প্রোগ্রামিং প্রতিযোগিতার শুরুর গল্প

Dawn of Programming Contest

মোঃ মাহবুবুল হাসান

মার্চ, ২০১৫

সূচীপত্র

নকশা তালিকা

সারণী তালিকা

অধ্যায় ১

প্রোগ্রামিং প্রতিযোগিতায় হাতে খড়ি

১.১ শুরুর কথা

প্রতিযোগিতা মানেই আনন্দ। আমরা ফুটবল দেখি, ক্রিকেট দেখি, টেনিস দেখি এরকম হরেক রকমের খেলা আমরা ঘণ্টার পর ঘণ্টা দেখি। রাত জেগে ঘুম হারাম করে দেখলেও কিন্তু আমরা ক্লান্ত হই না, বরং খেলা শেষে আমরা হই হই করে আনন্দে মেতে উঠি বা কষ্টে কারো সাথে কথা না বলে বিপক্ষ দলকে শাপ শাপান্ত করতে থাকি। প্রোগ্রামিং প্রবলেম সল্ভিং এও ঠিক একই রকম মজা। তুমি খেতে বসে দেখবে দ্রুত খাচ্ছ কারণ তুমি হাত ধুতে গিয়ে একটা সমস্যার সমাধান পেয়ে গেছ! অথবা দেখবে ক্লাসে তোমার টিচার এর পড়ানোর দিকে মন নেই, তুমি দিব্যি তোমার পাশের বন্ধুর সাথে আগের রাতের কন্টেস্ট এর প্রবলেম নিয়ে আলোচনা করছ। অথবা এও হতে পারে যে গভীর রাতে তুমি কম্পিউটার এ আছো, আর তোমার মা বকা দিতে দিতে আসতে আসতে বলবে - "সারারাত গেম খেলা হচ্ছে না?" কিন্তু তোমার স্ক্রিনের দিকে অবাক হয়ে বলবে- "এসব কি করিস?" আর তুমি হেসে বলবে- "কোডিং করছি, তুমি বুঝবে না, যাও ঘুমাও"। আসলে এই মজা যে পেয়েছে সেই শুধু বুঝবে আমি কি বলছি! হয়তো এখন তুমি আমার কথা বিশ্বাস করবে না, কিন্তু এক সময় তুমি বুঝবে আমি কি বুঝাতে চাচ্ছি। তোমরা যেন সবাই সেই আনন্দটা পাও এই আশা নিয়েই শুরু হোক আমাদের প্রোগ্রামিং প্রতিযোগিতায় হাতে খড়ি।

১.২ প্রোগ্রামিং প্রতিযোগিতা কি?

তোমরা যদি মনে কর যে প্রোগ্রামিং প্রতিযোগিতায় বুঝি কোন একটি প্রোগ্রামিং ল্যাঙ্গুয়েজ যে যত ভালো পারে সে তত ভালো করবে তাহলে জেনে রাখ তোমার এই ধারনা সম্পূর্ণ ভুল। আমরা কিন্তু সেই ছোট্ট বেলাতেই ১, ২, ৩, ৪ শিখেছি, শিখেছি যোগ করা, বিয়োগ করা, গুন করা, ভাগ করা। কিন্তু এখানেই কিন্তু গণিত এর শেষ হয় নাই। এর পরেও অনেক অনেক অনেক কিছু আমরা জেনেছি শিখেছি। প্রোগ্রামিং ল্যাঙ্গুয়েজও এখানে সেই ১, ২, ৩, ৪ এর মত। আমরা গনিতে সংখ্যাগুলোকে যেমন এই সব অঙ্ক দিয়ে প্রকাশ করে থাকি ঠিক তেমনি আমাদের প্রোগ্রামিং প্রতিযোগিতার সমস্যার সমাধানগুলো এই ল্যাঙ্গুয়েজ দিয়ে প্রকাশ করে থাকি। এই ল্যাঙ্গুয়েজ আমাদের সমাধান প্রকাশের একটি মাধ্যম মাত্র। এটি এমন একটি মাধ্যম যা আমাদের কম্পিউটার বুঝে থাকে। তোমরা মনে কর না যে কম্পিউটার নিজে নিজেই সব করে থাকে, আমি একটা সংখ্যা দিলে সে এমনি এমনিই বলে দিবে না যে সংখ্যাটা জোড় না বিজোড়। তোমাকে বলে দিতে হবে সে কেমনে বুঝবে যে সংখ্যাটা জোড় না বিজোড়। সে যা পারে তা হচ্ছে অনেক দ্রুত হিসাব করা আর অনেক বড় বড় জিনিস মেমোরিতে মনে রাখা। তুমি তাকে বলে দিবে কেমনে হিসাব করতে হবে, কখন কোথায় কি মনে রাখতে হবে। ব্যাস সে চোখের পলকে তোমাকে সেই হিসাব করে দিবে। কোন ভুলে সে করবে না। কিন্তু তুমি যদি ওকে বলতে ভুল কর তাহলে কিন্তু সেটা তোমার দোষ ওর না। প্রোগ্রামিং প্রতিযোগিতা হল তুমি ঠিক মত তোমার কম্পিউটারকে সমাধানের উপায় বলে দিতে পারছ কিনা তার প্রতিযোগিতা। বিভিন্ন ধরনের প্রোগ্রামিং প্রতিযোগিতা আছে। কে কত সুন্দর ভাবে ডিজাইন করতে পারে, কে কত সুন্দর করে লজিক দাঁর করাতে পারে, কে কত efficient সমাধান করতে পারে

ইত্যাদি। আমরা এই বই এ যেই প্রতিযোগিতা নিয়ে কথা বলব তা ACM প্রোগ্রামিং প্রতিযোগিতা নামে পরিচিত। এখানে দেখা হয় তোমার সমাধান কত দ্রুত একটা সমস্যার সমাধান করতে পারে, কত কম মেমরি নিতে পারে বা এমনও হতে পারে যে তোমার হাতে মেমোরি অনেক আছে কিন্তু সময় কম, সুতরাং সেক্ষেত্রে তুমি হয়তো মেমোরি বেশি ব্যাবহার করবে কিন্তু সময় কম লাগাবে। অর্থাৎ তুমি কত দক্ষতার সাথে তোমাকে দেয়া সীমাবদ্ধতার মাঝে সমস্যার সমাধান করতে পারছ।

প্রতিযোগিতা মোটামোটি দুই রকমের হয়ে থাকে। ACM Style এবং Informatics Olympiad Style. ACM প্রোগ্রামিং এ সাধারনত বিশ্ববিদ্যালয় লেভেল এর ছেলেমেয়েরা অংশগ্রহণ করে থাকে। বিভিন্ন রকমের টপিক থেকে প্রবলেম আসে, Number Theory, Calculus, Graph Theory, Game Theory, Dynamic Programming ইত্যাদি। এখানে আসলে টপিক এর সীমাবদ্ধতা নেই। এটি বেশিরভাগ সময় দলগত প্রতিযোগিতা হতে থাকে। অন্যদিকে Informatics Olympiad এ স্কুল কলেজ লেভেলের ছেলেমেয়েরা অংশ নিয়ে থাকে। এটিতে একটি নির্দিষ্ট সিলেবাস থেকে প্রশ্ন হয়ে থাকে। তবে প্রবলেম গুলো অনেক বেশি Algorithmic হয়ে থাকে। এটা Individual প্রতিযোগিতা।

১.৩ কেন করব?

[১] অনেকে মনে করতে পারে যে প্রোগ্রামিং প্রতিযোগিতায় যারা ভালো করেছে তারা Google, Facebook, Microsoft এর মত বড় বড় কোম্পানিতে চাকরি করে, অনেক অনেক টাকা কামায়। কিন্তু সত্যি কথা বলতে কি দিনের শেষে টাকাই সব কথা নয়। তোমার মনের সুখ কিন্তু অনেক বড় জিনিস। (কি বেশি আধ্যাত্মিক কথা হয়ে গেল?) তুমি আনন্দ নিয়ে প্রোগ্রামিং করবে। তাহলেই দেখবে তুমি ভালো করছ, তখন Google, Facebook, Microsoft এর মত কোম্পানি এমনিই তোমাকে নিয়ে যাবে। বা ঐ সব কোম্পানি কেন? হয়তো তুমি নিজেই একটা Google প্রতিষ্ঠা করে ফেলবে একদিন। অথবা তুমি হয়তো এমন একটা প্রোগ্রামিং ল্যাঙ্গুয়েজ বানায়ে ফেলবে যেটা হয়তো সারা বিশ্বের মানুষ ব্যাবহার করবে। এর মাঝে অন্যরকম মজা আছে। এই অপার্থিব আনন্দ যারা পেতে চাও তাদের জন্য এই প্রোগ্রামিং প্রতিযোগিতা।

১.৪ কেমনে শুরু করব?

এটা হল Internet এর যুগ। (আমি ২০১৩ সালের কথা বলছি... হয়তো অদূর ভবিষ্যতে internet বলে কিছুই থাকবে না, এর থেকেও আধুনিক জিনিস চলে আসবে) নাই বলে কথা নাই। তোমার সামনে internet আছে তুমি শিখ! শিখার উৎসের কিন্তু শেষ নেই। তুমি চাইলে Youtube এ সার্চ করে সেখানে ভিডিও লেকচার দেখতে পার। বা অন্যদেশের বই translator দিয়ে অনুবাদ করে পড়তে পার। বা তোমার থেকে যারা ভালো পারে তাদের কাছ থেকেও শিখতে পার। মোট কথা তোমার শিখার ইচ্ছা থাকলে তোমাকে কেউ দোমাতে পারবে না।

অনলাইনে বেশ কিছু website আছে যেখানে কিছু দিন পর পর Contest হয়। এসব জায়গায় পুরান কন্টেস্ট এর প্রবলেমও পাওয়া যায়। খেলোয়াড়রা যেমন main খেলার আগে প্র্যাকটিস করে, ঠিক তেমনি আমরা সেইসব পুরান প্রতিযোগিতার প্রবলেম সমাধান করে প্র্যাকটিস করতে পারি। বেশিরভাগ সাইট এই আবার কে কতটা সমাধান করেছে তার একটা তালিকা থাকে। দেখোতো তোমার কোন বন্ধু ঐ তালিকায় তোমার আগে আছে কিনা? থাকলে তার থেকে বেশি সমাধান কর। চেষ্টা কর তোমার সহপাঠীদের মাঝে সবচেয়ে আগিয়ে থাকার, এর পরে চেষ্টা কর দেশের সবার মাঝে আগিয়ে থাকার। চেষ্টা করতে থাকো। এক সময় বিশ্বে সবার মাঝে আগিয়ে থাকতে পারো কিনা দেখ। তুমি যদি না পার তাহলে কিন্তু হতাশ হবার কিছু নেই। এইযে তোমার আগানোর চেষ্টা, এটাই তোমাকে প্রতিযোগিতা না করা অন্য ১০০ জনের চেয়ে আগিয়ে রাখবে। তুমি নিজেই বুঝবে না তুমি অন্যদের তুলনায় কত দক্ষ হয়ে গিয়েছ! হয়তো তুমি প্রথম ১০ জনের এক জন না, কিন্তু আগে হয়ত তুমি ১০০০ জনের মাঝে ছিলেনা, এখন ১০০ জনের মাঝে এসেছ এটা কিন্তু সামান্য অর্জন না!

নিচে তোমাদের সুবিধার জন্য কিছু website এর লিংক এবং এদের সংক্ষিপ্ত বিবরনি দেয়া হলঃ

[১] সত্যি কথা বলতে এই সেকশন এর টাইটেল ছিল "কাদের জন্য এই বই না" এবং বলা বাহুল্য এতে বেশ খানিকটা অপ্রিয় সত্য কথা ছিল বৈকি।

uva.onlinejudge.org প্রোগ্রামিং প্রতিযোগিতার জন্য সবচেয়ে popular ওয়েব সাইট। এখানে প্রায়ই ৫ ঘণ্টার প্রতিযোগিতা হয়ে থাকে বিশেষ করে অক্টোবর থেকে ডিসেম্বর এই সময়ে। এছাড়াও এখানে আছে ৪০০০ এরও বেশি প্র্যাকটিস প্রবলেম।

icpcarchive.ecs.baylor.edu এটা uva ওয়েব সাইট এর ভাই। এখানে ১৯৮৮ সাল থেকে শুরু করে আজ পর্যন্ত হয়ে আসা বহু ICPC Regional Programming Contest এবং ACM ICPC World Finals এর প্রবলেম সমুহ আছে।

acm.sgu.ru রাশিয়ান প্রোগ্রামিং সাইট। এখানে তুলনামূলক ভাবে অনেক কম প্রবলেম আছে, কিন্তু একেকটা প্রবলেম সল্ভ করতে মাথার ঘাম পায়ে পরে!

acm.timus.ru আরও একটি রাশিয়ান সাইট। এখানে প্রবলেম গুলো বেশ কিছু ক্যাটাগরিতে ভাগ করা আছে। তোমরা যারা recently প্রোগ্রামিং শুরু করেছ তারা এইখানের Beginners Problem সেকশনের ২০টি প্রবলেম সল্ভ করে দেখতে পার। ওগুলো সল্ভ করতে কোন এলগোরিদম বা ডাটা স্ট্রাকচার এর দরকার হয় না, শুধু প্রোগ্রামিং ল্যাঙ্গুয়েজ জানলেই চলে।

www.codechef.com এখানে প্রতিমাসে তিন ধরনের কন্টেস্ট হয়। একটি ১০দিন ব্যাপি, একটি ৫ ঘণ্টার ACM স্টাইল আরেকটি ৪ ঘণ্টার IOI স্টাইল কন্টেস্ট। যারা ভালো করে তারা এখানে প্রাইজ পেয়ে থাকে।

www.codeforces.com প্রতি সপ্তাহে প্রায় ২টি করে কন্টেস্ট হয়ে থাকে এখানে। দুই Division এ হয়ে থাকে সেই কন্টেস্ট। Division 2 তে আছে Beginner রা, আর Division 1 এ আছে Advanced রা। এখানে প্রতি কন্টেস্ট শেষে তোমার Rating আপডেট হবে। সুতরাং তুমি তোমাকে তোমার দেশের বা বিশ্বের আর সবার সাথে তুলনা করতে পারবে!

www.topcoder.com/tc codeforces এর মতই এখানেও Rating system আছে। এখানে প্রতি মাসে প্রায় ৩ থেকে ৪ টি কন্টেস্ট হয়ে থাকে।

acm.hust.edu.cn/vjudge এটি মুলত প্র্যাকটিস কন্টেস্ট host করতে ব্যাবহার হয়ে থাকে। এর মাধ্যমে তুমি অন্যান্য ওয়েব সাইট এর প্রবলেম গুলি নিয়ে কন্টেস্ট host করতে পার।

www.lightoj.com এটি এখন পর্যন্ত একমাত্র বাংলাদেশী Online Judge (Practice ওয়েব সাইট গুলিকে আমরা Online Judge বা সংক্ষেপে OJ বলে থাকি)। এটা বানিয়েছে জানে আলম জান। উনি Dhaka University হতে ২০০৯ সালে ACM ICPC World Finals এ অংশ গ্রহন করেছেন।

www.z-trening.com এটি সাম্প্রতিক সময়ে প্রায়শই down থাকে। কিন্তু এটি Informatics Olympiad এর জন্য প্রস্তুতি নেবার জন্য খুবই ভালো ওয়েব সাইট।

ipsc.ksp.sk সমগ্র বিশ্ব ব্যাপি IPSC খুবই সমাদৃত একটি প্রতিযোগিতা। বছরে একবার এই কন্টেস্ট হয়ে থাকে এবং সবাই এই কন্টেস্ট করার জন্য মুখিয়ে থাকে। এখানে বিভিন্ন ধরনের প্রবলেম দেয়া হয়ে থাকে। Encryption, Music, New Language, Game আরও নানা টাইপ এর প্রোগ্রামিং সমস্যা দেয়া হয়ে থাকে যা অন্য কন্টেস্টগুলোতে দেখা যায়না বললেই চলে।

www.spoj.com এটি একটি Polish ওয়েব সাইট। এখানের সমস্যা গুলোও বেশ ভালো। বিভিন্ন দেশের Informatics Olympiad এর প্রবলেম গুলো এখানে পাওয়া যায়। এছাড়াও এখানে বিভিন্ন টপিক এর বেশ কঠিন কঠিন এবং শিক্ষণীয় প্রবলেম থাকে। কিন্তু অনেক প্রবলেম এর মাঝ থেকে বাছাই করা একটি কঠিন কাজ।

ace.delos.com/usacogate USA এর Informatics Olympiad দলকে প্রশিক্ষণ দেবার জন্য মুলত এই ওয়েব সাইট। সেপ্টেম্বর হতে এপ্রিল মাস পর্যন্ত প্রতিমাসে সাধারনত একটি করে কন্টেস্ট হয়ে থাকে IOI এর প্রস্তুতি স্বরূপ। এখানের offline প্রবলেমগুলো[১] Catagorywise আলাদা করা আছে, এবং তুমি কোন সমস্যা সমাধান করলে তার analysis পাবে।

[১] offline বলতে আমরা প্র্যাকটিস প্রবলেম বুঝে থাকি

কিছু চাইনিজ অনলাইন জাজ বেশ কিছু চাইনিজ OJ আছে। acm.pku.edu.cn, acm.zju.edu.cn, acm.tju.edu.cn এই তিনটি প্রধান বলতে পার। pku সাইটে আগে মাসিক কন্টেস্ট হতো এবং তার analysis পাবলিশ করা হতো। এখনো pku এবং zju সাইটে কন্টেস্ট হয়ে থাকে। tju ওয়েব সাইট এও অনেক প্রবলেম আছে, তবে মুলত এখানে প্র্যাকটিস কন্টেস্ট আয়োজন করা হয়ে থাকে।

projecteuler.net এই ওয়েব সাইটটিতে অনেক mathematical সমস্যা আছে যেগুলো তুমি হাতে কলমে সমাধান করতে পারবে না, কিন্তু কোড করে সমাধান করতে পারবে। কিন্তু এখানে মুল জিনিস হল তোমার mathematical skill. অ্যালগোরিদম স্কিল এর থেকে এখানে তোমার mathematical skill এরই চর্চা বেশি হয়।

uva সাইট এর কিছু হেল্পিং টুল uhunt.felix-halim.net ও uvatoolkit.com হল এমন দুটি টুল। এখানে তুমি নির্দিষ্ট ক্যাটাগরির প্রবলেম এর তালিকা পাবে, তোমার userid দিলে তোমার জন্য এরপরে কোন প্রবলেম সল্ভ করা ভালো হবে তার তালিকা পাবে। এছাড়াও দুজনের userid দিলে তাদের মাঝে সল্ভ করা প্রবলেম এর comparison দেখায়। উল্লেখ্য যে, দুই ভাই Felix Halim এবং Steven Halim প্রোগ্রামিং জগতে বেশ নামী নাম। তারা একটি বই লিখেছে যাতে ব্যবহৃত প্রবলেম এর তালিকাও এই ওয়েব সাইটটিতে পাওয়া যাবে।

www.e-maxx.ru এটি রাশিয়ান ভাষায় লিখা একটি ওয়েব সাইট। এখানে তুমি বিভিন্ন টপিক এর উপর লিখা article পাবে। তুমি Google Translator কিংবা Bing Translator ব্যাবহার করে পড়ে দেখতে পার!

infoarena.ro এটি রোমানিয়ান ভাষায় লিখা একটি ওয়েব সাইট। এখানেও বেশ কিছু ভালো article আছে।

http://www.hsin.hr/coci/ এটি Croatian Informatics Olympiad এর ওয়েব সাইট। এখানে বেশ কিছু Practice Contest হয়ে থাকে।

এখন কিছু বই এর নাম বলা যাক।

Introduction to Algorithms লেখক Thomas H. Cormen, Charles E. Leserson, Ronald L. Rivest এবং Clifford Stein

Programming Challenges লেখক Steven S Skiena এবং Miguel A Revilla

The Algorithm Design Manual লেখক Steven S Skiena

Algorithm Design লেখক Jon Kleinberg এবং Eva Tardos

Computational Geometry: Algorithms and Applications লেখক Mard de Berg, Otfried Cheong, Marc van Kreveld এবং Mark Overmars

Computational Geometry in C লেখক Joseph O'Rourke

Art of Programming Contest লেখক Ahmed Shamsul Arefin

Data Structures and Algorithms in C++ লেখক Michael T. Goodrich, Roberto Tamassia এবং David M. Mount

Competitive Programming লেখক Felix Halim এবং Steven Halim

অনেক কিছুই তো পেলে, কিন্তু এখানে সবকিছুর কিন্তু দরকার নেই। তোমার যেটা ভালো লাগবে তা থেকে শুরু করবে। যেই OJ তে প্রবলেম সল্ভ করতে ভালো লাগবে সেখানে সল্ভ করবে। যে বই পড়তে ভালো লাগবে সেই বই পড়বে। কিন্তু প্রধান কাজ হল করতে হবে। তুমি যদি নিজে না দেখে অন্যদের জিজ্ঞাসা করে বেরাও যে কি করব কি পড়বো, তাহলে কিন্তু হবে না। যত বেশি প্র্যাকটিস করবে তত ভালো করতে পারবে। আগেই বলেছি এটা internet এর যুগ, তুমি net এ সার্চ করলেই এসব বই এর preview দেখতে পাবে। সেখানে থেকেও তুমি বুঝতে পারবে কোন বইটা তোমাকে suite করছে।

১.৫ কি কি জানতে হবে?

শুরু করার জন্য তোমাকে শুধু একটি প্রোগ্রামিং ল্যাঙ্গুয়েজ জানলেই চলবে। ACM প্রতিযোগিতাগুলোতে C, C++, Java এই তিনটি ভাষা ব্যবহার হয়ে থাকে। অন্যদিকে Informatics Olympiad গুলোতে C, C++ এবং Pascal ব্যবহার হয়। আমরা বাংলাদেশের প্রোগ্রামিং প্রতিযোগিতাগুলোয় দেখে আসছি যে ৯৯.৯৯% মানুষই C/C++ ব্যবহার করে থাকে। এখানে বলে রাখা ভালো যে, অনেকে মনে করে C আর C++ একদম আলাদা। আসলে তা না। তুমি C তে যা যা লিখবা তা C++ এও চলবে। উপরন্ত C++ এ কিছু বাড়তি সুবিধা আছে যা আমরা আমাদের সুবিধার জন্য ব্যবহার করে থাকি। তবে এটা সত্য যে আমরা C++ বলতে আসলে যেই Object Oriented Programming বুঝে থাকি তার ছিটে ফোটাও আমরা আমাদের প্রোগ্রামিং প্রতিযোগিতায় ব্যবহার করি না বললেই চলে। সুতরাং তোমরা যদি C শিখ তাহলেই প্রোগ্রামিং প্রতিযোগিতা শুরু করে দিতে পারবে। তোমাকে পুরো C শিখে এর পরে শুরু করতে হবে তাও কিন্তু না। তোমাদের সুবিধার জন্য আমরা ২ নং অধ্যায়ে ধাপে ধাপে কিছু প্রোগ্রামিং প্রবলেম নিয়ে আলোচনা করব। তবে এই বই কিন্তু তোমাদের C শিখানোর জন্য না। তোমরা যেন C ঝালাই করে নিতে পার এজন্য ২ নং অধ্যায়ে খুব অল্প কথায় C এর কিছু জিনিস আমরা তুলে ধরেছি। মজার জিনিস হচ্ছে প্রায় সব ল্যাঙ্গুয়েজ এরই basic জিনিস সব একই রকম। if-else থাকবে, loop থাকবে, function, array সবই থাকবে, শুধু লিখবে একটু অন্য ভাবে। এজন্য তুমি যদি একটি ল্যাঙ্গুয়েজ জানো তাহলে অন্য ল্যাঙ্গুয়েজ এর code ও বুঝতে পারবে। মাঝে মাঝে কিছু জিনিস না বুঝলে internet তো আছেই!

অধ্যায় ২

C ঝালাই

২.১ একটি ছোট প্রোগ্রাম এবং ইনপুট আউটপুট

আমরা শুরু করব খুবই simple একটি প্রোগ্রাম দিয়ে (কোডঃ ২.1)। এই প্রোগ্রামটা তোমরা run করলে দেখবে যথারীতি 6 আর 2 এর যোগফল 8 দেখাবে। নিশ্চয়ই বুঝতে পারছ আমরা যোগ চিহ্ন এর জায়গায় বিয়োগ, গুন বা ভাগ চিহ্ন ব্যবহার করলে যথাক্রমে 4, 12 এবং 3 দেখাবে।

Listing ২.1 : simple code.cpp

```
1 #include<stdio.h>
2
3 int main()
4 {
5         printf("%d\n", 6 + 2);
6         return 0;
7 }
```

কিন্তু তোমরা ভাবতে পার যে এই একটা যোগ করতেই এত বড় কোড লিখতে হবে? আসলে খুব শীঘ্রই বুঝতে পারবে যে খুব ছোট কোড দিয়ে কেমনে অনেক বড় বড় কাজ করে ফেলা যায়। কেবল তো শুরু! যাই হোক, বেশি দূরে যাবার আগে সংক্ষেপে দেখে নেই প্রতিটা লাইন এর মানেঃ

Line 1 stdio.h নামের header file কে include করা। বিভিন্ন ধরনের header file আছে। এই ফাইলটির কাজ হল ইনপুট আউটপুট এর কাজ করা। stdio এর পূর্ণ অর্থ হল standard input output.

line 2 empty line. আমরা কোড এর সৌন্দর্যের জন্য এরকম ফাঁকা লাইন বা space বা tab দিয়ে থাকি। এতে করে পরবর্তীতে তোমারই কোড বুঝতে সুবিধা হবে।

line 3 এখান থেকে main function শুরু হয়েছে। যখন তোমার কোড run করবে তখন এই function থেকেই কাজ শুরু হয়। আর এই function একটি integer ডাটা return করে।

line 5 এখানে দুটি সংখ্যার যোগফল প্রিন্ট করা হচ্ছে। তুমি যদি একই সাথে যোগফল এবং বিয়োগফল প্রিন্ট করতে চাও তাহলেঃ printf("%d %d\n", 6 + 2, 6 - 2) এভাবে লিখতে পার।

line 6 main function টি 0 সংখ্যা return করবে।

এখন এই প্রোগ্রাম (কোডঃ ২.1) এর সমস্যা হল, তুমি যেই যেই সংখ্যা যোগ করতে চাও তোমাকে সেই দুটি সংখ্যা কোড এ গিয়ে পরিবর্তন করে আবার run করতে হবে। আমরা চাইলে user এর কাছ থেকে দুটি সংখ্যা চেয়ে তাদের যোগ করতে পারি। অর্থাৎ দুইটি সংখ্যা ইনপুট নিয়ে তাদের প্রসেস করে আমরা আউটপুট করতে চাই এবং এজন্য তোমাদের scanf ফাংশন ব্যবহার করতে হবে (কোডঃ ২.2)।

Listing ২.2 : simple input.cpp

```
#include<stdio.h>

int main()
{
        int a, b;
        scanf("%d %d", a, b);
        printf("%d\n", a + b);
        return 0;
}
```

এখানে a এবং b হল দুটি variable. আমরা বীজগণিত করার সময় যেমন অজানা কিছুর মান কে x, y, a, b, p এরকম ধরে থাকি, ঠিক তেমনি আমাদের C তেও variable আছে। আমরা ইনপুট নেবার আগে কিন্তু জানি না যে দুটি সংখ্যার মান কি। সুতরাং আমরা a এবং b নামে দুটি variable কে declare করেছি। এবং scanf ফাংশন দিয়ে তাতে মান ইনপুট নিয়েছি। তোমরা ভাবতে পার যে, আউটপুট এর সময় শুধু a আর b ছিল অথচ ইনপুট এর সময় কেন &a আর &b দেয়া হল? একটু পরেই এই জিনিসটা বুঝতে পারবে। আপাতত মনে কর যে এভাবেই লিখতে হবে। সুতরাং এখন এর পরের লাইনে আমরা $6 + 2$ এর জায়গায় শুধু $a + b$ লিখলেই ইনপুট দেয়া সংখ্যা দুটি যোগ করে দেখিয়ে দিবে। আমরা যোগ এর পরিবর্তে চাইলে বিয়োগ (-), গুন (*), ভাগ (/) বা ভাগশেষ চিহ্ন (%) ব্যবহার করতে পারি। [১]

এখন কথা হল, এতটুকু শিখেই কি একটা প্রবলেম সল্ভ করে ফেলা সম্ভব? অবশ্যই সম্ভব! আমরা তোমাদের practice এর জন্য এই অধ্যায়ের প্রতিটি সেকশন এর শেষে ঐ সেকশন সম্পর্কিত কিছু সমস্যা দিব। তোমাদের সুবিধার জন্য কিছু কিছু সমস্যার hints এই অধ্যায় এর শেষে থাকবে। যদি কোন সমস্যা Online Judge হতে নেয়া হয় তাহলে OJ এর নাম এবং প্রবলেম নাম্বার দেয়া থাকবে।

প্র্যাকটিস প্রবলেম

• Timus 1000 • Timus 1264 • Timus 1293 • Timus 1409

২.২ ডাটা টাইপ এবং math.h হেডার ফাইল

এখন কিছু experiment করা যাক। তোমরা আগের প্রোগ্রাম (কোড ২.2) এ যোগ এর জায়গায় গুন দাও ($a * b$) আর run করে 100000 এবং 100000 ইনপুট দাও। দেখবে মাথা খারাপ করা উত্তর দিয়েছে (ঋণাত্মক দিলেও কিন্তু অবাক হয় না)! না, তুমি তোমার প্রোগ্রামে ভুল কর নি। তুমি কিছু ছোট সংখ্যা দিলেই দেখবে তোমার প্রোগ্রাম দিব্যি সঠিক উত্তরটিই দিচ্ছে। আসলে সব কিছুর একটা সীমা আছে। (যারা python জাননা তাদের জ্ঞাতার্থে বলি, python এ যত বড়ই দাও না কেন কোন সমস্যা নেই!) আমরা যেই a বা b variable টা declare করেছি তা কিন্তু int টাইপ। একটি int টাইপ এর variable -2^{31} থেকে 2^{31}পর্যন্ত মান এর হিসাব নিকাশ করতে পারে।[২] এর হিসাবের range হল -2^{63} হতে $2^{63} - 1$ পর্যন্ত। আমরা int টাইপ এর ডাটা ইনপুট আউটপুট এর জন্য যেমন %d ব্যবহার করতাম ঠিক তেমনি long long এর জন্য %lld ব্যবহার করতে হবে।[৩]

এখন তুমি প্রোগ্রামটাতে গুনের পরিবর্তে ভাগ বসিয়ে দাও আর 3 কে 2 দিয়ে ভাগ দাও। উত্তর তো তুমি জানই 1.5 কিন্তু তোমার প্রোগ্রাম কত বলে? নিশ্চয়ই 1? কি ভাবছ? কম্পিউটার ভুল করেছে? মোটেও না। এটা সবসময় মনে রাখবে, কম্পিউটার কখনও ভুল করে না। তাহলে কাহিনী কি? কাহিনী হল, যদি a ও b দুটিই int হয় তাহলে a/b হবে তাদের ভাগফলের integer অংশ। তাহলের প্রশ্ন

[১] $a\%b$ এর মান হল a কে b দিয়ে ভাগ করলে যত ভাগশেষ থাকবে তত।

[২] তোমাদের কল্পনার সুবিধার্তে বলে রাখি $2^{31} \approx 2 * 10^9$. তুমি যদি আরেকটু বড় সংখ্যার হিসাব নিকাশ করতে চাও তাহলে int এর পরিবর্তে long long ব্যবহার করতে পার।

[৩] অনেক সময় দেখবে long long বলে কিছু নেই। তখন তোমাদের __int64 ব্যবহার করতে হবে এবং এর সাথে %I64d ব্যবহার করতে হবে।

আসতে পারে 1.5 কেমনে পাব? তুমি যদি দশমিকের সংখ্যা ব্যবহার করতে চাও তাহলে তোমাকে double বা float ব্যবহার করতে হবে।[১] দুরকম ডাটা টাইপ থাকার কারণ হল সেই int আর long long থাকার মত। float এর range কম, double এর range বেশি। তবে আমরা সবসময় বলে থাকি যে, float কখনও যেন ব্যবহার না কর। কারণ এর precision অনেক কম।[২] কিন্তু তাই বলে আবার int ব্যবহার না করে সবসময় long long ব্যবহার কর না। কারণ int এর তুলনায় long long বেশ slow. তোমরা প্রবলেম সল্ভিং এ একটু অভ্যস্ত হয়ে গেলে signed ও unsigned ডাটা টাইপ সম্পর্কেও জেনে রাখবে। কারণ প্রবলেম সল্ভিং এ মাঝে মাঝেই এর দরকার পরে।

আমরা এখন পর্যন্ত একটাই হেডার ফাইল দেখেছি (stdio.h). আরও একটি হেডার ফাইল দেখা যাক, এটা হল math.h হেডার ফাইল। নাম দেখেই বুঝা যাচ্ছে এখানে math সংক্রান্ত কিছু ফাংশন দেয়া আছে। আমাদের ক্যালকুলেটর এ যেমন অনেক ফাংশন আছে (যেমন sin, cos, tan, square root, square, cube ইত্যাদি) ঠিক তেমনি এই math.h হেডার ফাইল এ এধরনের বেশ কিছু ফাংশন আছে। টেবিল নং ২.১ তে math.h এর কিছু গুরুত্বপূর্ণ ফাংশন দেয়া আছে।

সারনী ২.১: math.h এর কিছু ফাংশনের তালিকা

sqrt(x)	এটা x এর square root নির্ণয় করে। x কে অবশ্যই অঋণাত্মক হতে হবে। নাহলে Run Time Error (RTE) হবে
fabs(x)	এটা x এর পরম মান নির্ণয় করে
sin(x), cos(x), tan(x)	x এর sin, cos, tan নির্ণয় করে থাকে। এখানে x কে radian এককে দিতে হবে।
asin(x), acos(x), atan(x)	x এর $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$ নির্ণয় করে থাকে। এখানে x কে অবশ্যই $[-1, 1]$ range এ হতে হবে।
atan(x), atan2(y, x)	যেহেতু $\Delta y = 1, \Delta x = 0$ হলে আমাদের $\tan^{-1}$ এর মান $atan$ ফাংশন দিয়ে বের করা যায় না, সেক্ষেত্রে atan2 ব্যবহার করা হয়।
pow(x, y)	এটি x^y নির্ণয় করে থাকে।
exp(x)	এটি e^x নির্ণয় করে।
log(x), log10(x)	এখানে log হল natural logarithm আর log10 হল 10 based logarithm.
floor(x), ceil(x)	যথাক্রমে floor এবং ceiling দেয়।

কোড ২.3 তে আমরা একটি বৃত্তের ক্ষেত্রফল নির্ণয় করার কোড দেয়া হল। এখানে খেয়াল কর আমরা ৯ নাম্বার লাইন এ কেমনে π এর মান নির্ণয় করলাম। আমরা জানি যে, $\cos \pi = -1$ সুতরাং acos(-1) হল π.[৩]

Listing ২.3 : circle area.cpp

```
1  #include<stdio.h>
2  #include<math.h>
3
4  int main()
5  {
6          double r, area, pi;
7
8          scanf("%lf", &r);
9
10         pi = acos(-1.);
11         area = pi * r * r;
12
13         printf("%lf\n", area);
```

[১] double ও float এ ইনপুট আউটপুট এর জন্য যথাক্রমে %lf ও %f ব্যবহার করা হয়

[২] তোমরা যদি double, float এসবের precision সম্পর্কে আরও জানতে চাও তাহলে http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=integersReals এই article পড়ে দেখতে পার।

[৩] অনেকের ভুল ধারনা আছে যে $\pi = \frac{22}{7}$, এটা কেবল মাত্র একটি approximation তুমি এই মান দিয়ে হিসাব করলে কখনই সঠিক উত্তর পাবে না।

```
14
15          return 0;
16  }
```

প্র্যাকটিস প্রবলেম

- দুইটি 2D বিন্দুর co-ordinate ইনপুট নিয়ে তাদের মাঝের দুরত্ব প্রিন্ট কর।
- একটি ত্রিভুজের তিনটি বাহুর দৈর্ঘ্য দেয়া আছে, তার তিনটি কোণ নির্ণয় কর।
- একটি ত্রিভুজের তিনটি বাহুর দৈর্ঘ্য দেয়া আছে, তার ক্ষেত্রফল নির্ণয় কর।
- একটি বৃত্তের ব্যাসার্ধ দেয়া আছে, পরিসীমা নির্ণয় কর।
- কোন একটি সংখ্যার বর্গমূলের কাছের integer সংখ্যাটা প্রিন্ট কর।
- একটি ত্রিভুজের vertex গুলোর co-ordinate দেয়া আছে, ক্ষেত্রফল প্রিন্ট কর।

২.৩ if - else if - else

এতক্ষণ আমরা যা করলাম তা কিন্তু একটা সাধারন ক্যালকুলেটর দিয়েও করা যায়। কিন্তু আমরা একটা ক্যালকুলেটরে কিন্তু লজিকাল কাজ করতে পারি না। যেমন আমরা যদি একটা সাল লিখি যেমন ধর ২০০৪ আমাদের ক্যালকুলেটর কিন্তু বলতে পারবে না যে সেটা leap year কিনা। একটি সাল তখনি leap year হবে যদি সেটা ৪০০ দ্বারা বিভাজ্য হয় বা ৪ দ্বারা ভাগ যায় কিন্তু ১০০ দ্বারা না যায়। সামনে এগুনোর আগে এই কথাটুকুর মানে ভালো করে চিন্তা করে নাও। এখানে এই হিসাবটা কিন্তু তুমি ক্যালকুলেটর এ করতে পারবে না। হ্যাঁ হয়তো তুমি এটা দেখতে পার যে সংখ্যাটা ৪০০ দ্বারা ভাগ যায় কিনা অথবা ১০০ বা ৪ দ্বারা ভাগ যায় কিনা এর পর তুমি নিজে সিদ্ধান্ত নিবে যে বছরটি leap year কিনা। কিন্তু তুমি একটা প্রোগ্রাম এর সাহায্যে সহজেই leap year নির্ণয় এর প্রোগ্রাম লিখতে পারবে। এর জন্য যা প্রয়োজন তাহলো if-else if-else. এই জিনিসটা কেমনে লিখতে হয় টা কোড ২.4 এ দেখানো হল। এটা কোন সঠিক কোড নয়, এখানে শুধুমাত্র syntax টা দেখানো হল। [১]

Listing ২.4 : simple if

```
1   if( condition1 )
2   {
3           //if condition1 is true
4   }
5   else if( condition2 )
6   {
7           //otherwise if condition2 is true
8   }
9   ...
10  else
11  {
12          //if no conditions are true
13  }
```

এখন প্রশ্ন হল আমরা condition লিখব কেমনে? আমাদের যেমন +, -, *, / চিহ্ন আছে (এদেরকে arithmetic operator বলা হয়) ঠিক তেমনি কিছু comparison operator আছে। যেমনঃ $<$, $>$, $<=$, $>=$, $==$(equal), ! $=$(not equal). আমরা একটা খুব সহজ কোড দেখি যা বলতে পারে যে ইনপুট সংখ্যাটা জোড় না বিজোড় (কোড ২.5

[১] syntax মানে হল লিখার নিয়ম। যেমন প্রায় প্রতিটি লাইন এর শেষে সেমিকলন দিতে হয়। এটাই syntax.

Listing ২.5 : odd even.cpp

```
#include<stdio.h>

int main()
{
        int a;
        scanf("%d", &a);

        if(a % 2 == 0)
        {
                printf("%d is even\n", a);
        }
        else
        {
                printf("%d is odd\n", a);
        }

        return 0;
}
```

তোমরা যদি একটু চিন্তা কর খুব সহজেই leap year এর জন্যও প্রোগ্রাম লিখে ফেলতে পারবে। আমরা প্রথমে দেখব সংখ্যা টা ৪০০ দ্বারা ভাগ যায় কিনা, না গেলে দেখব সংখ্যা টা ১০০ দ্বারা ভাগ যায় কিনা, তাহলে দেখব ৪ দ্বারা যায় কিনা। প্রোগ্রামটার কোড তোমাদের কোড ২.6 এ দেখানো হল।

Listing ২.6 : leap year.cpp

```
#include<stdio.h>

int main()
{
        int year;
        scanf("%d", &year);

        if(year % 400 == 0)
        {
                printf("%d is Leap Year\n", year);
        }
        else if(year % 100 == 0)
        {
                printf("%d is not Leap Year\n", year);
        }
        else if(year % 4 == 0)
        {
                printf("%d is Leap Year\n", year);
        }
        else
        {
                printf("%d is not Leap Year\n", year);
        }

        return 0;
}
```

তোমরা চাইলে কিন্তু এখানে curly brace () গুলো সরিয়ে ফেলতে পার। যদি কোন if / else if / else ব্লক এর ভিতরে কেবল মাত্র একটি লাইন থাকে তাহলে curly brace না দিলেও চলে। আরও একটি জিনিস, তাহলো তোমরা অর্থবোধক variable এর নাম ব্যবহার করলে দেখবে পরবর্তীতে তোমাদেরই কোডটা বুঝতে সুবিধা হবে।

leap year এর কোডটা কিন্তু অনেক বড়। আমরা চাইলেই এই কোডটাকে অনেক ছোট করে ফেলতে পারি। তবে এজন্য আমাদের জানতে হবে logical operator সম্পর্কে। logical operator

তিনটিঃ || (or), && (and), ! (not). দুইটা condition এর মাঝে || দিলে তাদের কোন একটি সত্য হলেই পুরটা সত্য হবে, && দিলে দুটো সত্য হলেই কেবল পুরোটা সত্য হবে আর ! কোন একটি condition এর সামনে দিলে ঐ condition মিথ্যে হলেই কেবল পুরোটা সত্য হবে। কোড ২.7 তে আমরা logical operator ব্যবহার করে কেমনে leap year নির্ণয় এর ছোট প্রোগ্রাম লিখা যায় তা দেখালাম। খেয়াল করলে দেখবে যে আমরা লজিকগুলোকে bracket বন্দি করেছি। জিনিসটা কিছুটা $5-(2+1)$ আর $5-2+1$ এর মত বা $5+2*1$ এর মতও মনে করতে পার। আমরা স্বভাবতই জানি যে গুন এর কাজ যোগ এর আগে হবে, কিন্তু আমরা যদি sure না হই তাহলে তাদের bracket বন্দি করে ফেলাই বুদ্ধিমানের মত কাজ।

Listing ২.7 : leap year2.cpp

```
#include<stdio.h>

int main()
{
        int year;
        scanf("%d", &year);

        if(year % 400 == 0 || (year % 100 != 0 && year % 4 == 0))
                printf("%d is Leap Year\n", year);
        else
                printf("%d is not Leap Year\n", year);

        return 0;
}
```

এখন তাহলে তোমার নিজের ইচ্ছামত কিছু লজিকাল প্রবলেম সল্ভ কর। নিচে কিছু প্র্যাকটিস প্রবলেম দেয়া হলঃ

প্র্যাকটিস প্রবলেম

- Palindrome হল সেই জিনিস যা সামনে থেকে পড়তেও যা, পিছন থেকে পড়তেও তা। যেমন কিছু Palindrome number হলঃ $1, 2, 3, \ldots 9, 11, 22, 33, \ldots 99, 101, 111, 121, \ldots$. তোমাকে n তম Palindrome Number প্রিন্ট করতে হবে। $(n < 100)$
- n এর মান দেয়া আছে, তোমাকে $\sum_{i=1}^{n} i*(n-i+1) = 1*n+2*(n-1)+\ldots n*1$ এর মান বের করতে হবে। (এটা কিন্তু if-else এর প্র্যাকটিস প্রবলেম!)
- দুইটি সংখ্যার মাঝে বড়টি প্রিন্ট কর। এর পরে তিনটি সংখ্যার জন্যও চেষ্টা করে দেখ।
- তিনটি সংখ্যা ইনপুট নিয়ে তাদের ছোট হতে বড় অনুসারে প্রিন্ট কর।
- একটি co-ordinate দেয়া আছে, তোমাকে বলতে হবে সেটা কোন quadrant এ পরে।
- Timus 1068

২.৪ Loop

Loop মানে হচ্ছে কোন একটা কাজ বার বার করা। যেমন ধর আমি চাইতেছি যে আমাদের এর আগের $a+b$ এর প্রোগ্রাম (কোড ২.2) টা বার বার চলুক। বা ধর আমরা চাইতেছি যে 1 থেকে 100 পর্যন্ত যোগ করতে চাই, অর্থাৎ প্রথমে আমি 1 যোগ করব, এর পর 2 , এর পর 3 এভাবে 100 পর্যন্ত। এইসব কাজ loop এর সাহায্যে করা হয়ে থাকে। C তে loop মুলত তিনটি। For loop, While loop, Do-While loop. আমরা আপাতত প্রথম দুইটি নিয়ে কথা বলব, পরবর্তী কোন এক অধ্যায় এ আমরা

তৃতীয়টির ব্যাপারে কথা বলব। আসলে সত্যি কথা বলতে, আমরা প্রথম দুটিই সাধারনত ব্যবহার করে থাকি। if-else এ যেমন একটি লাইন হলে curly brace দেয়ার দরকার হয়না এক্ষেত্রেও কিন্তু তাই। কোড ২.8 এ আমরা for loop ও while loop এর outline এবং কিছু উদাহরন দেখানো হয়েছে। আসলে তুমি জিনিসটা উদাহরন গুলো থেকে বুঝতে পারবে ভালো মত।[১] এখানের কোডটুকু কিন্তু main faunction এর ভিতরে রাখা হয় নাই, আমরা কোডকে সংক্ষিপ্ত রাখার জন্য এরকম করেছি।

Listing ২.8 : simple loop.cpp

```
1  //prototype(not syntactically valid line)
2  for(initialization ; condition ; increment/decrement) {}
3  while(condition is true) {}
4
5  //Examples
6  for(i = 1; i <= 10; i++) printf("%d\n", i); //prints from 1 to 10
7  for(i = 10; i >= 1; i--) printf("%d\n", i); //prints from 10 to 1
8  for(i = 1; i <= 10; i += 2) printf("%d\n", i); //prints odd numbers
       from 1 to 10
9
10 i = 5;
11 while(i <= 7) {printf("%d\n", i * 2); i++;} //prints 10, 12, 14
```

এটা জেনে রাখা ভালো যে for loop এর কোন জিনিসের পর কোন জিনিসের কাজ হয়। প্রথমে initialization হয়। এর পর condition যদি সত্যি হয় তাহলে সে ভিতরে ঢুকবে অন্যথা loop থেকে বেরিয়ে যাবে। যদি সত্য হয় তাহলে সে loop এর ভিতরে ঢুকবে। সব কাজ শেষে সে increment/decrement অংশে যাবে। এর পর আবার condition চেক করে আবার loop এর ভিতরে ঢুকবে বা বাইরে চলে যাবে। while loop সে তুলনায় অনেক সোজা। এটা condition চেক করবে, যদি সত্য হয় তাহলে ভিতরে ঢুকবে নাহলে বাইরে বেরিয়ে যাবে। যদি এটুকু বুঝে থাক তাহলে বলতো কোড ২.8 এর প্রতিটি loop এর শেষে i এর মান কত হয়? একটু চিন্তা করলে দেখবে, প্রথম for loop শেষে i এর মান 11, দ্বিতীয় for loop শেষে 0, তৃতীয় for loop শেষে 11 এবং while loop শেষে 8. loop ব্যবহার করে আরও কিছু উদাহরন কোড ২.9 এ দেখানো হল।

Listing ২.9 : simple loop2.cpp

```
1  //counts how many 2 divides 100
2  x = 100;
3  cnt = 0;
4  while(x % 2 == 0)
5  {
6          x = x / 2;
7          cnt++;
8  }
9
10 //finds out the highest number which is power of 2 and less than
       1000
11 x = 1;
12 while(x * 2 < 1000) x *= 2;
13
14 //same thing using for loop. Note a semicolon is after the for loop.
15 for(x = 1; x * 2 < 1000; x *= 2);
```

loopএর জন্য খুবই গুরুত্বপূর্ণ দুইটি keyword হলঃ break এবং continue. আমরা চাইলে যেকোনো সময় আমাদের loop ভেঙ্গে বের হয়ে যেতে পারি। আবার আমরা চাইলে যেকোনো সময় loop এর ভিতরে লিখা বাকি কাজ গুলি না করে loop এর পরবর্তী iteration এ চলে যেতে পারি। break ও continue সহ আরও কিছু উদাহরন তোমাদের কোড ২.10 এ দেখানো হল।

[১] i++ মানে হল i = i + 1 এবং i += 2 মানে হল i = i + 2. কোড এ থাকা double slash দ্বারা comment বুঝিয়ে থাকে। পরবর্তীতে কোড ভালো মতও বুঝার জন্য এভাবে comment করে রাখা হয়।

Listing ২.10 : simple loop3.cpp

```
//prints odd numbers from 1 to 10
for(i = 1; i <= 10; i++)
{
        if(i % 2 == 0) continue;
        printf("%d\n", i);
}

//prints only 1, 2 and 3
for(i = 1; i <= 10; i++)
{
        if(i > 3) break;
        printf("%d\n", i);
}

//takes input until the input is 0
//sometimes it is needed for OJs.
//EOF = End Of File.
while(scanf("%d", &a) != EOF)
{
        if(a == 0) break;
        printf("%d\n", a);
}

//in short
while(scanf("%d", &a) != EOF && a)
{
        printf("%d\n", a);
}
```

অনেক সময় আমাদের একটি loop এর ভিতরে আরেকটি loop লিখার প্রয়োজন হয়। যেমন, ধরা যাক আমাদের n দেয়া আছে আমাদের বের করতে হবেঃ $1 + (1 + 2) + (1 + 2 + 3) + \ldots + (1 + 2 + \ldots + n)$. এখন খেয়াল কর, আমাদের যদি শুধু $(1 + 2 + \ldots + n)$ বের করতে দেয়া হতো তাহলে কিন্তু কাজটা বেশ সহজ। একটা for loop 1 থেকে n পর্যন্ত চালিয়ে যোগফল বের করলেই হয়। কিন্তু এরকম 1 থেকে কত পর্যন্ত যোগ করতে হবে তাও কিন্তু এখানে পরিবর্তন হচ্ছে। প্রথমে 1 পর্যন্ত, এর পরে 2 পর্যন্ত এরকম করে শেষে n পর্যন্ত। সুতরাং আমরা যা করব তাহল একটা for loop দিয়ে আমরা upper bound টাকে বাড়াব আরেকটা for loop দিয়ে আমরা যোগ করব। [১] এই কোডটা কোড ২.11 এ দেখানো হলঃ

Listing ২.11 : simple loop4.cpp

```
sum = 0; //very important. many of you forgets to initialize
    variable
for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
                sum += j;
```

প্র্যাকটিস প্রবলেম

- নিচের সিরিজগুলো কোড লিখে সমাধান করঃ
  ১. $1 + 2 + 3 + \ldots + n$
  ২. $1^2 + 2^2 + 3^2 + \ldots + n^2$

[১] তোমরা চাইলে কিন্তু ভিতরের loop টার জায়গায় formula বসিয়ে দিতে পার, বা পুরো জিনিসটাই কিন্তু formula দিয়ে সল্ভ করা যায় :)

৩. $1^1 + 2^2 + 3^3 + \ldots + n^n$

৪. $1 + (2 + 3) + (4 + 5 + 6) + \ldots +$ nth term

৫. $1 - 2 + 3 - 4 + 5 \ldots$ nth term

৬. $1 + (2 + 3 * 4) + (5 + 6 * 7 + 8 * 9 * 10) + \ldots +$ nth term

৭. $1 * n + 2 * (n - 1) + \ldots + n * 1$

- n ইনপুট এর জন্য চিত্র ৩.১ এর পিরামিড গুলি প্রিন্ট করার প্রোগ্রাম লিখ।

```
*..      ***      ..*..      12321
**.      .**      .***.      .121.
***      ..*      *****      ..1..
```

```
..*..      ..1..
.***.      .121.
*****      12321
.***.      .121.
..*..      ..1..
```

নকশা ২.১: কিছু পিরামিড $n = 3$ এর জন্য

- Palindrome হল সেই জিনিস যা সামনে থেকে পড়তেও যা, পিছন থেকে পড়তেও তা। যেমন কিছু Palindrome number হলঃ $1, 2, 3, \ldots 9, 11, 22, 33, \ldots 99, 101, 111, 121, \ldots$. তোমাকে n তম Palindrome Number প্রিন্ট করতে হবে। $(n < 10^9)$ (এই সমস্যাটা আগের সেকশনে ছিল তবে কম মানের জন্য)

- কোন একটি সংখ্যা n Prime হবে যদি সেটি 1 থেকে বড় হয় এবং 1 ba n ছাড়া আর কোন ধনাত্মক সংখ্যা দ্বারা বিভাজ্য না হয়। তোমাকে n দেয়া আছে বলতে হবে এটি Prime কি Prime নয়।

- $n!$ নির্ণয় কর।

- n ও r দেয়া আছে, তোমাকে $\binom{n}{r} = \frac{n!}{r!(n-r)!}$ প্রিন্ট করতে হবে।

- x ও n দেয়া আছে, তোমাকে cos x এর মান maclaurine series এর সাহায্যে বের করতে হবে। cos x এর series টি হচ্ছে $1 - \frac{x^2}{2!} + x^4 4! + \ldots +$ nth term

- কিছু OJ এর প্রবলেমঃ – Timus 1083 – Timus 1086 – Timus 1209 – LightOJ 1001 – LightOJ 1008[১] – LightOJ 1010[২] – LightOJ 1015 – LightOJ 1022 – LightOJ 1053 – LightOJ 1069 – LightOJ 1072 – LightOJ 1107 – LightOJ 1116 – LightOJ 1136[৩] – LightOJ 1182 – LightOJ 1202 – LightOJ

[১] formula বের কর। তোমাকে quadratic equation বের করতে হতে পারে।

[২] pattern বের কর

[৩] A হতে B এর answer বের না করে 0 হতে B এর answer থেকে 0 থেকে $A - 1$ এর answer বিয়োগ করলে জিনিসটা সোজা হয়।

1211[১] – LightOJ 1216[২] – LightOJ 1294 – LightOJ 1305 – LightOJ 1311 – LightOJ 1331 – LightOJ 1433

– Last but not the least UVa 100

২.৫ Array ও String

ধর একটি Game Show তে 10 জন প্রতিযোগী আছে। Host একটি করে প্রশ্ন করে, প্রতিযোগীদের বাজার টিপে answer করতে হবে। answer ঠিক থাকলে 1 পয়েন্ট করে পাবে। Game শেষে যার পয়েন্ট সবচেয়ে বেশি সে জয়ী। একাধিক জনও বিজয়ী হতে পারে। এখন তোমাকে এর জন্য একটি প্রোগ্রাম লিখতে বলা হল। তুমি কি করবে? সবার পয়েন্ট এর হিসাব রাখার জন্য আলাদা আলাদা 10টি variable রাখবে, ধর variable গুলো হলঃ $a, b, \ldots j$। এর পর তোমাকে যদি বলা হয় প্রথম প্রতিযোগী সঠিক উত্তর দিয়েছে তাহলে তুমি a এর মান এক বাড়াবে, এরকম করে যেই প্রতিযোগী ঠিক উত্তর দিবে তার পয়েন্ট তুমি বাড়াবে। কিন্তু এই জিনিস কিন্তু অনেক ঝামেলার। কারণ, তোমাকে 10টা if-else লাগিয়ে চেক করতে হবে যে কোন variable এর মান তুমি বাড়াবা। আবার game শেষে তোমাকে অনেক গুলো if-else দিয়ে বের করতে হবে যে কে বা কারা কারা জয়ী (যদি আমার হিসাব ভুল না যায়, তোমাকে 20টা if-else লাগাতে হবে)। এখন যদি তোমার প্রতিযোগী সংখ্যা আরও বাড়ে তাহলে?

এই অসুবিধাকে দূর করার জন্যই আমাদের কাছে Array নামক জিনিসটা আছে। Array আর কিছুই না, অনেকগুলো variable এর সমন্বয়। আমরা যদি বলি int a[10] এর মানে int টাইপ এর 10টি variable তৈরি হয়ে যাবে। এদের নাম হবেঃ $a[0], a[1], \ldots a[9]$. মনে কর তোমাকে বলল যে প্রথম প্রতিযোগী $id = 1$ একটি সঠিক উত্তর দিয়েছে তাহলে কিন্তু $a[id - 1] + +$ করলেই প্রথম প্রতিযোগী এর variable এর মান এক বেড়ে যাবে।[৩] এখন সবার শেষে আসে maximum বের করার কাজ, চিন্তা করলে দেখবে একটি for loop এর সাহায্যে খুব সহজেই maximum সঠিক উত্তরটা পেয়ে যাবে। যেমনঃ 10 জন প্রতিযোগী এবং 100টি প্রশ্নের খেলায় বিজয়ী নির্ণয়ের প্রোগ্রামটির কোড ২.12।

Listing ২.12 : simple array.cpp

```
1  //initialization
2  for(i = 0; i < 10; i++) a[i] = 0;
3
4  for(i = 0; i < 100; i++)
5  {
6          scanf("%d", &id); //the player giving correct answer
7          a[id - 1]++; //increment players point
8  }
9
10 maximum_score = 0; //initializing max score
11 for(i = 0; i < 10; i++)
12         if(maximum_score < a[i]) //if ith players score is more than
               the max
13                 maximum_score = a[i]; //set max score to this value
14
15 printf("Winners are:\n");
16 for(i = 0; i < 10; i++)
17         if(maximum_score == a[i]) //if ith players score is maximum
```

[১] এই সমস্যার একটি সুন্দর solution আছে। আমি তোমাকে 2d এর জন্য সমাধান বলি। খেয়াল করলে দেখবে, দুইটি আয়তক্ষেত্রের intersection ও কিন্তু আরেকটি আয়তখেত্র। আমরা যদি এই আয়তক্ষেত্রের দুইটি কর্ণবিন্দু বের করতে পারি তাহলেই হয়ে যাবে। খেয়াল করলে দেখবে, এর নিচের বাম কোনার x হবে মুল আয়তক্ষেত্র দুটির নিচের বাম কোনার x এর যেটি বড় সেটি। একই ভাবে অন্যগুলিও বের করে ফেল। যদি আয়তক্ষেত্র দুটি intersect না করে তাহলেও কিন্তু তুমি এই দুই কর্ণের co-ordinate দেখে বুঝতে পারবে। নিজে করে দেখ মজা পাবে।

[২] formula টা বের করার জন্য কিন্তু তোমার internet এর সাহায্য নেবার দরকার নেই!

[৩] 0-indexing আর 1-indexing এর ব্যাপারটা খেয়াল রাখবে

```
18 printf("%d\n", i + 1); //print his id
```

আমরা এতক্ষণ শুধু int ও double টাইপ variable নিয়েই কাজ করেছি। কিন্তু যদি কারো নাম, বা শহরের নাম এসব নিয়ে কাজ করতে চাই তার জন্য কিন্তু আলাদা variable type আছে আর তাহল char. একটি char কেবল মাত্র একটি character রাখতে পারে। একটি নাম কিন্তু অনেক গুলো character এর সমন্বয়ে তৈরি হয়। যেমন, Rajshahi এখানে 8টি character আছে। সেজন্য আসলে কোন নাম বা string সরক্ষনের জন্য আমাদের char এর array ব্যবহার করতে হবে। যেমন, আমরা যদি একটি char city[10] নামে একটি array declare করি, এবং তাতে Rajshahi রাখি তাহলে $city[0] = R, city[1] = a, \ldots city[7] = i$. সবই ঠিক আছে তবে এর সাথে অতিরিক্ত একটি জিনিস থাকে তাহল null. city[8] এ এই null থাকে। null দেখে আমরা বুঝতে পারি যে শব্দটা আসলে city[0] থেকে শুরু করে কোন পর্যন্ত আছে। যখন আমরা city array টা প্রিন্ট করব তখন সে city[0] থেকে প্রিন্ট করা শুরু করবে যতক্ষণ না city[8] এ এসে null পায়। আমরা যদি city array তে রাখা নামটা প্রিন্ট করতে চাই তাহলে আমাদের লিখতে হবেঃ printf("%s", city) আর যদি আমরা কোন শহরের নাম ইনপুট নিতে চাই তাহলে আমাদের লিখতে হবেঃ scanf("%s", city). খেয়াল কর এখন আর আগের মত ইনপুট এর সময় & ব্যবহার করতে হচ্ছে না। তোমরা চাইলে এই null নিয়ে খেলা করতে পার, যেমন for loop চালিয়ে string এর length বের করা। বা একটি শহরের নাম ইনপুট নিয়ে তার প্রথম 3 অক্ষর কে প্রিন্ট করা (city[3] = 0; printf("%s", city);).

আমরা যদিও বলছি যে city[0] এ R আছে কিন্তু আসলে তা নেই। city[0] এ আছে 82. প্রতিটি character এর বদলে একটি করে value থাকে, একে বলা হয় ASCII value. www.lookuptables.com এ ascii value এর একটি টেবিল দেয়া আছে।[১] খেয়াল করলে দেখবে A হতে Z পর্যন্ত ascii value গুলো পরপর আছে এবং এরা হল 65 হতে 90, a হতে z এর মান গুলো হচ্ছে 97 থেকে 122 আর 0 হতে 9 এর মান হচ্ছে 48 হতে 57. মজার ব্যাপার হচ্ছে আমাদের এই ascii value আসলে মুখস্ত করার দরকার নেই। আমাদের যদি নেহায়েত ই দরকার হয় তাহলে আমরা কোন character কে %d দিয়ে প্রিন্ট করলেই দেখতে পাব কোন character এর ascii value. আরও মজার ব্যাপার হচ্ছে আমাদের ascii value আসলে তেমন লাগেই না, বরং A হতে Z, a হতে z বা 0 হতে 9 যে পর পর আছে এটুকু জানলেই আমরা অনেক কিছু করে ফেলতে পারি। যেমন আমরা যদি জানতে চাই যে কোন একটি character ধরা যাক ch বড় হাতের না ছোট হাতের, তাহলে আমরা কোড লিখবঃ if('a' <= ch && ch <= 'z'). (single quotation এর মাঝে কোন character রাখলে তার ascii value পাওয়া যায়।) যদি condition টি সত্য হয় তাহলে আমরা বুঝে যাব যে এটি ছোট হাতের। আবার ধরা যাক, আমরা যদি জানি যে, ch ছোট হাতের এবং আমরা চাই যে একে বড় হাতের বানাতে হবে আমরা শুধু লিখবঃ ch = ch - 'a' + 'A'. আবার আমরা ch এ থাকা digit কে একটা int variable এ মান হিসাবে নিতে চাই, তাহলে আমরা লিখবঃ d = ch - '0'. অর্থাৎ আমরা ascii value এর relative order দেখেই অনেক কঠিন কঠিন কাজ করে ফেলতে পারি।

String এর ইনপুট নিয়ে আরেকটু কথা বলা যাক। আমরা উপরে যে ভাবে scanf দিয়ে ইনপুট নিয়েছি তাতে একটা সীমাবদ্ধতা আছে আর তা হলঃ space যুক্ত string ইনপুট নেয়া যাবে না এভাবে। যেমন, আমরা যদি একটি sentence ইনপুট নিতে চাই "Facebook is a popular web media" এবং সেজন্য যদি scanf %s ব্যবহার করি তাহলে দেখব ঐ array তে কেবল Facebook শব্দটাই থাকবে। এর কারণ হল, আমরা যখন scanf দিয়ে পড়া শুরু করি তখন সে প্রথম non whitespace character খোঁজে, এবং ওখান থেকে সে পরবর্তী white character পর্যন্ত পড়ে।[২] তুমি যদি একটি space যুক্ত sentence পড়তে চাও তাহলে এভাবে পড়তে হবেঃ gets(s). এখানে s হল আমাদের char array এর নাম। gets প্রথম থেকে শুরু করে যতক্ষণ না একটি new line পাচ্ছে ততক্ষণ পড়তে থাকে। এখন এর ফলে, তুমি যদি একটি কোডে scanf এবং gets দুটিই একই সাথে ব্যবহার করতে চাও, তখন তোমাকে সাবধান হতে হবে। ধর তোমার প্রোগ্রাম প্রথমে n ইনপুট নিবে যেটা হচ্ছে মানুষের সংখ্যা। এর পরে nটা sentence ইনপুট নিতে হবে। তুমি মনে কর n ইনপুট নিলে scanf দিয়ে আর পরের sentence গুলি ইনপুট নিলে gets দিয়ে, তাহলে তোমার প্রথম sentence টা ঠিক মত ইনপুট হবে না। কারণ, scanf দিয়ে তুমি যখন n পড়েছ তখন সে n পড়া শেষ করেছে যখন একটি new line বা white character পেয়েছে এবং সে সেটা পড়ে নাই। এখন তুমি যদি gets

[১] copyright এর জন্য টেবিলটা এখানে কপি করা হল না।
[২] space, new line, tab এগুল হল white character

দিয়ে ইনপুট নাও তাহলে প্রথমেই সে ঐ new line পড়বে এবং ইনপুট পড়া শেষ করে দিবে। সুতরাং এক্ষেত্রে সমাধান হল তুমি একটি dummy gets ব্যবহার করবা। অর্থাৎ, তুমি এমনি এমনি scanf এর পরে একটি gets ব্যবহার করবা।

আচ্ছা এইযে তোমাদের বলা হল, nটা sentence পড়তে হবে। তোমরা কি ভেবে দেখছ এত গুলো sentence কেমনে রাখবে? খেয়াল কর, তোমার প্রত্যেক sentence এর জন্য কিন্তু একটি array দরকার। তাহলে n টি sentence এর জন্য কি করবে? sentence1[100], sentence2[100] এভাবে 100টি array declare করবা? মোটেও না, যা করবা তাহল array এর array! অর্থাৎ, sentence[20][100] এভাবে। এখানে sentence[0] এ থাকবে প্রথম sentence এরকম করে মোট 20টি sentence এখানে রাখা যাবে। আসা করি বুঝতেই পারছ, এটা শুধু string এর জন্য না, int, double সব ক্ষেত্রেই এরকম করে dimension বাড়ানো যাবে, একে multidimension array বলে। যেমন, তোমরা matrix জন্য 2 dimension array ব্যবহার করতে পার।

শেষ করব আরও একটি হেডার ফাইল এর কথা দিয়ে। string.h- string সম্পর্কিত ফাংশনগুলি এখানে আছে। এর কিছু গুরুত্বপূর্ণ ফাংশনগুলি হলঃ strlen - কোন একটি string এর length দেয়, strcmp - দুইটি string দিলে সে বলে দেয় কোনটি dictionary তে আগে আসবে, একে lexicographical order বলে, strcat - string concatenation এর জন্য, strcpy - string copy এর জন্য, memset - memory কে কোন নির্দিষ্ট কিছু দিয়ে fill করার জন্য ইত্যাদি। এছাড়াও তোমরা strtok, strstr এই ফাংশন দুটির ব্যবহার দেখতে পার।

মোটামোটি এই সব ফাংশনগুলিই intuitive. তবে memset এ কিছু critical ব্যপার আছে। মনে কর আমাদের কাছে a নামে একটা integer এর array আছে, আমরা এর সবগুল element কে 0দ্বারা initialize করতে চাই। তাহলে লিখতে হবেঃ memset(a, 0, sizeof(a)). তোমরা a এর জায়গায় তোমাদের array এর নাম দিবে শুধু, আর 0 er স্থানে তোমরা যেই মান দিয়ে fill করতে চাও তা। কিন্তু আসলে, সব মান এর জন্য এটা সঠিক ভাবে কাজ করবে না। আমরা সাধারনত কোন একটি array কে 0 বা -1 দ্বারা initialize করে থাকি, এই দুই ক্ষেত্রে আমাদের memset ঠিক মত কাজ করবে, কিন্তু আমরা যদি 1 দ্বারা initialize করতে চাই, তাহলে হবে না। [১]

প্র্যাকটিস প্রবলেম

- একটি দশমিক সংখ্যাকে বাইনারীতে convert কর।
- একটি array তে বাইনারী সংখ্যা দেয়া আছে, এর দশমিক মান বের কর।
- একটি array তে অনেক গুলি সংখ্যা দেয়া আছে তাদেরকে ছোট হতে বড় অনুসারে সাজাও। এভাবে সংখ্যাকে ছোট হতে বড় বা বড় হতে ছোট আকারে সাজানোকে sorting বলে।[২]
- একটি array তে 1এবং 0 আছে। তোমাকে বলতে হবে সবচেয়ে বেশি কতগুলো 1 পরপর আছে।
- একটি array আছে যার নাম number. তোমাকে অনেকগুল প্রশ্ন করা হবে। প্রতিটি প্রশ্ন হবে এমনঃ array এর iতম স্থান হতে j তম স্থানের যোগফল কত? যেহেতু প্রশ্ন অনেকগুলি তোমাকে উত্তর ও দ্রুত দিতে হবে।[৩]
- একটি string এর length বের কর। (library function ব্যবহার করে এবং না করে)

[১] তোমরা যদি এই বিষয়ে আরও জানতে চাও তাহলে Topcoder এর http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=integersReals আর্টিকেলটা পড়ে দেখতে পার।

[২] তোমরা দুইটি for loop ব্যবহার করে খুব সহজেই sort করতে পার। প্রথমে array এর প্রথম স্থানে সবচেয়ে ছোট সংখ্যাটি নিয়ে আসো, এর পরে দ্বিতীয় স্থানে দ্বিতীয় ছোট সংখ্যাটি আনো এভাবে। এখন খেয়াল কর, যখন একটা জায়গায় একটা সংখ্যা আনবে তখন ঐ সংখ্যাটা যেন হারিয়ে না যায়, সেজন্য চাইলে তুমি যেখান থেকে সংখ্যাটি আনবে সেখানে গিয়ে এই সংখ্যাটি রেখে আসবে। একে swap করা বলে।

[৩] তোমরা যদি মনে কর এ আর এমন কি! আমি প্রতিবার i হতে j পর্যন্ত for loop চালাব! না, এর থেকেও ভালো বুদ্ধি আছে, দেখো বের করতে পার কিনা!

- একটি শব্দে ছোট হাতের অক্ষর $(a, b, \ldots z)$ এবং বড় হাতের অক্ষর $(A, B, \ldots Z)$ এর সংখ্যা নির্ণয় কর।
- দুইটি string জোড়া লাগাও। অর্থাৎ একটি string যদি হয় water এবং অপর আরেকটি string যদি হয় melon তাহলে তাদের জোড়া লাগিয়ে নতুন string- watermelon বানাও।
- দুইটি string A এবং B দেওয়া আছে, বলতে হবে B সম্পূর্ণ ভাবে A এর ভিতরে আছে কিনা। যেমনঃ $A = bangladesh$ এবং $B = desh$ হলে বলা যায় যে B, A এর ভিতরে আছে। B Aএর ভিতরে কয়বার আছে তাও নির্ণয় কর। যেমনঃ aa শব্দটি aaa এর মাঝে দুইবার আছে।
- একটি sentence এ word এর সংখ্যা নির্ণয় কর। word গুলি একাধিক space দ্বারা আলাদা করা থাকতে পারে।
- দুইটি string দেয়া আছে বলতে হবে কোনটি lexicographically smaller. (library function ব্যবহার করে এবং না করে)
- আমি একটি তারিখ 21/9/2013 এরকম format এ দেব। তোমাকে এই string হতে দিন, মাস ও তারিখ আলাদা করতে হবে এবং তিনটি int variable এ রাখতে হবে।
- দুইটি string A এবং B দেওয়া আছে, বলতে হবে B A এর subsequence কিনা। B A এর subsequence হবে যদি A থেকে কিছু letter মুছে ফেললে B পাওয়া যায়। যেমনঃ $A = bangladesh$ এবং $B = bash$ হলে বলা যায় যে B, A subsequence কিন্তু $B = dash$ কিন্তু subsequence হবে না।
- কিছু OJ এর প্রবলেমঃ – Timus 1001 – Timus 1014 – Timus 1020 – Timus 1025 – Timus 1044 – Timus 1079 – Timus 1197 – Timus 1313 – Timus 1319 – LightOJ 1006 – LightOJ 1045 – LightOJ 1109 – LightOJ 1113 – LightOJ 1133 – LightOJ 1214 – LightOJ 1225 – LightOJ 1227 – LightOJ 1241 – LightOJ 1249 – LightOJ 1261 – LightOJ 1338 – LightOJ 1354 – LightOJ 1387 – LightOJ 1414

২.৬ Time এবং Memory Complexity

প্রোগ্রামিং প্রতিযোগিতায় Time এবং Memory Complexity খুবই গুরুত্বপূর্ণ জিনিস। সহজ কো-থায় বলতে গেলে একটি প্রোগ্রাম যতখানি Time নেয় বা যতখানি Memory নেয় তাকেই Time Complexity বা Memory Complexity বলে। তবে এই Time বা Memory কিন্তু second বা Byte এ মাপা হয় না। কেন?

খেয়াল করলে দেখবে দুইবছর আগে বাজারে যত ভালো computer পাওয়া যেতো এখন তার থেকে ঢের ভালো ক্ষমতার computer পাওয়া যায়। আগে যেই প্রোগ্রাম চলতে হয়তো 10s সময় নিত এখন সেই একই প্রোগ্রাম হয়তো 5s সময় নেয়। তোমরা হয়তো বলতে পার, কোন একটি প্রোগ্রাম আগে যেই পরিমাণ Memory নিত এখনও তাই নেয়। ঠিক! কিন্তু একটি প্রোগ্রাম ধর $n = 100$ এর জন্য যে পরিমাণ Memory বা Time ব্যয় করে $n = 1000$ এর জন্য হয়তো অন্য পরিমাণ Memory বা Time ব্যয় করে। এজন্য একটি solution কি পরিমাণ Time বা Memory নেয় তা আমরা সেই problem এর বিভিন্ন parameter এর উপর ভিত্তি করে হিসাব করে থাকি। এর মাধ্যমে আমরা exactly কি পরিমাণ Time বা Memory নেয় তা বের করি না, বরং একই সমস্যার একাধিক solution এর মাঝে তুলনা করার জন্য আমরা এই জিনিস ব্যবহার করে থাকি। কিছু উদাহরনে আসা করি জিনিসটা আরও বেশি পরিস্কার হবে।

আমরা loop এর সেকশনে একটি প্রবলেম নিয়ে কথা বলছিলামঃ $1 + (1 + 2) + (1 + 2 + 3) + \ldots + (1 + 2 + \ldots + n)$. এখন তুমি যদি দুইটি for loop চালিয়ে হিসাব কর, তাহলে তুমি সর্বমোট $1 + 2 + \ldots n = \frac{n^2+n}{2}$ টা যোগ করবে। আমরা আমাদের বুঝার সুবিধার্থে শুধু মাত্র n এর

সবচেয়ে বড় term টা বিবেচনা করি। এখানে n এর দুইটি term আছে, একটি n এবং আরেকটি n^2 এর term. আমরা শুধু n^2 এর term বিবেচনা করব, সুতরাং n এর term বাদ দিলে আমাদের থাকেঃ $n^2/2$. আমরা constant term ও বাদ দেই। সুতরাং আমরা যা পেলাম তাহলঃ n^2. এটাই আমাদের Time Complexity. আমরা বলে থাকি, আমাদের এই algorithm টি $O(n^2)$ [১]. যদি $n = 1000$ হয়ে থাকে তাহলে, $O(n^2) = 10^6$ খুব আরামে 1s এ চলবে, কিন্তু যদি $n = 10^6$ হয়? তাহলে এই কোড এক ঘণ্টাতেও শেষ হবে কিনা সন্দেহ আছে। তাহলে n এর মান বেশি হলে $O(n^2)$ algorithm এ Time Limit Exceed (TLE) পাবা। আমরা কি এটাকে optimize করতে পারি? খেয়াল করলে দেখবে যে, আমরা i এর for loop 1 থেকে n পর্যন্ত যদি চালাই এবং প্রতিবার $1 + 2 + \ldots + i$ এর মান আরও একটি for loop দিয়ে না বের করে যদি formula এর সাহায্যে বের করি ($\frac{i^2+i}{2}$) তাহলে আমাদের হিসাব $O(n)$ এ নেমে আসবে। কিন্তু যদি আমাদের n এর মান আরও বেশি হয়? ধর, $n = 10^12$? তাহলে কিন্তু সেই আগের মত অনেক সময় লাগবে এই কোড চলতে। সেক্ষেত্রে আমাদের চেষ্টা করতে হবে algorithm এর order আরও কমানো যায় কিনা। এবং আসলেই কমানো যায়ঃ

$$
\begin{aligned}
&1 + (1 + 2) + (1 + 2 + 3) + \ldots + (1 + 2 + \ldots + n) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} j \\
&= \sum_{i=1}^{n} \frac{i^2 + i}{2} \\
&= \frac{1}{2} \left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \right) \\
&= \frac{1}{2} \left(\frac{n(n+1)(2n+1)}{6} + \frac{n^2 + n}{2} \right)
\end{aligned}
$$

অর্থাৎ আমরা এমন একটি formula বের করেছি যা হিসাব করতে আমাদের কোন loop লাগেনা। শুধু কিছু যোগ বিয়োগ গুন করেই করে ফেলতে পারি, একে বলা হয় $O(1)$ algorithm.

এখন আসা যাক Memory Complexity তে। তোমরা আসা করি ফিবনাচি নাম্বার(Fibonacci Number) এর কথা শুনেছ। যারা শুন নাই তাদের জন্য বলি, nতম ফিবনাচি নাম্বার কে F_n দ্বারা প্রকাশ করা হয়। এর মানঃ

$$
F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}
$$

আমরা একটি array এর সাহায্যে খুব সহজেই এর কোডটা করতে পারি। F[0] = 0, F[1] = 1 এবং একটি for loop চালিয়ে F[i] = F[i - 1] + F[i - 2]. কিন্তু এখানে আমরা n সাইজ এর একটি array declare করছি। সুতরাং আমাদের Memory Complexity হল $O(n)$. কিন্তু আমরা কিন্তু খুব সহজেই কোন array ছাড়াই F_n হিসাব করে ফেলতে পারি। কোড ২.13 দেখলে বুঝবে আমরা মাত্র 3টি variable ব্যবহার করছি, সুতরাং আমাদের Memory Complexity $O(1)$.[২] এখানে যদিও Time Complexity আগের মত $O(n)$ ই রয়ে গেছে কিন্তু Memory Complexity $O(1)$ এ কমে এসেছে।[৩]

[১] এর উচ্চারণ order n^2

[২] $n = 0$ এ সঠিক উত্তর দিবে না।

[৩] তোমরা চেষ্টা করে দেখতে পার Time Complexity কেও কমাতে পার কিনা। পরবর্তী কোন অধ্যায়ে আমরা Time Complexity কে কমাবো।

Listing ২.13 : fibonacci.cpp

```
a = 0;
b = 1;
for(i = 2; i <= n; i++)
{
        c = a + b;
        a = b;
        b = c;
}

printf("nth Fibonacci = %d\n", b);
```

২.৭ Function এবং Recursion

ফাংশনকে তোমরা একটা ফ্যাক্টরি হিসাবে কল্পনা করতে পার। একে বিভিন্ন কাঁচামাল দিবে ভিতরে ভিতরে সে কিছু একটা করবে এবং কাজ শেষে তুমি তার ফলাফল পাবে। যেমন একটা জুস ফ্যাক্টরিতে তুমি ফল দিবে, চিনি দিবে, পানি দিবে আরও নানা কিছু উপকরন হিসাবে দিবে। তোমার জানার দরকার নেই ফ্যাক্টরি এর ভিতরে কেমনে কি হচ্ছে। সেটা ফ্যাক্টরি যে চালায় তার কাজ। সে হয়তো ফ্যাক্টরি কে এমন ভাবে বানিয়েছে যে, একটা মেশিন আছে যে ফল এর খোসা ছাড়াবে, আরেক মেশিন ফল হতে রস বের করবে, এক মেশিন তাতে পানি আর চিনি সঠিক পরিমানে মিশিয়ে জুস বানাবে, আরেক মেশিন সেই জুস গুলোকে পরিমাণ মত করে প্যাকাট এ ভরবে, এর পর আরেক মেশিন হয়তো প্যাকাট এর গায়ে স্ট্র লাগিয়ে দিবে। ব্যাস তোমার জুস তৈরি! তুমি এখন সেই জুস এর প্যাকাট এনে খাওয়া শুরু করবে। তোমার কিন্তু জানার দরকার নেই ফ্যাক্টরি এর ভিতরে কেমনে কি হচ্ছে। একি ভাবে ফাংশন হচ্ছে এমন একটা জিনিস যাকে তুমি কিছু দিবে সে হিসাব নিকাশ করে তোমাকে বলে দিবে যে তোমার কাজের উত্তর কি! যেমন, আমরা sqrt ফাংশন ব্যবহার করেছি। একে আমরা একটা সংখ্যা দিচ্ছি বিনিময়ে সে আমাদের ঐ সংখ্যার বর্গমূল বলে দিচ্ছে। আমরা কিন্তু জানি না, ভিতরে ভিতরে সে কিভাবে এই বর্গমূল নির্ণয় করছে।

C তে ফাংশনের মুলত 4টি অংশ আছে। প্রথমত, ফাংশনের parameter, অর্থাৎ কাঁচামাল। আমরা ফাংশনকে কিছু মান দিব এবং বলব এই মান অনুসারে কাজ করতে। দ্বিতীয়ত, return type অর্থাৎ আমরা এই ফাংশন থেকে কি ধরনের জিনিস বের করব। তৃতীয়ত, প্রদত্ত parameter এর ভিত্তিতে processing করা এবং চতুর্থত processing এর ফলাফল return করা। যেমন মনে করি আমাদের বলা হল কিছু ছাত্রের Grade নির্ণয় করতে হবে। আমাদের তাদের নাম্বার দেয়া হবে, এই নাম্বার এর উপর ভিত্তি করে আমাদের grade নির্ণয় করতে হবে। আমরা তাহলে একটা function লিখব যা parameter হিসাবে marks নিবে। এবং if-else দিয়ে চেক করে সে ফলাফল হিসাবে grade পাঠিয়ে দিবে (কোড ২.14)।

Listing ২.14 : grade.cpp

```
int grade(int marks)
{
        if(marks >= 80) return 5;
        else if(marks >= 60) return 4;
        else if(marks >= 50) return 3;
        else if(marks >= 40) return 2;
        else if(marks >= 33) return 1;
        else return 0;
}
```

পূর্বে একটি প্র্যাকটিস প্রবলেম হিসাবে LightOJ 1136 এই প্রবলেমটি দেয়া হয়েছিলো এবং বলা হয়েছিল "A হতে B এর answer বের না করে 0 হতে B এর answer থেকে 0 থেকে $A-1$ এর answer বিয়োগ করলে জিনিসটা সোজা হয়"। আমরা একই ধরনের দুইটি জিনিস আলাদা আলাদা

করে হিসাব না করে বরং একটি ফাংশন f লিখতে পারি যার parameter হবে n এবং এই ফাংশনটি 0হতে n পর্যন্ত এর জন্য answer বের করে। সুতরাং আমাদের উত্তর হবেঃ $f(B) - f(A-1)$. অনেক সহজেই আমাদের প্রবলেমটা সমাধান হয়ে যায়!

এবার আসা যাক Recursion এ। কোন এক অজানা কারণে recursion কে অনেকেই ভয় পায়! Mr Edsger Dijkstra বলেছেনঃ

> I learned a second lesson in the 60s, when I taught a course on programming to sophomores and discovered to my surprise that 10% of my audience had the greatest difficulty in coping with the concept of recursive procedures. I was surprised because I knew that the concept of recursion was not difficult. Walking with my five-year old son through Eindhover, he suddenly said "Dad, not every boat has a life-boat, has it?" "How come?" I said. "Well, the life-boat could have a smaller life-boat, but then that would be without one." It turned out.

Recursion আসলে কিছুই না, এটি হল এমন একটি ফ্যাক্টরি যা তার processing এর জন্য নিজের জিনিসই ব্যবহার করে। যেমন ফিবনাচি নাম্বার এর ক্ষেত্রে, আমরা জানি, $F_n = F_{n-1} + F_{n-2}$. এখন আমরা যদি এমন একটি ফাংশন লিখি যেটা আমাদের nতম ফিবনাচি নাম্বার দেয়, এবং সে সেই ফাংশনের ভিতরে হিসাবের জন্য $n-1$তম ও $n-2$তম ফিবনাচি নাম্বার কে ব্যবহার করে এই ফাংশনকে call করে তাহলে একে recursion বলা হয়। তবে recursion ফাংশন call কিন্তু এক পর্যায়ে শেষ হতে হবে, নাহলে কিন্তু এই call চলতেই থাকবে। আমরা যদি, $n = 3$তম ফিবনাচি বের করতে চাই সে, এটার মান বের করার জন্য $n - 1 = 2$তম এবং $n - 2 = 1$তম ফিবনাচি নাম্বার চাইবে, তারা ভিতরে গিয়ে আবার তার ছোট চাইবে এভাবে কিন্তু চলতে থাকবে। তাহলে এই অবস্থা থেকে মুক্তি কি? মুক্তি ফিবনাচি নাম্বার এর definition এই আছে।

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

আমাদের বলা আছে যে, $n \geq 1$ এর ক্ষেত্রেই কেবল এরকম $n-1$তম ও $n-2$তম ফিবনাচি নাম্বার দরকার হবে, অন্যথায় কি মান হবে তা বলা আছে। recursion ব্যবহার করে আমাদের ফিবনাচি নাম্বার নির্ণয় এর প্রোগ্রামটা কোড ?? তে দেয়া হল। মজার ব্যপার হল, আমরা এই recursive function এর মাধ্যমে বড় n এর জন্য ফিবনাচি সংখ্যার মান বের করতে পারব না (অনেক সময় লাগবে) কিন্তু loop এর সাহায্যে অনেক দূর পর্যন্ত খুব সহজেই বের করা যাবে। এর কারণ কি? তোমাদের ইতমধ্যেই কিন্তু Time Complexity নিয়ে বলেছি, তোমরা দুইটি algorithm এর Time Complexity বের করার চেষ্টা করে দেখতে পার। তাহলেই বুঝবে এমন কেন হল। যদি না বের করতে পার, চিন্তার কিছুই নেই, আমরা পরবর্তী এক অধ্যায়ে এটা নিয়ে আরও দেখব।

এই সেকশনের প্র্যাকটিস প্রবলেম গুলি একটু challenging. তোমরা ইতোমধ্যে array শিখেছ সেই array এবং recursion ব্যবহার করে সল্ভ করার মত কিছু প্রবলেম দেয়া হচ্ছেঃ

প্র্যাকটিস প্রবলেম

• Timus 1005 • Timus 1082 • Timus 1149
• LightOJ 1042 • LightOJ 1189

২.৮ File ও Structure

আমরা যখন কোড করি তখন দেখা যায় কোডে ভুল হয়, ভুল সংশোধন করে আবার আমরা চেক করে দেখি ঠিক আসে কিনা। এজন্য আমরা problem এ দেয়া sample test data দিয়ে চেক করে

থাকি বা আমাদের নিজেদের কোন case দিয়ে চেক করে থাকি। কিন্তু বার বার সেই case লিখা খুবই কঠিন কাজ। যদি case টা অনেক বড় হয় তাহলে তো কোন কথাই নেই। মাঝে মাঝে কোন কোন OJ তে file হতে ইনপুট আউটপুট করতে হয়। সুতরাং আমাদের file এর কাজও সামান্য জানতে হবে। File এ আসলে অনেক কিছু করা যায়, কিন্তু আমাদের বেশি কিছু পারার দরকার নেই। :) মনে করা যাক আমাদের input.txt ফাইল হতে ইনপুট নিতে হবে এবং output.txt ফাইল এ আউটপুট দিতে হবে। আমরা যা করব তা হল, পুরো কোডটা সাধারন ভাবেই লিখব তবে কোড এর শুরুতে দুইটি লাইন লিখবঃ freopen("input.txt", "r", stdin); এবং freopen("output.txt", "w", stdout);. তোমরা তোমাদের দরকার মত ফাইল নাম বসিয়ে নিবে তাহলেই হবে। আরও একটি উপায় আছে fopen ফাংশন এর মাধ্যমে কিন্তু তাতে তোমাকে ইনপুট আউটপুট এর জন্য ব্যবহার করা সব ফাংশন পরিবর্তন করতে হবে।

অনেকগুলো একই ধরনের জিনিস save করার জন্য আমরা array ব্যবহার করে থাকি। কিন্তু অনেক সময় আমাদের অনেক গুলো বিভিন্ন জিনিস save করার প্রয়োজন হতে পারে। যেমন, একটি ছাত্রের information. আমাদের তার নাম, পিতার নাম, ঠিকানা, জন্মসাল, ফোন নং ইত্যাদি বিভিন্ন information রাখতে হবে। আমরা যা করতে পারি তাহল বিভিন্ন information এর জন্য আলাদা আলাদা array. কিন্তু এতে করে এটা maintain করা একটু কষ্টকর হয়ে যায়। এর থেকে সুবিধার উপায় হল structure. এটি এমন একটি জিনিস যেখানে আমরা একই সাথে বিভিন্ন জিনিস একত্রে রাখতে পারি। কোড ২.15 এ আমাদের এই উদাহরণটি তুলে ধরা হল।

Listing ২.15 : structure.cpp

```
1  struct Student
2  {
3          char name[30], father[30], address[50];
4          int birth_date, birth_month, birth_year;
5          int phone;
6  };
7
8  Student s, student[50];
9
10 scanf("%s", s.name);
11 printf("%d\n", student[5].birth_date);
```

এখানে প্রথমেই আমরা Student নামে একটি structure declare করেছি যার মাঝে আমাদের সব প্রয়োজনীয় সকল variable declare করা হয়েছে। এখন এই Student নামটা একরকমের data type হিসাবে ব্যবহার করা যাবে। কোডটিতে খেয়াল কর আমরা একটি variable s এবং একটি Student টাইপ এর array $student$ তৈরি করা হয়েছে। এখন variable এর সাথে dot দিয়ে আমরা এর বিভিন্ন variable গুলি access করতে পারব। কোডটিতে নামে ইনপুট নেয়া বা জন্মতারিখ আউটপুট দেয়া দেখানো হয়েছে।

২.৯ bitwise operation

আমরা যেভাবে হিসাব নিকাশ করি বা সংখ্যা লিখি একে দশমিক সংখ্যা বা Decimal Number System বলে। আমাদের সর্বমোট 10টি অংক আছেঃ $0, 1, 2, \ldots 9$. কিন্তু আমাদের কম্পিউটার এর মাত্র দুইটি অংক আছেঃ 0 আর 1. আর এই Number System কে Binary বলা হয়। Binary তে প্রতিটি অংককে bit বলা হয়। আমরা যখন একটা variable এ 6 রাখি তখন আসলে সেখানে $0, 1$ দিয়ে তৈরি একটি সংখ্যা থাকে।

bitwise operation হচ্ছে এমন কিছু operation যা সরাসরি bit নিয়ে কাজ করে। আমরা যেসকল bitwise operator ব্যবহার করে থাকি তারা হলঃ & (bitwise and), | (bitwise or), ^ (bitwise xor), $\sim$ (1's complement), << (shift left), >> (shift right).

অধ্যায় ৩

Mathematics

৩.১ Number Theory

৩.১.১ Prime Number

Prime Number কে বাংলায় মৌলিক সংখ্যা বলে। একটি সংখ্যা n কে মৌলিক বলা হয় যদি ঐ সংখ্যাটি 1 এর থেকে বড় হয় এবং 1 বা n ছাড়া আর কোন সংখ্যা দ্বারা বিভাজ্য না হয়। এখন যদি একটি সংখ্যা n দিয়ে বলা হয় এটি Prime কিনা কেমনে করবা? মোটামোটি সংজ্ঞা থেকেই বুঝা যায় কেমনে করা উচিত। অবশ্যই n এর থেকে কোন বড় সংখ্যা দিয়ে n কে ভাগ করা যায় না। সুতরাং যদি 2 হতে $n-1$ এর মাঝের কোন একটি সংখ্যা দ্বারা n নিঃশেষে বিভাজ্য হয় তাহলে n Prime না। সুতরাং আমরা এই Idea এর উপর ভিত্তি করে যদি Primality চেক করার জন্য একটি ফাংশন লিখি তা দাঁড়াবে কোড ৩.1 এর মত।

Listing ৩.1 : isPrime1.cpp

```
1  //returns 1 if prime, otherwise 0
2  int isPrime(int n)
3  {
4          if(n <= 1) return 0;
5          for(int i = 2; i < n; i++)
6                  if(n % i == 0)
7                          return 0;
8
9          return 1;
10 }
```

এখন এর Time Complexity কত? Worst case অর্থাৎ কোডটি সবচেয়ে বেশি সময় নিবে যদি এটি Prime হয়। সেক্ষেত্রে for loop টি $n-2$বার চলবে, সুতরাং এটির Time Complexity $O(n)$. তোমরা ভাবতে পার, আচ্ছা আমরা তো জানি, 2 বাদে কোন জোড় সংখ্যা Prime না। সুতরাং for loop টা তো শুধু বিজোড় সংখ্যার উপর দিয়েই চালালেই হয়! ভালো বুদ্ধি। তাহলে আমাদের run time কত হবে? $O(n/2)$ আর আমরা বলেছি আমরা সকল constant term কে বাদ দেই। তাহলে এভাবে করলেও আমাদের run time $O(n)$ ই থাকে। হ্যা, অর্ধেক হবে কিন্তু এটা আমাদের order notation এ কোন ব্যাপারই না। যাই হোক, তাহলে আমরা কেমনে কমাবো? একটু চিন্তা করলে দেখবে যে, যদি এমন কোন d খুঁজে পাও যা n কে ভাগ করে তাহলে তুমি আরও একটি সংখ্যা কিন্তু খুঁজে বের করে ফেলেছ যেটা n কে ভাগ করেঃ n/d. অর্থাৎ কোন একটি সংখ্যার divisor (গুণনীয়ক) গুলি সবসময় জোড়ায় জোড়ায় থাকে। যেমন $n = 24$ হলে এর divisor গুলি হচ্ছেঃ $1, 2, 3, 4, 6, 8, 12, 24$ এবং তারা 4টি জোড়ায় আছেঃ $(1, 24), (2, 12), (3, 8), (4, 6)$. একটু চিন্তা করলে দেখবে প্রতিটি জোড়ার

ছোটটি সবসময় $\leq \sqrt{n}$ হবে। কেন? এটি direct প্রমাণ করা মনে হয় একটু কঠিন হবে, কিন্তু Proof by Contradiction কিন্তু খুবই সোজা। মনে কর ছোটটি $\sqrt{n}$ এর থেকেও বড়, তাহলে ঐ জোড়ার বড়টাতো বড় হবেই! আর জোড়াগুলি এমনভাবে বানানো হয়েছে যেন তাদের গুনফল n হয়। কিন্তু দুইটি $\sqrt{n}$ এর থেকে বড় সংখ্যার গুনফল কেমনে n হয়? অতএব জোড়ার ছোটটিকে অবশ্যই $\sqrt{n}$ এর সমান বা ছোট হতে হবে। এভাবে আমরা যদি কোড করি (কোড ৩.2) তাহলে আমাদের run time হবে $O(\sqrt{n})$. এখানে খেয়াল করতে পার যে আমরা আমাদের for loop এর condition টা $i * i \leq n$ লিখেছি, $i \leq sqrt(n)$ না। এর কিছু কারণ আছে। প্রথমত, বার বার sqrt হিসাব করা একটি costly কাজ। দ্বিতীয়ত, double ব্যবহার করলে কিন্তু precision loss হয়। এর ফলে sqrt(9) = 3 না হয়ে 2.9999999 বা 3.0000001 হলেও অবাক হবার কিছু নেই।[১] কিন্তু বার বার $i*i$ করাও কেমন জানি! তোমরা যা করতে পার তাহল loop শুরু হবার আগেই limit = sqrt(n + 1) করে নিতে পার। এর পর এই পর্যন্ত loop চালাবে। তাহলে বার বার sqrt ও করা লাগবে না গুন ও করা লাগবে না।

Listing ৩.2 : isPrime2.cpp

```cpp
//returns 1 if prime, otherwise 0
int isPrime(int n)
{
        if(n <= 1) return 0;
        for(int i = 2; i*i <= n; i++)
                if(n % i == 0)
                        return 0;

        return 1;
}
```

আরও কি উন্নতি করা যাবে run time? হ্যা যাবে, আসলে এটি $O(\log n)$ সময়েই করা যাবে! কারো যদি এতে আগ্রহ থাকে তাহলে internet এ সার্চ করে দেখতে পার।

Sieve of Eratosthenes

এটি Prime Number কে generate করার একটি দ্রুত উপায়। তোমরা লক্ষ্য করলে দেখবে যে পূর্বের $O(\sqrt{n})$ algorithm এ আমরা যা করেছি তা হল কোন একটি number নিয়ে তাকে কেউ ভাগ করে কিনা তা চেক করেছি। কিন্তু এর ফলে যা হয় তা হল, একটি সংখ্যা prime কিনা তা চেক করার জন্য অনেক সংখ্যা দ্বারা ভাগ করে দেখতে হয়। কিন্তু এই কাজটা যদি আমরা ঘুরিয়ে করি? অর্থাৎ কোন একটি সংখ্যাকে কে কে ভাগ করে এটা না দেখে বরং এই সংখ্যা কাকে কাকে ভাগ করে সেটা যদি দেখি তাহলেই আমাদের কাজ অনেক কমে যাবে। কারণ, এখানে আমরা শুধু মাত্র ঐসব সংখ্যার pair নিচ্ছি যারা একে অপরকে ভাগ করে। Sieve এর algorithm ঠিক এই কাজটাই করা হয়। এর মাধ্যমে তোমরা 1 হতে n এর মাঝের সব prime বের করে ফেলতে পারবে। শুধু তাই না, এই সীমার মাঝের কোন সংখ্যা দিলে সেটা prime কিনা সেটাও অনেক দ্রুত বলে দিতে পারবা। Algorithm টি কিন্তু খুবই সোজা! তুমি 2 হতে n পর্যন্ত সব সংখ্যা লিখ, এরপর প্রথম থেকে আসো, একটি করে সংখ্যা নিবা আর তার থেকে বড় তার যতগুলি multiple এখনও আস্ত আছে তা কেটে ফেল! এভাবে একে একে সব সংখ্যা নিয়ে কাজ করলে তোমার কাছে যেসব সংখ্যা অবশিষ্ট থাকবে সেগুলিই হল prime এবং এর বাইরে কিন্তু আর কোন prime নেই! তুমি কিন্তু এই কাজটা $\sqrt{n}$ পর্যন্তও করতে পার! আশা করি এতক্ষণে বুঝতে পারছ কেন! $n = 10$ এর জন্য আমরা টেবিল ৩.১ এ এই algorithm টি simulate করে দেখালাম।

তোমাদের সুবিধার জন্য এর implementation কোড ৩.3 এ দেওয়া হল। এই algorithm এর run time $O(n \log \log n)$. এই ফাংশন শেষে তোমরা একটি prime number এর লিস্ট পাবা এবং mark array থেকে বলতে পারবে কোনটি prime আর কোনটি না।

[১] মজার ব্যপার হল double কে এমনভাবে represent করা হয় যেন sqrt ফাংশনটি integer উত্তর এর ক্ষেত্রে সবসময় সঠিক উত্তর দেয়!

সারনী ৩.১: $n = 10$ এর জন্য sieve algorithm এর simulation

বিবরন	বর্তমান অবস্থা
initial অবস্থা	$2, 3, 4, 5, 6, 7, 8, 9, 10$
প্রথম uncut number = 2. আমরা 2 এর সকল multiple কেটে দেব	$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}$
পরবর্তী uncut number = 3. আমরা 3 এর সকল multiple কেটে দেব	$2, \cancel{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}$
আর দরকার নেই, $5 > \sqrt{10}$	$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}$

Listing ৩.3 : sieve.cpp

```cpp
int Prime[300000], nPrime;
int mark[1000002];

void sieve(int n)
{
        int i, j, limit = sqrt(n * 1.) + 2;

        mark[1] = 1;
        for(i = 4; i <= n; i += 2) mark[i] = 1;

        Prime[nPrime++]=2;
        for(i = 3; i <= n; i += 2)
                if(!mark[i])
                {
                        Prime[nPrime++] = i;

                        if(i <= limit)
                        {
                                for(j = i * i;j <= n;j += i * 2)
                                {
                                        mark[j] = 1;
                                }
                        }
                }
}
```

sieve এর দুইটি variation নিয়ে কথা বলা যায়।

Memory Efficient Sieve এখানে খেয়াল করলে দেখবে যে n যত বড়, তত বড় array এর দরকার হয় mark এর জন্য। আমরা কিন্তু জানি 2 ছাড়া সব জোড় সংখ্যা এর mark এ 1 থাকবে সুতরাং এই জিনিস খাটিয়ে আমরা memory requirement অর্ধেক করে ফেলতে পারি। আবার mark array এর প্রতিটি জায়গায় আমরা কিন্তু 0 আর 1 ছাড়া আর কিন্তু কিছু রাখি না। আমরা চাইলে, প্রতিটি int এ থাকা 32 bit কে কাজে লাগিয়ে একটা variable এ 32 টা information রাখতে পারি এবং memory requirement কে আরও 32 ভাগ করতে পারি।

Segmented Sieve অনেক সময় আমাদের 1 হতে n এর দরকার হয় না, a হতে b সীমার prime গুলির দরকার হয় যেখানে a, b হয়তো $10^{12} \sim 10^{14}$ range এর কিন্তু বারতি একটি শর্ত থাকে যে, $b - a \leq 10^6$. এসব ক্ষেত্রে আমরা $[a, b]$ range এ sieve চালাব। এর জন্য প্রথমেই আমাদের $\sqrt{b}$ পর্যন্ত সকল prime বের করে রাখা লাগতে পারে (prime দিয়ে করলে efficient হয় তবে চাইলে 2 হতে $\sqrt{b}$ পর্যন্ত সব সংখ্যা দিয়েও করতে পার।)

প্র্যাকটিস প্রবলেম

- একটি সংখ্যাকে prime factorize কর। অর্থাৎ এটি কোন কোন prime দ্বারা বিভাজ্য এবং সেই সব prime এর power গুলি বের কর।

৩.১.২ একটি সংখ্যার Divisor সমুহ

তুমি যদি কোন একটি সংখ্যার সকল divisor সমুহ বের করতে চাও তাহলে কোড ৩.2 এর মত $O(\sqrt{n})$ এ খুব সহজেই সকল divisor বের করে ফেলতে পার। কিন্তু আমরা কি sieve algorithm কে modify করে 1 হতে n পর্যন্ত প্রতিটি সংখ্যার সকল divisor বের করা কি সম্ভব? অবশ্যই সম্ভব, তবে এটির runtime $O(n \log n)$. কোড ৩.4 এ এর কোডটি দেখানো হল। এখানে তেমন কিছুই না, শুধু প্রতিটি সংখ্যার জন্য আমরা এর multiple এর list গুলোতে তাকে insert করে দেই। এখানে আমাদের কোন mark রাখার প্রয়োজন হয় না। তোমরা যদি মনে কর যে memory তো অনেক বেশি লেগে যাবে! না, n টি সংখ্যার divisor আসলে সর্বমোট $n \log n$ এর বেশি না। আমরা এই কোডে আমাদের সুবিধার জন্য STL এর vector ব্যবহার করেছি। vector না ব্যবহার করে Dynamic Linked List ব্যবহার করা যায়, কিন্তু জিনিসটা অনেক ঝামেলার হয়ে যায়।

Listing ৩.4 : all divisors.cpp

```
1  int mark[1000002];
2  vector<int> divisors[1000002];
3
4  void Divisors(int n)
5  {
6          int i, j;
7          for(i = 1; i <= n; i++)
8                  for(j = i; j <= n; j += i)
9                          divisors[j].push_back(i);
10 }
```

অনেক প্রবলেম এই divisor এর লিস্ট হয়তো লাগে না, কিন্তু প্রতিটি সংখ্যার divisor সমুহের sum বা সংখ্যা এর দরকার হয়। আশা করি কেমনে করবে তা বুঝতে পারতেছ!

কোন একটি সংখ্যার divisor নিয়ে যখন problem থাকে তখন আরেকটি method বেশ কাজে লাগে। ধরা যাক, $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ এখানে p_i হল prime সংখ্যা। একে কোন একটি সংখ্যার prime factorization বলে। এখন চিন্তা করে দেখ, d কে যদি n এর divisor হতে হয় তাহলে তার কি কি বৈশিষ্ট্য থাকতে হবে। প্রথমত, d এর ভিতরে p_i ছাড়া আর কোন prime divisor থাকা যাবে না। দ্বিতীয়ত, p_i এর power কিন্তু a_i এর থেকে বেশি হতে পারবে না। অর্থাৎঃ $d = p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}$ যেখানে $0 \le b_i \le a_i$. এই equation থেকে আমরা বলে দিতে পারি n এর divisor কয়টি আছে (NOD = Number of Divisor) বা তার divisor দের যোগফল কত (SOD = Sum of Divisor)!

$$NOD(n) = (a_1 + 1)(a_2 + 1) \dots (a_k + 1)$$

$$\begin{aligned} SOD(n) &= (1 + p_1 + p_1^2 + \dots + p_1^{a_1})(1 + p_2 + p_2^2 + \dots + p_2^{a_2}) \dots (1 + p_k + p_k^2 + \dots + p_k^{a_k}) \\ &= \frac{p_1^{a_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{a_2+1} - 1}{p_2 - 1} \cdot \dots \cdot \frac{p_k^{a_k+1} - 1}{p_k - 1} \end{aligned}$$

৩.১.৩ GCD ও LCM

GCD এর পূর্ণ রূপ হলঃ Greatest Common Divisor বাংলায় গরিষ্ঠ সাধারণ গুণনীয়ক (গসাগু) আর LCM এর পূর্ণ রূপ হলঃ Least Common Multiple বাংলায় লঘিষ্ঠ সাধারণ গুণিতক (লসাগু)।

যদি a এবং b দুইটি সংখ্যার গসাগু g এবং লসাগু l হয় আমরা বলতে পারিঃ $a \times b = g \times l$. সুতরাং আমরা যদি দুইটি সংখ্যার গসাগু বের করতে পারি তাহলে লসাগু খুব সহজেই বের হয়ে যাবে। প্রশ্ন হল আমরা গসাগু কেমনে বের করব? একটি উপায় হল a ও b এর মাঝে যেটি ছোট সেই সংখ্যা থেকে শুরু করে 1 পর্যন্ত দেখা, যেই সংখ্যা দিয়ে প্রথম ভাগ যাবে সেটিই গসাগু। কিন্তু এর run time $O(n)$. এর থেকে ভালো উপায় কিন্তু তোমরা জানো, ছোট বেলায় স্কুল এ থাকতে শিখেছ সেটি হল euclid এর পদ্ধতি। মনে কর আমাদের a আর b দিয়ে বলা হল এদের গসাগু বের করতে হবে, আমরা যা করব, a কে b দিয়ে ভাগ দিব। যদি নিঃশেষে ভাগ যায়, তাহলে b ই গসাগু কারন b এর থেকে বড় কোন সংখ্যা কিন্তু b কে ভাগ করে না (যদিও a কে ভাগ করতে পারে)। এখন যদি ভাগ না যায়, সেক্ষেত্রে আমরা ভাগশেষ c বের করব $a = k \cdot b + c$. এই c কিন্তু b এর থেকে ছোট হবে! ($a < b$ হলে কি হবে তা চিন্তা করে দেখতে পার!) এবং একটি সংখ্যা যদি a ও b কে ভাগ করে সেটা এই সমীকরণ অনুসারে c কেও করবে। সুতরাং আমরা এখন b ও c এর গসাগু বের করব। আগে ছিল a ও b এখন এদের একটি সংখ্যা ছোট হয়ে c হয়ে গেল। সুতরাং এই কাজটা যদি আমরা বার বার করতে থাকি এক সময় আমরা গসাগু পেয়ে যাব। তোমাদের মনে হতে পারে যে অনেক বার এই কাজ করতে হবে! কিন্তু আসলে কিন্তু তা না। এটার exact run time তোমাদেরকে বললে আপাতত বুঝবে না তবে এটুকু বিশ্বাস করতে পার যে long long এ যত বড় সংখ্যা ধরা সম্ভব তাদের যদি গসাগু বের করতে বলা হয় তাহলে $100 \sim 150$ ধাপের বেশি আসলে লাগবে না।[১] গসাগু নির্ণয়ের একটি recursive প্রোগ্রাম কোড ৩.5 এ দেয়া হল।

Listing ৩.5 : gcd.cpp

```
int gcd(int a, int b)
{
        if(b == 0) return a;
        return gcd(b, a % b);
}
```

৩.১.৪ Euler এর Totient Function (ϕ)

শুরুতেই আমরা জেনে নেই Totient Function কি জিনিস।

$\phi(n) =$ n এর থেকে ছোট বা সমান এমন কতগুলি সংখ্যা আছে যা n এর সাথে coprime

coprime অর্থ হল তাদের কোন সাধারণ factor নেই। যেমন, $\phi(12) = 4$ কারণ $2, 3, 4, 6, 8, 9, 10, 12$ এর সাথে 12 এর কোন না কোন সাধারণ factor আছে। $1, 5, 7, 11$ এই চারটি সংখ্যার সাথে কোন common factor নেই। যদি $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ হয় তাহলেঃ

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\dots\left(1 - \frac{1}{p_k}\right)$$

সুতরাং যদি কোন একটি সংখ্যার ϕ বের করতে হয় তাহলে তোমরা খুব সহজেই prime factorize করে বের করে ফেলতে পারবে। কথা হল prime বা divisor এর মত কি এখেত্রেও 1 হতে n এর ϕ এর মান কি দ্রুত বের করা সম্ভব? অবশ্যই সম্ভব, প্রতিটি prime p এর জন্য আমরা এর multiple এ গিয়ে তাকে p দিয়ে ভাগ ও $p - 1$ দিয়ে গুন করলেই হবে। এভাবে 1 হতে n পর্যন্ত সব prime এর জন্য এই কাজ করলেই আমরা সব সংখ্যার ϕ এর মান পেয়ে যাব। (কোড ৩.6)

Listing ৩.6 : sieve phi.cpp

```
int phi[1000006], mark[1000006];

void sievephi(int n)
```

[১]তোমাদের আগ্রহ থাকলে এই বিষয়ে wiki তে পড়তে পার। মজার ব্যপার হল এর runtime এর সাথে ফিবনাচি সংখ্যার একটা সম্পর্ক আছে!

```
{
        int i,j;

        for(i = 1; i <= n; i++) phi[i] = i;

        phi[1] = 1;
        mark[1] = 1;

        for(i = 2;i <= n;i += 2)
        {
                if(i != 2) mark[i] = 1;
                phi[i] = phi[i] / 2;
        }

        for(i = 3; i <=n; i += 2)
                if(!mark[i])
                {
                        phi[i] = phi[i] - 1;

                        for(j = 2 * i; j <= n;j += i)
                        {
                                mark[j] = 1;
                                phi[j] = phi[j] / i * (i - 1);
                        }
                }
}
```

৩.১.৫ BigMod

BigMod একটি অত্যন্ত গুরুত্বপূর্ণ একটি Method. এক জন bowler এর কাছে লাল বল আছে আবার সাদা বল ও আছে। তুমি n টি বল করলে, প্রতিটি বল হয় সাদা বল দিয়ে করবে না হয় লাল বল দিয়ে। তুমি কত ভাবে বল করতে পার? এখানে প্রতিটি বল করতে পারছ 2 ভাবে, সুতরাং সর্বমোট 2^n ভাবে বল করতে পারবে। কিন্তু n এর বড় মানের জন্য এটা অনেক বড় সংখ্যা হয়ে দাঁড়ায়। সেজন্য প্রায় বেশির ভাগ সময়ে আমাদের exact উত্তর না চেয়ে mod 10^7 বা এরকম একটি সংখ্যা দিয়ে দেয়া হয় যার দ্বারা mod করে result চাওয়া হয়। যেমন ধর তোমাদের জিজ্ঞাসা করলাম, 2^{100} mod 7 কত? কি করবে? 2^{100} বের করে এর পর 7 দিয়ে ভাগ করে উত্তর দিবে? খেয়াল কর, 2^{100} কিন্তু long long এও ধরবে না। তাহলে উপায়? তুমি চাইলে প্রতিবার গুন করে সাথে সাথেই mod করতে পার, এতে করে উত্তরটা শুধু int ব্যবহার করেই পেয়ে যাবে! কিন্তু তোমাকে যদি 100 এর থেকে আরও বড়, অনেক বড় ধর প্রায় 10^{18} দেয়া হয়? তাহলে কি করা সম্ভব? হ্যা করা সম্ভব এবং idea টাও অনেক সহজ। মনে কর তোমাকে 2^{100} mod 7 বের করতে বলা হয়েছে। তুমি কি করবে, 2^{50} mod 7 বের করবে, ধর এটি a, তাহলে 2^{100} mod 7 হবে $(a \times a)$ mod 7. অর্থাৎ তুমি তোমার কাজ কে অর্ধেক করে ফেললে। তোমার এখন আর for loop চালিয়ে 100টা 2 গুন করতে হবে না, 50টা 2 গুন করে এই গুনফল কে তার সাথেই গুন দিলে তুমি result পেয়ে যাবে। কিন্তু একই ভাবে তোমার কিন্তু 50 বার গুন করার দরকার নেই, 25 বার গুন করে আবার সেই গুনফল কে তার সাথেই গুন করে গুনফল তুমি পেয়ে যাবে। কিন্তু এবার? বিজোড় সংখ্যার বেলায়? এবার তো আর অর্ধেক করতে পারবা না। তাতে কি যায় আসে! তুমি 2^{24} বের করে তার সাথে 2 কে গুন করতে পার। যেহেতু power বিজোড় বলে এই কাজ করছ সুতরাং এর পরের ধাপে power অবশ্যই জোড় হবে এবং আবার তুমি 2 দিয়ে ভাগ করতে পারবে। এভাবে চলতে চলতে এক সময় power যখন 0 হয়ে যাবে তখন আমরা জানি কিন্তু যে $2^0 = 1$ সুতরাং এবার আমরা আমাদের অন্য গুন গুলো করে আমাদের উত্তর বের করে ফেলব।

তাহলে এই জিনিস আমরা কোড করব কেমনে? আমরা ধরে নেই আমাদের কাছে একটা Black Box আছে যাকে a, b, M দিলে a^b mod M বের করে দেয়। সে যা করবে তা হল যদি b জোড় হয় তাহলে আবার সেই Black Box কে $a, b/2, M$ দিবে এবং সেই ফলাফল দিয়ে নিজের ফলাফল বের করবে। আর যদি বিজোড় হয় তাহলে সে Black Box কে $a, b-1, M$ দিবে এবং তার ফলাফল থেকে নিজের ফলাফল বানিয়ে নিবে। এবং এই ভাবে কতক্ষণ চলবে? যতক্ষণ না, $b = 0$ হয়। এখানে Black

Box হল একটি function এবং এই ভাবে একটি function এর ভিতর থেকে একই function ব্যবহার করা কেই recursive function বলে। আমরা কোড ৩.7 এ এই প্রোগ্রামটি দিলাম। তবে প্রোগ্রামটিতে b বিজোড় এর ক্ষেত্রে একটু অন্যভাবে করা হল, আশা করি বুঝতে অসুবিধা হবে না। এই algorithm এর run time হল $O(\log n)$.

Listing ৩.7 : bigmod.cpp

```
1  int bigmod(int a, int b, int M)
2  {
3          if(b == 0) return 1 % M;
4          int x = bigmod(a, b / 2, M);
5          x = (x * x) % M;
6          if(b % 2 == 1) x = (x * a) % M;
7          return x;
8  }
```

এই ধরনের solving method কে divide and conquer বলা হয়ে থাকে। এখানে একটি বড় প্রবলেম কে ছোট ছোট ভাগে ভাগ করা হয় এবং তাদের সমাধান combine করে বড় প্রবলেম এর সমাধান বের করা হয়। আমি যখন BigMod দেখাই এর সাথে আরও একটি সমস্যা দেখাই। তাহল, $1 + a + a^2 + \ldots + a^{b-1} \bmod M$ বের করা। অবশ্যই এর আগের সমস্যার মত এখানেও b অনেক বড়। সুতরাং তুমি যে বার বার প্রতিটা term এর উত্তর বের করবে তা হবে না। এখানেও কিন্তু এর আগের মত বুদ্ধি খাটানো সম্ভব। আমরা $b = 6$ এর জন্য দেখি কেমনে একে দুই ভাগে ভাগ করা সম্ভব!

$$1 + a + a^2 + a^3 + a^4 + a^5 = (1 + a + a^2) + a^3(1 + a + a^2)$$

এভাবে যদি আমরা দুই ভাগ করি তাহলে আমাদের $bigmod(a, b/2, M)$ এর প্রয়োজন পরে। সর্বমোট $\log n$ ধাপ এবং প্রতি ধাপে আমাদের bigmod এর জন্য আরও $\log n$ সময় দরকার হয়, সুতরাং আমাদের run time $O(\log n^2)$. এটা খুব একটা বড় cost না। কিন্তু তবুও আমরা এর $O(\log n)$ সমাধান শিখব। আমরা equation টিকে অন্যভাবে দুইভাগ করার চেষ্টা করি।

$$\begin{aligned} 1 + a + a^2 + a^3 + a^4 + a^5 &= (1 + a^2 + a^4) + a(1 + a^2 + a^4) \\ &= (1 + (a^2) + (a^2)^2) + a(1 + (a^2) + (a^2)^2) \end{aligned}$$

অর্থাৎ আমরা আমাদের নতুন ফাংশন এর নাম যদি bigsum দেই তাহলে $bigsum(a, b, M)$ বের করতে আমরা $bigsum(a^2, b/2, M)$ বের করব। a যেন পরবর্তীতে overflow না করে সেজন্য আমরা আসলে $a^2 \bmod M$ পাঠাবো। b বিজোড় হলে আশা করি বুঝতে পারছ যে কি করব? তাও দেখাইঃ

$$1 + a + a^2 + a^3 + a^4 = 1 + a(1 + a + a^2 + a^3)$$

আশা করি কোডটা নিজেরাই করতে পারবে। তোমাদের একই ভাবে সমাধান করা যাবে এমন আরেকটি সমস্যা দেই প্র্যাকটিস এর জন্য।

প্র্যাকটিস প্রবলেম

- $1 + 2a + 3a^2 + 4a^3 + 5a^4 + \ldots + ba^{b-1} \bmod M$ বের কর।

৩.১.৬ Modular Inverse

যেহেতু অনেক হিসাবের উত্তর অনেক বড় আসে সুতরাং প্রায় সময়ই আমাদের exact উত্তর না চেয়ে কোন একটি সংখ্যা দিয়ে mod করে উত্তর চায়। এখন সেই হিসাবে যদি যোগ বিয়োগ গুন থাকে তাহলে কোন সমস্যা হয় না, কিন্তু যদি ভাগ থাকে তাহলে বেশ সমস্যা হয়ে যায়, কারণ $\frac{a}{b} \bmod M$ এবং $\frac{a \bmod M}{b \bmod M}$ এক কথা না। তুমি যদি b দ্বারা ভাগ করতে চাও তাহলে $b^{-1} \equiv M$ বের করতে হবে এবং এটি দিয়ে গুন করলেই b দ্বারা ভাগের কাজ হয়ে যাবে। M যদি prime হয় তাহলে, $b^{-1} \equiv b^{M-2} \bmod M$. আর যদি তা না হয়, তাহলে $b^{-1} \equiv b^{\phi M-1} \bmod M$ কিন্তু সেক্ষেত্রে M এবং b কে coprime হতে হবে। এবং এই মান আমরা bigmod ব্যবহার করে খুব সহজেই বের করে ফেলতে পারি।

৩.১.৭ Extended GCD

যদি a ও b এর গসাগু g হয় তাহলে এমন x এবং y খুঁজে পাওয়া সম্ভব যেন $ax + by = g$ হয়। টেবিল ৩.২ এ আমরা $a = 10$ এবং $b = 6$ এর জন্য x ও y বের করে দেখালাম। অনেকেই এই টেবিল দেখে ভরকিয়ে যায়। আসলে ভয় পাবার কিছু নেই। এই টেবিল এর প্রথম কলাম হল সাধারণ euclid এর গসাগু বের করার পদ্ধতি হতে পাওয়া। এখন গসাগু করার সময় অবশ্যই ছোট সংখ্যাকে কোন একটি সংখ্যা দিয়ে গুন করে বড়টি হতে বাদ দিয়েই ভাগশেষ বের করা হয়। এই গুন ও বিয়োগ এর কাজটা আমাদের পরের দুই কলামেও করতে হবে। এরকম করলে আমরা যখন প্রথম কলামে গসাগু পাব তখন দ্বিতীয় ও তৃতীয় কলামে x ও y এর মান পেয়ে যাব। এক্ষেত্রে আমাদের $x = -1$ এবং $y = 2$.

সারনী ৩.২: $a = 10$ ও $b = 6$ এর জন্য Extended GCD এর simulation

সংখ্যা	x	y	10x + 6y
10	1	0	10
6	0	1	6
4	1	-1	4
2	-1	2	2

আমরা Extended GCD কে সংক্ষেপে egcd বলে থাকি। এই প্রোগ্রামটির কোড ৩.৪ এ দেয়া হল। এখানে দেয়া egcd ফাংশনটি call করার সময় দুইটি variable এর reference ও পাঠাতে হবে যেখানে x এবং y এর মান থাকবে। এই ফাংশনটি গসাগু return করে।

Listing ৩.৪ : egcd.cpp

```
int egcd (int a, int b, int &x, int &y)
{
        if (a == 0)
        {
                x = 0; y = 1;
                return b;
        }

        int x1, y1;
        int d = egcd (b%a, a, x1, y1);

        x = y1 - (b / a) * x1;
        y = x1;

        return d;
 }
```

egcd ব্যবহার করেও আমরা Modular Inverse বের করতে পারি (তবে b ও M কে coprime হতে হবে)। কোন একটি b এর জন্য আমরা এমন একটি x বের করতে চাই যেন, $bx \equiv 1 \bmod M$.

অর্থাৎ, $bx = 1 + yM$ যেখানে y হল একটি integer. এখন, $bx - yM = 1$. আমরা জানি b ও M coprime সুতরাং আমরা এমন একটি x ও y পাব যেন, $bx - yM = 1$ হয় বা, $bx \equiv 1 \mod M$ হয়। তবে egcd ফাংশন x এর মান হিসেবে ঋণাত্মক সংখ্যা দিতে পারে, এ জিনিসটা খেয়াল রাখতে হবে। সেক্ষেত্রে x কে সঠিক ভাবে[১] M দ্বারা mod করে এর অঋণাত্মক মানটা বের করতে হবে।

৩.২ Combinatorics

৩.২.১ Factorial এর পিছের 0

$100!$ এর পিছনে কতগুলি শূন্য আছে? - এটি খুবই common একটি প্রশ্ন। তোমরা হয়তো ইতোমধ্যেই এর সমাধান জানো। যারা জানো না, তাদের জন্য বলি আমাদের বসে বসে $100!$ এর মত এত বিশাল সংখ্যা বের করার দরকার নেই। শুধু আমাদের জানতে হবে যে কত গুলি 10 গুন করা হচ্ছে। কিন্তু একটু খেয়াল করলে দেখব যে, $5! = 120$ কিন্তু এখানে আমরা কোন 10 গুন করিনি, এর পরও একটি 0 চলে এসেছে। কেন? কারণ আমরা 5 আর 2 গুন করেছি এরা আমাদের 10 দিয়েছে। অর্থাৎ আমাদের দেখতে হবে এই গুনফল এর মাঝে কত গুলি 2 আর কতগুলি 5 গুণিতক আকারে আছে। আসলে 2 কত বার আছে তা দেখার দরকার হয় না, কারণ 2 সবসময় 5 এর থেকে বেশি বার থাকবেই, সুতরাং আমাদের 5 গুনলেই চলবে। এখন আমরা চিন্তা করি, 5 কত বার আছে তা কেমনে বের করব। 1 হতে 100 এর মাঝে সর্বমোট $\lfloor 100/5 \rfloor = 20$ টি 5 এর গুণিতক আছে। এগুলি থেকে একটি করে 5 পাব। কিন্তু যেগুলি 25 দ্বারা ভাগ যায় তাদের থেকে কিন্তু আরও একটি করে 5 পাবো, আবার যেগুলো 125 দ্বারা বিভাজ্য তাদের থেকে আরও একটি করে পাবো। কিন্তু 125 কিন্তু আমাদের 100 থেকে বড় তাই আমাদের আর 5 এর বড় power গুলোকে দেখার প্রয়োজন নেই। সুতরাং আমাদের 5 এর মোট সংখ্যা $\lfloor 100/5 \rfloor + \lfloor 100/25 \rfloor = 20 + 4 = 24$. অর্থাৎ $100!$ এর পিছনে মোট 24 টা শূন্য থাকবে। আমরা যদি $n!$ এর ভিতরে একটি prime number p কতগুলি আছে সেটা বের করতে চাই তাহলে আমাদের সুত্র হচ্ছেঃ

$$\lfloor n/p \rfloor + \lfloor n/p^2 \rfloor + \lfloor n/p^3 \rfloor + \ldots \text{যতক্ষণ না শূন্য হয়}$$

৩.২.২ Factorial এর Digit সংখ্যা

$n!$ এর পিছনের 0 এর সংখ্যা নাহয় বুদ্ধি করে বের করা গেল, কিন্তু $n!$ এ কতগুলি ডিজিট আছে তা কেমনে বের হবে? তোমরা যদি কোন একটি calculator এ $50!$ করে দেখ তাহলে হয়তো 3.04140932×10^64 এরকম সংখ্যা দেখতে পাবে। এখান থেকে বুঝতে পারছি যে 3 এর পরও আরও 64 টা সংখ্যা আছে। এখন আমাদের জানা এমন কি কোন ফাংশন আছে যেটা ঐ 10 এর উপরে থাকা power টা আমাদের বলে দেবে? আছে, log. তোমরা তোমাদের calculator এ যদি এই সংখ্যার log নাও তাহলে দেখবে $64.4830\ldots$ এরকম একটি সংখ্যা আসবে। সুতরাং আমরা যদি এর floor নিয়ে এক যোগ করি তাহলেই আমরা number of digits পেয়ে যাব।[২] আমরা যেকোনো সংখ্যার digit সংখ্যা বের করতে চাইলে এই পদ্ধতি কাজে লাগে। তোমাদের হয়তো কেউ কেউ ভাবছ, log নেবার জন্য তো আমাদের আগে $50!$ বের করতে হবে এর পর না log! না, log এর একটি সুন্দর বৈশিষ্ট্য আছে আর তা হলঃ $\log(a \times b) = \log a + \log b$ অর্থাৎ $\log 50! = \log 1 + \log 2 + \ldots \log 50$.

৩.২.৩ Combination: $\binom{n}{r}$

n টি জিনিস হতে r টি জিনিস কত ভাবে নির্বাচন করা যায় তার ফর্মুলা হলঃ $\binom{n}{r} = \frac{n!}{(n-r)!r!}$. অনেক সময়ই n ও r এর মান দেয়া থাকলে আমাদের $\binom{n}{r}$ এর মান নির্ণয় করার দরকার হয়। একটি উপায় হল factorial সমুহের মান আলাদা আলাদা করে নির্ণয় করে গুন ভাগ করা। কিন্তু এতে একটা সমস্যা হয় আর তা হল overflow. যেমন $n = 50, r = 1$ হলে $50!$ বের করতে গেলে আমাদের

[১] x = (x mod M + M) mod M

[২] ceiling নিলে যদি আমাদের সংখ্যাটি 10^x টাইপ এর সংখ্যা হয় তাহলে আমাদের উত্তরটি সঠিক আসবে না।

মানটা overflow করবে কিন্তু আমরা জানি $\binom{50}{1} = 50$. তোমরা যদি ভেবে থাক যে double ডাটা টাইপ ব্যবহার করবা, তাহলে সেটা সম্ভব না। কারণ double ডাটা টাইপ তোমাকে মানের একটা approximation মান দিবে, কখনই exact মান দিবে না। [১] অনেকে যা করে তা হল, $(n-r)!$ বা $r!$ এর মাঝে যেটা বড় তা দিয়ে $n!$ এর সাথে আগেই কাটাকাটি করে ফেলে, এর পর উপরের গুলো গুন এবং নিচের গুলো গুন করে এই দুইটি সংখ্যা ভাগ করে ফলাফল পাওয়া যায়। ফলে উপরের উদাহরন এ উপরে শুধু 50 থাকে আর নিচে থাকে 1 ফলাফল 50. কিন্তু এই method এও উপরেরটা overflow করে যেতেই পারে যেখানে হয়তো ভাগ দেবার পর উত্তরটা আর overflow করবে না। আরও একটি উপায় আছে আর তাহল, আমরা জানি যে $\binom{n}{r}$ সবসময় একটি পূর্ণ সংখ্যা। এর মানে নিচে যেসব সংখ্যা আছে তারা সবাই কাটাকাটির সময় কাটা পরবে। তোমরা উপরের সংখ্যা গুলিকে একটি array তে নাও। এর পর একে একে নিচের সংখ্যা নাও আর ঐ array এর প্রথম থেকে শেষ পর্যন্ত যাও, gcd নির্ণয় করবা আর এই দুইটি সংখ্যাকে কাটবা যতক্ষণ না নিচ থেকে নেয়া সংখ্যাটা 1 হয়ে যায়। এভাবে কাজ করলে তোমার উত্তর কখনই overflow করবে না। এভাবে না না রকম ভাবে তোমরা $\binom{n}{r}$ এর মান নির্ণয় করতে পারবে, প্রতিটি পদ্ধতিরই সুবিধা অসুবিধা আছে, overflow, time complexity, memory complexity, implementation complexity ইত্যাদি। প্রবলেম এর উপর ভিত্তি করে তোমাদের বিভিন্ন পদ্ধতি অবলম্বন করার দরকার হতে পারে।

অনেক সময় আমাদের exact মান না চেয়ে কোন একটি সংখ্যা দ্বারা mod করার পরের মান চেয়ে থাকে। এক্ষেত্রেও আমাদের নানা রকম পদ্ধতি আছে। যদি mod করতে বলা সংখ্যাটি prime হয় তাহলে n! mod p, ModuloInverse(r! mod p, p) এবং ModuloInverse((n-r)! mod p, p) এই তিনটি সংখ্যা গুন করলেই আমারা আমাদের কাঙ্খিত সংখ্যা পেয়ে যাব। এক্ষেত্রে সাধারনত $p > n$ হয়ে থাকে। আমরা facotiral এর mod মান আগে থেকেই precalculate করে রেখে মাত্র $O(\log n)$ এই এই মান নির্ণয় করতে পারি। mod যদি prime হয় কিন্তু ছোট হয় তাহলে অন্য একটি উপায় আছে এবং সেটি হল Lucas' Theorem. এই theorem বলেঃ

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} (mod p)$$

যেখানে, $m = m_k p^k + \ldots + m_1 p^1 + m_0 p^0$ এবং $n = n_k p^k + \ldots + n_1 p^1 + n_0 p^0$.

অর্থাৎ তোমাকে শুধু $\binom{a}{b}$ গুলি জানতে হবে যেখানে $a, b < p$ জিনিসটা কিন্তু তোমরা precalculate করে রাখতে পার।

যদি সংখ্যাটা prime না হয় বা আমাদের অনেক দ্রুত ($O(1)$) $\binom{n}{r}$ এর মান বের করতে হয় তাহলে আমরা অনেক সময় $\binom{n}{r}$ এর মান precalculate করে $n \times n$ array তে রেখে দেই। এটা তৈরি করতে আমাদের $O(n^2)$ সময় লাগে। এই পদ্ধতির মূল ফর্মুলা হলঃ $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$ [২] কোড ৩.9 এ এই কোডটি দেয়া হল। তোমরা limncr এর মান পরিবর্তন করে দরকার মত $\binom{n}{r}$ এর মান generate করতে পার।

Listing ৩.9 : ncr.cpp

```
ncr[0][0] = 1;
int limncr = 10;
for(i = 1; i <= limncr; i++)
        for(j = 0; j <= limncr; j++)
        {
                if(j > i) ncr[i][j] = 0;
                else if(j == i || j == 0) ncr[i][j] = 1;
                else ncr[i][j] = ncr[i - 1][j - 1] + ncr[i - 1][j];
        }
```

[১] যদি তোমাকে বলা হয় $\sqrt{2}$ বা π এর মান লিখ তুমি কি তা লিখতে পারবে? পারবে না, তুমি যাই লিখ না কেন সেটা আসলে আসল মানের একটি approximation মান।

[২] চিন্তা করে দেখ base case কি হওয়া উচিত

৩.২.৪ কিছু special number

কিছু কিছু special number আছে যা প্রবলেম সমাধান করার সময় প্রায়ই আমরা তাদের সম্মুখিন হই। অনেক সময় এসব মান নির্ণয়ের উপায় আমাদের অন্য সমস্যা সমাধানেও বেশ কাজে লাগে। আমরা এরকম কিছু সংখ্যা এই সেকশনে দেখব। এখানে আমরা হয়তো এসব নাম্বার এর ফর্মুলা কেমনে বের করা যায় তা দেখব না। তোমরা চাইলে internet বা কোন বই থেকে দেখে নিতে পার।

Derangement Number

একটি বক্সে n জন তাদের টুপি রাখল। এরপর প্রত্যেকে একটি করে টুপি ঐ বাক্স থেকে তুলে নিল। কত উপায়ে তারা এমন ভাবে টুপি তুলে নিবে যেন কেউই তাদের নিজেদের টুপি না পায়! যেমন যদি $n = 3$ হয় তাহলে উত্তর 2 BCA এবং CAB এই দুই উপায়েই হতে পারে (আশা করি বুঝতে পারছ যে প্রথম জনের টুপি A, দ্বিতীয় জনের টুপি B ও তৃতীয় জনের টুপি C)। প্রথমে এটি অনেক সহজ মনে হলেও আসলে এই সমস্যাটার সমাধান একটু অন্য রকম। তোমরা চাইলে inclusion exclusion principle দিয়েও এটি সমাধান করতে পার কিন্তু এখানে তোমাদের recurrance এর মাধ্যমে সমাধান তা আমি দেখাব।

মনে কর D_n হল n তম Derangement Number অর্থাৎ n জন মানুষের জন্য উত্তর। এখন এদের মধ্য থেকে প্রথম জনকে নাও। সে নিজের বাদে অন্য $n-1$ জনের টুপি পরতে পারে। ধরা যাক সে X এর টুপি পরেছে। এখন এই X সেই প্রথম জনের টুপি পরতে পারে আবার নাও পারে। যদি প্রথম জনের টুপি সে পরে তাহলে বাকি $n-2$ জন নিজেদের মাঝে D_{n-2} ভাবে টুপি আদান প্রদান করতে পারে সুতরাং এটি হতে পারেঃ $(n-1)D_{n-2}$ ভাবে। আর যদি X প্রথম জনের টুপি না নেয় তাহলে আমরা ধরে নিতে পারি যে ঐ টুপির মালিক এখন X এবং তার ঐ টুপি পরা যাবে না। অর্থাৎ এখন আমাদের কাছে $n-1$ টা মানুষ ও $n-1$টা টুপি আছে যারা কেউই নিজেদের টুপি পরতে চায় না। এই ঘটনা ঘটতে পারে $(n-1)D_{n-1}$ উপায়ে। অর্থাৎ,

$$D_n = (n-1)D_{n-2} + (n-1)D_{n-1}$$

Catalan Number

- $2n$ সাইজের কতগুলি Dyck Word আছে? $2n$ সাইজের Dyck Word এ n টি X ও n টি Y থাকে এবং এর কোন prefix এ Y এর সংখ্যা X এর থেকে বেশি নয়। যেমনঃ $n = 3$ এর জন্য Dyck Word গুলি হলঃ XXXYYY, XYXXYY, XYXYXY, XXYYXY এবং XXYXYY.

- nটি opening bracket ও n টি closing bracket ব্যবহার করে কতগুলি সঠিক paranthese expression বানান যায়? যেমনঃ $n = 3$ এর জন্যঃ ((())), ()(()), ()()(), (())() এবং (()()).

- n leaf আলা কয়টি complete binary tree আছে?

- n বাহু বিশিষ্ট একটি polygon কে কত ভাবে triangulate করা যায়?

এরকম অনেক প্রশ্নের উত্তর হল Catalan Number. [১] n তম Catalon Number কে আমরা C_n লিখে থাকি। এর ফর্মুলা হলঃ

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1}$$

[১] তোমারা চাইলে http://en.wikipedia.org/wiki/Catalan_number হতে আরও interpretation গুলি পড়ে দেখতে পার।

Stirling Number of Second Kind

nটা আলাদা জিনিসকে k ভাগে যত ভাগে ভাগ করা যায় তাই হল Stirling Number of Second Kind. একে $\left\{ {n \atop k} \right\}$ দ্বারা প্রকাশ করা হয়। মনে কর $n = 3$ ও $k = 2$ এক্ষেত্রে উত্তর কিন্তু 3, $(AB, C), (AC, B), (BC, A)$. এখন এর recurrence বের করার জন্য আমরা কিছুটা Derangement Number বের করার মত করে চিন্তা করব। প্রথমে n টি জিনিস থেকে সর্বশেষ জিনিসটা নেই। এখন এই জিনিসটা একাই একটি ভাগে থাকতে পারে। সেক্ষেত্রে বাকি $n-1$ টা জিনিস $k-1$ ভাগে ভাগ করতে হবে। (খেয়াল কর, আমরা শুধু মাত্র শেষটাই আলাদা করে নিয়েছি) এটা সম্ভব $\left\{ {n-1 \atop k-1} \right\}$ ভাবে। আবার এটাও সম্ভব যে, বাকি $n-1$ টা জিনিস k ভাগে আছে আর শেষ জিনিসটা এদেরই কোন একটায় আছে। এই কোন একটি k টার মাঝের কোন একটি। সুতরাং এটি হতে পারে $k\left\{ {n-1 \atop k} \right\}$ ভাবে। অর্থাৎ,

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k\left\{ {n-1 \atop k} \right\}$$

এখন যেকোনো recurrance এর base case থাকে। $k = 1$ হলে সব জিনিসকে এক ভাগে কতভাবে ভাগ করা যায়? মাত্র এক ভাবেই তাই না? আরও যদি $k = n$ হয়? অর্থাৎ n টি জিনিসকে n ভাগে কত ভাবে ভাগ করা যায়? এক ভাবেই। অর্থাৎ base case হলঃ

$$\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1$$

Stirling Number of First Kind

nটা আলাদা জিনিসকে k টি cycle এ যত ভাগে ভাগ করা যায় তাই হল Stirling Number of First Kind. একে $\left[{n \atop k} \right]$ দ্বারা প্রকাশ করা হয়। মনে কর $n = 4$ ও $k = 2$ এক্ষেত্রে উত্তর কিন্তু 11, (AB, CD), (AD, BC), (AC, BD), (A, BCD), (A, BDC), (B, ACD), (B, ADC), (C, ABD), (C, ADB), (D, ABC), (D, ACB)). এর recurrence ও আমরা একই ভাবে বের করতে পারি। প্রথমে n টি জিনিস থেকে সর্বশেষ জিনিসটা নেই। এখন এই জিনিসটা একাই একটি cycle এ থাকতে পারে। সেক্ষেত্রে বাকি $n-1$ টা জিনিস $k-1$ cycle এ ভাগ করতে হবে। (খেয়াল কর, আমরা শুধু মাত্র শেষটাই আলাদা করে নিয়েছি) এটা সম্ভব $\left[{n-1 \atop k-1} \right]$ ভাবে। আবার এটাও সম্ভব যে, বাকি $n-1$ টা জিনিস k cycle এ আছে আর শেষ জিনিসটা এই $n-1$টার মাঝে কোন একটির পর থাকে। সুতরাং এটি হতে পারে $(n-1)\left[{n-1 \atop k} \right]$ ভাবে। অর্থাৎ,

$$\left[{n \atop k} \right] = \left[{n-1 \atop k-1} \right] + (n-1)\left[{n-1 \atop k} \right]$$

এখন যেকোনো recurrance এর base case থাকে। $k = 1$ হলে সব জিনিসকে একটি cycle এ কত ভাগে ভাগ করা যায়? এটি একটি common সমস্যা এর উত্তর $(n-1)!$ আর যদি $k = n$ হয়? অর্থাৎ n টি জিনিসকে n cycle এ কত ভাবে ভাগ করা যায়? এক ভাবেই। অর্থাৎ base case হলঃ

$$\left[{n \atop 1} \right] = (n-1)!, \left[{n \atop n} \right] = 1$$

৩.২.৫ Fibonacci Number

ফিবনাচি নাম্বার এর সাথে তোমরা ইতোমধ্যেই পরিচিত হয়ে গেছো এবং হয়তো $n \leq 10^6$ এর জন্য ফিবনাচি নাম্বারও বের করে ফেলেছ। কিন্তু যদি আরও বড় বের করতে বলা হয়, ধর $n \leq 10^18$ এর জন্য? (আমরা কিন্তু mod মান বের করব)। এর জন্য একটি সুন্দর method আছে। তোমরা যারা matrix জানো না তারা একটু matrix পড়ে নিতে পার। matrix ছাড়াও এটি করা যায় কিন্তু এটি এক-টি general method সুতরাং তোমরা এই method ব্যবহার করে আরও অন্য অনেক problem সল্ভ করতে পারবে। Matrix ব্যবহার করে আমরা লিখতে পারিঃ

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$
$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

একই ভাবে,

$$\begin{bmatrix} F_4 \\ F_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

সুতরাং আমরা লিখতে পারি,

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

তোমরা যদি ভেবে থাক ঐ matrix এর power তুলতে গেলে তো খবর হয়ে যাবে, তাহলে ভুল ভাববে। কিছুক্ষণ আগেই কিন্তু আমরা BigMod শিখে এসেছি, ওখানে ছিল একটি সংখ্যা এখানে আছে matrix. সংখ্যা গুন করার পরিবর্তে তুমি শুধু matrix গুন করবে তাহলেই হবে। আমি তোমাদের পরামর্শ দিব তোমরা নিজেরাই এই জিনিসটা কোড কর। প্রথম দিকে যদিও কোড করতে একটু সমস্যা হবে কিন্তু দু একবার নিজে থেকে কোড করলে দেখবে অনেক সহজ হয়ে যাবে। Stirling Number ও এই পদ্ধতিতে বের করা যায়, তবে সেক্ষেত্রে $k \times k$ আকারের matrix লাগে। সুতরাং বুঝতেই পারছ যে k কে ছোট হতে হবে কিন্তু n অনেক বড় হতেই পারে। চেষ্টা করে দেখতে পার সমাধান করতে পার কিনা।

৩.২.৬ Inclusion Exclusion Principle

Counting এর ক্ষেত্রে অত্যন্ত গুরুত্বপূর্ণ একটি method হল Inclusion Exclusion Principle. মনে কর তোমাকে বলা হল, 1 হতে 100 পর্যন্ত কতগুলি সংখ্যা আছে যারা 2 বা 3 দ্বারা ভাগ যায়? খেয়াল কর, এক্ষেত্রে কিন্তু বিভিন্ন ধরনের সংখ্যা আছে, শুধু 2 দ্বারা ভাগ যায় এমন, কিছু আছে শুধু 3 দ্বারা ভাগ যায়, কিছু আছে 2 আর 3 দুইটি দিয়েই যায়, আবার কিছু আছে যা কোনটা দিয়েই ভাগ যায় না। শুধু 2 দ্বারা কত গুলি ভাগ করা যায় তা বের করা কিন্তু খুব সহজ, $\lfloor 100/2 \rfloor = 50$ একই ভাবে শুধু 3 দ্বারা যায় এরকম আছে $\lfloor 100/3 \rfloor = 33$. এখন এদের যোগ করলেই কিন্তু তোমাদের উত্তর পেয়ে যাবে না। কারণ, প্রথম গ্রুপে $6, 12, 18 \ldots$ এরকম কিছু সংখ্যা আছে যারা দ্বিতীয় গ্রুপেও আছে। সুতরাং আমরা যদি এদের যোগ করে দেই এই জিনিস গুলো double count হয়ে যাবে। একটা সহজ উপায় হল এই double count বের করে তা বিয়োগ করে দেয়া। খেয়াল করলে দেখবে, এই double count গ্রুপ এ তারাই আছে যারা 2 ও 3 দুইটি দ্বারাই ভাগ যায় অর্থাৎ 6 দ্বারা ভাগ যায়। 6 দ্বারা ভাগ যায় এরকম মোট $\lfloor 100/6 \rfloor = 16$টি সংখ্যা আছে। সুতরাং আমাদের ফলাফল হবেঃ $50 + 33 - 16 = 67$. এভাবে count করার method ই হল inclusion exclusion principle. যদি 2টিরও বেশি জিনিস থাকে তাহলে কি করতে হবে একটু চিন্তা করে দেখতে পার। মনেকর তোমাকে $2, 3, 5, 7$এরকম চারটি সংখ্যা

দেয়া হল। তাহলে একটি দ্বারা ভাগ যায় এরকম সংখ্যা যোগ করে, দুইটি দিয়ে ভাগ যায় এরকম সংখ্যা আবার বিয়োগ করবা, কিন্তু তিনটি সংখ্যা দ্বারা ভাগ যায় এরকম সংখ্যা আবার যোগ করবা এবং চারটি সংখ্যা দ্বারা ভাগ যায় সংখ্যা গুলি বিয়োগ করবা। অর্থাৎ, বিজোড় সংখ্যার সময় যোগ ও জোড় এর ক্ষেত্রে বিয়োগ করতে হয়। এই ধরনের সমস্যায় সাধারনত time complexity হয়ে থাকে $O(2^n)$. তোমাদের বলে রাখি এসব ক্ষেত্রে bitmask ব্যবহার করে কোড করলে অনেক সহজেই কোড হয়ে যায়। যদি তোমাদের কাছে n টা সংখ্যা থাকে এবং কোনটা কোনটা দিয়ে ভাগ করবা এটা সিদ্ধান্ত নিতে চাও তাহলে তুমি n bit এর বিভিন্ন নাম্বার নিবা, কোন একটি bit এ 1 থাকার মানে ঐ সংখ্যা তুমি নিবা, 0 মানে নিবা না। এরকম করে খুব সহজে একটা loop দিয়েই তুমি এই নেয়া-না নেয়ার কাজটা করে ফেলতে পার।

৩.৩ সম্ভাব্যতা

কোন কারণে আমরা ছোট বেলা থেকে এই জিনিসের প্রতি একরকম ভীতি নিয়ে বেড়ে উঠি। কারণ এই জিনিস বুঝতে আমাদের কষ্ট হয় বা আমাদের হয়তো ভালো মত বুঝানো হয় না। এখানে আসলে খুব details এ তোমাদের এসব জিনিস দেখানো সম্ভব না কিন্তু খুব অল্প কথায় কিছু লিখার চেষ্টা করলাম।

৩.৩.১ Probability

মনে কর ক্রিকেট খেলা শুরুর আগে দুই ক্যাপ্টেইন টস করতে গেল। টস এর সময় Head পরবে না Tail পরবে? যেকোনো একটা পরতে পারে! আমরা বলে থাকি chance 50-50. এখন মনে কর লুডু খেলায় একটা ছক্কা আছে, কিন্তু এই ছক্কায় কোন 5নেই তার পরিবর্তে আরও একটি 6 আছে। এখন তোমাকে যদি জিজ্ঞাসা করা হয় এখানে কত পরবে? তুমি কিন্তু বলতে পারবে না যে এখানে যেকোনো একটা পরতে পারে! কারণ এখানে কিন্তু 5 নেই। আবার তুমি একটু যদি mathematical উত্তর দাও তাহলে বলবে এখানে 5 কখনই পরবে না, 6 পরার সম্ভাবনা $1, 2, 3, 4$ পরার থেকে বেশি আর $1, 2, 3, 4$ পরার সম্ভাবনা একই। আমরা কিন্তু কোন হিসাব করে এ কথা বলতেসি না, শুধু common sense থেকেই এই কথা বললাম। তাই না? এখন যদি তোমাকে বলা হয় ঠিক মত হিসাব করে বের কর কোনটা পরার সম্ভাবনা কত? তখন আসে হিসাব নিকাশ!

সম্ভাব্যতা অংকের মূল নীতি হলঃ

$$\text{কোন ঘটনা ঘটার সম্ভাব্যতা} = \frac{\text{কত ভাবে এই ঘটনা ঘটতে পারে}}{\text{কত ভাবে সকল ঘটনা ঘটতে পারে}}$$

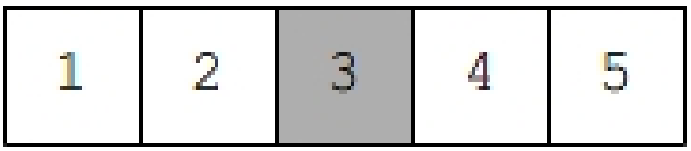
যেমন, আমাদের ক্রিকেট খেলার টস এ, Head বা Tail পরার probability হলঃ $\frac{1}{2}$ কারণ আমাদের মাত্র দুইভাবেই coin পরতে পারে Head ও Tail আর এদের মাঝে একটাই Head বা Tail. এখন ছক্কার ক্ষেত্রে যদি আমরা হিসাব করতে চাই 5 পরার probability কত, তাহলে কিন্তু $\frac{0}{6} = 0$. অর্থাৎ 5 পাবই না! আবার $1, 2, 3$ বা 4 পরার probability হল $\frac{1}{6}$ আর 6 পরার probability হল $\frac{2}{6}$. অর্থাৎ 6 পরার সম্ভাবনা কিন্তু অন্য কোন একটি পরার সম্ভাবনা থেকে বেশি।

তোমরা হয়তো π নির্ণয় করার নানা মজার মজার method দেখেছ। এমনই একটি method এখন বলব। তোমরা একটি বড় বর্গ আঁক। এর মাঝে একটি বৃত্ত আঁক যা চার বাহুকেই স্পর্শ করে। এখন তুমি ছোট ছোট কিছু জিনিস নাও (মনে কর n টা) ধর চুমকির মত জিনিস। এখন এগুলি বর্গক্ষেত্রের মাঝে সেগুলি ছড়িয়ে দাও। এখন তুমি গুনে দেখ বৃত্তের মাঝে কত গুলি আছে। ধরা যাক, b টা। তাহলে $\frac{4b}{n}$ হবে প্রায় π এর সমান। অদ্ভুত না? এমন কেন হল? সহজ কিন্তু। বৃত্তের radius যদি r হয় তাহলে বৃত্তের ক্ষেত্রফল πr^2 আর বর্গের ক্ষেত্রফল $4r^2$. সুতরাং তুমি যদি কোন একটি চুমকি randomly ফেল এর মাঝে তাহলে বৃত্তের মাঝে পরার সম্ভাবনা $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$. আবার আমরা আমাদের experiment এ n টা বিন্দু randomly ফেলেছিলাম এবং বৃত্তের মাঝে পরেছে b টা, সুতরাং আমরা experiment থেকে পাই বৃত্তের মাঝে একটি চুমকি পরার সম্ভাবনা $\frac{b}{n}$ অর্থাৎ $\frac{b}{n} \approx \frac{\pi}{4}$ এখান থেকেই আমরা পাই, $\pi \approx \frac{4b}{n}$. তুমি n যত বড় নিবে π এর মান তত ভালো পাবে।

৩.৩.২ Expectation

মনে কর তোমাকে বলা হল একটি coin টস করার পর যদি head পরে তাহলে তুমি 0 টাকা পাবে কিন্তু tail পরলে 100 টাকা পাবে। তুমি কত পাবার আশা কর? তুমি যদি বল যে আমি 100 টাকা পাবার আশা করি, তাহলে হল না, তুমি বেশি আশাবাদী হয়ে গেলে। কারণ তোমার 50% সম্ভাবনা আছে তুমি 0 টাকা জিতবে। আবার তুমি হতাশার সুরে যদি বল যে নাহ আমি একটা টাকাও পাবো না, আমার কপাল ভালো না। তাহলেও হল না। তুমি যদি একজন mathematician এর সুরে কথা বল তাহলে বলবে আমি 50 টাকা পাবার আশা করি। প্রথম প্রথম সবাই মনে করত পার এটা কেমন কথা? হয় আমি 0 পাবো নাহয় 100 পাবো, 50 কই থেকে আসল? কাহিনী হল, তুমি যদি এই টস ধর 10 বার করলে তাহলে এটা বলাই যায় যে 5 বার এর মত head পরবে আর বাকি 5 বার tail. অর্থাৎ তুমি 10 বারে মোট 500 টাকা পাবে, অর্থাৎ গড়ে তুমি 50টাকা পাবে। এই জন্যই এক্ষেত্রে তোমার expectation হল 50 টাকা। আরও একটি উদাহরন দেয়া যাক, ধর তুমি সাপ লুডু খেলতেছ তোমাকে যদি বলা হয় তোমার খেলা শেষ করতে কত চাল লাগতে পারে? তুমি কিন্তু হিসাব নিকাশ করে দেখাতে পারবে যে তোমার expected number of move কত। দেখা যাবে তুমি যদি বহুবার সাপ লুডু খেল তাহলে গড়ে ঐ সংখ্যক move লাগবে। কোন কিছুর expectation বের করার নিয়ম হচ্ছে যত রকম ঘটনা ঘটতে পারে তাদের probability গুন ঐ ঘটনা ঘটলে তোমার সেই কোন কিছু কত হবে। যেমন, coin টস এর ক্ষেত্রে তোমার দুইরকম ঘটনা ঘটতে পারে। head বা tail. head পরার probability 0.5 এবং এটি পরলে তুমি 0টাকা পাবে। আর যদি 0.5 probability তে tail পরে তাহলে তুমি পাবে 100টাকা। অতএব তোমার টাকা পাবার expectation হবে $0.5 \times 0 + 0.5 \times 100 = 50$.

আরও একটি উদাহরন হিসাবে ছোট পরিসরে সাপ লুডু বিবচনা করা যাক।



নকশা ৩.১: একটি ছোট লুডু খেলা

মনে কর তুমি 1 এ আছ। তুমি যদি 5 এ যাও তাহলেই খেলা শেষ হয়ে যাবে। মাঝের গাঢ় রঙ আলা 3 এ তুমি যেতে পারবে না কখনই। তোমাকে একটি ছক্কা দেয়া হল যেটা ছুড়লে 1 হতে 6 এর মাঝের কোন একটি সংখ্যা পরে এবং তুমি ওত সংখ্যক ঘর সামনে যেতে পারবে। কিন্তু সেই ঘর যদি 3হয় বা 5 পেরিয়ে যায় তাহলে আবারো তোমাকে চালতে হবে। Expected number of move কত?

ধরা যাক, T_i হল i এ থাকা কালীন সময়ে খেলা শেষ করার জন্য expected number of move. আমাদের T_1 বের করতে হবে। শেষ থেকে আসা যাক। $T_5 = 0$ কারণ তুমি যদি 5 এ থাকো তাহলে তো খেলা শেষ। কোন move না দিয়েই তোমার খেলা শেষ হয়ে যাবে। সে জন্য এটি শূন্য। এখন 4 এ আসা যাক। এখান থেকে খেলা শেষ করতে আমাদের লাগবে T_4 সংখ্যক move. খেয়াল কর, যদি 1 ব্যতিত কোন সংখ্যা পরে তাহলে কিন্তু তুমি 4 এই থাকবে, অর্থাৎ তোমার আরও T_4 সংখ্যক move লাগবে। ব্যপারটা আরও একটু পরিস্কার করা যাক, তুমি ধরেই নিয়েছ যে 4 থেকে খেলা শেষ করতে T_4 move লাগবে। এখন তোমার যদি এখানেই থাকতে হয় তাহলে তো T_4 move লাগবে তাই না? আর যদি 1 পরে তাহলে লাগবে T_5 move. অর্থাৎ $1/6$ probability তে লাগবে T_5 move আর $5/6$ probability তে লাগবে T_4 move. আর ভুলে যেও না এই মাত্র তুমি একটা চাল দিলে ছক্কা গড়িয়ে, সুতরাং $T_4 = 1 + \frac{1}{6}T_5 + \frac{5}{6}T_4$ এখান থেকে আমরা পাই, $T_4 = 6$. হিসাব টা কিন্তু ঠিকি আছে। ছক্কার ছয় দিক, আমরা আসা করতেই পারি যে 6 চালের মাঝে প্রতিটি সংখ্যা একবার না একবার আসবেই। সুতরাং আমাদের expectation 6. T_3 আমাদের লাগবে না, কারণ আমরা এখানে কখনই আসব না। এখন আসা যাক, 2 এ। যদি $1/6$ probability তে 2 পরে তাহলে লাগবে T_4 move, যদি $1/6$ probability তে 3 পরে তাহলে T_5 move লাগবে, আর বাকি $4/6$ probability তে যা পরবে তার জন্য আমাদের 2 এই থাকতে হবে অর্থাৎ T_2 move লাগবে। সুতরাং, $T_2 = 1 + \frac{1}{6}T_4 + \frac{1}{6}T_5 + \frac{4}{6}T_2$ অর্থাৎ $T_2 = 6$. আশা করি T_1 এর জন্য ফর্মুলা তুমি বুঝতে পারছ কি হবে, $T_1 = 1 + \frac{1}{6}T_2 + \frac{1}{6}T_4 + \frac{1}{6}T_5 + \frac{3}{6}T_1 \Rightarrow T_1 = 6$. অর্থাৎ expected number of

move হবে 6.

৩.৪ বিবিধ

৩.৪.১ Base Conversion

আমরা যেই সংখ্যা পদ্ধতি ব্যবহার করে থাকি তাকে দশমিক বলে কারণ এর base হল 10. কম্পিউটার যেই সংখ্যা পদ্ধতি ব্যবহার করে তার base হল 2 একে বলে বাইনারি। এরকম আরও কিছু বহুল প্রচলিত সংখ্যা পদ্ধতি আছে- অকটাল (base হল 8), হেক্সাডেসিমাল (base হল 16)। বলা হয়ে থাকে আমাদের হাতের দশ আঙ্গুলের জন্য আমাদের সংখ্যা পদ্ধতি হল Decimal বা দশমিক। কম্পিউটার এর জন্য 10 টি আলাদা আলাদা সংখ্যা নিয়ে হিসাব করা বেশ কষ্টকর এজন্য শুধু মাত্র voltage up down করে বুঝা যায় এমন একটি সংখ্যা পদ্ধতি ব্যবহার করা হয় কম্পিউটারে। এটিই হল বাইনারি। এই পদ্ধতিতে অঙ্ক আছে দুইটি 0 ও 1. আমরা কেমনে হিসাব করি একটু খেয়াল করঃ $0, 1, \ldots 9, 10, 11, 12, \ldots 19 \ldots$ অর্থাৎ আমাদের শেষ digit টা এক এক করে বাড়ে এর পর শেষ হয়ে গেলে এর বামের টা এক বাড়ে ওটাও শেষ হয়ে গেলে তারও বামের টা বাড়বে। বাইনারিও সে রকমঃ $0, 1, 10, 11, 100, 101, 110, 111, 1000 \ldots$। অন্যান্য number system এও একই রকম হয়ে থাকে।

এখন যদি একটু খেয়াল কর দশমিক number system এ 481 কে আমরা ভেঙ্গে লিখতে পারিঃ $4 \times 10^2 + 8 \times 10^1 + 1 \times 10^0$ একই ভাবে বাইনারি 1010 কেও আমরা এভাবে ভেঙ্গে লিখতে পারিঃ $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ যার মান দশমিক পদ্ধতি তে হবে 10. সুতরাং আমাদের যদি অন্য কোন number system এ একটি সংখ্যা দেয়া হয় তাকে আমরা খুব সহজেই আমাদের দশমিক number system এ পরিবর্তন করতে পারি। আমরা ডান থেকে iতম স্থান এ যাব এবং সেখানে থাকা অংককে $base^i$ দিয়ে গুন করে সবগুলি যোগ করলেই আমরা দশমিক পদ্ধতিতে সংখ্যাটি পেয়ে যাব।

কিন্তু আমরা যদি কোন একটি দশমিক সংখ্যাকে অন্য আরেকটি number system এ পরিবর্তন করতে চাই? ধরা যাক আমরা b base এ পরিবর্তন করতে চাই। তাহলে অবশ্যই আমাদের সংখ্যাটা হবে এরকমঃ $a_n \times b^n + \ldots + a_1 \times b^1 + a_0 \times b^0$ আমরা যদি এই সংখ্যা কে b দ্বারা ভাগ করি তাহলে ভাগশেষ হবে a_0 এবং ভাগ করার ফলে সব power গুলি কিন্তু 1 করে কমে গেছে, সুতরাং এর পরে আবারো b দ্বারা ভাগ করলে ভাগশেষ হবে a_1 এভাবে একে একে আমরা ডান থেকে বামের সব সংখ্যা পেয়ে যাব।

৩.৪.২ BigInteger

অনেক সময় দেখা যায় আমাদের প্রবলেম এ mod করতে বলা হয় না, আবার আমাদের উত্তরটাও বেশ বড় হয়। সেক্ষেত্রে আমাদের BigInteger ব্যবহার করতে হয়। যারা Java জানো তাদের জন্য এটা একটা advantage কারণ Java তে BigInteger নামে একটি library আছে। কিন্তু মাঝে মাঝে এটি বেশ Slow হওয়ায় দেখা যায় TLE খেতে হয়। আমরা কিন্তু খুব সহজেই C তে নিজেদের BigInteger এর হিসাব নিকাশের জন্য function লিখে ফেলতে পারি। যোগ বিয়োগ ও গুন বেশ সহজ। ভাগ একটু কঠিন। খেয়াল করলে দেখবে যে, আমরা যোগ বিয়োগ বা গুন সব সময় ডান দিক থেকে করি, এবং এই সময় যেই দুইটি সংখ্যা নিয়ে হিসাব করছি তাদের কে ডান দিকে একই বরাবর রাখা হয়, আমরা কিন্তু বাম দিকে একই বরাবর রাখি না, রাখলে ভুল হবে। কিন্তু আমরা যখন কম্পিউটারে array এর চিন্তা করছি তখন আমরা কল্পনা করি বাম দিক থেকে। এটা অনেক সময় সমস্যা হয়। যেমন মনে করা যাক আমরা 100 digit এর একটি সংখ্যার সাথে 50 digit এর একটি সংখ্যা যোগ করব। এখন এই দুইটি সংখ্যা যখন string আকারে input নিব তখন এটা বাম দিকে align হয়ে থাকে। এই অবস্থায় কোন হিসাব করা বেশ কঠিন। এ জন্য যা করা উচিত তাহল, সংখ্যাকে উলটিয়ে নেয়া, এতে করে সখ্যার একক এর অংক সবসময় আমাদের বামে থাকে, আর যেহেতু আমরা কম্পিউটারে সংখ্যাকে বাম align করে চিন্তা করছি সেহেতু আর কোন সমস্যা হবে না। আমরা এখন বাম দিক থেকে ডান দিকে যাব আর হিসাব করব। এসময় আরও একটি গুরুত্বপূর্ণ জিনিস খেয়াল রাখলে ভালো হয় যে, আমরা যেহেতু জানি না যে আমাদের উত্তরটা কত গুলি digit হবে সেহেতু আমাদের result রাখার array কে 0 দ্বারা initialize

করে নিতে হবে। এতে করে যোগ বিয়োগ বা গুন করতে কোন সমস্যা হবে না। এখন কথা হল কেমনে যোগ বিয়োগ গুন করব? খুবই সোজা, তুমি যেভাবে কর সেভাবে। তুমি নিজে একটু ভেবে দেখ কেমনে যোগ বিয়োগ কর? তুমি যোগ করার পর যোগফল 10 বা এর বড় হলে হাতে কিছু একটা থাকে, সেটা গিয়ে পরের ঘরের সাথে যোগ কর এভাবে চলতে থাকে। আবার বিয়োগ এর সময় তুমি কিছু ধার নাও যেটা পরে গিয়ে আবার শোধ করে দাও। ঠিক এই জিনিসগুলিই তোমাদের লজিকের মাধ্যমে লিখতে হবে। গুনের ক্ষেত্রে তোমাদের একটু ঝামেলা মনে হতে পারে। তোমরা একটু চিন্তা করে দেখ আমরা যে গুন করে সঙ্খ্যাগুলি পর পর লিখি এর পর যোগ করি তা না করে, যদি গুন করতে করতে যোগ করি? এটা আমাদের হাতে হাতে করতে বেশ কষ্ট হবে, কিন্তু কম্পিউটারে এই প্রোগ্রাম লিখা বেশ সহজ হবে।

৩.৪.৩ Cycle Finding Algorithm

UVa 11036 প্রবলেমটা দেখতে বেশ কঠিন হলেও এর idea টা কিন্তু খুব সহজ। একটা x এর ফাংশন দেয়া থাকবে যেমন ধরা যাক, $f(x) = x * (x + 1) mod 11$, এখন একটি n এর মানের জন্য $f(n), f(f(n)), f(f(f(n)))\ldots$ এই ধারার period বের করতে হবে। অর্থাৎ কত length বার বার repeat হবে। যেমন যদি $n = 1$ হয় তাহলে এই ধারার মান গুলি হবেঃ $2, 6, 9, 2, 6, 9, 2, 6, 9\ldots$ এখানে তিনটি সংখ্যা বার বার পুনরাবৃত্তি করছে, সুতরাং এখানে period হবে 3. একটু চিন্তা করলে দেখবে যে এখানে আসলে কখনও যদি আগের একটি সংখ্যা আসে, তাহলে এর পরের সংখ্যা গুলি আগের মত আসতে থাকবে। যেমন উপরের ধারায় চতুর্থ পদে 2 চলে এসেছে যা প্রথম পদের সমান, সুতরাং এর পঞ্চম পদ হবে দ্বিতীয় পদের সমান এবং এভাবে পর পর একই সংখ্যা আসতে থাকবে। আমাদের আসলে বের করতে হবে প্রথম repeatation কখন হবে। এই জিনিসটা একটা array রেখে খুব সহজেই করা যায়। আমাদের একটি একটি করে মান বের করব আর array তে দেখব যে এই জিনিসটা আগে এসেছিল কিনা। যদি না আসে তাহলে আমরা পরবর্তীর জন্য array তে লিখে রাখব যে এই সংখ্যাটা ধারার অমুক position এ এসেছিল। আর যদি আগেই এসে থাকে তাহলে, আগে কোন জায়গায় এসেছিল তা আমরা array থেকেই দেখতে পারব, আর এখন কোন পদে আসলাম তাও আমরা জানি। এই দুই থেকে আমরা এর period বের করে ফেলতে পারি। কিন্তু এভাবে array ব্যবহার করে করতে গেলে আমাদের memory complexity দাঁড়ায় $O(n)$ যদিও আমাদের time complexity ও $O(n)$ বা আরও সুনির্দিষ্ট ভাবে বলতে গেলে, $O(\lambda+\mu)$ যেখানে $\mu =$ শুরু থেকে cycle এর শুরু পর্যন্ত দুরত্ব এবং $\lambda =$ cycle এর length. আমরা চাইলে memory complexity কমিয়ে $O(1)$ এ নামাতে পারি এবং সেক্ষেত্রেও আমাদের time complexity একই থাকবে। এই method কে বলা হয় Floyd's Cycle Finding Algorithm.

এই algorithm টা বেশ মজার। ধারার শুরুতে একটি খরগোশ আর একটি কচ্ছপ রাখতে হবে। এর পর প্রতি ধাপে খরগোশ দুই ধাপ আর কচ্ছপ এক ধাপ করে যাবে। এক সময় না এক সময় দুজনে মিলিত হবেই। এখন খরগোশকে ঐ জায়গাতে রেখেই কচ্ছপ কে একধাপ এক ধাপ করে আগাতে হবে যতক্ষণ না সে আবার খরগোশ এর জায়গায় ফেরত আসে। যত ধাপে সে ফেরত আসে সেটাই হল period বা λ. এবার কচ্ছপ কে ঐ জায়গা তে রেখে খরগোশ কে আবার শুরুতে নিয়ে যাও তবে এবার খরগোশ আর কচ্ছপ দুজনই একধাপ একধাপ করে আগাবে যতক্ষণ না একত্র হয়। এটা খুব সহজেই প্রমাণ করা যায় যে যে কয় ধাপে মিলিত হল সেটাই μ.

অধ্যায় ৪

Sorting ও Searching

৪.১ Sorting

Sort করার অর্থ হল একটি নির্দিষ্ট ক্রমে সাজানো। আমরা ধর পরীক্ষার খাতা 1 রোল হতে 60 রোল পর্যন্ত যদি সাজাই তাহলে এটাকে sorting বলে। প্রায়ই আমাদের বিভিন্ন সংখ্যা sort করার প্রয়োজন হয়। শুধু সংখ্যা নয়, string, co-ordinate এরকম নানা কিছু sort করতে হতে পারে। আমরা এই সেকশনে কিছু sorting algorithm দেখব।

৪.১.১ Insertion Sort

সাধারনত আমরা real life এ এই ভাবে sorting করে থাকি। মনে কর আমাদের কাছে 60 জনের খাতা আছে। আমরা একটা করে খাতা নেই, আর sorted খাতা গুলির মাঝে এই খাতাকে সঠিক জায়গায় রাখি। আবার নতুন খাতা নিব আর ঠিক করে রাখা খাতা গুলির মাঝে এই নতুন খাতাকে ঠিক জায়গায় রাখব। এভাবে সবগুলি খাতাকে রাখা শেষ হলেই আমাদের sorting ও শেষ হয়ে যাবে। এখন যদি implementation এর কথা চিন্তা কর তাহলে মনে হবে এভাবে একটা একটা করে খাতা ঢুকানো মনে হয় কঠিন কাজ। কিন্তু ওত কঠিন না। মনে কর তোমার $1 \dots i-1$ খাতা গুলি sorted আছে, তুমি i তম খাতা ঢুকাবে, তুমি প্রথমে দেখ $i-1$ এর খাতাটা কি তোমার থেকে ছোট? তাহলে যেখানে আছে সেখানেই তোমার খাতার position আর যদি না হয় তাহলে $i-1$ এ থাকা খাতাকে i এ আনো আর এবার $i-2$ এর সাথে চেক কর। এভাবে একে একে চেক করতে থাক। একটা উদাহরন টেবিল ৪.১ এ দেয়া হল।

এর কোড টাও কিন্তু বেশ ছোট। কিন্তু কোড করতে সোজা হলেও এই algorithm এর time complexity $O(n^2)$. আমরা পরে দেখব এর থেকেও অনেক দ্রুত sorting করা সম্ভব। Insertion Sort এর প্রোগ্রাম কোড 8.1 এ দেয়া হলঃ

Listing 8.1 : insertion sort.cpp

```
for(i = 1; i <= n; i++)
{
        x = num[i];
        for(j = i - 1; j >= 1; j--)
                if(num[j] > x)
                        num[j + 1] = num[j];

        num[j + 1] = x;
}
```

সারনী ৪.১: Insertion Sort এর simulation

5, 8, 6, 1, 7, 9
5, 8, 6, 1, 7, 9
5, 8, 6, 1, 7, 9
5, 6, 8, 1, 7, 9
5, 6, 8, 1, 7, 9
5, 6, 1, 8, 7, 9
5, 1, 6, 8, 7, 9
1, 5, 6, 8, 7, 9
1, 5, 6, 8, 7, 9
1, 5, 6, 7, 8, 9
1, 5, 6, 7, 8, 9

৪.১.২ Bubble Sort

$O(n^2)$ এ যেসকল sorting algorithm আছে তাদের মাঝে Insertion Sort আমার সবচেয়ে সহজ মনে হলেও কোন কারণে অন্যরা bubble sort কে সহজ মনে করে থাকে। এই algorithm টাও বেশ মজার। তুমি প্রথম থেকে শেষ পর্যন্ত দুটি দুটি করে সংখ্যা নিতে থাকো। যদি দেখ আগেরটা পরেরটা থেকে বড় তাহলে swap কর। এই কাজ n বার কর। এই algorithm টা কিন্তু $O(n^2)$ সময় লাগবে। কারণ প্রথম বার যখন তুমি বাম থেকে ডানে যাবে তখন সবচেয়ে বড়টা অবশ্যই একদম ডান মাথায় গিয়ে পৌঁছাবে। পরের বারে দ্বিতীয় বড়টা ঠিক জায়গায় যাবে, এরকম প্রতিবারে একটি একটি করে সংখ্যা সঠিক স্থানে যাবে। তাই n বার এই কাজ করলে সব সংখ্যা সঠিক জায়গায় চলে যাবে। এই প্রোগ্রামের কোড 8.2 এ দেয়া হল। তোমরা চাইলে এই কোডে কিছু optimization করতে পার। যেমন, এমন হতে পারে যে n বার এর আগেই array টা sorted হয়ে গেছে সেই ক্ষেত্রে তুমি আগেই loop থেকে বের হয়ে যেতে পার। আবার প্রতিবার তোমার সকল pair চেক করার দরকার নাই কিন্তু। কারণ i বার চললে তুমি জানো যে শেষ i টা সংখ্যা ঠিক জায়গায় চলে গিয়েছে। তুমি এভাবে কিছু optimization করতে পার। কিন্তু যতই optimization কর না কেন এই algorithm এর worst case এ $O(n^2)$ সময়ই লাগবে। তোমরা কি সেই worst case টা বের করতে পারবে? [১]

Listing 8.2 : bubble sort.cpp

```
for(i = 1; i <= n; i++)
{
        for(j = 1; j < n; j++)
                if(num[j + 1] > num[j])
                {
                        temp = num[j];
                        num[j] = num[j + 1];
                        num[j + 1] = temp;
                }
}
```

[১]উত্তর টা হল যদি array টা প্রথমেই বড় থেকে ছোটতে সাজানো থাকে তাহলে তোমার $O(n^2)$ সময় লাগবেই।

৪.১.৩ Merge Sort

এই algorithm টা বেশ মজার। এর সাথেও আমাদের practical life এর কিছু মিল আছে। আবার আমরা সেই খাতা sorting এ ফিরে যাই। মনে কর তোমার কাছে 60টা খাতা আছে। তুমি এই খাতা কে দুইভাগ করে দুজন কে দিয়ে দিলে। তারা তাদের দুইভাগ sort করে তোমাকে দিল। এখন তুমি ঐ দুইটা sorted খাতার ভিতরে না দেখে শুধু উপরে দেখবে, যারটা ছোট তুমি সেই ভাগ থেকে খাতা নিবে এভাবে তুমি ছোট থেকে বড় সাজিয়ে ফেলবে। খুবই সোজা তাই না? এখন ঘটনাটায় আরেকটু পেঁচ লাগবে, তাই আগানোর আগে আমি যা বললাম তা ভালো করে কল্পনা করে নাও। এখন প্রশ্ন হল তুমি যেই দুইজন কে দিলে তারা কেমনে sort করবে? উত্তরটা হল আবারো দুই ভাগ করে। অর্থাৎ তুমি যেভাবে দুই জনকে ভাগ করে দিয়েছ, তারাও আবার দুইভাগ করে দুই জন কে দিবে এবং তাদের কাছ থেকে পাবার পর তারা ওটাকে সাজিয়ে তোমাকে দিবে। এবং এভাবেই চলতে থাকবে। এই কাজ করতে হবে recursive function এর মাধ্যমে। এই function কে তুমি যদি একটা array দাও সে একে দুই ভাগ করবে এবং আবার এই function কে call করে তাদের মাধ্যমে ঐ দুই ভাগ sort করে আনবে। এরপর এদের কে merge করে পুরো sorted ফলাফল কে সে return করবে। এই merge sort এর প্রোগ্রামটা কোড 8.3 এ দেয়া হল। খেয়াল কর যে, mergesort ফাংশনটি দুইটি variable কে parameter হিসাবে নেয়, lo এবং hi. এই দুই variable মূল array এর দুই মাথা নির্দেশ করে। এই ফাংশন এর কাজ হল এই দুই মাথার মাঝের অংশটুকু sort করা।

Listing 8.3 : merge sort.cpp

```
int num[100000], temp[100000];

//call mergesort(a, b) is you want to sort num[a...b]
void mergesort(int lo, int hi)
{
        if(lo == hi) return;
        int mid = (lo + hi)/2;

        mergesort(lo, mid);
        mergesort(mid + 1, hi);

        int i, j, k;
        for(i = lo, j = mid + 1, k = lo; k <= hi; k++)
        {
                if(i == mid + 1)                        temp[k] =
                    num[j++];
                else if(j == hi + 1)            temp[k] = num[i++];
                else if(num[i] < num[j])        temp[k] = num[i++];
                else                                            temp
                    [k] = num[j++];
        }

        for(k = lo; k <= hi; k++) num[k] = temp[k];
}
```

এই কাজ করার জন্য সে এই অংশ কে দুই ভাগে ভাগ করে এবং আবার mergesort ফাংশন call করে। যখন সে call করা ফাংশন থেকে ফেরত আসে তখন তার কাজ হল ঐ দুই অংশ merge করা। সে ঐ দুই অংশের শুরু থেকে দেখা শুরু করে। যেটা ছোট সেটাকে temp নামের array তে নিয়ে রাখে, এভাবে একে একে সব সংখ্যাকে temp নামের array তে সাজিয়ে রাখে। সবশেষে temp array এর সব সংখ্যা মূল array তে কপি করে দেয়। ফলে মূল array এর ঐ অংশটুকু sorted হয়ে যায়। আর base case টা কি হবে আসা করি বুঝতে পারছ।

এখন প্রশ্ন হল এই algorithm এর time complexity কত? ধরা যাক, আমাদের n সাইজ এর একটি array কে sort করতে বলা হয়েছে আর এর জন্য আমাদের সময় লাগে $T(n)$. আমাদের এই ফাংশন প্রথমেই n টি জিনিসকে সমান দুইভাগে ভাগ করে এবং তাদের উপর এই algorithm

চালায়। সুতরাং এই দুই ভাগ এর জন্য আমাদের সময় লাগবে $2T(\frac{n}{2})$. এখন আসা যাক আমাদের merge অংশে। আমাদের এক ভাগে আছে $n/2$টি সংখ্যা অন্যভাগেও আছে $n/2$ টি সংখ্যা। আমরা প্রতিবার এই দুইভাগের শুধু মাথার সংখ্যা চেক করে দেখি আর যেটি কম তাকে temp এ নেই। এভাবে চলতে থাকে। এই কাজটা কিন্তু n বার হবে কারণ প্রতিবার আমরা একটা করে সংখ্যা সরাচ্ছি। যদি n বার সরাই তাহলে সব সংখ্যা সরে যাবে। এই n বার শুধু আমরা দেখি যে কোনটা ছোট। সুতরাং এই পুরো কাজটার জন্য আমাদের লাগে $O(n)$ সময়। অর্থাৎ,

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \\
&= 2[2T(\frac{n}{4}) + \frac{n}{2}] + n \\
&= 4T(\frac{n}{4}) + 2n \\
&= 8T(\frac{n}{8}) + 3n \\
&\approx 2^{\log n} T(\frac{n}{n}) + n \log n \\
&\approx nT(1) + n \log n \\
&\approx n \log n
\end{aligned}$$

অর্থাৎ এই পদ্ধতিতে আমাদের time complexity হল $O(n \log n)$.

৪.১.৪ Counting Sort

তোমরা হয়তো অনেকেই শুনে থাকবে যে, sorting এর জন্য $O(n \log n)$ এর থেকে ভালো algortihm সম্ভব না এবং এই কথা সত্যি। কিন্তু তোমরা যেসব সংখ্যা sort করবে সেসব যদি non negative integer হয় এবং এদের সবচেয়ে বড় সংখ্যা যদি N হয় তাহলে counting sort এর time complexity হবে $O(N)$. এই algorithm টি বেশ সহজ। তোমাকে একটি auxiliary array নিতে হবে। তোমাকে যেসব সংখ্যা দিবে তাদের count এই auxiliary array তে এক বাড়াতে হবে। এরপর তুমি ঐ array এর 1 হতে N পর্যন্ত একটা loop চালাবা আর দেখবে i তম সংখ্যা কত গুলি আছে। ততগুলি i তুমি output array তে রাখবে। তাহলে এই output array তে তোমাকে দেয়া সব সংখ্যা sorted আকারে পাওয়া যাবে।

৪.১.৫ STL এর sort

আমাদের জীবনকে সহজ করার জন্য লাইব্রেরী ফাংশন হিসাবে sort নামে একটি ফাংশন দিয়ে দেয়া হয়েছে। এই ফাংশন ব্যবহার করতে হলে আমাদের STL এর algorithm নামক header file কে include করতে হবে। এখন আমরা যদি num নামক array এর 0 হতে $n-1$ পর্যন্ত sort করতে চাই, তাহলে আমাদের লিখতে হবেঃ sort(num, num + n). আমরা যদি 1 হতে n পর্যন্ত sort করতে চাই তাহলে আমাদের লিখতে হবেঃ sort(num + 1, num + n + 1). অর্থাৎ আমরা যদি a হতে b পর্যন্ত sort করতে চাই তাহলে আমাদের লিখতে হবেঃ sort(num + a, num + b + 1). এসব ক্ষেত্রে কিন্তু সবসময় ছোট হতে বড় তে sort হবে। যদি আমরা কোন একটি vector V কে sort করতে চাই তাহলে আমাদের লিখতে হবে, sort(V.begin(), V.end()).

এবার একটু কঠিন কাজ করা যাক। মনে কর আমাদের 2 dimension এ কিছু point দেয়া হল। এসব point এর x ও y co-ordinate আমাদের দেয়া আছে। আমাদের এমন ভাবে সাজাতে হবে যেন যাদের x ছোট তারা আগে থাকে, যদি কোন দুইটি বিন্দুর x সমান হয় তাহলে যেন যার y ছোট সে আগে থাকে। আমরা এই point এর x ও y সংরক্ষনের জন্য একটি structure ব্যবহার করব। অর্থাৎ আমাদের কাছে কোন একটি structure এর array আছে, আমাদের কে এই জিনিস sort করতে হবে। মনে কর আমাদের structure টার নাম Point এবং array টার নাম point. এখন তুমি যদি

শুধু sort(point, point + n) লিখ তাহলে কিন্তু হবে না। কারণ দুইটি Point এর মাঝে কোনটি ছোট তা কিন্তু কম্পিউটার এমনি এমনি বুঝবে না। তাকে বলে দিতে কি কি শর্ত মানলে আমরা বলতে পারি যে একটি Point আরেকটি Point এর থেকে ছোট। এ জন্য আমাদের একটি ফাংশন লিখতে হবে, ধরা যাক এই ফাংশন এর নাম cmp. এই ফাংশনের কাজ হল তাকে যদি দুইটি Point দেয়া হয়, যদি প্রথম Point দ্বিতীয় Point এর থেকে ছোট হয় তাহলে এই ফাংশন true return করবে আর যদি বড় বা সমান হয় তাহলে false return করবে। এখানে খুব ভাল করে খেয়াল রাখতে হবে যে, কোন ক্রমেই যেন, cmp ফাংশন cmp(A, B) ও cmp(B, A) উভয় ক্ষেত্রেই true না দেয়। আমরা কোড 8.4 এর মত করে এই ফাংশনটা লিখতে পারি।

Listing 8.4 : cmp.cpp

```
1  bool cmp(Point A, Point B)
2  {
3          if(A.x < B.x) return 1;
4          if(A.x > B.x) return 0;
5
6          if(A.y < B.y) return 1;
7          if(A.y > B.y) return 0;
8
9          return 0;
10 }
```

এই ফাংশনকে চাইলে আমরা আরও সংক্ষেপে কোড ?? এর মত করেও লিখতে পারি।

Listing 8.5 : improved cmp.cpp

```
1  bool cmp(Point A, Point B)
2  {
3          if(A.x != B.x) return A.x < B.x;
4          return A.y < B.y;
5  }
```

এরকম ফাংশনের উপস্থিতিতে তোমাকে sort করার জন্য লিখতে হবেঃ sort(point, point + n, cmp). অপারেটর overload করেও এই কাজ করা যায়। তবে সমস্যা হল, একবারই অপারেটর overload করা যায়। সুতরাং তুমি যদি একই ধরনের জিনিসকে যদি বিভিন্ন ভাবে sort করতে চাও তা সম্ভব হবে না। কোড 8.6 এ অপারেটর overload কেমনে করতে হয় তা দেখানো হল। এই ক্ষেত্রে তুমি sort(point, point + n) লিখলেই হবে।

Listing 8.6 : operator overload.cpp

```
1  struct Point
2  {
3          int x, y;
4  }point[100];
5
6  bool operator<(Point A, Point B)
7  {
8          if(A.x != B.x) return A.x < B.x;
9          return A.y < B.y;
10 }
```

একটু চিন্তা করে দেখত একটা integer এর array কে যদি বড় থেকে ছোট আকারে সাজাতে চাইলে কি করবে? [১]

[১]একটি ফাংশন declare করে করতে পার। অথবা চাইলে functional হেডার ফাইল ব্যবহার করেও করতে পারঃ sort(num, num + n, greater<int>).

অনেকে string sort করা নিয়ে বেশ ঝামেলায় পরে, কিন্তু STL এর string আমাদের জীবনকে অনেক সহজ করে দিয়েছে। কোড 8.7 এ আমরা কিছু string ইনপুট নিয়ে তা sort করা দেখালাম।

Listing 8.7 : string sort.cpp

```
#include<stdio.h>
#include<string>
#include<algorithm>
#include<vector>
using namespace std;

int main()
{
        int n, i;
        char s[100];
        vector<string> V;

        scanf("%d", &n);
        for(i = 0; i < n; i++)
        {
                scanf("%s", s);
                V.push_back(s);
        }

        sort(V.begin(), V.end());

        return 0;
}
```

৪.২ Binary Search

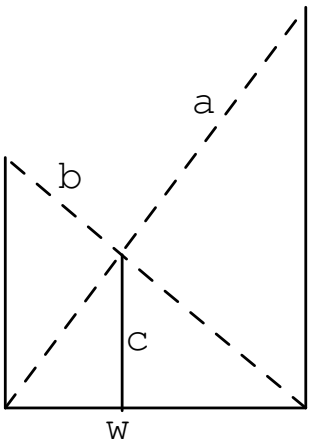
ধর তুমি একটা Game Show তে আছ। তোমার সামনে মোট 100টি বাক্স। এখন প্রতিটি বাক্সে একটি করে সংখ্যা থাকবে। তবে প্রথম বাক্সের সংখ্যা দ্বিতীয় বাক্সের সংখ্যার থেকে ছোট, দ্বিতীয় বাক্সের সংখ্যা তৃতীয় বাক্সের সংখ্যার থেকে ছোট এরকম করে আগের বাক্সের ভিতরে থাকা সংখ্যা পরের বাক্সের থেকে সবসময় ছোট হবে। এখন তোমাকে বের করতে হবে কোন বাক্সে 1986 আছে। এজন্য তুমি একটি একটি করে সব বাক্স খুলে দেখতে পার যে কোন বাক্সে 1986 আছে কিন্তু এক্ষেত্রে তোমাকে অনেক বাক্স খুলতে হবে, তোমার সময়ও বেশি লাগবে। কিন্তু তুমি যদি একটু বুদ্ধি খাটাও তাহলে হয়তো সব বাক্স না খুলেও বের করতে পার যে কই 1986 আছে। তুমি ঠিক মাঝের বাক্সটা খুল। যদি দেখ এটাই 1986 তাহলে তো হয়েই গেল। আর যদি দেখ এখানে 1986 এর থেকে বড় সংখ্যা আছে তার মানে তোমার সংখ্যা বামের অর্ধেক এ আছে আর নাহলে ডানের অর্ধেকে আছে। খেয়াল কর, তুমি এক ধাক্কায় 100 বাক্স থেকে 50 বাক্স কে কিন্তু বাদ দিয়ে ফেলতে পারতেস। একই ভাবে তুমি এই বাকি অর্ধেক কেও কিন্তু অর্ধেক করে ফেলতে পারবে। এরকম করতে করতে তুমি এক সময় 1986 খুব কম বাক্স খুলেই বের করে ফেলতে পারবে। তুমি কি বের করতে পারবে তোমার কত গুলি বাক্স খুলতে হবে? [১] এভাবে খুঁজার method কে আমরা binary search বলে থাকি। অনেক সময় এটি bisection method নামেও পরিচিত।

আরও একটি উদাহরন দেয়া যাক, মনে কর একটি array তে শুধু 0 ও 1 আছে। সব 0 সব 1 এর আগে থাকবে। তোমাকে প্রথম 1 খুঁজে বের করতে হবে। এটাও কিন্তু binary search দিয়ে করতে পার। তুমি মধ্য খানে গিয়ে দেখবে এটা 0 নাকি 1, যদি 0 হয় তাহলে তো এর ডানে থাকবে তোমার কাঙ্খিত জায়গা, আর যদি 1 হয় তাহলে এটা সহ বামে থাকতে পারে। এভাবে তুমি খুজতে থাকবে।

Binary Search ব্যবহার করে কিছু অদ্ভুদ সমস্যাও সমাধান করা যায়। অদ্ভুদ বললাম এই কারণে যে, প্রবলেম দেখে হয়তো কখনই মনে হবে না যে এখানে binary search ব্যবহার করা যায়, কিন্তু যায়! যেমন 8.১ নং চিত্রে একটা w প্রস্থের রাস্তার দুদিকে দুইটি দালান আছে। এখন রাস্তার এক মাথায়

[১] উত্তর কিন্তু মাত্র 7টি বাক্স :)

একটা মই রেখে অপর মাথা রাস্তার অন্য পারের দালানের মাথায় রাখা হল, একই ভাবে রাস্তার অন্য পাশ থেকেও আরেকটি মই রাখা হল। মই দুটির দৈর্ঘ্য a ও b. মই দুটি রাস্তা থেকে c উচ্চতায় ছেদ করে। a, b এবং c এর মান দেয়া আছে, $w = ?$.



নকশা ৪.১: c = ?

তোমরা যদি এখানে বিভিন্ন সুত্র খাটাও দেখবে খুব একটা সহজে কোন ফর্মুলা পাবে না w নির্ণয়ের জন্য। কিন্তু একটু অন্য ভাবে চিন্তা কর। তুমি যদি w এর মান জানো তাহলে কি তুমি c এর মান বের করতে পারবে? যেহেতু রাস্তার প্রস্থ দেয়া আছে আর আমরা মই এর দৈর্ঘ্য ও জানি সেহেতু আমরা প্রথমে দালান দুইটির উচ্চতা বের করি (পিথাগোরাস এর উপপাদ্য ব্যবহার করে)। ধরা যাক উচ্চতা দুইটি হল p ও q. এখন তোমরা যদি সদৃশকোনী ত্রিভুজের সুত্র খাটিয়ে আমরা দেখাতে পারি, $c = \frac{1}{\frac{1}{p}+\frac{1}{q}}$. অর্থাৎ আমরা যদি w এর মান জানি তাহলে c এর মান বের করে ফেলতে পারি। কিন্তু উল্টোটা কিন্তু কঠিন। ধরা যাক আমাদের দেয়া c এর মানের জন্য উত্তরটা হবে w'. যদি তুমি w এর মান হিসাবে w' এর থেকে বড় মান guess কর তাহলে, c এর মান প্রদত্ত মানের থেকে কম পাবে, আবার উল্টো ভাবে যদি তুমি w এর মান হিসাবে w' এর থেকে ছোট মান guess কর তাহলে c এর মান প্রদত্ত মানের থেকে বড় পাবে। কেবল w এর মান w' হলেই সঠিক দিবে। সুতরাং তোমরা w এর মানের উপর binary search চালাবে আর দেখবে c এর মান প্রদত্ত মানের থেকে বড় না ছোট সেই অনুসারে w এর মানের range ও পরিবর্তন করবে।

৪.৩ Backtracking

Backtracking কোন algorithm নয়, এটি একটি সাধারণ সল্ভিং method. আমরা যা সাধারণ ভাবে বুঝে থাকি তাই কোড করাটাই হল Backtracking. কিছু উদাহরন দিলে জিনিসটা পরিস্কার হবে।

৪.৩.১ Permutation Generate

মনে কর তোমাকে বলা হল 1 হতে n এর সকল permutation প্রিন্ট কর। যেমন $n = 3$ হলে তোমাকে $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2)$ এবং $(3, 2, 1)$ প্রিন্ট করতে হবে। এখন এই কাজ শুধু মাত্র for-loop দিয়ে করা কঠিন। খেয়াল কর, আমাদের n কিন্তু কত সেটা শুরুতে বলে দেয়া নায়। হয়তো বলা থাকবে যে, $n \leq 10$. যদি বলত যে $n = 3$ বা $n = 4$ তাহলে হয়তো আমরা $3, 4$টি nested loop লিখে কাজটা করতাম, কিন্তু যখন n বড় হয়ে যায় তখন এত বড় জিনিস লিখা আমাদের জন্য কষ্টকর যেমন তেমনি প্রতিটি n এর জন্য আমাদের আলাদা আলাদা করে হয়তো কোড

লিখতে হবে। শুধু loop দিয়ে করা একেবারে অসম্ভব না কিন্তু আমরা যেভাবে চিন্তা করতে অভ্যস্ত তাতে করে শুধু loop ব্যবহার করে আমাদের জন্য এই কাজ করা কঠিন হয়ে দাঁড়ায়। চিন্তা করে দেখ তোমাকে হাতে হাতে যদি এই কাজ করতে দেয়া হয় তুমি কেমনে করবে?

যদি তুমি কোন systematically না করে একে একে লিখতে থাকো নিজের ইচ্ছা মত তাহলে n এর মান বড় হলে এক সময় দেখবে যে আর কি কি বাকি আছে তা বের করা বেশ কঠিন কাজ হয়ে যাবে। এখন systematic উপায়টা কি? মনে কর $n = 3$ এর জন্য তুমি যা করবে তা হল, তুমি দেখবে কোন কোন সংখ্যা এখনো বসানো হয় নাই, এদের মাঝে সবচেয়ে ছোটটা (1) নিয়ে প্রথম জায়গায় বসাবা। এখন বাকি সংখ্যাগুলি থেকে যেটি ছোট (2) সেটি দ্বিতীয় ঘরে বসাবা। একই ভাবে তৃতীয় ঘরেও বসাও (3). আমরা $(1, 2, 3)$ পেয়ে গেলাম। এখন এর immediate আগে যা বসিয়েছ তা মুছে ফেল, অর্থাৎ এর আগে যে আমরা 3 বসিয়ে ছিলাম তা সরাও। এখন দেখ next কোন সংখ্যা এখনো বসানো হয় নাই। আসলে এই ক্ষেত্রে 3 এর পর আর কোন সংখ্যা বাকি নেই, তাহলে এর আগে যেই সংখ্যা বসিয়েছিলে তা মুছো অর্থাৎ আমাদেরকে 2 মুছতে হবে। একই ভাবে আমরা দেখব 2 এর পর কোন সংখ্যাটা এখনো বসানো হয় নাই (3) তাকে বসিয়ে পরের ঘরে (তৃতীয় ঘরে) যাও। এখন দেখ কোন সবচেয়ে ছোট সংখ্যা এখনো বসানো হয় নাই (2) তাকে বসাও। আমরা শেষ প্রান্তে চলে এসেছি এবং আরও একটি permutation $(1, 3, 2)$ আমরা পেয়ে গেলাম। এবার আমরা আবার immediate আগের সংখ্যা মুছে ফেলি (2). এক্ষেত্রে আর বসানোর মত কোন সংখ্যা বাকি নেই, সুতরাং আরও একধাপ আগে চলে যাই আর 3 কে মুছে ফেলি। এখন 3 এর থেকে বড় কোন সঙ্খ্যাও বাকি নেই সুতরাং আরও এক ধাপ পিছে গিয়ে 1কে মুছে next বড় সংখ্যা 2 বসাই। দ্বিতীয় ঘরে এসে সবচেয়ে ছোট সংখ্যা 1 বসাই ও তৃতীয় ঘরে এসে 3 বসিয়ে আমরা $(2, 1, 3)$ পাবো। এভাবে আমরা যদি করতে থাকি একে একে বাকি সব permutation ও পেয়ে যাব।

এখন এটা হাতে কর যতখানি সহজ কাজ কোডে করা ওত সহজ নাও মনে হতে পারে। প্রথমে খেয়াল কর আমরা প্রথমে প্রথম ঘরে সংখ্যা বসাব এর পর দ্বিতীয় ঘরে এর পর তৃতীয় ঘরে এরকম করে। সুতরাং তোমরা ভাবতে পার যে এই কাজ for loop দিয়ে করবা। কিন্তু একটু ভেবে দেখ, এর পরের ধাপে তোমাকে এক ধাপে পিছনে গিয়ে আবার সামনে আগাতে হবে, এর পরে আবার দুই ধাপ পিছনে গিয়ে আবার আগাতে হবে। এই কাজ আমরা আর যাই হোক loop দিয়ে করতে পারব না। আরেকটু ভালো করে চিন্তা করলে দেখবে যে আমরা যা করছি তাহল আমাদের কিছু সংখ্যা দেয়া আছে, আমরা সবচেয়ে ছোট সংখ্যা বসিয়ে বাকি সংখ্যা দিয়ে পরের অংশে permutation বানাচ্ছি। শেষ হয়ে গেলে next সংখ্যা বসিয়ে বাকি সংখ্যা দিয়ে আবার permutation বানাচ্ছি। অর্থাৎ জিনিসটা এমন, একটা black box আছে যাকে আমরা সংখ্যা দিলে সে permutation বানাবে। এর জন্য সে সবচেয়ে ছোট সংখ্যাকে রেখে দিবে এর পর বাকি সংখ্যা গুলিকে সে আবার একই ধরনের আরেক black box এ দিয়ে দেবে। ঐ অন্য black box এর কাজ হয়ে গেলে তুমি next বড় সংখ্যা রেখে দিয়ে বাকি সব সংখ্যা দিয়ে ঐ অন্য black box কে আবারো call করবা। আশা করি বুঝতে পারছ যে এই black box টা হল recursive function. কিন্তু এই recursive function টা Fibonacci বা Factorial এর মত সহজ নয়। যখনই একটা ফাংশন লিখবা আগে চিন্তা করে দেখ তোমার এই ফাংশনকে কি দিতে হবে, সে কি দিবে আর সে কি করবে? একে একে এই প্রশ্ন গুলির উত্তর দেয়া যাক। কি দিতে হবে? - যেসকল সংখ্যা এখনো বাকি আছে তাদের দিতে হবে, কি দিবে?- কিছুই দিবে না, কি করবে? - কিছু সংখ্যা ইতোমধ্যেই fix করা হয়েছে, বাকি সংখ্যা গুলি কে permute করে যেসব permutation হয় তাদের সবাই যেন প্রিন্ট হয় তা নিশ্চিত করতে হবে। একটু ভাবলে তোমরা বুঝবে যে যেসব সংখ্যা বসানো হয় নাই সেগুলো black box এ পাঠানোর সাথে সাথে তোমরা এখন পর্যন্ত কার পরে কাকে বসিয়েছ সেটাও পাঠাতে হবে। সুতরাং black box কে দুইটি জিনিস দিতে হবে, ১. এখন পর্যন্ত কোন সংখ্যা গুলো বসানো হয়েছে এবং কি order এ ২. কোন কোন সংখ্যা এখনো বসানো বাকি আছে। base case হল যখন সব সংখ্যা বসানো হয়ে যাবে তখন এবং সেই number sequence আমরা প্রিন্ট করব। এখন পর্যন্ত আমরা দেখেছি কেমনে একটি ফাংশনে একটি integer বা double পাঠানো যায় কিন্তু একটি array কেমনে পাঠাতে হয় তা আমাদের অজানা। standard নিয়ম হচ্ছে তুমি pointer ব্যবহার করে পাঠাবা কিন্তু তোমাদের বেশির ভাগই pointer ভয় কর। আরেকটা উপায় হচ্ছে vector ব্যবহার করা। তবে এটা বেশ slow. আরেকটা উপায় হল global array ব্যবহার করা। আমরা দুই ধরনের array রাখব। একটি array তে থাকবে কোন সংখ্যা ব্যবহার করা হয়েছে কোন সখ্যা ব্যবহার করা হয় নাই তা (0 মানে ব্যবহার করা হয় নাই, 1 মানে ব্যবহার করা

হয়েছে)। আরেকটা array তে এখন পর্যন্ত বসানো নাম্বারগুলি পর পর থাকবে। black box এর ভিতরে তুমি একে একে 1 হতে n পর্যন্ত নাম্বার গুলি চেক করবে যে আগে বসানো হয়েছে কিনা যদি না বসানো হয়ে থাকে তাহলে সেই সংখ্যা বসাবে এবং এটাও লিখে রাখবে যে এই নাম্বারটা ব্যবহার করা হয়ে গেছে। এবার পরবর্তী black box এর কাছে যাবে, সেও একই ভাবে কাজ করবে। কাজ শেষে সে যখন ফিরে আসবে, তখন দুইটা জিনিস করতে হবে তাহল বসানো সংখ্যাকে সরাতে হবে আর তুমি যে লিখে রেখেছ যে এই সংখ্যা ব্যবহার হয়েছে সেটা পরিবর্তন করে লিখতে হবে যে এটা এখনো ব্যবহার করা হয় নাই। এই কাজ যদি না কর তাহলে দেখবে মাত্র একটি permutation প্রিন্ট করেই তোমার প্রোগ্রাম শেষ হয়ে যাবে। তাহলে কি black box এ আমাদের কিছুই পাঠানোর দরকার নেই? আসলে দরকার নেই তবে আমরা আমাদের কোডকে সহজ করার জন্য একটি জিনিস পাঠাবো আর তা হল, আমরা কোন ঘরে এখন সংখ্যা বসাবো সেটা। যদিও এই জিনিস আমরা কোন কোন সখ্যা ব্যবহার করা হয়েছে সেই array থেকে খুব সহজেই বের করতে পারি কিন্তু আমরা যদি এই সখ্যা parameter হিসাবে পাঠাই তাহলে আমাদের কাজ অনেক সহজ হয়ে যাবে। আরও একটি জিনিস পাঠাতে হবে আর তাহল n এর মান, তবে এটি চাইলে global ও রাখতে পার। permutation প্রিন্ট করার প্রোগ্রামটা দেখানোর আগে আরেকটা জিনিস চিন্তা করে দেখতে পার- বসানো সংখ্যা কি সরানর আদৌ দরকার আছে? শুধু কি এই সংখ্যা অব্যবহৃত আছে সেই কাজটা করাই কি যথেষ্ট নয়? তোমাদের জন্য permutation প্রিন্ট করার প্রোগ্রাম কোড 8.8 এ দেয়া হল।

Listing 8.8 : permutation.cpp

```
1  int used[20], number[20];
2
3  //call with: permutation(1, n)
4  //make sure, all the entries in used[] is 0
5  void permutation(int at, int n)
6  {
7          if(at == n + 1)
8          {
9                  for(i = 1; i <= n; i++) printf("%d ", number[i]);
10                 printf("\n");
11                 return;
12         }
13
14         for(i = 1; i <= n; i++) if(!used[i])
15         {
16                 used[i] = 1;
17                 number[at] = i;
18                 permutation(at + 1, n);
19                 used[i] = 0;
20         }
21 }
```

৪.৩.২ Combination Generate

n টি সংখ্যা দেয়া থাকলে তাদের থেকে k টি করে সংখ্যা নিয়ে সকল combination প্রিন্ট করতে হবে। যেমন $n = 3$ ও $k = 2$ হলে আমাদের প্রিন্ট করতে হবেঃ $(1, 2), (1, 3)$ এবং $(2, 3)$. আমরা আগের মত যেমন তেমন ভাবে চিন্তা না করে systemetic চিন্তা করব। দুইভাবে আমরা এই প্রবলেম এর সমাধান খেয়াল করতে পারি। প্রথম উপায়টা হল, আমরা একে একে 1 থেকে n পর্যন্ত যাব আর ঠিক করব এই সংখ্যা কে নিব কি নিব না। একদম শেষে গিয়ে যদি আমরা দেখি যে আমরা k টা সংখ্যা নিয়ে ফেলেছি তাহলে তো হয়েই গেল। আর না হলে আমরা ফেরত যাবো এটা প্রিন্ট না করে। এখন খেয়াল কর, এই সমাধান সঠিক থাকলেও আমাদের সময় কিন্তু অনেক বেশি লাগবে। আমরা প্রতিটি সংখ্যার কাছে গিয়ে গিয়ে একবার নিচ্ছি আরেকবার নিচ্ছি না। সুতরাং মোট 2^n বার কাজ করে এর পর তার থেকে $\binom{n}{k}$ বার প্রিন্ট করছি। আমরা কি এই কাজ $\binom{n}{k}$ সময়ে করতে পারি না? পারি। মাত্র একটি লাইন লিখলেই আমাদের এই কাজ হয়ে যাবে। আমরা যদি কখনও দেখি যে আমাদের যেসব সংখ্যা নেবার

ব্যপারে এখনো decision নেয়া বাকি আছে তাদের সবাইকে নিলেও যদি আমাদের k টা সংখ্যা না হয় তাহলে আর পরবর্তী ফাংশন call এর দরকার নেই। এখান থেকেই ফিরে গেলে হয়। আমাদের এভাবে সমাধান এর প্রোগ্রাম কোড 8.9 এ দেয়া হল। এই কোড এর লাইন 7 এর জন্য আমাদের প্রোগ্রাম $O(2^n)$ হতে $O\binom{n}{k}$ হবে।

Listing 8.9 : combination1.cpp

```
1  int number[20];
2  int n, k;
3
4  //call with: permutation(1, k)
5  void combination(int at, int left)
6  {
7          if(left > n - at + 1) return;
8
9          //you can use left == 0 to make it a little bit more faster
10         //in such case you dont need following if(left) condition
11         if(at == n + 1)
12         {
13                 for(i = 1; i <= k; i++) printf("%d ", number[i]);
14                 printf("\n");
15                 return;
16         }
17
18         if(left)
19         {
20                 number[k - left + 1] = at;
21                 combination(at + 1, left - 1);
22         }
23
24         combination(at + 1, left);
25 }
```

দ্বিতীয় পদ্ধতির ক্ষেত্রে আমরা প্রত্যেক ঘরে যাবো, এরপর এখানে আগে বসানো হয় নাই এমন একটি সংখ্যা বসাবো এবং এভাবে একে একে k ঘরে যখন আমরা সংখ্যা বসিয়ে ফেলব তখন আমরা একটা number combination পেয়ে যাবো। তবে এক্ষেত্রে আমাদের $(1, 2)$ এর সাথে সাথে $(2, 1)$ ও প্রিন্ট হয়ে যাবে। তোমরা যদি n টা সংখ্যা থেকে kটা করে সংখ্যা নিয়ে তাদের permutation প্রিন্ট করতে চাও তাহলে এভাবে করলেই হবে। কিন্তু যদি combination প্রিন্ট করতে চাও তাহলে আরও একটা কাজ করতে হবে আরে তা হল, তুমি নতুন যেই সংখ্যা বসাবে সেটা যেন আগের সংখ্যা থেকে বড় হয়। এই কাজ করার জন্য সবচেয়ে ভালো উপায় হল তুমি parameter হিসাবে সর্বশেষ বসানো সংখ্যাটা পাঠিয়ে দাও এরপর যখন তুমি loop চালাবে তখন 1 থেকে না চালিয়ে এই সংখ্যা থেকে চালালেই হবে। এই সমাধানটা আমাদের প্রথম সমাধান এর থেকে একটু হলেও better বলা যায়। কারণ, আমাদের আগের সমাধান worst case এ recursion এ n depth পর্যন্ত যায়, কিন্তু আমাদের দ্বিতীয় সমাধান k depth পর্যন্ত যাবে। আমাদের দ্বিতীয় সমাধানের প্রোগ্রামের কোড 8.10 এ দেয়া হল। আশা করি 14 নাম্বার লাইন বাদে বাকি কোডটুকু বুঝতে তোমাদের সমস্যা হবে না। আমরা যেভাবে এর আগের বার একটা if দিয়ে $O(2^n)$ হতে $O\binom{n}{k}$ আনা হয়েছে এখানেও সেরকম কাজ করা হয়েছে। তবে পার্থক্য হল আমরা এর আগে আলাদা ভাবে condition চেক করে ছিলাম, এবার for loop এর upper bound হিসাবে এই কাজটা করেছি। এই দুই উপায়ের মাঝে তেমন কোন পার্থক্য নেই, আমরা দুই কোডে দুইভাবে করে দেখালাম তোমরা যাতে দুই উপায়ের সাথে পরিচিত থাকে সেজন্য। তোমরা ভেবে দেখতে পার $i \leq n - k + at$ এই condition টা কই থেকে এল? আমরা যদি i কে at এ বসাই তাহলে আমাদের সংখ্যা বাকি থাকে $n - i$ টি আর আমাদের ঘর বাকি থাকে $k - at$ টি। ঘরের থেকে সংখ্যা কম হওয়া যাবে না, তাই $k - at \leq n - i \leftarrow i \leq n - k + at$.

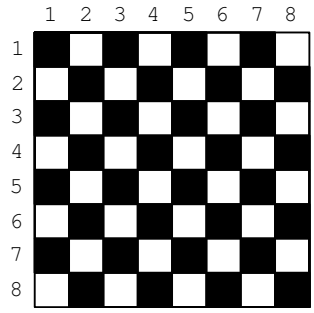
Listing 8.10 : combination2.cpp

```
int number[20];
int n, k;

//call with: permutation(1, 0)
void combination(int at, int last)
{
        if(at == k + 1)
        {
                for(i = 1; i <= k; i++) printf("%d ", number[i]);
                printf("\n");
                return;
        }

        for(i = last + 1; i <= n - k + at; i++)
        {
                number[at] = i;
                combination(at + 1, i);
        }
}
```

৪.৩.৩ Eight Queen

এটি একটি বিখ্যাত সমস্যা। তোমরা যারা দাবা খেলা জানো আশা করি তাদের বুঝতে কোন সমস্যা হবে না। সমস্যাটা হল একটি দাবার বোর্ডে (দাবা বোর্ড 8×8 হয়ে থাকে) 8 টা Queen (আমরা অনেক সময় বাংলায় এদের মন্ত্রী বলে থাকি) কে কত ভাবে বসানো যায় যেন কোন Queen ই অন্য কোন Queen কে attack না করে। একটি Queen অপর আরেকটি Queen কে attack করতে পারবে যদি তারা একই Row বা একই Column বা একই Diagonal বরাবর থাকে। সমস্যাটা খুব একটা কঠিন না। আমাদের যা করতে হবে তা হল প্রত্যেক row তে গিয়ে গিয়ে একটা করে queen বসাতে হবে, সব row তে queen বসানো শেষ হয়ে গেলে আমরা দেখব কোন একটি queen অপর আরেকটি queen কে attack করে কিনা। যেহেতু আমরা প্রতি row তে একটি করে queen বসাচ্ছি সেহেতু কোন দুইটি queen একই row তে আছে কিনা তা দেখার দরকার নেই। শুধু একই column এ আছে কিনা তা দেখতে হবে আর একই diagonal এ আছে কিনা তা। একই column এ আছে কিনা এটা চেক করা খুব সহজ, কিন্তু একই diagonal এ আছে কিনা সেটা দেখা বেশ tricky. দুই ধরনের diagonal হতে পারে। এক ধরনের diagonal উপরের বাম দিক থেকে শুরু করে নিচের ডান দিকে যায় অন্য diagonal গুলি উপরের ডান দিক থেকে নিচের বাম দিকে যায়। মনে করি আমাদের দাবা বোর্ড এর row গুলি উপর থেকে নিচে 1 হতে 8 পর্যন্ত নাম্বার করা এবং column গুলি বাম থেকে ডান দিকে 1 হতে 8 পর্যন্ত নাম্বার করা (চিত্র ৪.২। এখন একটু খেয়াল করলে দেখবে যেসকল diagonal উপরের বাম দিক থেকে নিচের ডান দিকে যায় তাদের row ও column এর বিয়োগফল একই হয় এবং যেসকল diagonal উপরের ডান দিক থেকে নিচের বাম দিকে যায় তাদের row ও column এর যোগফল একই হয়।

তাহলে আমরা প্রতি row তে queen বসানোর পর দুইটি দুইটি করে queen নিয়ে দেখব যে তাদের column বা diagonal একই কিনা। এরকম করে করলে একটা সমস্যা হল, এমনও হতে পারে যে আমরা প্রথম দুইটি queen কে একই column এ বসিয়ে ফেলেছি এর পর বাকি 6টি queen কে আমরা কিন্তু অনেক ভাবে বসাতে পারি, যেভাবেই বসাই না কেন আমরা কোন valid placement পাবো না। সুতরাং আমরা প্রতিবার queen বসানোর আগে বা পরে চেক করে দেখতে পারি যে এখন পর্যন্ত বসানো queen গুলো কেউ কারো সাথে attacking position এ আছে কিনা। একটু চিন্তা করলে দেখবে, আসলে সব queen pair চেক করার দরকার নাই। শুধু মাত্র নতুন বসানো queen এর সাথে আগের বসানো queen গুলোকে চেক করলেই হয়। একটু চিন্তা করলে দেখবে এখানে আমাদের ধরে ধরে আগে বসানো প্রতিটি queen এর সাথে চেক করার দরকার হবে না, যদি আমরা এমন কিছু array রাখি যারা বলে দিবে যে অমুক column বা অমুক diagonal এ কোন queen আছে কিনা।



নকশা ৪.২: দাবা বোর্ড

অর্থাৎ আমরা কোন একটি queen বসানোর সময় array গুলিতে লিখে দিব যে অমুক column, অমুক diagonal এ queen বসেছে। তাহলে নতুন queen বসানোর আগে শুধু আমরা চেক করে দেখব যে যেই column বা diagonal এ আমরা queen বসাতে চাচ্ছি তা আদৌ ফাঁকা আছে কিনা। অর্থাৎ আমাদের কোন loop লাগবে না, শুধু if-else দিয়েই হয়ে যাবে। খেয়াল কর আমরা কিন্তু ধীরে ধীরে আমাদের সমাধানকে যতটুকু সম্ভব optimized করছি। আমরা আমাদের প্রাথমিক সমাধান এর থেকে অনেক দূরে চলে এসেছি ঠিকি কিন্তু আমরা এমন সব কিছু করেছি যাতে করে আমাদের প্রোগ্রাম আগের তুলনায় অনেক অনেক কম সময় নেয়। তোমরা যখন এই জিনিস কোড করে দেখবে তখন প্রতিটি improvement যোগ করার আগে দেখবে তোমাদের কোড কত সময় নেয়। এতে করে তোমরা বুঝবে এসব optimization দেখতে অনেক সহজ হলেও এরা performance এর দিক থেকে অনেক অনেক এগিয়ে দেয় তোমাকে। হয়তো 8×8 বোর্ড এর জন্য এসব optimization এর প্রভাব তুমি নাও বুঝতে পার। তোমরা চাইলে $9 \times 9, 10 \times 10$ এসব বোর্ড এও চেক করে দেখতে পার। আমরা এখানে আলোচনা করা সকল optimization ব্যবহার করে লিখা প্রোগ্রাম কোড 8.11 এ দেখালাম। যদি তোমরা $n = 8$ দাও তাহলে 8×8 বোর্ড হবে, বা তোমরা চাইলে অন্যান্য মাপের বোর্ড এর জন্যও এই প্রোগ্রাম রান করে দেখতে পার।

Listing 8.11 : nqueen.cpp

```
1  int queen[20]; //queen[i] = column number of queen at ith row
2  int column[20], diagonal1[40], diagonal2[40]; //arrays to mark if
       there is queen or not
3
4  //call with nqueen(1, 8) for 8 queen problem
5  //make sure column, diagonal1, diagonal2 are all 0 initially
6  void nqueen(int at, int n)
7  {
8          if(at == n + 1)
9          {
10                 printf("(row, column) = ");
11                 for(i = 1; i <= n; i++) printf("(%d, %d) ", i, queen
                       [i]);
12                 printf("\n");
13                 return;
14         }
15
16         for(i = 1; i <= n; i++)
17         {
18                 if(column[i] || diagonal1[i + at] || diagonal2[n + i
                       - at]) continue;
19                 queen[at] = i;
20                 //note that, i - at can be negative and we cant have
                       array index negative
21                 //so we are adding offset n with this.
```

```
22                  column[i] = diagonal1[i + at] = diagonal2[n + i - at
                        ] = 1;
23                  nqueen(at + 1, n);
24                  column[i] = diagonal1[i + at] = diagonal2[n + i - at
                        ] = 0;
25              }
26  }
```

তোমরা যদি এতটুকুতেই হাঁফ ছেড়ে মনে কর যাক optimization শেষ হল, তাহলে বলে রাখি আরও একটি খুব সহজ optimization আছে যার ফলে তোমরা তোমাদের run time কে একদম অর্ধেক করতে পারবে। চিন্তা করে দেখ সেই optimization টা কি! আসলে optimization এর শেষ নেই। Backtracking এর ক্ষেত্রে যে যত optimization যোগ করতে পারবে তার কোড তত ভালো কাজ করবে। তবে খেয়াল রাখতে হবে সেই সব optimization এর জন্য আবার না অনেক বেশি সময় লেগে যায়! যেমন আমরা যদি বসানোর সময় শুধু array তে না দেখে আগের সব queen এর সাথে যদি চেক করতে যাই তাহলে দেখা যাবে অনেক বেশি সময় লেগে যাবে। সুতরাং আমাদের এই জিনিসও খেয়াল রাখতে হবে।

৪.৩.৪ Knapsack

মনে কর এক চোর চুরি করতে গিয়েছে। তার কাছে একটা থলে আছে যাতে খুব জোর W ওজনের জিনিস নেয়া যাবে। এখন সেই চোর চুরি করতে গিয়ে n টা জিনিস দেখতে পেল। প্রতিটি জিনিসের ওজন w_i এবং ঐ জিনিস বিক্রি করলে সে v_i টাকা পাবে। এখন সবচেয়ে বেশি কত টাকার জিনিস তুমি চুরি করতে পারবে যেন সেসব জিনিসের মোট ওজন W এর থেকে বেশি না হয়? এক্ষেত্রে limit গুলি খুব গুরুত্বপূর্ণ। $n \leq 50, w_i \leq 10^{12}$ এবং $v_i \leq 10^{12}$. তাহলে আমরা কি করব? আগের মত একে একে 1 থেকে n পর্যন্ত যাবো, কোন জিনিস নিব, কোন জিনিস নিবো না শেষে গিয়ে দেখব যে ওজন W এর থেকে বেশি হয়েছে কিনা। বেশি হলে এটা সমাধান হবে না, আর তা না হলে আমরা এরকম সকল সমাধান এর মাঝে যেক্ষেত্রে দাম সবচেয়ে বেশি হয় সেটাই সমাধান হবে। বুঝতেই পারছ এত সাধারণ সমাধান হলে এখানে আরে আলোচনার কিছু থাকত না! প্রথমে হিসাব করে দেখ এই সমাধানের run time কত? $O(2^n)$. অবশ্যই $n \leq 50$ এর জন্য এটা একটা বিশাল মান। প্রথমত আগের মত আমরা প্রতিটি জিনিস নেবার পরেই দেখব এর ওজন W এর থেকে বেশি হয়ে গেছে কিনা, তা হয়ে গেলে পরের জিনিস গুলো দেখার কোন মানে নেই, এখান থেকেই ফিরত যাওয়া উচিত। আরও কি কি optimization থাকতে পারে? খেয়াল কর যেগুলো বাকি আছে তাদের সবার ওজন মিলেও যদি আমাদের সর্বমোট ওজন W এর থেকে বেশি না হয় তাহলে বাকি সবগুলো নিয়ে ফেলাই বুদ্ধিমানের মত কাজ। আবার দেখো, যেসব ওজন বাকি আছে তাদের মাঝে সবচেয়ে যেটি ছোট তাকে নিলেই যদি আমাদের মোট ওজন W এর থেকে বেশি হয়ে যায় তাহলে আমাদের আর এগিয়ে লাভ নেই। ওজন নিয়ে বেশ অনেক optimization ই হয়ে গেছে। দাম দিয়ে কি কিছু optimization করা যায়? যায়, ধর এখন পর্যন্ত আমরা যেসব সমাধান বের করেছি তার মাঝে সবচেয়ে ভালো যেই সমাধান তা থেকে আমরা V টাকা পাই। এখন আমরা সমাধান করার মাঝামাঝি পর্যায়ে যদি দেখি আমরা ইতোমধ্যে যত টাকা পেয়ে গেছি আর এখনও যত জিনিস বাকি আছে তাদের দাম সহ যদি V এর থেকে বেশি না হয় তার মানে এখান থেকে আরও এগিয়ে কোন লাভ নেই। এরকম নানা optimization সহ আমাদের এই প্রোগ্রাম এর run time অনেক কমে যায়।

অধ্যায় ৫

ডাটা স্ট্রাকচার

Algorithm হচ্ছে একটি প্রবলেম সমাধানের পথ আর Data Structure হচ্ছে ডাটা কে সাজিয়ে রাখার জিনিস। অনেক সময় কোন একটি অ্যালগোরিদম এর efficiency ডাটা স্ট্রাকচার এর উপর নির্ভর করে। খুব সহজ একটি উদাহরন দেয়া যাক। মনে কর তোমাকে একে একে একটি করে সংখ্যা দেয়া হবে 1 থেকে n এর মাঝে তোমাকে বলতে হবে এই সংখ্যাটা এর আগে এসেছিলো কিনা। তুমি কেমনে করবে? একটা উপায় হল number এর একটি array রাখা। যখন কোন সংখ্যা আসবে তখন ঐ array তে খুঁজে দেখ এর আগে ঐ সংখ্যা এসেছিলো কিনা যদি না থাকে তাহলে এর array এর শেষে এই সংখ্যাটা রাখ। আরেকটি উপায় হল এমন একটি array রাখ যেখানে তোমার লিখা থাকবে যে একটি সংখ্যা এর আগে এসে ছিল কিনা। খেয়াল কর এখানে তুমি সংখ্যা গুলি রাখবে না শুধু কোন সংখ্যা এসেছিলো কিনা তা রাখবে। এটা করা খুব সহজ। একটি array তে তুমি শুধু 0 বা 1 রাখবে এসেছিলো কি আসে নাই তার উপর ভিত্তি করে। এখন এই দুই ভাবে ডাটা রাখার জন্য উপকার অপকার দুই রকমই আছে। প্রথম পদ্ধতিতে আমাদের যদি m টা সংখ্যা দেয়া হয় তাহলে $O(m)$ memory লাগবে এবং প্রতিবার প্রশ্নের জন্য তোমার $O(m)$ সময় ও লাগবে। কিন্তু পরের পদ্ধতিতে আমাদের $O(n)$ memory লাগবে যেখানে n হল আমাদের ইনপুট এর সবচেয়ে বড় মান কিন্তু আমাদের প্রতিটি প্রশ্নের জন্য মাত্র $O(1)$ সময় লাগবে। দুই algorithm এর মূল নীতি কিন্তু একই, যেই সংখ্যা দেয়া হয়েছে তা আছে কিনা দেখতে হবে, না থাকলে সেই সংখ্যা আমাদের ঢুকিয়ে রাখতে হবে। কিন্তু আমাদের ডাটা কে representation এর ভিন্নতার কারণে আমাদের runtime বা memory requirement কিন্তু আলাদা হয়ে গেছে। একটি পদ্ধতি বড় n কিন্তু ছোট m এ কাজ করবে ভালো, আরেকটা ঠিক উলটো। সুতরাং আমরা কেমন করে ডাটা কে সংরক্ষন করছি তা অনেক গুরুত্বপূর্ণ।

৫.১ Linked List

মনে কর আমরা ফেসবুক এর n জনের মাঝের সম্পর্ক একটি ডাটা স্ট্রাকচার এ রাখতে চাই। কেমনে রাখব? নানা উপায় আছে, একে একে আমরা তিনটি উপায় দেখি এবং তাদের সুবিধা অসুবিধাও।

উপায় ১ একটা $n \times n$ সাইজ এর $0-1$ matrix রাখা। যদি i তম মানুষ আর j তম মানুষের মাঝে বন্ধুত্ব থাকে তাহলে matrix এর $[i][j]$ তম জায়গায় 1 রাখব অন্যথায় 0 রাখব। এই পদ্ধতিতে আমাদের memory লাগবে $O(n^2)$ কিন্তু কোন দুজনের মাঝে বন্ধুত্ব আছে কিনা বা যদি নতুন দুজনের মাঝে বন্ধুত্ব হয় তাহলে আমাদের ডাটা স্ট্রাকচার update করতে সময় লাগবে $O(1)$. একে আমরা adjacency matrix বলে থাকি।

উপায় ২ একটা $n \times n$ সাইজ এর matrix রাখব। i তম row তে থাকবে i তম মানুষের সাথে যারা যারা বন্ধু আছে তাদের একটা লিস্ট। আমরা আরেকটি array তে লিখে রাখব যে কোন এক জন মানুষের কত জন বন্ধু আছে। সুতরাং আমরা জানি $[i][1]$ থেকে $[i][friend[i]]$ পর্যন্ত i তম

মানুষের সব বন্ধু আছে, এখানে friend[i] হল i তম মানুষের friend এর সংখ্যা। এই পদ্ধতিতে আমাদের memory লাগবে আগের মতই $O(n^2)$ কিন্তু কোন দুজনের মাঝে বন্ধুত্ব আছে কিনা তা নির্ণয় এর জন্য আমাদের সময় লাগবে $O(friend[i])$ কারণ আমাদের পুরো বন্ধুর লিস্ট চেক করে দেখতে হবে। কিন্তু নতুন একজন বন্ধু যোগ করতে আমাদের সময় লাগবে $O(1)$ কারণ আমরা কিন্তু খুব সহজেই friend[i] এর মান এক বাড়িয়ে দিয়ে matrix এর $[i][friend[i]]$ স্থানে নতুন বন্ধুকে রাখতে পারি। একে adjacency list এর array implementation বলে।

উপায় ৩ আমরা আলাদা আলাদা করে n জন বন্ধুর জন্য লিস্ট রাখব। আগে থেকেই $n \times n$ ঘরের জায়গা declare না করে আমরা নতুন নতুন বন্ধু আসলে তাদের জন্য dynamically memory তৈরি করে তাদের লিস্ট এ ঢুকিয়ে দেব। এটি হল adjacency list এর linked list implementation. আগের method এর সাথে পার্থক্য শুধু memory complexity তে। এই পদ্ধতিতে যত গুলি freindship ঠিক তত memory লাগবে।

তাহলে লিংক লিস্ট হল array এর ভাই। array তে আগে থেকেই এর সাইজ আমাকে বলে দিতে হয় কিন্তু লিংক লিস্ট এ তার দরকার হয় না। কিন্তু আমরা বেশির ভাগই dynamic memory allocation কে খুব ভয় করে থাকি। তাদের জন্য একটা উপায় হল, প্রোগ্রামিং কন্টেস্ট এর বেশির ভাগ সময়ই আগে থেকেই বলা থাকে যে তোমার সবচেয়ে বেশি কত বড় ইনপুট হতে পারে। এই মান থেকে তোমরা সহজেই বুঝতে পারবে যে তোমাদের লিস্ট এ সবচেয়ে বেশি কত গুলি element দরকার হতে পারে। ঠিক তত সাইজ আগে থেকে declare করে রাখলেই হয়। তোমরা ভাবতে পার যে এটা তো array ই হয়ে গেল। না, মনে কর এর আগের উদাহরন এ আমরা বলে দিলাম মোট 10^6 এর বেশি বন্ধু জোড়া হবে না, কিন্তু মোট 10^4 জন বন্ধু হতে পারে। অর্থাৎ, তুমি যদি এটা array পদ্ধতিতে করতে চাও তাহলে তোমাকে $10^4 \times 10^4$ মেমরি আগে থেকেই declare করে রাখতে হচ্ছে। কিন্তু তুমি যদি linked list পদ্ধতিতে কর তাহলে শুধু 10^6 সাইজ এর মেমরি declare করলেই হয়ে যাবে। এভাবে তোমরা dynamically memory allocate না করেই আগে থেকে define করা একটা array থেকে একে একে জায়গা নিয়ে তোমার কাজ করতে পারবে। আমরা এখানে এই array পদ্ধতিতে কেমনে linked list করা যায় তা দেখাব, তোমরা নিজেরা চাইলে dynamically কেমনে করা যায় তা চিন্তা করে দেখতে পার।

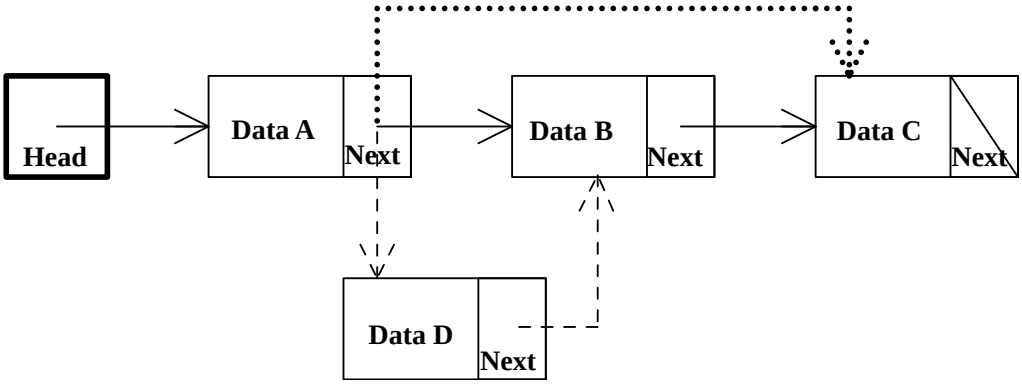
প্রতিটি লিংক লিস্ট এর একটি করে হেড থাকে। এই হেড ঐ লিস্ট এর শুরুকে point করে থাকে। যদি এই লিস্ট ফাঁকা হয় তাহলে এর হেড এ একটি terminal মান থাকে। terminal মান হল এমন একটি মান যা কোন node কে point করে না কিন্তু এই মান দেখলে আমরা বুঝি যে আমাদের লিস্ট এখানেই শেষ হয়ে গেছে। আমরা terminal মান হিসাবে সাধারনত -1 বা 0 এরকম মান ব্যবহার করে থাকি। এবং অবশ্যই খেয়াল রাখতে হবে যে এই মান অন্য কোন মানে যেন না বুঝায়। এখন আমরা মুলত 3 ধরনের কাজ করে থাকি একটি লিংক লিস্ট এর উপর- Insert, Delete এবং Search.

Search

আমরা প্রথম থেকে শুরু করব। যতক্ষণ না terminal মান পাচ্ছি ততক্ষণ প্রতিটি জায়গায় দেখব এর data তে আমারা যেই মান খুঁজছি সেটা আছে কিনা।

Insert

যদি হেড এ terminal মান থাকে তাহলে আমরা একটি নতুন node নিব। এর ডাটা অংশে আমরা ডাটা রাখব আর next অংশতে বর্তমানে হেড যাকে point করে আছে তার id রাখব। যদি লিস্ট আগে থেকে ফাঁকা না থাকে এবং কোন একটি node (যেমন চিত্র ৫.২ এ A ও B এর মাঝে D) এর পরে বসাতে চাই, তাহলে যা করতে হবে তা হল, D এর next পয়েন্ট করবে A এর next কে এবং A এর next point করবে D কে।



নকশা ৫.১: লিংক লিস্ট

Delete

যদি আমাদের হেড এর পরের node কেই delete করতে হয় তাহলে হেড ঐ node এর next যাকে point করে আছে তাকে point করায়ে দিলেই হবে। আর যদি অন্য কোন node হয় (যেমন চিত্রের B) তাহলে আমরা A এর next কে B এর next এ point করিয়ে দেব।

এই কাজ আমরা ইচ্ছা করলে structure দিয়েও করতে পারি বা data ও next এর জন্য দুইটি array রেখেও করতে পারি। আমরা কোড ৫.1 এ হেড এ adjacency list এর মাধ্যমে কোন একটি গ্রাফ এর edge কে insert ও delete এবং search করার জন্য কোড দিলাম। মুলত এই তিনটি জিনিসই প্রয়োজন হয়। আশা করি অন্য ফাংশন গুলি দরকার হলে তোমরা নিজেরা লিখে নিতে পারবে। (যেমন কোন একটি node খুঁজে delete বা ঐ node এর পরে insert করার জন্য তোমরা search ফাংশন কে পরিবর্তন করলেই হবে)। তবে একটি জিনিস মনে রাখতে হবে যে ব্যবহারের পূর্বে head array এর প্রতিটি জিনিস যেন -1 দ্বারা initialize করা থাকে।

Listing ৫.1 : linkedlist.cpp

```
1  int head[10000]; //total node = 10000, 0 to 9999. Initilized to -1
2  int data[100000], next[100000]; //total edge = 100000
3  int id;
4
5  //add node y in the list of x
6  void insert(int x, int y)
7  {
8          data[id] = y;
9          next[id] = head[x];
10         head[x] = id;
11 }
12
13 //erase first node from head of x
14 void erase(int x)
15 {
16         head[x] = next[head[x]];
17 }
18
19 //search node y in x's list
20 int search(int x, int y)
21 {
22         for(p = head[x]; p != -1; p = next[p])
23                 if(data[p] == y)
24                         return 1; //found
25         return 0; //not found
```

```
26 }
```

লিংক লিস্ট এর কিছু variation আছে। যেমন doubly linked list এই লিংক লিস্ট এর শুধু next না previous pointer ও থাকে। এছাড়াও আছে, circular লিংক লিস্ট। এই লিস্ট এর শেষ next এ কোন terminal মান থাকে না বরং এটা আবার শুরুকে point করে থাকে। সুখের কথা হচ্ছে আমরা এখন আর আলাদা করে linked list বানাই না, STL এর vector কে এই কাজে ব্যবহার করে থাকি, তবে এটা মনে রাখতে হবে যে এখানে মধ্যখানে insert বা delete আমাদের linked list এর মত $O(1)$ না। তবে dynamically memory allocation এর কাজ হয়ে থাকে।

৫.২ Stack

আমি ঠিক জানি না Stack কে বাংলায় কি বলে তবে হয়তো একে স্তূপ বলা যায়। আমরা বলে থাকি যে কাগজের স্তূপ জমে গেছে বা থালার স্তূপ জমে গেছে। এখানে স্তূপ আসলে কেমন জিনিস? আমরা যখন একটার পর একটা থালা রাখি তখন কেমনে রাখি? নতুন যা থালা আসে তা সব থালার উপরে রাখি, আর যদি একটা থালা নিই তাহলে উপর থেকে নেই। এটাই Stack। stack এ কোন জিনিস প্রবেশ করান কে push এবং কোন জিনিস তুলে নেয়াকে pop বলে। একে আমরা LIFO বা Last In First Out বলি কারণ সবার পরে যে ঢুকেছে সেই আগে বের হবে। আমরা চাইলে একটা array এবং হেড কে নির্দেশ করার জন্য একটা pointer রেখে খুব সহজেই stack বানিয়ে ফেলতে পারি। তবে এতে সমস্যা হল আমাদের আগে থেকেই অনেক বড় array declare করে রাখতে হবে। অন্য একটি উপায় হল linked list এর মাধ্যমে করা। তবে এখন আমাদের জন্য STL এ stack দেওয়াই আছে। stack এর বিভিন্ন implementation কোড ৫.2 এ দেয়া হল।

Listing ৫.2 : stack.cpp

```
1  /* array implementation */
2  sz = 0                    //initialization
3  s[sz++] = data; //push
4  return s[--sz]; //pop and return
5  if(sz)                    //check whether there is something in stack
6
7  /* linked list implementation */
8  head = -1, sz = 0;
                        //initialization
9  node[sz] = data; next[sz] = head; head = sz++;    //push
10 ret = node[head]; head = next[head]; return ret; //pop & return
11 if(head != -1)
                        //check
12
13 /* STL */
14 #include<stack>
15 using namespace std;
16
17 stack<int> S;                                  //declare, replace int by
       the type you want
18 while(!S.empty()) S.pop();  //initialization after used
19 S.push(5);                                          //push
20 S.top();                                           //return top element
       , but doesnt pop
21 S.pop();                                           //pop but doesnt
       return
22 S.size();                                          //size of the stack
23 S.empty();                                         //returns 1 if empty
```

এখন এত সাধারণ জিনিস ব্যবহার করে কেমনে কঠিন সমস্যা সমাধান করা সম্ভব? একটা উদাহরন দেয়া যাক।

৫.২.১ $0-1$ matrix এ সব 1 আলা সবচেয়ে বড় আয়তক্ষেত্র

সমস্যাঃ ধরা যাক একটি $n \times n$ ডাইমেনশন এর $0-1$ matrix দেয়া আছে। আমাদের এমন সব আয়তক্ষেত্র বের করতে হবে যার সবগুলি সংখ্যা 1। এদের মাঝে সবচেয়ে বড় ক্ষেত্রফলটি প্রিন্ট করতে হবে।

সমাধানঃ প্রথমে আমরা row এর উপর দিয়ে loop চালাব। আমরা ধরে নেব যে এই row ই হল আমাদের আয়তক্ষেত্রের নিচের বাহু। সুতরাং আমরা এখন এমন একটি আয়তক্ষেত্র বের করতে চাই যার ভিতরে সব গুলি সংখ্যা 1 এবং এর ক্ষেত্রফল সবচেয়ে বেশি। আমরা যা করব তা হল প্রতিটি কলামে গিয়ে ঐ row থেকে তার উপর দিকে যতগুলি 1 আছে তার count বের করতে হবে। এর ফলে আমরা আসলে n টি সংখ্যা পাবো। আমাদের এমন একটি sub-range বের করতে হবে যার দৈর্ঘ্য এর সাথে এই sub range এ থাকা সংখ্যা গুলির মাঝে সবচেয়ে কমটি গুন করলে যেই গুনফল হয় তাকে maximize করতে হবে। আমরা যা করব তাহল প্রথমে একটি ফাঁকা stack নিব। এর পর array এর প্রথম স্থান থেকে শুরু করে একটি করে সংখ্যা নিব। যদি আমাদের stack ফাঁকা থাকে তাহলে ঐ সংখ্যা আর এই সংখ্যা যে প্রথম স্থান এ আছে তা stack এ push করব। আর যদি ইতোমধ্যে কিছু সংখ্যা stack এ থাকে তাহলে আমরা stack এর উপরের element এর সাথে দেখব যে stack এ থাকা সংখ্যা আমাদের বর্তমান সংখ্যার থেকে ছোট না বড়। যদি বড় হয় তাহলে আগের মতই এই সংখ্যা এবং এই স্থান stack এ রাখব। আর যদি ছোট হয়, তাহলে stack থেকে একটি একটি করে element তুলতে থাকব যতক্ষণ না আমরা ছোট সংখ্যা পাই। প্রতিবার আমরা যা করব তা হল, ঐ স্থান ও আমাদের বর্তমান স্থানের মাঝের দুরত্বকে ঐ সংখ্যা দ্বারা গুন করতে হবে। এখন এই গুনফল গুলোর মাঝে সবচেয়ে বড়টাই আমাদের উত্তর। যখন আমরা আমাদের বর্তমান সংখ্যার থেকে ছোট একটি সংখ্যা পাবো তখন আমরা সর্বশেষ যেই স্থান stack থেকে pop করেছিলাম সেই স্থান ও আমাদের বর্তমান সংখ্যা push করতে হবে।

এখন কেন এই জিনিস কাজ করবে? একটু চিন্তা করলে দেখবে যে, আমরা আসলে stack এ যা রাখছি তা হল, আমাদের বর্তমান স্থান থেকে কতদুর পর্যন্ত কত height এর 1 পাওয়া যায় তা। আমরা যখন stack থেকে একটা height তুলে ফেলছি কারণ আমাদের নতুন যেই height সেটা ঐ height হতে ছোট আর আমাদের পক্ষে ঐ height এর সমান উচু আয়তক্ষেত্র আসলে বানান সম্ভব না। তাই ঐ element কে তুলে ফেলা হচ্ছে।

৫.৩ Queue

আমরা কিন্তু queue শব্দটা আমাদের ভাষায় প্রায়ই ব্যবহার করে থাকি। যেমন বাস এর queue, ব্যাংক এ বিল জমা দেবার queue. এখানে আসলে কি হয়? নতুন যে আসবে সে queue এর একদম শেষে দাঁড়াবে আর যদি আমরা একজন কে বের করতে চাই তাহলে শুরু থেকে বের করব। এ জন্য একে FIFO বা First In First Out অর্থাৎ যে সবার প্রথমে ঢুকেছিল সেই সবার আগে বের হবে। যেমন বাসের queue তে যে সবার আগে এসেছিলো সেই সবার সামনে থাকবে এবং বাস আসলে প্রথমেই সে বাসে ঢুকবে আবার নতুন কেউ আসলে সে সবার শেষেই দাঁড়াবে কাউকে ডিঙ্গিয়ে সামনে যাবে না। একই রকম stack এর মত এর implementation এর জন্য আমরা array ব্যবহার করতে পারি, বা linked list ও ব্যবহার করতে পারি। তবে STL এ queue নামে ডাটা স্ট্রাকচার দেওয়াই আছে। queue এর বিভিন্ন implementation কোড ৫.3 এ দেয়া হল।

Listing ৫.3 : queue.cpp

```
1 /* array implementation */
2 head = tail = 0;  //initialization
3 q[tail++] = data; //push
4 return s[head++]; //pop and return
5 if(head == tail)  //check whether there is something in stack
6
7 /* STL */
8 #include<queue>
```

```
9  using namespace std;
10
11 queue<int> Q;                              //declare, replace int by
       the type you want
12 while(!Q.empty()) Q.pop();   //initialization after used
13 Q.push(5);                                  //push
14 Q.front();                                 //return front
       element, but doesnt pop
15 Q.pop();                                   //pop but doesnt
       return
16 Q.size();                                  //size of the queue
17 Q.empty();                                 //returns 1 if empty
```

৫.৪ Graph এর representation

Graph হল সহজ অর্থে সম্পর্ক। অনেক ধরনের entity এর মাঝে সম্পর্ক বুঝাতে আমরা graph ব্যবহার করে থাকি। যেমন কিছু আগেই আমরা দেখে এসেছি যে facebook এ কতগুলি মানুষের মাঝে বন্ধুত্ব এর সম্পর্ক বুঝানর জন্য আমরা graph ব্যবহার করেছি। এই গ্রাফ কিন্তু আমাদের লেখ চিত্রের গ্রাফ না। তবে এখানেও আঁকার জিনিস আছে তবে ছক কাগজের দরকার নেই :P তুমি যেসব মানুষের মাঝে সম্পর্ক নির্ণয় করবা তারা এক এক জন একটি করে node বা vertex। আমরা node বা vertex বুঝাতে একটি বিন্দু একে থাকি। এখন দুজন মানুষের মাঝে বন্ধুত্ব আছে এটা নির্দেশ করার জন্য তাদের মাঝে লাইন টেনে থাকি একে edge বলা হয়। তোমরা লক্ষ্য করলে দেখবে একটা map এ বিভিন্ন শহরের মাঝে রাস্তা রেললাইন এসব জিনিস দাগ কেটে দেখানো থাকে। এসব ক্ষেত্রে আমরা শহর গুলিকে vertex ও রাস্তা গুলিকে edge হিসাবে কল্পনা করতে পারি আর তাহলে আমাদের এই বিশাল ম্যাপ একটা গ্রাফ হয়ে যায় যা বিভিন্ন শহরের মাঝে রাস্তার সম্পর্ক দেখায়।

এখন গ্রাফ এর সমস্যা সমাধানের সময় আমাদের এই গ্রাফ কে মেমরি তে রাখতে হবে। আমরা তো আর কম্পিউটারে ছবি একে রাখতে পারি না, আমাদের কে এই vertex গুলির একটি করে number দিতে হয় আর কোন নাম্বার এর সাথে কোন নাম্বার vertex এর সম্পর্ক আসে তা adjacency list বা adjacency matrix এর সাহায্যে রাখতে হয়। আমরা কিন্তু ইতোমধ্যেই এই দুইটির নাম আর কেমনে করতে হয় তা দেখে ফেলেছি!

আমাদের এই facebook এর উদাহরনে friendship কিন্তু mutual বা bidirectional অর্থাৎ A যদি B এর বন্ধু হয় তাহলে B ও A এর বন্ধু হবে। কিন্তু অনেক সময় এই সম্পর্ক bidirectional না হয়ে directional হয়ে থাকে। যেমন facebook এর follower. A যদি B কে follow করে এর মানে এই না যে B ও A কে follow করছে। অর্থাৎ এখানে সম্পর্ক এক তরফা। আমরা আগের গ্রাফ কে বলে থাকি undirected graph ও পরের গুলিকে directed graph. প্রথম ক্ষেত্রে যদি A ও B এর মাঝে যদি undirected edge থাকে তাহলে A এর লিস্টে B কে এবং B এর লিস্টে A কে রাখতে হয়। আর যদি directed edge হয় তাহলে শুধু A এর লিস্টে B কে রাখলেই হবে যদি A এর থেকে edge B এর দিকে হয়।

৫.৫ Tree

Tree একটি special ধরনের গ্রাফ যেখানে n টি vertex এর জন্য $n-1$ টি edge থাকে এবং পুরো গ্রাফ connected থাকে। connected থাকার অর্থ হল ঐ গ্রাফের এক node থেকে অপর node এ তুমি এক বা একাধিক edge ব্যবহার করে যেতে পারবে। যদি আমরা node কে শহর আর edge কে রাস্তা মনে করি তাহলে বলা যায়, প্রতিটি শহর থেকেই অন্য সকল শহরে যাওয়া যায়। এই গ্রাফের কিছু properties আছে। যেমন এই গ্রাফে কোন cycle নাই, অর্থাৎ তুমি যদি কোন node থেকে শুরু কর কোন edge দুইবার ব্যবহার না করে তুমি কোন মতেই ঐ node এ ফিরতে পারবে না। তুমি কোন একটি node থেকে অপর node এ সবসময় uniquely যেতে পারবে মানে তোমার যাবার

রাস্তা একটাই থাকবে (অবশ্যই তুমি এক রাস্তা একাধিক বার ব্যবহার করবা না)। অনেক সময় একটি special node দেয়া থাকে যাকে বলা হয় root. এক্ষেত্রে tree এর edge গুলিকে directed ভাবে কল্পনা করা হয়। edge এর direction হবে root থেকে কোন node এ যেতে হলে ঐ edge দিয়ে তুমি যেদিকে যাবা সেদিক। তোমরা চাইলে root কে ধরে ঐ tree কে ঝুলিয়ে দিতে পার। তাহলে উপর থেকে নিচের দিকে হবে ঐ edge গুলি। কোন edge এর উপরের node কে parent বলা হয়, আর নিচের node কে ঐ parent এর child বলা হয়। কোন node এর parent এর অন্যান্য child কে এই node এর sibling বলে। যদি কোন node থেকে তুমি উপরে root এর দিকে যেতে থাকো তাহলে পথে যেসব node পাবা তাদের ancestor বলে। কোন node থেকে উপরে না উঠে শুধু নিচে নামতে থাকলে যেসব node পাওয়া যায় তাদের descendant বলে। tree এর ক্ষেত্রে level নামে একটা টার্ম আছে, এই টার্ম থেকে বুঝা যায় কোন একটি node আমাদের root থেকে কত গভীরে আছে। আমরা বলে থাকি root আছে level 0 তে, এর এক ধাপ নিচের গুলি level 1 এ। আমরা অনেক সময় level এর পরিবর্তে depth ও বলে থাকি। একটি tree এর সবচেয়ে গভীরের node যদি $h-1$ depth এ থাকে তাহলে আমরা সেই tree এর height h বলে থাকি।

যেহেতু tree এক ধরনের গ্রাফ সেহেতু তুমি একটি গ্রাফ কে adjacency matrix বা list এর মাধ্যমে যেভাবে represent করেছো সেভাবে করা যায়। কিন্তু tree এর আলাদা বৈশিষ্ট্য এর জন্য একে অন্য আরও ভাবেও প্রকাশ করা যায়।

Child List এটা কিছুটা directed graph এ adjacency list এর মত। আমরা প্রতিটি node এর child লিস্ট রাখব। আমরা চাইলে যেকোনো node থেকে শুরু করে শুধু নিচের দিকে যেতে পারি। এই representation এর ক্ষেত্রে বেশি ভাগ সময় আমরা root থেকে শুরু করে নিচের দিকে যেতে থাকি। একে আমরা top down representation বলতে পারি।

Parent link এক্ষেত্রে আমরা প্রতিটি node এর parent রেখে থাকি। এই representation এর ক্ষেত্রে আমরা উপর থেকে নিচে যেতে পারি না। কিন্তু কোন একটা node থেকে উপরে উঠতে পারি। একে আমরা bottom up representation বলতে পারি।

অনেক সময় আমাদের Child list ও Parent link দুইটিই একই সাথে দরকার হয়ে থাকে। যদি প্রতিটি node এর খুব জোর দুইটি child থাকে তাহলে সেই tree কে binary tree বলা হয়। আমরা ছবি আঁকার সময় যে child কে বাম দিকে রাখি তাকে left child ও অপরটিকে right child বলে। এছাড়াও tree সম্পর্কিত আরও অনেক term আছে আমরা ধীরে ধীরে সেসব জানব।

৫.৬ Binary Search Tree (BST)

এই binary tree এর প্রতিটি node এ একটি করে মান থাকে। এখন মান গুলি এমন ভাবে থাকে যেন এর left subtree এর সকল মান [১] এই node এ থাকা মান থেকে ছোট হয় আর right subtree এর সকল মান এর থেকে বড় হয়। আমরা link list এর মত করে এই জিনিস বানাতে পারি। সেক্ষেত্রে প্রতিটি node এ আমাদের দুইটি link এর দরকার হবে, একটি left child এর জন্য অপরটি right child এর জন্য। অনেক সময় parent এর জন্যও আলাদা link রাখা হয়। একটি Binary Search Tree তে কোন একটি সংখ্যা আছে কিনা তা খুঁজে বের করা বেশ সহজ। তুমি কোন একটি node এ গিয়ে দেখবা যে তুমি যেই সংখ্যা খুঁজছ সেটা এখানে থাকা সংখ্যার থেকে ছোট না বড়। যদি সমান হয় তাহলে তো পেয়েই গেলা, আর যদি ছোট হয় তাহলে বাম দিকে যাবা আর বড় হলে ডান দিকে যাবা। এরকম করে তুমি insert ও করতে পারবা। delete করা একটু কঠিন। অনেক সময় প্রোগ্রামিং কন্টেস্টে আমরা সত্যিকার ভাবে delete না করে প্রতিটি node এ একটি করে flag রাখি। delete করলে সেই flag কে আমরা off করে দিলেই হয়।

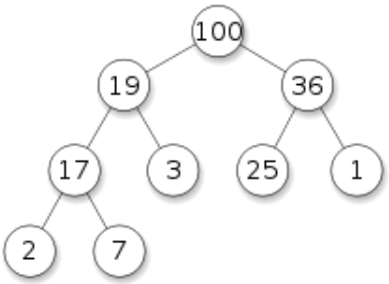
এখন কথা হল, একটা array তে সংখ্যা না রেখে আমরা এরকম tree আকারে সংখ্যা রাখলে লাভ কি? খেয়াল কর, একটি সংখ্যা খুঁজার সময় আমরা যখন একটি node এ থাকা সংখ্যার সাথে আমার সংখ্যাকে তুলনা করি তখন সেই তুলনার ভিত্তিতে আমরা একদিক বাদ দিয়ে আরেক দিকে যেতে পারি।

[১] subtree হল মূল tree এর একটি অংশ

এখন এই ভাগা ভাগি যদি ঠিক অর্ধেক হয় তাহলে আমরা প্রতিবার অর্ধেক সংখ্যা বাদ দিতে পারি। ঠিক আমাদের শিখে আশা binary search এর মত। তাহলে আমরা যদি ঠিক অর্ধেক অর্ধেক করে রাখতে পারি তাহলে আমরা $O(\log n)$ এই সার্চ করতে পারব। তাহলে আমাদের binary search এর সাথে এর পার্থক্য কই? খেয়াল কর, binary search এ আমরা কিন্তু কোন একটি সংখ্যা কে insert বা delete করতে পারি না। কিন্তু আমরা আমাদের এই BST তে কোন সংখ্যা insert বা delete করতে পারি। কিন্তু একটু ভাবলে বুঝবা যে সাধারণ ভাবে প্রবেশ করালে কিন্তু আমাদের BST অনেক লম্বা হয়ে যেতে পারে, যেমন 1 এর ডানে 2, 2 এর ডানে 3 এরকম করে n পর্যন্ত যদি সংখ্যা থাকে তাহলে কিন্তু $O(n)$ সময় লেগে যাবে। যাতে এরকম সমস্যা না হয় সেজন্য আমাদের BST কে balance করে নিতে হয় যেন tree এর height বেশি বড় না হয়। এরকম ধরনের কিছু ডাটা স্ট্রাকচার আছে যেমন AVL Tree, Red Black Tree, Treap ইত্যাদি। তোমরা চাইলে এসব জিনিস internet এ দেখতে পার।

৫.৭ Heap বা Priority Queye

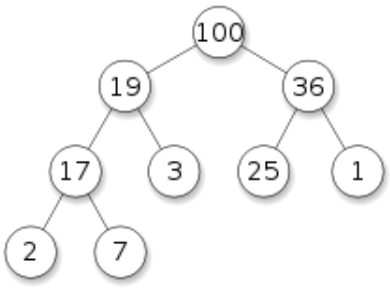
এটিও এক ধরনের binary tree. আরও শুদ্ধ ভাবে বলতে হলে complete binary tree. এই tree এর শেষ level বাদে প্রতিটি level এ সর্বচ্চো সংখ্যক node থাকবে। শুধু শেষ লেভেলটি পূর্ণ নাও হতে পারে, তবে সেক্ষেত্রেও বাম থেকে ডান দিকে node গুলি সাজানো থাকে। চিত্র ?? এ তোমাদের জন্য একটি heap দেয়া আছে।



নকশা ৫.২: Heap

Heap দুই রকম হতে পারে, Max Heap, Min Heap. Max Heap এর বৈশিষ্ট্য হচ্ছে কোন node এ থাকা মান তার যেকোনো descendant এর থেকে বড় হবে। অর্থাৎ root এ এই heap এর সবচেয়ে বড় মান থাকবে, তার left child এ থাকবে left subtree এর মাঝের সবচেয়ে বড় মান এরকম। আশা করি বুঝতেই পারছ Min Heap কি রকম হয়। যদি কোন heap এ n টি node থাকে তাহলে সেই heap এর height $\log n$ হয়। আমরা এই ডাটা স্ট্রাকচারটি তখন ব্যবহার করে থাকি যখন আমাদের অনেকগুলি মান একে একে আসতে থাকে এবং আমাদের মাঝে মাঝে সবচেয়ে বড় মানটি দরকার পরে এবং এই বড় মানটি সরিয়ে ফেলতে হয়। ফলে পরে যখন আবারো সবচেয়ে বড় মান এর দরকার হয় তখন এর পরবর্তী বড় মানটি দিতে হয়। যেমন চিত্র ?? এ আমাদের সবচেয়ে বড় মান চাইলে 100 দিতে হবে, এর পরে আবারো বড় মান চাইলে 36 দিতে হবে এরকম। একটু মনে মনে ভাব তো তোমাদের যদি একটি heap বানাতে বলি কেমনে কোড করবে? যদি ভেবে থাকো link list এর মত করে link রেখে রেখে তাহলে তোমরা ঠিক ভেবেছ। কিন্তু এর থেকেও সহজ উপায় আছে (এবং এর drawback ও আছে!)। তোমরা যদি আগের মত left child link ও right child link রেখে রেখে কর এবং dynamically memory assign কর তাহলে আগে থেকে আমাদের বড় array declare করার দরকার হয় না। কিন্তু যদি আমরা বড় array declare করে করতে চাই তাহলে একটা সহজ উপায় আছে। চিত্র ?? এ আমরা heap এর জন্য array কেমনে indexing করলে সহজ হয় তা দেখালাম। খেয়াল করলে দেখবে, কোন একটি node এর index যদি i হয় তাহলে এর left

child এর index হবে $2i$ এবং এর right child এর index হবে $2i+1$. তাহলে দেখ সুন্দর করে পর পর level by level আমাদের index হয়ে যাবে। যদি কোন node i এ থেকে তার parent এ যেতে চাও তা অনেক সহজে $i/2$ করে যেতে পারবে, মনে রেখ এখানে integer division হচ্ছে।



নকশা ৫.৩: Heap array numbering

Heap এ insert করা খুব সহজ। যদি heap এ ইতোমধ্যে n টা সংখ্যা থাকলে নতুন সংখ্যা তোমরা $n+1$ এ বসাও। এর পর তুমি parent দিয়ে root পর্যন্ত যেতে থাকবে, যদি দেখ parent তোমার থেকে ছোট তাহলে swap করবে, এরকম যতক্ষণ না তোমার parent তোমার থেকে বড় না হয় ততক্ষণ এই কাজ করলেই হবে। যেহেতু আমাদের height $\log n$ সুতরাং আমাদের insertion এ সময় লাগবে $O(\log n)$. যদি Max Heap হতে এর root অর্থাৎ সবচেয়ে বড় সংখ্যা remove করতে চাও তাহলে যা করতে হবে তা হল এর শেষ সংখ্যা কে এনে root এ বসাতে হবে। এর পর দেখতে হবে তোমার left child বড় না right child. ওদের যেটি বড় তা যদি আবার তোমার থেকেও বড় হয় তাহলে তার সাথে swap কর এবং এভাবে নিচে নামতে থাকো। এভাবে $O(\log n)$ এ আমরা সবচেয়ে বড় সংখ্যা কে remove করতে পারি। একটু চিন্তা করলে তোমরা কোন একটি node কে modify বা remove ও করতে পারবে। কিন্তু একটা জিনিস খেয়াল রাখবে, এখানে কিন্তু কোন একটি সংখ্যা খুব দ্রুত খুঁজে পাওয়া সম্ভব না। তোমাকে কোন সংখ্যা খুজতে হলে সবগুলো node এ তোমাকে চেক করতে হতে পারে worst case এ।

Heap কে আমরা কখনও কখনও priority queue বলে থাকি। আমরা queue এর উদাহরন দিতে বাস এর লাইন এর কথা বলেছিলাম, এখন মনে কর, বাসের লাইন ওরকম আগে আসলে আগে যাবেন এরকম না হয়ে কে কত গন্যমান্য ব্যক্তি তার উপর ভিত্তি করে হবে। অর্থাৎ এটা হল priority queue. যত জন মানুষ আছে তাদের মাঝে সেই যাবে যার priority সবচেয়ে বেশি। এই জিনিসই কিন্তু আমাদের heap. STL এ priority queue বানিয়ে দেয়া আছে। কোড ৫.4 এ তোমাদের এই STL এর ব্যবহার দেখানো হল। তোমরা চাইলে শুধু int না কোন structure এরও priority queue বানাতে পার তবে সেক্ষেত্রে তোমাদের operator overload করতে হবে।

Listing ৫.4 : priority queue.cpp

```
#include<priority_queue>
using namespace std;

priority_queue<int> PQ; //declare a max heap
PQ.push(4);                    //insert
PQ.top();                      //maximum element
PQ.pop();                      //pop max
PQ.size();                     //returns size of heap
PQ.empty();                    //returns 1 if heap is empty
```

৫.৮ Disjoint set Union

মনে কর তোমাকে কিছু কোম্পানির নাম দেয়া আছে। প্রথমে সব কোম্পানির মালিক আলাদা আলাদা। এর পর একে একে বলা হবে যে অমুক কোম্পানির মালিক অমুক কোম্পানি কিনে নিয়েছে। মাঝে মাঝে প্রশ্ন করা হবে যে, এই কোম্পানির মালিক কে? বা এই কোম্পানির মালিক আসলে কত গুলি কোম্পানির মালিক? বা সে যত কোম্পানি ক্রয় করেছে তাদের মাঝে সবচেয়ে বেশি লোকজন কাজ করে কোন কোম্পানিতে এরকম নানা প্রশ্ন করা হতে পারে। এই সব ক্ষেত্রে Disjoin Set Union ডাটা স্ট্রাকচার ব্যবহার করে সমাধান করা সম্ভব। একে অনেকে Union Find ও বলে থাকে।

এই ডাটা স্ট্রাকচার এর জন্য আমাদের একটি মাত্র array আছে p. $p[i]$ এর মানে হল i কোম্পানির মালিক হল $p[i]$ কোম্পানির মালিক। প্রথমে সকল i এর জন্য $p[i] = i$. এর পরে কখনও যদি তোমাকে বলে a কোম্পানির মালিক কে? তখন তুমি এর $p[a]$ দেখবে যদি এটি a এর সমান হয় তাহলে তো হয়েই গেল আর না হলে তার $p[]$ দেখবে, এরকম করে চলতে থাকবে। এখন কথা হল এতে তো অনেক সময় লাগার কথা, যদি আমাদের $p[]$ এর array টা এমন থাকে যে, $p[1] = 2, p[2] = 3, \ldots p[n-1] = n$ তাহলে যদি $a = 1$ হয় তাহলে প্রতিবার $O(n)$ সময় লাগবে। এখন খেয়াল কর তুমি যদি একবার 1 এর জন্য বুঝে যাও যে n হল আসল মালিক তাহলে কি তুমি $p[1] = n$ লিখতে পার না? একই ভাবে, তুমি 1 এর মালিক খুঁজার সময় $2, 3, \ldots n-1$ এর ভিতর দিয়ে গিয়েছ এবং সব শেষে তুমি জেনেছ যে তোমাদের সবার মালিক হল n সুতরাং এখন তুমি চাইলে সবার মালিক পরিবর্তন করে n করে দিতে পার। এতে করে কোন ক্ষতি নাই, বরং তুমি এতক্ষণ যে অনেক বড় chain পার করে যে সে আসল উত্তর বের করছ এখন আর ওত বড় chain ডিঙ্গাতে হবে না। একে আমরা Find বলে থাকি। Find এর কোড ৫.5 এ দেখতে পার।

Listing ৫.5 : union find.cpp

```
1  int p[100]; //initially p[i] = i;
2
3  int Find(int x)
4  {
5          if(p[x] == x) return x;
6          return p[x] = Find(p[x]);
7  }
8
9  void Union(int a, int b)
10 {
11         p[Find(b)] = Find(a);
12 }
```

এখন আশা যাক, a এর মালিক যদি b এর মালিক কে কিনে নেয় তাহলে কি করবে? যদি ভেবে দেখ যে, $p[b] = a$ করবে তাহলে ভুল। কারণ b কিন্তু কোম্পানির মালিক না। b কোম্পানির মালিক কে? এই যে কিছু ক্ষণ আগে বের করা হলঃ $Find(b)$. সুতরাং আমরা যা করব তা হল, $p[Find(b)] = a$ এর মানে হল b এর মালিক এখন a এর দ্বারা নিয়ন্ত্রিত। তোমরা চাইলে $p[Find(b)] = Find(a)$ ও করতে পার। একে Union বলে। এর কোডও ৫.5 এ আছে।

৫.৯ Square Root segmentation

একটা ছোট সমস্যা দিয়ে শুরু করি। মনে কর 0 হতে $n-1$ পর্যন্ত nটি দান বাক্স আছে। প্রথমে প্রতিটিতে 0 টাকা করে আছে। একজন করে আসে আরে i তম বাক্সে t টাকা দান করে চলে যায়। মাঝে মাঝে তোমাকে জিজ্ঞাসা করা হবে যে i হতে j পর্যন্ত বাক্সগুলিতে মোট কত টাকা আছে। তুমি কত efficiently এই সমস্যা সমাধান করতে পারবে? এখন খুব সাধারণ একটি সমাধান হল i বাক্সে টাকা রাখতে হলে ঐ বাক্সের টাকার পরিমাণ বাড়িয়ে দেবঃ $amount[i] += t$ আর query করলে i হতে j পর্যন্ত $amount$ যোগ করব। কিন্তু এখানে update অপারেশন মাত্র $O(1)$ সময় নিলেও query অপেরেশন worst case এ $O(n)$ সময় নিবে। তাহলে আমাদের এক্ষেত্রে query এর জন্য সময় update এর সময়

থেকে বেশি। এখন সব গুলি সংখ্যা না যোগ করে কেমনে আমরা অনেক সংখ্যার যোগফল বের করতে পারি? যদি আমরা 0 হতে x বাক্সতে থাকা টাকার পরিমাণ $total[x]$ এ রাখি তাহলে খুব সহজেই $total[j] - total[i-1]$ করে i হতে j বাক্সে থাকা মোট টাকার পরিমাণ পেয়ে যেতে পারি ($i = 0$ এর ক্ষেত্রে একটু সাবধানতা অবলম্বন করতে হবে), এক্ষেত্রে আমাদের query হয়ে যায় $O(1)$. কিন্তু এই যে $total[x]$ এটা নির্ণয় এর জন্য আমাদের i এ t টাকা update এর সময় i হতে n পর্যন্ত $total$ এর পরিমাণ t করে বৃদ্ধি করতে হবে। অর্থাৎ এক্ষেত্রে আমাদের update হয়ে যাবে $O(n)$. আমাদের আসলে এর মাঝামাঝি কোন একটি পদ্ধতি অবলম্বন করতে হবে, যেন কোনটিই যেন খুব বড় না হয়ে যায়। প্রথম পদ্ধতিতে আমাদের update এ অনেক কম সময় লেগেছে কারণ আমরা খুব ছোট একটি জায়গায় পরিবর্তন করেছি, আবার দ্বিতীয় পদ্ধতিতে আমাদের query করতে কম সময় লেগেছে কারণ, অনেক গুলি সংখ্যার যোগফল আমরা এক জায়গায় রেখেছিলাম। আমরা যেটা করতে পারি তা হল, 0 হতে x পর্যন্ত সকল সংখ্যার যোগফল একত্র করে না রেখে কিছু কিছু করে সংখ্যার যোগফল একত্র করে রাখব। ধরা যাক এই কিছুর পরিমাণ হল k. অর্থাৎ, প্রথম k টি সংখ্যা (0 হতে $k-1$ বাক্সের টাকার পরিমাণ) একত্রে $sum[0]$ এ থাকবে, দ্বিতীয় k টি সংখ্যার যোগফল (k হতে $2k-1$ বাক্সের টাকার পরিমাণ) একত্রে $sum[1]$ এ থাকবে এরকম করে প্রতি k টি করা সংখ্যার যোগফল একত্রে থাকবে। তুমি যদি একটু ভালো করে চিন্তা কর তাহলে দেখবে i তম স্থানের সংখ্যা আসলে $sum[i/k]$ এ থাকে। সুতরাং update অপারেশনের সময় তোমাকে $amount[i]$ বৃদ্ধির সাথে সাথে $sum[i/k]$ কেও বাড়াতে হবে। সুতরাং আমাদের update হয় $O(1)$ সময়ে। query এর সময় আমরা আলাদা আলাদা করে যোগ না করে বেশির ভাগ স্থান এভাবে গুচ্ছ গুচ্ছ করে যোগ করব। এখন ধরা যাক আমাদের বলা হল i হতে j পর্যন্ত যোগ করতে হবে। এখন i আছে $x = i/k$ তে আর j আছে $y = j/k$ এ। এখন যদি দেখা যায়, $x = y$ তাহলে আমরা i হতে j পর্যন্ত একটি loop চালাব, যদি তারা আলাদা আলাদা range এ হয় তাহলে, i হতে x range এর শেষ পর্যন্ত যোগ করব, j range এর শুরু হতে j পর্যন্ত যোগ করব আর $x+1$ হতে $y-1$ sum গুলি যোগ করব।x range এর শুরুর মাথা হল kx এবং শেষ মাথা হল $k(x+1)-1$. প্রথম দুইটি যোগ করতে আমাদের খুব জোর $2k$ অপারেশন লাগবে, আর sum গুলির যোগ করতে আমাদের n/k টি যোগ করতে হতে পারে, কারণ যেহেতু প্রতিটি range এর সাইজ k সুতরাং আমাদের মোট range এর সংখ্যা n/k. তাহলে আমাদের update এর জন্য সময় লাগবে, $O(k + n/k)$. তোমরা যদি ক্যালকুলাস জেনে থাকো বা inequality নিয়ে একটু ঘাটাঘাটি করে থাকো তাহলে জানো এই মানটি সর্ব নিম্ন হবে যদি $k = \sqrt{n}$ হয়। এবং এই ক্ষেত্রে query অপারেশনের জন্য $O(\sqrt{n})$ সময় লাগে। $O(n)$ এর তুলনায় $O(\sqrt{n})$ কিন্তু অনেক কম! এই পদ্ধতিকেই Square Root Segmentation বলা হয়।

তোমরা চিন্তা করে দেখতে পার, আমাদের update অপারেশনে শুধু i কে t পরিমাণ না বাড়িয়ে যদি বলা হয় i হতে j পর্যন্ত সবাইকে t পরিমাণ বাড়াতে হবে তাহলে কি সমাধান করতে পারতে? এই সমস্যার সমাধানও আগের সমস্যার মতই তবে প্রতি range এর জন্য এক্ষেত্রে আলাদা আরেকটা variable রাখতে হবে যা নির্দেশ করবে এই range এর সকল $amount$ এর সাথে অতিরিক্ত কত যোগ করলে তুমি আসল টাকার পরিমাণ পাবে। আশা করি এতক্ষণে বুঝতে পারছ যে update এর সময় range গুলির ভিতরে ভিতরে গিয়ে প্রতিটিকে না বাড়িয়ে তুমি ঐ নতুন variable এর মান শুধু বাড়িয়ে দিলেই হয়ে যাবে! এক্ষেত্রে তোমার update এর জন্য ও $O(\sqrt{n})$ সময় লাগবে।

৫.১০ Static ডাটায় Query

এর আগে যা আলোচনা করলাম তাতে আমরা query করি, update করি। কিন্তু যদি কোন update না করা লাগে? অর্থাৎ, প্রথমেই array এর সকল সংখ্যা দিয়ে দেয়া হবে তোমাকে এর পরে query এর উত্তর দিতে হবে। আগের ডাটায় কোন রকম পরিবর্তন হবে না। এটা যদি sum এর জন্য query হয় তাহলে তো খুবই সোজা, তোমরা 1 হতে i পর্যন্ত যোগফল বের করে রাখবে, এর পর তোমাকে যদি বলে i হতে j এর যোগফল কত? তাহলে 1 হতে j এর যোগফল থেকে 1 হতে $i-1$ এর যোগফল বাদ দিলেই $O(1)$ সময়ে উত্তর দিতে পারবে প্রতি query এর। এবং এর জন্য preprocessing সময় লাগবে $O(n)$.

কিন্তু আমাদের query যদি sum না হয়ে max বা min হয়? একটি উপায় হল আগের মত Square Root Segmentation ব্যবহার করা। সেক্ষেত্রে আমাদের preprocessing সময় লা-

গবে $O(n)$ আর query এর জন্য সময় লাগবে $O(\sqrt{n})$. Tarjan এর একটি বিখ্যাত research আছে এই ব্যপারে, সে preprocessing $O(n)$ সময়ে এবং query $O(1)$ সময়ে করতে পারে, তবে সেই method টি বেশ complex. তোমরা চাইলে পড়ে দেখতে পার এই ব্যপারে internet এ। আমরা এখন যেই method দেখব তাতে আমাদের preprocessing সময় লাগবে $O(n \log n)$ আর query সময় লাগবে $O(1)$. যেহেতু max এবং min বের করার method প্রায় একই আমরা এখানে max বের করব। সংক্ষেপে এই পদ্ধতিটি হবে এরকমঃ

১. প্রথমে আমরা প্রতি 1 সাইজের segment এর max বের করবঃ $[1,1],[2,2],[3,3],\ldots$

২. এর পর আমরা প্রতি 2 সাইজের segment এর max বের করবঃ $[1,2],[2,3],[3,4],[4,5]\ldots$

৩. এর পর আমরা প্রতি 4 সাইজের segment এর max বের করবঃ $[1,4],[2,5],[3,6],[4,7]\ldots$

৪. এর পর আমরা প্রতি 8 সাইজের segment এর max বের করবঃ $[1,8],[2,9],[3,10],[4,11]\ldots$

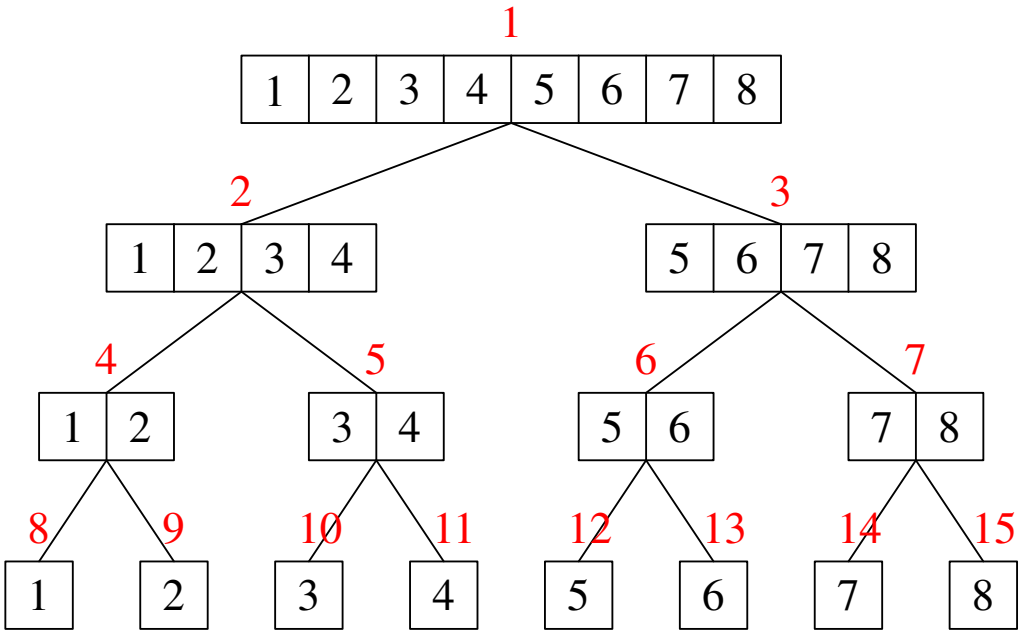
৫. এভাবে i তম ধাপে আমরা 2^i সাইজের segment এর max বের করবঃ $[1,2^i],[2,2^i+1]\ldots$.

এভাবে চলতে থাকবে যতক্ষণ $2^i \leq n$ হয় অর্থাৎ $i \leq \log n$. এখন মনে কর আমরা x হতে শুরু করে 2^i সাইজের max কেমনে বের করব? খুব সহজ, x হতে শুরু করে 2^i সাইজের max হবে x হতে শুরু করে 2^{i-1} সাইজের max এবং $x+2^{i-1}$ হতে শুরু করে 2^{i-1} সাইজের max এই দুইটি সংখ্যার max এর সমান। অর্থাৎ, $table[i][x] = max(table[i-1][x], table[i-1][x+(1<<i)]$. প্রতি ধাপে আমাদের $O(n)$ সময় লাগছে। যেহেতু সর্বমোট $O(\log n)$ টি ধাপ আছে সুতরাং আমাদের মোট সময় লাগবে $O(n \log n)$. এখন তোমাকে যদি জিজ্ঞাসা করে i হতে j এর max কত? তোমাকে সবচেয়ে ছোট x বের করতে হবে যেন $2^x \leq j-i+1$ হয়। এটি তুমি চাইলে একটি loop চালিয়ে বের করতে পার সেক্ষেত্রে $O(\log n)$ সময় লাগবে, তবে তোমরা প্রথমেই যদি একটি array বানাতে পার যেখানে প্রতিটি সংখ্যার জন্য এই মান বের করা থাকে, তাহলে সেই array ব্যবহার করে তোমরা খুব সহজেই $O(1)$ এ x এর মান নির্ণয় করতে পারবে। তাহলে $max(table[x][i], table[x][j-(1<<x)+1])$ ই হল আমাদের কাঙ্ক্ষিত মান।

৫.১১ Segment Tree

Binary Search Tree কোড করা বেশ কষ্টকর ব্যাপার। সে তুলনায় এরই জাত ভাই Segment Tree অনেক ভদ্র। জাত ভাই এই অর্থে যে এখানেও BST এর মতই insert, delete, update ইত্যাদি অপারেশন করা যায় তবে এক্ষেত্রে আমাদের সংখ্যাগুলি 1 হতে n এর মাঝে সীমাবদ্ধ থাকে। শুধু সংখ্যা insert বা delete না, কোন একটি index এ চাইলে আমি কিছু সংখ্যা রাখতে পারি বা একটি range এ আমরা query ও করতে পারি। তবে হুট করে মাঝে নতুন একটি index বসিয়ে দিতে পারব না। অর্থাৎ তুমি যদি চাও যে 1 আর 2 এর মাঝে নতুন একটি জিনিস বসাবে তা হবে না। তাহলে এর সাহায্যে কি কি করা যায়? একটা ছোট খাটো তালিকা বানানো যাকঃ কোন একটি range এ query যেমনঃ সংখ্যা গুলির যোগফল, সবচেয়ে বড় সংখ্যা, জোড় সংখ্যা গুলির যোগফল ইত্যাদি; কোন একটি সংখ্যাকে পরিবর্তন করা; কোন একটি range এর প্রতিটি সংখ্যাকে update করা যেমনঃ নির্দিষ্ট সংখ্যা যোগ করা ইত্যাদি। এটি কোড করা তুলনামুলকভাবে অনেক সহজ। সাধারনত Segment Tree বব্যবহার করে আমরা যেসব সমস্যা সমাধান করতে পারি Square Root Segmentation ব্যবহার করেও করতে পারি। তবে Square Root Segmentation এর ক্ষেত্রে আমাদের complexity হয় $O(\sqrt{n})$ আর Segment Tree এর ক্ষেত্রে হয় $O(\log n)$। Square Root Segmentation এর ক্ষেত্রে আমরা একটি সমস্যা নিয়ে আলোচনা করছিলাম, আমাদের update হল array এর একটি স্থানে একটি সংখ্যা যোগ করা আর query হল array এর কোন একটি range এর যোগফল প্রিন্ট করা। আমরা এই সেকশনে দেখব কেমনে এই সমস্যায় update ও query দুটিই Segment Tree ব্যবহার করে $O(\log n)$ সময়ে করা যায়।

৫.১১.১ Segment Tree Build করা



নকশা ৫.৪: Segment Tree Build

$n = 8$ এর জন্য Segment Tree চিত্র ৫.৪ এ দেখানো হল। আপাতত লাল সংখ্যাগুলিকে বাদ দাও। আমাদের কাছে মোট 8 টি জায়গা আছে $[1, 8]$। আমরা যা করব তা হল এই জায়গা গুলিকে সমান দুই ভাগে ভাগ করবঃ $[1, 4]$ এবং $[5, 8]$. যদি আমাদের কাছে $[L, R]$ এরকম একটি range থাকে তাহলে একে দুই ভাগ করলে দাঁড়াবে $[L, mid]$ এবং $[mid + 1, R]$ যেখানে $mid = (L + R)/2$ (এখানে কিন্তু integer division হচ্ছে)। এখন এভাবে আমরা সকল range কে দুই ভাগে ভাগ করতে থাকব যতক্ষণ না আমাদের segment এ একটি মাত্র সংখ্যা থাকে, অর্থাৎঃ $L = R$. মনে কর না যে আমাদের দেখানো segment tree তে n একটি 2 এর power বলে এটা সম্ভব হয়েছে। যদি $n = 3$ হয় তাহলে আমাদের প্রথম segment $[1, 3]$ কে ভাঙ্গলে আমরা পাবো $[1, 2]$ ও $[3, 3]$ এবং $[1, 2]$ কে ভাঙ্গলে $[1, 1]$ ও $[2, 2]$। এখন কথা হল, আমরা তো খুব সুন্দর করে কাগজে কলমে ছবি একে ফেললাম কিন্তু এটা কোড এ করব কেমনে? এবার লাল সংখ্যা গুলি খেয়াল কর। আমরা প্রতিটি segment এর একটি করে নাম্বার দিয়েছি। ছবির মত করে নাম্বার দেয়ার একটা বিশেষত্ব আছে। খেয়াল করলে দেখবে, কোন নাম্বার x এর বামে নিচে (left child) সবসময় $2x$ এবং ডানে নিচে সবসময় $2x + 1$ থাকে। আবার তার উপরে (parent) $x/2$ হয়। এই ট্রিক খাটিয়ে আমরা খুব সহজেই একটা segment tree বানাতে পারি। কোড ৫.6 এ কিভাবে একটি Segment Tree বানানো যায় তা দেখানো হল। এখানে আমাদের প্রয়োজন মত কোড পরিবর্তন করতে হবে। যেমন যদি বলা থাকে যে আমাদের পুরো segment প্রথমে ফাঁকা তাহলে আমরা 0 দ্বারা initialize করব। আবার অনেক সময় আমাদের বলা থাকে কোন ঘরে কত সংখ্যা আছে, সেক্ষেত্রে আমরা একদম শেষ segment এ গিয়ে সঠিক সংখ্যা বসাব বা ফিরে এসে সঠিক যোগফল রাখব (sum[at] = sum[at * 2] + sum[at * 2 + 1]). খেয়াল করলে দেখবে আমাদের tree এর প্রথম level এ আছে মাত্র 1 টি node, দ্বিতীয় level এ আছে মাত্র 2 টি, এর পরে 4 টি, এরকম করে $8, 16 \ldots n$ টি node, যা যোগ করলে দাঁড়ায় $2n$. সুতরাং আমাদের time complexity $O(n)$.

Listing ৫.6 : segmentTreeBuild.cpp

```
void build(int at, int L, int R)
{
        //do initialization like: sum[at] = 0
        if(L == R)
        {
                //might need to do, something like: sum[at] = num[L]
                return;
        }
        int mid = (L + R)/2;
        build(at * 2, L, mid);
        build(at * 2 + 1, mid + 1, R);
        //do initialization like: sum[at] = sum[at * 2] + sum[at * 2
             + 1] etc.
}
```

Segment Tree এর ক্ষেত্রে Time Complexity এর থেকেও important বলা যায় Space Complexity. কারন অনেকেই ভুল সাইজের array declare করার জন্য Run Time Error পেয়ে থাকে। সবসময় মনে রাখবে, তোমার n যত ঠিক তার 4 গুন বা তার বেশি সাইজের array declare করতে হবে। এর কারন হল n কিন্তু সবসময় এরকম 2 এর power এ থাকবে না। 2 এর power এ থাকলে এটি দুই গুন। কিন্তু 2 এর power এ না থাকলে আসলে এই memory সাইজ একদম ঠিক ভাবে বের করা একটু কঠিন হয়। আমরা জানি x এবং $2x$ এর মাঝে অবশ্যই একটি 2 এর power আছে। আর আমরা জানি 2 এর power এর ক্ষেত্রে দ্বিগুণ লাগে, সুতরাং আমরা 4 গুন সাইজ declare করে থাকি। অনেকে n এর পরের 2 এর power এর দ্বিগুন declare করে। তাহলেও হবে, কিন্তু সেক্ষেত্রে একটু হিসাব নিকাশ করতে হবে।

৫.১১.২ Segment Tree Update করা

প্রথমে update দিয়ে শুরু করা যাক। আমরা কোন একটি সংখ্যাকে বাড়াতে চাই। আমরা root থেকে শুরু করব। আমাদের root হল $[1, 8]$ এবং ধরা যাক আমরা 3 কে update করতে চাই। আমরা যা করব তাহল আমাদের সংখ্যা টা যেদিকে আছে সেদিকে যাব অর্থাৎ, root হতে $[1, 4]$ এ যাব, এর পর $[3, 4]$ এবং সবশেষে $[3, 3]$। এবং ফেরার পথে আমরা build এর সময় যেভাবে sum এর array কে যেভাবে populate করে ছিলাম ঠিক সেভাবে আমরা sum এর array কে update করব। অর্থাৎ আমাদের update ফাংশন দেখতে ৫.7 এর মত হবে। যেহেতু আমাদের tree এর height $\log n$ সুতরাং আমাদের update এর time complexity ও হবে $O(\log n)$।

Listing ৫.7 : segmentTreeQuery.cpp

```
void update(int at, int L, int R, int pos, int u)
{
        //sometimes instead of using if-else in line 11 and 12
        //you can use: if(at < L || R < at) return;
        if(L == R)
        {
                sum[at] += u;
                return;
        }

        int mid = (L + R)/2;
        if(pos <= mid) update(at * 2, L, mid, pos, u);
        else update(at * 2 + 1, mid + 1, R, pos, u);

        sum[at] = sum[at * 2] + sum[at * 2 + 1];
}
```

৫.১১.৩ Segment Tree তে Query করা

এখন আসা যাক query তে। আমরা জানতে চাই $[l, r]$ এই range এ থাকা সংখ্যা গুলির যোগফল কত। আমরা আগের মত tree এর root হতে শুরু করে ধীরে ধীরে নিচের দিকে যেতে থাকব। যদি কখনও দেখি আমরা এখন যেই node এ আছি তার range আমাদের query range এর বাইরে তাহলে তো আর এখান থেকে নিচে যাবার দরকার নেই, তাই না? সুতরাং আমরা এখান থেকেই বলব যে এই range এর জন্য উত্তর 0. যদি আমরা এমন একটি node এ থাকি যা পুরোপুরি আমাদের query range এর ভিতরে তাহলেও কিন্তু নিচে যাবার দরকার নেই, সেক্ষেত্রে আমরা আমাদের এই node এর sum এর মান return করব। যদি এই দুই case এর কোনটিই না হয় এর মানে দাঁড়ায় যে আমাদের query range আসলে এই node এর দুই child এই কিছু কিছু করে আছে। সুতরাং আমরা দুইদিকেই যাব এবং দুই দিক থেকে আসা sum কে যোগ করে return করব। এর কোড তোমরা কোড ৫.৪ তে দেখতে পাবে।

Listing ৫.৪ : segmentTreeQuery.cpp

```
int query(int at, int L, int R, int l, int r)
{
        if(r < L || R < l) return 0;
        if(l <= L && R <= r) return sum[at];

        int mid = (L + R)/2;
        int x = query(at * 2, L, mid, l, r);
        int y = query(at * 2 + 1, mid + 1, R, l, r);

        return x + y;
}
```

এখন কথা হল এর time complexity কত! আমাদের মনে হতে পারে এর time complexity অনেক বেশি! কেউ কেউ ভাবতে পারে যেহেতু আমাদের tree তে $O(n)$ সংখ্যক node আছে তাই এর time complexity ও $O(n)$. না! খেয়াল কর যদি কখনও যদি $[1, 1]$ ও $[2, 2]$ আমাদের query range এর মাঝে থাকে তার মানে দাঁড়ায় আমরা আসলে $[1, 2]$ থেকেই ফিরে যাব। অর্থাৎ তুমি যদি আসলে অনেক বেশি range কে cover করতে চাও তাহলে একটা বড় range থেকেই তুমি ফিরত যাবে। তা নাহয় বুঝা গেল কিন্তু complexity টা আসলে কত? একটু চিন্তা করে দেখ, আমরা কখন কাজ করতেসি? যখন নিচে নামতেসি। কখন নিচে নামতেসি? যখন আমাদের node এর range আমাদের query range এর সাথে partially overlap করে। খেয়াল কর, আমাদের tree এর কোন level এ কিন্তু দুইটার বেশি কিন্তু partially overlap করা node থাকবে না, তাই না? বাকি গুলো হয় বাইরে নাহয় একদম ভিতরে হবে। আমরা তখনই নিচে নামি যখন partial overlap হয়। যেহেতু প্রতি level এ partial overlap এর সংখ্যা 2 আর আমাদের level আছে $\log n$ টি সুতরাং আমাদের time complexity হবে $O(\log n)$. আমরা হয়ত এই জিনিস অন্য ভাবে প্রমান করতে পারতাম, কিন্তু আমি এখানে এভাবে দেখালাম। কারন এখানে time complexity যে আসলে অনেক বেশি না সেটা আমরা tree এর structure দেখে প্রমান করলাম। এরকম প্রমান আর পাবে। তোমরা যদি কখনও Link Cut Tree নিয়ে পড়ার সুযোগ পাও তখন সেখানে এরকম প্রমান দেখতে পাবে। এবং সেই প্রমান আমার কাছে অনেক amazing লেগেছিল!

৫.১১.৪ Lazy without Propagation

মনে কর তোমাদের কে বলা হল যে n টি বাল্ব পর পর আছে এবং শুরুতে তারা সবাই off. এখন একটি অপারেশনে i হতে j পর্যন্ত সকল বাল্ব toggle করতে বলা হতে পারে। [১] আবার তোমাকে কখনও কখনও জিজ্ঞাসা করা হতে পারে যে q তম বাল্বটি on আছে নাকি off? তুমি এই সমস্যার সমাধান Segment Tree ব্যবহার করে $O(\log n)$ এ করতে পারবে। এক্ষেত্রে idea হল যখন তুমি i হতে

[১] toggle অর্থ হল on থাকলে off করা বা off থাকলে on করা।

j কে toggle করবে তখন কিন্তু তুমি এই সীমার মাঝে প্রতিটি বাল্বকে update করতে পারবে না, তোমাকে গুচ্ছ ধরে update করতে হবে। ধর তোমার কাছে $n = 8$ টি বাল্ব আছে আর তোমাকে 1 হতে 4 পর্যন্ত বাল্ব update করতে বলা হল। তুমি যা করবে তা হল Segment Tree এর শুধু $[1, 4]$ এর segment এ লিখে রাখবে যে এই সীমার প্রতিটি বাল্ব toggle করা হয়েছে। (চিত্র ৫.৪ এর সাথে তুলনা করতে পার) যদি তোমাকে বলে 1 হতে 3 পর্যন্ত toggle করতে হবে, তাহলে তুমি $[1, 2]$ এবং $[3, 3]$ এই সীমা দুইটি update করবে। update এর সময় শুধু তুমি লিখে রাখবে যে এই সীমাটি কত বার toggle হয়েছে। লাভ কি? ধর তোমাকে জিজ্ঞাসা করল 3 এর অবস্থা কি? তুমি যা করবে, root থেকে $[3, 3]$ পর্যন্ত যাবে এবং গুনবে এটি যেই যেই সীমা দিয়ে যায় সেসব সীমা কতবার করে toggle হয়েছে। এথেকেই তুমি তোমার উত্তর পেয়ে যাবে। তাহলে query যে মাত্র $O(\log n)$ এ হচ্ছে তাতো খুব সহজেই বুঝা যায়, কিন্তু update? আমরা কিন্তু ইতোমধ্যেই সাবসেকশন ৫.১১.৩ তে এরকম কিছু একটা প্রমান করে এসেছি। সুতরাং আমাদের update ও $O(\log n)$. কোড ৫.9 এ Query ও Update এর কোড দেয়া হল। আমাদের এই সমাধানে আমরা যে একদম নিচ পর্যন্ত না গিয়ে উপরেই কিছু একটা লিখে রেখে শেষ করে ফেলেছি update এর কাজ, একেই Lazy বলা হয়। আমরা এখানে Lazy কে কিন্তু ভেঙ্গে নিচে নামায় নাই, সে জন্য একে Without Propagation বলে। ভেঙ্গে নিচে নামানোর মানে কি তা পরবর্তী সাবসেকশনেই পরিস্কার হয়ে যাবে।

Listing ৫.9 : lazyWithoutPropagation.cpp

```
void update(int at, int L, int R, int l, int r)
{
        if(r < L || R < l) return;
        if(l <= L && R <= r) {toggle[at] ^= 1; return;}

        int mid = (L + R)/2;
        update(at * 2, L, mid, l, r);
        update(at * 2 + 1, mid + 1, R, l, r);
}

//returns 1 if ON, 0 if OFF
int qurery(int at, int L, int R, int pos)
{
        if(pos < L || R < pos) return 0;
        if(L <= pos && pos <= R) return toggle[at];

        int mid = (L + R)/2;
        if(pos <= mid) return query(at * 2, L, mid, pos) ^ toggle[at
            ];
        else return query(at * 2 + 1, mid + 1, R, pos) ^ toggle[at];
}
```

৫.১১.৫ Lazy With Propagation

মনে করা যাক উপরের সমস্যায় আমাদের কোন একটি বাল্ব সম্পর্কে না জিজ্ঞাসা করে জিজ্ঞাসা করা হবে যে l হতে r এর মাঝে কত গুলি বাল্ব on আছে! বলে রাখা ভাল যে এই সমস্যাও একটু চিন্তা করলে Without Propagation এ সমাধান করা সম্ভব। কিন্তু আমরা এখানে দেখাব কেমনে এই সমস্যা with propagation এ সমাধান করা যায়। সমাধানে যাবার আগে আমাদের একটু চিন্তা করা দরকার আমাদের এই সমস্যার সমাধানের জন্য tree এর প্রতি node এ কি কি জিনিস দরকার! প্রথমত Lazy দরকার, অর্থাৎ এই node এর প্রতিটি বাল্ব কি toggle করা হয়েছে কি হয় নাই এবং আর দরকার এই range er কতগুলি বাল্ব এখন on আছে। off এর সংখ্যা কিন্তু দরকার নাই, কারন তুমি যদি on এর সংখ্যা জানো তাহলে off এর সংখ্যা এমনিতেই বেরিয়ে আসবে। সুতরাং build পর্যায়ে আমাদের প্রতি node এ লিখতে হবে $toggle = 0$ এবং $on = 0$. এখন আসা যাক আমরা কেমনে update করব। আগের মতই আমরা দেখব যদি আমাদের বর্তমান node এর সীমা যদি query range এর সম্পূর্ণ বাইরে হয় তাহলে কিছু করব না, যদি partially ভিতরে হয় তাহলে সেভাবেই আমরা ডানে বা

বামে যাব (বা উভয় দিকে)। এখন আসা যাক যদি সম্পূর্ণ ভাবে ভিতরে হয় তাহলে কি করব। খুব সহজ, $toggle^{=}1$ করব, এবং $on = R - L + 1 - on$ করব। আশা করি বুঝা যাচ্ছে এই দুই লাইন এ আসলে কি করা হচ্ছে। কিন্তু শুধু এটুকু করলে কিন্তু হবে না। কেন? একটু দুরের চিন্তা কর। মনে কর তুমি $[1, 4]$ কে এভাবে update করলে। এর পর যদি তোমাকে বলে $[1, 2]$ কে update করতে হবে। তুমি কি করবে? ঐ node এ গিয়ে একি ভাবে update করে আসবে তাই না? কিন্তু যদি এর পরে তোমাকে query করে $[1, 4]$ এ কতগুলি on আছে, তখন তুমি কেমনে উত্তর দিবে? তুমি কিন্তু নিচে $[1, 2]$ তে পরিবর্তন করে এসেছ, সুতরাং তুমি $[1, 4]$ থেকেই উত্তর দিতে পারবে না এখন, কারন $[1, 2]$ এর পরিবর্তন $[1, 4]$ এ কিন্তু নেই। [১] তাহলে উপায় কি? আগে দেখ আমাদের সমস্যাটা কি! আমরা যে $[1, 4]$ এর update এর সময় সেখান থেকেই ফিরে গিয়েছি সেটা সমস্যা। আমরা যদি তা না করে একদম নিচ পর্যন্ত নামতাম এবং node গুলির on ঠিক মত update করতাম তাহলেই হয়ে যেত। কিন্তু তা করলে আমাদের time complexity বেড়ে যাবে। তাহলে উপায় কি? উপায় হল, তুমি এখানেই Lazy রেখে যাবে, কিন্তু যদি কখনও এর থেকে নিচে নামতে হয় তাহলে তখন তুমি এই Lazy কে এক ধাপ নামিয়ে দিবে। অর্থাৎ, আমার যতক্ষণ না দরকার পরবে ততক্ষণ আমরা Lazy নামাব না। এই যে Lazy কে দরকার এর সময় নিচে নামান একেই বলে Propagation. আমরা $[1, 4]$ এর update এর পর যখন $[1, 2]$ কে update করব তার আগেই অর্থাৎ যখন আমরা $[1, 4]$ থেকে নিচে নামতে চাইব তখন আমরা দেখব এখানে কোন Lazy আছে কিনা, যদি থাকে তাহলে তাকে আগে Propagate করব, এর পর নিচে নামব। সেখানে দেখব Lazy আছে কিনা, থাকলে তা Propagate করে আবার নিচে নামব। এরকম করে চলতে থাকবে। একই ভাবে Query এর সময় ও আমরা যদি কোন node দিয়ে নিচে নামতে চাই, নামার আগেই আমাদের দেখে নিতে হবে এখানে কোন Lazy আছে কিনা এবং সেই অনুসারে তাকে দরকার হলে নিচে নামাতে হবে।

এখন আমরা Lazy কে কেমনে নামাতে পারি? তার আগে চিন্তা করে দেখ, একটা Lazy কে নামালে কে কে পরিবর্তন হতে পারে? আমার node এর toggle ও on, আমার left ও right child এর toggle ও on. Lazy থাকা মানে হল $toggle = 1$. একে নিচে নামান মানে, আমার Left ও Right child এর $toggle^{=}1$ হবে। এবং একই সাথে on ও পরিবর্তন হবে। এবং আমার বর্তমান node এর $toggle = 0$ হবে, কিন্তু on পরিবর্তন হবে না। (কারন আমরা যখন toggle করে ছিলাম তখনই আসলে on পরিবর্তন করেছিলাম) একটু চিন্তা করলে দেখবে যে, যদি পর পর দুইবার তোমাকে $[1, 4]$ এ toggle করতে বলে তাহলে কিন্তু তোমাকে এর মাঝে propagation করার দরকার নেই। কারন তুমি নিচে নামছ না, শুধু দুইবার $toggle^{=}1$ এবং $on = R - L + 1 - on$ করতে হবে। এবং এর ফলে প্রথম বারে যেই Lazy জমতো সেটা পরের update এর কারনে cancel হয়ে যাচ্ছে। অর্থাৎ আমাদের প্রব্লেম এ আসলে lazy cancel ও হতে পারে। আসলে cancel হল কি হল না তা নিয়ে তোমাকে চিন্তা করতে হবে না, যদি দেখ $toggle = 1$ এর মানে তোমার এখানে lazy আছে, শেষ! তাহলে এবার এর কোড দেখা যাক। কোড ৫.10 এ এই কোড দেয়া হল।

Listing ৫.10 : lazyWithPropagation.cpp

```
1  void Propagate(int at, int L, int R)
2  {
3          int mid = (L + R)/2;
4          int left_at = at * 2, left_L = L, left_R = mid;
5          int right_at = at * 2 + 1, right_L = mid + 1, right_R = R;
6
7          toggle[at] = 0;
8          toggle[left_at] ^= 1;
9          toggle[right_at] ^= 1;
10
11         on[left_at] = left_R - left_L + 1 - on[left_at];
12         on[right_at] = right_R - right_L + 1 - on[right_at];
13 }
14
15 void update(int at, int L, int R, int l, int r)
16 {
```

[১] একটু চিন্তা করলে তোমরা without propagation এ তাহলে কি করতে হবে তা বের করে ফেলতে পারবে

```
17          if(r < L || R < l) return;
18          if(l <= L && R <= r) {toggle[at] ^= 1; on[at] = R - L + 1 -
                on; return;}
19
20          if(toggle[at]) Propagate(at, L, R);
21
22          int mid = (L + R)/2;
23          update(at * 2, L, mid, l, r);
24          update(at * 2 + 1, mid + 1, R, l, r);
25
26          on[at] = on[at * 2] + on[at * 2 + 1];
27  }
28
29  int qurery(int at, int L, int R, int l, int r)
30  {
31          if(r < L || R < l) return;
32          if(l <= L && R <= r) return on[at];
33
34          if(toggle[at]) Propagate(at, L, R);
35
36          int mid = (L + R)/2;
37          int x = query(at * 2, L, mid, l, r);
38          int y = query(at * 2 + 1, mid + 1, l, r);
39
40          return x + y;
41  }
```

৫.১২ Binary Indexed Tree

সংক্ষেপে একে BIT বলা হয়। এটা Segment Tree এর মতই একটি ডাটা স্ট্রাকচার তবে এটি একটু জটিল, কিন্তু মজার ব্যপার হল এর কোড খুবই ছোট। তুমি পুরো ডাটা স্ট্রাকচার না বুঝেও ব্যবহার করতে পারবে। সত্যি কথা বলতে আমি নিজেও এই ডাটা স্ট্রাকচার খুব ভাল মত বুঝি না। কিন্তু এটা ব্যবহার করতে আমার খুব একটা কষ্ট হয় না। এটা ঠিক BIT দিয়ে তুমি যা যা করতে পারবা Segment Tree দিয়েও প্রায় সবই তুমি তা করতে পারবা, আসলে সত্যি বলতে Segment Tree দিয়ে এমন কিছু করা যায় যা আসলে তুমি BIT দিয়ে করতে পারবা না। কিন্তু BIT এর সুবিধা হল, এটি অনেক ছোট কোড, এর জন্য n সাইজের মেমরি লাগে এবং এটি অনেক fast. তোমরা এর সম্পর্কে আর বিস্তারিত জানতে চাইলে টপকোডার এর article পড়তে পার।

খুব সংক্ষেপে বলতে হলে বলা যায়, BIT এ তোমরা দুই ধরনের operation করতে পার।

১. কোন একটি স্থান idx কে v পরিমান বৃদ্ধি update(idx, v)

২. শুরু হতে idx পর্যন্ত যোগফল বের করা read(idx)

আরও বেশ কিছু operation করা যায় যা আসলে অত বহুল ব্যবহৃত না। তোমরা topcoder এর tutorial এ দেখতে পার।

কোড ৫.11 এ read এবং update দেখানো হল। এখানে MaxVal হল n এর মান। অর্থাৎ তোমার array যত বড় আর কি!

Listing ৫.11 : bit.cpp

```
1   int read(int idx)
2   {
3           int sum = 0;
4
5           while (idx > 0)
```

```
6           {
7                   sum += tree[idx];
8                   idx -= (idx & -idx);
9           }
10
11          return sum;
12  }
13
14  void update(int idx ,int val)
15  {
16          while (idx <= MaxVal)
17          {
18                  tree[idx] += val;
19                  idx += (idx & -idx);
20          }
21  }
```

অধ্যায় ৬

Greedy টেকনিক

Greedy মানে তো সবাই বুঝো? এর মানে হল লোভী। যেমন ধর তোমাকে একটা buffet তে নিয়ে গিয়ে ছেড়ে দিলে কি করবে? তুমি হাপুস হুপুস করে খাওয়া শুরু করে দেবে তাই না? যদি একটু বুদ্ধিমান হউ তাহলে হয়তো সবচেয়ে দাম যেই খাবারের সেইটা বেশি বেশি করে খাবা! কারন যার দাম কম তা হয়তো তুমি পরে কিনে খেতেই পারবে? যেমন যদি buffet তে গলদা চিংড়ি থাকে আর জিলাপি থাকে, তাহলে নিশ্চয় জিলাপি খেয়ে পেট ভরানোর থেকে চিংড়ি খেয়ে পেট ভরানো বুদ্ধিমানের মত কাজ হবে? Greedy মানে যে সবসময় বেশি বেশি করে নেয়া তা কিন্তু না। অনেক সময় কম কম নেয়াও লাভ জনক। যেমন তোমাকে বলা হল একটি গাড়ি কিনতে। এখন গাড়ি গুলো একেকটা একেক পরিমান তেল খায়! নিশ্চয় যেই গাড়ি সবচেয়ে কম তেল খায় সেটা কেনাই বুদ্ধিমানের মত কাজ তাই না? যদিও বাস্তব জীবনে আরও অনেক factor আছে! যাই হক, তো Greedy মানে হল অন্য কিছু না দেখে যার মান কম বা বেশি তাকে সবসময় বাছাই করা।

৬.১ Fractional Knapsack

Greedy algorithm এর জন্য এটি খুবই common সমস্যা। মনে কর একটা চোর একটি মুদি দোকানে ঢুকেছে চুরি করতে। সেখানে চাল আছে, ডাল আছে, চিনি, লবন এরকম নানা জিনিস আছে। এখন সে সব জিনিস চুরি করতে পারবে না। কারন তার কাছে যেই থলে আছে তার ধারন ক্ষমতা ধরা যাক 20 kg. তাহলে সে কিভাবে চুরি করলে সবচেয়ে বেশি লাভবান হবে? খুবই সহজ। যেই জিনিসটার দাম সবচেয়ে বেশি তুমি সেই জিনিস আগে নেয়া শুরু করবে। যদি দেখ ঐ জিনিস নেয়া শেষ এবং এখনও থলে তে কিছু জায়গা বাকি আছে তাহলে তুমি পরবর্তী দামি জিনিস নেয়া শুরু করবে। এরকম করে যতক্ষণ না তোমার থলের ধারন ক্ষমতা না শেষ হচ্ছে তুমি নিতে থাকবে। এখানে খেয়াল কর, দাম বেশি মানে কিন্তু প্রতি kg এর দাম। ধর চাল আছে 1 kg আর দাম 100 টাকা, আর ডাল আছে 500g কিন্তু এর দাম 60 টাকা তাহলে কিন্তু ডাল নেয়া লাভজনক হবে কারন, ডাল এর দাম প্রতি kg তে 120 টাকা!

এই সমাধান ঠিক আছে যদি তুমি কোন জিনিসের যেকোনো পরিমান নিতে পার। সমস্যাটা যদি চাল ডাল না হয়ে electronics এর দোকান হয় তাহলে তুমি আর এভাবে সমাধান করতে পারবে না। তুমি তো আর একটা tv ভেঙ্গে এর অর্ধেক চুরি করবে না তাই না? tv হোক laptop হোক আর mobile ই হোক তুমি যাই নিবা না কেন পুরোটা নিবা। তাহলে কিন্তু আমাদের greedy মেথড কাজ করবে না। উদাহরন দেয়া যাক, মনে কর 1 টা tv দাম 15000 টাকা এবং ওজন 15kg, দুইটা monitor আছে যাদের ওজন 10kg করে এবং প্রতিটার দাম 9000 টাকা। তোমার কাছে 20 kg জিনিস নেবার থলে আছে। তুমি কি করবে? tv নেয়া কিন্তু বোকামি হবে যদিও এর প্রতি kg তে দাম বেশি তাও তোমার দুইটা monitor নিলে লাভ হবে সবচেয়ে বেশি। সুতরাং এটা মনে করার কিছু নাই যে Greedy মেথড সবসময় কাজ করবে। যখন তুমি জিনিসের চাইলে "কিছু" অংশ নিতে পারবে তখন তাকে বলা হয় Fractional Knapsack আর যদি তোমাকে পুরোপুরি নিতে হয় তাহলে তাকে বলা হয় 0-1 knapsack (এটি পরবর্তী অধ্যায় এ আমরা দেখব কেমনে সমাধান করতে হয়)।

৬.২ Minimum Spanning Tree

এই সেকশন এর নাম দেখে ভয় পাবার কিছু নেই। খুবই সহজ জিনিস। আমরা আগেই জেনে এসেছি Tree কাকে বলে, এখন দেখে নেয়া যাক Minimum Spanning Tree কি জিনিস। মনে কর আমাদের একটা weighted graph দেয়া আছে (weight গুলি ধনাত্মক) অর্থাৎ কিছু vertex, কিছু edge এবং সেই edge গুলির weight. তোমাকে এখন এদের মাঝ থেকে কিছু edge বাছাই করতে হবে যেন তাদের weight এর যোগফল সর্বনিম্ন হয় এবং সকল vertex যেন connected হয়। এটা নিশ্চয় বুঝতে পারছ যে তোমার যদি n টি vertex থাকে তাদের connected করতে আসলে তোমার সর্বনিম্ন $n-1$ টা edge লাগবেই। এবং তুমি যদি তোমার বাছাই করা edge গুলির weight এর যোগফল সর্বনিম্ন করতে চাও তাহলে অবশ্যই $n-1$ টার বেশি edge নিবে না। আর n vertex এবং $n-1$ edge ওয়ালা একটি connected graph হল tree. অর্থাৎ আমাদের সকল vertex কে connected করতে আমরা যেসকল edge নির্বাচিত করব তাদের weight এর যোগফল সরবনিম্ন করতে চাইলে যেই graph টি দাঁড়ায় সেটিই হল Minimum Spanning Tree (সংক্ষেপে MST). এখানে Minimum আর Tree শব্দ দুইটি তো বুঝছই? Spanning অর্থ connected মনে করতে পার। এখানে আমরা MST বের করার জন্য দুইটি algorithm এর কথা বলব। তোমরা কেউ কেউ মনে করতে পার যে হয়তো এই সেকশনটি গ্রাফ এর অধ্যায়ে থাকলে ভাল হত। কিন্তু আমরা যেই দুইটি algorithm আলোচনা করব তারা আসলে Greedy টাইপ বলা যায়। আর তোমরা কেমনে একটি গ্রাফ কে represent করা যায় তাতো শিখেই ফেলেছ! সুতরাং চিন্তা কি! আমাদের পরবর্তী দুইটি সেকশনের জন্য ধরে নেই যে আমাদের প্রদত্ত গ্রাফে n টি vertex ও m টি edge আছে।

৬.২.১ Prim's Algorithm

যেহেতু আমাদের সবগুলি vertex কে connected করতে হবে সুতরাং আমরা যেকোনো vertex থেকে শুরু করতে পারি। এখন আমরা দেখব, এই vertex এর সাথে যেই যেই edge আছে তাদের মাঝে কার weight সবচেয়ে কম। যার সবচেয়ে কম সেই edge আমরা নিব এবং তাহলে আমাদের এখন দুইটা vertex ও একটি edge হয়ে গেল। এখন দেখব, এই দুইটি vertex থেকে যেসব edge বের হয়েছে তাদের মাঝে কার weight সবচেয়ে কম তাকে নিব এভাবে নিতে থাকব যতক্ষণ না আমাদের সব vertex নেয়া হয়ে যায়। এটা আশা করি বুঝছ যে যখন এই সবচেয়ে কম weight এর edge নিচ্ছ তখন সেই edge এর এক মাথা তোমার ইতমধ্যে বানানো tree এর ভিতরে যেন থাকে এবং অপর মাথায় যেন আমরা এখনও নির্বাচন করি নাই এরকম vertex থাকে। দুই মাথাই যদি আমাদের tree এর মাঝে থাকে তাহলে কিন্তু লাভ নেই! কারন তারা তো ইতোমধ্যেই connected, শুধু শুধু এই edge নিয়ে weight এর যোগফল বাড়ানোর কি কোন মানে আছে?

এখন কথা হল এই algorithm এর time complexity কত! প্রথমত তুমি n বার এই নতুন vertex নির্বাচন করার কাজ করছ। এবং প্রতিবার হয়তো তুমি সব edge দেখছ। সুতরাং তোমার complexity দাঁড়ায় $O(nm)$. একে তুমি খুব সহজেই $O(n^2)$ করতে পার। খেয়াল কর, মনে কর তুমি প্রথমে a vertex নিয়েছিলে এবং আমাদের বর্তমান process এ প্রতিবার a এর সাথে লাগান সব edge প্রতি বার চেক করছ। কিন্তু প্রতিবার চেক করার কি দরকার আছে? তুমি যখন একটা নতুন vertex আমাদের tree তে অন্তর্ভুক্ত করবে তখন এর সাথে লাগান সব edge দেখবে, দেখে সেই edge এর অপর প্রান্ত কত কম খরচে আমাদের tree তে নেয়া যায় সেটা update করবে। মনে কর, আমরা a vertex যখন নিয়েছি তখন দেখেছি যে b কে আমরা 10 cost এ অন্তর্ভুক্ত করতে পারি, কিন্তু c অন্তর্ভুক্ত হয়ে জাবার পর হয়তো দেখলে যে b কে আরও কম খরচে তুমি অন্তর্ভুক্ত করতে পারবে! তখন তুমি b এর cost কে update করবে। এবং প্রতিবার দেখবে, কোন কোন vertex এখনও নির্বাচন করা হয় নাই তাদের মাঝ হতে সবচেয়ে কম খরচের vertex কে তুমি অন্তর্ভুক্ত করবে। তাহলে কি দাঁড়ালো? তুমি আবারও n বার কাজ করবে, প্রতিবার সব অনির্বাচিত vertex যাচাই করে দেখবে যে কোনটি সবচেয়ে কম, তাকে নিবে। এর পর এর সাথে লাগান সব edge বরাবর গিয়ে দেখবে যে তার অপর প্রান্তে কোন অনির্বাচিত vertex আছে কিনা, থাকলে তার cost কে update করবে। এবং এই কাজ n বার করতে হবে। তাহলে আমাদের complexity কত? n বার কাজ করছি আর প্রতিবার n টা vertex আমরা check করে দেখছি আর আমরা খুব জোর $2m$ বার edge চেক করছি, সুতরাং

আমাদের complexity দাঁড়ায় $O(n^2 + m)$ যা আসলে $O(n^2)$ বলা যায় কারন $m < n^2$ তাই না?

তোমরা কিন্তু চাইলে একে আরও improve করতে পারবে। আমরা যে প্রতিবার n টা vertex ঘুরে ঘুরে দেখছি কোনটার cost সবচেয়ে কম তা না করে তুমি যদি একটা Min-Heap রাখ (C++ এ priority queue বা set) তাহলে তোমার এই algorithm আসলে $O(m \log n)$ এ কাজ করবে। বুঝতেই পারছ এই পদ্ধতি তখনই ভাল কাজ করবে যখন m তোমার n^2 এর থেকে বেশ ছোট হবে। আরও ভাল heap ব্যবহার করে এর complexity আরও কমান যায়। তোমরা যদি interested হউ একটু internet বা বই ঘেটে ঘুটে দেখতে পার।

৬.২.২ Kruskal's Algorithm

এটিও বেশ সহজ algorithm. কোন কারনে যখনি MST আমাকে কোড করতে হয় আমি Kruskal's algorithm ই implement করে থাকি। হয়তো আমার কাছে এটি সহজ লাগে সেজন্য! এই algorithm বুঝানো খুবই সহজ, কোড করাও অনেক সহজ কিন্তু যেভাবে কোড করতে হবে সেটা বুঝানো একটু কষ্টকর। তুমি যা করবা তাহল সবচেয়ে কম weight এর edge নিবা, দেখবা এর দুই মাথার vertex দুইটি ইতোমধ্যেই একই tree বা component এ আছে কিনা, থাকলে এই edge নিবা না। না থাকলে নিবা। শেষ! এখন প্রশ্ন হচ্ছে কেমনে বুঝবা যে দুইটি vertex একই tree তে আছে কিনা! Disjoint Set Union. খেয়াল কর প্রথমে সকল vertex আলাদা আলাদা set. আমরা যখনি একটি edge নিচ্ছি তখন দুইটি set কে জোরা লাগানর চেষ্টা করছি। এবং পরবর্তীতে চেক করছি যে, এই দুইটি vertex একই set এ আছে কিনা! এর time complexity কত? সহজ, edge গুলিকে weight অনুযায়ী sort করতে $O(m \log m)$ এবং প্রতি edge এর জন্য আমরা find করছি বা দুইটি set কে union করছি যাদের complexity আমরা $O(1)$ ধরে নিতে পারি। সুতরাং $O(m \log m + m) = O(m \log m)$.

খেয়াল কর আমরা কিন্তু এই দুই algorithm এই greedily সবচেয়ে কম খরচের vertex বা edge নির্বাচন করে পুরো প্রব্লেম সমাধান করে ফেলেছি। তোমাদের যদি এই algorithm দুটির কোনটিতে সন্দেহ হয় তাহলে প্রমান করে দেখতে পার কেন এই greedy ভাবে edge নির্বাচন করলে সঠিক উত্তর দিবে।

৬.৩ ওয়াশিং মেশিন ও ড্রায়ার

মনে কর তুমি একটি কাপড় কাঁচার কোম্পানি চালাও। তোমার কাছে কিছু সেট কাপড় আছে। তুমি প্রতিটি সেট কে প্রথমে ওয়াশিং মেশিনে দিবে এবং এর পর তাকে ড্রায়ার এ দিবে। তুমি কিন্তু প্রথমে ড্রায়ার পরে ওয়াশিং মেশিনে দিতে পারবে না। এখন প্রতি সেট এর জন্য তোমার জানা আছে যে সেটি ওয়াশিং মেশিন এ কত সময় নিবে এবং ড্রায়ার এ কত সময় নিবে। অবশ্যই একই সাথে কয়েক সেট কাপড় তুমি কোন মেশিনে দিতে পারবে না কিন্তু একই সাথে দুই সেট কাপড় দুই মেশিন এ দিতে পারবে। তুমি কত কম সময়ে সব কাপড় পরিস্কার করে ফেলতে পারবে?

সমস্যাটা যত না সুন্দর এর সমাধান তার থেকে বেশি সুন্দর। মনে কর i তম সেটের জন্য a_i হল ওয়াশিং মেশিন এ দরকারি সময় আর b_i হল ড্রায়ার এ দরকারি সময়। এখন মনে কর optimal order এ দুইটি পাশাপাশি সেট হল i আর j. অর্থাৎ তুমি চাইতেছ যে ঠিক i কাজ এর পর পর j কাজ করবা তাহলে এই দুইটি কাজ করতে তোমার কত সময় লাগবে? $a_i + max(b_i, a_j) + b_j$। আর যদি j কাজ আগে করতে তাহলে তোমার সময় লাগত $a_j + max(b_j, a_i) + b_i$। অবশ্যই $a_i + max(b_i, a_j) + b_j \leq a_j + max(b_j, a_i) + b_i$. এখন মনে কর k হল আরেক সেট কাজ এবং $a_j + max(b_j, a_k) + b_k \leq a_k + max(b_k, a_j) + b_j$ অর্থাৎ k সেট j সেট এর পর করা ভাল। এই দুইটি ইকুয়েশন যদি সত্যি হয় তাহলে প্রমান করা যায় যে $a_i + max(b_i, a_k) + b_k \leq a_k + max(b_k, a_i) + b_i$ অর্থাৎ i কাজের পর k কাজ করা ভাল (এটি তোমরা proof by contradiction এর মাধ্যমে একটু খেটেখুটে করতে পার)। তাহলে কি দাঁড়ালো? আমরা কাজ গুলোকে আসলে একটা order এ সাজাতে পারি। তবে কোন কাজের আগে কোন কাজ আসবে সেটা নির্ণয় করার জন্য আমাদের উপরের ইকুয়েশন এর

সাহায্য নিয়ে দেখতে হবে যে কোন সেট আগে করলে কম সময় নেয়। এভাবে কাজ গুলোকে সাজালে আমরা optimal order পাবো।

এই সমাধান বের করা মোটেও সহজ নয়, সুতরাং তোমরা যদি একবার পড়ে এই সমাধান না বুঝো তাহলে আরও কয়েকবার পড়ে দেখ। একটু দুই একটা উদাহরন হাতে হাতে করে দেখ।

অধ্যায় ৭

Dynamic Programming

Dynamic Programming একটি অত্যন্ত গুরুত্বপূর্ণ এবং বলা যায় সবচেয়ে কঠিন টপিক প্রোগ্রামিং কন্টেস্টে। এটিকে কঠিন বলার কারন হল এতে ভাল করার এক মাত্র উপায় হল practice করা এবং বেশি বেশি করে এই টাইপের প্রব্লেম দেখা। এখানে আসলে শেখানোর তেমন কিছু নেই। এই মেথোডের প্রধান বিষয় হল বড় জিনিসের সমাধান ছোট জিনিসের সমাধান থেকে আসবে!

৭.১ আবারও ফিবোনাচি

মনে কর তোমাকে বলা হল, এমন কয়টি array আছে যার সংখ্যাগুলি 1 বা 2 এবং তাদের যোগফল n হয়। যেমন যদি $n = 4$ হয় তাহলে তুমি মোট 5 ভাবে array বানাতে পারবেঃ $1, 1, 1, 1$, $1, 1, 2$, $1, 2, 1$, $2, 1, 1$ এবং $2, 2$. এখন কথা হল এই সমস্যা কেমনে সমাধান করব! খেয়াল কর, আমাদের array এর প্রথম সংখ্যা হয় 1 হবে নাহলে 2. যদি 1 হয় বাকি অংশটুকু $n - 1$ সংখ্যা এর ক্ষেত্রে যত ভাবে array পাওয়া যায় ঠিক তত ভাবে সাজানো সম্ভব। আবার যদি 2 হয় তাহলে $n - 2$ কে যতভাবে সাজানো যায় ততভাবে। ধরা যাক, n কে সাজানো যায় $way(n)$ ভাবে, তাহলে $way(n) = way(n - 1) + way(n - 2)$. এখন এখানে কিছু সমস্যা আছে। প্রথমত সবসময় কিন্তু তুমি শুরুতে 1 বা 2 নিতে পারবে না। যেমন যখন $n = 0$ তখন শুরুতে 1 নেওয়া যায় না, আবার $n = 0$ বা 1 হলে শুরুতে 2 নিতে পারবে না। অর্থাৎ এই ফর্মুলা কাজ করবে যদি $n > 1$ হয়। সেক্ষেত্রে $way(2) = way(1)+way(0)$. $way(1)$ বা $way(0)$ এর মান কিন্তু আমরা এই ফর্মুলা ব্যবহার করে বের করতে পারব না। কারন আমরা আগেই বলেছি এই ফর্মুলা কাজ করবে যদি $n > 1$ হয়। আমরা এই দুইটি মান হাতে হাতে বের করব। $way(1)$ মানে হল 1 কে আমরা কত ভাবে সাজাতে পারব। খুব সহজ, এক ভাবে আর সেটা হলঃ 1. অর্থাৎ $way(1) = 1$. এখন আশা যাক, $way(0)$ এর মান কত হবে। একটু অবাক লাগতে পারে কিন্তু $way(0) = 1$. তোমরা ভাবতে পার 0 কে তো সাজানোই যাবে না সুতরাং 0 হওয়া উচিত। কিন্তু আমি যদি বলি অর্থাৎ ফাঁকা array এর যোগফল 0 তাহলে কি খুব একটা ভুল হবে? আচ্ছা তমাদের অন্য ভাবে বুঝানর চেষ্টা করি। $way(2)$ এর মান কত? 2 তাই না? কারনঃ 2 এবং $1, 1$ এই দুইটি হল $n = 2$ এর জন্য উত্তর। আর আমরা জানি, $way(2) = way(1) + way(0)$ এখন আমরা জানি $way(2) = 2$ এবং $way(1) = 1$ তাহলে তো $way(0) = 1$ হবেই তাই না? এরকম কেন হল? দেখ, তুমি $n = 2$ এর জন্য যদি প্রথমে 1 নাও তাহলে বাকি $2 - 1 = 1$ তুমি একভাবে সাজাতে পারবে কারন $way(1) = 1$ বা 1. এখন তুমি যদি সামনে 2 নাও তাহলে কিন্তু বাকি আর কিছু নিতে পারবে না, অর্থাৎ কিছু না নিতে পারা হল এক ভাবে নেওয়া!!! অর্থাৎ $way(0) = 1$. আমরা আগেই বলে এসেছি (যখন আমরা recursive function শিখেছি) যখন আমাদের এরকম ফর্মুলা কাজ করবে না সেটাকে বলা হয় base case. এখন চল এটাকে কোড করি। প্রায় সব DP [১] প্রব্লেম এর কোড দুই ভাবে করা যায়। Iteratively এবং Recursively. Iterative ভাবে সমাধান করলে কোড দেখতে কোড ৭.1 এর মত হবে আর recursively করলে কোড ৭.2 এর মত হবে।

[১] Dynamic Programming কে সংক্ষেপে আমরা DP বলে থাকি

Listing ৭.১ : fibIterative.cpp

```
way[0] = way[1] = 1;
for(i = 2; i <= n; i++)
        way[i] = way[i - 1] + way[i - 2];
```

Listing ৭.২ : fibRecursive.cpp

```
int way(int n)
{
        if(n == 0 || n == 1) return 1;
        return way(n - 1) + way(n - 2);
}
```

৭.২ Coin Change

এই ধরনের প্রব্লেম এর মুল জিনিস হল, তোমার কাছে কিছু কয়েন আছে ধর 1 টাকা, 2 টাকা 8 টাকা এর। তোমাকে একটা পরিমান বলা হবে ধরা যাক 50 টাকা। প্রশ্ন হল তুমি তোমার কাছে থাকা কয়েন গুলি ব্যবহার করে এই টাকা বানাতে পারবা কিনা? পারলে কত ভাবে পারবা? আবার যেই কয়েন গুলি দেয়া আছে সেগুলি কখনও কখনও বলা থাকে যে সেগুলি একবারের বেশি ব্যবহার করতে পারবে না কখনও কখনও বলা থাকে যে যত খুশি ব্যবহার করতে পারবে আবার কখনও কখনও একটা সীমা বলা থাকে।

৭.২.১ Variant 1

তোমাদের কিছু কয়েন দেয়া আছে এবং প্রতিটি কয়েন তুমি যত বার খুশি ব্যবহার করতে পারবে। মনে কর এই কয়েন গুলো হলঃ $coin[1 \ldots k]$. এখন প্রশ্ন হল তুমি n বানাতে পারবে কিনা?

ধরা যাক $possible[i]$ হল i পরিমান বানাতে পারব কিনা। যদি পারি তাহলে এর মান হবে 1 আর না পারলে 0. আমাদের বের করতে হবে $possible[n]$. তুমি স্বাভাবিক ভাবে চিন্তা কর তুমি যদি হাতে হাতে বের করতে চাইতে যে n বানানো সম্ভব কিনা কেমনে চিন্তা করলে ভাল হত? যেটা করা যায় তা হল, $n - coin[1]$ বা $n - coin[2]$ বা ... $n - coin[k]$ এর কোন একটি যদি বানানো সম্ভব হয় তাহলেই n বানানো সম্ভব নাহলে না। অর্থাৎ আমরা বড় একটি মান এর জন্য উত্তর বের করতে ছোট মানের সমাধান ব্যবহার করছি। এটাই DP! সুতরাং $1 \ldots n$ প্রতিটি মান এ গিয়ে তুমি k টি কয়েন ব্যবহার করে তুমি দেখবে যে ছোট মানটি বানানো যায় কিনা গেলে এই বড় মান ও বানানো যাবে। এর time complexity হল $O(nk)$.

Listing ৭.৩ : variant1.cpp

```
possible[0] = 1
for(i = 1; i <= n; i++)
    for(j = 1; j <= k; j++)
        if(i >= coin[j])
            possible[i] |= possible[i - coin[j]];
```

৭.২.২ Variant 2

তোমাদের কিছু কয়েন দেয়া আছে এবং প্রতিটি কয়েন তুমি যত বার খুশি ব্যবহার করতে পারবে। বলতে হবে n পরিমান তোমরা কত ভাবে বানাতে পারবে। এখানে কয়েন এর order এ যায় আসে। অর্থাৎ $1 + 3$ আর $3 + 1$ কে আমরা আলাদা বিবেচনা করব। অর্থাৎ তোমাকে যদি 1 আর 2 টাকার কয়েন

দেয়া হয় তাহলে তুমি 4 টাকা মোট 5 ভাবে বানাতে পারবেঃ $1+1+1+1$, $1+1+2$, $1+2+1$, $2+1+1$ এবং $2+2$.

ধরা যাক $way[n]$ হল কত ভাবে n বানানো যায়। এখন n বানানোর জন্য তুমি প্রথমে $coin[1]$ ব্যবহার করতে পার বা $coin[2]$ বা প্রদত্ত kটা কয়েন এর যেকোনো টি। যদি $coin[1]$ ব্যবহার কর তাহলে বাকি থাকে $n - coin[1]$ পরিমান যা তুমি $way[n - coint[1]]$ ভাবে বানাতে পারবে। অর্থাৎ আগের মতই কিছুটা! তোমার time complexity দাঁড়াবে $O(nk)$.

Listing ৭.4 : variant2.cpp

```
way[0] = 1
for(i = 1; i <= n; i++)
    for(j = 1; j <= k; j++)
        if(i >= coin[j])
            way[i] += way[i - coin[j]];
```

৭.২.৩ Variant 3

যদি আমাদের variant 1 এর সমস্যায় বলা হত যে প্রতিটি কয়েন তুমি একবারের বেশি ব্যবহার করতে পারবে না তাহলে?

খেয়াল কর আগের পদ্ধতিতে আমরা যা করেছি তাহল প্রতিটি n এ গিয়ে আমরা সব কয়েন নিয়ে চেষ্টা করেছি। ধর 10 এ গিয়ে 2 নিয়ে চেষ্টা করেছি আবার 8 এ গিয়েও। সুতরাং আসলে আমরা 10 বানানোর জন্য 2 কে একাধিক বার ব্যবহার করছিলাম যেটা এখন করা যাবে না! এর মানে আমরা এখন n বানানোর জন্য যদি i তম কয়েন ব্যবহার করতে চাই আমাদের দেখতে হবে, $n - coin[i]$ পরিমান $i-1$ পর্যন্ত কয়েন ব্যবহার করে বানানো যায় কিনা। অর্থাৎ আমরা বানাতে পারব কিনা সেটা এখন আর শুধু পরিমান এর উপর নির্ভর করতেসে না, কত পরিমান এবং কোন কয়েন পর্যন্ত ব্যবহার করা হয়েছে এই দুইটি জিনিস আমাদের জানতে হবে। আমরা যদি দেখতে চাই যে, n পরিমান i পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা তাহলে আমাদের দুইটা জিনিস দেখতে হবে তাহল n পরিমান $i-1$ পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা আর $n - coin[i]$ পরিমান $i-1$ পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা। অর্থাৎ আমাদের DP তে এখন দুইটি parameter. অর্থাৎ আমাদের 2D array লাগবে এই সমস্যা সমাধান করতে। এই সমাধানে আমাদের time ও space উভয় complexity ই $O(nk)$.

আমরা চাইলে space complexity কমিয়ে $O(n)$ করতে পারি। এজন্য খেয়াল কর, আমরা প্রথম i টা কয়েন ব্যবহার করে কোন কোন পরিমান বানাতে পারি সেটা জানার জন্য শুধু আমাদের জানতে হয় প্রথম $i-1$ টি কয়েন ব্যবহার করে কোন কোন পরিমান বানানো যায়। সুতরাং প্রতিবার আমাদের শুধু দুইটি row লাগে (প্রথম থেকে কয়টি কয়েন ব্যবহার করা হচ্ছে সেটি row আর কোন পরিমান বানাতে হবে সেটাকে column হিসাবে বিবেচনা করে দেখ)। আরও মজার ব্যপার হল এই আপডেট এর সময় যদি তুমি পরিমানের উরধক্রমে না গিয়ে বড় থেকে যদি ছোট তে যাও তাহলে কিন্তু দুইটি row এর দরকার হয় না, একটি হলেই হয়ে যায় কারন আমরা আগের পরিমান এর উপর আপডেট করতে পারি।

৭.২.৪ Variant 4

বুঝতেই পারছ আমরা Variant 3 এর জন্য জানতে চাইব কত ভাবে বানানো সম্ভব! এটা কিন্তু variant 2 এর মত হবে।

৭.২.৫ Variant 5

আমরা variant 2 তে $1+2+1$ এবং $2+1+1$ কে আলাদা ভেবেছিলাম কিন্তু যদি আলাদা না হয়? ধরা যাক $way[n][i]$ হল প্রথম i টি কয়েন ব্যবহার করে n কে কত ভাবে বানানো যায়। এখন

n বানানোর পথে আমরা প্রথমে $coin[i]$ ব্যবহার করতেও পারি নাও পারি। যদি ব্যবহার করি তাহলে মোট $way[n - coin[i]][i]$ উপায় আর যদি না করি তাহলে $way[n][i - 1]$ উপায়। শেষ :)

৭.৩ Travelling Salesman Problem

মনে কর তুমি একদিন রাজশাহী বেড়াতে গেলে। সেখানে তোমার n জন বন্ধুর বাড়ি। তুমি একে একে তাদের সবার বাড়ি যেতে চাও। তাদের সবার বাড়ির দুরত্ব তুমি জানো। তুমি প্রথমে গিয়ে তোমার সবেয়ে ভাল বন্ধু 1 এর বাসায় যাবে এর পর একে একে সবার বাসা ঘুরে আবারও 1 এর বাসায় ফেরত আসবে। সবচেয়ে কম মোট কত দুরত্ব অতিক্রম করে তুমি সবার বাসা ঘুরতে পারবে? এটি হল Travelling Salesman Problem. আমরা এতক্ষণ একটি প্রব্লেম কে DP ভাবে সমাধান করার জন্য যা করেছি তাহল বড় একটি সমস্যা কে ছোট সমস্যা দ্বারা সমাধান করেছি। আরেকটি উপায় হল একই রকম জিনিস খুজে বের করা। যেমন আমাদের এই সমস্যার ক্ষেত্রে খেয়াল কর, তুমি মনে কর $1 - 2 - 3 - 4$ এই ভাবে চার জন বন্ধুর বাসা ঘুরেছ বাকি আছে $5 \ldots n$ বন্ধুরা এই বাকি বন্ধুদের বাসা ঘুরতে তোমার যেই সবচেয়ে কম খরচ সেটা $1 - 3 - 2 - 4$ ঘুরার পর বাকি বন্ধু দের বাসা ঘুরে ফেলার জন্য সবচেয়ে কম খরচের সমান। অর্থাৎ, কোন এক সময় তোমাকে শুধু জানতে হবে তুমি কোন কোন বন্ধুর বাসা ঘুরে ফেলেছ এবং এখন তুমি কই আছ। বিভিন্ন ভাবে আমরা একই state এ আসতে পারি যেমন উপরের উদাহরনে আমরা প্রথম চার জন বন্ধুর বাসা দুই ভাবে ঘুরে এখন 4 এর বাসায় আছি। অর্থাৎ তোমার state হল তুমি কার কার বাসা ঘুরে ফেলেছ$(1, 2, 3, 4)$ আর এখন কই আছ(4)। এখন কই আছি সেটা শুধু একটা নাম্বার কিন্তু তুমি কই কই ঘুরে ফেলেছ এই জিনিস অনেক গুলি নাম্বারের সেট। আমরা DP এর সময় state কে array এর parameter হিসাবে লিখি। এই ক্ষেত্রে আমরা একটি সেট কে কেমনে নাম্বার আকারে লিখতে পারি? খেয়াল কর, আমাদের মোট n জন বন্ধু আছে, কার কার বাসায় গিয়েছি তাদের 1 আর কার কার বাসায় এখনও যাওয়া হয় নাই তাদের 0 দ্বারা লিখতে পারি। তাহলে n টা $0 - 1$ দ্বারা আমরা কার কার বাসায় গিয়েছি সেটা বানিয়ে ফেলতে পারি। কিন্তু তুমি যদি array এর dimension n টা নিতে চাও তাহলে নিশ্চয় কোড করা খুব একটা সুখকর হবে না? এখানে একটা মজার tricks আছে তাহল তুমি এই $0-1$ সংখ্যাকে binary ফর্ম এ ভাবতে পার। যেমনঃ তোমার যদি $1, 2, 4$ নাম্বার বন্ধুর বাসা ঘুরা হয়ে থাকে তাহলে তোমার নাম্বার হবেঃ $0000 \ldots 1011 = 7$. এখন এই সংখ্যা কত বড় হতে পারে? 2^n কারন একটি বন্ধু থাকতে পারে নাও পারে। তাহলে আমাদের state কত বড়? $n \times 2^n$ এবং প্রতি state এ গিয়ে তুমি অন্যান্য সবার বাসায় যাবার চেষ্টা করবে (n ভাবে)। সুতরাং আমাদের time complexity হবে $O(n^2 2^n)$. কোড

Listing ৭.5 : tsp.cpp

```
int dp[1<<20][20]; //Assume that there are 20 friends
//mask = friends i visited, at = last visited friend
int DP(int mask, int at)
{
        int& ret = dp[mask][at];
        //Assume that we initialized dp with -1
        if(ret != -1) return ret;

        ret = 1000000000; //initialize ret with infinity
        //dist contains distance between every two nodes
        for(int i = 0; i < n; i++) //n = number of friend
                if(!(mask & (1<<i)))
                        ret = MIN(ret, DP(mask | (1<<i), i) + dist[
                            at][i]);

        return ret;
}
```

৭.৪ Longest Increasing Subsequence

সংখ্যার একটি sequence আছে। এই sequence থেকে কিছু সংখ্যা (হয়তো একটিও না) মুছে ফেলতে হবে যেন বাকি সংখ্যা গুলি increasing order এ থাকে। আমাদের লক্ষ্য হল সবচেয়ে দীর্ঘ increasing subsequence[১] বানানো। আমরা যদি এটি DP এর মাধ্যমে সমাধান করতে চাই তাহলে ভাবতে হবে আমরা কোন ছোট সমস্যা সমাধান করতে পারি। ধরা যাক আমাদের কে n টি সংখ্যা দেয়া আছে। এর LIS (Longest Increasing Subsequence) বের করতে হবে। আমরা যদি প্রথম $n-1$ টির LIS জানি তাহলে কি কোন লাভ আছে? চিন্তা করে দেখা যাক। আমরা nতম সংখ্যা কে কার পিছে বসাব? এমন একটি সংখ্যার পিছে যা nতম সংখ্যার থেকে ছোট ধরা যাক $a[i]$ (a হল মূল sequence এবং $i < n$). এখন এরকম তো অনেক $a[i]$ আছে যেন $a[i] < a[n]$ কিন্তু কোনটির পিছনে? যে সবচেয়ে ছোট তার পিছনে? না, কারনঃ $1, 2, 3, 4$ এদের মাঝে কার পিছনে আমরা ধর 5 কে বসাতে চাইব? নিশ্চয় 1 না। আমরা 4 এর পিছনে বসাতে চাইব কারন প্রথমত 5 এর থেকে 4 ছোট আর দ্বিতীয়ত এই 4 পর্যন্তই সবচেয়ে বড় LIS আছে। সুতরাং আমাদের যা করতে হবে তাহল প্রথম n টি সংখ্যার LIS ($LIS[n]$) হল, $LIS[i]+1$ এর মাঝে সবচেয়ে বড় মান যেখানে $a[i] < a[n]$। এই পদ্ধতির time complexity $O(n^2)$. আমরা খুব সহজেই Segment Tree ব্যবহার করে এটিকে $O(n \log n)$ করতে পারি। আমরা n এ এসে $1 \dots a[n]-1$ পর্যন্ত query করব আর শেষে $a[n]$ এ আপডেট করব।

৭.৫ Longest Common Subsequence

দুইটি string: S এবং T দেয়া থাকবে, আমাদের এমন একটি string বের করতে হবে যা S এবং T উভয়েরই subsequence হয় এবং সবচেয়ে longest হয়। এই প্রব্লেম এর ক্ষেত্রে আমাদের state হবেঃ যদি আমাদের S এবং T সম্পূর্ণ ভাবে না দিয়ে S এর প্রথম s টি এবং T এর প্রথম t টি letter দেয়া হয় তাহলে Longest Common Subsequence (LCS) কত? যদি S[s] = T[t] হয় (1 indexing ধরে) তাহলে কিন্তু আমাদের উত্তর হল S এর প্রথম $s-1$ এবং T এর $t-1$ এর যত উত্তর তার থেকে এক বেশি। আর যদি $S[s] \neq T[t]$ হয় তাহলে S এর প্রথম $s-1$ ও T এর প্রথম t letter এর ক্ষেত্রে উত্তর আর S এর প্রথম s ও T এর $t-1$ letter এর ক্ষেত্রে উত্তর এর মাঝে যেটি বড় সেটি। এখন যদি আমাদের শুধু উত্তরের দৈর্ঘ্য নয় সেরকম একটি string ও প্রিন্ট করতে বলে তাহলে আমরা প্রথমে dp table অর্থাৎ উত্তরের table বানিয়ে নেব। এর পর দুইটি string এরই শেষ থেকে আসতে হবে। যদি শেষ character দুইটি একই হয় তাহলে আমরা সেটি নেবই। আর না হলে আমরা dp টেবিল থেকে দেখব যে কোন string থেকে শেষ character বাদ দেওয়া উচিত, প্রথম না শেষ। খেয়াল করে দেখ, এভাবে করলে সমস্যা হল আমরা string টা উলটো দিক থেকে তৈরি করতেসি। সুতরাং যেহেতু আমাদের string কে সামনের দিক থেকে প্রিন্ট করতে হবে সেহেতু হয় আমাদের কোন একটি স্ট্রিং এ character গুলি নিয়ে পরে reverse করে প্রিন্ট করতে হবে অথবা এই পুরো কাজটা recursively করতে হবে। আরেকটা বুদ্ধি হল আমরা যদি সামনের দিক থেকে dp না করে যদি পিছন থেকে dp করি তাহলে আর string উলটো করার ঝামেলা থাকে না। সাধারন একটা while লুপ দিয়েই আমরা পুরো স্ট্রিং প্রিন্ট করে ফেলতে পারি। খেয়াল কর, আমি এখানে অনেক উপায়ে স্ট্রিং প্রিন্ট (একে generally বলা হয় path printing. যেকোনো dp সমস্যায় শুধু মান না মান টা কেমনে হয় সেটাও চেয়ে থাকে, একে আমরা path printing বলে থাকি।) করার পদ্ধতি বললাম, একেক সমস্যার ক্ষেত্রে একেক উপায়ে path print করা সহজ হয়।

[১] একটি sequence থেকে কিছু সংখ্যা মুছে ফেললে যা বাকি থাকে তাই subsequence. এখানে কিন্তু বাকি সংখ্যা গুলির order একই থাকতে হবে।

অধ্যায় ৮

গ্রাফ

আগেই বলে রাখি যেকোনো গ্রাফ এর algorithm কোড করার সময় stl ব্যবহার করলে কোড অনেক ছোট ও simple হয়। সুতরাং আমরা এই চ্যাপটার এ যেসব কোড দেখাব তাদের বেসির ভাগেই stl ব্যবহার করা। কোড গুলি বুঝতে stl খুব ভালভাবে বুঝতে হবে তা না। তুমি শুধু মাথায় রেখো যে আমরা কি কোড করছি আর কোন লাইনে কি করতে চাওয়া হচ্ছে তাহলেই তোমরা stl এর কোডগুলি বুঝতে পারবে। ব্যবহার দেখতে দেখতে কোন জিনিস শিখলে সে জিনিস মনে থাকে অনেক দিন।

৮.১ Breadth First Search (BFS)

এটি একটা গ্রাফ এ ঘুরে বেড়ানোর (traverse) একটি টেকনিক। মনে কর গ্রাফের s নোড হতে তুমি ঘুরা শুরু করবে। তুমি যা করতে পারো তাহল, শুরুর নোড এর থেকে যেখানে যেখানে যাওয়া যায় সেখানে সেখানে যাবে, এর পর তাদের থেকে নতুন নতুন যেখানে যাওয়া যায় সেখানে যাবে এভাবে যত জায়গায় যাওয়া সম্ভব সবখানে যাবে। যেখানে একবার গিয়েছো সেখানে তো আবার যাবার কোন মানে নাই, তাই যখন কোন edge দিয়ে অন্য প্রান্তে যাবার সময় দেখবে যে অন্য প্রান্ততে already গিয়েছ তখন আর সেখানে যাবার কোন মানে নাই। যদি তোমার মোট vertex থাকে V টা এবং edge থাকে E টা তাহলে আমাদের এই ঘুরে বেড়াতে সময় লাগবে $O(V + E)$. কারন তোমরা প্রতি নোড এ একবারের বেশি যাচ্ছ না আর প্রতি edge এর ক্ষেত্রে দুই মাথা থেকে তুমি দুইবার যাবার চেষ্টা করবে। এর কোড ৮.1 এ দেয়া হল।

Listing ৮.1 : bfs.cpp

```
1  #include<vector>
2  #include<queue>
3  using namespace std;
4
5  vector<int> adj[100]; // adj[a].push_back(b); for an edge from a to
       b.
6  int visited[100]; // 0 if not visited, 1 if visited
7
8  // s is the starting vertex
9  // n is the number of vertices (0 ... n - 1)
10 void bfs(int s, int n)
11 {
12         for(int i = 0; i < n; i++) vis[i] = 0;
13
14         queue<int> Q;
15         Q.push(s);
16         visited[s] = 1;
17
```

```
18          while(!Q.empty())
19          {
20                  int u = Q.front();
21                  Q.pop();
22
23                  for(int i = 0; i < adj[u].size(); i++)
24                          if(visited[adj[u][i]] == 0)
25                          {
26                                  int v = adj[u][i];
27                                  visited[v] = 1;
28                                  Q.push(v);
29                          }
30          }
31  }
```

৮.২ Depth First Search (DFS)

এটি গ্রাফে ঘুরে বেড়ানোর আরেকটি উপায়। এই পদ্ধতিতে যা করা হয় তাহল শুরু নোড s এ শুরু করে তুমি যেতেই থাকবে, যতক্ষণ না তুমি এমন নোড এ পৌছাও যেখানে এর আগে এসেছিলে। সেরকম কোন নোডে পৌঁছালে তোমাকে পিছিয়ে যেতে হবে। পিছিয়ে গিয়ে তুমি অন্য edge দিয়ে যাবার চেষ্টা করবে। খেয়াল কর, এই কাজটা কিন্তু কিছুটা recursive গোছের। তুমি একটা নোডে আছো, তোমার কাজ হল এর কোন একটা edge দিয়ে বের হওয়া। যদি গিয়ে দেখ যে সেখানে আগেই এসেছিলে তাহলে ফিরে এসো আর না হলে recursively একই কাজ কর। এর কোড ৮.2 এ দেয়া হল। একটি জিনিস বলে রাখি, তুমি চাইলে dfs ফাংশন call করার আগেই কিন্তু চেক করে দেখতে পারো যে যেখানে তুমি এখন যেতে চাচ্ছো সেখানে আগেই গিয়ে ছিলেকিনা। অনেক সময় দেখা যায় যে dfs ফাংশন call করার cost অনেক বেশি (অনেক সময় বড় বড় state পরিবর্তন করতে হয়) সেক্ষেত্রে আগে থেকে চেক করে যাওয়া বুদ্ধিমানের মত কাজ।

Listing ৮.2 : dfs.cpp

```
1   #include<vector>
2   using namespace std;
3
4   vector<int> adj[100];
5   int vis[100];
6
7   // call it by dfs(s)
8   // before calling, make vis[] all zero.
9   void dfs(int at)
10  {
11          if(vis[at]) return; // if previously visited.
12          vis[at] = 1;
13
14          for(int i = 0; i < vis[at].size(); i++)
15                  dfs(vis[at][i]);
16  }
```

এই কোডের complexity ও কিন্তু $O(V+E)$ কিন্তু এর একটি সমস্যা হল এতে সর্বোচ্চ V টি পর পর recursive call হতে পারে। সুতরাং আমাদের compiler এর default stack size যদি কম হয় তাহলে এই কোডে stack overflow হতে পারে। বেশির ভাগ সময়ই online judge গুলিতে এই সমস্যা হয় না। কিন্তু যদি সমস্যা হয় তাহলে আমাদের পুরোপুরি manually এই recursive এর কাজ stack এর মাধ্যমে করতে হবে। stack ব্যবহার করে DFS এর কোড ?? এ দেয়া হল।

Listing ৮.3 : dfsStack.cpp

```cpp
#include<vector>
#include<stack>
using namespace std;

vector<int> adj[100];
int edge_id[100];
int vis[100];

// s is starting vertex
// n is number of vertices
void dfs(int s, int n)
{
        for(int i = 0; i < n; i++) edge_id[i] = vis[i] = 0;

        stack<int> S;
        S.push(s);
        while(1)
        {
                int u = S.top();
                S.pop();

                while(edge_id[u] < adj[u].size())
                {
                        // start looking into edges, from the place we left
                        int v = adj[u][edge_id[u]];
                        edge_id[u]++; // update edge pointer to check next time
                        if(vis[v] == 0) // if the vertex is not already visited
                        {
                                vis[v] = 1;
                                S.push(u); // order of push important for dfs
                                S.push(v); // first we will check v, then we will come back to u
                                // note, stack is last in first out. So v will be popped before u
                                break;
                        }
                }
        }
}
```

৮.৩ DFS ও BFS এর কিছু সমস্যা

৮.৩.১ দুইটি node এর দুরত্ব

মনে কর একটি গ্রাফ আর দুইটি নোড দিয়ে বলা হল যে তাদের দুরত্ব বের কর। দুরত্ব বলতে কয়টি edge পার করে যেতে হয় সেটা বুঝানো হচ্ছে এখানে। কেমনে করবে? এটি কিন্তু BFS দিয়ে খুব সহজেই সমাধান করা যায়। খেয়াল করে দেখ, BFS এর ক্ষেত্রে কিন্তু শুরুর নোড থেকে যেই পথে অন্য একটি নোড এ যাওয়া হয় তা কিন্তু shortest path এই যাওয়া হয়। সুতরাং যখন আমরা কোন নোড থেকে আরেকটি নতুন নোড এ যাব তখন আমরা বলতে পারি নতুন নোড এর দুরত্ব যেখান থেকে আসা হচ্ছে তার থেকে এক বেশি। সুতরাং আমাদের যেই দুইটি নোড দেয়া আছে তাদের একটি থেকে BFS শুরু করলে আমরা অন্য নোড এ যাবার দুরত্ব পেয়ে যাব।

খেয়াল কর আমরা কিন্তু এই সমস্যা DFS দিয়ে সমাধান করতে পারব না। কারন DFS দিয়ে সব-সময় shortest path এ যাওয়া হয় না। মনে কর A, B ও C তিনটি নোড। প্রত্যেকটি থেকে অন্য দুইটি নোড এ যাওয়া যায়। এখন A হতে DFS শুরু করলে আমরা B তে যাব এর পর C তে যাব। খেয়াল কর, A থেকে C তে এক ধাপে যাওয়া গেলেও আমরা DFS করে গেলে দুই ধাপে যাচ্ছি।

এখন যদি আমাদের এই দুইটি নোডের মাঝে shortest path টাও প্রিন্ট করতে বলে? path printing টেকনিক মোটামোটি সব ক্ষেত্রেই একি রকম। তুমি কোন নোডে যাবার সময় লিখে রাখবে এখানে কেমনে এসেছ। যেমন BFS এর ক্ষেত্রে তুমি কোন নতুন নোডে আসার সময় লিখে রাখবে কোন নোড থেকে এখানে এসেছ। তাহলেই হবে।

তিনটি গ্লাস ও পানি

খুব কমন একটি পাজল হল, তোমাকে 3 ও 5 লিটারের দুইটি ফাঁকা গ্লাস আর 8 লিটারের একটি পানি ভর্তি গ্লাস দেয়া থাকলে তুমি 4 লিটার পানি আলাদা করতে পারবে কিনা। যদি শুধু প্রশ্ন হয় পারবে কিনা তাহলে DFS করেই করতে পারবে। আর যদি চায় সবচেয়ে কম কতবার ঢালাঢালি করে? তাহলে তোমাকে BFS করতে হবে। এটা বুঝা গেল যে এটা BFS দিয়ে সমাধান করা যাবে কিন্তু নোড ই বা কই আর edge ই বা কই? আসলে BFS করতে যে আসলে vertex বা edge লাগবে এই ধারনা ঠিক না। আমাদের যা জানতে হবে তাহল আমরা কোথায় আছি আর এখান থেকে আমরা কোথায় কোথায় যেতে পারি। কিছুটা DP এর মত চিন্তা করতে পারো। আমরা যেখানে আছি সেটাকে একটা state আকারে represent করতে হবে- এটাই মূল জিনিস। যেমন আমাদের এই প্রবলেম এর state হতে পারে তিনটি পাত্রে কত খানি করে পানি আছে। সুতরাং আমরা 1-dimensional array তে visited না রেখে একটা 3-dimensional array তে visited রাখতে পারি আর queue তে একটি নাম্বার না রেখে তিনটি নাম্বার একত্রে structure করে রাখতে পারি (structure এর queue). এভাবে BFS করলেই এই সমস্যা সমাধান হয়ে যাবে।

এই সাবসেকশন শেষ করার আগে আরেকটি জিনিস, তা হল তোমরা চাইলে কিন্তু এই state কে দু-ইটি নাম্বার দিয়ে represent করতে পারো আর তাহল প্রথম দুই পাত্রে কত খানি করে পানি আছে। কা-রন 8 থেকে ঐ পরিমান বাদ দিলে তুমি তৃতীয় পাত্রের পানির পরিমান পেয়ে যাবে। এই optimization এর ফলে তোমার run time এ কোন পরিবর্তন হবে না তবে memory কম লাগবে। তোমরা চাইলে queue তে তিনটি নাম্বার ই রাখতে পারো এতে করে কষ্ট করে 8 থেকে বিয়োগ করার কাজ করতে হবে না আবার সেই সাথে আমরা visited রাখার সময় প্রথম দুইটি সংখ্যা ব্যবহার করব যাতে আমাদের memory কম লাগে। এই ট্রিক্সটা প্রায়ই কাজে লাগে। যাতে বেশি computation এর দরকার না হয় সেজন্য আমরা queue তে সব জিনিসই রেখে দেব কিন্তু visited বা memorization এর জন্য যাতে কম জায়গা লাগে সেজন্য আমরা ছোট visited matrix ব্যবহার করব।

UVa 10653

তোমাকে একটি গ্রিড দেয়া থাকবে। সেই সাথে তোমার শুরুর জায়গা আর শেষ গন্তব্য দেয়া থাকবে। তোমাকে সবচেয়ে কম কত সময়ে গন্তব্যে পৌঁছান যায় তা বলতে হবে। এটা খুব ভাল মতই বুঝা যাচ্ছে যে গ্রিড এর একেকটি সেল হল একেকটি নোড। তোমরা যারা প্রথম প্রথম প্রোগ্রামিং করতেসো তারা হয়তো প্রতি সেল কে $1, 2, \ldots RC$ এভাবে নাম্বার দিবে কিন্তু এর থেকে সুবিধা হবে তুমি যদি নোডকে (r, c) ভাবে represent কর। আর ৮.4 এর মত দুইটি matrix রাখ।

Listing ৮.4 : cellbfs.cpp

```
1  int dr[] = {-1, 0, 1, 0};
2  int dc[] = {0, 1, 0, -1};
3
4  int valid(int r, int c)
5  {
6          return r >= 0 && r < R && c >=0 && c < C;
7          // also may be check if (r, c) is empty.
```

```
8         // you may also check if the cell is visited by bfs.
9 }
```

তাহলে (r, c) থেকে নতুন যেসব cell এ যেতে পারবে সেসব হল $(r + dr[i], c + dc[i])$ (i এর একটি লুপ 0 হতে 3 পর্যন্ত চালাও) আর এই নতুন cell টি আদৌ valid কিনা তা জানার জন্য ৮.4 কোড এর valid ফাংশনকে call করে দেখ।

UVa 10651

এটি তুমি BFS বা DFS যেকোনো টি দিয়েই সমাধান করতে পারবে। কারন এখানে সর্বনিম্ন কয়টি peeble থাকবে অর্থাৎ কোন কোন game configuration এ যাওয়া যাবে সেটিই মূল জিনিস।

0 ও 1 cost এর গ্রাফ

ধরা যাক তোমাকে একটি weighted গ্রাফ দেয়া হল যার edge cost হয় 0 নাহয় 1. এই ক্ষেত্রে এক নোড থেকে আরেক নোড এ যাবার shortest path বের করার একটি সহজ উপায় হল BFS এর মত কাজ করা। তুমি যখনি 0 দিয়ে যেতে চাইবে তখন queue এর শুরুতে add করবা, আর 1 দিয়ে যেতে চাইলে queue এর শেষে। আসলে দুই দিকে add করতে পারলে সেটা আর queue থাকে না, এর আরেক নাম হল deque. STL এ deque বলে built-in ডাটা স্ট্রাকচার আছে। আর নেবার সময় সবসময় সামনে থেকে নিবা। এক্ষেত্রে deque এর সাইজ n এর থেকে বেশ বড় প্রায় m এর সমান হয়ে যেতে পারে মনে হয়। এজন্য তোমরা যদি একটা $dist$ এর অ্যারে রাখো তাহলে এই সাইজ কমিয়ে $2n$ বানিয়ে ফেলতে পারো।

৮.৪ Single Source Shortest Path

Shortest Path প্রবলেম হল, কোন একটা weighted graph এ এক নোড থেকে আরেক নোড এ যাবার সর্বনিম্ন cost বের করার প্রবলেম। সাধারনত আমরা দুই ধরনের Shortest Path প্রবলেম দেখে থাকি। Single source shortest path এবং All pair shortest path. Single source shortest path প্রবলেম এ আমরা এক নোড থেকে অন্য সকল নোড এ যাবার সর্বনিম্ন cost বের করে থাকি আর All Pair Shortest Path প্রবলেম এ আমাদের প্রতিটি নোড থেকে অন্য সকল নোড এ যাবার cost বের করতে হয়। তোমরা হয়তো ভাবতে পারো যে তাহলে Single Source Single Destination Shortest Path বলে আরও একটা কিছু বলছি না কেন? কারন হল, Single Source Shortest Path এর মাধ্যমে এই Single Destination এর variation টা সল্ভ করা যায় আর তাছাড়া মূল কারন হল, Single Destination এর variation সমাধান করার জন্য আসলে simpler কোন algorithm নেই। খেয়াল কর আমি কিন্তু simpler বলেছি। আমি এখানে Single Source Shortest Path এর সাথে তুলনা করছি। অর্থাৎ, আমরা Single Source Shortest Path এর জন্য যেসকল algorithm দেখব, Single Destination এর জন্য তার থেকেও efficient algorithm আসলে আমার জানা নেই। তবে হ্যা, হয়তো খুবই সামান্য optimization করতে পারবে কিন্তু আসলে worst case এ একই time complexity পাবে। এসব কথা এখন বুঝতে না পারলেও সমস্যা নেই, নিচের algorithm গুলি বুঝে এসে এই কথা গুলো পড়লে আসা করি বুঝতে পারবে আমি কোন optimization এর কথা বলছি, বা কেন বলছি যে Single Destination এর variation এ আমরা Single Source Shortest Path এর algorithm ই ব্যবহার করব।

Single Source Shortest Path এর জন্য দুইটি algorithm খুব বেশি ব্যবহার করা হয়। আসলে এই দুইটির বাইরে আমার জানাও নেই। একটি হল Dijkstra's Algorithm[১] আর আরেকটি হল BellmanFord Algorithm.

[১] আমি আসলে জানি না আসল উচ্চারন কি! অনেকে অনেক ভাবে উচ্চারন করে। আমিও বিভিন্ন বয়সে বিভিন্ন উচ্চারন করতাম, ছোট বেলায় ডিজিকস্ট্রা বড় হয়ে ডায়াকস্ট্রা। মাঝে মনে হয় আরও অনেক কিছুই বলতাম। তবে মোটামোটি সবাই ডায়াকস্ট্রা ই বলে অভ্যস্ত।

৮.৪.১ Dijkstra's Algorithm

সাধারনত এই algorithm ব্যবহার করা হয় যদি সব edge এর cost অঋণাত্মক (non-negative) হয়। একে বিভিন্ন ভাবে implement করলে বিভিন্ন time complexity পাওয়া সম্ভব। আমরা $O(n^2)$ দিয়ে শুরু করি।

- প্রথমে একটি n সাইজের array নেই, ধরা যাক এর নাম dist (distance এর সংক্ষিপ্ত রূপ) এবং এর প্রতিটি element কে ইনফিনিটি cost দেই। অনেকে ইনফিনিটি হিসাবে খুব বড় সংখ্যা যেমন $1'000'000'000$ ব্যবহার করে থাকে। অনেক সময় কেউ কেউ -1 কে ব্যবহার করে থাকতে পারে। তুমি যাই ব্যবহার কর না কেন তোমার বাকি কোডটা সেই অনুসারে লিখলেই হবে।

- যেহেতু আমরা Single Source Shortest Path সমাধান করতেছি, সুতরাং আমাদের কাছে একটি source বা যেখানে থেকে আমাদের যাত্রা শুরু সেই নোড আছে। ধরে নেই সেটা s। তাহলে আমরা উপরের array তে s এর পজিশনে 0 বসাবো। এর মানে হল, s এ পৌঁছানর cost হল 0.

- আরও একটা n সাইজের array নেই যার নাম ধরা যাক visited এবং এর প্রতিটি স্থান 0 দ্বারা initialize করি।

- এখন আমাদের এই ধাপটা বার বার করতে হবে। এই ধাপে আমরা দেখব কোন কোন নোড এর visited এ 0 আছে, তাদের মাঝ থেকে যেই নোডের cost, dist array তে সবচেয়ে কম তাকে select করি। ধরা যাক সেই নোডটা হল u. এখন visited array তে u এর পজিশনে 1 বসিয়ে দেই। এবার আমরা দেখব u থেকে কোথায় কোথায় যাওয়া যায়? ধরা যাক, u থেকে v তে যাবার জন্য একটি edge আছে যার cost হল c. আমরা দেখব কোনটি ছোট $dist[v]$ নাকি $dist[u] + c$? অর্থাৎ আমরা দেখতে চাচ্ছি যে v তে already যেভাবে যাওয়া যায় সেটা ভাল নাকি আমরা যদি প্রথমে u তে এসে এর পর $u -> v$ edge ব্যবহার করে যাই তাহলে সেটা ভাল হবে। যদি $dist[u]+c$ কম হয় তাহলে $dist[v]$ কে এই মান দিয়ে update করি।[১] এভাবে আমরা একে একে u এর সাথে লাগান সব edge চেক করব।[২] এভাবে যতক্ষণ না আমাদের সব নোড visited হয়ে যায় (অর্থাৎ visited array তে সবাই 1 হওয়া পর্যন্ত) ততক্ষণ আমরা এই প্রসেস চালাতে থাকব।

- এখন তুমি dist এর array তে s থেকে সকল নোড এর সর্বনিম্ন distance তুমি পেয়ে যাবে।

এখানে আমাদের তৃতীয় ধাপ কিন্তু চলবে n বার। এবং প্রতিবার আমরা সব নোড পর্যবেক্ষণ করে বের করছি আমাদের ঐ ধাপের u কে হবে। সুতরাং আমরা n বার n সমান কাজ করছিঃ $O(n^2)$. এর পরে প্রতি u এর জন্য আমরা এর সাথে লাগান edge গুলি চেক করছি, এর মানে হল প্রতিটা edge আসলে খুব জোর 2 বার চেক হবে। সুতরাং আমাদের complexity হবে $O(n^2 + m)$ বা $O(n^2)$ (কারন $m \leq n^2$).

এখন আমরা এই complexity কে চাইলেই কমিয়ে $O(m \log m)$ করতে পারি। খেয়াল করে দেখ আমরা একটি ধাপে লুপ চালিয়ে কোন unvisited নোড মিনিমাম সেটা বের করেছিলাম। তা না করে আমরা চাইলে STL এর priority queue ব্যবহার করতে পারি। যা করতে হবে তা হল, একটি structure এর priority queue বানাতে হবে। এর পর যখনি কোন নোড এর cost আপডেট করা হবে তখনই সেই নোডকে cost সহ priority queue তে পুশ করে দিতে হবে। আর প্রতিবার মিনিমাম cost এর নোড সিলেক্ট করার সময় priority queue থেকে মিনিমাম cost এর strcut এর object নিয়ে দেখতে হবে সেখানে যে নোড আছে সেটা কি visited কিনা। যদি visited হয়ে থাকে তাহলে এটা নিয়ে আর কাজ করার দরকার নেই। খেয়াল কর আমরা এই method এ কিন্তু একটি নোড একাধিকবার পুশ করছি। আমরা ধরে নিতে পারি প্রতি edge এর জন্য একবার করে পুশ হয় কোন না

[১] update করা মানে হল পরিবর্তন করা, বা ভাল মান দিয়ে পরিবর্তন করা।

[২] খেয়াল কর, আমাদের মূল লক্ষ্য হল u থেকে যেই যেই edge দিয়ে যাওয়া যায় তাদের update করা। সুতরাং আমাদের গ্রাফ directed বা undirected যাই হোক না কেন সেইভাবে কাজ করলেই এই algorithm কাজ করবে।

কোন নোড। সুতরাং m বার পুশ হয় m সাইজের heap বা priority queue তে, সুতরাং আমাদের complexity $O(m \log m)$.

যারা মনোযোগ দিয়ে পড়েছ আসা করি বুঝতে পারছ যে আমাদের আরও optimization এর সুযোগ আছে। আমরা যদি priority queue তে বার বার পুশ না করে আগের পুশ করা নোড এর cost আপডেট করতে পারতাম তাহলে কিন্তু complexity কমে যেতো। সুতরাং তোমাদের যদি আরও efficient করার প্রয়োজন হয় তাহলে নিজেরা heap বানিয়ে করতে পারো কিন্তু এতে আরও অনেক কোড করতে হয় বলে আমরা সহজে নিজে থেকে heap বানাই না।

যারা STL এর set সম্পর্কে জানো তারা হয়তো মনে করতে পারো priority queue ব্যবহার না করে set ব্যবহার করলে তো complexity আরও কমতে পারে! কারন set এ চাইলে remove করা যায়, সুতরাং আমরা আমাদের complexity $O(m \log n)$ তে নামিয়ে ফেলতে পারি। কিন্তু সমস্যা হল, set এর internal algorithm অনেক কমপ্লেক্স আরও definitely বলতে গেলে বলতে হয়, priority queue আসলে একটা heap আর set আসলে একটা red black tree. Red black tree এর internal structure অনেক কমপ্লেক্স বিধায় এদের দুজনের মোটামোটি সব অপারেশন এর complexity $O(\log n)$ হলেও set এর constant factor আসলে বেশি। সুতরাং বেশির ভাগ সময়েই দেখা যায়, priority queue, set এর থেকে dijkstra algorithm এ ভাল perform করছে। তবে যদি কখনও তোমরা খুব dense গ্রাফ এর সম্মুখিন হউ অর্থাৎ $m \approx n^2$ তাহলে priority queue না ব্যবহার করে set ব্যবহার করলে ভাল performance পাবা।

এখন আশা করি নিজেরাই বুঝতে পারছ কেন এই algorithm ঋণাত্মক edge cost এর ক্ষেত্রে কাজ করবে না। ঋণাত্মক edge cost থাকলে বড় সমস্যা হল এই algorithm অনুসারে গ্রাফে প্রসেস করতে থাকলে এক সময় দেখা যাবে visited নোড এরও cost কমবে কিন্তু আমরা এই algorithm এ visited নোড কে দুই বার প্রসেস করি না। যদি আমরা বার বার প্রসেস করতাম আর গ্রাফে negative cycle না থাকত তাহলে এই algorithm ই negative edge cost এও কাজ করত তবে সে ক্ষেত্রে আমাদের complexity আসলে খুব একটা ভাল হবে না, আমি নিজেও নিশ্চিত না complexity কত হবে, মনে হয় $O(m^2 log m^2)$ এর মত কিছু হবে। তবে এভাবে যে negative edge cost ওয়ালা গ্রাফে shortest path বের করা সম্ভব তা জেনে রাখা ভাল। যদি আমার দুর্বল স্মৃতিশক্তি আমার সাথে দুষ্টুমি না করে তাহলে আমার মনে হয় আমাকে এভাবেও কিছু প্রবলেম সল্ভ করতে হয়েছিল।

৮.৪.২ BellmanFord Algorithm

এটিও single source shortest path বের করার একটি algorithm এবং এটি dijkstra এর তুলনায় অনেক সহজ এবং ঋণাত্মক edge cost এ এটি কাজ করে। তাহলে আমরা dijkstra শিখলাম কেন? কারন এর complexity $O(mn)$ যা dijkstra এর তুলনায় অনেক বেশি। যদি গ্রাফে negative cycle ও থাকে তাহলে এই algorithm তা বুঝতে পারে। negative cycle হল গ্রাফের এমন একটি cycle যেখানে edge cost এর sum ঋণাত্মক হয়। অর্থাৎ তুমি একটা নোড থেকে শুরু করে বিভিন্ন edge হয়ে আবার শুরুর নোড এ ফিরে আসবে আর দেখতে পাবে যে তোমার edge cost এর sum ঋণাত্মক হয়ে গেছে। এটি কেন সমস্যার কারন বুঝতে পারছ তো? কারন হল, negative cycle এ আছে এমন একটি নোড এ তুমি যদি যেতে পারো তাহলে সেই নোড এ পৌঁছানর খরচ তুমি কিন্তু negative cycle ব্যবহার করে কমাতেই থাকতে পারো। সুতরাং ঐ সকল নোড এর minimum cost আসলে undefined বা negative infinity বা এরকম অনেক কিছুই বলতে পারো। অনেক প্রবলেমই আছে যেখানে তোমাকে বলতে বলবে কোন কোন নোড এরকম negative cycle এ আছে বা শুরুর নোড থেকে কোন কোন নোড এ যাওয়া যায় যারা negative cycle এ আছে। এসব ক্ষেত্রে আমরা BellmanFord algorithm ব্যবহার করতে পারি। খেয়াল কর, negative cycle এ থাকা মানেই শুরুর নোড থেকে negative infinity cost এ পৌঁছান না!!

এই algorithm কে দুইটি অংশে ভাগ করা যায়।

প্রথম অংশে আমরা shortest path বের করব। প্রথমত আমাদের একটি n সাইজের $dist$ এর অ্যারে নিতে হবে যার সকল element হবে infinity কেবল source হবে 0. এখন আমাদের একটি কাজ n বার করতে হবে। কাজটি হল, সব edge একে একে নিতে হবে,ধরা যাক একটি edge হল a থেকে b তে এবং তার cost হল c (যদি গ্রাফটি bidirectional হয় তাহলে অন্য দিকের edge টাও

আলাদা ভাবে consider করতে হবে)। এখন তোমাকে দেখতে হবে, $dist[b]$ বড় নাকি $dist[a] + c$ বড়। সেই অনুসারে আপডেট করতে হবে। এই ধাপটা n বার চালালেই তোমাদের shortest path বের হয়ে যাবে। খেয়াল কর, আমরা বাইরের লুপ চালাচ্ছি n বার আর ভিতরের edge এর লুপ চলছে m বার সুতরাং আমাদের complexity হবে $O(mn)$. তোমরা চাইলে এখানে একটা ভাল optimization করতে পারো এবং এটি প্রায়ই কাজে লাগে বিশেষ করে যখন mincost maxflow তে আমরা যখন bellman-ford ব্যবহার করে থাকি[১]। optimization টা হল, আমরা যখনি দেখব n এর লুপের ভিতরের edge এর লুপে কোন edge কোন আপডেট ঘটায় নাই, তাহলে n এর লুপ কে break করে ফেলো। কারন পরের অন্য কোন লুপে আর কোন আপডেট হবে না। আর আরেকটা কাজও করতে পারো, সেটা হল, bellman ford চালানর আগে edge গুলোর order তুমি randomize করে ফেলো। এতে সুবিধা হল কেউ যদি bellman ford এর জন্য বাজে case বানায়ও, তুমি edge এর order পরিবর্তন করে ফেলায় সেটা আর থাকবে না।

এখন দ্বিতীয় অংশে আশা যাক। দ্বিতীয় অংশে আমরা বের করব গ্রাফে negative cycle আছে কিনা। এটা করার জন্য যা করতে হবে তা হল, আমাদের আগের n এর লুপ এর ভিতরের অংশ আরেকবার চালাতে হবে। যদি দেখ এই $n + 1$ তম বারে আবারও কোন edge দ্বারা নোড এর cost আপডেট করা যায় তাহলেই বুঝবা যে তোমার গ্রাফে negative cycle আছে।

৮.৫ All pair shortest path বা Floyd Warshall Algorithm

আমাদের dijkstra algorithm এর complexity ছিল $O(m \log n)$ এর মত। আমরা যদি সব নোড থেকেই dijkstra চালাতাম তাহলে all pair shortest path বের করতে সময় লাগত প্রায় $O(nm \log n)$ এর মত। যদি গ্রাফ টা খুব একটা dense না হয় ($m \approx n^2$) তাহলে n বার dijkstra চালানই ভাল। সত্যি কথা বলতে যেখানে floyd warchall চালাতে হবে সেখানে বার বার dijkstra চালালেই হয়ে যাবার কথা। তাহলে আমরা floyd warshall শিখব কেন? এর একমাত্র কারন হল এর কোড খুবই ছোট ও সহজ। মাত্র পাঁচ লাইন আর সেই পাঁচ লাইনের মাঝে তিনটি for-loop। এটি মনে রাখাও খুব সহজ। প্রথমেই আমরা algorithm টা দেখে নেইঃ

Listing ৮.5 : apsp.cpp

```
for(k = 1; k <= n; k++)
  for(i = 1; i <= n; i++)
          for(j = 1; j <= n; j++)
                  if(w[i][j] > w[i][k] + w[k][j])
                          w[i][j] = w[i][k] + w[k][j];
```

শুধু মনে রাখতে হবে যে, প্রথমে k এর লুপ এর পর i আর j. এখানে শেষ দুই লাইনে কি করা হচ্ছে তাতো বুঝতে পারছ? দেখা হচ্ছে যে, i থেকে j তে যাবার cost কি k হয়ে যাওয়ার cost থেকে ভাল না খারাপ। এর পর আমরা ভাল cost দিয়ে আপডেট করে দেব। তোমরা যারা এখনও ভাবছ যে w এর array তে কি আছে তাদের জন্য বলছি, এই array এর initial ভ্যালু হবে infinity. এর পর যদি তোমার গ্রাফে i হতে j ওয়ে মাঝে কোন edge থাকে তাহলে তার cost হবে $w[i][j]$ এর মান। যদি একাধিক edge থাকে তাহলে সর্বনিম্ন টা নিবে। যদি গ্রাফ টা undirected হয় তাহলে একই সাথে $w[j][i]$ তেও সেই মান দিয়ে দিবে।

এখন প্রশ্ন হল এর complexity কত? খুবই সহজ $O(n^3)$.

আরও একটি প্রশ্ন হল, ঋণাত্মক edge cost এ floyd warshall কি কাজ করবে? হ্যা করবে। শুধু তাই না, গ্রাফে কোন কোন নোড দিয়ে negative cycle যায় তাও বের করা যাবে। তুমি শুরুতে সকল $w[i][i]$ এ 0 নিবে। এর পর floyd warshall চালানর পর যদি দেখ যে কোন একটি $w[i][i]$ এ negative মান এর মানে হল ঐ নোড দিয়ে একটি negative cycle গিয়েছে।

[১]যাক আমার দুর্বল স্মৃতিশক্তি আমার সাথে দুষ্টামি করে নাই! আমি mincost maxflow তে negative cost এর edge এর জন্য dijkstra করেছি বহুবার।

৮.৬ Dijkstra, BellmanFord, Floyd Warshall কেন সঠিক?

Dijkstra কেন সঠিক এটা আসলে ব্যখ্যা করার কিছু নেই। তুমি প্রতিবার সবচেয়ে কম cost এ যাওয়া যায় এমন vertex কে visited করছ আর যেহেতু তোমার edge cost অঋণাত্মক সেহেতু আগের visited কোন নোডে আসলে আরও কম খরচে তুমি যেতে পারবে না।

BellmanFord এ ভিতরের লুপ এ কি করছি তাতো বুঝা যায়, যেটা বুঝা যায় না সেটা হল কেন সেই কাজ n বার করলেই আমরা shortest পাথ পাবো! খেয়াল কর, তুমি যদি s হতে shortest path বের করতে চাও সব জায়গার তাহলে প্রতিটি জায়গায় তুমি n এর থেকে কম edge দিয়ে পৌছাতে পারবে। এখন ভিতরের লুপ দিয়ে কিন্তু আমরা এই কাজ টাই করছি। যদি মনে করে থাকো একবার চালালেই তো সব বের হয়ে যাওয়া উচিত। না, এখানে কিন্তু edge এর order টা খুব important. যদি কেও চায় তাহলে সে edge এর order এমন ভাবে দিতে পারে যে একবার লুপ ঘুরলেই সব জায়গার shortest path বের হয়ে যাবে, আবার কেউ যদি চায় তাহলে n বারই ঘুরাতে পারবে। [১]

Floyd warshall কেন সঠিক এটা বুঝা একটু ঝামেলা। এর জন্য যেটা বুঝতে হবে সেটা হল k এর লুপ টা বাইরে কেন[২]? এখানের k এর লুপকে Bellman Ford এর বাইরের লুপ এর মত ভাবলে হবে না। ভিতরের দুইটি লুপ n বার ঘুরান এর উদ্দেশ্য না, এর উদ্দেশ্য হল i হতে j তে যাবার সময় যদি k দিয়ে যাওয়া হয় তাহলে সেটা ভাল হয় কিনা এটা বুঝা। আরও ভাল করে বলতে, যদি ভিতরের লুপ k বার ঘুরে এর মানে হবে, i হতে j পর্যন্ত শুধু $1, 2, \ldots k$ দিয়ে গেলে সবচেয়ে কম যত cost এ যাওয়া যায় টা $w[i][j]$ তে থাকবে। সুতরাং আমরা যদি n পর্যন্ত লুপ চালাই তাহলে আসলে shortest path পেয়ে যাব।

৮.৭ Articulation vertex বা Articulation edge

একটি undirected গ্রাফ এ যদি কোন নোড কে মুছে ফেললে গ্রাফটা disconnected হয়ে যায় তাহলে তাকে articulation vertex বলে। একই ভাবে যদি কোন edge কে মুছে ফেললে গ্রাফটা disconnected হয়ে যায় তাহলে তাকে articulation edge বা articulation bridge বলে। DFS ব্যবহার করে খুব সহজেই articulation vertex বা edge বের করে ফেলা যায়। DFS এর একটা powerful প্রয়োগ হল এটি। এটা করার জন্য আমাদের কয়েকটি জিনিসের সাথে পরিচয় হতে হবেঃ dfsStartTime, dfsEndTime এবং low. Articulation vertex বা edge বের করতে এদের সবগুলিই যে দরকার তা নয়, কিন্তু এদের ব্যবহার করে আমরা বেশ জটিল জটিল প্রবলেম সমাধান করে ফেলতে পারি। বিশেষ করে Informatics Olympia লেভেলে এধরনের অনেক প্রবলেম দেখা যায়। যতদূর মনে পরে 2006 সালের IOI এ এরকম একটি প্রবলেম ছিল। যাই হোক, dfsStartTime ও dfsEndTime খুবই সহজ জিনিস। তুমি dfsTime বলে একটা variable রাখবে যার initial মান হবে 0. এর পর তোমরা dfs করার সময় যখনি কোন একটা নতুন unvisited নোডে আসবে তখনই dfsStartTime এ dfsTime এর বর্তমান সময় নোট করবে আর কোন নোডের সকল child এর visit শেষ হয়ে গেলে সেই time টা dfsEndTime এ মার্ক করে রাখবে। আর প্রতিবার নতুন vertex কে visit করার সময় dfsTime কে এক করে বাড়াবে। এতো গেল dfsStartTime আর dfsEndTime. low একটু জটিল জিনিস। আমরা তো জানি dfs পুরো গ্রাফে একটা tree এর মত করে আগায়। মানে কোন নোডের যদি unvisited adjacent vertex থাকে তাহলে আমরা সেটা visit করি (নিচে নামি) আর যদি কোন unvisited adjacent vertex না থাকে তাহলে ফেরত যাই (parent এ ফেরত যাই)। যদি সেই নোড থেকে কোন visited নোড এ যাওয়া যায় তা অবশ্যই এর ancestor হবে অর্থাৎ ঐ নোড থেকে root এর path এর মাঝেই থাকবে (চিন্তা করে দেখ) অথবা তার subtree তে থাকবে। যদি কোন নোড থেকে তার ancestor এ যাওয়া যায় তাহলে সেই edge কে আমরা back edge বলি। low[u] হল u নোড বা u এর subtree তে থাকা নোডগুলি থেকে সবচেয়ে উপরে (root এর কাছের) যেই নোড এ যাওয়া যায় তার index.

[১] আসলে $n - 1$ বার ঘুরালেই হয়। কোন এক ঐতিহাসিক কারনে আমরা সবসময় n বার বলে থাকি।

[২] আগে আমি বেশ কয়েকবার এই ভুল করতাম, প্রায় সময় k এর লুপ ভিতরে দিয়ে থাকতাম ভাবতাম একই তো কথা! কিন্তু এক কথা না।

low[u] বের করার জন্য যা করতে হবে তা হলঃ ধরা যাক v হল u এর adjacent কোন vertex. যদি v ইতোমধ্যেই visited হয়ে যায় তাহলে হয় v হবে u এর parent বা parent না (ancestor)। যদি ancestor হয় তাহলে তার dfsStartTime দিয়ে low[u] কে update করতে হবে। আর যদি parent হয় তাহলে একটু সতর্ক হতে হবে। parent থেকে যেই edge দিয়ে আমরা u তে এসেছি যদি সেই edge হয় এটা তাহলে আমরা কোন কিছু করব না, আর যদি এটা ভিন্ন edge হয় তাহলে আগের মত update করতে হবে। যদি আমাদের গ্রাফ multi edge গ্রাফ না হয় তাহলে আমাদের এটা নিয়ে ভাবার কিছু নেই। এখন যদি আমাদের v নোডটা আগে থেকে visited না হয় তাহলে তার dfs করতে হবে recursively এবং low[v] দিয়ে low[u] কে update করতে হবে। এখানে update করা মানে হল minimum value টা বের করা। এই low[] ভ্যালু কিন্তু কোন একটি নোড এর dfsStartTime, আমাদের এই time যত কম হবে ততই সেই নোড root এর কাছাকাছি হবে।

এখন আমাদের সব দরকারি value বের করা হয়ে গেছে। এই value গুলো দেখে আমরা বলতে পারব কোন কোনটা articulation vertex আর কোন কোনটা articulation edge। নোড u, articulation vertex হবে ১. যদি এটি root হয় এবং এর একাধিক child থাকে অথবা ২. এটি root না হয় এবং $low[u] \geq dfsStartTime[u]$ হয়। এখন আশা করি তোমরা একটু চিন্তা করলেই বুঝবে কেন এই দুইটি condition এর একটি সত্য হলে সেই নোডটি articulation vertex হবে বা কোন নোড articulation vertex হলে কেন এই দুই condition এর একটি সত্য হবে।

যদি উপরের condition দুইটা বুঝে থাকো তাহলে আশা করি $u-v$ edge কখন articulation edge হবে তাও বের করে ফেলতে পারবে। condition টা হল, u যদি v এর parent হয় তাহলে $low[v] > dfsStartTime[u]$ হতে হবে। খেয়াল কর, কোন back edge কিন্তু কখনই articulation edge হতে পারবে না। সুতরাং আমাদের শুধু dfs tree এর edge গুলি check করলেই হবে।

৮.৮ Euler path এবং euler cycle

আমরা প্রথমে শুধু undirected graph নিয়ে ভাবব। Euler path [১] হল কোন একটি গ্রাফে যদি একটি vertex থেকে যাত্রা শুরু করে প্রতিটি edge, exactly একবার করে ঘুরে কোন একটি vertex এ যদি যাত্রা শেষ করা যায় তাহলে তাকে euler path বলে। আর যদি শুরু ও শেষের vertex একই হয় তাহলে তাকে euler cycle বা euler circuit বলে। কোন একটি গ্রাফে euler path বা cycle আছে কিনা তা বের করা খুবই সহজ। প্রথম শর্ত হল গ্রাফ কে connected হতে হবে। এখন যদি সব গুলি নোড এর degree জোড় হয় তাহলে গ্রাফ এ euler cycle আছে (euler cycle থাকা মানে কিন্তু euler path ও থাকা, কিন্তু উল্টোটা সত্য নয়)। আর যদি এই গ্রাফ এর শুধুমাত্র দুইটি নোড odd degree ওয়ালা হয় তাহলেও গ্রাফটাতে euler path থাকবে তবে সেক্ষেত্রে আমাদের অবশ্যই ঐ দুইটি নোড এর কোন একটি থেকে যাত্রা শুরু করতে হবে। এরকম হবার কারন হল, শুরু আর শেষের নোড বাদে বাকি সব নোড এর ক্ষেত্রে আমরা কিন্তু একবার ঢুকলে বের হতে হয় সুতরাং আমাদের edge গুলি জোড়ায় জোড়ায় থাকে বা বলতে পারি মাঝের সব vertex গুলির degree হবে জোড়। এখন যদি cycle হয় তাহলে যেখান থেকে শুরু করেছি সেখানেই শেষ করেছি সুতরাং সেই নোড এর degree ও জোড় হবে। কিন্তু যদি এটা cycle না হয়ে path হয় তাহলে দেখ শুরু আর শেষ vertex আলাদা এবং তাদের degree হবে বিজোড়। এটা তো আমরা প্রমান করলাম যে euler path বা cyle হলে এরকম property থাকবে। কিন্তু এরকম property থাকলেই যে euler path বা cycle হবে তা কিন্তু প্রমান করি নাই। সেটা প্রমান করাও কিন্তু খুব একটা কঠিন না। তোমরা induction ব্যবহার করে খুব সহজেই প্রমান করতে পারবে।

এখন কোড করে কেমনে আমরা euler path বা cycle বের করতে পারব? এটাও খুব সহজ, dfs এর মত তুমি কোন একটি vertex থেকে যাত্রা শুরু কর। তবে এখানে আমাদের vertex এর জন্য কোন visited থাকবে না, থাকবে edge এর জন্য। খেয়াল রেখো কোন একটা edge কিন্তু দুই দিকের কোন একদিক থেকেই visit করা যায়। এখন কোন একটি vertex এ আমরা দাঁড়িয়ে দেখব যে এর থেকে বের হওয়া কোন কোন edge এখনও visited হয় নাই, যদি এমন কোন edge বাকি

[১] Euler এর উচ্চারন অয়লার

থাকে তাহলে সেটা দিয়ে বের হয়ে যাব এবং আগের মতই ঘুরতে থাকব। আর যদি দেখি এই vertex এর সাথে লাগান সব edge ই visited হয়ে গেছে তাহলে এই vertex কে print করে দেব। খেয়াল কর এই প্রসেস এ কিন্তু একটা vertex কিন্তু অনেক বার visit হতে পারে।

এবার দেখা যাক directed গ্রাফ এ কেমনে আমরা euler path বা cycle বের করতে পারি। আসলে আমি এই paragraph লিখার আগে এই জিনিস এর সম্মুখিন হই নাই বা হলেও মনে নেই। সুতরাং আমি একটু internet ঘেঁটে ঘুটে যা দেখলাম তাহল directed গ্রাফ এর euler path বা cycle বের করা প্রায় পুরোপুরি undirected গ্রাফ এর মত। আমরা কোন একটা নোড থেকে শুরু করব, এর outgoing কোন edge দিয়ে বের হব যতক্ষণ কোন না কোন outgoing edge বাকি থাকে। যখনি শেষ হয়ে যাবে আমরা print করে দিব এবং আগের নোড এ ফিরে যাব, ঠিক আগের dfs এর মত। আশা করি বুঝতে পারছ যে আমাদের euler path বা cycle এর ঠিক উলটো order প্রিন্ট হয়েছে! আরেকটা জিনিস, তাহল path print করার আগে প্রতিটি নোড এর outdegree আর indegree একটু দেখে নিতে হবে। প্রতিটি নোড এর indegree আর outdegree সমান হতে হবে কেবল একটি নোড এর indegree, outdegree হতে এক বেশি হতে পারবে এবং একটি নোড এর outdegree, indegree এর থেকে এক বেশি হতে পারবে। তাহলে তারা যথাক্রমে path এর শেষ ও শুরু হবে। কিন্তু সবার যদি in আর out degree সমান হয় তাহলে যেকোনো নোড থেকে শুরু করে সেখানে ফেরত আশা যাবে। তবে আগের মতই connected ব্যপার টা একবার দেখে নিতে হবে। খেয়াল রাখতে হবে যেন, শুরুর নোড থেকে যেন সব জায়গায় যাওয়া যায় আর শেষের নোড এ যেন সব জায়গা থেকে যাওয়া যায়।

৮.৯ টপলজিকাল সর্ট (Topological sort)

একটি directed গ্রাফে নোডগুলিকে এমন ভাবে অর্ডার করতে হবে যেন, যদি u থেকে v তে কোন directed edge থাকে তাহলে সর্টেড অ্যারেতে u, v এর আগে থাকবে। এটা তখনি সম্ভব যখন ঐ গ্রাফে কোন directed cycle থাকবে না। cycle থাকলে তো ঐ cycle এর নোড গুলিকে তুমি কোন ভাবেই অমন অর্ডার দিতে পারবে না তাই না? আমরা এই অ্যালগোরিদম ব্যবহার করে কোন directed গ্রাফে cycle আছে কিনা তাও বের করে ফেলতে পারব।

আমরা দুই ভাবে topological sort করতে পারি। একটি হল BFS দিয়ে আরেকটি DFS দিয়ে। আমার কাছে BFS দিয়ে তুলনামুলক সহজ মনে হয়, এছাড়াও এখানে stack overflow নিয়ে মাথা ঘামাতে হয় না। কিন্তু DFS একটু তুলনামুলক ভাবে ছোট হয়ে থাকে ১-২ লাইন। প্রথমে দেখা যাক আমরা BFS দিয়ে কেমনে সমাধান করতে পারি।

প্রথমে আমাদের একটি indegree এর অ্যারে লাগবে যেখানে প্রতিটি নোড এর indegree লিখা থাকবে। এবার একটি queue তে সেসব নোড রাখতে হবে যাদের indegree শূন্য। এবার queue থেকে একে একে নোড তুলার পালা। একটা করে নোড তুলবো আর তার থেকে যেসব edge বের হয়ে গেছে তাদের দেখে দেখে অন্য প্রান্তের নোড এর indegree কমায়ে দিব। যদি অন্য প্রান্তের indegree শূন্য হয়ে যায় তাহলে তাকে queue তে ঢুকিয়ে দিব। এভাবে যতক্ষণ না queue ফাঁকা হয়ে যায় ততক্ষণ চলতে থাকবে। queue তে নোড যেই order এ ঢুকেছে সেটাই কিন্তু topological sort এর অর্ডার। তবে একটা জিনিস, যদি queue ফাঁকা হয়ে যাবার পরেও দেখ যে কোন নোড এর indegree এখনও শূন্য হয় নাই তার মানে গ্রাফটায় cycle আছে। এটি কিন্তু খুবই intuitive একটা অ্যালগোরিদম। কেন কাজ করছে তা খুব সহজেই বুঝা যায়।

এবার আশা যাক DFS দিয়ে কেমনে এটা সমাধান করা যায়। ধরা যাক T হল আমাদের রেজাল্ট এর একটি অ্যারে, আর visited আরেকটি অ্যারে যার initial মান 0. এখন আমরা একে একে প্রতিটি নোড দেখব আর যদি সেই নোড এর visited এর মান এখনও 2 না হয়ে থাকে তাহলে তার DFS কল করব। DFS এর প্রথমেই আমরা যা করব তাহল এর visited এর মান 1 করে দেব। এর পর এখান থেকে যেই যেই directed edge বের হয়েছে তাদের দেখব। যদি অন্য প্রান্তের নোড visited এর মান 1 হয়ে থাকে এর মানে হল আমরা একটা cycle পেয়ে গেছি সুতরাং নোড গুলির কোন topological order নেই। যদি visited এর মান 2 হয় তাহলে আমাদের করার কিছু নেই। আর যদি 0 হয় তাহলে আমরা ঐ নোড এর জন্য DFS কল করব। এভাবে প্রতিটি নোড এর জন্য প্রসেসিং শেষ হলে আমরা সেই নোড এর visited কে 2 করে দেব এবং তাকে T তে ঢুকিয়ে দেব এবং DFS থেকে return

করব। এভাবে সব নোড এর জন্য DFS শেষ হয়ে গেলে আমরা T তে topological sort উলটো অর্ডারে পাবো।

৮.১০ Strongly Connected Component (SCC)

একটি directed গ্রাফে যখন একটি নোড u থেকে v তে যাওয়া যায় এবং একই সাথে v থেকে u নোড এ যাওয়া যায় তখন আমরা বলব ঐ দুইটি নোড একই SCC তে আছে। খেয়াল করে দেখ, যদি u আর v একই SCC তে থাকে আর v আর w ও একই SCC তে থাকে তাহলে u আর w ও একই SCC তে থাকবে কারন তুমি w থেকে v তে যেতে পারো আর v থেকে u তে যেতে পারো, একই ভাবে u থেকেও v হয়ে তুমি w তে যেতে পারো। সুতরাং u আর w একই SCC তে। একটু চিন্তা করলে বুঝবে যে এর মানে দাঁড়ায় তুমি পুরো গ্রাফকে আসলে অনেকগুলি SCC তে ভাংতে পারবে যেন কোন একটি নোড কোন একটি এবং কেবল মাত্র একটি SCC এর অংশ। যেমন ধর, যদি আমাদের edge গুলি হয়ঃ $(u, v), (v, w), (w, u)$ তাহলে এখানে কেবল মাত্র একটি SCC: u, v, w. আবার ধরা যাক আমাদের edge গুলি হলঃ $(u, v), (w, v)$ তাহলে কিন্তু তিনটি SCC: u, v, w. আবার $(u, v), (v, u), (w, u), (w, v)$ হলে দুইটি SCC: u, v, w.

এখন আমরা SCC বের করার অ্যালগোরিদম শিখব। এর জন্য দুইটি বহুল প্রচলিত অ্যালগোরিদম আছে। একটি হল Kosaraju's algoritm (2-dfs অ্যালগোরিদম) আরেকটা হল Tarjan's algorithm. আমি আসলে শুধু প্রথমটাই জানি। এটা করা বা বলা সহজ, কিন্তু আমার কাছে মনে রাখা বেশ কষ্টকর মনে হয় এবং বুঝাও একটু কষ্টকর মনে হয়। প্রথমে একটি visited এর অ্যারে নাও এবং সব নোড কে unvisited করে দাও। এখন একে একে নোড গুলি চেক কর, যদি কোন unvisited নোড পাও তাহলে তার জন্য dfs কল কর। dfs এর ভিতরে যা করবা তাহল, ঐ নোড কে visited করে দিবা আর ঐ নোড থেকে যদি কোন unvisited নোড এ যাওয়া যায় তাহলে তার dfs কল করবা। adjacent সব নোড visited হয়ে গেলে dfs থেকে return করার আগে একটা লিস্টের শেষে (ধরা যাক তার নাম L) ঐ নোডকে পুশ করে যাবা। তাহলে সব নোড visited হয়ে গেলে কিন্তু আমাদের ঐ L লিস্টে সব নোড থাকবে। এবার যেটা করতে হবে তাহল ঐ গ্রাফের সব edge কে উলটো করে দিতে হবে (আসলে তুমি শুরুতেই দুইটি adjacency list বানায়ে নিবা একটা ঠিক দিকে আরেকটা উলটো দিকে)। তুমি যেই নোড এর লিস্ট L বানায়ে ছিলা ওটাকেও উলটো করতে হবে। এবার আমরা আরও একটা DFS করব। আমাদের আবার সব নোড কে unvisited করে দিতে হবে। এখন আমরা L এর সামনের দিক থেকে একটা একটা করে নোড নিব এবং সে যদি visited না হয়ে থাকে তার জন্য dfs কল করব। খেয়াল রেখো, এবার dfs এ কিন্তু আমরা উলটো গ্রাফ ব্যবহার করছি। এই dfs এর সময় যেই যেই নোড visited হবে তারা একটা SCC [১]. এভাবে আমরা সব SCC পেয়ে যাব।

এই বই লিখতে গিয়ে আমি Tarjjan এর SCC অ্যালগোরিদম দেখলাম। খুব একটা কঠিন না। এই অ্যালগোরিদমটা কিছুটা Articulation Bridge বা Vertex বের করার মত। তবে মনে রাখতে হবে এবার আমরা directed গ্রাফ নিয়ে কাজ করছি। সবসময়ের মত প্রতিটি নোড visited না হওয়া পর্যন্ত আমরা প্রতিবার একটি একটি করে unvisited নোড নিয়ে তার জন্য dfs কল করব। dfs এর শুরুতে আমরা তার startTime আর low কে বর্তমান time এর মান দারা আপডেট করে time এর মান এক বাড়ায়ে দেব। সেই সাথে এই নোডকে একটা stack এ পুশ করব। এবার এই নোড থেকে যেখানে যেখানে যাওয়া যায় তাদের দেখার পালা। যদি অপর নোডটি আগে থেকেই visited হয় তাহলে আমরা বর্তমান নোড এর low কে ওপর নোড এর startTime দিয়ে আপডেট করব (minimum নিব) আর যদি unvisited হয় তাহলে তার জন্য dfs কল করব। এভাবে সব neighbor এর জন্য প্রসেসিং শেষ হলে আমাদের দেখতে হবে যে তার low এর মান startTime এর মানের সমান কিনা, অর্থাৎ আমরা তার neighbor দিয়ে তার থেকেও আগের কোন নোড এ যেতে পারি কিনা। যদি যেতে পারি (low এর মান startTime এর থেকেও কম) তাহলে এখানে আর কিছু করার নেই। আর যদি সমান হয়, তাহলে আমাদের stack থেকে নোড তুলতেই থাকব যতক্ষণ না আমাদের বর্তমান নোড পাই। এই সব নোডগুলি হল একটি SCC.

[১] তোমরা চাইলে dfs এর ফাংশন কে একটি number দিয়ে দিতে পারো যে এটা হল SCC এর নাম্বার এটা দিয়ে visited কে মার্ক করতে, বা চাইলে একটা লিস্ট ও parameter এ দিয়ে দিতে পারো যেন visited নোড গুলি ঐ লিস্ট এ রাখা হয়।

৮.১১ 2-satisfiability (2-sat)

(a or b) and (!b or c) and (!a or !c) এই equation এর একটি সমাধান হতে পারেঃ a = 1, b = 0, c = 0. কিন্তু অনেক সময় কোন সমাধান নাও থাকতে পারে, যেমনঃ (a or b) and (a or !b) and (!a or b) and (!a or !b). Formally বলতে গেলে a, b, c এগুলোকে variable বলা হয় আর দুইটি করে variable বা তাদের not নিয়ে or করে যে এক একটি pair বানানো হয় তাদের clause বলে। অনেক গুলি clause এর and করে বড় equation বা statement তৈরি করা হয়। আমাদের লক্ষ্য হল, variable গুলিতে এমন মান assign করা যায় কিনা যেন আমাদের statement টা সঠিক হয়। যেহেতু প্রত্যেকটা clause এ দুইটি করে term থাকে সেজন্য একে 2-sat প্রবলেম বলা হয়। তোমাদের জানার জন্য বলে রাখি 3-sat প্রবলেম হল NP আর দুনিয়ার মোটামোটি অনেক প্রবলেম এদিক ওদিক করে 3-sat এ কনভার্ট করা যায়।

যদি আমাদের statement এ n টি variable থাকে তাহলে আমাদেরকে একটি 2n নোড এর directed গ্রাফ বানাতে হবে। x দিয়ে যদি একটি variable থাকে তাহলে x এর জন্য একটা নোড আরেকটা !x এর জন্য। এখন ধরা যাক (!x or y) হল একটি clause, তাহলে একে আমরা দুই ভাবে লিখতে পারিঃ $x \rightarrow y$ আর $!y \rightarrow !x$. [১] বা (!x or y) কে এভাবে তুমি interpret করতে পারো যদি !x মিথ্যা হয় তাহলে y সত্য হবে, অথবা যদি y মিথ্যা হয় তাহলে !x সত্য হবে। আমরা একে গ্রাফে directed edge দিয়ে প্রকাশ করব। (!x or y) এর ক্ষেত্রে আমাদের edge হবে x থেকে y এর দিকে আর !y থেকে !x এর দিকে। অন্যভাবে বলতেঃ যদি x সত্য হয় তাহলে y ও সত্য হবে, আর যদি !y সত্য হয় তাহলে !x ও সত্য হবে। এখন আমরা এভাবে প্রত্যেকটা clause এর জন্য দুইটি করে edge দিব। সব edge আঁকা শেষে আমাদের দেখতে হবে যে, x আর !x (এখানে x হল যেকোনো variable, অর্থাৎ তোমাকে n টি variable এর জন্য চেক করে দেখতে হবে।) এর জন্য x থেকে !x এ আর !x থেকে x এ দুইদিকেই path আছে কিনা। যদি থাকে তাহলে আমাদের 2-sat সমাধান করা যাবে না। আর যদি এমন কোন variable খুজে পাওয়া না যায় তাহলে সমাধান করা যাবে। আমরা দুইটি নোড থেকে একে অপরের দিকে যাওয়া যায় কিনা কেমনে সহজে বের করতে পারি? SCC! যদি আমাদের গ্রাফকে SCC তে ভাঙ্গার পরে দেখি x ও !x একই component এ পরে তাহলে বুঝবো একে ওপরের দিকে যাওয়া যায়, আর না হলে যাবে না।

এখন কথা হল একই component এ পরলে সমস্যা কই? খেয়াল কর, যদি কখনও x থেকে y এ যাওয়া যায় এর মানে দাঁড়াবে, x সত্য হলে y সত্য হবে। তাহলে যদি x থেকে !x এ পাথ থাকে তার মানে দাঁড়াবে x সত্য হলে !x সত্য হবে। অর্থাৎ x কে অবশ্যই মিথ্যা হতে হবে। এটা সমস্যা না। সমস্যা হবে যদি একই সাথে !x থেকেও x এ পাথ থাকে। তাহলে x কে অবশ্যই সত্য হতে হবে। x যেহেতু একই সাথে সত্য আর মিথ্যা হতে পারবে না সেহেতু আমাদের 2-sat এরও সমাধান থাকবে না।

কথা হল আমরা বের করলাম কেমনে 2-sat সমাধানযোগ্য কিনা তা বের করা যায়। সমাধান কেমনে বের করা যায়? এর উপায় হল, তুমি scc এর প্রত্যেক component কে contract (সংকুচিত) করে একটি নোড বানাও। এই পরিবর্তিত গ্রাফ অবশ্যই একটি DAG হবে (DAG = Directed Acyclic Graph) অর্থাৎ এই directed গ্রাফে কোন cycle নাই। যেহেতু কোন cycle নাই তাই এর topological order আছে। আমাদের যা করতে হবে, এই অর্ডার এর শেষ থেকে আসতে হবে আর প্রত্যেক component কে true দেবার চেষ্টা করতে হবে। যদি দেখ তোমার এই component এ এমন একটা variable আছে যার মান আগে থেকেই assing করা হয়ে গেছে আর তোমার এই এখন true করতে চাওয়া মান এর সাথে conflict করছে তাহলে false দিবা। তুমি চাইলে চিন্তা করে দেখতে পারো বা প্রমানও করতে পারো কেন এই greedy প্রসেসটা ঠিক ভাবে মান assign করছে।

[১] যারা implication sign এর মানে জানো না তাদের জন্য বলি, $a \rightarrow b$ কেবল মাত্র (a = 1, b = 0) এর জন্য false এছাড়া সবসময় true. অর্থাৎ একে এভাবে ভাবতে পারঃ a সত্য হলে b ও সত্য হবে, a মিথ্যা হলে b যা খুশি তাই হোক যায় আসে না।