

গ্রাফ থিওরি: স্টেবল ম্যারেজ প্রবলেম

বেশ কিছুদিন ডিপি নিয়ে লেখার পর আবার গ্রাফ থিওরিতে ফিরে এলাম। আজকে আমরা একটা সহজ কিন্তু ইন্টারেস্টিং প্রবলেম দেখবো। স্টেবল ম্যারেজ(Stable Marriage) প্রবলেম এক ধরনের বাইপারটাইট ম্যাচিং প্রবলেম, তবে এটা শেখার জন্য অন্য কোনো অ্যালগোরিদম জানার প্রয়োজন নেই।

মনে করি n টা ছেলে আর n টা মেয়ে আছে। এখন তাদের মধ্যে বিয়ে দিতে হবে এমন ভাবে যেমন বিয়ে “স্টেবল” হয়। প্রত্যেকের সাথেই প্রত্যেকের বিয়ে দেয়া সম্ভব তবে প্রতিটা ছেলে আর মেয়ের কিছু পছন্দ আছে, প্রত্যেকেই চাইবে তার পছন্দের মানুষকে বিয়ে করতে। যদি ছেলে ৩জনের নাম Tom, Bob, Peter, আর মেয়ে ৩জনের নাম Alice, Mary, Lucy হয় তাহলে ছেলেদের পছন্দের তালিকা হতে পারে এরকম:

আবার মেয়েদের পছন্দের তালিকাটা হতে পারে এরকম:

এখন কিভাবে বিয়ে দিলে বিয়ে স্টেবল হবে? আগে বুঝা দরকার স্টেবল বলতে কি বুঝাচ্ছি। ধরো নিচের মতো করে বিয়ে দেয়া হলো:

Tom -> Alice, Mary, Lucy
Bob -> Lucy, Mary, Alice
Peter -> Mary, Alice, Lucy

Shafaetsplanet.com

এই ম্যাচিং/বিয়েটা স্টেবল না, কারণ টম লুসির থেকে মেরিকে বেশি পছন্দ করে, আবার মেরি পিটারের থেকে টমকে বেশি পছন্দ করে। তাই টম আর মেরি বিয়ে ভেঙে একসাথে চলে আসতে পারে। যদি A, B ছেলে আর C, D মেয়ে হয় আর A-C এবং B-D কে বিয়ে দেয়া হয় তাহলে বিয়ে স্টেবল হবেনা যদি নিচের দুটি স্টেটমেন্টই সত্য হয়:

Alice-> Bob, Tom, Peter
Mary-> Tom, Bob, Peter
Lucy-> Peter, Bob, Tom

১. A যদি C এর থেকে D কে বেশি পছন্দ করে।
২. D যদি B এর থেকে A কে বেশি পছন্দ করে।

Shafaetsplanet.com

২টি স্টেটমেন্ট সত্য হলে A আর D বিয়ে ভেঙে চলে আসবে! তবে যেকোনো একটা স্টেটমেন্ট মিথ্যা হলে বিয়ে স্টেবল হবে।

Tom -> Lucy
Bob -> Alice
Peter -> Mary

Not stable!

১৯৬২ সালে David Gale আর Lloyd Shapley প্রমাণ করে সমান

সংখ্যক ছেলে আর মেয়ের জন্য সমসময় স্টেবল ম্যারেজ প্রবলেমের একটি

সমাধান আছে। তারা খুব সহজ একটা অ্যালগোরিদম আবিষ্কার করেন সমস্যাটি সমাধানে জন্য। অ্যালগোরিদমটি এরকম:

Shafaetsplanet.com

১. প্রথমে প্রতিটি অবিবাহিত ছেলে তার সবথেকে পছন্দের মেয়েটাকে প্রস্তাব পাঠাবে যাকে সে এখনো প্রস্তাব পাঠায়নি, মেয়েটি অলরেডি এনগেজড হলেও সমস্যা নাই, একটি মেয়েকে একাধিক ছেলে প্রস্তাব পাঠাতে পারে। একটি ছেলে কখনো একটি মেয়েকে দুইবার প্রস্তাব পাঠাবেনা।
২. এবার প্রতিটা মেয়ে তাকে যারা প্রস্তাব পাঠিয়েছে তাদের মধ্যে থেকে যাকে সবথেকে পছন্দ তাকে নির্বাচিত করবে, বাকি সবাইকে বাতিল করে দিবে। মেয়েটি আগেই কাওকে পছন্দ করে থাকলে তাকেও বাতিল করে দিবে।
৩. এখনো কেও অবিবাহিত থাকলে ১ম ধাপের পুনরাবৃত্তি হবে।

অ্যালগোরিদমটি কেনো কাজ করে? ধরি A-C এবং B-D এর বিয়ে দেয়া হয়েছে। তাহলে বিয়ে ভাঙবে A যদি D কে বেশি পছন্দ করে এবং D যদি A কে বেশি পছন্দ করে। কিন্তু উপরের অ্যালগোরিমে সেটা সম্ভবনা। কারণ:

A যদি D কে বেশি পছন্দ করে তাহলে সে D কে আগে প্রস্তাব পাঠাবে, D রাজি না হলে বা ছেড়ে দিলেই একমাত্র C কে প্রস্তাব পাঠাবে।

D যদি A কে বেশি পছন্দ করে তাহলে সে অন্য যে কাওকে ছেড়ে দিয়ে A কে বিয়ে করবে।

আর D যদি A কে বিয়ে না করে অন্য কাওকে করে তারমানে সে অন্য কাওকেই বেশি পছন্দ করে, এক্ষেত্রে বিয়ে ভাঙার সম্ভাবনা

এই অ্যালগোরিথমটা স্টেবল ম্যাচিং দিবে ঠিকই তবে অপটিমাল রেজাল্ট নাও দিতে পারে। প্রতিটি ছেলের জন্য রেজাল্ট অপটিমাল হবে, কিন্তু মেয়েদের জন্য অপটিমাল নাও হতে পারে, অর্থাৎ এমন স্টেবল ম্যাচিং থাকতে পারে যেটাও কোনো একটি মেয়ে আরো পছন্দের কাণ্ডকে বিয়ে করতে পারতো। অর্থাৎ যে প্রস্তাব পাঠাবে তার জন্য রেজাল্ট অপটিমাল হবে।

অ্যালগোরিদমটি কোডে ইমপ্লিমেন্ট করা খুব সহজ। **preference** লিস্ট তোমাকে ইনপুট দেয়া থাকবে। কে কাকে প্রস্তাব পাঠিয়েছে, কে কার সাথে এখন এনগেজড এই তথ্যগুলো অ্যাারেতে রেখে সহজেই কোডটা লিখে ফেলতে পারবে। [wikipedia](https://en.wikipedia.org/wiki/Stable_marriage_problem) তে দেয়া সুডোকোডটা এরকম:

Python

```

1  function stableMatching {
2      Initialize all  $m \in M$  and  $w \in W$  to free
3      while  $\exists$  free man  $m$  who still has a woman  $w$  to propose to {
4           $w = m$ 's highest ranked such woman to whom he has not yet proposed
5          if  $w$  is free
6              ( $m, w$ ) become engaged
7          else some pair ( $m', w$ ) already exists
8              if  $w$  prefers  $m$  to  $m'$ 
9                  ( $m, w$ ) become engaged
10                  $m'$  becomes free
11             else
12                 ( $m', w$ ) remain engaged
13     }
14 }
```

প্রবলেম:

[Light OJ: Employment](#)

[Codechef: Stable Marriage](#)

[Uva: Chemical Attraction](#)

[Codechef: Blocking](#)

গ্রাফ থিওরি নিয়ে অন্যান্য লেখা

গ্রাফ থিওরিতে হাতেখড়ি – ১

তুমি কি জানো পৃথিবীর প্রায় সব রকমের প্রবলেমকে কিছু রাস্তা আর শহরের প্রবলেম বানিয়ে সলভ করে ফেলা যায়? গ্রাফ থিওরির যখন জন্ম হয় তখন তোমার আমার কারোই জন্ম হয়নি, এমনকি পৃথিবীতে কোন কম্পিউটারও ছিলোনা! সেই সতেরো দশকের শেষের দিকে লিওনার্দ অয়লার গ্রাফ থিওরি আবিষ্কার করেন কনিসবার্গের সাতটি ব্রিজের সমস্যার সমাধান করতে। তখনই সবার কাছে পরিষ্কার হয়ে যায় যে যেকোন প্রবলেমকে শহর-রাস্তার প্রবলেম দিয়ে মডেলিং করে চেনা একটা প্রবলেম বানিয়ে ফেলতে গ্রাফ থিওরির জুড়ি নেই। আর যখনই তুমি একটা প্রবলেমকে চেনা কোন প্রবলেমে কনভার্ট করে ফেলতে পারবে তখন সেটা সমাধান করা অনেক সহজ হয়ে যাবে।

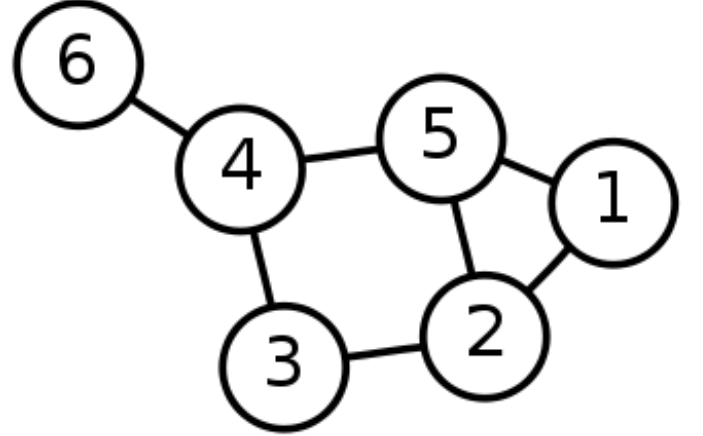
গ্রাফ থিওরির অনেক অ্যাপ্লিকেশন আছে। সবথেকে কমন হলো এক শহর থেকে আরেক শহরে যাবার দ্রুততম পথ খুঁজে বের করা। তুমি হয়তো জানো সার্ভার থেকে একটা ওয়েবপেজ তোমার পিসিতে পৌঁছাতে অনেকগুলো রাউটার পার করে আসতে হয়, গ্রাফ থিওরি দিয়ে এক রাউটার থেকে আরেকটাতে যাবার পথ খুঁজে বের করা হয়। যুদ্ধের সময় একটা দেশের কোন কোন রাস্তা বোমা দিয়ে উড়িয়ে দিলে দেশের রাজধানী সব শহর থেকে বিচ্ছিন্ন হয়ে যাবে সেটাও বের করে ফেলা যায় গ্রাফ থিওরি দিয়ে। আমরা গ্রাফ অ্যালগোরিদমগুলো শেখার সময় আরো অনেক অ্যাপ্লিকেশন দেখবো।

আমাদের এই হাতেখড়ির লক্ষ্য হবে প্রথমেই গ্রাফ থিওরির একদম বেসিক কিছু সংজ্ঞা জানা, তারপর কিভাবে গ্রাফ মেমরিতে স্টোর করতে হয় সেটা জানা, এরপরে গ্রাফের কিছু বেসিক অ্যালগোরিদম জানা এবং সাথে সাথে মজার কিছু প্রবলেম দেখা। সব সংজ্ঞা একসাথে দেখলে মাথায় থাকবেনা, তাই একদম না জানলেই নয় সেরকম কিছু সংজ্ঞা প্রথমে দেখবো, এরপর প্রয়োজনমতো আরো কিছু সংজ্ঞা জেনে নিবো।

গ্রাফ কি?

ধরা যাক ৬টি শহরকে আমরা ১,২,৩,৪,৫,৬ দিয়ে চিহ্নিত করলাম। এবার যে শহর থেকে যে শহরে সরাসরি রাস্তা আছে তাদের মধ্যে লাইন টেনে দিলাম:

শহরগুলোর নাম ১,২ ইত্যাদি দিয়ে দিতে হবে এমন কোন কথা নেই, তুমি চাইলে ঢাকা, চট্টগ্রাম ইত্যাদি দিতে পারো। এটা খুবই সাধারণ একটা গ্রাফ যেখানে কিছু শহরের মধ্যের রাস্তাগুলো দেখানো হয়েছে। গ্রাফ থিওরির ভাষায় শহরগুলোকে বলা হয় **নোড(Node)** বা **ভারটেক্স(Vertex)** আর রাস্তাগুলোকে বলা হয় **এজ(Edge)**। গ্রাফ হলো কিছু নোড আর কিছু এজ এর একটা কালেকশন।



গ্রাফে নোড দিয়ে অনেককিছু বুঝাতে পারে, কোন গ্রাফে হয়তো নোড দিয়ে শহর বুঝায়, কোন গ্রাফে এয়ারপোর্ট, কোন গ্রাফে আবার হয়তো দাবার বোর্ডের একটা ঘর বুঝাতে পারে! আর এজ দিয়ে বুঝায় নোডগুলোর মধ্যের সম্পর্ক। কোন গ্রাফে এজ দিয়ে দুটি শহরের দূরত্ব বুঝাতে পারে, কোন গ্রাফে এক এয়ারপোর্ট থেকে আরেক এয়ারপোর্টে যাবার সময় বুঝাতে পারে, আবার দাবার বোর্ডে একটা ঘরে ঘোড়া থাকলে সেই ঘর থেকে কোন ঘরে যাওয়া যায় সেটাও বুঝাতে পারে।

নিচের ছবিতে দাবার বোর্ডটাও একটা গ্রাফ। প্রতিটা ঘর একটা করে নোড। যে ঘরে ঘোড়া আছে সেখান থেকে এজগুলো দেখানো হয়েছে:

এককথায় নোডের কাজ কোন একধরনের অবজেক্টকে রিপ্রেজেন্ট করা আর এজ এর কাজ হলো দুটি অবজেক্টের মধ্যে সম্পর্কটা দেখানো।

অ্যাডজেসেন্ট নোড:

A নোড থেকে B নোডে একটা এজ থাকলে B কে বলা হয় A এর অ্যাডজেসেন্ট নোড। সোজা কথায় অ্যাডজেসেন্ট নোড হলো এজ দিয়ে সরাসরি কানেক্টেড নোড। একটা নোডের অনেকগুলো অ্যাডজেসেন্ট নোড থাকতে পারে।

ডিরেক্টেড গ্রাফ আর আনডিরেক্টেড গ্রাফ:

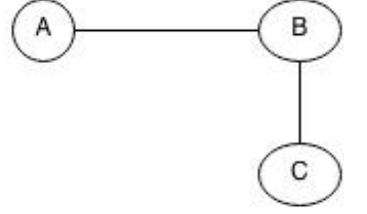
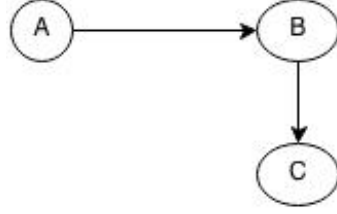
ডিরেক্টেড গ্রাফে এজগুলোতে তীরচিহ্ন থাকে, তারমানে এজগুলো হলো একমুখী(Unidirectional), আনডিরেক্টেড গ্রাফে এজগুলো দ্বিমুখী(Bidirectional)। নিচের ছবি দেখলেই পরিষ্কার হবে:

বামের ছবির গ্রাফ ডিরেক্টেড, ডানেরটা আনডিরেক্টেড।

ওয়েটেড আর আনওয়েটেড গ্রাফ:

অনেক সময় গ্রাফে এজগুলোর পাশে ছোট করে ওয়েট(Weight) বা কস্ট(Cost) লেখা থাকতে পারে:

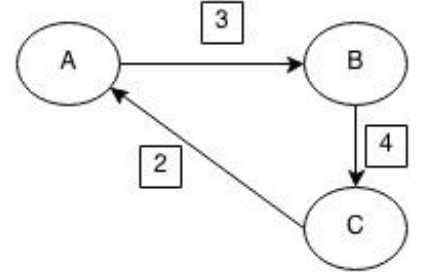
এই ওয়েট বা কস্ট দিয়ে অনেককিছু বুঝাতে পারে, যেমন দুটি শহরের দূরত্ব কত কিলোমিটার, অথবা রাস্তাটি দিয়ে যেতে কত সময় লাগে, অথবা রাস্তা দিয়ে কয়টা গাড়ি একসাথে যেতে পারে ইত্যাদি। আগের গ্রাফগুলো ছিলো আনওয়েটেড, সেক্ষেত্রে আমরা ধরে নেই সবগুলো এজের ওয়েটের মান এক(১)। সবগুলো ওয়েট ১ হলে আলাদা করে লেখা দরকার হয়না।



পাথ:

পাথ(Path) হলো যে এজগুলো ধরে একটা নোড থেকে আরেকটা নোডে যাওয়া যায়। অর্থাৎ এজের একটা সিকোয়েন্স।

এক নোড থেকে আরেক নোডে যাবার অনেকগুলো পাথ থাকতে পারে। ছবিতে A থেকে D তে যাবার দুইটা পাথ আছে। A->B,B->C,C->D হলো একটা পাথ, এই পাথের মোট ওয়েট হলো ৩+৪+২=৯। আবার A->D ও একটা পাথ হতে পারে যেই পাথের মোট কস্ট ১০। যে পাথের কস্ট সবথেকে কম সেটাকে শর্টেস্ট পাথ বলে।

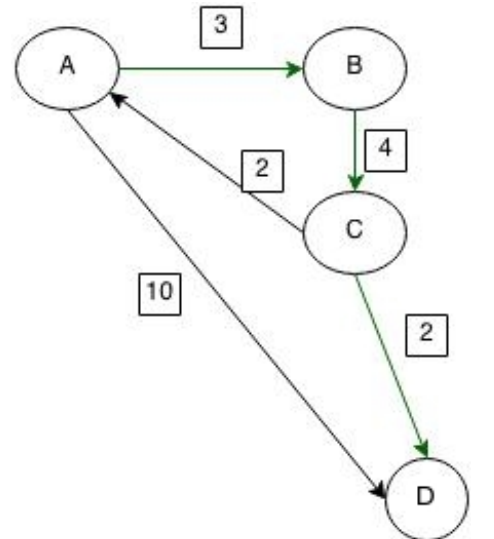


ডিগ্রী:

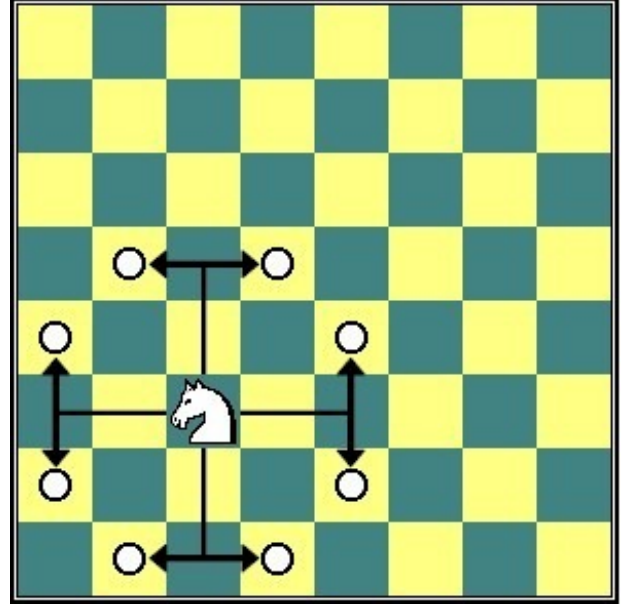
ডিরেক্টেড গ্রাফে একটা নোডে কয়টা এজ প্রবেশ করেছে তাকে ইনডিগ্রী, আর কোন নোড থেকে কয়টা এজ বের হয়েছে তাকে আউটডিগ্রী বলে। ছবিতে প্রতিটা নোডের ইনডিগ্রী আর আউটডিগ্রী দেখানো হয়েছে:

আনডিরেক্টেড গ্রাফে ইন বা আউটডিগ্রী আলাদা করা হয়না। একটা নোডের যতগুলো অ্যাডজেসেন্ট নোড আছে সেই সংখ্যাই নোডটার ডিগ্রী।

হ্যান্ডশেকিং লেমা একটা জিনিস আছে যেটা বলে একটা বিজোড় ডিগ্রীর নোডের সংখ্যা সবসময় জোড় হয়। উপরের গ্রাফে A আর C এর ডিগ্রী ৩, এরা বিজোড় ডিগ্রীর নোড। তাহলে বিজোড় ডিগ্রীর নোড আছে ২টা, ২ হলো একটা জোড় সংখ্যা। হ্যান্ডশেকিং করতে সবসময় ২টা হাত লাগে, ঠিক সেরকম একটা এজ সবসময় ২টা নোডকে যোগ করে। তুমি একটু চিন্তা করে দেখো:



২টা জোড় ডিগ্রীর নোডকে এজ দিয়ে যোগ করলে ২টা নতুন বিজোড় ডিগ্রীর নোড তৈরি হয়।



২টা বিজোড় ডিগ্রীর নোডকে এজ দিয়ে যোগ করলে ২টা বিজোড় ডিগ্রীর নোড কমে যায়।

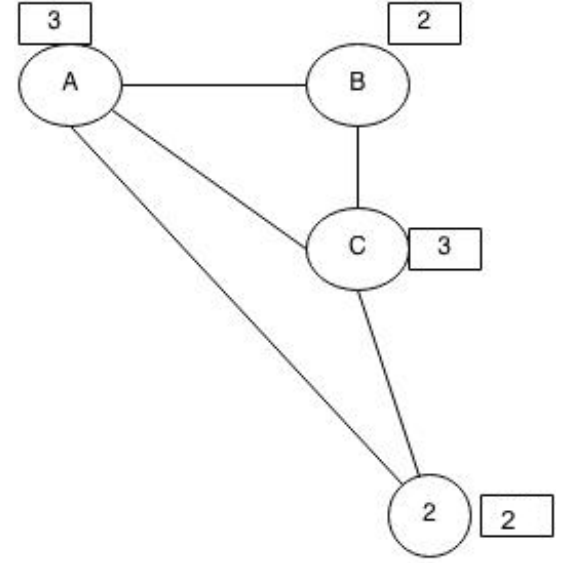
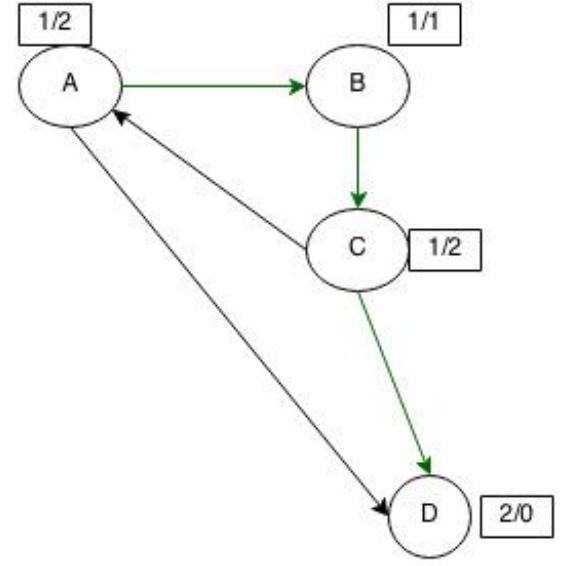
১টা জোড় আর একটা বিজোড় ডিগ্রীর নোড যোগ করলে মোট বিজোড় ডিগ্রীর নোড সমান থাকে(এক পাশে কমে, আরেক পাশে বাড়ে)।

তাহলে দেখা যাচ্ছে হয় ২টা করে বাড়তেসে বা ২টা করে কমতেসে বা সমান থাকছে, তাই বিজোর ডিগ্রীর নোডের সংখ্যা সবসময় জোড়।

একইভাবে এটাও দেখানো যায় একটা গ্রাফের ডিগ্রীগুলোর যোগফল হবে এজসংখ্যার দ্বিগুণ। উপরের গ্রাফে ডিগ্রীগুলোর যোগফল ১০, আর এজসংখ্যা ৫।

এগুলো গেল একেবারেই প্রাথমিক কথাবার্তা। পরবর্তি লেখায় তুমি জানতে পারবে কিভাবে [ড্যারিয়েবলে গ্রাফ স্টোর](#) করতে হয়। আশা করি গ্রাফ থিওরীতে তোমার যাত্রাটা দারুণ আনন্দের হবে, অনেক কিছু শিখতে পারবে।

[অন্যান্য পর্ব](#)



গ্রাফ থিওরিতে হাতেখড়ি – ২ (টিউটোরিয়াল)

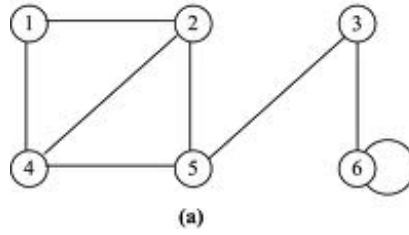
আগের পোস্টে আমরা দেখেছি গ্রাফ থিওরি কি কাজে লাগে, আর এলিমেন্টারি কিছু টার্ম শিখেছি। এখন আমরা আরেকটু ভিতরে প্রবেশ করবো। প্রথমেই আমাদের জানা দরকার একটা গ্রাফ কিভাবে ইনপুট নিয়ে স্টোর করে রাখা যায়। অনেকগুলো পদ্ধতির মধ্যে দুটি খুব কমন:

১. অ্যাডজেসেন্সি ম্যাট্রিক্স(adjacency matrix)

২. অ্যাডজেসেন্সি লিস্ট(adjacency list)

অ্যাডজেসেন্ট(adjacent) শব্দটার অর্থ “কোন কিছুর পাশে”। যেমন তোমার পাশের বাড়ির প্রতিবেশিরা তোমার অ্যাডজেসেন্ট। গ্রাফের ভাষায় এক নোডের সাথে আরেকটা নোডে যাওয়া গেলে ২য় নোডটি প্রথমটির অ্যাডজেসেন্ট। এই পোস্টে আমরা ম্যাট্রিক্সের সাহায্যে কোন নোড কার অ্যাডজেসেন্ট অর্থাৎ কোন কোন নোডের মাঝে এজ আছে সেটা কিভাবে স্টোর করা যায় দেখবো। ম্যাট্রিক্স বলতে এখানে জাস্ট ২-ডি অ্যারে বুঝানো হয়েছে, তাই ঘাবড়ে যাবার কিছু নেই!

গ্রাফের পাশে একটি টেবিল দেখতে পাচ্ছ। এটাই আমাদের অ্যাডজেসেন্সি ম্যাট্রিক্স। ম্যাট্রিক্সের $[i][j]$ ঘরে 1 থাকে যদি i থেকে j তে কোনো এজ থাকে, না থাকলে 0 বসিয়ে দেই।



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

(b)

এজগুলো ওয়েটেড হতে পারে, যেমন ঢাকা থেকে

চট্টগ্রামে একটা এজ দিয়ে বলে দিতে পারে শহর দুটির দূরত্ব ৩০০ কিলোমিটার। তাহলে তোমাকে ম্যাট্রিক্সে ওয়েটও বসাতে হবে।

উপরের গ্রাফটি বাইডিরেকশনাল বা আনডিরেক্টেড, অর্থাৎ ১ থেকে ২ এ যাওয়া গেলে ২ থেকে ১ এও যাওয়া যাবে। যদি গ্রাফটি ডিরেক্টেড হতো তাহলে এজগুলোর মধ্যে তীরচিহ্ন থাকতো। তখনো আমরা আগের মতো করেই ম্যাট্রিক্সে স্টোর করতে পারবো। নিচের ছবিতে সুন্দর করে সবগুলো কেস দেখানো হয়েছে:

একটা ব্যাপার লক্ষ্য করো, গ্রাফ আনডিরেক্টেড হলে ম্যাট্রিক্সটি সিমিট্রিক হয়ে যায়, অর্থাৎ $mat[i][j]=mat[j][i]$ হয়ে যায়।

ছোট একটা এক্সারসাইজ:

কল্পনা কর একটি গ্রাফ যার ৩টি নোড আছে edge সংখ্যা ৩, এবং সবগুলো edge bidirectional। edge গুলো হলো ১-২(cost ৫), ২-৩(cost ৮), ১-৩(cost ৩)। এটার adjacency matrix টা কেমন হবে?

চট করে নিজেই খাতায় একে ফেলতে চেষ্টা কর এবং নিচের উত্তরের সাথে মিলিয়ে দেখো:

```
0 5 3
5 0 8
3 8 0
```

আশা করি বুঝতে পারছ কিভাবে ম্যাট্রিক্সটি আকলাম। না বুঝলে উপরের অংশটা আরেকবার পড়ে ফেল।

গ্রাফ ইনপুট যেভাবে দেয়া হবে:

ঠিক উপরের ম্যাট্রিক্সটা প্রোগ্রামিং প্রবলেমে ইনপুট হিসাবে দিয়ে দেয়া হতে পারে, শুরুতে শুধু নোড সংখ্যা বলে দিবে। লক্ষ্য কর এই ম্যাট্রিক্সটা ইনপুট নিতে আমাদের এজ সংখ্যা জানা জরুরী না। আমাদের একটি ভ্যারিয়েবল লাগবে নোড সংখ্যা ইনপুট নিতে, আরেকটি ২-ডি আরে লাগবে ম্যাট্রিক্স ইনপুট নিতে।

C++

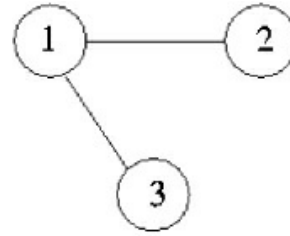
```
1 int N;
2 int matrix[100][100]; //এই সর্বোচ্চ ১০০ নোডের গ্রাফ স্টোর করা যাবে।
3
4 //ডিক্লেয়ার করার পরে ইনপুট নেবার পালা। খুব সহজ কাজ:
5 scanf("%d",&N);
6 for(int i=1;i<=N;i++)
7 for(int j=1;j<=N;j++)
8 scanf("%d",&matrix[i][j]);
```

সরাসরি ম্যাট্রিক্স না দিয়ে নোড সংখ্যা, edge সংখ্যা বলে দিয়ে edge গুলো কি কি বলে দিতে পারে, এভাবে:

```
3 3 //৩ টা নোড এবং ৩টা এজ
1 2 5 //node1-node2-cost
2 3 8
1 3 3
```

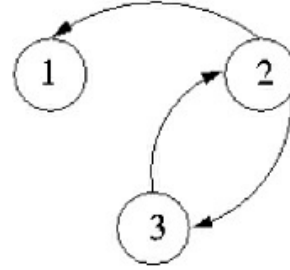
এটা ইনপুট নিব এভাবে:

C++



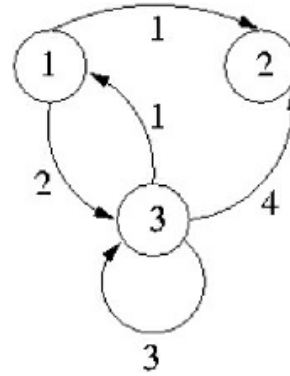
Undirected

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$



Directed

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



Directed Weighted

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 1 & 4 & 3 \end{pmatrix}$$

```

1  int Node,Edge;
2  int matrix[100][100];
3  scanf("%d%d",&Node,&Edge);
4  for(i=0;i<Edge;i++)
5  {
6  int n1,n2,cost;
7  scanf("%d%d%d",&n1,&n2,&cost);
8  matrix[n1][n2]=cost;
9  matrix[n2][n1]=cost;
10 }

```

আরো অনেক উপায়ে প্রবলেমে গ্রাফ ইনপুট দিতে পারে। নোডের নম্বর এলোমেলো হতে পারে, যেমন ৩টি নোডকে ১,২,৩ দিয়ে চিহ্নিত না করে ১০০,১০০০০,৪০০ নামে চিহ্নিত করা হতে পারে। সেক্ষেত্রে আমাদের ম্যাপিং করতে হবে। অর্থাৎ ১০০ কে আমরা ম্যাপ করব ১ দিয়ে, মানে ১০০ বলতে বুঝব ১, ১০০০০ বলতে বুঝব ২। `index` নামক একটি `array` রেখে `index[100]=1; index[100000]=2;` এভাবে চিহ্নিত করে দিলেই চলবে। পরে নোড নম্বর ইনপুট দিলে আমার ইনডেক্স থেকে আমাদের দেয়া নম্বর বের করে আনব। ব্যাপারটাকে বলা হয় অ্যারে কম্প্রেশন, তুমি বিস্তারিত জানতে চাইলে পরে কোনো সময় [আমার এই লেখাটা দেখতে পারো](#)।

অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সমস্যা:

মেমরি একটা বিশাল প্রবলেম, এজ যতগুলোই থাকুকনা কেন তোমার লাগছে $N*N$ সাইজের ম্যাট্রিক্স যেখানে N হলো নোড সংখ্যা। ১০০০০ টা নোড হলো $N*N$ ম্যাট্রিক্সের সাইজ দাড়াবে $8*10000*10000$ বাইট বা প্রায় ৩৮১ মেগাবাইট! এজ কম হলে এটা মেমরির বিশাল অপচয়।

কোনো একটা নোড u থেকে অন্য কোন কোন নোডে যাওয়া যায় বের করতে হলে আমাদের N টা নোডের সবগুলো চেক করে দেখতে হবে, টাইমের বিশাল অপচয়!

অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সুবিধা:

$u-v$ নোডের মধ্যে কানেকশন আছে নাকি বা `cost` কত সেটা খুব সহজেই `mat[u][v]` চেক করে জেনে যেতে পারি।

এই সমস্যাগুলো দূর করে দিবে অ্যাডজেসেন্সি লিস্ট, সাথে নতুন কিছু সমস্যাও হাজির করবে! তোমরা পরের পর্বে সেটা শিখবে। তার আগে তোমাকে একটা জিনিস শিখতে হবে, সেটা হলো `C++` এর স্ট্যান্ডার্ড টেমপ্লেট লাইব্রেরি (STL)। আমরা STL এর ভেক্টর ব্যবহার করে কাজ করবো কারণ এটা ব্যবহার করা খুব সহজ। তুমি নিচের দুটি লিংকের সাহায্যে খুবই সহজে শিখতে পারবে:

১: <http://sites.google.com/site/smilitude/stl> এটি ফাহিম ভাইয়ের ব্লগের লিংক, তার টিউটোরিয়াল গুলো অদ্বিতীয়।

২: <http://www.cplusplus.com/reference/stl/> STL এর বিভিন্ন ফাংশনের কাজ শেখার জন্য সেরা সাইট।

ভেক্টর ব্যবহার শেখা হয়ে গেলে পড়া শুরু করো পরের পর্ব: **adjacency list**

গ্রাফ থিওরিতে হাতেখড়ি – ৩ (ভ্যারিয়েবলে গ্রাফ স্টোর-২)

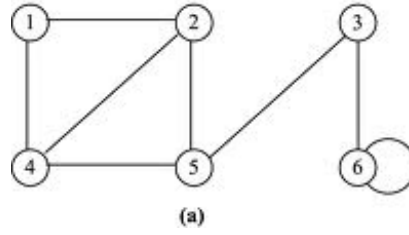
আগের পর্ব

সবগুলো পর্ব

এই পর্বে গ্রাফ স্টোর করার ২য় পদ্ধতি অ্যাডজেসেন্সি লিস্ট নিয়ে লিখব। এ পদ্ধতিতে গ্রাফ স্টোর করে কম মেমরি ব্যবহার করে আরো **efficient** কোড লেখা যায়। এ ক্ষেত্রে আমরা ডায়নামিক্যালি মেমরি অ্যালোকেট করব, ভয়ের কিছু নেই সি++ এর **standard template library(STL)** ব্যবহার করে খুব সহজে কাজটা করা যায়। আগের লেখার শেষের দিকে **STL** এর উপর কয়েকটি টিউটোরিয়ালের লিংক দিয়েছি, আশা করছি ভেক্টর কিভাবে কাজ করে এখন তুমি জানো।

অ্যাডজেসেন্সি লিস্ট শুনতে যতটা ভয়ংকর শুনায়, ব্যাপারটি আসলে তেমনই সহজ। আমরা আবার আগের পোস্টের ছবিটিতে ফিরে যাই:

এবার বাজার করার লিস্টের মত একটি লিস্ট বানাই:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

১ নম্বর নোডের সাথে যুক্ত আছে>>> ২
এবং ৪ নম্বর নোড

২ নম্বর নোডের সাথে যুক্ত আছে>>>

১, ৪ এবং ৫ নম্বর নোড

৩ নম্বর নোডের সাথে যুক্ত আছে>>> ৫ এবং ৬ নম্বর নোড

৪ নম্বর নোডের সাথে যুক্ত আছে>>> ১, ২ এবং ৫ নম্বর নোড

৫ নম্বর নোডের সাথে যুক্ত আছে>>> ২, ৩ এবং ৪ নম্বর নোড

৬ নম্বর নোডের সাথে যুক্ত আছে>>> ৩ এবং ৫ নম্বর নোড

এটাই অ্যাডজেসেন্সি লিস্ট, কোন নোডের সাথে কোন নোড যুক্ত আছে সেটার একটা তালিকা। কিন্তু কোড করার সময় কিভাবে এই লিস্টটা স্টোর করবো?

প্রথম উপায়(অ্যারে):

সাধারণ ২ডি অ্যারে ব্যবহার করে লিস্টটি স্টোর করা যায়। যেমন:

```
arr[1][1]=2, arr[1][2]=4;
```

```
arr[2][1]=1; arr[2][2]=4, arr[2][3]=5;
```

কিন্তু এভাবে স্টোর করলে কিছু সমস্যা আছে:

সমস্যা ১. আমাদের ৬টি নোড আছে। প্রতি নোডের সাথে সর্বোচ্চ ৬টি নোড যুক্ত থাকতে পারে(ধরে নিচ্ছি দুটি নোডের মধ্যে ১টির বেশি সংযোগ থাকবেনা)। এ ক্ষেত্রে আমাদের লাগবে [6][6] আকারের ইন্ডিক্সার অ্যারে। যদি ১ নম্বর নোডের সাথে মাত্র ২টি নোড যুক্ত থাকে তাহলে বাকি `array[1][0], array[1][1]` কাজে লাগবে, `array[1][2]` থেকে `array[1][6]` পর্যন্ত জায়গা কোনো কাজেই লাগবেনা। মনে হতে পারে এ আর এমন কি সমস্যা। কিন্তু চিন্তা কর ১০০০০ টি নোড আছে এমন একটি গ্রাফের কথা। `[10000][10000]` integer অ্যারে তুমি ব্যবহার করতে পারবেনা, **memory limit** অতিক্রম করে যাবে, এছাড়া এভাবে মেমরি অপচয় করা ভালো প্রোগ্রামারের লক্ষণ নয় :)। অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সময় যেমন মেমরির সমস্যা হয়েছিলো, এখনও সেই সমস্যা রয়ে যাবে।

সমস্যা ২. অ্যারের কোন ইনডেক্সে কয়টি এলিমেন্ট আছে তার হিসাব রাখতে প্রতি ইনডেক্সের জন্য আরেকটি ভ্যারিয়েবল মাইনটেইন করতে হবে।

দ্বিতীয় উপায়(ভেক্টর):

সমস্যাগুলো দূর করতে আমরা **STL vector** বা **list** ব্যবহার করে গ্রাফ স্টোর করব। ভেক্টর/লিস্ট তোমাকে লিস্টের সাইজ বলে দিতে হবেনা, খালি সর্বোচ্চ নোড সংখ্যা বলে দিলেই হবে। এই টিউটোরিয়ালে আমি ভেক্টর ব্যবহার করব কারণ **list** এ বেশ কিছু সমস্যা আছে।

১০০০০০ নোডের গ্রাফ ইনপুট দেয়ার সময় কখনো ম্যাট্রিক্স হিসাবে দিবেনা, তাহলে ইনপুটের আকারই মাত্রাতিরিক্ত বিশাল হয়ে যাবে। আগের পোস্টে

২য় উদাহরণে যেভাবে দেখিয়েছি সেরকম ইনপুট দিতে পারে, অর্থাৎ প্রথমে নোড আর এজ সংখ্যা বলে দিয়ে তারপর কোন নোডের সাথে কে যুক্ত আছে বলে দিবে। উপরের গ্রাফের জন্য ইনপুট:

```
6 8 //node-edge
1 2 //node1-node2
1 4
2 4
2 5
4 5
5 3
3 6
6 6
```

এটি ভেক্টর দিয়ে ইনপুট নিব এভাবে:

C++

```
1 #include
2 #include
3 using namespace std;
4 #define MAX 100000 //maximum node
5 vector<int>edges[MAX];
6 vector<int>cost[MAX]; //parallel vector to store costs;
7 int main()
8 {
9     int N,E,i;
10
11     scanf("%d%d",&N,&E);
12     for(i=1;i<=E;i++)
13     {
14         int x,y;
15         scanf("%d%d",&x,&y);
16         edges[x].push_back(y);
17         edges[y].push_back(x);
18         cost[x].push_back(1);
19         cost[y].push_back(1);
20     }
21     return 0;
22 }
```

cost নামক ভেক্টরটি এ গ্রাফের ক্ষেত্রে দরকার ছিলনা, তবে ওয়েটেড গ্রাফে অবশ্যই দরকার হবে। নিশ্চয়ই বুঝতে পারছ edge ও cost ভেক্টর দুটি সমান্তরাল ভাবে কাজ করবে, অর্থাৎ edge ভেক্টরের যে পজিশনে তুমি দুটি নির্দিষ্ট নোডের কানেকশন পাবে cost ভেক্টরের সেই পজিশনেই তুমি cost পাবে।

যদি ১০০০ বা তার কম নোড থাকে তাহলে ম্যাট্রিক্স বা লিস্ট কোনো ক্ষেত্রেই মেমরি সমস্যা হবেনা। তাও আমরা ভেক্টর দিয়েই গ্রাফ স্টোর করব। কারণ, চিন্তা কর তোমাকে ১০০টা নোডের ম্যাট্রিক্সে ১ এর সাথে কি কি সংযুক্ত আছে বের করতে matrix[1][0],matrix[1][1].....matrix[1][99] এভাবে ১০০টি পজিশন চেক করে কোনটায় কোনটায় ০ নেই বের করতে হবে, ১ নম্বর নোডের সাথে যতটি নোডই সংযুক্ত থাকুকনা কেন। তাই এখানেও অ্যারে আমাদের বাড়তি সুবিধা দিতে পারছেন।

একটা নোডের সাথে কোন কোন নোড যুক্ত আছে বের করা:

ধরো তুমি ১ নম্বর নোডের সাথে যুক্ত সবগুলো নোডের নম্বর চাও, তুমি তাহলে edges[1] এর সাইজ পর্যন্ত লুপ চালাবে এভাবে:

C++

```

1 </pre>
2 size=edges[1].size();
3 for(int i=0; i < size ; i++)
4 printf("%d ",edges[1][i]);
5 <pre>

```

১ >> ০ ১ ০ ০ ০ ১ ১ ০ পুরোটা ঘুরে আসার থেকে ১ >> ২,৬,৭ ঘুরে আসতে কম সময় লাগবে,তাই নয়কি? ।



অ্যাডজেসেন্সি ম্যাট্রিক্স কখন লিস্ট অপেক্ষা সুবিধাজনক?

যদি কোনো প্রবলেমে তোমার u, v নোডের এর মধ্যে কোনো এজ আছে নাকি চেক করতে বলে তখন লিস্ট ব্যবহার করলে তোমাকে লুপ চালিয়ে চেক করতে হবে, কিন্তু ম্যাট্রিক্সে জাস্ট $matrix[u][v]$ ইনডেক্স চেক করেই বলে দিতে পারবে তাদের মধ্যে কানেকশন আছে নাকি।

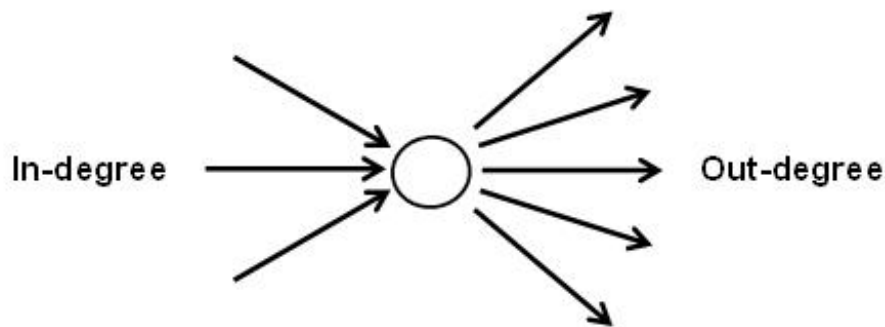
এক্সারসাইজ:

এ পর্যন্ত বুঝে থাকলে তুমি মোটামুটি bfs, dfs এর মত বেসিক অ্যালগোরিদম গুলো শেখার জন্য প্রস্তুত। পরবর্তি লেখাটি পড়ার আগে একটি ছোট **exercise** করে ফেল। এমন একটি কোড লিখ যেটায় উপরের মত করে ইনপুট দিলে নিচের কাজগুলো করে:

১. একটি **adjacency list** তৈরি করে। (গ্রাফটিকে **directed** ধরে নিবে, **bidirectional** নয়)

২. কোন নোডের সাথে কয়টা নোড যুক্ত আছে, নোডগুলো কি কি সেগুলো প্রিন্ট করে।

৩. **indegree** হলো একটি নোডে কয়টি নোড প্রবেশ করেছে, **outdegree** হলো ঠিক তার উল্টোটা। প্রতিটি নোডের **outdegree** ও **indegree** প্রিন্ট কর।

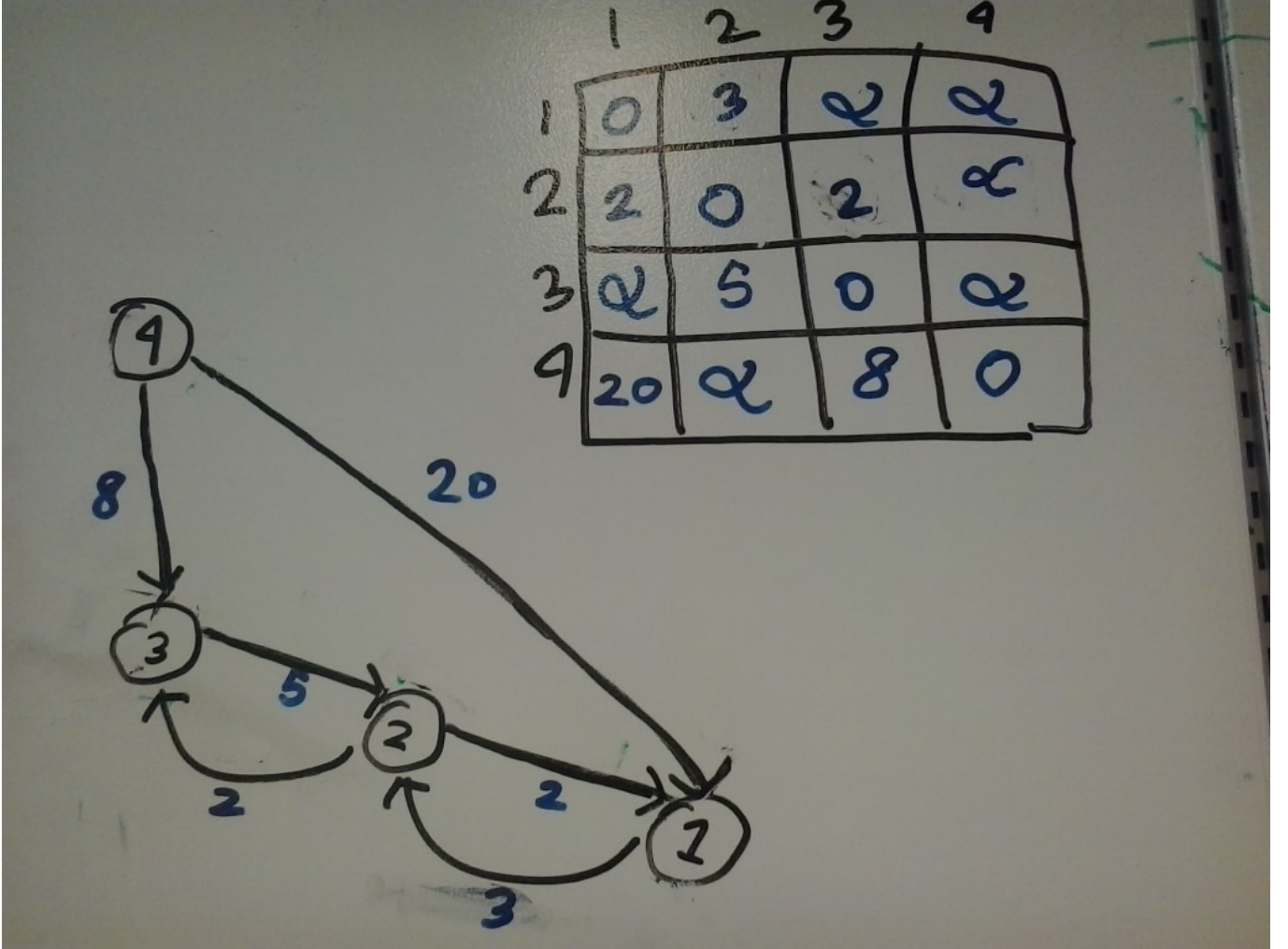


পর্ব-৪,বিএফএস:<http://www.shafaetsplanet.com/planetcoding/?p=604>

গ্রাফ থিওরিতে হাতেখড়ি ১০: ফ্লয়েড ওয়ার্শল

ফ্লয়েড ওয়ার্শল সম্ভবত সব থেকে ছোট আকারের গ্রাফ অ্যালগোরিদম, মাত্র ৩লাইনে এটা লেখা যায়! তবে ৩ লাইনের এই অ্যালগোরিদমেই বোঝার অনেক কিছু আছে। ফ্লয়েড ওয়ার্শলের কাজ হলো গ্রাফের প্রতিটা নোড থেকে অন্য সবগুলো নোডের সংক্ষিপ্ততম দূরত্ব বের করা। এ ধরনের অ্যালগোরিদমকে বলা হয় “অল-পেয়ার শর্টেস্ট পাথ” অ্যালগোরিদম। এই লেখাটা পড়ার আগে অ্যাডভেন্সেস ম্যাট্রিক্স সম্পর্কে জানতে হবে।

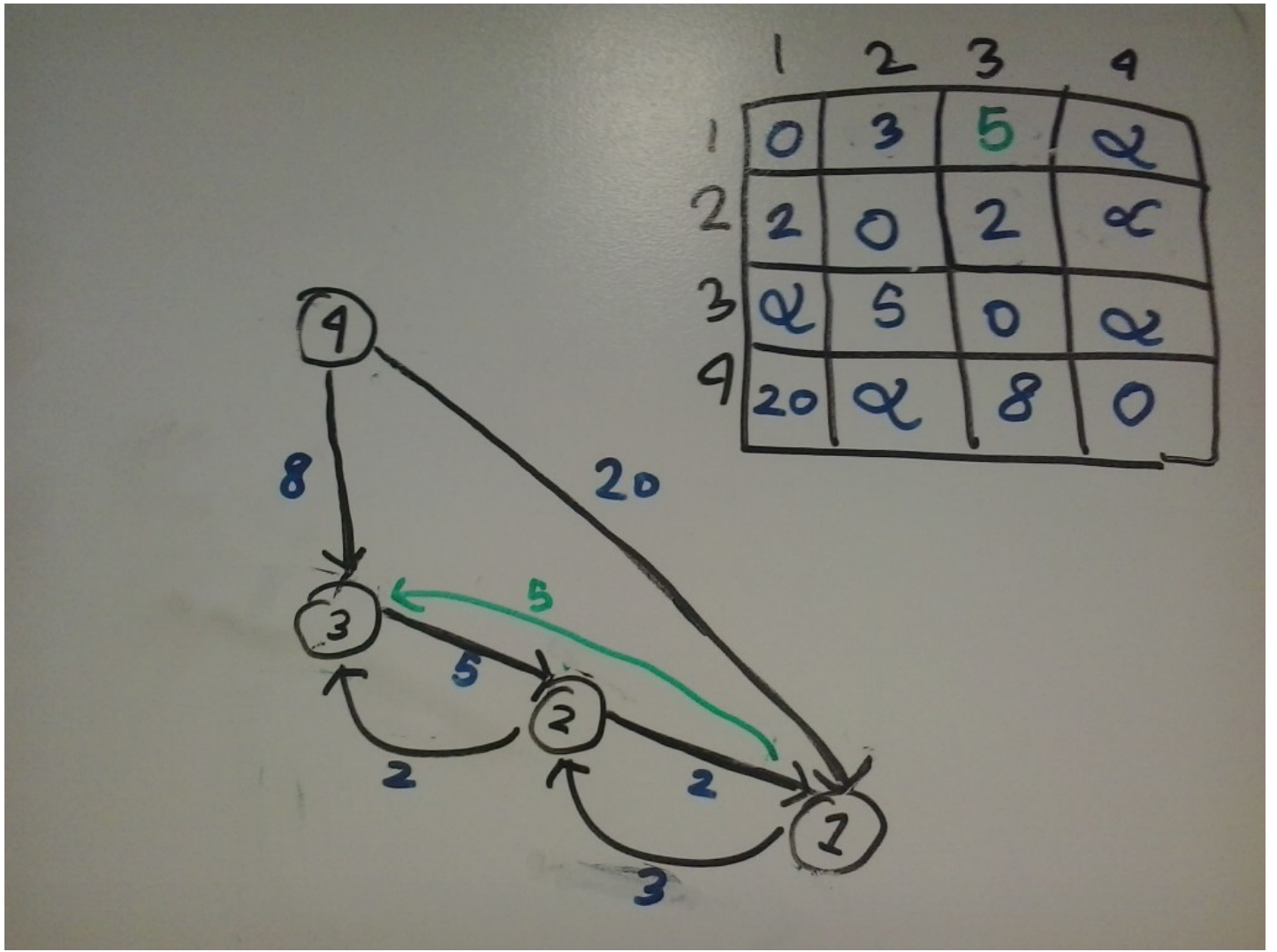
আমরা একটা গ্রাফের উপর কিছু সিমুলেশন করে সহজেই অ্যালগোরিদমটা বুঝতে পারি। নিচের ছবিটা দেখ:



ছবিতে চার নোডের একটা ওয়েটেড ডিরেক্টেড গ্রাফ দেখা যাচ্ছে। আর উপরে ডান কোনায় একটা ম্যাট্রিক্স। ম্যাট্রিক্সের u, v তম ঘরে বসানো হয়েছে $u-v$ এজ এর ওয়েট বা কস্ট। যাদের মধ্যে সরাসরি এজ নেই সেসব ঘরে অসীম বা ইনফিনিটি বসিয়ে দেয়া হয়েছে। আর কোনাকুনি ঘরগুলোতে মান ০ কারণ নিজের বাসা থেকে নিজের বাসাতেই যেতে কোন দূরত্ব অতিক্রম করতে হয় না!

এখন মনে করো “২” নম্বর নোডটাকে আমরা “মাকের নোড” হিসাবে ধরলাম। মাকের নোডকে আমরা বলবো k । তারমানে এখন $k=2$ । (আমরা একে একে সব নোডকেই মাকের নোড হিসাবে ধরবো, এটা যেকোন অর্ডারে করা যায়)

এখন যেকোন এক জোড়া নোড (i, j) নাও। ধরি $i=1, j=3$ । আমরা চাই u থেকে v তে যেতে, k নোডটাকে মাঝে রেখে। তাহলে আমাদের i থেকে k তে যেতে হবে, তারপর k থেকে থেকে j তে যেতে হবে। কিন্তু লাভ না হলে আমরা এভাবে যাব কেন? আমরা k কে মাঝে রেখে যাবো কেবল যদি মোট কস্ট(cost) কমে যায়। ১ থেকে ৩ এর বর্তমান দূরত্ব $matrix[1][3]=\text{ইনফিনিটি}$ । আর যদি $k=2$ কে মাঝখানে রাখি তাহলে দূরত্ব দাঁড়াবে $matrix[1][2] + matrix[2][3] = 3 + 2 = 5$ । তারমানে কস্ট কমে যাচ্ছে! আমরা গ্রাফটা আপডেট করে দিতে পারি এভাবে:



আমরা ২ কে “মাকের নোড” হিসাবে ব্যবহার করে ১ থেকে ৩ এ গিয়েছি মোট ৫ কস্ট এ। গ্রাফে তাহলে ১ থেকে ৩ এ সরাসরি একটা এজ দিয়ে দিতে পারি ৫ কস্ট এ।

এখন আবার ধর $i=2, j=4$ । আর আগের মতই $k=2$ । এবার $matrix[2][4]=\infty$ । এদিকে $matrix[2][2] + matrix[2][4] = 0 + \infty$ । এবার কিন্তু দূরত্ব কমলো না। তাই গ্রাফ আপডেট করার দরকার নেই। নিশ্চয়ই বুঝতে পারছো আপডেটের শর্তটা হবে এরকম:

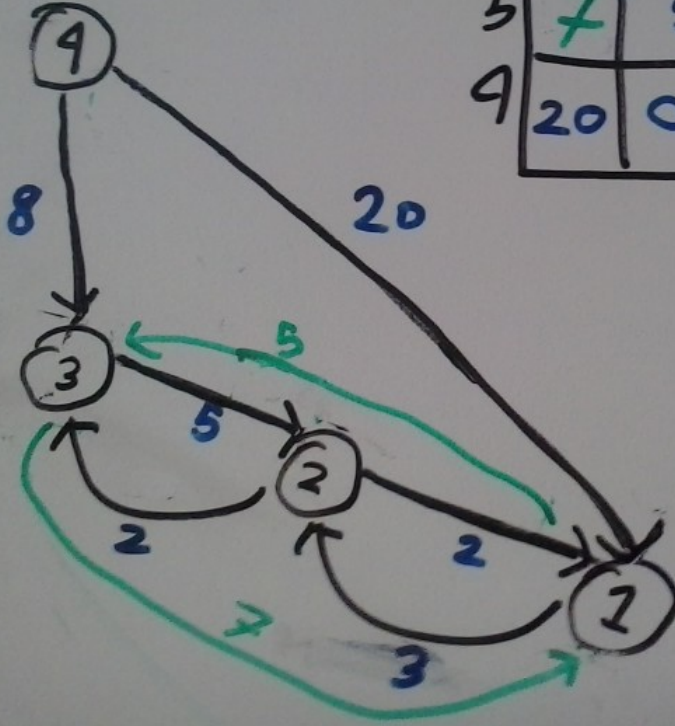
```
if(matrix[i][k] + matrix[k][j] < matrix[i][j])
    matrix[i][j] = matrix[i][k] + matrix[k][j]
```

অথবা আমরা একলাইনে লিখতে পারি:

```
matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])
```

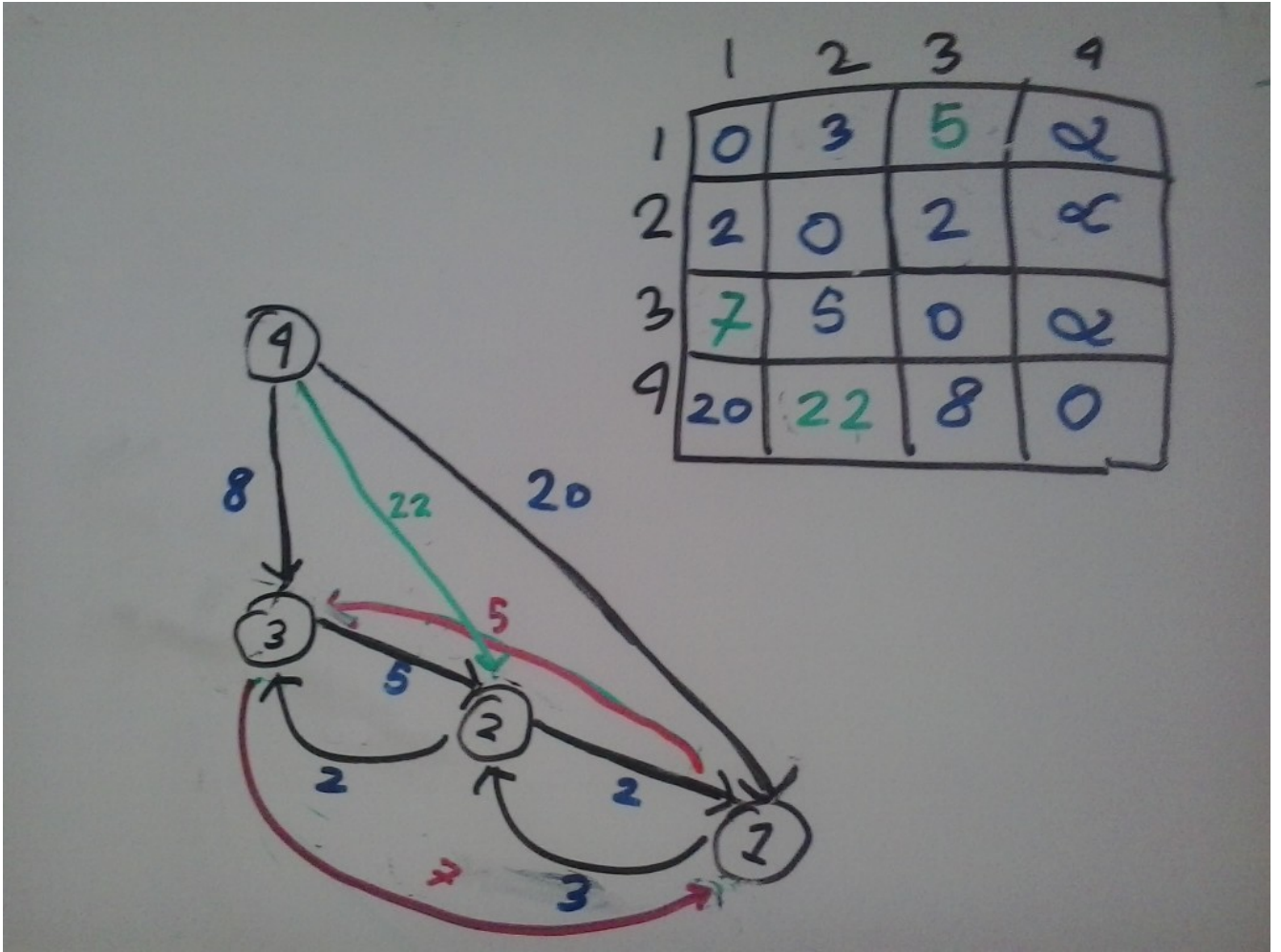
এখন $k=2$ স্থির রেখে আমরা i, j এর সবরকমের কম্বিনেশন নিবো। তুমি নিজেই চিন্তা করলে দেখবে $k=2$ এর জন্য $i=3, j=1$ এই কম্বিনেশনে আমরা আরেকটা নতুন এজ পাবো, বাকি গ্রাফ আগের মতই থাকবে।

	1	2	3	4
1	0	3	5	∞
2	2	0	2	∞
3	7	5	0	∞
4	20	∞	8	0



এবার আমরা $k=1$ কে মাঝের সোঁদ হিসাবে চিন্তা করি। মাথা খাটাতে চাইলে নিচে দেখার আগে নিজেই খাতায় একে ফেলতে পারো নতুন এজগুলো।

এবার একটা মাত্র এজ যোগ হবে। $i=4, j=2$ হলে আমরা ৪ থেকে ১ হয়ে ২ নম্বর নোডে যেতে পারি ২২ কস্ট এ। তাহলে গ্রাফটা হবে এরকম:

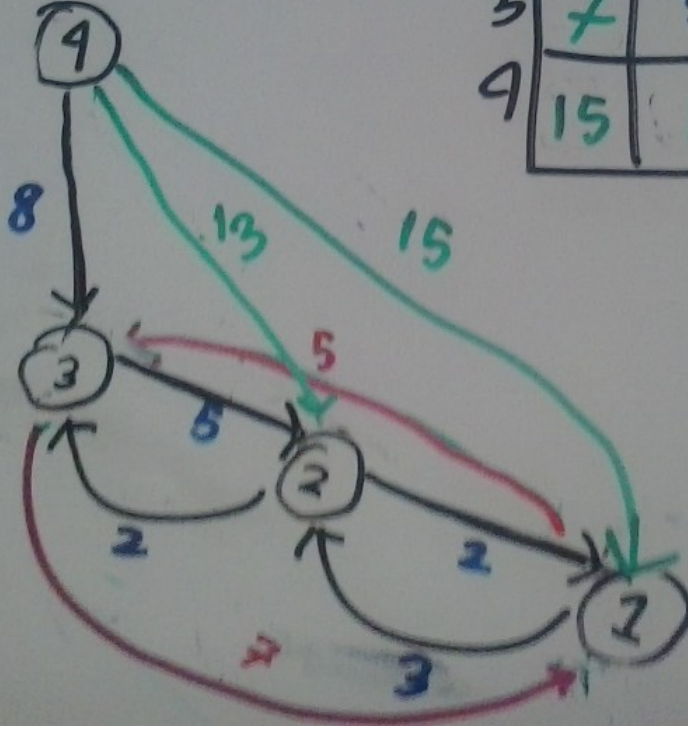


এবার ৩ নম্বর নোডকে মাকের নোড হিসাবে ধরবো। আবারো নিজে চেষ্টা করে তারপর নিচের অংশ দেখ।

এবার কিছু মজার জিনিস ঘটবে। ৪ থেকে ১ এ আগে লাগছিল ২০ কস্ট। এখন ৩ কে মাঝে রেখে ৪ থেকে ১ এ গেলে লাগবে $৮+৭=১৫$ কস্ট। লক্ষ্য কর একদম শুরুতে ৩ থেকে ১ এ আমাদের এজ ছিল না। কিন্তু আপডেট করার সময় আমরা এজ বসিয়ে দিয়েছি, এখন ৩ থেকে ১ এ যদিও সরাসরি চলে যাচ্ছি, মূল গ্রাফে আসলে $৩ \rightarrow ২ \rightarrow ১$ পথে যাচ্ছি।

একই ভাবে ৪ থেকে ২ এর ২২ কস্ট এর পথটা আপডেট হয়ে ১৩ হয়ে যাবে।

	1	2	3	4
1	0	3	5	2
2	2	0	2	∞
3	7	5	0	∞
4	15	13	8	0



$k=8$ এর জন্য আর কোন আপডেট হবে না কারণ কোন নোড থেকে 8 এ যাওয়া যায় না।

এখন আমরা ম্যাট্রিক্স দেখেই বলে দিতে পারছি কোন নোড থেকে কোন নোডে কত কস্ট এ যাওয়া যায়। ইনফিনিটি থাকা মানে সেই নোডে যাবার পথ নেই।

ইনপুট থেকে অ্যাডজেসেন্সি ম্যাট্রিক্স বানাবার পর তাহলে কাজ খুবই সহজ:

C++

```

1 for k from 1 to |V|
2   for i from 1 to |V| \ |V| = number of nodes
3     for j from 1 to |V|
4       if matrix[i][j] > matrix[i][k] + matrix[k][j]
5         matrix[i][j] ← matrix[i][k] + matrix[k][j]

```

কোডে আমরা k এর লুপটা ১ থেকে চালাচ্ছি যদিও উদাহরণে আগে ২ নিয়েছি। এটা আসলে যেকোন অর্ডারেই করা যায়, তুমি আগে ১ নিলেও দেখবে অ্যালগোরিদম কাজ করবে।

এভাবেতো আমরা শুধু পাথের কস্ট পেলাম, পাথটা কিসের পাব?

ধরো আমাদের একটা ম্যাট্রিক্স আছে $next[][]$ । এখন $next[i][j]$ দিয়ে আমরা বুঝি i থেকে j তে যেতে হলে পরবর্তি যে নোড এ যেতে হবে সেই নোডটা। তাহলে একদম শুরুতে সব i, j এর জন্য $next[i][j] = j$ হবে। কারণ শুরুতে কোন “মাঝের নোড” নেই এবং তখনও শর্টেস্ট পাথ বের করা শেষ হয় নি।

এখন আমরা যখন $matrix[i][j]$ আপডেট করবো লুপের সেটার মানে হলো মাঝে একটা নোড k ব্যবহার করে আমরা যাবো। লক্ষ্য কর আমরা কিন্তু মূল গ্রাফে সরাসরি এজ দিয়ে i থেকে k তে নাও যেতে পারি, আমরা শুধু জানি i, j নোড দুটোর মাঝে একটা নোড k আছে যেখানে

আমাদের যেতে হবে j তে যাবার আগে। i থেকে k তে যাবার পথে পরবর্তি যে নোডে যেতে হবে সেটা রাখা আছে $next[i][k]$ তে! তাহলে $next[i][j] = next[i][k]$ হয়ে যাবে।

```
1  for k from 1 to |V|
2    for i from 1 to |V|
3      for j from 1 to |V|
4        if matrix[i][k] + matrix[k][j] < matrix[i][j] then
5          matrix[i][j] ← matrix[i][k] + matrix[k][j]
6          next[i][j] ← next[i][k]
7
8
9  findPath(i, j)
10   path = [i]
11   while i ≠ j
12     i ← next[i][j]
13     path.append(i)
14   return path
```

findpath ফাংশনে আমরা j কে ফিক্সড রেখে $next$ অ্যারে ধরে আগাছি যতক্ষণ না j তে পৌঁছাছি। তাহলে আমরা পেয়ে গেলাম পথ!

ট্রান্সিটিভ ক্লোজার(Transitive Closure):

ধরো আমাদের অ্যাডজেন্সি ম্যাট্রিক্সটা এরকম:

$matrix[i][j] = 1$ যদি i থেকে j তে সরাসরি এজ থাকে
 $matrix[i][j] = 0$ যদি এজ না থাকে

এখন আমরা এমন একটা ম্যাট্রিক্স তৈরি করতে চাই যেটা দেখে বলে দেয়া যাবে i থেকে j তে এক বা একাধিক এজ ব্যবহার করে যাওয়া যায় কিনা। আমরা চাইলে উপরের মত করে 0 এর জায়গায় ইনফিনিটি দিয়ে শর্টেস্ট পথ বের করে কাজটা করতে পারতাম। কিন্তু এক্ষেত্রে “OR” আর “AND” অপারেশন ব্যবহার আরো দ্রুত কাজটা করা যায়। এখন আপডেটের শর্তটা হয়ে যাবে এরকম:

$matrix[i][j] = matrix[i][j] \vee (matrix[i][k] \wedge matrix[k][j])$

এটার মানে $matrix[i][j]$ তে তখনই 1 বসবে যখন হয় “ $matrix[i][j]$ তে 1 আছে” অথবা “ $matrix[i][k]$ এবং $matrix[k][j]$ ” দুটোতেই 1 আছে। তারমানে হয় সরাসরি যেতে হবে অথবা মাঝে একটা নোড k ব্যবহার করে যেতে হবে।

কমপ্লেক্সিটি:

৩টা নেস্টেড লুপ ঘুরছে নোড সংখ্যার উপর, টাইম কমপ্লেক্সিটি $O(n^3)$ । ২ডি ম্যাট্রিক্স ব্যবহার করায় স্পেস কমপ্লেক্সিটি $O(n^2)$ ।

কিছু প্রশ্ন:

১. k এর লুপটা কি i, j লুপের ভিতর দিলে অ্যালগোরিদম কাজ করত?

২. গ্রাফে নেগেটিভ কস্ট থাকলে ফ্লয়েড ওয়ারশাল কাজ করবে কি?

বিলেটেড প্রবলেম:

[Page Hopping](#)

[05-2 Rendezvous](#)

[Minimum Transport Cost](#)

[Asterix and Obelix](#)

আরো অনেক প্রবলেম

হয়্যাপি কোডিং!

গ্রাফ থিওরিতে হাতেখড়ি ১১: বেলম্যান ফোর্ড

বেলম্যান ফোর্ড গ্রাফে শর্টেস্ট পথ বের করার একটা অ্যালগোরিদম। এই অ্যালগোরিদম একটা নোডকে সোর্স ধরে সেখান থেকে সব নোডের সংক্ষিপ্ততম বা শর্টেস্ট পথ বের করতে পারে। আমরা একদম শুরুতে এই কাজ করার জন্য ব্রেডথ ফার্স্ট সার্চ শিখেছি। কিন্তু **বিএফএস(BFS)** যেহেতু ওয়েটেড গ্রাফে কাজ করে না তাই এরপর আমরা শিখেছি ডায়াক্সট্রা অ্যালগোরিদম। এখন বেলম্যান ফোর্ড শিখব কারণ আগের কোনো অ্যালগোরিদমই নেগেটিভ ওয়েট এর এজ আছে এমন গ্রাফে কাজ করে না।

আমরা **ডায়াক্সট্রা** শেখার সময় রিল্যাক্সেশন নামের একটা ব্যাপার শিখেছিলাম। তোমার যদি মনে না থাকে বা ডায়াক্সট্রা না শিখে থাকো তাহলে আমরা প্রথমে একটু ঝালাই করে নেই আরেকবার। মনে থাকলে পরের অংশটা বাদ দিয়ে সরাসরি **এখানে যেতে পার**।

এজ রিল্যাক্সেশন:

ধর একটা গ্রাফে সোর্স থেকে প্রতিটা নোডের ডিসটেন্স/কস্ট রাখা হয়েছে $d[]$ অ্যারেতে। যেমন $d[3]$ মানে হলো সোর্স থেকে বিভিন্ন এজ পার হয়ে ৩ নম্বর নোড এ আসতে মোট $d[3]$ ডিসটেন্স পার করতে হয়েছে। যদি ডিসটেন্স জানা না থাকে তাহলে ইনফিনিটি অর্থাৎ অনেক বড় একটা মান রেখে দিবে। আর $cost[u][v]$ তে রাখা আছে $u-v$ এজ এর কস্ট।

ধর তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসি তে গেলে ১০ মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে ২৫ মিনিটে। তাহলে ফার্মগেটকে সোর্স ধরে আমরা বলতে পারি:

$$d[টিএসসি]=১০, d[কার্জনহল]=২৫$$

এখন তুমি দেখলে টিএসসি থেকে ৭ মিনিটে কার্জনে চলে যাওয়া যায়,

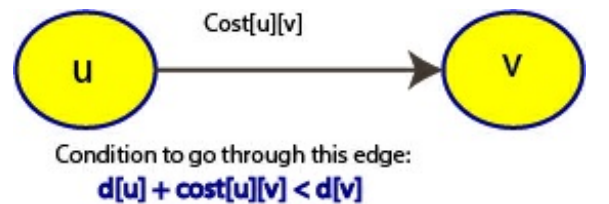
$$cost[টিএসসি][কার্জন]=৭$$

তাহলে তুমি ২৫ মিনিটের জায়গায় মাত্র $১০+৭=১৭$ মিনিটে কার্জনহলে যেতে পারবে। যেহেতু তুমি দেখেছো:

$$d[টিএসসি]+cost[টিএসসি][কার্জন] < d[কার্জনহল]$$

তাই তুমি এই নতুন রাস্তা দিয়ে কার্জন হলে গিয়ে $d[কার্জনহল]=d[টিএসসি]+cost[টিএসসি][কার্জন]$ বানিয়ে দিতেই পারো!!

উপরের ছবিটা সেটাই বলছে। আমরা u থেকে v তে যাবো যদি $d[u]+cost[u][v] < d[v]$ হয়। আর $d[v]$ কে আপডেট করে $d[v]=d[u]+cost[u][v]$ বানিয়ে দিবে। ভবিষ্যতে যদি কার্জনহলে অন্য রাস্তা দিয়ে আরো কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে কম্পেয়ার করে আপডেট করি দিবে। ব্যাপারটা অনেকটা এরকম:



C++

```
1 if(d[u]+cost[u][v] < d[v])
2 {
3   d[v]=d[u]+cost[u][v];
4 }
```

এটা হলো এজ রিল্যাক্সেশন। এখন আমরা বেলম্যান ফোর্ড শেখার জন্য তৈরি।

বেলম্যান ফোর্ড

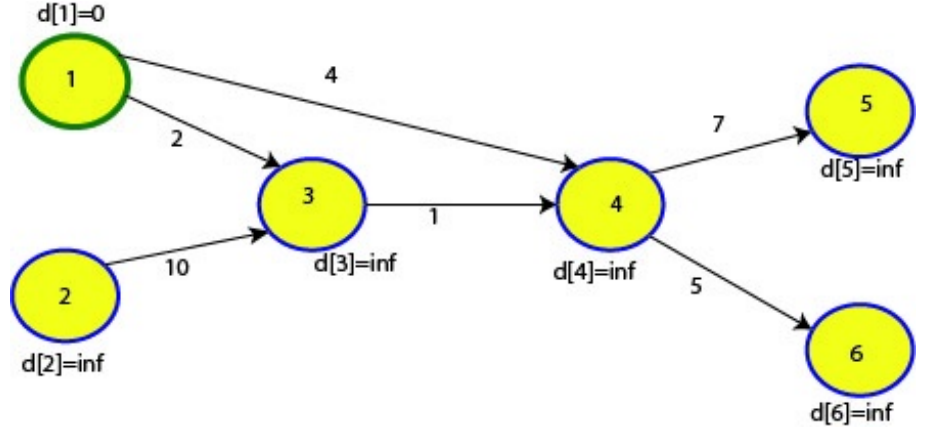
নিচের গ্রাফে আমরা ১ থেকে শুরু করে প্রতিটা নোডে যাবার শর্টেস্ট পাথ বের করতে চাই:

শুরুতে $d[1]=0$ কারণ ১ হলো সোর্স।
বাকিসবগুলোতে ইনফিনিটি রেখেছি কারণ আমরা এখনও জানিনা শর্টেস্ট পাথের কস্ট কত।

তুমি এজ রিল্যাক্স কিভাবে করতে হয় এরই মধ্য দিয়ে শিখে গেছ। এখন কাজ হলো সবগুলো এজকে একবার করে রিল্যাক্স করা, যেকোন অর্ডারে। একবার 'গ্রাফ রিল্যাক্স' করার মানে হল গ্রাফটার সবগুলো এজকে একবার করে রিল্যাক্স করা। আমি নিচের অর্ডারে রিল্যাক্স করতে চাই,

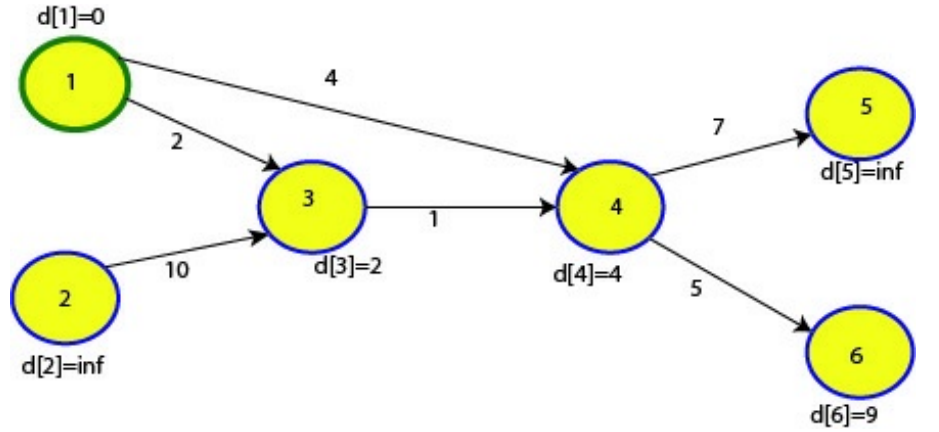
Serial	1	2	3	4	5	6
Edge	4 -> 5	3 -> 4	1 -> 3	1 -> 4	4 -> 6	2 -> 3

তুমি চাইলে অনায়াসে যেকোনো অর্ডারেও এজগুলো নিতে পারতে। এখন চিন্তা কর এজগুলোকে একবার রিল্যাক্স করলে আমরা $d[]$ অ্যাারেতে কি পার? সোর্স থেকে শুরু করে সর্বোচ্চ ১টা এজ ব্যবহার করে অনায়াসে নোডে যাবার শর্টেস্ট পাথের কস্ট আমরা পেয়ে যাব। উপরের ছবিতে রিল্যাক্স করার পর $d[]$ এর মানগুলো আপডেট করে দাও। করার পর ছবিটা নিচের মত হবার কথা:

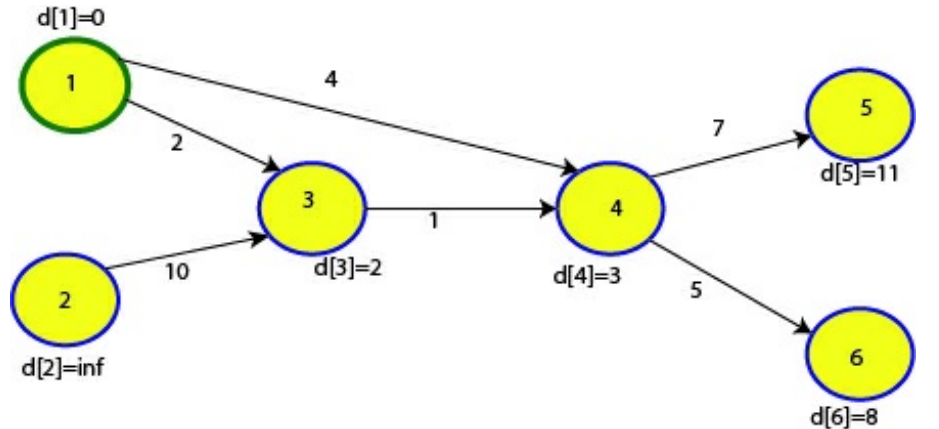


এজ রিল্যাক্স করার সময় কিছু নোড এর কস্ট আপডেট করতে পারি নি কারণ $d[u]+cost[u,v] < d[v]$ শর্তটা পূরণ করে নি। বাকি এজগুলো আপডেট করার পর $d[]$ অ্যাারের এর মান উপরের ছবির মত পেয়েছি। ১ নম্বর নোড থেকে শুরু করে সর্বোচ্চ ১টা এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথ এখন আমরা জানি!

এখন সর্বোচ্চ ২টা এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথের কস্ট বের করতে আরেকবার রিল্যাক্স করে ফেলি! আবারো যেকোন অর্ডারে করা যাবে, তবে প্রথমে যে অর্ডারে করেছি সেভাবেই প্রতিবার করা কোড লেখার সময় সুবিধাজনক।



একটা ব্যাপার লক্ষ্য কর, ১ থেকে ৬ তে যাবার শর্টেস্ট পথে ৩ টা এজ আছে (১->৩, ৩->৪, ৪->৬) এবং পথের দৈর্ঘ্য $২+১+৫=৮$ । মাত্র ২বার রিল্যাক্স করলেও আমরা এখনই $d[6]$ তে ৮ পেয়ে গেছি, অথচ আমাদের এখন সর্বোচ্চ ২টা এজ ব্যবহার করে শর্টেস্ট পাথের কস্ট পাবার কথা। এটা নির্ভর করে তুমি কোন অর্ডারে এজ রিল্যাক্স করেছ তার উপর। সে কারণে এ যাবার শর্টেস্ট পাথ ১০ হলেও $d[5]$ এ এখনো ১০ পাইনি। **X** বার 'গ্রাফ রিল্যাক্স' করলে সর্বোচ্চ **X** টা এজ ব্যবহার করে সোর্স প্রতিটা নোডে যাবার শর্টেস্ট পাথ তুমি নিশ্চিত ভাবে পাবে। **X** এর থেকে বেশি এজের ব্যবহার করে প্রতিটা নোডে যাবার শর্টেস্ট পাথ তুমি **X** বার গ্রাফ রিল্যাক্সের পর পেতেও পার, নাও পেতে পার, সেটা এজ এর অর্ডারের উপর নির্ভর করে।



এখন ৩য় বারের মত রিল্যাক্স করি:

এবার শুধু মাত্র ৫ নম্বর নোড আপডেট হবে।

এরপরে আমরা আর যতই আপডেট করি, $d[]$ অ্যারেতে কোনো পরিবর্তন হবে না, আমরা ১ থেকে প্রতিটা নোডে যাবার শর্টস্ট পাথ পেয়ে গিয়েছি।

এখন স্বাভাবিকভাবেই প্রশ্ন আসবে যে রিল্যাক্স কয়বার করতে হবে? গ্রাফে যদি নোড n টা থাকে তাহলে এক নোড থেকে অন্য নোডে যেতে সর্বোচ্চ $n-1$ টা এজ ব্যবহার করতে হবে।

তারমানে কোনো নোড সর্বোচ্চ $n-1$ বার আপডেট হতে পারে। তাই রিল্যাক্স করার লুপটাও

চালাতে হবে $n-1$ বার। তবে আমরা যেরকম উপরের গ্রাফে দেখেছি ৩বারের পরেই আর কোন নোড আপডেট করা যাচ্ছে না, সেরকম হলে আর নতুন করে রিল্যাক্স করার দরকার নাই।

এখন নিচের মাত্র ৩নোডের গ্রাফটায় বেলম্যান ফোর্ড অ্যালগোরিদম চালাও, অর্থাৎ যতক্ষণ কোন নোড আপডেট করা যায় ততক্ষণ পুরো গ্রাফটা রিল্যাক্স কর:

Serial	1	2	3
Edge	2 -> 3	1 -> 2	3 -> 1

প্রথমবার রিল্যাক্স করার পর পাব:

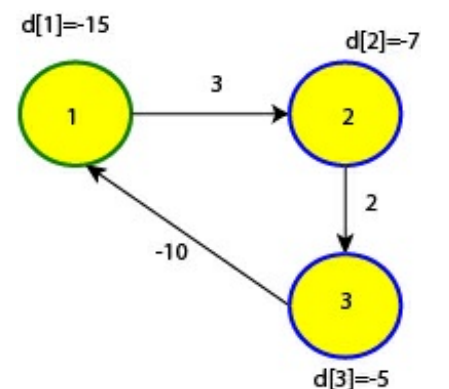
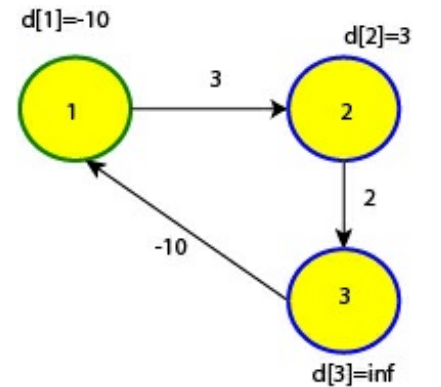
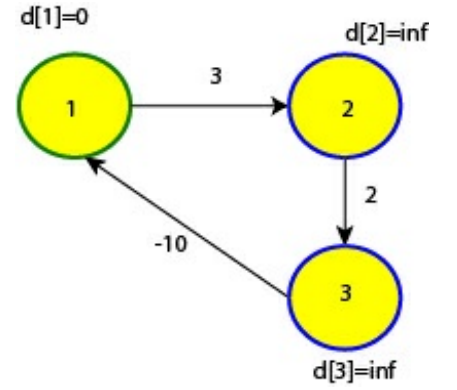
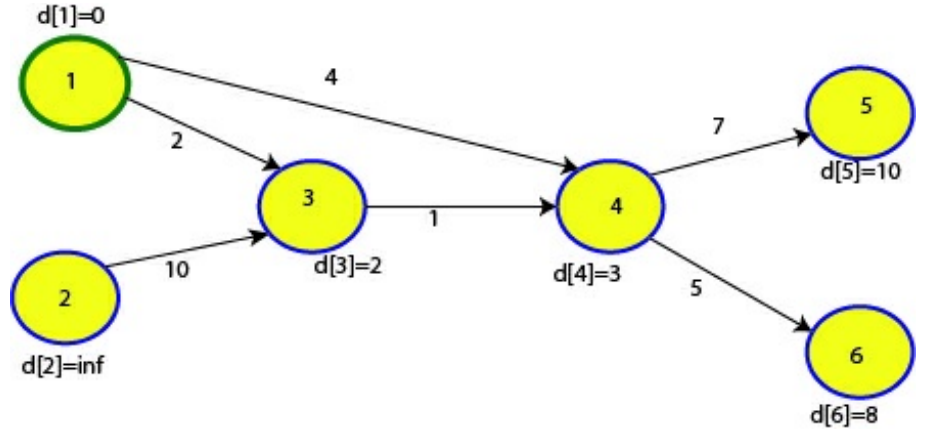
২য়বার করার পর:

৩ নোডের গ্রাফে সোর্স থেকে কোনো নোডে শর্টস্ট যেতে ২টার বেশি এজ লাগবে না, ৩য় বার রিল্যাক্স করার চেষ্টা করলে কোনো নোড আপডেট হবার কথা না। কিন্তু এই গ্রাফে আপডেট হচ্ছে:

এটা হচ্ছে কারণ $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ সাইকেলটার মোট ওয়েট নেগেটিভ ($3+2-10=-5$)। তাই তুমি যতবার এই সাইকেলে ঘুরবে শর্টস্ট পাথ তত ছোট হতে থাকবে। তাই নেগেটিভ সাইকেল থাকলে এবং সোর্স থেকে সেই নেগেটিভ সাইকেলে যাবার পথ থাকলে সোর্সের শর্টস্ট পাথ আনডিফাইনড বা অসংজ্ঞায়িত। যদি $n-1$ বার গ্রাফ রিল্যাক্স করার পর দেখি যে n তম বারও কোনো নোডের কস্ট(cost) আপডেট করা যায় তখন বুঝতে হবে আমরা নেগেটিভ সাইকেলে গিয়ে পরেছি, শর্টস্ট পাথ বের করা সম্ভব না।

আমাদের কোড তাহলে হবে এরকম:

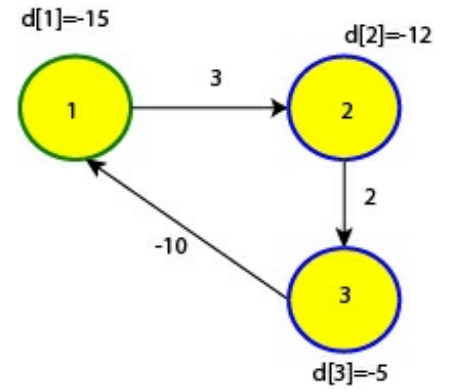
bellman
C++



```

1 //Bellman ford algorithm
2 /*
3 Sample input file:
4 3 3
5 1 2 2
6 2 3 3
7 3 1 -100
8 */
9 #include
10 #include
11 using namespace std;
12
13 #define MAXE 10000
14 #define MAXN 100
15 int main() {
16
17     freopen("in","r",stdin);
18     int node,edge;
19     int d[MAXN],edge_u[MAXE],edge_v[MAXE],edge_cost[MAXE];
20     cin>>node>>edge;
21     for(int i=1;i<=node;i++) d[i]=1000000000;
22     d[1]=0; //distance to source is 0
23     for(int i=1;i<=edge;i++)
24     {
25         cin>>edge_u[i]>>edge_v[i]>>edge_cost[i];
26     }
27     int neg_cycle=false;
28     for(int step=1;step<=node;step++)
29     {
30         int updated=false;
31         for(int i=1;i<=edge;i++)
32         {
33             int u=edge_u[i],v=edge_v[i];
34             if(d[u]+edge_cost[i]<d[v])
35             {
36                 updated=true;
37                 if(step==node)neg_cycle=true; //If we can update in n'th
38 step, there is a negative cycle
39                 d[v]=d[u]+edge_cost[i];
40             }
41         }
42         if(updated==false)break; //If we can't update in a any step, no
43 need to try anymore
44     }
45
46     if(neg_cycle==false)
47     {
48         for(int i=1;i<=node;i++)cout<<d[i]<<endl;
49     }
50     else puts("Negative cycle detected");
51     return 0;
52 }

```



পাথ প্রিন্ট করা:

বিএফএস বা ডায়াব্রুট্টাতে যেভাবে পাথ প্রিন্ট করে ঠিক সেভাবে এখানেও পাথ প্রিন্ট করা যাবে। `previous[]` নামের একটা অ্যারে নাও। `previous[v]=u` মানে হলো `v` তম নোডে তুমি `u` থেকে এসেছ। শুরুতে অ্যারেতে ইনফিনিটি থাকবে। `u->v` এজটা রিল্যাক্স করার সময় `previous[v]=u` করে দাও। এখন তুমি `previous` অ্যারে দেখে সোর্স থেকে যেকোন নোডের পাথ বের করে ফেলতে পারবে।

কমপ্লেক্সিটি:

সর্বোচ্চ $n-1$ বার প্রতিটা এজকে রিল্যাক্স করতে হবে, টাইম কমপ্লেক্সিটি $O(n \cdot e)$ ।

চিত্রা করার জন্য কিছু প্রবলেম:

- তোমাকে একটা গ্রাফ দিয়ে বলা হল শর্টেস্ট পাথে সর্বোচ্চ x টা এজ থাকতে পারে। এবার কিভাবে শর্টেস্ট পাথ বের করবে? ([UVA 11280](#))
- একটি গ্রাফে কোন কোন নোড নেগেটিভ সাইকেলের অংশ কিভাবে বের করবে? (হিন্টস: স্ট্রংলি কানেক্টেড কম্পোনেন্ট + বেলম্যান ফোর্ড)

রিলেটেড কিছু প্রবলেম:

[UVA 558\(সহজ\)](#)

[LOJ 1108](#)

[UVA 10449](#)

হ্যাপি কোডিং!

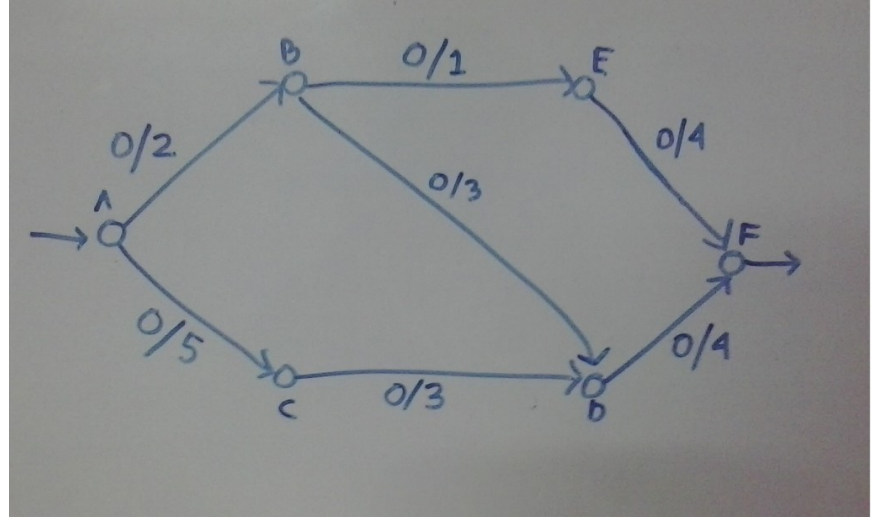
গ্রাফ থিওরিতে হাতেখড়ি ১২ – ম্যাক্সিমাম ফ্লো (১)

এই লেখায় আমরা গ্রাফে ম্যাক্সিমাম ফ্লো বের করার অ্যালগোরিদম শিখবো। ম্যাক্স ফ্লো এর ধারণাটা ব্যবহার করে বেশ কিছু ইন্টারেস্টিং প্রবলেম সলভ করা যায়, তাই এটা শেখা খুবই গুরুত্বপূর্ণ। এই লেখাটা পড়ার আগে তোমাকে [গ্রাফ থিওরি](#)র বেসিক অ্যালগোরিদমগুলো, বিশেষ করে শর্টেস্ট পাথ বের করার অ্যালগোরিদমগুলো ভালো করে শিখে নিবে।

প্রথমেই আমরা দেখি একটি খুবই সাধারণ ম্যাক্স ফ্লো প্রবলেম:

চিত্র: ১

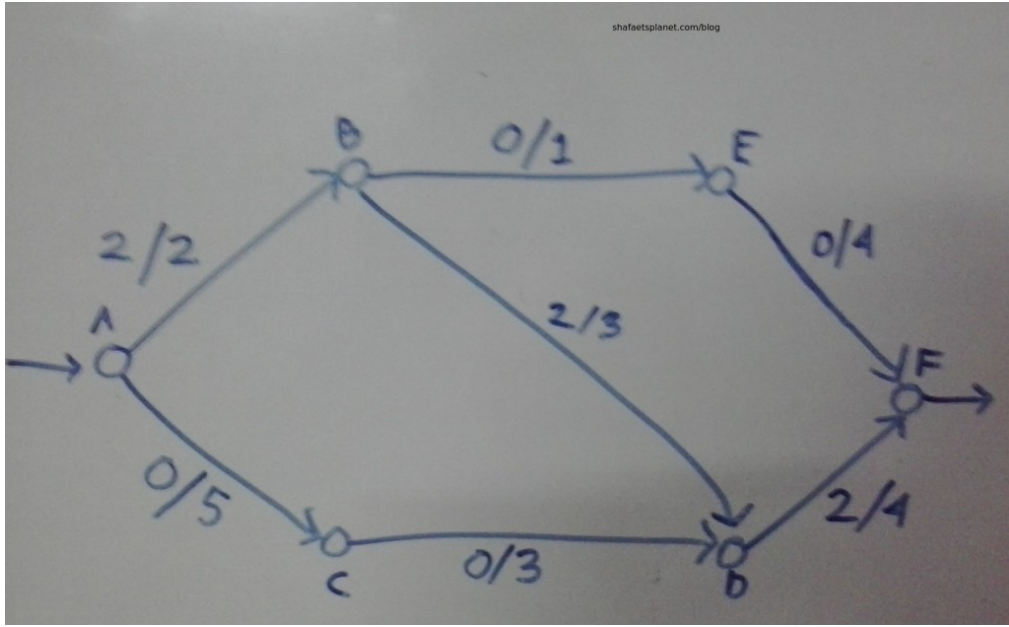
তোমাকে চিত্র-১ এর মত একটা গ্রাফ দেয়া আছে। মনে কর গ্রাফের প্রতিটা এজ একটা করে পানির পাইপ। প্রতিটা পাইপ দিয়ে প্রতি সেকেন্ডে কত লিটার পানি প্রবাহিত হতে পারবে সেটার একটা সীমা আছে যেটাকে বলা হয় পাইপের ক্যাপাসিটি। আর কোনো পাইপ দিয়ে সেকেন্ডে যতটুকু পানি যাচ্ছে সেটা হলো পাইপটার ফ্লো। প্রতিটা এজের সাথে “ফ্লো/ক্যাপাসিটি” উল্লেখ করে দেয়া হয়েছে। যেমন $A \rightarrow B$ এজ দিয়ে সেকেন্ডে সর্বোচ্চ ২লিটার পানি যেতে পারে এবং এই মুহূর্তে সেকেন্ডে পানি যাচ্ছে ০ লিটার। শুধুমাত্র A নোড দিয়ে পানি প্রবেশ করানো যায় এবং শুধুমাত্র F নোড দিয়ে পানি বের হয়ে যায়। A কে বলা হয় সোর্স এবং F হলো সিংক। তোমাকে বলতে হবে A থেকে F এ প্রতি সেকেন্ডে সর্বোচ্চ কত লিটার পানি প্রবাহিত করা যাবে? অর্থাৎ পানির “ফ্লো” সর্বোচ্চ কত হতে পারে?



মূল সমস্যা সমাধানের আগে আমরা ছোট একটা সমস্যা সমাধান করি। $A \rightarrow C \rightarrow D \rightarrow F$ এই পথটা ব্যবহার করে পানি সোর্স থেকে সিংক এ গেলে সর্বোচ্চ কত লিটার পানি প্রবাহিত হতে পারবে? ছবিতে দেখা যাচ্ছে $A \rightarrow C$ এজের ক্যাপাসিটি ৫, $C \rightarrow D$ এজের ক্যাপাসিটি ৩, $D \rightarrow F$ এজের ক্যাপাসিটি ৪। এখানে সর্বনিম্ন ক্যাপাসিটি হলো ৩, তাই আমরা কোনো সময় ৩লিটারের বেশি পানি এই পথে পাঠাতে পারবোনা।

কোনো পথের সর্বনিম্ন ক্যাপাসিটির এজ সেই পথের ফ্লো নিয়ন্ত্রণ করে। বোতলের সরু মুখের সাথে তুলনা দিয়ে এই ব্যাপারটাকে বলা হয় বোটলনেক(bottleneck)। যেমন ধরো তোমার বাসায় ৫MBps এর ইন্টারনেট কানেকশন আছে, এখন তুমি একটা মুভি ডাউনলোড করতে চাচ্ছ। এখন মুভির সার্ভার থেকে তোমার বাসায় ডাটা প্যাকেট আসার আগে অনেকগুলো নেটওয়ার্ক পার হয়ে আসে, কোনো একটা নেটওয়ার্কে গতি মাত্র 1MBps। এখন নেটওয়ার্কের বাকি অংশের গতি যতই বেশি হোক না কেন তুমি 1MBps এর বেশি গতিতে ডাউনলোড করতে পারবে না, নেটওয়ার্কের সবথেকে ধীরগতির অংশই তোমার ডাউনলোডের গতি নিয়ন্ত্রণ করবে।

এখন দেখি কিভাবে সর্বোচ্চ পানির ফ্লো পাবার সমস্যাটার সমাধান করা যায়। সমাধান খুবই সহজ, আমরা প্রতিবার যেকোনো একটা করে পথ দিয়ে পানি পাঠাতে থাকবো যতক্ষণ পাইপ ক্যাপাসিটি থাকে। কিন্তু এভাবে কি আমরা সর্বোচ্চ ফ্লো বা প্রবাহ পাবো? পাবো তবে সেজন্য একটু বুদ্ধি খাটাতে হবে। প্রথমে আমরা একটা পথে পানির ফ্লো পাঠিয়ে দেখি কি ঘটে। যেকোনো একটা পথ বেছে নিতে পারি, আমি বুঝানোর সুবিধার জন্য $A \rightarrow B \rightarrow D \rightarrow F$ পথটা বেছে নিলাম। এই পথে সর্বনিম্ন ক্যাপাসিটি হলো ২। তাহলে এই পথে ২লিটার পানির ফ্লো পাঠিয়ে দেবার পর গ্রাফটা দেখতে হবে এরকম:

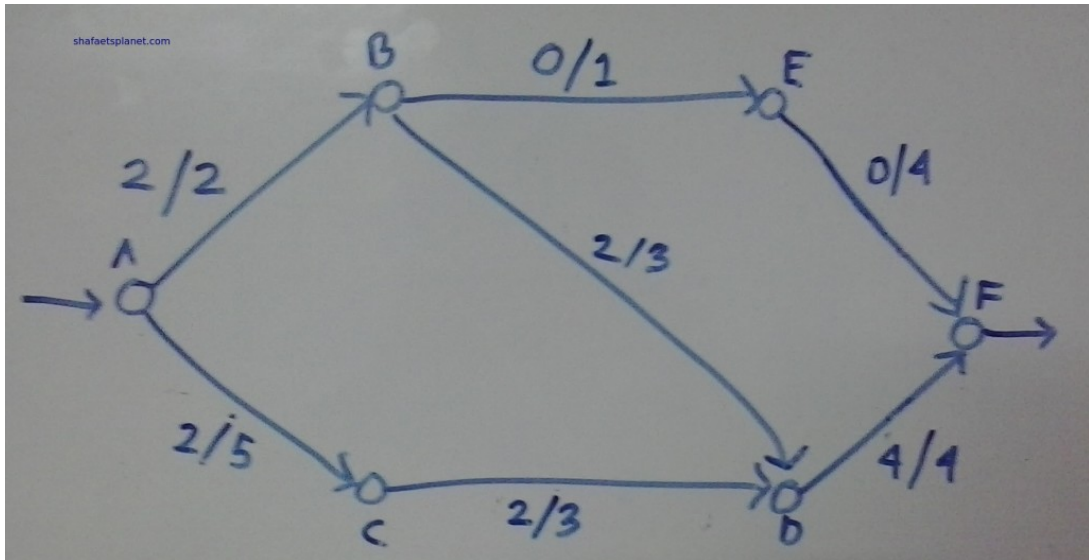


চিত্র-২: A->B->D->F পথে ফ্লো পাঠানোর পর

আমরা A->B->D->F পথের প্রতিটি এজের ফ্লো ২ বাড়িয়ে দিয়েছি। লক্ষ্য করো A->B এজের আসল ক্যাপাসিটি ২ এবং ফ্লো পাঠানো হয়েছে ২, তাই এই এজ নিয়ে আর কোনো ফ্লো পাঠানো যাবে না। তেমনি B->D এজের ক্যাপাসিটি ৩ কিন্তু ফ্লো পাঠানো হয়েছে ২, তাই এই এজটা দিয়ে আরো ৩-২=১ ফ্লো পাঠানো সম্ভব। আমরা বলতে পারি B->D এজের **residual** ক্যাপাসিটি=১, residual শব্দটার অর্থ হলো কোনো কিছু ব্যবহার করে ফেলার পর বেচে যাওয়া অংশ।

residual ক্যাপাসিটি = এজ এর ক্যাপাসিটি – ব্যবহৃত ক্যাপাসিটি বা ফ্লো এর পরিমাণ।

এবার একইভাবে A->C->D->F পথটা নির্বাচিত করি। এই পথে D->F এজের আসল ক্যাপাসিটি ৪ হলেও ২ ফ্লো আগেই পাঠানো হয়েছে, তাই বর্তমান ক্যাপাসিটি বা **residual** ক্যাপাসিটি ৪-২=২। এই পথটা দিয়ে ২ ফ্লো পাঠানো সম্ভব।



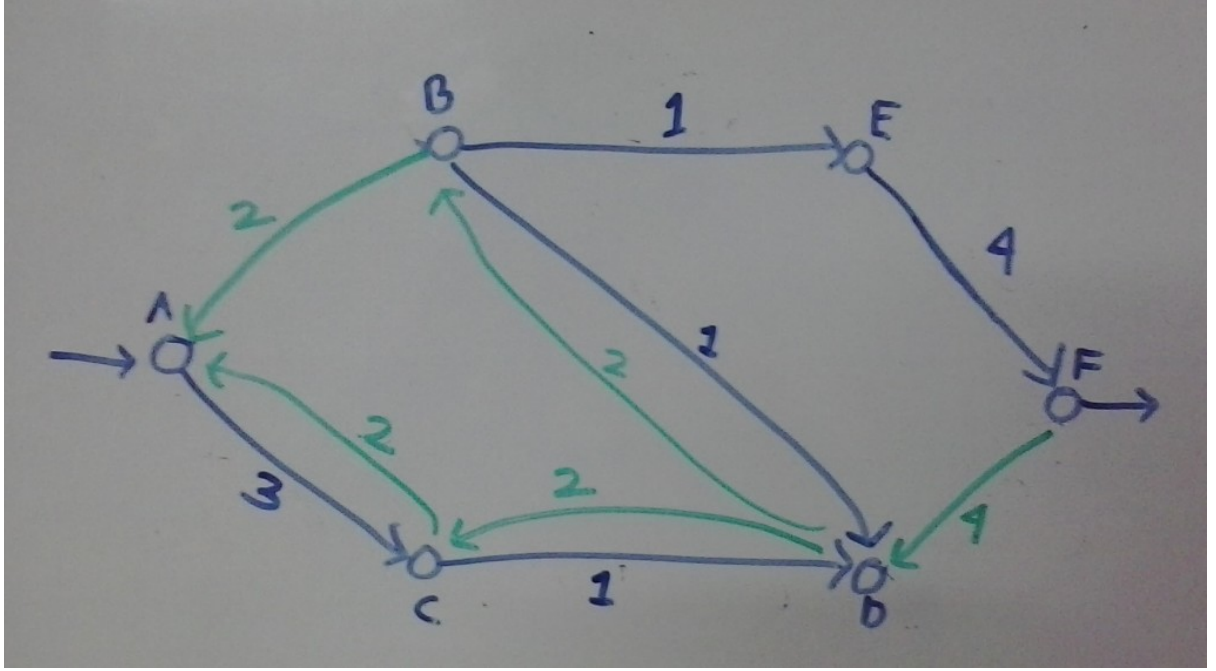
চিত্র-৩: A->C->D->F পথে ফ্লো পাঠানোর পর

আমরা এখন মোট ২+২=৪ ফ্লো পেলাম, অর্থাৎ পানি প্রবাহের হার এখন সেকেন্ডে ৪ লিটার। এই গ্রাফে কি আরো বেশি পানির ফ্লো পাঠানো সম্ভব? A->B এবং D->F এজ এরই মধ্যে সর্বোচ্চ ক্যাপাসিটিতে পৌঁছে গেছে, এই ২টা এজের কোনোটাকে না নিয়ে সিংক এ যাবার পথ গ্রাফে নেই। তাই দেখে মনে হতে পারে আর ফ্লো পাঠানো সম্ভব না। কিন্তু আমরা এখন বুদ্ধি খাটিয়ে আরো বেশি ফ্লো পাঠিয়ে দিব!

আমাদের এখন প্রতিটা এজের **residual** ক্যাপাসিটি ব্যবহার করে **residual** গ্রাফ আঁকতে হবে কিভাবে আরো ফ্লো পাঠানো সেটা বুঝতে হবে। **residual** গ্রাফ আঁকার নিয়ম হলো:

১. গ্রাফের প্রতিটা এজ (u,v) এর ক্যাপাসিটি হবে এজ টার **residual** ক্যাপাসিটির সমান।
২. প্রতি এজ (u,v) এর জন্য উল্টো এজ (v,u) এর **residual** ক্যাপাসিটি হবে (u,v) এজ এ ফ্লো এর সমান।

তারমানে কোনো এজ দিয়ে যতটুকু ফ্লো পাঠিয়েছি সেটা হবে উল্টো এজের **residual** ক্যাপাসিটি। তাহলে **residual** গ্রাফে মূল এজ এবং উল্টো এজের **residual** ক্যাপাসিটির যোগফল হবে মূল এজ এজের মোট ক্যাপাসিটির সমান। চিত্র-৪ এর গ্রাফটার **residual** গ্রাফটা হবে এরকম:



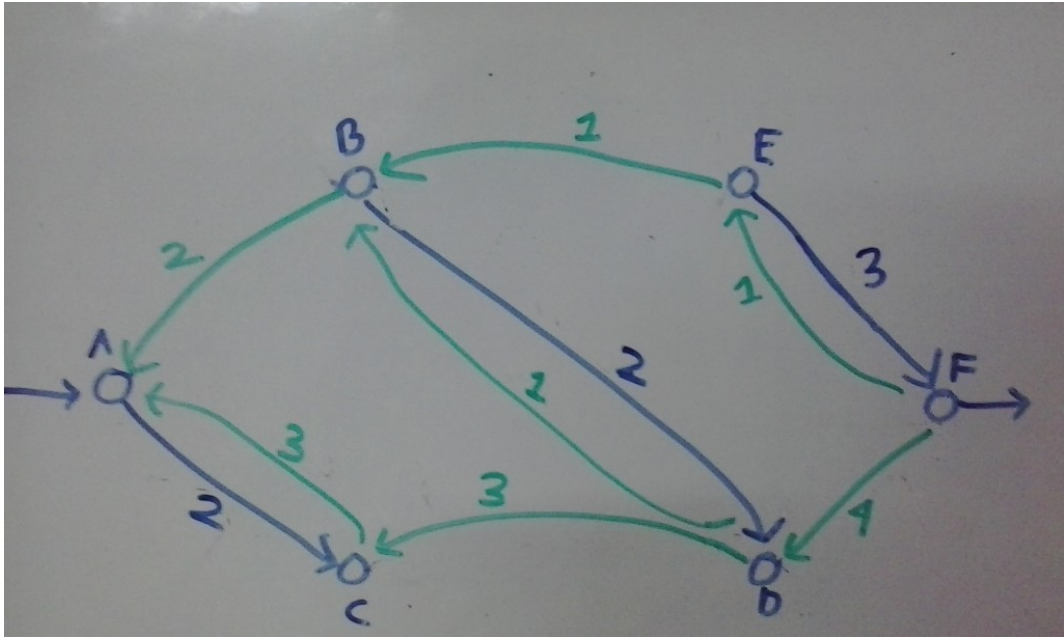
চিত্র-৪: ৩নং চিত্রের গ্রাফের **residual** গ্রাফ

সবুজ রঙ দিয়ে উল্টো এজ এবং তাদের **residual** ক্যাপাসিটি দেখানো হয়েছে। ০ ক্যাপাসিটির এজগুলোকে গ্রাফে একে দেখাইনি। $A \rightarrow C$ এজ দিয়ে আগে ২ ফ্লো পাঠানো হয়েছে বলে উল্টো এজ এর **residual** ক্যাপাসিটি ২, এবং এজটার আরো ৩ ফ্লো পাঠানোর ক্ষমতা আছে তাই মূল এজ এর **residual** ক্যাপাসিটি ৩। ঠিক এভাবে অন্যান্য এজগুলো আঁকা হয়েছে, তুমি নিজে একে যাচাই করে নিতে পারো ঠিকমত বুঝেছো নাকি বা আমার ছবিতে ভুল আছে নাকি।

এখন উল্টো এজ এ ফ্লো পাঠানোর মানে কি? আমরাতো পাইপের উল্টো দিক দিয়ে পানি পাঠাতে পারবো না বা রাস্তার উল্টো দিক দিয়ে গাড়ি চালাতে পারবো না। উল্টো দিকে ফ্লো পাঠানোর মানে হলো মূল ফ্লো টাকে বাতিল করে দেয়া! আমরা যদি $D \rightarrow B$ এজ দিয়ে ২ ফ্লো পাঠাই তার মানে $B \rightarrow D$ এজ দিয়ে আসা ২লিটার পানির প্রবাহকে আমরা বাতিল করে দিচ্ছি। তখন **residual** গ্রাফে মূল এজের **residual** ক্যাপাসিটি যাবে বেড়ে, এবং উল্টো এজের **residual** ক্যাপাসিটি যাবে কমে, যোগফল আগের মতই থাকবে!

এখন চিন্তা করে দেখো চিত্র-৪ দিয়ে আরো ফ্লো সিংক এ পাঠানো যায় নাকি। মনে রাখবে চিত্র-৩ আর চিত্র-৪ এ একই গ্রাফকেই ভিন্নভাবে আঁকা হয়েছে।

চিত্র ৪ এ দেখা যাচ্ছে $A \rightarrow C \rightarrow D \rightarrow B \rightarrow E \rightarrow F$ পথে আরো ১ ফ্লো পাঠানো সম্ভব! এই পথে $D \rightarrow B$ হলো উল্টো এজ। তারমানে $D \rightarrow B$ এজ এর ১ ফ্লো বাতিল করে দিয়ে আমরা গ্রাফে মোট ফ্লো ১ বাড়িয়ে ফেলতে পারছি। এখন **residual** গ্রাফ কিরকম হবে দেখি:



চিত্র-৫: A->C->D->B->E->F পথে ফ্লো পাঠানোর পর residual গ্রাফ

তুমি যদি এ পর্যন্ত বুঝে থাকো তাহলে চিত্র-৫ থেকে মূল গ্রাফটাও নিজে একে নিতে পারবে। চিত্র-৪ এ C->D এজ দিয়ে ১ ফ্লো পাঠানো হয়েছে। তাই চিত্র-৫ এ C->D এজের residual ক্যাপাসিটি কমে ১-১=০ হয়ে গিয়েছে এবং D->C এজের residual ক্যাপাসিটি বেড়ে ২+১=৩ হয়ে গিয়েছে। এভাবে সবগুলো এজ আঁকা হয়েছে।

এই গ্রাফে সোর্স থেকে সিংক এ পানি পাঠানোর আর কোনো পথ নেই। আমরা ৩টি পথে মোট ২+২+১=৫ ফ্লো পেয়েছি, অর্থাৎ পানি প্রবাহের হার প্রতি সেকেন্ডে ৫ লিটার। এটাই গ্রাফটার জন্য ম্যাক্সিমাম ফ্লো।

residual গ্রাফে আমরা যখন একটা পথ বের করি সেই পথের একটা বিশেষ নাম আছে, সেটা হলো **অগমেন্টেড পাথ(Augmented path)**। তাহলে আমাদের অ্যালগোরিদম খুব সহজ, যতক্ষণ সম্ভব আমরা residual graph এ একটা অগমেন্টেড পাথ খুঁজে বের করবো এবং সেই পথে ফ্লো পাঠিয়ে দিবো! এটার নাম Ford-Fulkerson অ্যালগোরিদম। এটার সুডোকোড এরকম:

Max flow pseudo-code

- 1 Input: A graph $G = (V, E)$ with flow capacity cap , source node s and sink node t .
- 2 Output: Calculate maximum flow from s to t .
- 3 Initialize:
 - 4 1. $total_flow = 0$
 - 5 2. Residual Capacity $C_f(u, v) = cap(u, v)$ for each edge (u, v) in the graph
- 6 Algorithm:
 - 7 1. While there is a path from s to t such that $C_f[u][v] > 0$ for all edges (u, v) in path:
 - 8 1.1. min_res_cap = minimum residual capacity among all the edges (u, v) in the path.
 - 9 1.2. For each edge (u, v) in the path:
 - 10 $C_f(u, v) = C_f(u, v) - min_res_cap$
 - 11 $C_f(u, v) = C_f(u, v) + min_res_cap$
 - 12 1.3. $total_flow = total_flow + min_res_cap$
 - 13 3. Return $total_flow$

সহজ কথায় একটা ২-ডি অ্যারে $C_f[u][v]$ তে প্রতিটা এজের residual ক্যাপাসিটি থাকবে, যে এজগুলো গ্রাফে নাই সে এজের residual ক্যাপাসিটি ০। এখন তুমি সোর্স থেকে সিংক এ যাবার এমন একটা পথ বের করবে যে পথের প্রতিটি এজের residual ক্যাপাসিটি ০ থেকে বড়। পথের এজগুলোর মধ্যে মিনিমাম residual ক্যাপাসিটির মান খুঁজে বের করবে। ফ্লো পাঠানোর পর প্রতিটা এজের residual ক্যাপাসিটি কমে যাবে এবং উল্টা এজের residual ক্যাপাসিটি বেড়ে যাবে। মিনিমাম residual ক্যাপাসিটিটা মোট ফ্লো এর মানের সাথে যোগ হতে থাকবে। সবশেষে চাইলে তুমি আসল ক্যাপাসিটি থেকে residual capacity বিয়োগ করে কোনো এজ এ ফ্লো এর পরিমাণ নির্ণয় করতে পারো।

গ্রাফটা ডিরেক্টেড হতে হবে এমন কোনো কথা নেই। বাইডিরেকশনাল এজও থাকতে পারে। A-B বাইডিরেকশনাল এজের ক্যাপাসিটি ১০ হলে $C_f[A][B] = C_f[B][A] = ১০$ হবে। এখন আগের মতই অগমেন্টেড পাথ বের করে ম্যাক্সিমাম ফ্লো বের করতে পারবে। তবে এক্ষেত্রে কোন নোডে ফ্লো কত হচ্ছে সেটা বের করতে হলে তোমাকে আরেকটু বুদ্ধি খাটাতে হবে।

অ্যালগোরিদমে ২নম্বর ধাপে পাথ বের করার জন্য [বিএফএস/ডিএফএস/বেলম্যান ফোর্ড](#) ইত্যাদি অ্যালগোরিদম ব্যবহার করা যেতে পারে। প্রবলেমের ধরণ অনুযায়ী এই ধাপে একেক অ্যালগোরিদম একেক ধরণের সুবিধা দিবে, সেগুলো তুমি প্রবলেম সলভ করতে করতে জানতে পারবে। আর পাথ বের করার জন্য বিএফএস বা যেটাই ব্যবহার করোনা কেন, প্রতিটা নোডে যাবার সময় নোডটার প্যারেণ্ট কে সেটা মনে রাখতে হবে কোনো একটা অ্যারেতে।

তুমি যদি বিএফএস ব্যবহার করে সমাধান করো তাহলে সেটাকে বলা হয় **এডমন্ড কার্প** অ্যালগোরিদম। ফোর্ড-ফুলকার্সন অ্যালগোরিদমে শুধু পাথ খুঁজে বের করার কথা বলে হয়েছে, সেটা যেকোনোভাবে বের করা যেতে পারে, আর এডমন্ড-কার্প বিএফএস ব্যবহার করে কাজটা করতে বলা হয়েছে অর্থাৎ ফোর্ড-ফুলকার্সন ইমপ্লিমেন্ট করার একটা উপায় হলো এডমন্ড কার্প অ্যালগোরিদম যার কমপ্লেক্সিটি $O(VE^2)$ ।

এখন তোমাদের জন্য চিন্তা করার মত কয়েকটা প্রশ্ন:

প্রশ্ন ১: আমাদের প্রবলেমে সোর্স এবং সিংক ছিলো একটা। কিন্তু গ্রাফে একাধিক নোড দিয়ে পানি প্রবেশ করলে এবং একাধিক নোড দিয়ে পানি বের হয়ে গেলে কিভাবে অ্যালগোরিদমটা পরিবর্তন করবে?

প্রশ্ন ২: যদি প্রতিটা নোডের কিছু ক্যাপাসিটি থাকে, অর্থাৎ একটা নোড দিয়ে কত সর্বোচ্চ ফ্লো পাঠানো যাবে সেটা নির্দিষ্ট করা থাকে তাহলে কিভাবে সমাধান করবে?

প্রশ্ন ৩: দুটি নোডের মধ্যে একাধিক এজ থাকলে কি করবে?

প্রশ্ন ৪: দুই বন্ধু একই নোড থেকে যাত্রা শুরু করে একই গন্তব্যে পৌঁছাতে চায় কিন্তু দুইজনেই চায় ভিন্ন ভিন্ন রাস্তা ব্যবহার করে যেতে, তারমানে একই এজ কখনো ২জন ব্যবহার করতে পারবে না। গ্রাফটি দেয়া হলে তুমি কি বলতে পারবে এরকম ২টি পথ আছে নাকি? এ ধরণের পথকে **এজ ডিসজয়েন্ট পাথ** বলে।

এই প্রশ্নগুলো নিয়ে আলোচনা করা হয়েছে [পরের পর্বে](#)।

প্রোগ্রামিং কনটেক্সটে ম্যাক্সিমাম ফ্লো প্রবলেম এ কোডিং এর থেকে অনেক কঠিন হলো গ্রাফটা কিভাবে বানাতে হবে সেটা বের করা, তাই অনেক সমস্যা সমাধান করে প্র্যাকটিস করতে হবে। শুরু করার জন্য কয়েকটা প্রবলেম দিয়ে দিলাম:

http://lightoj.com/volume_showproblem.php?problem=1153

http://lightoj.com/volume_showproblem.php?problem=1155

<http://uva.onlinejudge.org/external/110/11045.html>

হ্যাপি কোডিং!

(এখন থেকে কোনো লেখায় সি++/জাভায় সোর্স-কোড দেয়া হবে না এবং আগের লেখাগুলোর সি++ সোর্স-কোডগুলোও ধীরে ধীরে সরিয়ে নেয়া হবে, তুমি যদি অ্যালগোরিদমটা বুঝে থাকো তাহলে নিজেই কোড লিখতে পারবে। আর বুঝতে সমস্যা হলে বা বিশেষ কোনো কারণে কোড চাইলে সরাসরি আমার সাথে [যোগাযোগ কর](#))

গ্রাফ থিওরিতে হাতেখড়ি ৫: মিনিমাম স্প্যানিং ট্রি(প্রিমস অ্যালগোরিদম)

(সিরিজের অন্যান্য পোস্ট)

মিনিমাম স্প্যানিং ট্রি অনেক ধরনের প্রবলেম সলভ করতে কাজে লাগে। প্রথমে আমরা জানবো “মিনিমাম স্প্যানিং ট্রি” কি, তারপরে দেখবো প্রিমস অ্যালগোরিদম।

একটি গ্রাফ থেকে কয়েকটি নোড আর এজ নিয়ে নতুন একটি গ্রাফ তৈরি করা হলে সেটাকে বলা হয় সাবগ্রাফ। স্প্যানিং ট্রি হলো এমন একটি সাবগ্রাফ যেটায়:

* মূল গ্রাফের সবগুলো নোড আছে।

* সাবগ্রাফটি একটি ট্রি। ট্রিতে কখনো সাইকেল থাকেনা, edge থাকে $n-1$ টি যেখানে n হলো নোড সংখ্যা।

একটি গ্রাফের অনেকগুলো স্প্যানিং ট্রি থাকতে পারে, যে ট্রি এর এজ গুলোর কস্ট/ওয়েট এর যোগফল সব থেকে কম সেটাই মিনিমাম স্প্যানিং ট্রি। (n টি নোডকে সবমিলে $n-1$ টি edge দিয়ে একসাথে যুক্ত করা সম্ভব, তাই $n-1$ টির বেশি এজ আমরা নিবোনা)

মনে করি নিচের গ্রাফের প্রতিটি নোড হলো একটি করে বাড়ি। আমাদের বাড়িগুলোর মধ্যে টেলিফোন লাইন বসাতে হবে। আমরা চাই সবথেকে কম খরচে লাইন বসাতে। এজ গুলোর ওয়েট লাইন বসানোর খরচ নির্দেশ করে:

আমরা অনেক ভাবে লাইন বসাতে পারতাম। ছবিতে লাল এজ দিয়ে টেলিফোন লাইন বসানোর একটি উপায় দেখানো হয়েছে। টেলিফোন লাইনগুলো একটি সাবগ্রাফ তৈরি করেছে যেটায় অবশ্যই $n-1$ টি এজ আছে, কোনো সাইকেল নেই কারণ অতিরিক্ত এজ বসালে আমাদের খরচ বাড়বে, কোনো লাভ হবেনা। মিনিমাম স্প্যানিং ট্রি বের করার সময় আমরা এমন ভাবে এজগুলো নিবো যেন তাদের এজ এর যোগফল মিনিমাইজ হয়।

এখন নিচের গ্রাফ থেকে কিভাবে আমরা mst বের করব?

গ্রিডি(greedy) অ্যাপ্রোচে খুব সহজে মিনিমাম স্প্যানিং ট্রি বের করা যায়। আমরা এখন প্রিমস অ্যালগোরিদম কিভাবে কাজ করে দেখব। তুমি যদি আগে ক্রসকাল শিখতে চাও তাহলেও সমস্যা নেই, [সরাসরি পরের পর্বে](#) চলে যেতে পারো।

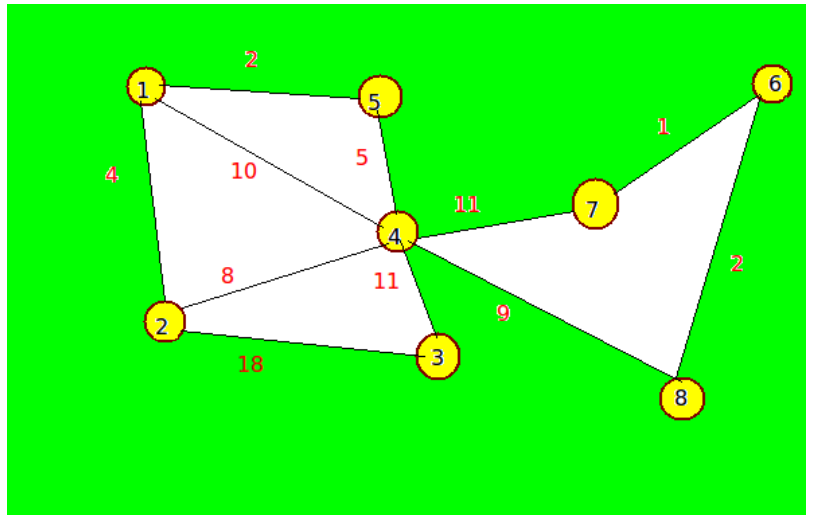
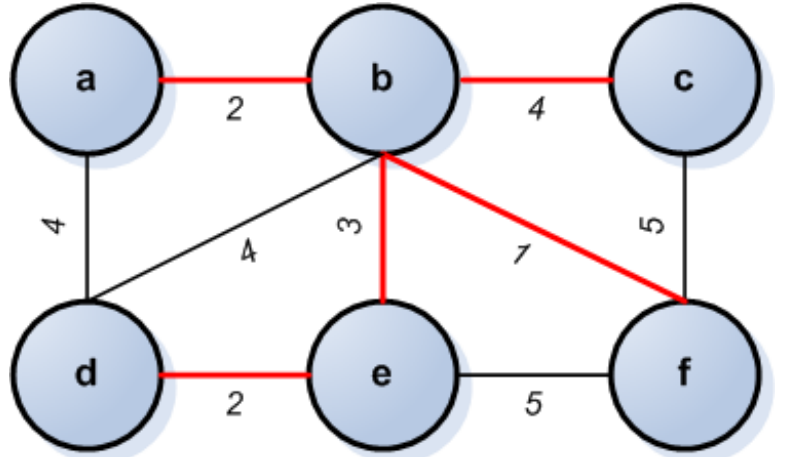
আমরা প্রথমে যেকোনো একটি সোর্স নোড নিব। ধরি সোর্স হলো ১। ১ থেকে যতগুলো এজ আছে সেগুলোর মিনিমাম টিকে আমরা সাবগ্রাফে যোগ করব। নিচের ছবিতে নীল এজ দিয়ে বুঝানো হচ্ছে এজটি সাবগ্রাফে যুক্ত করা হয়েছে:

এবার সোর্স ১ এবং ৫ নম্বর নোড থেকে মোট যত এজ আছে(আগের এজগুলো সহ) তাদের মধ্যে মিনিমাম টি নিব:

এবার নিব ১, ২ এবং ৫ নম্বর নোড থেকে মোট যত এজ আছে(আগের এজগুলো সহ) তাদের মধ্যে মিনিমাম:

পরের ধাপটি গুরুত্বপূর্ণ। ১, ২, ৫, ৪ থেকে যত এজ আছে তাদের মধ্য মিনিমাম হলো ২-৪, কিন্তু ২ নম্বর নোড এবং ৪

নম্বর নোড দুইটাই অলরেডি সাবগ্রাফের অংশ, তারা আগে থেকেই কানেক্টেড, এদের যোগ করলে সাবগ্রাফে সাইকেল তৈরি হবে, তাই ২-৪ এজটি নিয়ে আমাদের কোনো লাভ হবেনা। আমরা এমন প্রতিবার এজ নিব যেন নতুন আরেকটি নোড সাবগ্রাফে যুক্ত হয়। তাহলে ৪-৮ হবে আমাদের পরের



চয়েস।

এভাবে শেষ পর্যন্ত আমরা পাবো:

নীলরং এর এই সাবগ্রাফটাই আমাদের মিনিমাম স্প্যানিং ট্রি।
বাকি এজগুলো মুছে দিলে থাকে:

তাহলে টেলিফোন লাইন বসাতো মোট খরচ:

$8+2+5+11+18+2+1=57$ । একটি গ্রাফে অনেকগুলো
মিনিমাম স্প্যানিং ট্রি থাকতে পারে, তাই অনলাইন জাজে
mst সংক্রান্ত প্রবলেম অনেক ক্ষেত্রেই special judge দিয়ে
কাজ করে, অর্থাৎ সঠিক আউটপুটগুলোর যেকোনো একটা
প্রিন্ট করলেই চলে।

আমাদের সুডোকোড হবে এরকম:

[text]

* Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).

* Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V , $E_{new} = \{\}$

* Repeat until $V_{new} = V$:

o Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)

o Add v to V_{new} , and (u, v) to E_{new}

* Output: V_{new} and E_{new} describe a minimal spanning tree

(Wiki)

[/text]

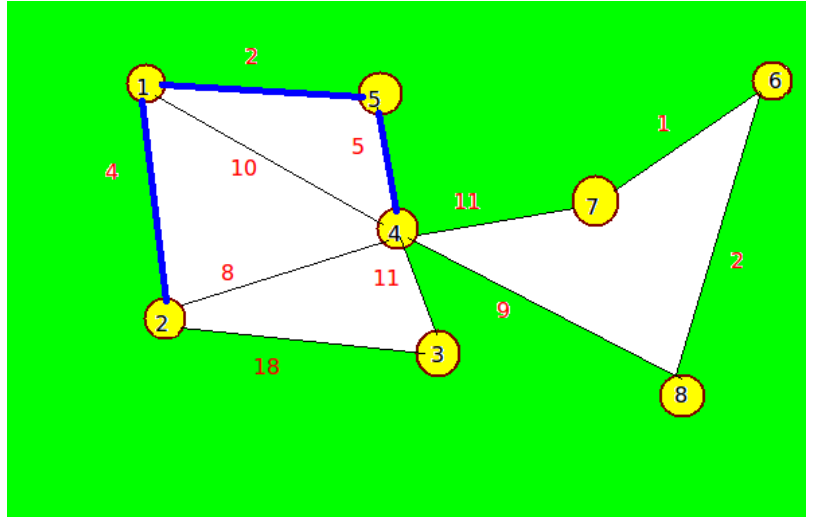
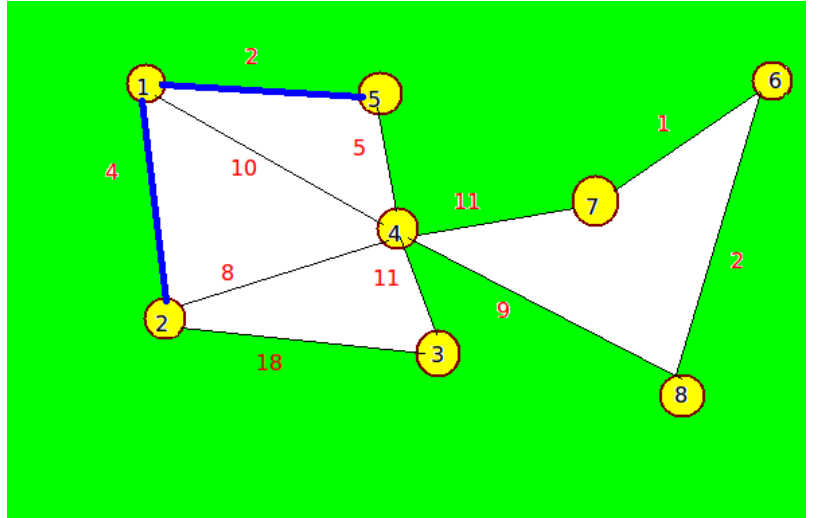
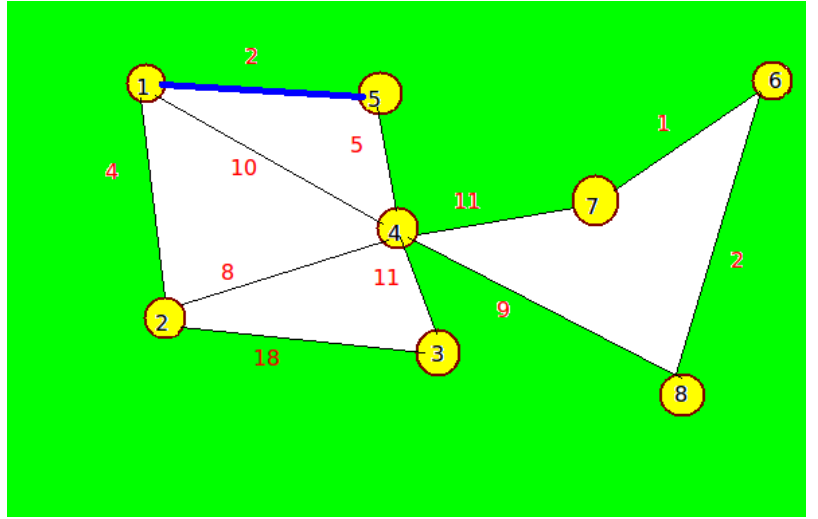
এখন মাথায় প্রশ্ন আসতে পারে কি ভাবে প্রিমস অ্যালগোরিদম ইম্প্লিমেন্ট করব? বারবার লুপ চালিয়ে নেইড অ্যাপ্রোচে কোড লিখলে তোমার কোড টাইম লিমিটের মধ্যে রান না করার সম্ভাবনাই বেশি।

আমাদের সমস্যার সমাধান হলো ‘প্রায়োরিটি কিউ’। আশা করি এই ডাটা স্ট্রাকচারটি সম্পর্কে সবাই জানে। প্রতিটি নোড ডিজিট করার সাথে সাথে নতুন এজগুলোকে ‘প্রায়োরিটি কিউ’তে ঢুকিয়ে রাখবে। প্রায়োরিটি কিউতে এজগুলোকে ওয়েট অনুযায়ী সর্টড রাখবে। তাহলে প্রতিবার কিউ এর টপ এ পাবে এখন পর্যন্ত নেয়া নোডগুলো থেকে বের হওয়া সবথেকে কম ওয়েট এর এজ, সেটাকে সাবগ্রাফে যোগ করবে।

নিজে [priority queue](#) না বানিয়ে `stl` বা `java`

`collections` ব্যবহার কর। সি++ এ [অপারেটর ওভারলোডিং](#) করে `priority queue` সর্ট করা সব থেকে সহজ। আমরা প্রথমে ডাটা নামক একটি `structure` ডিক্লেয়ার করবো:

C++



```

1 struct data
2 { int u,v,cost;
3   bool operator < ( const data& p ) const
4 { //overloading operator
5   return cost > p.cost;
6 }
};

```

এরপর **priority queue** ডিক্লেয়ার করে সেটায় 'data' টাইপের ভ্যারিয়েবল পুশ করলে কিউ তে অটোমেটিক সর্ট হয়ে যাবে। যেহেতু এটা অ্যালগোরিদমের টিউটোরিয়াল, সি++ এর না, তাই এসব নিয়ে আর বেশি লিখবোনা, কোথায় সমস্যা হলে মন্তব্য অংশে বা আমার সাথে যোগাযোগ করে জানাতে পারো, আর তার আগে [এই লিংকটা](#) একবার দেখো।

mst বের করার জন্য আরেকটি অ্যালগোরিদম আছে যা **ক্রসকাল অ্যালগোরিদম** নামে পরিচিত যা কন্টেস্টেন্টদের মধ্যে বেশি জনপ্রিয়। **kruskal** শিখতে হলে তোমাকে ডিসজয়েন্ট সেট নামের একটি চমককার ডাটা স্ট্রাকচার এর সাথে অবশ্যই পরিচিত হতে হবে। তারপর ক্রসকাল অ্যালগোরিদম শেখা খুব সহজ হবে, প্রিমস শিখলেও ক্রসকাল অবশ্যই শিখতে হবে। ডিসজয়েন্ট সেট শেখার জন্য নিচের টিউটোরিয়ালগুলো দেখো:

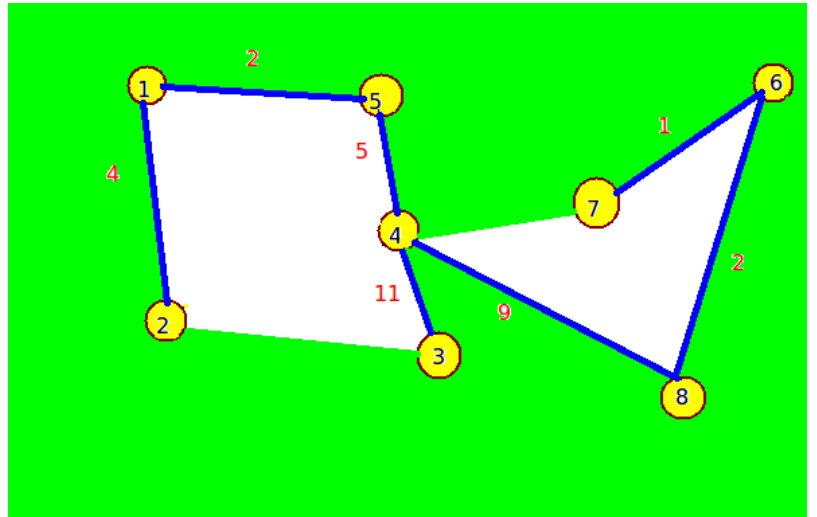
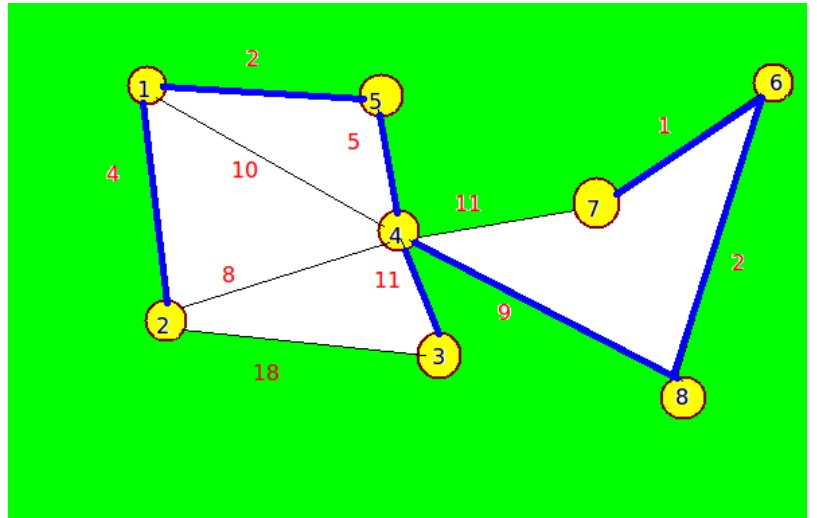
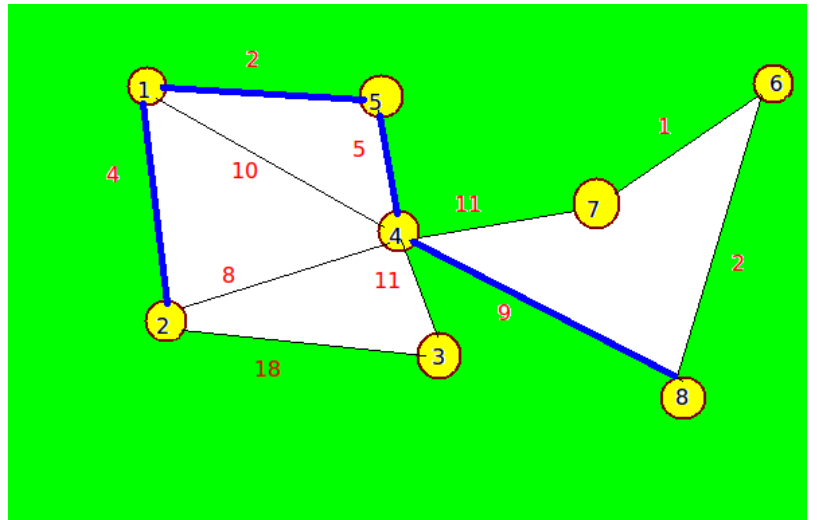
[আমার লেখা টিউটোরিয়াল](#)

[topcoder tutorial](#)

[Wikipedia](#)

[Coreman's introduction to algorithm](#)

অ্যালগোরিদমটা ইমপ্লিমেন্ট করার পর অবশ্যই নিচের সমস্যা গুলো সমাধানের চেষ্টা করবে, আবারো বলবো প্রবলেম সলভ না করলে অ্যালগোরিদম শেখা অর্থহীন কারণ মাথা খাটিয়ে প্রবলেম সলভিং হলো স্কিল ডেভেলপ করার সব থেকে ভালো উপায়।



<http://uva.onlinejudge.org/external/5/544.html>(Straight forward)

<http://uva.onlinejudge.org/external/9/908.html>

<http://uva.onlinejudge.org/external/100/10034.html>(Straight forward)

<http://uva.onlinejudge.org/external/112/11228.html>

<http://uva.onlinejudge.org/external/104/10462.html>(2nd best mst)

spoj:

<http://www.spoj.pl/problems/MST/>(Straight forward)

গ্রাফ থিওরিতে হাতেখড়ি ৬: মিনিমাম স্প্যানিং ট্রি(ক্রসকাল অ্যালগোরিদম)

(অন্যান্য পোস্ট)

আগের পোস্টে আমরা প্রিমস অ্যালগোরিদম ব্যবহার করে **mst** নির্ণয় করা দেখেছি। **mst** কাকে বলে সেটাও আগের পোস্টে বলা হয়েছে। এ পোস্টে আমরা দেখবো **mst** বের করার আরেকটি অ্যালগোরিদম যা ক্রসকালের অ্যালগোরিদম নামে পরিচিত। এটি **mst** বের করার সবথেকে সহজ অ্যালগোরিদম। তবে তোমাকে অবশ্যই ডিসজয়েন্ট সেট ডাটা স্ট্রাকচার সম্পর্কে জানতে হবে,না জানলে এই পোস্টটি অবশ্যই দেখে আসো।

এই পোস্টে নিজের আকা ছবি ব্যবহার করবোনা। উইকিতে ক্রসকাল নিয়ে খুব সুন্দর করে লেখা আছে,আমি ওখানকার ছবিগুলোই ব্যবহার করে সংক্ষেপে অ্যালগোরিদমটা বুঝানোর চেষ্টা করবো।

নিচের গ্রাফটি দেখো:

প্রথমে আমাদের ট্রিতে একটি এজও নেই। আমরা মূল গ্রাফের এজগুলোকে **cost** অনুযায়ী সর্ট করে ফেলবো। সব থেকে কম **cost** এর এজ আগে নিবো,বেশি **cost** এর এজ পরে নিবো। দুটি এজের **cost** সমান হলে যেকোনো একটি আগে নিতে পারি। তারপর একটি করে এজ নিবো আর দেখবো এজের দু প্রান্তের নোডগুলোর মধ্যে ইতোমধ্যে কোনো পথ আছে নাকি,যদি থাকে তাহলে এজটি নিলে সাইকেল তৈরি হবে,তাই এজটা আমরা নিবোনা। বুঝতেই পারছো প্রিমসের মত এটিও একটি 'গ্রিডি' অ্যালগোরিদম।

উপরে AD আর CE হলো সবথেকে কম **cost** এর এজ। আমরা AD কে সাবগ্রাফের অন্তর্ভুক্ত করলাম।

একই ভাবে এরপ CE তারপর DF,AB এবং BE কে যোগ করবো:

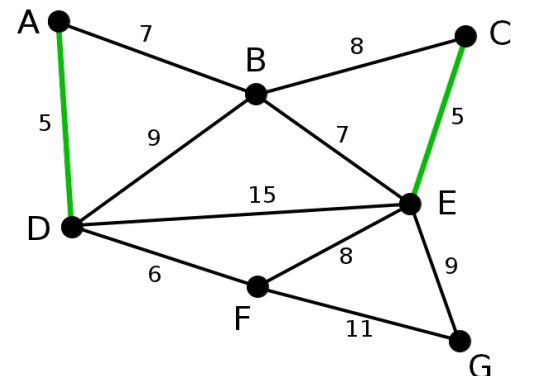
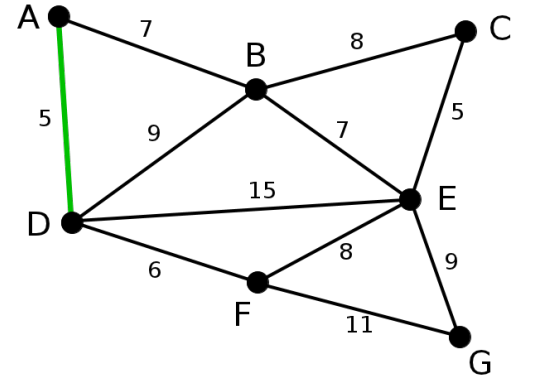
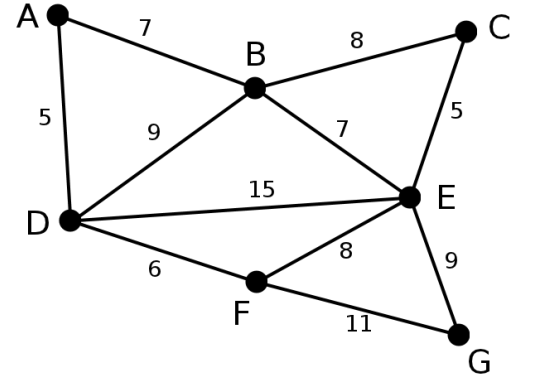
এরপর সবথেকে ছোট্ট এজ হলো EF, এটাকে আমরা নিতে পারবোনা কারণ EF নিলে একটি সাইকেল তৈরি হয়ে যাবে,E থেকে F তে যাবার রাস্তা আগে থেকেই আছে,তাই এজটি নেয়ার কোনো দরকার নেই। এভাবে BC,DB সহ লাল রঙের এজগুলো বাদ পড়বে কারণ এরা সাইকেল তৈরি করে।

সবশেষে EG যোগ করলে আমরা **mst** পেয়ে যাবো।

এখন আমরা ইমপ্লিমেন্টেশনে আসি। আমাদের প্রথম কাজ হলো সর্ট করা। পরের কাজ হলো একটি একটি এজ নিয়ে চেক করা যে দু প্রান্তের নোড দুটির মধ্য পথ আছে নাকি,অর্থাৎ তারা একই কম্পোনেন্টের ভিতর আছে নাকি। এটা চেক করতে লাগবে ডিসজয়েন্ট সেট। ডিসজয়েন্ট সেট নিয়ে [টিউটোরিয়ালে](#) দেখিয়েছিলাম কিভাবে দুটি নোড একই সাবগ্রাফে আছে নাকি বের করতে হয়। তুমি সেই কাজটিই এখানে করবে। তারপর একই সাবগ্রাফে না থাকলে আগের মত Union ফাংশন কল দিয়ে তাদের একসাথে নিয়ে আসবে আর এজটি একটি ভেক্টর বা অ্যারেতে সেভ করে রাখবে।

নিচে একটা ইমপ্লিমেন্টেশন দিলাম, আশা করি এটা কপি না করে নিজে বুঝে লিখবে:

C++



```

1  struct edge
2  {
3      int u,v,w;
4      bool operator < ( const edge& p ) const
5      {
6          return w < p.w;
7      }
8  };
9  int pr[MAXN];
10 vector<edge>e;
11 int find(int r)
12 {
13     return (pr[r]==r) ? r: find(pr[r]);
14 }
15 int mst(int n)
16 {
17     sort(e.begin(),e.end());
18     for(int i=1;i<=n;i++)pr[i]=i;
19
20     int count=0,s=0;
21     for(int i=0;i<(int)e.size();i++)
22     {
23         int u=find(e[i].u);
24         int v=find(e[i].v);
25         if(u!=v)
26         {
27             pr[u]=v;
28             count++;
29             s+=e[i].w;
30             if(count==n-1) break;
31         }
32     }
33     return s;
34 }
35
36 int main(){
37     // READ("in");
38     int n,m;
39     cin>>n>>m;
40     for(int i=1;i<=m;i++)
41     {
42         int u,v,w;
43         cin>>u>>v>>w;
44         edge get;
45         get.u=u; get.v=v; get.w=w;
46         e.push_back(get);
47     }
48     cout<<mst(n)<<endl;
49     return 0;
50 }
51 }

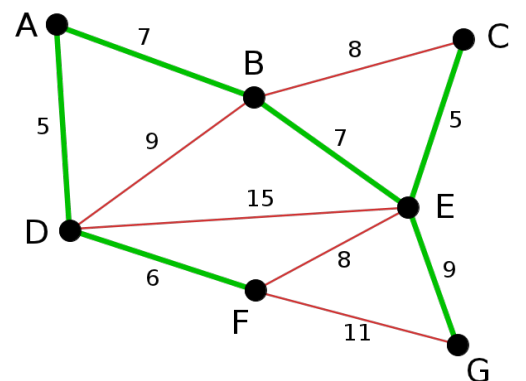
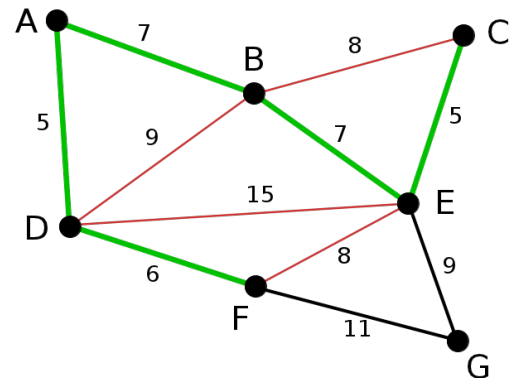
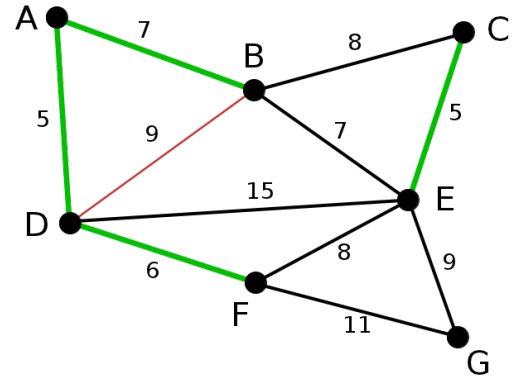
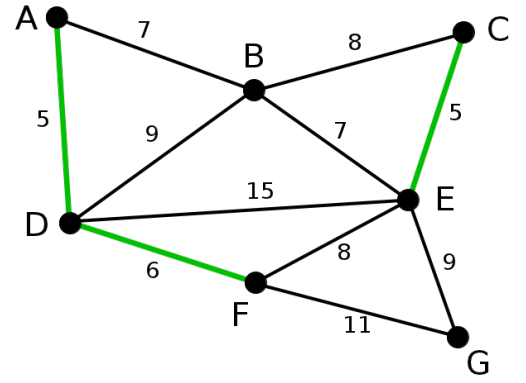
```

কমপ্লেক্সিটি অ্যানালাইসিস:

মনে করি E হলো এজ সংখ্যা। এজগুলোকে সর্ট করতে হবে, সেটার কমপ্লেক্সিটি $E \log E$, এরপরে শুধু এজগুলোর উপর লিনিয়ার লুপ চালাতে হবে। তাহলে মোট কমপ্লেক্সিটি $O(E \log E)$ ।

mst সম্পর্কিত অনেকগুলো সহজ প্রবলেম দিয়েছি প্রিমস এর টিউটোরিয়ালে, ওগুলো সমাধান করে প্র্যাকটিস করতে পারো। আরেকটু ভালো প্রবলেম করতে চাইলে দেখো:

২য় সেরা স্প্যানিং ট্রি?



অনেক সময় প্রবলেমে বলা হয় সেকেন্ড বেস্ট MST বের করতে। এটা আমরা ব্রুট ফোর্স দিয়ে বের করতে পারি। MST বের করা পর যে এজগুলো পাবো সেগুলার প্রত্যেকটা একবার করে বাদ দিয়ে নতুন করে MST বের করতে হবে, এভাবে করে যে MST টা মিনিমাম হবে সেটাই সেকেন্ড বেস্ট MST।

<http://uva.onlinejudge.org/external/103/10369.html>

<http://uva.onlinejudge.org/external/117/11733.html>

গ্রাফ থিওরিতে হাতেখড়ি ৭: টপোলজিকাল সর্ট

(অন্যান্য পর্ব)

মনে কর তোমার হাতে কিছু কাজের একটা তালিকা আছে, কাজগুলো অবশ্যই শেষ করতে হবে। কাজগুলো হলো অফিসে যাওয়া, সকালে নাস্তা করা, টিভিতে খেলা দেখা, কিছু ই-মেইলের উত্তর দেয়া, বন্ধুদের সাথে ডিনার করা ইত্যাদি। কাজগুলো কিন্তু আপনি যেকোনো অর্ডারে করতে পারবেনা, কিছু শর্ত মানতে হবে। যেমন অফিসে যাবার আগে নাস্তা করতে হবে, খেলা দেখার আগে অফিসে যেতে হবে, ডিনারে বসার আগে ইমেইলের উত্তর দিতে হবে।

তুমি শর্তগুলোর তালিকা করে ফেললে:

১. সকালের নাস্তা → অফিস (ক → খ এর মানে হলো ‘খ’ কাজটি করার আগে ‘ক’ কাজটি করতে হবে)
২. সুট-টাই পড়া → অফিস
৩. অফিস → ইমেইল
৪. অফিস → ডিনার
৫. অফিস → খেলা
৬. ইমেইল → ডিনার
৭. ইমেইল → খেলা
৮. ডিনার → খেলা

তুমি এখন কোন কাজ কখন করবে? উল্টাপাল্টা অর্ডারে করলে তোমার কাজ ভুল হয়ে যাবে, ইমেইল না করে খেলা দেখতে বসলে তুমি ক্ল্যায়েন্ট হারাবে, তাই অর্ডারিং খুব জরুরি।

এটা একটি “টাস্ক শিডিউলিং” প্রবলেম। কোন কাজের পর কোন কাজ করতে হবে সেটা আমাদের বের করতে হবে। অর্থাৎ এটা এক ধরনের সার্টিং যাকে টপোলজিকাল সার্টিং বলে। আমরা এ টিউটোরিয়ালে এজ সরিয়ে বা ইনডিগ্রী কমিয়ে টপসর্ট বের করবো।

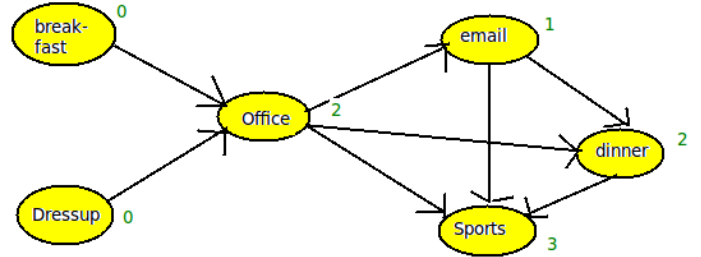
আমরা প্রথমেই সমস্যাটাকে নিচের গ্রাফ দিয়ে মডেলিং করবো:

উপরের ছবিতে প্রতিটা কাজ একটি করে নোড দিয়ে দেখানো হয়েছে। ব্রেকফাস্ট থেকে অফিসের দিকে তীরচিহ্ন দিয়ে বুঝানো হচ্ছে যে অফিসে আসার আগে ব্রেকফাস্ট করতে হবে। উপরের ৮টি শর্ত ছবিতে ৮টি ডিরেক্টেড এজ দিয়ে দেখানো হয়েছে।

প্রতিটি নোডের পাশে ছোট করে কিছু সংখ্যা দেখতে পাচ্ছে। যেমন অফিসের সাথে ০, ডিনারের সাথে ২ ইত্যাদি। এগুলো দিয়ে বুঝাচ্ছে একটি কাজ অন্য কয়টি কাজের উপর নির্ভরশীল। যেমন ডিনারের আগে তোমাকে অফিস, ইমেইল এই ২টা কাজ করতে হবে, ডিনার নোডটিতে ২টি তীরচিহ্ন প্রবেশ

করেছে, আর আমরা পাশে লিখে দিয়েছি “২”। ঠিক এভাবে ইমেইলের পাশে লেখা হয়েছে ১। এ সংখ্যাগুলোকে **indegree** বলা হয়।

লক্ষ্য কর ব্রেকফাস্ট এবং ড্রেসআপ কোনো কাজের উপর নির্ভরশীল নয়, তাই তাদের পাশে ০ লেখা হয়েছে। তারমানে আমরা এ দুটি কাজের যেকোনোটা দিয়ে দিন শুরু করতে পারি। মনে করি তুমি নাস্তা আগে খেতে চাও। নাস্তা খেয়ে নেবার পর যেসব কাজ ব্রেকফাস্টের উপর নির্ভরশীল ছিল তারা আর সেটার উপর নির্ভরশীল থাকলোনা, গ্রাফটা হয়ে গেল এরকম:



ব্রেকফাস্ট থেকে অফিসের তীরচিহ্ন সরিয়ে দিয়েছি। এখন অফিস আর মাত্র ১টি কাজের উপর নির্ভরশীল(আগে ছিল ২টির উপর)। এবার তোমাকে ড্রেসআপ করতে হবে, কারণ এখন একমাত্র এই কাজটিই কারো উপর নির্ভরশীল নয়:

ড্রেসআপ থেকে অফিসের তীরচিহ্ন সরিয়ে দেয়া হয়েছে, আর কোনো কাজ নেই, এবার তুমি অফিসে যাবার জন্য প্রস্তুত। অফিসে যাবার উপর যারা নির্ভরশীল তাদের তীরচিহ্নগুলো এখন সরিয়ে দিতে পারি:

এখন ইমেইল “নোড” এর নির্ভরশীলতা ০ হয়ে গিয়েছে:

সবশেষে ডিনার করে খেলা দেখতে বসা:

এখন তুমি কাজের অর্ডারিং পেয়ে গিয়েছো, নাস্তা করা, অফিসের পোশাক পড়া, অফিসে যাওয়া, ইমেইল করা, ডিনার করা, খেলা দেখা।

এটাকেই বলা হয় টপোলজিক্যাল সর্ট(topological sort) বা টপসর্ট। মূলত কাজের অর্ডারিং খুঁজে বের করতে এই অ্যালগোরিদমটি ব্যবহার করা হয়। কম্পিউটার তার অভ্যন্তরে বিভিন্ন কাজের অর্ডারিং ঠিক করতে টপসর্ট ব্যবহার করে। অনেক রিয়েল লাইফ প্রয়োগ থাকায় টপসর্ট কম্পিউটার সায়েন্সে খুবই গুরুত্বপূর্ণ একটি টপিক। টপসর্ট বের করার আরেকটি পদ্ধতি আছে যার নাম “ডেপথ ফার্স্ট সার্চ”, এটা নিয়ে আলোচনা করেছি এই লেখায়।

উপরের অ্যালগোরিদমটি $O(n^2)$ এ কাজ করে। একটি গ্রাফের অনেকগুলো সর্টেড অর্ডার থাকতে পারে, যেমন উপরের সমস্যায় তুমি নাস্তা খাবার আগে জামা-কাপড় পরতে পারতে। তোমাকে লেক্সিকোগ্রাফিকালি ছোটটা প্রিন্ট করতে বলতে পারে অথবা যেটা ইনপুটে আগে আছে সেটাকে আগে প্রিন্ট করতে বলতে পারে, আশা করি এসব কন্ডিশন সহজে হ্যান্ডল করতে পারবে।

ইমপ্লিমেন্টেশন নিয়ে তেমন কিছু বলার নেই। সবগুলো নোডের indegree বের করবে। তারপর যার indegree শূন্য তার সাথে যাদের এজ আছে তাদের indegree ১ কমিয়ে দিবে, তারপর আবার খুঁজবে কার indegree এখন শূন্য। এক নোডকে কখনো ২বার নিবেনা। ছবিতে এজ উঠিয়ে দেখিয়েছি বুঝানোর জন্য, তোমার ম্যাট্রিক্স থেকে এজ উঠানোর দরকার নেই, indegree কমালেই চলবে।

অনেক সময় বলতে পারে যতরকম ভাবে topsort করা যায় সবগুলো বের করতে। তখন তোমাকে backtracking এর সাহায্য নিতে হবে।

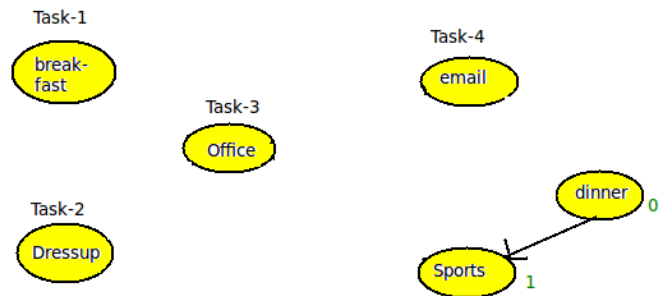
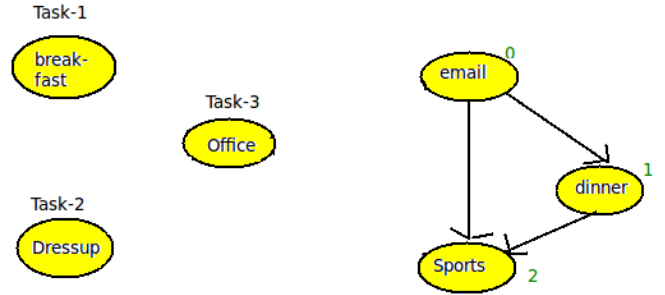
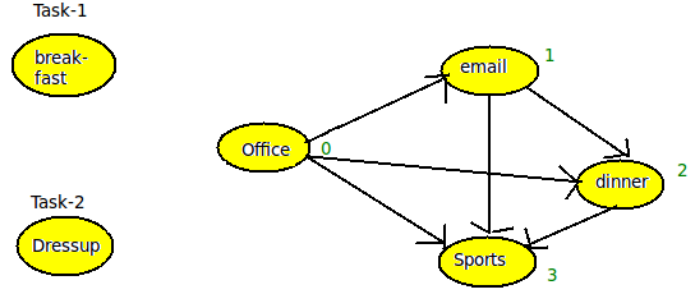
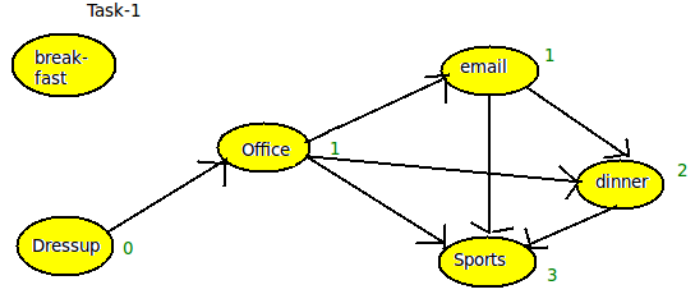
নিচের সমস্যাগুলো সমাধান করে ফেলো:

<http://uva.onlinejudge.org/external/103/10305.html>(Easy, straight-forward, special judge)

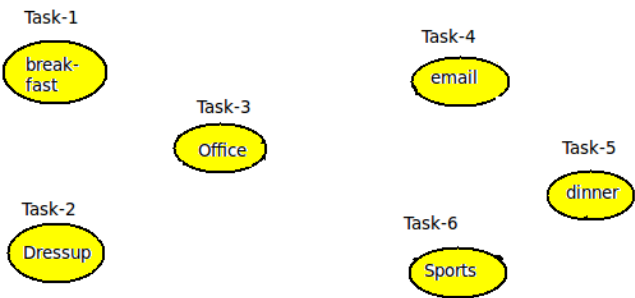
<http://uva.onlinejudge.org/external/110/11060.html>(Easy)

<http://uva.onlinejudge.org/external/1/124.html>(Medium, All possible topsort)

<http://uva.onlinejudge.org/external/4/452.html>(Medium)



(অন্যান্য পোস্ট)



গ্রাফ থিওরিতে হাতেখড়ি ৮: ডেপথ ফার্স্ট সার্চ এবং আবারো টপোলজিকাল সর্ট

বেশ কিছুদিন পর [সিরিজের](#) পরের পর্ব লিখতে বসলাম। এই পর্বে অ্যানিমেটেড জিআইএফ ইমেজ ব্যবহার করেছি।

আগের পর্বগুলো পড়ে থাকলে হয়তো ডেপথ ফার্স্ট সার্চ বা ডিএফএস এতদিনে নিজেই শিখে ফেলেছে। তারপরেও এই টিউটোরিয়ালটি পড়া দরকার কিছু কনসেপ্ট জানতে।

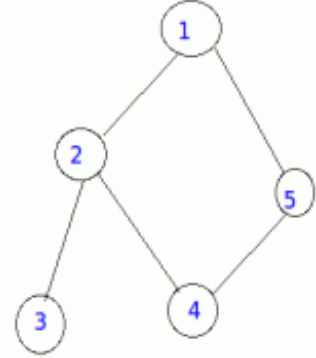
bfs এ আমরা গ্রাফটাকে লেভেল বাই লেভেল সার্চ করেছিলাম,নিচের অ্যানিমেশনের মতো করে:

(যদি অ্যানিমেশন দেখতে সমস্যা হয় তাহলে সবগুলো ছবি ডাউনলোড করতে পারো [এখান থেকে](#))

এবার আমরা কোনো নোড পেলে সাথে সাথে সে নোড থেকে আরো গভীরে চলে যেতে থাকবো,যখন আর গভীরে যাওয়া যাবেনা তখন আবার আগের নোডে ফিরে এসে অন্য আরেক দিকে যেত চেষ্টা করবো,এক নোড কখনো ২বার ভিজিট করবোনা। আমরা নোডের ৩টি রং(কালার) দিবা:

সাদা নোড= যে নোড এখনো খুঁজে পাইনি/ভিজিট করিনি।
 গ্রে বা ধূসর নোড= যে নোড ভিজিট করেছি কিন্তু নোডটি থেকে যেসব চাইল্ড নোডে যাওয়া যায় সেগুলো এখনো ভিজিট করে শেষ করিনি, অর্থাৎ নোডটিকে নিয়ে কাজ চলছে।
 কালো নোড= যে নোডের কাজ সম্পূর্ণ শেষ।

BFS



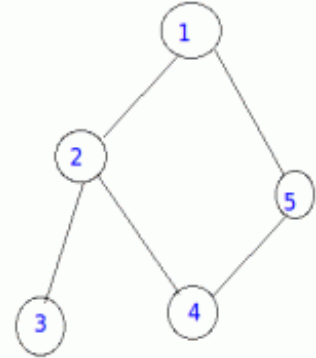
Created by: Shafaet, university of Dhaka
 GIF courtesy: <http://www.createagif.net/>

এবার আমরা অ্যানিমেশন দেখতে পারি:

আশা করি ডিএফএস কিভাবে কাজ করে এটা পরিষ্কার,খুব সহজ জিনিস এটা। এবার আমরা একটা খুব গুরুত্বপূর্ণ টার্ম শিখবো,সেটা হলো ব্যাকএজ(backedge)। অ্যানিমেশনে লক্ষ করো ৫-১ কে ব্যাকএজ বলা হয়েছে। এর কারণ হলো তখনও ১ এর কাজ চলছে,৫ থেকে ১ এ যাওয়া মানে এমন একটা নোড ফিরে যাওয়া যাকে নিয়ে কাজ এখনো শেষ হয়নি,তারমানে অবশ্যই গ্রাফে একটি সাইকেল আছে। এ ধরনের এজকে ব্যাকএজ বলে,dfs এ যদি কোনো সময় একটি গ্রে নোড থেকে আরেকটি গ্রে নোডে যেতে চেষ্টা করে তাহলে সে এজটি ব্যাকএজ এবং গ্রাফে অবশ্যই সাইকেল আছে। dfs এর সোর্স নোড এবং adjacency list এর উপর নির্ভর করে সাইকেলে যে কোনো এজকে ব্যাকএজ হিসাবে পাওয়া যেতে পারে,যেমন ১ থেকে আগে ২ এ না গিয়ে ৫ এ গেলে পরে ২-১ কে ব্যাকএজ হিসাবে পাওয়া যেতো।

আর যখন আমরা স্বাভাবিক ভাবে গ্রে থেকে সাদা নোডে যাচ্ছি তখন সে এজগুলোকে বলা হয় টি এজ।

আনডিরেক্টেড গ্রাফের ক্ষেত্রে আগে ভিজিট করা কোনো নোডে ফিরে গেলেই সেটা ব্যাকএজ,কালার চেক না করলেও হয়। তবে ডিরেক্টেড গ্রাফের ক্ষেত্রে অবশ্যই করতে হবে। আরেকটি ছোট অ্যানিমেশন:



Created by: Shafaet, university of Dhaka
 GIF courtesy: <http://www.createagif.net/>

২-৩এর এজটাকে ব্যাকএজ বলা যাচ্ছেনা, কারণ ৩ এর কাজ আগেই শেষ হয়ে গেছে। dfs এর কমপ্লেক্সিটি $O(V+E)$ ।

আমরা টপোলজিকাল সর্টের সমস্যা সমাধান করেছিলাম বারবার indegree উঠিয়ে। এবার আমরা খুব সহজে dfs দিয়ে এটা করবো। টপোলজিকাল কি সেটা না জানলে আগে [এই পোস্টটা পড়ো](#), তারপর আগাও।

মনে করি আমাদের এজগুলো হলো: ২-১, ২-৩, ৩-৪, ১-৪। অর্থাৎ ১ নম্বর কাজ করার আগে ২ নম্বরটি করতে হবে ইত্যাদি। এবার আমরা dfs চালানোর সময় একটি স্টপওয়াচ চালু করে দিবো। আর কোনো নোড নিয়ে কাজ শুরু করলে ঘড়ি দেখে নোডটি starting time/discovery time লিখে রাখবো, কাজ শেষ হলো নোডটির finishing time লিখে রাখবো।

finishing time দেখে আমরা সহজেই টপসর্ট করতে পারি। যে নোডটি সবার আগে আসবে তার finishing time অবশ্যই সবথেকে বেশি হবে, কারণ প্রথম নোডের উপর নির্ভরশীল সব নোড ঘুরে আসার পরে সে নোডের finishing time assign করা হয়। [uva 11504-dominos](#) প্রবলেমে আগে নোডগুলোকে finishing time দিয়ে সর্ট করে তারপর আবার dfs চালাতে হয়, প্রবলেমটা চেষ্টা করো।

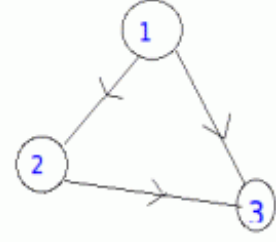
ডিএফএস দিয়ে আমরা যেসব কাজ করি সেগুলোর অনেকগুলোই bfs দিয়ে করতে পারি। bfs এ সাধারণত টাইম কমপ্লেক্সিটি কম হয় তবে dfs কোডিং করতে খুব কম সময় লাগে। একটা সিম্পল dfs এর সুডোকোড এরকম:

Python

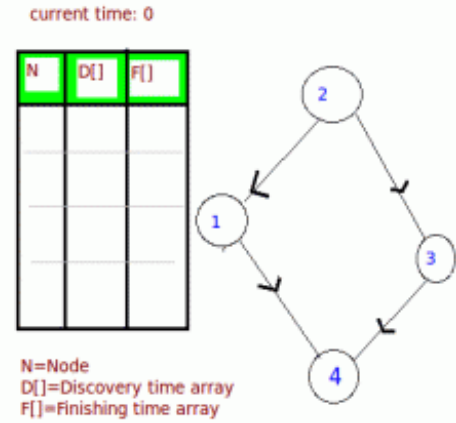
```

1  DFS (V, E)
2
3  for each vertex u in V[G]
4  do color[u] ← WHITE
5   $\pi[u] \leftarrow \text{NIL}$   $\triangleright \pi[]$  is parent array
6  time ← 0
7  for each vertex u in V[G]
8  do if color[u] ← WHITE
9  then DFS-Visit(u)  $\triangleright$  build a new DFS-tree from u
10
11 DFS-Visit(u)
12
13 color[u] ← GRAY  $\triangleright$  discover u
14 time ← time + 1  $\triangleright$  stopwatch
15 d[u] ← time
16 for each vertex v adjacent to u  $\triangleright$  explore (u, v)
17 do if color[v] ← WHITE
18 then  $\pi[v] \leftarrow u$   $\triangleright$  saving parent to print path
19 DFS-Visit(v)
20 color[u] ← BLACK
21
22 f[u] ← time  $\triangleright$  we are done with u
23 #http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgo/depthSearch.htm

```



Created by: Shafaet, university of Dhaka
Gif courtesy: <http://www.createagif.net/>



Created by: Shafaet, university of Dhaka
Gif courtesy: <http://www.createagif.net/>

[অনেকে বলছে ফিনিশিং টাইম সেভ করার আগে টাইমের মান বাড়িয়ে নিতে হবে নাহলে ফিনিশিং এবং ডিসকভারী টাইম সমান হয়ে যাবে। সেটাই

আসলে করা উচিত, আমি লেখার সময় ধরে নিয়েছিলাম সমান হলে কোন সমস্যা নেই, তবে কিছু প্রবলেমের ক্ষেত্রে সমস্যা না হলেও কোন কোন ক্ষেত্রে হতে পারে]

নিচের প্রবলেমগুলো সলভ করতে চেষ্টা করো:

<http://uva.onlinejudge.org/external/2/280.html>

<http://uva.onlinejudge.org/external/115/11518.html>

<http://uva.onlinejudge.org/external/104/10452.html>

যদি অ্যানিমেশন দেখতে সমস্যা হয় তাহলে সবগুলো ছবি ডাউনলোড করতে পারো [এখান থেকে](#) ।

এরপরে [এই আর্টিকেলটা](#) পড়ে ফেলো বিস্তারিত জানার জন্য, আমার লেখা পড়ে তুমি বেশিকটা শিখতে পারবে, বিস্তারিত জানতে এবং কঠিন প্রবলেম সলভ করতে আরো অনেক কিছু জানতে হবে।

গ্রাফ থিওরিতে হাতেখড়ি-১২ – ম্যাক্সিমাম ফ্লো (২)

আগের পর্বে আমরা দেখেছি কিভাবে ফোর্ড-ফুলকারসন পদ্ধতি ব্যবহার করে ম্যাক্সিমাম ফ্লো বের করতে হয়। এই পর্বে ম্যাক্সিমাম ফ্লো সমস্যার সহড কিছু ভ্যারিয়েশন দেখবো।

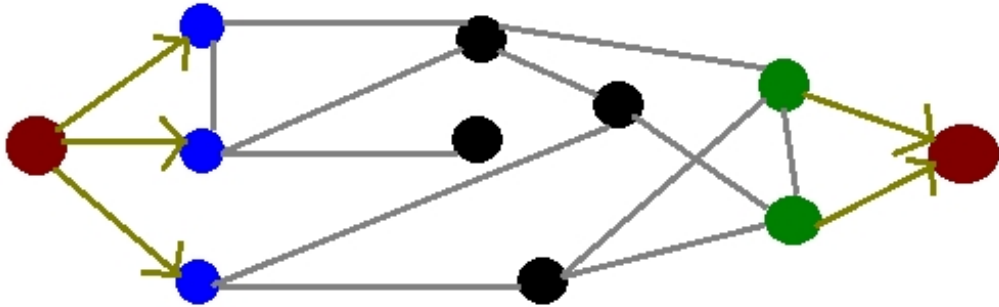
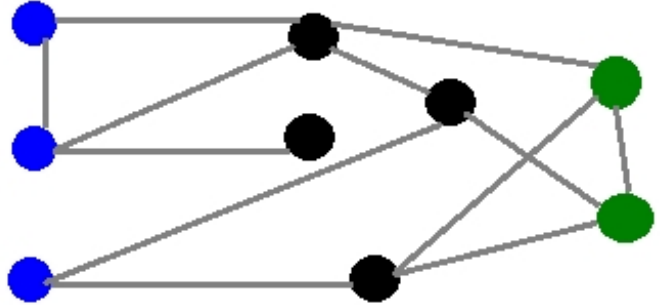
একাধিক সোর্স/সিংক:

আগের পর্বে একটা প্রশ্ন করেছিলাম এরকম “আমাদের প্রবলেমে সোর্স এবং সিংক ছিলো একটা। কিন্তু গ্রাফে একাধিক নোড দিয়ে পানি প্রবেশ করলে এবং একাধিক নোড দিয়ে পানি বের হয়ে গেলে কিভাবে অ্যালগোরিদমটা পরিবর্তন করবে?” অর্থাৎ একাধিক সোর্স বা সিংক থাকলে কি করতে হবে সেটা জানতে চাওয়া হয়েছে।

চিত্র-১ এ বাম পাশের নীল নোডগুলো হলো সোর্স এবং ডানের সবুজ নোডগুলো হলো সিংক।

চিত্র -১: একাধিক সোর্স এবং সিংক সহ একটি গ্রাফ

এ ধরনের গ্রাফে এডমন্ড কার্প অ্যালগোরিদম প্রয়োগ করার সহজ উপায় হলো সুপার-সোর্স এবং সুপার সিংক বানিয়ে নেয়া। সুপার সোর্স হলো এমন একটা নোড যেটা সবগুলো সোর্সের সাথে ডিরেক্টেড এজ দিয়ে যুক্ত। ঠিক সেভাবে, সুপার সিংক প্রতিটি সিংকের সাথে ডিরেক্টেড এজ নিয়ে সংযুক্ত। এবং এই এজগুলোর ক্যাপাসিটি হবে অসীম বা ইনফিনিটি।



চিত্র-২: সুপার সোর্স এবং সুপার সিংক

চিত্র-২ তে সুপার সোর্স এবং সুপার সিংক দেখানো হয়েছে। ইনফিনিটি হিসাবে বেছে নিতে পারো সবগুলো এজের সম্মিলিত ক্যাপাসিটির থেকে বড় কোনো মানকে। এখন সাধারণ ফ্লো অ্যালগোরিদম ব্যবহার করেই এই গ্রাফে ম্যাক্সিমাম ফ্লো বের করতে পারবে।

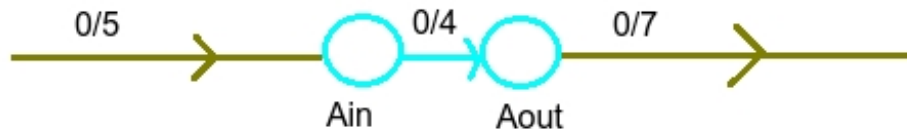
নোড ক্যাপাসিটি:

এতক্ষণ সবগুলো গ্রাফে এজের নির্দিষ্ট ক্যাপাসিটি ছিলো, নোডের ছিলো না। কিন্তু বাস্তবে অনেক সমস্যায় নোডের ক্যাপাসিটি থাকতে পারে। যেমন ধরো কোনো একটা দেশে প্রতিটা রাস্তার পাশাপাশি প্রতিটা শহরের নির্দিষ্ট গাড়ি ধারণ ক্ষমতা আছে, সেই দেশের গ্রাফ চিত্র-৩ এর মতো হতে পারে:



চিত্র-৩: নোড ক্যাপাসিটি

ছবিতে পুরো গ্রাফটা না একে শুধু একটা নোড আর ২টি এজ একেছি, নোডটাতে ঢোকার এজ এর ক্যাপাসিটি ৫, যে এজটি বাইরে চলে গেছে তার ক্যাপাসিটি ৭, এদিকে নোডের নিজের ক্যাপাসিটি ৪। আগে শেখা অ্যালগোরিদমে আমরা এজের ক্যাপাসিটির হিসাব রাখার জন্য একটা অ্যারে ব্যবহার করেছিলাম, এখনও আমরা সেই অ্যারেটা ব্যবহার করেই কাজ করতে পারবো, বুদ্ধিটা হলো নোডটাকে দুই ভাগে ভাগ করে ফেলা, এবং ভাগ দুটিকে নতুন এজ দিয়ে যোগ করে দেয়া। চিত্র-৪ দেখলেই পরিষ্কার হবে ব্যাপারটা:



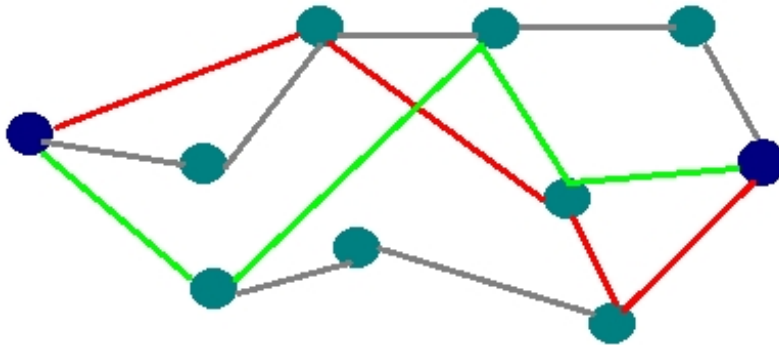
চিত্র-৪: A নোডটিকে দুইভাগ করা হয়েছে

আমরা A নোডটা Ain এবং Aout এই দুটি নোডে ভাগ করেছি। এখন আসল গ্রাফ যতগুলো এজ A তে প্রবেশ করেছে সেগুলো প্রবেশ করবে Ain এ এবং আসল গ্রাফে যতগুলো এজ A থেকে বাইরে গিয়েছে সেগুলো এখন বাইরে যাবে Aout থেকে। Ain থেকে Aout এ একটা এজ প্রবেশ করবে যেটার ক্যাপাসিটি হবে এজ এর ক্যাপাসিটির সমান।

এখন নিশ্চিত্তে তুমি আগের অ্যালগোরিদম ব্যবহার করতে পারো। কোড লেখার সময় কিভাবে নোড দুইভাগ করবে, আনডিরেক্টেড গ্রাফের ক্ষেত্রে ব্যাপারটা কিরকম হবে সেগুলো চিন্তা করা তোমার কাজ!

এজ ডিসজয়েন্ট পাথ:

দুই বন্ধু একই নোড থেকে যাত্রা শুরু করে একই গন্তব্যে পৌছাতে চায় কিন্তু দুইজনেই চায় ভিন্ন ভিন্ন রাস্তা ব্যবহার করে যেতে, তারমানে একই এজ কখনো ২জন ব্যবহার করতে পারবে না। এধরণের পথকে এজ ডিসজয়েন্ট পাথ বলে। তোমাকে বলতে হবে কোনো একটা গ্রাফে দুটি এজ ডিসজয়েন্ট পাথ আছে নাকি। চিত্র-৫ এ একটা উদাহরণ দেখানো হয়েছে:



ছবিতে দুইজনই বামের নীল নোডটা থেকে যাত্রা শুরু করে ডানের নীল নোডটাতে যেতে চায়। লাল এবং সবুজ রঙ ব্যবহার করে দুটি এজ-

ডিসজয়েন্ট পাথ দেখানো হয়েছে।

সাধারণ ম্যাক্স-ফ্লো ব্যবহার করেই এজ ডিসজয়েন্ট পাথ বের করা যায়। শুরু নোডকে সোর্স এবং গন্তব্য নোডকে সিংক ধরবে। এবার সবগুলো এজ এর ক্যাপাসিটি বানিয়ে দাও ১ এর সমান। এখন যদি তুমি সোর্স থেকে সিংকে দুই ফ্লো পাঠাতো পারো সেটার মানে হলো দুটি ডিসজয়েন্ট পাথ আছে। প্রতিটা এজের ক্যাপাসিটি ১ হওয়াতে ২ ফ্লো যে দুটি পথে গিয়েছে তাদের মধ্যে কমন এজ থাকা সম্ভব না।

ঠিক একই ভাবে তুমি একটা গ্রাফে সর্বোচ্চ কয়টা ডিসজয়েন্ট পাথ থাকা সম্ভব অথবা দুই বন্ধুর জায়গায় K টা বন্ধু থাকলে কি হতো বের করে ফেলতে পারবে।

এখন প্রশ্ন হলো তুমি যদি প্রতিটা রাস্তার নির্দিষ্ট দৈর্ঘ্য থাকে এবং ডিসজয়েন্ট পাথ দুটির মোট দৈর্ঘ্য মিনিমাইজ করতে চাও তাহলে ফ্লো এর অ্যালগোরিদমটা কিভাবে পরিবর্তন করবে? এটা বের করতে পারলে [uva 10806](#) সমস্যাটা সমাধান করে ফেলো, সমস্যাটার নামের ভিতরেই কিভাবে সমাধান করতে হবে বলা আছে!

সময়ের অভাবে আজকের পর্ব এখানেই শেষ। মিন-কাট এবং ম্যাচিং নিয়ে আলোচনার জন্য আরেকটা পর্ব অপেক্ষা করতে হবে। কনটেস্টে ম্যাক্স-ফ্লো প্রবলেমের কঠিন অংশ হলো গ্রাফটা কিভাবে তৈরি করবো, এজগুলো কিভাবে যোগ করবো, কোন এজের ক্যাপাসিটি কত এগুলো বের করা, এসব করার পর ফ্লো অ্যালগোরিদম চালিয়ে দেয়া সহজ কাজ। তাই তোমাকে প্রচুর প্র্যাকটিস করে এই জিনিসগুলো আয়ত্তে আনতে হবে।

কিছু প্রবলেম:

[Down Went Titanic](#)
[Clever Naming Pattern](#)
[Diagonal Sum](#)

হ্যাপি কোডিং!

গ্রাফ থিওরিতে হাতেখড়ি-৪(ব্রেডথ ফার্স্ট সার্চ-bfs)

[[পুরনো লেখা, বিএফএস এর ২টি পর্বকে মার্জ এবং আপডেট করা হয়েছে। ২-ডি গ্রিডে বিএফএস নিয়ে আলোচনা যোগ করা হয়েছে।]]

আগের [পর্বগুলোতে](#) আমরা দেখেছি কিভাবে ম্যাট্রিক্স বা লিস্ট ব্যবহার করে গ্রাফ স্টোর করতে হয়। এবার আমরা প্রথম অ্যালগোরিদম দেখবো এর দিকে যাবো। শুরুতেই আমরা যে অ্যালগোরিদমটা শিখব তার নাম ব্রেডথ ফার্স্ট সার্চ(breadth first search,bfs)।

বিএফএস এর কাজ হলো গ্রাফে একটা নোড থেকে আরেকটা নোডে যাবার শর্টেস্ট পাথ বের করা। বিএফএস কাজ করবে শুধুমাত্র আন-ওয়েটেড গ্রাফের ক্ষেত্রে, তারমানে সবগুলো এজের কন্স্ট হবে ১।

বিএফএস অ্যালগোরিদমটা কাজ করে নিচের ধারণাগুলোর উপর ভিত্তি করে:

১. কোনো নোডে ১ বারের বেশি যাওয়া যাবেনা
২. সোর্স নোড অর্থাৎ যে নোড থেকে শুরু করছি সেটা ০ নম্বর লেভেলে অবস্থিত।
৩. সোর্স বা 'লেভেল ০' নোড থেকে সরাসরি যেসব নোডে যাওয়া যায় তারা সবাই 'লেভেল ১' নোড।
৪. 'লেভেল ১' নোডগুলো থেকে সরাসরি যেসব নোডে যাওয়া যায় তারা সবাই 'লেভেল ২' নোড। এভাবে লেভেল এক এক করে বাড়তে থাকবে।
৫. যে নোড যত নম্বর লেভেলে, সোর্স থেকে তার শর্টেস্ট পথের দৈর্ঘ্য তত।

উপরে লেখাগুলো পুরোপুরি মাথায় না ঢুকলেও ক্ষতি নেই,মোটামুটি বুঝলেই হলো,এখন আমরা ছবি দেখে বাকিটা পরিষ্কার করব।

ধর তুমি ১ নম্বর শহর থেকে ১০ নম্বর শহরে যেতে চাও।
প্রথমে আমরা সোর্স ধরলাম ১ নম্বর নোডকে। ১ তাহলে একটা 'লেভেল ০' নোড। ১ কে ডিজিটেড চিহ্নিত করি।

১ থেকে সরাসরি যাওয়া যায় ২,৩,৪ নম্বর নোডে। তাহলে ২,৩,৪ হলো 'লেভেল ১' নোড। এবার সেগুলোকে আমরা ডিজিটেড চিহ্নিত করি এবং সেগুলো নিয়ে কাজ করি। নিচের ছবি দেখ:

লাল নোডগুলো নিয়ে আমরা এখন কাজ করবো। রঙিন সবগুলো নোড ডিজিটেড, এক নোডে ২বার কখনো যাবোনা। ২,৩,৪ থেকে শর্টেস্ট পথে যাওয়া যায় ৬,৭,৮ এ। সেগুলো ডিজিটেড চিহ্নিত করি:

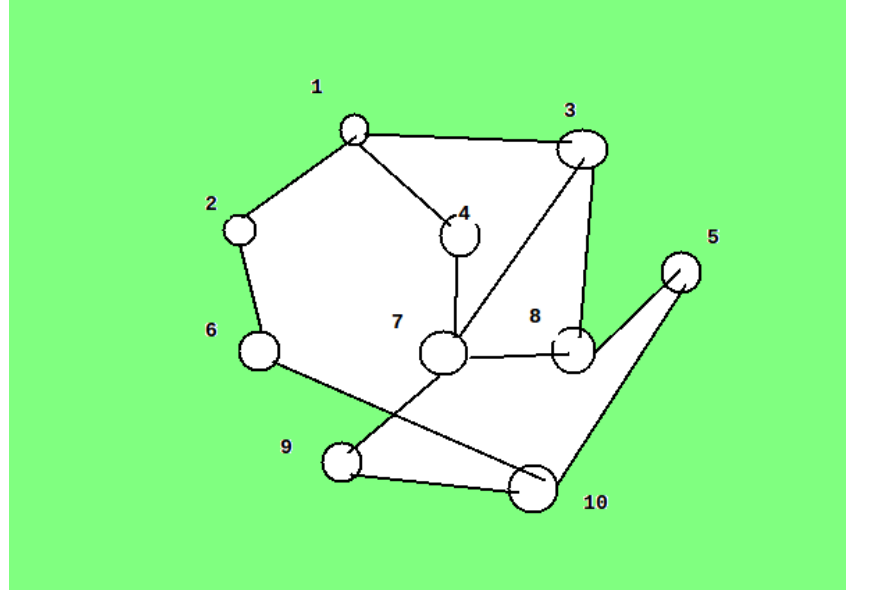
লক্ষ কর যে নোডকে যত নম্বর লেভেলে পাচ্ছি,সোর্স থেকে তার শর্টেস্ট পথের দৈর্ঘ্য ঠিক তত। যেমন ২নম্বর লেভেলে ৮কে পেয়েছি তাই ৮ এর দূরত্ব ২। ছবিগুলোকে একেকটা লেভেলের একেক রং দেয়া হয়েছে। আর লাল নোড দিয়ে বুঝানো হয়েছে আমরা এখন ওগুলো নিয়ে কাজ করছি। আমরা ১০ এ পৌছাইনি তাই পরের নোডগুলো ডিজিট করে ফেলি:

আমরা দেখতে পাচ্ছি ২টি লেভেল পার হয়ে ৩ নম্বর লেভেলে আমরা ১০ কে পাচ্ছি। তাহলে ১০ এর শর্টেস্ট পথ ৩। লেভেল বাই লেভেল গ্রাফটাকে সার্চ করে আমরা শর্টেস্ট পথ বের করলাম। যেসব এজ গুলো আমরা ব্যবহার করিনি সেগুলোকে বাদ দিয়ে ছবিটিকে একটু ঘুরিয়ে নিচের মত করে আকতে পারি:

লক্ষ্য কর গ্রাফদুটি একই,খালি নোডগুলো ঘুরানো হয়েছে। যেসব এজ ব্যবহার করিনি সেগুলো হালকা করে দিয়েছি,এই এজ গুলো বাদ দিলে গ্রাফটি একটি ট্রি হয়ে যায়। এই ট্রি টাকে বলা হয় বিএফএস ট্রি।

তারমানে আমাদের কাজ গুলো সোর্স থেকে লেভেল ১ নোডগুলোতে যাওয়া, তারপর লেভেল ১ এর নোডগুলো থেকে লেভেল ২ নোডগুলো খুঁজে বের করা, এভাবে যতক্ষন না গন্তব্যে পৌছে যাচ্ছি অথবা সব নোড ডিজিট করা শেষ হয়ে গিয়েছে ততক্ষন কাজ চলতে থাকবে।

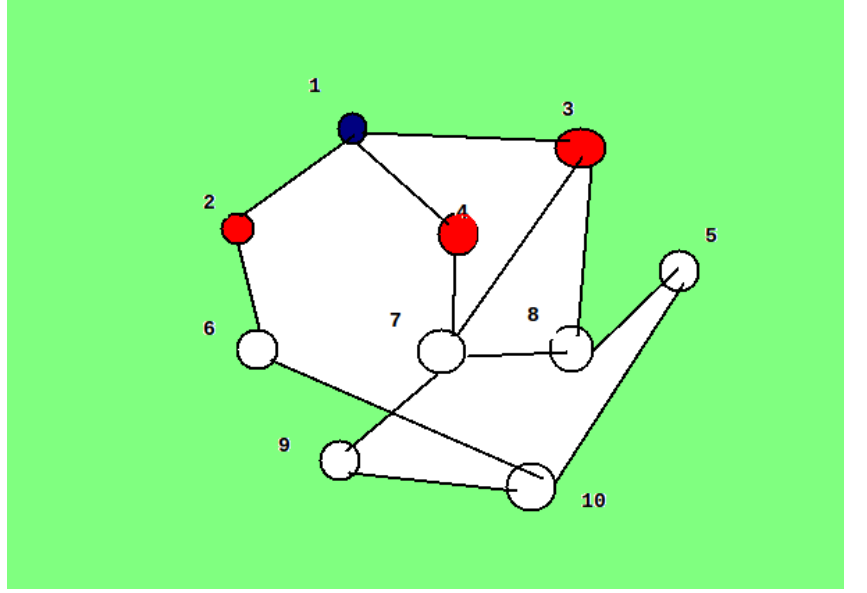
কিউ ডাটা স্ট্রাকচারটার সাথে আশা করি সবাই পরিচিত। কিউ হলো হুবুহু বাসের লাইনের মতো ডাটা স্ট্রাকচার। যখন একটা সংখ্যা কিউতে যোগ



করা হয় তখন সেটা আগের সবগুলো সংখ্যার পিছে গিয়ে দাড়ায়, যখন কোন একটা সংখ্যা বের করে ফেলা হয় তখন সবার প্রথমে সংখ্যাটা নেয়া হয়। একে বলা ফাস্ট ইন ফাস্ট আউট। আমরা বিএফএস এ কিউ কাজে লাগাতে পারি। লেভেল ১ থেকে যখন কয়েকটা নতুন লেভেল ২ নোড পাবো সেগুলোকে কিউতে বা লাইনে অপেক্ষা করিয়ে রাখবো, আর সবসময় প্রথম নোডটা নিয়ে কাজ করবো। তাহলে বড় লেভেলের নোডগুলো সবসময় পিছের দিকে থাকবে, আমরা ছোট লেভেলগুলো নিয়ে কাজ করতে করতে আগাবো। উপরের গ্রাফের জন্য এটা আমরা সিমুলেট করে দেখি:

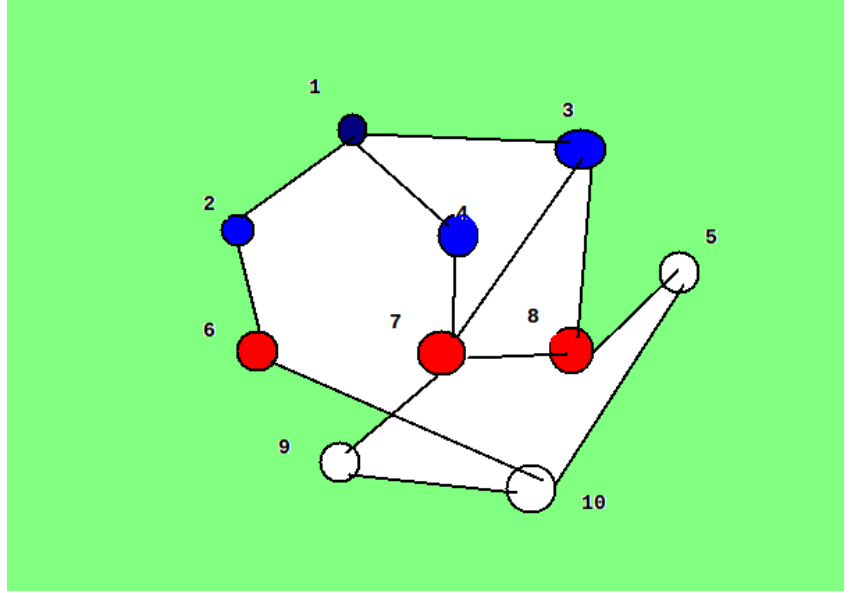
প্রথমে কিউতে সোর্স পুশ করবো:

কিউ: [১]



১ এর লেভেল হবে ০ বা লেভেল[১]=০। এবার বিএফএস শুরু করবো। প্রথমে কিউ এর সবার সামনের নোডটাকে নিয়ে কাজ করবো। সবার সামনে আছে ১, সেখান থেকে যাওয়া যায় ২,৩,৪ এ। ২,৩,৪ এ এসেছি ১ থেকে, তাহলে লেভেল[২]=লেভেল[১]+১=১, লেভেল[৩]=লেভেল[১]+১=১, লেভেল[৪]=লেভেল[১]+১=১। এদেরকে কিউতে পুশ করে রাখি:

কিউ: [২, ৩, ৪, ১]



১ আর কোন কাজে আসবেনা, ১ কে পপ করি বা ফেলে দেই।

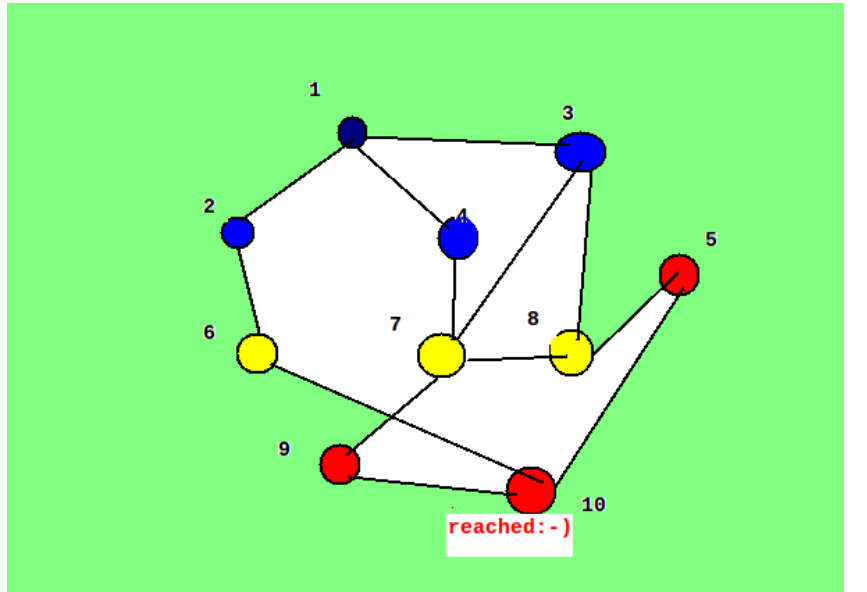
কিউ: [২, ৩, ৪]

এবার ৪ নিয়ে কাজ করি। ৪ থেকে যাওয়া যায় ৭ এ। ৭ এ এসেছি ৪ থেকে, লেভেল[৭]=লেভেল[৪]+১=২। ৭ কে কিউতে পুশ করি:

কিউ: [৭, ২, ৩, ৪]

এখন ৪ কে পপ করি:

কিউ: [৭, ২, ৩]



৩ থেকে ৭,৮ এ যাওয়া যায়। ৭ কে এরই মধ্যে নিয়েছি, শুধু ৮ পুশ করতে হবে। লেভেল[৮]=লেভেল[৩]+১=২।

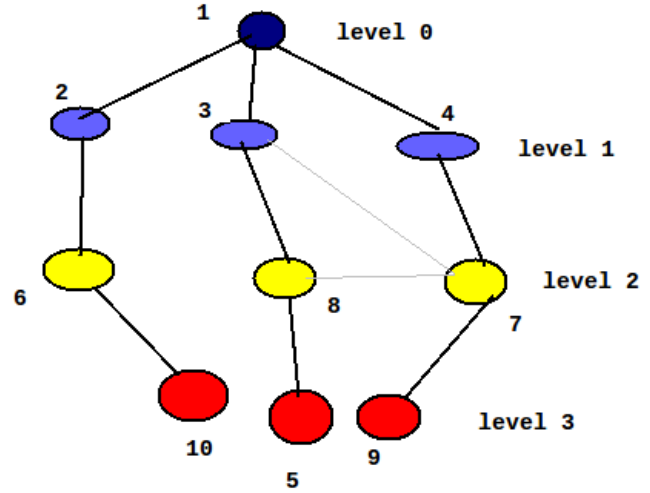
কিউ: [৮, ৭, ২, ৩]

ও পপ করি:

কিউ: [৮, ৭, ২]

এভাবে যতক্ষণনা কিউ খালি হচ্ছে ততক্ষণ কাজ চলতে থাকবে। লেভেল[] অ্যারের মধ্যে আমরা পেয়ে যাবো সোর্স থেকে সবগুলো নোডের দূরত্ব!

সি++ এর স্ট্যান্ডার্ড টেমপ্লেট লাইব্রেরির কিউ ব্যবহার করে আমরা খুব সহজেই কোড করতে পারি। প্রথমে নিজে চেষ্টা করো, তারপর নিচের কোডটা দেখো:



C++

```
1  vector<int>G[100];
2  void bfs(int n,int src)
3  {
4      queue<int>Q;
5      Q.push(src);
6      int visited[100]={0},level[100];
7      int parent[100];
8      visited[src]=1;
9      level[src]=0;
10     while(!Q.empty())
11     {
12         int u=Q.front(); //Q এর সামনের নোড নিয়ে আমরা কাজ করবো
13         for(int i=0;i<G[u].size();i++)
14         {
15             int v=G[u][i];
16             if(!visited[v])
17             {
18                 level[v]=level[u]+1;
19                 parent[v]=u;
20                 visited[v]=1;
21                 Q.push(v);
22             }
23         }
24         Q.pop();
25     }
26     for(int i=1;i<=n;i++)
27         printf("%d to %d distance %d",src,i,level[i]);
28 }
```

ঠিক যেভাবে সিমুলেট করেছি সেভাবেই কোডটা লিখেছি, আশা করি বুঝতে সমস্যা হচ্ছেনা।

শুধু পাথের দৈর্ঘ্য যথেষ্ট না, পাথটাও দরকার হতে পারে। লক্ষ্য করো আমরা u থেকে v তে যাবার সময় parent[v]=u করে দিচ্ছি। আমরা প্রতিটা নোডের জন্য জানি কোন নোড থেকে সেই নোডে এসেছি। তাহলে আমরা যে নোডের জন্য পাথ বের করতে চাই সেই নোড থেকে তার প্যারেন্ট নোডে যেতে থাকবো যতক্ষণনা সোর্সে পৌঁছে যাই। খুবই সহজ কাজ, পাথ বের করার কোড করা তোমার উপর ছেড়ে দিলাম।

কমপ্লেক্সিটি:

প্রতিটা নোডে একবার করে গিয়েছি, প্রতিটা এজ এ একবার গিয়েছি। তাহলে কমপ্লেক্সিটি হবে $O(V+E)$ যেখানে V হলো নোড সংখ্যা এবং E হলো এজ সংখ্যা।

কখনো কখনো ২-ডি গ্রিডে বিএফএস চালানো লাগতে পারে। যেমন একটা দাবার বোর্ডে একটি ঘোড়া আর একটা রাজা আছে। ঘোড়াটা মিনিমাম কয়টা মুভে রাজার ঘরে পৌঁছাতে পারবে? অথবা একটা ২-ডি অ্যারেতে কিছু সেল ব্লক করে দেয়া হয়েছে, এখন কোনো সেল থেকে আরেকটি সেলে

মিনিমাম মুভে পৌছাতে হবে, প্রতি মুভে শুধুমাত্র সামনে-পিছে-বামে-ডানে যাওয়া যায়। আগে নোডকে আমরা প্রকাশ করছিলাম একটা মাত্র সংখ্যা দিয়ে, এখন নোডকে প্রকাশ করতে হবে দুটি সংখ্যা দিয়ে, রো(row) নাম্বার, এবং কলাম নাম্বার। তাহলে আমরা নোড রিপ্রেজেন্ট করার জন্য একটা স্ট্রাকচার বানিয়ে নিতে পারি এরকম:

```
| struct node{int r,c};
```

অথবা আমরা সি++ এর “পেয়ার” ব্যবহার করতে পারি।

```
| pair<int,int>
```

এ ক্ষেত্রে ডিজিটেড, প্যারেট, লেভেল অ্যাংগুলা হবে ২ ডিমেনশনের, যেমন `visited[10][10]` ইত্যাদি। কিউতে নোডের বদলে স্ট্রাকচার পুশ করবো, অর্থাৎ কিউটা হবে এরকম: `queueQ`। আর কোন একটা ঘর থেকে অন্য ঘরে যাবার সময় চেক করতে হবে বোর্ডের বাইরে চলে যাচ্ছে কিনা। একটা স্যাম্পল কোড দেখি:

C++

```
1  #define pii pair
2  int fx[]={1,-1,0,0}; //ডিরেকশন অ্যারে
3  int fy[]={0,0,1,-1};
4  int cell[100][100]; //cell[x][y] যদি -১ হয় তাহলে সেলটা ব্লক
5  int d[100][100],vis[100][100]; //d means destination from source.
6  int row,col;
7  void bfs(int sx,int sy) //Source node is in [sx][sy] cell.
8  {
9      memset(vis,0,sizeof vis);
10     vis[sx][sy]=1;
11     queue<pii>q; //A queue containing STL pairs
12     q.push(pii(sx,sy));
13     while(!q.empty())
14     {
15         pii top=q.front(); q.pop();
16         for(int k=0;k<4;k++)
17         {
18             int tx=top.uu+fx[k];
19             int ty=top.vv+fy[k]; //Neighbor cell [tx][ty]
20             if(tx>=0 and tx<row and ty>=0 and ty<col and cell[tx][ty]!=-1 and vis[tx][ty]==0) //Check if the
21             neighbor is valid and not visited before.
22             {
23                 vis[tx][ty]=1;
24                 d[tx][ty]=d[top.uu][top.vv]+1;
25                 q.push(pii(tx,ty)); //Pushing a new pair in the queue
26             }
27         }
28     }
```

তুমি যদি ডিরেকশন অ্যারের ব্যাপারটা না বুঝো তাহলে [এই লেখাটা](#) পড়লে আরো কিছু ডিটেইলস জানতে পারবে।

প্র্যাকটিসের জন্য প্রবলেম:

[Bicoloring](#)(Bipartite checking)

[A Node Too Far](#)(Shortest path)

[Risk](#)(Shortest path)

[Bombs! NO they are Mines!!](#)(bfs in 2d grid)

[Knight Moves](#)(bfs in 2d grid)

[We Ship Cheap](#)(Printing path)

Word Transformation(strings)

পরের পর্ব

গ্রাফ থিওরিতে হাতেখড়ি-৯ (ডায়াক্সট্রা) | শাফায়েতের ব্লগ

আমরা শুরুতেই শিখেছি কিভাবে শর্টেস্ট পাথে এক জায়গা থেকে আরেক জায়গায় যেতে হয়। সেজন্য আমরা শিখেছি **ব্রেডথ ফার্স্ট সার্চ** নামের একটি সার্চিং অ্যালগোরিদম। অ্যালগোরিদমটি চমককার কিন্তু সমস্যা হলো সে ধরে নেয় প্রতিটি রাস্তা দিয়ে যেতে সমান সময় লাগে, মানে সব এজ এর কস্ট সমান। প্র্যাকটিকাল লাইফে বেশিভাগ ক্ষেত্রেই এটা অচল হয়ে পড়ে, তখন আমাদের দরকার পরে ডায়াক্সট্রা। প্রথমে নাম শুনে আমার ধারণা হয়েছিলো ডায়াক্সট্রা খুবই ভয়ংকর কোনো জিনিস কিন্তু আসলে বিএফএস লেখার মতোই সহজ ডায়াক্সট্রা লেখা, আমি তোমাদের দেখানোর চেষ্টা করবো কিভাবে বিএফএসকে কিছুটা পরিবর্তন করে একটা প্রায়োরিটি কিউ যোগ করে সেটাকে ডায়াক্সট্রা বানিয়ে ফেলা যায়।

ডায়াক্সট্রা শুরু করার আগে আমরা পাথ রিল্যাক্সেশন(relax) নামের একটা ছোট্ট জিনিসের সাথে পরিচিত হই। ধরো সোর্স থেকে প্রতিটা নোডের ডিসটেন্স রাখা হয়েছে $d[]$ অ্যারেতে। যেমন $d[3]$ মানে হলো সোর্স থেকে বিভিন্ন এজ পার হয়ে ৩ এ আসতে মোট $d[3]$ ডিসটেন্স লেগেছে। যদি ডিসটেন্স জানা না থাকে তাহলে ইনফিনিটি অর্থাৎ অনেক বড় একটা মান রেখে দিবো। আর $cost[u][v]$ তে রাখা আছে $u-v$ এজ এর কস্ট।

ধরো তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসি তে গেলে ১০ মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে ২৫ মিনিটে। তাহলে তোমার কাছে ইনফরমেশন আছে:

$$d[টিএসসি]=১০, d[কার্জনহল]=২৫$$

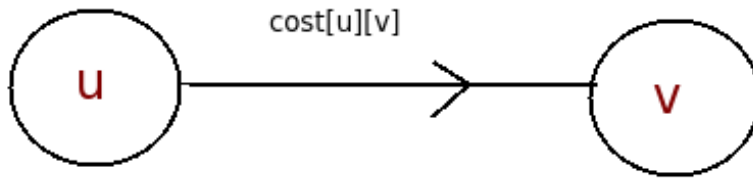
এখন তুমি দেখলে টিএসসি থেকে ৭ মিনিটে কার্জনে চলে যাওয়া যায়,

$$cost[টিএসসি][কার্জন]=৭$$

তাহলে তুমি ২৫ মিনিটের জায়গায় মাত্র $১০+৭=১৭$ মিনিটে কার্জনহলে যেতে পারবে। যেহেতু তুমি দেখেছো:

$$d[টিএসসি]+cost[টিএসসি][কার্জন] < d[কার্জনহল]$$

তাই তুমি এই নতুন রাস্তা দিয়ে কার্জন হলে গিয়ে $d[কার্জনহল]=d[টিএসসি]+cost[টিএসসি][কার্জন]$ বানিয়ে দিতেই পারো!!



We will go through this edge if

$$(d[u] + \text{cost}[u][v] < d[v])$$

$d[u]$ = distance from source to u

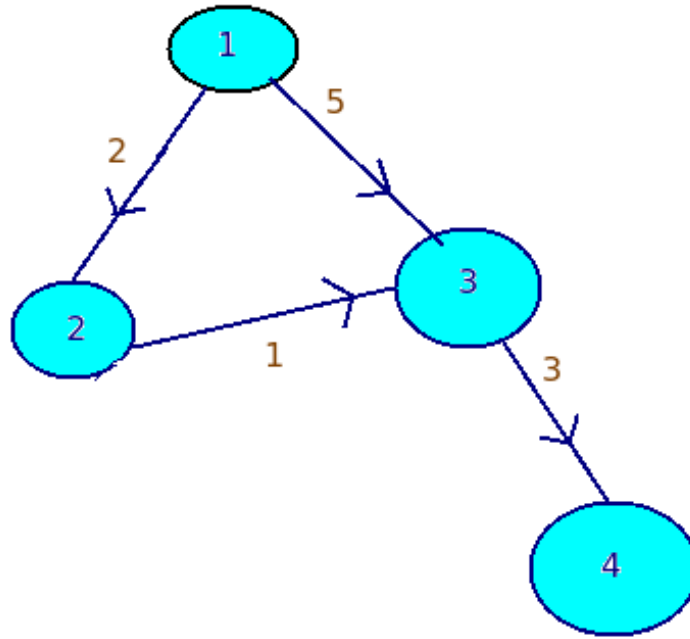
উপরের ছবিটা সেটাই বলছে। আমরা u থেকে v তে যাবো যদি $d[u] + \text{cost}[u][v] < d[v]$ হয়। আর $d[v]$ কে আপডেট করে $d[v] = d[u] + \text{cost}[u][v]$ বানিয়ে দিবো। ভবিষ্যতে যদি কার্জনহলে অন্য রাস্তা দিয়ে আরো কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে কম্পিয়ার করে আপডেট করি দিবো। ব্যাপারটা অনেকটা এরকম:

C++

```
1 if(d[u]+cost[u][v] < d[v])
2 {
3   d[v]=d[u]+cost[u][v];
4 }
```

উপরের অংশটা যদি বুঝে থাকো তাহলে ডায়াক্সট্রা বোঝার ৬০% কাজ হয়ে গেছে। না বুঝে থাকলে আবার পড়ো।

বিএফএস নিশ্চয়ই তুমি ভালো করে বুঝো। [বিএফএস](#) এ আমাদের একটা নোডে কখনো দুইবার যাওয়া দরকার হয়নি, আমরা প্রতিবার দেখেছি একটা নোড ভিজিটেড কিনা, যদি ভিজিটেড না হয় তাহলে সেই নোডকে কিউতে পুশ করে দিয়েছি এবং ডিসটেন্স ১ বাড়িয়ে দিয়েছি। ডায়াক্সট্রাতে আমরা একই ভাবে কিউ তে নোড রাখবো তবে ভিজিটেড দিয়ে আপডেট না করে নতুন এজকে “রিল্যাক্স” বা আপডেট করবো উপরের পদ্ধতিতে। নিচের গ্রাফটা দেখো:



ধরে নেই সোর্স হলো ১ নম্বর নোড। তাহলে

$d[1]=0, d[2]=d[3]=d[4]=\text{infinity (a large value)}$

ইনফিনিটি কারণ ২, ৩, ৪ এর দূরত্ব আমরা এখনো জানিনা, আর সোর্সের দূরত্ব অবশ্য শূন্য। এখন তুমি আগের বিএফএস এর মতোই সোর্স থেকে যতগুলো নোডে যাওয়া যায় সেগুলো আপডেট করার চেষ্টা করো, আপডেট করতে পারলে কিউতে পুশ করো। যেমন ১-২ এজটা ধরে আমরা আগাবো কারণ $d[1]+2 < d[2]$ এই শর্তটা পূরণ হচ্ছে। তখন $d[2]$ হয়ে যাবে ২, একই ভাবে ১ থেকে ৩ এ গেলে $d[3]$ হয়ে যাবে ৫।

কিন্তু ৫ তো ৩ নম্বর নোডে যাওয়ার শর্টেস্ট ডিসটেন্স না! আমরা বিএফএস এ দেখেছি একটা নোড একবারের বেশি আপডেট হয়না, সেই প্রোপার্টি এখানে কাজ করছেনা। ২ নম্বর নোড থেকে ২-৩ এজ ধরে এগিয়ে আবার আপডেট করলে তখন $d[3]$ তে $d[2]+1=3$ পাবো। তাহলে আমরা দেখলাম এক্ষেত্রে একটা নোড অনেকবার আপডেট হতে পারে। (প্রশ্ন: সর্বোচ্চ কত বার?)

আমরা তাহলে আগের বিএফএস এর কোডের আপডেট অংশ একটু পরিবর্তন করি যাতে একটা নোড বার বার আপডেট হতে পারে:

C++

```

1  vectorG[100];
2  int cost[100][100];
3
4  void shortest_path(int n,int src)
5  {
6      int d[100];
7      for(int i=1;i<=n;i++)
8          d[i]=1000000000;
9      queueQ;
10     Q.push(src);
11     d[src]=0;
12     while(!Q.empty())
13     {
14         int u=Q.front();
15         for(int i=0;i<(int)G[u].size();i++)
16         {
17             int v=G[u][i];
18
19             if(d[u]+cost[u][v]<d[v]) // update/relaxation
20             {
21                 d[v]=d[u]+cost[u][v];
22                 Q.push(v);
23             }
24         }
25         Q.pop();
26     }
27 }

```

আমরা ঠিক আগের বিএফএস এর কোডেই জাস্ট কন্স্ট বসায় বারবার আপডেট করছি! এই কোড তোমাকে সোর্স থেকে প্রতিটা নোডের শর্টেস্ট পাথ বের করে দিবে কিন্তু কমপ্লেক্সিটির দিক থেকে এটা খুবই বাজে! এজন্য আমাদের লাগবে একটা প্রায়োরিটি কিউ।

বিএফএস এ আমরা যখন ১ নোড থেকে অনেকগুলো নোডে যাচ্ছি তখন সেই নোডগুলো থেকে আবার নতুন করে কাজ করার সময় “আগে আসলে আগে পাবেন” ভিত্তিতে কাজ করছি। যেমন উপরের গ্রাফে ১ থেকে আগে ৩ নম্বর নোডে এবং তারপর ২ নম্বর নোডে এ গলে আগে ৩ নিয়ে কাজ করছি, এরপর ২ নিয়ে কাজ করছি।

ভালো করে দেখো এখানে কি সমস্যাটা হচ্ছে। ৩ নিয়ে আগে কাজ করলে আমরা ৪ এর ডিসটেন্সকে আপডেট করে দিচ্ছি ডিসটেন্স $৫+৩=৮$ হিসাবে। পরবর্তীতে যখন ২ দিয়ে ৩ কে আবার আপডেট করা হচ্ছে তখন ৩ এর ডিসটেন্স হয়ে গিয়েছে ৩, এবার ৪ এর ডিসটেন্সকে আবার আপডেট করছি $৩+৩=৬$ হিসাবে। ৪ কে মোট দুইবার আপডেট করা লাগলো।

বিজ্ঞানী ডায়াব্রুটা চিন্তা করলেন যদি এই “আগে আসলে আগে পাবেন” ভিত্তিতে কাজ না করে সবথেকে কাছে নোডগুলোকে আগে প্রসেস করি তাহলে অনেক কমবার আপডেট করা লাগে। আমরা যদি ২ কে নিয়ে আগে কাজ করতাম তাহলে ৩ আগেই আপডেট হয়ে যেত এবং ৪ কে একবার আপডেট করেই শর্টেস্ট ডিসটেন্স পেয়ে যেতাম! একটু হাতে কলমে সিমুলেট করে দেখো। আইডিয়াটা হলো যেকোনো সময় কিউ তে যতগুলো নোড থাকবে তাদের মধ্যে যেটা সোর্স থেকে সবথেকে কাছে সেটা নিয়ে আগে কাজ করবো। এজন্যই আমরা কিউ এর জায়গায় বসিয়ে দিবো একটি প্রায়োরিটি কিউ যে কিউতে নোড পুশ করার সাথে সাথে কাছের নোডটাকে সামনে এনে দিবে। পার্থক্য হলো আগে খালি নোড নাম্বার পুশ করেছি, এখন বর্তমান ডিসটেন্স অর্থাৎ $d[u]$ এর মানটাও পুশ করতে হবে।

নিচে একটা সম্পূর্ণ ডায়াব্রুটার কোড দিলাম যেটা ১ থেকে n তম নোডে যাবার শর্টেস্ট পাথ বের করে এবং পাথটাও প্রিন্ট করে:

C++

```

1  #define mx 100002
2  vectorg[mx],cost[mx];
3  struct node
4  {
5      int u,w;
6      node(int a,int b){u=a; w=b;}
7      bool operator <; ( const node& p ) const { return w > p.w; }
8  };
9  int d[mx],par[mx];

```

```

10 int dijkstra(int n)
11 {
12     memset(d,63,sizeof(d));
13     memset(par,-1,sizeof(par));
14     priority_queue<int> q;
15     q.push(node(1,0));
16     d[1]=0;
17     while(!q.empty())
18     {
19         node top=q.top(); q.pop();
20         int u=top.u;
21
22         if(u==n) return d[n];
23         for(int i=0;i<(int)g[u].size();i++)
24         {
25             int v=g[u][i];
26             if(d[u]+cost[u][i]<d[v])
27             {
28                 d[v]=d[u]+cost[u][i];
29                 par[v]=u;
30                 q.push(node(v,d[v]));
31             }
32         }
33     }
34     return -1;
35 }
36 int main(){
37     int n,e;
38     cin>>n>>e;
39     for(int i=0;i<e;i++)
40     {
41         int u,v;
42         int w;
43         cin>>u>>v>>w;
44         g[u].push_back(v);
45         g[v].push_back(u);
46         cost[u].push_back(w);
47         cost[v].push_back(w);
48     }
49     int ret=dijkstra(n);
50     if(ret== -1) puts("No path!");
51     else
52     {
53         int u=n;
54         vector<int> out;
55         while(u!= -1)
56         {
57             out.push_back(u);
58             u=par[u];
59         }
60         reverse(out.begin(),out.end());
61         for(int i=0;i<(int)out.size();i++)
62             cout<<out[i]<<endl;
63         puts("");
64     }
65 }
66
67
68
69

```

এখানে কয়েক জায়গায় একটু ভিন্নতা আছে। একটা হলো `cost[][]` অ্যারেতে কস্ট না রেখে গ্রাফের অ্যাডজেন্সি লিস্টের পাশাপাশি কস্ট এর লিস্ট রেখেছি, অ্যারে রাখলে নোড বেশি হলে সমস্যা করবে। শুরুতেই `node` নামের একটা স্ট্রাকচার বানিয়ে নিয়েছি যেখানে নোডের নম্বরের সাথে কস্ট

টাও রেখে দেয়া হয়েছে। কস্ট অনুসারে অপারেটর ওভারলোড করেছি যাতে প্রায়োরিটি কিউতে যার কস্ট কম সেটা আগে আসে। বাকি কাজ আগের মতোই।

একটা ব্যাপার লক্ষ্য করো, টার্গেট নোড পপ হবার সাথে সাথে আমরা কস্ট রিটার্ন করে দিয়েছি। এটার কারণ কিউতে বাকি যত নোড আছে সবাই কস্ট আরো বেশি বা সমান তাই ভবিষ্যতে টার্গেট নোডের আর কখনো আপডেট হবার সম্ভাবনা নেই।

নেগেটিভ এজ থাকলে কি ডায়াগ্রামটা অ্যালগোরিদম কাজ করবে? যদি নেগেটিভ সাইকেল থাকে তাহলে ইনফিনিট লুপে পড়ে যাবে, বারবার আপডেট করে কস্ট কমাতে থাকবে। যদি নেগেটিভ এজ থাকে কিন্তু সাইকেল না থাকে তাহলেও কাজ করবে না। তবে তুমি যদি টার্গেট পপ হবার সাথে সাথে রিটার্ন করে না দাও তাহলে কাজ করবে কিন্তু সেটা তখন আর মূল ডায়াগ্রামটা অ্যালগোরিদম থাকবে না।

কমপ্লেক্সিটি:

বিএফএস এর কমপ্লেক্সিটি ছিলো $\log(V+E)$ যেখানে V হলো নোড সংখ্যা আর E হলো এজ সংখ্যা। এখানেও আগের মতোই কাজ হবে তবে প্রায়োরিটি কিউ তে প্রতিবার সর্ট করতে $\log V$ কমপ্লেক্সিটি লাগবে। মোট: $\log(V\log V+E)$ ।

নেগেটিভ সাইকেল নিয়ে কাজ করতে হলে আমাদের জানতে হবে বেলম্যান ফোর্ড অ্যালগোরিদম। সেখানেও এজ রিল্যাক্স করে আপডেট করা হয়, একটা নোডকে সর্বোচ্চ $n-1$ বার আপডেট করা লাগতে পারে সেই প্রোপার্টি কাজে লাগানো হয়। বেলম্যান নিয়ে আলোচনা হবে অন্য আরেকদিন, তুমি চাইলে এখনই গুগল করে শিখে নিতে পারো। আর ডায়াগ্রামটা ভালো করে শিখতে নিচের প্রবলেমগুলো ঝটপট করে ফেলো:

Dijkstra?
Not the Best
New Traffic System

হ্যাপি কোডিং!

(গ্রাফ থিওরি নিয়ে সবগুলো লেখা)

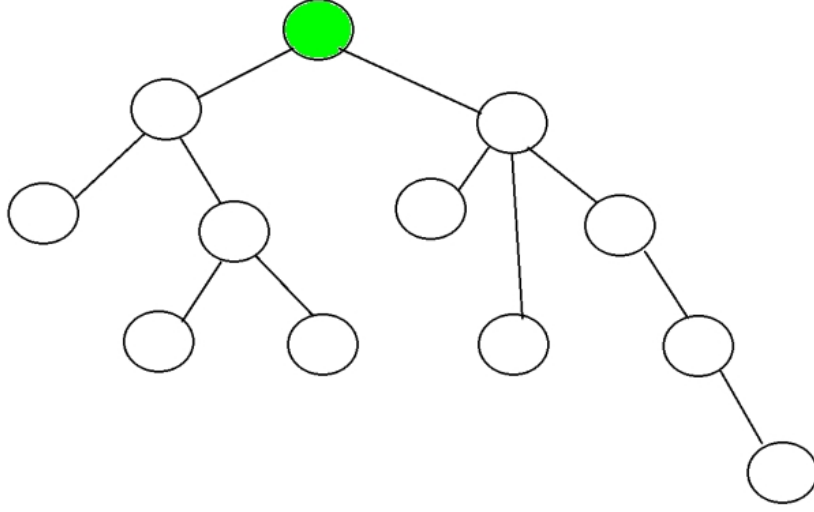
টি ডায়ামিটার (অ্যালগোরিদমের প্রমাণ সহ)

[[অনেক আগে প্রমাণ ছাড়াই টি এর ডায়ামিটার বের করা নিয়ে লিখেছিলাম, এবার প্রমাণ যোগ করে নতুন করে লিখলাম]]

টি হলো এমন একটা আনডিরেক্টেড গ্রাফ যেটার সব নোড থেকে সব নোডে যাওয়া যায় এবং কোনো সাইকেল নেই। এখন আমাদের টি এর সবথেকে দূরের দুটা নোড খুজে বের করতে হবে, একেই বলা হয় টি এর ডায়ামিটার।

মনে করো কিছু কম্পিউটারের মধ্যে নেটওয়ার্ক কেবল লাগানো হয়েছে নিচের ছবির মতো করে। এখন তুমি জানতে চাইতেই পারো কোন দুটি কম্পিউটার সবথেকে দূরে আছে।

shafaetsplanet.com/blog



এটা বের করা খুব সহজ, এজন্য তোমার জানতে হবে বিএফএস বা ডিএফএস এর যে কোন একটা। আনডিরেক্টেড টি তে যেকোন নোডকেই রুট ধরা যায়, আমরা মনে করি উপরের সবুজ নোডটা টি এর রুট।

আমাদের প্রথম কাজ কাজ হলো রুট হতে সবথেকে দূরের নোডটা খুজে বের করা। সেই নোডটাকে মনে করি X। একাধিক নোডের দূরত্ব সবথেকে দূরের নোডের দূরত্বের সমান হলেও সমস্যা নেই, যেকোন একটাকে সিলেক্ট করতে হবে। এই কাজটা আমরা ডিএফএস/বিএফএস চালিয়ে বের করতে পারি।

এখন ২য় কাজ হলো X নোড থেকে শুরু আরেকটি ডিএফএস/বিএফএস চালিয়ে X থেকে সবথেকে দূরের নোড খুজে বের করা। মনে করি নোডটা হলো Y।

X আর Y এর মধ্যকার দূরত্বই টি এর ডায়ামিটার! উপরের ছবিতে ডায়ামিটার ৬।

প্রমাণ:

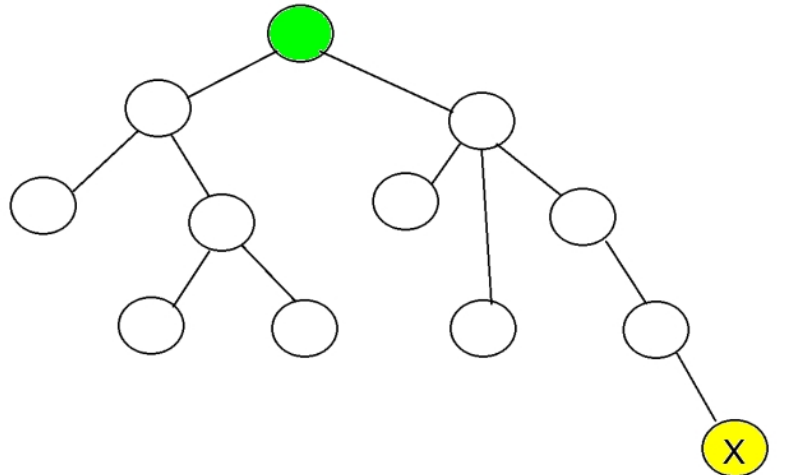
এই অ্যালগোরিদমটা সবারই জানা তবে প্রমাণটা অনেকেই জানে না, আমি নিজেও প্রমাণ জেনেছি অনেক পরে।

আমরা যদি প্রমাণ করতে পারি ১ম ধাপে খুজে পাওয়া X সবসময়ই ডায়ামিটারের একটা প্রান্ত হবে তাহলেই অ্যালগোরিদমটা প্রমাণ হয়ে যায়। কারণ X

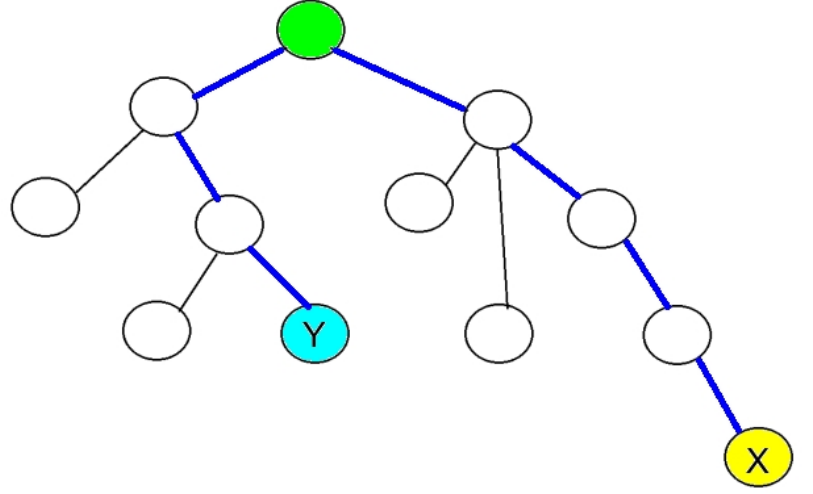
যদি নিশ্চিতভাবে ডায়ামিটারের একটা প্রান্ত হয় তাহলে ২য় ধাপটা অবশ্যই সঠিক, X থেকে সবথেকে দূরের নোডই হবে টি এর ডায়ামিটার।

এটা আমরা প্রমাণ করবো “প্রফ বাই কন্ট্রাডিকশন” এর সাহায্যে। এটা বহুল ব্যবহৃত একটা পদ্ধতি। আমরা প্রমাণ করতে চাই X নিশ্চয়ই কোনো

shafaetsplanet.com/blog



ডায়ামিটারের প্রাপ্ত। কিন্তু তা যদি না হয়, অর্থাৎ X কোনো ডায়ামিটারের প্রাপ্ত না হলে এমন একটা ডায়ামিটার থাকবে যার দুই প্রান্ত (ধরি) a এবং b , এখানে a এবং b যেকোনো দুটি নোড হতে পারে (X ছাড়া)। $a-b$ ডায়ামিটার ব্যবহার করে আমরা দেখাবো এমন আরেকটা ডায়ামিটার পাওয়া সম্ভব যেটা $a-b$ এর চেয়ে বড় বা সমান এবং যার এক প্রান্তে X আছে। (উল্লেখ্য, X হলো রুট থেকে সবচেয়ে দূরের একটা নোড)।



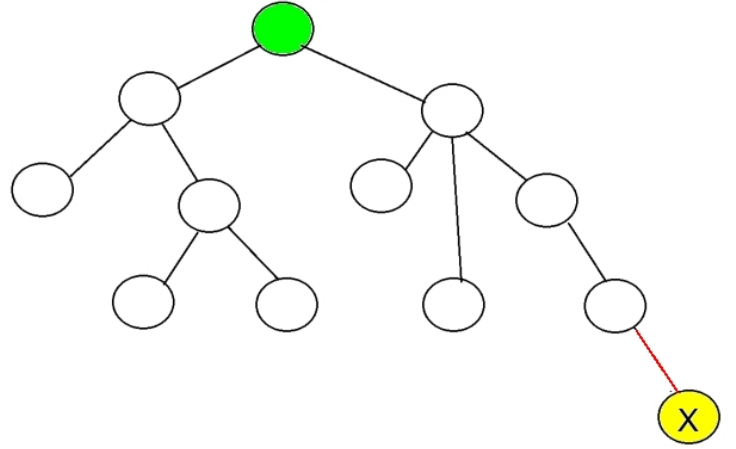
এখন আবিষ্কারি(arbitrarily) দুটি নোড a আর b সিলেক্ট করি। নোড দুটির মধ্যকার পথ রুটের কাছে যে নোডে এসে মিলবে সেটার নাম দেই h , অর্থাৎ h হলো নোড দুইটার কমন অ্যানসেস্টর। (অনেকেই হয়তো বুঝে গেছো আমরা এখানে lowest common ancestor এর কথা বলছি)।

কিন্তু রুট থেকে b এর দূরত্ব, রুট থেকে X এর দূরত্বের কম বা সমান হতে বাধ্য, বেশি হবে না কারণ রুট থেকে সবথেকে দূরের নোড হলো X ।

অর্থাৎ:

$$distance(root, b) \leq distance(root, X)$$

আবার এটাও নিশ্চিত যে রুট থেকে b যত দূরে, h থেকে b এর দূরত্ব তার থেকে কম বা সমান। (সমান হবে যদি h আর রুট একই নোড হয় অর্থাৎ a আর b যদি রুটের দুইপাশে থাকে)

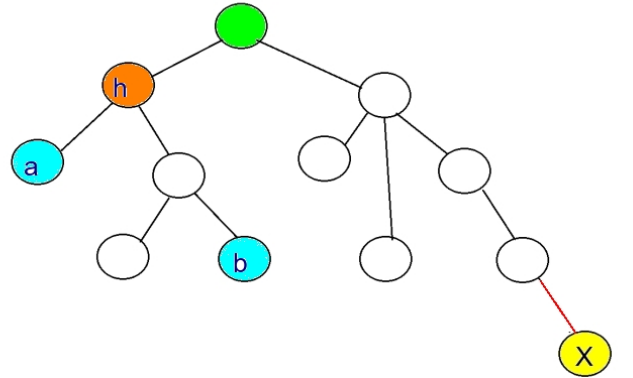


$$distance(h, b) \leq distance(root, b)$$

তাহলে,

$$distance(h, b) \leq distance(root, b) \leq distance(root, X)$$

$$distance(h, b) \leq distance(root, X)$$



তাহলে আমরা $h-b$ পাথটা $root-X$ পাথ দিয়ে রিপ্লেস করে দিলে এবং

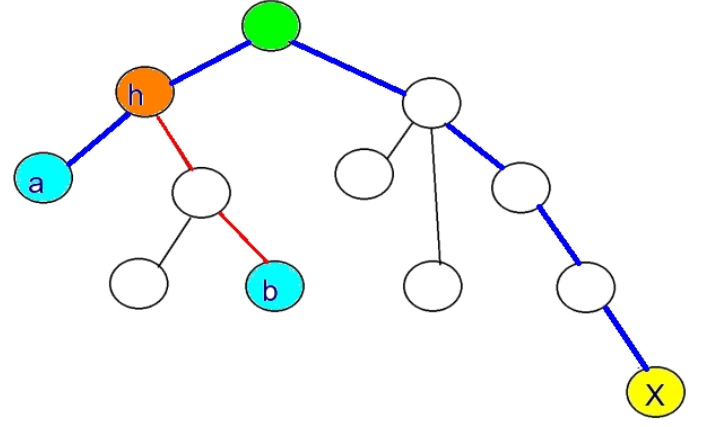
$h-x$ এর মধ্যে পাথ বসিয়ে দিলে (যদি h আর x সমান না হয়) অবশ্যই আগের সমান বা আগের থেকে বড় একটা পাথ পাবো!

তারমানে আমরা যে X কে বাদ দিয়ে যেই দুইটা নোডই সিলেক্ট করিনা কেন, X কে নিয়ে তার থেকে বড় বা সমান একটা পথ পাওয়া যাবে। তারমানে X অবশ্যই ডায়ামিটারের একটা প্রান্ত!

কমপ্লেক্সিটি:

বিএফএস/ডিএফএস এর কমপ্লেক্সিটির সমান: $O(V+E)$
যেখানে V হলো নোডসংখ্যা এবং E হলো এজ সংখ্যা।

এই অ্যালগোরিদমটা জেনারেল গ্রাফে কাজ করবেনা, শুধু ট্রি এর ক্ষেত্রে করবে। জেনারেল গ্রাফে লংগেস্ট পথ বের করার প্রবলেম *এন-পি হার্ড*, আর ডায়ামিটার বের করার প্রবলেম লংগেস্ট পথ প্রবলেমেরই স্পেশাল কেস।



Farthest Nodes in a Tree

Farthest Nodes in a Tree (II)

হ্যাপি কোডিং!

কৃতজ্ঞতা স্বীকার:

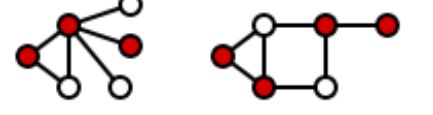
<http://stackoverflow.com/questions/20010472/proof-of-correctness-algorithm-for-diameter-of-a-tree-in-graph-theory>

<http://apps.topcoder.com/forums/?module=Thread&threadID=668470&start=0&mc=12#1216875>

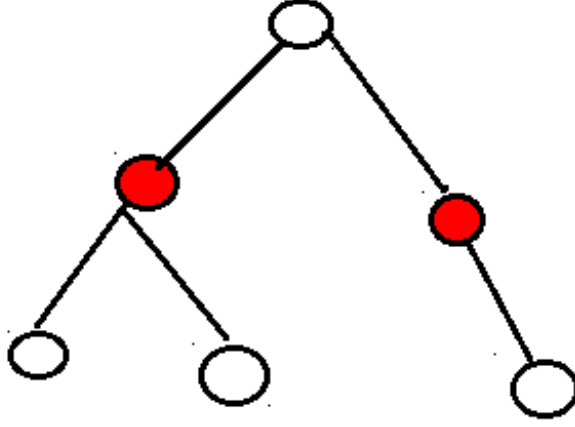
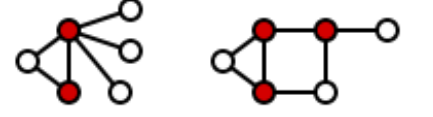
মিনিমাম ভারটেক্স কভার প্রবলেম(রিপোস্ট) | শাফায়েতের ব্লগ

(মিনিমাম ভারটেক্স কভার নিয়ে অনেক আগে লিখেছিলাম,লেখাটা বুঝতে অনেকের সমস্যা হওয়াতে নতুন করে লেখাটি এডিট করলাম। আশা করি এখন বুঝতে সমস্যা হবেনা)

মিনিমাম ভারটেক্স কভার একটি ক্লাসিক গ্রাফ প্রবলেম। ধরা যাক একটি শহরে কিছু রাস্তা আছে,এখন প্রতি রাস্তায় মোড়ে আমরা পাহারাদার বসাতে চাই। কোনো নোডে পাহারাদার বসালে সে নোডের সাথে যুক্ত রাস্তাগুলো একাই পাহারা দিতে পারে। উপরের ছবিতে নোডগুলো হলো রাস্তার মোড়। এখন সব কয়টা রাস্তা পাহারা দিতে ন্যূনতম কয়জন পাহারাদার দরকার? ছবিতে লাল নোডগুলোতো পাহারাদার বসানো হয়েছে। এটা অপটিমাল না,নিচের ছবির মত বসালে পাহারাদার কম লাগত:



এটি একটি NP-hard প্রবলেম,অর্থাৎ এই প্রবলেমের কোনো পলিনমিয়াল টাইম সলিউশন নেই। তবে গ্রাফটি যদি **Tree** হয় অর্থাৎ $n-1$ টা edge থাকে আর কোনো সাইকেল না থাকে তাহলে **ডাইনামিক প্রোগ্রামিং** বা ম্যাক্স ফ্লো/বাইপারটাইট ম্যাচিং এর সাহায্যে প্রবলেমটি সলভ করা সম্ভব।



Minimum vertex Cover in a Tree Graph

ডাইনামিক প্রোগ্রামিং সলিউশনটা আমি বিস্তারিত লিখছি,তারপর ম্যাক্স ফ্লো/বাইপারটাইট ম্যাচিং দিয়ে কিভাবে করতে হয় লিখবো।

ডিপি সলিউশনে ২টি কেস আমাদের লক্ষ্য করতে হবে:

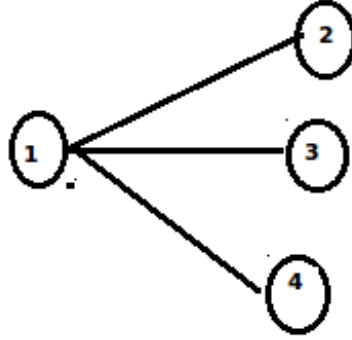
- কোনো নোডে পাহারাদার না বসালে তার সাথে সংযুক্ত সব নোডে অবশ্যই পাহারাদার বসাতে হবে,এছাড়া সব রাস্তা কভার হবে না। অর্থাৎ যদি u আর v সংযুক্ত থাকে তাহলে u তে পাহারাদার না বসালে v তে অবশ্যই বসাতে হবে।
- কোনো নোডে পাহারাদার বসালে সংযুক্ত নোডগুলোতে পাহারাদার বাসানো বাধ্যতামূলক না তবে বসালে লাভ হতে পারে। তাই u তে পাহারাদার বসালে v তে পাহারাদার একবার বসিয়ে এবং একবার না বসিয়ে দেখবো কোনটা লাভজনক

সব ডিপির প্রবলেমের মতো এখানেও একটা রিকার্সিভ ফাংশন ডিফাইন করবো। আমাদের স্টেট হবে বর্তমানে কোন নোডে আছি,এবং সেই নোডে কোনো পাহারাদার বসানো হয়েছে নাকি।

$F(u, 1)$ = বর্তমানে u নম্বর নোডে আছি এবং এই নোডে পাহারাদার আছে। $f(u, 1)$ রিটার্ন করবে বাকি নোডগুলোতে মোট পাহারাদার সংখ্যা।

$F(u,0)$ = বর্তমানে u নম্বর নোডে আছি এবং এই নোডে পাহারাদার নাই। $f(u,0)$ রিটার্ন করবে বাকি নোডগুলোতে মোট পাহারাদার সংখ্যা।

ধরি ১ নম্বর নোডের সাথে ২,৩,৪ নম্বর নোড যুক্ত।



বুঝাই যাচ্ছে ১ নম্বর নোডে পাহারা না বসালে অবশ্যই ২,৩,৪ সবগুলোয় পাহারা বসাতে হবে। তাহলে আমরা বলতে পারি:

$F(1,0)=F(2,1)+F(3,1)+F(4,1) + 0$, অর্থাৎ ১ এর সাথে সংযুক্ত সব নোডগুলোতে পাহারা বসালে প্রয়োজনীয় মোট পাহারাদার সংখ্যা।

সবশেষে ০ যোগ করছি কারণ বর্তমান নোডে পাহারাদার বসাইনি।

এবার $F(1,1)$ এর মান বের করি। ১ নম্বর নোডে পাহারা বসালে সংযুক্ত নোডগুলোতে পাহারা বসালেও চলে, না বসালেও চলে, তবে যেটা অপটিমাল রেজাল্ট দেয় সেটা আমরা নিব:

$$F(1,1)=1+\min (F(2,1),F(2,0)) +\min (F(3,1),F(3,0)) +\min (F(4,1),F(4,0))$$

১ নম্বর নোডে পাহারাদার বসানো তাই সবশেষে ১ যোগ হচ্ছে, প্রতি নোডে একবার পাহারা বসিয়ে, আবার না বসিয়ে দেখছি কোনটা অপটিমাল।

একটা ব্যাপার লক্ষ রাখতে হবে যে প্যারেন্ট নোড নিয়ে কখনো হিসাব করবোনা। উপরের ছবিতে ১ থেকে ২ এ গেলে $\text{parent}[2]=1$, তাই ২ থেকে আবার ১ নম্বর নোডে যাবোনা।

এবার **base case** এ আসি। কোনো নোড থেকে নতুন কোনো নোডে যাওয়া না গেলে ১ বা ০ রিটার্ন করে দিতে হবে, পাহারাদার বসালে ১, না বসালে ০। কোনো ট্রি তে একটি মাত্র নোড থাকলে ১ রিটার্ন করতে হবে (কিছু প্রবলেমে ০ ও রিটার্ন করতে হতে পারে)।

Spoj এর [PT07X\(vertex cover\)](#) প্রবলেমটি **straight forward** প্রবলেম। এটার জন্য আমার কোডটা এরকম:

C++

```

1  #define MAXN 100002
2  int dp[MAXN][5];
3  int par[MAXN];
4  vector<int>edges[MAXN];
5
6  int f(int u,int isGuard)
7  {
8      if(edges[u].size()==0)return 0;
9      if(dp[u][isGuard]!=-1) return dp[u][isGuard];
10     int sum=0;
11     for(int i=0;i<(int)edges[u].size();i++) {
12         int v=edges[u][i];
13         if(v!=par[u]){
14             par[v]=u;
15             if(isGuard==0) sum+=f(v,1);
16             else sum+=min(f(v,1),f(v,0));
17         }
18     }
19     return dp[u][isGuard]=sum+isGuard;
20 }
21
22 int main()
23 {
24     memset(dp,-1,sizeof(dp));
25     int n;
26     scanf("%d",&n);
27     for(int i=1;i<n;i++){
28         int u,v;
29         scanf("%d%d",&u,&v);
30         edges[u].push_back(v);
31         edges[v].push_back(u);
32     }
33     int ans=0;
34     ans=min(f(1,1),f(1,0));
35     printf("%d\n",ans);
36     return 0;
37 }

```

আমি টি এর root সবসময় ১ ধরে কোড লিখেছি। ৩৮ নম্বর লাইনে মেইন ফাংশনে root এ পাহারাদার একবার বসিয়ে আর একবার না বসিয়ে অপটিমাল রেজাল্ট টা নিচ্ছি।

ফাংশনে u হলো current node, isguard কারেন্ট নোডে পাহারাদার আছে নাকি নাই সেটা নির্দেশ করে।

১০ নম্বর লাইনে টি এর সাইজ ১ হলে ১ রিটার্ন করে দিয়েছি।

১৩ নম্বর লাইনে লুপের ভিতর current নোড থেকে সবগুলো child নোডে যাচ্ছি। কারেন্ট নোডে পাহারাদার না থাকলে পরেরটায় বসাচ্ছি, আর থাকলে ২ভাবেই চেষ্টা করছি। ১৫ নম্বর লাইনের কন্ডিশন দিয়ে প্যারেন্ট নোডে যেতে দিচ্ছি না।

সবশেষে sum+isGuard রিটার্ন করছি। অর্থাৎ কারেন্ট নোডে পাহারাদার থাকলে ১ যোগ করছি, নাহলে ০।

মোটামুটি এই হলো ডিপি সলিউশন। টি তে সাইকেল না থাকায় এটা অবশ্যই বাইপারটাইট গ্রাফ। ১৯৩১ সালে [Dénés König](#) প্রমাণ করেন কোনো বাইপারটাইট গ্রাফে maximum matching=minimum vertex cover। এটা গ্রাফ থিওরির অনেক min-max থিওরেমের একটা যেখানে কিছু একটা ম্যাক্সিমাইজ করলে অন্য আরেকটা কিছু মিনিমাইজ হয়। তুমি যদি ম্যাক্সিমাম ম্যাচিং এর অ্যালগোরিদম জানো তাহলে টি টা বাইকালারিং করে ম্যাচিং বের করলেই ভারটেক্স কভার বের হয়ে যাবে। কোড সহজ হলেও complexity বেড়ে যাবে, তাই নোড বেশি থাকলে কাজ করবেনা। আবার ম্যাক্সিমাম ম্যাচিং যেহেতু ম্যাক্স-ফ্লো এর একটি ভ্যারিয়েশন তাই ফ্লো চালিয়েও সমাধান করা সম্ভব।

এরকম আরেকটা প্রবলেম [uva-10243\(fire fire fire\)](#)। আমি প্রথমে এটা সমাধান করে পরে spoj এর টা করেছি।