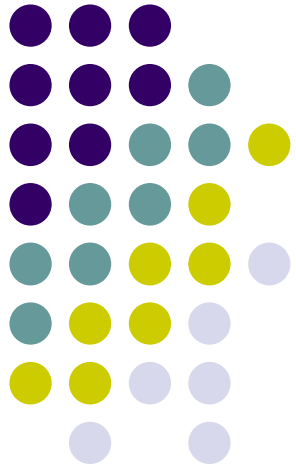
Mastering Data Structures and Algorithms: A Practical Approach

Sorting and Searching

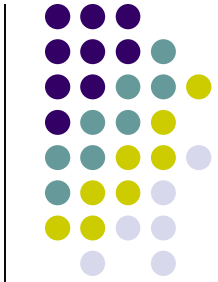


By Dr. Juan C. Gomez

Fall 2018

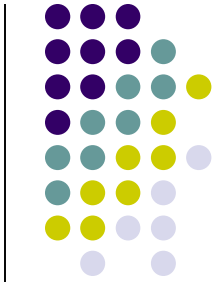
UC Santa Cruz Extension in Silicon Valley

Overview



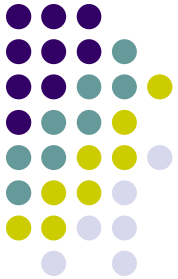
- Common Sorting Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
  - Linear Sort
    - Bucket Sort
    - Radix Sort

Overview



- External Sorting
- Searching Algorithms
  - Binary Search
- Sample Problems

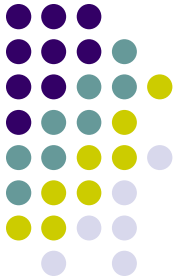
Bubble Sort



Bubble Sort | Runtime: $O(n^2)$ **average and worst case. Memory:** $O(1)$.

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly "bubble" up to the beginning of the list.

Selection Sort

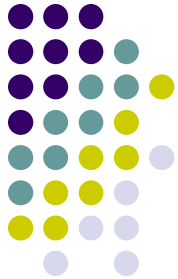


Selection Sort | Runtime: $O(n^2)$ **average and worst case. Memory:** $O(1)$.

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

Insertion Sort

Shell Sort



```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps)
{
    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped order
    # keep adding one more element until the entire array is gap sorted
    for (i = gap; i < n; i += 1)
    {
        # add a[i] to the elements that have been gap sorted
        # save a[i] in temp and make a hole at position i
        temp = a[i]
        # shift earlier gap-sorted elements up until the correct location for a[i] is found
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }
}
```

Shell Sort

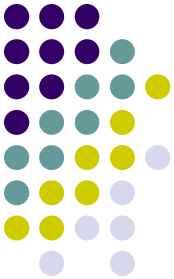


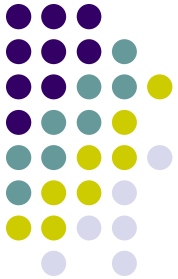
Figure 7.3 Shellsort after each pass, using {1, 3, 5} as the increment sequence

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

Shell Sort

General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Pratt, 1971[1]

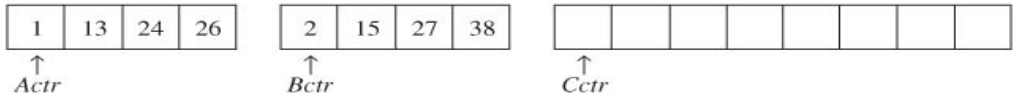
Merge Sort



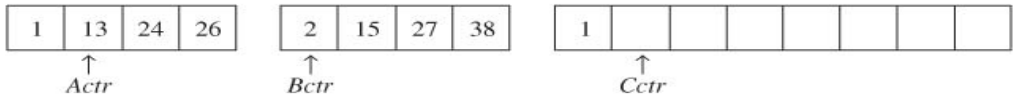
Merge Sort | Runtime: O(n log(n)) **average and worst case. Memory:** Depends.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single-element arrays. It is the "merge" part that does all the heavy lifting.

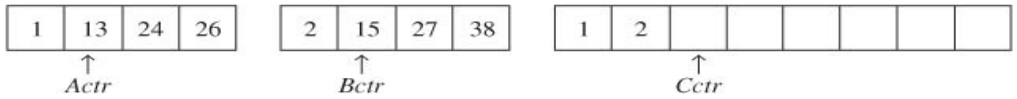
Merge Sort



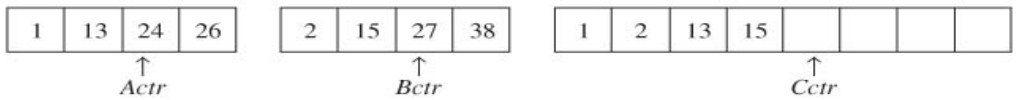
If the array *A* contains 1, 13, 24, 26, and *B* contains 2, 15, 27, 38, then the algorithm proceeds as follows
2 are compared.



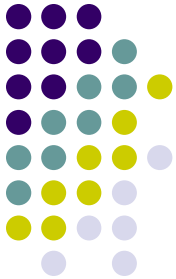
2 is added to *C*, and then 13 and 15 are compared.



13 is added to *C*, and then 24 and 15 are compared. This proceeds until 26 and 27 are compared.



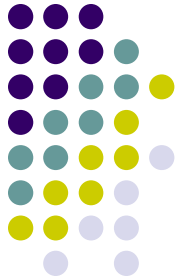
Quick Sort



Quick Sort | Runtime: $O(n \log(n))$ **average,** $O(n^2)$ **worst case. Memory:** $O(\log(n))$.

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

Heap Sort



Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Heap Sort

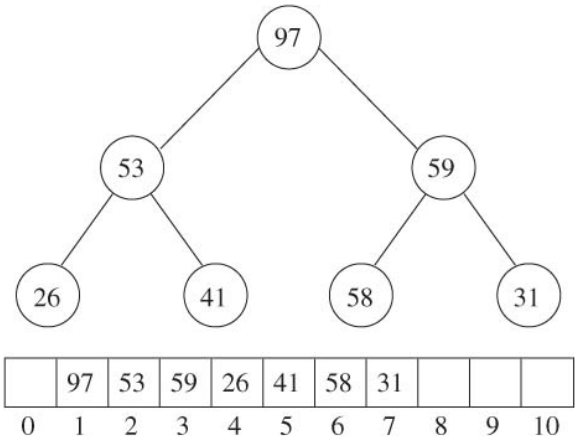
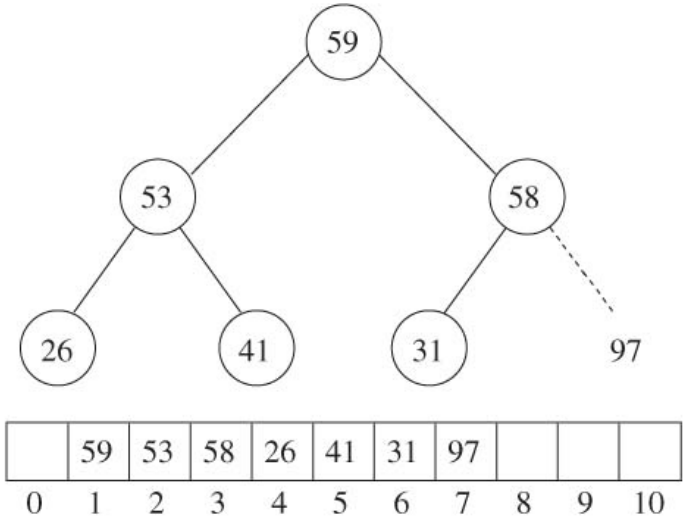
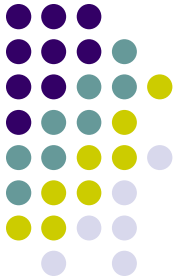


Figure 7.6 (*Max*) heap after `buildHeap` phase

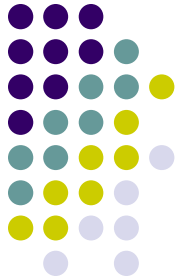


Bucket Sort



A simple example is **bucket sort**. For bucket sort to work, extra information must be available. The input $A_1, A_2, \ldots, A_N$ must consist of only positive integers smaller than M. (Obviously extensions to this are possible.) If this is the case, then the algorithm is simple: Keep an array called `count`, of size M, which is initialized to all 0's. Thus, `count` has M cells, or buckets, which are initially empty. When A_i is read, increment `count[`A_i`]` by 1. After all the input is read, scan the `count` array, printing out a representation of the sorted list. This algorithm takes $O(M + N)$; the proof is left as an exercise. If M is $O(N)$, then the total is $O(N)$.

Radix Sort



Radix Sort | Runtime: $O(kn)$ **(see below)**

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Radix Sort

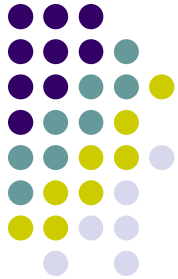
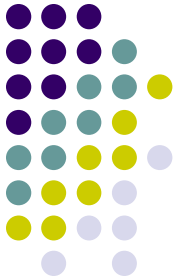


Figure 7.22 Radix sort trace

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Radix Sort



```
1     /*
2     * Radix sort an array of Strings
3     * Assume all are all ASCII
4     * Assume all have same length
5     */
6     public static void radixSortA( String [ ] arr, int stringLen )
7     {
8           final int BUCKETS = 256;
9           ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];
10
11          for( int i = 0; i < BUCKETS; i++ )
12              buckets[  i ] = new ArrayList<>( );
13
14          for( int pos = stringLen - 1; pos >= 0; pos-- )
15          {
16              for( String s : arr )
17                  buckets[ s.charAt( pos ) ].add( s );
18
19              int idx = 0;
20              for( ArrayList<String> thisBucket : buckets )
21              {
22                  for( String s : thisBucket )
23                      arr[ idx++ ] = s;
24
25                  thisBucket.clear( );
26              }
27          }
28    }
```

Radix Sort

Figure 7.24 Counting radix sort for fixed-length strings

```
1     /*
2      * Counting radix sort an array of Strings
3      * Assume all are all ASCII
4      * Assume all have same length
5      */
6     public static void countingRadixSort( String [ ] arr, int stringLen )
7     {
8         final int BUCKETS = 256;
9
10        int N = arr.length;
11        String [ ] buffer = new String [ N ];
12
13        String [ ] in = arr;
14        String [ ] out = buffer;
15
16        for( int pos = stringLen - 1; pos >= 0; pos-- )
17        {
18            int [ ] count = new int [ BUCKETS + 1 ];
19
20            for( int i = 0; i < N; i++ )
21                count[ in[ i ].charAt( pos  ) + 1 ]++;
22
23            for( int b = 1; b <= BUCKETS; b++ )
24                count[ b ] += count[  b - 1 ];
25
26            for( int i = 0; i < N; i++ )
27                out[ count[ in[ i ].charAt( pos ) ]++ ] = in[ i ];
28
29              // swap in and out roles
30            String [ ] tmp = in;
31            in = out;
32            out = tmp;
33        }
34
35            // if odd number of passes, in is buffer, out is arr; so copy back
36        if( stringLen % 2 == 1 )
37            for( int i = 0; i < arr.length; i++ )
38                out[ i ] = in[ i ];
39    }
```

Radix Sort

Figure 7.25 Radix sort for variable length strings

```
1       /*
2        * Radix sort an array of Strings
3        * Assume all are all ASCII
4        * Assume all have length bounded by maxLen
5        */
6       public static void radixSort( String [ ] arr, int maxLen )
7       {
8           final int BUCKETS = 256;
9
10          ArrayList<String> [ ] wordsByLength = new ArrayList<>[ maxLen + 1 ];
11          ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];
12
13          for( int i = 0; i <  wordsByLength.length; i++ )
14              wordsByLength[  i ] = new ArrayList<>( );
15
16          for( int i = 0; i <  BUCKETS; i++ )
17              buckets[  i ] = new ArrayList<>( );
18
19          for( String s : arr )
20              wordsByLength[ s.length( ) ].add( s );
21
22          int idx = 0;
23          for( ArrayList<String> wordList : wordsByLength )
24              for( String s : wordList )
25                  arr[ idx++ ] = s;
26
27          int startingIndex = arr.length;
28          for( int pos = maxLen - 1; pos >= 0; pos-- )
29          {
30              startingIndex -= wordsByLength[ pos + 1 ].size( );
31
32              for( int i = startingIndex; i < arr.length; i++ )
33                  buckets[ arr[ i ].charAt( pos ) ].add( arr[ i ] );
34
35              idx = startingIndex;
36              for( ArrayList<String> thisBucket : buckets )
37              {
38                  for( String s : thisBucket )
39                      arr[ idx++ ] = s;
40
41                  thisBucket.clear( );
42              }
43          }
44      }
```

External Sort

The basic external sorting algorithm uses the merging algorithm from mergesort. Suppose we have four tapes, T_{a1}, T_{a2}, T_{b1}, T_{b2}, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes. Suppose the data are initially on T_{a1}. Suppose further that the internal memory can hold (and sort) *M* records at a time. A natural first step is to read *M* records at a time from the input tape, sort the records internally, and then write the sorted records alternately to T_{b1} and T_{b2}. We will call each set of sorted records a **run.** When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shellsort.

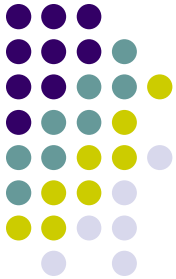
T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If $M = 3$, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

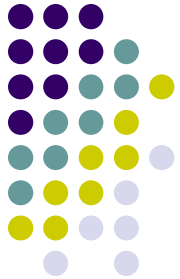
Now T_{b1} and T_{b2} contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto T_{a1}. Recall that merging two sorted lists is simple; we need almost no memory, since the merge is performed as T_{b1} and T_{b2} advance. Then we take the next run from each tape, merge these, and write the result to T_{a2}. We continue this process, alternating between T_{a1} and T_{a2}, until either T_{b1} or T_{b2} is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes and repeat the same steps, this time using the *a* tapes as input and the *b* tapes as output. This will give runs of $4M$. We continue the process until we get one run of length *N*.

Binary Search: Iterative



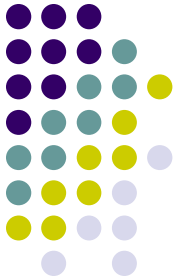
```
1   int binarySearch(int[] a, int x) {
2      int low = 0;
3      int high = a.length - 1;
4      int mid;
5
6      while (low <= high) {
7         mid = (low + high) / 2;
8         if (a[mid] < x) {
9            low = mid + 1;
10        } else if (a[mid] > x) {
11           high = mid - 1;
12        } else {
13           return mid;
14        }
15     }
16     return -1; // Error
17  }
```

Binary Search: Recursive



```
19 int binarySearchRecursive(int[] a, int x, int low, int high) {
20    if (low > high) return -1; // Error
21
22    int mid = (low + high) / 2;
23    if (a[mid] < x) {
24       return binarySearchRecursive(a, x, mid + 1, high);
25    } else if (a[mid] > x) {
26       return binarySearchRecursive(a, x, low, mid - 1);
27    } else {
28       return mid;
29    }
30 }
```

Sample Problems

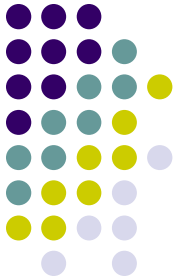


10.1 Sorted Merge: You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

Hints: #332

pg 396

Sample Problems



10.2 **Group Anagrams:** Write a method to sort an array of strings so that all the anagrams are next to each other.

Hints: #177, #182, #263, #342

pg 397

10.3 **Search in Rotated Array:** Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

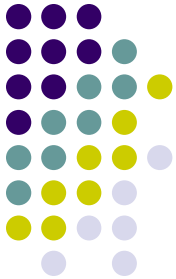
Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

Hints: #298, #310

pg 398

Sample Problems



10.4 **Sorted Search, No Size:** You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element `x` occurs. If `x` occurs multiple times, you may return any index.

Hints: #320, #337, #348

pg 400

10.5 **Sparse Search:** Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

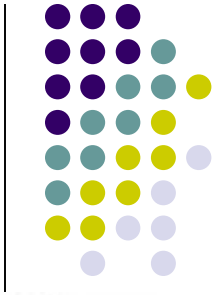
Input: `ball, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}`

Output: 4

Hints: #256

pg 401

Sample Problems



10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

Hints: #207

pg 402

10.7 **Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

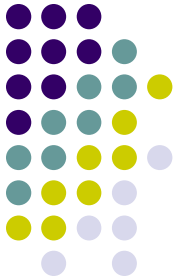
FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

Hints: #235, #254, #281

pg 403

Sample Problems



10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

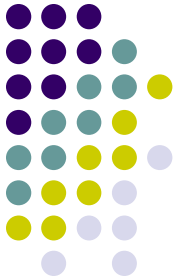
pg 406

10.9 Sorted Matrix Search: Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

Sample Problems



10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

pg 406

10.9 Sorted Matrix Search: Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

Sample Problems

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

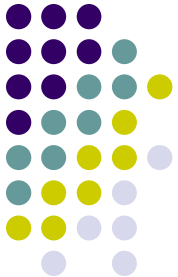
EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

```
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

Hints: #301, #376, #392

Sample Problems



10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

Hints: #196, #219, #231, #253, #277, #292, #316

pg 414