Sorting

Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

Weekly Objectives

- This week, we study various sorting algorithms
- Objectives are
  - Understanding their performances
    - Particularly, why they have such performances
  - Understanding the algorithms
  - Able to implement the algorithms
    - Selection Sort
      - Just comparison in the sequence with two index iterations
    - Merge Sort
      - Divide and conquer approach
    - Heap Sort
      - Tricky in the complexity proof
    - Quick Sort
      - Importance of a pivot
    - Counting Sort and Radix Sort
      - O(N) search

Sorting

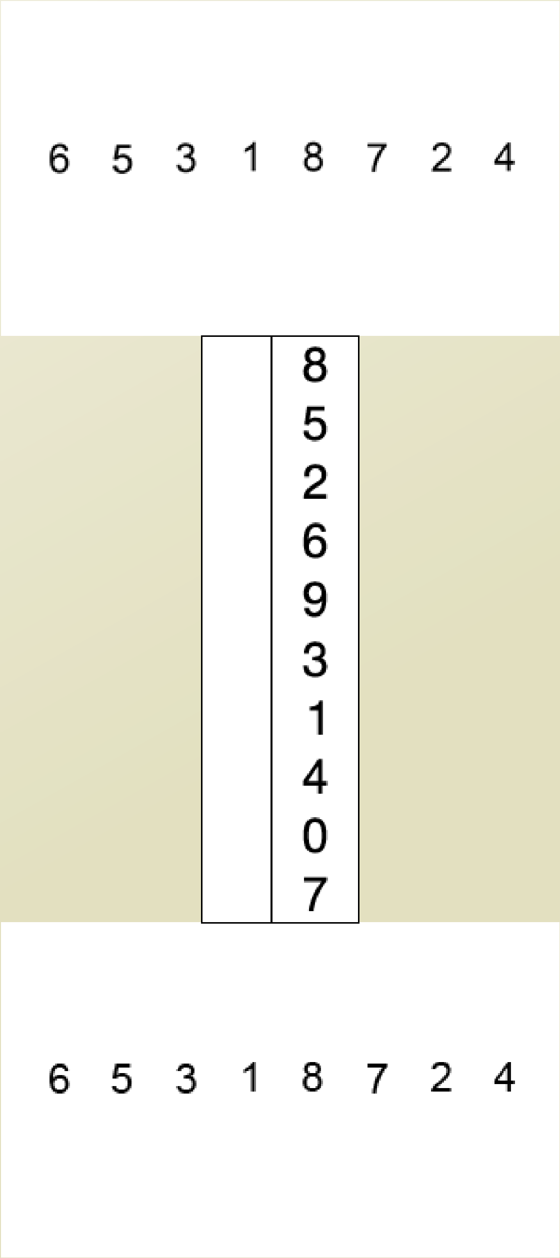
- https://www.toptal.com/developers/sorting-algorithms
- Without a manipulation on data
  - Just a chunk of data is useless to users
  - Data should be structured for
    - Users
      - Data display
      - Maybe, sorted table
    - Computers
      - Data structure
      - Maybe, heap, BST, hash….
  - Most of human decisions asks
    - Best case
    - Worst case
    - Sorting!

US | NASDAQ | AMEX | NYSE

Symbol	Name	Last Trade	Change	Volume
MSFT	Microsoft Corporation	24.87 4:00PM EST	↑ 0.57 (2.35%)	46,764,166
QQQ	PowerShares QQQ Trust, Series 1	54.72 4:00PM EST	↑ 1.84 (3.48%)	46,668,922
CSCO	Cisco Systems, Inc.	18.01 4:00PM EST	↑ 0.51 (2.91%)	45,991,635
INTC	Intel Corporation	23.46 4:00PM EST	↑ 0.73 (3.21%)	39,752,819
SIRI	Sirius XM Radio Inc.	1.77 4:00PM EST	↑ 0.02 (0.86%)	35,552,773
MU	Micron Technology, Inc.	5.62 4:00PM EST	↑ 0.12 (2.18%)	22,267,678
ORCL	Oracle Corporation	29.87 4:00PM EST	↑ 1.13 (3.93%)	21,061,860
DELL	Dell Inc.	14.98 4:00PM EST	↑ 0.76 (5.34%)	20,926,029
YHOO	Yahoo! Inc.	15.35 4:00PM EST	↑ 0.25 (1.66%)	18,913,258
SINA	Sina Corporation	59.73 4:00PM EST	↓ 3.42 (5.42%)	18,040,277
AMAT	Applied Materials, Inc.	10.40 4:00PM EST	↑ 0.24 (2.36%)	17,515,614
NVDA	NVIDIA Corporation	14.83 4:00PM EST	↑ 0.79 (5.63%)	17,265,314
CMCSA	Comcast Corporation	21.75 4:00PM EST	↑ 0.75 (3.57%)	17,094,159
GILD	Gilead Sciences, Inc.	39.80 4:00PM EST	↑ 0.52 (1.32%)	14,341,184
ATVI	Activision Blizzard, Inc	12.16 4:00PM EST	↑ 0.41 (3.49%)	14,166,509
PEIX	Pacific Ethanol, Inc.	1.24 4:00PM EST	↓ 0.02 (1.59%)	13,917,823
BRCD	Brocade Communications Systems,	5.29 4:00PM EST	↑ 0.18 (3.52%)	13,258,228

O(N^2) Sorting

- Sorting algorithm
  - Worst case O(N^2) sorting
  - Without a divide-and-conquer approach
  - Sequential comparisons with two index iterations
    - Usually there is a nested loop that ranges
      - Outer loop: from the first to the end
      - Inner loop:
        - from the outer loop's index to the end
        - Or, from the first to the outer loop's index
  - Variants
    - Insertion Sort
    - **Selection Sort**
    - Bubble Sort
  - Pros and Cons?
    - Cons: time complexity
    - Pros?
      - Easy to implement



Selection sort algorithm

- Examples of algorithms
  - Insertion, deletion, search of linked lists, stacks, queues…
  - Sorting of linked lists…
    - Various sorting methods
      - Bubble sort, Quick sort, Merge sort…
- Selection Sort(list)
  - For itr1=0 to length(list)
    - For itr2=0 to length(list)
      - If list[itr1] < list[itr2]
        - Swap list[itr1], list[itr2]
  - Return list
- This program uses
  - Data structure: List
  - Algorithm: Selection sort

```python
import random

def performSelectionSort(lst):
    for itr1 in range(0, len(lst)):
        for itr2 in range(itr1+1, len(lst)):
            if lst[itr1] < lst[itr2]:
                lst[itr1], lst[itr2] = \
                lst[itr2], lst[itr1]
    return lst

N = 10
lstNumbers = list(range(N))
random.shuffle(lstNumbers)

print(lstNumbers)
print(performSelectionSort(lstNumbers))

lstNumbers2 = [2, 5, 0, 3, 3, 3, 1, 5, 4, 2]

print(lstNumbers2)
print(performSelectionSort(lstNumbers2))
```

```
[9, 3, 6, 7, 1, 5, 0, 2, 4, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[2, 5, 0, 3, 3, 3, 1, 5, 4, 2]
[5, 5, 4, 3, 3, 3, 2, 2, 1, 0]
```

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Example of selection sort execution

- Let's observe the execution of the selection sort

```python
import random

def performSelectionSort(lst):
    for itr1 in range(0, len(lst)):
        for itr2 in range(itr1+1, len(lst)):
            if lst[itr1] < lst[itr2]:
                lst[itr1], lst[itr2] = \
                lst[itr2], lst[itr1]
    return lst
```

[2, 5, 0, 3, 3, 3, 1, 5, 4, 2]

→ (itr1 = 0, itr2=1..9) = 9 iterations

→ (itr1 = 0, itr2 = 1)

→ 2 < 5, Hit and swap!!!

→ list[0] = 5, list[1] = 2 from now

→ (itr1 = 0, itr2 = 2)

→ 5<0, No hit

→ (itr1 = 0, itr2 = 3)

→ 5<3, No hit

→ .....

→ (itr1 = 1, itr2=2..9) = 8 iterations

→ ....

→ ....

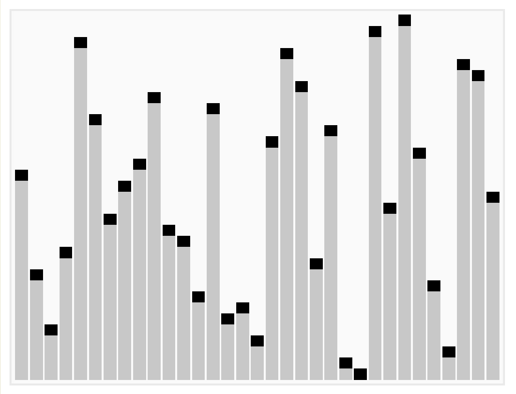
→ (itr1 = 8, itr2=9..9) = 1 iterations

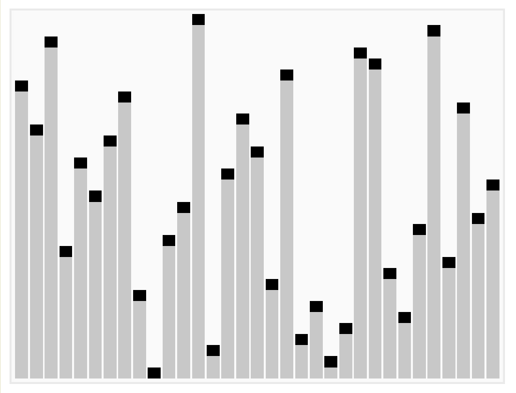
→ .....

- Total iterations
  - = 9+8+....+1
  - =45 iterations
  - $=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$

O(NlogN) Sorting

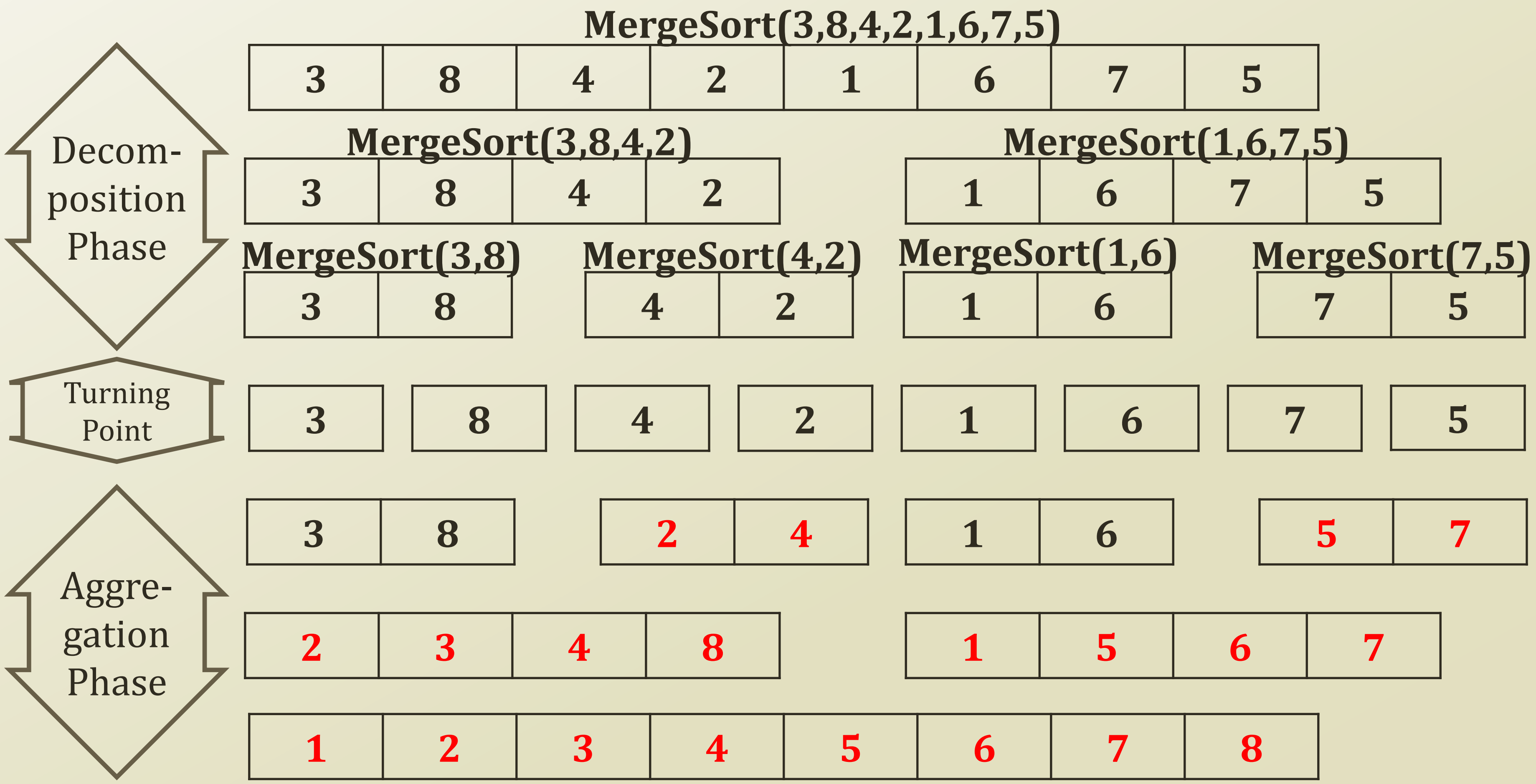
- Sorting algorithm
  - Worst case $O(N^2)$ or O(NlogN) sorting
  - Average case O(NlogN) sorting
  - With a divide-and-conquer approach
  - Divide the target sequence into multiple sequences
    - Recursively perform sorting of the sub-sequences
    - Problem is
      - How to divide
  - Variants
    - **Quick Sort**
    - **Heap Sort**
    - **Merge Sort**
  - Pros and Cons?
    - Cons: bad division leads into $O(N^2)$ time complexity
    - Pros: relatively good time complexity





Merge Sort

- Merge sort: One example of recursive programming
  - Decompose into two smaller lists
  - Aggregate to one larger and sorted list



Implementation Example: Merge Sort

```python
import random

def performMergeSort(lstElementToSort):
    if len(lstElementToSort) == 1:
        return lstElementToSort

    lstSubElementToSort1 = []
    lstSubElementToSort2 = []
    for itr in range(len(lstElementToSort)):
        if len(lstElementToSort)/2 > itr:
            lstSubElementToSort1.append(lstElementToSort[itr])
        else:
            lstSubElementToSort2.append(lstElementToSort[itr])

    lstSubElementToSort1 = performMergeSort(lstSubElementToSort1)
    lstSubElementToSort2 = performMergeSort(lstSubElementToSort2)

    idxCount1 = 0
    idxCount2 = 0
    for itr in range(len(lstElementToSort)):
        if idxCount1 == len(lstSubElementToSort1):
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
        elif idxCount2 == len(lstSubElementToSort2):
            lstElementToSort[itr] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
        elif lstSubElementToSort1[idxCount1] > lstSubElementToSort2[idxCount2]:
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
        else:
            lstElementToSort[itr] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
    return lstElementToSort
```

Decomposition

Recursion

Aggregation

Execution Code

```python
lstRandom = []
for itr in range(0, 10):
    lstRandom.append( random.randrange(0, 100))
print(lstRandom)
lstRandom = performMergeSort(lstRandom)
print(lstRandom)
```

- Code execution timing!
  - Before Recursion

  = Before Branching out
  - After Recursion

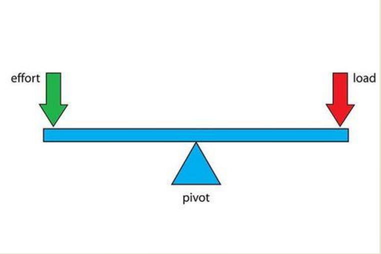
  = After Branching out

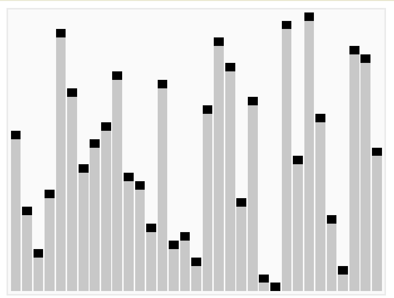
Heap sort

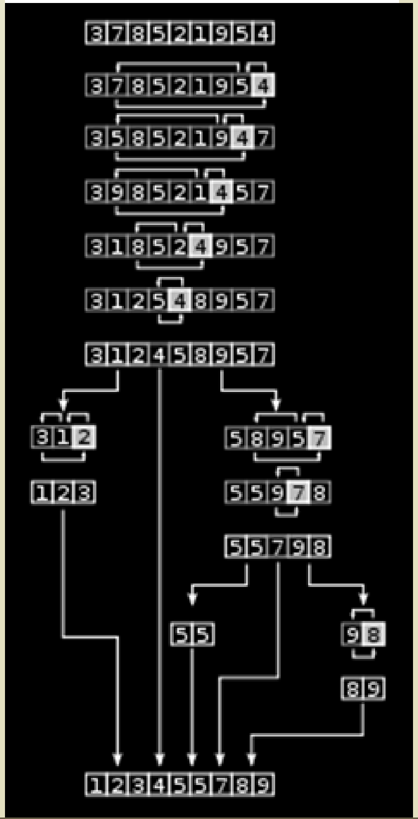
- Priority queue
  - Repeated, dequeue with the highest priority
  - = dequeue the maximum value
  - Well-utilizable for sorting
  - Particularly
    - Binary heap enables the dequeueing with O(logN)
    - For dequeueing all elements, it takes O(NlogN)
      - Same to the sorting all of the elements
- How to perform a sorting with a heap (= heap sort)
  - Given a list whose index ranges from 0 to N
  - Firstly, Build the binary heap through insertions = O(NlogN)
    - N items to insert
    - Percolation takes maximum logN
      - Is it true? Any better way?
  - Secondly, take out one element at a time = O(NlogN)
    - For itr in range(0, N):
      - Sorted[itr] = Heap.getHighestPriority()

Quick sort

- Basic idea
  - QuickSort(Sequence)
    - Given a sequence
    - Select a pivot
      - Pivot = a threshold to divide the sequence into two sub-sequences
    - Divide the sequence into two sub-sequences
      - Sequence with values less than the pivot
      - Sequence with values greater than the pivot
    - Return
      - QuickSort(sequence with less) + Pivot + QuickSort(sequence with greater)
- Merge sort forces to divide the sequence in the middle
  - Always the similar size of the sub-sequence
- This divides the sequence with the pivot selection







Importance of pivot in quick sort

- What-if the pivot is biased
  - Let's assume that
    - Pivot is always the smallest number
    - Then?
      - Just another selection sort
  - Same to the $O(N^2)$ sorting algorithms
  - Hence the pivot selection is important
  - Pivot selection approach
    - Median
    - Random
- Practically, merge sort is more preferable because?
  - Doesn't have to worry about the pivot selection

8 comparisons

3	7	8	5	2	1	9	5	4
1	3	7	8	5	2	9	5	4
1	2	3	7	8	5	9	5	4
1	2	3	7	8	5	9	5	4
1	2	3	4	7	8	5	9	5
1	2	3	4	5	7	8	9	5
1	2	3	4	5	5	7	8	9
1	2	3	4	5	5	7	8	9
1	2	3	4	5	5	7	8	9
1	2	3	4	5	5	7	8	9

Worst case pivot selection

Implementation of quick sort

```python
import random

N = 10
lstNumbers = list(range(N))
random.shuffle(lstNumbers)


def performQuickSort(seq, pivot = 0):
    if len(seq) <= 1:
        return seq
    pivotValue = seq[pivot]
    less = []
    greater = []
    for itr in range(len(seq)):
        if itr == pivot:
            continue
        elif seq[itr] > pivotValue:
            greater.append(seq[itr])
        elif seq[itr] <= pivotValue:
            less.append(seq[itr])
    ret = performQuickSort(less)+[pivotValue]+performQuickSort(greater)
    return ret

print(performQuickSort(lstNumbers))
```

Random number generation

Pivot selection: always pick the first element

Dividing the sequence into two pieces

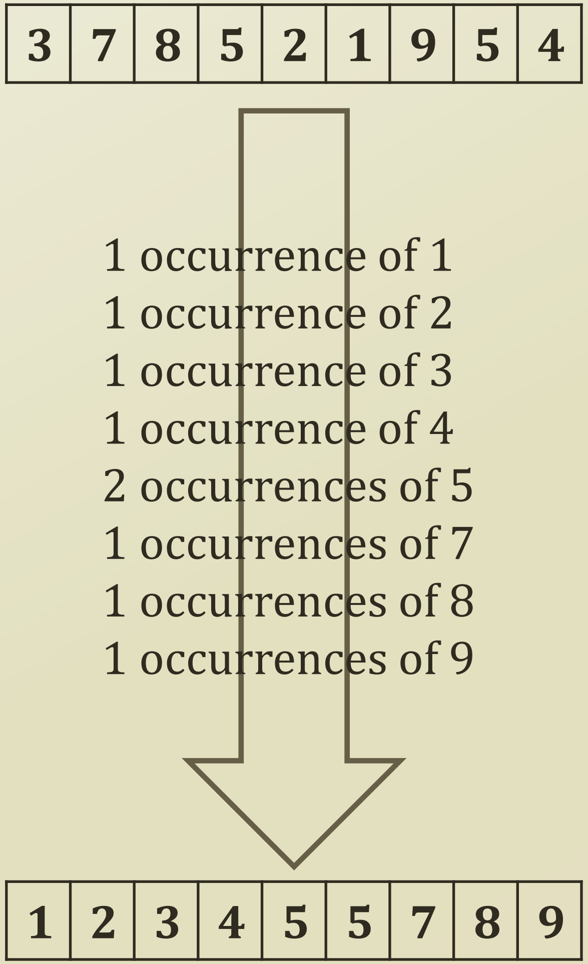
Recursive calls

O(N) Sorting

- Sorting algorithm
  - Average case O(N) sorting
  - Not comparison-based approach
    - The best performance of the comparison based approach is O(NlogN)
    - Therefore, should not be based upon comparisons
  - Rather based upon counting and numeric properties
  - Variants
    - **Radix Sort**
    - **Count Sort**
  - Pros and Cons?
    - Cons: assumptions and not comparison-based
    - Pros: best time complexity

Counting Sort

- Assumption
  - The sequence contains integers ranging from 0 to K
- Count the occurrence and produce a sequence based upon the counts
- Basic idea
  - For itr from 0 to N
    - Value = sequence[itr]
    - Count[value] = Count[value] + 1
  - For itr1 from 0 to K
    - For itr2 from 0 to Count[itr1]
      - Print itr1
- Time complexity
  - O(N+R)
  - R = the range of the sequence values
  - N = the size of the sequence



Implementation of counting sort

```python
import random

N = 10
lstNumbers = list(range(N))
random.shuffle(lstNumbers)

print(lstNumbers)

def performCountingSort(seq):
    max = -9999
    min = 9999
    for itr in range(len(seq)):
        if seq[itr] > max:
            max = seq[itr]
        if seq[itr] < min:
            min = seq[itr]
    counting = list(range(max-min+1))
    for itr in range(len(counting)):
        counting[itr] = 0
    for itr in range(len(seq)):
        value = seq[itr]
        counting[value-min] = counting[value-min] + 1
    cnt = 0
    for itr1 in range(max-min+1):
        for itr2 in range(counting[itr1]):
            seq[cnt] = itr1 + min
            cnt = cnt + 1
    return seq

print(performCountingSort(lstNumbers))
```

Random number generation

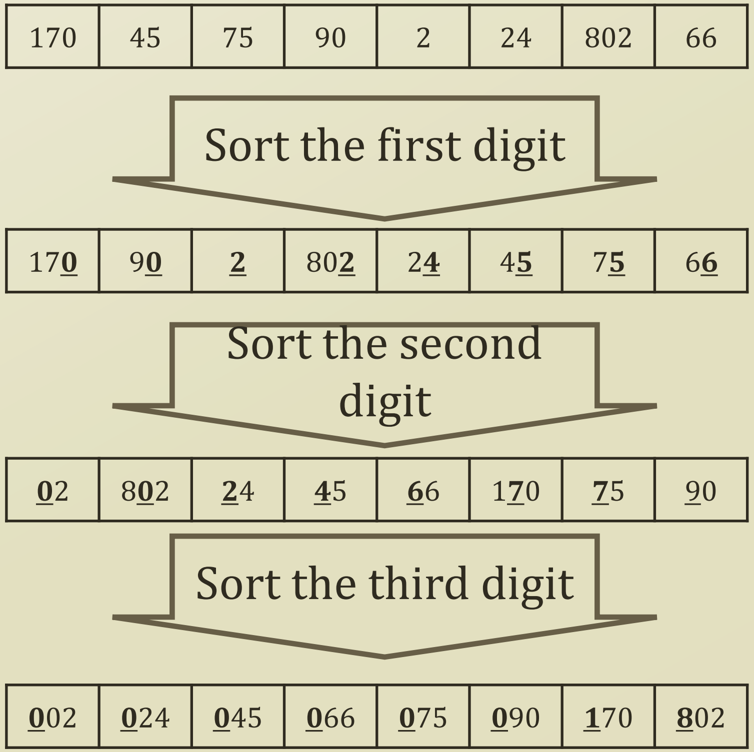
Preparing the counting space

Perform counting

Print the counted numbers

Radix Sort

- Assumption
  - The sequence contains integers
- Sort from the least important digit to the most important digit
  - Sort from 1000**2** to **1**0002
- Basic idea
  - For itr1 from 0 to D
    - Prepare a bucket list ranging from 0 to 9
    - For itr2 from 0 to N
      - digit = itr1th digit of seq[itr2]
      - Place a seq[itr2] in bucket[digit]
    - cnt = 0
    - For itr2 from 0 to 9
      - For itr3 from bucket[itr2]
        - seq[cnt] = bucket[itr2][itr3]
        - Cnt = cnt + 1
- Time complexity
  - O(ND)
  - D = the digit number of the largest value
  - N = the size of the sequence
  - Is this a good approach?



Implementation of radix sort

```python
import random
import math

N = 10
lstNumbers = list(range(N))
random.shuffle(lstNumbers)

print(lstNumbers)

def performRadixSort(seq):
    max = -999999

    for itr in range(len(seq)):
        if seq[itr] > max:
            max = seq[itr]
    D = int(math.log10(max))

    for itr1 in range(0,D+1):
        buckets = []
        for itr2 in range(0,10):
            buckets.append([])
        for itr2 in range(len(seq)):
            digit = int( seq[itr2] / math.pow(10, itr1) ) % 10
            buckets[digit].append(seq[itr2])
        cnt = 0
        for itr2 in range(0,10):
            for itr3 in range(len(buckets[itr2])):
                seq[cnt] = buckets[itr2][itr3]
                cnt = cnt + 1
    return seq

print(performRadixSort(lstNumbers))
```

Random number generation

Finding the digit number

Placing values into buckets

Printing the partially sorted values

Performance of sorting algorithms

	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$
Merge Sort	O(NlogN)	O(NlogN)
Heap Sort	O(NlogN)	O(NlogN)
Quick Sort	O(NlogN)	$O(N^2)$
Counting Sort	O(N+R)	O(N+R)
Radix Sort	O(ND)	O(ND)

- In the real world
  - Many people do not concern the time complexity of the sorting
  - Why?
    - Most of time, people rely on the database and "DESC" and "ASC"
    - Most of time, people do not give too much thought on this issue
      - Not a good idea
- You need to consider the cost of your system
  - Development
  - Maintenance

Further Reading

- Introductions to Algorithms by Cormen et al.
  - pp. 127-173