1. Partition
   1. Given a set of numbers, split them into 2 halves.
      1. ½ for values less than p
      2. ½ for values greater than p
   2. Example
      1. 8, 16, 13, 2, 9, 7, 15, 3, 14, 4, 1
   3. Advantage over MergeSort
      1. Does not have to copy/paste into dynamically-allocated arrays.
   4. Algorithm
      1. Choose a number to partition
         1. 8
      2. Keep an index pointer on the lowest number (i.e., closest to partition)
         1. 16
      3. Keep an index pointer on the highest number (i.e., furthest from partition).
         1. 1
      4. Swaps
         1. Swap 1 and 16
         2. Swap 13 and 4
         3. Don’t swap 2 and 3
         4. Swap 9 and 3
         5. Don’t swap 3 and 15
         6. 7 is now in the middle
         7. Finally, Swap 8 with 7
         8. 8 is now in the middle
      5. Next, recursively QuickSort the remaining halves
   5. Code
      1. quicksort(int\* array, int low, int high) {
         1. if (low < high) {
            1. int part\_index = partition(array, low, high);
            2. quicksort(array, low, part\_index – 1);
            3. quicksort(array, part\_index + 1, high);
         2. }
      2. }
      3. partition(int\* vals, int low, int high) {
         1. int temp;
         2. int i, lowpos;
         3. //Base case that should never occur
         4. if (low == high) return low;
         5. //Pick a random partition element and swap it into index low.
         6. i = low + rand() % (high – low + 1);
         7. swap(&vals[low], &vals[i]);
         8. //Store the index of the partition element
         9. lowpos = low;
         10. //Update low ptr
         11. low++;
         12. //Run partition so long as high and low values don’t cross
         13. while (low <= high){
             1. //Move low pointer until finding a value too large for this side
             2. while (low < = high && vals[low] <= vals[lowpos]) low++;
             3. //Move high pointer until finding a value too small for this side
             4. while (high >= low && vals[high] > vals[lowpos]) high--;
             5. //Swap values on the wrong side
             6. if (low < high) swap(&vals[low], &vals[high]);
         14. }
         15. //Swap partition element into its correct location
         16. swap(&vals[lowpos], &vals[high]);
         17. //Return index of the partition element
         18. return high;
      4. }
      5. Runtime
         1. O(*n*)
         2. Best case: O(*n*) + left\_side + right\_side = O(*n*) + 2T(*n*/2) = 2T(*n*/2) + 0(*n*)
         3. Pattern🡪 2k(3/4)k(*n*)*n*log*n*
         4. Worst case: O(*n*log*n*)
            1. Even on the average case, QuickSort beats MergeSort
2. How to Comparison Sort Quickly Using QuickSort
   1. When array size is large
      1. Use the median of a group of 3-5 randomly-selected items to pick the partition element.
         1. Takes an overhead cost.
   2. InsertionSort or SelectionSort work better for arrays with ≤40 elements.
      1. How to improve unnecessary calls in regular QuickSort
         1. if(low < high)
            1. if(high – low < 40) insertionsort(array, low, high);
         2. else
            1. <recursion>