Topic 1: Complexity

Session 1: Running Time of Program

Erratum: Complexity - Running time of Program: Empirical and…

Analysis of Linear Search Algorithm A (Successful Search)

```
      i=0   i<size   i++    S[i]==num     found=true       break
       1 + 1*(n) + 1*(n-1) + (2*n) + 1 + 1 =  4n + 2
Ts(n) = 4n + 2
```

Topic 1: Complexity

Session 3: Order of Running Time of an Algorithm

Erratum: S3

On slide 14, Example 2: The value of b should be 3/5 and not 5/3

Topic 2: Sorting

Session 4: Comparison Based Sorting

Counting Sort

As we have seen in the lectures, the lower worst case bound for any comparison based sort is O(n*log(n)), where n is the size of the input array. Can we do any better?

It turns out that in some particular cases we can. If all elements are in the range 0 to k, we can sort the array in O(n+k) time. This is achieved using counting sort.

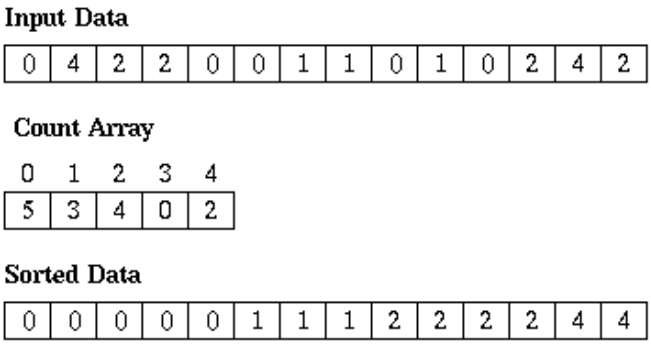
Description

As we know that all the numbers are in the range 0 to k, we can just make an array of size k+1 to store the count of each number.

In the end we can just print each element the number of times it occurs.

Time Complexity: O(n+k)

In the following image e have k = 5 and n = 14.



Algorithm

Step 1: Take an array of size k + 1 with all values initialised to 0. Call this the count array.

Step 2: Scan the input array from left to right. Suppose x is the current element that is scanned, then increase count[x] by 1.

Step 3: The count array gives the final result.

C++ Implementation

```cpp
#include <bits/stdc++.h>
using namespace std;

void countSort(int a[], int n, int k){

        int count[k+1];
        for(int i=0;i<=k;i++) count[i] = 0;
        // this stores the count of each element

        for(int i=0;i<n;i++){
                count[a[i]]++;
        }

        int c = 0;
        for(int i=0;i<=k;i++){
                for(int j=0;j<count[i];j++) {a[c] = i; c++;}
        }
}

int main(){
        int n;
        cin>>n;
        int a[n];
        for(int i=0;i<n;i++) cin>>a[i];
        countSort(a,n,100);
        for(int i=0;i<n;i++) cout<<a[i]<<" ";cout<<endl;
}
```

Topic 2: Sorting

Session 4: Comparison Based Sorting

Radix Sort

We saw in counting sort how to sort an array of size n if all elements are in the range 1 to k, in 0(n+k) time. What if the elements vary from 1 to n^2. Counting sort will take $O(n^2)$ time in this case. Can we sort such an array in linear time? Radix sort comes to our rescue.

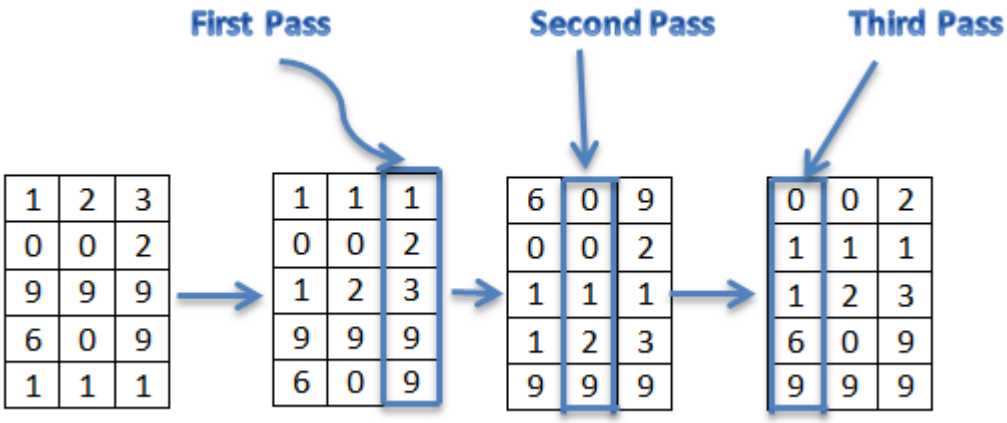
Radix sort is a very interesting sorting method. It is used to sort data by grouping elements on the basis of digits that share the same significant position. That is, we sort digit by digit, from the least to the most significant digit. Counting sort is used as a part of radix sort.

Description

As we can see in the image, we sort the numbers on the basis of the least significant digit to the most significant digit (in base b).

Suppose the numbers vary from 1 to k. The number of digits in base b is $O(\log_b(k))$. We use counting sort to sort the numbers on the basis of these digits (we know that the digits range from 0 to b-1 in time O(n+b). Thus the overall complexity is $O((n+b)*\log_b(k))$.

Note that if $k = n^2$, we choose b = n and we get complexity O(n)!



Algorithm

Step 1: Find the maximum possible most significant digit. Call this D. Note that $D = \log_b(k)$ where b is the chosen base and k is the maximum value that the numbers of the array can take.

Step 2: Iterate from the least significant to the most significant digit (i = 0 to D).

Step 3: In each iteration sort the numbers in the given array on the basis of their digit at position i. Note that the digits vary from 0 to b-1 since we have represented each number in base b.

Step 4: Make sure that for numbers that have the same digit at position i, their relative position remains the same. (Think why?)

Step 4: The array we obtain finally is sorted.

C++ Implementation

```cpp
#include <bits/stdc++.h>
using namespace std;

void radixSort(int a[], int n, int b, int k){
        // b is the base in which we are working

        for(int p=1; p<k; p*=b){

                int count[b];
                for(int i=0;i<b;i++) count[i] = 0;

                for(int i=0;i<n;i++){  // (a[i]/p)%b gives the current digit
                        count[(a[i]/p)%b]++;
                }

                for(int i=1;i<b;i++) count[i] += count[i-1];
                // This gives us the cumulative sum and hence the actual
                // position in array for the variables

                int newarr[n];

                for(int i=n-1;i>=0;i--){
                // start from end to maintain that:
                // for same count, relative position in array remains same
                        newarr[count[(a[i]/p)%b] - 1] = a[i];
                        count[(a[i]/p)%b]--;
                }

                for(int i=0;i<n;i++) a[i] = newarr[i];

        }
}

int main(){
        int n;
        cin>>n;
        int a[n];
        for(int i=0;i<n;i++) cin>>a[i];
        radixSort(a,n,n,n*n);
        for(int i=0;i<n;i++) cout<<a[i]<<" ";cout<<endl;
}
```

Topic 2: Sorting

Session 5: Insertion Sort

Shell Sort

Shell sorting is a very interesting sorting method which basically uses insertion sort to sort elements. So if it uses insertion sort then how it is better than insertion sort? Well that's the interesting part. To get the intuition, shell sort applies insertion sort first to small sequences of largely interleaved elements which makes the array kind of partially sorted and then it repeatedly reduces the interleaving between the elements of sequence and reapplies insertion sort untill we get the sorted array. Already confused ? don't worry, It is lot to grasp in one sentence.

Problem with Insertion sort

Insertion sort is a great algorithm, because it’s very intuitive and it is easy to implement, but the problem is that it makes many exchanges for an element which is very far from its correct place. Thus these elements may slow down the performance of insertion sort a lot. That is why in 1959 Donald Shell proposed an algorithm that tries to overcome this problem by comparing items of the list that lie far apart

Description

First lets define h-sorted list. We say a list is h-sorted if starting from any position every hth element is in sorted order. *For e.g. 2 1 4 3 is 2-sorted list while 3 1 2 4 is not since starting from initial position every second element i.e. 3 and 2 is not in sorted order.*

Shell sort takes a gap sequence (lets call it gap_seq). This sequence must be a decreasing sequence and must end with 1. Now for each element **gap_seq[i]** in the gap sequence, shell sorts makes the array gap_seq[i] sorted by sorting every gap_seq[i] interleaved elements.

For e.g. take gap_seq = {5, 3, 1}

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

As the example illustrates, the subarrays that Shellsort operates on are initially short; later they are longer but almost ordered. In both cases insertion sort works efficiently.

How to choose gap size

Not a cool thing about Shell sort is that we've to choose "the perfect" gap sequence for our list. However this is not an easy task, because it depends a lot of the input data. The good news is that there are some gap sequences proved to be working well in the general cases.

Shell Sequence

Donald Shell proposes a sequence that follows the formula FLOOR(N/2k),
then for N = 1000,
we get the following sequence: [500, 250, 125, 62, 31, 15, 7, 3, 1]
Shell sequence for N=1000: (500, 250, 125, 62, 31, 15, 7, 3, 1)

Pratt Sequence

Pratt proposes another sequence that's growing with a slower pace than the Shell's sequence. He proposes successive numbers of the form 2p3q or [1, 2, 3, 4, 6, 8, 9, 12, …].

Pratt sequence: (1, 2, 3, 4, 6, 8, 9, 12, ...)

Knuth Sequence

Knuth in other hand proposes his own sequence following the formula (3k – 1) / 2 or [1, 4, 14, 40, 121, …]

Knuth sequence: (1, 4, 13, 40, 121, ...)

Of course there are many other gap sequences, proposed by various developers and researchers, but the problem is that the effectiveness of the algorithm strongly depends on the input data.

Pseudo Code

Source : https://en.wikipedia.org/wiki/Shellsort

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps)
{
    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped order
    # keep adding one more element until the entire array is gap sorted
    for (i = gap; i < n; i += 1)
    {
        # add a[i] to the elements that have been gap sorted
        # save a[i] in temp and make a hole at position i
        temp = a[i]
        # shift earlier gap-sorted elements up until the correct location for
a[i] is found
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }
}
```

Topic 2: Sorting

Session 6: Heap sort

Optimal File Merging

You have been given n files with sizes a1, a2, ..., an. As in the merge step of merge sort, it is given that merging two files of sizes ai and aj takes time equal to ai+aj and that the size of the resulting file is also ai+aj. Assuming you can merge files only two at a time, what is the minimum time required to merge all the files?

Input :

The first line contains an integer n.

The next line contains n integers a1, a2, a3, ... , an

Output : A single integer - the minimum required time

Examples :

Input:	Output:
3 2 3 4	14

Example Testcase 2 :

Input:	Output:
5 1 3 6 4 3	38

Solution

A naive approach is to check all possible orders of merging. But clearly this takes exponential time.

Consider the following example. Suppose there are 3 files with sizes 2,3,4.

Possible methods:

Merge file 2 and 3 to get 2,7 - Time taken = 7

Then merge with file 1 - Time taken = 9

Total time = 7 + 9 = 16

Note that if we merge file 1 and 2 first total time taken is (2+3) + ((2+3) + 4) = 14

It is seen that at each step we should merge the 2 files that have the smallest sizes since these sizes are also added in time taken later (when merging with other files).

An efficient solution can be implemented using a min-heap (c++ priority queue) in which each insert and delete-min operation takes O(logn)

Complexity : O(nlogn)

C++ Implementation

```cpp
#include <bits/stdc++.h>

using namespace std;

int main(){

        int n;
        cin>>n;
        priority_queue< int, vector<int>, greater<int> > min_heap;

        for(int i=0;i<n;i++){
                int k;
                cin>>k;
                min_heap.push(k);
        }

        long long ans;

        while(min_heap.size()>1){
                int x = min_heap.top(); min_heap.pop();
                int y = min_heap.top(); min_heap.pop();
                ans += x+y;
                min_heap.push(x+y);
        }

        cout<<ans<<endl;
}
```

Topic 2: Sorting

Session 7: Merge Sort

Counting Inversions

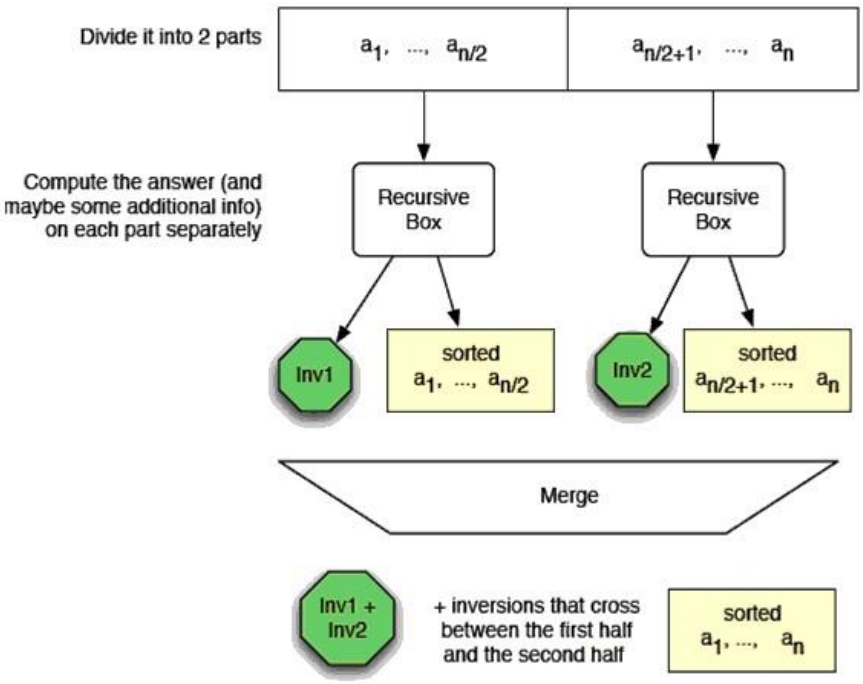
Two elements a[i] and a[j] form an inversion pair if a[i] > a[j] and i < j, you have to count the total number of inversion pairs possible in a given array.
For e.g. - The sequence 3, 7, 1, 5, 9 has three inversions (3, 1), (7, 1), (7, 5).

Naive approach :
Iterate over all pairs and check whether it forms an inversion pair or not. if yes, increment the inversion count.

```
inversionCount :
    int count = 0;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i+1; j < n; j++)
        {
            if (arr[i] > arr[j])
            count++;
        }
    }
```

Complexity : O(n^2)

Divide and Conquer approach (modified merge sort) :
DIVIDE: size of sequence n to two vectors of size n/2
CONQUER: count recursively inversion pairs in two vectors
COMBINE: this is a trick part (to do it in linear time)



We are going to modify mergesort algorithm in order to compute number of inversions. But first recall mergesort algorithm:

```
MergeSort(arr[], l,  r)
If r > l
     // Find the middle point to divide the array into two halves:
     middle m = (l+r)/2
     // Call mergeSort for first half:
     A = mergeSort(arr, l, m)
     // Call mergeSort for second half:
     B = mergeSort(arr, m+1, r)
     // Merge the two halves sorted in step 2 and 3:
     C = merge(A, B)
```

We are going to modify the merge step so that now instead of just producing the sorted output vector C from A, B but this time we are also going to count the number of inversions.

Following is the pseudocode for the merge step :

```
merge (A, B)
    a = b = count = 0
    C = empty vector
    while a < |A| and b < |B|: // not at end of a vector
        C.push_back(min(A[a], B[b]))
        if B[b] == min(A[a], B[b]):
            b = b + 1
        else
            a = a + 1
    Append the non-empty vector to C // the one whose elements are still left
    return C
```

We are going to modify the merge step so that now instead of just producing the sorted output vector C from A, B but this time we are also going to count the number of inversions.
Following is the pseudocode for the merge step:

```
merge (A, B)
    a = b = count = 0
    C = empty vector
    while a < |A| and b < |B|: // not at end of a vector
        C.push_back(min(A[a], B[b]))
        if B[b] == min(A[a], B[b]):
            b = b + 1
        else
            a = a + 1
    Append the non-empty vector to C // the one whose elements are still left
    return C
```

Every time A[a] is appended to the output, no new inversions are encountered, since A[a] is smaller than everything left in vector B.
If B[b] is appended to the output, then it is smaller than all the remaining items in A, we increase the number of count of inversions by the number of elements remaining in A.
Following is the pseudo code for modified merging and counting step:

```
merge-and-count(vector<int> A, vector<int>  B) ->
    a = b = count = 0
    C = empty vector
    while a < |A| and b < |B|: // not at end of a vector
        C.push_back(min(A[a], B[b]))
        if B[b] == min(A[a], B[b]):
            b = b + 1
            count += |A| - a //increment by number of elements left in A
        else
            a = a + 1
    Append the non-empty vector to C // the one whose elements are still left
    return count and C
```

Complexity : O(nlog(n)) !! proof similar as merge sort.

Topic 2: Sorting

Session 8: Quick Sort

Sorting based problem:

Find Median of an Array

Given an array of size n (a1, a2, ..., an), we wish to find its median element.

The median is the middle element in the sorted array. We define it to be as follows:

The median is an element of the array such that there are at least n/2 elements in the array that are greater than or equal to it, and at least n/2 elements in the array that are less than or equal to it (where we round down if n/2 is not an integer).

We wish to find any one of the medians for an array of even length.

For example:

1) 2 4 1 3 5

The sorted array is 1 2 3 4 5 and the median is 3.

2) 1 5 3 1 4 2

The sorted array is 1 1 2 3 4 5 and both 2 and 3 are medians.

A Basic Algorithm

A naive approach would be to first sort the array and then just compute the median as the middle element. Suppose a1, a2, ..., an is sorted then $a_{n/2}$ is definitely a median.

Sorting takes O(nlogn) and then finding the middle element takes O(1).

Complexity: O(nlogn)

A Better Algorithm

Here, we describe an algorithm that computes the kth greatest element of an array in **O(n)**. For median k = n/2. In fact we are able to solve this more general problem in O(n)!

The algorithm is based on divide and conquer and is very much similar to quick sort.

Suppose T(n) is the time taken to compute the kth largest element of an array of size n.

Step 1: For small n (say n < 6) we just sort the array and find the answer directly -- *O(1) since n<6.*

Step 2: For large n (n>=6), we divide the array into n/5 segments each of size 5 (ignoring the last segment which may have lesser elements) -- *O(n)*

Step 3: For each of these segments we we find the median using step 1 (with k = n/2) -- *O(1)*n/5 = O(n)*

Step 4: Now we have n/5 medians. We find the median of these elements recursively -- *T(n/5)* (with k = (n/5)/2).

Step 5: Now we consider the element obtained, say x (at position P in the image). As you can see in the image, we have many blocks, each containing 5 elements (say in a sorted order from up to down after step 1). Consider that finally the n/5 lists obtained are also sorted (from left to right) on the basis of their medians.

Then we have that all elements in the upper left rectangle are less than or equal to the element at P. Also all the elements in the lower right block must be greater than or equal to the element at P.

The size of each of these blocks is greater than or equal to 3n/4 as can be seen from the image (the total number of elements is n).

Step 6: Now we iterate over the initial array and find all the elements that are less than or equal to x and put them in a block L. We put the rest of the elements in block R. -- *O(n)*

Thus we have [L] x [R] as the array.

From step 5, it can be seen that the size of each L and R is greater than or equal to n/4. Also sum of sizes of L and R equals n-1. Thus each of the sizes must also be less than or equal to 3n/4.

Step 7: Let size of L be l and size of R be r. We make 3 cases:

Case 1: If l = n/2 - 1. Then x is the median element. -- O(1)
Case 2: If l > n/2 - 1. Then we recursively find the kth largest element in the left block. -- T(l)

Case 3: l < n/2 - 1. Then we recursively find the (k - n/2)th largest element in the right block -- T(r)
From step 6, we know T(l) <= T(3n/4) and T(r) <= T(3n/4).
Thus the time for this step is less than or equal to T(3n/4).
Also it can be seen that **we finally have the answer after this step**.

Complexity

Adding the complexity for each step, we get the following:
T(n) <= T(n/5) + T(3n/4) + O(n)
Solving this we can get T(n) <= O(20*n) = O(n)

Complexity: O(n)

Topic 3: Searching (Graph Based)

Session 9: Graph Traversal Algorithm (BFS)

Note

Till Topic 2: Sorting we have used function based notations in the algorithm. Henceforth, we will also use object based notations.

Example 1 (From Topic 1)
In quick sort algorithm, the length of sequence '**S**' is expressed as 'length(S)'. This was a functional way of representing. The object based notation for the same is 'S.length()'.

Example 2 (From Topic 2)
Session 9, BFS, Slide 4. Appending the vertex '**v**' to the new sequence 'S_0', is expressed as 'S_0.append(v)'. The same could have been expressed in functional notation as 'append(S_0,v)'.

Please note that we will mostly be using object based notations for the remaining sessions. We may use functional notations in certain cases where it is more clear to represent.

Topic 3: Searching (Graph Based)

Session 9: Graph Traversal Algorithm (BFS)

Bridges in a Graph

An edge in a graph is called a bridge or cut-edge iff removing that edge from the graph increases the number of connected components in the graph. Suppose G is a cyclic graph

As you can see, removing any edge from G does not create any new connected component. Thus we can equivalently say that an edge is a bridge if and only if it is not contained in any cycle.

Naive Approach

The simplest approach would be to remove an edge and then check that if removing that edge increases the number of components or not. Checking the number of connected components could be done by a simple DFS. If you don't know how to count no. of connected components in the graph, then it would be a good exercise to think about it.

Complexity of this approach would be : O(E*(V+E)) , O(V+E) for counting number of connected components and we have to do this after removing each edge thus O(E*(V+E)).

DFS based approach O(V+E)

Before getting into the algorithm make yourself comfortable with the following fact:

Let's say we are in the DFS, looking through the edges starting from vertex **u**. The current edge **(u, v)** is a bridge if and only if none of the vertices **v** and its descendants in the DFS traversal tree has a back-edge to vertex **u** or any of its ancestors. Indeed, this condition means that there is no other way from **u** to **v** except for edge **(u, v)**.

To check this efficiently we will maintain an array **time[v]** which will store the entry time of the vertex **v**. We introduce an array **fup** which will let us check the fact for each vertex **v**. **fup[v]** is the minimum of **time[v]**, the entry

times **time[p]** for each node **p** that is connected to node **v** via a back-edge **(v,p)** and the values of **fup[to]**for each vertex **to** which is a direct descendant of **v** in the DFS tree:

fup[v]=min{**time[v]**, **time[p]** for all **p** for which **(v,p)** is a back edge, **fup[to]** for all to for which **(v,to)** is a tree edge}

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex **v** has a child **to** for which **fup[to]≤time[v]**. If **fup[to]=time[v]**, the back edge comes directly to **v**, otherwise it comes to one of the ancestors of **v**.

Thus, the current edge **(v,to)** in the DFS tree is a bridge if and only if **fup[to]>time[v]**.

C++ Implementation

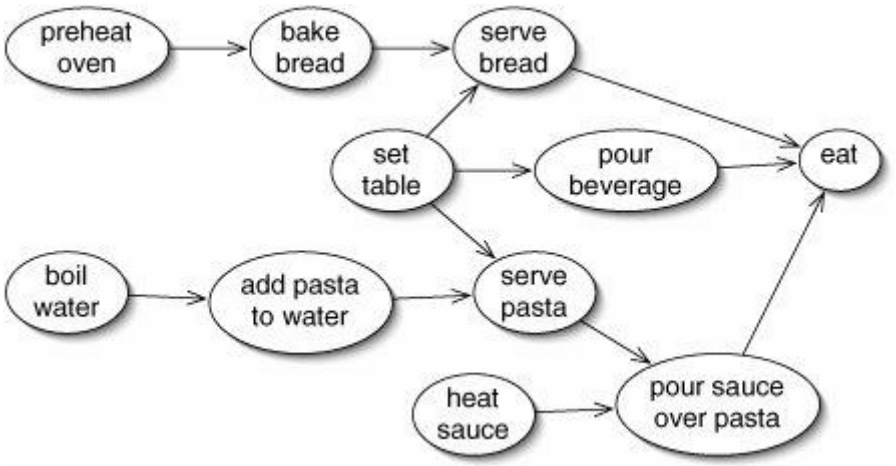
```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i = 0; i < g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p)  continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                cout << "edge (" << v << ", " << to << ") is a bridge\n";
        }
    }
}

void find_bridges() {
    timer = 0;
    for (int i = 0; i < n; ++i)
        used[i] = false;
    for (int i = 0; i < n; ++i)
        if (!used[i])
            dfs (i);
}
```

Topic 3: Searching (Graph Based)

Session 9: Graph Traversal Algorithm (BFS)

Topological Sort-ing Using BFS

In the words of wikipedia "a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering". To further understand this problem suppose you represent the typical recipe of your breakfast as a graph. The graph might look like this -



Now notice that you cannot serve bread without preheating oven or eat before serving the table. In topological sorting you are simply asked to provide a feasible order in which to do processes. For example, preheat oven - bake bread - set table - pour beverage - boil water - add pasta to water - serve pasta - heat sauce - pour sauce over pasta - eat is a feasible order to do things while

eat - bake bread - set table - pour beverage - boil water - add pasta to water - serve pasta - heat sauce - pour sauce over pasta - preheat oven is not.

Notice that the first node in the topological ordering must not have any incoming edge to it which leads the result that no topological ordering is possible when there exist a cycle in the graph since whichever node you would pick first in the cycle you would always have an incoming edge to it. Also, topological ordering is not unique, it could be easily be seen in the example provided above. *For example, set table - preheat oven - bake bread - pour beverage - boil water - add pasta to water - serve pasta - heat sauce - pour sauce over pasta - eat* is also a valid topological ordering.

Solution :

So the basic intuition behind this is to modify dfs algorithm in such a way that we visit every node only after all the nodes reachable from that node have been visited.

Note that this provides a topological ordering in reverse order so to get the correct topological ordering we will simply reverse the ordering got by this algorithm.

First briefly recall the dfs algorithm. In DFS, we start from a vertex, we first visit it and then recursively call DFS for its adjacent vertices. Here we want that all vertices reachable from a particular vertex must appear before that vertex thus we would modify dfs such that when we see a unvisited vertex we will first recursively call topological sorting to all its adjacent vertices so that we visit them before the vertex itself and then push the vertex into the order. In this way, a vertex is pushed to order only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the order. Remember, this will give you topological sorting in opposite order.

The above algorithm is simply DFS with an additional vector to store the ordering. So time complexity is same as DFS which is **O(V+E)**.

Note : This algorithm assumes there are no cycles inside the graph. A cycle in the graph implies that no topological ordering is possible. Also it provides one of the topological ordering.

C++ Implementation :

```
vector<int> adj[1000]; //adjacency list to represent the graph
vector<bool> visited(1000, false); //boolean vector to check whether this
vertex is visted or not
vector<int> order; //vector order to store the topological ordering

//function to construct the topological ordering takes starting vertex
void topsort(int u)
{
        //if this vertex is already visited then return
        if (visited[u])
        {
                return;
        }

        //else mark this vertex as visited
        visited[u] = true;
```

```
        //recursively call topological sorting to all its adjacent vertices
        for (int i = 0; i < adj[u].size(); i++)
        {
                int v = adj[u][i];
                topsort(v);
        }

        //push this vertex in order
        order.push_back(u);
}

int main()
{

        //create edges in the graph
        adj[5].push_back(2);
        adj[5].push_back(0);
        adj[4].push_back(0);
        adj[4].push_back(1);
        adj[2].push_back(3);
        adj[3].push_back(1);

        //call topological sorting
        for (int u = 0; u < 6; u++)
        {
                topsort(u);
        }

        //reverse the vector to get the actual ordering
        reverse(order.begin(), order.end());

        //print the ordering
        for(int i = 0; i < 6; ++i)
        {
                cout << order[i] << " ";
        }
}
```

Topic 3: Searching (Graph Based)

Session 9: Graph Traversal Algorithm (BFS)

Cycle In A Directed Graph

You have been given a directed graph. You need to check whether this graph contains a cycle or not. Note that 'a cycle is a path of edges and vertices where a vertex is reachable from itself'.

Input :
The first line contains an integer n (the number of vertices in the graph) and an integer m (the number of edges).

The next m lines contains contain two integers x and y each, denoting an edge from vertex x to vertex y.

Output :
Print "YES" if the graph contains a cycle, else print "NO".

Example Testcase 1 :
Input:

```
4 4
1 2
1 3
1 4
2 3
```

Output: NO

Example Testcase 2 :
Input:

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

Output: YES

Solution

This problem can be solved using a DFS. At each point in the DFS we maintain a list of nodes that are ancestors of the current node. Now if there is an edge from the current node to a node in the maintained list (called a back edge), there must be a cycle.

If there is no such edge, we don't have a cycle.

Complexity: O(n+m)

C++ Implementation

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<bool> visited, recStack;
vector<vector<int> > graph;


bool findCycle(int curr){
        for(int j=0;j<graph[curr].size();j++){
                int x = graph[curr][j];
                if(recStack[x]) return true;
                // Edge from node to ancestor found

                if(!visited[x]){
                        visited[x] = true;
                        recStack[x] = true;
                        if(findCycle(x)) return true;
                        recStack[x] = false;

                }
        }
        return false;
}


int main(){
        int n,m;
        cin>>n>>m;

        graph.resize(n);
        visited.resize(n,0);
        recStack.resize(n,0);
        // recStack maintains current ancestors

```

```
        for(int i=0;i<m;i++){
                int x,y;
                cin>>x>>y;
                x--;y--; // for 0-based indexing
                graph[x].push_back(y);
        }

        bool found = false;

        for(int i=0;i<n && !found;i++){// Perform DFS on graph
                if(!visited[i]){
                        visited[i] = true;
                        // Put current node in list of ancestors
                        recStack[i] = true;
                        // Recursive DFS
                        if(findCycle(i)) found = true;
                        // Remove current node in list of ancestors
                        recStack[i] = false;
                }
        }

        if(found) cout<<"YES"<<endl;
        else cout<<"NO"<<endl;
}
```

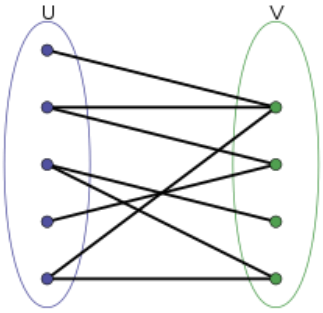
Topic 3: Searching (Graph Based)

Session 9: Graph Traversal Algorithm (BFS)

Checking If A Graph Is Bipartite

You have been given a graph (undirected). You need to check whether this graph is bipartite or not.

A bipartite graph is one in which the vertices can be partitioned into 2 sets (both non empty) such that all edges are from vertices of one set to the other set, i.e, if vertices a and b belong to the same set, there is no edge between them.



Input :
The first line contains an integer n (the number of vertices in the graph) and an integer m (the number of edges).

The next m lines contain two integers x and y each, denoting an edge between vertex x and vertex y.

Output :
Print "YES" if the graph is bipartite, else print "NO".

Example Testcase 1 :
Input: Output: YES

2 1
2 1

Example Testcase 2 :
Input: Output: NO

3 3
1 2
2 3
1 3

Solution

This problem can be easily solved using a depth-first search (or even a breadth first search).

Consider 2 colours (say red and blue). We will be colouring each vertex.

Call a vertex a source vertex and assign it a colour. Assign all its children the opposite colour and so on. If we encounter a vertex which has the same colour as its parent, the graph cannot be bipartite.

If no such vertex is found, the graph must be bipartite.

Complexity: O(n+m)

C++ Implementation

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){
	int n,m;
	cin>>n>>m;

	vector<bool> visited(n,0);
	// Initially no vertex is coloured
	vector<int> color(n,-1);
	vector<vector<int> > graph(n);

	for(int i=0;i<m;i++){
		int x,y;
		cin>>x>>y;
		x--;y--;	// for 0-based indexing
		graph[x].push_back(y);
		graph[y].push_back(x);
	}

	bool isBipartite = true;

	// Perform DFS using stack
	for(int i=0;i<n && isBipartite;i++){
		if(!visited[i]){
			stack<int> dfs;
			dfs.push(i);
			visited[i] = true;

			color[i] = 0;

```

```
                        while(!dfs.empty() && isBipartite){
                                int curr = dfs.top();
                                dfs.pop();

                                for(int j=0;j<graph[curr].size() &&
isBipartite;j++){
                                        int x = graph[curr][j];
                                        if(!visited[x]){
                                                visited[x] = true;
                                                color[x] = 1-color[curr];
                                                dfs.push(x);
                                        }
                                        else if(color[x] == color[curr])
isBipartite = false;
                                }
                        }
                }
        }

        cout<<(isBipartite?"YES":"NO")<<endl;
}
```