Subrajana

1

r.
ASHWINI

Algorithm and Complexity

$T(n) = 2T(n/2) + n$ $(n \log_2 n)$ KALPITA

$T(n) = T(n/5) + T(4n/5) + n$ $(n \log_{5/4} n)$

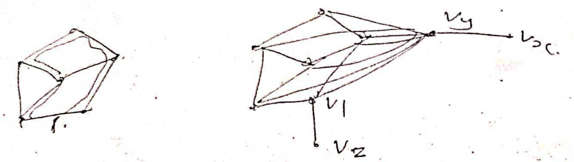
$= 3T(n/4) + cn^2$ (n^2)

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

$= 3T(n/2) + n.$ $(n \log_2 n)$

$= T(n/6) + T(5n/6) + n.$ $(n \log_{6/5} n)$

$= T(n/3) + T(2n/3) + n.$ $(n \log_{3/2} n)$



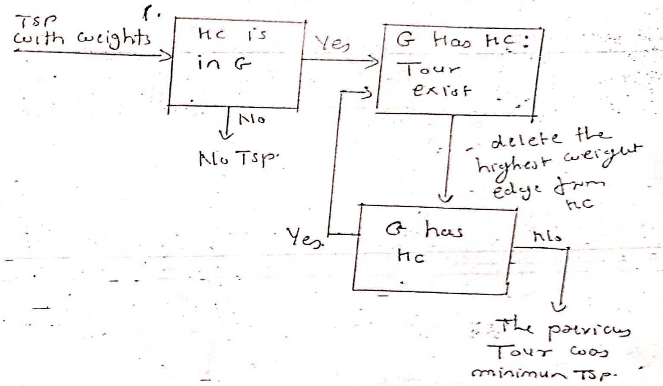
Appr. of chinise remainder

Greedy with heuristic Approach.

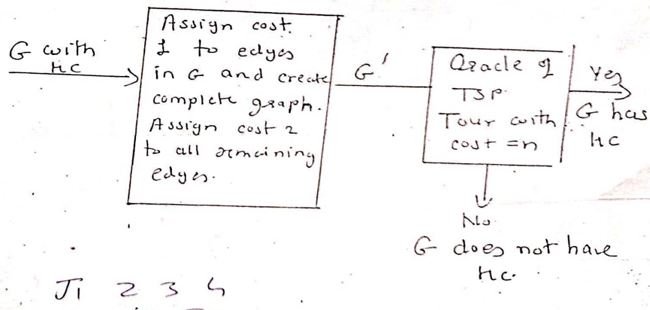
- Fibonacci series

FB Tree insert

HC is NP Hard:



TSP is NP Hard.

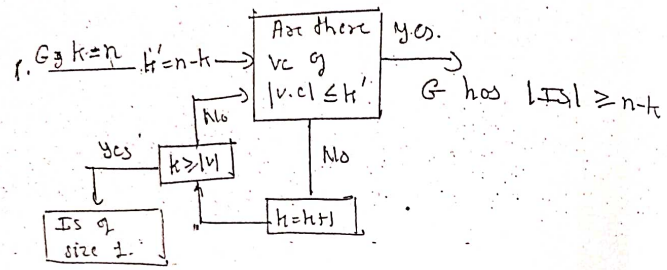


J1	2	3	4
16	15	5	8
1	2	1	1

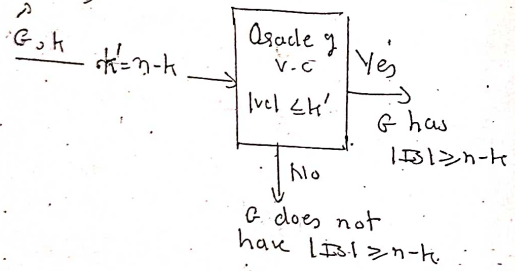
J1	J2	J3
1	2	3

Vertex Cover is NP Hard.

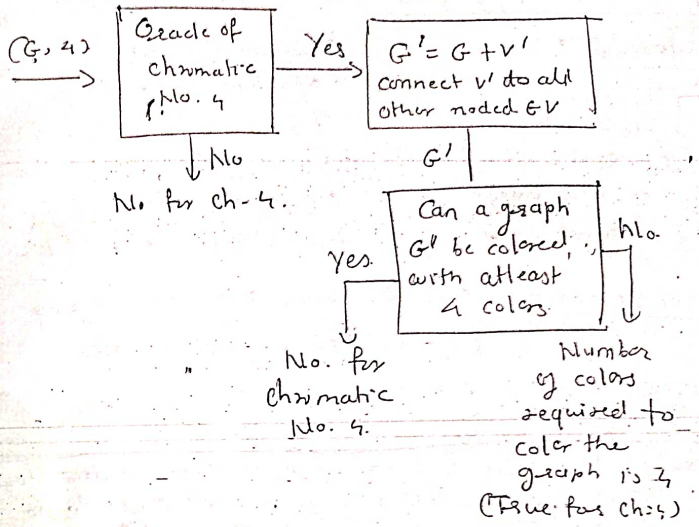
Search Version of VC is NPh



Decision Version of V.C.



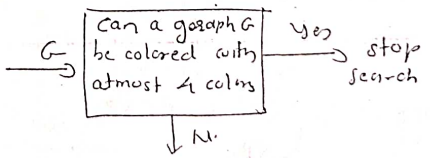
Chromatic number 4 is NPh.



Search

For a graph G find number of colors ≤ 4. if it exist.

If decision version has polynomial time algorithm, then search has a polynomial time algorithm.



Sat is - Np hard
NP Co

Np-Hard & NP-Complete

Polynomial Time

- Linear Search - n
- Binary -"- - log n
- Insertion Sort - n^2
- Merge Sort - n log n
- Matrix Multiplication - n^3

Exponential Time

- 0/1 knapsack - 2^n
- TSP - 2^n
- SOS - 2^n
- Graph coloring - 2^n
- hamiltonian - 2^n

Scope of Research - O(1) for Search, Sort, multi
Polynomial time for Expo time algo

$n^{10} < 2^n$
$n^{100} < 2^n$

when research is going continuously we want work done should be usefull so we have Guideline or framework made to do research on this algo i.e NP Hard & NP-Complete

2 point Guideline → Show similarity (relation) in all Expo time algo so that if one is solved all will be solved
→ write Non deterministic algo (polynomial time)

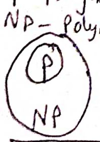
Non-deterministic for 0/1 knapsack (preserve research work for future scope)

```
Algo NSearch(A, n, key)
{
   j = Choice(); ND - 1
   if (key = A[j])
   {
      write(i);
      Success(); ND - 1
   }
   else { write(0);
      failure(); } ND - 1
}
                 O(1)
```

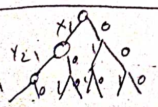
1	5	7	8	3
0	1	2	3	4

Majic → Technique logic

P - Polyno + deterministic
NP - Polyno + ND



P ⊆ NP



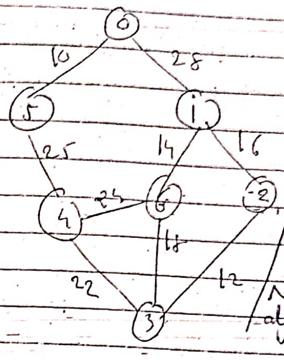
Base problem - Satisfiability - CNF - propositional calculus formula using boolean
{x_1, x_2, x_3} - x_i

Eg $CNF = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$ → what value of x_i is true
disjunction, conjunction (AND)

NP-Hard relationship can be shown using reduction
NpH → Sat $\propto$ 0/1 NpH   Sat $\propto$ L
I_1 I_2 L → Np hard
transitive prop - reduction on $L_1 \propto L_2$

Sat of L1 → P

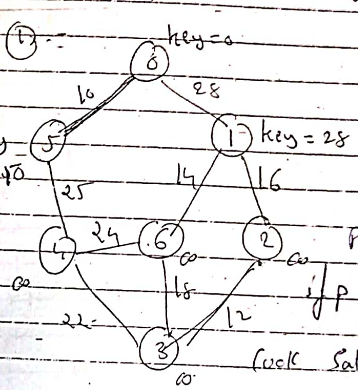
Evaluate minimum spanning tree using Prim's al



Initially add root node 0 to V-Q.

relation
└ Np hard } - research
Np complete
ND → P → inversion
algo

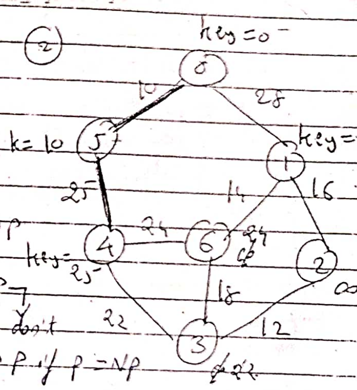
(1)



P ⊆ NP
if P = NP → don't
Cook Sat is in P if P = NP

key[5] = 10 which is minimum so add node 5
and adj[5] = {4}
so key[4] = 25

(2)

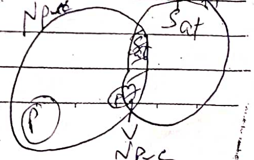


Now minimum is 25
so u = 4, add node 4.
→ adj[4] = {6, 3}.
key[6] = 24
key[3] = 22

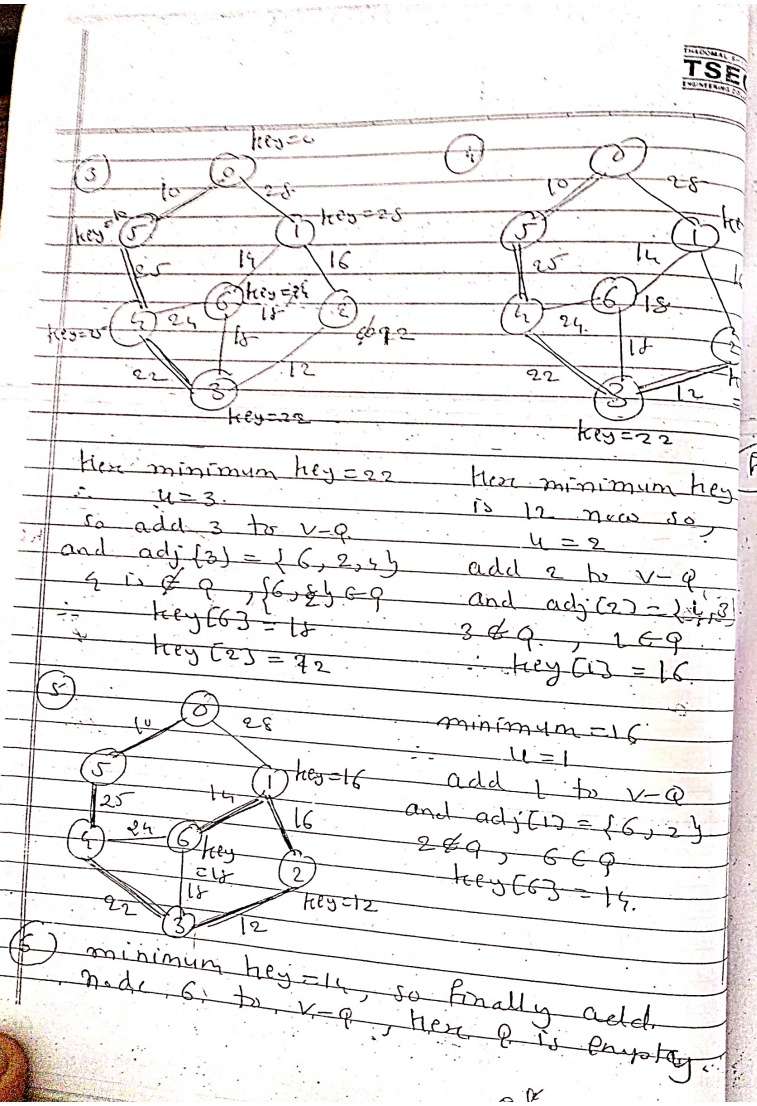
O/P P = 10, 8, 12 M = 8
w = 5, 4, 3
$x_i = (w_1, w_2, w_3)$

x_1	x_2	x_3
0	0	0
0	0	1
0	1	1

8 (2^3) 2^n → similar to Exp algo



Cook's theorem



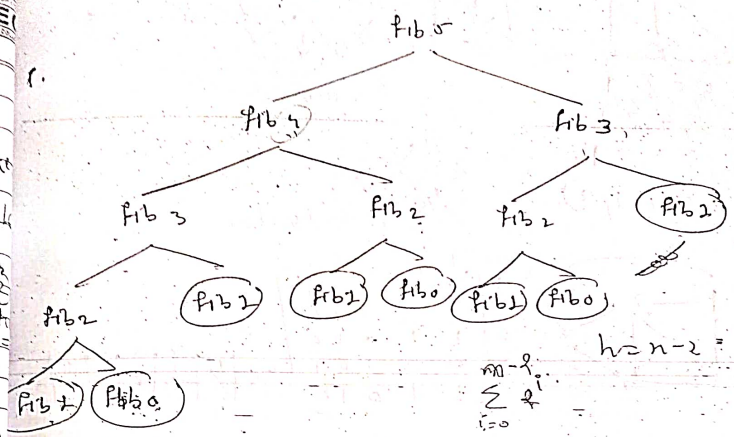
Here minimum key = 22
∴ u = 3.
so add 3 to V-Q
and adj[3] = {6, 2, 4}
4 is ∉ Q, {6, 2} ∈ Q
∴ key[6] = 18
key[2] = 12

Here minimum key is 12 now so,
u = 2
add 2 to V-Q
and adj[2] = {4, 3}
3 ∉ Q, 1 ∈ Q
∴ key[1] = 16

minimum = 16
∴ u = 1
add 1 to V-Q
and adj[1] = {6, 2}
2 ∉ Q, 6 ∈ Q
key[6] = 14.

minimum key = 14, so finally added node 6 to V-Q, Here Q is empty.

Fib (n) = Fib(n-1) + Fib(n-2)



h = n-2

$\sum_{i=0}^{n-2} 2^i$

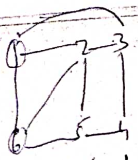
n	fn call
0	1
1	1
2	3
3	5
4	9
5	15
6	!
7	!

0 1 2 3 4 5 6 7 8 9 (Fib)

1 1 2 3 5 8 13 21

Complexity = $O(2^{h}-1)$ cf
= $O(2^{n-1})$ hhb
= $O(2^{n-1})$

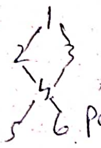
Np hard
exponential



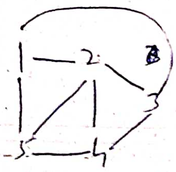
X Junction, Articulation Point

1, 2, 3, 4, 5, 6, 1
1, 2, 6, 5, 4, 3, 1
1, 6, 2, 5, 4, 3, 1

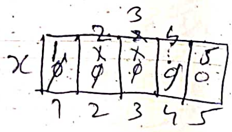
Same cycle
X (2, 3, 4, 5, 6, 1, 2)



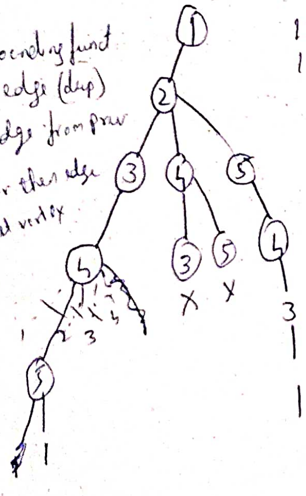
X Pendant vertices



	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

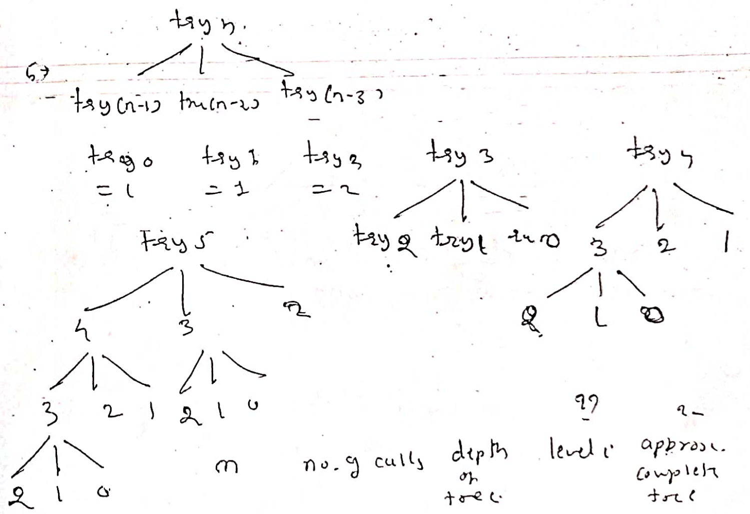


3 branching funct
Similar edge (dep)
having edges from prev
of last ver then edge
to first vertex



12345-1
12543-1

```
int try (int n)
{
   if (n < 3) return (1);
   return ( try (n-3) + try (n-1) - try (n-2));
}
main (int m)
{ int n;
  for (n=0; n<n; n++)
     print ("\n %d ", try (n));
}
```



n	no. of calls	depth of tree	level	approx. complete tree
0	0	0	1	
1	0	–	1	
2	0	–	1	
3	4	1	2	
4	7	2	3	
5	13	3	4	
6	24	4	5	

$O(n) = (3^{n-1})$

(1) if $m < 0$

$O(n) = c_{h1} + c_i + c_c$ declarations : No print statements o/p

$= O(1)$.

(2) if $m = 0$

$O(n) = c_{h1} + c_i + c_c$

$O(1)$

(3) $m = 3$.

$O(n) = c_{h1} + c_i + 3[c_c + 3(c_{print} + c_f) + c_m] + c_c$.

$= c_{h1} + c_i + 3c_c + 3c_{print} + 3c_f + c_m + c_c$

$= c_{h1} + c_i + 4c_c + 3c_{print} + 3c_f$

$O(3^{n-1})$

(4) $m > 3$

$= c_h + c_i + m[c_c + c_f + c_{print} + c_m] +$

$O(3^{n-1})$.

Approximation of algorithms: (Cont.)

Many important problems are NP complete (NPC), which are likely to be quite hard to solve exactly. We cannot just forget these problems, as they are so important in practical life.

We can try several approaches.

1. Try exponential time algorithm:

An optimal solution is found. But it is not feasible if the problem size is large.

2. Try general optimization methods:

e.g: branch-and-bound, genetic algo, neural nets. Some is hard to show how good they are compared with the optimal solution.

3. Try approximate Algorithms.

Generally fast, but may not get an optimal solution. But they can be proved to be close to the optimal solution.

So among many approaches, one approach to solving NPC-optimization problems is the use of fast (i.e polynomial bounded) algorithms that are not guaranteed

do give best solution but will give one that is close to the optimal. Such algorithms are called approximation algorithms.

In many applications an approximate solutions is good enough, especially when the time required finding an optimal solution is considered.

The goal of an approximation is to come as close to the optimal value as possible in reasonable amount of time. i.e

1) For the travelling salesman problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find the shast cycle.

2) For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices and the approximation problem is to find the vertex cores with few vertices.

Performance Ratio:

Suppose we work on an optimization problem where each solution carries a cost. An approximate algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For eg; we are looking for a minimum size of vertex-cover (VC). An approximate algorithm returns a VC for us, but the size (cost) may not be minimum.

Another example is, we are looking for a maximum size ~~of~~ Independent set (IS). An approximate algorithm returns an IS for us, but the size (cost) may not be maximum.

Let C be the cost of solution returned by an approximation algorithm, and c^* be the cost of the optimal solution.

We say that the approximate algorithm has an approximation Ratio $p(n)$ for an input size n, where

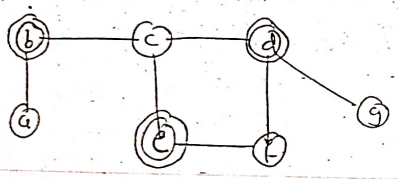
$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq p(n)$$

In otherwords, the approximation ratio measures how bad the approximate solution is compared to the optimal solution.

Example of Approximate Algorithm:

Vertex - Cover:

A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.



$VC = \{b, d, e\}$

The decision vertex-cover problem was proven NPC; we want to solve the optimal version of VC problem i.e we want to find a minimum size vertex cover of a given graph. We call such a VC an optimal VC C^*.

C^* - optimal soln
C - Approx soln

An approximate Algo:

```
Approx-VC (G)
{
    C = Empty-set;
    E' = E
    while E' is not empty do
    { let (u,v) be any edge in E';
      add u and v to C;
      remove from E' all edges incident to
      u or v;
    }
    return C;
}
```

Breadth First Search:

BFS trees have a nice property.

- Every edge of G can be classified into one of three groups.

1) Some edges are in T themselves.
2) Some connect two vertices at the same level of T.
3) And the remaining ones connect two vertices on the two adjacent levels. It is not possible for an edge to skip a level.

Therefore a BSF-search tree really is a shortest path tree starting from its root.

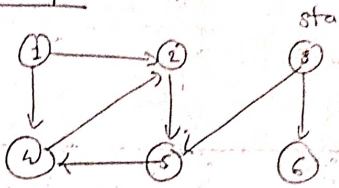
- Every vertex have a path to root, with path length equal to its level, and no path can skip a level so this really is a shortest path.
- BSF discover all vertex at level k before discovering any vertices at level k+1.
- Uses Queue.

```
BSF (G, s)
for each vertex u ∈ V[G] - {s}
   do color[u] ← white
      d[u] ← ∞
      π[u] ← NIL.
color[s] ← Gray
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE (Q, s).
while Q ≠ ∅
   do u ← DEQUEUE(Q)
      for each v ∈ adj[u]
         do if color[v] = white
            then color[v] = gray
               d[v] ← d[u] + 1
               π[v] ← u
               ENQUEUE (Q, v).
      color[u] ← Black
```

Analysis:

Total Running time of BSF is $O(V+E)$. Thus, BSF runs in linear time in the size of the adjacency-list representation of G.

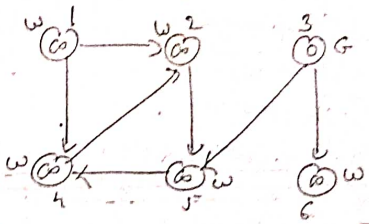
Example:



Adjacency list

1 : 2, 4
2 : 5
3 : 5, 6
4 : 2
5 : 4
6 : -

Initially

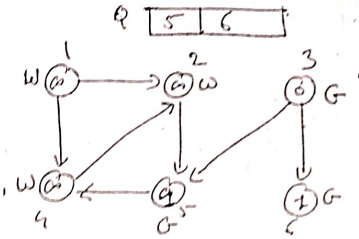


Q: [3]
color[3] = Gray

(I) u = 3

adj[3] = {5, 6}
color[5] = G d[5] = 1 π[5] = 3
color[6] = G d[6] = 1 π[6] = 3

Q [5 | 6]

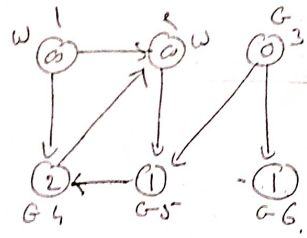


G = Gray
B = Black
W = White

(II) Now u = 5

adj[5] = {4}
color[4] = Gray
d[4] = 2
π[4] = 5

Q. [6 | 4]



(III) Now $u=6$ $adj[6] = \{-\}$

(IV) $u = 4$.

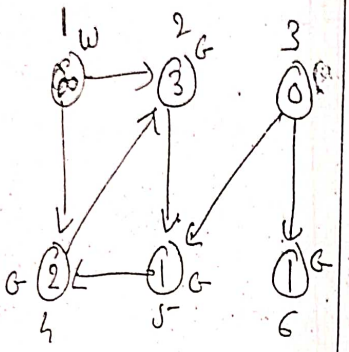
$adj[4] = 2$.

$color[2] = Gray$

$\pi[2] = 4$

$d[2] = 3$.

Q: 2.



(V) Now $u = 2$

$adj[2] = 5$

since $color[5] \neq$ white, don't process 5.

Now $\emptyset$ $Q = \emptyset$

∴ Thus vertex 1 cannot be reachable from source node 3.

Depth First Search:

General idea behind a DFS is to begin with a starting node A. First we examine the starting node A. Then we examine each node N along a path P which begins at A; that is we process a neighbour of A first, then neighbour of neighbour of A and so on. After coming to a closed end, that is to the end of the path P, we backtrack on P until we can continue along another path P! And so on---

- This algo. is similar to an inorder traversal of a binary tree.
- Here we uses a stack instead of Q.

In DFS, each vertex u has two time-stamps:

(1) $d[v]$ i.e discovery time-records
(2) $f[v]$ i.e finishing time-records

```
DFS (G).
  for each vertex u ∈ V[G]
      do color[u] ← white
         π[u] ← NIL
  time ← 0.
  for each vertex u ∈ V[G]
      do if color[u] = white
         then DFS-VISIT(u).

DFS-VISIT(u).
  color[u] ← Gray
  time ← time+1
  d[u] ← time.
  for each v ∈ adj[u]
      do if color[v] = white
         then π[v] ← u.
              DFS-VISIT(v).
  color[u] ← Black
  f[u] ← time ← time+1.
```

Analysis

Running time = $\Theta(V+E)$

Compare

BSF	DSF
(i) Examines each node in level k before examining level k+1	(i)
(ii) If goal is unknown and undefined use BSF	(ii) For a known goal use DSF
(iii)	(iii)

Analysis of algorithm control structures

The analysis of an algorithm is calculated by considering its individual instructions. The individual instructions are calculated and then according to the control structures, we combine these times.

1. Sequencing.

Assume our algo. consists two parts A & B

$t_A = O(n)$

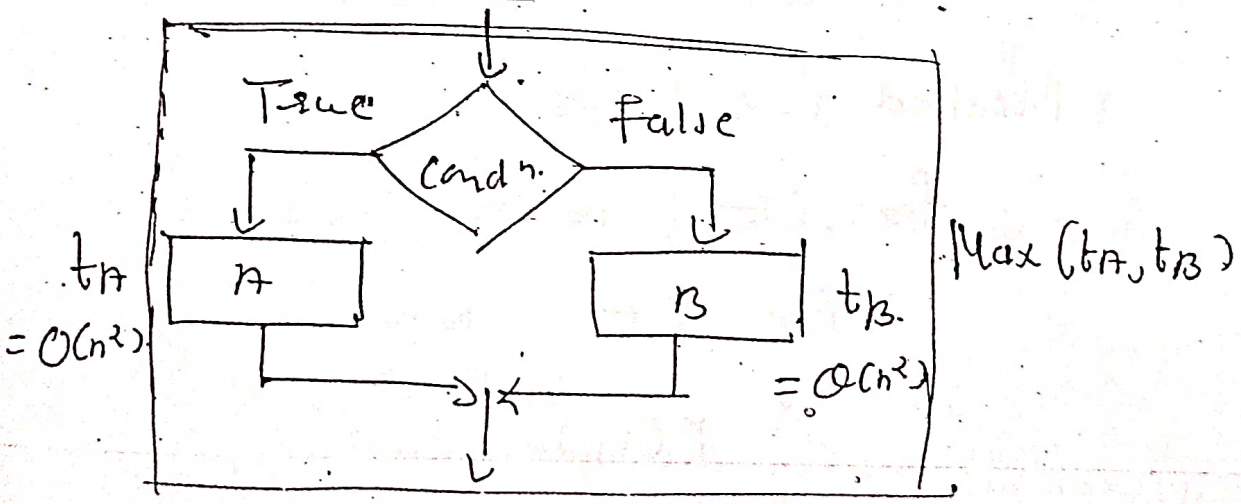
$t_B = \Theta(n^2)$.

Then computation time $= t_A + t_B$

$= \max(t_A, t_B)$

$= O(n^2)$

2. If-then-else



The total computational time is according to the conditional rule "if-then-else"

Total computation time $= \max(t_A, t_B)$

$\max(O(n^2), O(n^2)) = O(n^2)$.

3. For loop

Consider the following loop

```
for i ← 1 to m
{ p(i)
}
```

If the computation time t_i of $P(i)$ varies as a function of "i", then total time for the loop is given not by a multiplication but by a sum

i.e $\sum_{i=1}^{m} t_i$ time.

i.e $\sum_{i=1}^{m} Q(1) = Q \sum_{i=1}^{m} 1 = Q(m)$

Nested for loops

```
for i ← 1 to m
{ for j ← 1 to m
  { P(ij)
  }
}
```

$\sum_{i=1}^{m} \sum_{j=1}^{m} t_{ij} = Q(m^2)$

Program for Binary Search

```
1. binary-search (A[], N, x).
   begin
      start = 1, end = N, found = 0;
   while ((start ≤ end) && found == 0)
   begin
      mid = (start + end)/2;
      if (A[mid] == x) then
         element found at mid;
         found = 1
         return 1;
      else if (A[mid] > x) then
         end = mid - 1;
      else
         start = mid + 1.
      endif
   end
      return 0;
```

Complexity.

① Best case occurs when element is found at mid.
∴ $\Omega(1)$.

② Worst case

Each comparison reduces the sample size in half.

∴ $= O(\log_2 n)$.

Example   key $x = 33$

11 33 54 6 9 8 12

↑ start   ↑ mid   ↑ end

$x <$ mid $\Rightarrow$ end = mid − 1

11 33 54 6 9 8 12

Example $x = 9$

6 9 12 15 20 25 30

↑ start   ↑ mid   ↑ end

Here $x <$ mid

∴ end = mid − 1

6 9 12 15 20 25 30

↑ start ↑ mid ↑ end

x = mid   Hence found = 1

Question: Explain the practical applicability of "Complexity of Analysis of algorithms" (04)

OR

Write short note on "applicability of Algorithm complexity".

Soln

Algorithms can be evaluated by a variety of criteria. Most often we are interested in the rate of growth of the time or space required to solve larger instances of programs.

- The time needed by an algorithm is expressed as a function of the size of a problem is called time complexity of the algorithm. The limiting behavior of the complexity as size increases is called the asymptotic time complexity.
- Analogous definitions can be made for space complexity and asymptotic space complexity.

Practical Applicability of Algorithms.

1) The Human Genome project has the goal of identifying all the 100000 genes in human DNA, determining the sequences of 3 billion chemical base pairs that make up human DNA, storing this info. in a database and developing tools for data analysis.

- Each of the above mentioned steps require a sophisticated algorithms.

2) The Internet enables people all around the world to quickly access and retrieve large amounts of information. In order to do so clever algorithms are employed to manage and manipulate this large volume of data. Examples of problems which must be solved include
- finding good routes on which data will travel
- & using a search engine quickly find pages on which particular information resides.

3) In matrix multiplication when we are given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices. We wish to determine their product $A_1 A_2 \ldots A_n$. Because matrix multiplication is associative, there are several legal multiplication orders. The no. of all possible multiplication orders is exponential in n, and so trying all possible orders may take a very long time. So we need to determine an efficient algorithm which will give us the order in which the matrices should be multiplied (an optimal soln).

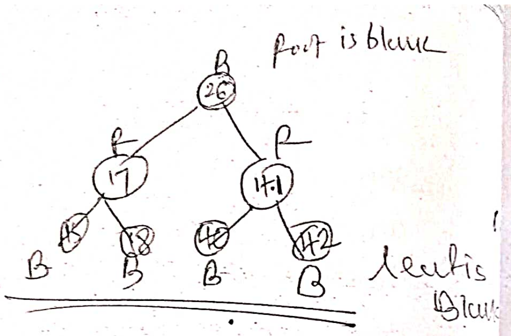
Analysis of complexity of Algorithm:

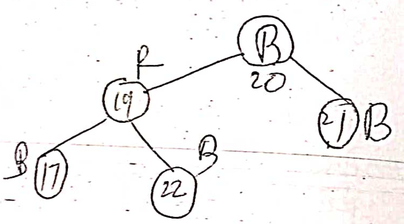
Our usual measure of efficiency of any algorithm is a speed; i.e how long an algorithm takes to produce its result. However, there are some problems, for which no efficient solution is known. Such problems are known as NP-complete.

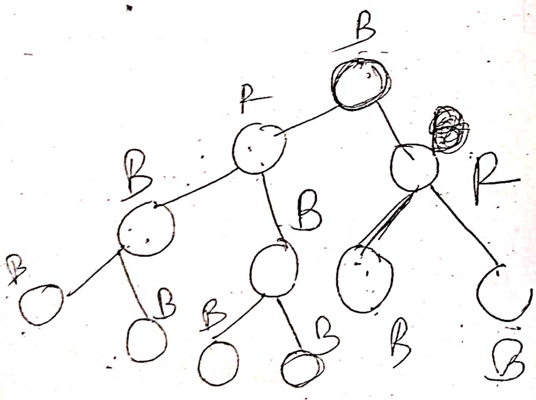
It is unknown, whether or not efficient algorithms exist for NP-complete problems. And the set of NPC problems has a remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exists for all of them.

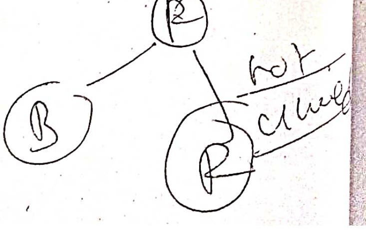
The several NPC problems are similar, but not identical, to problems for which we do not know of efficient algorithm.

- It is valuable to know about NPC problems bcos some of them arise often in real applications. If you are called upon to produce an efficient algorithm for an NPC prob, you are likely to spend a lot of time in a fruitless search. If you can show that the problem is NPC, you can instead spend your time developing an efficient algorithm that gives a good, but not best possible, solution.









Red Black Trees:

A Red-Black tree is a binary search tree with one extra bit of storage per node: that is color, which can be either RED or BLACK.

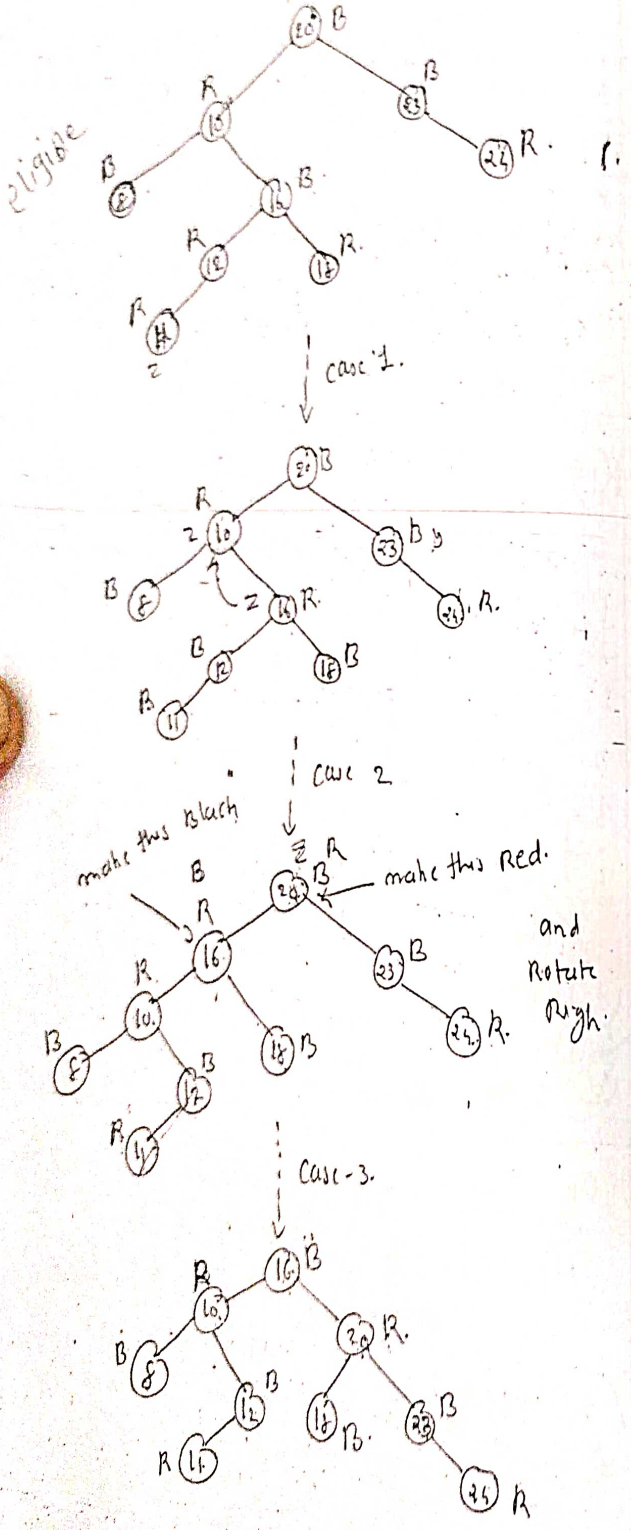
- Red Black tree ensures that no path from the root to a leaf is more than twice as long as any other, so the tree is approximately balanced.
- Each node on the tree have five fields color, key, left, Right, and p.

Properties:

1. Every node is either Red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is Red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Appln
Data structure — computation geometry
Complete fair scheduling used in Linux kernel
Functional programming — to construct Associative Array
O(log n)

Insertion (suppose node z is to be inserted).



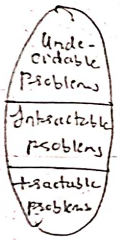

```
RBINSERT-FIXUP (T, z)
while (color[p[z]] == RED)
    do if (p[z] = left[p[p[z]]])
        then y ← Right[p[p[z]]]
            if color[y] == RED
            then color[p[z]] = BLACK
                 color[y] = BLACK.
                 color[p[p[z]]] = RED.
                 z ← p[p[z]].                    } case 1
        else if (z == Right[p[z]])
            then z ← p[p[z]]
                 left-Rotate(T, z).              } case 2
        color[p[z]] ← BLACK
        color[p[p[z]]] ← RED.
        Rotate-Right(T, p[p[z]]).                } case 3
    else (same as then clause
          with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

1. Insert a new node the way it is done in binary search tree
2. Color the node Red.
3. If an inconsistency arises for the Red-Black tree, fix the tree according to the type of discrepency. (case 1, case 2, case 3)

Problem Complexity:

The universe of problems may be broadly divided into the following categories

1) Undecidable
2) intractable
3) tractable



(1) Example of undecidable problem are the
- halting problems of Turing machine.
- Ward correspondence problem
- tiling a rectangular area with given tiles.

(2) The intractable problems includes
- finding Hamiltonian cycle in a given graph.
- finding clique in a given graph
- The set partitioning problem etc.

(3) The intractable problem have non-polynomial time complexity.

(3) The tractable problems include
- sorting
- searching
- matrix multiplication etc.

They run in polynomial time.

Computational Models:

The most important motivation for formal models of computation is the desire to discover the interent computational difficulty of various problems.

We would like to prove lower bound on computation time.

For each problem we must select an appropriate model which will accurately reflect the actual computation time on a real computer.

There are basically three computational models.

1] RAM (Random Access Machine)
2] RASP (Random Access stored Program)
3] Turing machine.

RAM

(1) A random Access m/c is models a one-accumulator computer in which instructions are not permitted to modify themselves.

(2) A RAM consists of
- Read only input-tape
- a program
- a memory
- and a write-only output tape.

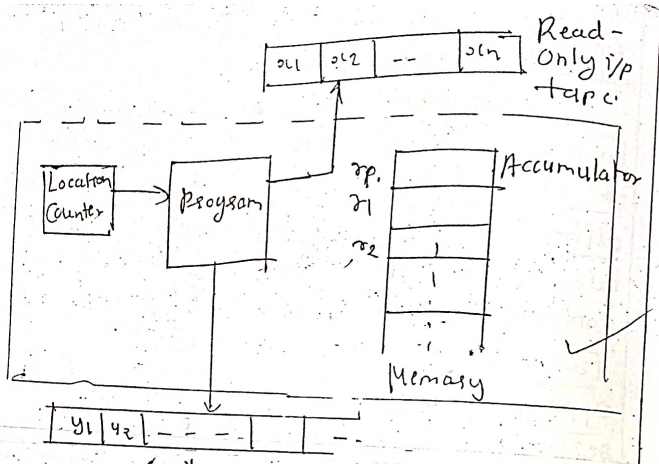


Fig: A RAM machine.

(3) The input tape is a sequence of squares each of which holds an integer. Whenever a symbol is read from i/p tape, the tape head moves one square to the right.

(4) The o/p tape is write-only tape divided into squares which are initially all blank. When a write instruction is executed, an integer is printed in the square of the o/p tape that is currently under the output tape head, and the tape head is moved one square to the right. Once an output symbol has been written it cannot be changed.

(5) The memory consists of sequence of registers, $r_0, r_1, \ldots r_i, \ldots$ each of which is capable of holding an integer of arbitrary size. We place no upper bound on the no. of registers that can be used. This abstraction is valid in case where:

1. The size of the problem is small enough to fit in the main memory of a computer.
2. The integer used in the computation are small enough to fit in one computer word.

(6) The program for a RAM is not stored in RAM memory. Thus we are assuming that the program does not modify itself. We assume there are arithmetic instructions, input-output instructions, indirect addressing and branching instructions. All computations take place in the first register r_0 called the accumulator.

Opcode	Address
LOAD	operand
STORE	"
SUB	"
WRITE	"
⋮	
JUMP	label
JZERO	label

RAM Instructions

Each Instruction consists of two parts:
- operation code
- address.

An operand can be one of the following:
1. $=i$, indicating the integer i itself
2. A non-negative integer i, indicating the contents of register i.
3. $*i$, indicating indirect addressing.

The Location counter is used to determine the next instruction to execute.

Initially $c(i) = 0$ for all $i \geq 0$ (where $c(i)$ is the contents of register i), the location counter is set to the first instruction in P, and the output tape is all blank. After execution of k^{th} instruction P, the location counter is automatically set to $k+1$, unless the k^{th} instruction is jump or branch kind of instruction.

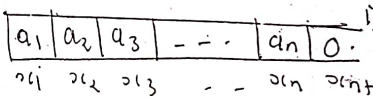
In general, a RAM program defines a mapping from input tapes to output tapes.

The mapping can be interpreted as variety of ways.
1. As a function ✓
2. As a language. ✓

Suppose a program P always reads an integers from the input tape and writes at most one integer on the output tape. If, when $a_1, a_2, \ldots, a_n$ are the integers in the first n squares of the i/p tape, P writes y on the first square of the o/p tape and subsequently halts. Then we say P computes function

$$f(a_1, a_2, \ldots, a_n) = y.$$

Another way to interpret a RAM program is as an accepter of lang. A lang. is considered as set of strings over some alphabets. We place an input string $s = a_1, a_2 \ldots a_n$ on the input tape, we place 0 a symbol of endmarker



The input string s is accepted by a RAM program P if P reads all of s and end marker, write a 1 in the first square of output tape and halts.

For input strings not in the lang accepted by P, P may print the symbol other than 1 on the o/p tape and halts.

cost criteria for RAM program:

① Uniform cost criteria:
Each RAM instruction requires one unit of time and each register requires one unit of space.

② Logarithmic cost criteria:
takes into account the limited size of real memory word.

$$d(i) = \begin{cases} \lfloor \log|i| \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

2] RASP (Random Access stored Prog. M/c).

① Since RAM programs are not stored in the RAM memory, the program cannot modify itself.

② RASP which is similar to RAM, stores the program in memory and can modify itself.

③ The instruction set of RASP is identical to that for a RAM, except indirect addressing is not permitted since it is not needed.

④ The overall structure of a RASP is also similar to that of RAM, but the program of a RASP is assumed to be in the registers of memory.

⑤ Each RASP instruction occupies two consecutive memory registers.
The first register holds an encoding of (opcode).
The second register holds the address.
- Integers are used to encode the instructions.

Inst.	Encoding
LOAD i	1
LOAD =i	2
JUMP i	3

The state of RASP can be represented by,
1. The memory map c, where c(i), i≥0 is the contents of register i and
2. The location counter, which indicates the first of the two consecutive memory ~~location~~ registers from which the current instruction is to be taken.

Initially location counter is set at some specified register. The initial contents of RAM registers are not all zeros, since the program has been loaded into the memory. Accumulator contains 0 initially. After each instruction is executed, the location counter is increased by 2 except in the case of JUMP.

The time complexity of RASP program can measured using either uniform cost criteria or the logarithmic cost. In logarithmic cost however, we must charge not only for evaluating an operand but also for accessing the instruction itself.

3] Turing Machine:

One can use a more primitive model on which the time complexity of problems is polynomially related to their complexity on the RAM model.

A Multitape Turing Machine: (TM).

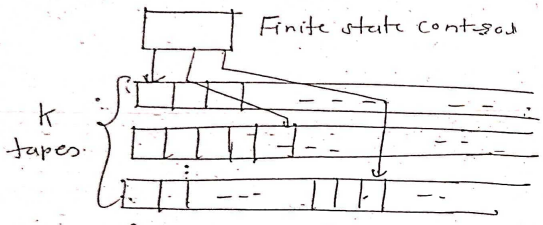


Fig A Multi tape Turing machine

(i) It consists of k numbers of tapes, which are infinite to the right.

(2) Each tape is contains cells, each of which holds one of a finite number of tape symbols.

(3) One cell on each tape is scanned by a tape head, which can read and write.

(4) Operation of the turing machine is determined by the by a primitive program called a finite control. The finite control is always in one of the a finite number of states.

One computational step of TM consists of the following.

TM may do one of the following operations

1. Change the state of the finite control.
2. Print new tape symbols over the current symbols in any or all of the cells under tape heads
3. Move any or all of the tape heads, independently, one cell (L) left or (R) right or keep them (S) stationary.

We denote k-tape TM by the seven tuple.

$(Q, T, I, \delta, b, q_0, q_f)$.

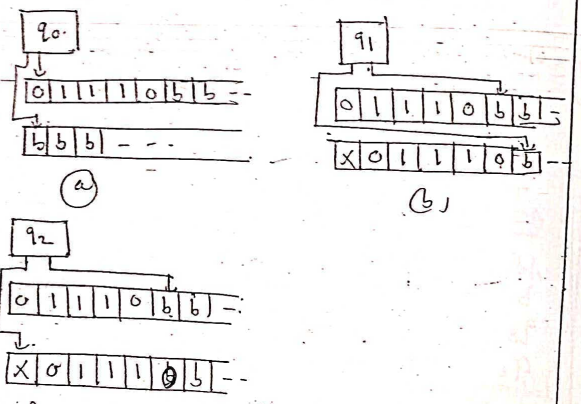
- Q – set of states
- T – tape symbols
- I – set of Input symbols: $I \subseteq T$
- δ – the next-move function
- b – is the blank
- q_0 – The initial state
- q_f – The final (accepting) state

The turing m/c can be made to recognize a lang. as follows. The tape symbols of the turing m/c include the alphabet of the lang, called the input symbols, a special symbol blank (denoted by b), and other symbols. Initially the first tape holds a string of I/p symbols, one symbol per cell, starting with the leftmost cell

All other tapes are completely blank.

The string of input symbols is accepted if and only if the TM, started at the designated initial state, with all tape heads at the left ends of their tapes, makes the sequence of moves in which it enters the accepting step.

Example Two tape turing machine recognizing palindromes.



1. The first cell on tape 2 is marked with X, and the i/p is copied from tape 1. (Fig (a))
2. The tape head on tape 2 is moved to the X (Fig (c))
3. Repeatedly, the head of tape 2 is moved right one cell and the head of tape 1 is moved left one cell, comparing the resp. symbols. If all symbols match, the i/p is a palindrome & turing m/c enters accepting state q_5. Otherwise TM enters halt without accepting.

Deterministic & Non Deterministic algo

1. In deterministic algo, the result of every operation is uniquely defined. Such algorithms agree with the way programs are executed on a computer.

We can allow algorithms to contain operations whose outcomes are not uniquely defined but are limited to specified set of possibilities. The machine executing such operations are allowed to choose any one of these outcomes subject to the termination condition to be defined later. This leads to the concept of the non-deterministic algorithm

1. choice(S) arbitrarily chooses one of the element of set S.
2. Failure() signals an unsuccessful completion.
3. Success() " " " successful completion.

The assignment statement

x = choice(1,n) could result in x being assigned one of the integers in the range [1,n]. There is no rule specifying how this choice is made.

The failure() and the success() signals are used to define a computation of the algorithm.

Searching an element in an array using non-deterministic algo.

```
j = choice(1,n);
if a[j] = x then
  { write(j);
    success();
  }
else
  { write(0);
    failure();
  }
```

Question: Define decision, search, count and optimal versions of any one NPC problem.
OR Versions of problems.

Decision problems:

(1) Any problem for which the answer is either "yes" or "no" is called as decision problem.
(2) Algorithm for a decision problem is called a decision algorithm.
(3) Decision problem asks for a limit and asks if there is a solution that has a value above (for maximization) or below (for minimization) the limit provided.
(4) Decision problems will reply with just yes or no answers.

Optimization problem:

(1) Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as optimization problem.
(2) Optimization algo. is used to solve the optimization problem.
(3) Optimization problem looks for specific result that is usually a minimum or maximum value.
(4) Optimization problem will supply as their result an answer to the problem.

Counting Problem:

The output of this class of algorithms is a natural number. This algorithms counts the number of solutions.

Search Problem: for a graph G finding number of colours $\leq$, if it exist

If decision version has $\exists$ a polynomial time algorithm then search has the polynomial time algorithm.

⇒ For every optimization problem, we have corresponding decision and counting problems.

For eg; given a description of MST problem we can have a decision problem like finding whether the given weighted graph has a minimum spanning Tree with weight $\leq k$.

⇒ The corresponding counting problem could be: how many spanning trees can be constructed for a given graph with weight $\leq k$.

⇒ And the optimization version would be finding the spanning tree with a minimum weight.

Consider the knapsack problem.

Given a profit vector $p = (p_1, p_2, \ldots p_n)$ and a weight vector $w = (w_1, w_2, \ldots, w_n)$ and a knapsack of capacity c, the decision, counting and optimization problems may be formulated as:

Decision:

Does there exist an n-tuple $(x_1, x_2, \ldots x_n) \in \{0,1\}^n$ such that

$\sum_{i=1}^{n} p_i x_i \geq k$ where k is a given profit value and $\sum_{i=1}^{n} w_i x_i \leq c$?

Counting:

How many n-tuples are there such that $\sum_{i=1}^{n} p_i x_i \geq k$, where k is a given profit value and $\sum_{i=1}^{n} w_i x_i \leq c$?

Optimization:

Find an n-tuple $(x_1, x_2, \ldots x_n) \in \{0,1\}^n$ such that $\sum_{i=1}^{n} p_i x_i$ is maximized subject to $\sum_{i=1}^{n} w_i x_i \leq c$.

Question: What are co-NP problems & state and prove any problem to be co-NP.

co-NP

A Language L is co-NP if $L' = \bar{L}$ and $L' \in NP$

Vertex Cover:

Consider a graph $G = (V, E)$ (undirected), a vertex cover is a subset $V' \subseteq V$ such that, if (u, v) is an edge in G, then either $u \in V'$ or $v \in V'$ (or both). The size of the vertex cover is the number of vertices in it.

Decision version of VC:

Does a graph $G = (V, E)$ (undirected), has vertex cover of size $\leq k$.

Let S be the set containing vertex cover of size $\leq k$ i.e $|S| \leq k$

Complement of VC or co-NP of VC.

Does an undirected graph $G = (V, E)$, has vertex cover of atleast size $k+1$.

To prove that VC is NP: let $L = VC$

Step 1: Consider graph $G = G1$

Step 2: Apply maximum operation on degree of nodes $\forall n$, where n is the no. of nodes in a graph.

i.e $n_1 = \max(\forall_n \deg(n))$

Step 3: $S \leftarrow S \cup n_i$

Step 4: $G' \leftarrow G' -$ edges (ni)

Step 5: $n \leftarrow n-1$

Step 6: Go to step 2 unless $n \neq \phi$ or all nodes are covered

Step 7: If $|S| > k$

{ Revert back to previous choice i.e Apply back tracking & start again by choosing another node.

Go to step 2. }

else { if $|s| = k$ or $|s| < k$

Go to step 8. }

Step 8: Stop and return S.

Complexity of VC:

The 1st choice of V.C. can be in n_{C_r} i.e from 'n' nodes we can get r nodes. i.e By applying greedy approach.

But if $|s| > k$ then reverting back to previous choice & applying backtracking we get a power set of nodes i.e 2^n

$\therefore$ Complexity of $VC = 2^n$.

Hence $VC \in NP$.

TSP is NP complete:

Problem: A sales person must visit n cities where every city is connected to all other cities. There is a positive cost involved in visiting a city from another. The sales person must visit each city exactly once, and return to the start city. The problem is to determine whether there is a tour of length atmost k.

Proof

Step 1: We show that TSP $\in$ NP. Generate sequence of n cities, in the tour, we check randomly that this sequence contains each vertex (city) exactly once and sum up the costs of the edges and check whether the sum is at most k. This can be carried out in polynomial time.

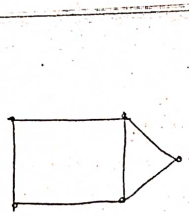
Step 2:

We select the HC as a known NPC problem to be reduced to TSP.

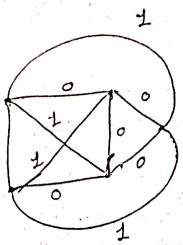
$HC \leq_p TSP$.

Let $G = (V, E)$ be an instance of the HC problem.

We construct an instance of TSP as follows.



$G = (V, E)$



$G_C = (V, E_1, \omega)$

we form a weighted complete graph $G_C = (V, E_1, w)$, where $E_1 = \{(i,j) \mid i,j \in V\}$, and define the edge cost function ω by

$$\omega(i,j) = \begin{cases} 0 & , (i,j) \in E \\ 1 & , (i,j) \notin E \end{cases}$$

The instance of TSP is then $\langle G_C, 0 \rangle$.

We show that the graph G has a HC if and only if graph G_C has a tour of cost at most 0.

1) Suppose that G has Hamiltonian cycle C. Each edge in C belongs to E and thus has a cost 0 in G_C.
Thus, C is a tour in G_C with cost 0.
Hence, if G_C has tour of cost 0 then G has a Hamiltonian cycle.

Hence if G has HC, then G_C has a Tour of cost 0.

2) Suppose that G_C has a tour c_1 of cost at most 0. Since the cost of edge in E_1 is ~~either~~ 0 and 1, the cost of tour c_1 is exactly 0. Therefore c_1 contains only edges belonging to E.
We conclude that c_1 is a Hamiltonian cycle in graph G.

Step 3 : The above reduction can be carried out in polynomial time. Hence TSP is NPC.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

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

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

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

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

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

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

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

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

RAT.

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

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

$\log_2 7 = 2.81$

$O(n^{2.81})$

Traveling salesperson problem. NP problem

We are given a set of cities & a cost to travel between each of these cities. The goal is to determine the order we should visit all of the cities (once), returning to the starting city at the end, while minimizing the total cost

application - ① the order in which streets should be visited to collect garbage efficiently, ② deciding on the route a truck should take to make deliveries in the shortest time possible.

③ choosing the routing of a packet so that information is transmitted quickly between all nodes in a network

complexity → $O(n!)$

8 cities 40320 possible orderings of cities } we have to examine all possibilities

10 cities 3628800 —"—

To find most efficient route we would have to examine all of the possibilities.

15 cities — if algo can do 100 of these calculations per sec it would take 4 centuries.

if 400 comp. 1 year

if 20 cities one billion comp. parallel working 9 months

It would be faster & cheaper to travel to all of them via any path than to wait for algo to find shortest path.

deterministic algo would take long time to complete.

TSP — Graph
city — node
path — edge
cost of travel between cities — weight

shortest path algo. — won't work for TSP.

TSP
(1) must visit to all node.
(2) must visit to a node only once.
(3) It should return to the starting point.

shortest path algo
(1) shortest path tells us the way to reach 2 nodes.
(2) can visit a node more than once.
(3) no such expectation.

TSP is NP — 2 step process would solve it.

1 step — Non deterministically generate a list of cities, diff order of cities will be generated.

algo (1) list of city name.
O(N) this process would run in O(N).
(2) generate a random no.
(3) the corresponding city name.
(4) remove it from list to prevent it from appearing twice.

IInd step — (1) determine cost of traveling to cities in the order specified.
O(N²) worst case — have a look at the cost of each successive pair of cities in the list.

both steps are executed in polynomial complexity.
So TSP is in the class NP.

Hamilton Path —
In a graph HP is one that visits every node of graph exactly once.

Hamilton circuit — if the path returns to starting node it is called a Hamilton circuit.

A graph does not need to be complete to have HP or HC.

HC ≤ TSP

each node in graph — city
edge — cost of traveling between 2 equivalent cities a value of 1.
nodes don't have edge connecting them, we assign cost of traveling = 2.
} this converts graph into cities.

we now solve TSP.
if there is HC in graph — TSP will be able to find some traveling b/w cities that have just cost of 1.
if there is no HC in graph — Soln of TSP finds will travel b/w at least one pair of cities with cost of 2.

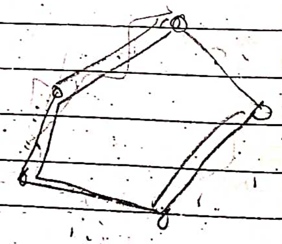
if there are N nodes in graph = that is HC if TSP path is of length N.
if there is no HC if — " — > length N.

(3) The power of reduction process is that if any NPC problem can be reduced to class P problem, all NP problems must have PT soln.
So far all such attempts have failed.

Optimisation version of TSP. — (minimisation pro...
we are interested in finding path that has low...

decision version - TSP : we would ask if there is a ...
that has a cost below some limit c.
ans will vary based on limit c.
if c is large ans yes. } In most case
if c is very small ans no } to provide ... very long ...

↓ related ...
time need to -ed
optimisation ...



Question: To prove Hamiltonian Cycle is ...
$HP \leq_p HC$.

NP Completeness:

A language L is NP-complete if-
1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$
where L' is known NPC problem
L is unknown NPC problem.

problem statement:

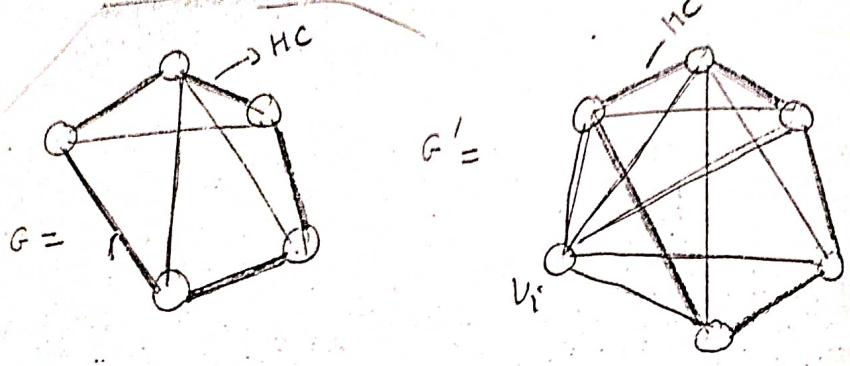
Definition (HC) : Hamiltonian ~~Path~~ cycle connects all nodes of graph exactly once.
In other words, "A hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex ~~in V~~ and visits each vertex exactly once".

HP : Hamiltonian path is a path connecting all the vertices of a graph exactly once and length of $HP = (n-1)$ for a graph of nodes n.

Decision Version: Whether or not a given graph has a hamiltonian cycle is a decision version.

Optimization Version:

To find the shortest path that covers all the vertices exactly once.



construction: (mapping).

$G(V, E)$.
V = set of vertices
E = set of edges.

$G' = (V + v_i, E + e(v_i, v), \forall v \in V)$
i.e G' be a graph
(V + v_i, E + all edges connecting v_i to all nodes)

Step 1: We need to prove that
HC ∈ NP.

HC (Hamiltonian cycle) starts at any node (n)
so to select 1st node we have (1) choice
to select IInd node we have (n-1) choices.
to select ~~IIIrd~~ node we have (n-2).
!

$1 \times (n-1) \times (n-2) \times \cdots\cdots 2 \times 1$
$= (n-1)!$

Complexity of HC = $(n-1)!$

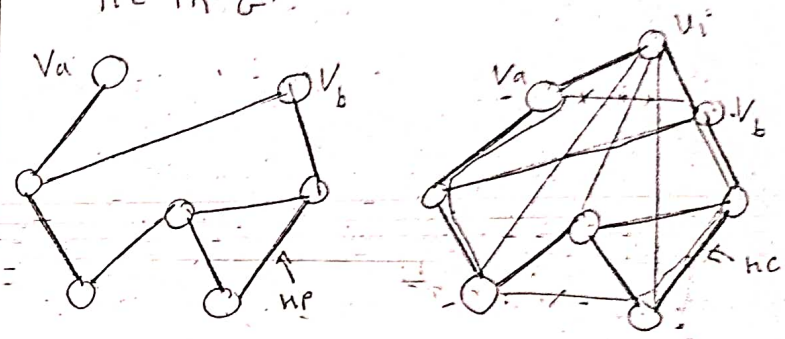
Hence HC ∈ NP

Step 2:

To prove HC is HP complete, we use HP as a known NPC problem.

$HP \leq_p HC$

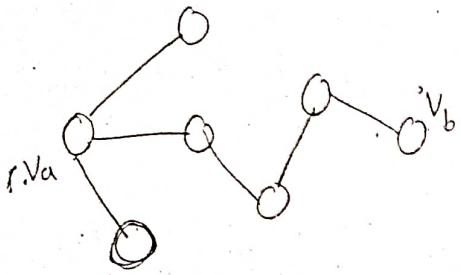
a) If there exist HP in G, then there exist HC in G'.



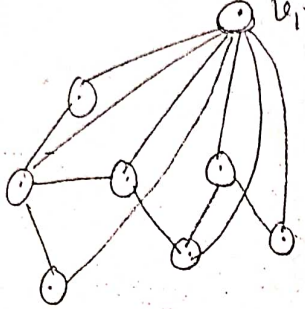
If G has HP then any of 2 neighbors of v_i in G' would be end points of that HP. Thus, G' will have HC connecting to these two end points of HP in G through vertex v_i.

Hence, If G has HP, then G' has HC. proved.

b) If G has no HP, then G' has no HC.
That is, If G does not have HP then G' can have at most HP but not HC.



Instance of G

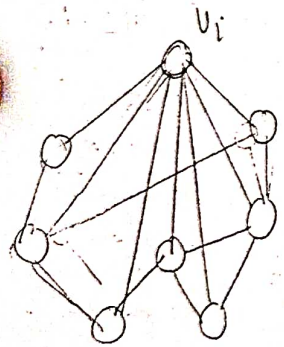


Instance of G'

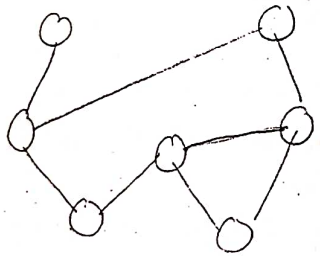
If cycle exist in G', then it will have to pass through at least one node twice.

Hence if G does not have HP, G' cannot have HC

c) If there exist HC in G', then there exist HP in G.



instance of G'



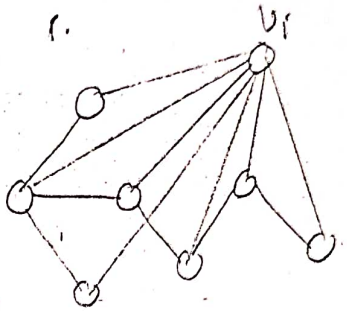
Instance of G

If G' has HC then elimination of v_i from G' and all its neighbouring edges, if does not give HP, it implies that v occused at least twice in the cycle, which is contradiction.

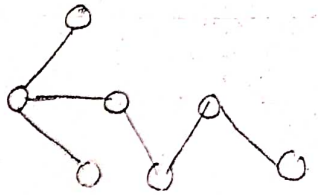
Hence if G does not have HP, G' cannot have HC.

Hence, if G' has HC, G Has HP.

d) If G' does not have HC, then G has no HP.

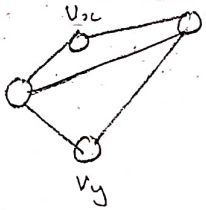


Instance of G'



Instance of G

If G' does not have HC, it means that at some pair of points other than V_i, cycle is left uncomplete.



As V_i is connected to all

neither $V_{3c} = V_i$

nor $V_y = V_i$

Some pair of nodes in G, are not connected by an edge.

∴ G cannot have HP.

Hence HC is an NPC problem.

Applications:

1) Distribution of Question papers in University Examination.
2) Vehicle routing problem.

Question: Prove that clique is NPC

$VC \leq_p$ clique
(Known) (Unknown)

NP-Completeness:

A lang. L is NP-complete if

1. $L \in NP$ and
2. $L' \leq_p L$ for every $L' \in NP$

where L' is a known NPC problem.
L is an unknown NPC problem.

Clique

Definition: Clique:

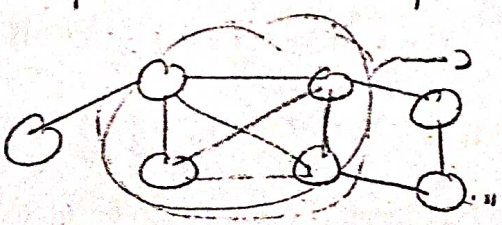
Given a graph $G = (V, E)$ if there is a subset $C \subseteq V$ which contains vertices which are completely connected ~~to each other~~ (all are connected to each other directly) by edges in E, then C is called a clique.

Decision Version:

Given a graph G, does a graph contain a clique of size $\geq k$ is the decision version of VC problem.

Optimization Version:

Finding the maximum ~~size~~ value of k is the optimization problem of clique.



clique of size 4.

Step 2: To prove karp reducibility we select ~~et~~ vertex cover as a known NPC problem.

$VC \leq_p$ clique

VC A vertex cover VC of an undirected graph $G(V, E)$ is a subset $S \subseteq V$ such that if (u,v) is an edge in G, then either $u \in S$ or $v \in S$. $\sim (u, v) \in S$

Mapping

Construct a graph $G' = (V, E')$ which is the complement of graph $G = (V, E)$. Where if $(u,v) \in E'$ then $(u,v) \notin E$.

Relationship between VC & Clique.

If there is a VC of size k in a graph G then there will be a clique of size (n-k) in G' (i.e in complement of G)

a) If G has $|vc| \leq k$ then, G' has a clique of size $|clique| \geq n-k$.

i.e $|vc| \leq k$

$-|vc| \geq -k$

$n - |vc| \geq n-k$

$\therefore |clique| \geq n-k$

Hence, if G has $|vc| \leq k$ then, G' has a $|clique| \geq n-k$ is proved.

b) If G does not have a ~~|vc|~~ $|vc| \leq k$, then G' will not have $|clique| \geq n-k$

$\therefore$ Given $|vc| > k$

$\therefore -|vc| < -k$

$n - |vc| < n-k$

$\therefore |clique| < n-k$

Hence, If G does not have a VC of size $\leq k$ then G' cannot have a clique of size $\geq n-k$

c) If G' has a $|clique| \geq n-k$, then G has a $|vc| \leq k$.

Given $|clique| \geq n-k$

$-|clique| \leq k-n$

$n - |clique| \leq k$

$\therefore |vc| \leq k$

Hence proved.

d) If G' does not have a $|clique| \geq n-k$ then G cannot have $|vc| \leq k$

That is to prove if G' has a $|clique| < n-k$ then G has $|vc| > k$.

Given $|clique| < n-k$

$\therefore -|clique| > k-n$

$n - |clique| > k$

$|vc| > k$

Hence proved.

All of the above four arguments proved that clique is NPC.

Step 1 (We show that clique problem belongs to NP.

We guess (a (non-deterministic)) subset X of the vertices V of the graph. Checking whether X is a clique, can be accomplished in polynomial time by checking whether for every pair of vertices $u, v \in X$, the edge (u,v) belongs to E and counting that the number of vertices in X is at least k.

This can be done is polynomial time Hence. clique $\in$ NP.

Note The problem is NP if it can be solved using non-deterministic algo. in polynomial time.

Question: Prove that Vertex Cover is NPC, using clique as a known problem.

clique $\leq_p$ VC.

NP-completeness.

A lang L is NPcomplete if

1. $L \in NP$.
2. $L' \leq_p L$ for every $L' \in NP$

where L' is a known NPC problem
L is an unknown NPC problem.

Problem statement:

definition (VC): A vertex cover VC of an undirected graph $G(V,E)$ is a subset $S \subseteq V$ such that if (u,v) is an edge in G, then either $u \in S$ or $v \in S$.

definition (clique).

Given a graph $G = (V,E)$ if there is a subset $C \subseteq V$ which contains vertices which are all connected to each other by edges in E, then C is called a clique.

Decision version:

Given a graph G whether or not G has a vertex cover of size $\leq k$ is the decision version of VC problem.

Optimization Version:

Finding the minimum value of k. is the optimization version of problem.

Step 1: Write NP algorithm

Prove VC ∈ NP.

Step 2: Mapping.

Construct a graph $G' = (V, E')$ which is the complement of graph $G = (V, E)$ where if $(u,v) \in E'$ then $(u,v) \notin E$

Relationship between Clique & VC.

If there is a clique of size k in a graph G then there will be VC of size (n-k) in G' (i.e complement of graph 'G').

→ The reduction algorithm takes an instance $\langle G, k \rangle$ of clique problem as input and computes G'.

Output of reduction algo. is $(G', |V|-k)$ of VC problem.

$G = (V, E)$ → [Reduction] → $(G', |V|-k)$, $G'(V, E')$

[Clique $\leq_p$ VC] ✓

a) If G has a $|\text{clique}| \geq k$ then G' has $|VC| \leq n-k$

$|\text{clique}| \geq k$

$-|\text{clique}| \leq -k$

$n - |\text{clique}| \leq n-k$

$\therefore |VC| \leq n-k$

Hence proved.

b) If G does not have an $|\text{clique}| \geq k$ then G' does not have $|VC| \leq n-k$

Given $|\text{clique}| < k$

$-|\text{clique}| > -k$

$\therefore n - |\text{clique}| > n-k$

$|VC| > n-k$

c) If G' has a $|VC| \leq n-k$ then G has a $|\text{clique}| \geq k$

Given $|VC| \leq n-k$

$-|VC| \geq -n+k$

$n - |VC| \geq k$

$|\text{clique}| \geq k$

Hence proved.

d) If G' does not have a ~~clique~~ $|VC| \leq n-k$ then G cannot have $|\text{clique}| \geq k$

Given $|VC| > n-k$

$-|VC| < k-n$

$n - |VC| < k$

$|\text{clique}| < k$

Hence proved VC is NPC.

Applications:

1) Routing.

Question To prove that Independent set (IS) is NP-complete.

Problem statement: (Independent set)

An independent set of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that there is no edge between any pair of vertices in V'.

Vertex Cover: A vertex cover cover vc of an undirected graph $G(V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge in G, then either $u \in V'$ or $v \in V'$.

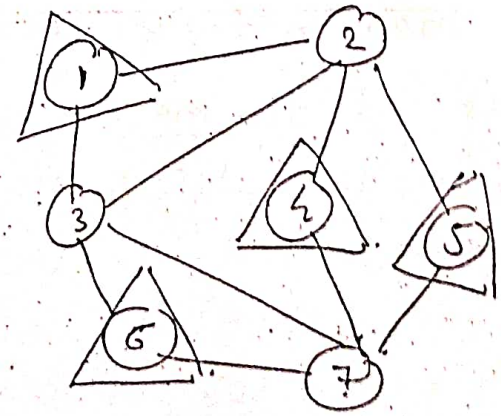
decision version: (Independent set).

Given a graph G and a number k, does G contain an IS of size atleast k.

Optimization Version: (Independent set).

Finding the maximum value of k is the optimization.

e.g. Independent set



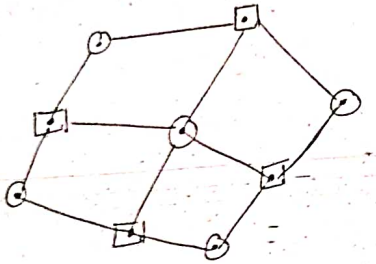
IS = {1, 4, 5, 6}

To prove that IS is NPC, we take a known problem as Vertex Cover (VC)

$VC \leq_p IS$
(Known) (Unknown)

Step 2

Consider the following graph



$G = (V, E)$

$|V| = 9$

VC → ⊙ $|VC| = 5$

IS → ⊡ $|IS| = 4$.

Thus $|IS| = n - |VC|$

Note: [Every graph has a vertex cover and independent set.]

→ VC is minimization problem

IS is maximization problem.

Every edge has at the most one node in VC. Hence node outside VC i.e $|n-VC|$ have all neighbours only in VC.

To prove IS is NPC, we must prove

$VC \leq_p IS$
(Known) (Unknown)

a) If there exist a solution for VC then there exist a solution for IS.

That is to prove that,

If G has $|VC| \leq k$ then $|IS| \geq n-k$

→ Consider L.H.S

That is there exist a soln for VC., thus implies.

$|VC| \leq k$

$-|VC| \geq -k$

$n - |VC| \geq n-k$

$\therefore IS \geq n-k$

Hence, if G has a $|VC| \leq k$ then ~~then~~ there exist an $|IS| \geq n-k$ proved.

b) If G does not have a $|VC| \leq k$ then G does not have $|IS| \geq n-k$

That is if $|VC| > k$ then $|IS| < n-k$

Given $|VC| > k$

$-|VC| < -k$

$n - |VC| < n-k$

$|IS| < n-k$

Hence "If G does not have a $|VC| \leq k$ then G does ~~no~~ cannot have $|IS| \geq n-k$" proved.

c) "If G has an $|IS| \geq n-k$ then G has a $|VC| \leq k$"

Given that $|IS| \geq n-k$

$-|IS| \leq k-n$

$n-|IS| \leq k$

$|VC| \leq k$

Hence, "If G has an $|IS| \geq n-k$ then G has a $|VC| \leq k$", proved.

d) If G does not have $|IS| \geq n-k$ then G cannot have $|VC| \leq k$".

That is to prove that

If $|IS| < n-k$ then $|VC| > k$.

∴ Given $|IS| < n-k$

$-|IS| > k-n$

$n-|IS| > k$

$|VC| > k$

Hence, "If G does not have $|IS| \geq n-k$ then G cannot have $|VC| \leq k$", proved.

Hence IS is NPC.

Step 1. We need to prove that

$IS \in NP$

⇒ NP algorithm for IS is as follows.

Independent set ( )

1. Initialize $S \leftarrow \emptyset$
2. consider ~~Initialize~~ $G' = G$
3. Apply min operation on degree of nodes of G', $\forall n$ where n is the no. of nodes. Let m_i is the node with minimum degree.
4. Add m_i to Independent set S.

   $S \leftarrow S \cup m_i$.
5. $G' \leftarrow G' - \{neighbors(m_i)\}$
5. $n \leftarrow n - \{|neighbors(m_i)| + 1\}$
6. Goto step 2 unles $n \neq 0$ (i.e until all nodes are copied).
7. If $|S| < k$ then apply back-tracking by chosing another node.
8. Go to step 2

   If $|s| \geq k$ then return S.

Complexity : $O(2^n)$

Thus $IS \in NP$.

- If n is no. of digits, add, sub algo runs in $\Theta(n)$ time

- But std algo for multi runs in $O(n^2)$ time, costly

- D & Cong

n digits (8)

$\frac{n}{2}$ $\frac{n}{2}$ 4 4

$A = 4213$

$B = 5128$

let $m = n/2$ (n)

$A[n-1 \ldots 0]$

let $w = A[n-1 \ldots m]$ $x = A[m-1 \ldots 0]$

$y = B[n-1 \ldots m]$ $z = B[m-1 \ldots 0]$

w x A

y z B

wy xy [wz xz]

wz xz [wy xy]

wy wz+xy xz

If w, x, y, z are n/2 dig

$A = w.10^m + x$

$B = y.10^m + z$

$\frac{42}{}\frac{13}{}$

$A = 42 \cdot 10^m$ 4200 + 13

2 produc

$A.B = (w.y)10^{2m} + (w.z + x.y)10^m + x.z$

A = 1234, B = 1234

w = 12, x = 34

y = 12, z = 34

A.B

$u = (a+b)(c+d)$

~~v~~ $= a \times c$ (V)

~~w~~ $= b \times d$ (W)

~~x ... 10^{2m} +~~

$X = V \cdot 10^{2n} + (u - v - w) \times 10^{n/2} + w$

n = 1

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

Question To prove that 3-coloring is NPC.

NP-completeness:

A language L is NP-complete if

1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$

where L' is a known NPC problem.

L is an unknown NPC problem.

Problem statement:

A vertex 3-coloring of a graph $G = (V, E)$ is a function $C: V \to \{1, 2, 3\}$ such that any two adjacent nodes in a graph should not have same color.

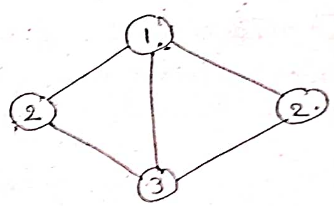
Decision Version:

Can nodes of G be coloured using exactly three colours.

Optimization Version:

The minimum value of by which nodes of graph G are colored.

eg



1 - Red
2 - green
3 - blue

Desig. freq mobile radio
Assign wavelength for wL Devices

Step 1: We need to prove that 3-coloring $\in$ NP

(algorithm and proof is on the last page)

To prove that 3-coloring is NPC we use known NPC problem 3-SAT

step 2: Let $G=(V,E)$ be an instance of 3-coloring.

Consider an instance of 3-SAT as follows:

$(a_i + a_j + a_k)$ where $a_i \Rightarrow x_i \mid \overline{x_i}$

$a_j \Rightarrow x_j \mid \overline{x_j}$

$a_k \Rightarrow x_k \mid \overline{x_k}$

Map an instance of 3-SAT into 3-coloring

The constraints for 3-colorability is

1) Every node is colored.

i.e $\forall v_i \in V \; (x_{iR} + x_{iG} + x_{iB})$.

that is node v_i has color Red or Green or blue

$$C_1 = \prod_{i=1}^{|V|} (x_{iR} + x_{iG} + x_{iB})$$

(Clauses)

2) If Every node should have exactly one color.

$$C_2 = \prod_{i=1}^{|V|} (\overline{x_{iR}}\,\overline{x_{iG}} + \overline{x_{iR}}\,\overline{x_{iB}} + \overline{x_{iG}}\,\overline{x_{iB}})$$

(node i can't have two colours)

3) If $e(i,j)$ exists then i and j cannot have the same colour.

$$C_3 = \prod_{i=1}^{|E|} (\overline{x_{iR}}\,\overline{x_{jR}} + \overline{x_{iG}}\,\overline{x_{jG}} + \overline{x_{iB}}\,\overline{x_{jB}})$$

(i.e node i and j of the same edge can't have same colour.)

step 3

a) If 3-SAT has a solution, then vertex-3 colorability has a solution.

If 3-SAT has a solution, implies that clauses C_1, C_2 and C_3 are true.

C_1 = True $\Rightarrow$ Every node has either Red or Green or Blue colour.

C_2 = True $\Rightarrow$ No node has (or more) two colours. i.e every node has exactly one colour

C_3 = True $\Rightarrow$ No neighboring nodes have same colours.

$C_1 \cdot C_2 \cdot C_3$ = True $\Rightarrow$ graph G is 3-colorable.

step 2

b) If 3-SAT does not have solution, then vertex 3-colorability do cannot have solution.

If 3-SAT does not have solution implies that $C_1 \cdot C_2 \cdot C_3$ = false.

This implies that either C_1 or C_2 or C_3 is false.

C_1 = False $\Rightarrow$ Then exist a node which is not coloured.

C_2 = False $\Rightarrow$ Then exist a node with two/more colours

C_3 = false $\Rightarrow$ Both nodes of an edge have same colour.

Thus if $C_1 \cdot C_2 \cdot C_3$ = false implies Graph G is not 3-colorable.

c) If graph G is 3-colorable then.
3-SAT has solution.
Graph is 3-colorable implies that
1) Every node has atleast one colour.
⟹ c_1 is true
2) No node has two (or more) colours
⟹ c_2 is true
3) Every pair of neighbors have distinct colours.
⟹ c_3 is true.
Since $c_1 c_2 c_3$ = True, 3-SAT has a solution

d) If graph is not 3-colorable, then.
3-SAT cannot have a solution.
If graph is not 3-colorable then,
1. There exist atleast one node that does not have colour.
∴ c_1 is false
2. At least one node has two or more colours.
∴ c_2 is false
3. At least one pair of neighbor has same colour.
∴ c_3 is false.
Even if one of constraints is false, 3-SAT cannot have a solution.

Hence. 3-coloring is Npc problem.

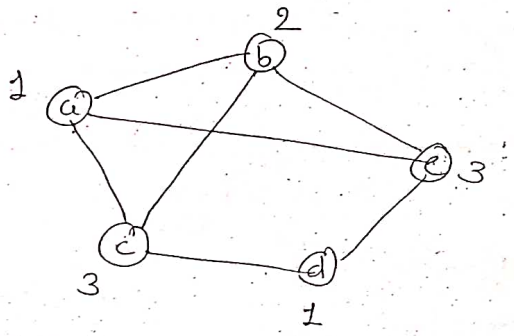
Graph coloring: $O(nm^n)$

→ Let $G=(V,E)$ be a connected graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same colour yet only m colours are used. This is termed as the m-colorability decision problem.

→ The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored, such a way that no two adjacent nodes can have same colours. This integer is referred to as the chromatic number of the graph.

→ Note that if d is the degree of a given graph, then it can be colored with d+1 colors.

eg



Applications: 1) Time table scheduling
2) Assigning frequences to mobile radios.
3) Wavelength assignment for wireless comm devices

(step 1)

Algorithm: mcoloring

(1) We represent a graph by its adjacency matrix G[1:n, 1:n] where G[i,j] = 1 if there is an edge from i to j and G[i,j] = 0 otherwise.

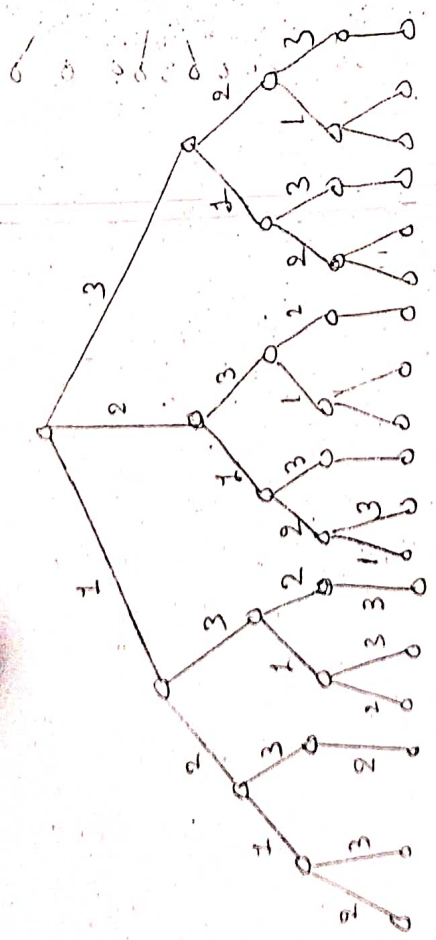
(2) The colours are represented by the integers 1, 2, ..., m

(3) Solutions are given by n-tuple (x1, x2, ... xn) where xi is the colour of node i.

```
Algorithm mcoloring(k) // k is the index
                       // of the next vertex
                       // to color
{
    repeat
    { // Generate all legal assignments
      // for x[k].
      NextValue(k); // Assign to x[k] a
                    // legal color.
      if (x[k] = 0) then return; // no new
                                 // color possible
      if (k = n) then // At most m colors
                      // have been used to
                      // color the n vertices.
      write (x[1..n]);
      Else
          mcoloring(k+1);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1] ... x[k-1] have been assigned integer
// values in the range [1 ... m] such that
// adjacent vertices have distinct integers.
// A value for x[k] is determined in the
// range [0, m]. x[k] is assigned a next
// highest numbered color while maintaining
// distinctness from the adjacent vertices of
// vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] = (x[k] + 1) mod (m+1); // Next
                                     // highest color
        if (x[k] = 0) then return; // All colors
                                   // have been used
        for j = 1 to n do
        { // Check if this color is distinct
          // from adjacent colors.
          if ((G[k,j] ≠ 0 and x[k] = x[j])
              then break;
        }
        if (j = n+1) then return; // New color
                                  // found
    } until (false);
}
```

State space tree for m-coloring. (3-coloring).



Time Complexity of m-coloring = $O(n m^n)$.

Hence it NP

$$\sum_{i=0}^{n-1} m^i = m^0 + m^1 + m^2 + \dots + m^{n-1} = \frac{(m^{n+1} - 2)}{(m+1)} = O(m^n)$$

Question: To prove Vertex cover is NPC.

NP-completeness:

A language L is NP-complete iff

1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$

where L' is a known NPC problem
L is an unknown problem.

Problem statement:

definition (VC): A vertex cover VC of an undirected graph $G(V, E)$ is a subset $S \subseteq V$ such that if (u,v) is an edge in G, then either $u \in S$ or $v \in S$

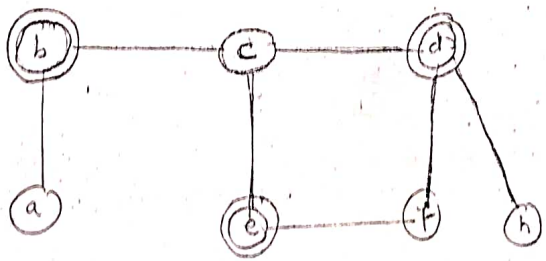
Decision Version: (VC)

Given a graph G whether or not G has a vertex cover of size $\leq k$ is the decision version of vertex cover problem.

optimization version: (VC)

Finding the minimum value of k is the optimization problem.

e.g.



$VC = \{b, d, e\}$

Step 1 We need to prove that

$VC \in NP$

NP algorithm for VC.

1. Consider graph $G = G'$
2. Apply maximum operation on degree of nodes of n, when n is the number of nodes.
   i.e $n_1 = max\ (\forall_n\ deg(n))$
3. $S \leftarrow S \cup n_1$
4. $G' \leftarrow G' -$ edges (n_1)
5. $n \leftarrow n-1$
6. Goto step 2 unless $n \neq \phi$ as all nodes are covered.
7. If $|s| > k$ then
   apply back tracking and start again by choosing another node.
   Go to step 2.
   else if $|s| = k$ or $|s| < k$
   Go to step 8.
8. Stop & return S.

Complexity of VC.

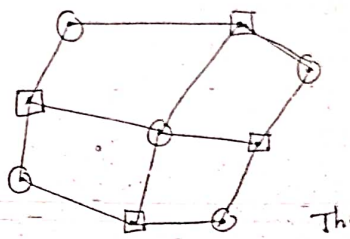
The first choice of VC can be in n_{C_r} i.e from n nodes we can get r nodes. i.e by applying greedy approach. But if $|s| > k$, then reverting back to previous choice & applying back tracking we get a power set of nodes i.e

complexity of $V.C = O(2^n)$ Hence $VC \in NP$.

Step 2: To prove that VC is NPC, we take known NPC problem as IS.

$$IS \leq_p VC$$
(known) (Unknown).

Consider the following graph.



$G = (V, E)$

$|V| = 9$

VC → ⊙ $|vc| = 5$

IS → □ $|IS| = 4$

Thus $|vc| = n - |IS|$

Note Every graph has a vertex cover and independent set] VC is minimization prob IS is maximization prob.

- A vertex cover VC of an undirected graph $G(V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge in G, then either $u \in V'$ or $v \in V'$.

a) If there exist a solution for IS then there exist a solution for VC.

That is to prove that,

If there exist an independent set $|IS| \geq k$ then there exist a vertex cover $|vc| \leq n-k$

Consider the L.H.S.

That is if there exist an independent set $|IS| \geq k$

$$|IS| \geq k$$
$$\therefore -|IS| \leq -k$$
$$n - |IS| \leq n - k$$

$\therefore |vc| \leq n-k$

Hence, if G has $|IS| \geq k$, then G has a $|vc| \leq n-k$ proved.

b) If G does not have an independent set $|IS| \geq k$ then G cannot have a vertex covers $|vc| \leq n-k$.

That is if $|IS| < k$ then $|vc| > n-k$

→ Consider L.H.S.

$$|IS| < k$$
$$\therefore -|IS| < -k$$
$$n-|IS| < n-k$$
$$\therefore |vc| < n-k$$

Hence, If G does not have an independent set $|IS| \geq k$ then G cannot have a vertex cover $|vc| \leq n-k$ proved.

c) If G has a vertex cover $|vc| \leq n-k$ then, there exist an Independent set $|IS| \geq k$

Consider the L.H.S.

If G has a $|vc| \leq n-k$

$$\therefore |vc| \leq n-k$$
$$-|vc| \geq k-n$$
$$\therefore n-|vc| \geq k$$
$$|IS| \geq k$$

Hence, if G has a $|vc| \leq n-k$ then there exist an $|IS| \geq k$ proved.

d) If G does not have a $|vc| \leq n-k$ then G cannot have an independent set $|IS| \geq k$

That is to prove that,

if $|vc| > n-k$ then $|IS| < k$

Consider the L.H.S;

$$|vc| > n-k$$
$$\therefore -|vc| < k-n$$
$$n-|vc| < k$$
$$|IS| < k$$

Thus, if G does not have a vertex cover $|vc| \leq n-k$, then G cannot have an independent set $|IS| \geq k$.

Hence, Vc is NPC

Applications of VC

1. Stating Routing.
2. placing post offices to villages.

Goal of vertex cover is to minimize the no. of vertices that cover the whole area.

Question: To prove edge 3-coloring is NPC.

3-SAT $\leq_p$ Edge 3-coloring.

NP-completeness:

A lang. L is NP-complete if

1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$

where L' is a known NPC problem.

Problem statement:

definition (Edge 3 coloring):

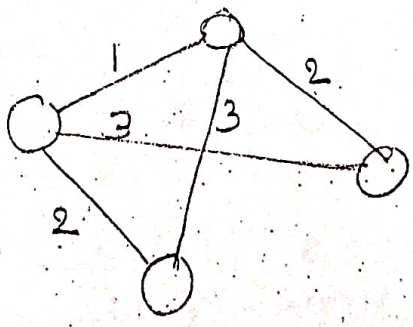
An edge 3-coloring of a graph $G=(V,E)$ is a function $f: E \to \{1,2,3\}$ such that $C(e(i,j))$ is not same as color of any of its adjacent edge.

Decision Version:

Can edges of G be colored using exactly 3 colors?

Optimization Version:

To find the minimum no. of colors by which a graph can be colored is an optimization version.
↓
an edges of



Step 1: We need to prove.
edge 3-coloring ∈ NP.

NP algorithm:

For G(V,E)

if (degree of graph ≥ 3)

"The edges are not 3-colorable"

return.

Else

Repeate the following process for every edge.

select any edge (V_i, V_j)

Select color for (V_i, V_j) that is not assigned to any of its adjacent edge.

If no color available

"The edges are not colorable"

return;

Else.

color edge (V_i, V_j) with that color.

If all edges are colored.

"Edges are 3-colorable"

return.

Time Complexity

$O(3^e)$ brute force

Hence

edge 3-coloring ∈ NP.

Step 2: To prove edge-3-coloring is NPC

We use 3-SAT as known problem.

Mapping.

Let G = (V,E) be an instance of edge-3-coloring.

Consider an instance of 3-SAT as follows.

$(a_i + a_j + a_k)$ when $a_i \Rightarrow x_i \mid \overline{x_i}$

$a_j \Rightarrow x_j \mid \overline{x_j}$

$a_k \Rightarrow x_k \mid \overline{x_k}$

Map an instance of 3-SAT into edge 3-coloring

The constraint for edge 3-colorability is

1) Every edge has to be colored.

$$c_1 = \prod_{i=1}^{|E|} \left(e_{i,j}R + e_{i,j}B + e_{i,j}G \right)$$

2) Every edge should have exactly one color

$$c_2 = \prod_{i=1}^{|E|} \left(\overline{e_{i,j}R}\ \overline{e_{i,j}G} + \overline{e_{i,j}R}\ \overline{e_{i,j}B} + \overline{e_{i,j}G}\ \overline{e_{i,j}B} \right)$$

3) If edge $e(V_i, V_j)$ exists then no adjacent edge $e(V_i, V_k)$ or $e(V_j, V_m)$ can have same colors that of $e(V_i, V_j)$.

$$C_3 = \prod_{i=1}^{|E'|} \left(\overline{e_{i,j,R}}\ \overline{e_{i,k,R}} + \overline{e_{i,j,B}}\ \overline{e_{i,k,B}} + \overline{e_{i,j,G}}\ \overline{e_{i,k,G}} \right)$$

Step 3

a) If 3-SAT has a solution, then edge 3-coloring has a solution.

If 3-SAT has a solution, this implies

$C_1 \cdot C_2 \cdot C_3 = T$

$C_1 = T$ => Every edge has either a Green or Blue or Red color.

$C_2 = T$ => Every edge has exactly one color.

$C_3 = T$ => No adjacent edges have the same color.

Therefore. $C_1 \cdot C_2 \cdot C_3 = T$ implies that graph G is edge 3-colorable.

b) If 3-SAT doesnot have solution, then edge 3-coloring cannot have solution.

If 3-SAT doesnot have solution. implies that

$C_1 \cdot C_2 \cdot C_3 = F$

This implies that at least one of three clauses (C_1, C_2 or C_3) is false.

$C_1 = F$ => there exist an edge which is not colored

$C_2 = F$ => there exist an edge which has ~~more~~ too/more ~~than~~ colors.

$C_3 = F$ => Two adjacent edges have same color.

Hence, If 3-SAT does not have solution, Graph G is not edge 3-colorable.

c) If edges of graph are 3-colorable, then 3-SAT has solution.

If edges of graph are colorable then

1) Every ~~node~~ edge is colored so $C_1 = T$

2) Every edge has exactly one color so $C_2 = T$

3) Adjacent edges have different colors so $C_3 = T$

Since all constraints are satisfied, 3-SAT has a solution.

d) If edges of graph are not 3-colorable then 3-SAT cannot have a solution.

If edges of graph are not 3-colorable this implies that

1) Every edge is not colored hence $C_1 = F$.

2) There exist an edge that has more than one color hence $C_2 = F$

3) Adjacent edges have same color hence $C_3 = F$

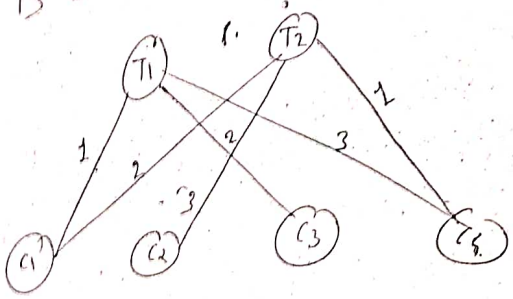
Even if one of the constraints is false, 3-SAT cannot have solution.

Hence,

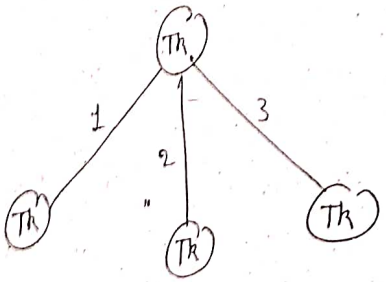
edge-coloring is NPC.

Applications

1) class Room Scheduling



2) Wireless Radio Networks



color of the edges corresponds to assignment of different frequencies.

Ford-Fulkerson Algorithm:

The Ford-Fulkerson method is iterative. We start with $f(u,v) = 0$ for all $u, v \in V$, giving an initial flow of value 0.

→ At each iteration we increase the flow value by finding an "augmented path", which we can think of simply as a path from the source s to the sink t along which we can push more flow, and then augmenting the flow along this path.

→ We repeat this process until no augmenting path can be found.

```
FORD-FULKERSON-METHOD (G, s, t)
1. initialize flow f to 0
2. while there exist an augmenting path p
3.     do augment flow f along p.
4. return f

FORD-FULKERSON (G, s, t)
1.  for each edge (u,v) ∈ E
2.  do f[u,v] ← 0
3.     f[v,u] ← 0
4.  while there exist a path p from s to t in
    the residual network Gf
5.  do cf(p) ← min { cf(u,v) : (u,v) is in p}
6.  for each edge (u,v) in p
7.     do f[u,v] ← f[u,v] + cf(p)
8.        f[v,u] ← − f[u,v]
```

Application:

Some real-life problems, like those involving the flow of liquids through pipes, current through wires, and delivery of goods, can be modeled using flow n/w.

Flow Networks

A flow network is a directed graph $G=(V,E)$ such that,

1. For edge $(u,v) \in E$, we associate a positive (non negative) capacity $c(u,v) \geq 0$. If $(u,v) \notin E$, then $c(u,v)=0$
2. There are two distinguished points, the source s, and the sink t.
3. For every vertex $v \in V$, there is a path from s to t containing v.

(1) MST- Kruskal (G, w)

1. $A \leftarrow \emptyset$ ——> O(1)
2. For each $v \in V[G]$
   do make-set[v]
3. Sort the edges in E ~~else~~ in increasing order of their weight —— O(E lg E)
4. For each edge (u,v) in E, taken in increasing order of weight
   do if findset(u) ≠ findset(v)
   $A \leftarrow A \cup \{(u,v)\}$
   UNION(u,v)
   O((V+E) x(V))
5. Return A.

Complexity O(E lg V)

(2) MST-PRIMS (G, w, r).

1. For each $v \in V[G]$ ——> O(V)
   $\pi[v] \leftarrow$ NIL
   key[v] $\leftarrow \infty$
   key[r] $\leftarrow 0$.
2. $Q \leftarrow V[G]$
3. while (Q ≠ empty) } V lg V
   $u \leftarrow$ extract-min(Q) — lg V
   for each $v \in adj[u]$ ——> O(E)
   do if $v \in Q$ and $w(u,v) <$ key[v]
   then $\pi[v] \leftarrow u$
   key[v] $\leftarrow w(u,v)$

Complexity : O(E lg V)

(3) Bellman Ford (G, s).

1. Initialize-single-source (s) → O(V)
2. for i ← 1 to |v|-1 → O(VE)
   for each edge (u,v) ∈ E
1. do Relax (u,v,w).
3. for each edge (u,v) ∈ E → O(E).
   do if d[v] > d[u] + w(u,v)
   then return false
4. Return True.

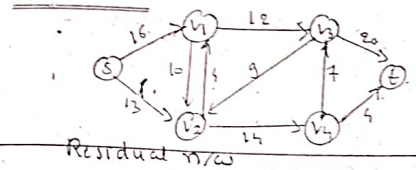
Complexity: O(EV).

(4) Warshall's_APS.(W)

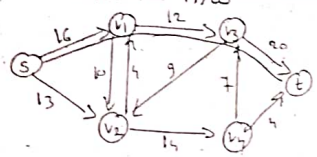
n ← rows[W]
D^(0) ← W
for k ← 1 to n
  for i ← 1 to n
    for j ← 1 to n
      d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)).
Return D^(n).

Complexity: O(n^3)

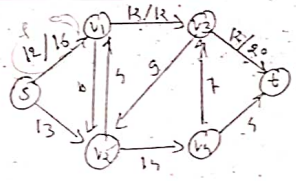
Max-flow



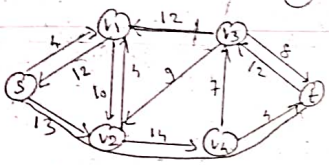
Residual n/w



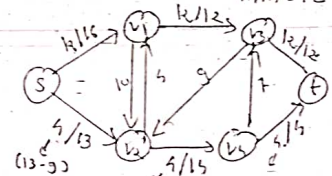
Net Flow



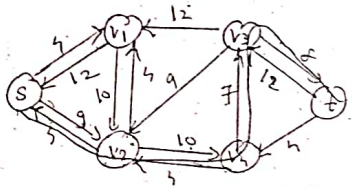
(a)



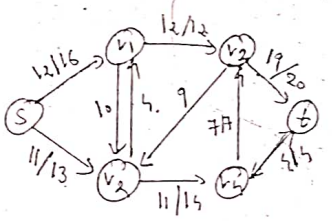
min = 12



(b)

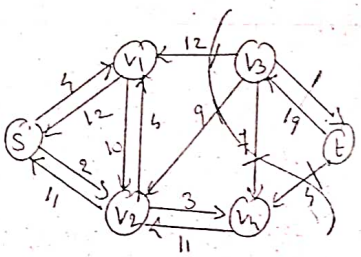


min = 4.



(c)

min = 7



cut =

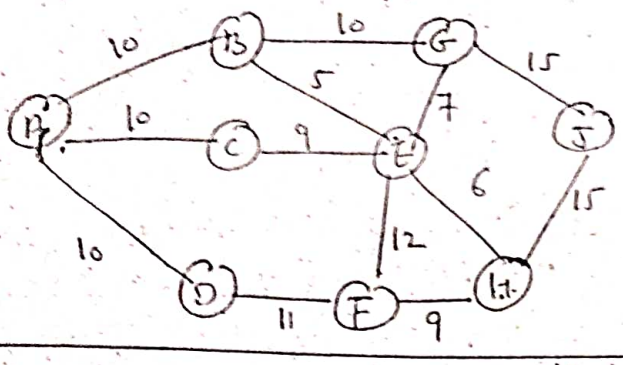
It has no augmenting path so max flow = 12+4+7
= 23.

Max-Flow.

BB-11325

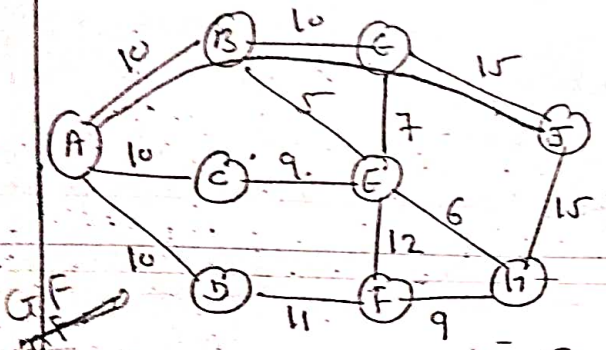
source = A

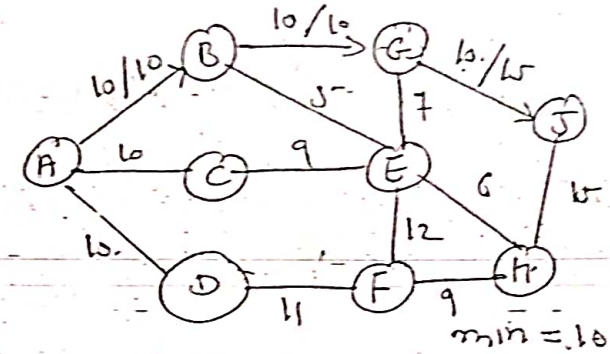
Sink = J



Residu nw & augmented path

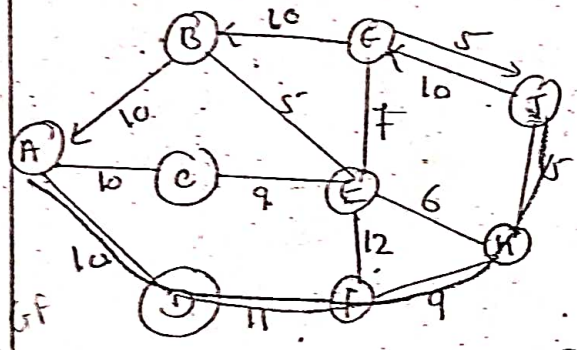
net Flow..

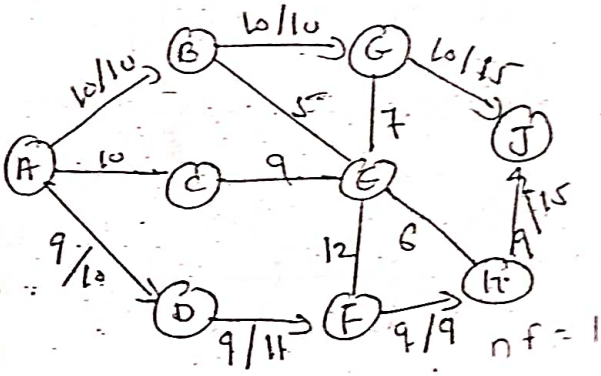




min = 10

(a)

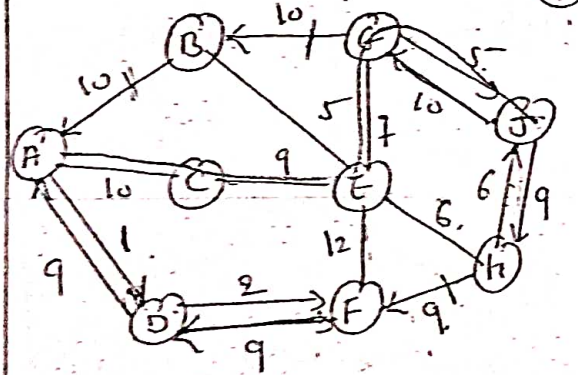


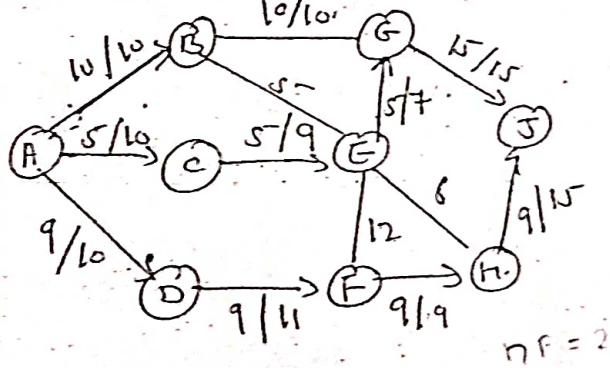


nf = 19

min = 9.

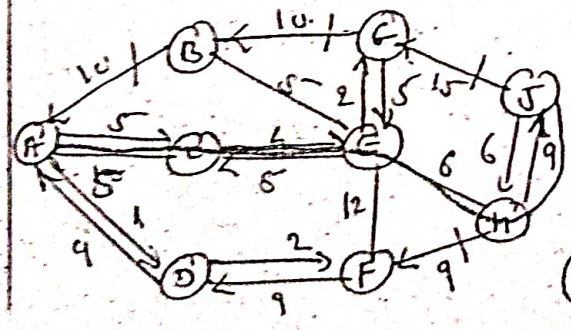
(b)

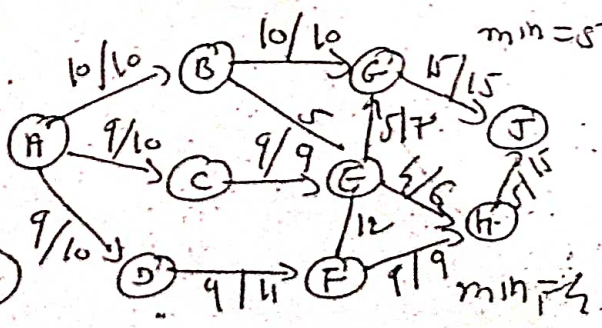




nf = 24

min = 5

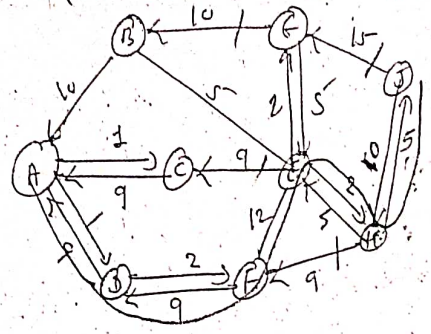


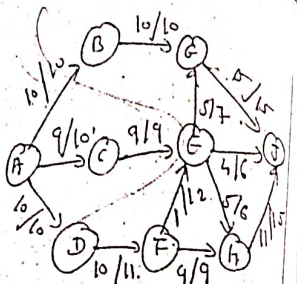


(c)

min = 4

nf = 28





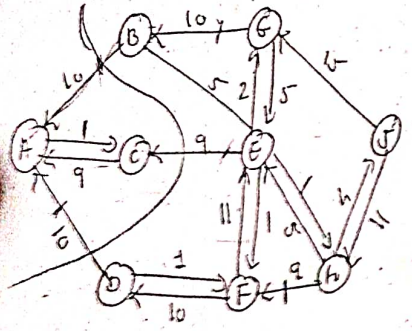
min = 1

No augmented path possible ∴

max flow = 10 + 9 + 5 + 4

= 28

max cut



No augmented path possible

∴ max flow = 10 + 9 + 5 + 4 + 1

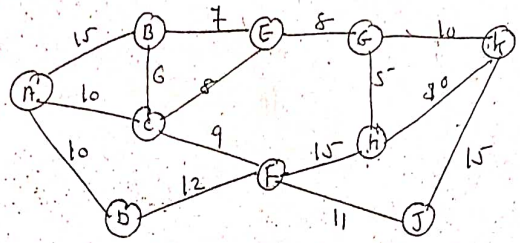
= 29

S = {A, C, E} T = {B, D, F, G, H, J}

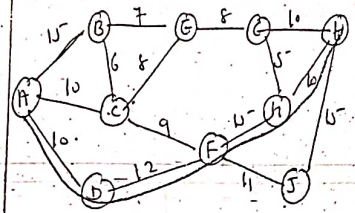
Max-Flow BB-9950

source - A

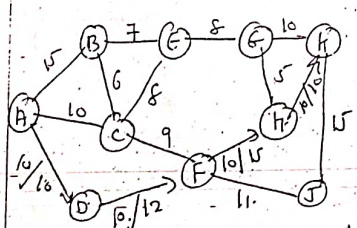
sink - K



Residual network

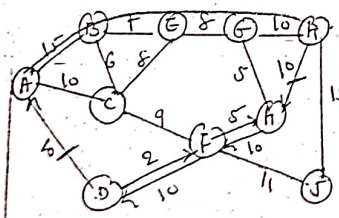


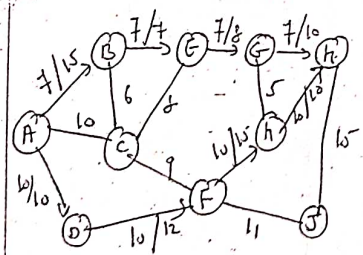
Net Flow



min = 10

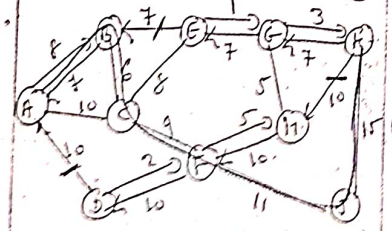
(a)

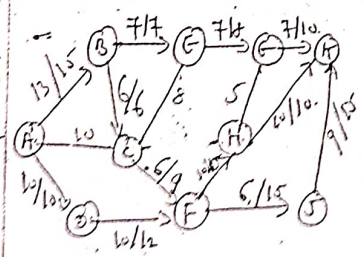


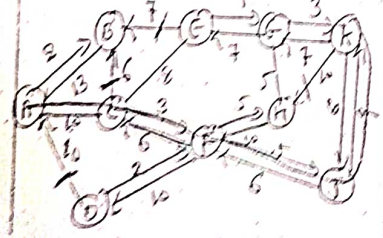


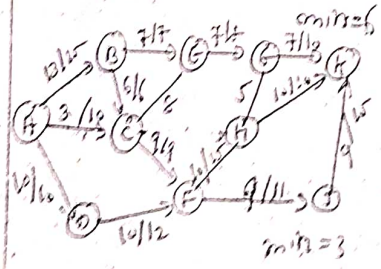
min = 7

(b)

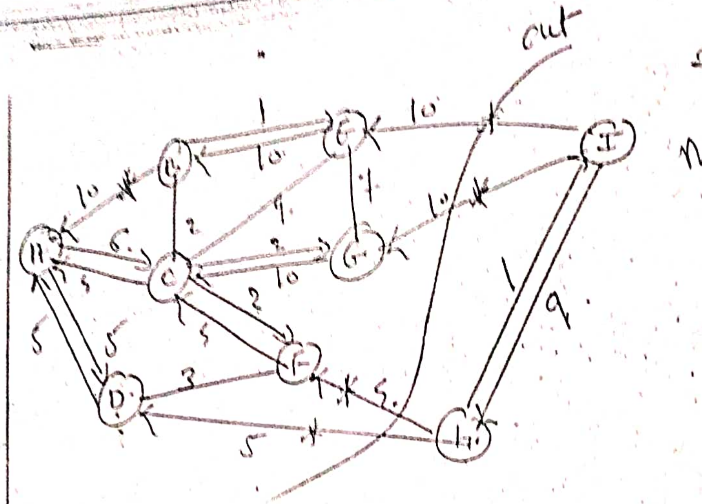








min = 3

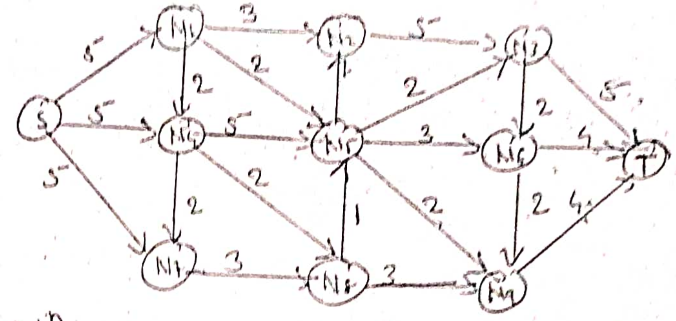


min=3

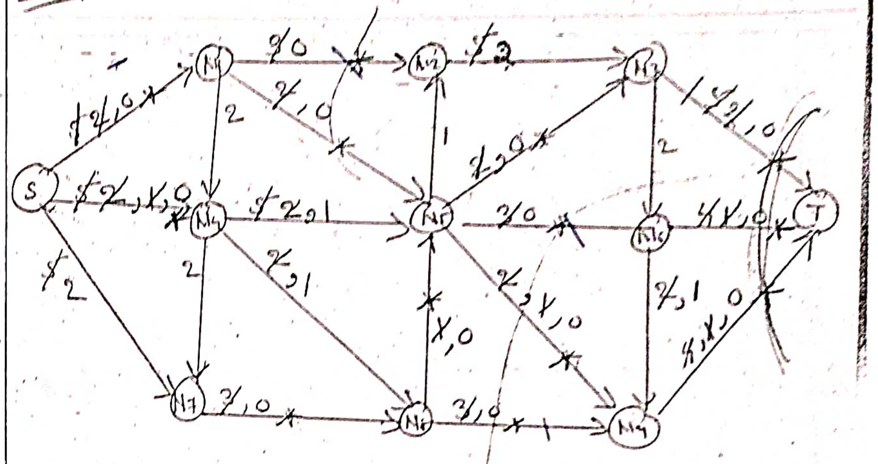
max flow = 10+10+5+4 = 28

BB-7502

Evaluate the maximum flow from S to T.

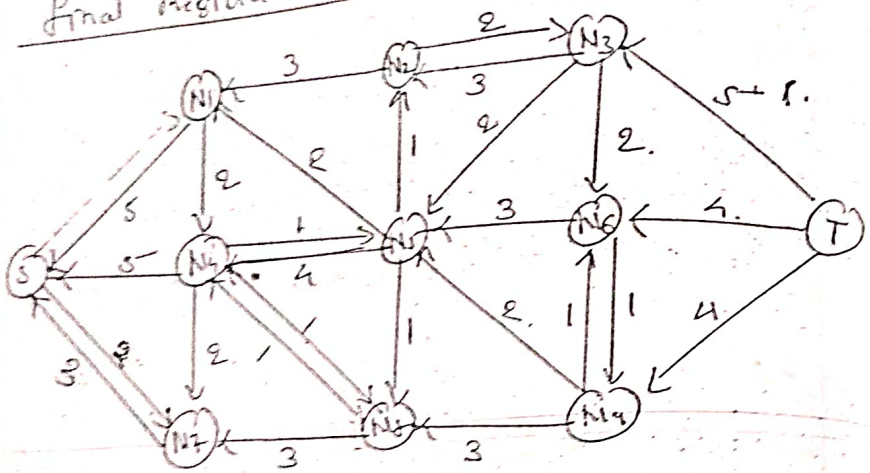


Sol^n

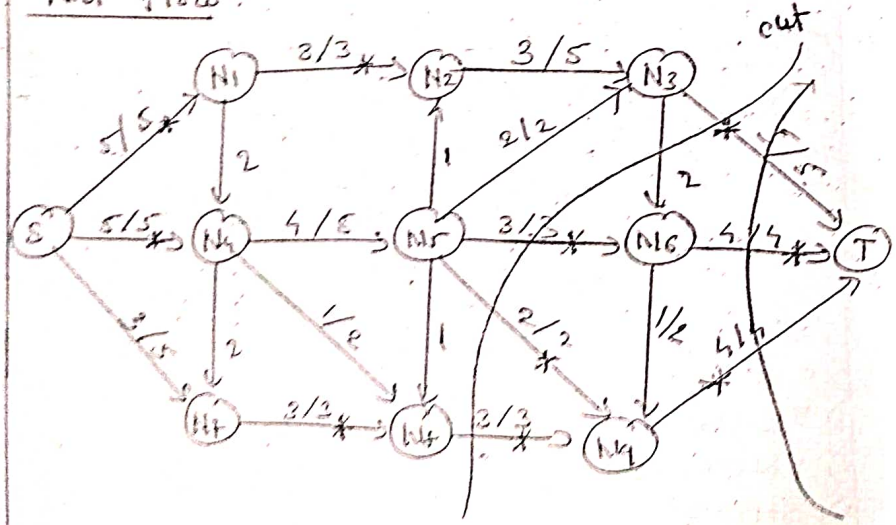


Path	Max flow
1) $SN_1N_2N_3T$	3
2) $SN_4N_5N_6T$	3
3) $SN_7N_8N_9T$	3
4) $SN_1N_5N_3T$	2
5) $SN_4N_8N_5N_9T$	1
6) $SN_4N_5N_9N_6T$	1
Max flow	12

Final Residual Network



Net Flow:

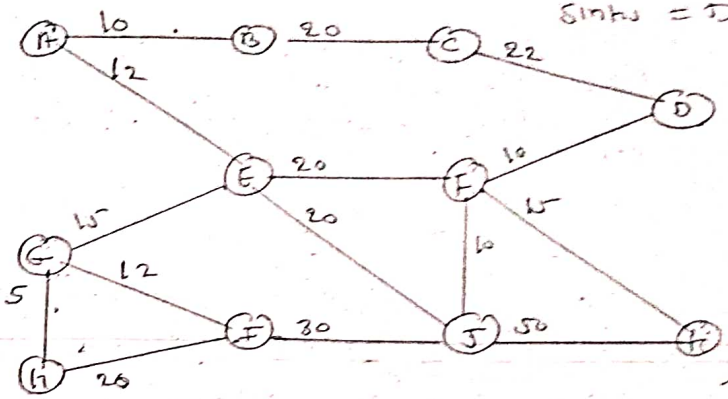


Max Flow

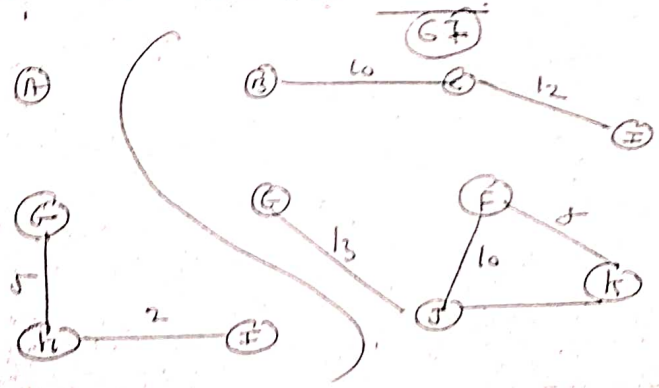
Multiple Sources Multiple sinks.

sources - A, G, H

sinks = D, K



Source	sink	path	capacity
A	D	ABCD	10
A	D	AEFD	10
A	K	AGFK	02
A	K	GGFK	05
G	K	GGJK	07
G	K	GIJK	12
H	K	HIJK	15



Question: Show how Quick sort can be made to run in $O(n \log n)$ time in the worst case.

Ans

- Quick sort is based on the divide-and-conquer method.
- Three steps are followed for sorting a subarray $A[p \ldots r]$.

1) **Divide**: Partition the array $A[p..r]$ into two subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

2) **Conquer**: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

3) **Combine**: Since subarrays are sorted in place, we don't need to combine them: the entire array $A[p:r]$ is sorted now.

The following procedure implements quicksort.

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

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

⇒ Partioning the array.

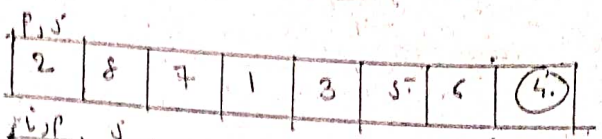
```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p-1
3.  for j ← p to r-1
4.      do if A[j] ≤ x
5.          then i ← i+1
6.               exchange A[i] ↔ A[j]
    end for.
7.  exchange A[i+1] ↔ A[r]
8.  return i+1
```

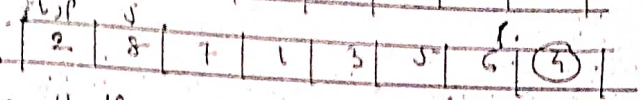
Explaination.

→ PARTITION always selects an element $x = A[r]$ as a pivot element around which to partition the subarray $A[p..r]$.

⇒ The following example illustrate whole partitioning procedure step by step.

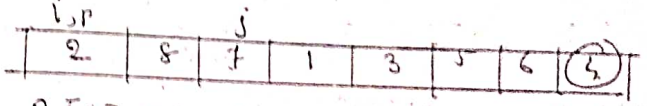
Example

a) 

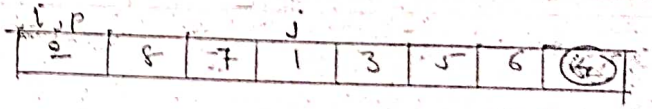
b) 

In the first step 2 is exchange with 2.

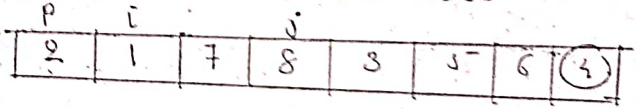
c) Now $A[j] = 8 \not< 4$ so don't exchange simply increment j.

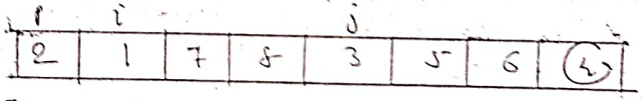


d) $A[j] = 7 \not< 4$, so increment also j

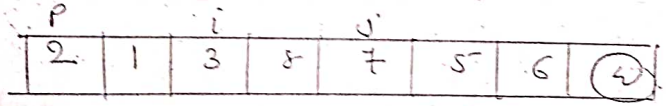


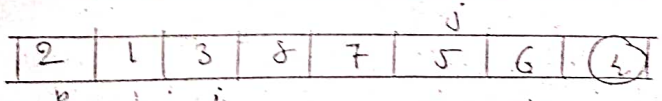
e) Here $A[j] = 1 < 4$, so exchange $A[i+1] = A[i]$ and $A[j]$

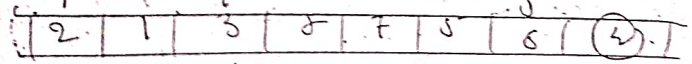
∴ 

f) 

$A[j] = 3 < 4$, so $i \leftarrow i+1$ and exchange $A[i+1] = A[2]$ with $A[j]$

g) 

h) 

i) 

finally exchange $A[i+1]$ and $A[r]$

j) 2 1 3 4 7 5 6 8

Performance of QuickSort:

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

⇒ Worst-Case Partitioning:

- The worst case behavior for quick sort occurs when the partitioning routine produces one subproblem with n-1 elements and one with 0 elements.
- Let us assume that this worst case partitioning arises in each recursive call.
- The partitioning costs $\Theta(n)$ time. Since the recursive call on array of size 0 returns $T(0) = \Theta(1)$, the recurrence for the running time is

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

If sum the cost incured at each level of recursion, we get arithmetic series, which evaluates to $\Theta(n^2)$.

- Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$.
- The $\Theta(n^2)$ running time occurs when the input array is already sorted.

Best case partitioning:

- The best case occurs when, PARTITION produces two subproblems, each of size no more than $n/2$. Since one is of size $\lfloor n/2 \rfloor$ and is of size $\lfloor n/2 - 1 \rfloor$.
- In this case quicksort runs much faster.
- The recurrence for the running time is,

$$T(n) \leq 2T(n/2) + \Theta(n)$$

Using Master Theorem's case 2, we have a solution

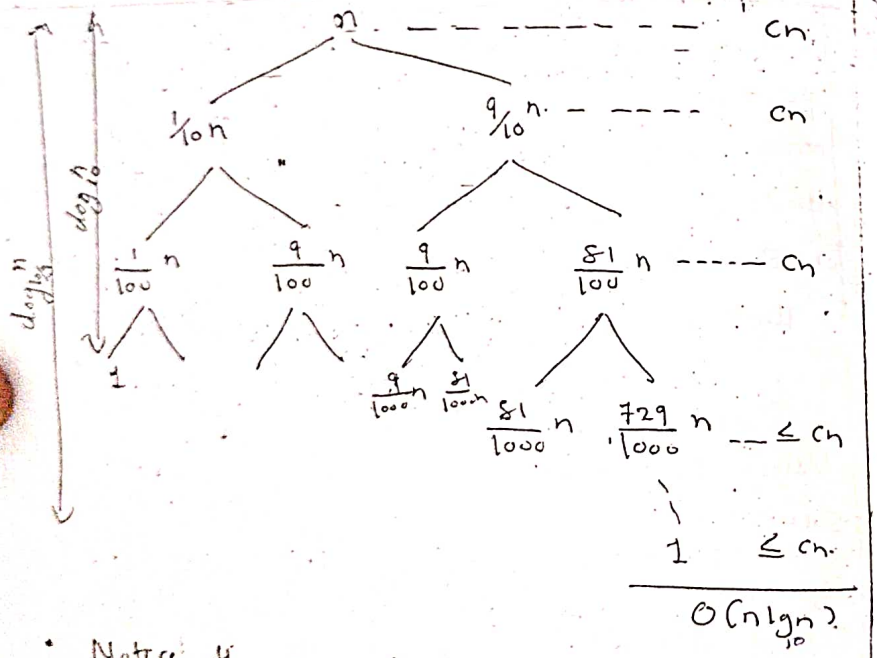
$$T(n) = o(n \lg n).$$

- Thus, the equal balancing of the two sides of the partition at each level of the recursion produces an asymptotically faster algorithm.

Balanced partitioning (Avg case partitioning)

- The average case running time of quicksort is much closer to best case.
- for eg; the partitioning algo. always produces 9-to-1 proportional split, which at first glance ~~quite~~ seems quite unbalanced.
- We obtain the recurrence

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



- Notice the every level of a tree has a cost cn until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$ and then the levels have cost at most cn.
- The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$
- ∴ The total cost of quicksort is $O(n \lg n)$.

A Randomized Version of Quicksort

- Quicksort has an average time of $O(n \lg n)$ on n elements, its worst case time is $O(n^2)$.
- We can use modified version of quicksort which performs well on every input.
- Instead of always using A[r] as the pivot, we will use a randomly chosen element from the subarray A[p..r].
- We do so by exchanging element A[r] with an element chosen at random from A[p..r].
- This modification, in which we randomly sample the range p..r, ensures that the pivot element $x = A[r]$ is equally likely to be any of the r−p+1 elements in subarray.
- Because, the pivot element is chosen randomly, we expect the split of the i/p array to be reasonably well balanced on avg.

RANDOMIZED_PARTITION (A, p, r)

i ← Random (p, r)

exchange (A[r] ↔ A[i]).

return PARTITION (A, p, r).

RANDOMIZED_QUICKSORT (A, p, r)

if p < r

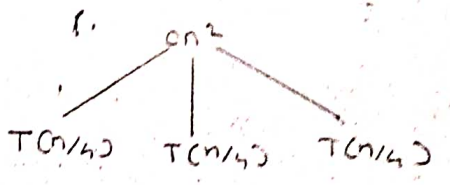
then q ← RANDOMIZED_PARTITION (A, p, r)

RANDOMIZED-QUICKSORT (A, p, q-1)

RANDOMIZED-QUICKSORT (A, q+1, r)

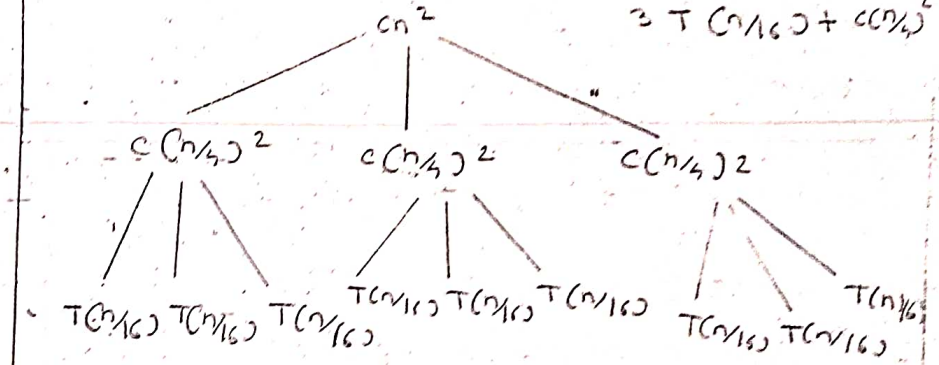
Refer Cormen for complexity.

Example Draw a recursion tree for recurrence $T_n = 3T(n/4) + cn^2$

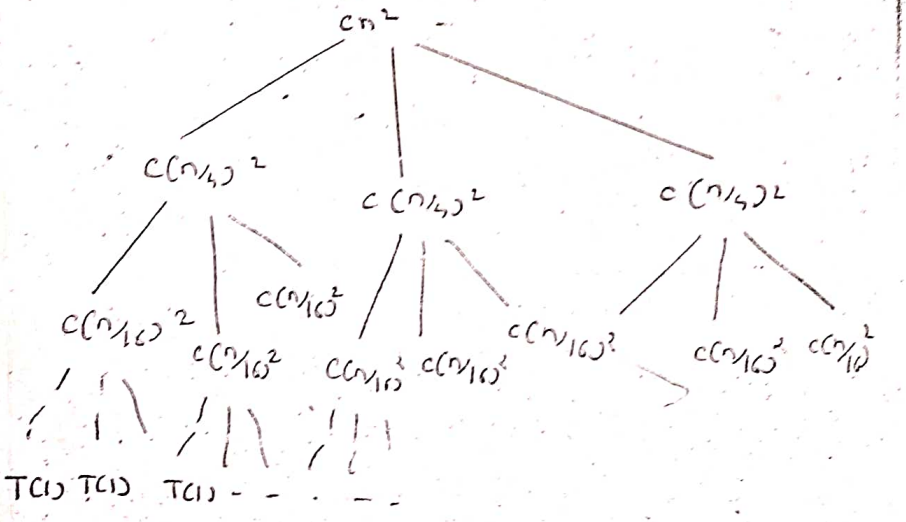


(a)

but $T(n/4) = 3T(n/16) + c(n/4)^2$



(b)



Question No - 1912

Use mathematical induction to show that when n is exact power of 2, the solution of the recurrence

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

is $T(n) = n \log n$ (CO2K)

Soln

Step 1: For the smallest value of $n = 2$

L.H.S = $T(n) = T(2) = 2$

R.H.S = $n \log_2 n = 2 \log_2 2 = 2$.

L.H.S = R.H.S. Hence proved.

Step 2: Assume it true for $n = 2^k$

$$T(2^k) = 2^k \log_2 2^k$$

$$T(2^k) = 2^k . k \quad \text{——— (1)}$$

Step 3:

Prove for $n = 2^{k+1}$.

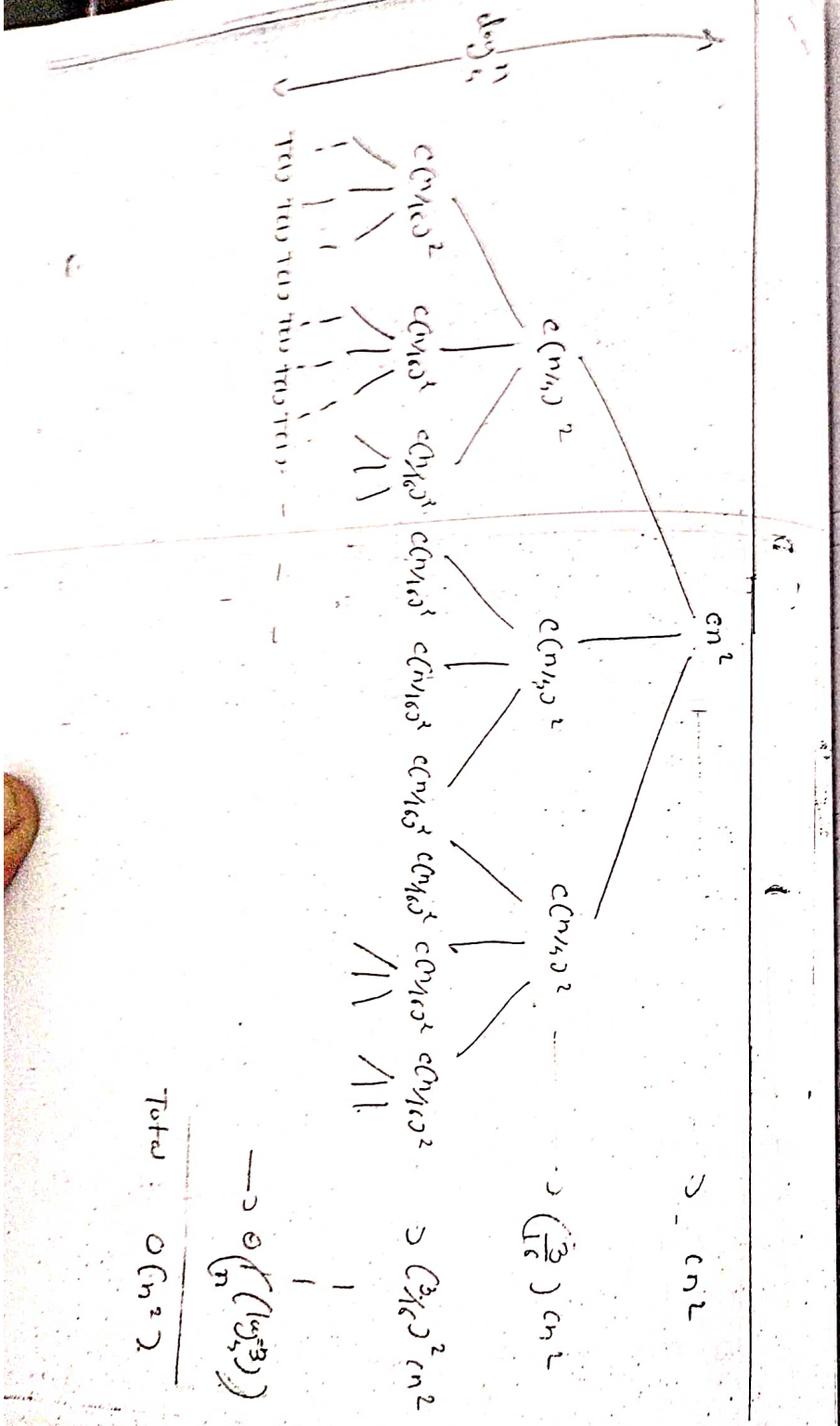
$$T(2^{k+1}) = 2^{k+1} \log_2 2^{k+1}$$

$$= 2^{k+1} . (k+1)$$

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

$$= 2k . 2^k + 2 . 2^k \quad \Rightarrow \text{from (1)}$$

But $n = 2^k$ and $k . 2^k = T(2^k)$



$= 2\,T(2^k) + 2n$

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

$= T(2n)$

$= T(2 \cdot 2^k)$

$= T(2^{k+1})$

$= L.H.S.$

Hence proved.

The same problem can be solved by ~~substitution~~ method.

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

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

$$= 4\,T(n/4) + n + n$$

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

$$= 4\,[2\,T(n/8) + n/4] + 2n$$

$$= 8\,T(n/8) + n + 2n$$

$$= 8\,T(n/8) + 3n.$$

In general, we say that,

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

for any $1 \le k \le \log_2 n$

Now we know that

$$n = 2^k$$

$$\therefore T(n) = n\,T(n/n) + \log_2 n \cdot n.$$

$$= n\,T(1) + n \cdot \log_2 n$$

$$\therefore T(n) = O(n \lg n).$$

Single Source Shortest Path.

Given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

1) Single destination shortest path.

Find a shortest path to a given destination vertex t from each vertex $v \in V$.

2) Single pair shortest path:

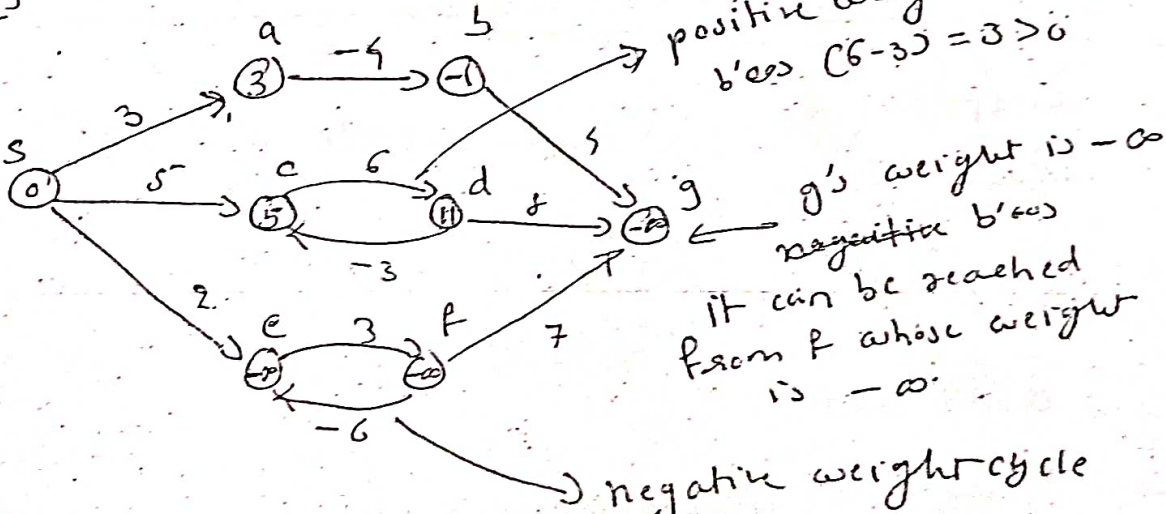
Find a shortest path from u to v.

3) All-pair shortest path:

Find a shortest path from u to v for every pair of vertices u and v.

Negative weight cycle.

e.g



Assign $-\infty$ to negative weight cycle nodes.

Note:

① Dijkstra's algorithm, assume that all edge weights in the graph are non-negative.

② Bellman-ford algorithm, allow negative weights in the graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.

Relaxation: Imp.

Most of the shortest path algorithms use the technique of relaxation.

- For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a shortest path estimate.

```
INITIALIZE-SINGLE-SOURCE (G,s)
for each v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

→ Here $\pi[v]$ represents the predecessor of vertex v.

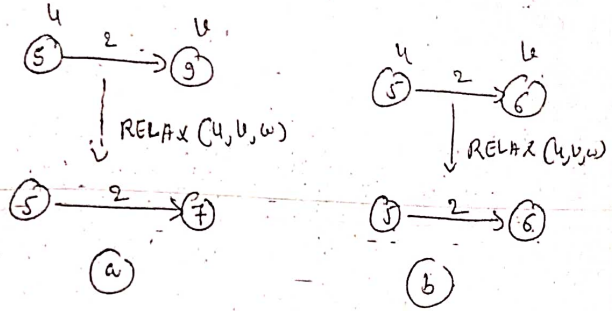
→ We initialize.

$\pi[v] = NIL$ for $\forall u \in V$

$d[v] = \infty$ for $\forall v \in V - \{s\}$

and $d[s] = 0$

The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, we update $d[v]$ and $\pi[v]$.




```
RELAX (u, v, w)
1   if d[v] > d[u] + w(u,v)
2      then d[v] ← d[u] + w(u,v)
3           π[v] ← u.
```

- In figure (a) relaxing an edge (u,v) decreased a shortest path estimate, whereas in (b) no estimate changes.

Note: ① In Dijkstra's algorithm, each edge is relaxed exactly once.

② In Bellman-Ford algorithm, each edge is relaxed many times.

Properties of shortest path & relaxation.

1) **Triangle Inequality.**
For any edge $(u, v) \in E$, we have
$\delta(s, v) \leq \delta(s, u) + w(u, v)$.

2) **Upper Bound property.**
We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.

3) **No path property.**
If there is no path from s to v, then we have $d[v] = \delta(s, v) = \infty$

4) **Convergence property.**

Bellman-Ford Algorithm: IMP

Question paper code KK-2612
Discuss Bell-man ford algo. with the help of example (10M)

→ The Bellman-Ford algorithm solves the single source ~~source~~ shortest paths problem in the general case in which edge weights may be negative.

→ Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \to R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from source s.

→ If there is such cycle, the algorithm indicates that no solution exists.

→ If there is no such cycle, the algorithm produces the shortest paths and their weights.

→ The algorithm uses relaxation, progressively decreasing an estimate $d[v]$ on the weight of the shortest path from the source s to each vertex $v \in V$, until it achieves the actual shortest path weight $\delta(s, v)$.

The algorithm returns true, if and only if the graph contains no negative weight cycles that are reachable from source s.

```
BELLMAN-FORD (G, w, s)
1.  INITIALIZE-SINGLE-SOURCE (G, s)        -> Θ(V)
2.  for i ← 1 to |V|-1                     } Θ(E)
3.      do for each edge (u,v) ∈ E         }
4.          do RELAX (u, v, w)             }
5.  for each edge (u,v) ∈ E                } O(E)
6.      do if d[v] > d[u] + w(u,v)         }
7.          then return False              }
8.  return True.
```

```
INITIALIZE-SINGLE-SOURCE (G, s)
1.  for each vertex v ∈ V
2.      do d[v] ← ∞
3.         π[v] ← NIL
4.  d[s] ← 0
```

```
RELAX (u, v, w)
1.  if d[v] > d[u] + w(u,v)
2.  then d[v] ← d[u] + w(u,v)
        and π[v] ← u.
```

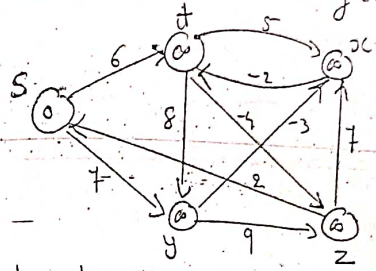
Here,

$d[v]$ indicates the shortest path estimate for vertex $v \in V$.

$\pi[v]$ indicates the predecessor of vertex $v \in V$.

$w(u,v)$ indicates weight of edge connecting vertex u to vertex v.

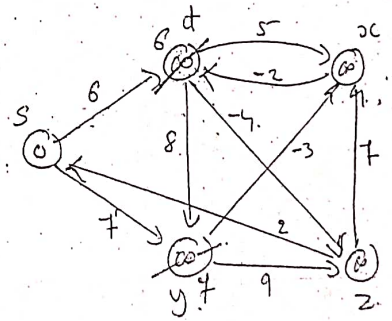
Example: Consider the graph.



Initially
$d[v] = \infty$
$d[s] = 0$
$\pi[v] = NIL$

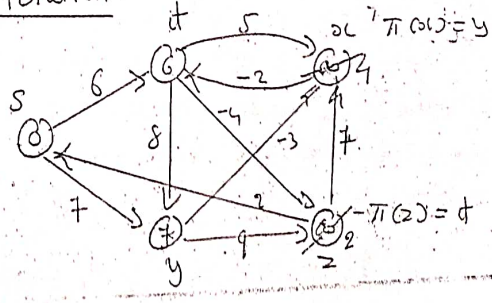
Total edges : (s,t), (s,y), (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,s), (z,x)

I^st iteration

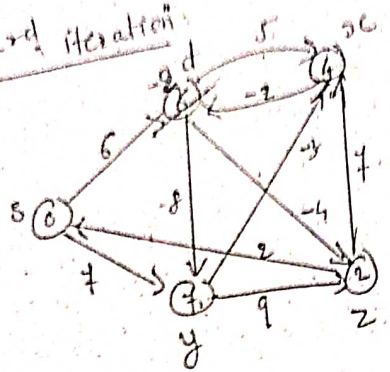


$\pi[t] = s$.
$\pi[y] = s$.

II^nd iteration.

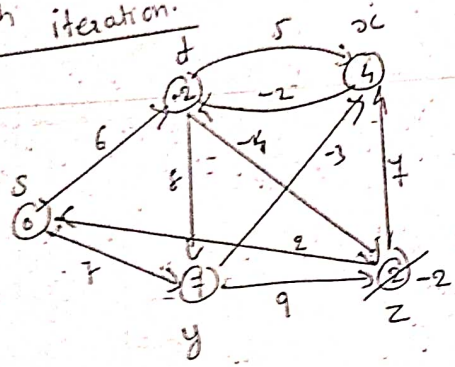


III rd iteration



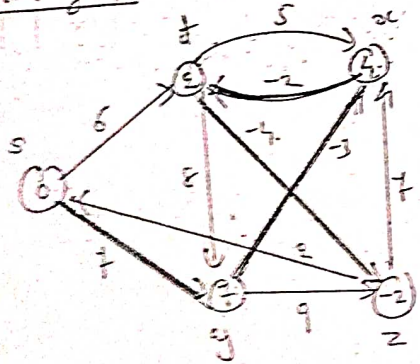
$\pi[4] = x$.

IV th iteration.



$\pi[2] = t$

Final graph.



$\pi[2] = t \rightarrow \pi[t] = x \Rightarrow \pi[x] = y$

$\Rightarrow \pi[y] = s$

∴ shortest path is $s \rightarrow y \rightarrow x \rightarrow t \rightarrow z$

cost $= 7 - 3 - 2 - 4 = -2$

Feasible Solution Problem

Problem 1. Carmen page 600

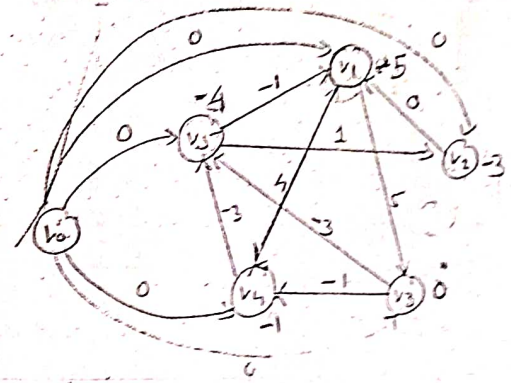
Find the feasible solution for the following system of difference constraints.

$x_1 - x_2 \leq 0$, $x_1 - x_5 \leq -1$, $x_2 - x_5 \leq 1$

$x_3 - x_1 \leq 5$, $x_4 - x_1 \leq 4$, $x_4 - x_3 \leq -1$

$x_5 - x_3 \leq -3$, $x_5 - x_4 \leq -3$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ -1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$



(v_2, v_1) (v_0, v_1)
(v_5, v_1) (v_3, v_2)
(v_5, v_2) (v_1, v_3)
(v_1, v_3) (v_3, v_4)
(v_1, v_4) (v_4, v_5)
(v_3, v_4)
(v_3, v_5)
(v_4, v_5)

Iteration 1

0	1	2	3	4	5
0	-1	0	0	-1	-4

Iteration 2

0	1	2	3	4	5
0	-5	-3	0	-1	-4

Iteration 3

0	1	2	3	4	5
0	-5	-3	0	-1	-4

Iteration 4

0	1	2	3	4	5
0	-5	-3	0	-1	-4

solution is : $x_1 = -5$, $x_2 = -3$, $x_3 = 0$, $x_4 = -1$, $x_5 = -4$

Example 2 paper code BB#456

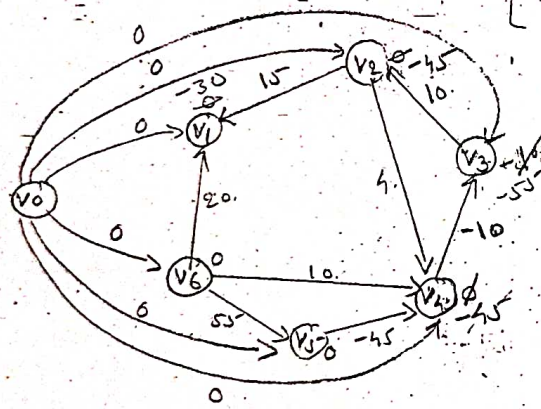
Find a feasible solution for the following.

① $x_1 - x_4 \le 10$ ② $x_2 - x_3 \le 10$ ③ $x_3 - x_4 \le -10$ ④ $x_4 - x_6 \le 10$ ⑤ $x_4 - x_5 \le -45$ ⑥ $x_5 - x_6 \le 55$ ⑦ $x_1 - x_6 \le 20$ ⑧ $x_4 - x_2 \le 4$.

6 unknowns

Soln

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \le \begin{bmatrix} 15 \\ 10 \\ -10 \\ 10 \\ -45 \\ 55 \\ 20 \\ 4 \end{bmatrix}$$



$C(v_2,v_1)$
$C(v_3,v_2)$
$C(v_4,v_3)$
$C(v_6,v_4)$
$C(v_5,v_4)$
$C(v_6,v_5)$
$C(v_6,v_1)$
$C(v_6,v_4)$
$C(v_0,v_1)$
$C(v_0,v_2)$
$C(v_0,v_3)$
$C(v_0,v_4)$
$C(v_0,v_5)$
$C(v_0,v_6)$

Iteration 1

0	1	2	3	4	5	6
0	0	0	-10	-45	0	0

Iteration 2

0	1	2	3	4	5	6
0	0	0	-55	-45	0	0

Iteration 3

0	1	2	3	4	5	6
0	0	-45	-55	-45	0	0

Iteration 4.

0	1	2	3	4	5	6
0	-30	-45	-55	-45	0	0

Iteration 5

0	1	2	3	4	5	6
0	-30	-45	-55	-45	0	0

solution is :

$x_1 = -30$ $x_2 = -45$ $x_3 = -55$

$x_4 = -45$ $x_5 = 0$ $x_6 = 0$

Problem 3 paper code HK-8618.

Find a feasible solution or determine that there is no feasible solution exist for the following system of difference constraints:

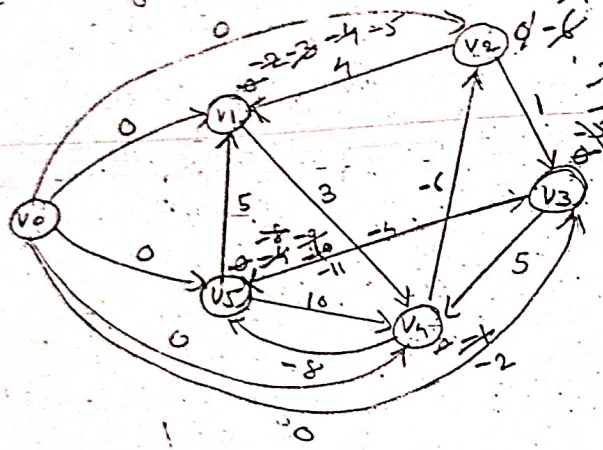
$x_1 - x_4 \le 4$, $x_1 - x_5 \le 5$, $x_2 - x_4 \le -6$

$x_3 - x_2 \le 1$, $x_4 - x_1 \le 3$, $x_4 - x_3 \le 5$

$x_4 - x_5 \le 10$, $x_5 - x_3 \le -4$, $x_5 - x_4 \le -8$

Soln

5 unknowns



(v_2, v_1)
(v_2, v_3)
(v_5, v_3)
(v_5, v_1)
(v_4, v_1)
(v_3, v_4)
(v_3, v_5)
(v_4, v_2)
(v_3, v_4)
(v_4, v_5)
(v_0, v_1)
(v_0, v_2)
(v_0, v_3)
(v_0, v_4)
(v_0, v_5)

Iteration 1

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

Iteration 2

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

Iteration 3

0	1	2	3	4	5
0	-3	-6	-5	0	-9

Iteration 4

0	1	2	3	4	5
0	-4	-7	-5	-1	-9

Iteration 5

0	1	2	3	4	5
0	-4	-7	-6	-1	-10

Iteration 6

0	1	2	3	4	5
0	-5	-8	-6	-2	-10

Iteration 7

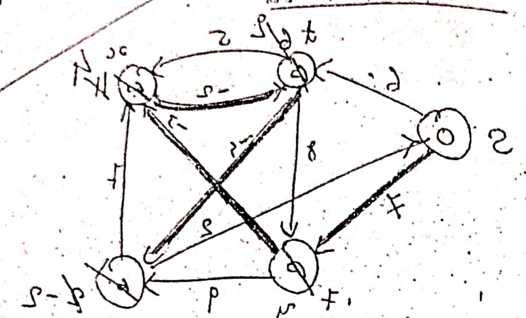
0	1	2	3	4	5
0	-5	-8	-7	-2	-11

Iteration 8

0	1	2	3	4	5

No feasible solution, since negative weight cycle is present.

Problem 4: Find feasible solution.

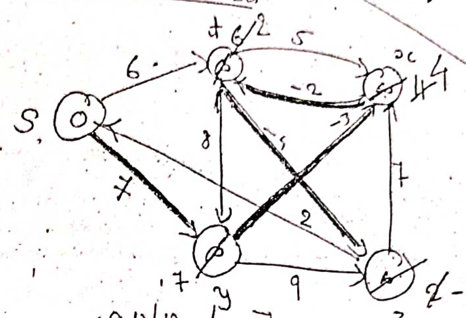


Complexity of Bellman Ford algo:

O(VE)

1. Initialize-single-source(G, s) takes Θ(V) time.
2. Relaxing each edge for |V|-1 times takes Θ(E) time.
3. The last for loop (for checking the negative weight cycle) takes O(E) times.

Bellman Ford



Iteration 1

	s	t	x	y	z
d[v]	0	6	∞ 4	7	2
π[v]	0	s	y	s	t

Iteration 2

s	t	x	y	z
0	2	4	7	2
0	x	y	s	t

Iteration 3

s	t	x	y	z
0	2	4	7	-2
0	x	y	s	t

Iteration 4

s	t	x	y	z
0	2	4	7	-2
0	x	y	s	t

BackTracking

start with z, z's predecessor is t
it's predecessor is x - - -

z → t → x → y → s

Ans: s → y → x → t → z shortest path.

All pair shortest path

The all pair shortest path computes path from each vertex v to every other vertex u, using standard single-source algorithms.

The Warshall's Algorithm:

Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in weighted, directed graph. (Negative weights are allowed)

- The algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a shortest path $p = \langle v_1, v_2, \ldots v_m \rangle$ is any vertex of p other than v_1 or v_m i.e any vertex in the set $\langle v_2, \ldots v_{m-1} \rangle$.
- Let the vertices of G be $V = \{1, 2, \ldots, n\}$ and consider a subset $\{1, 2, \ldots k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all the paths from i to j whose intermediate vertices are drawn from $\{1, 2, \ldots k\}$ and let p be the minimum weighted path from among them.

The Floyd-Warshall algorithm exploits a relationship between path p and shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots k-1\}$.

All pair shortest path

The all pair shortest path computes path from each vertex v to every other vertex u, using standard single-source algorithms.

The Warshall's Algorithm:

Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in weighted, directed graph. (negative weights are allowed)

- The algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a shortest path $p = \langle v_1, v_2, \ldots v_m \rangle$ is any vertex of p other than v_1 or v_m i.e any vertex in the set $\langle v_2, \ldots v_{m-1} \rangle$.
- Let the vertices of G be $V = \{1, 2, \ldots, n\}$ and consider a subset $\{1, 2, \ldots k\}$ of vertices for some k. for any pair of vertices $i, j \in V$, consider all the paths from i to j whose intermediate vertices are drawn from $\{1, 2, \ldots k\}$ and let p be the minimum weighted path from among them.

The Floyd-Warshall algorithm exploits a relationship between path p and shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots k-1\}$.

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 $\{1,2,\ldots,k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k\}$

$$i \xrightarrow{p_1} k \xrightarrow{p_2} j$$

A recursive definition is

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

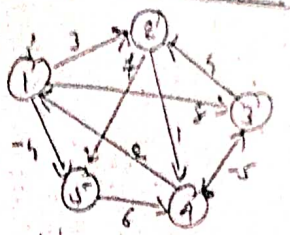
Warshall (W)

```
n ← rows[W]
D(0) ← W
For k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D(n)
```

Complexity → $\Theta(n^3)$

Allpair shortest Path:

1.



$D^{(0)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$\pi^{(0)}$

	1	2	3	4	5
1	N	1	1	N	1
2	N	N	N	2	2
3	N	3	3	N	N
4	4	N	N	N	N
5	N	N	N	5	N

a. k = 1

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$\pi^{(1)}$

	1	2	3	4	5
1	N	1	1	N	1
2	N	N	N	2	2
3	N	3	3	N	N
4	4	1	4	N	1
5	N	N	N	5	N

b. k = 2

$D^{(2)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$\pi^{(2)}$

	1	2	3	4	5
1	N	1	1	2	1
2	N	N	N	2	2
3	N	3	N	2	2
4	4	1	4	N	1
5	N	N	N	5	N

c. k = 3

$D^{(3)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$\pi^{(3)}$

	1	2	3	4	5
1	N	1	1	2	1
2	N	N	N	2	2
3	N	3	N	2	2
4	4	3	4	N	1
5	N	N	N	5	N

d. k = 4

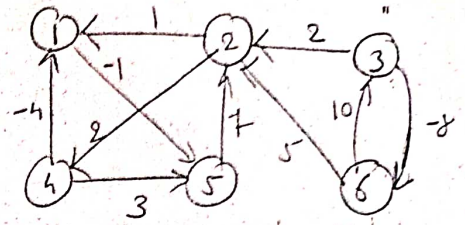
$D^{(4)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	3	0	-5	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\pi^{(4)}$

	1	2	3	4	5
1	N	1	4	4	1
2	4	N	4	2	1
3	4	3	N	2	1
4					
5					

$$D^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$



$D^{(0)} =$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

$D^{(1)} =$ $k=1$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	-5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

$D^{(2)} =$ $k=2$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(3)} =$ $k=3$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(4)}$ =

	1	2	3	4	5
1	0	∞	∞	∞	-1
2	-2	0	∞	2	-3
3	0	2	0	4	-1
4	-4	∞	∞	0	-5
5	-5	-7	∞	9	0
	3	5	10	7	2

∞
∞
-8
∞
∞
0

k=4

$D^{(5)}$ =

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	-2	0	∞	2	-3	∞
3	0	2	0	4	-1	-8
4	-4	2	∞	0	-5	∞
5	-5	-7	∞	9	0	6
6	3	5	10	7	2	0

k=5

$D^{(6)}$ =

	0	6	∞	∞	-1	∞
	-2	0	∞	2	-3	∞
	-5	-3	0	-1	-6	-∞
	-4	2	∞	0	-5	∞
	5	7	∞	9	0	∞
	3	5	10	7	2	0

k=5

$D^{(0)}$ =

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$D^{(1)}$ indicates k=1

In $D^{(1)}$ we will find paths for all nodes to all other nodes via node 1

i.e
- 1 to 1 via 1
- 1 to 2 via 1
- ⋮
- 1 to 5 via 1

} for node 1

Then
- 2 to 1 via 1
- 2 to 2 via 1
- ⋮
- 2 to 5 via 1

} for node 2

- 5 to 1 via 1
- 5 to 2 via 1
- 5 to 3 via 1
- ⋮
- 5 to 5 via 1

} node 5

Easy method suppose we are processing node=4

	1	2	3	4	5
1	0	3	8	∞	-4
2					
3					
4	2	∞	-5	0	∞
5					

Here k=1

Then first mark (1,0) then circle entire row 1. i.e it's circle.

Then add (4,1) + (1,1) and compar it wit (4,1), whichever is lesser replace cell (4,1)

→ then add (4,1) + (4,2) and compar with (4,2) which ever is minimum replace cell (4,2) with it.

Eg ① $a_{4}=2$

(4,1)
$2+0=2 \not< 2$

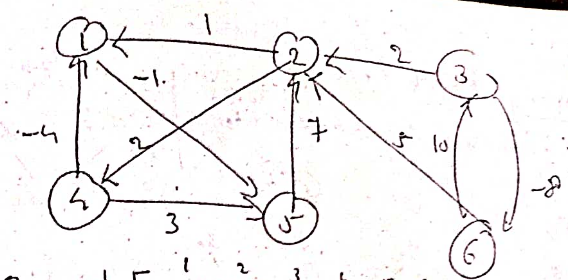
(4,2)
$2+3=5 < \infty$ replace

(4,3)
$2+8=10 > -5$

(4,4)
$2+\infty=\infty > 0$

(4,5)
$2-4=-2 < \infty$ replace.

1.



D^{0} =

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

$D^{(1)}$ =
K=1

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	-5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

$D^{(2)}$ =
K=2

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

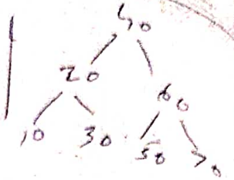
} for 6th row

$D^{(3)}$ =
K=3

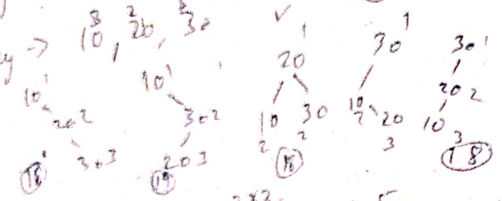
	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

Optimal Binary Search Tree

Key -> 10, 20, 30, 40, 50, 60, 70 (log n)



Key -> 10, 20, 30



height bal BST.

freq OBST.

n key -> $\frac{2n_{C_n}}{n+1} = \frac{2\times3_{C_3}}{3+1} = 5$

	1	2	3	4
key	10	20	30	40
freq	4	2	6	3

i \ j	0	1	2	3	4
0	0	4[1]	8[1]	20[3]	26[3]
1		0	2[2]	10[3]	16[3]
2			0	6[3]	12[3]
3				0	3[4]
4					0

$w(0,4) = \sum_{i=1}^{4} f(i)$ 15

j-i=0: 0-0=0, 1-1=0, 2-2=0, 3-3=0, 4-4=0

j-i=1: 1-0=1 (0,1), 2-1=1 (1,2), 3-2=1 (2,3), 4-3=1 (3,4)

c(0,1) c(1,2) c(2,3) c(3,4)
10 20 30 40
4 2 6 3

c(0,2)
c(0,0) + c(1,2) + w(0,2) = 0 + 2 + 6 = 8

c(0,1) + c(2,2) + w(0,2) = 4 + 0 + 6 = 10

j-i=2: 2-0=2 (0,2), 3-1=2 (1,3), 4-2=2 (2,4)

$C[i,j] = \min_{i<k\le j}\{C[i,k-1] + C[k,j]\} + w(i,j)$

c(1,3)

j-i=3: 3-0=(0,3), 4-1=(1,4)

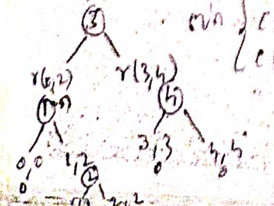
c(0,3) w(0,3) = 12

min { c(0,0) + c(1,3) + w(0,3) ; c(0,1) + c(2,3) + –"– ; c(0,2) + c(3,3) + –"– }
0+10+12 ; 4+6+12 ; 8+0+12 = 20

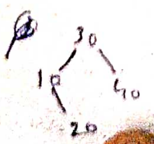
C(1,4)
C1,1 + 2,4
1,2 + 3,4
1,3 + 4,4

j-i=4: 4-0=(0,4)

r(0,4) = 3



c(0,4)
Min { 0,0 + 1,4 ; 0,1 + 2,4 }



Minimum Cost Spanning Tree:

Let G be a connected, un-directed graph G = (V, E), where V is the set of vertices and E be the set of possible edges between the pairs of vertices, and for each edge (u,v) ∈ E, we have a weight w(u,v) specifying a cost to connect u and v.

We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight

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

Since, T is acyclic and connects all of the vertices, it must form a tree called the 'spanning tree'.

There are two algorithms to find minimum spanning tree.

1) Krushkal's algorithm
2) Prim's

If we use binary heaps then complexity of the above two algo. is O(E log V)

-> Prim's algorithm can be improved, by using Fibonacci heaps, and can be speed upto run in time O(E + V log V), which is an improvement if |V| is much smaller than |E|.

The two algorithms are greedy algos. The greedy strategy advocates making the choice that is the best at the moment.

Kruskal's algorithm:

```
MST-KRUSKAL(G, w)
1.  A ← ∅                                  ⟶ O(1)
2.  for each vertex v ∈ V[G]
3.      do MAKE-SET(v)
4.  Sort the edges of E into non-decreasing order by weight w.   ⟶ O(E lg E)
5.  for each edge (u,v) ∈ E, taken in nondecreasing order by weight
6.      do if FIND-SET(u) ≠ FIND-SET(v)
7.          then A ← A ∪ {(u,v)}
8.               UNION(u,v)
9.  Return A
```

Let $G = (V, E)$ is a connected, un-directed graph. where V is a set of vertices in G
E is a set of all edges in G

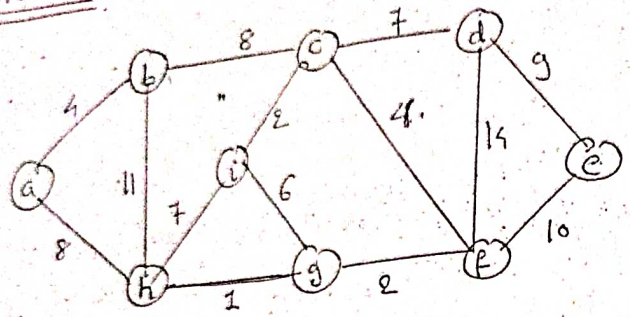
At any time of instance set A, where $A \subseteq T$ represents the subset of minimum spanning tree T of graph G. We add edge (u,v) to A by using union operation $\{A \cup edge(u,v)\}$ such that, after adding edge(u,v) to A, the resultant A is also a minimum spanning tree.
→ We call such an edge a safe edge for A, since it can be safely added to A while maintaining minimum spanning tree properties.

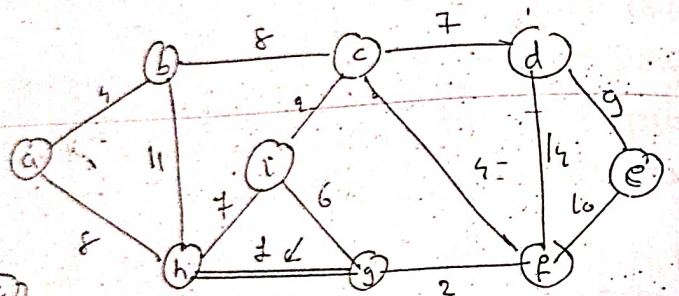
1. ~~Initially~~ A is ~~initialized~~ to ∅.

2. Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight.
→ Let C_1 and C_2 denote two trees that are connected by edge(u,v). Since (u,v) must be light edge connecting C_1 to some other tree, implies that (u,v) is a safe edge for C_1
→ Kruskal's algorithm is greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

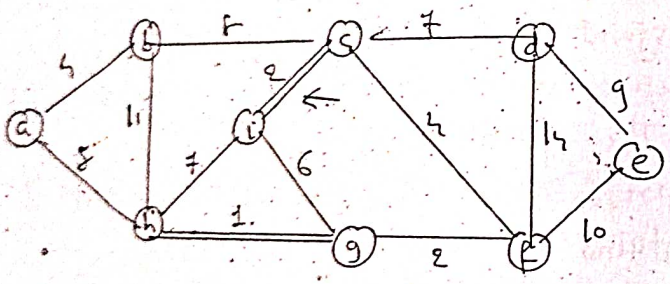
Kruskal's algorithm uses a dis-joint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest.
→ The operation FIND-SET(u) returns a representative element from the set that contains u. Thus we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). The combining of trees is accomplished by UNION procedure.

Problem 1 dec-2004 code - BB - 7395

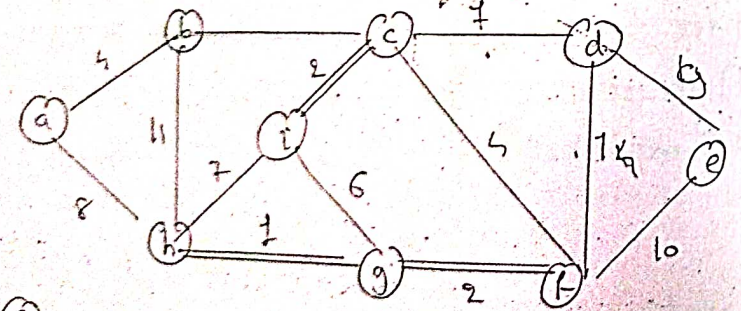




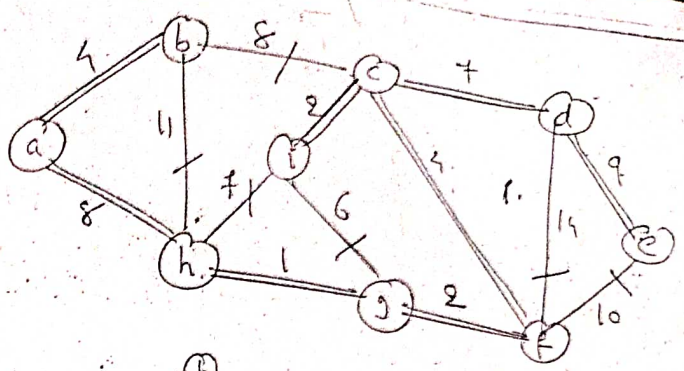
(a)

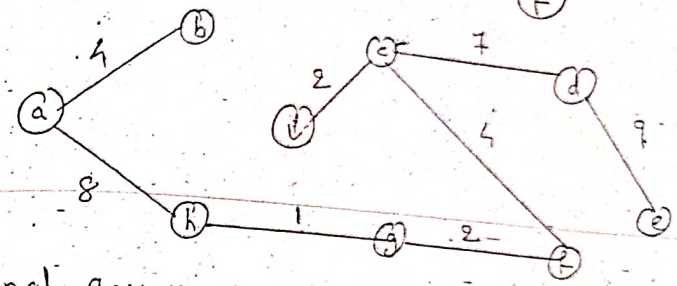


(b)



(c)

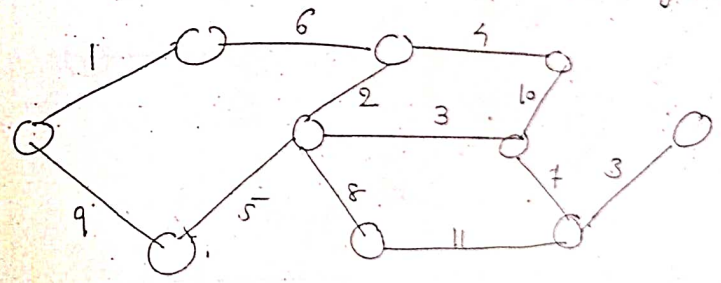




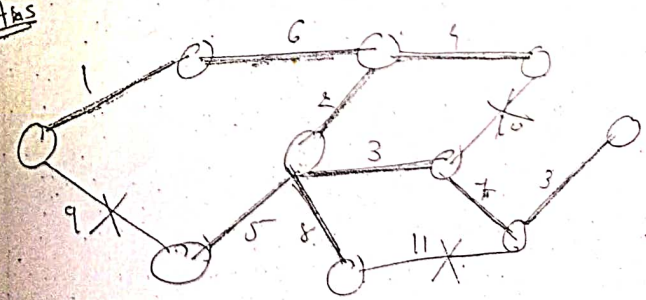
Final answer Cost = 37

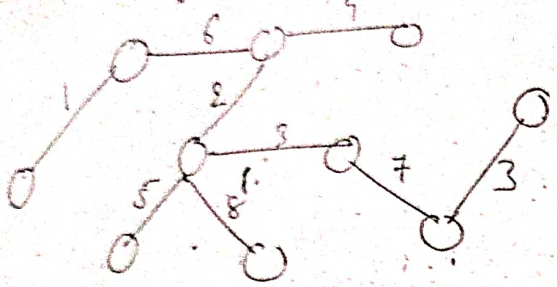
Problem 2 2006, Nov. paper code - 6B7756

Evaluate the minimum cost spanning tree



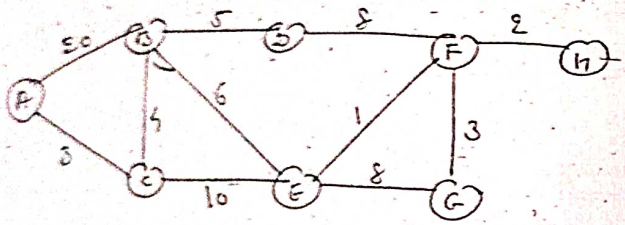
Ans



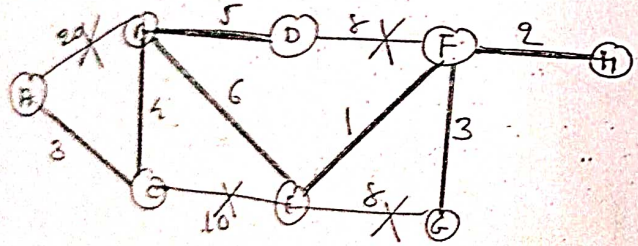


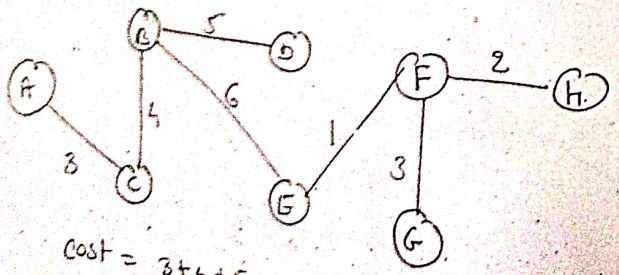
cost = 39

Problem 3 Evaluate the minimum weight spanning tree. Paper code-BB-7996.



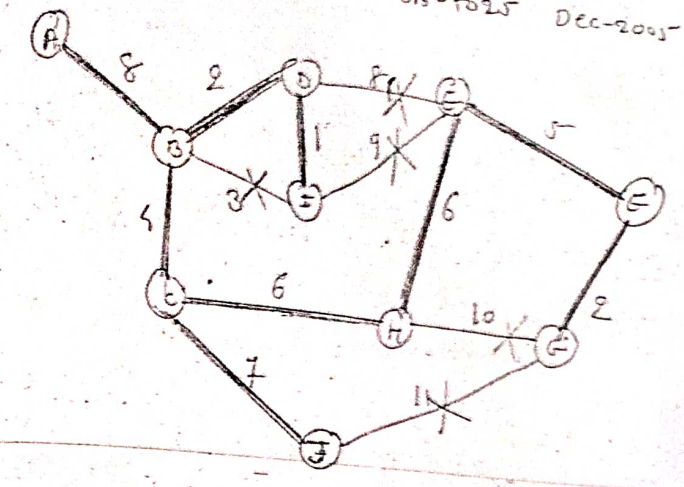
Ans



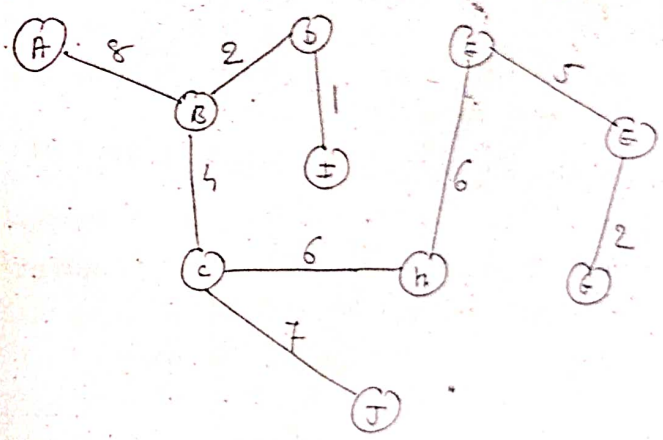


cost = 3+4+5+6+1+2+2 = 24.

Problem: 4 Paper code: BB-7025 Dec-2005-

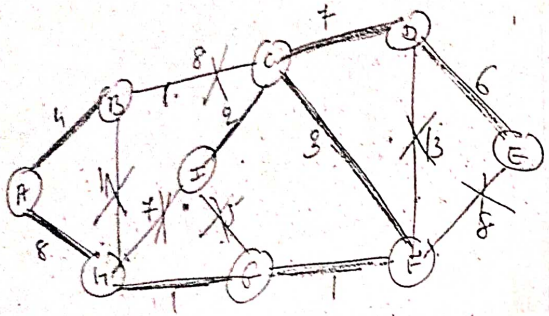


Ans

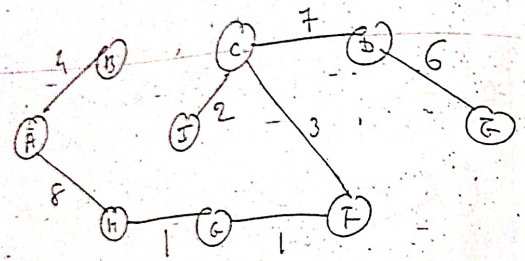


Cost = 41

Problem 5 Dec-2004 code-7382



Ans



Cost = 32

Complexity of Kruskal's algo: 10

(1) Initializing the set A in line 1 takes O(1) time.

(2) Time to sort edges requires O(ElgE)

(3) The total time to perform O(E) FIND-SET and UNION operations along with |V| MAKE-SET operations is,

$$O((V+E)\alpha(V))$$ time,

where α is a very slowly growing function.

(4) Because G is assumed to be connected, we have $|E| \geq |V|-1$, and so disjoint-set operations take $O(E\alpha(V))$ time.

(5) Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$

The total running time of Kruskal's algorithm is,

$$O(E \lg E)$$

$$T(n) = O(1) + O(E \log E) + O((V+E)\log$$
$$+ O(E \log E)$$
$$= O(E \log E) + O(V \log E) + O(E \log$$
$$+ O(E \log E)$$
$$= O(E \log E) \quad / \boxed{O(E \log V)}$$

Applications of MST

(1) Spanning trees can be used to obtain an independent set of circuit equations for an electric network.

(2) In a computer network, network topology forms a graph whose nodes are nothing but routers and edges are links between pair of routers. Distance between two routers is considered as cost of the link.

→ A minimum spanning tree is formed that includes all the routers but contain no loops.

→ When a packet arrives at a router, a router can use minimum spanning tree to broadcast it all other routers, avoiding looping of packets.

→ Thus this method saves the bandwidth and generates the ~~ex~~ absolute no. of packets necessary to do the job.

Minimum Cost Spanning tree

Prim's Algorithm

definition (minimum spanning tree):

Let G be a connected, undirected graph $G = (V, E)$, where V is the set of vertices and E be the set of possible edges between the pairs of vertices. And for each edge $(u,v) \in E$, we have a weight $w(u,v)$ specifying a cost to connect u and v.

We then find an acyclic subset $T \subseteq E$ that connects all the vertices and whose total weight

$$W(T) = \sum_{(u,v) \in T} w(u,v) \text{ is minimized.}$$

Since, T is acyclic and connects all of the vertices, it must form a tree called the spanning tree.

Prim's algorithm operates much like Dijkstra's shortest path algorithm.

- Let $G = (V, E)$ is a undirected, connected graph where V is a set of vertices in G
  E is a " " edges in G.
- At any time instance set A, when $A \subseteq T$ represents the subset of minimum spanning tree T of graph G.

- Prim's algorithm has a property that the edges in A always form a single tree.
- Prim's algorithm uses a greedy method to obtain an MST.
- At each step, it chooses an edge to be added in set A that results in a minimum increase in the sum of the costs of the edges so far included in A.

⇒ The tree starts from an arbitary root vertex r and grows until the tree spans all the vertices in V.

- At each step a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$. This rule adds only edges which are safe for A, therefore when algorithm terminates, the edges in A form a minimum spanning tree.

Algorithm.


```
MST_PRIM (G, ω, r).
1  for each u ∈ V[G]
2      do key[u] ← ∞
3         π[u] ← NIL
4  key[r] ← 0
5  Q ← V[G]
6  while Q ≠ 0
7      do u ← EXTRACT_MIN(Q)
8         for each v ∈ adj[u]
9             do if v ∈ Q and ω(u,v) < key[v]
10                then π[v] ← u
11                     key[v] ← ω(u,v).
```

→ For each vertex v, key[v] is the minimum weight of any edge connected to v to a vertex in the tree.

key[v] = ∞ if there is no such edge.

→ All vertices that are not in the tree reside in a min-priority queue Q based on a key field.

→ The field π[v] names the parent of v in tree.

→ When the algo. terminates, the min-priority Q is empty.

→ The minimum spanning tree A for G is: Thus

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

Working of algo:

1) Line 1-5 sets the key of each vertex to ∞ (except the root r) and ~~Root~~
2) Root is's key is set to 0.
3) The parent of each vertex is set to NIL.
4) Initialize the minimum priority queue Q to contain all the vertices.

The algorithm maintains the following three parts:

1. $A = \{(v, \pi(v)) : v \in V - \{r\} - Q\}$.
2. The vertices already placed in to the minimum spanning tree A are those in V−Q.
3. For all vertices $v \in Q$, if $\pi(v) \neq NIL$, then $key[v] < \infty$ and $key[v]$ is the weight of the light edge $(v, \pi[v])$ connecting v to some vertex already placed in to A.

Complexity of Prim's Algo.

(1) If Q is implemented as binary heap,

→ Line 1-5 → O(V).

→ The body of the while loop is executed in |V| times, since EXTRACT-MIN() takes O(lg V) time, the total time for all EXTRACT-MIN() is O(V lg V).

→ The for loop in line 8-11 is executed in O(E) times.

→ Within the for loop the test if $v \in Q$ can be implemented in constant time

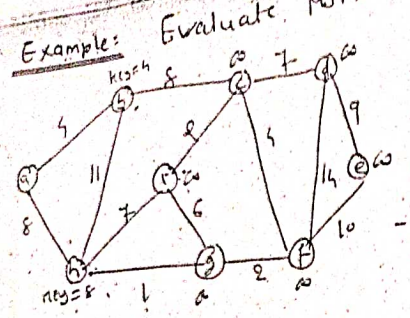
→ The assignment $key[v] \leftarrow w(u,v)$ can be implemented in O(lg V) times.

Thus,

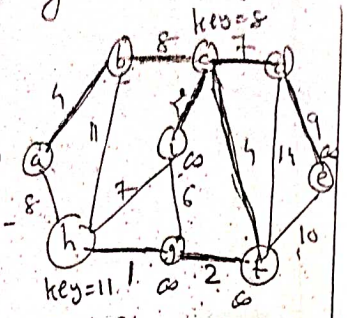
Total time = O(V lg V + E lg V)

= O(E lg V).

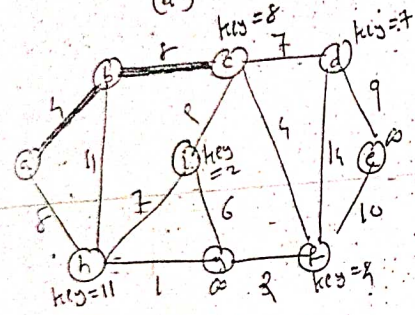
Example: Evaluate MST. using Prim's algo.



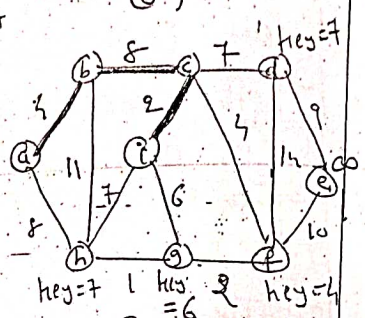
(a)



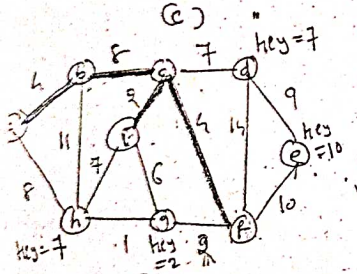
(b)



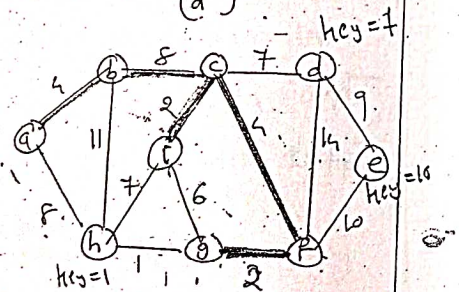
(c)



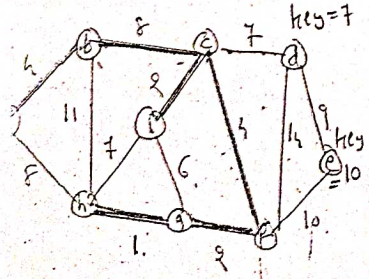
(d)



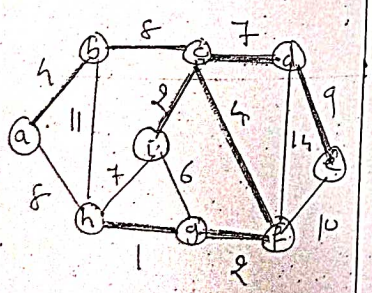
(e)



(f)

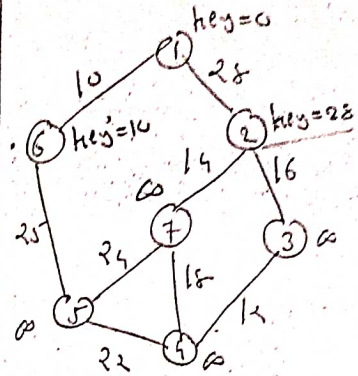


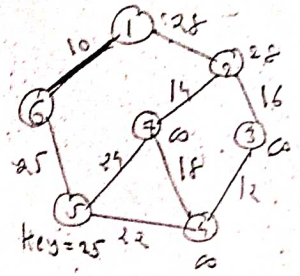
(g)



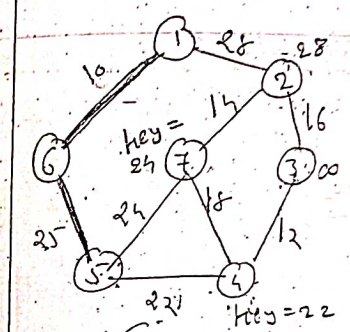
(h)

Problem: Solve minimum spanning tree using Prim's algo

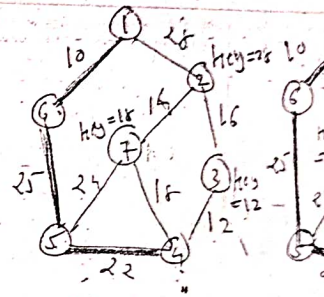




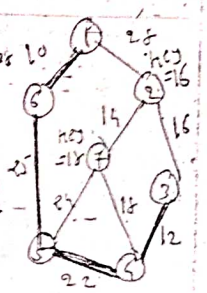
(a)



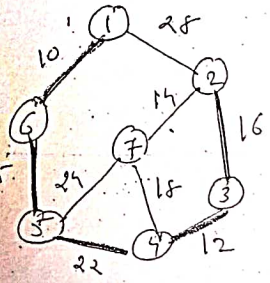
(b)



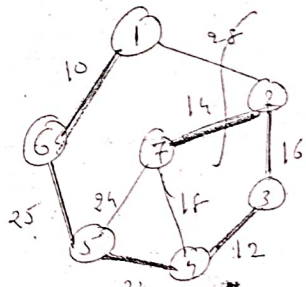
(c)



(d)

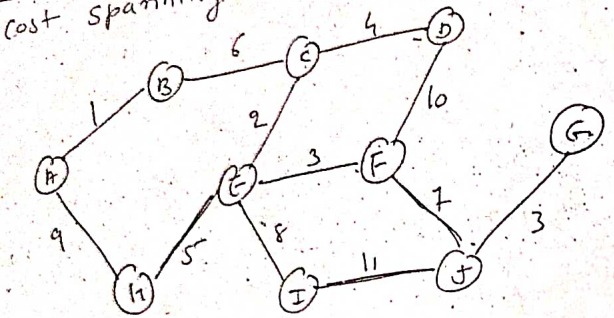


(e)

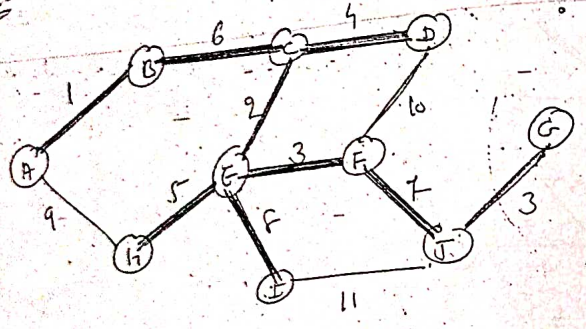


(f)

Example: Evaluate the minimum cost spanning tree using Prim's algo.



Ans:



Knapsack Problem:

Q-1 Evaluate the complexity of 0-1 knapsack and fractional knapsack. Is fractional knapsack a good approximation of 0-1 K.S. (13M)

Q-2. State 0-1 knapsack problem. State how it can be solved using greedy algorithm. Evaluates its runtime complexity. (08).

Fractional knapsack. O(n log n)

- x_1 are given n objects and a knapsack or bag.
- Object i has a weight w_i
- knapsack has a capacity m.
- Each object i has a profit p_i.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then profit of $p_i x_i$ is earned.

Objective: To fill the knapsack to maximize the total profit earned.

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad \text{---} (1)$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \quad \text{---} (2).$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers.

Greedy approach to knapsack

Consider the following instance of knapsack problem: $n=3$ $m=20$

$(P_1, P_2, P_3) = (25, 24, 15)$ and

$(w_1, w_2, w_3) = (18, 15, 10)$.

1st approach: greedy by profit.

Here, we try to fill the knapsack by including next object with largest profit.

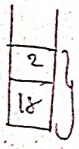
- If the object under consideration doesn't fit, then fraction of it is included to fill the knapsack.

Soln Since item 1 has maximum profit = 25 we fill the sack with item 1.

Sack

Now sack has left with capacity 2.

→ Since item 2 has next highest profit we take the fraction of ~~sack~~ to item 2 to fill the sack. Add the profit $= 24 \times \frac{2}{15} = 3.2$



$m=20$ Total profit

$= 25 + 3.2 = 28.2$.

2nd approach: greedy by profit per unit

or greedy by density of capacity.

- This approach tries to balance between the rate at which profit increases and the rate at which capacity is used.
- At each step we include the object which has the maximum profit per unit of capacity.
- This means that objects are considered in order of the ratio P_i/w_i.

Soln

Item	Profit	weight	P_i/w_i
1	25	18	1.38
2	24	15	1.60
3	15	10	1.50.

Sort them according to P_i/w_i in descending order

Item	P_i	w_i	P_i/w_i
2	24	15	1.6
3	15	10	1.50
1	25	18	1.38

→ Since item 2 has maximum P_i/w_i ratio, we fill the sack ~~wo~~ with item 2

Sack

Profit earned = 24.

→ Next highest P_i/w_i is of item 3. So fill the sack with the fraction of item 3.

→ Remaining sack capacity $= 20 - 15 = 05$

Profit earned $= 15 \times \frac{5}{10} = 7.5$

$\therefore$ Total profit $= 24 + 7.5 = \boxed{31.5}$

Thus greedy method using profit as its measure does not guarantees optimal soln. But greedy method using p_i/w_i as its measure always obtain an optimal solution.

Algorithm

```
Greedy-fractional-knapsack (w, v, M)
For i = 1 to n
    do x[i] = 0
       weight = 0
while weight < M
    do i = best remaining item
       If weight + w[i] ≤ M
          then x[i] = 1   // 100% item is taken
               weight = weight + w[i]
          else
               x[i] = (M - weight)/w[i]
               weight = M
return x
```

M is the capacity of knapsack

v[i] is the value (profit) of i^{th} item
w[i] is the weight of i^{th} item

Analysis

1) If the items are already sorted into decreasing order of v_i/w_i then the while loop takes time in $O(n)$. Therefore the total time including the sort is in $O(n \lg n)$

2) If we keep items in heap with largest v_i/w_i at the root (max heap). Then
   - Creating heap takes $O(n)$ time
   - while loop now takes $O(\log n)$ time

0-1 knapsack Problem

Problem statement (knapsack):

We are Given n objects and a knapsack or bag of maximum capacity W.

- Each object has a weight w_i and a value (profit) v_i.
- What items should we take in knapsack in order to maximize profit?

There are two versions of problem:

1) Fractional knapsack:

We can take fractions of items, meaning that the items can be broken into smaller pieces, so that we can include a fraction x_i of item i, where $0 \le x_i \le 1$

2) 0-1 knapsack:

Items cannot be broken into smaller pieces, we need to take an item or leave it, but may not take a fraction of an item.

Properties of fractional knapsack problem:

- Exhibit greedy choice property.
  - ⇒ Greedy algorithm exists.
- Exhibit optimal substructure property.

Properties of 0-1 knapsack problem:

- Exhibit no greedy choice property
  - ⇒ no greedy algo. exists.
- Exhibit optimal substructure property.
- Only dynamic programming algo exists.

Dynamic Programming solution to 0-1 knapsack:

- Let i be the highest-numbered item in an optimal solution S for W pounds.
- Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and the value to the solution S is v_i plus the value of the subproblem. (when v_i is the value of the i^{th} item).
- We can express this fact in the following formula:

Define $c[i, w]$ to be the solution for item 1, 2, 3, ..., i and maximum weight w. Then,

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max\{v_i + c[i-1, w-w_i], & \text{if } i > 0 \text{ \& } w \ge w_i \\ \quad c[i-1, w]\} & \end{cases}$$

- This says that the value of the solution to i items either include i^{th} item, or do not include i^{th} item.
- If it includes i^{th} item, then $c[i, w]$ will be v_i plus subproblem solution for $(i-1)$ and the weight excluding w_i.
  - i.e $v_i + c[i-1, w-w_i]$

- If it does not include i^{th} item, then $c[i,w]$ will be, subproblem's solution for $(i-1)$ items and the same weight.
- That is, if we pick up item i, then we take v_i value, and we can choose from items $w-w_i$ and get $c[i-1, w-w_i]$ additional value.
- On the other hand, if we do not pick up item i, we can choose from item $1,2,\ldots i-1$ upto the weight limit w, and get $c[i-1, w]$ value.
- The better of these two choices should be made.

Inputs

1) maximum weight w (i.e Such capacity)
2) The no. of items n
3) Sequence $v = \langle v_1, v_2, \ldots v_n \rangle$ represents value of i^{th} item, $i \le i \le n$.
4) Sequence $w = \langle w_1, w_2, \ldots w_n \rangle$ represents weight of i^{th} item, $1 \le i \le n$.
5) Two dimensional array $c[0\ldots n, 0\ldots w]$ whose entries are computed in a row-major order.

At the end of the computation $c[n,w]$ contains the maximum value that can be picked up into the knapsack.

Dynamic 0-1 knapSack (v, w, n, W)

```
For  w = 0 To W
     do c[0,w] = 0  // Initialize 1st row to 0
For  i = 1 to n
     do c[i,0] = 0  // Initialize 1st column to 0
     For  w = 1 to W
          Do if (wi ≤ w)
               Then if  vi + c[i-1, w-wi] ≤ c[i-1, w]
                    Then c[i,w] = vi + c[i-1, w-wi]
               else
                    c[i,w] = c[i-1,w]
          else
               c[i,w] = c[i-1,w]
```

Finding set of items:

The set of items to take can be deduced from the table, starting at $c[n,w]$ = and ~~c[i~~ backtracking where the optimal values came from.

If $c[i,w] = c[i-1,w]$ then item i is not the part of the solution, and we continue to trace $c[i-1,w]$. Otherwise item i is the part of solution, and we continue tracing with $c[i-1, w-W]$.

Analysis: (0-1 knapsack).

Overall complexity: $\Theta(nw)$

→ $\Theta(nw)$ times to fill the c-table which has $(n+1).(w+1)$ entries, each require $O(1)$ time to compute.

→ $O(n)$ time to trace the solution, b'cos the tracing process starts ~~with~~ n-row n of the table and moves up 1 row at each step.

Question: Show that the greedy strategy does not work for the 0-1 knapsack.

Soln

Problem: "A thief robbing a store finds n items; the i^{th} item is worth value v_i dollars and weight w_i pounds. He 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?

This is called the 0-1 knapsack problem, because each item he either be taken or left behind; the thief cannot take a fractional amount of an item or an item more than once.

To show that the greedy strategy does not work for the 0-1 knapsack., consider the following example.

items	value v_i	weight w_i	v_i/w_i
1	$60	10	6
2	$100	20	5
3	$120	30	4

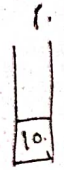
and knapsack capacity W is 50.

→ Here there are total three items, and knapsack can hold 50 pounds.

→ The value/pound for item1 is 6, for item2 5 and for item3 it is 4.

We use using a greedy by density approach to 0-1 knapsack.

→ The greedy strategy, therefore selects item1 first.



profit = $60

→ Next it selects item2 with weight 20

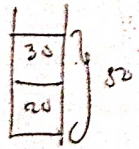
∴ profit = $60 + $100 = $160.

→ The next item is item3 cannot be taken because ~~including~~ adding weight of item3 to the sack exceeds the total capacity 50.

The above method does not give the optimal solution.

The optimal solution is.

① Select item 2 and item 3.



Profit = $100 + $120
= $220.

Thus, we can say that, greedy method does not work for 0-1 knapsack.

Problems based on knap-sack:

1) Find a fractional knapsack solution for a sack with capacity 100

item	capacity	value
A	90	30
B	50	50
C	20	70
D	35	20

Soln: calculate value/capacity ratio for each item.

item	value	capacity	value/capacity
A	30	90	0.33
B	50	50	1.0
C	70	20	3.5
D	20	35	0.57

Sort in descending order of value/capacity

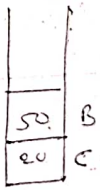
item	value	capacity	value/capacity
C	70	20	3.5
B	50	50	1.0
D	20	35	0.57
A	30	90	0.33

① 

Sack.

Profit = 70.
earned

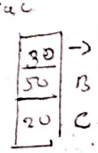
m = 100 - 20
= 80. left

② 

Profit = 70 + 50
earned = 120

m = 100 - 70
= 30. left

③ Take fraction of item D to fill the sac



Profit earned
= 120 + (30 × 20/35)
= 120 + 17.1
= 137.1 Ans

2) Given 3 items & a knapsack of 150 kgs to hold it, with the following constraint.
paper code : BB-8392 (F14)

1. item	weight (kg)	value (Rs)
1	30	180
2	60	300
3	90	360

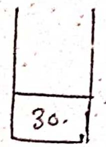
Sol^n $m = 150$ kg.

item	w_i	P_i	P_i/w_i
1	30	180	6
2	60	300	5
3	90	360	4

- Sort in descending order of P_i/w_i (already sorted)

item	w_i	P_i	P_i/w_i
1	30	180	6.
2	60	300	5
3	90	360	4

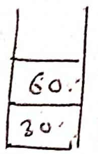
(1)



Profit = 180.

m = 150 - 30
= 120 left

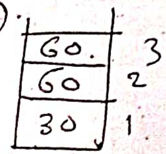
(2)



Profit = 180 + 300
= 210.

m = 150 - 90
= 60 left

(3)



Profit.
$= 210 + \left(360 \times \frac{60}{90}\right)$
$= 210 + 240$
$= \boxed{450 \text{ Rs}}$ Ans

Ans : 450 Rs

Difference between greedy and dynamic programming :

1. Because the both greedy and dynamic programming strategies show optimal-substructure property, one might be tempted to generate a dynamic programming solution to a problem when a greedy solution is suffice, or one might mistakenly think that a greedy solution works when in fact a dynamic programming solution is required.

Intro

- The most important difference between greedy algorithms and dynamic programming is that we don't solve every optimal subproblem with greedy algorithms. Greedy algorithms can be used to produce sub-optimal solutions. This solutions are not always optimal.

greedy ✓ optimal

- In dynamic programming, we make choice at each step, but the cot choice may depend on the solutions to sub-problems. In greedy algorithm, we make whatever choices seems best at the moment and then solve the sub-problems arising after the choice is made.

Choice

- Thus, dynamic programming solves the sub problems bottom up, whereas greedy strategy usually progresses in top-down fashion, making one greedy choice after another, interactively reducing each given problem instance to a smaller one.

bottom up / top down

Question paper code BB-7756

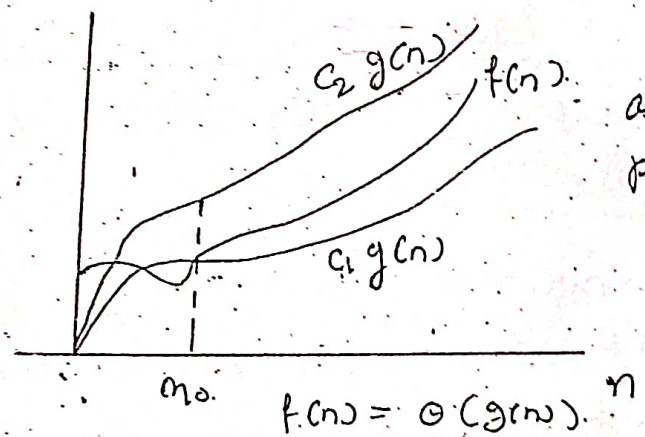
Define O, Θ and Ω. State their inter-relationship. (6M).

Asymptotic notations are used to describe the asymptotic running time of an algorithm.

1] Θ Notation: (Theta).

Definition: The function $f(n) = \Theta(g(n))$ if and only if there exists positive constants c_1, c_2 and n_0 such that

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.



$g(n)$ should be as small as possible.

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

Here the function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constant c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for sufficient large n.

As shown in the fig. for all value of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and or at below $c_2 g(n)$.

- We say that $g(n)$ is an asymptotically tight bound for $f(n)$.
- Thus theta notation gives both upper and the lower bound on $f(n)$.
- The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically non-negative. Consequently, the function $g(n)$ itself must be asymptotically non-negative.

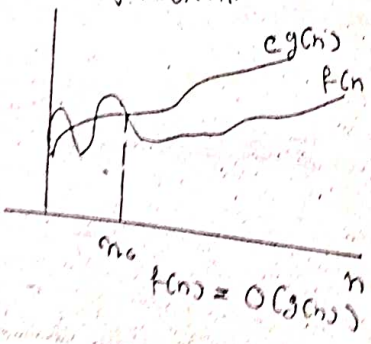
2] O - Notation. (Big O).

- The Θ-notation asymptotically bounds a function $f(n)$ from above and below.
- When we have only an asymptotic upper bound, we use O-notation.

Definition: The function $f(n) = O(g(n))$ if and only if there exist positive constant c and n_0 such that

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

- We use O-Notation to give an upper bound on a function.



$g(n)$ should be as large as possible.

- As shown in the diagram, for all values of n to the right of n_0 ($n \geq n_0$), the value of function $f(n)$ is on or below $g(n)$.
- We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$.

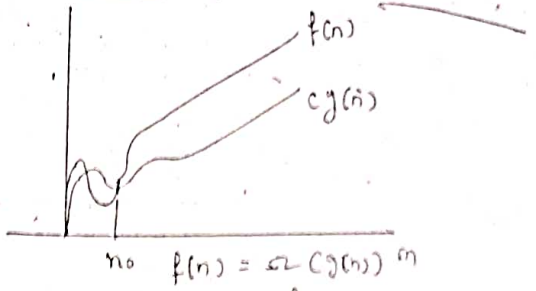
Note: $f(n) = \Theta(g(n))$ implies that $f(n) = O(g(n))$

3] Ω - Notation (Omega)

- Ω Notation provides asymptotic lower bound on function.

Definition: The function $f(n) = \Omega(g(n))$ if and only if there exist positive constant c and n_0 such that

$f(n) \geq c\,g(n)$, for all $n \geq n_0$



As shown in the figure, for all $n \geq n_0$, the value of $f(n)$ is on or above $g(n)$.

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

Imp

RSA Public-key Cryptosystem: (10M).

→ Public key crypto system can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them.

→ A public-key cryptosystem also enables a party to append an unforgeable "digital signature" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document.

It can be checked by any one, forged by no one, yet loses its validity if any bit of the message is altered.

It therefore provides authentication of both
- the identity of the signer and
- the contents of the signed message.

It is a perfect tool for electronically signed business contracts, electronic checks, electronic purchase orders etc.

others: electronic communication which requires Authentication

c key cryptosystem

In public key crypto system, each cipant has both a public key and a et key.

Each key is a piece of information. example, in the RSA cryptosystem, ch key consists of a pair of integers. Suppose the participants are "A" and "B" then we denote their their public and secret key as P_A, S_A for A and P_B, S_B for B.

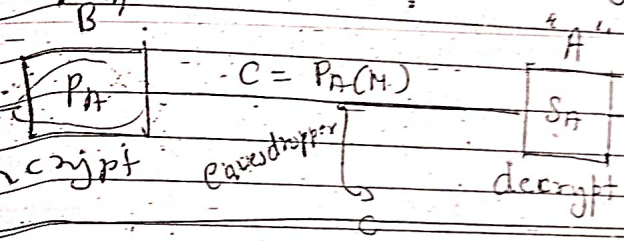
Each participant creates his own public and secret keys. Each keeps his secret key secret, but can expose his public key to anyone. Infact, often public keys are available in a public directory, so that any participant can easily obtain the public key of any other participant.

The function corresponding to A's public key P_A is denoted by $P_A(\,)$ and the function corresponding to A's secret key S_A is denoted by $S_A(\,)$

public key and secret key for any cipant is a "matched pair" such they specify functions that are of each other.

$$M = S_A(P_A(M))$$

$$M = P_A(S_A(M)) \quad \text{for any message } M$$



In public-key cryptosystem, it is al that no one but "A" be able to ute the function $S_A(\,)$ in any practical nt of time.

The assumption that only A can ute $S_A(\,)$ must hold even though one knows P_A and can compute $P_A(\,)$,

public-key encryption works as follows:

Suppose B wishes to send a message A.

1) B obtains A's public key P_A (from a public directory).
2) B computes the cipher text
$C = P_A(M)$ where M is the message.
3) When A receives the cipher text C, he applies his secret key S_A to retrieve the original message.
$M = S_A(C)$.

Because $S_A()$ and $P_A()$ are inverse functions A can compute M from C.

Digital Signatures:

Suppose A wishes to send B digitally signed response M'.
The digital signature scenario proceeds as follows.
1) A computes his digital signature σ for the message M' using his secret key S_A
$\sigma = S_A(M')$
2) A sends the message/signature pair (M', σ) to B.
3) When B receives (M', σ), he can verify that it originated from A using A's public key P_A.

$M' = P_A(\sigma)$

If the equation holds, then B concludes that the message M' was actually signed by A.

If the equation doesn't hold, B concludes either that the message M' or the digital signature σ was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

An important property of a digital signature is that it is verifiable by anyone who has access to the signer's public key.

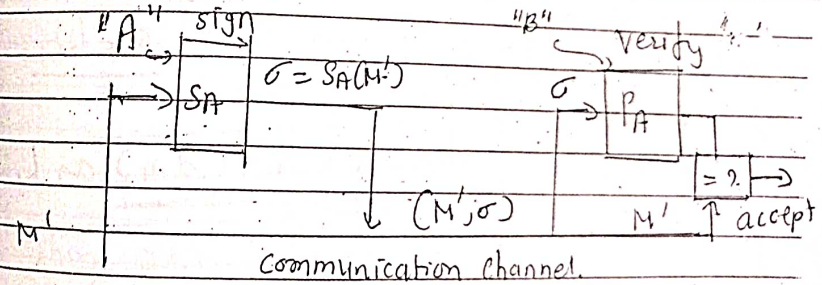


Fig: Digital signature in public key cryptosystem.

$S_A(\cdot)$
M

TSEC

1.

The RSA algorithm:

1. Select at random two large prime numbers p and q
2. Compute n by the equation $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, where

$$\phi(n) = (p-1)(q-1)$$

4. Compute d as the multiplicative inverse of e modulo $\phi(n)$

$$\therefore\ d = e^{-1} \text{ modulo } \phi(n).$$

5. Publish the pair $P = (e, n)$ as his RSA public key.
6. Keep secret the pair $S = (d, n)$ as his RSA secret key

1) The transformation of message M associated with public key $P = (e, n)$ is

$$P(M) = M^{e} \pmod{n}$$

2) The transformation of cipher text C associated with a secret key $S = (d, n)$ is

$$S(C) = C^{d} \pmod{n}.$$

TSEC

Problem 1. (OB-1996)

1.

Consider a RSA key set with $p = 11$, $q = 29$, $n = 319$ and $e = 3$. What value d should be used in the secret key? What is the encryption of $M = 200$? (10M)

Soln

Here $n = pq = 11 \times 29 = 319$

$\phi(n) = (p-1)(q-1) = 10 \times 28 = 280$

Now

$\therefore d = e^{-1} \bmod \phi(n)$

$\therefore de = 1 \bmod \phi(n)$

$de = 1 \bmod 280$

$(d \times e) \bmod 280 = 1$

~~$\therefore d \times e = 281$~~

~~$\therefore d = \frac{281}{e} = \frac{281}{3} = 93.6$~~

$de = 1 + k\,\phi(n)$
$de = 1 + k(280)$
$d = \frac{1 + k(280)}{3}$
for $k = 2$
we get $d = 187$
↳ find the first value of k for which remainder is 0.

Thus $d = 93$; $e = 3$, $p = 11$, $q = 29$

$P(M) = M^{e} \pmod{n}$

$P(200) = (200)^{3} \pmod{319}$

$= 8000000 \,\%\, 319$

$= 118$

$8000000 \,\%\, 319$
$= 25078.3699$
so to get ans
0.3699×319
$= 117.99$

decryption

If we receive $e = 118$, objective is to obtain original plain text M.

$$S(C) = C^{987} \pmod{n}$$

$$S(118) = (118)^{987} \pmod{319}$$

Problem :- BB-8392

$p = 17$, $q = 23$, $n = 391$, $e = 9$

(1) $d = ?$

(2) what is the encryption of the message $m = 100$. ?

Soln

$n = p \times q = 17 \times 23 = 391$

problem 3:-

$p = 47$ $q = 59$ $d = 157$, $e = ?$

Find C for $M = 65$

Soln

$n = p \times q = 47 \times 59 = 1773$

$\phi(n) = (p-1)(q-1) = 46 \times 58 = 2668$

$de = 1 \bmod \phi(n)$ $\quad de = 1 + k\phi(n)$

$de = 1 \bmod (2668)$ $\quad \therefore e = \frac{1 + k\phi(n)}{d}$

$(157 \times e) \bmod 2668 = 1$ $\quad$ for $k = 1$

$e = 17$ $\quad e = \frac{1 + (1)(2668)}{157}$

$e = 17$

$C = (M)^e \bmod n$

$= (65)^e \bmod 1773$

$= (65)^{17} \bmod 1773$

$n = 2773$ $\quad k = 1$

$\phi(n) = 2668$ $\quad e = 17$

$de = e^{-1} \text{ modulo } \phi(n)$ $\quad P(65) = m^e \bmod n$

$de = 1 \bmod \phi(n)$ $\quad P(65) = (65)^{17} \bmod 2773$

$de = 1 + k \bmod (2668)$ $\quad 65 \cdot 6 \times 10^{30}$

$e = \frac{1 + k \bmod (2668)}{d}$ $\quad q \quad \therefore 0.38 \times 10^{27}$

Complexity of RSA:

Assume that the public key (e,n) and secret key (d,n) satisfy
$d, e = O(1)$ & $\lg d \leq \beta$ & $\lg n \leq \beta$

Then applying a public key requires $O(1)$ modular multiplications and uses $O(\beta^2)$ bit operations.

Applying a secret key requires $O(\beta)$ modular multiplications, using $O(\beta^3)$ bit operations.

Complexity of Recursive functions:

- When an algorithm contains a recursive call to itself, its running time is often described by a recurrence.
- A recurrance is an equation or inequality that describes a function in terms of its value on smaller inputs.

for e.g. Worst case running time of the Merge-sort can be described as.

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

whose solution is $T(n) = \Theta(n \lg n)$.

There are basically three methods for solving recurrences

1) Substitution method
2) Recursion-tree method
3) Master method.

The Substitution method:

There are two steps to be followed

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

∴ The substitution method can be used to establish either upper or lower bounds on a recurrences.

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

we guess that the solution is

$T(n) = O(n \lg n)$.

Our method is to prove that

$T(n) \le cn \lg n$, $c > 0$.

We assume that the bound holds for $\lfloor n/2 \rfloor$,

i.e $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.

Substituting this into recurrence,

$$T(n) \le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n$$
$$\le cn \lg(n/2) + n$$
$$\le cn \lg n - cn \lg 2 + n$$
$$= cn \lg n - cn + n$$
$$\le cn \lg n$$

2] Recursion Tree Method.

Here, each node represents the cost of a single subproblem.

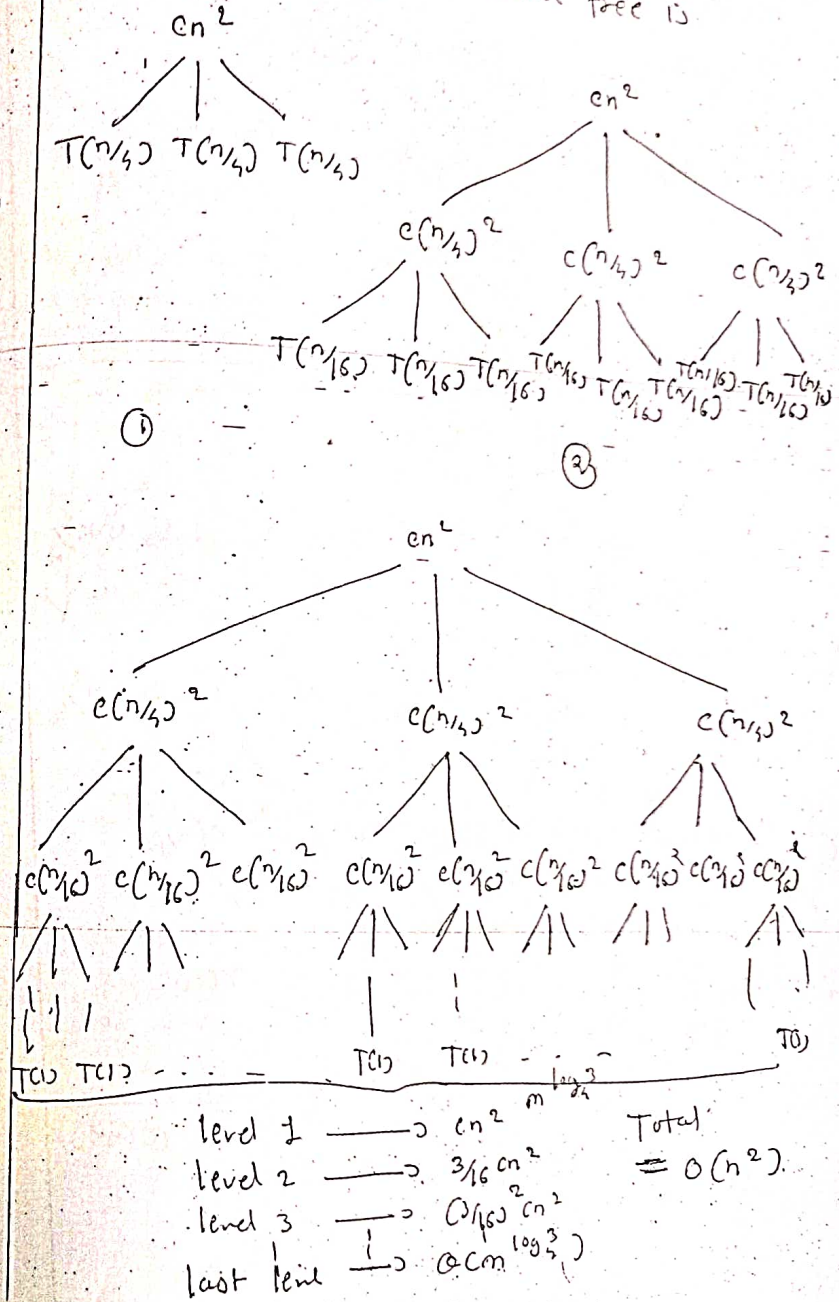
We sum the costs within each level of the tree to obtain a set of per level cost, and then we sum all the per level costs to determine the total cost of all levels of the recursion.

Tree Recursion is used to describe the running time of a divide & conquer algorithms.

eg Consider the recurrence

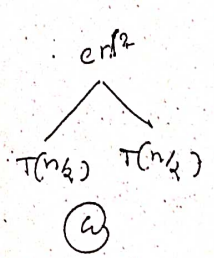
$T(n) = 3T(n/4) + \Theta(n^2)$

The corresponding recurrence tree is

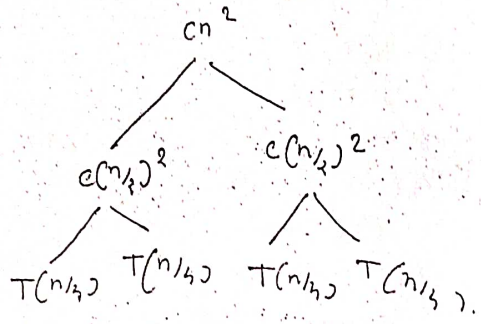


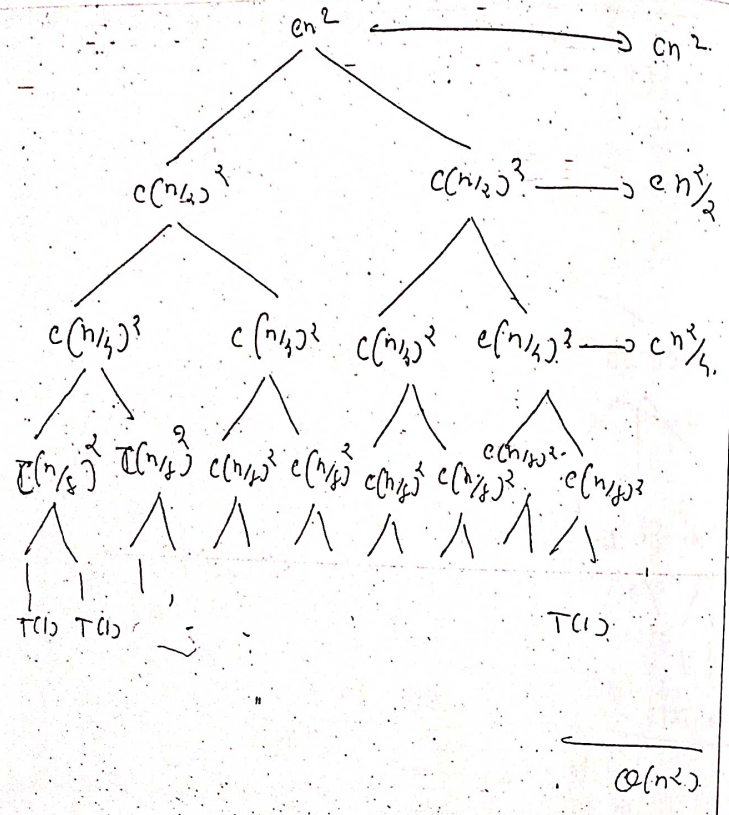
Example Construct a recursion tree for the recurrence

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

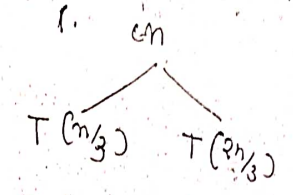


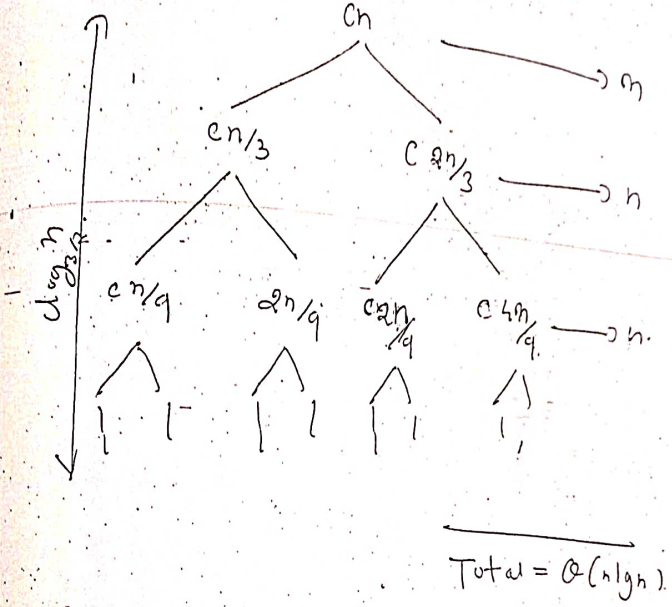
(a)





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





$$T(n) = n + n + n + \cdots + \log_{3/2} n \text{ times}$$

$$= O(n \lg n)$$

The master Method:

- The master method can be used to solve recurrances of the form.

$$T(n) = a\,T(n/b) + f(n) \quad \text{---(1)}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

- The master method requires memorization of 3 cases.
- The recurrance (1) describes the running time of an algorithm that devides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants.
- The a subproblems are solved recursively, each, in time $T(n/b)$.
- The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

The master Theorem:

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

$$T(n) = a\,T(n/b) + f(n).$$

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

The $T(n)$ can be bounded asymtotically as.

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

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

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

- In each of the three cases, we compare function $f(n)$ with the function $n^{\log_b^a}$.
- The solution to the ~~accusance~~ accurrance is determined by the larger of the two functions.

Examples

Solve the following recurrances using Master method.

1) $T(n) = 9T(n/3) + n$

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

$n^{\log_b^a} = n^{\log_3^9} = n^2$ $\quad n^{\log_3^{3^2}}$ $\quad n^{2-1}$

$\therefore f(n) = n^{(\log_b^a - \epsilon)}$ $\quad n^{2-1}$

$\therefore$ case 1 $\quad T(n) = \Theta(n^{\log_b^a}) = O(n^2)$ Ans

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

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

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

$\therefore f(n) = n^{\log_b^a} = 1$

Case - II $\quad T(n) = \Theta(n^{\log_b^a} \lg n)$

$= \Theta(1 \lg n)$

$= \Theta(\lg n)$

$n^{(1)} = n^{(2)-\epsilon}$ Ans

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

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

$n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793}) \quad 0.793 + 0.2 \quad 1 - 0.2$

$f(n) = \Omega\left(n^{(\log_4 3) + \epsilon}\right) \quad \epsilon = 0.2$

Case : 3 $\quad T(n) = \Theta(f(n))$

$= O(n \lg n)$.

4.) Paper kk - 8618

$T(n) = 9T(n/3) + n^3$

Soln $\quad a = 9 \quad b = 3 \quad f(n) = n^3 \quad n^{2+1}$

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

$\therefore f(n) = \Omega\left(n^{\log_3 9 + \epsilon}\right)$ for $\epsilon = 1$

$\therefore$ Case : 3 $\quad \therefore T(n) = \Theta(f(n))$

$= \Theta(n^3)$

Ans

5) $T(n) = 9T(n/3) + n^2$

$\therefore a = 9, \quad b = 3 \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

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

Case 2 $\quad T(n) = \Theta(n^{\log_b a} \lg n) = (n^2 \lg n)$.

6) paper code BN - 75722

Solve the recurrance using Master method.

(i) $T(n) = 9T(n/3) + n^2$

solved

(ii) $T(n) = 16T(n/4) + n$

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

$n^{\log_b a} = n^{\log_4 16} = n^2 = O(n^{2-1})$

$\therefore n^{(\log_b a) - \epsilon} = f(n)$ for $\epsilon = 1$

$\therefore$ Case - I $\quad \therefore T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Ans

(iii) $T(n) = 3T(n/4) + n \log n$

solved

(iv) $T(n) = 2 \cdot T(n/4) + \sqrt{n}$

$a = 2 \quad b = 4 \quad f(n) = \sqrt{n} = n^{0.5}$

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

Here $f(n) = \Theta(n^{\log_b a})$

Case : 2 $\quad T(n) = \Theta(n^{\log_b a} \lg n)$

$= \Theta(n^{0.5} \lg n) = \Theta(\sqrt{n} \lg n)$

Ans

7) page 130 Jahani

$T(n) = 9\,T(n/3) + 4n^6$, $n \geq 3$.

$f(n) = 4n^6 = \Theta(n^6)$ $\quad n^{2+4}$

Sol^n $a = 9$ $\quad b = 3$

$\therefore n^{\log_b a} = n^{\log_3 9} = n^2 = O(n^2)$

$\therefore n^{\log_b a + \epsilon} = f(n)$, $\epsilon = 4$

Case-III $\therefore T(n) = \Theta(f(n))$

$= \Theta(n^6)$.

---

8) $\therefore T(n) = 7\,T(n/2) + 18n^2$, $n \geq 2$.

Sol^n $a = 7$ $\quad b = 2$ $\quad f(n) = 18n^2 = O(n^2)$.

$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$ $\quad$ [crossed out] $n^{2.81-0.8}$

$\therefore n^{(\log_b a) - \epsilon} = f(n)$, $\epsilon = 0.81$

Case: I $\quad T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$

---

9) $T(n) = 4\,T(n/2) + n$. Kerman page: 4

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

$n^{\log_b a} = n^{\log_2 4} = n^2$ $\quad n^{2-1}$

$\therefore n^{\log_b a - \epsilon} = f(n)$ $\quad \epsilon = 1$

Case I $\quad T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$ Ans

---

10) $\therefore T(n) = 4\,T(n/2) + n^2$

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

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

Case: 2 $\quad T(n) = \Theta(n^{\log_b a} \lg n)$

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

---

11) $T(n) = 4\,T(n/2) + n^3$

Sol^n $a = 4$ $\quad b = 2$ $\quad f(n) = n^3$

$n^{\log_b a} = n^{\log_2 4} = n^2$ $\quad n^{2+1}$

$n^{\log_b a + \epsilon} = f(n)$, $\epsilon = 1$.

Case: 3 $\quad T(n) = \Theta(f(n)) = \Theta(n^3)$ Ans

Example. Karman page 75

The recurrance $T(n) = 7\,T(n/2) + n^2$ describes the running time of an algorithm A. A competing algorithm A' has a running time of $T'(n) = a\,T'(n/4) + n^2$. What is the largest integer value for a such that A' is faster than A?

Soln For A

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

$n^{\log_b^a} = n^{\log_2^7} = n^{2.81}$

$\Rightarrow f(n) = n^{\log_b^a - \epsilon}$

Case-I $T(n) = \Theta\left(n^{\log_b^a}\right) = \Theta(n^{2.81})$

Now A' should be faster than A means Running time of A' $\leq$ Running Time of A

$T'(n) \leq T(n)$

$n^{\log_4^a} \leq n^{\log_2^7}$

$n^{\log_4^a} \leq n^{2.81}$

$\log_4^a \leq 2.81$

$a \leq 4^{2.81}$

$\boxed{a \leq 49}$

Example. Substitution method.

1) paper Feb-2016

Use the substitution method to show that the following recurrance is $O(n)$.

$$T(n) = \begin{cases} 0 & \text{, if } n = 0,1,2,3 \\ T(n-4)+1 & \text{, if } n > 3 \end{cases}$$

Soln

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

$= T(n-8) + 1 + 1$

$= T(n-12) + 1 + 1 + 1$

$= T(n-16) + 1 + 1 + 1 + 1$

$= 1 + \cdots + 1 + 1 + 1 + 1$

$= O(n)$

Optimal search binary tree.

	0	1	2	3
key	10	12	16	21
freq	4	2	6	3

Creat BST "root of search min"

	0	1	2	3
0	4	8(0)	20(2)	26(2)
1		2	10(2)	16(2)
2			6	9(2)
3				3

Dynamic Programming: Imp

- Dynamic programming, like divide-and-conquer method, solves problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms partition the problem into independent subproblems, solves the subproblems recursively and then combine their solutions to solve the original problems.
- In contrast, dynamic programming is applicable when the subproblems are not independent, that is when the subproblems share subproblems.
- In this context, a divide-and-conquer method, does more work than necessary, repeatedly solves every subproblem. solving the common subproblems.
- A dynamic programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding a work of recomputing the answer everytime the subproblem is encountered.

Dynamic programming is typically applied to optimization problem. In such problem there can be many possible solutions. Each solution has a value and we wish to find solution with optimal value.

The development of a dynamic-programming algorithm can be divided into 4 steps

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a (bottom-up) fashion.
4. Construct an optimal solution from computed information.

Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism — we take a problem, split it up and solve the smaller problems that are created.

Dynamic prog. is a bottom-up mechanism — we solve all possible small problems and then combine them to obtain solutions for bigger problems.

Common characteristics of dynamic programming problems:

1. The problem can be divided into stages with a decision required at each stage. (stage / decision)
   for eg; in the shortest path problem, they were defined by the structure of the graph. The decision was where to go.
2. Each stage has a number of states associated with it. The states for shortest path problem was the node reached. (state)
3. The decision at one stage transforms one state into a state in the next stage. The decision of where to go next defined where we arrived in the next stage. (decision / transform)
4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem it was not necessary to know how we got to a node. (principle of optimality)
5. There exist a recursive relationship that identifies the optimal decision for stage j, given that stage $j+1$ has already been solved. (recursive)
6. The final stage must be solvable by itself. (final stage)

The big skill in the dynamic programming is to take a problem and determine stages and states so that all of the above holds.

Assembly-line scheduling (Manufacturing problem)

The Colonel Motors corporation produces automobile in a factory that has two assembly lines. An automobile chassis enters each assembly line, has parts added to it at a number of stations, and a finished auto exit at the end of ~~ea~~ the line.

- Each assembly line has n stations numbered $j = 1, 2, 3, \ldots, n$.
- Let $S_{i,j}$ denote the j^{th} station on line i where $i = 1$ or 2.
- Let $a_{i,j}$ denote the assembly time required at station $S_{i,j}$.
- There is also an entry time e_i for the chassis to enter assembly line i and an exit time x_i for the completed auto to exit line i.
- The time to go from one station to next station within the same assembly line is negligible.

[Normally, once a chasis enters an assembly line, it passes through that line only. Occasionally when a special rush order comes in, and the customer wants the automobile to be manufactured as quickly as possible. This time, the factory manager may switch the partially completed auto from one assembly line to another.]

Chassis

- Let $t_{i,j}$ denote the time to transfer a chassis away from assembly line i after having gone through station $S_{i,j}$, where $i = 1, 2$ and $j = 1, 2, \ldots, n-1$.
- The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time for one auto.

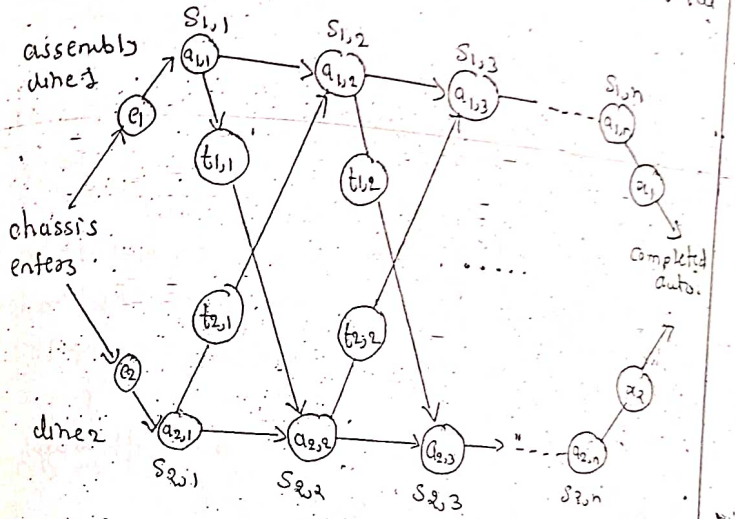


Fig: A manufacturing problem to find the fastest way through a factory for an auto.

Step: 1. The structure of the fastest way through the factory.

- Let us consider the fastest way for a chassis to get from the starting point through station $S_{1,j}$.

- If $j=1$, then is only one way that the chassis could have entered, and so it is easy to determine how long it takes to get through $S_{1,j}$. (for $j=1$).
- For $j=2,3,\ldots,n$, however, there are two choices:
  1). The chassis could have come from $S_{1,j-1}$ and then directly to $S_{i,j}$
  2) Alternatively, the chassis could have come from $S_{2,j-1}$ and then been transferred to station $S_{1,j}$. The transfer time is $t_{2,j-1}$.

[More generally, we use optimal substructure to show that we can construct an optimal solution to a problem from optimal solutions to subproblems. In this example, a fastest way through station $S_{1,j}$ is computed by taking a fastest way through station $j-1$ on either line 1 or 2.]

Thus, the fastest way through station $S_{1,j}$ is either

- the fastest way through $S_{1,j-1}$ and then directly through $S_{1,j}$ or.
- the fastest way through $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.

Similarly, a fastest way through station $S_{2,j}$ is either

- a fastest way through station $S_{2,j-1}$ and then directly to station $S_{2,j}$ or
- a fastest way through station $S_{1,j-1}$, a transfer from line 1 to 2, and then through station $S_{2,j}$.

step: 2 A recursive solution.

A second step to dynamic-programming is to define the value of an optimal solution recursively in terms of the optimal solutions to subproblems.

Let $f_i[j]$ denote the fastest possible time to get a chassis from the starting point through station $S_{i,j}$.

Let f^* denote the fastest time to get a chassis all the way through the factory.

so $f^* = \min(f_1[n]+x_1, f_2[n]+x_2)$

→ for $j=1$, we can calculate $f_1[1]$ and $f_2[1]$.

$$\left.\begin{aligned} f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{aligned}\right\} \text{——①}$$

→ for $j=2,3,\ldots,n$, we have to calculate $f_1[j]$ and $f_2[j]$.

Now, we know that, the fastest way to $S_{1,j}$ is either through a fastest way through $S_{1,j-1}$, and then directly to $S_{1,j}$ or a fastest way through station $S_{2,j-1}$, and a transferred time from line 2 to line 1, and then through station $S_{1,j}$.

So we can calculate

$$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$$

for $j = 2, 3, \ldots, n$

Similarly we have,

$$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$$

for $j = 2, 3, \ldots, n$.

Combining the above equations with eqn (1),

$$f_1[j] = e_1 + a_{1,1} + \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$$

$$f_2[j] = e_2 + a_{2,1} + \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

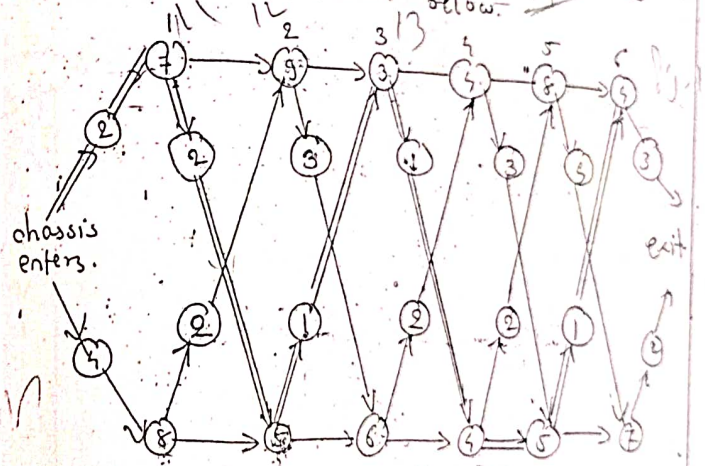
Step 3: Computing the fastest time

The $f_i[j]$ values give the values of optimal solutions to subproblems.

→ Let us now define, $l_i[j]$ to be the line no. 1 or 2, whose station $j-1$ is used in a fastest way through station $S_{i,j}$. Here $i = 1, 2$ and $j = 2, 3, 4, \ldots, n$

→ We also define, l^* to be the line whose station n is used in a fastest way through the entire factory.

For example, An instance of an assembly line problem is shown below:



Calculate $f_1[j]$ and $f_2[j]$ for each station on line 1 and 2.

	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37
$l_1[j]$		1	2	1	1	2
$l_2[j]$		1	2	1	2	2

$f^* = \min(f_1[6], f_2[6])$

$= 35$

$l^* = 1$

Now to find out f^* ~~we backtrack~~,

$$f^* = \min \{ f_1[6] + x_1 , f_2[6] + x_2 \}$$

$$= \min \{ \underset{L_1}{35+3} , \underset{L_2}{37+2} \}$$

$f^* = 38$ from line 1., Thus $l^* = 1$

Using the value l^* and $l_i[j]$, we can trace the fastest way through the factory. We backtrack.

$S_{1,6} \rightarrow S_{2,5} \rightarrow S_{2,4} \rightarrow S_{1,3} \rightarrow S_{2,2} \rightarrow S_{1,1}$

so the optimal path is

$S_{1,1} \rightarrow S_{2,2} \rightarrow S_{1,3} \rightarrow S_{2,4} \rightarrow S_{2,5} \rightarrow S_{1,6}$

Step 3: Computing the fastest times.

At this point we can write a recursive algorithm to compute the fastest way through the factory.

```
Fastest-way (a, t, e, x, n)
    f1[1] ← e1 + a1,1
    f2[1] ← e2 + a2,1
for j ← 2 to n
  do if f1[j-1] + a1,j ≤ f2[j-1] + t2,j-1 + a2,j
     then f1[j] ← f1[j-1] + a1,j
          l1[j] ← 1
     else
          f1[j] ← f2[j-1] + t2,j-1 + a2,j
          l1[j] ← 2

     if f2[j-1] + a2,j ≤ f1[j-1] + t1,j-1 + a2,j
     then f2[j] = f2[j-1] + a2,j
     else      l2[j] = 2
          f2[j] = f1[j-1] + t1,j-1 + a2,j
               l2[j] = 1

if  f1[n] + x1 ≤ f2[n] + x2
then  f* = f1[n] + x1
      l* = 1
else
      f* = f2[n] + x2
      l* = 2
```

Step 4: Constructing the fastest way through the factory:

Having computed $f_1[j]$, $f_2[j]$, f^* and l^* we can construct sequence of stations used in the fastest way.

```
Print_stations (d, n)
    i ← l*
    print "line" i, "station" n
    for j ← n downto 2
        do i ← di[j]
           print "line" i, "station" j-1
```

Time Complexity:

for fastest_way ( ) → $\Theta(n)$

for Print_station ( ) → $\Theta(n)$.

Huffman codes:

- Data can be encoded efficiently using Huffman codes. It is widely used and very effective technique for compressing data; saving of 20% to 90%, depending upon the characteristics of file being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurances of each character to build up an optimal way of representing each character as a binary string.

(i) Fixed Length code:

Each letter is represented by an equal no. of bits.

e.g:

a	000	e	100
b	001	f	101
c	010	g	110
d	011	h	111

(ii) Variable length code:

performs better than fixed-length code, by giving frequent characters short code word and infrequent characters long code words.

e.g:

a	0
b	101
c	110
d	111
e	1101
f	1100

no. of bits = $(45\times1 + 13\times3 + 12\times3 + 16\times3 + 9\times4 + 5\times4)\times1000$ = 2.24×10^5 bits

for: a b c d e f
45 13 12 16 9 5

ix codes:

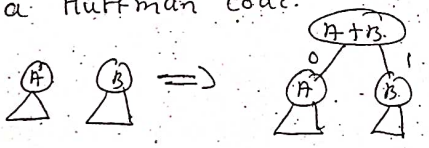
The prefixes of an encoding of one character must not be equal to a complete encoding of another character.

e.g: 1100 and 11001 are not valid codes b'cos 1100 is a prefix of 11001.

- This constraint is called prefix constraint.
- Codes in which no code word is a prefix of any other code word are called prefix codes.
- Huffman is a prefix code.

Greedy algo for constructing Huffman code.

Huffman invented a greedy algo. that constructs an optimal prefix code called a Huffman code.



The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.

→ Let C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$.

→ A priority Q, keyed on f, is used to identify the two least-frequent objects to merge together.

- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN (C)
n ← |C|
Q ← C
for i ← 1 to n-1
    do z ← Allocate-Node()
       x ← left[z] ← EXTRACT-MIN(Q)
       y ← right[z] ← EXTRACT-MIN(Q)
       f[z] ← f[x] + f[y]
       INSERT (Q, z)
return EXTRACT-MIN(Q).
```

Complexity:

Here we assume that the Q is implemented as a binary heap.

$O(n \lg n)$.

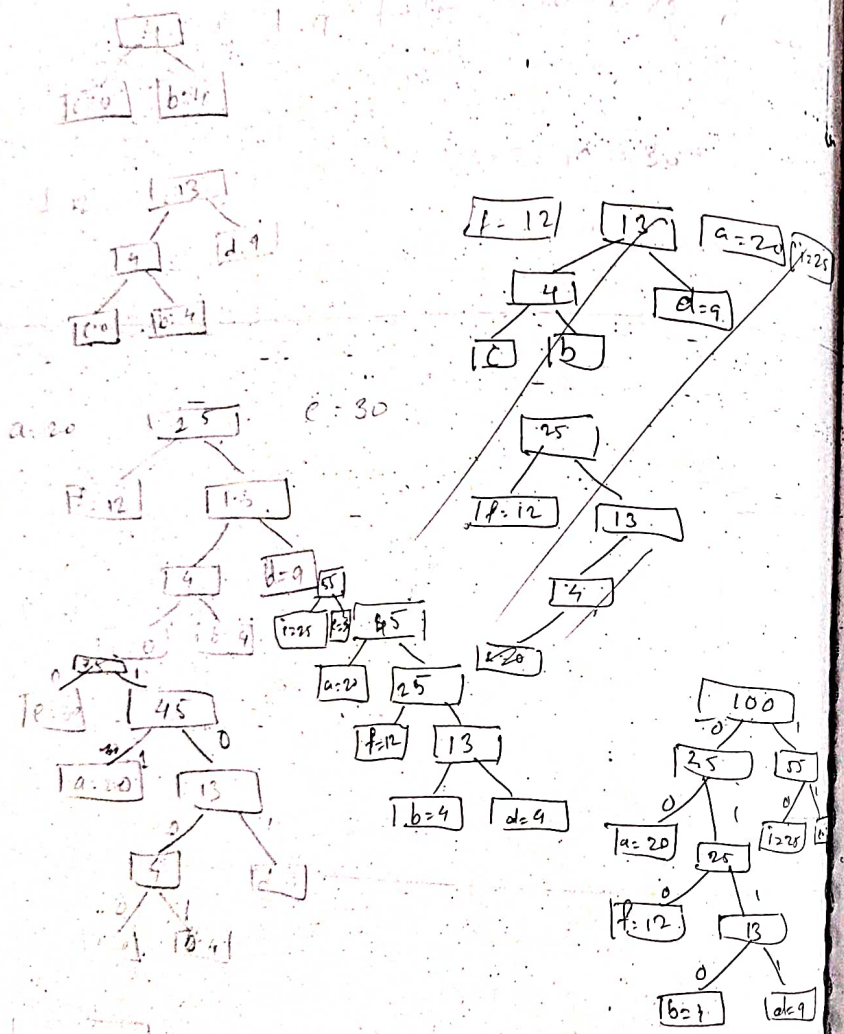
Applications

Widely used in data communication as a data compression technique.

BB-113295

Generate Huffman code for

a=20 b=4 c=0 d=9 f=12 i=25 e=30



e = 0
d = 100
i = 101
a = 1000
b = 1001

a = 00
b = 0110
c = —
d = 0111
f = 010
i = 10
e = 11

Bitonic Sorting:

Q-1 Explain bitonic sorting.

Q-2 What is bitonic sorting sequence? Using a diagram show the effect of a half cleaner on a sequence of 8 elements.

00011110.

Ans Bitonic sequence: "A sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically-decreasing."

for eg; the sequence $\langle 1,4,6,8,3,2 \rangle$, $\langle 6,9,4,2,3,5 \rangle$ and $\langle 9,8,3,2,4,6 \rangle$ are all bitonic.

- The zero-one sequence that are bitonic have a simple structure.

They have some form of $0^i 1^j 0^k$ or the form $1^i 0^j 1^k$, for some $i,j,k \geq 0$

Note that the sequence that is either monotonically increasing or monotonically decreasing is also bitonic."

Bitonic Sorter: The bitonic sorter is a comparison network that sorts bitonic sequences of 0's and 1's.

The half-cleaner:

The bitonic sorter is composed of several stages, each of which is called a half-cleaner.

Each half cleaner is a depth of 1. in which input line i is compared with line $i+n/2$ for $i=1,2,\dots,n/2$

- When a bitonic sequence is supplied as input to a half-cleaner, the half-cleaner produces an output sequence in which smaller values are in the top half, larger values are in the bottom half, and both halves are bitonic.

eg ①

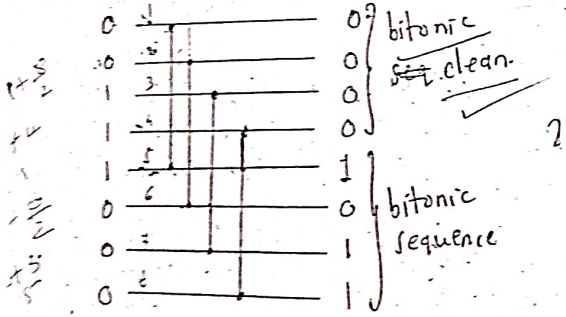
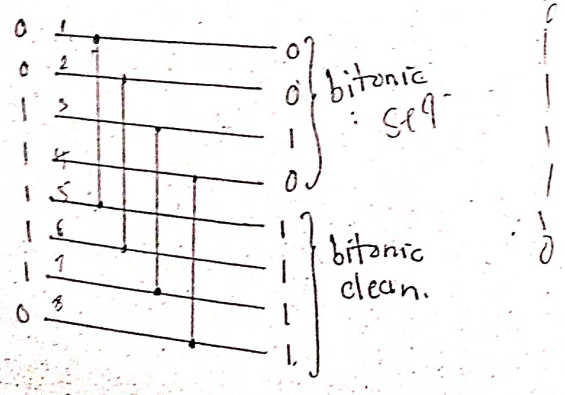


fig: o/p of half-cleaner

eg ②



- Note, that atleast one of the halves is a clean (consisting of all 0's or all 1's).

The bitonic sorter:

By recursively combining half-cleaners, we can build a bitonic sorter which is a network that sorts bitonic sequences.

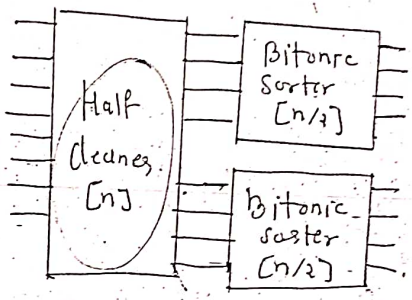
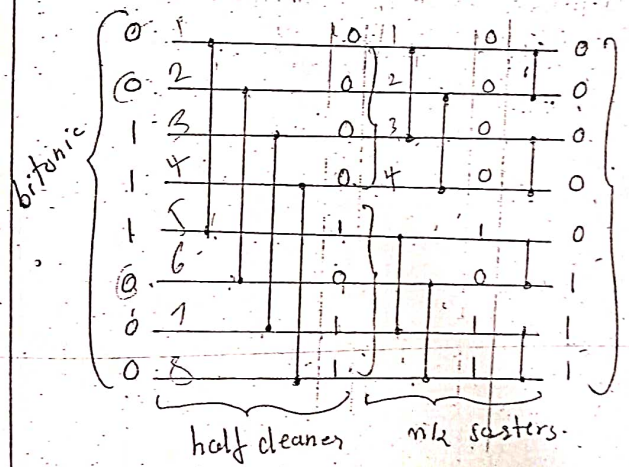


fig: bitonic sorter



The first stage of bitonic sorter[n] contains the half cleaner[n], which produces two bitonic sequences of $n/2$, such that every element in the top half is atleast as small as every element in bottom half. We can complete the sort by using two copies of bitonic-sorter[$n/2$] to sort the two halves recursively.

Depth of bitonic sorter:

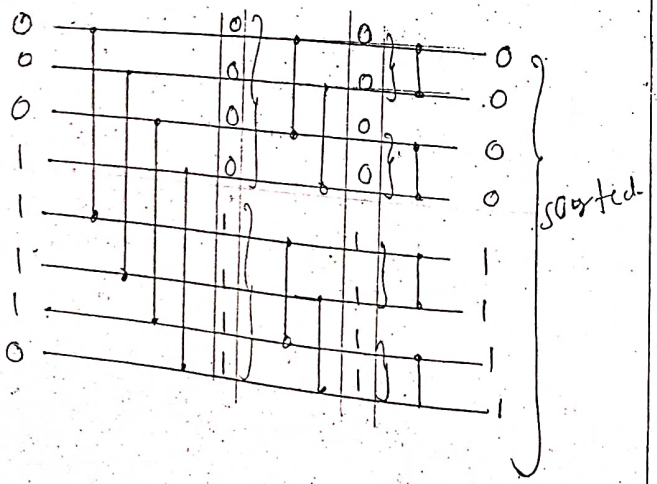
The depth $D(n)$ of bitonic-sorter[n] is given by,

$$D(n) = \begin{cases} 0 & \text{if } n=1 \\ D(n/2)+1 & \text{if } n = 2^k \text{ and } k \geq 1. \end{cases}$$

whose solution is $\boxed{D(n) = \log n}$

Sort the following sequence:

00011110.



Merging network:

These are networks that can merge two sorted input sequences into one sorted o/p sequence.

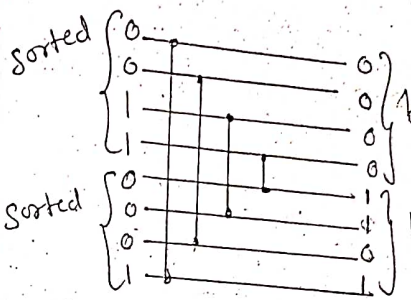


Fig
first stage of merger.

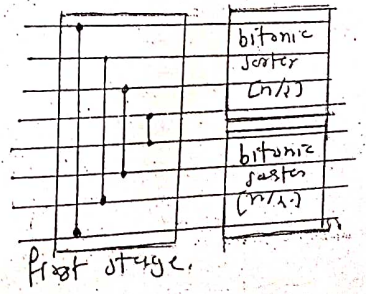
- Given the two sorted sequences, if we reverse the order of the second sequence and then concate the two sequences, the resulting sequence is bitonic. The merging n/w is based on this principle.

e.g. $x = 0011$ $y = 0001$

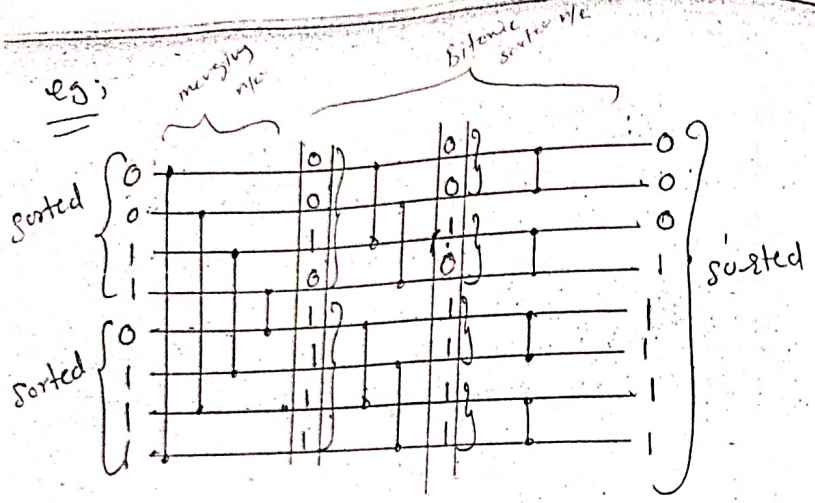
$y^R = 1000$

Concate x and y = 00111000
bitonic

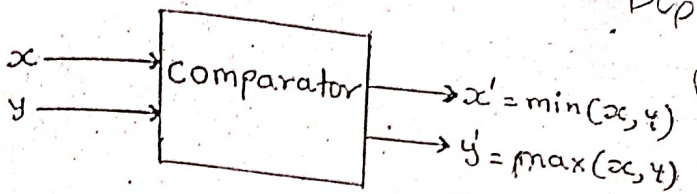
- We construct the merger by modifying first half-cleaner of the bitonic sorter.



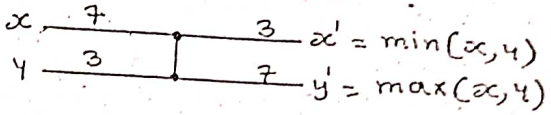
eg;



Fig; A n/w that merges two sorted input sequences.



A comparator with inputs x & y and outputs x' & y'.



Same comparator drawn as a single vertical line..

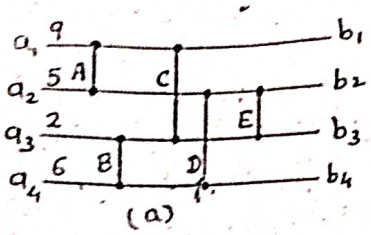
We shall assume that each comparator operates in O(1) time.

A wire transmits a value from place to place.

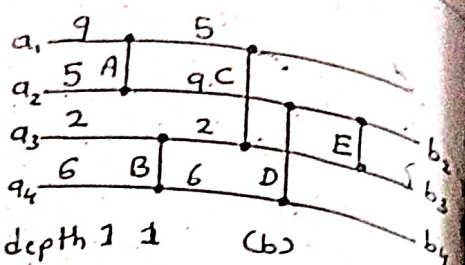
Wires can connect the output of one comparator to the input of another,

Otherwise they are either n/w input wires or network output wires.

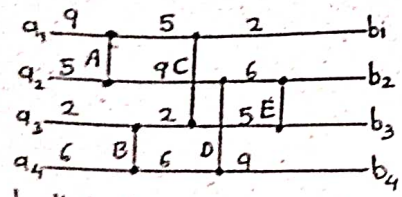
We will assume that a comparison network contains n input wires $a_1, a_2, \ldots, a_n$ through which the values to be sorted enter the n/w & n o/p wires $b_1, b_2, \ldots, b_n$ which produces the results computed by n/w.



(a)

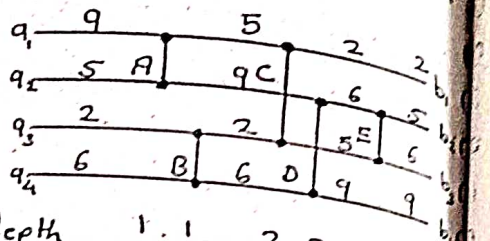


depth 1 1 (b)



depth 1 1 2 2

(c)



depth 1 1 2 2 3

(d)

Each comparator produces o/p values only when both of its input values are available to it.

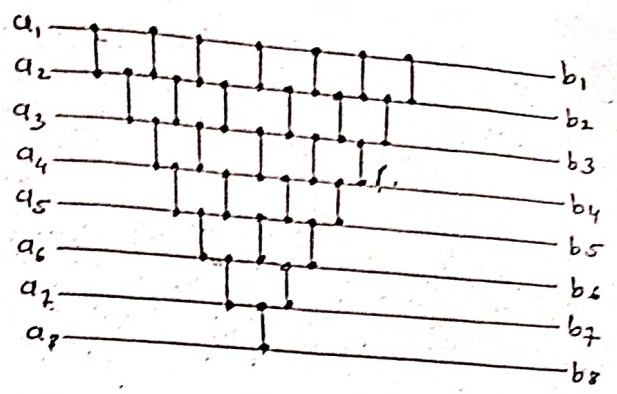
In the above figure 'a' a suppose that sequence (9, 5, 2, 6) appear on the i/p wires at time

Assume that each comparator requires one time unit to compute o/p values.

Comparators A & B produces their o/p's at time 1 as shown in figure (b).

Depth:- An i/p wire of a comparison N/w has depth 0.

Now if a comparator has two input wires with depths d_x & d_y then its output wires have depth $\max(d_x, d_y) + 1$



A sorting network based on insertion sort.

A sorting N/w is a comparison network for which o/p sequence is monotonically increasing for every i/p sequence.

Zero-one principle:- says that if a sorting network works correctly when each input is drawn from the set $\{0, 1\}$ then it works correctly on arbitrary input numbers.

A bitonic sorting network

This is a comparison network that can sort any bitonic sequence.

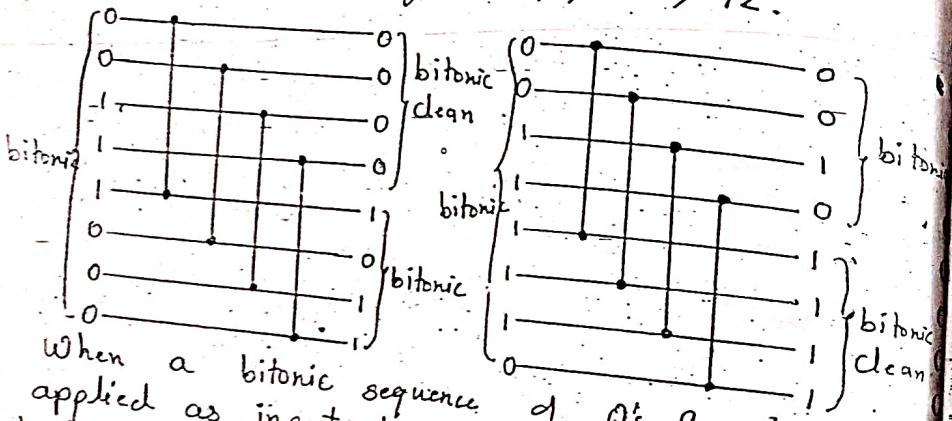
Bitonic sequence:- is a sequence that increases monotonically and then monotonically decreases or it can be monotonically decreasing and then monotonically increasing sequence.

Bitonic zero-one seq^n : $0^i 1^j 0^k$ or $1^i 0^j 1^k$

The half-cleaner:-

A bitonic sorter is composed of several stages each is called half cleaner.

Each half cleaner is a comparison n/w of depth 1 in which line i is compared with line $i+n/2$ for $i=1,2,\ldots,n/2$.

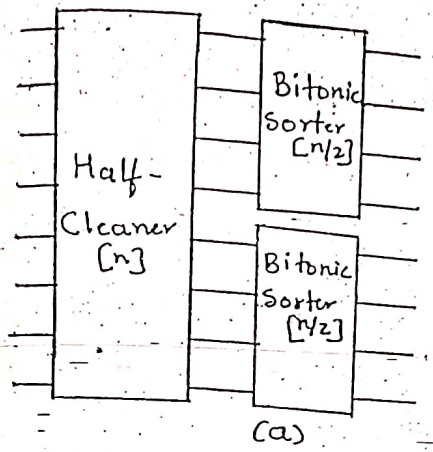


When a bitonic sequence of 0's & 1's is applied as input to a half cleaner, the half-cleaner produces an output sequence in which smaller values are in the top half, larger values are in the bottom half & both halves are bitonic.

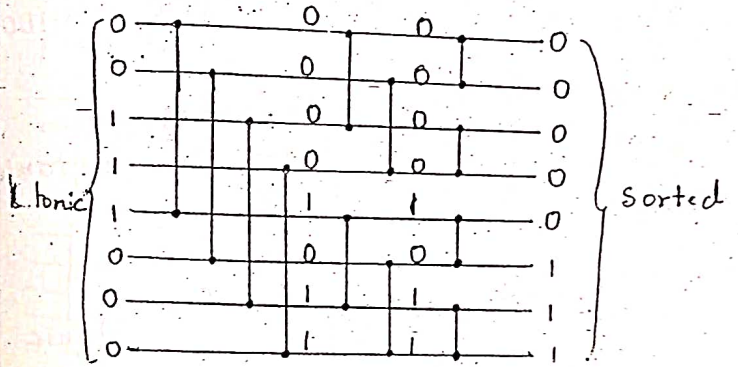
In fact at least one of the halves is clean - consisting of either all 0's or all 1's. hence it is called "half-cleaner".

The bitonic Sorter

By recursively combining half-cleaners we can build a bitonic sorter, which is a network that sorts bitonic sequences.



(a)



(b)

The comparison network Bitonic-Sorter[n] shown here for n=8.

A Merging network.

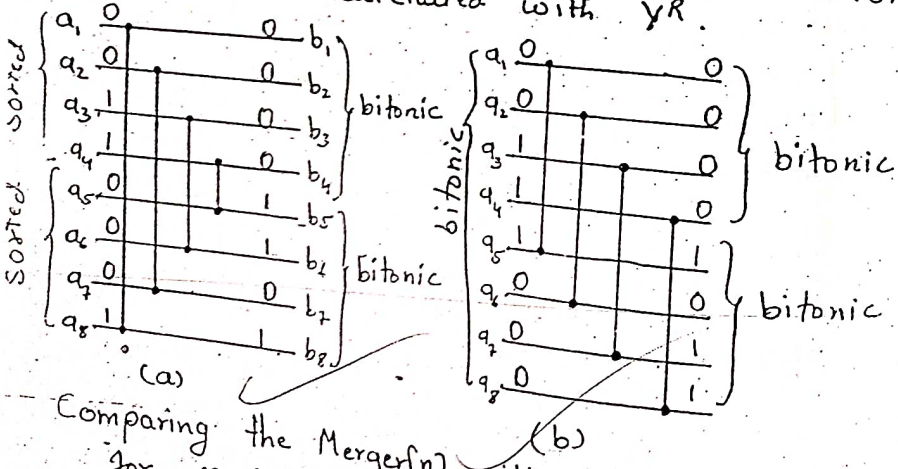
Our Sorting network will be constructed from merging network.

Merging network can merge two sorted input sequences into one sorted output sequence.

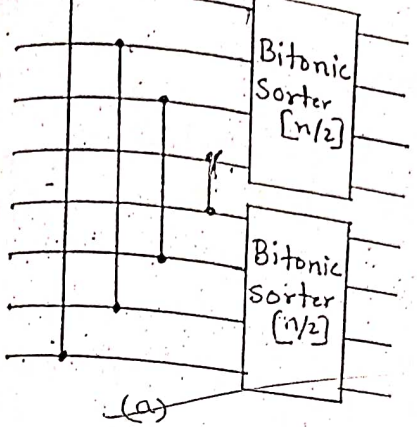
Ex. Given two sorted sequences X = 00000111 & Y = 00001111 we reverse Y to get Y^R = 11110000.

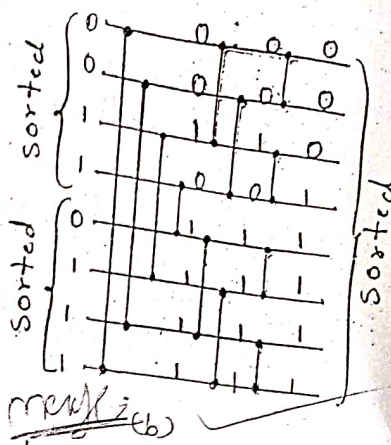
Concatenating X & Y^R yields 0000011111110000 which is bitonic.

Thus to merge the two input sequences X & Y, it suffices to perform the bitonic sort on X concatenated with Y^R.



Comparing the Merger[n] with Half-Cleaner[n] for n=8

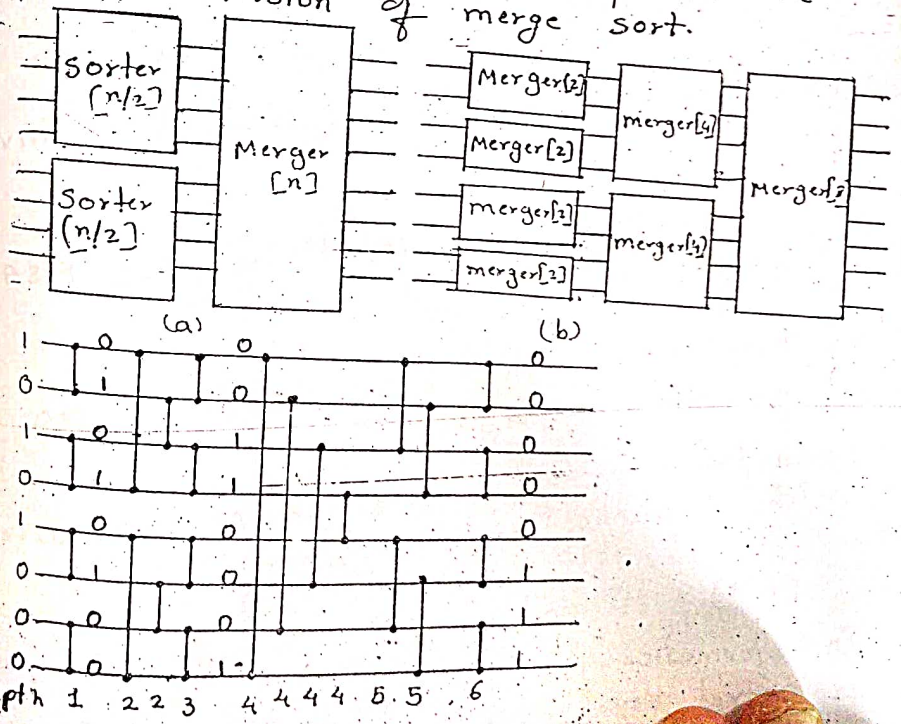




A network that merges two sorted input sequences into one ~~output~~ sorted o/p sequence

A Sorting network

The sorting network Sorter[n] uses the merging network to implement a parallel version of merge sort.



RSA cryptosystem

In RSA public-key cryptosystem a partn... creates his public & secret keys with the following procedure.

1.) Select at random two large prime numbe... p & q such that $p \neq q$.
The primes p & q might be say 512 bits each.

2.) Compute n by the equation $n = pq$.

3.) Select a small odd integer e that is relatively prime to $\phi(n)$,
$\phi(n) = (p-1)(q-1)$.

4.) Compute d as the multiplicative inverse of e, modulo $\phi(n)$.

5.) Publish the pair $P = (e, n)$ as his RSA public key

6.) Keep secret the pairs $S = (d, n)$ as his RSA secret key.

Transformation of a message M associated with a public key $P = (e, n)$ is

$$P(M) = M^e \bmod n$$

Transformation of a ciphertext C associated with a secret key $S = (d, n)$ is

$$S(C) = C^d \bmod n.$$

Example

$p = 3$, $q = 11$, $n = 33$

$\phi(n) = (p-1) \times (q-1) = 20$

$d = 7$, $M = 19$ (Message)

$7e = 1 \bmod 20$

$\therefore (7e - 1)/20 = x$

$\therefore 7e - 1 = 20x$

$e = \frac{20x + 1}{7}$

$= \frac{20+1}{7}$ if $x = 1$

$= 21/7 = 3$

$\therefore e = 3$

$ed \bmod \phi(n) = 1$

$ed \equiv 1 \bmod \phi(n)$

M	M^3	$C = M^3 \bmod 33$	C^7	$C^7 \bmod 33$
19	6859	28	$(28)^7$	19
			$= 13492985 12$	

↑ encryption of message M

↑ decryption of ciphertext to message M.

correspondence bet^n

eq^n mod n_i such that $i = 1$ to k

eq^n mod n such that $n = (n_1 \times n_2 \times \dots n_k)$

$n = (n_1 \times n_2 \times n_3 \times \dots \times n_k)$ such that $(n_1, n_2, n_3, \dots)$ are relatively prime pairs

$a \longleftrightarrow (a_1, a_2, a_3, \dots a_k)$

$b \longleftrightarrow (b_1, b_2, b_3 \dots b_k)$

So

if $(a+b) \bmod n \equiv (a_1+b_1) \bmod n_1, (a_2+b_2) \bmod n_2$

$(a-b) \bmod n \equiv (a_1-b_1) \bmod n_1, (a_2-b_2) \bmod n_2$

$(a_k-b_k) \bmod n_k$

$(a \cdot b) \bmod n \equiv a_1 b_1 \pmod{n_1} (a_2 b_2) \bmod n_2 \dots (a_k b_k) \bmod n_k$

Thus if $(n_1, n_2, n_3 \dots n_k)$ are pairwise prime then

$x \equiv a_1 \bmod n_1$

$x \equiv a_2 \bmod n_2$ & so on for all $(a_1$ to $a_k) \& (n_1$ to $n_k)$

Reverse so^n modulo n.

Chinese Remainder Theorem:

"Provides correspondence between a system of equations modulo a set of pairwise relatively prime moduli and an equation modulo their product"

Theorem:

Let $n = n_1 n_2 \dots n_k$ when n_i are pairwise relatively prime.

→ Consider the correspondence

$$a \longleftrightarrow (a_1, a_2, \dots, a_k) \quad \text{---} (1)$$

when $a \in Z_n$ and $a_i \in Z_{n_i}$ and

$a_i \equiv a \bmod n_i$ for $i = 1, \dots k$

Then mapping (1) is a one-to-one correspondence between Z_n and the cartesian product $Z_{n_1} \times Z_{n_2} \times \dots \times Z_{n_k}$

→ Operations performed on the elements of Z_n can be equivalently performed on the corresponding k-tuples by performing the operations independently in each co-ordinate positions in the appropriate system.

That is, if

$a \longleftrightarrow (a_1, a_2, \dots, a_k)$,

$b \longleftrightarrow (b_1, b_2, \dots, b_k)$

Then,

$(a+b) \bmod n \leftrightarrow ((a+b_1) \bmod n_1, \ldots, (a_k+b_k) \bmod n_k)$,

$(a-b) \bmod n \leftrightarrow ((a_1-b_1) \bmod n_1, \ldots, (a_k-b_k) \bmod n_k)$;

$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k)$.

Thus, if $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and

$$n = n_1 n_2 n_3 \ldots n_k$$

Then for any integers $a_1, a_2, \ldots, a_k$, the set of simultaneous equations

$$x \equiv a_i \pmod{n_i} \quad \text{for } i = 1, 2, \ldots k$$

has a unique solution modulo n for the unknown x.

Example

If $n_1, n_2, \ldots n_k$ are pairwise relatively prime and $n = n_1 n_2 \ldots n_k$ then for all integers x and a.

$$x \equiv a \pmod{n_i} \quad \text{for } i = 1, 2, \ldots k$$

if and only if,

$$x \equiv a \pmod{n}.$$

Suppose we are given two equations.

$$a \equiv 2 \pmod 5$$
$$a \equiv 3 \pmod{13}$$

so that $a_1 = 2$, $n_1 = 5$

$a_2 = 3$, $n_2 = 13$ and $n = n_1 \times n_2 = 5 \times 13 = 65$

and we wish to compute $a \bmod 65$, since $n = 65$.

→ Because

$$13^{-1} \equiv 2 \pmod 5$$

and $5^{-1} \equiv 8 \pmod{13}$.

→ we have,

$$c_1 = 13\,(2 \bmod 5) = 26$$
$$c_2 = 5\,(8 \bmod 13) = 40.$$

65 | 80

→
$$a \equiv a_1 \times c_1 + a_2 \times c_2 \pmod{65}$$
$$= 2 \times 26 + 3 \times 40 \pmod{65}$$
$$= 42 \pmod{65}.$$

Table of chinese remainder for $n = 65$.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Since $c_1 = 26$, moving down in above table increases a by 26, since $c_2 = 40$, moving right will increase a by 40.

Here $n = 65$ where $n_1 = 5$, $n_2 = 13$

Example Generate table for 33.

$n = 33$ $m_1 = 3$, $n_2 = 11$

	0	1	2	3	4	5	6	7	8	9	10
0	0	12	24	3	15	27	6	18	30	9	21
1	22	1	13	25	4	16	28	7	19	31	10
2	11	23	2	14	26	5	17	29	8	20	32

$x \bmod 3 = 0$ 12 8.
$x \bmod 11 = 1$

$x \bmod 3 = 1$
$x \bmod 11 = 0$ 22

Example- Find all solutions to the equations.

$x \equiv 4 \pmod 5$ and

$x \equiv 5 \pmod{11}$.

Soln Here $n_1 = 5$, $n_2 = 11$ $n = n_1 n_2 = 55$

We have to find x such that

$x \,\%\, 5 = 4$ and $x \,\%\, 11 = 5$

	0	1	2	3	4	5	6	7	8	9	10
0	0	45	35	25	15	5	50	40	30	20	10
1	11	1	46	36	26	16	6	51	41	31	21
2	22	12	2	47	37	27	17	7	52	42	32
3	33	23	13	3	48	38	28	18	8	53	43
4	44	34	24	14	4	49	39	29	19	9	54

$11^{-1} = 4 \bmod 5$ { $11 \bmod 5 = 1 \Rightarrow (5-1) = 4$

$5^{-1} = 9 \bmod 11$ { $5 \bmod 11 = 5 \Rightarrow (11-5) = 6$

$c_1 = n_2 (4 \bmod 5) = 11 (4 \bmod 5) = 44$

$c_2 = n_1 (6 \bmod 11) = 5 (6) = 30$

$a = (a_1 \times c_1 + a_2 \times c_2) \bmod n$

$= (4 \times 44) + (5 \times 30) \bmod 55$

$= 51$

Example for $n_0 = 55$ 3

$n_1 = 5$ $n_2 = 11$ $n = 11 \times 5 = 55$

Given two equations

$a \equiv 2 \pmod 5$

$a \equiv 3 \pmod{11}$

Since

$11^{-1} = 1 \bmod 5$ $\frac{1 + k(\cdot)}{a}$

$5^{-1} = 9 \bmod 11$

we have

$c_1 = 11 \times (1 \bmod 5) = 11$

$c_2 = 5 \times (9 \bmod 11) = 45$

The following table shows is drawn for $n = 55$

	0	1	2	3	4	5	6	7	8	9	10
0	0	45	35	25	15	5	50	40	30	20	10
1	11	1	46	36	26	16	6	51	41	31	21
2	22	12	2	47	37	27	17	7	52	42	32
3	33	23	13	3	48	38	28	18	8	53	43
4	44	34	24	14	4	49	39	29	19	9	54

Note that since $c_1 = 11$, moving down every row in a table increases a by 11 Since $c_2 = 45$, moving right in a table increases a by 45.

Hence $n = 65$ where $n_1 = 5$, $n_2 = 13$

Example Generate table for 33.

$n = 33$ $n_1 = 3$, $n_2 = 11$

	0	1	2	3	4	5	6	7	8	9	10
0	0	12	24	3	15	27	6	18	30	9	21
1	22	1	13	25		16	28	7	19	31	10
2	11	23	2	14	26	5	17	29		20	32

$x \bmod 3 = 0 \; 1 \; 2$

$x \bmod 11 = 1$

$x \bmod 3 = 1$

$x \bmod 11 = 0$ 22

Example - Find all solutions to the equations.

$x \equiv 4 \pmod 5$ and

$x \equiv 5 \pmod{11}$.

Soln Here $n_1 = 5$, $n_2 = 11$ $n = n_1 n_2 = 55$

We have to find x such that

$x \,\%\, 5 = 4$ and $x \,\%\, 11 = 5$

	0	1	2	3	4	5	6	7	8	9	10
0	0	45	35	25	15	5	50	40	30	20	10
1	11	1	46	36	26	16	6	51	41	31	21
2	22	12	2	47	37	27	17	7	52	42	32
3	33	23	13	3	48	38	28	18	8	53	43
4	44	34	24	14	4	49	39	29	19	9	54

$11^{-1} = 1 \bmod 5$ { $11 \bmod 5 = 1 \Rightarrow (1 \times (5-1)) = 4$

$5^{-1} = 9 \bmod 11$ { $5 \bmod 11 = 5 \Rightarrow (11-5) = 6$

$c_1 = n_2 (4 \bmod 5) = 11 (4 \bmod 5) = 44$

$c_2 = n_1 (6 \bmod 11) = 5 (6) = 30$

$a = (a_1 \times c_1 + a_2 \times c_2) \bmod n$

$= (4 \times 44) + (5 \times 30) \bmod 55$

$= 51$

Example for $n = 55$

$n_1 = 5$ $n_2 = 11$ $n = 11 \times 5 = 55$

Given two equations

$a \equiv 2 \pmod 5$

$a \equiv 3 \pmod{11}$

Since

$11^{-1} = 1 \bmod 5$ $\quad \frac{1 + k(n)}{a}$

$5^{-1} = 9 \bmod 11$

we have

$c_1 = 11 \times (1 \bmod 5) = 11$

$c_2 = 5 \times (9 \bmod 11) = 45$

The following table is drawn for $n = 55$

	0	1	2	3	4	5	6	7	8	9	10
0	0	45	35	25	15	5	50	40	30	20	10
1	11	1	46	36	26	16	6	51	41	31	21
2	22	12	2	47	37	27	17	7	52	42	32
3	33	23	13	3	48	38	28	18	8	53	43
4	44	34	24	14	4	49	39	29	19	9	54

Note that since $c_1 = 11$, moving down every row in a table increases a by 11 since $c_2 = 45$, moving right in a table increases a by 45.

* How to find multiplicative inverse.

→ Given two equations

$a \equiv 6 \pmod 5$

$a \equiv 3 \pmod{13}$

We want to find

(1) $13^{-1} \bmod 5$

(2) $5^{-1} \bmod 13$

(I) $13^{-1} \bmod 5$ — Here $n=5$, $a=13$

Then $x = \frac{1 + (k \times n)}{a} = \frac{1 + (k \times 5)}{13}$

Find smallest k such that we get remainder $=0$

In this case for $k = 5$

We get $x = \frac{1 + (25)}{13} = \frac{26}{13} = 2$

Thus quotient $= 2$ and remainder $= 0$

$\therefore$ $\boxed{13^{-1} = 2 \bmod 5}$

(II) $5^{-1} \bmod 13$ Here $n=13$, $a=5$

$x = \frac{1 + (k \times n)}{a} = \frac{1 + (k \times 13)}{5}$

Find smallest k such that we get remainder $=0$.

For $k = 3$

we get $x = \frac{40}{5} = 8$.

Thus quotient $= 8$ and remainder $= 0$

$\therefore$ $\boxed{5^{-1} = 8 \bmod 13}$

Applications of Chinese Remainder Th.

(1) In the RSA algorithm calculations are made modulo n, where n is a product of two large prime numbers p and q.

→ Making calculations in $\mathbb{Z}/n\mathbb{Z}$ is very time-consuming.

→ By using Chinese remainder theorem, these calculations can be made in the isomorphic ring $\mathbb{Z}/p\mathbb{Z} \oplus \mathbb{Z}/q\mathbb{Z}$ instead.

(2) The chinese remainder Th may also be used to construct an elegant Godel numbering for sequencing, which is needed to prove Godels incompleteness Theorem

$\mathbb{Z}/p^2\mathbb{Z} \oplus \mathbb{Z}/q^2$

$\mathbb{Z} \cong \mathbb{Z}$

Godel numbering for sequencing

Matrix chain Multiplication:

This algorithm solves the problem of matrix-chain multiplication.

- We are given a sequence of (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied and we wish to compute the product

$$A_1 A_2 \ldots A_n \quad \text{——— (1)}$$

- We can evaluate the eq (1) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.
- A product of matrices is <u>fully parenthesized</u> if it is either a single matrix or the product of two fully parenthized matrix products, surrounded by parantheses.
- Matrix multiplication is associative, and so all parenthizations yield the same product.

For e.g. if the chain of matrices is $\langle A_1 A_2 A_3 A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthized in five distinct ways.

$(A_1 (A_2 (A_3 A_4)))$,
$(A_1 ((A_2 A_3) A_4))$,
$((A_1 A_2) (A_3 A_4))$,
$((A_1 (A_2 A_3)) A_4)$,
$(((A_1 A_2) A_3) A_4)$.

• The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

• To illustrate the different costs incurred by different parenthesizations of a matrix product, consider a problem of a chain $\langle A_1 A_2 A_3 \rangle$. Suppose that the dimension of

A_1 is 10×100
A_2 is 100×5
A_3 is 5×50

• If we multiply according to parenthesization $((A_1 A_2) A_3)$. we perform $10 \times 100 \times 5 = 5000$ scalar multiplication to obtain 10×5 matrix product $A_1 A_2$, plus another $10 \times 5 \times 50 = 2500$ scalar multiplications to multiply $(A_1 A_2)$ by matrix A_3. Thus total multiplications in this case are $5000 + 2500 = 7500$.

• If instead, we multiply according to the parenthesization $(A_1 (A_2 A_3))$ then, we perform $100 \times 5 \times 50 = 25000$ scalar multiplication to obtain 100×50 matrix product $(A_2 A_3)$, plus another $10 \times 100 \times 50 = 50000$ scalar multiplications to multiply A_1 by $(A_2 A_3)$. Thus total of $50000 + 25000 = 75000$ scalar multiplications.

Thus computing the product according to the first parenthesization is 10 times faster than the second one.

Problem : Given a chain $\langle A_1, A_2, A_3, \ldots, A_n \rangle$ of n matrices, where $i = 1, 2, 3, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Our goal is to determine an order for multiplying matrices that has the lowest cost.

• Let denote the number of alternative parenthesization of a sequence of n matrices by $P(n)$.

• when $n = 1$, there is just one matrix and therefore only one way to fully parenthesize the matrix product.

• when $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between k^{th} and $(k+1)^{st}$ matrices for any $k = 1, 2, \ldots, n-1$. Thus

$$P(n) = \begin{cases} 1 & \text{, if } n = 1 \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{, if } n \geq 2 \end{cases}$$

Step 1 : The structure of an optimal parenthe--

• Let $A_{i..j}$ where $i \leq j$, denote the matrix that results from evaluating the product

$A_i A_{i+1} \ldots A_j$

- Now if $i < j$, then any parenthesization of a product $A_i A_{i+1} \cdots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we compute the product $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce final product $A_{i..j}$.
- The cost of this product is the cost of $A_{i..k}$ and the cost of $A_{k+1..j}$.
- Thus we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems.

(Optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$) finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

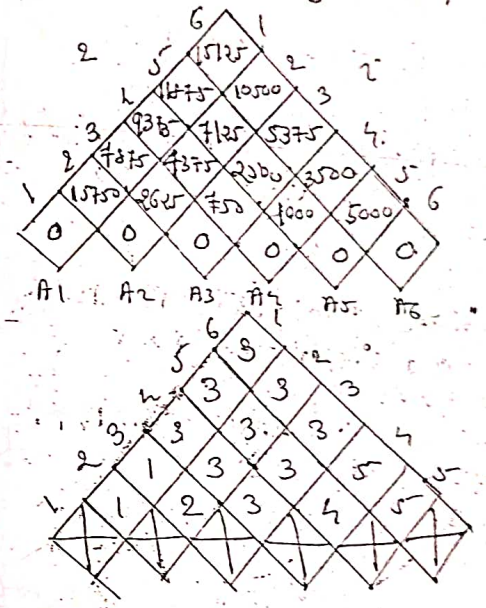
Step 2: A recursive solution:

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.

We can define $m[i,j]$ recursively as follows:

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

Step 3: Computing the optimal costs.



matrix	dimension
A_1	30×35 p_1
A_2	35×15 p_2
A_3	15×5 p_3
A_4	5×10 p_4
A_5	10×20 p_5
A_6	20×25 p_6

→ $m[1,2] = m[1,1] + m[2,2] + P_0 P_1 P_2$

$k=1$ $= 0 + 0 + (30 \times 35 \times 15)$

$= 15,750.$

→ $m[2,3] = m[2,2] + m[3,3] + P_1 P_2 P_3$

$k=2$ $= 0 + 0 + (35 \times 15 \times 5)$

$= 2625$

→ $m[3,4] = m[3,3] + m[4,4] + P_2 P_3 P_4$

$k=3$ $= (15 \times 5 \times 10) = 750.$

→ $m[4,5] = m[4,4] + m[5,5] + P_3 P_4 P_5$

$k=4$ $= (5 \times 10 \times 20) = 1000$

→ $m[5,6] = m[5,5] + m[6,6] + P_4 P_5 P_6$

$k=5$ $= 0 + 0 + (10 \times 20 \times 25)$

$= 5000$

→ $m[1,3]$.

for $k=1$ → $m[1,1] + m[2,3] + P_0 \times P_1 \times P_3$

$= 0 + 2625 + 30 \times 35 \times 5$

$= 7875$

for $k=2$ → $m[1,2] + m[3,3] + P_0 \times P_2 \times P_3$

$= 15750 + 0 + (30 \times 15 \times 5)$

$= 18,000.$

→ $m[2,4]$

$k=3$ → $m[2,3] + m[4,4] + P_1 \times P_3 \times P_4$

$= 2625 + (35 \times 05 \times 10)$

$= 7875$ 4375

$k=4$ → $m[2,2] + m[3,4] + P_1 \times P_2 \times P_4$

$= 0 + 750 + (35 \times 15 \times 10)$

$= 6000$

→ $m[3,5]$

for $k=3$ → $m[3,3] + m[4,5] + P_2 \times P_3 \times P_5$

$= 0 + 1000 + (15 \times 5 \times 20)$

$= 2500$

for $k=4$ → $m[3,4] + m[5,5] + P_2 \times P_4 \times P_5$

$= 750 + 0 + (15 \times 10 \times 20)$

$= 3750.$

→ $m[4,6]$

for $k=4$ → $m[4,4] + m[5,6] + P_3 \times P_4 \times P_6$

$= 0 + 5000 + (5 \times 10 \times 25)$

$= 6250$

for $k=5$ → $m[4,5] + m[6,6] + P_3 \times P_5 \times P_6$

$= 1000 + (5 \times 20 \times 25)$

$= 6000$ 3500

→ $m[1,4]$

for $k=1$ → $m[1,1] + m[2,4] + P_0 \times P_1 \times P_4$

$= 0 + 4375 + (30 \times 35 \times 10)$

$= 14,875$

for $k=2$ → $m[1,2] + m[3,4] + P_0 \times P_2 \times P_4$

$= 15750 + 750 + (30 \times 15 \times 10)$

$= 21000$

for $k=3$ → $m[1,3] + m[4,4] + P_0 \times P_3 \times P_4$

$= 7875 + 0 + (30 \times 5 \times 10)$

$= 9375$

→ $m[2,5]$

for $k=2$ → $m[2,2] + m[3,5] + P_1 \times P_2 \times P_5$

$= 0 + 2500 + (35 \times 15 \times 20)$

$= 13,000$

for $k=3$ → $m[2,3] + m[4,5] + P_1 \times P_3 \times P_5$

$= 2625 + 1000 + (35 \times 5 \times 20)$

$= 7125$

for $k=4$ → $m[2,4] + m[5,5] + P_1 \times P_4 \times P_5$

$= 4375 + (35 \times 10 \times 20)$

$= 11375$

$m[3,6]$

for $k=3 \rightarrow m[3,3] + m[4,5] + P_2 \times P_3 \times P_6$

$= 0 + 3500 + (15 \times 5 \times 25)$

$= 5375$

for $k=4 \rightarrow m[3,4] + m[5,6] + P_2 \times P_4 \times P_6$

$= 750 + 5000 + (15 \times 10 \times 25)$

$= 9500$

for $k=5 \rightarrow m[3,5] + m[6,6] + P_2 \times P_5 \times P_6$

$= 2500 + 0 + (15 \times 20 \times 25)$

$= 10,000$

$\rightarrow m[1,5]$

for $k=1 \Rightarrow m[1,1] + m[2,5] + P_0 \times P_1 \times P_5$

$= 0 + 7125 + (30 \times 35 \times 20)$

$= 28,125$

for $k=2 \Rightarrow m[1,2] + m[3,5] + P_0 \times P_2 \times P_5$

$= 15750 + 2500 + (30 \times 15 \times 20)$

$=$

for $k=3 \rightarrow m[1,3] + m[4,5] + P_0 \times P_3 \times P_5$

$= 7875 + 1000 + (30 \times 5 \times 20)$

$= 11,875$

for $k=4 \rightarrow m[1,4] + m[5,5] + P_0 \times P_4 \times P_5$

$= 9375 + 0 + (30 \times 10 \times 20)$

$= 15,375$

$\rightarrow m[2,6] = 10,500$

$\Rightarrow m[1,6] = 15,125$

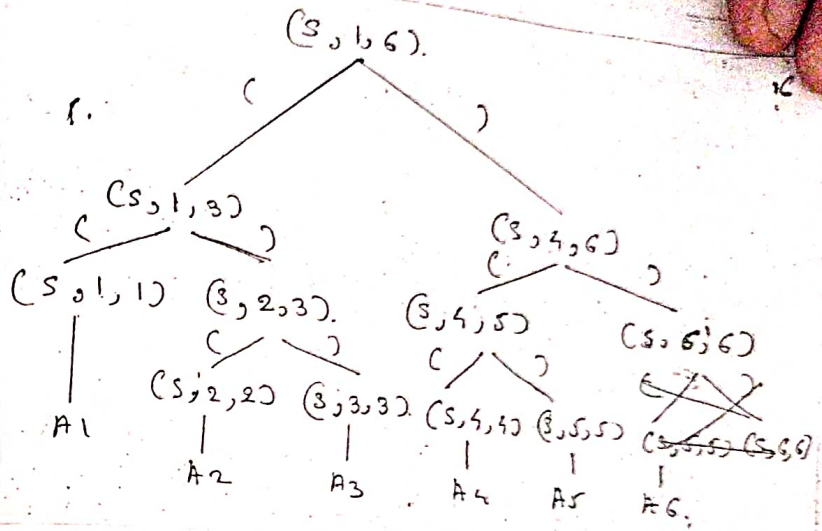
ALGORITHM

MATRIX-CHAIN-ORDER(i)

```
n ← length[i] - 1
for i ← 1 to n
    do m[i, i] = 0
for d ← 2 to n
    do for i ← 1 to n-d+1
        do for j ← i+d-1
            m[i, j] ← ∞
            for k ← i to j-1
                do q ← m[i, k] + m[k+1, j]
                        + Pi-1 Pk Pj
                if q < m[i, j]
                then m[i, j] ← q
                     s[i, j] ← k
return m and s.
```

Step 4 : Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
    then print "A" i
else
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j]+1, j)
    Print ")"
```



((A1 (A2 A3)) ((A4 A5) A6))

((A1 (A2 A3)) ((A4 A5) A6))

(S, i, j)

(S, i, s[i, j]) (S, s[i, j]+1, j)

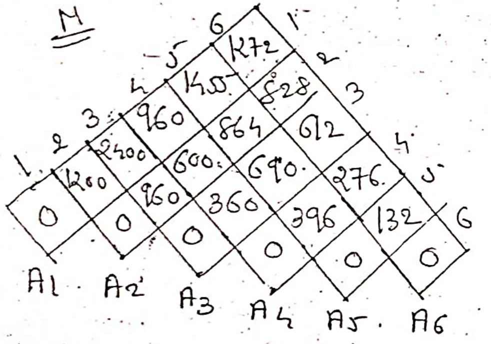
Problem: Find an optimal solution for matrix chain multiplication with dimension sequence.

$\langle 15, 8, 10, 12, 3, 11, 4 \rangle$

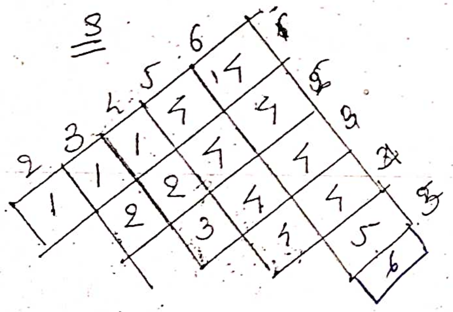
Sol^n

A1	15×8	P_1
A2	8×10	P_2
A3	10×12	P_3
A4	12×3	P_4
A5	3×11	P_5
A6	11×4	P_6

M



S



$\rightarrow m[1,2] =$

for $k=1 \rightarrow m[1,1] + m[2,2] + P_0 \times P_1 \times P_2$ $P_{i-1}\ P_k\ P_j$ $i=1,\ k=1,\ j=2$

$= 0 + 0 + (15 \times 8 \times 10)$

$= 1200$

$\rightarrow m[2,3]$

for $k=2 \rightarrow m[2,2] + m[3,3] + P_1 \times P_2 \times P_3$

$= 0 + 0 + (8 \times 10 \times 12)$

$= 960$

$\rightarrow m[3,4] = P_2 \times P_3 \times P_4$

for $k=3$ $= 10 \times 12 \times 3$

$= 360$

$\rightarrow m[4,5] = P_3 \times P_4 \times P_5$

$= (12 \times 3 \times 11)$

$= 396$

$\rightarrow m[5,6] = P_4 \times P_5 \times P_6$

$= (3 \times 11 \times 4)$

$= 132$

$\rightarrow m[1,3]$

for $k=1 \rightarrow m[1,1] + m[2,3] + P_0 \times P_1 \times P_3$ $P_{i-1}\ P_k\ P_j$

$= 960 + (15 \times 8 \times 12)$

$= 2,400$

for $k=2 \rightarrow m[1,2] + m[3,3] + P_0 \times P_2 \times P_3$

$= 1200 + 0 + (15 \times 10 \times 12)$

$= 3000$

$\rightarrow m[2,4] = m$

for $k=2 \rightarrow m[2,2] + m[3,4] + P_1 \times P_2 \times P_4$

$= 360 + (8 \times 10 \times 3)$

$= 600$

for $k=3 \rightarrow m[2,3] + m[4,4] + P_1 \times P_3 \times P_4$

$= 960 + (8 \times 12 \times 3)$

$= 1248$

→ m[3,5]

for k=3 → m[3,3] + m[4,5] + $P_2 \times P_3 \times P_5$
= 0 + 396 + (10 × 12 × 11)
= 1716

k=4 = m[3,4] + m[5,5] + $P_2 \times P_4 \times P_5$
= 360 + (10 × 3 × 11)
= 690.

→ m[4,6]

for k=4 → m[4,4] + m[5,6] + $P_3 \times P_4 \times P_6$
= 0 + 132 + (12 × 3 × 4)
= 276

for k=5 → m[4,5] + m[6,6] + $P_3 \times P_5 \times P_6$
= 396 + 0 + (12 × 11 × 4)
= 924

→ m[1,4]

for k=1 → m[1,1] + m[2,4] + $P_0 \times P_1 \times P_4$
= 600 + (15 × 8 × 3)
= 960

for k=2 → m[1,2] + m[3,4] + $P_0 \times P_2 \times P_4$
= 1200 + 360 + (15 × 10 × 3)
= 2010

for k=3 → m[1,3] + m[4,4] + $P_0 \times P_3 \times P_4$
= 2400 + 0 + (15 × 12 × 3)
= 2940

→ m[2,5]

k=2 → m[2,2] + m[3,5] + $P_1 \times P_2 \times P_5$
= 690 + (8 × 10 × 11)
= 1570

k=3 → m[2,3] + m[4,5] + $P_1 \times P_3 \times P_5$
= 960 + 396 + (8 × 12 × 11)
= 2412

k=4 → m[2,4] + m[5,5] + $P_1 \times P_4 \times P_5$
= 600 + 0 + (8 × 3 × 11)
= 864.

→ m[3,6]

k=3 → m[3,3] + m[4,6] + $P_2 \times P_3 \times P_6$
= 276 + (10 × 12 × 4)
= 756

k=4 → m[3,4] + m[5,6] + $P_2 \times P_4 \times P_6$
= 360 + 132 + (10 × 3 × 4)
= 612

k=5 → m[3,5] + m[6,6] + $P_2 \times P_5 \times P_6$
= 690 + 0 + (10 × 11 × 4)
= 1130.

→ m[1,5]

for k=1 → m[1,1] + m[2,5] + $P_0 \times P_1 \times P_5$
= 864 + (15 × 8 × 11)
= 2184

for k=2 → m[1,2] + m[3,5] + $P_0 \times P_2 \times P_5$
= 1200 + 690 + (15 × 10 × 11)
= ~~4776~~ 3540

for k=3 → m[1,3] + m[4,5] + $P_0 \times P_3 \times P_5$
= 2400 + 396 + (15 × 12 × 11)
= 4776.

for k=4 → m[1,4] + m[5,5] + $P_0 \times P_4 \times P_5$
= 960 + (15 × 3 × 11)
= 1455

$m[2,6]$

$k=2 \rightarrow m[2,2] + m[3,6] + p_1 \times p_2 \times p_6$

$= 0 + 612 + (8 \times 10 \times 4)$

$= 932$

$k=3 \rightarrow m[2,3] + m[4,6] + p_1 \times p_3 \times p_6$

$= 960 + 276 + (8 \times 12 \times 4)$

$= 1620$

$k=4 \rightarrow m[2,4] + m[5,6] + p_1 \times p_4 \times p_6$

$= 600 + 132 + (8 \times 3 \times 4)$

$= 828$

$k=5 \rightarrow m[2,5] + m[6,6] + p_1 \times p_5 \times p_6$

$864 + 0 + (8 \times 11 \times 4)$

$= 1216$

$\rightarrow m[1,6]$

$k=1 \rightarrow m[1,1] + m[2,6] + p_0 \times p_1 \times p_6$

$= 0 + 828 + (15 \times 8 \times 4)$

$= 1308$

$k=2 = m[1,2] + m[3,6] + p_0 \times p_2 \times p_6$

$1200 + 612 + (15 \times 10 \times 4)$

$= 2412$

$k=3 \rightarrow m[1,3] + m[4,6] + p_0 \times p_3 \times p_6$

$= 2400 + 276 + (15 \times 12 \times 4)$

$= 3396$

$k=4 \rightarrow m[1,4] + m[5,6] + p_0 \times p_4 * p_6$

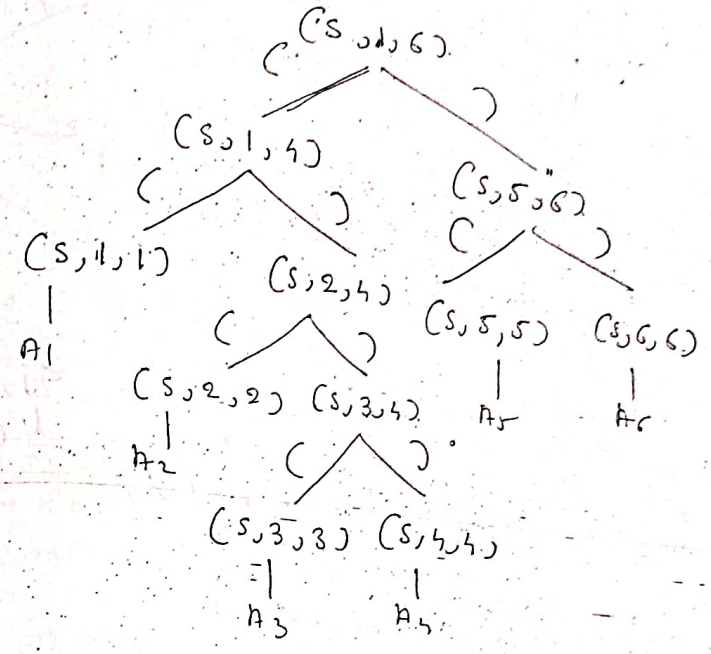
$= 960 + 132 + (15 \times 3 \times 4)$

$= (1272)$

$k=5 \rightarrow m[1,5] + m[6,6] + p_0 \times p_5 \times p_6$

$1455 + 0 + (15 \times 11 \times 4)$

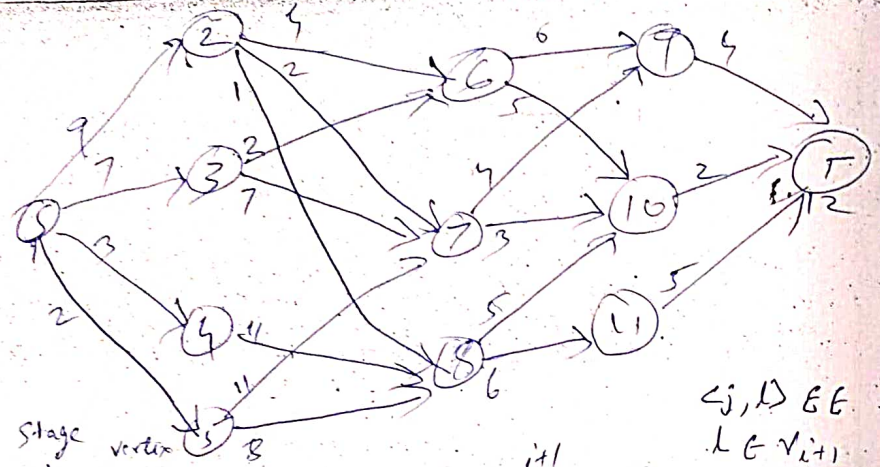
$= 2115$



Ans ~~((A1)(A2)(A3A4))~~

((A1 (A2 (A3A4))) (A5A6))

((A1 (A2 (A3A4))) (A5A6))



$\langle j, l \rangle \in E$
$l \in V_{i+1}$

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

$$Cost(3,3) = \min\{c(3,6) + Cost(3+1, 6),\ c(3,7) + Cost(3+1, 7)\}$$

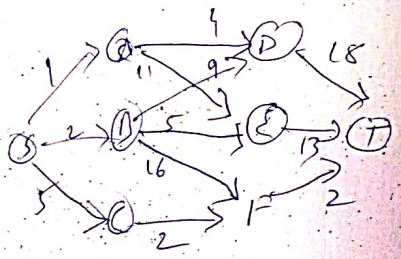
V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d	23	7	6	8	8	10	10	10	12	12	12	12

$d(1,1) = 2$
$d(2,2) = 7$
$d(3,7) = 10$
$d(4,10) = 12$

$Cost(5,12) = 0$
$Cost(4,9) = 4$
$10 = 2$
$11 = 5$

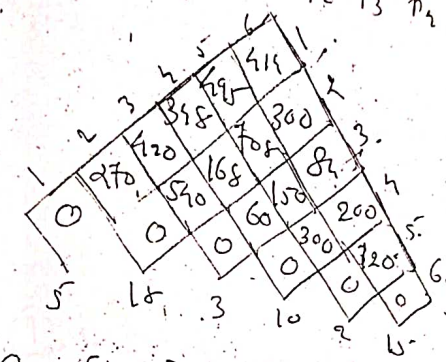
$$Cost(3,6) = \min\{(6,9) + Cost(4,9),\ (6,10) + Cost(4,10)\}$$

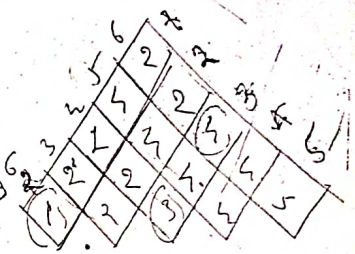
$$Cost(3,7) = \min\{(7,9) + Cost(4,9),\ (7,10)\ Cost(4,10)\}$$



9
$S \to 2 \to 7 \to T$

$\langle 5, 18, 3, 10, 2, 15, 4 \rangle$
$P_0\ P_1\ P_2\ P_3\ P_4\ P_5\ P_6$





① $m[1,3,3]$

$k=1$

$= m[1,1] + m[2,3] + P_0 \times P_1 \times P_3$

$0 + 540 + (5 \times 18 \times 10)$

$= 1440$

$k=2$

$m[1,2] + m[3,3] + P_0 \times P_2 \times P_3$

$270 + 0 + (5 \times 3 \times 10)$

$= 420$

② $m[2,4]$

$k=2$

$P_2 \times P_2 \times P_4$

$18 \times 3 \times 2$

$= 108 + 60$

$= 168$

$k=3$

$P_1 \times P_3 \times P_4$

$18 \times 10 \times 2 = 360$

$+ m[2,3] + m[4,4]$

$360 + 540 + 0 =$

③ $m[3,5]$

$k=3$

$P_2 \times P_3 \times P_5$

$3 \times 10 \times 15 = 450$

$k=4$

$P_2 \times P_4 \times P_5$

$3 \times 2 \times 15 = 90 +$

$m[3,4] + m[5,5]$

$= 90 + 60 + 0 = 150.$

④ $m[3,6]$

$k=4$

$p_3 \times p_4 \times p_6$

$10 \times 2 \times 4$

$= 80 + m[3,4] + m[5,6]$

$= 80 + 120 = 200$

$k=5$

$p_3 \times p_5 \times p_6$

$10 \times 15 \times 4 = 600$

⑤ $m[1,4]$

$k=1$

$p_0 \times p_1 \times p_4$

$5 \times 18 \times 2$

$= 180$

$180 + m[1,1] + m[2,4]$

$= 180 + 168$

$= 348$ ✓

$k=2$

$p_0 \times p_2 \times p_4$

$5 \times 3 \times 2$

$= 30$

$+ m[1,2] + m[3,4]$

$= 30 + 270 + 60$

$= 360.$

$k=3$

$p_0 \times p_3 \times p_4$

$5 \times 10 \times 2$

$= 100$

$100 + m[1,3] + m[4,4]$

$= 100 + 420$

$= 520.$

⑥ $m[2,5]$

$k=2$ $p_1 \times p_2 \times p_5$

$18 \times 3 \times 15$

$= 810$

$810 + m[2,2] + m[3,5]$

$= 810 + 0 + 150$

$= 960$

$k=3$

$p_1 \times p_3 \times p_5$

$18 \times 10 \times 15$

$180 \times 15 + m[2,3] + [4,5]$

$+ 540 + 300$

$k=4$

$p_1 \times p_4 \times p_5$

$18 \times 2 \times 15$

$= 540$

$520 + m[2,4] + m[5,5]$

$520 + 168 = 708$

⑦ $m[3,6]$

$k=3$

$p_2 \times p_3 \times p_6$

$3 \times 10 \times 4$

$k=4$

$p_2 \times p_4 \times p_5$

$3 \times 2 \times 4$

$+ m[3,4] + m[5,5]$

$= 24 + 60 = 84$

$k=5$

$p_2 \times p_5 \times p_6$

$3 \times 15 \times 3$

⑧ $m[1,5]$

$k=1$

$p_0 \times p_1 \times p_5$

$5 \times 18 \times 15$

$k=2$

$p_0 \times p_2 \times p_5$

$5 \times 3 \times 15$

$= 225$

$+ m[1,2] + m[3,5]$

$= 225 + 270 + 150$

$= 645$

$k=3$

$p_0 \times p_3 \times p_5$

$5 \times 10 \times 15$

$k=4$

$p_0 \times p_4 \times p_5$

$5 \times 2 \times 15$

$= 150$

$150 + m[1,4] + m[5,5]$

$= 150 + 348$

$= 498$ ✓

⑨ $m[2,6]$

$k=2$

$p_1 \times p_2 \times p_6$

$18 \times 3 \times 4$

$= 216$

$+ m[2,2] + m[3,6]$

$= 216 + 84$

$= 300$ ✓

$k=3$

$p_1 \times p_3 \times p_6$

$18 \times 10 \times 4$

$k=4$

$p_1 \times p_4 \times p_6$

$18 \times 2 \times 4$

$= 144$

$+ m[2,3] + m[4,4]$

$= 144 + 168 + 120$

$= 432$

$k=5$

$p_1 \times p_5 \times p_6$

$18 \times 15 \times 4$

⑩ $m[1,6]$

$k=1$

$p_0 \times p_1 \times p_6$

$5 \times 18 \times 4$

$= 360$

$k=2$

$p_0 \times p_2 \times p_6$

$5 \times 3 \times 4$

$= 60$ ✓

$60 + m[1,2] + m[3,6]$

$= 60 + 270 + 84$

$= 414$ ✓

$k=3$

$p_0 \times p_3 \times p_6$

$5 \times 10 \times 4$

$= 200$

$k=4$

$p_0 \times p_4 \times p_6$

$5 \times 2 \times 4$

$= 40$ ✓

$40 + m[1,4] + m[5,6]$

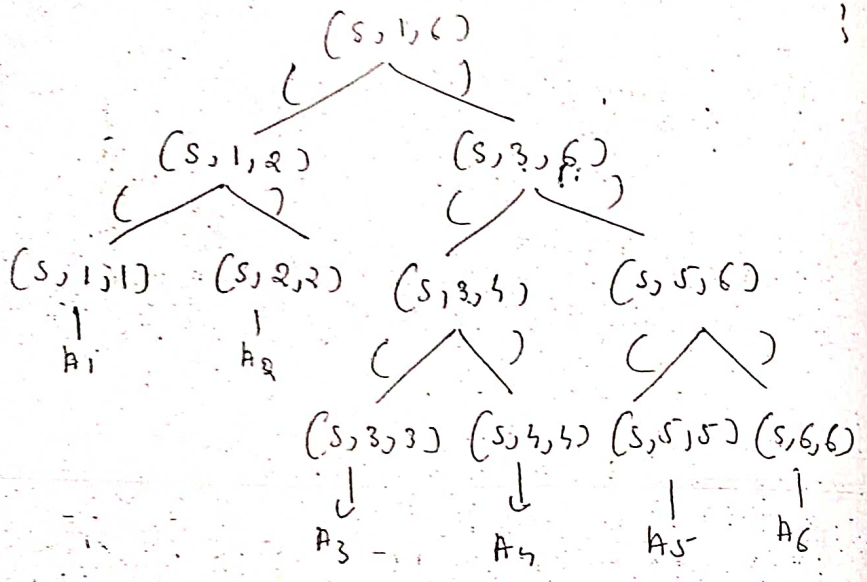
$= 40 + 348 + 120$

$= 508$

$k=5$

$p_0 \times p_5 \times p_6$

$5 \times 15 \times 4$



((A1A2)((A3A4)(A5A6)))

Problems based on Hashing

Problem 1: Consider inserting keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length m=11 using open addressing, with the primary hash function

$$h'(k) = k \bmod m$$

Illustrate the result of inserting the keys using:

1. Linear probing
2. quadratic probing with $c_1=1$, $c_2=3$
3. Using double hashing where

$$h_2(k) = k \bmod (m-1)$$

soln 1. Linear probing.

for linear probing, Here m=11.

$$h(k,i) = (h'(k)+i) \bmod m$$

→ $h(10,0) = (h'(10)+0) \bmod 11$
$= (10 \bmod 11) \bmod 11$
$= 10 \bmod 11$
$= \boxed{10}$

→ $h(22,0) = (h'(22)+0) \bmod 11$
$= (22 \bmod 11) \bmod 11$
$= 0 \bmod 11$
$= \boxed{0}$

→ $h(31,0) = (h'(31)+0) \bmod 11$
$= (31 \bmod 11) \bmod 11$
$= 9 \bmod 11$
$= \boxed{9}$

0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

→ $h(4,0) = (h'(4) + 0) \bmod 11$
$= (4 \bmod 11) \bmod 11$
$= 4 \bmod 11$
$= \boxed{5}$

→ $h(15,0) = (h'(15) + 0) \bmod 11$
$= (15 \bmod 11) \bmod 11$
$= 4 \bmod 11$
$= 4 \longrightarrow$ collision

$h(15,1) = ((15 \bmod 11) + 1) \bmod 11$
$= [5] \bmod 11$
$= \boxed{5}$

→ $h(28,0) = [(28 \bmod 11) + 0] \bmod 11$
$= 6 \bmod 11$
$= \boxed{6}$

→ $h(17,0) = [(17 \bmod 11) + 0] \bmod 11$
$= 6 \bmod 11$
$= 6 \longrightarrow$ collison

$h(17,1) = [(17 \bmod 11) + 1] \bmod 11$
$= 7 \bmod 11$
$= \boxed{7}$

→ $h(88,0) = [(88 \bmod 11) + 0] \bmod 11$
$= 0 \longrightarrow$ collison

$h(88,1) = [(88 \bmod 11) + 1] \bmod 11$
$= 1 \bmod 11$
$= \boxed{1}$

→ $h(59,0) = [(59 \bmod 11) + 0] \bmod 11$
$= 4 \bmod 11$
$= 4 \longrightarrow$ collison

$h(59,1) = [5 \bmod 11] = 5$ — $h(59,4) = 8 \bmod 11$
$h(59,2) = [6 \bmod 11] = 6$
$h(59,3) = [7 \bmod 11] = 7$ $= \boxed{8}$

2. Quadratic probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

→ $h(10,0) = [(10 \bmod 11) + 0 + 0] \bmod 11$
$= 10 \bmod 11$
$= \boxed{10}$

→ $h(22,0) = [(22 \bmod 11) + 0 + 0] \bmod 11$
$= 0 \bmod 11$
$= \boxed{0}$

→ $h(31,0) = [(31 \bmod 11) + 0 + 0] \bmod 11$
$= 9 \bmod 11$
$= \boxed{9}$

→ $h(4,0) = [(4 \bmod 11) + 0 + 0] \bmod 11$
$= 4 \bmod 11$
$= \boxed{4}$

→ $h(15,0) = [(15 \bmod 11) + 0 + 0] \bmod 11$
$= 4 \bmod 11$
$= 4 \longrightarrow$ collision.

$h(15,1) = [(15 \bmod 11) + (1 \times 1) + (3 \times 1)] \bmod 11$
$= [4 + 1 + 3] \bmod 11$
$= 8 \bmod 11$
$= \boxed{8}$

→ $h(28,0) = [(28 \bmod 11) + 0 + 0] \bmod 11$
$= 6 \bmod 11$
$= \boxed{6}$

→ $h(17,0) = [(17 \bmod 11) + 0 + 0] \bmod 11$
$= 6 \bmod 11$
$= 6 \longrightarrow$ collision.

$h(17,1) = [(17 \bmod 11) + (1\times1) + (3\times1)] \bmod 11$

$= [6+1+3] \bmod 11$

$= 10 \bmod 11$

$= 10 \longrightarrow$ collision

$h(17,2) = [(17 \bmod 11) + (1\times2) + (3\times4)] \bmod 11$

$= (6+2+12) \bmod 11$

$= 20 \bmod 11$

$= 9 \longrightarrow$ collision

$h(17,3) = [(17 \bmod 11) + (1\times3) + (3\times9)] \bmod 11$

$= [6+3+27] \bmod 11$

$= 36 \bmod 11$

$= \boxed{3}$

$\rightarrow h(88,0) = [(88 \bmod 11) + 0 + 0] \bmod 11$

$= 0 \bmod 11$

$= 0 \longrightarrow$ collision

$h(88,1) = [(88 \bmod 11) + (1\times1) + (3\times1)] \bmod 11$

$= [0+1+3] \bmod 11$

$= 4 \longrightarrow$ collision

$h(88,2) = [(88 \bmod 11) + (1\times2) + (3\times4)] \bmod 11$

$= [0+2+12] \bmod 11$

$= 14 \bmod 11$

$= 3 \longrightarrow$ collision.

$h(88,3) = [(88 \bmod 11) + (1\times3) + (3\times9)] \bmod 11$

$= 30 \bmod 11$

$= 9 \longrightarrow$ collision.

$h(88,4) = [(88 \bmod 11) + (1\times4) + (3\times16)] \bmod 11$

$= [0+4+48] \bmod 11$

$= 52 \bmod 11$

$= 8 \longrightarrow$ collision

$h(88,5) = [0 + (1\times5) + (3\times25)] \bmod 11$

$= 180 \bmod 11$

$= 3 \longrightarrow$ collison

$h(88,6) = [0 + (1\times6) + (3\times36)] \bmod 11$

$= [6+108] \bmod 11$

$= 114 \bmod 11$

$= 4 \longrightarrow$ collison

$h(88,7) = [0 + (1\times7) + (3\times49)] \bmod 11$

$= [7+147] \bmod 11$

$= 154 \bmod 11$

$= 0 \longrightarrow$ collison

$h(88,8) = [0 + (1\times8) + (3\times64)] \bmod 11$

$= [8+192] \bmod 11$

$= 200 \bmod 11$

$= \boxed{2}$

$\rightarrow h(59,0) = [(59 \bmod 11) + 0 + 0] \bmod 11$

$= [4 \bmod 11]$

$= 4 \rightarrow$ collison

$h(59,1) = [(59 \bmod 11) + (1\times1) + (3\times1)] \bmod 11$

$= (4+1+3) \bmod 11$

$= 8 \bmod 11$

$= 8 \longrightarrow$ collison

$h(59,2) = [4 + (1\times2) + (3\times4)] \bmod 11$

$= [4+2+12] \bmod 11$

$= 18 \bmod 11$

$= \boxed{7}$

0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Ans quadratic probing.

3. Double Hashing.

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

$h_1(k) = h'(k) = k \bmod m$

$h_2(k) = k \bmod (m-1)$

→ $h(10,0) = [h_1'(10) + (0 \times h_2(10)] \bmod 11$
$= [(10 \bmod 11) + 0] \bmod 11$
$= \boxed{10}$

→ $h(22,0) = [(22 \bmod 11) + 0] \bmod 11$
$= \boxed{0}$

→ $h(31,0) = [(31 \bmod 11) + 0] \bmod 11$
$= \boxed{9}$

→ $h(4,0) = [(4 \bmod 11) + 0] \bmod 11$
$= \boxed{4}$

→ $h(15,0) = [(15 \bmod 11) + 0] \bmod 11$
$= 4 \longrightarrow$ collison

$h(15,1) = [(15 \bmod 11) + (1 \times (15 \bmod 10))] \bmod 11$
$= [4 + 5] \bmod 11$
$= 9 \longrightarrow$ collison.

$h(15,2) = [(15 \bmod 11) + (2 \times (15 \bmod 10))] \bmod 11$
$= [4 + (2 \times 5)] \bmod 11$
$= 14 \bmod 11$
$= \boxed{3}$

→ $h(28,0) = [(28 \bmod 11) + 0] \bmod 11$
$= 6 \bmod 11$
$= \boxed{6}$

→ $h(17,0) = [(17 \bmod 11) + 0] \bmod 11$
$= 6 \bmod 11$
$= 6 \longrightarrow$ collison

$h(17,1) = [(17 \bmod 11) + (1 \times 17 \bmod 10)] \bmod 11$
$= [6 + 7] \bmod 11$
$= 13 \bmod 11$
$= \boxed{2}$

→ $h(88,0) = [(88 \bmod 11) + 0] \bmod 11$
$= 0 \longrightarrow$ collison

$h(88,1) = [(88 \bmod 11) + (1 \times 88 \bmod 10)] \bmod 11$
$= (0 + 8) \bmod 11$
$= \boxed{8}$

→ $h(59,0) = [(59 \bmod 11) + 0] \bmod 11$
$= 4 \bmod 11$
$= 4 \longrightarrow$ collison

$h(59,1) = (4 + 9) \bmod 11 = 13 \bmod 11$
$= 2 \longrightarrow$ collison

$h(59,2) = [4 + (2 \times 9)] \bmod 11$
$= 22 \bmod 11$
$= 0 \longrightarrow$ collison

$h(59,3) = [(59 \bmod 11) + (3 \times 59 \bmod 10)] \bmod 11$

$= (4 + 27) \bmod 11$

$= 31 \bmod 11$

$= 9 \longrightarrow$ collision

$h(59,4) = [(59 \bmod 11) + (4 \times 59 \bmod 10)] \bmod 11$

$= (4 + 36) \bmod 11$

$= 40 \bmod 11$

$= \boxed{7}$

0	22
1	
2	17
3	13
4	4
5	
6	28
7	59
8	88
9	31
10	10

Ans: double hashing.

Problem: 2: Consider inserting keys 10, 22, 31, 4, 15. If we have a hash table of length 5. Using open addressing, with the primary hash function

$h'(k) = k \bmod m$

Illustrate the result of inserting the key using

1. Linear probing
2. quadratic probing with $c_1 = 1$, $c_2 = 3$
3. Using double hashing where

$h_2(k) = k \bmod (m-1)$.

Soln 1. **Leanear probing**. $h(k,i) = (h'(k) + i) \bmod m$

$\rightarrow h(10,0) = (10 \bmod 5) \bmod 5$

$= 0 \bmod 5$

$= 0$

$\rightarrow h(22,0) = (-22 \bmod 5) \bmod 5$

$= 3 \bmod 5$

$= 3$

$\rightarrow h(31,0) = (31 \bmod 5) \bmod 5$

$= (1 \bmod 5)$

$= 1$.

$\rightarrow (4,0) = (-4 \bmod 5) \bmod 5$

$= 1 \bmod 5$ "

$= 1 \longrightarrow$ collision

$(4,1) = ((-4 \bmod 5) + 1) \bmod 5$

$= (1+1) \bmod 5$

$= 2$.

0	10	-5
1	31	-4
2	4	-3
3	22	-2
4	15	-1

→ h(15,0) = (15 mod 5) mod 5
= 0 mod 5
= 0 ⟶ collison

h(15,1) = (0+1) mod 5
= 1 ⟶ collison

h(15,2) = (0+2) mod 5
= 2.

h(15,3) = (0+3) mod 5
= 3 ⟶ collison

h(15,4) = (0+4) mod 5
= 4.

2- Quadratic probing.

h(k,i) = (h'(k) + (i x c1) + (i² x c2)) mod m

→ h(10,0) = (10 mod 5) mod 5
= 0

0	10	-5
1	31	-4
2		-3
3	-22	-2
4		-1

→ h(-22,0) = (-22 mod 5) mod 5
= 3 mod 5
= 3

→ h(31,0) = (31 mod 5) mod 5
= 1

→ h(-4,0) = (-4 mod 5) mod 5
= 1 mod 5
= 1 ⟶ collision

h(-4,1) = ((-4 mod 5) + (1x1) + (1x3)) mod 5
= (1+1+3) mod 5
= 0 ⟶ collision.

h(-4,2) = [(-4 mod 5) + (2x1) + (4x3)] mod 5
= [1+2+12] mod 5
= 15 mod 5
= 0 ⟶ collision

h(-4,3) = [(-4 mod 5) + (3x1) + (9x3)] mod 5
= [1+3+27] mod 5
= 31 mod 5
= 1 ⟶ collision.

h(-4,4) = [(-4 mod 5) + (4x1) + (16x3)] mod 5
= [1+4+48] mod 5
= 53 mod 5
= 3 ⟶ collison.

h(-4,5) = [(-4 mod 5) + (5x1) + (25x3)] mod 5
= [1+5+75] mod 5
= 81 mod 5
= 1 ⟶ collision

h(-4,6) = [(-4 mod 5) + (6x1) + (36x3)] mod 5
= (1+6+108) mod 5
= 115 mod 5
= 0 ⟶ collision

h(-4,7) = [(-4 mod 5) + (7x1) + (49x3)] mod 5
= (1+7+147) mod 5
= 155 mod 5
= 0.

$h(4,8) = [1 + (1\times8) + (64\times32)] \bmod 5$

$= [1+8+1928] \bmod 5$

$= 201 \bmod 5$

$= 1$

$h(4,9) = [1 + (1\times9) + (81\times32)] \bmod 5$

$= [1+9+2432] \bmod 5$

$= 253 \bmod 5$

$= 3$

Q-3 $h'(k) = k \bmod m$ $\quad m=11$

keys = 23, 52, 68, -11, 25, -2, 78, -13, 112, 0, 11

1) Linear probing
2) Quadratic probing $c_1=1$ $c_2=3$
3) Double hashing $h''(k) = 2 + (k \bmod m-1)$

Soln (I) Linear probing $m=11$

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
-11	23	68	25	78	112	0	11	52	-2	-13
0	1	2	3	4	5	6	7	8	9	10

$k=23$ $\quad 23 \% 11 = 1$

$k=52$ $\quad 52 \% 11 = 8$

$k=68$ $\quad 68 \% 11 = 2$

$k=-11$ $\quad -11 \% 11 = 0$

$k=25$ $\quad 25 \% 11 = 3$

$k=-2$ $\quad -2 \% 11 = 9$

$k=78$ $\quad 78 \% 11 = 1$ — collision
Next slot = 4

$k=-13$ $\quad -13 \% 11 = 9$ collision
— Next slot = 10

$k=112$ $\quad 112 \% 11 = 2$
Next slot = 5

$k=0$ $\quad 0 \% 11 = 0$
Next slot = 6

$k=11$ $\quad 11 \% 11 = 1$

(II). Quadratic Probing

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
-11	23	68	52	0	78	-13	11	52	-2	112
0	1	2	3	4	5	6	7	8	9	10

$k=78$ $\quad 78 \% 11 = 1$

$h(78,1) = (1 + (1\times1) + (3\times1)) \% 11 = 5$

$k=-13$ $\quad -13 \% 11 = 9$

$h(-13,1) = (9+1) \% 11 = 2$

$h(-13,2) = (9 + 2 + (4\times3)) \% 11 = 1$

$h(-13,3) = (9+3+(9\times3)) \% 11 = 9$

$k = 112$ $\quad 112 \% 11 = 2$

$h(112,1) = (2+4) \% 11 = 6$

$h(112,2) = (2+14) \% 11 = 5$

$h(112,3) = (2+30) \% 11 = 10$

$k = 0 \quad 0 \% 11 = 0$

$h(0,1) \quad (0+4) \% 11 = 4$

Q-1

$m = 11 \quad c_1 = 1, c_2 = 3$

6, 22, -59, 18, 111, -2, 7, 20, 32, 0, 4

(I) Linear Probing

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
22	111	20	32	0	4	6	-59	18	-2	7
0	1	2	3	4	5	6	7	8	9	10

1) $6 \% 11 = 6$
2) $22 \% 11 = 0$
3) $-59 \% 11 = 7$
4) $18 \% 11 = 7 \rightarrow$ select next slot $= 8$
5) $111 \% 11 = 1$
6) $-2 \% 11 = 9$
7) $7 \% 11 = 7 \rightarrow$ select next slot $= 10$
8) $20 \% 11 = 9 \rightarrow$ select " " $= 2$
9) $32 \% 11 = 10 \rightarrow$ " " " $= 3$
10) $0 \% 11 = 0 \rightarrow$ " " " $= 4$
11) $4 \% 11 = 4 \rightarrow$ select " " $= 5$

(II) Quadratic probing.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
22	111	20	32	7	4	6	-59	0	-2	18
0	1	2	3	4	5	6	7	8	9	10

1) $6 \% 11 = 6$ 2) $22 \% 11 = 0$ 3) $-59 \% 11 = 7$
4) $18 \% 11 = 6 \rightarrow$ collision
$h(18,1) = (6+1+(3)) \% 11 = 10$
5) $111 \% 11 = 1$
6) $-2 \% 11 = 9$
7) $7 \% 11 = 7$
$h(7,1) = (7+4) \% 11 = 0$
$h(7,2) = (7+14) \% 11 = 9$
$h(7,3) = (7+30) \% 11 = 3$
8) $20 \% 11 = 9$
$h(20,1) = (9+4) \% 11 = 2$

$32 \% 11 = 10$

$h(32,1) = (10+1) \% 11 = 3$

$0 \% 11 = 0$

$h(0,1) = (0+1) \% 11 = 1$

$h(0,2) = (0+1(2)) \% 11 = 3$

$h(0,3) = (0+3(3)) \% 11 = 9$

Q-2 $h(k) = k \bmod m$, $m = 11$

52, 23, 78, 3, 0, 69, 29, 11, 13, -2, -8

1) Linear probing
2) Quadratic probing $c_1 = 1$, $c_2 = 3$
3) Double Hashing $h_2(k) = 7 + (k \% (m-1))$

(I) Linear Probing

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	23	78	3	69	11	13	29	52	-2	-8
0	1	2	3	4	5	6	7	8	9	10

1) $52 \% 11 = 8$
2) $23 \% 11 = 1$
3) $78 \% 11 = 1 \rightarrow$ next slot $= 2$
4) $3 \% 11 = 3$
5) $0 \% 11 = 0$
6) $69 \% 11 = 3 \rightarrow$ next slot $= 4$
7) $29 \% 11 = 7$
8) $11 \% 11 = 1 \rightarrow$ next slot $= 5$
9) $13 \% 11 = 2 \rightarrow$ next slot $= 6$
10) $-2 \% 11 = 9$
11) $-8 \% 11$

(II) Quadratic Probing.

0	23	13	3	11	78	-8	69	52	-2	29
0	1	2	3	4	5	6	7	8	9	10

1) $52 \% 11 = 8$
2) $23 \% 11 = 1$
3) $78 \% 11 = 1$

$h(78,1) = (1+1) \% 11 = 5$

4) $3 \% 11 = 3$
5) $0 \% 11 = 0$
6) $69 \% 11 = 3$

$h(69,1) = (3+1) \% 11 = 4$

7) $29 \% 11 = 7$

$h(29,1) = (7+1) \% 11 = 1$

$h(29,2) = (7+1(4)) \% 11 = 10$

8) $11 \% 11 = 0$

$h(11,1) = (0+4) \% 11 = 4$

9) $13 \% 11 = 2$

10) $-2 \% 11 = 9$

(III) Double Hashing.

$h_2(k) = 2 + (k \% (m-1))$.

$h(k,i) = ((k\%m) + (2 + (k\%(m-1)))) \% m$.

			s/e	f	s/e.f
1	1	s=0;	1	1	1
m+1	1	for(i=1; i<=m; i++)	1	n+1	n+1
m(n+1)	1	{ for(j=1; j<=n; j++)	0	—	0
m.n	1	{ s=s+a[i];	1	n	n
0	0	}	0	—	0
0	0	}	0	—	0
					2n+2

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

$= O(n)$

Ch = n times

(III) Double Hashing.

$h_2(k) = 2 + (k \% (m-1))$

$\therefore h(k,i) = (h'(k) + h_2(k)) \% m$

$= ((k\%m) + i(2 + (k\%(m-1)))) \% m$

0	1	2	3	4	5	6	7	8	9	10
78	23	2	3					52		

(3) $k = 78$ $\quad 78 \% 11 = 1$ collision

$\therefore h(78,1) = (1 + 1(2 + (78\%10))) \% 11$

$= (1 + (2+8)) \% 11 = 0$

(4) $k = 3$ $\quad 3 \% 11 = 3$

(5) $k = 0$ $\quad 0 \% 11 = 0$ collision

$h(0,1) = (0 + 1(2 + (0\%10))) \% 11$

$= 2 \% 11 = 2$

(c) $k = 69$ $\quad 69 \% 11 = 3$ collision

$h(69,1) = (3 + 1(2 + (69\%10))) \% 11$

$= (3 + (2+9)) \% 11 = 3$

$h(69,2) = (3 + 2(2+9)) \% 11$

$= 25 \% 11 = 3$

$h(69,3) = (3 + 3(2+9)) \% 11$

$36 \% 11 = 3$

$h(69,3) = (3 + 4(11)) \% 11$

$47 \% 11 = 3$

2nd

Last

(II) Quadratic Probing.

$h(k,i) = (k \% m + (i \times c_1) + (i^2 \times c_2)) \% m$

i		
i=1	$1 + (3 \times 1)$	$= 4$
2	$2 + (3 \times 4)$	$= 14$
3	$3 + (3 \times 9)$	$= 30$
4	$4 + (3 \times 16)$	$= 52$
5	$5 + (3 \times 25)$	$= 80$
6	$6 + (3 \times 36)$	$= 114$
7	$7 + (3 \times 49)$	$= 154$
8	$8 + (3 \times 64)$	$= 200$
9	$9 +$	

175

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9	10
0	23	13	3	11	78	-8	69	52	-2	29.

① $k = 78$ $78 \% 11 = 7$ collision

$h(78,1) = (1+4) \% 11 = 5$

② $k = 3$ $3 \% 11 = 3$.

③ $k = 0$ $0 \% 11 = 0$

④ $k = 69$ $69 \% 11 = 3$ collision

$h(69,1) = (3+4) \% 11 = 7$

⑤ $k = 29$ $29 \% 11 = 7$ collision

$h(29,1) = (7+4) \% 11$

$= 11 \% 11 = 0$.

$h(29,2) = (7+14) \% 11$

$21 \% 11 = 10$.

⑥ $k = 11$ $11 \% 11 = 0$

$h(11,1) = (0+4) \% 11 = 4$

⑦ $k = 13$ $13 \% 11 = 2$.

⑧ $k = -2$ $-2 \% 11 = 9$

⑨ $k = -8$. $-8 \% 11 = 3$

$h(-8,1) = (3+4) \% 11 = 7$

$(3+14) \% 11 = 6$