *Compares adjacent* array elements
  - *Exchanges* their values if they are *out of order*
- Smaller values *bubble up* to the top of the array
  - Larger values sink to the bottom

```
int pass = 1;
boolean exchanges;
do {
  exchanges = false;
  for (int i = 0; i < a.length-pass; i++)
    if (a[i].compareTo(a[i+1]) > 0) {
      T tmp = a[i];
      a[i] = a[i+1];
      a[i+1] = tmp;
      exchanges = true;
    }
pass++;
} while (exchanges);
```

TABLE 10.4
Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$