AST20105 Data Structures & Algorithms

CHAPTER 11 – SORTING II

Instructed by Garret Lai

Before Start

- Last chapter, we learnt
  - Insertion sort
  - Selection sort
  - Bubble sort
  - Quicksort
  - Mergesort
- This chapter, we are going to learn
  - Bucket sort
  - Counting sort
    &
  - Radix sort

Lower Bound of Comparison-based Sorting

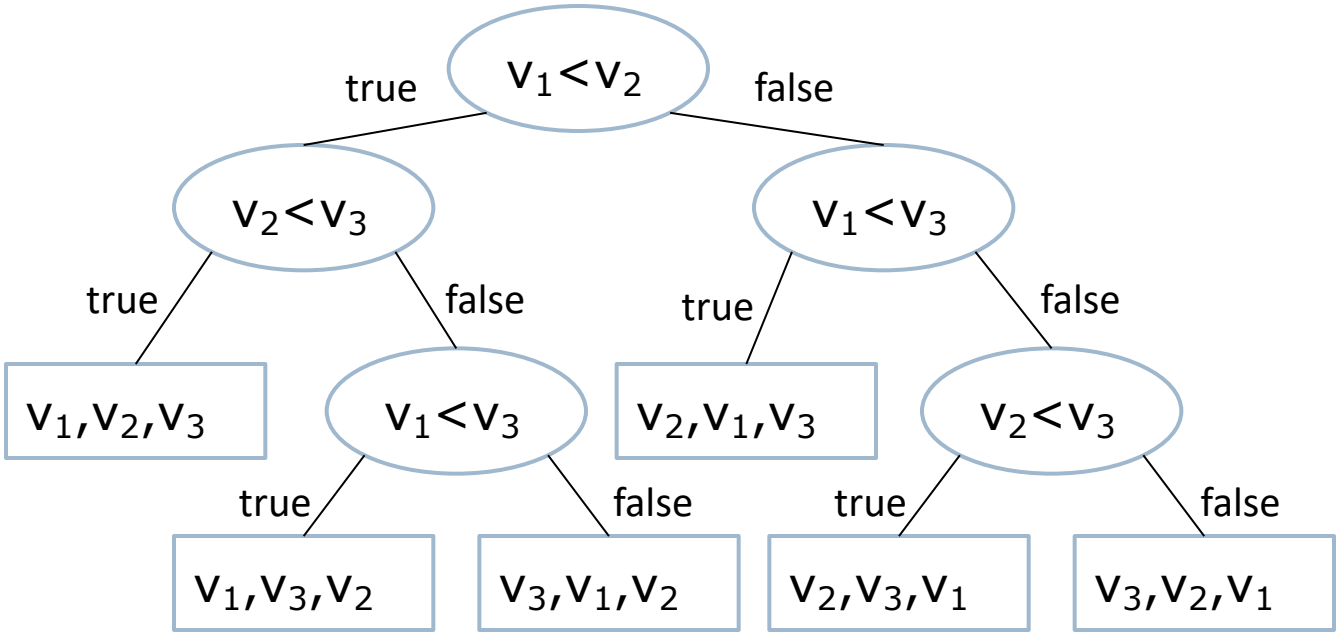
- All the sorting algorithms covered so far are all based on element comparison, i.e.
  - They sort by doing comparisons between pairs of elements

- It is interesting to derive the lower bound on the running time (i.e. number of steps) of any algorithm that does comparisons to sort elements, $v_1, v_2, v_3, \ldots, v_n$

- To do so, we represent the sequence of comparisons using "decision tree"

Lower Bound of Comparison-based Sorting

- The decision tree for sorting 3 elements, i.e. v_1, v_2 and v_3 is as follows:



- Leaf refers to a permutation of elements in sorted order
- Total number of possible order of 3 elements is 3! = 6

Comparison-based Sorting Lower Bound

- Worst case number of comparisons performed corresponds to maximal height of the tree, therefore lower bound on height refers to the lower bound on sorting

- Theorem:
  Any decision tree sorting n elements has height $\Omega(n \log n)$

Comparison-based Sorting Lower Bound

- Proof:
  - Assume elements are distinct numbers of 1 to n
  - There must be n! leaves
    (one for each permutation of n elements)
  - Tree of height h has at most 2^h leaves

$$2^h \geq n!$$

$$\log_2 2^h \geq \log_2(n!)$$

$$h \geq \log_2(n!)$$

$$= \log_2(n(n-1)(n-2)...(2)(1))$$

$$= \log_2(n) + \log_2(n-1) + \log_2(n-2) + ... + \log_2(2) + \log_2(1)$$

$$\geq \log_2(n) + \log_2(n-1) + \log_2(n-2) + ... + \log_2(n/2)$$

$$\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = \frac{n}{2}(\log_2(n) - \log_2 2) = \frac{n}{2}\log_2(n) - \frac{n}{2}$$

$$= \Omega(n \log n)$$

Therefore, any sorting algorithm based on comparisons between elements requires Ω(nlogn) comparisons.

Question

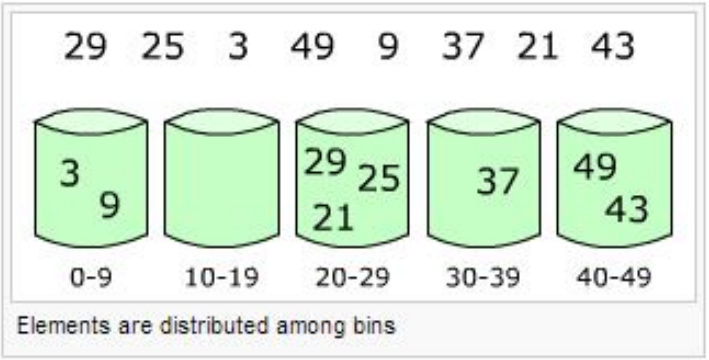
- Can we have better algorithm if we have got some information about the input data?
  - Yes!

Bucket Sort – Algorithm (Comparison-based)

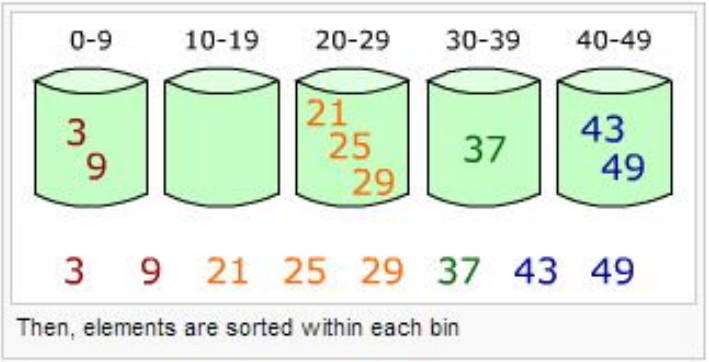
- Assume n integers to be sorted, each in range 1 to k stored in array A

- Algorithm:
  - Create an k buckets (array B, an array of linked list) so that *n* elements can be evenly inserted into the buckets. (Not always possible)
  - Put elements into those buckets.
  - Apply some sorting algorithm to sort elements in each bucket
  - Finally, take the elements out and join them to get the sorted result.

Bucket-Sort Algorithm (Comparison-based)



Elements are distributed among bins



Then, elements are sorted within each bin

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Bucket Sort (Comparison-based) Example

- Assume the elements in the following array, namely A, would like to be sorted, each in the range 1 to 12

arr[] A	22	45	87	2	64	10	12	77	59	7	81	31

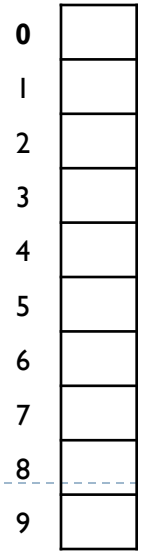
- Things to note:
  - # of elements, n = 12
  - min value = 2 and max value = 87 (so, range is about 0 – 100)
  - # of buckets = 10 (The # of buckets may vary but should be in good estimation)
- arr[] B with 10 elements (buckets) is created:
- Divider (to allocate elements into buckets)

  = ceil((max + 1) / # of buckets)

  = ceil ((88 + 1) / 10)

  = ceil(8.9)

  = 9



10

Bucket Sort (Comparison-based) Example

- Assume the elements in the following array, namely A, would like to be sorted, each in the range 1 to 12

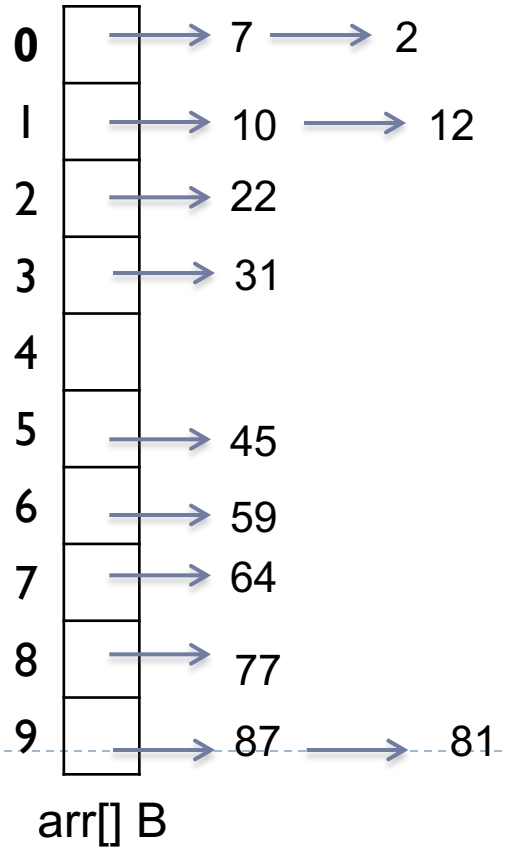
arr[] A	22	45	87	7	64	10	12	77	59	2	81	31

- Put elements in A to buckets with the following formula:

  B[j] = arr[i]

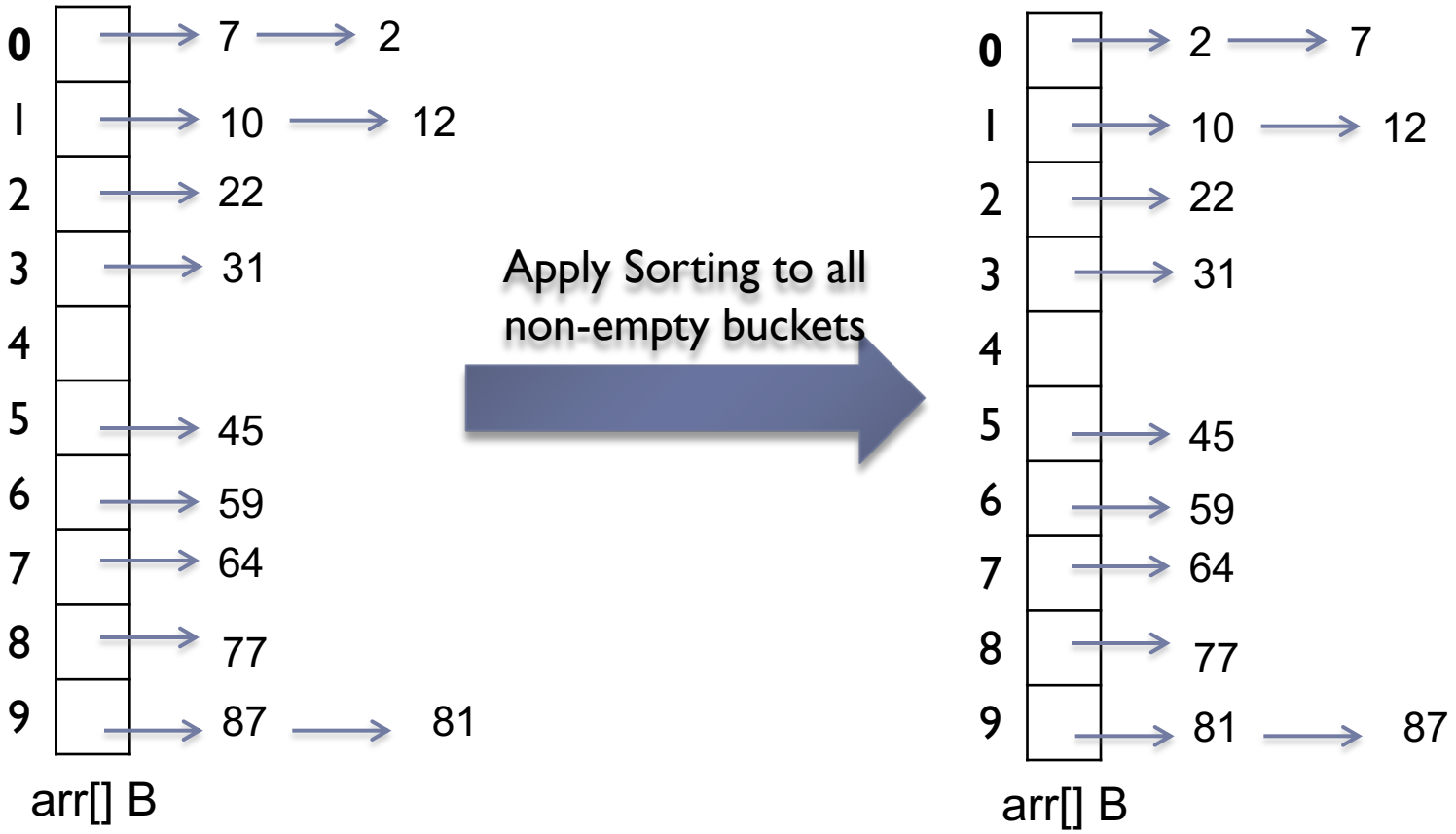
  (j = floor (arr[i] / divider)

- For example:
  - i = 0, so arr[0] = 22, divider = 9
  - j = floor(22 / 9)
  - j = floor(2.4444...)
  - j = 2



11

Bucket Sort (Comparison-based) Example



12

Bucket Sort (Comparison-based) Best Case and Worst Case Analysis

- Create an array B with k buckets and initialize all pointers
  - O(k)
- For each element in A at index I, insert it to B list until all elements in A are inserted (Assume n elements in A)
  - O(n)
- Sort through all buckets in array B with insertion algorithm
  - O(n) – Best scenario: elements are evenly distributed into buckets
  - $O(n^2)$ – Worst scenario: elements are clustered into certain bucket(s)
- Improved version:
  - Since time complexity largely depends on the scenario during sorting. We could use mergesort or quicksort to achieve O(n lg n) time complexity.

Counting Sort - Algorithm

- Assume n integers to be sorted, each in range 1 to k stored in array A
- Algorithm:
  - Create an array B of size k+1 and initialize it to 0
  - For each element in A at index i, increment B[A[i]] by 1 until all elements in A are inserted
  - Go through array B and put B[i] number of elements i back to A until the scanning is done

Counting Sort - Example

- Assume the elements in the following array, namely A, would like to be sorted, each in the range 1 to 12

A	3	4	1	2	10	12	7	8

- Step 1: Create an array B of size k+1 and initialize it to 0

B	0	0	0	0	0	0	0	0	0	0	0	0	0

- Step 2: For each element in A at index i, increment B[A[i]] by 1 until all elements in A are inserted

B	0	1	1	1	1	0	0	1	1	0	1	0	1

- Step 3: Go through array B and put all non-zero elements back to A

A	1	2	3	4	7	8	10	12

Counting Sort - C++ Code

```
void countingSort(int arr[], int n)
{
   int min, max, k = 0;
   min = max = arr[0];
   for(int i=1; i<n; i++)
   {
      min = (arr[i] < min) ? arr[i]:min;
      max = (arr[i] > max) ? arr[i]:max;
   }
   int size = max - min + 1;
   int* b = new int[size];
   for(int i=0; i<size; i++)
      b[i] = 0;
   for(int i=0; i<n; i++)
      b[arr[i] - min]++;
   for(int i=min; i<=max; i++)
      for(int j=0; j<b[i-min]; j++)
         arr[k++] = i;
   delete [] b;
}
```

Best Case and Worst Case Analysis of Counting Sort

- Create an array B of size k+1 and initialize it to 0
  - O(k)
- For each element in A at index i, increment B[A[i]] by 1 until all elements in A are inserted (Assume n elements in A)
  - O(n)
- Go through array B and put B[i] number of elements i back to A until the scanning is done
  - O(n+k)
- Total running time = O(n+k)

Counting Sort - Pros and Cons

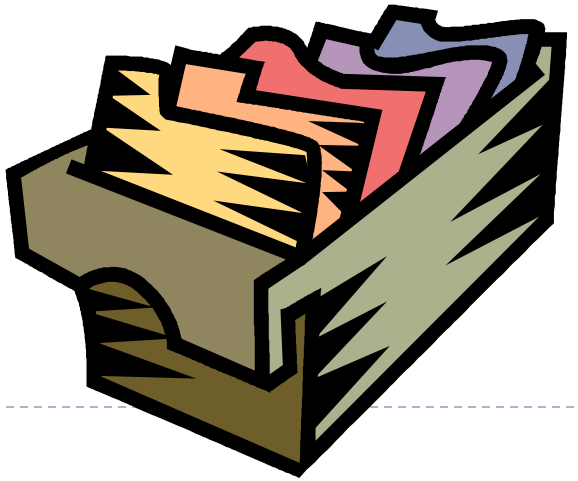
- Pros
  - Work well for arrays of values in small range
- Cons
  - Require extra memory during sorting, O(n+k)

Radix Sort

0..1..2..3..4..5..6..7..8..9

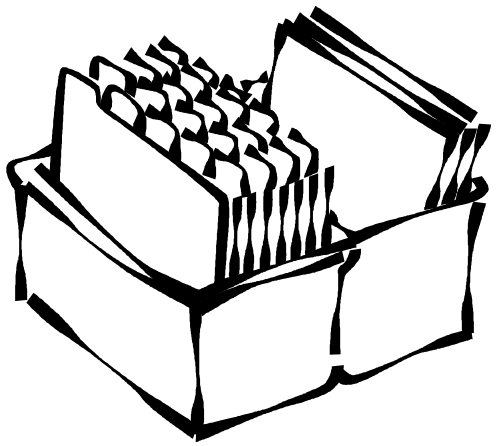
Radix Sort

- Radix sort is a popular way of sorting used in everyday life.
  - To sort library cards
    - Create as many piles of cards as letters in the alphabet, each pile containing authors whose names start with the same letter.



Radix Sort

- Radix sort is a popular way of sorting used in everyday life.
  - To sort library cards
    - Then, each pile is sorted separately using the same method, namely, piles are created according to the second letter of the authors' names.



    - This process continues until the number of times the piles are divided into smaller piles equals the number of letters of the longest name.

Radix Sort

- When sorting library cards, we proceed from left to right. However, it may be inconvenient for sorting lists of integers because they may have an unequal number of digits.
  - E.g. 23, 123, 234, 567, 3
  - If applied, the result would be
    - 123, 23, 234, 3, 567

Radix Sort

- To get around this problem, zeros can be added in front of each number to make them of equal length so that the list becomes
  - 023, 123, 234, 567, 003
  - Result: 003, 023, 123, 234, 567

Radix Sort

- Another way to sort integers is by proceeding right to left:
  - When sorting integers, 10 piles numbered 0 through 9 are created
  - And initially, integers are put in a given pile according to their rightmost digit
    - So that 93 is put in pile 3.

Radix Sort

- Then, piles are combined and the process is repeated, this time with the second rightmost digit
  - In this case, 93 ends up on pile 9.
- The process ends after the leftmost digit of the longest number is proceeded.

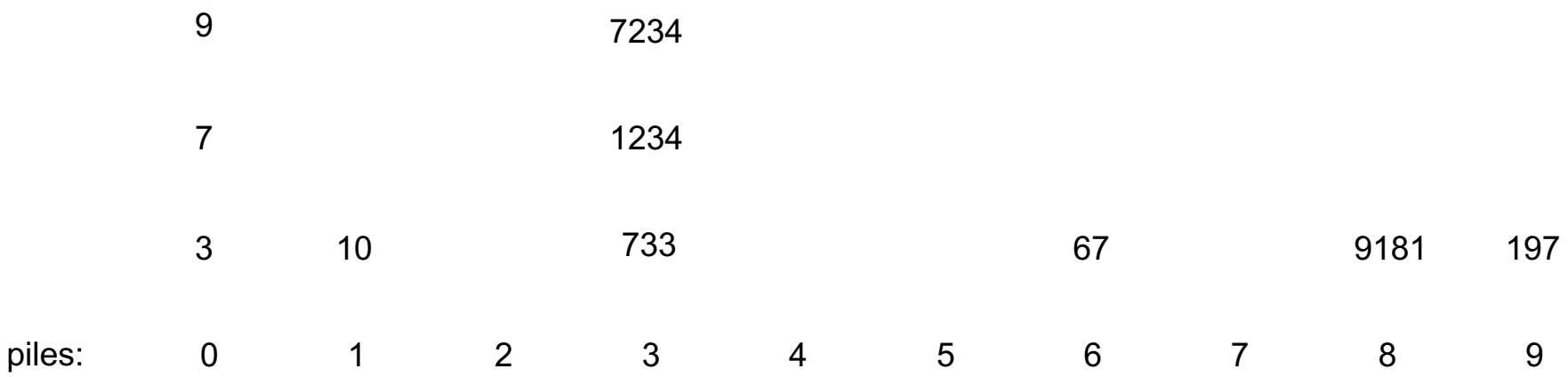
Radix Sort

- 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

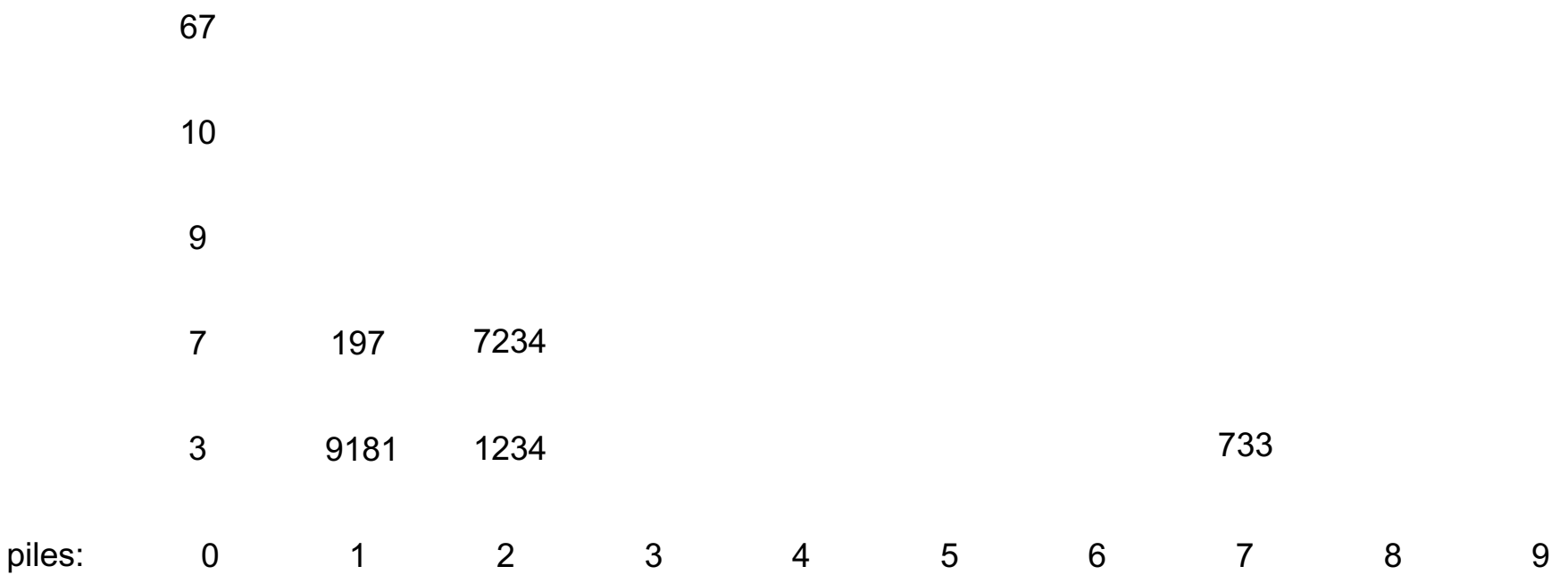
Radix Sort

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



Radix Sort

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



Radix Sort

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

Radix Sort

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

Best Case and Worst Case Analysis of Radix Sort

- k passes of counting sorts, each digit in range 0 to r
  - Each pass takes O(n+r)

- Total running time
  = O(nk + rk)
  - If r and k are constants, total running time = O(n)
  - If r and k are O(n), total running time = $O(n^2)$

Radix Sort - Pros and Cons

- Pros
  - Stable sorting algorithm, since it does not change the relative order of elements with equal keys
  - Applicable to wide range of applications
  - Relatively easy to implement with the help of counting sort

- Cons
  - Require extra memory during sorting if counting sort is used

Conclusion

Conclusion

- The following tables compares the run times for different sorting algorithms and different numbers of integers being sorted.

- They were all run on a PC.

Conclusion

		10000			20000	
	Ascending	Random	Descending	Ascending	Random	Descending
Insertion	.00	3.18	6.54	.00	13.51	26.69
Selection	5.65	5.17	5.55	20.82	22.03	22.41
Bubble	5.22	10.88	15.6	20.87	45.09	1m 2.51
Merge	.00	.05	.00	.00	.05	.05
Radix	.44	.44	.44	.82	.88	.88
Quicksort	.00	.05	.00	.00	.05	.05

Conclusion

		40000			80000	
	Ascending	Random	Descending	Ascending	Random	Descending
Insertion	.00	53.66	1m 54.35	.05	3m 46.62	7m 40.94
Selection	1m 24.47	1m 31.01	1m 30.96	5m 55.64	6m 6.30	6m 21.13
Bubble	1m 27.55	2m 59.40	4m 15.68	6m 6.91	12m 27.15	17m 14.02
Merge	.16	.22	.22	.49	.49	.44
Radix	1.7	1.76	1.76	3.46	3.52	3.52
Quicksort	.06	.11	.05	.11	.28	.17

Conclusion

- The above tables indicates that the run time for elementary sorting methods, which are squared algorithms, grows approximately by a factor of 4 after the amount of data is doubled,

- whereas the same factor for non-elementary methods, is approximately 2.

Conclusion

- The table also shows that quicksort is the fastest algorithm among all sorting methods; most of time, it runs at least twice as fast as any other algorithm.

CHAPTER 10 END