Notes
on
Design and Analysis
of
Algorithm

By

Dr. Satyasundara Mahapatra

(Module –I)

Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.

Design and Analysis of Algorithms.

This course is intended to be a gentle introduction about the algorithm specification and design strategies. The main problems of the course with its spirit are also introduced. An overview of algorithms and their place in modern computing systems is discussed below.

→ What are algorithms?
→ Why is the study of algorithms worthwhile?
→ What is the role of algorithms relative to other technologies used in computer?

The answers to these questions are play a big role to this course.

Algorithm:

It is nothing but a solution for a problem. The defination of Algorithm" is any well-defined computational procedure that takes some value or set of values, as input and produces some value, or set of values as output." Hence an algorithm is a sequence of computational steps that transform the input into the output.

It is relatively easy to design an algorithm for a problem, that is somehow solve it. However quite some clearness

is needed in designing an algorithm that are also fast in nature. i.e algorithm which give answer very quickly. This will create a major challange in this course.

The main goal of this course is designing fast algorithm, which is a big task. As we know that designing any thing is an art. So in some sense we must be creative with very very well defined design techniques, which are used for this purpose.

The goal of this course is to study these techniques and apply these techniques for developing or synthesizing new ideas which helps the society.

Prerequisites:

- → Programming Concept.
- → Data Structure
- → Discreate Mathematics.

Approach:

- → Analytical
- → Build a mathematical model of a computer
- → Design Algorithm for the models and study their properties. (facts)
- → Reason about the algo and prove the different facts (or properties); about the time taken.

Overview:

→ Few lectures on Basic framework i.e "Fast"?

→ Techniques for fast Algorithms (from different Areas: Optimization, Graph Theory, Geometry).

→ NP - Completeness Theory.

For Example1:

For the Problem: Given two numbers m & n. find their greatest common divisor.

The solutions are two types:

Algo 1: Semple (Learned in School)

Algo 2: Euclid Method.

~~Algo~~: ~~Simple factoring Algo~~.

Input: two integers m & n.

output: Largest integer that divides both.

Algo 1: (Simple Factoring Algo)

1. Factorize m by prime numbers $m_1, m_2, \cdots, m_k$ such that
$m = m_1 \times m_2 \times m_3 \times \cdots m_k$.
2. Factorize n by prime numbers $n_1, n_2, \cdots, n_j$ such that $n = n_1 \times n_2 \times \cdots \times n_j$
3. Identify common factors and then multiply them and return the result.

Algo 2:

```
Euclid (m, n)
{
  While m does not dived n
      r = n mod m;

      n = m;
      m = r;
  end
  return m;
}.
```

Example:

$m = 36$ $n = 48$ Find GCD.

By using Algo 1.

$m = 2 \times 2 \times 3 \times 3$ (i.e Prime factors of 36)

$n = 2 \times 2 \times 2 \times 2 \times 3$ (i.e Prime factors of 48)

The common factors are 2, 2, 3

Hence GCD = $2 \times 2 \times 3 = 12$

By using Algo-2

Euclid (36, 48)

Iteration 1 → 36 divide 48 → Ans → no.

So

$r = 48 \bmod 36.$

$= 12$

$n = 36$

$m = 12$

Iteration 2 → 12 divide 36 → Ans (Yes)

r = 0

so exit from Loop.

return current value of m as GCD i.e 12.

Comparison between Algo 1 and Algo 2

Algo 1

no. of division required to calculate Prime factor for m = 4

no. of division required to calculate Primefactor for n = 5

So total no. of division for calculating primefactor is = 4 + 5 = 9 nos.

Algo 2

Required only two division for two iteration.

So the total no. of division for calculating primefactor is = 2 nos.

Let us take one more example with Large numbers of m and n and see what happend.

Example:

m = 434, n = 966 Find GCD.

By using Algo 1.

m = 2 × 7 × 31

n = 2 × 7 × 139

The common factors are 2, 7

Hence GCD = 2 × 7 = (14)

Algo 2

Euclid (434, 966)

Iteration 1 → 434 divide 966 → Ans no.

$r = 966 \bmod 434 = 98$

$n = 434$

$m = 98$

Iteration 2 → 98 divide 434 → Ans no.

$r = 434 \bmod 98 = 42$

$n = 98$

$m = 42$

Iteration 3 → 42 divide 98 → Ans no.

$r = 98 \bmod 42 = 14$

$n = 42$

$m = 14$

Iteration 4 → 14 divide 42 → Ans yes.

So exit from the loop.

Return current value of $m = (14)$ as GCD.

So the Algo 2 required only 4 numbers & division, which is 1 division per 1 iteration,

But in case of Algo 1 it looks that for m there is only 2 divisions and for n there is only 2 divisions, but that is not quite correct. To check that there is no factor between 2 and 7 we will required checking for

2 and 3, 3 and 5, as well and 7 we have to check that 139 is prime we will again check by many numbers. which create many and many divisions. So divisions than Algo 1 required many more Algo 2 i.e Euclid.

In fact we will see that for Larger numbers the divisions are many more. for Prime factors. But in case of Euclid the result come in very few divisions. and the Algo 2 is terminated.

Next we argue that Euclid is correct and work perfectly. i.e Proof of Correctness of Euclid.

(Note: we don't want to go detail in proof but the requirement is to know about the main idea behind Euclid theorems.

Main idea in fact:

If m divides n, then
GCD(m,n) = m, if not then
GCD(m,n) = GCD(n mod m, m)

Proof: of main idea.

Suppose g is GCD.
then m = ag, n = bg where a, b are relatively prime.

So As the loop of Euclid executes, m & n might change. But their GCD does not.

It was observed that though the value of m & n are changed time to time but their GCD does not. It remain same or we preserved GCD.

As a result if we go out from the loop this is because of m divide n. So the question is why when the loop will be terminated?

Let's see.

At Initial state:-
the variables are : m & n.

after one iteration : n mod m m
the variables are

So what is happened?

After one iteration the value of m changed to n mod m and the value of n changed to m. It was observed that after every iteration the values are decreased. How long Hence the question is How long can it keep on decreasing? Well it has to remain positive. It cannot be zero. become zero. It might decrease at least 1 on each iteration. So the loop exit. which prove that Euclid algorithm is correct.

→ Now we show that why the first argument of Euclid is smaller than the second. This is required only for Analysis.

Euclid. GCD (m, n) where m < n

For example
EuclidGCD (36, 48)
EuclidGCD (48, 36).

for Euclid GCD (48, 36)

$m = 48$, $n = 36$.

So after first iteration the value changed to $m = 36$ and $n = 46$. i.e the first iteration is only required for exchange of data. Hence

Euclid GCD (48, 36) was not Analised and it prove that $m < n$. (i.e m always Less than n)

The main result is now to prove:

Theorem:

If Euclid is called with value $p \& q$ i.e Euclid (p, q) then if $p < q$, then in each iteration the sum of the values of variables of m, n will decrease at least by a factor $3/2$.

i.e. if sum is 6 down to 4
to → 40

The implication of this theorem is that the sum will drop very fast and the no. of iteration

no. of iteration $\leq \log_{3/2} p+q$.

Beginning of iteration

$m = p$

$n = q$.

After the one iteration

$m = p'$

$n = q'$

Our goal is Estimate $\frac{p+q}{p'+q'}$ (p+q upon p'+q' and prove that this ratio is 3/2

As we know that.

The new value of n is the old value of m

So.

$$n = q' = p.$$

The new value of m is the old value of n mod old value of m.

So $$m = p' = q \bmod p. < p.$$

Another fact is

$$p' + q' = \text{remainder} + \text{divisor}.$$

{divisor < dividend}

p q.

$$\leq \text{dividend}$$

$$= q.$$

Now it is just a matter of Algebra.

After we do the addition.

$$p' + q' + 2(p'+q') \leq p + p + 2(q).$$

$$3(p'+q') \leq 2(p+q)$$

$$\text{So} \quad p'+q' < \frac{p+q}{3/2}$$

Which conclude the analysis of Algorithm.

Conclusion of our discussion.

* Study properties of Euclid computing and this helps in designing fast algorithm.
* The basic way we did this analysis i.e counting iterations will be useful in rest of the course.

Example-2

Let us take a sorting algorithm named as insertion sort for Analysis. This algorithm sort a sequence of numbers into nondecreasing order. The formal definition of sorting problem is given below.

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

Output : A permutation (reording) $\langle a_1', a_2', \cdots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

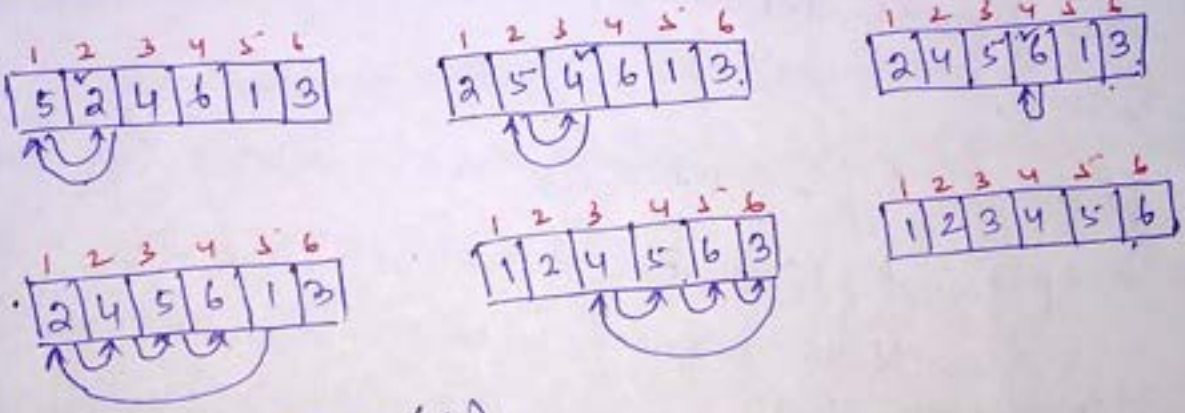
Insertion sort is an efficient algorithm for sorting a small number of elements. This algorithm works the way many people sort a hand of playing cards.

Let us start with an empty left hand and the cards face down on the table to learn the basic procedure of Insertion sort. Remove one card at a time from the table and insert it into the correct position in the left hand.

To find the correct position for a card, the comparision required for it with

each of the cards already in hand. from right to left.

The pseudocode for insertion sort as a procedure called Insertion Sort, which takes as a parameter an array A[1..n] containing a sequence of length n that is to be sorted. This algorithm sorts the input value in place. i.e rearrange the value within the same array A. Hence the input array A contain the sorted output sequence after the insertion sort procedure is finished. as shown below.



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

true before the next iteration.

The INSERTION SORT algorithm works on the following array A.

$A = \langle 5, 2, 4, 6, 1, 3 \rangle$

The index j indicates the "current card" being inserted into the hand.

At the beginning of each iteration of the for loop for index j, the subarray consisting of elements $A[1..j-1]$ already sorted on hand, and the remaining subarray $A[j+1..n]$ corresponds to the pile of cards available on the table.

The fact ~~~~ is the elements of $A[1..j-1]$ are the elements originally in positions 1 through $j-1$, are now in sorted order. These properties of $A[1..j-1]$ is know as loop invariant and stated as follow:

"At the start of each iteration of the for loop of line 1-8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order".

The loop invariant help us to understand why an algorithm is correct. The three important things about the loop invariant is given below.

1. Initialization: it is true prior to the first iteration of the loop.
2. Maintenance: if it is true before an iteration of the loop, it remains true before the next iteration.

3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Let us see how these properties hold for insertion sort.

Initialization:

We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1..j-1]$, therefore consists of only single element i.e $A[1]$. which is actually the original element in $A[1]$. This subarray is sorted and implies that the loop invariant holds prior to first iteration of the loop.

Maintenance:

The second property shows that, each iteration maintains the loop invariant. The body of the for loop i.e (lines 4-7) works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$ and so on by one position to right until all the values of array are sorted. or find their actual positions in that array. i.e the point where the value is sorted. (line no. 8) in position

A[j]. The subarray A[1..j] then consists of the elements originally in A[1..j] but in sorted order. Incrementing j for the next iteration of the for loop i.e line no. 1 preserves the loop invariant.

Another formal treatment of the second property also show a loop invariant for the while loop (i.e line 5 ->).

Termination: At Last we check, what happens when the loop terminate. i.e when j > A.length. This is because the j value is incremented by 1 after each ~~loop~~ iteration. ~~At last~~ At Last the value of j = n+1. and substituting n+1 for j satisfy the condition of loop invariant. At this point the sub array A[1..n] or A[1..j] consists of the elements originally but in sorted order.

Hence the algorithm is correct.

Analysis of INSERTION SORT.

The actual time taken by the INSERTION SORT Procedure is depend on the Input. In general

The time taken by an algorithm grows with the size of the input. Hence to describe the running time of a program as a function of the size of its input. This depend on the two terms i.e "running time" and "size of input".

Input Size:

For a given problem the input size n can be characterize appropriately as follows:

→ Sorting :- The no. of item to be sorted
→ Graphs :- The no. of vertices or edges.
→ Numerical :- The no. of bits needed to represent a number.

The choice of an input is depends on the elementary operation of an algorithm. i.e comparisons, Additions and multiplication.

Running Time:

The running time of an algorithm or a particular input is the number of primitive operations or 'steps' executed. As we know that a constant amount of time is required to execute each line of an algorithm. One line may take a different amount of time than another line.

Lets assume that each execution of the i^{th} line take c_i time as constant. For calculating

The running time of INSERTION SORT uses notation cost c_i for all the statement or line. which is more concise and more easily manipulated. This notation will also play a bigger role, whether one algorithm is more efficient or than another.

Let us analyse the Insertion sort procedure with the time "cost" of each statement and the number of times each statement is executed.

where

c_i represents for cost, which calculate the cost of each execution of i^{th} line.

t_j represents for number of times each statement executed.

INSERTION SORT (A)	cost.	time.
1. for j = 2 to A.length	c_1	n
2. key = A[j]	c_2	$n-1$
3. // Insert A[j] into the sorted sequence A[1..j-1]	$c_3 = 0$	$n-1$
4. i = j-1	c_4	$n-1$
5. while i>0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6. A[i+1] = A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. i = i-1	c_7	$\sum_{j=2}^{n} (t_j-1)$
8. A[i+1] = key	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed. i.e a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to

the total running time. To compute T(n), the running time of INSERTION-SORT of an input of n values, is the sum of the product of the cost and time column:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

where

$$\sum_{j=2}^{n} t_j = 2 + 3 + 4 \cdots + n$$

$$= \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (t_j - 1) = 1 + 2 + 3 + \cdots + n - 1$$

$$= \frac{n(n+1)}{2} - n$$

$$= \frac{n(n-1)}{2}$$

$$\Rightarrow T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8 (n-1)$$

As $c_3 = 0$.

$$= c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8 (n-1)$$

$$= c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 \left(\frac{n^2 + n - 2}{2} \right) + c_6 \left(\frac{n^2 - n}{2} \right) + c_7 \left(\frac{n^2 - n}{2} \right) + c_8 n - c_8$$

$$= c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 \frac{n^2}{2} + c_5 \frac{n}{2} - c_5 - c_6 \frac{n^2}{2} - c_6 \frac{n}{2} + c_7 \frac{n^2}{2} - c_7 + c_8 n - c_8$$

$$\Rightarrow \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8).$$

Let $\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{8} = a$

$$c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} = b.$$

and $-(c_2 + c_4 + c_5 + c_8) = c$

$\Rightarrow$ $an^2 + bn + c$ for constants a, b and c, which depend on the statement costs c_i. It is thus a quadratic function of n. Thus the worst case running time of Insertion Sort is in total $= an^2 + bn + c$, the longest running time for any input size n. i.e $O(n^2)$

Best case Analysis of Insertion Sort.

INSERTION-SORT (A)	Cost	time
1. for $j \leftarrow 2$ to A.length.	c_1	n
2. Key $\leftarrow$ A[j]	c_2	n-1
3. // Insert A[j] into the sorted sequence A[1..j-1]	$c_3 = 0$	n-1
4. $i \leftarrow j-1$	c_4	n-1
5. while $i > 0$ and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6. A[i+1] $\leftarrow$ A[i]	c_6	0
7. $i \leftarrow i-1$	c_7	0
8. A[i+1] $\leftarrow$ key	c_8	n-1

Hence

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5\left(\sum_{j=2}^{n} t_j\right) + c_6 \cdot 0 + c_7 \cdot 0 + c_8(n-1)$$

$$= c_1 n + c_2 n - c_2 + 0 + c_4 n - c_4 + c_5\left(\sum_{j=2}^{n} t_j\right) + 0 + 0 + c_8 n - c_8 .$$

In case of Best case the t_j is alway 1.so

$t_j = 1$. $\because \sum_{j=2}^{n} (1) = n-1$

$$\Rightarrow c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5(n-1) + c_8 n - c_8$$

$$\Rightarrow c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 n - c_5 + c_8 n - c_8$$

$$\Rightarrow (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

$$\Rightarrow an - b$$

Hence the complexity is $O(n)$.

Example : 3.

Analysis of Bubble Sort.

Lets apply a Bubble Sort Algo on following data set.

$$A = \langle 54, 26, 93, 17, 77 \rangle$$

Pass 1:

54 26 93 17 77 — Exchange between A_1 & A_2

26 54 93 17 77. — no Exchange.

26 54 93 17 77 — Exchange between A_3 & A_4

26 54 17 93 77. — Exchange between A_4 & A_5

26 54 17 77 93.

Pass-2:

26 54 17 77 93. No Exchange.

26 54 17 77 93. Exchange between A_2 & A_3

26 17 54 77 93. No Exchange.

26 17 54 77 93. [77 got its actual Position]

Pass-3:

26 17 54 77 93. Exchange between A_1 & A_2

17 26 54 77 93. No Exchange

17 26 54 77 93 [54 got its actual Position]

Pass-4:

17 26 54 77 93. No Exchange.

17 26 54 77 93 [26 got its actual Position]

Hence the final sorted array is as follows.

$\langle 17\ 26\ 54\ 77\ 93 \rangle$

Bubble Sort (A).

	Cost	Time
1. for $i \leftarrow 1$ to $n-1$ do,	c_1	n
2. for $j \leftarrow 1$ to $n-i$ do,	c_2	$\sum_{j=1}^{n-i} t_j$
3. If $a[j] > a[j+1]$	c_3	$\sum_{j=1}^{n-i} t_j - 1$)
4. then swap $(a[j], a[j+1])$	c_4	$\sum_{j=1}^{n-i} t_j - 1$

$$T(n) = c_1 n + c_2 \left(\sum_{j=1}^{n-i} t_i\right) + c_3 \left(\sum_{j=1}^{n-i} t_j - 1\right) + c_4 \left(\sum_{j=1}^{n-i} t_j - 1\right)$$

{ where $i = 1, 2, 3 \ldots$ }.

Hence for worst case analysis i start from 1.

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

Similarly

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

$$= \frac{n(n+1)}{2} - 2n + 1$$

$$= \frac{n(n+1) - 2n + 2}{2}$$

$$= \frac{n^2 + n - 4n + 4}{2}$$

$$= \frac{n^2 - 3n + 4}{2}$$

Hence

$$T(n) = c_1 n + c_2 \left(\frac{n(n-1)}{2} \right) + c_3 \left(\frac{n^2 - 3n + 4}{2} \right) + c_4 \left(\frac{n^2 - 3n + 4}{2} \right)$$

After solving the above equation it comes in the form of quadratic equation:

i.e $an^2 + bn + c$

$\therefore$ Time complexity of Bubble sort is $O(n^2)$.
worst

The best case analysis of Bubble sort is discussed in next page.

Best case Analysis of Bubble Sort.

Bubble Sort (A)	Cost	Time
1. for $i \leftarrow 1$ to $n-1$	c_1	n
2. for $j \leftarrow 1$ to $n-i$	c_2	$\sum_{j=1}^{n-i} t_j$
3. if $(a[j] > a[j+1])$	c_3	0
4. swap $(a[j], a[j+1])$	c_4	0.

for best case t_j value for each iteration is always 1. Therefore $\sum_{j=1}^{n-i} t_j = n$.

$$T(n) = c_1 n + c_2 \left(\sum_{j=1}^{n-i} t_j \right) + c_3 0 + c_4 0$$

$$= c_1 n + c_2 n + 0 + 0.$$

$$= n(c_1 + c_2).$$

So $T(n) = O(n)$.

Example - 4 (Analysis of Selection Sort.)

Q. Analyze the best case and worst case of Time complexity of Selection sort. (Home Assignment).

Selection Sort (A)

1. for $i \leftarrow 1$ to $n-1$ do.
2. $min \leftarrow i$
3. for $j \leftarrow i+1$ to n do.
4. if $a[j] < a[min]$
5. $min \leftarrow j$
6. swap $(a[i], a[min])$.

Best case - $O(n)$
Worst case - $O(n^2)$.

Example: 5 (Analysis of Merge Sort)

There are many ways to design algorithms. For example insertion sort is incremental in nature. (i.e having sorted $A[1..j-1]$, place $A[j]$ correctly, so that $A[1..j]$ is sorted.

Another approach is Divide and conquer.

Divide: the problem into number of subproblems

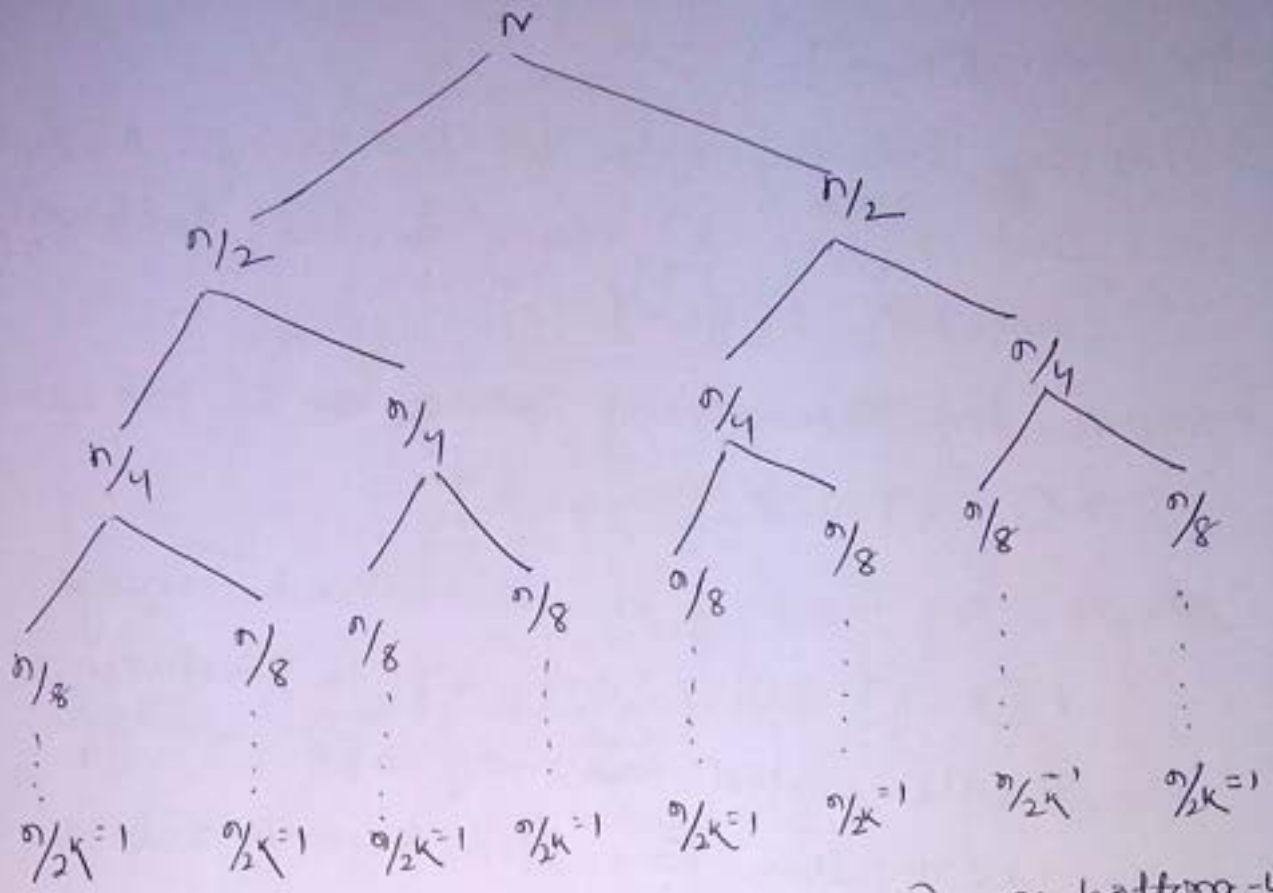
Conquer: the subproblems by solving them recursively.

Base case: If the problems are small enough, just solve them by brute force.

Combine: the subproblem solutions to give a solution to the original problem.

Mergesort is based on the ~~paradigm~~ Divide, conquer and Combine paradigm. The divide phase is a top-down approach, where the input array is split into half recursively until the array size reduces to one. i.e Forcee a given problem of size n, break it into two subproblems of size $n/2$, again break each of these subproblems into two subproblems of size $n/4$ and so on till the subproblem size reduces to $n/2^k = 1$

for some integer k. as given below figure



The conquer phase is based on bottom-up approach. i.e combination of two sorted array of size one to get the sorted arrays of size two, and combine two sorted arrays of size two to get a sorted array of size 4. and so on. In general combination of two sorted array of size n/2 produced a sorted array of size n. Conquer phase happens when the recursion bottoms out and makes use of a black box which takes two sorted array and merges these two to produce one sorted array

In general, we are dealing with subproblems. we state each subproblem as sorting a subarray A[p..r]. Initially, p=1 and r=n.

But these values change as we recurse through subproblems.

To sort $A[p..r]$

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$ where q is the halfway point of $A[p..r]$

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$

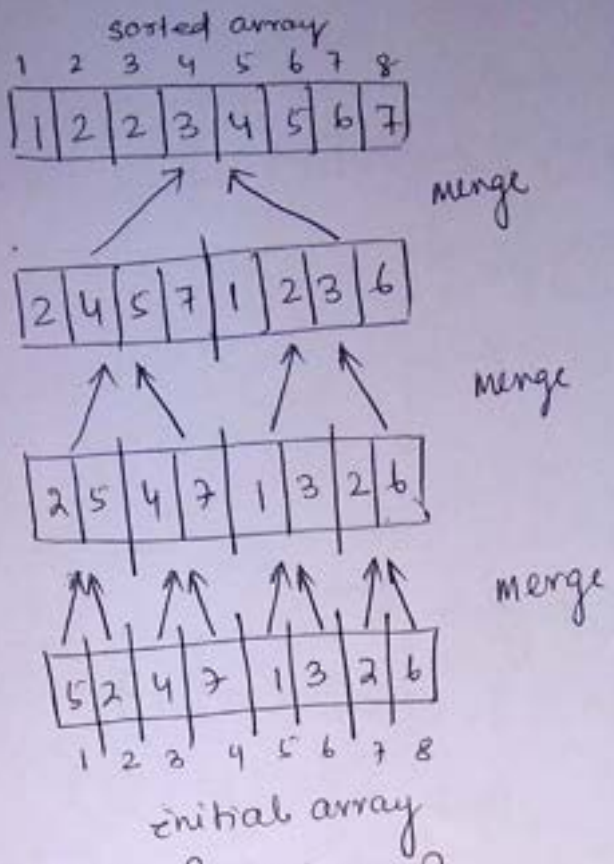
Combine: by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we will define a procedure MERGE (A, p, q, r)

The recursion bottoms stop when the subarray has just 1 (one) element.

```
MERGE-SORT (A, p, r)
if (p < r)                          → check for base case
  then q ← ⌊(p+r)/2⌋                → Divide
       MERGE-SORT (A, p, q)         → Conquer
       MERGE-SORT (A, q+1, r)       → Conquer
       MERGE (A, p, q, r)           → Combine.
```

Initial call: MERGE-SORT $(A, 1, n)$

Lets see a bottom-up view for $n = 8$ (i.e combine.



What is Merging Procedure?

Input: Array A and indices p, q, r such that

→ $p \leq q < r$.

→ Subarray $A[p..q]$ is sorted and subarray $A[q+1..r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

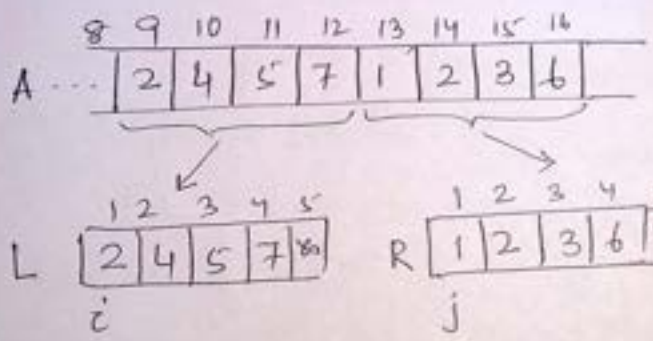
Output: The two subarrays are merged into a single sorted subarray in $A[p..r]$

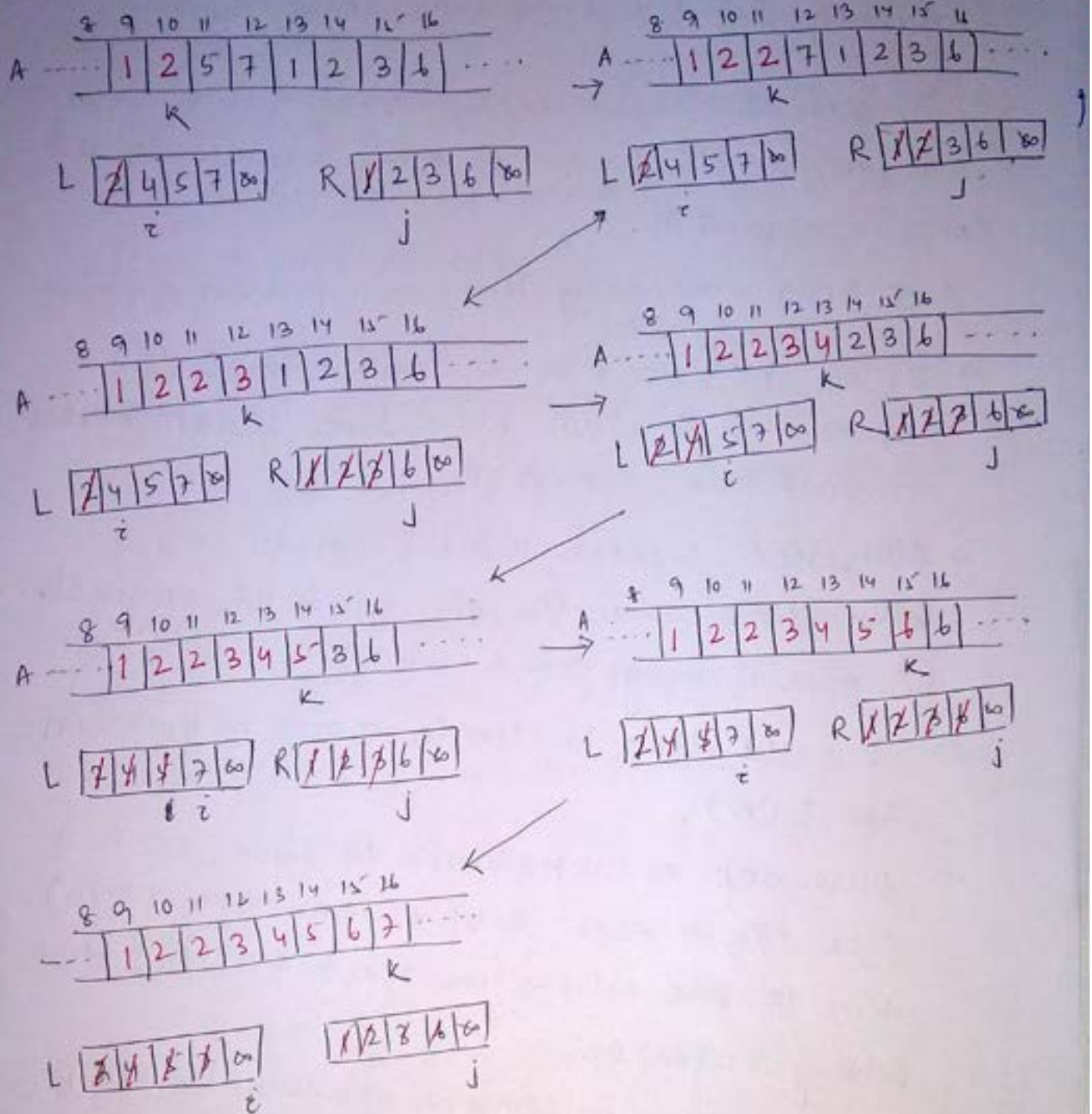
Pseudocode

```
MERGE(A, p, q, r)
n1 ← p+q-1
n2 ← r-q
create arrays L[1..n1+1] and R[1..n2+1]
for i ← 1 to n1
    do L[i] ← A[p+i-1]
```

```
for j ← 1 to n2
    do R[j] ← A[q+j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i = 1
j = 1.
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i+1
          else A[k] ← R[j]
               j ← j+1
```

Example : A call of MERGE (A, 9, 12, 16)





Running time of Merging:

The first two for loop take $\Theta(n_1+n_2)=\Theta(n)$ time. The last for loop makes n iterations, each taking constant time, for $\Theta(n)$ time.

Hence total time $= \Theta(n)$.

Analyzing divid and conquer Algorithms

In general recurrence equation are used to describe the running time of a divide-and conquer algorithm.

Let T(n) = running time on a problem of size n

→ If the problem size is small enough (i.e basecase) then the brute-force solution takes constant time. i.e $\Theta(1)$.

→ Otherwise, suppose that we divide into a subproblems, each 1/b the size of the original. (In case of merge sort, a = b = 2).

→ Let the time to divide a size n problem be D(n).

→ There are a subproblems to solve, each of size n/b ⇒ each subproblems take T(n/b) time to ~~solve~~ solve ⇒ we spend aT(n/b) time solving subproblems.

→ Let the time to combine solutions be C(n)

Hence the recurrence equation for Merge sort is :-

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

Analyzing Merge Sort.

For simplicity assume that n is power of 2 (i.e 2^k)
$\Rightarrow$ each divide step yields two sub problems both of size exactly $n/2$.

One base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and r
$\Rightarrow D(n) = \Theta(1)$

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since
Hence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(n). & \text{if } n > 1 \end{cases}$$

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together, give function that is linear in n:
$\Theta(n)$, Hence the recurrence for merge sort running time is modified as follows.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

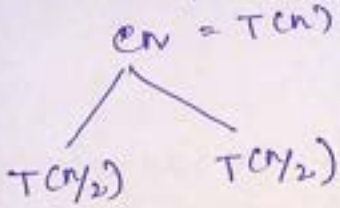
→ Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps.

Hence we rewrite the recurrence as follows

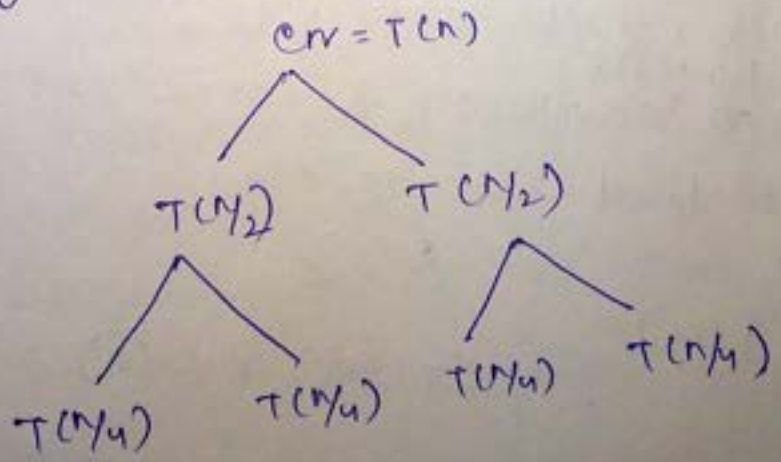
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

By drawing a recursion tree, the successive expansions of the recurrence is explained below.

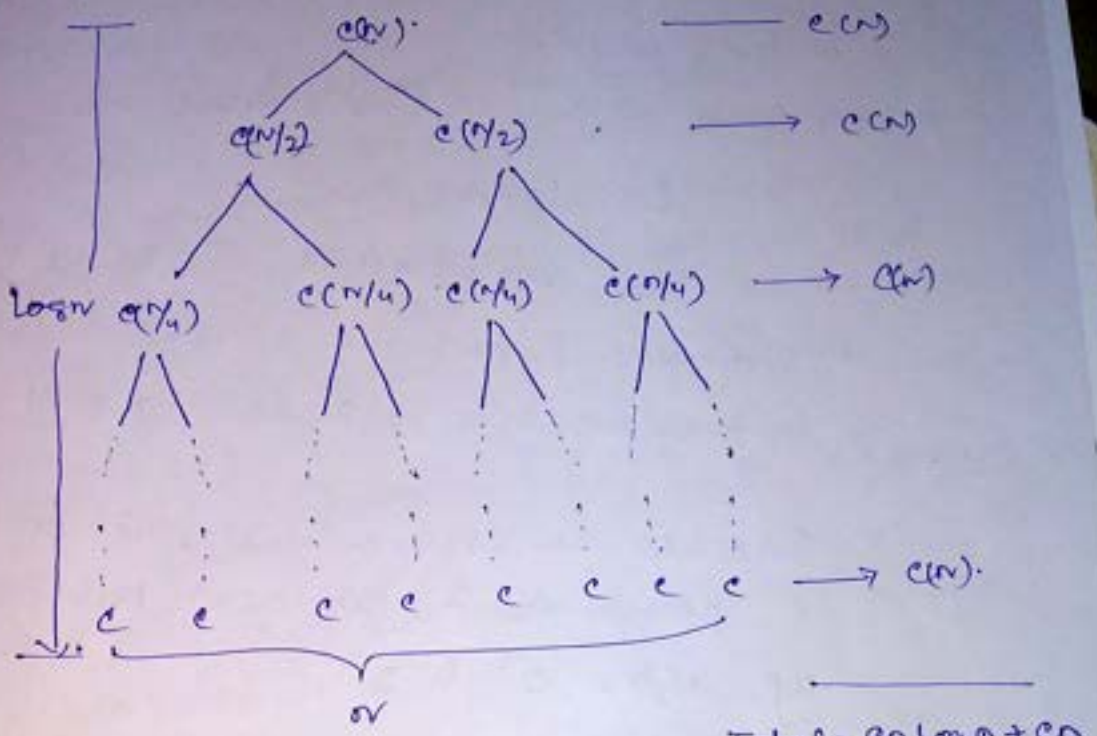
→ For the original problem, we have a cost of cn, plus two subproblems, each costing $T(n/2)$:



→ For each of the size $n/2$ subproblems, we have a cost of $c(n/2)$, plus two subproblems, each costing $T(n/4)$:



→ Continue expanding until the problem sizes get down to 1:



Explanation

→ Each level has cost cn
  ⇒ The top level has cost cn
  ⇒ The next level has 2 subproblems. each contributing cost c(n/2).
  ⇒ The next level has 4 subproblems, each contributing cost c(n/4)
⇒ It was observed that as we go down one level, the number of subproblems doubles, but the cost per subproblem halves ⇒ cost per level stays the same.

→ It was also observed that there are $\lg n + 1$ levels (i.e height of tree is $\lg n$).

⇒ use induction

⇒ Base case : $n = 1 \Rightarrow$ 1 level

for $\lg n + 1$ level

$\lg 1 + 1 = 0 + 1 = 1$.

⇒ Inductive hypothesis is that a tree for problem size of 2 has $\lg 2^i + 1 = i + 1$ level

⇒ Because we assume that the problem size is power of 2, the next problem size up after 2^i is 2^{i+1}

⇒ A tree for a problem size of 2^{i+1} has one more level than the size 2^i tree

⇒ $i + 2$ levels.

⇒ Since $\lg 2^{i+1} + 1 = i + 2$, we have done with inductive argument.

→ The total cost is the sum of costs at each level. Have $\lg n + 1$ level, each costing $cn \Rightarrow$ total cost is $cn \lg n + cn$.

→ By ignoring low-order term of cn and constant coefficient $c \Rightarrow O(n \lg n)$

Iterative analysis of Mergesort. Recurrence relation.

The basic operation in the mergesort is the key comparison. It is evident that mergesort on just one element requires only one comparison, and Let c_1 be the time for this. when $n > 1$. The total task of merge sort divided into three parts. (i.e Divide, conquer and combine). Let $T(n)$ denote the time required to sort n elements. The recurrence of $T(n)$ is given below.

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ D(n) + T\left(\frac{n}{2}\right) + T\left(n/2\right) + C(n) & \text{if } n > 1 \end{cases}$$

Here $D(n)$ represent the time to divide the task.

Let $D(n) = c_2$

$C(n)$ represent the time to merge two list so that the output list is of size n.

In order to simplify the calculation, let us consider the following assumptions.

1. no. of elements in the list is a power of two, i.e $n = 2^k$
2. Consider the merge operation whose worst case time complexity is $C(n) = n$.

Hence the above equation can be written as follow,

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ c_2 + 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Let us solve the recurrence relation by using iterative method

$$T(n) = 2T(n/2) + c_2 + n$$

$$= 2\left[2T(n/4) + n/2 + c_2\right] + n + c_2$$

$$= 2^2\left[2T(n/2^2) + \right] + n + c_2 + n + c_2$$

$$\Rightarrow 2^2 T\left(\frac{n}{2^2}\right) + 2n + 2c_2$$

$$\Rightarrow 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{4} + c_2\right] + 2n + 2c_2$$

$$\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + n + c_2 + 2n + 2c_2$$

$$\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + 3n + 3c_2$$

$$\vdots \quad \vdots \quad \vdots$$

for k^{th} Term

$$\Rightarrow 2^k T\left(\frac{n}{2^k}\right) + kn + kc_2$$

Since $n = 2^k$ and $T(1) = c_1$ finaly, we get.

$$\Rightarrow n\,T(1) + n\log_2 n + nc_2$$

where. $\log n = 2^k$.

Apply both side log.

$$\log n = \log 2^k$$

$$= k \log 2, \quad \because \log 2 = 1.$$

$$\therefore k = \log_2 n$$

So $T(n) = n c_1 + n \log_2 n + c_2 n$

Divide — Conquer — Combine

The best case occurs when the list to be sorted is almost sorted in order. In such a situation, the merge operation requires $n/2$ operations and time complexity can be obtained as

$$T(n) = O(n \log_2 n) + O(n).$$

As we know that the goal of Design and analysis are:

1. Design Algorithm.
2. Analysis time taken on RAM model.
3. Entire analysis / detail analysis not applicable to other computers.

Let's take some example to calculate the time required on the RAM.

$10n^3 + 5n^2 + 7$ → "Cubic in n" in one RAM

$2n^3 + 3n + 79$ → "Cubic in n" in Another RAM

Hence the conclusion ~~comes generated that~~ will be for any computer is 'cubic in n' for all computers. and also conclud a thing that the above two fonctions are come in one class. or same class. In next page we discuss the classes of functions.

Classes of functions: (Growth of a function)

Asymptotic notation : formal way or notation to speak about functions and classify them.

Asymptotic Analysis : classifying the behaviours of a functions

To classify the functions we need to know about the two kind of features. They are given below.

1. Functions $10n^3 + 5n^2 + 17$

$2n^3 + 3n + 79$

should 'go into same class.

"Constant multipliers should be ignored".

which is our desired.

2. Give more importance to behaviour as $n \to \infty$

Let us discuss three main type of notation which required to classify the functions. They are:

1. Θ - notation
2. O - notation
3. Ω - notation.

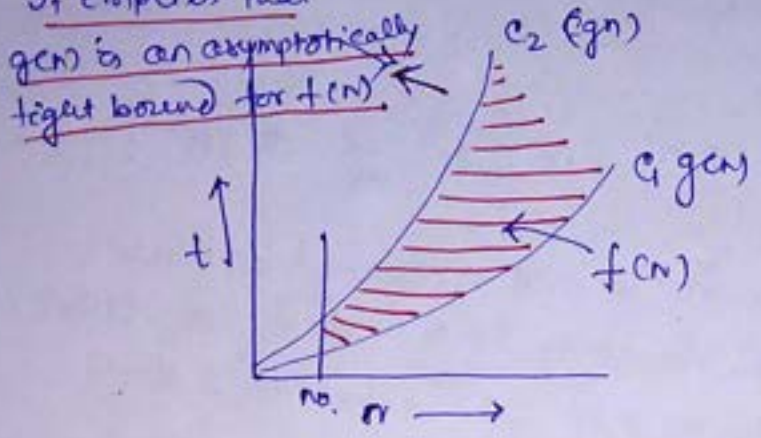
1. Θ - notation:- (Theta)

Lets take to functions $f()$ & $g()$ as non-negative functions of non negative arguments.

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. $\}$

* It determine the tight bound.
* It find average time complexity
* If $f(n) = \Theta(g(n))$ then $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ where $c > 0$ and $n_0 \geq 1$

It implies that $g(n)$ is an asymptotically tight bound for $f(n)$.



Lets take few examples:

Example 1: $f_1(n) = 10n^3 + 5n^2 + 17 \in \Theta(n^3)$

Proof of Ex-1:

It was clearly visible that.

$10n^3 \leq f_1(n)$

$\Rightarrow 10n^3 \leq f_1(n) \leq (10+5+17)n^3 = 32n^3$.

Hence $c_1 = 10$ and $c_2 = 32$.

$\therefore c_1 n^3 \leq f(n) \leq c_2 n^3$ for all $n \geq 1 = n_0$.

So we conclude that $f(n)$ belongs to $\Theta(n^3)$ class.

Example-2: $f(n) = 2n^3 + 3n + 79 \in \Theta(n^3)$.

Proof of Ex-2

It was clearly visible that

$2n^3 \leq f(n)$.

$\Rightarrow 2n^3 \leq f(n) \leq (2+3+79)n^3 = 84n^3$.

Hence $c_1 = 2$ and $c_2 = 84$

which implies that

$$c_1(n^3) \leq f(n) \leq c_2(n^3) \text{ for all } n \geq 1 = n_0.$$

Therefore we conclude that $f(n) \in \Theta(n^3)$ class

Example. 3

$$f_8(n) = 10n^3 + n \lg n \in \Theta(n^3).$$

This function is not belonging to "cubic class". Because a cubic function looks like a cubic polynomial. It has a form of $ax^3 + bx^2 + cx + d \ldots$.

Hovere.

$10^3 \leq f(n)$ As $n \log n < n^3$.

$\Rightarrow 10n^3 \leq f(n). \leq 11n^3.$

$c_1 = 10$ and, $c_2 = 11$, and $n_0 = 1$.

for which. $c_1(n^3) \leq f(n) \leq c_2(n^3)$.

Therefore we conclude that $f(n) \in \Theta(n^3)$. class.

Example - 4

$$f(n) = 27n^2 + 11n \in \Theta(n^2)$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$27n^2 \leq f(n) \leq (27 + 16)n^2$$

So $c_1 = 27$, $c_2 = 43$ $\forall$ $n \geq 1 = n_0$.

Hence $27n^2 + 16n \in \Theta(n^2)$

Example - 5

$$f(n) = 3n+2 \in \Theta(n).$$

as $3n+2 \geq 3n$ for all $n \geq 1$ and

$3n+2 \leq 4n$ for all $n \geq 2$

so $c_1 = 3$ and $c_2 = 4$ $\forall$ $n \geq 2$. Hence.

$$3n+2 \in \Theta(n).$$

Example - b (Method 1).

$f(n) = (n+a)^b$. $\in \Theta(n^b)$.

as. $0 \leq c_1 n^b \leq (n+a)^b \leq c_2 n^b$. $\forall$ $n \geq n_0$.

note that

$n+a \leq n+|a| \leq 2n$ where $|a| \leq n$.

and

$n+a \geq n-|a| \geq \frac{1}{2}n$ where $|a| \leq \frac{n}{2}$
$\Rightarrow 2|a| \leq n$

when

$$0 \leq \frac{1}{2}n \leq n+a \leq 2n.$$

Since $b > 0$, the inequality still holds when all parts are raised to the power of b, then

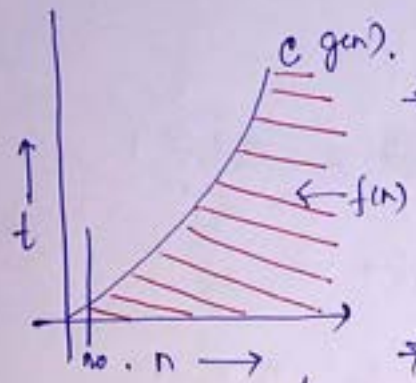
$$0 \leq \left(\frac{1}{2}n\right)^b \leq (n+a)^b \leq (2n)^b$$

$$\Rightarrow 0 \leq \left(\frac{1}{2}\right)^b n^b \leq (n+a)^b \leq (2)^b n^b$$

Thus $c_1 = (\frac{1}{2})^b$, $c_2 = (2)^b$ $\forall$ $n_0 \geq 2|a|$
satisfy the definition. $(n+a)^b \in \Theta(n^b)$.

2. O - notation (Big-Oh)

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0 \}$.



→ It determines upperbound time complexity (i.e worst case time complexity).

→ If $f(n) = O(g(n))$ then

$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$ $\forall$ $c > 0$, $n_0 \geq 1$

↓

It implies that $g(n)$ is an asymptotic upper-bound for $f(n)$

Lets solve some examples.

Example - 1

$f(n) = 3n+2 \in O(n)$

as $3n+2 \leq 4n$ $\forall$ $n \geq 2$.

Hence $c = 4$ and $n_0 = 2$, Hence

$3n+2 \in O(n)$.

Example - 2

$f(n) = 10n^2 + 4n + 2 \in O(n^2)$.

as $10n^2 + 4n + 2 \leq 11n^2$ $\forall$ $n \geq 5$

Hence $c = 11$ and $n_0 = 5$, Hence

$10n^2 + 4n + 2 \in O(n^2)$

Example-3

$f(n) = 1000n^2 + 100n - 6 \in O(n^2)$

as $1000n^2 + 100n - 6 \leq 1001n^2 \quad \forall\ n \geq 100$.

So $c = 1001$, $n_0 = 100$ Hence

$1000n^2 + 100n - 6 \in O(n^2)$.

Example-4

$f(n) = 6 \cdot 2^n + n^2 \in O(2^n)$.

as $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n \quad \forall\ n \geq 4$

So $c = 7$, $n_0 = 4$ Hence

$6 \cdot 2^n + n^2 \in O(2^n)$.

Example-5 $f(n) = 2^{n+1} \in O(2^n)$.

Method-1

as $2^{n+1} = 2^n \cdot 2$ for all $n \geq 1$

$= 2 \cdot 2^n$.

$f(n)$. c $g(n)$.

As per the definition of Big Oh.

$f(n) \leq c \cdot g(n)$.

$2^{n+1} \leq 2 \cdot 2^n$ for all $n \geq 1$

So $c = 2$ and $n_0 = 1$

Hence $2^{n+1} \in O(2^n)$.

Method - 2

As per the Big oh definition

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c.$$

$$\Rightarrow \lim_{n \to \infty} \frac{2^{n+1}}{2^n} \leq c.$$

$$\Rightarrow \lim_{n \to \infty} \frac{2^n \cdot 2^1}{2^n} \leq c$$

$$\Rightarrow \lim_{n \to \infty} 2 \leq c \quad \text{is True.}$$

Hence $2^{n+1} \in O(2^n)$.

Example: 6. $f(n) = 2^{2n} \notin O(2^n)$

Method - 1

As $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n$

where $c \Rightarrow 2^n$ is not true as no constant is greater than all 2^n. which creats an contradiction.

$\therefore 2^{2n} \notin O(2^n)$.

Method 2

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

$$\lim_{n \to \infty} \frac{g^{2n}}{2^n} \leq c$$

$$\lim_{n \to \infty} \frac{2^n \cdot 2^n}{2^n} \leq c.$$

$$\lim_{n \to \infty} 2^n \leq c.$$

$\Downarrow$

is not true as no constant is greater than all 2^n. Hence $2^{2n} \notin O(2^n)$

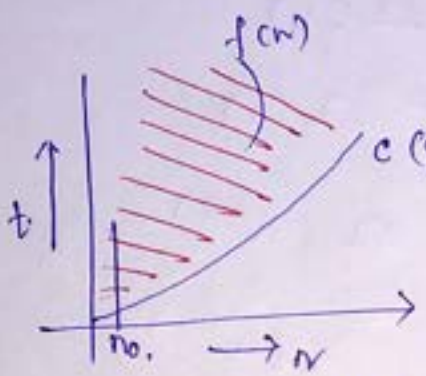
3. Ω notation (Omega)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$.

→ It determine lower bound time complexity (i.e best case time complexity)

→ If $f(n) = \Omega(g(n))$ then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq c$$



→ It implies that g(n) is an asymptotic lower bound for f(n).

Lets solve some examples.

Example 1: $f(n) = 3n + 2 \in \Omega(n)$

as $3n + 2 \geq 3n$ for all $n \geq 1$

so $c = 3$ and $n_0 = 1$ Hence

$3n + 2 \in \Omega(n)$

Example 2: $f(n) = 3n + 2 \notin \Omega(n^2)$.

as $3n + 2$ on no circumstances is grater than $3n^2$ for any positive value of n;

Hence $3n + 2 \notin \Omega(n^2)$.

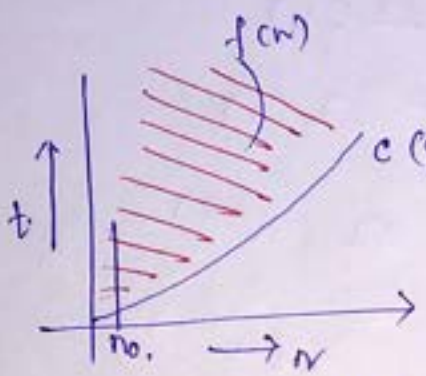
3. Ω notation (Omega)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$.

→ It determine lower bound time complexity (i.e best case time complexity)

→ If $f(n) = \Omega(g(n))$ then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq c$$



→ It implies that g(n) is an asymptotic lower bound for f(n).

Lets solve some examples.

Example 1: $f(n) = 3n + 2 \in \Omega(n)$

as $3n + 2 \geq 3n$ for all $n \geq 1$

so $c = 3$ and $n_0 = 1$ Hence

$3n + 2 \in \Omega(n)$

Example 2: $f(n) = 3n + 2 \notin \Omega(n^2)$.

as $3n + 2$ on no circumstances is grater than $3n^2$ for any positive value of n;

Hence $3n + 2 \notin \Omega(n^2)$.

Example-3. $f(n) = 6\,2^n + n^2 \in \Omega(2^n)$

as $6 \cdot 2^n + n^2 \geq 2^n$ for all $n \geq 1$

So $c = 1$ and $n_0 = 1$ Hence

$6 \cdot 2^n + n^2 \in \Omega(2^n)$.

It was also observed that for following examples.

$f(n) = 3n + 3 \in \Omega(1)$

$f(n) = 10n^2 + 4n + 2 \in \Omega(1)$

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(n^{100})$.

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(n^{50 \cdot 2})$

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(n^2)$

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(n)$.

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(1)$

$f(n) = 6 \cdot 2^n + n^2 \in \Omega(1)$

Example-6 of Page-(21) for Ω-notation.

Method-2.

$f(n) = (n+a)^b \in \Omega(n^b)$.

As we know that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$

$\therefore \lim_{n \to \infty} \frac{(n+a)^b}{n^b} = \lim_{n \to \infty} \frac{n^b \left(1 + \frac{a}{n}\right)^b}{n^b}$ $\left[\because \frac{a}{n} = 0\right]$

$\Rightarrow c$

Hence $(n+a)^b \in \Omega(n^b)$

o - notation.

$o(g(n)) = \{ f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that

$0 \leq f(n) < c\,g(n)$ for all $n \geq n_0 \}$.

- gt determines upperbound that is asymptotically tight.

→ If $f(n) = o(g(n))$ then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Example1. $f(n) = 2n$, $g(n) = n^2$

Prove that $f(n) \in o(g(n))$ or $o(n^2)$

Proof: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$

$\lim_{n \to \infty} \frac{2n}{n^2} = 0.$

$\lim_{n \to \infty} \frac{2}{2n} = 0$

∴ True. ∴ $f(n) \in o(n^2)$.

Example 2

$f(n) = 2n^2$ $g(n) = n^2$

Prove that $f(n) \in o(g(n))$ or $o(n^2)$.

$\lim_{n \to \infty} \frac{2n^2}{n^2}$

$\lim_{n \to \infty} \frac{2}{1} = 0$ ∴ false Hence $f(n) \notin o(n^2)$ is false

ω - notation.

$\omega(g(n)) = \{ f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0 \}$.

→ It determines lowerbound that is asymptotically tight.

→ If $f(n) = \omega(g(n))$ then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Example-1

$f(n) = 2n^2 + 16$, $g(n) = n^2$

Prove that. $f(n) \in \omega(g(n))$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \quad \lim_{n \to \infty} \frac{2n^2 + 16}{n^2} = \lim_{n \to \infty} 2 + \frac{16}{n^2}$$

$$= \lim_{n \to \infty} 2 \neq \infty .$$

∴ False so $f(n) \in \omega(g(n))$ or $\omega(n^2)$.

Example-2

$f(n) = n^2$, $g(n) = \log n$,

Prove that $f(n) \in \omega(g(n))$ or $\omega(\log n)$

Proof: $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^2}{\log n} = \lim_{n \to \infty} \frac{2n}{1/n} = \infty$$

∴ True so $f(n) \in \omega(\log n)$

Some examples of o-notation. (little oh).

$n^{1.9999} = o(n^2)$

$n^2/\lg n = o(n^2)$

$n^2 \neq o(n^2)$ [$\because 2 < 2$ is not True)

$n^2/1000 \neq o(n^2)$.

Some examples of ω-notation. (little omega)

$n^{2.0001} = \omega(n^2)$

$n^2 \lg n = \omega(n^2)$

$n^2 \neq \omega(n^2)$

Comparisons of functions.

Relational properties:

Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

Same for O, Ω, o and ω.

Reflexivity:

$f(n) = \Theta(f(n))$

Same for O and Ω

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Comparisions:

$f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$

$f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$

Trichotomy:

If a and b are two real numbers then one of the following conditions must be true.

$$a > b, \quad a < b, \quad a = b.$$

Although any two real numbers can be compared, not all functions are asymptotically comparable.

i.e

for two functions $f(n)$ and $g(n)$ it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$. For example.

Let $f(n) = n$ and $g(n) = n^{1+\sin n}$

As we know that $0 \leq 1 + \sin n \leq 2$ (i.e $1+\sin n$ oscillate between 0 and 2). Hence not comparable.

[note: Standard notations and common functions read from book Page no. 53 to 59] Introduction to Algorithms (3rd Edn) by cormen.

These functions are:

1. Monotonicity
2. Floor and ceiling
3. Modular Arithmatic
4. Polynomials
5. Exponentials.
6. Logarithms.
7. Factorials.
8. Functional iteration
9. Iterated Logarithemic functions.

Recurrences.

A recurrence is a function is defined in terms of

* one or more base cases.
* itself with smaller arguments.

Examples:

→ $$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+1 & \text{if } n>1 \end{cases}$$

Solution: $T(n) = n$.

→ $$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2)+n & \text{if } n>1. \end{cases}$$

Solution: $T(n) = n \lg n + n$

→ $$T(n) = \begin{cases} 0 & \text{if } n=2 \\ T(\sqrt{n})+1 & \text{if } n>2 \end{cases}$$

Solution: $T(n) = \lg \lg n$.

→ $$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/3)+T(2n/3)+n & \text{if } n>1 \end{cases}$$

Solution: $T(n) = O(n \lg n)$

These recurrence functions are solved by the following methods. They are

1. Iteration method.
2. Recursion Tree Method
3. Master Method.
4. Substitution Method.

1. Iteration Method.

In the iteration method we iteratively "unfold" the recurrence until we get or see a the pattern.

for example:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n) = T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

$T(n) = T(n-1) + 1$ ⟶ for order 1

$= [T(n-2) + 1] + 1$ ⟶ for order 2

i.e if $T(n) = T(n-1) + 1$.

then $T(n-1) = T(n-2) + 1$ by using the same iterative formula. Hence.

$= [T(n-3) + 1] + 1 + 1$ ⟶ for order 3

$= [T(n-4) + 1] + 1 + 1 + 1$ ⟶ for order 4

$\vdots$

$= [T(n-k) + 1] + 1 + 1 + \cdots + 1$ ⟶ for order k

If we take $T(n-k) = T(1) = 1$

then $n - k = 1$.

$\Rightarrow k = 1 - n$.

Put the value of k in above equation.

$\Rightarrow T(n-k) + 1 + \cdots + 1 + 1$

$\Rightarrow T(1) + \underbrace{1 + 1 + 1 \cdots + 1}_{k\text{th order}}$.

$\Rightarrow$

$\Rightarrow T(1) + K$ $\quad \because T(1) = 1$, $K = n-1$

$\Rightarrow 1 + n - 1$

$\Rightarrow n$

$\therefore T(n) = n.$

Example 2

Solve the following recurrance by using iterative method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \text{or } T(n-1) + n & \text{if } n > 1. \end{cases}$$

$\Rightarrow T(n) = T(n-1) + n.$ — order-1

$= [T(n-2) + (n-1)] + n.$ — order-2

$\Rightarrow [T(n-3) + (n-2)] + (n-1) + n.$ — order-3

$\Rightarrow [T(n-4) + (n-3)] + (n-2) + (n-1) + n.$ — order-4

$\Rightarrow [T(n-k) + (n-k-1)) + \cdots + (n-2) + (n-1) + n$ — order k

$n-k+1$

As the assumption is

$T(n-k) = T(1) = 1$

$\because n-k+1$

$\therefore 1 + 1 = 2$

So $n - k = 1$

$k = 1 - n$

$\Rightarrow T(1) + (1+1) + \cdots + (n-2) + (n-1) + n$

$\Rightarrow 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$

$= \frac{n(n+1)}{2}$ [∵ Sum of Arithmetic Progression (AP)]

$= \frac{n^2+n}{2}$

$= \frac{n^2}{2} + \frac{n}{2}$

Hence $T(n) = n^2$

Example 3 : Solve the following recurrance by using iterative method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n^2 & \text{if } n > 1 \end{cases}$$

$\Rightarrow T(n) = T(n-1) + n^2$

$= T(n-2) + (n-1)^2 + n^2$

$= T(n-3) + (n-2)^2 + (n-1)^2 + n^2$

$\vdots$

$= T(n-k) + (n-k+1)^2 + \cdots + (n-2)^2 + (n-1)^2 + n^2$

Assumption $n - k = 1$

$\Rightarrow T(1) + 2^2 + \cdots + (n-2)^2 + (n-1)^2 + n^2$

$\Rightarrow 1^2 + 2^2 + \cdots + (n-2)^2 + (n-1)^2 + n^2$

$\Rightarrow \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

$= \frac{(n^2+n)(2n+1)}{6}$

$\therefore T(n) = \Theta(n^3)$

Example-4 Solve the following recurrence function by using iterative method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n\,T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

$T(n) = n\,T(n-1) + 1$

$= n[(n-1)\,T(n-2) + 1] + 1$

$= n(n-1)\,T(n-2) + n + 1$

$= n(n-1)[(n-2)\,T(n-3) + 1] + n + 1$

$= n(n-1)[(n-2)\,T(n-3) + n(n-1) + n + 1$

$\vdots$

for k^{th} order

$= n(n-1)(n-2)\cdots(n-k+1)\,T(n-k) +$

$+ n(n-1)(n-2)\cdots(n-k+2)$

$+ \cdots + n(n-1)(n-2) + n(n-1) + n + 1$

$\therefore T(n-k) = T(1) = 1$

$n - k = 1$

$k = n-1$

$\Rightarrow n(n-1)(n-2)\cdots(2)\cdot(1) + n(n-1)(n-2)\cdots 3 + \cdots$

$n(n-1)(n-2) + n(n-1) + n + 1$

$\Rightarrow n! + (n-2)! + (n-1)! + \cdots + 1$

$\therefore T(n) = n!$

Example - a. Solve for using Iterative method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n - 1 & \text{if } n > 1 \end{cases}$$

$$T(n) = T(n-1) + n - 1$$

$$= T(n-2) + (n-2) + (n-1)$$

$$= T(n-3) + (n-3) + (n-2) + (n-1)$$

for k^{th} order $= T(n-k) + (n-k) + (n-k+1) + \cdots (n-3) + (n-2) + (n-1)$

$$= 1 + 1 + 2 + 3 + \cdots + (n-2) + (n-1).$$

$$= 1 + \left(\frac{n(n+1)}{2} - n\right)$$

$$= 1 + \left(\frac{n^2 + n - 2n}{2}\right)$$

$$= 1 + \left(\frac{n^2 - n}{2}\right) =$$

$$= \frac{n^2}{2} - \frac{n}{2} + 1.$$

$\therefore T(n) = n^2.$

Example - b $T(n) = \begin{cases} 1 & \text{for } n = 1 \\ T(n-1) + \log n. \end{cases}$

$$T(n) = T(n-1) + \log n$$

$$= T(n-2) + \log(n-1) + \log n$$

$= T(n-3) + \log(n-2) + \log(n-1) + \log(n)$

for k^{th} order

$= T(n-k) + \log(n-k+1) + \cdots + \log(n-2) + \log(n-1) + \log n$

As per assumption $n-k=1$ Hence

$= T(1) + \log(2) + \log 3 + \cdots + \log n.$

$= 1 + \log 2 + \log 3 + \cdots + \log n.$

$= 1 + \log(2 \cdot 3 \cdot 4 \cdots \cdots n)$

$= 1 + \log(n!)$

$\Rightarrow \log(n!) + 1$

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

Example - 7

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + n^2 & \text{if } n > 1 \end{cases}$$

$T(n) = 8T(n/2) + n^2$

$= 64T(n/4) + 8(n/2)^2 + n^2$

$= 64 \cdot 8T(n/8) + 64(n/4)^2 + 8(n/2)^2 + n^2$

$= 512\, T(n/8) + 64\left(\frac{n}{4}\right)^2 + 8(n/2)^2 + n^2$

$= 8^3 T(n/8) + 8^2(n/4)^2 + 8^1(n/2)^2 + n^2$

$= 8^3 T(n/2^3) + 8^2(n/2^2)^2 + 8^1(n/2^1)^2 + n^2$

for k^{th} order

$$\Rightarrow 8^k T\left(\frac{n}{2^k}\right) + 8^{k-1} * \left(\frac{n}{2^{k-1}}\right)^2 + 8^{k-2} * \left(\frac{n}{2^{k-2}}\right)^2$$

$$+ \cdots + 2^3 \left(\frac{n}{2^2}\right)^2 + 2^2 \left(\frac{n}{2}\right)^2 + n^2$$

$$\Rightarrow 8^k T\left(\frac{n}{2^k}\right) + n^2 \left[\frac{8^{k-1}}{2^{k-1}} + \frac{8^{k-2}}{2^{k-2}} + \cdots + 1\right]$$

$$= 8^k T\left(\frac{n}{2^k}\right) + 8^{k-1}\left(\frac{n^2}{4^{k-1}}\right) + 8^{k-2}\left(\frac{n^2}{4^{k-2}}\right)$$

$$+ \cdots + 2^3\left(\frac{n^2}{4^2}\right) + 2^2\left(\frac{n}{4}\right) + n^2$$

$$= 8^k T\left(\frac{n}{2^k}\right) + n^2\left[\frac{8^{k-1}}{4^{k-1}} + \frac{8^{k-2}}{4^{k-2}} + \cdots + 1\right]$$

$$= 8^k T\left(\frac{n}{2^k}\right) + n^2\left[2^{k-1} + 2^{k-2} + \cdots + 1\right]$$

Assumption: $T\left(\frac{n}{2^k}\right) = T(1)$

$\because \frac{n}{2^k} = 1$

$n = 2^k$

$\log n = \log 2^k$

$k \log 2 = \log n$

$k = \log_2 n$

$$= 8^{\log n} T(1) + \sum_{i=0}^{\log n - 1} 2^i n^2.$$

$$\Rightarrow 8^{\log n} + n^2 \sum_{i=0}^{\log n - 1} 2^i$$

$$\Rightarrow 8^{\log n} + n^2 \{ \underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^{\log n - 1}}_{\text{is a geometric or exponential series. ie.}} \}$$

$$\Rightarrow$$

$$\sum_{i=0}^{n} x^k = 1 + x + x^2 + \cdots + x^k = \frac{x^{n+1} - 1}{x - 1}$$

Hence.

$$\Rightarrow 8^{\log n} + n^2 \left(\frac{2^{\log n - 1 + 1} - 1}{2 - 1} \right)$$

$$\Rightarrow n^2 \left(2^{\log n} - 1 \right) + 8^{\log n}.$$

$$\Rightarrow n^2 \cdot 2^{\log n} - n^2 + 8^{\log n}.$$

$$\therefore T(n) = O(n^3).$$

Example - 8

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/2) + n^2 & \text{if } n > 1 \end{cases}$$

(Strassen Algo.)

note: $\log_2 7 = 2.81$

$T(n) = 7\,T(n/2) + n^2$

$= 7\left[7T(n/4) + (n/2)^2\right] + n^2$

$= 49\,T(n/4) + 7\,\frac{n^2}{4} + n^2$

$= 49\left[7\,T(n/8) + \left(\frac{n^2}{8}\right)^2 + 7\,\frac{n^2}{4} + n^2\right.$

$= 486\,T\left(\frac{n}{8}\right) + 49\frac{n^2}{11} + 7\,\frac{n^2}{4} + n^2$

$= \vdots\; 7^3\,T\left(\frac{n}{2^3}\right) + 7^2\left\{\frac{n^2}{4^2}\right\} + 7^1\left(\frac{n^2}{4^1}\right) + n^2$

$= \vdots$

for k^{th} order $= 7^k\,T\left(\frac{n}{2^k}\right) + 7^{k-1}\,\frac{n^2}{4^{k-1}} + 7^{k-2}\,\frac{n^2}{4^{k-2}}$

$+ \cdots + 7^2\,\frac{n^2}{4^2} + 7\,\frac{n^2}{4} + n^2$

Assume that $n/2^k = 1$

$\Rightarrow n = 2^k$

$\log n = \log 2^k$

$\therefore k = \log_2 n$

$\Rightarrow 7^{\log n}\,T(1) + n^2\left[\frac{7^{\log n-1}}{4^{\log n-1}} + \frac{7^{\log n-2}}{4^{\log n-2}} + \cdots + 1\right]$

$= 7^{\log n} + n^2 \sum_{i=0}^{\log n-1} \left(\frac{7}{4}\right)^i$

$= 7^{\log n} + n^2 \sum_{i=0}^{\log n-1} \frac{7^i}{4^i}$

$$\Rightarrow n^2 \cdot \sum_{i=0}^{\log n - 1} \left(\frac{7}{2^2}\right)^i + 7^{\log n}.$$

$$\Rightarrow n^2 \cdot \Theta\left(\frac{7}{2^2}\right)^{\log n - 1} + 7^{\log n}$$

$$\Rightarrow n^2 \cdot \Theta \frac{7^{\log n}}{2^{2\log n - 1}} \left(\frac{7}{2^2}\right)^{\log n} \cdot \left(\frac{7}{2^2}\right)^{-1} + 7^{\log n}$$

$$\Rightarrow n^2 \cdot \Theta \left(\frac{7^{\log n}}{(2^2)^{\log n}} \right) + 7^{\log n}$$

$$\Rightarrow \quad \because a^{\log_b n} = n^{\log_b a}$$

$$\Rightarrow n^2 \cdot \Theta \left(\frac{7^{\log n}}{(2^2)^{\log n}} \right) + 7^{\log n}$$

$$\Rightarrow n^2 \cdot \Theta \left(\frac{7^{\log n}}{n^{2\log 2}} \right) + 7^{\log n}$$

$$\Rightarrow \Theta(7^{\log n}) \qquad [\because a^{\log_c b} = b^{\log_c a}]$$

$$\Rightarrow \Theta(n^{\log 7}) \qquad [\because \log_2 7 = 2.81]$$

$$\Rightarrow \Theta(n^{2.81}).$$

Note:

$$(2^2)^{\log n} = 2^{2\log n} = n^{2\log 2}$$
$$= n^{2 \cdot 1}$$
$$= n^2$$

Example - 9 $T(n) = \begin{cases} 1 & \text{if } n=0 \\ 2T(n-1)+1 & \text{if } n>1 \end{cases}$

$T(n) = 2T(n-1) + 1$

$= 2[2T(n-2)+1] + 1$

$= 4T(n-2) + 2 + 1$

$= 4[2T(n-3)+1] + 2 + 1$

$= 8T(n-3) + 4 + 2 + 1$

$= 8[2T(n-4)+1] + 4 + 2 + 1$

$= 16T(n-4) + 8 + 4 + 2 + 1$

$= 2^4 T(n-4) + 2^3 + 2^2 + 2^1 + 2^0$

.
.
.

for k^{th} order

$= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \cdots + 2^3 + 2^2 + 2^1 + 2^0$

Base condition $T(0) = 1$

So $n-k = 0$.

So $n = k$

$= 2^n T(0) + 2^{n-1} + 2^{n-2} + \cdots + 2^3 + 2^2 + 2^1 + 2^0$

(It is a G.P. series so sum is equal to

$= \frac{2^{n+1} - 1}{2-1} = 2^{n+1} - 1 = 2^n \cdot 2^1 - 1$

Hence $T(n) = O(2^{n+1} - 1)$

or we can say it is exponential i.e $O(2^n)$

Example - 10

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T\left(\frac{7n}{10}\right) + n & \text{if } n>1 \end{cases}$$

$$T(n) = T\left(\frac{7n}{10}\right) + n$$

$$= T\left(\frac{7}{10} \cdot \frac{7n}{10}\right) + \frac{7n}{10} + n$$

$$= T\left(\frac{7^2}{10^2} n\right) + \frac{7}{10} n + n$$

$$= T\left(\frac{7^3}{10^3} n\right) + \frac{7^2}{10^2} n + \frac{7}{10} n + n$$

for k^{th} order $= T\left(\frac{7^k}{10^k} n\right) + \frac{7^{k-1}}{10^{k-1}} n + \cdots + \frac{7^2}{10^2} n + \frac{7}{10} n + n$

Base case $T(1) = 1$

Hence $\frac{7^k}{10^k} n = 1$

$$n 7^k = 10^k$$

$$n = \left(\frac{10}{7}\right)^k$$

$$\log n = \log \left(\frac{10}{7}\right)^k$$

$$\log n = k \log \frac{10}{7}$$

$$k = \frac{\log n}{\log 10/7} = \log_{10/7} n$$

Hence the equation =

$$T(n) = T(1) + n\left[\left(\frac{7}{10}\right)^{\log_{10/7} n - 1} + \cdots + \left(\frac{7}{10}\right)^2 + \frac{7}{10} + 1\right]$$

$$< 1 + n\left[1 + \frac{7}{10} + \left(\frac{7}{10}\right)^2 + \left(\frac{7}{10}\right)^3 + \cdots + \left(\frac{7}{10}\right)^{\log_{10/7} n - 1}\right]$$

$$< 1 + n\left[1 + \frac{7}{10} + \left(\frac{7}{10}\right)^2 + \cdots + \infty\right] \quad \because \left(\frac{7}{10}\right)^{\log_{10/7} n - 1} < 1$$

$$< 1 + n\left[\frac{1}{1 - 7/10}\right] \quad \because \text{Series is in G.P}$$

$< \frac{10n}{3}$.

Hence $T(n) = O(n)$.

Recursion Tree

It is a different way to look at the iteration method. In recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocation.

In this method:-

First, ~~cost~~ a set of pre-level costs are obtained by sum the cost of each level of the tree.

Second, to determine the total cost of all levels of recursion, we sum all the pre level cost.

This method is best used to generate good guess. which we can be used by substitution method (discussed in next topic) for ~~proof~~ verification, for generating good guess and not to prove anything by using recurrence tree, we can tolerate a certain amount of "sloppiness". in our analysis.

For example, we can ignore floors and ceilings when solving the recurrances. because they usually do not affect the final guess.

Example:

Recurrence: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

we drop the floor and write a recursion tree as $T(n) = 3T(n/4) + cn^2$.

Construction of recursion Tree.

$T(n)$ ~~D(n)~~

fig-(a) ~~fig-D(n)~~.

figure (a) shows $T(n)$, which progressively expands in (b)-(d) to form recursion tree.

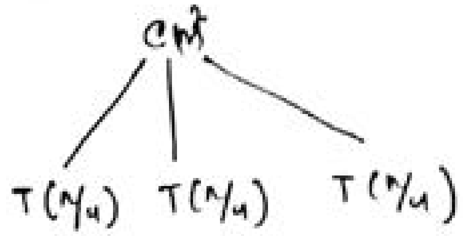


fig-(b)

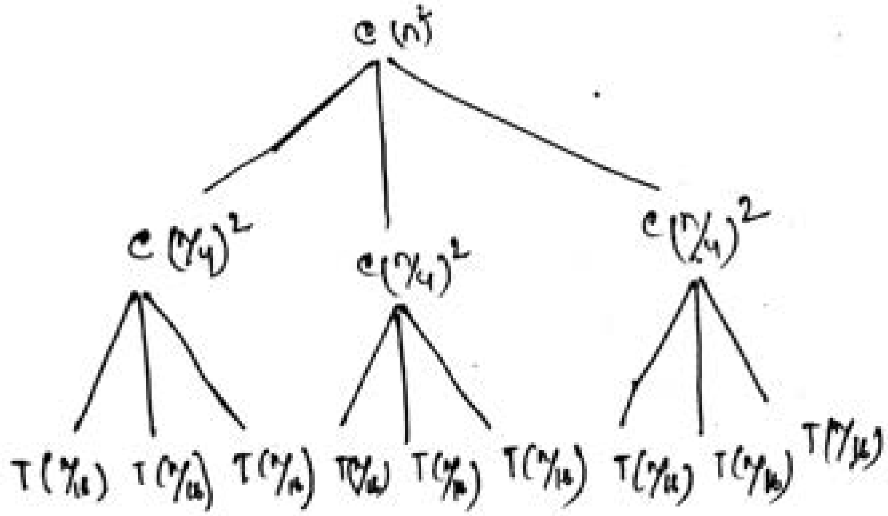
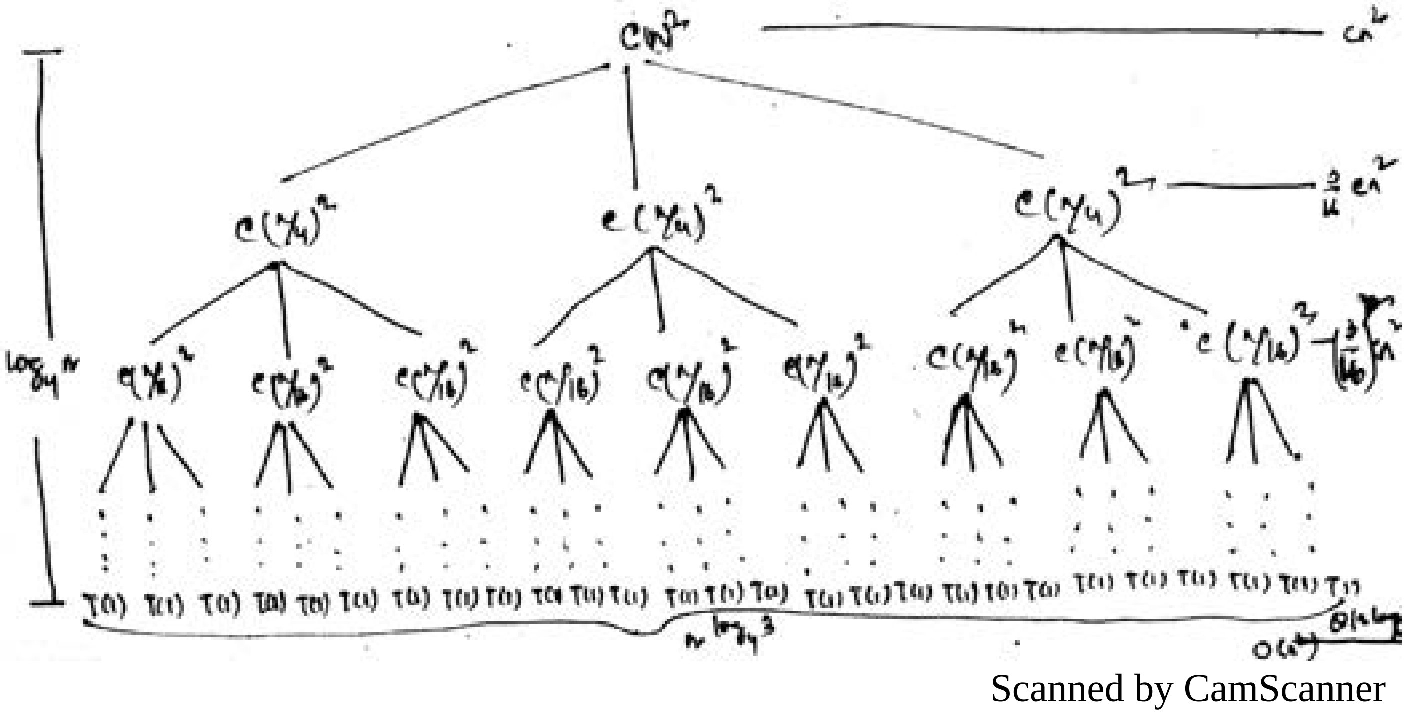


fig-(c)



→ The top node has cost cn^2, because the first call to the function does cn^2 unit of work, aside from the work done inside the recursive subcalls.

→ The nodes on the second layer all have cost $c(n/4)^2$, because the functions are now being called on problems of size $(n/4)$, and the functions are doing $c(n/4)^2$ units of work, aside from the work done inside their recursive subcalls, etc.

→ The bottom layer (i.e base case) is special because each of them contribute $T(1)$ to the cost.

Analysis:

→ First, we find the height of the recursion tree.

→ Observe that a node at depth i reflects a subproblem of size $n/4^i$.
i.e The subproblem size hits $n=1$, when $n/4^i=1$

or $n/4^i = 1$

$\Rightarrow n = 4^i$

$\Rightarrow \boxed{i = \log_4 n}$ ← height of tree

So the tree has $\log_4 n + 1$ levels.

→ Secondly, we determine the cost of each level of the tree.

→ The number of nodes at depth i is 3^i.
So each node at depth i (i.e $i = 0, 1, 2, \cdots, \log_4 n - 1$) has cost $c(n/4^i)^2$.

→ Hence the total cost of level is

total cost at level $i = 3^i c(n/4^i)^2$

$= 3^i . c . \frac{n^2}{16^i}$

$= \left(\frac{3}{16}\right)^i c n^2$.

However, the bottom level is special. Each of the bottom nodes contributes cost $= T(1)$. Hence the cost of bottom level is $3^i = 3^{\log_4 n}$

$= n^{\log_4 3}$

Here $i \leftarrow$ height of the tree i.e $\log_4 n$

So the total cost of entire tree is

$$T(n) = cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \cdots$$

$$\cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

The left term is just a sum of Geometric series. So $T(n)$ evaluates to.

$$= \left(\frac{\left(\frac{3}{16}\right)^{\log_4 n - 1 + 1} - 1}{\frac{3}{16} - 1} \right) cn^2 + \Theta(n^{\log_4 3})$$

The above equation looks very complicated, so we can bound it (from above) by the sum of the infinite (∞) series of G.P.

i.e $\sum_{i=0}^{\infty} x^k = \frac{1}{1-x}$

Hence

$$\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \left(\frac{1}{1-\frac{3}{16}}\right) cn^2 + \Theta(n^{\log_4 3})$$

Since $\Theta(n^{\log_4 3})$ are also in $O(n^2)$, the whole expression is now converted to $O(n^2)$.

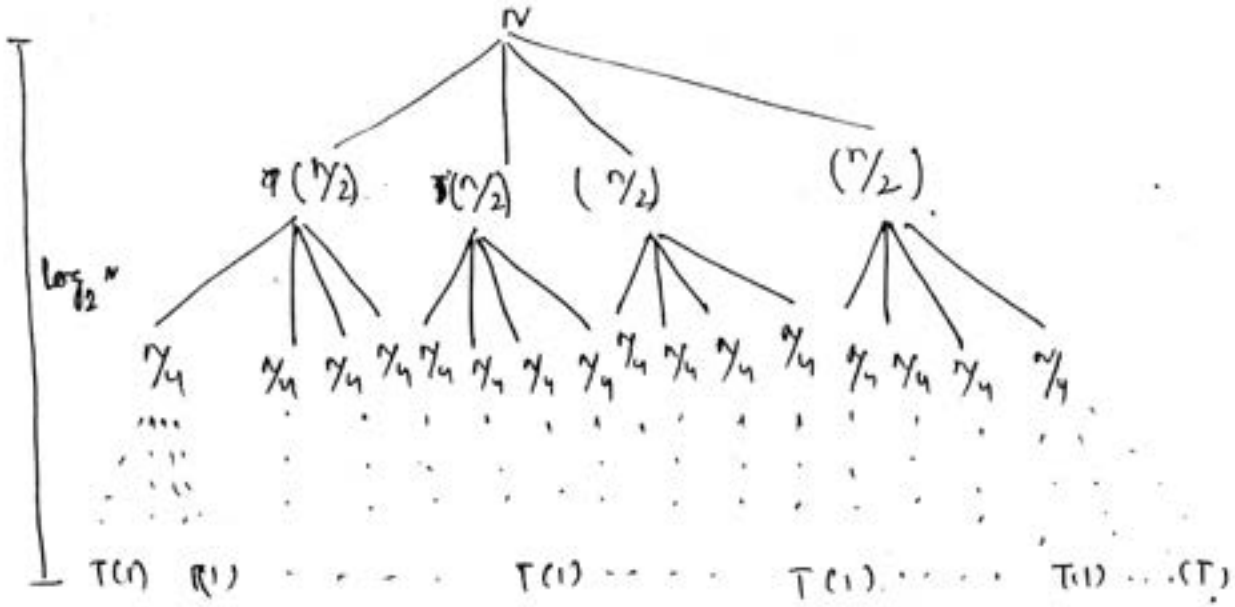
Therefore we guess that $T(n) = O(n^2)$.

Example 2

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4T(n/2) + n & \text{if } n > 1 \end{cases}$$

Solve the recurrence function by using recurrence tree method.

Let us draw the recurrence tree.



Analysis.

→ First, we find the height of the recursion tree.

Observe that a node at depth i reflects a subproblem of size $\frac{n}{(2)^i}$

i.e the subproblem size hits $n = 1$ when $\frac{n}{2^i} = 1$

$$\Rightarrow \frac{n}{2^i} = 1$$

$$n = 2^i$$

$$\boxed{\log_2 n = i} \leftarrow \text{height of the tree.}$$

So the tree has $\log_2 n + 1$ level.

→ Secondly, we determine the cost of each level of the tree.

→ The number of node at depth i is 4^i

So each node at depth i (i.e $i = 0, 1, \cdots, \log n - 1$) has cost $\left(\frac{n}{2^i}\right)$

→ Hence the total cost at level $i = 4^i \cdot \frac{n}{2^i} = \left(\frac{4}{2}\right)^i n$

$= 2^i n$.

→ → However the bottom level is special. Because each bottom nodes contribute cost = T(1). Hence the bottom level cost is $4^i = 4^{\log_2 n} = n^{\log_2 4}$

$= n^2$

So the total cost of entire tree is,

$$T(n) = \sum_{i=0}^{\log_2 n - 1} \left(\frac{4}{2}\right)^i n + n^2$$

$$= n \sum_{i=0}^{\log_2 n - 1} \left(\frac{4}{2}\right)^i + n^2$$

$$= n\left[\frac{4/2^{\log_2 n - 1 + 1} - 1}{4/2 - 1}\right] + n^2$$

$$= n\left[\frac{\frac{4}{2}^{\log_2 n} - 1}{1}\right] + n^2$$

$$= n\left[\frac{2^{\log_2 n} - 1}{1}\right] + n^2$$

$$= n\left[n^{\log_2 2} - 1\right] + n^2$$

$$= n\left[n - 1\right] + n^2$$

$$= n^2 - n + n^2$$

$$= 2n^2 - n$$

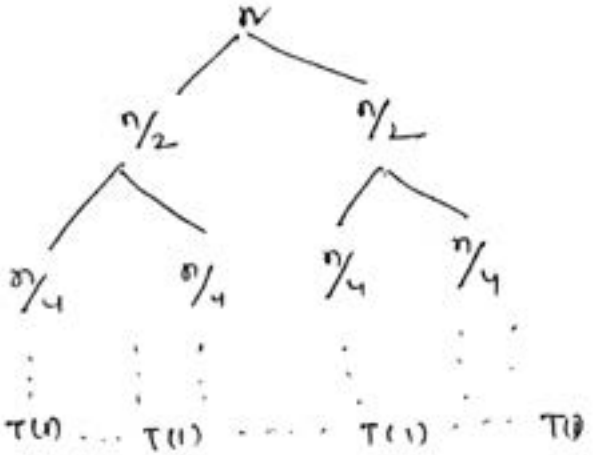
Therefore we guess that $T(n) = O(n^2)$.

Example: 3 Solve the following recurrence by using Recurrence Tree Method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n) = 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

↳ Recurrence equation of Merge Sort.

Let draw the recursion Tree



Analysis:

→ First we calculate the height of the recursion tree.

Observe that a node at depth i reflects a subproblem of size $n/2^i$

i.e the subproblem size hits $n=1$ when $\frac{n}{2^i} = 1$

$$\Rightarrow \frac{n}{2^i} = 1$$
$$n = 2^i$$

$$\boxed{\log_2 n = i}$$ ← height of the tree.

→ Secondly we determine the cost of each level of the tree.

→ The number of node at depth i is 2^i

So each node at depth i (i.e $i = 0, 1, \cdots, \log_2 n-1$) has cost $\left(\frac{n}{2^i}\right)$

→ Hence the total cost at level $i = 2^i \cdot \frac{n}{2^i}$

$$= \left(\frac{2}{2}\right)^i n$$
$$= (1)^i n$$
$$= [illegible]$$

→ However the bottom level is [illegible] special, Because each bottom nodes contribute cost $T(1)$ Hence the bottom level cost is [illegible]

$$2^i = 2^{\log_2 n} = n^{\log_2 2} = n.$$

So the total cost of entire tree is

$$T(n) = \sum_{i=0}^{\log n - 1} n + n = [illegible]$$
$$= [illegible]$$

$$= n \sum_{i=0}^{\log_2 n - 1} 1 + n$$

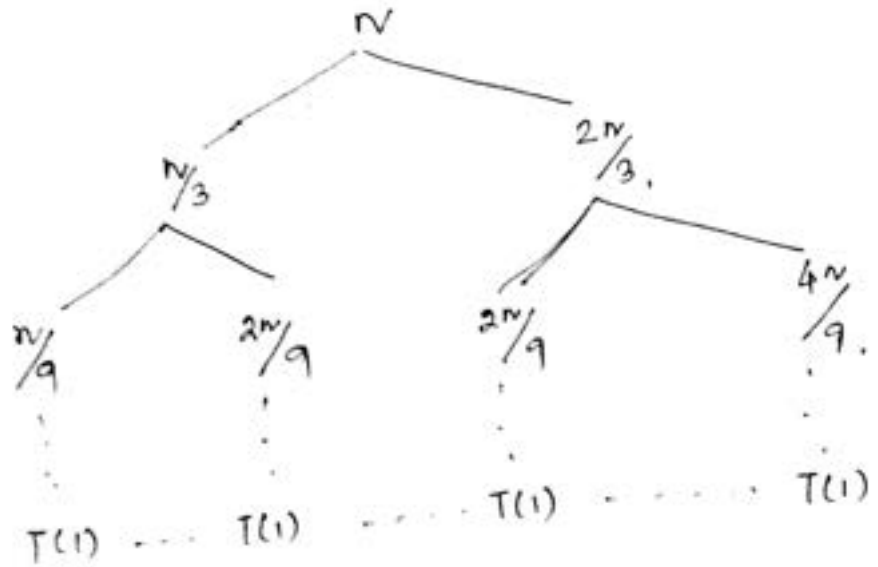
$$= n (\log_2 n) + n$$

Hence for we guess that $T(n) = O(n^0 \log_2 n)$

Example 4 Solve the following recurrence equation by Recursion Tree method.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(\frac{n}{3}) + T(\frac{2n}{3}) + n & \text{if } n > 1 \end{cases}$$

Let us draw the recursion Tree.



Analysis.

First calculate the height of the recursion tree. In this the tree is not equally divided. So we got two height. one → for left subtree and one for right subtree.

Height for the left subtree:

Observed that a node at depth i reflects a subproblem of size $n/3^i$

i.e the subproblem size hits $n = 1$, when $\frac{n}{3^i} = 1$

$\Rightarrow n/3^i = 1$

$\Rightarrow n = 3^i$

$\Rightarrow \log n = \log 3^i$

$\therefore \boxed{i = \log_3 n}$ ← height of the left subtree.

So the tree has $\log_3 n + 1$ levels on left side.

Similarly calculate the height for right side:

For right hand side of the tree we observe that a node at depth i reflects a subproblem of size $\frac{n}{(3/2)^i}$.

i.e the subproblem size hits $n = 1$ when $\frac{n}{(3/2)^i} = 1$

$\Rightarrow \frac{n}{(3/2)^i} = 1$

$\Rightarrow n = (3/2)^i$

$\Rightarrow \log n = \log (3/2)^i$

$\Rightarrow \boxed{i = \log_{3/2} n}$ ← height of right subtree

So the tree has $\log_{3/2} n + 1$ level on right side.

From both the side height is $\log_{3/2} n$. Hence we calculate the cost of each level of the tree for worst case.

→ The number of node at depth i is $= \frac{n}{3^i} + \frac{n}{(3/2)^i}$

→ Hence the total cost at level i =

$$= 1^i \cdot \left(\frac{n}{3^i} + \frac{n}{(3/2)^i} \right)$$

$$= 1^i \cdot n \cdot \left[\left(\frac{1}{3} \right)^i + \left(\frac{2}{3} \right)^i \right]$$

$$= n.$$

However the bottom level is special. Because each node at bottom level contribute the cost of T(1)

Hence the bottom level cost is

$$1^i = 1^{\log_{3/2} n} = 1$$

So the total cost of entire tree is

$$T(n) = \sum_{i=0}^{\log_{3/2} n - 1} n + \Theta(n^{\log_{3/2} 1})$$

$$= n \left(\log_{3/2} n \right) + 1.$$

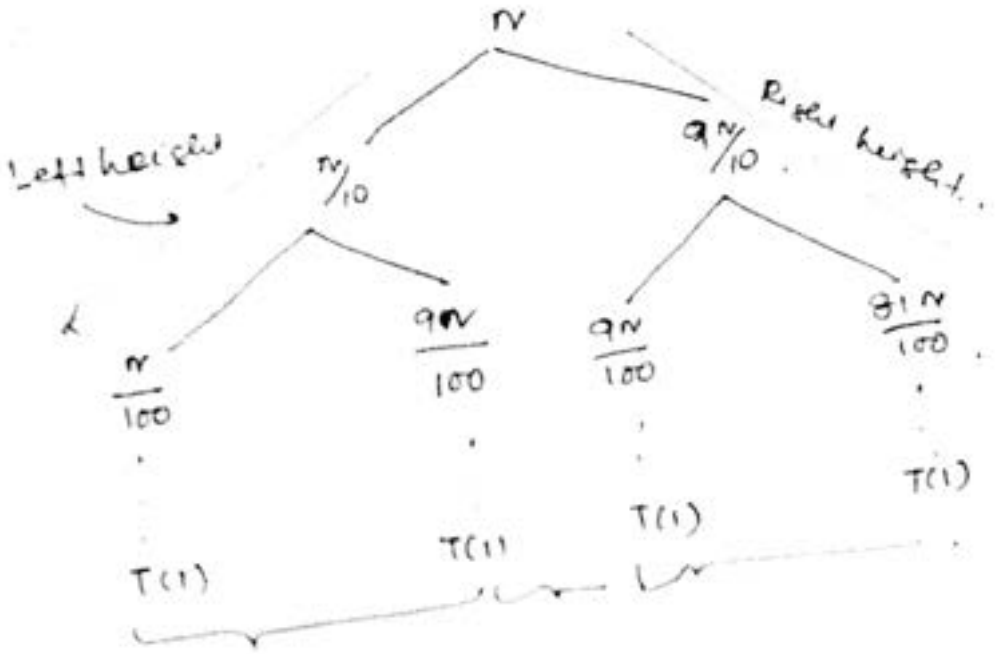
$$= n \log_{3/2} n$$

As we take the higher order, we guess that the height is $T(n) = O(n \log_{3/2} n)$.

Example-5 Solve the following recurrence function by using recursion Tree.

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ T(n/10) + T\left(\frac{9n}{10}\right) + n \end{cases}$$

Let us first draw the recursion Tree and then calculate the height and total cost.



Analysis:

Height of the Left side of the tree at i^{th} level is $n/100^i$ and the subproblem size is 1

$n = 1$ when $\frac{n}{100^i} = 1$

$\Rightarrow n = 100^i$

$\Rightarrow \log n = \log 100^i$

$\Rightarrow \boxed{i = \log_{100} n}$ ← minimum height of the tree.

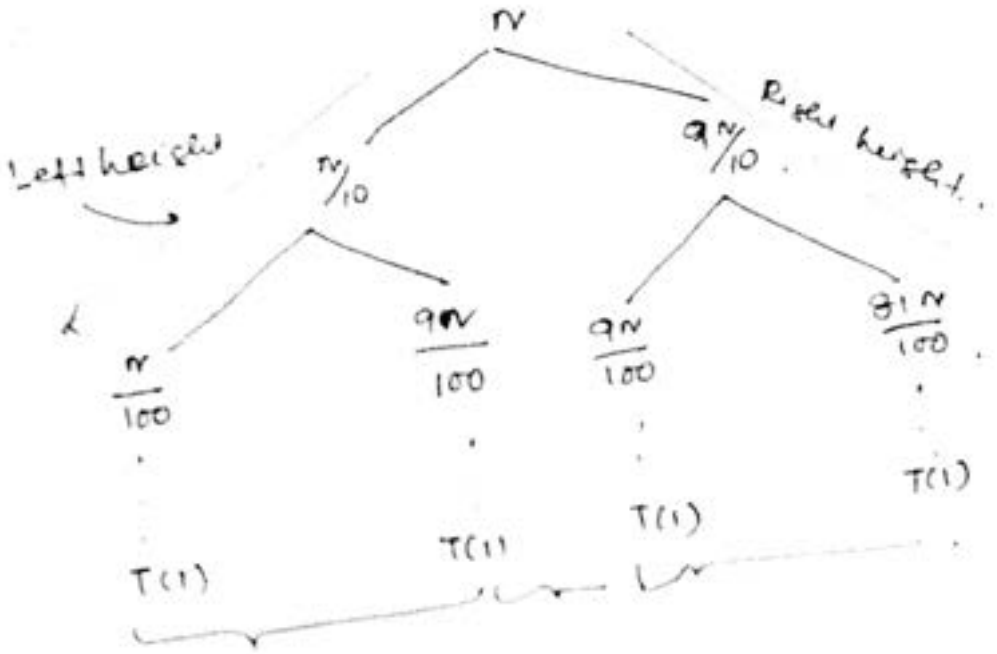
Height of the right side of the tree at i^{th} level is $\frac{9^i n}{(100/9)^i}$ and the subproblem size is 1

$n = 1$ when $\frac{n}{(100/9)^i} = 1$

$\Rightarrow n = (100/9)^i$

$\log n = i \log(100/9)$

$\boxed{i = \log_{100/9} n}$ ← Maximum height of the tree.



Analysis:

Height of the Left side of the tree at i^{th} level is $n/100^i$ and the subproblem size is 1

$n = 1$ when $\frac{n}{100^i} = 1$

$\Rightarrow n = 100^i$

$\Rightarrow \log n = \log 100^i$

$\Rightarrow \boxed{i = \log_{100} n}$ ← minimum height of the tree.

Height of the right side of the tree at i^{th} level is $\frac{9^i n}{(100/9)^i}$ and the subproblem size is 1

$n = 1$ when $\frac{n}{(100/9)^i} = 1$

$\Rightarrow n = (100/9)^i$

$\log n = i \log(100/9)$

$\boxed{i = \log_{100/9} n}$ ← Maximum height of the tree.

→ The total cost at i^{th} level is $= n\left(\frac{1}{10} + \frac{9}{10}\right)^i$

$= n.$

→ The total cost at bottom level $= 1^{\log_{10/9} n} = 1.$

So the total cost of the entire tree is

$$T(n) = \sum_{i=0}^{\log_{10/9} n - 1} n + 1$$

$$= n \sum_{i=0}^{\log_{10/9} n - 1} 1 + 1$$

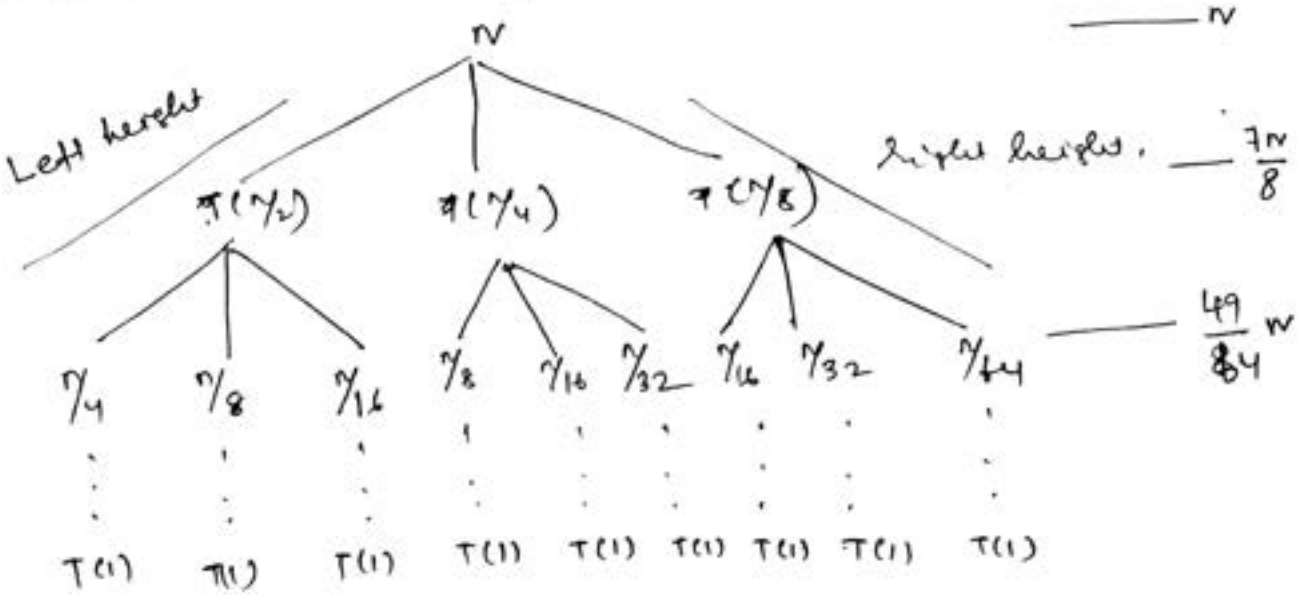
$$= n \log_{10/9} n + 1,$$

Hence we guess that height is $T(n) = O(n \log_{10/9} n)$

Example-6 Solve the following recurrence function by using recursion tree method:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + T(n/4) + T(n/8) & \text{if } n > 1 \end{cases}$$

Sol: Let us first draw the recursion tree.



Left height or i^{th} level.

$$T\left(\frac{n}{2^i}\right) = T(1)$$

$$n/2^i = 1$$

$$n = 2^i$$

$$i = \log_2 n$$

Right height or i^{th} level.

$$T\left(\frac{n}{8^i}\right) = T(1)$$

$$n/8^i = 1$$

$$n = 8^i$$

$$\boxed{i = \log_8 n.}$$

↓

Heighest height.

Hence the total cost of the tree at level i

$$= \left(\frac{n}{2^i} + \frac{n}{4^i} + \frac{n}{8^i}\right)$$

$$= n + \frac{7n}{8} + \frac{49}{64} n + \cdots + \infty.$$

$$= n\left[1 + \left(\frac{7}{8}\right)^1 + \left(\frac{7}{8}\right)^2 + \cdots + \infty\right]$$

$$= n\left[\frac{1}{1 - 7/8}\right]$$

$$= n\left[1/1/8\right] = 8n = cn.$$

So the total cost of the tree is

$$\sum_{i=0}^{\log_8 n - 1} n + T(1).$$

$$= n \sum_{i=0}^{\log_8 n - 1} + 1$$

$$= n \log_8 n + 1$$

Hence we guess that for higher order $T(n) = O(n \log_8 n)$

Master Method.

The master method depends on the following theorem:

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

[Note: Beyond this intuition, we must be aware of some technicality: i.e not only $f(n)$ must be smaller than $n^{\log_b a}$, but also it must be polynomially smaller. It means $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$.]

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n; then $T(n) = \Theta(f(n))$.

[Note: Beyond this intuition, we must be aware of some technicality: i.e not only $f(n)$ must be larger than $n^{\log_b a}$, but also it must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$.

Note:

'Polynomially larger' means that the ratio of the functions falls between two polynomials, asymptotically. Specifically $f(n)$ is polynomially greater than $g(n)$ if and only if there exist generalized polynomials (i.e fractional exponents are allowed) $p(n)$, $q(n)$ such that the following inequality holds asymptotically.

$$p(n) \leq \frac{f(n)}{g(n)} \leq q(n)$$

This is more strict than merely asymptotically larger, and gives limits on how much faster the functions can grow. For example.

n is not polynomially larger than $\frac{n}{\lg(n)}$ because the rate of the growth is wrong. To see this, notice that there's no polynomial such that

$$p(n) \leq \frac{f(n)}{g(n)}$$

$$p(n) \leq n / n/\log n$$

$$p(n) \leq \log n.$$

Another simple example of this is e^n which is not polynomially larger than n^k for any k.

Let us solve one example with Master method to solve this type of concept.

Example-1 $\quad T(n) = 4T(n/2) + \frac{n^2}{\log n}$.

So $a = 4 \quad b = 2$ and $f(n) = \frac{n^2}{\log n}$.

So Hence $n^{\log_b a} = n^{\log_2 4} = n^2$.

We might mistakenly think that case 1 should apply, since $f(n) = \frac{n^2}{\lg n}$ is asymptotically smaller than $n^{\log_b a} = n^2$.

But the problem is that it is not polynomially smaller. i.e the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n^2/\log n}{n^2} = \frac{1}{\log n}$ is asymptotically greater than n^ϵ for any positive constant ϵ. i.e.

$n^\epsilon < \frac{1}{\log n}$ for any value of $\epsilon > 0$.

↓

for this case of recurrance it is not true.

Hence. Master method is not applicable here.

Example. 2

$$T(n) = 2T(n/2) + n\log n.$$

$a = 2$, $b = 2$ $f(n) = n\log n$.

Hence $n^{\log_b a} = n^{\log_2 2} = n$.

We might mistakenly think that case 3 should apply, since $f(n) = n\log n$ is asymptotically larger than $n^{\log_b a} = n$. But the problem is that it is not polynomially larger. i.e the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n\lg n}{n}$

$= \lg n$ is asymptotically smaller than n^ϵ for any positive constant ϵ. i.e.

$\log n > n^\epsilon$ for any value of $\epsilon > 0$

↳ is not true for this case.

Hence Master method is not applicable. bcoz.

Example-3 $T(n) = 9T(n/3) + n.$

$a=9 \quad b=3 \quad$ and $f(n) = n.$

calculate $n^{\log_b^a}$

i.e $n^{\log_3^9} = n^2$

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

i.e $n^2 > n.$

So $T(n) = \Theta(n^2).$

Example-4. $T(n) = T(2n/3) + 1$

$a=1 \quad b=3/2 \quad f(n)=1$

$n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$

So case 2 is applicable. Hence the Complexity is $T(n) = \Theta(n^{\log_b^a} \cdot \lg n)$

$= \Theta(1 \cdot \lg n)$

$= \Theta(\lg N).$

Example-5 $T(n) = 3T(n/4) + n \lg n.$

$a=3 \quad b=4 \quad f(n) = n \lg n.$

$n^{\log_b^a} = n^{\log_4^3} = n^{0.793}.$

Since $f(n) = \Omega(n^{\log_4^3 + \epsilon})$ where $\epsilon = 0.2$

case 3 is applicable if we show that the regularity condition holds for $f(n)$. For sufficiently large n.

we have that $a f(n/b) \leq c f(n)$.

$3 \, n/4 \log n/4 \leq c \, n \lg n$.

for. $c = 3/4$.

So case 3 is applicable hence $T(n) = \Theta(n \lg n)$

Example-6

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

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

So $n^{\log_b a}$, $n^{\log_2 3} = n^{1.585}$

Since $f(n) = \Omega(n^{\log_2 3 + \epsilon})$ where $\epsilon \approx 0.42$ Case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $a f(n/b) \leq c f(n)$.

$3 (n/2)^2 \leq c n^2$

$3 \frac{n^2}{4} \leq c n^2$

$\Rightarrow c = 3/4$

So case 3 is applicable hence $T(n) = \Theta(n^2)$.

Example-7 $T(n) = 4T(n/2) + n^2$

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

$n^{\log_b a} = n^{\log_2 4} = n^2$

Since $f(n) = \Theta(n^{\log_b a}) = n^2$, case 2 is applicable. Hence $T(n) = \Theta(n^2 \lg n)$.

Example-8

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

$a = 1 \quad b = 2 \quad f(n) = 2^n.$

$$n^{\log_b^a}, \quad n^{\log_2^1} = n^0 = 1$$

It seems that case 3 is applicable because $f(n) = 2^n$ is asymptotically larger than $n^{\log_b^a} = n^0$. But we have to prove that it must be polynomially larger and the regularity condition.

Polynomially larger :-

$$n^0 \leq \frac{f(n)}{n^{\log_b^a}} \leq n^{\epsilon}. \quad (\text{where } \epsilon > 0).$$

polynomially larger

Hence the condition satisfy.

Regularity condition.

$$a f(n/b) \leq c f(n).$$

$$1 \quad 2^{n/2} \leq c 2^n. \quad = 2^{n/2} \leq \tfrac{1}{2} 2^n \quad [\because c = \tfrac{1}{2}]$$

$$2^{n/2} \leq c 2^n.$$

Hence for sufficiently $c < 1$

large n.

Hence case 2 is applicable. $\therefore T(n) = \Theta(2^n)$.

Example-9 $\quad T(n) = 2^n T(n/2) + n^n$

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

Here a is not a constant variable. So Master Method is not applicable.

Example-10 $T(n) = 16T(n/4) + n$.

$a = 16 \quad b = 4 \quad f(n) = n$.

$n^{\log_b a} = n^{\log_4 16} = n^2$

Since $f(n) = O(n^{\log_b a + \epsilon})$ and $f(n)$ is polynomially smaller than $n^{\log_b a}$. case 1 is applicable.

Hence $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Example-11 $T(n) = 0.5\,T(n/2) + \frac{1}{n}$.

$a = 0.5 \quad b = 2 \quad f(n) = 1/n$.

As per the master method Definition $a \geq 1$, but here $a < 1$, so master method is not applicable.

Example-12 $T(n) = 2T(n/4) + n^{0.51}$

$a = 2 \quad b = 4 \quad f(n) = n^{0.51}$

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

Since $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $f(n)$ is polynomially longer than $n^{\log_b a}$, where $\epsilon \geq 0.01$ case 3 applies if we can show that the regularity condition holds for $f(n)$. for sufficiently large n.

we have that $a\,f(n/b) \leq c\,f(n)$

$\Rightarrow 2(n/4)^{0.51} \leq c\,n^{0.51}$

$2(n/4)^{0.51} \leq \frac{1}{2} n^{0.51}$.

$c = 1/2$

which satisfy the case 3. Hence $T(n) = \Theta(n^{0.51})$

Substitution Method.

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

In this method we substitute the guessed soln. for the function and apply with the help of inductive hypothesis for smaller values, hence the name "substitution method". This method is very powerful, but we must be able to guess the form of the answer in order to apply it.

We can use substitution method to establish either upper or lower bounds on a recurrence.

For example:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Here, we guess that the solution is $T(n) = O(n \lg n)$

The substitution method requires us to prove that

$$T(n) \leq cn \lg n \text{ for an appropriate}$$

Choice of the constant $c > 0$

We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. After substituting $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$ gives a way to the above recurrence function or arguments:

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &\leq cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \end{aligned}$$

where the last step holds as long as $c \geq 1$.

In general Mathematical induction will tell us how our solution holds for the boundary conditions.

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and if you have creativity. Fortunately though, you can use some heuristics to help you become a good guesser. We can use recursion tree to generate good guesses.

Example 2: Find the asymptotic bound for the following recurrence equation,

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n.$$

The above problem may looks difficult because of the added "17" in the argument. But this additional term cannot substantially affect the solution to the recurrence. When n is large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is not that large: both cut n nearly evenly in half. So we make the guess that $T(n) = O(n \lg n)$ and can verify as correct by using the substitution method.

Changing Variable Method:

Sometimes, the recurrence functions are looks very difficult to some. But a little algebric manipulation can make a difficult or unknown recurrence to simple so that it can be solvable by any one of the above methods. Consider the following recurrence as examples.

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

which looks difficult.

we can simplify this recurrence with the help of changing variable method.

For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers. Renaming $m = \log n$ which change the recurrence to following form.

Let see how?

Let $n = 2^m$.

Hence the recurrence equation converted to,

$$T(2^m) = 2T(2^{m/2}) + \log_2 2^m$$

$\because \log_b a^n = n \log_b a$

$$= 2T(2^{m/2}) + m \log_2 2$$

$$= 2T(2^{m/2}) + m.$$

Let ~~$T(2^m)$~~ $= S(m)$

$$T(2^{m/2}) = S(m/2)$$

Hence the recurrence relation again converted to

$$S(m) = 2\,2\,S(m/2) + m.$$

now apply master method.

$$a = 2 \quad , b = 2 \qquad f(m) = m.$$

$$m^{\log_b a} = n^{\log_2 2} = m.$$

Since $f(m) = m^{\log_b a}$, then $T(m) = \Theta(n^{\log_b a} \lg m)$

$$T(n) = \Theta(m \lg m) = \Theta(\lg n \lg \lg n)$$

Example-2 $T(n) = 2T(\sqrt{n}) + 1$

Let $n = 2^m$.

$\log n = \log 2^m$.

$\log_2 n = m$

$T(2^m) = 2T(2^{m/2}) + 1$

Let $S(m) = T(2^m)$

$S(m/2) = T(2^{m/2})$

So equation replace with:

$S(m) = 2S(m/2) + 1$

now Apply Master Method.

$a = 2 \quad b = 2 \quad f(m) = 1$

$m^{\log_b a} = m^{\log_2 2} = m$

Example-3

$T(n) = 3T(n/2) + n^2 \log n$

Let $n = 2^m$

Hence $\log n = m$

$T(2^m) = 3T(2^{m-1}) + 4^m \cdot m$

divide both side by 4^m.

$\frac{T(2^m)}{4^m} = \frac{3T(2^{m-1})}{4^m} + \frac{4^m \cdot m}{4^m}$

$= \frac{3T(2^{m-1})}{4^m} + m$

$= \frac{4 \cdot 3T(2^{m-1})}{4 \cdot 4^m} + m$

$\because n = 2^m$

$n/2 = 2^m/2 = 2^{m-1}$

$n = 2^m$

$n^2 = (2^m)^2$

$= (2^2)^m$

$= 4^m$

$$\frac{T(2^m)}{4^m} = \frac{3\,T(2^{m-1})}{4\cdot(4^{m-1})} + m.$$

Let $S(m) = \frac{T(2^m)}{4^m}$.

Hence the above equation can be written as.

$$\boxed{S(m) = \frac{3}{4} S(m-1) + m.}$$

$\Downarrow$

whose complexity is $O(n^2)$.

$$\therefore \frac{T(2^m)}{4^m} = O(m^2).$$

$$T(2^m) = O(m^2 \cdot 4^m)$$

$$T(n) = O((\log n)^2 \cdot 4^{\log_2 n})$$

$$= O((\log n)^2 \cdot n^{\log_n 4})$$

$$= O((\log n)^2 \cdot n^2).$$

Example-4 $T(n) = 2T(n/2) + \frac{n}{\log n}$

Let $n = 2^m \Rightarrow m = \log n$.

$$\Rightarrow T(2^m) = 2T\left(\frac{2^m}{2}\right) + \frac{2^m}{m}$$

$$T(2^m) = 2T(2^{m-1}) + \frac{2^m}{m}.$$

divide both side by 2^m.

$$\frac{T(2^m)}{2^m} = \frac{2T(2^{m-1})}{2^m} + \frac{2^m \cdot}{2^m \cdot m}$$

$$\frac{T(2^m)}{2^m} = \frac{2T(2^{m-1})}{2^{m-1}} + \frac{1}{m}$$

Let $S(m) = \frac{T(2^m)}{2^m}$

Hence the above equation can be written as.

$$\boxed{S(m) = S(m-1) + \frac{1}{m}}$$

$$= \frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + \cdots + \frac{1}{2} + 1$$

$$= \sum_{i=1}^{m} \frac{1}{i} \quad \text{(Harmonic series)}.$$

$$= \log m + O(1)$$

$$\therefore \quad \frac{T(2^m)}{2^m} = \log m + O(1).$$

$$= O(\log m).$$

$$T(2^m) = O(\log m \cdot 2^m).$$

$$= O(\lg \lg n \cdot \log n).$$

Example-5:

$$T(n) = 2T(\sqrt{n}) + \frac{\log_2 2^n}{\log \log_2 2^n}$$

Let $n = 2^m \Rightarrow m = \log n$.

$$T(2^m) = 2T(2^{m/2}) + \frac{\log_2 2^m}{\log \log_2 2^m}$$

$\because \log_2 2^m = \log_2 2^{2^m} = \log_2 4^m = m \log_2 4 = m \log 2$
$= \log 2^m.$

$$= 2T(2^{m/2}) + \frac{m \log_2 2}{\log m \log_2 2}$$

$$T(2^m) = 2T(2^{m/2}) + \frac{m}{\log m} \quad [\because \log_2 2 = 1]$$

Let $m = 2^p \Rightarrow \log m = p.$

$$T(2^{2^p}) = 2T(2^{2^p/2}) + \frac{2^p}{p}.$$

$$T(2^{2^p}) = 2T(2^{2^{p-1}}) + \frac{2^p}{p}.$$

Divide both side by 2^p.

$$\frac{T(2^{2^p})}{2^p} = \frac{2T(2^{2^{p-1}})}{2^p} + \frac{2^p}{p \cdot 2^p}$$

$$\frac{T(2^{2^p})}{2^p} = \frac{T(2^{2^{p-1}})}{2^{p-1}} + \frac{1}{p}.$$

Let $S(p) = \frac{T(2^{2^p})}{2^p}$

$$\boxed{S(p) = S(p-1) + \frac{1}{p}.}$$

$$S(P) = S(P-2) + \frac{1}{P-1} + \frac{1}{P}$$

$$\vdots$$

$$= \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{P-2} + \frac{1}{P-1} + \frac{1}{P}$$

[$\because$ Harmonic Series]

$$= O(\log P).$$

$$\therefore \quad \frac{T(2^{2^P})}{2^P} = O(\log P)$$

$$T(2^{2^P}) = O(\log P \cdot 2^P).$$

$$T(2^m) = O(\log \log m \cdot m).$$

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

Counting Sort:

→ Counting Sort is a type of sorting, which runs in linear time.

→ It is to determine the sorted order, this sort is not based on comparision techniques.

→ This algorithm use counting technique to determine the sorted order.

→ Counting sort assumes that each of the elements in an integer in the range 1 to K or 0 to K-1, for some integer K takes only $\theta(n)$ time. (when $K = O(n)$)

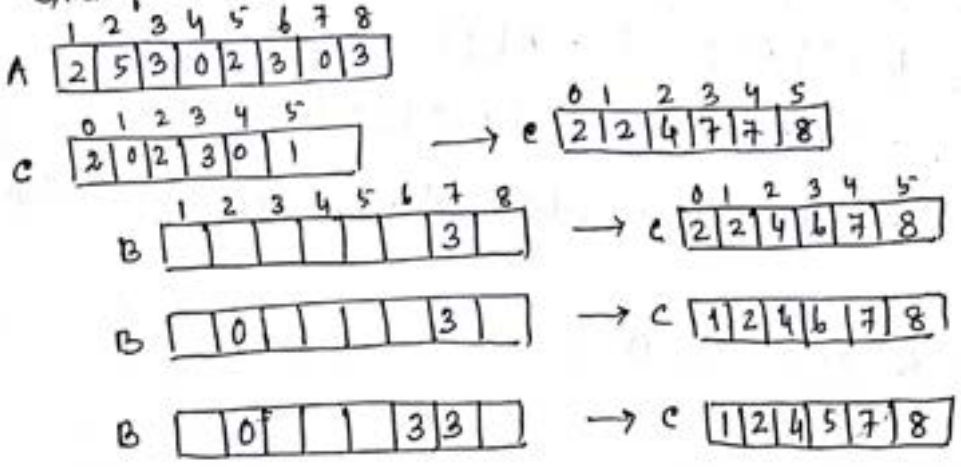
→ The basic idea of Counting sort is, for each input element X, determine the number of elements less than X.

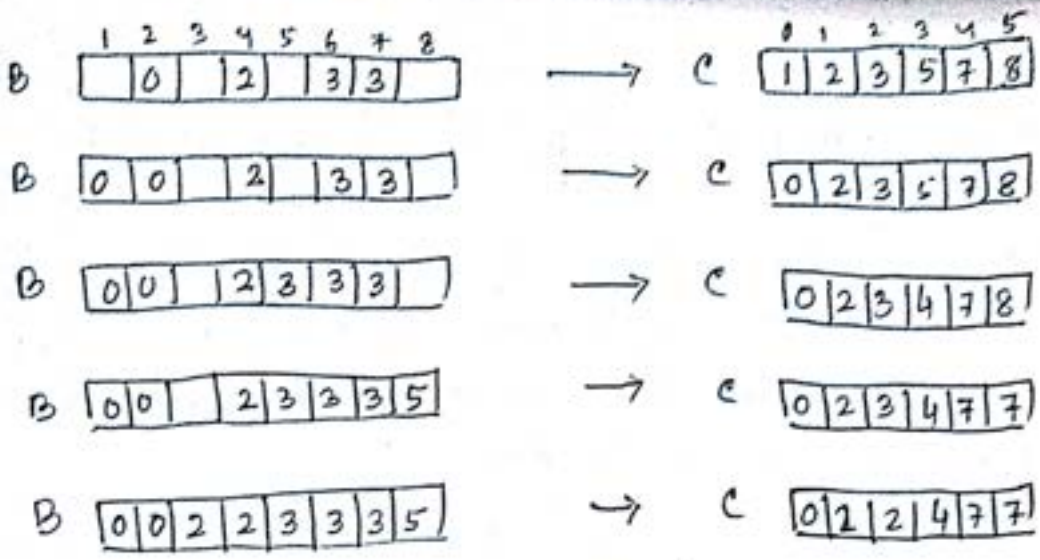
→ For example, if if 5 number of elements are less than X, then X belongs to position 6 in the output array or list.

For the code of counting sort, Let us assume that the input array is A[1..n], and thus the length of the array is 'n'. We also required two other arrays. They are:-

1. array B[1..n] holds the sorted output.
2. array C[0..K] provides temporary working storage.

Let's illustrate the counting sort with a suitable example. (i.e <2, 5, 3, 0, 2, 3, 0, 3>)





In the above figure the execution of counting sort is illustrated with an input array A[1..8], where each element of A is a nonnegative integer and the maximum value of all input is no longer than 5 i.e K = 5.

The Algorithm of counting sort is given below:

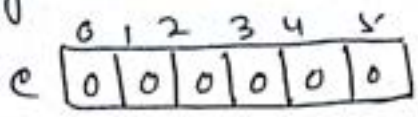
Counting Sort (A, B, K)

1. Let C[0..K] be a new array (temporarily).
2. for i ← 0 to K
3. C[i] ← 0
4. for j ← 1 to A.length
5. C[A[j]] ← C[A[j]] + 1
6. for i ← 1 to K
7. C[i] ← C[i] + C[i-1]
8. for j ← A.length down to 1
9. B[C[A[j]]] ← A[j]
10. C[A[j]] ← C[A[j]] - 1

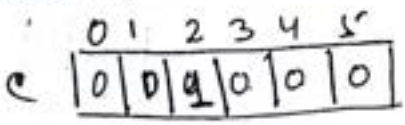
Let us execute the algorithm on the following array.

⟨2, 5, 3, 0, 2, 3, 0, 3⟩

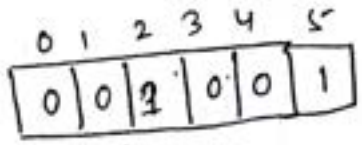
→ After the for loop of line no. 2-3 initializes the c array looks like below.



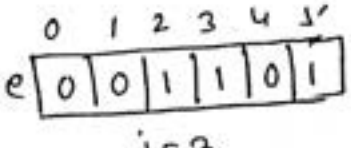
→ when we execute line no. 4 and 5 the values of c array ~~are~~ changes time to time. as shown below.



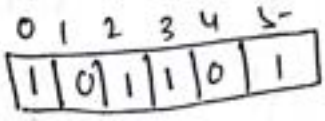
j = 1



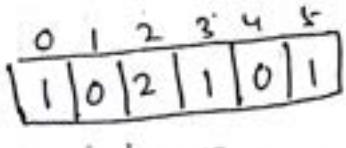
j = 2



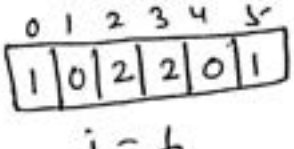
j = 3



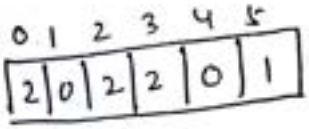
j = 4



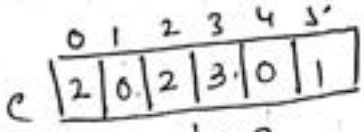
j = 5



j = 6

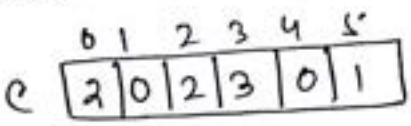


j = 7

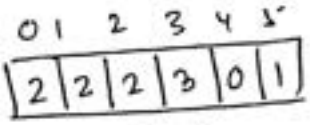


j = 8

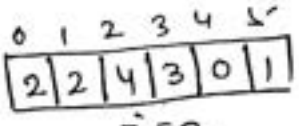
→ when we execute the line no. 6 and 7 the values of c array are changes time to time as shown below.



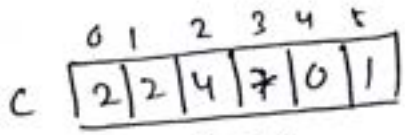
i = 0



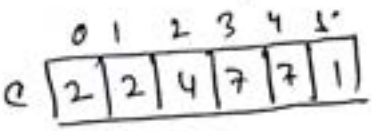
i = 1



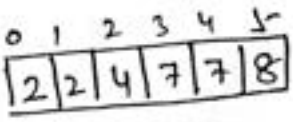
i = 2



i = 3

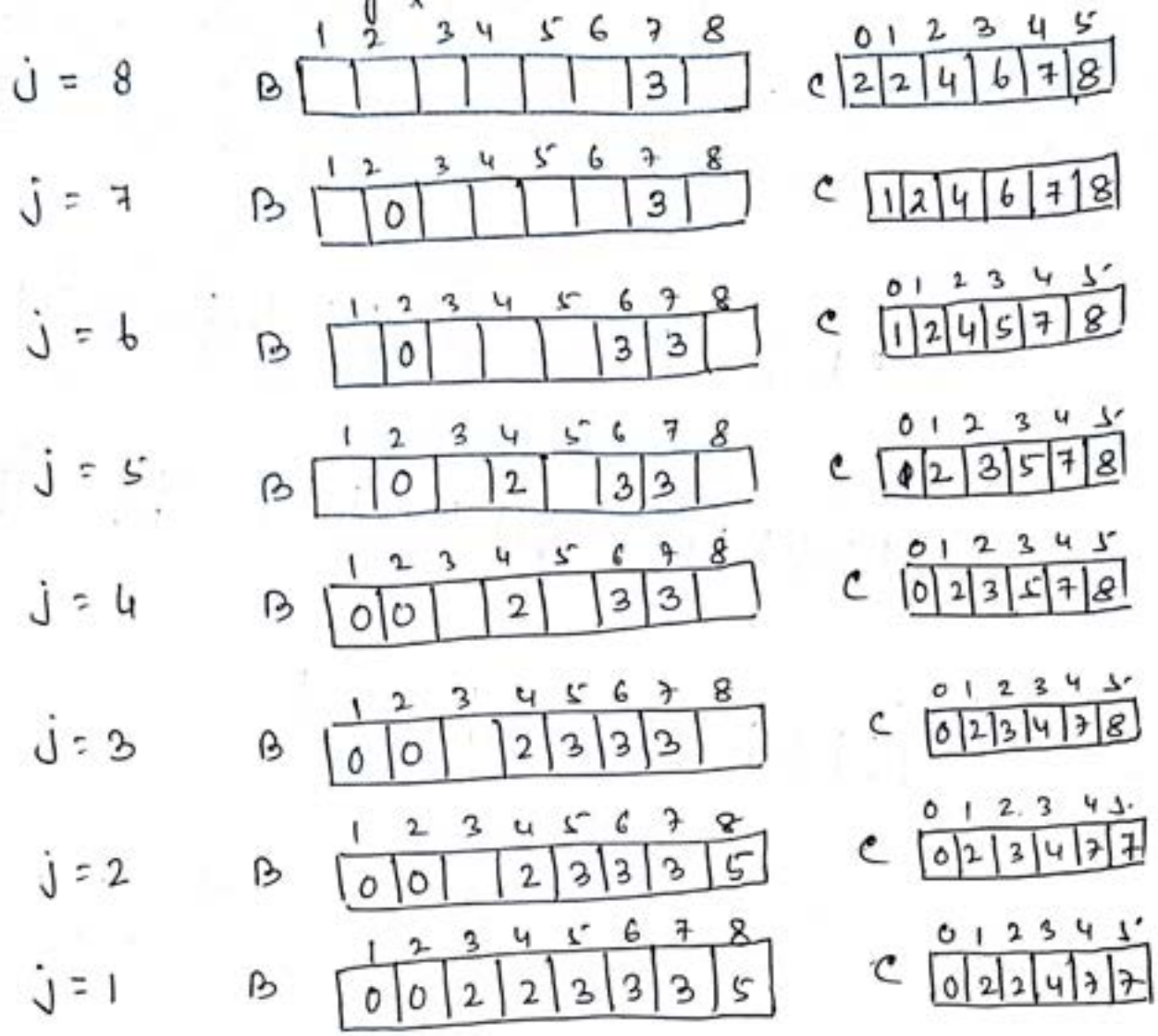


i = 4



i = 5

→ when we execute the line no. from 8 to 10, the following changes are done time to time in B[1..n] array as well as in C[0..k] array are shown below.



Analysis of Counting Sort.

Now calculate the time required to green the counting sort.

The for loop 2-3 takes — $\Theta(k)$ times.

The for loop 4-5 takes — $\Theta(n)$ times.

The for loop 6-7 takes — $\Theta(k)$ times.

The for loop 8-10 takes — $\Theta(n)$ times.

Hence the overall time is $\Theta(n+k)$. But in Practice we say $\Theta(n)$ when $k = O(n)$.

Heap Sort.

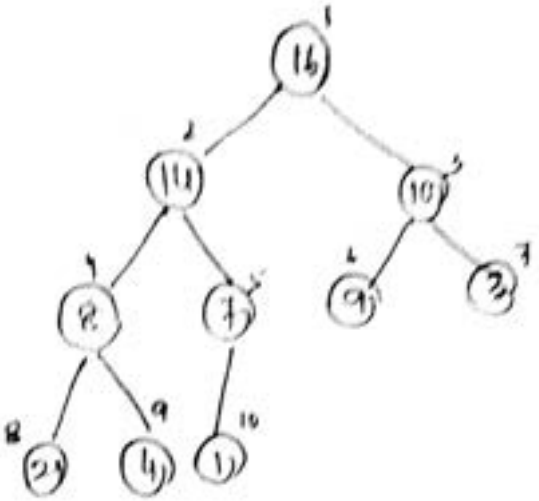
Heap sort is a comparison-based sorting algorithm. It is an in-place sorting algorithm, but is not a stable sort ~~sorting~~, because a sorting algorithm is said to be stable if the two elements have the same key, and remain in the same order or positions after sorting. Hence heapsort is not stable as the operations on the heap can change the relative order of equal elements.

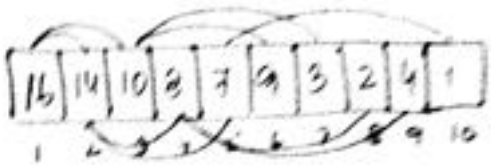
What is heap?

A heap is a tree with some special properties i.e. the parent node of the heap must be greater than or equal ($\geq$) or smaller than or equal ($\leq$) than the value of its children. This is called heap property.

A heap also has the additional property that all leaves should be at h or h-1 levels (where h is the height of the tree) for some $h > 0$ (complete binary tree). i.e the heap should form a complete binary tree.

The heap data structure is an array that we can view as a nearly complete binary tree as shown in the next page.





Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left upto a point.

An array A that represents a heap is an object with two attributes, i.e

→ A.length → gives the number of elements in the array A.

→ A.heap-size → how many elements in the heap are stored within array A.

i.e although A[1 . A.length] may contain numbers but only those elements A[1 .. A.heap-size] are valid elements of the heap. we can other wise says that

$$0 \leq A.\text{heap-size} \leq A.\text{length}.$$

The root of the tree is always A[1], and give the index i of the node. we can easily compute the indices of its parent, left child and right child by using following formulas.

(i) parent(i) ← return $\lfloor i/2 \rfloor$

(ii) Left(i) ← return 2i

Since a heap of n elements is based on a complete binary tree, its height is $O(\lg n)$. The following basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. These operations are.

1. Max – Heapify runs in $O(\lg n)$ time.
2. Build – Max – Heap runs in linear time.
3. Heap sort runs in $O(n \lg n)$ time.
4. Max Heap – Insert, Heap – Extract – Max, Heap – Increase key and Heap – Maximum procedures which run in $O(\lg n)$ time. which allow the heap data structure to implement a priority queue.

Maintaining the heap property.

In order to maintain the max – heap property, we call procedure Max – heapify. Its inputs are an array A and an index i into the array.

```
Max - heapify (A, i)
1. l ← left(i)
2. r ← right(i)
3. if l ≤ A.heap-size and A[l] > A[i]
4.       largest ← l
5. else largest ← i
6. if r ≤ A.heap size and A[r] > A[largest]
7.       largest ← r
8. if largest ≠ i
9. exchange A[i] with A[largest]
10. Max - heapify (A, Largest)
```

return

iii) right (i) ← (2i + 1)

There are two kinds of binary heap or heap:

1. max-heap
2. Min-heap

In both kinds the value in the nodes satisfy a heap-property based on the kind of heap. In max-heap the property is max-heap property, which is given below.

"For every node i other than root

$A[parent(i)] \geq A[i]$

i.e parent always greater than or equal to it's child. Hence the largest element of the max-heap is stored on the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

A min-heap is organized in opposite way: i.e

"for every node i other than root

$A[parent(i)] \leq A[i]$,

i.e parent always smaller than or equal to its child. Hence the smallest element of the min-heap is stored on the root and the subtree rooted at a node contains values no smaller than that contained at the node itself.

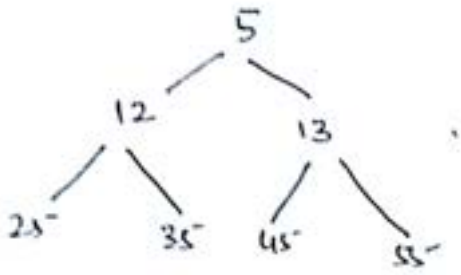
In general for heap sort algorithm we use max-heaps. and Min-heaps commonly implement priority queues.

Analysis of Max-Heapify

Dry Run of Max-Heapify(A,2) with an Example.

Show the operation of Max-Heapify on following array starting from A[2].

5, 12, 13, 25, 35, 45, 55



for i = 2

Max-Heapify (A, 2)

l = 4

r = 5

4 ≤ 7 & A[4] > A[2]

25 > 12 True

Largest ← 4

5 ≤ 7 & A[5] > A[4]

35 > 25 True.

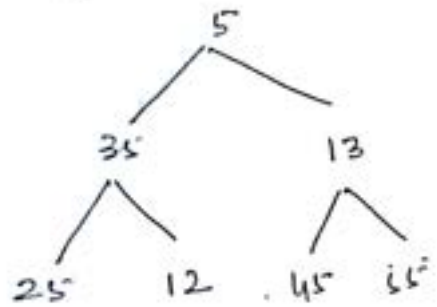
Largest ← 5

If 5 ≠ 2 True

Swap [A[5] ↔ A[2]]

Hence A[2] = 35 & A[5] = 12

So the tree (Modified)



for i = 3.

Max-Heapify (A, 3).

l = 6 r = 7

6 ≤ 7 and A[6] > A[3]

45 > 13 True

Largest ← 6

7 ≤ 7 and A[7] > A[6]

55 > 45 True

Largest ← 7

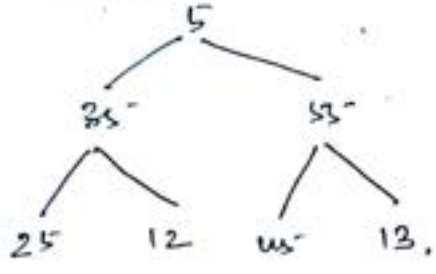
If 7 ≠ 3

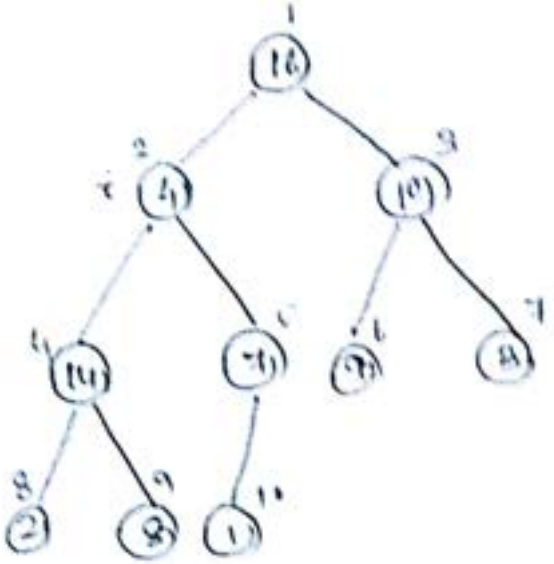
then Swap [A[7], A[3]]

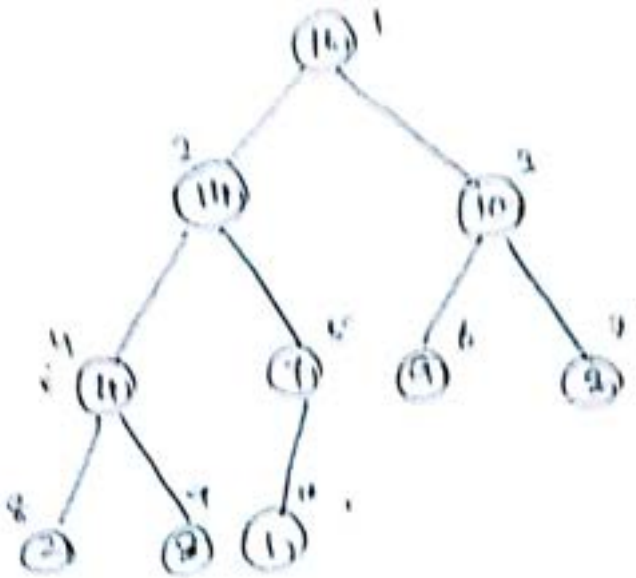
Hence A[7] ← ~~35~~ 13

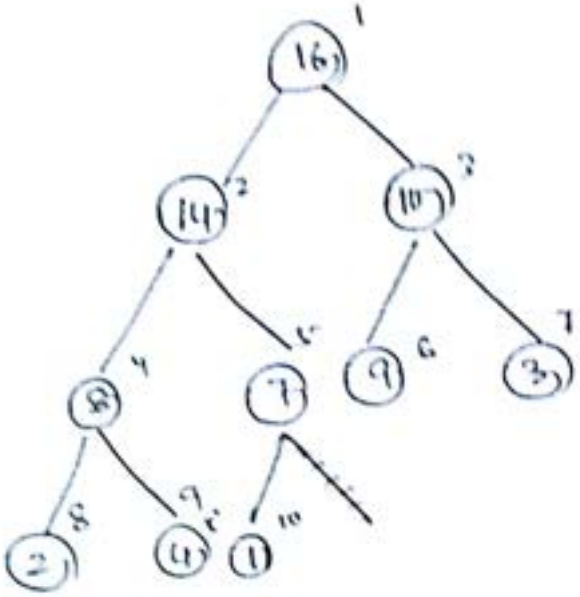
A[3] ← 55

So the tree Modified









The above figures illustrate the action of Max-heapify. At each step, the largest of the elements A[i], A[left(i)], and A[Right(i)] is determined, and its index is stored in largest. If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element and A[i] swap with A[largest] and satisfy the heap property. Consequently Max-heapify called recursively on that subtree.

by applying Master Method / Theorem

$a = 1 \quad b = 3/2 \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Since $f(n) = n^{\log_b a} = 1$, so case 2 of Master Method is applicable. Hence $T(n) = \Theta(n^{\log_b a} \log n)$

$= \Theta(1 \log n)$

$= \Theta(\log n)$.

Proof of Max. height in Heap tree

The children subtree of heap tree each have size almost $2n/3$ - the worst case occurs when the bottom level of the tree is exactly half full.

Let us assume that the total no. of node in a tree is N.

i.e =

Number of nodes in the tree (N)

= 1 + (no. of nodes in Left Subtree)

+ (no. of nodes in Right Subtree).

no. of nodes in Left Subtree

$= 1 + 2 + 3 + \cdots + 2^{h+1} = \frac{2^{h+1+1} - 1}{2-1} = 2^{h+2} - 1$ —(1)

no. of nodes in Right Subtree

$= 1 + 2 + 3 + \cdots + 2^{h} = \frac{2^{h+1} - 1}{2-1} = 2^{h+1} - 1$ —(2)

Q. Show the operation of Max-heapify on the following array starting from A[2]

⟨27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0⟩

Q. Show the operation of Max-heapify on the following array starting from A[2]

⟨5, 3, 17, 10, 84, 19, 6, 22, 9, 3, 10⟩

Q. Show the operation of Max-heapify on the following array starting from A[2]

⟨2, 1, 3, 5, 50, 7, 9, 100, 80, 120, 50⟩

Analysis of Maxheapify:-

→ The running time of Max-heapify on a subtree of size n rooted at a given node i is $\Theta(1)$ time to create and fix the relationship between A[i], A[left(i)] and A[right(i)].

+

→ The time to run Max-Heapify on a subtree rooted at one of the children of node i (with recursive call).

[note: The children subtrees each have size at most $2n/3$ - the worst case occurs when the bottom level of the tree is exactly half-full] [Proof in next page]

Hence the running time of Max-heapify by the recurrence is given below.

$$T(n) \leq T(2n/3) + \Theta(1).$$

Thus plugging in,

no. of nodes in the tree = 1 + no. of nodes in left side subtree + no. of nodes in right side subtree.

$$N = 1 + 2^{h+2} - 1 + 2^{h+1} - 1$$

$$N = 1 + 2 \cdot 2^{h+1} + 2^{h+1} - 2$$

$$N = 2^{h+1}(2+1) - 1$$

$$N = 2^{h+1} \cdot 3 - 1$$

$$N - 1 = 2^{h+1} \cdot 3$$

$$\boxed{\frac{N-1}{3} = 2^{h+1}}$$

Put the value of 2^{h+1} in eqn-1 i.e height of the left subtree.

$$2^{h+2} - 1 = 2 \cdot 2^{h+1} - 1 = 2 \cdot \frac{n+1}{3} - 1$$

$$= \frac{2n+2}{3} - 1$$

$$= \frac{2n}{3} - \frac{1}{3}.$$

$$= \frac{2n-1}{3} < \frac{2n}{3}.$$

Hence the upper bound for the maximum number of nodes in a subtree for a tree with N nodes is

$2N/3$.

Building a heap.

The Max-heapify procedure convert an array[1...n] where n = A.length into a max heap in a bottom up manner.

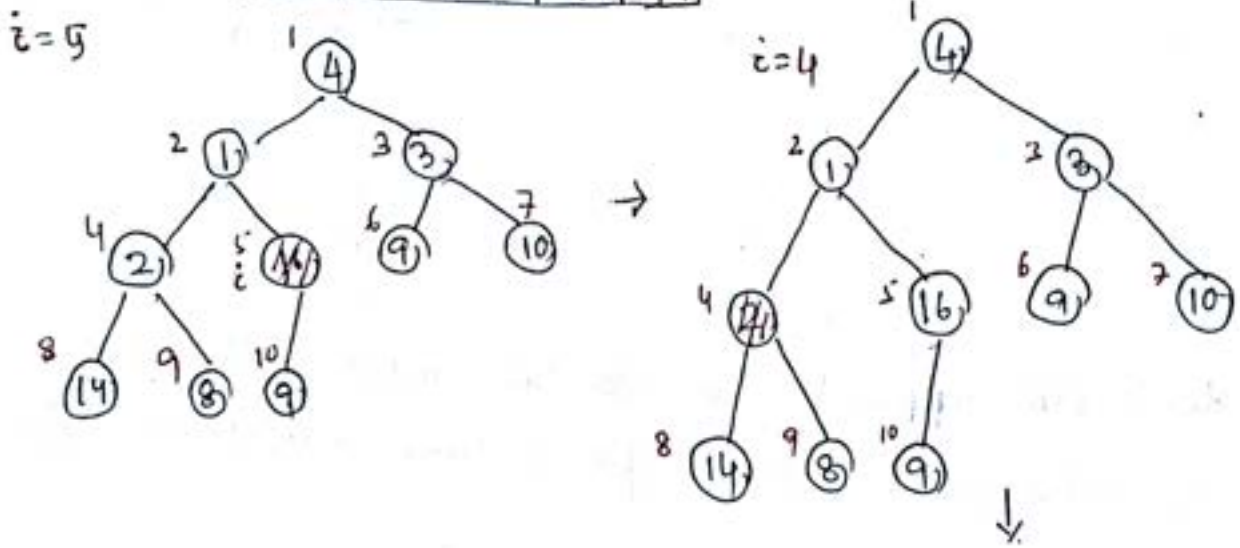
The procedure ~~build heap~~ Build-Max-heap goes through the other nodes of the tree and runs Max-Heapify on each one.

```
Build Max-heap (A)
1. A.heap-size = A.length
2. for i ← ⌊A.length/2⌋ down to 1
3.     Max-Heapify (A, i)
```

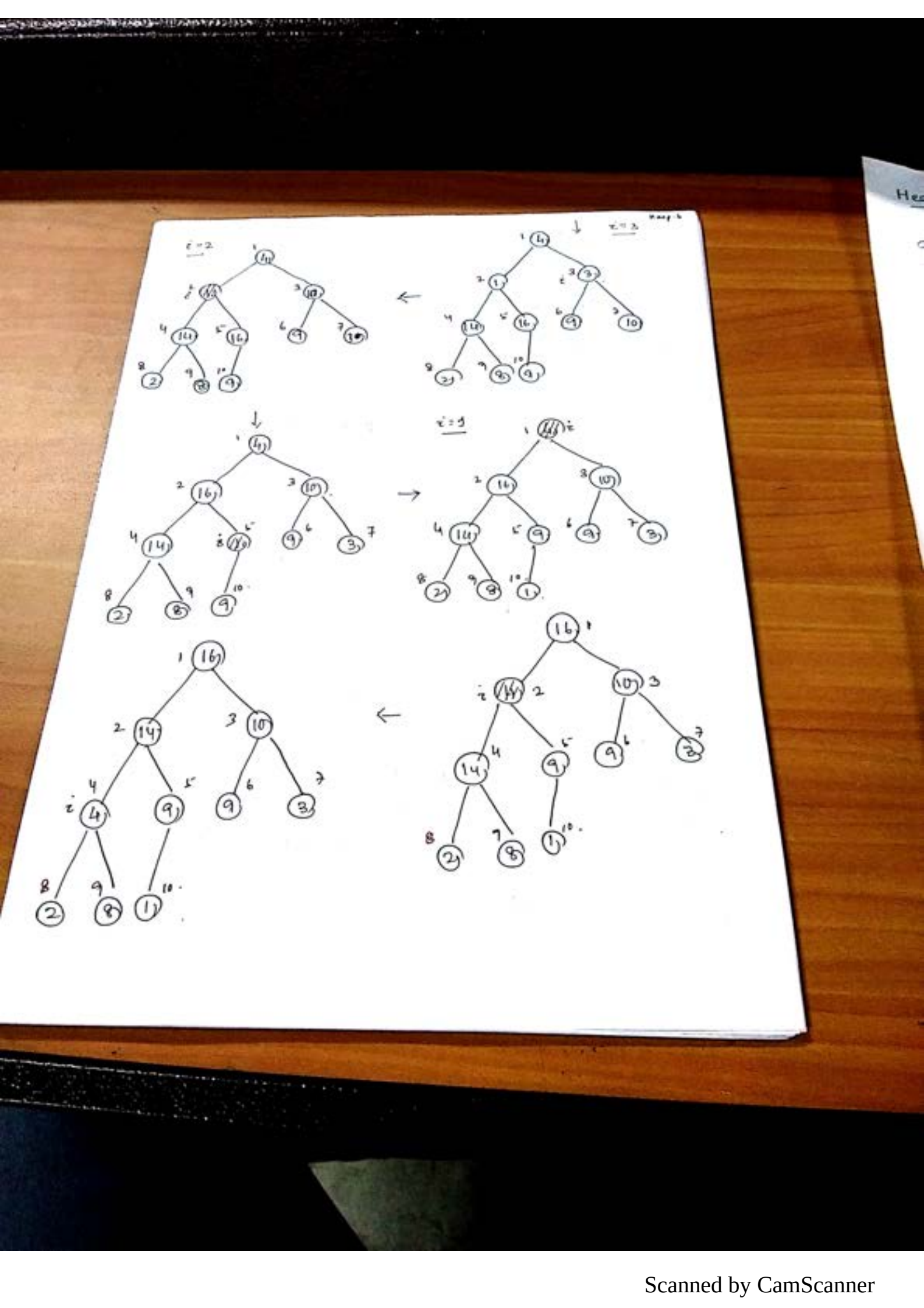
The action of Build Max-heap is give below with an example as dry run.

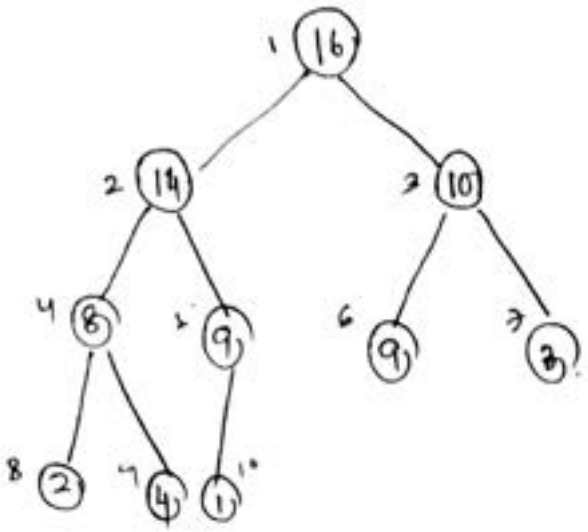
A

4	1	3	2	16	9	10	14	8	7



next page





← final heap tree.

The running time of Build-Max-heap is as follows.

→ Each call to Max-heapify costs $O(\lg n)$ time.

→ Build-Max-heap makes $O(n)$ such calls.

Hence the running time is $O(n \lg n)$.

The Heapsort algorithm.

The heapsort algorithm starts by using Build-max-heap to build a max-heap on the input array $A[1..n]$ where $n = A.length$. and finally the maximum element of the array is stored in $A[1]$ position.

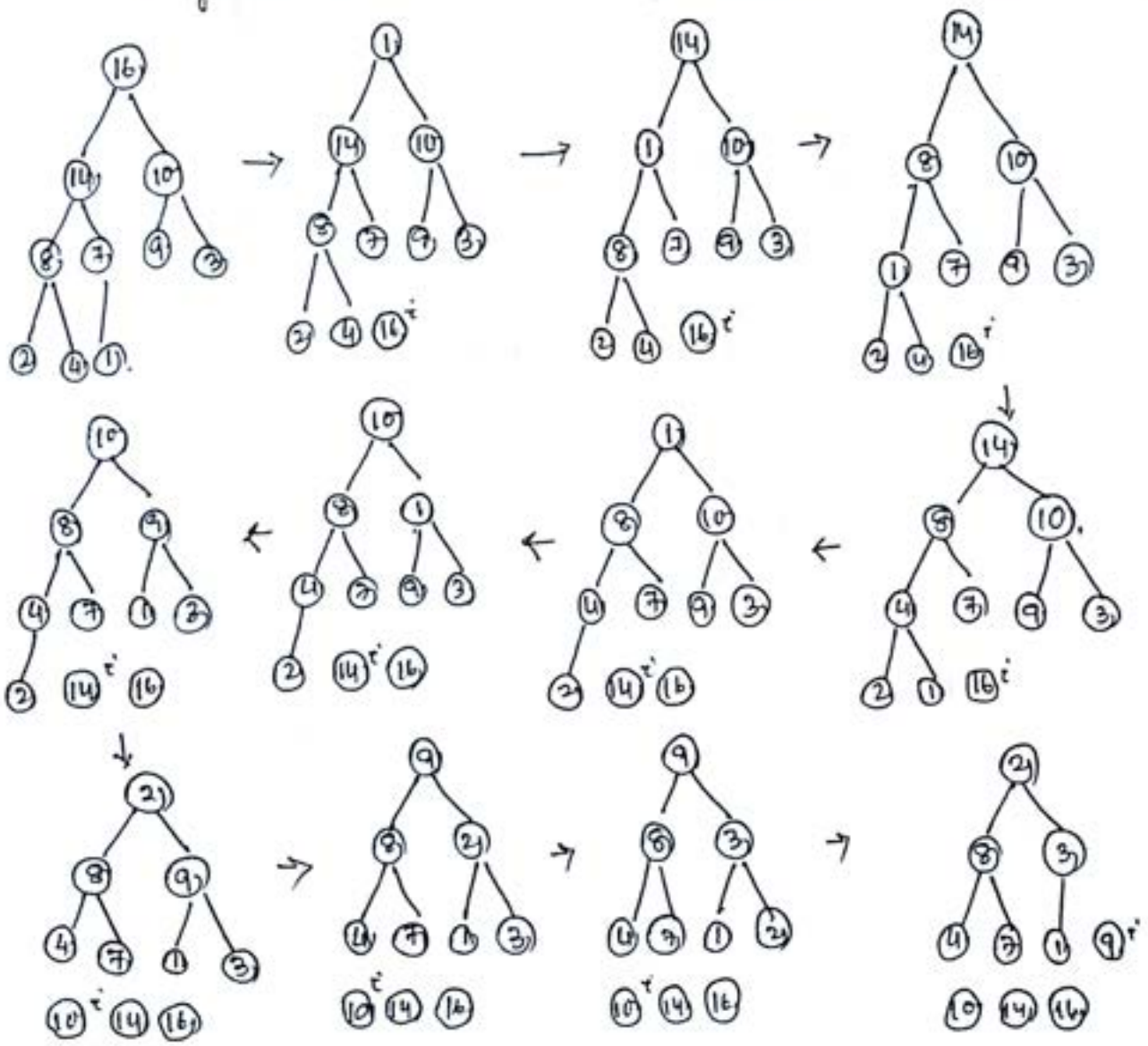
When we start to print, we first display or print the $A[1]$ position element and then replace the last position element with $A[1]$. (i.e $A[n]$ with $A[1]$. and reduce the array size by 1. The new root element might violate the

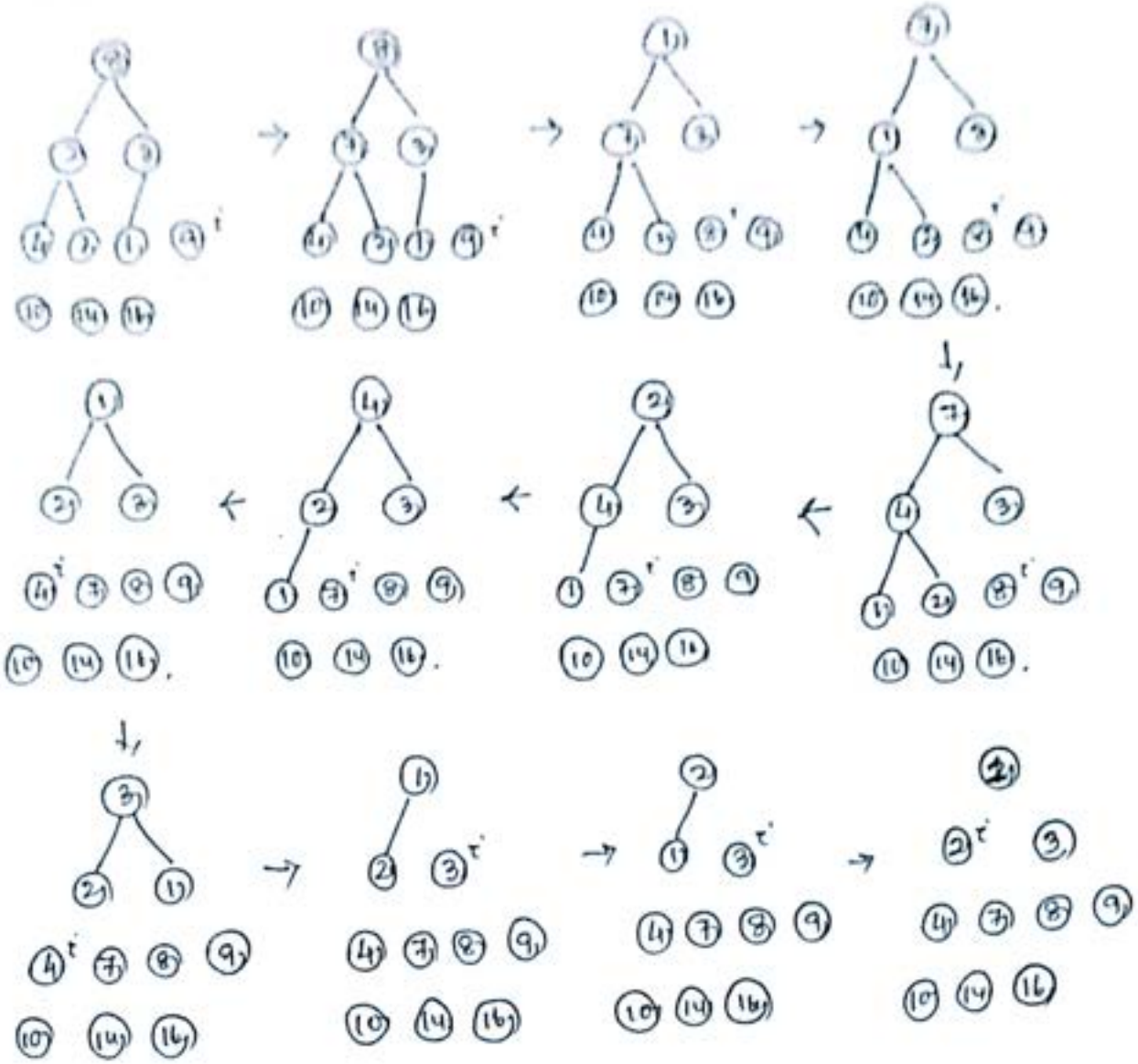
Max-heap property (i.e Parent >, child). The requirement is to restore the max heap property again. This thing done by calling max.heapify(A,1). The heapsort algorithm then repeat this process for the max-heap of size n-1 down to a heap of size 2.

Heapsort (A)

1. Build-max-heap (A)
2. for i ← A.length down to 2
3. exchange A[1] with A[i]
4. A.heap.size = A.heap-size − 1
5. Max-heapify (A,1)

A dry run of Heapsort is given below with an example:





1	2	3	4	7	8	9	10	14	16

(The resulting sorted array)

In case of Heap-Sort, the Build-max-heap takes $O(n)$ time and for each of the $n-1$ call to Max-heapify takes $O(\lg n)$ time. Hence the Heap-Sort procedure takes $O(n \lg n)$ time for execution.

Priority queues.

→ Priority queue is a popular implementation of Heapsort.

→ A priority queue is a data structure for maintaining a set 'A' of elements, each with an associated value called a key.

→ Priority queues are two types.

1. max-priority queues.
2. min-priority queues.

Here we focus on max-priority queues, which is basically based on the concept of max-heap.

A max-priority queue supports the following operations:-

i) Heap-maximum (A) →

* returns the element of A with the largest key.

i.e

```
Heap-Maximum (A)
1. return A[1]
```

∴ This procedure takes Θ(1) time for execution.

ii) Heap-Extract-Max (A)

* Remove and return the element of A with the largest key.
* It is similar to the for loop body of the Heapsort procedure.

Heap-Extract-Max(A)

1. If A.heap-size < 1
2. error "heap underflow"
3. Max = A[1]
4. A[1] = A[A.heap-size]
5. A.heap-size = A.heap_size - 1
6. Max-heapify(A,1)
7. Return Max.

The running time of Heap-Extract-Max is $O(\lg n)$ since it performs only a constant amount of work i.e only one time.

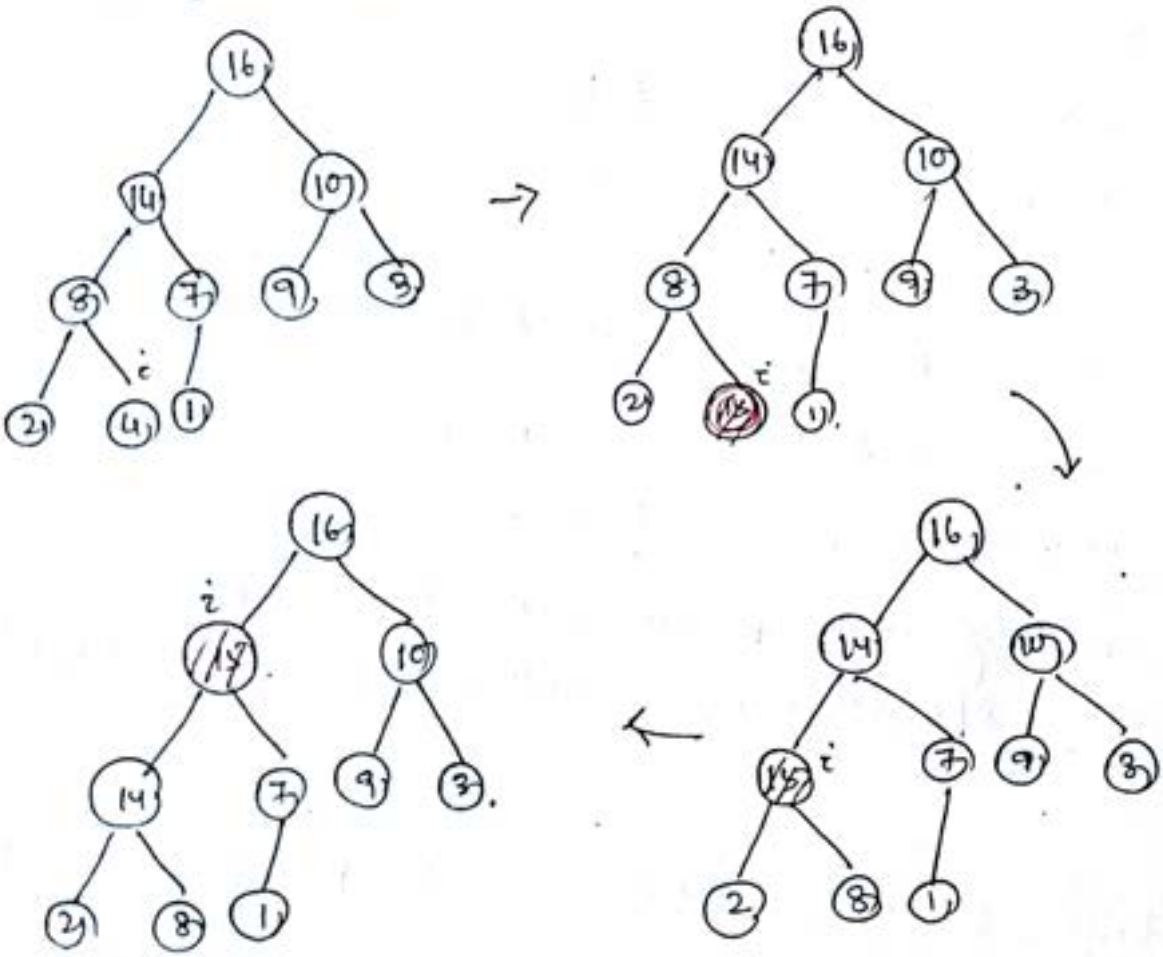
iii) Heap-Increase-key(A, i, k) : Its "increases the value of element's key to the new value k, which is assumed to be at least as large as x's current key value."

An index i into the array identifies the priority-queue element whose key will be increase. The procedure first updates the key of the element A[i] to its new value. Because increasing the key of A[i] might violate the max-heap property. So Heap-Increase-key Procedure traverse the path of newly increased key (i.e item changed) repeatedly compares an element to its parent, exchanging their key and continuing if the element's key is larger, and terminating if the element's key is smaller, or till the max-heap property holds.

Heap-Increase-key (A, i, key)

1. If key < A[i]
2. error "new key is smaller than current key"
3. A[i] = key
4. while i > 1 and A[Parent(i)] < A[i]
5. swap (A[i] with A[Parent(i)])
6. i = parent(i).

A dry run of Heap-Increase-key is given below with example. where i = 9 key = 15



The running time of Heap-Increase-key on an n-element heap is O(lg n), since the path traced and updated time to time upto root has length O(lg n).

iv) Max_heap_Insert(A, key)

"Insert the element key into the set (i.e Array) A, which is equivalent to the operation $A = A \cup \{key\}$"

It takes a key as a new item and inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls Heap increase key to set the key of this new node to its correct value and maintain the max-heap properly.

Max-Heap-Insert(A, key)

1. A.heap-size = A.heap-size + 1
2. A[heap-size] = $-\infty$
3. Heap-Increase-key(A, heap-size, key)

The running time of Max-Heap-Insert on an n-element heap is $O(\lg n)$.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.

Q. Illustrate the operation of Max-Heap-Insert(A, 10) on the heap A = ⟨15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1⟩

Description of QuickSort.

QuickSort is based on three-step process of Divide-and-Conquer.

→ To sort the subarray $A[p..r]$:

Divide: Partition $A[p..r]$ into two possible subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element in the first subarray $A[p..q+1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1..r]$.

Conquer: Sort the subarrays by recursive calls to Quicksort.

Combine: No work is needed to combine the subarrays, because they are sorted in their place.

→ Perform the divide step by a procedure Partition, which returns the index q that marks the position separating the subarrays.

```
Quicksort (A, p, r)
  if p < r
      q ← Partition (A, p, r)
      Quicksort (A, p, q-1)
      Quicksort (A, q+1, p)
```

(Note: Initial call is Quicksort (A, 1, r)

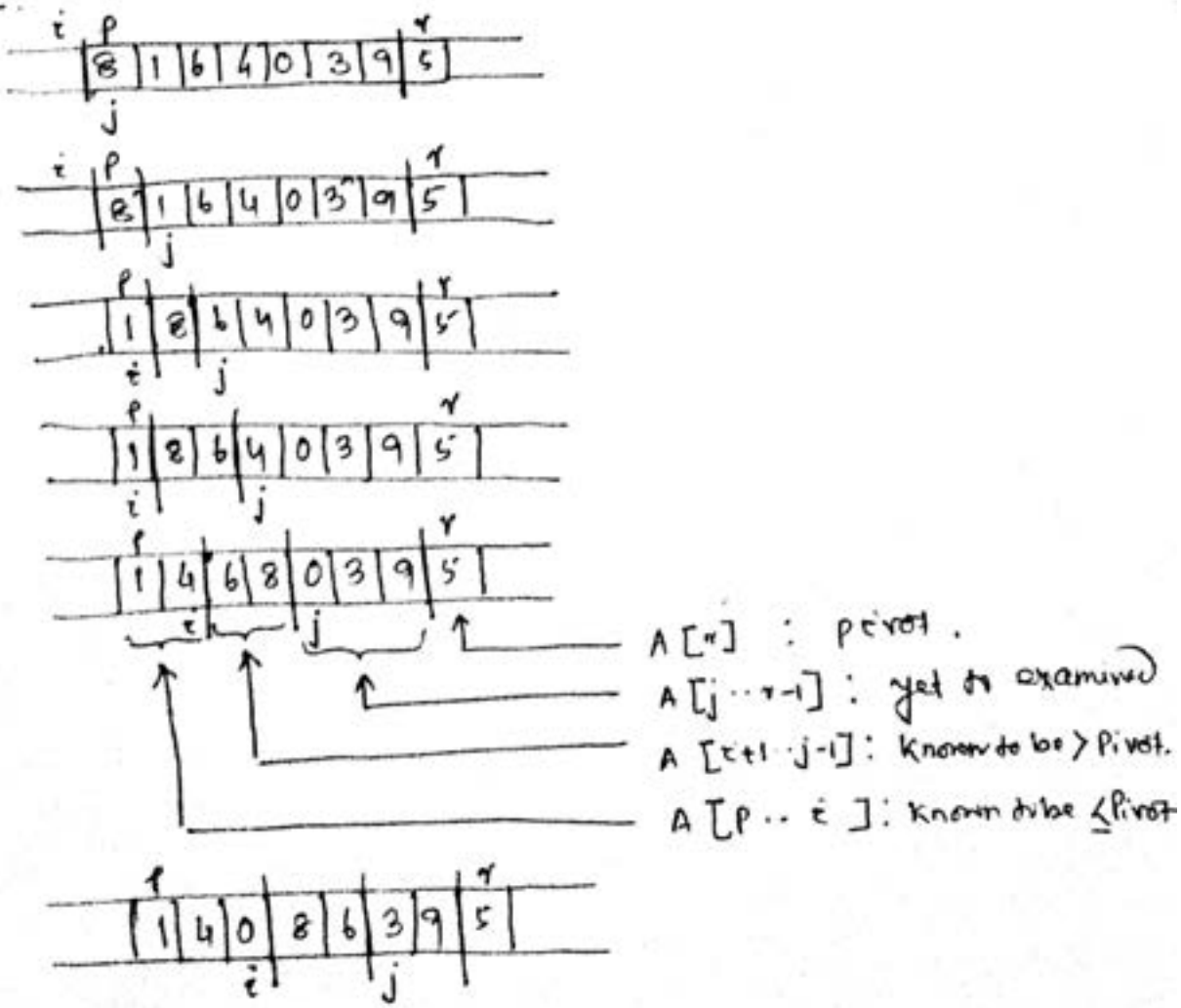
Partitioning:

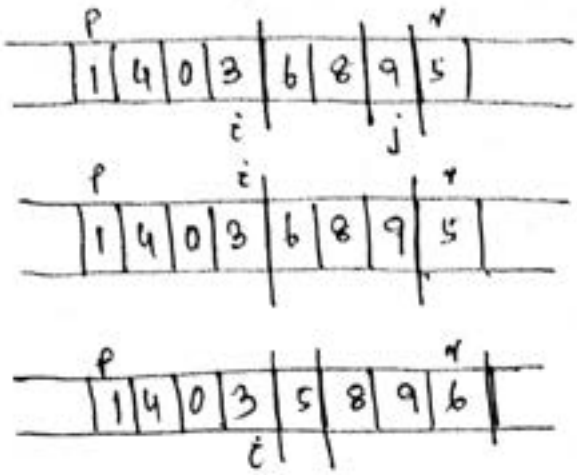
Partition subarray $A[p..r]$ by the following procedure:

```
Partition (A, p, r)
    x ← A[r]
    i ← p-1
    for j ← p to r-1
        if A[j] ≤ x
            i ← i+1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return (i+1)
```

- Partition always select the last element $A[r]$ in the subarray $A[p..r]$ as the pivot - the element around which to partition.

Example: On an 8-element subarray:





(The index j disappear it is no longer needed once the for loop is exited.)

It was observed that as the procedure executes, the array is partitioned into four regions,

Loop invariant:

1. All entries in $A[p..i]$ are $\leq$ pivot.
2. All entries in $A[i+1..j-1]$ are $\geq$ pivot.
3. $A[r]$ = pivot.

Past the fourth region i.e $A[j..r-1]$ whose entries have not yet been examined, so we at that time we don't know how they compare to the pivot.

Correctness:

Use the loop invariant to prove correctness of Partition:

Initialization: Before the loop starts, all the conditions of loop invariant are satisfied, because r is pivot and the subarrays $A[p..i]$ and $A[i+1..j-1]$ are empty.

Maintenance: While the loop is running, if $A[j] <$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then increment only j.

Termination: When the loop terminates, $j = r$, so all elements in A are partitioned into one of the following cases:

$A[p..i] \leq$ pivot

and

$A[i+1..r-1] >$ pivot.

and

$A[r] =$ pivot.

The last two lines of Partition move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray. i.e by swapping $A[r]$ with $A[i+1]$

Time for partitioning:

$\Theta(n)$ to partition an n-element subarray

Performance of Quicksort.

The running time of quicksort depends on the partitioning of the subarrays:

→ If the subarrays are balanced, then the quicksort can run as fast as mergesort.

→ If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case.

* Occurs when the subarrays are completely unbalanced.
* Have '0' elements in one subarray and $n-1$ elements in the other subarray.

* Get the recurrance (O(n) plus two recursive calls on list of size 0 and n-1)

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$= T(n-1) + \Theta(n)$$

$$= \Theta(n^2).$$

* Same running time as insertion sort.
* In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best Case

* Occurs when the subarrays are completely balanced, every time.
* Each subarray has $\leq n/2$ elements
* Get the recurrance

$$T(n) = 2T(n/2) + \Theta(n)$$

$$= \Theta(n \lg n)$$

Balanced partitioning

* Quicksort's average running time is much closer to the best case than to the worst case.
* Imagine that partition produces a 9-to-1 split.
* Get the recurrance

$$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$$

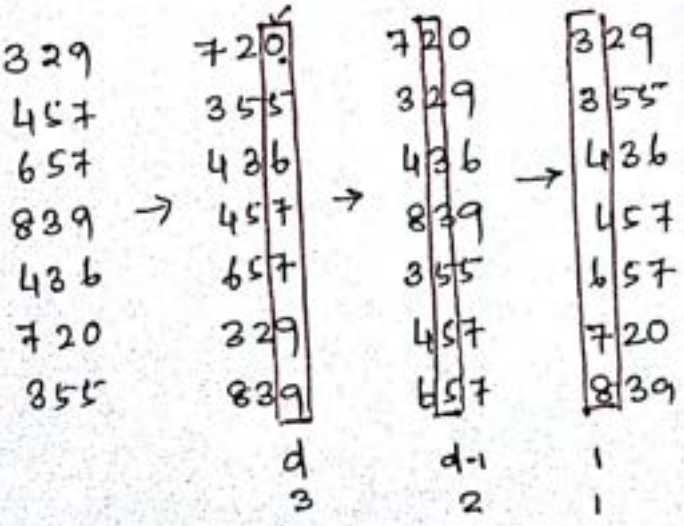
$$= O(n \lg n)$$

Radix Sort:

* Radix Sort is an algorithm, where sorting is also not based on comparisons between the elements.
* This sort takes linear time to sort an array.
* The working principles of Radix sort is given below,
  1. First a list of d-digit numbers is taken as input. where d is the size of a number. For example, if we take a number 345 then the d value is 3.
  2. Then the d-digit number would occupy a field of d columns.
  3. Then we start to sort all the numbers of the list on the basis of d^{th} column. And then we again sort the array again with $(d-1)^{th}$ column. This process is continue until we sort the list with 1st column.
  4. Hence d numbers of passes are required to sort a list.

* This sort is a stable sort.

An example of Radix sort is given below, where the operation of Radix sort on a list of seven 3-digit numbers is illustrated,



The Radix Sort is straightforward. The following procedure assumes that each element in the n-element array A has d-digits, where digit 1 is the lowest-order digit and digit d is the highest order digit.

Radix-Sort(A, d)

1. for i ← d down to 1
2. Use a stable sort to sort array A on digit i (i.e. call Counting Sort.)

Hence we call d times the counting sort. Hence the time complexity of Radix Sort is $\Theta(d(n+k))$.

Bucket Sort or (bin sort)

Bucket sort is a comparison sort algorithm that operates on elements by dividing them into different buckets and then sorting these buckets individually. Each bucket is sorted individually using a separate sorting algorithm or by applying the bucket sort algorithm recursively.

This sort is very useful when the input is uniformly distributed over a range.

Working Principle of Bucket Sort.

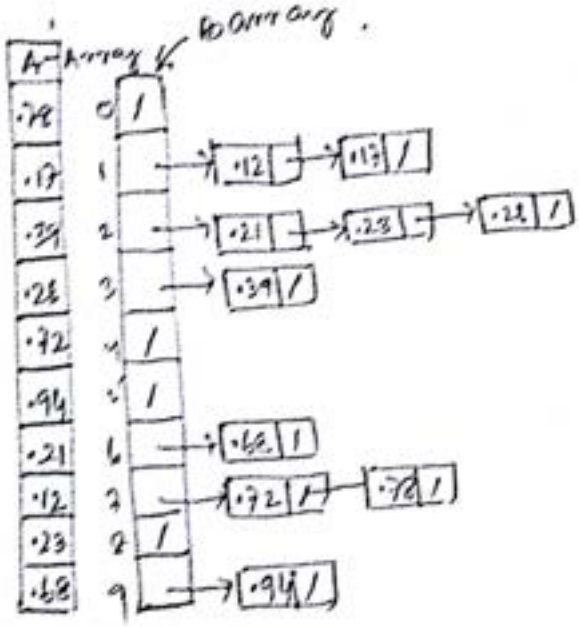
1. Create an empty array
2. Loop through the original array and put each object in a 'bucket'.
3. Search each of the non-empty buckets.
4. Check the buckets in order and then put all objects back into the original array. i.e Sort each bucket using a simple algo. (e.g insertion sort)

Characteristics of Bucket Sort:

1. Bucket Sort assumes that the input is drawn from a Uniform distribution.
2. The computational complexity estimates involves the number of buckets.
3. Bucket sort can be exceptionally fast because of the way elements are assigned to buckets, typically using an array where the index is the value.
4. This means that more auxiliary memory is required for the buckets at the cost of running time than more comparison sorts.

Let us illustrate the bucket sort on the following array.

<.78, .17, .39, .26, .72, .94, .21, .12, .23, .68>



The algorithm of bucket sort.

1. let B[0..(n-1)] be a new array.
2. n ← A.length
3. for i ← 0 to n-1
4. make B[i] an empty list.
5. for i ← 1 to n
6. insert A[i] into list B[⌊n·A[i]⌋]
7. for i ← 0 to n-1
8. Sort list B[i] with insertion sort.
9. concatenate the list B[0], B[1], ... B[n-1] together in order.

The total expected time to sort all element in bucket sort is linear.

Shell Sort:

→ A highly efficient sorting technique based on "Insertion Sort"

→ This sort avoids large number of shifts as in case of Insertion Sort.

→ The Shell sort is named after its inventor Donald Shell, who published this algorithm in 1959.

→ Shell sort works by comparing elements that are distant in nature. (i.e faraway)

→ The distance of comparison (i.e gap) between elements are decreased as the algorithm runs until the last phase, where the adjacent elements are compared.

→ Due to the above reason, Shell sort is some times reffered as "Diminishing Increment sort".

The distance of comparison or gap is maintain by following methods.

i) divided by 2 (two) (Donald Shell)

ii) Knuth's method.

(i.e gap ← gap * 3 + 1

initially gap start with 1)

Let us execute an example with the help of Knuth's gap method on following array.

⟨35, 33, 42, 10, 14, 19, 27, 44⟩

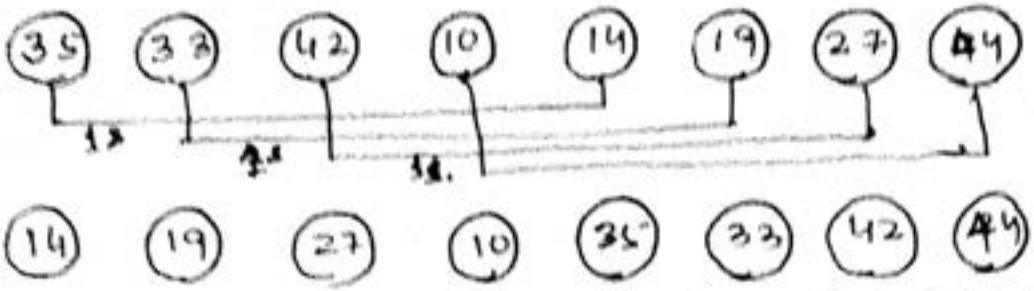
First we take the gap = 4 by using Knuth Method.

i.e (gap = 1 (initially)

gap = gap * 3 + 1

= 1 * 3 + 1

= 4.



we compare each variable and swap them (if required) for original array only.

Again use Knuth's formula for finding the gaps.

i.e $gap = gap \times 3 + 1$

$\Rightarrow gap = \frac{gap - 1}{3.} = \frac{4-1}{3} = 1$

now shell sort uses the insertion sort to sort the array. The step by step depiction is given below.

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 19 19 27 35 33 42 44

14 19 19 27 35 33 42 44

19 19 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 34 33 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

Hence only four no. of swap is required to sort the total array. So total seven (7) number of swap required to sort the above array by shell sort.

Algorithm for Shell Sort.

```
1. gap = 1
2. While (gap < A.length/3)
3. {
4.     gap = gap * 3 + 1;
5. }
6. While (gap > 0)
7. {
8.  for (outer = gap ; outer < A.length; outer++)
9.  {
10.     Ins_value = A[outer];
11.     inner = outer;
12.     While(inner > gap-1 && A[inner-gap] > Ins_value)
13.     {
14.         A[inner] = A[inner-gap];
15.         inner = inner - gap;
16.     }
17.     A[inner] = Ins_value;
18. }
19. gap = (gap-1)/3
20.}
```

Let us dry run the above algo with the help of same example i.e

< 35, 33, 42, 10, 14, 19, 27, 44 >

A.length = 8 & gap = 1

After the first ~~loop~~ time execution the gap = 4.

now gap > 0. i.e 4 > 0

now. outer = 4 ; outer < 8 ; ~~or~~ outer++.

Ins_value = A[outer] = A[4] = 14

Inner = outer So inner = 4

line no. 12 is True. ~~So~~ Hence the new, ~~changed~~ array

< (14), 33, 42, 10, (35), 19, 27, 44 >

now outer = 5

5 < 8 True.

ins_value = A[5] = 19

inner = 5

Line no. 12 is True. Hence the new changed array is

< 14, (19), 42, 10, 35, (33), 27, 44 >

now outer = 6

6 < 8 True.

ins. value = A[6] = 27

inner = 6

Line no. 12 is True. Hence the new changed array is

< 14, 19, (27), 10, 35, 33, (42), 44 >

now outer = 7

7 < 8 True.

ins_value = A[7] = 44

inner = 7.

Line no. 12 is False. Hence no change in array

gap = (gap - 1)/3. = (4-1)/3 = 1

now ✓ outer = 1

1 < 8 True.

ins_value = A[1] = 19

inner = 1

Line no. 12 = False. So no change in array

✓ outer = 2

2 < 8 True.

ins_value = A[2] = 27

inner = 2

Line no. 12 = False, So no change in array

✓ outer = 3

3 < 8 True.

ins_value = A[3] = 10.

inner = 2

Line no 12 = True, So the new array

Finally the sorted array is

< 10, 14, 19, 27, 33, 35, 42, 44 >

The recursive Shell Sort Algorithm is given below.

```
gap = 1
while (gap < A.length/3)
{
    gap ← gap * 3 + 1;
}
Recursive_Shell_Sort (A, A.length, gap)
{
    if (gap <= 0)
        return A;
    else
      for (outer = gap; outer < A.length; outer++)
      {
        ins_value = A[outer];
        inner = outer;
        while (inner > gap-1 && A[inner-gap] >= ins_value)
        {
          A[inner] = A[inner-gap];
          inner = inner - gap;
        }
        A[inner] = ins_value;
      }
      Recursive_Shell_Sort (A, A.length, (gap-1)/3)
}
```

< 14, 19, ~~24~~ 10, 27, 35, 33, 42, 44 >

inner : inner - interval = 3 - 1 = 2

Line no. 12 is True, Hence the new changed array again

< 14, 10, 19, 27, 35, 33, 42, 44 >

inner : inner - interval = 2 - 1 = 1

Line no. 12 is again True. The new changed array is

< 10, 14, 19, 27, 35, 33, 42, 44 >

inner = 0.

Line no. 12 is False. Hence

✓ outer = 4

4 < 8 True.

inner = 4

Ins. value = A[4] = 35

Line no. 12 is false, Hence outer again increment.

✓ outer = 5

5 < 8 True.

inner = 5

Ins. value = A[5] = 33

Line no. 12 is True. The new changed array is

< 10, 14, 19, 27, 33, 35, 42, 44 >

inner = 4

Line no. 12 is is Faalse, Hence outer value increased

✓ outer = 6

6 < 8 True.

Ins. value = A[6] = 42

inner = 6

Line no. 12 is false so outer increment by 1

outer = 7

7 < 8 True.

Ins_value = A[7] = 44

inner = 7

Line no. 12 is false, so outer increment by 1

outer = 8

8 < 8 false. interval decrement to zero.

Hence algorithm is stop.

Analysis of Shell Sort.

→ Shell sort is efficient for medium size lists.

→ For bigger lists, the algorithm is not the best choice.

→ But it is the fastest of all $O(n^2)$ sorting algorithms.

→ The best case in shell sort is when the array is already sorted in the right order. i.e $O(n)$.

→ The worst case time complexity is based on the gap sequence. That's why various scientists give their gap intervals. They are

1. Donald shell ~~gave~~ give the gap interval $n/2$.
2. Knuth gave the gap interval = $g = g*3+1$
3. Hibbard give the gap interval = 2^{k-1}.

Hence the worst/Avg. case complexity = $O(n \log^2 n)$

Q. Apply shell sort on the following array.

$\langle 20, 35, 18, 8, 14, 41, 3, 39 \rangle$