Name: Pattarapon Buathong Student ID: 62070504012

M04 Divided and Conquer

1. Objectives / Outcome

1) Students will be able to recognize the concept of divided and conquer techniques.
2) Students will be able to design and implement the divided and conquer techniques for the simple problem with programming language.

2. Related Contents

Divide-and-conquer algorithms work according to the following general plan:

1. **A problem is divided into several subproblems of the same type**, ideally of about equal size.
2. **The subproblems are solved** (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. **The solutions to the subproblems are combined to get a solution to the original problem.**

The divide-and-conquer technique is diagrammed in Figure 1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case. The general divide-and-conquer recurrence is

$$T(n) = aT(n/b) + f(n),$$

where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem:

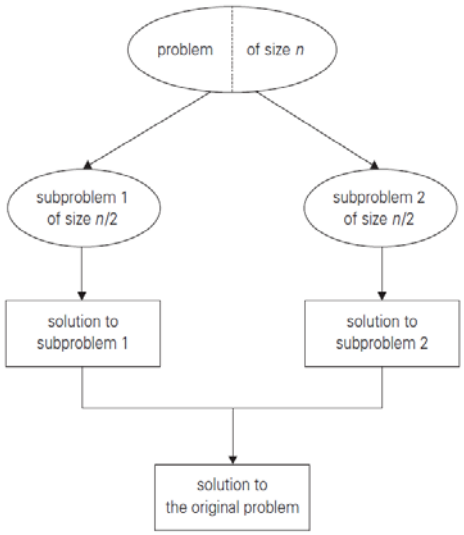


Fig 1. Divide-and-conquer technique

Master Theorem : $$T(n) \in \begin{cases} \Theta(n^d) & if\ a < b^d, \\ \Theta(n^d \log n) & if\ a = b^d, \\ \Theta(n^{\log_b a}) & if\ a > b^d. \end{cases}$$

Name: Pattarapon Buathong Student ID: 62070504012

2.1. Mergesort

Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor ... n-1]$ sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

```
1.  ALGORITHM MergeSort (Arr)
2.  // Input : An array of Arr[left … right]
3.  //Output : An array Arr[left … right] sorted in non-decreasing order (Ascending + Equal)
4.  If Arr has more than one element, then
5.      m <- [(left + right)/2]
6.      MergeSort(Arr[left … right])        # Sort first half of Arr
7.      MergeSort(Arr[m+1 … r])             # Soft second half of Arr
8.      Merge(Arr, left, m, right)          # Merge two halves
```

```
1.  ALGORITHM Merge ( B[0…p-1], C[0…q-1], A[0… p+q-1] )
2.  // Input : Array B[0…p-1] and C[0…q-1] both sorted already
3.  //Output : Sorted array A[0… p+q-1] of the elements of B and C
4.  i <- 0; j <- 0; k <- 0
5.  while i<p and j<q do
6.          if B[i] <= C[j] then
7.              A[k] <- B[i]
8.              i <- i + 1
9.        else
10.             A[k] <- B[i]
11.             j <- j + 1
12. If i = p  then
13.    copy C[j…q-1] to A[k…p+q-1]
14. else
15.   copy B[i…p-1] to A[k…p+q-1]
```

Name: Pattarapon Buathong Student ID: 62070504012

Example: The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 .

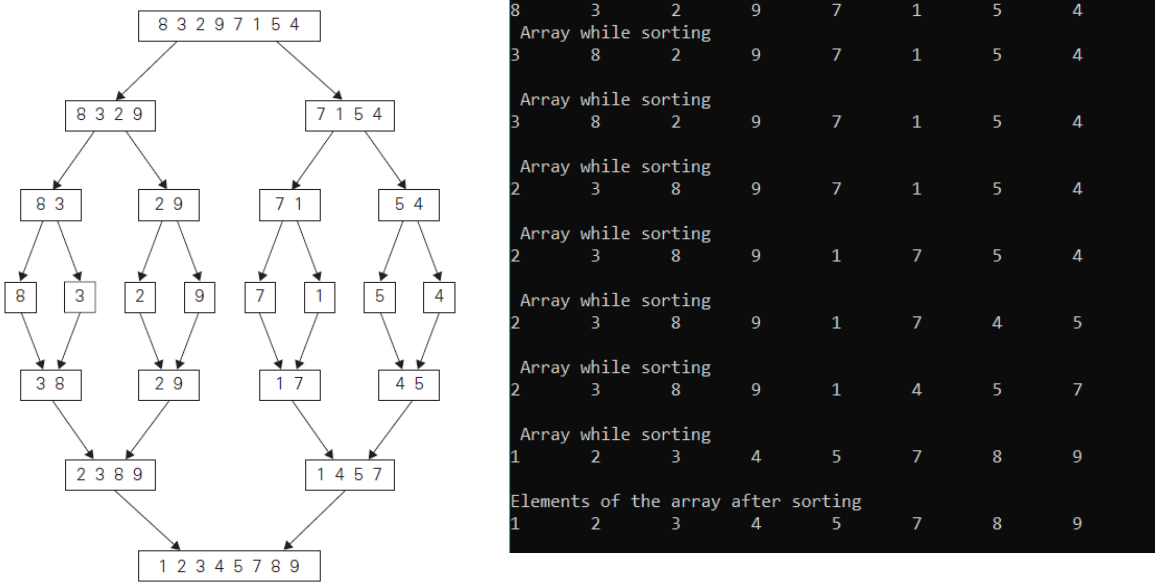


Fig 2 : Example of mergesort operation

The step to merge ,or copy, the array into result is show as follows:-

i=0				j=0				k=0				
2	3	8	9	1	4	5	7	1				
i=0					j=1				k=1			
2	3	8	9	1	4	5	7	1	2			
	i=1				j=1					k=2		
2	3	8	9	1	4	5	7	1	2	3		
		i=2			j=1						k=3	
2	3	8	9	1	4	5	7	1	2	3	4	

How efficient is mergesort?: The basic operation of sorting is the key comparison, in this case including i) Split the list into two half, ii) Sort each half, and iii) Merge both half. Note that the recurrence relation for the number of key comparisons C(n) as follows:-

$$C(n) = \begin{cases} 2 \cdot C(n/2) + C_{merge}(n) \; ; \; n > 1 \\ 0 \qquad\qquad\qquad\qquad\quad ; \; n = 1 \end{cases}$$

$C_{merge}(n)$ is the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two

arrays becomes empty before the other one contains just one. Therefore, for the worst case, $C_{merge}(n) = n - 1$, the complexity of merge sort is

$$C_{worse}(n) \in \boldsymbol{\theta}((n \cdot log_2(n)).$$

2.2. Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

A[0] … A[s-1] *A[s]* *A[s+1] … A[n-1]*

All are <= A [s] All are >= A [s]

After a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently. Here is pseudocode of quicksort: call ***Quicksort(A[0..n – 1])*** where

---

1. ***ALGORITHM*** *QuickSort (Arr)*
2. *// Input : Array Arr[left … right] of orderable elements*
3. *//Output : Array Arr[left … right] in non-decreasing order*
4. *If left < right then*
5. *s <- Partition(Arr[left … right]) # We will use Hoare's partition*
6. *QuickSort(Arr[left … s-1])*
7. *QuickSort(Arr[s+1 … right])*

As the quickselect in chapter 4, the way to divide start by selecting **a pivot**—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. Unlike the Lomuto algorithm, the subarray from both ends, comparing the subarray's elements to the pivot, as follows:

Name: Pattarapon Buathong Student ID: 62070504012

Original list					
		i -->		<-- j	
p	All are <= p	>=p	...	<=p	All are >= p

	<-- j		i -->	
p	All are <= p	<=p	>=p	All are >= p

	<-- j = i -->		
p	All are <= p	=p	All are >= p

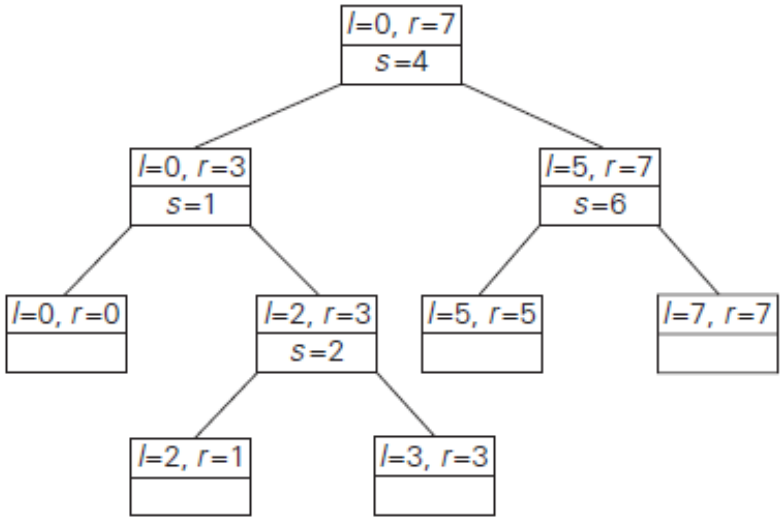
The pseudocode implementing this partitioning procedure is :

```
1.  ALGORITHM HoarePartition (Arr)
2.  // Input : Array Arr[left … right] of orderable elements
3.  //Output : A partition of A[left … right] with the split position as return value
4.  p <- A[left]
5.  i <- left
6.  j <- right + 1
7.  repeat
8.        repeat i <- i +1 until A[i] >= p
9.        repeat j <- j – 1 until A[j] <= p
10.       swap(A[i], A[j])
11.  Until i >= j
12.  swap(A[i], A[j])          # Undo last swap when i>=j
13. Swap(A[low], A[j])
14. Return j
```

Example of quicksort operation. Array's transformations with **pivots** shown in blue highlight. Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained as follows:-

Name: Pattarapon Buathong Student ID: 62070504012

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
				i	*j*		
5	3	1	4	8	2	9	7
				j	*i*		
5	3	1	4	2	8	9	7
	i		*j*				
2	3	1	4	5	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	j	*i*					
2	1	3	4				
1	2	3	4				
			i j				
		3	4				
		j	*i*				
		3	4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					7	8	9



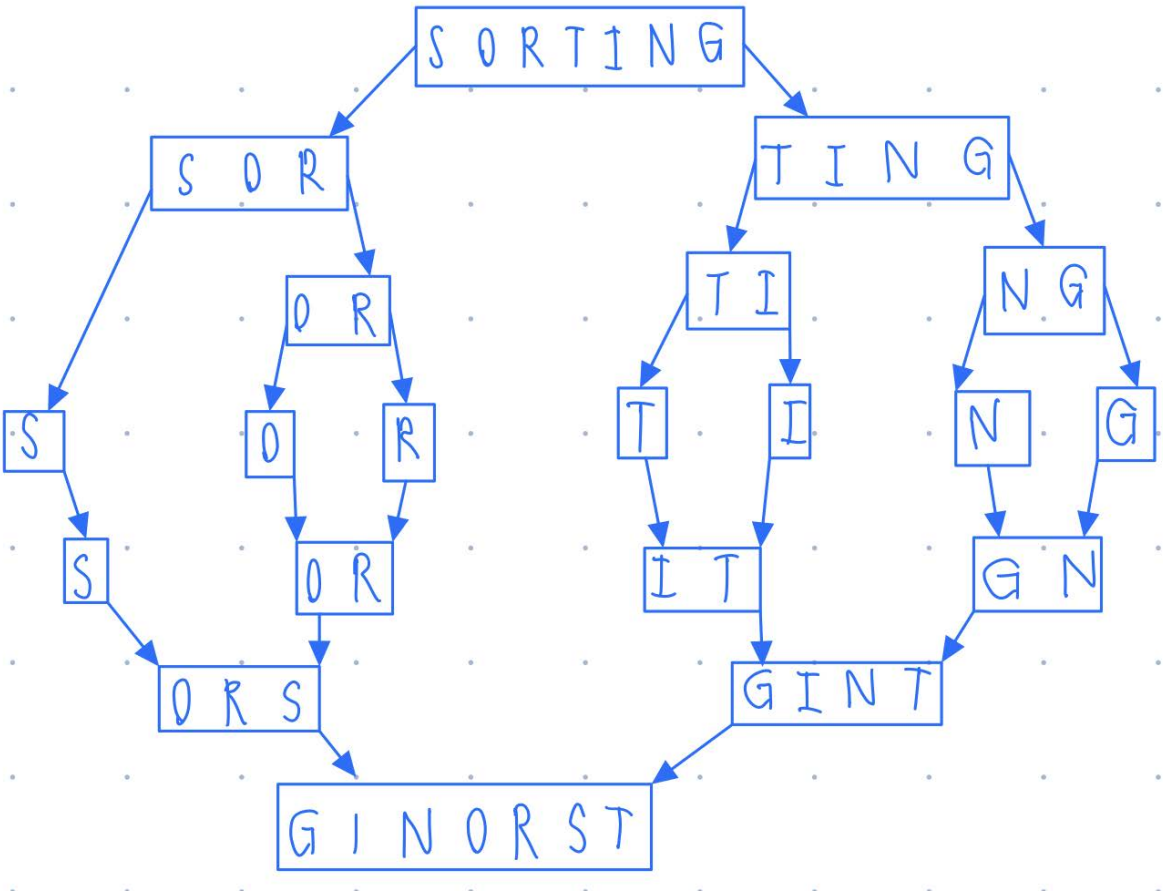
Name: Pattarapon Buathong Student ID: 62070504012

How efficient is Quicksort?:

- Best case $\boldsymbol{\theta}$(n log2n)
  - All the splits happen in the middle of subarrays
- Worst case $\boldsymbol{\theta}$((n2)
  - Input is a strictly increasing array (problem already solved).
  - One of the two subarrays after splitting is always empty.
- Average case $\boldsymbol{\theta}$((n log2n)
  - Split position is equally likely.
  - Actual C(n) = 1.39n log2n

3. In class exercise

3.1. Apply merge sort to the list 'S', 'O', 'R', 'T', 'I', 'N', 'G' in the alphabetical order.



Note: Python can automatically implement the ASCII in each alphabets.

Name: Pattarapon Buathong Student ID: 62070504012

3.1.1. Show the algorithm operation on the list.

ALGORITHM merge_sort(Arr)

```
1.  // Input: unsorted list Arr[a … b]
2.  // Output: alphabetically sorted list
3.  If Arr has more than one element, then:
4.      middle <- (length of Arr) / 2       # Floor division
5.      left <- Arr[a … middle]
6.      right <- Arr[middle … b]
7.      merge_sort(left)
8.      merge_sort(right)
9.      i <- 0; j <- 0; k <- 0
10.     repeat
11.       If left[i] <= right[j] then
12.           Arr[k] = left[i]; i <- i+1
13.       else
14.           Arr[k] = right[j]; j += 1
15.       k += 1
16.     until i >= left's length or j >= right's length
17.
18.     // Copying list
19.     If i equal to left's length, then
20.       repeat
21.           Arr[k] = right[j]; i <- i+1; k <- k+1
22.       until j >= right's length
23.     else
24.       repeat
25.           Arr[k] = left[i]; i <- i+1; k <- k+1
26.       until i >= left's length
```

Name: Pattarapon Buathong Student ID: 62070504012

3.1.2. Count the number of comparisons made and compare it to insertion sort.

['S', 'O', 'R', 'T', 'I', 'N', 'G'] = 7 elements

Comparing merge sort with insertion sort

In merge sort,

$$\boldsymbol{C_{worse}(n) = nlog_2(n) - n + 1 = 7log_2(7) - 7 + 1 = 13.65 \approx 14}$$

Or, $\boldsymbol{O(nlogn)}$

In insertion sort,

$$\boldsymbol{C(n) = \frac{n(n-1)}{2} = \frac{7 \times 6}{2} = 21}$$

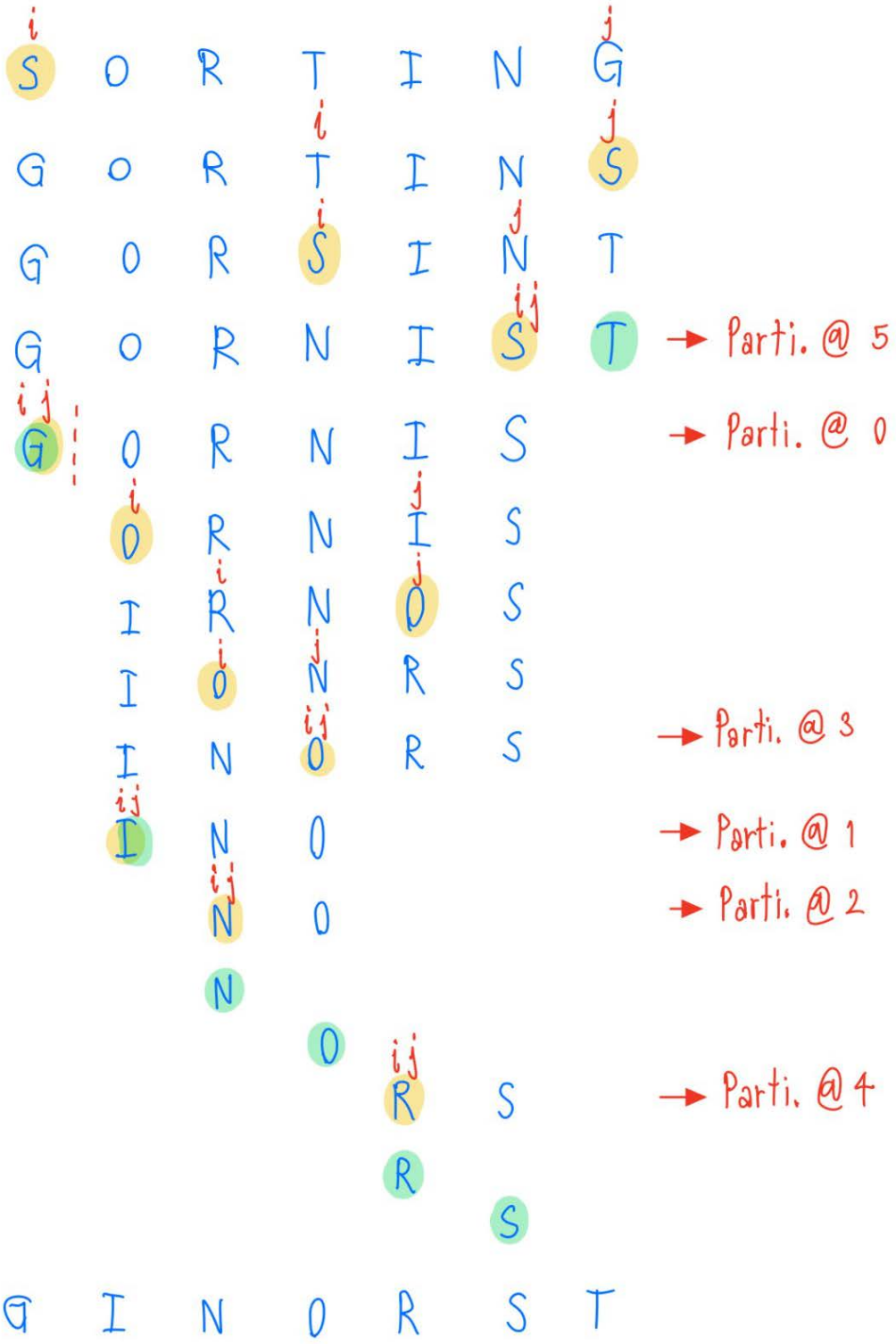
Or, $\boldsymbol{O(n^2)}$

Merge sort has lower number of comparisons than insertion sort, so we can conclude that, merge sort is faster than insertion sort, in the worst cases.

Name: Pattarapon Buathong Student ID: 62070504012

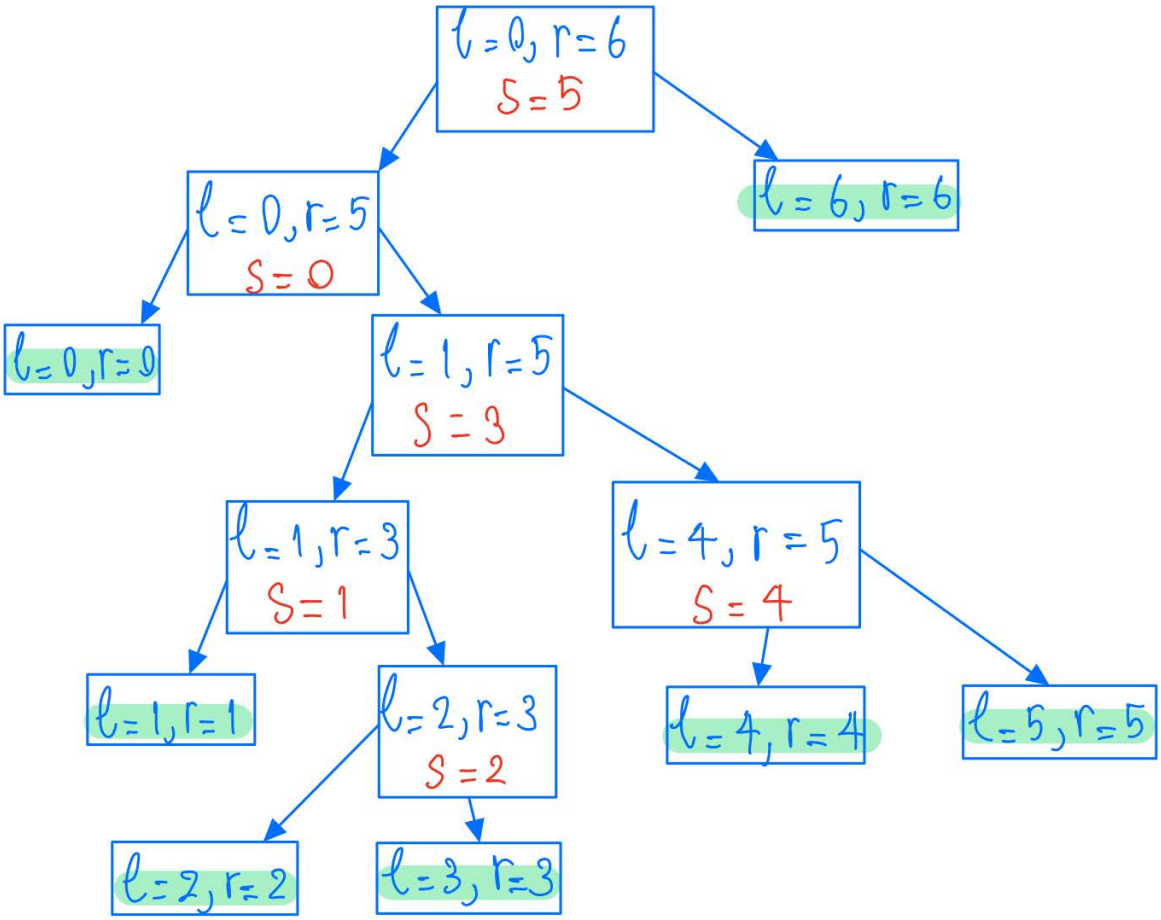
3.2. Apply quicksort to the list 'S', 'O', 'R', 'T', 'I', 'N', 'G' in alphabetical order.

3.2.1. Show the array's transformations.



Name: Pattarapon Buathong Student ID: 62070504012

3.2.2. Draw the tree of recursive calls made.



Name: Pattarapon Buathong Student ID: 62070504012

4. LAB Assignments (Ass 02)

4.1. Ass 02 Merge sort: Complete the functions MergeSort() and Merge() discussed in the class. The program accepts the list size n, generates n random numbers between 0 to 999 for the list, and applies the Merge sort. **Submit the screen outputs for n = 7 and n = 20 and the Python code to LEB2.**

4.1.1. Design Algorithms

ALGORITHM

```
1.  //Input: number (length) of desired random list
2.  //Output: Non-decreasing sorted list
3.  n <- Input size of list as an integer
4.  Arr <- [0]*n    # initiate the size of the list
5.  For i=0 to n, do:
6.      Arr[i] <- random integer (range 1 to 1000)      # use 'random' package
7.  ALGORITHM MergeSort(Arr):
8.  If Arr has more than one element, then:
9.      middle <- (length of Arr) / 2       # Floor division
10.   left <- Arr[a … middle]
11.   right <- Arr[middle … b]
12.   merge_sort(left)
13.   merge_sort(right)
14.   i <- 0; j <- 0; k <- 0
15.   repeat
16.     If left[i] <= right[j] then
17.         Arr[k] = left[i]; i <- i+1
18.     else
19.         Arr[k] = right[j]; j += 1
20.     k += 1
21.   until i >= left's length or j >= right's length
22.   // Copying list
23.   If i equal to left's length, then
24.     repeat
25.         Arr[k] = right[j]; i <- i+1; k <- k+1
26.     until j >= right's length
27.   else
28.     repeat
29.         Arr[k] = left[i]; i <- i+1; k <- k+1
30.     until i >= left's length
```

Name: Pattarapon Buathong Student ID: 62070504012

The sample run:

```
Enter the number of elements: 5
Elements of the array before sorting
[58, 294, 129, 479, 902]
Elements of the array after sorting
[58, 294]
[479, 902]
[129, 479, 902]
[58, 129, 294, 479, 902]
Enter the number of elements: 10
Elements of the array before sorting
[563, 991, 264, 862, 53, 59, 579, 544, 70, 195]
Elements of the array after sorting
[563, 991]
[53, 862]
[53, 264, 862]
[53, 264, 563, 862, 991]
[59, 579]
[70, 195]
[70, 195, 544]
[59, 70, 195, 544, 579]
[53, 59, 70, 195, 264, 544, 563, 579, 862, 991]
Enter the number of elements:
```

Name: Pattarapon Buathong Student ID: 62070504012

4.2. Ass 02 Quick sort : Complete the function QuickSort() discussed in the class. The program accepts the list size n, generates n random numbers between 0 to 999 for the list, and applies the Quick sort. **Submit the screen shot outputs for n = 7 and n = 20 and the Python code to LEB2.**

4.2.1. Design Algorithms

ALGORITHM

```
1.  //Input: number (length) of desired random list
2.  //Output: Non-decreasing sorted list
3.  n <- Input size of list as an integer
4.  Arr <- [0]*n    # initiate the size of the list
5.  For i=0 to n, do:
6.      Arr[i] <- random integer (range 1 to 1000)      # use 'random' package
7.  ALGORITHM HoarePartition(Arr, low, high):
8.      p <- A[left]
9.      i <- left
10.     j <- right + 1
11.     repeat
12.         repeat i <- i +1 until A[i] >= p
13.         repeat j <- j – 1 until A[j] <= p
14.         swap(A[i], A[j])
15.     Until i >= j
16.     swap(A[i], A[j])      # Undo last swap when i>=j
17.     Swap(A[low], A[j])
18.     Return j
19. ALGORITHM QuickSort(Arr, low, high):
20.     If Arr has more than one element, then:
21.         Return Arr
22.     Else If low < high, then:
23.         parti <- HoarePartition(Arr, low, high)
24.         Quicksort(Arr, 0, parti)          # Recursive
25.         Quicksort(Arr, parti+1, high)  # Recursive
26.
```

Name: Pattarapon Buathong Student ID: 62070504012

The sample run:

```
Enter the number of elements: 5
Elements of the array before sorting
[472, 716, 628, 539, 16]
Elements of the array after sorting
[16, 472, 539, 628, 716]
Enter the number of elements: 10
Elements of the array before sorting
[751, 356, 365, 314, 478, 510, 412, 710, 476, 987]
Elements of the array after sorting
[314, 356, 365, 412, 476, 478, 510, 710, 751, 987]
Enter the number of elements:
```

5. Additional Questions (OPTIONAL)

5.1. How does this algorithm (merge sort and/or quicksort) compare with the brute-force algorithm for this problem?

Ans: Merge sort and quick sort is the "**Divide and Conquer**" algorithm, which divides the array into the smaller instances, solve the smaller instances recursively, then solutions of the smaller instances will be combined to get the solution of the original problem.

Brute-force algorithm, for example, Bubble sort, Selection sort. It is one of the simplest algorithms, but the time complexity is quite high. It compares each pair of elements and then swap it, then move to the next element, it will do like this until the last element of the array.

5.2. Find the order of growth for solutions of the following recurrences.

5.2.1.$T(n) = 4T(n/2) + n, T(1) = 1$

5.2.2.$T(n) = 4T(n/2) + n2, T(1) = 1$

5.2.3.$T(n) = 4T(n/2) + n3, T(1) = 1$

Name: Pattarapon Buathong Student ID: 62070504012

Form: $T(n) = aT(\frac{n}{b}) + f(n)$

1) $T(n) = 4T(n/2) + n$, $T(1) = 1$
Master's theorem: compare $n^{\log_b a}$ with $f(n)$
$a = 4, b = 2$; $n^{\log_2 4} = n^2 > f(n) = n$
$\therefore T(n) \in \theta(n^2)$ //

2) $T(n) = 4T(n/2) + n^2$, $T(1) = 1$
Master's theorem: compare $n^{\log_b a}$ with $f(n)$
$a = 4, b = 2$; $n^{\log_2 4} = n^2 = f(n) = n^2$
$\therefore T(n) \in \theta(n^2 \log n)$ //

3) $T(n) = 4T(n/2) + n^3$, $T(1) = 1$
Master's theorem: compare $n^{\log_b a}$ with $f(n)$
$a = 4, b = 2$; $n^{\log_2 4} = n^2 < f(n) = n^3$
$\therefore T(n) \in \theta(n^3)$ //

5.3. For the version of quicksort given in this section:

5.3.1. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

Ans: It is the worst-case input. Because, at first compare of the quicksort, it will take 5 comparisons, and pivot "5" will be at the same position. Now we have 5 elements remaining, if we continue comparing, it will take 4, 3, 2, and 1 comparison in the following epoch, which is the worst case of the quicksort: $O(n^2)$.

5.3.2. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

Ans: Strictly decreasing array is the worst-case input. For example, let array contains [6,5,4,3,2,1]. If we select 1 as the pivot, it will take 5 comparisons in the first round, and then 4, 3, 2, and 1 in the following round. Which is also $O(n^2)$ as the first question.

6. Reference

Levitin, A. (2012) Introduction to the Design & Analysis of Algorithms (3rd edition). Pearson. (Chapter 5.1 and 5.2)