

# HANDBOOK OF ALGORITHMS

## Part-2

*Courtesy of*  
*Shafaet Ashraf*

**For personal use only**  
**NOT FOR SALE**

Shafaet Cover.docx

Part\_0 - Introduction.pdf

Algorithm Notations Cover.pdf

Algorithm Notations.pdf

Basic Algo Shafaet Cover.docx

Some Basic Algorithm & Techniques.pdf

Number Theory cover.docx

Number Theory.pdf

Data Structures Shafaet Cover.docx

Data Structure.pdf

Graph Cover.docx

Graph Theory.pdf

DP Shafaet Cover.docx

Dynamic Programming.pdf

Game Theory Shafaet Cover.docx

Game Theory.pdf

String Shafaet Cover.docx

String Algorithms.pdf

কনে আমি প্রযোগ্যরাম শিখিবো\_ \_ শাফায়তেরে ব্লগ.pdf

কম্পিউটার বজ্জিণন \_ শাফায়তেরে ব্লগ.pdf

প্রযোগ্যরাম কন্টেন্ট এবং অনলাইন জাজে হাতখেড়ি \_ শাফায়তেরে ব্লগ.pdf

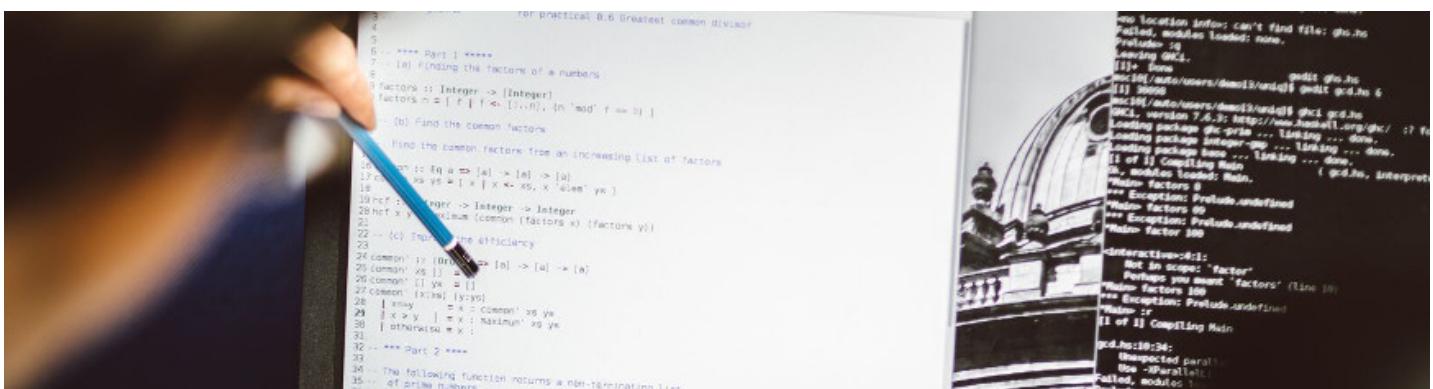
# কেন আমি প্রোগ্রামিং শিখবো?

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

জানুয়ারি ২১, ২০১৩

সকালে উঠেই **টপকোডারে** এ লেখা দেখলাম “একটি শিশুকে একই আইফোন দিলে সে দিনরাত অ্য়েঞ্জি বার্ডস খেলবে, শিশুটিকে কোডিং শিখালে সে আইফোনটার জন্য সফটওয়্যার তৈরি করবে” দারুণ এই লেখাটা দেখে মনে হলো কেন আমরা প্রোগ্রামিং বা কোডিং শিখবো সেটা নিয়ে বাংলায় কিছু লিখি। এ লেখাটি প্রোগ্রামিং নিয়ে যাদের কোনো ধারণা নেই বা খুব সামান্য ধারণা আছে তাদের আগ্রহী করে তোলার একটি ছোট্ট প্রচেষ্টা।



কম্পিউটার একটি অসম্ভব ক্ষমতাবান কিন্তু নির্বোধ একটি যন্ত্র। একটি যন্ত্র ৫০জন সাধারণ মানুষের কাজ একাই করতে পারে কিন্তু ৫০টি যন্ত্র একটি অসাধারণ মানুষের কাজ করতে পারেনা(Hubbard, Elbert)। প্রোগ্রামিং শিখে আমরা একেকজন হয়ে উঠতে পারি সেই মানুষটি যে এই যন্ত্রকে ইচ্ছামত কথা শোনাতে পারে। তুমি যা বলবে যেভাবে কম্পিউটার তাই করবে, এটাই হলো সোজা কথায় প্রোগ্রামিং। হয়তো বলতে পারো এখনইতো কম্পিউটার সেটা করে, আমি গান শুনাতে বললে সে শুনিয়ে দেয়, আমি গেম খেলতে চাইলে সে আমার সাথে খেলতে শুরু করে। কিন্তু আসল ব্যপারটা হলো একজন প্রোগ্রামার আগেই কম্পিউটারকে বলে রেখেছে যে তুমি গান শুনতে চাইলে সে যেন শুনিয়ে দেয়। সে যদি বলে রাখতো গেম খেলতে চাইলে পড়তে বসার উপর্যুক্ত দিতে তাহলে কম্পিউটার তাই করতো, তোমার কিছু করার থাকতোনা। প্রোগ্রামার হলো সে যার কথায় কম্পিউটার উঠা-বসা করে। দারুণ একটা ব্যাপার এটা, তাইনা?

কিন্তু তুমি কেন প্রোগ্রামিং শিখবে? বড় বড় কথা বলার আগে সবথেকে প্রথম কারণ আমি বলবো কারণ “প্রোগ্রামিং দারুণ মজার একটি জিনিস!”। কম্পিউটারের সাথে অন্য যন্ত্রের বড় পার্থক্য হলো এটা দিয়ে কতরকমের কাজ করানো যায় তার সীমা নেই বললে খুব একটা ভুল হবেনা। তাই প্রোগ্রামিং জানলে যে কতকিছু করা যায় তার তালিকা করতে বসলে শেষ করা কঠিন। তুমি দিনের পর দিন প্রোগ্রামিং করেও দেখবে জিনিসটা বোরিং হচ্ছেনা, প্রায় প্রতিদিনই নতুন মজার কিছু শিখছো, নতুন নতুন টেকনোলজী আবিষ্কারের সাথে সাথে তুমি আরো অনেক রকম কাজ করতে পারছো অথবা তুমিই করছো নতুন আবিষ্কার! আজ হয়তো জটিল কোনো সমীকরণ সমাধান করার জন্য ফাংশন লিখছো, কাল এসব ভালো লাগছেনা বলে লাল-নীল রঙ দিয়ে একটি অ্যানিমেশন বানাতে বসে গেলে, তোমার সৃষ্টিশীলতার সবটুকুই কাজে লাগাতে পারবে প্রোগ্রামিং এর জগতে।

ছবি: শাহরিয়ার মণ্ডুজর, বিশ্বের সবচেয়ে সম্মানজনক প্রোগ্রামিং প্রতিযোগীতার  
বাংলাদেশি জাজ



*A computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match. – Bill Bryson*

একটি স্কুল-কলেজ পড়য়া ছেলেমেয়ে কম্পিউটার বা মোবাইল দিয়ে কি করে? রাশিয়া-চীনের ছেলেমেয়েরা অনেকেই হয়তো অ্যাসেম্বলিতে কোড লিখে, কিন্তু জরিপ না করেও বলা যায় আমাদের দেশে বেশিভাগই মুভি দেখা, ফেসবুক , গেমস ছাড়া খুব বেশি কিছু করেনা। আসলে কম্পিউটার দিয়ে কি করা যায় তার ধারণাও অনেকের নাই। ছেলে বা মেয়েটিকে প্রোগ্রামিং শিখিয়ে

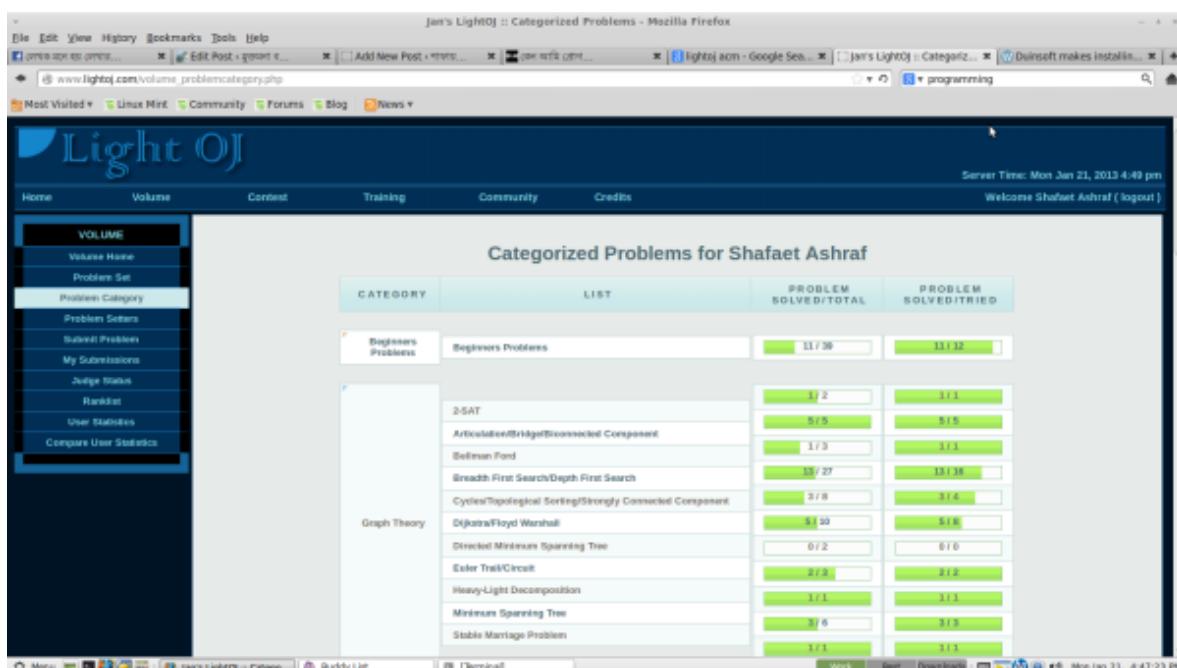
দেয়া হলে তার জগৎটাই পাল্টে যাবে। সে তখন সারাদিন গেমস না খেলে হয়তো একটি গেমস বানিয়ে ফেলবে। আমি বাংলাদেশেরই কিছু স্কুল-কলেজ পড়য়া প্রোগ্রামারদের জানি যারা বাংলা কিবোর্ড নিয়ে কাজ করে, ওপেন সোর্স কমিউনিটিতে অবদান রাখে। প্রোগ্রামিং জানলে তুমি বুঝতে পারবে কম্পিউটার শুধু বিনোদনের যন্ত্র নয়, কম্পিউটার তৈরা করা হয়েছিল এর ক্ষমতাকে ব্যবহার করে বড় বড় গবেষণা, হিসাব করার জন্য, তুমি যদি গবেষণা নাও করো অন্তত এই ক্ষমতাটা ব্যবহার শিখবে, সৃষ্টিশীল অনেক কাজ করতে পারবে। কম্পিউটারের জগতে অসাধারণ কিছু অগ্রগতি হয়েছে খুব কম বয়সী প্রোগ্রামারদের দিয়ে, বিল গেটস স্কুলে থাকতেই চমকে দেয়ার মত কিছু প্রোগ্রাম লিখেছিলেন, প্রোগ্রামিং কন্টেন্টে হাইরেটেড কোডারদের অনেকেই স্কুল-কলেজ এখনও শেষ করেনি।

ছবি: মেহেদি হাসান, তৈরি করেছেন আমাদের সবার প্রিয় অন্ন কিবোর্ড, তিনি মেডিকেলের একজন ছাত্র

প্রোগ্রামিং করা মানে আনন্দের সাথে শেখা। এই শেখাটা খালি কম্পিউটারের মধ্য সীমাবদ্ধ না, অধিকাংশ ভালো প্রোগ্রামারদের খুবই ভালো গাণিতিক এবং লজিকাল জ্ঞান থাকে। দাবা খেলার মতোই প্রোগ্রামিং পুরোটাই লজিকের খেলা, কোন কাজের পর কোনটা করলে কি হবে, কিভাবে করলে আরো দ্রুত ফলাফল আসবে এইসব নিয়ে চিন্তা করতে করতে মাস্টিক্সের লজিকাল সেক্টরটা ডেভেলপ করে। আমার মতে চিন্তা করার মত আনন্দের এবং গুরুত্বপূর্ণ কাজ ২য়টি নেই। বিশেষ করে কম বয়সে প্রোগ্রামিং শিখালে সে চিন্তাশক্তি বৃদ্ধির যেই সুফলটা পাবে সেটা সারাজীবন কাজে লাগবে, সে যদি প্রোগ্রামিং পরে ছেড়েও দেয় তারপরেও চিন্তা করার ক্ষমতাটা থেকে যাবে।

প্রোগ্রামিং কি শুধু কম্পিউটারের সাইন্স যারা পড়ে বা পড়তে চায় তারা শিখবে? সেটার কোনো যুক্তি নেই, তুমি যেই বিষয় নিয়েই পড়ছো বা পড়তে চাও, প্রোগ্রামিং তুমি আনন্দের জন্যই শিখতে পারো এবং চাইলে তোমার কাজেও লাগাতে পারো। তুমি বিজ্ঞানের যেকোনো বিষয়ে লেখাপড়া করলেতো কথাই নেই, তোমার গবেষণায় প্রতি মুহূর্তে কম্পিউটারের লাগবে, তুমি বিজ্ঞেস, আর্টিস পড়লেও প্রোগ্রামিং কাজে লাগবে। তুমি কোম্পানির জন্য দারুণ একটি ওয়েবসাইট বানাতে পারো, একটি সফটওয়্যার বানাতে পারো যেটা যেসব কাজ বোরিং সেগুলো স্বয়ংক্রিয় ভাবে করে দিবে! আমি অনেক সময় ছোটো-খাটো কিন্তু বোরিং কাজ করার সময় চট করে একটা স্ক্রিপ্ট লিখে ফেলি, তারপর সেটাকে কাজ করতে দিয়ে ঘুম দেই!

প্রোগ্রামিং শেখা কি খুব কঠিন? উত্তর হলো হ্যা, যদি তোমার আগ্রহ না থাকে এবং কেও তোমাকে জোর করে শেখায়। যদি একবার মজা পেয়ে যান তাহলে এরপর কারো শেখানো লাগবেনা, নিজেই সব শিখে ফেলতে পারো। আমার উপদেশ হবে ২-৩ সপ্তাহ প্রোগ্রামিং করার পর যদি তোমার ভালো না লাগে তাহলে জোর করে করার দরকার নাই, এটা তোমার জন্য না, অন্য যেটা ভালো লাগে সেই কাজ করো। যদি একবার ভালো লাগে বাজী ধরে বলতে পারি কোড লিখতে লিখতে তুমি প্রায়ই খাবার কথাও ভুলে যাবে। যেকোন কাজের জন্যই সবথেকে গুরুত্বপূর্ণ ব্যাপার হলো ভালো লাগা, যেটা ভালো লাগেনা সেটা করার কোনো অর্থ আমি দেখিনা কারণ দুইদিন পর যা শিখসি সব ভুলে যাবো।



ছবি: lightoj, ঢাকা বিশ্ববিদ্যালয়ের জানে আলম জানের তৈরি করা অনলাইন জাজ যেখানে প্রবলেম সলভ করে সারা পৃথিবীর কোডারা

শুরু কিভাবে করবে? তোমার যদি ইন্টারনেট কানেকশন থাকে তাহলে কথাই নেই, ইন্টারনেটে অসংখ্য টিউটোরিয়াল আছে। ইংরেজীর পাশাপাশী বাংলা কিছু ভালো রিসোর্সও তুমি পাবে। যেমন শুন্দেহ রাগিব হাসানের shikkok.com ওয়েবসাইট বা ফাহিম ভাইয়ের [পাইথন সাইট](#)। এছাড়া খান একাডেমিতেও প্রোগ্রামিং এর ভিডিও আছে, বরাবরের মতই খুবই সুন্দর করে বুঝিয়েছেন সালমান খান। ইন্টারনেট না থাকলে তোমাকে বই জোগাড় করতে হবে, ব্যক্তিগত ভাবে বিগিনারদের জন্য আমি ইন্টারনেটের থেকে বইকেই বেশি গুরুত্ব দিবো। প্রোগ্রামিং এর বইয়ের অভাব নেই দোকানে, তামিম শাহরিয়ার সুবিন ভাইয়ের একটি দারুণ বাংলা বই আছে। তবে একটা ব্যাপারে সতর্ক থাকবে “৭দিনে প্রোগ্রামিং শেখা” এই ধরণের চটকদার বইয়ের বা সাইটের ধারেকাছে যাবে, এগুলো সবকিছু ঝাপসা ভাবে শেখাবে, হার্ভার্ড শিল্ডের বইয়ের মত নামকরা এবং ভালো বই দেখে শিখো, বেসিক জিনিসগুলো পরিষ্কার হবে। এছাড়া লাগবে প্রোগ্রামিং এর জন্য কিছু সফটওয়্যার, এগুলোও সহজেই জোগাড় করতে পারবে। এরপর শুরু করে দাও কোড লেখা!! প্রথম ২ সপ্তাহ আপনার বেশ ঝামেলা লাগবে কারণ বিষয়টা নতুন, একটু পরপর আটকে যাবে, তারপর হঠাৎ দেখবেন সবকিছু সহজ হয়ে গিয়েছে, মূল্যের মধ্যেই ১০০ লাইনের কোড লিখে ফেলেছো। প্রোগ্রামিং শেখার প্রধান শর্ত হলো হাল ছাড়া যাবেনা। প্রথম দিকে কোনো কোড কপি পেস্ট করবেনা, নিজের হাতে লিখবে।

*A good programmer is someone who looks both ways before crossing a one-way street. — Doug Linder, systems administrator*

চাকরী-ক্যারিয়ার নিয়ে সবার মধ্যেই অনেক টেনশন থাকে। আনন্দের জন্য প্রোগ্রামিং শিখলেও এটা তোমার ক্যারিয়ারে খুবই গুরুত্বপূর্ণ। তুমি প্রোগ্রামিং জানলে নিশ্চিত থাকতে পারো কাজের কোনো অভাব জীবনে হবেনা। তুমি কোনো চাকরী না করেও ফ্রি-ল্যান্স কাজ করতে পারবে, এমনকি ছেটোখাট একটা কোম্পানিও খুলে বসতে পারবে। আমি আশেপাশে অনেককে দেখেছি কয়েক বন্ধু মিলে একটি ছোট কোম্পানি খুলে স্বাধীনভাবে কাজ করে, কি দারুণ একটা ব্যাপার। প্রোগ্রামিং জানার আরেকটি দারুণ ব্যাপার হলো তুমি ভালো কোনো কাজ করলে খুব সহজেই সারা বিশ্ব জেনে যাবে। পৃথিবীর আরেক প্রান্তের মানুষ তোমার বানানো সফটওয়্যার দিয়ে গান শুনবে, তোমার অপারেটিং সিস্টেম বুট করবে, আবার পিসি হ্যাং করলে হয়তো আপনাকেই গালি দিবে!! গুগলের মতো কোম্পানিতে কাজ করতে চাইলে তোমার কিছু করতে হবেনা, আপনার কাজের খ্যাতিতে তারাই তোমাকে এসে অফার দিবে। তবে প্রোগ্রামিং শেখার উদ্দেশ্য কখনোই গুগলে চাকরী বা খ্যাতি অর্জন হওয়া উচিত নয়, শিখবে আনন্দের জন্য, জানার জন্য।

সি বা জাভার মতো প্রোগ্রামিং ল্যাংগুয়েজ শেখা মানেই কিন্তু তুমি প্রোগ্রামিং শিখে ফেলোনি। ল্যাংগুয়েজ শেখা খুব সহজ কাজ, প্রথমে একটা কষ্ট করে শিখে ফেললে এরপর যেকোনো ল্যাংগুয়েজ শেখা যায়। তোমাকে খুবই ভালো লজিক ডেভেলপ করতে হবে, অ্যালগোরিদম আর ডাটা স্ট্রাকচার নিয়ে পড়ালেখা করতে হবে, গণিত জানতে হবে, তাহলেই তুমি একজন ভালো প্রোগ্রামার হয়ে উঠবে। তবে ভয়ের কিছু নেই, সবই তুমি ধীরে ধীরে শিখে ফেলতে পারবো, শুধু লাগবে চেষ্টা আর সময়। এটা আশা করবেনা যে ৬ মাসে তুমি অনেক ভালো প্রোগ্রামার হয়ে যাবে তবে লেগে থাকলে ২-৩ বছরে অবশ্যই মোটামুটি ভালো একটা লেভেলে তুমি পৌছাতে পারবে।

তুমি যদি কম্পিউটার সাইন্সের সুট্টেন্ট হও তাহলে এইসব কথাই তুমি হয়তো জানো, শুধু বলবো প্রোগ্রামিং কে আর ৫টা সাবজেক্টের মতো ভেবোনা, খালি সিজিপিএ বাড়াতে কোডিং শিখলে তোমার মতো অভাগা কেও নাই, প্রোগ্রামিং উপভোগ করার চেষ্টা করো, জানার আনন্দে শিখো।

আমার স্বপ্ন আমাদের দেশে একটা চিন্তা করার সংস্কৃতি তৈরি হবে। মানুষ একে অন্যের ব্যক্তিগত ব্যাপারে মাথা ঘামাবেনা, বরং মাথা ঘামাবে গাণিতিক সমস্যা নিয়ে, পাজল নিয়ে, অ্যালগোরিদম নিয়ে। ছেলেমেয়েরা তাদের মেধা গেমস খেলার কাজে না লাগিয়ে কাজে লাগাবে পৃথিবীর উন্নয়নে। বই পড়া, গণিত চর্চা করার পাশাপাশি প্রোগ্রামিং শিখা এই সংস্কৃতি শুরু করতে বিশাল একটি ভূমিকা রাখতে পারে। আমি মনে করি বর্তমান যুগে প্রোগ্রামিং শেখাটা অন্য যেকোন বিষয় শেখার মতই গুরুত্বপূর্ণ, কারণ আমাদের সব কাজে কম্পিউটার লাগে। তাই আপনার আশেপাশের ছেলেমেয়েদের গেমস খেলতে দেখলে তাদের প্রোগ্রামিং সম্পর্কে জানাও, উৎসাহিত করো, অবশ্যই জোর করে শেখানোর কোনো মানে হয়না, যার ভালো লাগবে সে শিখবে তবে সবাই অন্তত জানুক প্রোগ্রামিং কি, এছাড়া কিভাবে শেখার জন্য উৎসাহিত হবে? অনেকেই ইউনিভার্সিটিতে আসার আগে জানেনা প্রোগ্রামিং বলে একটা বন্ধ আছে! আর তোমার প্রোগ্রামিং জানলে অন্যদেরও শিখতে সাহায্য করো, এভাবেই পরিবর্তন একসময় আসবেই, সবাই লজিক দিয়ে ভাবতে শিখবে, চিন্তা করার সংস্কৃতি তৈরি হবে।

শেষ করছি আমার খুব প্রিয় আরেকটি কোটেশন দিয়ে:

*craftsman-ship has its quiet rewards, the satisfaction that comes from building a useful object and making it work. Excitement arrives with the flash of insight that cracks a previously intractable problem. The spiritual quest for elegance can turn the hacker into an artist. There are pleasures in parsimony, in squeezing the last drop of performance out of clever algorithms and tight coding. — steven skiena & miguel reville*

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# কম্পিউটার বিজ্ঞান

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ৮, ২০১৩

“আলকেমিস্ট” বইয়ের রুটিওয়ালা স্বপ্ন দেখতো পর্যটক হবার। কিন্তু সেই স্বপ্নকে সত্যি করার জন্য রাখালবালক হয়ে দেশবিদেশ ঘুরে না বেড়িয়ে সে স্বপ্নকে চাপা দিয়েছিলো কারণ মানুষের চোখে রুটিওয়ালার জীবন রাখালবালকের থেকে বেশি সম্মানের। তাই সমাজকে খুশি করতে গিয়ে তার স্বপ্ন পূরণ করা হয়নি। আমরা বেশিভাগই একেকজন রুটিওয়ালা, নিজের স্বপ্ন পূরণ না করে আগে চিন্তা করি কি নিয়ে পড়ালেখা করলে বেশি টাকার চাকরী পাওয়া যাবে। একটা বিষয়ের “ব” না জেনে ছুট করে বিশ্ববিদ্যালয়ে ভর্তি হয়ে যাই চাকরির আশায়, তারপর একসময় হতাশ ঘিরে ধরে।

কম্পিউটার সায়েন্স খুবই জনপ্রিয় একটা বিষয়। এই বিষয়টা নিয়ে মানুষের মধ্যে অনেক ভূল ধারণা প্রচলিত আছে, এই সাবজেক্টটাতে কি পড়ানো হয় এ বিষয়ে সবার ভালো ধারণা নেই। সবথেকে দুর্ভজনক ব্যাপার হলো এই সাবজেক্টে কি পড়ানো হয় সেটা না জেনেই বহু ছাত্রছাত্রী ভর্তি হয়ে যায় চাকরীর বাজার ভালো এই কারণে, এটা শুধু কম্পিউটার সায়েন্স না বরং বাংলাদেশে যেকোনো সাবজেক্টের জন্যই সত্য। এরপর যেটা হয় সেটা হলো খাপ খাওয়াতে না পেরে অনেকেই হতাশ হয়ে পড়ে।

আমার সৌভাগ্য যে কম্পিউটার সায়েন্স সম্পর্কে ভালো না জানলেও প্রোগ্রামিং আমার আগে থেকেই ভালো লাগতো এবং কম্পিউটার নিয়ে অনেক আগ্রহ ছিলো হোটোবেলা থেকে তাই এখন আনন্দময় সময় কাটাতে পারছি, কিন্তু সবার এই সৌভাগ্য হয়না, এবং এটা শুধু সিস্টেমের দোষ না সাথে নিজেদের অসচেতনতাও বিশাল অংশে দায়ী। আমি অনেক বন্ধুবান্ধবকে দেখেছি ভর্তি পরীক্ষায় ভালো রেজাল্ট করে চাকরীর বাজার দেখে সাবজেক্ট বেছে নিয়ে এখন হতাশ হয়ে পড়েছে।

““We are told from childhood onward that everything we want to do is impossible. We grow up with this idea, and as the years accumulate, so too do the layers of prejudice, fear and guilt. There comes a time when our personal calling is so deeply buried in our soul as to be invisible. But it's still there” – The alchemist, Paulo Coelho”

আমার এই লেখার উদ্দেশ্য কম্পিউটার সায়েন্সে কি কি পড়ানো হয় সেটা সবাইকে জানানো, বিশেষ করে কম বয়েসীদের যারা এখনো কোনো বিষয় বেছে নেয়নি এবং তাদের অভিভাবকদের। ভর্তি পরীক্ষা নিয়ে অনেক গাইডলাইন থাকলেও সাবজেক্ট নিয়ে গাইডলাইনের অভাব আছে। আর যেগুলো আছে সেগুলোতেও কম্পিউটার সায়েন্সে ক্রিয়েতিভিত্তি লাগে, মুখস্থ কম করতে হয় এসব কথাবার্তা শুধু লিখে রাখসে, আসলে কি কি পড়ানো হয় সেটা লিখেনি।

বাংলাদেশের মানুষের এক অংশের ধারণা কম্পিউটার সায়েন্সে ওয়ার্ড, পাওয়ারপয়েন্ট এসব শেখানো হয়, এবং এগুলো যেহেতু পাড়ার দোকানদারও ভালো পারে তাই কম্পিউটার সায়েন্স পড়ার কোনো মানে নাই। আবার কম্পিউটার সম্পর্কে ভালো জানে, টুকটাক প্রোগ্রামিংও কিছুটা জানে এমন মানুষের ধারণা এখানে শুধু ইচটিমিএল, পিএইচপিতে এ ওয়েবসাইট বানানো শেখায়, যেগুলো কম্পিউটার সায়েন্স না পড়লেও শেখা যায়, ফিল্যান্সারদের মধ্যে এই ধারণাটা বেশি কাজ করে।

এখন আমরা তাহলে এক এক করে দেখি কম্পিউটার সায়েন্সে কি কি টপিক পড়তে হবে। টপিকগুলো সম্পর্কে সংক্ষেপে আমরা জানবো। এরপরে নিজেই সিদ্ধান্ত নেয়া যাবে উপরের ধারণা গুলো কতটা সত্যি। টপিকগুলো সম্পর্কে জেনে একজন কাজ হবে তোমার ভেবে দেখা এই সাবজেক্টটা কি সত্যিই তোমার পছন্দ নাকি তুমি ইলেক্ট্রিকাল, মেকানিকাল, ফিজিক্স বা অন্য কোনো সাবজেক্ট পড়বে।

**প্রোগ্রামিং:** কম্পিউটার সায়েন্স বললে অবধারিত ভাবে প্রথমে চলে আসে প্রোগ্রামিং। প্রোগ্রামিং হলো কম্পিউটারকে কথা শুনানোর উপায়, বোকা কম্পিউটারকে দিয়ে ইচ্ছামত কাজ করিয়ে নেয়া। কম্পিউটার যেহেতু মানুষের ভাষা বুঝেনা তাকে বোঝাতে হয় বিশেষ ভাষায় যাকে বলে প্রোগ্রামিং ল্যাংগুয়েজ। প্রোগ্রামিং ল্যাংগুয়েজ শিখে সফটওয়্যার, ওয়েবসাইট যেমন বানানো সন্তু তেমনি গাণিতিক সমস্যা সমাধান করা সন্তু, রকেটের গতিপথ নির্ণয় করা সন্তু, কোয়ান্টাম মেকানিক্স নিয়ে গবেষণা সন্তু, ডিএনএ অ্যানালাইসিস করা সন্তু। এককথায় বলতে গেলে প্রোগ্রামিং এর জ্ঞান আধুনিক যুগের সুপারপাওয়ার যেটা দিয়ে পৃথিবীকে নিয়ন্ত্রণ করা সন্তু।

বাংলাদেশে বেশিভাগ বিশ্ববিদ্যালয়ে প্রথম সেমিস্টারে শিখানো হয় সি ল্যাংগুয়েজ যেটাকে বলা যেতে পারে প্রোগ্রামারদের মাত্রভাষা। এরপরে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং কোর্সে শিখানো হয় জাভা। সফটওয়্যার প্রোগ্রামিং এবং বড় বড়

প্রজেক্টের জন্য জাভা জনপ্রিয়। কোনো কোনো বিশ্ববিদ্যালয়ে জাভা শেখানোর সময় অ্যান্ড্রয়েডে প্রোগ্রামিং শেখানো হয়।

প্রোগ্রামিং ল্যাংগুয়েজ কোর্স দুই ভাগে করানো হয়। ক্লাসরুমে কিছু থিওরিটিকাল কথাবার্তা থাকে তবে এই কোর্সের মূল অংশ হয় ল্যাবে। সেখানে প্রোগ্রামিং ল্যাংগুয়েজ দিয়ে বিভিন্ন সমস্যা সমাধান করতে দেয়া হয়। সবশেষে সাধারণত একটা প্রজেক্ট করতে দেয়া হয়, সেখানে সুটডেন্টরা ক্রিয়েটিভিটি দেখানোর সুযোগ পায়।

কম্পিউটার সায়েন্সের আরো অনেক অংশ থাকলেও ভালো প্রোগ্রামিং জানা এই সাবজেক্টে ভালো করার পূর্বশর্ত।

প্রবর্তীতে অ্যাসেম্বলী ল্যাংগুয়েজের জন্য আরেকটি কোর্স থাকে। এখানে একদম লো লেভেলে সরাসরি মেমরির বিভিন্ন অংশ নিয়ে কাজ করা যায়। অ্যাসেম্বলী ল্যাংগুয়েজ জানতে সাহায্য করে কম্পিউটার কিভাবে মেমরিতে বিট/বাইট হিসাব করে ডাটা রাখে, একদম হার্ডওয়্যার লেভেলে কিভাবে ডাটা রাখা হয় সেটা তুমি জানতে পারবে। অ্যাসেম্বলী ভালোভাবে শিখলে ভাইরাস বানানোর মতো মজার কাজ করা সম্ভব, প্রোগ্রামিং করে মেমরির বিভিন্ন অংশ কন্ট্রোল করা সম্ভব।

**অ্যালগোরিদম:** তুমি কি জানো কিভাবে বড় একটা ফাইলকে কমপ্রেস করে সাইজ কমিয়ে ফেলা সম্ভব হয়? গুগল কিভাবে লাখ লাখ ওয়েবসাইট থেকে চোখের পলকে দরকারি ডাটা খুজে আনে? কিভাবে লাখ লাখ টেরাবাইটের ডিএনএ সিকোয়েন্স থেকে খুজে বের করা হয় জীবনের রহস্য? এধরণের প্রবলেম সলভিং এর হাতেখড়ি হয় অ্যালগোরিদম কোর্স। অ্যালগোরিদম হলো একটি সমস্যাকে সমাধান করার বিভিন্ন ধাপ। অ্যালগোরিদম কোর্সে মূলত শেখানো হয় প্রবলেম সলভিং টেকনিক। সেখানে এধরণের রিয়েল লাইফ প্রবলেম সরাসরি সলভ করা শেখাবেনা, তবে এই কোর্সটা একটা বেস তৈরি করবে, কিছু কমন প্রবলেম সলভিং টেকনিক শিখিয়ে দিবে। কম্পিউটারে মেমরি এবং সময় কম খরচ করে কিভাবে সমস্যা সমাধান করা যায় সেগুলো শেখানো হয় এখানে। এই কোর্সেও ল্যাব এবং থিওরি দুটোই থাকে। এই কোর্সটা করার সময় তুমি বুঝতে পারবে যে শুধু প্রোগ্রামিং ল্যাংগুয়েজ জেনে কোনো লাভ নেই, সেটা ব্যবহার সমস্যা সমাধান করতে জানতে হবে।

**ডাটা স্ট্রাকচার:** ফেসবুক কিভাবে এত মানুষের তথ্য সংরক্ষণ করে? এলোমেলো ভাবে সংরক্ষণ করলে তাড়াতাড়ি খুজে পাওয়া সমস্যা, তাই তথ্য সংরক্ষণ করার নির্দিষ্ট কিছু টেকনিক আছে। দ্রয়ারে লেভেল করে কাগজপত্র রাখলে যেমন সহজে খুজে পাওয়া যায় তেমনি কিছু নির্দিষ্ট স্ট্রাকচার ফলো করে ডাটা সেভ করলে সহজে সেটা কাজের সময় পাওয়া যায়। ডাটা স্ট্রাকচার কোর্স এগুলো সম্পর্কে ধারণা দেয়া হয়।

অ্যালগোরিদম এবং ডাটা স্ট্রাকচার হলো কম্পিউটার সায়েন্সের মূল ভিত্তি, প্রতিটা টপিকে এগুলো কাজে লাগে।

**গণিত:** কম্পিউটার সায়েন্সের সুটডেন্টদের ভালো গণিত জানার দরকার হয়। এটার কারণটা সবার কাছে পরিষ্কার না। নতুন অ্যালগোরিদম বা ডাটা স্ট্রাকচার ডিজাইন করার সময় এগুলো কতটা ভালো কাজ করবে সেটা নির্ধারণ করতে গণিত দরকার হয়। একটা সমস্যা অনেক ভাবে সমাধান করা যায়, কোন পদ্ধতিটা সবথেকে ভালো, কোনটা কম মেমরিতে কম সময়ে কাজ করবে এসব হিসাবের জন্য গণিতের জ্ঞান খুব দরকার। তুমি যত ভালো গণিত জানবে তোমার প্রবলেম সলভিং ক্ষিল তত ভালো হবে। কম্পিউটার সায়েন্সকে এজন্য “অ্যাপ্লাইড ম্যাথ” বলতেও শুনেছি অনেককে।

গণিতের অনেক কঠিন কঠিন সমস্যাও আজকাল কম্পিউটার দিয়ে সলভ করা হয়, এমনকি থিওরেম প্রমাণও করা হয়। গণিতের মধ্যে জানা লাগবে মূলত কম্বিনেটরিক্স, প্রোবাবিলিটি, নাস্বার থিওরি, জিওমেট্রি এবং লিনিয়ার অ্যালজেব্রা ইত্যাদি। এছাড়া কিছু ক্যালকুলাস শেখানো হয়। প্রোবাবিলিটির জন্য আলাদা কোর্স করানো হয় এবং খুবই ইন্টারেস্টিং কিছু প্রবলেম সলভ করানো হয় সেখানে। কারো ঘদি গণিত ভালো লাগে তার জন্য কম্পিউটার সায়েন্সে ভালো করা সহজ হয় যায়। আবার কেও গণিতের রিয়েল লাইফ অ্যাপ্লিকেশন শিখতে চাইলেও এটা তার জন্য একটা ভালো সাবজেক্ট হতে পারে।

**অপারেটিং সিস্টেম এবং সিস্টেম প্রোগ্রামিং:** এখানে শিখানো হয় কিভাবে অপারেটিং সিস্টেম কাজ করে। তবে তারমানে এই না যে উইন্ডোজ চলানো শেখানো হয়! এখানে শিখায় অপারেটিং সিস্টেমের ইন্টারনাল স্ট্রাকচার কিরকম। কম্পিউটারে একই সাথে ১০টা কাজ করলে অপারেটিং সিস্টেমকে ১০টা কাজের জন্য বিশেষ শিডিউল তৈরি করতে হয়, সেগুলো এখানে শিখানো হয়। তারপর ধরো ২টা প্রোগ্রাম একই সাথে প্রিন্টার ব্যবহার করতে চাইলো, কাকে অপেক্ষায় রেখে কাকে দিবে এ ধরণের রিসোর্স ম্যানেজমেন্টের কাজও শিখানো হয়।

সিস্টেম প্রোগ্রামিং এ মুক্তসোর্স অপারেটিং সিস্টেম ইউনিক্স/লিনাক্সের সোর্সকোড নিয়ে ঘাটাঘাটি করা হয়। এই কোর্সটা ভালোমত করলে শেখা যাবে কিভাবে মডেম বা বিভিন্ন ডিভাইসের জন্য ড্রাইভার তৈরি করা যায়, কিভাবে অপারেটিং সিস্টেমের ভিতরের কোড মডিফাই করা যায়। ভবিষ্যতে কেও অপারেটিং সিস্টেম তৈরি করতে চাইলে বা এই সম্পর্কিত রিসার্চ করতে চাইলে সেটা ব্যাকগ্রাউন্ড তৈরি হয় এখানে।

**ডাটাবেস:** ওয়েবসাইট বা বড়বড় সফটওয়্যার বিশাল ডাটা রাখে “ডাটাবেস” এর ভিত্তি। ডাটাবেসে কিভাবে ডাটা রাখতে হয়, কিভাবে সেখান থেকে ডাটা নিয়ে আসতে হয় ইত্যাদি নিয়ে এই কোর্স।

**আর্টিফিশিয়াল ইন্টেলিজেন্স(এ.আই):** নাম শুনেই বোঝা যাচ্ছে কি নিয়ে এই কোর্স, থিসিসের জন্য খুবই পপুলার একটা টপিক এটা। কিভাবে রোবটকে দিয়ে কাজ করানো যায়, গেমের ক্যারেক্টার নিজে নিজে বিভিন্ন সিদ্ধান্ত নেয়, কম্পিউটার কিভাবে দাবা খেলে এই ধরণের দারুণ ইন্টারেস্টিং সব টপিক এ.আই এর অন্তর্ভূত। ল্যাবে প্রোগ্রামিং করে এসব টেকনিক

ইমপ্লিমেন্টও করতে হয়। গেম প্রোগ্রামিং বা রোবোটিকস এ আগ্রহ থাকলে এই কোর্সটি তোমার খুব পছন্দ হবে। নিউরাল নেটওয়ার্ক, জেনেটিক অ্যালগোরিদমের মতো দারুণ সব জিনিস শিখতে পারবে।

**কম্পাইলার:** সাদামাটা ভাষায় কম্পাইলার জিনিসটার কাজ প্রোগ্রামিং ল্যাংগুয়েজকে মেশিন ভাষায় পরিণত করা যাতে হার্ডওয়্যার সেটা বুঝতে পারে। প্রোগ্রামিং ল্যাংগুয়েজ কিভাবে কাজ করে, কিভাবে নিজের প্রোগ্রামিং ল্যাংগুয়েজ তৈরি করা যায় এগুলো নিয়েই কম্পাইলার কোর্স। প্রোগ্রামিং ল্যাংগুয়েজের ইনস্ট্রুকশন গুলো কম্পিউটার কিভাবে মেশিন কোড বানিয়ে কাজ করে সেগুলো শেখানো হয় এখানে। মানুষের ভাষার মতো প্রোগ্রামিং ভাষারও গ্রামার থাকে, সেগুলো নিয়ে জানতে পারবে।

**গ্রাফিক্স:** আমরা কম্পিউটার গেমস বা অ্যানিমেটেড মুভিতে এত সুন্দর গ্রাফিক্স দেখি তার পিছনে আছে প্রচুর ম্যাথমেটিকাল থিওরী। যেমন গেমসে লাইটিং, শেডিং কিভাবে বাস্তবসম্মত করা যায় তার পিছনে আছে অনেক থিওরী। গেমারদের কাছে প্রচলিত শব্দ “এন্টি এলিয়াসিং”, “শেডিং” ইত্যাদি সম্পর্কে এই টপিকে বিস্তারিত পড়ানো হয়। প্রচুর জ্যামিতি দরকার হয় এখানে। যেমন একটা 3-ডি বাক্সকে ৪৫ ডিগ্রী ঘূরালে কো-অর্ডিনেট কর পরিবর্তন হবে এসব হিসাব এখানে করতে হয় এবং ল্যাবে সেই অনুযায়ি গ্রাফিকাল প্রোগ্রাম লিখতে হয়, তাই শেখাটা শুধু মুখস্থে সীমাবদ্ধ থাকেনা। তুমি কলেজে থাকতে হয়তো বৃত্ত, সরলরেখার ইকুয়েশন শিখেছো, এখানে শিখবে ওগুলো ব্যবহার করে কিভাবে গ্রাফিক্স তৈরি হয়।

**নেটওয়ার্কিং:** কিভাবে একটা কম্পিউটারকে আরো ১০ কম্পিউটারের সাথে কানেক্ট করতে হয়, কিভাবে ইন্টারনেট কাজ করে, কিভাবে নেটওয়ার্কে ডাটা প্যাকেট পাঠানো হয় এবং অন্য প্রান্তে রিসিভ করা হয় ইত্যাদি শেখানো হয়। নেটওয়ার্ক সিকিউরিটি এবং এই টপিকের অন্তর্ভুক্ত। জানতে পারবে ল্যান, ডিএনএস সার্ভার, আইপিভি-৬, ক্রিপ্টোগ্রাফি ইত্যাদি সম্পর্কে।

**ডিস্ট্রিবিউটেড সিস্টেম:** তুমি হয়তো জানো বড় বড় যেসব কাজে অনেক মেমরি, শক্তি দরকার হয় সেসব কাজে অনেকগুলো কম্পিউটারকে একসাথে ব্যবহার করে কাজ করা হয়। বড় বড় রিসার্চের কাজ এভাবে করা হয়, ওয়েবসাইটগুলোতেও অনেকগুলো সার্ভার একসাথে কাজ করে। কিভাবে ডিস্ট্রিবিউটেড সিস্টেম ডিজাইন করা যায় সেটা নিয়েই এই কোর্স।

**সফটওয়্যার ইঞ্জিনিয়ারিং:** আধুনিক সফটওয়্যার ডিজাইনের টেকনিক পড়ানো হয় এই টপিকে। সব বিশ্ববিদ্যালয়ের কারিকুলামে সম্ভবত এটা নেই।

এখন আসি হার্ডওয়্যার রিলেটেড কিছু টপিকে। এ ব্যাপারে আগ্রহ এবং জ্ঞান কম বলে কম কথা শেষ করছি। কম্পিউটার সায়েন্স হার্ডওয়্যার নিয়ে অনেক কিছু পড়ানো হয়। কম্পিউটার আর্কিটেকচারে কম্পিউটারে মূল গঠন পড়ানো হয়। কিভাবে কম্পিউটারের তথ্য যাবার পথ বা “ডাটা বাস” কাজ করে, এজিপির সাথে পিসিআই এবং পার্থক্য, ক্যাশ মেমরি ইত্যাদি শেখানো হয়। ইলেক্ট্রিকাল সার্কিটের মতো কম্পিউটারে থাকে বিশেষ ডিজিটাল সার্কিট, সেগুলো নিয়ে পড়ানো হয় ডিজিটাল সিস্টেমস টপিকে। এছাড়া বিভিন্ন ডিভাইস যোগ করলে কিভাবে সেটা কম্পিউটারের সাথে যোগাযোগ করে এসব পড়ানো হয় হার্ডওয়্যার অংশে। ল্যাবে বিভিন্ন ধরণের সার্কিট তৈরি করতে হয়। প্রজেক্ট অংশে অনেক ক্রিয়েটিভিটি দেখানোর সুযোগ থাকে, কেও কেও রোবট তৈরি করে, আমাদের ক্লাসের একজন মোবাইল দিয়ে নিয়ন্ত্রণ করা যায় এমন খেলনা গাড়ি বানিয়েছিলো।

এই হলো মোটামুটি কম্পিউটার সায়েন্সে যা যা পড়ানো হয় তার সামারি। অনেক কিছু হয়তো বাদ পড়ে গিয়েছে এই মূহূর্তে মাথায় না থাকার কারণে। কম্পিউটার সায়েন্সের আকর্ষণীয় একটা দিক হলো “কনটেস্ট”। সারাবছরই বিভিন্ন বিশ্ববিদ্যালয় বা কোম্পানি প্রোগ্রামিং কনটেস্ট এবং সফটওয়্যার কনটেস্ট আয়োজন করে। প্রোগ্রামিং কনটেস্টে মূলত অলিম্পিয়াড স্টাইলে অ্যালগোরিদমের সাহায্যে প্রবলেম সলভ করতে হয়। এখানে সুযোগ আছে সারা বিশ্বের বড় বড় প্রোগ্রামারদের সাথে কনটেস্ট করার। বাংলাদেশের মানুষের গর্ব করার মতো জিনিস খুব বেশি নেই, তবে প্রোগ্রামিং কনটেস্ট অবশ্যই সেই অন্ন জিনিসগুলোর একটা, অনেকবছর ধরেই বাংলাদেশিরা এসব কনটেস্টে ভালো ফল করছে। সফটওয়্যার কনটেস্টে মূলত বিভিন্ন সফটওয়্যার, ওয়েব সাইট ডিজাইন করতে হয়, মোবাইল বিশেষ করে অ্যান্ড্রয়েড ভিত্তিক মোবাইলের সফটওয়্যার কনটেস্টেও বাংলাদেশিরা অংশ নেয়। আবার তুমি চাইলে হার্ডওয়্যার কনটেস্টও করতে পারো, দারুণ একটা রোবট বানিয়ে চমকে দিতো পারো সবাইকে।

আরেকটি আকর্ষণীয় দিক হলো গবেষণার সুযোগ। কম্পিউটার সায়েন্স শেষ বর্ষে কিছু ক্রেডিট থাকে গবেষণা বা প্রজেক্টের জন্য। প্রতি বছরই বাংলাদেশের অনার্সের ছাত্রা ভালো ভালো জার্নালে পেপার প্রেসে প্রেস করে থাকে। কম্পিউটার সাইন্সের গবেষণার একটা সুবিধা হলো নিজের ডেক্টপ কম্পিউটার ব্যবহার করেই বড় বড় গবেষণা করে ফেলা যায়, কোটি টাকা যন্ত্রপাতির দরকার হয় না(অবশ্যই দরকার হয়, তবে সেগুলো ছাড়াও অনেক কাজ করা যায়)।

কম্পিউটার সায়েন্স অনার্স করে তুমি চাইলে অন্যান্য বিষয় নিয়েও পড়ালেখা করতে পারো কারণ এখন সবকাজে কম্পিউটার দরকার হয়।

যেমন রিসেন্টলি অ্যাস্ট্রোইনফরমেটিক্স নামের একটা সাবজেক্ট নিয়ে গবেষণা হচ্ছে, এদের কাজ মহাকাশ গবেষণায় কম্পিউটারকে আরো ভালোভাবে কিভাবে ব্যবহার করা যায় সেটা নিয়ে কাজ করা। তুমি চাইলে ফিজিক্সের লাইনে গিয়ে

কোয়ান্টাম কম্পিউটার নিয়ে কাজ করতে পারো। আবার বায়োলজী ভালো লাগলে বায়োইনফরমেটিক্স নিয়ে পড়ালেখা করতে পারো, কাজ করতে পারবো ন্যানোটেকনোলজী, ডিএনএ/প্রোটিন অ্যানালাইসিস নিয়ে। এরকম হাজারটা সম্ভাবনা তোমার সামনে থাকবে।

আশা করি লেখাটা পড়ার পর কম্পিউটার সায়েন্সে কি পড়ানো হয় সে সম্পর্কে একটা ধারণা তৈরি হয়েছে এবং ভুল ধারণাগুলো ভেঙে দিয়েছে। চাকরির বাজার নিয়ে কিছু বলবোনা, যে আগেই চাকরির চিন্তা করে সাবজেক্ট চয়েস করে তার জন্য এ লেখা নয়, তবে তারপরেও শুধু বলে রাখি বর্তমানে কম্পিউটার সায়েন্স পরে আমি কাওকে বেকার বসে থাকতে দেখিনি।

ভর্তি পরীক্ষার্থীদের বলবো, যেসব সাবজেক্ট পড়ার কথা ভাবছো প্রতিটা সম্পর্কে ভালোভাবে জানার চেষ্টা করো, সেখানে কি কি পড়ানো হয় সেটা খোজ নাও, এবং যদি সেগুলো তোমার সত্যিই পছন্দ হয় তাহলেই ভর্তি হয়ে যাও, কোনো সাবজেক্টই ১ নম্বর-২ নম্বর না, ইলেক্ট্রিকাল মোটেও মেকানিকালের থেকে “ভালো সাবজেক্ট” না, সেটাই তোমার কাছে “ভালো সাবজেক্ট” যেটা তোমার পছন্দ, সেটা যদি পৃষ্ঠিবিজ্ঞানের মতো মানুষের চোখে পিছের দিকের সাবজেক্ট হয় তাহলেও সেটা ভালো, মানুষই তৈরি বিভাজন অনুসরণ করলে নিজের সাথে প্রতারণা করা হয়।

সকল ভর্তি পরীক্ষার্থীদের প্রতি শুভকামনা, আশা করি ভয়াবহ এই সময় সহজে পার করে ফেলতে পারবে।

*“What’s the world’s greatest lie?” the boy asked, completely surprised. “It’s this: that at a certain point in our lives, we lose control of what’s happening to us, and our lives become controlled by fate. That’s the world’s greatest lie.”- The alchemist,Paulo Coelho*

# প্রোগ্রামিং কনটেস্ট এবং অনলাইন জাজে হাতেখড়ি

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

জানুয়ারি ৭, ২০১৩

(আমাকে অনেকে মেইল করে জিজ্ঞাসা করে কিভাবে সে প্রোগ্রামিং কনটেস্টের সাথে যুক্ত হতে পারে? তাদের হয়তো কয়েক বাক্যে কিছুটা বুঝিয়ে বলার চেষ্টা করি কিন্তু সব কথা বা গাইডলাইন এভাবে দেয়া সম্ভব হয়না। এই লেখাটা তাদের জন্য যারা সামান্য প্রোগ্রামিং শিখেছে বা শেখার কথা ভাবছে এবং প্রোগ্রামিং কনটেস্টে অংশ নেবার কথা ভাবছে। আমি যখন প্রথম শুরু করেছিলাম তখন যেসব প্রশ্ন মাথায় এসেছিলো বা অনেকেই যেধরণের প্রশ্ন করে সেগুলোর উত্তর দিতে চেষ্টা করবো এখানে এবং কিভাবে অনলাইন জাজে প্রবলেম সলভ করতে হয় সেটা দেখাবো।)

তোমার বয়স যতই হোক বা যেখানেই পড়ালেখা করো তুমি প্রোগ্রামিং কনটেস্টে অংশ নিতে পারবে, এমনকি তুমি যদি স্কুলে পড়ো তাহলেও। ইনফরমেটিক্স অলিম্পিয়াড হলো স্কুল-কলেজের সুটডেন্টদের জন্য প্রোগ্রামিং কনটেস্ট, বাংলাদেশ জাতীয় পর্যায়ের অলিম্পিয়াডে ভালো করে তুমি অংশ নিতে পারবে আন্তর্জাতিক ইনফরমেটিক্স অলিম্পিয়াডে। তুমি যদি স্কুল-কলেজ থেকেই সিরিয়াসলি কনটেস্ট করা শুরু করো তাহলে খুবই ভালো সম্ভাবনা আছে বিশ্ববিদ্যালয় পর্যায় গিয়ে তুমি দেশের সেরা একজন কনটেস্টেন্ট হতে পারবে, রাশিয়া-চীনের মতো যেসব দেশ কনটেস্টে সবথেকে ভালো তারা খুব কম বয়সে এটা শুরু করে। তুমি যদি এরই মধ্যে কোনো বিশ্ববিদ্যালয়ে ১ম বা ২য় বর্ষে পড়ালেখা করো তাহলেও মোটেও দেরী হয়ে যায়নি, তুমি বিভিন্ন বিশ্ববিদ্যালয়ের আয়োজিত ন্যাশনাল লেভেলের কনটেস্টগুলোতে অংশ নিতে পারবে। তবে তোমার লক্ষ্য হবে আকাশ ছোয়ার, তুমি হয়তো একসময় অংশ নিবে acm icpc ওয়ার্ল্ড ফাইনালে, প্রোগ্রামিং কনটেস্টের সবথেকে সম্মানজনক প্রতিযোগীতায়। এটার জন্য তোমাকে খুবই ভালো ফলাফল করতে হবে রিজিয়নাল কনটেস্টে। প্রতি বছরই বাংলাদেশ থেকে ১টি বা ২টি টিম ওয়ার্ল্ড ফাইনালে অংশ নেয়।

এগুলো ছাড়াও বিভিন্ন ওয়েবসাইটে নিয়মিত অনলাইন কনটেস্ট হয়, সেগুলোতে অংশ নিতে পারে যে কেও, শুধু তোমাকে ইমেইল আইডি দিয়ে সাইটে রেজিস্টার করতে হবে। সেখানে তুমি সারা বিশ্বের প্রোগ্রামারদের সাথে প্রতিযোগীতা করবে, তোমাকে পারফরমেন্স অনুযায়ী গ্র্যান্ডম্যাস্টার/এক্সপার্ট ইত্যাদি রেটিং দেয়া হবে, সেখানে দেখতে পারবে বাংলাদেশি প্রোগ্রামাররা ইন্টারন্যাশনাল পর্যায়েও খুবই ভালো ফলাফল করে। প্রোগ্রামিং কনটেস্ট টপ-রেটেড কোডারদের অনেকের বয়স ১৫-১৬ বা তারও কম তাই তোমার ইউনিভার্সিটিতে যাওয়া পর্বত অপেক্ষা করা দরকার নেই।

এখন প্রথম প্রশ্ন হলো তুমি শুরু করবে কিভাবে? প্রথম কাজ অবশ্যই প্রোগ্রামিং শেখা। তুমি যদি এখনো প্রোগ্রামিং না শিখে থাকো তাহলে আগে সেটা শুরু করে দাও। সি++ এর একটা ভালো বই যোগাড় করো ([এখান থেকে ডাউনলোড](#) করে নিতে পারো) আর বেসিক জিনিসগুলো শিখে ফেলো। প্রথমেই খুব বেশি শেখা দরকার নেই, বেসিক লজিক-লুপ চালানো-ইনপুট/আউটপুট নেয়া শিখেই তুমি প্রবলেম সলভিং প্র্যাকটিস করতে পারবে, একই সাথে শিখতে থাকবে আরো অ্যাডভান্সড টপিকগুলো। ল্যাংগুয়েজ শেখা আসলে কনটেস্টের প্রস্তুতির বড় অংশ না, এটা শিখতে বেশি সময় লাগবেনা, এরপরে অনেক বেশি সময় দিতে হবে প্রবলেম সলভিং টেকনিকগুলো শিখতে তবে সেই সময়টা তোমার একদমই বোরিং লাগবেনা সেটার নিশ্চয়তা দিতে পারি, এই শেখা চলতে থাকবে তুমি যতদিন কনটেস্ট করবে ততদিন, শেখার কোনো শেষ নেই।

প্রোগ্রামিং ল্যাংগুয়েজ ছাড়াও কি কি বিষয় শেখা লাগবে? তুমি প্রোগ্রামিং কনটেস্টে অংশ নেয়ার প্রস্তুতি নিছি, তুমি কোনো ওয়েব ডেভেলপার না শুধু ল্যাংগুয়েজ শিখে গুগলে সার্চ করে ফাংশন কপি-পেস্ট করে কাজ শুরু করবে, বা তুমি পরীক্ষার প্রস্তুতি নিচ্ছনা যে কোনো একটা বই মুখস্থ করে ফেলবে। তুমি প্রোগ্রামিং কনটেস্ট করবে তাই তোমার জানার কোনো লিমিট থাকবেনা। তোমাকে খুবই ভালো লজিক ডেভেলপ করতে হবে, তোমাকে গণিতে বিশেষ করে কম্পিউটেরিস্ট, প্রোবাবিলিটি আর নাস্বার থিওরিতে অবশ্যই ভালো হতে হবে, অনেক অ্যালগোরিদম আর ডাটা স্ট্রাকচার কিভাবে কাজ করে তোমাকে জানতে হবে এবং সেগুলো কোডে ইমপ্লিমেন্ট করতে জানতে হবে। কঠিন কঠিন শব্দগুলো শুনে ভয়ের কিছু নেই, কনটেস্ট করতে করতেই তুমি এগুলো শিখে যাবে, আগেই সবকিছু শিখে কেও শুরু করেনা। তোমাকে এই লেখার শেষ অংশ কিছু রিসোর্স দিবো শুরু করার জন্য।

এখন আসি ঘেটা শিখাতে মূলত লেখাটা শুরু করেছিলাম সেই অংশে, অনলাইন জাজে হাতেখড়ি। তুমি যখন একটা প্রবলেম সলভ করবে সেটা সঠিক নাকি সেটা বলে দিবে অনলাইন জাজ। অনলাইন জাজ আসলে এক ধরণের অনলাইন সফটওয়্যার যে তোমার কোড পরীক্ষা করে কয়েক সেকেন্ডের মাঝে বলে দেয় সেটা সঠিক নাকি। প্র্যাকটিস করার জন্য অসংখ্য অনলাইন জাজ আছে। তুমি হয়তো জানো এইসব জাজের মধ্যে একটি আছে একজন বাংলাদেশির তৈরি করা, জাজটির নাম [lightoj](#), তৈরি করেছেন ঢাকা বিশ্ববিদ্যালয়ের জানে আলম জান যিনি বর্তমানে গুগলে কাজ করছেন। তবে আমরা শুরু করবো বাংলাদেশিদের মধ্যে সবথেকে পপুলার UVa online judge দিয়ে, এটা তৈরি করেছে স্পেনের ভ্যালাডোলিড ইউনিভার্সিটি, লিঙ্কটা হলো <http://uva.onlinejudge.org/>। তুমি শুরুতেই এখানে একটা অ্যাকাউন্ট খুলে ফেলো।

The screenshot shows the UVa Online Judge interface. On the left, the 'Main Menu' includes links for Home, My Account, Contact Us, TOOLS on the Old UVa OJ Site, ACM-ICPC Live Archive, Logout, and Online Judge (with sub-links for Quick Submit, Migrate submissions, My Submissions, My Statistics, My uHunt with Virtual Contest Service, and Browse Problems). The 'Browse Problems' section is highlighted. The main content area displays a table titled 'Title' with columns for 'Total Submissions / Solving %' and 'Total Users / Solving %'. The table lists various contest volumes from Volume C to Volume CXVI. A specific row for 'Volume C' is highlighted.

| Title        | Total Submissions / Solving % | Total Users / Solving % |
|--------------|-------------------------------|-------------------------|
| Volume C     |                               |                         |
| Volume CI    |                               |                         |
| Volume CII   |                               |                         |
| Volume CIII  |                               |                         |
| Volume CIV   |                               |                         |
| Volume CV    |                               |                         |
| Volume CVI   |                               |                         |
| Volume CVII  |                               |                         |
| Volume CVIII |                               |                         |
| Volume CIX   |                               |                         |
| Volume CX    |                               |                         |
| Volume CXI   |                               |                         |
| Volume CXII  |                               |                         |
| Volume CXIII |                               |                         |
| Volume CXIV  |                               |                         |
| Volume CXV   |                               |                         |
| Volume CXVI  |                               |                         |

অ্যাকাউন্ট খুলে তুমি বামে browse problems থেকে “Contest Volumes” লিংকটাতে যাও। সেখানে দেখবে উপরের মতো volume C, volume CI ইত্যাদি লেখা আছে। তুমি volume C তে ক্লিক করো। সেখানে দেখবে প্রবলেমের একটা লিস্ট এসে গিয়েছে।

The screenshot shows the same UVa Online Judge interface, but the 'Browse Problems' section is now active. It displays a table of problems with columns for 'Title', 'Total Submissions / Solving %', and 'Total Users / Solving %'. The table lists 18 problems, each with a green checkmark icon. The last problem listed is '10018 - Reverse and Add'.

| Title                                    | Total Submissions / Solving % | Total Users / Solving % |
|--|-------------------------------|-------------------------|
| 10001 - Longest Paths                    | 23702 / 20.12%                | 5626 / 58.28%           |
| 10001 - Garden of Eden                   | 4092 / 12.09%                 | 1275 / 67.69%           |
| 10002 - Center of Masses                 | 8570 / 24.48%                 | 1815 / 64.85%           |
| 10003 - Cutting Sticks                   | 20999 / 42.36%                | 6143 / 87.11%           |
| 10004 - Bicoloring                       | 34279 / 42.36%                | 9604 / 87.29%           |
| 10005 - Packing polygons                 | 6992 / 11.30%                 | 1679 / 45.40%           |
| 10006 - Carmichael Numbers               | 25252 / 11.30%                | 6503 / 82.79%           |
| 10007 - Count the Trees                  | 5977 / 11.30%                 | 2302 / 69.68%           |
| 10008 - What's Cryptanalysis?            | 21150 / 56.61%                | 9799 / 93.33%           |
| 10009 - All Roads Lead Where?            | 9192 / 11.30%                 | 2507 / 73.75%           |
| 10010 - Where's Waldorf?                 | 20528 / 11.30%                | 5674 / 75.29%           |
| 10011 - Where Can You Hide?              | 3870 / 11.30%                 | 519 / 31.25%            |
| 10012 - How Big Is It?                   | 6703 / 11.30%                 | 1609 / 64.57%           |
| 10013 - Super long sums                  | 39999 / 11.30%                | 7498 / 62.00%           |
| 10014 - Simple calculations              | 7715 / 11.30%                 | 2686 / 65.38%           |
| 10015 - Joseph's Cousin                  | 6211 / 11.30%                 | 2618 / 45.40%           |
| 10016 - Flip-Flop the Squarelotron       | 2221 / 11.30%                 | 899 / 73.75%            |
| 10017 - The Never Ending Towers of Hanoi | 4297 / 11.30%                 | 1334 / 45.40%           |
| 10018 - Reverse and Add                  | 60970 / 11.30%                | 14410 / 85.12%          |

ডানের কলামগুলোতে দেয়া আছে কতজন প্রবলেমটা সলভ করেছে আর কতজন চেষ্টা করেছে। যে প্রবলেম যত বেশি মানুষ সলভ করেছে সেটা তত সহজ হবার সম্ভাবনা বেশি। তুমি নিচে ক্রল করে 10055 – Hashmat the Brave Warrior প্রবলেমটায় ক্লিক করো। প্রবলেমটি সেট করেছেন বাংলাদেশের Shahriar Manzoor, তিনি খুবই বিখ্যাত প্রবলেমসেটার এবং পুরো পৃথিবীতে হাতেগোণা অল্প যে কয়জন মানুষ তিনি ওয়ার্ল্ড ফাইনালের জাজ হতে পেরেছেন তাদের একজন। প্রবলেমটা একবার তুমি রিডিং পরো তারপর আমি ব্যাখ্যা করছি।

যেকোনো প্রবলেমকে ৩টা অংশ ভাগ করা যায়। প্রথম ভাগে থাকে প্রবলেমের বর্ণনা। এরপরে থাকে “Input” অংশ। এই অংশ বলা থাকে তোমাকে কি ধরণের ভ্যালুর জন্য প্রবলেমটা সলভ করতে হবে। “Output” অংশে থাকে কিভাবে আউটপুট প্রিন্ট করতে থাকে। কয়েকটি sample input/output দেয়া হয় বোঝার সুবিধার জন্য। কিন্তু এটা কিন্তু জাজ এর Input না। জাজের একটি ইনপুট ফাইল থাকে, hashmat প্রবলেমটার জন্য সেটা হতে পারে এরকম:

```

1 86 83
2 15 77
3 35 93
4 92 86
5 21 49
6 27 62
7 59 90
8 26 63
9 26 40
10 36 72
11 68 11
12 29 67
13 30 82
14 23 62
15 35 67
16 2 29
17 58 22
18 67 69
19 56 93
20 42 11
21 73 29
22 19 21
23 37 84
24 24 98
25 70 15
26 26 13

```

তবে সত্যিকারের জাজ ইনপুট ফাইলে কি আছে সেটা খালি প্রবলেম সেটাররা জানেন, তোমাকে বা আমাকে সেটা দেয়া হবেনা, তোমাকে খালি বলা হবে ইনপুট ফাইলে কি ধরণের ইনপুট আছে, ইনপুট ফাইল কত বড়, যে সংখ্যা বা স্ট্রিং ইনপুট দেয়া হবে সেগুলো কত বড় ইত্যাদি। প্রতিটি ইনপুটের জন্য তোমাকে আউটপুট প্রিন্ট করতে হবে প্রবলেমে বলা ইনস্ট্রাকশন অনুযায়ী। জাজের কাছে একটা answer ফাইল আছে। তুমি যেসব আউটপুট প্রিন্ট করবে জাজ সেটাকে আরেকটি আউটপুট ফাইলে নিয়ে যাবে। তারপরে answer ফাইলের এর সাথে তোমার কোডথেকে পাওয়া আউটপুট মিলিয়ে দেখবে। এই কারণে তুমি আউটপুটে অতিরিক্ত কোনো কিছু প্রিন্ট করতে পারবেনা, যেমন hashmat প্রবলেমে তুমি প্রতি লাইনে সুন্দর করে “the answer is 10” এভাবে প্রিন্ট করলে হবেনা, যেভাবে বলেছে সেভাবে শুধুমাত্র উত্তরটা প্রিন্ট করতে হবে। তোমাকে ফাইল নিয়ে চিন্তা করতে হবেনা, তুমি সাধারণভাবেই printf,scanf দিয়ে কাজ করবে, ফাইলের ব্যাপার জাজ হ্যান্ডেল করবে।

এখন তুমি একটা সলিউশন লিখে ফেলবে। সলিউশনটা হতে পারে এরকম:

```

1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     long long a,b;
6     while(scanf("%lld%lld",&a,&b)==2)
7     {
8         long long ans=a-b;
9         if(ans<0) ans=ans*-1;
10        printf("%lld\n",ans);
11    }
12 }
13
14

```

আমি সরাসরি কোড না দিয়ে ইমেজ দিলাম যাতে কেও কপি-পেস্ট করে প্রথম সলিউশন লেখার আনন্দ মিস না করে। while এর ভিতরে scanf এর এরকম ব্যবহার হয়তো তুমি আগে দেখেনি, আপাতত জেনে রাখো এভাবে লিখলে যতক্ষণ ২টি সংখ্যা ইনপুট দেয়া হবে ততক্ষণ তোমার কোড কাজ করবে, ফাইল যখন শেষ হবে(EOF=end of file) তখন কোড টার্মিনেট করবে। এই কোডটা লিখে স্যাম্পল ইনপুট গুলো কিবোর্ড দিয়ে ইনপুট দিয়ে দেখো যে আউটপুটের সাথে মিলছে নাকি। মনে রাখবে শুধুমাত্র আউটপুট অংশটা ভিন্ন একটা ফাইল নিয়ে জাজের সাথে মিলিয়ে দেখা হবে।

এবার বামে quick submit এ ক্লিক করো। প্রবলেম আইডি হলো 10055 যেটা প্রবলেমের উপরেই লেখা আছে। ল্যাংগুয়েজ সিলেক্ট করো C++ 4.5.3।

PEGWIKI SPOJ List TopCoder Forums Dynamic Programming ... rist Live Cricket Scores ... CC

**Quick Submit**

Problem ID: 10055

Language:  C++ 4.5.3 - GNU C++ Compiler with options: -lm -lcrypt -O2 -pipe -DONLINE\_JUDGE

Paste your code...

```
#include<iostream>
using namespace std;
int main()
{
    long long a,b;
    while(scanf("%lld%lld",&a,&b)==2)
    {
        long long ans=a-b;
        if(ans<0) ans=ans*-1;
        printf("%lld\n",ans);
    }
    return 0;
}
```

এরপর সাবমিট বাটনে ক্লিক করো। জাজের সার্ভার ঠিকঠাক থাকলে কয়েক সেকেন্ডে জাজ হয়ে যাবে। বামে my submissions এ ক্লিক করো এখন। জাজ হয়ে গেলে দেখাবে “Accepted”। তারমানে তোমার কোড সঠিক, এইমাত্র তুমি অনলাইন জাজে

তোমার প্রথম প্রবলেম সলভ করেছো!! UVa'তে সার্ভারের অনেক সময় সমস্যা করলে verdict আসতে দেরি করে, তখন একটু অপেক্ষা করতে হবে, তবে সেটা খুব কমই হয়। তুমি কোড সাবমিট করার পর যেসব verdict পেতে পারো সেগুলো হলো:

**Wrong answer:** তোমার সলিউশনে ভুল আছে। sample input এর জন্য তোমার কোড সঠিক উত্তর দিলেও জাজের মূল ইনপুট ফাইলের জন্য দিতে পারছেন। যেমন ধরো hashmat প্রবলেমে বলা আছে  $2^{32}-1$  পর্যন্ত ইনপুট দেয়া হবে, সাধারণ integer এর লিমিট হলো  $2^{31}-1$ , তাই long long ব্যবহার না করলে তুমি wrong answer পাবে। wrong answer আসলে চিন্তা করো কোন ইনপুটের জন্য তোমার প্রোগ্রাম কাজ করছেন।

**Time limit exceeded:** প্রবলেমের উপর হয়তো লেখা দেখেছো time limit: 3 seconds, তারমানে সবগুলো ইনপুটের জন্য মোট ৩ সেকেন্ডের মাঝে তোমাকে আউটপুট দিতে হবে, যদি টাইম পার হয়ে যায় তাহলে এই verdict পাবে। তখন চিন্তা করো কিভাবে কোড অপটিমাইজ করা যায়। আমরা সাধারণত ধরে নেই ১ সেকেন্ডে মোটামুটি ১০<sup>৮</sup> পর্যন্ত লুপ চালানো যাবে।

**Run time error:** তোমার কোড এক্সেকিউশন করার সময় কোনো একটা কারণে বন্ধ হয়ে গিয়েছে। যেমন হয়তো কোথাও তুমি শুণ্য দিয়ে কাওকে ভাগ করার চেষ্টা করছো বা তোমার অ্যারের সাইজ বেশি ছোটো।

**Presentation error:** তুমি যদি কোডে অতিরিক্ত space প্রিন্ট করো তাহলে এই verdict পাবে। তারমানে তোমার সলিউশন ঠিকই আছে তবে প্রিন্টিং ফরমেট ঠিক নাই। সাধারণত খুব সহজেই এই ইরোর দূর করা সম্ভব।

**Accepted:** তোমার সলিউশন সঠিক, অভিনন্দন, এখন অন্য আরেকটি প্রবলেম সলভ করো।

এখন তুমি বামে My uHunt with Virtual Contest Service লিংকে ক্লিক করো। সিঙ্গাপুর ন্যশনাল ইউনিভার্সিটির ফেলিক্স হালিম এই সাইটটা তৈরি করেছেন। এখানে খুব সহজেই তোমার সব statistics দেখতে পারবে, সাবমিট করার পর চাইলে এখানেই তুমি verdict দেখতে পারো। নিচের দিকে "Next Problems to Solve" অংশ থেকে তুমি সলভ করার মতো প্রবলেম খুজে নিতে পারবে, প্রথম দিকে সহজ যত প্রবলেম আছে সব সলভ করে ফেলবে, এরপর দক্ষতার বাড়াতে কঠিন প্রবলেম সলভ করা শুরু করবে। "World Ranklist" অংশে দেখতে পারবে তোমার অবস্থান কোথায়। উপরের দিকে search problems ঘরে প্রবলেম id বিসিয়ে খুব সহজে যেকোনো প্রবলেমের লিংক খুজে বের করতে পারবে। এছাড়া এই সাইটে আরো কিছু জাজের টুলস আছে, একটু গুতাগুতি করে বুঝে নিবে।

প্রথম দিকে তোমার input/output ফরমেট নিয়ে একটু সমস্য হতে পারে। eof পর্যন্ত ইনপুট নেয়া, blank line পেলে ব্রেক করা এসব ব্যাপার নতুন লাগতে পারে। এজন্য তুমি এই ফাইলটা ডাউনলোড করে নাও:

### ইনপুট আউটপুট টেকনিক+অ্যারে+স্ট্রিং+ডাটাটাইপ+পয়েন্টার

এখানে কয়েকটা ছোটোছোটো pdf ফাইল আছে, প্রবলেম সলভিং শুরু করার জন্য খুবই কাজের এগুলো।

যে বইগুলো তোমার পড়া খুবই দরকার সেগুলো হলো knuth এর [concrete math](#), rosen এর discrete math, কোরম্যান বা শাহনীর [algorithm](#), স্টিভেন-ফেলিক্স হালিমের [competitive programming](#)। বইগুলো ডাউনলোড করে ধীরে ধীরে পড়তে থাকো, বিশেষ করে শেষ বইটা।

প্রবলেম সলভ করতে গিয়ে অনেক সময়ই তুমি আটকে যাবে, বারবার wrong answer থাবে। তখন তুমি কি করবে? প্রথম কাজ হলো আরো ভালোভাবে চিন্তা করো। এরপরেও না পারলে সাহায্য নাও। হতে পারে তুমি এমন জায়গায় থাকো যেখানে সাহায্য করার মতো শিক্ষক বা বড় ভাই নাই। ইন্টারনেটের যুগে এটা কোনো ব্যপারই না, তুমি বিভিন্ন ফোরামের সাহায্য নাও। তুমি জান ভাইয়ের [lightoj](#) তে রেজিস্টার করলে একটি চমৎকার ফোরাম পাবে যেখানে অনেকেই সাহায্য করবে। uva সহ প্রায় সব অনলাইন জাজেরই ফোরাম আছে, তুমি গুগলে সার্চ করলেই লিংক পাবে সেগুলোর। সেখানে তুমি বড় বড় প্রোগ্রামারদের সাথে কথা বলো, দেখো তারা কিভাবে প্র্যাকটিস করে। বাংলাদেশের প্রোগ্রামারদের সাথে যোগাযোগ করার জন্য চমৎকার জায়গা হতে পারে ফেসবুক, তুমি নিচের গুরুপ গুলোতে জয়েন করো:

[প্রোগ্রামিং-বাংলা ইনফরমেটিক্স ব্লগ](#)

[BD programmers](#)

[Bangladesh Informatics Olympiad](#)

এসব জায়গায় তুমি সাহায্য করার মতো অনেক উদার মানুষকে পাবে। [ফাহিম ভাইয়ের সাইটে](#) তুমি দারুণ সব রিসোর্স পাবে, বিভিন্ন অনলাইন জাজের কোনটার কি ফিচার আছে সেগুলো জানতে পারবে। তোমার যেসব লিংক দরকার হতে পারে তার একটি কালেকশন [আমি তৈরি করেছি](#), আশা করি তোমার কাজে লাগবে। তুমি uva'র সহজ প্রবলেমের একটা তালিকা পাবে এখানে, এগুলো দিয়ে সলভ শুরু করতে পারো।

বুঝতেই পারছো তুমি যেখানেই থাকোনা কেনো প্রোগ্রামিং কনটেস্টে ভালো করার মতো সব রিসোর্স ইন্টারনেটেই আছে, তোমার কাজ হলো সেগুলো ব্যবহার করে খুব ভালো করে প্র্যাকটিস করা। অনেকবার তোমার মনে হবে "আমি পারছিনা", এরকম ভেবে অনেকেই হাল ছেড়ে দেয়, তুমি ছাড়বেনা, একসময় তুমি ভালো করতে বাধ্য। নেটে তুমি অনেক কোড পাবে যেগুলো সাবমিট

করে তুমি accepted পেতে পারো কিন্তু ভুলেও সেই কাজ করবেনা, তাহলে বিশাল ক্ষতি হবে, তোমাকে এ ব্যাপারে নিজের কাছে সৎ থাকতে হবে। কিছুটা দক্ষতা আসার পর বেশি প্রবলেম সলভ করার থেকে বেশি গুরুত্ব দিবে ভালো প্রবলেম সলভ করাকে, অনলাইন জাজে তুমি কয়টা প্রবলেম সলভ করেছো সেটা কেও দেখবেনা, দেখবে তুমি বিভিন্ন কনটেস্টে কেমন করো সেটা।

সবশেষে বলতে চাই তোমার চলার পথ হয়তো খুব সহজ হবেনা, অনেকেই নিরুৎসাহিত করবে, অনেক শিক্ষক পছন্দ করবেনা তোমার কনটেস্ট করা, অনেক সময় অন্য কাজের সাথে তাল মিলিয়ে কনটেস্ট করা কঠিন হবে, এগুলোই বাস্তবতা বাংলাদেশে। এসবের মধ্যে কেও কেও হাল ছেড়ে দিবে, বাকিরা কোনো কিছুতেই হাল ছাড়বেনা, জয় একমাত্র তাদেরই প্রাপ্য, তোমাকে বেছে নিতে হবে তুমি কোন দলে থাকবে। আমি কনটেস্ট শুরু করি ২০১০ এ, বেশ কিছু ন্যাশনাল আর রিজিয়নাল কনটেস্ট করেছি, কোনোভাবেই আমাকে সিনিয়র কনটেস্টেন্ট বা খুব ভালো কনটেস্টেন্ট বলা যায়না, তবে এই ৩ বছরে অনেক অভিজ্ঞতা হয়েছে, অনেক ভালো ভালো কনটেস্টেন্টের সাথে কাজ করার সুযোগ হয়েছে। অনেক সময়ই ranklist এ পিছের দিকে ছিটকে পড়েছি আবার পরেরবার সেটা রিকভার করে প্রথমদিকেও এসেছি, এখনও চেষ্টা করছি ইম্প্রুভ করার, সবমিলিয়ে কনটেস্ট করা একটা দারুণ অভিজ্ঞতা। তুমি অনেক অনেক কিছু শিখতে পারবে, অ্যালগোরিদমের বাইরেও তুমি শিখবে কিভাবে এক্সট্রিম প্রেশারে কাজ করতে হয়, খারাপ সিচুয়েশন থেকে রিকভার করতে হয়, কিভাবে প্রতিযোগীতা করতে হয় দেশের সেরাদের সাথে। তুমি হয়তো কখনো টেলিভিশনে ক্রিকেট খেলা দেখতে দেখতে শেষ ওভারে দম বন্ধ করে বসে থেকেছো, তারপর একটা ছক্কা মারার পর সবাইকে চমকে চিঢ়কার করেছো, তুমি ঠিক সেই অভিজ্ঞতা পাবে কনটেস্টে, হয়তো শেষ মিনিটে একটি প্রবলেম মিলিয়ে হারিয়ে দিবে সবাইকে, কখনো অনলাইন কনটেস্টে বাংলাদেশের পতাকা তুলে ধরবে অনেক উপরে, এই অসাধারণ অনুভূতি তুমি টাকা দিয়ে কিনতে পারবেনা।

আপাতত এখানেই শেষ করছি। তোমার আরো কোনো প্রশ্ন থাকলে অবশ্যই জানাও, তোমাকে সাহায্য করতে পারলে খুশি হবো। শুভকামনা থাকলো।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# HANDBOOK OF ALGORITHMS

## Section Algorithm Notations

*Courtesy of*  
*Shafaet Ashraf*

অ্যালগোরিদিম কম্প্লকেস্টিভিগি “O” নথেটশেন) \_ শাফায়তেরে ব্লগ.pdf

কম্প্লকেস্টিক্লাস(P-NP, টুরং মশেন ইত\_যাদী) \_ শাফায়তেরে ব্লগ.pdf

হাল্টিং প্ৰৱলমে \_ শাফায়তেরে ব্লগ.pdf

# অ্যালগোরিদম কমপ্লেক্সিটি(বিগ “O” নোটেশন)

 shafaetsplanet.com/planetcoding/

শাফায়েত

10/12/2012

তুমি যখন একটা অ্যালগোরিদমকে কোডে ইমপ্লিমেন্ট করবে তার আগে তোমার জানা দরকার অ্যালগোরিদমটি কতটা কার্যকর। অ্যালগোরিদম লেখার আগে নিজে নিজে কিছু প্রশ্নের উত্তর দিতে হবে,যেমন:

১. আমার অ্যালগোরিদম কি নির্ধারিত সময়ের মধ্যে ফলাফল এনে দিবে?
২. সর্বোচ্চ কত বড় ইনপুটের জন্য আমার অ্যালগোরিদম কাজ করবে?
৩. আমার অ্যালগোরিদম কতখানি মেমরি ব্যবহার করছে?

আমরা অ্যালগোরিদমের কমপ্লেক্সিটি বের করে প্রথম দুটি প্রশ্নের উত্তর দিতে পারি। একটি অ্যালগোরিদম যতগুলো “ইনস্ট্রাকশন” ব্যবহার করে কাজ করে সেটাই সোজা কথাই সেই অ্যালগোরিদমের কমপ্লেক্সিটি। দুটি নম্বর গুণ করা একটি ইনস্ট্রাকশন, আবার একটি লুপ ১০০ বার চললে সেখানে আছে ১০০টি ইনস্ট্রাকশন। ফলাফল আসতে কতক্ষণ লাগবে সেটা সিপিউর প্রসেসরের উপর নির্ভর করবে, কমপ্লেক্সিটি আমাদের cputime বলে দিবেনা, কমপ্লেক্সিটি আমাদের বলে দিবে আমাদের অ্যালগোরিদমটি তুলনামূলকভাবে কতটা ভালো। অর্থাৎ এটা হলো অ্যালগোরিদমের কার্যকারিতা নির্ধারণের একটা ক্ষেত্র। আর BIG O নোটেশন হলো কমপ্লেক্সিটি লিখে প্রকাশ করার নোটেশন।

নতুন প্রোগ্রামিং কনটেস্টেন্টরা প্রায়ই কমপ্লেক্সিটির ব্যাপারে খেয়াল না করেই ব্র্যাট-ফোর্স অ্যালগোরিদম লিখে ফেলে এবং কোড time limit exceed করে। আর যারা শুধুমাত্র সফটওয়্যার নিয়ে কাজ করে তাদের মধ্যে কমপ্লেক্সিটি নিয়ে চিন্তা করার প্রবণতা আরো কম। এই লেখাটা বিগিনার প্রোগ্রামারদের জন্য যারা এখনো সঠিকভাবে কমপ্লেক্সিটি হিসাব করা শিখেন।

আমরা একটা উদাহরণ দিয়ে শুরু করি। আমাদের একটি ফাংশন আছে যার নাম myAlgorithm,আমরা সেই ফাংশনের কমপ্লেক্সিটি বের করবো। মনে করো ফাংশনটি এরকম:

C++

```
1 int myAlgorithm1(int n)
2 {
3     int x=n+10;
4     x=x/2;
5     return x;
6 }
```

এই অ্যালগোরিদমটি nn এর ভ্যালু যাই হোকনা কেন সবসময় একটি constant সংখ্যক ইনস্ট্রাকশন নিয়ে কাজ করবে। কোডটিকে মেশিন কোডে পরিণত করলে যোগ-ভাগ মিলিয়ে ৩-৪ ইনস্ট্রাকশন পাওয়া যাবে,আমাদের সেটা নিয়ে ম্যাথব্যাখার দরকার নাই। প্রসেসর এত দ্রুত কাজ করে যে এত কম ইনস্ট্রাকশন নিয়ে কাজ করতে যে সময় লাগে সেটা নিয়ে আমরা চিন্তাই করিনা,ইনস্ট্রাকশন অনেক বেশি হলে আমরা চিন্তা করি,আর লুপ না থাকলে সাধারণত চিন্তা করার মত বেশি হয়না।

অ্যালগোরিদমের কমপ্লেক্সিটি হলো  $O(1)$ ,এর মানে হলো ইনপুটের আকার যেমনই হোকনা কেন একটি constant টাইমে অ্যালগোরিদমটি কাজ করা শেষ করবে।

এবার পরের কোডটি দেখ:

C++

```

1 int myAlgorithm2(int n)
2 {
3     int sum=0;
4     for(int i=1;i<=n;i++)
5     {
6         sum+=i;
7         if(sum>=1000) break;
8     }
9     return sum;
10 }
```

এই কোডে একটি লুপ চলছে এবং সেটা  $n$  এর উপর নির্ভরশীল।  $n=100$  হলে লুপ 100 বার চলবে। লুপের ভিতরে বা বাইরে কয়টি ইনস্ট্রাকশন আছে সেটা নিয়ে চিন্তা করবোনা, কারণ সেটার সংখ্যা খুবই কম। উপরের অ্যালগোরিদমের কমপ্লেক্সিটি  $O(n)$  কারণ এখানে লুপটি  $n$  বার চলবে। তুমি বলতে পারো  $sum$  যদি 1000 এর থেকে বড় হয় তাহলে `break` করে দিচ্ছ,  $n$  পর্যন্ত চলার আগেই লুপটি `break` হয়ে যেতে পারে। কিন্তু প্রোগ্রামাররা সবসময় worst case বা সবথেকে খারাপ কেস নিয়ে কাজ করে! এটা ঠিক যে লুপটি আগে `break` করতেই পারে, কিন্তু worst case এ সেটা  $n$  পর্যন্তইতো চলবে।

worst case এ যতগুলো ইনস্ট্রাকশন থাকবে সেটাই আমাদের কমপ্লেক্সিটি!

C++

```

1 int myAlgorithm3(int n)
2 {
3     int sum=0;
4     for(int i=1;i<=n;i++)
5     {
6         for(int j=i;j<=n;j++)
7         {
8             sum+=(i+j);
9         }
10    }
11    return sum;
12 }
```

উপরের কোডে ভিতরের লুপটা প্রথমবার  $n$  বার চলছে, পরেরবার  $n-1$  বার। তাহলে মোট লুপ চলছে  $n+(n-1)+(n-3)+\dots+1=n\times(n+1)/2=(n^2+n)/2$  বার।  $n^2$  এর সাথে  $n^2+nn^2+n$  এর তেমন কোনো পার্থক্য নেই। আবার  $n^2/2n^2/2$  এর সাথে  $n^2$  এর পার্থক্যও খুব সামান্য। তাই কমপ্লেক্সিটি হবে  $O(n^2)$ ।

কমপ্লেক্সিটি হিসাবের সময় constant factor গুলোকে বাদ দিয়ে দিতে হয়। তবে কোড লেখার সময় constant factor এর কথা অবশ্যই মাথায় রাখতে হবে।

উপরের তিনি অ্যালগোরিদমের মধ্যে সবথেকে সময় কম লাগবে কোনটির? অবশ্যই  $O(1)$  এর সময় কম লাগবে এবং  $O(n^2)$  এর বেশি লাগবে। এভাবেই কমপ্লেক্সিটি হিসাব করে অ্যালগোরিদমের কার্যকারিতা তুলনা করা যায়। পরের কোডটি দেখো:

C++

```

1 int myAlgorithm4(int n,int *val,int key)
2 {
3     int low=1,high=n;
4     while(low<=high)
5     {
6         int mid=(low+high)/2;
7         if(key<val[mid]) low=mid-1;
8         if(key>val[mid]) high=mid+1;
9         if(key==val[mid]) return 1;
10    }
11    return 0;
12 }

```

এটা একটা বাইনারি সার্চের কোড। প্রতিবার  $\text{low}+\text{high}=n+1$  বা  $n$  এর মান দুই ভাগে ভাগ হয়ে যাচ্ছে। একটি সংখ্যাকে সর্বোচ্চ কতবার ২ দিয়ে ভাগ করা যায়? একটি হিসাব করলেই বের করতে পারবে সর্বোচ্চ ভাগ করা যাবে  $\log_2 n \log_2 n$  বার। তারমানে  $\log_2 n \log_2 n$  ধাপের পর লুপটি ব্রেক করবে। তাহলে কমপ্লেক্সিটি হবে  $O(\log_2 n)O(\log_2 n)$ ।

এখন ধরো একটি অ্যালগোরিদমে কয়েকটি লুপ আছে, একটি  $n^4 n^4$  লুপ আছে, একটি  $n^2 n^2$  লুপ আছে আর একটি  $\log n \log n$  লুপ আছে। তাহলে মোট ইনস্ট্রুকশন:  $n^4 + n^3 + \log n n^4 + n^3 + \log n$  টি। কিন্তু  $n^4 n^4$  এর তুলনায় বাকি টার্মগুলো এতো ছোটে যে সেগুলোকে বাদ দিয়েই আমরা কমপ্লেক্সিটি হিসাব করবো,  $O(n^4)O(n^4)$ ।  
রিকার্সিভ ফাংশনে depth এর উপর কমপ্লেক্সিটি নির্ভর করে, যেমন:

C++

```

1 int myAlgorithm5(int n)
2 {
3     if(n==1) return 1;
4     return n*myAlgorithm5(n-1);
5 }

```

এই অ্যালগোরিদমে সর্বোচ্চ depth হলো  $n$ , তাই কমপ্লেক্সিটি হলো  $O(n)O(n)$ । নিচে ছোট করে আরো কিছু উদাহরণ দিলাম:

|  |
|--|
| $f(n)f(n)=\text{ইনস্ট্রুকশন সংখ্যা}$   |
| $f(n)=n^2+3n+112$  |
| $f(n)=n^2+3n+112$ হলে কমপ্লেক্সিটি $O(n^2)O(n^2)$ /  |
| $f(n)=n^3+999n+112$  |
| $f(n)=n^3+999n+112$ হলে কমপ্লেক্সিটি $O(n^3)O(n^3)$ /  |
| $f(n)=6\times\log(n)+n\times\log n$  |
| $f(n)=6\times\log(n)+n\times\log n$ হলে কমপ্লেক্সিটি $O(n\times\log n)O(n\times\log n)$ /              |
| $f(n)f(n) = 2n+n^2+1002n+n^2+100$  |
| $f(n)f(n) = 2n+n^2+1002n+n^2+100$ হলে কমপ্লেক্সিটি $O(2n)O(2n)$ / (এটাকে exponential কমপ্লেক্সিটি বলে) |

বিগিনারদের আরেকটি কমন ভুল হলো এভাবে কোড লেখা:

C++

```

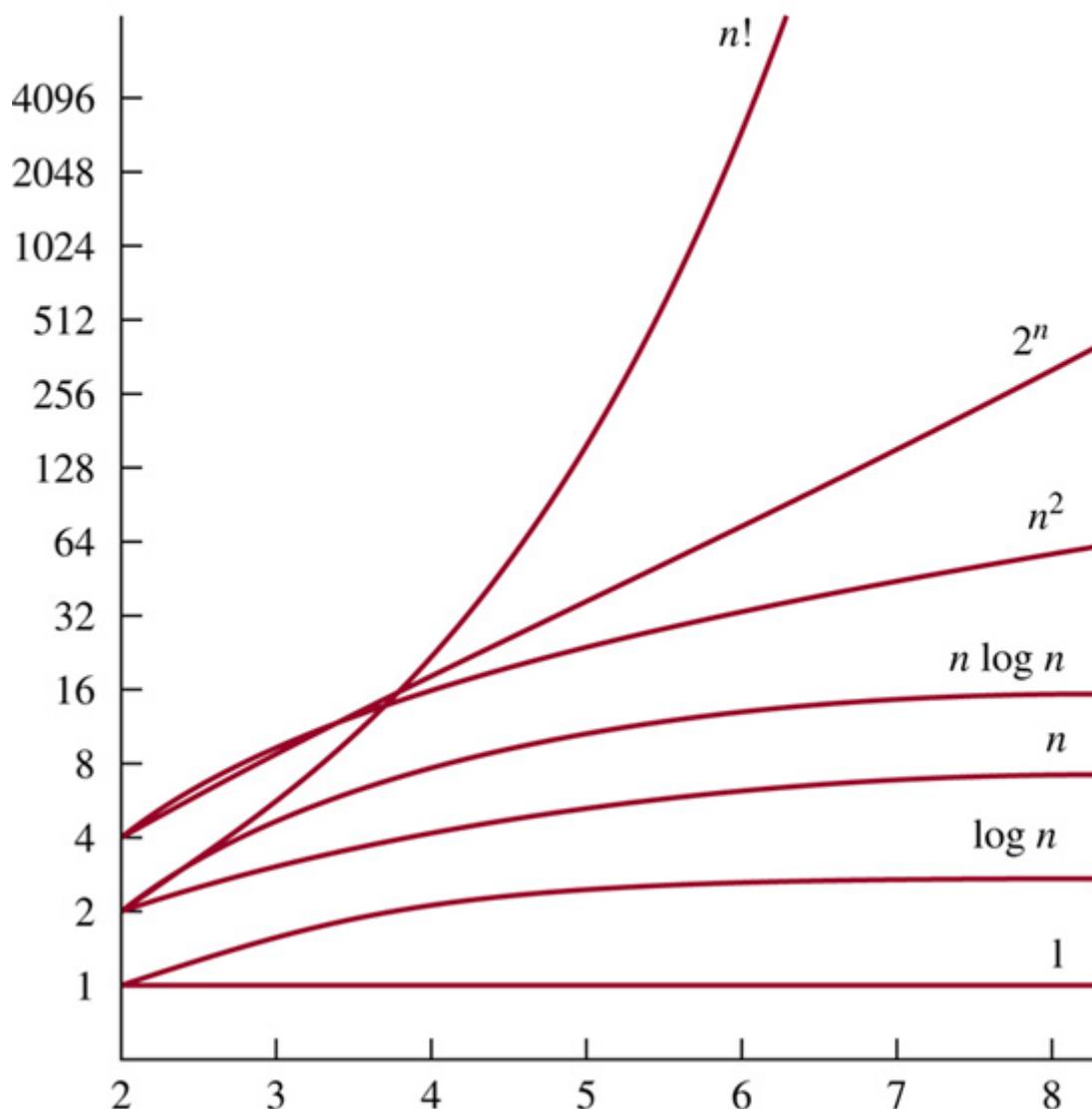
1 int myAlgorithm6(char *s)
2 {
3     int c=0;
4     for(int i=0;i<strlen(s);i++)
5     {
6         if(s[i]=='a') c++;
7     }
8     return c;
9 }
```

s স্ট্রিং এর দৈর্ঘ্য  $|s|$  হলে এখানে কমপ্লেক্সিটি হলো  $O(|s|^2)O(|s|^2)$ । কেন ক্ষয়ার হলো? কারণ  $\text{strlen}(s)$  ফাংশনের নিজের কমপ্লেক্সিটি হলো  $O(|s|)O(|s|)$ , একে লুপের মধ্যে আরো  $O(|s|)O(|s|)$  বার কল করা হয়েছে। তাই  $\text{strlen}(s)$  এর মান আগে অন্য একটি ভ্যারিয়েবলের রেখে তারপর সেটা দিয়ে লুপ চালাতে হবে, তাহলে  $O(|s|)O(|s|)$  এ লুপ চলবে।

প্রোগ্রামিং কন্টেস্টে আমরা ধরে নেই জাজ এর পিসি ১ সেকেন্ডে মোটামুটি 108108 টা ইনস্ট্রাকশন রান করতে পারবে। এটা জাজ-পিসি অনুসারে কমবেশি হতে পারে, যেমন টপকোডার আরো অনেক বেশি ইনস্ট্রাকশন ১ সেকেন্ডে রান করতে পারে কিন্তু spoj বা কোডশেফ তাদের পেন্টিয়াম ৩ পিসি দিয়ে 107107 টাও সহজে রান করতে পারেন। অনসাইট ন্যশনাল কন্টেস্টে আমরা ১ সেকেন্ডে 108108 ধরেই কোড লিখি। কোড লেখার আগে প্রথমে দেখবে তোমার অ্যালগোরিদমের worst case কমপ্লেক্সিটি কত এবং টেস্ট কেস কয়টা এবং দেখবে টাইম লিমিট কত। অনেক নতুন প্রোগ্রামার অ্যালগোরিদমের কমপ্লেক্সিটি সঠিক ভাবে হিসাব করলেও টেস্ট কেস সংখ্যাকে গুরুত্ব দেয়না, এ ব্যাপারে সতর্ক থাকতে হবে।

নিচের গ্রাফে বিভিন্ন কমপ্লেক্সিটির অ্যালগোরিদমের তুলনা দেখানো হয়েছে:

© The McGraw-Hill Companies, Inc. all rights reserved.



(x axis=input size, y axis=number of instructions)

কনটেস্টে প্রবলেমের ইনপুট সাইজ দেখে অনেক সময় expected algorithm অনুমান করা যায়। যেমন  $n=100$  হলে সম্ভাবনা আছে এটা একটা  $n^3$  কমপ্লেক্সিটির ডিপি প্রবলেম, বা ম্যাক্সিমাম-ম্যাচিং প্রবলেম।  $n=10^5$  হলে সাধারণ  $n \log n$  কমপ্লেক্সিটিতে প্রবলেম সলভ করতে হয় তাই সম্ভাবনা আছে এটা একটা বাইনারি সার্চ বা সেগমেন্ট ট্রি এর প্রবলেম।

আজ এই পর্যন্তই। কমপ্লেক্সিটি নিয়ে আরো অনেক কিছু জানার আছে, বিস্তারিত পড়তে:

[A Gentle Introduction to Algorithm Complexity Analysis](#)

[Complexity and Big-O Notation](#)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# কমপ্লেক্সিটি ক্লাস(P-NP, টুরিং মেশিন ইত্যাদি)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ১৪, ২০১৩

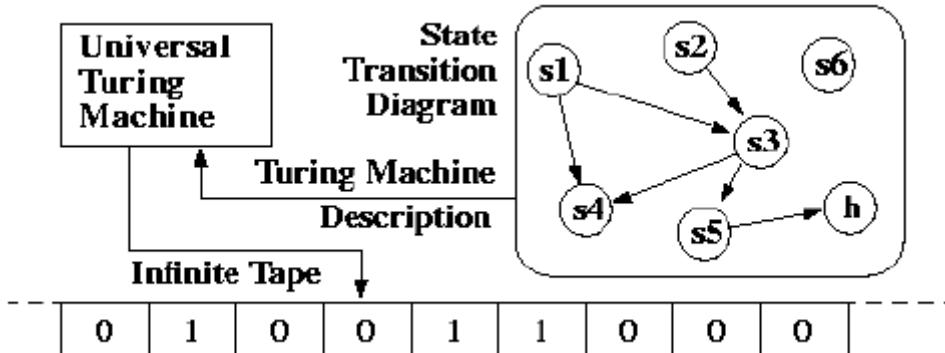
আমরা যখন প্রবলেম সলভ করতে করতে মাথার চুল ছিঁড়ে ফেলি তখন প্রায়ই এমন কিছু প্রবলেম সামনে আসে যেগুলো সলভ করতে গেলে সবথেকে শক্তিশালী কম্পিউটারও হাজার হাজার বছর লাগিয়ে দিবে। বড় বড় কম্পিউটার বিজ্ঞানীরা যখন দিন-রাত চিন্তা করেও এগুলো সলভ করতে পারলেননা তখন তারা এই প্রবলেমগুলোকে কিছু ক্যাটাগরিতে ফেলে দিয়ে বললেন “এই ক্যাটাগরির প্রবলেমগুলো সলভ করার সাধ্য এখনো আমাদের হয়নি, তোমরা কেও পারলে সলভ করে দিয়ে ১০ লক্ষ ডলার পুরস্কার নিয়ে যাও”। সেগুলোকেই আমরা NP-complete, NP-hard ইত্যাদি নামে চিনি।

NP ক্যাটাগরির প্রবলেমগুলো কম্পিউটার সায়েন্সে খুব বিখ্যাত। যেমন একটা গ্রাফে সবথেকে লম্বা পথ খুজে বের করা, ৩টা রঙ দিয়ে গ্রাফ কালারিং করা ইত্যাদি। এগুলোকে এখন পর্যন্ত কেও পলিনমিয়াল টাইমে সলভ করতে পারেনি, যেসব সলিউশন বের হয়েছে সেগুলো সামান্য বড় ইনপুটের জন্যই কাজ করেনা। মজার ব্যাপার হলো এই প্রবলেমগুলোর যেকোন একটা যদি তুমি সলভ করতে পারো তাহলে সবগুলো প্রবলেমই সলভ করা যাবে তোমার সলিউশনের সাহায্যে, অনেক বছর ধরে মানুষকে ভাবানো কঠিন সব প্রবলেম চোখের নিমিষে সলভ হয়ে যাবে একটা মাত্র সলিউশন দিয়ে, কম্পিউটার বিজ্ঞানের চেহারাটাই পাল্টে দিবে এটা। সেজন্যই এত বড় বড় পুরস্কার ঘোষণা করা হয়েছে NP প্রবলেম সলভ করা জন্য।

এই আর্টিকেলটা পড়ার আগে তোমাকে **অ্যালগোরিদম কমপ্লেক্সিটি** নিয়ে জানতে হবে। এছাড়া NP নিয়ে জানতে হলে আমাদের প্রথমে কয়েকটা টপিক নিয়ে কিছুটা জানতে হবে, আগে সেটা নিয়ে আলোচনা করবো। এই লেখাটা পড়ার পর তুমি P-NP, টুরিং মেশিন নিয়ে একটা ভালো ধারণা পাবে।

**টুরিং মেশিন:** ১৯৩৬ সালে এলান টুরিং একটা “হাইপোথেটিকাল” মেশিন ডিজাইন করেন, বাস্তবে তৈরি করেননি, শুধুই খিওরি দিয়েছেন। মেশিনটা অ্যালগোরিদম সিমুলেট করতে পারে। এই মেশিন দিয়ে ব্যাখ্যা করা যায় কিভাবে কম্পিউটার একটা প্রবলেম সলভ করতে পারে। টুরিং চিন্তা করতেন “একটা প্রবলেম কম্পিউটেবল” বলতে কি বোঝায়? একটা প্রবলেম “কম্পিউটেবল” মানে হলো “কিছু ইনস্ট্রাকশন থাকবে যেগুলো একটা কম্পিউটার ফলো করলে একটা কাজ শেষ হবে”, এই ইনস্ট্রাকশনগুলোই হলো অ্যালগোরিদম। ইনস্ট্রাকশনগুলো কম্পিউটার ফলো করতে পারবে নাকি সেটা নির্ভর করে মেশিনের সামর্থ্যের উপর। যেসব অ্যালগোরিদম টুরিং মেশিন দিয়ে সলভ করা যায় সেগুলো “টুরিং-কম্পুটেবল”।

টুরিং মেশিনে একটা অসীম লম্বা টেপ থাকে(নিচের ছবি)। একটা হেড টেপের কোনো জায়গায় পয়েন্ট করে প্রবলেমের একটা “স্টেট” নির্দেশ করে। প্রতিটা স্টেট থেকে কোন স্টেট যাওয়া যায় সেটা নির্ভর করে ওই প্রবলেমের “ট্রানজিশন ডায়াগ্রাম” এর উপর। সহজ কথায় ডায়াগ্রামে বলা থাকে কোন ইনপুটের জন্য কোন স্টেট থেকে কোন স্টেটে যেতে হবে সেটা।



আধুনিক কম্পিউটারগুলো টুরিং মেশিন কম্পিউটেবল। টুরিং মেশিন দিয়ে সলভ করা না গেলে কম্পিউটার দিয়ে সলভ করা যাবনা, তাই এটা নিয়ে আমাদের এত আগ্রহ।

**নন-ডিটারমিনিস্টিক টুরিং মেশিন:** ধরো তোমার ইনস্ট্রাকশন সেটে বলা আছে “তুমি যদি একটা ১০ নম্বর স্টেটে থাকো এবং ইনপুটে একটা “A” পাও তাহলে বামে যাও” এবং একই সাথে বলা থাকে “তুমি যদি একটা ১০ নম্বর স্টেটে থাকো এবং ইনপুটে একটা “A” পাও তাহলে ডানে যাও” তাহলে তুমি কোনটা করবে? একই স্টেটে একই ইনপুটের জন্য ভিন্ন ইনস্ট্রাকশন! নন-ডিটারমিনিস্টিক টুরিং মেশিন এই ধরণের সিচুয়েশনে অনুমানের উপর নির্ভর করবে। সে রেন্ডমলি অনুমান করে কোনো একটা পাথে চলে যাবে। হয়তো সেই অনুমান ভুল এবং কম্পিউটেশনটা ভুল আউটকাম দেবে। কিন্তু এখানে বিবেচ হলো এমন কোনো সঠিক সিদ্ধান্তের সিকুয়েন্স আছে কি না। নন-ডিটারমিনিস্টিক মেশিন সবসময় ঠিক রেজাল্ট দিবে এমন কোনো কথা নেই, যদি রেজাল্ট পজিটিভ হয় তাহলে অবশ্যই সলিউশন আছে, কিন্তু নেগেটিভ হলেও নিশ্চিত করে বলা যাবনা যে আসলে পজিটিভ রেজাল্ট আছে নাকি নেই। এজন্যই এধরণের মেশিনের নাম নন-ডিটারমিনিস্টিক।

**ডিটারমিনিস্টিক টুরিং মেশিন:** এই মেশিনের ইনস্ট্রাকশন সেটে এভাবে একই সাথে দুই রকম ইনস্ট্রাশন থাকবেন। প্রতি সিচুয়েশনে নির্দিষ্ট কোনদিকে যেতে হবে সেটা আগে থেকে জানা থাকবে।

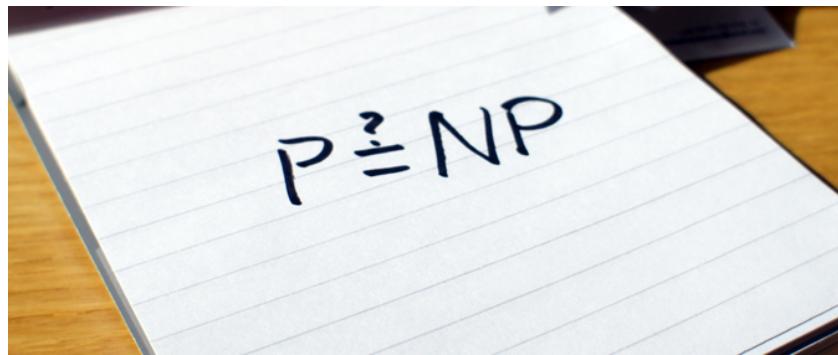
**আধুনিক কম্পিউটার ডিটারমিনিস্টিক টুরিং মেশিন** এর সমতুল্য। নন-ডিটারমিনিস্টিক ইনস্ট্রাকশন সেট নিয়ে কম্পিউটার কাজ করেন।

**পলিনমিয়াল কমপ্লেক্সিটি:** যেসব প্রবলেমের কমপ্লেক্সিটি ইনপুট সাইজের পাওয়ার সেগুলো পলিনমিয়াল কমপ্লেক্সিটি, যেমন  $O(n)$ ,  $O(n\log n)$ ,  $O(n^3)$  ইত্যাদি।

**P(Polynomial) প্রবলেম:** P বা পলিনমিয়াল ক্যাটাগরি প্রবলেমকে ডিটারমিনিস্টিক টুরিং মেশিন দিয়ে পলিনমিয়াল টাইমে সলভ করা যায়। যেমন শর্টেস্ট পাথ প্রবলেম, ন্যাপস্যাক প্রবলেম। শুধু যে সেগুলো সলভ করা যায় সেটা না, একটা “সার্টিফিকেট” দিলে সেটা পলিনমিয়াল টাইমে ভেরিফাই ও করা যায়। সার্টিফিকেট কি? ধরা যাক ইউলার পাথ প্রবলেমের কথা, একটা গ্রাফের সবগুলো এজ ভিজিট করতে হবে এবং কোন এজ দুইবার ভিজিট করা যাবেনা। এখানে সার্টিফিকেট হলো কিছু এজ এর সিকোয়েন্স ( $e_1, e_2, \dots, e_K$ )। আমাদেরকে এই সিকোয়েন্স দিলে সহজেই বলতে পারবো এটা একটা ভ্যালিড পাথ নাকি। প্রতিটা নোড ধরে আগাবো, দেখবো এক নোড দুইবার ভিজিট হয়েছে নাকি এবং সবগুলো নোড ভিজিট হয়েছে নাকি।  $O(E)$  তেই আমরা এটা ভেরিফাই করতে পারবো যেখানে E হলো এজ সংখ্যা। P প্রবলেম পলিনমিয়াল টাইমে সলভ করা যায়, এবং একটা সলিউশন(সার্টিফিকেট) ভ্যালিড নাকি সেটা পলিনমিয়াল টাইমে ভেরিফাই করা যায়।

**NP(Non-deterministic Polynomial) প্রবলেম:** NP প্রবলেমকে ডিটারমিনিস্টিক টুরিং মেশিন দিয়ে পলিনমিয়াল টাইমে ভেরিফাই করা যায়। কিন্তু পলিনমিয়াল টাইমে সলভ করা যায় কি যায় না সেটা প্রমাণ করা সম্ভব হয়নি। যেমন হ্যামিল্টন পাথ প্রবলেম, একটা গ্রাফের সব নোড ভিজিট করতে হবে, একটা নোড দুইবার ভিজিট করা যাবেনা। তোমাকে যদি কিছু নোডের সিকুয়েন্স দেয় তুমি সহজেই ইউলার পাথ এর মতো করে ভেরিফাই করতে পারবে। কিন্তু একটা গ্রাফে হ্যামিল্টন পাথ আছে নাকি নাই কিভাবে বলবে? এটা বের করার জন্য পলিনমিয়াল অ্যালগোরিদম এখন পর্যন্ত আবিষ্কার হয়নি। পলিনমিয়াল অ্যালগোরিদম না থাকলে আমরা সলভ করি এক্সপোনেন্টিসিয়াল অ্যালগোরিদম দিয়ে যেগুলোর কমপ্লেক্সিটি  $O(2^n), (k^n)$  এরকম। এই ধরণের কমপ্লেক্সিটি সাজেস্ট করে তোমাকে ব্রুট ফোর্সের সাহায্যে সবকরম পসিবিলিটি চেক করতে হবে, সাধারণত ব্যাকট্র্যাকিং দিয়ে এসব প্রবলেম সলভ করা হয়।

**P=NP?? :** প্রতিটা P প্রবলেমেই NP প্রবলেম ক্যাটাগরিতে পরে। কারণ সব P প্রবলেমকে পলিনমিয়াল টাইমে ভেরিফাই করা যায় যেটা NP ক্যাটাগরির একমাত্র শর্ত। তারমানে P হলো NP এর একটা সাবসেট। কিন্তু সব NP প্রবলেমই কি P প্রবলেম? যেসব প্রবলেমের সার্টিফিকেট পলিনমিয়াল টাইমে ভেরিফাই করা যায় সেগুলোর সবগুলোকেই কি পলিনমিয়াল টাইমে সলভ করা যায়? হ্যামিল্টন পাথ, স্যাট প্রবলেম ইত্যাদির ক্ষেত্রে আমরা জানিনা প্রবলেমগুলো P তে ফেলা যায় নাকি। আমরা জানিনা  $P=NP$  নাকি  $P \neq NP$ । এটা কম্পিউটার সায়েন্সের সবথেকে বিখ্যাত একটি ওপেন প্রবলেম। ক্লে ম্যাথমেটিকাল ইন্সিটিউট এটাকে ৭টি মিলেনিয়াম প্রাইজ প্রবলেম এর একটি ধরে ১০ লক্ষ ডলারের প্রাইজমানি ঘোষণা করেছে। তবে কম্পিউটার বিজ্ঞান এবং গণিতবিদরা ধারণা করে  $P \neq NP$ , কিন্তু শুধুমাত্র ধারণা কখনোই বিজ্ঞান নয়, আইনস্টাইন মুখে কি ধারণার কথা বললেন সেটা বিজ্ঞান নয়, তিনি পিয়ার রিভিউড জার্নালে যেটা প্রকাশ করেছেন সেটা বিজ্ঞান!



**NP hard:** এই ক্যাটাগরির প্রবলেমগুলো “অন্তত NP প্রবলেমের মতো” কঠিন। এই কথার মানে কি আর কঠিনকে কিভাবে পরিমাপ করে? অনেক সময়ই একটা প্রবলেমকে অন্য প্রবলেমে কনভার্ট করা যায়। ধরা যাক A একটা প্রবলেম যেটা NP ক্যাটাগরির এবং B প্রবলেমটা আমরা সলভ করতে চাই। A প্রবলেমকে যদি আমরা পলিনমিয়াল টাইমে B প্রবলেমে কনভার্ট পারি তাহলে নিশ্চিতভাবে বলা যায় B প্রবলেমটাও A প্রবলেমের মতোই কঠিন। **NP hard** প্রবলেমকে অনেক সময় NP ক্যাটাগরিতে ফেলা যায়না কারণ সেটা পলিনমিয়াল টাইমে ভেরিফাই করা যায় না।

NP কে পলিনমিয়াল টাইমে সলভ করা যায়না, ভেরিফাই করা যায়। **NP hard** কে পলিনমিয়াল টাইমে সলভ করা যায়না, ভেরিফাই করা যেতেও পারে, নাও যেতে পারে।

তাহলে আমরা এখন পর্যন্ত যতটুক জেনেছি:

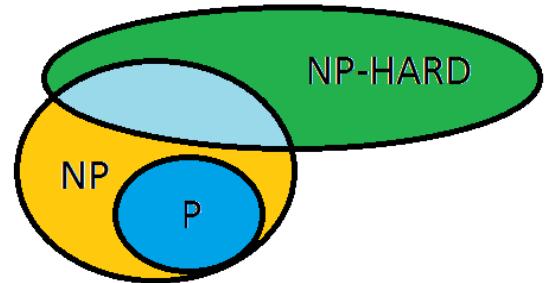
যদি কখনো প্রমাণ হয় যে  $P=NP$  তাহলে ছোট নীল সার্কেলটা আর দরকার হবে, হলুদ সার্কেলে  $P=NP$  লিখে দেয়া যাবে। NP-hard ক্যাটাগরির যেসব প্রবলেম পলিনমিয়াল টাইমে ভেরিফাই করা যায় সেগুলো NP ক্যাটাগরিতে, বাকিগুলো বাইরে। নীল অংশের প্রবলেমগুলো দুটো ক্যাটাগরিতে কমন, সেগুলোর নাম কি?

**NP complete:** একটা প্রবলেম NP-complete হবে কেবল যদি প্রবলেমটা NP-hard হয় এবং সেটা NP ক্যাটাগরীতেও পড়ে। **NP complete** প্রবলেমকে পলিনমিয়াল টাইমে অন্য যেকোনো NP প্রবলেমে কনভার্ট করা যায় এবং এই প্রবলেমের সার্টিফিকেটকে পলিনমিয়াল টাইমে ভেরিফাইও করা যায়।

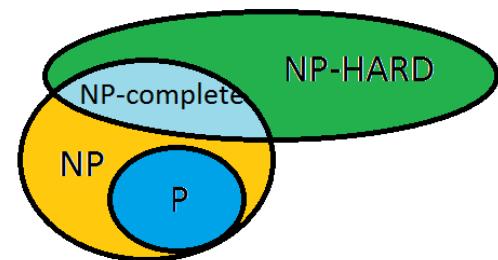
একটা প্রবলেম NP-hard প্রমাণ করতে হলে আমাদের সেটাকে পরিচিত কোনো NP-complete প্রবলেমকে সেই প্রবলেমে কনভার্ট করতে হবে। যেমন লংগেস্ট পাথ প্রবলেমে বলা হয়েছে গ্রাফে সবথেকে লম্বা পাথ বের করতে হবে যেন কোনো নোড সেই পাথে দুইবার না থাকে। হ্যামিল্টনিয়ান পাথ একটা পরিচিত NP-complete প্রবলেম, গ্রাফে যদি হ্যামিল্টনিয়ান পাথ থাকে তাহলে সবথেকে লম্বা পাথে  $n - 1$  টা এজ থাকবে কারণ  $n$  টা নোড একবার করে ভিজিট করতে হবে। হ্যামিল্টন প্রবলেমকে আমরা একটু ভিন্ন ভাবে বলতে পারি “একটা গ্রাফে কি এমন কোনো পাথ থাকা সম্ভব যেটায়  $k$  টা এজ আছে?”,  $k=n-1$  এর জন্য প্রবলেমটা সলভ করতে পারলে হ্যামিল্টন পাথের প্রবলেম সলভ হয়ে যাবে। এবং এই প্রবলেমটা সলভ করতে পারলে  $k$  এর বিভিন্ন মানের জন্য সলভ করে সবথেকে লম্বা পাথও বের করতে পারবো! তাহলে দেখা যাচ্ছে হ্যামিল্টন পাথ প্রবলেমকে একটু ভিন্ন ভাবে বর্ণনা করে আমরা লংগেস্ট পাথ প্রবলেমে কনভার্ট করলাম। যেহেতু হ্যামিল্টন পাথের পলিনমিয়াল সলিউশন আমরা জানিনা, লংগেস্ট পাথেরও জানিনা।

এবার হয়তো তুমি বুঝতে পারচো একটা মাত্র প্রবলেম পলিনমিয়াল টাইমে সলভ করতে পারলে কেন সবগুলো করা যাবে।

ডিসিশান প্রবলেম হলো সেগুলো যেগুলোকে yes/no দিয়ে উত্তর দেয়া যায়। যেমন A থেকে B তে কি 10 সেকেন্ডে পৌছানো সম্ভব? অপটিমাইজেশন প্রবলেম অ্যান্সারকে মিনিমাইজ বা ম্যাক্সিমাইজ করে। যেমন A থেকে B তে সব থেকে কম কত



$P \neq NP$  ধরে নিয়ে ভ্যান ডায়াগ্রাম

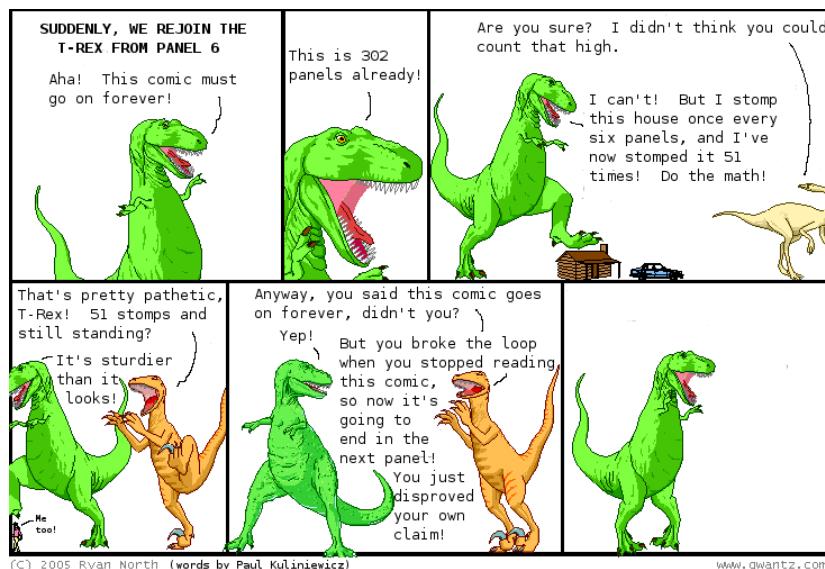


$P \neq NP$  ধরে নিয়ে ভ্যান ডায়াগ্রাম

সেকেন্ডে পৌছানো সম্ভব। উপরের ক্যাটাগরিগুলো প্রবলেমের ডিসিশান ভার্সন নিয়ে কাজ করে। যেকোনো অপটিমাইজেশন প্রবলেমকেই একটু ভিন্নভাবে বর্ণনা করে ডিসিশান প্রবলেম বানিয়ে ফেলা যায় যেটা আমরা উপরে হ্যামিল্টন পাথের ক্ষেত্রে করেছি।

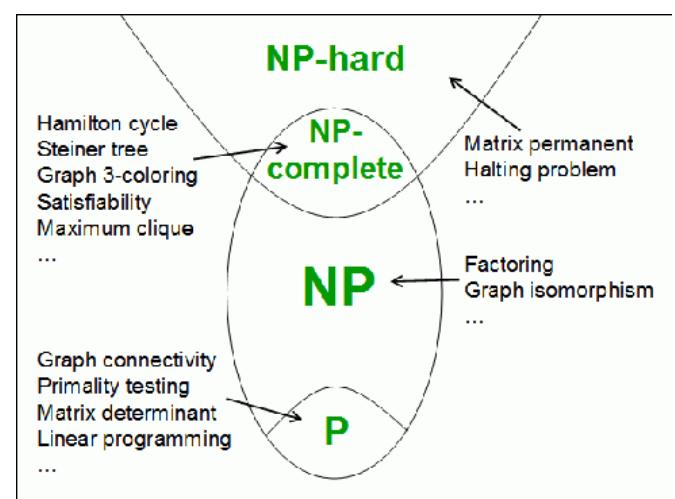
এখন প্রশ্ন হলো প্রথম np-complete প্রবলেমটা কি? প্রথম একটা np-complete প্রবলেম না থাকলে আমরা অন্য প্রবলেমকে np-complete বা hard ক্যাটাগরিতে ফেলতে পারছিনা যেহেতু আমরা এটা করছি প্রবলেমকে কনভার্ট করে। ধরা যাক তোমার কাছে কিছু বুলিয়ান ভ্যারিয়েবল আছে  $x, y, z$  ইত্যাদি। তুমি কিছু শর্ত দিলে এরকম: “ $X$  সত্য হলে  $Y$  মিথ্যা”, “ $Z$  এবং  $X$  দুইটাই মিথ্যা” ইত্যাদি। এখন তোমাকে  $x, y, z$  এ true/false ভ্যালু অ্যাসাইন করতে হবে যেন সব রিলেশন সত্য হয়। এটাকে বলা হয় k-sat। 2-স্যাটকে পলিনমিয়াল টাইমে সলভ করা যায়,  $k > 2$  হলে পলিনমিয়াল সলিউশন পাওয়া যায়না। NP ক্যাটাগরিতে রিলেশন সত্য হলে একটি NP complete প্রবলেম। এটাই ছিল প্রথম প্রমাণ। Cook প্রবর্তিতে কম্পিউটার সায়েন্সের সর্বোচ্চ টুরিং পুরষ্কার পান তার কাজের জন্য। k-sat প্রবলেম যদি তুমি পলিনমিয়াল টাইমে সলভ করতে পারো তার সময়ে তুমি আইনস্টাইন-নিউটন টাইপের মানুষের সমতুল্য হয়ে যাবে।

টুরিং এর একটা বিখ্যাত NP-hard প্রবলেম হলো **হাল্টিং প্রবলেম**। “একটা কম্পিউটার প্রোগ্রাম দেয়া হলো, বলতে হবে এটা কখনো শেষ হবে নাকি অসীম সময় ধরে চলতে থাকবে”। এটাকে পলিনমিয়াল টাইমে সলভ বা ভেরিফাই কিছুই করা যায় না। বিস্তারিত জানতে আমার [এই লেখাটা পড় কিন্তু](#) তার আগে একটা কমিকস পড়ি:



নিচের ছবিতে প্রতিটা ক্যাটাগরির কিছু প্রবলেম দেখানো হয়েছে:

প্রোগ্রামিং কনটেস্টে প্রায়ই NP প্রবলেম দেখা যায় তবে সেখানে ইনপুটের সাইজ অনেক কমিয়ে দেয়া হয় বা বিশেষ কোনো কন্ডিশন যোগ করে দেয়া হয় যাতে ব্যাকট্র্যাকিং করে 3-10 সেকেন্ডের মাঝে সলভ করা সম্ভব হয়। অনেক রিয়েল লাইফ সমস্যার পলিনমিয়াল সলিউশন না থাকায় বিজ্ঞানীরা চেষ্টা করছেন এক্সপোনেন্টিসিয়াল সলিউশনকেই যতটা সম্ভব উন্নত করার। ব্রাঞ্চ এন্ড বার্ড টেকনিকের সাহায্যে ব্যাকট্র্যাকিং এ সার্চ স্পেস কমিয়ে মোটামুটি দ্রুত সলিউশন বের করা যায়। অনেক সময় হিউরিস্টিক এর সাহায্য নেয়া হয়। ট্রাভেলিং সেলসম্যান, গ্রাফ কালারিং ইত্যাদি সলভ করার জন্য অনেক আধুনিক অ্যালগোরিদম আছে, এগুলো নিয়ে এখনো রিসার্চ চলছে, তোমার আগ্রহ থাকলে কিছু রিসার্চ পেপার নামিয়ে নিজেই পড়ালেখা করতে পারো।



রেফারেন্স:

<http://www.quora.com/What-are-P-NP-NP-complete-and-NP-hard>

<http://stackoverflow.com/questions/1857244/np-vs-np-complete-vs-np-hard-what-does-it-all-mean>

[http://en.wikipedia.org/wiki/P\\_versus\\_NP\\_problem](http://en.wikipedia.org/wiki/P_versus_NP_problem)

Introduction to Algorithms by Cormen

[http://en.wikipedia.org/wiki/Hamiltonian\\_path\\_problem](http://en.wikipedia.org/wiki/Hamiltonian_path_problem)

[http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Non-deterministic\\_Turing\\_machine.html](http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Non-deterministic_Turing_machine.html)

<http://plato.stanford.edu/entries/turing-machine/#Computability>

তানভীরুল ইসলামকে ধন্যবাদ নন-ডিটারমিনিস্টিক এর ব্যাপারটা আমাকে বুঝিয়ে বলার জন্য।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# হাল্টিং প্রবলেম

 shafaetsplanet.com/planetcoding/

শাফায়েত

সেপ্টেম্বর ২৭, ২০১৬

গণিত বা কম্পিউটার বিজ্ঞানের সব সমস্যাই কি সমাধানযোগ্য? আমরা জানি NP ক্যাটাগরির সমস্যাগুলোকে পলিনোমিয়াল সময়ে সমাধান করার সম্ভব নাকি সেটা জানা এখন পর্যন্ত সম্ভব হয় নি, কিন্তু ইনপুটের আকার ঘর্থেষ্ট ছোট হলে অথবা তোমার হাতে অসীম সময় এবং মেমরি থাকলে NP সমস্যাও এক্সপোনেনশিয়াল সময়ে সমাধান করা সম্ভব। কিন্তু এমন কিছু সমস্যা আছে যেটা তোমার হাতে যত বড় সুপার কম্পিউটারই থাকুক সমাধান করা সম্ভব না। এখানে আমি ধরে নিচ্ছি আমাদের কম্পিউটারগুলো [টুরিং মেশিন কম্পিউটেবল](#)। (টুরিং মেশিন কি মনে না থাকলে আগে আমার [এই লেখাটা পড়ো](#))

হাল্টিং প্রবলেম (Halting Problem) এমনই একটা সমস্যা যার কোনো সমাধান নেই। তোমাকে একটা কম্পিউটার প্রোগ্রাম এবং একটা ইনপুট দেয়া হলো। তোমাকে বলতে হবে সেই ইনপুটের জন্য প্রোগ্রামটা কি সীমিত সময়ে থামবে নাকি অসীম বা ইনফিনিটি লুপে আটকে যাবে?

Python

```
1 def my_program(x):
2     count = 0
3     while True:
4         if count == x:
5             break
6         count = count + 1
```

উপরের কোডে তুমি যদি  $x=-10$  ইনপুট দাও তাহলে প্রোগ্রামটা কখনোই থামবে না (কোডটা পাইথনে লেখা, ইন্টিজার ওভারফ্লো এর ভয় নেই)

তোমার কাজ হলো এমন একটা প্রোগ্রাম লেখা যেটা যেকোনো আরেকটা প্রোগ্রাম এবং ইনপুট দেখে বলে দিতে পারবে প্রোগ্রামটা ইনফিনিটি লুপে আটকে যাবে নাকি। এটাই হলো হাল্টিং প্রবলেম (Halting Problem)।

এখন আমরা “প্রক্রফ বাই কনট্রাডিকশন” মেথডের সাহায্যে প্রমাণ করবো যে হাল্টিং প্রবলেম সমস্যার সমাধান করা সম্ভব না। আমরা প্রথমে ধরে নিবো যে সমস্যাটির একটা সমাধান আছে এবং তারপর প্রমাণ করবো যে সমাধানটি অস্তিত্ব থাকা অসম্ভব।

ধরো আমাদের কাছে একটা প্রোগ্রাম আছে যেটা হাল্টিং প্রবলেম সমাধান করতে পারে। প্রোগ্রামটা হতে পারে এরকম:

```
1 will_it_halt(P, I)
2     if P(I) halts in finite time
3         return TRUE
4     else
5         return FALSE
6
```

এটা একটা কান্সনিক কোড। কোডটার প্যারামিটার হলো একটা প্রোগ্রাম  $P$  এবং  $P$  কে প্যারামিটার হিসাবে পাঠানো হবে এমন একটা ইনপুট। কোডটা কোন উপায়ে বলে দিতে পারে  $P$  প্রোগ্রামে। ইনপুট পাঠানো হলে সেটা সীমিত সময়ে থামবে নাকি থামবে না।

এ ধরণের কোডকে বলা হয় [Oracle Machine](#)। Oracle এর আক্ষরিক অর্থ হলো একজন ওঝা যে আধ্যাত্মিক ক্ষমতা দিয়ে সমস্যা সমাধান করতে পারে। আমাদের `will_it_halt` কোডও জাদুকরি ক্ষমতা দিয়ে হাল্টিং প্রবলেম সমাধান করতে পারে। এখন আমরা যদি প্রমাণ করতে পারি এমন কোনো Oracle Machine থাকা সম্ভব না যেটা দিয়ে হাল্টিং প্রবলেম সমাধান করা যায় তাহলেই আমাদের কাজ শেষ।

এখন আমরা আরেকটা প্রোগ্রাম লিখবো, মনে করো প্রোগ্রামটার নাম প্যারাডক্স (Paradox)।

```

1  paradox(program)
2      will_it_halt(program, program)
3          Run Forever
4      else
5          return TRUE
6

```

প্যারাডক্স ইনপুট হিসাবে একটা প্রোগ্রামকে গ্রহণ করে। একটা প্রোগ্রাম হলো কিছু স্ট্রিং দিয়ে লেখা কিছু ইনস্ট্রাকশন। তাই আমরা চাইলে প্যারামিটার হিসাবে প্যারাডক্স প্রোগ্রামটাকেই পাঠাতে পারি!

### 1 paradox(paradox)

এখন দ্বিতীয় লাইনে আমরা ওরাকল মেশিনকে জিজ্ঞেস করছি প্যারাডক্স প্রোগ্রামে ইনপুট হিসাবে প্যারাডক্স প্রোগ্রামটাকেই পাঠালে সেটা ইনফিনিটি লুপে আটকে যাবে নাকি যাবে না। যদি ওরাকল বলে যে ইনফিনিটি লুপে আটকে যাবে না, তাহলে আমরা তৃতীয় লাইনে সেটাকে ইনফিনিটি লুপে আটকে দিবো! আর ওরাকল যদি বলে যে ইনফিনিটি লুপে আটকে যাবে তাহলে আমরা প্রোগ্রাম থেকে বের হয়ে যাবো। তারমানে ওরাকল যা বলছে সেটা সত্যি না, বরং তার উল্টাটা ঘটছে। এটাই হলো কন্ট্রাডিকশন, এ থেকে আমরা বলতে পারি will\_it\_halt নামক ওরাকল মেশিনটার অস্তিত্ব থাকা সম্ভব না!

এটাই হলো হাল্টিং প্রবলেম যা সমাধানযোগ্য না সেটার প্রমাণ।

মোবাইলে কিছু কিছু অ্যাপ চালু করলে কোনো একটা বাগের জন্য ক্লীন ফ্রিজ হয়ে যায়, তখন ফোনের ব্যাটারি খুলে রিস্টার্ট করা ছাড়া উপায় থাকে না। এমন কোনো অ্যাপ কি তুমি লিখতে পারবে যেটা বলে দিতে পারবে কোন অ্যাপ চালালে ক্লীন ফ্রিজ হয়ে যেতে পারে? এটা হাল্টিং প্রবলেমেরই একটা ভ্যারিয়েশন যা [এই লেখাটায়](#) সুন্দর করে ব্যাখ্যা করা আছে, আগ্রহ থাকলে পড়তে পারো।

হাল্টিং প্রবলেম যদি সমাধানযোগ হতো তাহলে কি হত? তাহলে কোড ডিবাগিং করা অনেক সহজ হয়ে যেতে, কোড চালু না করেই আমরা বলতে পারতাম কোডটা ইনফিনিটি লুপে পড়বে নাকি। এর থেকেও গুরুত্বপূর্ণ ব্যাপার হলো, অনেক গাণিতিক সমস্যা সমাধান করা যেত এই will\_it\_halt প্রোগ্রামটা দিয়ে। একটা উদাহরণ দেই। গোল্ডবার্থ নামক একজন গণিতবিদ বলু বছর আগে বলে গেছেন “২ এর থেকে বড় যে কোন জোড় সংখ্যাকে দুটি প্রাইম সংখ্যার যোগফল হিসাবে লেখা সম্ভব”। কেও এখনো প্রমাণ করতে পারে নি যে কনজেকচারটা সত্য নাকি মিথ্যা। এখন আমি একটা প্রোগ্রাম লিখলাম প্রমাণ করার জন্য:

Python

```

1  def goldbach():
2      k = 4
3      while True:
4          ok = False
5          for p1 in range(k):
6              for p2 in range(k):
7                  if(prime(p1) && prime(p2) && p1+p2==k):
8                      ok = True
9          if not ok:
10             exit()
11         k += 2

```

এই প্রোগ্রামটা  $k=4$  থেকে শুরু করে প্রতিটি জোড় সংখ্যাকে ব্রুট ফোর্সের মাধ্যমে দুটি প্রাইম সংখ্যার যোগফল হিসাবে লেখার চেষ্টা করবো। যদি কোনো  $k$  এর জন্য  $ok = \text{false}$  হয় তাহলে প্রমাণ হবে যে কনজেকচারটা মিথ্যা, তখন প্রোগ্রামটা বন্ধ হয়ে যাবে। আর যদি কনজেকচারটা সত্য হয় তাহলে অসীম সময় ধরে প্রোগ্রামটা চলতে থাকবে। will\_it\_halt প্রোগ্রামটার অস্তিত্ব থাকলে এখন will\_it\_halt(goldback, null) এর আউটপুট দেখেই বলে দিতে পারতাম কনজেকচারটি সত্য নাকি মিথ্যা!

তুমি যদি এমন একটা গাণিতিক মডেল বের করতে পারো যা হাল্টিং প্রবলেমকে সমাধান করতে পারে তাহলে আমরা আমরা একটা [হাইপার কম্পিউটেশন](#) বা **super-Turing computation** মডেল পাবো, তখন আমরা এমন সব সমস্যা সমাধান করতে পারবো যা টুরিং মেশিন দিয়ে সমাধান করা সম্ভব না।

হাল্টিং প্রবলেম অনেক দার্শনিক প্রশ্নেরও জন্ম দিয়েছে। যেমন আমরা কি কখনো নিজের মস্তিষ্ক সম্পর্কে ১০০% জানতে পারবো? নাকি কোনো থিওরিটিকাল সীমাবদ্ধতার কারণে মস্তিষ্কের অনেক রহস্য আমরা কোনো জানতে পারবো না?

আজ এখানেই শেষ, হ্যাপি কোডিং!

রেফারেন্স

<http://www.cgl.uwaterloo.ca/csk/halt/>

<https://www.quora.com>If-the-halting-problem-was-solvable-what-would-be-the-implications>

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# HANDBOOK OF ALGORITHMS

## Section Some Basic Algorithms

Courtesy of  
*Shafaet Ashraf*

**বাইনারি সিরিজ - ১ \_ শাফায়তের ব্লগ.pdf**

**বাইনারি সিরিজ - ২ \_ শাফায়তের ব্লগ.pdf**

**ফ্লয়ডের সাইকলে ফাইন্ডিং অ্যালগরিদিম \_ শাফায়তের ব্লগ.pdf**

**ডেরিকেশন অ্যারে \_ শাফায়তের ব্লগ.pdf**

**মটি ইন দ্যা মডিল \_ শাফায়তের ব্লগ.pdf**

**ব্যাকট্ৰ্যাক্ৰি\_ পারমুটশেন জনোৱাটের \_ শাফায়তের ব্লগ.pdf**

# বাইনারি সার্চ - ১

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

জুন ৭, ২০১৫

তুমি নিশ্চয়ই লক্ষ্য করেছো ডিকশনারিতে লাখ লাখ শব্দ থাকলেও প্রয়োজনীয় শব্দটা খুজে পেতে কখনো খুব বেশি সময় লাগে না। এটার কারণ হলো শব্দগুলো অক্ষর অনুযায়ী সাজানো থাকে। তাই তুমি যদি **dynamite** শব্দটা ডিকশনারিতে খোজার চেষ্টা করো এবং এলোমেলোভাবে কোনো একটা পাতা খুলে **kite** শব্দটা খুজে পাও তাহলে তুমি নিশ্চিত হয়ে যেতে পারো যে তুমি যে শব্দটা খোজার চেষ্টা করছো সেটা বাম দিকে কোথাও আছে। আবার যদি তুমি **dear** শব্দটা খুজে পাও তখন তুমি আর বাম দিকের পাতাগুলোয় খোজার চেষ্টা করবে না। এভাবে অল্প সময়ের মধ্যে ডিকশনারিতে যেকোনো শব্দ খুজে পাওয়া যায়।

বাইনারি সার্চ হলো অনেকটা এরকম একটা পদ্ধতি যেটা ব্যবহার করে একটা অ্যারে থেকে কোনো একটা তথ্য খুজে বের করা যায়।

ধরো তোমার কাছে একটা অ্যারেতে অনেকগুলো সংখ্যা আছে এরকম:

| 100, 2, 10, 50, 20, 500, 100, 150, 200, 1000, 100100, 2, 10, 50, 20, 500, 100, 150, 200, 1000, 100

সংখ্যাগুলো এলোমেলোভাবে সাজানো থাকায় এখান থেকে একটা নির্দিষ্ট সংখ্যা খুজে পাওয়া সহজ না। তুমি যদি অ্যারে থেকে ৫০০ সংখ্যাটা খুজতে চাও তাহলে সবগুলো ইনডেক্স পরীক্ষা করে দেখতে হবে, এটাকে বলা হয় লিনিয়ার সার্চ। অ্যারের আকার যদি  $n$  হয় তাহলে লিনিয়ার সার্চের টাইম **কমপ্লেক্সিটি** হলো  $O(n)O(n)$ ।

কিন্তু যদি সংখ্যাগুলো নিচের মত সাজানো থাকে তাহলে খুজে পাওয়া অনেক সহজ হয়ে যায়:

| 2, 10, 20, 50, 100, 100, 100, 150, 200, 500, 10002, 10, 20, 50, 100, 100, 150, 200, 500, 1000

এখন যদি ১৫০ সংখ্যাটা খুজে বের করতে হয় তাহলে আমরা প্রথমে ঠিক মাঝের ইনডেক্সটা পরীক্ষা করবো। প্রথম ইনডেক্স ০ এবং শেষের ইনডেক্স ১০ হলে মাঝের ইনডেক্সটা হলো  $(0+10)/2$  বা ৫ তম ইনডেক্স।

| \$2, 10, 20, 50, 100, 100, 100, 150, 200, 500, 1000

মাঝের ইনডেক্সের সংখ্যাটা ১০০ যা ১৫০ এর থেকে ছোট, আমরা জেনে গেলাম যে সংখ্যাটা ডান পাশে কোথাও আছে, বামের অংশ আমাদের আর দরকার নাই।

| 100, 150, 200, 500, 1000

এখন বাকি অংশটা নিয়ে আবারো একই কাজ করবো। এবার মাঝের ইনডেক্সের সংখ্যাটা হলো ২০০ যেটা ১৫০ এর থেকে বড়। তারমানে ডানের অংশটা আমরা ফেলে দিতে পারি।

| 100, 150

এবার মাঝের সংখ্যাটা হলো ১০০ যা ১৫০ এর থেকে ছোট। আবারো বামের অংশ ফেলে দিবো, থাকবে শুধু:

এবার মাঝের ইনডেক্সের সংখ্যাটা হলো ১৫০। তারমানে কাঞ্জিত সংখ্যাটাকে খুজে পাওয়া গেছে।

বাইনারি সার্চে প্রতিবার অ্যারের ঠিক অর্ধেক অংশ আমরা বাতিল করে দিচ্ছি এবং বাকি অর্ধেক অংশে খুজছি। একটা সংখ্যা  $n$  কে সর্বোচ্চ কয়বার ২ দিয়ে ভাগ করা যায় যতক্ষণ না সংখ্যাটা ১ হয়ে যাচ্ছে? উন্নত হলো  $\log_2 n \log_2 n$ । তাই বাইনারি সার্চে সর্বোচ্চ  $\log_2(n) \log_2(n)$  সংখ্যক ধাপের পর আমরা দরকারি সংখ্যাটা খুজে পাবো, **কমপ্লেক্সিটি**  $O(\log_2 n) O(\log_2 n)$ ।

কিন্তু একটা সমস্যা হলো বাইনারি সার্চ করার আগে অবশ্যই সংখ্যাগুলোকে ছোট থেকে বড় বা বড় থেকে ছোট সাজিয়ে নিতে হবে, এই প্রক্রিয়াটাকে বলা হয় সার্টিং। তুমি যতই চেষ্টা কর না কেন একটা অ্যারে  $O(n \times \log_2 n) O(n \times \log_2 n)$  এর কম কমপ্লেক্সিটি সর্ট করতে পারবে না! \* তাহলে কেও বলতে পারে যে সর্ট করতে যে সময় লাগছে তার থেকে অনেক কম সময়ে লিনিয়ার সার্চ করেই আমরা সংখ্যাটা খুজে পেতে পারি, বাইনারি সার্চ কেন করবো? তোমার যদি একটা অ্যারেতে মাত্র ১ বার সার্চ করা দরকার হয় তাহলে কষ্ট করে সর্ট করে বাইনারি সার্চ করার থেকে লিনিয়ার সার্চ করা অনেক ভালো। কিন্তু যদি এমন হয় একটা অ্যারেতে অনেকবার সার্চ করা দরকার হবে? যেমন একটা ডিকশনারিতে বিভিন্ন সময় বিভিন্ন শব্দ খোজা দরকার হয়, সেক্ষেত্রে সর্ট করে রাখাই বুদ্ধিমানের কাজ।

নিচের পাইথন কোডে বাইনারি সার্চের ইম্প্লিমেন্টেশন দেখানো হয়েছে:

Python

```

1 def search(array, key):
2     begin=0
3     end=len(array)-1
4     index=None
5     while begin<=end:
6         mid=(begin+end)/2
7         if key == array[mid]:
8             index=mid #The value is found, save the index.
9             break
10        elif key > array[mid]: begin=mid+1 #Search the right portion
11        elif key < array[mid]: end=mid-1 #Search the left portion
12    return index #If the number is not found, index will contain None.
13
14
15 info=[100,2,10,50,20,500,100,150,200,1000,100]
16 info=sorted(info)
17 while True:
18     key = int(raw_input())
19     print search(info,key)

```

উপরের কোডে যদি আমরা ১০০ ইনপুট দেই তাহলে ৫ রিটার্ন করবে, কারণ অ্যারেটা সর্ট করার পর ৫ নম্বর ইনডেক্সে ১০০ আছে। কিন্তু সর্টেড অ্যারের দিকে তাকালে আমরা দেখছি যে ৪, ৫, ৬ এই সবগুলো ইনডেক্সেই ১০০ আছে। আমরা যদি চাই কোনো সংখ্যা একাধিকবার থাকলে সবথেকে বামের ইনডেক্সটা রিটার্ন করতে, তাহলে কোডটা কিভাবে পরিবর্তন করতে হবে? সেক্ষেত্রে বাইনারি সার্চ ফাংশনটা হবে এরকম:

Python

```

1 def search(array, key):
2     begin=0
3     end=len(array)-1
4     index=None
5     while begin<=end:
6         mid=(begin+end)/2
7         if key == array[mid]:
8             index=mid #One occurrence of the value is found, save the index
9             end=mid-1 #Continue searching the left portion after one occurrence is found
10            elif key > array[mid]: begin=mid+1
11            elif key < array[mid]: end=mid-1
12    return index #Index will contain None if the value is not found

```

শুধুমাত্র ১টা মাত্র লাইন পরিবর্তন করা হয়েছে, এবার কোনো সংখ্যা খুজে পাবার পর খোজা বন্ধ না করে বামের বাকি অংশটুকুতে খুজতে থাকবো।

**লোয়ার বাউন্ড:** তোমাকে একটা সর্টেড অ্যারে দেয়া আছে। তুমি নতুন একটা সংখ্যা XX সেই অ্যারেতে ঢুকাতে চাও। লোয়ার বাউন্ড হলো সবথেকে বামের ইনডেক্স যেখানে তুমি সংখ্যাটা ঢুকিয়ে বাকি সংখ্যাগুলোকে একঘর ডানে সরালে অ্যারেটা তখনো সর্টেড থাকবে। মনে করো অ্যারেটা এরকম:

| 10 20 20 30 30 40 50

তাহলে 20 এর লোয়ার বাউন্ড হলো ইনডেক্স ১ (০ বেসড ইনডেক্স), কারণ ১ নম্বর ইনডেক্সে তুমি ২০ সংখ্যাটাকে বসিয়ে বাকি সংখ্যাগুলোকে ডানে সরিয়ে দেয়ার পরেও অ্যারেটা সর্টেড থেকে যাচ্ছে। ঠিক সেরকম ২৫ এর জন্য লোয়ার বাউন্ড হলো ইনডেক্স ৩।

সহজ কথায় সবথেকে বামের যে ইনডেক্সে X এর সমান বা বড় কোনো সংখ্যা আছে সেই ইনডেক্সটাই হলো লোয়ার বাউন্ড। XX যদি অ্যারের প্রতিটা সংখ্যার থেকে বড় হয় তাহলে সর্বশেষ ইনডেক্সের পরবর্তী ইনডেক্সটা অর্থাৎ nn তম ইনডেক্সটা হলো লোয়ার বাউন্ড যেখানে nn অ্যারের আকার।

নিচের কোডে বাইনারি সার্চ করে লোয়ার ইনডেক্স খুজে বের করা হয়েছে, তারপর সেই ইনডেক্সে নতুন সংখ্যাটা বসিয়ে দেয়া হয়েছে।

Python

```

1 def searchLowerBound(array, key):
2     begin=0
3     end=len(array)-1
4     index=-1
5     while begin<=end:
6         mid=(begin+end)/2
7         if key == array[mid]:
8             index=mid
9             end=mid-1
10        elif key > array[mid]: begin=mid+1
11        elif key < array[mid]: end=mid-1
12    return begin
13
14
15 info=[100,2,10,50,20,500,100,150,200,1000,100]
16 info=sorted(info)
17 print info
18 while True:
19     X = int(raw_input())
20     lowerbound=searchLowerBound(info,X)
21     info.insert(lowerbound,X)
22     print "New array: ",info
23

```

(অ্যারেতে এভাবে সংখ্যা ইনসার্ট করতে  $O(n)O(n)$  সময় লাগে। অ্যারের জায়গায় লিংক লিস্ট ব্যবহার করে  $O(1)O(1)$  সময়ে ইনসার্ট করা যায় কিন্তু সেক্ষেত্রে বাইনারি সার্চ করা সম্ভব না কারণ লিংক লিস্টে যেকোনো ইনডেক্স রেন্ডম এক্সেস করা যায় না, পয়েন্টার ধরে আগাতে হয়। একটা উপায় হতে পারে বাইনারি সার্চ ট্রি আকারে তথ্যগুলো সাজিয়ে রাখা, সেটা আমরা এই লেখায় দেখবো না, তুমি চাইলে নিজে শিখে নিতে পারো।)

**আপার বাউন্ড:** আপার বাউন্ড হলো সব থেকে ডানের ইনডেক্স যেখানে তুমি নতুন সংখ্যাটা তুকালে অ্যারেটা সর্টেড থাকবে। তারমানে সবথেকে বামের যে ইনডেক্সে  $X$  এর বড় কোনো সংখ্যা আছে সেই ইনডেক্সটাই হলো আপার বাউন্ড। উপরের উদাহরণটায় ২০ এর আপার বাউন্ড হলো ইনডেক্স ৩। আপার বাউন্ড বের করার কোডটা তোমার বাড়ির কাজ হিসাবে থাকলো!

প্রথম পর্ব এখানেই শেষ। পরের পর্বে বাইসেকশন মেথড নিয়ে আলোচনা করবো।

প্র্যাকটিসের জন্য প্রবলেম:

[https://uva.onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show\\_problem&problem=1552](https://uva.onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=1552)

[http://www.lightoj.com/volume\\_showproblem.php?problem=1088](http://www.lightoj.com/volume_showproblem.php?problem=1088)

হ্যাপি কোডিং!

\* কাউন্টিং সর্ট বা রেডিভ সর্ট ব্যবহার করে  $O(n)O(n)$  এ সর্টিং করা যায় তবে সেগুলো কাজ করে বিশেষ ধরণের কিছু ইনপুটের ক্ষেত্রে। ইনপুট কি ধরণের হবে সেটার উপর কোনো শর্ত না থাকলে  $O(\log 2n)O(\log 2n)$  এর কমে সর্টিং করা সম্ভব না সেটা গাণিতিকভাবে প্রমাণ করা যায়।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# বাইনারি সার্চ – ২

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ১৭, ২০১৫

আগের লেখায় আমরা বাইনারি সার্চ কিভাবে কাজ করে দেখেছি। এখন একই পদ্ধতি ব্যবহার করে আমরা অন্যরকম কিছু সমস্যা সমাধান করবো। আমরা এখন যেটা শিখবো সেটা বাইসেকশন মেথড নামেই বেশি পরিচিত।

সহজ একটা সমস্যা সমাধান করতে করতে আমরা বাইসেকশন কিভাবে কাজ করে দেখবো। মনে করো তুমি যে ভাষা ব্যবহার করে প্রোগ্রামিং করছ সেখানে বর্গমূল বের করার জন্য কোনো ফাংশন নাই, তোমাকে নিজে ফাংশন লিখে নিতে হবে। আমরা  $\text{mysqrt}(X)$  নামের একটা ফাংশন লিখবো যেখানে  $X$  সংখ্যাটা পাঠালে সংখ্যাটার বর্গমূল রিটার্ন করবে,  $X$  সংখ্যাটা দশমিকযুক্ত হতে পারে, তবে শূণ্যের কম হবে না।

আমরা জটিল কোনো গাণিতিক হিসাবে ঘাবো না বর্গমূল বের করার জন্য, আমরা বাইনারি সার্চ করেই বর্গমূল বের করে ফেলবো!

একটু মনে করি আমরা একটা ছোট থেকে বড় সাজানো অ্যারেতে কোনো সংখ্যা খুজে বের করার সময় কি করছিলাম। মাঝখানের সংখ্যাটা যদি বেশি বড় হয় তাহলে ডানের অংশ বাদ দিয়ে দিচ্ছিলাম, বেশি ছোটো হলে বামের অংশটা বাদ দিয়ে দিচ্ছিলাম।

বর্গমূল বের করার সময় আমরা জানি যে  $X$  এর বর্গমূল অবশ্যই  $0$  থেকে  $X$  এর মধ্যের একটা সংখ্যা হবে। ধরো  $X=15$ । আমরা প্রথমে ঠিক মাঝের সংখ্যাটা নিবো। এক্ষেত্রে মাঝের সংখ্যা হলো  $7.5$  যাকে আবার বর্গ করলে পাওয়া যায়  $56.25$  যা  $15$  এর থেকে অনেক বড়। তারমানে  $7.5$  থেকে  $15$  পর্যন্ত বাকি সব সংখ্যার বগই  $15$  এর থেকে বড়, এই অংশটা আমরা বাদ দিয়ে দিতে পারি।

এখন তাহলে আমরা আবার  $0$  থেকে  $7.5$  এর মধ্যে খুজবো। মাঝের সংখ্যাটা হলো  $3.75$  যাকে বর্গ করলে পাই  $14.0625$  যা  $15$  থেকে ছোটো। তারমানে  $0$  থেকে  $3.75$  পর্যন্ত অংশে বর্গমূল পাবার কোনো সম্ভাবনা নেই। এখন আবার  $3.75$  থেকে  $7.5$  এর মাঝে খুজবো।

পুরো সিমুলেশনটা হাতেকলমে করছি না, আশা করি তুমি বুঝতে পারছো যে বর্গমূল না পাওয়া পর্যন্ত আমরা এভাবেই খুজতে থাকবো। নিচের কোডটা দেখো:

Python

```

1 def mysqrt(X):
2     low=0.0
3     high=X
4     while high-low>.0001:
5         mid=(low+high)/2
6         print low, high, mid, mid*mid
7         if mid*mid>X:
8             high=mid
9         else:
10            low=mid
11
12    print mid,mid*mid
13    return mid
14
15 mysqrt(15)

```

এই কোডটা চালালে দেখবে বর্গমূল আসছে  $3.87296676636$  যাকে আবার বর্গ করলে পাওয়া যায়  $14.9998715733$ । বুঝতেই পারছো দশমিক সংখ্যার প্রিসিশনের সমস্যার কারণে একদম সঠিক উত্তর পাওয়া যায় নি, কাছাকাছি একটা উত্তর

পাওয়া গিয়েছে। ৪নম্বর লাইনটা খুব গুরুত্বপূর্ণ, এখানে আমরা ঠিক করছি কতক্ষণ আমরা খোজা চালিয়ে যাবো, যত বেশিক্ষণ খুজবো সঠিক উত্তরের তত কাছাকাছি পৌছাতে পারবো। এখানে আমরা high এবং low এর পার্থক্য যতক্ষণ না খুব ছোট হয়ে যাচ্ছে ততক্ষণ খুজতেসি। তুমি .0001 এর জায়গায় আরো কোনো ছোটো সংখ্যা বসালে দেখবে আগের থেকে ভালো ফলাফল পাচ্ছো। যেমন .00000001 ব্যবহার করলে বর্গমূল পাবে 3.87298334594 যাকে আবার বর্গ করলে পাওয়া যায় 14.999999979।

তুমি চাইলে নির্দিষ্ট করে বলে দিতে পারো বাইসেকশন কয়টা ধাপ পর্যন্ত চলবে। তখন কোডটা হবে এরকম:

Python

```

1 def mysqrt(X):
2     low=0.0
3     high=X
4     for step in range(64):
5         mid=(low+high)/2
6         print low, high, mid, mid*mid
7         if mid*mid>X:
8             high=mid
9         else:
10            low=mid
11    print mid,mid*mid
12    return mid
13 mysqrt(15)
14 x
15

```

এবার আমরা high, low এর পার্থক্যের কথা চিন্তা না করেই ৬৪ বার সার্চ করেছি। একটা সংখ্যাকে ৬৪ বার হারিয়ে ভাগ করা মানে সংখ্যাটাকে প্রচন্ডরকম ছোটো করে ফেলা, তাই তুমি এভাবে সঠিক উত্তরের খুব কাছে পৌছে যাবে। তুমি চাইলে ৬৪ বারের জায়গায় ১০০ বা ২০০ বারও ভাগ করতে পারো আরো ভালো ফলাফলের জন্য, তবে সেক্ষেত্রে কোডের রানটাইমও বেড়ে যাবে। ধাপসংখ্যা নির্ধারণ করার সময় high-low এর মান কত বড় এবং একই সাথে প্রবলেমের টাইমলিমিটের দিকে লক্ষ্য রাখা উচিত।

বাইসেকশন ব্যবহার করে জ্যামিতির অনেক সমস্যা সমাধান করা যায়। ধরো তোমাকে নিচের মত একটা ত্রিভুজ দেয়া আছে:

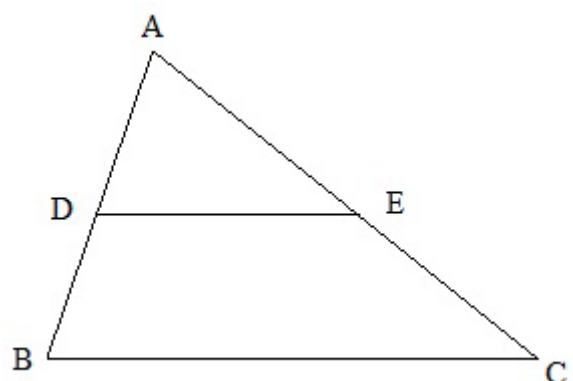
**AB, AC** আর **BC** এর দৈর্ঘ্য তোমাকে দেয়া আছে, আর বলা আছে যে

DE আর BC সমান্তরাল। এছাড়া **ADE** ত্রিভুজ এবং **BDEC**

ট্রাপিজিয়ামের ক্ষেত্রফলের অনুপাত R ও তোমাকে দেয়া আছে।

তোমাকে বলতে হবে **AD** এর দৈর্ঘ্য কত?

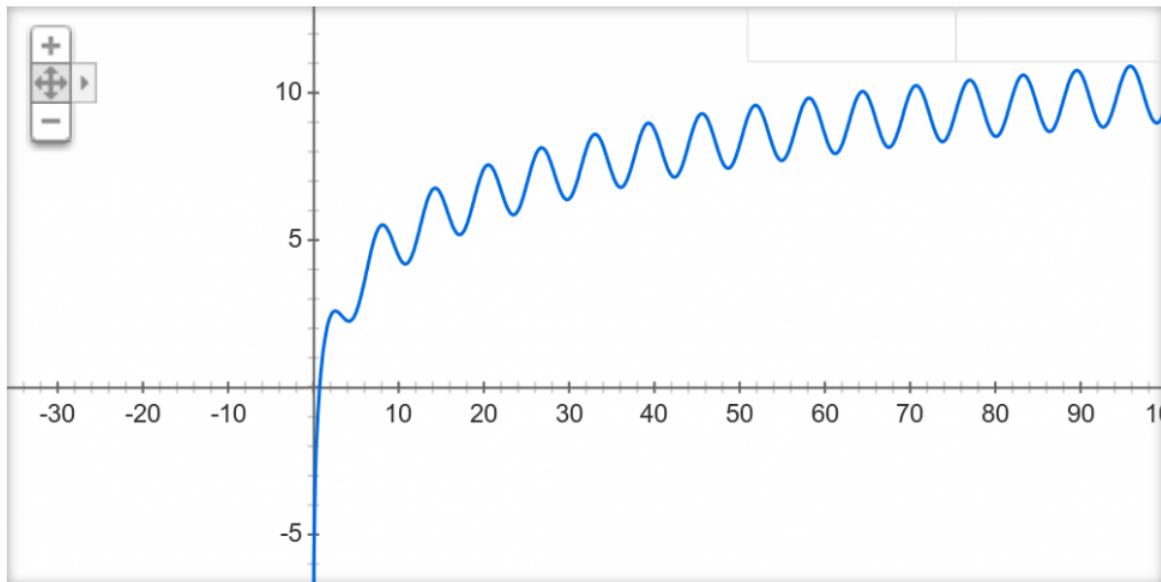
এটা আমরা সহজেই বাইসেকশন দিয়ে সমাধান করতে পারি। তুমি অনুমান করে নাও যে AD এর দৈর্ঘ্য হলো x। এখন AD এর দৈর্ঘ্য জানলে তুমি স্কুলে পড়া ত্রিভুজের অনুপাতের নিয়ম  $(AD/AB=AE/AC=DE/BC)$  দিয়ে সহজেই AE, EC, DE বের করে ফেলতে পারবে। এখন তুমি বাহুর দৈর্ঘ্য দিয়ে ADE এবং BDEC এর ক্ষেত্রফলও সহজে বের করতে পারবে। ক্ষেত্রফল জানার পর অনুপাত বের করে ফেল। যদি দেখ যে তুমি যে অনুপাতটা পেয়েছো সেটা R এর থেকে ছোটো তারমানে তুমি AD এর যে দৈর্ঘ্য x অনুমান করেছিলে সেটা আসল দৈর্ঘ্য থেকে ছোটো। তাহলে তুমি x থেকে high পর্যন্ত রেঞ্জে আবার খুজতে থাকো। আর যদি দেখো যে তোমার পাওয়া অনুপাতটা R এর থেকে বড় তাহলে 0 থেকে x পর্যন্ত রেঞ্জে খুজতে থাকো। high এর মান শুরুতে কত হবে সেটা চিন্তা করে বের করার কাজ তোমার :)।



বাইসেকশন কখন কাজ করবে কখন করবে না এটা বুঝতে পারা খুবই গুরুত্বপূর্ণ। বর্গমূল বের করা শেখার পর তুমি হয়তো মনে করলে  $\sin(x)+5*\log_{10}(x)=5.27$  ইন্টেগ্রেশনটার সমাধান বাইসেকশন দিয়ে করবে। তুমি অনুমান করলে  $x=50$ , তাহলে  $\sin(50)+5*\log_{10}(50)=9.26$ । এখন কি তুমি নিশ্চিত ভাবে বলতে পারো যে উত্তর ৫০ এর বামে আছে? পারবে না, কারণ x

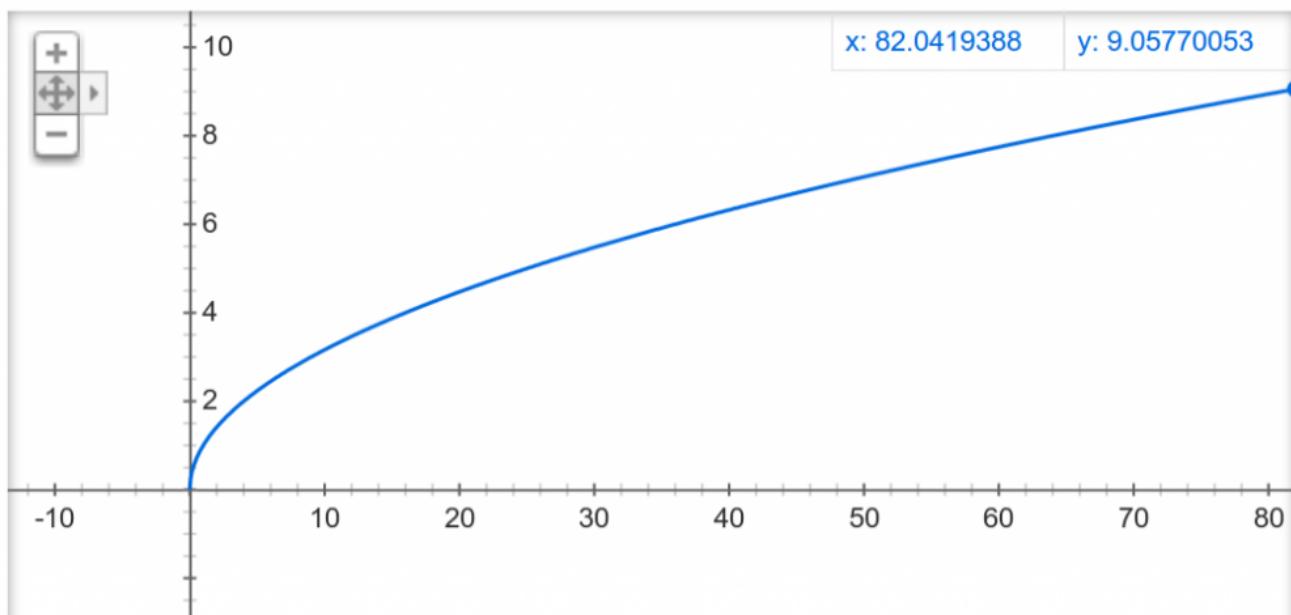
এর মান কমলে  $\sin(x)+5*\log(x)$  এর মান বাড়তেও পারে, কমতেও পারে।  $x$  এর বিভিন্ন মানের জন্য গুগলে গ্রাফটা প্লট করলে তুমি এরকম পাবে:

## Graph for $\sin(x)+5*\log(x)$



এই ইকুয়েশনের ক্ষেত্রে তুমি একটা রেঞ্জের মাঝখানের মান টা নিয়ে কিছু হিসাব-নিকাশ করে বাম বা ডান দিকে অংশ বাতিল করে দিতে পারো না, কারণ তোমার জ্ঞানার উপায় নাই কোনদিকে তোমার উত্তরটা আছে। বর্গমূলক করার সময় ইকুয়েশনটা ছিল  $x^2$ , সেটাকে প্লট করলে পাওয়া যায় এরকম:

## Graph for $\sqrt{x}$



এক্ষেত্রে বাইসেকশন কাজ করেছে কারণ তুমি সহজেই বাম বা ডানের অংশ ফেলে দিতে পেরেছো।

প্রবলেম সলভ করার সময় প্লট বের করা কোনো দরকার নেই, এটা শুধু মাত্র তোমাকে বুঝাতে দেখিয়েছি। তুমি নিশ্চিত হতে পারো যে মাঝখানের মানটা দেখে কোনো একটা অংশ বাতিল করে দেয়া যাবে শুধুমাত্র তখনই বাইসেকশন কাজ করবে। যেমন  $\log(x)+x^2=y$  এই ইকুয়েশন বাইসেকশন দিয়ে সমাধান করতে পারবে কারণ  $x$  এর মান বাড়ার সাথে সাথে  $y$  সবসময় বাড়বে, কিন্তু  $\tan(x)+x^2=y$  এটা বাইসেকশন দিয়ে সমাধান করতে পারবে না।

এবার বাইসেকশন দিয়ে একটা গ্রাফের প্রবলেম সমাধান করি। তোমাকে একটা গ্রাফ দেয়া আছে এরকম:

প্রতিটি এজ একেকটা রাস্তা যার একটা করে ওয়েট আছে। এখন তুমি A থেকে G তে যেতে চাও এমন ভাবে যেন সেই পথে সর্বোচ্চ ওয়েট এর মান যতটা সম্ভব কম হয়। যেমন A->D->F->G পথে সর্বোচ্চ ওয়েট ১১, আবার A->B->E->G পথে সর্বোচ্চ ওয়েট ৯।

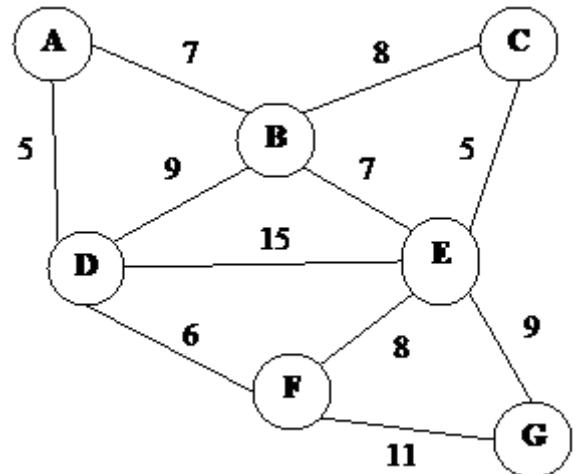
এটা সহজেই বাইসেকশন দিয়ে সমাধান করা সম্ভব। এক্ষেত্রে শুরুতে  $high=15$  কারণ কোনো এজের ওয়েট ১৫ এর থেকে বেশি না,  $low=0$  আর  $mid=7$ । এখন তুমি ৭ এর বড় সবগুলো এজকে গ্রাফ থেকে বাদ দিয়ে দাও আর দেখো যে A থেকে G তে যাওয়ার কোনো রাস্তা আছে নাকি। যদি না থাকে তাহলে ৭ এর বামে উত্তর থাকা সম্ভব না, বামের অংশ বাতিল করে আবার খুজতে থাকো। এক্ষেত্রে বাইসেকশন কাজ করবে কারণ মাঝের মান টা দেখে তুমি বাম বা ডানের অংশ বাতিল করে দিতে পারছো। (এই সমস্যাটা বাইসেকশনের বদলে ডায়াব্লক্স অ্যালগোরিদমকে কিছুটা পরিবর্তন করেও সমাধান করা সম্ভব।)

অনুশীলনের জন্য সমস্যা:

[http://www.lightoj.com/volume\\_showproblem.php?problem=1043](http://www.lightoj.com/volume_showproblem.php?problem=1043)

[http://www.lightoj.com/volume\\_showproblem.php?problem=1076](http://www.lightoj.com/volume_showproblem.php?problem=1076)

[https://uva.onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=35&category=116&page=show\\_problem&problem=989](https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=35&category=116&page=show_problem&problem=989)



# ফ্লয়েডের সাইকেল ফাইন্ডিং অ্যালগোরিদম

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ১৩, ২০১৬

তোমাকে একটা [লিংকড লিস্ট](#) দেয়া আছে, বলতে হবে লিংকড লিস্টে কোনো সাইকেল আছে নাকি। এটা খুবই কমন একটা ইন্টারভিউ প্রশ্ন, আমরা ফ্লয়েডের সাইকেল ফাইন্ডিং অ্যালগোরিদম দিয়ে এই সমস্যাটা সমাধান করা শিখবো।

ছবির [লিংকড লিস্টে](#) দেখা যাচ্ছে ৭ দৈর্ঘ্যের একটা সাইকেল আছে।

সাইকেল ডিটেক্ট করার সবথেকে সহজ উপায় হলো ডিকশনারি বা হ্যাশম্যাপ ব্যবহার করা। প্রথম নোড থেকে এক এক ঘর আগাতে

হবে এবং প্রতিটা নোডকে ডিকশনারিতে সেভ করে রাখতে হবে।

যদি কোনো নোডে গিয়ে দেখা যায় নোডটা আগে থেকেই

ডিকশনারিতে আছে তাহলে বুঝতে হবে লিংকড লিস্টটা সাইক্লিক।

এই অ্যালগোরিদমের টাইম [কমপ্লেক্সিটি](#) আর মেমরি কমপ্লেক্সিটি দুইটাই  $\$O(n)$ ।

মেমরি কমপ্লেক্সিটি  $\$O(1)$  এ নামিয়ে আনা সম্ভব ফ্লয়েডের অ্যালগোরিদম ব্যবহার করে। এই অ্যালগোরিদমটাকে অনেকে “খরগোশ-কচ্ছপ অ্যালগোরিদম” বলে।

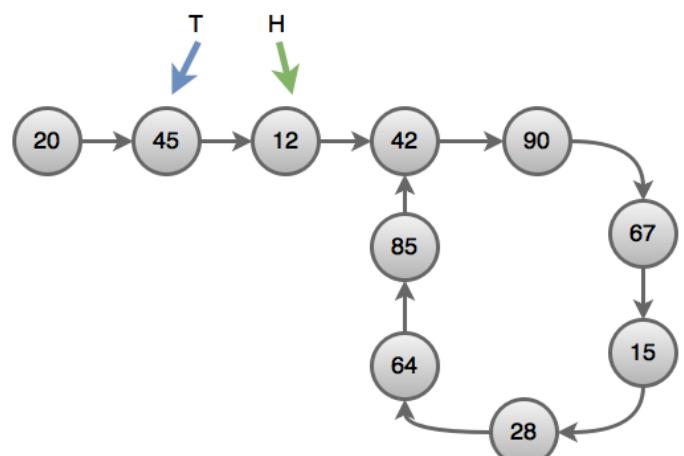
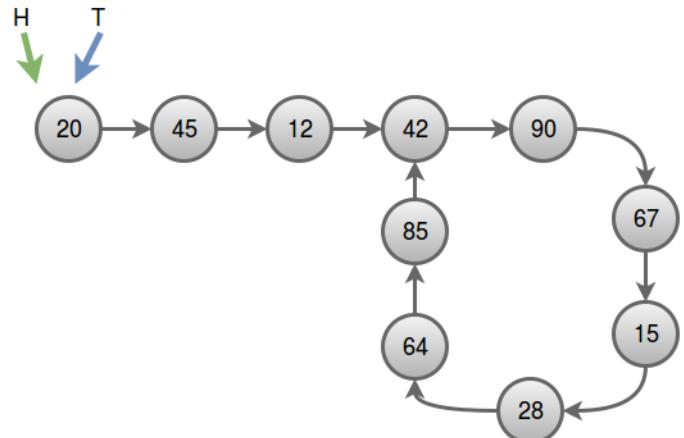
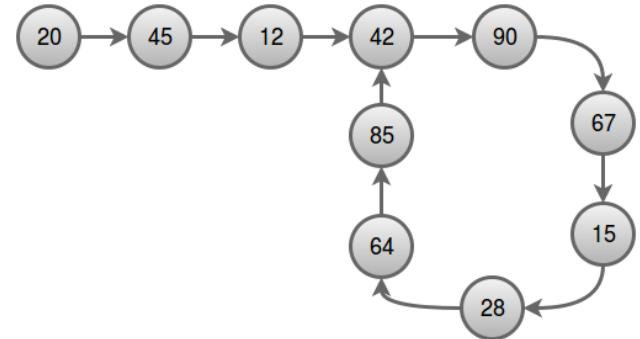
মনে করো আমাদের দুটি পয়েন্টার আছে, একটার নাম কচ্ছপ পয়েন্টার যেটাকে আমরা  $\$T\$$  (Tortoise) দিয়ে চিহ্নিত করবো, আরেকটার নাম খরগোশ পয়েন্টার যেটাকে আমরা  $\$H\$$  (Hare) দিয়ে চিহ্নিত করবো। শুরুতে দুইটা পয়েন্টারই লিংকড লিস্টের রুট নোড এ থাকবে।

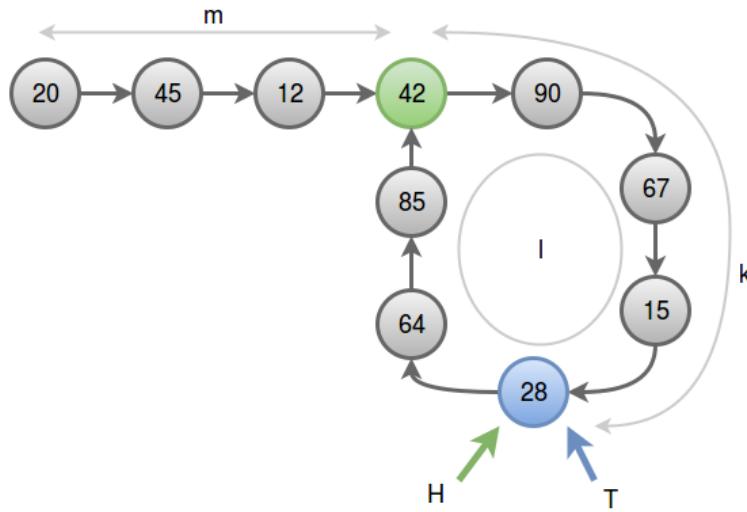
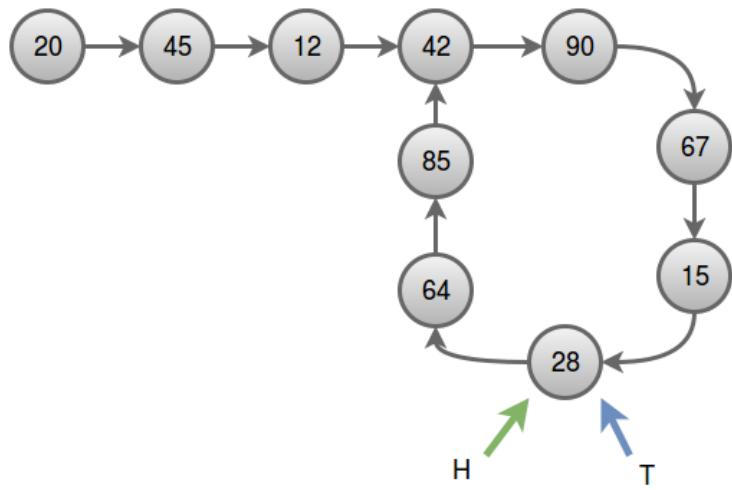
প্রতি সেকেন্ডে খরগোশ আগাবে দুইঘর কিন্তু কচ্ছপ আগাবে মাত্র এক ঘর। তাহলে ১ সেকেন্ড পরে পয়েন্টার দুটোর পজিশন হবে এরকম:

আরো ১ সেকেন্ড পরে  $\$T\$$  থাকবে  $\$12\$$  নম্বর নোডে,  $\$H\$$  থাকবে  $\$90\$$  নম্বর নোডে। এভাবে কয়েক ধাপ হাতে-কলমে সিমুলেট করলে দেখবে দুইটি পয়েন্টারই  $\$28\$$  নম্বর নোডে মিলিত হয়েছে।

দুটি পয়েন্টার একই নোডে মিলিত হওয়ার মানে হলো লিংকড লিস্টে অবশ্যই সাইকেল আছে। সাইকেল না থাকলে  $\$H\$$  পয়েন্টারটি সামনে আগাতে আগাতে লিংকড লিস্টের শেষ মাথায় চলে যেত।

এখন কিভাবে আমরা সাইকেলের প্রথম নোডটা খুজে বের করবো?





মনে করো,

$m$  = \$রুট নোড থেকে সাইকেলের প্রথম নোডের দূরত্ব

$k$  = \$সাইকেলের প্রথম নোড থেকে খরগোশ ও কচ্ছপের মিটিং পয়েন্টের দূরত্ব

$I$  = \$সাইকেলের দৈর্ঘ্য

এখন যদি কচ্ছপ মোট  $c_{\{T\}}$  বার সাইকেলে চক্র খেয়ে খরগোশের সাথে মিলিত হয় তাহলে কচ্ছপের অতিক্রম করা মোট দূরত্ব হবে:

$$D_{\{T\}} = m + c_{\{T\}} * I + k$$

খরগোশ যদি  $c_{\{H\}}$  বার সাইকেলে চক্র খেয়ে কচ্ছপের সাথে মিলিত হয় তাহলে খরগোশের অতিক্রম করা মোট দূরত্ব হবে:

$$D_{\{H\}} = m + c_{\{H\}} * I + k$$

যেহেতু খরগোশের গতি কচ্ছপের দ্বিগুণ তাই আমরা বলতে পারি:

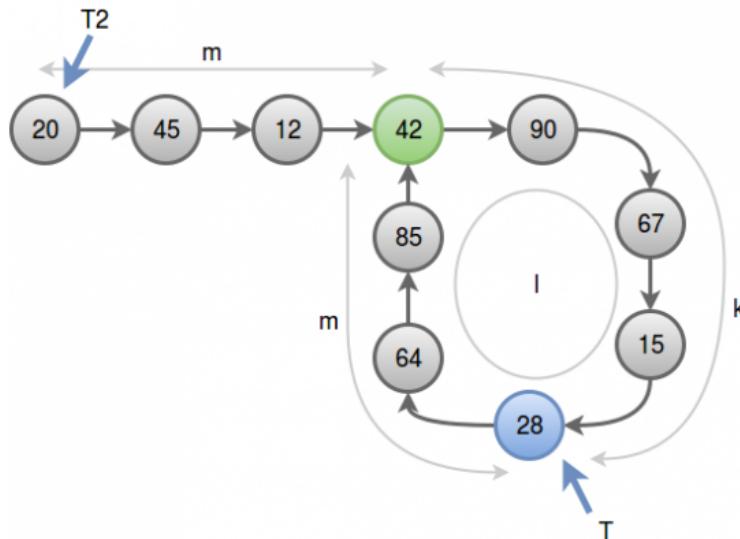
$$2 * (m + c_{\{T\}} * I + k) = m + c_{\{H\}} * I + k$$

এটাকে একটু ঘুরিয়ে লেখা যায়:

$$m + k = (c_{\{H\}} - 2 * c_{\{T\}}) * I$$

$m + k$  হলো সাইকেলের দৈর্ঘ্য, তারমানে  $m + k$  হলো সাইকেলের দৈর্ঘ্যের একটা গুণিতক। আর সেটার মানে হলো যদি তুমি সাইকেলের প্রথম নোড থেকে  $m+k$  দৈর্ঘ্য অতিক্রম করো তাহলে তুমি আবার প্রথম নোডে ফিরে আসবে। এটা বোঝাই এই অ্যালগোরিদমের সবথেকে গুরুত্বপূর্ণ অংশ।

এখন যদি তুমি মিটিং পয়েন্ট থেকে \$m\$ ঘর সামনে যাও তাহলেই তুমি সাইকেলের প্রথম নোডে আবার ফিরে আসবে, কারণ প্রথম নোড থেকে মিটিং পয়েন্টের দূরত্ব  $m$ । কিন্তু তুমি  $m$  বা  $k$  কারো মান ই জানো না, তাহলে কিভাবে  $m$  ঘর সামনে যাবে? সেটার জন্য খুবই সহজ আর মজার একটা উপায় আছে। তোমার নিশ্চয়ই মনে আছে যে রুট নোড থেকে সাইকেলের প্রথম নোডের দূরত্বও  $m$ ।



মনে করো খরগোশ এখন আর নেই, কিন্তু রুট নোড এ নতুন একটা কচ্ছপ পয়েন্টার  $T2$  হাজির হয়েছে, আর  $T$  সেই আগের মিটিং পয়েন্টেই আছে। এখন দুটি পয়েন্টারকেই এক ঘর করে আগাতে থাকলে তারা যেখানে মিলিত হবে সেটাই সাইকেলের প্রথম নোড!

এটাই হলো ফ্লয়েডের সাইকেল ডিটেকশন অ্যালগোরিদম। টাইম কমপ্লেক্সিটি এখনও  $O(n)$  ই আছে ([কেন?](#)) কিন্তু মেমরি কমপ্লেক্সিটি হয়ে গেছে  $O(1)$ ।

লিংকড লিস্টে সাইকেল ডিটেক্ট করা ছাড়াও এই অ্যালগোরিদম অনেক কাজে লাগে। যেমন কোনো গাণিতিক ফাংশন বা pseudo-random নাম্বার জেনারেটরের সাইকেল ডিটেক্ট করা।

অ্যালগোরিদমটা তুমি বুঝেছো নাকি পরীক্ষা করতে [uva 350 pseudo random numbers](#) সমস্যাটা সমাধান করতে পারো।

হ্যাপি কোডিং!



Now mobile friendly!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডি঱েকশন অ্যারে

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

1/25/2013

অনেক সময়ই এমন প্রবলেম থাকে যেখানে বলা হয় তুমি একটা 2-ডি অ্যারের কোনো এক পজিশনে আছো, সেখান থেকে তুমি উপরে-নিচে-বামে-ডানে যেতে পারবে। অথবা দাবার বোর্ডে একটা ঘোড়া আছে, তাকে ৮টা দিকে মুভ করানো যায়, এখন কোনো একটা পজিশনে শৰ্টেস্ট পাথে যেতে হবে। বিগিনার কোডের এ ধরণের প্রবলেমে ছোট একটা ট্রিকস না জানার কারণে কোডের সাইজ বিশাল বানিয়ে ফেলে।

ডি঱েকশন অ্যারের ট্রিকস্টা যারা যানেনা এ ধরণের প্রবলেমে তাদের কোড হয় অনেকটা এরকম:

C++

```

1 int x=5,y=3,row=5,col=5,nx,ny;
2 nx=x+1; ny=y;
3 if(nx>=1 && nx<=row && ny>=1 && ny<=col)
4 {
5     DO SOMETHING
6 }
7 nx=x-1; ny=y;
8 if(nx>=1 && nx<=row && ny>=1 && ny<=col)
9 {
10    DO SOMETHING
11 }
12 nx=x; ny=y+1;
13 .....
14 .....
15 .....

```

এভাবে বারবার একই লাইন লিখতে লিখতে কোডের চেহারা ভয়াবহ হয়ে যায়, আর কাওকে কোড দেখতে দিলে তারও পাগল হবার অবস্থা হয়! ৮ ডি঱েকশনে মুভ করা গেলেতো কথাই নেই। সবথেকে বড় সমস্যা হলো এক জায়গায় চেঞ্চ করলে সবজায়গায় চেঞ্চ করতে হয়।

একটা improvement হতে পারে if(nx>=1 && nx<=row && ny>=1 && ny<=col) এই লাইনের কন্ট্রিনটা একটা ম্যাক্রো বানিয়ে ফেলা। যেমন:

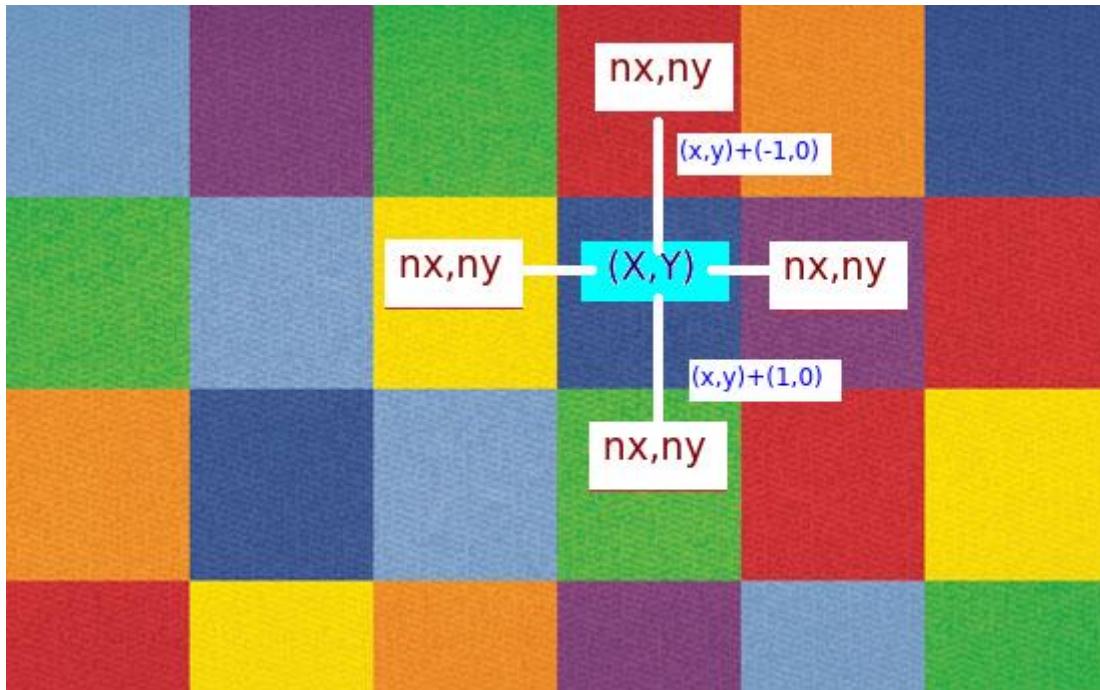
C++

```
1 #define valid(nx,ny) nx>=1 && nx<=row && ny>=1 && ny<=col
```

এখন if(valid) লিখলেই হচ্ছে। এরপরে DO SOMETHING অংশের কাজগুলোও একটা ফাংশন বানিয়ে ফেললে ঝামেলা কিছুটা কমে, এখন খালি ম্যাক্রো বা ফাংশনে চেঞ্চ করলে সব জায়গায় চেঞ্চ হয়ে যাবে। তারপরেও বার বার কন্ট্রিন চেক বা ফাংশন কল করতে হচ্ছে আমাদের। এজন্য আমরা ব্যবহার করবো ডি঱েকশন অ্যারে। ৪ দিকে মুভ করা যায় এটার অর্থ হলো:

1. current row এর সাথে ১ যোগ এবং current col এর সাথে ০ যোগ
2. current row এর সাথে -১ যোগ এবং current col এর সাথে ০ যোগ
3. current row এর সাথে ০ যোগ এবং current col এর সাথে ১ যোগ
4. current row এর সাথে ০ যোগ এবং current col এর সাথে -১ যোগ

তারমানে  $x, y$  পজিশন থেকে উপরে ঘেতে হলে  $(x, y) + (1, 0)$  করতে হবে, নিচে ঘেতে  $(x, y) + (-1, 0)$  করতে হবে, একইভাবে ডানে-বামে ঘেতে ঘোগ করতে হবে শুধু  $y$  এর সাথে।



আমরা দুটি অ্যারে ডিক্লেয়ার করি এভাবে:

C++

```
1 int fx[]={+1,-1,+0,+0};
2 int fy[]={+0,+0,+1,-1};
```

$fx[]$  দিয়ে বুঝাচ্ছি row এর সাথে কত ঘোগ করবো এবং  $fy[]$  দিয়ে বুঝাচ্ছি y এর সাথে কত ঘোগ করবো। এবার কাজ খুব সহজ হয়ে গেলো:

C++

```
1 #define valid(nx,ny) nx>=1 && nx<=row && ny>=1 && ny<=col
2 int x=5,y=3,row=5,col=5,nx,ny;
3 for(int k=0;k<4;k++)
4 {
5     int nx=x+fx[k]; //Add fx[k] with current row
6     int ny=y+fy[k]; //Add fy[k] with current col
7     if(valid(nx,ny)
8     {
9         DO SOMETHING;
10    }
11 }
```

তুমি 8 দিকে ঘেতে চাইলে অ্যারেটা হবে এরকম:

C++

```
1 int fx[]={+0,+0,+1,-1,-1,+1,-1,+1};
2 int fy[]={-1,+1,+0,+0,+1,+1,-1,-1};
```

একটু চিন্তা করলেই তুমি দাবার ঘোড়ার মুভের জন্যেও ডিরেকশন অ্যারে লিখতে পারবে। ৩-ডি তেও এটা কাজ করবে, তখন

`fx[]` নামের আরেকটা অ্যারে লাগবে।

তুমি যদি সম্পূর্ণ কোড চাও তাহলে এই বিএফএস এর কোডটা দেখতে পারো:

C++

```

1 //Problem link: http://acm.timus.ru/problem.aspx?space=1&num=1145
2 #define rep(i,n) for(int i=0; i<(int)n; i++)
3 #define pb(x) push_back(x)
4 #define mem(x,y) memset(x,y,sizeof(x));
5 #define pii pair
6 #define pmp make_pair
7 #define uu first
8 #define vv second
9 using namespace std;
10 #define READ(f) freopen(f, "r", stdin)
11 #define WRITE(f) freopen(f, "w", stdout)
12
13 int fx[]={1,-1,0,0};
14 int fy[]={0,0,1,-1};
15 int mr,mc,mx=0;
16 char w[1000][1000];
17 int d[1000][1000];
18 int r,c;
19 void bfs(int x,int y,int dep)
20 {
21     mem(d,63);
22     d[x][y]=0;
23     queue<pii>q;
24     q.push(pii(x,y));
25     while(!q.empty())
26     {
27         pii top=q.front(); q.pop();
28         if(d[top.uu][top.vv]>mx)
29         {
30             mx=d[top.uu][top.vv];
31             mr=top.uu;
32             mc=top.vv;
33         }
34         rep(k,4)
35         {
36             int tx=top.uu+fx[k];
37             int ty=top.vv+fy[k];
38             if(tx>=0 and tx<r and ty>=0 and ty<c and w[tx][ty]=='.'
39             and d[top.uu][top.vv]+1<=d[tx][ty])
40             {
41                 d[tx][ty]=d[top.uu][top.vv]+1;
42                 q.push(pii(tx,ty));
43             }
44         }
45     }
46     int main(){
47         // READ("in");
48         int sx,sy,cc=0;
49         cin>>r>>c;

```

```
50 swap(r,c);
51 rep(i,r)
52 cin>>w[i];
53 rep(i,r)
54 rep(j,c)
55 if(w[i][j]=='.'){sx=i;sy=j;cc++;}
56 bfs(sx,sy,0);
57 mx=0;
58 bfs(mr,mc,0);
59 if(cc==1) mx=0;
60 cout<<mx<<endl;
61 return 0;
62 }
63
64
65
```

---

এই কোডটা শুধু ডি঱েকশন অ্যারের ব্যবহার দেখানোর জন্য, প্রবলেমটার সলিউশন বের করার কাজ তোমার, ট্ৰি এর [ডায়ামিটাৰ](#) বের করতে বলা হয়েছে প্রবলেমটায়।

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# মিট ইন দ্য মিডল টেকনিক

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

2/3/2014

মিট ইন দ্য মিডল খুবই এলিগেন্ট একটা প্রবলেম সলভিং টেকনিক। এটার কাজ হলো প্রবলেমটাকে ঠিক দুইভাগে ভাগ করে ফেলে তারপর সেই দুইভাগকে কোনোভাবে মার্জ করে প্রবলেমটা সলভ করা। তবে ডিভাইড এন্ড কনকোয়ারের সাথে এটার পার্থক্য হলো ডিভাইড এন্ড কনকোয়ারে দুই ভাগে ভাগ করার পর ছোট ভাগগুলোকে বারবার ভাগ করা হয়, মিট ইন দ্য মিডলে আমরা শুধু একবার ভাগ করবো।

আমরা কিছু প্রবলেম দেখবো যেগুলোকে মিট ইন দ্য মিডলের সাহায্যে সলভ করা সম্ভব।

## প্রবলেম ১: সাম অফ ফোর

(দরকারি নলেজ: বাইনারি সার্চ)

তোমাকে ৪টা  $nn$  সাইজের অ্যারে A,B,C,D দেয়া আছে। প্রতিটা অ্যারে থেকে এক্স্যাক্টলি একটা করে ভ্যালু সিলেক্ট করতে হবে যেন তাদের যোগফল ০ হয়।

যেমন  $n=3, n=3$  এবং  $A=\{1,2,3\}$ ,  $B=\{-1,-4,-5\}$ ,  $C=\{1,5,8\}$ ,  $D=\{9,8,5\}$  হলে  $1+(-4)+8-5=0$  ।  $1+(-4)+8-5=0$  হতে পারে একটা সলিউশন।

## সলিউশন-১

প্রথমেই মাথায় যে সলিউশন আসে সেটা হলো ৪টা নেস্টেড লুপ চালিয়ে সব কম্বিনেশনে সংখ্যাগুলোকে যোগ করে দেখা। এটার কমপ্লেক্সিটি হবে  $O(n^4)O(n^4)$ । যদি  $n=1000, n=1000$  হয় তাহলে প্রোগ্রামটা শেষ হতে কয়েক ঘণ্টা লেগে যাবে।

আমাদের ৪টা মান a,b,c,da,b,c,d দরকার যাতে  $a+b+c+d=0$  হয়। এটাকে লেখা যায়,  $(a+b)=-(c+d)(a+b)=-(c+d)$ । a ভ্যালুটা পাবো A অ্যারে থেকে, b পাবো B অ্যারে, c পাবো C অ্যারে থেকে এবং d পাবো D অ্যারে থেকে।

এখন দুটি নেস্টেড লুপ চালিয়ে A এবং B অ্যারে দিয়ে যতগুলো কম্বিনেশন বানানো যায় সবগুলো বের করে ফেলি। তাহলে আমরা সবগুলো  $(a+b)$  পেয়ে গেলাম। সবগুলো ভ্যালুকে সর্ট করে রাখো বা ম্যাপে ইনসার্ট করে রাখো।

আবার দুটি নেস্টেড লুপ চালিয়ে C এবং D অ্যারে দিয়ে যতগুলো কম্বিনেশন বানানো যায় সবগুলো বের করি। এখন পেলাম সবগুলো  $c+d$ । প্রতিটা  $c+d$  এর জন্য  $a+b$  খুজে বের করো আগের সর্টেড ভ্যালুতে বাইনারি সার্চ চালিয়ে বা ম্যাপের মধ্যে খুজে।

এখন আমাদের কমপ্লেক্সিটি হলো  $O(n^2 * \log n^2)$ । যদি ভ্যালুগুলো ছোটো হয় তাহলে ম্যাপের জায়গায় সাধারণ বুলিয়ান ফ্ল্যাগ ব্যবহার করে  $O(n^2)$  এ সলভ করা সম্ভব।

## প্রবলেম ২: কয়েন চেঞ্জ

(দরকারি নলেজ: বাইনারি সার্চ, ব্যাকট্র্যাকিং অথবা বিটমাস্ক ব্যবহার করে সবগুলো সাবসেট জেনারেশন)

ধরো তোমার কাছে কিছু কয়েন আছে। বলতে হবে এগুলো থেকে কিছু কয়েন ব্যবহার করে নির্দিষ্ট একটা ভ্যালু তৈরি করা যায় নাকি। কোনো কয়েন একবারের বেশি ব্যবহার করা যাবেনা। যেমন কয়েনগুলো যদি হয় ১, ৩, ৬, ১০ তাহলে তুমি ১৩ বা ১১ বানাতে পারবে কিন্তু ৫০ বা ২ কিছুতেই বানাতে পারবেনা। কয়েনের সংখ্যা সর্বোচ্চ ৩০টা।

## সলিউশন

মনে করি কয়েনগুলার মান হতে পারে ১ থেকে ১০০ পর্যন্ত। তুমি যদি ডাইনামিক প্রোগ্রামিং জানো তাহলে নিশ্চয়ই ভাবছো এটাতে খুব সহজ প্রবলেম, সবগুলো কয়েনের ভ্যালুর যোগফল সর্বোচ্চ হতে পারে  $30 * 100$ , তাহলে  $30 * (30 * 100)$  কমপ্লেক্সিটিতে খুব সহজে প্রবলেমটা সলভ করা যাবে। এবার আমি প্রবলেমটাকে কঠিন করে দেই, ধরি কয়েনগুলোর ভ্যালু হতে পারে ১ থেকে  $10^{19}$ । এখন কমপ্লেক্সিটি হয়ে গেলো  $30 * (30 * 10^{19})$ । এটা ডাইনামিক প্রোগ্রামিং দিয়ে সলভ করা সম্ভবনা, মেমরি বা টাইম কোনোটাতেই কুলিয়ে উঠবেনা। এখন কি করা যেতে পারে? এখানে লক্ষ্য করার বিষয় হলো কয়েনের সংখ্যা অনেক কম!

কয়েনের সংখ্যা মাত্র ৩০ হলেও ৩০টা কয়েনের একটা সেটের সাবসেট সংখ্যা  $2^{30}$ । তারমানে সর্বোচ্চ  $2^{30}$ টা ভিন্ন ভ্যালু তৈরি করা যাবে কয়েনগুলো দিয়ে। তাই ব্যাকট্র্যাকিং করে সবগুলো ভ্যালু জেনারেট করা পসিবল না। কিন্তু কয়েন যদি ১৫টা

হতো তাহলে  $2^{n/2} = 32768$  টা সাবসেট থাকতো এবং সবগুলো ভ্যালু আমরা জেনারেট করতে পারতাম ব্যাকট্র্যাক করে! এই পর্যন্ত পড়ার পর একটু থেমে কিছুক্ষণ চিন্তা করো কিভাবে প্রবলেমটা সলভ করা সম্ভব। এরপরে নিচের অংশ পড়ো।

আমরা কয়েনগুলোকে দুই ভাগে ভাগ করে ফেলি। প্রতিটা ভাগে আছে ১৫টা করে কয়েন। এখন প্রথম ভাগের ১৫টা কয়েন নিয়ে সবগুলো ভ্যালু জেনারেট করে একটা অ্যারেতে রেখে দাও, মনে করি অ্যারেটার নাম A। একই ভাবে ২য় ১৫টা কয়েন নিয়ে সবগুলো ভ্যালু জেনারেট করে আরেকটা অ্যারেতে রেখে দাও, মনে করি অ্যারেটার নাম B। B অ্যারেটাকে সর্ট করে ফেলো, A কে সর্ট করার দরকার নেই।

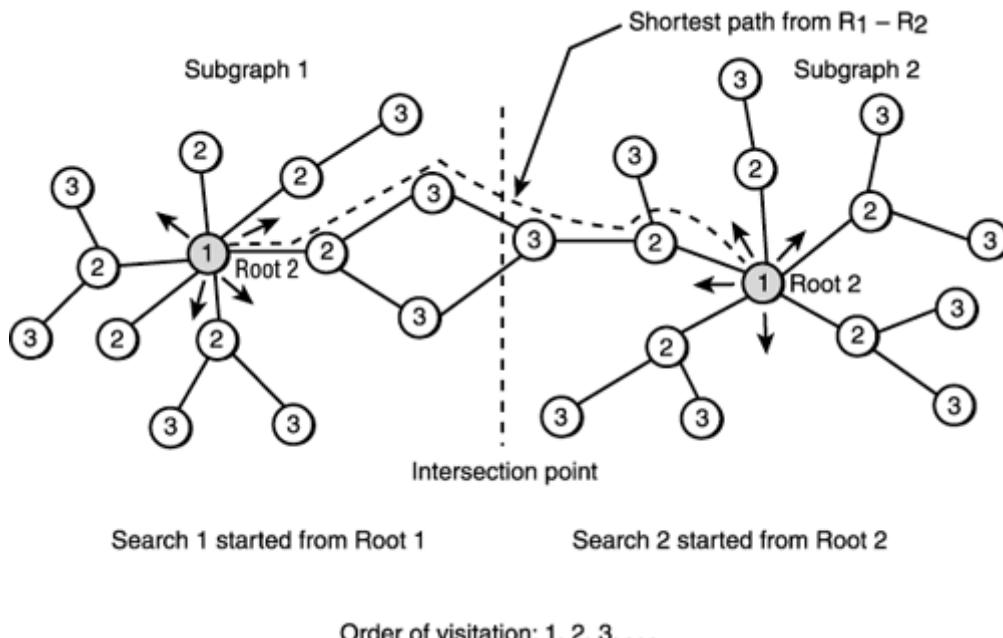
এখন বাকি কাজ সহজ। মনে করো তোমাকে X ভ্যালুটা বানাতে হবে। A অ্যারের উপর লুপ চালিয়ে সবগুলো ভ্যালু চেক করো। তাম ভ্যালু যদি হয় A[i] তাহলে তুমি চেক করো B অ্যারেতে X-A[i] ভ্যালুটা আছে নাকি। যদি থাকে তাহলে তুমি X বানাতে পারবে!! B তুমি সর্ট করে রেখেছো, তাহলে বাইনারি সার্চ করেই X-A[i] আছে নাকি চেক করতে পারবে। একটু ভাবলেই বুঝতে পারবে A অ্যারেতে প্রথম ১৫টা সবরকম কম্বিনেশন আছে, তাই X-A[i] কে A তে খোজার কোনো দরকার নাই।

$n$  টা কয়েনের জন্য তাহলে কমপ্লেক্সিটি হবে  $O(2^{(n/2)} * \log 2^{(n/2)})$ , কারণ আমরা  $2^{(n/2)}$  টা ভ্যালুর জন্য বাকি অর্ধেকের উপর বাইনারি সার্চ করছি।

### প্রবলেম ৩: বাইডিগ্রাফে সর্ট করা সার্ট

(দরকারি নলেজ: গ্রাফ থিওরি)

একটা গ্রাফে দুটি নোড দেয়া আছে, নোড দুটির মধ্যে শর্টেস্ট পাথ বের করতে হবে।



### সলিউশন:

মোটামুটি নিশ্চিত গ্রাফ থিওরি যারা পারো তারা মাথা চুলকিয়ে ভাবছো, এটা কোনো প্রবলেম হলো? বিএফএস দিয়েইতো সলভ করা যায়। কিন্তু এখানে আমরা চিন্তা করছি বিশাল গ্রাফের কথা যেখানে বিলিয়ন বিলিয়ন নোড আর এজ থাকতে পারে।

বিএফএস এ প্রতিটা নোড থেকে তার অ্যাডজেসেন্ট নোডগুলোতে যেতে হয়, সেগুলো থেকে আবার তার অ্যাডজেসেন্ট নোডগুলোতে যেতে হয়। প্রতিটা নোডের অ্যাডজেসেন্ট নোড সংখ্যাকে বলা হয় নোডটা ডিগ্রী। এখন যদি প্রতিটা নোডের এভারেজ ডিগ্রী হয়  $p$  আর শর্টেস্ট পাথ যদি হয়  $k$  তাহলে তোমাকে মোটামুটি  $O(p^k)$  টা নোড এক্সপ্লোর করতে হবে।

যদি তুমি সোর্স আর ডেস্টিনেশন দুই পাশ থেকে সার্চ শুরু করো যতক্ষণনা তারা একসাথে মিলছে তাহলে কমপ্লেক্সিটি হয়ে যাবে  $O(p^{k/2})$ । আগের কয়েন চেষ্টা প্রবলেম সলভ করেই বুঝতে পারছো এই অর্ধেক হওয়াটা এক্সপোনেনশিয়াল কমপ্লেক্সিটির ক্ষেত্রে এটা খুবই সিগনিফিকেন্ট উন্নতি।

গ্রাফ রিলেটেড আরো দুটি ইন্টারেস্টিং প্রবলেম হলো:

- ফেসবুকের গ্রাফে দুজন ইউজারের মধ্যে মিউচুয়াল ফ্রেন্ড আছে নাকি বের করতে হবে
- ধারণা করা হয় সোস্যাল নেটওয়ার্কে যেকোনো দুজন মানুষের দূরত্ব সর্বোচ্চ ৬টি নোড। দুটি ইউজারনেম ইনপুট দিলে কিভাবে এটা ভেরিফাই করবে?

**রেফারেন্স:** ইনফো এরিনা

**কৃতজ্ঞতা:** মাহবুবুল হাসান শান্ত ভাইকে ধন্যবাদ সাম অফ ফোর প্রবলেমের সলিউশনে ভুল ধরিয়ে দেয়ার জন্য।

সলভ করার জন্য প্রবলেম:

[Coin Change\(IV\)](#)

[Sum of Four](#)

[Funny Knapsack](#)

কোন প্রশ্ন থাকলে কমেন্টের ঘরে জোনাও বা যোগাযোগ করো: shafaet[dot]csedu[dot]com। অনুরোধ করছি যথেষ্ট চিন্তাভাবনা আর চেষ্টা না করে কোনো কোড ডিবাগ করতে পাঠাবেন।

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ব্যাকট্র্যাকিং: পারমুটেশন জেনারেটর

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

সেপ্টেম্বর ২০, ২০১৬

[পুরানো লেখা, নতুন করে গুছিয়ে লিখে আবার প্রকাশ করা হলো]

ব্যাকট্র্যাকিং একধরণের ব্রুটফোর্স টেকনিক। ব্রুটফোর্সের মতই এটা সম্ভাব্য সবধরণের বিন্যাস-সমাবেশ থেকে ফলাফল খুজে নিয়ে আসে। যেমন ধর তোমাকে ঢাকা থেকে চট্টগ্রামার ঘাবার সবথেকে ছেটো পথ খুজে বের করতে বলা হলো। তুমি ডায়ালক্স্ট্রার দেয়া অ্যালগোরিদম ব্যবহার না করে উত্তরা থেকে শাহবাগে ঘাবার ঘত পথ আছে সবগুলো খুজে বের করলে এবং তারপর তারমধ্যে থেকে সবথেকে ছেটো কোনটা সেট বের করলে, এটা হলো ব্রুটফোর্স বা কমপ্লিট সার্চ।

এই লেখাটা পড়ার আগে অবশ্যই রিকার্সন সম্পর্কে ভালো ধারণা থাকতে হবে।

সার্চস্পেসের আকার ছেটো হলো এটা খুবই কার্যকর একটা পদ্ধতি। সার্চস্পেস হলো কতটুকু অংশজুড়ে তোমার সলিউশন থাকতে পারে সেইটুকু। যেমন তোমাকে যদি  $10^{\text{সাইজের দুটি সেট}} \approx 2^{10}$  সেট দিয়ে বের করতে বলে ২য় সেট ১মটির সাবসেট কিনা তাহলে খুব সহজে তুমি ব্যাকট্র্যাক করে  $2^{10} = 1024$  টি সেট বের করে সবগুলোর সাথে মিলিয়ে দেখতে পারো, কিন্তু সেটের আকার  $100$  হলে তোমার এভাবে সলিউশন বের করার জন্য এই জীবনকাল যথেষ্ট নয়!

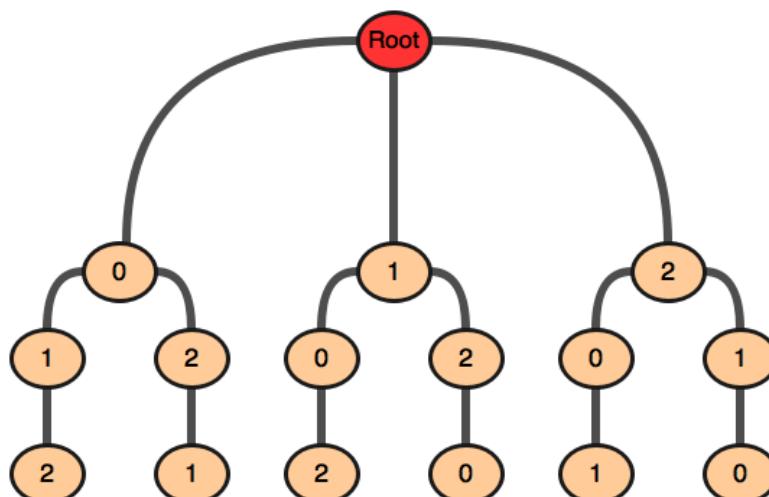
যেসব প্রবলেমের পলিনমিয়াল কোনো সলিউশন আমরা এখনো জানিনা অর্থাৎ **NP বা NP-hard ক্যাটাগরির** প্রবলেম সেগুলোকে ব্যাকট্র্যাক করেই সমাধান করতে হয়। ব্যাকট্র্যাকিং কোড লেখার আগে **কমপ্লেক্সিটির** ব্যাপারে খুব সতর্ক থাকতে হবে।

এখন আমরা দেখবো কিভাবে ব্যাকট্র্যাক করে 00 থেকে  $n-1$  পর্যন্ত সংখ্যাগুলোর প্রতিটি পারমুটেশন বের করা যায়। যদি  $n=2$  হয় তাহলে পারমুটেশনগুলো হবে:

সহজে বোঝার জন্য আমরা পারমুটেশনগুলোকে একটা ট্রি আকারে দেখি:

| Position | 0 | 1 | 2 |
|----------|---|---|---|
| 0        | 0 | 1 | 2 |
| 0        | 0 | 2 | 1 |
| 1        | 0 | 2 | 0 |
| 1        | 2 | 0 | 0 |
| 2        | 0 | 1 | 0 |
| 2        | 1 | 0 | 0 |

Permutations



এই ট্রি তে রুট থেকে যেকোনো পথে আগামে থাকলে একটা করে পারমুটেশন পাওয়া যাবে। আমরা একটা রিকার্সিভ ফাংশন লিখবো যেটা এই ট্রি এর সবগুলো পথে একবার করে ঘুরে আসবে। আমাদেরকে সত্যি সত্যি অ্যাডজেসেন্সি ম্যাট্রিক্স দিয়ে ট্রি বানিয়ে ফেলা দরকার নেই, ছবিটা দেয়া হয়েছে সহজে বোঝার জন্য।

মনে করো আমাদের ফাংশনের নাম হলো generate(idx)। idx মানে হলো এখন আমরা idx তম পজিশনে একটা সংখ্যা বসাতে চাই। নিচের সুড়োকোডটি দেখো, এরপর আমি ব্যাখ্যা করছি:

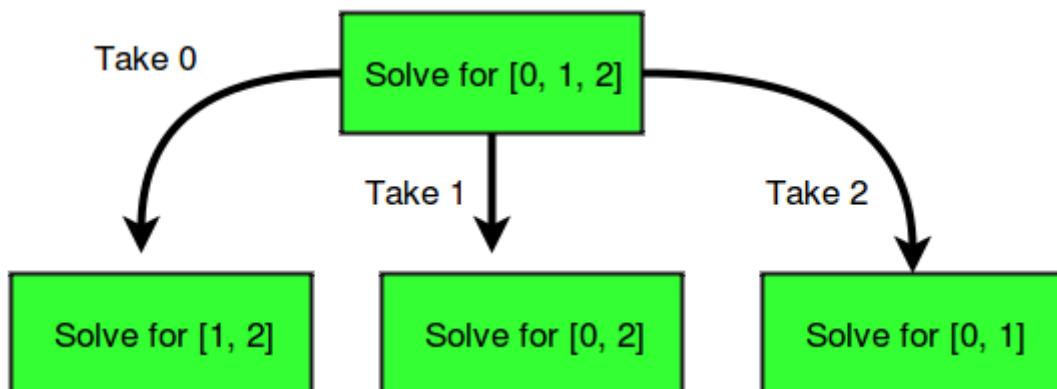
```

1 procedure generate(idx)
2     if idx == n
3         print position
4         return
5     for i from 0 to n
6         if taken[i] = False
7             taken[i] <- true
8             position[idx] = i
9             generate(idx+1)
10            taken[i] <- false

```

৫ নম্বর লাইনে আমি ০০ থেকে  $n$  পর্যন্ত একটি লুপ চালিয়েছি। ৬ নম্বর লাইনে দেখছি যে ii সংখ্যাটি এরই মধ্যে নেয়া হয়েছে নাকি। যদি না নেয়া হয়ে থাকে তাহলে idx তম পজিশনে ii বসিয়ে রিকার্সিভলি  $idx + 1$  বাকি পজিশনগুলোর জন্য প্রবলেমটা সলভ করছি।

ইন্টারেস্টিং ব্যাপার হলো রিকার্সিভলি generate( $idx + 1$ )। এর জন্য সলভ করার পর আমরা taken[i] <- false করে দিচ্ছি। কারণ ii তম সংখ্যাটি  $idx$  পজিশনে বসানোর পর আবার সেই পজিশন থেকে ii কে ফেলে দিয়ে  $i + 1$  কে একই পজিশনে বসিয়ে রিকার্সিভলি সলভ করবো।



ব্যাকট্র্যাকিং করার জেনারেল আইডিয়াটা হলো:

১. প্রতিটি ফাংশন কলে সম্ভাব্য অপশনগুলোর একটি বাছাই করো।
২. বাকি অপশনগুলো থেকে রিকার্সিভলি সমাধান বের করার চেষ্টা করো।
৩. বাছাই করা অপশনটি ফেলে দিয়ে অন্য আরেকটি নিয়ে আবার চেষ্টা করো।

এখন তোমার কাজ হবে সুড়োকোডটাকে আসল কোডে রূপান্তর করা। যদি এটা পারো তাহলে তুমি আরো কিছু একইরকম সমস্যা সহজেই সমাধান করতে পারবে। যেমন তোমাকে যদি একটা স্ট্রিং "abcd" দিয়ে বলা হয় সবরকম পারমুটেশন জেনারেট করতে তাহলে একইভাবে সহজেই করতে পারবে। কিন্তু স্ট্রিংটায় যদি এই ক্যারেক্টার একাধিকবার থাকে (যেমন "abbcdd") তাহলে একটু ঝামেলায় পড়বে, দেখবে একই পারমুটেশন বারবার জেনারেট হচ্ছে। এটা এড়তে তোমাকে একটা ফ্ল্যাগ রেখে একই পজিশনে একই ক্যারেক্টার একাধিকবার বসিয়ে রিকার্সিভলি সলভ করা বন্ধ করতে হবে।

প্র্যাকটিসের জন্য নিচের সমস্যাগুলো সমাধান করো:

[Determine The combination](#)  
[Prime Ring problem](#)

House of santa clause

All Walks of length n

Following orders

আপাতত এখানেই শেষ, পরের পর্বে ব্যাকট্র্যাকিং দিয়ে সমাধান করা যায় এমন কিছু সমস্যা দেখবো।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# HANDBOOK OF ALGORITHMS

## Section Number Theory

*Courtesy of*  
*Shafaet Ashraf*

মডুলার অ্যারথিমেটিকি \_ শাফায়তেরে ব্লগ.pdf

প্রাইম জনোরটের (Sieve of Eratosthenes) \_ শাফায়তেরে ব্লগ.pdf

বাটিওয়াইজ সভি(Bitwise sieve) \_ শাফায়তেরে ব্লগ.pdf

কম্বনিটেরকিস\_ অ্যারেওজমনেট এবং ডি-রেওজমনেট গণনা \_ শাফায়তেরে ব্লগ.pdf

# মডুলার অ্যারিথমেটিক

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

1/20/2012

-১৭-১৭ কে ৫৫ দিয়ে ভাগ করলে ভাগশেষ কত হয়? ২১০০০২১০০০ কে ১৭১৭ দিয়ে ভাগ করলে ভাগশেষ কত হয় সেটা কি তুমি ওভারফ্লো এড়িয়ে নির্ণয় করতে পারবে?  $O(n)O(n)$  এ পারলে  $O(\log 2n)O(\log 2n)$  কমপ্লেক্সিটিতে পারবে? যদি কোনো একটি উত্তর “না” হয় তাহলে এই পোস্ট তোমার জন্য। তবে তুমি যদি মডুলার ইনভার্স বা এডভান্সড কিছু শিখতে পোস্টটি খুলো তাহলে তোমাকে আপাতত হতাশ করতে হচ্ছে।

সি/জাভা সহ বেশিভাগ প্রোগ্রামিং ল্যাংগুয়েজে এ %% কে ভাগশেষ অপারেটর ধরা হয়। xx কে mm দিয়ে ভাগকরে ভাগশেষ বের করার অর্থ  $x \% m$  এর মান বের করা অথবা আমরা বলতে পারি xx কে mm দিয়ে mod করা। “determine answer modulo 1000” এ কথাটির অর্থ হলো উত্তরকে ১০০০১০০০ দিয়ে mod করে তারপর আউটপুট দিতে হবে।

একটি সমস্যা দিয়ে শুরু করি। তোমার ১০০টি বই আছে, তুমি কয়বাবে বইগুলো সাজাতে পারবে? খুব সহজ,  $100!100!$  ( $100$  ফ্যাক্টরিয়াল) ভাবে সাজাতে পারবে।  $100!100!$  ডিজিটের বিশাল একটি সংখ্যা। তাই আমি তোমাকে প্রবলেমটা সহজ করে দিলাম, ধরো তুমি xx উপায়ে বইগুলো সাজাতে পারবে, তাহলে তোমাকে xx কত সেটা বলতে হবে। অর্থাত  $100!$  বের করে ৯৭৯৭ দিয়ে ভাগ করে ভাগশেষটা বের করাই তোমার সমস্যা। (Determine 100 factorial modulo 97)

এটা কিভাবে করবে?  $100!100!$  এর মান তুমি বের করতে পারবেনা ৬৪৬৪৬৪টি আনসাইনড ইন্টিজার দিয়েও, এরা ২৬৪-১২৬৪-১ পর্যন্ত সংখ্যা নিয়ে কাজ করতে পারে, তাই ওভারফ্লো হবে। কিন্তু আমরা জানি আমাদের উত্তর কখনোই 97 এর বড় হবেনা কারণ কোনো সংখ্যাকে m দিয়ে mod করা হলে সংখ্যাটি m এর থেকে বড় হতে পারবেনা।

আমরা এ ধরণের সমস্যা সমাধান করতে সাহায্য নিবো দুটি সুত্রে:

$$(a+b)\%m=((a\%m)+(b\%m))\%m \quad (a+b)\%m=((a\%m)+(b\%m))\%m$$

$$(a * b)\%m=((a\%m)*(b\%m))\%m \quad (a * b)\%m=((a\%m)*(b\%m))\%m$$

nn সংখ্যক নম্বর a1,a2...ana1,a2...an এর জন্য সুত্র দুটি ব্যবহার করতে পারবে।

উপরের সমস্যাটিতে ২য় সুত্রটি লাগবে। তোমার বের করা দরকার  $100! \% 97$  অর্থাত:

$(100*99*98*.....*1)\%97$

তুমি যেটা করবে সেটা হলো গুণ করার সময় ২য় সুত্রের মত করে mod করতে থাকবে, তাহলে কোনো সময়ই overflow ঘটবেনা কারণ mod করলে প্রতি স্টেপে সংখ্যাটি ছোটো হয়ে যাচ্ছে। এটার কোড হতে পারে এরকম:

C++

```

1 int fact=1;
2 for(int i=1;i<=100;i++)
3 {
4 fact=((fact%97)*(i%97))%97;
5 }
6 printf("%d\n",fact);
7

```

এটার আউটপুট আসবে ০। অর্থাত  $100! \% 97 = 0$ । একটু খেয়াল করলেই বুঝবে এখানে আমরা ২য় সুত্রটি প্রয়োগ করেছি এটি করে সংখ্যা নিয়ে।

সুত্র দুটি কেনো কাজ করে সেটা জানা দরকার। আমি ১ম সুত্রটির প্রমাণ দেখাচ্ছি, ২য়টিও একইভাবে করা যায়। প্রমানটি আমার নিজের মত করে করা।

ধরি  $(x+y)\%5$  এর মান আমাদের বের করতে হবে। এখন যদি  $x\%5=c1$  আর  $y\%5=c2$  হয়, তাহলে  $xx$  কে আমরা লিখতে পারি  $5n1+c15n1+c1$  এবং  $yy$  কে লিখতে পারি  $5n2+c25n2+c2$  যেখানে  $n1n1$  আর  $n2n2$  দুটি ইন্টিজার। এটা একদম বেসিক রুল, আশা করে বুঝতে সমস্যা হচ্ছেনা। এখন:

$$\begin{aligned} & (x+y)\%5(x+y)\%5 \\ &= (5n1+c1+5n2+c2)\%5(5n1+c1+5n2+c2)\%5 \\ &= (5n1+5n2+c1+c2)\%5(5n1+5n2+c1+c2)\%5 \quad (1) \end{aligned}$$

এখানে  $5n1+5n2$  অবশ্যই 55 এর মাল্টিপল, তাই আমরা লিখতে পারি

$5n1+5n2=5N$   $n1+n2=N$  যেখানে  $N=n1+n2$

এবং  $c1+c2=C$

তাহলে (1) থেকে পাচ্ছি:

$$(5N+C)\%5(5N+C)\%5$$

এখন পরিষ্কার বোৰা যাচ্ছে যে উত্তর হলো  $C\%5C\%5$ ।  $CC$  কে আবার mod করতে হলো কারণ  $c1+c2c1+c2$  এর মান 5 এর থেকে বড় হতেই পারে। এখন

$$\begin{aligned} & ((x\%5)+(y\%5))\%5((x\%5)+(y\%5))\%5 \quad (2) \\ & = ((5n1+c1)\%5)+((5n2+c2)\%5))\%5((5n1+c1)\%5)+((5n2+c2)\%5))\%5 \\ & (5n1+c1)\%5=c1(5n1+c1)\%5=c1 \\ & (5n2+c2)\%5=c2(5n2+c2)\%5=c2 \end{aligned}$$

তাহলে 2 কে লিখতে পারি:

$$(c1+c2)\%5=C\%5(c1+c2)\%5=C\%5$$

তাহলে 1ম সুত্রটি প্রমাণিত হলো। তারমান যোগ করে mod করা আর আগে mod করে তারপর যোগ করে আবার mod করা একই কথা। সুবিধা হলে সংখ্যাটি কোনো স্টেপেই বেশি বড় হতে পারেনা। গুণের ক্ষেত্রেই একই সুত্র প্রযোজ্য।

নেগেটিভ সংখ্যার mod নিয়ে একটু আলাদা ভাবে কাজ করতে হয়। সি তে  $-17\%5-17\%5$  এর মান দেখায় -2। কিন্তু সচরাচর আমরা ভাগশেষের যে সংজ্ঞা ব্যবহার করি তাতে  $x\%m=px\%m=p$  হলে গাণিতিকভাবে

$m$  এর সবথেকে বড় থেকে বড় মাল্টিপল যেটা  $xx$  এর থেকে ছোট সেই সংখ্যাটিকে  $xx$  থেকে বিয়োগ করলে যে সংখ্যাটি পাওয়া যায় সেটাই  $p$ ।

যেমন  $23\%523\%5$  এর ক্ষেত্রে  $5\times8=20$   $5\times8=20$  হলো 55 এর সবথেকে বড় মাল্টিপল যেটা 2323 এর থেকে ছোট, তাই  $23\%5=23-(5\times4)=3$ ।  $23\%5=23-(5\times4)=3$ ।  $-17\%5-17\%5$  এর ক্ষেত্র খেয়াল করো -20-20 হলো 55 এর সবথেকে বড় মাল্টিপল যেটে -17-17 থেকে ছোট, তাই উত্তর হবে 33।

এই কেসটা handle করা একটি উপায় হলো নেগেটিভ সংখ্যাটিকে একটি 55 এর মাল্টিপল এর সাথে যোগ করা যেন সংখ্যাটি 0 থেকে বড় হয়ে যায়, তারপরে mod করা। যেমন:

$$\begin{aligned} & -17\%5 \\ & =(-17+100)\%5 \\ & =83\%5 \\ & =3 \end{aligned}$$

এটা উপরের সুত্রের প্রমাণের মত করেই কাজ করে, একটু গুতালেই প্রমাণ করতে পারবে। নেগেটিভ সংখ্যার mod নিয়ে কনটেস্ট সবসময় সতর্ক থাকবে, এটা wrong answer খাওয়ার একটা বড় কারণ হতে পারে।

এবার আসি সুপরিচিত big mod সমস্যায়। সমস্যাটি হলো তোমাকে  $(ab) \% m$  এর মান বের করতে হবে, aa, bb, mm তোমাকে বলে দেয়া হবে, সবগুলোর range  $2^{31}$  পর্যন্ত হতে পারে।  $100! \% 97$  বের করার মত করে সহজেই তুমি overflow না খেয়ে মানটি বের করতে পারবে, সমস্যা হলো তুমি লুপ চালিয়ে একটি একটি গুণ করে  $220000000000220000000000$  বের করতে চাইলে উত্তর পেতে সম্ভবত নাস্তা শেষ করে আসতে পারবে। আমরা চাইলে  $O(\log 2n)O(\log 2n)$  এ এটা করতে পারি।

লক্ষ করো

$$\begin{aligned} & 21002100 \\ & =(250)2(250)2 \\ & \text{এবং} \\ & (250)(250) \\ & =(225)2(225)2 \end{aligned}$$

এখন বলো 250250 বের করতে কি 226226, 227227 ইত্যদি বের করার দরকার আছে নাকি 225225 পর্যন্ত বের করে square করে দিলেই হচ্ছে? আবার 225225 পর্যন্ত আসতে  $(212)2(212)2$  পর্যন্ত বের করে square করে সাথে 2 গুণ করে দিলেই যথেষ্ট, অতিরিক্ত 2 গুণ করছি সংখ্যাটি বিজোড় সে কারণে। প্রতি স্টেপে গুণ করার সময় mod করতে থাকবে যাতে overflow না হয়। recursion ব্যবহার করে কোডটি লেখা জলের মত সোজা:

C++

```

1 #define i64 long long
2 i64 M;
3 i64 F(i64 N,i64 P)
4 {
5     if(P==0) return 1;
6     if(P%2==0)
7     {
8         i64 ret=F(N,P/2);
9         return ((ret%M)*(ret%M))%M;
10    }
11    else return ((N%M)*(F(N,P-1)%M))%M;
12 }
13

```

মন্তব্য অংশে “হাসান” একটি বিগ মডের সুন্দর রিকার্শন-ট্রি এর ছবির লিংক দিয়েছে, ছবিটা এরকম:

মডুলার অ্যারিথমেটিক ব্যবহার করে বিশাল আকারের ফলাফল কে আমরা ছোট করে আনতে পারি ফলাফলে বিভিন্ন প্রোপার্টি'কে নষ্ট না করে, তাই এটা গণিতে খুব গুরুত্বপূর্ণ। প্রোগ্রামিং কন্টেস্টে প্রায়ই বিভিন্ন প্রবলেমে মডুলার অ্যারিথমেটিক প্রয়োজন পড়বে, বিশেষ করে counting আর combinatorics এ যেখানে ফলাফল অনেক বড় হতে পারে, ফ্যাক্টরিয়াল নিয়ে কাজ করতে হতে পারে।

ভাগ করার সময় গুণ, আর যোগের মত সুত্র দুটি কাজ করেনা, এটার জন্য তোমাকে extended euclid আর modular inverse জানতে হবে।

সিপিউর জন্য mod খুব costly একটা অপারেশন। যোগ, গুণের থেকে mod করতে অনেক বেশি সময় লাগে। অপ্রয়োজনে mod ব্যবহার করলে কোড time limit exceed করতে পারে, তাই overflow হবার আশংকা না থাকলে সব জায়গায় mod করা দরকার নেই। আমার একটি কোড 3সেকেন্ডে time limit exceed হবার পর খালি কিছু mod সরিয়ে ১.৩ সেকেন্ড নামিয়ে এনেছি।

এখন চিন্তা করার জন্য একটি প্রবলেম। ধরো তোমাকে একটি অনেক বড় সংখ্যা(bignum) দিয়ে সেটাকে ২৩১২৩১ এর ছোট একটি সংখ্যা দিয়ে mod করতে বলা হলো।  $O(\text{length\_of\_bignum})O(\text{length\_of\_bignum})$  কমপ্লেক্সিটিতে কিভাবে করবে? সাহায্য:

$$23 = (0 * 10 + 2) * 10 + 3$$

$$1239 = (((0 * 10 + 1) * 10 + 2) * 10 + 3) * 10 + 9$$

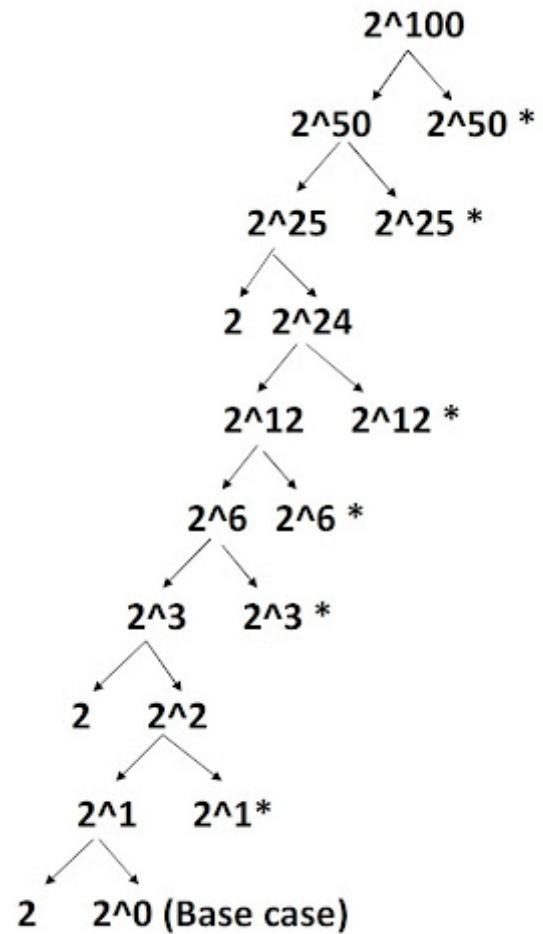
প্র্যাকটিসের জন্য প্রবলেম:

<http://uva.onlinejudge.org/external/3/374.html>

<http://uva.onlinejudge.org/external/101/10127.html>

AccessPress Staple | WordPress Theme: [AccessPress](#)

Staple by [AccessPress Themes](#)



# প্রাইম জেনারেটর (Sieve of Eratosthenes)

 shafaetsplanet.com/planetcoding/

শাফায়েত

7/17/2011

প্রাচীনকাল থেকেই গণিতবিদরা মাথা ঘামাচ্ছেন প্রাইম নাম্বার বা মৌলিক সংখ্যা নিয়ে। প্রাইম নাম্বারগুলো মধ্যে লুকিয়ে আছে বিষয়কর কিছু সৌন্দর্য। যেকোনো কম্পিউটার বা যৌগিক সংখ্যাকে একাধিক প্রাইমের গুণফল হিসাবে মাত্র একভাবে লেখা যায়, ঠিক যেমন সব যৌগিক পদার্থ একাধিক মৌলিক পদার্থের সমন্বয়ে তৈরি। প্রাচীনকাল থেকেই মানুষ প্রাইম নিয়ে গবেষণা করছে, চলছে এখনো। গাউস, ফার্মা, ইউলারের মত কিংবদন্তি গণিতবিদরা কাজ করেছেন প্রাইম নিয়ে।

দুট গতিতে প্রাইম সংখ্যা বের করার একটি পদ্ধতি আবিষ্কার করেন Eratosthenes,

২০০ খ্রিস্টপূর্বের একজন গ্রীক গণিতবিদ, বিজ্ঞানি ও কবি। ২২০০ বছরেরও পুরানো সেই

পদ্ধতি ব্যবহার করে আমরা আধুনিক কম্পিউটারে প্রাইম জেনারেট করি, খুব কম সময়ে বের করা যায় ১০কোটির নিচে সব প্রাইম সংখ্যা। এই অ্যালগোরিদমটি sieve of Eratosthenes নামে পরিচিত, প্রোগ্রামিং এর জগতে সুন্দরতম অ্যালগোরিদমগুলোর মধ্যে এটি একটি।



sieve এর শাব্দিক অর্থ হলো ছাকনি যা অপ্রয়োজনীয় অংশ ছেটে ফেলে (A sieve, or sifter, separates wanted elements from unwanted material using a woven screen such as a mesh or net)। Eratosthenes এর ছাকনি যৌগিক সংখ্যাগুলোকে ছেটে ফেলে দেয়।

*Sift the Twos and sift the Threes,  
The Sieve of Eratosthenes.  
When the multiples sublime,  
The numbers that remain are Prime  
(Traditional, collected from wikipedia)*

আমরা জানি প্রাইম সংখ্যা হলো সেসব সংখ্যা যাদের ১ এবং সেই সংখ্যাটি ব্যতিত কোনো সংখ্যা দিয়ে ভাগ করা যায়না, যেমন ২, ৩, ৫, ৭, ২৯ ইত্যাদি। অন্যভাবে বলা যায় সেসব সংখ্যাই প্রাইম যাদেরকে সংখ্যাটির বর্গমূলের সমান বা ছোটো কোনো প্রাইম দিয়ে ভাগ করা যায় না। এই সংজ্ঞাটাই অ্যালগোরিদমের মূল অংশ, তাই আগে আমরা এটা বুঝতে চেষ্টা করব। ফর্মালভাবে প্রমাণ না করে ব্যপারটি বুঝানোর চেষ্টা করি। যেকোনো সংখ্যাকে আমরা কয়েকটি প্রাইমের গুণফল হিসাবে লিখতে পারি যাদের প্রাইম ফ্যাক্টর বলা হয়:

$$n=p_1*p_2*p_3....*p_k$$

n যদি নিজেই প্রাইম হয় তাহলে  $n=p_1(n)$ । অন্যথায় অবশ্যই একাধিক প্রাইম ফ্যাক্টর থাকতে হবে। এবার চিন্তা করো কোনো সংখ্যা c কে দুটি সংখ্যার গুণফল  $c=a*b$  হিসাবে লিখলে a আর b এর একটি অবশ্যই সংখ্যাটির বর্গমূলের থেকে ছোট, অন্যটি বড়। a, b দুটো সংখ্যাই c এর বর্গমূলের থেকে বড় হলে গুণফল c থেকে বড় হতো (ঠিক যেমন  $c=a+b$  হলে a বা b এর একটি c এর অর্ধেকের থেকে ছোট অন্যটি বড়)।

এবার  $n=p_1*p_2*p_3....*p_k$  তে ফিরে আসি।  $p_1, p_2, p_3$  ইত্যাদির মধ্যে যে কোনো ২টি যদি n এর বর্গমূল থেকে বড় হয় তাহলে তাদের গুণফল n কে ছাড়িয়ে যাবে, তাই নয় কি? সর্বোচ্চ একটি প্রাইম ফ্যাক্টর বর্গমূলের বাইরে যেতে পারে, বাকি গুলো কে অবশ্যই ভিতরে থাকতে হবে।

তাহলে আমরা নিশ্চিত যে যৌগিক সংখ্যা কে তার বর্গমূলের থেকে ছোট কোনো প্রাইম দিয়ে ভাগ করা যাবে। ২য় সংজ্ঞাটি এখন আমাদের কাছে পরিষ্কার: "সেসব সংখ্যাই প্রাইম যাদেরকে সংখ্যাটির বর্গমূলের সমান বা ছোটো কোনো প্রাইম দিয়ে ভাগ করা যায় না"। বুঝতে না পারলে আরেকবার ভালো করে চিন্তা করে নিচের অংশ পড়ো।

এবার আমরা আমাদের ছাকনি ঢালু করি এবং প্রাইম বের করি। ২৫ এর নিচের সব প্রাইম আমরা বের করব। ২৫ এর বর্গমূল ৫, তাই ২৫ বা তার থেকে ছোট কোন সংখ্যাকে অবশ্যই ৫ বা তার থেকে ছোট কোনো প্রাইম দিয়ে ভাগ করা যাবে।

২ একটি প্রাইম কারণ ২কে তার বর্গমূলের নিচে কোনো সংখ্যা দিয়ে ভাগ করা যায়না। তাহলে ২ এর মাল্টিপলগুলো কেও প্রাইম নয় কারণ তাদের ২ দিয়ে ভাগ করা যায়, সেগুলোকে আমরা কেটে দেই:

| ২, ৪, ৬, ৮, ১০, ১২, ১৪, ১৬, ১৮, ২০, ২২, ২৪

২ এর পরের সংখ্যা ৩। ৩ যদি প্রাইম না হতো তাহলে ৩ এর বর্গমূলের নিচের কোনো প্রাইম ৩ কে বাদ দিয়ে দিত, যেহেতু ৩ বাদ পড়েনি তাই সংজ্ঞামতে ৩ প্রাইম। ৩ এর মাল্টিপল গুলো কে বাদ দেই:

| ৩, ৬, ৯, ১২, ১৫, ১৮, ২১, ২৪

পরের সংখ্যা ৪। ৪ বাদ পড়ে গিয়েছে আগেই। তারপর আছে ৫। ৫ যদি প্রাইম না হতো তাহলে আগেই ছাকনিতে কাটা পড়ত, ৫ এর মাল্টিপল গুলোকে কেটে দেই:

| ৫, ১০, ১৫, ২০, ২৫

আমাদের আর কাটাকাটি প্রয়োজন নেই। ২৫ এর বর্গমূল ৫, তাই ২৫এর নিচের সব সংখ্যার বর্গমূল ৫ থেকে ছোট। সুতরাং ২৫ এর নিচের সকল ঘোগিক সংখ্যা ৫ বা তার নিচের কোনো প্রাইম দিয়ে বিভাজ্য। যেহেতু আমরা ২, ৩, ৫ এর সব মাল্টিপল কেটে দিয়েছি, বাকি সংখ্যগুলো অবশ্যই প্রাইম। ছাকনির উপর থেকে সেগুলো সংগ্রহ করে নেই:

| ২, ৩, ৫, ৭, ১১, ১৩, ১৭, ১৯, ২৩

আমরা সিভের একটা কোড দেখি:

C++

```

1  bool status[1100002];
2  void siv()
3  {
4      int N=1000000;
5      int sq=sqrt(N);
6      for(int i=4;i<=N;i+=2) status[i]=1;
7      for(int i=3;i<=sq;i+=2){
8          if(status[i]==0)
9          {
10             for(int j=i*i;j<=N;j+=i) status[j]=1;
11         }
12     }
13     status[1]=1;
14 }
15

```

status অ্যারেটা দিয়ে নির্দেশ করে একটি সংখ্যা প্রাইম নাকি কম্পোজিট। status[i]=0 হলে i একটি প্রাইম। শুরুতে সব ইনডেক্সে ০ আছে, আমরা উপরের অ্যালগোরিদম অনুযায়ী নন প্রাইম সংখ্যা গুলোকে কেটে দিবো, অর্থাৎ j যদি নন-প্রাইম হয় status[j]=1 করে দিবো। ৮ নম্বর লাইনে শুরুতেই ২ এর সব মাল্টিপল কেটে দিলাম। এরপরের পরের লুপটা ৩ থেকে শুরু করে ২ করে বাড়াবো কারণ জোড় সংখ্যা নিয়ে আর চিন্তা করা দরকার নেই। ১০ নম্বর লাইনে এসে যদি status[i]=0 পাই তাহলে অ্যালগোরিদম অনুযায়ী i অবশ্যই প্রাইম কারণ। এখনও কাটা পড়েনি, এবার। এর সবগুলো মাল্টিপল কেটে দিবো, এজন্য j এর লুপ শুরু করবো  $2^{\infty}$  থেকে এবং বাড়াবো। পরিমাণ। আমাদের কাজ শেষ, নন-প্রাইম সংখ্যগুলো সব কেটে দিবে ভিতরের লুপটি, এখন status[i] এর মান দেখে আমরা i প্রাইম কিনা বের করতে পারবো।

সিভ দিয়ে প্রাইম জেনারেট করে খুব সহজে কোন সংখ্যার প্রাইম ফ্যাক্টর বা উৎপাদকে বিশ্লেষণ করা যায়। এই কাজটা তোমার হাতেই থাকলো :)।

সিভে প্রতিটি সংখ্যা প্রাইম নাকি নন-প্রাইম সেট আমরা একটি বুলিয়ান অ্যারে দিয়ে চেক করি। যত পর্যন্ত প্রাইম জেনারেট করব তত সাইজের অ্যারে লাগবে।  $10^8$  আকারের অ্যারে অনেক মেমোরি দখল করে। মেমোরি অপটিমাইজ করার জন্য অসাধারণ একটি পদ্ধতি হলো বিট ব্যবহার করা, একে bitwise সিভ বলা হয়। একটি ইন্টিজারে 32টি বিট থাকে যার প্রতিটিকে আমরা ফ্ল্যাগ হিসাবে ব্যবহার করতে পারি, সেটা নিয়ে বিস্তারিত আলোচনা পাবে [এখানে](#)।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# বিটওয়াইজ সিভ(Bitwise sieve)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

11/6/2011

বিটওয়াইজ সিভ প্রাইম সংখ্যা বের করার জন্য প্রচলিত অ্যালগোরিদম [Sieve of Eratosthenes](#) এ মেমরির ব্যবহার অনেক কমিয়ে আনা যায়। সাধারণ সিভে N পর্যন্ত প্রাইম জেনারেট করলে N সাইজের একটি অ্যারে ডিক্লেয়ার করতে হয়। অ্যারের প্রতিটি এলিমেন্ট একটি করে ফ্ল্যাগ হিসাবে কাজ করে যেটা দেখে আমরা বুঝি একটি সংখ্যা প্রাইম নাকি কম্পোজিট। বিটওয়াইজ সিভে আমরা ফ্ল্যাগ হিসাবে ইন্টিজার বা বুলিয়ান এর বদলে সরাসরি বিট ব্যবহার করবো।

এ টিউটোরিয়াল পড়ার আগে দুটি বিষয় তোমাকে জেনে আসতে হবে

১. Sieve of Eratosthenes এর সাধারণ ভার্সন, তুমি এটা আমার [এই পোস্টটি](#) পড়ে শিখতে পারবে সহজেই।

২. সি/সি++ এ বিটওয়াইজ অপারেটরের ব্যবহার। এটাও খুব সহজে শিখতে পারবে [এখান থেকে](#)। টিউটোরিয়ালটির ১ম ২টি অংশ খুব ভালো করে পড়ে ফেলো, বিটমাস্ক ডিপি, বিএফএস যখন শিখবে তখন অনেক কাজে লাগবে।

আশা করি এখন তুমি বিটওয়াইজ অপারেটর সম্পর্কে অনেক কিছু জানো, সাধারণ সিভ লিখতে কোনো সমস্যা হয়না তোমার। এবার আমরা শিখবো বিটওয়াইজ সিভ।

সাধারণ সিভে status বা flag অ্যারেটার কাজ কি? এই অ্যারের ইনডেক্সের মান দেখে আমরা বলতে পারি একটি সংখ্যা প্রাইম কিনা। ধরলাম তোমার status অ্যারেটা ইন্টিজার টাইপের। প্রতিটি ইন্টিজারের মধ্য আছে ৩২টি বিট। আমরা কেনে এতগুলো বিট ব্যবহার করবো খালি ০ বা ১ নির্দেশ করতে? আমরা অ্যারের যেকোনো ইনডেক্সের প্রতিটি বিট দিয়ে একটি সংখ্যা নির্দেশ করতে পারি।

তুমি যখন ইন্টিজার অ্যারেতে ১-৭ পর্যন্ত প্রাইম জেনারেট কর, তোমার অ্যারের অবস্থা বাইনারিতে থাকে এরকম:

C++

```

1 status[১]=০০০.....০০(৩২টি শূন্য)
2 status[২]=০০০.....০১(৩১টি শূন্য, ১টি ১)
3 status[৩]=০০০.....০১(৩১টি শূন্য, ১টি ১)
4 status[৪]=০০০.....০০(৩২টি শূন্য)
5 status[৫]=০০০.....০১(৩১টি শূন্য, ১টি ১)
6 status[৬]=০০০.....০০(৩২টি শূন্য)
7 status[৭]=০০০.....০১(৩১টি শূন্য, ১টি ১)
```

প্রতিটি ইনডেক্সে ৩১টি বিট কোনো কাজে লাগছেনা অথচ এই বিশাল সংখ্যক অব্যবহৃত বিট আমরা সহজেই কাজে লাগাতে পারি। আমরা ধরে নিবো:

C++

```

1 status[0] এর
2     >>> শূন্যতম বিট ০ এর প্রাইমালিটি নির্দেশ করে (সবথেকে ডানের বিট==০ তম বিট)
3     >>> ১ম বিট ১ এর প্রাইমালিটি নির্দেশ করে (সবথেকে ডানের বিট==০ তম বিট)
4     >>> ২য় বিট ২ এর প্রাইমালিটি নির্দেশ করে
5     >>> ৩য় বিট ৩ এর প্রাইমালিটি নির্দেশ করে
6     .....
7     >>> ৩১তম বিট ৩১ এর প্রাইমালিটি নির্দেশ করে
8
9 status[1] এর
10    >>> শূন্যতম বিট ৩২ এর প্রাইমালিটি নির্দেশ করে
11    >>> ১ম বিট ৩৩ এর প্রাইমালিটি নির্দেশ করে
12    .....
13
14 status[2] এর
15    >>> শূন্যতম বিট ৬৪ এর প্রাইমালিটি নির্দেশ করে
16    >>> ১ম বিট ৬৫ এর প্রাইমালিটি নির্দেশ করে

```

তাহলে ১-৭ পর্যন্ত প্রাইম জেনারেট করলে তোমার অ্যারের অবস্থা দাঢ়াবে:

| status[1]= 0000.....10101100(মোট ৩২টি বিট) (সবথেকে ডানের বিট==০ তম বিট)

৭টি সংখ্যার কাজ একটি ইনডেক্সেই শেষ!!। শুধু ৭টি নয়, আসলে ৩১টি সংখ্যার কাজ শেষ হবে ১টি ইনডেক্স(কারণ প্রতি ইনডেক্সের ৩১টি বিট ব্যবহার করছি আমরা)। এখন প্রশ্ন হলো কোনো সংখ্যার প্রাইমালিটি কত নম্বর ইনডেক্সের কত নম্বর বিট দিয়ে নির্দেশ করা হবে? খুব সহজ, সংখ্যাটি i হলে i/৩২ নম্বর ইনডেক্সের i%৩২ নম্বর বিট আমাদের চেক করতে হবে। তাহলে i=১ হলে চেক করবো ০ নম্বর ইনডেক্সের ১ নম্বর বিট, i=৩৩ হলে চেক করবো ১ নম্বর ইনডেক্সের ১ নম্বর বিট ইত্যাদি। (শুন্য বেসড ইনডেক্সিং)

কোডিং অংশ একদম সহজ। তুমি যেহেতু বিটওয়াজ অপারেটরের ব্যবহার জানো, কোনো সংখ্যার pos তম বিটে ১ বা ০ আছে নাকি সহজেই চেক করতে পারবে। pos তম বিটে নিজের ইচ্ছামত ১ বা ০ বসাতেই পারবে। আমাদের এখানে ০ বসানো দরকার নেই, ১ বসাতে পারলেই চলবে। দুটি ফাংশন লিখে ফেলি:

C++

```

1 bool Check(int N,int pos){return (bool)(N & (1<<pos));}
2 int Set(int N,int pos){ return N=N | (1<<pos);}

```

এবার সিভ লিখে ফেলি:

C++

```

1 int N =100,prime[100];
2 int status[100/32];
3 void sieve()
4 {
5     int i, j, sqrtN;
6     sqrtN = int( sqrt( N ) );
7     for( i = 3; i <= sqrtN; i += 2 )
8     {
9         if( check(status[i/32],i%32)==0)
10    {
11        for( j = i*i; j <= N; j += 2*i )
12        {
13            status[j/32]=Set(status[j/32],j % 32) ;
14        }
15    }
16 }
17 puts("2");
18 for(i=3;i<=N;i+=2)
19 if( check(status[i/32],i%32)==0)
20 printf("%d\n",i);
21 }
22

```

লক্ষ্য করো,আমরা সাধারণ সিভের মত করেই সব লিখেছি,তবে `status[i]` এর মান চেক করার বদলে `status[i/32]` এর `i%32` নম্বর বিটের মান চেক করেছি।

বিটওয়াইজ সিভ ব্যবহার করে  $10^{18}$  পর্যন্ত প্রাইম তুমি জেনারেট করে ফেলতে পারবে। সাধারণ সিভের থেকে সময় + মেমরি কম লাগবে। সাধারণ গুণ,ভাগ অপারেশনের থেকে বিটের অপারেশনগুলো দ্রুত কাজ করে। আমরা আরো কিছু অপটিমাইজেশন করতে পারি। যেমন তুমি উপরে দেয়া [টিউটোরিয়াল](#) পড়ে থাকলে জানো যে কাওকে ২ দিয়ে গুণ করা আর সংখ্যাটির বাইনারিকে ১ ঘর বামে শিফট করা একই কথা। আবার ২ দিয়ে ভাগ করা আর ১ ঘর ডানে শিফট করা একই কথা, ৩২ দিয়ে mod করা আর ৩১ দিয়ে AND করা একই কথা। তাহলে আমরা নিচের মত করে কোডটি লিখতে পারি:

C++

```

1 int status[(mx/32)+2];
2 void sieve()
3 {
4     int i, j, sqrtN;
5     sqrtN = int( sqrt( N ) );
6     for( i = 3; i <= sqrtN; i += 2 )
7     {
8         if( Check(status[i>>5],i&31)==0)
9         {
10            for( j = i*i; j <= N; j += (i<<1) )
11            {
12                status[j>>5]=Set(status[j>>5],j & 31) ;
13            }
14        }
15    }
16    puts("2");
17    for(i=3;i<=N;i+=2)
18    if( Check(status[i>>5],i&31)==0)
19    printf("%d\n",i);
20 }
21

```

ফাংশন ব্যবহার না করে ম্যাক্রো ব্যবহার করলে আরো কম সময় লাগবে। **প্রোগ্রামিং কনটেস্ট** খুব কমই বিটওয়াইজ সিভ ব্যবহার করা দরকার হয়, সাধারণ সিভেই কাজ চলে। তারপরেও এটা শিখলে বিটের কনসেপ্ট গুলো কিছুটা পরিষ্কার হবে, অন্য কোনো প্রবলেমে হয়তো মেমরি কমিয়ে ফেলতে পারবে। ডাইনামিক প্রোগ্রামিং এ আমরা প্রায়ই বিটমাস্কের ব্যবহার করি মূলত মেমরি কমানোর জন্য।

তোমার ইম্প্লিমেন্টেশন সঠিক নাকি চেক করতে নিচের সমস্যাটি সমাধান করে ফেলো:

<http://www.spoj.pl/problems/TDPRIMES/>

(নোট: অনেকের ভুল ধারণা আছে যে bool টাইপের ভ্যারিয়েবলের আকার ১বিট। আসলে bool এর আকার ৮বিট বা এক বাইট, char এর সমান। এর কারণ হলো কম্পিউটার ১ বাইটের ছোটো মেমরি সেগমেন্টকে অ্যাড্রেস করতে পারেনা, তাই ভ্যারিয়েবলের নৃন্যতম আকার ১ বাইট)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by [AccessPress Themes](#)

# কম্পিউটেরিঙ্ক: অ্যারেঞ্জমেন্ট এবং ডি-রেঞ্জমেন্ট গণনা

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

5/9/2013

কনটেস্ট প্রোগ্রামিং এর একটা দারুণ ব্যাপার হলো কনটেস্টেন্টদের শুধু ভালো প্রোগ্রামিং জ্ঞানলেই হয়না, সাথে ভালো গণিতও জ্ঞান দরকার হয়। বিশেষ করে কম্পিউটেরিঙ্ক আর প্রোগ্রামিং ভাষার ধারণা থাকলে অনেক ধরণের প্রবলেম সল্ভ করে ফেলা যায়।

৪টি টুপি পাশাপাশি সাজানো আছে, টুপিগুলোকে যথাক্রমে ১, ২, ৩, ৪ সংখ্যাগুলো দিয়ে চিহ্ন দেয়া হয়েছে। এখন টুপিগুলোকে এলোমেলো করে কতভাবে সাজানো যাবে? আমরা কয়েকভাবে সাজিয়ে চেষ্টা করি:

১, ২, ৩, ৪  
১, ৩, ২, ৪  
১, ৪, ২, ৩  
১, ৩, ৪, ২  
.....  
.....  
৪, ৩, ২, ১

মোট কতভাবে সাজানো যাবে? কলেজে করে আসা অংক থেকে তুমি সহজেই বলতে পারবে  
 $\text{factorial}(8)=28$   
 $\text{factorial}(8)=28$  ভাবে সাজানো যায়। এটাকে আমরা একটু প্রোগ্রামারের দৃষ্টিভঙ্গী থেকে দেখি। ৪টা জায়গা বা স্লট আছে, প্রতিটি স্লটে ১টি করে টুপি বসানো যায়। এখন প্রথম স্লটে ১, ২, ৩ বা ৪ এর যেকোনো একটা বসালে:

১, \_\_, \_\_, \_\_

প্রথম স্লটে টুপি কত ভাবে বসানো যায়? অবশ্যই ৪ ভাবে। এখন ২য় স্লটে কয়ভাবে বসানো যায়? একটা টুপি আমরা বসিয়ে ফেলেছি আগেরটায়, তাই ২য় স্লটে বসাতে পারবো  $4-1=3$  ভাবে। ঠিক এভাবে ৩য় স্লটে ২ভাবে এবং ২য় স্লটে ১ ভাবে। তাহলে মোট উপায়  $4 \times 3 \times 2 \times 1 = 24$  টা। ৪টার জায়গায়  $n$  টা টুপি থাকলে কি করতে? আমরা প্রোগ্রামার তাই বারবার কষ্ট করে হিসাব না করে ধূম করে একটা ফাংশন লিখে ফেলি। মনে করো ফাংশনটা হলো  $\text{permutation}(n)$ ।  $n=0$  হলে সাজানো যায় ১ ভাবে, তাহলে:

$\text{permutation}(0)=0$

$n>0$  হলে প্রথম স্লটে বসানো যায়  $n$  ভাবে, এরপরে সমস্যাটা ছোটো হয়ে দাঢ়ায় “ $n-1$  টা টুপি  $n-1$  টা স্লটে কতভাবে বসানো যায়?” অর্থাৎ সমস্যাটা  $\text{permutation}(n-1) \times \text{permutation}(n-1)$  হয়ে যায়। সাথে গুণ হবে  $n$  কারণ কারেন্ট স্লটে  $n$  ভাবে বসিয়েছি। তাহলে লিখতে পারি:

$\text{permutation}(n)=n \times \text{permutation}(n-1)$

আশা করি ব্যাপারটা পরিষ্কার। সহজ ব্যাপারটা নিয়ে এত কথা বললাম যাতে রিকার্শনটা পরিষ্কার হয় যেটা কাজে লাগবে ডি঱েঞ্জমেন্ট গোণার জন্য।

এখন ধরো ১, ২, ৩, ৪ এই ৪টা টুপির মালিক হলো যথাক্রমে সাকিব, নাসির, তামিম, রহিম। তারা খুবই ভালো বন্ধু বলে ঠিক করলো একজন আরেকজনের টুপি পড়ে ক্রিকেট খেলতে যাবে। কেও তার নিজের টুপি পড়তে পারবেনা, তাহলে বন্ধুত্ব থাকবেনা! এখন কতভাবে তারা টুপি পড়তে পারবে?

গণিতের ভাষায় এর নাম ডিরেঞ্জমেন্ট, এমন কয়টি পারমুটেশন আছে যেখানে কেও তার নিজের জায়গায় নেই।

১,৩,২,৪ ডি-রেঞ্জমেন্ট নয় কারন সাকিব আর রহিম তাদের নিজ নিজ টুপিই পড়ে আছে(১ ও ৪ নম্বর) ! ২,১,৪,৩ একটি ডি-রেঞ্জমেন্ট, সবাই তার বন্ধুর টুপি পড়েছে।

আমরা একটা ফাংশন বানাবো  $d(n)$  যেটা  $nn$  টা টুপি কতভাবে সাজানো যায় যাতে কেও তার নিজের টুপি না পায় সেটা বের করে দেয়।

প্রথম মানুষ সাকিবের কাছে ৪-১=৩টা চয়েস আছে, সে ১ নম্বর বাদে যেকোনো টুপি নিতে পারে। মনে করলাম সে তামিমের টুপি নিলো। এখন ২টা ঘটনা ঘটতে পারে:

১. পরের বার তামিম নিলো সাকিবের টুপি। এখন ৪-২=২ জন মানুষ বাকি, টুপিও বাকি ঠিক ৪-২=২ টা।

২. পরের বার তামিম সাকিব ছাড়া অন্য কারো টুপি নিলো। এখন মানুষ বাকি ৪-১=৩ জন। তামিম যেহেতু সাকিবের টুপি নিছেনা তাই ওটাকেই তার নিষিদ্ধ টুপি ধরতে হবে, আর বাকি সবার কাছে নিষিদ্ধ টুপি হলো তার নিজের টুপিটা। তাহলে এখন ৪-১=৩ জন মানুষের জন্য ৪-১=৩ টা করে চয়েস আছে। লক্ষ্য করো

[Shafaetsplanet.com/blog](http://Shafaetsplanet.com/blog)



Sakib

Nasir

Tamim

Rahim



Sakib

Nasir

Tamim

Rahim

সাকিব আর তামিম একজন আবেকজনের টপি নিলো। এখন ৪-২=২ জন মানুষের জন্য সমস্যাটি সমাধান করতে হবে।



Sakib

Nasir

Tamim

Rahim

সাকিব তামিমের টুপি নিয়েছে। কিন্তু তামিম সাকিবের টুপি নিবেনা, তাহলে ৪-১=৩টা মানুষের জন্য সমস্যাটি সমাধান করতে হবে।

দুই ক্ষেত্রেই মানুষ আর টুপির সংখ্যা সমান থাকচে। ৪ এর জায়গায়  $n$  ধরে ২টা কল্পনা মিলিয়ে সহজেই রিকার্সিভ রিলেশনটা লিখতে পারি:

$$d(n)=(n-1)*(d(n-1)+d(n-2))$$

$$\text{বেস কেস: } d(1)=0, d(2)=1$$

$$d(1)=0, d(2)=1$$

এই রিকার্সনটা কোড করার সময় মাথায় রাখতে হবে যে একই ফাংশন অনেকবার কল হচ্ছে, তাই ডিপি টেবিলে মানগুলো সেভ করে রাখতে হবে। তুমি ডাইনামিক প্রোগ্রামিং নিয়ে পড়ালেখা করতে পারো এ সম্পর্কে জানতে।

এবার আরেকটা মজার উপায়ে প্রবলেমটা সলভ করি।  $nCr_nCr$  বা  $(nr)(nr)$  এর সাথে তোমরা পরিচিত,  $n^n$  টা জিনিস থেকে  $n^n$  টি জিনিস করত্বাবে নেয়া যায় সেটাই প্রকাশ করে  $(nr)(nr)$ ।  $n^n$  টা টুপিকে মোট সাজানো যায়  $n^n!$  উপায়। এর মধ্যে যেসব পারমুটেশনে অন্তত একটি টুপি নিজের জায়গায় আছে তাদের বাদ দিলে ডিরেঞ্জমেন্ট পাওয়া যায়।  $n^n$  টি টুপি থেকে ১ টি টুপি নেয়া যায়  $(n1)(n1)$  উপায়ে, ১টি টুপিকে নিজের জায়গায় রেখে বাকি  $n-1$  টা টুপিকে সাজানো যায়  $(n-1)!(n-1)!$  উপায়ে। তাহলে  $n!-(n1) \times (n-1)!n!-(n1) \times (n-1)!$  বের করলেই ডিরেঞ্জমেন্ট বের হয়ে যাচ্ছেনা? কারণ আমরা মোট উপায় থেকে যেসব পারমুটেশনকে অন্তত ১ জন নিজের জায়গায় আছে তাদের বাদ দিচ্ছি।  $(n1)(n1)$  দিয়ে গুণ দিচ্ছি কারণ প্রতিবার ১জন কে ফিল্ড করে  $n-1n-1$  জনকে পারমুটেশন করতেসি।

কিন্তু এখানে একটা বড় সমস্যা আছে। ধরো তুমি তামিমের টুপিকে তামিমের কাছেই রেখে বাকি টুপিগুলো কয়ত্বাবে সাজানো যায় বের করলে। আবার নতুন করে সাকিবেরটা সাকিবের কাছে রেখে বাকিগুলো কয়ত্বাবে সাজানো যায় বের করলে। ভালোমত চিন্তা করে দেখ যেসব পারমুটেশনে সাকিবেরটা সাকিবের কাছে আছে আর তামিমেরটা তামিমের কাছে আছে সেগুলো কি ২বার গণনা করা হয়ে গেলো না?  $(n1) \times (n-1)!(n1) \times (n-1)!$  এ এই কারণে কিছু পারমুটেশন একাধিক বার ক্যালকুলেট করা হয়ে যাবে। সেগুলো আমরা কিভাবে বাদ দিবো? আমরা ১টা সংখ্যা ফিল্ড করে যখন গুনেছি তখন যেসব পারমুটেশনে ২টা সংখ্যা ফিল্ড সেগুলো একাধিক বার গুণে ফেলেছি, সেগুলো আমরা বাদ দিয়ে দেই।  $(n1) \times (n-1)!(n1) \times (n-1)!$  থেকে বাদ দিয়ে দিবো  $(n2) \times (n-2)!(n2) \times (n-2)!$ । একটু চিন্তা করলে বুঝতে পারবে এখানেও সমস্যা আছে, যেখানে ৩টা ফিল্ড সেগুলোকেও আমরা বাদ দিয়ে দিচ্ছি!! তাহলে সেটা আবার যোগ করে দাও। মাথা গুলিয়ে গেলে ভ্যান ডায়াগ্রামের কথা চিন্তা করো:



ভ্যান ডায়াগ্রামে ৩টা অংশের কমন এরিয়া বের করতে আমরা সবগুলো অংশ যোগ করি, তারপর যেসব অংশ দুটি বুত্তে আছে সেগুলো বাদ দেই, যেগুলো ৩টি বৃত্তে আছে সেগুলো আবার যোগ করে দেই, বৃত্ত আরো বেশি থাকলে এভাবে যোগ বিয়োগ চলতেই থাকে। দুটি সেট  $A, BA, B$  হলে  $|A \cup B| = |A| + |B| - |A \cap B|$ ।  $|A \cup B| = |A| + |B| - |A \cap B|$ । ঠিক এই কাজটি করবো এখানে। আমাদের ফর্মুলা হবে:

$$n!-(n1) \times (n-1)!+(n2) \times (n-2)!-\dots+(-1)^k \times (nk) \times (n-k)!+\dots+(-1)^{nn}-(n1) \times (n-1)!+(n2) \times (n-2)!-\dots+(-1)^k \times (nk) \times (n-k)!+\dots+(-1)^n$$

আমরা একবার যোগ করছি, একবার বিয়োগ করছি, এভাবে অপ্রয়োজনীয় অংশ বাদ দিয়ে ফলাফল পেয়ে যাচ্ছি। এ জিনিসটারই রাশভারী নাম হলো ইনকুলশন-এক্সকুলশন প্রিস্নিপাল।

আজ এ পর্যন্তই। সলভ করার জন্য প্রবলেম:

[www.topcoder.com/stat?c=problem\\_statement&pm=2013](http://www.topcoder.com/stat?c=problem_statement&pm=2013)

<http://uva.onlinejudge.org/external/112/11282.html>

[http://www.lightoj.com/volume\\_showproblem.php?problem=1095](http://www.lightoj.com/volume_showproblem.php?problem=1095)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by [AccessPress Themes](#)

# HANDBOOK OF ALGORITHMS

## Section Data Structure

*Courtesy of*  
*Shafaet Ashraf*

ডাটা স্ট্রাকচার \_ লিংকড লিস্ট \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার \_ স্ট্যাক \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ কডি এবং সার্কুলার কডি \_ শাফায়তেরে ব্লগ.pdf

স্লাইডিং রেণ্জ মনিমাম কুয়রো \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ ডিজিয়ান্টে সটে(ইউনিয়ন ফাইন্ড) \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ ট্রাই (প্রফিক্স ট্রি\_রডেক্স ট্রি) \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ সগেমনেট ট্রি-১ \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ সগেমনেট ট্রি-২ (লজে প্রপাগশেন) \_ শাফায়তেরে ব্লগ.pdf

অ্যারে কম্প্যুশন \_ শাফায়তেরে ব্লগ.pdf

লয়েস্টে কমন অ্যানসেস্ট্রি \_ শাফায়তেরে ব্লগ.pdf

ডাটা স্ট্রাকচার\_ বাইনারি ইনডকেস্ড ট্রি \_ শাফায়তেরে ব্লগ.pdf

# ডাটা স্ট্রাকচার : লিংকড লিস্ট

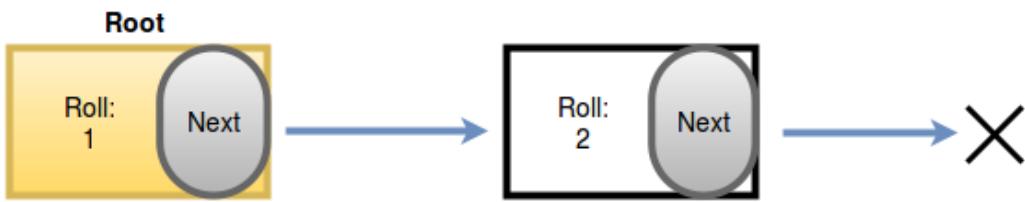
 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

জানুয়ারি ৩০, ২০১৬

লিংকড লিস্ট বেসিক একটা **ডাটা স্ট্রাকচার**। আমরা সাধারণত তথ্য রাখার জন্য অ্যারে ব্যবহার করি, তবে অ্যারের কিছু সীমাবদ্ধতা আছে যে কারণে অনেক সময় লিংকড লিস্ট ব্যবহারের দরকার হয়। লিংকড লিস্ট নিয়ে জানতে হলে অবশ্যই পয়েন্টার সম্পর্কে ধারণা থাকতে হবে।

লিংক লিস্টের প্রতিটা এলিমেন্ট কে বলবো আমরা নোড। প্রতিটা নোডে সাধারণত দুইটা তথ্য থাকে: ১) যে তথ্যটা আমরা সংরক্ষণ করতে চাচ্ছি ২) পরবর্তি তথ্যটা কোথায় আছে তার ঠিকানা।



ছবিতে দেখা যাচ্ছে প্রথম নোড এ একজন ছাত্রের রোল নম্বর লেখা আছে, এবং পরবর্তি ছাত্রের তথ্য কোন নোড এ আছে সেটা দেখিয়ে দিচ্ছে next নামের একটা পয়েন্টার। দ্বিতীয় নোডটাই শেষ নোড, তাই এই নোডের নেক্সট পয়েন্টার একটা null নোডকে পয়েন্ট করছে। প্রথম নোডকে আমরা বলবো রুট নোড।

অ্যারের সাথে লিংক লিস্টের একটা বড় পার্থক্য হলো অ্যারের তথ্যগুলো মেমরিতে পরপর সংরক্ষণ করা হয়। যদি অ্যারেটা একটা 44 বাইটের ইন্টিজার অ্যারে হয় এবং অ্যারের প্রথম এলিমেন্টটা যদি থাকে xx তম মেমরি সেল এ, তাহলে পরের ৩টি এলিমেন্ট x+4,x+8,x+12x+4,x+8,x+12 মেমরি সেল এ থাকবে। নিচের কোডটা রান করলেই প্রমাণ পাবে।

```

1 int main(){
2     int a[5];
3     for(int i=0;i<5;i++)
4     {
5         printf("%u\n",&(a[i])); #print address of each element
6     }
7     return 0;
8 }
  
```

সেজন্য অ্যারের প্রথম এলিমেন্টের অ্যাড্রেস জানলেই এরপর যেকোনো এলিমেন্টের অ্যাড্রেস সহজেই বের করে ফেলা যায়, ইন্টিজার অ্যারের pp তম এলিমেন্ট থাকে x+p\*4x+p\*4 অ্যাড্রেসে যেখানে xx হলো শূন্যতম এলিমেন্টের অ্যাড্রেস।

লিংকড লিস্টে তথ্যগুলো থাকে ছড়িয়ে-ছিটিয়ে, তাই প্রতিটা এলিমেন্টকে পরের এলিমেন্টের ঠিকানা সংরক্ষণ করে রাখতে হয়। এই পদ্ধতির কিছু সুবিধাও আছে, অসুবিধাও আছে, সেগুলো আমরা দেখবো।

একটা সি তে লিংকড লিস্ট তৈরির জন্য শুরুতেই একটা স্ট্রাকচার ডিফাইন করতে হবে, যেখানে থাকবে যে তথ্য সংরক্ষণ করতে চাই সেটা এবং পরবর্তী নোডের অ্যাড্রেস।

```

1  nstruct node
2  {
3      int roll;
4      node *next;
5  };
6  node *root=NULL;
7  int main(){
8      return 0;
9  }
10

```

node \*next হলো একটা পয়েন্টার যেটা একটা node এর অ্যাড্রেস সংরক্ষণ করবে।

node \*root হলো একটা পয়েন্টার যেটা সবসময় প্রথম নোডের অ্যাড্রেস সংরক্ষণ করবে। শুরুতে লিস্ট এ কোনো নোড নেই, তাই রুট পয়েন্টারের মান নাল(Null)। প্রথম নোডের অ্যাড্রেস ব্যবহার করে আমরা পরবর্তীতে অন্য নোডের তথ্য পড়তে পারবো।

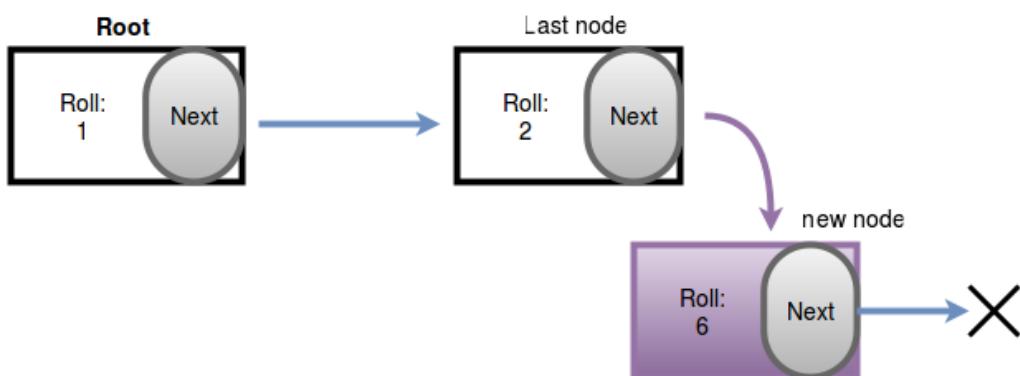
এখন আমাদের একটা ফাংশন দরকার যেটা ব্যবহার করে নতুন একটা নোড লিস্টে শেষে প্রবেশ করাতে পারবো। মনে করো ফাংশনটা রান্ডেম নাম append। এই ফাংশনটা লেখার সময় ইটা কেস মাথায় রাখতে হবে। প্রথম কেস হলো যে নোডটা প্রবেশ করাচ্ছি সেটাই লিঙ্ক লিস্টের প্রথম নোড কি না। যদি তাই হয়, তাহলে রুট পয়েন্টার ব্যবহার করে প্রথম নোডটা তৈরি করতে হবে।

```

1  void append(int roll)
2  {
3      if(root==NULL) //If the list is empty
4      {
5          root=new node(); //create new node in root
6          root->roll=roll;
7          root->next=NULL;
8      }
9  }

```

যদি লিঙ্কড লিস্টে আগেই কিছু নোড থাকে তাহলে আমাদেরকে শেষ নোডটা খুজে বের করতে হবে। তারপর শেষ নোডের নেক্সট পয়েন্টার ব্যবহার করে পরবর্তী নোডটা তৈরি করতে হবে।



```

1 void append(int roll)
2 {
3     if(root==NULL) //If the list is empty
4     {
5         root=new node();
6         root->roll=roll;
7         root->next=NULL;
8     }
9     else
10    {
11        node *current_node=root; //make a copy of root node
12        while(current_node->next!=NULL) //Find the last node
13        {
14            current_node=current_node->next; //go to next address
15        }
16        node *newnode = new node(); //create a new node
17        newnode->roll=roll;
18        newnode->next=NULL;
19        current_node->next=newnode; //link the last node with new node
20    }
21 }
22

```

আমরা প্রথমে লুপ চালিয়ে শেষ নোডটা বের করছি। শেষ নোড কোনটা বোঝা খুব সহজ, যেই নোডের নেক্সট পয়েন্টার নাল সেটাই শেষ নোড। এরপর নতুন একটা নোড তৈরি করে শেষ নোডের সাথে সেটা লিংক করে দিচ্ছি। আমাদের এই অ্যাপেন্ড ফাংশনের **কমপ্লেক্সিটি O(n)O(n)**।

লক্ষ্য করো, রুট পয়েন্টারকে আমরা সামনে নিচ্ছি না, সেটার একটা কপি তৈরি সেটাকে সামনে নিচ্ছি। কারণ রুট পয়েন্টারকে আমরা সামনে নিলে প্রথম নোডের অ্যাড্রেস হারিয়ে ফেলবো!

সবগুলো ছাত্রের রোল নম্বর প্রিন্ট করতে চাইলেও একইভাবে করতে পারবো। আগের মতই লুপ চালিয়ে শেষ নোড পর্যন্ত যাবো এবং সবগুলো মান প্রিন্ট করবো।

```

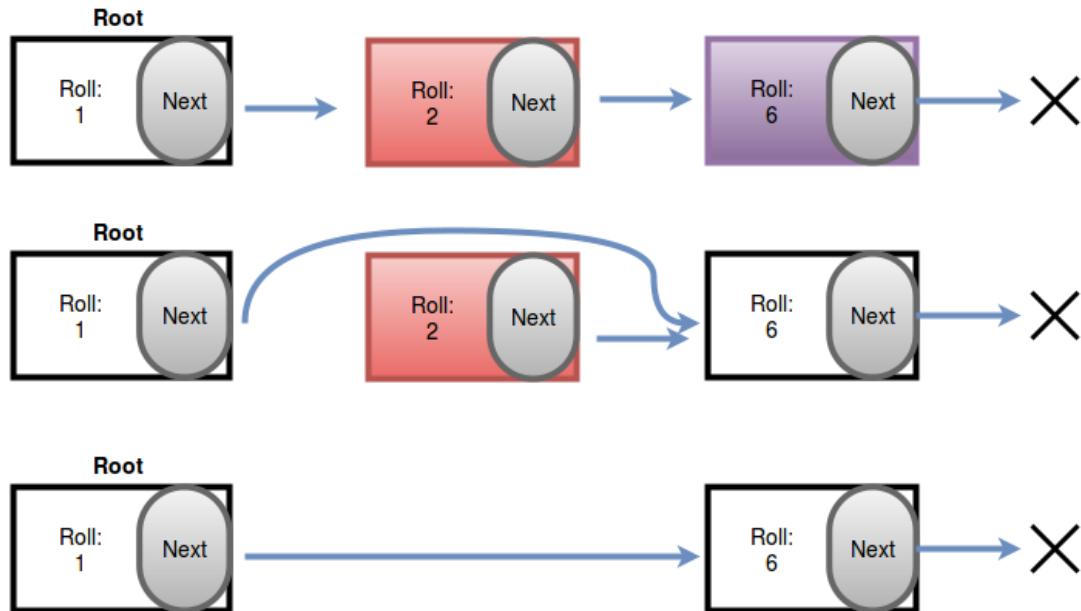
1 void print()
2 {
3     node *current_node=root;
4     while(current_node!=NULL) //loop until you reach null
5     {
6         printf("%d\n",current_node->roll);
7         current_node=current_node->next;
8     }
9 }
10 int main(){
11
12     append(1);
13     append(2);
14     append(6);
15     print();
16     return 0;
17 }

```

এখন তুমি যদি চাও শুধুমাত্র 10 তম ছাত্রের রোল প্রিন্ট করতে, তাহলে কি করবে? তোমাকে লুপ চালিয়ে 10 নম্বর নোড খুজে বের করে প্রিন্ট করতে হবে। কিন্তু অ্যারেতে আমরা roll[10]roll[10] লিখেই 10তম ছাত্রের রোল প্রিন্ট করে ফেলতে পারতাম। লিংকড লিস্টে তথ্যগুলো মেমরিতে পরপর সাজান্য নেই তাই রেন্ডম এক্সেস করা যায় না। লিংকড লিস্টে কোনো

ইনডেক্স খুজে বের করার কমপ্লেক্সিটি তাই  $O(n)O(n)$ , যেখানে অ্যারেতে  $O(1)O(1)$ । [পুরানো আমলের গানশোনার ফিতার ক্যাসেটগুলোর কথা মনে আছে? সেখানেও কোনো গানে লাফ দিয়ে চলে যাওয়া যেত না, ফিতা ঘুরিয়ে খুজে বের করতে হতো। এখানেও একই ব্যাপার ঘটছে!]

লিংক লিস্ট এর সুবিধা হলো চাইলেও কোনো তথ্য মাঝখান থেকে মুছে ফেলা যায়। অ্যারেতে তুমি চাইলেই মাঝখান থেকে একটা ইনডেক্স মুছে ফেলতে পারবে না, মুছতে হলে ডানের সব এলিমেন্টকে একঘর বামে টেনে এনে ফাকা জায়গা পূরণ করতে হবে, এবং সবার শেষের এলিমেন্টটাকে মুছে ফেলতে হবে। কিন্তু লিংকড লিস্টে তুমি সহজেই মাঝখান থেকে একটা নোড মুছে ফেলতে পারবে।



ছবিতে রোল ২ কে কিভাবে মুছে ফেলা যায় দেখানো হয়েছে। রোল ২ এর আগের নোড রোল ১ এর পয়েন্টারকে দিয়ে রোল ২ এর পরের নোড এর অ্যাড্রেস কে পয়েন্ট করানো হয়েছে, এবং মাঝের নোডটা মেমরি থেকে মুছে ফেলা হয়েছে।

লক্ষ্য করো, রুট নোডের আগে কোনো নোড নেই। তাই রুট নোড মুছে ফেলা আরো সহজ, শুধুমাত্র রুট পয়েন্টার এক ঘর এগিয়ে দিতে হবে এবং আগের নোডটা মুছে ফেলতে হবে।

```

1 void delete_node(int roll)
2 {
3     node *current_node=root;
4     node *previous_node=NULL;
5     while(current_node->roll!=roll) //Searching node
6     {
7         previous_node=current_node; //Save the previous node
8         current_node=current_node->next;
9     }
10    if(current_node==root) //Delete root
11    {
12        node *temp=root; //save root in temporary variable
13        root=root->next; //move root forward
14        delete(temp); //free memory
15    }
16    else //delete non-root node
17    {
18        previous_node->next=current_node->next; //previous node points the current node's next node
19        delete(current_node); //free current node
20    }
21 }
22 
```

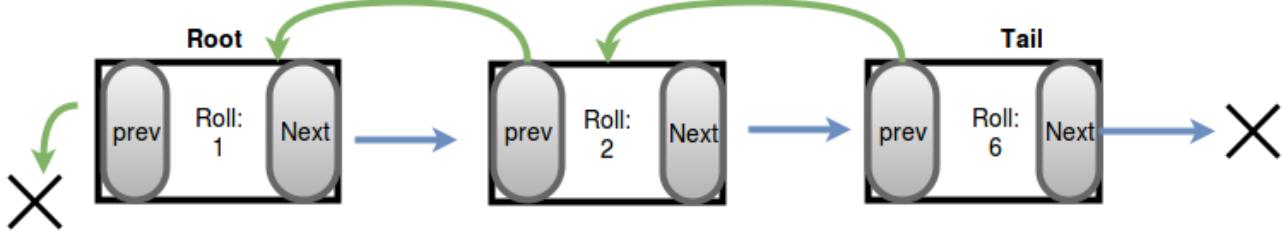
উপরের কোডে প্রথমে আমরা খুজে বের করেছি যে রোল নশ্বরটা মুছতে হবে সেই নোডটাকে। যদি সেটাই রুট নোড হয় তাহলে রুটকে একঘর এগিয়ে দিয়েছি, নাহলে উপরের ছবির মত করে মুছেছি।

লক্ষ্য করো, আমি `delete(node)` নামের একটা লাইব্রেরি ফাংশন ব্যবহার করেছি। মোছার সময় পয়েন্টার ঠিকঠাক করার পর অবশ্যই `delete` ফাংশন ব্যবহার করে মেমরি ফ্রি করে দিতে হবে, নাহলে লিংকড লিস্ট থেকে নোড মুছে গেলেও নোডটা মেমরিতে থেকে যাবে, অন্য কোনো প্রোগ্রাম সেটাকে ব্যবহার করতে পারবে না। লিংকড লিস্টের কোড লেখার সময় `delete()` ফাংশন ব্যবহার করতে ভুলে যাওয়া খুবই কমন একটা ভুল।

লিংকড লিস্ট এ তুমি চাইলে মাঝখানেও নোড যোগ করতে পারবে। এই পর্যন্ত বুঝে থাকলে তোমার কাজ হবে দুই নোড এর মাঝে নতুন নোড যোগ করার জন্য ফাংশন লেখা। এটা অনেকটা `delete-node` ফাংশনের মত করে লিখতে হবে। ফাংশনের প্যারামিটার হিসাবে নিবে `roll1`, `roll2`, তোমার ফাংশনের কাজ হবে `roll1` যে নোডে আছে সেটা খুজে বের করে সেটার পরে `roll2` নোডটা যোগ করা।

## বাইডিরেকশনাল লিংকড লিস্ট

আমাদের আগের কোড এ অ্যাপেন্ড অপারেশন ছিলো  $O(n)$ , লুপ চালিয়ে বারবার শেষ পর্যন্ত যেতে হচ্ছিলো। এছাড়া লিংকড লিস্টটা উল্টো দিক থেকে ট্রাভার্স করা সম্ভব হচ্ছিলো না।



উপরের ছবির লিংক লিস্টে প্রতি নোডে দুইটা পয়েন্টার ব্যবহার করা হয়েছে। `prev` পয়েন্টারটা প্রতিটা নোডের আগের নোডের অ্যাড্রেসকে পয়েন্ট করে আছে। এছাড়াও এআমরা কটা `tail` পয়েন্টার এর সাহায্যে শেষ নোডটার অ্যাড্রেস মনে রাখছি।

এখন লিংকড লিস্টের শেষে নতুন নোড যোগ করা সম্ভব  $O(1)$  কমপ্লেক্সিটিতে।

```

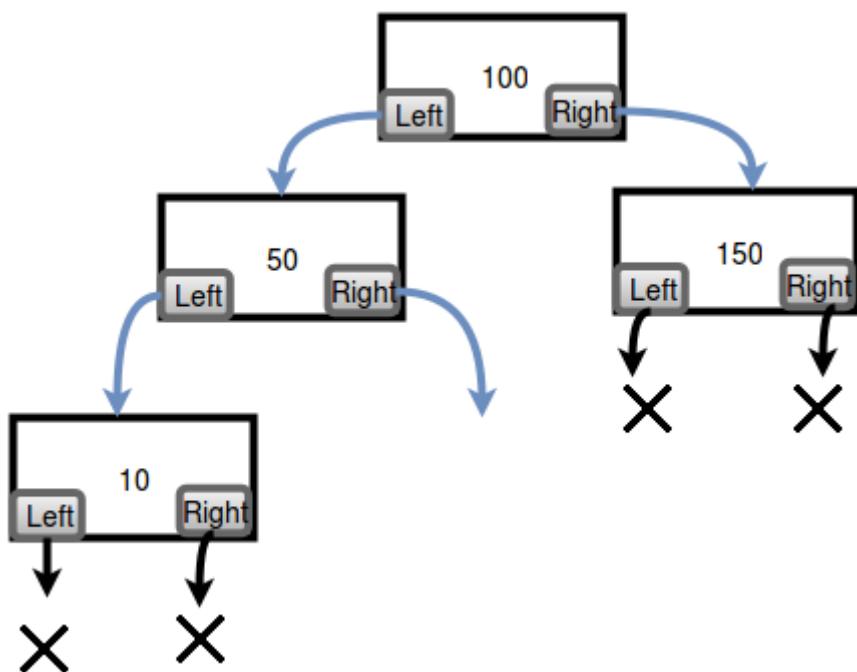
1 struct node
2 {
3     int roll;
4     node *next, *prev;
5 };
6     node *root, *tail;
7     void append(int roll)
8     {
9         if(root==NULL) //If the list is empty
10    {
11        root=new node();
12        root->roll=roll;
13        root->next=NULL;
14        tail=root;
15    }
16    else
17    {
18        node *newnode=new node();
19        newnode->roll=roll;
20        newnode->next=NULL;
21        tail->next=newnode; //add the new node after tail node
22        tail=tail->next; //move tail pointer forward
23    }
24 }

```

এবার append ফাংশন খুব সহজ হয়ে গিয়েছে। নতুন নোডটা tail এর সাথে যোগ করে টেইল এক ঘর এগিয়ে নেয়া হচ্ছে। নোড ডিলিট করার সময়েও এখন আর previous\_node ভ্যারিয়েবলটা রাখা দরকার নেই।

## বাইনারি সার্চ ট্রি

লিংকড লিস্টের আরেকটা ব্যবহার হলো বিভিন্ন ধরণের ট্রি তৈরি করা। যেমন নিচে একটা বাইনারি সার্চ ট্রি তৈরি করা হয়েছে:



বাইনারি সার্চ ট্রি তে বাম পাশের নোডে সবসময় ছোটো মান, ডানের নোডে সমান বা বড় মান থাকে। আমাদের left এবং right নামের দুইটা পয়েন্টার লাগবে।

```

1 struct node
2 {
3     int roll;
4     node *left, *right;
5     node() //initialize the node using null
6     {
7         left=NULL;
8         right=NULL;
9     }
10    };
11    node *root;

```

---

নতুন নোড ইনসার্ট করার সময় আগের মতই লুপ চালিয়ে শেষ নোডে আসতে হবে। লুপ চালানোর সময় বামে নাকি ডানে যাবে সেটা নোডের মান এবং নতুন মান তুলনা করে বের করতে হবে।

```

1 void insert(int roll)
2 {
3     if(root==NULL) //first node in tree
4     {
5         root=new node();
6         root->roll=roll;
7     }
8     else
9     {
10        node *current=root,*parent;
11        while(current!=NULL)
12        {
13            if(roll<current->roll)
14            {
15                parent=current; //keep track of parent node
16                current=current->left;
17            }
18            else
19            {
20                parent=current;
21                current=current->right;
22            }
23        }
24        node *newnode=new node();
25        newnode->roll=roll;
26        if(newnode->roll<parent->roll) parent->left=newnode;
27        else parent->right=newnode;
28    }
29 }

```

---

কোড ঠিক আছে নাকি বোঝার জন্য একটা প্রিন্ট ফাংশন লিখি:

```

1 void print_preorder(node *current)
2 {
3     if(current==NULL) return;
4     cout<<current->roll<<endl;
5     print_preorder(current->left);
6     print_preorder(current->right);
7 }
8

```

এই ফাংশনটা রিকার্সিভলি নোডগুলার মান প্রিন্ট করবে।

কোডটা যদি বুঝে থাকো তাহলে বাইনারি সার্চ ট্রি থেকে নোড মুছে ফেলার ফাংশনটা লিখে ফেলো। নোড মোছার সময় বেশ কয়েকটা ক্ষেত্রে এখানে আলোচনা করবো না। তুমি কোরম্যানের অ্যালগোরিদম বই থেকে বা গুগলে একটু সার্চ করে শিখে নিতে পারো।

লিংকড লিস্টে সাইকেল কিভাবে বের করতে হয় জানতে আমার [এই লেখাটা পড়ো](#)।

লিংকড লিস্টের কোড লেখার সময় কিছু কমন ভুল হয় শুরুর দিকে। যেমন পয়েন্টারের মান নাল হয়ে ঘাবার পরেও মান প্রিন্ট করার চেষ্টা করে বা আরো সামনে আগানোর চেষ্টা করা, সেক্ষেত্রে কোড রান টাইম ইরোর দিবে। এছাড়া মেমরি ফ্রি করতে ভুলে যাওয়াও খুব সাধারণ একটা ভুল। আরেকটা ভুল হলো রুট বা টেইল পয়েন্টারের মান বদলে ফেলা।

লিংকড লিস্ট শেখার পর স্ট্যাক, কিউ, বাইনারি ট্রি, হিপ ইত্যাদি ডাটা স্ট্রাকচারগুলো লিংকড লিস্ট দিয়ে ইমপ্লিমেন্ট করা শিখতে হবে। তুমি যদি কন্টেস্ট করো আর আরেকটু অ্যাডভান্সড কিছু শিখতে চাও তাহলে এখান থেকে [ট্রাই ডাটা-স্ট্রাকচার](#) কিভাবে লিংকড লিস্ট ব্যাবহার করে ইমপ্লিমেন্ট করে শিখে নিতে পারো।

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাটা স্ট্রাকচার : স্ট্যাক

[shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

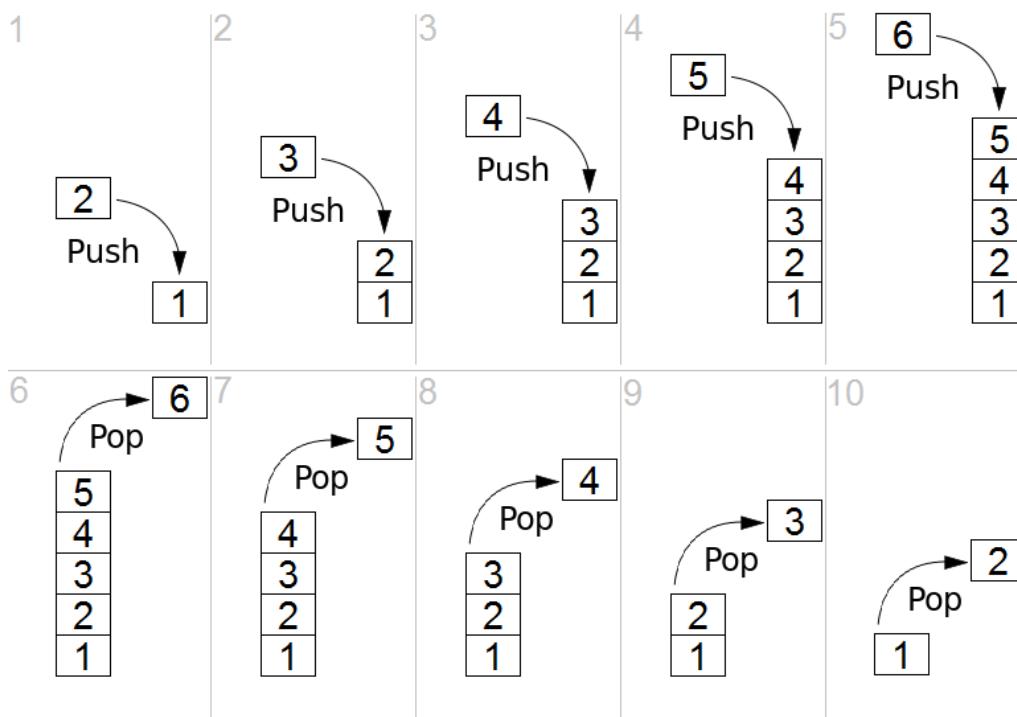
শাফায়েত

নভেম্বর ১১, ২০১৫

যেকোনো ডাটা স্ট্রাকচার কোর্সে একদম শুরুর দিকে যেসব ডাটা স্ট্রাকচার পড়ানো হয় তার মধ্যে স্ট্যাক অন্যতম। স্ট্যাককে বলা হয় LIFO বা লাস্ট-ইন-ফার্স্ট-আউট ডাটা স্ট্রাকচার। তুমি এভাবে চিন্তা করতে পারো, তোমার কাছে অনেকগুলো বই একটার উপর আরেকটা সাজানো আছে, তুমি চাইলে সবার উপরের বইটা সরিয়ে ফেলতে পারো(Pop), অথবা সবার উপরে আরেকটা বই রাখতে পারো(Push)। এটা হলো বইয়ের একটা স্ট্যাক। তুমি এই স্ট্যাকের উপরে ছাড়া কোনো জায়গায় বই তুকাতে পারবে না, উপরের বই ছাড়া কোনো বই সরাতে পারবে না, এগুলো করলে সেটা আর স্ট্যাক থাকবে না।

স্ট্যাক হলো কিছু বস্তুর(এলিমেন্ট) একটা সংগ্রহ। এখানে দুইরকম অপারেশন করা যায়:

- Push(new\_element): স্ট্যাকের উপরে নতুন বস্তুটা রাখো। যদি শুরুতে স্ট্যাকটা খালি হয়, তাহলে প্রথম জায়গায় বস্তুটা রাখতে হবে।
- Pop(): স্ট্যাকের উপরের বস্তুটা সরিয়ে ফেলো। যদি শুরুতে স্ট্যাকটা খালি হয় তাহলে এই অপারেশনটা করা সম্ভব না।



চিত্র ১: স্ট্যাক এ পুশ এবং পপ অপারেশন (সোর্স: উইকিপিডিয়া)

চিত্র ১ এ দেখা যাচ্ছে কিভাবে স্ট্যাকে পুশ এবং পপ অপারেশন করতে হয়। স্ট্যাকে আরেকটা অপারেশন থাকতে পারে Peek(), যেটা ব্যবহার করে সবার উপরের বস্তুটা কি জানা যাবে বস্তুটা পপ না করেই।

স্ট্যাকের আকার যদি সীমিত হয়, যেমন যদি ১০টার বেশি বস্তু রাখার সামর্থ্য কোনো স্ট্যাকের না থাকে তাহলে ১১তম বস্তু পুশ করলে স্ট্যাকটা “ওভারফ্লো” স্টেট এ চলে যাবে এবং তোমার প্রোগ্রাম ক্র্যাশ করবে। একই ভাবে খালি স্ট্যাক থেকে কিছু পপ করার চেষ্টা করলে স্ট্যাক ‘আন্ডারফ্লো’ স্টেটে চলে যাবে এবং প্রোগ্রাম ক্র্যাশ করবে। স্ট্যাক ব্যবহারের সময় এই দুই ব্যাপারে খুব সতর্ক থাকতে হয়।

স্ট্যাক ইম্প্লিমেন্ট করা খুব সহজ। সাধারণ অ্যারে ব্যবহার করেই নির্দিষ্ট আকারের স্ট্যাক ইম্প্লিমেন্ট করা যায়। সবার উপরের এলিমেন্টটা কোন ইনডেক্সে আছে সেটা একটা অতিরিক্ত ভ্যারিয়েবলে সেভ করে রাখতে হবে। নিচে পাইথনে একটা ইম্প্লিমেন্টেশন দেখানো হলো, একই ভাবে সি++ এও তুমি করতে পারবে:

## Python

```

1 class Stack:
2     def __init__(self, max_size): #initialize a stack of max_size
3         self.top_pointer = -1 #Keep track of top element using this
4         self.stack = [None for x in range(max_size)] #create a list of max_size
5
6     def push(self, new_element):
7         self.top_pointer = self.top_pointer + 1 #Move the pointer
8         self.stack[self.top_pointer] = new_element #Add the new_element to the top
9
10    def pop(self):
11        last_element = self.stack[self.top_pointer]
12        self.top_pointer = self.top_pointer - 1 #Move the pointer
13        return last_element #Pop the last element
14
15    def peek(self):
16        return self.stack[self.top_pointer]
17
18    def is_empty(self):
19        return top.pointer == -1

```

এই কোডে আমি কোনো ওভারফ্লো বা আন্ডারফ্লো হ্যান্ডেল করি নি। তুমি কোডটার উন্নতি করতে চাইলে এ ব্যাপারগুলো নিয়ে কাজ করতে পারো। যদি তুমি নির্দিষ্ট আকারের স্ট্যাক না চাও তাহলে লিংকড-লিস্ট ব্যবহার করতে হবে।

স্ট্যাকে প্রতিটা অপারেশনের কমপ্লেক্সিটি  $O(1)O(1)$

## স্ট্যাকের ব্যবহার:

আমরা ব্রাউজারে প্রায়ই আগের ওয়েবসাইটে ফিরে যেতে “Back” বোতামে চাপ দেই। এটা ও স্ট্যাক ব্যবহার করে তৈরি করা যায়। যখন নতুন ওয়েবসাইটে যাবে তখন আগের ওয়েবসাইটটাকে স্ট্যাকের উপরে রেখে দাও, ফিরে যেতে চাইলে স্ট্যাকের উপরের ওয়েব সাইটে ফেরত গিয়ে স্ট্যাক থেকে পপ করে দাও। এভাবে তুমি তোমার সফটওয়্যারে “undo” ফিচার বানাতে পারো।

প্রোগ্রামিং ল্যাঙ্গুয়েজে ফাংশন কল করার সময় স্ট্যাকের খুব গুরুত্বপূর্ণ ব্যবহার আছে। মনে করো তুমি একটা ফাংশন func1 থেকে func2 কল করছো, সেখান থেকে আবার func3 কল করছো:

## Python

```

1 def func_3():
2     return 42
3
4 def func_2():
5     x=20
6     y=func3()
7     return x+y
8
9 def func_1():
10    x=10
11    y=func2()
12    return x+y
13
14 print func1()

```

`func_1()` এ কিছু লোকাল ভ্যারিয়েবল আছে। তুমি যখন `func_1()` থেকে `func_2()` কে কল করছো তখন `func_1()` এর সব তথ্য একটা স্ট্যাকে ঢুকিয়ে রেখে নতুন ফাংশন `func_2()` কে লোড করা হয়। আবার যখন `func_3()` কে কল করছো তখন `func_2()` কে স্ট্যাকে ঢুকিয়ে রাখে হবে। যদি  $n+1$  টা ফাংশন থাকে তাহলে যখন  $n$  তম ফাংশন টা  $n+1$  তম ফাংশনকে কল করবে তখন স্ট্যাকের চেহারাটা হবে এরকম:

`func_3()` এর কাজ শেষ হবার পর আবার `func_2()` কে মেমরিতে লোড করা হবে এবং স্ট্যাক থেকে পপ করে দেয়া হবে। এটাকে কল-স্ট্যাক বলা হয়।  
রিকার্সিভ ফাংশনও এভাবে আগের স্টেটগুলোকে স্ট্যাকে ঢুকিয়ে রাখে।

হ্যাকাররা এক ধরণের আক্রমণ মাঝে মাঝে ব্যবহার করে যাকে বলা হয় “স্ট্যাক স্ম্যার্টিং”। আগ্রহী হলে এই [আর্টিকেলটা](#) পড়তে পারো।

স্ট্যাকের খুবই কমন একটা ব্যবহার হলো ব্রাকেট এর ব্যালেন্স বা ভারসাম্য ঠিক আছে সেটা পরীক্ষা করা (Parenthesis Balance)। মনে করো তোমাকে একটা ব্রাকেট সিকোয়েন্স  $S$  দেয়া হলো এরকম “( $\{$ )”। তোমাকে বলতে হবে সিকোয়েন্সটার ব্যালেন্স ঠিক আছে নাকি নেই। ব্যালেন্স ঠিক থাকবে যদি নিচের শর্ত গুলো পূরণ হয়:

- যদি  $S$  একটা  $O$  দৈর্ঘ্যের স্ট্রিং হয়।
- যদি  $A$  আর  $B$  দুইটা ব্যালেন্সড সিকোয়েন্স হয় তাহলে  $AB$  ও একটা ব্যালেন্সড সিকোয়েন্স। যেমন  $GA = "(\{\})"$  এবং  $B = "()"$  দুইটা ব্যালেন্সড সিকোয়েন্স হলে  $AB = "(\{\})()$  ও ব্যালেন্সড।
- যদি  $S$  একটা ব্যালেন্সড সিকোয়েন্স হয় তাহলে ( $S$ ) অথবা  $\{S\}$  ও ব্যালেন্সড। যেমন  $S = "{}()$  ব্যালেন্সড হলে ( $S$ ) = “( $\{\})()$ ” এটাও একটা ব্যালেন্সড সিকোয়েন্স।

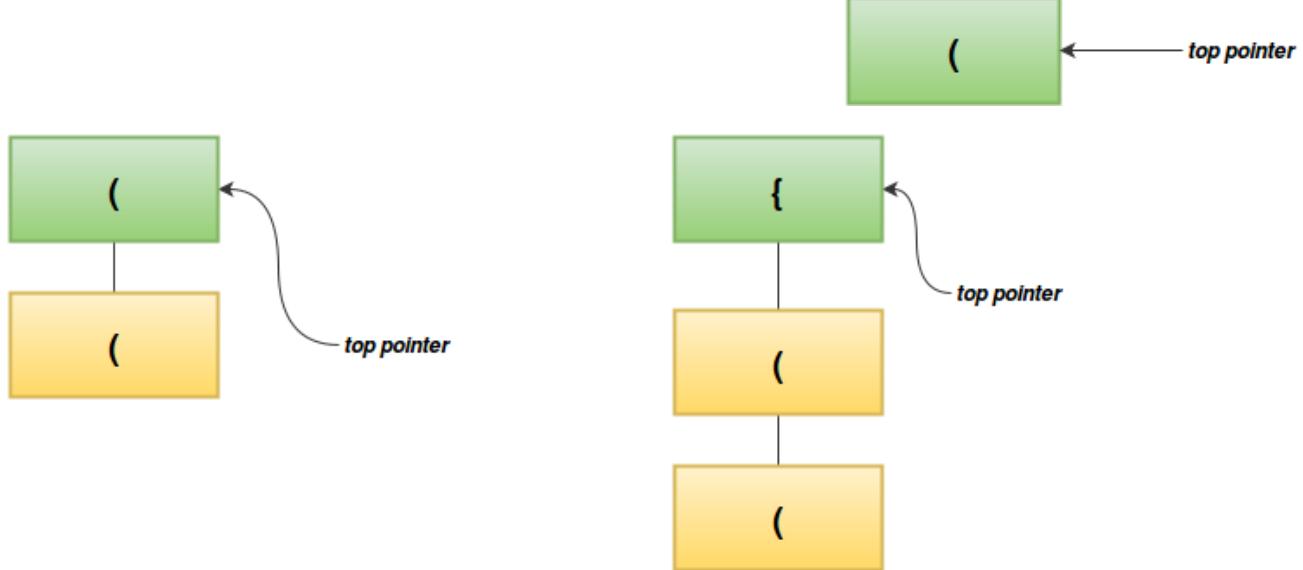
কিছু ভারসাম্যহীন সিকোয়েন্স হলো “(”, “({})”, “()()” ইত্যাদি।

“(“ আর “{“ কে আমরা বলবো “ওপেন ব্রাকেট”, আর “)”” এবং “}” কে বলবো “ক্লোজিং ব্রাকেট”।

এখন আমরা  $S = "(\{\})"$  ব্যালেন্সড নাকি পরীক্ষা করবো স্ট্যাক ব্যবহার করে। আমরা বাম থেকে ডানে একটা একটা ক্যারেক্টার নিয়ে কাজ করবো। কোনো “ওপেন ব্রাকেট” পেলেই সেটাকে স্ট্যাকে পুশ করবো।

এই সিকোয়েন্সের প্রথম ক্যারেক্টার “(“, আমরা এটাকে একটা স্ট্যাকে পুশ করবো।

২য় এবং ৩য় ক্যারেক্টার হলো “(“ এবং “{“। এদেরকেও পুশ করবো:



পরের ক্যারেক্টারটা হলো “}”。 ক্লোজিং ব্রাকেট যখন পাবো তখন আমরা দেখবো স্ট্যাকের উপরে কি আছে:

- যদি স্ট্যাকটা খালি হয় তাহলে সিকোয়েন্সটা ব্যালেন্সড না।
- যদি বর্তমান ক্যারেক্টারটা “}” হয় তাহলে স্ট্যাকের উপরে অবশ্যই “{“ থাকতে হবে।

- যদি বর্তমান ক্যারেকটারটা ")" হয় তাহলে স্ট্যাকের উপরে অবশ্যই "(" থাকতে হবে।

আমাদের স্ট্যাকের উপরে "(" আছে যেটা ")" এর সাথে ম্যাচ করে। আমরা ")" নিয়ে কিছু করবো না কিন্তু "(" কে স্ট্যাকের উপর থেকে পপ করে দিবো।

শেষ ক্যারেকটারটা হলো ")" যেটা "(" এর সাথে ম্যাচ করে, আবার আমরা পপ করবো:

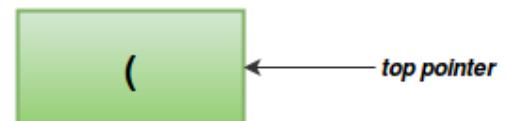
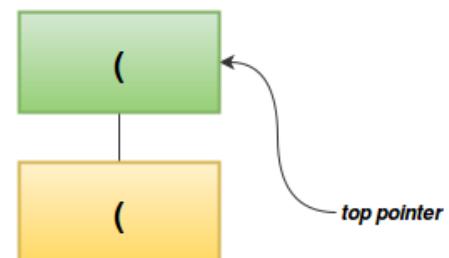
সব কাজ শেষে দেখা যাচ্ছে স্ট্যাকে এখনো একটা ব্রাকেট থেকে গেছে যেটা কারো সাথে মিলানো যাচ্ছে না! তারমানে আমাদের ব্রাকেট সিকোয়েন্সটা ব্যালেন্সড না। ব্যালেন্সড হলে সব ক্যারেকটার নিয়ে কাজ করার পর অবশ্যই স্ট্যাক খালি হয়ে যেত।

আমরা স্ট্যাক এর কোড ইতিমধ্যেই লিখেছি, এবার ব্রাকেট ব্যালেন্স করার কোডটা লিখে ফেলি:

Python

```

1 def checkBalance(s):
2     mystack=Stack(len(s))
3     if s=="":
4         return True
5     for c in s:
6         if c=="(" or c=="{":
7             mystack.push(c) #push the opening bracket
8         else:
9             if mystack.is_empty():
10                 return False
11             if c=="{" and mystack.peek()!=")": #the brackets dont match
12                 return False
13             if c=="(" and mystack.peek()!=")": #the brackets dont
14                 matchs
15             return False
16             mystack.pop() #pop matching brackets
17             if mystack.is_empty(): #stack must be empty at the end
18                 return True
19             return False
20
21
22
23     print checkBalance("{}")
24     print checkBalance("()(((")
25     print checkBalance("(){}")
```



অ্যালগোরিদমটা বুঝেছো নাকি পরীক্ষা করতে [UVA 674](#) প্রবলেমটা সমাধান করে ফেলো।

স্ট্যাকের আরেকটা গুরুত্বপূর্ণ ব্যবহার হলো গাণিতিক ইকুয়েশনের মান বের করা, যেমন  $(1+(2+5)*3)+(5*7)(1+(2+5)*3)+(5*7)$  এরকম একটা ইকুয়েশন দেয়া থাকলে মান বের করা খুব সহজ না। কিন্তু [রিভার্স পলিশ নোটেশনে](#) লিখলে স্ট্যাক দিয়ে খুব সহজে মান বের করা যায়। এটা এখন তুমি সহজেই উইকিপিডিয়া থেকে শিখে ফেলতে পারবে, আমি বিস্তারিত লিখবো না, আমার কাজ শুধু বেসিক জিনিসগুলো তোমাকে জানানো, কিন্তু ভালো করতে হলে নিজে কষ্ট করে শেখার কোনো বিকল্প নেই 😊

হ্যাপি কোডিং।



# ডাটা স্ট্রাকচার: কিউ এবং সার্কুলার কিউ

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

10/7/2016

কিউ একটা বেসিক ডাটা স্ট্রাকচার। এটাকে তুমি চিন্তা করতে পারো বাসের লাইনের মত, যে সবার সামনে দাঢ়িয়ে আছে সে সবার আগে উঠবে, নতুন কোনো যাত্রী আসলে সে লাইনের পিছনে দাঢ়াবে।

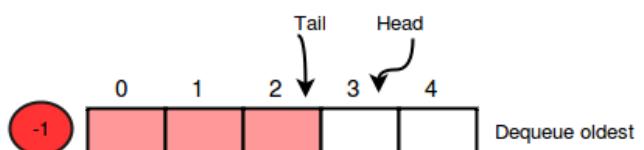
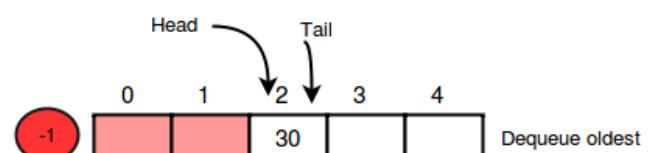
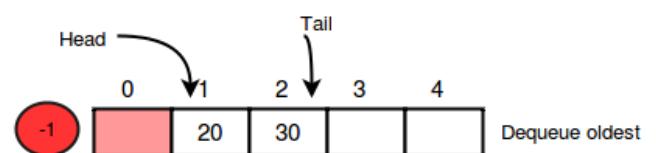
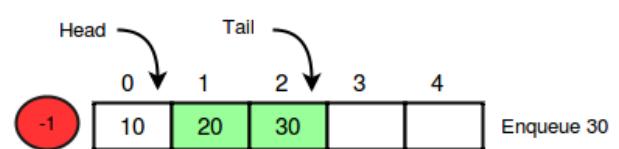
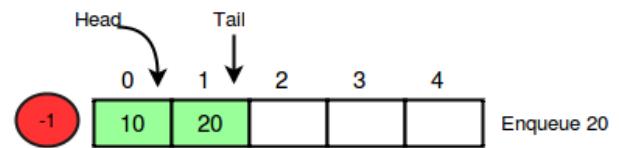
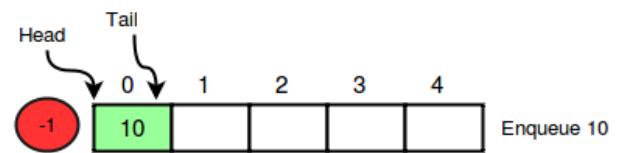
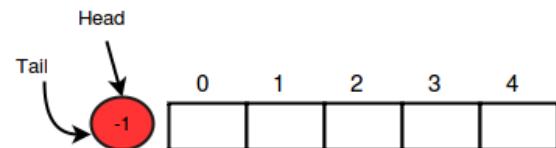
কিউতে দুইরকম অপারেশন থাকে। এনকিউ(Enqueue) মানে হলো কিউতে নতুন এলিমেন্ট যোগ করা এবং ডিকিউ(Dequeue) বা পপ মানে হলো সবথেকে পুরোনো এলিমেন্টটা কিউ থেকে সরিয়ে ফেলা।

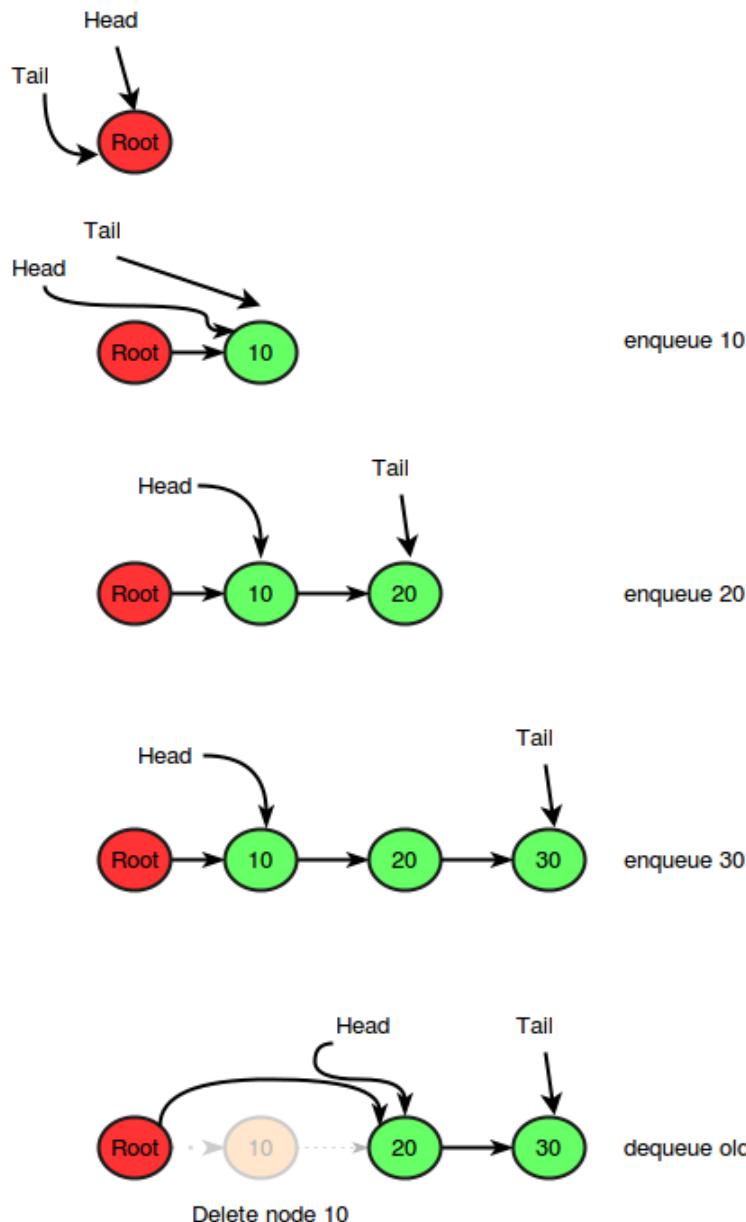
অ্যারে ব্যবহার করে আমরা ফিল্ড সাইজের কিউ ইম্প্লিমেন্ট করতে পারি। আমাদেরকে সবসময় দুইটা পয়েন্টার রাখতে হবে, হেড (Head) পয়েন্টার নির্দেশ করবে কিউয়ের সামনের এলিমেন্টের পজিশন এবং টেইল (Tail) পয়েন্টার নির্দেশ করবে পিছনের এলিমেন্টের পজিশন। একদম শুরুতে Head = -1, Tail = -1 রাখতে পারো। প্রতিটি এলিমেন্ট এনকিউ করার সময় টেইলকে এক পজিশন সামনে এগিয়ে নিয়ে সেই পজিশনে নতুন এলিমেন্টকে রাখতে হবে। ডিকিউ করার সময় শুধুমাত্র হেড একধাপ এগিয়ে নিতে হবে। একটা সিমুলেশন দেখলেই ব্যাপারটা পরিষ্কার হবে:

লক্ষ কর, শেষের এলিমেন্টটা ডিকিউ করার পর হেড টেইলের এর সামনে চলে গেছে। এতে কোনো সমস্য নাই, Head > Tail বা Head = -1 মানে হলো কিউটা পুরোপুরি খালি।

উপরের কিউতে সর্বোচ্চ ৫টা এলিমেন্ট রাখা যাবে। কিন্তু আরো বড় একটা সমস্যা আছে। Head বা Tail সবসময় সামনে আগাছে এবং ডিকিউ করার সময় অ্যারের যে জায়গাটা ফাকা হয়ে যাচ্ছে সেটা দ্বিতীয়বার ব্যবহার করার কোনো উপায় নেই! সর্বশেষ ধাপে Tail = 2 হয়ে গিয়েছে, যদিও কিউটা খালি। তারমানে প্রথম ৩টা পজিশন আর ব্যবহার করতে পারবে না। এই কারণে বাস্তবে কখনোই এভাবে কিউ ইম্প্লিমেন্ট করা হয় না, করলে মেমরির সর্বোচ্চ ব্যবহার করতে পারবে না।

একটা সমাধান হলো **লিংকড লিস্ট** ব্যবহার করা। লিংকড লিস্ট ব্যবহার করলে প্রথম সুবিধা হলো কিউ এর সাইজ ফিল্ড করে দেয়া দরকার নেই। আরেকটা সুবিধা হলো তুমি যখন ডিকিউ করবে তখন টেইল পয়েন্টার যে এলিমেন্টকে পয়েন্ট করে আছে তাকে মেমরি থেকে মুছে দিতে পারবে। লিংকড লিস্ট ব্যবহার করে একটা সিমুলেশন দেখি।

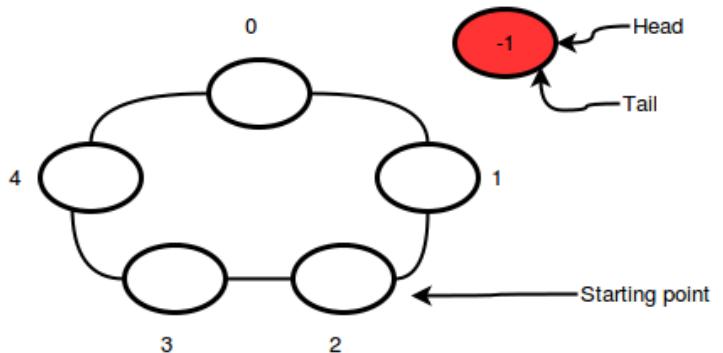


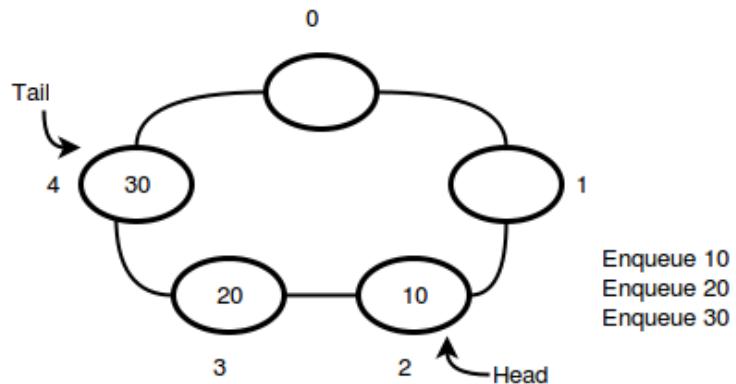


আমরা দেখতে পাচ্ছি এখানে কোন মেমরি অপচয় হবার সুযোগ নেই।

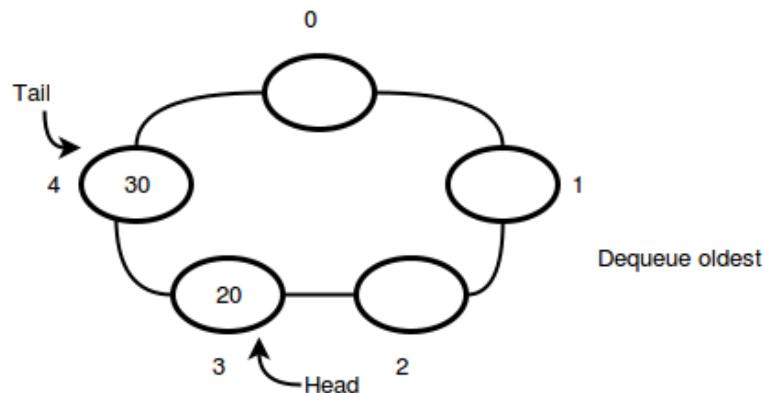
ফিল্ড সাইজের অ্যারে দিয়েও এমনভাবে কিউ ইম্প্লিমেন্ট করা ষাতে মেমরি অপচয় না হয়। একে বলা হয় সার্কুলার কিউ যা দেখতে অনেকটা এরকম:

সার্কুলার কিউতে তুমি শুরুতে ঘেকোনো পজিশনে  
এনকিউ করতে পারো, উদাহরণ হিসাবে ছবিতে 2 নম্বর  
ইনডেক্সকে স্টার্টিং পয়েন্ট ধরেছি। আগের মতোই কিউ  
10, 20 এবং 30 এনকিউ করার পর দেখতে হবে এরকম:





ডিকিউ ও একই ভাবে করবো:

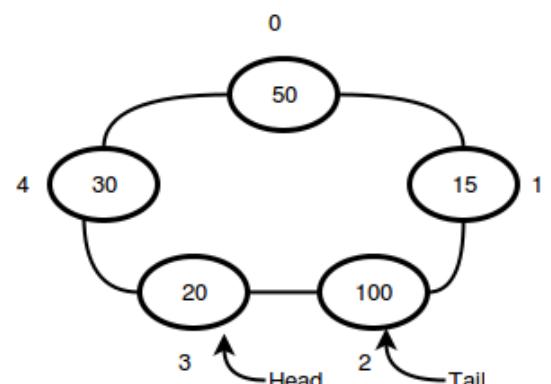


এখন মনে করো কিউটা সম্পূর্ণ ভরে গেছে:

এখন যদি আরো একটা নতুন এলিমেন্ট এনকিউ করতে চাও তাহলে  
তোমার প্রয়োজনের উপর ডিপেন্ড করে দুইরকম ঘটনা ঘটতে পারে।  
এনকিউ ফাংশন এরোর দিতে পারে যে জায়গা খালি নেই। অথবা  
সবথেকে পুরানো এলিমেন্টটা ফেলে দিয়ে সেই জায়গায় নতুন  
এলিমেন্টটা রাখতে পারো, এটাকে বলা হয় **সার্কুলার বাফার**।

এখন প্রশ্ন হলো কিভাবে বুঝবে যে সার্কুলার কিউ ফুল হয়ে গেছে নাকি?  
কিউ ফুল হলে অবশ্যই টেইলের পজিশন হেডের এক ধাপ পিছনে হবে।  
সার্কুলার কিউ এর একটা পাইথন কোড দেখ:

Circular Queue  
Python



```

1   Q = []
2   head = -1
3   tail = -1
4   capacity = 5
5   starting_point = 2
6   Q = [None for x in range(0, capacity)]
7   def enqueue(value):
8       global head
9       global tail
10      global capacity
11      global Q
12      global starting_point
13
14      if (tail + 1)%capacity == head:
15          print "Q is full"
16          return
17      if head == -1:
18          head = starting_point
19          tail = head
20          Q[tail] = value
21      else:
22          tail = (tail + 1)%capacity
23          Q[tail] = value
24
25  def deque():
26      global head
27      global tail
28      global capacity
29      global Q
30      global starting_point
31      if head == -1:
32          print "Q is already empty"
33          return
34      Q[head] = None
35      if head == tail:
36          head = -1
37          tail = -1
38      else:
39          head = (head +1)%capacity

```

আরেক ধরণের কিউ আছে যার নাম ডাবল এন্ডেড কিউ। ডাবল এন্ডেড কিউ এর দুই পাশেই এলিমেন্ট প্রবেশ করানো যায়, আবার দুইদিক থেকেই পপ করা যায়। ডাবল এন্ডেড কিউ দিয়ে সমাধান করা যায় এমন একটা মজার সমস্যা আলোচনা করেছি [স্লাইডিং রেঞ্জ মিনিমাম কুয়েরি নিয়ে লেখায়।](#)

প্রায়োরিটি কিউ নামের আরও এক ধরণের কিউ আছে। সেখানে প্রতিটা এলিমেন্টের একটা প্রায়োরিটি থাকে, পপ করার সময় যার প্রায়োরিটি বেশি সে আগে পপ হয়। প্রায়োরিটি কিউ ইমপ্লিমেন্ট করতে হলে হিপ ডাটা স্ট্রাকচার সম্পর্কে জানতে হবে, সেটা নিয়ে আরেকদিন আলোচনা করবো।

প্রোগ্রামিং কনটেন্টে কিউ এর সবথেকে কমন ব্যবহার হলো [ব্রেথড ফার্স্ট সার্চ](#)। প্রায়োরিটি কিউ ব্যবহার করে ডায়াক্রস্ট্রাকচার অ্যালগোরিদম ইমপ্লিমেন্ট করা হয়। আবার অপারেটিং সিস্টেম বিভিন্ন রকমের কিউ ব্যবহার করে টাঙ্ক শিডিউলিং এর জন্য।

প্রতিটি বড় প্রোগ্রামিং ল্যাংগুয়েজেই কিউ লাইব্রেরি আছে যা দিয়ে খুব সহজে কিউ ব্যবহার করা যায়। কিন্তু শেখার সময় নিজেকে অবশ্যই ইম্প্লিমেন্ট করতে হবে, নাহলে কিউ কিভাবে কাজ করে বুঝতে পারবে না।

আজ এই পর্ষ্ণত্বেই, হ্যাপি কোডিং!



# স্লাইডিং রেঞ্জ মিনিমাম কুয়েরি

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

অক্টোবর ১৩, ২০১৫

মনে করো তোমাকে একটা অ্যারে দেয়া হয়েছে যেখানে  $n$  টা সংখ্যা আছে। তোমাকে বলা হলো সেই অ্যারের  $m=3$  আকারের ষতগুলো সাবঅ্যারে আছে সবগুলো থেকে সবথেকে ছোটো সংখ্যাটা বের করতে হবে।

যেমন অ্যারেটা যদি হয়  $10, 2, 5, 9, 6, 8$  তাহলে  $m=3$  সাইজের সবগুলো সাবঅ্যারে হলো:

|                              |
|------------------------------|
| 10, 2, 5, সর্বনিম্ন সংখ্যা 2 |
| 2, 5, 9, সর্বনিম্ন সংখ্যা 2  |
| 5, 9, 6, সর্বনিম্ন সংখ্যা 5  |
| 9, 6, 8, সর্বনিম্ন সংখ্যা 8  |

তাহলে তোমার আউটপুট হবে  $[2, 5, 5, 8]$ ।

$mm$  এর মান ৩ না হয়ে ১ থেকে  $nn$  পর্যন্ত যেকোনো সংখ্যা হতে পারে।

$nn$  এর মান যদি ছোটো হয় তাহলে আমরা সহজেই প্রতিটা সাবঅ্যারের উপর লুপ চালিয়ে সমস্যাটা সমাধান করতে পারি। নিচের পাইথন কোডটি দেখো:

Python

```
1 def brute_rmq(arr,m):
2     res=[]
3     for i in range(0,len(arr)-m+1):
4         subarr=arr[i:i+m] #take subarray of size m, starting from index i
5         res.append(min(subarr)) #append the minimum element in result
6     return res
```

এই কোডের কমপ্লেক্সিটি  $O(n^2)O(n^2)$ ।

আমরা  $O(n\log n)O(n\log n)$  এ সমস্যাটা সমাধান করতে পারি [সেগমেন্ট ট্রি](#) ব্যবহার করে।

স্লাইডিং উইন্ডো এবং মনোটোনাস ডিকিউ ব্যবহার করে সমস্যাটা  $O(n)O(n)$  কমপ্লেক্সিটিতে সমাধান করা সম্ভব, সেটাই আজকে আমরা শিখবো। মনোটোনাস ডিকিউ বা ডাবল-এন্ডেড-কিউ হলো এমন একটা ডিকিউ যেখানে সংখ্যাগুলো সবসময় সর্টেড থাকে।

মনে করি অ্যারেতে সংখ্যাগুলো হলো  $[10, 50, 15, 12, 8]$  এবং  $m=3$ ।

আমরা বাম থেকে ডানে একটা একটা সংখ্যা নিয়ে কাজ করতে থাকবো। আমরা সংখ্যাগুলোকে এমনভাবে ডিকিউতে তুকাবো যেন সবথেকে ছোটো সংখ্যাটা সবসময় সবার ডানে থাকে। তাম ইনডেক্সে যখন থাকবো তখন  $(i-m+1, i)(i-m+1, i)$  সাবঅ্যারের সর্বনিম্ন সংখ্যাটাকে ডিকিউর সবথেকে ডানে পাওয়া যাবে।

প্রথম সংখ্যাটা হলো 10, এটাকে আমরা ডিকিউ তে বামদিক থেকে তুকাবো:

$DQ=[10]$

পরের সংখ্যাটা হলো 50, এটাকেও বামদিক থেকে তুকাবো:

DQ=[৫০, ১০]

পরের সংখ্যাটা হলো ১৫। এখন লক্ষ্য করো, এখন পর্যন্ত যতগুলো সংখ্যা পেয়েছি তাদের মধ্যে যারা ১৫ এর থেকে বড় তারা কথনেই সর্বনিম্ন সংখ্যা হতে পারবে না, কারণ তারা ১৫ এর বামে আছে এবং তারা যে সাবঅ্যারেতে আছে সেগুলোতে ১৫ ও অবশ্যই আছে। এটা বোঝাই অ্যালগোরিদমের সবথেকে গুরুত্বপূর্ণ অংশ। কোনো একটা সংখ্যা ডিকিউতে তুকানোর আগে সেই সংখ্যাটার থেকে যতগুলো বড় সংখ্যা ডিকিউতে আছে সেগুলো বের করে দিতে হবে।

DQ=[১৫, ১০]

তাহলে প্রথম ৩ আকারের সাবঅ্যারে [১০, ৫০, ১৫] এ সর্বনিম্ন সংখ্যা হলো ডিকিউ এর সর্বডানের সংখ্যা ১০।

পরের সংখ্যাটা হলো ১২। তাহলে আমরা ১৫ কে ফেলে দিয়ে ১২ কে তুকাবো।

DQ=[১২, ১০]

লক্ষ্য করো আমরা এখন  $i=3$  নম্বর ইনডেক্সে আছি এবং  $i-m+1=3-3+1=1$  নম্বর ইনডেক্সের বামের কোনো সংখ্যা আমাদের দরকার নেই কারণ সেগুলো রেঞ্জের বাইরে। কিউ এর সবার ডানের সংখ্যা ১০ মূল অ্যারের এর ০ তম ইনডেক্সে অবস্থিত, সেটাকে আমরা ফেলে দিতে পারি।

DQ=[১২]

তাহলে ২য় ৩ আকারের সাবঅ্যারে [৫০, ১৫, ১২] এ সর্বনিম্ন সংখ্যা হলো ডিকিউ এর সর্বডানের সংখ্যা ১২।

পরবর্তি সংখ্যাটা হলো ৪। আমরা ১২ ফেলে দিয়ে ৪ তুকাবো:

DQ=[৪]

তাহলে ৩য় ৩ আকারের সাবঅ্যারে [১৫, ১২, ৪] সর্বনিম্ন সংখ্যা হলো ডিকিউ এর সর্বডানের সংখ্যা ৪।

তাহলে  $O(n)O(n)$  কমপ্লিক্ষিটিতে আমরা সবগুলো রেঞ্জের সর্বনিম্ন সংখ্যাগুলো বের করে ফেললাম।

নিচের পাইথন কোডে উপরের অ্যালগোরিদমটা ইম্প্লিমেন্ট করা হয়েছে। পাইথন না জানলেও বুঝতে সমস্যা হবে না:

Python

```

1 def sliding_rmq(arr, m):
2     DQ = deque()
3     res=[]
4     for i,val in enumerate(arr):
5         while len(DQ) and DQ[0][0]>=val: #DQ[0][0] is the leftmost element of DQ
6             DQ.popleft()
7         DQ.appendleft((val,i)) #pushing a pair containing the value and the index
8         while len(DQ) and DQ[-1][1]<=i-m: #DQ[-1][1] is the index of the rightmost element of DQ
9             DQ.pop() #popping the out-of-range elements
10        if i>=m-1: #We got a m size range
11            print DQ[-1][0] #print the rightmost element of DQ
12            res.append(DQ[-1][0])
13    return res
14
15
16
17

```

---

### চিন্তা করার জন্য সমস্যা:

১. মনে করো তোমাকে  $n$  টা সংখ্যা এবং  $q$  টা রেঞ্জ দেয়া হয়েছে, রেঞ্জগুলো হলো  $[a_1, b_1], [a_2, b_2], \dots, [a_q, b_q]$ । প্রতিটা রেঞ্জের সর্বনিম্ন সংখ্যা বের করতে হবে। কিভাবে করবে?

২. <http://www.spoj.com/problems/PARSUMS/>

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাটা স্ট্রাকচার: ডিসজয়েন্ট সেট(ইউনিয়ন ফাইন্ড)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ২৮, ২০১১

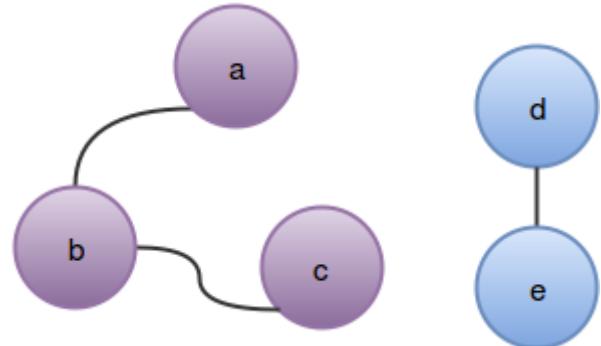
ডাটা স্ট্রাকচার কম্পিউটার সাইন্সের চমতকার অংশগুলোর একটি। আমরা অসংখ্য উপায়ে কম্পিউটারে ডাটা জমা রাখতে পারি। আমরা বাইনারি ট্রি বানাতে পারি, পরে সে গাছ বেয়ে বেয়ে logN এ ডাটা বের করে আনতে পারি, বাসের লাইনের মত কিউ বানাতে পারি, প্রিফিক্স ট্রি বা trie বানিয়ে খুব দুট স্ট্রিং সার্চ করতে পারি। আজ আমরা দেখবো অসাধারণ একটি ডাটা স্ট্রাকচার যার নাম “ডিসজয়েন্ট সেট”।

[kruskal's MST](#) বা [tarjan's offline LCA](#) ইত্যাদি অ্যালগোরিদম ইমপ্লিমেন্ট করতে ডিসজয়েন্ট সেট খুব গুরুত্বপূর্ণ। এটি ইমপ্লিমেন্ট করতে আমাদের একটি অ্যারে ছাড়া কিছু লাগবেনা।

প্রথমে আমরা দেখবো কি ধরণের কাজে আমাদের এই ডাটা স্ট্রাকচারটি দরকার। মনে করি A,B,C,D,E ৫ জন মানুষ। A-B যদি বন্ধু হয় এবং B-C যদি বন্ধু হয় তাহলে আমরা বলতে পারি A-C ও বন্ধু, অর্থাৎ তাদের বন্ধুত্ব transitive। এখন আমি বলে দিলাম যে A-B,B-C,D-E এরা পরস্পরের বন্ধু। এখন তোমাকে প্রশ্ন করলাম A-C কি পরস্পরের বন্ধু? B-D কি পরস্পরের বন্ধু? প্রথম প্রশ্নের উত্তর হবে “হ্যা”, ২য় প্রশ্নের উত্তর হবে “না”。 আমাদের তথ্যগুলোকে গ্রাফের মাধ্যমে দেখাতে পারি:

সহজেই বোঝা যাচ্ছে গ্রাফে যেসব নোড একই কম্পানেন্ট বা সাবগ্রাফের মধ্যে আছে তারা পরস্পরের বন্ধু। তাহলে দু-জন ব্যক্তি বন্ধু নাকি সেটা জানতে হলে আমাদের দেখতে হবে তারা একই সাবগ্রাফে আছে নাকি। সহজেই বিএফএস বা ডিএফএস চালিয়ে এটা বের করা যায়। কিন্তু এগুলোর থেকেও ভালো উপায় হলো “ডিসজয়েন্ট সেট”, কেনো এটা ভালো সেটা কিছুক্ষণ পরেই বুঝতে পারবে।

প্রতিটি মানুষ একটি করে নোড হলে ডিসজয়েন্ট সেট দিয়ে সমাধানের আইডিয়া এরকম:



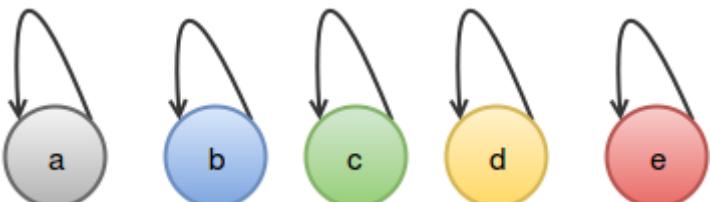
১. যারা পরস্পরের বন্ধু তারা সবাই একই সেটের অন্তর্গত
২. যতগুলো সাবগ্রাফ থাকবে ততগুলো সেট থাকবে
৩. প্রতিটি সেটের একটি *representative* থাকবে।
৪. দুটি নোডের *representative* একই হলে তারা একই সেটে আছে। সুতরাং দুজন ব্যক্তি বন্ধু নাকি বুঝতে আমাদের তারা যে সেটে আছে তার *representative* চেক করতে হবে।

ঘোলাটে লাগছে? সমস্যা নেই, আমরা গ্রাফিকালভাবে এখন ব্যাপারটি বুঝবো, সাথে কোডও দেখবো। শুরুতে কেও কারো বন্ধু নয়। তাহলে সবাই আলাদা আলাদা সেটে আছে। এবং যেহেতু সেটগুলোতে মাত্র একটি করে সদস্য তাই সেই সদস্যটিই হবে সেটের *representative*। ছবিটি দেখো:

আমরা সেটের *representative* কে সবসময় হলুদ রং দিবো।

এই অংশটি ইম্প্লিমেন্ট করবো এভাবে:

Python



```

1 #Python2.7.6
2 #assume a=1,b=2,c=3,d=4,e=5
3 element=5
4 par=[None]*(element+1) #Creating an array
5 with 6 elements
6
7
8 def makeset(n):
9     par[n]=n
10
11 for i in range(1,element+1):
12     makeset(i)

```

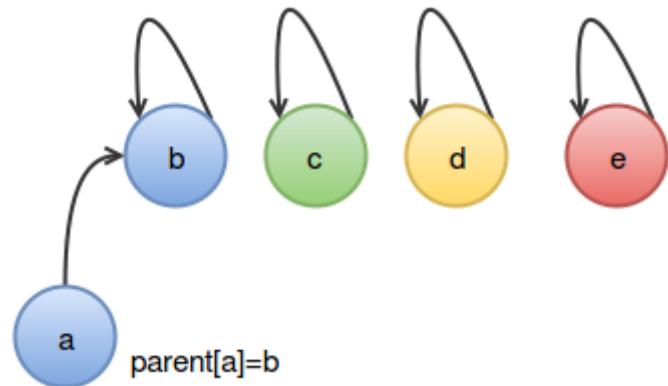
par নামক অ্যারেটা দিয়েই আমরা সেট বানাবো। শুরুতে সবার প্যারেন্ট সে নিজেই। প্যারেন্ট বলতে বুঝাচ্ছে নোডটি কার সাথে সংযুক্ত। একটি মাত্র সাধারণ অ্যারে ব্যবহার করে আমরা পুরো স্ট্রাকচারটি বানাবো! এটা একধরনের ট্রি স্ট্রাকচার।

এখন a আর b হঠাত ঠিক করলো যে তারা বন্ধু হবে। তাহলে তাদেরকে একই সেটের মধ্য আসতে হবে। a ঠিক করলো যে b এর representative কে নিজের representative বানিয়ে ফেলবে, তাহলেই তারা একই সেটে চলে আসবে। তাই a এসে b এর সাথে যুক্ত হয়ে গেলো কারণ b এর representative হলো b নিজেই। ছবি দেখো:

এখন c ভাবলো সেও a এর বন্ধু হবে। এখন c কার নিচে যুক্ত হবে? a এর নিচে c যুক্ত না হবেনা, c যুক্ত হবে a এর representative এর নিচে, তারমানে b এর নিচে।

d,e কেও বন্ধু বানিয়ে দেই:

পরের ধাপটি ভালো করে দেখো। এবার আমরা e আর c কে বন্ধু বানাবো। c যেই সেটে সেটার representative হলো b। তার সাথে আমরা যুক্ত করবো e এর representative কে। e এর representative হলো d। b কে d এর প্যারেন্ট বানিয়ে দিলাম। তাহলে b এখন e এরও representative হয়ে গেলো।



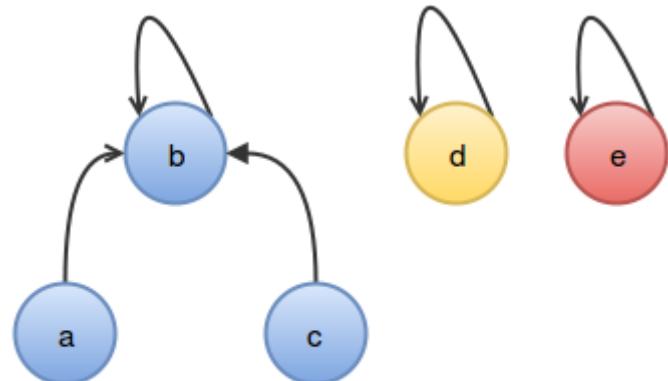
আমরা আবার একটু কোড দেখি:

Python

```

1 #Python2.7.6
2 def union(a,b):
3     u=find(a)
4     v=find(b)
5     if u==v:
6         print "They are already friends"
7     else:
8         par[u]=v #Or you can write par[v]=u too

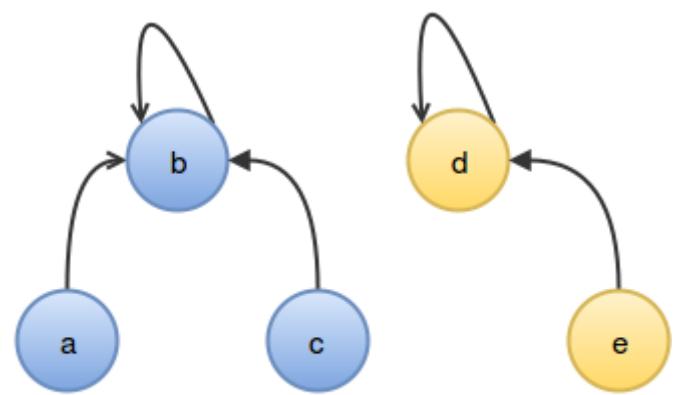
```



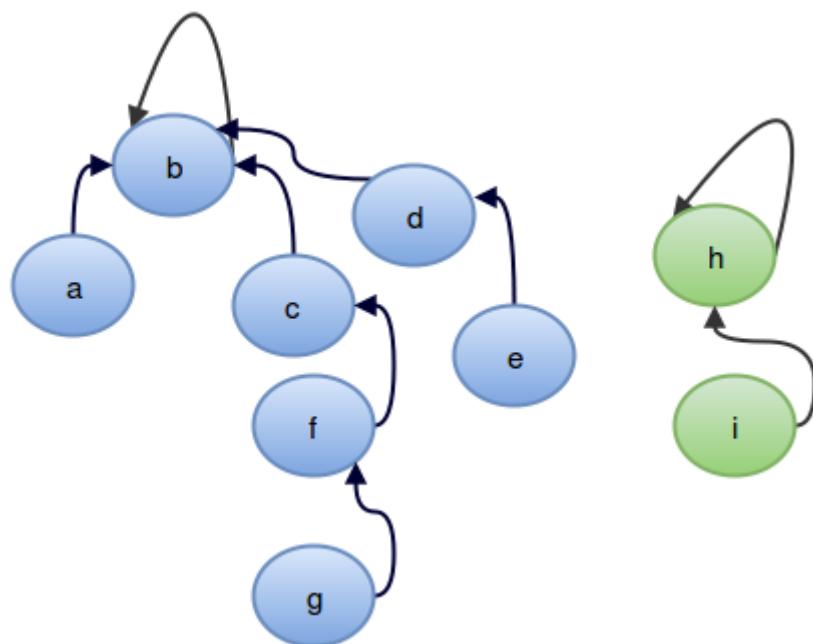
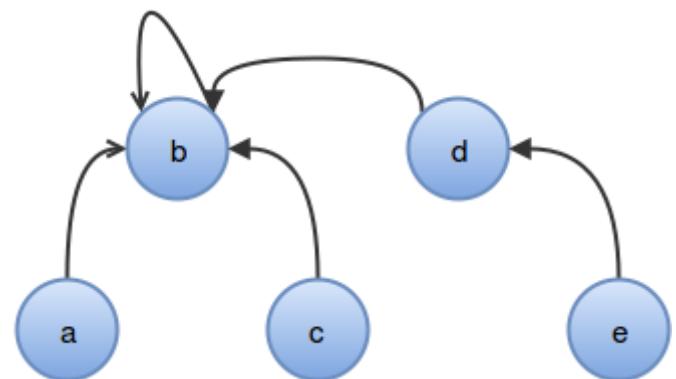
আমরা যে দুটি নোডকে বন্ধু বানাবো তাদের representative বের করলাম find() ফাংশন কল করে। (ফাংশনটির ডেফিনিশন পরে দেখবো) এবার একটিকে আরেকটির প্যারেন্ট বানিয়ে দিলাম।

parent[c]=representative(a)=b

এখন প্রশ্ন হলো representative খুজবো কিভাবে? মনোযোগদিয়ে এই পর্যন্ত পড়ে থাকলে একটা জিনিস চোখে পড়ার কথা, কোনো একটি সেটের **representative element** এর প্যারেন্ট সেই এলিমেন্ট নিজেই, অর্থাৎ এলিমেন্টটি যদি হয় r তাহলে  $par[r]=r$ । আমরা আরেকটু বড় গ্রাফ দেখি। মনে করি a,b,c,d,e এর বন্ধু হতে আরো কয়েকজন চলে এসেছে:



`parent[c]=representative(a)=b`



আগের সেটেই শুধু কিছু নোড যুক্ত করা হয়েছে পরের অংশ বোঝার সুবিধার জন্য। ধরি f এর representative বের করতে হবে। f এর প্যারেন্ট সে নিজে নয়, তাহলে তার representative খুজতে তার প্যারেন্টের প্যারেন্ট চেক করি। f এর প্যারেন্ট c। c ও representative নয় কারণ সে নিজেই নিজের প্যারেন্ট নয়, c এর প্যারেন্ট b এবং b নিজেই নিজের প্যারেন্ট। তাহলে b ই হলো representative। ফাংশনটি লিখে ফেলি:

Python

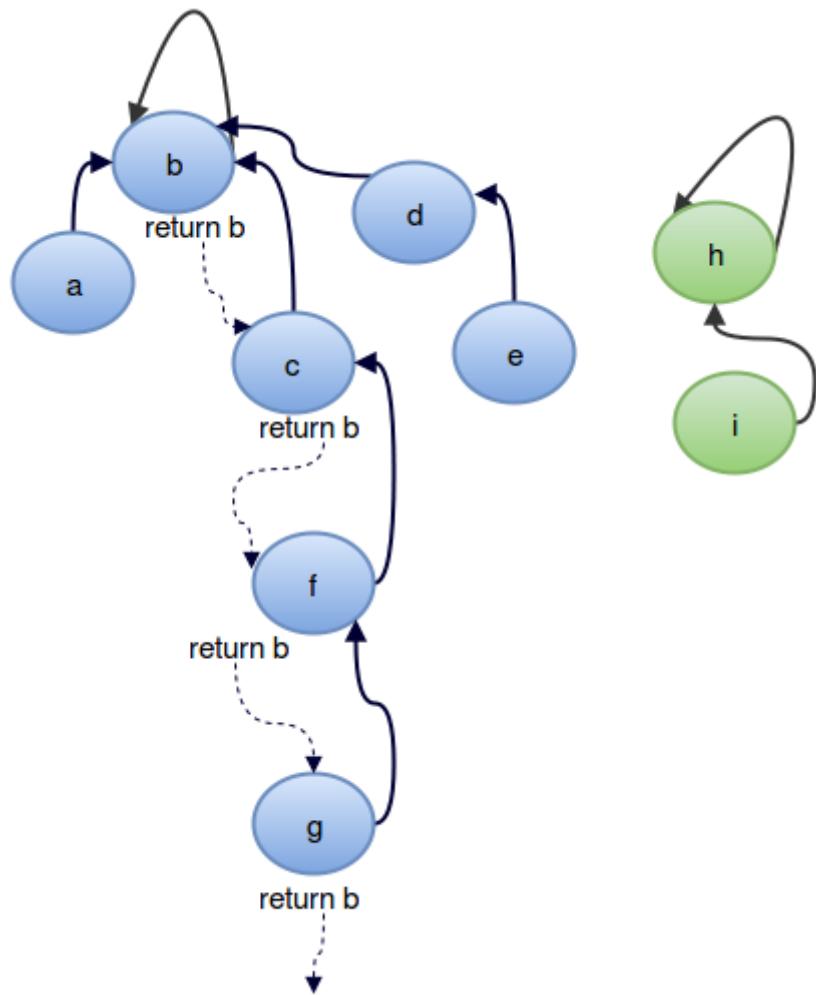
```

1 #python2.7.6
2 def find(r):
3     if par[r]==r: return r
4     return find(par[r])

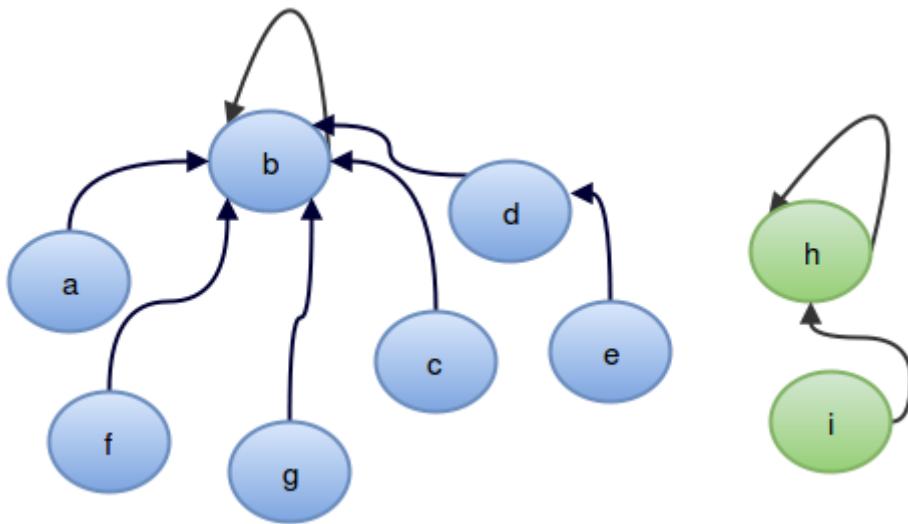
```

ফাংশনটি গাছ(tree) বেয়ে উপরে উঠলো যতক্ষণনা representative কে খুজে পাওয়া যায়। আগের ফাংশনদুটি(union এবং makeset) তাদের আসল কাজ করেছে  $O(1)$  এ। find() ফাংশনে এসে complexity'r প্রশ্ন এসে পড়েছে। worst case এ  $O(n)$  টাইম লাগতে পারে উপরের find() ফাংশনের,tree এর depth যত বেশি হবে তত টাইম বেশি লাগবে। ভাসিটির ল্যাবে লেখা কোডের জন্য এটা ঠিক আছে, কিন্তু কনটেস্ট বা অন্য কোথাও যদি নোড অনেক বেশি হয় তাহলে পারফরম্যান্স খারাপ দিবে।

এখানে একটি চমৎকার optimization আছে। আসলে এই optimization টাই ডিসজয়েন্ট সেট স্ট্রাকচারের সৌন্দর্য। প্রতিটি find() ফাংশন কল এর সাথে সাথে আমরা গ্রাফটি এমন ভাবে পরিবর্তন করবো যেন depth কমে যায়। g এর representative খুজতে আমরা g এর প্যারেন্ট f কে কল দিয়েছি। f তার প্যারেন্টকে কল দিয়ে representative খুজে রিটার্ন করেছে। রিটার্ন ভ্যালুগুলোকে এভাবে দেখতে পারি:



প্রতিটি নোডের নিচে রিটার্ন ভ্যালু লিখে দেয়া হয়েছে। প্রতিটি নোড তার আগের ফাংশন একই ভ্যালু রিটার্ন করে করে, যেটা সবশেষে আমরা পাই, এই ভ্যালুটাই representative। আমরা return find(par[r]) না লিখে যদি আগে লিখতাম par[r]=find(par[r]) এবং তারপর লিখতাম “return par[r]” তাহলে কি ঘটতো? রিটার্ন করার সময় রিটার্ন ভ্যালুটাকে নিজের প্যারেন্ট বানিয়ে ফেলতো, অর্থাৎ সবাই representative কেই বানিয়ে ফেলতো নিজের প্যারেন্ট! ফাংশন কলের সাথে সাথে গ্রাফটি হয়ে যেত এমন:



ত্রি এর depth কমে গিয়েছে, তাই নয় কি? এটাকে বলা হয় path compression। পরে আমরা যখন f এর representative খুজবো তখন আর মাত্র একটি ডাল(edge) বেয়ে উঠতে হবে। wiki তে সুন্দর করে লেখা আছে:

*path compression, is a way of flattening the structure of the tree whenever Find is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as Find recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly.*

মোটামুটি এই হলো আমাদের disjoint set। দুটি নোড একই সেটে আছে নাকি চেক করতে আমরা তাদের representative বের করে তারা সমান নাকি সেটা চেক করবো। দুটি নোড কে একই সেটে নিতে হলে তাদের representative বের করে একটিকে আরেকটির প্যারেন্ট বানিয়ে দিবো। [kruskal](#) এ দুটি নোড একই সাব-ট্রিতে আছে নাকি সেটা চেক করতে disjoint set ব্যবহার করা হয়। একই tree তে না থাকলে union ফাংশনটি কল করে এক করা হয়।

এখন যদি সত্যিই কিছু শিখতে চাও তাহলে ঝটপট কিছু প্রবলেম সলভ করে ফেলো:

[Graph connectivity](#)

[Nature](#)

[Virtual Friend\(\\*\)](#)

আরো জানতে চাইলে:

[টপকোডার টিউটোরিয়াল](#)

[Wikipedia](#)

[সি++ কোডগুলো বদলে পাইথনে লিখে দেয়া হলো, তবে সেজন্য বুঝতে সমস্যা হবার কথা না। সমস্যা হলে যোগাযোগ কর]

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by [AccessPress Themes](#)

# ডাটা স্ট্রাকচার: ট্রাই (প্রিফিক্স ট্রি/রেডিভি ট্রি)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

12/27/2013

ইন্টারনেট যারা ব্যবহার করে তারা সবাই মনে হয় কখনো না কখনো এটা ভেবে অবাক হয়েছে যে গুগলে সার্চ করলে কিভাবে এত তথ্য চোখের নিমিষে সামনে চলে আসে! চিন্তা করে দেখো হাজার কিলোমিটার দূরে দানবাকৃতির ডাটাবেস থেকে প্রযোজনীয় ডাটা খুঁজে তোমার কাছে পাঠিয়ে দিচ্ছে কয়েক সেকেন্ডের মধ্যে। শুধুমাত্র শক্তিশালী হার্ডওয়্যার, দুর্গতির ইন্টারনেট দিয়ে এটা সম্ভব না, এর পিছে আছে দারুণ কিছু সার্চিং অ্যালগোরিদম এবং ডাটা স্ট্রাকচার।

আমরা সেসব জটিল জিনিসে ঘাবোনা, আমরা আজকে শিখবো খুব সহজে ইমপ্লিমেন্ট করা যায় এমন একটা ডাটা স্ট্রাকচার যেটা ব্যবহার করে তুমি খুব দ্রুত অনেক অনেক নাম এর মধ্য থেকে একটা নাম খুঁজে বের করতে পারবে, ডাটাবেসে যত মিলিয়ন নামই থাকুক না কেন তুমি যে নামটা খুঁজে বের করতে চাও সেটাতে যতগুলো অক্ষর আছে সেটার উপর নির্ভর করবে কত সময় লাগবে সেটা খুঁজে বের করতে। তারমানে এখানে তোমার সার্চিং কমপ্লেক্সিটি ডাটাবেস সাইজের উপর নির্ভরশীল না, দারুণ একটা ব্যাপার তাইনা?

ধরো তোমাকে একটা ডিকশনারী দেয়া হলো যেখানে নিচের শব্দগুলো আছে:

```
algo
algea
also
tom
to
```

এখন আমরা এই ডিকশনারিটাকে এমনভাবে মেমরিতে রাখতে চেষ্টা করবো যেন খুব সহজে কোনো একটা শব্দ খুঁজে পাওয়া যায়। একটা উপায় হলো শব্দগুলোকে সর্ট করে রাখা যেটা রিয়েল লাইফে কাগজের ডিকশনারি গুলোতে রাখা হয়, তাহলে বাইনারি সার্চ করেই আমরা কোনো একটা শব্দ খুঁজে বের করতে পারবো। আরেকটা উপায় হলো প্রিফিক্স ট্রি বা সংক্ষেপে ট্রাই(trie) ব্যবহার করা। trie শব্দটা এসেছে “retrieval” শব্দটা থেকে। সেই হিসাবে এটার উচ্চারণ “ট্রি” হওয়ার কথা কিন্তু গ্রাফ থিওরীতে ট্রি এর আরো ব্যপক ব্যবহার আছে তাই এটাকে বলা হয় “ট্রাই”। প্রিফিক্স মানে হলো একটা স্ট্রিং এর শুরু থেকে কয়েকটা ক্যারেক্টার নিয়ে নতুন স্ট্রিং তৈরি করা। যেমন blog এর প্রিফিক্স হলো b,bl,blo এবং blog।

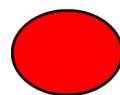
শুরুতে আমাদের একটা রুট নোড ছাড়া কিছুই নেই।

এখন আমরা algo শব্দটাকে যোগ করবো। নিচের ছবিতে দেখো কিভাবে শব্দটা যোগ করা হয়েছে। রুট নোড থেকে আমরা একটা এজ দিবো যেই এজ এর নাম হবে “a”। তারপর নতুন তৈরি হওয়া নোড থেকে “l” নামের একটা এজ তৈরি করবো। এভাবে “g” আর “o” এজ দুইটাও তৈরি করবো। লক্ষ্য করো নোডে আমরা কোনো তথ্য রাখছিনা, খালি নোড থেকে এজ বের করছি।

এখন আমরা algea শব্দটা যোগ করতে চাই। রুট থেকে “a” নামের এজ দরকার, সেটা অলরেডি আছে, নতুন করে যোগ করা দরকার নাই। ঠিক সেরকম a থেকে। এবং। থেকে g তেও এজ আছে। তারমানে “alg” অলরেডি ট্রাই তে আছে, আমরা শুধু e আর a যোগ করবো।

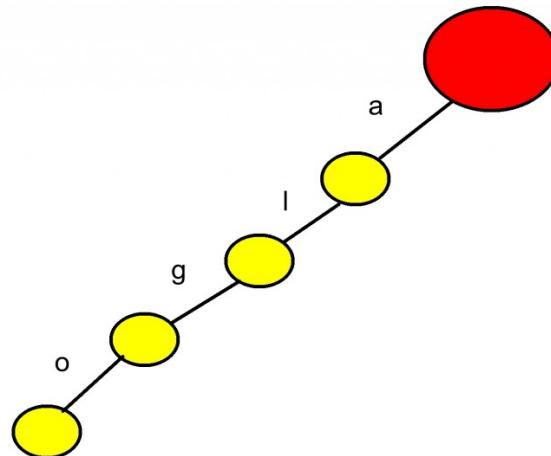
also শব্দটাকে যোগ করবো এবার। রুট থেকে “al” প্রিফিক্স এরইমধ্যে আছে, শুধু “so” যোগ করতে হবে।

এবার “tom” যোগ করি। এবার রুট থেকে নতুন এজ তৈরি করতে হবে কারণ tom এর কোনো প্রিফিক্স আগে যোগ করা হয়নি।



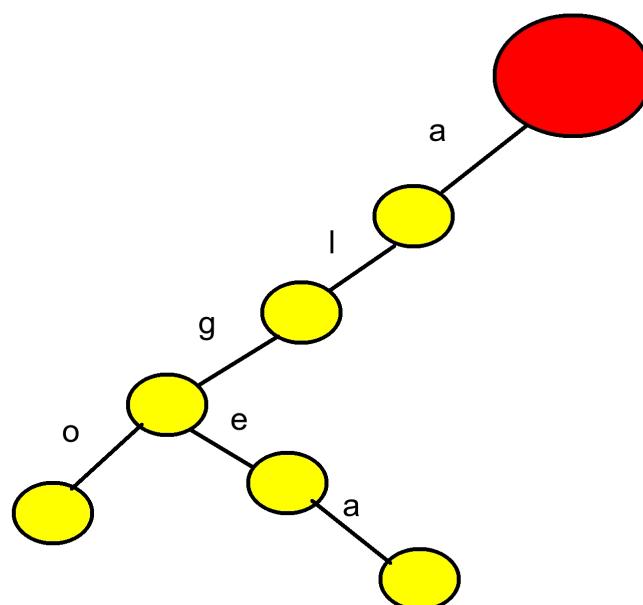
Empty tree with just the root

এখন “to” শব্দটা যোগ করবো কিভাবে?  
 “to” পুরোপুরি tom এর প্রিফিক্স তাই নতুন  
 কোনো এজ যোগ করা দরকার নাই।  
 আমরা যে কাজটা করতে পারি সেটা বলে  
 নোডগুলোতে কিছু এন্ড-মার্ক বসানো।  
 যেসব নোডে এসে অন্তত একটা শব্দ  
 কমপ্লিট হয়েছে সেসব নোডে আমরা  
 এন্ডমার্ক বসিয়ে দেই, ছবিতে সবুজ রঙ  
 হলো এন্ডমার্ক। আগের সব শব্দের জন্য  
 এবং সেই সাথে নতুন শব্দ “to” এর জন্য  
 এন্ডমার্ক বসালে ট্রাইটা এরকম দেখাবে:



নিচয়ই বুঝতে পারছো সবুজ মার্কগুলো  
 কেন বসিয়েছি। মার্ক দেখে সহজেই বোঝা  
 যাচ্ছে কোন কোন শব্দ ট্রাইতে আছে।  
 কোন ক্যারেক্টার নিচ্ছি সেই তথ্য থাকবে  
 এজ এ, আর এন্ডমার্কগুলো  
 থাকবে নোড এ।

এভাবে শব্দগুলো রাখার  
 সুবিধা কি? ধরো তোমাকে  
 বলা হলো “alice” শব্দটা  
 ডিকশনারিতে আছে কিনা  
 বলতে। তুমি শুরু থেকে ট্রাই  
 ধরে আগাতে থাকো। প্রথমে  
 দেখো রুট থেকে a নামের  
 এজ আছে নাকি, তারপর  
 চেক করো a থেকে। নামের  
 এজ আছে নাকি। এরপরে।  
 থেকে i নামের এজ খুজে  
 পাওয়া যাচ্ছেনা, তাই বলতে  
 পারো alice শব্দটা নেই।



“alg” শব্দটা খুজতে দিলে তুমি  
 root->a, a-> এবং l->g

এজগুলো সবই খুজে  
 পাবে, কিন্তু শেষ পর্যন্ত  
 কোনো সবুজ নোডে  
 যেতে

পারবেনা,

তারমানে alg

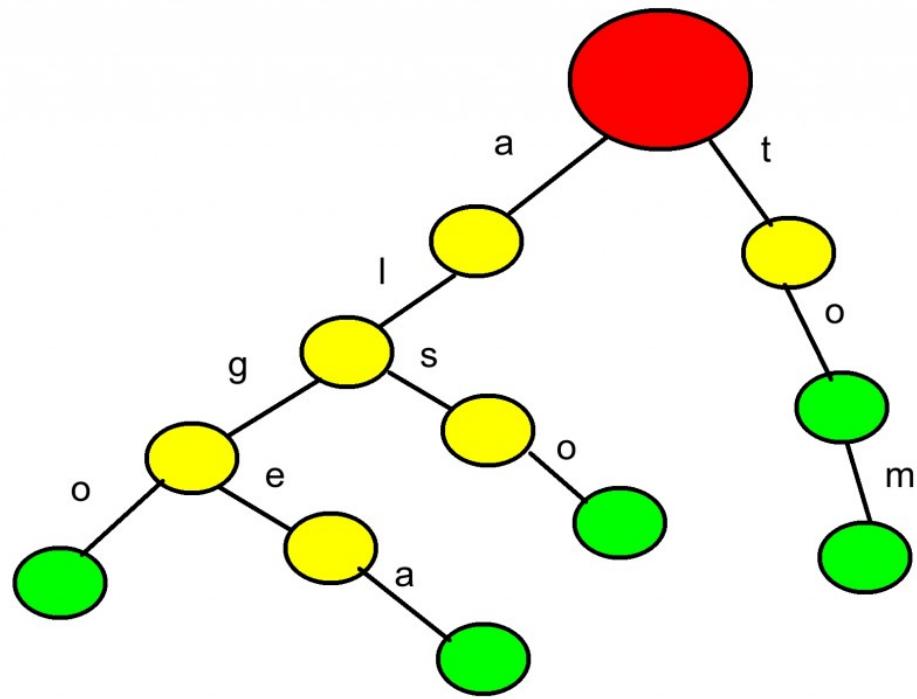
ও ডিকশনারিতে নেই। “tom” খুজতে গেলে তুমি একটা সবুজ নোডে গিয়ে শেষ করবে তাই শব্দটা ডিকশনারিতে আছে।

ট্রাই ইমপ্লিমেন্ট করার সহজ একটা উপায় হলো লিংকড লিস্ট ব্যবহার করা। লিংকড লিস্ট, পয়েন্টার এসব শুনে ভয়ের কিছুই  
 নেই, তুমি যদি লিংকলিস্ট ব্যবহার করতে অভ্যন্ত নাও হও আশা করি এই ইমপ্লিমেন্টেশনটা দেখে শিখে ফেলতে পারবে।  
 আমাদের প্রতিটা নোডে ২টি জিনিস থাকবে:

১. এন্ড-মার্ক রাখার জন্য একটা ভ্যারিয়েবল।

২. প্রতিটা নোড থেকে a,b,c,.....,x,y,z ইত্যাদি নামের এজ তৈরি হতে পারে। আমরা প্রতিটা ক্যারেক্টারের জন্য একটা  
 পয়েন্টারের সাহায্যে একটা নোড আরেকটার সাথে যোগ হবে। a নামের পয়েন্টার দিয়ে যোগ হলে বুঝতে  
 হবে কারেন্ট নোড থেকে a নামের একটা এজ আছে। শুরুতে সবগুলো পয়েন্টার “নাল” থাকবে।

আমরা প্রথমেই নোডটা তৈরি করে ফেলি:



```

1 struct node
2 {
3     bool endmark;
4     node *next[26+1];
5     node()
6     {
7         endmark=false;
8         for(int i=0;i<26;i++) next[i]=NULL; } }*root; int main(){ root=new node(); return 0; }

```

“next[]” অ্যারের প্রতিটা এলিমেন্ট আরেকটা নোডকে পয়েন্ট করে। `next[0]` দিয়ে নতুন নোডকে পয়েন্ট করা হলে সেই এজ এর নাম “`a`”, `next[1]` এর জন্য এজ এর নাম “`b`”, `next[25]` এর জন্য “`z`”। শুরুতে সবগুলো পয়েন্টার নাল। লক্ষ্য করো নোডের ভিতর একটা কনস্ট্রক্টর “`node()`” তৈরি করেছি। যখনই নতুন নোড তৈরির জন্য `new node()` কল করবো তখনই ভ্যারিয়েবলগুলোকে শূন্য বা নাল বানিয়ে দিবে। এটা না দিলে গার্বেজ ভ্যালু থাকতো। `root` ভ্যারিয়েবলটা হলো আমাদের রুট নোড, উপরের ছবিগুলোতে লাল রঙ এর নোড। আসলে রুট একটা পয়েন্টার, যখন `root=new node();` লাইনটা এক্সিকিউট হবে তখনই একটা নতুন নোড তৈরি করে `root` যে মেমরি অ্যাড্রেসকে পয়েন্ট করে সেখানে অ্যাসাইন করে দেয়া হবে। এটাকে একটু গালভরা ভাষায় বলে “`instance`” তৈরি করা। এবার আমাদের একটা ফাংশন লাগবে নতুন শব্দ ট্রাইতে যোগ করার জন্য:

```

1 void insert(char *str,int len) { node *curr=root; for(int i=0;i<len;i++) { int id=str[i]-'a'; if(curr-
2 >next[id]==NULL)
3     curr->next[id]=new node();
4     curr=curr->next[id];
5     }
6     curr->endmark=1;
7     }

```

রুট ভ্যারিয়েবলটা আমাদের সবসময় দরকার হবে তাই “`curr`” এর মধ্যে সেটার কপি তৈরি করে কাজ করি। যেহেতু পয়েন্টার নিয়ে কাজ করছি তাই রুট থেকে নতুন এজ তৈরি করা আর “`curr`” থেকে নতুন এজ তৈরি করা একই কথা। আমরা এখন শুধু `a-z` নিয়ে কাজ করছি, তাই অ্যাসকি ভ্যালুগুলোকে 0-25 এ কনভার্ট করে নিবো ‘`g`’ এর অ্যাসকি ভ্যালু বিয়োগ করে। `insert`

করা খুব সহজ, আমরা শুধু চেক করবো কারেন্ট নোড (curr) থেকে বর্তমানে যে ক্যারেকটারে আছি সেই নামের কোনো এজ আছে নাকি, না থাকলে নতুন নোড তৈরি করতে হবে। এরপরে সেই এজ ধরে আমরা পরের নোডে যাবো। সবার শেষ নোডটায় এন্ড-মার্ক true করে দিবো।

ইনসার্ট করার পর এখন সার্চ করবো। এটা আসলে ঠিক ইনসার্ট এর মতোই। পার্থক্য হলো যে এজটা দরকার সেটা না থাকলে তৈরি করে নিচ্ছিলাম, এখন এজ না থাকলে false রিটার্ন করে দিবো।

C++

```

1  bool search(char *str,int len)
2  {
3      node *curr=root;
4      for(int i=0;i<len;i++)
5      {
6          int id=str[i]-'a';
7          if(curr->next[id]==NULL) return false;
8          curr=curr->next[id];
9      }
10     return curr->endmark;
11 }
```

লক্ষ্য করো সবকাজ ইনসার্ট এর মতোই করেছি। সবার শেষে লাস্ট নোডটার এন্ডমার্ক রিটার্ন করে দিয়েছি। এন্ডমার্ক true হলে শব্দটা আছে, false হলে নাই।

আমাদের মূল কোড শেষ। আমরা এখন যেকোনো শব্দ ট্রাইতে যোগ করতে পারবো, আবার ট্রাই থেকে কোনো শব্দ খুজতে পারবো। অনেক সময় প্রতিটা টেস্টকেস এর জন্য ট্রাই তৈরি করতে গেলে মেমরি লিমিট নিয়ে সমস্যা হয়। তাই নিরাপদ উপায় হলো প্রতি কেস এর পর ব্যবহাত মেমরি-সেল গুলোকে ডিলিট করে দেয়া। শুধু root ডিলিট করলে হবেনা, প্রতিটা নোড করতে হবে। আমরা সে জন্য একটা রিকার্সিভ ফাংশন লিখতে পারি:

C++

```

1  void del(node *cur)
2  {
3      for(int i=0;i<26;i++) if(cur->next[i])
4          del(cur->next[i]);
5      delete(cur);
6  }
```

এই ফাংশনটা প্রতিটা নোডে গিয়ে আগে চাইল্ডগুলোকে ডিলিট করে এসে তারপর নোডটাকে ডিলিট করে দিবে।

সম্পূর্ণ কোডটা এরকম:

C++

```

1  struct node
2  {
3      bool endmark;
4      node *next[26+1];
5      node()
6      {
7          endmark=false;
8          for(int i=0;i<26;i++) next[i]=NULL; } }*root; void insert(char *str,int len) { node *curr=root; for(int
9          i=0;i<len;i++) { int id=str[i]-'a'; if(curr->next[id]==NULL)
10             curr->next[id]=new node();
11             curr=curr->next[id];
```

```

12 }
13 curr->endmark=true;
14
15 }
16 bool search(char *str,int len)
17 {
18 node *curr=root;
19 for(int i=0;i<len;i++)
20 {
21 int id=str[i]-'a';
22 if(curr->next[id]==NULL) return false;
23 curr=curr->next[id];
24 }
25 return curr->endmark;
26 }
27 void del(node *cur)
28 {
29 for(int i=0;i<26;i++)
30 if(cur->next[i])
31 del(cur->next[i]);
32
33 delete(cur);
34 }
35 int main(){
36
37 puts("ENTER NUMBER OF WORDS");
38 root=new node();
39 int num_word;
40 cin>>num_word;
41 for(int i=1;i<=num_word;i++)
42 {
43 char str[50];
44 scanf("%s",str);
45 insert(str,strlen(str));
46 }
47 puts("ENTER NUMBER OF QUERY");
48 int query;
49 cin>>query;
50 for(int i=1;i<=query;i++)
51 {
52 char str[50];
53 scanf("%s",str);
54 if(search(str,strlen(str))) puts("FOUND");
55 else puts("NOT FOUND");
56 }
57 del(root); //ট্রাইটা ধ্বংস করে দিলাম
58 return 0;
}

```

**কমপ্লেক্সিটি:** প্রতিটা শব্দ খুজতে হচ্ছে শব্দটার লেংথ পর্যন্ত, সার্চিং এর কমপ্লেক্সিটি  $O(\text{length})$ । প্রতিটা শব্দ ইনসার্ট করার কমপ্লেক্সিটিও একই। মেমরি কতখানি লাগবে সেটা ডিপেন্ড করে ইমপ্লিমেন্টেশন এবং শব্দগুলোর প্রিফিক্স কতখানি ম্যাচ করে তার উপর। উপরের ইমপ্লিমেন্টেশন দিয়ে প্রায়  $10^6$  টা ক্যারেকটার ট্রাইতে ইনসার্ট করা যাবে ( $10^6$  টা ওয়ার্ড নয়, ক্যারেকটার বা লেটার)।

তুমি চাইলে ট্রাই লিংকলিস্ট ছাড়া সাধারণ অ্যারে ব্যবহার করে ইমপ্লিমেন্ট করতে পারো, নিজে চেষ্টা করো!

## ট্রাই এর কিছু ব্যবহার:

১. একটা ডিকশনারিতে অনেকগুলো শব্দ আছে, কোনো একটা শব্দ আছে নাকি নাই খুজে বের করতে হবে। এই প্রবলেমটা আমরা উপরের কোডেই সলভ করেছি।
২. ধরো তোমার ৩ বন্ধুর টেলিফোন নম্বর হলো “৫৬৭৮”, “৮৩২২”, “৫৬৭”। তুমি যখন প্রথম বন্ধুকে ডায়াল করবে তখন ৫৬৭ চাপার সাথে সাথে ৩য় বন্ধুর কাছে ফোন চলে যাবে কারণ ৩য় বন্ধুর নাম্বার প্রথম জনের প্রিফিক্স। অনেকগুলো ফোন নম্বর দেয়া আছে, বলতে হবে এরকম কোনো নম্বর আছে নাকি যেটা অন্য নম্বরের প্রিফিক্স। ([UVA 11362](#))।
৩. একটা ডিকশনারিতে অনেকগুলো শব্দ আছে। এখন কোনো একটা শব্দ কয়বার “prefix” হিসাবে এসেছে সেটা বের করতে হবে। যেমন “al” শব্দটা উপরের ডিকশনারিতে ৩বার প্রিফিক্স হিসাবে এসেছে (algo, algea, also এই সবগুলো শব্দের প্রিফিক্স “al”)। এটা বের করার জন্য প্রতিটা নোডে একটা কাউন্টার ভ্যারিয়েবল রাখতে হবে, কোনো নোডে যতবার যাবে ততবার কাউন্টারের মান বাড়িয়ে দিবে। সার্চ করার সময় প্রিফিক্সটা খুজে বের করে কাউন্টারের মান দেখবে।
৪. মোবাইলের ফোনবুকে সার্চ করার সময় তুমি যখন কয়েকটা লেটার লিখে তখন সেই প্রিফিক্স দিয়ে কি কি নাম শুরু হয়েছে সেগুলো সার্জেশন বক্সে দেখায়। এটা তুমি ট্রাই দিয়ে ইমপ্লিমেন্ট করতে পারবে?
৫. দুটি স্ট্রিং এর “longest common substring” বের করতে হবে। (subsequence হলে ডিপি দিয়ে সহজে করা যায়, এখানে substring চেয়েছি)।  
(হিন্টস: একটা স্ট্রিং এর শেষ থেকে এক বা একাধিক ক্যারেক্টার নেয়া হলে সেটাকে স্ট্রিংটার সাফিক্স বলে, যেমন blog এর সাফিক্স g,og,log,blog। আর প্রতিটা substring ই কিন্তু কোনো না কোনো সাফিক্স এর প্রিফিক্স!! তাই সবগুলো সাফিক্সকে ট্রাইতে ইনসার্ট করলে কাজটা সহজ হয়ে যায়!)
- (অ্যাডভান্সড) সম্ভবত ২০১১তে ডেফোডিল ইউনিভার্সিটির ন্যাশনাল কনটেন্সে এসেছিলো প্রবলেমটা। একটা ডিকশনারি ইনপুট দেয়া থাকবে। প্রতিবার ডিকশনারির ২টা শব্দ কুয়েরি দিবে, বলতে হবে তাদের মধ্যে common prefix এর দৈর্ঘ্য কত। যেমন algo আর algea এর কমন প্রিফিক্স alg, দৈর্ঘ্য ৩। ট্রাইতে ডিকশনারিতে ইনসার্ট করে প্রতি কুয়েরিতে শব্দদুটি এন্ড-মার্ক থেকে LCA(lowest common ancestor) বের করে প্রবলেমটা সলভ করা যায়।

কিছু প্রবলেম:

[UVA 10226](#)

[\(UVA 11362 Phonebook\)](#)

[UVA 11488 Hyper prefix sets](#)

[POJ 2001 Shortest Prefix](#)

[POJ 1056](#)

হ্যাপি কোডিং!

# ডাটা স্ট্রাকচার: সেগমেন্ট ট্রি-১

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

জুন ১১, ২০১৩

তুমি হয়তো এরকম প্রবলেম কনটেস্টে দেখেছ, একটি ইন্টিজার অ্যারে দেয়া আছে আর অনেকগুলো কুয়েরি দেয়া আছে। প্রতিটি কুয়েরিতে বলেছে একটা রেঞ্জের মধ্যে সবগুলো সংখ্যার যোগফল বলতে। অ্যারের সাইজ  $10^4$ , কুয়েরির সংখ্যা  $10^4$ । বুঝতেই পারছো প্রতি কুয়েরিতে লুপ চালিয়ে যোগফল বের করতে পারবেন। কিভাবে প্রবলেমটি সলভ করবে?

এটা সলিউশন খুব সহজ, তোমাকে কিউমুলেটিভ সাম রাখতে হবে। ধরো একটি অ্যারে আছে  $sum[MAX]$ , তাহলে  $sum[i]$  তে রাখবে  $i$  থেকে  $n$  থেকে  $n$  নম্বর ইনডেক্সে পর্যন্ত সবগুলো সংখ্যার যোগফল।  $i$  থেকে  $j$  পর্যন্ত যোগফল বের করতে দিলে( $i < j$ )  $sum[j] - sum[i-1]$  হবে তোমার উত্তর। বুঝতে না পারলে নিচের উদাহরণটা দেখো:

ইনপুট:

```
arr[] = {4, -9, 3, 7, 1, 0, 2}
```

cumulative sum বের করবে:

```
sum[0] = 0;
(for i=1; i <= n; i++) sum[i] = sum[i-1] + arr[i];
```

তাহলে cumulative sum হবে:

```
sum[] = {4, -5, -2, 5, 6, 6, 8}
```

এটা একদম বাচ্চাদের কাজ, তুমি ৫মিনিটে কোড করে ফেলতে পারবে। কিন্তু প্রবলেমসেটার তোমাকে বিপদে ফেলতে বললো কুয়েরির করার মাঝে মাঝে অ্যারেটি বদলে দেয়া হবে!! মাঝে মাঝে তোমাকে বলবে  $i$ -তম ইনডেক্সের সংখ্যাটিকে  $x$  বানিয়ে দিতে, আবার আগের মতো যোগফলও বলতে বলবে। এখন কি করবে?

[shafaetsplanet.com/blog](http://shafaetsplanet.com/blog)



কুয়েরি:  $i$  থেকে  $j$  ইনডেক্স এর মধ্যে সবগুলো সংখ্যার যোগফল কত?

আপডেট:  $i$  তম ইনডেক্সের সংখ্যাটিকে বদলিয়ে  $x$  বানিয়ে দাও

এবার আর কিউমুলেটিভ সাম দিয়ে কাজ হবেনা, তোমার দরকার হবে সেগমেন্ট ট্রি নামের একটা ডাটা স্ট্রাকচার। ইউনিভার্সিটিতে ডাটা স্ট্রাকচার কোর্সে তোমাকে এটা পড়াবেনা, কিন্তু এটা ব্যবহার করে অনেক কাজ করা যায়।

পরের অংশে যাবার আগে তোমার কিছু জিনিস সম্পর্কে ধারণা পরিষ্কার থাকতে হবে। রিকার্শন সম্পর্কে কোনো রকম অস্পষ্টতা থাকলে আপাতত সামনে না আগানোই ভালো। এছাড়া তুমি যদি মার্জ সর্ট সম্পর্কে জানো তাহলে সেগমেন্ট ট্রি এখনই শেখা কি ঠিক হবে? ঠিক মার্জ সর্টের মতো সেগমেন্ট ট্রি ও “ডিভাইড এন্ড কনকোয়ার” পদ্ধতিতে কাজ করে।

ডিভাইড এন্ড কনকোয়ার পদ্ধতির মূল কথা হলো একটা প্রবলেমকে ভেঙে ছোটো ছোটো অংশ বানাও, আগে সেই ছোট অংশ সলভ করো এবং ছোটো অংশের সলিউশন থেকে বড় অংশের সলিউশন বের করো। আমরা তাই অ্যারেটাকে ২টা অংশে ভাগ করে ফেলবো এবং দুইটা ভাগের যোগফল আলাদা করে বের করবো।

sum=5+3

|   |    |   |   |   |   |   |
|---|----|---|---|---|---|---|
| 4 | -9 | 3 | 7 | 1 | 0 | 2 |
|---|----|---|---|---|---|---|

|   |    |   |   |
|---|----|---|---|
| 4 | -9 | 3 | 7 |
|---|----|---|---|

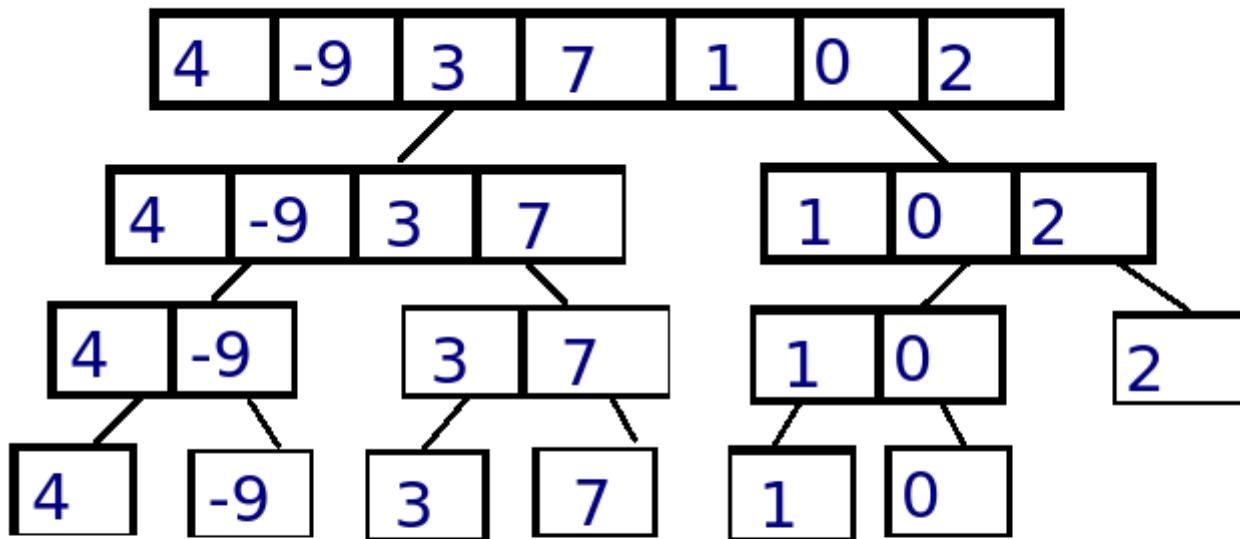
sum=5

|   |   |   |
|---|---|---|
| 1 | 0 | 2 |
|---|---|---|

sum=3

•

তুমি যদি বাম আর ডান পাশের ভাগের যোগফল আলাদা করে বের করতে পারো তাহলে খুব সহজেই বড় অংশটার যোগফল বের করতে পারবে। আমি বলার আগেই বুঝতে পারছো এরপরে কি করবো। ছোটো অ্যারেগুলোকে আরো টুকরা করবো যতক্ষণনা ১ সাইজের টুকরা পাই। ১ সাইজের টুকরোর যোগফল আমরা জানি, সেখান থেকে বড়গুলোর যোগফল বের করে ফেলবো।

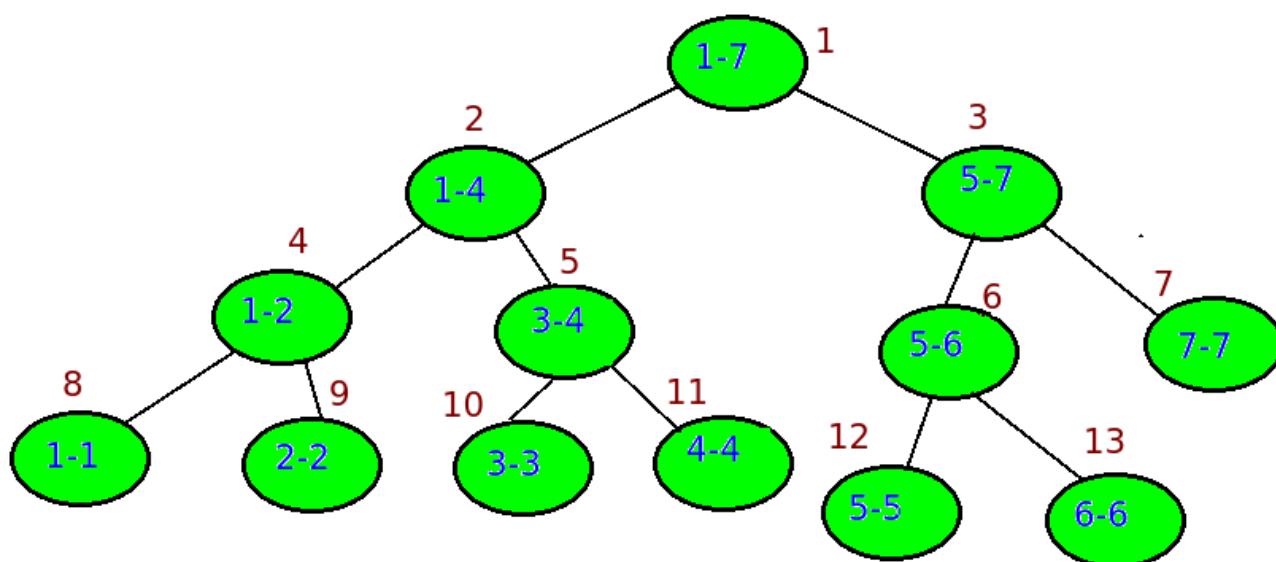


Sum of a segment = sum of (Left segment + Right segment)

Sum of a leaf = value of that leaf

ছবিটা দেখতে বিদ্যুটে হলেও জিনিসটা খুবই সহজ। আমরা অ্যারেটাকে ভাগতে হেট করে ফেলেছি, সবথেকে ছোট অংশের(লিফ নোড) যোগফল আমরা জানি, সেখান থেকে বড় গুলো সহজেই বের করতে পারবো বাম এবং ডানের অংশ যোগ করে।

ছবিটায় প্রতিটা সেগমেন্টকে যদি একটা নোড ধরি তাহলে একটা ট্রি তৈরি হয়ে গিয়েছে, প্রতিটা নোডে আছে একটা অংশ বা রেঞ্জের যোগফল। এটার নামই সেগমেন্ট ট্রি। এখন তোমার মনে হতে পারে এই জিনিস দিয়ে কিভাবে i থেকে j অংশের যোগফল বের করবে কারণ আমরাতো ভাঙ্গি সম্পূর্ণ অ্যারেটা আর সবশেষে পাচ্ছি সবটুকুর যোগফল। কিছুক্ষণের মধ্যে এটার উত্তর পাবে। আমরা ট্রি টাকে একটু অন্যভাবে দেখি:



খেয়াল করে দেখো আগের ট্রি টাই একেছি কিন্তু এবার সেগমেন্টগুলো পুরোটা না দেখিয়ে শুধু রেঞ্জটা লিখেছি। যেমন 3 নম্বর নোডে আছে 5 থেকে 7 ইনডেক্সের সবগুলোর যোগফল। নোডের নাম্বারিং টা গুরুত্বপূর্ণ। রুট নোড হবে 1, তার বামের নোড হবে  $1 \times 2 = 2$ , এবং ডানের নোড হবে  $(1 \times 2 + 1) = 3$ । অর্থাৎ রুট  $x$  হলে বামেরটা হবে  $2x$  এবং ডানেরটা  $2x+1$ । বাইনারি ট্রি অ্যারেতে স্টোর করার জন্য সুবিধাজনক পদ্ধতি এটা।

এখন আগে দেখি কিভাবে এই স্ট্রাকচারটা তৈরি করবো। নিচের কোডটি দেখো, ব্যাখ্যা আছে কোডের নিচে:

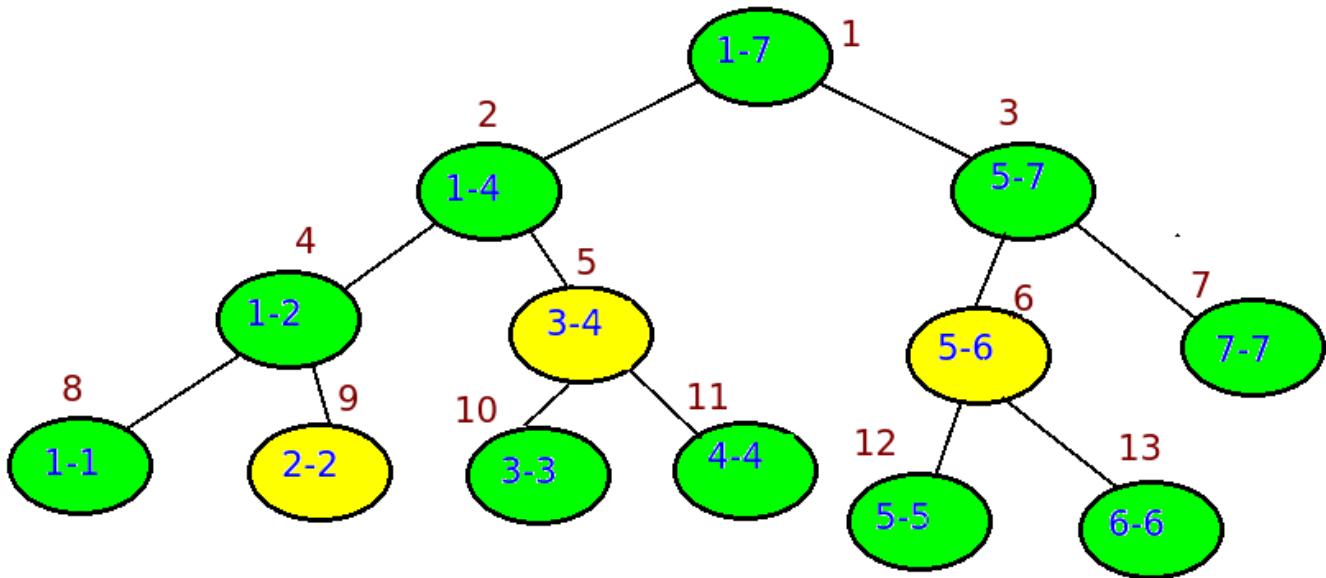
```

1 #define mx 100001
2 int arr[mx];
3 int tree[mx * 3];
4 void init(int node, int b, int e)
5 {
6     if (b == e) {
7         tree[node] = arr[b];
8         return;
9     }
10    int Left = node * 2;
11    int Right = node * 2 + 1;
12    int mid = (b + e) / 2;
13    init(Left, b, mid);
14    init(Right, mid + 1, e);
15    tree[node] = tree[Left] + tree[Right];
16 }
17 int main()
18 {
19     //READ("in");
20     int n;
21     cin >> n
22     repl(i, n) cin
23         >> arr[i];
24     init(1, 1, n);
25
26     return 0;
27 }
```

tree[] অ্যারেতে আমরা ট্রি টাকে স্টোর করবো। ট্রি অ্যারের সাইজ হবে ইনপুট অ্যারের ৩গুণ(কেন??)। init ফাংশনটি arr অ্যারে থেকে ট্রি তৈরি করে দিবে। init এর প্যারামিটার হলো node,b,e, এখানে node=বর্তমানে কোন নোডে আছি এবং b,e হলো বর্তমানে কোন রেঞ্জে আছি। শুরুতে আমরা নোড ১ এ থাকবো এবং ১-৭ রেঞ্জে থাকবো(ট্রি এর ছবিটা দেখো)।

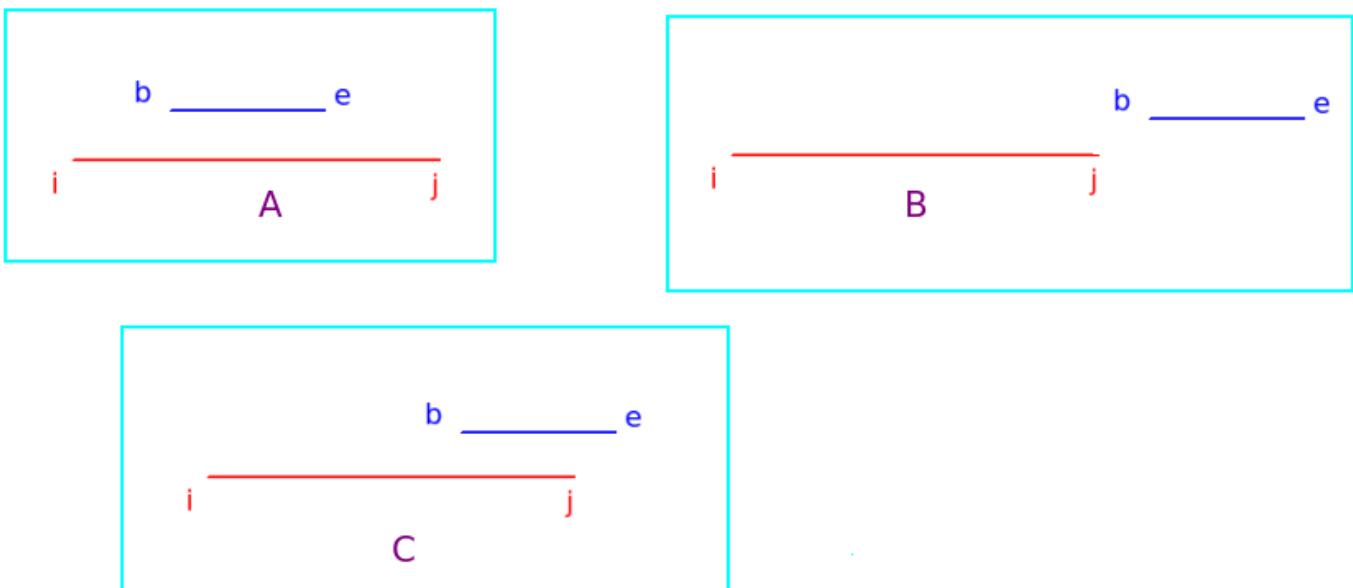
যদি ( $b==e$ ) হয়ে যায় তাহলে আমরা শেষ নোডে পৌছে গেছি, এখানে যোগফল হবে অ্যারেতে যে ভ্যালু আছে সেটাই, সেটাকে ট্রিতে স্টোর করে রিটার্ন করে দিলাম। যদি ( $b==e$ ) না হয় তাহলে অ্যারেটা কে দুই ভাগে ভাগ করতে হবে। বাম পাশের নোডের ইনডেক্স হবে  $node*2$  এবং ডান পাশেরটা  $node*2+1$ । এবং অ্যারেটাকে ভাগবো ঠিক মাঝখানে। এবার রিকার্সিভলি দুই পাশে init কল করলে বাম এবং ডান পাশের ছোটো অংশের যোগফল বের হয়ে যাবে। দুইপাশের কাজ শেষ হয়ে গেলে বর্তমান নোডের যোগফল হবে বাম এবং ডানের নোডের যোগফল।  
বুঝতে সমস্য হলে কোডটা হাতে-কলমে একবার সিমুলেট করো, তাহলেই পরিষ্কার হয়ে যাবে।

এইবার আমাদের একটা কুয়েরি ফাংশন দরকার যেটা i থেকে j এর মধ্যে সবগুলো সংখ্যার যোগফল বলে দিবে। ধরো i=2 এবং j=6। তাহলে লক্ষ্য করো নিচের হলুদ রঙের নোডগুলোর যোগফলই তোমার উত্তর:



২ থেকে ৬ ইনডেক্সের যোগফল বের করতে হলুদ নোডগুলোর যোগফল বের করাই যথেষ্ট

2-6 রেঞ্জের জন্য হলুদ নোডগুলো আমাদের রিলেভেন্ট নোড, বাকিগুলো এক্সট্রা। আমাদের কুয়েরি ফাংশনের কাজ হবে শুধু রিলেভেন্ট নোডগুলোর যোগফল বের করা। কোডটা init ফাংশনের মতোই হবে তবে কিছু কন্ডিশন অ্যাড করতে হবে। ধরো তুমি এমন একটা নোডে আছো যেখানে b-e রেঞ্জের যোগফল আছে। তুমি এই নোডটা রিলেভেন্ট কিনা সেটা কিভাবে বুবাবে? এখানে ৩ধরণের ঘটনা ঘটতে পারে:



কেস A: ( $b \geq i \&& e \leq j$ ) এরকম হলে কারেন্ট সেগমেন্টটা পুরোটাই  $i-j$  এর ভিতরে আছে, সেগমেন্টটা রিলেভেন্ট।

কেস B: ( $i > e \text{ || } j < b$ ) এরকম হলে কারেন্ট সেগমেন্টটা পুরোটাই  $i-j$  এর বাইরে আছে, এই সেগমেন্টটা নেয়ার দরকার নাই।

কেস C: কেস A,B সত্য না হলে এই সেগমেন্টের কিছু অংশ  $i-j$  এর মধ্যে, সেগমেন্টটাকে আরো ভেঙে নিচে গিয়ে রিলেভেন্ট অংশটা নিতে হবে।

তাহলে আমরা কুয়েরি ফাংশনে প্রতি নোডে গিয়ে দেখবো সেগমেন্টটা রিলেভেন্ট নাকি। যদি রিলেভেন্ট হয় তাহলে সেই নোডের যোগফল রিটার্ন করবো, যদি বাইরে চলে যায় তাহলে ০ রিটার্ন করে দিবো, অন্যথায় আমরা সেগমেন্টটা আরো ভেঙে রিলেভেন্ট অংশ খুজবো।

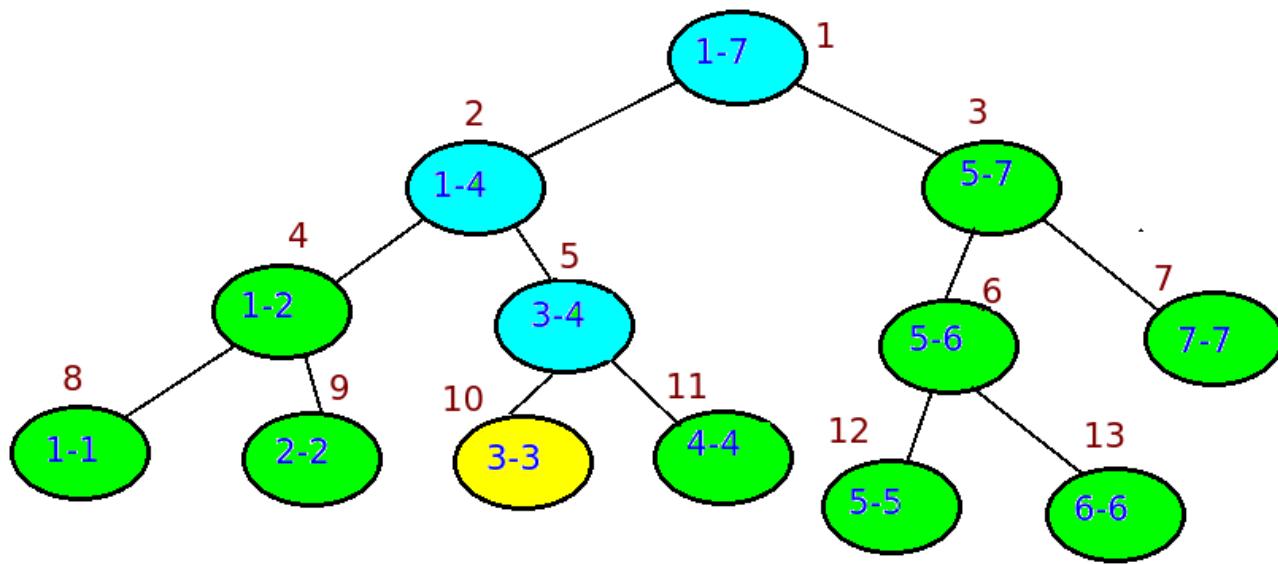
```

1 int query(int node, int b, int e, int i, int j)
2 {
3     if (i > e || j < b)
4         return 0; //বাইরে চলে গিয়েছে
5     if (b >= i && e <= j)
6         return tree[node]; //রিলেভেন্ট সেগমেন্ট
7     int Left = node * 2; //আরো ভাঁতে হবে
8     int Right = node * 2 + 1;
9     int mid = (b + e) / 2;
10    int p1 = query(Left, b, mid, i, j);
11    int p2 = query(Right, mid + 1, e, i, j);
12    return p1 + p2; //বাম এবং ডান পাশের যোগফল
13 }

```

init ফাংশনের মতোই কাজ করে কুয়েরি ফাংশনটা। i,j হলো যে রেঞ্জের যোগফল বের করতে হবে সেটা আর b,e হলো কারেন্ট নোডে যে রেঞ্জের যোগফল আছে সেটা।

সবশেষে আপডেট করা, যার জন্য কিউমুলিটিভ সাম ব্যবহার না করে ত্রি বানিয়েছি। তোমাকে বললো i=3 নম্বর ইনডেক্সের ভ্যালু x=10 করে দিতো। তারমানে ট্রি তে যেই নোডে 3-3 রেঞ্জের যোগফল আছে সেটা আপডেট করে দিবো(নিচের ছবির হলুদ নোড)। নোডটির ভ্যালু আপডেট হলে পথে যেসব নোড ছিলো(নীল নোড) সবগুলোর যোগফল বদলে যাবে, বাকি নোডগুলোর কোনো পরিবর্তন হবেনা কারণ 3 নম্বর নোড সেগুলো রেঞ্জের বাইরে।



আপডেটের কোডেও খুব বেশি পার্থক্য নেই:

```

1 void update(int node, int b, int e, int i, int newvalue)
2 {
3     if (i > e || i < b)
4         return; //বাইরে চলে গিয়েছে
5     if (b >= i && e <= i) { //রিলেভেন্ট সেগমেন্ট
6         tree[node] = newvalue; //নতুন মান বসিয়ে দিলাম
7         return;
8     }
9     int Left = node * 2; //আরো ভাঙ্গতে হবে
10    int Right = node * 2 + 1;
11    int mid = (b + e) / 2;
12    update(Left, b, mid, i, newvalue);
13    update(Right, mid + 1, e, i, newvalue);
14    tree[node] = tree[Left] + tree[Right];
15 }

```

i নম্বর ইনডেক্সে আপডেট করবো, এক্সট্রা সেগমেন্টগুলো শুরুতেই বাদ দিয়ে দিয়েছি। রিলেভেন্ট সেগমেন্টে গেলে নতুন মান বসিয়ে দিয়েছি, এইখানে কন্ডিশনটা if(b==e) লিখলেও চলতো কারণ সবসময় লিফ নোডে আপডেট করছি আমরা।

সেগমেন্ট ট্রি তো মোটামুটি এই ৩টা ফাংশন সবসময় থাকে init,query,update। অনেক সময় init এর কাজটা আপডেট দিয়ে করে ফেলা যায়। যেমন এখানে তুমি init কল করে ট্রি না বানিয়ে প্রতিটা নোড আলাদা করে আপডেট করে ট্রি বানাতে পারতে। ৩টা ফাংশন মিলিয়ে কোডটা হবে:

```

1 #define mx 100001
2 int arr[mx];
3 int tree[mx * 3];
4 void init(int node, int b, int e)
5 {
6     if (b == e) {
7         tree[node] = arr[b];
8         return;
9     }
10    int Left = node * 2;
11    int Right = node * 2 + 1;
12    int mid = (b + e) / 2;
13    init(Left, b, mid);
14    init(Right, mid + 1, e);
15    tree[node] = tree[Left] + tree[Right];
16 }
17 int query(int node, int b, int e, int i, int j)
18 {
19     if (i > e || j < b)
20         return 0; //বাইরে চলে গিয়েছে
21     if (b >= i && e <= j)
22         return tree[node]; //রিলেভেন্ট সেগমেন্ট
23     int Left = node * 2; //আরো ভাঙ্গতে হবে
24     int Right = node * 2 + 1;
25     int mid = (b + e) / 2;
26     int p1 = query(Left, b, mid, i, j);
27     int p2 = query(Right, mid + 1, e, i, j);
28     return p1 + p2; //বাম এবং ডান পাশের যোগফল
29 }
30 void update(int node, int b, int e, int i, int newvalue)
31 {

```

```

32     if (i > e || i < b)
33         return; //বাইরে চলে গিয়েছে
34     if (b >= i && e <= i) { //রিলেভেন্ট সেগমেন্ট
35         tree[node] = newvalue;
36         return;
37     }
38     int Left = node * 2; //আরো ভাঁওতে হবে
39     int Right = node * 2 + 1;
40     int mid = (b + e) / 2;
41     update(Left, b, mid, i, newvalue);
42     update(Right, mid + 1, e, i, newvalue);
43     tree[node] = tree[Left] + tree[Right];
44 }
45 int main()
46 {
47     READ("in");
48     int n;
49     cin >> n;
50     repl(i, n)
51         cin
52         >> arr[i];
53     init(1, 1, n);
54
55     update(1, 1, n, 2, 0);
56     cout << query(1, 1, n, 1, 3) << endl;
57     update(1, 1, n, 2, 2);
58     cout << query(1, 1, n, 2, 2) << endl;
59     return 0;
60 }
```

---

সেগমেন্ট ট্রি অ্যারেকে বারবার ২ভাগে ভাগ করে, ট্রি এর গভীরতা হবে সর্বোচ্চ  $\log(n)$  তাই প্রতিটা কুয়েরি আর আপডেটের কমপ্লেক্সিটি  $O(\log n)$ । `init` ফাংশনে ট্রি এর প্রতিটা নোডেই একবার যেতে হয়েছে তাই সেক্ষেত্রে কমপ্লেক্সিটি হবে প্রায়  $O(n\log n)$ ।

সেগমেন্ট ট্রি তুমি তখনই ব্যবহার করতে পারবে যখন দুইটা ছোটো সেগমেন্টকে একসাথে করে বড় সেগমেন্টের ফলাফল বের করা যায়। যোগফল ছাড়াও একটা রেঞ্জের মধ্যে সর্বোচ্চ বা সর্বনিম্ন মান তুমি বের করতে পারবে, বামপাশের সর্বোচ্চ মান এবং ডানপাশের সর্বোচ্চ মান জানলে কুট নোডেরটাও বের করা যায় খুবই সহজে।

এখানে একটা গুরুত্বপূর্ণ জিনিস বাদ পড়েছে। ধরো তোমাকে একটা ইনডেক্সে আপডেট করতে না বলে  $i$  থেকে  $j$  ইনডেক্সে আপডেট করতে বললো, তাহলে কি করবে? প্রতিটা লিফ নোডে আলাদা করে আপডেট করলে  $O(n\log n)$  হয়ে যাবে কমপ্লেক্সিটি যেটা TLE দিবে। এটার জন্য খুবই এলিগেন্ট একটা টেকনিক আছে যার নামে অলস(লেজি) প্রপাগেশন! সে সম্পর্কে পরের পর্বে জানবো, এখন তুমি যতটুকু শিখেছো সেটা দিয়ে [array queries](#) আর [Curious Robin Hood](#) প্রবলেমটা সলভ করো।

## পরের পর্ব- লেজি প্রপাগেশন

# ডাটা স্ট্রাকচার: সেগমেন্ট ট্রি-২ (লেজি প্রপাগেশন)

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

জুলাই ১৮, ২০১৩

সেগমেন্ট ট্রির সবথেকে এলিগেন্ট অংশ হলো লেজি প্রপাগেশন টেকনিক। আমরা [আগের পর্বে](#) যে সেগমেন্ট ট্রি যেভাবে আপডেট করেছি তাতে একটা বড় সমস্যা ছিলো। আমরা একটা নির্দিষ্ট ইনডেক্স আপডেট করতে পেরেছি, কিন্তু একটা রেঞ্জের মধ্যে সবগুলো ইনডেক্স আপডেট করতে গেলেই বিপদে পরে যাবো। সে কারণেই আমাদের লেজি প্রপাগেশন শিখতে হবে, প্রায় সব সেগমেন্ট ট্রি প্রবলেমেই এই টেকনিকটা কাজে লাগবে। এই পর্বটা পড়ার আগে অবশ্যই তোমাকে সেগমেন্ট ট্রির একদুটি প্রবলেম সলভ করে আসতে হবে, এছাড়া তোমার বুঝতে সমস্যা হবে। [আগের পর্বে](#) লেজি প্রপাগেশন দরকার হয়না এমন কয়েকটি প্রবলেম দিয়েছি, আগে সেগুলো সলভ করতে হবে।

তোমাকে একটি অ্যারে দেয়া আছে  $n$  সাইজের এবং তোমাকে কুয়েরি করা হচ্ছে  $i$  থেকে  $j$  ইনডেক্সের মধ্যে সবগুলো এলিমেন্টের যোগফল বলতে হবে। আর আপডেট অপারেশনে তোমাকে বলা হলো  $i$  থেকে  $j$  ইনডেক্সের মধ্যে সবগুলো সংখ্যার সাথে একটি নির্দিষ্ট সংখ্যা  $x$  যোগ করতে।

যেমন যদি অ্যারেটা শুরুতে হয়তো ছিলো এরকম:

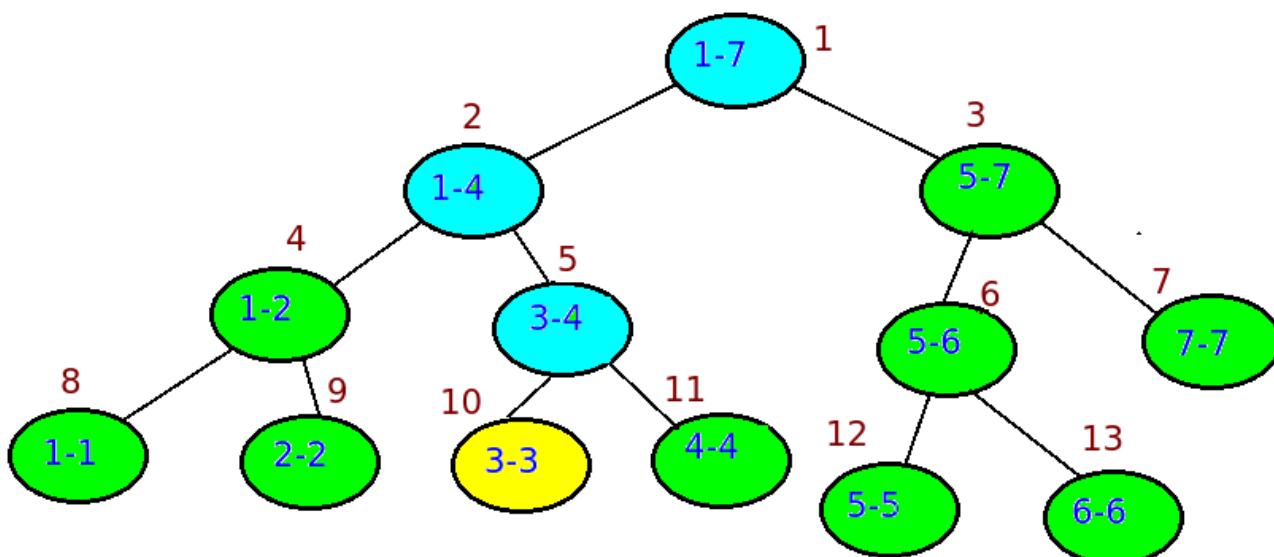
4 1 2 3 9 8 7

তাহলে  $i=3, j=5$  ইনডেক্সের মধ্যে সবগুলো সংখ্যার সাথে ২ যোগ করলে অ্যারেটা হবে:

4 1 2+2 3+2 9+2 8 7

আবার  $i=3, j=4$  ইনডেক্সের মধ্যে সবগুলো সংখ্যার যোগফল হবে  $2+2+3+2=9$ ।

আগের প্রবলেমে আমরা খালি একটা ইনডেক্স আপডেট করেছিলাম। তখন আমি শুধু ৩ নম্বর ইনডেক্সে আপডেট দেখানোর জন্য এই ছবিটা একেছিলাম:



যে নোডটি আপডেট করবো সেই নোডে পৌছানোর পথের সবগুলো নোড আপডেট হয়ে যাবে

আমরা সেগমেন্ট ট্রি এর একদম নিচে গিয়ে ৩ যেখানে আছে সেই নোডটা আপডেট করেছি এবং সেই পথের বাকিনোডগুলোকেও আপডেট করে দিয়েছি উপরে ওঠার সময়।

এখন তোমাকে  $i=1$  এবং  $j=8$  এই রেঞ্জের সবার সাথে  $x$  যোগ করতে বলা হলো। একটা সলিউশন হতে পারে তুমি আগের মতো করেই ১, ২, ৩, ৪ ইত্যাদি ইনডেক্সের আলাদা করে আপডেট করলে। তাহলে প্রতি আপডেটে কমপ্লেক্সিটি  $\log n$  এবং সর্বোচ্চ  $n$  টি আপডেটের জন্য  $n\log n$  সময় লাগবে। এটা খুব একটা ভালো সলিউশন না, আমরা এখনে  $\log n$  এই আপডেট করতে পারি।

### কিছু ডেফিনিশন:

যেকোনো ট্রি স্ট্রাকচারে লিফ নোড হলো সবথেকে নিচের নোডগুলো। লিফ ছাড়া বাকি সবনোড হলো ইন্টারনাল নোড।

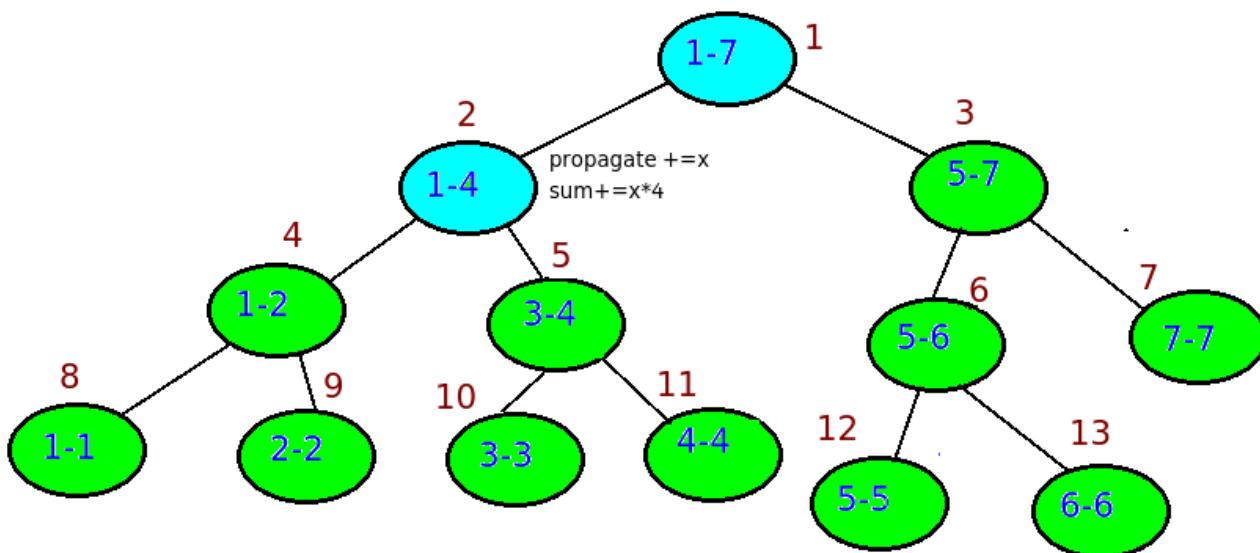
সেগমেন্ট ট্রি তে লিফ নোড গুলোতে কি থাকে? সেখানে থাকে কোনো একটি ইনডেক্সের অরিজিনাল ভ্যালু।

সেগমেন্ট ট্রি তে ইন্টারনাল(লিফ ছাড়া বাকি সব) নোডগুলোতে কি থাকে? সেখানে থাকে নিচে যতগুলো লিফ নোড আছে সবগুলোর মার্জ করা ফলাফল।

সেগমেন্ট ট্রি তে একটা নোডের রেঞ্জ হলো সেই নোডে যেসব ইনডেক্সের মার্জ করা রেজাল্ট আছে। যেমন ছবিতে ৩ নম্বর নোডের রেঞ্জ ৫ থেকে ৭।

যেমন উপরের ছবিতে ১০ নম্বর নোডে আছে ৩ নম্বর ইনডেক্সের ভ্যালু আর ২ নম্বর নোডে আছে ১, ২, ৩, ৪ নম্বর নোডের যোগফল। কি দরকার এতগুলো লিফ নোডে কষ্ট করে গিয়ে আপডেট করে আসা? তার থেকে আমরা কি ২ নম্বর নোডে এসে সেখানে এই ইনফরমেশনটা রেখে দিতে পারিনা “কারেন্ট নোডের নিচের সবগুলো ইনডেক্সের সাথে  $x$  যোগ হবে” ? তার মানে যখন দেখছি একটা নোডের রেঞ্জ যখন পুরোপুরি কুয়েরির ভিতরে থাকে তখন সেই নোডের নিচে আর না গিয়ে সেখানে প্রপাগেশন ভ্যালুটা সেভ করে রাখতে পারি। একটা নোডের প্রপাগেশন ভ্যালু হলো সেই নোডের রেঞ্জের মধ্যে সব ইনডেক্সের মধ্যে যে ভ্যালুটা যোগ হবে সেটা।

নিচের ছবিতে দেখো কিভাবে এই এক্সট্রা ইনফরমেশনটা সেভ করা হচ্ছে:



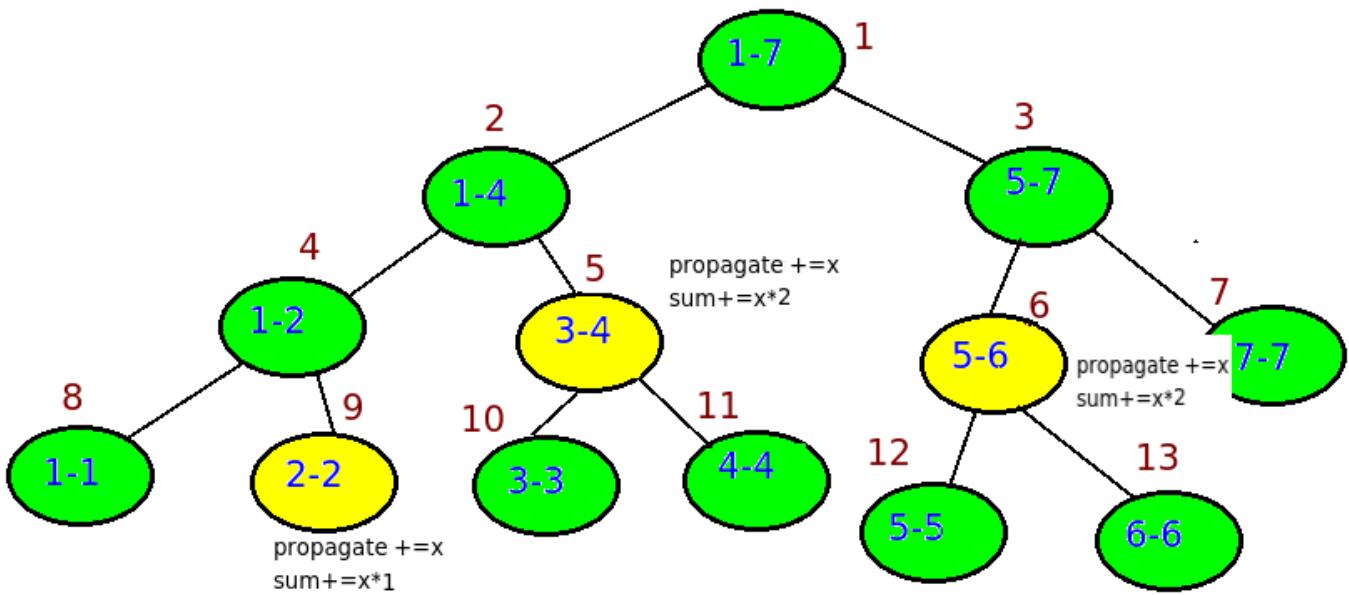
১ থেকে ৪ ইনডেক্সের সবার সাথে  $x$  যোগ করতে চাই

২ নম্বর নোডে গিয়ে বলে দিলাম নিচের সবার সাথে  $x$  যোগ হবে

২ নম্বর নোডের নিচে ৪ টি লিফ নোড আছে, সবার সাথে  $x$  যোগ করলে ২ নম্বর নোডের সাম এর সাথে  $4x$  যোগ হবে।

প্রতিটি নোডে যোগফল ছাড়াও আরেকটা ভ্যারিয়েবল রাখতে হবে যেটার নাম দিয়েছি *propagate*। এই ভ্যারিয়েবলটার কাজ হলো তার নিচের লিফ নোডগুলোর সাথে কত যোগ করতে হবে তার হিসাব রাখা। *propagate* এর মান শুরুতে থাকে শূন্য। এরপর যে রেঞ্জটা আপডেট করতে বলবে সেই রেঞ্জের “রিলেভেন্ট নোড” গুলোতে গিয়ে *propagate* এর সাথে  $x$  যোগ করে আসবো। (আগের পর্বেই জেনেছো “রিলেভেন্ট নোড” হলো যেসব নোডের রেঞ্জ পুরোপুরি কুয়েরির ভিতরে আছে)

আরেকটা উদাহরণ, ২ থেকে ৬ নম্বর নোড যদি আপডেট করতে হয় তাহলে হলুদ নোডগুলোতে গিয়ে বলে দিবো নিচের নোডগুলোর সাথে  $x$  যোগ করতে:



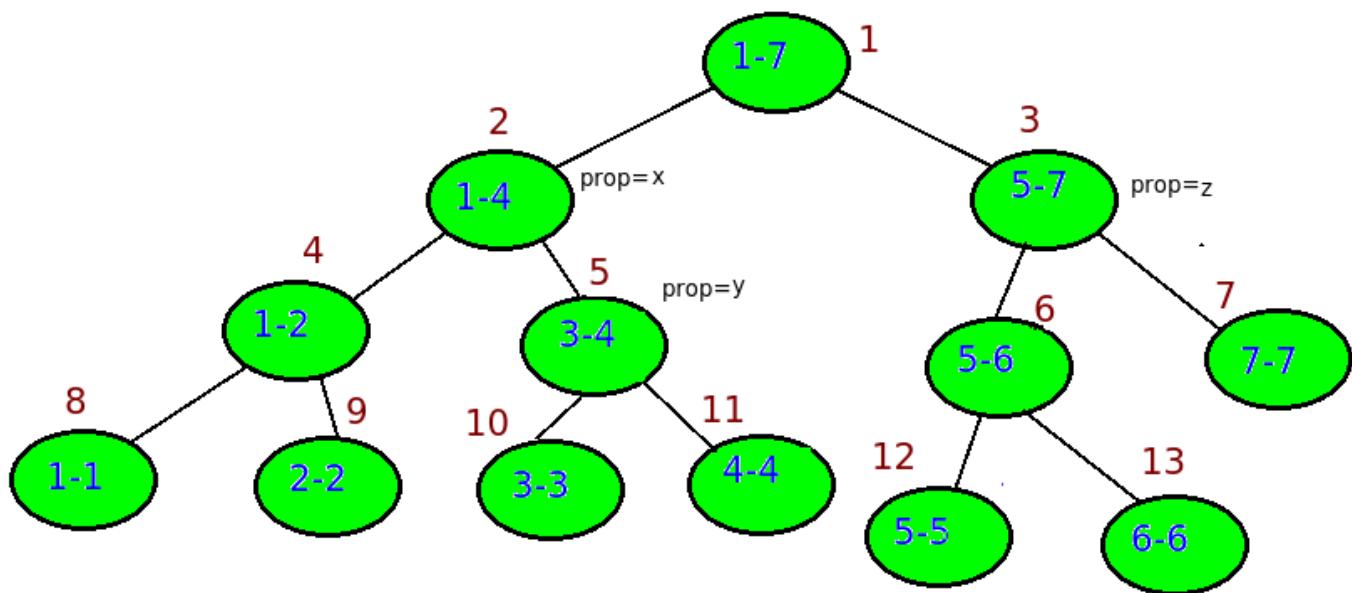
আগের আপডেট ফাংশনের মতোই কোনো একটা নোড আপডেট করার পর সেই পথের সবগুলো নোড আপডেট করে উঠতে হবে। আমরা কোডটা দেখি:

```

1 struct info {
2     i64 prop, sum;
3 } tree[mx * 3]; //sum ছাড়াও নিচে অতিরিক্ত কর যোগ হচ্ছে সেটা রাখবো prop এ
4 void update(int node, int b, int e, int i, int j, i64 x)
5 {
6     if (i > e || j < b)
7         return;
8     if (b >= i && e <= j) //নোডের রেঞ্জ আপডেটের রেঞ্জের ভিতরে
9     {
10        tree[node].sum += ((e - b + 1) * x); //নিচে নোড আছে e-b+1 টি, তাই e-b+1 বার x যোগ হবে এই রেঞ্জে
11        tree[node].prop += x; //নিচের নোডগুলোর সাথে x যোগ হবে
12        return;
13    }
14    int Left = node * 2;
15    int Right = (node * 2) + 1;
16    int mid = (b + e) / 2;
17    update(Left, b, mid, i, j, x);
18    update(Right, mid + 1, e, i, j, x);
19    tree[node].sum = tree[Left].sum + tree[Right].sum + (e - b + 1) * tree[node].prop;
20    //উপরে উঠার সময় পথের নোডগুলো আপডেট হবে
21    //বাম আর ডান পাশের সাম ছাড়াও যোগ হবে নিচে অতিরিক্ত যোগ হওয়া মান
22 }
  
```

আগের আপডেট ফাংশনের সাথে পার্থক্য হলো এখন একটা রেঞ্জে আপডেট করছি এবং কোনো নোডের রেঞ্জ আপডেট রেঞ্জের ভিতরে হলে আমরা নিচে না গিয়ে বলে দিচ্ছি যে নিচের ইনডেক্সগুলোতে x যোগ হবে।

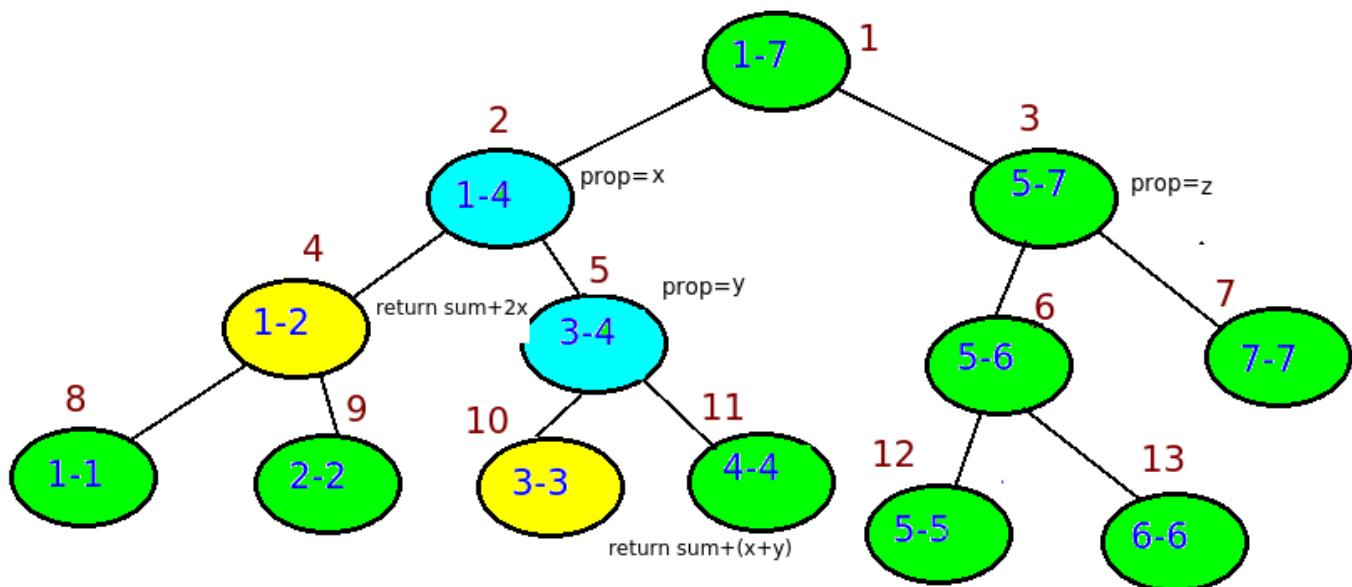
এখন মনে করো আমরা বেশ কয়েকবার আপডেট ফাংশন কল করেছি। নোডগুলোর প্রপাগেটেড ভ্যালুগুলা আপডেট হয়ে নিচের মতো হয়েছে:



যেসব নোডের ভ্যালু নিখিনি সেগুলোতে শুণ্য আছে মনে করো। তাহলে উপরের ছবির মানে হলো ১-৪ ইনডেক্সের সাথে x যোগ হবে, ৫-৭ এর সাথে z যোগ হবে ইত্যাদি।

এবার প্রশ্ন হলো কুয়েরি করবো কিভাবে?

ধরো আমাদের কুয়েরির রেঞ্জ হলো ১-৩। সাধারণভাবে আমরা আমাদের রিলেভেন্ট নোড ৪ আর ১০ থেকে ভ্যালু নিয়ে যোগ করে দিতাম। কিন্তু আমরা জানি ২ নম্বর নোডের নিচের সবার সাথে x যোগ হয়েছে, তাই ৮ নম্বর নোডের নিচের সবার সাথেও x যোগ হয়েছে! তাই আমরা ৪ নম্বর রেঞ্জে রাখা ভ্যালুর সাথে যোগ করে দিবো  $2*x$ , কারণ ৪ নম্বর নোডের রেঞ্জে ২টি ইনডেক্স আছে এবং তাদের সবার সাথে x যোগ হয়েছে। ঠিক একই ভাবে যেহেতু ২ এবং ৫ নম্বর নোডের নিচে আছে ১০, তাই ১০ নম্বর নোডের রেঞ্জে ১টি ইনডেক্স আছে এবং তার সাথে যোগ হবে  $(x+y)$ ।



তারমানে এটা পরিষ্কার যে কুয়েরি করা সময় কোনো নোডে যাবার সময় উপরের প্রপাগেটেড ভ্যালু গুলার যোগফল সাথে করে নিয়ে ঘেতে হবে। তাহলে কুয়েরির ফাংশনে carry নামের একটা প্যারামিটার যোগ করে দেই ঘার কাজ হবে ভ্যালুটা বয়ে নিয়ে যাওয়া:

```

1   int query(int node, int b, int e, int i, int j, int carry = 0)
2   {
3       if (i > e || j < b)
4           return 0;
5
6       if (b >= i and e <= j)
7           return tree[node].sum + carry * (e - b + 1); //সাম এর সাথে যোগ হবে সেই রেঞ্জের সাথে অতিরিক্ত যত
8   যোগ করতে বলেছে সেটা
9
10      int Left = node << 1;
11      int Right = (node << 1) + 1;
12      int mid = (b + e) >> 1;
13
14      int p1 = query(Left, b, mid, i, j, carry + tree[node].prop); //প্রপাগেট ভ্যালু বয়ে নিয়ে যাচ্ছে carry
15  ভ্যারিয়েবল
16      int p2 = query(Right, mid + 1, e, i, j, carry + tree[node].prop);
17
18      return p1 + p2;
19  }

```

---

আগের কোডের সাথে ডিফারেন্স হচ্ছে carry প্যারামিটারে এবং রিটার্নভ্যালুতে। শুরুতে হিসাব করে রাখা sum এর সাথে যোগ হচ্ছে অতিরিক্ত যে ভ্যালু যোগ হয়েছে সেটা।

মোটামুটি এই হলো লেজি প্রপাগেশনের কাহিনী। সামারী করলে দাঢ়ায়:

১. রেঞ্জের আপডেট O(logn) এ করতে চাইলে লেজি প্রপাগেশন ব্যবহার করতেই হবে।
২. লেজি প্রপাগেশনের কাজ হলো লিফ নোডে আপডেট না করে আগেই কোনো নোড আপডেট রেঞ্জের ভিতরে পড়লে সেখানে বলে দেয়া নিচের ইনডেক্সগুলো কিভাবে আপডেট হবে।
৩. কুয়েরি করার সময় উপরের নোডগুলোতে সেভ করা প্রপাগেশন ভ্যালুগুলো রিলেভেন্ট নোড ক্যারি করে নিয়ে আসতে হবে এবং সেই অনুযায়ী ভ্যালু রিটার্ন করতে হবে।

অনেক সময় প্রবলেমে বলতে পারে একটা রেঞ্জের মধ্যে সবগুলো সংখ্যাকে  $x$  দিয়ে বদলে দিতে ( $x$  যোগ নয়), সেক্ষেত্রে প্রপাগেশন ভ্যালু হিসাবে রাখতে হবে নিচের সবনোডকে কোন ভ্যালু দিয়ে বদলে দিতে হবে সেটা। কুয়েরি করার সময় carry ভ্যালুতেও সামান্য পরিবর্তন আসবে, কিরকম পরিবর্তন আসবে সেটা বের করার দায়িত্ব তোমার উপরে ছেড়ে দিলাম।

সেগমেন্ট ট্রির মূল টিউটোরিয়াল এখানেই শেষ। পরবর্তী কোনো পর্বে ইডি সেগমেন্ট ট্রি এবং কিছু প্রবলেম নিয়ে আলোচনা করবো। টিউটোরিয়ালের কোনো অংশ বুঝতে সমস্যা হলে ইমেইলে যোগাযোগ করতে পারো বা আরো ভালো হয় মন্তব্য অংশে সেটা জানালে।

প্র্যাকটিস প্রবলেম:

**HORRIBLE**

[Lightoj Segment Tree Section](#)

আগের পর্ব

# অ্যারে কম্প্রেশন

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

12/31/2012

খুবই সহজ কিন্তু দরকারি একটা টেকনিক নিয়ে আলোচনা করবো আজকে। STL এর ম্যাপ ব্যবহার করে আমরা অ্যারে কম্প্রেশন করবো। মনে করো কোনো একটা প্রবলেমে তোমাকে বলা হলো একটি অ্যারেতে ১ লাখটা সংখ্যা দেয়া থাকবে যাদের মান হবে ০ থেকে সর্বোচ্চ ১০০০। এখন যেকোনো আমি একটি সংখ্যা বললে তোমাকে বলতে হবে অ্যারের কোন কোন পজিশনে সংখ্যাটি আছে। যেমন মনে করো ইনপুট অ্যারেটা হলো:

```
| 100 252 108 512
```

খুবই সহজ সলিউশন হলো একটি ভেক্টর নেয়া। তম পজিশনে x সংখ্যাটি পেলে ভেক্টরের x তম পজিশনে পুশ করে রাখবে। তাহলে ইনপুট নেবার পর ভেক্টরগুলোর চেহারা হবে:

```
[0]->1 2 7
[1]->0 6 10
[2]->3 5 11
[3]->empty
[4]->8
[5]->4 9
```

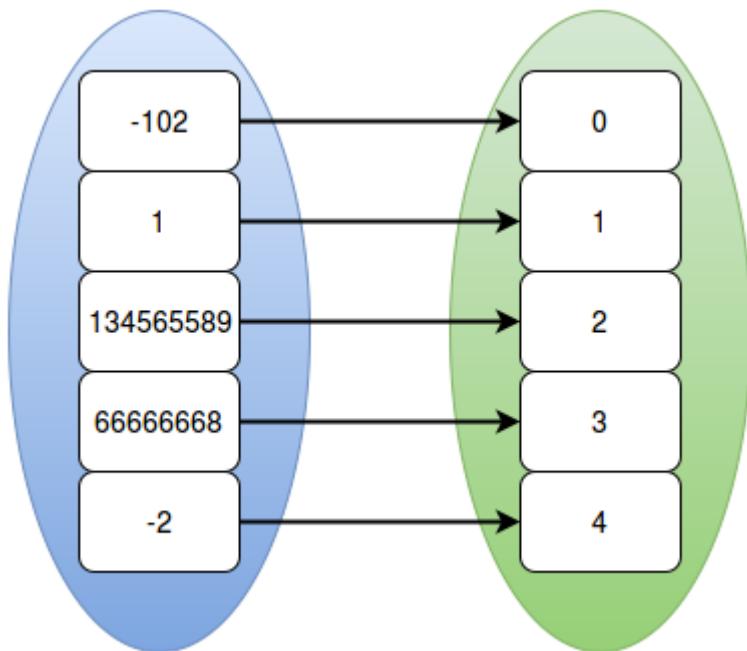
কোন সংখ্যা কোন কোন পজিশনে আছে তুমি পেয়ে গেলে। এখন যদি বলে 2 কোথায় কোথায় আছে তাহলে তুমি ২ নম্বর ইনডেক্সে লুপ চালিয়ে 3,5,11 প্রিন্ট করে দাও। 1001 সাইজের ২-ডি ভেক্টর দিয়েই কাজ হয়ে যাবে।

এখন তোমাকে বলা হলো সংখ্যাগুলো নেগেটিভও হতে পারে এবং সর্বোচ্চ  $2^{30}$  পর্যন্ত হতে পারে। এবার কি এই পদ্ধতি কাজ করবে? একটি সংখ্যা -100 বা  $2^{30}$  হলে তুমি সেটার পজিশনগুলা কোন ইনডেক্সে রাখবে? অবশ্যই তুমি এতো বড় ভেক্টর ডিক্রেয়ার করতে পারবেনা অথবা নেগেটিভ ইনডেক্স ব্যবহার করতে পারবেনা। ধরো ইনপুট এবার এরকম:

```
| input[]={-102,1,134565589,134565589,-102,66666668,134565589,66666668,-102,1,-2}
```

একটা ব্যাপার লক্ষ্য করো, সংখ্যা দেয়া হবে সর্বোচ্চ  $10^5$  বা ১ লাখটা। তাহলে অ্যারেতে মোট ভিন্ন ভিন্ন সংখ্যা হতে পারে কষটা? অবশ্যই সর্বোচ্চ ১ লাখটা। আমরা প্রতিটি ভিন্ন ভিন্ন সংখ্যাকে ছোটো কিছু সংখ্যার সাথে one-to-one ম্যাপিং করে ফেলবো। উপরের অ্যারেতে ভিন্ন ভিন্ন সংখ্যা গুলো হলো:

```
| -102,1,134565589,66666668,-2
```



দুইটা উপায় আছে ম্যাপিং করার। একটা হলো STL এর map ব্যবহার করে। map এর কাজ হলো যেকোনো একটি সংখ্যা, স্ট্রিং বা অবজেক্টকে অন্য একটি মান অ্যাসাইন করা যেটা আমরা উপরের ছবিতে করেছি। আমাদের এই প্রবলেমে কোন সংখ্যা কার থেকে বড় বা ছোটো সেটা দরকার নাই তাই সর্ট না করেই ম্যাপিং করে দিতে পারি। নিচের কোডটা দেখো:

C++

```

1 void compress()
2 {
3     map<int,int>mymap;
4     int input[]={-102,1,134565589,134565589,-102,66666668,134565589,66666668,-102,1,-2};
5     int assign=0,compressed[100],c=0,n[sizeof(input)/sizeof(int)]; //array size;
6     for(int i=0;i<n;i++){
7         int x=input[i];
8         if(mymap.find(x)==mymap.end()){//x not yet compressed
9             mymap[x]=assign;
10            printf("Mapping %d with %d\n",x,assign);
11            assign++;
12        }
13        x=mymap[x];
14        compressed[ c++ ]=x;
15    }
16    printf("Compressed array: ");
17    for(int i=0;i<n;i++) printf("%d ",compressed[i]);puts("");
18 }
```

কোডের আউটপুট হবে:

```

Mapping -102 with 0
Mapping 1 with 1
Mapping 134565589 with 2
Mapping 66666668 with 3
Mapping -2 with 4
Compressed array: 0 1 2 2 0 3 2 3 0 1 4
```

আমরা ইনপুট অ্যারেতে যখনই একটু নতুন সংখ্যা পাচ্ছি সেটাকে একটা ভ্যালু অ্যাসাইন করে দিচ্ছি এবং আসল ভ্যালুকে ম্যাপের ভ্যালু দিয়ে রিপ্লেস করে আরেকটি অ্যারেতে রেখে দিয়েছি। এরপরে আমরা নতুন অ্যারেটা ব্যবহার করে প্রবলেমটা সলভ করতে পারবো। কুয়েরিতে যদি বলে -2 কোথায় কোথায় আছে বের করো তাহলে তুমি আসলে 4 কোথায় কোথায় আছে বের করবে কারণ `mymap[-2]=4`।

অনেক সময় কম্প্রেস করার পরেও কোন সংখ্যাটি কার থেকে বড় এটা দরকার হয়। যেমন তোমাকে বলতে পারে যে একটি ভ্যালু নেয়ার পর ছোটো কোনো ভ্যালু আর নিতে পারবেনা। এই তথ্যটা কিভাবে রাখবে উপরের পদ্ধতিতে ১ এর থেকে -2 এ অ্যাসাইন করা ভ্যালু বড় হয়ে গিয়েছে। যদি তুমি সেটা না চাও তাহলে আগে ভিন্ন ভিন্ন ভ্যালুগুলো আরেকটা অ্যারেতে সর্ট করে নাও।

```
sorted[] = {-102, -2, 1, 66666668, 134565589}
```

এইবার sorted অ্যারেটা ব্যবহার করে ম্যাপিং করে ফেলো। এ ক্ষেত্রে আসলে যে সংখ্যাটি sorted অ্যারেতে যে পজিশনে আছে সেটাই হবে তার ম্যাপ করা ভ্যালু।

এটা যদি বুঝতে পারো তাহলে নিচয়ই ম্যাপ ছাড়াই বাইনারি সার্চ করে তুমি কম্প্রেস করে ফেলতে পারবে অ্যারেটাকে। ইনপুট অ্যারের উপর লুপ চালাও, প্রতিটি x এর জন্য দেখো sorted অ্যারেতে x এর অবস্থান কোথায়। সেই মানটা তুমি আরেকটা অ্যারেতে রেখে দাও:

```
input[] = {-102, 1, 134565589, 134565589, -102, 66666668, 134565589, 66666668, -102, 1, -2}
compressed_input[] = {0, 2, 4, 4, 0, 3, 4, 3, 0, 2, 1}
```

কুয়েরির সময়ও বাইনারী সার্চ করে কম্প্রেস করার মানটা বের করে কাজ করতে হবে।

তুমি ম্যাপ বা বাইনারি সার্চ যেভাবে ইচ্ছা কম্প্রেস করতে পারো। প্রতিবার map এক্সেস করতে logn কম্প্লেক্সিটি লাগে যেটা বাইনারী সার্চের সমান। stl এ বিভিন্ন class, ডাইনামিক মেমরির ব্যবহারের জন্য map কিছুটা স্লো, তবে টাইম লিমিট খুব tight না হলে খুব একটা সমস্যা হবেনা। map ব্যবহার করলে কোডিং সহজ হয়।

কোনো কোনো গ্রাফ প্রবলেমে নোডগুলো আর এজগুলো কে string হিসাবে ইনপুট দেয়। যেমন ধরো গ্রাফের তিটা এজ হলো:

```
3
BAN AUS
AUS SRI
SRI BAN
```

map এ string কেও integer দিয়ে ম্যাপিং করা যায়। এই সুবিধাটা ব্যবহার করে প্রতিটি নোডকে একটি ভ্যালু অ্যাসাইন করবো:

C++

```

1 map<string,int>mymap;
2 int edge,assign=0;
3 cin>>edge;
4 for(int i=0;i<edge;i++)
5 {
6     char s1[100],s2[100];
7     cin>>s1>>s2;
8     if(mymap.find(s1)==mymap.end()){
9         printf("Mapping %s with %d\n",s1,assign);
10        mymap[s1]=assign++;
11    }
12    if(mymap.find(s2)==mymap.end()) {
13        printf("Mapping %s with %d\n",s2,assign);
14        mymap[s2]=assign++;
15    }
16    int u=mymap[s1];
17    int v=mymap[s2];
18    cout<<"Edge: "<<u<<" "<<v<<endl;
19 }
```

অনেক সময় বলা হতে পারে নোডগুলোর মান হবে  $-2^{30}$  থেকে  $2^{30}$  পর্যন্ত কিন্তু সর্বোচ্চ নোড হবে ১০০০০ টা, তখন আমরা নোডগুলোকে একই ভাবে ছোটো ভ্যালু দিয়ে ম্যাপিং করে ফেলবো।

2-d grid ও কমপ্রেস করে ফেলা যায় অনেকটা এভাবে। সেটা নিয়ে আরেকদিন আলোচনা করতে চেষ্টা করবো, তবে নিজে একটু চিন্তা করলেই বের করতে পারবে।

প্রবলেম:

[Drunk](#)

[Babel](#)

[A Node Too Far](#)

# লোয়েস্ট কমন অ্যানসেস্টর

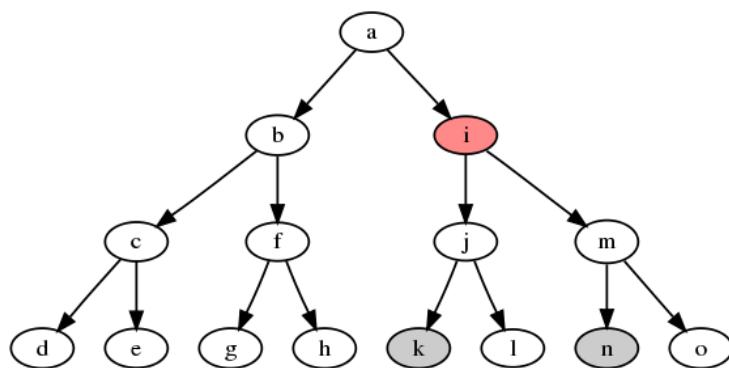
 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

মার্চ ১৩, ২০১৮

লোয়েস্ট কমন অ্যানসেস্টর জিনিসটা শুনতে একটু কঠিন মনে হলেও জিনিসটা সহজ আর খুবই কাজের। বেশ কিছু ধরণের প্রবলেম সলভ করে ফেলা যায় এই অ্যালগোরিদম দিয়ে। আমরা প্রথমে লোয়েস্ট কমন অ্যানসেস্টর বের করার ব্রুটফোর্স অ্যালগোরিদম দেখবো, তারপর স্পার্স টেবিল নামের নতুন একটা ডাটা স্ট্রাকচার শিখে কমপ্লেক্সিটি লগ এ নামিয়ে আনবো।

প্রথমেই আমরা জেনে নেই লোয়েস্ট কমন অ্যানসেস্টর বা এল.সি.এ(LCA) কি সেটা। নিচের ছবিটা দেখ:



ছবিতে k আর n নোডের প্যারেন্ট ধরে পিছে ঘেতে থাকলে তারা i নোডে এসে মিলবে। i হলো k,n এর লোয়েস্ট কমন অ্যানসেস্টর। 'a' ও দুইজনের কমন অ্যানসেস্টর কিন্তু i হলো 'লোয়েস্ট' বা সবথেকে কাছাকাছি।

মনে করো ট্রি টা দিয়ে বুঝানো হচ্ছে একটা অফিসে কে কার সুপারভাইজর বা বস। a হলো অফিসের হেড, b এবং i এর বস হলো a, আবার c এবং f এর বস হলো b ইত্যাদি। তাহলে লোয়েস্ট কমন অ্যানসেস্টর অ্যালগোরিদম দিয়ে দুইজন এমপ্লায়ির লোয়েস্ট/সব থেকে কাছের কমন বস খুজে বের করে ফেলতে পারি সহজেই।

প্রথমে আমরা একদম নেইভ একটা উপায় দেখি যেটা খুব বড় ইনপুটের জন্য কাজ করবেনা। মনে করি আমরা k,n এর এল.সি.এ বের করতে চাই, প্রথমেই k এর প্যারেন্ট ধরে উপরে উঠে সবগুলো অ্যানসেস্টরের লিস্ট করে ফেলি:

k এর অ্যানসেস্টর:

|   |   |   |
|---|---|---|
| j | i | a |
|---|---|---|

n এর অ্যানসেস্টর:

|   |   |   |
|---|---|---|
| m | i | a |
|---|---|---|

দুইটা লিস্টেই বাম থেকে ডানে যেতে থাকলে প্রথম যে দুইটা নোড কমন পাবো সেটাই হবে এল.সি.এ। এই উদাহরণে প্রথমে কমন পাবো  $i$ , তাই সেটাই হবে এল.সি.এ। 'a' ও একটা কমন এনসেস্টর কিন্তু সেটা "লোয়েস্ট" না। একটা ভিজিটেড অ্যারে বা ফ্ল্যাগ ব্যবহার করে খুব সহজেই আমরা এই অ্যালগোরিদম ইমপ্লিমেন্ট করতে পারি। প্রথমে  $k$  এর লিস্টে ঘারা আছে সেগুলোর ইনডেক্সের ফ্ল্যাগ অন করে দিবো, তারপর  $n$  এর লিস্টে লুপ চালিয়ে দেখবো ফ্ল্যাগ অন আছে কোন নোডের।

প্রতিটা লিস্ট বানাতে আমাদের সর্বোচ্চ  $n$  টা নোড ভিজিট করা লাগতে পারে, তাই কমপ্লেক্সিটি  $O(n)$ ।

$O(n)$  কমপ্লেক্সিটি খারাপ না যদি আমাদের মাত্র এক জোড়া নোডের এল.সি.এ বের করতে হয়। যদি আমাদের  $m$  জোড়া নোড দিয়ে বলে প্রতিটা পেয়ারের এল.সি.এ বের করতে তাহলে কমপ্লেক্সিটি হয়ে যাবে  $O(m*n)$ । এটাকে আমরা চাইলে  $O(m*\log(n))$  এ সলভ করতে পারি, অর্থাৎ প্রতি কুয়েরি কাজ করবে  $O(\log n)$  এ। তবে তার আগে কিছু প্রি-প্রসেসিং করে নিতে হবে।

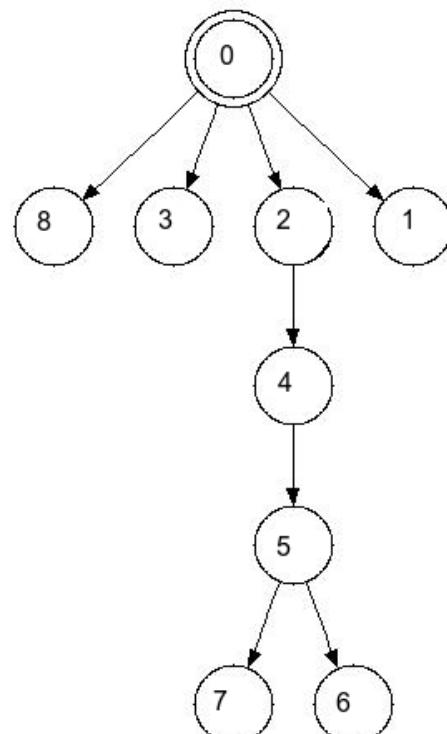
প্রতিটা নোডের জন্য আমরা সেভ করে রাখবো নোডটার:

- | প্রথম প্যারেন্ট বা  $2^0$  তম প্যারেন্ট
- | ২য় প্যারেন্ট বা  $2^1$  তম প্যারেন্ট
- | ৪থ প্যারেন্ট বা  $2^2$  তম প্যারেন্ট
- | ৮ম প্যারেন্ট বা  $2^3$  তম প্যারেন্ট

যদি  $K$ -তম প্যারেন্ট বলতে কিছু না থাকে তাহলে আমরা  $-1$  রেখে দিবো। যেমন রুট নোডের প্রথম প্যারেন্ট বলতে কিছু নেই, তাই প্রথম প্যারেন্ট হবে  $-1$ ।

তাহলে উপরের ট্রি এর জন্য নিচের মতো একটা 2-ডিমেনশনাল টেবিল তৈরি হবে:

| নোড | $2^0$ তম<br>প্যারেন্ট | $2^1$ তম<br>প্যারেন্ট | $2^2$ তম<br>প্যারেন্ট |
|-----|-----------------------|-----------------------|-----------------------|
| ০   | $-1$                  | $-1$                  | $-1$                  |
| ১   | ০                     | $-1$                  | $-1$                  |
| ২   | ০                     | $-1$                  | $-1$                  |



|   |   |    |    |
|---|---|----|----|
| ৩ | ০ | -১ | -১ |
| ৮ | ২ | ০  | -১ |
| ৫ | ৮ | ২  | -১ |
| ৬ | ৫ | ৮  | ০  |
| ৭ | ৫ | ৮  | ০  |
| ৮ | ০ | -১ | -১ |

তাহলে উপরের ট্রি এর জন্য নিচের মতো একটা ২-ডিমেনশনাল টেবিল তৈরি হবে:

এই টেবিলটাকে বলে **স্পোর্স টেবিল**। এই টেবিলটা তৈরি করে কি লাভ হলো? এখন তুমি কোনো নোডের  $k$  তম প্যারেন্ট খুব সহজেই খুজে বের করতে পারবে। যেমন কারো  $2^5$  তম প্যারেন্ট দরকার হলে প্রথমে  $2^5$  এর নিচে ২ এর সবথেকে বড় পাওয়ার  $2^{4+1}=16$  তম প্যারেন্ট খুজে বের করবে। তারপর  $16$  তম প্যারেন্টের ( $2^5-16$ )= $9$  তম প্যারেন্ট খুজে বের করবে একই পদ্ধতিতে! প্রতিটা সংখ্যাকে ২ এর পাওয়ারের যোগফল হিসাবে প্রকাশ করা যায় সেটারই সুবিধা নিচ্ছি আমরা। এভাবে করে  $O(\log n)$  কমপ্লেক্সিটিতে আমরা কারো  $k$  তম প্যারেন্ট বের করতে পারবো যেটা পরবর্তীতে LCA বের করতে কাজে লাগবে।

এখন কথা হলো উপরের টেবিলটা বানাবো কিভাবে?  $2^{10}$  তম প্যারেন্ট বের করা সহজ, ট্রি এর উপর একটা ডেপথ-ফার্স্ট-সার্চ বা ব্রেথড-ফার্স্ট-সার্চ চালিয়ে একটা অ্যারেতে সেভ করে রাখবো কার প্যারেন্ট কে। মনে করি অ্যারেটা হলো  $T$ । তাহলে উপরের ট্রি এর জন্য পাবো  $T[0]=-1$ ,  $T[1]=0$ ,  $T[6]=5$  ইত্যাদি।

এবার আমরা কলাম-বাই-কলাম টেবিলটা ভরাট করবো। প্রথমেই প্রথম কলাম ভরাট করবো যেটা আসলে  $T$  অ্যারের সমান।

টেবিলটা নাম  $P$  হলে প্রথম কলাম ভরাট করবো এভাবে,

```

1   for (i = 0; i < N; i++)
2       P[i][0] = T[i];

```

এখন চিন্তা করে দেখো একটা নোডের  $2^8=16$  তম প্যারেন্ট হলো নোডটার  $2^{10}=8$  ম প্যারেন্টের  $2^{10}=8$  ম প্যারেন্ট। তাহলে কোন নোডের  $2^j$  তম প্যারেন্ট হবে নোডটার  $2^{(j-1)}$  তম প্যারেন্টের  $2^{(j-1)}$  তম প্যারেন্ট! তাহলে যেকোন নোড  $i$  এর জন্য:

$$P[i][j] = P[P[i][j-1]][j-1] \text{ যেখানে } P[i][j-1] \text{ হলো } j-1 \text{ তম প্যারেন্ট।}$$

তাহলে পুরো টেবিলটা ভরাট করে ফেলতে পারি এভাবে:

```

1   for (j = 1; (1 << j) < N; j++)
2       for (i = 0; i < N; i++)
3           if (P[i][j - 1] != -1)
4               P[i][j] = P[P[i][j - 1]][j - 1];

```

$2^j$  এর মান যদি  $N$  এর ছোট না হয় তাহলে নতুন কোনো প্যারেন্ট পাবার সম্ভাবনা নেই, তাই লুপ চলবে ( $1 << j$ ) বা  $2^j$  ঘতক্ষণ  $N$  এর ছোট হবে।

এখন আমরা দুটি নোডের এল.সি.এ বের করবো। আমাদের আরেকটা অতিরিক্ত অ্যারে লাগবে  $L[]$  যেখানে থাকবে প্রতিটি নোডের ডেপথ বা লেভেল। ঝট নোডের লেভেল হবে ০।

এখন আমরা হলুদ নোড দুইটার এল.সি.এ বের করতে চাই:

প্রথম কাজ হবে এদেরকে একই লেভেলে নিয়ে আসা।  $p$  এর লেভেল থেকে ২ এর পাওয়ার বিয়োগ করতে থাকবো এবং একই সাথে স্পার্স টেবিল ব্যবহার করে উপরে উঠাতে থাকবো ঘতক্ষণা  $q$  এর লেভেলের সমান হয়।

```

1   for (i = log; i >= 0; i--)
2       if (L[p] - (1 << i) >= L[q]) //  $2^i$  তম
3   প্যারেন্টে যাও
    p = P[p][i];

```

এখন দুইজন সমান লেভেলে আসলো। এবার কাজ হবে দুইটা নোডকেও টেনে একসাথে উপরে উঠানো ঘতক্ষণা নোড দুটির প্যারেন্ট একসাথে এসে মিলে:

কোডটা হবে এরকম:

```

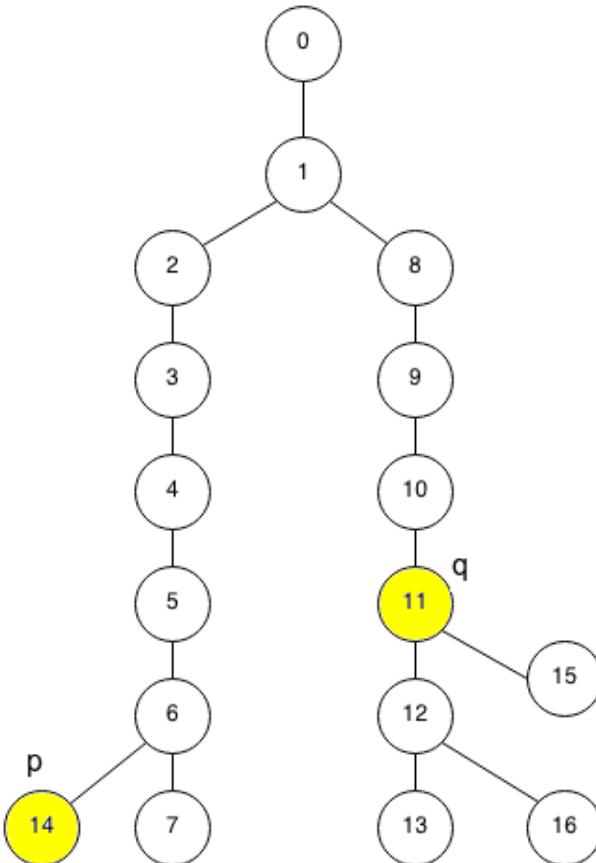
1   for (i = log; i >= 0; i--)
2       if (P[p][i] != -1 && P[p][i] != P[q][i])
3           p = P[p][i], q = P[q][i];

```

আমরা এমন জায়গায়  $p, q$  কে আনছি যেখানে তাদের দুইজনেরই প্যারেন্ট একই। তাহলে এখন  $p$  বা  $q$  এর প্যারেন্টই হবে এল.সি.এ।

C++

```
1   return T[p];
```



আমরা যদি লুপের মধ্যেই ২ আর ৮ নাস্তার নোডকে  $p, q$  তে না এনে ডিবেক্ট ১ নম্বর নোডে আনার চেষ্টা করতাম তাহলে কি ঘটতে পারতো?  $P[p][i] \neq P[q][i]$  এই শর্তটা না থাকলে কি হত? এটা চিন্তা করা তোমার কাজ 😊।

## কমপ্লেক্সিটি:

স্পার্স টেবিলে রো থাকবে  $n$  টা, প্রতিটা রো তে  $\log n$  টা কলাম থাকবে, প্রি-প্রোসেসিং কমপ্লেক্সিটি  $O(n \log n)$ ।

প্রতি কুয়েরিতে কমপ্লেক্সিটি  $O(\log n)$

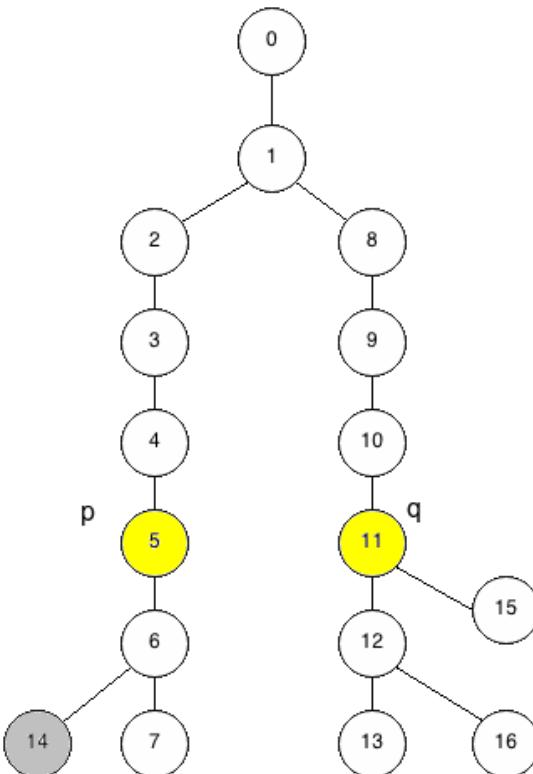
## রিলেটেড প্রবলেম:

১. একটা ওয়েটেড ট্রি দেয়া আছে। দুটি নোড দিয়ে বলা হলো তাদের মধ্যের দূরত্ব বের করতে হবে। ([Spoj QTree](#))

২. একটা ওয়েটেড ট্রি দেয়া আছে। দুটি নোড দিয়ে বলা হলো তাদের মধ্যকার পাথের সর্বোচ্চ ওয়েটের এজ এর মান বলতে হবে। ([LOJ A Secret Mission](#))

সলিউশন ১: শুরুতে ডিএফস চালিয়ে রুট থেকে প্রতিটা নোডের দূরত্ব সেভ করে রাখো। এখন নোড দুইটা  $a, b$  এবং তাদের কমন অ্যানসেস্টর  $L$  হলে সলিউশন হবে  $\text{dist}(\text{root}, a) + \text{dist}(\text{root}, b) - 2 * \text{dist}(\text{root}, L)$ ।

সলিউশন ২: স্পার্স টেবিলে অতিরিক্ত একটা ইনফরমেশন রাখতে হবে। প্রতিটা নোড থেকে  $2^0, 2^1, 2^2$  ইত্যাদি তম প্যারেন্টের জন্য, প্যারেন্ট থেকে ওই নোডের মধ্যকার পাথের সর্বোচ্চ ওয়েটের এজের মান সেভ করে রাখতে হবে। বিস্তারিত লিখলাম, এটা সলভ করা তোমার কাজ।



## আরো কিছু প্রবলেম:

[TJU Closest Common Ancestors](#)

[UVA Flea Circus](#)

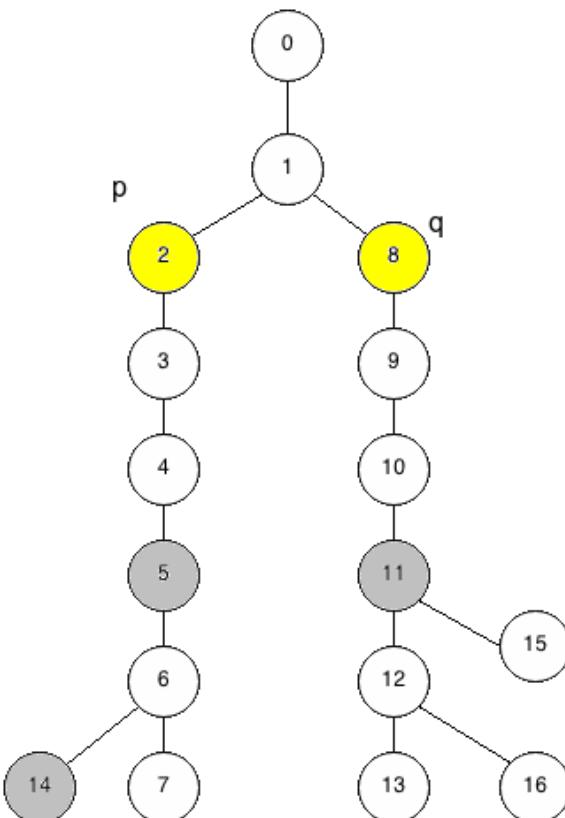
[LightOJ LCA Category](#)

## রিসোর্স:

এল.সি.এ বের করার আরো কিছু পদ্ধতি আছে যেগুলো পাবে [এখানে](#)। এগুলো ছাড়াও টারজানের একটা [অফলাইন অ্যালগোরিদম](#) আছে যেটা আগে সব কুয়েরি ইনপুট নিয়ে একটি ডিএফস চালিয়ে সবগুলো পেয়ারের এল.সি.এ বের করে দেয়।

## সম্পূর্ণ কোড:

ভেক্টর  $g$  তে ট্রি সেভ করতে হবে। প্রথমে  $\text{dfs}$  ফাংশন কল করে লেভেল, প্যারেন্ট ফিল্ড করে তারপর  $\text{lca\_init}$  কল করে প্রিপ্রসেসিং করতে হবে।




---

```
//LCA using sparse table
```

---

```
//Complexity: O(NlgN,lgN)
```

---

```
#define mx 100002
```

---

```
int L[mx]; //লেভেল
```

```
int P[mx][22]; //পার্স টেবিল
```

```
int T[mx]; //প্যারেন্ট
```

```
vector<int>g[mx];
```

```
void dfs(int from,int u,int dep)
```

```
{
```

```
T[u]=from;
```

```
L[u]=dep;
```

```
for(int i=0;i<(int)g[u].size();i++)
```

```
{
```

```
int v=g[u][i];
```

```
if(v==from) continue;
```

```
dfs(u,v,dep+1);
```

```
}
```

```
}
```

```
int lca_query(int N, int p, int q) //N=নোড সংখ্যা
```

```
{
```

```
int tmp, log, i;
```

```
if (L[p] < L[q])
```

```
tmp = p, p = q, q = tmp;
```

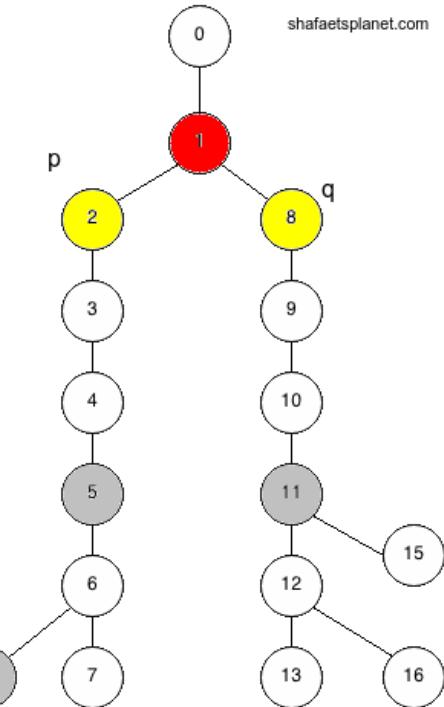
```
log=1;
```

```
while(1) {
```

```
int next=log+1;
```

```
if((1<L[p]))break;
```

```
log++;
```



```

    }

```

```

for (i = log; i >= 0; i--)

```

```

if (L[p] - (1 << i) >= L[q])

```

```

p = P[p][i];

```

```

if (p == q)

```

```

return p;

```

```

for (i = log; i >= 0; i--)

```

```

if (P[p][i] != -1 && P[p][i] != P[q][i])

```

```

p = P[p][i], q = P[q][i];

```

```

return T[p];

```

```

}

```

```

void lca_init(int N)

```

```

{

```

```

memset (P,-1,sizeof(P)); //শুরুতে সবগুলো ঘরে
-1 থাকবে

```

```

int i, j;

```

```

for (i = 0; i < N; i++)

```

```

P[i][0] = T[i];

```

```

for (j = 1; 1 << j < N; j++)

```

```

for (i = 0; i < N; i++)

```

```

if (P[i][j - 1] != -1)

```

```

P[i][j] = P[P[i][j - 1]][j - 1];

```

```
}
```

```
int main(void) {
```

```
    g[0].pb(1);
```

```
    g[0].pb(2);
```

```
    g[2].pb(3);
```

```
    g[2].pb(4);
```

```
    dfs(0, 0, 0);
```

```
    lca_init(5);
```

```
    printf( "%d\n", lca_query(5,3,4) );
```

```
    return 0;
```

```
}
```

[view raw LCA](#) hosted with ❤ by [GitHub](#)

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাটা স্ট্রাকচার: বাইনারি ইনডেক্সড ট্রি

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট ১৯, ২০১৮

বাইনারি ইনডেক্সড ট্রি খুবই চমৎকার একটা **ডাটা স্ট্রাকচার** যার সবথেকে ভালো দিক হলো কয়েক মিনিটেই কোড লিখে ফেলা যায়। আর খারাপ দিক হলো ভিতরে কি হচ্ছে বুঝতে একটু কষ্ট হয়, তবে তুমি লেখাটা মনযোগ দিয়ে পড়লে আমি নিশ্চিত কোন সমস্যা হবে না। বাইনারি ইনডেক্সড ট্রি বা BIT এর আরেক নাম হলো ফেনউইক(fenwick) ট্রি। এই লেখাটা পড়ার আগে তোমাকে বিটওয়াইজ অপারেশনগুলো(AND,OR ইত্যাদি) সম্পর্কে ভালো জানতে হবে।

আমরা একটা সহজ সমস্যা সমাধান করতে করতে বাইনারি ইনডেক্সড ট্রি সম্পর্কে জানব। এই লেখায় আমরা ধরে নিব অ্যারের ইনডেক্স ১ থেকে শুরু, BIT কিভাবে কাজ করে জানার পরে বুঝতে পারবে এটা কেন গুরুত্বপূর্ণ। তোমাকে একটা অ্যারে দেয়া হলো যার প্রতিটি পজিশনে ০ আছে। এখন তোমাকে অনেকগুলো অপারেশন দেয়া হবে। প্রতিটা অপারেশন হতে পারে নিচের যেকোন একটা:

\* ii তম ইনডেক্সে xx সংখ্যাটা যোগ কর।

\* 11 থেকেii তম ইনডেক্সের সাবঅ্যারের মোট যোগফল বল।

একদম ‘নেইভ(Naive)’ উপায়ে করলে আমরা  $O(1)O(1)$  এ প্রতি পজিশনে xx যোগ করব আর লুপ চালিয়ে  $O(n)O(n)$  এ যোগফল বের করব। BIT দিয়ে যোগফল  $O(\log n)$  এ বের করা সম্ভব, তবে সেক্ষেত্রে আপডেট অপারেশনও  $O(\log n)O(\log n)$  এ করতে হবে।

আমরা জানি প্রতিটা সংখ্যাকেই কিছু ২ এর পাওয়ারের যোগফল হিসাবে লেখা যায়। যেমন ১৩ এর বাইনারি হলো “১১০১”, বাইনারিতে ১ আছে ০তম, ২য় এবং ৩য় অবস্থানে। তাহলে ১৩ কে লেখা যায়  $2^0+2^2+2^3$  বা  $1+4+8$ । বাইনারি ইনডেক্স ট্রি তে আমরা এই প্রোপার্টি ব্যবহার করে কিছু কাজ করব। যে সংখ্যাটা অ্যারেতে যোগ করছি সেটাকে নয় বরং আমরা অ্যারের ইনডেক্সগুলোকে ২ এর পাওয়ারে যোগফল হিসাবে লিখে কিছু কাজ করব।

11 থেকে ii পর্যন্ত অ্যারের সামকে আমরা অ্যারের কিছু সেগমেন্টের যোগফল হিসাবে লিখতে পারি। মনে কর আমাদের একটা  $\text{sum}(i,j)\text{sum}(i,j)$  ফাংশন আছে যার কাজ হলো ii থেকে jj ইনডেক্সের মধ্যে অ্যারের সবগুলো এলিমেন্টের যোগফল বের করা। তাহলে ১৩ তম ইনডেক্সের এর জন্য যোগফল হতে পারে  $\text{sum}(1,5)+\text{sum}(6,10)+\text{sum}(11,12)\text{sum}(1,5)+\text{sum}(6,10)+\text{sum}(11,12)$ । এখানে ১৩কে আমরা ৩টি সেগমেন্টে ভাগ করেছি। বাইনারি ইনডেক্স ট্রি তে আমরা প্রতিটি সংখ্যাকে কিছু সেগমেন্টে ভাগ করব। তবে উপরে যেভাবে করেছি সেভাবে ইচ্ছামত করলে হবে না, এটা আমরা একটু বুদ্ধিমানের মত করব। যেহেতু “বাইনারি ইনডেক্সড ট্রি” ব্যবহার করব, তাই বুঝতে পারছ হয়ত যে অ্যারের ইনডেক্সগুলোর বাইনারি রিপ্রেজেন্টেশনকে ব্যবহার করে কোন কাজ করতে হবে।

কোন সংখ্যাকে যেহেতু ২এর পাওয়ারের যোগফল হিসাবে লেখা যায়, তারমানে সংখ্যাটা থেকে ২এর পাওয়ার বিয়োগ দিতে দিতে আমরা ০ বানিয়ে ফেলতে পারি। ১৩ থেকে একে একে ১, ৪ এবং ৮ বাদ দিলে আমরা পাবো:

$$13 - 1 = 12$$

$$12 - 8 = 4$$

$$4 - 4 = 0$$

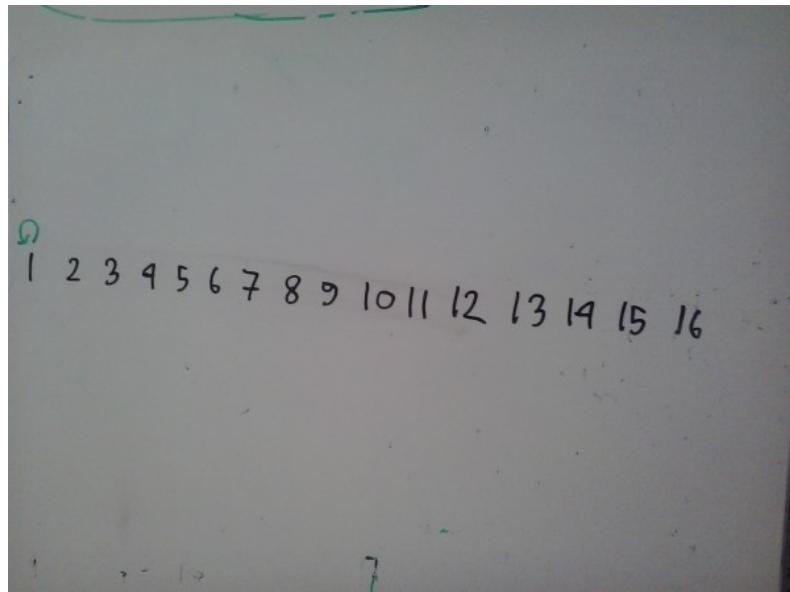
তাহলে ১৩ এর জন্য সেগমেন্ট হবে  $\text{sum}(1,8)+\text{sum}(9,12)+\text{sum}(13,13)\text{sum}(1,8)+\text{sum}(9,12)+\text{sum}(13,13)$ । আরেকটা উদাহরণ দেখলে পরিষ্কার হবে,  $6(\text{বাইনারিতে } 110) = 2^0+2^2$  এর জন্য একই ভাবে লিখতে পারি:

$$6 - 2 = 4$$

$$4 - 4 = 0$$

এবার তাহলে ৬ এর জন্য সেগমেন্ট হবে  $\text{sum}(1,4)+\text{sum}(5,6)$  $\text{sum}(1,4)+\text{sum}(5,6)$ ।

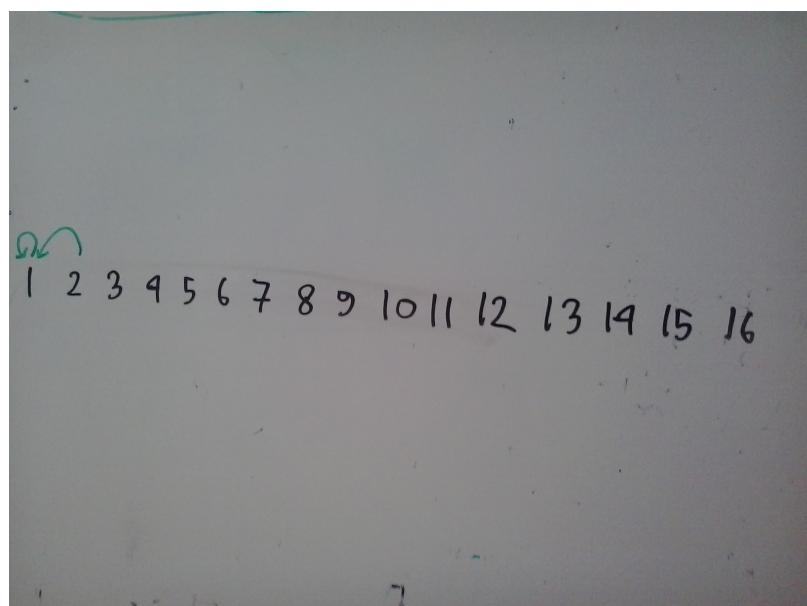
কোড কিভাবে করব সেটা এখন চিন্তা করা দরকার নেই, তুমি শুধু চিন্তা কর গাণিতিকভাবে ভাগটা কিভাবে করা হচ্ছে সেটা তুমি বুঝতে পারছ নাকি। এখন আমরা সেগমেন্টগুলো ছবিতে দেখব:



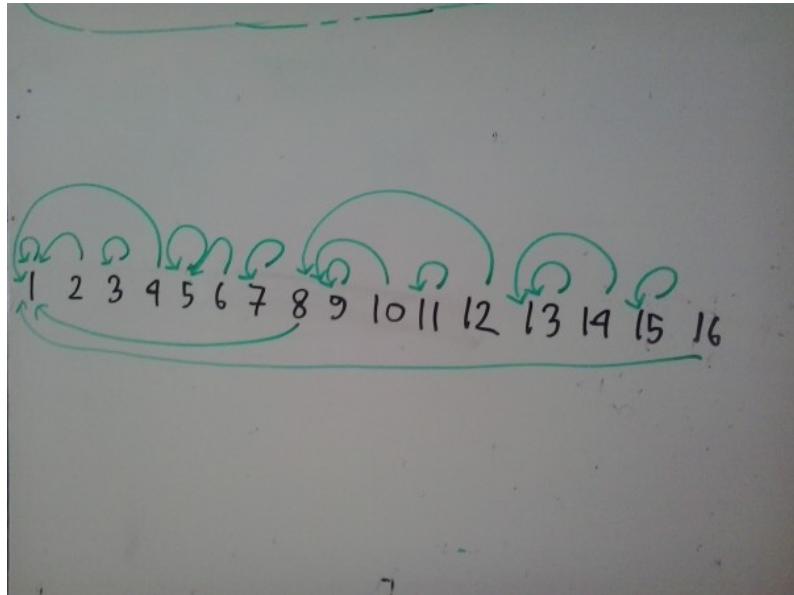
ছবিতে ১ এর জন্য সেগমেন্টটা চিহ্নিত করা হয়েছে। এরপরে আমরা ২ এর জন্য করব। ২ এর জন্য উপরের মত করে লিখতে পারি:

$$2 - 2 = 0$$

তাই সেগমেন্টটা হবে ১থেকে ২পর্যন্ত।



এখন তোমার কাজ ১৬ পর্যন্ত সবগুলো সংখ্যার জন্য এভাবে সেগমেন্ট একে ফেলা। কিভাবে সেগমেন্ট তৈরি হচ্ছে বোঝাটা বাইনারি ইনডেক্সড ট্রি এর সবথেকে দরকারি অংশ, তাই না বুঝলে আবার ভাল করে পড়। তোমার আকার সাথে নিচের ছবিটি মিলিয়ে দেখ:



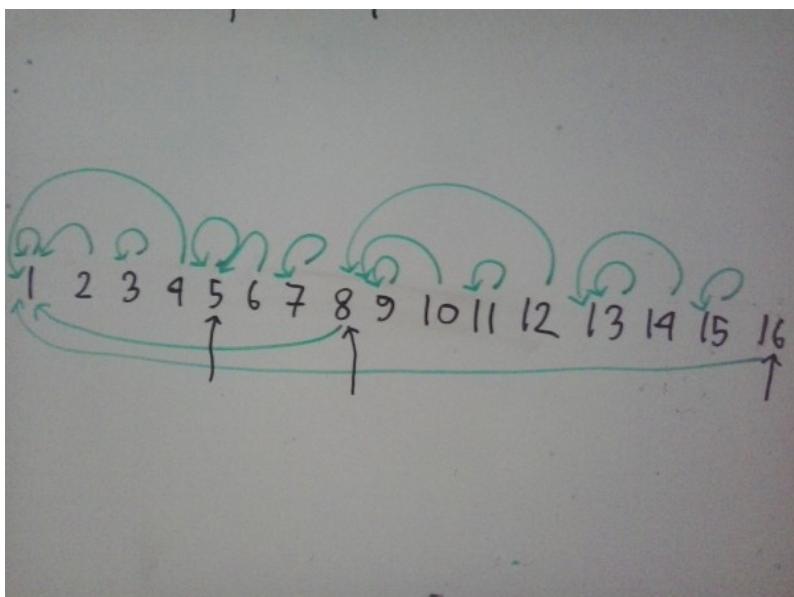
এই ছবি দেখেই আমরা প্রতিটা সংখ্যার জন্য সেগমেন্টগুলো বলে দিতে পারি। যেমন ১১ এর জন্য সেগমেন্টগুলো হবে  $(11,11), (10,9), (8,1) (11,11), (10,9), (8,1)$ । তুমি যদি নিজে বুঝে উপরের ছবিটা একে থাকে তাহলে তুমি বুঝে গেছ কিভাবে ছবি দেখে সেগমেন্ট বের করে ফেললাম।

আমরা আগেই  $\text{sum}(i,j)\text{sum}(i,j)$  ফাংশন ডিফাইন করেছি। কিন্তু সেই ফাংশনটা ব্যবহার করলে আমাদেরকে  $i$  থেকে  $j$  তে লুপ চালিয়ে যোগফল বের করতে হবে যেটা আমরা চাইনা। আমরা আরেকটা অ্যারে ডিফাইন করব  $\text{Tree}[]\text{Tree}[]$ । তুমি ছবিতে দেখতে পাচ্ছ অ্যারের প্রতিটি ইনডেক্স একটা সাবঅ্যারেকে কভার করে। যেমন ১২ নম্বর ইনডেক্স  $(8,12)$  সাবঅ্যারেকে কভার করে। আমরা এমন কিছু করার চেষ্টা করব যেন  $\text{Tree}[i]\text{Tree}[i]$  তে  $i$  তম ইনডেক্স যতটুকু সাবঅ্যারে কভার করে সেটাৰ যোগফল থাকে। তাহলে  $\text{Tree}[12]\text{Tree}[12]$  তে সেভ করা থাকবে  $\text{sum}(8,12)\text{sum}(8,12)$ , আবার  $\text{Tree}[14]\text{Tree}[14]$  তে সেভ করা থাকবে  $\text{sum}(13,14)\text{sum}(13,14)$ ।

এই কাজটা যদি করতে পারি তাহলে ১১ নম্বর ইনডেক্সের জন্য

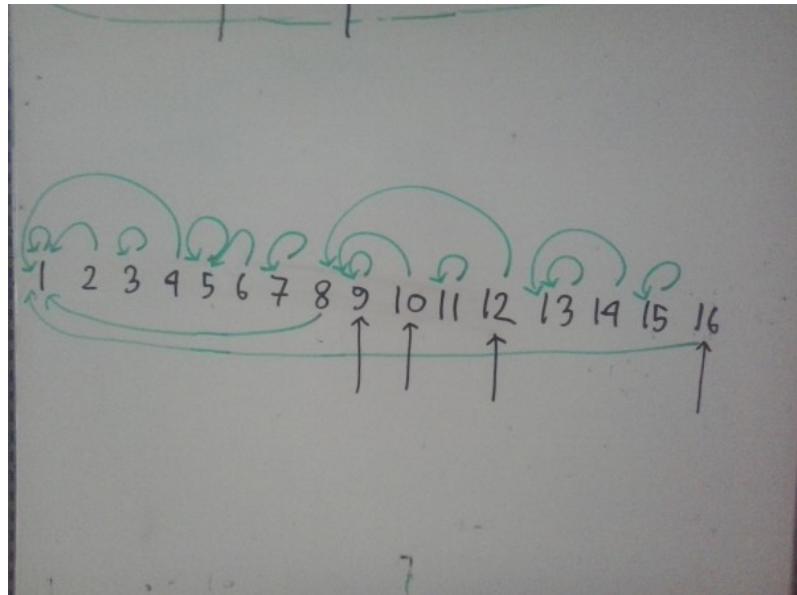
$\text{sum}(11,11)+\text{sum}(10,9)+\text{sum}(8,1)\text{sum}(11,11)+\text{sum}(10,9)+\text{sum}(8,1)$  বের না করে আমরা  $\text{tree}[11]+\text{tree}[10]+\text{tree}[8]\text{tree}[11]+\text{tree}[10]+\text{tree}[8]$  বের করলেই যোগফল পেয়ে যাবো।

এখন ধরি আমাকে বলা হল ৫ নম্বর ইনডেক্সে  $xx$  যোগ করতে চাই। আমাকে এখন বুদ্ধিমানের মত  $\text{Tree}$  অ্যারেতে কিছু ইনডেক্সে  $x$  সংখ্যাটা যোগ করে দিতে হবে। আমরা প্রথমে  $\text{Tree}[5]\text{Tree}[5]$  এ  $xx$  যোগ করব, এরপরে সবথেকে ছোট সেই ইনডেক্সে করব যেটা উপরের ছবিতে ৫কে কভার করে। সেই ইনডেক্সটা হলো ৬, তাহলে  $\text{tree}[6]$  এ  $xx$  যোগ করব। এরপরে ৬কে কভার করে ৮, তাই  $\text{tree}[8]$  এ  $x$  যোগ করব। এরপরে একই ভাবে  $\text{tree}[16]$  তে  $xx$  যোগ করব।



আমরা  $\text{tree}$  অ্যারেতে ছবিতে তীর চিহ্ন দেয়া ইনডেক্সগুলোতে  $xx$  যোগ করব। এখানে লক্ষ্য করার বিষয় হলো উপরে লেখা পদ্ধতিতে যোগফল বের করার সময় তুমি এই তৃতীয় ইনডেক্সের সর্বোচ্চ যেকোন এক্সেস করতে পারবে। যেমন ১৬ তে এক্সেস

করলে সরাসরি ৫, ৮ ইত্যাদিকে স্কিপ করে সরাসরি ১ এ চলে যেতে হবে। এই প্রোপার্টি থাকার কারণে একই আপডেট দুইবার গুণে ফেলার সম্ভাবনা নেই। আপডেট অপারেশন ঠিকমত বুঝেছো নাকি পরীক্ষা করতে ৯ এর জন্য উপরের মত করে তীব্রচিহ্ন বসায় এবং তারপর নিচের সমাধানের সাথে মিলিয়ে দেখ:



ছবিতে খেকে শুরু করে প্রতিবার এমন ইনডেক্সে গিয়েছি যেটা আগের ইনডেক্সটাকে পুরোটা কভার করে। যোগফল বের করার সময় কোনভাবেই এই ৪টা ইনডেক্সের একটার বেশি এক্সেস করা সম্ভব না!

এবার আমরা কোডের দিকে যাবো। প্রথমেই কুয়েরি অপারেশনটা ইমপ্লিমেন্ট করব। কোন সংখ্যা ২ এর পাওয়ার হলে তার বাইনারি রিপ্রেজেন্টেশন হয় ১, ১০, ১০০, ১০০০, ১০০..০০ এরকম, অর্থাৎ ১ এর পর কিছু শূন্য। এখন ১১ নাম্বার ইনডেক্সের জন্য কুয়েরি করতে চাই। ১১ কে বাইনারিতে লিখতে পারি ১০১১। আমরা প্রথমেই সবথেকে ডানের ১ খুজে বের করব এবং সেই পজিশন থেকে বাকি অংশটা মূল সংখ্যা থেকে বিয়োগ দিব। তাহলে শুরুতেই বিয়োগ হবে ১।

$$1011 - 1 = 1010 \text{ (অথবা দশমিক পদ্ধতিতে } 11 - 1 = 10)$$

আবারো সব থেকে ডানের ১ থেকে শুরু করে বাকিটা বিয়োগ দিলে পাব:

$$1010 - 10 = 1000 \text{ (অথবা দশমিক পদ্ধতিতে } 10 - 2 = 8)$$

এভাবে বিয়োগ করতে থাকবো যতক্ষণ শূন্য না পাই।

$$1000 - 1000 = 0 \text{ (অথবা দশমিক পদ্ধতিতে } 8 - 8 = 0)$$

লক্ষ্য কর ১১ এর জন্য দরকারি তৃতীয় ইনডেক্স ১১, ১০, ৮ আমরা পেয়ে গিয়েছি ২এর পাওয়ার বিয়োগ করে করে! ১৩ বা বাইনারিতে ১১০১ এর জন্য নিজে কর এবং তারপর নিচের সমাধান দেখ:

$$1101 - 1 = 1100 \text{ (অথবা দশমিক পদ্ধতিতে } 13 - 1 = 12)$$

$$1100 - 100 = 1000 \text{ (অথবা দশমিক পদ্ধতিতে } 12 - 8 = 4)$$

$$1000 - 1000 = 0 \text{ (অথবা দশমিক পদ্ধতিতে } 4 - 4 = 0)$$

১৩ এর জন্য দরকারি ইনডেক্স ১৩, ১২, ৮ পেয়ে গিয়েছি!

এখন প্রশ্ন হল এই কাজটা কোডে করব কিভাবে? নুপ চালিয়ে একটা একটা বিট পরীক্ষা করে করা যেতে পারে কিন্তু কাজ O(1) এই করা যায়। idx নম্বর ইনডেক্সের পরের ইনডেক্সটা হবে:

$$idx = idx - (idx \& -idx)$$

এটা কিভাবে কাজ করে বোঝার জন্য খাতাকলমে বিয়োগটা করে দেখ আমার ব্যাখ্যা পড়ার আগেই! আমি একটা উদাহরণ দিচ্ছি, ধরি  $idx=10$  বা বাইনারিতে 1010। তাহলে 2's complement পদ্ধতিতে  $-idx$  বা  $-10$  হবে  $0101+1=0110$ । (যারা ভুলে গেছ তাদের জন্য, 2's কমপ্লিমেন্ট বের করতে বিটগুলোকে উল্টে দিয়ে 1 যোগ করতে হয়।) এবার যদি বিটওয়াইজ AND করি তাহলে পাবো:

$$\begin{array}{r} 1010 \\ 0110 \\ \hline 0010 \end{array}$$

সবথেকে ডানের 1 থেকে বাকি অংশ পেয়ে গেলাম! এবার বিয়োগ করে দিলেই কাজ শেষ। এটা কেন কাজ করছে? আমরা বাইনারি সংখ্যাটাকে ঢটা অংশে ভাগ করতে পারি  $x1z$ । 1 হলো সবথেকে ডানের 1, x হলো তার আগের যেকোন বিট, z হলো 1 এর পরের থাকা 0 বিট(x বা z এ শৃঙ্খলা, একটা বা একাধিক বিট থাকতে পারে। এখন বিটগুলো উল্টেদিলে পাবো:  $x'0z'$ । x বা z অংশের বিটগুলোকে উল্টে দিয়ে পেয়েছি  $x'$  আর  $z'$ , তারমানে  $z'$  হলো কিছু 1 বিট। 2's কমপ্লিমেন্ট বের করতে 1 যোগ করলে পাবো:  $x'1z$ । এখন আগের সংখ্যা  $x1z$  আর  $x'1z$  কে AND করলে থাকে  $1z$ । x এর ভাগটা পুরোটা শূণ্য হয়ে যাবে তাই সেটা বাদ। তাহলে আমরা ডানের 1থেকে বাকি অংশ পেয়ে যাবো!

কোডটা তাহলে খুব সহজ:

C++

```
1 int query(int idx){
2     int sum=0;
3     while(idx>0){
4         sum+=tree[idx];
5         idx -= idx & (-idx);
6     }
7     return sum;
8 }
```

আপডেট অপারেশন কিভাবে করব? 1010 বা 10 এর পরের কোন ইনডেক্সটা 10 কে কভার করবে? এবার আমরা সবথেকে ডানের 1 কে বামপাশে শিফট করে দিব। 1010 এর সাথে 10 যোগ করলে পাই 1100 বা 12 যেটা 10কে পুরোটা কভার করে। ঠিক সেরকম 12 বা 1100 এর সাথে 100 যোগ করলে পাই 10000 যেটা 12কে পুরোটা কভার করে। এটা কিভাবে কাজ করছে সেটা চিন্তা করে বের করে ফেল!

আগের মত করেই শুধু বিয়োগের জায়গায় যোগ করলেই সবথেকে ডানের 1টা বামে সরে যাবে। তারমানে যে সংখ্যাটা যোগ করছি, নতুন ইনডেক্স থেকে কুয়েরির সময় তারথেকেও বড় একটা সংখ্যা বিয়োগ করা হবে তাই নতুন ইনডেক্সটা আগেরটাকে কভার করতে বাধ্য! কোডটা হবে এরকম:

C++

```

1 void update(int idx, int x, int n) //n is the size of the array, x is the number to add
2 {
3     while(idx<=n)
4     {
5         tree[idx]+=x;
6         idx += idx & (-idx);
7     }
8 }
```

এটাই হলো বাইনারি ইনডেক্সড ট্রি। এখন আমরা সহজেই কোন একটা ইনডেক্স আপডেট করতে পারব এবং 1 থেকে কোন ইনডেক্স পর্যন্ত যোগফল বের করতে পারব।

### কমপ্লেক্সিটি:

আপডেট এবং কুয়েরি সব ক্ষেত্রেই  $2$  এর পাওয়ার যোগ বা বিয়োগ করছি, প্রথমে ছোট পাওয়ার থেকে শুরু করে বড় পাওয়ার বিয়োগ করছি। তাই টাইম কমপ্লেক্সিটি হবে  $O(\log n)$ । শুধুমাত্র Tree অ্যারেটাতেই সব অপারেশন হচ্ছে, স্পেস কমপ্লেক্সিটি  $O(n)$ ।

লক্ষ কর যে এই প্রবলেমটা সেগমেন্ট ট্রি দিয়ে করা গেলেও এখানে মেমরি লাগছে অনেক কম, কোডও খুব ছোট। তবে সেগমেন্ট ট্রিতে বিভিন্ন সব রেঞ্জের ভিতর আপডেট করা যায়, এখানে করা যায় ১ নম্বর ইনডেক্স সহ রেঞ্জে।

**প্রবলেম ১:** কুয়েরি করার সময়  $i$  থেকে  $j$  ইনডেক্সের মধ্যের সাবঅ্যারের যোগফল বের করতে বললে কি করবে?

**প্রবলেম ২:** কুয়েরিতে যদি বলা হয় অ্যারের সর্বনিম্ন কোন ইনডেক্সে গেলে যোগফল  $X$  এর বেশি পাবো তাহলে ইনডেক্সটা কিভাবে বের করবে? ইন্টস:  $O(\log n * \log n)$ ।

অনলাইন জাজ থেকে কিছু প্রবলেম:

[Curious Robin Hood](#)

[Inversion Count](#)

[Nice Day](#)

ডিঃ বাইনারি ইনডেক্সড ট্রি এবং আরো কিছু প্রবলেম নিয়ে ভবিষ্যতে লেখার ইচ্ছা আছে। আজকে এই পর্যন্তই। হ্যাপি কোডিং!

# HANDBOOK OF ALGORITHMS

## Section Graph Algorithms

*Courtesy of*  
*Shafaet Ashraf*

গ্রাফ থিওরিতে হাতখেড়ি - ১ \_ শাফায়তেরে ব্লগ \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি – ২ (টিউটোরিয়াল) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি – ৩ (ভ্যারিবেলে গ্রাফ স্টোর-২) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি-৪ (ব্রথেড ফার্স্ট সারচ-bfs) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ৫ \_ মনিমাম স্প্যানিং ট্রি(প্রমি অ্যালগোরিদিম) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ৬ \_ মনিমাম স্প্যানিং ট্রি(ক্রুসকাল অ্যালগোরিদিম) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ৭ \_ টপোলজিকাল সর্ট \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ৮ \_ ডপেথ ফার্স্ট সারচ এবং আবারো টপোলজিকাল সর্ট \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি-৯ (ডায়াকস্ট্রো) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১০ \_ ফ্লয়ডে ওয়ারশল \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১১ \_ বলেম\_যান ফোর্ড \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১২ – ম্যাক্সিমাম ফ্লো (১) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি-১২ – ম্যাক্সিমাম ফ্লো (২) \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৩ \_ আর্টিকুলেশন পয়নেট এবং ব্রজি \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৪ – স্ট্রংলি কানকেটডে কম্পনেনেট \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৫ \_ স্ট্রিল ম্যারজে প্রবলমে \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৬ \_ মনিমাম ভারটকেস কভার প্রবলমে \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৭ \_ ট্রিডিয়ামটির \_ শাফায়তেরে ব্লগ.pdf

গ্রাফ থিওরিতে হাতখেড়ি ১৮ \_ লংগস্টে পাথ প্রবলমে \_ শাফায়তেরে ব্লগ.pdf

# গ্রাফ থিওরিতে হাতেখড়ি - ১

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

ডিসেম্বর ২১, ২০১০

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 লিওনার্ড অয়লার গ্রাফ থিওরি আবিষ্কার করেন কনিসবার্গের সাতটি বিজের 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 খুজে বের করা। তুমি হয়তো জানো সার্ভার থেকে একটা ওয়েবপেজ তোমার পিসিতে পৌছাতে অনেকগুলো রাউটার পার করে আসতে হয়, গ্রাফ থিওরি দিয়ে এক রাউটার থেকে আরেকটাতে যাবার পথ খুজে 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 মজোর 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?

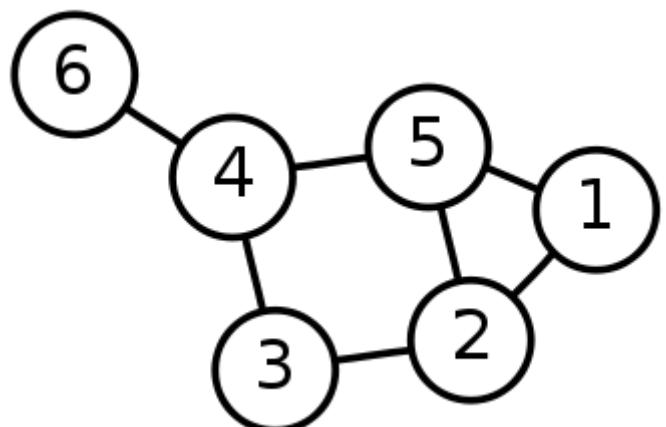
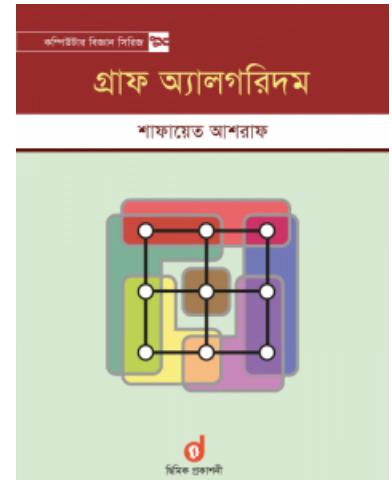
ধরা যাক ৬টি শহরকে আমরা ১,২,৩,৪,৫,৬ দিয়ে চিহ্নিত করলাম। এবার যে শহর থেকে যে শহরে সরাসরি রাস্তা আছে তাদের মধ্যে লাইন টেনে দিলাম:

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 বোর্ডাও একটা গ্রাফ। প্রতিটা ঘর একটা করে নোড। যে ঘরে ঘোড়া আছে সেখান থেকে এজগুলো দেখানো হয়েছে:

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

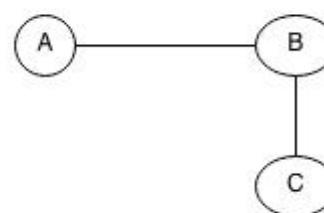
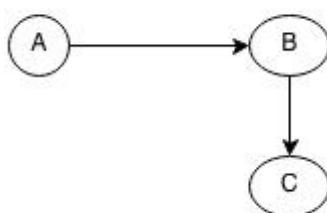
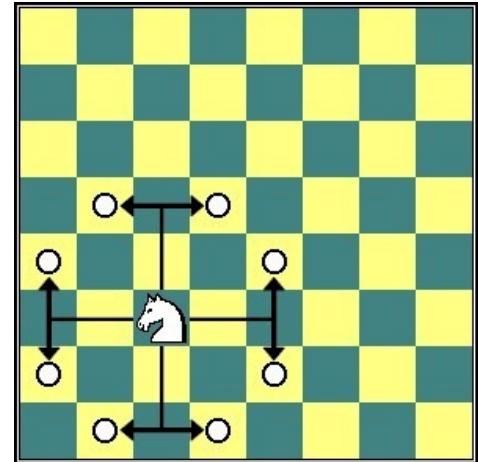


## অ্যাডজেসেন্ট নোড:

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 একমুখ্য(Unidirectional), আনডিরেক্টেড গ্রাফে এজগুলো দ্বিমুখ্য(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।

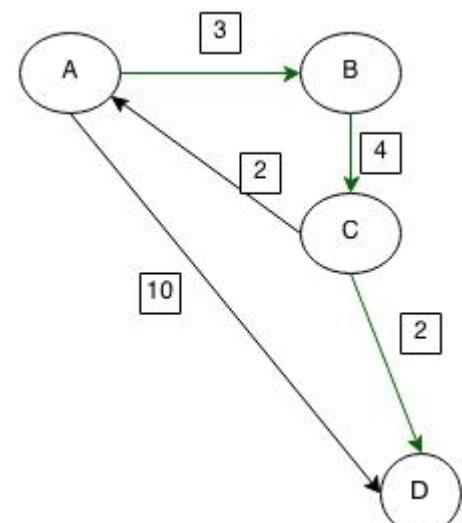
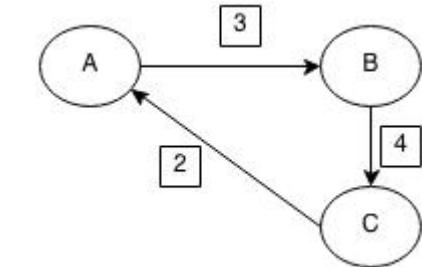
এক নোড থেকে আরেক নোডে যাবার অনেকগুলো পাথ থাকতে পারে। ছবিতে A থেকে D তে যাবার দুইটা পাথ আছে। A->B,B->C,C->D হলো একটা পাথ, এই পাথের মোট ওয়েট হলো  $3+4+2=9$ । আবার A->D ও একটা পাথ হতে পারে যেই পাথের মোট কস্ট ১০। যে পাথের কস্ট সবথেকে কম সেটাকে শর্টেস্ট পাথ বলে।

## 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 সংখ্যাটাই 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, এবং বিজোড় ডিগ্রীর নোড। তাহলে বিজোড় ডিগ্রীর নোড আছে ২টা, ২ হলো একটা জোড় সংখ্যা। হ্যান্ডশেক করতে সবসময় ২টা হাত লাগে, ঠিক সেরকম একটা এজ সবসময় ইটা নোডকে ঘোগ 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  
বাঢ়ে।

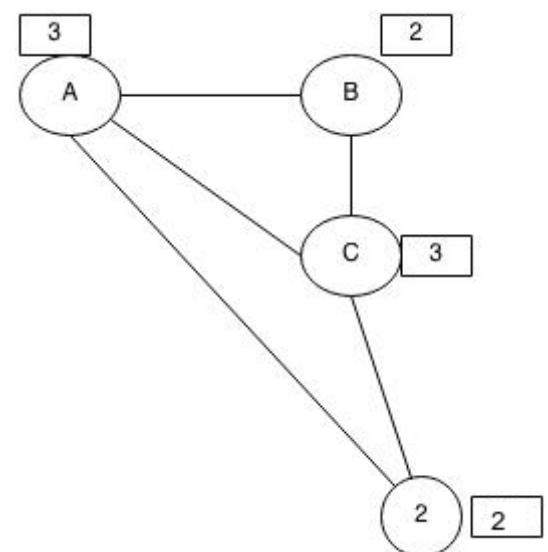
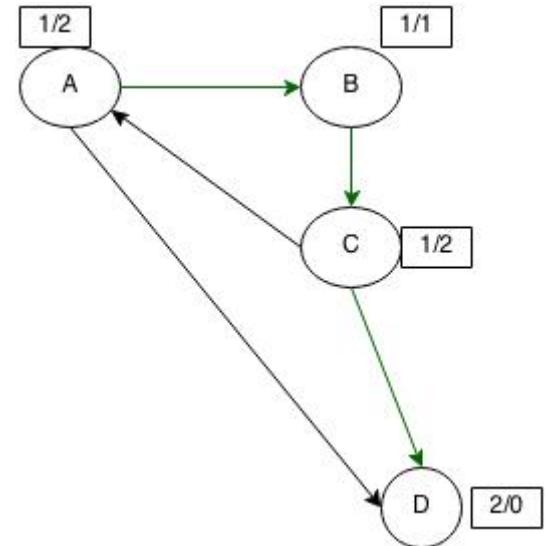
তাহলে দেখা যাচ্ছে হয় ২টা করে বাঢ়তেসে বা ২টা করে কমতেসে বা  
সমান থাকছে, তাই বিজোর ডিগ্রীর নোডের সংখ্যা সবসময় জোড়।

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 **ভ্যারিয়েবলে গ্রাফ স্টোর** করতে হয়। আশা করি গ্রাফ  
থিওরীতে তোমার যাত্রাটা দারুণ আনন্দের হবে, অনেক কিছু শিখতে  
পারবে।

### অন্যান্য পর্ব

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by  
[AccessPress Themes](#)



# গ্রাফ থিওরিতে হাতেখড়ি - ২ (ভ্যারিয়েবলে গ্রাফ স্টোর-১)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

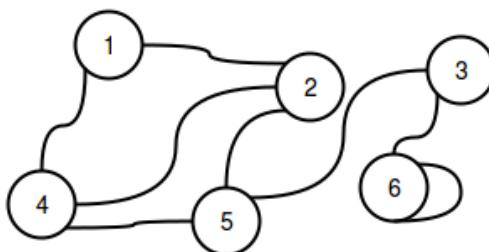
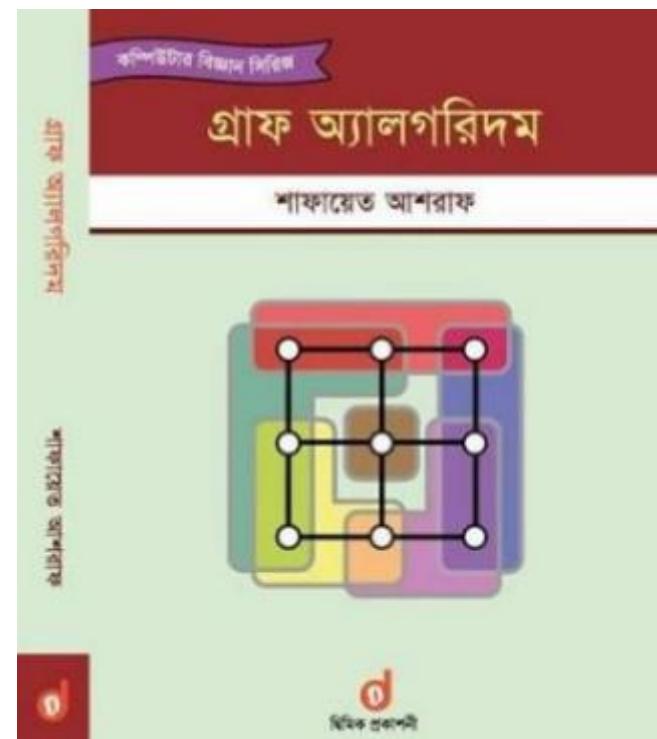
ডিসেম্বর ২৬, ২০১০

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 তায় 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 শুধুমাত্র ২-ডি অ্যারে বুঝানো হয়েছে, তাই ঘাবড়ে ঘাবার কিছু নেই!

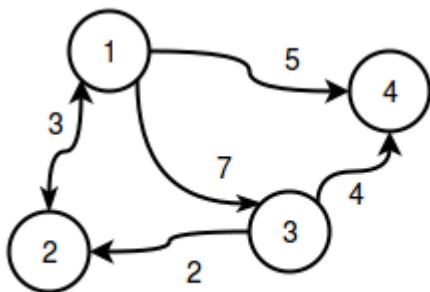


| Nodes | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| 1     | 0 | 1 | 0 | 1 | 0 | 0 |
| 2     | 1 | 0 | 0 | 1 | 1 | 0 |
| 3     | 0 | 0 | 0 | 0 | 1 | 1 |
| 4     | 1 | 1 | 0 | 0 | 1 | 0 |
| 5     | 0 | 1 | 1 | 1 | 0 | 0 |
| 6     | 0 | 0 | 1 | 0 | 0 | 1 |

গ্রাফের পাশে একটি টেবিল দেখতে পাচ্ছ। এটাই আমাদের অ্যাডজেসেন্সি ম্যাট্রিক্স। ম্যাট্রিক্সের  $[i][j]$   $[i][j]$  ঘরে 11 থাকে যদি  $i$  থেকে  $j$  তে কোনো এজ থাকে, না থাকলে 00 বসিয়ে দেই।

এজগুলা ওয়েটেড হতে পারে, যেমন ঢাকা থেকে ঢাকামে একটা এজ দিয়ে বলে দিতে পারে শহর দূরত্ব ৩০০ কিলোমিটার। তাহলে তোমাকে ম্যাট্রিক্সে ওয়েটেড 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 ডিরেক্টেড ওয়েটেড গ্রাফের অ্যাডজেসেন্সি ম্যাট্রিক্সের উদাহরণ দেখানো হয়েছে।



| Nodes | 1   | 2   | 3   | 4   |
|-------|-----|-----|-----|-----|
| 1     | inf | 3   | 7   | 5   |
| 2     | 3   | inf | inf | inf |
| 3     | inf | 2   | inf | 4   |
| 4     | inf | inf | inf | inf |

যেসব নোড এর ভিতর কোনো এজ নাই তাদেরকে এখানে ইনফিনিটি বা অনেক বড় একটা সংখ্যা দিয়ে দেখানো হয়েছে।

একটা ব্যাপার লক্ষ করো, গ্রাফ আনডিরেক্টেড হলে ম্যাট্রিক্সটি সিমেট্রিক হয়ে যায়, অর্থাৎ  $\text{mat}[i][j]=\text{mat}[j][i]$  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 অ্যারে 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:

```

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, ৮০০ 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।](#)

### অ্যাডজেসন্সি ম্যাট্রিক্স ব্যবহার করার সমস্যা:

মেমরি একটা বিশাল প্রবলেম, এজ যতগুলোই থাকুকনা কেন তোমার লাগছে  $N \times NN \times N$  সাইজের ম্যাট্রিক্স যেখানে  $NN$  হলো নোড সংখ্যা।  $10000 \times 10000$  টা নোড হলো  $N \times NN \times N$  ম্যাট্রিক্সের সাইজ দাঢ়াবে  $4 \times 1000 \times 10004 \times 1000 \times 1000$  বাইট বা প্রায় 381381 মেগাবাইট! এজ কম হলে এটা মেমরির বিশাল অপচয়।

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

### অ্যাডজেসন্সি ম্যাট্রিক্স ব্যবহার করার সুবিধা:

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

এই সমস্যাগুলো দূর করে দিবে অ্যাডজেসন্সি 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)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

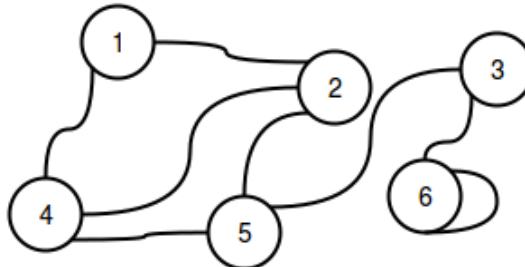
শাফায়েত

ডিসেম্বর ২৭, ২০১০

## আগের পর সবগুলো পর্ব

এই পর্বে গ্রাফ স্টোর করার ২য় পদ্ধতি অ্যাডজেসন্সি 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।

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



| Nodes | 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 |

এবার বাজার করার লিস্টের মত একটি লিস্ট বানাই:

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

## প্রথম উপায়(অ্যারে):

সাধারণ ২ড়ি 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;

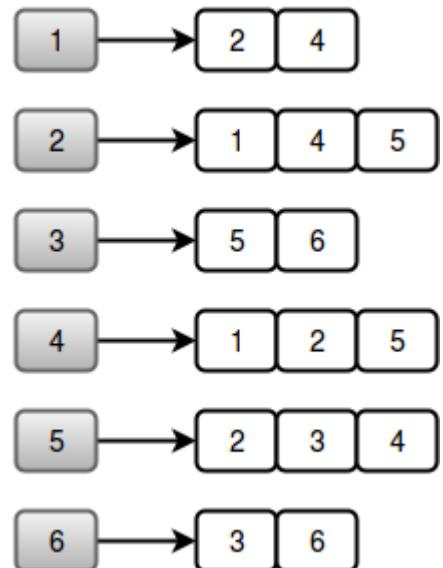
কিন্তু এভাবে স্টোর করলে কিছু সমস্যা আছে:

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 ইন্টিজার 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;)। অ্যাডজেসন্সি 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 মেইনটেইন করতে হবে।



## দ্বিতীয় উপায়(ভেক্টর):

সমস্যাগুলা দূর করতে আমরা STL vector বা list ব্যবহার করে গ্রাফ স্টোর করব। ভেক্টর/লিস্টে 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। নিচয়েই 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 size=edges[1].size();
2 for(int i=0; i < size ; i++)
3 printf("%d ",edges[1][i]);
```

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:

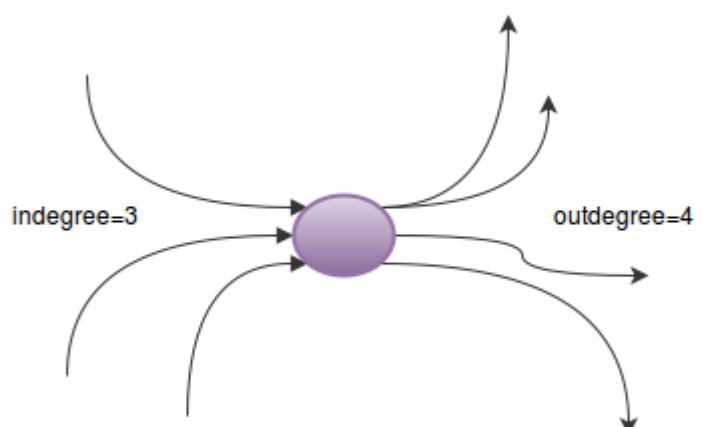
এ পর্যন্ত বুঝে থাকলে তুমি মোটামুটি bfs,dfs এর মত বেসিক অ্যালগোরিদম গুলো শেখার জন্য প্রস্তুত। পরবর্তি লেখাটি পড়ার আগে একটি ছোট exercise করে ফেল। এমন একটি কোড লিখ যেটায় উপরের মত করে ইনপুট দিলে নিচের কাজগুলো করে:

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>

AccessPress Staple | WordPress Theme:

AccessPress Staple by [AccessPress Themes](#)



# গ্রাফ থিওরিতে হাতেখড়ি-৪(ব্রেডথ ফাস্ট সার্চ)

 shafaetsplanet.com/planetcoding/

শাফায়েত

ফেব্রুয়ারি ২২, ২০১৮

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 ফাস্ট সার্চ(breadth first search,bfs)।

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

বিএফএস অ্যালগোরিদমটা কাজ করে নিচের ধারণারগুলোর 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 না বুঝলে আমরা একটা উদাহরণ 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 ভিজিটেড চিহ্নিত করি।

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

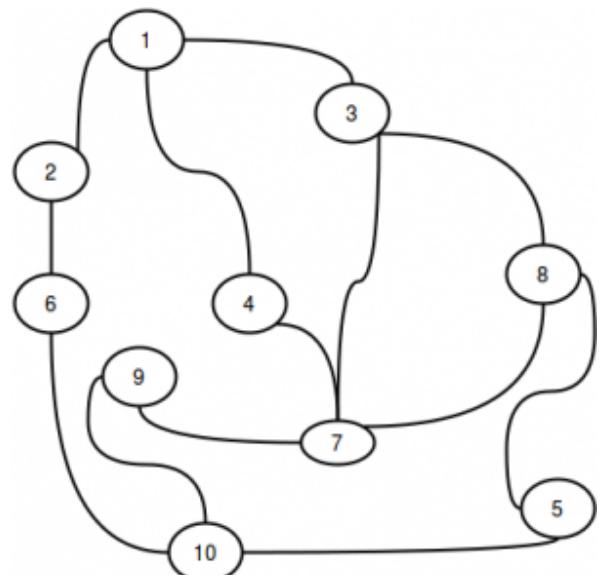
লাল নোডগুলো নিয়ে আমরা এখন কাজ করবো। রঙিন সবগুলো নোড ভিজিটেড, এক নোডে ২বার কখনো যাবোনা। ২, ৩, ৪ থেকে শর্টেস্ট পথে যাওয়া যায় ৬, ৭, ৮ এ। সেগুলো ভিজিটেড চিহ্নিত করি:

লক্ষ কর যে নোডকে যত নম্বর লেভেলে পাচ্ছি, সোর্স থেকে তার শর্টেস্ট পথের দৈর্ঘ্য ঠিক তত। যেমন ২নম্বর লেভেলে ৮কে পেয়েছি তাই ৮ এর দুরত্ব 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 ভিজিট করে ফেলি:

আমরা দেখতে পাচ্ছে 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 নিচের মত করে আকতে পারি:

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 খুজে বের করা, এভাবে যতক্ষণ না গন্তব্যে পৌছে যাচ্ছি অথবা সব নোড ভিজিট করা শেষ হয়ে গিয়েছে ততক্ষণ 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 দাঢ়ায়, যখন  
কোন একটা সংখ্যা বের করে ফেলা হয় তখন সবার প্রথমের সংখ্যাটা  
নেয়া হয়। একে বলা ফার্স্ট ইন ফার্স্ট 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:



১ এর লেভেল হবে ০ বা  $\text{লেভেল}[1] = 0$ । এবার বিএফএস শুরু  
করবো।

প্রথমে কিউ এর সবার সামনের নোডটাকে নিয়ে কাজ করবো।  
সবার সামনে আছে ১, সেখান থেকে যাওয়া যায় ৪, ৩, ২ এবং ৮, ৩,  
২ এ এসেছি ১ থেকে, তাহলে  $\text{লেভেল}[8] = \text{লেভেল}[1] + 1 = 1$ ,  
 $\text{লেভেল}[3] = \text{লেভেল}[1] + 1 = 1$ ,  $\text{লেভেল}[2] = \text{লেভেল}[1] + 1 = 1$

১ কে ফেল দিয়ে এদেরকে কিউতে পুশ করে রাখি:

এবার ৪ নিয়ে কাজ করি। ৪ থেকে যাওয়া যায় ৭ এ। তাহলে আমরা  
বলতে পারি  $\text{লেভেল}[7] = \text{লেভেল}[4] + 1 = 2$ । ৪ কে ফেলে দিয়ে ৭ কে  
কিউতে পুশ করি:

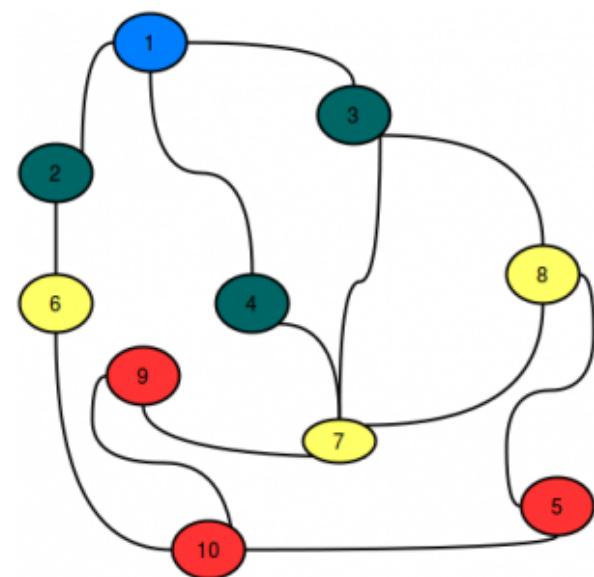
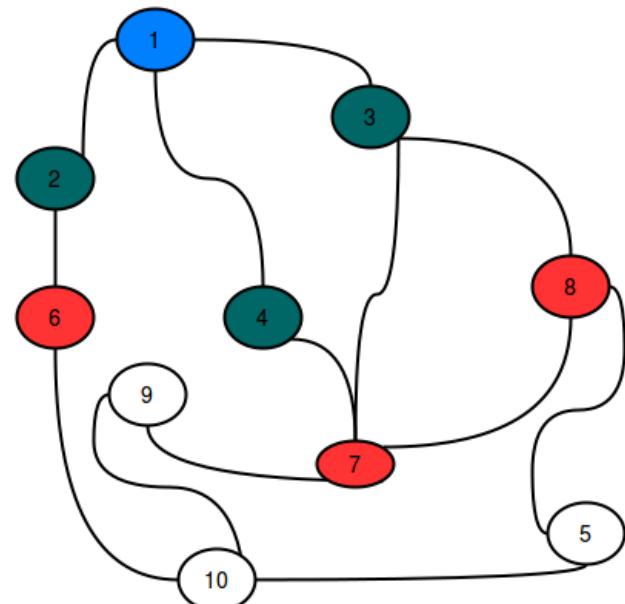
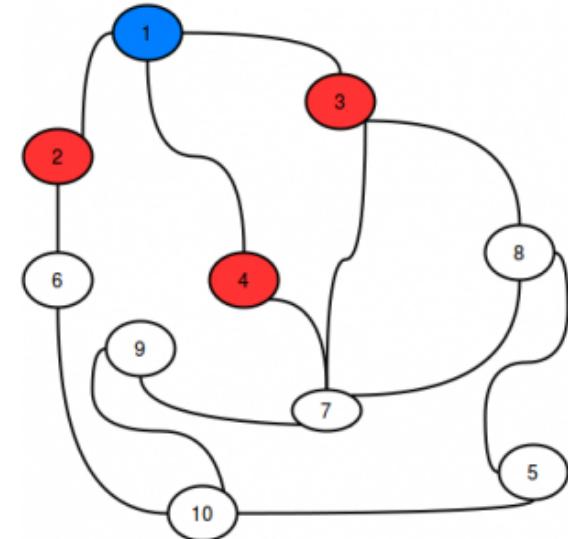
৩ থেকে ৭, ৮ এ যাওয়া যায়। ৭ কে এরই মধ্যে নিয়েছি, শুধু ৮ পুশ  
করতে হবে।  $\text{লেভেল}[8] = \text{লেভেল}[3] + 1 = 2$ ।

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

সুড়োকোড:

```

1 1 procedure BFS(G,source):
2 2   Q=queue(), level[] = infinity
3 3   Q.enqueue(source)
4 4   level[source]=0
5 5   while Q is not empty
6 6     u ← Q.pop()
7 7     for all edges from u to v in G.adjacentEdges(v)
8 do
9     if level[v] = infinity:
10        level[u]=level[v]+1;
11        Q.enqueue(v)
12    end if
13 12   end for
14 13   end while
14. Return distance;
```



ঠিক যেভাবে সিমুলেট করেছি সেভাবেই কোডটা লিখেছি, আশা করি বুজতে সমস্য হচ্ছেন।

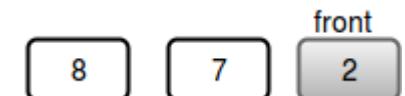
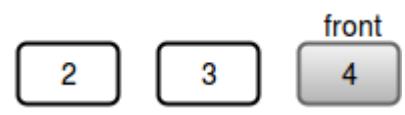
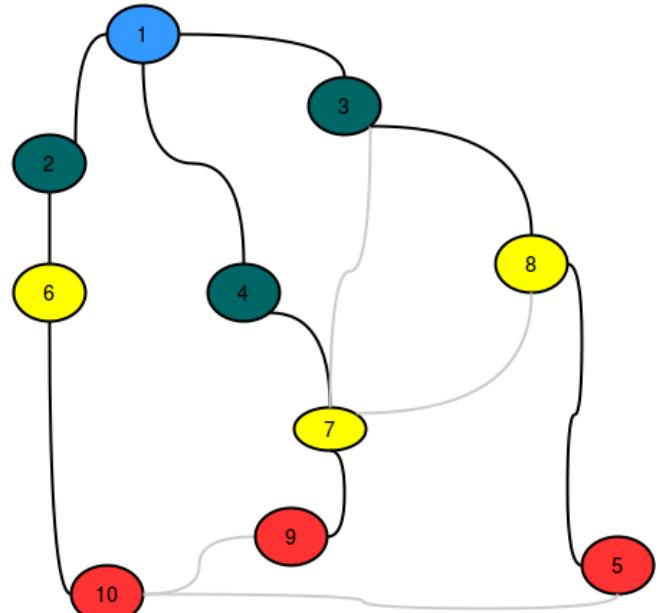
শুধু পাথের দৈর্ঘ্য ঘষেছে না, পাথটাও দরকার হতে পারে। লক্ষ্য করো আমরা  $uu$  থেকে  $vv$  তে যাবার সময়  $parent[v]=uparent[v]=u$  করে দিচ্ছি। আমরা প্রতিটা নোডের জন্য জানি কোন নোড থেকে সেই নোডে এসেছি। তাহলে আমরা যে নোডের জন্য পাথ বের করতে চাই সেই নোড থেকে তার প্যারেন্ট নোডে যেতে থাকবো যতক্ষণনা সোর্স পৌছে 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  $O(V+E)O(V+E)$  যেখানে  $VV$  হলো নোড সংখ্যা এবং  $EE$  হলো এজ সংখ্যা।

কখনো কখনো ২-ডি গ্রিডে বিএফএস চালানো লাগতে পারে।

যেমন একটা দাবার বোর্ডে একটি ঘোড়া আর একটা রাজা আছে। ঘোড়টা মিনিমাম কষট্টা মুভে রাজার ঘরে পৌছাতে পারবে? অথবা একটা ২-ডি অ্যারেতে কিছু সেল রাক 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 সি তে একটা স্ট্রাকচার বানিয়ে নিতে পারি এরকম:



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

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

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

এ ক্ষেত্রে ভিজিটেড, প্যারেন্ট, লেভেল অ্যারেগুলো হবে ২ ডিমেনশনের, যেমন  $visited[10][10]visited[10][10]$  ইত্যাদি। কিউতে নোডের বদলে স্ট্রাকচার পুশ করবো। আর কোন একটা ঘর থেকে অন্য ঘরে যাবার সময় চেক করতে হবে বোর্ডের বাইরে চলে যাচ্ছে কিনা। একটা স্যাম্পল সি++ 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 neighbor is
21            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।

বিএফএস শুধুমাত্র আন-ওয়েটেড গ্রাফে কাজ করে, ওয়েটেড গ্রাফে শর্টেস্ট পাথ বের করতে [ডায়াক্রস্ট্রা](#) অ্যালগোরিদম ব্যবহার করতে পারো। গ্রাফে নেগেটিভ সাইকেল থাকলে [বেলম্যান ফোর্ড](#) ব্যবহার করতে হবে।

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 ৫: মিনিমাম স্প্যানিং ট্রি(প্রিম অ্যালগোরিদম)

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

আগস্ট 8, ২০১১

(সিরিজের অন্যান্য পোস্ট)

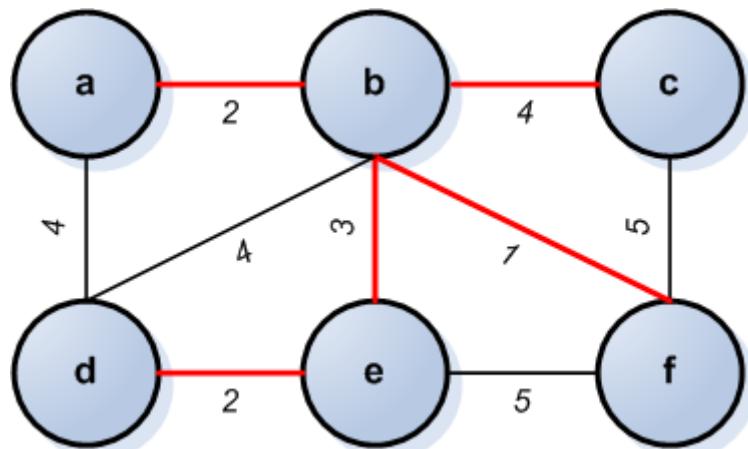
একটি গ্রাফ থেকে কয়েকটি নোড আর এজ নিয়ে নতুন একটি গ্রাফ তৈরি করা হলে সেটাকে বলা হয় সাবগ্রাফ। স্প্যানিং ট্রি হলো এমন একটি সাবগ্রাফ যেটায়:

\* মূল গ্রাফের সবগুলো নোড আছে।

\* সাবগ্রাফটি একটি ট্রি। ট্রিতে কখনো সাইকেল থাকেনা, এজ থাকে  $n-1$  টি যেখানে  $n$  ইলো নোড সংখ্যা।

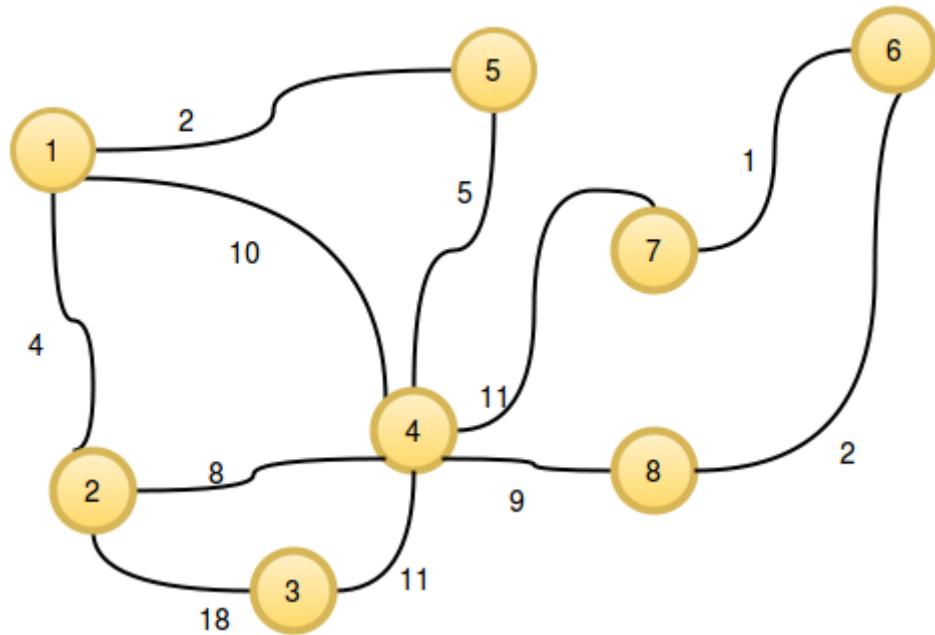
একটি গ্রাফের অনেকগুলো স্প্যানিং ট্রি থাকতে পারে, যে ট্রি এর এজ গুলোর কস্ট/ওয়েট এর যোগফল সব থেকে কম সেটাই মিনিমাম স্প্যানিং ট্রি। আমরা এই লেখায় প্রিম অ্যালগোরিদমের সাহায্যে মিনিমাম স্প্যানিং ট্রি বের করা শিখবো।

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:



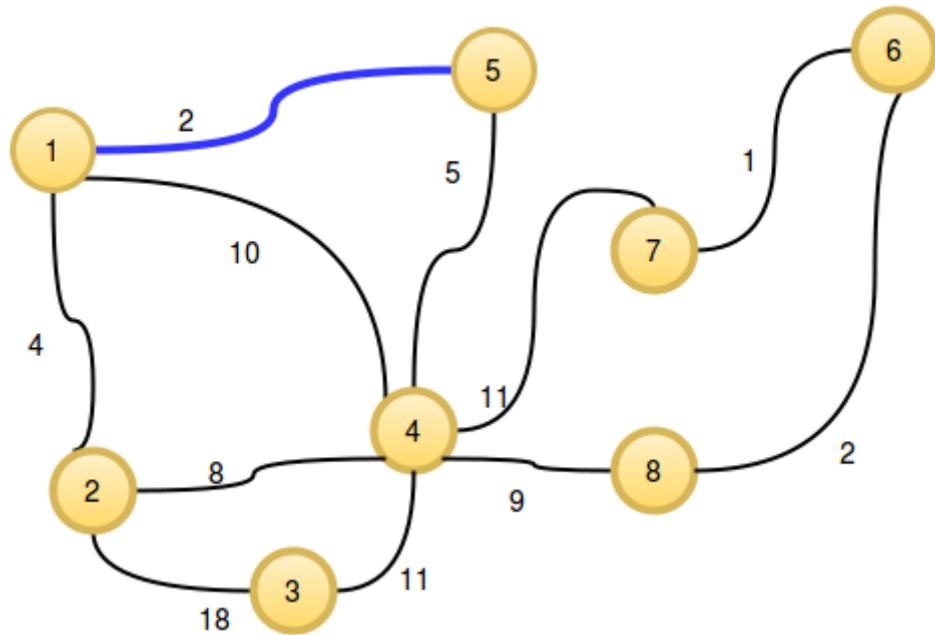
আমরা অনেক ভাবে লাইন বসাতে পারতাম। ছবিতে লাল এজ দিয়ে টেলিফোন লাইন বসানোর একটি উপায় দেখানো হয়েছে। টেলিফোন লাইনগুলো একটি সাবগ্রাফ তৈরি করেছে যেটায় অবশ্যই  $n-1$  টি এজ আছে, কোনো সাইকেল নেই কারণ অতিরিক্ত এজ বসালে আমাদের খরচ বাঢ়বে, কোনো লাভ হবেনা। মিনিমাম স্প্যানিং ট্রি বের করার সময় আমরা এমন ভাবে এজগুলো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

এখন নিচের গ্রাফ থেকে কিভাবে আমরা মিনিমাম স্প্যানিং ট্রি বের করব?

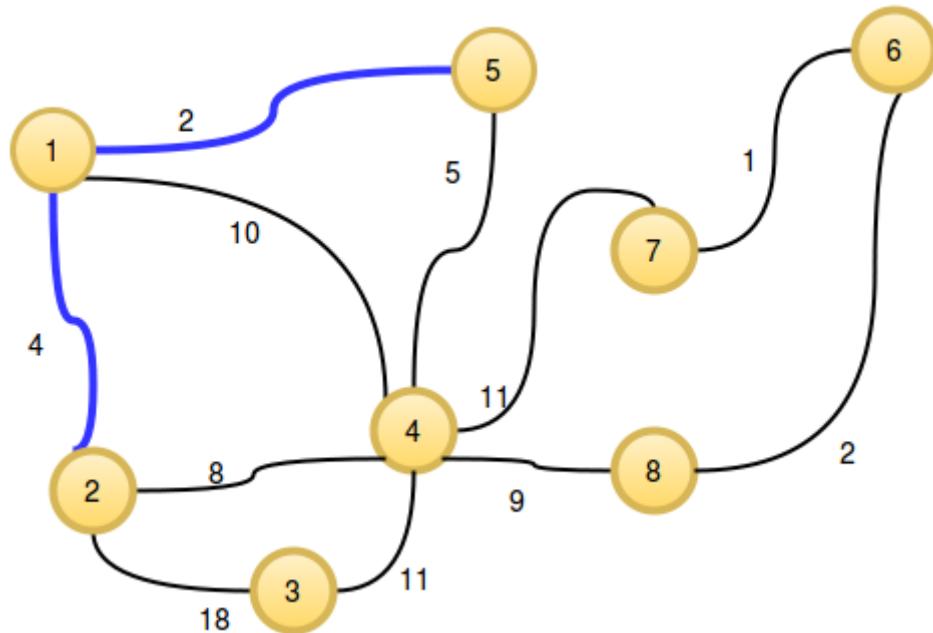


গ্রিডি(greedy) অ্যাপ্রোচে খুব সহজে মিনিমাম স্প্যানিং ট্রি বের করা যায়। আমরা এখন প্রিমস অ্যালগোরিদম কিভাবে কাজ করে দেখব। তুমি যদি আগে ক্রসকাল শিখতে চাও তাহলেও সমস্যা নেই, [সরাসরি পরের পর্বে](#) চলে যেতে পারো।

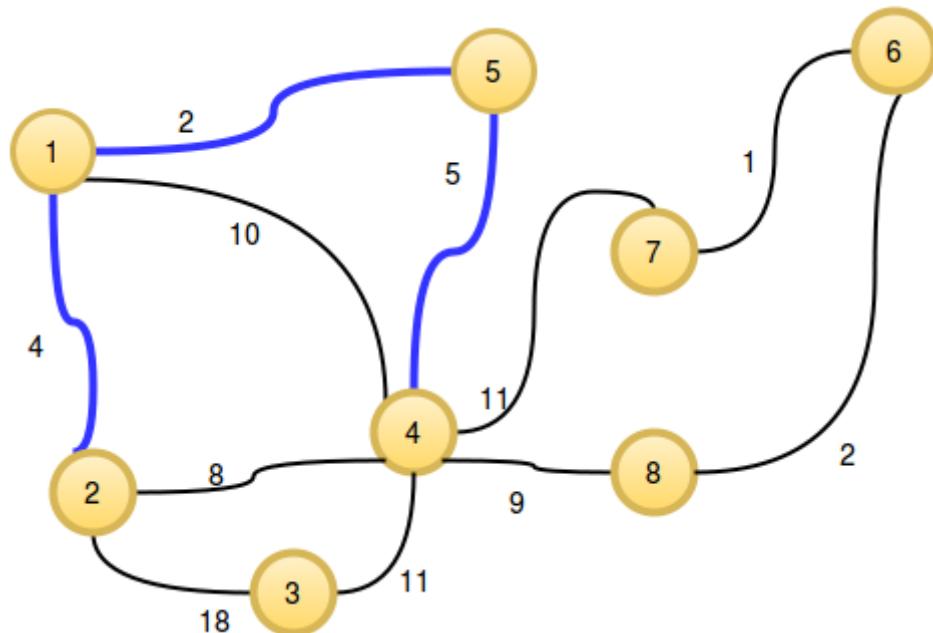
আমরা প্রথমে যেকোনো একটি সোর্স নোড নিব। ধরি সোর্স হলো ১। ১ থেকে যতগুলো এজ আছে সেগুলোর মিনিমাম টিকে আমরা সাবগ্রাফে যোগ করব। নিচের ছবিতে নীল এজ দিয়ে বুঝানো হচ্ছে এজটি সাবগ্রাফে যুক্ত করা হয়েছে:



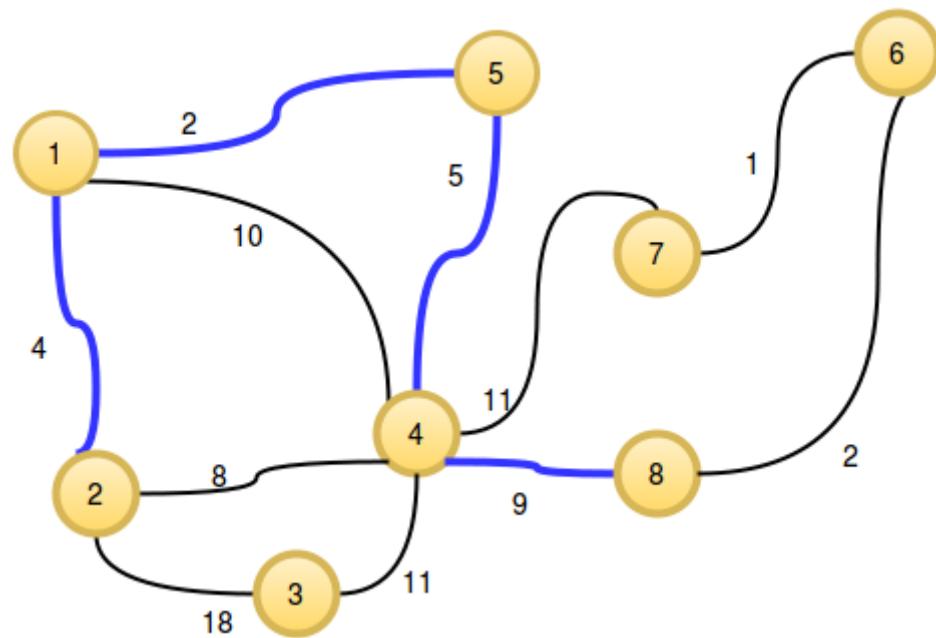
এবার সোর্স ১ এবং ৫ নম্বর নোড থেকে মোট যত এজ আছে(আগের এজগুলো সহ) তাদের মধ্যে মিনিমাম টি নিব:



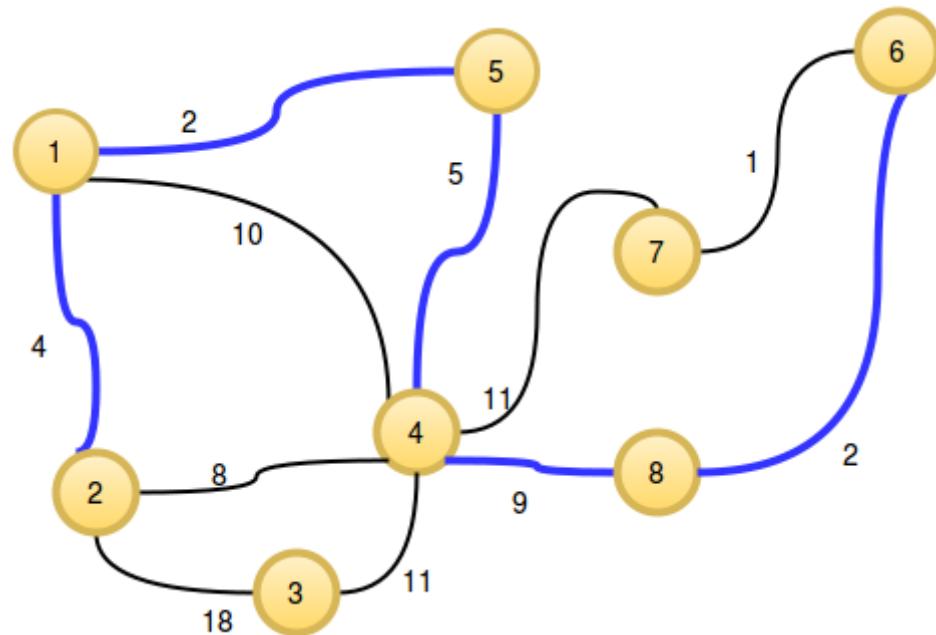
এবার নিব ১,২ এবং ৫ নম্বর নোড থেকে মোট যত এজ আছে(আগের এজগুলো সহ) তাদের মধ্যে মিনিমাম:



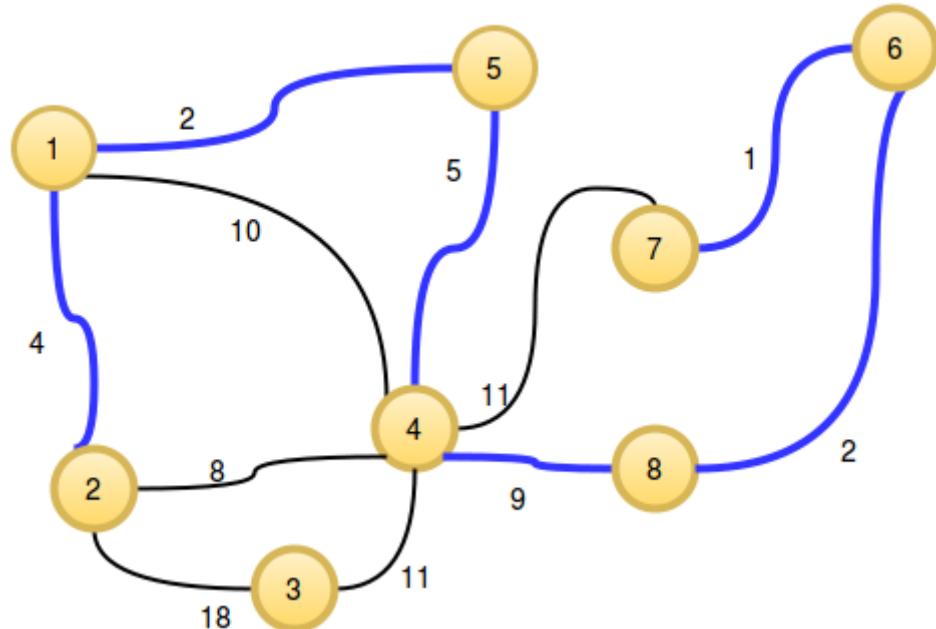
**পরের ধাপটি গুরুত্বপূর্ণ।** ১,২,৫,৪ থেকে যত এজ আছে তাদের মধ্য মিনিমাম হলো ২-৪, কিন্তু ২ নম্বর নোড এবং ৪ নম্বর নোড দুইটাই অলরেডি সাবগ্রাফের অংশ, তারা আগে থেকেই কানেক্টেড, এদের যোগ করলে সাবগ্রাফে সাইকেল তৈরি হবে, তাই ২-৪ এজটি নিয়ে আমাদের কোনো লাভ হবেনা। আমরা এমন প্রতিবার এজ নিব যেন নতুন আরেকটি নোড সাবগ্রাফে যুক্ত হয়। তাহলে ৪-৮ হবে আমাদের পরের চয়েস।



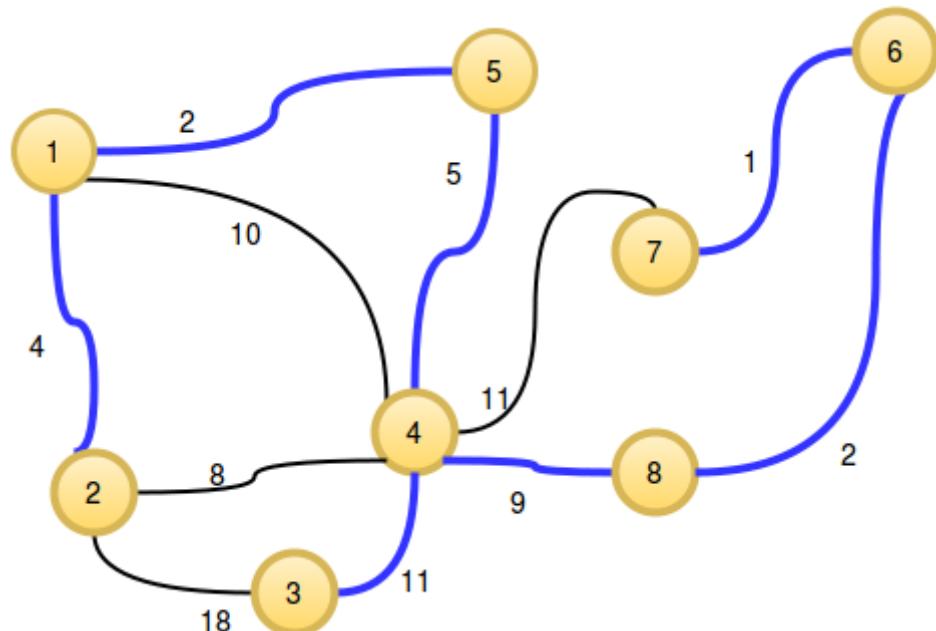
এরপর ৮-৬ যোগ করবো:



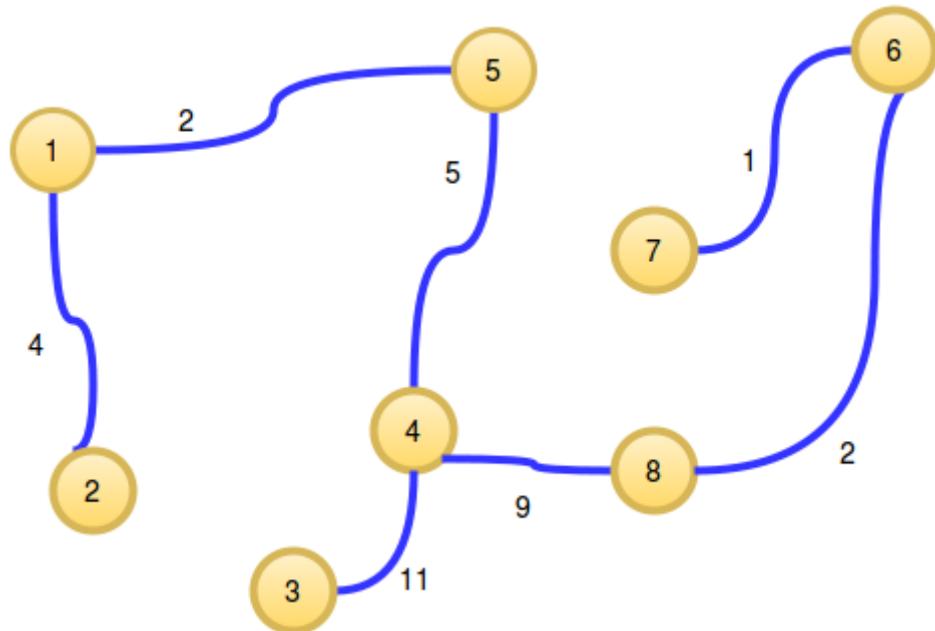
এরপর ৬-৭:



সবশেষে ৪-৩ যোগ করবো:



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



তাহলে টেলিফোন লাইন বসাতো মোট খরচ:  $4 + 2 + 5 + 11 + 9 + 2 + 1 = 38$ । একটি গ্রাফে এক বা একাধিক মিনিমাম স্প্যানিং ট্রি থাকতে পারে।

আমাদের সুড়েকোড হবে এরকম:

- 1 \* Input: A non-empty connected weighted graph with vertices  $V$  and edges  $E$  (the weights can be negative).
- 2 \* Initialize:  $V_{\text{new}} = \{x\}$ , where  $x$  is an arbitrary node (starting point) from  $V$ ,  $E_{\text{new}} = \{\}$
- 3 \* Repeat until  $V_{\text{new}} = V$ :
  - 5 o Choose an edge  $(u, v)$  with minimal weight such that  $u$  is in  $V_{\text{new}}$  and  $v$  is not (if there are multiple edges with the same weight, any of them may be picked)
  - 6 o Add  $v$  to  $V_{\text{new}}$ , and  $(u, v)$  to  $E_{\text{new}}$
- 7 \* Output:  $V_{\text{new}}$  and  $E_{\text{new}}$  describe a minimal spanning tree

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

রানটাইম কমাতে প্রায়োরিটি কিউ ব্যবহার করতে পারো। যখন নতুন একটা নোড  $V_{\text{new}}$  তে যোগ করছো তখন সেই নোডের অ্যাডজেসেন্ট সবগুলো এজ প্রায়োরিটি কিউতে ঢুকিয়ে রাখতে হবে। এখন প্রায়োরিটি কিউ থেকে সবথেকে মিনিমাম ওয়েটের এজটা লগারিদম কমপ্লেক্সিটিতে খুজতে পারবে। মোট কমপ্লেক্সিটি হবে  $O(E \log E)$ । তবে এজের বদলে কিউতে নোড পুশ করে  $O(E \log V)$  তে কমপ্লেক্সিটি নামিয়ে আনা যায়, সেটা কিভাবে করা যায় চিন্তা করে বের করো।

অ্যালগোরিদমটা ইম্প্লিমেন্ট করার পর অবশ্যই নিচের সমস্যা গুলো সমাধানের চেষ্টা করবে।

<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)

মিনিমাম স্প্যানিং ট্রি বের করার জন্য আরেকটি অ্যালগোরিদম আছে যা **ক্ৰসকাল অ্যালগোরিদম** নামে পরিচিত। পরের  
পৰ্বে আমরা সেটা শিখবো।

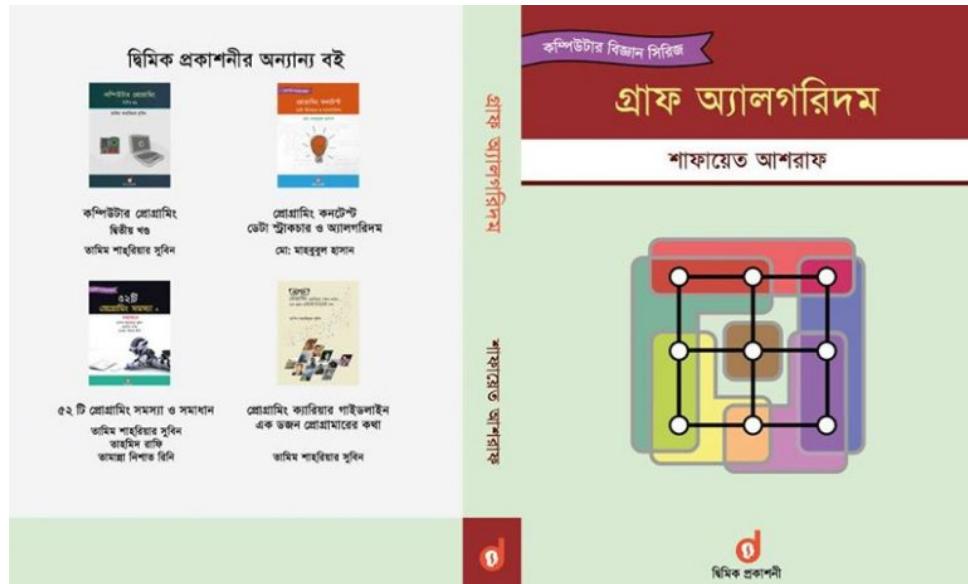
AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# গ্রাফ থিওরিতে হাতেখড়ি ৬: মিনিমাম স্প্যানিং ট্রি(ক্রসকাল অ্যালগরিদম)

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

9/30/2011



## (অন্যান্য পোস্ট)

আগের পোস্টে আমরা প্রিমস অ্যালগোরিদম ব্যবহার করে `mst` নির্ণয় করা দেখেছি। `mst` কাকে বলে সেটাও আগের পোস্টে বলা হয়েছে। এ পোস্টে আমরা `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।

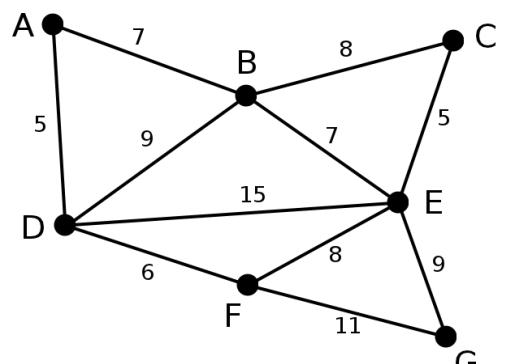
এই পোস্টে নিজের আকা ছবি ব্যবহার করবোনা। উইকিতে ক্রসকাল নিয়ে খুব সুন্দর করে লেখা আছে, আমি ওখানকার ছবিগুলোই ব্যবহার করে সংক্ষেপে অ্যালগোরিদমটা বুঝানোর চেষ্টা করবো।

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 অন্তর্ভৃত্ব করলাম।

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



এরপর সবথেকে ছোট 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।

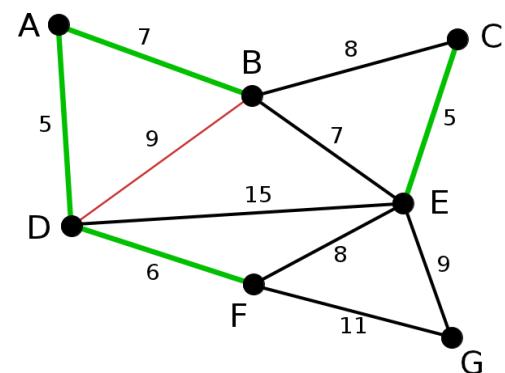
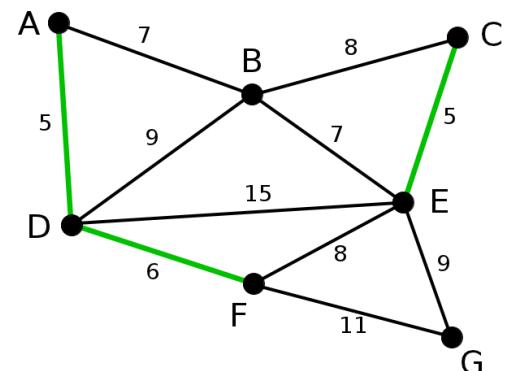
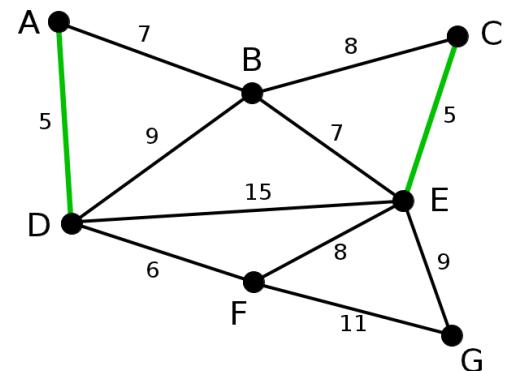
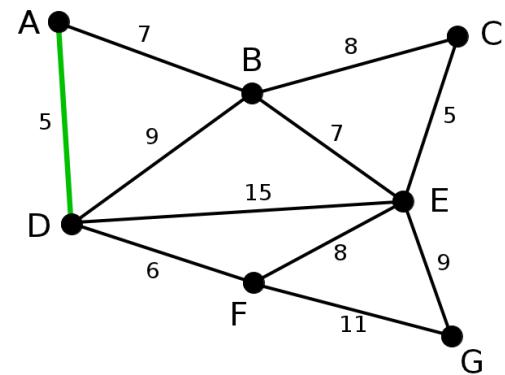
এখন আমরা ইম্প্লিমেন্টেশনে আসি। আমাদের প্রথম কাজ হলো সর্ট করা। পরের কাজ হলো একটি একটি এজ নিয়ে চেক করা যে দু প্রান্তের নোড দুটির মধ্য পথ আছে নাকি, অর্থাৎ তারা একই কম্পোনেন্টের ভিতর আছে নাকি।

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 ইম্প্লিমেন্টেশন দিলাম, আশা করি এটা কপি না করে নিজে বুঝে লিখবে:

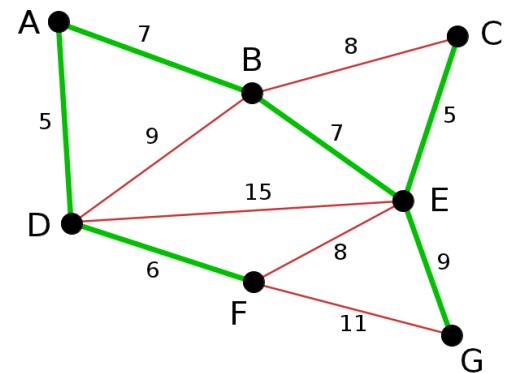
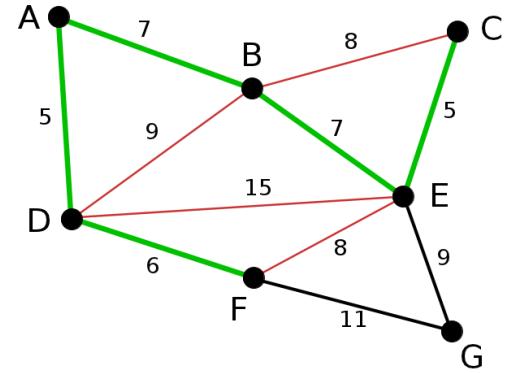
kruskal

C++



```

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



## କମପ୍ଲେକ୍ସିଟି ଅୟାଲାଇସିସ:

ମନେ କରି  $E$  ହଲୋ ଏଜ ସଂଖ୍ୟା। ଏଜଗୁଲୋକେ ସର୍ଟ୍ କରତେ ହବେ, ସେଟୋର କମପ୍ଲେକ୍ସିଟି  $O(E \log E)$ , ଏରପରେ ଶୁଧୁ

এজগুলোর উপর লিনিয়ার লুপ চালাতে হবে। তাহলে মোট **কমপ্লেক্সিটি**  $O(E \log E)$ ।

mst সম্পর্কিত অনেকগুলো সহজ প্রবলেম দিয়েছি প্রিমস এর টিউটোরিয়ালে, ওগুলো সলভ 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>

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# গ্রাফ থিওরিতে হাতেখড়ি ৭:টপোলজিকাল সর্ট

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

10/6/2011

## (অন্যান্য পর্য)

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।

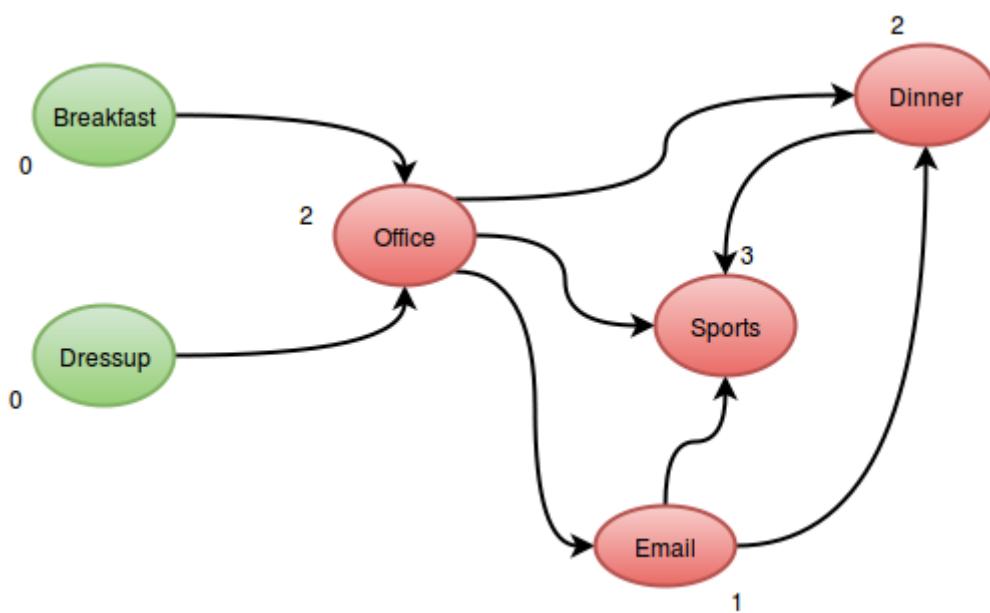
তুমি শর্তগুলোর তালিকা করে ফেললে:

1. সকালের নাস্তা —> অফিস (ক —> খ এর মানে হলো 'খ' কাজটি করার আগে 'ক' কাজটি করতে হবে)
2. সুট-টাই পড়া —> অফিস
3. অফিস —> ইমেইল
4. অফিস —> খেলা
5. অফিস —> খেলা
6. ইমেইল —> ডিনার
7. ইমেইল —> খেলা
8. ডিনার —> খেলা

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

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

আমরা প্রথমেই সমস্যাটাকে নিচের গ্রাফ দিয়ে মডেলিং করবো:

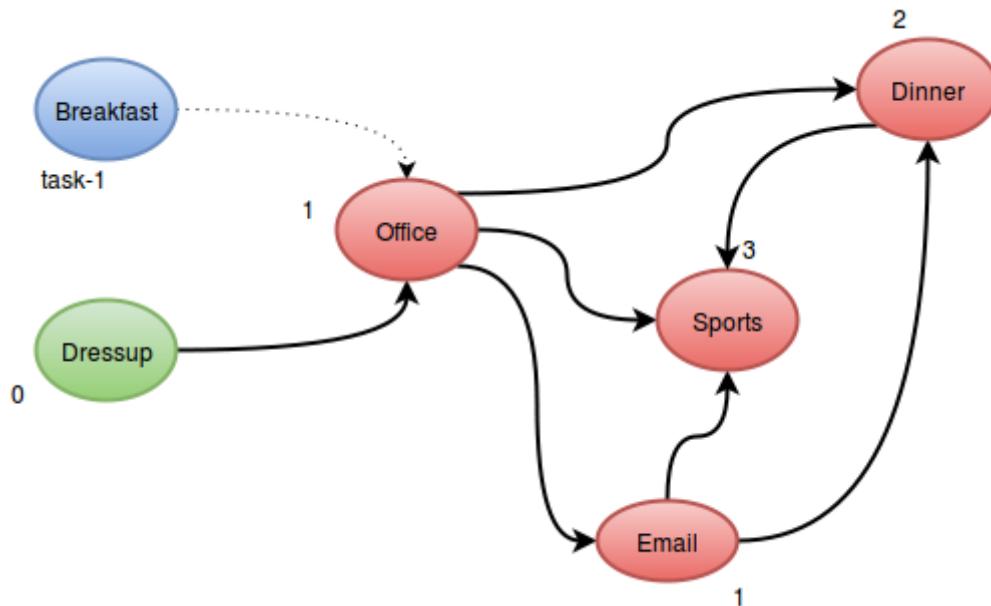


উপরের ছবিতে প্রতিটা কাজ একটি করে নোড দিয়ে দেখানো হয়েছে। ব্রেকফাস্ট থেকে অফিসের দিকে তীরচিহ্ন দিয়ে বুঝানো হচ্ছে যে অফিসে আসার আগে ব্রেকফাস্ট করতে হবে। উপরের ৮টি শর্ত ছবিতে ৮টি ডিরেক্টেড এজ দিয়ে দেখানো হয়েছে।

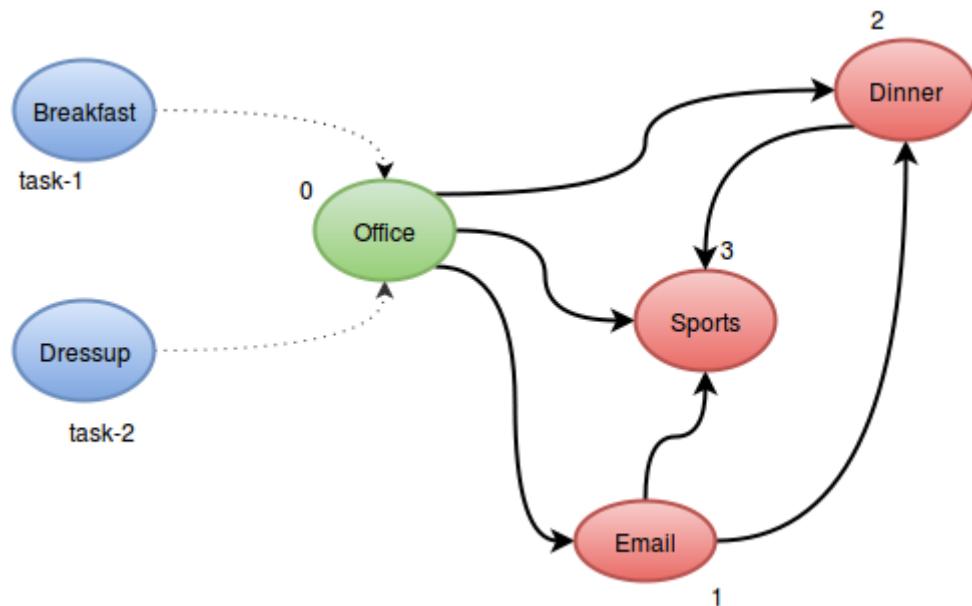
প্রতিটি নোডের পাশে ছোট করে কিছু সংখ্যা দেখতে পাচ্ছে। যেমন অফিসের সাথে ০, ডিনারের সাথে ২ ইত্যাদি। এগুলো দিয়ে বুঝাচ্ছে একটি কাজ অন্য কয়টি কাজের উপর নির্ভরশীল। যেমন ডিনারের আগে তোমাকে অফিস, ইমেইল এই ২টা

কাজ করতে হবে, দিনার নোডটিতে ২টি তীরচিহ্ন প্রবেশ করেছে, আর আমরা পাশে লিখে দিয়েছি “২”。 ঠিক এভাবে ইমেইলের পাশে লেখা হয়েছে ১। এ সংখ্যাগুলোকে indegree বলা হয়।

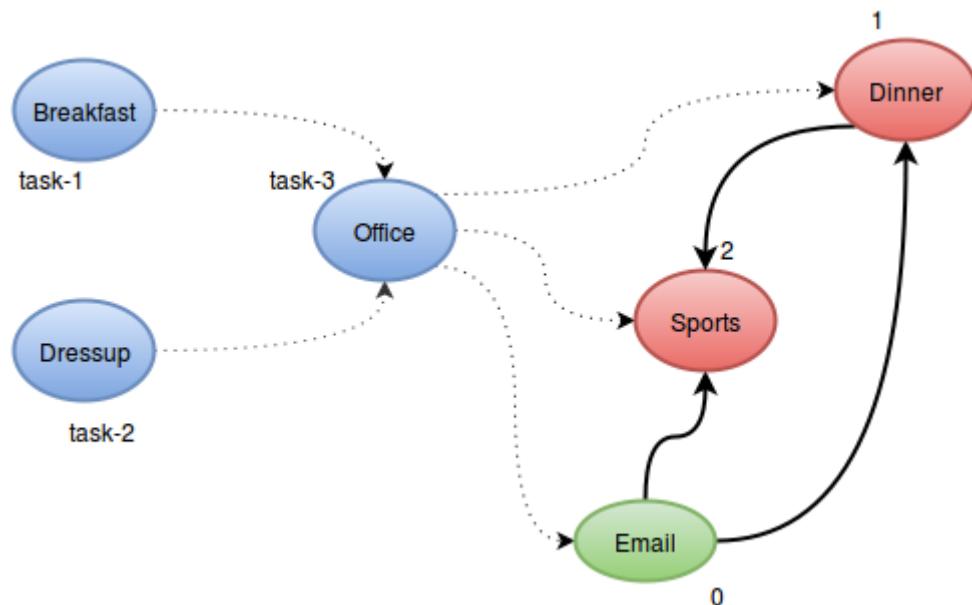
লক্ষ্য কর ব্রেকফাস্ট এবং ড্রেসআপ কোনো কাজের উপর নির্ভরশীল নয়, তাই তাদের পাশে ০ লেখা হয়েছে। তারমানে আমরা এ দুটি কাজের যেকোনোটা দিয়ে দিন শুরু করতে পারি। মনে করি তুমি নাস্তা আগে খেতে চাও। নাস্তা খেয়ে নেবার পর যেসব কাজ ব্রেকফাস্টের উপর নির্ভরশীল ছিল তারা আর সেটার উপর নির্ভরশীল থাকলোনা, গ্রাফটা হয়ে গেল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



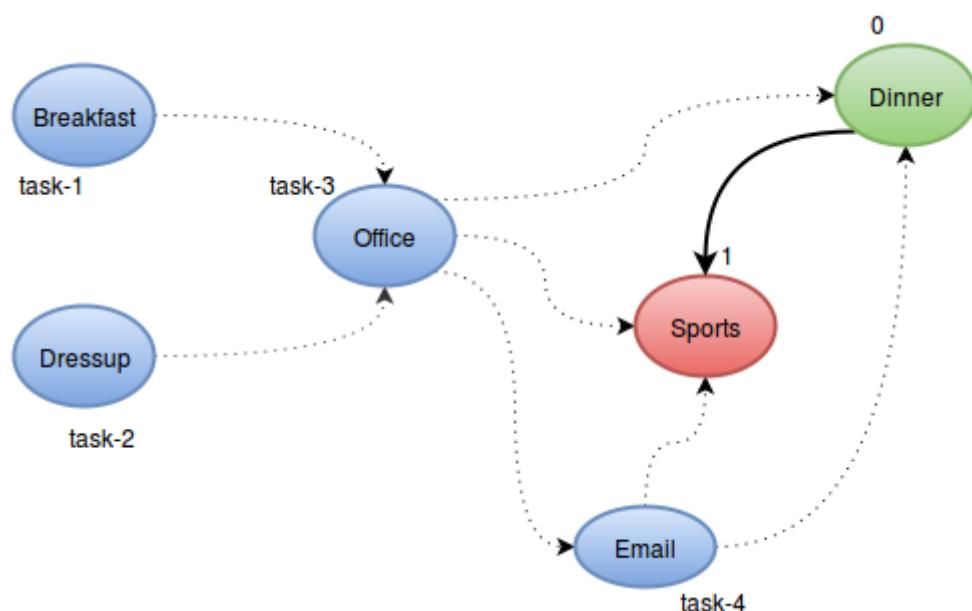
ব্রেকফাস্ট থেকে অফিসের তীরচিহ্ন সরিয়ে দিয়েছি। এখন অফিস আর মাত্র ১টি কাজের উপর নির্ভরশীল(আগে ছিল দুটির উপর)। এবার তোমাকে ড্রেসআপ করতে হবে, কারণ এখন একমাত্র এই কাজটিই কারো উপর নির্ভরশীল নয়:



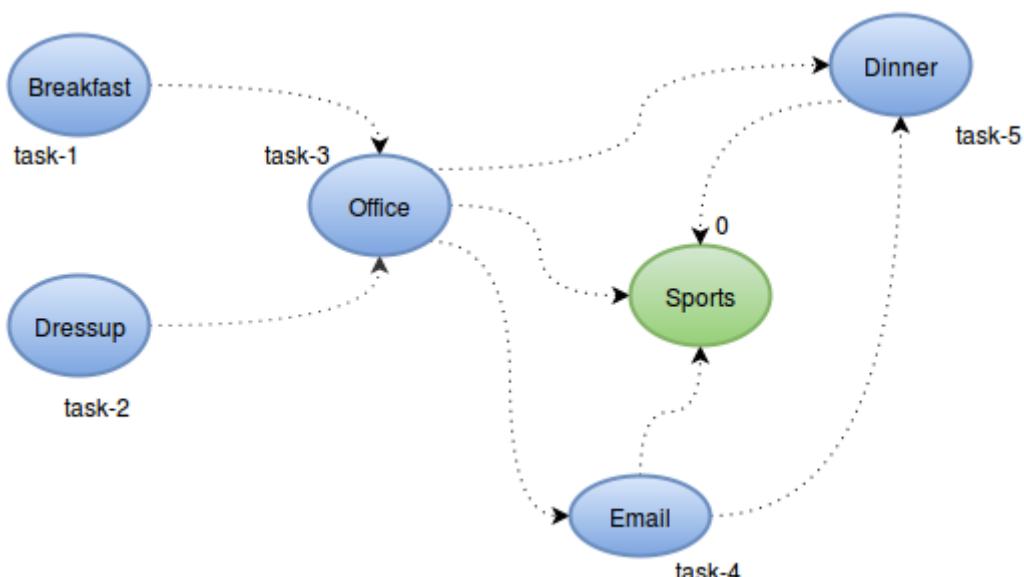
ড্রেসআপ থেকে অফিসের তীরচিহ্ন সরিয়ে দেয়া হয়েছে, আর কোনো কাজ নেই, এবার তুমি অফিসে ঘাবার জন্য প্রস্তুত। অফিসে ঘাবার উপর যারা নির্ভরশীল তাদের তীরচিহ্নগুলো এখন সরিয়ে দিতে পারি:

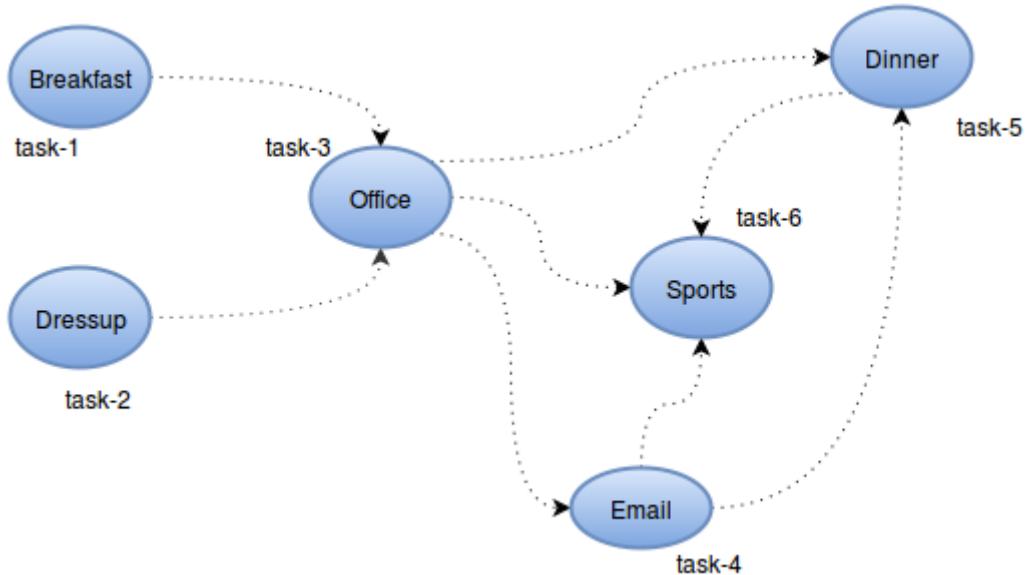


এখন ইমেইল “নোড” এর নির্ভরশীলতা ০ হয়ে গিয়েছে:



এরপর ডিনার:





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

এটাকেই বলা হয় টপোলজিক্যাল সর্ট (topological sort) বা টপসর্ট। মূলত কাজের অর্ডারিং খুজে 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 1 কমিয়ে দিবে, তারপর আবার খুজবে কার indegree এখন শূন্য। এক নোডকে কখনো 2বার 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)

[\(অন্যান্য পোস্ট\)](#)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# গ্রাফ থিওরিতে হাতেখড়ি ৮: ডেপথ ফাস্ট সার্চ এবং আবারো টপোলোজিকাল সর্ট

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

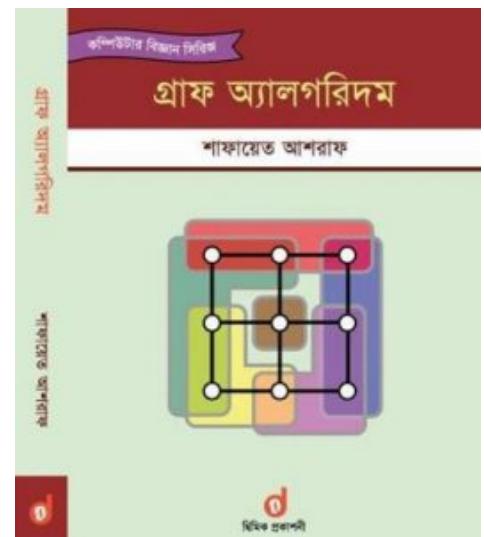
শাফায়েত

3/8/2012

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

**বিএফএস** এ আমরা গ্রাফটাকে লেভেল বাই লেভেল সার্চ করেছিলাম, নিচের ছবির মতো করে:

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 যেতে চেষ্টা করবো, এক নোড কখনো ২বার ভিজিট করবোনা। আমরা নোডের তৃতীয় রং(কালার) দিবো:



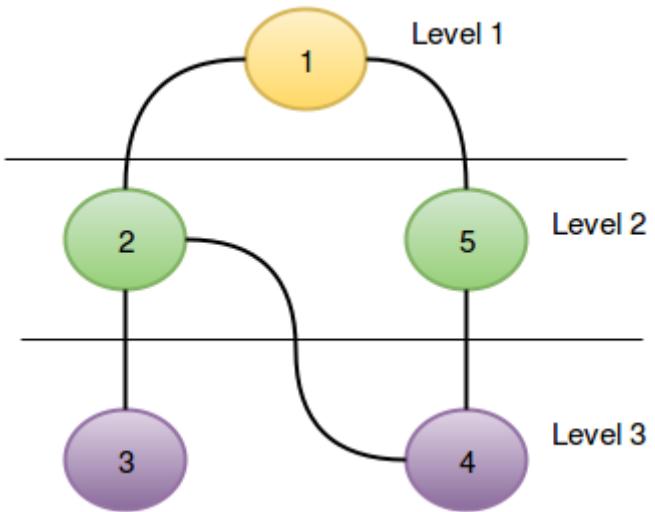
সাদা নোড = যে নোড এখনো খুঁজে পাইনি/ভিজিট করিনি।

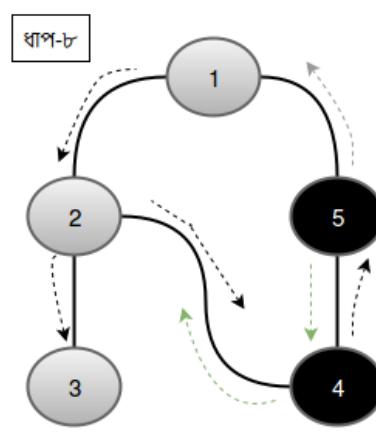
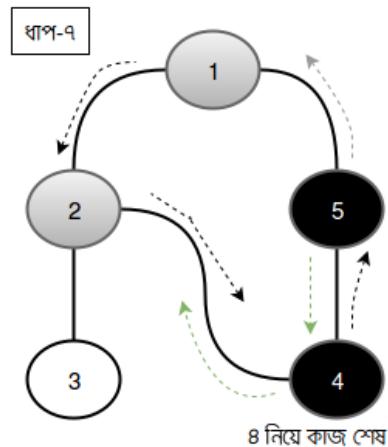
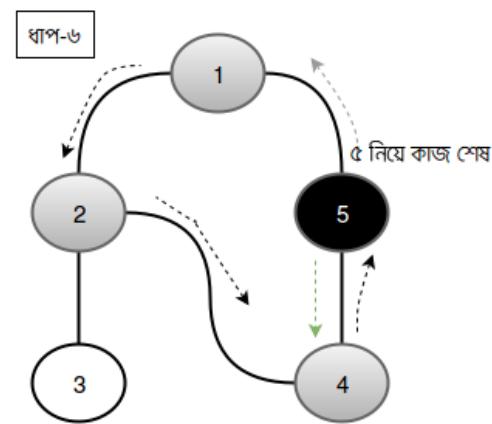
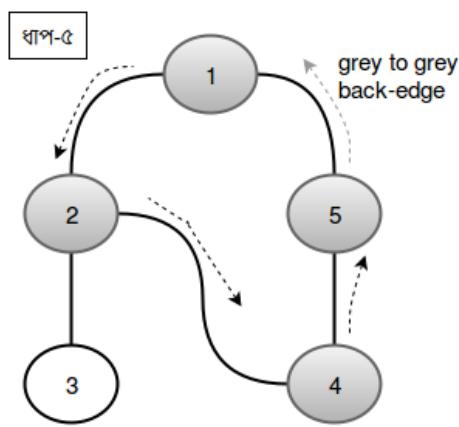
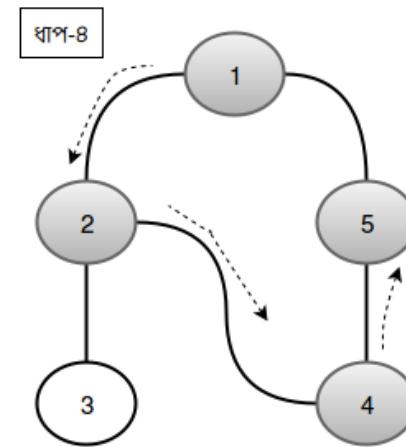
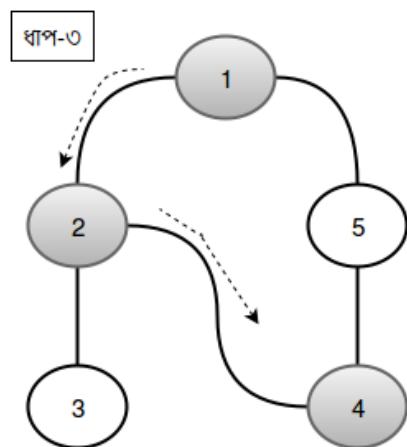
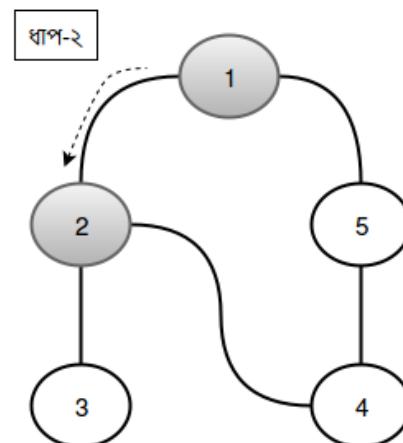
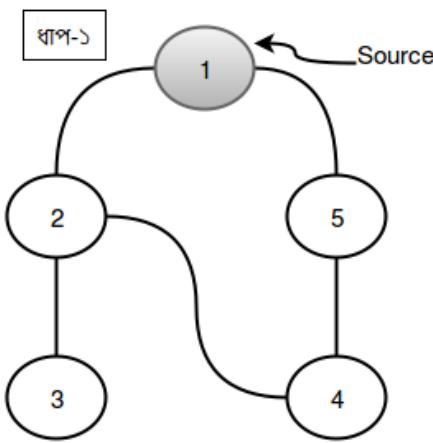
গ্রে বা ধূসর নোড = যে নোড ভিজিট করেছি কিন্তু নোডটি থেকে যেসব চাইল্ড নোডে যাওয়া যায় সেগুলো এখনো ভিজিট করে শেষ

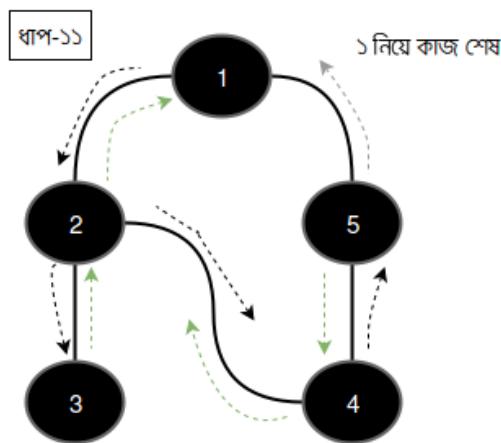
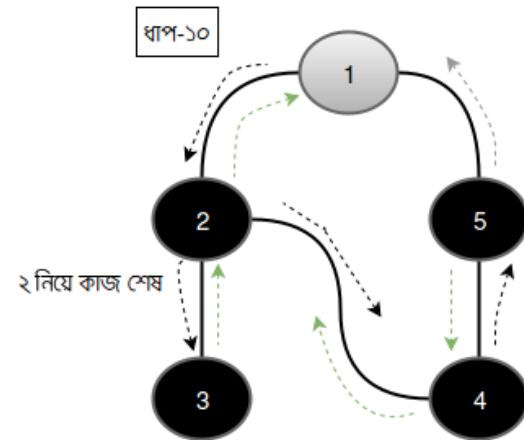
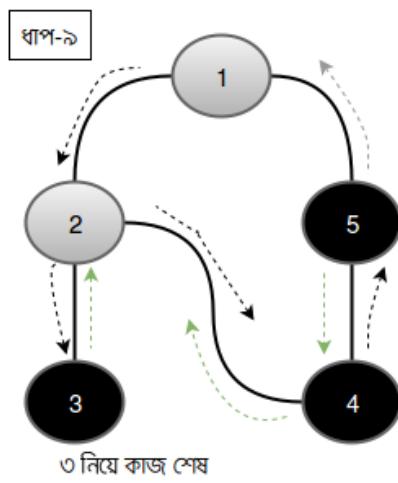
করিনি, অর্থাত নোডটিকে নিয়ে কাজ চলছে।

কালো নোড = যে নোডের কাজ সম্পূর্ণ শেষ।

এবার আমরা ধাপগুলো দেখতে পারি:



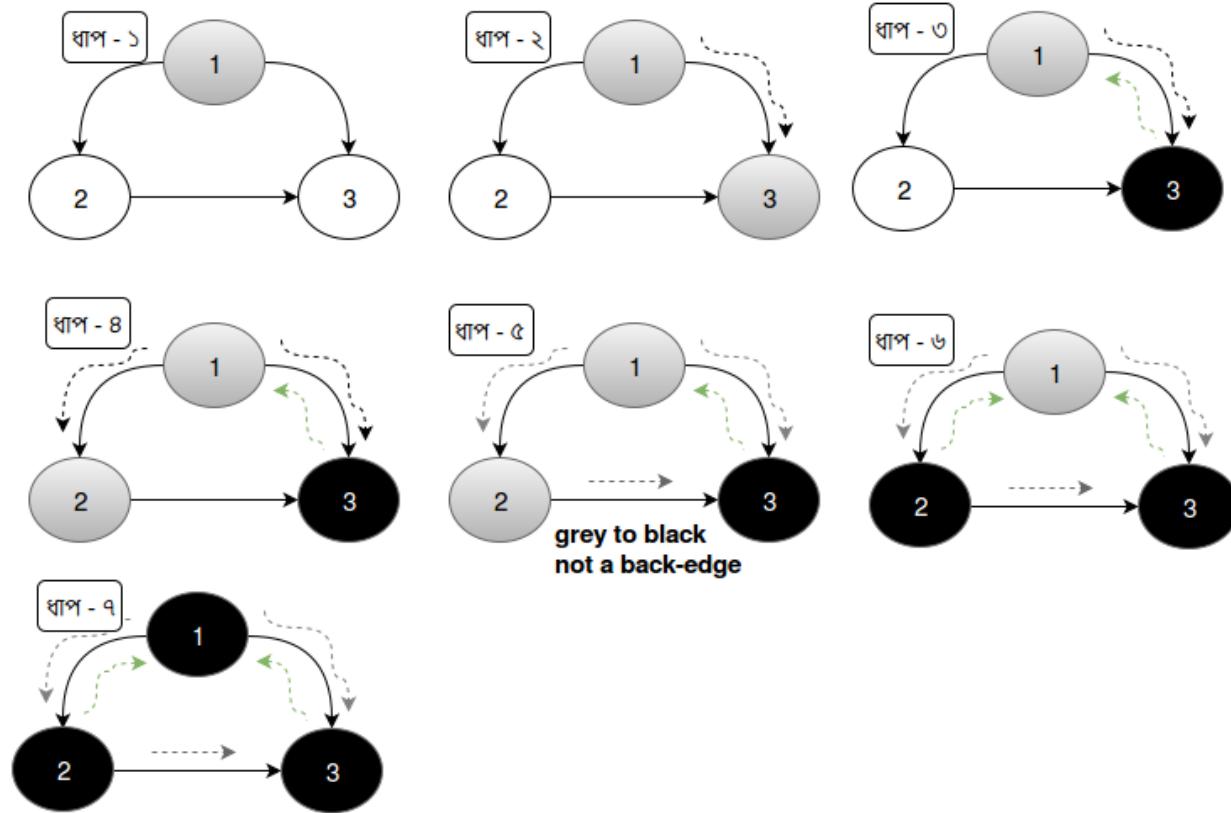




আশা করি ডিএফএস কিভাবে কাজ করে এটা পরিষ্কার, খুব সহজ জিনিস এটা। এবার আমরা একটা খুব গুরুত্বপূর্ণ টার্ম শিখবো, সেটা হলো ব্যাকএজ(backedge)। লক্ষ করো ৫-১ কে ব্যাকএজ বলা হয়েছে। এর কারণ হলো তখনও ১ এর কাজ চলছে, ৫ থেকে ১ এ যাওয়া মানে এমন একটা নোড ফিরে যাওয়া যাকে নিয়ে কাজ এখনো শেষ হয়নি, তারমানে অবশ্যই গ্রাফে 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 নোড ভিজিট করার অর্ডার 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 ট্রি এজ। শুধুমাত্র ট্রি এজ গুলো রেখে বাকি এজগুলো মুছে দিলে যে গ্রাফটা থাকে তাকে বলা হয় ডিএফএস ট্রি।

আনডিরেক্টেড গ্রাফের ক্ষেত্রে আগে ভিজিট করা কোনো নোডে ফিরে গেলেই সেটা ব্যকএজ, কালার চেক না করলেও হয়। তবে ডিরেক্টেড গ্রাফের ক্ষেত্রে অবশ্যই করতে হবে। পরের ছবিটা দেখো:



২-৩এর এজটাকে ব্যাকএজ বলা যাচ্ছেনা, কারণ ৩ এর কাজ আগেই শেষ হয়ে গেছে।

প্রতিটা নোড আর এজ নিয়ে একবার করে করছি, 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 সেগুলোই bfs দিয়ে করতে পারি। bfs এ সাধারণত টাইম কমপ্লেক্সিটি কম হয় তবে dfs কোডিং করতে খুব কম সময় লাগে। একটা সিম্পল dfs এর সুড়োকোড এরকম:

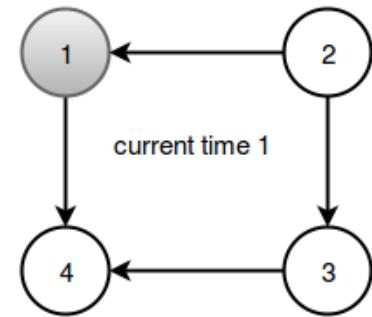
```

1 procedure DFS(G, source):
2   U ← source
3   time ← time+1
4   d[u] ← time
5   color[u] ← GREY
6   for all edges from u to v in G.adjacentEdges(v) do
7     if color[v] = WHITE
8       DFS(G,v)
9     end if
10   end for
11   color[u] ← BLACK
12   time ← time+1
13   f[u] ← time
14   return

```

$d[]$ =discovery time  
 $f[]$ =finishing time

| node | $d[]$ | $f[]$ |
|------|-------|-------|
| 1    | 1     | null  |
| 2    | null  | null  |
| 3    | null  | null  |
| 4    | null  | null  |



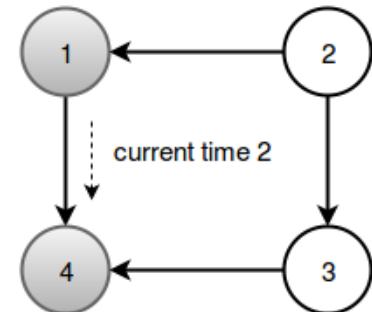
নিচের প্রবলেমগুলো সলভ করতে চেষ্টা করো:

<http://uva.onlinejudge.org/external/2/280.html>  
<http://uva.onlinejudge.org/external/115/11518.html>  
<http://uva.onlinejudge.org/external/104/10452.html>

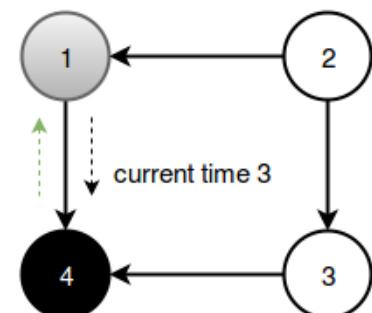
এরপরে [এই আর্টিকেলটা](#) পড়ে ফেলো বিস্তারিত জানার জন্য, আমার লেখা পড়ে তুমি বেসিকটা শিখতে পারবে, বিস্তারিত জানতে এবং কঠিন প্রবলেম সলভ করতে আরো অনেক কিছু জানতে হবে।

AccessPress Staple | WordPress Theme:  
[AccessPress Staple](#) by [AccessPress Themes](#)

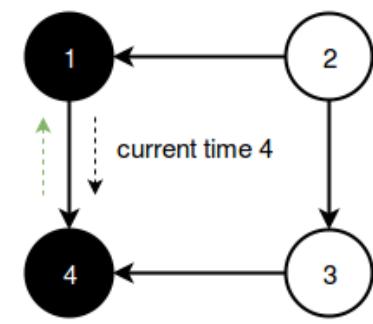
| node | $d[]$ | $f[]$ |
|------|-------|-------|
| 1    | 1     | null  |
| 2    | null  | null  |
| 3    | null  | null  |
| 4    | 2     | null  |



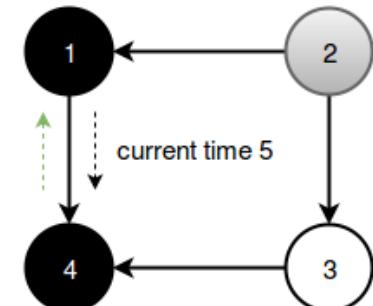
| node | $d[]$ | $f[]$ |
|------|-------|-------|
| 1    | 1     | null  |
| 2    | null  | null  |
| 3    | null  | null  |
| 4    | 2     | 3     |



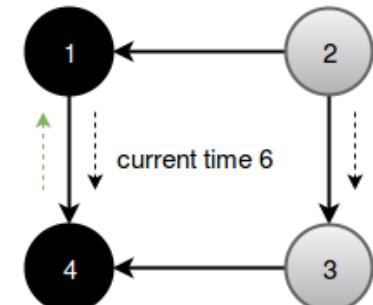
| node | d[]  | f[]  |
|------|------|------|
| 1    | 1    | 4    |
| 2    | null | null |
| 3    | null | null |
| 4    | 2    | 3    |



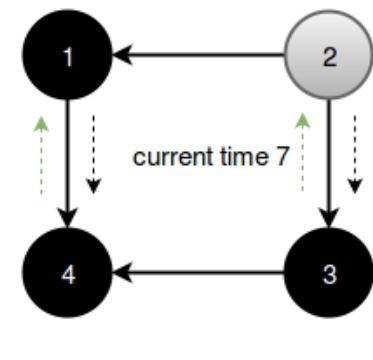
| node | d[]  | f[]  |
|------|------|------|
| 1    | 1    | 4    |
| 2    | 5    | null |
| 3    | null | null |
| 4    | 2    | 3    |



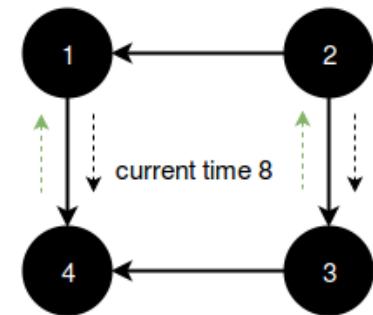
| node | d[] | f[]  |
|------|-----|------|
| 1    | 1   | 4    |
| 2    | 5   | null |
| 3    | 6   | null |
| 4    | 2   | 3    |



| node | d[] | f[]  |
|------|-----|------|
| 1    | 1   | 4    |
| 2    | 5   | null |
| 3    | 6   | 7    |
| 4    | 2   | 3    |



| node | d[] | f[] |
|------|-----|-----|
| 1    | 1   | 4   |
| 2    | 5   | 8   |
| 3    | 6   | 7   |
| 4    | 2   | 3   |



ফিনিশিং টাইম অনুযায়ী সর্ট করে পাই: ২,৩,১,৪

# গ্রাফ থিওরিতে হাতেখড়ি-৯ (ডায়াক্স্ট্র্যাট্রা)

 shafaetsplanet.com/planetcoding/

শাফায়েত

এপ্রিল ৯, ২০১৩

আমরা শুরুতেই শিখেছি কিভাবে শর্টেস্ট পাথে এক জায়গা থেকে আরেক জায়গায় যেতে হয়। সেজন্য আমরা শিখেছি **ব্রেডথ ফার্স্ট সার্চ** নামের একটি সার্টিং অ্যালগোরিদম। অ্যালগোরিদমটি চমৎকার 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 ডায়াক্স্ট্র্যাট্রা। প্রথমে নাম শুনে আমার ধারণা হয়েছিলো ডায়াক্স্ট্র্যাট্রা খুবই ভয়ংকর কোনো জিনিস কিন্তু আসলে বিএফএস লেখার মতোই সহজ ডায়াক্স্ট্র্যাট্রা লেখা, আমি তোমাদের দেখানোর চেষ্টা করবো কিভাবে বিএফএসকে কিছুটা পরিবর্তন করে একটা প্রয়োরিটি কিউ যোগ করে সেটাকে ডায়াক্স্ট্র্যাট্রা বানিয়ে ফেলা যায়।

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

ধরো তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসি তে গেলে 10 মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে 25 মিনিটে। তাহলে তোমার কাছে ইনফরমেশন আছে:

$$d[\text{টিএসসি}] = 10, d[\text{কার্জনহল}] = 25$$

এখন তুমি দেখলে টিএসসি থেকে 7 মিনিটে কার্জনে চলে যাওয়া যায়,

$$cost[\text{টিএসসি}/\text{কার্জন}] = 7$$

তাহলে তুমি 25 মিনিটের জায়গায় মাত্র  $10 + 7 = 17$  মিনিটে কার্জনহলে যেতে পারবে। যেহেতু তুমি দেখছো:

$$d[\text{টিএসসি}] + cost[\text{টিএসসি}/\text{কার্জন}] < d[\text{কার্জনহল}]$$

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

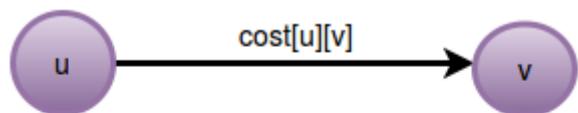
উপরের ছবিটা সেটাই বলছে। আমরা  $u$  থেকে  $v$  তে যাবো যদি  $d[u] + cost[u][v] < d[v]$  হয়। আর  $d[v] = d[u] + cost[u][v]$  কে আপডেট করে  $d[v] = d[u] + cost[u][v]$  বানিয়ে দিবো। ভবিষ্যতে যদি কার্জনহলে অন্য রাস্তা দিয়ে আরো কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে কম্পেয়ার করে আপডেট করি দিবো। ব্যাপারটা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

C++

```
1 if d[u] + cost[u][v] < d[v]:
2   d[v] = d[u] + cost[u][v]
```

উপরের অংশটা যদি বুঝে থাকো তাহলে ডায়াক্স্ট্র্যাট্রা বোৰাৰ 60% 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



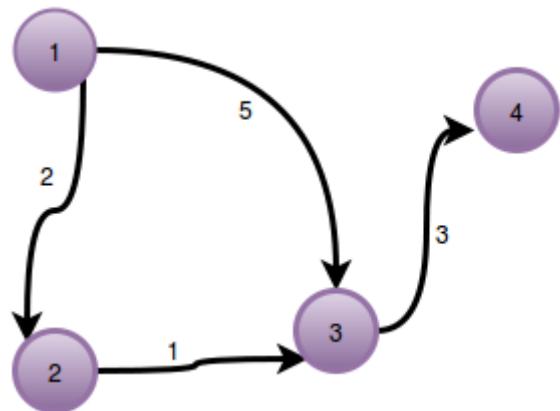
if  $d[u] + cost[u][v] < d[v]$   
 $d[v] = d[u] + cost[u][v]$

ডিস্টেন্স ১ বাড়িয়ে দিয়েছি। ডায়াক্রস্ট্র্যাক্স এমন 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 সেগুলা আপডেট করার চেষ্টা করো, আপডেট করতে পারলে কিউতে পুশ করো। যেমন ১ – ২ এজটা ধরে আমরা আগবো কারণ  $d[1]+2 < d[2]$  এই শর্তটা পূরণ হচ্ছে। তখন  $d[2]=2$  হয়ে যাবে ২, একই ভাবে ১ থেকে ৩ এ গেলে  $d[3]=3$  হয়ে যাবে ৩।



কিন্তু ৫ তো ৩ নম্বরনোডে যাওয়ার শর্টেস্ট ডিস্টেন্স 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  $d[3]=3$  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 সুড়েকোডের আপডেট অংশ একটু পরিবর্তন করি যাতে একটা নোড বার বার আপডেট হতে পারে:

```

1 1 procedure BFSmodified(G,source):
2 2   Q = queue(), distance[] = infinity
3 3   Q.enqueue(source)
4 4   distance[source] = 0
5 5   while Q is not empty
6 6     u ← Q.pop()
7 7     for all edges from u to v in G.adjacentEdges(v) do
8 8       if distance[u] + cost[u][v] < distance[v]
9 9         distance[v] = distance[u] + cost[u][v]
10 10        Q.enqueue(v)
11 11     end if
12 12   end for
13 13 end while
14 14 Return distance

```

আমরা ঠিক আগের বিএফএস এর কোডেই জাস্ট কস্ট 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 প্রয়োরিটি 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 গেলে 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 ডিস্টেন্সকে আপডেট করে দিচ্ছি ডিস্টেন্স  $5 + 3 = 8$  হিসাবে। পরবর্তীতে যখন ২ দিয়ে ৩ কে আবার আপডেট করা হচ্ছে তখন ৩ এর ডিস্টেন্স হয়ে গিয়েছে ৩, এবার ৪ এর ডিস্টেন্সকে আবার আপডেট করছি  $3 + 3 = 6$  হিসাবে। ৪ কে মোট দুইবার আপডেট করা লাগলো।

বিজ্ঞানী ডায়াক্রস্ট্র্যাক্স চিন্তা করলেন যদি এই “আগে আসলে আগে পাবেন” ভিত্তিতে কাজ না করে সবথেকে কাছের 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 ডিস্টেন্স পেয়ে যেতাম! একটু হাতে কলমে সিমুলেট করে দেখো। আইডিয়াটা হলো যেকোনো সময় কিউতে 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 নাস্বার পুশ করেছি, এখন বর্তমান ডিস্টেন্স অর্থাত  $d[u]d[u]$  এর মানটাও পুশ করতে হবে।

নিচে একটা সম্পূর্ণ ডায়াক্রস্ট্র্যাট্রি সুড়োকোড দিলাম যেটা ১ থেকে  $nn$  তম নোডে যাবার শর্টেস্ট পাথ বের করে এবং পাথটাও প্রিন্ট করে:

C++

```

1   1 procedure dijkstra(G, source):
2   2     Q = priority_queue(), distance[] = infinity
3   3     Q.enqueue(source)
4   4     distance[source]=0
5   5     while Q is not empty
6   6       u = nodes in Q with minimum distance[]
7   7       remove u from the Q
8   8       for all edges from u to v in G.adjacentEdges(v) do
9   9         if distance[u] + cost[u][v] < distance[v]
10 10           distance[v] = distance[u] + cost[u][v]
11 11           Q.enqueue(v)
12 12       end if
13 13     end for
14 14   end while
15 15   Return distance

```

এখানে আগের সুড়োকোডের থেকে কয়েক জায়গায় একটু ভিন্নতা আছে। এখানে সাধারণ কিউ এর জায়গায় প্রায়োরিটি কিউ ব্যবহার করা হয়েছে। কিউ থেকে পপ হবার সময় তাই সোর্স থেকে এখন পর্যন্ত পাওয়া সবথেকে কাছের নোডটা পপ হচ্ছে এবং সেটা নিয়ে আগে কাজ করছি।

উপরের সুড়োকোডে সোর্স থেকে বাকি সব নোডের দূরত্ব বের করা হয়েছে। তুমি যদি শুধু একটা নোডের দূরত্ব বের করতে চাও তাহলে সেটা যথন কিউ থেকে পপ হবে তখনই রিটার্ন করে দিতে পারো।

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

### কমপ্লেক্সিটি:

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

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।

ডায়াক্রস্ট্র্যাট্রি ভালো করে শিখতে নিচের প্রবলেমগুলো ঝটপট করে ফেলো:

[Dijkstra?](#)[Not the Best](#)[New Traffic System](#)

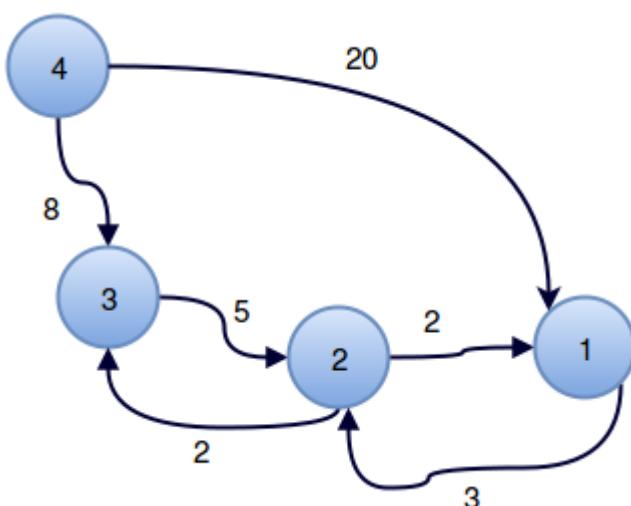
হ্যাপি কোডিং!

(গ্রাফ থিওরি নিয়ে সবগুলো লেখা)



ফ্লয়েড ওয়ার্শল সম্বিত 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 সংক্ষিপ্তম দূরত্ব বের করা। এ ধরণের অ্যালগোরিদমকে বলা হয় “অল-পেয়ার শর্টেস্ট পাথ” অ্যালগোরিদম। এই লেখাটা পড়ার আগে [অ্যাডজেসেন্সি ম্যাট্রিক্স](#) সম্পর্কে জানতে হবে।

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

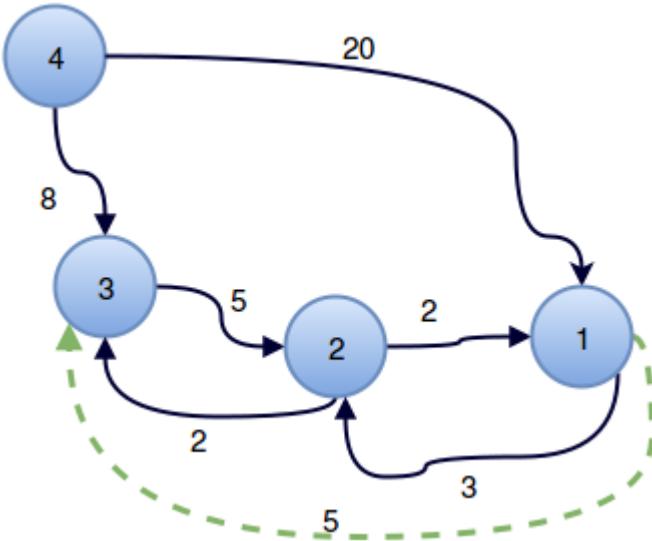


| Node | 1   | 2   | 3   | 4   |
|------|-----|-----|-----|-----|
| 1    | 0   | 3   | inf | inf |
| 2    | 2   | 0   | 2   | inf |
| 3    | inf | 5   | 0   | inf |
| 4    | 20  | inf | 8   | 0   |

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 “মাঝের নোড” হিসাবে ধরলাম। মাঝের নোডকে আমরা বলবো  $k$ । তারমানে এখন  $k=2$ । (আমরা একে একে সব নোডকেই মাঝের নোড হিসাবে ধরবো, এটা যেকোন অর্ডারে করা যায়)

এখন যেকোন এক জোড়া নোড  $(i,j)$  নাও। ধরি  $i=1, v=3$ । 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  $\text{matrix}[1][3]=\text{ইনফিনিটি}$ । আর যদি  $k=2$  কে মাঝখানে রাখি তাহলে দূরত্ব দাঢ়াবে  $\text{matrix}[1][2] + \text{matrix}[2][3] = 3 + 2 = 5$ । তারমানে কস্ট কমে যাচ্ছে! আমরা গ্রাফটা আপডেট করে দিতে পারি এভাবে:



| Node | 1   | 2   | 3 | 4   |
|------|-----|-----|---|-----|
| 1    | 0   | 3   | 5 | inf |
| 2    | 2   | 0   | 2 | inf |
| 3    | inf | 5   | 0 | inf |
| 4    | 20  | inf | 8 | 0   |

আমরা ২ কে “মাঝের 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।

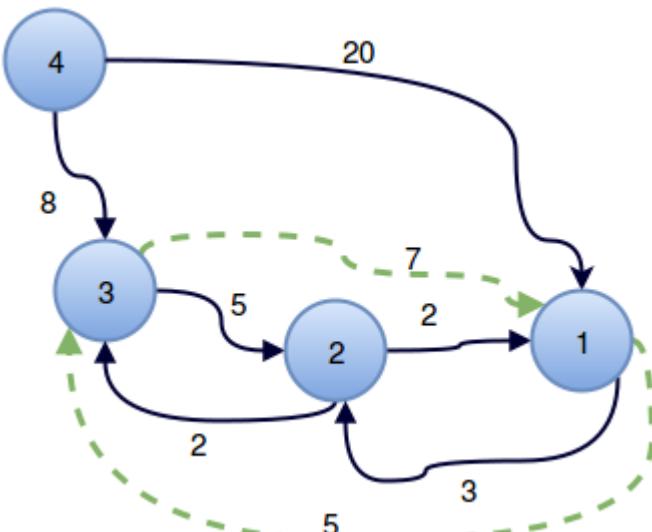
এখন আবার ধর  $i=2, j=4$ । আর আগের মতই  $k=2$ । এবার  $\text{matrix}[2][4]=\text{ইনফিনিটি}$ । এদিকে  $\text{matrix}[2][2] + \text{matrix}[2][4] = 0 + \text{ইনফিনিটি}$ । এবার কিন্তু দূরত্ব কমলো না। তাই গ্রাফ আপডেট করার দরকার নেই। নিচয়েই বুঝতে পারছো আপডেটের শর্তটা হবে এরকম:

```
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])
```

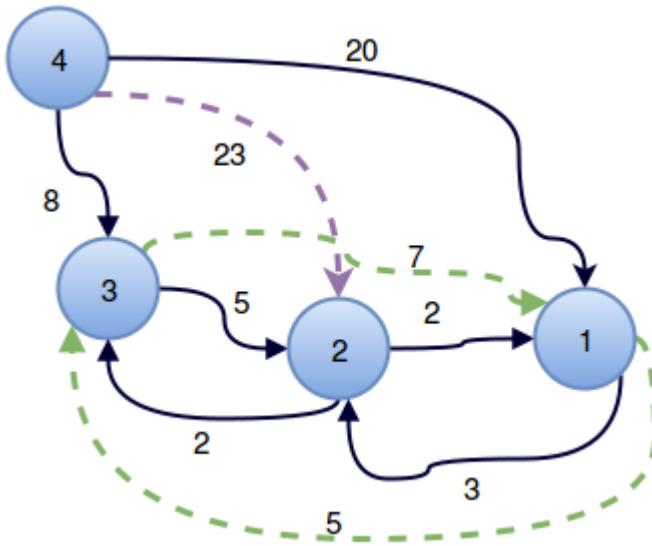
এখন  $k=2$  স্থির রেখে আমরা  $i,j$  এর সবরকমের কম্বিনেশন নিবো। তুমি নিজেই চিন্তা করলে দেখবে  $k=2$  এর জন্য  $i=3, j=1$  এই কম্বিনেশনে আমরা আরেকটা নতুন এজ পাবো, বাকি গ্রাফ আগের মতই থাকবে।



| Node | 1  | 2   | 3 | 4   |
|------|----|-----|---|-----|
| 1    | 0  | 3   | 5 | inf |
| 2    | 2  | 0   | 2 | inf |
| 3    | 7  | 5   | 0 | inf |
| 4    | 20 | inf | 8 | 0   |

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

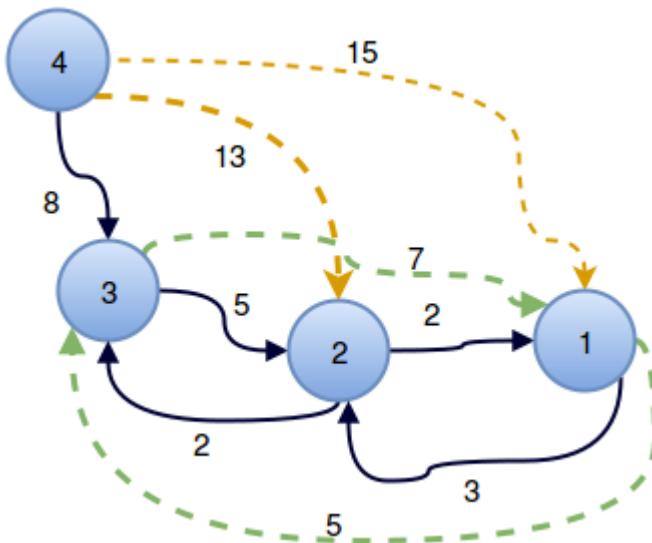
এবার একটা মাত্র এজ যোগ হবে।  $i=4, j=2$  হলে আমরা 8 থেকে 1 হয়ে 2 নম্বর নোডে যেতে পারি ২৩ কস্ট এ। তাহলে গ্রাফটা হবে এরকম:



| Node | 1  | 2  | 3 | 4   |
|------|----|----|---|-----|
| 1    | 0  | 3  | 5 | inf |
| 2    | 2  | 0  | 2 | inf |
| 3    | 7  | 5  | 0 | inf |
| 4    | 20 | 23 | 8 | 0   |

এবার 3 নম্বর নোডকে মাঝের নোড হিসাবে ধরবো। আবারো নিজে চেষ্টা করে তারপর নিচের অংশ দেখ।

এবার কিছু মজার জিনিস ঘটবে। 8 থেকে 1 এ আগে লাগছিল ২০ কস্ট। এখন 3 কে মাঝে রেখে 8 থেকে 1 এ গেলে লাগবে  $8+7=15$  কস্ট। লক্ষ্য কর একদম শুরুতে 3 থেকে 1 এ আমাদের এজ ছিল না। কিন্তু আপডেট করার সময় আমরা এজ বসিয়ে দিয়েছি, এখন 3 থেকে 1 এ যদিও সরাসরি চলে যাচ্ছি, মূল গ্রাফে আসলে  $3 \rightarrow 2 \rightarrow 1$  পথে যাচ্ছ। একই ভাবে 8 থেকে 2 এর ২৩ কস্ট এর পথটা আপডেট হয়ে ১৩ হয়ে যাবে।



| Node | 1  | 2  | 3 | 4   |
|------|----|----|---|-----|
| 1    | 0  | 3  | 5 | inf |
| 2    | 2  | 0  | 2 | inf |
| 3    | 7  | 5  | 0 | inf |
| 4    | 15 | 13 | 8 | 0   |

$k=8$  এর জন্য আর কোন আপডেট হবে না কারণ কোনো 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 কিভাবে 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 ফিল্ড রেখে next অ্যারে ধরে আগাচ্ছি যতক্ষণ না j তে পৌছাচ্ছি। তাহলে আমরা পেয়ে গেলাম পাথ!

### ট্রান্সিভিভ ক্লোজার(Transitive Closure):

ধরো আমাদের অ্যাডজেন্সি ম্যাট্রিক্সটা এরকম:

- 1 matrix[i][j] = 1 যদি i থেকে j তে সরাসরি এজ থাকে
- 2 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 o এর জায়গায় ইনফিনিটি দিয়ে শর্টেস্ট পাথ বের করে কাজটা করতে পারতাম। কিন্তু এক্ষেত্রে “OR” আর “AND” অপারেশন ব্যবহার আরো দ্রুত কাজটা করা যায়। এখন আপডেটের শর্টটা হয়ে যাবে এরকম:

---

```
1 matrix[i][j] = matrix[i][j] || (matrix[i][k] && matrix[k][j])
```

---

এটার মানে  $\text{matrix}[i][j]$  তে তখনই ১ বসবে যখন হয় “ $\text{matrix}[i][j]$  তে ১ আছে” অথবা “ $\text{matrix}[i][k]$  এবং  $\text{matrix}[k][j]$ ” দুটোতেই ১ 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  $O(n^3)$ । ২ড়ি 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 ওয়ার্শল কাজ করবে কি?

### রিলেটেড প্রবলেম:

[Page Hopping](#)

[05-2 Rendezvous](#)

[Minimum Transport Cost](#)

[Asterix and Obelix](#)

[আরো অনেক প্রবলেম](#)

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# গ্রাফ থিওরিতে হাতেখড়ি ১১: বেলম্যান ফোর্ড

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

অক্টোবর ১৭, ২০১৮

বেলম্যান ফোর্ড গ্রাফে শর্টেস্ট পাথ বের করার একটা অ্যালগোরিদম। এই অ্যালগোরিদম একটা নোডকে সোর্স ধরে সেখান থেকে সব নোডের সংক্ষিপ্তম 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 ডায়াক্স্ট্রা অ্যালগোরিদম। এখন বেলম্যান ফোর্ড শিখব কারণ আগের কোনো অ্যালগোরিদমই নেগেটিভ ওয়েট এর এজ আছে এমন গ্রাফে কাজ করে না।

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

## এজ রিল্যাক্সেশন:

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

ধর তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসি তে গেলে 10 মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে 25 মিনিটে। তাহলে ফার্মগেটকে সোর্স ধরে আমরা বলতে পারি:

$$d[\text{টিএসসি}] = 10, d[\text{কার্জন হল}] = 25$$

এখন তুমি দেখলে টিএসসি থেকে 7 মিনিটে কার্জনে চলে যাওয়া যায়,

$$cost[\text{টিএসসি}/\text{কার্জন হল}] = 7$$

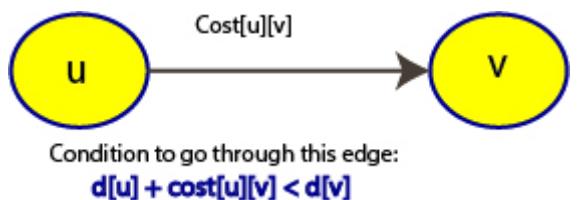
তাহলে তুমি 25 মিনিটের জায়গায় মাত্র  $10 + 7 = 17$  মিনিটে কার্জ নহলে যেতে পারবে। যেহেতু তুমি দেখেছো:

$$d[\text{টিএসসি}]+cost[\text{টিএসসি}/\text{কার্জন হল}] < d[\text{কার্জন হল}]$$

তাই তুমি এই নতুন রাস্তা দিয়ে কার্জন হলে গিয়ে  $d[\text{কার্জন হল}] = d[\text{টিএসসি}] + cost[\text{টিএসসি}/\text{কার্জন হল}]$  বানিয়ে দিতেই পারো!!

উপরের ছবিটা সেটাই বলছে। আমরা  $uu$  থেকে  $vv$  তে যাবো যদি  $d[u]+cost[u][v] < d[v]$  হয়। আর  $d[v]=d[u]+cost[u][v]$  কে আপডেট করে  $d[v]=d[u]+cost[u][v]+cost[u][v]=d[u]+2cost[u][v]$  বানিয়ে দিবো। ভবিষ্যতে যদি কার্জনহলে অন্য রাস্তা দিয়ে আরো কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে কম্পেয়ার করে আপডেট করি দিবো। ব্যাপারটা অনেকটা এরকম:

C++



```

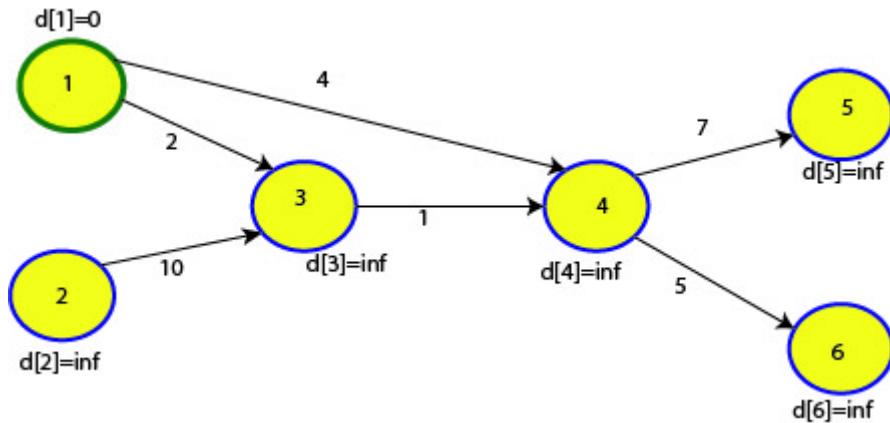
1   if(d[u]+cost[u][v] < d[v])
2   d[v] = d[u] + cost[u][v];

```

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

## বেলম্যান ফোর্ড

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

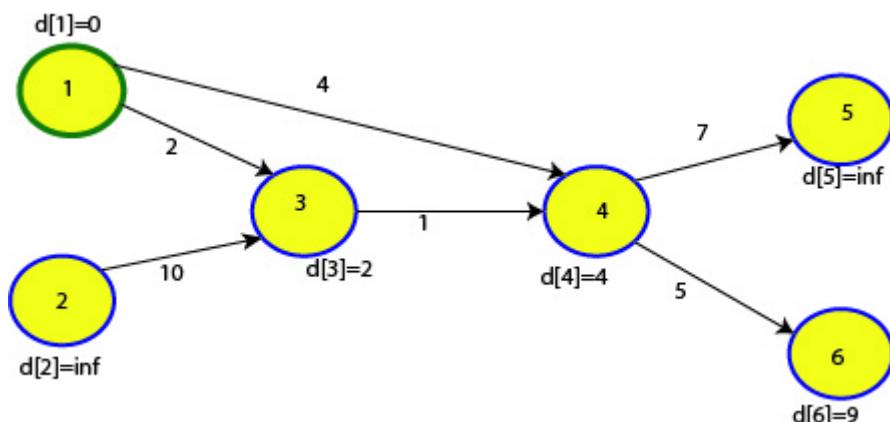


শুরুতে  $d[1]=0$  কারণ ১ হলো সোর্স। বাকিসবগুলোতে ইনফিনিটি রেখেছি কারণ আমরা এখনও জানিনা শর্টেস্ট পাথের কস্ট কত।

তুমি এজ রিল্যাক্স কিভাবে করতে হয় এই মধ্যে 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,

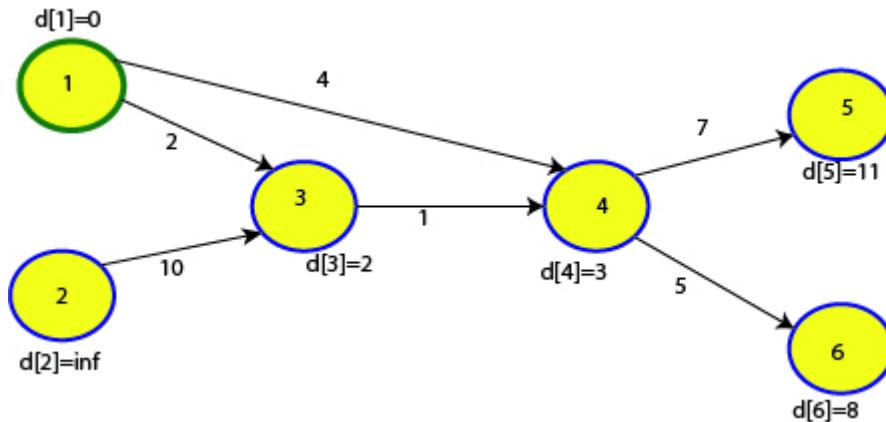
| Serial | 1      | 2      | 3      | 4      | 5      | 6      |
|--------|--------|--------|--------|--------|--------|--------|
| Edge   | 4 -> 5 | 3 -> 4 | 1 -> 3 | 1 -> 4 | 4 -> 6 | 2 -> 3 |

তুমি চাইলে অন্য যেকোনো অর্ডারেও এজগুলো নিতে পারতে। এখন চিন্তা কর এজগুলোকে একবার রিল্যাক্স করলে আমরা  $d[]$  অ্যারেতে কি পাব? সোর্স থেকে শুরু করে সর্বোচ্চ ১টা এজ ব্যবহার করে অন্যান্য নোডে ঘাবার শর্টেস্ট পাথের কস্ট আমরা পেয়ে ঘাব। 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  $d[u] + \text{cost}[u][v] < d[v]$  শর্তটা পূরণ করে নি। বাকি এজগুলো আপডেট করার পর  $d[]$  অ্যারের এর মান উপরের ছবির মত পেয়েছি। ১ নম্বর নোড থেকে শুরু করে সর্বোচ্চ একটি এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথ এখন আমরা জানি!

এখন সর্বোচ্চ ২টা এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথের cost 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 ( $1 \rightarrow 3$ ,  $3 \rightarrow 4$ ,  $4 \rightarrow 6$ ) এবং পথের দৈর্ঘ্য  $2+1+5=8$ । মাত্র ২বার রিল্যাক্স করলেও আমরা এখনই  $d[6]$  তে ৮ পেয়ে গেছি, অথচ আমাদের এখন সর্বোচ্চ ২টা এজ ব্যবহার করে শর্টেস্ট পাথের cost পাবার কথা। এটা নির্ভর করে তুমি কোন অর্ডারে এজ রিল্যাক্স করেছ তার উপর। সে কারণে ৫ 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 নিশ্চিতভাবে 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।

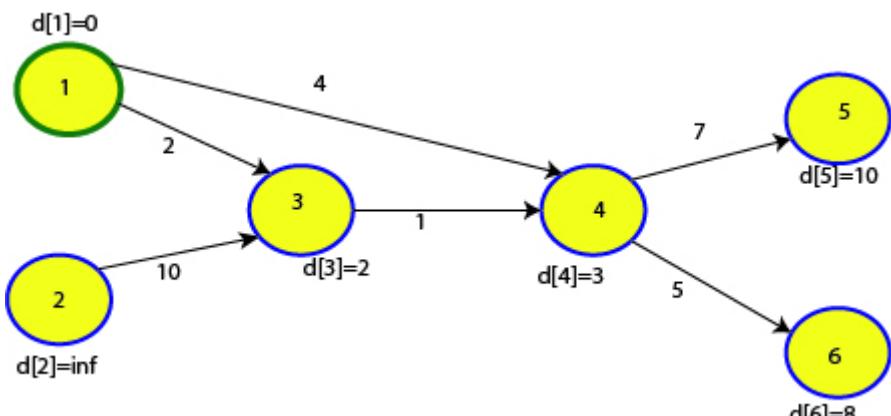
এরপরে আমরা আর যতই আপডেট করি,  $d[]$  অ্যারেতে কোনো পরিবর্তন হবে না, আমরা ১ থেকে প্রতিটা নোডে যাবার শর্টেস্ট পাথ পেয়ে গিয়েছি।

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 তুরারের 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   | $2 \rightarrow 3$ | $1 \rightarrow 2$ | $3 \rightarrow 1$ |



প্রথমবার রিল্যাক্স করার পর পাব:

২য়বার করার পর:

৩ নোডের গ্রাফে সোর্স থেকে কোনো নোডে শর্টেস্ট 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  $1 \rightarrow 2 \rightarrow 3 \rightarrow 11 \rightarrow 2 \rightarrow 3 \rightarrow 1$  সাইকেলটার মোট ওয়েট নেগেটিভ ( $3+2-10=-5$ )। তাই তুমি যতবার এই সাইকেলে ঘূরবে শর্টেস্ট পাথ তত ছোট হতে থাকবে। তাই নেগেটিভ সাইকেল থাকলে এবং সোর্স থেকে সেই নেগেটিভ সাইকেলে যাবার পথ থাকলে সোর্সের শর্টেস্ট পাথ আনডিফাইনড বা অসংজ্ঞায়িত। যদি  $n-1$  বার গ্রাফ রিল্যাক্স করার পর দেখি যে  $n$  তম বারও কোনো নোডের cost আপডেট করা যায় তখন বুঝতে হবে আমরা নেগেটিভ সাইকেলে গিয়ে পরেছি, শর্টেস্ট পাথ বের করা সম্ভব না।

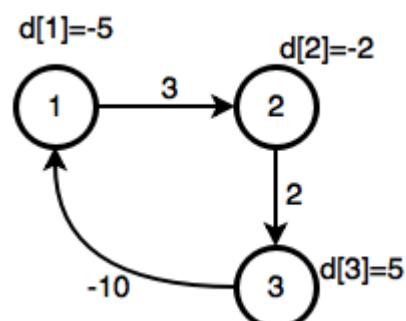
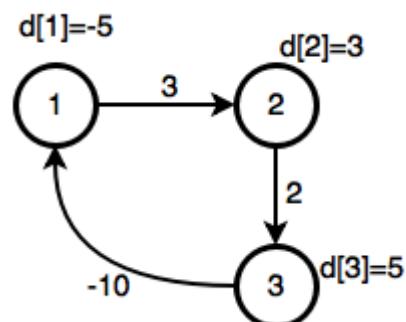
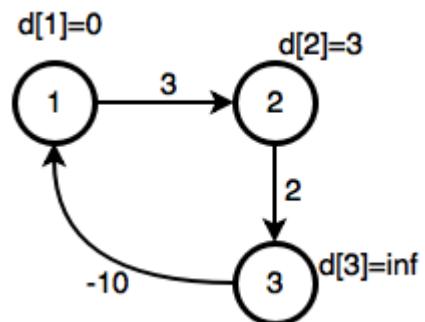
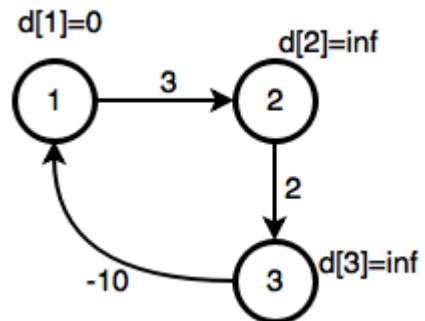
আমাদের সুভাবকোড তাহলে হবে এরকম:

bellman

```

1 Input: A non-empty connected weighted graph G with vertices G.V
2 and edges G.E
3 procedure Bellman Ford(G,source):
4   1 Let distance[] ← infinity
5   2 Let N ← number of nodes
6   3 distance[source] = 0
7   4 for step from 1 to N-1
8     5       for all edges from (u,v) in G.E
9       6           if distance[u] + cost[u][v] < distance[v]
10      7               distance[v] = distance[u] + cost[u][v]
11      8           end if
12      9       end for
13    10 end for
14    11 for all edges from (u,v) in G.E
15      12       if distance[u] + cost[u][v] < distance[v]
16      13           return "Negative cycle detected"
17      14       end if
18    15 end for
16 return distance

```



পাথ প্রিন্ট করা:

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

কমপ্লেক্সিটি:

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

চিন্তা করার জন্য কিছু প্রবলেম:

- তোমাকে একটা গ্রাফ দিয়ে বলা হল শট্টেস্ট পাথে সর্বোচ্চ xx টা এজ থাকতে পারে। এবার কিভাবে শট্টেস্ট 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)

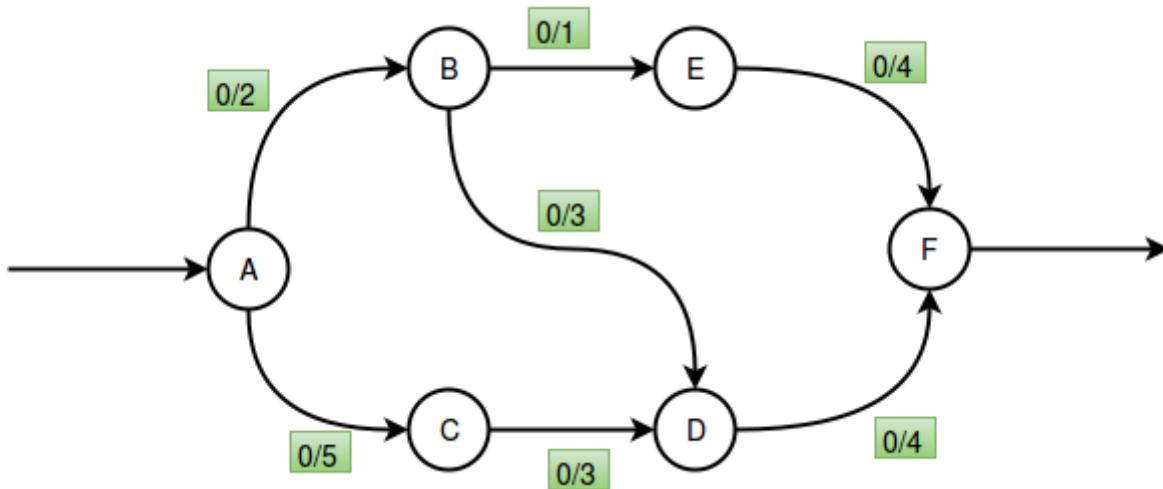
 shafaetsplanet.com/planetcoding/

শাফায়েত

জানুয়ারি ২৩, ২০১৫

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

প্রথমেই আমরা দেখি একটি খুবই সাধারণ ম্যাক্স ফ্লো প্রবলেম:



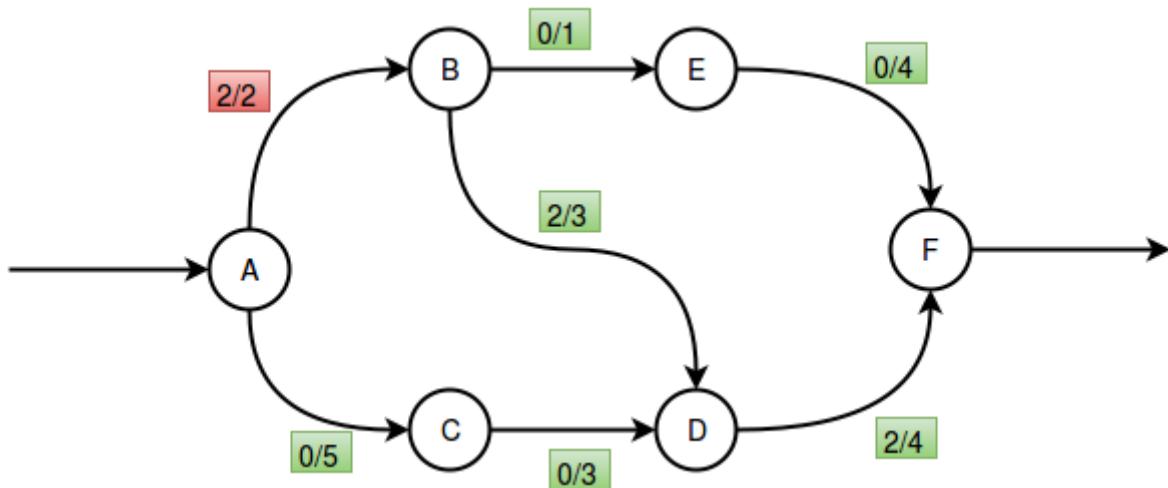
চিত্র: ১

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->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->C->D->F এই পথটা ব্যবহার করে পানি সোর্স থেকে সিংক এ গেলে সর্বোচ্চ কত লিটার পানি প্রবাহিত হতে পারবে? ছবিতে দেখা যাচ্ছে A->C এজের ক্যাপাসিটি ৫, C->D এজের ক্যাপাসিটি ৩, D->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 ১MBps। এখন নেটওয়ার্কের বাকি অংশের গতি যতই বেশি হোক না কেন তুমি ১MBps 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 পাইপে 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->B->D->F পথটা বেছে নিলাম। এই পথে সর্বনিম্ন ক্যাপাসিটি হলো ২। তাহলে এই পথে ২লিটার পানির ফ্লো পাঠিয়ে দেবার পর গ্রাফটা দেখতে হবে এরকম:

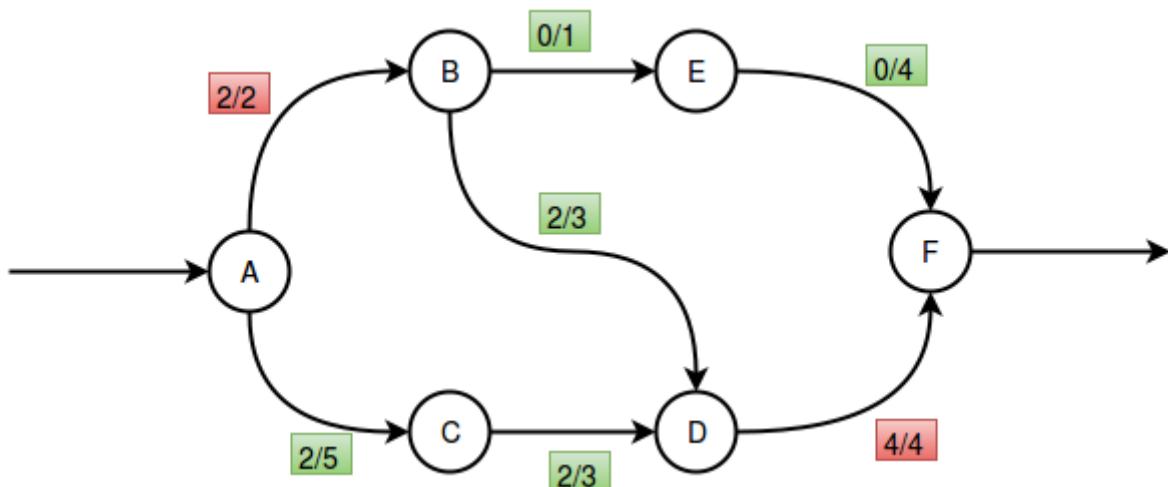


চিত্র-২: A-&gt;B-&gt;D-&gt;F পথে ফ্লো পাঠানোর পর

আমরা A->B->D->F পথের প্রতিটি এজের ফ্লো ২বাড়িয়ে 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  $3-2=1$  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**  $4-2=2$ । এই পথটা দিয়ে ২ ফ্লো পাঠানো সম্ভব।



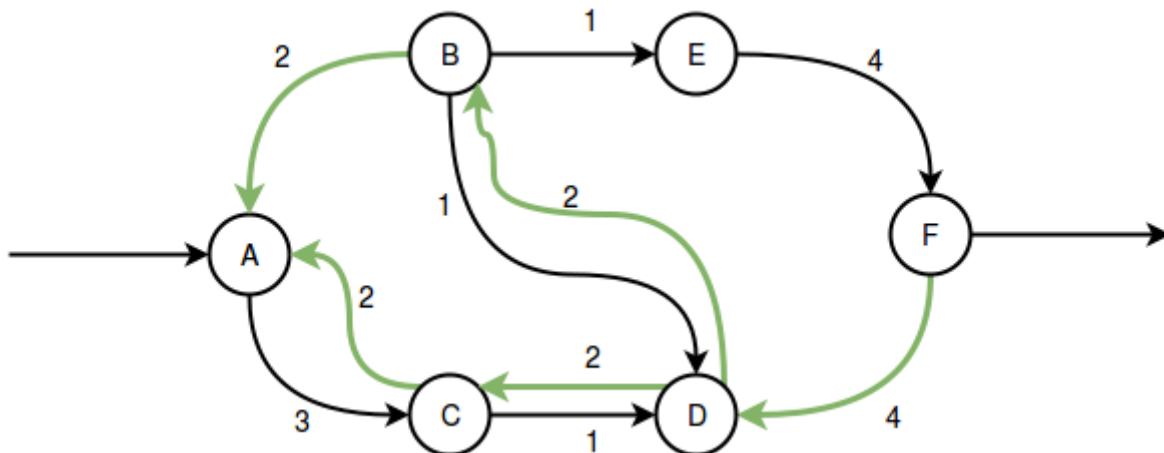
চিত্র-৩: A-&gt;C-&gt;D-&gt;F পথে ফ্লো পাঠানোর পর

আমরা এখন মোট  $2+2=4$  ফ্লো পেলাম, অর্থাৎ পানি প্রবাহের হার এখন সেকেলে ৪ লিটার। এই গ্রাফে কি আরো বেশি পানির ফ্লো পাঠানো সম্ভব? A->B এবং D->F এজ এরই মধ্যে সর্বোচ্চ ক্যাপাসিটিতে পৌছে 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 বেশ ফ্লো পাঠিয়ে দিব!

আমাদের এখন প্রতিটা এজের residual ক্যাপাসিটি ব্যবহার করে **residual গ্রাফ** আক়তে হবে কিভাবে আরো ফ্লো পাঠাবো সেটা বুঝতে হলে। residual গ্রাফ আকার নিয়ম হলো:

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

তারমানে কোনো এজ দিয়ে যতটুকু ফ্লো পাঠায়েছি 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:



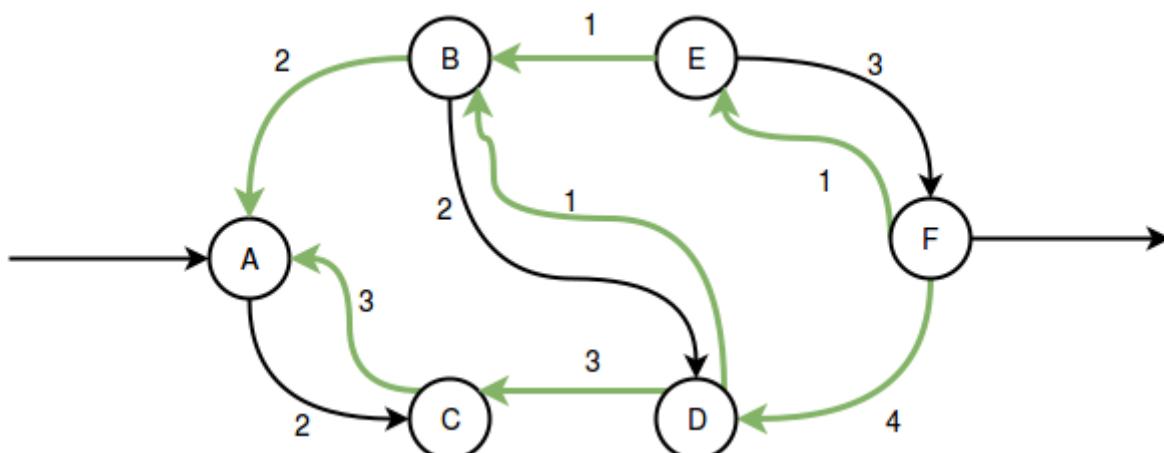
চিত্র-৪: ৩নং চিত্রের গ্রাফের residual গ্রাফ

সবুজ রঙ দিয়ে উল্টো এজ এবং তাদের residual ক্যাপাসিটি দেখানো হয়েছে। ০ ক্যাপাসিটির এজগুলোকে গ্রাফে একে দেখাইনি। A->C এজ দিয়ে আগে ২ফ্লো 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 ঘাটাই 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->B এজ দিয়ে ২ফ্লো পাঠাই তার মানে B->D এজ দিয়ে আসা খলিটার 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->C->D->B->E->F পথে আরো ১ ফ্লো পাঠানো সম্ভব। এই পথে D->B হলো উল্টো এজ। তারমানে D->B এজ এর ১ ফ্লো বাতিল করে দিয়ে আমরা গ্রাফে মোট ফ্লো ১বাড়িয়ে 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  $1-1=0$  হয়ে গিয়েছে এবং D->C এজের residual ক্যাপাসিটি বেড়ে  $2+1=3$  হয়ে গিয়েছে। এভাবে সবগুলো এজ আঁকা হয়েছে।

এই গ্রাফে সোর্স থেকে সিংক এ পানি পাঠানোর আর কোনো পথ নেই। আমরা ৩টি পথে মোট  $2+2+1=5$  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 পথ(Augmented path)**। তাহলে আমাদের অ্যালগোরিদম খুব সহজ, যতক্ষণ সন্তুষ্ট আমরা residual graph এ একটা অগমেন্টেড পাথ খুজে বের করবো এবং সেই পথে ফ্লো পাঠিয়ে দিবো! এটার নাম Ford-Fulkerson অ্যালগোরিদম। এটার সুড়েকোড 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 Cf(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 Cf[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       Cf(u,v) = Cf(u,v) - min_res_cap
11       Cf(v,u) = Cf(v,u) + 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 খুজে 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 ঘোগ 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  $C_f[A][B] = C_f[B][A]=10$  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 ২ন্দৰ 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 ফোর্ড ব্যবহার করোনা কেন, প্রতিটা নোডে যাবার সময় নোডটার প্যারেন্ট 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 খুজে 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 ঘার 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 পৌছাতে 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 হটি পথ আছে নাকি? এ ধরণের পথকে **এজ ডিসজঘেন্ট** পাঠ বলে।

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

প্রোগ্রামিং কনটেস্টে 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=1153)

[http://lightoj.com/volume\\_showproblem.php?problem=1155](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](#))

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# গ্রাফ থিওরিতে হাতেখড়ি-১২ – ম্যাক্সিমাম ফ্লো (২)

 [shafaetsplanet.com/planetcoding/](http://shafaetsplanet.com/planetcoding/)

শাফায়েত

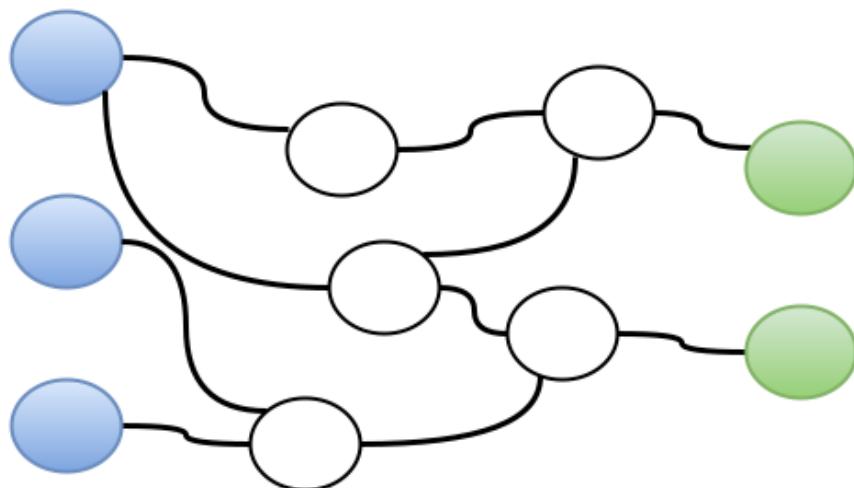
মার্চ ৩০, ২০১৫

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

## একাধিক সোর্স/সিংক:

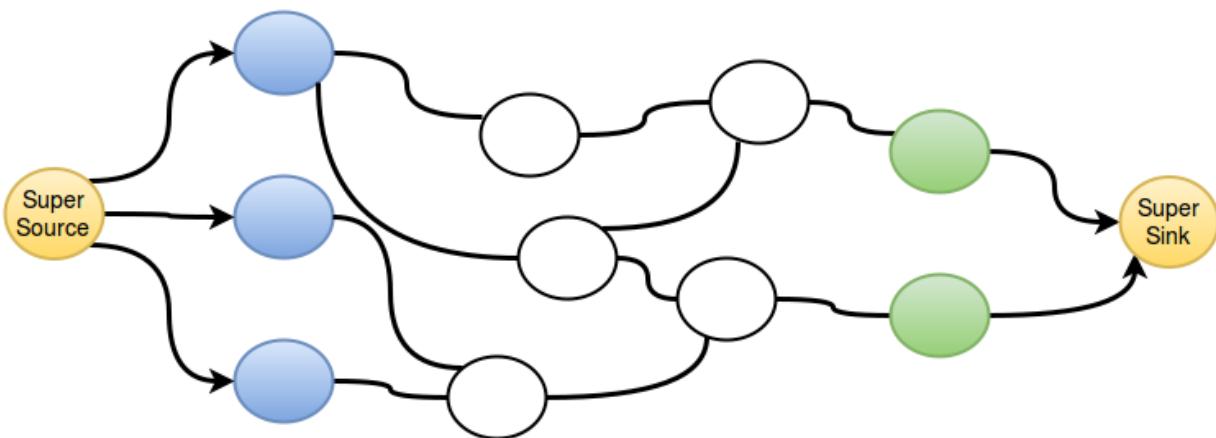
আগের পর্বে একটা প্রশ্ন করেছিলাম এরকম “আমাদের প্রবলেমে সোর্স এবং সিংক ছিলো একটা। কিন্তু গ্রাফে একাধিক নোড দিয়ে পানি প্রবেশ করলে এবং একাধিক নোড দিয়ে পানি বের হয়ে গেলে কিভাবে অ্যালগোরিদমটা পরিবর্তন করবে?” অর্থাৎ একাধিক সোর্স বা সিংক থাকলে কি করতে হবে সেটা জানতে চাওয়া হয়েছে।

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 ধরণের 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।

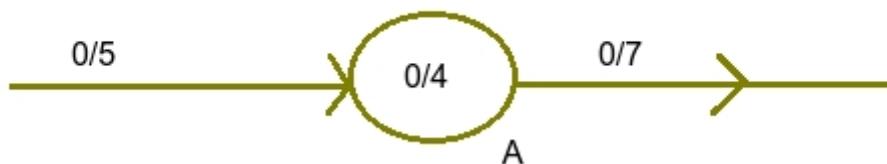


চিত্র-২: সুপার সোর্স এবং সুপার সিংক

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।

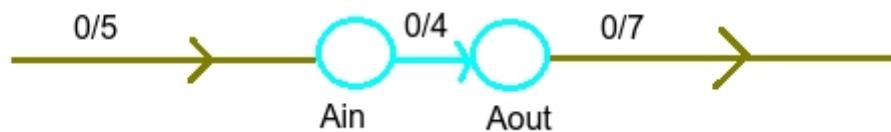
## নোড ক্যাপাসিটি:

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:



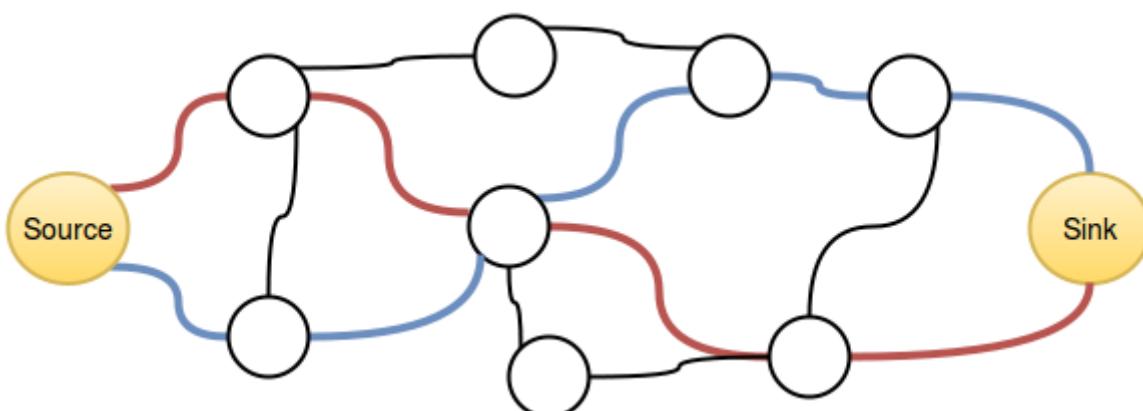
চিত্র-৪: A নোডটিকে দুইভাগ করা হয়েছে

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 নিচিতে 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 সোর্স নোডটা থেকে যাত্রা শুরু করে ডানের সিংক নোডে যেতে চায়। লাল এবং নীল রঙ ব্যবহার করে দুটি এজ-ডিসজয়েন্ট পাথ দেখানো হয়েছে।

সাধারণ ম্যাক্স-ফ্লো ব্যবহার করেই এজ ডিসজয়েন্ট পাথ বের করা যায়। শুরুর 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!

আজকের পর্ব এখানেই শেষ। মিন-কাট এবং ম্যাচিং নিয়ে আলোচনার জন্য আরেকটা পর্ব অপেক্ষা করতে হবে। কনটেন্টে 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 করবে এই জিনিসগুলো আয়ত্তে আনতে হবে।

কিছু প্রবলেম:

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

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by [AccessPress Themes](#)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

11/29/2015

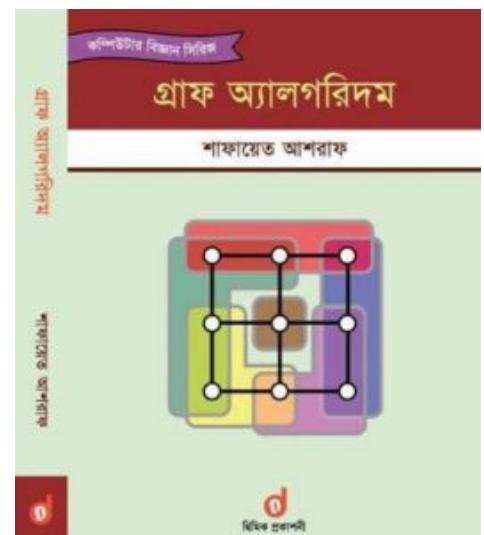
আর্টিকুলেশন পয়েন্ট হলো আনডিরেক্টেড গ্রাফের এমন একটা নোড যেটা গ্রাফ থেকে মুছে ফেললে বাকি গ্রাফটুকু একাধিক কম্পোনেন্ট এ ভাগ হয়ে যাব।

উপরের ছবিতে ১, ৩ অথবা ৪ নম্বর নোড এবং সেই নোডের অ্যাডজেসেন্ট এজগুলোকে মুছে দিলে গ্রাফটা একাধিক ভাগ হয়ে যাবে, তাই ১, ৩ ও ৪ হলো এই গ্রাফের আর্টিকুলেশন পয়েন্ট। আর্টিকুলেশন পয়েন্টকে অনেকে কাট-নোড(cut node) , আর্টিকুলেশন নোড বা ক্রিটিকাল পয়েন্ট (critical point) ও বলে।

আর্টিকুলেশন পয়েন্ট বের করার একটা খুব সহজ উপায় হলো, ১টা করে নোড গ্রাফ থেকে মুছে দিয়ে দেখা যে গ্রাফটি একাধিক কম্পোনেন্ট এ বিভক্ত হয়ে গিয়েছে নাকি।

```

1 1 procedure articulationPointNaive(G):
2 2     articulation_points=[]
3 3     for all nodes u in G
4 4         G.removeNode(u)
5 5         if get_number_of_component(G)>1
6 6             articulation_points.add(u)
7 7         end if
8 8         G.addBackNode(u)
9 9     end for
10 10    return articulation_points
  
```



কম্পোনেন্ট সংখ্যা ডিএফএস বা বিএফএস দিয়ে খুব সহজে বের করা যাব। এই পদ্ধতিতে  $VV$  বার ডিএফএস চালাতে হবে যেখানে  $VV$  হলো নোড সংখ্যা, মোট কমপ্লেক্সিটি  $O(V \times (V+E))O(V \times (V+E))$  বা  $O(V3)O(V3)$  কারণ সর্বোচ্চ এজ সংখ্যা  $V2V21$ ।

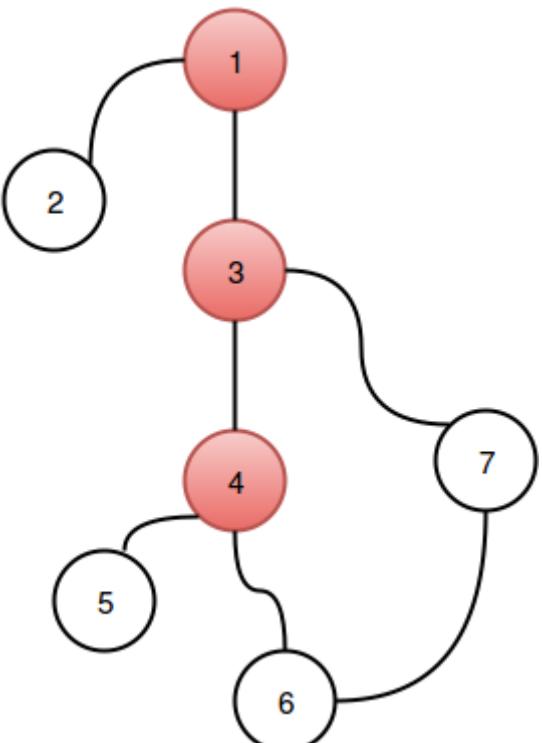
এখন আমরা একবার মাত্র ডিএফএস চালিয়ে আর্টিকুলেশন পয়েন্ট বের করবো। এই অ্যালগোরিদম শেখার জন্য ডিএফএস এর ডিসকভারি/ফিনিশিং টাইম এবং ট্রি এজ ও ব্যাক এজ নিয়ে ধারণা থাকতে হবে।

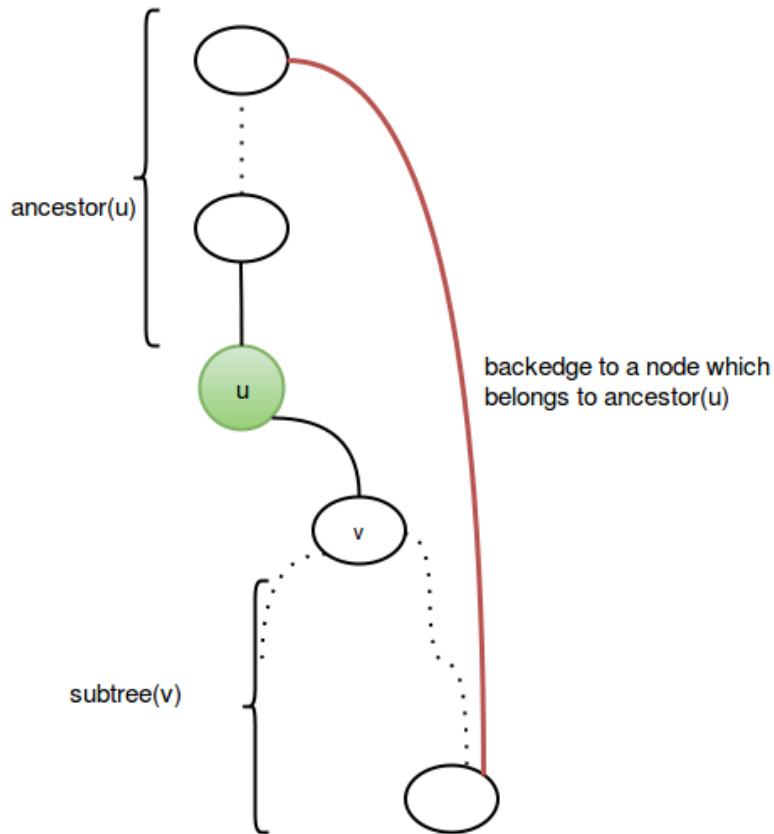
একটা গ্রাফে ডিএফএস চালালে যেসব ট্রি এজ পাওয়া যায় সেগুলো নিয়ে তৈরি হয়ে ডিএফএস ট্রি।

দুটি ক্ষেত্রে থাকতে পারে। যদি একটা নোড ট্রি এর রুট হয় তাহলে একভাবে কাজ করবো, রুট না হলে আরেকভাবে কাজ করবো।

একটা নোড  $uu$  যদি ট্রি এর রুট হয় এবং ডিএফএস ট্রি তে নোডটার একাধিক চাইল্ড নোড থাকে তাহলে নোডটা আর্টিকুলেশন পয়েন্ট।

রুট ছাড়া বাকি নোডের জন্য কাজটা একটু জটিল।



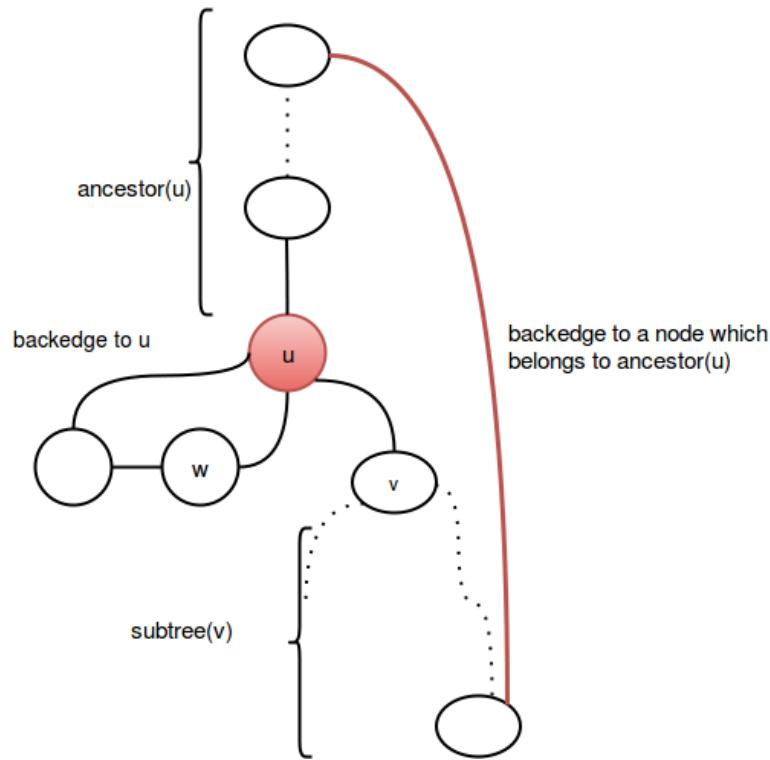


ডিএফএস ট্রি এর একটা এজ  $u-v-u-v$  এর কথা চিন্তা করো। রুট থেকে  $uu$  তে আসার পথে যেসব নোড ভিজিট করেছে তাদের আমরা বলবো  $\text{ancestor}(u)\text{ancestor}(u)$ । এখন  $vv$  যে সাবট্রি এর রুট সেই সাবট্রির সবগুলো নোডের সেটকে আমরা বলবো  $\text{subtree}(v)\text{subtree}(v)$ ।

এখন  $uu$  একটা আর্টিকুলেশন পয়েন্ট হবে যদি মূল গ্রাফে  $uu$  কে মুছে দিলে  $\text{subtree}(v)\text{subtree}(v)$  এর নোডগুলো একটা আলাদা কম্পোনেন্ট এ পরিণত হয়।  $\text{subtree}(v)\text{subtree}(v)$  আলাদা কম্পোনেন্ট এ পরিণত হবে যদি না মূল গ্রাফে সাবট্রি  $\text{subtree}(v)\text{subtree}(v)$  এর কোনো নোড থেকে  $\text{ancestor}(u)$  তে একটা ব্যাকএজ থাকে। যদি ব্যাকএজ থাকে তাহলে নোড  $uu$  এবং অ্যাডজেসেন্ট এজগুলো মুছে গেলেও  $\text{ancestor}(u)\text{ancestor}(u)$  থেকে ব্যাকএজ দিয়ে  $\text{subtree}(v)\text{subtree}(v)$  তে পৌছানো যাচ্ছে, নতুন কম্পোনেন্ট তৈরি হচ্ছে না।

$u$  এর ঘেকোনো একটা চাইল্ড নোড  $vv$  এর জন্য যদি  $\text{subtree}(v)\text{subtree}(v)$  থেকে  $\text{ancestor}(u)\text{ancestor}(u)$  তে পৌছানো না যায়, তাহলে  $uu$  আর্টিকুলেশন পয়েন্ট,  $uu$  কে মুছে দিলে সেইসব  $\text{subtree}(v)\text{subtree}(v)$  নতুন কম্পোনেন্ট এ পরিণত হবে যাদের সাথে  $\text{ancestor}(u)\text{ancestor}(u)$  এর কোনো ব্যাকএজ সংযোগ নেই।

নিচের ছবিতে  $\text{subtree}(v)\text{subtree}(v)$  যদিও ব্যাকএজ দিয়ে  $\text{ancestor}(u)\text{ancestor}(u)$  এর সাথে সংযুক্ত,  $\text{subtree}(w)\text{subtree}(w)$  থেকে  $\text{ancestor}(u)\text{ancestor}(u)$  তে ব্যাকএজ নেই। তাই  $uu$  একটা আর্টিকুলেশন পয়েন্ট।



এবার প্রথম গ্রাফটায় ফিরে আসি। গ্রাফের নোডগুলো ১,২,৩,৪,৬,৭,৫ এই অর্ডারে ভিজিট করলে আমরা প্রতিটা নোডের যা ডিসকভারি টাইম পাবো সেটা পাশে ছোটো করে লেখা হয়েছে:

ডিসকভারি টাইম কিভাবে বের করতে হয় না বুঝলে [ডিএফএস নিয়ে টিউটোরিয়ালটা](#) দেখো।  $d[]$  দিয়ে আমরা ডিসকভারি টাইম বুঝাবো।

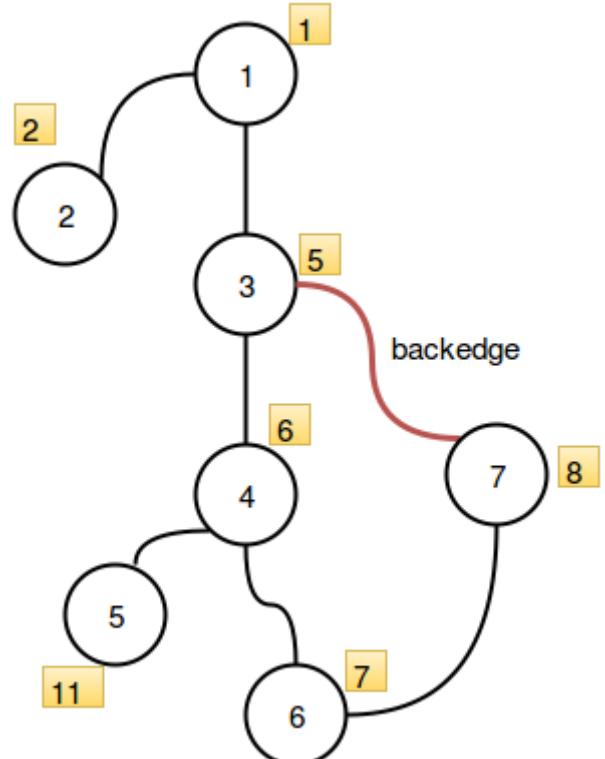
গ্রাফের ব্যাকএজ টা লাল এজ দিয়ে দেখানো হয়েছে। বাকি কালো এজগুলো ডিএফএস ট্রি এর অংশ। 11 হলো রুট নোড।

ডিএফএস ট্রি তে রুট নোড 11 এর চাইল্ড সংখ্যা এখানে ২টা (২ এবং ৩)। তাই 11 একটা আটিকুলেশন পয়েন্ট।

লক্ষ্য করো নোড  $\text{ancestor}(u)\text{ancestor}(u)$  এর যেকোনো নোডের ডিসকভারি টাইম  $d[u]d[u]$  এর থেকে ছোটো। আবার  $uu$  এর অ্যাডজেসেন্ট যেকোনো এজ  $u-vu-v$  এর জন্য  $\text{subtree}(v)\text{subtree}(v)$  এর সব নোডের ডিসকভারি টাইম  $d[u]d[u]$  এর থেকে বড়। এখন  $\text{subtree}(v)\text{subtree}(v)$  এর কোনো নোড থেকে যদি এমন একটা ব্যাকএজ  $v-wv-w$  থাকে যেন  $d[w]< d[u]d[w]< d[u]$  হয় তাহলে বুঝতে হবে তুমি  $u-vu-v$  এজ পার হয়ে  $\text{subtree}(v)\text{subtree}(v)$  দিয়ে  $\text{ancestor}(u)\text{ancestor}(u)$  তে পৌছে গেছো এবং  $w\in\text{ancestor}(u)w\in\text{ancestor}(u)$ । তারমানে  $u$  মুছে দিলেও  $\text{subtree}(v)\text{subtree}(v)$  থেকে  $ww$  তে পৌছানো যাবে।

যেমন 44 নম্বর নোডের কথা চিন্তা করো। 44 এর ডিসকভারি টাইম  $d[4]=6d[4]=6$  এবং  $\text{ancestor}(4)=\{1,2,3\}\text{ancestor}(4)=\{1,2,3\}$ । এখন 4-6 এজটার কথা ভাবি।  $\text{subtree}(6)\text{subtree}(6)$  এ একটা ব্যাকএজ 7-37-3 আছে, এবং  $d[3]=5d[3]=5$  যা  $d[4]d[4]$  এর থেকে ছোটো। তারমানে  $3\in\text{ancestor}(4)3\in\text{ancestor}(4)$ । তাহলে তুমি 44 নোডটা মুছে দিলেও  $\text{subtree}(6)\text{subtree}(6)$  ব্যাকএজের মাধ্যমে  $\text{ancestor}(4)\text{ancestor}(4)$  এর সাথে সংযুক্ত থাকবে।

এবার আমরা আরেকটা ভ্যারিয়েবল ডিফাইন করবো  $\text{low}[u]\text{low}[u]$ । মনে করো  $\text{subtree}(u)\text{subtree}(u)$  এবং  $\text{subtree}(u)\text{subtree}(u)$  এর সাথে ব্যাকএজ দিয়ে সংযুক্ত সবগুলো নোডের একটা সেট বানানো হলো, সেটা টা হলো  $\{x_1,x_2\dots x_m\}\{x_1,x_2\dots x_m\}$ । তাহলে  $\text{low}[u]\text{low}[u]$  হবে  $\min(d[x_1],d[x_2],\dots,d[x_m])\min(d[x_1],d[x_2],\dots,d[x_m])$ ।



যেমন 4 নম্বর নোডের জন্য  $\text{subtree}(u)=\{5,6,7\}$   $\text{subtree}(u)=\{5,6,7\}$  এবং  $\text{subtree}(u)\text{subtree}(u)$  এর সাথে ব্যাকএজ দিয়ে যুক্ত আছে নোড 3। তাহলে  $\text{low}[u]=\min(d[5], d[6], d[7], d[3])=5$   $\text{low}[u]=\min(d[5], d[6], d[7], d[3])=5$ ।

এখন চিন্তা করো কোনো একটা এজ  $u-v-u$  এর জন্য  $d[u]>\text{low}[v]$  হবার অর্থ কি?  $d[u]$  এর থেকে ডিসকভারি টাইম ছোটো একমাত্র  $\text{ancestor}(u)\text{ancestor}(u)$  সেটের নোডগুলোর।  $\text{subtree}(v)$  এর কোনো নোড ব্যাকএজ দিয়ে  $\text{ancestor}(u)$  এর সাথে যুক্ত, সেজন্য  $\text{low}[v]$  এর মান  $d[u]$  এর থেকে কমে গিয়েছে। যদি  $d[u]\leq\text{low}[v]$   $d[u]\leq\text{low}[v]$  হয়, তাহলেই শুধুমাত্র  $u$  একটা আর্টিকুলেশন পয়েন্ট হবে।

আগের গ্রাফেই ডিসকভারি টাইমের পাশাপাশি  $\text{low}[u]$   $\text{low}[u]$  এর মানগুলোও দেখি:

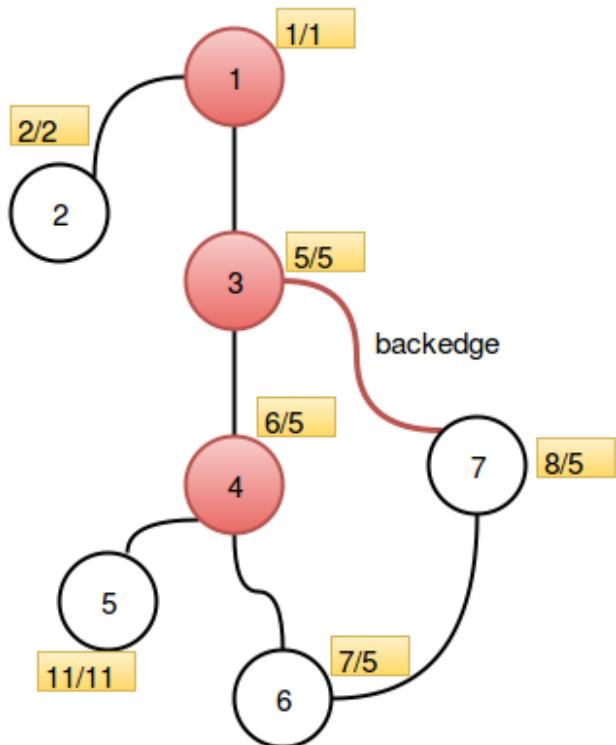
তাহলে আমরা আর্টিকুলেশন পয়েন্ট বের করার একটা

অ্যালগোরিদম পেয়ে গিয়েছি। প্রতিটা নোডের জন্য  $d[u]$ ,  $\text{low}[u]$   
বের করতে পারলেই কাজ শেষ।  $\text{low}[u]$  বের করা কঠিন কিছু না,  
সুড়োকোড দেখলেই পরিষ্কার হবে:

```

1     articulation_point[] ← false
2     visited[] ← false
3     low[] = d[u] ← 0
4     time ← 0
5     1 Procedure FindArticulationPoint(G, u):
6     2   time ← time + 1
7     3   low[u] = d[u] ← time
8     4   visited[u] ← true
9     5   no_of_children ← 0
10    6   for each edge u to v in G.adjacentEdges(u) do
11    7     if(v == parent[u]) continue
12    8     if visited[v] //This is a backedge
13    9       low[u] = min(low[u], d[v])
14   10     end if
15   11     if not visited[v] //This is a tree edge
16   12       parent[u] = v
17   13       FindArticulationPoint(G, v)
18   14       low[u] = min(low[u], low[v])
19   15       if d[u] ≤ low[v] and u is not root:
20   16         articulation_point[u] = true
21   17       end if
22   18       no_of_children = no_of_children + 1
23   19     end if
24   20     if(no_of_children > 1 u is root):
25   21       articulation_point[u] = true
26   22   end if
27   23 end for

```



ব্রিজ জিনিসটা আর্টিকুলেশন পয়েন্টের মতই। গ্রাফ থেকে যে এজ তুলে দিলে গ্রাফটা একাধিক কম্পোনেন্টে ভাগ হয়ে যায় তাকেই বলা হয় ব্রিজ।

উপরের গ্রাফে 4-54-5, 1-21-2, আর 1-31-3 এই তিনি এজ হলো ব্রিজ।

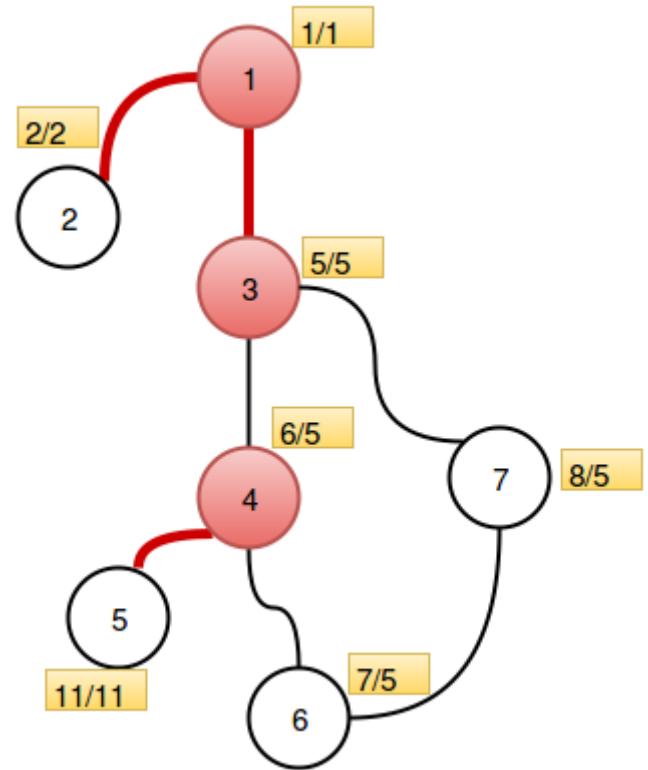
ব্রিজ আর আর্টিকুলেশন পয়েন্টের সুড়োকোডের পার্থক্য খালি এক জায়গায় ১৫ নম্বর লাইনে  $d[u]\leq\text{low}[v]$  এর জায়গায়  $d[u]\leq\text{low}[v]$  লিখতে হবে। এটা কেন কাজ করে তুমি সহজেই বুঝতে পারবে যদি তুমি সুড়োকোডটা বুঝে থাকো, তাই আর ব্যাখ্যা করলাম না।

দুটি নোডের মধ্যে একাধিক এজ থাকলে অবশ্য এটা কাজ করবে না। তখন কি করতে হবে সেটা চিন্তা করা তোমার কাজ!

সলভ করার জন্য কিছু প্রবলেম পাবে [এখানে](#)।

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress](#)  
[Staple by AccessPress Themes](#)



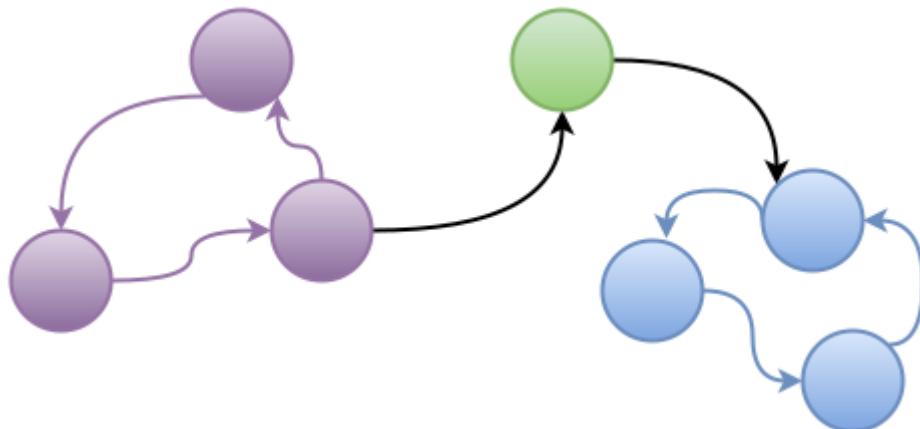
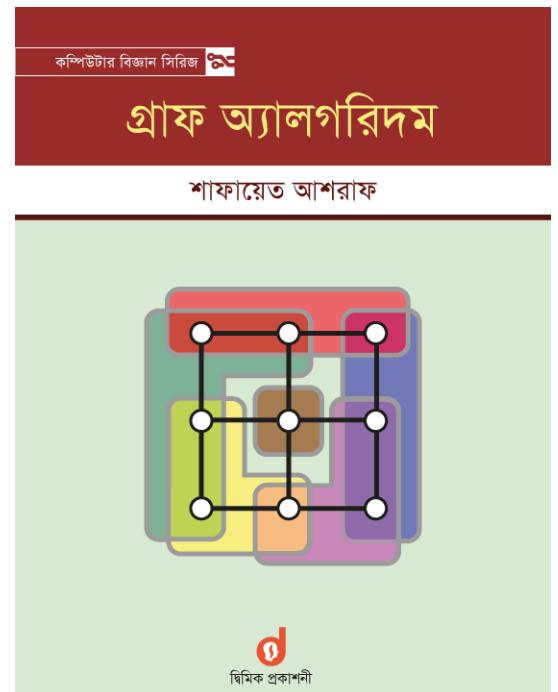
# গ্রাফ থিওরিতে হাতেখড়ি ১৪ – স্ট্রংলি কানেক্টেড কম্পোনেন্ট

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

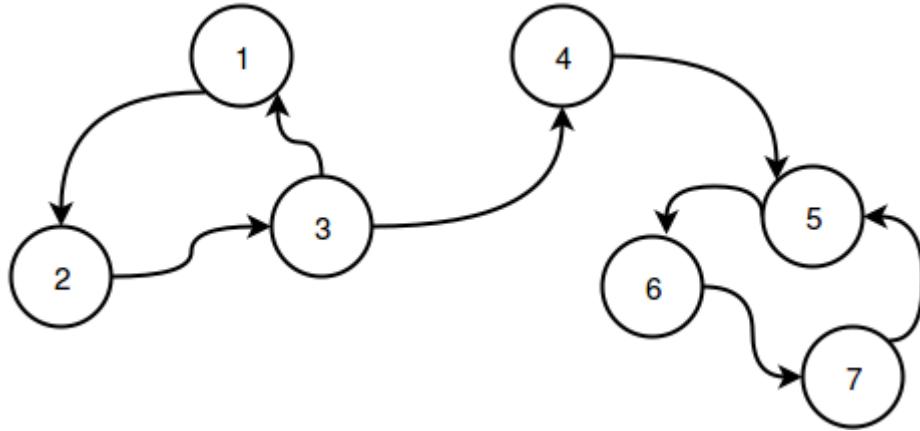
11/29/2015

একটা ডিরেক্টেড গ্রাফের স্ট্রংলি কানেক্টেড কম্পোনেন্ট বা SCC হলো এমন একটা কম্পোনেন্ট যার প্রতিটা নোড থেকে অন্য নোডে যাবার পথ আছে। নিচের ছবিতে একটা গ্রাফের প্রতিটা স্ট্রংলি কানেক্টেড কম্পোনেন্ট আলাদা রঙ দিয়ে দেখানো হয়েছে।



ডেপথ ফার্স্ট সার্চ এর ফিনিশিং টাইমের ধারণা ব্যবহার করে আমরা  $O(V+E)O(V+E)$  তে একটা গ্রাফের স্ট্রংলি কানেক্টেড কম্পোনেন্ট গুলোকে আলাদা করে ফেলতে পারি। এই লেখাটা পড়ার আগে অবশ্যই টপলোজিকাল সার্টিং আর ডেপথ ফার্স্ট সার্চ এর ডিসকভারি এবং ফিনিশিং টাইম সম্পর্কে ধারণা থাকতে হবে।

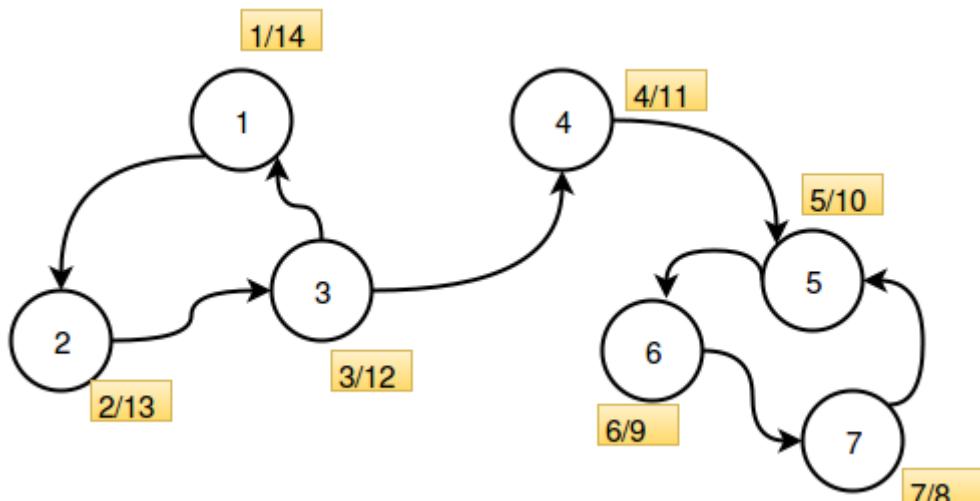
নিচের গ্রাফটা দেখ:



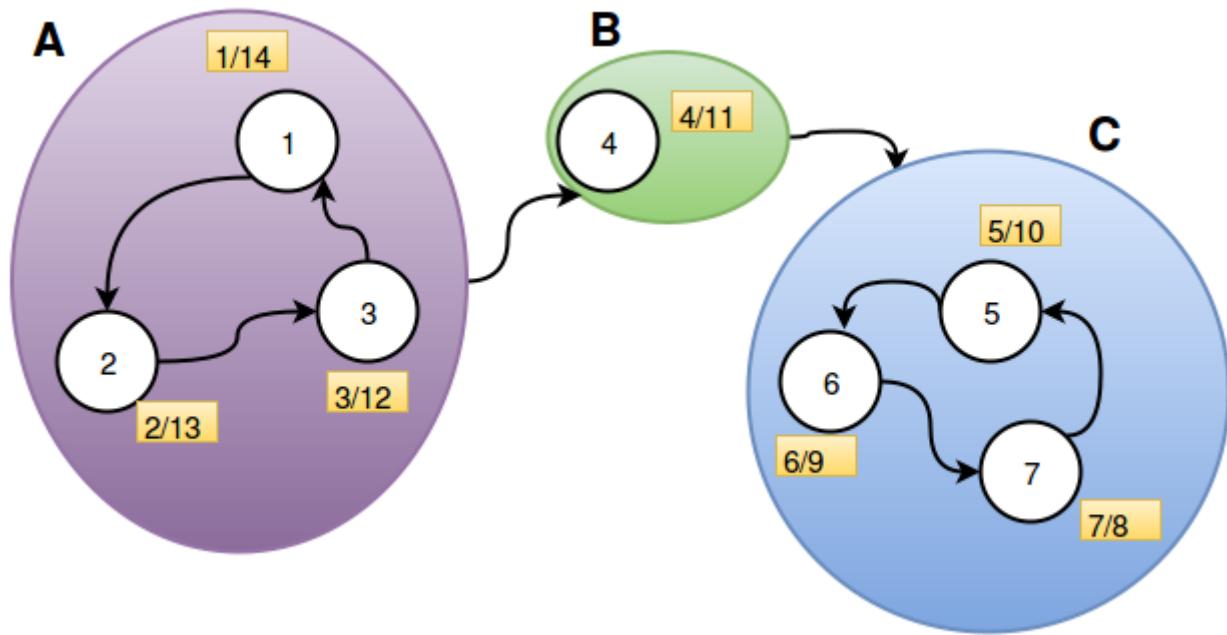
প্রথমেই একটা ভুল পদ্ধতিতে অনেকে SCC বের করার চেষ্টা করে। সেটা হলো যেকোনো নোড থেকে ডিএফএস চালিয়ে যেসব নোডে যাওয়া যায় তাদেরকে একটা কম্পোনেন্ট হিসাবে ধরা। কিন্তু খুব সহজেই বোঝা যায় এটা কাজ করবে না, উপরের গ্রাফে ১ থেকে ডিএফএস চালালে সবগুলো নোড ভিজিট করা যাবে, কিন্তু ১ থেকে ৪ এ যাওয়া গেলেও ৪ থেকে ১ এ যাবার কোনো পথ নেই, তাই এরা একই কানেক্টেড কম্পোনেন্ট এর অংশ না। এই পদ্ধতিতে সমস্যা হলো ডিএফএস কানেক্টেড কম্পোনেন্ট থেকে বের হয়ে অন্য কম্পোনেন্ট এ চলে যায়। এই সমস্যা সমাধান করতে আমরা একটু বুদ্ধিমানের মত ডিএফএস চালাবো।

দুটি নোড  $u, v$  একই SCC তে থাকবে শুধুমাত্র যদি  $u$  থেকে  $v$  তে যাবার পথ থাকে এবং  $v$  থেকে  $u$  তে যাবারও পথ থাকে।

প্রথমে আমরা ১ থেকে ডিএফএস চালিয়ে সবগুলো নোডের ডিসকভারি টাইম আর ফিনিশিং টাইম লিখে ফেলি। নোডগুলো ১, ২, ৩, ৪, ৫, ৬, ৭ অর্ডারে ভিজিট করলে আমরা নিচের ছবির মত স্টার্টিং/ফিনিশিং টাইম পাবো:



এখন বোঝার সুবিধার জন্য স্ট্রিংলি কানেক্টেড কম্পোনেন্টের সবগুলো নোডকে একটা বড় নোড মনে করি:



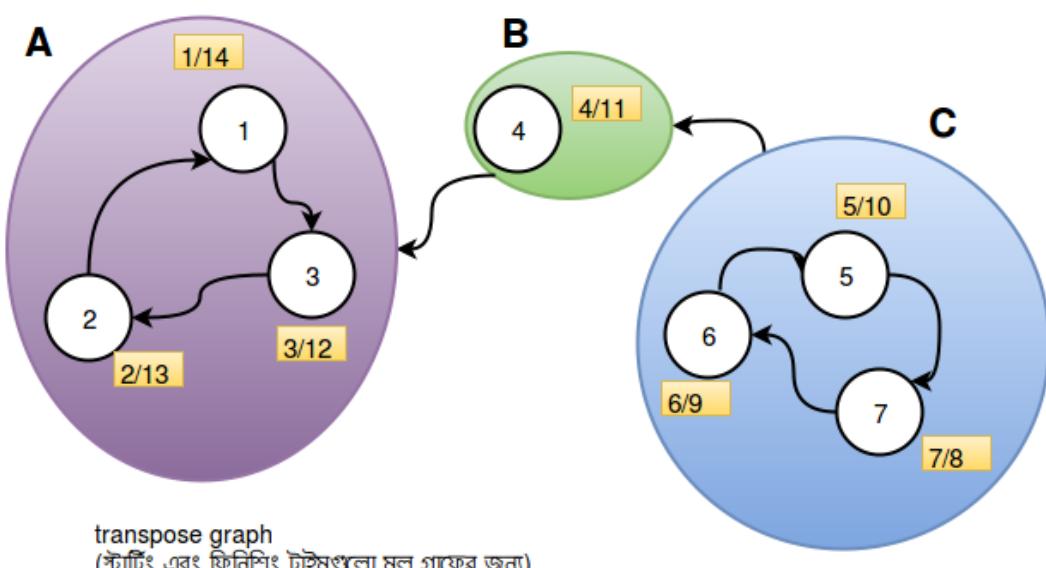
লক্ষ্য করো, গ্রাফটাকে এভাবে 'ডিকম্পোজ' করার পর গ্রাফটিতে আর কোনো সাইকেল থাকা সম্ভব না, অর্থাৎ গ্রাফটি একটি ড্যাগ বা ডিরেক্টেড অ্যাসাইন্সেশন গ্রাফে পরিণত হয়েছে। এখন বড় নোডগুলোকে সহজেই টপোলজিকাল অর্ডারে সাজানো সম্ভব, অর্ডারটা হবে A,B,C।

এখন লক্ষ্য করো ড্যাগে একটা নোড D1 থেকে অন্য নোড D2 এ যাওয়া যায় তাহলে D1 টপোলজিকাল অর্ডারে D2 এর আগে অবশ্যই থাকবে। আবার আমরা আগেই জানি যে টপোলজিকাল অর্ডারে যে আগে থাকে তার ফিনিশিং টাইম বেশি হয় কারণ অন্যান্য সব নোডের কাজ শেষ করে ওই নোডে ফিরে আসতে হয়।

তাহলে D1 যদি টপোলজিকাল অর্ডারে D2 এর আগে থাকে তাহলে যেসব ছোটো ছোটো নোড নিয়ে D2 গঠিত হয়েছে তাদের সবার ফিনিশিং টাইম অবশ্যই D1 এর সব নোডের থেকে কম হবে।

এখন u থেকে v তে যাওয়া গেলেই তারা একই SCC এর অন্তর্ভুক্ত না, v থেকে u তে যাবার পথও থাকতে হবে। অথবা আমরা বলতে পারি 'উল্টো-গ্রাফ' এও u থেকে v তে যাবার পথ থাকতে হবে!

যদি গ্রাফের এজগুলো উল্টে দেয়া হয়, তাহলেও SCC গুলো একই থাকে। একে বলা হয় ট্রান্সপোজ গ্রাফ(transpose) , ট্রান্সপোজ গ্রাফে সাইকেল গুলোর কোনো পরিবর্তন হয় না। মূল গ্রাফে যদি u-v একই SCC এর মধ্যে থাকে তাহলে তারা অবশ্যই একই সাইকেলের মধ্যে থাকবে। এটাই আমাদের অ্যালগোরিদমের মূল ভিত্তি।



উপরের ছবিতে আগের গ্রাফের এজগুলো উল্টে দেয়া হয়েছে। ডিসকভারি এবং ফিনিশিং টাইম আগেরটাই লেখা আছে।

এখন লক্ষ্য করো তুমি যদি শুরুতে টপলোজিকাল অর্ডারে আগে থাকা নোডগুলো থেকে ডিএফএস চালাও অর্থাৎ যার ফিনিশিং টাইম বড় সেখান থেকে শুরু করো তাহলে তুমি প্রথম প্রথম SCC টা পেয়ে যাবে।

উপরের গ্রাফে ১ এর ফিনিশিং টাইম সবথেকে বেশি (14)। ১ থেকে ডিএফএস চালালে তুমি যেতে পারবে {1,2,3} নোডগুলোতে যারা একই SCC'র অংশ। এবার {১,২,৩} নোডগুলো গ্রাফ থেকে মুছে ফেল। এরপর ৪ এর ফিনিশিং টাইম বড়। ৪ থেকে শুধুমাত্র {৪} এ যাওয়া যায়। এরপর ৫ থেকে ডিএফএস চালাবো, সেখান থেকে যাওয়া যায় {৫,৬,৭} নোডগুলোতে যারা একটি SCC এর অংশ।

যেসব নোডগুলো একই কম্পোনেন্ট এর অংশ তাদের কে আমরা আলাদা লিস্টে সেভ করে রাখবো নিজের সুড়েকোডটা দেখো:

```

1   1     procedure DFS(G, u):
2   5       color[u] ← GREY
3   6       for all edges from u to v in G.adjacentEdges(u) do
4   7           if color[v]=WHITE
5   8               DFS(G,v)
6   9           end if
7  10      end for
8  11      stk.add(source)
9  13      return
10
11 14      procedure DFS2(R,u, mark)
12 15          components[mark].add(u) //save the nodes of the new component
13 16          visited[u] ← true
14 17          for all edges from u to v in R.adjacentEdges(u) do
15 18              if visited[v] ← false
16 19                  DFS2(R,v, mark)
17 20              end if
18 21          end for
19 22          return
20
21 23      procedure findSCC(G):
22 24          stk ← an empty stack
23 25          visited[] ← null
24 26          color[] ← null
25 27          components[] ← null
26 28          mark=0
27 29          for each u in G
28 30              if color[u]=WHITE
29 31                  DFS(G,u)
30 32              end if
31 33          end for
32 34          R=reverseEdges(G)
33 35          while stk not empty
34 36              u=stk.removeTop()
35 37              if visited[u]=false
36 38                  mark=mark+1 //A new component found, it will be identified by 'mark'
37 39                  DFS2(R,u,mark)
38 40              end if
39 41          end for
40 42          return components

```

কোডটা একটু বড় মনে হলেও বোঝা খুব সহজ। প্রথমে একটা ডিএফএস চালিয়ে ফিনিশিং টাইম অনুযায়ী নোডগুলো সর্ট করছি। একটা স্ট্যাক ব্যবহার করে কাজটা করছি। যার ফিনিশিং টাইম কম সে কাজ আগে শেষ করে ১১ নম্বর লাইনে আসবে, তখন সেই নোডটা স্ট্যাকে ঢুকিয়ে রাখবো। সবশেষে স্ট্যাকের উপরে যে নোড থাকবে তার ফিনিশিং টাইম হবে সব

থেকে বেশি। এর পর ২য় ডিএফএস চালিয়ে কম্পোনেন্টগুলো আলাদা করে ফেলবো। mark নামের ভ্যারিয়েবল টা ব্যবহার করছি প্রতিটা কম্পোনেন্ট এর আলাদা নাম দেয়ার জন্য, ছবিতে যেভাবে A,B,C নাম দেয়া হয়েছে।

সলভ করার জন্য কিছু প্রবলেম পাবে এখানে।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

8/15/2012

বেশ কিছুদিন ডিপি নিয়ে লেখার পর আবার গ্রাফ থিওরিতে ফিরে এলাম। আজকে আমরা একটা সহজ কিন্তু ইন্টারেস্টিং প্রবলেম দেখবো। স্টেবল ম্যারিজ(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 যেনো 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 3জনের নাম Tom, Bob, Peter, আর মেয়ে 3জনের নাম Alice, Mary, Lucy হয় তাহলে ছেলেদের পছন্দের তালিকা হতে পারে এরকম:

তালিকাটা বেশি থেকে কম পছন্দের ক্রমে করা হয়েছে।  
যেমন টম এলিসকে বেশি পছন্দ করে, লুসিকে কম পছন্দ করে।

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:

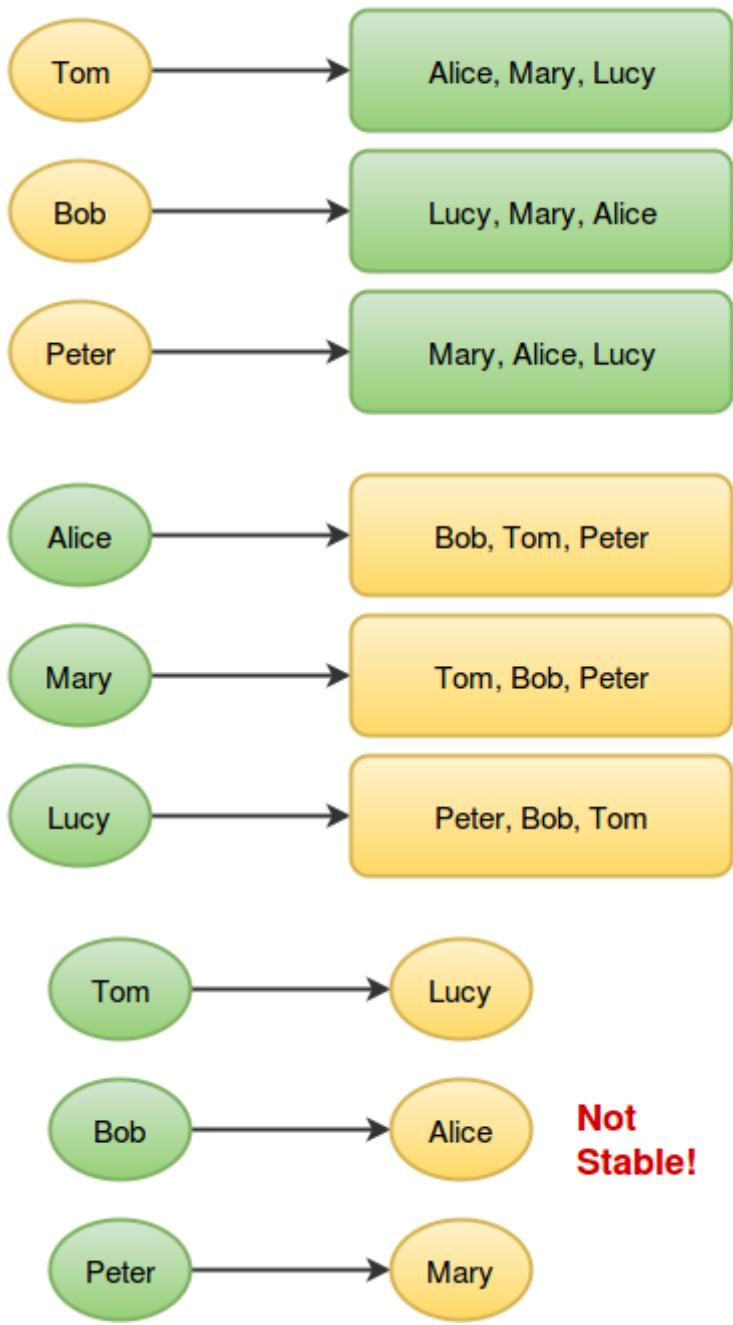
এই ম্যাচিং/বিয়েটা স্টেবল না, কারণ টম লুসির থেকে মেরিকে বেশি পছন্দ করে, আবার মেরি পিটারের থেকে টমকে বেশি পছন্দ করে। তাই টম আর মেরি বিয়ে ভেঙে একসাথে চলে আসতে পারে। যদি A,B ছেলে আর C,D মেয়ে হয় আর A-C এবং B-D কে বিয়ে দেয়া হয় তাহলে বিয়ে স্টেবল হবেনা যদি নিচের দুটি স্টেটমেন্টই সত্য হয়:

- 1. A যদি C এর থেকে D কে বেশি পছন্দ করে।
- 2. D যদি B এর থেকে A কে বেশি পছন্দ করে।

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

১৯৬২ সালে David Gale আর Lloyd Shapley প্রমাণ করেন, সমান সংখ্যক ছেলে আর মেয়ের জন্য সমসময় স্টেবল ম্যারিজ প্রবলেমের একটি সমাধান আছে। তারা খুব সহজ একটা অ্যালগোরিদম আবিষ্কার করেন সমস্যাটি সমাধানে জন্য। অ্যালগোরিদমটি এরকম:

- 1. 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 ভাঙ্গে A যদি D কে বেশি পছন্দ করে এবং D যদি A কে বেশি পছন্দ করে। কিন্তু উপরের অ্যালগোরিমে সেটা সন্তুষ্ট করে। 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 ভাঙ্গার সন্তুষ্ট করে।

এই অ্যালগোরিদমটা 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 কাওকে 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 অ্যারেতে রেখে সহজেই কোডটা লিখে ফেলতে পারবে। [wikipedia](#) তে দেয়া সুড়োকোডটা এরকম:

Python

```

1 function stableMatching {
2   Initialize all m ∈ M and w ∈ W to free
3   while ∃ 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](#)

# মিনিমাম ভারটেক্স কভার প্রবলেম

 shafaetsplanet.com/planetcoding/

শাফায়েত

10/4/2012

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 নৃন্যতম কয়জন পাহারাদার দরকার? ছবিতে লাল নোডগুলোতে পাহারাদার বসানো হয়েছে। এটা অপটিমাল না, নিচের ছবির মত বসালে পাহারাদার কম লাগত:

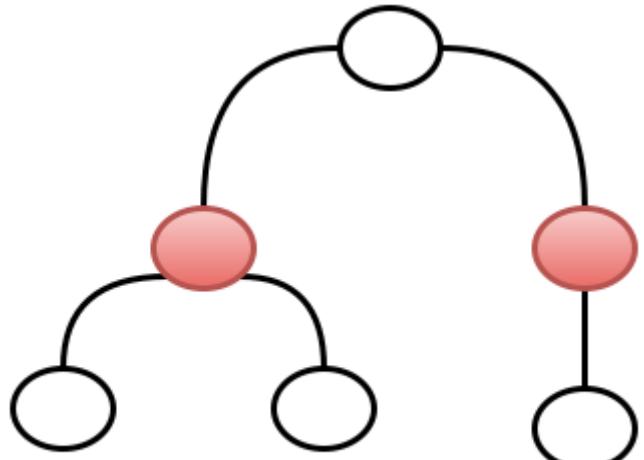
এটি একটি NP-hard প্রবলেম, অর্থাৎ এই প্রবলেমের কোনো পলিনমিয়াল টাইম সলিউশন নেই। তবে গ্রাফটি যদি Tree হয় অর্থাৎ  $n-1$  টা edge থাকে আর কোনো সাইকেল না থাকে তাহলে ডাইনামিক প্রোগ্রামিং বা ম্যাক্স ফ্লো/বাইপারটাইট ম্যাচিং এর সাহায্যে প্রবলেমটি সলভ করা সম্ভব।



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

ডিপি সলিউশনে ২টি কেস আমাদের লক্ষ্য করতে হবে:

1. 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।
2. কোনো নোডে পাহারাদার বসালে সংযুক্ত নোডগুলোতে পাহাদার 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।



Minimum Vertex Cover in a Tree

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

$$\begin{aligned}
 F(u, 1) &= \text{বর্তমানে } U \text{ নম্বর নোডে আছি এবং এই নোডে পাহারাদার আছে} / f(u, 1) \text{ রিটার্ন করবে বাকি} \\
 &\text{নোডগুলোতে মোট পাহারাদার সংখ্যা} \\
 F(u, 0) &= \text{বর্তমানে } U \text{ নম্বর নোডে আছি এবং এই নোডে পাহারাদার নাই} / f(u, 0) \text{ রিটার্ন করবে বাকি} \\
 &\text{নোডগুলোতে মোট পাহারাদার সংখ্যা}
 \end{aligned}$$

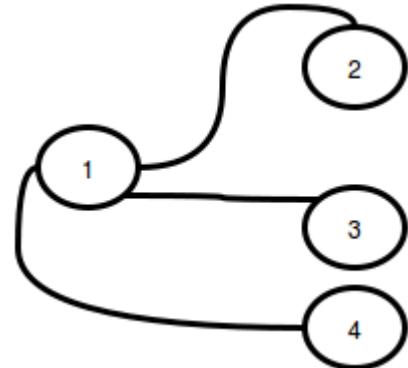
ধরি ১ নম্বর নোডের সাথে ২, ৩, ৪ নম্বর নোড যুক্ত।

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 বসাচ্ছি 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 1 বা 0 রিটার্ন করে দিতে হবে, পাহারাদার বসালে 1, না বসালে 0। কোনো ট্রি তে একটি মাত্র নোড থাকলে 1 রিটার্ন করতে হবে (কিছু প্রবলেমে ০ ও 1 রিটার্ন করতে হতে পারে)।

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

```

1 #define MAXN 100002
2 int dp[MAXN][5];
3 int par[MAXN];
4 vectoredges[MAXN];
5
6 int f(int u, int isGuard)
7 {
8     if (edges[u].size() == 0)
9         return 0;
10    if (dp[u][isGuard] != -1)
11        return dp[u][isGuard];
12    int sum = 0;
13    for (int i = 0; i < (int)edges[u].size(); i++) {
14        int v = edges[u][i];
15        if (v != par[u]) {
16            par[v] = u;
17            if (isGuard == 0)
18                sum += f(v, 1);
19            else
20                sum += min(f(v, 1), f(v, 0));
21        }
22    }
23    return dp[u][isGuard] = sum + isGuard;
24 }
25
26 int main()
27 {
28     memset(dp, -1, sizeof(dp));
29     int n;
30     scanf("%d", &n);
31     for (int i = 1; i < n; i++) {
32         int u, v;
33         scanf("%d%d", &u, &v);
34         edges[u].push_back(v);
35         edges[v].push_back(u);
36     }
37     int ans = 0;
38     ans = min(f(1, 1), f(1, 0));
39     printf("%d\n", ans);
40     return 0;
41 }
```

আমি ট্রি 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 ট্রি 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 কস্টিশন দিয়ে প্যারেন্ট নোডে যেতে দিচ্ছিনা।

সবশেষে sum+isGuard রিটার্ন করছি। অর্থাত কারেন্ট নোডে পাহারাদার থাকলে 1 ঘোগ করছি, নাহল 0।

মোটামুটি এই হলো ডিপি সলিউশন। ট্রি তে সাইকেল না থাকায় এটা অবশ্যই বাইপারটাইট গ্রাফ। ১৯৩১ সালে Dénes 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 ট্রি 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।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ট্রি ডায়ামিটার

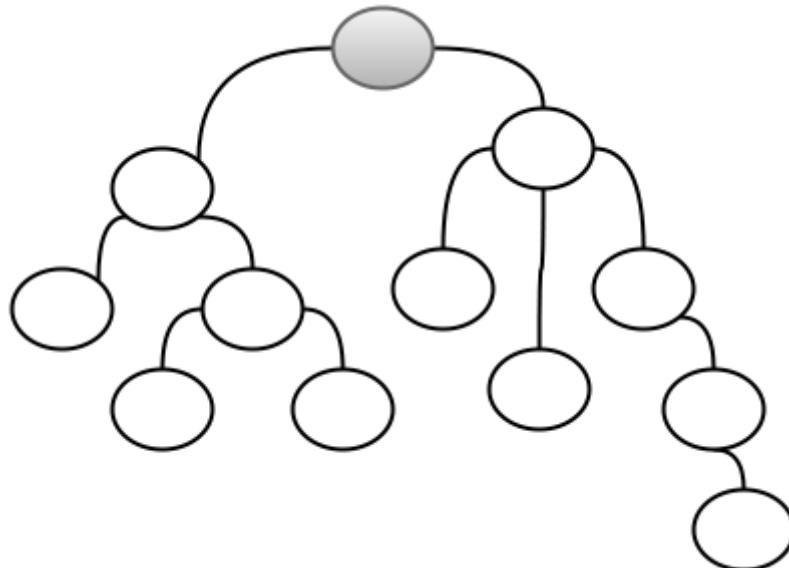
 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

1/8/2014

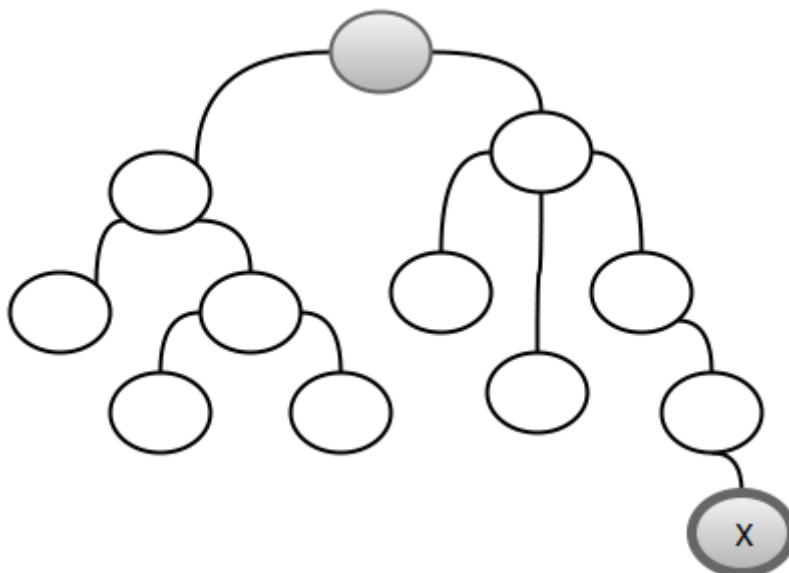
ট্রি হলো এমন একটা আনডিরেক্টেড গ্রাফ যেটার সব নোড থেকে সব নোডে যাওয়া যায় এবং কোনো সাইকেল নেই। এখন আমাদের ট্রি এর সবথেকে দূরের দুটা নোড খুজে বের করতে হবে, একেই বলা হয় ট্রি এর ডায়ামিটার।

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

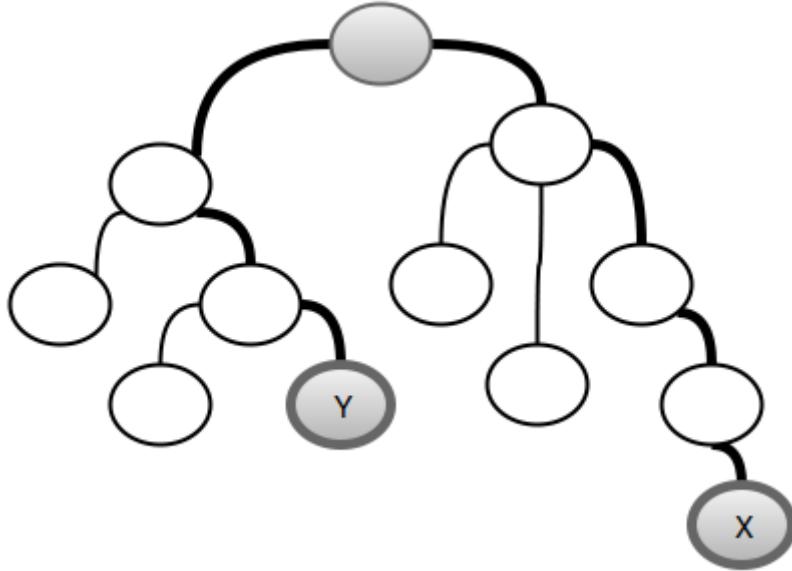


এটা বের করা খুব সহজ, এজন্য তোমার জানতে হবে বিএফএস বা ডিএফএস এর যে কোন একটা। আনডিরেক্টেড ট্রি তে যেকোন নোডকেই রুট ধরা যায়, আমরা মনে করি উপরের ধূসর নোডটা ট্রি এর রুট।

আমাদের প্রথম 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।

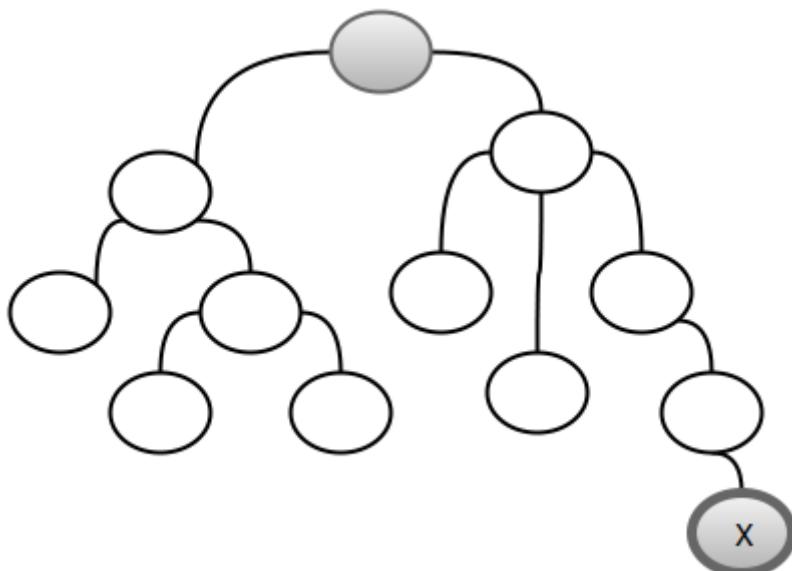


X আর Y এর মধ্যকার দূরত্বই ত্রি এর ডায়ামিটার! উপরের ছবিতে ডায়ামিটার ৭।

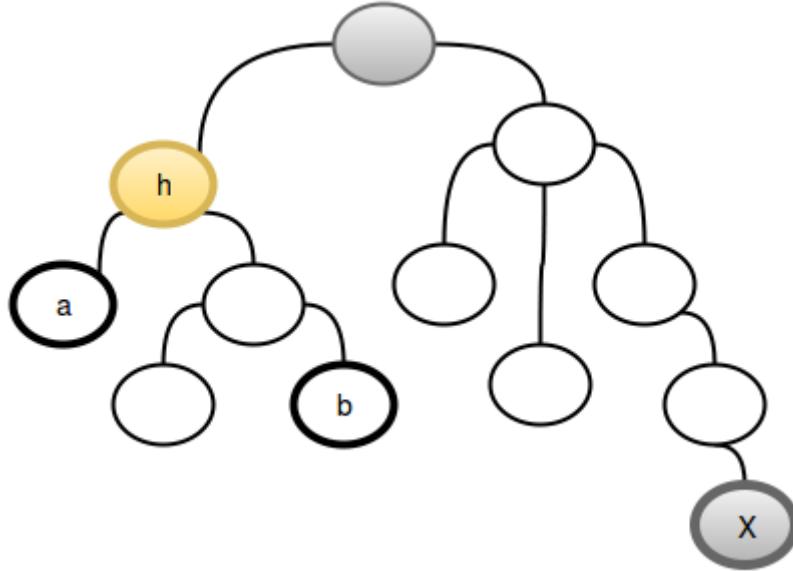
### প্রমাণ:

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 ত্রি এর ডায়ামিটার।

এটা আমরা প্রমাণ করবো “প্রক্রফ বাই কনট্রাডিকশন” এর সাহায্যে। এটা বহুল ব্যবহৃত একটা পদ্ধতি। আমরা প্রমাণ করতে চাই X নিশ্চয়ই কোনো ডায়ামিটারের প্রান্ত। কিন্তু তা যদি না হয়, অর্থাৎ X কোনো ডায়ামিটারের প্রান্ত 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 আবিট্রালি(arbitrarily) দুটি নোড a আর b সিলেক্ট করি। নোড দুটির মধ্যকার পাথ রুটের কাছে যে নোডে এসে মিলবে সেটার নাম দেই h, অর্থাৎ h হলো নোড দুইটার কমন অ্যানসেস্টর। (অনেকেই হয়তো বুঝে গেছে 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 হবেনা কারণ রুট থেকে সবথেকে দূরের নোড হলো X।

অর্থাৎ:

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

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)

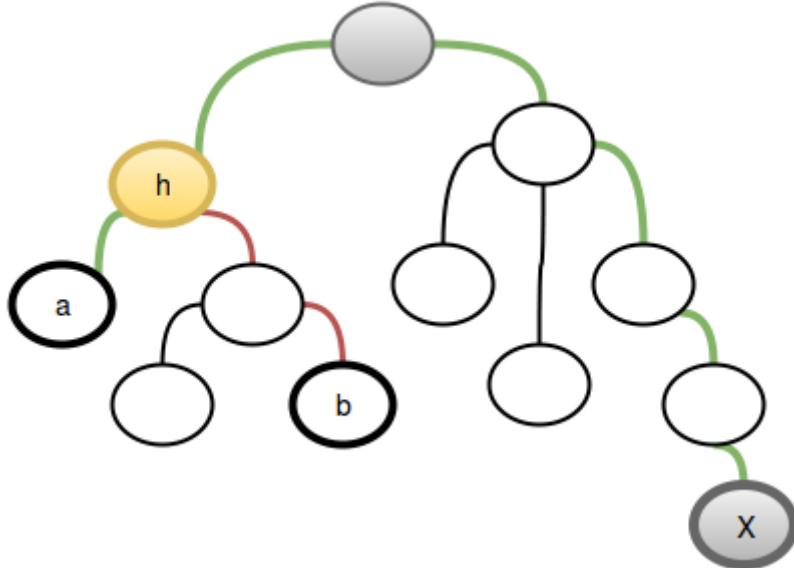
$$| \quad distance(h,b) \leq distance(root,b)$$

তাহলে,

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

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

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 পাথ 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 ত্রি এর ক্ষেত্রে করবে। জেনারেল গ্রাফে লংগেস্ট পাথ বের করার প্রবলেম [এন-পি হার্ড](#), আর ডায়ামিটার বের করার প্রবলেম লংগেস্ট পাথ 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>

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# লংগেস্ট পাথ প্রবলেম

 shafaetsplanet.com/planetcoding/

শাফায়েত

জুলাই ২১, ২০১৬

তোমাকে একটা আনওয়েটেড গ্রাফ এবং একটা সোর্স নোড দেয়া আছে। তোমাকে সোর্স নোড থেকে সর্বোচ্চ দৈর্ঘ্যের পাথ বের করতে হবে। এটাই হলো লংগেস্ট পাথ প্রবলেম। প্রশ্ন হলো কিভাবে প্রবলেমটা সলভ করবে?

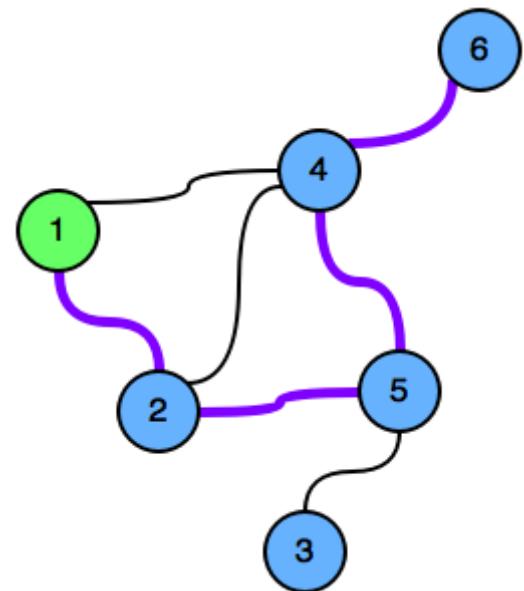
এটা আমার খুবই প্রিয় একটা ইন্টারভিউ প্রশ্ন। এখন পর্যন্ত প্রায় ৭-৮টা ইন্টারভিউতে আমি এই প্রশ্ন করেছি, মাত্র ২জন সহজেই উত্তর দিতে পেরেছে, ১জন হিন্টস দেয়ার পর পেরেছে, বাকিরা সবাই ভুল উত্তর দিয়েছে। অ্যালগোরিদম কোর্সে এ+ অনেকেই পায়, কিন্তু এধরণের প্রশ্ন করলে বোঝা যায় ক্যান্ডিডেট আসলে কতখানি জানে।

প্রশ্নটা করার পর সবাই প্রথমে যে ভুল করে সেটা হলো জিঞ্জেস করে না সর্বোচ্চ দৈর্ঘ্যের পাথের সংজ্ঞা কি, আমি চাইলে দুটি নোডের মধ্যে অসীম সংখ্যকবার আসা-যাওয়া করে দৈর্ঘ্য অসীম করে ফেলতে পারি। তাই লংগেস্ট পাথের আরো ভালো কোনো সংজ্ঞা দরকার সমাধান করা জন্য। ধৰা যাক আমি এমন একটা লংগেস্ট পাথ চাই যেখানে কোনো নোডে একবাবের বেশি যাওয়া যাবে না। এখন আর দৈর্ঘ্য অসীম হবার কোনো সুযোগ নেই।

ছবিতে ১ নম্বর নোড থেকে সর্বোচ্চ দৈর্ঘ্যের পাথ দেখানো হয়েছে যার দৈর্ঘ্য হলো ৪ ( $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 6$ )।

সমাধান নিয়ে চিন্তা করার আগে পরের প্রশ্ন হলো গ্রাফটা কি ডিরেক্টেড নাকি আনডিরেক্টেড। ধরে নাও গ্রাফটা আনডিরেক্টেড এবং সর্বোচ্চ ১০০,০০০ টা নোড থাকতে পারে।

এই প্রবলেমের অনেক ধরণের ভুল সমাধান দিতে দেখেছি ক্যান্ডিডেটদের, যেমন:



- কেও কেও গ্রাফের এজগুলোর ওয়েট -1 ধরে শর্টেস্ট পাথ বের করার চেষ্টা করে। এটা কোনোভাবেই কাজ করবে না, ইনফিনিটি লুপে আটকে যাবে।
- কেও কেও ডায়নামিক প্রোগ্রামিং ব্যবহার করে সমাধান করতে চেষ্টা করে, এটাও কাজ করবে না এবং লুপে আটকে যাবে।
- ম্যাক্স ফ্লো ব্যবহার করতে চায় অনেকে কিন্তু সেটাও কাজ করবে না।

তাহলে সমাধান টা কি? মনে করি গ্রাফটার নোড সংখ্যা  $n$ । যেহেতু  $n$  এর মান অনেক বড় হতে পারে, তাই অবশ্য পলিনোমিয়াল সলিউশন দরকার। এখন এমন একটা গ্রাফের কথা চিন্তা করো যেটায় সোর্স থেকে লংগেস্ট পাথের দৈর্ঘ্য হলো  $n-1$ । যেমন নিচের গ্রাফটা:

এই গ্রাফটিতে লংগেস্ট পাথের দৈর্ঘ্য হলো  $n-1$ । যেখানে  $n=6$ । তারমানে হলো গ্রাফটাতে এমন একটা সোর্স নোড আছে যেখান থেকে যাত্রা শুরু করে সবগুলো নোড একবাবে করে ভ্রমণ করা যায় কোনো নোড পুনরাবৃত্তি না করেই। এ ধরণের পাথকে বলা হয় **হ্যামিল্টন পাথ**। তুমি যদি গ্রাফথিওরি নিয়ে কিছুটা পড়ালেখা করে থাকো তাহলে অবশ্য জানার কথা যে হ্যামিল্টন পাথ প্রবলেম একটা **এন-পি কমপ্লিট (NP-complete)** প্রবলেম। এন-পি কমপ্লিট কোনো প্রবলেমকে পলিনমিয়াল সময়ে সমাধান করার কোনো উপায় এখন পর্যন্ত কারো জানা নেই।

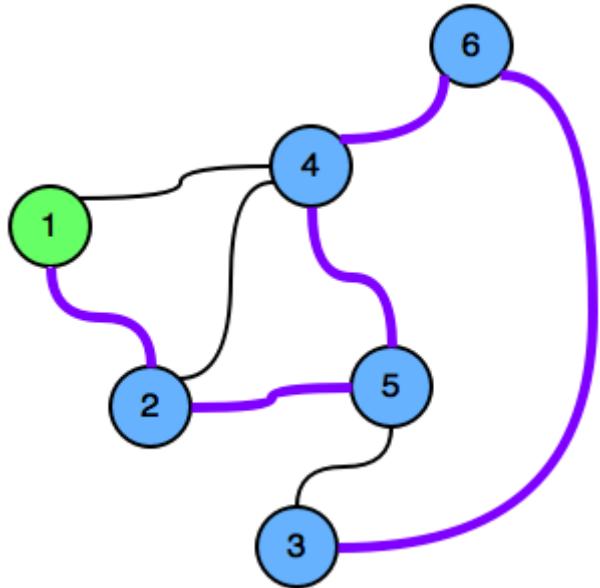
দেখতেই পাচ্ছ হ্যামিল্টন পাথ প্রবলেমকে খুব সহজেই লংগেস্ট পাথ প্রবলেমে কনভার্ট করা যায়। একটা গ্রাফে কোনো সোর্স থেকে এমন একটা লংগেস্ট পাথ বের করতে হবে যেন পাথের দৈর্ঘ্য হয়  $n-1$ , এটাই হ্যামিল্টন পাথ প্রবলেম। লংগেস্ট পাথ প্রবলেমকে পলিনমিয়াল সময়ে সমাধান করা গেলে হ্যামিল্টন পাথ প্রবলেমও সমাধান হয়ে যেত!

তাহলে দেখতেই পাচ্ছ যে লংগেস্ট পাথ প্রবলেমের কোনো পলিনমিয়াল সলিউশন নেই। নোড সংখ্যা যদি খুব অল্প হয় তাহলে **ব্যাকট্র্যাকিং** করে এই সমস্যা সমাধান করা সম্ভব, কিন্তু এটার কমপ্লেক্সিটি এক্সপোনেনশিয়াল(exponential)। লংগেস্ট পাথ হলো একটা এন-পি হার্ড প্রবলেম।

এখন প্রশ্ন হলো কেন হ্যামিল্টন পাথ প্রবলেম [এন-পি কমপ্লিট কিন্তু](#)  
লংগেস্ট পাথ এন-পি হার্ড? যদি এটার উত্তর না জেনে থাকো তাহলে  
আমার [এই লেখাটা](#) পড়ে ফেলো।

হ্যাপি কোডিং

AccessPress Staple | WordPress Theme: [AccessPress Staple](#)  
by [AccessPress Themes](#)



# HANDBOOK OF ALGORITHMS

## Section Dynamic Programming

Courtesy of  
*Shafaet Ashraf*

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-১(শুরুর কথা) \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-২ \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-৩ \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-৪ \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-৫ \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-৬ \_ লংগসেট কমন সাবসকিয়েন্স \_ শাফায়তেরে ব্লগ.pdf

ডাইনামিকি প্রযোগেরামং এ হাতখেড়ি-৭ \_ ম\_যাট্রকিস চেইন মাল্টিপ্লিকিশন \_ শাফায়তেরে ব্লগ.pdf

# ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-১(শুরুর কথা)

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

4/29/2012

কনটেস্ট প্রোগ্রামিং করতে গিয়ে আমরা অনেক কিছুই শিখি, আমাদের গ্রাফ থিওরির অ্যালগোরিদম গুলো শিখতে হয়, জটিল সব ডাটা স্ট্রাকচার চোখের পলকে ইমপ্লিমেন্ট করতে হয়, এমনকি জ্যামিতি পর্যন্ত শিখতে হয় আমাদের। তবে প্রায় সকলেই স্বীকার করবে আমরা যেসব শিখি তারমধ্যে সবথেকে শৈল্পিক একটা বিষয় হলো ডাইনামিক প্রোগ্রামিং। এই সিরিজে আমি কিছু ক্লাসিক ডাইনামিক প্রোগ্রামিং প্রবলেম নিয়ে বিস্তারিত আলোচনা করবো যেগুলো পড়ে তুমি অনেকগুলো প্রবলেম নিজে সলভ করে ফেলতে পারবে। আমি জানি এরপর তুমি নিজেই এগিয়ে যেতে পারবে।

(এই সিরিজটি পড়া শুরুর করার আগে তোমাকে যেটা শিখতে সেটা হলো রিকার্শন। তুমি এটা শিখতে পারবে ফাইম ভাইয়ের [দারুণ এই লেখাটি](#) পড়ে, লেখাটিতে ডাইনামিক প্রোগ্রামিং নিয়েও আলোচনা করা হয়েছে।)

ডাইনামিক প্রোগ্রামিং কোনো নির্দিষ্ট অ্যালগোরিদম নয় বরং একটি প্রবলেম সলভিং টেকনিক যা ব্যবহার করে বিভিন্ন সমস্যাকে খুবই কম সময়ের মধ্যে সলভ করা যায়। রিয়েল লাইফ অনেক প্রবলেম কম সময়ের মধ্যে একমাত্র ডাইনামিক প্রোগ্রামিং ব্যবহার করেই করা সম্ভব এবং বিশেষ করে বাংলাদেশের ন্যাশনাল কনটেস্টগুলোতে ডাইনামিক প্রোগ্রামিং প্রবলেমের একাধিক প্রবলেম সবসময় থাকে।

এখন কাজের কথায় আসি। শুরুতেই কঠিন শব্দ দিয়ে ডাইনামিক প্রোগ্রামিং এর সংজ্ঞা বললে যে কেও ভয় পেয়ে যাবে, তাই আমরা শুরু করবো ছেট্‌ট একটা উদাহরণ দিয়ে, তারপর কিছু ফর্মাল কথাবার্তায় যাবো। ইটালিয়ান গণিতবিদ Leonardo Pisano Bigollo যাকে আমরা ফিবোনাচি নামে চিনি খরগোশের বংশবৃদ্ধি পর্যবেক্ষণ করতে গিয়ে একটা নাস্তার সিরিজ আবিষ্কার করে বসলেন। সিরিজটি এরকম:

0, 1, 1, 2, 3, 5, 8, 13, 21.....

লক্ষ্য করো ১ম দুটি সংখ্যা ছাড়া প্রতিটি সংখ্যা হলো আগের দুটি সংখ্যার যোগফল। আমরা একটি ফাংশন কল্পনা করি  $F(n)$  যা  $n$  তম ফিবোনাচি সংখ্যা রিটার্ন করে, অর্থাৎ  $F(n)=n$  তম ফিবোনাচি সংখ্যা।

তাহলে:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(2) &= F(1) + F(0) = 1 \\ F(3) &= F(2) + F(1) = 2 \\ F(4) &= F(3) + F(2) = 3 \end{aligned}$$

তাহলে আমরা জেনারেলভাবে বলতেই পারি:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

এখানে  $F(0)$  এবং  $F(1)$  হলো আমাদের রিকার্সিভ ফাংশনের জন্য যে বেসকেস দরকার সেটা। আমরা খুব সহজে C++ এ কোড লিখে ফিবোনাচি সংখ্যা বের করতে পারি:

C++

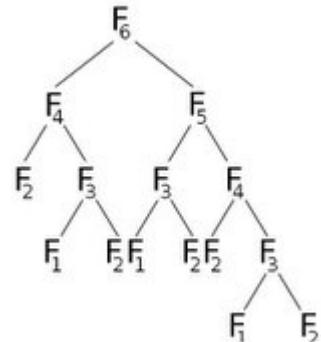
```

1 int F( int n ) {
2   if( n == 0 ) return 0;
3   if( n == 1 ) return 1;
4   return F( n-1 ) + F( n-2 );
5 }
```

এখন ধরো তুমি  $F(6)$  কে কল করলে। সে আবার কল করবে  $F(5)$  আর  $F(4)$  কে, এবা আরও কিছু ফাংশনকে কল করবে। আমরা ছবি আকারে দেখি কে কাকে কল করবে:

এখন খুব ভালো করে কিছু ব্যাপার লক্ষ্য করো। ধরো  $F(6)$  আগে কল করছে  $F(5)$  কে এবং তারপরে কল করছে  $F(4)$  কে এবং সবশেষে এ দুটোর যোগফল তোমাকে দিচ্ছে। এখন ছবিতে দেখো,  $F(5)$  কল দিচ্ছে  $F(4)$  এবং  $F(3)$  কে। যখন  $F(5)$  হিসাব করা শেষ তখন অবশ্যই  $F(5)$  যাদেরকে কল দিচ্ছে তাদের হিসাব করাও শেষ হয়ে গেছে, তাই নয় কি? তারমানে  $F(5)$  হিসাব করতে গিয়ে  $F(4)$  এবং  $F(3)$  আমরা হিসাব করে ফেলেছি, এমনকি  $F(2)$  ও হিসাব করে ফেলেছি। ( $F(1)$  আর  $F(0)$  বেস কেস, তাদের মান আমরা শুরু থেকেই জানি)।

এখন  $F(6)$  কিন্তু  $F(5)$  কে কল করার পর আবার  $F(4)$  কে কল করছে। কিন্তু আমরাতো  $F(4)$  এর মান হিসাব করেই ফেলেছি, কি দরকার আবার হিসাব করার? আগের মানটাই কি আমরা আবার ব্যবহার করতে পারিনা?



এখানেই আমরা একটা ছোট ট্রিকস খাটাবো। কোনো একটি ফাংশনের হিসাব শেষ হয়ে গেলে আমরা একটি টেবিলে মানটি সেভ করে রাখবো। পরবর্তিতে একই ফাংশনকে আবার কল করলে আমরা পুরো হিসাব আবার না করে আগের মানটি রিটার্ন করে দিবো।

C++

```

1 int dp[20];
2 //শুরুতে ডিপি অ্যারের সবগুলো ইনডেক্সে -1 বসিয়ে নাও
3 //যেমন for(int i=0;i<20;i++)dp[i]=-1; (এই কাজটা মেইন ফাংশনে করবে)
4 //কোনো ঘরে -1 থাকা মানে হচ্ছে ঘরটা খালি
5 int F( int n ) {
6   if( n == 0 ) return 0;
7   if( n == 1 ) return 1;
8   if( dp[n]!=-1 ) return dp[n];
9   else
10  {
11    dp[n] = F( n-1 ) + F( n-2 );
12    return dp[n];
13  }
14 }
```

উপরের কোডে আমরা সব কাজ প্রথম কাজের মতো করছি শুধু সামান্য একটু memoization টেকনিক ব্যবহার করেছি। শুরুতে  $dp$  অ্যারের সবগুলো পজিশনে -1 রেখে নাও, তারমানে সবগুলো পজিশন খালি। তুমি যখন  $F(4)$  কল করবে তখন আগে দেখো  $dp[4]$  খালি নাকি, খালি হওয়া মানে এখনও হিসাব করা হয়নি, তাই  $F(4)$  এর মান হিসাব করে  $dp[4]$  এ রেখে রিটার্ন করে দাও। যদি খালি না হয় তারমানে আগেই  $dp[4]$  এর মান হিসাব করা হয়ে গেছে!! তাহলে তুমি শুধু  $dp[4]$  রিটার্ন করে দাও।

যে সহজ কাজটা করে আমরা সময় বাচিয়ে ফেললাম সেটাকেই কম্পিউটার প্রোগ্রামার কঠিন ভাষায় বলে ডাইনামিক প্রোগ্রামিং বা ডিপি। আমরা যদি মানগুলো অ্যারেতে সেভ করে না রাখতাম তাহলে একি ফাংশন বারবার কল হয়ে প্রচুর সময় নষ্ট করতো, তুমি নিজে  $F(20)$  এর জন্য একবার সেভ করে আরেকবার না করে পরীক্ষা করে দেখতে পারো পার্থক্যটা কতখানি।

**৩টি বৈশিষ্ট্য থাকলে একটি প্রবলেমকে ডিপি দিয়ে সলভ করা সম্ভব:**

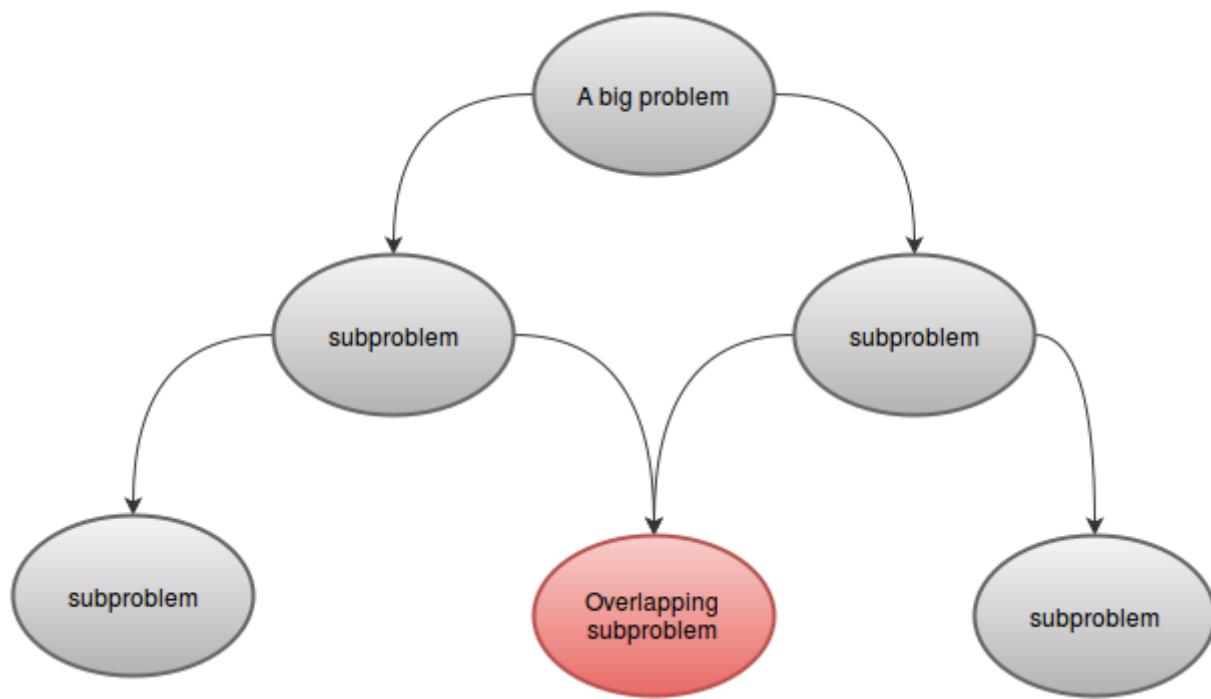
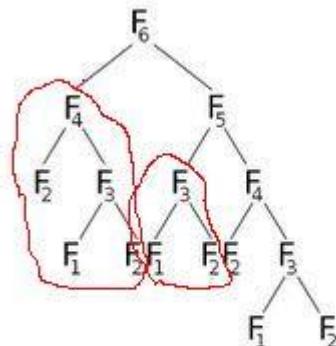
১. **subproblems:** প্রবলেমগুলোকে ছোটো এক বা একাধিক সাবপ্রবলেমে ভাগ করা যেতে হবে। যেমন  $F(4)$  এর মান বের

করার প্রবলেমটাকে আমরা F(3) এবং F(2) এই দুইভাগে ভাগ করে ফেলতে পারি। সাবপ্রবলেমগুলো মূল প্রবলেমের অনুকূপ হয়, অর্থাৎ যেভাবে মূল প্রবলেম সলভ করা যায় সেভাবেই সাবপ্রবলেম সলভ করা যায়।

**২. overlapping subproblems** থাকে। এটার মানে হলো সাবপ্রবলেমের গুলোর মধ্যে কমন অংশ থাকবে যে কারণে একই ফাংশন একাধিক ফাংশন হতে কল হবে।

যেমন F(6) এবং F(5) এ দুটো প্রবলেমের সাবপ্রবলেম গুলোর মধ্য F(4), F(3) ইত্যাদি ওভারল্যাপিং। এ কারণেই আমরা অ্যারেতে ভ্যালু সেভ করে রাখি। নিচের ছবিটি দেখো:

বামের এই লাল দিয়ে ঘেরা সাবট্রি গুলো ডানেও আছে। একাধিক স্থানে থাকার কারণেই এদের বলছি ওভারল্যাপিং সাবট্রি বা সাবপ্রবলেম। ফলে ডানপাশে আমরা একবার এগুলোকে কম্পিউট করে টেবিলে জমিয়ে রাখছি। পরে বামের অংশে এসে স্রেফ টেবিল থেকে মানটা দেখে নিচ্ছি। জেনারেলাইজ করে আরেকটি ছবি দিলাম:



ছবি: লাল নোডটা দুটো সাবপ্রবলেমকে ওভারল্যাপ করেছে

**৩. optimal substructure:** এটা ফিবোনাচির উদাহরণ দিয়ে বুঝানো কঠিন। ধরো তোমাকে কোনো একটা ফাংশন G(x) এর ভ্যালুকে মিনিমাইজ করতে বলেছে আর তুমি এটা জানো যে G(x) নির্ভর করে G(y) এবং G(z) এর উপর। এখন যদি G(y) এবং G(z) কে মিনিমাইজ করে G(x) কে মিনিমাইজ করা যায় তাহলে প্রবলেমটির optimal substructure প্রোপার্টি আছে। এমন হতেই পারে যে G(y) কে মিনিমাইজ এবং G(z) কে মিনিমাইজ বা ম্যাক্সিমাইজ কিছুই না করে G(x) কে মিনিমাইজ করা যায়, তাহলে optimal substructure প্রোপার্টি নেই। যদি সাবপ্রবলেমের অপটিমাল সলুশন থেকে মূল প্রবলেমের অপটিমাল সলুশন পাওয়া যায় তাহলেই শুধুমাত্র এই প্রোপার্টি আছে বলা যাবে।

ফিবোনাচির প্রবলেমটা আসলে ডাইনামিক প্রোগ্রামিং এর খুব বেশি ভালো উদাহরণ না কারণ কোনো কিছু ম্যাক্সিমাইজ বা মিনিমাইজ করতে হয়না, তবে শুরুতে বুঝানোর জন্য এটাই সবথেকে ভালো উদাহরণ।

প্রথম পর্ব এখানেই শেষ। পরবর্তী পর্বে ডিপির স্টেট নিয়ে কিছু আলোচনা করা হবে এবং 2d, 3d অ্যারে ব্যবহার করে ফাংশনের মান সংরক্ষন করা দেখানো হবে। এখন তোমার হোমওয়ার্ক (!! ) হবে binomial coefficient  $nCr$  এর মান ডাইনামিক প্রোগ্রামিং এর সাহায্যে বের করা। রিকার্শনটি হলো  $nCr = (n-1)Cr + (n-1)C(r-1)$ । বেস কেস বের করার দায়িত্ব ও তোমার।

পরবর্তী পর্বে সমস্যাটির সমাধান বলা হবে। hint: ২ডি টেবিলে মান সেভ করতে হবে। আর সলভ করা চেষ্টা করো এই প্রবলেমটি: [http://www.lightoj.com/volume\\_showproblem.php?problem=1006](http://www.lightoj.com/volume_showproblem.php?problem=1006), এটা সলভ করতে পারলে তুমি টিউটোরিয়ালটি ভালোমত বুঝেছ, না পারলে আরেকবার পড়ো এবং চিন্তা করো। ডিপি প্রবলেম সলভ করতে হলো প্রচুর চিন্তা করতে হবে, টিউটোরিয়াল পড়ে বেসিক জিনিস ছাড়া খুব বেশি কিছু শিখতে পারবেনা।

২য় পর্ব

# ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-২

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

মে ২৩, ২০১২

## (১ম পর্ব) (সবগুলো পর্ব)

১ম পর্বে আমরা জেনেছি ডাইনামিক প্রোগ্রামিং কাকে বলে, প্রবলেমে কি রকমের বৈশিষ্ট্য থাকলে সেটা ডাইনামিক প্রোগ্রামিং এর সাহায্যে সমাধান করা যায়। আমরা দেখেছি ডিপি দিয়ে কিভাবে ফিবোনাচি সংখ্যার রিকার্শনের রানটাইম অনেক কমিয়ে আনা যায়। তবে ডিপি এমন একটা জিনিস যে এতকিছু জেনেও তুমি কিছুই সমাধান করতে পারবেনা যদিনা খুব ভালো করে প্র্যাকটিস করো আর চিন্তা করো। তবে এটা শুনে ভয়ের একদমই কিছু নেই, প্র্যাকটিস করতে থাকলে কিছুদিন পর দেখবে অনেক সহজেই রিকার্সিভ ফাংশন বের করে ডিপি প্রবলেম সলভ করে ফেলতে পারছো, আমার কাজ হলো তোমাকে শুরু করিয়ে দেয়া। সব শেষে [hexabonacci](#) নামের একটি প্রবলেম সলভ করতে দিয়েছিলাম, আশা করি সবাই প্রবলেমটি খুব সহজেই সমাধান করে ফেলেছে। সমাধান করতে না পারলে আমি বলবো আগের পর্বটা আরেকবার খুব ভালোভাবে পড়ে ফেলতে আর কোনো জায়গায় না বুঝলে মন্তব্য অংশে জানাতে। প্রবলেমটির সাথে ফিবোনাচি প্রবলেমটার খুব একটা পার্থক্য নেই বলে আমি সমাধান নিয়ে আলোচনা করছিনা।

## ১। টা বন্তুর মধ্য থেকে r। টা বন্তু কত ভাবে নেয়া যায়?

এখন আমরা  $nCrnCr$  এর মান কিভাবে ডাইনামিক প্রোগ্রামিং এর সাহায্যে বের করা যায় সেটা নিয়ে আলোচনা করবো।  $n$  টা জিনিসের মধ্যে  $r$  টা জিনিস কতভাবে নেয়া যায় সেটাই  $nCrnCr$  বা  $nCr(n,r)nCr(n,r)$ । এখন যেকোনো একটি জিনিসের কথা চিন্তা কর। তুমি যদি জিনিসটা না নাও তাহলে বাকি  $n-1n-1$  টা জিনিসের মধ্য থেকে আরো  $r$  টা জিনিস নিতে হবে, সেটা করা যায়  $nCr(n-1,r)nCr(n-1,r)$  ভাবে। আর যদি তুমি জিনিসটা নাও তাহলে বাকি  $n-1n-1$  টা জিনিসের মধ্যে থেকে আরো  $r-1r-1$  টা জিনিস নিতে হবে, সেটা করা যায়  $nCr(n-1,r-1)nCr(n-1,r-1)$  ভাবে। এই দুইটার যোগফলই হলো মোট উপায়।

$$nCr = (n-1)Cr + (n-1)C(r-1) \\ nCr = (n-1)Cr + (n-1)C(r-1)$$

আমরা যদি মনে করি  $nCr(n,r)$  হলো একটি ফাংশন যা  $n$  আর  $r$  এর মানকে প্যারামিটার হিসাবে প্রহণ করে তাহলে আমরা উপরের রিলেশন তাকে এভাবে লিখতে পারি:

$$nCr(n,r) = nCr(n-1,r) + nCr(n-1,r-1) \\ nCr(n,r) = nCr(n-1,r) + nCr(n-1,r-1)$$

এবার হয়তো জিনিসটা আরো স্পষ্ট হয়েছে তোমার কাছে। কিন্তু রিকার্শনটা শেষ হবে কখন? অর্থাৎ বেস কেস কি? আমরা জানি  $nCr(n,1)=n$  এবং  $nCr(n,n)=1$ । এ দুটি শর্ত বেস কেস হিসাবে ব্যবহার করলে কোডটা হবে:

Python

```
1 def nCr(n, r):
2     if r==1: return n
3     if n==r: return 1
4     return nCr(n-1, r) + nCr(n-1, r-1)
```

ফাংশন কল গুলো লক্ষ্য করলে দেখতে পাবে ফিবোনাচির মতো এখানেও একটি ফাংশন বার বার কল হচ্ছে:

আমরা এই অতিরিক্ত ফাংশন কল সহজেই এড়াতে পারি আগের মতো একটা টেবিল রেখে। পার্থক্য হলো এবার টেবিলটা হবে ২-ডি, আর কোনো পার্থক্য নেই। আগে টেবিলটার সব ঘরে -1-1 রেখে নিবো, -1-1 দিয়ে বুঝাচ্ছে ঘরটি খালি আছে।

Python

```

1  table = [[-1 for i in range(0,50)] for i in range(0, 50)] //Create a 50*50 table with -1 in each cell
2  def nCr(n, r):
3      if r==1: return n
4      if n==r: return 1
5      if table[n][r] != -1:
6          return table[n][r]
7      table[n][r] = nCr(n-1, r) + nCr(n-1, r-1)
8      return table[n][r]

```

আমাদের সবগুলো ডিপি কোডই মোটামুটি একটা ফরমেট মেনে চলবে, সেটা এরকম:

```

1  def function(state1, state2, ....,
2  staten)
3
4  Return if reached any base case
5  Check table and return if the value
6  is already computed
    Calculate the value recursive for this
    state
    Save the value in the table and
    return

```

$nCr(n, r)$  এর ক্ষেত্রে প্যারামিটার ছিলো  $nn$  এবং  $rr$ । বুঝতেই পারছো যতগুলো প্যারামিটার থাকবে মান টেবিলে সেভ করার জন্য তত ডাইমেনশনের টেবিল লাগবে। সেটা বেশি লাগলে মেমরিও বেশি লাগে। অ্যারের প্রতিটি ডাইমেনশনের সাইজ প্রয়োজনমতো প্যারামিটার অনুসারে কমবেশি হবে। প্যারামিটারকে আমরা সাধারণত বলে থাকি স্টেট(State)।

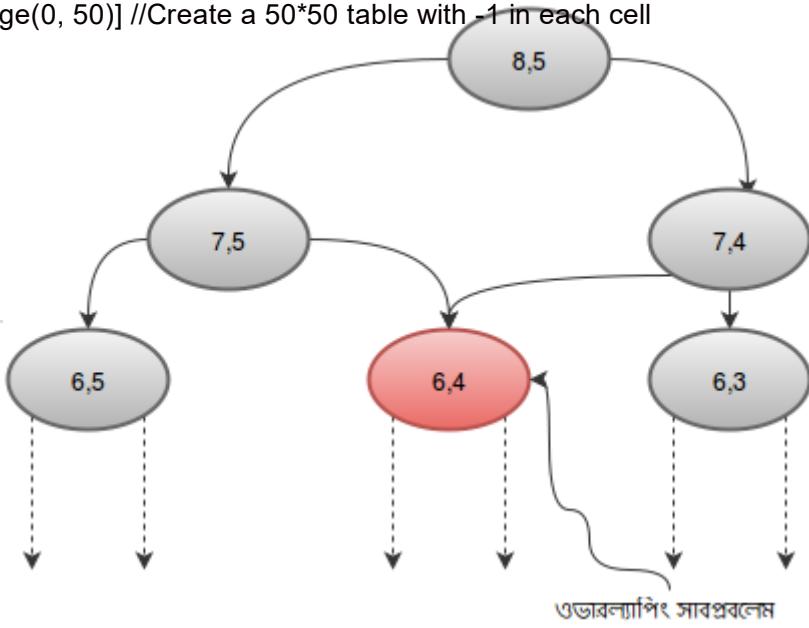
ডিপিতে সবথেকে গুরুত্বপূর্ণ অংশ হলো স্টেট বুঝতে পারা। ধরো তোমাকে ৫০মিনিটের মধ্যে একটা ক্লাস ধরতে হবে। এখন রাস্তায় তুমি কোন অবস্থায় আছো সেটা আমি জানতে পারবো যদি তুমি আমাকে দুটি তথ্য দাও: তুমি এই মুহূর্তে কোন জয়গায় আছো আর তুমি বাসা থেকে বের হবার পর কয় মিনিট পার হয়েছে। যেমন হয়তো তুমি ফার্মগেটে আছো আর বাসা থেকে বের হবার পর ২০ মিনিট পার হয়েছে। তুমি কি রঙের জামা পড়েছো বা তুমি কোন জুতা পড়েছো এটা কিন্তু এখানে গুরুত্বপূর্ণ না তাই এটা “স্টেট” এর মধ্যে পড়েনা।

## ০-১ ন্যাপস্যাক

কোনো এক রাতে তুমি চুরি করতে বের হলে!! বন্ধুর বাসায় জানালা দিয়ে ঢুকে দেখলে প্রচুর জিনিসপত্র, কিন্তু তোমার চুরি করার থলেতে জায়গা আছে মাত্র ১০ইউনিট, এর বেশি নিলে থলে ছিড়ে যাবে। প্রতিটা জিনিসের ওজন আছে আর একেক জিনিসের মূল্যও একেকরকম।

- ১. মানিব্যাগ: ১ পাউন্ড, ১২০টাকা
- ২. কোরম্যানের-বই: ৭ পাউন্ড, ৪০০টাকা
- ৩. ডিভিডি কালেকশন: ৪ পাউন্ড, ২৮০ টাকা
- ৪. ফেলুদা-সমগ্র: ৩ পাউন্ড, ১৫০টাকা
- ৫. ফুটবল: ভর: ৪ পাউন্ড, ২০০টাকা

কোনো জিনিস নিলে পুরোটাই নিতে হবে, ৪টি ডিভিডির ২টি তুমি নিতে পারবেনা, ফেলুদা সমগ্রের অর্ধেক ছিড়ে আনতে পারবেনা। প্রথমদিন চুরি করতে গিয়েছো এই জন্য কিছু না ভেবেই তুমি ফটাফট দামি জিনিসগুলো ভরতে থাকলে। প্রথমেই তুমি ৪০০টাকার কোরম্যানের বই নিয়ে নিলে, তারপর ১৫০টাকার ফেলুদা সমগ্র নিয়ে বাসায় ফিরে আসলে, তোমার লাভ হলো ৫৫০ টাকা। বাসায় এসে হিসাব করে দেখলে তুমি যদি **লোভীর মতো (greedy)** দামি জিনিসগুলো আগে না নিয়ে একটু ভেবে-চিন্তে নিতে তাহলে ২৮০টাকার ডিভিডি, ২০০টাকার ফুটবল আর ১২০টাকার মানিব্যাগ নিয়ে ফিরতে পারতে, তোমার লাভ হতো ৬০০টাকা। **greedy** অ্যালগোরিদমে অপটিমাল রেজাল্ট না পাওয়ায় তুমি চিন্তা করলে সবরকমের কমিশনেশনে চেষ্টা করবে, প্রতিবার একটা করে জিনিস থলেতে ভরবে আর দেখবে আর কত লাভ করা যায়, প্রয়োজনে জিনিসটা থলে থেকে



নামিয়ে রেখে আরেকটি নিয়ে চেষ্টা করবে। এবং সুবিধার জন্য তুমি লিস্টের সিরিয়াল অনুযায়ী জিনিস নিয়ে চেষ্টা করবে, ৩ নম্বর জিনিসের পরে ১ নম্বর জিনিস নিতে চেষ্টা করবেন। কোনো সময় তোমার অবস্থা বুঝাতে ২টা তথ্যই যথেষ্ট।

- 1. তুমি এখন কত নম্বর জিনিস ট্রাই করছো।
- 2. এখন পর্যন্ত তুমি কত ইউনিট জিনিস নিয়েছো।

তুমি একটা ফাংশন লিখে ফেললে যেটা জিনিসপত্রের লিস্ট থেকে অপটিমাল রেজাল্ট বের করে দিতে পারে। ফাংশনটির রিটার্ন টাইপ আর প্যারামিটার হবে এরকম:

```
1 func(int i,int w)
```

মনে করি। নম্বর বস্তুটির ভর হলো weight[i] আর মূল্য cost[i]। আর ব্যাগের capacity=CAP। প্রতিবার তুমি। নম্বর জিনিসটি নিতে পারো যদি ব্যাগে জায়গা থাকে, অথবা তুমি। নম্বর জিনিসটি না নিয়ে i+1 তম জিনিস ট্রাই করতে পারো।। নম্বর জিনিসটি যদি তুমি না ও তাহলে লাভ হবে cost[i] + পরবর্তি স্টেটে লাভ, তাহলে আমরা cost[i] যোগ করে পরবর্তি স্টেটে চলে গিয়ে কত লাভ হয় সেটা হিসাব করবো।।

```
1 if(w+weight[i]<=CAP)
2   profit1 = cost[i] + func(i + 1,w + weight[i])
3 else
4   profit1=0;
```

। নম্বর জিনিসটি যদি তুমি না না ও তাহলেও লাভ বেশি হতে হবে তাই সেটাও আমাদের হিসাব করতে হবে:

```
1 profit2=func(i+1,w) //নতুন ওজন যোগ হচ্ছেনা,i+1 তম বস্তু নিয়ে ট্রাই করছি।
```

লক্ষ্য করো এক্ষেত্রে কারেন্ট প্রফিট বা ওজনের কোনো পরিবর্তন হচ্ছেনা, কোনো কিছু না নিয়েই পরের স্টেটে গিয়ে দেখছি লাভ কত হবে। তাহলে সহজেই বোঝা যাচ্ছে অপটিমাল রেজাল্টের জন্য আমাদের profit1 আর profit2 এর মধ্যে বড়টা নিতে হবে,সেই মানটা আমরা রিটার্ন করে দিবো।।

```
1 return max(profit1,profit2)
```

তাহলে যদি তুমি func(1,0) কল করো তাহলে রিকার্শন তোমাকে রেজাল্ট বের করে দিবে। func(1,0) কল করার কারণ হলো শুরুতে তুমি ।। নম্বর জিনিসটা নিয়ে ট্রাই করবে এবং শুরুতে ব্যাগে সম্পূর্ণ খালি। যখন সবগুলো জিনিস নিয়ে ট্রাই করা হয়ে যাবে তখন রিকার্শন শেষ হবে। যদি nn টি জিনিস থাকে তাহলে বেস কেস হবে:

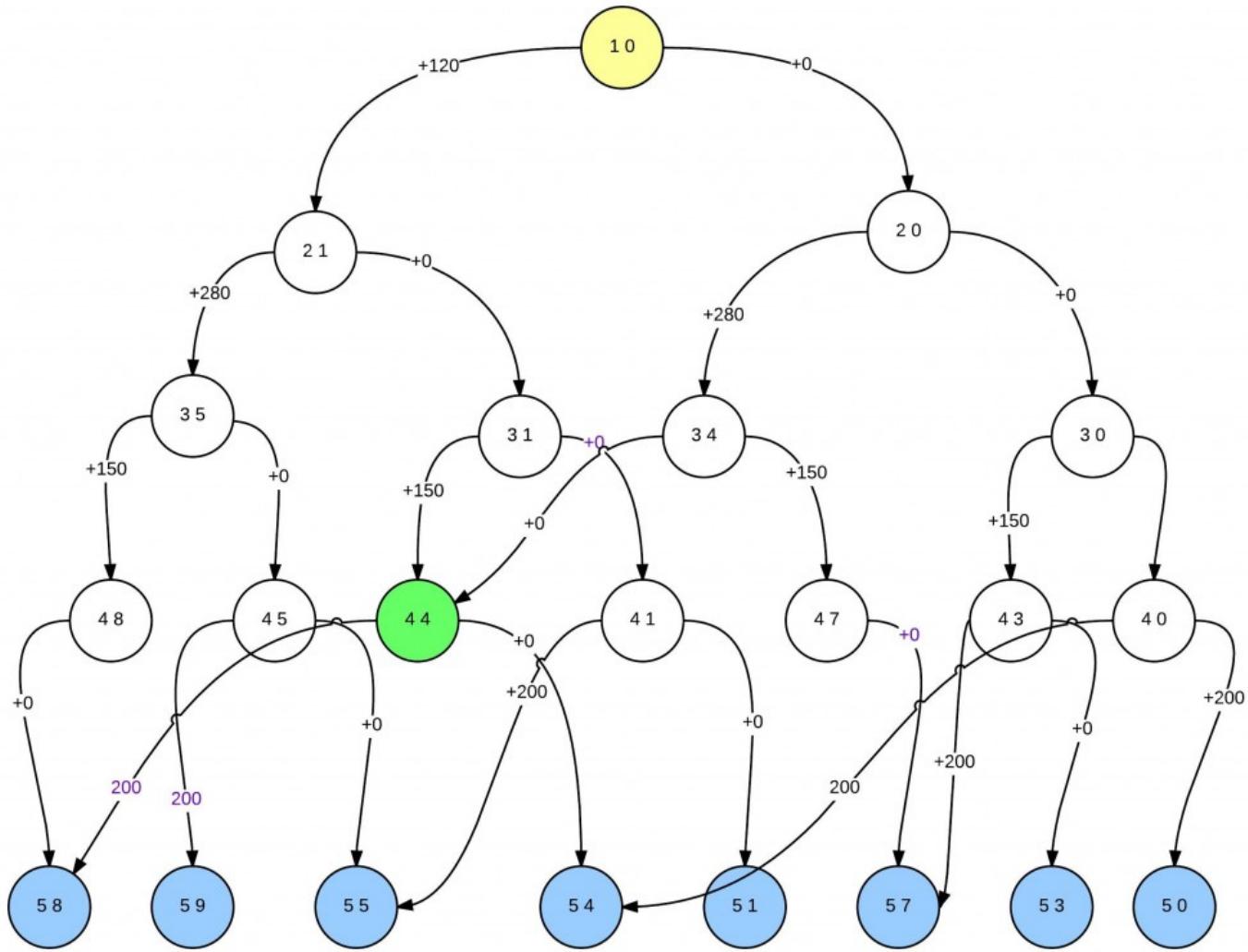
```
1 if(i==n+1) //Base Case
2   return 0
```

শুন্য রিটার্ন করছি কারণ সবকিছু নেয়া হয়ে গেলে আর প্রফিট করা সম্ভবনা। তোমার বোঝার সুবিধার জন্য সম্পূর্ণ একটা কোড দিয়ে দিলাম:

এই কোডে উপরের উদাহরণের ইনপুটটা নিচের মতো করে দিলে দিলে ৬০০ আউটপুট আসবে

```
4 10 // ৫ টা জিনিস, ১০ ক্যাপাসিটি
1 120 // প্রতিটা জিনিসের ওজন এবং দাম
4 280
3 150
4 200
```

আমরা উপরের ইনপুটের জন্য কিভাবে ফাংশন কলে হচ্ছে তার একটা সিমুলেশন দেখি:



ছবিতে প্রতিটা নোড প্রকাশ করছে আমরা এখন কোন জিনিসটা নিয়ে কাজ করছি এবং ব্যাগে কত ওজনের জিনিস আছে। তারপর জিনিসটা একবার ব্যাগে ভরে(যদি জায়গা থাকে) নতুন নোডে যাচ্ছ, আরেকবার না ভরে নতুন জায়গায় যাচ্ছ। নীল রঙ এর নোডগুলো হলো বেস কেস, কারণ আর কোনো জিনিস নেয়ার মত নেই। লক্ষ্য কর সবুজ রঙ এর নোডটায় একাধিক পথে আসা যায়, তাই আমরা প্রতিটি নোডের রিটার্ণ ভ্যালু ডিপি অ্যারেতে সেভ করে রাখবো।

### কমপ্লেক্সিটি:

আমাদের কোডে ভিন্ন স্টেট সংখ্যা  $n \times capn \times cap$  টা যেখানে  $nn$  হলো জিনিস সংখ্যা আর  $capcap$  হলো ব্যাগের ক্যাপাসিটি। ফাংশনের ভিতর কোনো লুপ নেই। তাই মোট  $O(n \times cap)$  টাইম কমপ্লেক্সিটিতে কোডটা চলবে। ডিপি অ্যারের আকার হবে  $n \times capn \times cap$ , তাই স্পেস কমপ্লেক্সিটি ও  $O(n \times cap)$ ।

এখন ছেট্ট একটা প্রশ্ন, ডিপি প্রবলেমে টাইম আর স্পেস কমপ্লেক্সিটি ভিন্ন হবে কখন?

এটাই হলো ক্লাসিকাল 0-1 knapsack প্রবলেম, ন্যাপস্যাক শব্দটির অর্থ হলো থলে। 0-1 নাম দেয়ার কারণ হলো তুমি কোনো জিনিস অর্ধেক নিতে পারবেনা, হয় পুরোটা নিবে অথবা নিবেনা। fractional-knapsack বলে আরেক ধরণের প্রবলেম আছে, স্টোর আলোচনা অন্য কোনোদিন হবে। ন্যাপস্যাকে সলিউশন প্রিন্ট করা শিখতে [সিরিজের চতুর্থ পর্বটা](#) দেখো।

এখন তোমার এখন কাজ হলো এটার সম্পূর্ণ কোড ইমপ্লিমেন্ট করা। [uva 10130](#) প্রবলেমটি সলভ করতে চেষ্টা করো।

ডাইনামিক প্রোগ্রামিং এ ভালো করার একমাত্র উপায় হলো প্রচুর চিন্তা করা। তোমাকে অনুরোধ করবো পরবর্তি পর্ব পড়ার আগে অবশ্যই আগের পর্বের সমস্যাগুলো নিয়ে খুব ভালো করে অন্তত ১দিন চিন্তা করবে।

(সবগুলো পর্ব)

[পরের পর্ব-কয়েন চেঞ্চ, রক ক্লাইম্বিং](#)



## ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-৩ (কয়েন চেঞ্জ + রক ক্লাইম্বিং)

 shafaetsplanet.com/planetcoding/

শাফায়েত

জুলাই ১৩, ২০১২

আগের পর্যবেক্ষণে পদে থাকলে তুমি এখন ডাইনামিক প্রোগ্রামিং নিয়ে বেসিক ব্যাপারগুলো কিছুটা শিখে গিয়েছো, যত প্রবলেম সলভ করবে তত দক্ষতা বাঢ়বে। ডিপিতে আসলে কোনো নির্দিষ্ট অ্যালগোরিদম না থাকায় আমাদের চিন্তা করতে হয় অনেকে বেশি, একেকটি ডিপি প্রবলেম একেক ধরণের, তবে তুমি যদি ন্যাপস্যাক, কয়েন চেঞ্জের মতো ক্লাসিক কিছু ডিপি প্রবলেমের সলিউশন জানো তাহলে তুমি বুঝতে পারবে কিভাবে তোমার চিন্তাকে এগিয়ে নিয়ে যেতে হবে, কিভাবে ডিপির সেট নির্ধারণ করতে হবে, তখন তুমি নতুন ধরণের ডিপি প্রবলেমও সলভ করবে ফেলতে পারবে। আমি এরই মধ্যে nCrnCr নির্ণয় আর ০-১ ন্যাপস্যাকের ডিপি সলিউশন নিয়ে আলোচনা করেছি, আরো কিছু ক্লাসিক বা স্ট্যান্ডার্ড প্রবলেম নিয়ে সামনে আলোচনা করবো।

### কয়েন চেঞ্জ:

এখন আমরা দেখবো কয়েন চেঞ্জ প্রবলেম। আসলে ন্যাপসাক শেখার পরে কয়েন চেঞ্জ তুমি এমনিই পারবে তারপরেও লিখছি যাতে ব্যাপারগুলো আর পরিষ্কার হয়।

তোমার কাছে কিছু কয়েন আছে যাদের মূল্য 5, 8, 11, 15, 185, 8, 11, 15, 18 ডলার। প্রতিটা কয়েন অসীম সংখ্যকবার আছে, তুমি যেকোনো কয়েন যতবার ইচ্ছা নিতে পারো। তাহলে তোমার coin অ্যারেটা হতে পারে এরকম:

```
| coin[] = {5, 8, 11, 15, 18};
```

এখন তোমাকে এই কয়েনগুলো নিয়ে নির্দিষ্ট কোনো ভ্যালু বানাতে হবে। ধরি সংখ্যাটি হলো makemake। make=18make=18 হলে আমরা 5+5+85+5+8 এভাবে 1818 বানাতে পারি। তোমাকে বলতে হবে কয়েনগুলো দিয়ে ভ্যালুটি বানানো যায় নাকি যায়না।

প্রথমেই আমরা চিন্তা করি ডিপিতে সেট কি হবে। আমরা একটি একটি কয়েন নিয়ে সংখ্যাটি বানাতে চেষ্টা করতে থাকবো। তাহলে এই মুহূর্তে কোন কয়েন নিচ্ছি সেটা সেট সেট রাখতে হবে, আর আগে যেসব কয়েন নিয়েছি সেগুলোর মোট ভ্যালু কত সেটা রাখতে হবে। ফাংশনটির নাম call হলে প্রোটোটাইপ হবে:

```
1 int call(int i,int amount)
```

এরপর অনেকটা আগের মতোই কাজ। প্রথমে || নম্বর কয়েন নিতে চেষ্টা করবো:

```
1 if(amount+coin[i]<=make)
2   ret1=call(i,amount+coin[i]);
3 else
4   ret1=0;
```

এখানে i+1i+1 কল না করে আবার ii কল করছি কারন এক কয়েন অনেকবার নেয়া সন্তুষ্ট। যদি এক কয়েন একাধিক বার নেয়া না যেতো তাহলে i+1i+1 কে কল দিতাম। amount+coin[i]amount+coin[i] যদি makemake এর থেকে বড় হয় তাহলে কয়েনটি নেয়া সন্তুষ্ট। কয়েন যদি না নেই তাহলে আমরা পরবর্তী কয়েনে চলে যাবো:

```
1 ret2=call(i+1,amount);
```

ret1ret1 আর ret2ret2 এর কোনো একটি true হলেও make বানানো যাবে। তাহলে সবশেষে লিখবো:

```
1 return ret1|ret2;
```

আর বেসকেস হবে হলো, যদি সব কয়েন নিয়ে চেষ্টা করার পর makemake বানানো যায় তাহলে return 1, অন্যথায় return 0। সম্পূর্ণ কোড:

```

1 int coin[]={5,8,11,15,18}; //value of coins available
2 int make; //our target value
3 int dp[6][100];
4 int call(int i,int amount)
5 {
6     if(i>=5) { //All coins have been taken
7         if(amount==make) return 1;
8         else return 0;
9     }
10    if(dp[i][amount]==-1) return dp[i][amount]; //no need to calculate same state twice
11    int ret1=0,ret2=0;
12    if(amount+coin[i]<=make) ret1=call(i,amount+coin[i]); //try to take coin i
13    ret2=call(i+1,amount); //dont take coin i
14    return dp[i][amount]=ret1|ret2; //storing and returning.
15
16 }
17 int main()
18 {
19     // freopen("in","r",stdin);
20     while(cin>>make)
21     {
22         memset(dp,-1,sizeof(dp));
23         cout<<call(0,0)<<endl;
24     }
25     return 0;
26 }
```

এখন যদি তোমাকে বলা হতো যে কোনো একটি ভ্যালু কর্তব্য বানাতে হবে বলতে হবে তাহলে কি করতে? যেমন ১৮ বানানো ঘায় ২ ভাবে, এক্ষেত্রে  $ret1|ret2$  রিটোর্ন না করে  $ret1 + ret2$  রিটোর্ন করে দাও, তাহলে যত ভাবে বানানো ঘায় সবগুলো ঘোগ হয়ে যাচ্ছে। [uva-674](#) প্রবলেমটিতে এটাই করতে বলা হয়েছে, রাটপট সলভ করে ফেলো।

এবার মজার একটি অপটিমাইজেশন দেখো। আমরা প্রতিবার make ইনপুট নেয়ার পর ডিপি অ্যারে নতুন করে initialize বা ক্লিয়ার করেছি। যদি সেটা করতে না হতো তাহলে অনেক কম সময় লাগতো, কারণ একই মান বারবার হিসাব করা লাগবেনা। কিন্তু ক্লিয়ার করতে হচ্ছে কারণ ফাংশনটি বাইরের একটি প্লোবাল ভ্যারিয়েবলের উপর নির্ভরশীল, “`if(amount==make) return 1;`” এই লাইনটাই ঝামেলা করছে, makemake এর মান প্রতি কেসের জন্য আলাদা, তাই প্রতিবার নতুন করে সব হিসাব করতে হচ্ছে। আমরা যদি makemake কে একটা স্টেট হিসাবে রাখি তাহলে কাজ হয় কিন্তু স্টেট বিশাল হয়ে যায়। এর থেকে আমরা সমস্যাটাকে উল্টায় ফেলি। মনে করো তোমার কাছে শুরুতে ২০ টাকা আছে, বিভিন্ন অ্যামাউন্টের কয়েন দান করে দিয়ে তোমাকে শুন্য টাকা বানাতে হবে। কোডটা এবার হবে এরকম:

```

1 int coin[]={5,8,11,15,18}; //value of coins available
2 int make=18; //we will try to make 18
3 int dp[6][100];
4 int call(int i,int amount)
5 {
6     if(i>=5) { //All coins have been taken
7         if(amount==0) return 1;
8         else return 0;
9     }
10    if(dp[i][amount]==-1) return dp[i][amount]; //no need to calculate same state twice
11    int ret1=0,ret2=0;
12    if(amount-coin[i]>=0) ret1=call(i,amount-coin[i]); //try to take coin i
13    ret2=call(i+1,amount); //dont take coin i
14    return dp[i][amount]=ret1|ret2; //storing and returning.
15
16 }
17 int main()
18 {
19     // freopen("in","r",stdin);
20     memset(dp,-1,sizeof(dp));
21     while(cin>>make)
22     {
23         cout<<call(0,make)<<endl;
24     }
25     return 0;
26 }
```

খেয়াল করে দেখো ঠিক আগের মতোই কাজ করেছি, শুধু ঘোগ করার জায়গায় বিয়োগ করে makemake থেকে শুন্য বানানোর চেষ্টা করেছি। লাভটা হলো এখন ফাংশনটি কোনো পরিবর্তনশীল প্লোবাল ভ্যারিয়েবলের উপর নির্ভর করেনা, তাই মেইন ফাংশনে ডিপি অ্যারে লুপের মধ্যে ক্লিয়ার করা দরকার নাই। প্রতিবার কয়েন একই থাকছে বলে এই ট্রিকস্টা কাজ করছে, কয়েনের মান পরিবর্তন হলে কাজ করবেনা। ডিপির প্রবলেমে অনেকসময় টেস্টকেস অনেক বেশি দেয় যাতে বার বার ক্লিয়ার করলে টাইম লিমিট পাস না করে।

### কমপ্লেক্সিটি

ডিপি অ্যারের সাইজ হবে কয়েন সংখ্যা  $\times$  সর্বোচ্চ যত ভ্যালু বানাতে হবে। তাহলে মেমরি কমপ্লেক্সিটি  $O(\text{numberofcoin} \times \text{make})$ । যদি ভিতরে কোনো এক্সট্রা লুপ চলতো তাহলে সেটা ও টাইম কমপ্লেক্সিটির সাথে ঘোগ হতো।

লাইটওজেতে [1231 – Coin Change \(I\)](#) প্রবলেমে কিছু কয়েন দিয়ে একটি ভ্যালু কয়ভাবে বানানো ঘায় সেটা বের করতে বলেছে, তবে প্রতিটি কয়েন সর্বোচ্চ কর্তব্য ব্যবহার করা ঘায় সেটা বলা আছে, ন নম্বর কয়েন

বেতারব্যবহারকরায়াবেতেইকন্ডিশনটাদুইভাবতেমিহ্যান্ডলেকরতপেপারেো। তয়একটিস্টেটেরাখতপেপারেো। বেতারব্যবহারকরায়াবেতেইকন্ডিশনটাদুইভাবতেমিহ্যান্ডলেকরতপেপারেো।

তয়একটিস্টেটেরাখতপেপারেো। taken\_ystেটাবলদেভিবেষ্টেটাবলদেভিবেনম্বৰতমিকযবারনিয়েছেো, নম্বৰতমিকযবারনিয়েছেো, C\_{i\\$ বার ব্যবহার হয়ে গেলে পৰবৰ্তী কয়েনে চলে যাও।

```
1 int call(int i, int taken_i, int amount)
```

২য় উপায় হলো ফাংশনের ভিতরে C/C++ পর্যন্ত একটি লুপ চালিয়ে কয়েনটি যতবার নেয়া সন্তুষ্ট ততবার নিয়ে অ্যামাউন্টটি বানাতে চেষ্টা করো, এক্ষেত্রে মেমরি কম লাগবে। তাহলে আজকের ২য় কাজ হলো এই প্রবলেমটা সলভ করা।

কয়েন চেঞ্চ প্রবলেমের আরেকটি নাম হলো subset sum problem, কারণ কিছু নম্বৰের সেট থেকে একটি সাবসেট আমাদের নিতে হয় যেটাৰ যোগফল এক নির্দিষ্ট ভ্যালুৰ সমান।

### রক-ক্লাইন্সিং প্রবলেম

তোমাকে একটি ২ডি গ্ৰিড দেয়া হলো:

|    |    |    |
|----|----|----|
| -1 | 2  | 5  |
| 4  | -2 | 3  |
| 1  | 2  | 10 |

তুমি শুৰুতে আছো (0,0) সেলে। তুমি শুধু ৩দিকে যেতে পারো:

|                               |
|-------------------------------|
| ( <i>i</i> + 1, <i>j</i> )    |
| ( <i>i</i> + 1, <i>j</i> - 1) |
| ( <i>i</i> + 1, <i>j</i> + 1) |

প্ৰতিটি সেলে গেলে তোমার পয়েন্টের সাথে ওই সেলের সংখ্যাটি যোগ হয়। তুমি সৰ্বোচ্চ কত পয়েন্ট বানাতে পারবে? এই প্রবলেমকে রক ক্লাইন্সিং প্রবলেমও বলা হয়। উপৱের গ্ৰিডে সৰ্বোচ্চ পয়েন্ট  $7 = -1 + -2 + 10$ । এই প্রবলেমের জন্য:

**স্টেট:** তুমি এখন কোন সেল এ আছো  
**এক থেকে অন্য স্টেটে যাওয়াৰ উপায়:** প্ৰতিটা সেল থেকে ৩দিকে যাবার চেষ্টা করো, যেদিকে সৰ্বোচ্চ পয়েন্ট পাবে সেটা রিটাৰ্ণ কৰো।  
**বেসকেস:** যদি গ্ৰিডেৰ বাইৱে চলে যাও তাহলে আৱ কিছু নেয়া যাবেনা, শুণ্য রিটাৰ্ণ কৰো।

### C++

```
1 #define inf 1 << 28
2 int mat[10] = {
3     { -1, 2, 5 },
4     { 4, -2, 3 },
5     {
6         1, 2, 10,
7     }
8 };
9 int dp[10][10];
10 int r = 3, c = 3;
11 int call(int i, int j)
12 {
13     if (i >= 0 && i < r and j >= 0 and j < c) //if still inside the array
14     {
15         if (dp[i][j] != -1)
16             return dp[i][j];
17         int ret = -inf;
18         //try to move to 3 direction,also add current cell's point
19         ret = max(ret, call(i + 1, j) + mat[i][j]);
20         ret = max(ret, call(i + 1, j - 1) + mat[i][j]);
21         ret = max(ret, call(i + 1, j + 1) + mat[i][j]);
22         return dp[i][j] = ret;
23     }
24     else
25         return 0; //if outside the array
26 }
27 int main()
28 {
29     // READ("in");
30     mem(dp, -1);
31     printf("%d\\n", call(0, 0));
32     return 0;
33 }
```

৩দিকে মুভ কৰাৰ কল্ডিশন কেনো দেয়া হয়েছে? adjacent ৪টি সেলে মুভ কৰতে দিলে সমস্যা কোথায় হতো? তাহলে একটি সাইকেল তৈৰি হতো, একটি ফাংশনেৰ কাজ শেষ হবাব আগেই রিকাৰ্ণনে ঘুৱে ফাংশনটি আবাৰ কল হতো, এই সলিউশন কাজ কৰতো না। এখনে আমৱা যে ৩দিকে মুভ কৰছি তাতে কোনো সাইকেল তৈৰি হচ্ছেনা, অৰ্থাৎ কোনো স্টেট থেকে শুৰু কৰে সেই স্টেটে আবাৰ ফিৰে আসতে পারছিনা। সাইকেল যদি তৈৰি হতো তাহলে রিকাৰ্ণিভ ফাংশন আজীবন চলতেই থাকতো। তাই ডিপি সলিউশন লেখাৰ সময় অবশ্যই খেয়াল রাখতে হবে সাইকেল তৈৰি হচ্ছ নাকি।

1004 – Monkey Banana Problem এই প্রবলেমটা অনেকটা উপরের প্রবলেমের মতো, সলভ করতে কোনো সমস্যা হবেনা। তুমি যদি বিএফএস/ডিএফএস পারো তাহলে uva-11331 প্রবলেমটি সলভ করে ফেলো, (হিন্ট: বাইকালারিং+ন্যাপস্যাক)। এছাড়া এগুলো ট্রাই করো:

[Uva 11137: Ingenuous Cubrency\(Coin change\)](#)

[Codeforces 118D: Caesar's Legions\(4 state\)](#)

[Lightoj 1047: Neighbor House](#)

[Timus 1017: Staircases](#)

চেষ্টা করো সবগুলো প্রবলেম সলভ করতে, আটকে গেলে সমস্যা নাই, চিন্তা করতে থাকো, এছাড়া ন্যাপস্যাক আর কয়েন চেঙ্গ সম্পর্কিত যতগুলো পারো প্রবলেম সলভ করে ফেলো, যেহেতু তুমি এখন বেসিক পরবর্তী পর্বগুলোতে ডিটেইলস কমিয়ে নতুন নতুন প্রবলেম আর টেকনিক নিয়ে বেশি আলোচনা করবো।

হ্যাপি কোডিং!

[সবগুলো পর্ব](#)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-8

 shafaetsplanet.com/planetcoding/

শাফায়েত

8/25/2012

০-১ ন্যাপস্যাক কয়েন চেঞ্জ প্রবলেমে আশা করি তোমার এখন দক্ষতা এসে গিয়েছে। এই পর্বে আমরা দেখবো LIS, একই সাথে দেখবো কিভাবে ডিপিতে সলিউশন প্রিন্ট করতে হয়, এটা নিয়ে তোমাদের অনেকেরই সমস্যা হয়েছে বলে জানিয়েছো। এছাড়া আগের পর্বে light oj 1231 প্রবলেমটি সলভ করতে বলেছিলাম, আমার সলিউশন পাবে লেখার একদম শেষে।

প্রথমেই শুরু করি LIS এবং এটার সলিউশন প্রিন্ট করা দিয়ে। LIS হলো Longest increasing subsequence। মনে করো তোমাকে একটি অ্যারে বা sequence দেয়া আছে:

এই অ্যারে থেকে কিছু সংখ্যা মুছে দিয়ে এবং অর্ডার ঠিক রেখে আমরা বিভিন্ন subsequence পেতে পারি। যেমন:

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 0 | 9 | 2 | 7 | 3 | 4 |

Shafaetsplanet.com/blog

5 7  
5 9 2  
0 3 4  
0 2 3 4  
7 3  
5 0 9 2 7 3 4 (গুন্যটি সংখ্যা বাদ দেয়া হয়েছে)  
....

একটা sequence এ  $n$  টি সংখ্যা থাকলে subsequence থাকতে পারে মোট  $2^n$  টি(কেন??)। increasing subsequence (IS) এ প্রতিটা সংখ্যা তার আগের সংখ্যাটির থেকে বড় হবে। উপরে ১ম, ৩য়, ৪র্থ subsequence গুলো increasing। যতগুলো IS আছে তারমধ্যে সবথেকে বড়টা খুজে বের করাই আমাদের লক্ষ্য। উপরে ৪নশ্বরটির length 8 এবং সবগুলো IS এর মধ্যে এটাই সব থেকে বড়।

প্রবলেমটাকে গ্রাফ দিয়ে মডেলিং করলে খুব সহজে সলভ করা যায়। গ্রাফ থিওরি না জানলেও কোনো সমস্যা হবেনা, গ্রাফ দেখাচ্ছি খালি বোঝার সুবিধার জন্য। প্রথমেই মনে করি অ্যারের প্রতিটি ইনডেক্স হলো একটি করে নোড:

এখন দুটি শর্ত খেয়াল করো।  $u$  নশ্বর নোড থেকে  $v$  নশ্বরে নোডে যাওয়া যাবে যদি:



1.  $v > u$  হয় / কারণ আমাদের মূল sequence এর অর্ডার ঠিক রাখতে হবে।
2.  $\text{value}[v] > \text{value}[u]$  হয়, কারণ sequence টা increasing হতে হবে।

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 0 | 9 | 2 | 7 | 3 | 4 |

তাহলে ১ থেকে ২ এ যেতে পারবোনা কারণ  $\text{value}[1] < \text{value}[2]$ , তবে ১ থেকে ৩ এ যেতে পারবো। তাহলে আমরা গ্রাফের edge গুলো আস্কি:

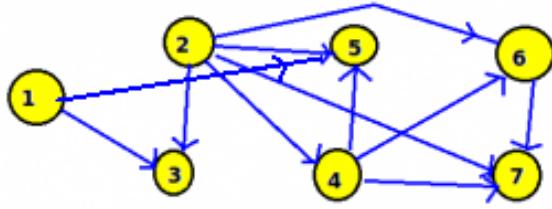
Shafaetsplanet.com/blog

দেখতে হিজিবিজি হলেও ভয় পাবার কিছু নাই, উপরের শর্ত দুটি মেনে গ্রাফটি আকা হয়েছে। গ্রাফের যেকোনো পাথ অনুসরণ করলে আমরা একটা ইনক্রিসিং সিকোয়েন্স পাবো। যেমন গ্রাফে ২-৬-৭ পাথে গেলে ০-৩-৪ সাবসিকোয়েন্সটা পাবো। তাহলে বুঝতেই পারছো সবথেকে লম্বা পথটাই আমার সলিউশন, যতদূরে যেতে পারবো সিকোয়েন্স তত বড় হবে, এক্ষেত্রে একটি path এ যে কয়টি নোড আছে সেটাই পাথের দৈর্ঘ্য।

মনে করি আমাদের একটা ফাংশন আছে  $\text{longest}(u)$  যেটা  $u$  নশ্বর নোড থেকে  $\text{longest path}$  রিটার্ন করে। যেমন  $\text{longest}(6)=2$  কারণ ৬ নশ্বর নোড থেকে খালি ৬-৭ এই পাথে যাওয়া যায়। এখন লক্ষ করো:

কোনো নোড থেকে longest path হবে সেই নোড  
থেকে যেসব নোডে যাওয়া যায় সেগুলো থেকে  
longest path এর maximum + 1 /

আমরা সহজেই একটা রিকার্সিভ রিলেশন বের করে ফেলতে  
পারি:



|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 0 | 9 | 2 | 7 | 3 | 4 |

Shafaetsplanet.com/blog

$$\text{longest}(2) = 1 + \max(\text{longest}(3), \text{longest}(4), \text{longest}(5), \text{longest}(6), \text{longest}(7))$$

$$\text{longest}(1) = 1 + \max(\text{longest}(3), \text{longest}(5))$$

$$\text{longest}(3) = 1 \text{ (এটা একটা বেসকেস কারণ } 3 \text{ থেকে কোথাও যাওয়া যায়না)}$$

.....

অর্থাৎ  $u$  থেকে  $v_1, v_2, v_3, \dots, v_k$  নোডে যাওয়া গেলে:

$$\text{longest}(u) = 1 + \max(\text{longest}(v_1), \text{longest}(v_2), \dots, \text{longest}(v_k))$$

এখন আমরা সহজেই কোড করে প্রতিটা নোড থেকে longest path বের করে ফেলতে পারি:

C++

```

1 #define mx 1000
2 int n=7;
3 int value[]={-100000,5,0,9,2,7,3,4};
4 int dp[mx],dir[mx];
5 int longest(int u)
6 {
7     if(dp[u]!=-1) return dp[u];
8     int maxi=0;
9     for(int v=u+1;v<=n;v++) //1ম শর্ত,v>u
10    {
11        if(value[v]>value[u]) //2য় শর্ত, value[v]>value[u]
12        {
13            if(longest(v)>maxi) //সর্বোচ্চ মানটা নিবো
14            {
15                maxi=longest(v);
16                dir[u]=v;
17            }
18        }
19    }
20 }
21 dp[u]=1+maxi; //1 ঘোগ হবে কারণ u নম্বর নোডটাও পাথের মধ্যে আছে
22 return dp[u];
23 }
24 int main()
25 {
26     READ("in");
27     memset(dp,-1,sizeof dp);
28     memset(dir,-1,sizeof dir);
29     int LIS=0,start;
30     for(int i=1;i<=n;i++)
31     {
32         printf("longest path from: %d\n",longest(i));
33         if(longest(i)>LIS)
34         {
35             LIS=longest(i);
36             start=i;
37         }
38     }
39     printf("LIS = %d Starting point %d\n",LIS,start);
40     return 0;
41 }
42

```

longest(u) প্রতিটা নোড u থেকে longest path রিটার্ন করে, এদের মধ্যে আবার যেটা ম্যাক্সিমাম সেটাই আমাদের LIS। এই সলিউশনটা কমপ্লেক্সিটি  $O(n^2)$ (আরো ভালো একটা সলিউশন আছে [বাইনারী সার্চ](#) ব্যবহার করে)। এখন সলিউশন প্রিন্ট করার পালা।

আসলে মুল কাজটা আমরা উপরের কোডেই করে ফেলসি। dir[] নামের একটি অ্যারে রেখেছি যে প্রতিটা নোডের জন্য যে ডিরেকশনে সর্বোচ্চ মান পাওয়া যায় সেটা সেভ করে রাখে। কোনো নোড থেকে অন্য কোথাও যাওয়া না গেলে ডিরেকশন হবে -1। এখন সলিউশন ফাংশন হবে এরকম:

C++

```

1 void solution(int start)
2 {
3     while(dir[start]!=-1)
4     {
5         printf("index %d value %d\n",start,value[start]);
6         start=dir[start];
7     }
8 }
```

অর্থাৎ ডিরেকশন অ্যারে দেখে আমরা সামনে যাবো যতক্ষণ যাওয়া যায়।

এখানে লক্ষ্য করার কিছু বিষয় হলো উপরের গ্রাফটিতে কোনো সাইকেল নেই দেখে আমরা ডিপি করতে পেরেছি, এধরণের গ্রাফকে বলা হয় **directed acyclic graph** বা ড্যাগ। কোনো প্রবলেমকে তুমি যদি ড্যাগ এ কনভার্ট করতে পারো তাহলে খুব ভালো সম্ভাবনা আছে যে প্রবলেমটি ডিপি চালিয়ে সলভ করা যাবে। গ্রাফে সাইকেল থাকলে longest path একটি np-complete প্রবলেম, অর্থাৎ পলিনমিয়াল সলিউশন জানা নেই। এখন একটি প্রশ্ন: একটা sequence এর কয়টি LIS আছে সেটা বের করতে বলা হলে কি করবে? চিন্তা করে জানাও :-)

আমরা জানি ০-১ন্যাপস্যাকে স্টেট থাকে ২টা, সেক্ষেত্রে ডিরেকশন অ্যারের স্টেটও হবে ২টা। নিচের কোডটা দেখো:

C++

```

1 int dir[][]={{-1}};
2 int dp[][]={{-1}};
3 int func(int i,int w) //i নম্বর আইটেম নিয়ে চেষ্টা করা হচ্ছে, w ওজনের জিনিস নেয়া হয়েছে
4 {
5     .....
6     //BASE CASE
7     .....
8
9     if(w+weight[i]<=CAP) //i নম্বর জিনিসটি নিবো
10        profit1=cost[i]+func(i+1,w+weight[i])
11    else
12        profit1=0;
13    profit2=func(i+1,w) // i নম্বর জিনিসটি নিবো না
14    if(profit1>profit2){dir[i][w]=1; return dp[i][w]=profit1;}
15    else {dir[i][w]=2; return dp[i][w]=profit2;}
16 }
```

ন্যাপস্যাকের কোডটাকে সামন্য পরিবর্তন করে ডিরেকশন রাখা হয়েছে। আমাদের starting state হলো ( $i=1, w=0$ ), কারণ শুরুতে আমরা ১ নম্বর জিনিসটা নিয়ে ট্রাই করবো এবং এখন পর্যন্ত নেয়া জিনিসের ওজন ০। ডিপি শেষ হবার পর যদি:

|   |
|---|
| $dir[i][w] = 1$ হয় তাহলে $i$ নম্বর জিনিসটি নিবো, $i$ প্রিন্ট করে আমরা চলে যাবো $(i+1, w+weight[i])$ স্টেটে।<br>$dir[i][w] = 2$ হয় তাহলে $i$ নম্বর জিনিসটি নিবো না, $i$ প্রিন্ট না করেই আমরা চলে যাবো $(i+1, w)$ স্টেটে।<br>$dir[i][w] = -1$ হলে আমরা থেমে যাবো। |
|---|

এটাই হলো ডিপির সলিউশন প্রিন্টের মূল কথা। ডিপি অ্যারের পাশাপাশি ডিরেকশন অ্যারেতে সেভ করে রাখবো কোন স্টেট থেকে কোন স্টেটে যাচ্ছি সেটা। এরপর starting state থেকে ডিরেকশন অনুযায়ী আগাতে থাকবো আর প্রিন্ট করবো।

এক স্টেট থেকে একাধিক স্টেট গেলে, আমরা সেভ করবো যে স্টেট গেলে  
ম্যাঞ্চিমাইজ বা মিনিমাইজ বা যে স্টেট থেকে ভ্যালিড আউটপুট পাওয়া যায় শুধু স্টেট

```
struct next
{
    int a,b;
    next(int _a,int _b){a=_a,b=_b;}
    next(){}
}direction[100][100];
int dp[100][100];
int call(int a,int b)
{
    .....
    .....
    int ret1=call(a+1,b);
    int ret2=call(a,b+1);
    if(ret1>ret2) {
        dp[a][b]=ret1;
        direction[a][b]=next(a+1,b);
    }
    else {
        dp[a][b]=ret2;
        direction[a][b]=next(a,b+1);
    }
    return dp[a][b];
}
```

ডিপি অ্যাবেতে বিটার্ণ ভ্যালু সেভ করছি,  
আর ডিবেকশন অ্যাবেতে কোন দিকে যাচ্ছি স্টেট সেভ করছি!

প্রিন্ট করার সময় আমরা ডিবেকশন অ্যাবের দেখানো  
পথে শুধু সামনে এগিয়ে যাবো!

```
void go(int a,int b)
{
    //Dont forget to return from function when base case reached
    //otherwise you'll fall into infinite loop
    int next_a=dir[a][b].a;
    int next_b=dir[a][b].b;
    if(next_a!=a) puts("Took a");
    else puts("Took b");
    go(next_a,next_b);
}
```

আজকের পর্ব এখানেই শেষ। light oj 1231 এর সলিউশন: <http://pastebin.com/vmpWp0g1>, mod করার অংশটা বুঝতে না  
পারলে [এই লেখাটা দেখো](#)। সলভ করার জন্য প্রবলেম:

[Diving for gold](#)  
[Sense of Beauty](#)  
[Bicolored Horses](#)  
[Testing the CATCHER](#)  
[How Many Dependencies?](#)  
[Longest Path](#)

[পরের পর্ব-বিটমাস্ক ডিপি](#)  
[সবগুলো পর্ব](#)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-৫

 shafaetsplanet.com/planetcoding/

শাফায়েত

12/18/2012

আশা করি তুমি এখন [lis,knapsack,coin-change](#) প্রবলেম সলভ করতে পারো খুব সহজেই, ডিপির সলিউশন প্রিন্ট করতেও তোমার সমস্যা হয়না। এখন আমরা একটু অন্যরকম ডিপি দেখবো যেটার নাম বিটমাস্ক ডিপি। নামটা শুনে ভয় লাগলেও জিনিসটা সহজ, অনেক ক্ষেত্রেই বিটমাস্ক ডিপি প্রবলেম পড়ার সাথে সাথে সলিউশন মাথায় চলে আসে। তবে এই পর্বটা পড়ার আগে তোমাকে বিট নিয়ে কাজ করা শিখতে হবে, যেমন কোনো নির্দিষ্ট পজিশনের বিট অন করা/অফ করা ইত্যাদি। এজন্য তুমি এই চমৎকার [টিউটোরিয়ালটা](#) দেখতে পারো, পুরোটা খুবই ভালো করে পড়বে কারণ এটা তোমাদের অনেক জায়গায় কাজে লাগবে। আমি এই টিউটোরিয়ালে বিট অপারেশন নিয়ে লিখছিনা কারণ অপ্রাসঙ্গিক হয়ে যাবে।

আমরা শুরুতেই ৩টি ফাংশন ডিফাইন করি।

C++

```
1 int Set(int N,int pos){return N=N | (1<<pos);}
2 int reset(int N,int pos){return N= N & ~(1<<pos);}
3 bool check(int N,int pos){return (bool)(N & (1<<pos));}
```

Set ফাংশনটি N সংখ্যাটির pos তম পজিশনের বিট ১ করে দেয়, reset ফাংশনটি ০ করে দেয় এবং check ফাংশনটি pos তম বিটে কি আছে সেটা রিটার্ন করে। যেকোনো বিটমাস্ক ডিপি প্রবলেমে ফাংশন ৩টি দরকার হবে, বিশেষ করে Set এবং check।

আমরা একটা প্রবলেম দিয়ে শুরু করি। মনে করো তোমাকে nটা দোকান থেকে n টা জিনিস কিনতে হবে। জিনিসগুলো কিনতে তোমার a<sub>0</sub>,a<sub>1</sub>,a<sub>2</sub>,...,a<sub>(n-1)</sub> টাকা লাগে। তোমার শহরটা খুব অদৃশুম্ভ তুমি যখন একটা জিনিস কিনে আরেক দোকানে যাও তখন সেই দোকানদার তোমার আগের কেনা জিনিসগুলো দেখে তার দোকানের জিনিসের দাম বাড়িয়ে দেয়!! কত দাম বাড়াবে সেটা নির্ভর করবে তুমি আগে আগে কোন কোন দোকানে গিয়েছো সেটার উপর। ধরো n=2, তাহলে তোমাকে নিচের মতো একটা ম্যাট্রিক্স দেয়া থাকবে:

|    |    |
|----|----|
| 10 | 10 |
| 90 | 10 |

এখন,

যদি  $(i==j)$  হয় তাহলে  $matrix[i][j]=matrix[i][i]=i$ ’ তম জিনিসটির আসল দাম।  
যদি  $(i!=j)$  হয় তাহলে  $matrix[i][j]=j$ ’ তম জিনিসটি আগে কিনলে i’ তম জিনিসটির সাথে যোগ হওয়া বাঢ়তি দাম।

তুমি যদি শুরুতে ০ তম জিনিসটা কিনো তাহলে দাম পড়বে ১০টাকা, এরপর ১নম্বর জিনিসটা কিনলে সেটার দাম হবে ১০+৯০ টাকা, কারণ  $matrix[1][0]=0$  নম্বর জিনিসের আগে ১ নম্বর জিনিস কিনলে যোগ হওয়া বাঢ়তি দাম=৯০ আর ১ নম্বর জিনিসের আসল দাম=১০, তাহলে মোট খরচ  $10+(10+90)=110$ । কিন্তু তুমি যদি ১নম্বর জিনিসটা আগে কিনো তাহলে মোট খরচ  $10+(10+10)=30$  টাকা।  
বুঝতেই পারছো তোমার কাজ হলো খরচ মিনিমাইজ করা। n এর মান সর্বোচ্চ ১৫।

n এর মান খুব কম বলে বিটমাস্ক ডিপি দিয়ে প্রবলেমটি সহজেই সলভ করা যাবে। ডিপিতে আমাদের প্রথম কাজ হলো সেট নির্ণয় করা। এই কেনাকাটার যেকোনো সময় আমাদের অবস্থা কি কি তথ্য দিয়ে প্রকাশ করা যায়? “এখন পর্বত্ত কোন কোন জিনিস কেনা হয়েছে” এই তথ্যটাই যথেষ্ট, তাইনা? এটা জানলে আমরা পরবর্তি আরেকটি জিনিস কেনার সময় বাড়তি কর

খরচ যোগ হবে জানতে পারবো, পরবর্তিতে যেই জিনিসটা কিনলে মোট খরচ কম হবে সেটা আমরা কিনবো। মনে করি এখন কথা হলো স্টেটটা রাখবো কি ভাবে?

একটা উপায় হলো  $n$  টি জিনিসের জন্য এভাবে  $n$ টা স্টেট রাখা function( $a_0, a_1, a_2, \dots, a_{n-1}$ ), কিন্তু  $n$  এর মান বদলালে তুমি প্যারামিটার সংখ্যা বদলাবে কি ভাবে? আর ১৫টি প্যারামিটার নিয়ে কাজ করলে কোডটা ভয়াবহ জটিল হয়ে যাবে।

হয় উপায় হলো বিটমাস্ক। একটি ইন্টিজারে ৩২টি বিট থাকে। আমরা সেই সুবিধাটাই নিবো। ১ নম্বর বিট ১ হলে আমরা ১ নম্বর জিনিসটা নিয়েছি, ০ হলে নেইনি। ৩ নম্বর বিট ১ হলে আমরা ৩ নম্বর জিনিসটা নিয়েছি, ০ হলে নেইনি। ১ এবং ৩ নম্বর বিট ১ হলে আমরা ২টা জিনিসই নিয়েছি।

শুরুতে আমাদের স্টেট থাকবে ০ বা বাইনারিতে “0000000”। তারমানে আমরা কোনো জিনিস এখনও কিনিনি। ০তম এবং ১তম জিনিস কেনা হয়ে গেলে স্টেট হবে ৩ বা “000011” এবং  $n=2$  এর জন্য এটাই আমাদের base case।  $n=4$  এর জন্য base case হলো ১৫ বা “0001111”। leading zero নিয়ে চিন্তা করা দরকার নেই, এটা বোঝার সুবিধার্থে দেয়া হয়েছে। একটু চিন্তা করলেই বুঝতে পারবে  $mask=(2^n)-1$  হলে সেটা হবে base case কারণ তখন বাইনারিতে প্রথম  $n$  টা বিট ১ থাকবে, আমরা তখন শুন্য রিটার্ন করে দিবো তখন কারণ আর কোনো জিনিস কেনা বাকি নেই।

C++

```

1 int dp[(1<<15)+2];
2 int call(int mask)
3 {
4     if(mask==(1<<n)-1) return 0;
5     if(dp[mask]!=-1) return dp[mask];
6         //Rest of the calculation
7 }
```

বেসকেস বুঝলাম, এরপরে আমাদের কাজ হবে যেসব জিনিস কেনা হয়নি সেগুলা নিয়ে চেষ্টা করে দেখা।  $mask$  এর ি তম পজিশনে যদি ০ থাকে তাহলে i তম জিনিসটি কেনা এখনও বাকি আছে।

C++

```

1 int dp[(1<<15)+2];
2 int call(int mask)
3 {
4     if(mask==(1<<n>-1)) return 0;
5     if(dp[mask]!=-1) return dp[mask];
6         //Rest of the calculation
7     int ans=1<<28; //Infinite, a large value
8     for(int i=0;i<n;i++)
9     {
10         if(check(mask,i)==0)
11         {
12             //Rest of the code
13         }
14     }
15     return dp[mask]=ans;
16 }
```

আমরা  $n$  পর্যন্ত লুপ চালিয়ে বের করে নিলাম কোনটা কোনটা নেয়া বাকি আছে। এখন i তম জিনিসটার আসল দাম হলো  $price=w[i][i]$ । এই  $price$  এর সাথে  $w[i][j]$  যোগ হবে যদি  $i=j$  হয় এবং j নম্বর জিনিসটা আগেই কেনা হয়ে থাকে।  $mask$  এর j তম বিট চেক করে আমরা বলতে পারি j তম জিনিসটা কেনা হয়েছে নাকি।

C++

```

1 int dp[(1<<14)+2];
2 int call(int mask)
3 {
4     if(mask==(1<<n)-1) return 0;
5     if(dp[mask]!=-1) return dp[mask];
6     int ans=1<<28;
7     for(int i=0;i<n;i++)
8     {
9         if(check(mask,i)==0)
10        {
11            int price=w[i][i];
12            for(int j=0;j<n;j++)
13                if(i!=j and check(mask,j)!=0) price+=w[i][j];
14            int ret=price+call(Set(mask,i));
15            ans=min(ans,ret);
16        }
17    }
18    return dp[mask]=ans;
19 }
20
21

```

j এর লুপটা দিয়ে আমরা মোট দাম বের করে নিলাম। এখন i তম জিনিসটি কিনলে পরবর্তি স্টেট কি হবে? শুধু mask এর i তম বিটটি 1 করে দিতে হবে। আমরা call(Set(mask,i)) এভাবে i তম জিনিস কিনে পরবর্তি স্টেটে চলে গেলাম। এভাবে প্রতিটি জিনিস কিনে যেটায় দাম মিনিমাম হয় সেটা রিটার্ন করে দিলাম। কাজ শেষ! সম্পূর্ণ কোড:

C++

```

1 int w[20][20];
2 int n;
3 int dp[(1<<15)+2];
4 int call(int mask)
5 {
6     if(mask==(1<<n)-1) return 0;
7     if(dp[mask]!=-1) return dp[mask];
8     int mn=1<<28;
9     for(int i=0;i<n;i++)
10    {
11        if(check(mask,i)==0)
12        {
13            int price=w[i][i];
14            for(int j=0;j<n;j++)
15            {
16                if(i!=j and check(mask,j)!=0)
17                {
18                    price+=w[i][j];
19                }
20            }
21            int ret=price+call(Set(mask,i));
22            mn=min(mn,ret);
23        }
24    }
25    return dp[mask]=mn;
26 }
27 int main()
28 {
29     mem(dp,-1);
30     cin>>n;
31     for(int i=0;i<n;i++)
32     {
33         for(int j=0;j<n;j++)
34         {
35             scanf("%d",&w[i][j]);
36         }
37     }
38     int ret=call(0);
39     printf("%d\n",ret);
40
41     return 0;
42 }
43
44
45
46

```

আমাদের call ফাংশনটি কয়টি ভিন্ন স্টেটে থাকতে পারে?  $n$  টি বিটের প্রতিটি হয় 0 হবে নাহয় 1 হবে, তাহলে স্টেট থাকতে পারে  $2^{15}$  টি। আর ভিতরে একটা  $n^2$  লুপ চলছে তাই মোট complexity  $(2^n)*(n^2)$ ।

বিটমাস্ক ডিপি চেনার সবথেকে সহজ উপায়  $n$  এর মান দেখা।  $n$  এর মান ১৬ বা তার কম হলে খুব ভালো সম্ভাবনা আছে যে প্রবলেমটিকে বিটমাস্ক ডিপি দিয়ে সলভ করা যাবে।

**বিটমাস্ক ডিপি কখন ব্যবহার করবো?** বিটমাস্ক লাগবে আমাদের তখনই যখন আগের স্টেটে কোন কোন জিনিস/ডিজিট/নোড ইত্যাদি ব্যবহার করা হয়েছে সে তথ্যটি আমার বর্তমান স্টেটে লাগবে। সেই তথ্য অনুযায়ী আমরা বর্তমান স্টেট থেকে

নতুন ডিজিট/নোড ইত্যাদি নিবো এবং সেই বিটটি অন করে দিয়ে সামনের স্টেটে যাবো। যখন  $n$  টি বিট অন হয়ে যাবে তখন বেসকেস রিটাৰ্ণ করে দিবো।

আমার সলভ করা প্রথম বিটমাস্ক ডিপি হলো [uva 10651](#)। আশা করি প্রবলেমটি এখন সহজেই করতে পারবে। প্রবলেমটায় একটি মাস্কের সাহায্যে কোনো সময় বোর্ডের কি অবস্থা সেই তথ্যটা রাখবে, এবং সে অবস্থায় যতগুলো চাল দেয়া সম্ভব সবগুলো দিয়ে মিনিমামটা রিটাৰ্ণ করে দিবে।

আপাতত এগুলোই ছিলো বিটমাস্ক ডিপির বেসিক। সামনের পর্যগুলোতে আরো বিস্তারিত আলোচনার চেষ্টা করবো। এখন নিচের প্রবলেমগুলো সলভ করার চেষ্টা করো, ১ম প্রবলেমটি নিয়ে এই পর্বে আলোচনা করেছি:

[Pimp My Ride](#)

[False Mirror](#)

[Agent 47](#)

[Painful Bases](#)

**সবগুলো পর্ব**

ব্লগের রেগুলার আপডেট পেতে [রেজিস্টার](#) করতে অনুরোধ করছি।

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাইনামিক প্রোগ্রামিং: লংগেস্ট কমন সাবসিকোয়েন্স

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

4/25/2014

ডাইনামিক প্রোগ্রামিং এর সম্ভবত সবথেকে গুরুত্বপূর্ণ দুটি উদাহরণ হলো লংগেস্ট কমন সাবসিকোয়েন্স এবং এডিট ডিসট্রেন্স বের করা কারণ এদের অনেক প্র্যাক্টিকাল অ্যাপ্লিকেশন আছে। এই লেখাটা পড়ার আগে আমি আশা করবো তোমরা কিছু বেসিক ডাইনামিক প্রোগ্রামিং যেমন কয়েন চেঞ্জ, ন্যাপস্যেক পারো। যদি না পারো তাহলে আমার [আগের লেখাগুলো](#) দেখতে পারো। এই লেখায় লংগেস্ট কমন সাবসিকোয়েন্স বের করা, সলিউশন প্রিন্ট করা এবং সবগুলো সম্ভাব্য সলিউশন বের করা দেখবো। তুমি যদি আগেই এসব টপিক নিয়ে জানো তাহলে লেখার শেষে রিলেটেড প্রবলেম সেকশন দেখো, সেখানকার শেষ ৩টা প্রবলেম সলভ করতে কিছুটা চিন্তা-ভাবনা করা লাগবে।

## সাবসিকোয়েন্স

মনে করো ১টি স্ট্রিং আছে “ABC”। এই স্ট্রিংটা থেকে শূন্য, এক বা একাধিক অক্ষর মুছে দিলে যা থাকে সেটাই স্ট্রিংটার সাবসিকোয়েন্স। “ABC” এর সাবসিকোয়েন্সগুলো হলো {"ABC", "A", "B", "C", "AB", "AC", "BC", ""}। সবগুলো অক্ষর মুছে দিলে যা থাকে, অর্থাৎ খালি বা এম্পটি(empty) স্ট্রিং ও একটা সাবসিকোয়েন্স।

প্রতিটা অক্ষরের জন্য আমাদের হাতে ২টি অপশন ছিলো, আমরা সেটা নিতে পারি বা মুছে ফেলতে পারি। তাহলে স্ট্রিং এর দৈর্ঘ্য  $n$  হলো সাবসিকোয়েন্স থাকতে পারে  $2^n$  টা।

## লংগেস্ট কমন সাবসিকোয়েন্স(LCS)

দুটি স্ট্রিং এর মধ্যে যতগুলো কমন সাবসিকোয়েন্স আছে তাদের মধ্যে সবথেকে লম্বাটাই লংগেস্ট কমন সাবসিকোয়েন্স(LCS)।

যেমন “HELLOM” এবং “HMLLD” এর মধ্যে “H”, “HL”, “HLL”, “HM” ইত্যাদি কমন সাবসিকোয়েন্স আছে। “HLL” হলো লংগেস্ট কমন সাবসিকোয়েন্স যা দৈর্ঘ্য ৩।

## ব্রুটফোর্স অ্যালগোরিদম:

আমরা ব্যাকট্র্যাকিং করে ২টা স্ট্রিং থেকে সবগুলো সাবসিকোয়েন্স জেনারেট করতে পারি। এরপরে ২টি করে সাবসিকোয়েন্স নিয়ে স্ট্রিং কম্পেয়ার করে দেখতে পারি তারা “কমন” কি না। আগেই দেখেছি মোট সাবসিকোয়েন্স হবে  $2^n$  টা, এরপরে আবার এদের মধ্যে কম্পেয়ার করতে হবে!  $n$  এর মান ২০-২৫ পার হলেই এই অ্যালগোরিদম শেষ হতে কয়েক বছর লেগে যেতে পারে! তাই আমাদের ভালো অ্যালগোরিদম দরকার।

## ডাইনামিক প্রোগ্রামিং:

ডাইনামিক প্রোগ্রামিং এর মূল কথা কি মনে আছে? প্রবলেমটাকে ছোট ছোট ভাগ করো, সেই ভাগ গুলো আগে সলভ করো আর সেগুলো জোড়া লাগিয়ে বড় প্রবলেমটার সমাধান বের করে ফেলো। কয়েন চেঞ্জের সময় আমরা প্রতিটা কয়েন নিয়ে বা ফেলে দিয়ে বাকি কয়েনগুলোর জন্য সলভ করেছি। এখানে প্রতিটা ক্যারেকটারের জন্য সেই কাজ করবো!

মনে করো “HELLOM” এবং “HMLLD” এদের মধ্যে LCS বের করতে চাও। শুরুতে তুমি আছো ২টা স্ট্রিং এরই প্রথম ক্যারেকটারে।

# HELLOM HMLLD

নীল রঙ দিয়ে বুঝাচ্ছে তুমি কোন স্ট্রিং এর কোন ক্যারেক্টারে আছো। নীল রঙের ক্যারেক্টার থেকে শুরু করে বাকি স্ট্রিংটুকুর জন্য তোমাকে প্রবলেমটা সলভ করতে হবে।

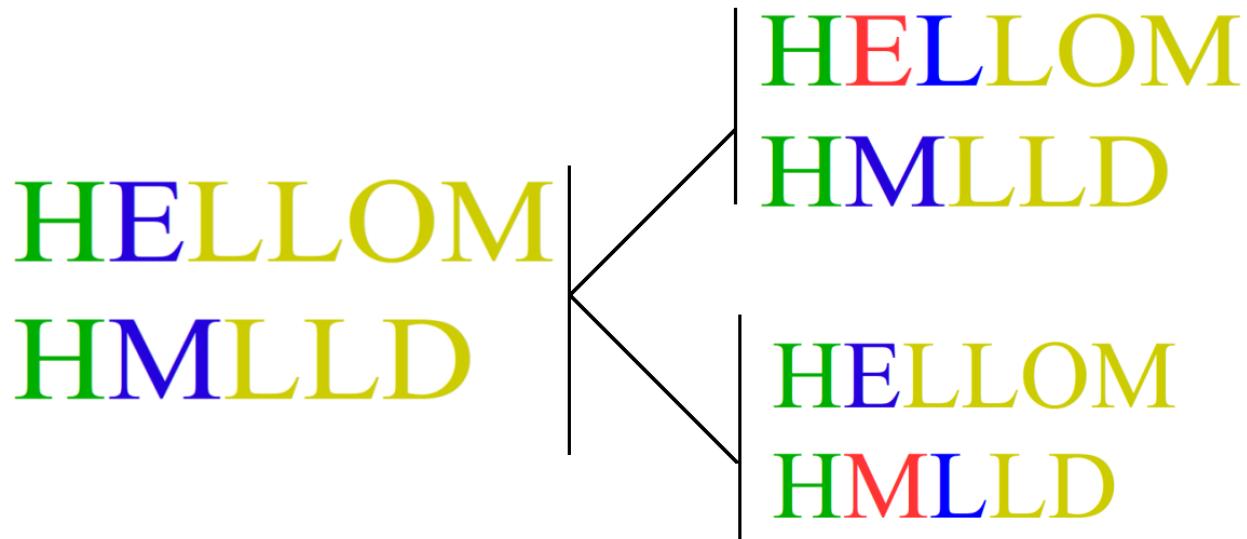
এখন লক্ষ্য করো নীল রঙের অক্ষর দুটি একই। তারমানে তুমি কিছু চিন্তা না করেই এই অক্ষরটাকে LCS এর মধ্যে ঢুকিয়ে দিতে পারো এবং বাকি স্ট্রিংটুকুর জন্য রিকার্সিভলি প্রবলেমটা সলভ করতে পারো।

[shafaetsplanet.com/blog](http://shafaetsplanet.com/blog)

# HELLOM HMLLD

আমরা H অক্ষরটাকে নিয়ে নিয়েছি। এখন ২টি স্ট্রিং এই আমরা ২য় অক্ষরে আছি। নীল অক্ষর থেকে বাকি স্ট্রিংটুকুর জন্য আমরা এখন সলভ করবো। তাহলে প্রবলেমটা ছোট হয়ে গেলো, আমাদের “ELLOM”, আর “MLLD” এর মধ্যের LCS বের করতে হবে।

এবার নীল রঙের অক্ষরদুটি একই না। তারমানে অন্তত ১টাকে বাদ দিয়ে LCS হিসাব করতে হবে। আমরা একবার উপর থেকে E বাদ দিয়ে হিসাব করবো, আরেকবার নিচ থেকে M বাদ দিয়ে হিসাব করবো।



তারমানে  $\text{LCS}(\text{"ELLOM"}, \text{"MLLD"})$  বের করতে আমরা  $\text{LCS}(\text{"LLOM"}, \text{"MLLD"})$  বের করবো এবং  $\text{LCS}(\text{"ELLOM"}, \text{"LLD"})$  বের করবো। এই দুইটার মাঝে যেটা বড় সেট নিবো!

সবগুলো সেটট ছবি একে দেখাবো না, তুমি নিশ্চয়ই বুঝে গেছো আমাদের কাজ কি হবে। মনে করো ফাংশন  $\text{calcLCS}(i, j)$  প্রথম স্ট্রিং এর  $i$  তম ক্যারেকটার এবং  $2j$  স্ট্রিং এর  $j$  তম ক্যারেকটার থেকে বাকি স্ট্রিং এর LCS বের করে। অর্থাৎ  $i, j$  হলো ১ম আর  $2j$  স্ট্রিং এর নীল রঙের ক্যারেকটার। আর স্ট্রিং দুটি হলো A আর B। তাহলে ফাংশনটা ডিফাইন করা যায় এভাবে:

$$\text{calcLCS}(i, j) = 1 + \text{calcLCS}(i+1, j+1) \text{ যদি } A[i] == B[j] \text{ হয়}$$

$$\text{calcLCS}(i, j) = \max(\text{calcLCS}(i+1, j), \text{calcLCS}(i, j+1)) \text{ যদি } A[i] != B[j] \text{ হয়}$$

রিকার্সনটা রানটাইম ইরোর দেয়ার আগে শেষ হবে না যদি না বেস-কেস দাও। কোন সেটটের জন্য আমরা হিসাব না করেই উত্তর বলে দিতে পারবো? প্রতি ধাপে স্ট্রিংগুলো ছোট হতে হতে কোন স্ট্রিং যদি “এম্পটি” হয়ে যায় তাহলে তার সাথে অন্য যেকোন স্ট্রিং এর LCS ০ হবে।

$$\text{calcLCS}(i, j) = 0 \text{ যদি } A[i] == \text{NULL} \text{ বা } B[j] == \text{NULL} \text{ হয়।}$$

একটু খেয়াল করলেই দেখবে একই ফাংশন বারবার কল হবে। তাই আমাদের মেমরাইজেশন করতে হবে যেটা আমরা অন্যসব ডিপি প্রবলেমে করি। পুরো কোডটা হতে পারে এরকম:

LCS

C++

```

1 #define MAXC 1000
2 char A[MAXC],B[MAXC];
3 int lenA,lenB;
4 int dp[MAXC][MAXC];
5 bool visited[MAXC][MAXC];
6 int calcLCS(int i,int j)
7 {
8     if(A[i]=='\0' or B[j]=='\0') return 0;
9     if(visited[i][j])return dp[i][j];
10
11    int ans=0;
12    if(A[i]==B[j]) ans=1+calcLCS(i+1,j+1);
13    else
14    {
15        int val1=calcLCS(i+1,j);
16        int val2=calcLCS(i,j+1);
17        ans=max(val1,val2);
18    }
19    visited[i][j]=1;
20    dp[i][j]=ans;
21    return dp[i][j];
22 }
23 int main() {
24     scanf("%s%s",A,B);
25     lenA=strlen(A);
26     lenB=strlen(B);
27     printf("%d\n",calcLCS(0,0));
28     return 0;
29 }

```

### সলিউশন প্রিন্ট:

এখন কথা হলো LCS কত বড় সেটাতো পেলাম, কিন্তু স্ট্রিংটা পাবো কিভাবে? এটাও খুব সহজ, calcLCS ফাংশনটা যে পথে এগিয়েছে সে পথে এগিয়ে গেলেই আমরা স্ট্রিংটা পেয়ে যাবো। আমরা আরেকটা ফাংশন লিখতে পারি, মনে করো ফাংশনটা হলো printLCS(i,j)। এখানেও i,j দিয়ে স্ট্রিং দুটির ইনডেক্স বুঝাচ্ছে। এখন  $A[i]==B[j]$  হলে আমরা অক্ষরটিকে প্রিন্ট করে  $(i+1,j+1)$  সেটে চলে যাবো। আর  $A[i]!=B[j]$  হলে আগেই হিসাব করা ডিপি অ্যারে থেকে  $dp[i+1][j]$  আর  $dp[i][j+1]$  এর মান দেখে সামনে আগাবো, যেদিকে গেলে ম্যাক্সিমাম দৈর্ঘ্য পাওয়া যাবে সেদিকে যাবো। নিচের কোডটা দেখলে পরিষ্কার হবে:

Printing LCS

C++

```

1 string ans;
2 void printLCS(int i,int j)
3 {
4     if(A[i]=='\0' or B[j]=='\0'){
5         cout<<ans<<endl;
6         return;
7     }
8     if(A[i]==B[j]){
9         ans+=A[i];
10        printLCS(i+1,j+1);
11    }
12    else
13    {
14        if(dp[i+1][j]>dp[i][j+1]) printLCS(i+1,j);
15        else printLCS(i,j+1);
16    }
17 }
```

অর্থাৎ আমাদের calcLCS ফাংশন যেদিকে গিয়ে সর্বোচ্চ মান পেয়েছে আমরা সে পথ ধরেই আগাচ্ছি।

এখন প্রশ্ন আসতে পারে যে সলিউশনতো একাধিক থাকতে পারে, সবগুলো পাবো কিভাবে? যেমন “hello” আর “loxhe” এই দুইটা স্ট্রিং এর LCS দুইটা, “he” এবং “lo”। সবগুলো সলিউশন চাইলে **ব্যাকট্র্যাকিং** করতে হবে। ধরো printAll(i,j) ফাংশনটা সবগুলো সলিউশন প্রিন্ট করে। আগের মতো যদি  $A[i]==B[j]$  তাহলে  $(i+1,j+1)$  এ যাবো। আর যদি  $A[i]!=B[j]$  হয় তাহলে  $dp[i+1][j]$  আর  $dp[j][i+1]$  এর যেটা বড় সেদিকে আগাবো, পার্থক্য হলো যদি  $dp[i+1][j]==dp[i][j+1]$  হয় তাহলে আমরা দুইদিকেই আগাবো।

## ALL LCS

C++

```

1 string ans;
2 void printAll(int i,int j)
3 {
4     if(A[i]=='\0' or B[j]=='\0'){
5         cout<<ans<<endl;
6         return;
7     }
8     if(A[i]==B[j]){
9         ans+=A[i];
10        printAll(i+1,j+1);
11        ans.erase(ans.end()-1); //Delete last character
12    }
13    else
14    {
15        if(dp[i+1][j]>dp[i][j+1]) printAll(i+1,j);
16        else if(dp[i+1][j]<dp[i][j+1]) printAll(i,j+1);
17        else
18        {
19            printAll(i+1,j);
20            printAll(i,j+1);
21        }
22    }
23 }
```

কোডটা অনেকটা আগের মতোই। “ans.erase(ans.end()-1);” এরকম একটা অতিরিক্ত লাইন দেখতে পাচ্ছো। এটার কাজ স্ট্রিং এর শেষ ক্যারেক্টারটা মুছে ফেলা। এটা কেন করছি? এই লাইনটা মুছে ফেললে সমস্যা কি হবে? এটা তুমি চিন্তা করে বের করো, লাইনটা মুছে চালিয়ে দেখো কি হয়।

### কমপ্লেক্সিটি:

স্ট্রিং দুটির দৈর্ঘ্য  $n$  এবং  $m$  হলে  $\text{calcLCS}()$  ফাংশনটি মোট  $n*m$  টা স্টেটে থাকতে পারে। তাহলে কমপ্লেক্সিটি  $O(n*m)$ ।

### প্র্যাকটিস প্রবলেম:

[Longest Common Subsequence](#)

[The Twin Towers](#)

[History Grading](#)

[Is Bigger Smarter?](#)

### রিলেটেড প্রবলেম ১: এডিট ডিসটেন্শন(লেভেল-১)

তোমাকে দুটি স্ট্রিং  $A, B$  দেয়া আছে। তুমি শুধুমাত্র  $A$  স্ট্রিংটার উপর ঢটা অপারেশন করতে পারো, কোনো একটা ক্যারেক্টার বদলে দিতে পারো, কোন ক্যারেক্টার মুছে ফেলতে পারো, যেকোন পজিশনে নতুন ক্যারেক্টার ঢুকাতে পারো। তারমানে চেপে, ডিলিট, ইনসার্ট হলো তোমার ৩টা অপারেশন। এখন তোমার কাজ মিনিমাম অপারেশনে  $A$  স্ট্রিংটাকে  $B$  বানানো। যেমন “blog” কে “bogs” বানাতে তুমি। মুছে ফেলে স্ট্রিং এর শেষে  $s$  ইনসার্ট করতে পারো।  
(হিন্টস: LCS এর মতোই দুইটা ইনডেক্স  $i, j$  কে স্টেট রাখতে হবে। এখন তুমি চিন্তা করো স্ট্রিং  $A$  থেকে কোন ক্যারেক্টার মুছে ফেললে  $i, j$  এর পরিবর্তন করকম হবে। ঠিক সেভাবে বাকি ২টি অপারেশনের জন্য কিভাবে  $i, j$  পরিবর্তন হবে সেটা বের করো)

[প্রবলেম লিংক](#)

### রিলেটেড প্রবলেম ২: লেক্সিকোগ্রাফিকালি মিনিমাম লংগেস্ট কমন সাবসিকোয়েন্স(লেভেল-২)

২টি স্ট্রিং এর যতগুলো LCS আছে তাদের মধ্য থেকে লেক্সিকোগ্রাফিকালি মিনিমাম টা বের করতে হবে। “hello” আর “loxhe” এর মধ্যে লেক্সিকোগ্রাফিকালি মিনিমাম LCS হলো “he”।

[প্রবলেম লিংক](#)

### রিলেটেড প্রবলেম ৩: লংগেস্ট কমন ইনক্রিপ্শন সাবসিকোয়েন্স(লেভেল- ৩)

এই প্রবলেমে শুধু এমন সব সাবসিকোয়েন্স বিবেচনা করতে হবে যারা ছোট থেকে বড় সাজানো। এদের মধ্যে লংগেস্ট কমন সাবসিকোয়েন্সটা বের করতে হবে। [প্রবলেম লিংক](#)

### রিলেটেড প্রবলেম ৪: ভিন্ন ভিন্ন লংগেস্ট কমন সাবসিকোয়েন্স(লেভেল- ৩)

২টি সিন্ট্রিং এর মধ্যে কয়টা ভিন্ন ভিন্ন LCS আছে বলতে হবে। সিন্ট্রিং এর দৈর্ঘ্য সর্বোচ্চ ১০০০ হতে পারে, ব্যাকট্র্যাকিং করে করা যাবে না। [প্রবলেম লিংক](#)

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# ডাইনামিক প্রোগ্রামিং: ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

জুন ১৮, ২০১৮

আমরা এবার আরো একটি ক্লাসিক ডাইনামিক প্রোগ্রামিং প্রবলেম দেখবো যেটার নাম ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন। এটা শেখা খুবই গুরুত্বপূর্ণ কারণ এটার ধারণা ব্যবহার করে অনেক ধরণের সমস্যা সমাধান করে ফেলা যায়। এই লেখাটা পড়ার আগে তোমার ডাইনামিক প্রোগ্রামিং এর ধারণা থাকতে হবে। এছাড়া ম্যাট্রিক্স নিয়েও ধারণা থাকতে হবে।

আমি নিশ্চিত তোমরা সবাই ম্যাট্রিক্স গুণের শর্তগুলো জানো, তাও আমি মনে করিয়ে দিতে চাই। ধরি আমাদের দুটি ম্যাট্রিক্স আছে  $A_1, A_2, A_1, A_2$  এবং তাদের ডিমেনশন  $m \times nm \times n$  আর  $p \times qp \times q$ । তাহলে কয়েকটি প্রোপার্টি আমাদের জন্য গুরুত্বপূর্ণ:

- ম্যাট্রিক্স দুটি গুণ করা যাবে তখনই যদি  $n=pn=p$  হয়। তারমানে প্রথম ম্যাট্রিক্সের কলাম সংখ্যা, দ্বিতীয় ম্যাট্রিক্সের রো এর সংখ্যার সমান হতে হবে।
- যদি আগের শর্ত পূরণ হয় তাহলে গুণ করার পর আমরা  $A_3, A_3$  ম্যাট্রিক্স পাবো যার ডিমেনশন  $m \times qm \times q$ ।
- ম্যাট্রিক্স গুণ করার সময় আমাদের কিছু সংখ্যাকে গুণ করে যোগ করতে হয় যেগুলোকে আমরা ক্ষেলার গুণ বলতে পারি। আমাদের মোট ক্ষেলার গুণ করা লাগবে মোট  $m \times n \times qm \times n \times q$  বা  $m \times p \times qm \times p \times q$  বার।

তুমি দুটি ম্যাট্রিক্স খাতায় লিখে গুণ করে যাচাই করে দেখতে পারো ব্যাপারগুলো। আরেকটা জিনিস মনে রাখবে যে  $A_1 \times A_2, A_1 \times A_2$  আর  $A_2 \times A_1, A_2 \times A_1$  এক না, অর্থাৎ অর্ডার ভঙ্গ করে আমরা ম্যাট্রিক্স গুণ করতে পারবো না।

এই লেখায় যখন সংখ্যাগুণের কথা বা শুধু “গুণ” এর কথা বলা হয়েছে তখন ম্যাট্রিক্সগুণ করার সময় ভিতরে যে সংখ্যাগুলো গুণ করতে হয় সেটা বুঝানো হয়েছে।

তাহলে যদি আমাদের তৃটা ম্যাট্রিক্স থাকে  $A_1, A_2, A_3$  যাদের ডিমেনশন  $m \times nm \times n, n \times pn \times p, p \times qp \times q$  তাহলে  $A_4 = A_1 \times A_2 \times A_3, A_4 = A_1 \times A_2 \times A_3$  ম্যাট্রিক্সের ডিমেনশন হবে  $m \times qm \times q$ । উপরের ২য় প্রোপার্টি থেকেই এই ব্যাপারটা বোঝা যাচ্ছে। এখন  $A_1 \times A_2 \times A_3$  এই ম্যাট্রিক্স গুণটা আমরা দুইভাবে করতে পারি, ব্রাকেট দিয়ে সেগুলো এভাবে দেখানো যায়:  $(A_1 \times A_2) \times A_3, A_1 \times (A_2 \times A_3), (A_1 \times A_2) \times A_3, A_1 \times (A_2 \times A_3)$ । অর্থাৎ আমরা  $A_1 \times A_2, A_1 \times A_2$  এর সাথে  $A_3, A_3$  কে গুণ করতে পারি, অথবা  $A_1, A_1$  কে  $A_2 \times A_3, A_2 \times A_3$  এর সাথে গুণ করতে পারি। তবে অর্ডার অবশ্যই ঠিক রাখতে হবে,  $(A_1 \times A_3) \times A_2, (A_1 \times A_3) \times A_2$  এটা ভ্যালিড নাও হতে পারে।

বুবতে পারছো অর্ডার ঠিক রেখে অনেকভাবে ব্রাকেট বসিয়ে গুণ করা যায়। কিন্তু কিভাবে ব্রাকেট বসাচ্ছি সেটা খুবই গুরুত্বপূর্ণ। ধরা যাক  $A_1, A_2, A_3, A_1, A_2, A_3$  এর ডিমেনশন  $10 \times 100, 100 \times 5, 5 \times 50$ ।

তাহলে

$(A_1 \times A_2) \times A_3, (A_1 \times A_2) \times A_3$  এই ব্রাকেটিং এ মোট সংখ্যা গুণ করতে হবে  $(10 \times 100 \times 5) + (10 \times 5 \times 50) = 7500$  বার  $A_1 \times (A_2 \times A_3), A_1 \times (A_2 \times A_3)$  এই ব্রাকেটিং এ মোট সংখ্যা গুণ করতে হবে  $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 75,000$  বার (এখানে মাল্টিপ্লিকেশন বলতে ম্যাট্রিক্স গুণ করার সময় কয়বার ভিতরে ক্ষেলার গুণ করা হচ্ছে সেটা বুঝানো হয়েছে, ৩নম্বর শর্ত দেখো)

২য় উপায়ে  $10$  গুণ বেড়ে গিয়েছে ক্যালকুলেশনের পরিমাণ! ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন পদ্ধতি ব্যবহার করে আমরা এমন একটা “ব্রাকেটিং” বের করবো যাতে সংখ্যা গুণের পরিমাণ সবথেকে কম হয়। তোমাকে  $A_1, A_2, \dots, A_n, A_1, A_2, \dots, A_n$  এরকম অনেকগুলো ম্যাট্রিক্সের শুধুমাত্র ডিমেনশন দেয়া থাকবে, বলতে হবে মিনিমাম কয়টা সংখ্যা গুণ করে  $A_1 \times A_2, \dots, A_n, A_1 \times A_2, \dots, A_n$  বের করা যায়। আমরা ধরে নিচ্ছি ডিমেনশনগুলো ভ্যালিড, অর্থাৎ প্রথম শর্ত পূরণ করে।

আমরা ডিভাইড এন্ড কনকোয়ার পদ্ধতিতে এটা সলভ করবো। ডাইনামিক প্রোগ্রামিং দরকার হবে কারণ একই সাবপ্রবলেম বারবার আসবে। ধরি  $n$  এর মান ৫, তাহলে  $A_1, A_2, A_3, A_4, A_5$  এই ৫টা ম্যাট্রিক্স আছে, তুমি  $A_1 * A_2 * A_3 * A_4 * A_5$  বের করতে কয়টা সংখ্যা গুণ লাগে সেটা বের করতে চাও। এখন দেখো আমরা বিভিন্ন ভাবে ম্যাট্রিক্সগুলোকে দুইভাবে ভাগ করে ফেলতে পারি, যেমন একটা উপায় হলো এরকম:

$$(A_1 * A_2) * (A_3 * A_4 * A_5) (A_1 * A_2) * (A_3 * A_4 * A_5)$$

তারমানে আমরা কোন একটা ব্রাকেটিং এ  $A_{\text{left}} = A_1 * A_2$   $A_{\text{left}} = A_1 * A_2$  বের করবো এবং কোন একটা ব্রাকেটিং এ  $A_{\text{right}} = A_3 * A_4 * A_5$   $A_{\text{right}} = A_3 * A_4 * A_5$  বের করবো। আমরা জানিনা কিভাবে সেগুলো বের করবো, তবে কেও যদি আমাদের বলে দেয়  $A_{\text{left}}$   $A_{\text{left}}$  আর  $A_{\text{right}}$   $A_{\text{right}}$  বের করতে কয়টা সংখ্যা গুণ করা লাগে তাহলে আমরা বলে দিতে পারবো মোট কয়টা সংখ্যা গুণ করা লাগে। মোট সংখ্যা গুণের সংখ্যা হবে:

$$\text{মোট গুণের সংখ্যা} = A_{\text{left}} * A_{\text{left}} \text{ নির্ণয় করতে গুণের সংখ্যা} + A_{\text{right}} * A_{\text{right}} \text{ নির্ণয় করতে গুণের সংখ্যা} + A_{\text{left}} * A_{\text{right}} * A_{\text{left}} * A_{\text{right}} \text{ নির্ণয় করতে গুণের সংখ্যা}$$

বা ৩য় শর্ত থেকে শেষ টার্মটাকে লিখতে পারি:  $A_{\text{left}} * A_{\text{left}}$  এর রো সংখ্যা  $\times$   $A_{\text{left}} * A_{\text{left}}$  এর কলাম সংখ্যা  $\times$   $A_{\text{right}} * A_{\text{right}}$  এর কলাম সংখ্যা

$$\text{মোট গুণের সংখ্যা} = A_{\text{left}} * A_{\text{left}} \text{ নির্ণয় করতে গুণের সংখ্যা} + A_{\text{right}} * A_{\text{right}} \text{ নির্ণয় করতে গুণের সংখ্যা} + A_{\text{left}} * A_{\text{left}} \text{ এর রো সংখ্যা} \times A_{\text{left}} * A_{\text{left}} \text{ এর কলাম সংখ্যা} \times A_{\text{right}} * A_{\text{right}} \text{ এর কলাম সংখ্যা}$$

তারমানে কেও যদি ম্যাজিকালি  $A_{\text{left}} * A_{\text{left}}$  এর  $A_{\text{right}} * A_{\text{right}}$  বের করতে দেয় তাহলেই আমরা মোট সংখ্যা বের করতে পারবো। কিন্তু আমরাতো আরো অনেকভাবে ভাগ করতে পারতাম, যেমন:

$$(A_1) * (A_2 * A_3 * A_4 * A_5) (A_1) * (A_2 * A_3 * A_4 * A_5)$$

$$(A_1 * A_2 * A_3) * (A_4 * A_5) (A_1 * A_2 * A_3) * (A_4 * A_5)$$

$$(A_1 * A_2 * A_3 * A_4) * (A_5) (A_1 * A_2 * A_3 * A_4) * (A_5)$$

আমরা প্রতিটা উপায়েই ভাগ করবো এবং ভাগ করার পর ম্যাজিকালি  $A_{\text{left}} * A_{\text{left}}$  এবং  $A_{\text{right}} * A_{\text{right}}$  নির্ণয় করতে কয়টা সংখ্যা গুণ করা লাগে সেটা বের করে ফেলে মোট সংখ্যা বের করে ফেলবো। যেভাবে ভাগ করলে মোট সংখ্যাটা মিনিমাম হয় সেটাই আমার উত্তর!

তুমি যদি রিকার্শন ভালো করে বুঝে থাকো তাহলে এতক্ষণে বুঝে গিয়েছো ম্যাজিকালি কিভাবে কাজটা করা হবে। বাম আর ডান পাশের ভাগগুলোকে আমরা রিকার্সিভলি সলভ করে ফেলবো একইভাবে, অর্থাৎ সেগুলোকে আবার অনেকভাবে ভাগ করে অপটিমাল উত্তরটা বের করে আনবো! তাহলে আমাদের অ্যালগোরিদম দাঢ়ালো খুব সহজ:

যত উপায়ে সম্ভব ভাগ করো

রিকার্সিভলি ছোট ভাগগুলোর জন্য সমাধান করো

বাম আর ডান পাশ মার্জ করে মোট গুণের সংখ্যা বের করো

সবগুলো উপায়ের মধ্যে সবথেকে ভালোটা নাও

তাহলে চিন্তা করো আমাদের রিকার্শনের প্যারামিটার বা স্টেট কি হবে? কি কি তথ্য আমাকে দিলে আমি প্রবলেমটা সলভ করতে পারি? আমার শুধু জানা দরকার ইনপুট ম্যাট্রিক্সগুলোর কোন অংশ নিয়ে আমি এখন কাজ করছি। তারমানে আমরা শুরুর পয়েন্ট আর শেষের পয়েন্ট স্টেট হিসাবে রাখবো।

int f(int beg, int end)

এরকম হবে ফাংশনের প্যারামিটার বা স্টেট।  $f$  ফাংশনটা  $\text{beg}$  থেকে  $\text{end}$  পর্যন্ত ম্যাট্রিক্সগুলোকে সবথেকে অপটিমালি গুণ করলে মোট কষট্টা সংখ্যা গুণ করা লাগে সেটা বলে দিবো! রিকার্শন থামবে কখন? অর্থাৎ বেস কেসটা কি? যখন দেখবো একটা বা তারথেকে কম ম্যাট্রিক্স আছে ( $b > e$ ) ( $b > e$ ) তখন আমরা জানি একটা গুণও করা লাগবেনা, শূন্য রিটার্ন করে দিবো।

তাহলে আমরা এখন একটা কোড লিখে ফেলি। সবথেকে ভালো হয় যদি তারআগে তুমি নিজেই একবার চেষ্টা করো, সাহায্য না নিয়ে সলভ করতে পারলে যে আনন্দ পাওয়া যায় তার তুলনা নেই! আর না পারলেও মন খারাপের কিছু নেই। ইনপুট হিসাবে তুমি প্রতিটি ম্যাট্রিক্সের রো এবং কলাম সংখ্যা নিবে। আউটপুট হবে অপারেশন সংখ্যা।

স্টেট পাওয়ার পরে আমাদের কাজ হবে এক স্টেট থেকে অন্য স্টেটে কিভাবে যাবো, অর্থাৎ রিকারেন্সি রিলেশনটা বের করা। এই প্রবলেমে আমরা একটা করে মিডপয়েন্ট সিলেক্ট করে বাম আর ডানের পাশের জন্য প্রবলেমটা রিকার্সিভলি সলভ করবো এবং তাদের মার্জ করবো। রিকারেন্সিটা তাহলে হবে এরকম:

$$f(\text{beg}, \text{end}) = \begin{cases} 0, & \text{if } \text{beg} \geq \text{end}. \\ \min_{\text{beg} \leq \text{mid} \leq \text{end}} f(\text{beg}, \text{mid}) + f(\text{mid} + 1, \text{end}) + \text{row}[\text{beg}] * \text{col}[\text{mid}] * \text{col}[\text{end}] & \text{otherwise.} \end{cases}$$

এবার আমরা এটাকে কোডে রূপান্তর করে ফেলি:

C++

```

1 #define MAX 100
2 int row[MAX], col[MAX];
3 int dp[MAX][MAX];
4 bool visited[MAX][MAX];
5 int f(int beg,int end)
6 {
7     if(beg>=end) return 0;
8     if(visited[beg][end]) return dp[beg][end];
9     int ans=1<<30; //2^30 কে ইনফিনিটি ধরছি
10    for(int mid=beg; mid<end;mid++) //দুইভাগে ভাগ করছি
11    {
12        int opr_left = f(beg, mid); //opr = multiplication operation
13        int opr_right = f(mid+1, end);
14        int opr_to_multiply_left_and_right = row[beg]*col[mid]*col[end];
15        int total = opr_left + opr_right + opr_to_multiply_left_and_right;
16        ans = min(ans, total);
17    }
18    visited[beg][end] = 1;
19    dp[beg][end] = ans;
20    return dp[beg][end];
21 }
22
23 int main()
24 {
25     int n;
26     cin>>n;
27     rep(i,n)cin>>row[i]>>col[i];
28     cout<<f(0,n-1)<<endl;
29 }
```

খুবই সহজ একটা কোড, দুইভাগে ভাগ করছি আর ছোট ভাগটা সলভ করছি। একই অংশের জন্য বারবার সলভ করতে চাইনা তাই ডিপি অ্যারেতে সেভ করে রাখছি, যদি দেখি কোন একটা স্টেট আগে ভিজিট করা হয়েছে তখন পুরোনো রেজাল্ট রিটার্ন করে দিচ্ছি!

## কমপ্লেক্সিটি:

beg আর end এর মান হতে পারে 11 থেকে n পর্যন্ত। তাহলে ভিন্ন স্টেট আছে প্রায়  $n^2n^2$  টা। প্রতিটা স্টেটে আবার nn পর্যন্ত লুপ চলতে পারে। তাহলে টাইম কমপ্লেক্সিটি  $O(n^3)$ । মেমরি লাগবে  $O(n^2)O(n^2)$ ।

## রিলেটেড প্রবলেম:

সরাসরি ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশনের প্রবলেম হয়তো তুমি কনটেস্টে পাবেনা তবে এভাবে দুই প্রান্তকে স্টেট ধরে বিভিন্নভাবে ভাগ করার আইডিয়া দিয়ে অনেক প্রবলেম সলভ করতে পারবে যে কারণে এটা শেখা এত গুরুত্বপূর্ণ

১. <http://www.spoj.com/problems/MIXTURES/>

২. [Cutting Sticks](#)

[ধন্যবাদ শান্তি ভাই এবং তানভীর ভাইকে অনেকগুলো ভুল শুধরে দেয়ার জন্য]

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# HANDBOOK OF ALGORITHMS

## Section Game Theory

*Courtesy of*  
*Shafaet Ashraf*

অ্যালগোরিদিম গমে থওরি-১ \_ শাফায়তেরে ব্লগ.pdf

অ্যালগোরিদিম গমে থওরি-২ (নমি গমে) \_ শাফায়তেরে ব্লগ.pdf

অ্যালগোরিদিম গমে থওরি-৩ (স্প্রিংবাক-গ্রান্ডসিংখ্যা) \_ শাফায়তেরে ব্লগ.pdf

# অ্যালগোরিদম গেম থিওরি – ১

 [shafaetsplanet.com/planetcoding/](http://www.shafaetsplanet.com/planetcoding/)

শাফায়েত

অক্টোবর ২২, ২০১৫

আমরা বাস্তবে যে সব খেলাধুলা করি সেগুলোতে আমরা খেলার শুরুতেই বলে দিতে পারি না কে খেলাতে জিতবে, আমরা এটাও ধরে নিতে পারি না যে সব খেলোয়াড়ই সবসময় সেরা চাল দিবে। এছাড়া অনেক খেলায় ভাগ্যেরও সহায়তা দরকার হয়। যেমন তাস খেলায় আমরা জানি না প্রতিপক্ষের কাছে কি কি কার্ড আছে, বা লুডু খেলায় আমরা জানি না ছক্কা বা ডাইস এ কখন কোন সংখ্যাটা আসবে। এই সিরিজে সময় আমরা মূলত মাত্র এমন সব গেম নিয়ে কাজ করবো যার নিচের বৈশিষ্ট্যগুলো আছে:

১. গেমের বোর্ড, চাল ইত্যাদি সম্পর্কে পূর্ণাঙ্গ তথ্য আমাদের কাছে আছে, প্রতিপক্ষ কি অবস্থায় আছে সেটাও আমরা জানি।
২. খেলায় কোনো ভাগ্যের সহায়তা দরকার হয় না।
৩. খেলা শেষে কেও একজন জিতবে বা হারবে।
৪. খেলায় দুইজন মাত্র খেলোয়াড় থাকবে, প্রথম ও দ্বিতীয় খেলোয়াড় পালাবদল করে চাল দিতে থাকবে।
৫. দুইজন খেলোয়াড়ই খেলার যেকোনো সময় সবথেকে সেরা চালটি দিবে, কেও কোনো ভুল চাল দিবে না। কোনো খেলোয়াড়ের জয়ের সম্ভাবনা থাকলে সে অবশ্যই জিতবে, সম্ভাবনা না থাকলে প্রতিপক্ষের ভুলে সে কখনো জিততে পারবে না। এককথায়, প্রতিটা খেলোয়াড় সবসময় ‘অপটিমাল’ চালটি দিবে।

আমরা যে ধরণের গেমগুলো দেখবো সেগুলো উপরের শর্তগুলো মেনে চলে, অ্যালগোরিদম গেম আরো অনেক রকম হতে পারে। এই লেখাটা পড়ার আগে [ডাইনামিক প্রোগ্রামিং](#) এর বেসিক নিয়ে জানা থাকলে বুঝাতে সুবিধা হবে।

এই ধরনের খেলায় একদম শুরুতে বা খেলার মাঝে খেলার নিয়মগুলো দেখে এবং দুই খেলোয়াড়ের বর্তমান অবস্থা দেখেই বলে দেয়া যায় যে দুইজনই অপটিমাল চাল দিলে কে খেলায় জিতবে। বাস্তবে এ ধরণের খেলা বোরিং মনে হলেও গেম থিওরি অনেক ব্যবহার হয়, তবে এই টিউটোরিয়ালে আমরা গেম থিওরির একটা ছোটো সাবসেট ‘অ্যালগোরিদম গেম’ নিয়ে জানবো।

ছোটোবেলায় টিভিতে ম্যাগাজিন অনুষ্ঠানে একটা খেলা দেখেছিলাম, টেবিলে কিছু মার্বেল রাখা থাকে, একজন দর্শককে স্টেজে নিয়ে আসা হয়। দর্শকের সাথে উপস্থাপক একটা খেলা খেলে এরকম, প্রথম চালে দর্শক বা ১ম খেলোয়াড় ২টা, ৩টা বা ৫টা মার্বেল তুলে নিতে পারবে, পরের চালে উপস্থাপক বা ২য় খেলোয়াড় ২, ৩ বা ৫টা মার্বেল তুলে নিতে পারবে, এরপর দর্শক আবার একই ভাবে মার্বেল তুলবে। এভাবে চলতে থাকবে ঘতক্ষণ না কারো পক্ষে আর চাল দেয়া সম্ভব না হয়। শেষবার যে মার্বেলগুলো তুলেছে সে খেলাটা জিতে যাবে।

অনুষ্ঠানে দেখা যেত একের পর এক দর্শক আসছে আর হেরে যাচ্ছে, কোনোভাবেই উপস্থাপককে হারানো সম্ভব হচ্ছে না। আমরা এখন খেলাটাকে বিশ্লেষণ করলেই বুঝে যাবো কেন তাকে হারানো সম্ভব হয় নি সবথেকে বুদ্ধিমান দর্শকের পক্ষেও।

এই খেলায় খেলোয়াররা কোনো সময় কি অবস্থানে আছে সেটা জানার জন্য আমাদের দুইটা তথ্য দরকার, বর্তমানে কয়টা মার্বেল আছে, এবং কার চাল দেয়ার পালা। যদি ৫টা মার্বেল থাকে তাহলে এখন যে খেলোয়াড়ের চাল দেয়ার পালা সে একটা ‘উইনিং পজিশন’ বা ‘বিজয়ী অবস্থা’ তে চলে গিয়েছে, সে কোনো ভুল না করলে তাকে আর হারানো সম্ভব না, ৫টা মার্বেল তুলে নিয়ে সে খেলাটা শেষ করে দিবে। কিন্তু টেবিলে যদি ১টা বা ০টা মার্বেল থাকে তাহলে যে খেলোয়াড়ের চাল দেবার পালা সে ‘লুজিং পজিশন’ বা ‘পরাজিত অবস্থা’ য় চলে গিয়েছে। আমরা এখন দেখাবো যে টেবিলে মার্বেলের সংখ্যা ঘটই হোক না কেন, প্রথম খেলোয়ার অর্থাৎ যে এখন চাল দিবে পরাজিত অবস্থা না বিজয়ী অবস্থায় আছে আগেই জেনে যাওয়া সম্ভব। আমরা ধরে নিচ্ছি কোনো খেলোয়ারই ভুল চাল দিবে না।

মনে করি টেবিলে মার্বেল আছে  $n$  টা।

৩টা জিনিস আমাদের মাথায় রাখতে হবে:

১. যেসব অবস্থা থেকে আর কোনো চাল দেয়া সম্ভব না সেগুলো পরাজিত অবস্থা বা লুজিং পজিশন। যেমন এই খেলায়  $n=1=n=1$  এবং  $n=0=n=0$  হলো লুজিং পজিশন। এগুলোকে টার্মিনাল পজিশন বলা হয় কারণ এই অবস্থা থেকে আর কোনো চাল দেয়া সম্ভব না।
২. যদি আমরা টার্মিনাল পজিশনে না থাকি তাহলে দেখতে হবে কোনো চাল দিয়ে প্রতিপক্ষকে একটা লুজিং পজিশনে ফেলে দেয়া যায় নাকি। যদি যায় তাহলে বর্তমান খেলোয়ার উইনিং পজিশন এ আছে। যেমন  $n=5=n=5$  হলে আমি চাইলে ২, ৩ বা ৫টি মার্বেল তুলে নিয়ে টেবিলে  $5-2=3$  টি অথবা  $5-3=2$  টি অথবা  $5-5=0$  টি মার্বেল রেখে দিতে পারি। আমরা আগেই জানি

$n=0$  একটা লুজিং পজিশন যেখান থেকে কোনো খেলোয়াড়ের পক্ষে জেতা সম্ভব না। যেহেতু টেবিলে ৫টা মার্বেল থাকলে প্রতিপক্ষকে লুজিং পজিশনে নিয়ে যাওয়া সম্ভব হচ্ছে,  $n=5$   $n=5$  একটা উইনিং পজিশন, বর্তমান খেলোয়াড় কোনো ভুল না করলে তাকে এই অবস্থা থেকে কিছুতেই হারানো সম্ভব না।

৩. বর্তমান অবস্থা থেকে কোনো চাল দিয়ে প্রতিপক্ষকে একটা লুজিং পজিশনে নিয়ে যাওয়া না গেলে বর্তমান খেলোয়ার লুজিং পজিশনে আছে, প্রতিপক্ষ ভুল না করলে তার পক্ষে আর কিছুতেই জেতা সম্ভব না।

এখন আমরা সহজেই যে কোনো  $n$  এর জন্য বর্তমান খেলোয়াড় কি অবস্থায় আছে বের করে ফেলতে পারি নিচের কোড দিয়ে:

Python

```

1 def can_win(n):
2     if n==0 or n==1: return 0
3     if(can_win(n-2)==0 or can_win(n-3)==0 or can_win(n-5)==0):
4         return 1
5     return 0

```

আমরা প্রতিটা  $n$  এর জন্য নিচের মত আউটপুট পাবো:

| n          | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| can_win(n) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1  | 1  | 1  | 1  | 0  | 0  |

০ মানে হলো লুজিং পজিশন, ১ মানে উইনিং। এখান থেকে স্পষ্ট যে  $n=15$   $n=15$  এর জন্য বর্তমান খেলোয়াড় কখনো জিততে পারবে না। ১৫টা মার্বেল থেকে ২, ৩ বা ৫টা মার্বেল তুলে আমরা ১৩, ১২ বা ১০টা মার্বেল রেখে দিতে পারি, কিন্তু টেবিল থেকে দেখা যাচ্ছে সবগুলোই উইনিং পজিশন, তাই বর্তমান অবস্থা থেকে জেতা সম্ভব না। টিভির সেই অনুষ্ঠানে যদি উপস্থাপক ১৫টি মার্বেল নিয়ে দর্শককে প্রথম খেলোয়াড় বানিয়ে খেলা শুরু করে এবং সে যদি বুদ্ধিমান হয় তাহলে তাকে হারানো অসম্ভব!

আমি নিশ্চিত এখন তোমরা বুঝে গিয়েছো কিভাবে এধরণের সমস্যা সমাধান করতে হয়। চিন্তা করার মতো কিছু সমস্যা দিলাম, সবগুলোই উপরের সমস্যাটার মতো:

১. এলিস আর বব একটা খেলা খেলছে, তাদের কাছে একটা সংখ্যা আছে  $p=1$   $p=1$ । প্রতি চালে একজন খেলোয়ার  $pp$  এর সাথে ২ থেকে ৯ এর ভিতর একটা সংখ্যা গুণ করতে পারে। গুণ করার পর সংখ্যাটা  $xx$  এর থেকে বড় হয়ে গেলে সে জিতে যাবে।  $xx$  এর মান দেয়া আছে, এলিস সবসময় প্রথম চাল দেয়। খেলায় কে জিতবে? ([UVA 847](#))

২. [UVA 11489](#)

৩. আবেকটু কঠিন সমস্যা সমাধান করতে চাইলে [UVA 10891](#)

পরের পর্বে অ্যালগোরিদম গেম এর সবথেকে গুরুত্বপূর্ণ বিষয় “নিম গেম” নিয়ে আলোচনা করবো।

রেফারেন্স:

<https://www.topcoder.com/community/data-science/data-science-tutorials/algorithm-games/>

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# অ্যালগোরিদম গেম থিওরি-২ (নিম গেম)

 shafaetsplanet.com/planetcoding/

শাফায়েত

জানুয়ারি ৩, ২০১৬

আগের পর্বে গেম থিওরি কিছু বেসিক শিখেছি, এবারের পর্বে আমরা জানবো নিম-গেম নিয়ে। নিম-গেম খুবই গুরুত্বপূর্ণ কারণ অনেক ধরণের গেমকে নিম গেম এ রূপান্তর করে ফেলা যায়।

নিম-গেম এ দুইজন খেলোয়ার আর কিছু পাথরের স্তুপ(pile) থাকে। প্রতি চালে একজন খেলোয়াড় যেকোনো একটা স্তুপ থেকে এক বা একাধিক পাথর তুলে নিতে পারে। কেও চাল দিতে ব্যর্থ হলে হেরে যাবে। অর্থাৎ শেষ পাথরটা যে তুলে নিয়েছে সে গেমে জিতবে।

উপরের ছবিতে ৩টা পাথরের স্তুপ দেখা যাচ্ছে, প্রথমটায় ৬টা, দ্বিতীয়টায় ৯টা এবং তৃতীয়টায় ৩টা পাথর আছে।

আগের গেমগুলোর মতো এখানেও প্রত্যেক খেলোয়াড় অপটিমাল পদ্ধতিতে খেলবে, কেও কোনো ভুল চাল দিবে না। তোমাকে কোন স্তুপে কয়টি পাথর আছে সেটা দেখে বলতে হবে প্রথম খেলোয়ার জিতবে নাকি হারবে।

এখন মনে করো  $n$  টা স্তুপে যথাক্রমে  $a_1, a_2, \dots, a_n$  টা পাথর আছে। খেলায় প্রথম খেলোয়াড় হারবে শুধুমাত্র যদি  $xorsum = a_1 \oplus a_2 \oplus \dots \oplus a_n = 0$  হয়। এখানে  $\oplus$  দিয়ে xor অপারেটর বুঝানো হচ্ছে।

উপরের ছবির উদাহরণে  $xorsum = 6 \oplus 9 \oplus 3 = 12 \neq 0$  হলে  $xorsum = 6 \oplus 9 \oplus 3 = 12 \neq 0$ । এখানে xorsum শুন্য না, তাই অপটিমাল পদ্ধতিতে খেললে প্রথম খেলোয়াড়কে হারানো সম্ভব না।

এখন প্রশ্ন হলো এটা কেন আর কিভাবে কাজ করছে?  $xorsum = 0$  যদি লুজিং স্টেট হয়ে থাকে তাহলে বর্তমান খেলোয়াড় যেভাবেই চাল দিক না কেন সে গেমটাকে একটা উইনিং স্টেট এ নিয়ে যাবে।

মনে করো  $n=4$ ,  $n=4$  এবং স্তুপ গুলোতে পাথরের সংখ্যার সেট  $\{9, 7, 11, 5\}$ । এদেরকে বাইনারিতে নিচের মত করে লিখি:

1 0 0 1 (= 9)  
0 1 1 1 (= 7)  
1 0 1 1 (= 11)  
0 1 0 1 (= 5)

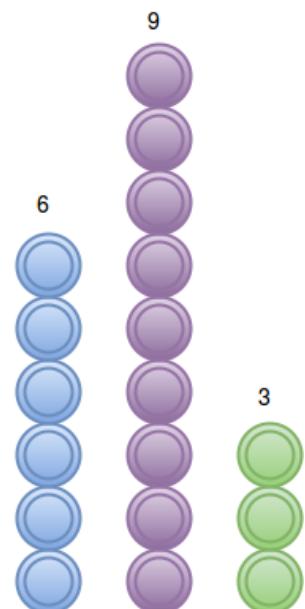
আমরা বাইনারি সংখ্যাগুলোকে একটা  $4 \times 4 \times 4$  গ্রিড হিসাবে চিন্তা করতে পারি। লক্ষ্য করো প্রতিটা কলামেই জোড় সংখ্যক 11 আছে, তাই  $xorsum = 0$  হবে, এটা একটা লুজিং স্টেট। এখন বর্তমান খেলোয়াড় একটা স্তুপ বেছে নিয়ে কিছু পাথর উঠিয়ে নিলো। সে এই কাজটা যেভাবেই করুক, কোনো একটা সারির অন্তত ১টি 11 কে 00 তে পরিবর্তন করতে হবে। ফলে অন্তত একটা কলামে বিজোড় সংখ্যক 11 থেকে যাবে এবং  $xorsum > 0$  হয়ে যাবে।

তারমানে বর্তমান স্টেট এ  $xorsum = 0$  হলে তুমি যেভাবেই চাল দাও না কেনো  $xorsum > 0$  হয়ে যাবে।

এখন বর্তমান স্টেট এ যদি  $xorsum > 0$  হয় তাহলে দেখানো যায় যে বর্তমান খেলোয়াড়ের পক্ষে এমন চাল দেয়া সম্ভব যাতে গেমটা লুজিং স্টেট এ চলে যায়, অর্থাৎ  $xorsum = 0$  হয়ে যায়।

মনে করো পাথরের সংখ্যার সেট  $\{5, 14, 9, 5\}$ ।

0 1 0 1 = (5)  
1 1 1 0 = (14)  
1 0 0 1 = (9)  
0 1 0 1 = (5)



বাম থেকে ২য় এবং ৩য় কলামে বিজোড় সংখ্যক 11 আছে, তাই xorsum>0xorsum>0 হবে। এখন আমি এমন চাল দিতে চাই যেনো xorsum=0 হয়ে যায়। এজন্য প্রথমেই সবথেকে বামের কলামটা খুজে বের করবো যেটাই বিজোড় সংখ্যা 1 আছে, এক্ষেত্রে সেটা ২য় কলাম। এবার ২য় কলামে 11 আছে এমন একটা রো বেছে নিবো, এক্ষেত্রে সেটা হতে পারে প্রথম, দ্বিতীয় বা চতুর্থ রো। এবার সেই রো এর ২য় কলামের 11 টাকে 00 বানিয়ে দাও এবং অন্যান্য কলামের 00 বা 11 কে এমন ভাবে পরিবর্তন করো যেন প্রতিটা কলামে জোড় সংখ্যক পাথর থাকে। তাহলেই xorsum=0xorsum=0 হয়ে গেলো!

তারমানে বর্তমান স্টেট এ xorsum>0xorsum>0 হলে বর্তমান খেলোয়াড় সহজেই এমন চাল দিতে পারবে যেনো xorsum=0xorsum=0 হয়ে যায়।

তাহলে দেখা যাচ্ছে লুজিং স্টেট (xorsum=0xorsum=0) থেকে যেভাবেই চাল দেয়া হোক না কেনো শুধুমাত্র উইনিং স্টেট (xorsum>0xorsum>0) এ যাওয়া যায়, আবার বুদ্ধিমানের মত খেললে উইনিং স্টেট (xorsum>0xorsum>0) থেকে সবসময় লুজিং স্টেট এ যাওয়া যায় (xorsum=0xorsum=0)। তাই শুরুতে xorsum>0xorsum>0 হলে প্রথম খেলোয়াড়কে কখনোই হারানো সম্ভব না সে যদি অপটিমাল পদ্ধতিতে খেলে।

এখন আমরা নিম গেম এর কিছু ভ্যারিয়েশন দেখি।

### মিজেরা(Misere) নিম

**Misere** একটা ফ্রেঞ্চ শব্দ। মিজেরা নিমগেম এ যে খেলোয়ার শেষ পাথরটা তুলে নিবে সে হেরে যাবে। মিজেরা নিম এও xorsum>0xorsum>0 উইনিং পজিশন। তবে প্রথম খেলোয়াড়কে স্ট্রাটেজি কিছুটা পরিবর্তন করতে হবে। শেষ চালে সবগুলো পাথর তুলে না নিয়ে একটা মাত্র পাথর রেখে দিতে হবে, দ্বিতীয় খেলোয়াড় তখন শেষ পাথরটা তুলতে বাধ্য হবে। তবে যদি প্রতিটা স্ট্রেচ ঠিক ১টা করে পাথর থাকে তখন আর xorsum দিয়ে কাজ হবে না। তখন দেখতে হবে স্ট্রেচের সংখ্যা জোড় নাকি বেজোড়। যদি বেজোড় সংখ্যা স্ট্রেচ থাকে এবং প্রতিটাতে ১টা করে পাথর থাকে তাহলে বর্তমান খেলোয়াড়কে হারানো সম্ভব না।

### প্রাইম পাওয়ার গেম

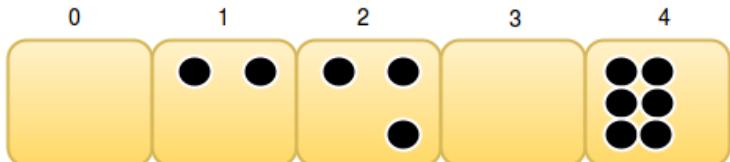
একটা সংখ্যা  $nn$  দেয়া আছে। একজন খেলোয়াড় তার চালে  $nn$  কে কোনো একটা প্রাইম সংখ্যার পাওয়ার দিয়ে ভাগ করতে পারে। সংখ্যাটা যদি 1 হয়ে যায় তাহলে বর্তমান খেলোয়াড় জিতে যাবে।

লক্ষ্য করো যেকোনো সংখ্যা  $n$  কে কিছু প্রাইম সংখ্যার গুণফল হিসাবে লেখা যায়। যেমন  $n=56700=56700$  হলে আমরা লিখতে পারি  $n=(2\times 2)\times(3\times 3\times 3\times 3)\times(5\times 5)\times(7)=22\times 34\times 52\times 71n=(2\times 2)\times(3\times 3\times 3\times 3)\times(5\times 5)\times(7)=22\times 34\times 52\times 71$ । এখন তুমি মনে করে 8টা পাথরের সংখ্যার সেট  $\{2,4,2,1\}\{2,4,2,1\}$ । এখন এটা নিম গেম এ পরিণত হয়েছে, তুমি যেকোনো একটা স্ট্রেচ থেকে এক বা একাধিক পাথর তুলে নিতে পারো!

### নিম্বল(Nimble)

নিম্বল গেম এ  $nn$  টা ঘর থাকে যাদেরকে 00 থেকে  $n-1n-1$  দিয়ে চিহ্নিত করা হয়। প্রতিটা ঘরে এক বা একাধিক কয়েন থাকে। নিচের ছবি দেখো:

**প্রতি চালে কোনো একটা ঘর থেকে একটামাত্র কয়েন সরিয়ে বামের কোনো একটা ঘরে রাখা যায়। যে চাল দিতে পারবে না সে হেরে যাবে। সবগুলো পাথর যখন 0-তম ঘরে চলে আসবে তখন আর চাল দেয়া সম্ভব হবে না।**



লক্ষ্য করো 2 নম্বর ঘরে 3টা পাথর আছে এবং প্রতিটা কয়েনকে ঠিক 2বার করে সরানো সম্ভব। আবার 1 নম্বর ঘরের প্রতিটি কয়েনকে 1বার এবং 8 নম্বর ঘরের প্রতিটি কয়েনকে 8 বার করে সরানো সম্ভব।

আমরা ii-তম ঘরের প্রতিটা কয়েনকে ii আকারের পাথরের স্ট্রেচ হিসাবে চিন্তা করতে পারি কারণ কয়েনটা ঠিক ii বার সরানো সম্ভব। তাহলে উপরের উদাহরণে পাথরের স্ট্রেচের সেট হবে এরকম  $\{1,1,2,2,2,4,4,4,4,4\}\{1,1,2,2,2,4,4,4,4,4\}$ । 6টা স্ট্রেচের আকার 8 কারণ 8 নম্বর ঘরে 6টা কয়েন আছে। এখন গেমটা সাধারণ নিমগেমে পরিণত হয়েছে!

### ম্যাট্রিক্স গেম

একটা  $n\times m\times m$  আকারের গ্রীড দেয়া আছে। প্রতিটি ঘরে কিছু পাথর রাখা আছে। একজন খেলোয়াড় যেকোনো একটা সারিয়ে এক বা একাধিক কলাম থেকে কিছু পাথর তুলে নিতে পারে। প্রতি চালে অন্তত একটা পাথর তুলতেই হবে। শেষ পাথরটা যে তুলবে সে জিতে যাবে।

এই গেমটাকেও সহজে সাধারণ নিম গেম এ রূপান্তর করা যায়। প্রতিটি সারির মোট পাথরসংখ্যাকে একটা স্তুপ হিসাবে চিন্তা করতে হবে। তারপর xorsum বের করলেই কাজ শেষ।

## পোকার নিম

পোকার নিম হলো একধরণের **বোগাস নিম**। বোগাস নিম এ পাথর শুধু তোলাই যায় না, পাথর যোগও করা যায়। পোকার নিম এর নিয়ম হলো যেকোনো স্তুপে এক বা একাধিক পাথর যোগ করা যাবে। কোনো স্তুপ এর আকার যদি শুরুতে হয় mm তাহলে পাথর যোগ করার পরেও আকার mm এর বেশি হতে পারবে না। আর গেমটা যাতে অসীম সময় ধরে না চলে সে জন্য কোনো খেলোয়াড় kk বার এর বেশি পাথর যোগ করতে পারবে না।

|    |   |   |   |
|----|---|---|---|
| 12 | 4 | 6 | 1 |
| 6  | 0 | 4 | 2 |
| 9  | 5 | 1 | 3 |

এই গেম এ যদি প্রথম খেলোয়াড় উইনিং স্টেট এ থাকে ( $xorsum > 0$ ) তাহলে ২য় খেলোয়াড় যখন কিছু পাথর যোগ করবে, প্রথম খেলোয়াড় পাথরগুলো সরিয়ে ফেলে গেমটাকে আবার এই স্টেট এ নিয়ে আসবে! তাই সাধারণ নিম এর মতই xorsumxorsum দেখে বলে দেয়া যায় গেম এ কে জিতবে।

**পরের পর্বে** আমরা আরো কিছু ভ্যারিয়েশন দেখবো এবং গ্রান্ডি সংখ্যা নিয়ে জানবো। এখন লাইটওজে তে [এই সমস্যাগুলো](#) সমাধান করার চেষ্টা করো। নিম এর [আরো কিছু ভ্যারিয়েশন](#) পাবে এখানে।

# অ্যালগোরিদম গেম থিওরি ৩ (স্প্র্যাগ-গ্রান্ডি সংখ্যা)

 shafaetsplanet.com/planetcoding/

শাফায়েত

4/16/2016

আমরা এখন **নিম-গেম** কিভাবে সমাধান করতে হয় জানি, এখন আমরা গ্রান্ডি সংখ্যা দিয়ে কিছু কম্পোজিট গেম এর সমাধান করবো।

একটা গেম এর কথা চিন্তা করি যেখানে  $s$  টা পাথরের একটা স্তুপ(pile) আছে, প্রতিবার কোনো খেলোয়াড় ১টা বা ২টা পাথর তুলে নিতে পারে, শেষ পাথরটা যে তুলে নিবে সে জিতবে। পাথরের সংখ্যা দেখে তোমাকে বলতে হবে অপটিমাল পদ্ধতিতে খেললে প্রথম খেলোয়াড় জিততে পারবে কিনা। প্রথম পর্ব পড়ে থাকলে এটা সমাধান করা তোমার জন্য খুবই সহজ।

আমাদের সুড়েকোড়া হবে এরকম:

```

1 def can_win(s):
2     if s==0: return 0
3     if s==1: return 1
4     if(can_win(s-1)==0 or can_win(s-2)==0):
5         return 1
6     return 0

```

এখন ধরো পাথরের স্তুপের সংখ্যা একটার বদলে যদি  $n$  টা এবং প্রতিটা স্তুপে  $s_1, s_2, \dots, s_n$  টা পাথর আছে। কোনো খেলোয়াড় যেকোনো একটা স্তুপ থেকে ১ বা ২টা পাথর তুলে নিতে পারে, শেষ পাথরটা যে তুলবে সে জিতবে। এবার কিভাবে সমাধান করবে? এধরণের গেমকে কম্পোজিট গেম বলা হয়।

আমরা প্রতিটা স্তুপের জন্য একটা গ্রান্ডি সংখ্যা নির্ধারণ করবো। একটা স্তুপের গ্রান্ডি সংখ্যা  $g$  হলে আমরা ধরে নিতে পারি সেই স্তুপটা নিমগেম এ  $g$  টা পাথরের একটা স্তুপের সমতুল্য।  $n$  টা স্তুপের জন্য আমরা  $n$  টা গ্রান্ডি সংখ্যা  $g[s_1], g[s_2], \dots, g[s_n]$  পাবো, এবং এরপর নিমগেমের মতো  $g[s_1] \oplus g[s_2] \oplus \dots \oplus g[s_n]$  বের করলেই আমরা বুঝে যাবো গেমটায় কে জিতবে।

প্রতিটা স্টেট এর জন্য গ্রান্ডি সংখ্যা বের করার নিয়ম হলো, ওই স্টেপ থেকে অন্য যেসব স্টেট এ যাওয়া যায় সেগুলার গ্রান্ডি সংখ্যা প্রথম বের করতে হবে। মনে করো তাদের সেট হলো  $X = \{x_1, x_2, \dots\}$ । এখন  $S$  স্টেট এর জন্য গ্রান্ডি সংখ্যা এই সেট এ নাই সেটাই হবে বর্তমান স্টেট এর গ্রান্ডি সংখ্যা। একটা উদাহরণ দেখলে ব্যাপারটা পরিষ্কার হবে।

মনে করো কোনো একটা স্তুপে ০টা পাথর আছে। আমাদের স্টেট হলো  $s=0$ । আমরা জানি এটা একটা লুজিং পজিশন, আমরা ধরে নিতে পারি  $s=0$  এর জন্য গ্রান্ডি সংখ্যা  $g[0]=0$ ।

যদি স্তুপে ১টা পাথর থাকে তাহলে  $s=1$ । এখন থেকে ১টা পাথর তুলে  $s=0$  স্টেট এ যাওয়া যায়। সেট  $X$  এ তাহলে আছে  $X=\{g[0]\}=\{0\}$ । ১ হলো সর্বনিম্ন সংখ্যা যা  $X$  এ নেই, তাহলে আমরা ধরে নিবো  $g[1]=1$ ।

যদি স্তুপে ২টা পাথর থাকে তাহলে  $s=2$ । এখন থেকে ১টা বা ২টা পাথর তুলে  $s=1$  বা  $s=0$  স্টেট এ যাওয়া যায়। সেট  $X$  এ তাহলে আছে  $X=\{g[1], g[0]\}=\{0, 1\}$ । ২ হলো সর্বনিম্ন সংখ্যা যা  $X$  এ নেই, তাহলে আমরা ধরে নিবো  $g[2]=2$ ।

যদি স্তুপে ৩টা পাথর থাকে তাহলে  $s=3$ । এখন থেকে ১টা বা ২টা পাথর তুলে  $s=2$  বা  $s=1$  স্টেট এ যাওয়া যায়। সেট  $X$  এ তাহলে আছে  $X=\{g[2], g[1]\}=\{2, 1\}$ । ৩ হলো সর্বনিম্ন সংখ্যা যা  $X$  এ নেই, তাহলে আমরা ধরে নিবো  $g[3]=0$ ।

একই ভাবে হিসাব করলে তুমি পাবে  $g[4]=1$ ,  $g[5]=2$  ইত্যাদি।

এখন যদি ৩টা স্তুপে  $\{5, 2, 3\}, \{5, 2, 3\}$  টা পাথর থাকে তাহলে তুমি বলতে পারো গেমটা  $\{g[5], g[2], g[3]\}=\{2, 2, 0\}$ ।  $\{g[5], g[2], g[3]\}=\{2, 2, 0\}$  টা পাথর নিয়ে সাধারণ নিম গেম খেলার সমতুল্য। এখন তুমি xorsumxorsum দেখে বলতে পারো খেলাটায় কে জিতবে।

গ্রান্ডি সংখ্যা বের করার জন্য সাধারণ সুড়েকোড হতে পারে এরকম:

```

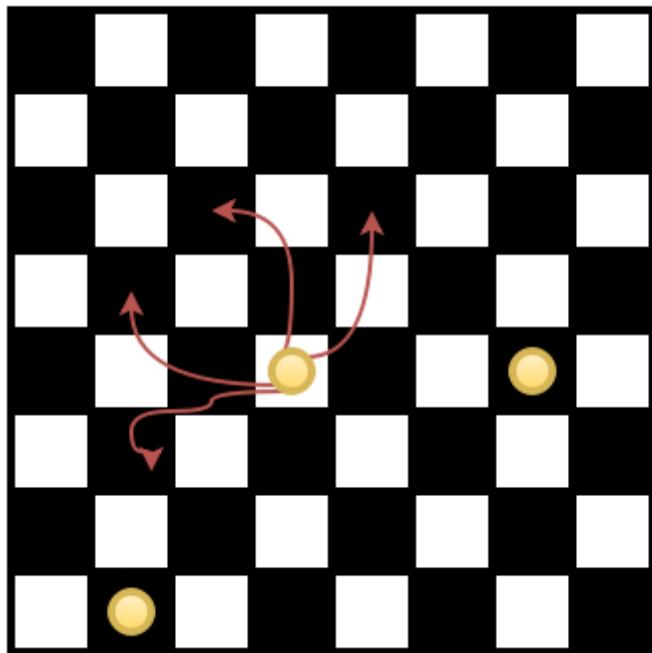
1 def grundyNumber (current_state):
2     moves[] = possible positions to which I can move from current_state
3     set X;
4     for (all new_state in moves)
5         X.insert(grundyNumber(new_state));
6
7     int ret=0;
8     while (s.contains(ret)) ret++;
9     #ret is the smallest non-negative integer not in the set s;
10    return ret;

```

এ পর্যন্ত যদি বুঝে থাকো তাহলে নিচের সমস্যাটা সমাধান করো:

একটা দাবার বোর্ডে kk টা কয়েন রাখা আছে। কোনো কয়েন  $(x,y)$  ঘরে থাকলে প্রতি চালে কয়েনটিকে  $(x-2, y+1)$  অথবা  $(x-2, y-1)$  অথবা  $(x-1, y-2)$  অথবা  $(x+1, y-2)$  ঘরে সরানো যায় , তবে কয়েনটি বোর্ডের বাইরে সরানো যাবে না।

(1,1)



(8,8)

প্রতি চালে বর্তমান খেলোয়াড় উপরের নিয়ম মনে একটা কয়েন সরাতে পারবে। যে খেলোয়াড় কয়েন সরাতে পারবে না সে হেবে যাবে। তোমাকে কয়েনগুলোর স্থানাংক দেয়া থাকবে, বলতে হবে খেলায় কে জিতবে।

এই দাবার বোর্ডে প্রতিটা ঘরের জন্য গ্রান্ডি সংখ্যা হিসাব করলে তুমি নিচের মতো একটা টেবিল পাবে:

লক্ষ্য করে দেখো যেসব ঘর থেকে অন্য কোনো ঘরে যাওয়া যায় না তাদের গ্রান্ডি সংখ্যা ০, তারমানে সেসব ঘরগুলো হলো লুজিং পজিশন। বাকি ঘরগুলোর জন্য গ্রান্ডি সংখ্যা হিসাব করতে হলে প্রথমে একটা ঘর থেকে অন্য যেসব ঘরে যাওয়া যায় তাদের গ্রান্ডি সংখ্যা হিসাব করতে হবে। মনে করো সেই সংখ্যাগুলোর সেট হলো SS। সর্বনিম্ন যে নন-নেগেটিভ সংখ্যা SS এ নেই সেটাই হবে বর্তমান ঘরের গ্রান্ডি সংখ্যা।

আজকে এই পর্যন্তই গ্রান্ডি সংখ্যা কিভাবে নির্ণয় করতে হয় বুঝে থাকলে [এই সমস্যাগুলো সমাধান](#) করো। এই পদ্ধতিটা কেন কাজ করে সেটাও চিন্তা করে বের করার চেষ্টা করো।

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 2 | 1 | 0 | 0 | 1 | 1 |
| 1 | 2 | 2 | 2 | 3 | 2 | 2 | 2 |
| 1 | 1 | 2 | 1 | 4 | 3 | 2 | 3 |
| 0 | 0 | 3 | 4 | 0 | 0 | 1 | 1 |
| 0 | 0 | 2 | 3 | 0 | 0 | 2 | 1 |
| 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 |
| 1 | 1 | 2 | 3 | 1 | 1 | 2 | 0 |

হ্যাপি কোডিং!

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress Themes

# HANDBOOK OF ALGORITHMS

## Section String Algorithm

*Courtesy of*  
*Shafaet Ashraf*

ରବନ୍ଦି-କାର୍ପ ସ୍ଟ୍ରେଟ୍ ମ୍ୟାଚଟି \_ ଶାଫାୟତେରେ ବଲଗ.pdf

# রবিন-কার্প স্ট্রিং ম্যাচিং

 shafaetsplanet.com/planetcoding/

শাফায়েত

12/2/2016

রবিন-কার্প (Rabin-Carp) একটি স্ট্রিং ম্যাচিং অ্যালগোরিদম। দুটি স্ট্রিং দেয়া থাকলে এই অ্যালগোরিদমটি বলে দিতে পারে যে ২য় স্ট্রিংটি প্রথম স্ট্রিং এর সাবস্ট্রিং কিনা। রবিন-কার্প রোলিং হ্যাশ টেকনিক ব্যবহার করে স্ট্রিং ম্যাচিং করে। যদিও স্ট্রিং ম্যাচিং এর জন্য কেবলমাত্র অ্যালগোরিদম ব্যবহার করাই ভালো, কিন্তু রবিন-কার্প শেখা গুরুত্বপূর্ণ মূলত রোলিং হ্যাশ কিভাবে কাজ করে সেটা শেখার জন্য।

এই লেখাটা পড়ার আগে **মডুলার অ্যারিথমেটিক সম্পর্কে** জেনে আসতে হবে।

স্ট্রিং ম্যাচিং করার সময় প্রথম স্ট্রিং টাকে আমরা বলবো টেক্সট (Text) এবং দ্বিতীয়টিকে প্যাটার্ন (Pattern)। আমাদের কাজ হলো টেক্সট এর মধ্যে প্যাটার্ন খুজে বের করা।

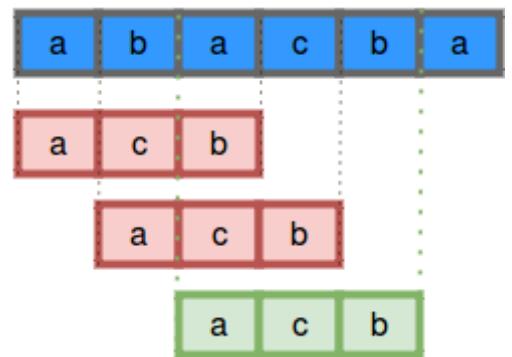
প্রথমে আমরা একটা ব্রুটফোর্স অ্যালগোরিদমের কথা ভাবি। আমরা টেক্সট এর প্রতিটা সাবস্ট্রিং বের করে প্যাটার্নের সাথে মিলিয়ে দেখতে পারি:

ছবিতে abacbaabacba টেক্সট এর ভিতর acbabc প্যাটার্নটা খোজা হচ্ছে।

```

1 function naive_matching(text, pattern){
2     n = text.size()
3     m = pattern.size()
4     for(i = 0; i < n; i++) {
5         for(j = 0; j < m && i + j < n; j++)
6             if(text[i + j] != pattern[j]) break; // mismatch found, break
7         the inner loop
8         if(j == m) // match found
9     }
}

```



নেইভ স্ট্রিং ম্যাচিং অ্যালগোরিদমের কমপ্লেক্সিটি  $O(n \cdot m)$ , যেখানে  $n$  হলো টেক্সট এর দৈর্ঘ্য এবং  $m$  হলো প্যাটার্ন এর দৈর্ঘ্য।

আমাদের যদি দুটি স্ট্রিং তুলনা করার সময় একটা একটা ক্যারেক্টার না দেখে ইন্টিজারের মতো  $O(1)$  এ তুলনা করতে পারতাম তাহলে আমরা খুব দ্রুত স্ট্রিং ম্যাচিং করতে পারতাম। হ্যাশিং টেকনিক ব্যবহার করে আমরা স্ট্রিং কে ইন্টিজারে পরিণত করতে পারি। রবিন-কার্প অ্যালগোরিদম এ সেটারই সুবিধা নেয়া হয়েছে।

আমরা যেকোন স্ট্রিংকে একটা Base-BBase-B সংখ্যা হিসাবে কল্পনা করতে পারি যেখানে BB এর মান অ্যাসকিতে যতগুলো ক্যারেক্টার আছে তার সমান বা বড়। তাহলে আমরা একটা ফাংশন লিখতে পারি যেটা ss কে Base-BBase-B সংখ্যা থেকে Base-10Base-10 সংখ্যায় রূপান্তর করবে। এটাই হতে পারে আমাদের হ্যাশ ফাংশন:

$$\text{Hash}(s) = s_0 \cdot B^{m-1} + s_1 \cdot B^{m-2} + \dots + s_{m-1} \cdot B^1 + s_m \cdot B^0$$

এই ফাংশনে প্যারামিটার হিসাবে একটা স্ট্রিং পাঠ্যনো হয়েছে।  $s_i$  দিয়ে বুঝানো হয়েছে ii তম ক্যারেক্টারের অ্যাসকি ভ্যালু। এই হ্যাশ ফাংশন দিয়ে যেকোনো স্ট্রিং এর জন্য ভিন্ন ভিন্ন হ্যাশভ্যালু পাওয়া যাবে। কিন্তু সমস্যা হলো ওভারফ্লো, হ্যাশভ্যালুর মান সহজেই ৬৪-বিট এর বড় হয়ে যাবে। এই জন্য আমাদেরকে হ্যাশ ভ্যালুটাকে MM দিয়ে ভাগ করে ভাগশেষ (modulo) নিতে হবে। তাহলেই সংখ্যাটা MM এর থেকে ছেটো হয়ে যাবে:

$$\text{Hash}(s) = (s_0 \cdot B^{m-1} + s_1 \cdot B^{m-2} + \dots + s_{m-1} \cdot B^1 + s_m \cdot B^0) \bmod M$$

কিন্তু এইক্ষেত্রে সমস্যা হলো একাধিক স্ট্রিং এর হ্যাশভ্যালু একই হয়ে যেতে পারে। এই সমস্যাটাকে বলা হয় হ্যাশ কলিশন (hash collision)। তবে BB এবং MM যদি প্রাইম সংখ্যা হয় এবং MM এর মান অনেক বড় হয় তাহলে কলিশন করার সম্ভাবনা

খুব কমে যায়। (তবে সম্ভাবনা কমে গেলেও একদম শুণ্য হয়ে যায় না, যে জন্য রবিন কার্পেরও worse case complexity  $O(n*m)O(n*m)$ , সে কথায় পরে আসছি)

এখন প্রথমেই আমাদের কাজ হবে প্যাটার্নের হ্যাশ ভ্যালু বের করা। এরপর টেক্সট এর প্রতিটা mm দৈর্ঘ্যের সাবস্ট্রিং এর জন্য হ্যাশভ্যালু বের করে Hash(pattern)Hash(pattern) এর সাথে মিলিয়ে দেখতে হবে। এখন প্রশ্ন হলো প্রতিটা mm দৈর্ঘ্যের সাবস্ট্রিং এর জন্য হ্যাশভ্যালু কিভাবে বের করবো? যদি প্রতিটা সাবস্ট্রিং কে তুমি উপরের হ্যাশ ফাংশনে পাঠাও তাহলে কমপ্লেক্সিটি হয়ে যাবে  $O(n*m)O(n*m)$ । আমাদেরকে একটা পদ্ধতি বের করতে হবে যেন স্ট্রিং এর উপর শুধু একটা লুপ চালিয়েই  $O(n)O(n)$  এ প্রতিটা mm দৈর্ঘ্যের সাবস্ট্রিং এর জন্য হ্যাশভ্যালু বের করা যায়। এখানেই রোলিং হ্যাশ পদ্ধতি কাজে লাগবে।

মনে করো  $H_0H_1$  হলো ss এর ii তম ইনডেক্সের শুরু হয়েছে এমন mm দৈর্ঘ্যের স্ট্রিং এর হ্যাশ ভ্যালু। তাহলে আমরা লিখতে পারি:

$$H_0 = s_0 \cdot B^{m-1} + s_1 \cdot B^{m-2} + \dots + s_{i-1} \cdot B^0 H_1 = s_i \cdot B^{m-1} + s_{i+1} \cdot B^{m-2} + \dots + s_{i+m-1} \cdot B^0$$

এখন যদি  $m=3$   $m=3$  হয় তাহলে  $H_0H_1$  এবং  $H_1H_0$  কে লিখতে পারি:

$$H_0 = s_0 \cdot B^2 + s_1 \cdot B^1 + s_2 H_0 = s_0 \cdot B^2 + s_1 \cdot B^1 + s_2$$

$$H_1 = s_1 \cdot B^2 + s_2 \cdot B^1 + s_3 H_1 = s_1 \cdot B^2 + s_2 \cdot B^1 + s_3$$

এখন দেখো  $H_1H_0$  কে কিভাবে  $H_0H_0$  এর মাধ্যমে প্রকাশ করা যায়:

$$H_1 = ((s_0 \cdot B^2 + s_1 \cdot B^1 + s_2) - (s_0 \cdot B^2)) \times B + s_3 H_1 = ((s_0 \cdot B^2 + s_1 \cdot B^1 + s_2) - (s_0 \cdot B^2)) \times B + s_3$$

$$H_1 = (H_0 - S_0 \cdot B^2) \cdot B + s_3 H_1 = (H_0 - S_0 \cdot B^2) \cdot B + s_3$$

সাধারণভাবে বলা যায়:

$$H_i = (H_{i-1} - S_{i-1} \cdot B^{m-1}) \cdot B + s_i + m - 1 H_i = (H_{i-1} - S_{i-1} \cdot B^{m-1}) \cdot B + s_i + m - 1$$

এখন এই সূত্র ব্যবহার করে খুব সহজেই  $O(n)O(n)$  এ প্রতিটা mm দৈর্ঘ্যের সাবস্ট্রিং এর হ্যাশ ভ্যালু বের করা যাবে। শুরুতে প্রথম mm ক্যারেক্টারের জন্য হ্যাশভ্যালু বের করে নিয়ে এরপর রোলিং হ্যাশ পদ্ধতিতে বাকিগুলো বের যাবে।

রবিন-কার্পের একটা [সি++ কোড](#) দেখি:

Rabin-carp

C++

```

1 //Implementation of Rabin Carp String Matching Algorithm
2 //https://github.com/Shafaet/Programming-Contest-
3 Algorithms/blob/master/Useful%20C%2B%20Libraries/rabin-carp.cpp
4 #include
5 using namespace std;
6
7 typedef long long i64;
8
9 //Returns the index of the first match
10 //Complexity O(n+m), this is unsafe because it doesn't check for collisions
11
12 i64 Hash(const string &s, int m, i64 B, i64 M){
13     i64 h = 0, power = 1;
14     for(int i = m-1; i>=0; i--){
15         h = h + (s[i] * power) % M;
16         h = h % M;
17         power = (power * B)%M;
18     }
19     return h;
20 }
```

```

21 int match(const string &text, const string &pattern) {
22     int n = text.size();
23     int m = pattern.size();
24     if(n < m) return -1;
25     if(m == 0 or n == 0)
26         return -1;
27
28     i64 B = 347, M = 1000000000+7;
29
30     //Calculate B^(m-1)
31     i64 power = 1;
32     for(int i=1;i<=m-1;i++)
33         power = (power * B) % M;
34
35     //Find hash value of first m characters of text
36     //Find hash value of pattern
37     i64 hash_text = Hash(text, m, B, M);
38     i64 hash_pattern = Hash(pattern, m, B, M);
39
40     if(hash_text == hash_pattern){ //returns the index of the match
41         return 0;
42         //We should've checked the substrings character by character here as hash collision might happen
43     }
44
45     for(int i=m;i<n;i++){
46
47         //Update Rolling Hash
48         hash_text = (hash_text - (power * text[i-m]) % M) % M;
49         hash_text = (hash_text + M) % M; //take care of M of negative value
50         hash_text = (hash_text * B) % M;
51         hash_text = (hash_text + text[i]) % M;
52
53         if(hash_text==hash_pattern){
54             return i - m + 1; //returns the index of the match
55             //We should've checked the substrings character by character here as hash collision might
56             happen
57         }
58     }
59     return -1;
60 }
61
62
63 int main() {
64     cout<<match("HelloWorld", "ello")<<endl;
65     return 0;
66 }
```

এই কোডের কমপ্লেক্সিটি  $O(n+m)O(n+m)$ । কিন্তু এখানে একটা বড় সমস্যা আছে। যখন `hash_text` আর `hash_pattern` মিলে যাচ্ছে তখন আমরা ধরে নিচ্ছি যে স্ট্রিং দুটি মিলে গিয়েছে। কিন্তু আগে বলেছিলাম যে ভাগশেষ(mod) নেয়ার কারণে একাধিক স্ট্রিং এর একই হ্যাশভ্যালু আসতে পারে (কলিশন হতে পারে)। সে ক্ষেত্রে এই কোড ভুল আউটপুট দিবে।

কলিশনের কারণে ভুল এড়ানোর একটা উপায় হলো, যখনই `hash_text = hash_pattern` হবে তখন আবার লুপ চালিয়ে ক্যারেক্টার বাই ক্যারেক্টার মিলিয়ে দেখা। কিন্তু সেক্ষেত্রে worse case কমপ্লেক্সিটি  $O(n*m)O(n*m)$  হয়ে যাচ্ছে যেটা ব্রুট ফোর্সের কমপ্লেক্সিটির সমান।

আরেকটা উপায় হলো ডাবল হ্যাশিং করা। ডাবল হ্যাশিং মানে হলো ভিন্ন ভিন্ন B এবং M ব্যবহার করে প্রতিটা স্ট্রিং এর জন্য দুটি করে হ্যাশ ভ্যালু বের করবো। সেক্ষেত্রে দুই জায়গাতেই কলিশনের সম্ভাবনা খুবই কমে যাবে। চাইলে ২বারের বেশিও হ্যাশিং করা যায়। ডাবল হ্যাশিং করলে প্রোগ্রামিং কনটেস্টে বেশিভাগ সময় পার পেয়ে যাওয়া যাবে, কিন্তু বাস্তব ক্ষেত্রে এটা খুব একটা ভালো উপায় না, কারণ কলিশনের সম্ভাবনা কমে গেলেও একদম শূন্য হয়ে যাচ্ছে না।

এসব কারণে রবিন-কার্প স্ট্রিং ম্যাচিং এর জন্য তেমন একটা ব্যবহার করা হয় না। কিন্তু রোলিং হ্যাশ টেকনিক অনেক ধরণের সমস্যা সমাধান করতে কাজে লাগে যে কারণে আমরা রবিন-কার্প শিখি।

### চিন্তা করার জন্য সমস্যা:

তোমাকে দুটি স্ট্রিং s1,s2s1,s2 দেয়া আছে। তোমাকে এদের লংগেস্ট কমন সাবস্ট্রিং এর দৈর্ঘ্য বের করতে হবে। অর্থাৎ সবথেকে বড় স্ট্রিং বের করতে হবে যেটা দুটি স্ট্রিং এরই সাবস্ট্রিং।

### সমাধান:

প্রথমে চিন্তা করো যেকোনো ইন্টিজার kk এর জন্য কি তুমি বের করতে পারবে যে kk দৈর্ঘ্যের কোনো লংগেস্ট কমন সাবস্ট্রিং আছে নাকি। এটা সহজেই  $O(n)O(n)$  কমপ্লেক্সিটিতে বের করা যাবে রোলিং হ্যাশ ব্যবহার করে। s1s1 এর প্রতিটা kk দৈর্ঘ্যের সাবস্ট্রিং এর হ্যাশভ্যালু বের করো, এরপর s2s2 এর প্রতিটা kk দৈর্ঘ্যের সাবস্ট্রিং এর হ্যাশভ্যালু বের করো। যদি কোনো হ্যাশভ্যালু কমন থাকে তারমানে kk দৈর্ঘ্যের কোনো লংগেস্ট কমন সাবস্ট্রিং আছে।

এখন লংগেস্ট কমন সাবস্ট্রিং এর সর্বোচ্চ দৈর্ঘ্য হতে পারে

$\text{maxlen}=\min(s1.length,s2.length)$  maxlen= $\min(s1.length,s2.length)$ । এখন তুমি 00 থেকে maxlen maxlen এর উপর বাইনারি সার্চ চালিয়ে সহজেই সমস্যাটা সমাধান করতে পারবে। কমপ্লেক্সিটি হবে  $O(n \cdot \log n)O(n \cdot \log n)$ ।

হ্যাপি কোডিং!

রেফারেন্স: [টপকোডার](#)

AccessPress Staple | WordPress Theme: [AccessPress Staple](#) by AccessPress

Themes

