1 Asymptotic sufficiently Analysis

Introduction

- An Algorithm is a combination of a sequence of finite steps to solve a problem. All steps should be finite and compulsory (If you are removing any one step, then the algorithm will be affected).
- Which algorithm is best suitable for a given application depends on the input size, the architecture of the computer, and the kind of storage to be used: disks, main memory, or even tapes.
- It should terminate after a finite time and produce one or more outputs by taking zero or more inputs.

Need For Analysis

- Problem may have many candidate solution, the majority of which do not solve the problem in hand. Finding solution that does, or one that is 'best', can be quite a challenge.
- While running a program, how much time it will take and how much space it will occupy is known as analysis of program or analysis of Algorithm.
- An algorithm analysis mostly depends on time and space complexity.
  Generally, we perform following types of analysis
  - **Worst case:**
    The maximum number of steps or resources required to solve a problem.
  - **Best case:**
    The minimum number of steps or resources required to solve a problem.
  - **Average case:**
    The average number of steps or resources required to solve a problem.
  - **Amortized:**
    A sequence of operations applied on the input of size n averaged over time.
- The order of growth of running time curve that varies with size of input helps us to know the efficiency of an algorithm and we can compare the relative performance of an alternative algorithm.
- We study asymptotic efficiency of an algorithm to know how the running time of an algorithm increases with the size of input.

Asymptotic Notations

- We use asymptotic notation primarily to describe the running times of algorithms.
- With the help of Asymptotic notation, we compare two positive functions in terms of running time.

θ-Notation

Let f(n) and g(n) be two positive functions

$f(n) = \theta(g(n))$

if and only if

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

and

$f(n) \geq c_2 . g(n)$

$\forall\ n \geq n_0$

such that there exists three positive constant $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 1$

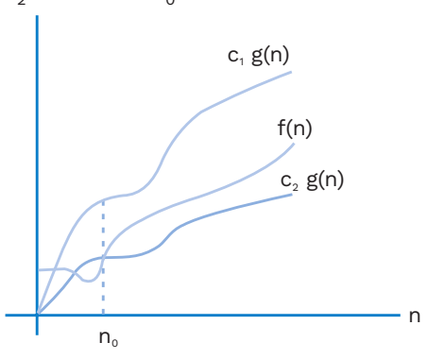


Fig. 1.2 (A)

- $f(n) \leq c_1 . g(n)$ and $f(n) \geq c_2 . g(n)$, $\forall n$, $n \geq n_0$

O-Notation [Pronounced "big-oh"]

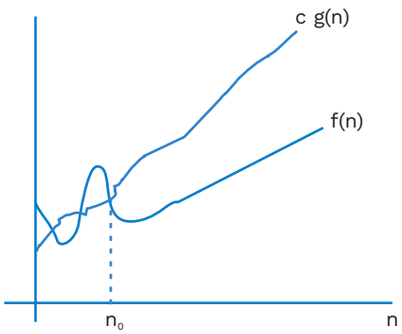
Let f(n) and g(n) be two positive functions

$f(n) = O(g(n))$

if and only if

$f(n) \leq c \cdot g(n), \forall n, \geq n_0$

such that $\exists$ two positive constants $c > 0$, $n_0 \geq 1$.



f(n) = O(g(n))

Fig. 1.2 (B)

e.g.,

f(n) = 3n + 2, g(n) = n

f(n) = O(g(n))

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

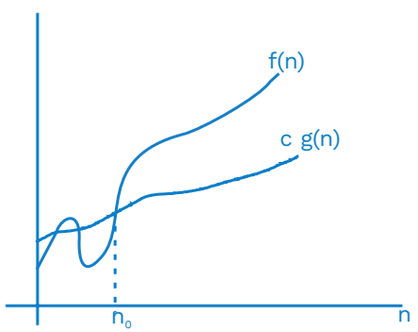
$3n + 2 \leq cn$

$c = 4$

$n_0 \geq 2$

Ω-Notation: [Pronounced "big-omega"]

- Ω notation provides an asymptotic lower bound for a given function g(n), denoted by Ω(g(n)). The set of functions
  $f(n) = \Omega(g(n))$
  if and only if
  $f(n) \geq c \cdot g(n), \forall n \geq n_0$
  such that $\exists$ two positive constants $c > 0$, $n_0 \geq 1$.



$f(n) = \Omega\ (g(n))$

Fig. 1.2 (C)

Theorem to Remember

Two functions, f(n) and g(n) are equivalent if and only if f(n) = O(g(n)) and f(n) = Ω(g(n))

o-Notation [Pronounced "little-oh"]

For a function f(n), we denote o(g(n)) as the set of functions

f(n) = o(g(n))

iff

$f(n) < c \cdot g(n), \forall c > 0$ whenever $n \geq n_0$

$n_0 \geq 1$.

ω-notation: [Pronounced 'little-omega"]

- By analogy, ω-notation is to Ω-notation as o-notation is to O-notation.
  For a function f(n), we denote ω(g(n)) as the set of functions.
  $f(n) = \omega(g(n))$
  if and only if
  $f(n) > c \cdot g(n), \forall c > 0$ whenever $n \geq n_0$
  $n_0 \geq 1$.

Some of the important properties of asymptotic notation are:

Transitivity:

- If $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$ then $f(n) = \theta(h(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
- If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$
- If $f(n) = o(g(n))$ and $g(n) = o(h(n))$ then $f(n) = o(h(n))$
- If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ then $f(n) = \omega(h(n))$

Reflexivity:

- $f(n) = \theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

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 $g(n) = \omega(f(n))$

Rack Your Brain

Consider the following c program

```
unacademy(int n)
{
  int n = 2^2^k
  for(i=1, i< = n, i++)
  {
    j = 2;
    while(j < = n)
    {
      j = j^2;
      printf("prepladder");
    }
  }
}
```

Express number of times prepladder is printed using O notation.

Comparison Function

1. Which is asymptotically larger: log(log*n) or log* (logn)

Note:
log*n (read "log star" of n) to denote the iterated logarithm

- The growth of iterated logarithm function is very slow.
  log*2 = 1,
  log*4 = 2,
  log*16 = 3,
  log*65536 = 4,
  log* (2^{65536}) = 5

$$\Rightarrow \lim_{n\to\infty} \frac{\log(\log^* n)}{\log^* (\log n)}$$

$$= \lim_{n\to\infty} \frac{\log(\log^* 2^n)}{\log^* (\log 2^n)}$$

[we have $\log^* 2^n = 1 + \log^* n$]

$$= \lim_{n\to\infty} \frac{\log(1+\log^* n)}{\log^* n}$$

$$= \lim_{n\to\infty} \frac{\log(1+n)}{n}$$

$$= \lim_{n\to\infty} \frac{1}{1+n}$$

[$\therefore$ We have log*(logn) = 0 is asymptotically larger]

2. Table below, gives whether A is O, o, Ω, ω or θ of B. Assume $k \geq 1$, $\varepsilon > 0$ and $C > 1$ are constants.

A	B	O	o	Ω	ω	θ
$\log^k n$	n^ε	w	Yes	No	No	No
n^k	C^n	Yes	Yes	No	No	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No	No	No
2^n	$2^{n/2}$	No	No	Yes	Yes	No
$n^{\log c}$	$C^{\log n}$	Yes	No	Yes	No	Yes
log(n!)	$\log(n^n)$	Yes	No	Yes	No	Yes

Note:

- log(n!) = θ(nlogn) (by string's approximation)
- Fibonacci numbers are co-related to the golden ratio ϕ and to its conjugate $\hat{\phi}$, which are two roots of the equation $x^2 = x + 1$
  and are given by the following formulas

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6183 \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -.61803$$

Specifically, we have

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

Where
$F_0 = 0$
$F_1 = 1$
$F_i = F_{i-1} + F_{i-2}$
in Fibonacci series.

Note:
In exam do not apply logarithm blindly, first cancel the common terms then apply logarithm.

Example 1:

Determine which is faster growing function between 2^n and n^2

Solution:
Apply log
$\log 2^n$, $\log n^2$
nlog2, 2logn

Substitute large value in n(eg, n = 2^{100})
2^{100} log2, 2log2^{100}
2^{100} , 100*2
$\therefore$ 2^n grows faster than n^2

Example 2:

Determine which is faster growing function between $n^{\sqrt{n}}$ and $n^{\log n}$

Solution:

Apply log

$\sqrt{n}$ logn, logn logn

Substitute n = 2^{16}

$\sqrt{2^{16}}$ log 2^{16}, log 2^{16} log 2^{16}

2^8 * 16 , 16 * 16

$\therefore$ $n^{\sqrt{n}}$ grows faster than $n^{\log n}$

4. Following are some of the functions in increasing order of asymptotic (big-O) growth.
4loglogn < 4logn < $n^{1/2}$log^4(n) < 5n < n^4 < $(\log n)^{5\log n}$ < $n^{\log n}$ < $n^{n/8}$ < 5^n < 5^{5n} < $(n/4)^{n/4}$ < $n^{n/4}$ < 4^{n^4} < 4^{4^n} < 5^{5^n}

Verify the above given order by applying logarithm and substituting large value in 'n'.

Rack Your Brain

Arrange the following functions in asymptotic decreasing order.

- $f_1 = 2^{\log 2^n}$
- $f_2 = n^{2024/16}$
- f_3 = nlogn
- $f_4 = n^{\log}$

Recurrences

Recurrences go hand in hand with the divide and conquer paradigm, because they give us a natural way to differentiate the running times of divide and conquer algorithms.

Definitions

Same Recursive program written in the form of a mathematical relation is called Recurrence relation.

Example: T(n) = T(2n/3) + T(n/3) + θ(n)

There are three methods of solving a recurrence relation:

1. Substitution Method
2. Recursive - Tree Method
3. Master's theorem

a) Substitution method
Guess a bound and use mathematical induction method to prove if the guess is correct.

b) Recursion (Recursive) - tree method
When more than 1 recursive call is present in given recurrence relation then we go for the recursive tree method.

c) Master theorem:
If any recurrence relation is there in the form of:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where a and b are positive constant and a $\geq$ 1 and b > 1

The substitution method for solving recurrences.

The substitution method to solve recurrences comprises two steps:

1. Write (Guess) the asymptotic (upper or lower) bound on the Recursive Program.
2. Writing a recurrence relation is nothing but writing the Recursive program in the form of Mathematical relation .

For Example:

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

Let's guess that the solution is T(n) = O(nlogn)

↳ The substitution method requires us to prove

T(n) $\leq$ c nlogn

Let's start by assuming that this bound holds for all positive m < n, in particular for m = $\lfloor n/2 \rfloor$,

$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$

Substituting into the recurrence yields

$T(n) \le 2\left(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)\right) + n$

$\le cn\log(n/2) + n$

= cnlogn – cnlog2 + n

≤ cnlogn, step holds as long as c ≥ 1.

Examples:

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

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

[T(n – k) becomes 1 for k = n – 1]

= T(n-n+1) + (n-n+1+1) + 2 + 3 + ... +n
= T(1) + 2 + 3 + ... + n = 1 + 2 + 3 + ... +n
= $\frac{n(n+1)}{2}$

$O(n^2)$

2.
```
function ( )
 {
     if(n>1)
  return(function(n-1))
 }
```

Above function can be written as equation form as

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

T(n) = 1 + 1 + T(n-2)
= 2 + T(n-2)
= 3 + T(n-3)
= 4 + T(n-4)
⋮
⋮
= K + T (n-k)

T(n-k) will eventually become 1 when k = n – 1,

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

∴ T(n) = O(n)

Note:
Sometimes, a small algebraic manipulation can make an unknown recurrence similar to a familiar one.

Example: $T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \log n$

which looks difficult

- We can simplify this recurrence, by changing the variables.
- For convenience, renaming m = logn yields

  $T(2^m) = 2T(2^{m/2}) + m$

  // Lets say, $T(2^m) = S(m)$ to produce the new recurrence

  $S(m) = 2S\ (m/2) + m$

  // s(m) = O(mlogm)
- After back substitution from s(m) to T(n), we obtain
  $T(n) = T(2^m) = S(m) = O(m\log m)$
  = O(logn log logn)

Rack Your Brain

Consider the recurrence relation

$T(n) = \sqrt{n}\ T(\sqrt{n}) + 100n$

express T(n) in θ notation

The Recursive (recursion) tree method for solving recurrences:

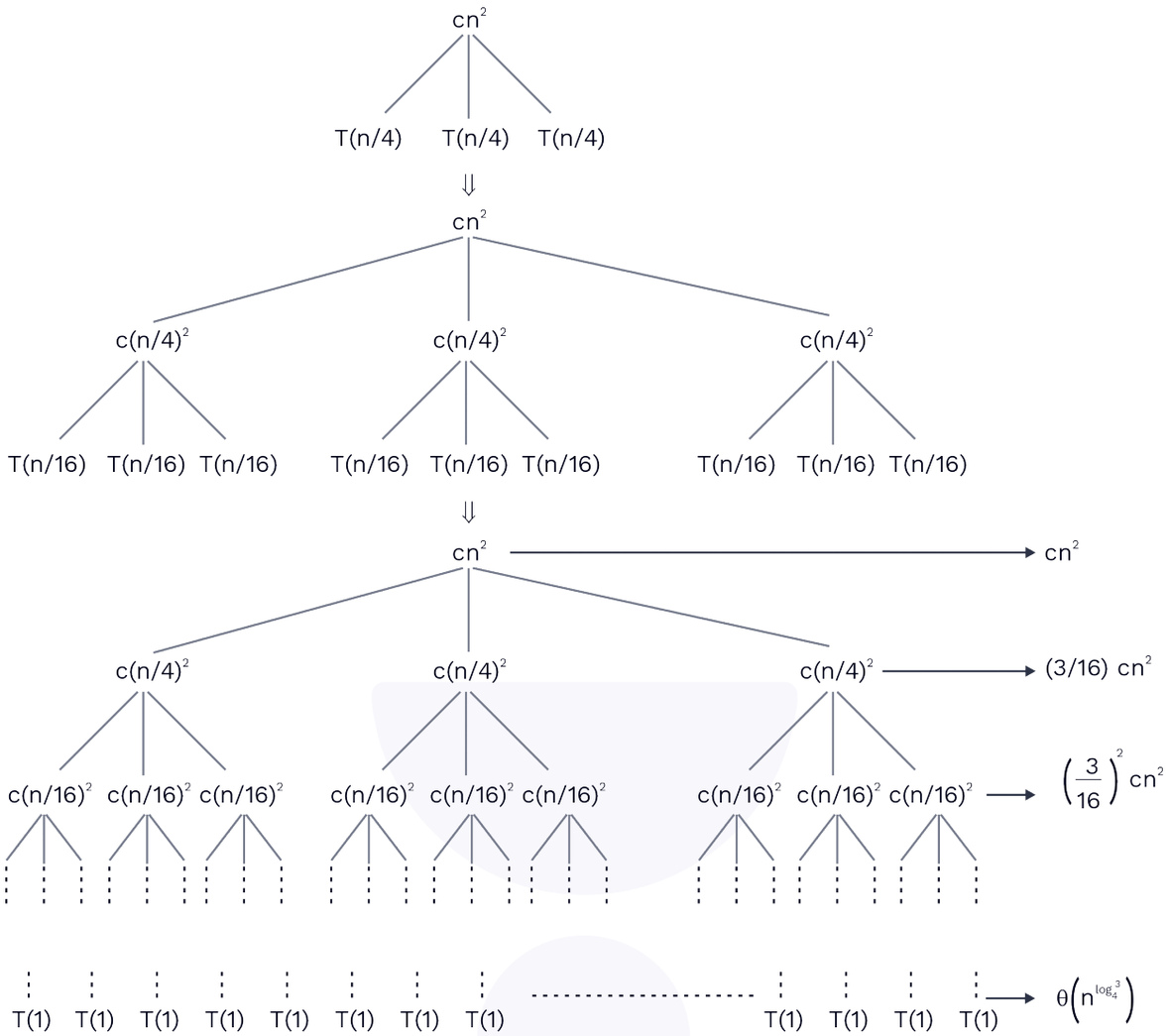
- Recursive tree method produces N-ary tree. (Where n is the recursive function call).
- Then we calculate recursive terms at each level of N-ary tree.

Examples:

i) $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

or

$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + cn^2$



- At every level, subproblem size decreases by factor 4, and we must go up to the boundary condition.
- The subproblem size for node at depth 'i' is $(n/4^i)$.

  At every level, calculate the non recursive terms.
- At each level, there exists three function calls.
- At depth 'i' ,every node has a cost of $c\left(\frac{n}{4^i}\right)^2$.
- Multiplying, we can see that the total cost over all nodes at depth i, for i = 0, 1, 2, ..., $\log_4 n-1$, is $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$
- The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost T(1), for a total cost of $n^{\log_4 3} T(1)$, which is $\theta\left(n^{\log_4 3}\right)$, since we assume that T(1) is a constant.
- To find the total complexity , we find the cost at each level and sum all the costs at each level.

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

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

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

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

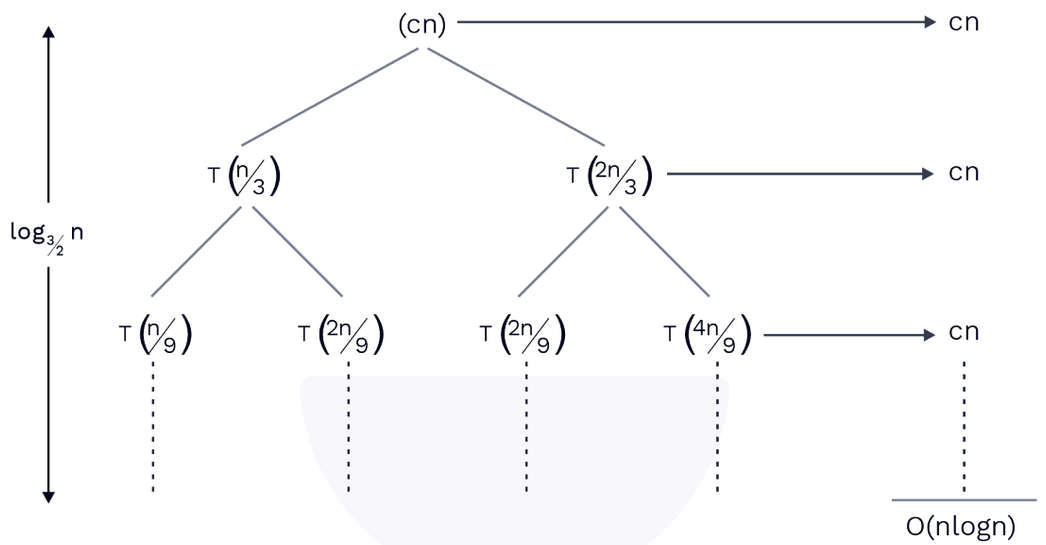
$$= \frac{16}{13} cn^2 + \theta\left(n^{\log_4 3}\right)$$

$$= O(n^2)$$

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

Example:

ii) $T(n) = T\left(n/3\right) + T\left(2n/3\right) + O(n)$



- The largest simple path from the root to a leaf is

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

  When $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

- By intuition, we expect the solution to the recurrence to be at most the number of levels multiplied with the cost of each level, or $O(cn\log_{3/2} n) = O(n\log n)$

  Not every level in the tree contributes to a cost of cn.

- Indeed, we can use the substitution method to verify that O(nlogn) is an upper bound for the solution to the recurrence. We have

  $$T(n) \le T(n/3) + T(2n/3) + cn$$

  $$\le d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn$$

  Where d is a suitable positive constant.

  $$= \left(d\left(n/3\right)\log n - d\left(n/3\right)\log 3\right)$$

  $$+ \left(d\left(2n/3\right)\log n - d\left(2n/3\right)\log\left(3/2\right)\right) + cn$$

  $$= dn\log n - d\left(\left(n/3\right)\log 3 + \left(2n/3\right)\log\left(3/2\right)\right) + cn$$

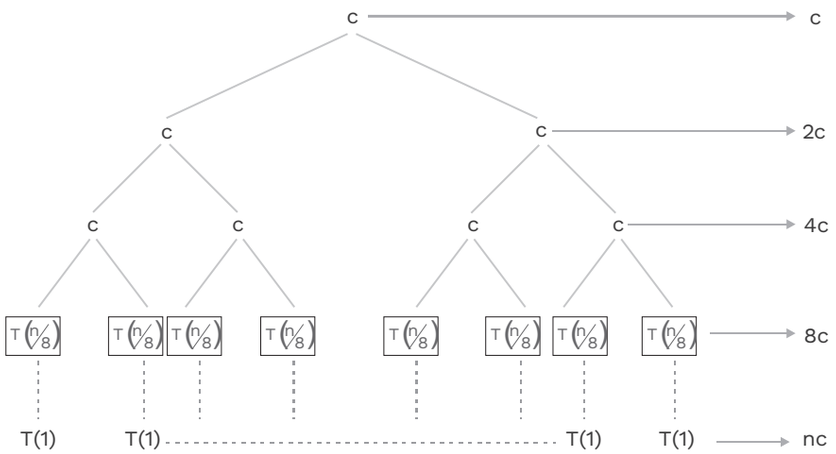
  $$= dn\log n - d\begin{pmatrix}\left(n/3\right)\log 3 + \left(2n/3\right)\log 3 \\ -\left(2n/3\right)\log 2\end{pmatrix} + cn$$

  $$= dn\log n - dn\left(\log 3 - 2/3\log 2\right) + cn$$

  $$\le dn\log n, \text{ as long as } d \ge c/\left(\log 3 - \left(2/3\right)\log 2\right)$$

Example:

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



- The total cost over all nodes at depth i for i = 0, 1, 2,..., logn-1, is $2^i c$

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

$$= c + 2c + 4c + ...nc$$

$$= c(1 + 2 + 4 + ... + n) \text{ assume } (n = 2^k)$$

$$= c(1 + 2 + 4 + ... + 2^k)$$

$$= c(1 + 2^1 + 2^2 + ... + 2^k)$$

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

$$= c(2^{k+1} - 1)$$

$$= c(2n - 1)$$

$$= O(n)$$

Rack Your Brain

Consider the recurrence relation
T(n) = T(n / 2) + 2T(n / 5) + T(n / 10) + 4n
express T(n) in 'big-oh' (O) notation

The master's theorem procedure for solving recurrences:

The master theorem is used to solve recurrence relation if and only if it is in the form of

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad ...(i)$$

Where a ≥ 1 and b > 1 are positive constant and f(n) is a positive function.

- The recurrence (1) describes the running time of an algorithm that divides into 'a' subproblems where each is restricted by size $\frac{n}{b}$.
- The 'a' subproblems are solved recursively, each in time $T\left(\frac{n}{b}\right)$.
- The function f(n) covers the cost of dividing the problem and combining the results of the subproblems.

Master's theorem:

Let f(n) is a positive function and T (n) is defined recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Where a >= 1 and b > 1 are two positive constants.

Then T(n) follows asymptotic bounds:

Case 1:

If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$

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

Case 2:

If $f(n) = \theta\left(n^{\log_b a}\right)$, then $T(n) = \theta\left(n^{\log_b a} \log n\right)$

Case 3:

If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$,

and if $a f\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$

and all sufficiently large n, then

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

- In case 1, the function $n^{\log_b a}$ is larger, then the solution is given by
  $T(n) = \theta\left(n^{\log_b a}\right)$.
- In case 3, if function f(n) is larger, then the solution is $T(n) = \theta(f(n))$.
- In case 2, if two functions are the same size, then multiply by a logarithmic factor, and the solution is $T(n) = \theta\left(n^{\log_b a} \log n\right)$
  $= \theta\left(f(n) \log n\right)$

Note:
The all three cases of Master's Theorem does not cover all the possibilities of f(n):

- There is a gap between cases 1 and 2 when f(n) is smaller than nlogba but not polynomially smaller.
- Similarly, there is a gap between cases 2 and 3 when f(n) is larger than nlogba but not polynomially larger.
- If the function f(n) falls into one of these gaps, you cannot use the master method to solve the recurrence.

Examples:

1. $T(n) = 9T\left(\frac{n}{3}\right) + n$

   For this recursion, we have a = 9, b = 3, f(n) = n, and thus we have that
   $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$.

   Since $f(n) = O\left(n^{\log_3 9 - \epsilon}\right)$. Where $\epsilon = 1$, we can apply case 1 of master theorem.

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

2. $T(n) = T\left(\frac{2n}{3}\right) + 1$

   $a = 1, b = \frac{3}{2}$, $f(n) = 1$

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

   Case 2 of master theorem applies,
   since $f(n) = \theta\left(n^{\log_b a}\right) = \theta(1)$

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

3. $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

   we get a = 3, b = 4, f(n) = nlogn, and
   $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

   Since $f(n) = \Omega\left(n^{\log_4 3 + \epsilon}\right)$, where $\epsilon \approx 0.2$.

   Case 3 master theorem applies,
   For sufficiently large n, we have that
   $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \log n$
   $= cf(n)$ for $c = \frac{3}{4}$.
   Consequently, by case 3
   $T(n) = \theta(n \log n)$

4. $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$,

   Even though it appears to have proper form a = 2, b = 2, f(n) = nlogn, and $n^{\log_b a} = n$.
   This is not polynomial larger so this does not belong to case 3
   $f(n) / n^{\log_b a} = (n \log n) / n = \log n$
   is asymptotically less than n^{ϵ} for any positive constant ϵ.
   Consequently, the recurrence falls into the gap between case 2 and case 3.

5. $T(n) = 8T\left(\frac{n}{2}\right) + \theta\left(n^2\right)$

   a = 8, b = 2, and $f(n) = \theta(n^2)$,
   $n^{\log_b a} = n^{\log_2 8} = n^3$
   since n^3 is polynomially larger than f(n)
   i.e $f(n) = O(n^{3-\epsilon})$ for $\epsilon = 1$
   case 1 applies
   $T(n) = \theta(n^3)$

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

   a = 7, b = 2, $f(n) = \theta(n^2)$
   $n^{\log_b a} = n^{\log_2 7}$
   $f(n) = O(n^{\log 7 - \epsilon})$ for $\epsilon = 0.8$
   Case 1 applies
   $\therefore$ $T(n) = \theta(n^{\log_2 7})$

Rack Your Brain

Give the tight asymptotic bound for the recurrence relation
$T(n) = 2T(n / 4) + \sqrt{n}$

Previous Years' Question

Consider the following three functions.
$f_1 = 10^n$ $f_2 = n^{\log n}$ $f_3 = n^{\sqrt{n}}$
Which one of the following options arranges the functions in the increasing order of asymptotic growth rate?
(A) f_3, f_2, f_1 (B) f_2, f_1, f_3
(C) f_1, f_2, f_3 (D) f_2, f_3, f_1
Solution: (D) **[GATE CS 2021 (Set-1)]**

Previous Years' Question

There are n unsorted arrays: $A_1, A_2,..., A_n$. Assume that n is odd. Each of $A_1, A_2,..., A_n$ contains n distinct elements. There are no common elements between any two arrays. The worst-case time complexity of computing the median of the medians of $A_1, A_2,..., A_n$ is **[GATE CS 2019]**
(A) O(n) (B) O(nlogn)
(C) $O(n^2)$ (D) $O(n^2 \log n)$
Solution: (C)

Previous Years' Question

Consider the following C function
[GATE CSE - 2017 (Set-2)]

```
int fun (int n) {
  int i, j;
  for (i = 1; i < = n; i++) {
    for (j = 1 ; j < n ; j+=i) {
      printf ("%d %d", i, j) ;
      }
   }
}
```

Asymptotic notation of fun in terms of θ notation is
(A) $\theta(n\sqrt{n})$ (B) $\theta(n^2)$
(C) θ (n logn) (D) $\theta(n^2 \log n)$
Solution:(C)

Previous Years' Question

Consider the following functions from positive integers to real numbers:

$$10, \sqrt{n}, n, \log_2 n, \frac{100}{n}$$

The correct arrangement of the above functions in increasing order of asymptotic complexity is:
[GATE CSE 2017 (Set-1)]

(A) $\log_n n, \frac{100}{n}, 10\sqrt{n}, n$

(B) $\frac{100}{n}, 10, \log_2 n, \sqrt{n}, n$

(C) $10\frac{100}{n}, \sqrt{n}, \log_2 \sqrt{n}, n$

(D) $\frac{100}{n}, \log_2 n, 10, \sqrt{n}, n$

Solution: (B)

Previous Years' Question

In an adjacency list representation of an undirected simple graph G = (V, E), each edge (u, v) has two adjacency list entries: [v] in the adjacency list of u, and [u] in the adjacency list of v. These are called twins of each other. A twin pointer is a pointer from an adjacency list entry to its twin. If |E| = m and |V| = n, and the memory size is not a constraint, what is the time complexity of the most efficient algorithm to set the twin pointer in each entry in each adjacency list?
[GATE CSE 2016 (Set-2)]

(A) Θ(n2)

(B) Θ(n + m)

(C) $\Theta(m^2)$

(D) $\Theta(n^4)$

Solution: (B)

Chapter Summary

- **Types of analysis**
  - Worst case
  - Best case
  - Average case
  - Amortized.
- **Asymptotic notations**
  - **Big O notation:**
    $f(n) = O(g(n))$ if there exist constants n_0 and c such that $f(n) \leq c\ g(n)$ for all $n \geq n_0$
  - **Big omega notation:**
    $f(n) = \Omega(g(n))$ if there exist constants n_0 and c such that $f(n) \geq c\ g(n)$ for all $n \geq n_0$
  - **Theta (θ) notation:**
    If $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = \theta(g(n))$
- **Master's theorem**
  $T(n) = aT(n/b) + f(n)$ where a 1 and $b \geq 1$ be constant, f(n) be the function.
  - If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = (n^{\log_b a})$
  - If $f(n) = (n^{\log_b a})$, then $T(n) = (n^{\log_b a})$
  - If $f(n) = (n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$, and $af(n/b) \leq c\ f(n)$ for some constant $c < 1$ for all n and all sufficiently large $T(n) = \theta(f(n)$

2 Divide and Conquer

The divide and conquer strategy involves three main steps.

- **Step 1:** Problem is divided into some subproblems that are smaller parts of the same problem.
- **Step 2:** Solve (Conquer) the subproblems in a recursive manner. If the subproblem is small, then solve it directly.
- **Step 3:** Combine solutions of all the subproblems into the original problem's solution.
- When the subproblem is not small enough to find solution directly (without recursion), i.e., with base case, it is divided into smaller subproblems.

The Maximum Subarray Problem

Problem:

Given an array A[1...n], one should find a continuous subarray that sums up to the maximum value.

Input: Array A[1...n]

Output:

The maximum sum of values possible in the subarray A[i...j] and the indices i and j.

> **Note:**
> Maximum subarray might not be unique.

Example:

Array A	5	–4	3	4	–2	1
	1	2	3	4	5	6

Maximum subarray: A[1...4] (i = 1, j = 4) and sum = 8.

Divide and Conquer Strategy:

1. **Divide:** The array A[l...h] is divided into two subarrays of as equal size as possible by calculating the mid.
2. **Conquer:** By algorithm given below.
   a) Finding maximum subarray of A[l...m] and A[m+1...h]
   b) Finding a maximum subarray that crosses the midpoint.
3. **Combine:** Returning the maximum of the above tree.

This process of finding solution works because any subarray may lie completely on one side of the midpoint, on the other side, or crossing the midpoint.

Algorithm:

```
FIND-MAXIMUM-CROSSING-SUBARRAY (Arr, l, m, h)
l-sum = −∞
sum = 0
    for i ← m downto l
        sum ← sum + Arr[i]
        if sum > l-sum
            l-sum ← sum
            max-left ← i
    r-sum ← −∞
    sum ← 0
    for j ← m + 1 to h
        sum ← sum + Arr[j]
        if sum > r-sum
            r-sum ← sum
            max-right ← j
    return (max-left, max-right, l-sum + r-sum)
```

Algorithm returns the indices i and j and the sum of two subarrays.

- If the subarray Arr[l...h] contain n entries, then n = h–l+1.
- Since, each iteration of two for loops takes $\theta(1)$ time, and the number of iterations is to be counted to get the total time complexity.

- The first for loop makes m-l+1 iterations, and the second for loop makes h-m iterations.

Total number of iterations = (m-l+1) + (h-m)

= h-l+1

= n

∴ FIND-MAXIMUM-CROSSING-SUBARRAY(Arr, l, m, h) takes θ(n) time.

- With the help of the FIND-MAXIMUM-CROSSING-SUBARRAY procedure in hand that runs in linear time, pseudocode for divide and conquer can be written to solve the maximum subarray sum problem.

Find-Maximum-Subarray(Arr, L, H)

1. **If** h == l
2. **Return** (l, h, Arr[l])
3. Else m = $\lfloor (l + h) / 2 \rfloor$
4. (L-low, l-high, l-sum) = FIND-MAXIMUM-SUBARRAY(Arr, l, m)
5. (R-low, r-high, r-sum) = FIND-MAXIMUM-SUBARRAY (Arr, m+1, h)
6. (Cross-low, cross-high, cross-sum) = FIND-MAXIMUM-CROSSING-SUBARRAY (Arr, l, m, h)
7. **If** l-sum ≥ r-sum and l-sum ≥ cross-sum
8. **Return** (l-low, l-high, l-sum)
9. Else if r-sum ≥ l-sum and r-sum ≥ cross-sum
10. **Return** (r-low, r-high, r-sum)
11. Else **return**(cross-low, cross-high, cross-sum)

Remarks:

1. Initial call: FIND-MAXIMUM-SUBARRAY (Arr, l, h).
2. Base case is used when the subarray can not be divided further, i.e., with 1 element.
3. Divide is based on the value of m.
   Conquer by the two recursive calls to FIND-MAXIMUM-SUBARRAY and a call to FIND-MAXIMUM-CROSSING-SUBARRAY.
   Combine the solutions by finding which of the three results gives the maximum sum.
4. **Complexity:**
   T(n) = 2.T(n/2) + θ(n) + θ(1)
   = θ(nlogn) by masters theorem.

Previous Years' Question

Consider a sequence of 14 elements:
A= [-5, -10, 6, 3, -1, -2, 13, 4, -9, -1, 4, 12, -3, O].
The sequence sum S(i, j) = ~t=i A[k]. Determine the maximum of S(i, j), where 0 ~ i ~j < 14.
(Divide and conquer approach may be used.)
Solution: Sum = 6 + 3 - 1 - 2 + 13 + 4 - 9 - 1 + 4 + 12 = 29

Strassen's Algorithm for Matrix Multiplication

Given 2 square matrices A and B of size n×n each, find the product of two matrices.

Naive method:

SQUARE–MATRIX–MULTIPLY (A, B)

1. n ← A.rows
2. let C be a new n×n matrix
3. **for** i ← 1 down to n
4. **for** j ← 1 down to n
5. C_{ij} ← 0
6. **for** k ← 1 down to n
7. $C_{ij} \leftarrow C_{ij} + a_{ik} . b_{kj}$
8. **return** C

a) Because each of the triply–nested loops runs for exactly n iterations, and each execution of line 7 takes constant time.

∴ Time complexity = $O(n^3)$ time for naive method

A simple divide and conquer algorithm:

- Divide matrices A and B into four n/2 × n/2 matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (1)$$

So that we rewrite the equation C = A.B as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} . \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$
$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$
$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$
$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

- By using these equations, create a straight forward recursive, divide–and–conquer algorithm.

SQUARE–MATRIX–MULTIPLY–RECURSIVE (A,B)

1. N = A. rows
2. Let C be a new n×n matrix
3. **If** n == 1
4. $C_{11} = a_{11} . b_{11}$
5. Else partition A, B, and C as in equations (1)
6. C_{11} = SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{11}, B_{11}) + SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{12}, B_{21})
7. C_{12} = SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{11}, B_{12}) + SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{12}, B_{22})
8. C_{21} = SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{21}, B_{11}) + SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{22}, B_{21})
9. C_{22} = SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{21}, B_{12}) + SQUARE–MATRIX–MULTIPLY–RECURSIVE (A_{22}, B_{22})
10. **Return** C

- In the above method, we do eight multiplications for matrices of size n/2×n/2 and 4 additions.
- Addition of 2 matrices takes $O(n^2)$ time

So, the time complexity can be given as

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

By masters theorem, the time complexity is $O(n^3)$, which is the same as the naive method.

Strassen's method:

- Strassen's method makes the recursion tree sparse.
- Instead of eight recursive multiplications of n/2x n/2 matrices, it performs only seven.
- Strassen's method also uses the divide and conquer strategy, i.e., it divides the matrix into some smaller matrices of order n/2 x n/2.
- In Strassen's matrix multiplication, the result of four sub-matrices is calculated using the formula below:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{bmatrix}$$

Where

$P_1 = A_{11} (B_{12} - B_{22})$

$P_2 = B_{22} (A_{11} + A_{12})$

$P_3 = B_{11} (A_{21} + A_{22})$

$P_4 = A_{22} (B_{21} - B_{11})$

$P_5 = A_{11} (B_{11} + B_{22}) + A_{22} (B_{11} + B_{22})$

$= (A_{11} + A_{22}) (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) (B_{11} + B_{12})$

So, a total of 7 multiplications are required.

Time complexity of Strassen's method:

Adding and subtracting two matrices takes $O(n^2)$ time.

Hence, the time complexity taken by the algorithm can be written as

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

By master's theorem, the time complexity is $O(n\log_2 7)$, which is approximately $O(n^{2.8074})$

NOTE:

Strassen's method is not suggested for practical purposes due to the following reasons:

- More number of constants are used and naive methods performs better in most of the cases.
- There are better methods available for sparse matrices in particular.
- The sub-matrices occupy more space in recursion method.
- Strassen's algorithm gives more errors due to precision limitations in computer arithmetic on non-integer value.

Problem of Sorting

Input: An array A of size n.

Output: Reordering of the array such that A[0] ≤ A[1] ≤ ... ≤ A[n–1].

Properties of Sorting algorithms:

In place:

"A sorting algorithm is said to be in place if a maximum of O(1) (in some cases it can be increased) elements are stored outside the array at any moment.

E.g., Insertion-sort is in place.

Stability:

A sorting algorithm is said to be stable if the order of repeated elements is not changed.

E.g., Insertion sort, merge sort, etc.

Adaptive:

A sorting algorithm falls into the adaptive sort family if it takes advantage of the existing order in its input.

E.g., Insertion sort

Merge sort:

The merge sort algorithm uses divide and conquer approach.

Divide:

Divide the array of n elements each into two subarrays of n/2 elements each.

Conquer:

Sort the subarrays in the recursive method by using merge sort.

Combine:

- Merge the two subarrays to result in the sorted array.
- The main operation of merge sort algorithm is to merge two sorted arrays.
- MERGE(A,p,q,r) is used to merge the arrays such that p ≤ q < r.It assumes the two arrays A[p...q] and A[q+1...r] are in sorted order.

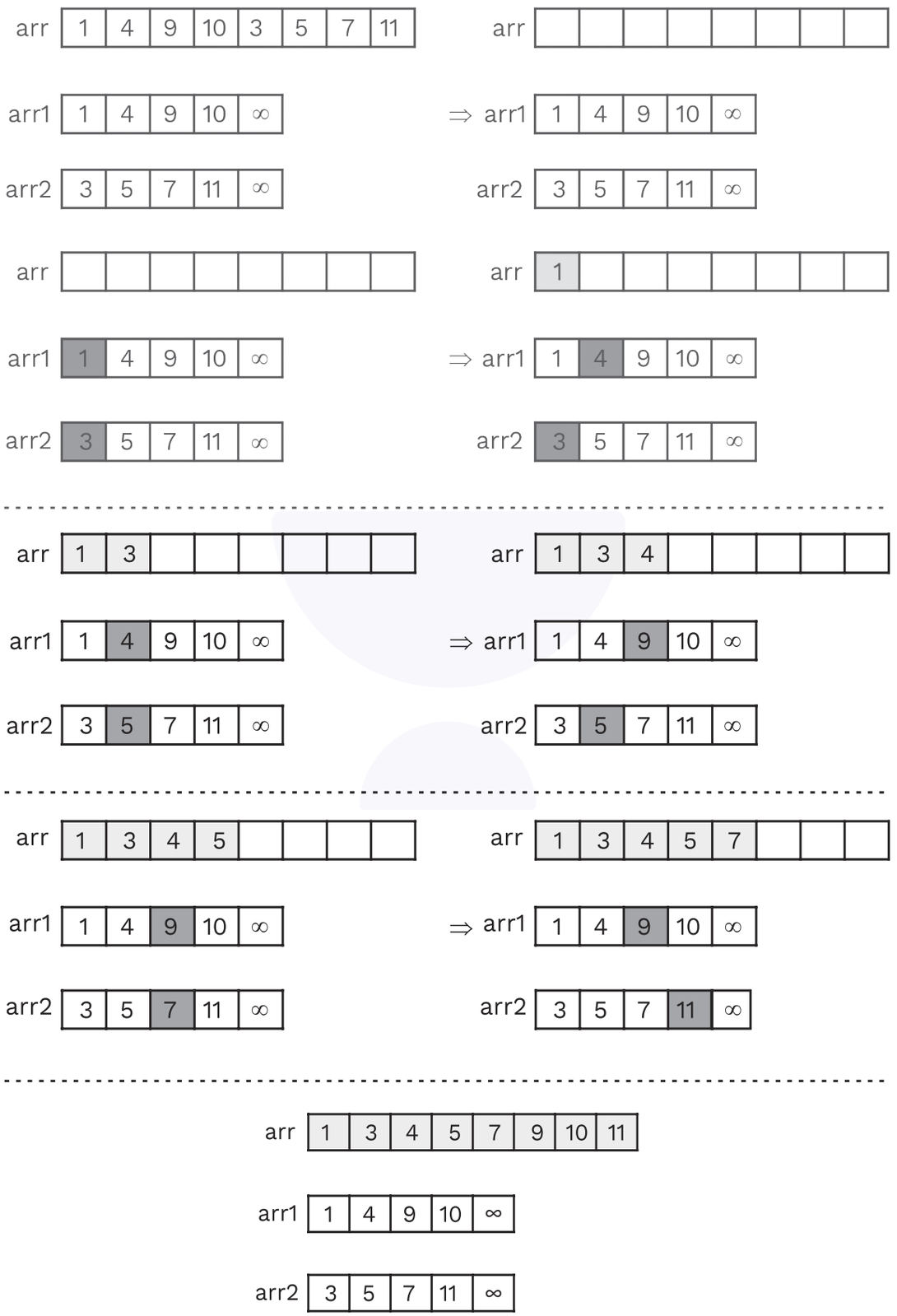
Algorithm

MERGE_FUNCTION (arr, low, mid, high)

1. Let n1 ¬ (mid – low) + 1
2. Let n2 ¬ high – mid
3. Let arr1[1...(n1 + 1)] & arr ¬ 2[1...(n2 + 1)] be the new arrays
4. for i ¬ 1 to n1
5. arr1[i] ¬ arr[low + i – 1]
6. for j ¬ 1 to n2
7. arr2[j] ¬ arr[mid + j]
8. arr1[n1 + 1] ¬ ∞
9. arr2[n2 + 1] ¬ ∞
10. i ¬ 1
11. j ¬ 1
12. for k ¬ low to high
13. if arr1[i] ≤ arr2[j]
14. arr[k] ¬ arr1[i]
15. i ¬ i + 1
16. else arr[k] = arr2[j]
17. j ¬ j + 1

- First line finds the length n_1 of the subarray arr[low...mid], and 2nd line finds the length n_2 of the subarray arr[mid+1...high].
- arr1 and arr2 arrays are created with lengths n_1 and n_2, respectively.
- The extra position in each array holds the sentinal.
- For loop of the line (4-5 and 6-7) copies subarray arr[low...mid] into arr1[1...n_1] and arr[mid + 1...high] into arr2[1...n_2].
- Traverse arr1 and arr2 simultaneously from left to right, and write the smallest element of the current positions to arr.

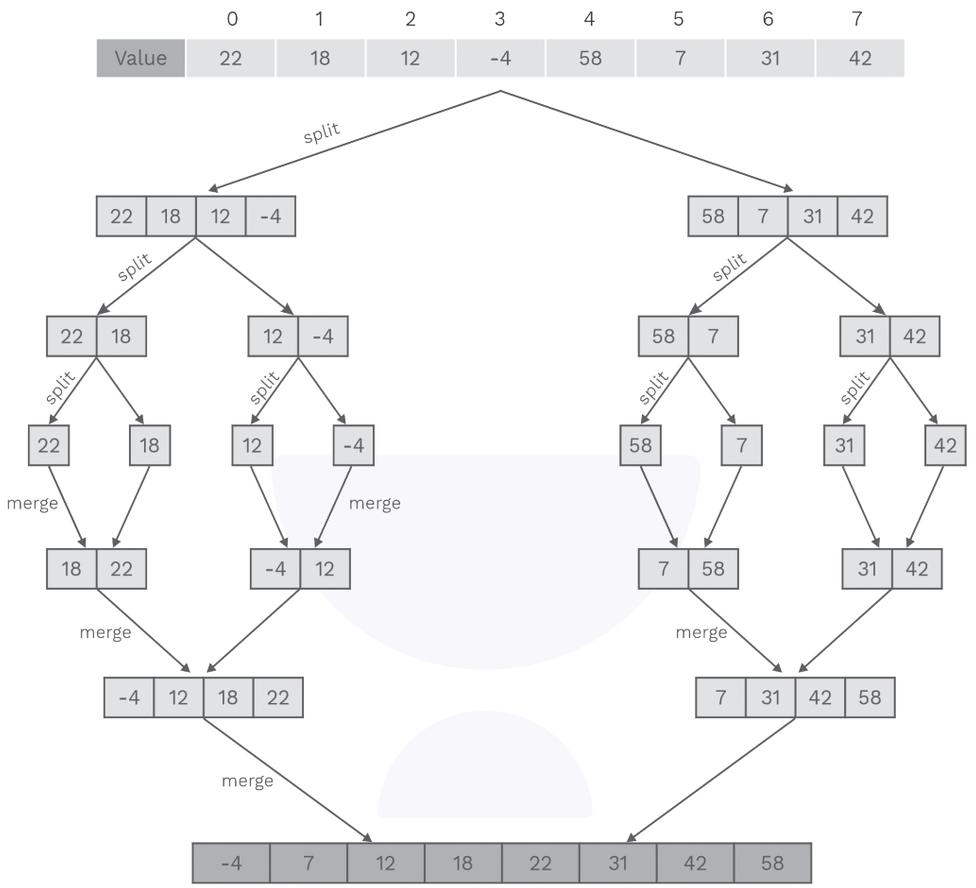
Example:


arr 1 4 9 10 3 5 7 11
arr
arr1 1 4 9 10 ∞
⇒ arr1 1 4 9 10 ∞
arr2 3 5 7 11 ∞
arr2 3 5 7 11 ∞
arr
arr 1
arr1 1 4 9 10 ∞
⇒ arr1 1 4 9 10 ∞
arr2 3 5 7 11 ∞
arr2 3 5 7 11 ∞
arr 1 3
arr 1 3 4
arr1 1 4 9 10 ∞
⇒ arr1 1 4 9 10 ∞
arr2 3 5 7 11 ∞
arr2 3 5 7 11 ∞
arr 1 3 4 5
arr 1 3 4 5 7
arr1 1 4 9 10 ∞
⇒ arr1 1 4 9 10 ∞
arr2 3 5 7 11 ∞
arr2 3 5 7 11 ∞
arr 1 3 4 5 7 9 10 11
arr1 1 4 9 10 ∞
arr2 3 5 7 11 ∞

MERGE-SORT (arr, low, high)

1. **if** low < high
2. mid = $\lfloor (low + high) / 2 \rfloor$
3. MERGE-SORT (arr, low, mid)
4. MERGE-SORT (arr, mid+1, high)
5. MERGE (arr, low, mid, high)

Example:



Analysis:

- Even the Merge sort works correctly on odd-length arrays. Recurrence analysis is simplified if we assume the input array length is of power of 2.

Break down of running time:

Divide:

This step finds the middle of the subarray in constant time, i.e., O(1).

Conquer:

Solving two subproblems of $\frac{n}{2}$ size each recursively results in $2T\left(\frac{n}{2}\right)$ time complexity.

Combine:

MERGE_FUNCTION procedure applied on an n-sized subarray takes time $\theta(n)$.

Recurrence for the worst-case running time T(n) of merge sort.

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

By Masters Theorem:

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

Rack Your Brain

Think about the properties of the merge sort and other kind of implementation of merge sort

Hint: There is an in place merge sort implementation.

Previous Years' Question

Assume that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

(A) 256 (B) 512

(C) 1024 (D) 2048

Solution: (B)

Quick sort:

Quick sort uses 3 step divide and conquer approach to sort an array A[p...r].

Divide:

Rearrange the array A into two subarrays A[p...q-1] and A[q+1...r].

The partition procedure computes the index q such that the elements A[p...q-1] are smaller than A[q] and the elements A[q+1...r] are greater.

Conquer:

Sort the 2 subarrays A[p...q-1] and A[q+1...r] recursively in quicksort.

Combine:

Since all the subarrays are already sorted, no extra work is needed to combine them together. Array A[p...r] is now sorted.

Algorithm:

QUICKSORT_ALGO (Arr, low, high)

1. if low < high
2. mid ← PARTITION_OF_ARRAY (Arr, low, high)
3. QUICKSORT_ALGO (Arr, low, mid-1)
4. QUICKSORT_ALGO (Arr, mid+1, high)

The main part of the algorithm is the PARTITION_OF_ARRAY procedure, which makes the subarray Arr[low...high] in place.

PARTITION_OF_ARRAY (Arr, low, high)

1. x ← Arr[high]
2. i ← low-1
3. for j ← low to high-1
4. i f Arr[j]<= x
5. i ← i + 1
6. swap Arr[i] with Arr[j]
7. swap Arr[i + 1] with Arr[high]
8. return i + 1

Example:

(A)

i, low	j						high
2	8	7	1	3	5	6	4

(B)

low, i	j						high
2	8	7	1	3	5	6	4

(C)

low, i		j					high
2	8	7	1	3	5	6	4

(D)

low, i			j				high
2	8	7	1	3	5	6	4

(E)

low	i			j			high
2	1	7	8	3	5	6	4

(F)

low		i			j		high
2	1	3	8	7	5	6	4

(G)

low		i				j	high
2	1	3	8	7	5	6	4

(H)

low		i					high
2	1	3	8	7	5	6	4

(I)

low		i					high
2	1	3	4	7	5	6	8

PARTITION_OF_ARRAY applied on an Array:

- Array entry A[r] be the pivot element x.
- Orange colour elements are all in the first partition values no greater than x.
- Pink colour elements are in the second partition with values greater than x.
- Blue colour elements, have not yet been put in one of the first two partitions.

Analysing quicksort:

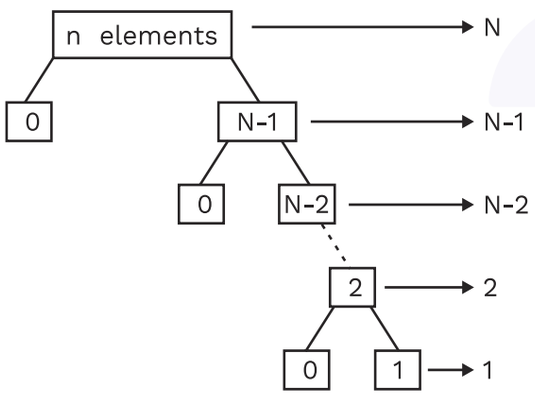
The choice of pivot is most crucial:

- A incorrect pivot may lead to worst-case time complexity $O(n^2)$, whereas a pivot that divides the array into 2 equal-sized subarrays gives the best case time complexity, i.e., O(nlogn).

The worst-case choice:

The pivot may be the largest or the smallest of all the elements in the array.

- When the first iteration is done, then one subarray contains 0 elements, and the other subarray has n-1 elements.
- Quicksort is applied recursively on this second subarray of n-1 elements.



However, quicksort is fast on the "randomly scattered" pivots.

The partition step needs atleast n-1 comparisons.

- The recurrence for the running time is

$$\left.\begin{aligned} T(n) &= T(n-1)+\Theta(n) \\ T(1) &= \theta(1) \end{aligned}\right\}$$

T(n)=T(n-1)+Θ(n)

=T(n-1)+cn (removing Θ with constant)

=T(n-2)+c(n-1)+cn

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

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

=c(1+2+3+...+n)

=c(n(n+1)/2)=O(n^2).

Average case:

T(n): Average number of comparisons during quicksort on n elements.

$$T(1)=\theta(1), T(0)=\theta(1)$$

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

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

$$nT(n)=2\sum_{i=1}^{n}\left(T(i-1)\right)+(n-1)n \quad \text{...(i)}$$

For (n-1)

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

$$(n-1)T(n-1)=2\sum_{i=1}^{n-1}\left(T(i-1)\right)+(n-1)(n-2) \quad \text{...(ii)}$$

Subtracting equation (2) from (1), we get

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

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

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

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

where $g(n)=\frac{T(n)}{n+1}$

Taking RHS-

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

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

$$= \frac{2}{n} - \frac{4}{n} + \frac{4}{n+1}$$

$$= \frac{4}{n+1} - \frac{2}{n}$$

Now,

$$g(n) - g(n-1) = \frac{4}{n+1} - \frac{2}{n}$$

Hence,

$$g(n) = \frac{4}{n+1} + \left(2\sum_{j=2}^{n}\frac{1}{j}\right) - 2$$

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

$$= \frac{4}{n+1} + 2H(n) - 4$$

Now,

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

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

$T(n) = 2(n+1)\ \log_e n + 1.16(n+1) - 4n$

$T(n) = 2n\log_e n - 2.84n + O(1)$

$T(n) = 2n\log_e n$

The average number of comparisons during quicksort on n elements approaches.

$2n\log_2 n - 2.84n$

$= 1.39n\ \log_2 n - O(n)$

So, the average case time complexity as quicksort is O(nlogn).

Previous Years' Question

Let p be a quicksort program to sort numbers in ascending order using the first element as pivot. Let t and t be the number of comparisons 1 2 made by p for the inputs {1,2,3,4,5} and {4,1,5,3,2}, respectively.
Which one of the following holds?

[CS 2014 (Set-1)]

(A) t = 5 1
(B) t1 < t2
(C) t1 > t2
(D) t = t 1 2

Solution: (C)

Previous Years' Question

In quicksort, for sorting n elements, the (n/4) th smallest element is selected as pivot using an O(n) time algorithm. What is the worstcase time complexity of the quicksort?

[CS 2009]

(A) θ(n)
(B) θ(nlogn)
(C) θ(n2)
(D) θ(n logn) 2

Solution: (B)

Previous Years' Question

Randomised quicksort is an extension of quicksort where the pivot is chosen randomly. What is the worst-case complexity of sorting n numbers using randomised quicksort?

[CS 2001]

(A) O(n)
(B) O(nlogn)
(C) $O(n^2)$
(D) O(n!)

Solution: (C)

Randomised quicksort:

- Assume all elements are distinct.
- Partition around a random element, i.e., the pivot is chosen randomly from the set of elements.
- This implies the splits n-1, n-2, ..., n-1 have the same probability of 1/n.

Note:
Randomisation is a general tool to improve algorithms with bad worst-case but good or average-case complexity.

```
Randomised_Partition (Arr, p, r)
  {
      i ← Random(p, r)      /* it generate a random number between p
                               and r and including p and r */
      exchange Arr[r] ↔ Arr[i]
      return Partition (Arr, p, r)
  }
Randomised_Quicksort (Arr, p, r)
{
  if p < r
   q ← Randomised_Partition (Arr, p, r)
   Randomised_Quicksort (Arr, p, q-1)
   Randomised_Quicksort (Arr, q+1, r)
}
```

Time complexity:

- Let's assume T(n) be the number of comparisons needed to sort n numbers using quicksort.
- Since each split occurs with probability 1/n. In quicksort, every number requires atleast (n-1) comparison because the (n-1) number of elements has to be compared with the pivot element.

Let the pivot is i^{th} smallest element. Then we get (i-1) element on the left side and (n-i) element on the right side. And we have to do randomised-quicksort on (i-1) th and $(n-i)^{th}$ element. So, it take T(i-1) and T(n-i) time, respectively.

$\therefore$ T(n) = T(i-1) + T(n-i) + (n-1) with probability 1/n.

↑

this is because atleast (n-1) comparison is required.

Hence,

$$T(n) = \frac{1}{n}\sum_{j=1}^{n}\left(T(j-1) + T(n-j) + n - 1\right)$$

(replacing i with j for simplicity)

= This is the expectation.

$\sum_{j=1}^{n} T(j-1)$ = This quantity varies from T(0) to T(n-1)

$\sum_{j=1}^{n} T(n-j)$ = This quantity varies from T(n-1) to T(0)

So, every term T(0) to T(n-1) appears twice.

So, we write it as:

$$= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + \frac{1}{n}\sum_{j=1}^{n} n - 1$$

$$= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + \frac{n(n-1)}{n}$$

$$= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + n - 1 \qquad ...(3)$$

$$= O(n \log_2 n)$$

$\therefore$ The expected number of comparisons is O(nlogn). But the worst-case time complexity of randomised quicksort is $O(n^2)$.

Note:

- The running time of quicksort does not depend on the input. It depends on the random numbers provided by the generation.
- Thus, for the same input the program might take 2sec today and 5sec tomorrow.
- The average time taken over many different runs of the program would give us the expected time.
- Same as saying that we are taking average over all possible random number sequences provided by the generator.

Solution of quicksort( ) recurrence relation:

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n - 1 \qquad \text{...(i) (from eq (3))}$$

Note that T(0) = 0, T(1) = 0.

Now, $T(n-1) = \frac{2}{(n-1)}\sum_{i=0}^{n-2} T(i) + (n-2)$

or, $(T(n-1) - n + 2) = \frac{2}{(n-1)}\sum_{i=0}^{n-2} T(i)$

Multiply both sides by $\frac{(n-1)}{n}$

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

or, $\frac{2}{n}\sum_{i=0}^{n-2} T(i) = \frac{(n-1)}{n} = \left(T(n-1) - n + 2\right)$ ...(ii)

from (1) and (2);

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

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

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

So, we got $T(n) = \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n}$

$$< \left(\frac{n+1}{n}\right)T(n-1) + 2 \qquad \left[2 * \frac{(n-1)}{n} < 2\right]$$

$$\left[\text{ex} : 2 * \frac{4}{5} = 2 * 0.8 = 1.6\right]$$

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

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

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

$$< \frac{n+1}{(n-1)}\left(\frac{n-2+1}{n-2}T(n-3) + 2\right) + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-1)}\left(\frac{n-1}{(n-2)}T(n-3) + 2\right) + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-2)}T(n-3) + \frac{2(n+1)}{(n-1)} + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-2)}T(n-3) + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{n+1}{(n-2)}\left[\frac{n-3+1}{n-3}T(n-4) + 2\right]$$

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

$$< \frac{n+1}{(n-2)}\left[\frac{n-2}{(n-3)}T(n-4) + 2\right]$$

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

$$< \frac{n+1}{(n-3)}T(n-4) + \frac{2(n+1)}{(n-2)} + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{(n+1)}{(n-3)}T(n-4) + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)}\right] + 2$$

$$< \frac{(n+1)}{\left(n-(n+3)\right)}T(2)$$

$$+ 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)} + \ldots + \frac{1}{4}\right] + 2$$

$$< \frac{(n+1)}{(n-n+1)} T(0) \cdot$$

$$+1)\left[\frac{1}{n}+\frac{1}{(n-1)}+\frac{1}{(n-2)}+\ldots+\frac{1}{3}+\frac{1}{2}\right]+2$$

$$< 2(n+1)\underbrace{\left[\frac{1}{n}+\frac{1}{(n-1)}+\frac{1}{(n-2)}+\ldots+\frac{1}{2}\right]}+2$$

$$\left[\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\ldots+\frac{1}{(n-1)}+\frac{1}{n}\right.$$

$$\left.\begin{aligned}&= \text{harmonic series}\\ &= (\log n)\end{aligned}\right]$$

$$< 2(n+1)(\log n-1)+2$$

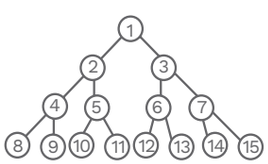
$$= O(n \log n)$$

Heap sort:

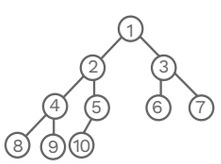
- Heap is a data structure used to store and manage information. It is the design technique used by heapsort.
- A binary heap is an almost complete binary tree.

- The tree is said to be almost complete if all the levels are filled except the last level. The nodes in the last level are also filled from left to right.

Binary tree representation:



A full binary tree of height 3.



A complete binary tree with 10 nodes and height 3.

Heap as a tree:

- Root:
  First element in the array
- parent i) = $\lfloor i/2 \rfloor$:
  Returns index of node's parent
- left i) = 2i :
  Returns index of node's left child.
- right i) = 2i + 1:
  Returns index of node's right child.
  Height of a binary heap is O(logn).

There are two types of binary heaps:

Max–heaps and min–heaps

In both the heaps, the values in the nodes satisfy the heap property.

In a max heap, the max–heap property is that for every node i other than the root node, A[PARENTi)] ≥ A[i]

A min–heap property is that for every node i other than the root,

A[PARENT i)] ≤ A[i]

The smallest valued node in a min-heap is at the root.

- For the heap sort algorithm, we use max–heap.
- Min–heaps commonly implement priority queues.

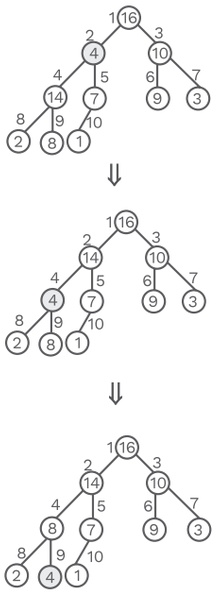
Heap operations:

- Build–Max–Heap:
  Produce a max–heap from an unordered array
- Max_heapify:
  Changes every single violation of the heap property in every subtree at its root.
- Insert, extract_max, heap sort

Max_heapify:

- Assume that the trees rooted at left i) and right i) are max_heap.
- If element A[i] violates the max_heap property, correct violation by "trickling" element A[i] down the tree, making the subtree rooted at index i a max_heap.

Max–heapify (example):



- **Max–Heapify Pseudocode**

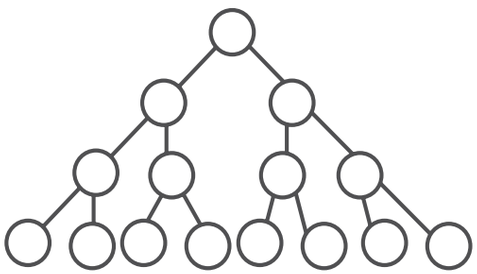
  MAX–HEAPIFY(Arr, i)
  1. l ← left_node i)
  2. r ← right_node i)
  3. if (l ≤ Arr.size_of_heap && Arr[i]<Arr[l])
  4. maximum ← l
  5. else maximum ← i
  6. if (r ≤ Arr.size_of_heap && Arr[maximum] < Arr[r])
  7. maximum ← r
  8. if (maximum ≠ i)
  9. swap Arr[i] and Arr[maximum]
  10. MAX–HEAPIFY (Arr, maximum)
- Convert array A[1...n] into a max–heap, where n=A.length.
- The elements in the subarray $A\left[\left(\lfloor n/2 \rfloor + 1\right)...n\right]$ are all leaves of the tree, and so each is a 1–element heap to begin with.

Build–max–heap (a):

1. A. size_of_heap = A. length
2. for i = $\lfloor A.length / 2 \rfloor$ down to 1
3. MAX–HEAPIFY (A, i)

Note:
The number of nodes present in a complete binary tree at height h is $\left\lceil \frac{n}{2^{h-1}} \right\rceil$, where n is the total nodes of the tree.

- Observe, however, that MAX–HEAPIFY takes O(1) time for nodes that are a level above the leaves, and O(L) for the nodes that are L levels above the leaves.



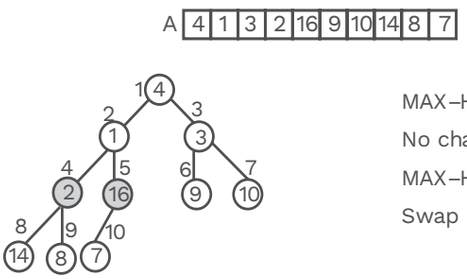
We have $\lceil n/4 \rceil$ nodes with level logn

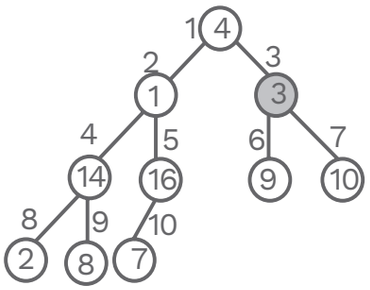
We have $\lceil n/8 \rceil$ nodes with level logn–1,

We have 1 root node that is 0 level above the leaves.

- Total amount of work in BUILD–MAX–HEAP for loop can be summed as
  n/4 (1C) + n/8 (2C) + n/16 (3C) + ... + 1($\log_n$C)
  substitute n/4 = 2^K
  C 2^K ($1/2^0 + 2/2^1 + 3/2^2$ + ...$(K+1)/2^K$)
  The term in brackets is bounded by a constant.
- This means that build–max–heap is O(n).

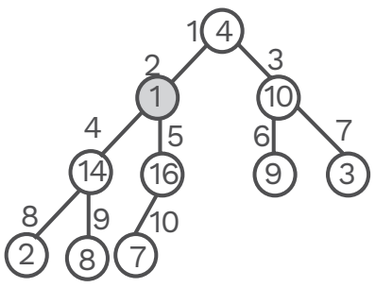
Build–max–heap (example):





MAX–HEAPIFY (A, 3)

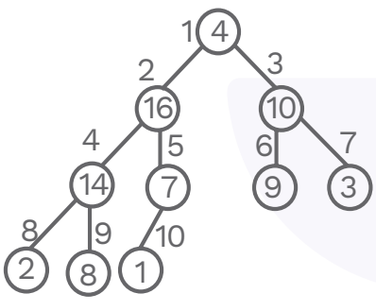
Swap A[3] and A[7]



MAX–HEAPIFY (A, 2)

Swap A[2] and A[5]

Swap A[5] and A[10]

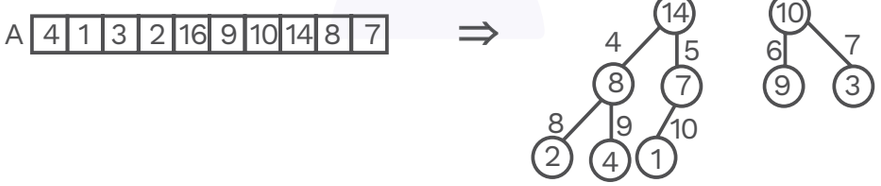


MAX–HEAPIFY (A, 1)

Swap A[1] with A[2]

Swap A[2] with A[4]

Swap A[4] with A[9]



Heap Sort

Sorting strategy:

1. A max-heap is built using an unordered array.
2. Maximum element A[i] is found.
3. Exchange the values of A[n] and A[1], which results in changing the maximum to the end of the array.
4. Delete node n from the heap.
5. The new root added may violate the max-heap property but its children are already max-heaps. Run max-heap to solve this.
6. Go to step 2 if the heap is not empty.

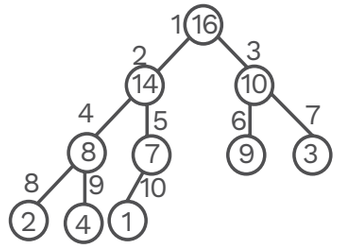
Heapsort(A)

{

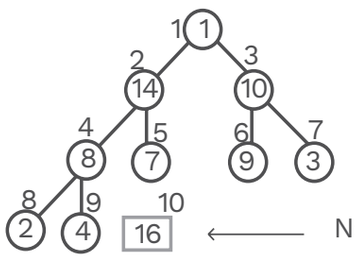
1. BUILD–MAX–HEAP(A)
2. **for** (i = A.length() **down to** 2)
3. swap A[1] with A[i]
4. A.heap–size = (A.heap–size–1)
5. MAX–HEAPIFY (A, 1)

}

Heap–sort demo:



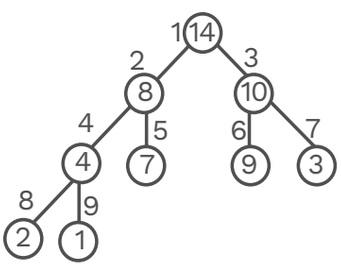
⇓ Swap A[10] and A[1]

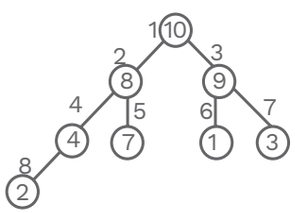


heap – size = 9

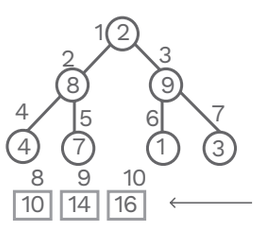
← Not part of heap

⇓ MAX–HEAPIFY (A, 1)

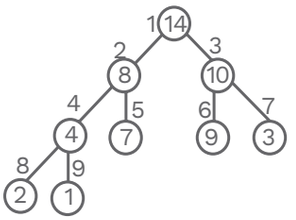




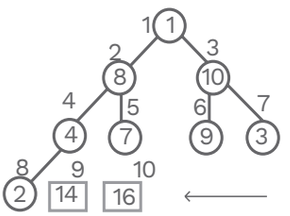
⇓ Swap A[8] and A[1]



← Not part of heap



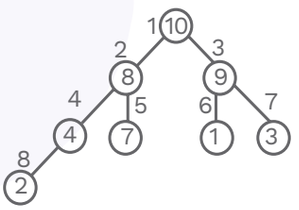
⇓ Swap A[9] and A[1]



← Not part of heap

MAX–HEAPIFY (A, 1)

⇓



Running time:

- The heap is empty after n iterations.
- Each iteration does a swap and a max-heapify operation, which involves O(logn) time.
- So it takes O(nlogn) in total.

Note:
Heapsort is an efficient algorithm and has its own uses, but quicksort beats it, if it is implemented correctly.

- The most commonly used application of heaps is priority queues.

- Same as with heaps, priority queues are of two types,
  Max-priority queue
  Min-priority queue
- A max priority queue has the following operations:
  Insert(S,x)
  Adds an element to set S ,i.e., S=S U {x}
  Maximum(S)
- Returns the largest element in the set.
  Extract max(S)
- Returns the largest element in S and deletes it from the set.
  Increase(S,x,k)
- Increases the value of x to k, and it is assumed that k≥x.
- Apart from these, a min-priority queue supports the operations MINIMUM, EXTRACT MIN, INSERT and DECREASE KEY.
- The procedure HEAP MAXIMUM on the max-priority queue implements the maximum operations in θ(1) time.

Heap–maximum (A):

1. return A[1]

The procedure HEAP–EXTRACT–MAX implements the EXTRACT–MAX operation.

Heap–extract–max (A):

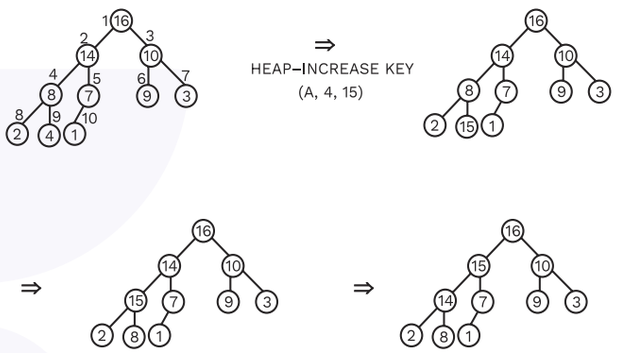
1. **if** A. heap–size <1
2. error "heap underflow"
3. max = A[1]
4. A[1] = A[A. heap–size]
5. A. heap–size = A. heap–size – 1
6. MAX–HEAPIFY (A, 1)
7. **return** max

- The HEAP-EXTRACT-MAX takes O(logn) since it performs O(1) the amount of time on the top of O(logn) time for MAX_ HEAPIFY.
- The HEAP-INCREASE_KEY implements the INCREASE KEY operation.

Heap_increase_element (Arr, i, key):

1. if key < Arr[i]
2. ERROR /* since the key is lesser than the element */
3. Arr[i] ← key
4. while (i > 1) && [Arr[PARENT(i)] < Arr[i]]
5. exchange Arr[i] with Arr[PARENT(i)]
6. i ← PARENT(i)

The HEAP-INCREASE-KEY takes O(log n) time on an n element heap. The path from the updated node in the 3rd line to the root has length O(logn).



- The procedure MAX–HEAP–INSERT implements the INSERT operation.

Max–heap–insert (A, key):

1. A. heap–size = A. heap–size + 1
2. A [A. heap–size] = – ∞
3. HEAP–INCREASE–KEY (A, A. heap–size, key)

The running time of MAX–HEAP–INSERT on an n–element heap is O(logn).

Note:
Time–taken to perform some operations in Max–heap is given below

Fing max	Delete max	Insert	Increase	Decrease key	Find min	Search	Delete
O(1)	O(Logn)	O(Logn)	O(Logn)	O(Logn)	O(n)	O(n)	O(n)

Rack Your Brain

- Give an O(nlogk) – Time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. Hint : Use a min-heap for k–way merging.
- Write Pseudocode for the procedures HEAP MINIMUM, HEAP–EXTRACT–MIN, HEAP–DECREASE–KEY, and MIN–HEAP INSERT that implement a min–priority queue w ith a min–heap.

Previous Years' Question

The number of elements that can be sorted in O(logn) time using heap sort is **[CS 2013]**

(A) $\emptyset(1)$

(B) $\emptyset\left(\sqrt{\log n}\right)$

(C) $\emptyset\left(\frac{\log n}{\log\log n}\right)$

Solution: (C)

Previous Years' Question

Consider a max heap, represented by the array 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is **[CS 2015]**

(A) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35
(B) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
(C) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15
(D) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

Solution: (B)

Previous Years' Question

Consider the following array of elements (89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100). The minimum number of interchanges needed to convert it into a maxheap is: **[CS 2015]**

(A) 4 (B) 5
(C) 2 (D) 3

Solution: (D)

Insertion Sort

Insertion sort is an efficient algorithm to sort arrays with a small number of elements.

- It is the same as sorting and playing cards in your hand.
- Start with an empty hand and pick one card at a time from the cards that are faced down on the table.
- Check the card in the right hand with the cards in the left hand. Compare it with cards from right to left and place it in its correct position.
- At any given time, the cards in the left hand are sorted and are the top cards in the file.

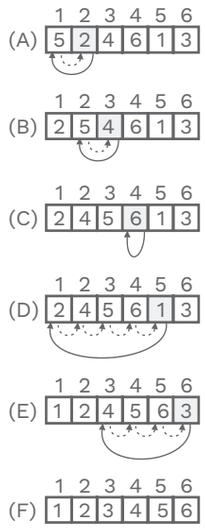
Pseudocode for insertion sort:

Insertion–sort (A):

1. **for** j from 2 to A. length
2. Key = A[j]
3. // Insert A[j] into sorted sequence A[1...j–1]
4. i = j – 1
5. While i > 0 & A[i] > Key
6. A[i+1] = A[i]
7. i = i –1
8. A[i + 1] = Key

- INSERTION–SORT, takes as a parameter an array A[1--n] containing a sequence of n length that is to be sorted.
- The algorithm sorts the input within the array A, with at most O(1) or O(logn) numbers of them are stored outside the array.

- The operation of INSERTION–SORT on array A = {5, 2, 4, 6, 1, 3}



Analysis of Insertion Sort

Worst-case behaviour on an array of n length:

- Inner loop could be executed 'i' times
- 'i' swaps per loop ⇒ $O(n^2)$ total swaps

Rack Your Brain

What sort of input leads to the worst case time complexity in insertion sort?

Best case behaviour on an array of length 'n'

- When the input array is sorted
- Inner loop executed 0 times ⇒ 0 swaps
- While condition is entry condition (always performed at least once)

So, O(n) comparisons are in the best case to verify the array is indeed sorted.

Binary Insertion Sort

BINARY–INSERTION–SORT (A, n)

for j ← 2 to n
insert A[j] Key into the (already sorted) sub–array A[1...j–1]

use binary search to find its right position

- Binary search will take O(logn) time. However, shifting the elements after insertion will still take O(n) time

Complexity:

O(nlogn) comparisons

$O(n^2)$ swaps. So overall time complexity is O(n^2).

Rack Your Brain

Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

NOTE:

INSERTION_SORT gives time complexity depending on how nearly the input sequence is sorted. It is better for almost sorted sequences.

Previous Years' Question

Consider the following array

23	32	45	69	72	73	89	97

Which algorithm out of the following options uses the least number of comparisons (among the array elements) to sort the above array in ascending order?

[CS-2021 (Set-1)]

(A) Selection sort
(B) Merge sort
(C) Insertion sort
(D) Quicksort using the last element as pivot

Solution: (C)

Bubble sort:

Bubble sort is a popular yet inefficient sorting algorithm.

- It works by repeatedly swapping adjacent elements that are out of order.

Bubble sort (Arr)

1. for i ← 1 up to Arr. length – 1
2. for j ← Arr. length down to i + 1
3. if Arr [j] < Arr [j – 1]
4. swap Arr[j] and Arr[j–1]

Analysis:

Worst-Case

- The i-th iteration of the "for loop" of lines 1–4 will cause "n – i" iterations of the for loop of lines 2–4, each with constant time execution, so the worst–case running time of bubble sort is $O(n^2)$

Selection Sort

The selection sort finds the minimum repeatedly and places it at the beginning.

- It divides the array into two subarrays. One is sorted, and another one is unsorted.
- The minimum element is found in an unsorted array and is added at the end of the sorted array.

Example:

A = {1, 20, 6, 30, 42}

Find minimum element in A[0...4] and place it at the beginning.

(A)

0	1	2	3	4
1	20	6	30	42

Find minimum element in A[1...4] and place it at the beginning of A[1...4]

(B)

0	1	2	3	4
1	6	20	30	42

Find minimum element in A[2...4] and place it at the beginning of A[2...4]

(C)

0	1	2	3	4
1	6	20	30	42

Find minimum element in A[3...4] and place it at the beginning of A[3...4]

(D)

0	1	2	3	4
1	6	20	30	42

Pseudocode of selection sort:

Sleection–sort (A):
for j ← 1 to n–1
smallest ← j
for i ← j + 1 to n
if A[i] < A[smallest]
smallest ← i
Exchange A[j] ↔ A[smallest]

Time complexity:

- (n–1) comparisons are needed to find the minimum element from an array of 'n' elements.
- The size of an unsorted array decreases to (n–1) after the minimum element is placed in its proper position, and then (n–2) comparisons are required to find the minimum in the unsorted array.

Therefore,

$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$ comparisons and n swaps

$\therefore$ Time complexity = $O(n^2)$

Counting Sort

- Counting sort assumes every element in the n sized array is in range 0 to k, where k is an integer.

The algorithm:

Input

Array [1...n], where A[j] $\in$ [1,2,3...k]

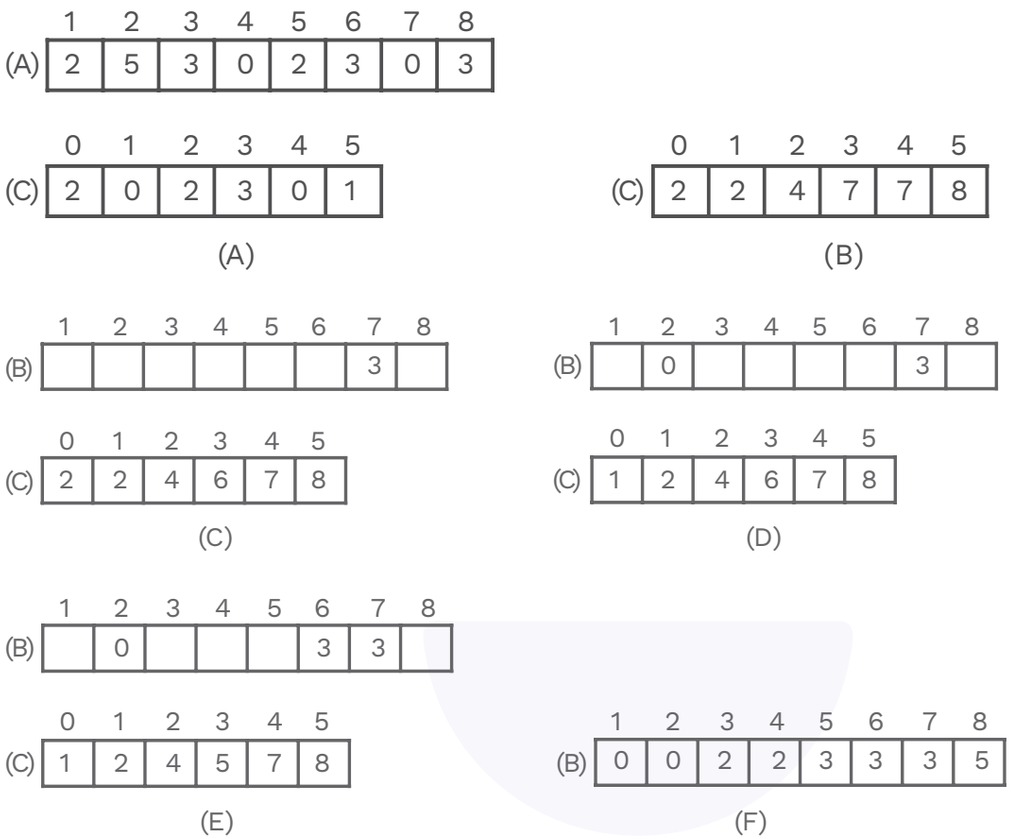
Output

The array [1...n] holds the sorted output and the array count[0...k] provides temporary working storage.

Counting_sort (array, array_size):

1. Max ← maximum element in the array and create an array Count of size max and initialize with zeroes
2. For j ← 0 to size
3. Find the total count of each unique element and store the count at jth index in count array
4. For i ← 1 to max
5. Find the cumulative sum and store it in count array itself
6. For j ← size down to 1 restore the elements to array
7. Decrement count_arr[arr[j]] and make arr[count_arr[arr[j]]]=j

Counting sort example:



Finding the maximum element takes O(n) time.

Creating an array and initializing it to zeroes takes O(k). therefrore, total time complexity=O(n+k) space complexity=O(k) for creating new array.

Since the elements are changed in-place,it is an inplace algorithm

Rack Your Brain

Why don't we always use counting sort?
Hint: Depends on the range K of elements.

Bucket Sort

- Counting sort assumes every element in the array of size n is in the range 0 to k, where k is an integer.
- Bucket sort assumes the input is given a uniform distribution function and runs in O(n) time.
- Same as counting sort, bucket sort is also fast since it assumes something about the input.
- Counting sort thinks that the input consists of n integers in a smaller range, whereas bucket sort assumes that the input is generated by a random process that distributes elements in the array uniformly and independently over an interval [0, 1).
- Bucket sort divides the [0, 1) interval into n equal-sized intervals (known as buckets) and distributes n input elements into the buckets.
- Since the numbers are uniformly and independently distributed over [0, 1), the chances of getting more elements in the same bucket are very, very less.

- To get the output, sort the elements in each bucket and list them in order to get the final sorted order.
- Bucket sort assumes that the input is an array A of n elements, and each element A[i] satisfies the condition $0 \leq A[i] \leq 1$.
- The code requires an auxiliary array B[0...n-1] of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

Bucket-sort (A):

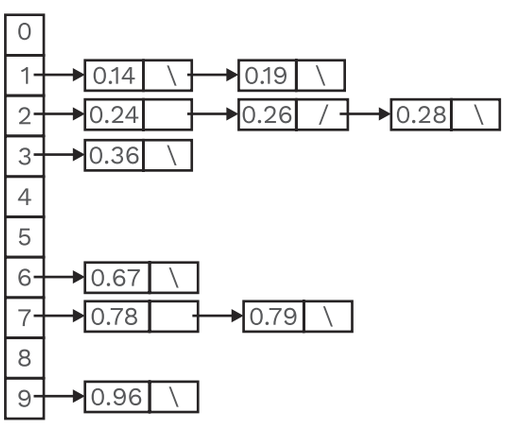
1. Let B[0....size-1] be a new array created.
2. n = A.length
3. For itr<-0 to size-1
4. Make B[itr] empty
5. For itr<-1 to size
6. Do insert A[itr] into list B[n A[itr]]
7. For itr<-0 to size-1
8. Do sort list B[itr] with insertion-sort
9. Merge all lists B[0], B[1],.., B[size-1] together in order

Example:

Let array A have 10 elements as shown below:

A	0.79	0.19	0.36	0.28	0.78	0.96	0.24	0.14	0.26	0.67
	1	2	3	4	5	6	7	8	9	10

The array B[0...9] of sorted lists (buckets) after line 8 of the algorithm are shown below:



Time Complexity

Best Case

- In case if all the elements are uniformly distributed, then every insertion take O(1) time (assuming that every number that we are going to insert next has to be just inserted at the beginning of the list).
- We are going to insert all the number 'n' times, then the total time complexity is going to be O(n) for all the insertion, and then we have to scan all the number one time, so again O(n).
- Therefore, T(n) = O(n).

Worst-case:

- Let us assume that we have 'k' bucket, and since elements, are uniformly distributed. So, each bucket might get $\left(\frac{n}{k}\right)$ element as a list and if we are going to apply insertion sort with all the element, which are present in a bucket then we might have to compare with $\left(\frac{n}{k}\right)$ element then in that case time complexity is $O\left(\left(\frac{n}{k}\right)n\right) = O\left(n^2\right)$.
- Therefore, $O(n^2)$ is the worst-case time complexity.

Space complexity:

- If we assume that the number of bucket is 'k', then 'k' cells have been dedicated for the bucket, and for all the n-elements, we should have space of size n.
- So, space complexity = O(n+k).

Radix-sort:

- Alike counting and bucket sorting algorithms, even radix-sort assumes that the input has some information.
- It assumes that the input values are to be sorted from the base, i.e., means all numbers in the array have d digits.
- In radix sort, sorting done with each digit from the last to the first.
- A stable sort algorithm is used to sort all the numbers by least significant, then last but one, so on.
- Counting sort gives a time complexity of $O(nd) \approx O(n)$ for this procedure in radix sort.

Algorithm:

- Radix sort (A, d) /* Each key in A[1...n] is a d digit integer, and if the
- For (i=1 to d) do keys are not of d-digit, then we have to make 'd'
- Use a stable sorting digit number by appending zero. */
- Algorithm to sort A
- On digit i.
- /* digits are numbers 1 to d from right to left */

Example:

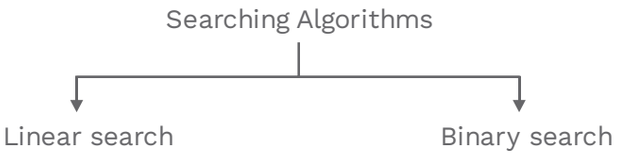
- The operation of radix sort on a list of seven 3-digit numbers is shown below:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- Let n d-digited numbers be given, in which each digit can take k values possible.
- RADIX-SORT sorts the input numbers correctly in $\theta(d(n+k))$ time. If the intermediate stable algorithm used to sort the numbers takes d(n+k).
- When each digit is between 0 to k-1 and k is not too large, counting sort is the best choice. Then each pass over n d-digited numbers takes $\theta(n+k)$. Such passes are d. Therefore, radix sort takes $\theta(d(n+k))$ time in total.
- When d is constant and K = O(n), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

Searching

- Searching is a process of locating a specific element among a set of elements.
- If the required element is found, the search is considered successful. Otherwise, it is unsuccessful.

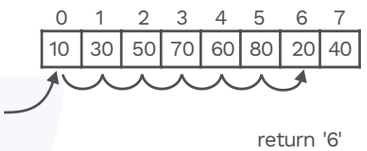


Linear Search

- Given an array arr[ ] of n elements and a search element 'x' to be searched in arr[ ]
- Begin with the leftmost element of arr[ ] and compare x with each element of arr[ ] one by one.
- Return the index if x matches an element.
- Return -1 if x does not match any of the elements.

Example:

Find 20, i.e., x = 20.



Time complexity of the linear search is written as O(n)

Note:
Binary search and hash tables are used more significantly in practice compared to Linear search since they allow faster searching.

Binary Search

- Binary search is one of the most efficient search algorithms available.
- It is based on the divide-conquer strategy.

Note:
Binary search algorithms can be applied only on sorted arrays.

- As a result, the elements must be arranged in a certain order.
- If the elements are numbers, use either ascending or descending order.
- If the elements are strings, use dictionary order.

To use binary search on an array that is not sorted.

- Sort the array first using the sorting technique.

- After that, employ the binary search algorithm.

Binary search (A, low, high, key)

if high < low

return (low-1)

$$mid \leftarrow \left\lfloor low + \left(\frac{high - low}{2} \right) \right\rfloor$$

if key = A[mid]

return mid

else if key < A[mid]

return Binary search (A, low, mid-1, key)

else

return Binary search (A, mid+1, high, key)

- Every call to the binary search algorithm is dividing the array into two equal half's and comparing the key with middle element. Based on the comparison with middle element, recursing to (left/right) half.

$\therefore T(n) = T(n/2) + C$

By master method

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

- O(1) space in case of iterative implementation, and O(logn) recursion call stack space in case of recursive implementation.

Chapter Summary

Sorting Algorithms	Time Complexcity			Space Complexity Worst Case
	Best Case	Average Case	Worst Case	
Bubble Sort	0(n)	$\theta(n^2)$	$0(n^2)$	0(1)
Insertion Sort	0(n)	$\theta(n^2)$	$0(n^2)$	0(1)
Selection Sort	$0(n^2)$	$\theta(n^2)$	$0(n^2)$	0(1)
Merge Sort	0(nlogn)	θ(nlogn)	0(nlogn)	0(n)
Quick Sort	0(nlogn)	θ(nlogn)	$0(n^2)$	0(n)
Heap Sort	0(nlogn)	θ(nlogn)	0(nlogn)	0(n)

Sorting Algorithms	In-place	Stable
Bubble Sort	Yes	Yes
Insertion Sort	Yes	Yes
Selection Sort	Yes	No
Merge Sort	No	Yes
Quick Sort	Yes	No

- The algorithm that can sort a dynamic input added while sorting is called online sorting algorithm.
  Eg: Insertion sort algorithm.
- The application and implementation details play a big role in determining which algorithm is best.
- Quicksort out performs heapsort, it is practically used for sorting large input arrays since its worst case time complexity is 0(nlogn).
- If stability and space issues are considered, then merge sort is the best choice.
- When the input array is nearly sorted or the input size is small, insertion sort is preferable.

3 Greedy Techniques

Greedy Algorithms

Algorithms for optimisation often follow a number of steps, each with a set of options.

- Dynamic programming is used to solve numerous optimisation challenges.
- Greedy Algorithms select a path that is optimal at that moment, expecting it leads to an optimal solution for the entire problem in future.
- A greedy algorithm won't work all the time.

Greedy algorithms have two main components: Greedy choice property and optimal substructure.

Greedy choice property:

The greedy-choice property is the first essential component.

Selecting locally optimal solutions which may lead us to global optimal solutions.

In other words, we choose the choice that appears to be the best in the current solving problem by not taking into account the results of subproblems.

Optimal substructure:

- If the global optimal solution includes the local optimal solutions, the problem has an optimal substructure. It is very important in both dynamic programming and greedy paradigms.

Huffman Codes

- Huffman coding compresses data very effectively.
- Consider the information as a string of characters.
- Huffman's algorithm finds an optimal way of expressing a character in binary. It gives less number of bits to characters with higher frequency and more number of bits to characters with lower frequency.

Example:

Let's say there exists a file with 130 characters. The characters and their respective frequencies in the file are given below:

Character	A	B	C	D	E	F	G
Frequency	7	11	12	19	21	25	35

Solution:

For 7 characters, we need a minimum of 3 bits to represent.

Character	A	B	C	D	E	F	G
Frequency	7	11	12	19	21	25	35
Fixed length Codeword	000	001	010	011	100	101	111

Variable length codeword:

Character	A	B	C	D	E	F	G
Frequency	7	11	12	19	21	25	35
Fixed length Codeword	0110	0111	010	110	111	00	10

For fixed length number of total bits = (7 + 11 + 12 + 19 + 21 + 25 + 35) × 3 = 390 bits

For variable length number of total bits = 4×7 + 4×11 + 3×12 + 3×19 + 3×21 + 2×25 + 2×35 = 348 bits.

Prefix codes:

- Prefix codes are codes which do not have the same prefix. These are used to encode and decode the characters in an optimal way.

Constructing a Huffman code:

- Huffman designed the Huffman code, a greedy approach for constructing an optimised prefix code.

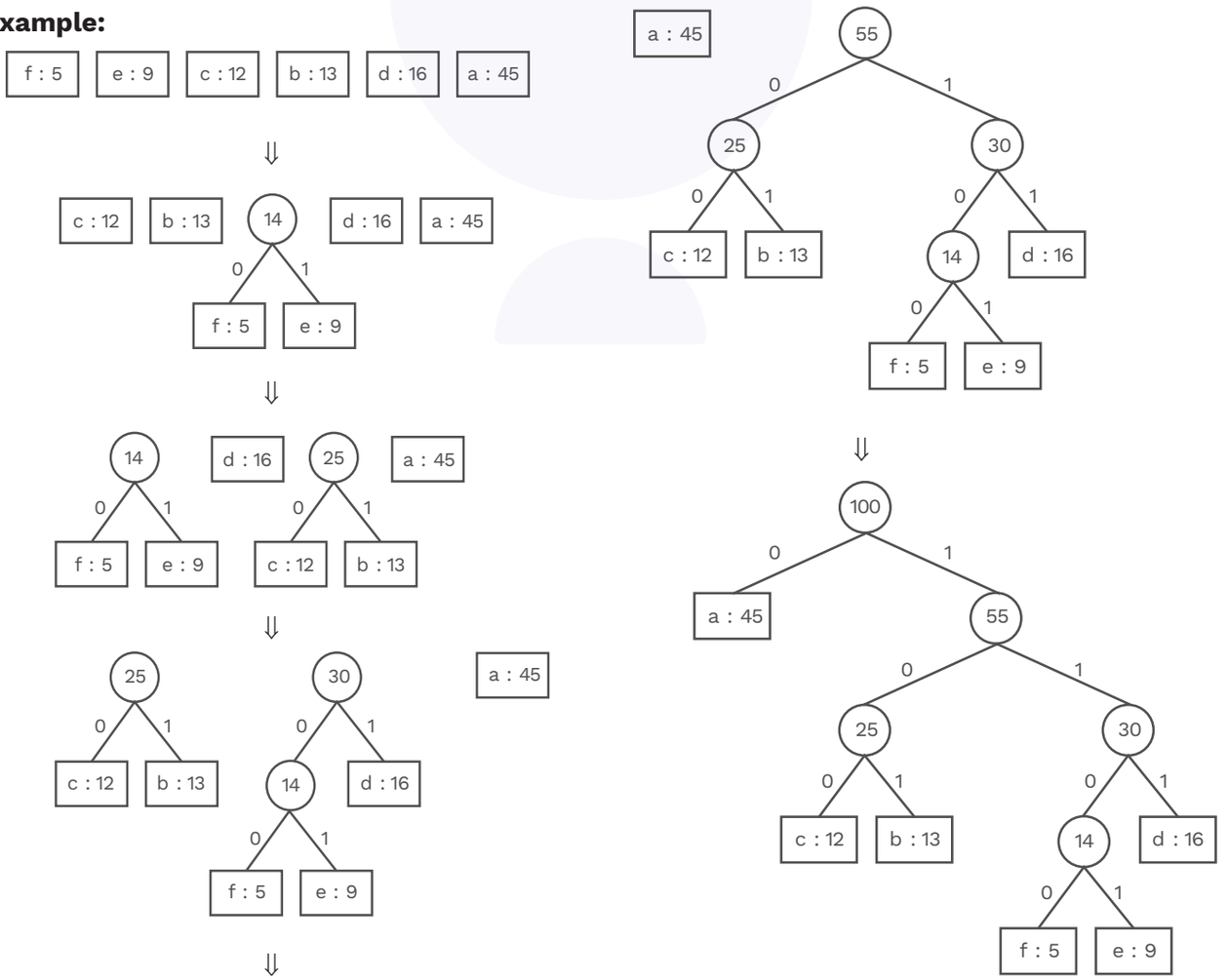
Pseudocode:

Huffman(C) // C represents the set of n character

1. n = |C|
2. Q = Create a min priority queue and make the string C inserted into it
3. for i = 1 to n–1
4. allocate a new node z
5. z.left = x = EXTRACT_MIN(Q)
6. z.right = y = EXTRACT_MIN(Q)
7. z.freq = x.freq + y.freq
8. INSERT (Q,z)
9. return EXTRACT_MIN(Q) // returns the root of the tree.

- In the pseudocode above, we assume that C is a character set of size n.
- Where character c ∈ C denotes an item with the c.freq attribute indicating its frequency.
- In a bottom-up approach, the algorithm constructs the tree T that corresponds to the optimal code.
- To produce the final tree, it starts with a set of |C| leaves and executes a series of |C|-1 "merging" operations.
- All the frequencies are in min priority Queue Q. The algorithm identifies two characters with the least frequency, removes their freq in min priority Queue, sums up the two frequencies then adds them to the Priority queue, which works based on the freq attribute.

Example:



- The ideal prefix code is represented by the final tree.
- The straightforward path from the root to the letter is the letter's codeword.

Analysis:

- We assume that the Queue is implemented as a binary min heap in the algorithm.
- In line 2, the Queue Q is intialised in O(n) for a character set of n characters (using BUILD_MIN_HEAP).
- From lines 3-8, the for loop iterates for exactly n-1 times, and each heap operation takes O(logn), which results in O(nlogn) time.
- Hence, the total time taken is O(nlogn), where n is the number of characters present in the character set C.

Previous Years' Question

A message is made up entirely of characters from the set X = (P, Q, R, S, T). The table of probabilities of each character is shown below:

Character	Probability
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	1.00

A message of 100 characters over X is encoded using Huffman coding. Then the excepted length of the encoded message in bits is __ **[2017 (Set-2)]**

(A) 225 (B) 226
(C) 227 (D) 228

Solution: (A)

Previous Years' Question

Consider the string abbccddeee. Each letter in the string must be assigned a binary code satisfying the following properties:

1. For any two letters, the code assigned to one letter must not be a prefix of the code assigned to the other letter.

2. For any two letters of the same frequency, the letter which occurs earlier in the dictionary order is assigned a code whose length is at most the length of the code assigned to the other letter.

Among the set of all binary code assignments which satisfy the above two properties, what is the minimum length of the encoded string? **[2021 (Set-2)]**

(A) 21 (B) 23
(C) 25 (D) 30

Solution: (B)

Fractional Knapsack Problem

- Let's assume a thief went to rob a store containing n items. Let w_i be the weight of the ith item and v_i is the value of the ith item.He has a knapsack that can handle a weight of w (integer).
- The items can be picked in fractions, and the problem is to find the set of items whose weight sums up to a weight w and whose value is maximum.

Example:

- When a thief walks into a store, he notices the following items.

Items	A	B	C
Cost	100	10	120
Weight	2 pounds	2 pounds	3 pounds

Knapsack holds, i.e (w) = 4 pounds

- Since, the thief can take a fraction of an item

$$\text{Solution} = \begin{cases} \text{2 pounds of item A} \\ + \\ \text{2 pounds of item C} \end{cases}$$

Value = 100 + 80
= 180

Greedy solution for fractional knapsack:

- Given a set of items I.

Weight	w_1	w_2	...	w_n
Cost	C_1	C_2	...	C_n

Let P denote the problem of selecting items from I, with weight limit K, such that the resulting cost (value) is maximum.

(1) Calculate $v_i = \frac{C_i}{w_i}$, for i = 1, 2..., n

(2) Sort the items by decreasing v_i. Let the sorted item sequence be 1,2,...,i,...n and the corresponding v and weight be v_i and w_i, respectively.

(3) Let k be the current weight limit (initially, k = K)

- In each iteration, we choose item i from the head of the unselected list; if k > = w_i, we take item i and update k = k–w_i, then consider the next unselected item.
- If k < w_i, we take a fraction f of item i, i.e., we only take f = $\frac{k}{w_i} (< 1)$ of item I, which weights exactly k. Then the algorithm is finished.
- The algorithm can take an item in fraction, i.e., an item need not be considered completely.

Previous Years' Question

Consider the weights and values of items listed below. Note that there is only one unit of each item.

Item number	Weight (in kgs)	Value (in Rupees)
1	10	60
2	7	28
3	4	20
4	2	24

The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximised. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt} A greedy algorithm sorts the items by their value to weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. The value of V_{opt}-V_{greedy} is

Range [16 to 16] **[NAT]**

Time complexity:

- The sorting of all items in decreasing order of their cost/weight ratio is the most time-consuming step. It takes O(n logn) time.
- Iterating through every item in step (3) takes O(n) time if the items are already arranged in the required order.

Total time complexity = O(nlogn) + O(n) = O(nlogn)

Job Sequencing Problem

- Given a list of jobs, each with its own deadline and associated profit if the jobs are completed before the deadline.
- Given that each job takes a single unit of time, the shortest deadline possible for any job is 1.
- When only one job can be scheduled at a time, how can the total profit be maximized?

Example:

Input:

Jobs	A	B	C	D
Deadline	1	4	1	1
Profit	10	20	30	40

Output:

The following is a list of jobs in order of maximum profit.

D, B

Algorithm:

1. First sort (in decreasing order) the profit of given the input.
2. Iterate on jobs in order of decreasing profit.

For each job follow this process:

- Find a time slot i, such that slot is empty and i < deadline and i is a job with the current greatest profit. Keep the job in this slot and mark this slot filled.

Time complexity:

- For sorting the jobs in decreasing order of profit O(nlogn)
- For each job, checking the open time slot between deadline and 0 is O(n)
  for 'n' jobs $\rightarrow$ n * O(n)
  $= O(n^2)$

$\therefore$ Total time complexity = $O(nlogn) + O(n^2)$
$= O(n^2)$

Representation of graphs:

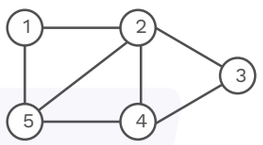
There are two common ways to express a graph G=(V,E): adjacency lists and adjacency matrices.

Adjacency list representation:

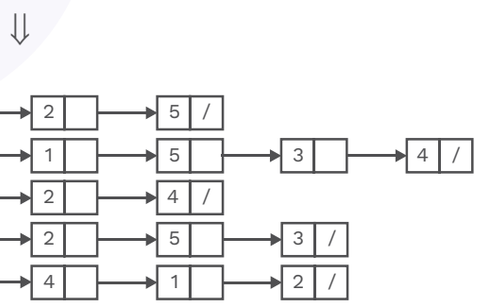
The Graph G(V,E) is expressed as an array Adj of size |V|, one for each element of Adj is a list for each vertex in V.

- For each vertex u in the set of vertices V, there exists an entry in the array Adj i.e. Adj[u] with list of all vertices adjacent to u.

i.e. Adj[u] has links of all the vertices one by one adjacent to u in G.



// An undirected graph G with 5 vertices and 7 edges



- We can readily adapt adjacency lists to represent weighted graphs.
- Simply store weight w(u,v) of the edge (u,v) $\in$ E with vertex v in u's adjacency list.

Weighted graphs:

Definitions

Graphs in which a weight w is given to each edge (u, v).

- If the length of an entry in an adjacency list is given by the number of vertices linked in that entry, then the sum of all

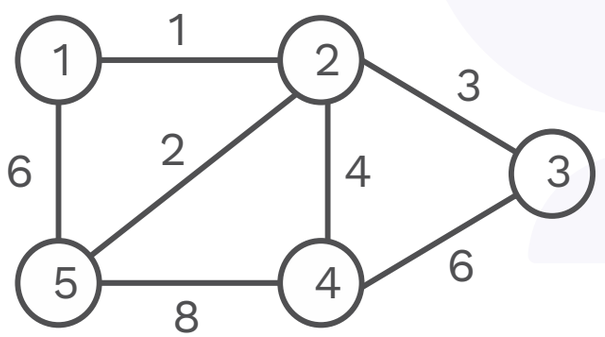
lengths of entries in the adjacency list is |E| for the directed graph and 2|E| for the undirected graph.

- Since an edge (u,v) is represented both in u and v entry for the undirected graph and only in u entry for the directed graph.
- Both the directed and undirect graphs need Θ(V+E) memory for their adjacency list representation.
- One of the disadvantages of the adjacency list is its O(V) time complexity to search whether an edge (u,v) exists in the graph or not.

Adjacency-Matrix representation:

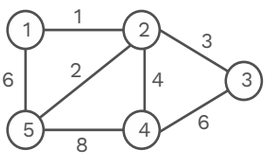
- In an adjacency matrix, a graph G(V,E) is represented as a matrix represented in |V| x |V| size.

$$a_{ij} = \begin{cases} 1 \text{ if } (i,j) \in E, \\ 0 \text{ otherwise} \end{cases}$$



The adjacency-matrix representation of G.

- An adjacency matrix of a graph requires $\Theta(V^2)$ space, independent of the number of edges in the graph.
- An adjacency matrix can represent a weighted graph also.
- For example, if G(V,E) is a weighted graph with edge-weight function w, we can simply store the weight w(u,v) of the edge (u,v) ∈ E as the entry in a row u and column v of the adjacency matrix.



⇓

	1	2	3	4	5
1	0	1	0	0	6
2	1	0	3	4	2
3	0	3	0	6	0
4	0	4	6	0	8
5	6	2	0	8	0

Minimum Spanning Trees

Minimum spanning tree T is a subset of the set of Edges E in a connected and undirected graph G(V, E) such that it forms an acyclic graph with

$w(T) = \sum_{(u,v)\in T} w(u,v)$, where w(T) is minimised.

- Since T is acyclic and covers(spans) all the vertices in the graph, it is known as the minimum spanning tree of Graph G.
- The problem of finding subset Tree T in a set of Edges E such that it covers all the vertices and gives minimum total weight is called the minimum spanning tree problem.

Note:
A complete graph can have maximum nn-2 spanning trees, where n is the number of nodes.

Definitions

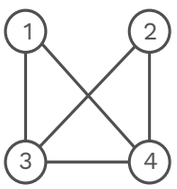
A complete graph is a graph in which each pair of graph vertices is connected by an edge.
A complete graph with n vertices is represented by symbol K_n.

- For non–complete graphs, we can use Kirchhoff's theorem to find number of spanning trees.

Kirchhoff's theorem:

1. For the given graph, create an adjacency matrix.
2. Substitute the degree of nodes for all the diagonal elements.
3. Substitute -1 for all non-diagonal 1's
4. Calculate the co-factor for any element.
   a) The total number of spanning trees for that graph is the cofactor we get.

Example:



// given graph G

1.

	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

// Adjacency matrix representation of graph G

2.

	1	2	3	4
1	2	0	1	1
2	0	2	1	1
3	1	1	3	1
4	1	1	1	3

// Substituting degree of the node for all the diagonal elements

3.

	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

// Substituting -1 for all non diagonal 1's

4. Co-factor of (1,1)
   = 2(9-1) – (-1) (-3-1) -1 (1+3)
   = 16 – 4 – 4
   = 8

∴ Total 8 spanning trees possible for given graph.

The two commonly used algorithms to solve minimum spanning tree problems are:

1. Kruskal's algorithm
2. Prim's algorithm

Kruskal's algorithm:

MST_KRUSKAL (G,w)

1. A = ϕ
2. for each vertex v $\in$ G.V
3. MAKE_SET(v)
4. For every edge (u, v) $\in$ G.E (Mentioned in increasing order by weight(u, v)):
5. if FIND-SET(u) $\neq$ FIND-SET(v):
6. A = A $\cup$ {(u, v)}
7. UNION(u, v) return A

- Set A is a set containing all the edges in the MST in the given graph, which is initially empty.
- The edge added first is always the least weighted edge in the entire graph.
- Kruskal's algorithm comes under the greedy algorithm since it chooses the least possible weight edge to add to the MST, at each step.
- **FIND_SET(u):** Operation that returns an element representing the set that is containing the vertex u.
- This operation helps in comparing whether two vertices u, v belongs to the same tree. (By equalising FIND_SET(u) and FIND_SET(v)).
- **UNION:** Operation used to combine 2 trees.
- Line 1 and Line 3 initialise the set A to empty and creates |V| individual trees for each vertex v $\in$ G.V.
- The for loop in the 5th line sorts edges in non-decreasing order.
- An edge (u,v) is not added to forest A if u and v belong to the same tree since it forms a cycle.
- Otherwise, the edge is added to the forest (done by the 7th line), and the two trees of vertices u and v are merged.

Time complexity analysis:

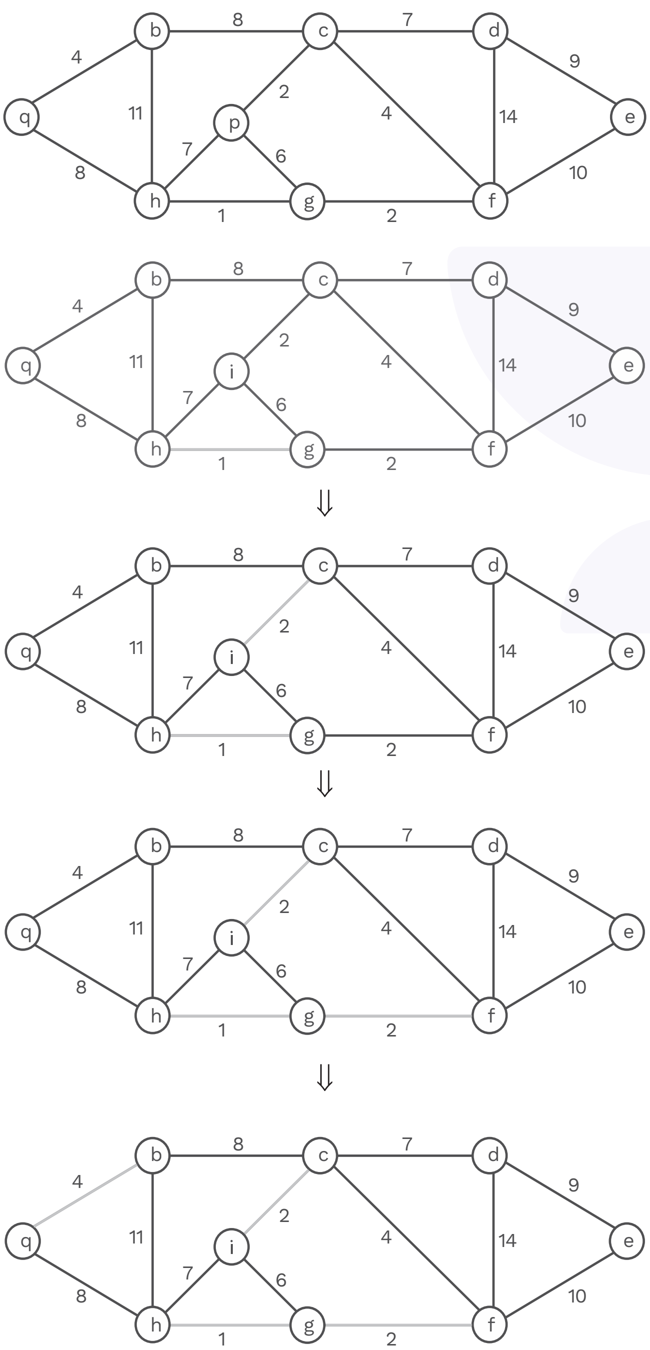
- Initializing set A to empty in Line 1 takes O(1) time.

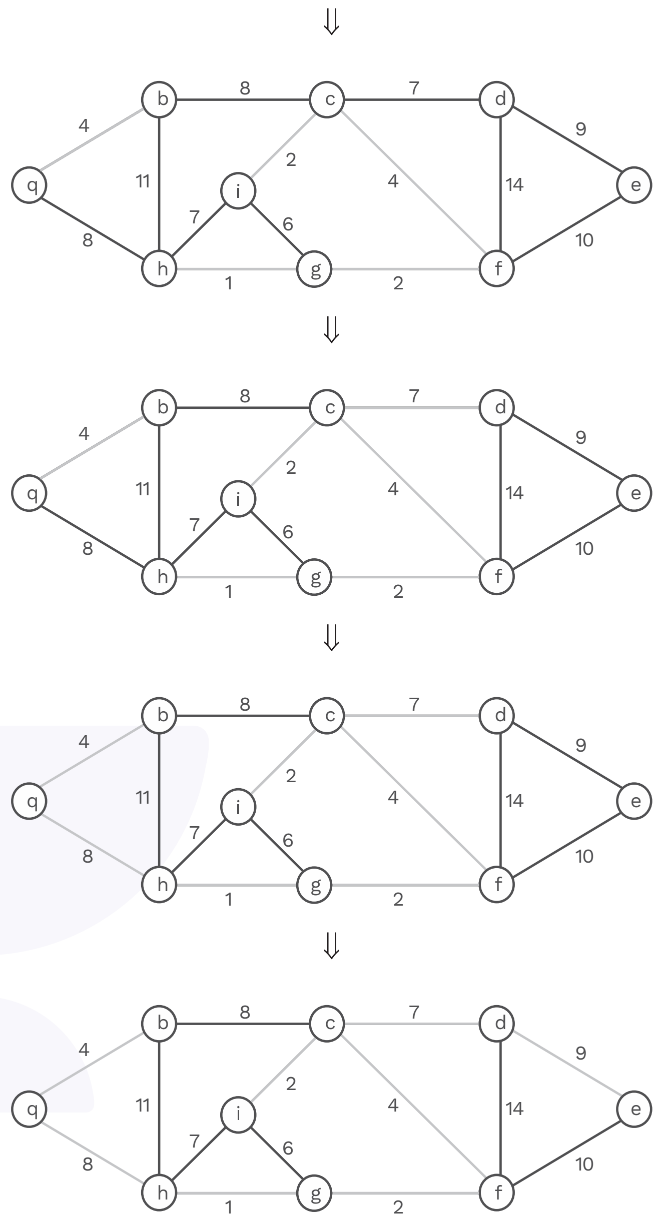
- Sorting edges takes O(ElogE) in line 4.
- For loop takes O(V) in 2nd and 3rd lines.
- For loop takes O(ElogV) to perform O(E) UNION and FIND_SET operations on disjoint_set forest in 5th and 8th lines.

∴ T(n) = O(1) + O(V) + O(ElogE) + O(ElogV) = O(ElogE) + O(VlogV) = O(ElogE) = O(Elog(V(V-1)/2)) = O(ElogV)

Example:

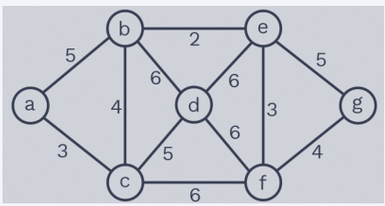
Applying Kruskal's algorithm on this graph gives,





Previous Years' Question

Consider the following graph:



Which one of the following is not the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

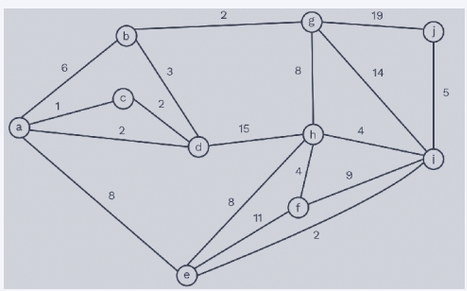
[2009]

(A) (b,e) (e,f) (a,c) (b,c) (f,g) (c,d)
(B) (b,e) (e,f) (a,c) (f,g) (b,c) (c,d)
(C) (b,e) (a,c) (e,f) (b,c) (f,g) (c,d)
(D) (b,e) (e,f) (b,c) (a,c) (f,g) (c,d)

Solution: (D)

Previous Years' Question

What is the weight ofa minimum spanning tree of the following graph?



(A) 29 (B) 31
(C) 38 (D) 41

Solution: (B)

Prim's algorithm:

PRIM_MST(G, w, r)

1. for each u ∈ G.V
2. u. key = ∞
3. u.π = NIL
4. r.key = 0
5. Q = G.V
6. while Q is not empty
7. u = Edge with minimum weight in Q
8. for each v ∈ G.Adj[u]
9. if v ∈ Q and w(u,v) < v.key
10. v.π = u
11. v.key = w(u,v)

- Prim's algorithm always assumes that the edges in set A form a single tree.
- The tree starts with a vertex v and grows till it spans all the vertices in the given Graph.
- At each step, a new edge is added to Tree A, which is the lightest among all the edges that are connected to the vertices in Tree A. This edge results in connecting an isolated vertex that is not included in Tree A.
- Prim's algorithm comes under the greedy algorithm since it selects a vertex that adds the minimum weight possible to the resulting minimum spanning tree.
- A connected graph G and a root r to be grown as a minimum spanning tree are the inputs given to the pseudocode above.
- The minimum priority queue Q contains all the vertices that are not in Tree A, based on key attributes.
- For a vertex v, v.key is the value of the minimum weighted edge among the edges connecting v to any other vertex in the tree. v.key=-∞, if no such edge exists.
- The attribute v.π names the parent of v in the tree
- Algorithm maintains the set A

$$A = \{(v, v \cdot \pi) : v \in V - \{r\} - Q\}$$

- When the algorithm terminates, the min-priority queue Q is empty
- The minimum spanning tree A for G is thus

$$A = \{(v, v \cdot \pi) : v \in V - \{r\}\}$$

- Lines 1-5 in pseudocode, make the key attribute of each vertex initialised to ∞ except the root. Since the root is the first vertex to be processed, its key is initialised to 0.

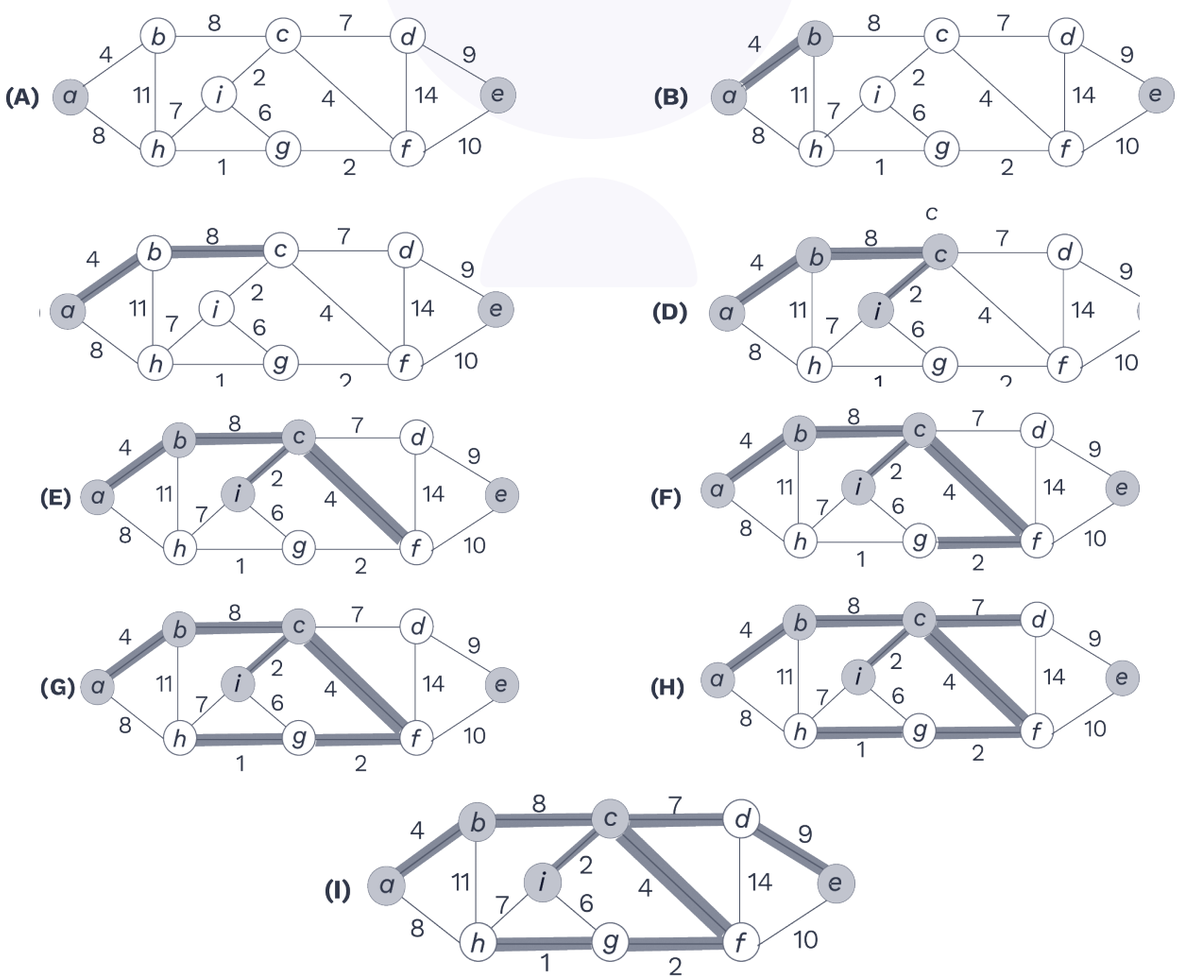
- Initialise each vertex's parent to NIL.
- Initialise the minimum priority queue Q with the set of all vertices in the graph.
- 7th Line finds the vertex u ∈ Q such that it is one of the vertices at the end of the light-weight edge that crosses the cut (V-Q, Q), except in the first iteration, in which u = r because of the 4th line.
- Deleting u from the set Q adds it to the set V-Q vertices in the tree, resulting in the addition of (u, u.π) to A.
- The for loop of the 8th to 11th lines updates the key and π attributes of each vertex v adjacent to vertex u but not the key and π attributes in the tree.

Running time of prim's algorithm:

- If we implement Q as a binary min-heap, BUILD-MIN-HEAP procedure to perform lines 1-5 in O(v) time.
- EXTRACT_MIN operation takes O(logV) time and is repeated for |V| times in the while loop.Therefore the total time is O(VlogV).
- The for loop in lines 8-11 executes O(E) times altogether since the sum of the lengths of all adjacency lists in 2|E|
- Within the for loop, line 9 takes constant time.
- The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap which is O(logv) time
- Thus, the total time for Prim's algorithms is with vertices 'V' and edges 'E' O(vlogv + Elogv) = O(Elogv)

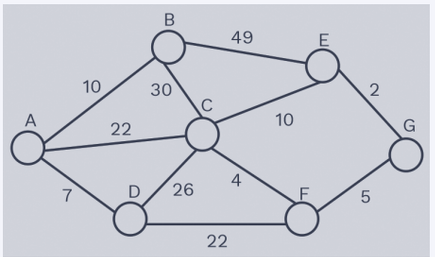
Note:
The running time of prim's algorithm by using the fibonacci heap is O(ElogV).

Example:



Previous Years' Question

Consider the undirected graph below:



Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree? **[2004]**

(A) (E, G), (C, F), (F, G), (A, D), (A, B), (A, C)
(B) (A, D), (A, B), (A, C), (C, F), (G, E), (F, G)
(C) (A, B), (A, D), (D, F), (F, G), (G, E), (F, C)
(D) (A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

Solution: (D)

Single Source Shortest Path

- Let G = (V, E) be a weighted digraph, with real-valued weight W assigned to every edge E.
- The weight w(p) of path P = $<v_0, v_1, v.....,v_k>$ is the sum of the weights of its constituent edges

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest-path weight $\delta(u,v)$ from u to v

$$\delta(u,v) = \begin{cases} \min\{w(p): u \overset{p}{\rightsquigarrow} v\}, & \text{if there is a path from u to v,} \\ \infty, & \text{otherwise} \end{cases}$$

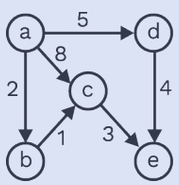
- A shortest path from vertex u to vertex v is then defined as path P with weight w(P) = $\delta(u,v)$.

The problem:

Let G (V,E) be the given digraph with positive edge weights. There exists a source vertex (distinguished from other vertices) S∈V from which the shortest path to every other vertex in the Graph needs to be found out.

Note:

Any subpath of a shortest path must also be a shortest path
eg:



δ(a,e) = 6 i.e(a,b,c,e)
δ(a,e) is a shortest path

Relaxation:

- For each vertex v ∈ V, maintain an attribute v.d which is an upper bound on the weight of the shortest path from source s to v, called as the shortest-path estimate.
- Initialise the estimates of the shortest path and predecessors by the following takes θ(v) time.

INITIALISE-SINGLE-SOURCE (G,S)

1. For each vertex v ∈ G.V
2. v.d = ∞
3. v.π = NIL
4. s.d = 0
   a) After initialisation, we have v.π = NIL for all v ∈ V, s.d = 0, and v.d = ∞ for
   b) v ∈ V – {s}
   c) An edge (u,v) is said to be relaxed after testing given that particular edge can improve the shortest path found so far. If yes, then v.d and v.π are updated.
   d) The following code performs a relaxation step on edge (u,v) in O(1) time

RELAX (u, v, w)

1. if v.d > u.d + w (u,v)
2. v.d = u.d + w(u,v)
3. v.π = u

Dijkstra's algorithm:

- Dijkstra's algorithm is a single source shortest path algorithm.
- It maintains a set of vertices S apart from the set of all vertices V, that keeps track of vertices to which the shortest path is found.
- The algorithm selects a vertex u ∈ V-S with the minimum path estimate, adds the vertex u to S and relaxes all the edges leaving from u.

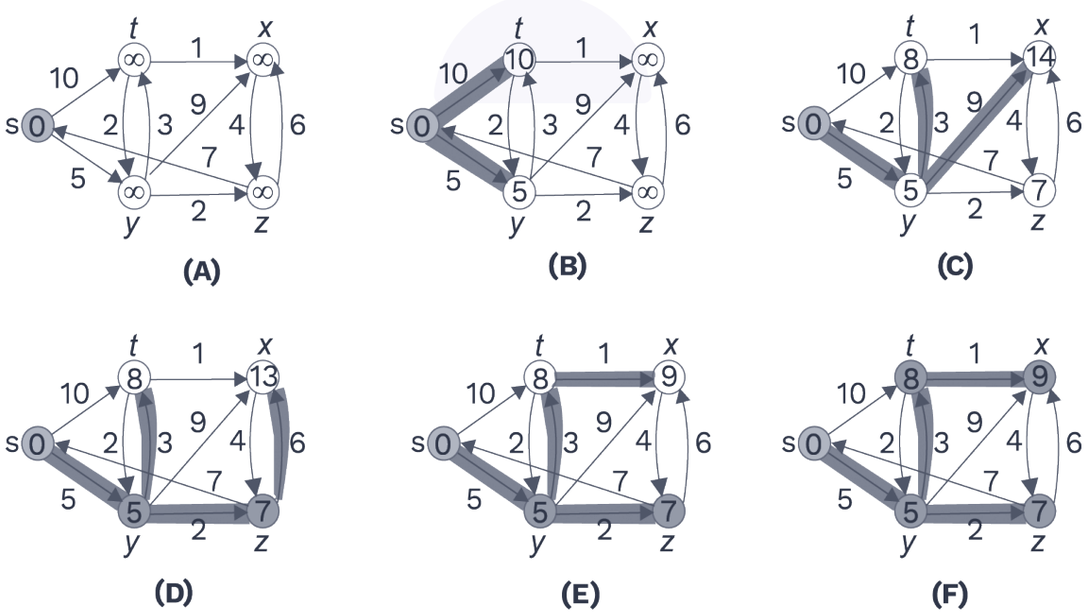
DIJKSTRA (G, w, s)

1. Initialise the source s
2. S = ϕ
3. Q = G.V
4. While Q ≠ ϕ
5. u = EXTRACT-MIN(Q)
6. S = S U {u}
7. For each vertex v ∈ G.adj[u]
8. RELAX (u, v, w)
   a) Because Dijkstra's algorithms always choose the "lightest" or "closest" vertex in V-S to add to set S, it falls into a greedy strategy.
   b) Min-priority queue Q of vertices, keyed by their d values, is used in the above algorithm.

Analysis of Dijkstra's algorithm:

- The initialisation uses only O(n) time,
- Each vertex is processed exactly once so condition in line 4 and EXTRACT-MIN are called exactly once for every vertex, i.e. V times in total.
- The inner loop for each V∈Adj[u] is called once for each edge in the graph.
- Each call of the inner loop does O(1) work plus, possibly, one decrease-key operation.
- Recalling that all the priority queue operations require O(log|Q|) = O(logV) time
- Thus, we have
  = O(V+V+VlogV +ElogV)
  = O((V+E) logV)

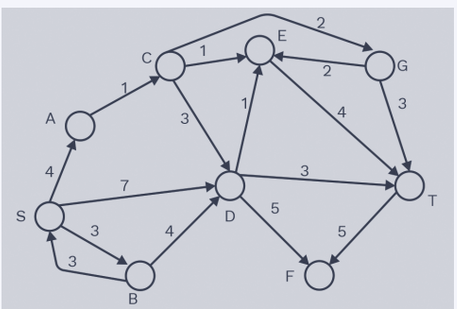
Example:



Previous Years' Question

Consider the directed graph shown in figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to vis discovered.

[2012]



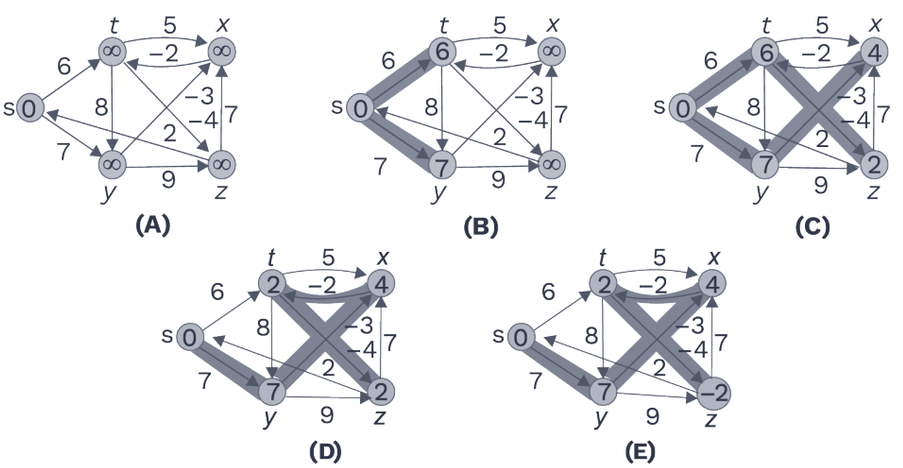
(A) SDT (B) SBDT
(C) SACDT (D) SAC ET

Solution: (D)

The Bellman-Ford algorithm:

- The Bellman-Ford algorithm is a single-source shortest path algorithm. It works on digraphs with negatively weighted edges, also.
- The Bellman-Ford algorithm results in a boolean value indicating whether a negative weight cycle is present or not in the graph.

Example:



- If there is no negative weighted cycle, then the algorithm produces the shortest path weights.
- Bellman–Ford algorithm assumes that a vertex v can be reached from vertex u with at most |V|-1 edges which result, in no negative edge cycle.
- It finds the shortest path involving 1 edge at first, then two and so on |V| -1 edges.
- If the cost found in involving |V| passes and |V|-1 passes are the same, then there is no negative edge cycle present.
- The edges are relaxed repeatedly, decreasing the estimate v.d on the weight of the shortest path to all the vertices from the source s until the actual shortest-path weight $\delta(s,v)$ is achieved.

BELLMAN-FORD (G, w, s)

1. INITIALISE_SINGLE_SOURCE(G, s)
2. For i = 1 to (number of vertices) - 1
3. For every edge (u, v) $\in$ G.E
4. RELAX (u, v, w)
5. For every edge (u, v) $\in$ G.E (Edges of graph)
6. If v.d > u.d + w (u, v)
7. Return FALSE
8. Return TRUE

- The Bellman-ford algorithm runs in time O(VE), since the initialisation in line 1 takes $\theta(v)$ time, each of the |V|-1 passes over the edges in lines 2-4 takes $\theta(E)$ time, and the for loop of lines 5-7 takes O(E) time.

Previous Years' Question

What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of n vertices?

(A) $0(n^2)$ (B) $0(n^2 \log n)$ (C) $0(n^3)$ (D) $0(n^3 \log n)$

Solution: (C)

Solved Examples

1. Consider the weights and values of items listed below.

Item number	Weight (in kgs)	Value (in Rupees)
1	1	2
2	4	28
3	5	25
4	3	18
5	3	9

The task is to pick a subset of these items by using greedy method such that their total weight is no more than 15 kgs and their total values is maximized.
The total value of items picked by the greedy algorithms is denoted by V_{greedy}"
The value of V_{greedy} is ------

Solution: 80

Now, sort the object in non-increasing order of value/weight ratio.

Item No.	Weight (in kg)	Value (in rupees)	Value/ weight
2	4	28	7
4	3	18	6
3	5	25	5
5	3	9	3
1	1	2	2

First we pick item 2 then item 4 then item 3 and at last item 5. Total capacity of this 4 object is 15 kg and total value is 80. So, overall profit is 80.

2. Consider the weights and values of items listed below.

Item number	Weight (in kgs)	Value (in Rupees)
1	4	8
2	8	4

The task is to pick a subset of these items by using greedy method such that their total weight is no more than 4 kgs and their total value is maximized.
The total value of items picked by the greedy algorithm is denoted by V_{greedy}.
The value of V_{greedy} is ----

Solution: 8

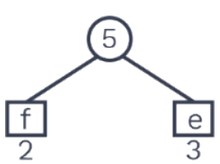
Here, the value / weight ratio of both the item is 2. So, we can take any item 1st.

Let's take item 1 first. Hence, the total capacity is going to be 4, and the profit is 8.

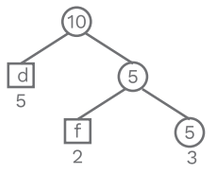
3. The characters a to f have the set of frequencies as shown below:
a:40, b:30, c:20, d:5, e:3, f:2
A Huffman code is used to represent the characters.
What is the weighted external path length?

Solution: 205

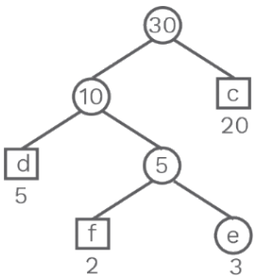
Step 1: $\overset{a}{40}, \overset{b}{30}, \overset{c}{20}, \overset{d}{5}, \overset{e}{3}, \overset{f}{2}$



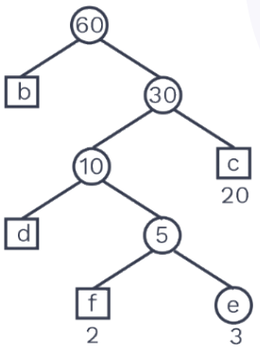
Step 2: a b c d
40, 30, 20, 5, 5



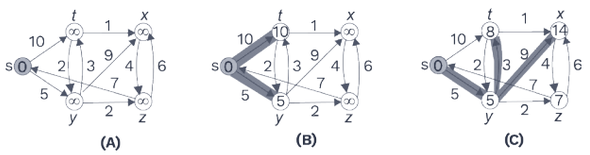
Step 3: a b c
40, 30, 20, 10



Step 4: a b
40, 30, 30



Step 5:



Weighted external path length = 1 × 40 + 30 × 2 + 4 × 5 + 20 × 3 + 2 × 5 + 3 × 5

= 40 + 60 + 20 + 60 + 10 + 15

= 205

4. The characters a to h have the set of frequencies given below:
a:2, b:2, c:3, d:4, e:6, f:9, g:14, h:22
A Huffman code is used to represent the characters.
What is the sequence of characters corresponding to the following code?
11000001011011000100101001 1

(A) hdegfcab

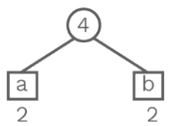
(B) abcdefgh

(C) hdegcfab

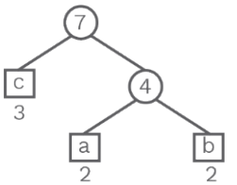
(D) hdgefcab

Solution: (A)

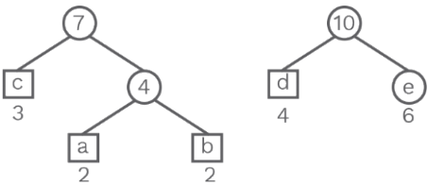
a b c d e f g h
2 2 3 4 6 9 14 22



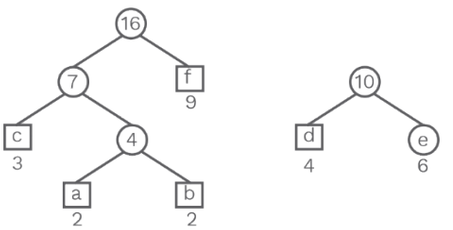
c d e f g h
4 3 4 6 9 14 22



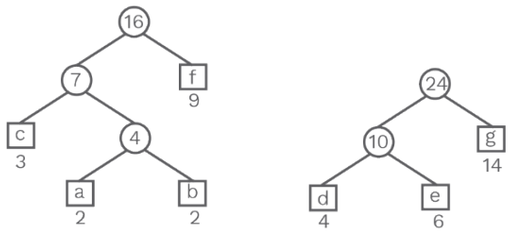
d e f g h
7 4 6 9 14 22



f g h
7 10 9 14 22

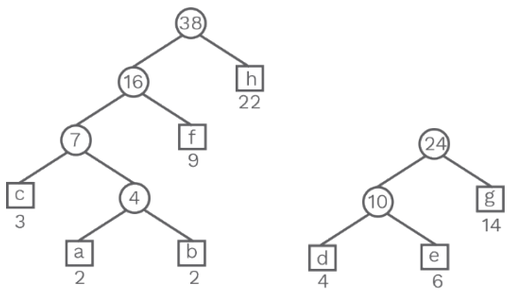


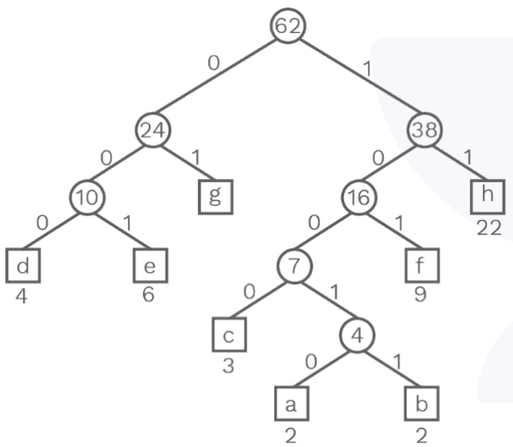
g h
10 16 14 22



h

16 24 22





So, the Huffman code is

d: 000
e: 001
g: 01
h: 11
f: 101
c: 1000
a: 10010
b: 10011

Group the string into characters from right to left:

11 000 001 01 101 1000 10010 10011

h d e g f c a b

5. If job J = (J_1, J_2 , J_3 , J_4) are given their processing time T_i = (1, 1, 1, 1) profit p; = (6, 8, 5, 10) and deadline are D_i = (2, 1, 1, 2). Maximum how many job can be done?

(A) 1

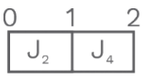
(B) 2

(C) 3

(D) All

Solution: (B)

The gantt chart is shown below:



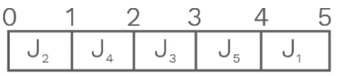
Profit = 8 + 10 = 18

6. If job J = (J_1, J_2 , J_3 , J_4 , J_5, J_6) are given their processing time T; = (1, 1, 1, 1, 1, 1) profit p, = (200, 180, 190, 300, 120, 100) and deadline are D, = (5, 3, 3, 2, 4, 2). What is the maximum profit?

Solution: 990

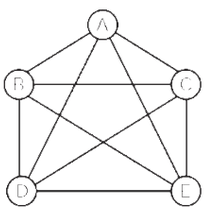
	J_1	J_2	J_3	J_4	J_5	J_6
Deadline	5	3	3	2	4	2
Profit	200	180	190	300	120	100

The gantt chart is shown below:



Profit = 990

7. What is the number of spanning tree possible from the given graph?



Solution: 125

Since it is a complete graph. So, the number of spanning trees for a complete graph k_n is $n^{(n-2)}$, (where n = no. of vertices)

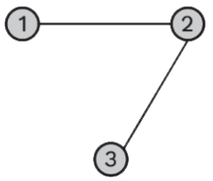
Here, n = 5

So, the number of spanning trees = $5^{(5-2)}$

= $5^{(3)}$

= 125

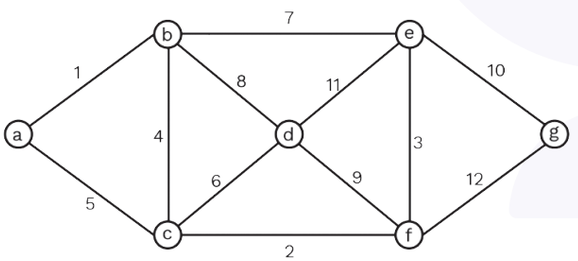
8. What is the number of spanning tree possible from the given graph?



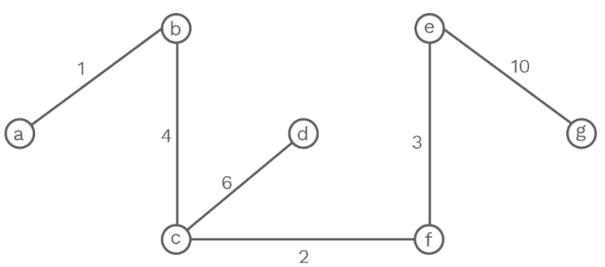
Solution: 1

The given graph is itself a spanning tree. So, only 1 spanning tree in possible.

9. Consider the undirected graph given below. What is the minimum possible weight of a spanning tree T ?



Solution: 26



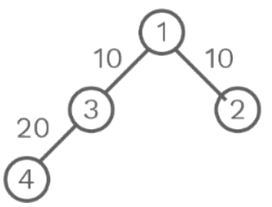
This is a minimum cost spanning tree.

10. Consider a complete undirected graph with vertex set {1, 2, 3, 4}. Entry W;i in the matrix W below is the weight of the edge {i,j}. What is the minimum possible weight of a spanning tree T?

$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 10 & 50 \\ 10 & 0 & 40 & 30 \\ 10 & 40 & 0 & 20 \\ 50 & 30 & 20 & 0 \end{bmatrix} \end{array}$$

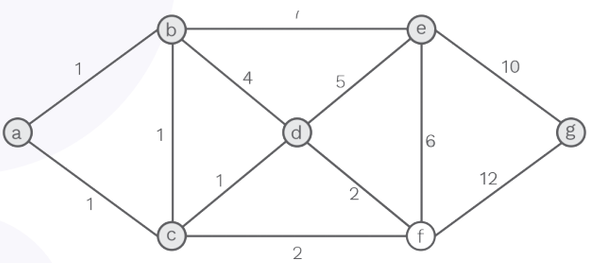
Solution: 40

The minimum spanning tree in tree is shown below:



Minimum cost of spanning tree = 40.

11. Consider the undirected graph below:



Using Prim's algorithms to construct a minimum spanning tree starting with node A, which one of the following sequences of node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree? (MSQ)

(A) (a, b), (a, c), (c, d), (c, f), (d, e), (e, g)

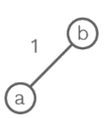
(B) (a, c), (a, b), (c, d), (c, f), (d, e), (e, g)

(C) (a, b), (a, c), (c, d), (d, f), (d, e), (e, g)

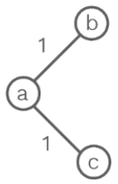
(D) (a, b), (a, c), (b, c), (d, f), (d, e), (e, g)

Solution: (A), (B) & (C)

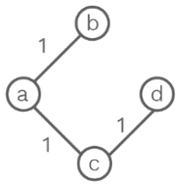
Step 1: We can either connect (a, b) or (a, c). We are first connecting ab.



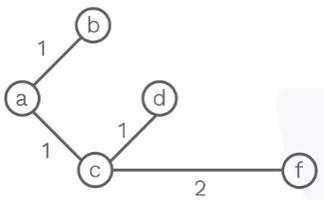
Step 2: Connecting (a, c) as shown below.



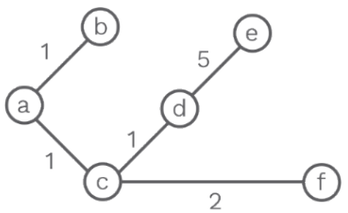
Step 3: Connect (c, d)



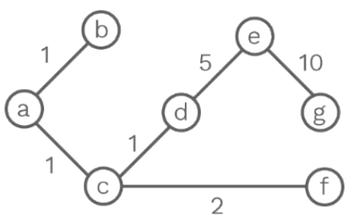
Step 4: Connect (c, f)



Step 5: Connect (d, e)

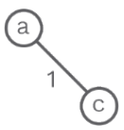


Step 6: Connect (e, g)

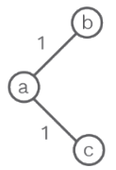


In this graph, we can get another sequences of edges as well to construct a minimum spanning tree, as shown below:

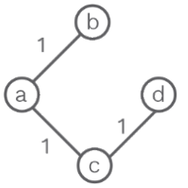
Step 1: First connect (a, c)



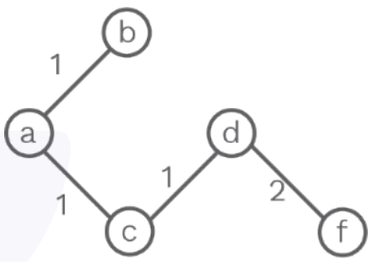
Step 2: Connect (a, b)



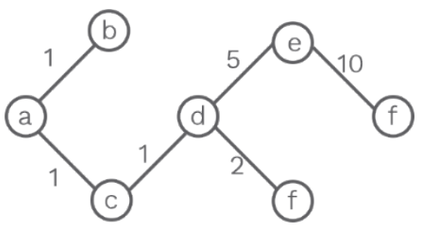
Step 3: Connect (c, d)



Step 4: Connect (d, f)



Step 5: Connect (d, e)



Similarly, we will get another sequence as

(a, b), (b, c), (c, d), (d, f), (d, e), (e, g).

12. What is the time complexity of Prim's algorithm with and without min heap respectively, (where E = edge and V = vertices)

(A) $O(ElogV^2)$ and $O(V2)$

(B) $O(V^2)$ and $O(V^3)$

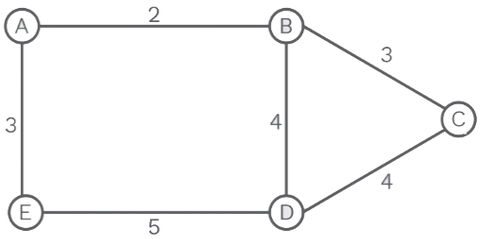
(C) $O(ElogV)$ and $O(V^3)$

(D) $O(VlogV + E)$ and $O(V^2)$

Solution: (A)

The time complexity of Prim's algorithm with and without min-heap is $O(ElogV)$ and $O(V^2)$.

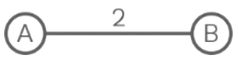
13. The number of distinct minimum spanning trees for the weighted graph below is ---



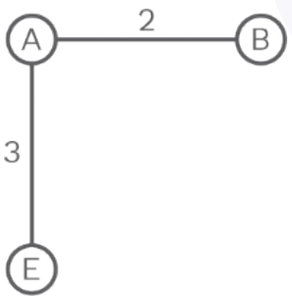
Solution: 2

By using Kruskal's algorithm we can add first (A, B) as show below:

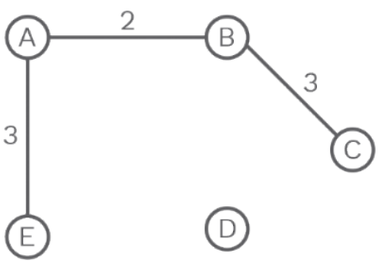
Step 1:



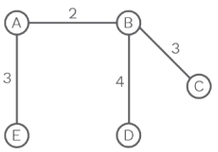
Step 2: Now add either (B, C) or (A, E). We are adding (A, E).



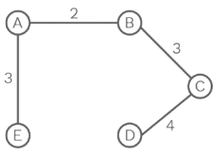
Step 3: Now, add (B, C)



Step 4: Now, we can add either (B, D) or (D, E). So, we will get two minimum spanning trees as shown below:



Or



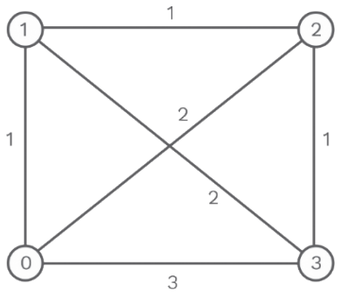
14. A complete, undirected, weighted graph 'G' is given on the vertex {O, 1, ... , n-1} for any fixed 'n'. Draw the minimum spanning tree of G if

(A) The weight of the edge (u, v) is lu - vi

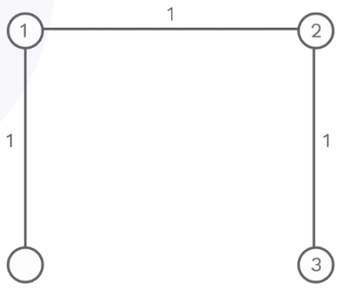
(B) The weight of the edge (u, v) is (u + v)

Solution: (A)

Let's take n = 4. So, the complete, undirected and weighted graph is shown below.

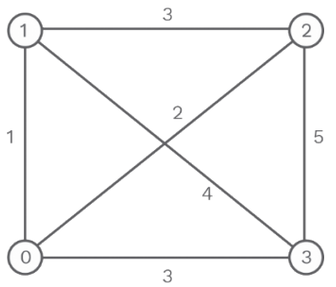


The minimum spanning tree is:

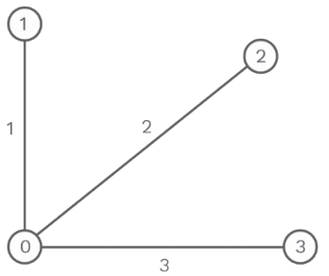


So, this is a line-graph. In this case, we will always get line graph.

(b) Similarly, for n = 4

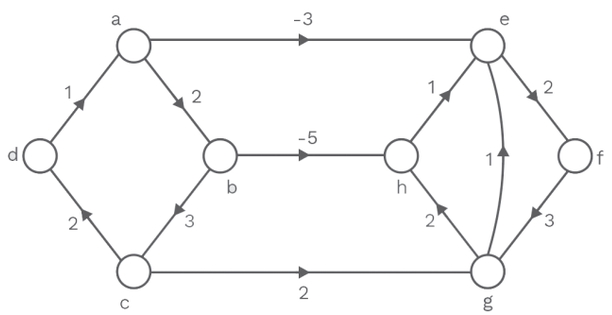


The minimum spanning tree for this graph is



This type of graph is called a star graph. Here, we will always get a star graph as a minimum spanning tree.

15. Find the single source shortest path node a to all other vertices. The graph is given below:



Solution:

	a	b	c	d	e	f	g	h
a	⓪	∞	∞	∞	∞	∞	∞	∞
e		2	∞	∞	(-3)	∞	∞	∞
f		2	∞	∞		(-1)	∞	∞
g		2	∞	∞			(2)	∞
b		(2)	∞	∞				4
h			(5)	∞				(-3)
c				(5)				
d								

The order in which single source shorted path computed is

a → e → f → g → b → h → c → d

Chapter Summary

Huffman Coding:

- Huffman's greedy algorithm builds an optimum manner of expressing each character as a binary string using a table that shows how often each character appears i.e., its frequency.

Applications of Huffman Coding:

- They're used by compression formats like gzip, and other.
- It is useful when there's a string of characters that appear frequently.

Fractional knapsack problem:

- Given n items worth value V; each and weight W; select items whose weight sums to a given value is w. The items can be taken in fractions.

Job sequencing problem:

- Given a list of jobs, each with its own deadline and associated profit if completed before the deadline.

Minimum spanning trees:

- MST is a subset of the set of Edges E in a connected and undirected graph G(V, E) such that it forms an acyclic graph.
- Difference between Prim's Algorithm and Kruskal's algorithm.

Prim's algorithm	Kruskal's algorithm
• It begins to construct the minimum spanning tree from any vertex in the graph. • The time complexity of Prim's algorithm is $O(V^2)$ where V is the number of vertices, and it can be improved to O(ElogV) using Fibonacci heaps. • Prim's algorithm returns a connected component and only works with connected graphs. • In dense graph, Prim's algorithm performs better	• The minimum spanning tree is built from the vertex in the graph with the least edge connected to it. • The time complexity of Kruskal's algorithm is O(ElogV), where V is the number of vertices • Kruskal's algorithm can both generate forests and work on disconnected components at any time • In sparse graphs, Kruskal's algorithm performs better.

Single-source shortest path:

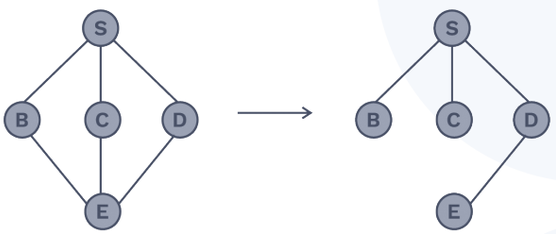
- It is a problem of finding shortest path from a source to all the vertices in a graph.
- Difference between Dijkstra's algorithm and Bellman Ford's algorithm.

Prim's algorithm	Kruskal's algorithm
• When there is a negative weight edge, Dijkstra's Algorithms does not work properly all the time. • The time complexity is O(ElogV). • It follows greedy approach, performs better	• When there is a negative weight edge, Bellman Ford's algorithm detects the negative weight cycle. • ts time complexity is O(VE). • It follows dynamic programming approach.

4 Graph-Based Algorithms

Breadth-First Search

- BFS is a simple algorithm used to traverse a graph, and it is a model used in many important graph algorithms.
- BFS can compute the distance(smallest number of edges) from s to other reachable vertices. It is the idea used behind Prim's minimum spanning tree and Dijkstra's single-source shortest path algorithms.
- BFS for a Graph G(V, E) explores the edges of G to "discover" each edge that is reachable from S, where V= set of vertices, E=Set of edges, S belongs to V, and S is the source vertex, and is distinguishable from other vertices.



- BFS uses white, grey, and black colours for the vertices to keep track the progress.
  White – not discovered and not explored
  Grey – discovered but not explored
  Black – discovered and explored
  Discovered – traversed the vertex
  Explored – the vertices that are reachable from a vertex V are discovered.
- The vertices that are adjacent to a black vertex will either be in black or grey.
- Grey vertices may have adjacent white vertices, and at a point, it acts as a border between white and black-coloured vertices
- BFS starts with source vertex S and then includes edges that connect S to its adjacent white vertices.
- For an edge (u,v) that is added to the BFS, u is the predecessor or parent to v in the Breadth-first tree.

Approach:

- The algorithm assumes that the graph G(V,E) is represented in an adjacency list, and the color of each vertex is stored in the u colour and the predecessors of u in u.π.(If there are no predecessors present for a vertex, S for example, then S.π =NIL).
- A queue Q is used to store the grey vertices.
- For a vertex u ∈ V, u.d stores the distance from the source to the vertex u.

```
BFS (G, s)
//initializing all vertices to white color, infinite distance and connected to null
 1. For every vertex u ∈ G.V – {s}
 2. u.color = WHITE
 3. u.d = ∞
 4. u.π = NULL
 5. s.color = GREY
 6. s.d = 0
 7. s.π = NULL
 8. Q = ∅
 9. ENQUEUE(Q,s):
10. While Q is empty
11. u = DEQUEUE(Q)
12. For each v ∈ G.Adj[u]
13. If (v.color == WHITE)
14. v.color = GREY
15. v.d = u.d + 1
16. v.π = u
17. ENQUEUE(Q, v)
18. u.color = BLACK
```

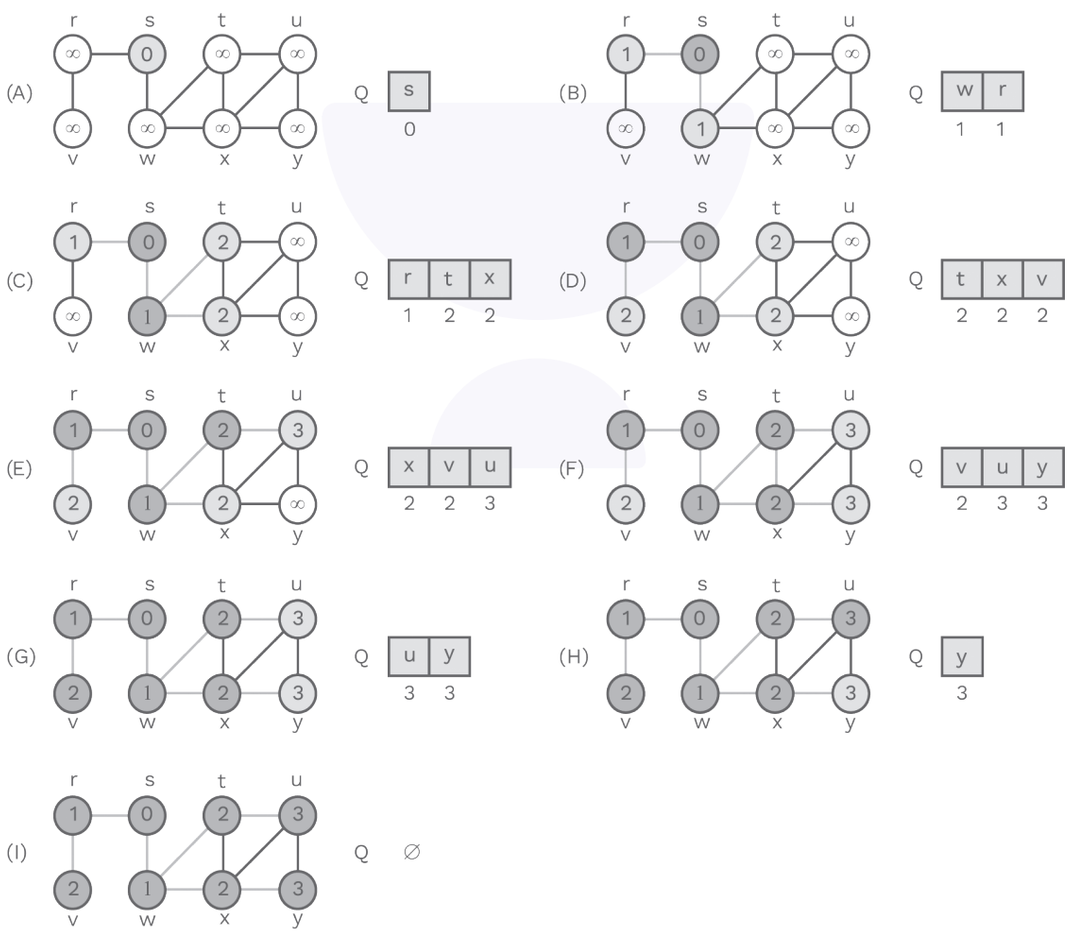
- Line 1-4 except the source vertex all the other vertex's colours are initialised to white, u.d vector to infinity, and u.π to null.
- Line 5 colours the source vertex to grey, line 6 initialises s.d to zero, and in line 7 s.π is initialised to null.
- In line 8, the queue Q is initially empty.
- Line 9 adds source vertex to the queue.
- Line 11 takes the first element in the queue into u, and for every vertex v adjacent to u (by means of for loop) it marks it into grey (line 14) if it is white (line 13) and intialises all the vertices v.d vector to u.d+1 (line 15), and predecessor of v, i.e., v.π as u(line 16).

- Line 17 adds each vertex discovered into queue and line 18 marks the explored vertex into black, i.e., u.colour = black.
- This procedure (from line 10 to line 18) is done until the queue gets empty.
- The order of vertices resulting from BFS depends on the order of adjacent vertices visited in line 12: the BFS may be different, but the distances computed by the algorithm do not.

Analysis

- All the vertices in the graph are white at first except the source S, and the vertices are not changed into white further in the algorithm, and vertices are enqueued and dequeued only once, which is tested in line 13.
- The enqueuing process and dequeuing processes of a vertex take O(1) time which in turn takes O(V) time.
- The algorithm scans the adjacency list of each vertex which takes O(E) time.
- Since initialisation takes O(V) time, the total time taken is O(V+E).
- Hence, BFS runs in time equal to the size of the adjacency-list representation of G, i.e., O(V+E).

Example:



Traversal vs. Search

- Traversal of a graph or tree involves examining every node in the graph or tree.
- Search may or may not involve all the nodes in visiting the graph in systematic manner.
- Traversal is a search, which involves all the nodes in the graph.
- By using Breadth-First Search (BFS), we can even find out whether a graph is connected or not.

- We can initialise the visited array of all the vertex to be zero (i.e., visited [i to n] = 0) and then run the breadth-first search starting from one of the vertexes, and after the search finishes, we should examine the visited array.
- In case if visited [ ] array indicates that some of the vertices are not visited, then the reason is definitely that the graph is not connected.
- Therefore, by using breadth-first search, we can say that whether the graph is connected or not.

Breadth-First Traversal

BFT can be executed using BFS.

Algorithm:

```
BFT (G, n) /*G is the graph, and 'n' is the number of vertices */
{
    for i = 1 to n do
            visited [i] = 0; /* visited[] is a global array of vertices. '0' value indicate it is not visited and '1' indicate it is visited.*/
    for i = 1 to n do
        if(visited [i] == 0) then
                BFS(i);
}
```

- For the time complexity of breadth-first traversal (BFT), we have to do an aggregate analysis.
- Aggregate analysis considers overall work done.
- In case if we are going to use adjacency list representation, then every node is going to have an adjacency list.
- Whenever we call BFS, then some part of the adjacency list will be completely visited, exactly one.
- Next time, when we call BFS on the remaining nodes, then the remaining nodes which are present on this list will be also visited exactly one.
- Therefore, overall, on average, all the nodes will be visited exactly one.
- In case of the undirected graph, the adjacency list contains 2E nodes.
- For initialisation of visited [ ] array, it takes O(V) time.
- Therefore, the total time complexity = O(V + 2E)
  = O(V + E)
- The time complexity is the same for both the directed graph as well as for undirected graph.
- Space complexity is also going to be the same as Breadth-First Search.

Conclusion

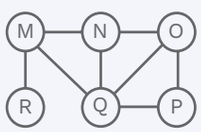
- The time and space complexity of breadth-first traversal is the same as breadth- first search.
- For a given graph,BFT calls BFS on every node.When BFS is called on a single node ,that means we are working on smaller part of the graph and then continue with the remaining part.
- So, it is as good as running the Breadth-First search on the entire graph exactly once.

Applications of Breadth First Traversal

- Shortest path and minimum spanning tree for weighted graph
- Path finding
- To test if a graph is bipartite
- Finding all nodes within one connected component
- Cycle detection in an undirected graph
- Social networking websites
- Crawlers in search engines

Previous Years' Question

The Breadth-First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



(A) MNOPQR (B) NQMPOR
(C) QMNPRO (D) QMNPOR

Solution: (D) **[2017 (Set-2)]**

Depth First Search (DFS)

- Depth-first search (DFS) is an algorithm for searching the vertices of a graph (traversing a graph to be specific), that works by exploring as far into the graph as possible before backtracking.
- **Input:**
  Graph G(V,E), where V is the set of vertices and E is the set of edges.
- To keep track of progress, a depth-first search colours each vertex
  - **White:** Vertices are white before they are discovered
  - **Gray:** Vertices are grey when they are discovered, but their outgoing edges are still in the process of being explored.
  - **Black:** Vertices are black when the entire subtree starting at the vertex has been explored
- The moment DFS discovers a vertex v in an adjacency list of already visited u,the algorithm marks the predecessor of attribute v. π=u.
- Depending upon the source vertex, there will be many subtrees possible for a DFS.
- The predecessor subgraph of a DFS is defined as

$$G_\pi = (V, E_\pi), \text{ where}$$
$$E_\pi = \{(V.\pi, V) : v \in V \text{ and } v.\pi \neq \text{NULL}\}$$

- The predecessor subgraph of DFS forms a forest with several depth-first trees.The edges in E.π are tree edges.
- DFS timestamps each vertex apart from creating a depth-first tree.The two timestamps given to a vertex:v.d is used to record when the vertex is discovered(changes v to grey colour) and v.f is used to assign appropriate finishing time after examining v's adjacency list (changes v to black).
- Since the adjacency matrix has at most $|V|^2$ entries,the timestamps range between 1 and $|V|^2$. The timestamp of discovering the vertex and finishing the vertex are v.d and v.f such that v.d<v.f, for every vertex $v \in V$.
- Vertex u is WHITE before u.d, GREY between u.d and u.f, and BLACK after u.f.
- The code below is DFS with Graph G(directed or undirected) as input. Time is a global variable.

```
DFS (G):
1.  For each vertex u ∈ G.V
2.  u.color ← WHITE
3.  u.π ← NULL
4.  Time ← 0
5.  For each vertex u ∈ G.V
6.  If u.color is WHITE
7.  DFS-VISIT(G,u)
end
DFS-VISIT(G,u)
1.  Time ← time + 1 // white vertex u has
    just been discovered
2.  u.d ← time
3.  u.color ← GREY
4.  For each v ∈ G.Adj[u] // explore edge
    (u,v)
5.  if v.color is WHITE
6   v.π ← u
7.  DFS-VISIT(G,v)
8.  u.color ← BLACK // blacken u; it is
    finished
9.  Time ← time + 1
10. u.f ← time
end
```

Analyzing DFS(G):

- All the vertices are coloured white and their π attributes are initialised to null in lines 1-3 of DFS (G).
- The time variable is reset in line 4.
- From line 5 to line 7, For any vertex $u \in V$ is applied DFS-VISIT(G,u) if it is white.
- For each time DFS_VISIT(G,u) is called on a vertex u, then u becomes the new root.
- This DFS-VISIT(G,u) returns the vertex with u.d and u.f initialised.

Analyzing DFS-VISIT(G,u):

- The global variable time is incremented in line, 1, and the new value of discovery time u.d is updated in line 2. The vertex is coloured grey in line 3.
- From the 4th to 6th line, the vertices that are adjacent to input vertex u are checked. If they are white, their predecessor, i.e., v.π, is initialised to u.
- Since every vertex adjacent to u is considered in line 4, DFS explores the edge (u,v) for $v \in$ Adj[u].

- After all the vertices adjacent to u are explored,8th line to 10th line in algorithm colours the vertex to black,increments time, and u.f is noted.

> **Note:**
> The order of vertices returned by DFS depends on the order of vertices discovered in line 5 of DFS algorithm, and line 4 of DFS-VISIT algorithm.

- Apart from the time to execute calls of DFS_VISIT,the loops in lines 1-3, and 5-7 in DFS gives Θ (V) time complexity.
- The algorithm DFS_VISIT discovers every vertex exactly once, and the vertex on which DFS_VISIT is called should be a white vertex, and the DFS_VISIT will colour it to grey at the very first step.
- The loop lines 4-7 execute |Adj[v]| times in the execution of DFS-VISIT algorithm. $\sum_{v \in V} |\, Adj[v] \,| = \Theta(E)$, the total time for executing lines 4–7 of DFS-VISIT is Θ (E). The total time taken by DFS is therefore Θ (V +E).

Depth-First Traversal

- Depth-first traversal is also exactly the same as Breadth-first traversal.
- Here, instead of calling BFS inside that traversal function, we will call DFS.
- The time and space complexity of depth-first Traversal and depth-first Search is same.

```
DFT (G, n)/* G is the graph & n is the number of vertices */
{
for i = 1 to n do
visited[i] = 0; // visited [] is an array
for i = 1 to n do
if(visited[i] == 0) then
DFS(i);
}
```

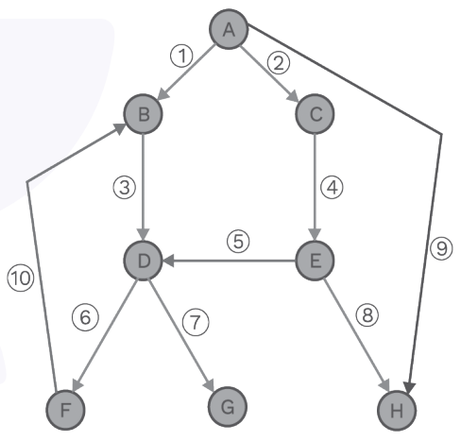
Conclusion

- Time complexity in case of adjacency list
  BFS = BFT = DFS = DFT = O(V + E)
- Time complexity in case of adjacency matrix
  BFS = BFT = DFS = DFT = $O(V^2)$
- Space complexity of
  BFS = BFT = DFS = DFT = O(V).

Applications of Depth First Search

- Detecting cycle in a graph:
  If there is a back edge in DFS, it has a cycle. Hence DFS can detect a cycle in a graph.
- Path finding:
  The DFS algorithm can be tweaked to find a path between two given vertices, u and z.
- Used as logic behind topological sorting
- To test if a graph is bipartite
- Finding strongly connected components of a graph

Triee edge, Back edge and Cross edges in DFS of graph



- **Tree Edge [Red edges]**
  Formed after applying DFS on a graph.
- **Forward Edge [vertex A to vertex H]**
  Edge (u,v), in which v is a descendent of u.
- **Back Edge [vertex F to vertex B]**
  Edges (u,v) in which v is the ancestor of u.
- **Cross Edge [vertex E to vertex D]**
  Edges(u,v) in which v is neither ancestor nor descendent to u.

Undirected graph

- In an undirected graph, forward, and backward edges are the same.
- Cross edges are not possible in an undirected graph.

Difference between DFS and BFS

Depth-First Search	Breadth-First Search
1. Backtracking is possible from a dead end. 2. Stack is used to traverse the vertices in LIFO order. 3. Search is done in one particular direction.	1. Backtracking is not possible. 2. Queue is used to traverse the vertices in FIFO order. 3. The vertices at the same level are maintained in parallel.

Topological sort:

- DFS is the logic behind topological sort in a directed acyclic graph (DAG).
- A topological sort of a DAG G=(V, E) is a linear ordering of all its vertices, such that if G contains an edge (u, v), then u appears before v in the ordering.
- If a cycle exists in a graph, there will be no linear ordering possible.

Topological Sorting(G)

Step 1: Call DFS on the graph, so that it calculates the finishing time of each vertex.

Step 2: Based on the finishing time, insert them into a linked list and return the list of vertices.

Analysis

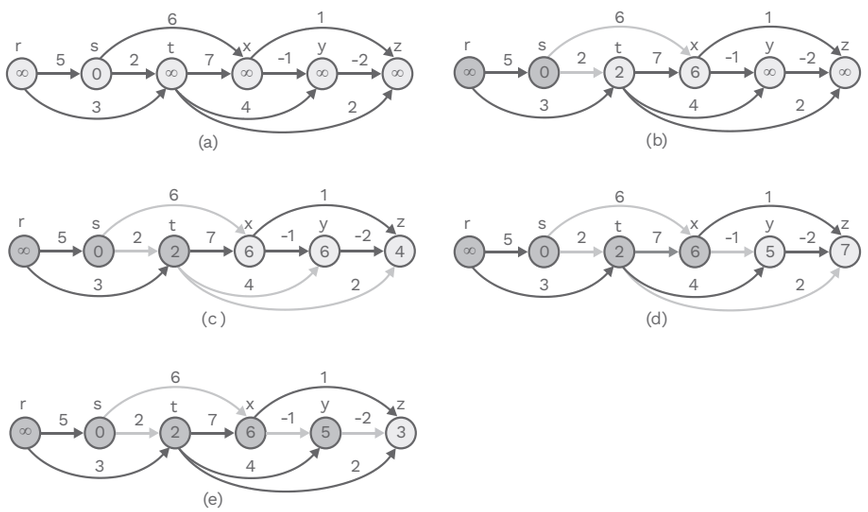
- DFS takes Θ (V + E) time, and O(V) to add all vertices one by one into the linked list.

1. Therefore, the topological sort takes O(V + E) time in total.
   DAG-SHORTEST-PATHS(G, w, s)
   a) sort the vertices of G topologically
   b) start with source
   c) for each vertex u, taken in topologically sorted order
   d) for each vertex v $\in$ G.Adj[u]
   e) RELAX(u, v, w)

Analysis

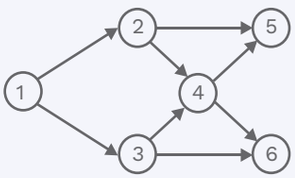
- The topological sort of line 1 takes Θ (V +E) time.
- The call of INITIALISE-SINGLE-SOURCE in line 2 takes Θ (V) time.
- The for loop of lines 3–5 makes one iteration per vertex.
- Altogether, the for loop of lines 4–5 relaxes each edge exactly once.
- Because each iteration of the inner for loop takes Θ (1) time.
- The total time is Θ (V +E).

Examples of a directed acyclic graph (DAG):



Previous Years' Question

Consider the DAG with consider V={1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering? **[2007]**

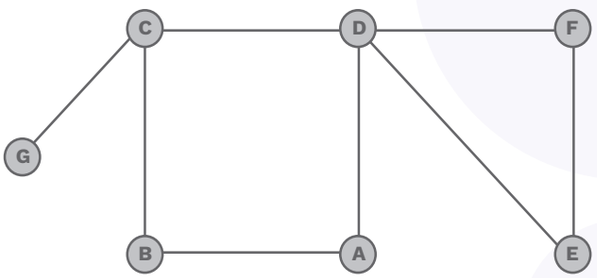


(A) 1 2 3 4 5 6
(B) 1 3 2 4 5 6
(C) 1 3 2 4 6 5
(D) 3 2 4 1 6 5

Solution: (D)

Solved Examples

1. Consider the following graph (G)



Number of cut vertex or articulation points is _____.

Solution: 2

In an undirected graph, a cut vertex (or articulation point) is a vertex and if we remove it then the graph splits into two disconnected components.

Removal of "D" vertex divides the graph into two connected components ({E, F} and {A, B, C, G}).

Similarly, removal of "C" vertex divides the graph into ({G} and {A, B, C, F}).

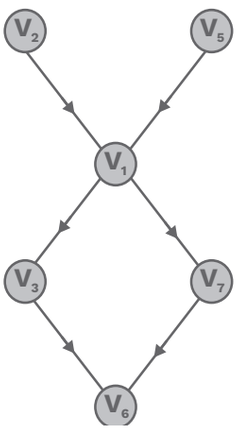
For this graph D and C are the cut vertices.

2. Topological sort can be applied to
(A) Undirected graph
(B) All types of graphs
(C) Directed acyclic graph
(D) None of the above

Solution: (C)

Topological sort can be applied to directed acyclic graphs.

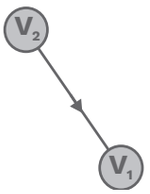
3. Consider the directed acyclic graph with V={V_1, V_2, V_3, V_5, V_6, V_7} shown below.



Which of the following is not a topological ordering?
(A) $V_2V_5V_1V_7V_3V_6$
(B) $V_5V_2V_1V_7V_3V_6$
(C) $V_1V_5V_2V_7V_3V_6$
(D) None of the above

Solution: (C)

Here, every edge has a dependency, like this



this edge means that V_2 comes before V_1 in topological ordering. Initially, V_2 and V_5 doesn't have any dependency. So any one of them can be done independently.

So, either start with V_2 or V_5.

So, the topological ordering is given below:

$V_2V_5V_1V_7V_3V_6$

Or

$V_5V_2V_1V_7V_3V_6$

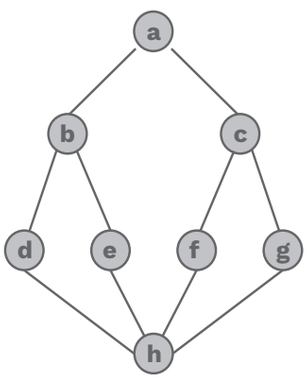
Another topological ordering is also possible, but $V_1V_5V_2V_7V_3V_6$ is not correct topological ordering because it starts with V_1 before V_2 or V_5.

Hence, (C) the correct option.

4. Consider the following sequence of nodes for the undirected graph given below.

1. a b d h e f c g
2. a c g h f e b d
3. a b d h f c g e
4. a b d h g c f e

If a Depth-First Search (DFS) is started at node a using stack data structure. The nodes are listed in the order they are first visited.



Which all of the above is/are not possible output(s)?

(A) 1,3, and 4 only

(B) 1 and 3 only

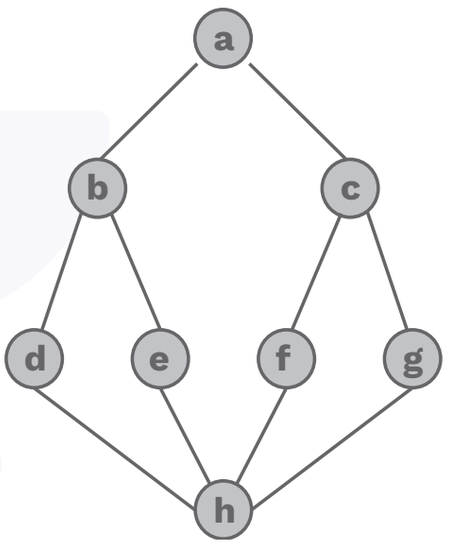
(C) 1, 2, 3, and 4 only

(D) None of the above

Solution: (D)

All the sequences of nodes are the possible output of Depts First Search (DFS).

5. The Breadth-First Search algorithm has been implemented using the queue data structure. The possible order of visiting the nodes of the following graph is (MSQ)



(A) abcdefgh

(B) acbfgdeh

(C) abcedgfh

(D) abchdefg

Solution: (A), (B), and (C)

The sequence of nodes given in options (a), (b), and (c) are the correct possible order of visiting the nodes by using breadth-first search, because breadth-first search visits the "breadth" first, i.e., if it is visiting a node, then after visiting that node, it will visit the neighbour nodes (children) first before moving on to the next level neighbours.

Chapter Summary

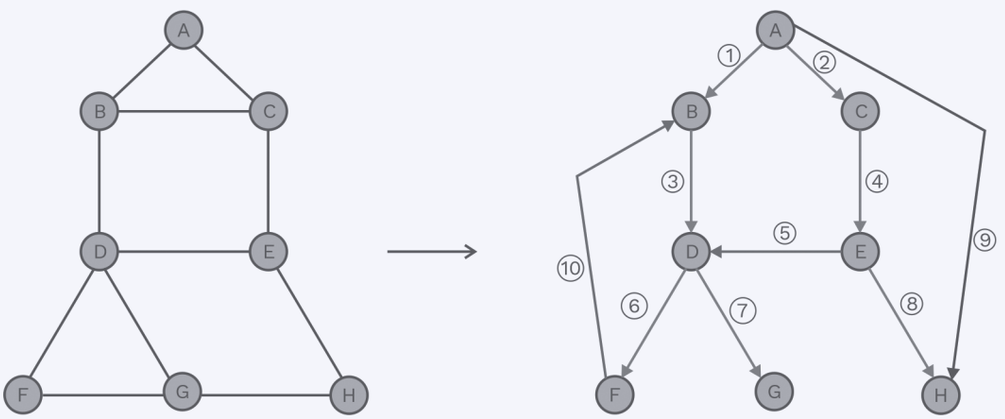
- **BFS** – Simple algorithm for traversing a graph (Breadth-wise).
- **Traversal Vs. Search** – Traversal goes through each vertex, but the search may or may not.
- **DFS** – Algorithm used for traversing a graph (depth-wise).

Difference between DFS and BFS

Depth First Search	Breadth-First Search
1. Backtracking is possible from a dead end. 2. Vertices from which exploration is incomplete are processed in a LIFO order. 3. Search is done in one particular direction.	1. Backtracking is not possible. 2. The vertices to be explored are organised as a FIFO queue. 3. The vertices at the same level are maintained in parallel.

- **Tree edge, Back edge, and Cross edge in DFS of a Graph:**

Tree edge: Formed after applying DFS on a graph.Red edges are tree edges.
Forward edge: Edge (u,v) in which v is a descendent of u.
eg: 9.
Back edge: Edges (u,v) in which v is the ancestor of u.
eg : 10.



Cross edge : Edges(u,v) in which v is neither ancestor nor descendent to u.
eg : 5

- **Topological Sort :** "A topological sort of a DEG G=(V, E) is a linear ordering of all its vertices, such that if G contains an edge (u, v), then u appears before v in the ordering."

5 Dynamic Programming

- By using dynamic Programming, we can avoid recalculating the same result multiple times.
- Using the divide and conquer technique; we divide bigger problems into simpler subproblems as the algorithm requires.
- Dynamic programming works on both top-down and bottom-up approach/ technique.
- We usually start with dividing the large task into smaller subtasks, storing the results of those subtasks to be used by another subtask at the later stages of the problem solving, until we get the desired solution.
- Using this technique complexity of various algorithms can be reduced.
- The greedy method only generates one decision sequence at a time, and dynamic programming takes care of all the possibilities.
- The problems with overlapping subproblems are generally solved by the dynamic programming technique; problems that do not have a similar type of pattern are difficult to solve using the dynamic programming technique.
- Dynamic programming follows four steps:
  - Identify the pattern of a suitable solution.
  - Recursively calculate the solution.
  - Use the bottom-up technique to calculate the value of the solution.
  - Build a suitable solution from the calculated result.
- Steps 1 -3 form the foundation of a DP technique.
- If we need only the value of a solution, then skip step 4.
- During step 4, we store additional information throughout step 3 so that we can easily calculate a suitable solution.

Understanding Dynamic Programming

Before jumping to problems, let us try to understand how dynamic programming works through examples.

Fibonacci series:

A series of numbers in which the present number is the sum of the last two numbers.

The Fibonacci series is defined as follows:

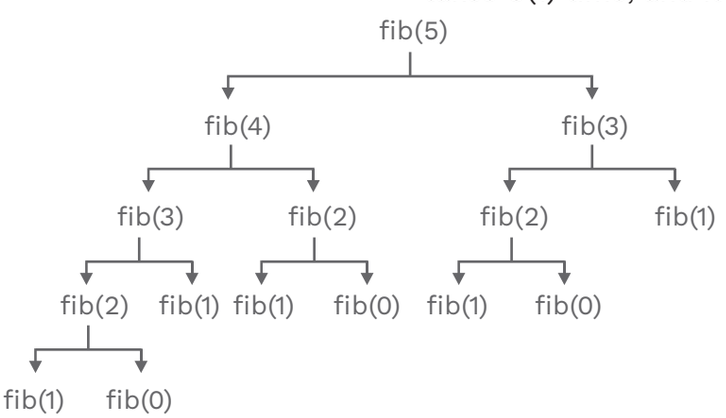
Fib(n) = 0, if n = 0

1, if n = 1

Fib(n-1) + Fib(n-2), if n > 1

Recurrence relation and time complexity:

- The recurrence call T(n) is divided into T(n-1) and T(n-2) till the base conditions T(1) and T(0).
- Therefore, the depth of the tree will be O(n).
- The number of leaves in a full binary tree of depth n gives 2^n. Since each recursion takes O(1) time, and it takes $O(2^n)$.



- In the above case, fib(2) was evaluated three-time (over-lapping of subproblems)
- If the n is large, then other values of fib (subproblems) are appraised which leads to an exponential-time algorithm.
- It is better, instead of solving the same problem, again and again, to store the result once and reuse it; it is a way to reduce the complexity.
- Memoization works like this: Begin with a recursive function and add a table that maps the parameters of the function to the results calculated by the function.
- When the function is called with identical parameter values that have already been called once, the stored value is returned from the table.

Improving:

- Now, we see how DP bring down this problem complexity from exponential to polynomial.
- We have two methods for this:
- 1. bottom-up approach: Start with the lower value of input and slowly calculate a higher value.

```
int fib[n];

int fib(int n)
{
    if (n < = 1)
        return n;
    fib[0] ← 0
    fib[1] ← 1
    for ← 2 to n
        fib[i] ← fib[i-1] + fib[i-2];
    return fib[n];
}
```

- 2. Top-down approach: We just save the values of recursive calls and use them in future.
- The implementation

```
int fib[n];

for (i ← 0 to n)
fib[i] ← 0; /* initialisation */

int fib (int n)
{
    if(n == 0) return 0;
    if(n == 1) return 1;
    if(fib[n]! = 0) /*check if already calculated*/
        return fib[n];
    return fib[n] ← fib (n-1) + fib (n-2);
}
```

- In both methods, Fibonacci Series complexity is reduced to O(n).
- Because we use already computed values from the table.
  TC: O(n).
  SC: O(n).

Note:
Both the approaches can be derived for the dynamic programming problems.

Matrix Chain Multiplication

Problem:

Let's say a series of matrices are given, $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, with their value, what is the right way to parenthesize them so that it provides the minimum the number of total multiplication. Assume that we are using standard matrix multiplication and not Strassen's matrix multiplication algorithm.

Input:

Chain of matrices $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, where A_i has dimensions or size $P_{i-1} \times P_i$. The value is specified in an array P.

Goal:

Finding a way of multiplying matrices(in parenthesized form) such that it gives the optimal number of multiplications.

Solution:

- We know the fact of mathematics that matrix multiplication is associative.
- Parenthesizing does not have any effect on the result.

- For example – 4 matrices A, B, C, and D, the number of ways could be:
  $(A(BC))D = ((AB)C)D = (AB)(CD)$
  $= A(B(CD)) = A((BC)(D))$
- Number of ways we can parenthesis the matrix = $\frac{(2n)!}{(n+1)!n!}$,
  Where n = number of matrix - 1
- Multiplying $A_{(p\times q)}$ matrix with $B_{(q\times r)}$ requires p*q*r multiplications, i.e. the number of scalar multiplication.
- Different ways to parenthesize matrix produce a different number of scalar multiplication.
- Choosing the best parenthesizations using brute force metthod gives $O(2^n)$.
- This time, complexity can be improved using Dynamic Programming.
- Let M[i, j] represent the minimum number of scalar multiplications required to multiply $A_i...A_j$.

$$M[i,j] = \begin{cases} 0 & \text{, if } i = j \\ \underset{i\le k<j}{\text{Min}}\{M[i,k] + M[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

- By using the above recursive equation, we find point k that helps to minimise the number of Scalar multiplications.
- After calculating all the possible values of k, the value which gives the minimum number of scalar multiplications is selected.
- For reconstructing the optimal parenthesization, we use a table (say, S[i, j]).
- By using the bottom-up technique, we can evaluate the M[i, j] and S[i, j].
  /* p is the size of the matrices, Matrix i has the dimension p[i-1] × p[i].
  M[i,j] is the least cost of multiplying matrices i through j.
  S[i ,j] saves the multiplication point, and we use this for backtracking, and length is the size of p array.*/
  Void MatrixChainOrder (int p[], int length)

```
{
    int n = length – 1, M[n] [n], S[n] [n];
    for i ← 1 to n
    M[i] [i] ← 0; // fills in matrix by diagonals
    for l ← 2 to n // l is chain length
    {
        for i ← 1 to n – l + 1                          {
            int j ← i + l – 1;
            M[i] [j] ← MAX VALUE;
            /* Try all possible division points i...k and k...j */
            for k ← i to j – 1
                {
                    int this.cost ← M[i] [k] + M[k+1] [j] + p[i-1] * p[k] * p[j];
                        if(this.cost <M[i] [j])
                            {
                                M[i] [j] ← this.cost;
                                S[i] [j] ← k;
                            }
                }
        }
    }
}
```

Time complexity = $O(n^3)$
Space complexity = $O(n^2)$.

Top-Down Dynamic Programming Matrix Chain Multiplication

- Top-down method is also called memoization or memoized.
- Both the top-down method and bottom-up method are going to use the same unique problem.
- The basic similarity between top-down dynamic programming and bottom-up dynamic programming is that the number of function, calls in top-down dynamic programming and the number of shells in bottom-up are almost the same.
- In the top-down method, we are not going to compute any function, twice. Whenever we compute a function, for the 1st time, we save it in the table and next time, when we want that function we take it from the table.

```
MEMOIZED_MATRIX_CHAIN(P)
{
    1. n = p . length – 1 /* p is a sequence of all dimensions, and n is equal to the number
                             of matrixces */
    2. Let m[1....n, 1....n] be a new table
                                        /* m[i, j] represents the least number of scalar
                                    multiplication needed to multiply Ai....Aj */
    3. for i = 1 to n
    4.       for j = 1 to n
    5.              m[i, j] = ∞
    6. return LOOKUP_CHAIN (m, p, 1, n)
}
    LOOKUP_CHAIN (m, p, i, j)
{
    7. if m[i, j] < ∞        /* these two line check whether it is visited for
                                                    the 1st time or not */
    8.       return m[i, j]
    9. if i == j
    10.      m[i, j] = 0
    11. else for k = i to j – 1
    12.      q = LOOKUP_CHAIN(m, p, i, k) + LOOKUP_CHAIN (m, p, k+1, j) + p_{i-1} p_k p_j
    13. if q < m[i, j]
    14.      m[i, j] = q
    15. return m[i, j]
}
```

Analysis:

- In lines 3 to 5, initially, it is placing infinity in the entire table.
- It is done so to know whether it is the 1st time program has visited some entry, or it has already computed.
- If the entry is ∞, then the it means that the program has visited that shell for the 1st time.

Time complexity:

- Number of distinct sub-problem is $O(n^2)$, and whenever the program call any distinct sub-problem, the for-loop in the worst-case run for 'n' times.
- Therefore, the time complexity is $O(n^3)$ same as the bottom-up method.
- Actual space complexity is more because of recursion, but the order of space complexity is not going to change.
- The depth of the tree is going to be O(n). When the depth of the recursion tree is O(n), then the stack size required is O(n) because, at any time, the maximum

number of elements that will be present in the stack will be equal to the depth of the recursion tree and $O(n^2)$ for table m[n, n]. Therefore, space complexity = $O(n^2) + O(n) = O(n^2)$

- Space complexity is also same as bottom-up method.

Previous Years' Question

Four matrices M_1, M_2, M_3 and M_4 of dimensions p x q, q x r, r x s and s x t respectively can be multiplied in several ways with different number of total scalar multiplications. For example, when multiplied as $((M_1 \times M_2) \times (M_3 \times M_4))$ the total number of multiplications is pqr + rst + prt.
When multiplied as $(((M_1 \times M_2) \times M_3) \times M_4)$, the total number of scalar multiplications is pqr + prs + pst.
If p = 10, q = 100, r = 20, s = 5 and t = 80, then the minimum number
of scalar multiplications needed is:
(A) 248000 (B) 44000
(C) 19000 (D) 25000
Solution: (C) **[GATE 2011]**

Longest common subsequence:

Given two strings X and Y of length m and n, respectively. Finding the longest common subsequence in both the strings from left to right, which need not be continuous. For example, if X = "ABCBDAB" and Y = "BDCABA", then LCS(X,Y) = {"BCBA", "BCAB"}. As we can see, there are several optimal solutions.

Brute force approach:

Subsequence X[1...m] (with m being the length of subsequence X) and if it is a subsequence of Y[1...n] (with n being the length of subsequence Y) checking this takes O(n) time, and there are 2^m subsequences possible for X. The time complexity thus sums up to $O(n2^m)$, which is not good for large sequence.

Recursive solution:

- Before DP Solution, we form a recursive solution for this, and later examine the LCS problem.
- Given two strings, "ABCBDAB" and "BDCABA", draw the lines from the letters in the first string to the corresponding letter in the second, no two lines should cross each other.

```
A  B  C  B  D  A  B
   |  |  |     |
   BDCAB     A
```

- Given X and Y, characters may or may not match.
- We can see that the first character of both strings are not same.
- Next, go to the second character in one of the strings and check.
- Lets go to the second character in the first string, and the remaining subproblem is followed this way, solving it recursively.

Optimal Substructure of An Lcs

- Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle Y_1, Y_2, ..., Y_n \rangle$ be sequences, and let $Z = \langle Z_1, Z_2, ..., Z_k \rangle$ be any LCS of X and Y.
- If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.
- This characterises that the LCS of two series contains within it an LCS of prefixes of the two series.
- LCS has optimal-substructure property as a problem.
- In recursive solutions, there are overlapping subproblems.

Recursive solution:

- The optimal substructure implies that we should examine either 1 or 2 subproblems when discovery LCS of $X = \langle x_1, x_2, ...,x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$.

- If $x_m = y_n$, then we solve 1 subproblem. Evaluating longest common subsequence on X_{m-1} and Y_{n-1}.
- If X_m ! = Y_n, then we must solve two subproblems. finding an LCS of X_{m-1} and Y and finding the LCS X and Y_{n-1}.
- Whichever of these two LCSs is longer is the LCS of X and Y; these cases exhaust all possibilities.
- One of the optimal subproblem solutions seems like inside a longest common subsequence of X and Y.
- As in the matrix chain multiplication problem, a recursive solution to the LCS problems requires to initiate a recurrence for the value of an optimal solution.
- c[i, j] = length of the LCS of the sequences $<x_1, x_2, ...x_i>$ and $<y_1, y_2, ...y_j>$
- If (i = 0 or j = 0) then one of the sequence has length 0, and LCS has length 0.
- The recursive formula comes from the optimal substructure of the LCS problem.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

When $X_i = Y_j$, we consider the subproblem of evaluating an LCS of X_{i-1} and Y_{j-1} else we evaluate two subproblems, LCS of X_i and $Y_{(j-1)}$, LCS of $X_{(i-1)}$ and Y_j

In the LCS problem; we have only O(mn) different subproblems, we can use DP to evaluate the problem.

Method LCS-LENGTH grab 2 sequences $X = <x_1, x_2, ..., x_m>$ and $Y = <Y_1, Y_2, ..., Y_n>$ as inputs.

- We store c[i, j] values in a table c[0..m, 0..n], and evaluate the entries in Row-Major order. (the procedure fills in the first row of C from left to right, then the second row, and so on).
- b[1...m, 1...n] helps us in constructing an optimal solution.
- b[i, j] points to the table entry corresponding to the optimal subproblem computed and selected when evaluating c[i, j].
- The procedure returns the table b and c, c[m, n] contains the length of an LCS of X and Y.

LCS_LENGTH(X, Y)

1. m ← X · length
2. n ← Y · length
3. Let b[1...m, 1...n] and c[0...m, 0...n] be new tables
4. for i ← 1 to m
5. c[i,0] ← 0
6. for j ← 0 to n
7. c[0,j] ← 0
8. for i ← 1 to m
9. for j ← 1 to n
10. if x_i == y_j
11. c[i,j] ← c[i-1, j-1] + 1
12. b[i,j] ← "↖"
13. elseif c[i-1, j] ≥ c[i, j-1]
14. c[i,j] ← c[i-1, j]
15. b[i,j] ← "↑"
16. else c[i,j] ← c[i, j-1]
17. b[i,j] ← "←"
18. return c and b

- The running time of the procedure is θ(mn), since each table entry takes θ(1) time to compute.
- Space complexity is θ(mn) because of tables.

Example:

Let X = <P, Q, R, Q, S, P, Q> and Y <Q, S, R, P, Q, P> be two sequences.

i		j	0	1	2	3	4	5	6
			Y_1	Q	S	R	P	Q	P
0	X_1		0	0	0	0	0	0	0
1	P		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	Q		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	R		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	Q		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	S		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	P		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	Q		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- The c and b are two dimensional matrices that stores the length of subsequence and the appropriate arrows respectively.
- Initialise the X row and Y column to zero and start with c[1,1] onwards.
- The length can be computed row wise or column wise.
- Let us consider row wise here:
  Starting with c[1, 1] P ≠ Q therefore
  c[1, 1] = max {c[0, 1], c[1, 0]} = 0
  b [1, 1] = "↑"
  c[1, 2] = max {c[1, 1], c[0, 2]} = 0
  b[1, 2] = "↑" since P ≠ S.
  c[1, 3] = max {c[0, 3], c[1, 2]} = 0
  b[1, 3] = "↑" since P ≠ R.
  c[1, 4] = c[0, 3] + 1 = 1,
  b[1, 4] = "↖" since P = P.
- Similiarly all the values in each row are calculated until c[7, 6].
- To find the possible subsequences possible, b[i, j] is used to reconstruct the subsequence from the table.
  PRINT-LCS(b, X, i, j)

  1. If i == 0 or j == 0
  2. Return
  3. If b[i, j] == "↖"
  4. PRINT-LCS(b, X, i-1, j-1)
  5. Print X_i
  6. Elseif b[i, j] == "↑"
  7. PRINT-LCS(b, X, i-1, j)
  8. Else PRINT-LCS(b, X, i, j-1)

- This procedure takes time O(m+n), since it decrements at least one of i and j each recursive call.

Previous Years' Question

A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of length m and n, respectively with indexes of X and Y starting from 0.
We wish to find the length of longest common sub-sequence (LCS) of X[m] and Y[n] is as l(m, n), where an incomplete recursive definition for the function l(i, j) to compute the length of the LCS of X[m] and Y[n] is given below:
l(i,j) = 0, if either i = 0 or j = 0
= expr1, if i, j>0 and X[i-1] = Y[j-1]
= expr2, if i,
j>0 and X[i-1] * Y[j-1]
Which one of the following options is correct?
(A) expr1 = l(i-1,j)+1
(B) expr1 = l(i,j-1)
(C) expr2 = max (l(i-1,j), l(i,j-1))
(D) expr2 = max(l(i-1,j-1),l(l,j))
Solution: (C) **[GATE 2009]**

Previous Years' Question

Consider the data given in the above question. The value of l(i,j) could be obtained by dynamic programming based on the correct recursive definition of l(i,j) of the form given above, using an array L[M, NJ, where M = m + 1 and N = n + 1, such that L[i,jJ = l(i,j).
Which one of the following statements would be true regarding the dynamic programming solution for the recursive definition of l(i,j)?

(A) All elements L should be initialized to O for the value of l(i, j) to be properly computed.

(B) The values of l(i, j) may be computed in a row major order or column major order of L[M, N]

(C) The values of l(i, j) cannot be computed in either row major order or column major order of L[M, NJ

(D) L[p, qJ needs to be computed before L[r, sJ if either p < r or q < s.

Solution: (B) **[GATE 2009]**

Previous Years' Question

Consider Two Strings A = "Qpqrr" And B = "Pqprqrp". Let X Be The Length Of The Longest Common Subsequence (Not Necessarily Contiguous) Between A And B And Let Y Be The Number Of Such Longest Common Subsequences Between A And B. Then X + 10y = ____

Solution: 34 **[Gate 2014 (Set-2)]**

Multistage Graph:

- A multistage graph G = (V,E) is a directed graph in which the vertices are partitioned into K ≥ 2 disjoint sets of V_i, 1 ≤ i ≤ k
- In addition, if < u, v > is an edge in E, then u is in V_i and v belongs to V_{i+1} for some i, 1 ≤ i < k.
- Let 's' and 't' be the source and destination respectively.
- The sum of the costs of the edges on a path from source (s) to destination (t) is the path's cost.
- The objective of the MULTISTAGE GRAPH problem is to find the minimum path from 's' to 't'.
- Each set V_i defines a stage in the graph. Every path from 's' to 't' starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.
- This MULISTAGE GRAPH problem can be solved in 2 ways.
  - Forward Method
  - Backward Method

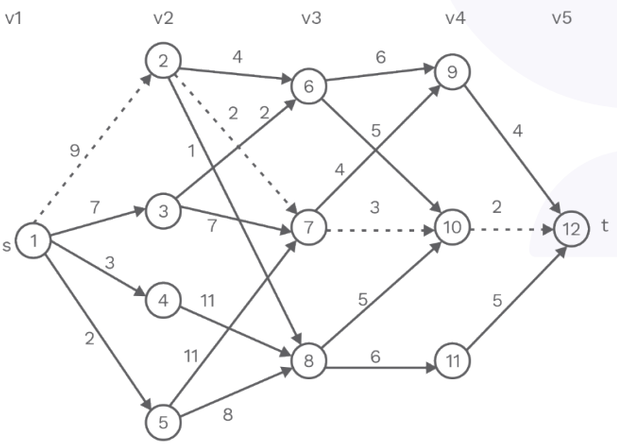
Forward method:

```
Algorithm FGraph (G,k,n,p)
// The input is a k-stage graph G=(V,E) with 'n' vertex.
// Indexed in order of stages and E is a set of edges.
// and c[i,j] is the cost of edge <i,j> is a minimum-cost path.
{
    cost[n]=0.0;
    for j=n-1 to 1 do
    {
        // compute cost[j],
        // let 'r' be the vertex such that <j,r> is an edge of 'G' &
        // c[j,r]+cost[r] is minimum.
        cost[j] = c[j,r] + cost[r];
        d[j] = r;
    }
// find a minimum cost path.
    P[1] = 1;
    P[k]=n;
    For j=2 to k-1 do
    P[j]=d[P[j-1]];
}
```

- Assume that there are 'k' stages in a graph.
- In this FORWARD approach, we find out the cost of each and every node starting from the 'k' th stage to the 1st stage.
- We will find out the path (i.e.) minimum cost path from source to the destination (i.e,) [Stage-1 to Stage-k].

- Maintain a cost matrix cost[n] which stores the distance from any vertex to the destination.
- If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path and will be stored in the distance array 'd'.
- Thus, we can find out the minimum cost path from each and every vertex.
- Finally cost(1) will give the shortest distance from source to destination.
- For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the destination.
- For a 'k' stage graph, there will be 'k' vertex in the path.

Example:



- In the above graph V1...V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

STEPS: DESTINATION, D

- Cost (12) = 0 D (12) = 0
- Cost (11) = 5 D (11) = 12
- Cost (10) = 2 D (10) = 12
- Cost (9) = 4 D (9) = 12

For forward approach,

Cost (i,j) = min {c (j,l) + Cost (i+1,l) }
$l \in V_i + 1$
$(j,l) \in E$

cost(8) = min {c (8,10) + cost (10), c(8,11) + Cost(11)}
= min (5+2, 6+5)
= 7
cost(8) = 7 =>D(8) = 10
cost(7) = min(c (7,9) + cost(9), c(7,10) + cost(10))
= min(4+4, 3+2)
= min(8,5)
= 5
cost(7) = 5 => D(7) = 10
cost(6) = min(c (6,9) + cost(9), c(6,10) + cost(10))
= min(6+4, 5+2)
= min(10, 7)
= 7
cost(6) = 7 =>D(6) = 10
cost(5) = min(c (5,7) + cost(7), c(5,8) + cost(8))
= min(11+5, 8+7)
= min(16,15)
= 15
cost(5) = 15 =>D(5) = 8
cost(4) = min(c (4,8) + cost(8))
= min(11+7)
= 18
cost(4) = 18 =>D(4) = 8
cost(3) = min(c (3,6) + cost(6), c(3,7) +cost(7))
= min(2+7, 7+5)
= min(9, 12)
= 9
cost(3) = 9 =>D(3) = 6
cost(2) = min(c (2, 6) + cost(6), c(2,7) + cost(7), c(2,8) + cost(8))
= min(4+7, 2+5, 1+7)
= min(11, 7, 8)
= 7
cost(2) = 7=>D(2) = 7
cost(1)= min(c (1,2)+cost(2), c(1,3) + cost(3), c(1,4) + cost(4), c(1,5) + cost(5))
= min(9+7, 7+9, 3+18, 2+15)
= min(16, 16, 21, 17)
= 16
cost(1)= 16 =>D(1) = 2
Start from vertex -2
D (1) = 2
D (2) = 7

D (7) = 10
D (10) = 12

So, the minimum-cost path is,

Backward method:

- It is similar to forward approach, but differs only in two or three ways.
- Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
- Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
- To find out the path start from vertex 'k', then the distance array D (k) will give the minimum cost neighbour vertex which in turn gives the next nearest neighbour vertex and proceed till we reach the destination.

Algorithm: backward method:

```
Algorithm BGraph (G,k,n,p)

// The input is a k-stage graph G=(V,E) with 'n' vertex.

// Indexed in order of stages and E is a set of edges.

// and c[i,j] is the cost of edge <i,j> (i,j are the vertex number), p[k] is a minimum cost path.

{
    bcost[1]=0.0;
    for j=2 to n do
    {
            // compute bcost[j]
            // let 'r' be the vertex such that <r,j> is an edge of 'G' &
            // bcost[r]+c[r,j] is minimum.
    bcost[j] = bcost[r] + c[r,j];
            d[j]=r;
    }
// find a minimum cost path.
    P[1]=1;
    P[k]=n;
    For j= k-1 to 2 do
    P[j]=d[P[j+1]];
}
```

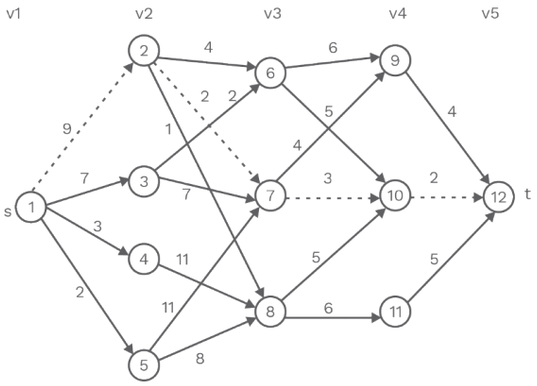


Fig. 5.1

Cost(1) = 0 => D(1)= 0
Cost(2) = 9 => D(2)=1
Cost(3) = 7 => D(3)=1
Cost(4) = 3 => D(4)=1
Cost(5) = 2 => D(5)=1
Cost(6) = min(c (2,6) + cost(2), c(3,6) + cost(3))
= min(13,9)
cost(6) = 9 => D(6)=3
Cost(7) = min(c (3,7) + cost(3), c(5,7) + cost(5), c(2,7) + cost(2))
= min(14,13,11)
cost(7) = 11 =>D(7)=2
Cost(8) = min(c (2,8) + cost(2), c(4,8) + cost(4), c(5,8) + cost(5))
= min(10,14,10)
cost(8) = 10 =>D(8)=2
Cost(9) = min(c (6,9) + cost(6), c(7,9) + cost(7))
= min(15, 15)
cost(9) = 15 =>D(9)=6
Cost(10) = min(c(6,10) + cost(6), c(7,10) + cost(7)), c(8,10) + cost(8))
= min(14,14,15)
cost(10) = 14 =>D(10)=6
Cost(11) = min(c (8,11) + cost(8))
cost(11) = 16 =>D(11)=8
Cost(12) = min(c(9,12) + cost(9), c(10,12) + cost(10), c(11,12) + cost(11))
= min(19,16,21)
cost(12) = 16 =>D(12) = 10

Start from vertex-12:

D(12) = 10
D(10) = 6
D(6) = 3
D(3) = 1

So the minimum cost path is,

$$1 \xrightarrow{7} 3 \xrightarrow{2} 6 \xrightarrow{5} 10 \xrightarrow{2} 12$$

The cost is 16.

Travelling Salesman Problem

- The traveling salesman problem (TSP) is to find the shortest possible route that visits each city exactly once and returns to the starting point given a set of cities and the distance between each pair of cities.
- Take note of the distinction between the Hamiltonian cycle and the TSP. The Hamiltonian cycle problem entails determining whether a tour exists that visits each city exactly once. The problem is to find a minimum weight Hamiltonian cycle. We know that Hamiltonian tours exist (because the graph is complete), and there are many of them.
- Let the number of vertices in the given set be 1, 2, 3, 4,...n. Let's use 1 as the starting and ending points for the output. We find the minimum cost path with 1 as the starting point, i as the ending point, and all vertices appearing exactly once.
- Let's say the cost of this path is cost(i), then the cost of the corresponding cycle is cost(i) + dist(i, 1), where dist(i, 1) is the distance between from i to 1. Finally, we return the value that is the smallest of all [cost(i) + dist(i, 1)]. So far, this appears to be straightforward. The question now is how to obtain cost(i).
- We need a recursive relation in terms of sub-problems to calculate the cost(i) using dynamic programming.
- Let's say C(S, i) is the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.
- We begin with all subsetds of size 2 and calculate C(S, i) for all subsets where S is the subset, then calculate C(S, i) for all subsets of size 3, and so on. It's worth noting that 1 must appear in each subset. For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S| > 1, we define C(S,1) = ∞ since the path cannot both start and end at 1.

Now, let's express C(S,j) in terms of smaller subproblems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, C(S – {j}, i), plus the length of the final edge. d_{ij}. We must pick the best such i:

$$C(S, j) = \min_{i \in S: i \neq j} C(S - \{j\}, i) + d_{ij}.$$

The subproblems are ordered by |S|. Here's the code.

C({1},1) = 0

for s = 2 to n:

for all subsets $S \subseteq \{1, 2, \ldots, n\}$ of size s and containing 1:

C(S,1) = ∞

for all $j \in S, j \neq 1$:

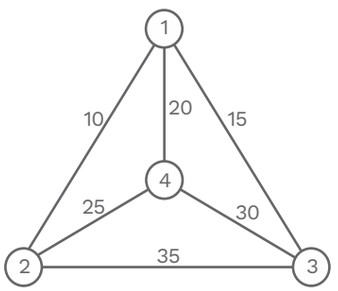
$$C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\}$$

return $\min_j C(\{1, \ldots, n\}, j) + d_{j1}$

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is, therefore $O(n^2 2^n)$.

Example:

- Consider the graph



- Matrix representation of the above graph

	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0

- Lets start from node 1
  C(4, 1) = 20 C(3,1) = 15 C(2,1) = 10
  C({3},2) = d(2,3) + C(3,1) = 50
  C({4},2) = d(2,4) + C(4,1) = 45
  C({2},3) = d(3,2) + C(2,1) = 45
  C({4},3) = d(3,4) + C(4,1) = 50
  C({2},4) = d(4,2) + C(2,1) = 35
  C({3},4) = d(4,3) + C(3,1) = 45
  C({3,4,2}) = min(d(2,3) + C({4},3), d(2,4) + C({3},4))
  = min(85,70)
  = 70
  C({2,4},3) = min(d(3,2) + C({4},2), d(3,4) + C({2},4))
  = min(80,65)
  = 65
  C({2,3},4) = min(d(4,2) + C({3},2), d(4,3) + C({2},3))
  = min(75, 75)
  = 75
- Finally
  C({2,3,4},1) = min(d(1,2) + C({3,4},2), d(1,3) + C({2,4},3), d(1,4)+C({2,3},4))
  = min(80,80,95)
  = 80
  ∴ Optimal tour length is 80
  Optimal tour 1-2-4-3-1

0-1 Knapsack problem:

The 0-1 knapsack problem is as follows. A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because, for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

Dynamic Programming Approach

- Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on n – 1 items with maximum weight W – w.
- Thus, to take a bottom-up approach, we must solve the 0-1 knapsack problem for all items and possible weights smaller than W.
- We'll build an n + 1 by W + 1 table of values where the rows are indexed by item, and the columns are indexed by total weight.
- For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of a knapsack including items 1 through i – 1 with max weight j, or the total value of including items 1 through i – 1 with max weight j – i.weight and also item i.
- To solve the problem, we simply examine the [n,W] entry of the table to determine the maximum value we can achieve.
- To read the items we include, start with entry [n,W]. In general, proceed as follows: If entry [i,j] equals entry [i–1, j], don't include item i, and examine entry [i–1, j], next.
- If entry [i,j] doesn't equal entry [i – 1, j], include item i and examine entry [i – 1, j – i].weight next.

Algorithm 0-1 Knapsack(n,W)

1. Initialize a 2D matrix KS of size (n+1) x (W+1)
2. Initialize 1st row and 1st column with zero
3. For itr<- 1 to n
4. For j<-1 to W
5. If(j< W[itr])
6. KS[itr,j]<- K[itr-1][j]
7. Else
8. KS[itr,j]<-max(KS[itr-1,j] , KS[itr-1 , j])
9. End for
10. End for

End

Analysis:

- Since the time to calculate each entry in the table KS[i,j] is constant; the time complexity is Θ(n x W). Where n is the number of items and W is the capacity of the Knapsack.

Recurrence relation:

- In a Knapsack problem, we are given a set of n items where each item i is specified by a size/weight w_i and a value P_i. We are also given the size bound W, the size (capacity) of our Knapsack.

$$KS(i,w) = \begin{cases} \max(P_i + KS(i-1, w-w_i), KS(i-1,w)); \text{if } w_i \le w \\ KS(i-1,w); \text{if } w_i > w \\ 0 \quad ; \text{if } i = 0 \text{ or } w = 0 \end{cases}$$

Where KS (i,w) is the best value that can be achieved, for instance, with only the first i items and capacity w.

Example:

Item	1	2	3
Weight (in kgs)	1	2	3
Values (In rupees)	10	15	40

Capacity of bag = W = 6 kgs

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

$$KS(1,1) = \max\begin{cases} P_1 + KS(0,0), \\ KS(0,1) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,2) = \max\begin{cases} P_1 + KS(0,1), \\ KS(0,2) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,3) = \max\begin{cases} P_1 + KS(0,2), \\ KS(0,3) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,4) = \max\begin{cases} P_1 + KS(0,3), \\ KS(0,4) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,5) = \max\begin{cases} P_1 + KS(0,4), \\ KS(0,5) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,6) = \max\begin{cases} P_1 + KS(0,5), \\ KS(0,6) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(2,1) = KS(1,1) = 10$$

$$KS(2,2) = \max\begin{cases} P_2 + KS(1,0), \\ KS(1,2) \end{cases} = \max\begin{cases} (15+0), \\ 10 \end{cases} = 15$$

$$KS(2,3) = \max\begin{cases} P_2 + KS(1,1), \\ KS(1,3) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,4) = \max\begin{cases} P_2 + KS(1,2), \\ KS(1,4) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,5) = \max\begin{cases} P_2 + KS(1,3), \\ KS(1,5) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,6) = \max\begin{cases} P_2 + KS(1,4), \\ KS(1,6) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(3,1) = KS(2,1) = 10$$

$$KS(3,2) = KS(2,2) = 15$$

$$KS(3,3) = \max\begin{cases} P_3 + KS(2,0), \\ KS(2,3) \end{cases} = \max\begin{cases} (40+0), \\ 25 \end{cases} = 45$$

$$KS(3,4) = \max\begin{cases} P_3 + KS(2,1), \\ KS(2,4) \end{cases} = \max\begin{cases} (40+10), \\ 25 \end{cases} = 50$$

$$KS(3,5) = \max\begin{cases} P_3 + KS(2,2), \\ KS(2,5) \end{cases} = \max\begin{cases} (40+15), \\ 25 \end{cases} = 55$$

$$KS(3,6) = \max\begin{cases} P_3 + KS(2,3), \\ KS(2,6) \end{cases} = \max\begin{cases} (40+25), \\ 25 \end{cases} = 65$$

Subset Sum Problem

Problem:

Given a sequence of n positive numbers A_1, A_2.....A_n, give an algorithm which checks whether there exists a subset of A whose sum of all numbers is T.

- This is a variation of the Knapsack problem. As an example, consider the following array: A = [3, 2, 4, 19, 3, 7, 13, 10, 6, 11] Suppose if we want to check whether there is any subset whose sum is 17. The answer is yes, because the sum of 4 + 13 = 17 and therefore {4, 13} is such a subset.
- Greedy method is not applicable in subset-sum. If we try to be greedy, then we don't know whether to take the smallest number or the highest number. So, there is no way we could go for greedy.

Example:

Let set = {6, 2, 3, 1}. Check whether there is a subset whose sum is 5.

Solution:

Here, there is a subset {3, 2} whose sum is 5 Here greedy method fails; if we try to be greedy on a smaller number, then it will take 1, and so if we take 1, then the remaining sum is going to be 4. So, there is no number or subset which makes to 4.

So, greedy doesn't work in subset-sum.

Brute-fore method:

- If there are 'n' numbers, then any number can be present or absent in subset. So, every a number has 2 options. Hence, the number of subsets is equal to 2^n.
- Brute force method examines every subset, i.e, equal to 2^n. To examine each sub-problem, it will take O(n) time.
- Therefore, time complexity = number of subproblem * time taken by each sub-problem. = $0(2^n) * O(n) = O(n2^n)$

Recursive equation:

- Let us assume that SS(i, S) denote a subset-sum from a_1 to a_i whose sum is equal to some number 'S'.
- First, we check the base condition. If there are no elements i.e., i = 0, and we want to produce some sum S then it is not possible. So, it is false.
- If there are no elements i.e., i = 0, and sum S = 0 is possible. So, it is true because Sum = 0 is always possible.
- This above two are base conditions.
- Using i^{th} element if this sum 'S' has to be possible, then there are two cases:

  Case 1: If we include the i^{th} element in our subset, then a sum of $(S - a_i)$ should be possible with the remaining (i-1) elements.

  Case 2: If we don't include the i^{th} element in our subset, then a sum of (S) should be possible with the remaining (i-1) elements. So, the recursive equation look like as shown below:

$$SS(i,s) = \begin{cases} SS(i-1,S); S < a_i \\ SS(i-1,S-a_i) \ V \ S \ S(i-1,S); S \geq a_i \\ true; S = 0 \\ False; i = 0, S \neq 0 \end{cases}$$

- If the problem is a subset-sum of SS(n,w) (where n positive numbers $A_1A_2...A_n$ and w is a sum). Then the number of unique sub-problem is O(nw).

Solved Examples

1. Given the set S = {6, 3, 2, 1}. Is there any subset possible whose sum is equal to 5.

Solution:

Since here, the number of elements = 4 and the sum is going to be 5 then the number of problem = 4 × 5 = 20.

Sum →

number of element ↓	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T

- Whenever we want the sum = 0 then it is always possible with whatever element because a null set is going to be sum = 0.
- Index (i, j) indicate, with ith element is sum j possible.
  SS(1,1) = SS(0,1) = False
  [Since the 1st element weight is '6'. So, it can't lead to sum of 1 then we have to go for SS(i-1,S) i.e., SS(0, 1)]
  Similarly,
  SS(2,1) = SS(1,1) = False
  SS(2,2) = SS(1,2) = False
  SS(2,3) = SS(1,0) VSS(1,3) = True V False = True
  SS(2,4) = SS(1,1) V SS(1,4) = False V False = False
  SS(2,5) = SS(1,2) V SS(1,5) = False V False = False
  SS(3,1) = SS(2,1) (Here, S < i) = False
  SS(3,2) = SS(2,0) V SS(2,2) = True V False = True
  SS(3,3) = SS(2,1) V SS(2,3) = False V True = True
  SS(3,4) = SS(2,2) V SS (2,4) = False V False = False
  SS(3,5) = SS(2,3) V SS (2,5) = True V False = True
  SS(4,1) = SS(3, 0) V SS(3,1) = True V False = True
  SS(4,2) = SS(3,1) V SS(3,2) = False V True = True
  SS(4,3) = SS(3,2) V SS (3,3) = True V True = True
  SS(4,4) = SS(3,3) V SS(3,4) = True V False = True
  SS(4,5) = SS(3,4) V SS(3,5) = False V True = True
- Since, the final answer is in shell (4,5). So, the final answer is true. The final answer will always be present in (n, w).
- The subset sum can be computed either in row major order or column major order.

Time complexity:

- Here, the number of the subproblem is (nw), and the time required to calculate each subproblem is O(1). Hence, time complexity = (nw) × O(1) = O(nw).

Note:

- Either to use dynamic programming or not depends on the value of w.
- If 'w' a is big number, then the brute force method gives better time complexity i.e. $O(2^n)$ otherwise dynamic programming.

Conclusion:

Time complexity = $\min \begin{cases} O(2^n) \\ O(nw) \end{cases}$

- If w is n! then 2^n is going to be better than O(nw).

Space complexity:

Space complexity is required for the table. So, O(nw) is the space complexity.

All Pairs Shortest Path Floyd Warshall

Problem:

Given a weighted directed graph G = (V, E), where V = {1, 2, ...,n}. Find the shortest path between all pair of nodes in the graph.

- We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithms |V| times, once for each vertex as the source.
- If all the edges of the graph contain the positive weight, then apply Dijkstra's algorithm.
- If we use the linear-array implementation of the min-priority queue, the running time is $O(V^3 + VE)$, and we know, $E = V^2$ so, $O(V^3)$.
- The binary min-heap implementation of the min-priority queue yields a running time of O(V+ElogV), which is an improvement if the graph is sparse.
- Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \log V + VE)$.
- Instead, we must run the slower Bellman-ford algorithm once from each vertex.
- The resulting running time is $O(V^2E)$, which for the dense graph is $O(V^4)$
- In the Floyd-Warshall algorithm, the negative-weight edge is allowed.

- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $P = \langle V_1, V_2, \ldots, V_l \rangle$ is any vertex of P other than V_1 or V_l, that is, any vertex in the set $\{V_2, V_3, \ldots, V_{l-1}\}$
- The Floyd-Warshall algorithm relies on the following observation.
- Under our assumption that the vertices of G are V = {1,2,...,n}, let us consider a subset {1,2,...,k} of vertices for some k.
- For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from {1,2,...k} and Let p be a minimum-weight path from among them. (Path P is simple)
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1,2,...,k-1}.
- The relationship depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1,2,...,k-1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,...,k-1} is also the shortest path from i to j with all intermediate vertices in the set {1,2,...,k}.
- If k is an intermediate vertex of path p, then we decompose p into $i \xrightarrow{P_1} k \xrightarrow{P_2} j$, as shown below:

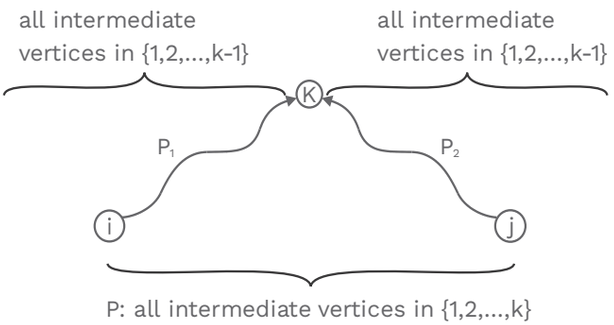


Fig. 5.2

- P_1 is a shortest path from vertex i to vertex k with all intermediate vertices in the set {1,2,....,k}. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set {1,2,...,k-1}. Therefore, P_1 is a shortest path from i to k with all intermediate vertices in the set {1,2,...,k-1}.
- Similarly, P_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1,2,...,k-1}.

Recurrence relation:

- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2,...,k}.
- When k=0, a path from vertex i to vertex j with no intermediate vertex.
- Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$.

  We define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

- Because for any path, all intermediate vertices are in the set {1,2,...,n}, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for $i, j \in V$.
- Based on recurrence relation, we can us the following bottom-up procedure to compute the value $d_{ij}^{(k)}$ in order of increasing values of k.

Floyd-Warshall(W)

1. $n = W \cdot rows$
2. $D^{(0)} = W$
3. For k = 1 to n
4. Let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new n × n size matrix
5. For i = 1 to n
6. For j=1 to n
7. $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8. Return $D^{(n)}$

- Its input is an n × n matrix W. The procedure returns the matrix $D^{(n)}$ of shortest-path weights.
- The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-7 because each execution of line 7 takes O(1) time, the algorithm runs in time $\theta(n^3)$.

eg:

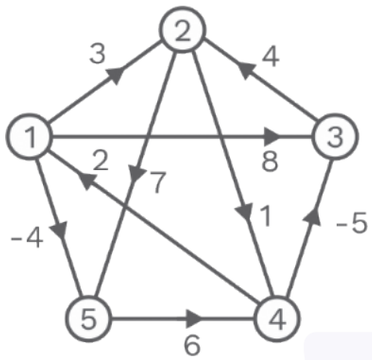


Fig. 5.3

$$D^{(0)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&\infty&-5&0&\infty\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(1)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&5&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(2)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&4&-4\\\infty&0&\infty&1&7\\\infty&4&0&5&11\\2&5&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(3)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&4&-4\\\infty&0&\infty&1&7\\\infty&4&0&5&11\\2&-1&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(4)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&-1&4&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{bmatrix}\end{array}$$

$$D^{(5)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&1&-3&2&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{bmatrix}\end{array}$$

Space complexity:

Here, space complexity is $O(n^3)$, but we can reduce the space complexity to $O(n^2)$.

Previous Years' Question

The Floyd-Warshall algorithm for all-pair shortest paths computation is based on:
(A) Greedy paradigm.
(B) Divide-and-Conquer paradigm
(C) Dynamic Programming paradigm
(D) Neither Greedy nor Divide-and Conquer nor Dynamic programming paradigm
Solution: (C) **[GATE 2016 (Set-2)]**

Optimal Binary Search Trees

Problem:

- Given a set of n(sorted) keys A[1...n], build the best binary search tree for the elements of A. Also, assume that, each element is associated with the frequency, which indicates the number of times that particular item is searched in the binary search tree.
- To reduce the total search time, we need to construct BST (Binary search tree).
- Understand the problem by taking 5 elements in the array. Let us assume that the given array is A=[3, 12, 21, 32, 35]. To represent these elements, there are many ways and below are two of them.

- The search time for an element depends which level node is present.
- The average number of comparisons for the first tree is: $\frac{1+2+2+3+3}{5} = \frac{11}{5}$ and for the second tree, the average number of comparisons: $\frac{1+2+3+3+4}{5} = \frac{13}{5}$.

  Among the two, the first tree is giving better results.
- Here, frequencies are not given, and if we want to search all elements, then the above simple calculation is enough for deciding the best tree.
- If the frequencies are given, then the selection depends on the frequencies of the elements and also the depth of the elements.
- For simplicity, let us assume that, the given array is A and the corresponding frequencies are in array F. F[i] indicates the frequency of i^{th} element A[i].
- With this, the total search time S(root) of the tree with root can be defined as

$$S(root) = \sum_{i=1}^{n} (depth(root, i) + 1) \times F[i])$$

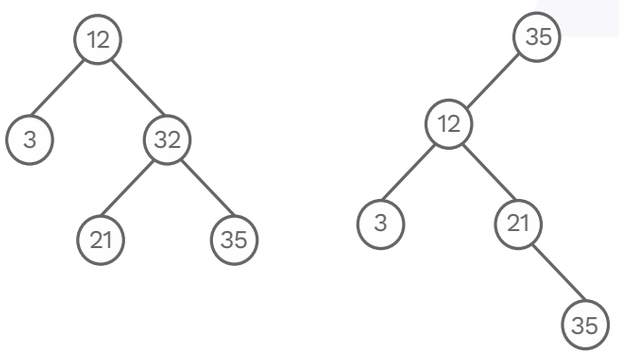


Fig. 5.4

- In the above expression, depth (root, i) + 1 indicates the number of comparisons for searching the i^{th} element.
- Since we are trying to create a binary search tree, the left subtree elements are less than the root element, and the right subtree elements are greater than the root element.
- If we separate the left subtree time and right subtree time, then the above expressions can be written as:

$$S(root) = \sum_{i=1}^{r-1} (depth(root, i) + 1) \times F[i]) + \sum_{r}^{n} F[i] + \sum_{i=r+1}^{n} (depth(root, i) + 1) \times F[i])$$

  Where "r" indicates the positions of the root element in the array.
- If we replace the left subtree and right subtree times with their corresponding recursive calls, then the expression becomes:

$$S(root) = S(root \rightarrow left) + S(root \rightarrow right) + \sum_{i=1}^{n} F[i]$$

Implementation:

```
Node optimalBST(int keys[], int freq[])
{
      int n = keys.length;
      int cost[][] = new int[n][n];
      int root[][] = new int[n][n];
      for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)
      cost[i][j] = inf;
      /*
      cost[i][j] is the minimum cost of optimal
      subtree formed by the vertices[i to j]
      root[i][j] represents the index of the root
      in the subtree with vertices [i to j]
      */
      // cost of the optimal binary search tree
      // index starts at 0
intminCost=optimalCost(0,n-1,freq,cost,root);
NodeoptimalTreeRoot=buildOptimalTree(0,n-
1,keys,root);
      return optimalTreeRoot;
}
int optimalCost(int i, int j , int freq[], int cost[]
[], int root[][]){
            // base conditions
            if(i<j) return 0;
            else if(i==j) return freq[i];
            // using stored values
            if(cost[i][j] < inf) return cost[i][j];
            int minCost = inf;
            int minRoot = i;
```

```
        for(int r=i; r<=j; r++){
            // root can be any vertex from i to j
            int c = optimalCost(i,r-1,freq,cost,root) + optimalCost(r+1,j,freq,cost,root);
            if(c<minCost){
                minCost=c;
                minRoot = r;
            }
        }
        int freqSum = 0;
        for(int k=i;k<=j;k++)
            freqSum+=freq[k];
            cost[i][j] = minCost + freqSum;
            root[i][j] = minRoot;
            return cost[i][j];
}
Node buildOptimalTree(int i,int j,int keys[], int root[][]){
        // base conditions
        if(i<j) return null;
        if(i==j) return new Node(keys[i]);
        // getting the index of optimal root of subtree[i,j] stored in the matrix
        int rindex = root[i][j];
        Node node = new Node(keys[rindex]);
        node.left = buildOptimalTree(i,rindex-1,keys,root);
        node.right = buildOptimalTree(rindex+1,j,keys,root);
        return node;
}
```

Conclusion:

- We can determine whether the given problem can be solved using a dynamic approach based on the two properties:-
  - Optimal substructure: An optimal solution to a problem contains the optimal solution to subproblem.
  - Overlapping subproblems: A recursive solution the contains a small number of similar subproblems repeated many times.
- Bottom-up programming depend on values to calculate and order of evaluation. In this approach, we solve the sub-problems first only and based on the solution of the sub-problems we determine the solution to the larger problem.
- In top-down programming, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into subproblems, and these subproblems are solved, and the solutions are remembered, in case they need to be solved again.

Note:
Some problems can be solved with both techniques.

Solved Examples

1. Let A_1, A_2, A_3 and A_4 be four matrices of dimensions 1 × 2, 2 × 1, 1 × 4 and 4 × 1 respectively. The minimum number of scalar multiplications required to find the product $A_1A_2A_3A_4$ using the basic matrix multiplication method is _____

Solution: 7

The unique function call which are made in m[1, 4] are given below:

0 (1, 1)	0 (2, 2)	0 (3, 3)	0 (4, 4)
2 (1, 2)	8 (2, 3)	4 (3, 4)	
6 (1, 3)	6 (2, 4)		
7 (1, 4)			

$$M[i, j] = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \le k < j} \{M[i,k] + M[k+1, j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Number of scalar multiplication

$A_1A_2 = 1 \times 2 \times 1 = 2$

$A_2A_3 = 2 \times 1 \times 4 = 8$

$A_3A_4 = 1 \times 4 \times 1 = 4$

$$A_1A_2A_3 = m(1,3) = \min \begin{cases} m(1,1) + m(2,3) + 1 \times 2 \times 4 \\ m(1,2) + m(3,3) + 1 \times 1 \times 4 \end{cases}$$

$$= \min \begin{cases} 0 + 8 + 8 \\ 2 + 0 + 4 \end{cases} = 6$$

$$A_2A_3A_4 = m(2,4) = \min \begin{cases} m(2,2) + m(3,4) + 2 \times 1 \times 1 \\ m(2,3) + m(4,4) + 2 \times 4 \times 1 \end{cases}$$

$$= \min \begin{cases} 0 + 4 + 2 \\ 8 + 0 + 8 \end{cases} = 6$$

$$A_1A_2A_3A_4(1,4) = \min \begin{cases} m(1,1) + m(2,4) + 1 \times 2 \times 1 \\ m(1,2) + m(3,4) + 1 \times 1 \times 1 \\ m(1,3) + m(4,4) + 1 \times 4 \times 1 \end{cases}$$

$$= \min \begin{cases} 0 + 6 + 2 \\ 2 + 4 + 1 \\ 6 + 0 + 4 \end{cases} = \min \begin{cases} 8 \\ 7 \\ 10 \end{cases} = 7$$

Hence, 7 is the minimum number of scalar multiplication required to find the product $A_1A_2A_3A_4$.

2. Let us define A_iA_{i+1} as an explicitly computed pair for a given parenthesization if they are directly multiplied. For example, in the matrix multiplication chain $A_1A_2A_3A_4$ using parenthesization $\mathbf{A_1((A_2A_3)A_4)}$, A_2 A_3 are only explicitly computed pairs.

Consider the matrix given in question number 1 that minimises the total number of scalar multiplication, the explicitly computed pairs is/are

(A) A_1A_2 and A_3A_4 only

(B) A_2A_3 only

(C) A_1A_2 only

(D) A_3A_4 only

Solution: (A)

In question 1, we got the optimal scalar as 7 in (A_1A_2), (A_3A_4). So, A_1A_2, and A_3A_4 are explicitly computed pairs.

3. Consider two strings X = "ABCBDAB" and Y = "BDCABA". Let u be the length of the longest common subsequences (not necessarily contiguous) between X and Y and let V be the number of such longest common subsequences between X and Y. then V + 10u is_____

Solution: 43

		0	1	2	3	4	5	6
		Y	B	D	C	A	B	A
0	X	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Here, the subsequence are shown below:

X(2 3 4 6) X(4 5 6 7) X(2 3 6 7)

Y(1 3 5 6) Y(1 2 4 5) Y(1 3 4 5)

B C B A B D A B B C A B

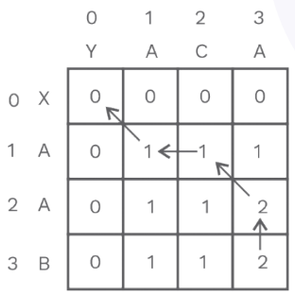
The length of the longest common subsequence is 4 and there are 3 subsequence of length 4.

So, u = 4 and v = 3

So, v + 10u = 3 + 40 = 43

4. Consider two string X = "AAB" and Y = "ACA". Let u be the length of the longest common subsequence (not necessarily contiguous) between X and Y and let v be the number of such longest common subsequences between X and Y.
Then u + 10v is_____

Solution: 12

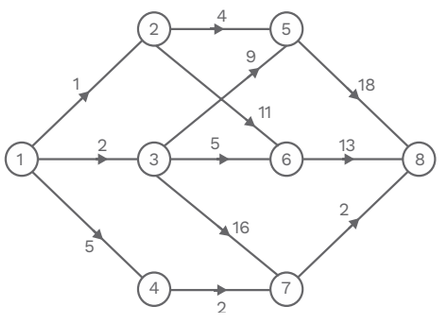


Here, the subsequence is AA. So, the length of subsequence is 2, and there is only 1 occurrence of subsequence.

Hence, u = 2, v = 1

u + 10v = 2 + 10*1 = 12.

5. Consider the multistage graph K = 4 then find out the minimum cost path from node 1 to node 8?



Solution: 9

The path is from $1 \rightarrow 4 \rightarrow 7 \rightarrow 8$, which incur a minimum cost as 9.

T[i] denotes the minimum cost from ith node to node 8.

$$\text{So, } T[i] = \underset{(\text{for all } j=(i+1) \text{ to } n)}{\text{minimum}} \{\text{cost}(i, j) + T[j]\}$$

$$T[8] = 0 \quad j = (i+1) \text{ to } n$$

$$T[7] = \text{minimum}\{\text{cost}[7,8] + T[8]\}$$

$$= \text{minimum}\{2+0\} = 2$$

$$T[6] = \text{minimum}\begin{cases}\text{cost}[6,7] + T[7] \\ \text{cost}[6,8] + T[8]\end{cases}$$

$$T[6] = \text{minimum}\begin{cases}\infty + 2 \\ \qquad = 13 \\ 13 + 0\end{cases}$$

$$T[5] = \text{minimum}\begin{cases}\text{cost}[5,6] + T[6] \\ \text{cost}[5,7] + T[7] \\ \text{cost}[5,8] + T[8]\end{cases}$$

$$T[5] = \text{minimum}\begin{cases}\infty + 13 \\ \infty + 2 \quad = 18 \\ 18 + 0\end{cases}$$

$$T[4] = \text{minimum}\begin{cases}\text{cost}[4,5] + T[5] \\ \text{cost}[4,6] + T[6] \\ \text{cost}[4,7] + T[7] \\ \text{cost}[4,8] + T[8]\end{cases}$$

$$T[4] = \text{minimum}\begin{cases}\infty + 18 \\ \infty + 13 \quad = 4 \\ 2 + 2 \\ \infty + 0\end{cases}$$

$$T[3] = \text{minimum}\begin{cases}\text{cost}[3,4] + T[4] \\ \text{cost}[3,5] + T[5] \\ \text{cost}[3,6] + T[6] \\ \text{cost}[3,7] + T[7] \\ \text{cost}[3,8] + T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}\infty + 4 \\ 9 + 18 \\ 5 + 13 = 18 \\ 16 + 2 \\ \infty + 0\end{cases}$$

$$T[2] = \text{minimum}\begin{cases}\text{cost}[2,3]+T[3]\\ \text{cost}[2,4]+T[4]\\ \text{cost}[2,5]+T[5]\\ \text{cost}[2,6]+T[6]\\ \text{cost}[2,7]+T[7]\\ \text{cost}[2,8]+T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}\infty+18\\ \infty+4\\ 4+18\\ 11+13\\ \infty+2\\ \infty+0\end{cases} = 22$$

$$T[1] = \text{minimum}\begin{cases}\text{cost}[1,2]+T[2]\\ \text{cost}[1,3]+T[3]\\ \text{cost}[1,4]+T[4]\\ \text{cost}[1,5]+T[5]\\ \text{cost}[1,6]+T[6]\\ \text{cost}[1,7]+T[7]\\ \text{cost}[1,8]+T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}1+22\\ 2+18\\ 5+4\\ \infty+18\\ \infty+13\\ \infty+2\\ \infty+0\end{cases} = 9$$

Hence, 9 is the minimum cost path from node 1 to node 8.

6. Shortest path in the multistage graph can be found by using

(A) Greedy method

(B) Dynamic method

(C) Either by greedy method or dynamic method

(D) None of above

Solution: (B)

Greedy method fails in finding the shortest path in the multistage graph.

By using dynamic programming, we can get the shortest path in the multistage graph.

7. Consider the weights and values of the items listed below.

Item Number	Weight (in kgs)	Values (in Rupees)
1	1	10
2	2	12
3	4	28

The task is to pick a subset of these items such that their total weight is no more than 6kg and their total value is maximised. Moreover, no item may be split. The total values of items picked by 0/1 knapacks is_____

Solution: 40

object \ weight	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

If there is no object, then the maximum profit (value) is going to be 0. This is indicated by 1st row.

Similarly, if the weight is 0, then also maximum profit is 0. This is indicated by 1st column.

Let ks(i,w) indicate maximum profit if considering i's number of elements and maximum weight occupied is w.

By using recurrence equation:

$$ks(i,w) = \begin{cases}\max(p_i + ks(i-1, w-w_i), ks(i-1,w)), w_i \le w\\ 0; i = 0 \text{ or } w = 0\\ ks(i-1,w); w_i > w\end{cases}$$

$$ks(1,1) = \max\begin{cases}p_1 + ks(0,0)\\ ks(0,1)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,2) = \max\begin{cases}p_1 + ks(0,1)\\ ks(0,2)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,3) = \max\begin{cases}p_1 + ks(0,2)\\ ks(0,3)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,4) = \max\begin{cases} p_1 + ks(0,3) \\ ks(0,4) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(1,5) = \max\begin{cases} p_1 + ks(0,4) \\ ks(0,5) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(1,6) = \max\begin{cases} p_1 + ks(0,5) \\ ks(0,6) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(2,1) = ks(1,1) = 10$$

$$ks(2,2) = \max\begin{cases} p_2 + ks(1,0) \\ ks(1,2) \end{cases} = \max\begin{cases} 12+0 \\ 10 \end{cases} = 12$$

$$ks(2,3) = \max\begin{cases} p_2 + ks(1,1) \\ ks(1,3) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,4) = \max\begin{cases} p_2 + ks(1,2) \\ ks(1,4) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,5) = \max\begin{cases} p_2 + ks(1,3) \\ ks(1,5) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,6) = \max\begin{cases} p_2 + ks(1,4) \\ ks(1,6) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(3,1) = ks(2,1) = 10$$

$$ks(3,2) = ks(2,2) = 12$$

$$ks(3,3) = ks(2,3) = 22$$

$$ks(3,4) = \max\begin{cases} p_3 + ks(2,0) \\ ks(2,4) \end{cases} = \max\begin{cases} 28+0 \\ 22 \end{cases} = 28$$

$$ks(3,5) = \max\begin{cases} p_3 + ks(2,1) \\ ks(2,5) \end{cases} = \max\begin{cases} 28+10 \\ 22 \end{cases} = 38$$

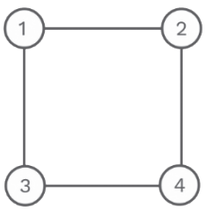
$$ks(3,6) = \max\begin{cases} p_3 + ks(2,2) \\ ks(2,6) \end{cases} = \max\begin{cases} 28+12 \\ 22 \end{cases} = 40$$

8. Subset-sum can be computed by
(A) Row-major order only
(B) Column-major order only
(C) Either by row-major for column-major only
(D) None of the above

Solution: (C)

It can be computed by eighter row-major or column-major only.

9. What is the minimum cost of the travelling salesman problem, if starting vertex is 1?



Cost matrix:–

	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Solution: 5

$$T(1,\{2,3,4\}) = \min\begin{cases} (1,2) + T(2,\{3,4\}) \\ (1,3) + T(3,\{2,4\}) \\ (1,4) + T(4,\{2,3\}) \end{cases}$$

$$= \min\begin{cases} 1+4 \\ 2+7 \\ 3+4 \end{cases}$$

$$= 5$$

$$T(2,\{3,4\}) = \min\begin{cases} (2,3) + T(3,\{4\}) \\ (2,4) + T(4,\{3\}) \end{cases} = \min\begin{cases} 4+8 \\ 2+2 \end{cases} = 4$$

$$T(3,\{2,4\}) = \min\begin{cases} (3,4) + T(4,\{2\}) \\ (3,2) + T(2,\{4\}) \end{cases} = \min\begin{cases} 5+5 \\ 2+5 \end{cases} = 7$$

$$T(4,\{2,3\}) = \min\begin{cases} (4,2) + T(2,\{3\}) \\ (4,3) + T(3,\{2\}) \end{cases} = \min\begin{cases} 4+5 \\ 1+3 \end{cases} = 4$$

$$T(3,\{4\}) = (3,4) + T(4,1) = 5+3 = 8$$

$$T(4,\{3\}) = (4,3) + T(3,1) = 1+1 = 2$$

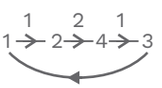
$$T(2,\{4\}) = (2,4) + T(4,1) = 2+3 = 5$$

$$T(4,\{2\}) = (4,2) + T(2,1) = 4+1 = 5$$

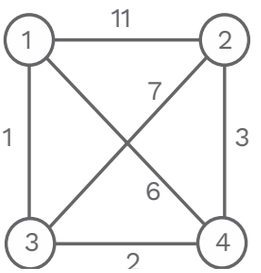
$$T(2,\{3\}) = (2,3) + T(3,1) = 4+1 = 5$$

$$T(3,\{2\}) = (3,2) + T(2,1) = 2+1 = 3$$

So, minimum cost is 5 and the path is



10. Find the length of the shortest path between all pair vertices for the given graph G.



(A)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(B)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 3 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(C)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(D) None of above

Solution: (A)

Let D^i = set of all shortest path between every pair in such a way that the path is allowed to go through node 0 to node i.

So,

$D^0=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^1=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^2=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^3=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 1 & 3 \\ 8 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^4=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Chapter Summary

- The major components of dynamic programming are:

  Recursion: Solves subproblems recursively.
  Memoization: Stores the result of sub-problems.
  Dynamic programming = Recursion + Memoization
- The two dynamic programming properties, which tells whether it can solve the given problem or not are: Optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems.

  Overlapping subproblems: When a large problem is break down into smaller sub-problems, then there are many some sub-problems which are the same and repeated many times; these are known as overlapping sub problem.
- Basically, there are two approaches for solving DP problems:
  - Bottom-up dynamic programming
  - Top-down dynamic programming

  Bottom-up dynamic programming: In this method, we evaluate the function starting with the smallest possible input argument value, and then we step through possible values, and slowly increase the input argument value.
  While computing the values, we store all computed values in a table (memory).
  As larger arguments are evaluated, pre-computed values for smaller arguments can be used.
- Top-down dynamic programming: In this method, the problem is broken into subproblems, and these subproblems are solved, and the solutions are remembered, in case they need to be again solved. Also, we save the each computed value as the final action of a recursive function, and as the first action we check if the pre-computed value exists.
- Example of dynamic programming
  - Fibonacci series
  - Matrix chain multiplication
  - Longest common subsequence
  - Subset sum problem
  - 0/1 knapnack
  - Multistage graph
  - Travelling salesman problem
  - All-pairs shortest path Floyd–Warshall
- Fibonacci Series:
  Recurrence equation:
  Fib(n) = 0, if n = 0
  = 1, if n = 1
  = Fib(n-1) + Fib(n-2), if n>1
  Solving the fibonacci series by using dynamic programming takes time and space complexity as O(n).
- For all problems, it may not be possible to find both top-down and bottom-up programming solution.

Matrix Chain Multiplication

Problem: Given a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ with their dimensions, what is the best way to parenthesize them so that it produces the minimum scalar multiplication.

Number of ways we can parenthesis the matrix = $\frac{(2n)!}{(n+1)!n!}$

Where n = number of matrix – 1

Let M[i, j] represents the least number of multiplications needed to multiply $A_i \ldots A_j$.

$$M[i, j] = \begin{cases} 0, \text{if } i = j \\ \text{Min}\{M[i,K] + M[K+1, j] + P_{i-1}P_KP_j\} \text{ if } i < j \end{cases}$$

- Bottom-up matrix chain multiplication
  Time complexity = $O(n^3)$
  Space Complexity = $O(n^2)$.
  Top down dynamic programming of matrix chain multiplication
  Time complexity = $O(n^3)$
  Space Complexity = $O(n^2)$
- Longest common subsequence: Given two strings: string X of length m [X(1...m)], and string Y of length n [Y(1...n)], find longest common subsequence: The longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings for example X = "ABCBDAB" and Y = "BDCABA", the LCS(X,Y) = {"BCBA", "BDAB", "BCAB"}
- Brute force approach of longest common subsequence is $O(n2^m)$
- Recursive equation of the LCS is

$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j, \\ \max\left(C[i, j-1], C[i-1, j]\right), \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Dynamic programming approach of longest common subsequence takes $\theta(mn)$ as both time complexity and space complexity.
- **Multistage graph:** It is a directed graph in which the nodes can be divided into a set of stages such that all edges are from one stage to the next stage only, and there will be no edges between vertices of the same stage.
  Greedy methods fails to find the shortest path from source to destination in multistage graph.
- Let 'S' be the source node and 'T' be the target node and let at stage 2 have nodes A, B and C. So, the recursive equation look like as shown below:

$$M(N-1, i) = i \rightarrow T$$

$$M(1, S) = \text{minimum} \begin{cases} S \rightarrow A + M(2, A) \\ S \rightarrow B + M(2, B) \\ S \rightarrow C + M(2, C) \end{cases}$$

  M is a function which represents the shortest path with minimum cost.
- Time complexity of bottom-up dynamic programming of multistage graph problem is O(E).

0/1 knapsack problem: In this problem, 'n' distinct objects are there, which are associated with two integer value s and v where, s represents the size of the object and v represents the value of the object. We need to fill a knapsack of total capacity w with items of maximum value. Hence, we are not allowed to take partial of items.
Let us say KS(i,w) represent knapsack. Here, i is the ith elements to be considered, and the capacity of knapsack remaining is w.
The recurrence equation is:

$$KS(i,w)=\begin{cases}\max\left(P_i+KS(i-1,w-w_i),KS(i-1,w)\right);w_i\le w\\0;i=0\ \text{or}\ w=0\\KS(i-1,w);w_i>w(w_i\ \text{is the weight of}\ i^{th}\ \text{element})\end{cases}$$

Time complexity of 0/1 knapsack = minimum $\begin{cases}O(2^n)\\O(nw)\end{cases}$ (n is the number of object and w is the total capacity of the knapsack.)

- **Subset sum problem:** Given a sequence of n positive numbers A_1, A_2,....A_n, give an algorithm which checks whether there exists a subset of A whose sum of all numbers is T.
- Brute force method time complexity of subset sum problem is $O(2^n)$.
- Let us assume that SS(i,S) denote sum from a_1 to a_i whose sum is equal to some number 'S'.
- Recursive equation of subset-sum is given below:

$$SS(i,S)=\begin{cases}SS(i-1,S);S<a_i\\SS(i-1,S-a_i)\ \text{VSS}(i-1,S);S\ge a_i\\\text{true};\ S=0\\\text{False};i=0,S\ne 0\end{cases}$$

- Time complexity of subset-sum = minimum $\begin{cases}O(2^n)\\O(nw)\end{cases}$

All Pair Shortest Path Floyd Warshall

Problem: Given a weighted directed graph G=(V,E), where V = {1, 2, ...,n}. Find the shortest path between all pair of nodes in the graph.

- The running time of Dijkstra's algorithm if we use to find all pairs shortest path is O(VElogV).
- The running time of Bellman-Ford algorithm if we use to find all pair shortest path is $O(V^2E)$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ...,k}
The recurrence equation is:

$$d_{ij}^{(k)}=\begin{cases}w_{ij} & ,\text{if}\ k=0\\\min\left(d_{ij}^{(k-1)},d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right),\text{if}\ k\ge 1\end{cases}$$

- The time complexity of all pairs shortest path Floyd–Warshall in is $O(V^3)$. (where V = number of vertices)
- The space complexity of all pairs shortest pat Floyd–Warshall is $O(V^3)$, but it can be reduced to $O(V^2)$. (where V = number of vertices).