USACO GOLD

Student's Book

DO NOT SHARE

STAR LEAGUE

STAR LEAGUE INC

Copyright© by *Star League Inc*. All rights reserved.

DO NOT SHARE

Dear USACO Gold Contestant

You just passed USACO Silver level and proved that you have the ability of algorithmical thinking and you can implement basic algorithms like binary search, recursion, DFS, BFS etc. and apply these algorithms on solutions of the problems.

Now you have a USACO Gold path in front of you. Knowing the guidelines and your level on the track can help and motivate you to study and finish the USACO Gold content easily. Here is an overall guideline of the USACO Gold curriculum.

1. USACO Silver Review
2. Bitset & Binary operations
3. Data Structure
    a. Stack, Queue, Vector, Set, Map, PriorityQueue- Gold Level Applications
4. Graph Theory
    a. DFS-BFS on graphs
    b. Topological Sort
    c. Minimum Spanning Tree(Prim)
    d. Shortest Path-(Dijkstra, Bellman-Ford)
    e. All Shortest Paths(Floyd-Warshall)
5. Dynamic Programming
    a. Introduction
    b. Basic Problems(Longest Increasing Subsequence, Maximum Subarray Sum)
    c. Knapsack, Coin Change, Subset Sum
    d. DP on Strings Longest Common Sub-sequence, Edit Distance
    e. Number Game
6. Introductory Geometric Algorithms
    a. Points, Line, Slope, Intersection, In-out of shapes, Polygon area
7. Greedy Problems
    a. USACO Gold Greedy Problems

StarLeague wishes the best on your journey to the USACO Platinum level.

DO NOT SHARE

Silver Review

USACO Gold contestant must be implementing these problems comfortably.

- Sorting & Searching Algorithms (Binary Search)
- Complete Search / Brute Force
- Recursion
- DFS / BFS
- Floodfill

Broken Necklace:

A nice example for Brute force / Complete Search problem.

N<=350 Beads.

Trying all possible positions to break the necklace and count number of beads that can be collected is $O(N^2)$ which is good enough.

		b	r	b	r	r	r	b	b	b	r	r	r	r	r	b	r	r	b	b	r	b	b	b	b	r	r	r	r	b
left	1	2	1	1	1	2	3	1	2	3	1	2	3	4	5	1	1	2	1	2	1	1	2	3	4	1	2	3	4	1
right	1	1	1	3	2	1	3	2	1	5	4	3	2	1	1	2	1	2	1	1	4	3	2	1	4	3	2	1	2	1

```
for (pos:0->len) {
   res= count(pos, right)  + count (pos-1, left)
   best=  max(best, res);
}

int count(pos, dir) {
   if right dir = 1
   else dir=-1
   //detect color
   while (a[pos+x]=='w'
      x+=dir;
   color = a[pos+x]
   //Count number of beads
   while (a[pos+x]=='w' || a[pos+x]==color
      x+=dir;
   return (x); //if nefative return -x
}
```

After getting number of beads to left and right finding th max total is the answer to the problem.

DO NOT SHARE

Balanced Teams

This problem is a nice sample for Silver Recursion problems. Implement a `solve(cow)` recursion function which will try all possible teams for current cow and call the function for the next cow. Of course the base case when all the cows assigned with a team.

```
solve (int cow)
   if (cow == N) {
      calculateTeamPowers();
      ans= min(ans, maxTeam- minTeam)
      return
   }
   for (team:0->4) {
      if (numberofPlayers[team]<3) {
         assign palyer to team
         solve(next cow)
      }
   }
```

Recursion will assign following teams to the players.

Cow id	1	2	3	4	5	6	7	8	9	10	11	12
Teams[]	1	1	1	2	2	2	3	3	3	4	4	4
	1	1	1	2	2	2	3	3	4	3	4	4
	1	1	1	2	2	2	3	3	4	4	3	4
	1	1	1	2	2	2	3	3	4	4	4	3
	1	1	1	2	2	2	3	4	3	3	4	4
	1	1	1	2	2	2	3	4	3	4	3	4
	1	1	1	2	2	2	3	4	3	4	4	3

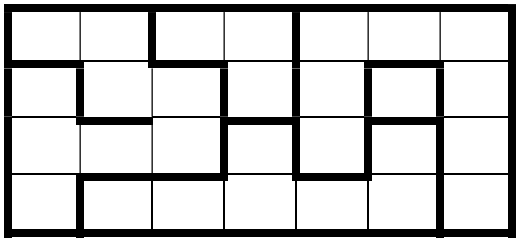
This is another Complete Search problem implemented using Recursion.

DO NOT SHARE

The Castle – Adopted From IOI'94

This is a problem that contains many of silver subjects. It has recursion, floodfill, bits, etc.

Given map of a castle. You are asked to find the number of rooms, largest room and what can be the max room size if you remove one of the walls.



This is a straightforward floodfill application.

1-Floodfill the rooms while counting them.

```
roomno=1;
for (i:0->N)
   for (j:0->M)
      if (!visited[i,j])
         floodfill(i,j,++roomno);


floodfill(r,c,roomno) {
   visited[r,c] = roomno;
   roomsize[roomno]++;        //new cell in the same room. Increment roomsize
   for (each neighbor cell)
      if (nowall(cell,neighbor) & not visited[neighbor])
         floodfill(neighbor, roomno)
}
```

Above code will genarate following visited and roomsize arrays.

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

roomsize:

9	3	8	1	7

How to find Max roomsize after removing one wall?

Traverse entire visited matrix. If you see a cell with neighboring a different roomno, check if total of the sizes of the two rooms is bigger than the maximum.

DO NOT SHARE

```
for (i:0->N)
    for (j:0->N)
       if (visited[i,j] != visited[neighbor])
          ans = max(ans, roomsize[visited[i,j]]+roosize[visited[neighbor]]
```

How to check existence of the walls?

1	West		2	North		4	East		8	South
0001	1		0010	2		0100	4		1000	8
0011	3		0011	3		0101	5		1001	9
0101	5		0110	6		0110	6		1010	10
0111	7		0111	7		0111	7		1011	11
1001	9		1010	10		1100	12		1100	12
1011	11		1011	11		1101	13		1101	13
1101	13		1110	14		1110	14		1110	14
1111	15		1111	15		1111	15		1111	15
(n%2)			((n/2)%2)			((n/4)%2)			((n/8)%2)	

Testing the last bit of an integer is same as testing the modulo 2 of the number.

Testing the 2nd least significant bit of an integer. Shift right (by dividing with 2)Now it is the last bit use previous method to test its value.

Testing the 3rd least significant bit. Shift right twice (by dividing with 4) Now it is the last bit. Test the last bit with modulo 2.

…

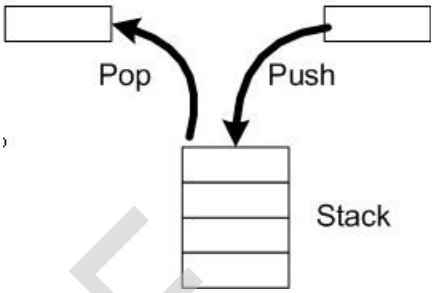
```
    if (castle[i][j] %2 ==0)     //no wall to west
    if ((castle[i][j]/2) %2 ==0) //no wall to north
    if ((castle[i][j]/4) %2 ==0) //no wall to east
    if ((castle[i][j]/8) %2 ==0) //no wall to south
```

DO NOT SHARE

Data Structures

Stack Data Structure

A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.



- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "pushing" onto the stack. "Popping" off the stack is synonymous with removing an item.

The order may be LIFO(Last In First Out). A PEZ® dispenser as an analogy:

Stack supports two main methods:

push(o): Inserts object o onto top of stack
Input: Object; *Output*: none
pop(): Removes the top object of stack and returns it; if stack is empty an error occurs
Input: none; *Output*: Object

The following support methods should also be defined:

size(): Returns the number of objects in stack
Input: none; *Output*: integer
isEmpty(): Return a boolean indicating if stack is empty.
Input: none; *Output*: boolean
top(): return the top object of the stack, without removing it; if the stack is empty an error occurs.
Input: none; *Output*: Object

How to understand a stack practically?
There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO order.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

- Balancing of symbols
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem, Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver.

Implementation:
Stack can be implemented using an array, also there is built in Stack data structure in programming languages.

DO NOT SHARE

Array Based Stack

Create a stack using an array by specifying a maximum size N for our stack, e.g. N = 1,000.

The stack consists of an N-element array S and an integer variable t, the index of the top element in array S

Array indices start at 0, so we initialize t to -1

Pseudo-code

```
Algorithm size():
  return t +1

Algorithm isEmpty():
  return (t<0)

Algorithm top():
  if isEmpty() then
    throw a StackEmptyException
  return S[t]

Algorithm push(o):
  if size() = N then
    throw a StackFullException
  t ← t + 1
  S[t] ← o

Algorithm pop():
  if isEmpty() then
    throw a StackEmptyException
  e←S[t]
  S[t]←null
  t←t-1
  return e
```

- Each of the above method runs in constant time(O(1))
- The array implementation is simple and efficient.
- There is an upper bound, N, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.

DO NOT SHARE

C++ Imlementation arrays

```cpp
#define MAX_SIZE 101
// integer array to store the stack
int A[MAX_SIZE];
// variable to mark top of stack in array
int top = -1;

// Push operation to insert an element on top of stack.
void Push(int x) {
  if(top == MAX_SIZE -1) { // overflow case.
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}

// Pop operation to remove an element from top of stack.
void Pop() {
    // If stack is empty, pop should throw error.
    if(top == -1) {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}

// Top operation to return element at top of stack.
int Top() {
    return A[top];
}

// This function will return 1 (true)
//if stack is empty, 0 (false) otherwise
int IsEmpty(){
    if(top == -1) return 1;
    return 0;
}
```

Java Implementaion using arrays

```java
/* Java program to implement basic stack
   operations */
class Stack
{
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = -1;
    }

    boolean push(int x)
    {
        if (top >= MAX)
        {
            System.out.println("Stack Overflow");
            return false;
        }
        else
        {
            a[++top] = x;
            return true;
        }
    }

    int pop()
    {
        if (top < 0)
        {
            System.out.println("Stack Underflow");
            return 0;
        }
        else
        {
            int x = a[top--];
            return x;
        }
    }
}
```

DO NOT SHARE

Riddle until next lesson:

Re implement the stack with min() function which will return minimum elemnet in the stack.

version 1: Implement it in O(N) Easy, starightforward

version 2: implement it in O(1). (Interview question)

References

https://gist.github.com/mycodeschool/6878252

http://www.dcs.gla.ac.uk/~pat/52233/slides/StacksQueuesLists1x1.pdf

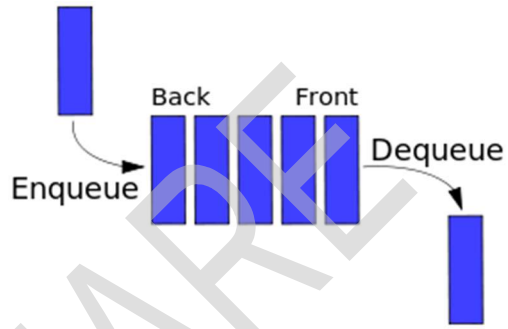
https://www.geeksforgeeks.org/stack-data-structure-introduction-program/

Queue

A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle. A good example of queue is any queue of consumers for a resource where the consumer that came front is served front.

•Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
•Elements are inserted at the *rear* (enqueued) and removed from the *front* (dequeued)

The queue supports two fundamental methods:



enqueue(*o*): Insert object *o* at the rear of the queue
Input: Object; *Output*: none

dequeue(): Remove the object from the front of the queue and return it; an error occurs if the queue is empty
Input: none; *Output*: Object

These support methods should also be defined:
size(): Return the number of objects in the queue
Input: none; *Output*: integer
isEmpty(): Return a boolean value that indicates whether the queue is empty
Input: none; *Output*: boolean
front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty
Input: none; *Output*: Object

Array based queue

- Create a queue using an array in a circular fashion
- A maximum size N is specified, e.g. N = 1,000.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element
  - r, index of the element after the rear one
- "normal configuration"

- "wrapped around" configuration

- what does f=r mean?

DO NOT SHARE

Pseudocode

```
Algorithm size():
  return (N - f + r) mod N
Algorithm isEmpty():
  return (f = r)
Algorithm front():
  if isEmpty() then
    throw a QueueEmptyException
  return Q[f]
Algorithm dequeue():
  if isEmpty() then
    throw a QueueEmptyException
  temp ← Q[f]
  Q[f] ← null
  f ← (f + 1) mod N
  return temp
Algorithm enqueue(o):
  if size = N - 1 then
    throw a QueueFullException
  Q[r] ← o
  r ← (r +1) mod N
```

DO NOT SHARE

C++ Imlementation arrays

```cpp
#define SIZE 10
int queue[SIZE], front = 0, rear = 0;

void enQueue(int value){
      queue[rear++] = value;
}

void deQueue(){
      return queue[front++];
}

//using a circular queue

void enQueue(int value)
{
  queue[rear] = value;
  rear = (rear + 1) % SIZE;
}

int deQueue()
{
  int x = queue[front];
  front = (front + 1) % SIZE;
  return x;
}
```

Java Implementaion using arrays

```java
// Java program for array implementation of queue

// A class to represent a queue
class Queue
{
    int front, rear, size;
    int  capacity;
    int array[];

    public Queue(int capacity) {
         this.capacity = capacity;
         front = this.size = 0;
         rear = capacity - 1;
         array = new int[this.capacity];

    }

    // Queue is full when size becomes equal to
    // the capacity
    boolean isFull(Queue queue)
    {  return (queue.size == queue.capacity);
    }

    // Queue is empty when size is 0
    boolean isEmpty(Queue queue)
    {  return (queue.size == 0); }

    // Method to add an item to the queue.
    // It changes rear and size
    void enqueue( int item)
    {
        if (isFull(this))
            return;
        this.rear = (this.rear + 1)%this.capacity;
        this.array[this.rear] = item;
        this.size = this.size + 1;
        System.out.println(item+ " enqueued to queue");
    }

    // Method to remove an item from queue.
    // It changes front and size
    int dequeue()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        int item = this.array[this.front];
        this.front = (this.front + 1)%this.capacity;
        this.size = this.size - 1;
        return item;
    }

    // Method to get front of queue
    int front()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        return this.array[this.front];
    }
```

DO NOT SHARE

References:

http://www.dcs.gla.ac.uk/~pat/52233/slides/StacksQueuesLists1x1.pdf
https://www.geeksforgeeks.org/queue-using-stacks/
https://introcs.cs.princeton.edu/java/43stack/

Puzzle of Week (Implement Stack min in O(1))

Operation	Stack						S.min()	minStack		
s push(10)	10						10	10		
S.push (7)	10	7					7	10	7	
s.push (12)	10	7	12				7	10	7	
S.push (6)	10	7	12	6			6	10	7	6
S.push (15)	10	7	12	6	15		6	10	7	6
s.pop()	10	7	12	6			6	10	7	6
s.push (9)	10	7	12	6	9		6	10	7	6
s.pop()	10	7	12	6			6	10	7	6
s.pop()	10	7	12				7	10	7	
s.pop()	10	7					7	10	7	
s.pop()	10						10	10		

I. Use a second stack to keep the minimum

```
void push (int x)
{
    if ((os.empty()) || (x <= smin.top()))
        os.push(x);
        smin.(x)
    }
    else
        os.push(x);
}
```

```
int pop()
{
    int x = os.top();
    if (x == smin.top())
        sminpop()
    os.pop()
    return x;
}
```

```
int min()
{
    return smin.top()
}
```

II. Use a stack of pair.

```
void push(x)
{
    if ((s.empty() ||) (x <= (s.top())->second)
        s.push({x, x});
    else
        s.push({x, (s.top())->second})
}
```

```
int pop()
{
    int x = s.top()->first;
    s.pop();
    return x;
}
```

```
int min
{
    return (s.top()->second);
}
```

DO NOT SHARE

Problem: 3 operations defined on a list. Add, Remove and Find.

Given N consecutive operations, You are asked to output result of Find operations as Yes/No.

(No multiple instances of any value. Remove operation removes value completely)

Sample Input:	Sample output
8	
Add 3	Yes
Add 8	No
Add 3	Yes
Find 8	
Add 9	
Remove 3	
Find 3	
Find 9	

Based on N our solution approach changes.

I. $N \leq 20.000$ $O(N^2)$ solution is acceptable.

Keep the data in array.

	Insert	Find	Remove
Cost	$O(1)$	$O(N)$	$O(N)$
			$O(1)$? how move the last element to position.

$N * O(A + FN + R)$ $O(N^2)$

We are safe both Memory and Time Complexity.

Tricky Solution:

If you know the limit of the numbers that will appear on the list $\leq 10^6$

$\Rightarrow$ Keep a boolean array for inverse index.

Add 123.	$\Rightarrow$	A[123]=1;	$O(1)$
Find 124	$\Rightarrow$	if (A[124]==1)	$O(1)$
Remove 125	$\Rightarrow$	A[125]=0;	$O(1)$

$N \times O(1, 1, 1) \Rightarrow O(N)$

DO NOT SHARE

II $N \leq 100.000$

Memory ok.

$A, F, R < 100.000 \Rightarrow$ Array does not work

$O(A + FN + R) = O(N^2)$

q) How to decrease cost of Find to $\log N$?

Keep array sorted

q) How do A, R change when we have array sorted?

	Insert	Find	Remove
Cost	$\log N + N$	$\log N$	$\log N + N$

$N\ O(A + F + R)$

$N \times (N + \log N,\ \log N,\ N + \log N)$

$O(N^2 + N \log N) \Rightarrow$ worse than the worst solution.

Too much motivated to keep array sorted, we lost track of efficiency.

$\Rightarrow$ We need a data structure

$\Rightarrow$ keeps data sorted (Finds in $\log N$)

$\Rightarrow$ Inserts and Removes elements in $\log N$

$\Rightarrow$ Keeps unique elements.

$\Rightarrow$ That is Set.

	Insert	Find	Remove
Cost	$\log N$	$\log N$	$\log N$

$\Rightarrow N\ O(A, F, R)$

$N \times O(\log N, \log N, \log N) \Rightarrow O(N \log N)$

DO NOT SHARE

Set in C++

std::set is an associative container and header that need to be include for it is, *#include<set>*

- It's doesn't allow duplicate elements i.e. it only contains unique elements.
- Std::set can contain element of any specified type in template argument i.e.
- std::set internally store elements in balanced binary tree. That is Insert, Find and Remove takes O(logN)
- By default std::set uses the operator < for comparing two elements and but if user passes the external sorting criteria i.e. comparator then it uses that instead of default operator <.[1]

```
#include <iostream>
#include <set>
#include <iterator>
using namespace std;

int main()
{
    // empty set container
    set <int, greater <int> > gquiz1;
    // insert elements in random order
    gquiz1.insert(40);
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50); // only one 50 will be added to the set
    gquiz1.insert(10);
    // printing set gquiz1
    set <int, greater <int> > :: iterator itr;
    cout << "\nThe set gquiz1 is : ";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr){
        cout << '\t' << *itr;
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    set <int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the set gquiz2
    cout << "\nThe set gquiz2 after assign from gquiz1 is : ";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr){
        cout << '\t' << *itr;
    }
    cout << endl;

    // remove all elements up to 30 in gquiz2
    cout << "\ngquiz2 after removal of elements less than 30 : ";
    gquiz2.erase(gquiz2.begin(), gquiz2.find(30));
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr){
        cout << '\t' << *itr;
    }

    // remove all elements with value 50 in gquiz2
    int num;
    num = gquiz2.erase (50);
    cout << "\ngquiz2.erase(50) : ";
    cout << num << " removed \t" ;
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr){
        cout << '\t' << *itr;
    }
    cout << endl;
    //lower bound and upper bound for set gquiz1
    cout << "gquiz1.lower_bound(40) : " << *gquiz1.lower_bound(40) << endl;
    cout << "gquiz1.upper_bound(40) : " << *gquiz1.upper_bound(40) << endl;

    //lower bound and upper bound for set gquiz2
    cout << "gquiz2.lower_bound(40) : " << *gquiz2.lower_bound(40) << endl;
    cout << "gquiz2.upper_bound(40) : " << *gquiz2.upper_bound(40) << endl;
    return 0;
}
```
[2]

The output of the above program is :

[1] http://thispointer.com/stdset-tutorial-part-1-set-usage-details-with-default-sorting-criteria/
[2] https://www.geeksforgeeks.org/set-in-cpp-stl/

DO NOT SHARE

```
The set gquiz1 is :      60    50    40    30    20    10

The set gquiz2 after assign from gquiz1 is : 10      20    30    40    50    60

gquiz2 after removal of elements less than 30 : 30    40    50    60
gquiz2.erase(50) : 1 removed     30    40    60
gquiz1.lower_bound(40) : 40
gquiz1.upper_bound(40) : 30
gquiz2.lower_bound(40) : 40
gquiz2.upper_bound(40) : 60
```

Set in Java[3]

- Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
- Basically, Set is implemented by HashSet, LinkedSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

```
import java.util.*; public class Set_example
{    public static void main(String[] args)
   {
      Set<String> hash_Set =new HashSet<String>();
       hash_Set.add("Geeks");
       hash_Set.add("For");
       hash_Set.add("Geeks");
       hash_Set.add("Example");
       hash_Set.add("Set");
      System.out.print("Set output without the duplicates");

      System.out.println(hash_Set);

      System.out.print("Sorted Set after passing into TreeSet");
      Set<String> tree_Set =new TreeSet<String>(hash_Set);
      System.out.println(tree_Set);
   }
}
```

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

Output:

Set output without the duplicates[Geeks, Example, For, Set]
Sorted Set after passing into TreeSet[Example, For, Geeks, Set]

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

3 HTTPS://WWW.GEEKSFORGEEKS.ORG/SET-IN-JAVA/

DO NOT SHARE

Map · Intro

Wednesday, January 31, 2018 5:29 PM

Problem: Given N numbers between $0..10^9$. Count occurences of each value.

Sample Input:

8
3
9
3
8
10
9
8
11
10
10

Sample output:

8	3
3	2
9	2
10	3
11	1

Idea I. Using Array

Put the numbers in an array. Starting from first element count each value and remove them from array (change to -1)

8	3	9	3	8	10	9	8	11	10	10
-1	3	9	3	-1	10	9	-1	11	10	10
-1	-1	9	-1	-1	10	9	-1	11	10	10
-1	-1	-1	-1	-1	10	-1	-1	11	10	10

⋮

8	3
3	2
9	2

⋮

Time Complexity $O(N^2)$ ⇒ Solution is valid while $N \leq 20.000$

Idea II: Inverse indexing

Use an array and inverse index each value used.

0	1	2	3	...	8	9	10	11	...
			2	.....	3	2	3	1	

In order to use values as indexes of array we have to declare an array of that size.

A[999 999 999] needs an array of size 10^9.

⇒ We need an array of size max. number in the list

⇒ This solution is not applicable under USACO memory limits.

DO NOT SHARE

Map

A map combines a key and a value to efficiently store/access data.

Map<int, int> A; // stores a value for each integer.

A[8] = 3; stores only (8,3) as (Key, Value) pair which is only 2 integers.
it would need array of size 9 integers at least.

A[999999999] = 1; An array like this would need 10^9 int memory space,
map avoids this, stores 2 integers. (Key, Value) pair only.

Java Map Implementation

```
HashMap<Integer,Integer> hmap= new HashMap<>();
read N
for (i:0->N)
{
    read num
    //if the number did not appear before
    if (!hmap.containsKey(num))
       //add num to map with value 1
       hmap.put(num,1);
    //num was seen before
    else
    //increment value by 1
       tmap.put(x, tmap.get(x)+1);
}
```

C++ Map Implementation

```
unordered_map<int,int> count;
read N
for (i:0->N)
{
    read num
    //if the number did not appear before
    if (count.find(num)==count.end())
       //add num to map with value 1
       count[num]=1;
    //num was seen before
    else
    //increment value by 1
       count[num]++;
}
```

Data Structure you can use

Limit	Time Complexity	C++	java
$N \leq 20.000$	$O(N^2)$ acceptable	Array	Array
$N \leq 100.000$	$O(N \log N)$	map	TreeMap
$N \leq 1.000.000$	$O(N)$	unordered_map	HashMap

DO NOT SHARE

Maps C++ :

> C++ Standard Template Library provides maps and sets which are implemented internally using balanced red black trees. We explore maps here for now, although set is very much similar.

Map declaration :

```
map<int, int> A; // O(1) declaration which declares an empty tree map.
```

Insert a new key, value pair K, V:

```
A[K] = V; // O(log n). Note that we expect key K to be unique here. If you have keys that will repeat, take a look at multimaps.
```

Delete a key K:

```
A.erase(K); // O(log n)
```

Search if a key K exists in map:

```
A.find(K) != A.end()  // O(log n)
```

Search for the value with key K:

```
A[K]    // O(log n)
```

Find minimum key K in the map (if the map is not empty) :

```
(A.begin())->first    // O(1)
```

Find maximum key K in the map (if map is not empty) :

```
(A.rbegin())->first    // O(1)
```

Iterate over keys in sorted order :

```
for (map<int,int>::iterator it = A.begin(); it != A.end(); ++it) {        // it->first has the key, it->second has the value.  }
```

Find closest key K > x :

```
(A.upper_bound(x))->first    // O(log n). Do need to handle the case when x is more than or equal to the max key in the map.
```

Find closest key K >= x :

```
(A.lower_bound(x))->first    // O(log n). Do need to handle the case when x is more than the max key in the map.
```

Size (number of entries in the map) :

```
A.size()
```

DO NOT SHARE

Maps JAVA

Java has TreeMap which is very similar to C++ map. Also implemented using Red Black trees.

Map declaration :

```
TreeMap<Integer, Integer> A = new TreeMap<Integer, Integer>(); // O(1) declaration which declares an empty tree map.
```

Insert a new key, value pair K, V:

```
A.put(K, V); // Note that we expect key K to be unique here. If you have keys that will repeat, they will be replaced.    // O(log n)
```

Delete a key K:

```
A.remove(K); // O(log n)
```

Search if a key K exists in map:

```
A.containsKey(K)  // O(log n)
```

Search for the value with key K:

```
A.get(K)     // O(log n)
```

Find minimum key K in the map (if the map is not empty) :

```
A.firstKey() OR A.firstEntry().getKey()     // O(1)
```

Find maximum key K in the map (if map is not empty) :

```
A.lastKey() OR A.lastEntry().getKey()     // O(1)
```

Iterate over keys in sorted order :

```
for (Map.Entry<Integer, Integer> entry : A.entrySet()) {
    // entry.getKey() has the key, entry.getValue() has the value
}
```

Find closest key K >= x :

```
A.ceilingEntry(x).getKey()     // O(log n). Do need to handle the case when x is more than the max key in the map.
```

Find closest key K <= x :

```
A.floorEntry(x).getKey()     // O(log n). Do need to handle the case when x is smaller than min key in the map
```

Size (number of entries in the map) :

```
A.size()
```

DO NOT SHARE

C++ :

C++ has unordered_map which is implemented via hashing. We will discuss maps in C++ as well later, which in practice seem to behave faster in C++.

Declaration

```
unordered_map<int, int> A; // declares an empty map. O(1)
```

Adding an key, value pair :

```
A.insert(key, value); // O(1) time on average
```

Find the value for key = K:

```
if (A.find(K) == A.end()) return null; // means that K does not exist in A.
else return A[K];      // O(1) average case. Rare worst case O(n).
```

Size (number of elements) of the vector :

```
A.size()   // O(1)
```

Erase from the map :

```
if (A.find(K) != A.end()) A.erase(A.find(K));
  OR A.erase(K);
```

DO NOT SHARE

JAVA :

Declaration :

```
HashMap<Integer, Integer> A = new HashMap<Integer, Integer>(); // declares an empty map.
```

Adding an key, value pair :

```
A.put(key, value); // O(1) time on average
```

Find the value for key = K:

```
A.get(K) // null if the key K is not present
A.containsKey(K) tells if the key K is present.
// Both operations O(1) average time. O(n) rare worst case
```

Size (number of elements) of the vector :

```
A.size()   // O(1)
```

Erase from the map :

```
A.remove(K);
```

DO NOT SHARE

Priority Queue

Wednesday, February 21, 2018 2:01 PM

```
priority_queue<int> pq;

pq.push(3); pq.push(7); pq.push(1);

while (!pq.empty())
    cout<< pq.top()<<endl;
    pq.pop();
```

Output:

7
3
1

```
PriorityQueue<String> pq=new PriorityQueue<String>();

pq.add("Bob");
pq.add("Alice");
pq.add("Denis");
pq.add("Charlie");

while (pq.size()!=0)
    System.out.println(pq.remove)
```

Output:

Alice
Bob
Charlie
Denis

Priority Queue member functions

```
pq.empty()
pq.size()

pq.push(x)
pq.top()      retrieves value. does not remove
pq.pop()      removes the top. does not return
```

```
pq.empty()
pq.size()

pq.add(x)
pq.poll()     retrieves and removes the top
pq.peek()     retrieves the top. does not remove it
```

q) What if you need priority-queue of your class type and you need to write your comparison

```
Struct Point
  {
    int x,y;
  };

  bool operator < (Point a, Point b)
  {
    if a.x == b.x
        return (a.y < b.y)
    return a.x < b.x
  }
```

```
class Element implements Comparable <Element>
{
    int rating;

    public int compareTo (Element other)
    {
        return this.rating - other.rating
    }
```

DO NOT SHARE

Fighters(PQ)

There are n fighters and each fighter has a rating. When 2 fighters fight, the one having bigger rating always beat and gets other's rating. The loser leaves the game. A tv channel wants to arrange and show matches, they have to pay to the players as much as their ratings. TV channel wants to maximize the total profit by decreasing total cost. According to given input, what is minimum cost the TV channel should pay to the players? You are given N (number of fighters) and then N integers as the ratings of the fighters.

input:	input:
3 1 1 1	5 2 2 3 1 4
output:	output:
5	27

q1) How many matches there will be?

each match eliminates 1 fighter.

n-1 fighters will be eliminated ⇒ n-1 matches

q2) What should be best strategy to select fighters?

The cost of the match is total of the ratings.

⇒ It is meaningless to use higher cost fighters more while there are lower cost fighters.

⇒ Strategy must be choose always minimum cost 2 fighters.

q3) Which data structure is ideal? (we need min(), insert(x) operations N times)

Multiset, Map or Priority_Queue are ideal data structures.

they give min in O(1)

insert in O(log N)

DO NOT SHARE

C++ Solution

```cpp
#include <bits/stdc++.h>
 using namespacestd;
int main() {
     priority_queue<int> pq;

    int n,cost=0,x,f1,f2;
    cin>>n;
    for (int i=0;i<n;i++) {
         cin>>x;
         pq.push(-x);
    }

//Solution using Priority Queue
    while (pq.size()) {
         f1= pq.top(); pq.pop();
         f2= pq.top(); pq.pop();
         cost+=f1+f2;
         pq.push(f1+f2);
    }
    cout << -cost<< endl;
     return 0;
}
```

Java Solution

```java
import java.util.Scanner;
import java.util.PriorityQueue;

public class Fighters {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
         Scanner sc = new Scanner(System.in);
         int n=sc.nextInt();
         for (int i=0;i<n;i++) {
           int x=sc.nextInt();
           pq.add(x);  //for priority_queue implementation
         }
         //Solution using Priority queue
        int ans=0;
        while (pq.size() > 1) {
           int match = pq.poll() + pq.poll();
           pq.add(match);
           ans+= match;
        }
         System.out.println(ans);
    }
}
```

q) Could we use Set instead of Priority Queue

Set also keeps elements sorted also has complexity find min item in O(1). Insert, delete, find O(logN). But it keeps one instance for each value. C++ users can use multiset, which is

```
#include <bits/stdc++.h>
using namespacestd;
int main() {
    multiset<int> s;
    int n,cost=0,x,f1,f2;
    cin>>n;
    for (int i=0;i<n;i++) {
        cin>>x;
        s.insert(x);
    }
    //Solution using MultiSet
    while (s.size()>1) {
        f1= *s.begin();
        s.erase(s.begin());
        f2= *s.begin();
        s.erase(s.begin());
        cost+=f1+f2;
        s.insert(f1+f2);
    }
    cout << cost<< endl;
}
```

Java users can switch to C++, If they want to benefit multiSet ☺

No MultiSet for Java.

Java users can simulate Multiset using a TreeMap. Here is an implementation to same problem using TreeMap.

C++ Solution	Java Solution

`#include <bits/stdc++.h>`
` using namespacestd;`
`int main() {`
`  map<int,int> m;`
 	
`  int n,cost=0,x,f1,f2;`
`  cin>>n;`
`  for (int i=0;i<n;i++) {`
`  cin>>x;`
`  if (m.find(x)==m.end())`
`  m[x]=0;`
`  m[x]++;`
`  }`
`  //Solution using MAP`
`  while (!m.empty()) {`
`  f1= m.begin()->first;`
`  m[f1]--;`
`  if (m[f1]==0)`
`  m.erase(m.find(f1));`
`  if (m.empty()) break;`
`  f2= m.begin()->first;`
`  m[f2]--;`
`  if (m[f2]==0)`
`  m.erase(m.find(f2));`
`  cost+=f1+f2;`
`  if (m.find(f1+f2)==m.end())`
`  m[f1+f2]=0;`
`  m[f1+f2]++;`
`  }`
`  cout<<cost<<endl;`
`}`
	`import java.util.Scanner;`
	`import java.util.TreeMap;`
	`public class Fighters {`
	`  public static void main(String[] args) {`
	`  TreeMap<Integer, Integer> tmap = new TreeMap<>();`
	`  Scanner sc = new Scanner(System.in);`
	`  int n=sc.nextInt();`
	`  for (int i=0;i<n;i++) {`
	`  int x=sc.nextInt();`
	`  if (!tmap.containsKey(x)) tmap.put(x, 1);`
	`  else tmap.put(x, tmap.get(x)+1);`
	`  }`
	`  //solution using TreeMap`
	`  while(tmap.size()>=1) {`
	`  int f1=tmap.firstKey(); //get smallest value`
	`  tmap.put(f1,tmap.get(f1)-1); //remove from map`
	`  if (tmap.get(f1)==0) tmap.remove(f1);`
	`  if (tmap.size()==0) break;`
	`  int f2=tmap.firstKey(); //get second smallest value`
	`  tmap.put(f2,tmap.get(f2)-1); //remove from map`
	`  if (tmap.get(f2)==0) tmap.remove(f2);`
	`  int cost=f1+f2;`
	`  if (!tmap.containsKey(cost)) tmap.put(cost, 1);`
	`  else tmap.put(cost, tmap.get(cost)+1);`
	`  ans+=cost; //add the match cost to total`
	`  }`
	`  System.out.println(ans);`
	`  }`
	`}`

DO NOT SHARE

Cow Dance

Wednesday, February 21, 2018 6:52 PM

$N \leq 10{,}000$ cows

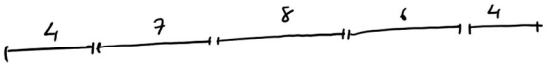
K size of stage for K cows.

each cow dances for d_i duration.

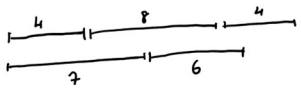
Show ends when the last cow completes at time T.

T_{max} is given. What is smallest K that allows to finish before T_{max}.

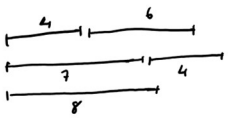
$N=5$ $T_{max}=8$ 4 7 8 6 4

K=1 

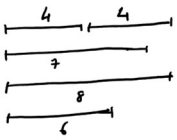
$= 29$

K=2 

$= 16$

K=3 

$= 11$

K=4 

$= 8 \leq T_{max}$

We have to solve two problems

1. If a K value is valid
2. How to find the smallest K value efficiently.

1) Find if a K value is valid

Simulate cows dance on stage. When K cows on the stage we need to know the earliest time a cow will leave stage (Smallest number among K numbers) $\Longrightarrow$ Use a priority_queue

```
bool isok(int K)
{   priority_queue<int, vector<int>, greater<int>> pq;
    cur=0;
    for (i: 0 -> n)
    {
        if (pq.size()==k) {      // stage is full
            cur = pq.top();      // earliest time a cow will leave stage
            pq.pop();
        }
        pq.push(cur + d[i])
        if (cur + d[i] > Tmax) return false
    }
    return True;
}
```

```
bool isok(int K)
{
    multiset<int> s;
    cur=0;
    for (i: 0 -> n)
    {
        if (s.size() == k)
        {
            cur = *s.begin();
            s.erase(s.begin());
        }
        s.insert(cur+d[i])
        if cur+d[i] > Tmax return false
    }
    return true
}
```

DO NOT SHARE

2) Find smallest K value

a)

```
for (k: 1 → N)
    if isok(k)
        cout << k
        break
```

Time Complexity: $O(N\,N\log N)$

$O(N^2 \log N)$

not good enough since $N = 10^4$

b) Find ideal K value using binary Search

```
min = 1;   max = n;
while (min < max) {
    mid = (min + max)/2
    if (isok(mid)  max = mid;
    else  min = mid + 1;
}
```

Time Complexity: $O(\log N\ N\log N)$

$O(N \log^2 N)$

solution is ok now ✓

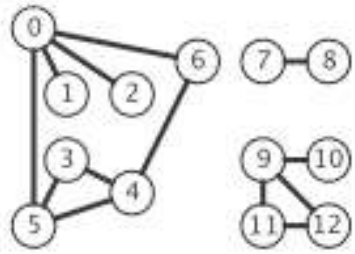
DO NOT SHARE

Graphs

Pairwise connections between items play a critical role in a vast array of computational applications. The relationships implied by these connections lead to a host of natural questions: Is there a way to connect one item to another by following the connections? How many other items are connected to a given item? What is the shortest chain of connections between this item and this other item? The table below illustrates the diversity of applications that involve graph processing.

application	item	connection
map	intersection	road
web content	page	link
circuit	device	wire
schedule	job	constraint
commerce	customer	transaction
matching	student	application
computer network	site	connection
software	method	call
social network	person	friendship

A **graph** is a set of **vertices** and a collection of **edges** that each connect a pair of vertices. We use the names 0 through V-1 for the vertices in a V-vertex graph.



The **degree** of a vertex is the number of edges incident on it.

A **subgraph** is a subset of a graph's edges (and associated vertices) that constitutes a graph.

A **path** in a graph is a sequence of vertices connected by edges. A **simple path** is one with no repeated vertices.

DO NOT SHARE

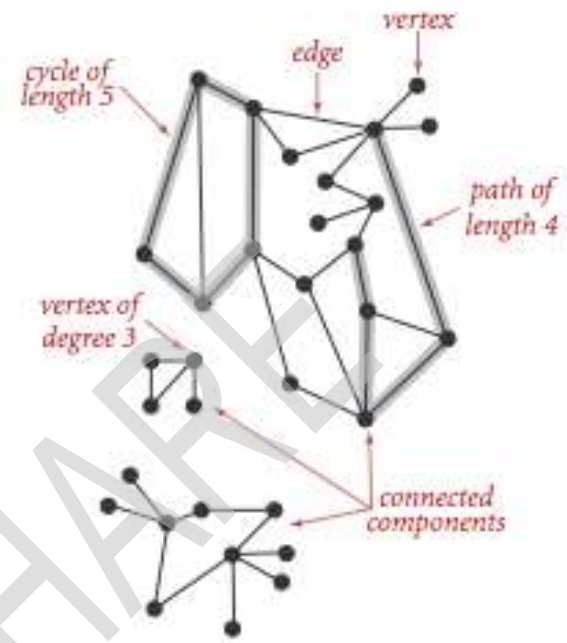
A **cycle** is a path (with at least one edge) whose first and last vertices are the same. A **simple cycle** is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

The **length** of a path or a cycle is its number of edges.

We say that one vertex is **connected to** another if there exists a path that contains both of them.

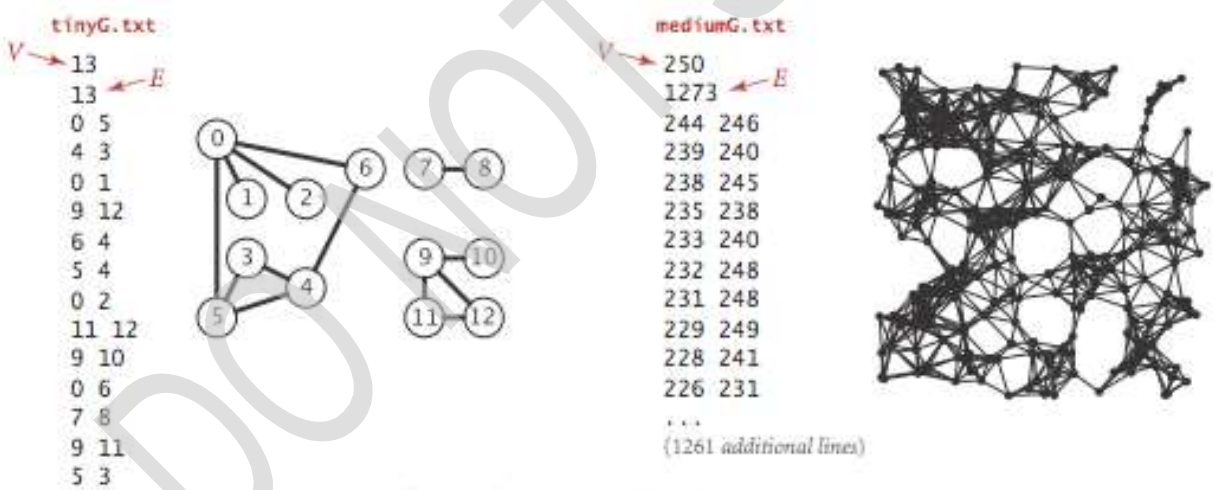
A graph is **connected** if there is a path from every vertex to every other vertex.

A graph that is not connected consists of a set of **connected components**, which are maximal connected subgraphs.



Anatomy of a graph

Input format for unweighted graph V, E and E pairs of values netween 0 and V-1 each pair denoting an edge.



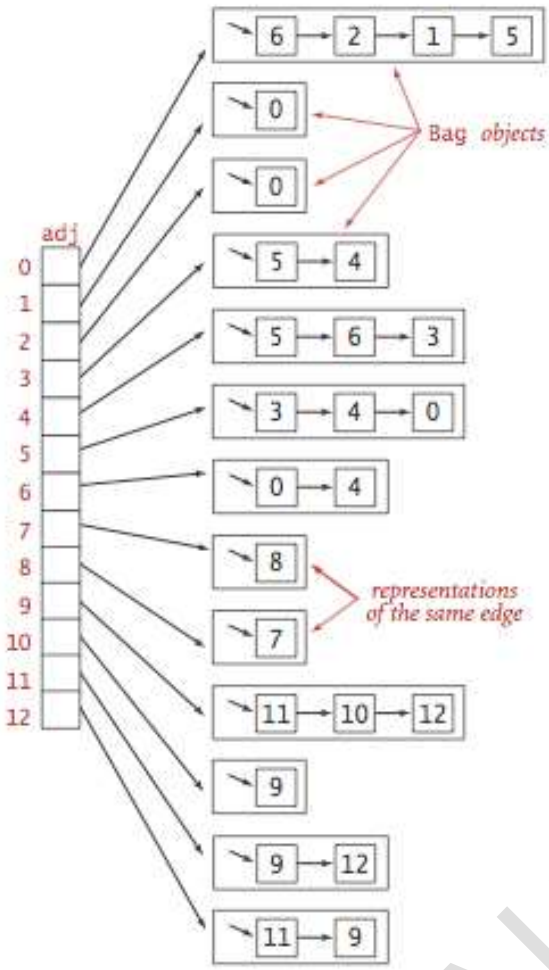
Input format for Graph constructor (two examples)

The density of a graph is the proportion of possible pairs of vertices that are connected by edges. A sparse graph has relatively few of the possible edges present; a dense graph has relatively few of the possible edges missing. Generally, we think of a graph as being sparse if its number of different edges is within a small constant factor of V and as being dense otherwise

DO NOT SHARE

Graph representation.

We can represent a graph on a 2D matrix or on an array of lists (adjacency list).



Adjacency-lists representation (undirected graph)

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		1	1			1	1						
1	1												
2	1												
3					1	1							
4				1		1	1						
5	1			1	1								
6	1				1								
7									1				
8									1				
9											1	1	1
10										1			
11										1			1
12										1		1	

DO NOT SHARE

This Graph implementation achieves the following performance characteristics:

- Space usage proportional to V + E
- Constant time to add an edge
- Time proportional to the degree of v to iterate through vertices adjacent to v

(constant time per adjacent vertex processed)

This Graph implementation achieves the following performance characteristics:

- Space usage V*V
- Constant time adding an edge
- Constant time to check adjacency of two edges

Java Adjacency List Implementations

```java
public class GFG {
    // A graph is an array of adjacency lists.
    // Size of array will be V (number of vertices in graph)
    static class Graph {
        int V;
        LinkedList<Integer> adjListArray[];

        Graph(int V) {
            this.V = V;
            // define the size of array as number of vertices
            adjListArray = new LinkedList[V];
            // Create a new list for each vertex such that adjacent nodes can be stored
            for(int i = 0; i < V ; i++){
                adjListArray[i] = new LinkedList<>();
            }
        }
    }

    // Adds an edge to an undirected graph
    static void addEdge(Graph graph, int src, int dest) {
        // Add an edge from src to dest.
        graph.adjListArray[src].add(dest);
        // Since graph is undirected, add an edge from dest to src also
        graph.adjListArray[dest].add(src);
    }

  // Driver program to test above functions
    public static void main(String args[]) {
        int V = 5;
        Graph graph = new Graph(V);
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 4);
    }
}
```

This Graph implementation maintains a vertex-indexed array of lists of integers. Every edge appears twice: if an edge connects v and w then w appears in v's list and v appears in w's list. The constructor allocates list for each vertex to store neighboring vertices.

C++ Implementations

```cpp
// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```cpp
// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
```

DO NOT SHARE

Adjacency Matrix implementation

```java
public class AdjMatrixGraph {
    private static final String NEWLINE = System.getProperty("line.separator");
    private int V;
    private int E;
    private boolean[][] adj;

    // empty graph with V vertices
    public AdjMatrixGraph(int V) {
        if (V < 0) throw new RuntimeException("Number of vertices must be nonnegative");
        this.V = V;
        this.E = 0;
        this.adj = new boolean[V][V];
    }

    // random graph with V vertices and E edges
    public AdjMatrixGraph(int V, int E) {
        this(V);
        if (E < 0) throw new RuntimeException("Number of edges must be nonnegative");
        if (E > V*(V-1) + V) throw new RuntimeException("Too many edges");

        // can be inefficient
        while (this.E != E) {
            int v = StdRandom.uniform(V);
            int w = StdRandom.uniform(V);
            addEdge(v, w);
        }
    }

    // number of vertices and edges
    public int V() { return V; }
    public int E() { return E; }


    // add undirected edge v-w
    public void addEdge(int v, int w) {
        if (!adj[v][w]) E++;
        adj[v][w] = true;
        adj[w][v] = true;
    }

    // does the graph contain the edge v-w?
    public boolean contains(int v, int w) {
        return adj[v][w];
    }
```

[4]

underlying data structure	space	add edge v–w	check whether w is adjacent to v	iterate through vertices adjacent to v
list of edges	E	1	E	E
adjacency matrix	V^2	1	1	V
adjacency lists	$E + V$	1	*degree*(v)	*degree*(v)
adjacency sets	$E + V$	$\log V$	$\log V$	$\log V$ + *degree*(v)

Order-of-growth performance for typical Graph implementations

DO NOT SHARE

[4] https://algs4.cs.princeton.edu/41graph/AdjMatrixGraph.java.html

Problem #1

Given graph G and a source node S, Find the vertices in the graph that are connected to the source.

Search(G{V,E}, int S) //Write search function.

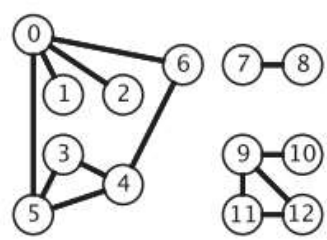
```
public class TestSearch
{
   public static void main(String[] args)
   {
      Graph G = new Graph(new In(args[0]));
      int s = Integer.parseInt(args[1]);
      Search search = new Search(G, s);

      for (int v = 0; v < G.V(); v++)
         if (search.marked(v))
            StdOut.print(v + " ");
      StdOut.println();

      if (search.count() != G.V())
         StdOut.print("NOT ");
      StdOut.println("connected");
   }
}
```

Sample graph-processing client (warmup)

```
% java TestSearch tinyG.txt 0
0 1 2 3 4 5 6
NOT connected

% java TestSearch tinyG.txt 9
9 10 11 12
NOT connected
```

tinyG.txt

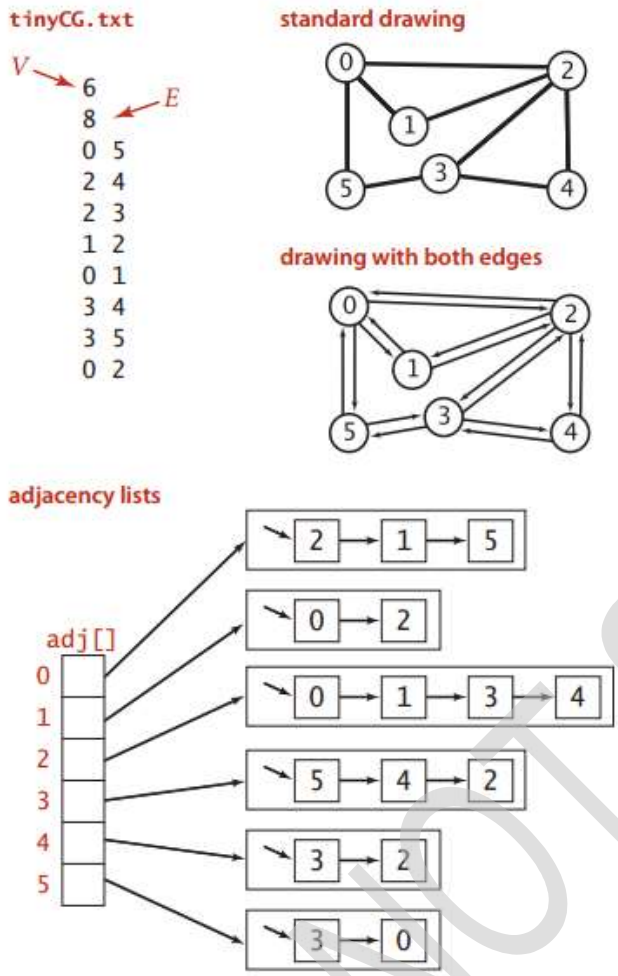
```
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



DO NOT SHARE

Depth-first search

Depth-first search is a classic recursive method for systematically examining each of the vertices and edges in a graph.



A connected undirected graph

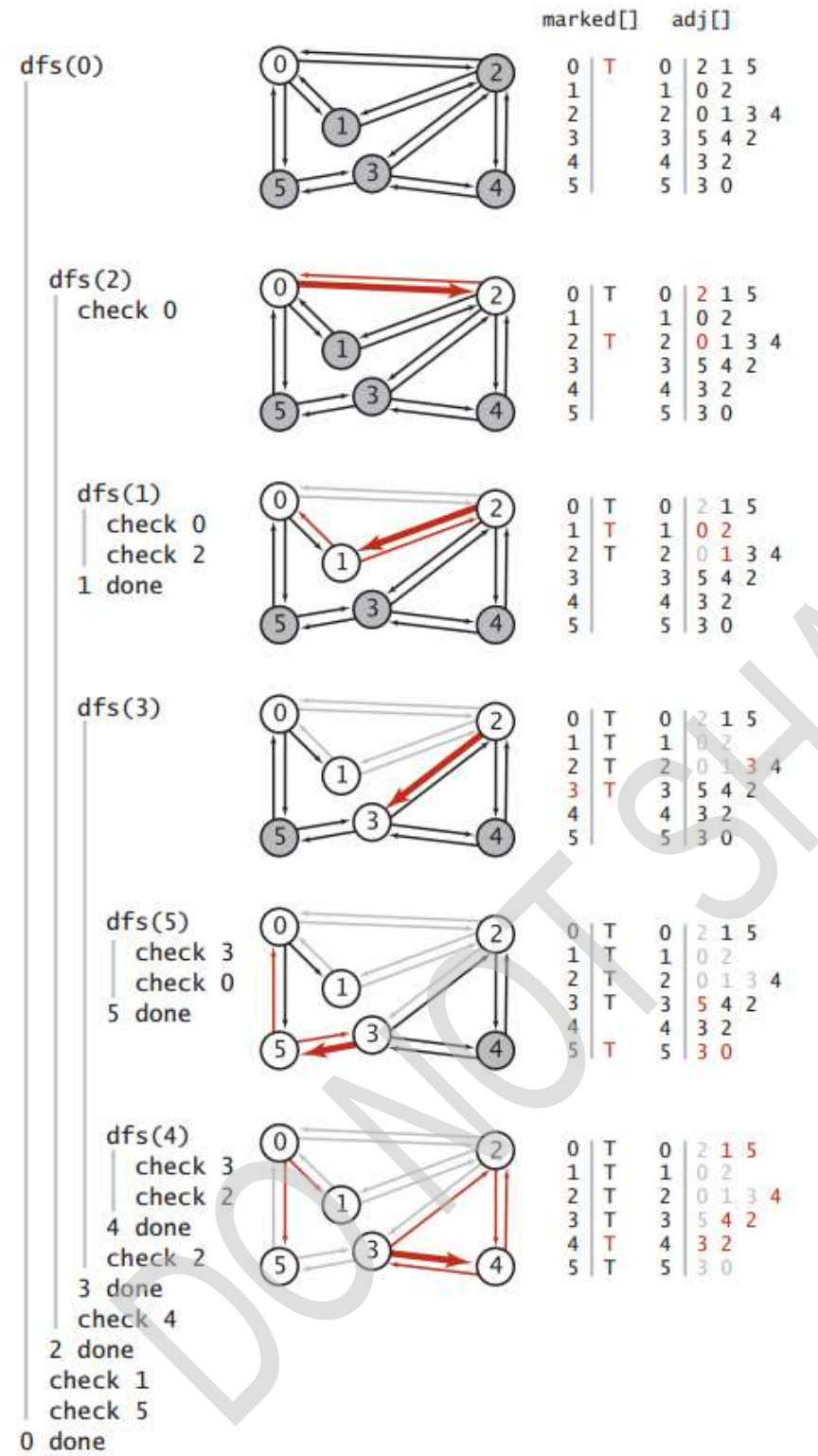
To visit a vertex

- Mark it as visited.
- Visit (recursively) all the vertices that are adjacent to it and that have not been marked yet.

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   count++;
   for (int w : G.adj(v))
      if (!marked[w]) dfs(G, w);
}
```

https://algs4.cs.princeton.edu/41graph/DepthFirstSearch.java.html

DO NOT SHARE


marked[] adj[]
dfs(0)
dfs(2)
check 0
dfs(1)
check 0
check 2
1 done
dfs(3)
dfs(5)
check 3
check 0
5 done
dfs(4)
check 3
check 2
4 done
check 2
3 done
check 4
2 done
check 1
check 5
0 done

DO NOT SHARE

Problem 2.

Single-source paths. Given a graph and a source vertex s, support queries of the form Is there a path from s to a given target vertex v? If so, find such a path.

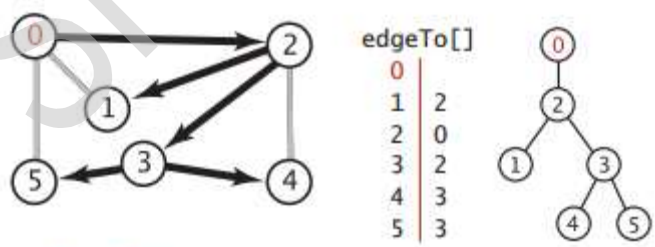
Finding paths

How to find paths?

It is easy to modify depth-first search to not only determine whether there exists a path between two given vertices but to find such a path (if one exists)

To accomplish this, we remember the edge `v-w` that takes us to each vertex `w` for the *first* time by setting `edgeTo[w]` to `v`. In other words, `v-w` is the last edge on the known path from s to `w`. The result of the search is a tree rooted at the source; `edgeTo[]` is a parent-link representation of that tree.

```
// depth first search from v
private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

Trace of `pathTo(5)` computation

DO NOT SHARE

Problem 3.

Single-source shortest paths. Given a graph and a source vertex s, support queries of the form *Is there a path from s to a given target vertex v*? If so, find a *shortest* such path (one with a minimal number of edges).

Breadth-first search

Breadth-first maze exploration

Breadth-first search is a classic method based on this goal. To find a shortest path from s to v, we start at s and check for v among all the vertices that we can reach by following one edge, then we check for v among all the vertices that we can reach from s by following two edges, and so forth.

To implement this strategy, we maintain a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, then perform the following steps until the queue is empty:

- Remove the next vertex v from the queue.
- Put onto the queue all unmarked vertices that are adjacent to v and mark them.

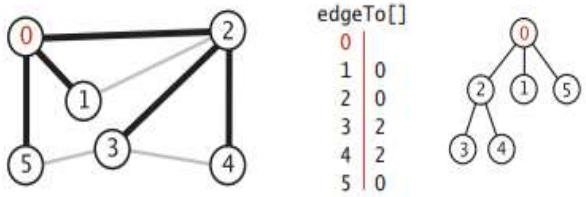
```
queue.push(root)
while queue not empty:
    node := queue.pop()
    for child in node.children:
        queue.push(child)
```

This solution only works for graphs without cycles. Can you modify it to work for any undirectional connected graph?

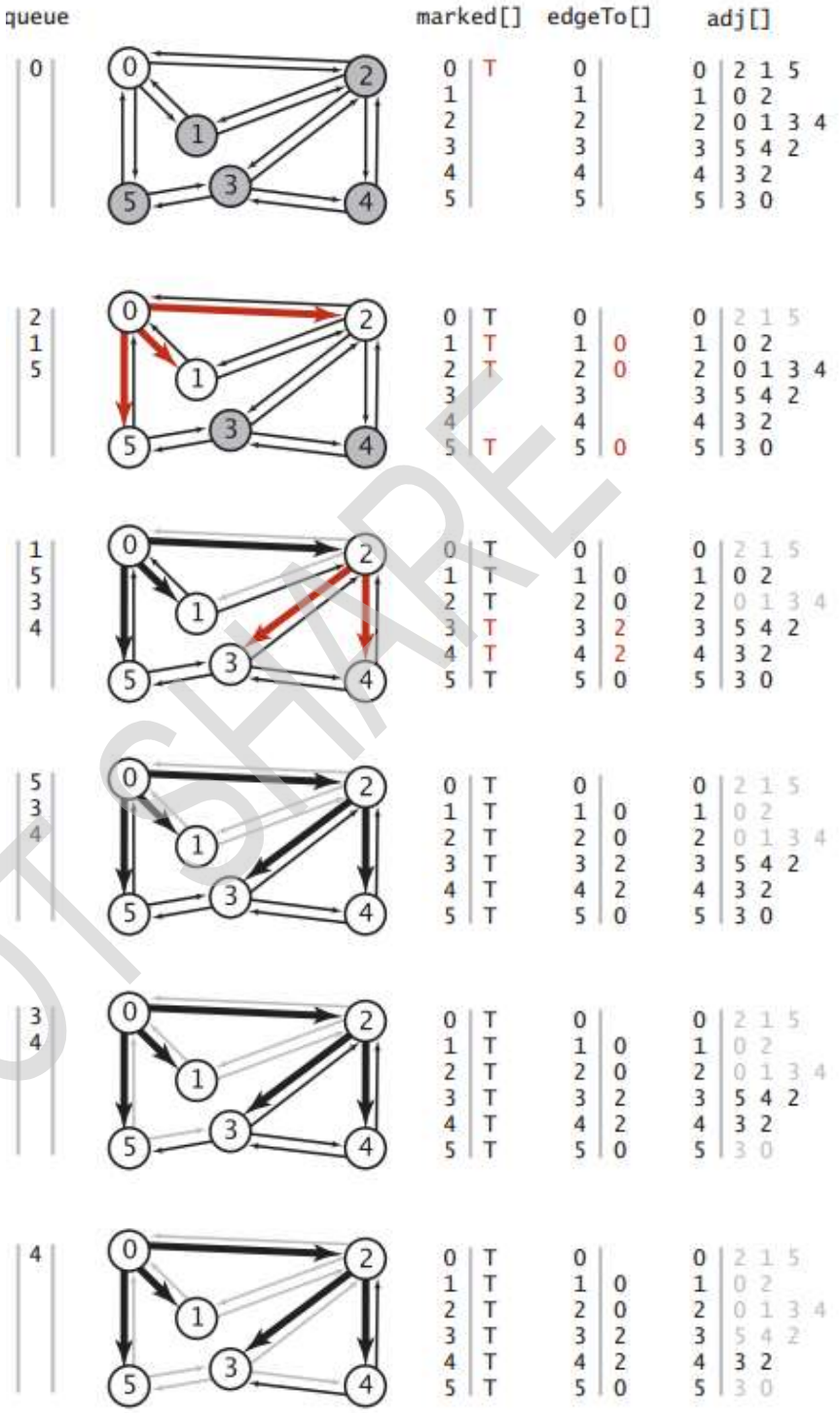
https://algs4.cs.princeton.edu/41graph/BreadthFirstPaths.java.html

DO NOT SHARE

```
private void bfs(Graph G, int s)
{
   Queue<Integer> queue = new Queue<Integer>();
   marked[s] = true;          // Mark the source
   queue.enqueue(s);          //   and put it on the queue.
   while (!q.isEmpty())
   {
      int v = queue.dequeue(); // Remove next vertex from the queue.
      for (int w : G.adj(v))
         if (!marked[w])       // For every unmarked adjacent vertex,
         {
            edgeTo[w] = v;     //   save last edge on a shortest path
            marked[w] = true;  //   mark it because path is known,
            queue.enqueue(w);  //   and add it to the queue.
         }
   }
}
```



Outcome of breadth-first search to find all paths from 0



Trace of breadth-first search to find all paths from 0

DO NOT SHARE

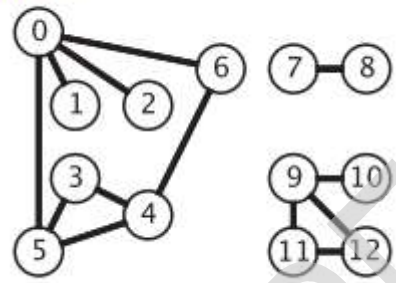
Problem 4

Given graph G, find the connected components of a graph.

Sample input and output:

```
% more tinyG.txt
13 vertices, 13 edges
0: 6 2 1 5
1: 0
2: 0
3: 5 4
4: 5 6 3
5: 3 4 0
6: 0 4
7: 8
8: 7
9: 11 10 12
10: 9
11: 9 12
12: 11 9

% java CC tinyG.txt
3 components
6 5 4 3 2 1 0
8 7
12 11 10 9
```

tinyG.txt

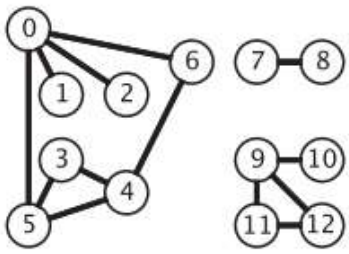



```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}
```

In main()

```
marked = new boolean[G.V()];
id = new int[G.V()];
for (int s = 0; s < G.V(); s++)
   if (!marked[s])
   {
      dfs(G, s);
      count++;
   }
```

DO NOT SHARE



	count	marked[]													id[]											
		0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12
dfs(0)	0	T													0												
dfs(6)	0	T						T							0						0						
check 0																											
dfs(4)	0	T				T		T							0				0		0						
dfs(5)	0	T				T	T	T							0				0	0	0						
dfs(3)	0	T			T	T	T	T							0			0	0	0	0						
check 5																											
check 4																											
3 done																											
check 4																											
check 0																											
5 done																											
check 6																											
check 3																											
4 done																											
6 done																											
dfs(2)	0	T		T	T	T	T	T							0		0	0	0	0	0						
check 0																											
2 done																											
dfs(1)	0	T	T	T	T	T	T	T							0	0	0	0	0	0	0						
check 0																											
1 done																											
check 5																											
0 done																											
dfs(7)	1	T	T	T	T	T	T	T	T						0	0	0	0	0	0	0	1					
dfs(8)	1	T	T	T	T	T	T	T	T	T					0	0	0	0	0	0	0	1	1				
check 7																											
8 done																											
7 done																											
dfs(9)	2	T	T	T	T	T	T	T	T	T	T				0	0	0	0	0	0	0	1	1	2			
dfs(11)	2	T	T	T	T	T	T	T	T	T	T	T			0	0	0	0	0	0	0	1	1	2		2	
check 9																											
dfs(12)	2	T	T	T	T	T	T	T	T	T	T	T	T		0	0	0	0	0	0	0	1	1	2		2	2
check 11																											
check 9																											
12 done																											
11 done																											
dfs(10)	2	T	T	T	T	T	T	T	T	T	T	T	T	T	0	0	0	0	0	0	0	1	1	2	2	2	2
check 9																											
10 done																											
check 12																											
9 done																											

Trace of depth-first search to find connected components

Implementation :

https://algs4.cs.princeton.edu/41graph/CC.java.html

DO NOT SHARE

Other Problems

Cycle detection: Is a given graph acyclic. Following code uses depth-first search to determine whether a graph has a cycle, and if so return one. It takes time proportional to V + E in the worst case.

```
private void dfs(Graph G, int v, int u)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w])
          dfs(G, w, v);
      else if (w != u) hasCycle = true;
}
```

Two-colorability: Can the vertices of a given graph be assigned one of two colors in such a way that no edge connects vertices of the same color? Following code uses depth-first search to determine whether a graph has a bipartition; if so, return one; if not, return an odd-length cycle. It takes time proportional to V + E in the worst case.

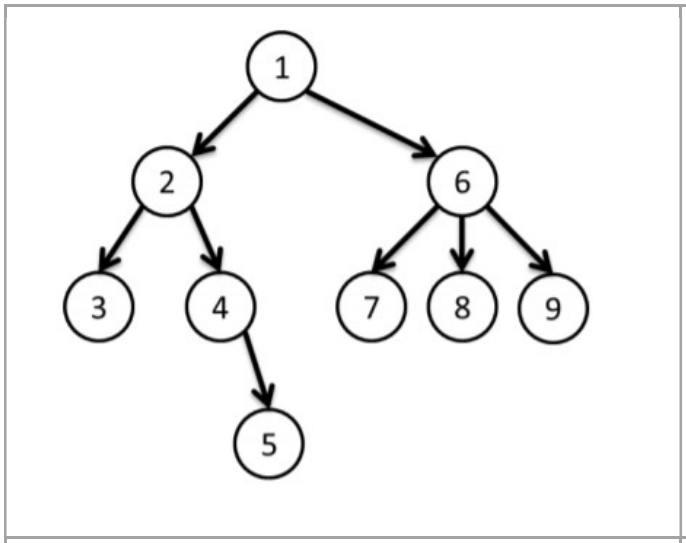
```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w])
      {
          color[w] = !color[v];
          dfs(G, w);
      }
      else if (color[w] == color[v]) isTwoColorable = false;
}
```

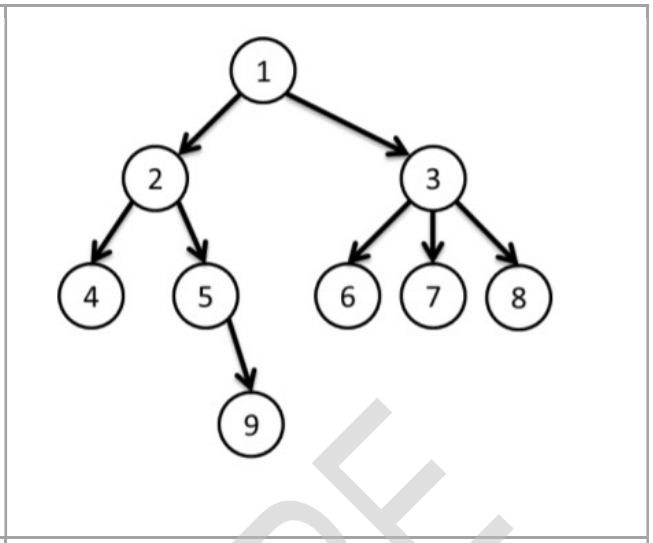
problem	solution
single-source connectivity	DepthFirstSearch
single-source paths	DepthFirstPaths
single-source shortest paths	BreadthFirstPaths
connectivity	CC
cycle detection	Cycle
two-colorability (bipartiteness)	TwoColor

References:
https://algs4.cs.princeton.edu/41graph/>
https://algs4.cs.princeton.edu/41graph/DepthFirstSearch.java.html
https://algs4.cs.princeton.edu/41graph/BreadthFirstPaths.java.html
a

DO NOT SHARE

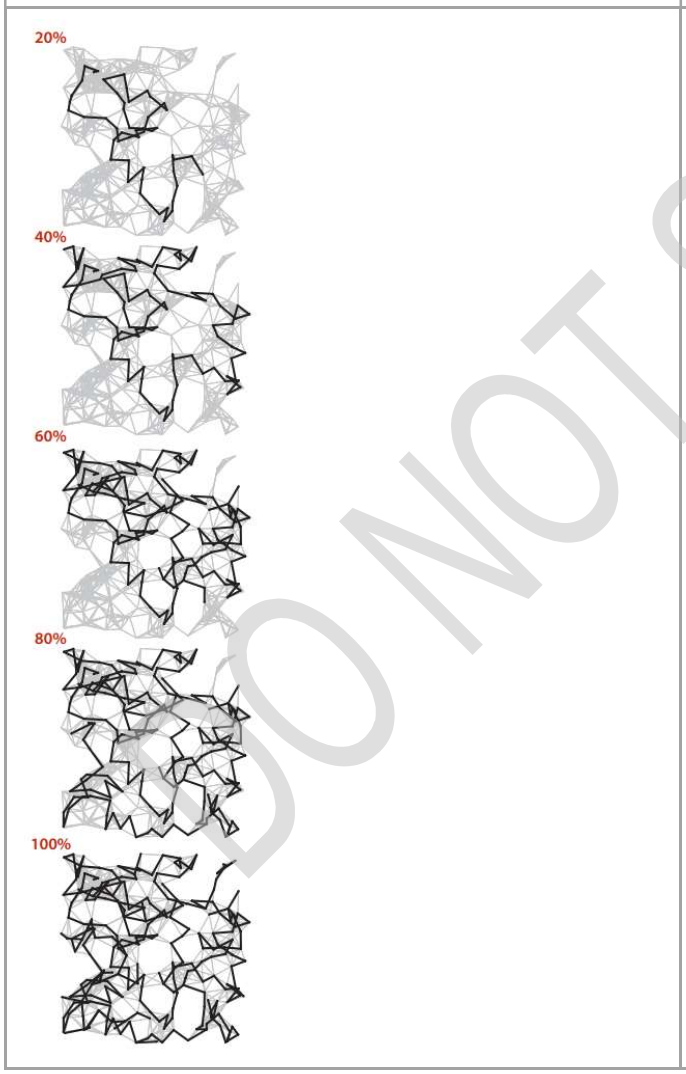
Summary

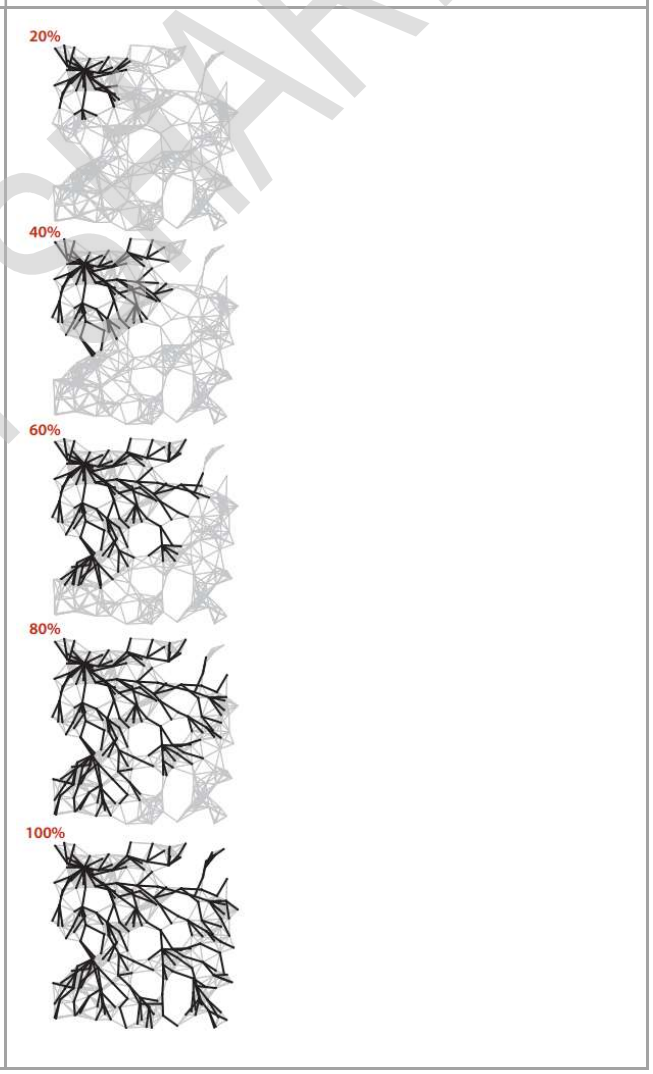




Node Traversal With Depth First Search [5]

Node Traversal With Breadth First Search





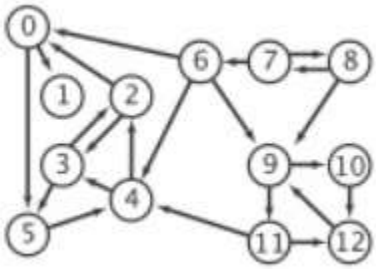
DO NOT SHARE

DFS is analogous to one person exploring a maze. BFS is analogous to a group of searchers exploring by fanning out in all directions, each unrolling his or her own ball of string. When more than one passage needs to be explored, we imagine that the searchers split up to explore all of them

[5] https://activities.tjhsst.edu/sct/lectures/1112/bfsdfs100711.pdf

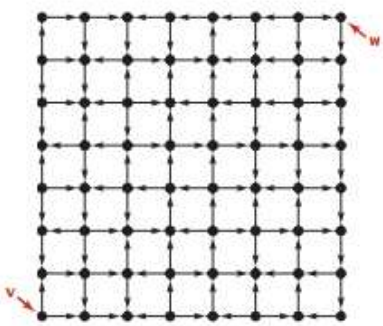
Directed Graphs

In *directed graphs*, **edges are one-way**: the pair of vertices that defines ***each edge*** is an ordered pair that **specifies a one-way adjacency**. Many applications (graphs that represent the web, scheduling constraints, or telephone calls) are naturally expressed in terms of directed graphs.



The *outdegree* of a vertex in a digraph is the number of edges
going *from* it; the *indegree* of a vertex is the number of edges going *to* it.

A *directed path* in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence. A directed cycle is a directed path (with at least one edge) whose first and last vertices are the same.



Is w reachable from v in this digraph?

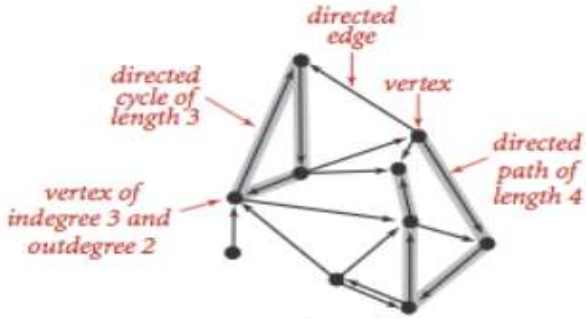
A vertex w is *reachable* from a vertex v if there is a directed path from v to w.

Appreciate the distinction between reachability in digraphs and connectivity in undirected graphs. although you are likely to be able to tell at a glance whether two vertices in a small undirected graph are connected, a directed path in a digraph is not so easy to spot.

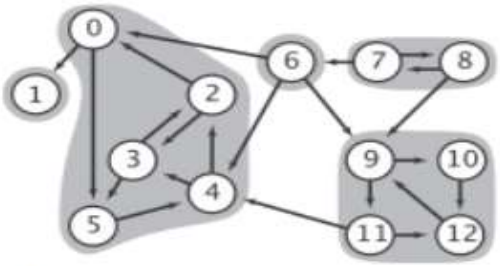
We say that two vertices v and w are *strongly connected* if they are mutually reachable: there is a directed path from v to w and a directed path from w to v.A digraph is *strongly connected* if there is a directed path from every vertex to every other vertex.

A digraph that is not strongly connected consists of a set of *strongly-connected components*, which are maximal strongly-connected subgraphs.

A *directed acyclic graph* (or DAG) is a digraph with no directed cycles.



Anatomy of a digraph

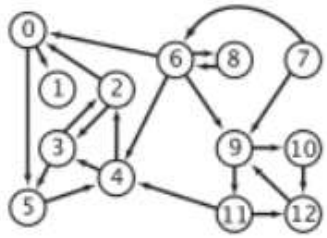


A digraph and its strong components

DO NOT SHARE

tinyDG.txt

V → 13
22 ← E
4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
5 4
0 5
6 4
6 9
7 6

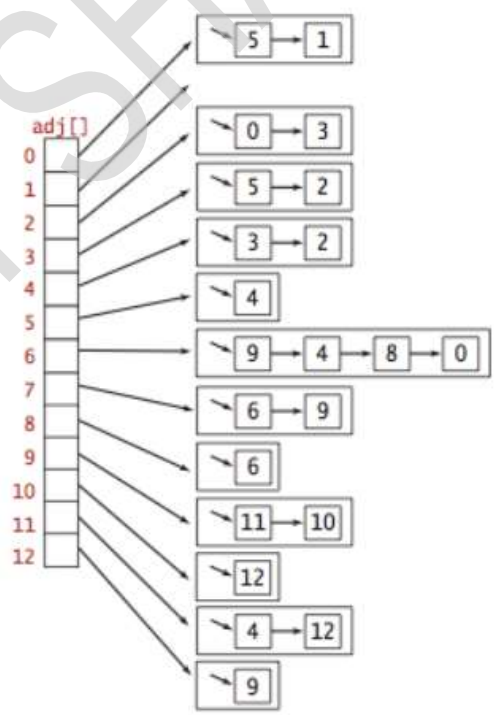


Adjacency Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		1				1							
1													
2	1			1									
3			1			1							
4			1	1									
5					1								
6	1				1				1	1			
7							1			1			
8							1						
9											1	1	
10													1
11					1								1
12										1			

```
// add directed edge v->w
public void addEdge(int v, int w) {
    if (!adj[v][w]) E++;
    adj[v][w] = true;
}
```

Adjacency List




```
public void addEdge(int v, int w) {
    adj[v].add(w);
    indegree[w]++;
    E++;
}
```

adjacency-lists implementation
https://algs4.cs.princeton.edu/42digraph/Digraph.java.html
adjacency-matrix implementation
https://algs4.cs.princeton.edu/42digraph/AdjMatrixDigraph.java.html

DO NOT SHARE

Previous chapter problems remain same and can be solved using same methods. Please answer the following questions and then compare your answers below.

Reachability in digraphs.

1. *Single-source reachability:* Given a digraph and source s, is there a directed path from s to v? If so, find such a path.
2. *Multiple-source reachability:* Given a digraph and a *set* of source vertices, is there a directed path from *any* vertex in the set to v

```
% java DirectedDFS tinyDG.txt 1
1

% java DirectedDFS tinyDG.txt 2
0 1 2 3 4 5

% java DirectedDFS tinyDG.txt 1 2 6
0 1 2 3 4 5 6 9 10 11 12
```

3. *Single-source directed paths:* given a digraph and source s, is there a directed path from s to v? If so, find such a path
4. *Single-source shortest directed paths*: given a digraph and source s, is there a directed path from s to v? If so, find a shortest such path

Answers

1. We can use depth-first search to solve this problem. Here is an implementation: https://algs4.cs.princeton.edu/42digraph/DirectedDFS.java.html
2. We can use depth-first search to solve this problem. We call DFS for each source Here is an implementation https://algs4.cs.princeton.edu/42digraph/DirectedDFS.java.html

```
private void dfs(Digraph G, int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w]) dfs(G, w);
}

for (int s : sources)
   if (!marked[s]) dfs(G, s);
```

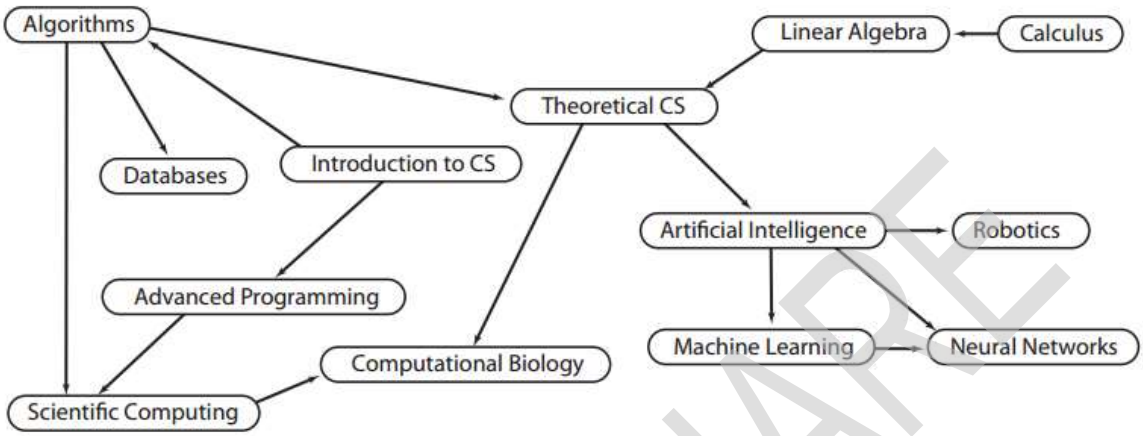
3. We can use depth-first search to solve this problem. We apply little twist to keep the path info we should have a parent array. Here is an implementation https://algs4.cs.princeton.edu/42digraph/DepthFirstDirectedPaths.java.html
4. We can use breadth-first search to solve this problem. Here is an implementation https://algs4.cs.princeton.edu/42digraph/BreadthFirstDirectedPaths.java.html

DO NOT SHARE

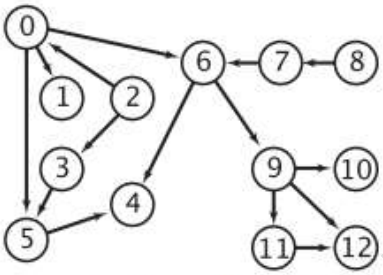
Scheduling problems.

Given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

Consider a college student planning a course schedule, under the constraint that certain courses are prerequisite for certain other courses, as in the example below.



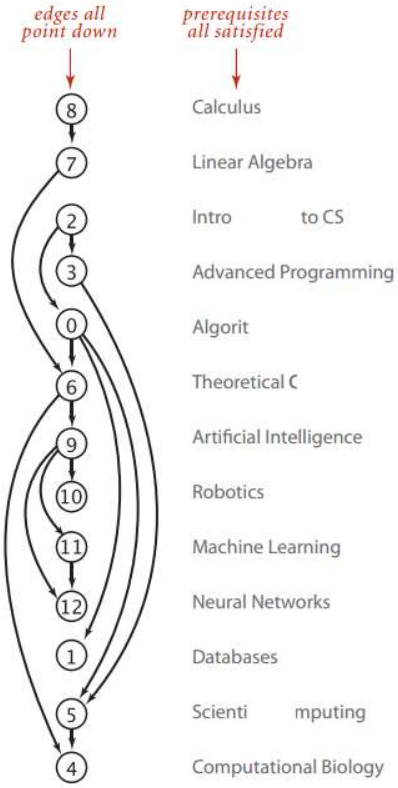
A precedence-constrained scheduling problem



Standard digraph model

For any such problem, a digraph model is immediate, with vertices corresponding to jobs and directed edges corresponding to precedence constraints.

Topological sort. Given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible)



Topological sort

Cycles in digraphs

If job x must be completed before job y, job y before job z, and job z before job x, then someone has made a mistake, because those three constraints cannot all be satisfied. In general, if a precedence-constrained scheduling problem has a directed cycle, then there is no feasible solution. To check for such errors, we need to be able to solve the following problem:

Directed cycle detection. Does a given digraph have a directed cycle? If so, find the vertices on some such cycle, in order from some vertex back to itself. A graph may have many cycles, we only ask for one cycle, not all of them.

Definition. A *directed acyclic graph* (DAG) is a digraph with no directed cycles.

Solving the directed cycle detection problem thus answers the following question: *Is a given digraph a DAG* ?

DO NOT SHARE

```
bool detectCycleDFS(int v)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and
part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to
this vertex
        for (int i=0;i<V;i++) {
            if (graph[v][i])
                if (!visited[i]&&detectCycleDFS(i))
                    return true;
                else if (recStack[i])
                    return true;
        }
        recStack[v] = false;  // remove the vertex
from recursion stack
    }
    return false;
}

bool isCyclic()
{
    for(int i = 0; i < V; i++)
        if (detectCycleDFS(i))
            return true;

    return false;
}
```

```
private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v)) {

        // short circuit if directed cycle found
        if (cycle != null) return;

        // found new vertex, so recur
        else if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }

        // trace back directed cycle
        else if (onStack[w]) {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x]) {
                cycle.push(x);
            }
            cycle.push(w);
            cycle.push(v);
            assert check();
        }
    }
    onStack[v] = false;
}
```

Proposition. A digraph has a topological order if and only if it is a DAG.

Proof: If the digraph has a directed cycle, it has no topological order. Conversely, the algorithm that we are about to examine computes a topological order for any given DAG.

Kahn Topological Sort

Algorithm:

Steps involved in finding the topological ordering of a DAG:

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

```
for each edge(src,dest) in Edges
    indegree[dest]++
```

DO NOT SHARE

```
void Graph::topologicalSort()
{
    vector<int> in_degree(V, 0);

    // Step 1
    for (int u=0; u<V; u++)
    {
        for (size_t i=0;i<adj[u].size();i++)
            in_degree[adj[u][i]]++;
    }

    // Step 2
    queue<int> q;
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0)
            q.push(i);

    int cnt = 0;
    vector <int> top_order;

    // Step 3
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        top_order.push_back(u);

        for (size_t i=0;i<adj[u].size();i++)
            // If in-degree becomes zero, add it to queue
            if (--in_degree[adj[u][i]] == 0)
                q.push(adj[u][i]);

        cnt++;
    }

    // Step 5
    if (cnt != V)
    {
        cout << "There exists a cycle in the graph\n";
        return;
    }

    // Print topological order
    for (int i=0; i<top_order.size(); i++)
        cout << top_order[i] << " ";
    cout << endl;
}
```

DO NOT SHARE

Proposition. Reverse postorder in a DAG is a topological sort.

Proof: Consider any edge v->w. One of the following three cases must hold whendfs(v) is called (see the diagram on page 583):

- dfs(w) has already been called and has returned (w is marked).
- dfs(w) has not yet been called (w is unmarked), so v->w will cause dfs(w) tobe called (and return), either directly or indirectly, before dfs(v) returns
- dfs(w) has been called and has not yet returned when dfs(v) is called.

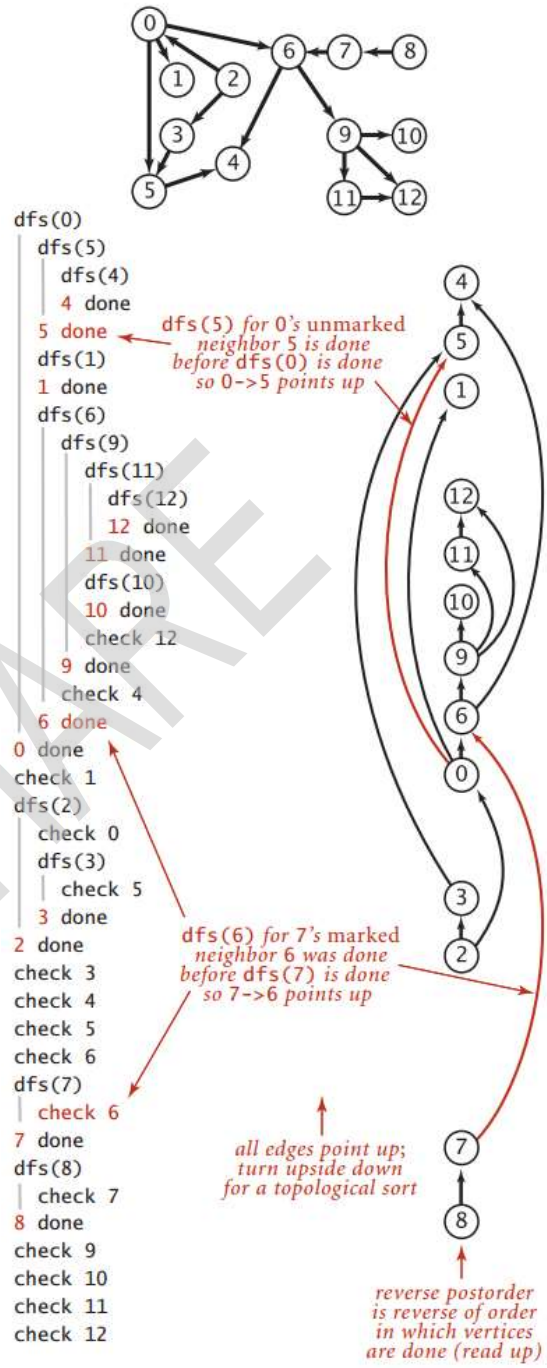
```
private void dfs(Digraph G, int v)
{
   pre.enqueue(v);

   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);

   post.enqueue(v);
   reversePost.push(v);
}
```

The key to the proof is that this case is impossible in a DAG, because the recursive call chain implies a path from w to v and v->w would complete a directed cycle.In the two possible cases, dfs(w) is done before dfs(v), so w appears *before* v in postorder and *after* v in reverse postorder. Thus, each edge v->w points from a vertex earlier in the order to a vertex later in the order, as desired.

Question: What is the complexity of Topological sort with

DFS. O(V+E)

We use one depth-first search to ensure that the graph has no directed cycles, and another to do the reverse postorder ordering. Both involve examining all the edges and all the vertices, and thus take time proportional to $V+E$



Reverse postorder in a DAG is a topological sort

USACO problem

2013 February Silver 3 Milk Scheduling

DO NOT SHARE

All Top Sort

```
//  Main recursive function to print all possible
//  topological sorts
void Graph::alltopologicalSortUtil(vector<int>& res,
                                   bool visited[])
{
    // To indicate whether all topological are found
    // or not
    bool flag = false;

    for (int i = 0; i < V; i++)
    {
        //  If indegree is 0 and not yet visited then
        //  only choose that vertex
        if (indegree[i] == 0 && !visited[i])
        {
            //  reducing indegree of adjacent vertices
            list<int>:: iterator j;
            for (j = adj[i].begin(); j != adj[i].end(); j++)
                indegree[*j]--;

            //  including in result
            res.push_back(i);
            visited[i] = true;
            alltopologicalSortUtil(res, visited);

            // resetting visited, res and indegree for
            // backtracking
            visited[i] = false;
            res.erase(res.end() - 1);
            for (j = adj[i].begin(); j != adj[i].end(); j++)
                indegree[*j]++;

            flag = true;
        }
    }

    //  We reach here if all vertices are visited.
    //  So we print the solution here
    if (!flag)
    {
        for (int i = 0; i < res.size(); i++)
            cout << res[i] << " ";
        cout << endl;
    }
}
```

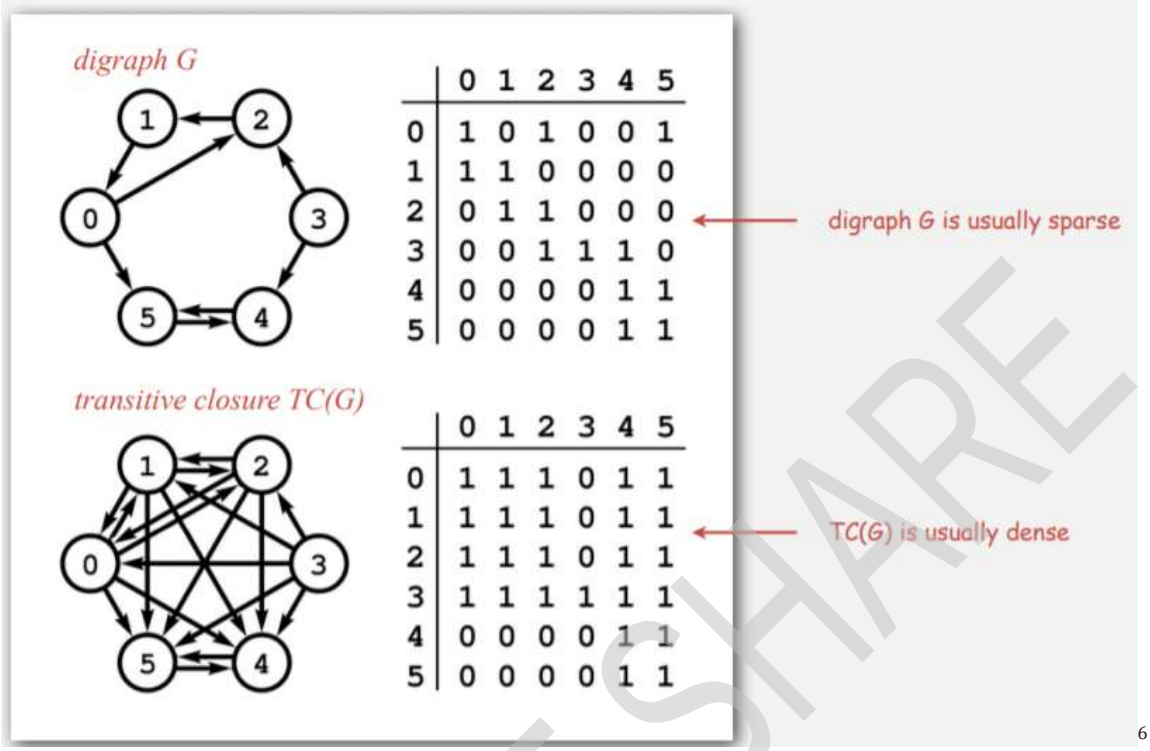
USACO Problem

Training gate 4.4.3 Frame up

DO NOT SHARE

Transitive Closure

Problem The transitive closure of a digraph G is another digraph with a directed edge from v to w if there is a directed path from v to w in G.



[6]

```
public class TransitiveClosure {
   private boolean[][] tc;

   public TransitiveClosure(Digraph G) {
      tc = new boolean[G.V()][G.V()];
      for (int v = 0; v < G.V(); v++)
         dfs(G, v, v);
   }
                          run dfs from every vertex

   // reachability from s, made it to v
   private void dfs(Digraph G, int s, int v) {
      tc[s][v] = true;
      for (int w : G.adj(v))
         if (!tc[s][w]) dfs(G, s, w);
   }

   public boolean reachable(int v, int w) { return tc[v][w]; }
                          is w reachable from v?
```

[7]

~ V E preprocessing time, ~ V^2 space, constant query time.

We have another transitive closure algorithm which needs

[6] https://www.cs.princeton.edu/courses/archive/spr10/cos226/lectures/13-42DirectedGraphs.pdf
[7] https://www.cs.princeton.edu/courses/archive/fall05/cos226/lectures/digraph.pdf

DO NOT SHARE

O(V^3) preprocessing and O(V^2) space

Algorithm Warshall
Input: Adjacency matrix A of relation R on a set of n elements
Output: Adjacency matrix T of the transitive closure of R.
Algorithm Body:

```
T := A [initialize T to A]
for j := 1 to n
    for i := 1 to n
        if T_{i,j} = 1 then
            a_i := a_i ∨ a_j [form the Boolean OR of row i and row j, store it in a_i]
        next i
    next j
end Algorithm Warshall
```

[8]

```
For each intermediate node
    For each source node
       If path form source node to intermediate node
          All nodes accessible by intermediate are accessible by source
```

```
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on a path from i to j,
            // then make sure that the value of reach[i][j] is 1
            reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
        }
    }
}
```

Sample Problems:

USACO 2016 December Silver Problem 3 MooCast

USACO 2006 December Silver Problem 1 Cow Picnic

DO NOT SHARE

[8] https://www.dartmouth.edu/~matc/DiscreteMath/V.6.pdf

Minimum Spanning Tree

Friday, June 22, 2018 3:41 PM

Greedy Algorithm: Repeatedly making locally best choice/decision ignoring effect on future

q) What is a Tree?

Brown body, supporting branches and green leaves in most species. (wikipedia) 😜

Tree is Connected, acyclic graph. A simple exact definition.

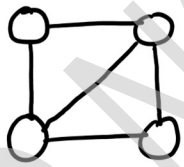
1) Connected ⇒ N vertices at least n-1 edges to be connected
2) Acyclic ⇒ N vertices at most n-1 edges to be acyclic.

// make sure you understand this before going further.

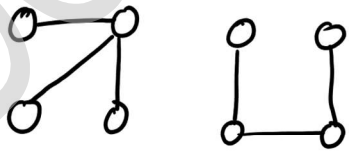
Spanning Tree?

Subgraph of G

subset of edges of G that form a tree and connect all the vertices.



q) how many spanning trees can you form for this graph?



. . .

3 edges to be selected from 5. $C\binom{5}{3} = 10$

2 of selection breaks rule. $10 - 2 = 8$

1 2 3 4 5 6 7 8

DO NOT SHARE

Tree – MST

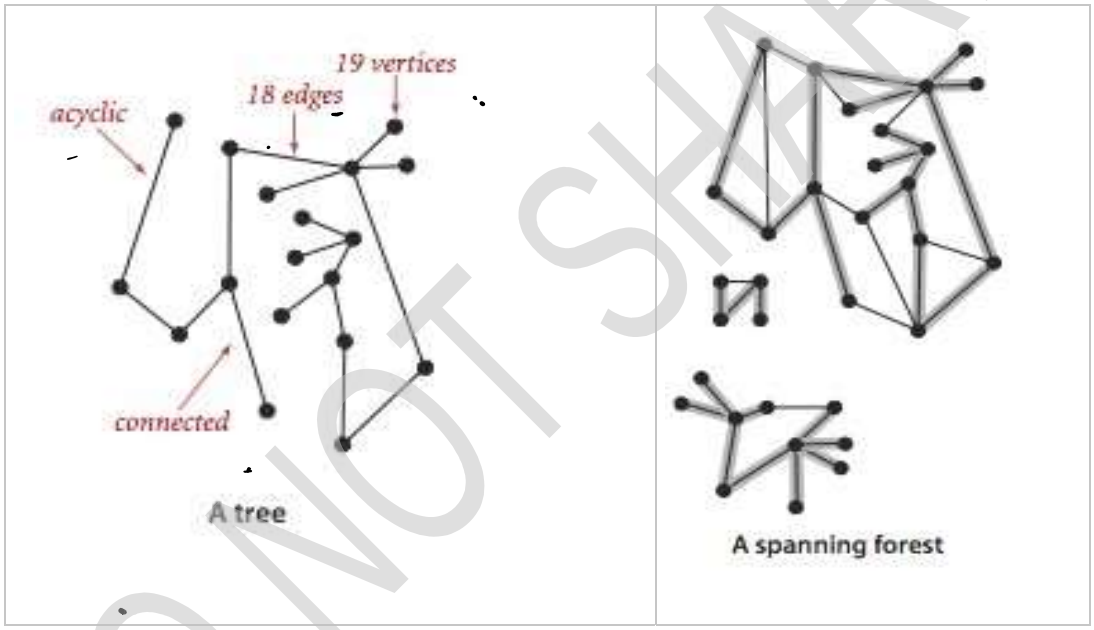
An *acyclic graph* is a graph with no cycles.

A *tree* is an acyclic connected graph.

A *forest* is a disjoint set of trees.

A *spanning tree* of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A *spanning forest* of a graph is the union of the spanning trees of its connected components.

A *bipartite graph* is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set.



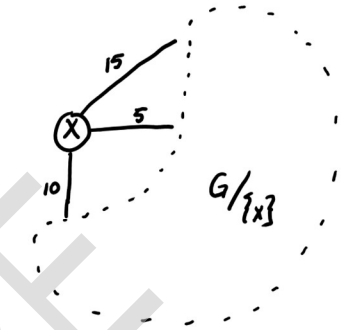
- *G* has *V*-1 edges and no cycles.
- *G* has *V*-1 edges and is connected.
- *G* is connected, but removing any edge disconnects it.
- *G* is acyclic, but adding any edge creates a cycle.
- Exactly one simple path connects each pair of vertices in *G*.

DO NOT SHARE

Prim's Minimum Spanning Tree Algorithm

Given a graph $G = \{V, E\}$ & edge weights $w : E$

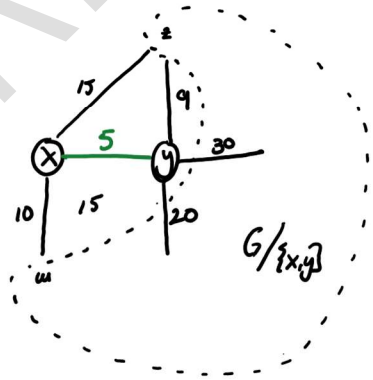
Find a spanning tree (T) of G with minimum total weight. $\min\left(\sum_{e \in T} w(e)\right)$



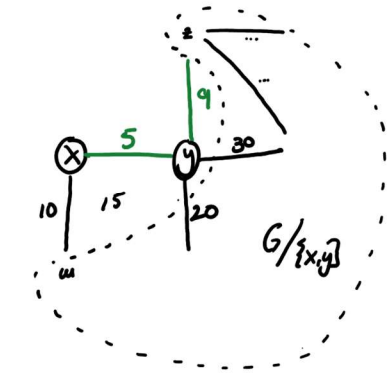
Suppose that for a given graph G you know the resulting MST T.

Then for each node x there is an edge connecting x to the rest of the graph.

If this is MST then we must be connecting x to the rest of MST with edge 5. Because all edges $(5, 10, 15)$ ends in inside T. To have MST we must have chosen the minimum one



Now take x, y as a group. We must have a connection between MST $T/\{x,y\}$ and $\{x,y\}$ group. And our options are $10, 20, 30, 9 \Rightarrow$ We must be connecting $\{x,y\}$ to the rest of T with edge of cost 9.



And so on...

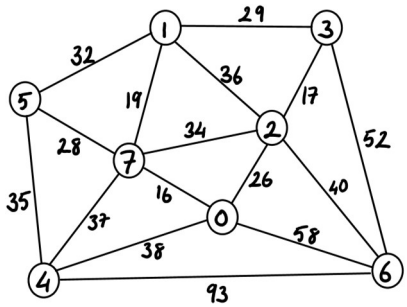
9) When you get $y \to z = 9$ do you have to keep info $x \to z = 15$ or when you see $y \to w = 15$ while you have $x \to w = 10$ should you consider as option?

No. Keep only one (minimum) distance value for each vertex.

This process is what Prim's algorithm suggests. If you would come in 1958 we would call this algorithm with your name. :)

DO NOT SHARE

Sample Run



initialize dist[] with INF &

visited[] with False

	0	1	2	3	4	5	6	7
dist	INF	INF	INF	INF	INF	INF	INF	INF
visited	F	F	F	F	F	F	F	F

Choose an arbitrary node to start with.

for e.g. node 0. Set its dist to 0.

dist	0	INF	INF	INF	INF	INF	INF	INF
visited	F	F	F	F	F	F	F	F

Repeat N times

Choose not visited & min dist node.

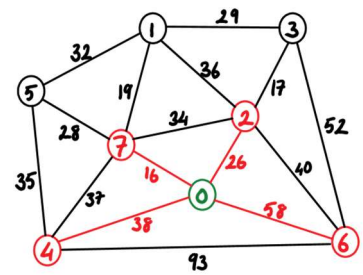
Set visited [node] = True

Relax neighbors of min node.

DO NOT SHARE

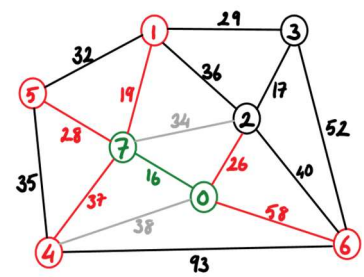
Step 1 — choose vertex 0

dist	0	INF	26	INF	38	INF	58	16
visited	T	F	F	F	F	F	F	F



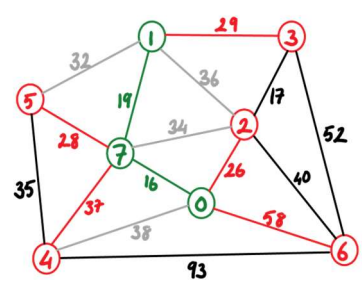
Step 2 — choose vertex 7

	0	1	2	3	4	5	6	7
dist	0	19	26	INF	37	28	58	16
visited	T	F	F	F	F	F	F	T



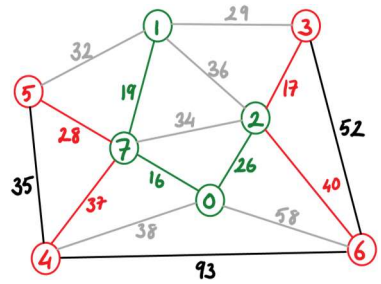
Step 3 — choose vertex 1

	0	1	2	3	4	5	6	7
dist	0	19	26	29	37	28	58	16
visited	T	T	F	F	F	F	F	T



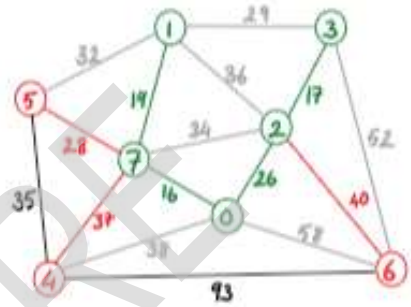
Step	choose vertex
4	2

	0	1	2	3	4	5	6	7
dist	0	19	26	17	37	28	40	16
visited	T	T	T	F	F	F	F	T



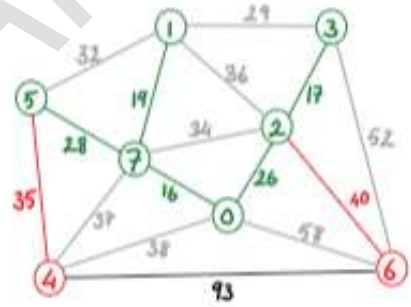
Step	choose vertex
5	3

	0	1	2	3	4	5	6	7
dist	0	19	26	17	37	28	40	16
visited	T	T	T	T	F	F	F	T



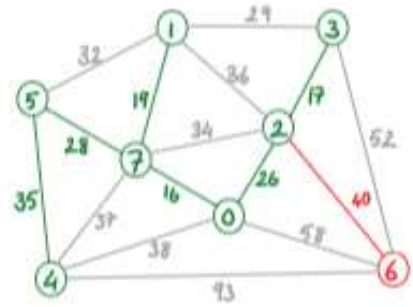
Step	choose vertex
6	5

	0	1	2	3	4	5	6	7
dist	0	19	26	17	35	28	40	16
visited	T	T	T	T	F	T	F	T



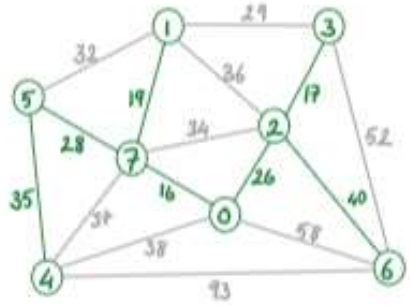
Step	choose vertex
7	4

	0	1	2	3	4	5	6	7
dist	0	19	26	17	35	28	40	16
visited	T	T	T	T	T	T	F	T



Step	choose vertex
8	6

	0	1	2	3	4	5	6	7
dist	0	19	26	17	35	28	40	16
visited	T	T	T	T	T	T	T	T



DO NOT SHARE

Prim's Algorithm Pseudocode

set dist[] all vertices INF

start with an arbitrary vertex and set dist[init]=0

Repeat

Choose min dist & not chosen vertex

Relax the dist[] array with new vertex.

how many times I have to repeat?

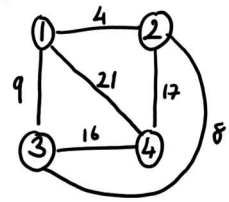
Resulting MST must contain N vertices ⇒ repeat N times.

```
dist [N] = {INF}
dist [init_node] = 0;
for (i: 0 → N)
{
    //find min dist & not visited vertex
    for (j:0 → N)
        if (!visited [j] && dist [j] < dist [min])
            min = j
    visited [min] = true;
    // Apply new coming vertex effect on distances
    for (j:0 → N)
        if (mat [min] [j] && dist [j] > mat [min] [j])
            dist [j] = mat [min] [j]
}
```

DO NOT SHARE

Agri Net

Friday, June 22, 2018 4:34 PM



$N \leq 100$ farms

Given distances between farms.

What is sum of minimum length wire to connect all farms?

Solution :

A perfect Minimum Spanning Tree application

Set all dist to INF.

Choose a vertex to start from. Set its dist to 0.

Repeat N times

Choose min dist & not visited vertex

Apply its adjacent edges effect on dist[].

1	2	3	4
INF	INF	INF	INF
F	F	F	F

q) Does it matter, which farm to start with?

No.

Choose a random farm to start. Say farm 1. Set its dist to 0.

1	2	3	4
0	INF	INF	INF
F	F	F	F

Step	Vertex
1	1
2	2
3	3
4	4

1	2	3	4
0	4	9	21
T	F	F	F

1	2	3	4
0	4	8	17
T	T	F	F

1	2	3	4
0	4	8	16
T	T	T	F

1	2	3	4
0	4	8	16
T	T	T	T

q) what is output?

Total of distance array. 28

DO NOT SHARE

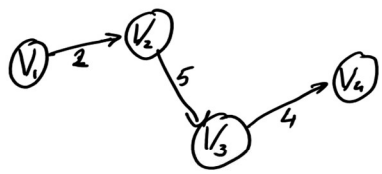
Shortest Path

Perhaps the most intuitive graph-processing problem is one that you encounter regularly, when using a map application to get directions from one place to another. A graph model is immediate: vertices correspond to intersections and edges correspond to roads, with weights on the edges that model the cost, perhaps distance or travel time.

A ***shortest path*** from vertex s to vertex t is a directed path from s to t with the property that no other such path has a lower weight.

$G(V, E)$ with $w: E \to \mathbb{R}$

Path $p = V_1 \to V_2 \to \dots V_k$ has weight $w(p) = \sum_{i=1}^{k-1} w(V_i - V_{i+1})$



Shortest path from u to v is a path of minimum weight from u to v.

Solution idea: Find all paths from u to v.

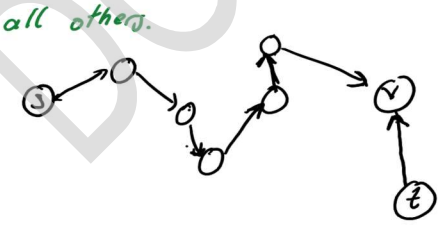
Choose the minimum of them.

q) What is the rate of number of paths to vertices?

Linear	aV
Polynomial	aV^2
Exponential	2^V ⇐ exp.

for each 2 nodes we add
paths multiplied by 2

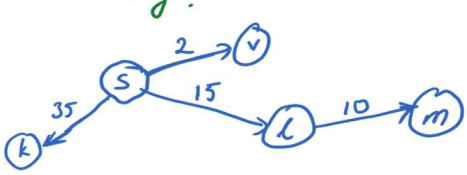
q) Shortest path from u to v. Can it be found without finding shortest path from u to all others.



You found a shortest path from s to v

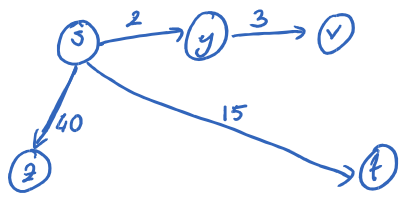
maybe there is a vertex t which you did not find shortest path (to t from s) and from t to v it will be shorter.

In order to find shortest path from s to v. You have to find shortest paths from s to all others. Falsify?



We can find shortest path s to v without finding from s to any other.

DO NOT SHARE



Before finding shortest path s to v

I fill find shortest path to all vertices

that has weight less than shortest path(s→v)

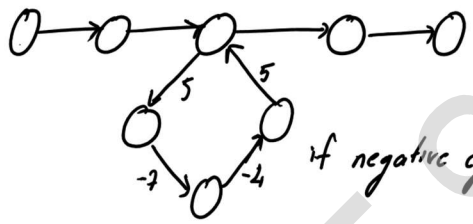
Another greedy algorithm:

(Repetadly makes the locally best choice ignoring effect on future)

Negative edge weight

Some shortest paths may not exist.

q) When we can't have a shortest path?



if negative cycle makes it shorter by cycling.

weight sp $-\infty$

if no negative cycle ⇒ there is a solution. maybe < 0.

q) When else shortest path may not exist?

if no path from s to v. weight sp $= \infty$

(not connected)

A subpath of a shortest path is a shortest path. Is it?

s ⟶ v contains k

s→k & k→v are also shortest paths.

If there would be a shorter path s⟶k then I would have choosen that subpath.

DO NOT SHARE

Triangle inequality

for all vertices $u, v, x \in V$

$$sp(u,v) \leq sp(u,x) + sp(x,v)$$

$sp(u,v)$ can be at most $sp(u,x) + sp(x,v)$

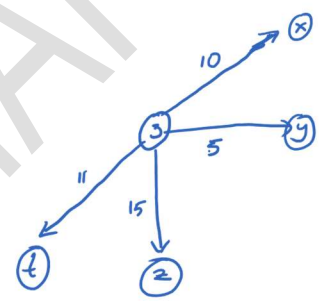
Single source shortest path

From given source vertex $s \in V$

Find shortest path weight $sp(s,v)$ for all $v \in V$

working on all edges > 0 $\Rightarrow$ sp exists provided path exists.

<u>Greedy Approach</u>: Remember MST last lesson. We have a similar approach.



Start with a source vertex s which has edges to some vertices say x, y, z, t and no more. We can not be sure about x, y, t but if we select the minimum weight edge 5 to y. for sure we can not find a smaller path to y. (passing over others).

	s	x	y	z	t	k	l
dist:	0	10	5	15	11	∞	∞ ...

To x we may find a lower cost path over y

To z we may find a lower cost path over y

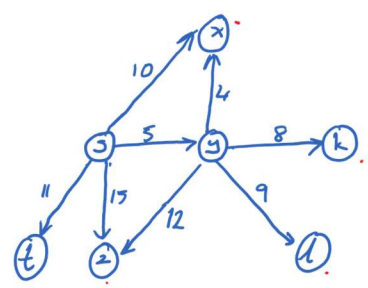
To t we may find a lower cost path over y.

But to y we can not find a lower cost path over x, z and t they are already bigger.

$\Rightarrow$ We found the answer for one vertex y.

Now we can apply its effect on distances to update cost values.

Whichever vertex y has a connection recalculate their costs.



	s	x	y	z	t	k	l
dist	0	.	5	.	11	.	. ...

x, z, k, l will be checked for update

if $(s,y) + (y,d) < (s,d) \Rightarrow$ update (s,d)

DO NOT SHARE

if $(s,y)+(y+d) < (s,d) \Rightarrow$ update (s,d)

	s	x	y	z	t	k	l	
dist	0	9	5	15	11	13	14	. .

Step2: Among the distances choose unvisited min dist edge as new solution and apply its effect.

Possible moves 9, 11, 13 and 14
to x, t, k and l

q) Why don't count 0 and 5
They are already visited.

We choose edge to x of dist 9 and relax the dist array with the connections of x.

Step i:

- choose min dist vertex as new part of solution. (choose min)
- Apply new vertex connections effect on the dist values (relax)

q) How many times repeat this operation.

a) V vertices $\Rightarrow$ V times.

Always have a set of vertices S which you have the answer. Add one new (min dist) vertex at each step

choose min of crossing edges

DO NOT SHARE

Implementation

Monday, November 27, 2017 8:13 PM

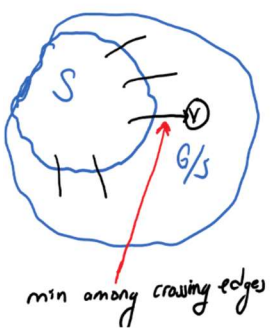
Dijkstra's Alg.

set all dist to ∞
dist [s] = 0;
repeat V times
    choose min dist & !visited vertex v.
    update dist with min vertex edges.

q) What if G is not connected. How to adapt algorithm?

a) we won't have a valid min vertex.

q) Suppose you are looking for a specific shortest path $u \to v$
and you selected v as min dist vertex at an intermediate step
Can you stop searching or is it possible you may find a better solution?



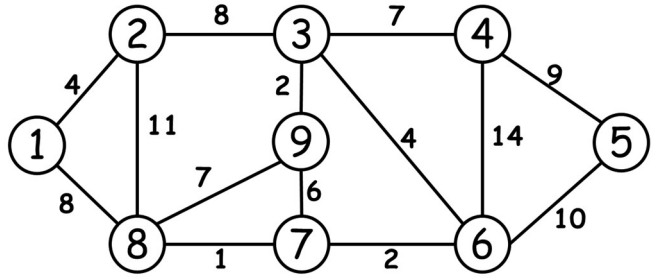
a) Remember we have a set of vertices that we know answer/solution
And at each step we get one more vertex from G/S and add to S.
When we choose a vertex as min that means we found optimal solution for that vertex. It won't be updated anymore. You can stop searching if you are only interested in v.

```
dist[v] = ∞
dist[s] = 0
Repeat V times
    //choose min
    cur = -1;
    for (i: 0 → V)
        if (!visited[i] && (cur == -1 || dist[cur] > dist[i]))
            cur = i;
    // if dist[cur] = ∞ ⇒ graph is not connected
    //relax
    for (i: 0 → V)
        if (dist[i] > dist[cur] + mat[cur][i])
            dist[i] = dist[cur] + mat[cur][i];
```

```
for each v: Adj[cur]
    if dist[v] > dist[cur] + mat[cur][v]
        dist[v] = dist[cur] + mat[cur][v]
```

DO NOT SHARE

Dijkstra Sample Run



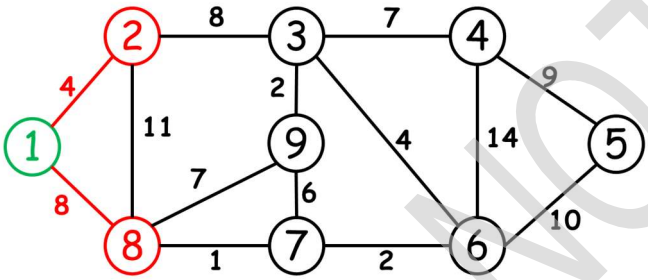

```
Initialize dist[] with INF
          visited[] with False
```

	1	2	3	4	5	6	7	8	9
Dist	INF	INF	INF	INF	INF	INF	INF	INF	INF
visited	F	F	F	F	F	F	F	F	F

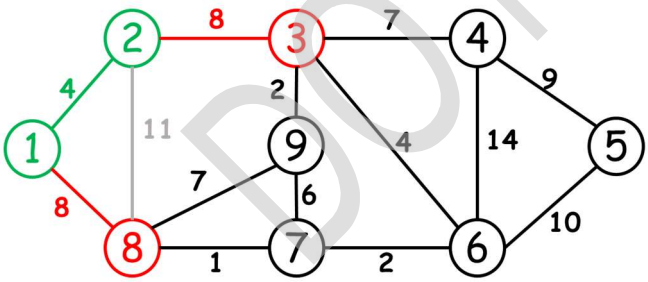
```
Set source vertex dist to 0
```

	1	2	3	4	5	6	7	8	9
Dist	0	INF	INF	INF	INF	INF	INF	INF	INF
visited	F	F	F	F	F	F	F	F	F

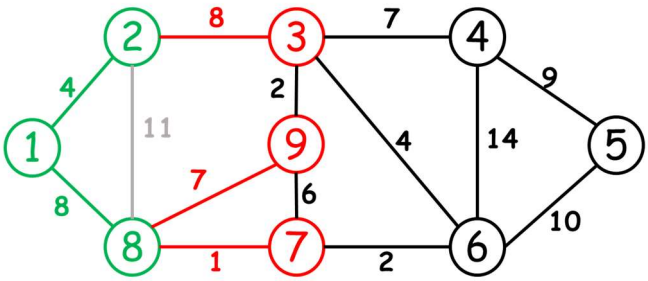
```
Repeat N times
   Choose not visited & min dist vertex
   Relax dist array with min vertex
```



	1	2	3	4	5	6	7	8	9
Dist	0	4	INF	INF	INF	INF	INF	8	INF
visited	T	F	F	F	F	F	F	F	F

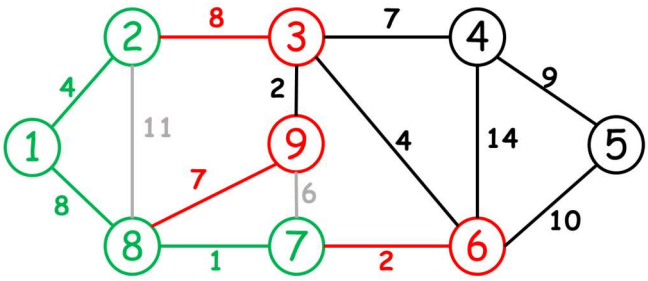


	1	2	3	4	5	6	7	8	9
Dist	0	4	12	INF	INF	INF	INF	8	INF
visited	T	T	F	F	F	F	F	F	F

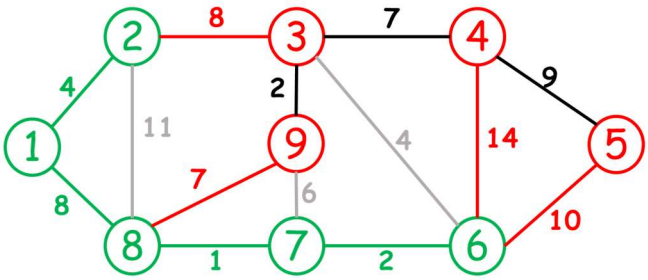


	1	2	3	4	5	6	7	8	9
Dist	0	4	12	INF	INF	INF	9	8	15
visited	T	T	F	F	F	F	F	T	F

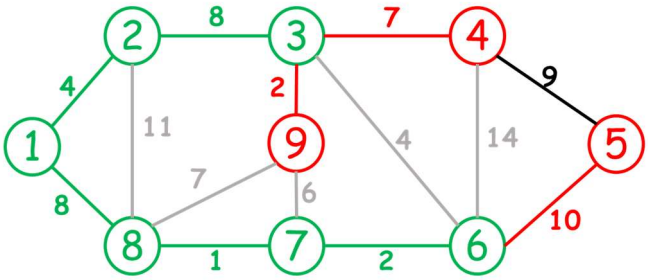
DO NOT SHARE



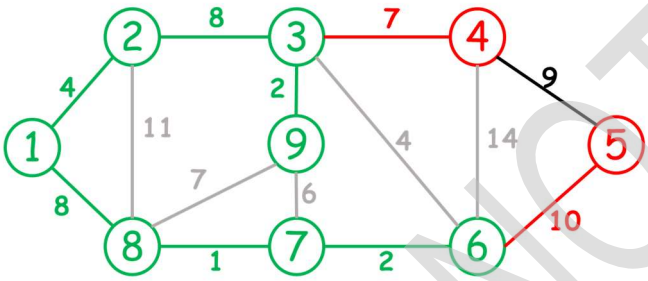
	1	2	3	4	5	6	7	8	9
Dist	0	4	12	INF	INF	11	9	8	15
visited	T	T	F	F	F	F	T	T	F



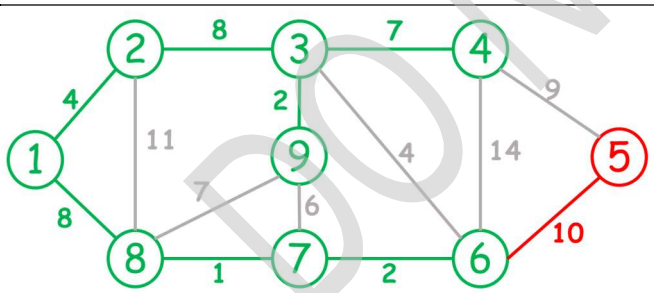
	1	2	3	4	5	6	7	8	9
Dist	0	4	12	25	21	11	9	8	15
visited	T	T	F	F	F	T	T	T	F



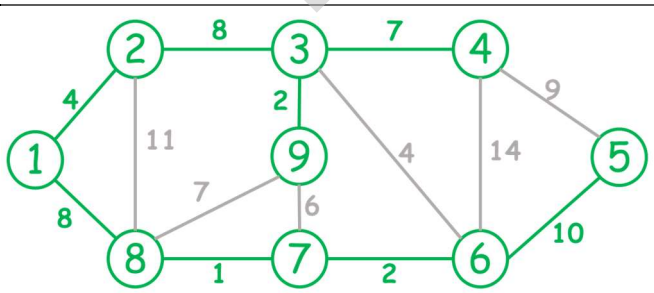
	1	2	3	4	5	6	7	8	9
Dist	0	4	12	19	21	11	9	8	14
visited	T	T	T	F	F	T	T	T	F



	1	2	3	4	5	6	7	8	9
Dist	0	4	12	19	21	11	9	8	14
visited	T	T	T	F	F	T	T	T	T



	1	2	3	4	5	6	7	8	9
Dist	0	4	12	19	21	11	9	8	14
visited	T	T	T	T	F	T	T	T	T



	1	2	3	4	5	6	7	8	9
Dist	0	4	12	19	21	11	9	8	14
visited	T	T	T	T	T	T	T	T	T

DO NOT SHARE

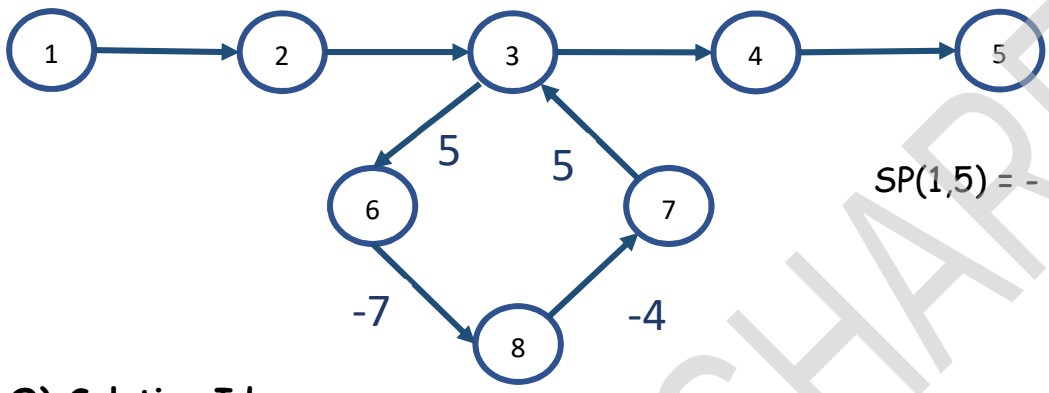
Shortest Path II

Given a graph G = {V,E} and w:E-> R^+ and a source vertex s. Dijkstra Algorithm finds shortest path for all destinations.

q) What if the graph contains negative edges also?

Given a graph G = {V,E} and w:E->R. (Notice that we don't have any restriction on weights now.) Given a source vertex s. Find Shortest Path weight SP(s,d) ∀ d ∈ V.

Recall that if G has negative cycle there is no any Shortest Path

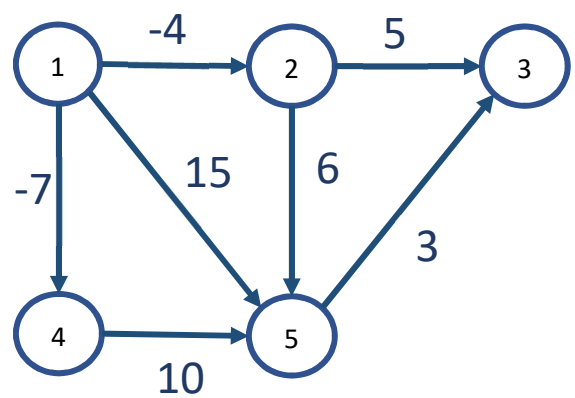


SP(1,5) = - INF

Q) Solution Idea

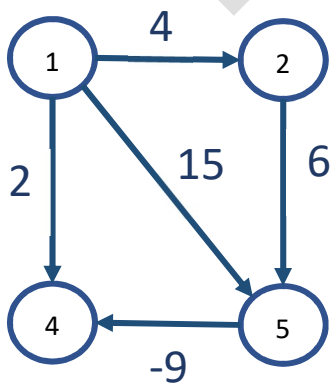
Dijkstra solves the problem if all edges are positive

1. Scan entire graph
2. Find min weighted edge
3. Add that much to all edges
4. Apply Dijkstra



Q) What is SP(1)? How does it change when you apply the solution idea?

q) Find a counter example graph this idea will not work?



Q) What is SP Tree by Dijkstra?

Q) What is the correct SP Tree?

We have to develop a new approach!! We will learn a new algorithm!

DO NOT SHARE

Bellman-Ford Shortest Path Solution

Bellman-Ford Algorithm finds Shortest Path weight SP(s,d) ∀ d ∈ V.

Or reports if there exists a negative cycle.

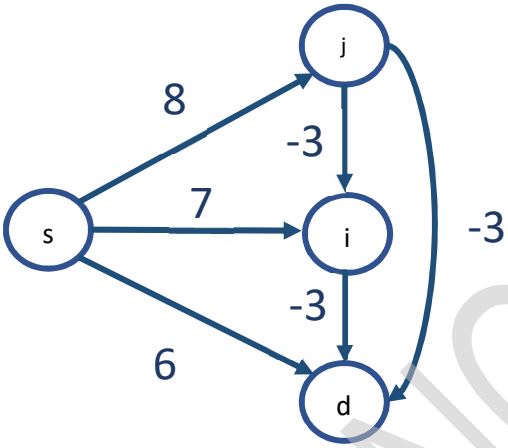
Here is how it works:

Suppose you have a negative weighted edge that ends in v.

This edge effects d[v]. If it is more than d[u] -9 then it must be updated.

Run this for all edges => negative weighted edges will be felt from one away vertices.

```
for (each edge (u,v))
    if (dist[v] > dist[u] + w(u,v) )
    dist[v] = dist[u] + w(u,v);
```



One run will update all one length paths

From S	j	i	d	...
Dist[]	INF	INF	INF	INF
Updated dist[]	8	7	6	

Second run will show us 2 length shortest paths

Dist[]		8-3=5	8-3=5 7-3=4	INF

See the dist[] d over i, d over j, i over j

Third run will show us 3 length shortest paths

Dist[]			5-3=2	INF

See the dist[] d over i over j

Run relaxation i times => at least i length shortest paths are seen. (though they can be seen earlier, based on the order of edges we process)

q) Maximum number of vertices on a Shortest path from s to v?

V-1

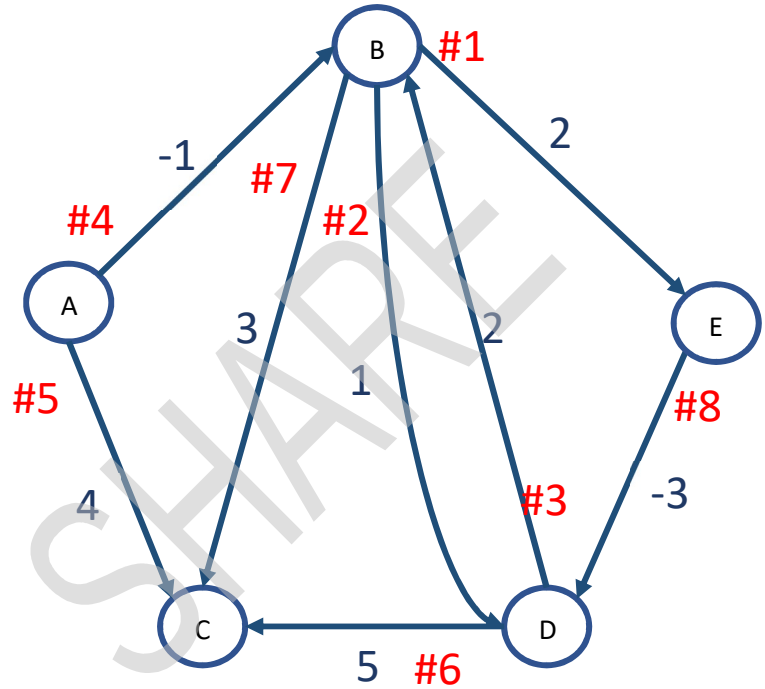
DO NOT SHARE

```
Run V-1 times
    for (each edge (u,v))
       if (dist[v] > dist[u] + w(u,v) )
       dist[v] = dist[u] + w(u,v);
```

Running this code V-1 times will make even the longest length (V-1 vertex passing) paths visible.

Bellman – Ford Pseudocode

```
d[] = INF
d[source] = 0
for (i:1-V)
for (each edge (u,v))
if (dist[v] > dist[u] + w(u,v) )
        dist[v] = dist[u] + w(u,v);
```



We have to number the edges since this time we are processing over the edges

Initialization

	A	B	C	D	E
Dist	INF	INF	INF	INF	INF
Dist	0	INF	INF	INF	INF

1st Run over edges

	A	B	C	D	E
Dist	0	INF	INF	INF	INF
Dist	0	-1	INF	INF	INF
Dist	0	-1	4	INF	INF
Dist	0	-1	2	INF	INF

Run #	Edge
1	4
1	5
1	7

DO NOT SHARE

2nd run

	A	B	C	D	E
dist	0	-1	2	INF	INF
dist	0	-1	2	INF	1
dist	0	-1	2	0	1
dist	0	-1	2	-2	1

Run #	Edge
2	1
2	2
2	8

3rd & 4th run

	A	B	C	D	E
dist	0	-1	2	-2	1
dist	0	-1	2	-2	1

No updates

dist	0	-1	2	-2	1

No updates

Run #	Edge
3	1..8
4	1..8

Shortest Paths from A	0	-1	2	-2	1

Q)If you don't have any update on ith run. Can we stop?

Yes! No update on ith run means no update on remaining runs. No need to check same conditions with same values. We can stop.

Q)How to understand if there is a negative cycle?

Run relaxation once more. If you are still able to relax dist[] =>negative cycle

DO NOT SHARE

Bellman - Ford Implementation

q) How to store the weights?

Adjacency list Array of size V Each item is vector of pair<dest, cost>

```
vector<pair<int,int>> adj[V];
dist[V] = INF
dist[source] = 0
for (i:0->V) {
    //Go over all the edges
    for (j:0->V)
         for (e:0->adj[j].size()) {
              d = adj[j][e].first;
              cost = adj[j][e].second;
              //Relax dist[d] if necessary
              if (dist[d] >      ?    )
                   dist[d] =     ?     ;
         }
}
```

Answer: dist[j] + cost

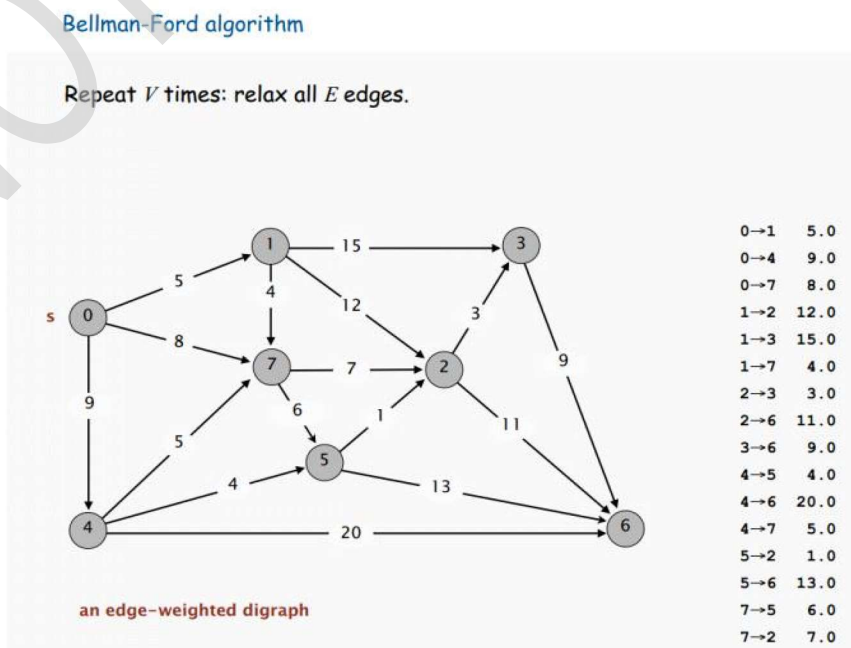
Q) Complexity?

O(VE)

For further practise run demo

on following graph.

(observe at ith run solutions less then i length are known)



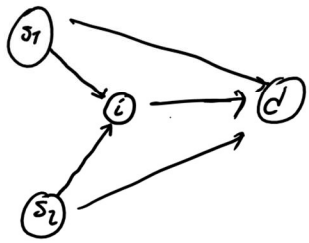
Sample Question: Job Hunt USACO '09 Nov

DO NOT SHARE

All Shortest Paths

Tuesday, December 5, 2017 6:40 PM

Given a graph $G(V,E)$, $w: E \to \mathbb{R}$ find $sp(u,v)$ for all $u,v \in V$.



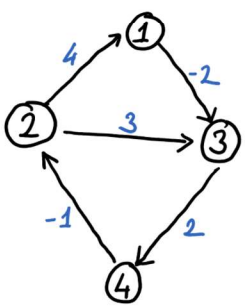
for each intermediate vertex i

for each vertex pair (s,d)

if $c(s,d) > c(s,i) + c(i,d)$

$c(s,d) = c(s,i) + c(i,d)$

```
for i: 0 -> V
    for each s: 0 -> V
        for each d: 0 -> V
            if mat[s,d] > mat[s,i] + mat[i,d]
                mat[s,d] = mat[s,i] + mat[i,d];
```



initial state

0	∞	-2	∞
4	0	3	∞
∞	∞	0	2
∞	-1	∞	0

$i=1$

0	∞	-2	∞
4	0	2	∞
∞	∞	0	2
∞	-1	∞	0

$i=2$

0	∞	-2	∞
4	0	2	∞
∞	∞	0	2
3	-1	1	0

$i=3$

0	∞	-2	0
4	0	2	4
∞	∞	0	2
3	-1	1	0

$i=4$

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

q) How can I detect negative cycle on Worshall?

a) if $i,j < 0$ & $j,i < 0 \Rightarrow$ negative cycle

$c(i,j) + c(j,i) < 0 \Rightarrow$ negative cycle $O(V^2)$

a) Find a better negative cycle detection $O(V)$

Cycle $\Rightarrow$ start and finish same vertex.

mat[i,j] gives min dist from i to j.

if mat[i,i] < 0 $\Rightarrow$ negative cycle.

DO NOT SHARE

Best Spot

Tuesday, December 5, 2017 7:05 PM

$P \leq 500$ Pastures

$F \leq P$ Favorite

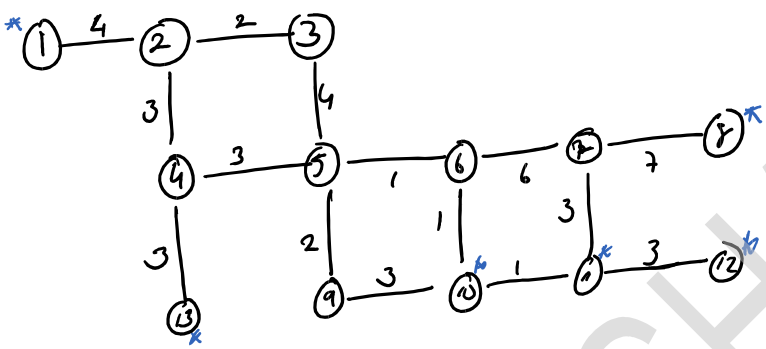
$C \leq 8000$ bi directional paths

$T_i \leq 892$ cost of path e_i

Find Pasture that has min avg dist to favorites.

1) Floyd-Warshall

2) Find the vertex with min total cost to favorite pastures



DO NOT SHARE

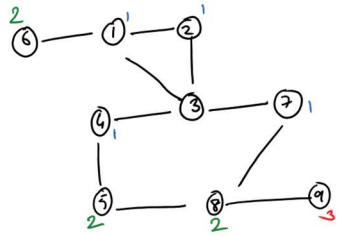
Shortest Path on unweighted graphs

Tuesday, December 12, 2017 3:24 PM

Given a graph $G\{V, E\}$ all edges have the same weight.

q) What is the most efficient way of finding shortest path from a source vertex $s \in V$?

a) BFS



run BFS from 3.

queue : 3 1 2 4 7 6 5 8 9

	1	2	3	4	5	6	7	8	9
dist	1	1	0	1	2	2	1	2	3

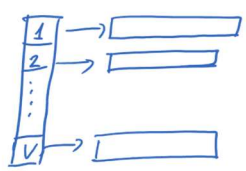
```
d[] = -1;
d[source] = 0;
q.push(source)
while (q ≠ ∅)
    cur = q.front()
    for each v ∈ adj[cur]
        if (d[v] == -1)
            d[v] = d[cur]+1;
            q.push(v)
```

a) Time complexity

$O(V+E)$ $\sum_{i=1}^{V} deg(v_i) = E$

q) Data structure to store the graph.

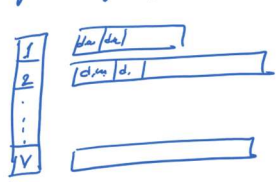
vector<int> adj[V];



q) What would be the data structure if we need to store weights

```
struct edge
{
    int dest;
    int dist;
};
```

vector<edge> adj[V];



vector<pair<int, int>> adj[V];

DO NOT SHARE

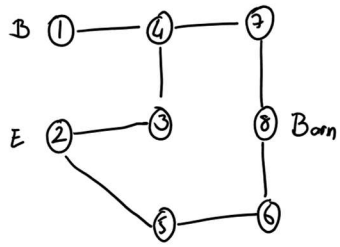
Piggy Back

Tuesday, December 12, 2017 5:22 PM

$N \leq 40.000$ fields

$M \leq 40.000$ paths

B, E, P



1 → 8

1 8

2 8

$X \rightarrow bfs(X)$

$d[1] * B$ ← Bessie coming field X

$d[2] * E$ Elsie coming field X

$d[N] * P$ B+E piggbad to field N.

ans

BFS

db 1 → X

de 2 → X

dn N → X

$$\min_{x=1 \to N} \left[bfs(1,x) * B + bfs(2,x) * E + bfs(N,x) * P \right]$$

```
vector <int>  bfs (int source)
{
    vector <int>  dist (n+1);
    queue<int>  q;
    q.push (source)
    visited [source] = 1;
    while !q.empty
        cur = q.front()
        for i: 0 → adj [cur].size()
            if (! visited [adj [cur] [i]])
            {
                q.push ( adj [cur] [i]
                visited [v] = 1;
                dist [v] = dist [cur] + 1;
            }
    return dist
}

vector<int>  adj [V];

for i: 0 → m
{
    cin u,v
    adj [u]. push_back (v)
    adj [v]. push_back (u)
}

mm = INF
for i: 1 → N
    minimum = min ( minimum, b * db[i] + e * de[i] * p * dn[i])
```

DO NOT SHARE

Shortest Path using heap

Remember Shortest path Dijkstra algorithm in short

```
d[] = INF
d[source] = 0
for (i:1-V)
    find min dist & unvisited vertex
    relax dist[]
```

Dijkstra Algorithm

```
    dist[V] = INF
    visited[V] = false
    dist[source] = 0
    for (i:0->V) {
    //Choose !visited & min dist vertex
       min=-1;
       for (j:0->V)
            if (!visited[j])
                 if (min=-1 || dist[min]>dist[j])
                     min=j;
       visited[min] = True
    //Relax dist[] with the new edges from curmin vertex
       for (int j:0->V)
            if (dist[j] >                          ){
                 dist[j] = dist[min] + mat[min][j];

            }
    }
```

q) What is time complexity

a) $O(V^2)$

We waste $O(V)$ each time to find the min. But we have seen data structures that give min in $O(1)$. (Don't forget this time insertion will be costly)

Problem: Given graph $G\{V,E\}$ and $w: E \to \mathbb{R}$. and a source vertex $s \in V$.

Find Shortest paths $s \to v$ $\forall v \in V$. with complexity better than $O(V^2)$

1) using priority-queue / min heap we can decrease cost of finding minimum from $O(V)$ to $O(1)$
2) Using adj list we can decrease relax cost from $O(V)$ to $O(\deg(\text{vertex}) \log V)$

```
d[] = INF
d[s] = 0
for i:0 -> V
    find min dist & unvisited
    Relax dist[]
```

$\Rightarrow$ V decreased to $O(1)$

$\Rightarrow$ V decreased to $O(\deg(v) \log V)$

$O(E \log V)$

(Recall $\sum_{i=0}^{V} \deg(v_i) = E$)

DO NOT SHARE

```
dist[] = INF
visited[] = false
dist[source] = 0
q.push({0,s})
while ( !q.empty()) {
   //get min dist vertex
   cur  = q.top().second;
   cost = q.top().first;
   q.pop();
   if (visited[cur]) continue;
   visited[cur] =true;
   for (i:0->adj[cur].size){
         a = adj[cur][i].first();
         b = adj[cur][i].second();
         if (dist[a] >  cost + b   ){
                 dist[a] = cost + b;
                 q.push({dist[a],a})
         }
  }
```

q) Complexity

O(VlogV+ElogV)

O((E+V)logV) If sparse graph E~cV => we say VlogV ~linear

q) Can we achieve same with adj matrix?

No. If we use adj matrix we must trace all the row => which makes O(V) complexity

q) When it comes to update the dist value in priority queue. PQ is sorted based on dist but we need update based on vertex value. =>Sequential Search?

Just add the new cost and vertex value to the PQ. Since smaller cost will be seen first we will set the vertex as visited and skip next time.

Q) Is it okay to run outer loop V times?

No. on some runs min dist will be a visited vertex. We will skip that run without deciding dist of a vertex.

DO NOT SHARE

Packet Delivery

Given 1 ≤ N ≤ 50,000 barns and 1 ≤ M ≤ 50,000 paths connecting them.

Each path has a cost 0≤ C_i ≤ 1,000

Find the Shortest path from barn 1 to barn N

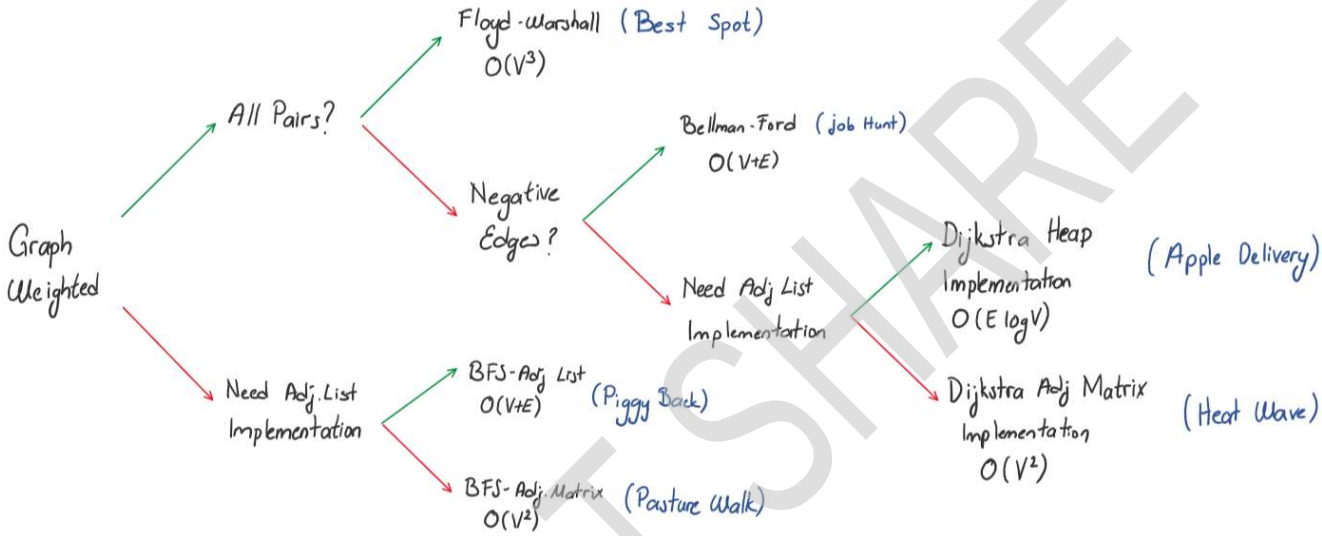
E~ cV => sparse graph

```
vector<pair<int, int> > adj[MAXN];
priority_queue<pair<int, int>,vector<pii>,greater<pii> > Q;

    for (int i = 1; i <= N; i++)
       best[i] = INF;

    for (int i = 0; i < M; i++) {
        cin>>a>>b>>c;
       adj[a].push_back(make_pair(b, c));
       adj[b].push_back(make_pair(a, c));
    }

    Q.push(make_pair(0, 1));
    while (!Q.empty()) {
       c = Q.top().first;
       a = Q.top().second;
       Q.pop();

       if (a == N)break;
       if (best[a] < c) continue;

       for (int i = 0; i < adj[a].size(); i++) {
          int next = adj[a][i].first, cost = adj[a][i].second;
          if (c + cost < best[next]) {
             best[next] = c + cost;
             Q.push(make_pair((c + cost), next));
          }
       }
    }
```

DO NOT SHARE

Shortest Paths Summary

situation	algorithm	time complexity	sparse ($E \sim cV$)	dense ($E \sim V^2$)
unweighted ($w=1$)	BFS	$O(V+E)$	$O(V)$	$O(V^2)$
non negative edges	Dijkstra	$O((E+V)\log V)$	$O(V \log V)$	$O(V^2)$
Negative edges	Bellman-Ford	$O(VE)$	$O(V^2)$	$O(V^3)$
All Shortest paths	Floyd-Warshall	$O(V^3)$	$O(V^3)$	$O(V^3)$



9) SP gives shortest path $s \to v$ $\forall v \in V$.

how to find shortest path $v \to s$ $\forall v \in V$?

if paths are bidirectional $\Rightarrow$ they are same

if paths are directional $\Rightarrow$ Reverse the edges and run Dijkstra.

Sample questions Silver cow party (sparty - 2007 February Silver)

Dueling GPS (gpsduel - 2014 Open Silver)

DO NOT SHARE