Lecture 1 - Introduction to Design and analysis of algorithms

What is an algorithm?

Algorithm is a *set of steps to complete a task*.

For example, Task: to make a cup of tea. Algorithm:

- add water and milk to the kettle,
- boilit, add tea leaves,
- Add sugar, and then serve it in cup.

An ***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***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The algorithm describes a specific computational procedure for achieving that input/output relationship.

"a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it".

Characteristics of an algorithm:-

- Must take an input.
- Must give some output(yes/no,valueetc.)
- Definiteness –each instruction is clear and unambiguous.
- Finiteness –algorithm terminates after a finite number of steps.
- Effectiveness –every instruction must be basic i.e. simple instruction.

Design Strategy of Algorithm

1. **Divide and Conquer**

   Divide ilie original problem into a set of sub problems, Solve every sub problem individually, recursively. Combine the solutions of sub problems (top level) into a solution of the whole original problem.

2. **Dynammic Programming**

   Dynamic programming is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping subproblems and optimal solution

3. **Branch and Bound**

   In a branch and bound is used for optimization problem; In this a tree of sub problems formed.

4. **Greedy Approach**

Greedy algorithms seek to optimize a function by making choices (greedy criterion) which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best.

5. **Backtracking**

Backtracking algorithms are based on a depth-first recursive search. A backtracking algorithm : first tests to see if a solution has been found, and if so, returns it ; otherwise For each choice that can be made at this point, Make that choice, Recurrence and If the recursion returns a solution, return ok. At last If no choices remain, return failure.

6. **Randomized Algorithm**

Analysis of algoritmns

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure.

We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time .

Running Time of an algorithm is the time taken by the algorithm to execute the successfully. The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine independent as possible

The analysis of an algorithm is to evaluate the performance of the algorithm based on the Input size, Running time (worst-case and average-case) :The running time of an algorithm on a particular input is the number of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.

Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbf{N} = \{0, 1, 2, \ldots\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes.

It is a way to describe the characteristics of a function in the limit. It describes the rate of growth of functions. Focus on what's important by abstracting away low-order terms and constant factors. a**symptotic Notations** are the expressions that are used to represent the complexity of an algorithm.

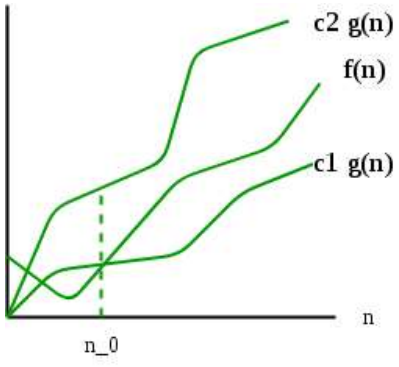
Best Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.**Worst Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.**Average Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Θ-notation

Θ notation denote the tight bound for a given function .For a given function g(n), we denote by Θ(g(n)) the set of functions and read as Θ of g(n). It is defined as

Θ(g(n)) = {f(n) : there exist positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0}.

A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be "sandwiched" between c1g(n) and c2g(n), for sufficiently large n.



f(n) = theta(g(n))

Question : Consider the function f(n)= n^2 /2 – 3 n . Show that f(n) = Θ (n^2

Solution: According to the definition

Θ(g(n)) = {f(n) : there exist positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0}.

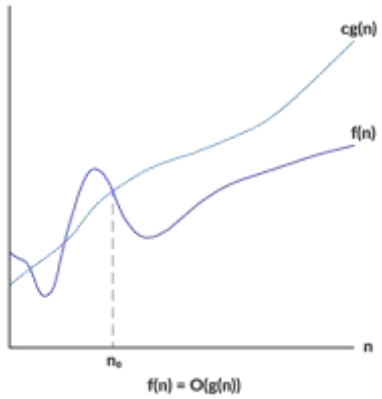
so, we must determine positive constants c1, c2, and n0 such that
c1 n2 ≤ 1/2n^2 - 3n ≤ c2 n2 for all n ≥ n0.
Dividing by n^2 yields
c1 ≤ 1/2 - 3/n ≤ c2.

The right-hand inequality can be made to hold for any value of $n \geq 1$ by choosing $c2 \geq 1/2$. Likewise, the left-hand inequality can be made to hold for any value of $n \geq 7$ by choosing $c1 \leq 1/14$. Thus, by choosing $c1 = 1/14$, $c2 = 1/2$, and $n0 = 7$, we can verify that $1/2n2 - 3n = \Theta(n2)$.

***O*-notation**

Big O notation specifically describes worst case scenario. It represents the upper bound running time complexity of an algorithm When we have only an ***asymptotic upper bound***, we use *O*-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n") the set of functions. It is defined as

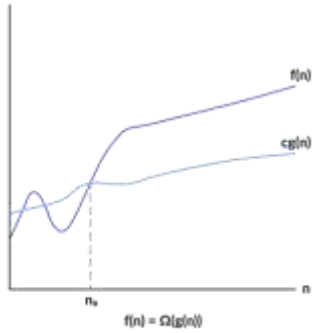
O(g(n)) = {f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0}.



Ω-notation

Just as *O*-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "bigomega of g of n") the set of functions. It is defined as

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and $n0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n0\}$.



Theorem : For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Note: The growth patterns above have been listed in order of increasing "size." That is,

$O(1), O(\lg(n)), O(n \lg(n)), O(n^2), O(n^3), \ldots, O(2^n)$.

<u>*o*-notation</u>

The asymptotic upper bound provided by *O*-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use *o* notation
to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n0\}$.

For example, $2n = o(n^2)$, but $2n2 \ne o(n2)$.

The function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity, *i.e.*,

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

For example, $2n = o(n^2)$

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

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n0\}$.

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that if the limit exists.

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

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Theorem :Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Problem : Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

Problem: Is $2n+1 = O(2n)$? Is $22n = O(2n)$?

Recurrences

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g. the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

T(n)= Θ(1) ; if n=1 or 2T(n/2) + Θ(n) ;if n>1 whose solution can be found as T(n)=Θ(nlog n)

A recurrence is an equation or inequality that describes a function interms of its values on smaller input. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

T (n) = θ (1) if n=1 or
T (n) = 2T (n/2)+ θ (n) if n>1

There are four methods for solving Recurrence:

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

Master Method

The Master Method is used for solving the following types of recurrence
T (n) = a T(n/b)+ f (n) with a≥1 and b≥1 be constant & f(n) be a function and Let T (n) is defined on non-negative integers by the recurrence.

Case1: If f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ε >0, then it follows that:

T (n) = Θ $\left(n^{\log_b a}\right)$

Example: T (n) = 8 T (n/2)+100 n^2 apply master theorem on it.

Solution: Compare T (n) = 8 T (n/2)+100 n^2 with

T (n) = a T(n/b)+ f (n) with a≥1 and b≥1
a = 8, b=2, f (n) = 1000 n^2, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$
1000 n^2 = O ($n^{3-\varepsilon}$)
If we choose ε=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ $\left(n^{\log_b a}\right)$

Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F (n) = Θ $\left(n^{\log_b a} \log^k n\right)$ then it follows that: T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

Example: T (n) = 2 **T(n/2)+ 10 n** solve the recurrence by using the master method.

Solution As compare the given problem with **T (n) = a T(n/b)+ f (n)** with a≥1 and b≥1

a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ $\left(n^{\log_b a} \log^k n\right)$, we will get

10n = Θ (n^1) = Θ (n) which is true.

Therefore: T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

= Θ (n log n)

Case 3: If it is true f(n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ for some constant ε >0 and it also true that: a f $\left(\frac{n}{b}\right) \le cf(n)$ for some constant c<1 for large value of n ,then :

1. T (n) = Θ((f (n))

Example: Solve the recurrence relation:

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

Solution:

Compare the given problem with T (n) = a T (n/b) + f(n) , a>1 and b>=1

a= 2, b =2, f (n) = n^2, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ put ε =1, then the equality will hold.

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

Now we will also check the second condition:

$2\left(\frac{n}{2}\right)^2 \le cn^2 \Rightarrow \frac{1}{2}n^2 \le cn^2$

If we will choose c =1/2, it is true:

$\frac{1}{2}n^2 \le \frac{1}{2}n^2$ ∀ n ≥1

So it follows: T (n) = Θ ((f (n))

T (n) = Θ(n^2)

$T(n) = 3T(n/2) + n2$

Here,

$a = 3$

$n/b = n/2$

$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

Problem-01: Solve the following recurrence relation using Master's theorem $T(n) = 3T(n/2) + n^2$

Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-$a = 3, b = 2, k = 2, p = 0$

Now, $a = 3$ and $b^k = 2^2 = 4$.

Clearly, $a < b^k$.

So, we follow case-03.

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

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

Thus, **$T(n) = \theta (n^2)$**

Problem-02:Solve the recurrence relation using Master's theorem- $T(n) = 2T(n/2) + n\log n$

Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-$a = 2, b = 2$

Now, $a = 2$ and $b^k = 2^1 = 2$.

Clearly, $a = b^k$.

So, we follow case-02.

Since p = 1, so we have-

$$T(n) = \theta (n^{\log_b a}.\log^{p+1} n)$$

$$T(n) = \theta (n^{\log_2 2}.\log^{1+1} n)$$

Thus,

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

Problem-03:

Solve the following recurrence relation using Master's theorem-

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

Solution- We compare the given recurrence relation with T(n) = aT(n/b) + f(n).
Then, we have-a = 2,b = 4

Now, a = 2 and $b^k = 4^{0.51} = 2.0279$.So, we follow case-03.

so we have-

$$T(n) = \theta (n^k \log^p n)$$

$$T(n) = \theta (n^{0.51} \log^0 n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta (n^{0.51})}}$$

Problem-04: Solve the following recurrence relation using Master's theorem-

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

Solution- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
Then, we have- $a = \sqrt{2}$, b = 2

Now, $a = \sqrt{2} = 1.414$ and $b^k = 2^0 = 1$.

Clearly, $a > b^k$.

So, we follow case-01.

So, we have-

$$T(n) = \theta (n^{\log_b a})$$

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

$$T(n) = \theta (n^{1/2})$$

Thus,

$T(n) = \theta (\sqrt{n})$

Problem-05:Solve the following recurrence relation using Master's theorem-

$$T(n) = 8T(n/4) - n^2 \log n$$

Solution-

- The given recurrence relation does not correspond to the general form of Master's theorem.
- So, it can not be solved using Master's theorem.

Problem-06: Solve the following recurrence relation using Master's theorem-

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

Solution:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-$a = 3, b = 3$

Now, $a = 3$ and $b^k = 3^1 = 3$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 0$, so we have-

$$T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$$

$$T(n) = \theta (n^{\log_3 3} . \log^{0+1} n)$$

$$T(n) = \theta (n^1 . \log^1 n)$$

Thus,

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

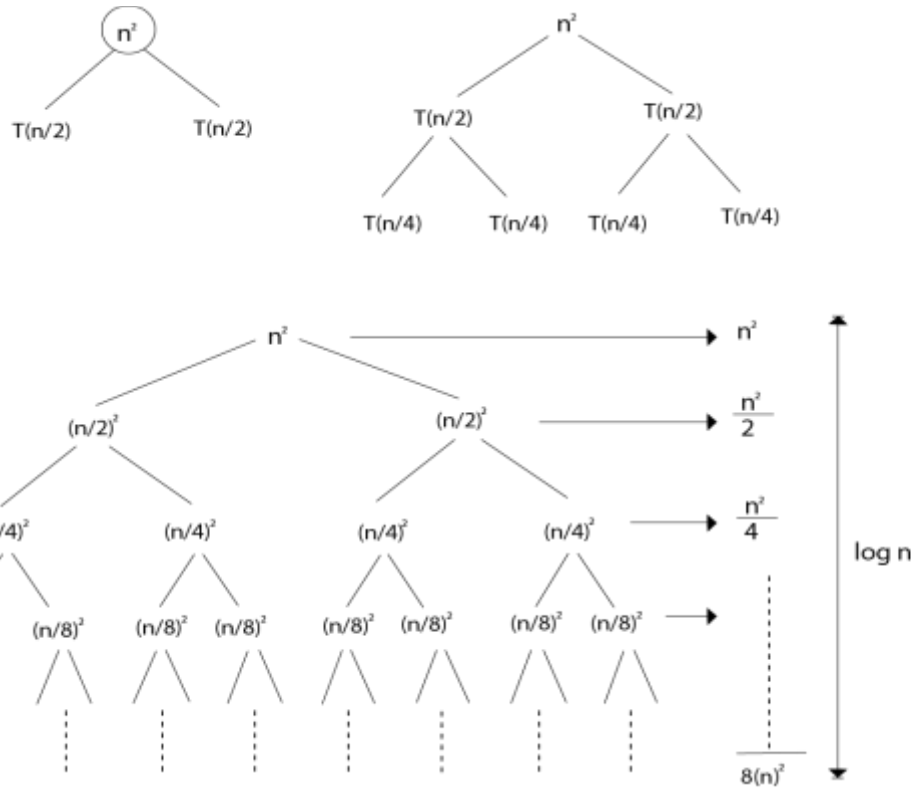
Recursion Tree Method

Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded. we consider the second term in recurrence as root. It is useful when the divide & Conquer algorithm is used.

In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion. Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm.

Example 1 Consider $T(n) = 2T(n/2) + n^2$ We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

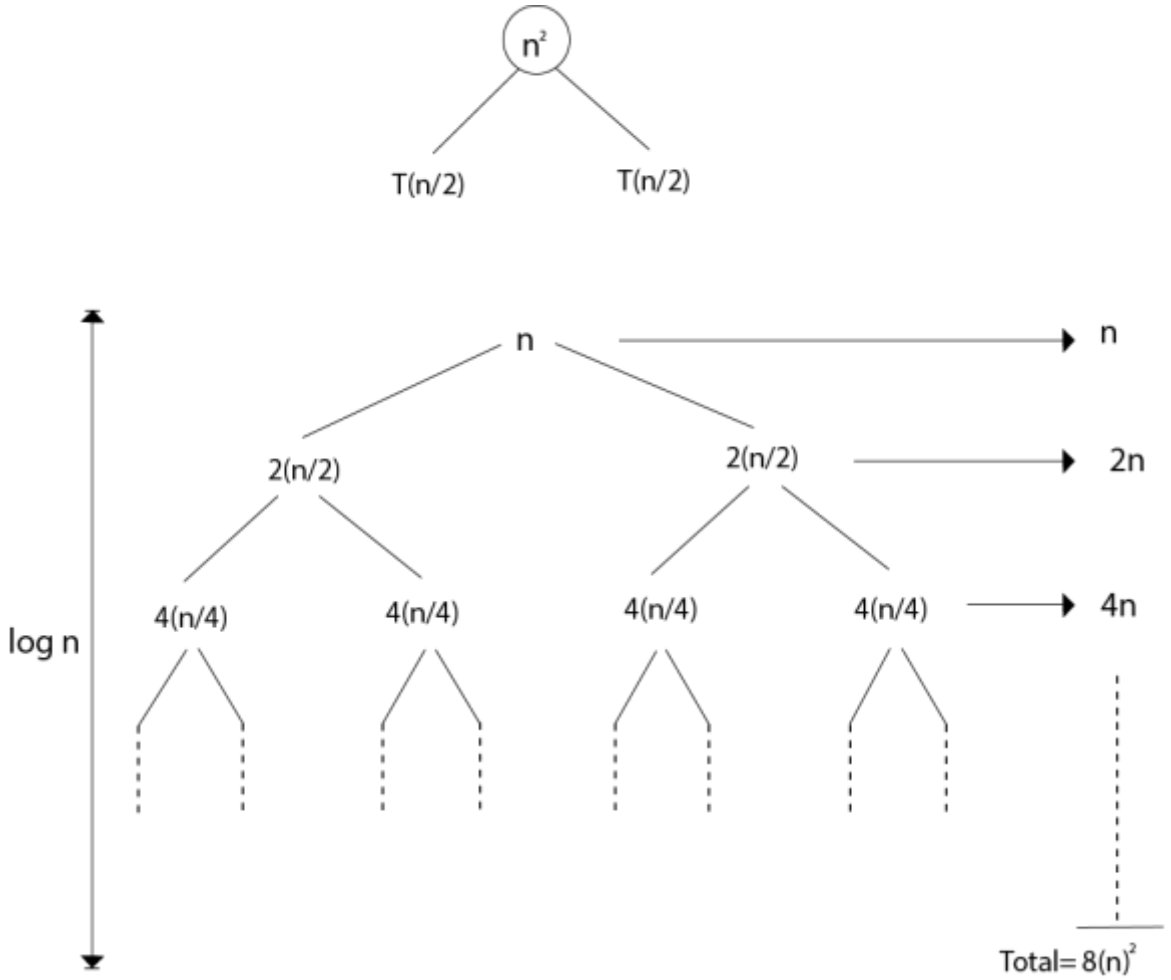
$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

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

Example 2: Consider the following recurrence T (n) = 4T (n/4) +n

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence



We have $n + 2n + 4n + \ldots\ldots \log_2 n$ times

$= n\,(1 + 2 + 4 + \ldots\ldots \log_2 n \text{ times})$

$$= n\frac{(2^{\log_2 n} - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

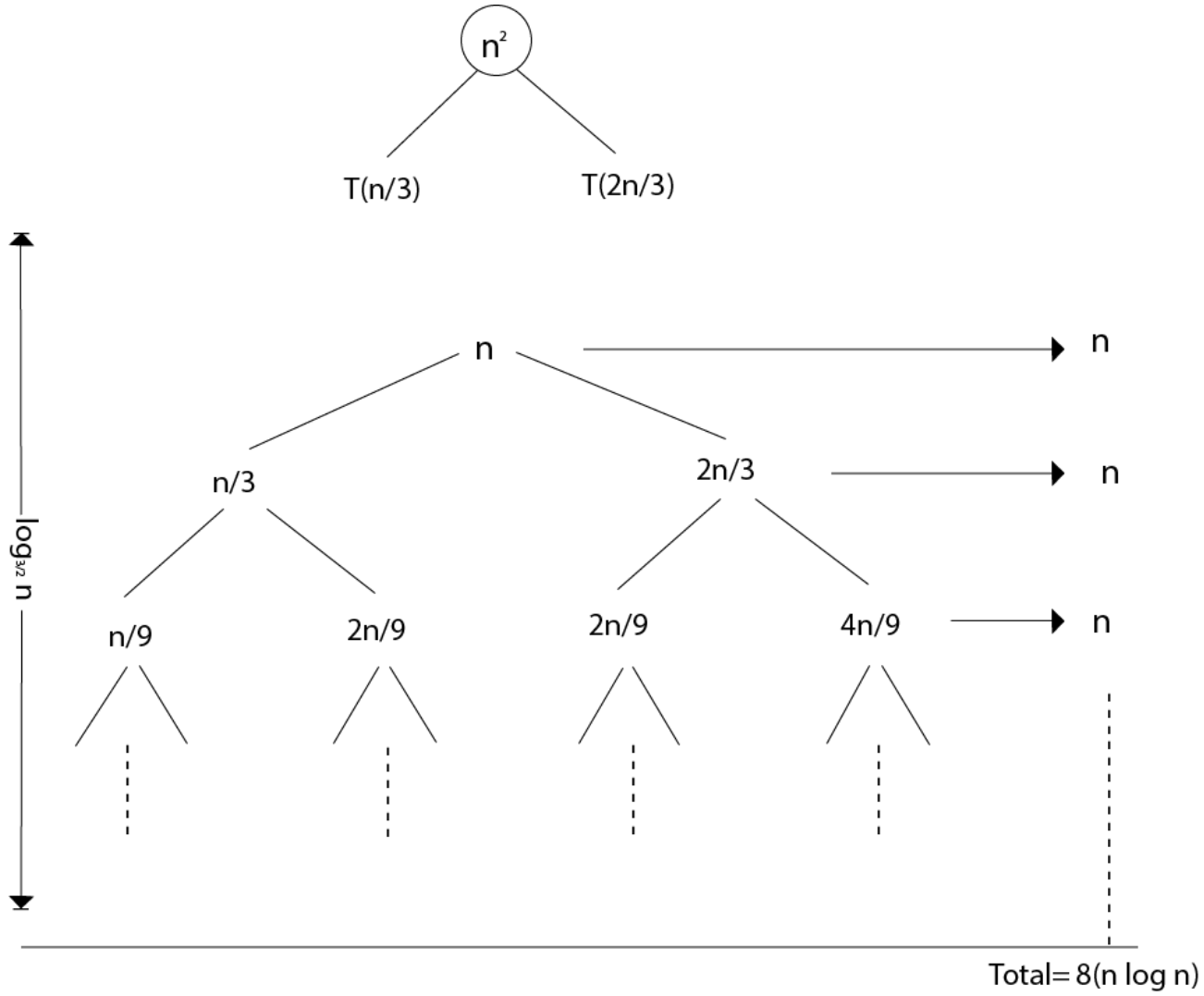
$$\mathbf{T\,(n) = \theta(n^2)}$$

Example 3: Consider the following recurrence

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

Obtain the asymptotic bound using recursion tree method.

Solution: The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots 1$$

Since $\left(\frac{2}{3}\right) n=1$ when $i=\log\frac{3}{2}\ n$.

Thus the height of the tree is $\log\frac{3}{2}\ n$.

$$T(n) = n + n + n + \ldots\ldots + \log\frac{3}{2}\ n \text{ times.} = \boldsymbol{\theta(n \log n)}$$

Problem-02: Solve the following recurrence relation using recursion tree method-

$$T(n) = T(n/5) + T(4n/5) + n$$

Solution-

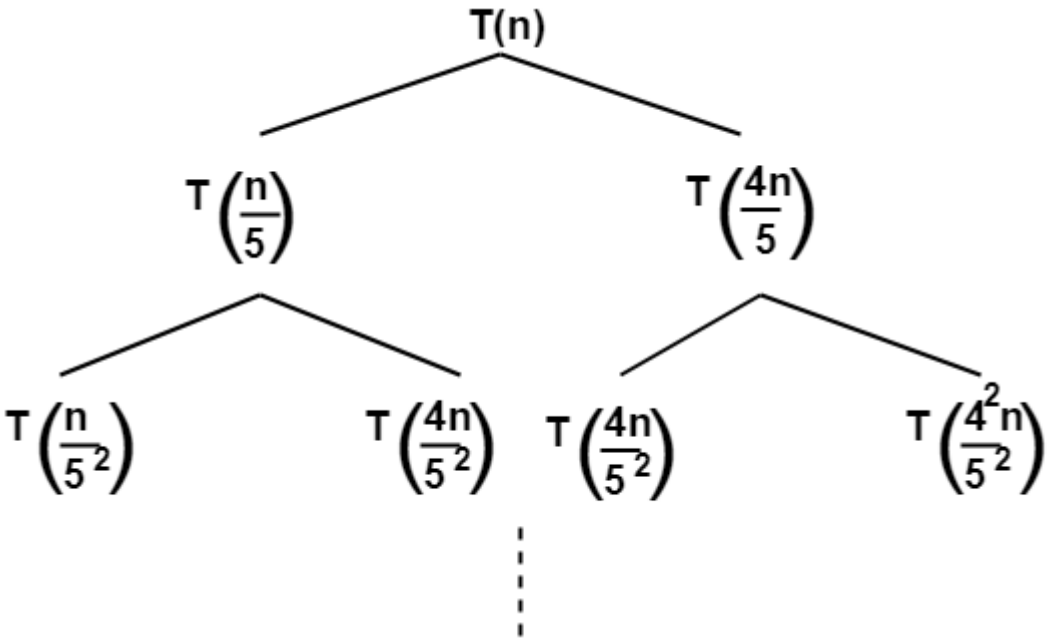
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems- one of size n/5 and another of size 4n/5.
- Then, sub-problem of size n/5 will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size 4n/5 will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

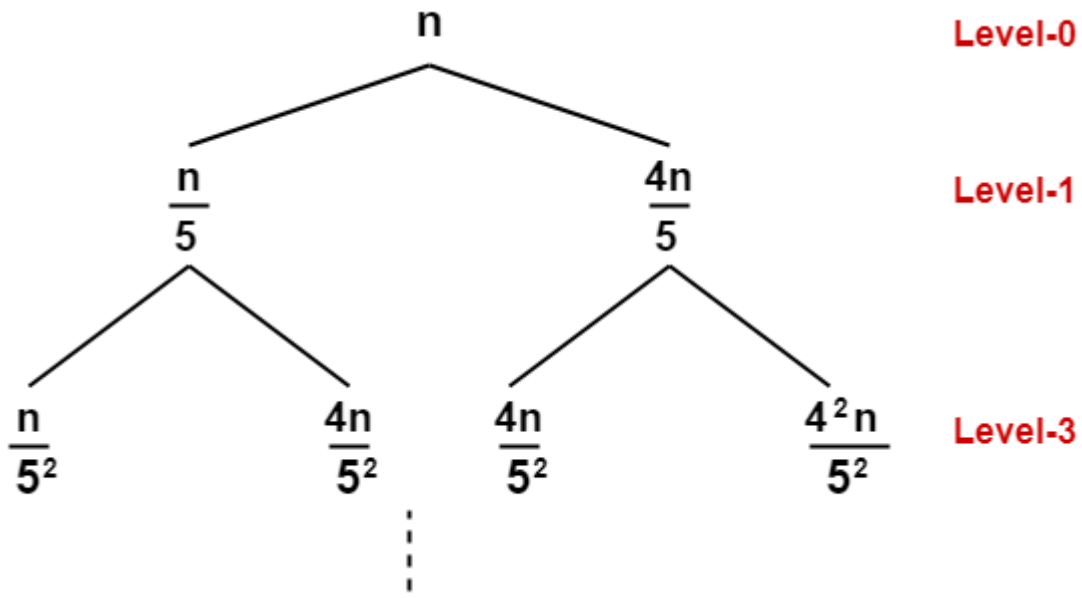
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.
- The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02:

Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = n/5 + 4n/5 = n
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

Step-03:

Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-

- Size of sub-problem at level-0 = $(4/5)^0 n$
- Size of sub-problem at level-1 = $(4/5)^1 n$
- Size of sub-problem at level-2 = $(4/5)^2 n$

Continuing in similar manner, we have-

$$\text{Size of sub-problem at level-i} = (4/5)^i n$$

Suppose at level-x (last level), size of sub-problem becomes 1. Then-

$$(4/5)^x n = 1$$

$$(4/5)^x = 1/n$$

Taking log on both sides, we get-

$$x\log(4/5) = \log(1/n)$$

$$x = \log_{5/4} n$$

∴ Total number of levels in the recursion tree = $\log_{5/4} n + 1$

Step-04:

Determine number of nodes in the last level-

- Level-0 has 2^0 nodes i.e. 1 node
- Level-1 has 2^1 nodes i.e. 2 nodes
- Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level-$\log_{5/4} n$ has $2^{\log_{5/4} n}$ nodes

Step-05:

Determine cost of last level-

Cost of last level = $2^{\log_{5/4} n} \times T(1) = \theta(2^{\log_{5/4} n}) = \theta(n^{\log_{5/4} 2})$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4} n \text{ levels}} + \theta(n^{\log_{5/4} 2})$$

$= n\log_{5/4} n + \theta(n^{\log_{5/4} 2})$

$= \boldsymbol{\theta(n\log_{5/4} n)}$

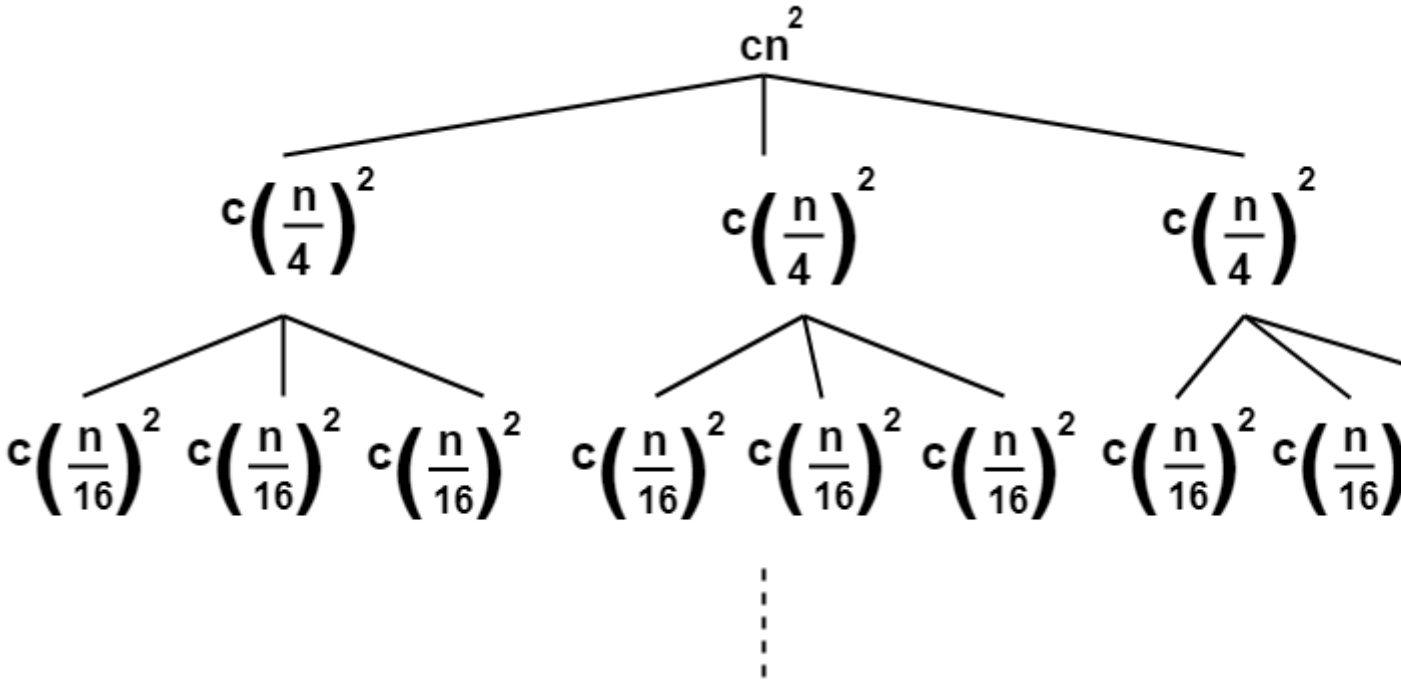
Problem-03: Solve the following recurrence relation using recursion tree method-

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

Solution-

Step-01:

Draw a recursion tree based on the given recurrence relation-



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

Step-02:

Determine cost of each level-

- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-03:

Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have-

$$\text{Size of sub-problem at level-i} = n/4^i$$

Suppose at level-x (last level), size of sub-problem becomes 1. Then-

$$n/4^x = 1$$

$$4^x = n$$

Taking log on both sides, we get-

$$x\log 4 = \log n$$

$$x = \log_4 n$$

∴ Total number of levels in the recursion tree = $\log_4 n + 1$

Step-04:

Determine number of nodes in the last level-

- Level-0 has 3^0 nodes i.e. 1 node
- Level-1 has 3^1 nodes i.e. 3 nodes
- Level-2 has 3^2 nodes i.e. 9 nodes

Continuing in similar manner, we have-

$$\text{Level-}\log_4 n \text{ has } 3^{\log_4 n} \text{ nodes i.e. } n^{\log_4 3} \text{ nodes}$$

Step-05:

Determine cost of last level-

$$\text{Cost of last level} = n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16} cn^2 + \frac{9}{(16)^2} cn^2 + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \} + \theta(n^{\log_4 3})$

Now, $\{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \}$ forms an infinite Geometric progression.

On solving, we get-

$= (16/13)cn^2 \{ 1 - (3/16)^{\log_4 n} \} + \theta(n^{\log_4 3})$

$= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$

$= \mathbf{O(n^2)}$

Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

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

We have to show that it is asymptotically bound by O (log n).

Solution:

For T (n) = O (log n)

We have to show that for some constant c

1. T (n) ≤c logn.

Put this in given Recurrence Equation.

T (n) ≤c log **(n/2)**+ 1

$\leq c \log \left(\frac{n}{2}\right) + 1 = c \log n - c \log_2 2 + 1$

$\leq c \log n$ for $c \geq 1$

Thus **T (n) =O logn**.

Example2 Consider the Recurrence

T (n) = 2T **(n/2)** + n n>1

Find an Asymptotic bound on T.

Solution:

We guess the solution is O (n (logn)).Thus for constant 'c'.
T (n) ≤c n logn
Put this in given Recurrence Equation.
Now,

T (n) ≤2c $\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)$+n
≤cnlogn-cnlog2+n
=cn logn-n (clog2-1)
≤cn logn for (c≥1)
Thus **T (n) = O (n logn)**.

Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example1: Consider the Recurrence

1. T (n) = 1 **if** n=1
2. = 2T (n-1) **if** n>1

Solution:

T (n) = 2T (n-1)
= 2[2T (n-2)] = 2^2T (n-2)
= 4[2T (n-3)] = 2^3T (n-3)
= 8[2T (n-4)] = 2^4T (n-4) (Eq.1)

Repeat the procedure for i times

T (n) = 2^i T (n-i)
Put n-i=1 or i= n-1 in (Eq.1)
T (n) = 2^{n-1} T (1)
= 2^{n-1} .1 {T (1) =1 .....given}
= 2^{n-1}

Example2: Consider the Recurrence

1. T (n) = T (n-1) +1 and T (1) = θ (1).

Solution:

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

= T (n-4) +4 = T (n-5) +1+4
= T (n-5) +5= T (n-k) + k

Where k = n-1

T (n-k) = T (1) = θ (1)

T (n) = θ (1) + (n-1) = 1+n-1=n= θ (n).

e.g. $T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^{3)} +3n$

After k iterations ,$T(n)=2^kT(n/2^k)+kn$----------(1)

Sub problem size is 1 after $n/2^k=1$ => k=logn

So,afterlogn iterations ,the sub-problem size will be 1.

So,when k=logn is put in equation 1

T(n)=nT(1)+nlogn

- nc+nlogn (say c=T(1))
- O(nlogn)

SUBSTITUTION METHOD:

The substitution method comprises of 3 steps

i. Guess the form of the solution
ii. Verify by induction
iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method". This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g.recurrence equation: T(n)=4T(n/2)+n

step 1: guess the form of solution

T(n)=4T(n/2)

- F(n)=4f(n/2)
- F(2n)=4f(n)
- $F(n)=n^2$

So, T(n) is order of n^2 Guess

$T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$

$T(n)=4T(n/2)+n$

$<=4c(n/2)^3+n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 -n)$ is always positive So what we assumed was true.

- $T(n)=O(n^3)$

Step 3: solve for constants

$Cn^3/2-n>=0$

- $n>=1$
- $c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume,$T(k)<=ck^2$

so,we should prove that $T(n)<=cn^2$

$T(n)=4T(n/2)+n$

- $4c(n/2)^2+n$
- cn^2+n

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c_1 k^2-c_2k$, then we can find that $T(n) = O(n^2)$

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Algorithm

```
bubbleSort( A )
    for i = 1 to length[A]
    do
        for j = length [A] downto i+1

         do:
            if (A[j] < A[j-1]
             then swap( A[j], A[j-1] )
```

Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 to length[A] c_1

do for j ← length[A] downto i + 1 c_2

Comparisons: ≈ $n^2/2$ do if A[j] < A[j -1] c_3

Exchanges: ≈ $n^2/2$ then exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n} (n-i+1) + c_3 \sum_{i=1}^{n} (n-i) + c_4 \sum_{i=1}^{n} (n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n} (n-i)$$

$$\text{where } \sum_{i=1}^{n} (n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, $T(n) = \Theta(n^2)$

27

Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Selection Sort

Alg.: SELECTION-SORT(A)

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

8	4	6	9	2	3	1

30

Example :

Example

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8

1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

29

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*)	cost	times
n ← length[*A*]	c_1	1
for j ← 1 to n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if *A*[i] < *A*[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange *A*[j] ↔ *A*[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

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

31

Insertion Sort

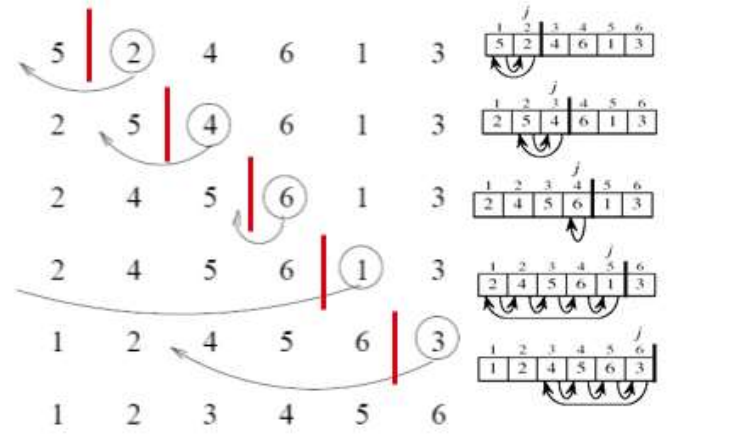
We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of
elements. Insertion sort works the way many people sort a hand of playing cards.

Algorithm

INSERTION-SORT(*A*)

1 for *j* ← 2 to *length*[*A*]	**C1**
2 do *key* ← *A*[*j*]	**C2**
3 ▹ Insert *A*[*j*] into the sorted sequence *A*[1 _ *j* - 1].	**C3**
4 *i* ← *j* – 1	**C4**
5 while *i* > 0 and *A*[*i*] > *key*	**C5**
6 do *A*[*i* + 1] ← *A*[*i*]	**C6**
7 *i* ← *i* – 1	**C7**
8 *A*[*i* + 1] ← *key*	**C8**

Insertion Sort



12

Analysis of Insertion Sort

INSERTION-SORT(A)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

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

19

For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $tj = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$T(n) = c1n + c2(n - 1) + c4(n - 1) + c5(n - 1) + c8(n - 1)$
$= (c1 + c2 + c4 + c5 + c8)n - (c2+ c4 + c5 + c8)$.

This running time can be expressed as $an + b$ for *constants* a and b that depend on the statement costs ci ; it is thus a ***linear function*** of n.

Worst Case

If the array is in reverse sorted order-that is, in decreasing order-the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \_ j - 1]$, and so $tj = j$ for $j = 2, 3, \ldots, n$. Noting that

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

And

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

the running time of INSERTION-SORT is

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

This worst-case running time can be expressed as $an2 + bn + c$ for constants a, b, and c that again depend on the statement costs ci ; it is thus a ***quadratic function*** of n.

Divide and Conquer approach

The divide-and-conquer paradigm involves three steps at each level of the recursion:

• **Divide** the problem into a number of subproblems.
• **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
• **Combine** the solutions to the subproblems into the solution for the original problem.

Merge Sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numberswhich we will assume is stored in an array A [1...n]. Theobjective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

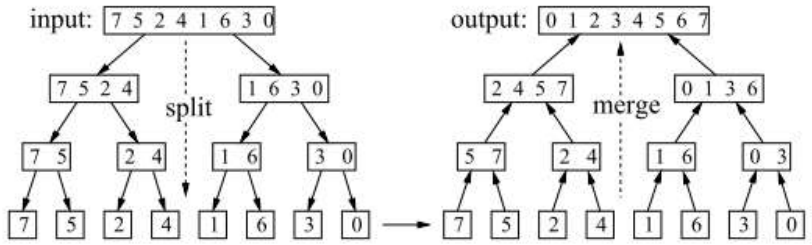
How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 .

Conquer: Sort each subsequence (by calling MergeSort recursively on each).
Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage,which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The "divide" phase is shown on the left. It works top-down splitting up the list into smaller sublists. The "conquer and combine" phases areshown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We'll assume that the procedure that merges two sorted list is available to us.

Algorithm

The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[7]

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

```
MERGE(A, p, q, r)
1 n1 ← q - p + 1
2 n2 ← r - q
3 create arrays L[1 _ n1 + 1] and R[1 _ n2 + 1]
4 for i ← 1 to n1
5    do L[i] ← A[p + i - 1]
6 for j ← 1 to n2
7    do R[j] ← A[q + j]
8 L[n1 + 1] ← ∞
9 R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
```

```
12 for k ← p to r
13    do if L[i] ≤ R[j]
14          then A[k] ← L[i]
15                  i ← i + 1
16          else A[k] ← R[j]
17                  j ← j + 1
```

Analysis of Merge Sort

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

• **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

• **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort

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

By using master method $T(n) = \theta(n \log n)$.

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. Quicksort, like merge sort, is based on the divide-and-conquer paradigm.

The following procedure implements quicksort.

QUICKSORT(*A*, *p*, *r*)

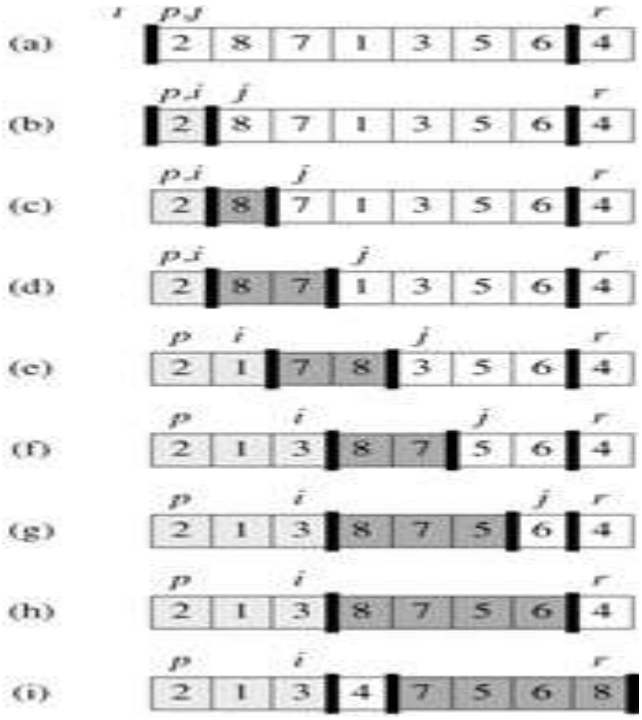
```
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q - 1)
4         QUICKSORT(A, q + 1, r)
```

To sort an entire array *A*, the initial call is QUICKSORT(*A*, 1, *length*[*A*]).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray *A*[*p*.. *r*] in place.

PARTITION(A, p, r)
1 $x \leftarrow A[r]$
2 $i \leftarrow p - 1$
3 for $j \leftarrow p$ **to** $r - 1$
4 do if $A[j] \leq x$
5 then $i \leftarrow i + 1$
6 exchange $A[i] \leftrightarrow A[j]$
7 exchange $A[i + 1] \leftrightarrow A[r]$
8 return $i + 1$



Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements.

Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

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

After solving we get T(n) = $\Theta(n^2)$.

Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n2)$ running time occurs when the input array is already completely sorted-a common situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

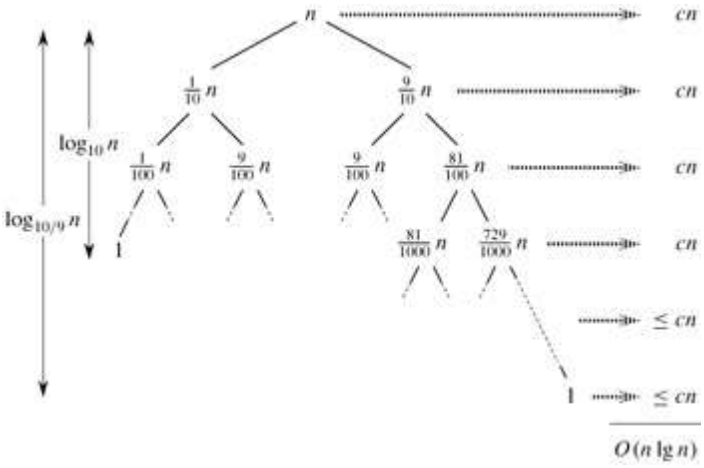
In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then
$T(n) \leq 2T(n/2) + \Theta(n)$,
which by case 2 of the master theorem has the solution $T(n) = O(n \lg n)$.

Balanced partitioning
The average-case running time of quicksort is much closer to the best case.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$T(n) \leq T(9n/10) + T(n/10) + cn$

on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term. Notice that every level of the tree has cost cn, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most cn. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is therefore $O(n \lg n)$.



recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. T (n) = $O(n \lg n)$.

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible .

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Maintaining the Heap Property: Heapify is a procedure for manipulating heap Data Structure. It is given an array A and index I into the array. The subtree rooted at the children of A [i] are heap but node A [i] itself may probably violate the heap property i.e. A [i] < A [2i] or A [2i+1]. The procedure 'Heapify' manipulates the tree rooted as A [i] so it becomes a heap.

MAX-HEAPIFY (A, i)

1. l ← left [i]
2. r ← right [i]
3. if l≤ heap-size [A] and A[l] > A [i]
4. then largest ← l
5. Else largest ← i
6. If r≤ heap-size [A] and A [r] > A[largest]
7. Then largest ← r
8. If largest ≠ i
9. Then exchange A [i] A [largest]
10. MAX-HEAPIFY (A, largest)

Analysis:

The maximum levels an element could move up are Θ (log n) levels. At each level, we do simple comparison which O (1). The total time for heapify is thus O (log n).

Building a Heap:

BUILDHEAP (array A, int n)

1 for i ← n/2 down to 1

2 do

3 HEAPIFY (A, i, n)

HEAP-SORT ALGORITHM:

HEAP-SORT (A)

1. BUILD-MAX-HEAP (A)
2. For I ← length[A] down to Z
3. Do exchange A [1] ←→ A [i]
4. Heap-size [A] ← heap-size [A]-1
5. MAX-HEAPIFY (A,1)

Analysis: Build max-heap takes O (n) running time. The Heap Sort algorithm makes a call to 'Build Max-Heap' which we take O (n) time & each of the (n-1) calls to Max-heap to fix up a new heap. We know 'Max-Heapify' takes time O (log n)

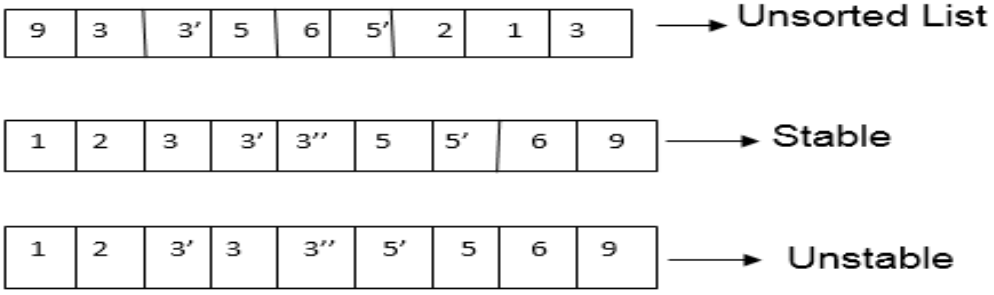
The total running time of Heap-Sort is O (n log n).

Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Some Sorting Algorithm is stable by nature like Insertion Sort, Merge Sort and Bubble Sort etc. Sorting Algorithm is not stable like Quick Sort, Heap Sort etc.

A Stable Sort is one which preserves the original order of input set, where the comparison algorithm does not distinguish between two or more items. A Stable Sort will guarantee that the original order of data having the same rank is preserved in the output.



Linear Time Sorting

We have sorting algorithms that can sort "n" numbers in O (n log n) time. Merge Sort and Heap Sort achieve this upper bound in the worst case, and Quick Sort achieves this on Average Case.

Merge Sort, Quick Sort and Heap Sort algorithm share an interesting property: the sorted order they determined is based only on comparisons between the input elements. We call such a sorting algorithm **"Comparison Sort"**.

There is some algorithm that runs faster and takes linear time such as Counting Sort, Radix Sort, and Bucket Sort but they require the special assumption about the input sequence to sort.
Counting Sort and Radix Sort assumes that the input consists of an integer in a small range.
Bucket Sort assumes that a random process that distributes elements uniformly over the interval generates the input.

Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, forsome integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time.\

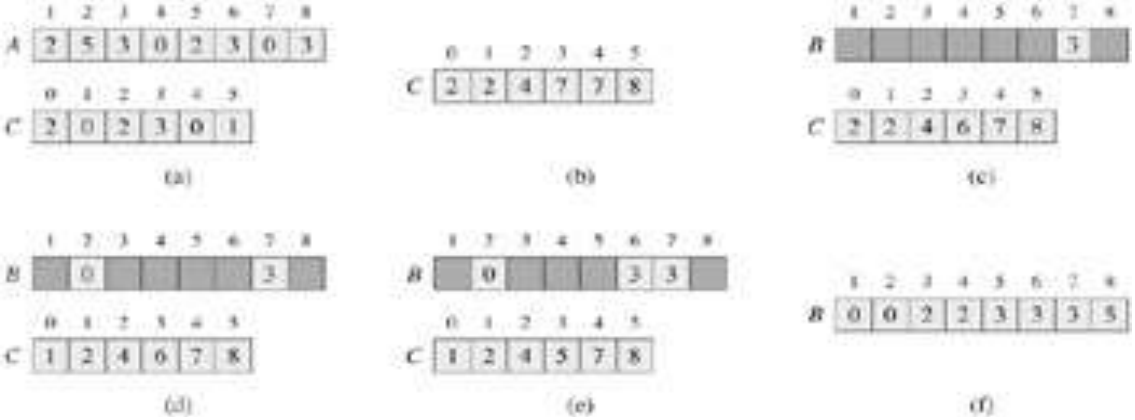
In the code for counting sort, we assume that the input is an array $A[1 \_ n]$, and thus $length[A] = n$. We require two other arrays: the array $B[1 \_ n]$ holds the sorted output, and the array $C[0 \_ k]$ provides temporary working storage.

COUNTING-SORT(A, B, k)

1 **for** $i \leftarrow 0$ **to** k
2 **do** $C[i] \leftarrow 0$
3 **for** $j \leftarrow 1$ **to** $length[A]$
4 **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5 ▷ $C[i]$ now contains the number of elements equal to i.
6 **for** $i \leftarrow 1$ **to** k
7 **do** $C[i] \leftarrow C[i] + C[i - 1]$
8 ▷ $C[i]$ now contains the number of elements less than or equal to i.
9 **for** $j \leftarrow length[A]$ **downto** 1
10 **do** $B[C[A[j]]] \leftarrow A[j]$
11 $\boldsymbol{C[A[j]] \leftarrow C[A[j]] - 1}$

Analysis of Running Time: Overall time is **θ(k+n)** time.

- For a loop of step 1 to 2 take θ(k) times
- For a loop of step 3 to 4 take θ(n) times
- For a loop of step 6 to 7 take θ(k) times
- For a loop of step 9 to 11 take θ(n) times



Bucket Sort

Bucket Sort runs in linear time on average. Like Counting Sort, bucket Sort is fast because it considers something about the input. Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the interval [0,1].

To sort n input numbers, Bucket Sort

1. Partition μ into n non-overlapping intervals called buckets.
2. Puts each input number into its buckets
3. Sort each bucket using a simple algorithm, e.g. Insertion Sort and then
4. Concatenates the sorted lists.

Bucket Sort considers that the input is an n element array A and that each element A [i] in the array satisfies $0 \leq A[i] < 1$. The code depends upon an auxiliary array B [0....n-1] of linked lists (buckets) and considers that there is a mechanism for maintaining such lists.

BUCKET-SORT (A)

1. n ← length [A]
2. for i ← 1 to n
3. do insert A [i] into list B [n A[i]]
4. for i ← 0 to n-1
5. do sort list B [i] with insertion sort.
6. Concatenate the lists B [0], B [1] ...B [n-1] together in order.

Example: Illustrate the operation of BUCKET-SORT on the array. A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 068)

Solution:

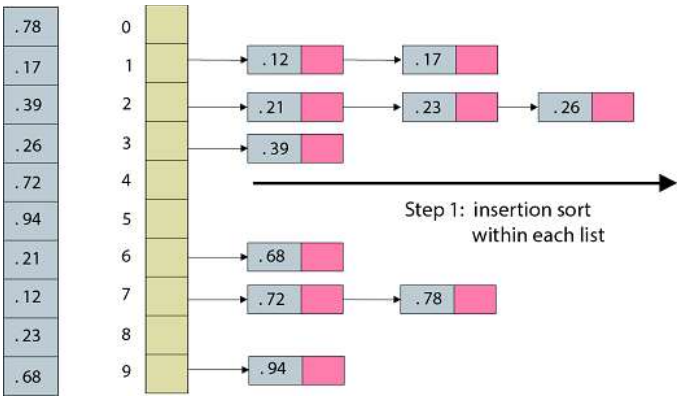


Fig: Bucket sort: step 1, placing keys in bins in sorted order

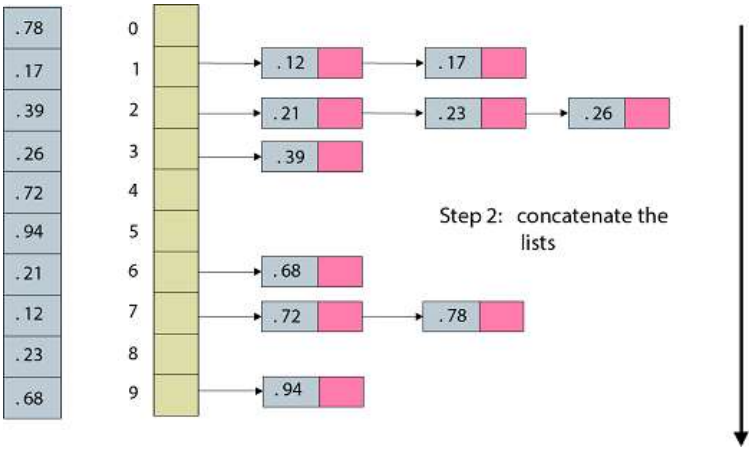


Fig: Bucket sort: step 2, concatenate the lists

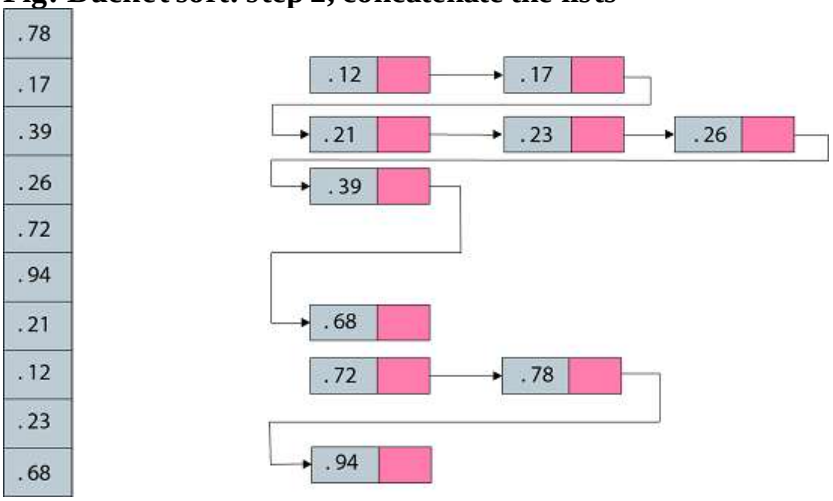


Fig: Bucket sort: the final sorted sequence

Radix Sort

Radix Sort is a Sorting algorithm that is useful when there is a constant'd' such that all keys are d digit numbers. To execute Radix Sort, for p =1 towards 'd' sort the numbers with respect to the Pth digits from the right using any linear time stable sort. The Code for 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.

Here is the algorithm that sorts A [1.n] where each number is d digits long.

Algorithm

RADIX-SORT (array A, int n, int d)

1 for i ← 1 to d

2 do stably sort A to sort array A on digit i

Example: The first Column is the input. The remaining Column shows the list after successive sorts on increasingly significant digit position. The vertical arrows indicate the digits position sorted on to produce each list from the previous one.

1. 576 49[4] 9[5]4 [1]76 176
2. 494 19[4] 5[7]6 [1]94 194
3. 194 95[4] 1[7]6 [2]78 278
4. 296 → 57[6] → 2[7]8 → [2]96 → 296
5. 278 29[6] 4[9]4 [4]94 494
6. 176 17[6] 1[9]4 [5]76 576
7. 954 27[8] 2[9]6 [9]54 954