Sorting (cont.)

Instructor: Dr. Sunho Lim (Ph.D., Assistant Professor)

Lecture 18

sunho.lim@ttu.edu

Adapted partially from Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning; and Algorithms and Data Structures, Douglas Wilhelm Harder, Mmath

Efficient Sorting Algorithms: Shell Sort

- In insertion sort,
  - work well when the input element is "almost sorted"
  - inefficient to move the elements
- Shell sort
  - improve over insertion sort
  - number of data movements

Efficient Sorting Algorithms: Shell Sort (cont.)

- The pseudo code of shell sort:

```
divide data into h subarrays;
        for i = 1 to h
            sort subarray data_i;
        sort array data;
```

  - if *h* is **too small**
    - the subarrays could be too large and the resulting sort would be inefficient
  - if *h* is **too big,**
    - too many subarrays
  - use several different subarrays,
    - apply the same process separately to each subarray

Efficient Sorting Algorithms: Shell Sort (cont.)

- The pseudo code of shell sort (cont.):

```
determine numbers h_t . . . h_1 of ways of dividing array
data into subarrays;
for (h=h_t; t > 1; t--, h=h_t)
    divide data into h subarrays;
for i = 1 to h
    sort subarray data_i;
sort array data;
```

  - called,
    - diminishing increment sort, shell sort, or shell's method

Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort,
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

```
63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33
```

Result:

```
63  19  7   90  81  36  54  45        63  19  7   9   41  36  33  45
72  27  22  9   41  59  33            72  27  22  90  81  59  54
```

```
63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54
```

Result:

```
63  19  7   9   41        22  19  7   9   27
36  33  45  72  27        36  33  45  59  41
22  90  81  59  54        63  90  81  72  54
```

```
22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54
```


Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort, (cont.)
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

```
22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54
```

Result:

```
22  19  7          9   19  7
9   27  36         22  27  36
33  45  59         33  45  54
41  63  90         41  63  59
81  72  54         81  72  90
```

```
9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90
```


Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort, (cont.)
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

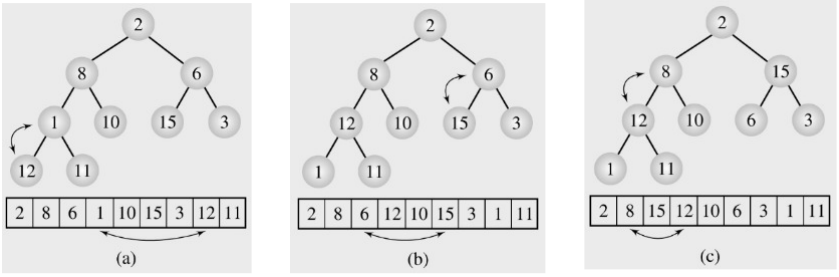
	Result:
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

Efficient Sorting Algorithms: Heap Sort

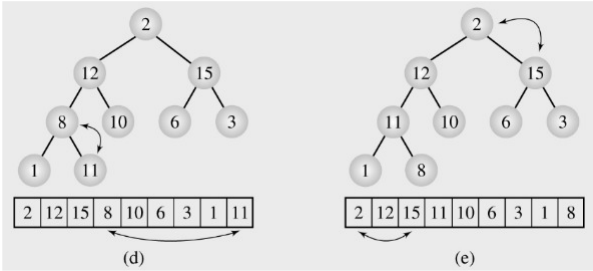
- Motivation
  - selection sort, fairly inefficient (O(n^2)),
    - relatively few moves of the data
    - recall, selection sort finds the smallest element in the list and places it first, then the next smallest, etc.
  - if the comparison portion of the sort can be improved?
    - performance can likewise improve
- Two phrases:
  - for **ascending order**,
    - place the **largest element last** in the array, then put the next largest in front of that, etc.
  - 1st: build a heap out of the data
  - 2nd: remove the largest item from the heap and insert that item into the sorted array

Efficient Sorting Algorithms: Heap Sort (cont.)

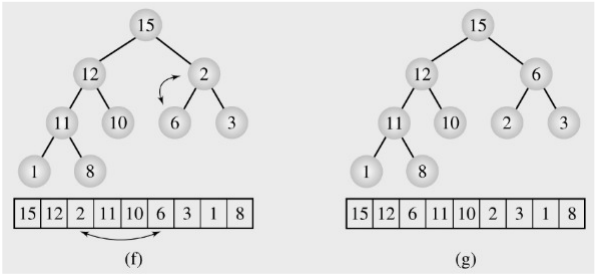


swap

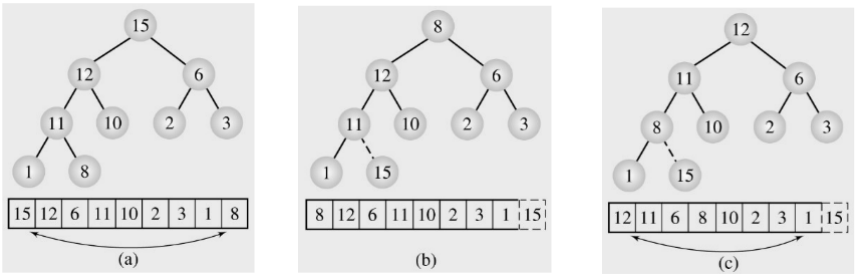
Efficient Sorting Algorithms: Heap Sort (cont.)



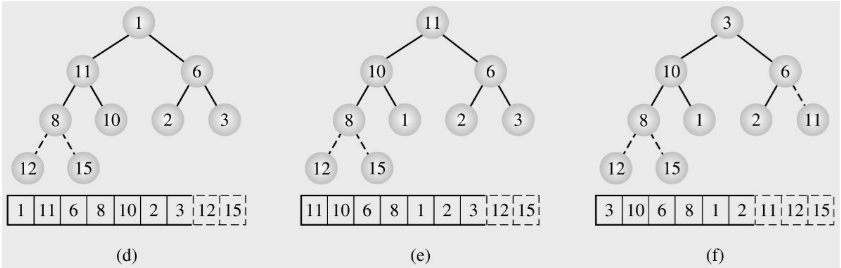
Efficient Sorting Algorithms: Heap Sort (cont.)



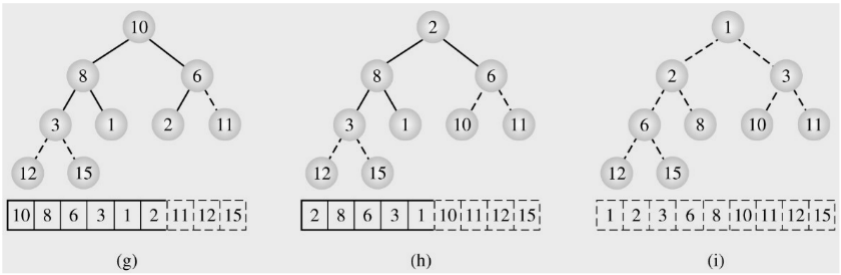
Efficient Sorting Algorithms: Heap Sort (cont.)



Efficient Sorting Algorithms: Heap Sort (cont.)



Efficient Sorting Algorithms: Heap Sort (cont.)



Efficient Sorting Algorithms: Heap Sort (cont.)

```
template<class T>
void heapsort(T data[], int n) {
  for (int i = n/2 - 1; i >= 0; --i) // create a heap;
    moveDown (data,i,n-1);
  for (int i = n-1; i >= 1; --i) {
    swap(data[0],data[i]); // move the largest item to data[i];
    moveDown(data,0,i-1);  // restore the heap property;
  }
}
```

- In the 1st step, create the heap, O(n)
- In the 2nd step,
  - exchange n – 1 times the root with the element in position i
  - restore the heap n – 1 times
    - in worst case, cause moveDown() to iterate lg i times to bring the root down to the level of the leaves, $\sum_{n=1}^{n-1}(\lg i)$, O(n log n)
- In total, O(n) + O(n log n) + O(n – 1) = O(n log n)

Efficient Sorting Algorithms: Quick Sort

- Motivation
  - shell sort
    - divide the original array into subarrays, sort those
    - divide the partially sorted array into new subarrays to be sorted
    - until the entire array is in order
  - divide-and-conquer

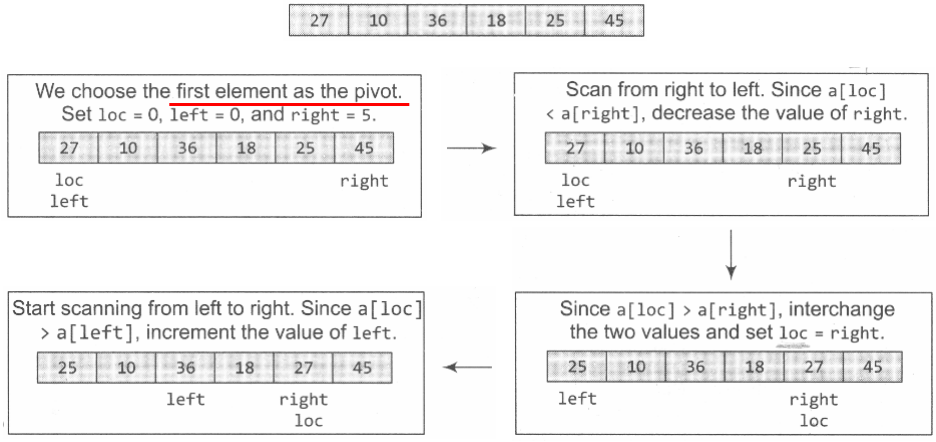
Efficient Sorting Algorithms: Quick Sort (cont.)

```
quicksort(array[])
  if length(array) > 1
    choose bound; // partition array into subarray1 and subarray2
    while there are elements left in array
      include element either in subarray1 = {el: el ≤ bound}
        or in subarray2 = {el: el ≥ bound};
    quicksort(subarray1);
    quicksort(subarray2);
```

- Two operations for partitioning the array
  - choose a bound (or **pivot**)
  - move the elements to the proper subarrays
- Choosing the bound is non-trivial
  - the goal is to have the two subarrays to be nearly equal in length

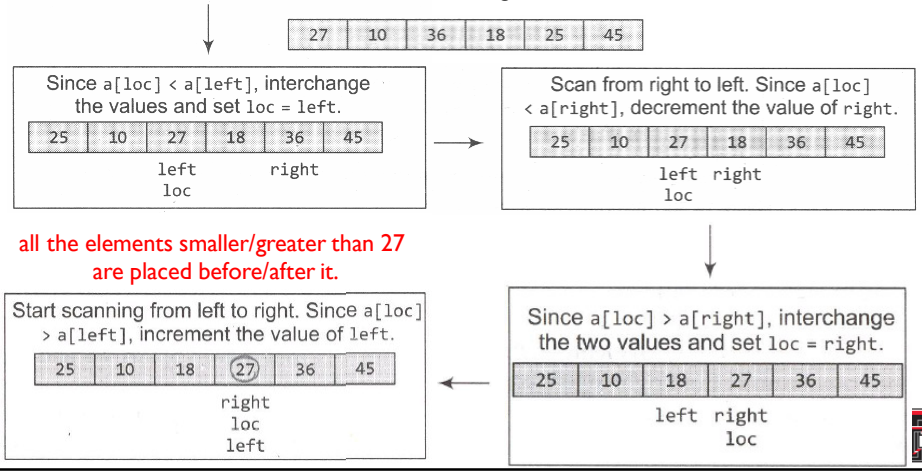
Efficient Sorting Algorithms: Quick Sort (cont.)

- e.g., sort the elements using quick sort algorithm



Efficient Sorting Algorithms: Quick Sort (cont.)

- e.g., sort the elements using quick sort algorithm

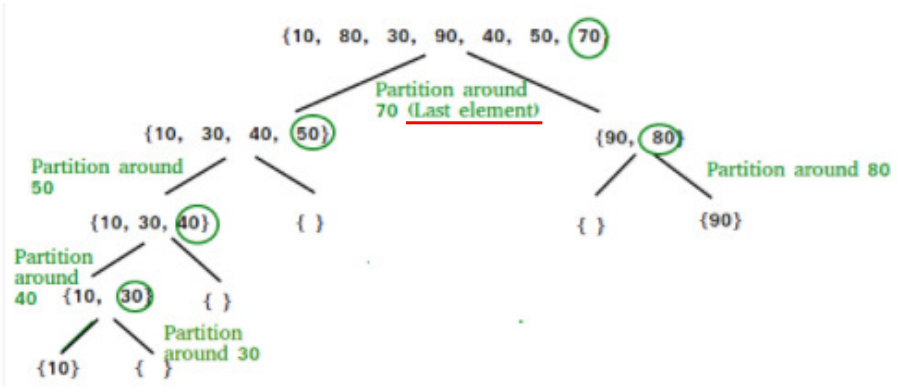


Efficient Sorting Algorithms: Quick Sort (cont.)

- Choosing the pivot element
  - e.g., randomly choose an element as pivot, pick first element as pivot, pick last element as pivot, or pick median as pivot
  - O(n log n)
  - depending on the element chosen as the pivot
- Faster than
  - insertion, selection, and bubble sorts
- Complex and massively recursive…

Efficient Sorting Algorithms: Quick Sort (cont.)

- e.g., pick the last element as a pivot



Efficient Sorting Algorithms: Merge Sort

- Motivation
  - quick sort, difficulty of the partitioning process
  - numerous techniques of choosing a bound (or pivot)
    - no assurance that any approach will result in arrays that are equal in size
- Three major operations:
  - **divide**
    - partition the n-element array to be sorted into two sub-array of n/2 elements
  - **conquer**
    - sort the two sub-arrays recursively
  - **combine**
    - merge the two sorted sub-arrays of size n/2 to produce the sorted array of n elements

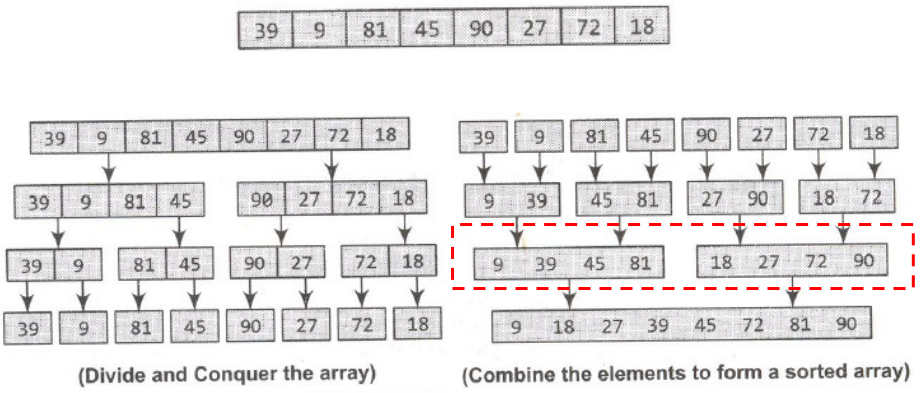
Efficient Sorting Algorithms: Merge Sort (cont.)

```
mergesort(data[])
  if data have at least two elements
    mergesort(left half of data);
    mergesort(right half of data);
    merge(both halves into a sorted list);
```

- Key operation
  - Merge the sorted halves of the array into a single array
  - these halves must be sorted, which occurs by merging the sorted halves of these halves

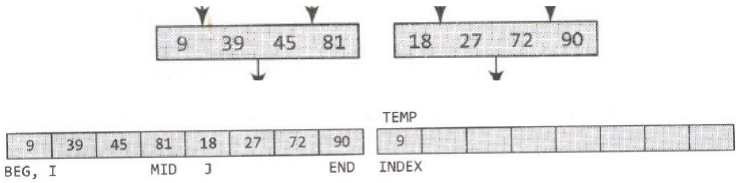
Efficient Sorting Algorithms: Merge Sort (cont.)

- e.g., sort the array using merge sort



Efficient Sorting Algorithms: Merge Sort (cont.)

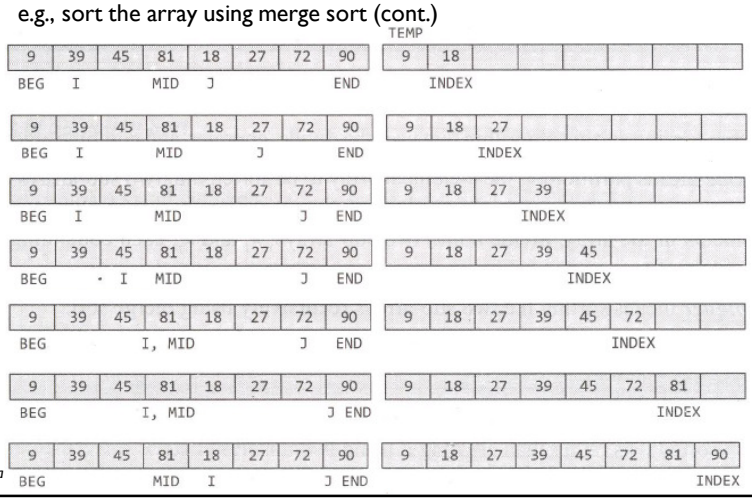
- e.g., sort the array using merge sort (cont.)



Efficient Sorting Algorithms: Merge Sort (cont.)

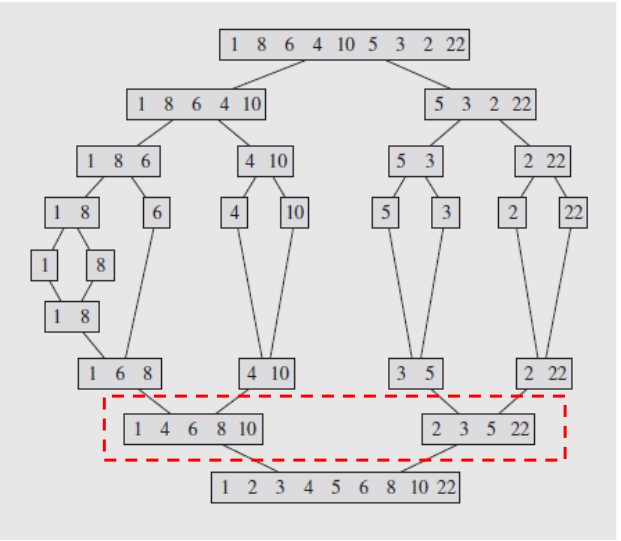
when I > MID than copy the remaining elements of the right sub-array in TEMP

- e.g., sort the array using merge sort (cont.)



Efficient Sorting Algorithms: Merge Sort (cont.)

- e.g., the array [1 8 6 4 10 5 3 2 22] sorted by merge sort
- Drawback of merge sort?
  - additional storage for merging array



Efficient Sorting Algorithms: Radix (or Bucket) Sort

- A **non-comparative** sorting algorithm
  - sort data with integer keys by grouping keys by the individual digits which share the same significant **position** and **value**
  - e.g., a list of sorted names
    - 26 radix (or 26 buckets) – 26 alphabet letters
- Frequently used in everyday applications
  - sort library cards by author
    - create a separate pile for each card based on the **first letter** of the name
    - each pile is then further sorted into smaller piles based on the **second letter** of the name

Efficient Sorting Algorithms: Radix (or Bucket) Sort

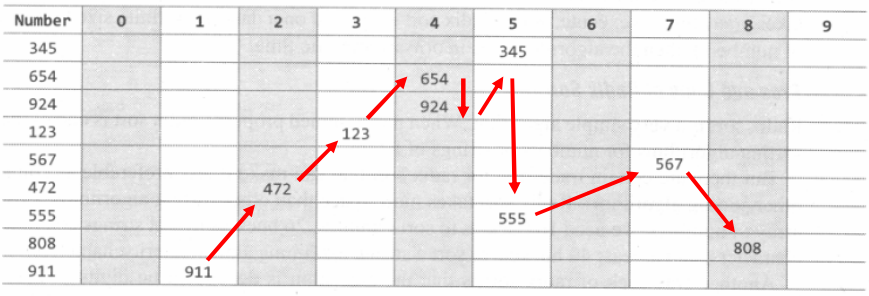
```
radixsort()
  for d = 1 to the position of the leftmost digit of longest
     number distribute all numbers among piles 0 through 9
     according to the dth digit;
     put all integers on one list;
```

- e.g., sort the list [23 123 234 567 3]
  - [123 23 234 3 567] →
  - [003 023 123 234 567]

Efficient Sorting Algorithms: Radix (or Bucket) Sort (cont.)

- Sort the numbers using radix sort

345, 654, 924, 123, 567, 472, 555, 808, 911

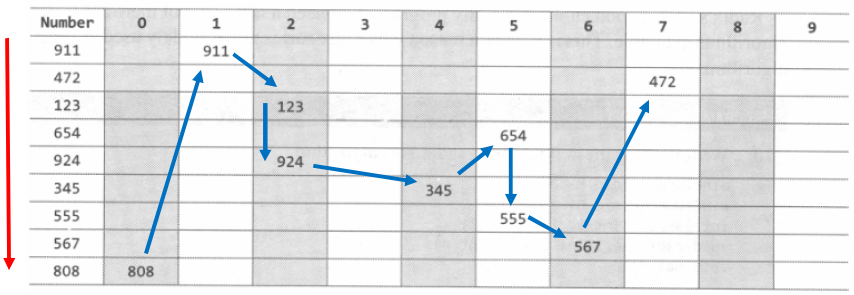


sort based on the right-most digit

Efficient Sorting Algorithms: Radix (or Bucket) Sort (cont.)

- Sort the numbers using radix sort (cont.)

345, 654, 924, 123, 567, 472, 555, 808, 911

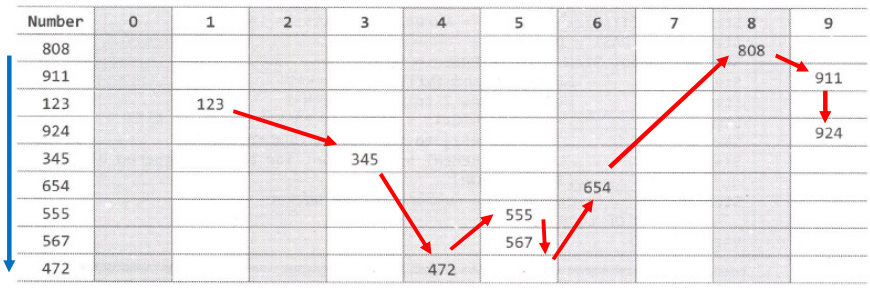


sort based on the second right-most digit

Efficient Sorting Algorithms: Radix (or Bucket) Sort (cont.)

- Sort the numbers using radix sort (cont.)

345, 654, 924, 123, 567, 472, 555, 808, 911



sort based on the third right-most digit

123, 345, 472, 555, 567, 654, 808, 911, 924.

Efficient Sorting Algorithms: Radix (or Bucket) Sort (cont.)

- Another example,

data = [10 1234 9 7234 67 9181 733 197 7 3]

								7		
				3	7234			197		
	10	9181		733	1234			67		9
piles:	0	1	2	3	4	5	6	7	8	9

data = [10 9181 733 3 1234 7234 67 197 7 9]

	9			7234						
	7			1234						
	3	10		733			67		9181	197
piles:	0	1	2	3	4	5	6	7	8	9

Efficient Sorting Algorithms: Radix (or Bucket) Sort (cont.)

data = [3 7 9 10 733 1234 7234 67 9181 197]

piles:	67									
	10									
	9									
	7	197	7234					773		
	3	9181	1234							
	0	1	2	3	4	5	6	7	8	9

data = [3 7 9 10 67 9181 197 1234 7234 733]

piles:	733									
	197									
	67									
	10									
	9									
	7									
	3	1234						7234		9181
	0	1	2	3	4	5	6	7	8	9

data = [3 7 9 10 67 197 733 1234 7234 9181]