

3. Data Structure and Algorithms 2

Sorting

목차

- ▶ $O(N^2)$ Sorting
- ▶ Merge Sort
- ▶ Heap Sort
- ▶ Quick Sort
- ▶ Counting Sort
- ▶ Radix Sort

1. $O(N^2)$ Sorting

Sorting

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%)	46,666,922	
CSCO	Cisco Systems, Inc.	18.01 4:00PM EST	↑ 0.51 (2.81%)	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 (0.88%)	35,552,773	
MR	Moron Technology, Inc.	5.62 4:00PM EST	↑ 0.12 (2.18%)	22,261,678	
ORCL	Oracle Corporation	29.87 4:00PM EST	↑ 1.13 (3.83%)	21,061,880	
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 (1.68%)	18,913,258	
SINA	Sina Corporation	59.73 4:00PM EST	↓ 3.42 (5.42%)	18,040,277	
AMAT	Applied Materials, Inc.	19.40 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.	29.80 4:00PM EST	↑ 0.52 (1.32%)	14,341,184	
ATVI	Activision Blizzard, Inc.	12.16 4:00PM EST	↑ 0.41 (3.49%)	14,165,509	
PEIX	Pacific Ethanol, Inc.	1.24 4:00PM EST	↑ 0.01 (0.81%)	13,917,823	
BRCD	Brocade Communications Systems, Inc.	5.29 4:00PM EST	↑ 0.18 (3.52%)	13,256,228	

Windows 정품 인증
[설정]으로 이동하여 Windows를 정품 인증합니다.

- Sorting Algorithms Animations : <https://www.toptal.com/developers/sorting-algorithms>
- Sorting은 다양한 용도로 쓰인다.
- 이름별, 주식상황별 등
- 다양한 상황에 맞춰 sort algorithms 구현
- 문제는 수십만개를 수십만명이 동시에 들어와서 알아보려고 할 때 핸들링이 어렵다
- DB의 sort를 사용해도 된다. 그 이상의 sort algorithm을 구현하기 힘들다.
- 하지만 DB아니더라고 요소요소에 sort algorithm이 쓰일때가 많다.

1. $O(N^2)$ Sorting

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

- Order of N^2 is selection sorting 계열
- 기본적으로 효율적이고 쉽게 구현 가능
- 왜냐하면 tree structure 같은 구조가 활용되지 않음
- Index를 두개를 돌리고 sequential comparison을 함
- Time complexity 혼자서 돌려보기에는 좋으나 논문을 쓰기위해 분석을 한다든가 현실에 적용하기에는 좋지 않다.
- 하지만 쉽게 구현이 가능

1. $O(N^2)$ Sorting

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

```
import random

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

print lstNumbers

def performSelectionSort(lstNumbers):
    for itr1 in range(0, N):
        for itr2 in range(itr1+1, N):
            if lstNumbers[itr1] > lstNumbers[itr2]:
                lstNumbers[itr1], lstNumbers[itr2] = \
                    lstNumbers[itr2], lstNumbers[itr1]
        return lstNumbers

print performSelectionSort(lstNumbers)
```

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

Windows 정품 인증

본 정품 인증으로 이동하여 Windows를 정품 인증합니다.

- itr2 = 0 보다 효율적으로 하기 위해서 itr2 = itr + 1
- Maximum이 항상 앞으로 가는 algorithm
- List 뿐만 아니라 array 등 다양하게 사용할수 있다.

1. $O(N^2)$ Sorting

Example of selection sort execution

```
import random
N = 10
lstNumbers = range(N)
random.shuffle(lstNumbers)

print lstNumbers

def performSelectionSort(lstNumbers):
    for itr1 in range(0, N):
        for itr2 in range(itr1+1, N):
            if lstNumbers[itr1] > lstNumbers[itr2]:
                lstNumbers[itr1], lstNumbers[itr2] = \
                    lstNumbers[itr2], lstNumbers[itr1]
        return lstNumbers

print performSelectionSort(lstNumbers)
```

Handwritten notes:

Outer loop: $[2, 5, 0, 3, 3, 3, 1, 5, 4, 2]$
→ $(itr1 = 0, itr2 = 1..9) = 9$ 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:
◇ $= 9 + 8 + \dots + 1$
◇ $= 45$ iterations
◇ $= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{1}{2}n$
 $O(n^2)$

- $itr1 = 0, itr2 = 1..9$
- Outer loop는 N번만큼
- 코드 구현은 쉽지만 성능이 너무 안나온다.
- 총 반복이 45번

2. Merge Sort

$O(N \log N)$ Sorting

- ◇ Sorting algorithm
 - ◇ Worst case $O(N^2)$ or $O(N \log N)$ sorting
 - ◇ Average case $O(N \log N)$ 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

- $O(N^2)$ 로직보다 좀 더 나은 $O(N \log N)$ 이다
- 이 로직은 divide and conquer를 사용
- 기본적으로 Target sequence를 multiple sequences로 쪼개서 비교하는것
- Recursion이 필요
- Good division이면 average case를 달성
- 단점은 worstcase면 $O(N^2)$ 같은 성능을 낸다.
- Comparison에서는 $O(N \log N)$ 이 베스트이다.

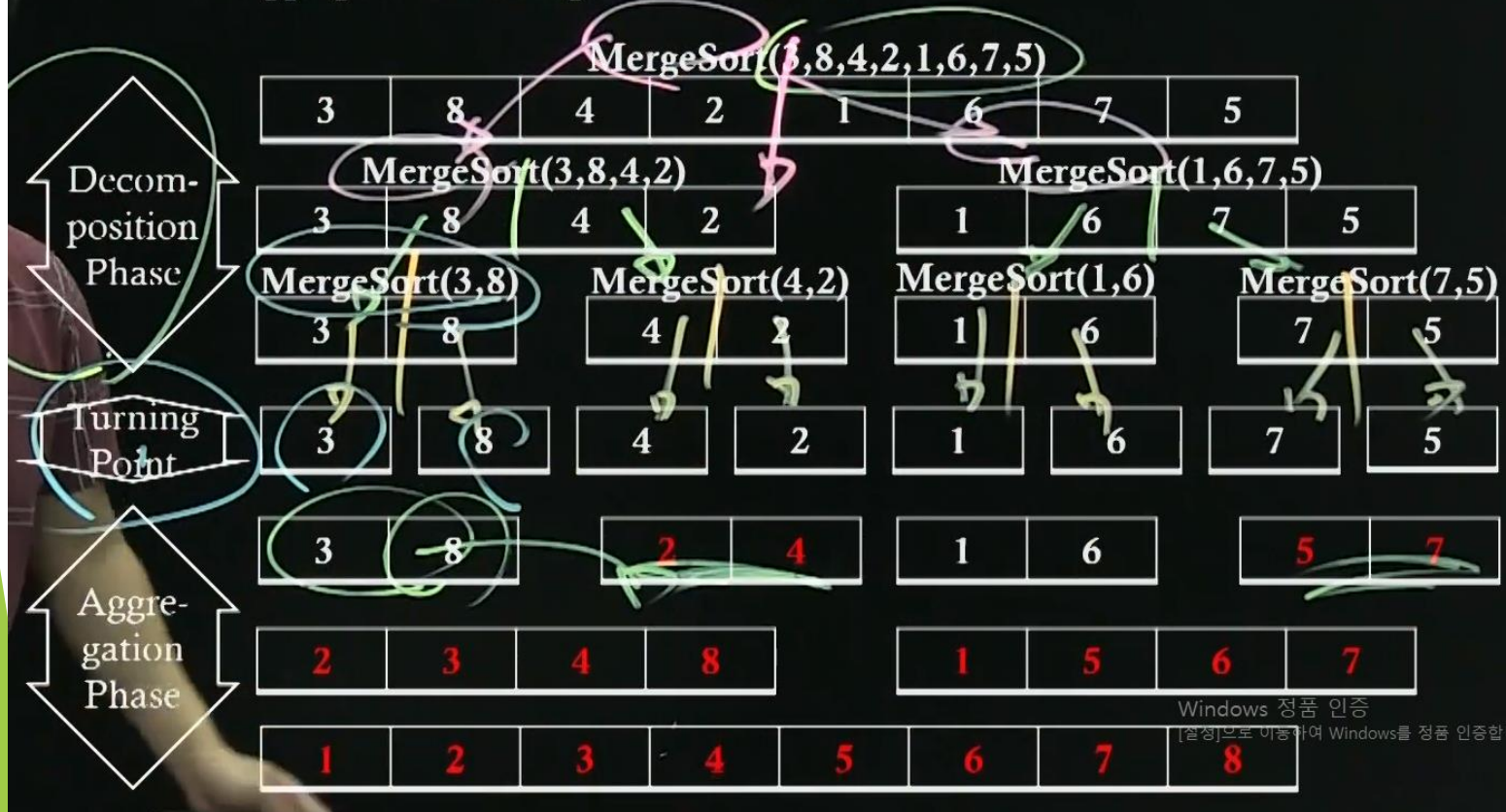
2. Merge Sort

Merge Sort

◇ Merge sort: One example of recursive programming

◇ Decompose into two smaller lists

◇ Aggregate to one larger and sorted list



- Merge Sort는 쪼갤수 없을 때까지 쪼개고 다시 aggregation하는 과정이다.
- Decom-position phase하고
- Recursion escape 으로 탈출한다.
- Sorting한다(작은것을 앞에 쓰거나 큰것을 앞에 쓴다)
- 마지막은 comparison이 필요 없다.

2. Merge Sort

Implementation Example: Merge Sort

```
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] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
        else:
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
    return lstElementToSort

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

Execution Code

Decomposition

Recursion

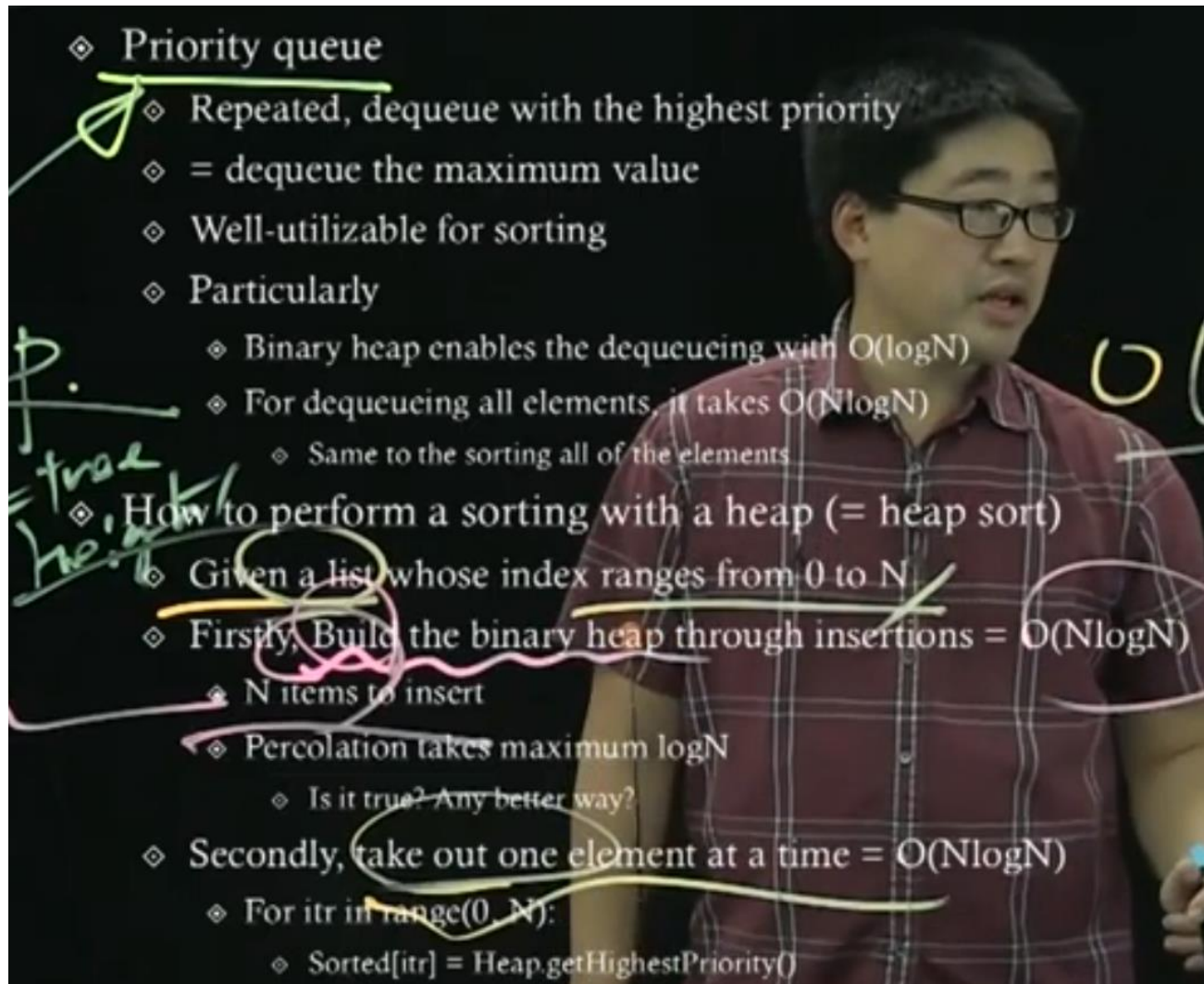
Aggregation

Windows 정품 인증
[설정]으로 이동하여 Windows를 정품 인증합니다.

- Decomposition :
- 절반으로 쪼개고 append하게 된다.
- Recursion :
- Recursive call 한다.
- Escape:
- 더 이상 쪼갤수 없으면 return한다.
- Aggregation :
- For loop 돌린다.
- 각 index의 0째부터 시작
- Elif 두번째가 비교문
- If 문 첫번째 리스트가 소진되었을때
- Elif 첫번째가 두번째 리스트가 소진 되었을때

3. Heap Sort

Heap sort



Priority queue

- Repeated, dequeue with the highest priority
- = dequeue the maximum value
- Well-utilizable for sorting
- Particularly
 - Binary heap enables the dequeuing with $O(\log N)$
 - For dequeuing all elements, it takes $O(N \log N)$
 - Same to the sorting all of the elements

How to perform a sorting with a heap (= heap sort)

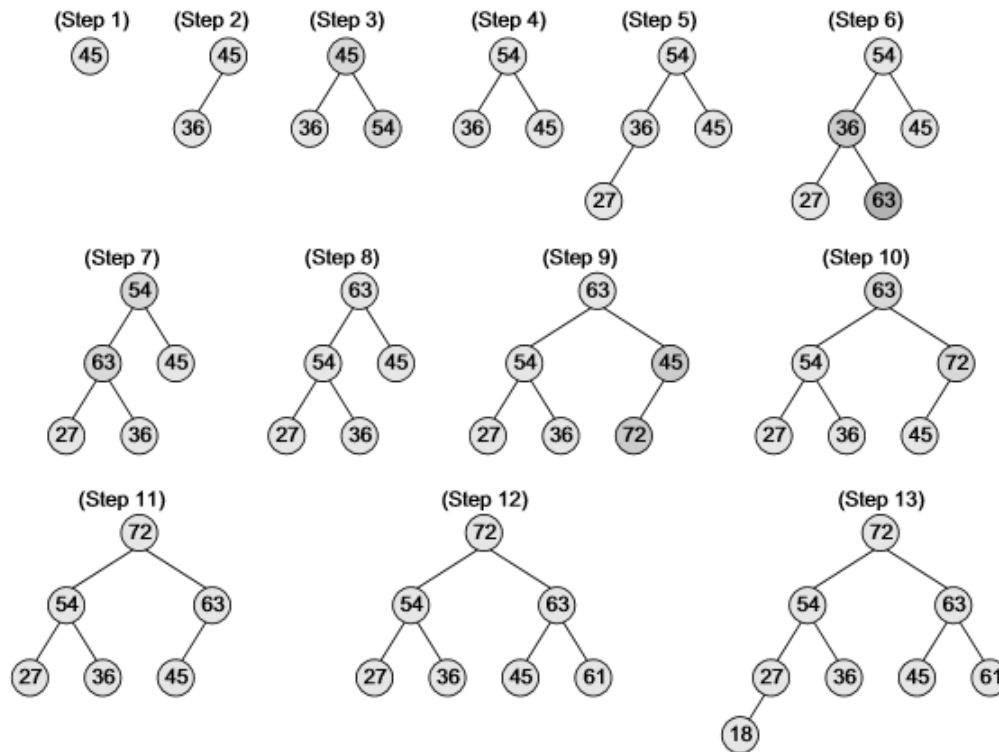
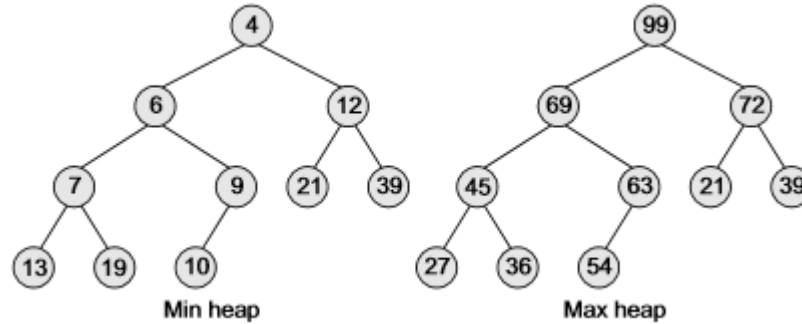
- Given a list whose index ranges from 0 to N
- Firstly, Build the binary heap through insertions = $O(N \log N)$
 - N items to insert
 - Percolation takes maximum $\log N$
 - Is it true? Any better way?
- Secondly, take out one element at a time = $O(N \log N)$
 - For itr in $\text{range}(0, N)$:
 - $\text{Sorted}[itr] = \text{Heap.getHighestPriority}()$

Handwritten notes on the blackboard:
- "P. tree height" with an arrow pointing to the "Build" step.
- "O(N log N)" written twice in yellow.

- Heap sort는 Merge sort와 동일한 성능을 내지만 structure만 좀 다르다.
- Priority queue를 활용
- Binary Heap 구조를 활용해서 Heap sort라 함
- Insert를 함으로써 binary heap을 build한다.
- Build하는 과정이 사실상 enqueue하는 것
- 이 때 percolation-up이 일어난다.
- Build가 되면 하나씩 빼면 된다.
- 빼는 과정은 percolation-down이 일어난다.

3. Heap Sort

Heap sort (보충)

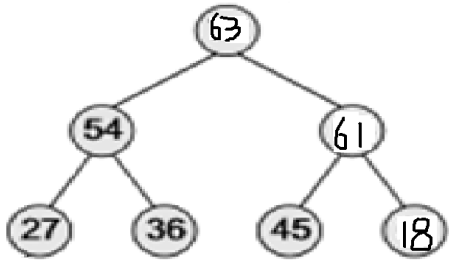
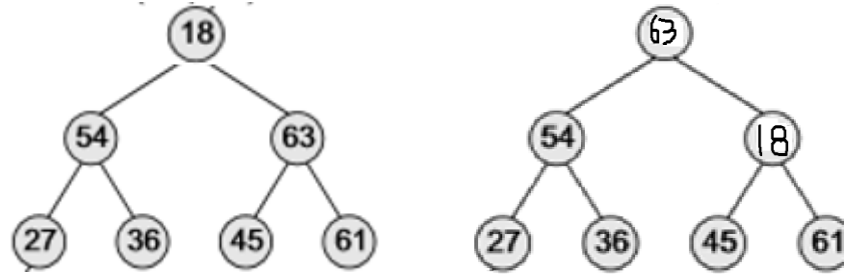
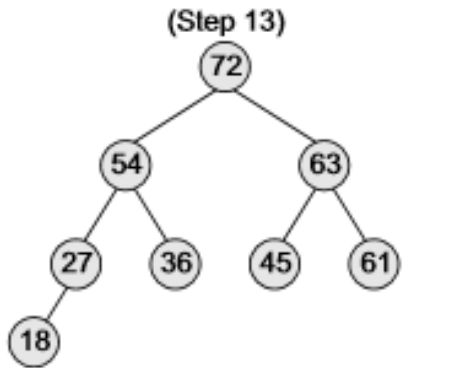
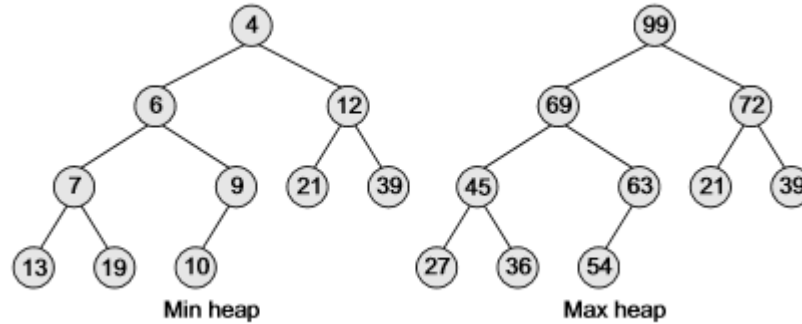


HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

- Heap sort 란?
- List의 값을 차례로 insert하여 binary heap 을 build 하고 root값을 계속 delete 하여 sort하는 것이다.
- 여기서 insert 하는 과정에 percolate-Up이 사용되고 enqueue 하는것과 같다.
- Delete 하는 과정은 percolate-Down이 사용되고 dequeue 하는것과 같다.
- Build(insert)
- List = [45, 36, 54, 27, 63, 72, 61, 18]
- List를 차례로 입력
- Min heap이면 작은 값이 위로, max면 큰 값이 위로
- Take out(delete)
- Build가 된 binary heap의 root부터 하나씩 delete
- Min heap 이면 작은값부터 정렬, max면 큰 값으로 정렬
- [72, 63, 61, 54, 45, 36, 27, 18]

3. Heap Sort

Heap sort (보충)

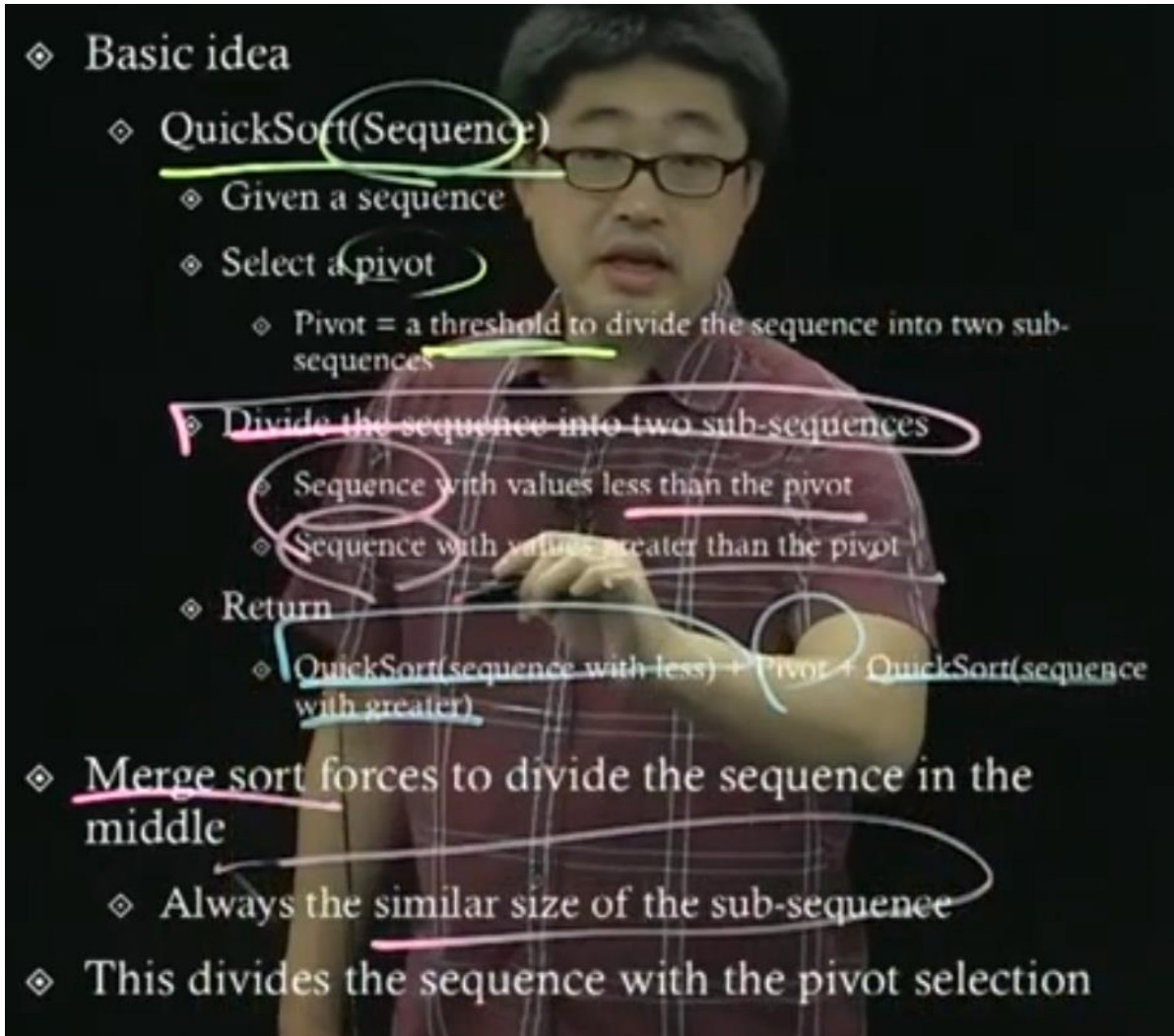


- Delete 과정
- Root 값인 72 제거하고 마지막 node에 있는 18을 root로 가져온다.
- 18의 자식들인 54와 63 중 큰값과 비교를 하여 큰 값보다 작으면 percolate-down한다.(18과 63의 자리를 바꾼다.
- 다시 18의 자식들인 45와 61 중 큰값과 비교하여 큰값보다 작으면 percolate-down 한다.(18과 61의 자리를 바꾼다.
- 위 과정을 반복한다.
- 자식을 노드가 없거나 자식의 노드보다 큰 값을 가지게 되면 escape한다.

- Heap sort 란?
- List의 값을 차례로 insert하여 binary heap을 build하고 root값을 계속 delete하여 sort하는 것이다.
- 여기서 insert 하는 과정에 percolate-Up이 사용되고 enqueue 하는 것과 같다.
- Delete 하는 과정은 percolate-Down이 사용되고 dequeue 하는 것과 같다.
- Build(insert)
- List = [45, 36, 54, 27, 63, 72, 61, 18]
- List를 차례로 입력
- Min heap이면 작은 값이 위로, max면 큰 값이 위로
- Take out(delete)
- Build가 된 binary heap의 root부터 하나씩 delete
- Min heap이면 작은값부터 정렬, max면 큰 값으로 정렬
- [72, 63, 61, 54, 45, 36, 27, 18]

4. Quick Sort

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

- Quick sort의 기본 매커니즘은 **sequence**가 들어오면 **pivot**을 선택함
- **Pivot**은 왼쪽과 오른쪽을 분리하는 기준점
- **Threshold**를 넘으면 오른쪽, 작으면 왼쪽
- **Sequence**를 **split**하는것
- 그 다음 **recursion**을 하게 됨
- **Pivot**은 중간 값을 잘 설정하는게 중요
- **Merge sort**는 항상 중간
- **Pivot**이 잘못 설정되면 양쪽의 **sequence**의 길이 차이가 많이 나게 된다.

4. Quick Sort

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

Worst case pivot
selection

Windows 12
[불량]으로 이동하여 windows를 강제로 종료합니다.

- Pivot을 가장 작은 숫자를 선택한다면?
- Selection sort랑 동일하게 됨
- Selection의 inner와 outer loop와 같음
- $O(N^2)$ sorting algorithms랑 같음
- 이럴경우 quick sort를 안하는게 낫다
- Median과 Random의 접근 방식이 있다.
- Merge sort가 기준점을 정하는데 걱정할 필요가 없어서 더 선호하기도 하다

4. Quick Sort

Implementation of quick sort

```
import random

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

print 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])
    return performQuickSort(less) + [pivotValue] + performQuickSort(greater)

print performQuickSort(lstNumbers)
```

Random number generation

Pivot selection: always pick the first element

Dividing the sequence into two pieces

Recursive calls

Windows 정품 인증 13
[설정]으로 이동하여 Windows를 정품 인증합니

- Pivot은 0로 설정(idx를 맨 처음으로 설정)
- Recursion을 해야하기 때문에 escape문이 있다.
- Pivotvalue는 항상 앞에 있는 값
- Pivotvalue보다 작은거=less, 큰거=greater 만듦
- Pivotvalue보다 크면 greater
- Pivotvalue보다 작으면 less
- Recursion이 두 파트가 일어남
- Return은 less + pivotvalue + greater로 sort가 됨
- Merge sort는 division 할때 sort 안했지만 quick sort는 division 할때 sort를 하게 됨
- Aggregation하기 전에 sort하는 것

5. Counting Sort

$O(N)$ Sorting

- ◇ Sorting algorithm
 - ◇ Average case $O(N)$ sorting
 - ◇ Not comparison-based approach
 - ◇ The best performance of the comparison based approach is $O(N \log N)$
 - ◇ 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

- $O(N)$ Sorting은 비교를 하지 않는다
- 비교를 하지 않았지만 비교를 한것과 같은 결과를 만들어 내야 $O(N \log N)$ 의 성능을 낼 수 있다.
- Counting and numeric properties기반하에 assumption 하는것
- 장점으로 가장 빠르다

5. Counting Sort

Counting Sort

전체화면을 종료하려면 Esc 을(를) 누르세요.

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

Hand-drawn diagram illustrating the Counting Sort algorithm:

Input sequence: 3 7 8 5 2 1 9 5 4

Count array (occurrences):

- 1 occurrence of 1
- 1 occurrence of 2
- 1 occurrence of 3
- 1 occurrence of 4
- 2 occurrences of 5
- 1 occurrences of 7
- 1 occurrences of 8
- 1 occurrences of 9

Sorted sequence: 1 2 3 4 5 5 7 8 9

- Counting sort는 integers만 가능
- 0 from K로 맥스 K를 알아야 함
- 각 숫자가 몇번 일어나는지 count함
- Count에 맞춰 sequeunce를 produce함
- Array를 max까지 만든다.
- 0으로 초기화한다.
- Value가 index가 돼 버린다.
- 원래 두번째 for loop는 $O(K * N)$
- K 가 값이 엄청 크다고 가정
- For irt1 from 0 to K 가 N보다 dominant하다
- Dominant 상황이 되면 K를 생각해야함
- K 가 R이 된다.
- $O(R)$ (rnage로 생각) = $O(K)$ 와 같다
- $O(N)$ 은 특정 인덱스 count
- 그래서 $O(N + R)$
- 다른 Counting sort Algorithm
- <https://www.cs.miami.edu/home/burt/learning/Csc517.091/workbook/countingsort.html>

5. Counting Sort

Implementation of counting sort

```
import random

N = 10
lstNumbers = 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 = 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

Windows 정품 인증 16
[설정]으로 이동하여 Windows를 정품

- Seq의 길이 만큼 한번 훑어서 min과 max을 알아낸다.
- 0부터 k까지의 저장소를 만들기 위함
- Counting을 0으로 초기화 한다.
- Value-min = min이 꼭 0이 아니어도 0부터 만들수 있게한 code
- 그 다음 counting 함
- Counting된 index값을 sequence로 print

6. Radix Sort

Radix Sort

Assumption

- ◆ The sequence contains integers
- ◆ Sort from the least important digit to the most important digit
 - ◆ Sort from 10002 to 10002

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]

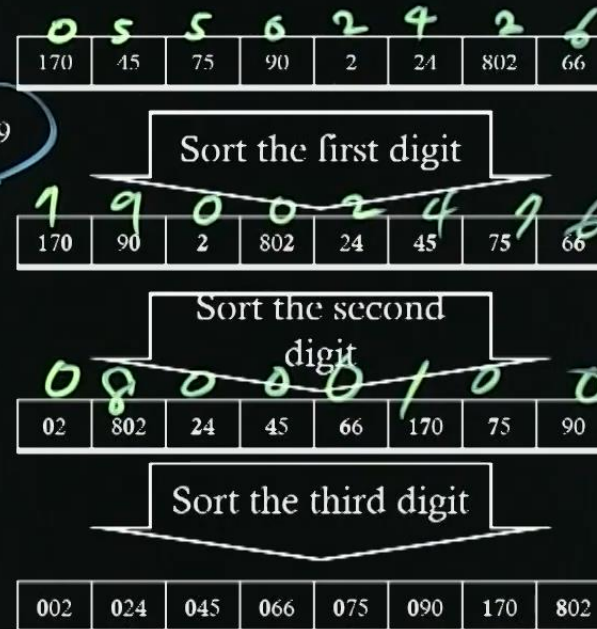
- ◆ seq[cnt] = bucket[itr2][itr3]

- ◆ 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?

Radix Sort



- Sequence는 integer여야 한다.
- Counting은 array의 길이가 길면 비효율적인데 이것을 보완하기 위해 만든것이 Radex Sort다
- 마지막 자리에 있는 digit부터 첫번째 digit까지 count하면서 sort
- Bucket에 넣고 순서대로 print하면 된다.

6. Radix Sort

Implementation of radix sort

```
import random
import math

N = 10
lstNumbers = 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

Windows 정품 인증 18
[설정]으로 이동하여 Windows를 정품 인증합니다

- Digit의 number를 알아보는 과정
- Log10을 활용
- Log10을 활용하면 소수점 자리가 나올수 있어서 int로 처리
- Outer loop는 digit를 돈다
- Buckets을 만든다.
- Pow가 10의 몇승의 자리인이 알아보는것
- Bucket에 자리수 별로 집어 넣고
- 다음 loop에서 bucket에 있는 값을 print한다.

6. Radix Sort

Performance of sorting algorithms

	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$
Heap Sort	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$
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