Name - Gurvinder Kaur Matharu
PRN - 20190802077.

DAA Assignment - 1.

Q1. Sort all the functions below in increasing order of asymptotic (Big-O) growth. If some have the same asymptotic growth, then be sure to indicate that. As usual, lg means base 2.

1. $5n$ (4)
2. $4\log n$ (2)
3. $4\log\log n$ (1)
4. n^4 (15)
5. $n^{1/2}\log^4 n$ (3)
6. $\lg^{5\lg n}$ (6)
7. $\lg^{\lg n}$ (7)
8. 5^n (9)
9. 4^{n^4} (13)
10. 4^{4^n} (14)
11. 5^{5^n} (15)
12. 5^{5n} (10)
13. $n^{n^{(1/5)}}$ (8)
14. $n^{n/4}$ (12)
15. $(n/4)^{n/4}$ (11)

$\Rightarrow 4\lg\lg n < 4\lg n < n^{1/2}\lg^4(n) < 5n < n^4 < (\lg n)^{5\lg n} < n^{\lg n} < n^{n^{1/5}} < 5^n < 5^{5n} < (n/4)^{n/4} < n^{n/4} < 4^{n^4} < 4^{4^n} < 5^{5^n}$

Question 2.

1. $T(n) = 2T(\frac{n}{4}) + 1$

$a = 2, \quad b = 4, \quad f(n) = 1$

$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

$n^{\log_b a} > f(n)$ case 1

$T(n) = \theta(n^{\log_4 2})$.

2. $T(n) = 2T(n/4) + n^{1/2}$

$a = 2, \quad b = 4., \quad f(n) = n^{1/2}$

$n^{\log_b a} = n^{\log_4 2} = n^{1/2}$

$f(n) = n^{\log_b a}$ case 2

$T(n) = \theta(n^{\log_4 2} \log n)$

3. $T(n) = 7T(n/3) + n^2$

$a = 7, \quad b = 3, \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_3 7} = n^{1.77}.$

$f(n) > n^{\log_b a}$ case 3

$T(n) = \theta(n^2)$

4. $T(n) = 7T(\frac{n}{2}) + n^2$

$a = 7, \quad b = 2, \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_2 7} = n^{2.807}$

$f(n) < n^{\log_b a}$ case 1

$T(n) = \theta(n^{\log_2 7})$

(5.) $T(n) = T(n-3) + n^2$

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

$\because a \geq 1, b > 1$ and $f(n)$ should be positive. Masters theorem cannot be applied here ($\because b = 0$).

(6.) $T(n) = 2T(n/2) + \frac{n}{\log n}$.

$a = 2, \quad b = 2, \quad f(n) = \frac{n}{\log n}$.

$n^{\log_2 2} = n$

$f(n) < n^{\log_b a}$ case 1

$T(n) = \theta(n^{\log_2 2}) = \theta(n)$

(7.) $T(n) = T(n-2) + \frac{n}{\log n}$.

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

$\because a \geq 1, b > 1$ and $f(n)$ should be positive, Masters theorem cannot be applied in this case. ($\because b = 0$).

Q3. Using substitution method show the recurrence eqn. for $T(n) = 2T(n/2) + \theta(n)$. Compute the upper bound & lower bound seperately with steps involved.

$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$

$T(n) < 2T\left(\frac{n}{2}\right) + cn$ (for some +ve const c)

For upper bound:

guess. $T(n) \leq dn\log n$ (for some +ve const. d)

$T(n) \leq 2T(\frac{n}{2}) + cn$

sub^n $= 2d(\frac{n}{2})\log(\frac{n}{2}) + cn$

$= dn\log n/2 + cn$

$= dn\log n - dn + cn$ (desired − residual)

$\therefore \leq dn\log n$

if $-dn + cn \leq 0$

$\boxed{d \geq c}$

$\therefore T(n) = O(n\log n)$

For lower bound:

$T(n) \geq 2T(n/2) + cn$

guess: $T(n) \geq dn\log n$

sub^n: $T(n) \geq 2T(\frac{n}{2}) + cn$

$= 2d(\frac{n}{2})\log(\frac{n}{2}) + cn$

$= dn\log n/2 + cn$

$= dn\log n - dn + cn$ (desired − residual)

$T(n) \geq dn\log n$

$-dn + cn \geq 0$

$\boxed{d \leq c}$

$\therefore T(n) = \Omega(n\log n)$

Q4. Write a short note on Merge sort with an example (show the working of merge sort). Write the pseudo-code for recursive & iterative method. Explain briefly the time complexity of Merge sort.

- Merge Sort algorithm follows the divide & conquer approach.

Divide - divide the problem into subproblems.

Conquer - conquer the sub-problems by solving them recursively.

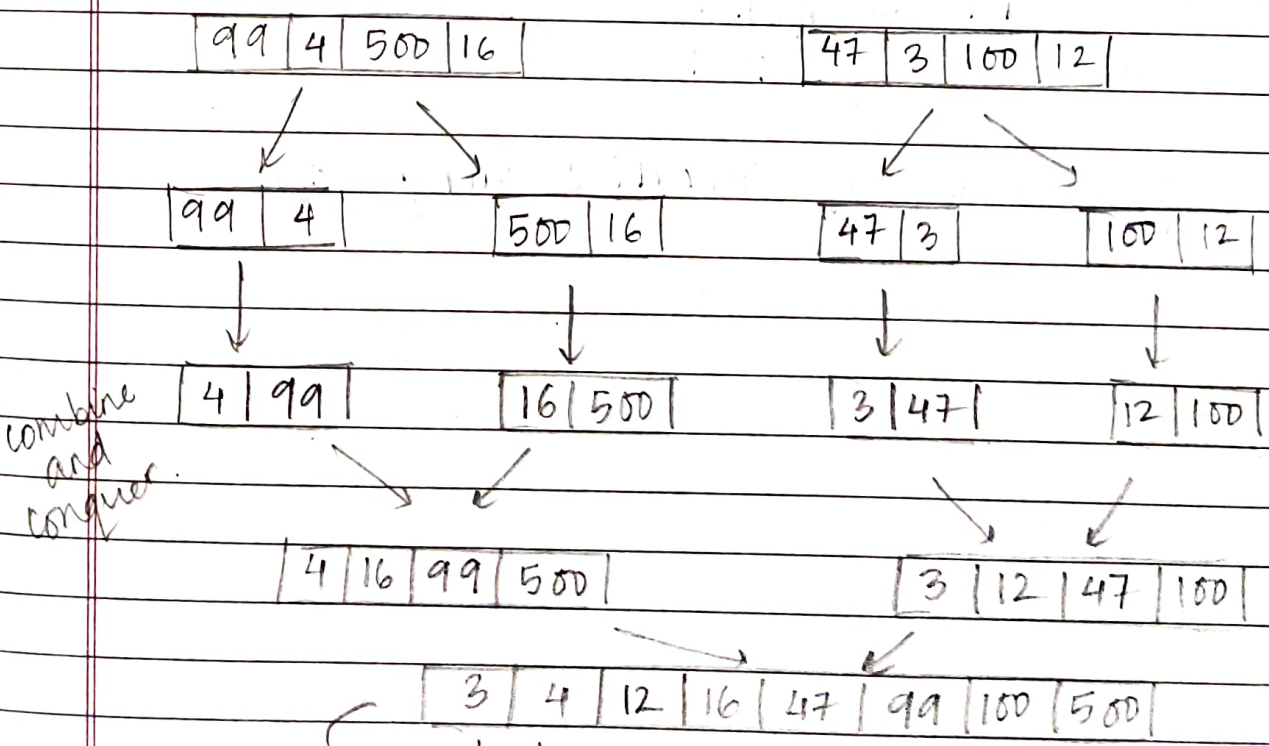
Combine the solutions to subproblems to get the final result.

Eg: Consider an array:

0	1	2	3	4	5	6	7
99	4	500	16	47	3	100	12

- The array is divided into 2 equal sub arrays.

(divide)



pseudocode for Recursive Merge Sort

→ MERGE_SORT(A)
if A.length > 1
    mid = (A.length)/2
    Divide the array into 2 parts
    L = A[: mid] // first part
    R = A[mid:] // 2nd part.
    MERGESORT(L)
    MERGESORT(R)
    i = 0
    j = 0
    K = 0
    while i < L.length & j < R.length
        if L[i] < R[j]
            A[K] ← L[i]
            i++ increment
        else
            A[K] ← R[j]
            j++ increment
        K++ increment
    while i < L.length
        A[K] ← L[i]
        i++ increment
        K++ increment
    while j < R.length
        A[K] ← R[j]
        j++ increment
        K++ increment

pseudocode for Iterative Merge sort.

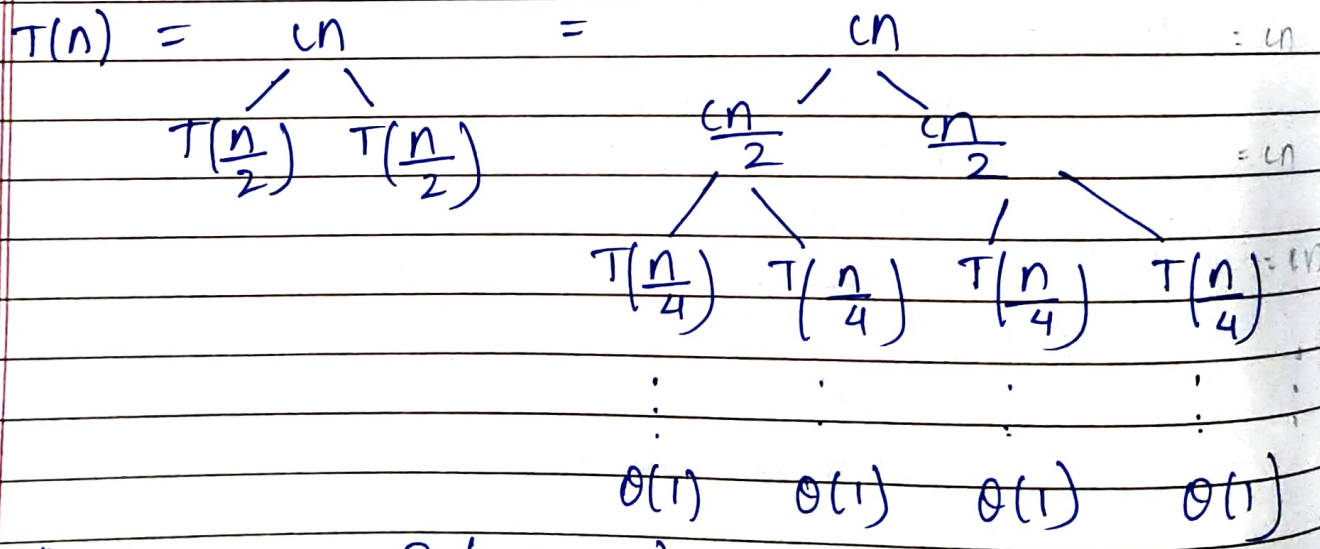
→ MERGE (A, L, m, r)
n1 = m - L + 1
n2 = r - m
Let L & R be the 2 new sub arrays.
for i = 0 to n1
    L[i] ← A[L + i]
for i = 0 to n2
    R[i] ← A[m + i + 1]
i = 0
j = 0
k = 1
while i < n1 & j < n2
    if L[i] > R[j]
        A[k] ← R[j]
        j++ increment
    else
        A[k] ← L[i]
        i++ increment
    k++ increment
while i < n1
    A[k] ← L[i]
    i++ increment
    k++ increment
while j < n2
    A[k] ← R[j]
    j++ increment
    k++ increment

```
MERGE SORT (A)
    size = 1
    while size < len(A) - 1
        left = 0
        while left < len(A) - 1
            mid = min((left + size - 1), (len(A)-1))
            right = ((2 * size + left - 1, len(A)-1)
                [2 * size + left - 1 > len(A) - 1])
            MERGE (A, left, mid, right)
            left = left + size * 2
        size = 2 * size.
```

⟹ Analysis of Merge sort.

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$$

$$T(n) = 2T(n/2) + cn$$



Best Case - $\Omega(n \log n)$
Average Case - $\theta(n \log n)$

Height of the tree = log N
No. of leaves = N (length of array)
Time complexity of Merge sort, T(n) = O(n log n)

Q5. Write pseudocode & explain with eg. different partition algorithm of quick sort. Explain briefly the time complexity of quick sort.

pseudocode –

① →
```
PARTITION (A, p, r)
    x = A[p]            // pivot
    i = p-1 (index)
    for j = p+1 to q
        if A[j] ≤ x
            then i = i+1
            exchange A[i] with A[j]
    exchange A[p] with A[i]
    return i  // to get the pivot element
                 position.
```

② → Hoare partition Algorithm.

```
HOARE PARTITION (A, p, r)
 x = A[p]
 i = p-1
 j = r+1
 while TRUE:
     repeat
         j = j-1
     until A[j] ≤ x
     repeat
         i = i+1
     until A[i] ≥ x
```

```
if i<j
    exchange A[i] with A[j]
else
    return j
```

(3) → 3 way Partition.

Let v be the pivot value

A[0] = pivot.

scan from left to right 'i'

if A[i] < v

exchange A[lt] with A[i]

& increment lt and i

lt → left side)

if A[i] > v

exchange A[gt] with A[i]

& decrement gt

gt → right side)

if A[i] = v

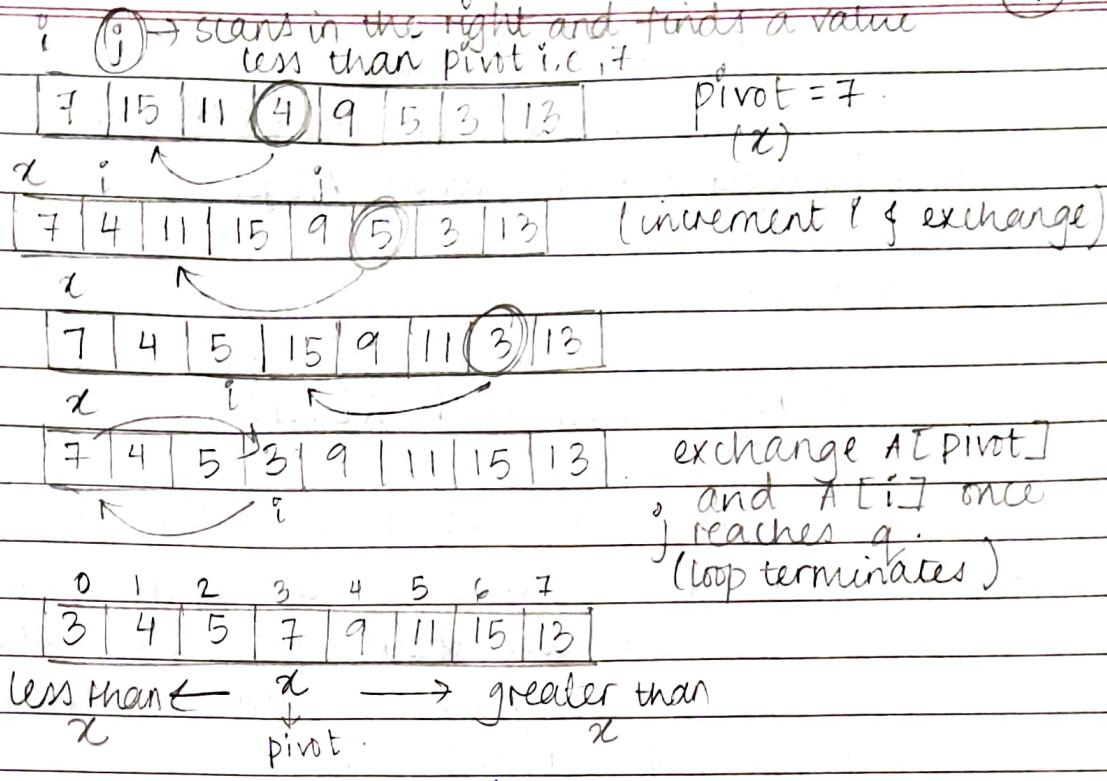
then increment i

```
QuickSort(A, p, r)
  if p<r
      q = PARTITION(A, p, r)
      QuickSort(A, p, q-1)
      QuickSort(A, q+1, r).
```

→ Consider an array:

7	15	11	4	9	5	3	13

(1) first partition method —

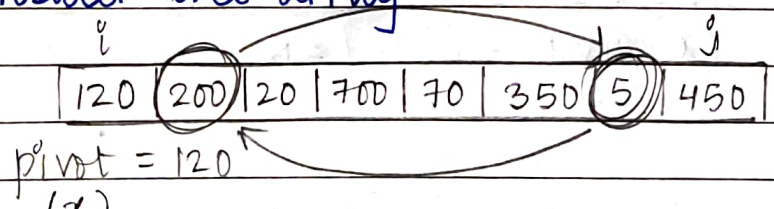


The array gets arranged in such a way that the elements on the left side of the pivot are less than the pivot value and elements on the right side are greater than the pivot value. Thus, the array keeps on dividing in a similar way and then the array gets sorted.

3	4	5	7	9	11	13	15

(2). Hoare Partition –

Consider the array –



Consider the first element of the array as pivot

pointer i → searches for elements greater than the pivot value

pointer j → searches for elements less than pivot value

pivot

120	5	20	700	70	350	200	450

- exchanging A[i] with A[j].

pivot ←

120	5	20	70	700	350	200	450

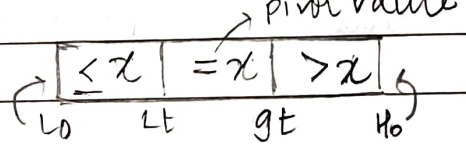
exchange A[0] with A[i].

70	5	20	120	700	350	200	450

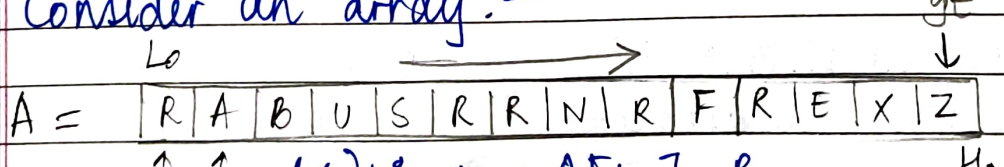
The array is arranged in such a way that the elements less than pivot (x) are on the left while the elements greater than pivot (x) are on the right. This keeps on repeating until the array gets sorted.

5	20	70	120	200	350	450	700

(3) 3-way partition.



Consider an array :—

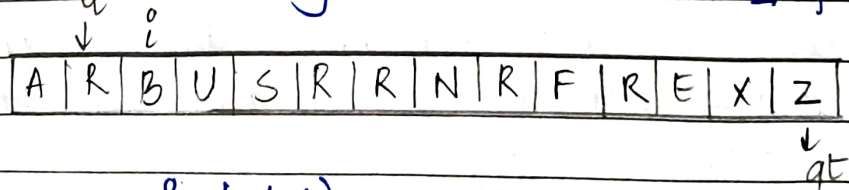


(x) pivot = A[Lo] = R

A[i] = A

x > A[i]

exchange A[i] ↔ A[lt] & increment i, lt

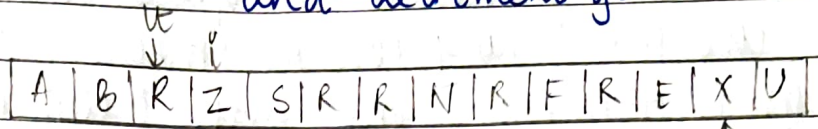


pivot (x) > A[i]

exchange A[i] ↔ A[lt] & increment i, lt

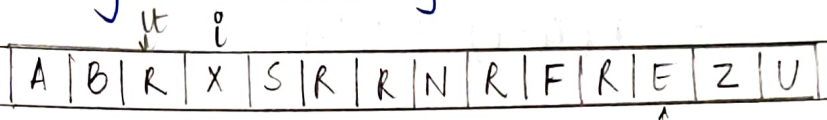
$x < A[i]$

exchange $A[i] \leftrightarrow A[gt]$ and decrement gt.



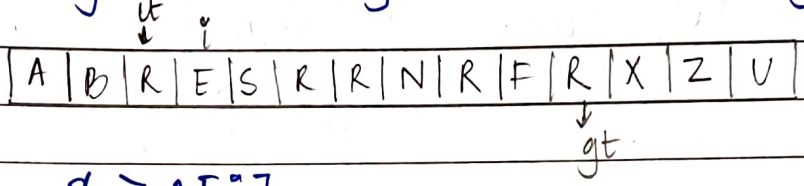
$x < A[i]$

exchange $A[i] \leftrightarrow A[gt]$ and decrement gt.



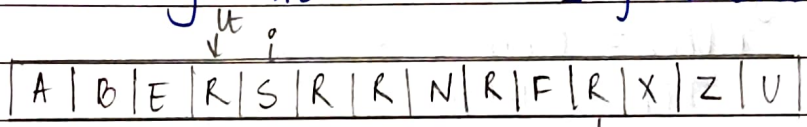
$x < A[i]$

exchange $A[i] \leftrightarrow A[gt]$ and decrement gt.



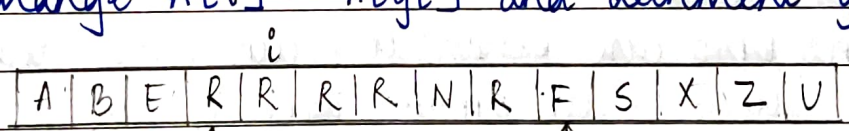
$x > A[i]$

exchange $A[i] \leftrightarrow A[lt]$ & increment i, lt.



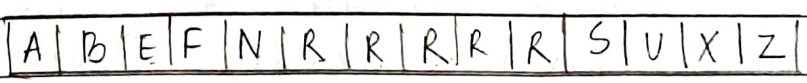
$x < A[i]$

exchange $A[i] \leftrightarrow A[gt]$ and decrement gt.



$x = A[i]$, increment i.

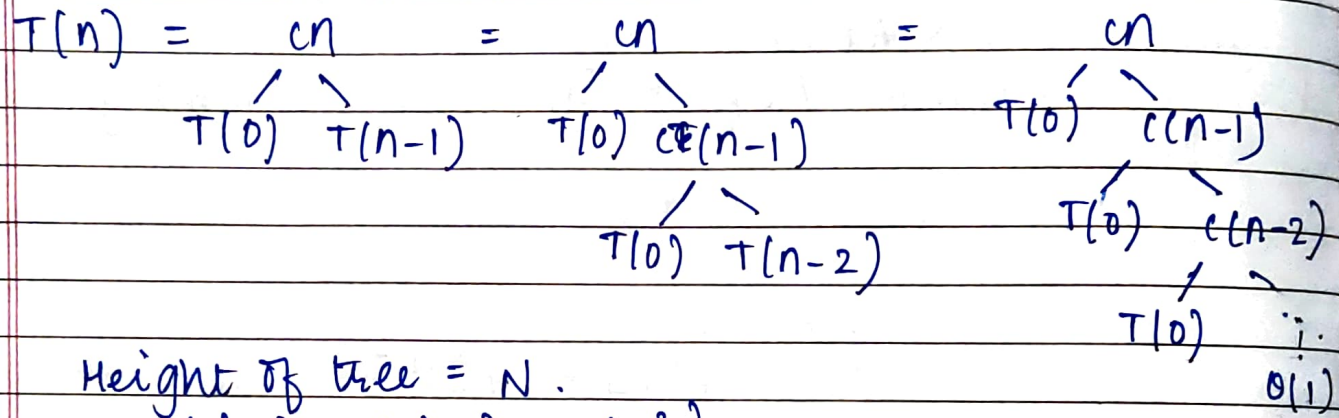
- The above process continues and at some point of time the array is arranged in such a way that the elements less than pivot are on the left and the elements greater than the pivot are on the right side, and then in a similar way the array is sorted.



Analysis of Quick Sort.
(assuming the elements to be distinct).

Worst case –
The worst case behaviour for quick sort occurs when the partitioning routine produces one sub problem with (n-1) elements & one with 0 elements.

$$T(n) = T(n-1) + T(0) + \theta(n)$$
$$= T(n-1) + \theta(n).$$



Height of tree = N.
$T(n) = \theta(n) + \theta(n^2)$
$T(n) = \theta(n^2)$.

Best case –
The best case behaviour occurs when the partition produces 2 sub-arrays each of size no more than $n/2$. In this, the quick sort runs much faster.
The recurrence for the running time is –

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n).$$

$$T(n) = n\log n.$$

Average Case –
$T(n) = \theta(n \log n)$ for average case running time. If the pivot is equally likely to end up

anywhere in subarray after partitioning, there's a 50% chance of getting a worst or a best case.