DATA STRUCTURES AND ALGORITHMS

Difference between Prims and Dijkstra Algorithm

Prim's Algorithm : $O(ELOGV)$ no need for cycle detection in Prims, but in Kruskal
Prim's algorithm constructs MST, all we care about is that Graph is connected.
It works only on undirected graphs. Prim's algorithm can work on negative weight cycles.

Due to the nature of these algorithms, in the case of a graph with a moderate number of edges, we should use Kruskal's algorithm. Prim's algorithm runs faster in a graph with many edges as it only compares a limited number of edges per loop, whereas Kruskal's starts by sorting all the edges in the list then going through them again to check if the edge is part of the minimal spanning tree (MST) or not. Hence although both algorithms have similar running time, the number of edges in the graph should be the main component when you are deciding between the algorithms. More edges compared to vertices, use Prim's, otherwise, use Kruskal's.

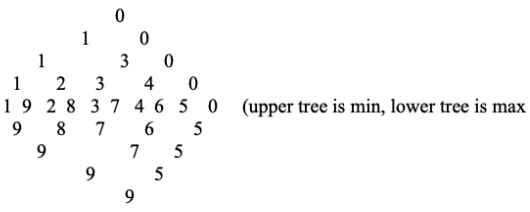
Dijkstra's Algorithm $O((E + V)logV)$
Dijkstra's algorithm creates the shortest path tree starting from the source node.
Length of any path from source to any other node is minimized.
Dijkstra's algorithm does not necessarily yield the correct solution in graphs containing negative edge weights. **BUT IT ALWAYS ENDS**

Sorting technique	Best case	Average case	Worst case
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort	O(nlogn)	O(nlogn)	$O(n^2)$
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)
Counting sort	Θ(k+n)	Θ(k+n)	Θ(k+n)
Radix sort	Θ(d(n+k))	Θ(d(n+k))	Θ(d(n+k))
Bucket sort	Θ(n)	Θ(n)	$\Theta(n^2)$

Radix Sort: d digits, base b and n numbers if we have.
Radix sort calls counting sort as subroutine d times.
Time Complexity : O(d(n+b))

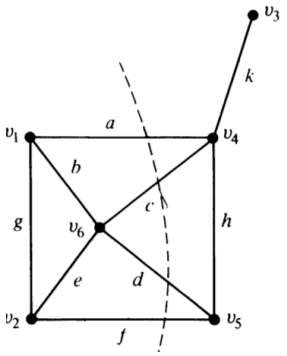
Counting sort:
O(n+k)
Space : O(n+k)

Second largest element n+ logn -2 comparisons.
Max and min element : 2(n-1) - n/2 comparisons basically.
Using tournament sort method, to find max and min element, we do n-1 comparisons to find max and n-1 comparisons again to find the minimum, but one step is we are overcounting, that is the n/2 steps
So we take 1.5n - 2 comparisons.



(upper tree is min, lower tree is max

You can clearly see we have 3n-2 comparisons. This is known as tournament sort algorithm
Second maximum



Number of comparisons in Radix sort

D- digits B-base N- numbers n*d*b is the answer
The number of min heaps on n nodes are (n=7)
7!/7*3*3 = 6!/9 = 80 and so on

If the DFS finishing time f[u] < f[v] for two vertices u and v in a directed graph G, and u and v are in the same DFS tree in the DFS forest, then u is an ancestor of v in the depth first tree ___[True/False] --- FALSE

Observe that

$$S(n) = \frac{I(n)+n}{n}, \quad U(n) = \frac{E(n)}{n+1}.$$

It can be proved by induction that

$$E(n) = I(n) + 2n.$$

Hence,

$$U(n) = \frac{I(n)+2n}{n+1}.$$

Eliminating $I(n)$ from Eqns. 1.1 and 1.3,

$$nS(n) = (n+1)U(n) - n,$$

$$\text{or,} \quad S(n) = \left(1 + \frac{1}{n}\right) U(n) - 1.$$

In a connected graph G, a *cut-set* is a set of edges† whose removal from G leaves G disconnected, provided removal of no proper subset of these edges disconnects G. For instance, in Fig. 4-1 the set of edges $\{a, c, d, f\}$ is a cut-set.

Algorithm SCC(G)
1. DFS(G)
2. find $G^T = (V, E^T)$ where $E^T = \{(v,u) : (u,v) \in E\}$
3. rearrange the vertices of G^T to get $G^T = (V^T, E^T)$ such that the vertices in G^T are in decreasing order of their finishing times obtained in step 1
4. DFS(G^T)
5. output the vertices of each DFS tree produced by step 4 as an SCC

Time complexity: Each of step 1 and step 2 takes $\Theta(V+E)$ time. Step 3 needs $\Theta(V)$ time when each vertex is relocated when its DFS finishes in step 1. Step 4 again takes $\Theta(V+E)$ time, thereby giving total time complexity as $\Theta(V+E)$.

Lemma 3.2 *If two vertices are in an SCC, then no path between them ever leaves the SCC.*

Proof: Let u and v belong to the same SCC. Let w be an arbitrary vertex in the path $u \rightsquigarrow w \rightsquigarrow v$. Since there is a path $v \rightsquigarrow u$, we have the path $v \rightsquigarrow u \rightsquigarrow w$, which implies there are paths $w \rightsquigarrow v$ and $v \rightsquigarrow w$. Hence w is in the SCC of v (and of u, there of).

Theorem 3.4 *In any DFS, all vertices in the same SCC are placed in the same DFS tree.*

$$\text{vertex connectivity} \leq \text{edge connectivity} \leq \frac{2e}{n},$$

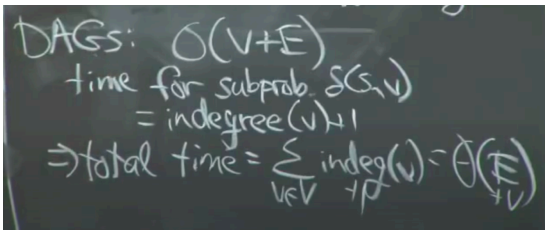
$$\text{maximum vertex connectivity possible} = \left\lfloor \frac{2e}{n} \right\rfloor.$$

	$(\log N)^{\frac{1}{2}}$ find	N insert	$(\log N)^{\frac{1}{2}}$ delete	$(\log N)^{\frac{1}{2}}$ decrease-key
Unsorted Array	$O(N(\log N))$	$O(N)$	$O(\log N))$	$O(\log N))$
Min-heap	$O(N(\log N))$	$O(N\log N)$	$O(\log N))$	$O((\log N))$
Sorted Array	$O((\log N))$	$O(N^2)$	$O(N(\log N))$	$O(N(\log N))$
Sorted doubly linked-list	$O(N(\log N))$	$O(N^2)$	$O((\log N))$	$O(N(\log N))$

The minimum number of edges $\lambda(G)$ whose deletion from a graph G disconnects G, also called the line connectivity. The edge connectivity of a disconnected graph is 0, while that of a connected graph with a graph bridge is 1.

Let $\kappa(G)$ be the vertex connectivity of a graph G and $\delta(G)$ its minimum degree, then for any graph,

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

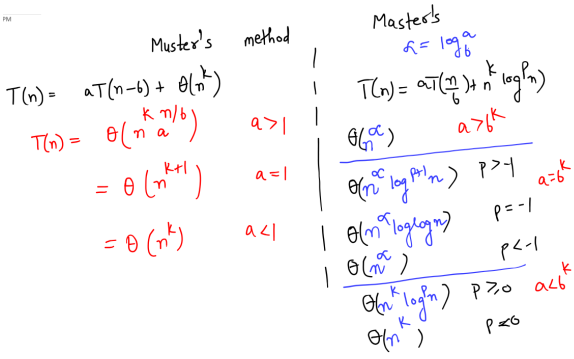


THERE CAN BE AT MOST $ceil\left(\frac{n}{2^{h+1}}\right)$ nodes of height H in any N element heap.

LOG(N!) COMPARISONS MINIMUM REQUIRED

D is TRUE as maximum element in a min-heap will be at the last level and it can be any one of the possible $n/2$ elements in the last level.

The comparison based sorting algorithms best case time complexity is Omega(nlogn) and worst case time complexity is O(n^2). Due to LOGN! rule



Suppose that you store 5 records in a hash table of size 9 by using open addressing with linear probing, and suppose that you have a good hash function so that the probability that a key is hashed into any of the 9 slots is $1/9$. For a particular slot in the hash table, what is the probability that this slot is empty, that is, none of the 5 keys hashes into this slot (rounded to 2 decimal points)?

72 views

Consider a hash table with n buckets, where external (overflow) chaining is used to resolve collisions. The hash function is such that the probability that a key value is hashed to a particular bucket is $\frac{1}{n}$. The hash table is initially empty and K distinct values are inserted in the table.

4.8k views

A. What is the probability that bucket number 1 is empty after the K^{th} insertion?
B. What is the probability that no collision has occurred in any of the K insertions?
C. What is the probability that the first collision occurs at the K^{th} insertion?

Let G be a simple undirected graph, T_D be a DFS tree on G, and T_B be the BFS tree on G. Consider the following statements.

Statement I: No edge of G is a cross with respect to T_D
Statement II: For every edge (u,v) of G, if u is at depth i and v is at depth j in T_B then $|i-j| = 1$

In both directed and undirected graphs as long as all the edge weights are the same BFS runs in $O(|V| + |E|)$ and gives the shortest path from the given source to all the other vertices.

Answer: In open addressing all keys go to the hash table itself. We have 5 records and so 5 slots in hash table will be taken. So, probability that a particular slot will be empty $= \frac{^8C_5}{^9C_5} = \frac{4}{9} = 0.444$

The expected number of comparisons in a successful linear search is $\lceil n/2 \rceil$ as the successful item is equally likely to be in any of the n locations.

In best case both linear search and binary search can work in $O(1)$ time as the searched item should be the first element.

In worst case also binary search can work in $\Omega(\log n)$ comparisons as after each search half the elements get eliminated.

In a DFS traversal, presence of a back edge - not cross edge -- indicates the existence of a cycle.

A node can be visited more than once in a DFS if there are multiple paths to it from the source which need not be a cycle.

The out-degree of each vertex will be the number of vertices in its adjacency list. So, for all the vertices, we can get the out-degree in $\Theta(m + n)$ time where m is the number of edges. Since $m = \Theta(n^2)$, this will give $\Theta(n^2)$.

In-degree can also be computed in $\Theta(m + n)$ time by maintaining an array of size n for the vertices and then traversing all the adjacency lists. Whenever we encounter a vertex we increment the corresponding entry in the array.

Topological sort uses DFS and so it can run in $\Theta(|V| + |E|)$ time when the graph is given in Adjacency List representation whereas it'll take $\Theta(|V|^2)$ if graph is given in Adjacency Matrix representation.

Topological sorting is possible only in acyclic graphs and there can be multiple possible topological orderings.

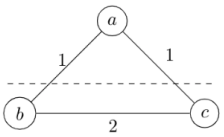
If in a depth-first traversal of a graph G with n vertices, k edges are marked as tree edges, then the number of connected components in G is $n - k$ as each connected component of l vertices gives $l - 1$ tree edges. i.e.,

$$k = \sum_{i=1}^{d} l_i - 1 = n - d,$$

where d is the number of connected components in G. So, $d = n - k$ and option A is false.

Option B is true and C is false as we can use BFS for cycle detection and though the time complexity of BFS is $\Theta(|V| + |E|)$, we can do this in $\Theta(|V|)$ as once a cycle is detected we can stop the BFS and this should happen within $O(|V|)$ time.

Option B is false as this will include multiple light edges crossing a cut if they are having the same weights.
Option D is also false as can be seen in the counter example given below.

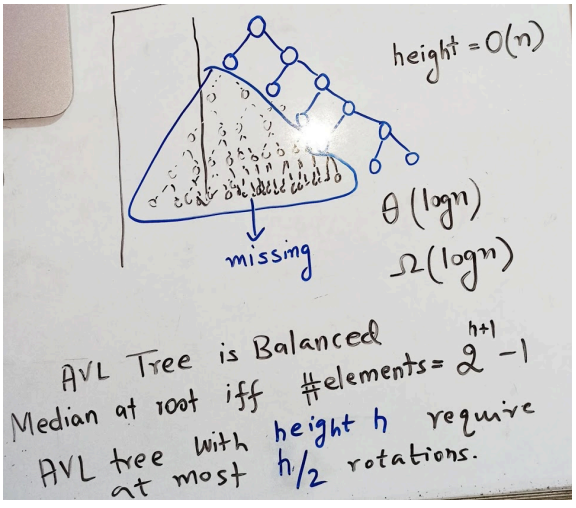


Here, for the cut set $\{a\}, \{b, c\}$ we have two light edges of weight 1 but there is a single MST $b - a - c$.

- getw() and putw() are related to FILE handling.
- putw() is use to write integer data on the file (text file).
- getw() is use to read the integer data from the file.
- getw() and putw() are similar to getc() and putc(). The only difference is that getw() and putw() are especially meant for reading and writing the integer data.

B. $\{(u, v)$
: there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing it
$\}$
forms a MST

A. Dijkstra's algorithm runs in $O(|E| \lg |V|)$ if priority queue is implemented with a binary min-heap
B. For dense graphs ($|E| = \Omega(|V|^2)$) Dijkstra's algorithm performs better if priority queue is implemented with an array
C. Dijkstra's algorithm has a worst case time complexity of $O(|V|^2)$
D. Dijkstra's algorithm has a worst case time complexity of $O(|V| \lg |V| + |E|)$ when implemented with a Fibonacci heap



Bellman-Ford algorithm always finds the correct shortest path even when edge weights are negative as long as there is no negative weighted cycle. If a negative weight cycle is not reachable from source, it wont detect it. Time complexity of Bellman Ford algorithm is $O(|V||E|)$ and is not better than that of Dijkstra's.

Correct answer: A, C.

A. Always finds the correct shortest path even when edge weights are negative as long as there is no negative weighted cycle
B. Always finds a negative weighted cycle, if one exists.
C. Finds whether any negative weighted cycle is reachable from the source.
D. Performs better than Dijkstra's algorithm for dense graphs

The reference is there in Cormen book. For asymptomatic growth taking a large value works but how large the value should be depends on the function. Also n, n/2 have same asymptomatic growth rate though they never cross each other when plotted in a graph.

In fact, C *doesn't* guarantee that a 1 will be shifted in when you right-shift a negative signed number; the result of right-shifting a negative value is implementation-defined.

However, if right-shift of a negative number *is* defined to shift in 1s to the highest bit positions, then on a 2s complement representation it will behave as an *arithmetic shift* - the result of right-shifting by N will be the same as dividing by 2^N, rounding toward negative infinity.

Which of the following function can be used to write only integer numbers?	fprintf()	putw()

What will be the output of the following program, for k=8?
 	
#include<stdio.h>
void main()
{
int i= 2, j=3, k;
scanf("%d",&k);
switch(k%2)
{
case 5<4:printf("k is even"); break;
case 10>9: printf("k is odd"); break;
default : printf("end");
}
}
	Compilation error	k is even

As for k=8, switch executes false condition which is condition 5<4 as 5<4 is false

(E) const and volatile are independent i.e. it's possible that a variable is defined as both const and volatile.

Answer: (E)

Explanation: In C, const and volatile are type qualifiers and these two are independent. Basically, const means that the value isn't modifiable by the program. And volatile means that the value is subject to sudden change (possibly from outside the program). In fact, C standard mentions an example of valid declaration which is both const and volatile. The example is "extern const volatile int real_time_clock;" where real_time_-clock may be modifiable by hardware, but cannot be assigned to, incremented, or decremented. So we should already treat const and volatile separately. Besides, these type qualifier applies for struct, union, enum and typedef as well.

```
#include "stdio.h"
int main()
{
   char a[] = { 'P', 'Q', 'R', 'S' };
   char* p = &a[0];
   *p++;
   printf("%c ", *++p);
}
```

What will be the output of the above question:

Here pointer is first pointing to P, now *p++ means now pointing to Q
*++p means, although precedence of * is greater but it can only be applied once we pre increment our pointer which will now point to R and now we will just print the value at the location where R is stored.

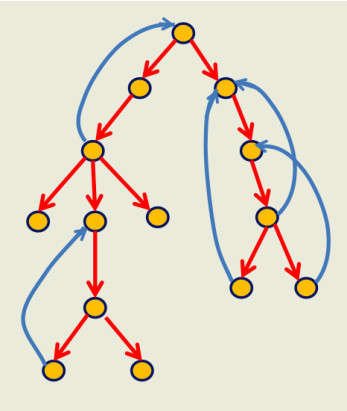
Let (*x*,*y*) be a non-tree edge.
Let *x* get visited before *y*.
Question:
If we remove all vertices visited prior to *x*, does *y* still lie in the connected component of *x* ?
Answer: yes.
➔
DFS pursued from *x* will have a path to *y* in **DFS** tree.
Hence *x* must be ancestor of *y* in the **DFS** tree.

An $O(m+n)$ time algorithm

- A formal **characterization** of the problem.
  (**articulation points**)
- Exploring **relationship** between articulation point & DFS tree.
- Using the relation **cleverly** to design an efficient algorithm.

Definition: A connected graph is said to be **biconnected** if there does not exit any vertex whose removal disconnects the graph.

Question: When can **root** be an **a.p.** ?
Answer: Iff it has **two or more** childrer



A trivial algorithms for checking bi-connectedness of a graph

- For each vertex *v*, determine if G\{*v*} is connected
  (One may use either **BFS** or **DFS** traversal here)

Time complexity of the trivial algorithm : O(mn)

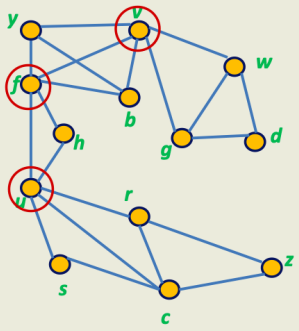
What will be output of the following program?

```
#include <stdio.h>
int main()
{
char str[] = "%d %c", arr[] = "bcdefa";
printf(str, 0[arr], 2[arr + 3]);
return 0;
}
```

Note: ASCII of a is 97

This is printing 98 a
0[arr] will print integer value of b which is 98
bcdefa is a 2d array 2[arr+3] is same as arr[2][3]
Which is same as third item of second 2d array which is pointing to a
According to c main should return int value but that is not a must

Detecting if a graph is biconnected or not is the application of DFS Tree. The trivial algorithm is brute force, whilst the other algo is divide and conquer.



This graph is NOT biconnected

The removal of any of {*v*,*f*,*u*} can destroy connectivity.

v,*f*,*u* are called the **articulation points** of *G*.

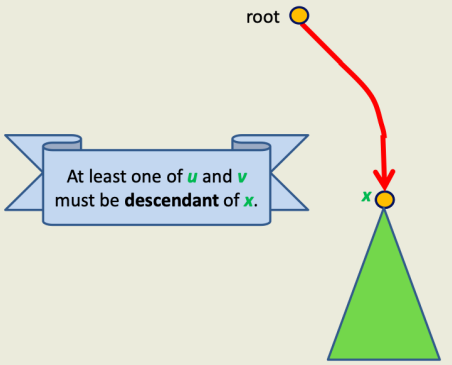
Articulation points and DFS

Let *G*=(*V*,*E*) be a connected graph.
Perform **DFS** traversal from any graph and get a DFS tree *T*.

- No leaf of *T* is an **articulation point**.
- root of *T* is an **articulation point** if and only if it has more than one child.
- For any internal node ... ??

Theorem1 : An internal node *x* is **articulation point** **if and only if**
it has a child *y* such that
there is **no** back edge
from **subtree**(*y*) to any ancestor of *x*.

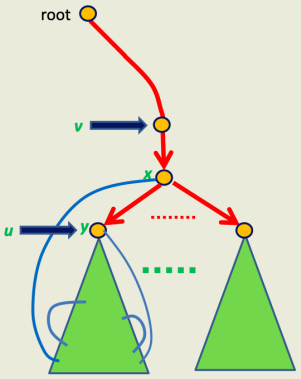
conditions for an internal node to be articulation point.



At least one of *u* and *v* must be **descendant** of *x*.

Where will *u* and *v* be relative to *x* ?

Necessary condition for *x* to be articulation point



Necessary condition:
x has **at least** one child *y* s.t.
there is **no** back edge
from **subtree**(*y*) to **ancestor** of *x*.

Is this condition **sufficient** also ?

yes.

7) What is the output of this C code?

```
#include <stdio.h>
#define nptl(m, n) " m ## n "

int main()
{
    printf("%s\n", nptl(k, l));
}
```

```
char *p = "wikipedia"; // valid C, deprecated in C++98/C++03, ill-formed as of C
p[0] = 'W'; // undefined behavior
```

Integer division by zero results in undefined behavior:[11]

```
int x = 1;
return x / 0; // undefined behavior
```

Certain pointer operations may result in undefined behavior:[12]

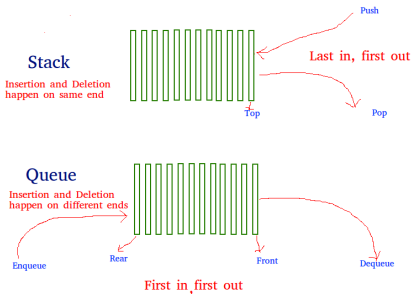
```
int arr[4] = {0, 1, 2, 3};
int *p = arr + 5;  // undefined behavior for indexing out of bounds
p = 0;
int a = *p;        // undefined behavior for dereferencing a null pointer
```

In C and C++, the relational comparison of pointers to objects (for less-than or greater-than comparison) is only defined if the pointers point to members of the same object, or elements of the same array.[13] Example:

```
int main(void)
{
  int a = 0;
  int b = 0;
  return &a < &b; /* undefined behavior */
}
```

This will print **m ## n** always
Undefined behaviour in C:
Here we are calculating the length of the linked list and then printing the len-126 + 1 th element from the beginning.

The number of operations and time complexity required for searching 126th element from end in an singly linked of size n is?	126, O(n)	n-125, O(n)



Consider a singly linked list having 2 pointers pointing first & last elements of the list. If n is length of singly linked list, which among the following operation takes O(n) time to perform in all cases (best as well as worst) ?

1. delete 1st element
2. interchange the 1st and 2nd elements of list
3. ✔ delete last element
4. add element at the end of list

Queue using stack
Enqueue O(n)
Dequeue O(1)

Infix to postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Autumn 2016

30

consider the following infix expression
(P+Q*R^S)/(P+Q^S)
What is the minimum size of stack required to convert this infix expression to postfix expression?
Note: For this question consider ^ as an exponential operator having the same precedence as * and /

Use of *goto* takes out the structural decomposition of the code and hence it becomes very difficult to verify or debug the code. As far as performance or memory impact is concerned, *goto* has no effect on them.

(1) Deleting a node whose location is given.

REASON:

In this question, we can simply eliminate the (2) and (4) from our choice, since in both the cases we need to visit every node one after other. Doubly LL will serve no extra good in these two cases as we need to proceed from starting node and go until the last note in LL or vice-versa .

(2) To insert a new node in the LL after some specific node, we need not go back to the previous node in the LL. And thus, doubly LL will serve no purpose in this case too.

At last, in the case of (1) we are supposed to delete the node (say x) whose location is given. Which means that before we remove node x from the LL we need to store the address of next node in the list, linked to node x in the address field of node just before node x (which currently store the address of node x). Since we need to go back-and-forth in the list during this operation, doubly linked list will be a good help in increasing the efficiency by decreasing the time required for the operation.

```
#include<stdio.h>
typedef int (*fun)(int);
 int fun2(int a)
{
return (a+1);
}
 int main()
{
 fun p1 = NULL, p2 = NULL;
p1 = &fun2; /* (1) */
p2 = fun2; /* (2) */
(*p1)(5); /* (3) */
p2(5); /* (4)*/
return 0;
}
```

Which of the following line is correct about above code?

Question: 10
Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?

1. ✔ Deleting a node whose location in given
2.

Deleting a node whose location is given : DOUBLY LINKED LIST IS MORE EFFICIENT IN THIS.
Function running normally no error nothing

Structures initialise the value to 0 by default
Generally function pointer points to code not data
Memory allocation and deallocation is not possible using function pointers.

When all pair of vertices are reachable then that is known as SCC

In computer **programming**, **aliasing** refers to the situation where the **same memory location** can be accessed using different names. For instance, if a function takes two pointers A and B which have the same value, then the name **A aliases** the name **B**.

The Loop Invariant is a property(among program variables) of a program loop that is true before and after each iteration of the loop.
It should be somewhat the *goal* of the loop.
Here our goal is to calculate X^Y.

Intuitively, we can see that among variables X, Y, a, b, res-

- The loop terminates when b becomes 0.
- After the loop terminates, X and Y remain unchanged and res contains the calculated value of X^Y

Loop invariant has to hold truth value true, before and after a loop

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are `&&`, `||`, `?:`, and `','`.) For example, in a statement like

```
x = f() + g();
```

`f` may be evaluated before `g` or vice versa; thus if either `f` or `g` alters a variable on which the other depends, `x` can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));   /* WRONG */
```

can produce different results with different compilers, depending on whether `n` is incremented before `power` is called. The solution, of course, is to write

Option B is correct as for the given inputs the program fragments does swapping of the values. Call-by-reference implies that the swapped values are reflected in the calling function as well.

PS: Procedure G is not doing proper swapping for all input values as for negative values, there is a chance of underflow in $c - d$.

```
int foo(int val) {
    int x=0;
    while(val > 0) {
        x = x + foo(val--);
    }
    return val;
}
```

```
int bar(int val) {
    int x = 0;
    while(val > 0) {
        x= x + bar(val-1);
    }
    return val;
}
```

All modern programming languages are CSL. Because they contain two features which cannot be handled by PDA.

The features are:

- variable declared before use and
- matching formal and actual parameters of functions.

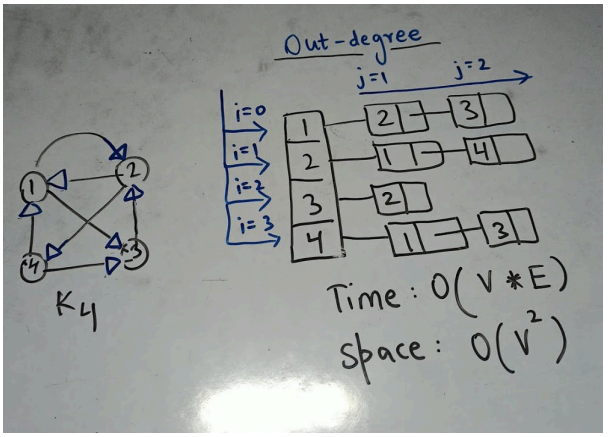
Arbitary goto,recursive call and repeat may enter infinite loop,and hence terminates program may not be able to answer if 'the program does terminate'.

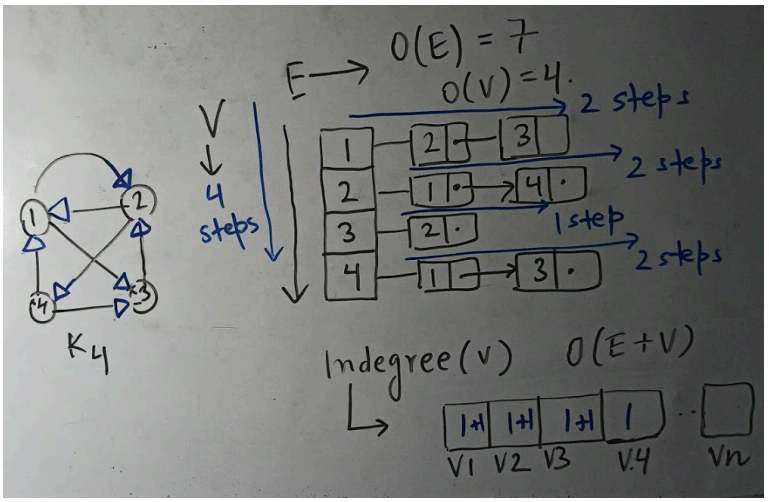
As per the rank-nullity theorem , the rank of a matrix and the nullity of a matrix add up to the number of columns in the matrix.

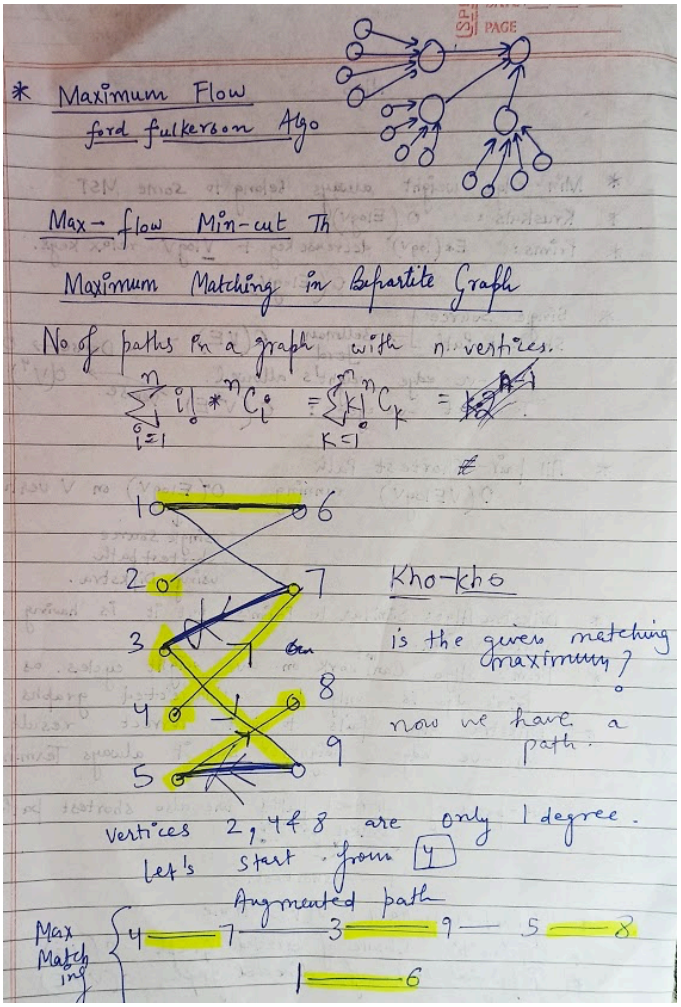
Specifically , rk(A)+nul(A)=n. (for a matrix of dimensions mXn)

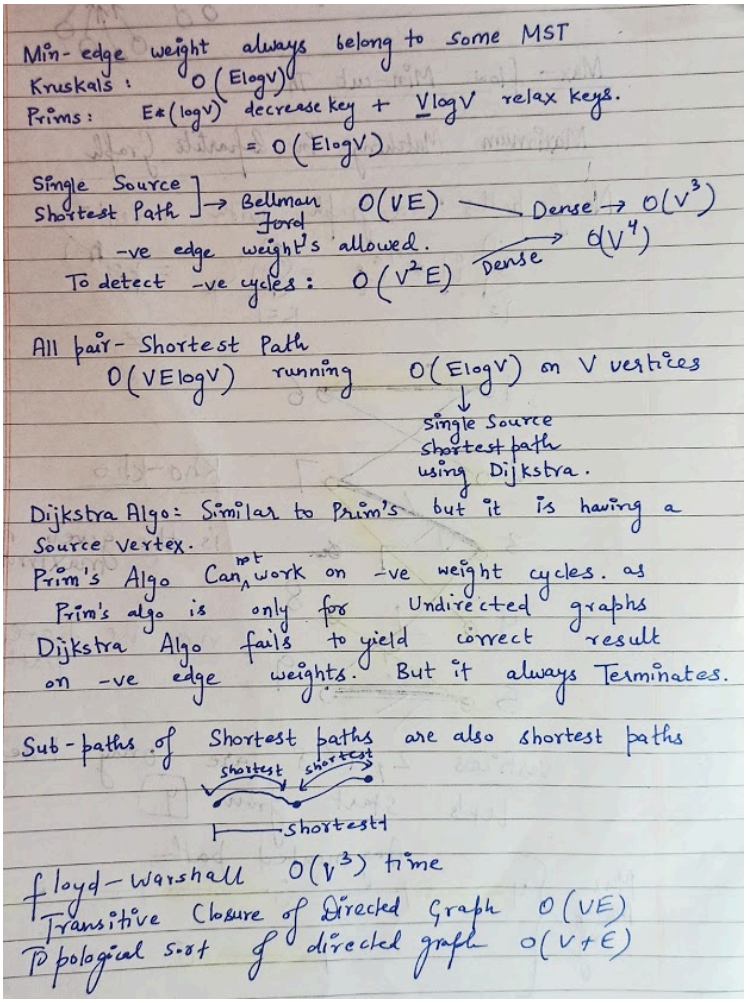
This is a very beautiful question on abnormal termination and infinite loop respectively
Foo incurs abnormal termination, as val is getting decremented every time
foo(3--) calls foo(3--) again and again and keeps on doing this until the stack is full, but note, here it is not an infinite loop as at every step we are making sure that v is getting decremented to 2 then 1 then 0. This problem is known as stack overflow
Bar on the other side, creates a problem as bar3 calls bar2 which inturn calls bar1 which calls bar0, now bar1 again calls bar0 as while condition becomes false, bar1 keeps on calling bar0 which creates issue, and we say infinite loop is there , but dont worry we have stack, which is going to tell us that infinite loop.
https://www.cs.bgu.ac.il/~comp161/wiki.files/ps9.pdf

> If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a constant-initializer which shall be an integral constant expression. In that case, the member can appear in integral constant expressions within its scope. **The member shall still be defined in a namespace scope** if it is used in the program and the namespace scope definition shall not contain an initializer.









1. int c = (a,b) ; // c= 0. first value of a is assigned to c and then overwritten by b. hence value of c is b.

2.

int c =(a++,b++) ;

printf("%d\n%d\n", c,b); // c=0 and b=1

3. In ternary operator:

int c =(a++,b++) ? a: b; // here value of b is considered as ternary condition to check whether a or b assigned to c. And then if condition is true execution will come to 'a'. Here value of a will be a+1, as it was post increment.

int c =(a++,b++) ? a: b;

Now coming to our program.

() have highest preference, hence it will be executed first. value of () will be value of b++. Value of b++ will be value of b since it is a post increment, it will be incremented afterwards, hence it is 0. Now as ternary condition is false, 2nd part of : will be returned to c. Now as we come out of (), b will be incremented now as b+1= 0+1=1. Hence the final value returned to c will be 1.