Data Structure - Spring 2022

9. Sorting - Lab

Walid Abdullah Al
Computer and Electronic Systems Engineering
Hankuk University of Foreign Studies

TA: **Seong Joo Kim**

Based on:
Goodrich, Chapter 6
Karumanchi, Chapter 5
Slides by Prof. Yung Yi, KAIST
Slides by Prof. Chansu Shin, HUFS

Computer Vision Lab
Hankuk University of Foreign Studies

Sorting-Review

- **Sequential methods:**
  - Insertion Sort
  - Other similar algorithms: selection sort, bubble sort
- **Divide-and-conquer methods**
  - Merge sort
    - Divide the data into two halves
    - Recur for each half until there one or less element
    - Merge the sorted halves returned from recursion
  - Quick sort
    - Choose a pivot x (usually, the first or last element is chosen)
    - Divide the data into L, E, G subsets
      - L: less than pivot, E: equal to pivot, G: greater than pivot
    - Recur for L and G
    - Merge as (L, E, G)

Algorithm Guide

with Python Code

Insertion sort

Algorithm InsertionSort(A):

Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

for k from 1 to n − 1 **do**

Insert A[k] at its proper location within A[0], A[1], . . ., A[k].

```python
def insertion_sort(A):
  """Sort list of comparable elements into nondecreasing order."""
  for k in range(1, len(A)):               # from 1 to n-1
    cur = A[k]                             # current element to be inserted
    j = k                                  # find correct index j for current
    while j > 0 and A[j-1] > cur:          # element A[j-1] must be after current
      A[j] = A[j-1]
      j -= 1
    A[j] = cur                             # cur is now in the right place
```

7	**4** A[j]	**8**	**2**	cur = 4

7 A[j]	**7**	**8**	**2**

4	**7**	**8**	**2**

4	**7** A[j-1]	**8** A[j]	**2**	cur = 8

4	**7**	**8**	**2** A[j]	cur = 2

4	**7**	**8**	**8**

4 A[j-1]	**7** A[j]	**7**	**8**

4 A[j]	**4**	**7**	**8**

2	**4**	**7**	**8**

Merge-sort: divide and conquer (DnC) method

- **Divide-and conquer is a general algorithm design paradigm:**
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S

- **The base case for the recursion are subproblems of size 0 or 1**

Merge-sort

1. ***Divide:*** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S; that is, S_1 contains the first $\lfloor n/2 \rfloor$ elements of S, and S_2 contains the remaining $\lceil n/2 \rceil$ elements.

2. ***Conquer:*** Recursively sort sequences S_1 and S_2.

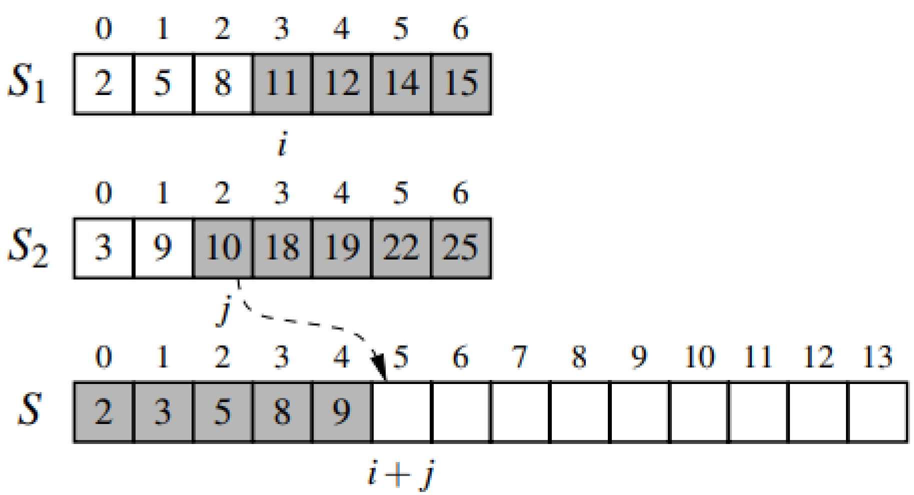
3. ***Combine:*** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

Merge-sort algorithm

```
def merge_sort(S):
  """Sort the elements of Python list S using the merge-sort algorithm."""
  n = len(S)
  if n < 2:
    return                                  # list is already sorted
  # divide
  mid = n // 2
  S1 = S[0:mid]                             # copy of first half
  S2 = S[mid:n]                             # copy of second half
  # conquer (with recursion)
  merge_sort(S1)                            # sort copy of first half
  merge_sort(S2)                            # sort copy of second half
  # merge results
  merge(S1, S2, S)                          # merge sorted halves back into S
```

How to merge S1, S2 into S?

```
def merge(S1, S2, S):
  """Merge two sorted Python lists S1 and S2 into properly sized list S."""
  i = j = 0
  while i + j < len(S):
    if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
      S[i+j] = S1[i]                    # copy ith element of S1 as next item of S
      i += 1
    else:
      S[i+j] = S2[j]                    # copy jth element of S2 as next item of S
      j += 1
```

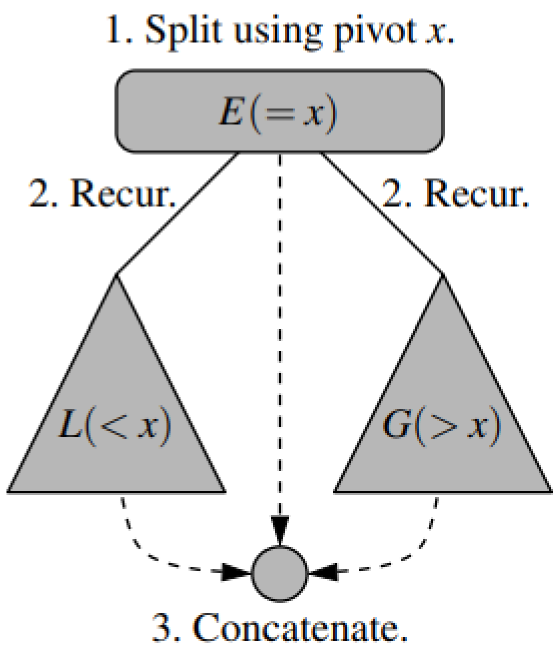


Quick-sort: another DnC method

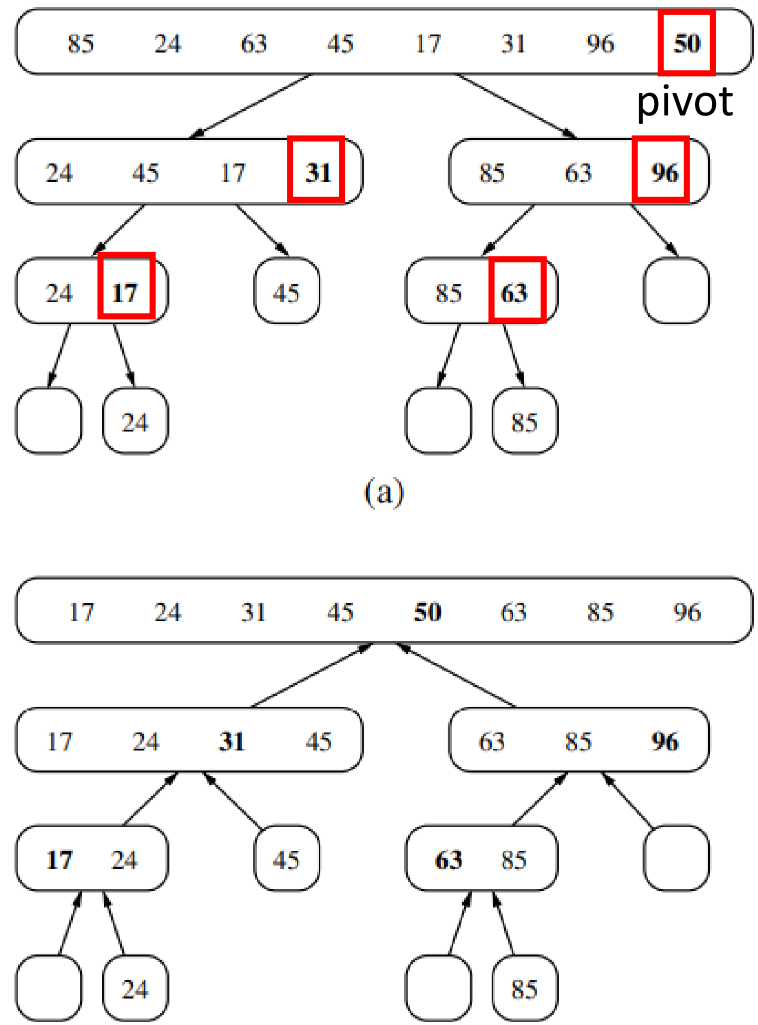
1. ***Divide:*** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the ***pivot***. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   - L, storing the elements in S less than x
   - E, storing the elements in S equal to x
   - G, storing the elements in S greater than x

   Of course, if the elements of S are distinct, then E holds just one element—the pivot itself.
2. ***Conquer:*** Recursively sort sequences L and G.
3. ***Combine:*** Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

Quick-sort: another DnC method



L-E-G 순으로 Concatenate



(a)

(b)

Quick-sort in Python

```
def quick_sort(S):
  """Sort the elements of queue S using the quick-sort algorithm."""
  n = len(S)
  if n < 2:
    return                              # list is already sorted
  # divide
  p = S.first( )                        # using first as arbitrary pivot
  L = LinkedQueue( )
  E = LinkedQueue( )
  G = LinkedQueue( )
  while not S.is_empty( ):              # divide S into L, E, and G
    if S.first( ) < p:
      L.enqueue(S.dequeue( ))
    elif p < S.first( ):
      G.enqueue(S.dequeue( ))
    else:                               # S.first() must equal pivot
      E.enqueue(S.dequeue( ))
  # conquer (with recursion)
  quick_sort(L)                         # sort elements less than p
  quick_sort(G)                         # sort elements greater than p
  # concatenate results
  while not L.is_empty( ):
    S.enqueue(L.dequeue( ))
  while not E.is_empty( ):
    S.enqueue(E.dequeue( ))
  while not G.is_empty( ):
    S.enqueue(G.dequeue( ))
```

Note that:
The book uses queue.

However, you should simply use **lists** to define L, E, G subsets.

Also, here the first element is chosen as pivot,
not the last
You may choose either one

Today's Tasks

- **On Goorm:**
  - Implement insertion sort, merge sort, and quick sort algorithms
  - Compare their experimental running time
- **On Eclass:**
  - Upload a report on "Running Time Analysis of Sorting Algorithms"
    - Introduction: briefly introduce the three sorting algorithms
    - Methods: describe how you implemented, and how you compared the running time
    - Results and Discussion: present your comparison results. Discuss your experimental observation compared with the Big-Oh running time.

The final code for your report

- **For your experiment**
  - Take input of n: the size of the array/list
  - Generate a list A with n random numbers
  - Measure time for sorting list A with different sorting algorithms
- **How to present the comparison results**
  - n vs time plot
    - Plot the running times for n=10,100,1000,10000,….
    - Use log-log plot
- **Bonus point**
  - Also, plot the Big-Oh times and actual running times in the same graph to compare