টপোলজিকাল সর্ট - smilitude

টপোলজিকাল সর্ট

কালিদাস খুব ভালো মানুষ। প্রতিদিন ঘুম থেকে উঠে দাঁত মাজার আগে একগ্লাস করে পানি খায়, আর একটা করে কবিতা লিখে। তারপর রোদ উঠলে খুব উদাস হয়ে গাছের পাতার ছায়ার দিকে তাকিয়ে থাকে আর কি কি জানি ভাবে। তারপর ওর বউ রুটি বানানোর বেলন দিয়ে একটা গুতা মারলে ওর মনে পড়ে ওর অফিসে যেতে হবে। তারপর ওর হেব্বি ঝামেলা লেগে যায়। মাঝে মাঝে ও মোজা পড়ার আগে জুতা পরে ফেলে, তারপর মোজা হাতে ফ্যাল ফ্যাল করে তাকিয়ে থাকে। আরো ঝামেলা লাগে যখন সে মোজা জুতা পরে তারপর খেয়াল করে ওর প্যান্ট পরা হয় নাই। সত্যিকারের গ্যান্জাম লাগে যখন সে সুপারম্যানের মত করে জামা কাপড় পরে।

শেষমেশ দেখা যায় ঠিকঠাকমত জামা কাপড় পরতে কালিদাসের কয়েক ঘন্টা লেগে গেছে, আর অফিসের বস খুব ক্ষেপে আছে ওর উপর - সব সময় দেরি করে আসে আর হিসেব করতে দিলে সেখানে গুটিগুটি করে কবিতা লিখে দেয়। কালিদাসের বাড়িতে একটা পেন্টিয়াম থ্রি কম্পিউটার আছে। ও নোটপ্যাডের সৌন্দর্য দেখে মুগ্ধ। কবিতা লিখতে ইদানিং নাকি আর কালি লাগে না! কালিদাস এখন একটা সফটওয়্যার কিনতে চায় যেটা ওকে হিসেব কষে দিবে কিসের পরে কি পরতে হবে।

টপোলজিকাল সর্ট

এ ধরণের প্রবলেমকে বলা হয় টপোলজিকাল সর্ট। যারা একটু পুরান ঘষা-খাওয়া প্রোগ্রামার, তারা আলসেমি করে টপ-সর্ট বলে। প্রবলেমটা হচ্ছে তোমাকে কিছু নিয়ম দেয়া আছে (যেমন, জুতার আগে মোজা, জুতার আগে প্যান্ট, জামার আগে গেন্জি) তোমাকে এমন একটা অর্ডার বের করতে হবে, যাতে এই ধরণের কোন নিয়ম ভঙ্গ না হয়। যেমন - মোজা, প্যান্ট, গেন্জি, জামা, জুতা । এই অর্ডারে জামা কাপড় পরলে কোন গ্যান্জাম লাগে না।

সহজ অ্যালগরিদম

টপোলজিকাল সর্ট করার একটা সহজ অ্যালগরিদম হচ্ছে, আমাদের যদি n টা আইটেম থাকে - আমরা একটা n পর্যন্ত লুপ চালাবো, তারপর প্রতিবার এমন একটা আইটেমকে খুঁজে বের করবো - যার আগে যাদের নেয়ার কথা সবাইকে নেয়া হয়ে গেছে। তারপর ওই আইটেমটা নিয়ে নিবো, আর বাকিদের মধ্যে আবার একটা আইটেম বের করবো। যদি কোন স্টেপে নেয়ার মতো কাওকে না পাওয়া যায়, তার মানে এখানে কোনভাবেই সর্টেড অর্ডার পাওয়া যাবে না।

```
int taken[55] =
{};
```

```
int n, take[55][55], list[55]
indegree[55];
int i, j,
k;

// when take[a][b] = 1, that means a must come before
b
// indegree[i] = number of items that that must come before
i
// when taken[i] = 1, means we already have taken ith
item
int invalid =
0;
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++) if( !indegree[j] && !taken[j]     )
{
        taken[j] =
1;
        list[i] =
j;
        // in this step we are taking item
j
        // we'd update the indegree[k] of items that depended on
j
        for(k=0; k<n;
k++)
            if( !taken[k] && take[j][k] ) --
indegree[k];

break;

}
    if( j == n )
{
        invalid =
1;

break;

}
}

if( invalid ) printf("There is no
solution\n");
else for(i=0; i<n; i++) printf("%d\n", list[i]
);
```

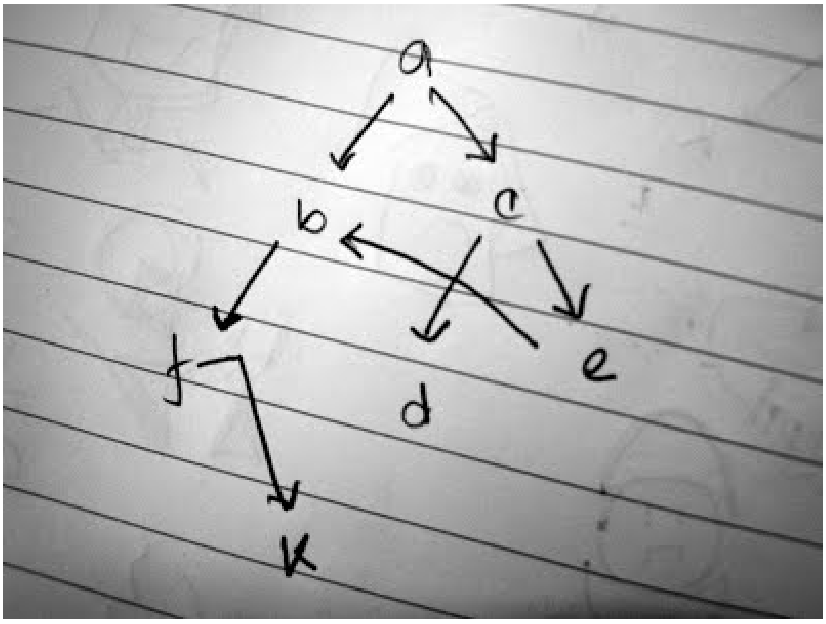
এই অ্যালগরিদমটার রানিং টাইম হচ্ছে O(n^2)। টপোলজিকাল সর্ট এর চে অনেক দ্রুত বের করা যায়। কিন্তু এই অ্যাপরোচটার সুবিধা হচ্ছে, এই অ্যালগরিদমটা লেখা বেশ সহজ। ছোট সাইজের n এর জন্য এটা খুব দ্রুত লিখে ফেলা যায় কন্টেস্টের সময়। আরো একটা সুবিধা হচ্ছে মাঝে মাঝে কন্টেস্টের প্রবলেমগুলোতে উল্টা পাল্টা কিছু কন্ডিশন জুড়ে দেয়। যেমন যদি বেশ কয়েকটা ঠিক অর্ডারিং থাকে, তাহলে আমাকে লেক্সিকোগ্রাফিকালি সবচে' আগের অর্ডারিংটা প্রিন্ট করতে হবে। এরকম কিছু থাকলে সেটা এখানে হ্যান্ডেল করা বেশ সহজ। আমি নেয়ার সময় শুধু দেখবো কাদের কাদেরকে নিতে পারি, আর তাদের মধ্যে সবার আগে কাকে নেয়া উচিত আমার। আর আরেকটা ভালো ব্যাপার হচ্ছে এটা বোঝার জন্য কোন অ্যাডভান্সড অ্যালগরিদম লাগে না, যে কাউকে বুঝিয়ে দেয়া যায় আমি আসলে কি করছি।

ডেপ্থ ফার্স্ট সার্চ

এই সলুশনটা বোঝার জন্য তোমাকে খুব অল্পকিছু STL আর অল্পকিছু রিকার্শন বুঝতে হবে। আর অল্প অল্প গ্রাফ থিওরি জানলে ভালো।

যদি আমাকে b আইটেমটাকে a আইটেমের আগে নিতে হয়, তার মানে হচ্ছে a আসলে b এর উপর নির্ভর করছে। আমরা b কে নেয়ার

আগে a কে নিতে পারবো না। আমরা যদি এখন a -> b এরকম একটা করে তীর চিহ্ন দেই, কে কার উপর নির্ভর করছে তার উপর হিসেব করে। তাহলে জিনিসটা এরকম একটা জিনিস দাড়াবে।



এই জিনিসটাকে আমরা বলি ডিরেক্টেড অ্যাসাইক্লিক গ্রাফ (DAG - Directed Acyclic Graph)। অ্যাসাইক্লিক মানে এখানে কোন সাইকেল (লুপ) নেই। আমাদের আগের অ্যালগরিদম এ কোন স্টেপে যদি আমরা একটাও আইটেম না নিতে পারি, তাহলে আমরা ধরে নেই যে এটা কোন সলুশন নেই। সেরকম কোন গ্রাফে যদি সাইকেল থাকে তাহলে সেটার কোন টপোলজিকাল অর্ডার থাকবে না। কেন? ধরো, আমাদের দুইটা নিয়ম আছে -

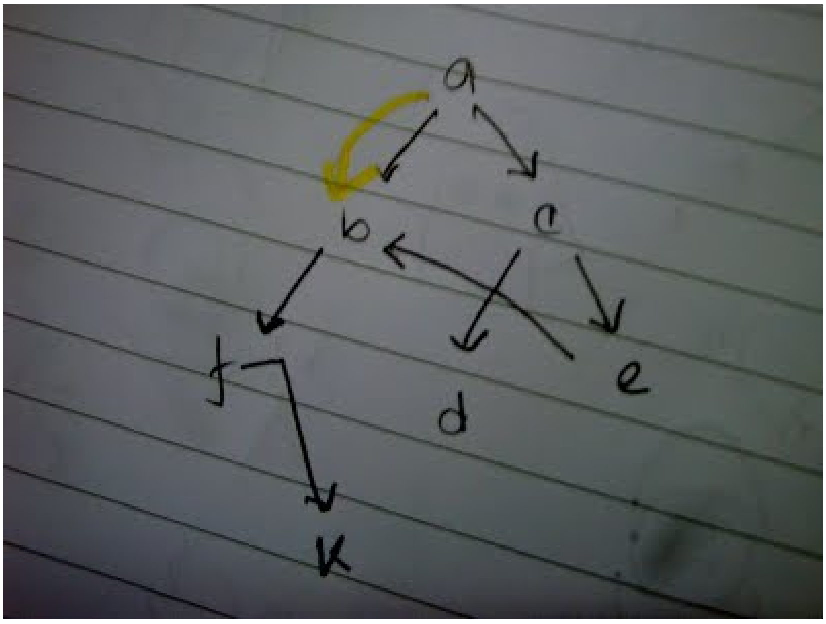
তো এখানে একটা লুপ তৈরী হয়ে গেছে, আমি যদি আগে মুরগী নেই, তাহলে দ্বিতীয় নিয়ম টা ভাঙা হবে - আর আগে ডিম নিলে প্রথম নিয়ম টা ভাঙা হবে - কোন ভাবেই একটা অর্ডারিং বের করা সম্ভব না, যার জন্য কোন নিয়ম ভাঙা হবে না। গ্রাফে যখন কোন সাইকেল থাকে, তখন এই সমস্যাটা হয়।

ডেপ্থ ফার্স্ট সার্চ দিয়ে কোন DAG এ আমরা যখন অর্ডারিং বের করি, তখন আমরা করি কি, কোন একটা আইটেমকে তখনই নেই যখন ওর আগে যাদেরকে নেয়ার কথা - সবাইকে নেয়া হয়ে যায়। তো অ্যালগরিদমটা হচ্ছে এরকম।

```
#define M
55
vector<int> ans,
depends[M];
int
taken[M];

// depends[i] contains all the items that item i is depending
on
// when taken[i] = 1, that means it's already
taken

void take( int p )
{
    if( !taken[ p ] )
{
        // i am taking all the items that i should take before
p
        for(int i=0; i<depends[p].size();
i++)
            take( depends[p][i]
);
        ans.push_back( p ); // now i can take
it
        taken[p] =
1;
```

```
}
}

int main()
{
    // input
routine
    //
.....

    for(int i=0; i<n; i++) take( i
);
    for(int i=0; i<n; i++) printf("%d\n", ans[i]
);

    return
0;
}
```

যদি খুব ভয় লাগে, তাহলে নিচের ছবিগুলোর দিকে তাকাও! :) ভয় খুব পঁচা জিনিস! ;)



আমরা প্রথমে a থেকে যাদেরকে a এর আগে নেয়ার কথা তাদেরকে নেবো। সেজন্য প্রথমে আমরা b কে নেয়ার জন্য কল করবো।

একই কারণে আমরা b থেকে f এ যাবো, কারণ f কে আমাদের b এর আগে নেয়ার কথা।

এরপর আমরা k তে যাবো, যেহেতু k কারো উপর নির্ভর করছে না, সেজন্য আমরা k কে লিস্টে ঢুকাবো। তারপর f এ ফিরে যাবো।

f শুধু k এর উপর নির্ভর করতো। k নেয়া শেষ, সুতরাং আমরা এখন f কে নিতে পারি। এরপর আমরা b তে ফিরে যাবো।

এবার আমরা b কে নিতে পারি, আমাদের f কে নেয়া শেষ। এরপর আমরা a তে ফিরে যাবো।

কিন্তু a কে নেয়ার আগে আমাদের c কেও নিতে হবে, কিন্তু c কে এখনো নেয়া হয় নাই। সুতরাং আমরা এখন c কে নিতে যাবো।

আবার একই ঝামেলা, c এর আগে যাদের নেবার কথা তাদের আগে নিতে হবে। তাদের নেয়ার আগে আমি c কে নিতে পারবো না।

d কে নিলাম! :)

কাজ শেষ হয় নাই। e কে নিতে হবে।

e কে নিতে এলুম দাদা।

এবার c কে নেয়া যায়।

অবশেষে কাম শেষ! :)

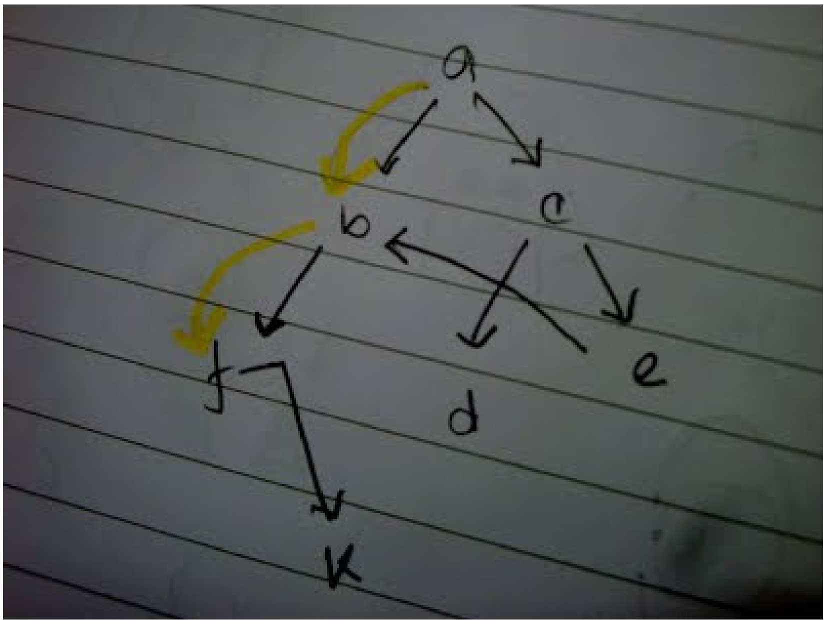
আরিকটু বাকি!

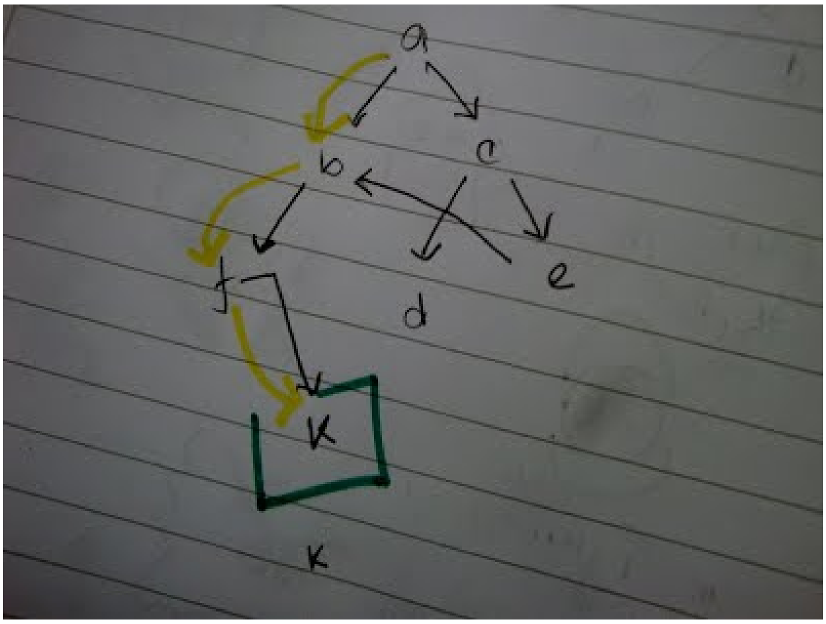
আমরা একটু আগে যেই গ্রাফটা নিয়ে কাজ করলাম, সেটা ছিলো একটা কানেক্টেড গ্রাফ। এমন ও হতে পারে যে গ্রাফটা কানেক্টেড নাও হতে পারে। সেক্ষেত্রে একই অ্যালগরিদম ধুমাধুম কাজ করবে - কোন ঝামেলা হবে না। এখন একটাই ঝামেলা বাকি, সেটা হচ্ছে, এমনও হতে পারে যে গ্রাফটাতে সাইকেল আছে। মানে ডিম আগে না মুরগী আগে, এই টাইপের একটা ক্যাঁচাল আছে কোথাও। সে ক্ষেত্রে আমাদের চেক

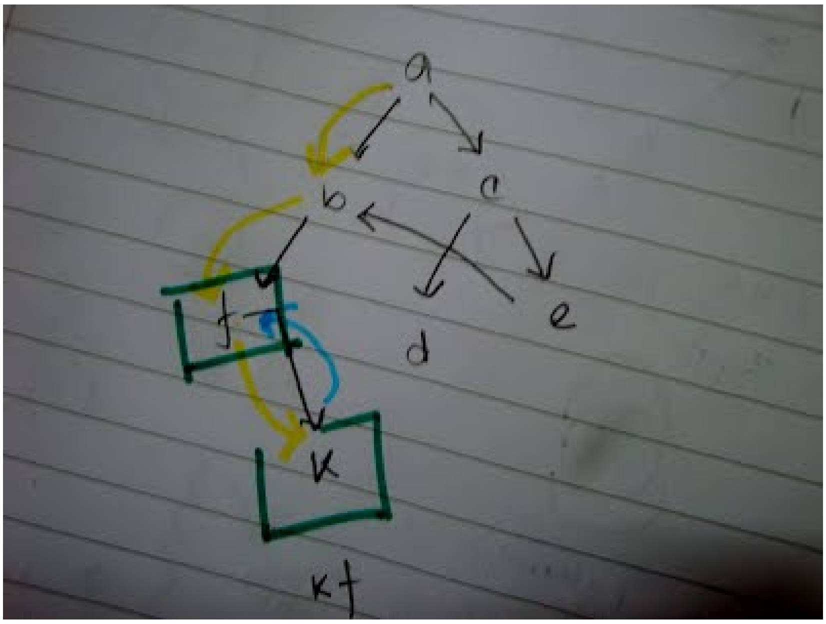
করে নিতে হবে গ্রাফটাতে সাইকেল আছে কিনা। গ্রাফে সাইকেল আছে কিনা, তার একটা সহজ পদ্ধতি হচ্ছে ফ্ল্যাগ রেখে একটা DFS চালিয়ে দেয়া। মানে ধরো সব নোডগুলো, যাদের এখনো ভিজিট করা হয় নাই, তাদের সবার ফ্ল্যাগ প্রথমে শূণ্য করে দিলাম। তারপর যখন ঢুকলাম তখন প্রসেস করার আগে এক করে দিলাম। প্রসেস শেষে দুই করে দিলাম। এখন আমরা যদি কোন নোডে ঢুকে দেখি সেটার স্ট্যাটাস দুই - তার মানে সেটা প্রসেস করা শেষ - সেটা আরেকবার প্রসেস করার দরকার নেই - স্ট্যাটাস শূণ্য মানে ওটা এখনো প্রসেস করাই হয় নাই। কিন্তু যদি আমরা ফ্লাগ এক পাই - এটার মানে হচ্ছে, এটাকে এখনো প্রসেসিং এখনো শেষ হয়নি, তার মানে যেই নোডটা থেকে আমি এই মাত্র ঢুকলাম এই নোডটায়, সেই নোডটাতে এই নোডটা হয়েই যাওয়া হয়েছে। তার মানে আমরা এই মাত্র একটা সাইকেল পেলাম।

কোডটা হবে এরকম।

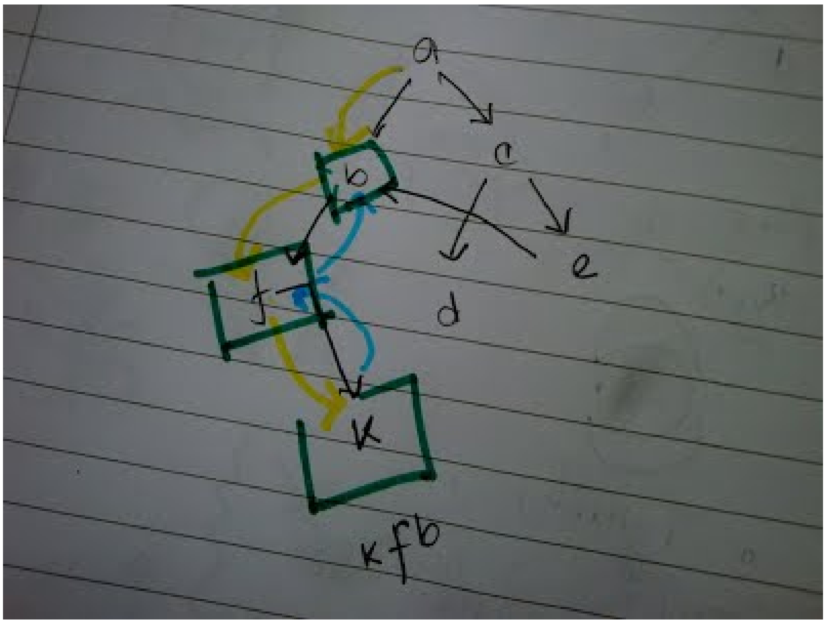
```
#define M 55
int visited[M], cycle = 0;
vector<int> edge[M];

void visit( int p )
{
```

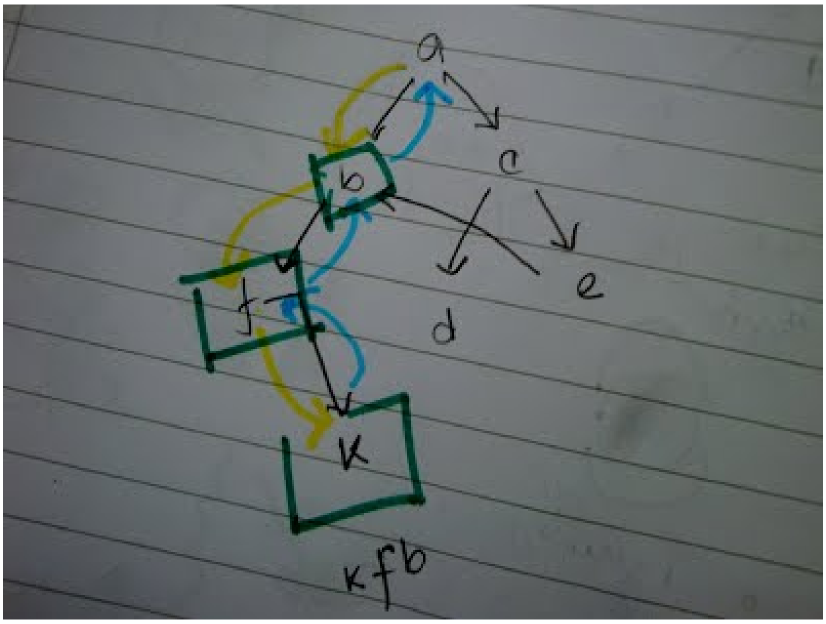


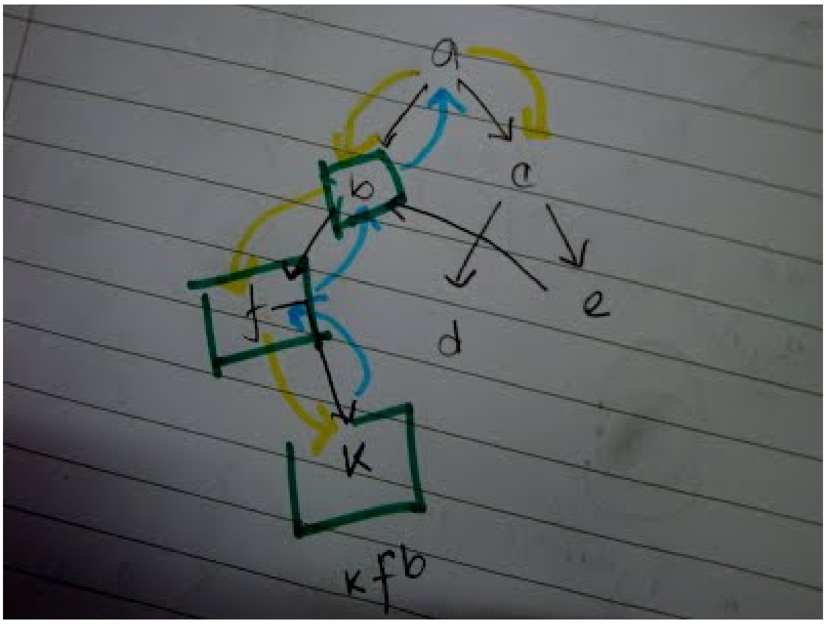




```
    if( visited[p] == 2 || cycle ) return;
```

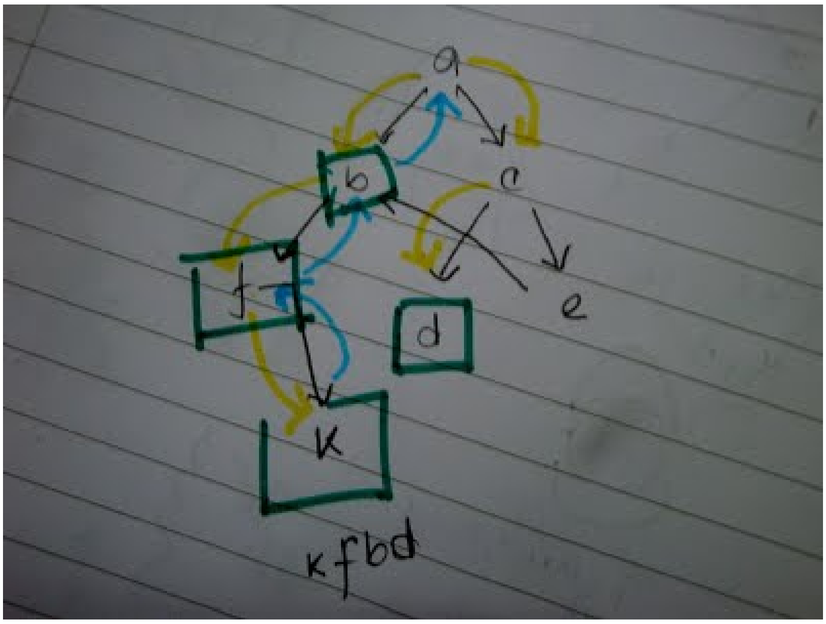


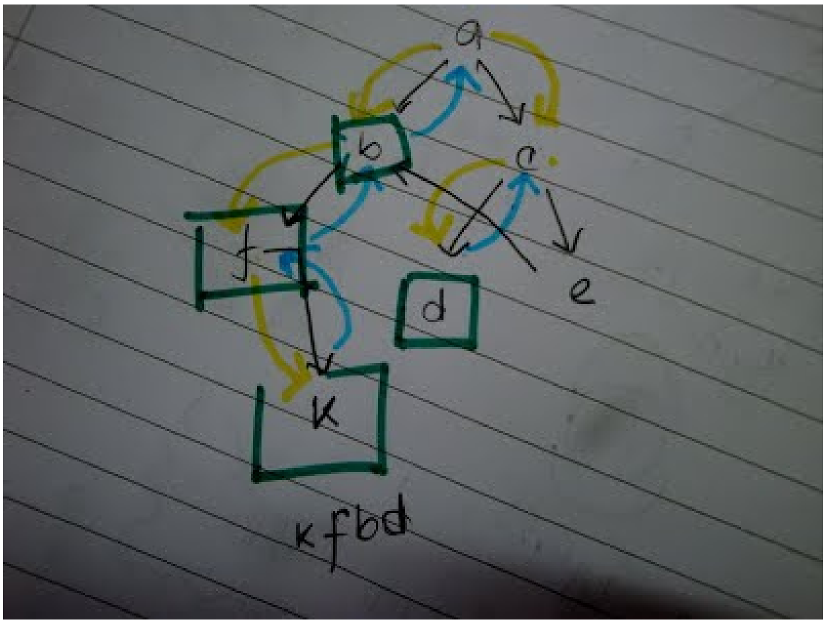


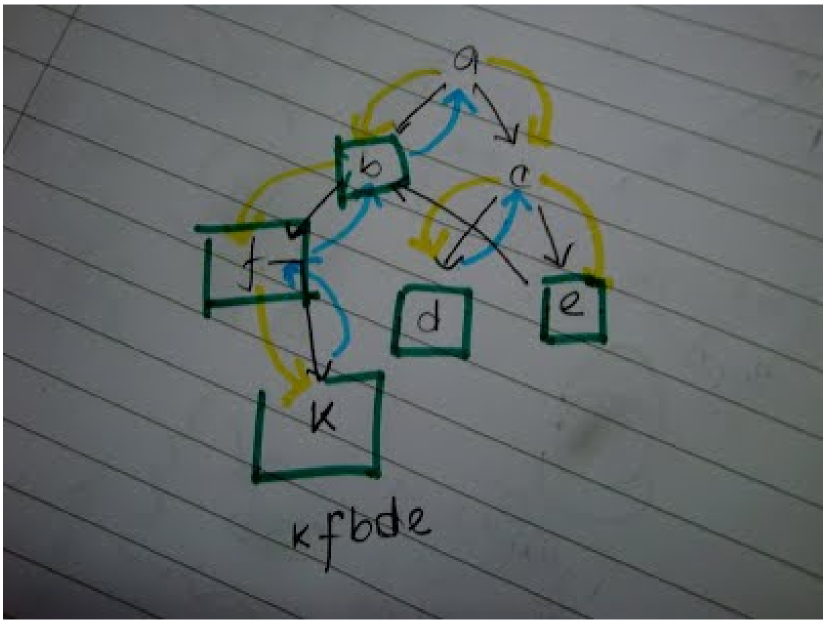


```
    // we won't process anything if we already have found a
cycle
```

```
    if( visited[p] == 1 )
{
```

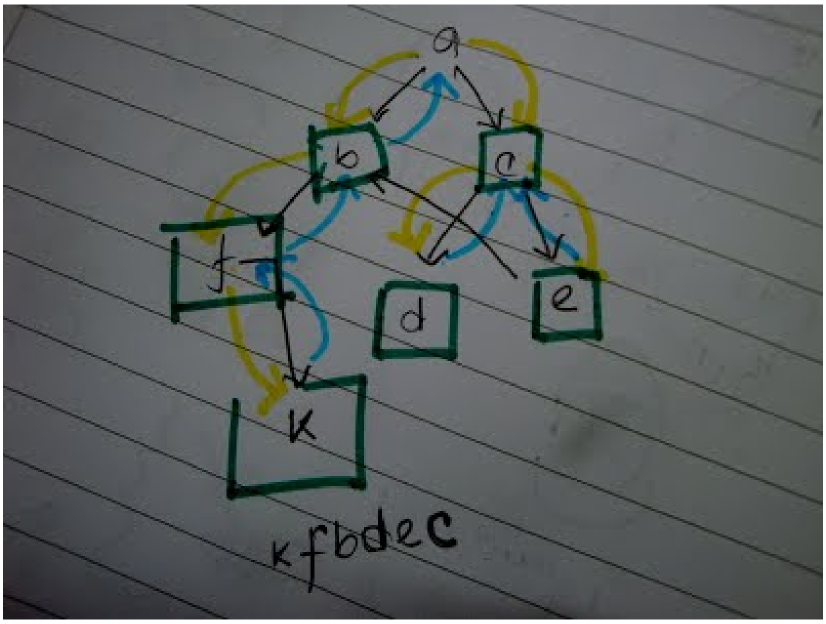


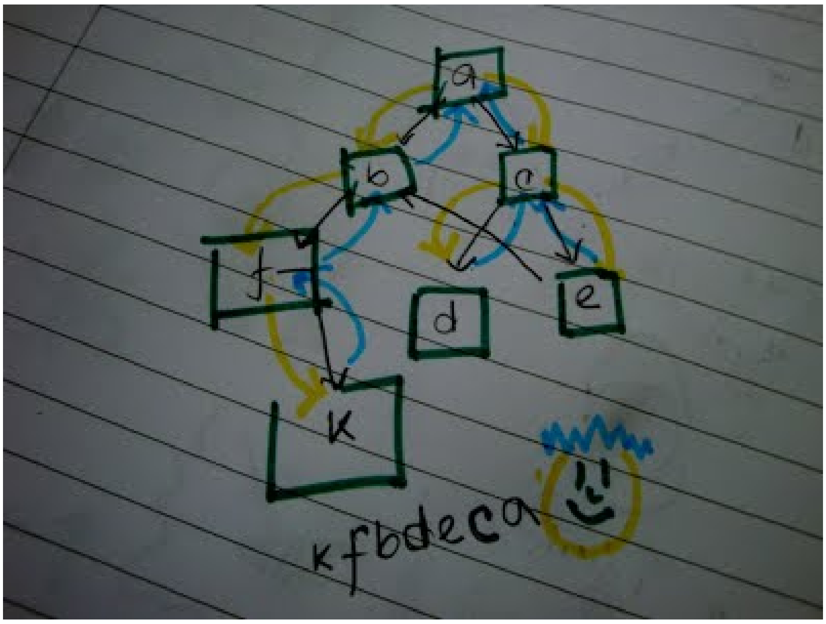




```
        cycle = 1; // found
cycle
```

```
return;

}

    visited[p] =
1;
```





```
    for(int i=0; i<edge[p].size(); i++) visit( edge[p][i]
);
    visited[p] =
2;
}
```

আমরা অনেক কিছু শিখে ফেললাম। :) এবার তুমি নিচের প্রবলেমগুলো নিয়ে গুতোগুতি করে দেখতে পারো।

শুভ কামনা! :)
অনেক ধন্যবাদ পড়ার জন্য।