Deakin University

Discrete Mathematics

OnTrack Submission

Create your Module

Submitted By:
Niranjan Shrestha
nshrestha
2020/10/03 18:19

Tutor:
Phillip Wyld

Outcome	**Weight**
Mathematical Skills	♦♦♦♦◊
Problem Solving	♦♦♦♦♦
Task Management	♦♦♦♦♦
Learning Skills	♦♦♦♦♦

Analysis of algorithm module

October 3, 2020

Introduction

Module learning objectives

For completing this module, you should be able to:

1. The necessity for analysis of algorithms and the difference between big-Oh, big Omega, big Theta.
2. Compute the Best, Worst and Average cases.

Learning resources

There is no textbook reading for this module. Below is the link provided for reading for Analysis of the algorithm.

- https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/
- https://yourbasic.org/algorithms/time-complexity-explained/
- https://www.programiz.com/dsa/asymptotic-notations/
- https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm/
- https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/

You may like to keep these link tab open as you work through the module, to provide further detail on the key concept you will cover.

Activity 1: The necessity for analysis of algorithms and the difference between big-Oh, big Omega, big Theta.

The exercises and resources provided on this page will help you to achieve the following learning objective:

Compare Time and Space Complexity using different notation.

The necessity for analysis of algorithms

Analysis of the algorithm is the study of run time and space requirements of the algorithms. The analysis will provide the runtime known as time complexity and the space required known as space complexity.

For the same input, the design of the algorithm can affect the time it takes to generate the output and the space required in the memory.

The analysis provides the theoretical estimation of the time and space the algorithm will take to execute.

In real life, if the programming is not fast enough, then it is not useful or not even functional and also if it uses too much memory, it will be obsolete for most of the user.

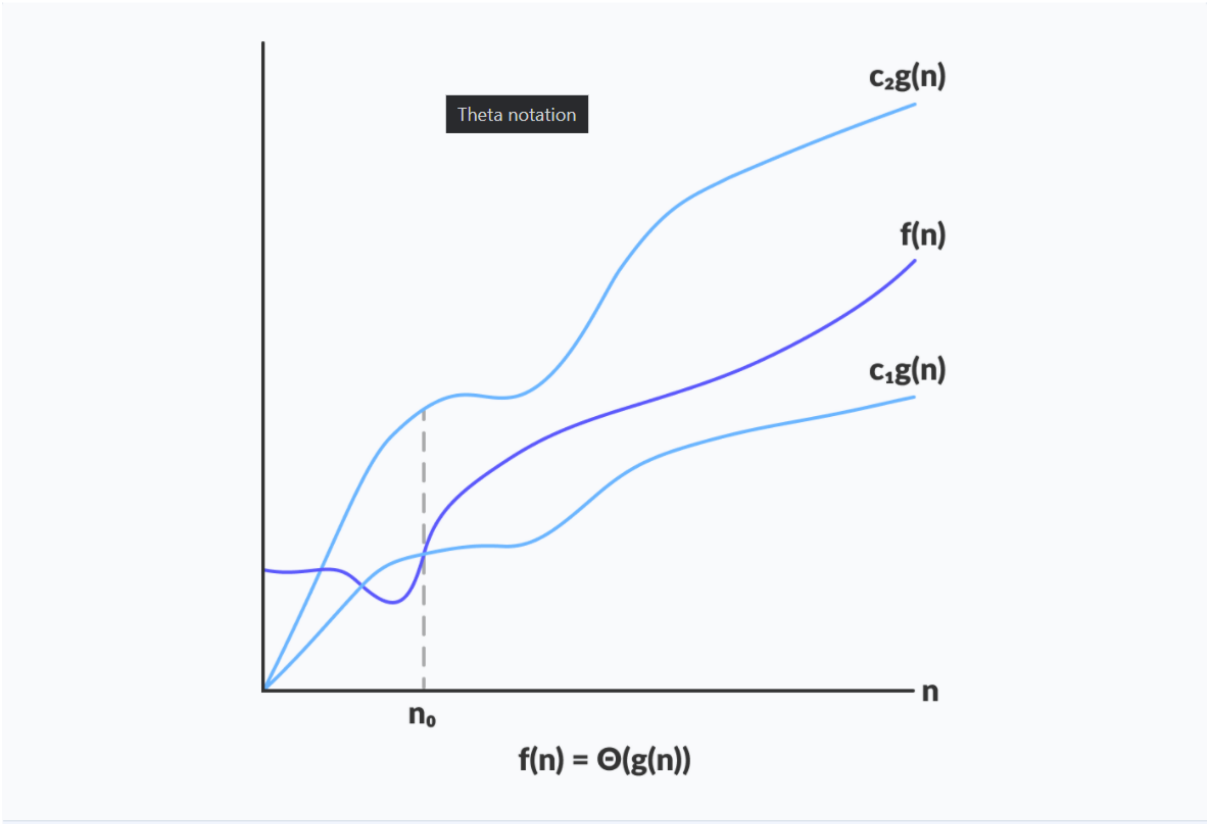
- Here is the link https://yourbasic.org/algorithms/time-complexity-explained/ available for this Activity.

You may like to have these open as you work through the material below. Specific references will be provided to help guide your reading.

[1]

Theta Notation (Θ-notation)

It represents the upper and lower bound of the algorithm and is used to analyse the average time complexity.



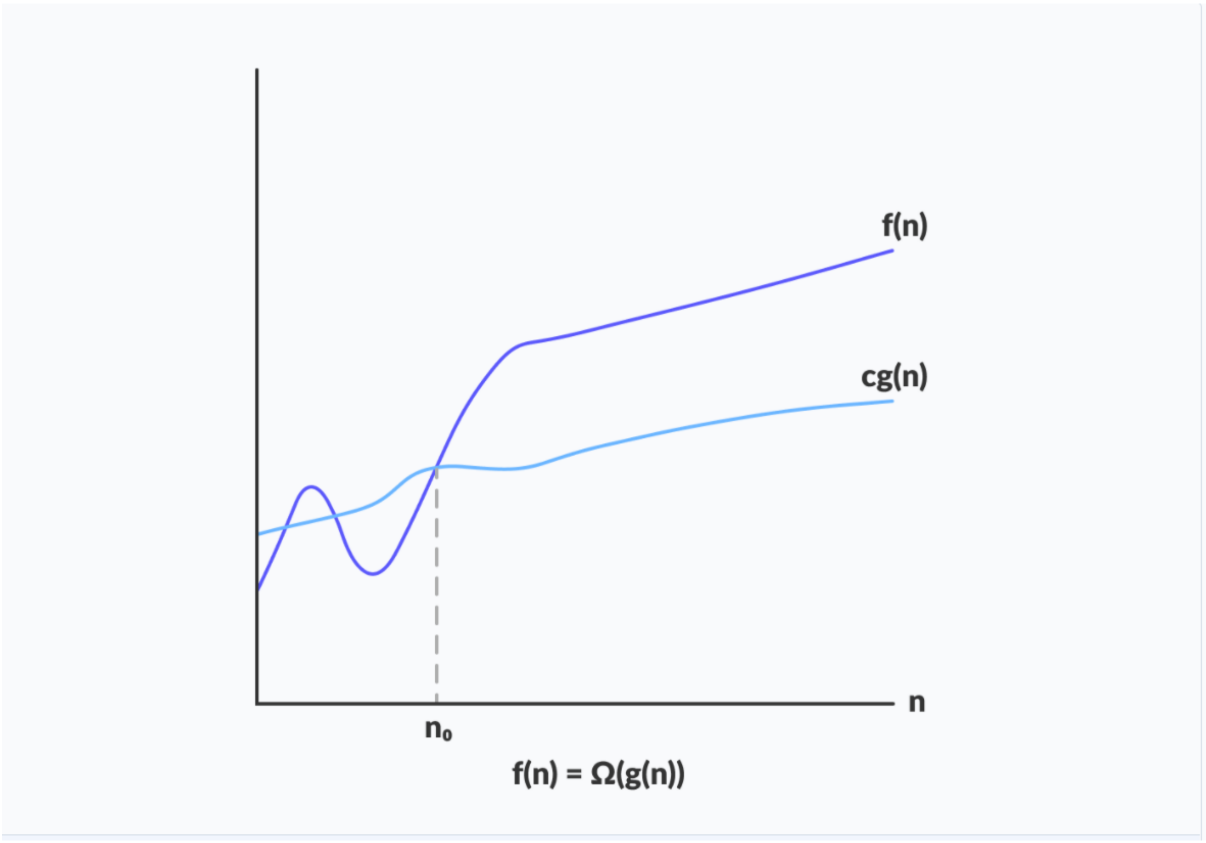
Theta bounds the function within constants factors

A function f(n) is said to be of order g(n), denoted by Θ (g(n)) if: c1 and c2 are positive constants such that we can get the average form c1g(n) and c2g(n), for large n.

$\Theta(g(n)) = \{f(n)$: there exist positive constants c1, c2 and n0 such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n0\}$

Omega Notation(Ω-notation)

The lower bound of an algorithm's running time is given by Ω-notation. It gives the algorithm's best-case complexity.



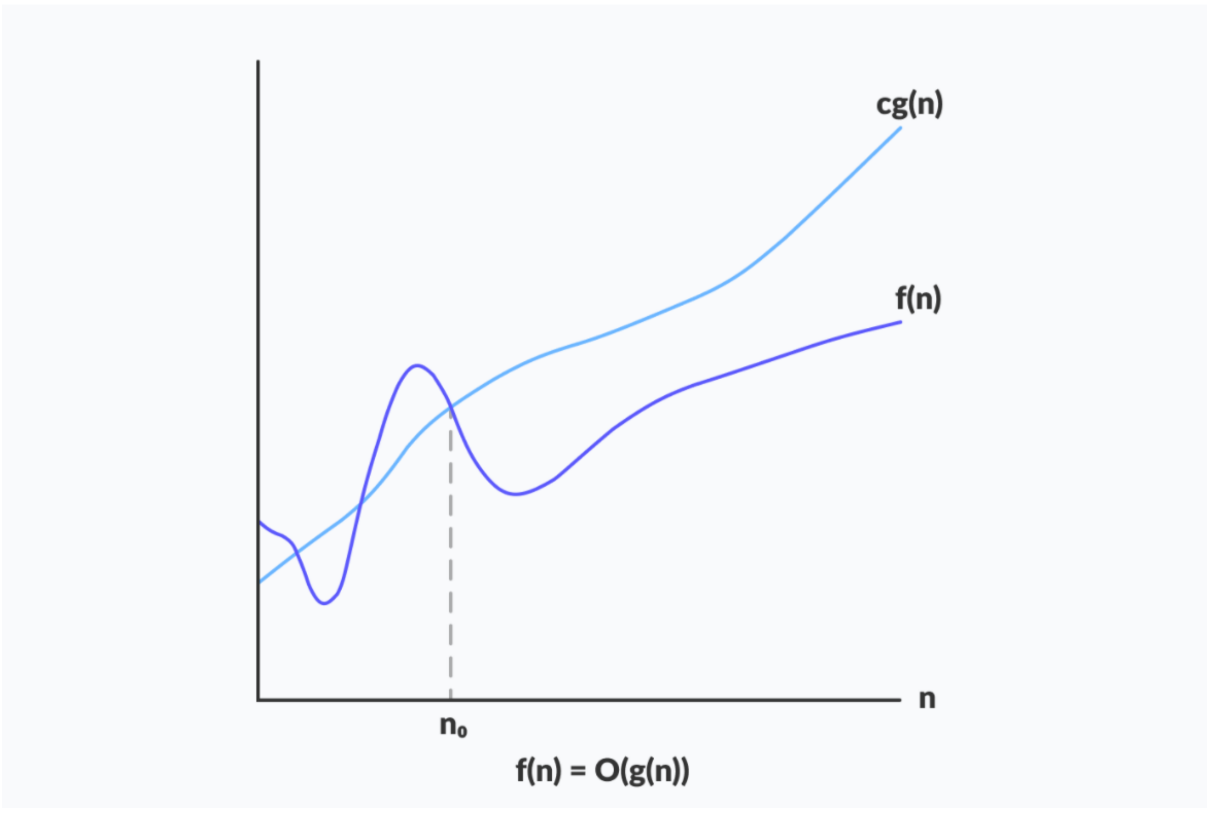
A function f(n) is said to be of order g(n), denoted by Ω (g(n))
If: c is a positive constant for large n, such that it is above cg(n).

Ω(g(n)) = {f(n): there exist positive constants c and n0
 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0}

The minimum time needed by the algorithm for the size of input n is given by Ω (g(n)).

Big O notation(O-notation)

Represent the upper limit of an algorithm’s runtime and it gives an algorithm’s worst-case complexity



A function f(n) is said to be of order g(n), denoted by O(g(n))
if: c is positive constants such that it lies between 0 and cg(n), for large n.

O(g(n)) = {f(n): there exist positive constants c and n0
such that 0 ≤ f(n) ≤ c2g(n) for all n ≥ n0}

It represents the worst-case running time of an algorithm since we are often interested in the worst-case scenario, it is commonly used to evaluate an algorithm.

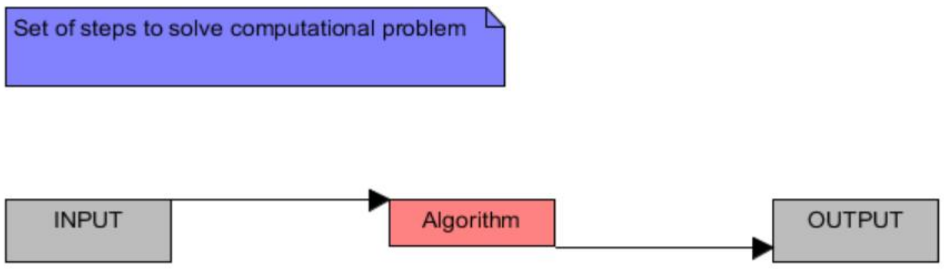
[2]

Activity 2: Compute the Best, Worst and Average cases

In the previous activity, we introduced the necessity of analysis of algorithm and demonstrated how the time and space complexity can be expressed through Theta Notation, Omega Notation and Big O notation.

In this activity, we will look at how to calculate the time and space complexity of the algorithm

Calculating the best and worst case



Input: If input here is an "n" size integer array and we have one "k" integer that we need to look for in that array.

Output: If the k element is found in the array, we must return 1, otherwise we must return 0.

Execution time can be varies depending on the different input. For example, if k is the first element of the array it might take 1 second and if it is 5th place it might take 5 seconds.

Exercise 1

Find the best case and provide worst-case complexity

```
int main ()
{
    int size =4;
    string fruits [size] = {"apple","ball","cat","dog"};
for (int i=0; i<4; i++)
{
if(fruits[i]=="apple")
write ("apple found");
}

    return 0;
}
```

Solution: Here, fruits [0] is equal to apple and it makes only one comparison. If the statement takes constant time for execution. Therefore, the best-case time complexity is O (1).

But if there was no element called apple in the array fruits then it will be the worst-case complexity of O (4).

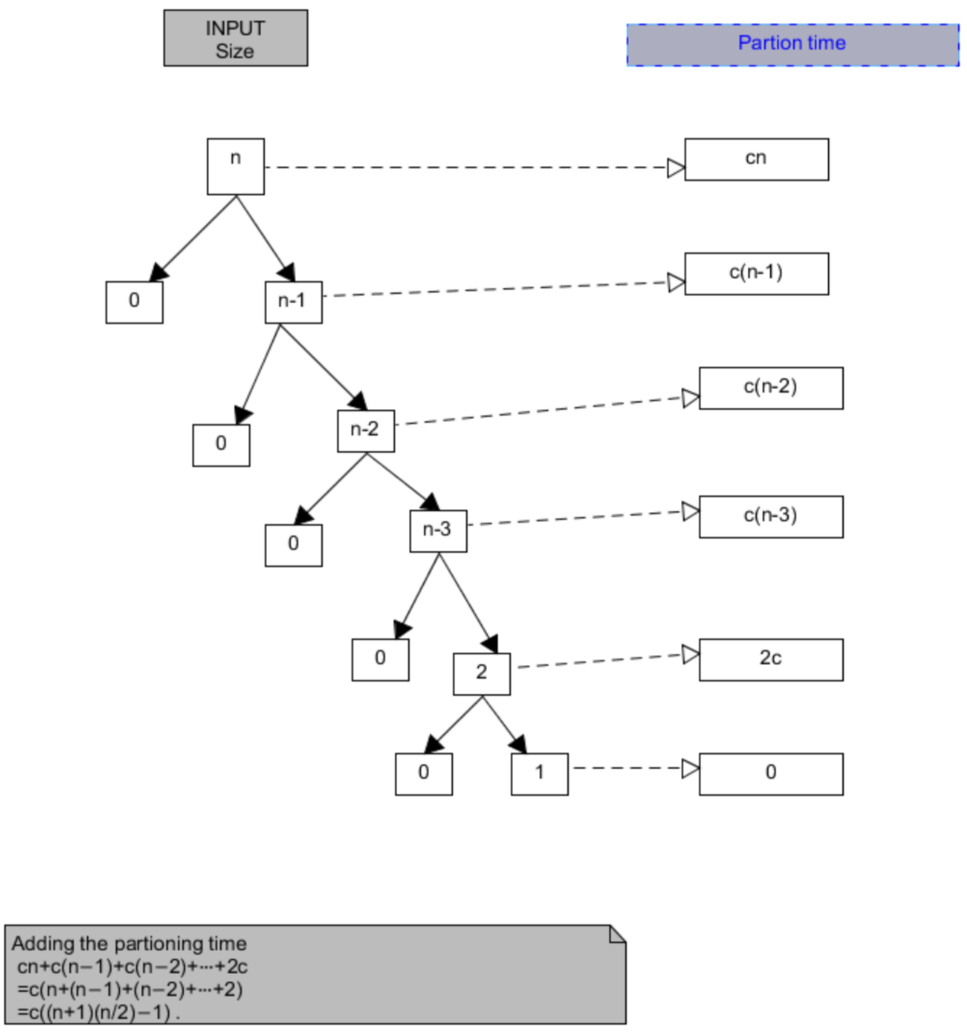
Example of computing Quick-sort algorithm Analysis in detail

Detail analysis quick sort algorithm will provide a good understanding of how to analysis compute time and space complexity in terms of the best, worst and average-case scenario.

Quicksort uses a divide and conquers algorithm. One element from the array is picked as the pivot and the splitting happens around it. There are many ways of picking pivots such as picking random element, the first element, last element or median.

The process is placing the smaller array element before the pivot and greater element after the pivot. The output time of Quicksort depends on the input size and the dividing technique.

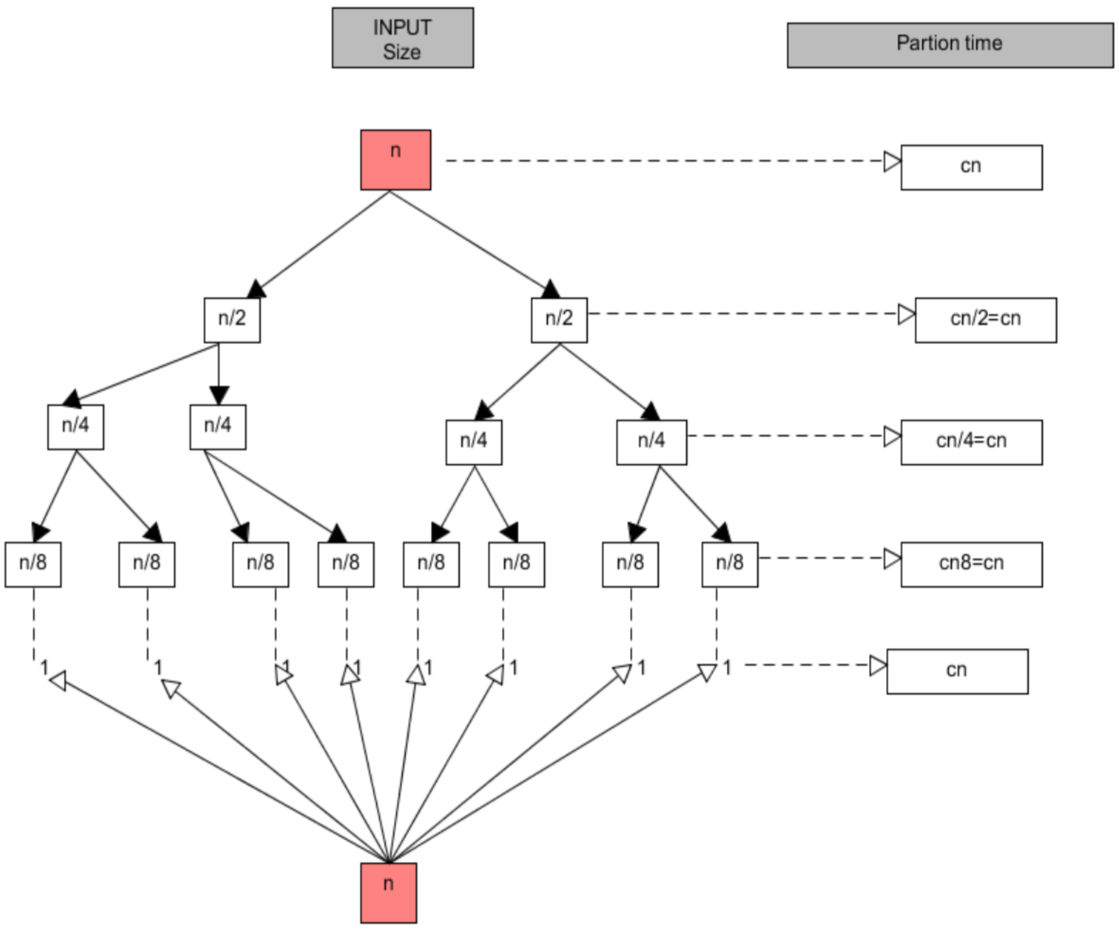
Worst-case



For the worst case, if we select the last element as the pivot and that element is the greatest element of an array which is already sorted worst case occurs because it has to compare with all the elements in the array.

We have c as constant and some lower-order terms. when we use big O notation, we ignore them to get the better estimation. So, the worst-case run time is O(n^2) for quicksort.

Best-case



If the pivot is the middle element of an array it doesn’t have to compare small element to the big element and vice versa. It is only comparing it to only half of the data.

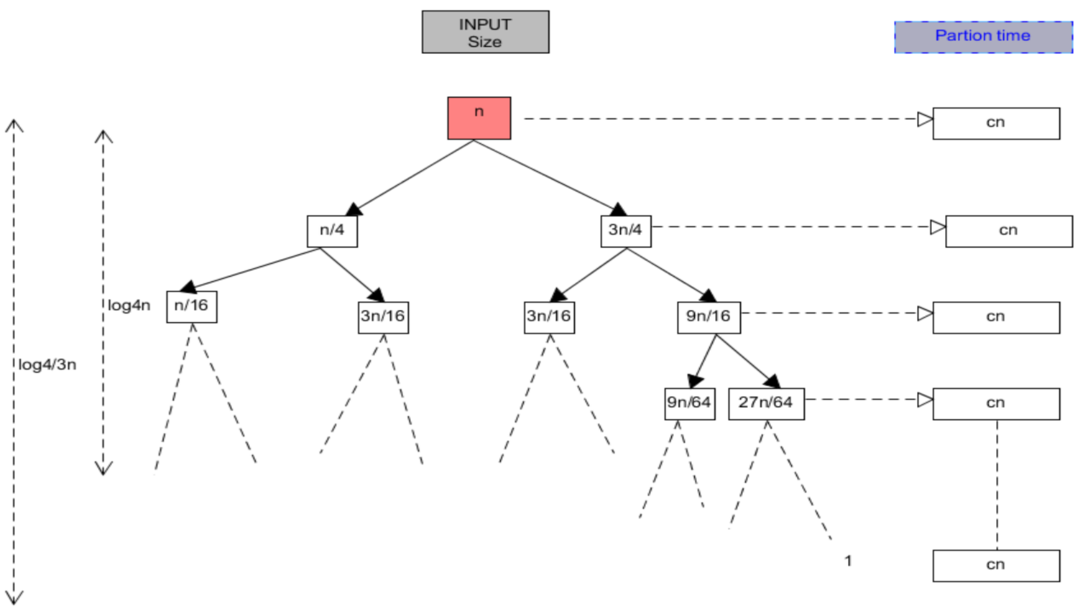
The total time will be some of the partition time for all levels. If we have k level in the tree. Then, we have ***k=log2n+1***.For example if n =16, then long2n+1 =5 as tree will have 5 level i.e., 16,8,4,2,1.

Therefore, Total time will be **cn(log2n+1)** for the best-case. As we use Big O notation for analysing run time, we discard lower order. Thus, the run time will be **O(nlog2n**).

Therefore, for the best-case scenario, it will take O (n log n) time.

Finding the Average-case complexity for Quicksort

To find the average case we need to consider all the permutations of the array and calculate the time taken by every permutation.



In the diagram above, we have split the array 3 to 1. Each time there is partition one side gets 3n/4 of the element and other side get n/4.

We have a smaller input size on the left. So, the left side is divided quicker to size 1. As shown in the figure after log4n level we get to size 1. If we start with problem size on 1 and multiply by 4, we reach will reach n.

For the right side, since it gets the ¾ size of the input multiplying it by 4/3 will give us original n.

Since the constant doesn’t matter in big O notation, the average run-time of Quicksort will be O(nlog2n).

We can split the array differently but the average time complexity will be the same.

[3][4]

Problem exercise:

1. What is the time, space complexity of following code:

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
        a = a + rand();
}
for (j = 0; j < M; j++) {
        b = b + rand();
}
```

2. What is the time complexity of following code:

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j > i; j--) {
                a = a + i + j;
        }
}
```

3. What is the time complexity of following code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
        for (j = 2; j <= n; j = j * 2) {
                k = k + n / 2;
        }
}
```

[5]

Click the link below for the solution:

https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/

Reference:

[1] Yourbasic.org. 2020. *How To Analyze Time Complexity: Count Your Steps*. [online] Available at: <https://yourbasic.org/algorithms/time-complexity-explained/> [Accessed 1October 2020].

[2] Programiz.com. 2020. *Asymptotic Analysis: Theta, Omega and Big-O Notation*. [online] Available at: <https://www.programiz.com/dsa/asymptotic-notations> [Accessed 2 October 2020].

[3] Afteracademy.com. 2020. *Time And Space Complexity Analysis Of Algorithm*. [online] Available at: <https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm> [Accessed 2 October 2020].

[4] GeeksforGeeks. 2020. *Analysis Of Algorithms* | *Set 2 (Worst, Average And Best Cases) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/> [Accessed 3 October 2020].

[5] GeeksforGeeks. 2020. *Practice Questions On-Time Complexity Analysis - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/> [Accessed 3 October 2020].