Sorting (Rosen, 6th edition) Carol Zander

So far, we have examined only $O(n^2)$ algorithms (bubble, insertion, selection). We will now look at more efficient algorithms. Most are recursive, but the first one is a variation of the insertion sort.

Lower bound on simple sorts

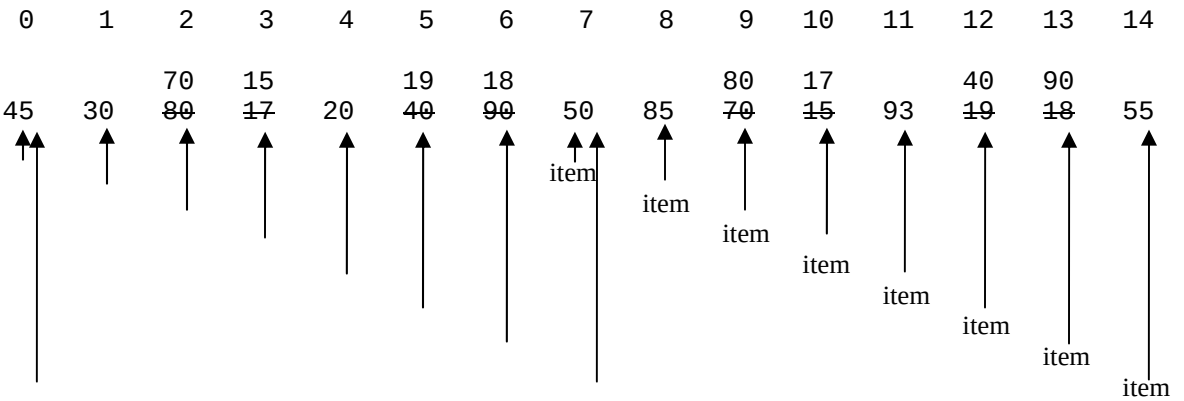
We see and (could prove) that if we use an algorithm that **only compares and swaps neighbors**, the algorithm must be **$O(n^2)$ on average**. We can determine the average number of comparisons using the concept of an ***inversion***. In any array, the number of inversions is any pair of numbers that are not in the correct order (no matter how far apart they are in the array). With the sorts we've seen so far, the reason they are $O(n^2)$ is that we can **never fix more than one inversion per swap** and we have seen that there are on average $O(n^2)$ of them. We can get around this by comparing and swapping elements that are not neighbors.

Shell sort

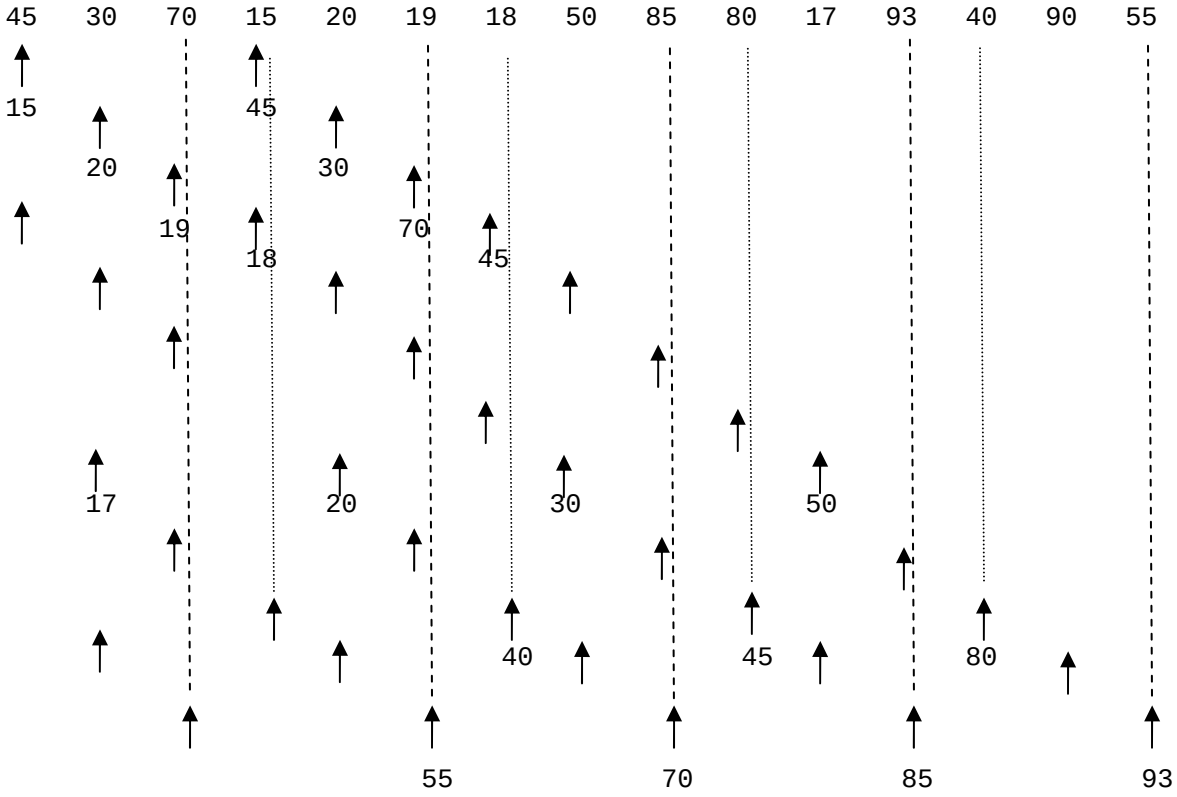
Shell sort is an easy to write algorithm that can do better than $O(n^2)$, but not as good as the best sorting algorithms. Donald Shell developed the algorithm in 1959. It is of interest primarily because it is **simple to write** and also because **the analysis of the algorithm is complex (and not completely understood).** The basic idea is that, rather than comparing neighbors, **compare (and swap, if necessary) elements that are farther away** (using some step between the elements). This allows multiple inversions to be fixed at once, but it doesn't completely sort the elements. Reduce the step and continue. Eventually, the step has to be reduced to one, at which point an insertion sort is being performed, but typically the array is nearly sorted by then.

```
void shellSort(int a[], int size) {
   int step = size/2;
   while (step > 0) {
      for (int unsorted = step; unsorted < size; unsorted++) {
         int i;
         int item = a[unsorted];

         // walk through array stepping by step amount, moving elements down
         for (i = unsorted; i >= step && item < a[i-step]; i-=step) {
            a[i] = a[i-step];
         }
         a[i] = item;
      }
      step = reduce(step);            // reduce step by chosen amount
   }
}
```

Using this notation, the length of the arrow represents comparisons in inner loop. step = 15/2 = 7



Changing notation, each row of arrows represents comparisons and moves in the inner loop. If a value changes, it is shown below the arrow. So, the most recent value in an element is at the bottom of a column. Let step = 7/2 = 3.



Let step = 3/2 = 1, and now it's a regular insertion sort.

15	17	19	18	20	55	40	30	70	45	50	85	80	90	93
15	17	19	18	20	55	40	30	70	45	50	85	80	90	93
15	17	19	18	20	55	40	30	70	45	50	85	80	90	93
15	17	18	19	20	55	40	30	70	45	50	85	80	90	93
15	17	18	19	20	55	40	30	70	45	50	85	80	90	93
15	17	18	19	20	55	40	30	70	45	50	85	80	90	93
15	17	18	19	20	40	55	30	70	45	50	85	80	90	93
15	17	18	19	20	30	40	55	70	45	50	85	80	90	93
15	17	18	19	20	30	40	55	70	45	50	85	80	90	93
15	17	18	19	20	30	40	45	55	70	50	85	80	90	93
15	17	18	19	20	30	40	45	50	55	70	85	80	90	93
15	17	18	19	20	30	40	45	50	55	70	85	80	90	93
15	17	18	19	20	30	40	45	50	55	70	80	85	90	93
15	17	18	19	20	30	40	45	50	55	70	80	85	90	93
15	17	18	19	20	30	40	45	50	55	70	80	85	90	93

Oddly, adding an extra loop to insertion sort – while (step > 0) – reduces the amount of work done by the algorithm. How should the step be reduced? Here are some options:

Reduce it by one at each step.	Regular insertion sort, bad - $O(n^2)$ best and worst case.
Divide it by two at each step.	This is okay. $O(n^2)$ worst case. $O(n^{1.5})$ average case.
Divide it by two and add one if it is even.	This is good. $O(n^{1.5})$ worst case. $O(n^{1.25})$ average case??
Divide it by 2.2.	This is even better. $O(n^{1.16})$ average case?

Quick sort (Hoare sort)

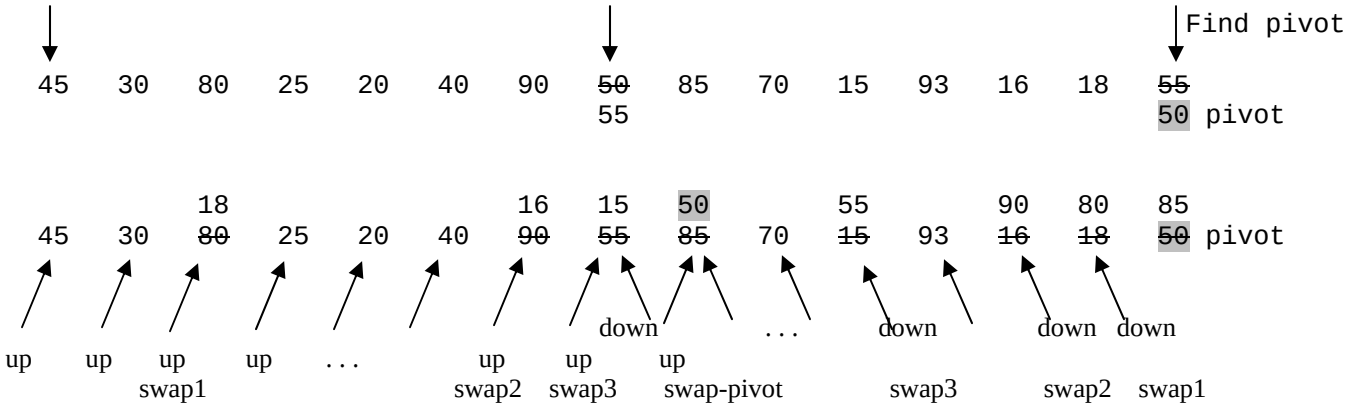
QuickSort (HoareSort in honor of Tony Hoare, its creator in 1960) is the **most commonly used sorting algorithm for large arrays** because it is (usually, meaning on average) both **time efficient (like you will see with merge sort)** and **space efficient (unlike merge sort).** It is a clever algorithm, uses a **divide-and-conquer approach**. In this sort, choose one particular element that is called the "pivot." The array is then divided into all elements that are less than the pivot (these are placed at the beginning of the array) and all elements that are greater than (or equal to) the pivot (these are placed at the end of the array). The pivot is placed in its sorted position. The algorithm is then recursively called on both of the smaller sets. When the algorithm finishes, the data is completely sorted. There are variations of this sort, mainly how the pivot is chosen and whether the pivot has to be an element in the array, but the algorithm is essentially the same.

```
//---------------------------------------------------------------------------
// median
// find median of low, center, and high values; order these and hide pivot.

int median(int a[], int low, int high) {
   int center = (low+high)/2;
   if(a[low] > a[center])  swap(a[low], a[center]);
   if(a[low] > a[high])    swap(a[low], a[high]);
   if(a[center] > a[high]) swap(a[center], a[high]);
   swap(a[center], a[high-1]);                        // hide the pivot
   return a[high-1];                                  // return pivot
}

//---------------------------------------------------------------------------
// partition
// separate array into 2 partitions: numbers < pivot, numbers > pivot
void partition(int a[], int pivot, int& up, int& down) {
   for (;;) {
      while (a[++up] < pivot);           // find value > pivot
      while (a[--down] > pivot);         // find value < pivot
      if (up < down)
         swap(a[up], a[down]);
      else
         break;
   }
}

//---------------------------------------------------------------------------
// quickSort
// sort an array.  continually partition into sets of numbers < pivot and
// numbers > pivot.  recursively continue until done.

void quickSort(int a[], int low, int high) {
   int up, down, pivot;

   // use quicksort unless the size is <= CUTOFF, then use insertion sort
   if (low+CUTOFF <= high) {
      pivot = median(a, low, high);
      up = low;
      down = high-1;

      // separate array into 2 partitions: numbers < pivot, numbers > pivot
      partition(a, pivot, up, down);
      swap(a[up], a[high-1]);             // put pivot in rightful position
      quickSort(a, low, up-1);
      quickSort(a, up+1, high);
   }
   else
      insertionSort(a, low, high-low+1);
}
```

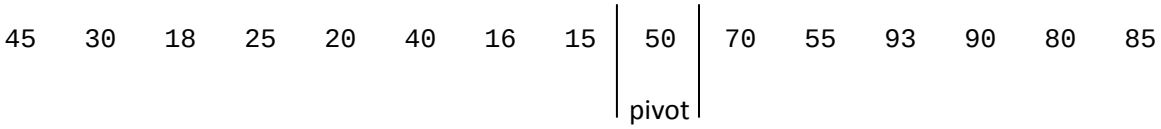
You want to choose a pivot that is as close to the middle of the range of numbers as possible, but you don't want to do too much work. The scheme above finds the median of the first, middle, and last elements. Your text suggests taking the first element. This is risky because if the array is already sorted, you end up with a running time as bad as the $O(n^2)$ algorithms.

To partition the array, one counter walks up the array, another walks down the array. When walking up, you look for an item larger than the pivot, when walking down, you look for an item smaller than the pivot. Swap them and continue the walking up and down and swapping until the up and down counters cross.

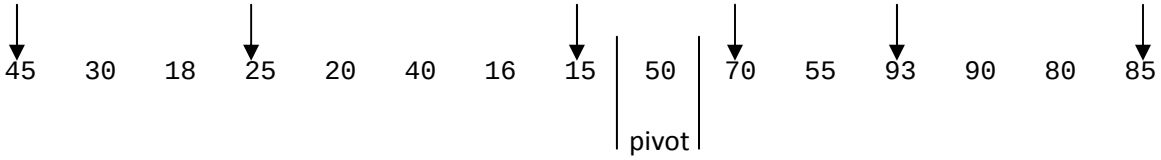
This is not 100% what the code does because when finding the median, the values are not always swapped in my example. If that is done, the numbers get sorted too quickly and you don't get a feel for the algorithm. We would need a much larger number of numbers.



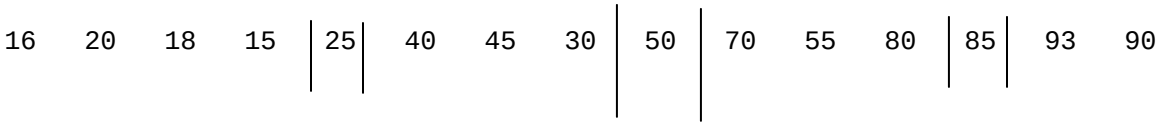
When the up and down cross, swap the pivot with the element at subscript *up*.



Do the same thing on each "half". For the bottom half, the pivot is 25. The upper half pivot is 85. Again, when finding the median, the numbers are not always swapped (although the pivot is hidden at the end).



After the up and down swapping, the array is shown.

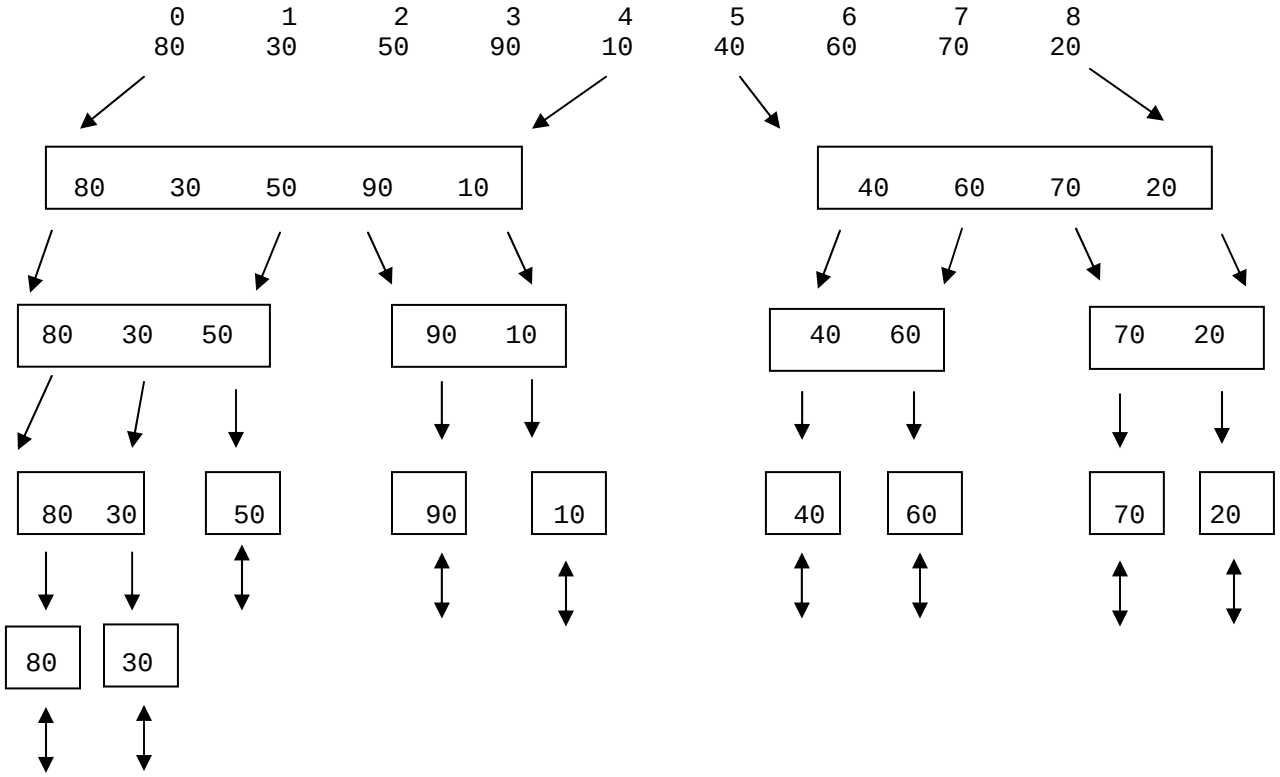


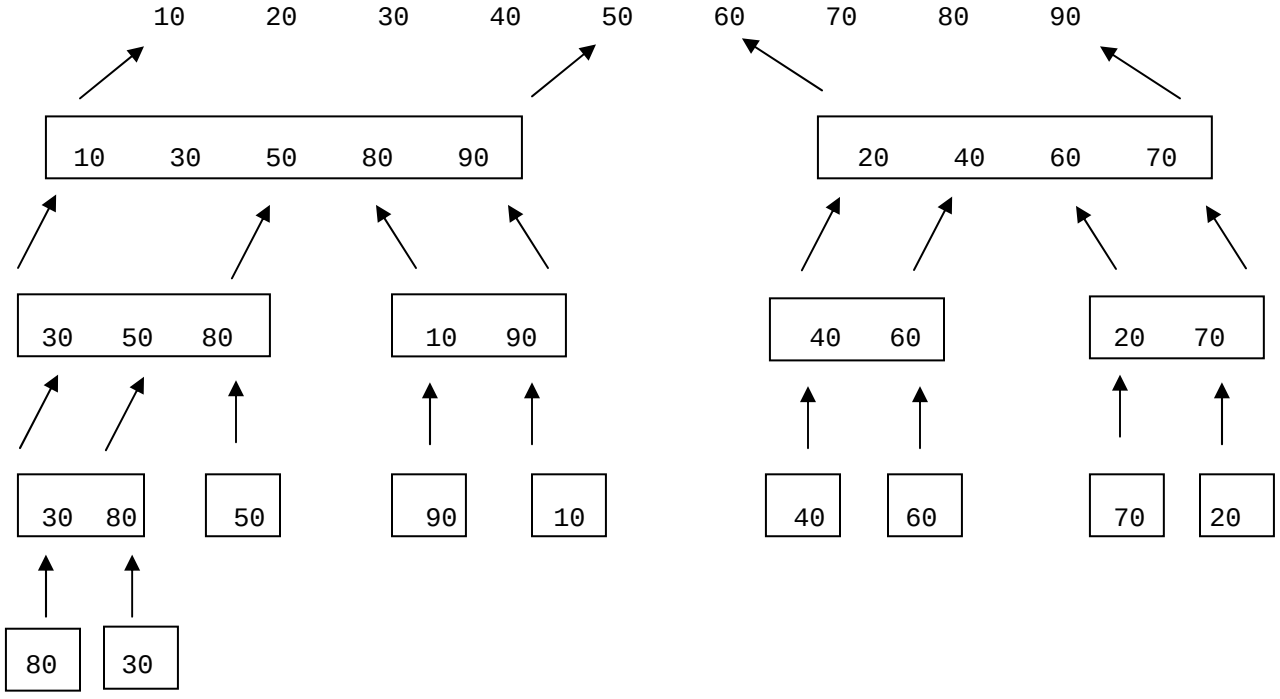
Merge sort

Merge sort is another divide-and-conquer algorithm. In general, merge sort works in a manner similar to hoare sort -- splits the data into two subsets, solves the problem on the smaller sets, and then combines the sets again. Merge sort is more straightforward -- breaks the array into two subsets by cutting it in the middle, sorts both subsets (recursively), and then combines them again.

```
//-----------------------------------------------------------------------
// merge
// merge two sorted arrays into one long sorted array

void merge(int a[], int low, int mid, int high) {
   int temp[high - low + 1];

   int low1 = low;
   int high1 = mid;
   int low2 = mid + 1;
   int high2 = high;

   // As long as both lists still have elements, add the next.
   int index;
   for (index = 0; (low1 <= high1) && (low2 <= high2); index++) {
      if (a[low1] < a[low2]) {
         temp[index] = a[low1];
         low1++;
      }
      else {
         temp[index] = a[low2];
         low2++;
      }
   }

   // One of the lists still has elements, so add them now.
   for (; low1 <= high1; index++, low1++)
      temp[index] = a[low1];

   for (; low2 <= high2; index++, low2++)
      temp[index] = a[low2];

   // Copy back into the array.
   for (index = 0; index < high - low + 1; index++)
      a[index + low] = temp[index];
}

//-----------------------------------------------------------------------
// mergeSort
// Break the array into two subsets by cutting it in the middle,
// sorts both subsets and combine

void mergeSort(int a[], int low, int high) {
   if (low < high) {
      int mid = (low + high) / 2;
      mergeSort(a, low, mid);
      mergeSort(a, mid + 1, high);

      merge(a, low, mid, high);
   }
}
```

When executing the merge sort, on the way down of the recursion, the items are split into parts, but the actual work of merging the smaller sets into sorted bigger sets is done on the way back up from the recursion. These execution trees show calling the recursive mergeSort, the “down”, and the returning, the “up”, separately.





Radix sort

The final sorting algorithm discussed is the radix sort, which is very different from the other sorting algorithms that we have looked at so far, since **no comparisons are ever done between array elements**. The basic idea is to **group the elements into sets**, where the sets can be placed in a known order.

An example is in sorting 3 digit numbers between 000 and 999. All numbers that start with a 0 can be placed in a group, all numbers starting with 1 in another, and so. After examining the first digit, the order to place the sets in is known, but the items must still be sorted within each set. This is performed using the same process on the second digit of the number (recursively or iteratively). In practice, this is usually done in the reverse order of the digits, so that the most important digit is examined last. However, for each step, this means to keep the elements with the same digit at the current position in the same order that they were in from the previous step to make sure that the numbers stay sorted correctly. Typically queues are used:

```
void radixSort(int a[], int size, int d) {      // d is the number of digits

      for (j = d; j > 0; j--){
            initialize 10 empty queues
            for (i = 0; i < size; i++){
                  k = jth digit of a[i]
                  insert a[i] into queue k
            }
            Replace the items in a with all the items in group 0, group 1, etc.
      }
}
```

The data to be sorted -- stored somewhere, possibly a data file:

```
  64    8    216    512    27    729    1    0    343    125    713    525    537    347    382
```

Pass 1 – sort by the least significant digit (put numbers into the queue, enqueue)

```
   0        1        2        3        4        5        6        7        8        9
---------------------------------------------------------------------------------------

   0        1      512      343       64      125      216       27        8      729
                   382      713              525              537
                                                              347
```

Pass 2 – sort by the next significant digit (dequeue, then enqueue; need counts of how many are in each queue because you may insert into a queue from a later pass before you have removed all the numbers from an earlier pass)

```
   0        1        2        3        4        5        6        7        8        9
---------------------------------------------------------------------------------------

   0      512      125      537      343               64               382
   1      713      525               347
   8      216       27
                   729
```

Pass 3 – sort by the third significant digit

```
   0        1        2        3        4        5        6        7        8        9
---------------------------------------------------------------------------------------

   0      125      216      343               512               713
   1                        347               525               729
   8                        382               537
  27
  64
```

Now empty out the queues starting at zero, and the numbers are sorted.