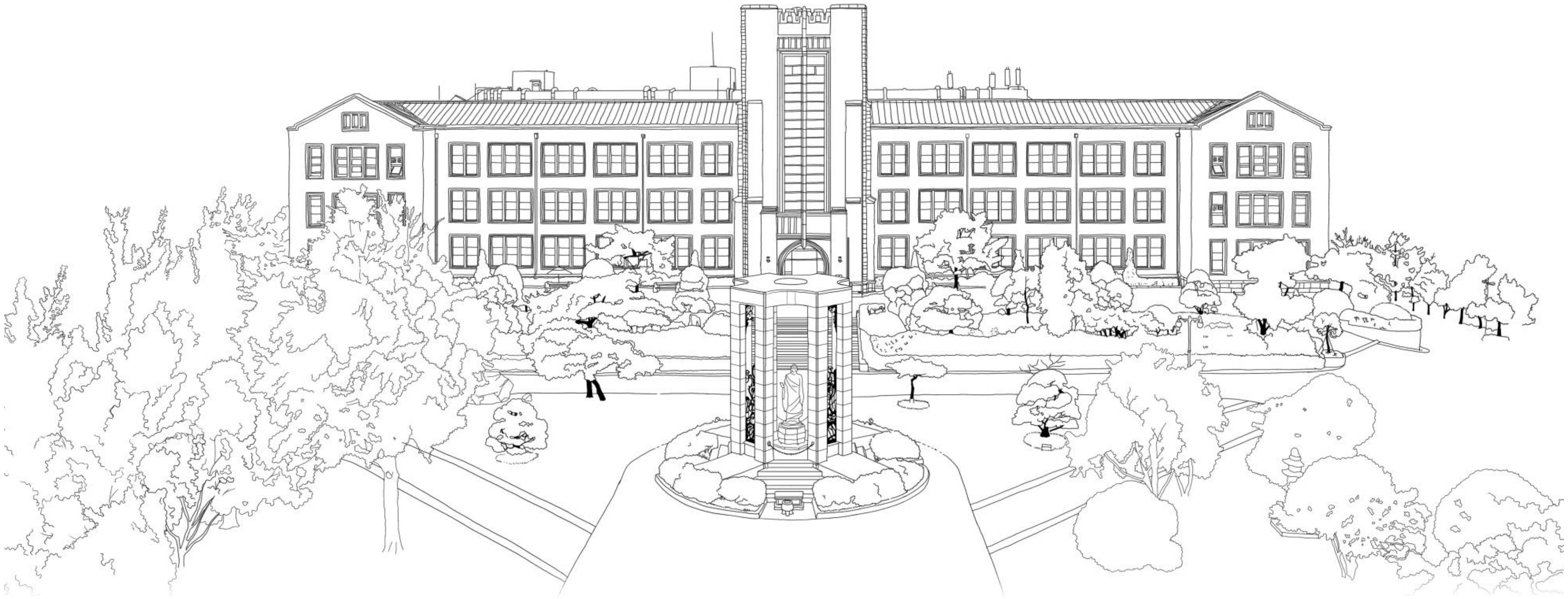


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

Contents

❖ Chapter.1 Foundations

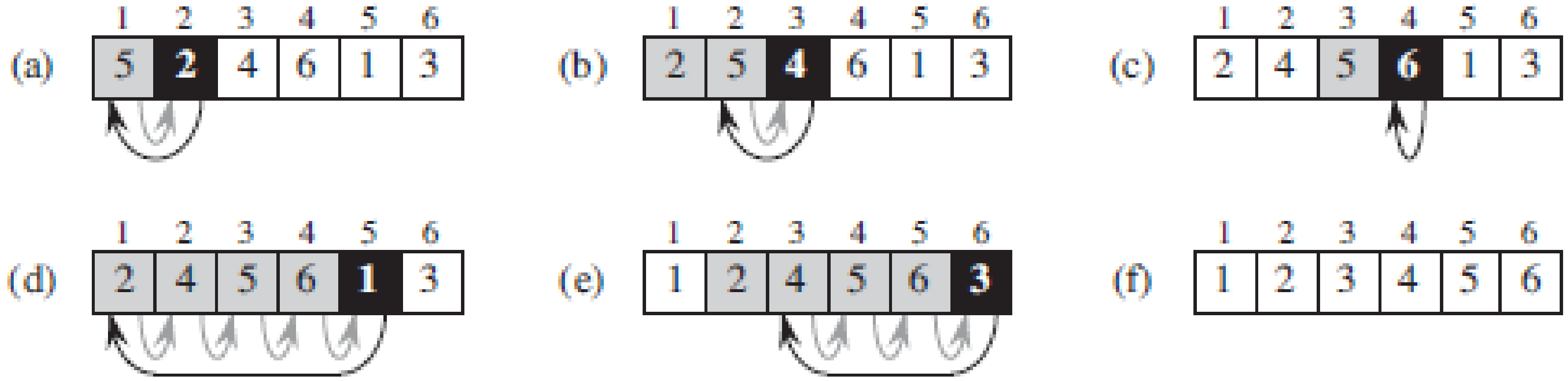
Getting Started

❖ Insertion Sort

- How does it work?

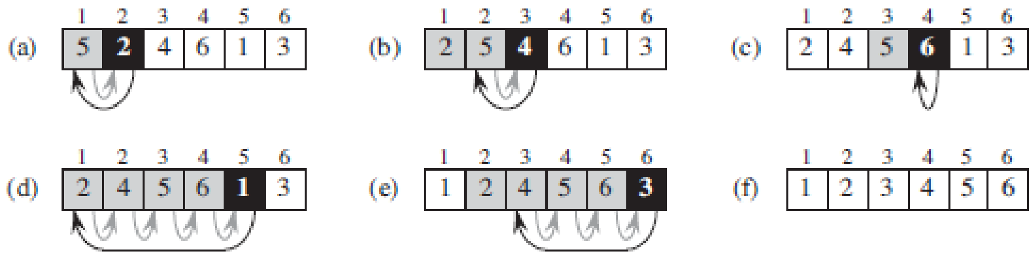
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.



❖ Insertion Sort

- Pseudocode




```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- ✓ Line 1~2: initialize Key value
- ✓ Line 5: compare Key value with shaded rectangle values
- ✓ Line 6~7: move shaded rectangles one position to the right
- ✓ Line 8: insert key value

Getting Started

❖ Insertion Sort

- Loop Invariant
  - ➢ Definition : Loop Invariant is a property of a program loop that is true before each iteration

  - ➢ Property
    - ✓ **Initialization** : It is true prior to the first iteration of the loop.
    - ✓ **Maintenance** : If it is true before an iteration of the loop, it remains true before the next iteration
    - ✓ **Termination** : When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

  - ➢ Similarity to mathematical induction

❖ Insertion Sort

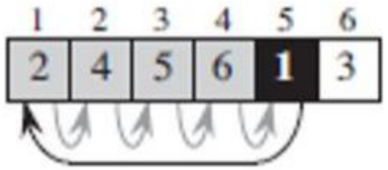
- Loop invariants of Insertion sort
  - ➢ At the start of each iteration of the for loop, the subarray $\boldsymbol{A[1..j-1]}$ consists of the elements originally in $\boldsymbol{A[1..j-1]}$, but in sorted order

Initialization

- ✓ $j = 2$
- ✓ $A[1..j-1] = A[1]$
- ✓ $A[1]$ is the original element

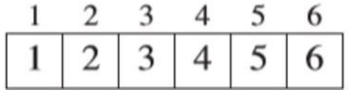
Maintenance

- ✓ Algorithm looks for appropriate position for $A[j]$, at which time $A[1..j]$ holds original elements & sorted
- ✓ Incrementing j for next iteration preserves the loop invariant



Termination

- ✓ $j = n + 1$
- ✓ $A[1..n]$ consists of the elements originally in $A[1..n]$ & sorted
- ✓ Entire array is sorted
  → loop invariants correct
  → algorithm is correct



❖ Analyzing Algorithms

- Input Size & Running Time
  - ➢ The time taken by an algorithm grows with the size of the input so describe the running as a function of the size of its input.
  - ➢ Size of Input
    - ✓ The best notion for input size depends on the problem being studied.
    - ✓ Sorting or discrete Fourier transform : number of items in the input
    - ✓ Multiplying two integers : total number of bits needed
    - ✓ Input to an algorithm is a graph : the numbers of vertices and edges
  - ➢ Running Time
    - ✓ The number of primitive operations or "steps" executed
    - ✓ Assume that each execution of the ith line takes time c_i which is constant

❖ Analyzing Algorithms

- Analysis of Insertion Sort
  - ➢ Cost : execution time of i th line
  - ➢ Times : number of times i th line executes
  - ➢ T_j : number of times the while loop test in line 5 is executed for value of j

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $\quad i = j-1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $\quad\quad i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

❖ Analyzing Algorithms

- Analysis of Insertion Sort
  - ➢ $\boldsymbol{T(n)}$: the running time of Insertion Sort on an input of n values
    - ✓ Sum the products of cost and times

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, . \end{aligned}$$

  - ➢ Input array is already sorted (Best Case)
    - ✓ T_j = 1 for j = 2, 3, ⋯, n

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \, . \end{aligned}$$

    - ✓ $T(n) = an + b$, it is a linear function

❖ Analyzing Algorithms

- Analysis of Insertion Sort
  - Input array sorted backwards (Worst Case)
    - ✓ $T_j = j$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- ✓ $T(n) = an^2 + bn + c$, it is quadratic function

Getting Started

❖ Analyzing Algorithms

- Worst-case & Average-case analysis
  - ➢ Concentrate on finding only the worst-case running time
    - ✓ The worst-case running time of an algorithm gives us an upper bound on the running time for any input
      1. Provides a guarantee that the algorithm will never take any longer
    - ✓ For some algorithms, the worst case occurs fairly often
      1. Searching a database for information when the information is not present in database
    - ✓ The "average case" is often roughly as bad as the worst case
      1. Half of the elements in A[1..j-1] are less than A[j] (key value)
      2. Tj = j/2 -> average-case running time is quadratic function

❖ Analyzing Algorithms

- Order of Growth
  - ➢ Make more simplifying abstraction and consider the term that really interests us

  - ➢ Abstractions made so far
    - ✓ Actual cost of each statements denoted as c_i
    - ✓ Worst case for Insertion sort : $an^2 + bn + c$

  - ➢ Additional Abstraction
    - ✓ Consider the leading term of the formula & ignore the coefficient
    - ✓ Insertion Sort : n^2
    - ✓ Worst case running time of Insertion Sort : $\Theta(n^2)$

❖ Designing Algorithms

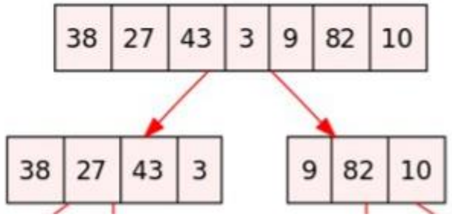
- Divide and Conquer Approach
  - Definition : break the problem into several subproblems that are similar to the original problem, solve the subproblems recursively, and combine these solutions to create a solution to the orignal problem
  - Steps
    - ✓ **Divide** : Divide the problem into a number of subproblems that are smaller instances of the same problem
    - ✓ **Conquer** : Conquer the subproblems by solving them recursively. If subproblem sizes are small enough, just solve them
    - ✓ **Combine** : Combine the solutions to the subproblems into the solution for the original problem

❖ Designing Algorithms

- Divide-and-conquer approach of Merge Sort

Divide

✓ Divide the n-element sequence to be sorted into two subsequences of $n/2$

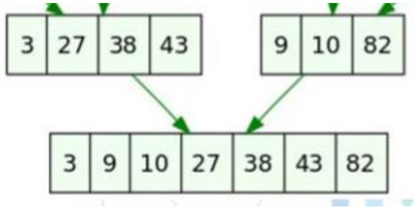


Conquer

✓ Sort the two subsequences recursively using merge sort

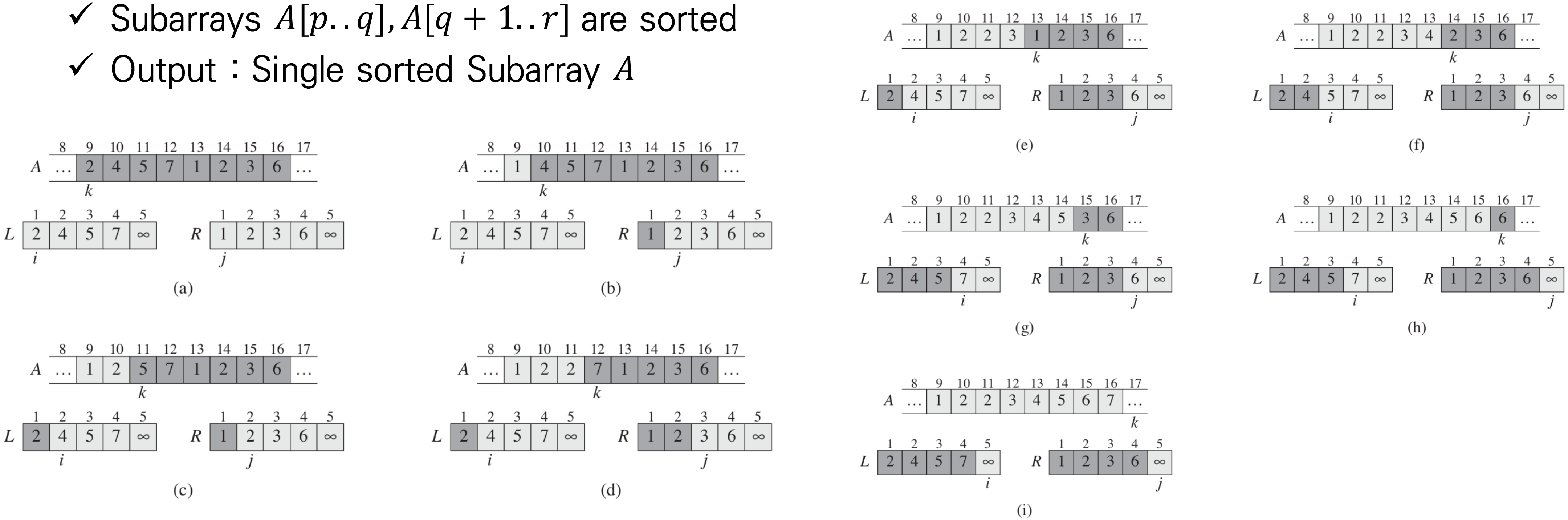
Combine

✓ Merge the two sorted subsequences to produce the sorted answer



❖ Designing Algorithms

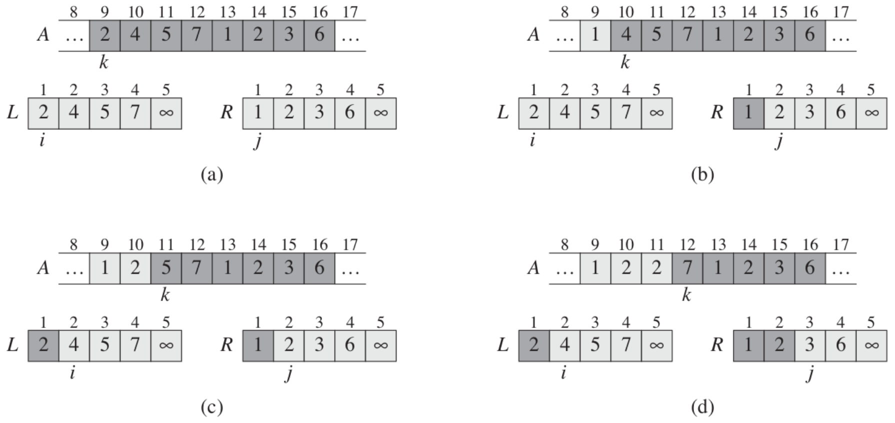
- Divide-and-conquer approach of Merge Sort
  - ➢ The key operation of the Merge sort algorithm is the Merge procedure
  - ➢ $\boldsymbol{MERGE(A, p, q, r)}$
    - ✓ Input : Subarray A + indices p, q, r ($p \le q < r$)
    - ✓ Subarrays $A[p..q], A[q+1..r]$ are sorted
    - ✓ Output : Single sorted Subarray A



❖ Designing Algorithms

- Divide-and-conquer approach of Merge Sort

```
MERGE(A, p, q, r)
1   n1 = q - p + 1
2   n2 = r - q
3   let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i - 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```



- ✓ Line 1~2 : computes the length of the subarrays
- ✓ Line 3~7 : create arrays L, R and copy the values from A
- ✓ Line 8~9 : put the sentinel values at the ends of the arrays
- ✓ Line 10~11 : initialize indices
- ✓ Line 12~17 : compare $L[i], R[j]$ and copy back into A

→ $MERGE(A, p, q, r)$ takes $\Theta(n)$ $times$

❖ Designing Algorithms

- Loop invariants of MERGE
  - ➢ At the start of each iteration of the for loop, the subarray $\boldsymbol{A[p..k-1]}$ contains the $\boldsymbol{k-p}$ smallest elements of $\boldsymbol{L[1..n1+1]}$ and $\boldsymbol{R[1..n2+1]}$ in sorted order
  - ➢ $\boldsymbol{L[i]}$ and $\boldsymbol{R[j]}$ are the smallest elements of their arrays that have not been copied back into $\boldsymbol{A}$

Initialization

- ✓ k = p
- ✓ $A[p..k-1]$ is empty
- ✓ i, j = 1
- ✓ $L[i], R[j]$ are the smallest elements because L, R are sorted subarray

Maintenance

- ✓ $A[p..k-1]$ contains the $k-p$ smallest elements
- ✓ For $L[i] \le R[j]$, when $L[i]$ copied back into $A[k]$, subarray $A[p..k]$ contains $(k-p+1)$ elements which are smallest elements

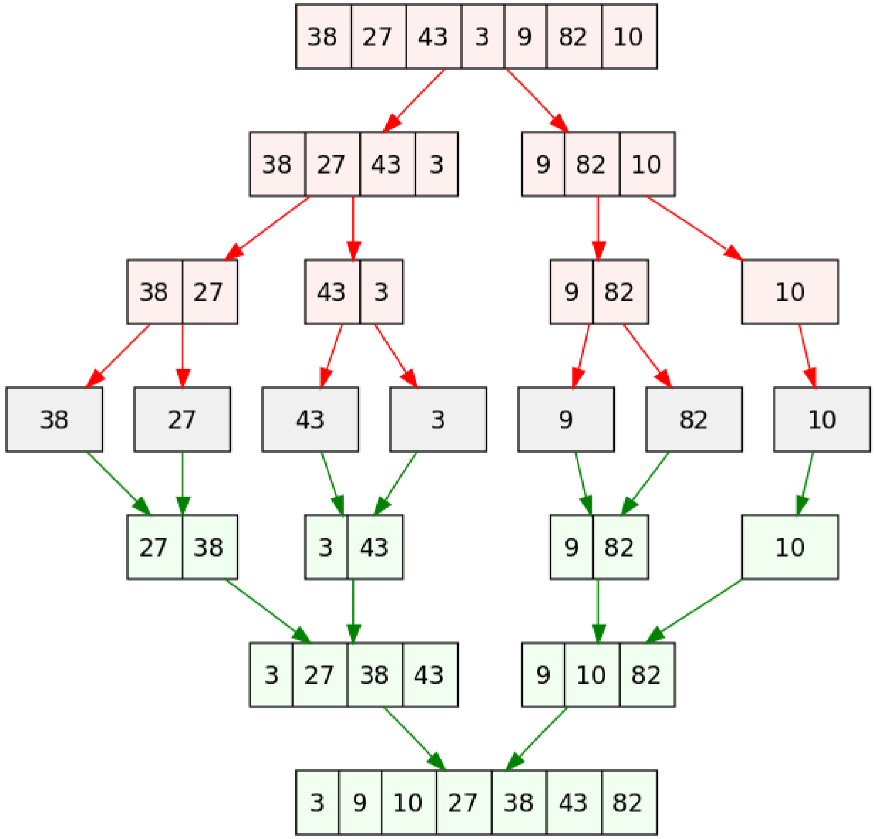
Termination

- ✓ L contains $n1$ elements
- ✓ R contains $n2$ elements
- ✓ $(n1+n2) = (r-p+1)$ = total elements
- ✓ $K = r + 1$
- ✓ $A[p..k-1] = A[p..r]$ contains $(k-p) = (r-p+1)$

❖ Designing Algorithms

- Merge Sort
  - ➢ MERGE-SORT(A, p, r)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```



- ✓ Line 2 : computes an index q that partitions $A[p..r]$ into two subarray
- ✓ Line 3~4 : sort subarrays L, R
- ✓ Line 5 : merge sorted subarrays L, R

❖ Designing Algorithms

- Analysis of Divide–and–conquer algorithm
  - ➢ **Recurrence equation (recurrence)**
    - ✓ algorithm contains a recursive call to itself, describe its running time by recurrence equation
    - ✓ Problem size(n) is small enough ($n \le c$) for some constant c, solution takes constant time $\Theta(1)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

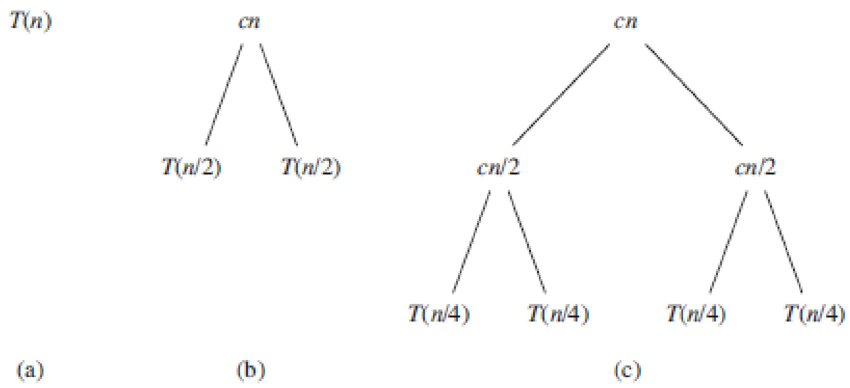
    - ✓ a : the number of subproblems
    - ✓ n/b : the input size of subproblems
    - ✓ $D(n)$: time to divide the problem
    - ✓ $C(n)$: time to combine the solutions

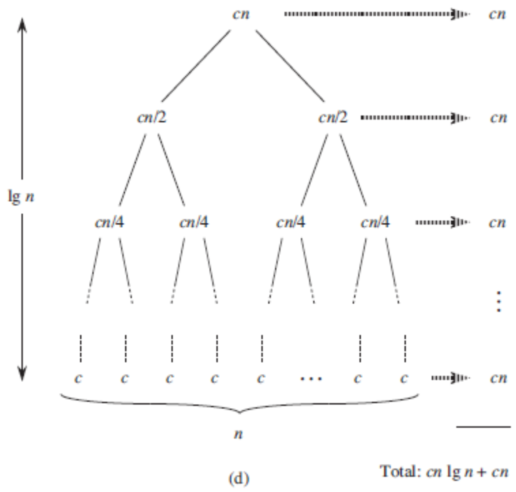
❖ Designing Algorithms

- Analysis of Merge Sort algorithm
  - ➢ Recurrence equation
    - ✓ Divide : just compute the middle of the subarray, $D(n) = \Theta(1)$
    - ✓ Conquer : recursively solve two problem $a, b = 2$, $aT(n/b) = 2T(n/2)$
    - ✓ Combine : already noted that MERGE procedure takes time $\Theta(n)$, $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases} \longrightarrow T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases} \longrightarrow T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

- ✓ $T(n)$ is $\Theta(n\log n)$ using "master theorem"





- ✓ Number of levels = $\log n + 1$
- ✓ Total cost = $cn(\log n + 1)$
  $= cn\log n + cn$
- ✓ $T(n) = \Theta(n\log n)$

Thank You!