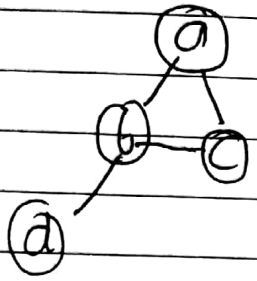
GRAPH

18/9/18

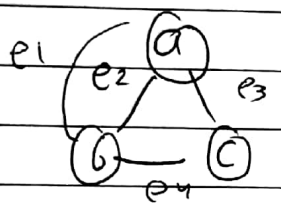
1. A tuple of $G = (V, E)$ — E: set of edges; V: set of vertices

$E \subseteq V \times V$

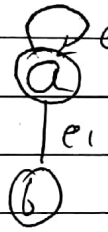


$V = \{a, b, c, d\}$

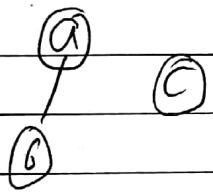
$E = \{(a,b), (b,c), (a,c), (b,d)\}$

- 

$e1 // e2$ parallel Edge

- 

e2 self loop

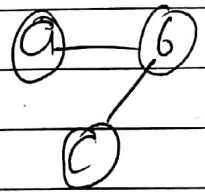
- 

$V = \{a, b, c\}$

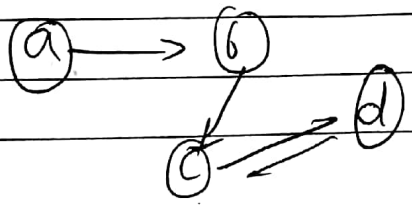
$E = \{(a,b), (b,a)\}$

disconnected graph

- Graph with no direction - Unidirectional graph
  or
  Undirected graph
  or
  ected graph



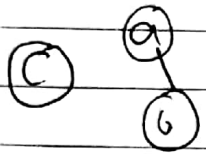
- Graph with direction - directional graph
  or
  directed graph
  or
  digraph



Simple graph:-

- It should not have parallel edges, self loops and disconnected graph.

Degree:-

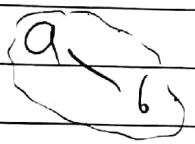


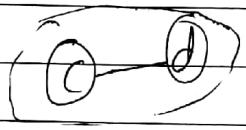
$d_a = 1$ $d_c = 0$

$d_b = 1$

$T \subseteq G$

Tree Graph

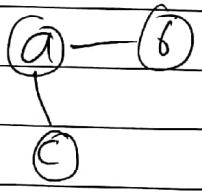




Components

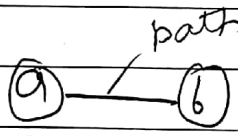
Adjacent nodes:-

Nodes connected with edge

eg 

a, b and a, c

Path:-



Complete graph:-

Graph where each node is connected to every other nodes of graph.

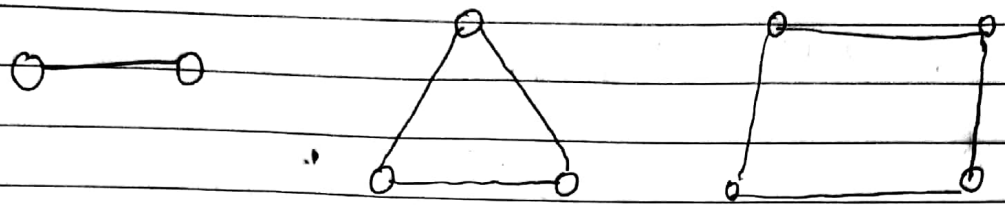
denoted by K_n

n - no. of nodes or vertices

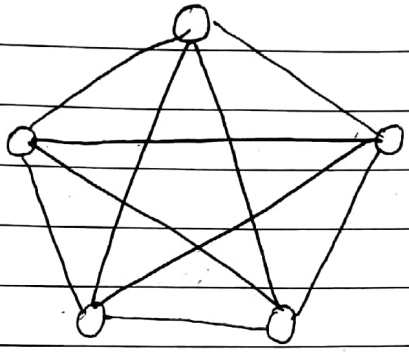
No. of edges = $^nC_2$ degree = $n-1$

__/__/____

K_2 K_3 K_4



K_5



No. of edges for complete directed graph - $n(n-1)$

" " " " " undirected graph - $\frac{n(n-1)}{2}$

$= {}^nC_2$

Regular Graph:-

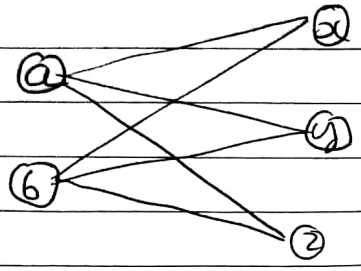
R

Vertex, degree

	d = 0	d = 1	d = 2	d = 3
V = 1	•			
V = 2	• •	—		
V = 3	• • •		△	
V = 4	• • • •	⌒	□	◇

Bipartite Graph (Bigraph)

Denoted by $K_{m,n}$ No. of edges $m \times n$

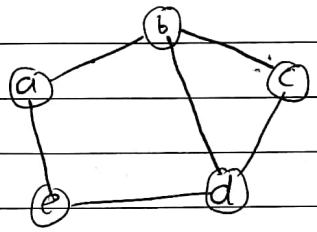


$K_{2,3}$

no. of edges = 6

Circuit - is path that begins and ends at same vertex, no repeated edges.

Walk -



abcd6 is called walk

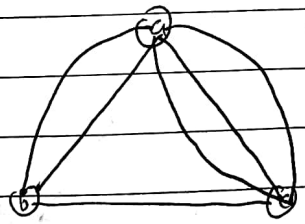
abcdea is called closed walk.

Trail - walk with no repeated edges

Path - Trail with no repeated vertices

Cycle - closed trail without repeated vertices other than starting or ending vertices

Multigraph:- It is a graph with multiple edges between nodes

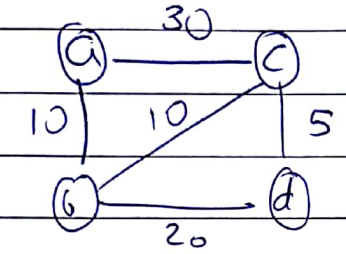


19/4/18

Representation of Graphs:-

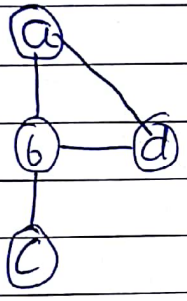
1. Set Representation $G = (V, E)$ $V = \{$ Vertices $\}$ edges $E = \{(0,1)\}$
2. Sequential Representation using adjacency matrix
3. Linked representation using linked list

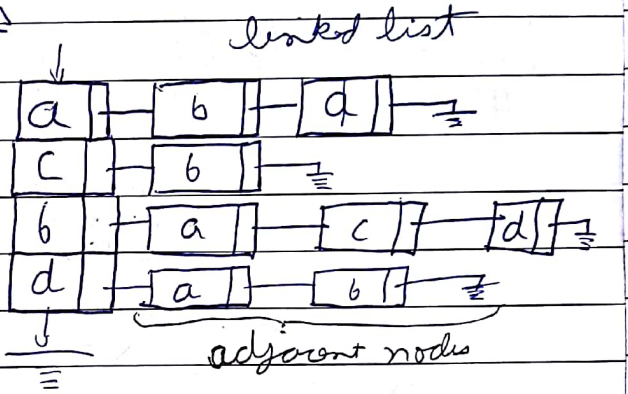
Weighted graph: a graph in which each edge carries a value



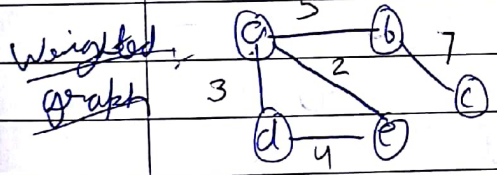
3. **Linked representation**

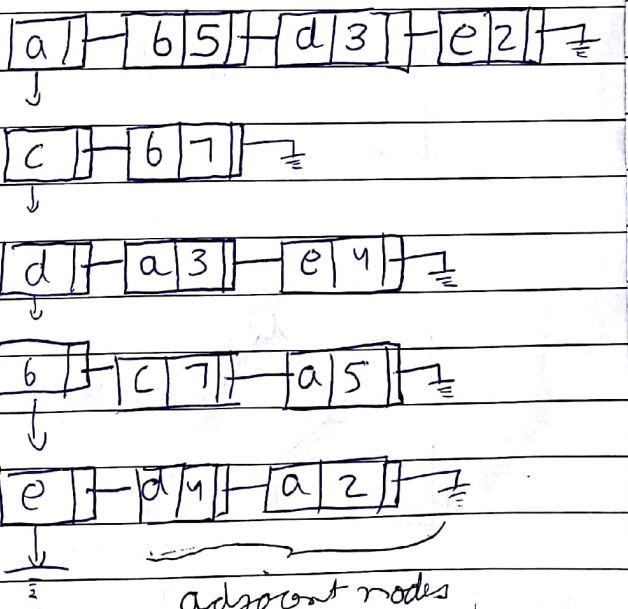
eg





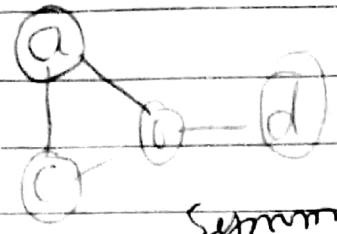
Weighted graph:





2 Sequential Representation :-

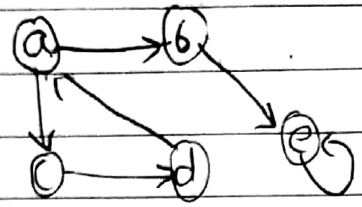
undirected graph



Symmetric

	a	b	c	d
a	0	1	1	0
b	1	0	1	1
c	1	1	0	0
d	0	1	0	0

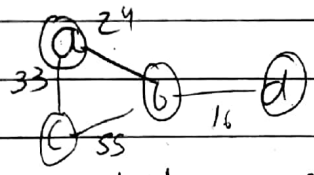
directed graph



Asymmetric

	a	b	c	d	e
a	0	1	1	0	0
b	0	0	0	0	1
c	0	0	0	1	0
d	1	0	0	0	0
e	0	0	0	0	1

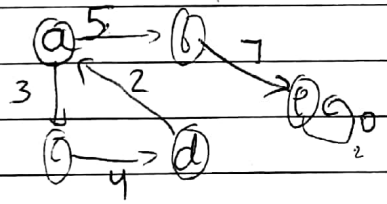
weighted undirected graph



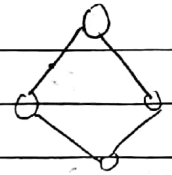
If there is no direct way or edge we write ∞

	a	b	c	d
a	0	24	33	∞
b	24	0	55	16
c	33	55	0	∞
d	∞	16	∞	0

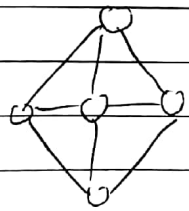
weighted directed graph



	a	b	c	d	e
a	-1	5	3	-1	-1
b	-1	-1	-1	-1	7
c	-1	-1	-1	4	-1
d	2	-1	-1	-1	-1
e	-1	-1	-1	-1	0



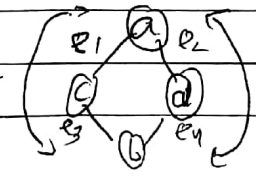
cycle



wheel

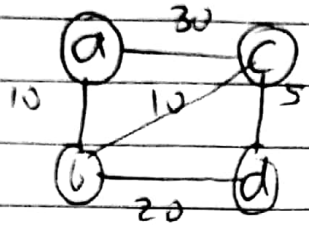
Path matrix

	e_1	e_2	e_3	e_4
P_1	0	1	0	1
P_2	1	0	1	0



20/4/18

Floyd Warshall Algorithm (Shortest Paths)



W :

	a	b	c	d
a	0	10	30	∞
b	10	0	10	20
c	30	10	0	5
d	∞	20	5	0

If Edges are in -ve we can replace 0 with ∞.

W^a =

	a	b	c	d
a	0	10	30	∞
b	10	0	10	20
c	30	10	0	5
d	∞	20	5	0

In W^a a's column and row will not change

W^b =

	a	b	c	d
a	0	10	20	30
b	10	0	10	20
c	20	10	0	5
d	30	20	5	0

In W^b b's column and row will not change

W^c =

	a	b	c	d
a	0	10	20	30
b	10	0	10	15
c	20	10	0	5
d	30	15	5	0

In W^c c's column and row will not change

W^d =

	a	b	c	d
a	0	10	20	30
b	10	0	10	15
c	20	10	0	5
d	30	15	5	0

In W^d d's column and row will not change

After W^a, W^b, W^c and W^d we will get shortest path matrix

H.W

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

Answer

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

Graph Traversal :-

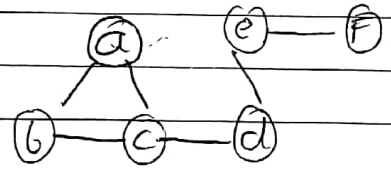
1) BFS (Breadth - First - Search)
2) DFS (Depth - First - Search)

BFS

- Look at possible path at same depth before you go at deeper level
- Queue

DFS

- Travel as far as you can down a path
- Back up as little as possible when you reach a dead end
- Stack

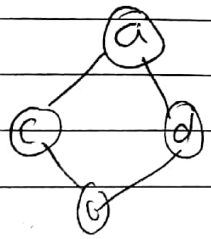


a, b, c

a, b, c, d, e, f - BFS

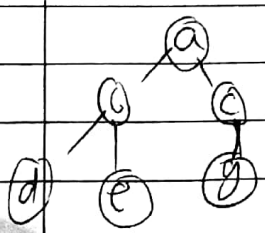
a, c, d, e, f

a, c, d, e, f, b - DFS



a, c, d, b BFS

a, c, b, d DFS

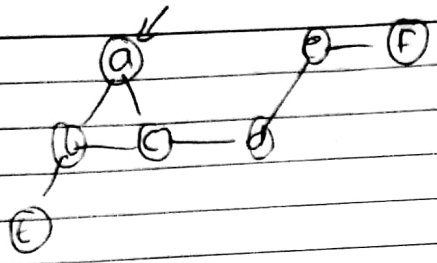


a, b, c, d, e, g BFS

or

a, b, c, g, d, e

a, c, g, b, d, e - DFS

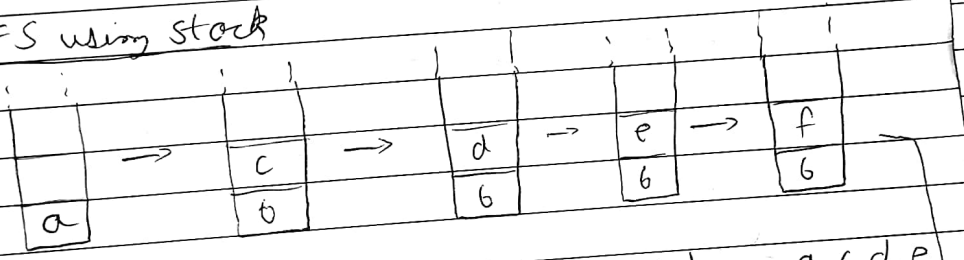


BFS using Queue

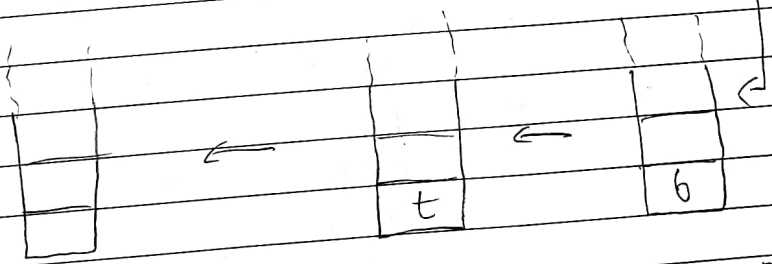
a		
b	c	
	c	t
t	d	
d		
e		
f		

a
a, b
a, b, c
a, b, c, t
a, b, c, t, d
a, b, c, t, d, e
a, b, c, t, d, e, f – BFS!

DFS using stack



a a, c a, c, d a, c, d, e



a, c, d, e, f, b, t – DFS a, c, d, e, f, b a, c, d, e, f

25/4/18

Hashing :-

To access data fast and decrease wastage of space we use concept of Hashing.

Hash function :-

$h(k) = k \bmod n$ $\quad n$ - Block size

Eg. Let Key be ID of 100 students
ID is of form 345610

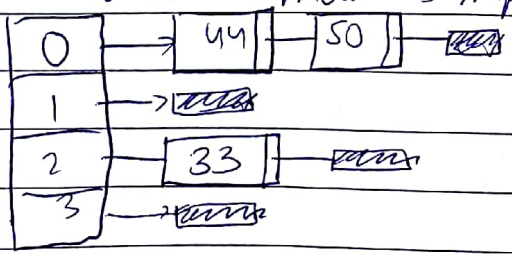
We decided to take $n = 100$
∴ Hash function says Last two digit

But if there are two students with ID
113062 and 103062
∴ Both have same location 62
This event is called Collision.

Collision Resolution :-

① Chaining :-

- For same $h(k)$ we will make a link list at specific index
- In this method is more wastage of data as some of the indexes remains empty.



__/__/__

Good Hash functions → Evenly distributed

1. Not for float values
2. To Insert characters into table we need to convert them into integers using ASCII table.

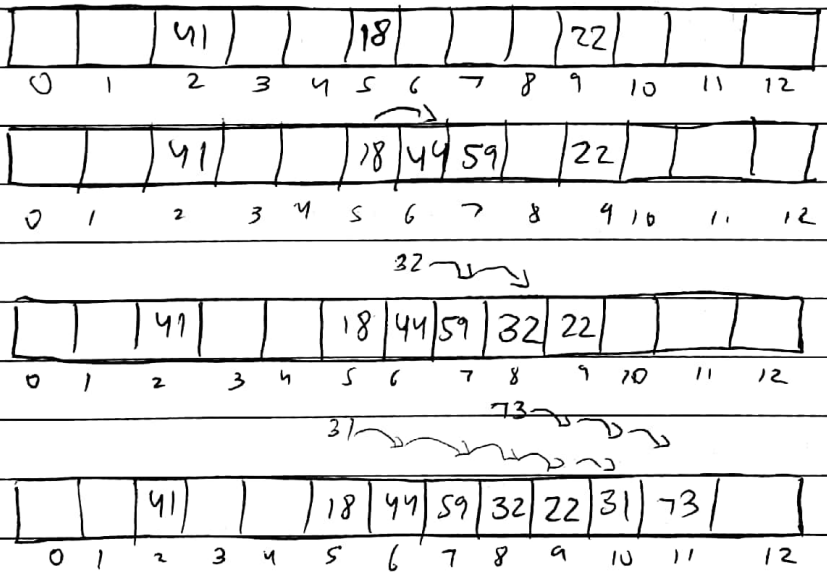
(2) Open addressing :-

a) Linear probing :- $h_i(X) = (Hash(X)+i) \% Size$

- If current location is used we try next one
- Uses less memory than chaining
- Slower as we need to search whole table.

eg 18, 41, 22, 44, 59, 32, 31, 73

$h(k) = k \bmod 13$



Algorithm :-

probe = h(k)

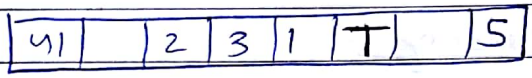
while (table[probe] is occupied)

probe = (probe + 1) % m → Size

Table[probe] = k

__/__/____

- Whenever there is a deletion in linear probing instead of setting value to null tombstone (a marker) is placed there.
- It is placed to tell to continue search as during searching whenever tombstone encounters it is ignored.



- If insert comes across T it puts element at that location and removes tombstone.

6) Quadratic probing:

Problems with linear probing:-

1) Primary clustering: Several attempts to resolve collision
2) Lazy deletion:- Find operation will fail because the link of collisions that lead to data cut.

- Quadratic probing removes Primary clustering

$$hi(X) = (Hash(X) + i^2) \% Size$$

c) Double Hashing:-

use another hash function hash2(X)

$$hi(X) = (Hash(X) + i * Hash2(X)) \% Size$$

- NO clustering

hash2(key) = PRIME – (key % PRIME) where
PRIME is a prime smaller than the TABLE SIZE.

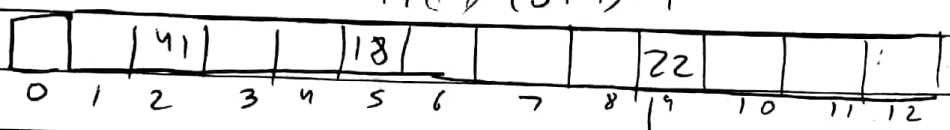
__/__/__

eg $h_1(k) = k \bmod 13$

$h_2(k) = 8 - (k \bmod 8)$

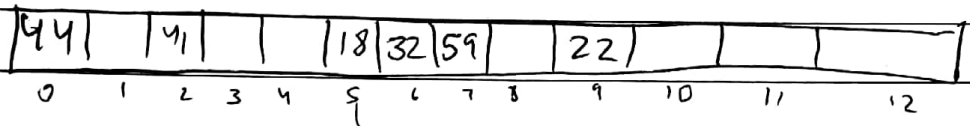
18, 41, 22, 44, 59, 32, 31, 73

$44 \; (+4) = (5+4) = 9$



9 is also occupied

$(5 + 4 \times 2) \% 13 = (5+8) \% 13 = 0$



5 is occupied for 31

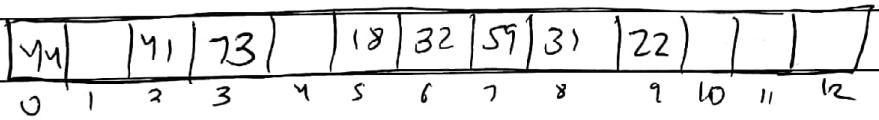
$h_1(k) = 31 \bmod 13 = 5$

$h_2(k) = 8 - 31 \bmod 8 = 1$

$h(31) = 6 \% 13 = 6$ occupied

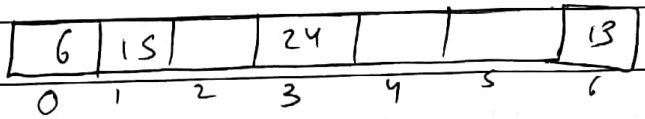
$= (5+2) \% 13 = 7$ occupied

$= (5+3) \% 13 = 8$

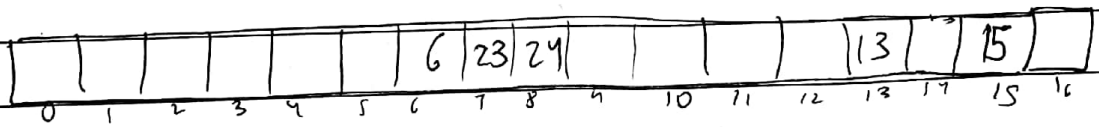


③ Rehashing :- when table half full or insertion fails

When the table size is too full, create new table at least twice as big. insert the values again with new hashfunction.



Insert 23



__/__/__

(9) Extendible hashing :-

- For fast searching
- It treats hash as bit string

We convert the integers into binary numbers and store them.

for eg 3333 → 000101

We convert the integers into no. using hash function

for eg $h(k) = k \bmod 64$

3333 → 5

1235 → 19

Afterwards these no. are converted to binary no.

For eg

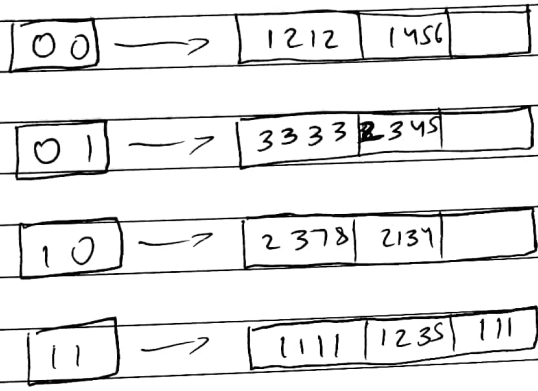
Key	Bit pattern
3333	000101
1235	010011
2378	001010
1456	111100
2134	010110
2345	101001
1111	010111
8231	100111
2222	101110
9991	001111

We have taken depth = 2 which means last two digits of bit patterns, bucket size = 3 which means at each index there is 3 available spaces.

__/__/____

1111, 3333, 1235, 2378, 1212, 1456, 2134, 1111, 8231, 2222, 9999

d = 2 Size = 3



d = 3 Size = 3