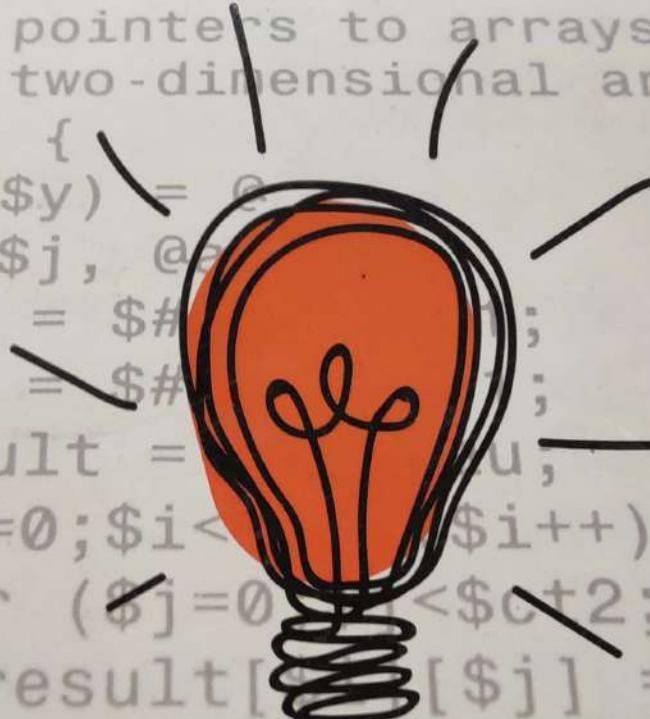


প্রোগ্রামিং কনটেক্ট

ডেটা স্ট্রাকচার ও অ্যালগরিদম

মো: মাহবুবুল হাসান

```
#-----  
# Returns a cross product of two vectors.  
# Takes two pointers to arrays as input  
# Returns a two-dimensional array.  
sub xProduct {  
    my ($x,$y) = @_;  
    my $i, $j, @e;  
    my $ct1 = $#{$x}; # items in $x  
    my $ct2 = $#{$y}; # items in $y  
    my $result = [ ];  
    for ($i=0;$i<=$ct1;$i++) {  
        for ($j=0;$j<=$ct2;$j++) {  
            $$result[$i][$j] = $$x[$i] *  
                # print "$i, $j, $$result[$i][$j]  
    }  
}
```



প্রোগ্রামিং কনটেন্স
ডেটা স্টোকচার ও অ্যালগরিদম

মোঃ মাহবুবুল হাসান

জুন, ২০১৬



দ্বিমিক প্রকাশনী

পাইকারী বিক্রেতা
মানিক মাইক্রো
নামাজ ঘর গলি, নীলক্ষেত্র, ঢাকা
০১৭৩৫৭৮২৯০৮

ভূমিকা

মোঃ মাহবুবুল হাসান শান্ত এর বইয়ের ভূমিকা লেখার ভার যখন আমাকে দেওয়া হলো তখন আমি বেশ অবাক হই, কিন্তু বইয়ের কনটেন্ট এর ব্যাপ্তি দেখে আরো অনেক বেশি অবাক হই। শান্তের বইয়ে UVa আর্কাইভ এর অনেক প্রবলেম ব্যবহার হয়েছে দেখে ভালো লাগলো, কারণ এটা হয়তো UVa সাইটের জনপ্রিয়তা বৃদ্ধিতে বড় ভূমিকা রাখবে। সেজন্য বইটির ইংরেজি অনুবাদেরও অপেক্ষায় থাকলাম।

তরুণ প্রজন্মের মধ্যে আমার দেখা সর্বশ্রেষ্ঠ শিক্ষক মনে হয় মনিরুল হাসান (তমাল)। কিন্তু আরেকটু তরুণ প্রজন্মের মধ্যে যদি খুঁজে দেখি তাহলে দুটো নামই মাথায়ে আসে- মোহাম্মদ মাহমুদুর রহমান এবং মোঃ মাহবুবুল হাসান শান্ত। মোটামুটি ভালো শিক্ষক হলেই যে সবসময় ভালো লেখক হয়না সেটা নিজেকে দিয়েই বুঝি কিন্তু শান্ত তার বুকানোর ক্ষমতাকে বই এর মধ্যে আনতে পেরেছে ভালোভাবেই তাই এই বইটি তরুণ প্রজন্মের জন্য অনেক উপকারী হবে সন্দেহ নেই। আজকে কেবলই মনে হচ্ছে কেন আমার বয়স আরো বিশ বছর কম হলো না, তাহলে এই বই দেখে আরো ভালোভাবে সবকিছু অনেক কম বয়সে শিখে ফেলতে পারতাম।

শান্তের প্রোগ্রামিং কনটেন্ট ক্যারিয়ার অনেক দীর্ঘ। তবে তাকে প্রথম ভালো ভাবে চিনি যখন "Dhaka Regional ২০০৫" এ শান্ত ও নাফি এর IOI গামী দল বাংলাদেশের সব বিশ্ববিদ্যালয়কে পেছনে ফেলে দ্বিতীয় স্থান দখল করে। "Lattice triangle" গণনার একটি সমস্যা সমাধান করে তারা সবাইকে তাক লাগিয়ে দিয়েছিল। সৌভাগ্যক্রমে সেই প্রবলেমের স্রষ্টা ছিলাম আমি। ভিসা জটিলতার কারণে তাদের IOI এ অংশগ্রহণ করা হয় নাই, নাহলে বাংলাদেশের IOI পদক অনেক আগেই আসতে পারত। শান্ত সন্তুষ্ট এখনো নানান কনটেন্টে অংশগ্রহণ করে, তাই তার চেয়ে দীর্ঘ কনটেন্ট ক্যারিয়ার খুব কম লোকেরই আছে। তার উপর শান্তের রয়েছে সমস্যার সমাধান করার সীমাহীন উৎসাহ ও ধৈর্য। জনশ্রুতি রয়েছে যে শান্ত তার বিয়ের দিনও বিভিন্ন সমস্যার সমাধান করেছে। কাজেই এত দীর্ঘ ক্যারিয়ার ও সময়ে শান্ত কী পরিমাণ সমস্যা সমাধান করেছে তা আন্দাজ করাও অনেকের পক্ষে কঠিন হবে। এই বইয়ে তাই নানা ধরনের সমস্যা সমাধান এর কথা উঠে এসেছে। বাংলা ভাষায় এমন বই আগে প্রকাশিত হয়নি এমনকি ইংরেজিতে অনুদিত হলেও এই বই যথেষ্ট সমাদৃত হবে বলে আমার বিশ্বাস।

সাধারণত দেশের বাইরে গিয়ে লোকজন প্রোগ্রামিং কনটেস্ট এবং প্রবলেমসেটিং কে ভূলে যায়, কিন্তু শান্ত এই দিক থেকে ব্যতিক্রম। এই বইয়ের পাঠক সংখ্যা কয়েক মিলিয়ন হলেই সেই ব্যতিক্রম অচেষ্টা সফল হবে। সেইসব মিলিয়ন প্রোগ্রামার বাংলাদেশকে অনেক সম্মানিত করবে। মনে রাখতে হবে যে পোশাক ও শ্রমিক রফতানি করে বৈদেশিক মুদ্রা অর্জন করা সম্ভব হলেও সম্মানের জন্য প্রয়োজন একটু সৃজনশীল কিছু। কৃতিম বুদ্ধিমত্তা (AI) ও রোবটের উপরের যুগে, প্রোগ্রামিং ছাড়া অন্য কিছুতে মানুষের প্রয়োজন থাকবে কিনা সেটা ও ভাবা দরকার :)।

শাহরিয়ার মজুর,

সভাপতি, বাংলাদেশ অ্যাসোসিয়েশন অব প্রবলেমসেটারস
জাঞ্জিং ডিরেক্টর, এসিএম আইসিপিসি ঢাকা রিজিওনাল ২০০৪-২০১৫
বিচারক, এসিএম আইসিপিসি ওয়ার্ল্ড ফাইনাল ২০০৩-২০১৬
এছাড়াও ২০০০ সাল থেকে ভ্যালাদলিদ অনলাইন জাজ (Valladolid Online Judge) এর
সাথে জড়িত

সূচিপত্র

১	প্রোগ্রামিং প্রতিযোগিতায় হাতেখড়ি	১১
১.১	শুরুর কথা	১১
১.২	প্রোগ্রামিং প্রতিযোগিতা কী?	১১
১.৩	কেন করব?	১৩
১.৪	কীভাবে শুরু করব?	১৩
১.৫	কী কী জানতে হবে?	১৮
২	C বালাই	১৯
২.১	একটি ছোট প্রোগ্রাম এবং ইনপুট আউটপুট	১৯
২.২	ডেটা টাইপ এবং math.h হেডার ফাইল	২১
২.৩	if - else if - else	২৪
২.৪	লুপ (Loop)	২৮
২.৫	অ্যারে (Array) ও স্ট্রিং (String)	৩৩
২.৬	টাইম কমপ্লেক্সিটি (Time Complexity) এবং মেমোরী কমপ্লেক্সিটি (Memory Complexity)	৪০
২.৭	ফাংশন এবং রিকার্শন (Recursion)	৪২
২.৮	ফাইল (File) ও স্ট্রাকচার (Structure)	৪৫
২.৯	বিটওয়াইজ অপারেশন (bitwise operation)	৪৬
৩	গণিত	৪৯
৩.১	সংখ্যাতত্ত্ব (Number Theory)	৪৯
৩.১.১	মৌলিক সংখ্যা (Prime Number)	৪৯
৩.১.২	একটি সংখ্যার গুণনীয়কসমূহ	৫৫
৩.১.৩	গ.সা.গ. (GCD) ও ল.সা.গ. (LCM)	৫৭
৩.১.৪	অয়লার এর টোশেন্ট ফাংশন (Euler's Totient Function - ϕ)	৫৮
৩.১.৫	BigMod	৬১
৩.১.৬	মডুলার ইনভার্স (Modular Inverse)	৬৪

3.1.7	Extended GCD	68
3.2	কম্বিনেটরিক্স (Combinatorics)	68
3.2.1	ফ্যাক্টোরিয়ালের পেছনের 0	68
3.2.2	ফ্যাক্টোরিয়ালের অঙ্ক (Digit) সংখ্যা	69
3.2.3	সমাবেশ (Combination): $\binom{n}{r}$	69
3.2.4	কিছু বিশেষ সংখ্যা	69
3.2.5	ফিবোনাচি সংখ্যা (Fibonacci Number)	72
3.2.6	ইনক্লুশন এক্সক্লুশন নীতি (Inclusion Exclusion Principle)	73
3.3	সন্তাব্যতা (Probability) ও এক্সপেক্টেশন (Expectation)	78
3.3.1	সন্তাব্যতা (Probability)	78
3.3.2	এক্সপেক্টেশন (Expectation)	78
3.4	বিবিধ	79
3.4.1	ভিত্তি পরিবর্তন (Base Conversion)	79
3.4.2	বিগ ইন্টিজার (Big Integer)	78
3.4.3	চক্র বা সাইকেল (Cycle) নির্ণয়ের অ্যালগরিদম	80
3.4.4	গাউসের এলিমিনেশন (Gaussian elimination)	81
3.5	প্রোগ্রামিং সমস্যা	86
3.5.1	অনুশীলনী	86
8	সর্টিং (Sorting) ও সার্চিং (Searching)	87
8.1	সর্টিং (Sorting)	87
8.1.1	ইনসার্শন সর্ট (Insertion Sort)	87
8.1.2	সিলেকশন সর্ট (Selection Sort)	89
8.1.3	বাবল সর্ট (Bubble Sort)	89
8.1.4	মার্জ সর্ট (Merge Sort)	91
8.1.5	কাউন্টিং সর্ট (Counting Sort)	93
8.1.6	STL এর sort ফাংশন	98
8.2	বাইনারি সার্চ (Binary Search)	99
8.3	টারনারি সার্চ (Ternary Search)	99
8.4	ব্যাকট্র্যাকিং (Backtracking)	100
8.4.1	সবরকম বিন্যাস বের করা (Permutation Generation)	101
8.4.2	সবরকম সমাবেশ বের করা (Combination Generation)	103
8.4.3	Eight Queen সমস্যা	106
8.4.4	Knapsack সমস্যা	109
8.5	প্রোগ্রামিং সমস্যা	110
8.5.1	অনুশীলনী	110

৫ ডেটা স্ট্রাকচার	১১১
৫.১ লিঙ্কড লিস্ট (Linked List)	১১২
৫.২ স্ট্যাক (Stack)	১১৭
৫.২.১ ০ – ১ ম্যাট্রিক্সে সব ১ ওয়ালা সবচেয়ে বড় আয়তক্ষেত্র	১১৯
৫.৩ কিউ (Queue)	১২০
৫.৪ গ্রাফ (graph) এর উপস্থাপন	১২২
৫.৫ ট্রি (Tree)	১২৩
৫.৬ বাইনারি সার্চ ট্রি (Binary Search Tree - BST)	১২৫
৫.৭ হৈপ (Heap) বা প্রায়োরিটি কিউ (Priority Queue)	১২৬
৫.৮ ডিসজয়েন্ট সেট ইউনিয়ন (Disjoint set Union)	১২৯
৫.৯ Square Root segmentation	১৩০
৫.১০ স্ট্যাটিক (Static) ডেটাতে কুয়েরি	১৩২
৫.১১ সেগমেন্ট ট্রি (Segment Tree)	১৩৩
৫.১১.১ সেগমেন্ট ট্রি তৈরী করা	১৩৪
৫.১১.২ সেগমেন্ট ট্রি আপডেট করা	১৩৫
৫.১১.৩ সেগমেন্ট ট্রি তে কুয়েরি করা	১৩৬
৫.১১.৪ Lazy without Propagation	১৩৮
৫.১১.৫ Lazy With Propagation	১৩৯
৫.১১.৬ একটি উদাহরণ	১৪২
৫.১২ বাইনারি ইনডেক্সড ট্রি (Binary Indexed Tree)	১৪৩
৫.১৩ প্রোগ্রামিং সমস্যা	১৪৮
৫.১৩.১ অনুশীলনী	১৪৮
৬ গ্রীডি টেকনিক (Greedy Technique)	১৪৫
৬.১ Fractional Knapsack	১৪৫
৬.২ মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree)	১৪৭
৬.২.১ প্রিম এর অ্যালগরিদম (Prim's Algorithm)	১৪৮
৬.২.২ ক্রুসকাল এর অ্যালগরিদম (Kruskal's Algorithm)	১৫২
৬.৩ ওয়াশিং মেশিন ও ড্রায়ার	১৫৪
৬.৪ হাফম্যান কোডিং (Huffman Coding)	১৫৫
৬.৫ প্রোগ্রামিং সমস্যা	১৫৮
৬.৫.১ অনুশীলনী	১৫৮
৭ ডায়নামিক প্রোগ্রামিং (Dynamic Programming)	১৫৯
৭.১ আবারও ফিবোনাচি	১৫৯
৭.২ কয়েন চেঞ্জ (Coin Change)	১৬২

৭.২.১	Variant 1	১৬৫
৭.২.২	Variant 2	১৬৮
৭.২.৩	Variant 3	১৬৮
৭.২.৪	Variant 4	১৬৮
৭.২.৫	Variant 5	১৬৮
৭.৩	ট্রাভেলিং সেলসম্যান সমস্যা (Travelling Salesman Problem)	১৬৯
৭.৪	দীর্ঘতম ক্রমবর্ধমান সাবসিকোয়েন্স (Longest Increasing Subsequence)	১৬৯
৭.৫	দীর্ঘতম সাধারণ সাবসিকোয়েন্স (Longest Common Subsequence)	১৭১
৭.৬	ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন (Matrix Chain Multiplication)	১৭৩
৭.৭	অপটিমাল বাইনারি সার্চ ট্রি (Optimal Binary Search Tree)	১৭৪
৭.৮	প্রোগ্রামিং সমস্যা	১৭৬
৭.৮.১	অনুশীলনী	১৭৬
৮	গ্রাফ	
৮.১	ব্রেডথ ফাস্ট সার্চ (Breadth First Search - BFS)	১৭৭
৮.২	ডেপথ ফাস্ট সার্চ (Depth First Search - DFS)	১৭৯
৮.৩	DFS ও BFS এর কিছু সমস্যা	১৮২
৮.৩.১	কম্পোনেন্ট (Component) বের করা	১৮৩
৮.৩.২	দুটি নোডের দূরত্ব	১৮৩
৮.৩.৩	তিনটি গ্লাস ও পানি	১৮৫
৮.৩.৪	UVa 10653	১৮৬
৮.৩.৫	UVa 10651	১৮৭
৮.৩.৬	০ ও 1 মূল্য (cost) এর গ্রাফ	১৮৭
৮.৪	সিঙ্গল সোর্স শর্টেস্ট পাথ (Single Source Shortest Path)	১৮৭
৮.৪.১	ডায়াকস্ট্রা'র অ্যালগরিদম (Dijkstra's Algorithm)	১৮৮
৮.৪.২	বেলম্যান ফোর্ড অ্যালগরিদম (Bellman Ford Algorithm)	১৯১
৮.৫	অল পেয়ার শর্টেস্ট পাথ (All pair shortest path) বা ফ্লয়েড ওয়ার্শল অ্যালগরিদম (Floyd Warshall Algorithm)	১৯২
৮.৬	ডায়াকস্ট্রা, বেলম্যান ফোর্ড, ফ্লয়েড ওয়ার্শল অ্যালগরিদম কেন সঠিক?	১৯৫
৮.৭	আর্টিকুলেশন ভার্টেক্স (Articulation vertex) বা আর্টিকুলেশন বাহু (Articulation edge)	১৯৬
৮.৮	অয়লার পাথ (Euler path) এবং অয়লার সাইকেল (euler cycle)	১৯৮
৮.৯	টপোলজিক্যাল সর্ট (Topological sort)	২০০
৮.১০	স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Strongly Connected Component - SCC)	২০১
৮.১১	2-satisfiability (2-sat)	২০৩
৮.১২	বাইকানেক্টেড কম্পোনেন্ট (Biconnected component)	২০৮

৮.১৩ ফ্লো (Flow) সম্পর্কিত অ্যালগরিদম	২০৬
৮.১৩.১ ম্যাক্সিমাম ফ্লো (Maximum flow)	২০৭
৮.১৩.২ মিনিমাম কাট (Minimum cut)	২১১
৮.১৩.৩ মিনিমাম কস্ট ম্যাক্সিমাম ফ্লো (Minimum cost maximum flow)	২১২
৮.১৩.৪ ম্যাক্সিমাম বাইপারটাইট ম্যাচিং (Maximum Bipartite Matching)	২১৩
৮.১৩.৫ ভার্টেক্স কাভার (Vertex cover) ও ইনডিপেন্ডেন্ট সেট (Independent set)	২১৫
৮.১৩.৬ ওয়েইটেড ম্যাক্সিমাম বাইপারটাইট ম্যাচিং (Weighted maximum bipartite matching)	২১৭
৮.১৪ প্রোগ্রামিং সমস্যা	২১৭
৮.১৪.১ অনুশীলনী	২১৭
 ৯ কিছু অ্যাডহক পদ্ধতি (Adhoc Technique)	২১৯
৯.১ কিউমিউলেটিভ যোগফল পদ্ধতি (Cumulative sum technique)	২১৯
৯.২ সর্বোচ্চ যোগফল পদ্ধতি (Maximum sum technique)	২২০
৯.২.১ একমাত্রিক সর্বোচ্চ যোগফল সমস্যা (One dimensional Maximum sum problem)	২২০
৯.২.২ দ্বিমাত্রিক সর্বোচ্চ যোগফল সমস্যা (Two dimensional Maximum sum problem)	২২২
৯.৩ প্যাটার্ন (Pattern) খোঁজা	২২৩
৯.৩.১ LightOJ 1008	২২৩
৯.৩.২ জোসেফাস সমস্যা (Josephus Problem)	২২৪
৯.৪ একটি নির্দিষ্ট সীমায় সর্বোচ্চ উপাদান	২২৬
৯.৪.১ একমাত্রিক (One Dimensional বা 1D)	২২৬
৯.৪.২ দ্বিমাত্রিক (Two Dimensional বা 2D)	২২৭
৯.৫ লীস্ট কমন অ্যানসেস্টর (Least Common Ancestor)	২২৭
৯.৬ প্রোগ্রামিং সমস্যা	২২৮
৯.৬.১ অনুশীলনী	২২৮
 ১০ জ্যামিতি (Geoemetry) এবং কম্পিউটেশনাল জ্যামিতি (Computational Geometry)	২৩০
১০.১ মৌলিক জ্যামিতি ও ত্রিকোণমিতি	২৩০
১০.২ স্থানাঙ্কভিত্তিক জ্যামিতি (Coordinate Geometry) এবং ভেক্টর (Vector) .	২৩২
১০.৩ কিছু কম্পিউটেশনাল জ্যামিতির অ্যালগরিদম	২৩৮
১০.৩.১ কনভেক্স হাল (Convex Hull)	২৩৮

10.3.2	নিকটতম বিন্দুজোড় (Closest pair of points)	280
10.3.3	পরস্পরচেছী রেখাংশ (Line segment intersection)	281
10.3.8	পিকের থেওরেম (Pick's theorem)	283
10.3.5	বহুভুজ সম্পর্কিত টুকিটাকি	288
10.3.6	লাইন সুইপ (Line sweep) এবং রোটেটিং ক্যালিপার্স (Rotating Calipers)	289
10.3.7	কিছু স্থানাঙ্ক সম্পর্কিত গণনা	292
10.8	প্রোগ্রামিং সমস্যা	298
10.8.1	অনুশীলনী	298
11	স্ট্রিং (String) সম্পর্কিত ডেটা স্ট্রাকচার ও অ্যালগরিদম	295
11.1	হ্যাশিং (Hashing)	295
11.2	নুথ-মরিস-প্র্যাট (Knuth-Morris-Pratt) বা KMP অ্যালগরিদম	297
11.2.1	KMP সম্পর্কিত কিছু সমস্যা	299
11.3	Z অ্যালগরিদম	299
11.3.1	Z অ্যালগরিদম সম্পর্কিত কিছু সমস্যা	299
11.4	ট্রাই (Trie)	299
11.5	আহো-কোরাসিক অ্যালগরিদম (Aho-corasick Algorithm)	299
11.6	সাফিক্স অ্যারে (Suffix Array)	299
11.6.1	সাফিক্স অ্যারে সম্পর্কিত কিছু সমস্যা	299
11.7	প্রোগ্রামিং সমস্যা	299
11.7.1	অনুশীলনী	299

অধ্যায় ১

প্রোগ্রামিং প্রতিযোগিতায় হাতেখড়ি

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 ঝান্ত হই না, বরং খেলা শেষে আমরা হই হই করে আনন্দে মেতে উঠি বা কষ্টে কারও সঙ্গে কথা না বলে বিপক্ষ দলকে শাপশাপান্ত করতে থাকি। প্রোগ্রামিং সমস্যা সমাধানেও ঠিক একই রকম মজা। তুমি খেতে বসে দেখবে দ্রুত খাচ্ছ কারণ তুমি হাত ধুতে গিয়ে একটা সমস্যার সমাধান পেয়ে গেছ! অথবা দেখবে ক্লাসে তোমার টিচারের পড়ানোর দিকে মন নেই, তুমি দিব্যি তোমার পাশের বন্ধুর সঙ্গে আগের রাতের কনটেন্টের প্রবলেম নিয়ে আলোচনা করছ। অথবা এও হতে পারে যে গভীর রাতে তুমি কম্পিউটারে আছো, আর তোমার মা বকা দিতে দিতে এসে বলবে - "সারারাত গেম খেলা হচ্ছে না?" কিন্তু তোমার স্ক্রিনের দিকে তাকিয়ে অবাক হয়ে বলবে - "এসব কী করিস?" আর তুমি হেসে বলবে - "কোডিং করছি, তুমি বুঝবে না, যাও ঘুমাও"। আসলে এই মজা যে পেয়েছে সেই শুধু বুঝবে আমি কী বলছি! হয়তো এখন তুমি আমার কথা বিশ্বাস করবে না, কিন্তু এক সময় তুমি বুঝবে আমি কী বোঝাতে চাচ্ছি। তোমরা যেন সবাই সেই আনন্দটা পাও এই আশা নিয়েই শুরু হোক আমাদের প্রোগ্রামিং প্রতিযোগিতায় হাতেখড়ি।

১.২ প্রোগ্রামিং প্রতিযোগিতা কী?

তোমরা যদি মনে কর যে প্রোগ্রামিং প্রতিযোগিতায় বুঝি কোনো একটি প্রোগ্রামিং ল্যাঙ্গুয়েজ যে যত ভালো পারে সে তত ভালো করবে তাহলে জেনে রাখ তোমার এই ধারণা সম্পূর্ণ ভুল। আমরা কিন্তু সেই ছোটবেলাতেই ১, ২, ৩, ৪ শিখেছি; শিখেছি যোগ, বিয়োগ, গুণ, ভাগ করা। কিন্তু এখনেই কিন্তু গণিতের শেষ হয়ে যায়নি। এর পরেও অনেক অনেক অনেক কিছু আমরা জেনেছি, শিখেছি। প্রোগ্রামিং ল্যাঙ্গুয়েজও এখানে সেই ১, ২, ৩, ৪ এর মতো। আমরা গণিতে সংখ্যাগুলোকে

যেমন এসব অঙ্ক দিয়ে প্রকাশ করে থাকি ঠিক তেমনি আমাদের প্রোগ্রামিং প্রতিযোগিতার সমস্যার সমাধানগুলো প্রোগ্রামিং ল্যাঙ্গুয়েজ দিয়ে প্রকাশ করে থাকি। এই ল্যাঙ্গুয়েজ আমাদের সমাধান প্রকাশের একটি মাধ্যম মাত্র। এটি এমন একটি মাধ্যম যা আমাদের কম্পিউটার বুবো থাকে। তোমরা মনে করো না যে কম্পিউটার নিজে নিজেই সব করে থাকে, আমি একটা সংখ্যা দিলে সে এমনি এমনিই বলে দেবে না যে সংখ্যাটা জোড় না বিজোড়। তোমাকে বলে দিতে হবে সে কীভাবে বুবে 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 কীভাবে হিসাব করতে হবে, কখন কোথায় কী মনে রাখতে হবে। ব্যাস! সে চোখের পলকে তোমাকে সেই হিসাব করে দেবে। কোনো ভুল দে করবে না। কিন্তু তুমি যদি ওকে বলতে ভুল কর তাহলে কিন্তু সেটা তোমার দোষ, ওর না। প্রোগ্রামিং প্রতিযোগিতা হলো তুমি ঠিক মতো তোমার কম্পিউটারকে সমাধানের উপায় বলে দিতে পারছ নি না তার প্রতিযোগিতা। বিভিন্ন ধরনের প্রোগ্রামিং প্রতিযোগিতা আছে। কে কত ছোট প্রোগ্রাম দিয়ে সমস্যা সমাধান করতে পারে, কে কত সুন্দর করে লজিক দাঁড় করাতে পারে, কে কত efficient সমাধান করতে পারে ইত্যাদি। আমরা এই বইয়ে যে প্রতিযোগিতা নিয়ে কথা বলব তা আমাদের কাছে ACM প্রোগ্রামিং প্রতিযোগিতা নামে পরিচিত। এখানে দেখা হয় তোমার সমাধান কত দ্রুত একটা সমস্যার সমাধান করতে পারে, কত কম মেমরি নেয় বা এমনও হতে পারে যে তোমার হাতে মেমোরি অনেক কম আছে কিন্তু সময় অনেক বেশি, সুতরাং সেক্ষেত্রে তোমাকে হয়তো মেমোরি কম, কিন্তু সময় বেশি ব্যবহার করতে হবে এরকম সমাধান বের করতে হবে। অর্থাৎ তুমি কত দক্ষতার সঙ্গে তোমাকে দেওয়া সীমাবদ্ধতার মধ্যে সমস্যার সমাধান করতে পারছ সেটাই দেখা হয় প্রোগ্রামিং প্রতিযোগিতায়।

বিশ্বে দুইটি বেশ বড় সড় প্রোগ্রামিং প্রতিযোগিতা আছে। একটি হলো ACM ICPC World finals আর আরেকটি হল International Olympiad for Informatics বা সংক্ষেপে IOI. ACM প্রোগ্রামিং এ বিশ্ববিদ্যালয় পর্যায়ের ছেলেমেয়েরা অংশগ্রহণ করে থাকে। বিভিন্ন রকমের টপিক থেকে প্রবলেম আসে: সংখ্যাতত্ত্ব (Number Theory), ক্যালকুলাস (Calculus), গ্রাফ থিওরী (Graph Theory), গেইম থিওরী (Game Theory), ডায়নামিক প্রোগ্রামিং (Dynamic Programming), সন্তাব্যতা (Probability) ইত্যাদি। এখানে আসলে টপিকের সীমাবদ্ধতা নেই। এটি দলগত প্রতিযোগিতা। বিশ্বের প্রায় 100 টি দল বা বিশ্ববিদ্যালয় প্রতি বছর ACM ICPC World Finals এ অংশ গ্রহণ করে থাকে। এইসব দল নির্বাচন হয় ICPC এর মাধ্যমে যা প্রতি বছর বিশ্বের বিভিন্ন স্থানে হয়ে থাকে। যেমন প্রতি বছর ঢাকাতে ICPC হয়ে থাকে এবং এতে যেই দল champion হয় তারা World finals এ যায়। অপর দিকে ইনফরমেটিক্স অলিম্পিয়াড এ স্কুল কলেজ লেভেলের ছেলেমেয়েরা অংশ নিয়ে থাকে। এটিতে একটি নির্দিষ্ট সিলেবাস থেকে প্রশ্ন হয়ে থাকে। যেমন সিলেবাসে calculus নেই। এর মানে এই না যে প্রবলেমগুলি সহজ হয় বরং এসব প্রতিযোগিতার সমস্যাগুলি অনেক বেশি algorithmic হয়ে থাকে। এটি individual প্রতিযোগিতা। প্রতিটি দেশ হতে চারজনের একটি দল অংশ নিয়ে থাকে। কন্টেন্টের rank অনুসারে এখানে স্বর্ণ, রৌপ্য ও ক্রোঞ্জ পদক দেয়া হয়। তোমরা হয়তো অনেকেই গণিত অলিম্পিয়াডের সাথে পরিচিত। গণিত অলিম্পিয়াডে যেমন গণিতের সমস্যা সমাধান করতে হয়, IOI তে তেমনি অ্যালগরিদমভিত্তিক সমস্যা সমাধান করতে হয়।

১.৩ কেন করব?

› অনেকে মনে করতে পারে যে প্রোগ্রামিং প্রতিযোগিতায় যারা ভালো করেছে তারা Google, Facebook, Microsoft এর মতো 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? যাই হোক, তুমি আনন্দ নিয়ে প্রোগ্রামিং করবে। তাহলেই দেখবে তুমি ভালো করছ, তখন Google, Facebook, Microsoft এর মতো কোম্পানি এমনিই তোমাকে নিয়ে যাবে। বা ওই সব কোম্পানি কেন? হয়তো তুমি নিজেই একটা Google প্রতিষ্ঠা করে ফেলবে একদিন। অথবা তুমি হয়তো এমন একটা প্রোগ্রামিং ল্যাঙ্গুয়েজ বানিয়ে ফেলবে যেটা সারা বিশ্বের মানুষ ব্যবহার করবে। এর মধ্যে অন্যরকম মজা আছে। এই অপার্থিব আনন্দ যারা পেতে চাও তাদের জন্য এই প্রোগ্রামিং প্রতিযোগিতা।

১.৪ কীভাবে শুরু করব?

এটা হলো ইন্টারনেট এর যুগ। হয়তো অদূর ভবিষ্যতে ইন্টারনেট বলে কিছুই থাকবে না, এর থেকেও আধুনিক জিনিস চলে আসবে। এই যুগে নেই বলে কথা নেই। তোমার সামনে ইন্টারনেট আছে তুমি শেখো! শেখার উৎসের কিন্তু শেষ নেই। তুমি চাইলে youtube এ সার্চ করে সেখানে ভিডিও লেকচার দেখতে পার। বা অন্য দেশের বই translator দিয়ে অনুবাদ করে পড়তে পার। বা তোমার থেকে যারা ভালো পারে তাদের কাছ থেকেও শিখতে পার। মোট কথা তোমার শেখার ইচ্ছা থাকলে তোমাকে কেউ দমাতে পারবে না। অনেকে বলে থাকে তারা নাকি অন্য কোডার এর কাছ থেকে সাহায্য পায় না। এটা ঠিক না, আমার মনে হয় আমাদের দেশে অনেকেই শেখাতে বেশ আগ্রহী। অনেক ফেসবুক গ্রুপ আছে যেখানে গোটা বাংলাদেশের অনেক প্রোগ্রামার আছে। সেসব গ্রুপে পোস্ট দিলেই দেখা যায় অনেকে সাহায্যের জন্য এগিয়ে আসে। এখন তাই বলে তুমি যদি কোন একটা সমস্যা পারছ না যেটা হয়তো তোমার বিশ্ববিদ্যালয়ের আরও 10 জন পারে, কিন্তু তুমি করলে কি Petr^১ কে মেইল করে বসলে^২ যে আমার কোড ডিবাগ (Debug) করে দাও বা এই সমস্যার আইডিয়া পাচ্ছি না আমাকে শেখায়ে দাও, তাহলে কি ঠিক হল?

অনলাইনে বেশ কিছু ওয়েবসাইট আছে যেখানে কিছু দিন পর পর প্রতিযোগিতা হয়। এসব জায়গায় পুরোনো প্রতিযোগিতার প্রবলেমও পাওয়া যায়। খেলোয়াড়রা যেমন আসল খেলার আগে প্র্যাকটিস করে, ঠিক তেমনি আমরা সেইসব পুরোনো প্রতিযোগিতার প্রবলেম সমাধান করে প্র্যাকটিস করতে পারি। বেশিরভাগ সাইটেই আবার কে কয়টি সমাধান করেছে তার একটা তালিকা থাকে। দেখতো তোমার কোনো বন্ধু ওই তালিকায় তোমার আগে আছে কিনা? থাকলে তার থেকে বেশি সমাধান কর। চেষ্টা কর তোমার সহপাঠীদের মধ্যে সবচেয়ে এগিয়ে থাকার, এর পরে চেষ্টা

^১সত্যি কথা বলতে এই সেকশনের টাইটেল ছিল "কাদের জন্য এই বই না" এবং বলা বাহুল্য এতে বেশ কিছু অপ্রিয় সত্য কথা ছিল বৈকি!

^২Petr কে না চিনলে https://en.wikipedia.org/wiki/Petr_Mitrichev পড়ে নিও!

কর দেশের সবার মধ্যে এগিয়ে থাকার। চেষ্টা করতে থাকো। এক সময় বিশ্বে সবার মধ্যে এগিয়ে থাকতে পার কিনা দেখ। তুমি যদি না পার তাহলে কিন্তু হতাশ হওয়ার কিছু নেই। এই যে তোমার এগিয়ে যাওয়ার চেষ্টা, এটাই তোমাকে প্রতিযোগিতা না করা অন্য 100 জনের চেয়ে এগিয়ে রাখবে। তুমি নিজেই বুঝবে না তুমি অন্যদের তুলনায় কত দক্ষ হয়ে গিয়েছ! হয়তো তুমি প্রথম 10 জনের একজন না, কিন্তু আগে হয়তো তুমি 1000 জনের মধ্যে ছিলেনা, এখন 100 জনের মধ্যে এসেছ এটা কিন্তু সামান্য অর্জন না!

নিচে তোমাদের সুবিধার জন্য প্রোগ্রামিং সম্পর্কিত কিছু ওয়েবসাইটের লিংক এবং এদের সংক্ষিপ্ত বিবরণ দেওয়া হলো:

- uva.onlinejudge.org প্রোগ্রামিং প্রতিযোগিতার জন্য সবচেয়ে জনপ্রিয় ওয়েবসাইট। এখানে প্রায়ই ৫ ঘণ্টার প্রতিযোগিতা হয়ে থাকে বিশেষ করে অন্টোবর থেকে ডিসেম্বর এই সময়ে। এছাড়াও এখানে আছে 8000 এরও বেশি প্র্যাকটিস প্রবলেম।
- icpcarchive.ecs.baylor.edu এটা uva ওয়েবসাইটের ভাই। এখানে 1988 সাল থেকে শুরু করে আজ পর্যন্ত হয়ে আসা বহু ICPC Regional Programming Contest এবং ACM ICPC World Finals এর প্রবলেমসমূহ আছে।
- acm.sgu.ru রাশিয়ান প্রোগ্রামিং সাইট। এখানে তুলনামূলকভাবে অনেক কম প্রবলেম আছে, কিন্তু একেকটা সমস্যা সমাধান করতে মাথার ঘাম পায়ে পড়ে! যদি কেউ এখানের সবগুলো সমস্যা সমাধান করে তাহলে আমার মনে হয় না সে কোথাও সহজে আটকাবে।
- acm.timus.ru আরও একটি রাশিয়ান সাইট। এখানে প্রবলেমগুলো বেশ কিছু ক্যাটাগরিতে ভাগ করা আছে। তোমরা যারা নতুন নতুন প্রোগ্রামিং শুরু করেছ তারা এই সাইটের Beginners Problem সেকশনের 20টি সমস্যা সমাধান করে দেখতে পার। এগুলো সমাধান করতে কোনো অ্যালগরিদম বা ডেটা স্ট্রাকচারের দরকার হয় না, শুধু প্রোগ্রামিং ল্যাঙ্গুয়েজ জানলেই চলে। এখানের সমস্যাগুলোও বেশ কঠিন হয়, হাজার হেক রাশিয়ান প্রবলেম বলে কথা! একবার আমাদের দেশের এক world finalist এর সঙ্গে কথা হচ্ছিল, সে গর্ব করে বলছিল যে সে একবার timus এর কোন এক প্রতিযোগিতার সমস্যা সমাধান করেছিল। আমি বললাম- হ্যাঁ timus এর প্রতিযোগিতার সমস্যাগুলো বেশ কঠিন হয়, কন্টেস্ট সময়ে একটা করতেই খবর খারাপ হয়। তখন সে বলে যে সে আসলে কন্টেস্ট এর তিন চারদিন পর সমাধান করেছিল। মানে এই সমস্যাগুলি এতোই কঠিন যে সেটা আসলে তিন চার দিন পর সমাধান করতে পারলেও বেশ বড় ব্যাপার বলা যায়! তার মানে কি তুমি কঠিন বলে এসব সাইট এর সমস্যা সমাধান করবে না? এসব সাইটে কন্টেস্ট করবে না? মোটেও না। তুমি যদি কঠিন কঠিন সমস্যা সমাধান না কর তোমার উন্নতি হবে না! হয়তো তুমি ICPC তে champion হয়ে world finals এ যাবে কিন্তু এরপর দেখবে আর চাইনিজ, পোলিস বা রাশিয়ানদের সাথে পেরে উঠছ না।
- www.codechef.com এখানে প্রতিমাসে তিন ধরনের প্রতিযোগিতা হয়। একটি 10 দিনব্যাপী, একটি 5 ঘণ্টার ACM স্টাইল আরেকটি 4 ঘণ্টার IOI স্টাইল প্রতিযোগিতা।

প্রতিযোগিতাগুলোতে যারা ভালো করে তারা এখানে পুরস্কার পেয়ে থাকে। প্রতিযোগিতা শেষে তারা editorial ও দিয়ে থাকে।

- www.codeforces.com প্রতি সপ্তাহে প্রায় 2টি করে প্রতিযোগিতা হয়ে থাকে এখানে। দুই division এ হয়ে থাকে সেই প্রতিযোগিতা। Division 2 তে আছে যারা beginner তারা, আর Division 1 এ আছে যারা advanced তারা। এখানে প্রতি প্রতিযোগিতা শেষে তোমার রেটিং আপডেট হবে। সুতরাং তুমি দেশের বা বিশ্বের আর সবার সঙ্গে নিজেকে তুলনা করতে পারবে! তোমার রেটিং অনুযায়ী তোমাকে রংও দেওয়া হয়। লাল রং বা Red coder মানে হলো এরা খুবই ভাল! এর মানে এই না যে orange বা purple যারা, তারা ফালতু! বরং orange বা purple হয়েও অনেকে পদক জিতেছে।
- www.topcoder.com/tc codeforces এর মতোই এখানেও রেটিং ব্যবস্থা আছে। এখানে প্রতি মাসে প্রায় 3 থেকে 4 টি প্রতিযোগিতা হয়ে থাকে। বলা হয়ে থাকে ডায়নামিক প্রোগ্রামিং শেখার জন্য এই সাইটের আগের প্রবলেমগুলো খুবই ভালো।
- acm.hust.edu.cn/vjudge এটি মূলত প্র্যাকটিস কন্টেস্ট আয়োজন করতে ব্যবহার হয়ে থাকে। এর মাধ্যমে তুমি অন্যান্য ওয়েবসাইট এর প্রবলেমগুলি নিয়ে কন্টেস্ট আয়োজন করতে পার।
- www.lightoj.com এটি বাংলাদেশের প্রথম অনলাইন জাজ (practice ওয়েবসাইটকে আমরা অনলাইন জাজ বা সংক্ষেপে OJ বলে থাকি)। এটা বানিয়েছেন জানে আলম জান। উনি ঢাকা বিশ্ববিদ্যালয় থেকে 2009 সালে ACM ICPC World Finals এ অংশগ্রহণ করেছেন।
- www.z-trening.com এটি ইনফরমেটিক্স অলিম্পিয়াড এর প্রস্তুতি নেওয়ার জন্য খুবই ভালো ওয়েবসাইট। তবে সাম্প্রতিক সময়ে প্রায়শই down থাকে।
- ipsc.ksp.sk সমগ্র বিশ্বব্যাপী IPSC খুবই সমাদৃত একটি প্রতিযোগিতা। বছরে একবার এই প্রতিযোগিতা হয়ে থাকে এবং সবাই এই কন্টেস্ট করার জন্য মুখিয়ে থাকে। এখানে বিভিন্ন ধরনের প্রবলেম দেওয়া হয়ে থাকে। এনক্রিপশন (Encryption), মিউজিক (Music), New Language, গেইম (Game) আরও নানা ধরনের প্রোগ্রামিং সমস্যা দেওয়া হয়ে থাকে যা অন্য কন্টেস্টগুলোতে দেখা যায় না বললেই চলে।
- www.spoj.com এটি একটি পোলিস ওয়েবসাইট। এখানের সমস্যাগুলোও বেশ ভালো। বিভিন্ন দেশের ইনফরমেটিক্স অলিম্পিয়াড এর প্রবলেমগুলো এখানে পাওয়া যায়। এছাড়াও এখানে বিভিন্ন টপিকের বেশ কঠিন কঠিন এবং শিক্ষণীয় প্রবলেম থাকে। কিন্তু অনেক প্রবলেমের মধ্যে থেকে বাছাই করা একটি কঠিন কাজ। তোমরা চাইলে ভালো কোনো প্রোগ্রামারের প্রোফাইল নির্বাচন করে তার সমাধান করা সব সমস্যা সমাধান করতে পার।

- ace.delos.com/usacogate মার্কিন যুক্তরাষ্ট্রের ইনফরমেটিক্স অলিম্পিয়াড দলকে প্রশিক্ষণ দেওয়ার জন্য মূলত এই ওয়েবসাইট। সেপ্টেম্বর থেকে এপ্রিল মাস পর্যন্ত প্রতিমাসে সাধারণত একটি করে কন্টেস্ট হয়ে থাকে IOI এর প্রস্তুতিস্বরূপ। এখানকার অফলাইন প্রবলেমগুলো^১ ক্যাটাগরি অনুযায়ী করা আছে এবং তুমি কোনো সমস্যা সমাধান করলে তার পর্যালোচনা বা অ্যানালাইসিস পাবে।
- 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 অ্যানালাইসিস পাবলিশ করা হতো। এখনো pku এবং zju সাইটে কন্টেস্ট হয়ে থাকে। tju ওয়েবসাইটেও অনেক প্রবলেম আছে, তবে মূলত এখানে প্র্যাকটিস কন্টেস্ট আয়োজন করা হয়ে থাকে। এসব সাইটে আসলে সহজ কঠিন সব ধরনের সমস্যা আছে। তবে আমার মনে হয়েছে চাইনিজ কঠিন সমস্যাগুলো রাশিয়ান কঠিন সমস্যাগুলোর থেকেও কঠিন হয়! একবার এক চাইনিজ প্রবলেমসেটার ঘোষণা দিয়েছিল যে, যদি কেউ তার অমুক সমস্যা পরবর্তী এক বছরে সমাধান করে তাহলে তাকে সে পুরস্কার দেবে। আমি ঠিক জানি না কেউ সমাধান করেছিল কিনা। এছাড়াও Rujia Liu এর কিছু সমস্যা তুমি uva তে খুঁজে পাবে (তার নাম দিয়ে আলাদা সেকশনই আছে uva তে), যার প্রতিটি সমস্যাই খুবই শিক্ষণীয়। Rujia Liu একসময় চীনের IOI দলকে প্রশিক্ষণ দিতেন, নিজেও ACM World Finals এ মেডেল জিতেছিলেন।
- projecteuler.net এই ওয়েবসাইটটিতে অনেক গাণিতিক সমস্যা আছে যেগুলো তুমি হাতে কলমে সমাধান করতে পারবে না, কোড করে সমাধান করতে হবে। কিন্তু এখানে মূল জিনিস হলো তোমার গাণিতিক দক্ষতা। অ্যালগরিদমিক স্কিলের থেকে এখানে তোমার গাণিতিক দক্ষতারই চর্চা বেশি হবে।
- uva সাইটের কিছু হেল্পিং টুল: uhunt.felix-halim.net ও uvatoolkit.com হলো এমন দুটি টুল। এখানে তুমি নির্দিষ্ট ক্যাটাগরির প্রবলেমের তালিকা পাবে, তোমার ইউজার আইডি দিলে তোমার জন্য এরপরে কোন সমস্যা সমাধান করা ভালো হবে তার তালিকা পাবে। এছাড়াও দুজনের ইউজার আইডি দিলে তাদের মধ্যে সমাধান করা প্রবলেম এর তুলনা বা comparison দেখায়। উল্লেখ্য যে, দুই ভাই Felix Halim এবং Steven Halim প্রোগ্রামিং জগতে বেশ নামি নাম। তারা একটি বই লিখেছেন যাতে ব্যবহৃত প্রবলেমের তালিকাও এই ওয়েবসাইটটিতে পাওয়া যাবে।
- www.e-maxx.ru এটি রাশিয়ান ভাষায় লেখা একটি ওয়েবসাইট। এখানে তুমি বিভিন্ন টপিকের উপরে লেখা আর্টিকেল পাবে। Google Translator কিংবা Bing Translator ব্যবহার করে পড়ে দেখতে পার।

¹অফলাইন বলতে আমরা প্র্যাকটিস প্রবলেম বুঝে থাকি

- infoarena.ro এটি রোমানিয়ান ভাষায় লেখা একটি ওয়েবসাইট। এখানেও বেশ কিছু ভালো আর্টিকেল আছে।
- www.hsin.hr/coci/ এটি ক্রোয়েশিয়ান ইনফরমেটিক্স অলিম্পিয়াড এর ওয়েবসাইট। এখানে বেশ কিছু প্র্যাকটিস কন্টেন্ট হয়ে থাকে।
- এখন কিছু বইয়ের নাম বলা যাক।
- Introduction to Algorithms লেখক Thomas Cormen, Charles Leiserson, Ronald 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 Mark 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 Goodrich, David Mount এবং Roberto Tamassia
- Competitive Programming লেখক Felix Halim এবং Steven Halim
- কম্পিউটার প্রোগ্রামিং লেখক তামিম শাহরিয়ার সুবিন

বেশ কিছু ভালো ভালো ব্লগও আছে।

- Petr এর ব্লগ petr-mitrichev.blogspot.com
- Bruce Merry এর ব্লগ <http://blog.bruce.merry.org.za>
- শাফায়েতের ব্লগ <http://www.shafaetsplanet.com/planetcoding/>

অনেক কিছুই তো পেলে, কিন্তু এখানে সবকিছুর কিন্তু দরকার নেই। তোমার যেটা ভালো লাগবে তা থেকে শুরু করবে। যেই OJ তে প্রবলেম সলভ করতে ভালো লাগবে সেখানে সলভ করবে। যে বই পড়তে ভালো লাগবে সেই বই পড়বে। কিন্তু প্রধান কাজ হলো করতে হবে। তুমি যদি নিজে না দেখে অন্যদের জিজ্ঞাসা করে বেড়াও যে কী করব কী পড়ব, তাহলে কিন্তু হবে না। যত বেশি প্র্যাকটিস করবে তত ভালো করতে পারবে। আগেই বলেছি এটা ইন্টারনেট এর যুগ, তুমি ইন্টারনেটে খোঁজ করলেই এসব বইয়ের preview দেখতে পাবে। সেখানে থেকেও তুমি বুঝতে পারবে কোন বইটি তোমার ভাল লাগছে।

১.৫ কী কী জানতে হবে?

শুরু করার জন্য তোমাকে শুধু একটি প্রোগ্রামিং ল্যাঙ্গুয়েজ জানলেই চলবে। ACM প্রতিযোগিতাগুলোতে C, C++, Java এই তিনটি ভাষা ব্যবহার হয়ে থাকে। অন্যদিকে ইনফরমেটিক্স অলিম্পিয়াড গুলোতে C, C++ এবং Pascal ব্যবহার হয়। আমরা বাংলাদেশের প্রোগ্রামিং প্রতিযোগিতাগুলোতে দেখে আসছি যে বেশির ভাগ মানুষই C বা 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 লিখবে তা C++ এও চলবে। উপরন্তু C++ এ কিছু বাড়ি সুবিধা আছে যা আমরা আমাদের সুবিধার জন্য ব্যবহার করে থাকি। তবে এটা সত্য যে আমরা C++ বলতে আসলে যেই অবজেক্ট অরিয়েন্টেড প্রোগ্রামিং (Object Oriented Programming) বুঝে থাকি তার ছিটে ফোটাও প্রোগ্রামিং প্রতিযোগিতায় ব্যবহার করি না বললেই চলে। সুতরাং তোমরা যদি C শেখো তাহলেই প্রোগ্রামিং প্রতিযোগিতা শুরু করে দিতে পারবে। তোমাকে পুরো C শিখে এর পরে শুরু করতে হবে তাও কিন্তু না। তোমাদের সুবিধার জন্য ২ নং অধ্যায়ে ধাপে ধাপে C এর কিছু কিছু জিনিস নিয়ে আলোচনা করা হয়েছে। তবে এই বই কিন্তু তোমাদের C শেখানোর জন্য না। তোমরা যেন C বালাই করে নিতে পার এজন্য ২ নং অধ্যায়ে খুব অল্প কথায় C এর কিছু মূল জিনিস তুলে ধরা হয়েছে। মজার জিনিস হচ্ছে প্রায় সব ল্যাঙ্গুয়েজেরই মূল জিনিস প্রায় একই রকম। if-else থাকবে, লুপ থাকবে, ফাংশন, অ্যারে সবই থাকবে, শুধু লিখবে একটু অন্যভাবে। এজন্য তুমি যদি একটি ল্যাঙ্গুয়েজ জানো তাহলে অন্য ল্যাঙ্গুয়েজের কোডও বুঝতে পারবে। মাঝে মধ্যে কিছু জিনিস না বুঝলে ইন্টারনেট তো আছেই!

অধ্যায় ২

C বালাই

২.১ একটি ছোট প্রোগ্রাম এবং ইনপুট আউটপুট

আমরা শুরু করব খুবই সহজ একটি প্রোগ্রাম দিয়ে (কোড ২.১)। এই প্রোগ্রামটা তোমরা run করলে দেখবে যথারীতি 6 আর 2 এর যোগফল 8 দেখাবে। নিচয়ই বুঝতে পারছ আমরা যোগ চিহ্নের জায়গায় বিয়োগ, গুণ বা ভাগ চিহ্ন ব্যবহার করলে যথাক্রমে 4, 12 এবং 3 দেখাবে।

কোড ২.১: simple code.cpp

```
১ #include<stdio.h>
২
৩ int main()
৪ {
৫     printf("%d\n", 6 + 2);
৬     return 0;
৭ }
```

কিন্তু তোমরা ভাবতে পার যে এই একটা যোগ করতেই এত বড় কোড লিখতে হয়? আসলে খুব শীঘ্ৰই বুঝতে পারবে যে খুব ছোট কোড দিয়ে কীভাবে অনেক বড় বড় কাজ করে ফেলা যায়। কেবল তো শুরু! যাই হোক, বেশি দূরে যাওয়ার আগে সংক্ষেপে দেখে নেই প্রতিটা লাইনের মানে:

Line 1 stdio.h নামের হেডার (header) ফাইল কে ইনক্লুড (include) করা হয়েছে। বিভিন্ন ধরনের হেডার ফাইল আছে। এই ফাইলটির কাজ হলো ইনপুট আউটপুটের কাজ করা। stdio এর পূর্ণ অর্থ হলো স্ট্যান্ডার্ড ইনপুট আউটপুট (standard input output).

Line 2 ফাঁকা লাইন। আমরা কোডের সৌন্দর্যের জন্য এরকম ফাঁকা লাইন বা স্পেস (space) বা

ট্যাব (tab) দিয়ে থাকি। এতে করে পরবর্তীতে কোড বুঝতে সুবিধা হয়।

Line 3 এখান থেকে মেইন ফাংশন (main function) শুরু হয়েছে। যখন তোমার কোড রান (run) করবে তখন এই ফাংশন থেকেই কাজ শুরু হয়। আর এই ফাংশন একটি ইন্টিজার (integer) ডেটা রিটার্ন (return) করে।

Line 5 এখানে দুটি সংখ্যার যোগফল প্রিন্ট করা হচ্ছে। তুমি যদি একই সঙ্গে যোগফল এবং বিয়োগফল প্রিন্ট করতে চাও তাহলে printf("%d %d\n", 6 + 2, 6 - 2) এভাবে লিখতে পার। বুঝতেই পারছ যে যখন কোনো একটি সংখ্যা প্রিন্ট করতে চাও তখন তোমাকে %d ব্যবহার করতে হবে। এখন তুমি নিজে একটা কাজ কর তা হলো এই লাইনকে পরিবর্তন করে লেখ printf("Summation is %d, Difference is %d\n", 6 + 2, 6 - 2). কী প্রিন্ট হয় দেখতো। আশা করি বুঝতে পারছ কীভাবে তোমার পছন্দ মতো কথাবার্তা এবং সেই সাথে তোমার হিসাবনিকাশের ফলাফল তুমি প্রিন্ট করতে পারবে। আরেকটি কথা, এই printf ফাংশনটি আউটপুটের কাজ করছে। আর এটি ব্যবহারের জন্যই আমরা প্রথম লাইনে stdio.h হেডার ফাইলটি ইনক্লুড করেছি। আমাদের আরও অনেক হেডার ফাইল আছে, আমরা ধীরে ধীরে সেসব সম্পর্কে জানব।

Line 6 মেইন ফাংশনটি 0 সংখ্যা রিটার্ন করবে। আমরা সবসময় 0 রিটার্ন করব। অন্য কিছু রিটার্ন করলে অপারেটিং সিস্টেম (operating system) মনে করে যে তোমার প্রোগ্রামটি ঠিক মত শেষ হয় নাই।

এখন এই প্রোগ্রাম (কোড 2.1) এর সমস্যা হলো, তুমি যেই যেই সংখ্যা যোগ করতে চাও তোমাকে সেই দুটি সংখ্যা কোডে গিয়ে পরিবর্তন করে আবার রান করতে হবে। আমরা চাইলে ইউজার এর কাছ থেকে দুটি সংখ্যা চেয়ে তাদের যোগ করতে পারি। অর্থাৎ দুটি সংখ্যা ইনপুট নিয়ে তাদের প্রসেস করে আমরা আউটপুট করতে চাই। এজন্য তোমাদের scanf ফাংশন ব্যবহার করতে হবে (কোড 2.2)।

কোড 2.2: simple input.cpp

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int a, b;
6     scanf ("%d %d", &a, &b);
7     printf ("%d\n", a + b);
8     return 0;
9 }
```

এখানে a এবং b হলো দুটি ভ্যারিয়েবল (variable) বা চলক। আমরা বীজগণিত করার সময় যেমন অজানা কিছুর মান কে x, y, a, b, p এরকম ধরে থাকি, ঠিক তেমনি আমাদের C তেও ভ্যারিয়েবল আছে। আমরা ইনপুট নেওয়ার আগে কিন্তু জানি না যে দুটি সংখ্যার মান কী। সুতরাং আমরা a এবং b নামে দুটি ভ্যারিয়েবল কে ডিক্লেয়ার (declare) করেছি এবং `scanf` ফাংশন দিয়ে তাতে মান ইনপুট নিয়েছি। তোমরা ভাবতে পার যে, আউটপুটের সময় শুধু a আর b ছিল অথচ ইনপুটের সময় কেন $\&a$ আর $\&b$ দেওয়া হলো? এই জিনিসটা বুঝতে তোমাদের পয়েন্টার (pointer) শিখতে হবে যা তুলনামূলক একটু কঠিন। আপাতত মনে কর যে এভাবেই লিখতে হবে। সুতরাং এখন এর পরের লাইনে আমরা $6 + 2$ এর জায়গায় শুধু $a + b$ লিখলেই ইনপুট দেওয়া সংখ্যা দুটি যোগ করে দেখিয়ে দিবে। আমরা যোগের পরিবর্তে চাইলে বিয়োগ (-), গুণ (*), ভাগ (/) বা ভাগশেষ চিহ্ন (%) ব্যবহার করতে পারি।¹

এখন কথা হলো, এতটুকু শিখেই কি একটা প্রবলেম সলভ করে ফেলা সম্ভব? অবশ্যই সম্ভব! তোমাদের অনুশীলনের জন্য এই অধ্যায়ের প্রতিটি সেকশনের শেষে ঐ সেকশন সম্পর্কিত কিছু সমস্যা দেওয়া হলো। যদি কোনো সমস্যা অনলাইন জাজ হতে নেওয়া হয় তাহলে OJ এর নাম এবং প্রবলেম নম্বর দেওয়া থাকবে।

অনুশীলনী

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

২.২ ডেটা টাইপ এবং `math.h` হেডার ফাইল

এখন কিছু পরীক্ষা নিরীক্ষা করা যাক। তোমরা আগের প্রোগ্রাম (কোড ২.২) এ যোগের জায়গায় গুণ দাও ($a * b$) আর `run` করে 100000 এবং 100000 ইনপুট দাও। দেখবে মাথা খারাপ করা উভয় দিয়েছে (ঝণাত্রক দিলেও কিন্তু অবাক হয়ে না)! না, তুমি তোমার প্রোগ্রামে ভুল করনি। তুমি ছোট সংখ্যা ইনপুট দিলেই দেখবে তোমার প্রোগ্রাম দিব্য সঠিক উত্তরটিই দিচ্ছে। তাহলে কাহিনি কী? আসলে সব কিছুর একটা সীমা আছে। যারা পাইথন (python) প্রোগ্রামিং ভাষা জান না তাদের জ্ঞাতার্থে বলি, পাইথনে যত বড় সংখ্যাই দাও না কেন কোনো সমস্যা নেই! সে হিসাব নিকাশ করে সঠিক উত্তরই দেবে। হয়তো সময় একটু বেশি লাগবে। যাই হোক, এখানে আমরা যেই a বা b ভ্যারিয়েবলটি ডিক্লেয়ার করেছি তা কিন্তু `int` টাইপ। একটি `int` টাইপের ভ্যারিয়েবল -2^{31} থেকে $2^{31} - 1$ পর্যন্ত মানের হিসাব নিকাশ করতে পারে।² তুমি যদি আরেকটু বড় সংখ্যার হিসাব নিকাশ করতে চাও তাহলে `int` এর পরিবর্তে `long long` ব্যবহার করতে পার। এর হিসাবের সীমা হলো -2^{63} থেকে $2^{63} - 1$ পর্যন্ত। আমরা `int` টাইপ এর ডেটা ইনপুট আউটপুটের জন্য যেমন `%d`

¹ $a \% b$ এর মান হলো a কে b দিয়ে ভাগ করলে যত ভাগশেষ থাকবে তত।

² তোমাদের কল্পনার সুবিধার জন্য বলে রাখি $2^{31} \approx 2 * 10^9$.

ব্যবহার করতাম ঠিক তেমনি long long এর জন্য %lld ব্যবহার করতে হবে।^১

এখন তুমি প্রোগ্রামটিতে গুণের পরিবর্তে ভাগ বসিয়ে দাও আর 3 কে 2 দিয়ে ভাগ দাও। উভয় তো তুমি জানই 1.5 কিন্তু তোমার প্রোগ্রাম কত বলে? নিশ্চয়ই 1? কী ভাবছ? কম্পিউটার ভূল করেছে? মোটেও না। এটা সবসময় মনে রাখবে, কম্পিউটার কখনও ভূল করে না। তাহলে কাহিনি কী? কাহিনি হলো, যদি a ও b দুটিই int হয় তাহলে a/b হবে তাদের ভাগফলের ইন্টিজার অংশ। এখন প্রশ্ন আসতে পারে 1.5 কীভাবে পাব? তুমি যদি দশমিক সংখ্যা ব্যবহার করতে চাও তাহলে তোমাকে double বা float ডেটা টাইপ ব্যবহার করতে হবে।^২ দুরকম ডেটা টাইপ থাকার কারণ হলো সেই int আর long long থাকার মতো। float এর সীমা কম, double এর সীমা বেশি। তবে আমি সবসময় বলে থাকি কেউ যেন কখনও float ব্যবহার না করে। কারণ এর precision অনেক কম।^৩ Precision মানে মনে করতে পার দশমিকের পরে কয় ঘর পর্যন্ত উভয় মনে রাখবে। মনে কর float হয়তো $\sqrt{2}$ এর জন্য শুধু 1.4 মনে রাখবে আর double মনে রাখবে 1.414 পর্যন্ত। এরকম আর কি। কিন্তু তাই বলে আবার int ব্যবহার না করে সবসময় long long ব্যবহার করো না। কারণ int এর তুলনায় long long বেশ slow。 তোমরা প্রবলেম সলভিংয়ে একটু অভ্যন্ত হয়ে গেলে signed ও unsigned ডেটা টাইপ সম্পর্কেও জেনে রাখবে। কারণ প্রবলেম সলভিংয়ে মাঝে মধ্যেই এদের দরকার হয়।

আমরা এখন পর্যন্ত একটাই হেডার ফাইল দেখেছি- stdio.h。 আরও একটি হেডার ফাইল দেখা যাক, এটা হলো math.h হেডার ফাইল। নাম দেখেই বোৰা যাচ্ছে এখানে গণিত সংক্রান্ত কিছু ফাংশন দেওয়া আছে। আমাদের ক্যালকুলেটরে যেমন অনেক ফাংশন আছে (যেমন sin, cos, tan, square root, square, cube ইত্যাদি) ঠিক তেমনি এই math.h হেডার ফাইলে এ ধরনের বেশ কিছু ফাংশন আছে। টেবিল ২.১ তে math.h এর কিছু গুরুত্বপূর্ণ ফাংশন দেওয়া হলো।

কোড ২.৩ এ একটি বৃত্তের ক্ষেত্রফল নির্ণয় করার কোড দেওয়া হলো। এখানে খেয়াল কর আমরা 10 নম্বর লাইনে কীভাবে π এর মান নির্ণয় করলাম। আমরা জানি যে, $\cos \pi = -1$ সূতরাং $\cos(-1)$ হলো π 。^৪

কোড ২.৩: circle area.cpp

```
১ #include<stdio.h>
২ #include<math.h>
৩
৪ int main()
৫ {
```

^১অনেক compiler এ long long এর পরিবর্তে _int64 আছে এবং এর সঙ্গে %I64d ব্যবহার করতে হয়।

^২double ও float এ ইনপুট আউটপুটের জন্য যথাক্রমে %lf ও %f ব্যবহার করা হয়।

^৩তোমরা 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।

^৪অনেকের ভুল ধারণা আছে যে $\pi = \frac{22}{7}$, এটা কেবলমাত্র একটি approximation. তুমি $\frac{22}{7}$ মান দিয়ে হিসাব করলে কখনই সঠিক উভয় পাবে না।

টেবিল ২.১: math.h এর কিছু ফাংশনের তালিকা

sqrt(x)	এটা x এর বর্গমূল (square root) নির্ণয় করে। x কে অবশ্যই অস্থানাঞ্চক হতে হবে। না হলে Run Time Error (RTE) হবে।
fabs(x)	এটা x এর পরম মান (absolute value) নির্ণয় করে।
sin(x), cos(x), tan(x)	x এর sin, cos, tan নির্ণয় করে থাকে। এখানে x কে রেডিয়ান (radian) এককে দিতে হবে।
asin(x), acos(x), atan(x)	x এর \sin^{-1} , \cos^{-1} , \tan^{-1} নির্ণয় করে থাকে। এখানে \sin^{-1} এবং \cos^{-1} এর ক্ষেত্রে x কে অবশ্যই $[-1, 1]$ সীমার মধ্যে হতে হয় এবং রিটার্ন মানটি রেডিয়ান এককে থাকে।
atan2(y, x)	উপরের asin, acos এর মতোই। তবে যেহেতু $\Delta y = 1$, $\Delta x = 0$ হলে \tan^{-1} এর মান atan ফাংশন দিয়ে বের করা যায় না (atan এ তোমাকে $\frac{\Delta y}{\Delta x}$ দিতে হবে আর তা দিতে গেলে division by zero হয়ে যাবে), সেহেতু সেক্ষেত্রে আমরা atan2 ব্যবহার করতে পারি।
pow(x, y)	এটি x^y নির্ণয় করে থাকে।
exp(x)	এটি e^x নির্ণয় করে।
log(x), log10(x)	এখানে log হলো স্বাভাবিক লগারিদম(natural logarithm) আর log10 হলো 10 ভিত্তিক লগারিদম।
Floor(x), ceil(x)	যথাক্রমে floor এবং ceiling দেয়।

```

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

অনুশীলনী

- দুটি দ্বিমাত্রিক (2D) বিন্দুর স্থানাঙ্ক (co-ordinate) ইনপুট নিয়ে তাদের মাঝের দূরত্ব প্রিন্ট কর।

- ঃ একটি ত্রিভুজের তিনটি বাহুর দৈর্ঘ্য দেওয়া আছে, তার তিনটি কোণ নির্ণয় কর।
- ঃ একটি ত্রিভুজের তিনটি বাহুর দৈর্ঘ্য দেওয়া আছে, তার ক্ষেত্রফল নির্ণয় কর।
- ঃ একটি বৃক্ষের ব্যাসার্ধ দেওয়া আছে, পরিসীমা নির্ণয় কর।
- ঃ কোনো একটি সংখ্যার বর্গমূলের কাছের পূর্ণসংখ্যাটি প্রিন্ট কর।
- ঃ একটি ত্রিভুজের শীর্ষবিন্দুগুলোর স্থানাঙ্ক দেওয়া আছে, ক্ষেত্রফল প্রিন্ট কর।

২.৩ if - else if - else

এতক্ষণ আমরা যা করলাম তা কিন্তু একটা সাধারণ ক্যালকুলেটর দিয়েও করা যায়। কিন্তু আমরা একটি ক্যালকুলেটরে কিন্তু লজিকাল কাজ করতে পারি না। যেমন আমরা যদি একটা সাল লিখি, ধর 2004, তাহলে আমাদের ক্যালকুলেটর কিন্তু বলতে পারবে না যে সেটা অধিবর্ষ বা লীপইয়ার (leap year) কি না। একটি সাল তখনি অধিবর্ষ হবে যদি সেটা 400 দিয়ে বিভাজ্য হয় বা 4 দিয়ে বিভাজ্য হয় কিন্তু 100 দিয়ে বিভাজ্য না হয়। কিছু উদাহরণ দেখা যাক। যেমন 2000 সাল কিন্তু অধিবর্ষ কারণ এটি 400 দিয়ে বিভাজ্য। কিন্তু 1900 সাল কিন্তু অধিবর্ষ না। কারণ এটি 4 এবং 100 দুইটি দিয়েই বিভাজ্য। তবে 2016 কিন্তু অধিবর্ষ। কারণ এটি 4 দিয়ে বিভাজ্য হলেও 100 দিয়ে বিভাজ্য নয়। আর যদি 4 দিয়েও বিভাজ্য না হয় তাহলে তো কোন কথাই নেই, যেমন 2015 চোখ বন্ধ করে অধিবর্ষ নয়। এই যে বিভিন্ন কিছু দিয়ে ভাগ করে করে লজিকালি অধিবর্ষ বের করার কাজ কিন্তু calculator এ সম্ভব না। হ্যাঁ হয়তো তুমি এটা দেখতে পার যে সংখ্যাটা 400 দিয়ে ভাগ যায় কিনা অথবা 100 বা 4 দিয়ে ভাগ যায় কিনা, এর পর তুমি নিজে সিদ্ধান্ত নিবে যে বছরটি অধিবর্ষ কি না। কিন্তু তুমি একটি প্রোগ্রামের সাহায্যে সহজেই একটি সাল অধিবর্ষ কি না তা নির্ণয় করতে পারবে। এর জন্য যা প্রয়োজন তা হলো if-else if-else। এই জিনিসটি কীভাবে ব্যবহার হয় তা কোড ২.৪ এ দেখানো হলো। এটা কোনো সঠিক কোড নয়, এখানে শুধুমাত্র সিনট্যাক্স (syntax) টি দেখানো হলো।^১

কোড ২.৪: simple if

```

1 if (condition1)
2 {
3     // if condition1 is true
4 }
5 else if (condition2)
6 {
7     // otherwise, if condition2 is true

```

¹syntax মানে হলো লেখার নিয়ম। যেমন প্রায় প্রতিটি লাইনের শেষে সেমিকলন দিতে হয়। এটাই সিনট্যাক্স।

```

8 }
9 ...
10 else
11 {
12     // if no conditions are true
13 }

```

এখন প্রশ্ন হলো আমরা শর্ত (condition) লিখব কীভাবে? আমাদের যেমন +, -, *, / চিহ্ন আছে (এদেরকে arithmetic অপারেটর বলা হয়) ঠিক তেমনি কিছু comparison অপারেটর আছে। যেমন: <, >, <=, >=, ==(equal), !=(not equal). আমরা একটা খুব সহজ কোড দেখি যা বলতে পারে যে ইনপুট সংখ্যাটা জোড় না বিজোড় (কোড ২.৫)।

কোড ২.৫: odd even.cpp

```

1 #include<stdio.h>
2
3 int main()
4 {
5     int a;
6     scanf ("%d", &a);
7
8     if (a % 2 == 0)
9     {
10         printf ("%d is even\n", a);
11     }
12     else
13     {
14         printf ("%d is odd\n", a);
15     }
16
17     return 0;
18 }

```

তোমরা যদি একটু চিন্তা কর খুব সহজেই অধিবর্ষ এর জন্যও প্রোগ্রাম লিখে ফেলতে পারবে। আমরা প্রথমে দেখব সংখ্যাটি 400 দিয়ে ভাগ যায় কিনা। গেলে তো হয়েই গেলো। আর যদি না যায় তাহলে দেখব সংখ্যাটি 100 দিয়ে ভাগ যায় কিনা। যদি যায় তাহলে তো অধিবর্ষ না। এরপর দেখব 4 দিয়ে যায় কিনা। প্রোগ্রামটার কোড তোমাদের কোড ২.৬ এ দেখানো হলো।

কোড ২.৬: leap year.cpp

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int year;
6     scanf("%d", &year);
7
8     if(year % 400 == 0)
9     {
10         printf("%d is Leap Year\n", year);
11     }
12     else if(year % 100 == 0)
13     {
14         printf("%d is not Leap Year\n", year);
15     }
16     else if(year % 4 == 0)
17     {
18         printf("%d is Leap Year\n", year);
19     }
20     else
21     {
22         printf("%d is not Leap Year\n", year);
23     }
24
25     return 0;
26 }
```

তোমরা চাইলে কিন্তু এখানে curly brace {} গুলো সরিয়ে ফেলতে পার। যদি কোনো if / else if / else ব্লকের ভেতরে কেবলমাত্র একটি লাইন থাকে তাহলে curly brace না দিলেও চলে। আরও একটি জিনিস, তা হলো তোমরা অর্থবোধক ভ্যারিয়েবলের নাম ব্যবহার করলে দেখবে পরবর্তীতে কোডটি বুঝতে সুবিধা হবে।

অধিবর্ষ নির্গয়ের কোডটি (কোড ২.৬) কিন্তু অনেক বড়। আমরা চাইলেই এই কোডটিকে অনেক ছোট করে ফেলতে পারি। তবে এজন্য আমাদের জানতে হবে লজিকাল অপারেটর (logical operator) সম্পর্কে। লজিকাল অপারেটর তিনটি: || (or), && (and), ! (not). দুইটি শর্তের এর মধ্যে || দিলে তাদের কোনো একটি সত্য হলেই পুরোটি সত্য হবে, && দিলে দুটি সত্য হলেই

কেবল পুরোটি সত্য হবে আর ! কোনো একটি শর্তের সামনে দিলে ওই শর্তটি মিথ্যা হলেই কেবল পুরোটি সত্য হবে এবং ভেতরের শর্ত সত্য হলে ! এর জন্য জিনিসটি মিথ্যা হয়ে যাবে। কোড ২.৭ এ আমরা লজিকাল অপারেটর ব্যবহার করে কীভাবে অধিবর্ষ নির্ণয়ের ছোট প্রোগ্রাম লেখা যায় তা দেখালাম। খেয়াল করলে দেখবে যে আমরা লজিকগুলোকে ব্র্যাকেট বন্দি করেছি। জিনিসটা কিছুটা $5 - (2 + 1)$ আর $5 - 2 + 1$ এর মতো বা $5 + 2 * 1$ এর মতোও মনে করতে পার। আমরা স্বত্বাবত্ত্ব জানি যে গুণের কাজ যোগের আগে হবে, কিন্তু আমরা যদি নিশ্চিত না হই তাহলে তাদের ব্র্যাকেট বন্দি করে ফেলাই বুদ্ধিমানের মতো কাজ। আমার জানা মতে লজিকাল অপারেটরদের মধ্যে `not(!)` সবার আগে, এরপর `and (&&)` আর সবশেষে `or (||)` হয়।

কোড ২.৭: leap year2.cpp

```

1 #include<stdio.h>
2
3 int main()
4 {
5     int year;
6     scanf ("%d", &year);
7
8     if (year % 400 == 0 ||
9         (year % 100 != 0 && year % 4 == 0))
10    printf ("%d is Leap Year\n", year);
11 else
12    printf ("%d is not Leap Year\n", year);
13
14 return 0;
15 }
```

এখন তাহলে তোমার নিজের ইচ্ছামতো কিছু লজিকাল প্রবলেম সলভ করতে পার। নিচে কিছু প্র্যাকটিস প্রবলেম দেওয়া হলো:

অনুশীলনী

ঃ প্যালিনড্রম (palindrome) হলো সেই জিনিস যা সামনে থেকে পড়তেও যা, পেছন থেকে পড়তেও তা। যেমন কিছু প্যালিনড্রম সংখ্যা হলো: 1, 2, 3, ..., 9, 11, 22, 33, ..., 99, 101, 111, 121, ...। তোমাকে n তম প্যালিনড্রম সংখ্যা প্রিন্ট করতে হবে। ($n < 100$)

∴ n এর মান দেওয়া আছে, তোমাকে $\sum_{i=1}^n i * (n - i + 1) = 1 * n + 2 * (n - 1) + \dots n * 1$ এর মান বের করতে হবে। (এটা কিন্তু if-else এর প্র্যাকটিস প্রবলেম!)

∴ দুটি সংখ্যার মধ্যে বড়টি প্রিন্ট কর। এর পরে তিনটি সংখ্যার জন্যও চেষ্টা করে দেখ।

∴ তিনটি সংখ্যা ইনপুট নিয়ে তাদের ছোট হতে বড় অনুসারে প্রিন্ট কর।

∴ একটি স্থানাঙ্ক দেওয়া আছে, তোমাকে বলতে হবে সেটা কোন quadrant এ পরে।

∴ Timus 1068

২.৪ লুপ (Loop)

লুপ (loop) মানে হচ্ছে কোনো একটা কাজ বার বার করা। যেমন ধর আমি চাচ্ছি যে আমাদের এর আগের $a + b$ এর প্রোগ্রামটি (কোড ২.২) বার বার চলুক। বা ধর আমরা চাচ্ছি যে ১ থেকে 100 পর্যন্ত যোগ করতে চাই, অর্থাৎ প্রথমে আমি 1 যোগ করব, এর পর 2, এর পর 3 এভাবে 100 পর্যন্ত। এসব কাজ লুপ এর সাহায্যে করা হয়ে থাকে। C তে লুপ মূলত তিনি রকমের। For লুপ, While লুপ, Do-While লুপ। আমরা আপাতত প্রথম দুটি নিয়ে কথা বলব, পরবর্তী কোনো এক অধ্যায়ে আমরা তৃতীয়টির ব্যাপারে কথা বলব। আসলে সত্যি কথা বলতে, আমরা প্রথম দুটিই সাধারণত ব্যবহার করে থাকি। if-else এ যেমন একটি লাইন হলে curly brace দেওয়ার দরকার হয় না এক্ষেত্রেও কিন্তু তাই। কোড ২.৮ এ for লুপ ও while লুপের সিনট্যাক্স এবং কিছু উদাহরণ দেখানো হয়েছে। আসলে তুমি জিনিসটা উদাহরণগুলো থেকে বুঝতে পারবে ভালো মতো।^১ এখানের কোডটুকু কিন্তু মেইন ফাংশনের ভেতরে রাখা হয়নি, আমরা কোডকে সংক্ষিপ্ত রাখার জন্য এরকম করেছি।

কোড ২.৮: simple loop.cpp

```
1 // prototype(not syntactically valid line)
2 for (initialization; condition; increment/decrement) {}
3 while (condition is true) {}

4 // Examples
5 // prints from 1 to 10
6 for (i = 1; i <= 10; i++) printf("%d\n", i);
7 // prints from 10 to 1
8 for (i = 10; i >= 1; i--) printf("%d\n", i);
```

¹i++ মানে হলো i = i + 1 এবং i += 2 মানে হলো i = i + 2. কোডে থাকা double slash (//) দিয়ে কমেন্ট বা মন্তব্য বুঝিয়ে থাকে। পরবর্তীতে কোড ভালো মতো বুঝার জন্য এভাবে কমেন্ট করে রাখা হয়।

```

10 // prints odd numbers from 1 to 10
11 for (i = 1; i <= 10; i += 2) printf("%d\n", i);
12
13 i = 5;
14 // prints 10, 12, 14
15 while (i <= 7) { printf("%d\n", i * 2); i++; }

```

এটা জেনে রাখা ভালো যে for লুপের কোন জিনিসের পর কোন জিনিসের কাজ হয়। প্রথমে ইনিশিয়ালাইজেশন (initialization) হয়। এরপর শর্ত যদি সত্য হয় তাহলে সে ভেতরে চুকবে অন্যথা লুপ থেকে বেরিয়ে যাবে। সব কাজ শেষে সে ইনক্রিমেন্ট (increment) / ডিক্রিমেন্ট (decrement) অংশে যাবে। এরপর আবার শর্ত যাচাই করে লুপের ভেতরে চুকবে বা বাইরে চলে যাবে। while লুপ সে তুলনায় অনেক সোজা। এক্ষেত্রে প্রথমে শর্ত যাচাই করবে, যদি সত্য হয় তাহলে ভেতরে চুকবে না হলে বাইরে বেরিয়ে যাবে। ভেতরের কাজ শেষে আবার শর্ত যাচাই হয়। এভাবে চলতে থাকবে। যদি এটুকু বুঝে থাক তাহলে বলো কোড ২.৮ এর প্রতিটি লুপের শেষে। এর মান কত হয়? একটু চিন্তা করলে দেখবে, প্রথম for লুপ শেষে । এর মান 11, দ্বিতীয় for লুপ শেষে 0, তৃতীয় for লুপ শেষে 11 এবং while লুপ শেষে 8. লুপ ব্যবহার করে আরও কিছু উদাহরণ কোড ২.৯ এ দেখানো হলো।

কোড ২.৯: simple loop2.cpp

```

1 // counts how many 2 divides 100
2 x = 100;
3 cnt = 0;
4 while (x % 2 == 0)
5 {
6     // we could write "x /= 2"
7     x = x / 2;
8     cnt++;
9 }
10
11 // finds out the highest number which is
12 // power of 2 and less than 1000
13 x = 1;
14 // if multiplying by 2 does not exceed
15 // 1000, multiply.
16 while (x * 2 < 1000)
17 {

```

```

18     x *= 2;
19 }
20
21 // same thing using for loop. Note there is a
22 // semicolon after the for loop. at the end of
23 // the loop, you will find desired value in x.
24 for (x = 1; x *= 2 < 1000; x *= 2);

```

লুপের জন্য খুবই গুরুত্বপূর্ণ দুটি কিওয়ার্ড (keyword) হলো: break এবং continue। আমরা চাইলে যেকোনো সময় আমাদের লুপ ভেঙে বের হয়ে যেতে পারি। আবার আমরা চাইলে যেকোনো সময় লুপের ভেতরে থাকা বাকি কাজগুলো না করে লুপের পরবর্তী ইটারেশন (iteration) এ চলে যেতে পারি। break ও continue সহ আরও কিছু উদাহরণ তোমাদের কোড ২.১০ এ দেখানো হলো।

কোড ২.১০: simple loop3.cpp

```

1 // prints odd numbers from 1 to 10
2 for (i = 1; i <= 10; i++)
3 {
4     // if even, continue the loop,
5     // don't go down.
6     if (i % 2 == 0) continue;
7     printf("%d\n", i);
8 }
9
10 // prints only 1, 2 and 3
11 for (i = 1; i <= 10; i++)
12 {
13     // if greater than 3,
14     // break the loop.
15     if (i > 3) break;
16     printf("%d\n", i);
17 }
18
19 // takes input until the input is 0
20 // sometimes it is needed for OJ's.
21 // EOF = End Of File.

```

```

22 // take input while it is not end of file.
23 while (scanf ("%d", &a) != EOF)
24 {
25     // if the input is 0 break the loop.
26     if (a == 0) break;
27     printf ("%d\n", a);
28 }
29
30 // or in short ...
31 while (scanf ("%d", &a) != EOF && a)
32 {
33     printf ("%d\n", a);
34 }

```

অনেক সময় আমাদের একটি লুপের ভেতরে আরেকটি লুপ লেখার প্রয়োজন হয়। যেমন, ধরা যাক আমাদের n দেওয়া আছে আর আমাদের বের করতে হবে: $1 + (1 + 2) + (1 + 2 + 3) + \dots + (1 + 2 + \dots + n)$. এখন খেয়াল কর, আমাদের যদি শুধু $(1 + 2 + \dots + n)$ বের করতে দেওয়া হতো তাহলে কিন্তু কাজটা বেশ সহজ। একটি for লুপ 1 থেকে n পর্যন্ত চালিয়ে যোগফল বের করলেই হয়। কিন্তু আমাদের সমস্যায় 1 থেকে কত পর্যন্ত যোগ করতে হবে তাও কিন্তু এখানে পরিবর্তন হচ্ছে। প্রথমে 1 পর্যন্ত, এরপরে 2 পর্যন্ত এরকম করে শেষে n পর্যন্ত। সুতরাং আমরা যা করব তা হলো একটা for লুপ দিয়ে আমরা upper bound টিকে নির্ধারণ করব আর আরেকটি for লুপ দিয়ে আমরা যোগ করব।^১ এই কোডটি কোড ২.১১ এ দেখানো হলো।

কোড ২.১১: simple loop4.cpp

```

1 // very important. many of you forget to
2 // initialize variable
3 sum = 0;
4 // i is the upper bound.
5 for(i = 1; i <= n; i++)
6     // we will sum up the numbers from 1 to i.
7     for(j = 1; j <= i; j++)
8         sum += j;

```

^১তোমরা চাইলে কিন্তু ভেতরের লুপটির জায়গায় সুত্র বসিয়ে দিতে পার, বা পুরো জিনিসটাই কিন্তু সুত্র দিয়ে করা যায় :)

অনুশীলনী

∴ নিচের সিরিজগুলো কোড লিখে সমাধান কর:

১. $1 + 2 + 3 + \dots + n$
২. $1^2 + 2^2 + 3^2 + \dots + n^2$
৩. $1^1 + 2^2 + 3^3 + \dots + n^n$
৪. $1 + (2 + 3) + (4 + 5 + 6) + \dots + \text{nth term}$
৫. $1 - 2 + 3 - 4 + 5 \dots \text{nth term}$
৬. $1 + (2 + 3 * 4) + (5 + 6 * 7 + 8 * 9 * 10) + \dots + \text{nth term}$
৭. $1 * n + 2 * (n - 1) + \dots + n * 1$

∴ n ইনপুটের জন্য চিত্র ২.১ এর পিরামিডগুলো প্রিন্ট করার প্রোগ্রাম লিখ। না তোমাকে সবগুলি পিরামিড একত্রে প্রিন্ট করতে হবে না। আলাদা আলাদা করে প্রিন্ট করলেই চলবে।

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

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

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

∴ প্যালিনড্রম হলো সেই জিনিস যা সামনে থেকে পড়তেও যা, পেছন থেকে পড়তেও তা। যেমন কিছু প্যালিনড্রম সংখ্যা হলো: $1, 2, 3, \dots, 9, 11, 22, 33, \dots, 99, 101, 111, 121, \dots$ তোমাকে n তম প্যালিনড্রম সংখ্যা প্রিন্ট করতে হবে। ($n < 10^9$) (এই সমস্যাটি আমের সেকশনে ছিল তবে কম মানের জন্য)

∴ কোনো একটি সংখ্যা n মৌলিক (prime) হবে যদি সেটি 1 থেকে বড় হয় এবং 1 বাই ছাড়া আর কোনো ধনাত্মক সংখ্যা দিয়ে বিভাজ্য না হয়। তোমাকে n দেওয়া আছে কলাতে হবে এটি মৌলিক কি মৌলিক নয়।

∴ $n!$ (n ফ্যাক্টোরিয়াল) নির্ণয় কর।

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

ঃ x ও n দেওয়া আছে, তোমাকে $\cos x$ এর মান ম্যাকলরিনের ধারা (maclaurine series) এর সাহায্যে বের করতে হবে। $\cos x$ এর ধারাটি হচ্ছে $1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots + \text{nth term}$

ঃ কিছু OJ এর প্রবলেম: – 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 1211^৪ – LightOJ 1216^৫ – LightOJ 1294
- LightOJ 1305 – LightOJ 1311 – LightOJ 1331 – LightOJ 1433
- Last but not the least UVa 100

২.৫ অ্যারে (Array) ও স্ট্রিং (String)

ধর একটি গেইম শো'তে 10 জন প্রতিযোগী আছে। উপস্থাপক একটি করে প্রশ্ন করে, প্রতিযোগীদের বাজার টিপে উত্তর করতে হবে। উত্তর ঠিক থাকলে 1 পয়েন্ট করে পাবে। গেইম শেষে যার পয়েন্ট সবচেয়ে বেশি সে জয়ী। একাধিক জনও বিজয়ী হতে পারে। এখন তোমাকে এর জন্য একটি প্রোগ্রাম লিখতে বলা হলো। তুমি কী করবে? যা করতে পারো তা হলো সবার পয়েন্টের হিসাব রাখার জন্য আলাদা আলাদা 10টি ভ্যারিয়েবল রাখবে, ধর ভ্যারিয়েবল গুলো হলো: a, b, \dots, j . এর পর তোমাকে যদি বলা হয় প্রথম প্রতিযোগী সঠিক উত্তর দিবে তার পয়েন্ট তুমি a এর মান এক বাড়াবে, এরকম করে যেই প্রতিযোগী ঠিক উত্তর দিবে তার পয়েন্ট তুমি বাড়াবে। কিন্তু এই সমাধানটি অনেক ঝামেলার। কারণ, তোমাকে 10টা if-else লাগিয়ে যাচাই করতে হবে যে কোন ভ্যারিয়েবলের মান তুমি বাড়াবে। আবার গেইম শেষে তোমাকে অনেকগুলো if-else দিয়ে বের করতে হবে যে কে বা কারা জয়ী। এখন যদি তোমার প্রতিযোগী সংখ্যা আরও বাড়ে তাহলে?

^১সুত্র বের কর। তোমাকে দ্বিঘাত সমীকরন (quadratic equation) এর সমাধান করতে হতে পারে।

^২প্যাটার্ন বের কর

^৩এই সূত্রটি বের করে না করে 0 হতে B এর উত্তর থেকে 0 থেকে $A - 1$ এর উত্তর বিয়োগ করলে সমাধানটি সহজ হয়।

^৪এই সমস্যার একটি সুন্দর সমাধান আছে। আমি তোমাকে 2D এর জন্য সমাধান বলি। খেয়াল করলে দেখবে, দুটি আয়তক্ষেত্রের intersection ও কিন্তু আরেকটি আয়তক্ষেত্র। আমরা যদি এই আয়তক্ষেত্রের দুটি কর্ণবিন্দু বের করতে পারি তাহলেই হয়ে যাবে। এখন দেখ এর নিচের বাম কোনার x হবে মূল আয়তক্ষেত্র দুটির নিচের বাম কোনার x এর যেটি বড় সেটি। একইভাবে নিচের বাম কোণার y , উপরের ডান কোণার x, y বের করে ফেল। যদি আয়তক্ষেত্র দুটি intersect না করে তাহলেও কিন্তু তুমি এই যে যেই দুইটি কর্ণের স্থানাঙ্ক বের করলে তা দেখেই বুঝতে পারবে। নিজে করে দেখ মজা পাবে।

^৫সুত্রটি বের করার জন্য কিন্তু তোমার ইন্টারনেটের সাহায্য নেওয়ার দরকার নেই!

এ অসুবিধা দূর করার জন্যই কম্পিউটার প্রোগ্রামিংয়ে অ্যারে (array) নামক জিনিসটা আছে। অ্যারে আর কিছুই না, অনেকগুলো ভ্যারিয়েবলের সমন্বয়। আমরা যদি বলি `int a[10]` এর মানে `int` টাইপের 10টি ভ্যারিয়েবল তৈরি হয়ে যাবে। এদের নাম হবে: $a[0], a[1], \dots a[9]$. মনে কর তোমাকে বলা হলো যে প্রথম প্রতিযোগী $id = 1$ একটি সঠিক উত্তর দিয়েছে তাহলে কিন্তু $a[id - 1]++$ করলেই প্রথম প্রতিযোগীর ভ্যারিয়েবলের মান এক বেড়ে যাবে।^১ এখন সবার শেষে আসে সর্বোচ্চ পয়েন্ট বের করার কাজ, চিন্তা করলে দেখবে একটি `for` লুপের সাহায্যে খুব সহজেই সর্বোচ্চ সঠিক উত্তরদাতাকে পেয়ে যাবে। যেমন: 10 জন প্রতিযোগী এবং 100টি প্রশ্নের খেলায় বিজয়ী নির্ণয়ের প্রোগ্রামটির কোড ২.১২।

কোড ২.১২: simple array.cpp

```

1 // initialization. all the scores are 0.
2 for (i = 0; i < 10; i++) a[i] = 0;
3
4 for (i = 0; i < 100; i++)
5 {
6     // the player giving correct answer
7     scanf("%d", &id);
8     // increment players point
9     a[id - 1]++;
10 }
11
12 // initializing max score
13 maximum_score = 0;
14 for (i = 0; i < 10; i++)
15     // if i'th players score is more than the max
16     if (maximum_score < a[i])
17         // set max score to this value
18         maximum_score = a[i];
19
20 printf("Winners are:\n");
21 for (i = 0; i < 10; i++)
22     // if i'th players score is the maximum
23     if (maximum_score == a[i])
24         // print his id

```

^১0-indexing আর 1-indexing এর ব্যাপারটি খেয়াল রাখবে।

আমরা এতক্ষণ শুধু int ও double টাইপ ভ্যারিয়েবল নিয়েই কাজ করেছি। কিন্তু যদি কারও নাম, বা শহরের নাম এসব নিয়ে কাজ করতে চাই তার জন্য কিন্তু আলাদা ডেটা টাইপ আছে আর তা হলো char。একটি char কেবলমাত্র একটি ক্যারেক্টার (character) রাখতে পারে। একটি নাম কিন্তু অনেক গুলো ক্যারেক্টারের সমন্বয়ে তৈরি হয়। যেমন, Rajshahi শব্দে ৪টি character আছে। সেজন্য আসলে কোনো নাম বা স্ট্রিং (string) সংরক্ষণের জন্য আমাদের char এর অ্যারে ব্যবহার করতে হবে। যেমন, আমরা যদি একটি char city[10] নামে একটি অ্যারে ডিক্লেয়ার করি এবং তাতে Rajshahi রাখি তাহলে $city[0] = R, city[1] = a, \dots, city[7] = i$ । সবই ঠিক আছে তবে এর সঙ্গে অতিরিক্ত একটি জিনিস থাকে তা হলো null (বা ০ কারণ null এর ASCII মান 0)। $city[8]$ এ এই null থাকে। null দেখে আমরা বুঝতে পারিয়ে, শব্দটা আসলে $city[0]$ থেকে শুরু করে কোন পর্যন্ত আছে। যখন আমরা city অ্যারেটি প্রিন্ট করব তখন সে $city[0]$ থেকে প্রিন্ট করা শুরু করবে যতক্ষণ না $city[8]$ এ এসে null পায়। আমরা যদি city অ্যারেতে রাখা নামটি প্রিন্ট করতে চাই তাহলে আমাদের লিখতে হবে: printf("%s", city) আর যদি আমরা কোনো শহরের নাম ইনপুট নিতে চাই তাহলে আমাদের লিখতে হবে: scanf("%s", city)। খেয়াল কর এখন আর আগের মতো ইনপুটের সময় & ব্যবহার করতে হচ্ছে না। তোমরা চাইলে এই null নিয়ে খেলা করতে পার, যেমন for লুপ চালিয়ে স্ট্রিংয়ের দৈর্ঘ্য (length) বের করা বা একটি শহরের নাম ইনপুট নিয়ে তার প্রথম ৩ অক্ষরকে প্রিন্ট করা ($city[3] = 0; printf("%s", city);$)।

এবার একটু কাহিনির পেছনের কাহিনি দেখি। আমরা যদিও বলছিয়ে $city[0]$ এ R আছে কিন্তু আসলে তা নেই। $city[0]$ এ আছে 82 (একটি int ভ্যারিয়েবলে $city[0]$ যদি রাখ বা %d দিয়ে যদি তুমি $city[0]$ কে প্রিন্ট কর তাহলে এই 82 দেখতে পাবে)। প্রতিটি ক্যারেক্টারের বদলে একটি করে মান থাকে, একে বলা হয় আসকি (ASCII) মান। www.lookuptables.com এ আসকি মানের একটি টেবিল দেওয়া আছে।^১ খেয়াল করলে দেখবে A হতে Z পর্যন্ত আসকি মানগুলো পরপর আছে এবং এরা হলো 65 হতে 90, a হতে z এর মানগুলো হচ্ছে 97 হতে 122 আর 0 হতে 9 এর মান হচ্ছে 48 হতে 57। মজার ব্যাপার হচ্ছে আমাদের এই আসকি মান আসলে মুখ্যত করার দরকার নেই। আমাদের যদি নেহায়েতই দরকার হয় তাহলে আমরা কোনো ক্যারেক্টারকে %d দিয়ে প্রিন্ট করলেই তার আসকি মান দেখতে পাব। আরও মজার ব্যাপার হচ্ছে আমাদের আসকি মান 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, কোনো একটি ক্যারেক্টার ধরা যাক ch বড় হাতের না ছোট হাতের, তাহলে আমরা কোড লিখব: if('a' <= ch && ch <= 'z') (single quotation এর মধ্যে কোন ক্যারেক্টার রাখলে তার আসকি মান পাওয়া যায়)। যদি শর্তটি সত্য হয় তাহলে আমরা বুঝে যাব যে এটি ছোট হাতের। আবার ধরা যাক, আমরা যদি জানি যে, ch ছোট হাতের এবং আমরা চাই একে বড় হাতের বানাতে হবে আমরা শুধু লিখব: ch = ch - 'a' + 'A'। আবার আমরা যদি ch এ থাকা digit কে একটা int ভ্যারিয়েবলে মান হিসাবে নিতে চাই, তাহলে আমরা লিখব: d = ch - '0'। অর্থাৎ আমরা আসকি মানের তুলনামূলক অবস্থান জেনেই

^১কপিরাইট জনিত কারনে টেবিলটা এখানে কপি করা হলো না।

অনেক কঠিন কঠিন কাজ করে ফেলতে পারি।

স্ট্রিংয়ের ইনপুট নিয়ে আরেকটু কথা বলা যাক। আমরা উপরে যেভাবে scanf দিয়ে ইনপুট নিয়েছি তাতে একটি সীমাবদ্ধতা আছে আর তা হলো: স্পেস যুক্ত স্ট্রিং ইনপুট নেওয়া যাবে না এভাবে। যেমন, আমরা যদি একটি বাক্য ইনপুট নিতে চাই "Facebook is a popular web media" এবং সেজন্য যদি scanf এ %s ব্যবহার করি তাহলে দেখব ওই অ্যারেতে কেবল Facebook শব্দটিই থাকবে। এর কারণ হলো, আমরা যখন scanf দিয়ে পড়া শুরু করি তখন সে প্রথম "non whitespace" ক্যারেক্টার খোঁজে, এবং ওখান থেকে সে পরবর্তী whitespace সে প্রথম "non whitespace" ক্যারেক্টার খোঁজে, এবং ওখান থেকে সে পরবর্তী whitespace ক্যারেক্টার পর্যন্ত পড়ে।^১ তুমি যদি একটি স্পেস যুক্ত বাক্য পড়তে চাও তাহলে এভাবে পড়তে হবে: gets(s). এখানে s হলো আমাদের char টাইপের অ্যারের নাম। gets প্রথম থেকে শুরু করে যতক্ষণ না একটি নিউ লাইন (new line) পাচ্ছে ততক্ষণ পড়তে থাকে। এখন এর ফলে, তুমি যদি একটি কোডে scanf এবং gets দুটিই একই সঙ্গে ব্যবহার করতে চাও, তখন তোমাকে সাবধান হতে হবে। ধর তোমার প্রোগ্রাম প্রথমে n ইনপুট নিবে যেটা হচ্ছে মানুষের সংখ্যা এবং এর পরে nটি বাক্যে তাদের নাম ইনপুট নিতে হবে। তুমি মনে কর n ইনপুট নিলে scanf দিয়ে আর পরের বাক্যগুলো ইনপুট নিলে gets দিয়ে, তাহলে তোমার প্রথম বাক্যটি ঠিক মতো ইনপুট হবে না। কারণ, scanf দিয়ে তুমি যখন n পড়েছ তখন সে n পড়া শেষ করেছে যখন একটি নিউ লাইন বা whitespace ক্যারেক্টার পেয়েছে এবং সে সেটা পড়েনি। এখন তুমি যদি gets দিয়ে ইনপুট নাও তাহলে প্রথমেই সে ওই নিউ লাইন পড়বে এবং ইনপুট পড়া শেষ করে দিবে। সুতরাং এক্ষেত্রে সমাধান হলো তুমি একটি ডামি (dummy) gets ব্যবহার করবে। অর্থাৎ, তুমি এমনি এমনি scanf এর পরে একটি gets ব্যবহার করবে। এরপর তুমি যদি লুপ চালিয়ে সবার নাম ইনপুট নাও তাহলে কোনই সমস্যা হবে না।

আচ্ছা এইয়ে তোমাদের বলা হলো, nটি বাক্য পড়তে হবে। তোমরা কি ভেবে দেখছ এতগুলো বাক্য কীভাবে রাখবে? খেয়াল কর, তোমার প্রত্যেক বাক্যের জন্য কিন্তু একটি অ্যারে দরকার। তাহলে n টি বাক্যের জন্য কী করবে? sentence1[100], sentence2[100] এভাবে 100টি অ্যারে ডিক্লেয়ার করবে? মোটেও না, যা করবে তা হলো অ্যারের অ্যারে! অর্থাৎ, sentence[20][100] এভাবে। এখানে sentence[0] এ থাকবে প্রথম বাক্য এরকম করে মোট 20টি বাক্য এখানে রাখা যাবে। আশা করি বুবাতেই পারছ, এটা শুধু স্ট্রিংয়ের জন্য না, int, double সব ক্ষেত্রেই এরকম করে dimension বাড়ানো যাবে, একে multidimension অ্যারে বলে। যেমন, তোমরা ম্যাট্রিক্স (matrix) এর জন্য 2 dimension অ্যারে ব্যবহার করতে পার।

এবার আরও একটি হেডার ফাইলের কথা বলা যাক। string.h- স্ট্রিং সম্পর্কিত ফাংশনগুলো এখানে আছে। এর কিছু গুরুত্বপূর্ণ ফাংশনগুলো হলো: strlen - কোনো একটি স্ট্রিংয়ের দৈর্ঘ্য দেয়, strcmp - দুটি স্ট্রিং দিলে সে বলে দেয় কোনটি ডিকশনারিতে আগে আসবে, একে lexicographical order বলে, strcat - দুটি স্ট্রিং জোড়া দেওয়ার জন্য, strcpy - স্ট্রিং কপি করার জন্য, memset - memory কে কোনো নির্দিষ্ট কিছু মান দিয়ে ভর্তি করার জন্য ইত্যাদি।

মোটামুটি এসব ফাংশনগুলোই সহজেই বোঝা যায়। তবে memset এ কিছু জটিল ব্যবার

^১স্পেস, নিউ লাইন (new line), ট্যাব এন্ডেলো হলো whitespace ক্যারেক্টার

আছে। মনে কর আমাদের কাছে a নামে একটি ইন্টিজারের অ্যারে আছে, আমরা এর সবগুলো উপাদানকে 0 দিয়ে ইনিশিয়ালাইজ করতে চাই। তাহলে লিখতে হবে: `memset(a, 0, sizeof(a))`। তোমরা a এর জায়গায় তোমাদের অ্যারের নাম দিবে শুধু, আর 0 এর স্থানে তোমরা যেই মান দিয়ে ইনিশিয়ালাইজ করতে চাও তা। কিন্তু আসলে, সব মানের জন্য এটা সঠিকভাবে কাজ করবে না। আমরা সাধারণত কোনো একটি অ্যারেকে 0 বা -1 দিয়ে ইনিশিয়ালাইজ করে থাকি, এই দুক্ষেত্রে আমাদের `memset` ঠিক মতো কাজ করবে, কিন্তু আমরা যদি 1 দিয়ে ইনিশিয়ালাইজ করতে চাই, তাহলে হবে না।^১

অনেক সময় জানার প্রয়োজন হয় যে আমাদের কোড কতখানি মেমোরী ব্যবহার করছে। যদি প্রোগ্রামে চার পাঁচটি ভ্যারিয়েবল থাকে তাহলে সেটি সাধারণত গায়ে লাগে না। কিন্তু যদি একটু বড় মাপের অ্যারে থাকে তাহলে অনেক সময় একটু সাবধান হতে হয়। যেমন বলা আছে তোমার মেমোরী লিমিট 256 মেগাবাইট। এখন তুমি মনে করলে 10^7 সাইজের একটি `int` এর অ্যারে নিবে। এটা কী তোমার দেয়া মেমোরী লিমিটে আঁটবে? হিসাব করে দেখতে হবে। হিসাব করার আগে আমাদের জানতে হবে কোন ডেটা টাইপ কতখানি মেমোরী নেয়। একটি `int` নেয় 4 বাইট, `long long` নেয় 8 বাইট, `double` নেয় 8 বাইট আর `char` নেয় 1 বাইট। আমি ঠিক নিশ্চিত না তবে আমার জানা মতে সাধারণত `bool` 1 বাইট জায়গা নিয়ে থাকে।^২ তাহলে 10^7 টি `int` জায়গা নেয় 4×10^7 বাইট। এখন আমরা জানি 1 kilogram হলো 1000gram অর্থাৎ kilo মানেই 1000, তাহলে কী 1000 বাইট মানে 1 কিলোবাইট? না। 2^{10} বা 1024 বাইট মানে হল 1 কিলোবাইট। একই ভাবে 2^{10} কিলোবাইট মানে 1 মেগাবাইট। এরকম করে গিগা, টেরা ইত্যাদিও। সূতরাং 4×10^7 বাইট মানে হল 38.14 মেগাবাইট। এভাবে তোমরা মেমোরী এর হিসাবনিকাশ করতে পারবে।

আরেকটা বিষয়, যদি কখনও দেখ যে তোমার প্রোগ্রাম run time error (RTE) খাচ্ছে তাহলে তোমার উচিত হবে অ্যারের সাইজ একবার দেখে নেওয়া। ধর তুমি ডিক্লেয়ার করেছ 10 সাইজের অ্যারে অথচ তুমি তোমার প্রোগ্রামে 12 তম স্থান ব্যবহার করছ, তাহলে RTE হবে। এরকম সবসময় হয় না অবশ্য। মনে কর তোমার এরকম দুইটি 10 সাইজের অ্যারে আছে A আর B। কম্পিউটার হয়তো এদের পর পর মেমোরী তে রেখেছে। এখন তুমি A এর 12 তম স্থান ব্যবহার করতে চাইছ। তাহলে এটি আসলে B এর দ্বিতীয় স্থান ব্যবহার করবে এবং কোন RTE হবে না। এটা যদি java হতো তাহলে তোমাকে সবসময় এক্সেপশন (exception) দিত এবং তুমি তা এক্সেপশন মেসেজ দেখেই বুঝে যেতে। কিন্তু C এর মত ল্যাঙ্গুয়েজে এটা সবসময় বোঝা যায় না।

শেষ করব একটু কঠিন জিনিস দিয়ে। মনে কর তুমি `gets` দিয়ে একটি বড় লাইন ইনপুট নিয়েছ। এখন তুমি চাইতেছ যে এই লাইনের প্রতিটি শব্দ আলাদা আলাদা করবা। যেমন: "Concrete Mathematics is a nice advanced math book" এই লাইন ইনপুট নিলে। এখন তুমি এখানের প্রতিটি শব্দ মানে "Concrete", "Mathematics", "is" এরকম করে

^১তোমরা যদি এ বিষয়ে আরও জানতে চাও তাহলে Topcoder এর <http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=integersReals> আর্টিকেলটি পড়ে দেখতে পার।

^২এজন্য অনেক সময় অপটিমাইজেশনের খাতিরে `int` এর অ্যারে ডিক্লেয়ার করে প্রতি `int` এ 32 টি 0-1 রাখা হয় আলাদা `bool` এর পরিবর্তে। তাহলে আমাদের মেমোরী এর প্রতিটি বিট ব্যবহার হবে (1 বাইট = 8 বিট)। তোমরা চাইলে `stl` এর `bitset` ব্যবহার করতে পার।

প্রতিটি শব্দকে আলাদা আলাদা করতে চাও। কেমনে করবে? একটি উপায় নিজে লুপ চালিয়ে এই কাজ করা। তোমরা যারা নতুন তাদেরকে আমি লুপ চালিয়ে নিজে নিজে করে দেখতে বলব। এতে আইডিয়া অনেক পরিষ্কার হয় এবং মনে কর তুমি এমন একটি ল্যাঙ্গুয়েজের খঙ্গে পড়লে যেখানে built in এরকম কোন ফাংশন নেই। তখন তুমি একটুও তয় না পেয়ে নিজে নিজে এই জিনিস করে ফেলতে পারবে। কিন্তু তুমি যদি শুরু থেকেই built in ফাংশনে অভ্যস্ত হয়ে যাও তাহলে এরকম কাজ করতে তোমার ঘাম বের হয়ে যাবে। যাই হোক আরেকটি উপায় হলো strtok ব্যবহার করা। তোমরা চাইলে <http://www.cplusplus.com/reference/cstring/strtok/> থেকে শিখে নিতে পার। তবে আমি এক্ষেত্রে stl ব্যবহার করি। কোড ২.১৩ এ আমি সাধারণত যেভাবে করি তা দেখালাম। এখানে আমি stl ব্যবহার করেছি। পাশে কমেন্ট দিয়ে একটু বোঝানৰ চেষ্টা করলাম কোন লাইনে কী হচ্ছে। এখন যদি ইনপুটে বিভিন্ন যতিচিহ্ন (কমা, সেমিকোলন ইত্যাদি) থাকে তাহলে আমি আগে লুপ চালিয়ে এসব যতিচিহ্ন কে স্পেস এ পরিণত করি। এর পর সেই একই কাজ।

কোড ২.১৩: sstream.cpp

```

1 #include<iostream>
2 #include<string>
3 using namespace std;
4
5 char line[100];
6 string s; // string from stl
7 gets(line); // input the line
8
9 // creates an istringstream from the input line
10 istringstream iS(line);
11 while (iS >> s)
12 {
13     // S is a word. Do whatever you want to do
14 }
```

অনুশীলনী

- একটি দশমিক সংখ্যাকে বাইনারিতে রূপান্তর কর।
- একটি অ্যারেতে বাইনারি সংখ্যা দেওয়া আছে, এর দশমিক মান বের কর।

- একটি অ্যারেতে অনেকগুলো সংখ্যা দেওয়া আছে তাদেরকে ছোট হতে বড় অনুসারে সাজাও। এভাবে সংখ্যাকে ছোট হতে বড় বা বড় হতে ছোট আকারে সাজানোকে সর্টিং (sorting) বলে।^১
- একটি অ্যারেতে অনেকগুলো 1 এবং 0 আছে। তোমাকে বলতে হবে সবচেয়ে বেশি কতগুলো 1 পরপর আছে।
- একটি অ্যারে আছে। তোমাকে অনেকগুলো প্রশ্ন করা হবে। প্রতিটি প্রশ্ন হবে এমন: অ্যারের i তম স্থান হতে j তম স্থানের যোগফল কত? যেহেতু প্রশ্ন অনেকগুলো তোমাকে উত্তরও দ্রুত দিতে হবে।^২
- একটি স্ট্রিংয়ের দৈর্ঘ্য বের কর (লাইব্রেরী ফাংশন ব্যবহার করে এবং ব্যবহার না করে)।
- একটি শব্দে ছোট হাতের অক্ষর ($a, b, \dots z$) এবং বড় হাতের অক্ষর ($A, B, \dots Z$) এর সংখ্যা নির্ণয় কর।
- দুটি স্ট্রিং জোড়া লাগাও। অর্থাৎ একটি স্ট্রিং যদি হয় water এবং অপর স্ট্রিং যদি হয় melon তাহলে তাদের জোড়া লাগিয়ে নতুন স্ট্রিং- watermelon বানাও।
- দুটি স্ট্রিং A এবং B দেওয়া আছে, বলতে হবে B সম্পূর্ণভাবে A এর ভেতরে আছে কি না। যেমন: $A = bangladesh$ এবং $B = desh$ হলে বলা যায় যে B , A এর ভেতরে আছে। B A এর ভেতরে কয়বার আছে তাও নির্ণয় করতে পারো। যেমন: aa শব্দটি aaa এর মধ্যে দুবার আছে।
- একটি বাক্যে শব্দের এর সংখ্যা নির্ণয় কর। শব্দগুলো একাধিক স্পেস দিয়ে আলাদা করা থাকতে পারে। তাই যদি স্পেস কয়টি আছে তা গুণে সমাধান করতে চাও তাহলে হবে না!
- দুটি স্ট্রিং দেওয়া আছে বলতে হবে কোনটি lexicographically smaller (লাইব্রেরী ফাংশন ব্যবহার করে এবং ব্যবহার না করে)।
- আমি একটি তারিখ 21/9/2013 এরকম ফরম্যাটে দেব। তোমাকে এই স্ট্রিং হতে দিন, মাস ও তারিখ আলাদা করতে হবে এবং তিনটি int ভ্যারিয়েবলে রাখতে হবে।

¹তোমরা দুটি for লুপ ব্যবহার করে খুব সহজেই sort করতে পার। প্রথমে অ্যারের 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 ওই সংখ্যাটা যেন হারিয়ে না যায়, সেজন্য তুমি যেখান থেকে সংখ্যাটি আনবে সেখানে গিয়ে এই সংখ্যাটি রেখে আসবে। একে swap করা বলে।

²তোমরা যদি মনে কর এ আর এমন কি! আমি প্রতিবার i হতে j পর্যন্ত for লুপ চালাব! না, এর থেকেও ভালো বুদ্ধি আছে, দেখো বের করতে পার কি না!

ঃ দুটি স্ট্রিং A এবং B দেওয়া আছে, বলতে হবে B A এর subsequence কি না। B A এর subsequence হবে যদি A থেকে কিছু অক্ষর মুছে ফেললে B পাওয়া যায়। যেমন: $A = \text{bangladesh}$ এবং $B = \text{bash}$ হলে বলা যায় যে B , A এর subsequence কিন্তু $B = \text{dash}$ কিন্তু subsequence হবে না।

কিছু OJ এর প্রবলেম: – 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 টাইম কমপ্লেক্সিটি (Time Complexity) এবং মেমোরী কমপ্লেক্সিটি (Memory Complexity)

প্রোগ্রামিং প্রতিযোগিতায় টাইম কমপ্লেক্সিটি (Time Complexity) এবং মেমোরী কমপ্লেক্সিটি (Memory Complexity) খুবই গুরুত্বপূর্ণ জিনিস। সহজ কথায় বলতে গেলে একটি প্রোগ্রাম যতখানি সময় নেয় বা যতখানি মেমোরী নেয় তাকেই টাইম কমপ্লেক্সিটি বা মেমোরী কমপ্লেক্সিটি বলে। তবে এই টাইম বা মেমোরী কিন্তু সেকেন্ড বা বাইটে মাপা হয় না। কেন?

খেয়াল করলে দেখবে দুবছর আগে বাজারে যত ভালো কম্পিউটার পাওয়া যেত এখন তার থেকে তের ভালো ক্ষমতার কম্পিউটার পাওয়া যায়। আগে যেই প্রোগ্রাম চলতে হয়তো 10 সেকেন্ড সময় নিত এখন সেই একই প্রোগ্রাম হয়তো 5 সেকেন্ড সময় নেয়। তোমরা হয়তো বলতে পার, কোনো একটি প্রোগ্রাম আগে যে পরিমাণ মেমোরী নিত এখনও তাই নেয়। ঠিক! কিন্তু একটি প্রোগ্রাম ধর $n = 100$ এর জন্য যে পরিমাণ মেমোরী বা সময় ব্যয় করে $n = 1000$ এর জন্য হয়তো অন্য পরিমাণ মেমোরী বা সময় ব্যয় করে, ঠিক 10 গুণ নাও হতে পারে। n সাইজের একটি $2D$ ম্যাট্রিক্সের জন্য কিন্তু n^2 মেমোরী দরকার হয়। সুতরাং এখন তুমি যদি $n = 100$ হতে $n = 1000$ এ মান বৃদ্ধি কর তাহলে মেমোরী কিন্তু 10 গুণ বাঢ়ছে না বরং 100 গুণ বাঢ়ছে। সুতরাং একটি সমাধান এর উপর ভিত্তি করে হিসাব করতে পারি। আমরা একই সমস্যার বিভিন্ন প্যারামিটার (parameter) করার জন্যও এই কমপ্লেক্সিটি ব্যাপারটি ব্যবহার করে থাকি। কিছু উদাহরণে আশা করি জিনিসটা আরও বেশি পরিষ্কার হবে।

আমরা লুপ এর সেকশনে একটি প্রবলেম নিয়ে কথা বলছিলাম: $1 + (1 + 2) + (1 + 2 + 3) + \dots + (1 + 2 + \dots + n)$. এখন তুমি যদি দুটি for লুপ চালিয়ে হিসাব কর, তাহলে তুমি সর্বমোট $1 + 2 + \dots + n = \frac{n^2 + n}{2}$ টা যোগ করবে। আমরা আমাদের বুকার সুবিধার্থে শুধু মাত্র n

এর সবচেয়ে বড় টার্মটি বিবেচনা করি। এখানে n এর দুটি টার্ম আছে, একটি n এবং আরেকটি n^2 এর টার্ম। আমরা শুধু n^2 এর টার্ম বিবেচনা করব, সুতরাং n এর টার্ম বাদ দিলে আমাদের থাকে: $n^2/2$. আমরা ধ্রুব সহগ (constant coefficient) ও বাদ দেই। সুতরাং আমরা যা পেলাম তা হলো: n^2 . এটাই আমাদের টাইম কমপ্লেক্সিটি। আমরা বলে থাকি, আমাদের এই অ্যালগরিদমটি $O(n^2)$ ^১. যদি $n = 1000$ হয়ে থাকে তাহলে, $O(n^2) = 10^6$ খুব আরামে 1 সেকেন্ডে চলবে, কিন্তু যদি $n = 10^6$ হয়? তাহলে এই কোড এক ঘণ্টাতেও শেষ হবে কিনা সন্দেহ আছে। তাহলে n এর মান বেশি হলে $O(n^2)$ অ্যালগরিদমে Time Limit Exceed (TLE) পাবে। আমরা কি এটাকে অপটিমাইজ করতে পারি? খেয়াল করলে দেখবে যে, আমরা i এর for লুপ 1 থেকে n পর্যন্ত যদি চালাই এবং প্রতিবার $1 + 2 + \dots + i$ এর মান আরও একটি for লুপ দিয়ে না বের করে যদি সুত্রের 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 মতো অনেক সময় লাগবে এই কোড চলতে। সেক্ষেত্রে আমাদের চেষ্টা করতে হবে অ্যালগরিদমের অর্ডার আরও কমানো যায় কি না। এবং আসলেই কমানো যায়:

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

অর্থাৎ আমরা এমন একটি সূত্র বের করেছি যা হিসাব করতে আমাদের কোনো লুপ লাগে না। শুধু কিছু যোগ আর গুণ করেই করে ফেলতে পারি, একে বলা হয় $O(1)$ অ্যালগরিদম।

এখন আসা যাক মেমোরী কমপ্লেক্সিটিতে। তোমরা আশা করি ফিবোনাচি সংখ্যার (Fibonacci Number) কথা শুনেছ। যারা শুনো নাই তাদের জন্য বলি, n তম ফিবোনাচি সংখ্যাকে F_n দিয়ে প্রকাশ করা হয়। এর মান:

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

^১এর উচ্চারণ order of n^2 .

আমরা একটি অ্যারের সাহায্যে খুব সহজেই এর কোডটা করতে পারি। $F[0] = 0$, $F[1] = 1$ এবং একটি for লুপ চালিয়ে $F[i] = F[i - 1] + F[i - 2]$. কিন্তু এখানে আমরা n সাইজের একটি অ্যারে ডিক্রেয়ার করছি। সুতরাং আমাদের মেমোরী কমপ্লেক্সিটি হলো $O(n)$. তবে আমরা কিন্তু খুব সহজেই কোনো অ্যারে ছাঢ়াই F_n হিসাব করে ফেলতে পারি। কোড ২.১৪ দেখলে বুঝবে আমরা মাত্র তিনি ভ্যারিয়েবল ব্যবহার করছি, সুতরাং এখানে আমাদের মেমোরী কমপ্লেক্সিটি $O(1)$.^১ যদিও টাইম কমপ্লেক্সিটি আগের মতো $O(n)$ ই রয়ে গেছে কিন্তু মেমোরী কমপ্লেক্সিটি $O(1)$ এ কমে অসেছে।^২

কোড ২.১৪: fibonacci.cpp

```

1 a = 0;
2 b = 1;
3 for(i = 2; i <= n; i++)
4 {
5     c = a + b;
6     a = b;
7     b = c;
8 }
9
10 printf("nth Fibonacci = %d\n", b);

```

২.৭ ফাংশন এবং রিকার্শন (Recursion)

ফাংশনকে তোমরা একটি ফ্যাক্টোরি হিসেবে কল্পনা করতে পার। একে বিভিন্ন কাঁচামাল দিবে, ভেতরে ভেতরে সে কিছু একটা করবে এবং কাজ শেষে তুমি তার ফলাফল পাবে। যেমন একটি জুস ফ্যাক্টোরিতে তুমি ফল দিবে, চিনি দিবে, পানি দিবে আরও নানান কিছু উপকরণ হিসেবে দিবে। তোমার জানার দরকার নেই ফ্যাক্টোরির ভেতরে কেমনে কী হচ্ছে। সেটা ফ্যাক্টোরি যে চালায় তার ছাড়াবে, আরেক মেশিন ফল থেকে রস বের করবে, একটা মেশিন আছে যে ফলের খোসা পরিমাণে মিশিয়ে জুস বানাবে, আরেক মেশিন সেই জুসগুলো পরিমাণ মতো করে প্যাকেটে ভরবে, এর পর আরেক মেশিন হয়তো প্যাকেটের গায়ে স্ট্রিলাগিয়ে দিবে। ব্যাস তোমার জুস তৈরি! তুমি এখন সেই জুসের প্যাকেট এনে খাওয়া শুরু করবে। তোমার কিন্তু জানার দরকার নেই ফ্যাক্টোরির

^১ $n = 0$ এ সঠিক উত্তর দিবে না।

^২তোমরা চেষ্টা করে দেখতে পার টাইম কমপ্লেক্সিটিকেও কমাতে পার কি না। পরবর্তী কোনো অধ্যায়ে আমরা টাইম কমপ্লেক্সিটিকেও কমাব।

ভেতরে কীভাবে কী হচ্ছে। একইভাবে ফাংশন হচ্ছে এমন একটি জিনিস যাকে তুমি কিছু দিবে সে হিসাব নিকাশ করে তোমাকে বলে দিবে যে তার কাজের উত্তর কী! যেমন, আমরা sqrt ফাংশন এর কথা শুনেছি। একে আমরা একটা সংখ্যা দেই বিনিময়ে সে আমাদের ওই সংখ্যার বর্গমূল বলে দেয়। আমরা কিন্তু জানি না, ভেতরে ভেতরে সে কীভাবে এই বর্গমূল নির্ণয় করছে।

C তে ফাংশনের মূলত 4টি অংশ আছে। প্রথমত, ফাংশনের প্যারামিটার, অর্থাৎ কাঁচামাল। আমরা ফাংশনকে কিছু মান দিব এবং বলব এই মান নিয়ে কাজ করতে। দ্বিতীয়ত, রিটার্ন টাইপ অর্থাৎ আমরা এই ফাংশন থেকে কী ধরনের জিনিস বের করব। তৃতীয়ত, প্রদত্ত প্যারামিটারের ভিত্তিতে প্রসেসিং করা এবং চতুর্থত প্রসেসিংয়ের ফলাফল রিটার্ন করা। যেমন মনে করি আমাদের বলা হলো কিছু ছাত্রের গ্রেড নির্ণয় করতে হবে। আমাদের তাদের নম্বর দেওয়া হবে, এই নম্বরের উপর ভিত্তি করে আমাদের গ্রেড নির্ণয় করতে হবে। আমরা তাহলে একটা ফাংশন লিখব যা প্যারামিটার হিসেবে পরীক্ষার নম্বর নিবে এবং if-else দিয়ে যাচাই করে সে ফলাফল হিসেবে গ্রেড পাঠিয়ে দিবে (কোড ২.১৫)।

কোড ২.১৫: grade.cpp

```

1 int grade (int marks)
2 {
3     if (marks >= 80) return 5;
4     else if (marks >= 60) return 4;
5     else if (marks >= 50) return 3;
6     else if (marks >= 40) return 2;
7     else if (marks >= 33) return 1;
8     else return 0;
9 }
```

আগে একটি অনুশীলনী হিসেবে LightOJ 1136 এই সমস্যাটি দেওয়া হয়েছিল এবং বলা হয়েছিল "A হতে B এর উত্তর বের না করে 0 হতে B এর উত্তর বের করে তা থেকে 0 হতে A - 1 এর উত্তর বিয়োগ করলে সমাধানটি সহজ হয়"। আমরা একই ধরনের দুটি জিনিস আলাদা আলাদা করে হিসাব না করে বরং একটি ফাংশন f লিখতে পারি যার প্যারামিটার হবে n এবং এই ফাংশনটি 0 হতে n পর্যন্ত এর জন্য উত্তর বের করবে। সুতরাং আমাদের উত্তর হবে: $f(B) - f(A - 1)$. অনেক সহজেই আমাদের সমস্যাটি সমাধান হয়ে যায়!

এবার আসা যাক রিকার্শন (Recursion) এ। কোনো এক অজানা কারণে রিকার্শন কে অনেকেই ভয় পায়! Mr Edsger Dijkstra এর একটি গল্প আছে:

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 Eindhoven, 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.

রিকার্শন আসলে কিছুই না, এটি হলো এমন একটি ফ্যাক্টোরি যা তার প্রসেসিংয়ের জন্য নিজেকেই ব্যবহার করে। যেমন ফিবোনাচি সংখ্যার ক্ষেত্রে, আমরা জানি, $F_n = F_{n-1} + F_{n-2}$. এখন আমরা যদি এমন একটি ফাংশন লিখি যেটা আমাদের n তম ফিবনাচি নম্বর দেয়, এবং সে সেই ফাংশনের ভেতরে হিসাবের জন্য $n - 1$ তম ও $n - 2$ তম ফিবনাচি নম্বরকে পাওয়ার জন্য নিজেকেই call করে তাহলে একে রিকার্শন বলা হয়। তবে রিকার্শন ফাংশন call কিন্তু এক পর্যায়ে শেষ হতে হবে, না হলে কিন্তু এই call চলতেই থাকবে। আমরা যদি, $n = 3$ তম ফিবোনাচি বের করতে চাই, তাহলে এটার মান বের করার জন্য সে $n - 1 = 2$ তম এবং $n - 2 = 1$ তম ফিবোনাচি নম্বর চাইবে, তারা ভেতরে গিয়ে আবার তার থেকে ছোট দুটি ফিবোনাচি চাইবে এভাবে কিন্তু চলতে থাকবে। তাহলে এই অবস্থা থেকে মুক্তি কী? মুক্তি ফিবোনাচি সংখ্যার সংজ্ঞাতেই 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, $n \geq 2$ এর ক্ষেত্রেই কেবল এরকম $n - 1$ তম ও $n - 2$ তম ফিবোনাচি সংখ্যা দরকার হবে, অন্যথায় কী মান হবে তা বলা আছে। রিকার্শন ব্যবহার করে আমাদের ফিবোনাচি সংখ্যা নির্ণয়ের প্রোগ্রামটা কোড ২.১৬ তে দেওয়া হলো। মজার ব্যাপার হলো, আমরা এই রিকার্সিভ ফাংশন (recursive function) এর মাধ্যমে বড় n এর জন্য ফিবোনাচি সংখ্যার মান বের করতে পারব না (অনেক সময় লাগবে) কিন্তু লুপের সাহায্যে অনেক দূর পর্যন্ত খুব সহজেই বের করা যাবে। এর কারণ কী? তোমাদের ইতোমধ্যেই কিন্তু টাইম কমপ্লেক্সিটি নিয়ে বলেছি, তোমরা দুটি অ্যালগরিদমের টাইম কমপ্লেক্সিটি বের করার চেষ্টা করে দেখতে পার। তাহলেই বুঝবে এমন কেন হলো। যদি না বের করতে পার, চিন্তার কিছুই নেই, আমরা পরবর্তী এক অধ্যায়ে এটা নিয়ে আরও দেখব।

কোড ২.১৬: recursive fibonacci.cpp

```

1 int fibonacci(int n)
2 {
3     if(n == 0) return 0;
4     if(n == 1) return 1;

```

```
    ৫     return fibonacci(n - 1) + fibonacci(n - 2);  
  ৬ }
```

এই সেকশনের প্র্যাকটিস প্রবলেমগুলো একটু কঠিন। তোমরা ইতোমধ্যে অ্যারেও শিখে ফেলেছ। সেই অ্যারে এবং রিকার্শন ব্যবহার করে সলভ করার মতো কিছু প্রবলেম দেয়া হচ্ছে:

অনুশীলনী

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

২.৮ ফাইল (File) ও স্ট্রাকচার (Structure)

আমরা যখন কোড করি তখন দেখা যায় কোডে ভুল হয়, ভুল সংশোধন করে আবার আমরা চেক করে দেখি ঠিক উভয় আসে কি না। এজন্য আমরা সমস্যায় দেওয়া স্যাম্পল টেস্ট দিয়ে যাচাই করে থাকি বা আমাদের নিজেদের কোনো কেইস (case) দিয়ে যাচাই করে থাকি। কিন্তু বার বার সেই কেইস লেখা খুবই বিরক্তিকর কাজ। যদি কেইসটি অনেক বড় হয় তাহলে তো কোনো কথাই নেই। মাঝে মধ্যে কোনো কোনো OJ তে ফাইলে ইনপুট আউটপুট করতে হয়। সুতরাং আমাদের ফাইলের কাজও সামান্য জানতে হবে। ফাইলে 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` ফাইলে আউটপুট দিতে হবে। আমরা যা করব তা হলো, পুরো কোডটা সাধারণভাবেই লিখব তবে কোডের শুরুতে (`মেইন ফাংশনের ভেতরে ঢুকেই`) দুটি লাইন লিখব: `freopen("input.txt", "r", stdin);` এবং `freopen("output.txt", "w", stdout);`. তোমরা তোমাদের দরকার মতো ফাইল নাম বসিয়ে নিবে তাহলেই হবে। এবার যদি তোমার প্রোগ্রাম রান কর তাহলে দেখবে `input.txt` থেকে ইনপুট পড়ছে আর `output.txt` তে আউটপুট লিখছে। আরও একটি উপায় আছে আর তা হলো `fopen` ফাংশনের মাধ্যমে। বেশির ভাগ বইয়ে আসলে `fopen` নিয়ে লেখা থাকে। কিন্তু তাতে তোমাকে ইনপুট আউটপুটের জন্য ব্যবহার করা সব ফাংশন পরিবর্তন করতে হবে। `scanf` এর পরিবর্তে লেখতে হবে `fscanf`, `printf` এর পরিবর্তে `fprintf` এরকম। কিন্তু এটা বেশ বিরক্তিকর। এছাড়া কেউ যদি `terminal` এ কম্পাইল ও রান করে থাকে তারা আরও সহজে ফাইল হতে ইনপুট ও আউটপুট করতে পারে। আমরা `terminal` হতে কম্পাইল করার জন্য লেখি `"g++ code.cpp"`. এরপর রান করার জন্য লেখি `"/a.out"` (লিনাক্স অপারেটিং সিস্টেমে) বা `"a.exe"` (উইন্ডোজ অপারেটিং সিস্টেমে)। আমরা যদি এই রান করার সময় লেখি `"a.exe <input.txt>output.txt"` তাহলে আমাদের ইনপুট আর আউটপুটের কাজ হয়ে যাবে। যদি শুধু ইনপুট নিতে চাও তাহলে আউটপুটের অংশটুকু মুছে ফেললেই হবে।

এখন একটু স্ট্রাকচার নিয়ে দেখা যাক। অনেকগুলো একই ধরনের জিনিস একত্রে রাখার জন্য আমরা অ্যারে ব্যবহার করে থাকি। কিন্তু অনেক সময় আমাদের অনেকগুলো বিভিন্ন জিনিস একত্রে

রাখার প্রয়োজন হতে পারে। যেমন, একটি ছাত্রের তথ্য: তার নাম, পিতার নাম, ঠিকানা, জন্মসাল, ফোন নম্বর ইত্যাদি বিভিন্ন তথ্য রাখতে হবে। আমরা যা করতে পারি তা হলো বিভিন্ন তথ্যের জন্য আলাদা আলাদা অ্যারে। কিন্তু এতে করে এটি নিয়ে কাজ করা বেশ সমস্যা হয়ে যায়। যেমন মনে কর আমরা চাচ্ছি যে একজন ছাত্রের সব তথ্য একটি ফাংশনে পাঠাব। এখন যদি আমাদের তথ্যগুলি আলাদা আলাদা অ্যারেতে থাকে তাহলে পাঠানো বেশ ঝামেলা। এর থেকে ভাল উপায় হলো স্ট্রাকচার। এটি এমন একটি জিনিস যেখানে আমরা একই সঙ্গে বিভিন্ন জিনিস একত্রে রাখতে পারি। কোড ২.১৭ এ আমাদের এই উদাহরণটি তুলে ধরা হলো।

কোড ২.১৭: 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 scanf ("%d", &student[10].phone);
12 printf ("%d\n", student[5].birth_date);

```

এখানে প্রথমেই আমরা **Student** নামে একটি স্ট্রাকচার ডিক্লেয়ার করেছি যার মধ্যে আমাদের প্রয়োজনীয় সব ভ্যারিয়েবল ডিক্লেয়ার করা হয়েছে। এখন এই **Student** নামটা এক রকমের ডেটা টাইপ হিসেবে ব্যবহার করা যাবে। কোডটিতে খেয়াল কর একটি ভ্যারিয়েবল **s** এবং একটি **Student** টাইপের অ্যারে **student** তৈরি করা হয়েছে। এখন ভ্যারিয়েবলের সঙ্গে **dot (.)** দিয়ে আমরা এর বিভিন্ন ভ্যারিয়েবলগুলো অ্যাক্সেস (access) করতে পারব। কোডটিতে নাম ও ফোন নম্বরে ইনপুট নেওয়া বা জন্মতারিখ আউটপুট দেওয়া দেখানো হয়েছে।

২.৯ বিটওয়াইজ অপারেশন (bitwise operation)

আমরা যেভাবে হিসাব নিকাশ করি বা সংখ্যা লেখি একে দশমিক সংখ্যা বা **Decimal Number System** বলে। আমাদের সর্বমোট 10 টি অঙ্ক আছে: 0, 1, 2, ..., 9. কিন্তু আমাদের **System** বলা হয়। বাইনারিতে প্রতিটি অঙ্ককে বিট বলা হয়। আমরা যখন একটি ভ্যারিয়েবলে 6

যাখি তখন আসলে সেখানে 0, 1 দিয়ে তৈরি একটি সংখ্যা থাকে।

বিটওয়াইজ অপারেশন হচ্ছে এমন কিছু অপারেশন যা সরাসরি বিট নিয়ে কাজ করে। আমরা যেসব বিটওয়াইজ অপারেটর সচরাচর ব্যবহার করে থাকি সেসব হল: & (bitwise and), | (bitwise or), ^ (bitwise xor), ~ (1's complement), << (shift left), >> (shift right).

<< হল left shift. আমরা জানি 5 হল বাইনারিতে 101. এখন একে 3 ঘর বামে শিফট (shift) করা মানে হলো এর ডানে 3টি 0 বসানো অর্থাৎ 101000. আর বাইনারিতে ডানে 0 বসানো মানে হলো দশমিকে 2 দিয়ে গুণ করা। অর্থাৎ এখানে আসলে আমরা 5 কে 2^3 বা 8 দিয়ে গুণ করছি। অন্যদিকে right shift হল কিছুটা left shift এর মতোই তবে পার্থক্য হলো এটি আসলে ডান দিকে যায়। যেমন 101 (বাইনারি) কে আমরা যদি ডানে এক ঘর সরাই তা হলে এটি হবে 10. দুই ঘর সরালে 1. তিন বা এর বেশি ঘর সরালে হবে 0. খেয়াল কর এগুলো কিন্তু প্রতিটিই বাইনারিতে লেখা। এখন কাহিনি হল আমার জানা মতে $5 * 8$ লেখার থেকে 5 << 3 লিখলে দ্রুত কাজ করে। বা একইভাবে $5 / 2$ লেখার থেকে 5 >> 1 দ্রুত কাজ করে। কিন্তু তাও আমরা তা সাধারণত লেখি না। কারণ সাধারণত প্রোগ্রামিং প্রতিযোগিতায় এ এতো low level এ আমাদের অপটিমাইজ করার প্রয়োজন হয় না। আমরা এসব ব্যবহার করি যখন আমরা বিটমাস্ক (bitmask) নিয়ে কাজ করি। মানে ধর আমাদের n টি জিনিস আছে। এর ভেতরে কে কে আছে কে কে নাই এটা বুঝাতে আমরা n টি 0 – 1 ব্যবহার করতে পারি। আর এই n যদি খুব ছোট ধর 20 এর মতো হয় তাহলে আমরা int এর 32 বিটের শেষ 20 বিট ব্যবহার করে এই আছে কি নাই জিনিসটা প্রকাশ করতে পারি। এই আছে কি নাই state টি manipulate করার জন্য আমরা left shift, right shift এবং সেই সঙ্গে bitwise and, or, xor এসব ব্যবহার করে থাকি। আরও বেশি কিছু বলার আগে চল আমরা and, or এসব সম্পর্কে জেনে নেই।

লজিকাল অপারেটরের মতো and এর মান 1 হবে যদি উভয়ের মানই 1 হয়, or এর মান 1 হবে যদি দুটির যেকোনো একটির মান 1 হয়, ~ এর মান 1 হয় যদি ওই বিটের মান 0 হয় (~ কিন্তু একটি সংখ্যার উপর হয়, অর্থাৎ এটি ইউনারি অপারেটর বা unary operator), xor এর মান 1 হয় যদি দুটির একটির মান 1 হয় (উভয়েই 1 হলে কিন্তু হবে না)। যেমন $1010 \& 1100 = 1000$, $1010 | 1100 = 1110$, $1010 ^ 1100 = 0110$. আর যদি 1100 এর ~ করতে হয় তাহলে ব্যপারটা নির্ভর করবে 1100 এর ডেটা টাইপের উপর। যদি int হয় তাহলে 28 টি 1 এর পর 0011. Long long হলে 60টি 1 এর পর। তবে আমি পারতপক্ষে ~ ব্যবহার করি না। কারণ সত্যি কথা বলতে এই যে বললাম 28 টি 1 হয় এটা আমি থিওরীর উপর ভিত্তি করে বলেছি। বাস্তবে আসলেই তাই হয় কি না তা আমার জানা নেই। কারণ আমার মন বলে যে signed বা unsigned ভেদে জিনিসটা আলাদা হবে (নাও হতে পারে)। আমি চাইলে বই লেখার সময়ে দেখে নিশ্চিত হয়ে নিতে পারি। কিন্তু কী দরকার? এটা না ব্যবহার করেই আমার কাজ চলে যায়। কনটেন্সের সময় শুধু শুধু ঝুঁকি নেবার কী দরকার আছে?

যাই হোক, এখন বলি বিটওয়াইজ অপারেশনের কিছু বহুল প্রচলিত ব্যবহার নিয়ে। মনে কর তুমি বের করতে চাও যে, তম বিট on নাকি off. তাহলে তোমাকে দেখতে হবে ($x \& (1 << i)$) এটি কী 0 কি না। যদি তুমি তম বিট on করতে চাও তাহলে ($x | (1 << i)$) করতে হবে। যদি

toggle করতে চাও তাহলে xor. যদি তম বিট off করতে চাও তাহলে আমি যা করি তা হলো
প্রথমে দেখি যে তম বিট on আছে কি না। যদি on থাকে তাহলে toggle করি। আর যদি off
থাকে তাহলে তো কিছু করার দরকার নেই। আরেকটি জিনিস প্রায়ই কাজে লাগে আর তা হলো শেষ
`k`টি বিট যেমন আছে তেমন রেখে বাকিগুলো off করে দেওয়া। সেক্ষেত্রে ($x \& ((1 << k) - 1)$)
করলেই হয়। কেন? কারণ $(1 << k)$ হলো একের ডানে `k`টি শূন্য। $((1 << k) - 1)$ মানে
হলো `k`টি এক (যেমন বাইনারিতে $1000 - 1 = 111$)। সুতরাং এটি দিয়ে যদি and করি তাহলে
আমার শেষের `k`টি ঘর যেমন ছিল তেমন থাকবে, বাকি সবাই শূন্য হয়ে যাবে। মোটামোটি এই সব
কাজ করতে পারলেই তুমি বিট দিয়ে যাবতীয় সব কাজ করতে পারবে। এসব জিনিস কাজে লাগে
মূলত দুটি জায়গায়। একটি হলো বিটমাস্ক ডায়নামিক প্রোগ্রামিং (dynamic programming)
আরেকটি হলো ছোট n এর জন্য ইনক্লুশন এক্সক্লুশন (inclusion exclusion) করার সময়।

অধ্যায় ৩

গণিত

৩.১ সংখ্যাতত্ত্ব (Number Theory)

৩.১.১ মৌলিক সংখ্যা (Prime Number)

Prime Number কে বাংলায় মৌলিক সংখ্যা বলে। একটি সংখ্যা n কে মৌলিক বলা হয় যদি ওই সংখ্যাটি ১ এর থেকে বড় হয় এবং ১ বা n ছাড়া আর কোনো ধনাত্মক সংখ্যা দিয়ে বিভাজ্য না হয়। এখন যদি একটি সংখ্যা n দিয়ে বলা হয় এটি মৌলিক কি না- তুমি কীভাবে সমাধান করবে? মোটামুটি সংজ্ঞা থেকেই বোঝা যায় কীভাবে করা উচিত। অবশ্যই n এর থেকে কোনো বড় সংখ্যা দিয়ে n কে ভাগ করা যায় না। সুতরাং যদি 2 হতে $n - 1$ এর মাঝের কোনো একটি সংখ্যা দিয়ে n নিঃশেষে বিভাজ্য হয় তাহলে n মৌলিক নয়। সুতরাং আমরা এই আইডিয়ার উপর ভিত্তি করে যদি মৌলিকত্ব (primality) যাচাই করার জন্য একটি ফাংশন লিখি তা দাঁড়াবে কোড ৩.১ এর মতো।

কোড ৩.১: 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     return 1;
9 }
```

এখন এর টাইম কমপ্লেক্সিটি কত? Worst কেইস অর্থাৎ কোডটি সবচেয়ে বেশি সময় নিবে যদি এটি মৌলিক হয়। সেক্ষেত্রে for লুপটি $n - 2$ সংখ্যকবার চলবে, সুতরাং এর টাইম কমপ্লেক্সিটি $O(n)$ । এখন আমরা ধীরে ধীরে এই টাইম কমপ্লেক্সিটিকে কমানোর চেষ্টা করব। কীভাবে? তোমরা ভাবতে পার, আচ্ছা আমরা তো জানি, 2 বাদে কোনো জোড় সংখ্যা মৌলিক নয়। সুতরাং for লুপটি তো শুধু বিজোড় সংখ্যার উপর দিয়ে চালালেই হয়! ভালো বুদ্ধি। তাহলে আমাদের রানটাইম (বা টাইম কমপ্লেক্সিটি) কত হবে? $O(n/2)$ আর আমরা বলেছি আমরা সব প্রক্রিয়াক সহগ (constant coefficient) পদকে বাদ দেই। তাহলে এভাবে করলেও আমাদের রানটাইম $O(n)$ ই থাকে। হ্যাঁ, অর্ধেক হবে কিন্তু এটি আমাদের অর্ডার নোটেশনে কোনো ব্যাপারই না। তাহলে আমরা কীভাবে কমাবো? একটু চিন্তা করলে দেখবে যে, যদি এমন কোনো d খুঁজে পাওয়া n কে ভাগ করে তাহলে তুমি আরও একটি সংখ্যা কিন্তু খুঁজে বের করে ফেলেছ যেটা n কে ভাগ করে: n/d . অর্থাৎ কোনো একটি সংখ্যার গুণনীয়ক (divisor) গুলো সবসময় জোড়ায় জোড়ায় থাকে। যেমন $n = 24$ হলে এর গুণনীয়কগুলো হচ্ছে: 1, 2, 3, 4, 6, 8, 12, 24 এবং তারা ৫টি জোড়ায় আছে: (1, 24), (2, 12), (3, 8), (4, 6). একটু চিন্তা করলে দেখবে প্রতিটি জোড়ার ছোটটি সবসময় $\leq \sqrt{n}$ হবে। কেন? এটি সরাসরি প্রমাণ করা মনে হয় একটু কঠিন হবে, কিন্তু proof by contradiction এ কিন্তু খুবই সোজা। মনে কর ছোটটি \sqrt{n} এর থেকেও বড়, তাহলে ওই জোড়ার বড়টি তো বড় হবেই! আর জোড়াগুলো এমনভাবে বানানো হয়েছে যেন তাদের গুণফল n হয়। কিন্তু দুটি \sqrt{n} এর থেকে বড় সংখ্যার গুণফল কীভাবে n হয়? অতএব জোড়ার ছোটটিকে অবশ্যই \sqrt{n} এর সমান বা ছোট হতে হবে। অন্যভাবে বলা যায় যে যদি কোন n মৌলিক না হয় তাহলে তার \sqrt{n} এর থেকে ছোট একটি গুণনীয়ক থাকবেই। সুতরাং আমাদের আর কষ্ট করে $n - 1$ পর্যন্ত লুপ না চালালেও হবে, আমরা চাইলে \sqrt{n} পর্যন্ত লুপ চালিয়েই বলে দিতে পারি যে n মৌলিক কিনা। এভাবে আমরা যদি কোড করি (কোড ৩.২) তাহলে আমাদের রানটাইম হবে $O(\sqrt{n})$. এখানে খেয়াল করতে পার যে আমরা আমাদের for লুপের শর্তটি $i * i \leq n$ লিখেছি, $i \leq \text{sqrt}(n)$ নয়। এর কিছু কারণ আছে। প্রথমত, বার বার বর্গমূল হিসাব করা একটি ব্যয়বহুল কাজ(কম্পিউটারের টাইম কমপ্লেক্সিটির দ্রষ্টিকোণ থেকে)। দ্বিতীয়ত, double ডেটাটাইপ ব্যবহার করলে কিন্তু precision loss হয়। এর ফলে $\text{sqrt}(9) = 3$ না হয়ে 2.9999999 বা 3.0000001 হলেও অবাক হওয়ার কিছু নেই।^১ কিন্তু বার বার $i * i$ করাও কেমন যেনো! তোমরা যা করতে পার তা হলো লুপ শুরু হওয়ার আগেই $\text{limit} = \text{sqrt}(n + 1)$ করে নিতে পার। এর পর এই পর্যন্ত লুপ চালাবে। তাহলে $\text{sqrt}(n)$ না করে $\text{sqrt}(n + 1)$ করলাম? সাবধানতার জন্য, যাতে কোন রকম precision loss বিশেষ কেইস (corner case) গুলো দিয়ে একটু পরীক্ষা করে দেখা উচিত। যেমন 1, 2, 3 ইত্যাদি ছোট মানের জন্য তোমার ফাংশন সঠিক উত্তর দেয় কি না।

কোড ৩.২: isPrime2.cpp

^১মজার ব্যপার হলো sqrt ফাংশন পূর্ণসংখ্যার উত্তরের ক্ষেত্রে সবসময় সঠিক উত্তর দেয় (আমার জানা মতে)!

```

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 * i <= n; i++)
6         if (n % i == 0)
7             return 0;
8     return 1;
9 }

```

আরও কি উন্নতি করা যাবে রানটাইম? হ্যাঁ যাবে, আসলে এটি $O(\log n)$ সময়েই করা যাবে! কারণ যদি এতে আগ্রহ থাকে তাহলে ইন্টারনেটে খোঁজ করে দেখতে পার।

সীভ অব ইরাটোসথেন্স (Sieve of Eratosthenes)

এটি মৌলিক সংখ্যা কে বের করার একটি দ্রুত উপায়। তোমরা লক্ষ করলে দেখবে যে আগের $O(\sqrt{n})$ অ্যালগরিদমে আমরা যা করেছি তা হলো কোনো একটি সংখ্যা নিয়ে তা মৌলিক কি না সেটা বের করেছি। কিন্তু এর ফলে যা হয় তা হলো, একটি সংখ্যা মৌলিক কি না তা যাচাই করার জন্য অনেক সংখ্যা দিয়ে ভাগ করে করে দেখতে হয়। তবে এই কাজটা যদি আমরা ঘুরিয়ে করি? অর্থাৎ কোনো 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 শুধুমাত্র ওইসব সংখ্যার জোড়া নিছি যারা একে অপরকে ভাগ করে। সীভ (Sieve) এর অ্যালগরিদমে ঠিক এই কাজটিই করা হয়। এর মাধ্যমে তোমরা 1 হতে n এর মধ্যের সব মৌলিক সংখ্যা বের করে ফেলতে পারবে। শুধু তাই না, এই সীমার মধ্যের কোনো সংখ্যা দিলে সেটা মৌলিক কি না সেটাও অনেক দ্রুত বলে দিতে পারবে। অ্যালগরিদমটি কিন্তু খুবই সহজ! তুমি 2 হতে n পর্যন্ত সব সংখ্যা লিখ, এরপর প্রথম থেকে শুরু করো, একটি করে সংখ্যা নিবে আর তার থেকে বড় তার যতগুলো গুণিতক (multiple) এখনও কাটা হয় নাই তাদের কেটে ফেল! এভাবে একে একে সব সংখ্যা নিয়ে কাজ করলে তোমার কাছে যেসব সংখ্যা অবশিষ্ট থাকবে সেগুলোই হলো মৌলিক এবং এর বাইরে 1 হতে n এর মাঝে কিন্তু আর কোনো মৌলিক সংখ্যা নেই! তুমি কিন্তু এই কাটাকুটির কাজটা \sqrt{n} পর্যন্তও করতে পার! অর্থাৎ তুমি যে একটি করে সংখ্যা নিয়েছিলে (ধরা যাক i) আর তার সব গুণিতক কেটে ফেলেছো, এই কাজটা $i > \sqrt{n}$ এর জন্য করার দরকার নেই। কারণ এরকম সংখ্যার যতগুলি গুণিতক আছে 1 হতে n এর ভিতরে তারা সবাই ইতিমধ্যে কাটা হয়ে গেছে। এমনকি তোমার আসলে যে কোনো i এর জন্য $2i, 3i, \dots$ এরকম সব গুণিতক কাটার দরকার নেই। কারণ $i * i$ এর থেকে ছোট i এর সব গুণিতক কিন্তু ইতিমধ্যেই কেটে ফেলেছ $2, 3, \dots$ দিয়ে। $n = 10$ এর জন্য আমরা টেবিল 3.1 এ এই অ্যালগরিদমটি সিমুলেশন (simulation) করে দেখালাম।

তোমাদের সুবিধার জন্য এর কোড 3.3 এ দেওয়া হলো। এই অ্যালগরিদমের রানটাইম

টেবিল ৩.১: $n = 10$ এর জন্য সীভ অ্যালগরিদমের সিমুলেশন

বিবরণ	বর্তমান অবস্থা
শুরুর অবস্থা	2, 3, 4, 5, 6, 7, 8, 9, 10
প্রথম সংখ্যা = 2. আমরা 2 এর চেয়ে বড় 2 এর সব গুণিতক কেটে দেব	2, 3, 2 , 5, 2 , 7, 2 , 9, 2
পরবর্তী সংখ্যা = 3. আমরা 3 এর চেয়ে বড় 3 এর সব গুণিতক কেটে দেব	2, 3 , 2 , 5, 2 , 7, 3 , 2
আর দরকার নেই, কারণ $5 > \sqrt{10}$	2, 3, 2 , 5, 2 , 7, 3 , 2

$O(n \log \log n)$. কেন? প্রমাণটা বেশ কঠিন। আগ্রহ থাকলে ইন্টারনেট তো আছেই! কোড ৩.৩ এ সীভ ফাংশন শেষে তোমরা একটি মৌলিক সংখ্যার তালিকা পাবে এবং mark অ্যারে থেকে বলতে পারবে কোনটি মৌলিক আর কোনটি মৌলিক নয়।

কোড ৩.৩: sieve.cpp

```

1 // I prefer vector
2 int Prime[300000], nPrime;
3 // 1 if not prime, 0 if prime
4 int mark[1000002];
5
6 void sieve(int n)
7 {
8     int i, j, limit = sqrt(n * 1.) + 2;
9
10    // 1 is not prime. you can also mark 0
11    mark[1] = 1;
12    // almost all the evens are not prime
13    for (i = 4; i <= n; i += 2) mark[i] = 1;
14
15    // 2 is prime
16    Prime[nPrime++] = 2;
17    // run loop for only odds
18    for (i = 3; i <= n; i += 2)
19        // if not prime, no need to do "multiple cutting"
20        if (!mark[i])

```

```

21
22         // i is prime
23         Prime[nPrime++] = i;
24
25         // if we don't do it, following
26         // i * i may overflow
27         if (i <= limit)
28         {
29             // loop through all odd multiples of i
30             // greater than i * i
31             for (j = i * i; j <= n; j += i * 2)
32             {
33                 // mark j not prime
34                 mark[j] = 1;
35             }
36         }
37     }
38 }

```

কোড ৩.৩ এ আমরা আসলে এক সঙ্গে অনেক কিছু অপটিমাইজ করেছি। প্রথমত ২ ছাড়া যেহেতু কোনো জোড় মৌলিক সংখ্যা নেই তাই আমরা যখন লুপ চালাবো তখন শুধু বিজোড় সংখ্যার উপর লুপ চালাবো (লাইন 18)। আর ২ দিয়ে তো কেবল অন্য সব জোড় সংখ্যা কাটা যায়, সেই কাটার কাজ আমরা আলাদা একটি লুপে করে ফেলেছি (লাইন 13)। এটা যদি না করতাম তাহলে শুধু শুধু জোড় সংখ্যার উপর দিয়ে লুপ চলবে। শুধু বলছি কারণ ২ বাদে বাকি জোড় সংখ্যার জন্য কোড পরের if কে (লাইন 20) অতিক্রম করতে পারবে না। এই অপটিমাইজেশন না করলে খুব একটা মনে হয় যায় আসে না। যাই হোক, যদি আমরা দেখি i কাটা যায়নি (লাইন 20) তাহলে আমরা এর গুণিতকগুলোর উপর লুপ চালিয়ে তাদের কেটে ফেলি (লাইন 31 – 35)। আর কাটা যায়নি মানে হলো সে মৌলিক এবং তাকে আমরা মৌলিক সংখ্যার অ্যারেতে ঢুকিয়ে দেই (লাইন 23)। অর্থাৎ আমরা বলতে পারি আমরা কেবল মৌলিক সংখ্যা দিয়েই কাটার কাজ করব, মৌলিক নয় এরকম কোনো সংখ্যাতে আসার আগেই সেটি অন্য কাউকে দিয়ে কাটা হয়ে গেছে! একটা উদাহরণ দেওয়া যাক। ধর কোনো একটি সংখ্যা হলো $p_1 \times p_2$. এখন p_1 আর p_2 এই দুটি মৌলিক কিন্তু আমাদের $p_1 \times p_2$ এর থেকে ছোট। তাই $p_1 \times p_2$ কিন্তু p_1 বা p_2 এর গুণিতক যখন কেটেছি তখনই কেটে ফেলেছি। তাই ভেতরের লুপ কেবলমাত্র মৌলিক সংখ্যার ক্ষেত্রেই চলবে। আবার মনে কর আমরা এখন একটি মৌলিক সংখ্যা p_1 এ আছি। $p_1 \times p_1$ এর থেকে ছোট সব p_1 এর গুণিতকের কিন্তু অবশ্যই p_1 এর থেকে ছোট একটি মৌলিক উৎপাদক আছে। তাই আসলে $p_1 \times p_1$ এর থেকে ছোট গুণিতকদের কাটার দরকার নেই। কাটার কাজ আমরা $p_1 \times p_1$ থেকে শুরু করতে পারি। মনে কর p_1 এর একটি গুণিতক হলো a ,

তাহলে এর পরের গুণিতক হবে $a + p_1$. কিন্তু আমাদের আসলে জোড় সংখ্যাগুলোকে যাচাই করার দরকার নেই তাই আমরা $p_1 \times p_1$ থেকে শুরু করে $2p_1$ দূরে দূরে গিয়ে গুণিতক কাটলেই পারি। এভাবে সব কিছু কাটা হয়ে গেলে আমরা সব মৌলিক সংখ্যা পেয়ে যাব। আমরা চাইলে যেসব সংখ্যার জন্য if এর শর্তটি সত্য হয় তাদেরকে একটি অ্যারে বা ভেক্টর (vector) এ ঢুকিয়ে রাখতে পারি। তাহলে আমরা 1 হতে n এর মাঝের সকল মৌলিক সংখ্যার একটি তালিকা পেয়ে যাব (যেমন আমাদের কোডে Prime অ্যারে)। একটা জিনিস, অনেকের $a[i++]$ এর মানে বুঝতে সমস্যা হয়। আমি আসলে আগের অধ্যায়ে pre-increment($++i$) আর post-increment($i++$) নিয়ে খুব সংক্ষেপে কথা বলেছি। এখন আরেকটু বলা যাক। Pre-increment মানে হলো আগে। এর মান বাড়াবে এর পর পুরো লাইনকে execute করবে। যেমন মনে কর x এর মান 3 থাকা অবস্থাতে আমরা লিখলাম $y = ++x$. তাহলে আগে x এর মান বেড়ে 4 হবে এরপর y এ x এর মান বসানো হবে, অর্থাৎ y এর মানও 4 হবে। কিন্তু আমরা যদি post-increment ব্যবহার করি অর্থাৎ $y = x++$ তাহলে আগে y এ x এর মান বসবে, মানে 3. এরপর x এর মান বেড়ে 4 হবে। $a[i+] = 3$ মানে হলো $a[i] = 3$ এবং i এর মান এক বেড়ে যাওয়া। অর্থাৎ আমাদের সীভ ফাংশন শেষে nPrime এ আমরা 1 হতে n এর মধ্যে কয়টি মৌলিক সংখ্যা আছে তা পেয়ে যাব। আর Prime এর অ্যারে তে তুমি সব মৌলিক সংখ্যাগুলো পেয়ে যাবে। সেই সাথে তুমি mark এর অ্যারে দেখে খুব দ্রুত বলে দিতে পারবে যে কোন একটি সংখ্যা মৌলিক কি না। যদি $mark[i]$ এর মান 1 হয় এর মানে এটি মৌলিক নয়, আর 0 মানে হলো i মৌলিক সংখ্যা। অনেকে মনে করতে পারে এই কোডটা আরও সহজ ভাবে করা যেতো। হ্যাঁ হয়তো আমি চাইলে মৌলিক সংখ্যার তালিকা বানানোর কাজ নাও করতে পারতাম, বা হয়তো জোড় সংখ্যার কেইস আলাদা করে কোড নাও করতে পারতাম কিন্তু যেহেতু মাঝে মাঝে এসব আসলেই করা লাগে তাই আমি একবারেই সব কিছু দেখিয়ে দিয়েছি। তোমরা চাইলে তোমাদের পছন্দ মতো সীভের কোড করে লাইব্রেরীতে রাখতে পারো যাতে যখনই দরকার হবে তখনই কপি-পেস্ট করতে পার। কিন্তু ছোট থাকতে আসলে এই কপি-পেস্ট করা উচিত না, তাহলে লাইব্রেরী কাছে না থাকলে কোড করা খুব কষ্টকর হয়ে যাবে। শেষ করব Prime অ্যারে এর সাইজ নিয়ে কথা বলে। অবশ্যই এই অ্যারের সাইজ n এর সমান করার দরকার নেই, কারণ 1 হতে n এর মাঝে n এর চেয়ে অনেক কম সংখ্যক মৌলিক সংখ্যা আছে। n এর মান যদি বেশ অ্যারে ছাড়া একবার কোড চালিয়ে দেখি যে nPrime এর মান কত হয়। এর পর সেইভাবে আমি অ্যারের সাইজ দেই।

যাই হোক, আশা করি সীভ মোটামুটি বোঝা গেছে। এবার সীভ এর দুটি ভ্যারিয়েশন (variation) নিয়ে কথা বলা যায়।

- ⦿ **Memory Efficient Sieve:** এখানে খেয়াল করলে দেখবে যে n যত বড়, তত বড় অ্যারের দরকার হয় mark এর জন্য। আমরা কিন্তু জানি 2 ছাড়া সব জোড় সংখ্যা এর mark এ 1 থাকবে সুতরাং এই তথ্য ব্যবহার করে আমরা মেমোরীর ব্যবহার অর্ধেক করে ফেলতে পারি। আবার, mark অ্যারে এর প্রতিটি জায়গায় আমরা কিন্তু 0 আর 1 ছাড়া আর কিছু রাখি না। আমরা চাইলে, প্রতিটি int এ থাকা 32 বিটকে কাজে লাগিয়ে একটা ভ্যারিয়েশন 32টি সংখ্যার তথ্য রাখতে পারি এবং ব্যবহৃত মেমোরীর পরিমাণ আরও 32 ভাগ করতে

পারি। এরকম আরও কিছু অপটিমাইজেশন খাটিয়ে আমরা চাইলে প্রয়োজনীয় মেমোরীর পরিমাণ অনেক কমিয়ে ফেলতে পারি। তোমরা চাইলে ইন্টারনেটে yarin's sieve লিখে খোঁজ করতে পারো।

- Segmented Sieve: অনেক সময় আমাদের ঠিক 1 হতে n না, বরং a হতে b এর মাঝের মৌলিক সংখ্যাগুলোর দরকার হয় যেখানে a, b হয়তো $10^{12} \sim 10^{14}$ সীমার মধ্যে থাকে কিন্তু বাড়তি একটি শর্ত থাকে যে, $b - a \leq 10^6$. এসব ক্ষেত্রে আমরা $[a, b]$ সীমায় সীভ চালাতে পারি। এর জন্য প্রথমেই আমাদের \sqrt{b} পর্যন্ত সব মৌলিক সংখ্যা বের করে রাখতে হবে। এবার আমরা একটি অ্যারে নিবো $b - a + 1$ দৈর্ঘ্যের। প্রথম জায়গা হবে a এর marker, পরেরটা $a + 1$ এর এরকম করে শেষেরটা b এর। যদিও a, b অনেক বড় কিন্তু $b - a$ কিন্তু অনেক ছোট। তাই আমরা চাইলেই $b - a + 1$ দৈর্ঘ্যের একটি অ্যারে নিতেই পারি। এবার আমরা আগের মতো \sqrt{b} এর থেকে ছোট বা সমান মৌলিক সংখ্যাগুলি দিয়ে এর সীমায় কাটাকুটির কাজ করব (অথবা চাইলে 2 হতে \sqrt{b} পর্যন্ত সকল সংখ্যা দিয়েও করতে পার তবে সেক্ষেত্রে একটু সময় বেশি লাগবে)। কাটাকুটি শেষে $[a, b]$ সীমার যেসব সংখ্যা এখনও কাটা যায় নায় তারা হলো মৌলিক সংখ্যা।

অনুশীলনী

- একটি সংখ্যাকে মৌলিক উৎপাদকে বিশ্লেষণ কর। অর্থাৎ এটি কোন কোন মৌলিক সংখ্যা দিয়ে বিভাজ্য এবং সেই সব মৌলিক সংখ্যার ঘাত (power) গুলো বের কর।

৩.১.২ একটি সংখ্যার গুণনীয়কসমূহ

তুমি যদি কোনো একটি সংখ্যার সব গুণনীয়কগুলো বের করতে চাও তাহলে কোড ৩.২ এর মতো $O(\sqrt{n})$ এ খুব সহজেই সব গুণনীয়ক বের করে ফেলতে পার। \sqrt{n} পর্যন্ত লুপ চালাবে আর দেখবে যে i কি n কে ভাগ করে কিনা। করলে i ও n/i দুটি সংখ্যাই n এর গুণনীয়ক। তবে $i = \sqrt{n}$ এর ক্ষেত্রে একটু সাবধান থাকবে কারণ $i = n/i$. তাই যদি সাবধান না হও তাহলে গুণনীয়কের তালিকাটা একটু বড় হয়ে যেতে পারে আর কি! এখন কথা হলো আমরা কি সীভ অ্যালগরিদমকে পরিবর্তন করে 1 হতে n পর্যন্ত প্রতিটি সংখ্যার সব গুণনীয়ক বের করতে পারি? অবশ্যই! তবে এটির রানটাইম $O(n \log n)$. কোড ৩.৪ এ এর কোডটি দেখানো হলো। এখানে তেমন কিছুই না, শুধু প্রতিটি সংখ্যার জন্য আমরা এর গুণিতকের তালিকাগুলোতে তাকে ঢুকিয়ে দেব। এখানে আমাদের কোনো mark রাখার প্রয়োজন হবে না। তোমরা যদি মনে কর যে মেমোরী তো অনেক বেশি লেগে যাবে! না, n টি সংখ্যার গুণনীয়ক আসলে সর্বমোট $n \log n$ এর বেশি না। আমরা এই কোডে আমাদের সুবিধার জন্য STL এর ভেষ্টের ব্যবহার করেছি। ভেষ্টের না ব্যবহার করে ডায়নামিক লিঙ্কড লিস্ট (Dynamic Linked List) ব্যবহার করা যায়, কিন্তু তাতে সমাধানটি অনেক ঝামেলার হয়ে যায়।

```

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

```

প্রশ্ন হতে পারে কেন টাইম কমপ্লেক্সিটি $n \log n$? খুব সহজ! 1 হতে n পর্যন্ত i এর গুণিতক হলো মোট n/i টি। আর সব i এর জন্য যদি n/i যোগ কর তাহলে এটি মোটামুটিভাবে $n \log n$ হয়।^১

অনেক সমস্যায় গুণনীয়কের তালিকা হয়তো লাগে না, কিন্তু প্রতিটি সংখ্যার গুণনীয়কগুলোর যোগফল বা গুণনীয়কের সংখ্যা দরকার হয়। আশা করি কীভাবে করবে তা বুঝতে পারছ! লিস্টে গুণনীয়কগুলি না ঢুকিয়ে আমরা গুণনীয়কগুলি যোগ করব বা গুণনীয়ক পেলে counter এক করে বাড়াবে দিব। শেষ!

কোনো একটি সংখ্যার গুণনীয়ক নিয়ে যখন সমস্যা থাকে তখন আরেকটি উপায় বেশ কাজে লাগে। ধরা যাক, $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ এখানে p_i হলো মৌলিক সংখ্যা। একে কোনো একটি সংখ্যার মৌলিক উৎপাদকে বিশ্লেষণ বলে। এখন চিন্তা করে দেখ, d কে যদি n এর গুণনীয়ক হতে হয় তাহলে তার কী কী বৈশিষ্ট্য থাকতে হবে। প্রথমত, d এর ভেতরে p_i ছাড়া আর কোনো মৌলিক উৎপাদক থাকা যাবে না। দ্বিতীয়ত, p_i এর ঘাত (power) কিন্তু a_i এর থেকে বেশি হতে পারবে না। অর্থাৎ: $d = p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}$ যেখানে $0 \leq b_i \leq a_i$. এই সমীকরণ থেকে আমরা বলে দিতে পারি n এর গুণনীয়ক কয়টি আছে (NOD = Number of Divisor) বা তার গুণনীয়কগুলোর যোগফল কত (SOD = Sum of Divisor)!

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

$$\begin{aligned} SOD(n) &= (p_0^0 + p_1^1 + \dots + p_1^{a_1})(p_2^0 + p_2^1 + \dots + p_2^{a_2}) \dots (p_k^0 + p_k^1 + \dots + p_k^{a_k}) \\ &= \frac{p_1^{a_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{a_2+1} - 1}{p_2 - 1} \cdot \dots \cdot \frac{p_k^{a_k+1} - 1}{p_k - 1} \end{aligned}$$

NOD এর সূত্র এরকম কেন হলো তাতে বুঝাই যাচ্ছে তাই না? কারণ p_1 এর ঘাত 0 হতে a_1 এর ভিতরে যেকোনোটি হতে পারে, অর্থাৎ $a_1 + 1$ রকম। এভাবে প্রতিটি ঘাত কত রকম হতে

^১তোমরা চাইলে নেটে harmonic sum এর approximation লিখে ইন্টারনেটে খোঁজ করতে পারো।

পারে তা গুণ করলে আমরা মোট কতগুলি উৎপাদক হতে পারে তা বের করে ফেলতে পারব। একইভাবে আমরা দেখি $p_1^{b_1}$ কী কী হতে পারে: $p_1^0, p_1^1 \dots p_1^{a_1}$. এরকম p_2, p_3 এসবের জন্যও আমরা পেতে পারিযে তারা কীরকম হতে পারে। এখন যদি আমরা SOD এর ফর্মুলা কে ভেঙ্গে দিয়ে লিখি মানে সব পদকে যদি গুণ করে ফেলি তাহলে দেখবে আমরা সব রকম গুণনীয়ক d পেয়ে যাব মানে $p_1^0 p_2^0, p_1^0 p_2^1, \dots$ এরকম সব গুণনীয়ক। এখন আমরা চাইলে প্রত্যেক $p_0^0 + p_0^1 + \dots p_k^{a_k}$ কে সূত্রের সাহায্যে লিখতে পারি। এভাবেই আমরা SOD এর সূত্র পেয়ে গেছি। যেমন মনে কর আমরা জানতে চাছি 12 এর SOD. $12 = 2^2 \times 3^1$. তাহলে $SOD(12)$ হবে $(2^0 + 2^1 + 2^2)(3^0 + 3^1)$ বা $2^0 3^0 + 2^0 3^1 + 2^1 3^0 + 2^1 3^1 + 2^2 3^0 + 2^2 3^1$ বা $1 + 3 + 2 + 6 + 4 + 12$. দেখলে তো কীভাবে সবগুলো পাওয়া গেল?

৩.১.৩ গ.সা.গু. (GCD) ও ল.সা.গু. (LCM)

গ.সা.গু. এর পূর্ণ রূপ হলো: গরিষ্ঠ সাধারণ গুণনীয়ক বা ইংরেজীতে যাকে বলে Greatest Common Divisor(GCD)। অর্থাৎ তোমাকে কিছু সংখ্যা দেওয়া থাকলে তাদের গ.সা.গু. হলো সবচেয়ে বড় সংখ্যা যা এদের সবাইকে ভাগ করে। অন্যদিকে ল.সা.গু. এর পূর্ণ রূপ হলো: লঘিষ্ঠ সাধারণ গুণিতক বা ইংরেজীতে Least Common Multiple(LCM)। কিছু সংখ্যা দেওয়া থাকলে তাদের ল.সা.গু. হবে সবচেয়ে ছোট ধনাত্মক সংখ্যা যাকে সবাই ভাগ করতে পারে। যেমন আমাদের যদি $\{12, 42, 54\}$ দেওয়া থাকে তাহলে এদের গ.সা.গু. হবে 6 আর ল.সা.গু. হবে 756. আমরা ছোট বেলাতে একটি মজার সূত্র শিখে এসেছি। সূত্রটা হলো যদি a এবং b সংখ্যা দুটির গ.সা.গু. g এবং ল.সা.গু. l হয় আমরা বলতে পারি: $a \times b = g \times l$. সুতরাং আমরা যদি দুটি সংখ্যার গ.সা.গু. বের করতে পারি তাহলে ল.সা.গু. খুব সহজেই বের হয়ে যাবে। প্রশ্ন হলো আমরা গ.সা.গু. কীভাবে বের করব? একটি উপায় হলো a ও b এর মধ্যে যেটি ছোট সেই সংখ্যা থেকে শুরু করে 1 পর্যন্ত দেখা, যেই সংখ্যা দিয়ে a এবং b উভয়েই প্রথম ভাগ যাবে সেটিই তাদের গ.সা.গু। কিন্তু এর রান্টাইম $O(n)$. এর থেকে ভালো উপায় কিন্তু তোমরা জানো, ছোটবেলায় স্কুলে থাকতে শিখেছ সেটি হলো ইউক্লিডের পদ্ধতি। মনে কর আমাদের a আর b দিয়ে বলা হলো এদের গ.সা.গু. বের করতে হবে। আমরা প্রথমে a কে b দিয়ে ভাগ দিব। যদি নিঃশেষে ভাগ যায়, তাহলে b ই গ.সা.গু. কারণ b এর থেকে বড় কোনো সংখ্যা কিন্তু b কে ভাগ করে না (যদিও a কে ভাগ করতে পারে)। এখন যদি b দিয়ে a 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 কী 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 গ.সা.গু. 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 গ.সা.গু. পেয়ে যাব। তোমাদের মনে হতে পারে যে অনেক বার এই কাজ করতে হবে! আসলে কিন্তু তা না। এটার প্রকৃত রান্টাইম তোমাদের বললে আপাতত বুঝবে না তবে এটুকু বিশ্বাস করতে পার যে long long ডেটাটাইপে যত বড় সংখ্যা ধরা সম্ভব তাদের যদি গ.সা.গু. বের করতে বলা হয় তাহলে 100 ~ 150 ধাপের

বেশি আসলে লাগবে না। ১ গ.সা.গু. নির্ণয়ের একটি রিকার্সিভ প্রোগ্রাম কোড ৩.৫ এ দেওয়া হলো। তবে যদি সংখ্যা দুইটির একটি যদি ০ হয় তাহলে কিন্তু এই পদ্ধতি কাজে নাও দিতে পারে (যদি $b = 0$ হয় তাহলে $a \% b$ run time error দিবে)। আশা করি সেক্ষেত্রে কি করবে তা নিজেরাই নির্ণয় করতে পারবে? তাও বলি, একটি উপায় হলো $a \% b == 0$ চেক না করে $b == 0$ চেক করা এবং যদি সত্যি হয় তাহলে a কে রিটার্ন করা।

কোড ৩.৫: gcd.cpp

```

1 int gcd(int a, int b)
2 {
3     if (a % b == 0) return b; // if b divides a, b is gcd
4     return gcd(b, a % b); // if not, find gcd of b and a % b
5 }
```

তাহলে আমরা এভাবে দুইটি সংখ্যার গ.সা.গু. বের করতে পারি। একাধিক সংখ্যার গ.সা.গু. বের করার জন্য আমাদের যা করতে হবে তা হলো প্রথম দুইটির গ.সা.গু. প্রথমে বের করব। এরপর তার সঙ্গে তৃতীয় সংখ্যার গ.সা.গু., এরপর তার সঙ্গে চতুর্থ এভাবে। আর ল.সা.গু.? আমরা উপরের দেখে আসা সূত্র ব্যবহার করে দুইটি সংখ্যার ল.সা.গু. পাব। এরপর প্রথম দুইটি সংখ্যার ল.সা.গু.র সাথে তৃতীয় সংখ্যার ল.সা.গু. বের করব এভাবে আমরা অনেকগুলি সংখ্যার ল.সা.গু. বের করে ফেলতে পারি।

৩.১.৪ অয়লার এর টোশেন্ট ফাংশন (Euler's Totient Function - ϕ)

শুরুতেই আমরা জেনে নিই টোশেন্ট ফাংশন (Totient Function) কী জিনিস।

$\phi(n) = n$ এর থেকে ছোট বা সমান এমন কতগুলো সংখ্যা আছে যা n এর সঙ্গে সহমৌলিক (coprime)

সহমৌলিক অর্থ হলো তাদের কোনো সাধারণ গুণনীয়ক নেই (অবশ্যই 1 এর থেকে বড় গুণনীয়ক এর কথা বলছি)। যেমন, $\phi(12) = 4$ কারণ 2, 3, 4, 6, 8, 9, 10, 12 এই আটটি সংখ্যার সঙ্গে 12 এর কোনো না কোনো সাধারণ গুণনীয়ক আছে। 12 এর থেকে ছোট বাকি 1, 5, 7, 11 এই চারটি সংখ্যার সঙ্গে কোনো সাধারণ গুণনীয়ক নেই। SOD, NOD এর মতো এরও একটি সূত্র আছে। যদি $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ হয় তাহলে:

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

^১তোমাদের আগ্রহ থাকলে এই বিষয়ে উইকিপিডিয়াতে পড়তে পার। মজার ব্যপার হলো এর রানটাইমের সঙ্গে ফিবোনাচি সংখ্যার একটা সম্পর্ক আছে!

সুতরাং যদি কোনো একটি সংখ্যার ϕ বের করতে হয় তাহলে তোমরা খুব সহজেই মৌলিক উৎপাদকে বিশ্লেষণ করে বের করে ফেলতে পারবে। কথা হলো মৌলিক সংখ্যা বা গুণনীয়কের মতো কি এক্ষেত্রেও 1 হতে n এর ϕ এর মান দ্রুত বের করা সম্ভব? অবশ্যই সম্ভব, প্রতিটি মৌলিক সংখ্যা p এর জন্য এর গুণিতকে গিয়ে তাকে p দিয়ে ভাগ ও $p - 1$ দিয়ে গুণ করলেই হবে। এভাবে 1 হতে n পর্যন্ত সব মৌলিক সংখ্যার জন্য এই কাজ করলেই আমরা সব সংখ্যার ϕ এর মান পেয়ে যাব। (কোড ৩.৬)

কোড ৩.৬: sieve phi.cpp

```

1 int phi[1000006], mark[1000006];
2
3 void sievephi(int n)
4 {
5     int i, j;
6
7     // initialization
8     for (i = 1; i <= n; i++) phi[i] = i;
9
10    phi[1] = 1;
11    mark[1] = 1;
12
13    for (i = 2; i <= n; i++)
14        if (!mark[i]) // if i is prime
15        {
16            for (j = i; j <= n; j += i)
17            {
18                mark[j] = 1;
19                // phi[j] will be divisible by i
20                // so no need to worry.
21                phi[j] = phi[j] / i * (i - 1);
22            }
23        }
24    }
25 }
```

আর যদি তোমার 1 হতে n পর্যন্ত সবগুলির ϕ ফাংশনের মান দরকার না হয় তাহলে তো মৌলিক উৎপাদকে বিশ্লেষণ করে উপরের সুত্র ব্যবহার করে খুব সহজেই কোনো একটি নির্দিষ্ট n এর জন্য ϕ ফাংশনের মান বের করতে পারবে। এর কোড ৩.৭ এ দেওয়া হলো। কেন এই কোড কাজ করে

সেটা একটু চিন্তা করে দেখা যাক। প্রথমত যেকোনো সংখ্যাকে সবচেয়ে ছোট যেই সংখ্যা দিয়ে ভাগ করা যায় সেটি একটি মৌলিক সংখ্যা। আমরা যদি এর পরের মৌলিক সংখ্যা বের করতে চাই (যা একে ভাগ করবে) তাহলে এই মৌলিক সংখ্যা দিয়ে যতবার আমাদের সংখ্যাকে ভাগ করা যায়, ততবার ভাগ দিতে থাকতে হবে। এরপর আবার পরের মৌলিক সংখ্যা খোঁজা শুরু। এভাবে আমরা সব মৌলিক উৎপাদক খুঁজে বের করতে পারি। এখন এই জায়গায় আমরা একটা অপটিমাইজেশন করতে পারি। আর তা হলো, খেয়াল কর n এর \sqrt{n} এর থেকে বড় কেবল একটিই মৌলিক উৎপাদক থাকতে পারে। তাই আমাদের লুপ \sqrt{n} এর বেশি চালানোর দরকার নেই। লুপ শেষে যদি আমরা দেখি n এখনও 1 হয় নাই তার মানে এখন n এ যেই সংখ্যা আছে তা মূল n কে ভাগ করা আরেকটি মৌলিক সংখ্যা। এখানে একটা জিনিস লক্ষ্যনীয় আর তা হলো আমরা কিন্তু n এর মান পরিবর্তন করছি। সুতরাং প্রথমে যদিও আমরা মূল n এর বর্গমূল পর্যন্ত চালাতে চাচ্ছিলাম কিন্তু এখন হয়তো পরিবর্তিত n এর বর্গমূল পর্যন্ত চালাচ্ছি। এতে কিন্তু কোন সমস্যা নেই। কারণ আমি যেই যুক্তিতে বলেছিলাম যে n এর বর্গমূল পর্যন্ত চালালেই হবে, সেই একই যুক্তিতে আমরা বলতে পারি এই পরিবর্তিত n এর বর্গমূল পর্যন্ত চালালেই হবে। একটু চিন্তা করে দেখ।

কোড ৩.৭: loop phi.cpp

```

1 int phi(int n) {
2     int ret = n;
3     for (int i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             // i is a prime dividing n.
6             while (n % i == 0) {
7                 // divide all the factors of i
8                 n /= i;
9             }
10            // same as: ret *= (1 - 1/p)
11            ret -= ret / i;
12        }
13    }
14    if (n > 1) {
15        // there can be only one prime greater
16        // than sqrt(n) that divides n
17        ret -= ret / n;
18    }
19    return ret;
20 }
```

এর কিছু ছোট খাটো বৈশিষ্ট্য জেনে রাখতে পারো।

যদি p একটি মৌলিক সংখ্যা হয় তাহলে $\phi(p) = p - 1$.

যদি p একটি মৌলিক সংখ্যা হয় এবং n একটি ধনাত্মক সংখ্যা হয় তাহলে $\phi(p^n) = p^n - p^{n-1}$.

যদি a এবং m এর মাঝে কোনো সাধারণ উৎপাদক না থাকে, অর্থাৎ যদি তারা সহমৌলিক হয় তাহলে $a^{\phi(m)} \equiv 1 \pmod{m}$. এটি অয়লারের সূত্র (Euler's theorem) নামে পরিচিত। m যদি মৌলিক সংখ্যা হয় তাহলে আমরা লিখতে পারি $a^{m-1} \equiv 1 \pmod{m}$. এটি ফার্মার লিটল থেওরেম (Fermat's little theorem) নামে পরিচিত।

৩.১.৫ BigMod

BigMod একটি অত্যন্ত গুরুত্বপূর্ণ পদ্ধতি। ধরা যাক তোমার কাছে অনেক লাল বল এবং সাদা বল আছে। তুমি n বার বোলিং করলে, প্রতিটি বল হয় সাদা বল দিয়ে করবে না হয় লাল বল দিয়ে। তুমি কতভাবে 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 সেই বড় উত্তর না চেয়ে $\pmod{10^7}$ বা এরকম একটি সংখ্যা দিয়ে দেওয়া হয় যা দিয়ে mod করে উত্তর চাওয়া হয়। যেমন ধর তোমাদের জিজ্ঞাসা করলাম, $2^{100} \pmod{7}$ কত? কী করবে? 2^{100} বের করে এর পর 7 দিয়ে ভাগ করে উত্তর দিবে? খেয়াল কর, 2^{100} কিন্তু long long ডেটাটাইপেও ধরবে না। তাহলে উপায়? তুমি চাইলে প্রতিবার গুণ করে সঙ্গে সঙ্গেই mod করতে পার, এতে করে উত্তরটা শুধু int ডেটাটাইপ ব্যবহার করেই পেয়ে যাবে! কিন্তু তোমাকে যদি 100 এর থেকে আরও বড়, অনেক বড়, ধর প্রায় 10^{18} দেওয়া হয়? তাহলে কি করা সম্ভব? হ্যাঁ করা সম্ভব এবং সমাধানের উপায়টিও অনেক সহজ। মনে কর তোমাকে $2^{100} \pmod{7}$ বের করতে বলা হয়েছে। তুমি কী করবে, $2^{50} \pmod{7}$ বের করবে, ধর এটি a , তাহলে $2^{100} \pmod{7}$ হবে $(2^{50} \times 2^{50}) \pmod{7}$ বা $((2^{50} \pmod{7}) \times (2^{50} \pmod{7})) \pmod{7}$ বা $(a \times a) \pmod{7}$. অর্থাৎ তুমি তোমার কাজকে অর্ধেক করে ফেললে। তোমার এখন আর লুপ ঢালিয়ে 100 টি 2 গুণ করতে হবে না, 50 টি 2 গুণ করে এই গুণফলকে তার সঙ্গেই গুণ দিলে তুমি উত্তর পেয়ে যাবে। কিন্তু একইভাবে তোমার কিন্তু 50 বার গুণ করার দরকার নেই, 25 বার গুণ করে আবার সেই গুণফলকে তার সঙ্গেই গুণ করে গুণফল তুমি পেয়ে যাবে। কিন্তু এবার? বিজোড় সংখ্যার বেলায়? এবার তো আর অর্ধেক করতে পারবে না। তাতে কী যায় আসে! তুমি 2^{24} বের করে তার সঙ্গে 2 কে গুণ করতে পার। যেহেতু ঘাত বিজোড় বলে এই কাজ করছ সুতরাং এর পরের ধাপে ঘাত অবশ্যই জোড় হবে এবং আবার তুমি 2 দিয়ে ভাগ করতে পারবে। আমি অবশ্য অন্য কাজ করি তা হলো 2^{25} বের করতে বললে আগের মতো আর অর্ধেক মানে $25/2 = 12$ এর জন্য মান বের করি (integer division করছি) মানে 2^{12} এবং আগের মতো একে এটি দিয়েই গুণ করে 2^{24} করে ফেলি। আর এর সঙ্গে একটা অতিরিক্ত 2 গুণ করে ফেলি তাহলে 2^{25} পেয়ে যাব। এভাবে

করলে আমার কাছে মনে হয় কোড বেশ ছোট হয়ে যায়। যাই হোক, এভাবে ঘাতকে অর্ধেক করতে থাকলে এক সময় 0 যখন হয়ে যাবে তখন আমরা base কেইস খাটাতে পারব, অর্থাৎ আমরা জানি যে $2^0 = 1$ সুতরাং এবার আমরা আমাদের অন্য গুণগুলো করে আমাদের উত্তর বের করে ফেলতে পারব। উদাহরণ দেওয়া যাক একটা। মনে কর আমাদের বের করতে হবে 2^5 , এর জন্য আমাদের বের করতে হবে $a = 2^2 [5/2 = 2]$. 2^2 এর জন্য বের করতে হবে $b = 2^1 [2/2 = 1]$, আর 2^1 এর জন্য বের করতে হবে $c = 2^0 [1/2 = 0]$. আমরা 0 ঘাত অর্থাৎ base কেইসে চলে এসেছি, সুতরাং আমরা জানি $c = 1$. তাহলে $b = 2^1 = 2^0 \times 2^0 \times 2 = 1 \times 1 \times 2 = 2$. একই ভাবে $a = 2^2 = 2^1 \times 2^1 = 2 \times 2 = 4$. এবং সবশেষে $2^5 = 2^2 \times 2^2 \times 2 = 4 \times 4 \times 2 = 32$. যদিও এখানে mod এনে আরও পাঁচ লাগানো যেতো কিন্তু উদাহরণটা খুব সহজ রাখার জন্য কোনো mod করা হলো না, তোমরা প্রতি গুণ এর শেষে যদি mod কর তাহলেই BigMod হয়ে যাবে।

এখন তাহলে এই জিনিস কোড করব কীভাবে? ধরে নিই আমাদের কাছে একটা Black Box আছে যাকে a, b, M দিলে $a^b \bmod M$ বের করে দেয়। সে যা করবে তা হলো সে আবার সেই Black Box কে $a, b/2, M$ দিবে এবং সেই ফলাফল দিয়ে নিজের ফলাফল বের করবে। যদি বিজোড় হয় তাহলে অতিরিক্ত a গুণ করবে এই আর কি! আর এভাবে কতক্ষণ চলবে? যতক্ষণ না, $b = 0$ হয়। এখানে Black Box হলো একটি ফাংশন এবং এভাবে একটি ফাংশন এর ভেতর থেকে একই ফাংশন ব্যবহার করাকেই রিকার্সিভ ফাংশন (recursive function) বলে যা আমরা আগের অধ্যায়ে দেখে এসেছি। আমরা কোড 3.8 এ এই প্রোগ্রামটি দিলাম।

কোড 3.8: 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 অ্যালগরিদমের রানটাইম হলো $O(\log n)$. কেন? কারণ প্রতিবার আমাদের n দুভাগ হচ্ছে। মনে কর আমাদের মোট k সংখ্যকবার bigmod কে কল করতে হয় রিকার্সিভভাবে। এর মানে n হলো 2^k এর মত সংখ্যা। অর্থাৎ $n = 2^k$ বা $k = \log_2 n$. এজন্যই bigmod এর টাইম কমপ্লেক্সিটি $O(\log n)$ ।¹

এই ধরনের সমাধানের পদ্ধতিকে divide and conquer বলা হয়ে থাকে। এখানে একটি

¹আমরা কম্পিউটার সায়েন্সে যত log ব্যবহার করি তা সবগুলিই সাধারণত 2 ভিত্তিক।

বড় সমস্যাকে ছোট ছোট ভাগ করা হয় এবং তাদের সমাধান জোড়া লাগিয়ে বড় সমস্যার সমাধান বের করা হয়। আমি যখন BigMod দেখাই এর সঙ্গে আরও একটি সমস্যা দেখাই। তা হলো, $1 + a + a^2 + \dots + a^{b-1} \bmod M$ বের করা। অবশ্যই এর আগের সমস্যার মতো এখনেও b অনেক বড়। সুতরাং তুমি যে বার বার প্রতিটি পদ (term) এর উপর বের করবে তা হবে না। এখনেও কিন্তু এর আগের মতো বুদ্ধি খাটানো সম্ভব। আমরা $b = 6$ এর জন্য দেখি কীভাবে একে দুভাগে ভাগ করা সম্ভব!

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

মনে কর আমাদের এই যোগফল বের করার ফাংশনের নাম bigsum। তাহলে b ঘাতের জন্য bigsum বের করতে আমরা $b/2$ ঘাতের জন্য bigsum এর সাহায্য নিতে পারি। এর সঙ্গে আমাদের bigmod এরও সাহায্য নিতে হবে উপরের a^3 বের করার জন্য। বিজোড়ের ক্ষেত্রে তোমরা চাইলে আগের মতো a^{b-1} কে আলাদা করে বের করতে পারো আর বাকি যোগফলকে bigsum দিয়ে বের করতে পারো। সর্বমোট $\log n$ ধাপ করতে হয় bigsum এর এবং প্রতি ধাপে আমাদের bigmod এর জন্য আরও $\log n$ সময় দরকার হয়, সুতরাং আমাদের রানটাইম $O((\log n)^2)$ । এটা খুব একটা বড় cost না। কিন্তু তবুও আমরা এর $O(\log n)$ সমাধান শিখতে পারি। আমরা সমীকরণটিকে অন্যভাবে দুভাগ 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 $\text{bigsum}(a, b, M)$ বের করতে আমরা $\text{bigsum}(a^2, b/2, M)$ বের করব। a যেন পরবর্তীতে ওভারফ্লো (Overflow) না করে সেজন্য আমরা আসলে a^2 এর বদলে $a^2 \bmod M$ পাঠাব। b বিজোড় হলে আশা করি বুঝতে পারছ যে কী করব? তাও দেখাই:

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

আশা করি কোডটা নিজেরাই করতে পারবে। একটা কথা, সেটা হলো তুমি যদি মনে করে থাকো যে এটা তো গুণোত্তর ধারা, আমরা গুণোত্তর ধারার সুত্র খাটালাম না কেন? কারণ সুত্রে ভাগ আছে: $\frac{a^k - 1}{a - 1}$ এবং কিছুক্ষণ পরেই শিখব যে mod এ সবসময় ভাগ করা সম্ভব না। যাই হোক, তোমাদের একইভাবে সমাধান করা যাবে এমন আরেকটি সমস্যা দেই অনুশীলনের জন্য।

অনুশীলনী

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

৩.১.৬ মডুলার ইনভার্স (Modular Inverse)

আগেই বলেছি অনেক হিসাবের উত্তর অনেক বড় আসে সুতরাং প্রায় সময়ই আমাদের সেই বড় উত্তর না চেয়ে কোনো একটি সংখ্যা দিয়ে mod করে উত্তর চাওয়া হয়। এখন সেই হিসাবে যদি যোগ, বিয়োগ, গুণ থাকে তাহলে কোনো সমস্যা হয় না, কিন্তু যদি ভাগ থাকে তাহলে বেশ সমস্যা হয়ে যায়, কারণ $\frac{a}{b} \mod M$ এবং $\frac{a \mod M}{b \mod M}$ এক কথা নয়। উদাহরণ তো খুব সহজ $\frac{12}{3} \mod 3 = \frac{12 \mod 3}{3 \mod 3} = \frac{0}{0}!!!$ 0 দিয়ে ভাগ? ভয়ংকর কথা! তুমি যদি b দিয়ে ভাগ করতে চাও তাহলে $b^{-1} \mod M$ বের করতে হবে এবং এটি দিয়ে গুণ করলেই b দিয়ে ভাগের কাজ হয়ে যাবে। আমরা কিছুক্ষণ আগেই শিখে এসেছি যে M যদি মৌলিক সংখ্যা হয় তাহলে $b^{M-1} \equiv 1 \mod M$ বা $b^{-1} \equiv b^{M-2} \mod M$ আর যদি মৌলিক সংখ্যা না হয় তাহলে $b^{\phi(M)} \equiv 1 \mod M$ বা $b^{\phi(M)-1} \equiv b^{-1} \mod M$. তবে এই দুক্ষেত্রেই M এবং b কে সহমৌলিক হতে হবে। আর তোমরা তো জানোই কীভাবে $b^{M-2} \mod M$ বা $b^{\phi(M)-1} \mod M$ বের করতে পারব? BigMod!

যদি M আর b সহমৌলিক না হয়? তাহলে এর কোনো নির্দিষ্ট ইনভার্স নেই। কেন? তার আগে একটা জিনিস, সেটা হলো $b^{-1} \mod M$ আসলে কি? এটা এমন একটি সংখ্যা যাকে b দিয়ে গুণ করলে কাটাকুটি গিয়ে 1 থাকবে। অর্থাৎ, $b \times b^{-1} \mod M \equiv 1$. যেমন $3^{-1} \equiv 3^5 \mod 7$ বা $243 \mod 7$ বা 5 আর $3 \times 5 = 15 \equiv 1 \mod 7$. কিন্তু আমরা যদি জানতে চাই $4^{-1} \mod 6$. ধরা যাক এটি x তাহলে $4x \equiv 1 \mod 6$ হতে হবে। কিন্তু আমরা যদি একে একে এর মান 0 হতে 5 পর্যন্ত চেষ্টা করি¹। কোনোটার জন্য কি $4x \mod 6$ কী 1 হয়? হয় না। এর মান সমাধান নেই। এটা প্রমাণ করাও বেশ সহজ। মনে কর $b^{-1} \equiv x \mod M$ আর b ও M যেহেতু সহমৌলিক নয় তার মানে তাদের একটি সাধারণ গুণনীয়ক আছে, ধরা যাক p. আমরা লিখতে পারি $bx \equiv 1 \mod M$ অর্থাৎ bx কে M দিয়ে ভাগ করলে ভাগশেষ 1. ধরা যাক ভাগফল হলো Q. তাহলে আমরা লিখতে পারি $bx - QM = 1$. কিন্তু দেখ বাম দিকের দুটি টার্ম bx ও QM কিন্তু দ্বারা ভাগ যায়। কিন্তু ডানদিকের 1 কিন্তু ভাগ যায় না। অর্থাৎ এরকম x আসলে কখনই থাকা সম্ভব না।

৩.১.৭ Extended GCD

যদি a ও b এর গ.সা.গ. 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 সাধারণ ইউক্লিডের গ.সা.গ. বের করার পদ্ধতি হতে পাওয়া। এখন গ.সা.গ. বের করার সময় অবশ্যই ছোট সংখ্যাকে কোনো একটি সংখ্যা দিয়ে গুণ করে বড়টি হতে বাদ দিয়েই ভাগশেষ বের করা হয়। এই গুণ ও বিয়োগের কাজটা আমাদের পরের দুই কলামেও করতে হবে। এরকম করলে

¹খেয়াল কর 6 চেষ্টা করা আর 0 চেষ্টা করা একই কথা, একই ভাবে 7 আর 1 একই কথা। তাই 0 হতে 5 চেষ্টা করা আর সব চেষ্টা করা একই কথা।

আমরা যখন প্রথম কলামে গ.সা.গু. পাব তখন দ্বিতীয় ও তৃতীয় কলামে x ও y এর মান পেয়ে যাব। এক্ষেত্রে আমাদের $x = -1$ এবং $y = 2$.

টেবিল ৩.২: $a = 10$ ও $b = 6$ এর জন্য Extended GCD এর সিমুলেশন (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 ৩.৯ এ দেওয়া হলো যদিও কোডটায় আমরা ঠিক উলটোভাবে x এবং y এর মান বের করেছি। কেন? প্রথমত যখন কেউ আমাকে এই কোডের মত করে পদ্ধতিটা আমাকে বর্ণনা করে তখন আমার কাছে egcd খুব কঠিন লাগে এবং আমি নিশ্চিত যে পরদিন না হলেও তিন চার দিন পরে আমি সেই পদ্ধতি ভুলে যাব। কিন্তু আমি যেভাবে দেখলাম সেটা অনেক সহজ এবং স্বজ্ঞাত। কিন্তু কোড করার সময় আমার এই পদ্ধতি একটু ঝামেলার। এখানে যেই কোড দিয়েছি সেভাবেই বেশির ভাগ মানুষ করে, আসলে মনে হয় বেশির ভাগ মানুষ কোড কপি-পেস্ট করে egcd এর, যদিও নিশ্চিত না আমি। যাই হোক, তুমি যদি $\text{egcd}(10, 6, x, y)$ এভাবে কল কর তাহলে দেখবে x ও y এর মাঝে তোমার প্রকৃত x ও y এর মান চলে এসেছে। এখানে যদি তুমি egcd ফাংশন একটু খেয়াল কর তাহলে দেখবে এটি x ও y কে একটু অন্যভাবে নিয়েছে। একে প্রোগ্রামিংয়ের ভাষায় রেফারেন্স ভ্যারিয়েবল (reference variable) বলে। তোমার চাইলে ইন্টারনেটে রেফারেন্স ভ্যারিয়েবল নিয়ে একটু দেখতে পার। মূল কথা হচ্ছে যদি তুমি ভ্যারিয়েবলকে রেফারেন্স হিসাবে না ও তাহলে ফাংশনটি তার মানকে পরিবর্তন করতে পারে।^১ এই ফাংশনটি গ.সা.গু. রিটার্ন করে।

কোড ৩.৯: egcd.cpp

```

1 int egcd (int a, int b, int &x, int &y)
2 {
3     if (a == 0)
4     {

```

^১সফটওয়্যার ইন্ডাস্ট্রি সাধারণ রেফারেন্স নেয়াকে নিরুৎসাহিত করা হয়। যদি তুমি কোন ফাংশনে প্যারামিটারের মান পরিবর্তন করতে চাও তাহলে তোমার উচিত পয়েন্টার (pointer) হিসাবে নেওয়া। আর যদি তুমি মান পরিবর্তন করতে না চাও তাহলে তুমি const রেফারেন্স হিসাবে নিবে। এর কারণ হলো মনে কর তুমি একটি ফাংশন কল করছ। তোমার কাছে একটি ভ্যারিয়েবল আছে যার মান আসলে তুমি পরিবর্তন করতে চাও না কিন্তু তুমি অসাবধানতাবস্থ খেয়াল কর নি যে তোমার ওই ফাংশন রেফারেন্স নেয়। তাহলে তোমার ভ্যারিয়েবলের মান পরিবর্তন হয়ে যেতে পারে। এজন্য const রেফারেন্স ব্যবহার করতে বলা হয়।

```

5         x = 0; y = 1;
6         return b;
7     }
8
9     int x1, y1;
10    int d = egcd (b%a, a, x1, y1);
11
12    x = y1 - (b / a) * x1;
13    y = x1;
14
15    return d;
16 }

```

egcd ব্যবহার করেও আমরা মডুলার ইনভার্স বের করতে পারি (তবে আগের মত b ও M কে সহমৌলিক হতে হবে)। কোনো একটি b এর জন্য আমরা এমন একটি x বের করতে চাই যেন, $bx \equiv 1 \pmod{M}$. অর্থাৎ, $bx = 1 + yM$ যেখানে y হলো একটি পূর্ণ সংখ্যা। এখন, $bx - yM = 1$. আমরা জানি b ও M সহমৌলিক সূতরাং আমরা এমন একটি x ও y পাব যেন, $bx - yM = 1$ হয় বা, $bx \equiv 1 \pmod{M}$ হয়। তবে egcd ফাংশন x এর মান হিসেবে ঝণাত্মক সংখ্যা দিতে পারে, এ জিনিসটি খেয়াল রাখতে হবে। সেক্ষেত্রে x কে সঠিকভাবে¹ M দিয়ে mod করে এর অংশণাত্মক মানটা বের করতে হবে।

৩.২ কম্বিনেটরিক্স (Combinatorics)

৩.২.১ ফ্যাক্টোরিয়ালের পেছনের ০

100! এর পেছনে কতগুলো শূন্য আছে? - এটি খুবই প্রচলিত একটি প্রশ্ন। তোমরা হয়তো ইতোমধ্যেই এর সমাধান জানো। যারা জানো না, তাদের জন্য বলি আমাদের বসে বসে 100! এর মতো এত বিশাল সংখ্যা বের করার দরকার নেই। শুধু আমাদের জানতে হবে যে কতগুলো 10 গুণ করা হচ্ছে। কিন্তু একটু খেয়াল করলে দেখব যে, $5! = 120$. এখানে আমরা কোনো 10 গুণ করিনি, এর পরও একটি 0 চলে এসেছে। কেন? কারণ আমরা 5 আর 2 গুণ করেছি, আর এরা আমাদের 10 দিয়েছে। অর্থাৎ আমাদের দেখতে হবে এই গুণফলের মধ্যে কতগুলো 2 আর কতগুলো 5 গুণিতক আকারে আছে। আসলে 2 কতবার আছে তা দেখার দরকার হয় না, কারণ 2 সবসময় 5 এর থেকে বেশি বার থাকবেই, সূতরাং আমাদের 5 গুণলেই চলবে। এখন আমরা চিন্তা করি, 5 কতবার আছে তা কীভাবে বের করব। 1 থেকে 100 এর মধ্যে সর্বমোট $\lfloor 100/5 \rfloor = 20$ টি 5 এর গুণিতক আছে।

¹ $x = (x \bmod M + M) \bmod M$

এগুলো থেকে একটি করে 5 পাব। কিন্তু যেগুলো 25 দিয়ে ভাগ যায় তাদের থেকে কিন্তু আরও একটি করে 5 পাব, আবার যেগুলো 125 দিয়ে বিভাজ্য তাদের থেকে আরও একটি করে পাব। কিন্তু 125 কিন্তু আমাদের 100 থেকে বড় তাই আমাদের আর 5 এর বড় ঘাতগুলোকে দেখার প্রয়োজন নেই। সুতরাং আমাদের 5 এর মোট সংখ্যা $[100/5] + [100/25] = 20 + 4 = 24$. অর্থাৎ 100! এর পেছনে মোট 24 টি শূন্য থাকবে। আমরা যদি $n!$ এর ভেতরে কোনো একটি মৌলিক সংখ্যা p করগুলো আছে সেটা বের করতে চাই তাহলে আমাদের সূত্র হচ্ছে:

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

৩.২.২ ফ্যাক্টোরিয়ালের অঙ্ক (Digit) সংখ্যা

$n!$ এর পেছনের 0 এর সংখ্যা না হয় বুদ্ধি করে বের করা গেল, কিন্তু $n!$ এ করগুলো অঙ্ক বা digit আছে তা কীভাবে বের হবে? তোমরা যদি কোনো একটি ক্যালকুলেটরে 50! করে দেখ তাহলে হয়তো $3.04140932 \times 10^{64}$ এরকম সংখ্যা দেখতে পাবে। এখান থেকে বুঝতে পারছি যে 3 এর পরও আরও 64টি সংখ্যা আছে। এখন আমাদের জানা এমন কি কোনো ফাংশন আছে যেটা ওই 10 এর উপরে থাকা ঘাতটি আমাদের বলে দেবে? আছে, \log (10 ভিত্তিক)। তোমরা তোমাদের ক্যালকুলেটরে যদি এই সংখ্যার \log নাও তাহলে দেখবে 64.4830... এরকম একটি সংখ্যা আসবে। সুতরাং আমরা যদি এর floor নিয়ে এক যোগ করি তাহলেই আমরা অঙ্কসংখ্যা পেয়ে যাব।^১ আমরা যেকোনো সংখ্যার অঙ্কসংখ্যা বের করতে চাইলে এই পদ্ধতি কাজে লাগে। তোমাদের হয়তো কেউ কেউ ভাবছ, \log নেওয়ার জন্য তো আমাদের আগে 50! বের করতে হবে এর পর না $\log!$ না, \log এর একটি সুন্দর বৈশিষ্ট্য আছে আর তা হলো: $\log(a \times b) = \log a + \log b$ অর্থাৎ $\log 50! = \log 1 + \log 2 + \dots + \log 50$.

৩.২.৩ সমাবেশ (Combination): $\binom{n}{r}$

n টি জিনিস হতে r টি জিনিস করতাবে নির্বাচন করা যায় তার সূত্র হলো: $\binom{n}{r} = \frac{n!}{(n-r)!r!}$. অনেক সময়ই n ও r এর মান দেওয়া থাকলে আমাদের $\binom{n}{r}$ এর মান নির্ণয় করার দরকার হয়। একটি উপায় হলো ফ্যাক্টোরিয়ালগুলোর মান আলাদা আলাদা করে নির্ণয় করে গুণ ভাগ করা। কিন্তু এতে একটা সমস্যা হয় আর তা হলো ওভারফ্লো। যেমন $n = 50, r = 1$ হলে $50!$ বের করতে গেলে আমাদের মানটা ওভারফ্লো করবে কিন্তু আমরা জানি $\binom{50}{1} = 50$. তোমরা যদি ভেবে থাক যে double ডেটাটাইপ ব্যবহার করবে, তাহলে সেটা সম্ভব না। কারণ double ডেটা টাইপ তোমাকে মানের একটা আসন্ন মান দিবে, কখনই প্রকৃত মান দিবে না।^২ অনেকে যা করে তা হলো, $(n - r)!$

^১ceiling নিলে হবে না। যদি আমাদের সংখ্যাটি 10^x ধরনের সংখ্যা হয় তাহলে আমাদের উভয়টি সঠিক আসবে না।

^২যদি 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 আসন্ন মান। এটা নিয়ে অবশ্য একটি সত্যিকার কৌতুক আছে। একদিন কনটেন্টের

বা $r!$ এর মধ্যে যেটা বড় তা দিয়ে $n!$ এর সঙ্গে আগেই কাটাকাটি করে ফেলে, এর পর উপরের গুলো গুণ এবং নিচের গুলো গুণ করে এই দুটি সংখ্যা ভাগ করে ফলাফল পাওয়া যায়। ফলে উপরের উদাহরনে উপরে শুধু 50 থাকে আর নিচে থাকে 1 ফলাফল 50। কিন্তু এই পদ্ধতিতেও উপরেরটি ওভারফ্লো করে যেতেই পারে যেখানে হয়তো ভাগ দেওয়ার পর উত্তরটা আর ওভারফ্লো করবে না। আরও একটি উপায় আছে আর তা হলো, আমরা জানি যে $\binom{n}{r}$ সবসময় একটি পূর্ণ সংখ্যা। এর মানে নিচে যেসব সংখ্যা আছে তারা সবাই কাটাকাটির সময় কাটা পরবে। তোমরা উপরের সংখ্যাগুলোকে একটি অ্যারেতে নাও। এর পর একে একে নিচের সংখ্যা নাও আর ওই অ্যারের প্রথম থেকে শেষ পর্যন্ত যাও, গ.সা.গ. নির্ণয় করবে আর এই দুটি সংখ্যাকে কাটবে যতক্ষণ না নিচ থেকে নেওয়া সংখ্যাটা 1 হয়ে যায়। সবশেষে উপরের সব সংখ্যা গুণ করলেই উত্তর পেয়ে যাবে। এভাবে কাজ করলে তোমার উত্তর কখনই ওভারফ্লো করবে না। আরেকটা উপায় হলো আগের মতো প্রথমে $(n - r)!$ দিয়ে উপরে নিচে ভাগ করে ফেল। এবার উপরের সংখ্যাগুলো থেকে সবথেকে ছোটটি নিয়ে উত্তরের সাথে গুণ কর, এর পর নিচের ছোটটি নিয়ে ভাগ কর, আবার উপরের ছোটটি নাও, এরপর নিচের ছোটটি নাও, এরকম করে গুণ ভাগ করতে পার। এরকম নানা ভাবে তোমরা $\binom{n}{r}$ এর মান নির্ণয় করতে পার, প্রতিটি পদ্ধতিরই সুবিধা অসুবিধা আছে, ওভারফ্লো, টাইম কমপ্লেক্সিটি, মেমোরী কমপ্লেক্সিটি, ইমপ্রিমেন্টেশন কমপ্লেক্সিটি ইত্যাদি। সমস্যার উপর ভিত্তি করে তোমাদের বিভিন্ন পদ্ধতি অবলম্বন করার দরকার হতে পারে।

আগেই দেখেছি অনেক সমস্যায় একেবারে সঠিক মান না চেয়ে কোনো একটি সংখ্যা দিয়ে mod করার পরের মান চেয়ে থাকে। এক্ষেত্রেও আমাদের নানা রকম পদ্ধতি আছে। যদি mod করতে বলা সংখ্যাটি মৌলিক সংখ্যা হয় তাহলে $n! \bmod p$, ModuloInverse($r! \bmod p$, p) এবং ModuloInverse($(n - r)! \bmod p$, p) এই তিনটি সংখ্যা গুণ করলেই আমরা আমাদের কাঞ্চিত সংখ্যা পেয়ে যাব। তবে এক্ষেত্রে অবশ্যই $p > n$ হতে হবে। আমরা ফ্যাক্টোরিয়ালের mod মান আগে থেকেই হিসাব করে রেখে মাত্র $O(\log n)$ এ এই মান নির্ণয় করতে পারি। mod যদি মৌলিক সংখ্যক হয় কিন্তু ছোট হয় তাহলে অন্য একটি উপায় আছে এবং সেটি হলো লুকাসের থেওরেম (Lucas' Theorem). এই থেওরেম বলে:

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

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

অর্থাৎ তোমাকে শুধু $\binom{a}{b}$ গুলো জানতে হবে যেখানে $a, b < p$. যদি p বেশ ছোট হয় তাহলে সব $\binom{a}{b}$ কিন্তু তোমরা আগে থেকে হিসাব করে রাখতে পার।

যদি সংখ্যাটা মৌলিক সংখ্যা 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 আগে থেকে হিসাব করে $n \times n$ সাইজের অ্যারেতে রেখে দেই। এটা তৈরি করতে আমাদের $O(n^2)$ সময় লাগে। এই পদ্ধতির মূল সূত্র হলো: $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$, কোড ৩.১০ এ এই কোডটি দেওয়া হলো। তোমরা limncr এর মান পরিবর্তন

id ও password জিজ্ঞাসা করেছিল তানান্দিম ভাই নাফিকে। নাফি বলে pi এর পুরো মান!

^১কোড দেখার আগেই চিন্তা করে দেখ base কেইস কী হওয়া উচিত।

করে দরকার মতো (^n_r) এর মান হিসাব করতে পার।

কোড ৩.১০: ncr.cpp

```
1 ncr[0][0] = 1;
2 int limncr = 10;
3 for(i = 1; i <= limncr; i++)
4     for(j = 0; j <= limncr; j++)
5     {
6         if(j > i) ncr[i][j] = 0;
7         else if(j == i || j == 0) ncr[i][j] = 1;
8         else ncr[i][j] = ncr[i - 1][j - 1] + ncr[i - 1][j];
9     }
```

একটা উদাহরণ দিয়ে সমাবেশের অংশটুকু শেষ করা যাক। মনে কর $R \times C$ আকারের একটি গ্রিড আছে। অর্থাৎ মোট $R \times C$ টি ঘর। এখন তোমাকে নিচের বাম কোণা হতে উপরের ডান কোণায় যেতে হবে। তুমি কেবল উপরে বা ডানে যেতে পার। মোট কতভাবে যেতে পারবে? খেয়াল কর, তুমি যেভাবেই যাও না কেন তোমাকে $R - 1$ বার উপরে যেতে হবে আর $C - 1$ বার ডানে যেতে হবে। অর্থাৎ এই উপর আর ডান যাওয়ার অপারেশনগুলো বিভিন্ন ক্রমে হতে পারে তবে মোট $R - 1$ বার উপরে আর $C - 1$ বার ডানে যেতে হবে। এটি মোট $\binom{(R-1+C-1)}{(R-1)}$ হতে পারে তাই না?

৩.২.৪ কিছু বিশেষ সংখ্যা

কিছু কিছু বিশেষ সংখ্যা আছে যা সমস্যা সমাধান করার সময় আমরা প্রায়ই সমূখীন হই। অনেক সময় এসব মান নির্ণয়ের উপায় আমাদের অন্য সমস্যা সমাধানেও বেশ কাজে লাগে। আমরা এরকম কিছু সংখ্যা এই সেকশনে দেখব।

ডিরেঞ্জমেন্ট সংখ্যা (Derangement Number)

একটি বক্সে n জন তাদের টুপি রাখল। এরপর প্রত্যেকে একটি করে টুপি ওই বাক্স থেকে তুলে নিল। কত উপায়ে তারা এমনভাবে টুপি তুলে নিবে যেন কেউই তাদের নিজেদের টুপি না পায়! যেমন যদি $n = 3$ হয় তাহলে উভয় $2. BCA$ এবং CAB এই দু উপায়েই হতে পারে। আশা করি বুঝতে পারছ যে প্রথম জনের টুপি A , দ্বিতীয় জনের টুপি B ও তৃতীয় জনের টুপি C . আর BCA মানে হলো প্রথমজন পেয়েছে দ্বিতীয় জনের টুপি, দ্বিতীয় জন পেয়েছে তৃতীয় জনেরটা আর তৃতীয়জন পেয়েছে প্রথম জনের টুপি।। এখন এই সমস্যা সমাধান করবে কীভাবে? প্রথমে এটি অনেক সহজ মনে হলেও আসলে এই সমস্যাটার সমাধান একটু জটিল। তোমরা চাইলে ইনক্লুশন এক্সক্লুশন নীতি

(inclusion exclusion principle) দিয়েও এটি সমাধান করতে পার কিন্তু এখানে তোমাদের
রিকারেন্স (recurrence) এর মাধ্যমে সমাধান করে দেখান হলো।

মনে কর D_n হলো n তম ডিরেঞ্জমেন্ট সংখ্যা অর্থাৎ n জন মানুষের জন্য উত্তর। এখন এদের
মধ্য থেকে প্রথম জনকে নাও। সে নিজের বাদে অন্য $n - 1$ জনের টুপি নিতে পারে। ধরা যাক সে
 X এর টুপি নিয়েছে। এখন এই X সেই প্রথম জনের টুপি নিতে পারে আবার নাও পারে। যদি প্রথম
জনের টুপি সে নেয় তাহলে বাকি $n - 2$ জন নিজেদের মধ্যে D_{n-2} ভাবে টুপি আদানপ্রদান করতে
পারে সুতরাং এটি হতে পারে: $(n - 1)D_{n-2}$ ভাবে। আর যদি X প্রথম জনের টুপি না নেয় তাহলে
আমরা ধরে নিতে পারি যে ওই টুপির মালিক এখন X এবং তার ওই টুপি নেয়া যাবে না। অর্ধাৎ
এখন আমাদের কাছে $n - 1$ টি মানুষ ও $n - 1$ টি টুপি আছে যারা কেউই নিজেদের টুপি নিতে চায়
না। এই ঘটনা ঘটতে পারে $(n - 1)D_{n-1}$ উপায়ে। এখানে $n - 1$ গুণ হচ্ছে কারণ আমরা X কে
 $n - 1$ উপায়ে নির্বাচন করতে পারি। সুতরাং আমাদের D_n এর রিকারেন্স সূত্র দাঁড়াচ্ছে,

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

আর এর base কেইস কী? আমরা $n = 2$ এর ক্ষেত্রে এই রিকারেন্স D_0 আর D_1 চায়। কিন্তু
 n যদি এর থেকেও ছোট হয় তাহলে ঝণাত্মক এর D এর মান চেয়ে বসে এই রিকারেন্স। সুতরাং
আমাদের base কেইস হবে $n < 2$ অর্থাৎ 0 আর 1. 1 এর ক্ষেত্রে তো সহজ উত্তর 0, কারণ
একজন লোক থাকলে সে তো আর নিজের টুপি না নিয়ে থাকতে পারবে না। কিন্তু যদি $n = 0$ হয়?
তাহলে উত্তর 1. কারণ কোন লোক নাই, সুতরাং টুপি একভাবেই বিতরণ করা যায় আর তা হলো
কাউকে কিছু না দিয়ে। তাহলে কি আমাদের রিকারেন্স $n = 2$ এর জন্য ঠিক উত্তর দিবে? দেখা
যাক, $D_2 = D_0 + D_1 = 1 + 0 = 1$. হ্যাঁ 2 জনের ক্ষেত্রে তো উত্তর 1, এটাতো আমরা জানিই।

কাটালান সংখ্যা (Catalan Number)

- $2n$ আকারের কতগুলো Dyck Word আছে? $2n$ আকারের Dyck Word এ n টি X ও n
টি Y থাকে এবং এর কোনো prefix এ Y এর সংখ্যা X এর থেকে বেশি নয়। যেমন: $n = 3$ এর জন্য Dyck Word গুলো হলো: XXXYYY, XYXXYY, XYXYXY, XXYYXY
এবং XXYYXY.
- n টি opening bracket বা '(' ও n টি closing bracket বা ')' ব্যবহার করে কতগুলি
সঠিক parentheses expression বানানো যায়? যেমন: $n = 3$ এর জন্য: ((())),
()(), ()(), ()() এবং ()()().
- n leaf ওয়ালা কয়টি কমপ্লিট বাইনারি ট্রি (complete binary tree) আছে?
- n বাহু বিশিষ্ট একটি বহুভুজকে কতভাবে triangulate করা যায়?

এককম অনেক প্রশ্নের উত্তর হলো কাটালান সংখ্যা।¹ n তম কাটালান সংখ্যা কে আমরা C_n লিখে থাকি। এর সূত্র হলো:

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

একেও আমরা চাইলে রিকারেন্স আকারে লিখতে পারি, তবে এই সুত্রটি বেশি দরকারি। তোমরা যেকোনো ডিসক্রিট স্থাথমেটিক্স (Discrete Mathematics) বইয়ে এই সূত্র কীভাবে সঠিক তা দেখে নিতে পার।

Stirling Number of Second Kind

n টি আলাদা জিনিসকে k ভাগে যত ভাবে ভাগ করা যায় তাই হলো Stirling Number of Second Kind. একে $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ দিয়ে প্রকাশ করা হয়। মনে কর $n = 3$ ও $k = 2$ এক্ষেত্রে উত্তর কিন্তু 3, (AB, C) , (AC, B) , (BC, A) . এই সমস্যায় (AB, C) আর (C, BA) একই কথা। এখন এর রিকারেন্স বের করার জন্য আমরা কিছুটা ডিরেঞ্জমেন্ট সংখ্যা বের করার মতো করে চিন্তা করব। প্রথমে n টি জিনিস থেকে সর্বশেষ জিনিসটি নেই। এখন এই জিনিসটি একাই একটি ভাগে থাকতে পারে। সেক্ষেত্রে বাকি $n - 1$ টি জিনিস $k - 1$ ভাগে ভাগ করতে হবে (খেয়াল কর, আমরা শুধুমাত্র শেষটাই আলাদা করে নিয়েছি)। এটা সম্ভব $\left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}$ ভাবে। আবার এটাও সম্ভব যে, বাকি $n - 1$ টি জিনিস k ভাগে আছে আর শেষ জিনিসটি এদেরই কোনো একটায় আছে। এই কোনো একটি k টির মধ্যের কোনো একটি। সুতরাং এটি হতে পারে $k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\}$ ভাবে। অর্থাৎ,

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\} + k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\}$$

এখন যেকোনো রিকারেন্স এর base কেইস থাকে। $k = 1$ হলে সব জিনিসকে এক ভাগে কতভাবে ভাগ করা যায়? মাত্র একভাবেই তাই না? আর যদি $k = n$ হয়? অর্থাৎ n টি জিনিসকে n ভাগে কত ভাবে ভাগ করা যায়? একভাবেই। অর্থাৎ base কেইস হলো:

$$\left\{ \begin{matrix} n \\ 1 \end{matrix} \right\} = \left\{ \begin{matrix} n \\ n \end{matrix} \right\} = 1$$

Stirling Number of First Kind

n টি আলাদা জিনিসকে k টি চক্র বা সাইকেল (cycle) এ যত ভাবে ভাগ করা যায় তাই হলো Stirling Number of First Kind. একে $[\begin{smallmatrix} n \\ k \end{smallmatrix}]$ দিয়ে প্রকাশ করা হয়। মনে কর $n = 4$ ও $k = 2$

¹তোমরা চাইলে http://en.wikipedia.org/wiki/Catalan_number হতে আরও interpretation গুলো পড়ে দেখতে পার।

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)). আশা করি এটা বুঝতে পারছ যে, AB আর BA কিন্তু একই সাইকেল (cycle) নির্দেশ করে, কারণ সাইকেলের আলাদা জায়গা থেকে শুরু করলে তুমি BA পাবে। আবার (AB, CD) আর (CD, AB) কিন্তু একই। কারণ সাইকেলের ক্রম কোনো ব্যাপার না। যাই হোক, আমরা stirling number of first kind এর রিকারেন্স ও আগের মতো একইভাবে বের করতে পারি। প্রথমে n টি জিনিস থেকে সর্বশেষ জিনিসটি নেই। এখন এই জিনিসটি একাই একটি সাইকেলে থাকতে পারে। সেক্ষেত্রে বাকি $n - 1$ টি জিনিস $k - 1$ সাইকেলে ভাগ করতে হবে (খেয়াল কর, আমরা শুধুমাত্র শেষটাই আলাদা করে নিয়েছি)। এটা সম্ভব $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$ ভাবে। আবার এটাও সম্ভব যে, বাকি $n - 1$ টি জিনিস k টি সাইকেলে আছে আর শেষ জিনিসটি এই $n - 1$ টির মধ্যে কোনো একটির পর থাকে। সুতরাং এটি হতে পারে $(n - 1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$ ভাবে। অর্থাৎ,

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$$

এখন এর base কেইস কী? $k = 1$ হলে সব জিনিসকে একটি সাইকেলে কত ভাগে ভাগ করা যায়? এটি একটি সাধারণ সমস্যা এর উত্তর $(n - 1)!$ আর যদি $k = n$ হয়? অর্থাৎ n টি জিনিসকে n সাইকেলে কতভাবে ভাগ করা যায়? একভাবেই। অর্থাৎ base কেইস হলো:

$$\begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)!, \quad \begin{bmatrix} n \\ n \end{bmatrix} = 1$$

৩.২.৫ ফিবোনাচি সংখ্যা (Fibonacci Number)

ফিবোনাচি সংখ্যার সঙ্গে তোমরা ইতোমধ্যেই পরিচিত হয়ে গেছ এবং হয়তো $n \leq 10^6$ এর জন্য ফিবোনাচি সংখ্যাও বের করে ফেলেছ (mod সহ)। কিন্তু যদি আরও বড় ফিবোনাচি সংখ্যা বের করতে বলা হয়, ধর $n \leq 10^{18}$ এর জন্য (আমরা কিন্তু mod মান বের করব)? এর জন্য একটি সুন্দর পদ্ধতি আছে। তোমরা যারা ম্যাট্রিক্স (matrix) জানো না তারা একটু ম্যাট্রিক্স পড়ে নিতে পার। ম্যাট্রিক্স ছাড়াও এটি করা যায় কিন্তু ম্যাট্রিক্স ব্যবহার করে এই সমাধানটি একটি general method, সুতরাং তোমরা এই পদ্ধতি ব্যবহার করে আরও অন্য অনেক সমস্যা সমাধান করতে পারবে। ম্যাট্রিক্স ব্যবহার করে আমরা লিখতে পারি:

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

একইভাবে,

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

সুতরাং আমরা লিখতে পারি,

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

তোমরা যদি ভেবে থাক ম্যাট্রিক্স এর ঘাত তুলতে গেলে তো খবর হয়ে যাবে, তাহলে ভুল ভাবছ। কিছুক্ষণ আগেই কিন্তু আমরা BigMod শিখে এসেছি, ওখানে ছিল একটি সংখ্যা, এখানে আছে ম্যাট্রিক্স। সংখ্যা গুণ করার পরিবর্তে তুমি শুধু ম্যাট্রিক্স গুণ করবে তাহলেই হবে। আমি তোমাদের পরামর্শ দিব তোমরা নিজেরাই এই জিনিসটা কোড কর। প্রথম দিকে যদিও কোড করতে একটু সমস্যা হবে কিন্তু দুই একবার নিজে থেকে কোড করলে দেখবে অনেক সহজ হয়ে যাবে। স্টারলিং সংখ্যাও এই পদ্ধতিতে বের করা যায়, তবে সেক্ষেত্রে $k \times k$ আকারের ম্যাট্রিক্স লাগে। সুতরাং বুবতেই পারছ যে k ছোট আর n অনেক বড় হলে এই পদ্ধতিতে তোমরা স্টারলিং সংখ্যার মান বের করতে পারবে। চেষ্টা করে দেখতে পার সমাধান করতে পার কি না।

৩.২.৬ ইনক্লুশন এক্সক্লুশন নীতি (Inclusion Exclusion Principle)

গণনা বা counting এর ক্ষেত্রে অত্যন্ত গুরুত্বপূর্ণ একটি পদ্ধতি হলো ইনক্লুশন এক্সক্লুশন নীতি (Inclusion Exclusion Principle)। মনে কর তোমাকে বলা হলো, 1 হতে 100 পর্যন্ত কতগুলো

সংখ্যা আছে যাদের 2 বা 3 দিয়ে ভাগ যায়? খেয়াল কর, সংখ্যা বিভিন্ন ধরনের হয়। কিছু সংখ্যা আছে শুধু 2 দিয়ে ভাগ যায় এমন, কিছু আছে শুধু 3 দিয়ে ভাগ যায়, কিছু আছে 2 আর 3 দুটি দিয়েই ভাগ যায়, আবার কিছু আছে যা কোনোটা দিয়েই ভাগ যায় না। শুধু 2 দিয়ে কতগুলো ভাগ করা যায় তা বের করা কিন্তু খুব সহজ, $[100/2] = 50 ((1, 2, 3 \dots 50) \times 2)$. একইভাবে শুধু 3 দিয়ে যায় এরকম আছে $[100/3] = 33 ((1, 2 \dots 33) \times 3)$. এখন এদের যোগ করলেই কিছু তোমাদের উভয় পেয়ে যাবে না। কারণ, প্রথম গ্রন্তি 6, 12, 18 ... এরকম কিছু সংখ্যা আছে যার দ্বিতীয় গ্রন্তি আছে। সুতরাং আমরা যদি এদের যোগ করে দেই এই জিনিসগুলো দুইবার করে গণনা করা হয়ে যাবে। একটা সহজ উপায় হলো কোন সংখ্যাগুলো দুটি গ্রন্তি আছে তা বের করে তা বিয়োগ করে দেওয়া। খেয়াল করলে দেখবে, দুটি গ্রন্তি তারাই আছে যারা 2 ও 3 দুটি দিয়েই ভাগ যায় অর্থাৎ 6 দিয়ে ভাগ যায়। 6 দিয়ে ভাগ যায় এরকম মোট $[100/6] = 16$ টি সংখ্যা আছে। সুতরাং আমাদের ফলাফল হবে: $50 + 33 - 16 = 67$. এভাবে গণনা করার পদ্ধতিই হচ্ছে ইনক্লুশন এক্সক্লুশন নীতি। যদি 2টির বেশি সংখ্যা দেওয়া থাকে তাহলে কী করতে হবে একটু সিদ্ধ করে দেখতে পার। মনে কর তোমাকে 2, 3, 5, 7 এরকম চারটি সংখ্যা দেওয়া হলো। তাহলে একটি দিয়ে ভাগ যায় এরকম সংখ্যা যোগ করে, দুটি দিয়ে ভাগ যায় এরকম সংখ্যা আবার যোগ করবে এবং চারটি সংখ্যা দিয়ে ভাগ যায় সংখ্যা গুলি বিয়োগ করবে। অর্থাৎ, বিজোড় সংখ্যার সময় যোগ ও জোড়ের ক্ষেত্রে বিয়োগ করতে হয়। এই ধরনের সমস্যায় সাধারণত টাইম কমপ্লেক্সিটি হয়ে থাকে $O(2^n)$. আরেকটা জিনিস তোমাদের বলে রাখি এসব ক্ষেত্রে বিটমাস্ক (bitmask) ব্যবহার করে কোড করলে অনেক সহজেই কোড হয়ে যায়। যদি তোমাদের কাছে n টি সংখ্যা থাকে এবং কোনটি কোনটি দিয়ে ভাগ করবে এটি সিদ্ধান্ত নিতে চাও তাহলে তুমি n বিটপে বিভিন্ন সংখ্যা নিবে, কোনো একটি বিটে। থাকার মানে ওই সংখ্যা তুমি নিবে, 0 মানে নিবে না। এরকম করে খুব সহজে একটি লুপ দিয়ে তুমি এই নেওয়া-না নেওয়ার কাজটা করে ফেলতে পার।

৩.৩ সন্তান্ত্বতা (Probability) ও এক্সপেক্টেশন (Expectation)

কোনো কারণে আমরা ছোটবেলা থেকে এই জিনিসের প্রতি একরকম ভীতি নিয়ে বেড়ে উঠি কারণ এই জিনিস বুঝতে আমাদের কষ্ট হয় বা আমাদের হয়তো ভালো মতো বোঝানো হয় না। এখানে আসলে খুব বিস্তারিতভাবে তোমাদের এসব জিনিস দেখানো সন্তুষ্ট না কিন্তু খুব অল্প কথা কিছু লিখার চেষ্টা করলাম।

৩.৩.১ সন্তান্ত্বতা (Probability)

মনে কর ক্রিকেট খেলা শুরুর আগে দুই ক্যাপ্টেন টস করতে গেল। টসের সময় Head পড়বে না Tail পড়বে? যেকোনো একটা পড়তে পারে! আমরা বলে থাকি সন্তান্ত্বনা 50 – 50. এখন মনে

কর লুড় খেলায় একটা ছক্কা আছে, কিন্তু এই ছক্কায় কোনো 5 নেই তার পরিবর্তে আরও একটি 6 আছে। এখন তোমাকে যদি জিজ্ঞাসা করা হয় এখানে কত পড়বে? তুমি কিন্তু নিশ্চিতভাবে বলতে পারবে না যে এখানে অমুক সংখ্যাই পড়বে। আবার তুমি একটু যদি গাণিতিক উন্নতি দাও তাহলে বলবে এখানে 5 কখনই পড়বে না, 6 পড়ার সম্ভাবনা $1, 2, 3, 4$ এর কোনও একটি পড়ার থেকে বেশি। আবার এও বলতে পার যে $1, 2, 3, 4$ এদের যেকোনোটি পড়ার সম্ভাবনা একই। আমরা কিন্তু কোনো হিসাব করে এ কথা বলছি না, শুধু বাস্তব বুদ্ধি থেকেই এই কথা বললাম। তাই না? এখন যদি তোমাকে বলা হয় ঠিক মতো হিসাব করে বের কর কোনটা পড়ার সম্ভাবনা কত? এই হিসাবটাই কীভাবে করতে হয় তা এখন দেখা যাক।

সম্ভাব্যতার মূল নীতি হলো:

$$\text{কোনো ঘটনা ঘটার সম্ভাব্যতা} = \frac{\text{কতভাবে এই ঘটনা ঘটতে পারে}}{\text{কতভাবে সকল ঘটনা ঘটতে পারে}}$$

যেমন, আমাদের ক্রিকেট খেলার টসে, Head পড়ার সম্ভাব্যতা হলো: $\frac{1}{2}$ কারণ আমাদের মাত্র দুভাবেই মুদ্রা বা কয়েন পড়তে পারে Head ও Tail আর এদের মধ্যে একটিই Head. এখন আমাদের কিছুক্ষণ আগের ছক্কার ক্ষেত্রে যদি আমরা হিসাব করতে চাই 5 পড়ার সম্ভাব্যতা কত, তাহলে কিন্তু $\frac{0}{6} = 0$. অর্থাৎ 5 পাবই না! আবার $1, 2, 3$ বা 4 পড়ার সম্ভাব্যতা হলো $\frac{1}{6}$ আর 6 পড়ার সম্ভাব্যতা হলো $\frac{2}{6}$. অর্থাৎ 6 পড়ার সম্ভাবনা কিন্তু অন্য কোনো একটি সংখ্যা পড়ার সম্ভাবনা থেকে বেশি।

তোমরা হয়তো π নির্ণয় করার নানা মজার উপায় দেখেছ। এমনই একটি পদ্ধতি এখন বলব। তোমরা একটি বড় বর্গ আঁক। এর মধ্যে একটি বৃত্ত আঁক যা চার বাহুকেই স্পর্শ করে। এখন তুমি ছোট ছোট কিছু জিনিস নাও, মনে কর n টি চুমকির মতো জিনিস। এখন এগুলো বর্গক্ষেত্রের মধ্যে বিক্ষিপ্তভাবে ছড়িয়ে দাও। এখন তুমি গুনে দেখ বৃত্তের মধ্যে কতগুলো আছে। ধরা যাক, b টি। তাহলে $\frac{4b}{n}$ হবে প্রায় π এর সমান। অঙ্গুত না? এমন কেন হলো? এর যুক্তি কিন্তু খুবই সহজ। বৃত্তের radius যদি r হয় তাহলে বৃত্তের ক্ষেত্রফল πr^2 আর বর্গের ক্ষেত্রফল $4r^2$. সুতরাং তুমি যদি কোনো একটি চুমকি বিক্ষিপ্তভাবে ফেল এর মধ্যে তাহলে বৃত্তের মধ্যে পড়ার সম্ভাবনা $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$. আবার আমরা আমাদের পরীক্ষায় n সংখ্যক বিন্দু বিক্ষিপ্তভাবে ফেলেছিলাম এবং বৃত্তের মধ্যে পড়েছে b সংখ্যকটি, সুতরাং আমরা পরীক্ষা থেকে পাই বৃত্তের মধ্যে একটি চুমকি পড়ার সম্ভাবনা $\frac{b}{n}$ অর্থাৎ $\frac{b}{n} \approx \frac{\pi}{4}$ এখান থেকেই আমরা পাই, $\pi \approx \frac{4b}{n}$. তুমি n এর মান যত বড় নিবে এই পরীক্ষা থেকে তত নিখুঁত π এর মান পাবে।

৩.৩.২ এক্সপেক্টেশন (Expectation)

মনে কর তোমাকে বলা হলো একটি কয়েন টস করার পর যদি head পড়ে তাহলে তুমি 0 টাকা পাবে কিন্তু tail পড়লে 100 টাকা পাবে। তুমি কত পাওয়ার আশা কর? তুমি যদি বল যে আমি 100 টাকা পাওয়ার আশা করি, তাহলে হলো না, তুমি বেশি আশাবাদী হয়ে গেলে। কারণ তোমার 50% সম্ভাবনা আছে তুমি 0 টাকা জিতবে। আবার তুমি হতাশার সুরে যদি বল যে নাহ! আমি একটি

টাকাও পাব না, আমার কপাল ভালো না। তাহলেও হলো না। তুমি যদি একজন গণিতবিদের মূলে
কথা বল তাহলে বলবে আমি 50 টাকা পাওয়ার আশা করি। প্রথম প্রথম সবাই মনে করত পার- এটা
কেমন কথা হলো? হয় আমি 0 পাব নাহয় 100 পাব, 50 কোথা থেকে আসল? কাহিনি হলো, তুমি
যদি এই টস 10 বার কর, তাহলে এটি বলাই যায় যে 5 বারের মতো head পড়েছে আর বাকি 5
বার tail. অর্থাৎ তুমি 10 বারে মোট 500 টাকা পাবে, অর্থাৎ গড়ে তুমি প্রতিবার 50 টাকা পাবে।
এই জন্যই এক্ষেত্রে তোমার এক্সপেন্সেন হলো 50 টাকা। আরও একটি উদাহরণ দেওয়া যাক, দ্বি
তুমি সাপ লুড় খেলছ। তোমাকে যদি বলা হয় তোমার খেলা শেষ করতে কত চাল লাগতে পারে?
তুমি কিন্তু হিসাবনিকাশ করে দেখাতে পারবে যে তোমার কতগুলো চাল লাগতে পারে। দেখা যাবে
তুমি যদি বহুবার সাপ লুড় খেল তাহলে গড়ে ওই সংখ্যক চাল লাগবে। কোনো কিছুর এক্সপেন্সেন
বের করার নিয়ম হচ্ছে যত রকম ঘটনা ঘটতে পারে তাদের সম্ভাব্যতা \times ওই ঘটনা ঘটলে তোমার
সেই কোনো কিছু কত হবে। যেমন, কয়েন টসের ক্ষেত্রে তোমার দুই রকম ঘটনা ঘটতে পারে।
head বা tail. head পড়ার সম্ভাব্যতা 0.5 এবং এটি পড়লে তুমি 0 টাকা পাবে। আর যদি 0.5
সম্ভাব্যতাতে tail পড়ে তাহলে তুমি পাবে 100 টাকা। অতএব তোমার টাকা পাওয়ার এক্সপেন্সেন
হবে $0.5 \times 0 + 0.5 \times 100 = 50$.

আরও একটি উদাহরণ হিসেবে চিত্র 3.1 এর ছোট পরিসরে সাপ লুড় বিবেচনা করা যাক।

1	2	3	4	5
---	---	---	---	---

চিত্র 3.1: একটি ছোট লুড় খেলা

মনে কর তুমি 1 এ আছ। তুমি যদি 5 এ যাও তাহলেই খেলা শেষ হয়ে যাবে। মধ্যের গাড়ি
রঙওয়ালা 3 এ তুমি যেতে পারবে না কখনই। তোমাকে একটি ছক্কা দেওয়া হলো যেটা ছুড়লে।
হতে 6 এর মধ্যের কোনো একটি সংখ্যা পড়ে এবং তুমি তত সংখ্যক ঘর সামনে যেতে পারবে।
কিন্তু সেই ঘর যদি 3 হয় বা 5 পেরিয়ে যায় তাহলে আবারও তোমাকে চালতে হবে। তুমি যখন 5-এ
পৌঁছাবে তখন খেলা শেষ হবে। এখন এই খেলা শেষ করতে কয়টি চাল লাগতে পারে অর্থাৎ চাল
সংখ্যার এক্সপেন্সেন কত?

ধরা যাক, T_i হলো i এ থাকাকালীন সময়ে খেলা শেষ করার জন্য কতটি চাল লাগবে তার
এক্সপেন্সেন। আমাদের T_1 বের করতে হবে। শেষ থেকে আসা যাক। $T_5 = 0$ কারণ তুমি যদি 5
এ থাকো তাহলে তো খেলা শেষ। কোনো চাল 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 চাল।
খেয়াল কর, যদি 1 ব্যতীত কোনো সংখ্যা পড়ে তাহলে কিন্তু তুমি 4 এই থাকবে, অর্থাৎ তোমার
আরও T_4 সংখ্যক চাল লাগবে। ব্যাপারটা আরও একটু পরিষ্কার করা যাক, তুমি ধরেই নিয়েছ যে
থেকে খেলা শেষ করতে T_4 সংখ্যক চাল লাগবে। এখন তোমার যদি এখানেই থাকতে হয় তাহলে

¹এখানে তো সাপ নাই কিন্তু একটি বাধা আছে। আমরা কী একে পাহাড় লুড় বলব?

তো T_4 সংখ্যক চাল লাগবে তাই না? আর যদি 1 পড়ে তাহলে লাগবে T_5 সংখ্যক চাল। অর্থাৎ $1/6$ সন্তাব্যতায় লাগবে T_5 সংখ্যক চাল আর $5/6$ সন্তাব্যতায় লাগবে T_4 সংখ্যক চাল। আর ভুলে যেও না এই মাত্র তুমি একটা চাল দিলে ছক্কা গড়িয়ে, সূতরাং $T_4 = 1 + \frac{1}{6}T_5 + \frac{5}{6}T_4$ এখান থেকে আমরা পাই, $T_4 = 6$. হিসাবটি কিন্তু ঠিকই আছে। ছক্কার ছয় দিক, আমরা আশা করতেই পারি যে 6 চালের মধ্যে প্রতিটি সংখ্যা একবার না একবার আসবেই। সূতরাং আমাদের এক্সপেন্শন 6. T_3 আমাদের লাগবে না, কারণ আমরা এখানে কখনোই আসব না। এখন আসা যাক, 2 এ। যদি $1/6$ সন্তাব্যতায় 2 পড়ে তাহলে লাগবে T_4 সংখ্যক চাল, যদি $1/6$ সন্তাব্যতায় 3 পড়ে তাহলে T_5 সংখ্যক চাল লাগবে, আর বাকি $4/6$ সন্তাব্যতায় যা পড়বে তার জন্য আমাদের 2 এই থাকতে হবে অর্থাৎ T_2 সংখ্যক চাল লাগবে। সূতরাং, $T_2 = 1 + \frac{1}{6}T_4 + \frac{1}{6}T_5 + \frac{4}{6}T_2$ অর্থাৎ $T_2 = 6$. আশা করি T_1 এর জন্য সুত্রটি কী হবে তুমি বুঝতে পারছ, $T_1 = 1 + \frac{1}{6}T_2 + \frac{1}{6}T_4 + \frac{1}{6}T_5 + \frac{3}{6}T_1 \Rightarrow T_1 = 6$. অর্থাৎ 1 এ থাকা কালীন সময়ে খেলা শেষ করতে চাল সংখ্যার এক্সপেন্শন হবে 6.

3.8 বিবিধ

3.8.1 ভিত্তি পরিবর্তন (Base Conversion)

আমরা যেই সংখ্যা পদ্ধতি ব্যবহার করে থাকি তাকে দশমিক বলে কারণ এর ভিত্তি (base) হলো 10. কম্পিউটার যেই সংখ্যা পদ্ধতি ব্যবহার করে তার ভিত্তি হলো 2, একে বলে বাইনারি। এরকম আরও কিছু বহুল প্রচলিত সংখ্যা পদ্ধতি আছে- অকটাল (যার ভিত্তি হলো 8), হেক্সাডেসিমাল (যার ভিত্তি হলো 16)। বলা হয়ে থাকে আমাদের হাতের দশ আঙুলের জন্য আমাদের সংখ্যা পদ্ধতি হলো Decimal বা দশমিক। কম্পিউটারের জন্য 10টি আলাদা আলাদা সংখ্যা নিয়ে হিসাব করা বেশ কষ্টকর এজন্য শুধু মাত্র ভোল্টেজ আপ-ডাউন করে বোঝা যায় এমন একটি সংখ্যা পদ্ধতি ব্যবহার করা হয় কম্পিউটারে। এটিই হলো বাইনারি। এই পদ্ধতিতে অঙ্ক আছে দুটি 0 ও 1. আমরা কীভাবে হিসাব করি একটু খেয়াল কর: $0, 1, \dots, 9, 10, 11, 12, \dots, 19, \dots$ অর্থাৎ আমাদের শেষ অঙ্কটি এক এক করে বাড়ে এর পর শেষ হয়ে গেলে এর বামেরটি এক বাড়ে ওটাও শেষ হয়ে গেলে তারও বামেরটি বাড়বে। বাইনারিও সে রকম: $0, 1, 10, 11, 100, 101, 110, 111, 1000, \dots$ । অন্যান্য সংখ্যা পদ্ধতিতেও একই রকম হয়ে থাকে।

এখন যদি একটু খেয়াল কর দশমিক সংখ্যা পদ্ধতিতে 481 কে আমরা ভেঙে লিখতে পারি: $4 \times 10^2 + 8 \times 10^1 + 1 \times 10^0$ একইভাবে বাইনারি 1010 কেও আমরা এভাবে ভেঙে লিখতে পারি: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ যার মান দশমিক পদ্ধতিতে হবে 10. সূতরাং আমাদের যদি অন্য কোনো সংখ্যা পদ্ধতিতে একটি সংখ্যা দেওয়া হয় তাকে আমরা খুব সহজেই আমাদের দশমিক সংখ্যা পদ্ধতিতে পরিবর্তন করতে পারি। আমরা ডান থেকে তম স্থানে যাব এবং সেখানে থাকা অঙ্ককে $base^i$ দিয়ে গুণ করে সবগুলো যোগ করলেই আমরা দশমিক পদ্ধতিতে সংখ্যাটি পেয়ে যাব।

কিন্তু আমরা যদি কোনো একটি দশমিক সংখ্যাকে অন্য আরেকটি সংখ্যা পদ্ধতির সংখ্যায়

পরিবর্তন করতে চাই? ধৰা যাক আমরা b ভিত্তির সংখ্যায় পরিবর্তন করতে চাই। তাহলে অবশ্যই আমাদের সংখ্যাটা হবে এরকম: $a_n \times b^n + \dots + a_1 \times b^1 + a_0 \times b^0$ আমরা যদি এই সংখ্যাকে b দিয়ে ভাগ করি তাহলে ভাগশেষ হবে a_0 এবং ভাগ করার ফলে সব ঘাতগুলো কিন্তু 1 করে কমে গেছে, সুতরাং এর পরে আবারও b দিয়ে ভাগ করলে ভাগশেষ হবে a_1 এভাবে একে একে আমরা ডান থেকে বামের সব সংখ্যা পেয়ে যাব। উদাহরণ দেয়া যাক, কিছুক্ষণ আগেই দেখেছি দশমিকে 10 হলো বাইনারির 1010. আমাদেরকে 10 দেয়া থাকলে এই 1010 কেমনে পেতে পারি? প্রথমে 2 দিয়ে ভাগ করব, ভাগশেষ 0, আর ভাগফল 5. আবার 5 কে 2 দিয়ে ভাগ, এবার ভাগশেষ 1 আর ভাগফল 2. আবার ভাগ করলে, ভাগশেষ 0 ও ভাগফল 1. শেষ বারের মত ভাগ করলে ভাগশেষ 1 আর ভাগফল 0. ভাগফল 0 হয়ে গেলে আমাদের আর ভাগ করার কোনো মানে নেই। এবার যেই ভাগশেষগুলো পেয়েছ তাদের উলটো দিক থেকে লিখ তাহলেই 1010 পেয়ে যাবে।

৩.৪.২ বিগ ইন্টিজার (Big Integer)

অনেক সময় দেখা যায় আমাদের প্রবলেমে mod করতে বলা হয় না, আবার আমাদের উত্তরটা বেশ বড় হয়। সেক্ষেত্রে আমাদের বিগ ইন্টিজার (BigInteger) ব্যবহার করতে হয়। যারা জাভা প্রোগ্রামিং ভাষা (Java programming language) জানো তাদের জন্য এটা একটা বাড়ি সুবিধা কারণ জাভাতে BigInteger নামে একটি লাইব্রেরী আছে। আমরা কিন্তু খুব সহজেই C তে নিজেদের BigInteger এর হিসাবনিকাশের জন্য ফাংশন লিখে ফেলতে পারি। যোগ, বিয়োগ ও গুণ বেশ সহজ। ভাগ একটু কঠিন। খেয়াল করলে দেখবে যে, আমরা কাগজে-কলমে যোগ, বিয়োগ বা গুণ সব সময় ডানদিক থেকে করি, এবং এই সময় যেই দুইটি সংখ্যা নিয়ে হিসাব করছি তাদেরকে ডানদিকে একই বরাবর রাখা হয়, আমরা কিন্তু বাম দিকে একই বরাবর রাখি না, রাখলে ভুল হবে। কিন্তু আমরা যখন কম্পিউটারে অ্যারেতে এসব সংখ্যা রাখার কথা চিন্তা করব তখন আমরা কল্পনা করি বামদিক থেকে। এটা অনেক সময় সমস্যা হয়। যেমন মনে করা যাক আমরা 100 অঙ্কের একটি সংখ্যার সঙ্গে 50 অঙ্কের একটি সংখ্যা যোগ করব। এখন এই দুটি সংখ্যা যখন স্ট্রিং আকারে ইনপুট নিব তখন এটা বামদিকে align হয়ে থাকে। কারণ দুজনেরই বামের অক্ষ বা Most Significant Digit (MSB) 0 ইনডেক্স থেকে শুরু হয়। (চিত্র ৩.২।

'1'	'1'	'8'	'3'
'7'	'5'		

চিত্র ৩.২: 1183 আর 75 দুইটি string এ আছে

এই অবস্থায় কোনো হিসাব করা বেশ কঠিন। আমাদেরকে ডানদিকে বরাবর align করতে হবে। এ জন্য যা করা উচিত তা হলো, সংখ্যাকে উল্টে নেওয়া, এতে করে সংখ্যার এককের অক্ষ

সবসময় 0 ইনডেক্সে থাকে, আর যেহেতু আমরা কম্পিউটারে সংখ্যাকে বাম দিকে align করে চিন্তা করছি সেহেতু আর কোনো সমস্যা হবে না। আর আমাদের অ্যারে এর বাকি ঘরগুলো 0 দিয়ে পূরণ করতে হবে। এতে কিন্তু কোনও সমস্যা হবে না, কারণ 57 আর 000057 তো একই কথা তাই না? চিত্র ৩.৩ এ এই পরিবর্তিত অবস্থা দেখানো হলো।

3	8	1	1	0	0	0
5	7	0	0	0	0	0

নকশা ৩.৩: 1183 আর 75 দুইটি উল্টো করে রাখা হয়েছে

অর্থাৎ আমরা দুটি সংখ্যার এককের ঘরের অঙ্ক বরাবর align করে ফেলেছি। আমরা এখন বামদিক থেকে ডানদিকে যাব আর হিসাব করব। এসময় আরও একটি গুরুত্বপূর্ণ জিনিস খেয়াল রাখলে ভালো হয়। আর তা হলো, আমরা যেহেতু জানি না যে আমাদের উত্তরটা কতগুলি অঙ্ক হবে সেহেতু আমাদের ফলাফল রাখার অ্যারেকে 0 দিয়ে ইনিশিয়ালাইজেশন (initialize) করে নিতে হবে। এতে করে যোগ, বিয়োগ বা গুণ করতে কোনো সমস্যা হবে না। এখন কথা হলো কীভাবে যোগ, বিয়োগ গুণ করব? খুবই সোজা, তুমি যেভাবে কাগজে কলমে কর ঠিক সেভাবে। তুমি নিজে একটু ভেবে দেখ কীভাবে যোগ, বিয়োগ কর? তুমি প্রথমে এককের ঘরের অঙ্কগুলি যোগ কর, এর পর দশকের এভাবে। যোগ করার পর যোগফল 10 বা এর বড় হলে হাতে কিছু একটা থাকে, সেটা গিয়ে পরের ঘরের সঙ্গে যোগ কর এভাবে চলতে থাকে। আবার বিয়োগের সময় তুমি কিছু ধার নাও যেটা পরে গিয়ে আবার শোধ করে দাও। ঠিক এই জিনিসগুলোই তোমাদের লজিকের মাধ্যমে লিখতে হবে। গুণের ক্ষেত্রে তোমাদের একটু ঝামেলা মনে হতে পারে। তোমরা একটু চিন্তা করে দেখ আমরা যে গুণ করে সংখ্যাগুলো পর পর লিখি এর পর যোগ করি তা না করে, যদি গুণ করতে করতে যোগ করি? এটা আমাদের হাতে হাতে করতে বেশ কষ্ট হবে, কিন্তু কম্পিউটারে এই প্রোগ্রাম লেখা বেশ সহজ হবে। গুণ করতে করতে যোগ কথাটা আসলে খুব একটা পরিষ্কার না। কিন্তু কীভাবে বুবাব ঠিক বুবতেও পারছি না। তাও চেষ্টা করে দেখা যাক। মনে কর আমরা চাচ্ছি 1111 এর সাথে 234 গুণ করতে। প্রথমে আমরা যা করব তা হলো গুণফলের সাথে 4444 যোগ করব (প্রথমে গুণফল ছিল 0)। এর পর যোগ করব 33330 বা দ্বিতীয় ঘর থেকে 3333. এর পর তৃতীয় ঘর থেকে 2222. এখন এইয়ে আমরা 4444 বা 3333 বের করছি এটা আসলে একবারে বের করে তারপর যোগ না করে বরং বের করার সময় যোগ করতে থাকতে পার। মানে মনে কর যখন তুমি এককের 4 বের করে ফেলেছ তখন তা যোগ করে ফেল, এর পর পরের 4 কে পরের ঘরের সাথে এভাবে আর যোগ করার মাঝে carry এর ব্যাপার (যোগ করার সময় হাতে যা থাকে তাকে carry বলে) ভুলে যেও না। মোটকথা BigInteger এর যোগ, বিয়োগ, গুণ এসব আসলে বাস্তব বুদ্ধি থেকে করার বিষয়। BigInteger এর ভাগও করা সম্ভব কিন্তু এটা বেশ ঝামেলার সেজন্য আমি পারতপক্ষে এটা কোড করতে যাই না। এসব ক্ষেত্রে আমি জাভাতে কোড করে ফেলি।

৩.৪.৩ চক্র বা সাইকেল (Cycle) নির্ণয়ের অ্যালগরিদম

UVa 11036 প্রবলেমটা দেখতে বেশ কঠিন হলেও এর ধারণাটি কিন্তু খুব সহজ। একটা x এর ফাংশন দেওয়া থাকবে যেমন ধরাযাক, $f(x) = x*(x+1) \bmod 11$, এখন একটি n এর মানের জন্য $f(n), f(f(n)), f(f(f(n))) \dots$ এই ধারার পিরিয়ড (period) বের করতে হবে। ধারার পিরিয়ড মানে হলো কতক্ষণ পর পর এই ধারার মান পুনরাবৃত্তি হবে। যেমন যদি $n = 1$ হয় তাহলে এই ধারার মানগুলো হবে: $2, 6, 9, 2, 6, 9, 2, 6, 9 \dots$ এখানে তিনটি সংখ্যা বার বার পুনরাবৃত্তি করছে, সুতরাং এখানে পিরিয়ড হবে 3. একটু চিন্তা করলে দেখবে যে এখানে আসলে কথনো যদি আগের একটি সংখ্যা আসে, তাহলে এর পরের সংখ্যাগুলো আগের মতো 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 পুনরাবৃত্তি কখন হবে। এই জিনিসটা একটা অ্যারে রেখে খুব সহজেই করা যায়। আমরা একটি একটি করে মান বের করব আর অ্যারে তে দেখব যে এই জিনিসটি আগে এসেছিল কি না। যদি ন আসে তাহলে আমরা পরবর্তীতে ব্যবহারের জন্য অ্যারেতে লিখে রাখব যে এই সংখ্যাটা ধারার অনুক স্থানে এসেছিল। আর যদি আগেই এসে থাকে তাহলে, আগে কোন জায়গায় এসেছিল তা আমরা অ্যারে থেকেই দেখতে পারব, আর এখন কোন পদে এলাম তাও আমরা জানি। এই দুই সংখ্যা থেকে আমরা এর ধারার পিরিয়ড বের করে ফেলতে পারি। যেমন উপরের উদাহরণে যখন প্রথম 2, 6 আর 9 পেলাম তখন লিখে রেখেছি যে আমরা 2 পেয়েছি প্রথম স্থানে, 6 পেয়েছি দ্বিতীয় স্থানে আর 9 পেয়েছি তৃতীয় স্থানে। এর পরে যখন আবার চতুর্থ স্থানে 2 পেলাম তখন অ্যারে থেকে দেখতে পাচ্ছি যে এটা আগে প্রথম স্থানে এসেছিল। এর মানে প্রথম স্থান হতে তৃতীয় স্থান আসলে ধারার পিরিয়ড। কিন্তু এভাবে অ্যারে ব্যবহার করে করতে গেলে আমাদের মেমোরী কমপ্লেক্সিটি দাঁড়ায় $O(N)$ যেখানে N হলো সর্বোচ্চ সম্ভাব্য মান। যদিও আমাদের টাইম কমপ্লেক্সিটি $O(n)$ বা আরও সুনির্দিষ্টভাবে বলতে গেলে, $O(\lambda + \mu)$ যেখানে $\mu =$ শুরু থেকে সাইকেলের শুরু পর্যন্ত দূরত্ব এবং $\lambda =$ সাইকেলের দৈর্ঘ্য। আমরা চাইলে মেমোরী কমপ্লেক্সিটি কমিয়ে $O(1)$ এ নামাতে পারি এবং সেক্ষেত্রেও আমাদের টাইম কমপ্লেক্সিটি একই থাকবে। এই পদ্ধতিকে বলা হয় ফ্লয়েডের সাইকেল নির্ণয়ের অ্যালগরিদম (Floyd's Cycle Finding Algorithm).

এই অ্যালগরিদমটি বেশ মজার। আমরা যেই ধারার সাইকেল বের করতে চাচ্ছি সেই ধারার শুরুতে একটি খরগোশ আর একটি কচ্ছপ রাখতে হবে। এর পর প্রতি বারে খরগোশ দুই ধাপ আর কচ্ছপ এক ধাপ করে যাবে (দুই ধাপ যাবে মানে $f(f(x))$ আর এক ধাপ যাওয়া মানে $f(x)$)। এক সময় না এক সময় দুজনে মিলিত হবেই (মিলিত হওয়ার মানে হলো দুজনের একই মান)। এখন খরগোশকে ওই জায়গাতে রেখেই কচ্ছপকে একধাপ একধাপ করে এগিয়ে নিতে হবে যতক্ষণ না সে আবার খরগোশের জায়গায় ফেরত আসে। যত ধাপে সে ফেরত আসে সেটাই হলো ধারার পিরিয়ড বা λ . এবার কচ্ছপ কে ওই জায়গাতে রেখে খরগোশ কে আবার শুরুতে নিয়ে যাও তবে এবার খরগোশ আর কচ্ছপ দুজনই একধাপ একধাপ করে এগোবে যতক্ষণ না একত্র হয়। এটা খুব সহজেই প্রমাণ করা যায় যে, যে কয় ধাপে মিলিত হলো সেটাই μ . খুবই অস্ত্রুত একটা অ্যালগরিদম!

মনে করো তোমাকে দেওয়া আছে $x + 2y = 5$ আর $3x + 2y = 7$. তোমাকে বলতে হবে x এবং y এর মান কত বা বলতে হবে এদের কোনো সমাধান নেই বা অসীম সংখ্যক সমাধান আছে? ^১ দুটি ভ্যারিয়েবল হলে তো জিনিসটি খুবই সহজ তাই না? হাতে হাতে করা যায়। কিন্তু যদি তোমাকে n টি ভ্যারিয়েবলওয়ালা n টি সমীকরণ দেয়? সত্যি কথা বলতে প্রথম যখন আমি নিজে এরকম সমস্যার সমুখীন হয়েছিলাম তখন বেশ ভয় লেগেছিল এবং বুঝছিলাম না যে এত কঠিন সমস্যা কীভাবে সমাধান করা যায়। যদি আমি ভুল না করে থাকি তাহলে সেটি বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয়ের কোনো এক প্র্যাকটিস কনটেক্টে ছিল। আমি আর নাফি দুজন একদলে ছিলাম আর অন্য দলের মধ্যে ছিল সেবারের বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয়ের ওয়ার্ল্ড ফাইনালিস্ট দল, যত দূর সন্তুষ্ট ডলার ভাই, মাহমুদ ভাই এবং সানি ভাইয়ের দল। কনটেক্টটায় কতগুলো সমস্যা ছিল ঠিক মনে নেই তবে উনারা 4 – 5টি সমস্যার সমাধান করেছিলেন যার একটিও আমরা পারিনি কিন্তু এই সমস্যাটার সমাধান আমরা করেছিলাম যা উনারা করেননি! সুতরাং বুঝতেই পারছ এই সমস্যাটা অতটো কঠিন নয়। সত্যি কথা বলতে খুবই সহজ। আমাদের স্কুল বা কলেজের গণিতের জ্ঞান দিয়েই এর সমাধান সন্তুষ্ট। ধর তোমাকে যদি 4টি ভ্যারিয়েবলের 4 টি সমীকরণ দেয় তাহলে তুমি হাতে হাতে কীভাবে সমাধান করবে? মনে কর ভ্যারিয়েবলগুলো হলো x_0, x_1, x_2 এবং x_3 . তুমি যা করবে তা হলো প্রথম সমীকরণ নিয়ে তাতে x_0 এর মান বের করে বাকিগুলোতে বসায়ে দাও, তাহলে কী হলো? বাকি 3 টি সমীকরণে 3 টি করে ভ্যারিয়েবল থাকবে। আবার তুমি এই কাজ করলে দুটি সমীকরণে দুটি ভ্যারিয়েবল থাকবে, এবং আরেকবার করলে একটি ভ্যারিয়েবল ও একটি সমীকরণ। এবার এই মান আগের সমীকরণগুলোতে বসাতে থাকলে একে একে সবগুলোর মান পেয়ে যাবে। এটাই হলো মূল ধারণা। তবে এই জিনিস হাতে হাতে করা যত সহজ কোডে করা অতটো সহজ হবে না। এই পদ্ধতিকে অন্য আরেকভাবে চিন্তা করতে পার। প্রথম সমীকরণ নাও, এটা ব্যবহার করে অন্য সব সমীকরণ থেকে প্রথম ভ্যারিয়েবল কে সরিয়ে দাও। এবার দ্বিতীয় সমীকরণ নাও আর সব সমীকরণ হতে দ্বিতীয় ভ্যারিয়েবলকে সরিয়ে দাও। এভাবে চলতে থাকলে প্রতিটি সমীকরণ এ একটিমাত্র ভ্যারিয়েবল বাকি থাকবে এবং তুমি তা থেকে তার মান বের করে ফেলতে পারবে। এখানে ভ্যারিয়েবল সরিয়ে দেওয়া করা মানে হলো যোগ, বিয়োগ আর গুণ করে কোন এক ভ্যারিয়েবল দূর করে ফেলা। মনে কর তোমার প্রথম সমীকরণ $x + 2y = 5$ আর দ্বিতীয় সমীকরণ $3x + 2y = 7$. আমরা চাচ্ছি প্রথম সমীকরণ ব্যবহার করে দ্বিতীয় সমীকরণ হতে প্রথম ভ্যারিয়েবল (x) কে দূর করতে। আমাদের যা করতে হবে তা হলো প্রথম সমীকরণ কে 3 দিয়ে গুণ করে দ্বিতীয় সমীকরণ থেকে বাদ দেয়া। তাহলে দ্বিতীয় সমীকরণ দাঁড়াবে $-4y = -8$. এটাই হলো ভ্যারিয়েবল দূর করা যাকে ইংরেজীতে বলে ভ্যারিয়েবল এলিমিনেশন (variable elimination). কিন্তু এভাবে করলে কিছু কিছু বিশেষ কেইস এসে পড়বে। যেমন দ্বিতীয় সমীকরণে এসে দেখলে যে এর দ্বিতীয় ভ্যারিয়েবল নেই, অর্থাৎ এর সহগ বা কো-এফিষিয়েন্ট (coefficient) হলো 0. এসব কেইস এড়িয়ে কীভাবে সহজে এটা সমাধান করা যায় তা আমরা এখন দেখব।

^১ ধর তোমার সমীকরণ হলো শুধু $2x + y = 0$ এর মানে এর অসীম সংখ্যক সমাধান আছে। আবার যদি আমাদের সমীকরণ হয় $2x = 1$ ও $x = 2$ তাহলে আবার আমাদের কোনও সমাধান নেই।

প্রথমত n টি সমীকরণকে এভাবে কল্পনা কর যেন তাদের সব ভ্যারিয়েবল বামদিকে আর ত্রুটক মান ডানদিকে থাকে। এখন একটি $n \times n$ ম্যাট্রিক্স A নাও আর আরেকটি n সাইজের অ্যারে B নাও। যেমন $x + 2y = 5$ আর $3x + 2y = 7$ এর ক্ষেত্রে A আর B হবে এরকম:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix}, B = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

আর একে সমীকরণ আকারে লিখলে দাঁড়ায়:

$$AX = B \mapsto \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

এখন একটি counter ভ্যারিয়েবল নিতে হবে ধরা যাক এর নাম e যার শুরুর মান হলো 0। এবার আমাদের একটা কাজ n সংখ্যকবার করতে হবে। i তম বারে ($i = 0 \dots (n-1)$) আমরা e তম সারি (row) তে যাব। i এর মানে হলো আমরা এখন কোন ভ্যারিয়েবল বিবেচনা করছি। এবার আমাদের $A[j][i]$ গুলো দেখতে হবে যেখানে $e \leq j < n$ এবং এদের মধ্যে সেই j আমাদের নির্বাচন করতে হবে যেন $A[j][i]$ এর মান সর্বোচ্চ হয় (ঠিক সর্বোচ্চ না আমাদের বের করতে হবে কার পরমমান বা absolute value সবচেয়ে বেশি হয়)। যদি সব $A[j][i]$ এর মান 0 হয় তাহলে কিছু না করে আমাদের i এর লুপ চালিয়ে যেতে হবে। ধরলাম সর্বোচ্চটি 0 নয়। এখন B এবং A উভয়ের e তম সারি এবং j তম সারিকে অদলবদল করতে হবে। এর পর উভয় ম্যাট্রিক্স (A ও B) এর e তম সারিকে $A[e][i]$ দিয়ে ভাগ করতে হবে। তাহলে দেখবে অন্যান্য মান পরিবর্তনের সঙ্গে সঙ্গে $A[e][i]$ পরিবর্তন হয়ে 1 হয়ে যাবে। এবার যা করতে হবে তা হলো $0 \leq j < n$ এবং $j \neq i$ এ প্রতিটি সারির জন্য $A[e]$ সারিকে $A[j][i]$ দিয়ে গুণ করে $A[j]$ হতে বিয়োগ করতে হবে। শুধু A এর সারি না এই কাজ কিন্তু B এর সঙ্গেও করতে হবে। আমরা কিন্তু আসলে এই প্রসেস দিয়ে একটি ভ্যারিয়েবল কে এলিমিনেট (eliminate) করছি। এভাবে সব সারি হতে i তম ভ্যারিয়েবল দূর করা হয়ে গেলে আমরা e এর মান এক বাড়িয়ে দেব এবং আমাদের i এর লুপকে n নাহয় তাহলে বুঝতে হবে এর একমাত্র সমাধান নেই। খেয়াল করতে হবে A এর কোন কোন সারি সম্পূর্ণ 0 সেসব সারির B দেখতে হবে। যদি সেসব সারির কোনো একটির B যদি 0 নাহয় B এর সেই সারিও 0 হয় তার মানে বুঝতে হবে অসংখ্য সমাধান আছে। আর যদি A এর শূন্য সারির জন্য এর মানে আমাদের একটিমাত্র সমাধান আছে আর সেই সমাধানের ভ্যারিয়েবলগুলোর মান B তে কমপ্লেক্সিটি হলো $O(n^3)$. তোমাদের সুবিধার জন্য আরও একটি উদাহরণ করে দেখানো হচ্ছে টেবিল ৩.৩ এ। আর সেই সাথে প্রতিটি ধাপের বর্ণনাও দেওয়া হলো।

টেবিল ৩.৩: গাউসের এলিমিনেশনের উদাহরণ

Operation	Equations	Matrix		
গুরু	$2x - y + 3z = 15$ $4x - 2y + 2z = 42$ $x + y - z = 9$	$\begin{bmatrix} 2 & -1 & 3 \\ 4 & -2 & 2 \\ 1 & 1 & -1 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} 15 \\ 42 \\ 9 \end{bmatrix}$
$\frac{E_1}{2}$	$x - \frac{1}{2}y + \frac{3}{2}z = \frac{15}{2}$ $4x - 2y + 2z = 42$ $x + y - z = 9$	$\begin{bmatrix} 1 & -\frac{1}{2} & \frac{3}{2} \\ 4 & -2 & 2 \\ 1 & 1 & -1 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} \frac{15}{2} \\ 42 \\ 9 \end{bmatrix}$
$E_2 - 4E_1$	$x - \frac{1}{2}y + \frac{3}{2}z = \frac{15}{2}$ $-4z = 12$ $x + y - z = 9$	$\begin{bmatrix} 1 & -\frac{1}{2} & \frac{3}{2} \\ 0 & 0 & -4 \\ 1 & 1 & -1 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} \frac{15}{2} \\ 12 \\ 9 \end{bmatrix}$
$E_3 - E_1$	$x - \frac{1}{2}y + \frac{3}{2}z = \frac{15}{2}$ $-4z = 12$ $+\frac{3}{2}y - \frac{5}{2}z = \frac{3}{2}$	$\begin{bmatrix} 1 & -\frac{1}{2} & \frac{3}{2} \\ 0 & 0 & -4 \\ 0 & \frac{3}{2} & -\frac{5}{2} \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} \frac{15}{2} \\ 12 \\ \frac{3}{2} \end{bmatrix}$
$E_2 \leftrightarrow E_3$	$x - \frac{1}{2}y + \frac{3}{2}z = \frac{15}{2}$ $+\frac{3}{2}y - \frac{5}{2}z = \frac{3}{2}$ $-4z = 12$	$\begin{bmatrix} 1 & -\frac{1}{2} & \frac{3}{2} \\ 0 & \frac{3}{2} & -\frac{5}{2} \\ 0 & 0 & -4 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} \frac{15}{2} \\ \frac{3}{2} \\ 12 \end{bmatrix}$
$\frac{2E_2}{3}$	$x - \frac{1}{2}y + \frac{3}{2}z = \frac{15}{2}$ $y - \frac{5}{3}z = 1$ $-4z = 12$	$\begin{bmatrix} 1 & -\frac{1}{2} & \frac{3}{2} \\ 0 & 1 & -\frac{5}{3} \\ 0 & 0 & -4 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} \frac{15}{2} \\ 1 \\ 12 \end{bmatrix}$
$E_1 + \frac{E_2}{2}$	$x + \frac{2}{3}z = 8$ $y - \frac{5}{3}z = 1$ $-4z = 12$	$\begin{bmatrix} 1 & 0 & \frac{2}{3} \\ 0 & 1 & -\frac{5}{3} \\ 0 & 0 & -4 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} 8 \\ 1 \\ 12 \end{bmatrix}$
$\frac{E_3}{-4}$	$x + \frac{2}{3}z = 8$ $y - \frac{5}{3}z = 1$ $z = -3$	$\begin{bmatrix} 1 & 0 & \frac{2}{3} \\ 0 & 1 & -\frac{5}{3} \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} 8 \\ 1 \\ -3 \end{bmatrix}$
$E_1 - \frac{2E_3}{3}$	$x = 10$ $y - \frac{5}{3}z = 1$ $z = -3$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\frac{5}{3} \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$= \begin{bmatrix} 10 \\ 1 \\ -3 \end{bmatrix}$

পরের পাতায় চলমান

টেবিল ৩.৩ -- পূর্বের পাতা থেকে

Operation	Equations	Matrix
$E_2 + \frac{5E_3}{3}$	$x = 10$ $y = -4$ $z = -3$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ -4 \\ -3 \end{bmatrix}$

ধাপ ১: মাঝে সমীকরণ আর ডানে তার ম্যাট্রিক্স বহিঃপ্রকাশ লিখা হয়েছে।

ধাপ ২: প্রথম সমীকরণকে আমরা ২ দিয়ে ভাগ করেছি x এর সহগ 1 বানানোর জন্য। যদি আমাদের প্রথম ধাপ হওয়া উচিত ছিল x এর সবচেয়ে বড় সহগ বের করা কিন্তু আসলে যেকোনো অশূন্য সহগ নিলেও যে হয় সেটা দেখানোর জন্য আমি প্রথম সমীকরণ অন্য কারো সঙ্গে অদলবদল না করেই 2 দিয়ে ভাগ করছি। তাহলে কেন সবচেয়ে বড় সহগ কে ধরে আনতে বলা হয়? আমার জানা মতে এটা করলে numerical stability বজায় থাকে বা precision error কম হয়।

ধাপ ৩: প্রথম সমীকরণ কে 4 দিয়ে গুণ করে দ্বিতীয় সমীকরণ থেকে বিয়োগ করছি। কেন? কারণ এতে করে দ্বিতীয় সমীকরণ হতে x উধাও হয়ে যাবে। একেই আমরা বলি প্রথম সমীকরণ দিয়ে দ্বিতীয় সমীকরণ হতে x কে এলিমিনেট (eliminate) করা।

ধাপ ৪: তৃতীয় সমীকরণ হতে প্রথম সমীকরণ বিয়োগ করে x ভ্যারিয়েবলকে দূর করছি। তাহলে প্রথম সমীকরণ বাদে বাকি সবার থেকে x উধাও হয়ে গেছে।

ধাপ ৫: এবার দেখো দ্বিতীয় সমীকরণ এ y এর সহগ 0। আমাদের কে নিচে y আছে এরকম কোনো সমীকরণ এর সঙ্গে এই সারিকে অদলবদল করতে হবে। যদি এরকম কোনো সারি না থাকত তাহলে আমরা বলতাম এই সমীকরণ এর কোনো একমাত্র সমাধান নাই। যাই হোক, আমরা এই ধাপে তৃতীয় সমীকরণ এর সঙ্গে দ্বিতীয় সমীকরণ কে অদলবদল করি।

ধাপ ৬: বর্তমান দ্বিতীয় সমীকরণ কে আমরা $\frac{2}{3}$ দিয়ে গুণ করে y এর সহগ কে 1 করে ফেলি।

ধাপ ৭: দ্বিতীয় সমীকরণ কে 2 দিয়ে ভাগ করে প্রথম সমীকরণের সঙ্গে যোগ করি। তাহলে প্রথম সমীকরণ হতে y উধাও হয়ে যাবে। তৃতীয় সমীকরণে যেহেতু কোনো y নাই তাই ভ্যারিয়েবল দূর করারও কিছু নেই।

ধাপ ৮: এবার শেষ সারিকে -4 দিয়ে ভাগ করে z এর সহগকে 1 করে ফেলি।

ধাপ ৯ ও ১০: যথাক্রমে প্রথম ও দ্বিতীয় সমীকরণ হতে z উধাও করা হয়েছে।

গাউসের এলিমিনেশন (Gaussian elimination) এর কোড আসলে কঠিন কিছু না।
 কোড ৩.১১ এ দেখানো হলো। এখানে আমরা ধরে নিয়েছি যে সমীকরণজোটের একটিমাত্র সমাধান
 আছে। যদি একটিমাত্র সমাধান না থাকে তাহলে কী করতে হবে তা নিজেরা পরিবর্তন করে নিতে
 পারবে আশা করি। আর আমি এখানে অদলবদল করার কাজে swap ফাংশন ব্যবহার করেছি। এটা
 তোমরা stl এর algorithm হেডার ফাইলে পাবে।

কোড ৩.১১: gauss.cpp

```

1  for (i = 0; i < n; i++)
2  {
3      id = i;
4      for (j = i + 1; j < n; j++)
5          if (fabs(mat[j][i]) > fabs(mat[id][i]))
6              id = j;
7
8      if (id != i)
9      {
10         for (j = i; j <= n; j++)
11             swap(mat[i][j], mat[id][j]);
12     }
13
14
15     for (j = 0; j < n; j++)
16         if(j != i)
17         {
18             factor = mat[j][i] / mat[i][i];
19
20             for(k = i; k <=n; k++)
21                 mat[j][k] -= factor * mat[i][k];
22         }
23 }
```

৩.৫ প্রোগ্রামিং সমস্যা

৩.৫.১ অনুশীলনী

খুব সহজ

- UvaLive 6823 □ UvaLive 6949 □ UvaLive 6977 □ UvaLive 6982 □ UvaLive 6988 □ UvaLive 7035

সহজ

- UvaLive 6843* □ UvaLive 6844 □ UvaLive 6847 □ UvaLive 6858
- UvaLive 6862 □ UvaLive 6909 □ UvaLive 6918 □ UvaLive 6921 □ UvaLive 6929 □ UvaLive 6962 □ UvaLive 6966 □ UvaLive 6969 □ UvaLive 6984
- UvaLive 6994 □ UvaLive 7024 □ UvaLive 7031 □ UvaLive 7032 □ UvaLive 7034 □ UvaLive 7040 □ UvaLive 7045 □ UvaLive 7083 □ UvaLive 7147
- UvaLive 7163* □ UvaLive 7169

সামান্য কঠিন

- UvaLive 6825* □ UvaLive 6831** □ UvaLive 6916*** □ UvaLive 6919
- UvaLive 6964 □ UvaLive 6974 □ UvaLive 6985* □ UvaLive 6987* □ UvaLive 7048** □ UvaLive 7060** □ UvaLive 7062

কঠিন

- UvaLive 7051***

অধ্যায় ৪

সর্টিং (Sorting) ও সার্চিং (Searching)

৪.১ সর্টিং (Sorting)

সর্ট বা sort করার অর্থ হলো একটি নির্দিষ্ট ক্রমে সাজানো। আমরা ধর পরীক্ষার খাতা ১ রোল হতে ৬০ রোল পর্যন্ত যদি সাজাই তাহলে এটাকে সর্টিং বলে। প্রায়ই আমাদের বিভিন্ন সংখ্যা সর্ট করার প্রয়োজন হয়। শুধু সংখ্যা নয়, স্ট্রিং, স্থানাঙ্ক এরকম নানা কিছু সর্ট করতে হতে পারে। আমরা এই সেকশনে কিছু সর্টিং অ্যালগরিদম দেখব।

৪.১.১ ইনসার্চন সর্ট (Insertion Sort)

সাধারণত আমরা বাস্তবে এভাবে সর্টিং করে থাকি। মনে কর আমাদের কাছে ৬০ জনের খাতা আছে। আমরা একটি করে খাতা নেই, আর ক্রমানুসারে সাজানো বা সর্টেড খাতাগুলোর মধ্যে এই খাতাকে সঠিক জায়গায় রাখি। আবার নতুন খাতা নিব আর ক্রমানুসারে সাজানো খাতাগুলোর মধ্যে এই নতুন খাতাকে ঠিক জায়গায় রাখব। এভাবে সবগুলো খাতাকে রাখা শেষ হলেই আমাদের সর্টিংও শেষ হয়ে যাবে। এখন যদি ইমপ্লিমেন্টেশনের কথা চিন্তা কর তাহলে মনে হবে এভাবে একটি একটি করে খাতা ঢুকানো মনে হয় কঠিন কাজ। কিন্তু ওত কঠিন না। মনে কর তোমার $1 \dots i - 1$ খাতাগুলো সাজানো আছে, তুমি i তম খাতা ঢুকাবে, তুম প্রথমে দেখ $i - 1$ এর খাতাটি কি তোমার থেকে ছোট? তাহলে যেখানে আছে সেখানেই তোমার খাতার অবস্থান আর যদি না হয় তাহলে $i - 1$ এ থাকা খাতাকে i এ আনো আর এবার $i - 2$ এর সঙ্গে মিলিয়ে দেখো। এভাবে একে একে তুলনা করতে থাক। একটি উদাহরণ টেবিল ৪.১ এ দেওয়া হলো।

এর কোডটিও কিন্তু বেশ ছোট। কিন্তু কোড করতে সোজা হলেও এই অ্যালগরিদমের টাইম কমপ্লেক্সিটি $O(n^2)$. আমরা পরে দেখব এর থেকেও অনেক দ্রুত সর্টিং করা সম্ভব। ইনসার্চন সর্টের প্রোগ্রাম কোড ৪.১ এ দেওয়া হলো:

টেবিল ৪.১: ইনসার্শন সর্টের সিমুলেশন

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
1, 5, 6, 7, 8, 9

কোড ৪.১: insertion sort.cpp

```

১ for(i = 1; i <= n; i++)
২ {
৩     x = num[i];
৪     j = i - 1;
৫     while (j >= 1 && num[j] > x)
৬     {

```

```

7     num[j + 1] = num[j];
8     j--;
9 }
10    num[j + 1] = x;
11 }
12 }
```

8.1.2 সিলেকশন সর্ট (Selection Sort)

$O(n^2)$ এর সকল সর্ট অ্যালগরিদমের মধ্যে আমার কাছে সবচেয়ে সহজ লাগে সিলেকশন সর্ট (Selection Sort). এর মূল ধারণা হলো তুমি শুরুতে প্রথম অবস্থানের সংখ্যাটি নির্বাচন করবে। এর পর এই সংখ্যার পরের সংখ্যাগুলো একে একে দেখবে। যদি দেখো পরের কোনো সংখ্যা তোমার নির্বাচিত অবস্থানের সংখ্যার থেকে ছোট তাহলে দুটি সংখ্যাকে অদলবদল করে দিবে। এভাবে সব সংখ্যা দেখা শেষ হয়ে গেলে তোমার নির্বাচিত অবস্থানে সবচেয়ে ছোট সংখ্যা থাকবে। এবার দ্বিতীয় অবস্থানের সংখ্যাটি নির্বাচন কর। বাকি সব সংখ্যা দিয়ে যাও আর যদি দেখো যেই সংখ্যা দিয়ে যাচ্ছ সেটি ছোট তাহলে অদলবদল। এভাবে প্রতিটি সংখ্যা এক এক করে নির্বাচন করলে দেখবে পুরো অ্যারেটি সর্ট হয়ে গেছে। একটি উদাহরণ টেবিল 8.2 এ দেখানো হলো। এর কোড 8.2 এ দেখানো হলো।

কোড 8.2: selection sort.cpp

```

1 for (int i = 1; i <= n; i++) {
2     for (int j = i + 1; j <= n; j++) {
3         if (num[i] > num[j]) {
4             // swap is inside algorithm header.
5             swap(num[i], num[j]);
6         }
7     }
8 }
```

8.1.3 বাবল সর্ট (Bubble Sort)

$O(n^2)$ এ যেসব সর্টিং অ্যালগরিদম আছে তাদের মধ্যে সিলেকশন সর্ট আমার সবচেয়ে সহজ মনে হলেও কোনো কারণে অন্যরা বাবল সর্ট (bubble sort) কে সহজ মনে করে থাকে। এই অ্যালগরিদমটিও বেশ মজার। তুমি প্রথম থেকে শেষ পর্যন্ত পাশাপাশি দুটি দুটি করে সংখ্যা নিতে

টেবিল ৪.২: সিলেকশন স্টের সিমুলেশন

⑤, ৮, ৬, ১, ৭
⑤, ৮, ৬, ১, ৭
⑤, ৮, ৬, ১, ৭
①, ৮, ৬, ৫, ৭
1, ⑧, ৬, ৫, ৭
1, ⑥, ৮, ৫, ৭
1, ⑤, ৮, ৬, ৭
1, ৫, ⑧, ৬, ৭
1, ৫, ⑥, ৮, ৭
1, ৫, ৬, ⑧, ৭
1, ৫, ৬, ৭, ৮
1, ৫, ৬, ৭, ৮

থাক। যদি দেখ আগেরটি পরেরটি থেকে বড় তাহলে অদলবদল কর। এই কাজ n সংখ্যকবার কর। এই অ্যালগরিদমটি কিন্তু $O(n^2)$ সময় লাগবে। কারণ প্রথম বার যখন তুমি বাম থেকে ডানে যাবে তখন সবচেয়ে বড়টি অবশ্যই একদম ডান মাথায় গিয়ে পৌছাবে। পরের বারে দ্বিতীয় বড়টি স্থিক জায়গায় যাবে, এরকম প্রতিবারে একটি একটি করে সংখ্যা স্থিক স্থানে যাবে। তাই n বার এই কাজ করলে সব সংখ্যা স্থিক জায়গায় চলে যাবে। এই প্রোগ্রামের কোড ৪.৩ এ দেওয়া হলো। তোমরা চাইলে এই কোডে কিছু অপটিমাইজেশন করতে পার। যেমন, এমন হতে পারে যে n বারে আগেই অ্যারেটি স্টেড হয়ে গেছে সেই ক্ষেত্রে তুমি আগেই লুপ থেকে বের হয়ে যেতে পার। আবার

প্রতিবার তোমার সব জোড়া সংখ্যা যাচাই করার দরকার নেই কিন্তু, কারণ; সংখ্যকবার চললে তুমি জান যে শেষ টি সংখ্যা ঠিক জায়গায় চলে গিয়েছে। তুমি এভাবে কিছু অপটিমাইজেশন করতে পার। কিন্তু যতই অপটিমাইজেশন কর না কেন এই অ্যালগরিদমের worst কেস এ $O(n^2)$ সময়ই লাগবে। তোমরা কি সেই worst কেইসটি বের করতে পারবে? :

কোড 8.3: bubble sort.cpp

```

1 for(i = 1; i <= n; i++)
2 {
3     for(j = 1; j < n; j++)
4         if(num[j + 1] > num[j])
5         {
6             temp = num[j];
7             num[j] = num[j + 1];
8             num[j + 1] = temp;
9         }
10 }
```

8.1.8 মার্জ সর্ট (Merge Sort)

এই অ্যালগরিদমটি বেশ মজার। এর সঙ্গেও আমাদের বাস্তব জীবনের কিছু মিল আছে। আবার আমরা সেই খাতা সর্টিংয়ে ফিরে যাই। মনে কর তোমার কাছে 60 টি খাতা আছে। তুমি এই খাতাকে দুভাগ করে দুজনকে দিয়ে দিলে। তারা তাদের দুভাগ সর্ট করে তোমাকে দিল। এখন তুমি ওই দুটি সর্টেড খাতার ভেতরে না দেখে শুধু উপরে দেখবে, যারটি ছোট তুমি সেই ভাগ থেকে খাতা নিবে এভাবে তুমি ছোট থেকে বড় সাজিয়ে ফেলবে। খুবই সোজা তাই না? এখন ঘটনাটায় আরেকটু প্যাঁচ লাগবে, তাই আগানোর আগে আমি যা বললাম তা ভালো করে কল্পনা করে নাও। ভালও মতো কল্পনা করে নিজেকে প্রশ্ন কর- তুমি যেই দুজনকে দিলে তারা কীভাবে সর্ট করবে? উত্তরটি হলো আবারও দুভাগ করে। অর্থাৎ তুমি যেভাবে দুজনকে ভাগ করে দিয়েছ, তারাও আবার দুভাগ করে দুজনকে দিবে এবং তাদের কাছ থেকে পাওয়ার পর তারা ওটাকে সাজিয়ে তোমাকে দিবে। এভাবেই চলতে থাকবে। বুঝতেই পারছ এই কাজ করতে হবে রিকার্সিভ ফাংশনের মাধ্যমে। এই ফাংশনকে তুমি যদি একটি অ্যারে দাও সে একে দুভাগ করবে এবং আবার এই ফাংশনকেই কল করে তাদের মাধ্যমে ওই দুভাগ সর্ট করে আনবে। এরপর এদেরকে মার্জ করে বা একত্র করে পুরো সর্টেড ফলাফলকে সে রিটার্ন করবে। এই মার্জ সর্ট এর প্রোগ্রামটি কোড 8.4 এ দেওয়া হলো। খেয়াল কর যে, mergesort ফাংশনটি দুটি ভ্যারিয়েবলকে প্যারামিটার হিসাবে নেয়, lo এবং hi

¹উত্তরটি হলো যদি অ্যারেটি শুরুতেই বড় থেকে ছোটতে সাজানো থাকে তাহলে তোমার $O(n^2)$ সময় লাগবেই।

(হাঁ, আমি জানি এই বানান ভুল, কিন্তু আমি low এবং high কে সংক্ষেপে এভাবে লিখি)। এই দুই ভ্যারিয়েবল মূল অ্যারের দুই মাথা নির্দেশ করে। এই ফাংশনের কাজ হলো এই দুই মাথার মাঝের অংশটুকু সর্ট করা (দুই মাথাসহ)।

কোড 8.8: merge sort.cpp

```

1 int num[100000], temp[100000];
2
3 //call mergesort(a, b) if you want to sort num[a...b]
4 void mergesort(int lo, int hi)
5 {
6     if(lo == hi) return;
7     int mid = (lo + hi)/2;
8
9     mergesort(lo, mid);
10    mergesort(mid + 1, hi);
11
12    int i, j, k;
13    for(i = lo, j = mid + 1, k = lo; k <= hi; k++)
14    {
15        if(i == mid + 1) temp[k] = num[j++];
16        else if(j == hi + 1) temp[k] = num[i++];
17        else if(num[i] < num[j]) temp[k] = num[i++];
18        else temp[k] = num[j++];
19    }
20
21    for(k = lo; k <= hi; k++) num[k] = temp[k];
22 }
```

এখানে mergesort ফাংশন সর্ট করার জন্য তার অংশকে দুভাগে ভাগ করে এবং প্রতি ভাগের জন্য আবার mergesort ফাংশন কল করে। যখন সে কল করা ফাংশন থেকে ফেরত আসে তখন তার কাজ হলো ওই দুই অংশ মার্জ করা বা একত্র করা। সে ওই দুই অংশের শুরু থেকে দেখা শুরু করে। যেটি ছোট সেটাকে temp নামের অ্যারেতে নিয়ে রাখে, এভাবে একে একে সব সংখ্যাকে temp নামের অ্যারেতে সাজিয়ে রাখে। সবশেষে temp অ্যারের সব সংখ্যা মূল অ্যারেতে কপি করে দেয়। ফলে মূল অ্যারের ওই অংশটুকু সর্টেড হয়ে যায়। আর base কেইসটি কী হবে আশা করি বুঝতে পারছ। এখন কোড 8.8 এ দেখতে পাচ্ছ যে লুপের ভেতরের if-else টা শুধু কে ছোট সেটাই দেখছে না, আরও কিছু দেখছে। যা দেখছে তা হলো, যদি বামের অ্যারে থেকে সংখ্যা নেয়া

শেষ হয়ে যায় তাহলে কোন ছোট-বড় তুলনা করা ছাড়াই ডানের অ্যারে থেকে সংখ্যা নিতে হবে। একইভাবে যদি ডানের অ্যারে যদি শেষ হয়ে যায় তাহলে তোমাকে বামের অ্যারে হতে নিতে হবে। যদি দুটি অ্যারেতেই সংখ্যা থাকে তাহলেই কেবল তাদের তুলনা করে ছোটটি নিতে হবে।

এখন প্রশ্ন হলো এই অ্যালগরিদমের টাইম কমপ্লেক্সিটি কত? ধরা যাক, আমাদের n সাইজ এর একটি অ্যারেকে সর্ট করতে বলা হয়েছে আর এর জন্য আমাদের সময় লাগে $T(n)$. আমাদের এই ফাংশন প্রথমেই n টি জিনিসকে সমান দুভাগে ভাগ করে এবং তাদের উপর এই অ্যালগরিদম চালায়। সুতরাং এই দুভাগের জন্য আমাদের সময় লাগবে $2T(\frac{n}{2})$. এখন আসা যাক আমাদের মার্জ (merge) অংশে। আমাদের এক ভাগে আছে $n/2$ টি সংখ্যা অন্যভাগেও আছে $n/2$ টি সংখ্যা। আমরা প্রতিবার এই দুভাগের শুধু মাথার সংখ্যা চেক করে দেখি আর যেটি কম তাকে temp অ্যারেতে নেই। এভাবে চলতে থাকে। এই কাজটি কিন্তু n বার হবে কারণ প্রতিবার আমরা একটি করে সংখ্যা সরাচ্ছি। যদি n বার সরাই তাহলে সব সংখ্যা সরে যাবে। এই n বার শুধু আমরা দেখি যে কোনটি ছোট। সুতরাং এই পুরো কাজটার জন্য আমাদের লাগে $O(n)$ সময়। অর্থাৎ,

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

অর্থাৎ এই পদ্ধতিতে আমাদের টাইম কমপ্লেক্সিটি হলো $O(n \log n)$.

8.1.5 কাউন্টিং সর্ট (Counting Sort)

তোমরা হয়তো অনেকেই শুনে থাকবে যে, সর্টিংয়ের জন্য $O(n \log n)$ এর থেকে ভালো অ্যালগরিদম সন্তুষ্ট না এবং এই কথা সত্যি। কিন্তু তোমরা যেসব সংখ্যা সর্ট করবে সেসব যদি অঞ্চলাত্মক সংখ্যা হয় এবং এদের সবচেয়ে বড় সংখ্যা যদি N হয় তাহলে কাউন্টিং সর্টের টাইম কমপ্লেক্সিটি হবে $O(N+n)$. এই অ্যালগরিদমটি বেশ সহজ। তোমাকে একটি সাহায্যকারী অ্যারে নিতে হবে। এরপর যেসব সংখ্যাকে সর্ট করতে হবে তাদের উপস্থিতি গণনা করে এই সাহায্যকারী অ্যারেতে এক বাড়াতে হবে। এরপর তুমি ওই সাহায্যকারী অ্যারের 1 হতে N পর্যন্ত একটি লুপ চালাবে আর দেখবে; তম সংখ্যা কতগুলো আছে। ততগুলো; তুমি আউটপুট অ্যারেতে রাখবে। তাহলে এই আউটপুট অ্যারেতে সব সংখ্যা সর্টেড আকারে পাওয়া যাবে।

৪.১.৬ STL এর sort ফাংশন

আমাদের জীবনকে সহজ করার জন্য লাইব্রেরি ফাংশন হিসেবে sort নামে একটি ফাংশন দিয়ে দেওয়া হয়েছে। এই ফাংশন ব্যবহার করতে হলে আমাদের STL এর algorithm নামক হেডুন ফাইলকে ইনক্লুড করতে হবে। এখন আমরা যদি num নামক অ্যারের ০ থেকে $n - 1$ পর্যন্ত সংজ্ঞা করতে চাই, তাহলে আমাদের লিখতে হবে: sort(num, num + n). আমরা যদি ১ থেকে n পর্যন্ত সর্ট করতে চাই তাহলে আমাদের লিখতে হবে: sort(num + 1, num + n + 1). অর্থাৎ আমরা যদি a থেকে b পর্যন্ত সর্ট করতে চাই তাহলে আমাদের লিখতে হবে: sort(num + a, num + b + 1). এসব ক্ষেত্রে কিন্তু সবসময় ছোট থেকে বড় তে সর্ট হবে। যদি আমরা কোনো একটি ভেষ্টের V কে সর্ট করতে চাই তাহলে আমাদের লিখতে হবে, sort(V.begin(), V.end()).

এবার একটু কঠিন কাজ করা যাক। মনে কর আমাদের দ্বিমাত্রিক (2D) এ কিছু বিন্দু দেওয়া হলো। এসব বিন্দুর x ও y স্থানাঙ্ক আমাদের দেওয়া আছে। আমাদের এমনভাবে সাজাতে হবে যেন যাদের x ছোট তারা আগে থাকে, যদি কোনো দুটি বিন্দুর x সমান হয় তাহলে যেন যার y ছোট সে আগে থাকে। আমরা এই বিন্দুর x ও y সংরক্ষণের জন্য একটি স্ট্রাকচার (structure) ব্যবহার করব। অর্থাৎ আমাদের কাছে কোনো একটি স্ট্রাকচারের অ্যারে আছে, আমাদেরকে এই জিনিস সর্ট করতে হবে। মনে কর আমাদের স্ট্রাকচারটির নাম Point এবং অ্যারেটির নাম point. এখন তুমি যদি শুধু sort(point, point + n) লিখ তাহলে কিন্তু হবে না। কারণ দুটি Point এর মধ্যে কোনটি ছোট তা কিন্তু কম্পিউটার এমনি এমনি বুঝবে না। তাকে বলে দিতে হবে কী কী শর্ত মানলে আমরা বলতে পারি যে একটি Point আরেকটি Point এর থেকে ছোট। এ জন্য আমাদের একটি ফাংশন লিখতে হবে, ধরা যাক এই ফাংশনের নাম cmp. এই ফাংশনের কাজ হলো তাকে দুটি Point দেওয়া হবে, যদি প্রথম Point দ্বিতীয় Point এর থেকে ছোট হয় তাহলে এই ফাংশনকে true রিটার্ন করতে হবে আর যদি বড় বা সমান হয় তাহলে false রিটার্ন করতে হবে। এখানে খুব ভাল করে খেয়াল রাখতে হবে যে, কোনো ক্রমেই যেন, cmp ফাংশন cmp(A, B) ও cmp(B, A) উভয় ক্ষেত্রেই true না দেয়। যদি এধরনের ভুল করে থাক তাহলে সাধারণত তুমি Run Time Error পেয়ে থাকবে। আমরা কোড ৪.৫ এর মতো করে এই ফাংশনটি লিখতে পারি।

কোড ৪.৫: cmp.cpp

```
১ bool cmp(Point A, Point B)
২ {
৩     if(A.x < B.x) return 1;
৪     if(A.x > B.x) return 0;
৫
৬     if(A.y < B.y) return 1;
৭     if(A.y > B.y) return 0;
৮
৯     return 0;
```

এই ফাংশনকে চাইলে আমরা আরও সংক্ষেপে কোড ৪.৬ এর মতো করেও লিখতে পারি।

কোড ৪.৬: 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 সর্ট করার জন্য লিখতে হবে: sort(point, point + n, cmp). অপারেটর ওভারলোড (operator overload) করেও এই কাজ করা যায়। তবে সমস্যা হলো, একবারই অপারেটর ওভারলোড করা যায়। সুতরাং তুমি যদি একই ধরনের জিনিসকে যদি বিভিন্নভাবে সর্ট করতে চাও তা সম্ভব হবে না। কোড ৪.৭ এ অপারেটর ওভারলোড কীভাবে করতে হয় তা দেখানো হলো। এক্ষেত্রে তুমি sort(point, point + n) লিখলেই হবে।

কোড ৪.৭: 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 দেখতো, একটি পূর্ণ সংখ্যার অ্যারেকে যদি বড় থেকে ছোট আকারে সাজাতে চাও তাহলে কী করবে? ১

অনেকে স্ট্রিং সর্ট করা নিয়ে বেশ ঝামেলায় পড়ে, কিন্তু STL এর string আমাদের জীবনকে অনেক সহজ করে দিয়েছে। কোড ৪.৮ এ আমরা কিছু স্ট্রিং ইনপুট নিয়ে তা সর্ট করা দেখালাম।

^১একটি ফাংশন লিখেও করতে পার। অথবা চাইলে ফাংশনাল হেডার ফাইল (functional header file) ব্যবহার করেও করতে পার: sort(num, num + n, greater<int>()).

```

1 #include<stdio.h>
2 #include<string>
3 #include<algorithm>
4 #include<vector>
5 using namespace std;
6
7
8 int main()
9 {
10     int n, i;
11     char s[100];
12     vector<string> V;
13
14     scanf("%d", &n);
15     for(i = 0; i < n; i++)
16     {
17         scanf("%s", s);
18         V.push_back(s);
19     }
20
21     sort(V.begin(), V.end());
22
23     return 0;
}

```

শেষ করার আগে একটা জিনিস। অনেকে এখনও stdlib হেতারের qsort ব্যবহার করে থাকে। qsort অর্থাৎ quick sort হলো average case $O(n \log n)$ কিন্তু worst কেইস যেতে হয় কারণ anti-qsort বলে একটি অ্যালগরিদম আছে যাকে n দিলে qsort এর জন্ম করলে anti-qsort ব্যবহার করে challenge করে ফেল। সুতরাং তোমরা qsort ব্যবহার কর না, অন্য কেইস কোড পেয়ে যাবে।

৪.২ বাইনারি সার্চ (Binary Search)

ধর তুমি একটি গেইম শো তে আছ। তোমার সামনে মোট 100 টি 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 ভেতরে 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। 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 কোথায় 1986 আছে। তুমি ঠিক মাঝের বাক্সটি খোল। যদি দেখ এটিই 1986 তাহলে তো হয়েই গেল। আর যদি দেখ এখানে 1986 এর থেকে বড় সংখ্যা আছে তার মানে তোমার সংখ্যা বামের অর্ধেকে আছে আর না হলে ডানের অর্ধেকে আছে। খেয়াল কর, তুমি এক ধাক্কায় 100 বাক্স থেকে 50 বাক্সকে কিন্তু বাদ দিয়ে ফেলতে পারছ। একইভাবে তুমি এই বাকি অর্ধেককেও 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 worst কেইসে তোমার কতগুলো বাক্স খুলতে হবে?^১ এভাবে খোঁজার পদ্ধতিকে আমরা বাইনারি সার্চ (binary search) বলে থাকি। অনেক সময় এটি বাইসেকশন পদ্ধতি (bisection method) নামেও পরিচিত।

আরও একটি উদাহরণ দেওয়া যাক, মনে কর একটি অ্যারেতে শুধু 0 ও 1 আছে। সব 0 সব 1 এর আগে থাকবে। তোমাকে প্রথম 1 খুঁজে বের করতে হবে। এটিও কিন্তু বাইনারি সার্চ দিয়ে করতে পার। তুমি মাঝখানে গিয়ে দেখবে এটি 0 নাকি 1, যদি 0 হয় তাহলে তো এর ডানে থাকবে তোমার কাঞ্চিত জায়গা, আর যদি 1 হয় তাহলে এটিসহ বামে থাকতে পারে। এভাবে তুমি খুঁজতে থাকবে।

তাহলে এখন প্রশ্ন হলো এটি কোড করবা কেমন করে। বিভিন্ন জন বিভিন্ন ভাবে বাইনারি সার্চের কোড করে। যদি তুমি সাবধান না হও বা না বুঝে অঙ্কের মত কোনো একটি কোড কে অনুসরণ কর তাহলে এটি হলফ করে বলাই যায় যে তুমি কমপক্ষে একবার ঝামেলায় পরবে। প্রথমত বাইনারি সার্চ করতে হয় একটি সীমার মধ্যে। মনে কর তোমার সীমার শুরু হলো lo আর শেষ হলো hi (আমি আমার সুবিধার জন্য পুরো $high$ বা low না লিখে সংক্ষেপে hi বা lo লিখি)। এখন সমস্যা ভেদে আমাদের কাজ বিভিন্ন হতে পারে। যেমন শুরু থেকে কিছু দূর হয়তো $f(x)$ সত্য, এর পর থেকে $f(x)$ মিথ্যা। এখন হয়তো তোমাকে x এর সবচেয়ে বড় মান বের করতে হবে যেখানে $f(x)$ সত্য। বা উল্টোটাও হতে পারে। হয়তো শুরুর কিছু অংশে $f(x)$ মিথ্যা। এর পর থেকে সত্য। তোমাকে x এর সবচেয়ে ছোট মান করতে হবে যেখানে $f(x)$ সত্য হয়। সব কিছু শুরু করার আগে তোমার নিশ্চিত হওয়া উচিত যে এই সীমার ভিতরে তোমার উত্তর আছে। যদি না থাকে তাহলে কি করবা তা করে ফেল। এই বিশেষ কেইসকে handle করার পর আমরা জানি আমাদের উত্তর lo হতে hi এর মধ্যে আছে। আমাদের লক্ষ্য হলো $lo = hi$ না হওয়া পর্যন্ত আমরা উত্তর খুঁজতে থাকব অর্থাৎ while ($lo < hi$)। আমাদের কাজ হবে মাঝের একটি সংখ্যা mid নেয়া এবং আমাদের সীমাকে কমিয়ে আনা। mid এর মান যাচাই করার পর বুঝতে পারলে যে তোমার উত্তর lo হতে mid এ

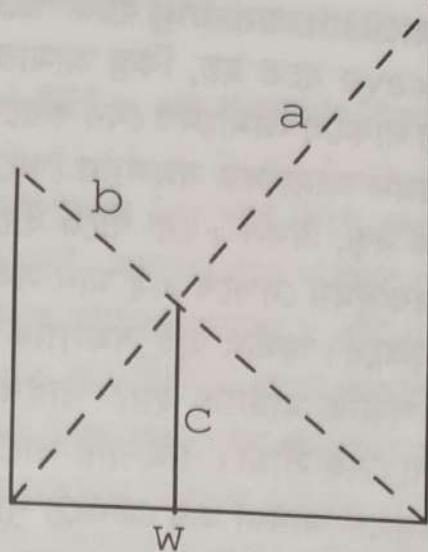
¹উত্তর কিন্তু মাত্র 7টি বাক্স :)

আছে। তাহলে তুমি hi কে কমিয়ে mid করবে। যদি বুঝ যে তোমার উত্তর lo হতে $mid - 1$ এ আছে, তাহলে একইভাবে hi কে কমিয়ে $mid - 1$ করবে। উল্টোও হতে পারে, হয়তো তুমি lo কে বাড়িয়ে mid বা $mid + 1$ করবে। কখন কি করছ তার উপর ভিত্তি করে তোমাকে ঠিক করতে হবে যে $mid = \frac{lo+hi}{2}$ নাকি $mid = \frac{lo+hi+1}{2}$ হবে। কেন? মনে কর তুমি $mid = \frac{lo+hi}{2}$ করছে। একইসঙ্গে তুমি লিখলে যে যদি $f(mid)$ সমাধান হয় তাহলে $lo = mid$ কর। এখন একটু চিন্তা করে দেখো যদি $hi = lo + 1$ হয় তাহলে কিন্তু এটি একটি অসীম লুপে পরে যাবে। কেন? কারণ এই ক্ষেত্রে $mid = \frac{lo+hi}{2} = \frac{2lo+1}{2} = lo$ আর তুমি এই ক্ষেত্রে $lo = mid = lo$ করছ, অর্থাৎ তোমার lo বা hi এর মান পরিবর্তন হচ্ছে না, আর যেহেতু while এর শর্ত হলো $lo < hi$ তাই এই লুপ সারা জীবন চলতে থাকবে। কিন্তু এখানে যদি তুমি $mid = \frac{lo+hi+1}{2}$ ব্যবহার করতে তাহলে কোনো সমস্যা হতো না। সুতরাং তোমাকে একটু ভেবে চিন্তে ঠিক করতে হবে যে mid এর মান কি হবে। এ জন্য আমি যেটা করি সেটা হলো $hi = lo + 1$ এর ক্ষেত্রে দেখি যে $f(mid)$ সত্য হোক আর মিথ্যা হোক তোমার lo বা hi এর মান পরিবর্তন হয় কিনা, যদি হয় তাহলে কোনো সমস্যা নেই। mid এর যেই মানের জন্য পরিবর্তন হবে সেই মান নিলেই হবে। 2008 সালের ওয়ার্ল্ড ফাইনালিস্ট সাল্বিল ইউসুফ সানি ভাইয়া যা করতেন তা আরও একটু নিরাপদ। তিনি যা করতেন তা হলো while ($hi - lo > 5$) এরকম একটি বড় পার্থক্যের সীমাকে দিয়ে বাইনারি সার্চ করতেন। এর পর একটি লুপ চালাতেন lo থেকে hi পর্যন্ত এবং বের করতেন কোনটি উনার কাঞ্চিত উত্তর। তুমি কিছুদিন কোজ করলে হয়তো তোমার নিজের একটি পদ্ধতি বের করে ফেলবে যেটিতে তুমি নিজে স্বাচ্ছন্দ্য বোধ করবে।

বাইনারি সার্চ ব্যবহার করে কিছু অন্তর্ভুক্ত সমস্যাও সমাধান করা যায়। অন্তর্ভুক্ত বললাম এই কারণে যে, প্রবলেম দেখে হয়তো কখনই মনে হবে না যে এখানে বাইনারি সার্চ ব্যবহার করা যায়, কিন্তু যায়! যেমন ৪.১ নং চিত্রে একটি w প্রস্ত্রের রাস্তার দুদিকে দুটি দালান আছে। এখন রাস্তার এক মাথায় একটি মই রেখে অপর মাথা রাস্তার অন্য পারের দালানের মাথায় রাখা হলো, একইভাবে রাস্তার অন্য পাশ থেকেও আরেকটি মই রাখা হলো। মই দুটির দৈর্ঘ্য a ও b . মই দুটি রাস্তা থেকে উচ্চতায় ছেদ করে। a, b এবং c এর মান দেওয়া আছে, $w = ?$.

তোমরা যদি এখানে বিভিন্ন সূত্র খাটাও দেখবে খুব একটি সহজে কোনো সূত্র পাবে না w নির্ণয়ের জন্য। কিন্তু একটু অন্যভাবে চিন্তা কর। তুমি যদি w এর মান জানো তাহলে কি তুমি c এর মান বের করতে পারবে? যেহেতু রাস্তার প্রস্ত্র দেওয়া আছে আর আমরা মইয়ের দৈর্ঘ্যও জানি সেহেতু আমরা প্রথমে দালান দুটির উচ্চতা বের করি (পিথাগোরাসের উপপাদ্য ব্যবহার করে)। ধরা যাক উচ্চতা দুটি হলো p ও q . এখন সদৃশকোনী ত্রিভুজের সূত্র 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 উল্টোটি কিন্তু কঠিন। ধরা যাক আমাদের দেওয়া c এর মানের জন্য উত্তরটি হবে w' . যদি তুমি w এর মান হিসেবে w' এর থেকে বড় মান নাও তাহলে, c এর মান প্রদত্ত মানের থেকে কম পাবে, আবার উল্টোভাবে যদি তুমি w এর মান হিসাবে w' এর থেকে ছোট মান নাও তাহলে c এর মান প্রদত্ত মানের থেকে বড় পাবে। কেবল w এর মান w' হলেই সঠিক দিবে।^১ সুতরাং তোমরা w এর মানের উপর বাইনারি সার্চ

^১কীভাবে বুঝলাম এই ছোট বড় কাহিনী? তেমন কিছু না একটু কল্পনা কর যে ধীরে ধীরে w বড় হচ্ছে এর মানে



নকশা 8.1: $W = ?$

চালাবে আর দেখবে C এর মান প্রদত্ত মানের থেকে বড় না ছোট সেই অনুসারে w এর মানের সীমাও পরিবর্তন করবে। এখন প্রশ্ন হতে পারে যে এরকম করে কতক্ষণ ভাগ করতে থাকবে? যদি উভয় পূর্ণ সংখ্যা হয় তাহলে তো বাইনারি সার্চ এক সময় না এক সময় শেষ হয়ে যাবে, কিন্তু যদি double ডেটাটাইপে উভয় হয় তাহলে তো এই লুপ চলতেই থাকবে। উপায় হলো যতক্ষণ না সীমার পরিধি খুব ছোট হয় ততক্ষণ এই কাজ করতে থাকবে। তবে এভাবে করা ঠিক না, কারণ আমরা জানি double ডেটাটাইপে হিসাব করলে বিভিন্নভাবে precision loss হয়। সেজন্য সবচেয়ে ভালো উপায় হলো এই কাজ 50 থেকে 100 বার করা। মানে তোমার টাইম লিমিটের ভেতর থেকে যতক্ষণ করা যায় আর কী, আবার তাই বলে অনেক বেশি বার করে লাভ নেই।

8.3 টারনারি সার্চ(Ternary Search)

মনে কর x অক্ষের উপর কিছু বিন্দুতে তোমার বন্ধুর বাসা আছে। তোমাকে এমন জায়গায় বাসা বানাতে হবে যেন তোমার বাসা থেকে তোমার সব বন্ধুর বাসায় যাওয়ার দূরত্বের যোগফল সর্বনিম্ন হয়। এই উভয় যদিও অন্যভাবে খুব সহজেই বের করা যায় কিন্তু আমরা শিখব টারনারি সার্চ (ternary search) ব্যবহার করে কীভাবে সমাধান করতে হয়। বাইনারি সার্চ এ সাধারণত আমাদের ফাংশনটি হয় ক্রমবর্ধমান (increasing) বা ক্রমহ্রাসমান (decreasing) হয়ে থাকে এবং সেই সঙ্গে আমাদের একটি লক্ষ্যমান y দেওয়া থাকে, আমাদেরকে এমন একটি x বের করতে হয় যেন তার জন্য ফাংশনের মান y হয়। কিন্তু টারনারি সার্চের ক্ষেত্রে ফাংশনটি একটু অন্যরকম হতে হয়। ফাংশনটিকে প্রথমে strictly increasing হতে হবে, এর পর কিছুক্ষণ সেঞ্চুরিক থাকতে পারবে, এর পর তাকে strictly decreasing হতে হবে। এটি আশা করি বলতে হবে না যে

ওই ছেদ বিন্দু ধীরে ধীরে নেমে যাচ্ছে, আবার উলটোটাও সত্য। মাঝে মাঝে যদি বিশ্বাস না হয় মানে intuition এর উপর যদি ভরসা না থাকে তাহলে প্রমাণও করতে পার।

চাইলে প্রথমে decreasing এবং পরে increasing হতে পারে। অনেকে বলে থাকে ফাংশনকে unimodal বা convex বা concave হতে হয়, কিন্তু আমার মনে হয় কথাটি সঠিক না। যাই হোক, এরকম একটি ফাংশন দেওয়া থাকলে আমাদের বের করতে হবে এই ফাংশনের সর্বোচ্চ মান (বা উল্টো ক্ষেত্রে সর্বনিম্ন) কত। যেমন আমাদের সমস্যার ক্ষেত্রে মনে কর $x = -\infty$ সেকেন্দে সব বাসার দূরত্বের যোগফল অনেক বড়, এখন x কে ধীরে ধীরে ডান দিকে সরাতে থাকলে এই দূরত্বের যোগফল কমতে থাকবে, একসময় দেখবে এই মান সর্বনিম্ন হয় (হয়তো কিছুক্ষণ সর্বনিম্ন থাকবে), এর পর আবার বাঢ়তে থাকবে। অর্থাৎ এই সমস্যার ফাংশনটি টারনারি সার্চ প্রয়োগে জন্য উপযুক্ত। এখন কীভাবে এই পদ্ধতি খাটাতে হয়? বাইনারি সার্চের মতোই এখানে একটি সীমা থাকে, ধরা যাক $[lo, hi]$ হলো সেই সীমা। বাইনারি সার্চে আমরা ঠিক মাঝের বিন্দু অর্থাৎ $(lo + hi)/2$ নিয়েছিলাম। কিন্তু এখানে আমরা এর ভেতরে দুটি বিন্দু নিব, ধরা যাক তারা A ও B ($A < B$). আমরা দেখব যে কোন বিন্দুতে y এর মান কম। যদি A তে কম হয় তাহলে আমরা $[lo, B]$ তে পরবর্তীবারে খোঁজ করব, আর যদি B তে কম হয় তাহলে $[A, hi]$ এই সীমায় খোঁজ করব। এখন A আর B এর মান কী রকম হলে সবচেয়ে ভাল হয় তা আশা করি চিন্তা করলেই নেব করতে পারবে, তাও বলি এই দুটি মান এমন হতে হবে যেন পুরো সীমাকে সমান তিন ভাগে ভাস হয়, অর্থাৎ $A = (2lo + hi)/3$ এবং $B = (lo + 2hi)/3$. আর এই কাজ করবার করবে ত আশা করি বাইনারি সার্চের সেকশন পড়ে থাকলে বুঝতে পারছ। আর রানটাইম? সেটা তো \log জাতীয় কিছু হবে!

এখন তোমাদের মাঝে যারা জ্ঞানী তারা চিন্তা করছ আচ্ছা আমরা তো চাইলে তিন ভাগ না করে দুভাগ করে ঠিক মাঝে mid এ আমাদের ফাংশনের মান আর mid - 1 (বা double ডেটাটাইপে ক্ষেত্রে mid - delta) এ ফাংশনের মান এই দুটি দেখে বুঝে যেতে পারি যে আমাদের কোন দিকে যেতে হবে। হ্যাঁ ঠিক। আমরা বুঝব। এটাকে এভাবে কল্পনা করতে পার। আমরা mid এ গিয়ে স্পর্শকের ঢাল বা slope দেখবো। সেই ঢাল দেখে আমরা বুঝে যাব আমাদের অপটিমাল বিন্দু কোন দিকে আছে। তাহলে আমরা দুভাগ করি না কেন? কেন তিনভাগ করি? যদি আমাদের $f(x)$ একটি double টাইপের ডেটা দেয় তাহলে $f(mid)$ আর $f(mid - delta)$ দেখে কোনদিকে যাব সেটা ঠিক করা উচিত না, precision error এর জন্য। যেহেতু delta খুব ছোট তাই $f(mid)$ আর $f(mid - delta)$ ও খুব কাছাকাছি। ব্যাপারটা ঝুঁকিপূর্ণ হয়ে যায়। কিন্তু তুমি যদি $f(x)$ এর ডেরিভেটিভ (derivative) অর্থাৎ $f'(x)$ যদি বের করতে পার তাহলে কোন সমস্যা নাই। তখন আসলে আর টারনারি সার্চ থাকে না, ব্যাপারটা $f'(x)$ এর উপর বাইনারি সার্চ হয়ে যায়।

8.8 ব্যাকট্র্যাকিং (Backtracking)

ব্যাকট্র্যাকিং (backtracking) কোনো অ্যালগরিদম নয়, এটি একটি সাধারণ সমাধান পদ্ধতি। আমরা যা সাধারণভাবে বুঝে থাকি তাই কোড করাটিই হলো ব্যাকট্র্যাকিং। কিছু উদাহরণ দিলে জিনিসটি পরিষ্কার হবে।

8.8.1 সবরকম বিন্যাস বের করা (Permutation Generate)

মনে কর তোমাকে বলা হলো 1 হতে n এর সবরকম বিন্যাস বা 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 শুধুমাত্র For লুপ দিয়ে করা কঠিন। খেয়াল কর, আমাদের n কিন্তু কত সেটি শুরুতে বলে দেওয়া নেই। হয়তো বলা থাকবে যে, $n \leq 10$. যদি নির্দিষ্টভাবে বলা থাকত যে $n = 3$ বা $n = 4$ তাহলে হয়তো আমরা 3, 4টি নেস্টেড লুপ লিখে কাজটি করতাম, কিন্তু যখন n বড় হয়ে যায় তখন এত বড় জিনিস লেখা আমাদের জন্য কষ্টকর।^১ আবার লুপের মাধ্যমে করতে চাইলে প্রতিটি n এর জন্য আমাদের আলাদা আলাদা করে হয়তো কোড লিখতে হবে। শুধু লুপ দিয়ে করা একেবারে অসম্ভব না কিন্তু কষ্টকর। চিন্তা করে দেখ তোমাকে হাতে হাতে যদি এই কাজ করতে দেওয়া হয় তুমি কীভাবে করবে? যদি তুমি কোনো বিশেষ পদ্ধতি অনুসরণ না করে একে একে লিখতে থাকো নিজের ইচ্ছা মতো তাহলে n এর মান বড় হলে এক সময় দেখবে যে আর কী কী বাকি আছে তা বের করা বেশ কঠিন কাজ হয়ে যাবে। এখন যৌক্তিক উপায়টি কী? মনে কর $n = 3$ এর জন্য তুমি যা করবে তা হলো, তুমি দেখবে কোন কোন সংখ্যা এখনো বসানো হয়নি, এদের মধ্যে সবচেয়ে ছোটটি (1) নিয়ে প্রথম জায়গায় বসাবে। এখন বাকি সংখ্যাগুলো থেকে যেটি ছোট (2) সেটি দ্বিতীয় ঘরে বসাবে। একইভাবে তৃতীয় ঘরেও বসাও (3). আমরা $(1, 2, 3)$ পেয়ে গেলাম। এখন এর ঠিক আগেই যা বসিয়েছ তা মুছে ফেল, অর্থাৎ এর আগে যে আমরা 3 বসিয়েছিলাম তা সরাও। এখন দেখ এর পরে আর কোন সংখ্যা এখনো বসানো হয়নি। আসলে এই ক্ষেত্রে 3 এর পর আর কোনো সংখ্যা বাকি নেই, তাহলে এর আগে যেই সংখ্যা বসিয়েছিলে তা মুছে অর্থাৎ আমাদেরকে 2 মুছতে হবে। একইভাবে আমরা দেখব 2 এর পর কোন সংখ্যাটি এখনো বসানো হয়নি (3) তাকে বসিয়ে পরের ঘরে (তৃতীয় ঘরে) যাও। এখন দেখ কোন সবচেয়ে ছোট সংখ্যা এখনো বসানো হয়নি (2) তাকে বসাও। আমরা শেষ প্রান্তে চলে এসেছি এবং আরও একটি বিন্যাস $(1, 3, 2)$ পেয়ে গেলাম। এবার আমরা আবার ঠিক আগের সংখ্যা মুছে ফেলি (2). এক্ষেত্রে আর বসানোর মতো কোনো সংখ্যা বাকি নেই, সুতরাং আরও একধাপ আগে চলে যাই আর 3 কে মুছে ফেলি। এখন 3 এর থেকে বড় কোনো সংখ্যাও বাকি নেই সুতরাং আরও এক ধাপ পিছে গিয়ে 1 কে মুছে এর পরের বড় সংখ্যা 2 বসাই। দ্বিতীয় ঘরে এসে সবচেয়ে ছোট সংখ্যা 1 বসাই ও তৃতীয় ঘরে এসে 3 বসিয়ে আমরা $(2, 1, 3)$ পাব। এভাবে আমরা যদি করতে থাকি একে একে বাকি সবরকম বিন্যাস পেয়ে যাব।

এখন এটি হাতে করা যতখানি সহজ কাজ, কোডে করা অত সহজ নাও মনে হতে পারে। সত্যি কথা বলতে হাতে করার থেকে এই জিনিসটি প্রোগ্রাম লেখা সহজ। 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 লুপ দিয়ে করবে। কিন্তু আমাদের এই সমাধান পদ্ধতিতে কখনও কখনও তুমি আবার পিছিয়ে আসো। এই পিছিয়ে আসার কাজ আর যাই হোক লুপ দিয়ে খুব একটা সহজে করতে পারবে না। আরেকটু ভালো করে চিন্তা করলে দেখবে যে আমরা যা করছি তা হলো আমাদের কিছু সংখ্যা দেওয়া আছে, আমরা সবচেয়ে ছোট সংখ্যা বসিয়ে বাকি সংখ্যা দিয়ে পরের

^১ হ্যাঁ, কষ্টকর কিন্তু অসম্ভব না। মনে হয় দুটি লুপ দিয়ে যেকোনো n এর জন্য সবরকম বিন্যাস বের করা সম্ভব।

অংশে বিন্যাস বানাচ্ছি। শেষ হয়ে গেলে পরবর্তী সংখ্যা বসিয়ে বাকি সংখ্যা দিয়ে আবার বিন্যাস বানাচ্ছি। অর্থাৎ জিনিসটি এমন যে, একটি ব্ল্যাকবক্স (black box) আছে যাকে আমরা সংখ্যা দিলে সে বিন্যাস বানাবে। এর জন্য সে সবচেয়ে ছোট সংখ্যাকে রেখে দিবে এর পর বাকিসংখ্যাগুলোকে সে আবার একই ধরনের আরেক ব্ল্যাকবক্সে দিয়ে দেবে। ওই অন্য ব্ল্যাকবক্স এর কাজ হয়ে গেলে তুমি পরবর্তী বড় সংখ্যা রেখে দিয়ে বাকি সব সংখ্যা দিয়ে ওই অন্য ব্ল্যাকবক্স কে আবারও কল করবে। আশা করি বুঝতে পারছ যে এই ব্ল্যাকবক্সটি হলো রিকার্সিভ ফাংশন। কিন্তু এই রিকার্সিভ ফাংশনটি ফিবোনাচি বা ফ্যাট্টোরিয়ালের মতো সহজ নয়। যখনই একটি ফাংশন লিখবে আগে চিহ্ন করে দেখ তোমার এই ফাংশনকে কী দিতে হবে, সে কী দিবে আর সে কী করবে? একে একে এই প্রশ্নগুলোর উত্তর দেওয়া যাক। কী দিতে হবে? - যেসব সংখ্যা এখনো বাকি আছে তাদের দিতে হবে, কী দিবে? - কিছু সংখ্যা ইতোমধ্যেই ঠিক করা হয়েছে, বাকি সংখ্যাগুলোকে বিন্যস্ত করে যেসব বিন্যাস হয় তাদের সবাই যেন প্রিন্ট হয় তা নিশ্চিত করতে হবে। একটু ভাবলে তোমরা বুঝবে যে যেসব সংখ্যা বসানো হয়নি সেগুলো ব্ল্যাকবক্সে পাঠানোর সঙ্গে সঙ্গে তোমরা এখন পর্যন্ত কার পরে কাকে বসিয়েছ সেটাও পাঠাতে হবে। সুতরাং ব্ল্যাকবক্সকে দুটি জিনিস দিতে হবে, ১. এখন পর্যন্ত কোন সংখ্যাগুলো বসানো হয়েছে এবং কী ক্রমে ২. কোন কোন সংখ্যা এখনো বসানো বাকি আছে। base কেইস হলো যখন সব সংখ্যা বসানো হয়ে যাবে তখন এবং সেই সংখ্যার ক্রম আমরা প্রিন্ট করব। এখন পর্যন্ত আমরা দেখেছি কীভাবে একটি ফাংশন একটি integer বা double টাইপের ডেটা পাঠানো যায় কিন্তু একটি অ্যারে কীভাবে পাঠাতে হয় তা আমাদের অজানা। সাধারণ নিয়ম হচ্ছে তুমি পয়েন্টার (pointer) ব্যবহার করে পাঠাবে কিন্তু তোমাদের বেশির ভাগই পয়েন্টার ভয় কর (বলতে দ্বিধা নেই যে আমি নিজেও পয়েন্টার বেশ ভয় করি)। আরেকটি উপায় হচ্ছে ভেক্টর (vector) ব্যবহার করা। তবে এটি বেশ ধীরগতির। আরেকটি উপায় হলো গ্লোবাল (global) অ্যারে ব্যবহার করা। আমরা দুধরনের অ্যারে রাখব। একটি অ্যারেতে থাকবে কোন সংখ্যা ব্যবহার করা হয়েছে কোন সংখ্যা ব্যবহার করা হয়নি তা (০ মানে ব্যবহার করা হয়নি, ১ মানে ব্যবহার করা হয়েছে)। আরেকটি অ্যারেতে এখন পর্যন্ত বসানো নম্বরগুলো পর পর থাকবে। ব্ল্যাকবক্সের ভেতরে তুমি একে একে ১ হতে n পর্যন্ত সংখ্যাগুলো যাচাই করবে যে আগে বসানো হয়েছে কিনা যদি না বসানো হয়ে থাকে তাহলে সেই সংখ্যা বসাবে এবং এটিও লিখে রাখবে যে এই সংখ্যাটি ব্যবহার করা হয়ে গেছে। এবার পরবর্তী ব্ল্যাকবক্সের কাছে যাবে, সেও একইভাবে কাজ করবে। কাজ শেষে সে যখন ফিরে আসবে, তখন দুটি জিনিস করতে হবে তা হলো বসানো সংখ্যাকে সরাতে হবে আর তুমি যে লিখে রেখেছ যে এই সংখ্যা ব্যবহার হয়েছে সেটি পরিবর্তন করে লিখতে হবে যে এটি এখনো ব্যবহার করা হয়নি। এই কাজ যদি না কর তাহলে দেখবে মাত্র একটি বিন্যাস প্রিন্ট করেই তোমার প্রোগ্রাম শেষ হয়ে যাবে। তাহলে কি ব্ল্যাকবক্সে আমাদের কিছুই পাঠানোর দরকার নেই? আসলে দরকার নেই তবে আমরা আমাদের কোডকে সহজ করার জন্য একটি জিনিস পাঠাব আর তা হলো, আমরা কোন ঘরে এখন সংখ্যা বসাব সেটা। যদিও এই জিনিস আমরা কোন কোন সংখ্যা ব্যবহার করা হয়েছে সেই অ্যারে থেকে খুব সহজেই বের করতে পারি কিন্তু আমরা যদি এই সংখ্যা প্যারামিটার হিসাবে পাঠাই তাহলে আমাদের কাজ অনেক সহজ হয়ে যাবে। আরও একটি জিনিস পাঠাতে হবে আর তা হলো n এর মান, তবে এটি চাইলে গ্লোবালও রাখতে পার। বিন্যাস প্রিন্ট করার প্রোগ্রামটি দেখানোর আগে আরেকটি জিনিস চিন্তা করে দেখতে

পার-বসানো সংখ্যা কি সরানোর আদৌ দরকার আছে? শুধু যে সংখ্যা অব্যবহৃত আছে সেই কাজটি করাই কি যথেষ্ট নয়? তোমাদের জন্য বিন্যাস প্রিন্ট করার প্রোগ্রাম কোড ৪.৯ এ দেওয়া হলো।

কোড ৪.৯: 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++)
10            printf("%d ", number[i]);
11        printf("\n");
12        return;
13    }
14
15    for(i = 1; i <= n; i++) if(!used[i])
16    {
17        used[i] = 1;
18        number[at] = i;
19        permutation(at + 1, n);
20        used[i] = 0;
21    }
22 }
```

8.8.2 সবরকম সমাবেশ বের করা (Combination Generation)

n টি সংখ্যা দেওয়া থাকলে তাদের থেকে k টি করে সংখ্যা নিয়ে সবরকম সমাবেশ বা combination প্রিন্ট করতে হবে। যেমন $n = 3$ ও $k = 2$ হলে আমাদের প্রিন্ট করতে হবে: $(1, 2), (1, 3)$ এবং $(2, 3)$. আমরা যেমন তেমনভাবে চিন্তা না করে আগের মতো যৌক্তিক চিন্তা করব। দুভাবে আমরা এই প্রবলেমের সমাধান খেয়াল করতে পারি। প্রথম উপায়টি হলো, আমরা একে একে 1 থেকে n পর্যন্ত যাব আর ঠিক করব এই সংখ্যাকে নিব কি নিব না। একদম শেষে গিয়ে যদি আমরা দেখি যে আমরা k টি সংখ্যা নিয়ে ফেলেছি তাহলে তো হয়েই গেল। আর না হলে আমরা

ফেরত যাব এটি প্রিন্ট না করে। এখন খেয়াল কর, এই সমাধান সঠিক থাকলেও আমাদের সময় কিন্তু অনেক বেশি লাগবে। আমরা প্রতিটি সংখ্যার কাছে দিয়ে গিয়ে একবার নিছিঃ আরেকবার নিছিঃ 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 নেওয়ার ব্যাপারে এখনো সিদ্ধান্ত নেওয়া বাকি আছে তাদের সবাইকে নিলেও যদি আমাদের k টি সংখ্যা না হয় তাহলে আর পরবর্তী ফাংশন কল এর দরকার নেই। এখান থেকেই ফিরে গেলে হয়। আমাদের এভাবে সমাধানের প্রোগ্রাম কোড ৪.১০ এ দেওয়া হলো। এই কোডের লাইন ৭ এর জন্য আমাদের প্রোগ্রাম $O(2^n)$ হতে $O(\binom{n}{k})$ হবে।

কোড ৪.১০: 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
10    // bit more faster in such case you dont
11    // need following if(left) condition
12    if(at == n + 1)
13    {
14        for(i = 1; i <= k; i++) printf("%d ", number[i]);
15        printf("\n");
16        return;
17    }
18
19    if(left)
20    {
21        number[k - left + 1] = at;
22        combination(at + 1, left - 1);
23    }
24
25    combination(at + 1, left);

```

দ্বিতীয় পদ্ধতির ক্ষেত্রে আমরা প্রত্যেক ঘরে যাব, এরপর এখানে আগের ঘরগুলোতে বসানো হয়নি এমন একটি সংখ্যা বসাব এবং এভাবে একে একে k ঘরে যখন আমরা সংখ্যা বসিয়ে ফেলব তখন আমরা একটি সংখ্যার সমাবেশ পেয়ে যাব। তবে এক্ষেত্রে আমাদের $(1, 2)$ এর সঙ্গে সঙ্গে $(2, 1)$ ও প্রিন্ট হয়ে যাবে। তোমরা যদি n টি সংখ্যা থেকে k টি করে সংখ্যা নিয়ে তাদের বিন্যাস প্রিন্ট করতে চাও তাহলে এভাবে করলেই হবে। কিন্তু যদি সমাবেশ প্রিন্ট করতে চাও তাহলে আরও একটি কাজ করতে হবে আর তা হলো, তুমি নতুন যেই সংখ্যা বসাবে সেটি যেন আগের সংখ্যা থেকে বড় হয়। এই কাজ করার জন্য সবচেয়ে ভালো উপায় হলো তুমি প্যারামিটার হিসেবে সর্বশেষ বসানো সংখ্যাটি পাঠিয়ে দাও এরপর যখন তুমি লুপ চালাবে তখন 1 থেকে না চালিয়ে এই সংখ্যা থেকে চালালেই হবে। এই সমাধানটি আমাদের প্রথম সমাধানের থেকে একটু হলেও ভালো বলা যায়। কারণ, আমাদের আগের সমাধান worst কেইসে রিকার্শনের n গভীরতা পর্যন্ত যায়, কিন্তু আমাদের দ্বিতীয় সমাধান k গভীরতা পর্যন্ত যাবে। আমাদের দ্বিতীয় সমাধানের প্রোগ্রামের কোড ৪.১১ এ দেওয়া হলো। আশা করি 14 নম্বর লাইন বাদে বাকি কোডটুকু বুঝতে তোমাদের সমস্যা হবে না। আমরা যেভাবে এর আগের বার একটি if দিয়ে $O(2^n)$ হতে $O(\binom{n}{k})$ আনা হয়েছে এখানেও সেরকম কাজ করা হয়েছে। তবে পার্থক্য হলো আমরা এর আগে আলাদাভাবে শর্ত যাচাই করে ছিলাম, এবার for লুপের উপরের সীমা (upper bound) হিসেবে এই কাজটি করেছি। এই দুই উপায়ের মধ্যে তেমন কোনো পার্থক্য নেই, আমরা দুই কোডে দুভাবে করে দেখালাম তোমরা যাতে দুই উপায়ের সঙ্গে পরিচিত থাক সেজন্য। তোমরা ভেবে দেখতে পার $i \leq n - k + at$ এই শর্তটি কোথা 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 $k - at \leq n - i \rightarrow i \leq n - k + at$.

কোড ৪.১১: combination2.cpp

```

1 int number[20];
2 int n, k;
3
4 //call with: permutation(1, 0)
5 void combination(int at, int last)
6 {
7     if(at == k + 1)
8     {
9         for(i = 1; i <= k; i++) printf("%d ", number[i]);
10        printf("\n");
11        return;
12    }
13}

```

```

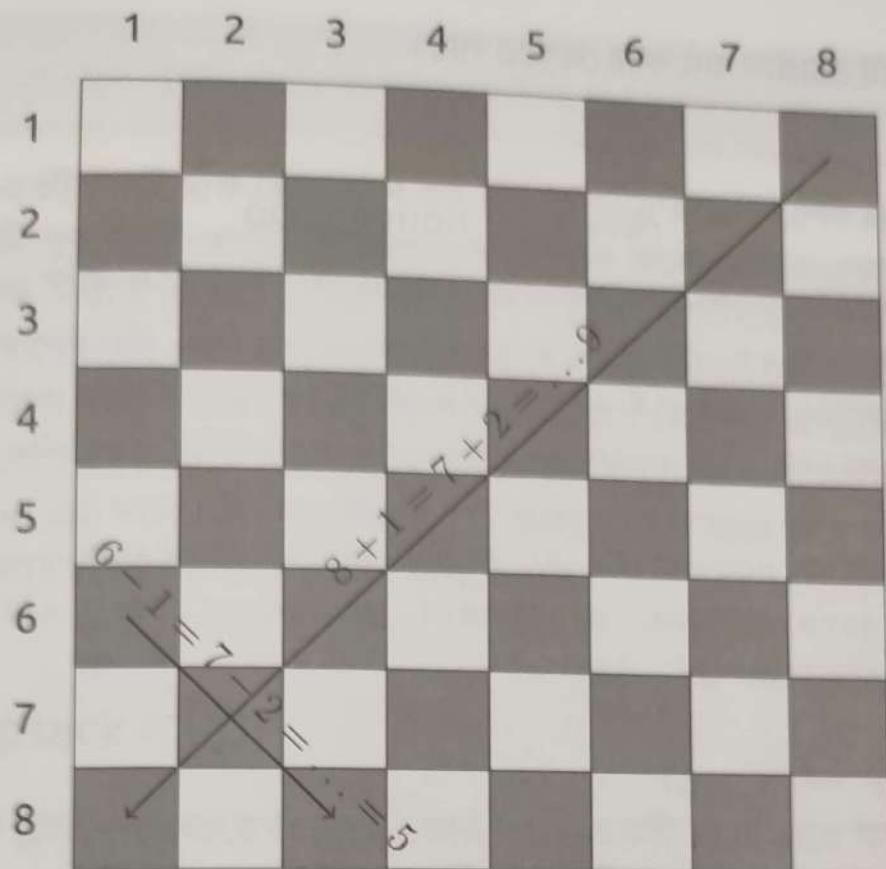
13
14     for(i = last + 1; i <= n - k + at; i++)
15     {
16         number[at] = i;
17         combination(at + 1, i);
18     }
19 }

```

8.8.3 Eight Queen সমস্যা

এটি একটি বিখ্যাত সমস্যা। তোমরা যারা দাবা খেলা জানো, আশা করি তাদের বুকাতে কোনো সমস্যা হবে না। সমস্যাটি হলো একটি দাবার বোর্ড (দাবা বোর্ড 8×8 হয়ে থাকে) 8 টি queen (আমরা অনেক সময় বাংলায় এদের মন্ত্রী বলে থাকি) কে কতভাবে বসানো যায় যেন কোনো queen ই অন্য কোনো queenকে আক্রমণ না করে। একটি queen অপর আরেকটি queen কে আক্রমণ করতে পারবে যদি তারা একই সারি (row) বা একই কলাম (column) বা একই কর্ণ (diagonal) বরাবর থাকে। সমস্যাটি খুব একটা কঠিন না। আমাদের যা করতে হবে তা হলো প্রত্যেক সারিতে গিয়ে গিয়ে একটি করে queen বসাতে হবে, সব সারিতে queen বসানো শেষ হয়ে গেলে আমরা দেখব কোনো একটি queen অপর আরেকটি queen কে আক্রমণ করে কিনা। যেহেতু আমরা প্রতি সারিতে একটি করে queen বসাচ্ছি সেহেতু কোনো দুটি queen একই সারিতে আছে কিনা তা দেখার দরকার নেই। শুধু একই কলামে আছে কিনা তা দেখতে হবে আর একই কর্ণ বরাবর আছে কিনা তা। একই কলামে আছে কিনা এটি যাচাই করা খুব সহজ, কিন্তু একই কর্ণ বরাবর আছে কিনা সেটি দেখা বেশ tricky. দুধরনের কর্ণ হতে পারে। এক ধরনের কর্ণ উপরের বাম দিক থেকে শুরু করে নিচের ডান দিকে যায় অন্য কর্ণগুলো উপরের ডান দিক থেকে নিচের বাম দিকে যায়। মনে করি আমাদের দাবা বোর্ডের সারিগুলো উপর থেকে নিচে 1 হতে 8 পর্যন্ত নম্বর করা এবং কলামগুলো বাম থেকে ডান দিকে 1 হতে 8 পর্যন্ত নম্বর করা (চিত্র 8.2)। এখন একটু খেয়াল করলে দেখবে যেসব কর্ণ উপরের বাম দিক থেকে নিচের ডান দিকে যায় সেসব কর্ণে থাকা ঘরগুলোর সারি ও কলামের বিয়োগফল একই হয় এবং যেসব কর্ণ উপরের ডান দিক থেকে নিচের বাম দিকে যায় তাতে থাকা ঘরগুলোর সারি ও কলামের যোগফল একই হয়।

তাহলে আমরা সব সারিতে queen বসানোর পর দুটি দুটি করে queen নিয়ে দেখব যে তাদের কলাম বা কর্ণ একই কিনা। এভাবে করলে একটি সমস্যা হলো, এমনও হতে পারে যে আমরা প্রথম দুটি queen কে একই কলামে বসিয়ে ফেলেছি এর পর বাকি 6 টি queen কে আমরা কিন্তু অনেকভাবে বসাতে পারি, যেভাবেই বসাই না কেন আমরা কোনো বৈধ সমাধান পাব না। সুতরাং আমরা প্রতিবার queen বসানোর আগে বা পরে যাচাই করে দেখতে পারি যে এখন পর্যন্ত বসানো queen গুলো কেউ কারও সঙ্গে আক্রমণাত্মক অবস্থানে আছে কিনা। একটু চিন্তা করলে দেখবে, আসলে সব queen জোড়ায় জোড়ায় যাচাই করার দরকার নেই। শুধুমাত্র নতুন বসানো queen



নকশা ৪.২: দাবা বোর্ড

এর সঙ্গে আগের বসানো queen গুলোকে যাচাই করলেই হয়। আরও একটু চিন্তা করলে দেখবে এখানে আমাদের ধরে ধরে আগে বসানো প্রতিটি queen এর সঙ্গে চেক করার দরকার হবে না, যদি আমরা এমন কিছু অ্যারে রাখি যারা বলে দিবে যে অমুক কলাম বা অমুক কর্ণে কোনো queen আছে কিনা। অর্থাৎ আমরা কোনো একটি queen বসানোর সময় অ্যারেগুলোতে লিখে দিব যে অমুক কলাম, অমুক কর্ণে queen বসেছে। তাহলে নতুন queen বসানোর আগে শুধু আমরা চেক করে দেখব যে যেই কলাম বা কর্ণে আমরা queen বসাতে চাচ্ছি তা আদৌ ফাঁকা আছে কিনা। অর্থাৎ আমাদের কোনো লুপ লাগবে না, শুধু if-else দিয়েই যাচাই হয়ে যাবে যে আমরা যেই কলাম বা কর্ণে queen বসাতে চাচ্ছি তা ফাঁকা আছে কিনা। খেয়াল কর আমরা কিন্তু ধীরে ধীরে আমাদের সমাধানকে যতটুকু সন্তুষ্ট অপটিমাইজেশন করছি। আমরা আমাদের প্রাথমিক সমাধানের থেকে অনেক দূরে চলে এসেছি ঠিকই কিন্তু আমরা এমন সব কিছু করেছি যাতে করে আমাদের প্রোগ্রাম আগের তুলনায় অনেক কম সময় নেয়। তোমরা যখন এই জিনিস কোড করে দেখবে তখন প্রতিটি ইমপ্রুভমেন্ট যোগ করার আগে ও পরে দেখবে তোমাদের কোড কত সময় নেয়। এতে করে তোমরা বুবাবে এসব অপটিমাইজেশন দেখতে অনেক সহজ হলেও এরা পারফর্ম্যান্সের দিক থেকে অনেক অনেক এগিয়ে দেয় তোমাকে। হয়তো 8×8 বোর্ডের জন্য এসব অপটিমাইজেশনের প্রভাব তুমি নাও বুবতে পার। তোমরা চাইলে 9×9 , 10×10 এসব বোর্ডেও যাচাই করে দেখতে পার। আমরা এখানে আলোচনা করা সব অপটিমাইজেশন ব্যবহার করে লেখা প্রোগ্রাম কোড ৪.১২ এ দেখলাম। যদি তোমরা $n = 8$ দাও তাহলে 8×8 বোর্ড হবে, বা তোমরা চাইলে অন্যান্য মাপের

বোর্ডের জন্যও এই প্রোগ্রাম রান করে দেখতে পার।

কোড 8.12: nqueen.cpp

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

তোমরা যদি এতটুকুতেই হাঁফ ছেড়ে মনে কর যাক অপটিমাইজেশন শেষ হলো, তাহলে বলে রাখি আরও একটি খুব সহজ অপটিমাইজেশন আছে যার ফলে তোমরা তোমাদের রান্টাইম কে একদম অর্ধেক করতে পারবে। চিন্তা করে দেখ সেই অপটিমাইজেশনটি কী! আসলে অপটিমাইজেশনের শেষ নেই। ব্যাকট্র্যাকিংয়ের ক্ষেত্রে যে যত অপটিমাইজেশন যোগ করতে পারবে তার কোড তত ভালো কাজ করবে। তবে খেয়াল রাখতে হবে সেই সব অপটিমাইজেশনের জন্য আবার না অনেক বেশি সময় লেগে যায়! যেমন আমরা যদি বসানোর সময় শুধু অ্যারেতে না দেখে আগের সব queen এর সঙ্গে যদি চেক করতে যাই তাহলে দেখা যাবে অনেক বেশি সময় লেগে যাবে। সুতরাং আমাদের এই জিনিসও খেয়াল রাখতে হবে।

8.8.8 Knapsack সমস্যা

মনে কর এক চোর চুরি করতে গিয়েছে। তার কাছে একটি থলে আছে যাতে খুব জোর W ওজনের জিনিস নেওয়া যাবে। এখন সেই চোর চুরি করতে গিয়ে n টি জিনিস দেখতে পেল। তাম জিনিসের ওজন w_i এবং ওই জিনিস বিক্রি করলে সে v_i টাকা পাবে। এখন সবচেয়ে বেশি কত টাকার জিনিস চোর চুরি করতে পারবে যেন সেসব জিনিসের মোট ওজন W এর থেকে বেশি না হয়? এক্ষেত্রে সীমান্তলো খুব গুরুত্বপূর্ণ। $n \leq 50$, $w_i \leq 10^{12}$ এবং $v_i \leq 10^{12}$. তাহলে আমরা কী করব? আগের মতো একে একে ১ থেকে n পর্যন্ত যাব, কোন জিনিস নিব, কোন জিনিস নিব না শেষে গিয়ে দেখব যে ওজন W এর থেকে বেশি হয়েছে কিনা। বেশি হলে এটি সমাধান হবে না, আর তা না হলে আমরা এরকম সকল সমাধানের মধ্যে যেক্ষেত্রে দাম সবচেয়ে বেশি হয় সেটাই সমাধান হবে। বুঝতেই পারছ এত সহজ সমাধান হলে এখানে আর আলোচনার কিছু থাকত না! তাহলে এই সমাধানের কামেলা কোথায়? প্রথমে হিসাব করে দেখ এই সমাধানের রান্টাইম কত? $O(2^n)$. অবশ্যই $n \leq 50$ এর জন্য এটি একটি বিশাল মান। তাহলে আমরা কীভাবে সমাধান করতে পারি? আমাদেরকে আগের eight queen সমস্যার মতো কিছু অপটিমাইজেশন বের করতে হবে। প্রথমত আগের মতো আমরা প্রতিটি জিনিস নেওয়ার পরেই দেখব এর ওজন W এর থেকে বেশি হয়ে গেছে কিনা, তা হয়ে গেলে পরের জিনিসগুলো দেখার কোনো মানে নেই, এখান থেকেই ফেরত যাওয়া উচিত। আরও কী কী অপটিমাইজেশন 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 মতো কাজ। আবার দেখ, যেসব ওজন বাকি আছে তাদের মধ্যে 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 অপটিমাইজেশনই হয়ে গেছে। দাম দিয়ে কী কিছু অপটিমাইজেশন করা যায়? যায়, ধর এখন পর্যন্ত আমরা যেসব সমাধান বের করেছি তার মধ্যে 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 দামসহ যদি V এর থেকে বেশি না হয় তার মানে এখান থেকে আরও এগিয়ে কোনো লাভ নেই। এরকম নানা অপটিমাইজেশন প্রয়োগ করলে আমাদের এই প্রোগ্রাম ও রানটাইম অনেক কমে যায়। ব্যাকট্র্যাকিংয়ের ক্ষেত্রে মূল সমস্যা হলো এর রানটাইম আসলে দেখে করা সম্ভব না। প্রতিটি অপটিমাইজেশন লিখার পর তোমাকে প্রোগ্রাম চালিয়ে চালিয়ে দেখতে হবে কী রকম সময় নেয় না নেয়।

৪.৫ প্রোগ্রামিং সমস্যা

৪.৫.১ অনুশীলনী

খুব সহজ

ঃ UvaLive 6983

সহজ

ঃ UvaLive 6802 ঃ UvaLive 6809 ঃ UvaLive 6836 ঃ UvaLive 6884 ঃ UvaLive 6906 ঃ UvaLive 6912* ঃ UvaLive 6948 ঃ UvaLive 6954 ঃ UvaLive 6973 ঃ UvaLive 6990* ঃ UvaLive 6998 ঃ UvaLive 7000 ঃ UvaLive 7014 ঃ UvaLive 7047 ঃ UvaLive 7058 ঃ UvaLive 7086

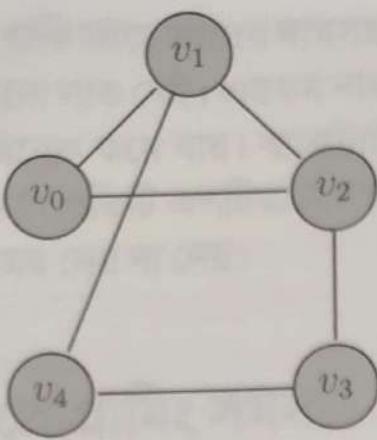
সামান্য কঠিন

ঃ UvaLive 6850 ঃ UvaLive 6999 ঃ UvaLive 7002 ঃ UvaLive 7006 ঃ UvaLive 7009 ঃ UvaLive 7053

অধ্যায় ৫

ডেটা স্ট্রাকচার

অ্যালগরিদম হচ্ছে একটি সমস্যা সমাধানের পথ আর ডেটা স্ট্রাকচার (Data Structure) হচ্ছে ডেটাকে সাজিয়ে রাখার জিনিস। অনেক সময় কোনো একটি অ্যালগরিদমের এফিসিয়েন্সি (efficiency) ডেটা স্ট্রাকচারের উপর নির্ভর করে। খুব সহজ একটি উদাহরণ দেয়া যাক। মনে কর তোমাকে একে একে একটি করে সংখ্যা দেওয়া হবে 1 থেকে n এর মধ্যে, তোমাকে বলতে হবে এই সংখ্যাটা এর আগে এসেছিল কিনা। তুমি কীভাবে করবে? একটি উপায় হলো সংখ্যার একটি অ্যারে রাখা। যখন কোনো সংখ্যা আসবে তখন ওই অ্যারেতে খুঁজে দেখ এর আগে ওই সংখ্যা এসেছিল কিনা যদি না থাকে তাহলে এই অ্যারের শেষে এই সংখ্যাটা রাখ। আরেকটি উপায় হলো এমন একটি অ্যারে রাখ যেখানে তোমার লেখা থাকবে যে কোনো একটি সংখ্যা এর আগে এসেছিল কিনা। খেয়াল কর এখানে তুমি সংখ্যাগুলো রাখবে না শুধু কোনো একটি সংখ্যা এসেছিল কিনা তা রাখবে। এটা করা খুব সহজ। একটি অ্যারেতে তুমি শুধু 0 বা 1 রাখবে এসেছিল কি আসেনি তার উপর ভিত্তি করে। এখন এই দুভাবে ডেটা রাখার সুবিধা অসুবিধা দুইই আছে। প্রথম পদ্ধতিতে আমাদের যদি m টি সংখ্যা দেওয়া হয় তাহলে $O(m)$ মেমোরী লাগবে এবং প্রতিবার প্রশ্নের জন্য তোমার $O(m)$ সময়ও লাগবে। কিন্তু পরের পদ্ধতিতে আমাদের $O(n)$ মেমোরী লাগবে যেখানে n হলো আমাদের ইনপুটের সবচেয়ে বড় মান কিন্তু আমাদের প্রতিটি প্রশ্নের জন্য মাত্র $O(1)$ সময় লাগবে। দুই অ্যালগরিদমের মূল নীতি কিন্তু একই, যেই সংখ্যা দেওয়া হয়েছে তা আছে কিনা দেখতে হবে, না থাকলে সেই সংখ্যা আমাদের ঢুকিয়ে রাখতে হবে। কিন্তু আমাদের ডেটা সংরক্ষণের ভিত্তির কারণে আমাদের রানটাইম বা মেমোরীর প্রয়োজনীয়তা কিন্তু আলাদা হয়ে গেছে। একটি পদ্ধতি বড় n কিন্তু ছোট m এ ভালো কাজ করবে, আরেকটি ঠিক উল্টো। সুতরাং আমরা কেমন করে ডেটাকে সংরক্ষণ করছি তা অনেক গুরুত্বপূর্ণ।



(a) গ্রাফ (Graph)

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

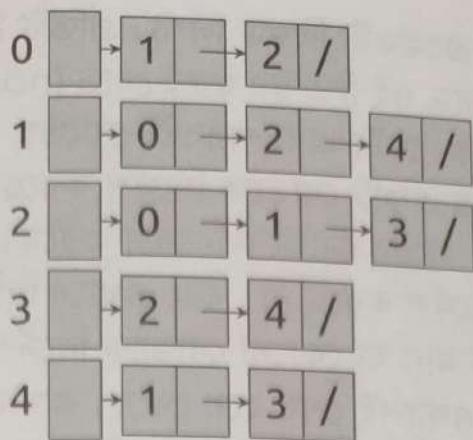
(b) অ্যাডজাসেন্সি ম্যাট্রিক্স (Adjacency Matrix)

নকশা ৫.১: একটি গ্রাফের অ্যাডজাসেন্সি ম্যাট্রিক্সের উপস্থাপন

৫.১ লিঙ্কড লিস্ট (Linked List)

মনে কর আমরা ফেসবুকের n জনের মাঝের সম্পর্ক একটি ডেটা স্ট্রাকচারে রাখতে চাই। কীভাবে রাখব? নানা উপায় আছে, একে একে আমরা তিনটি উপায় দেখি এবং তাদের সুবিধা অসুবিধাও।

- উপায় ১: একটি $n \times n$ আকৃতির $0 - 1$ ম্যাট্রিক্স রাখা। যদি i তম মানুষ আর j তম মানুষের মধ্যে বন্ধুত্ব থাকে তাহলে ম্যাট্রিক্সের $[i][j]$ তম জায়গায় 1 রাখব অন্যথায় 0 রাখব। এই পদ্ধতিতে আমাদের মেমোরী লাগবে $O(n^2)$ কিন্তু কোনো দুজনের মধ্যে বন্ধুত্ব আছে কিনা বা যদি নতুন দুজনের মধ্যে বন্ধুত্ব হয় তাহলে আমাদের ডেটা স্ট্রাকচার আপডেট করতে সময় লাগবে $O(1)$. একে আমরা অ্যাডজাসেন্সি ম্যাট্রিক্স (adjacency matrix) বলে থাকি। চিত্র ৫.১ এ আমরা একটি গ্রাফের অ্যাডজাসেন্সি ম্যাট্রিক্স দেখালাম। প্রতিটি edge এর জন্য খেয়াল কর ম্যাট্রিক্সের ওই জায়গায় 1 আছে। আবার যেখানে যেখানে edge নেই সেখানে সেখানে 0. যেমন 0 আর 1 এর মধ্যে edge আছে তাই ম্যাট্রিক্সের $[0][1]$ আর $[1][0]$ এ 1 আছে। আবার 0 আর 4 এর মধ্যে edge নেই তাই $[4][0]$ ও $[0][4]$ এ 0.
- উপায় ২: একটি $n \times n$ আকৃতির ম্যাট্রিক্স রাখব। i তম সারিতে থাকবে i তম মানুষের সঙ্গে যারা যারা বন্ধু আছে তাদের একটি তালিকা বা লিস্ট। আমরা আরেকটি আরেতে লিখে রাখব যে কোনো একজন মানুষের কত জন বন্ধু আছে। সুতরাং আমরা জানি $[i][1]$ থেকে $[i][\text{friend}[i]]$ পর্যন্ত i তম মানুষের সব বন্ধু আছে, এখানে $\text{friend}[i]$ হলো i তম মানুষের বন্ধুর সংখ্যা। এই পদ্ধতিতে আমাদের মেমোরী লাগবে আগের মতো $O(n^2)$ কিন্তু কোনো দুজনের মধ্যে বন্ধুত্ব আছে কিনা তা নির্ণয়ের জন্য আমাদের সময় লাগবে $O(\text{friend}[i])$ কারণ আমাদের পুরো বন্ধুর তালিকা যাচাই করে দেখতে হবে। কিন্তু নতুন একজন বন্ধু যোগ করতে আমাদের সময় লাগবে $O(1)$ কারণ আমরা কিন্তু খুব সহজেই $\text{friend}[i]$ এর মান এক বাড়িয়ে দিয়ে ম্যাট্রিক্সের $[i][\text{friend}[i]]$ স্থানে নতুন বন্ধুকে রাখতে



নকশা ৫.২: একটি গ্রাফের অ্যাডজাসেন্সি লিস্টের উপস্থাপন

পারি। একে অ্যাডজাসেন্সি লিস্ট (adjacency list) এর অ্যারে ইমপ্লিমেন্টেশন (array implementation) বলে।

উপায় ৩: আমরা আলাদা আলাদা করে n জন বন্ধুর জন্য তালিকা রাখব। আগে থেকেই $n \times n$ ঘরের জায়গা ডিক্লেয়ার (declare) না করে আমরা নতুন নতুন বন্ধু এলে তাদের জন্য ডায়নামিক মেমোরী অ্যালোকেশন (dynamic memory allocation) করে তাদের লিস্টে ঢুকিয়ে দেব। এটি হলো অ্যাডজাসেন্সি লিস্টের লিঙ্কড লিস্ট ইমপ্লিমেন্টেশন (linked list implementation)। আগের পদ্ধতির সঙ্গে পার্থক্য শুধু মেমোরী কমপ্লেক্সিটিতে। এই পদ্ধতিতে যতগুলো বন্ধু ঠিক তত মেমোরী লাগবে। চিত্র ৫.২ এ আমরা অ্যাডজাসেন্সি লিস্টের সাহায্যে একই গ্রাফের উপস্থাপন দেখালাম।

তাহলে লিঙ্কড লিস্ট হলো অ্যারের ভাই। অ্যারেতে আগে থেকেই এর দৈর্ঘ্য আমাকে বলে দিতে হয় কিন্তু লিঙ্কড লিস্টে তার দরকার হয় না। কিন্তু আমরা বেশির ভাগই ডায়নামিক মেমোরী অ্যালোকেশন (dynamic memory allocation) কে খুব ভয় করে থাকি। ডায়নামিক মেমোরী অ্যালোকেশনকে পাশ কাটানোর একটি উপায় হলো, প্রোগ্রামিং কন্টেন্সের 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 লিস্টে সবচেয়ে বেশি কতগুলো উপাদান দরকার হতে পারে। ঠিক তত আকার আগে থেকে ডিক্লেয়ার (declare) করে রাখলেই হয়। তোমরা ভাবতে পার যে এটা তো অ্যারেই হয়ে গেল। না, মনে কর এর আগের উদাহরণে আমরা বলে দিলাম মোট 10^6 এর বেশি বন্ধু জোড়া হবে না, কিন্তু মোট 10^4 জন ভিন্ন মানুষ হতে পারে। অর্থাৎ, তুমি যদি এটি অ্যারে পদ্ধতিতে করতে চাও তাহলে তোমাকে $10^4 \times 10^4$ মেমোরী আগে থেকেই ডিক্লেয়ার করে রাখতে হচ্ছে। কিন্তু তুমি যদি লিঙ্কড লিস্ট পদ্ধতিতে কর তাহলে শুধু 10^6 আকারের মেমোরী ডিক্লেয়ার করলেই হয়ে যাবে। এভাবে তোমরা ডায়নামিক মেমোরী অ্যালোকেশন না করেই আগে থেকে একটি অ্যারে থেকে একে একে জায়গা নিয়ে কাজ করতে পারবে। আমরা এখানে `define` করা একটি অ্যারে থেকে একে একে জায়গা নিয়ে কাজ করতে পারবে। আমরা এখানে এই অ্যারে পদ্ধতিতে কীভাবে লিঙ্কড লিস্ট করা যায় তা দেখাব, তোমরা নিজেরা চাইলে পয়েন্টার (pointer) ব্যবহার করে কীভাবে করা যায় তা চিন্তা করে দেখতে পার।

চিত্র ৫.২ এ প্রতিটি লাইন একেকটি লিঙ্কড লিস্ট। প্রতিটি লিঙ্কড লিস্টের একটি করে হেড (head) থাকে। সাধারণত এই হেড ওই লিস্টের শুরুর নোড (node) কে নির্দেশ বা point করে। প্রতিটি নোডের দুটি জিনিস থাকে, ডেটা এবং পরবর্তী নোডের পয়েন্টার। যেমন চিত্র ৫.২ এর প্রথম অ্যাডজাসেন্সি লিস্টের হেড একটি নোডকে নির্দেশ করছে বা পয়েন্ট (point) করছে যা ডেটা হলো ১ এবং এর পয়েন্টার পরের নোডকে নির্দেশ করছে। এর পরের নোডে আছে ২ এবং এর পয়েন্টার আসলে কাউকে নির্দেশ করছে না। বরং এখানে টার্মিনাল (terminal) মান আছে। টার্মিনাল মান হলো এমন একটি মান যা কোনো নোডকে নির্দেশ করে না কিন্তু এই মান দেখলে আমরা বুঝি যে আমাদের লিস্ট এখানেই শেষ হয়ে গেছে। কখনও কখনও হেডেই টার্মিনাল মান থাকে এর মানে এই লিস্ট ফাঁকা। আমরা টার্মিনাল মান হিসাবে সাধারণত -1 বা 0 এরকম মান ব্যবহার করে থাকি। খেয়াল রাখতে হবে যে এই মান অন্য কোনো মানে যেন না বোাবায়। এখন আমরা মূলত ৩ ধরনের কাজ করে থাকি একটি লিঙ্কড লিস্টের উপর- ইনসার্ট (Insert), ডিলিট (Delete) এবং সার্চ (Search). যদি এই তিনটি কাজ কীভাবে করা যায় বুঝতে পার তাহলে লিঙ্কড লিস্টের মাধ্যমে তোমরা যেকোনো কাজ করতে পারবা।

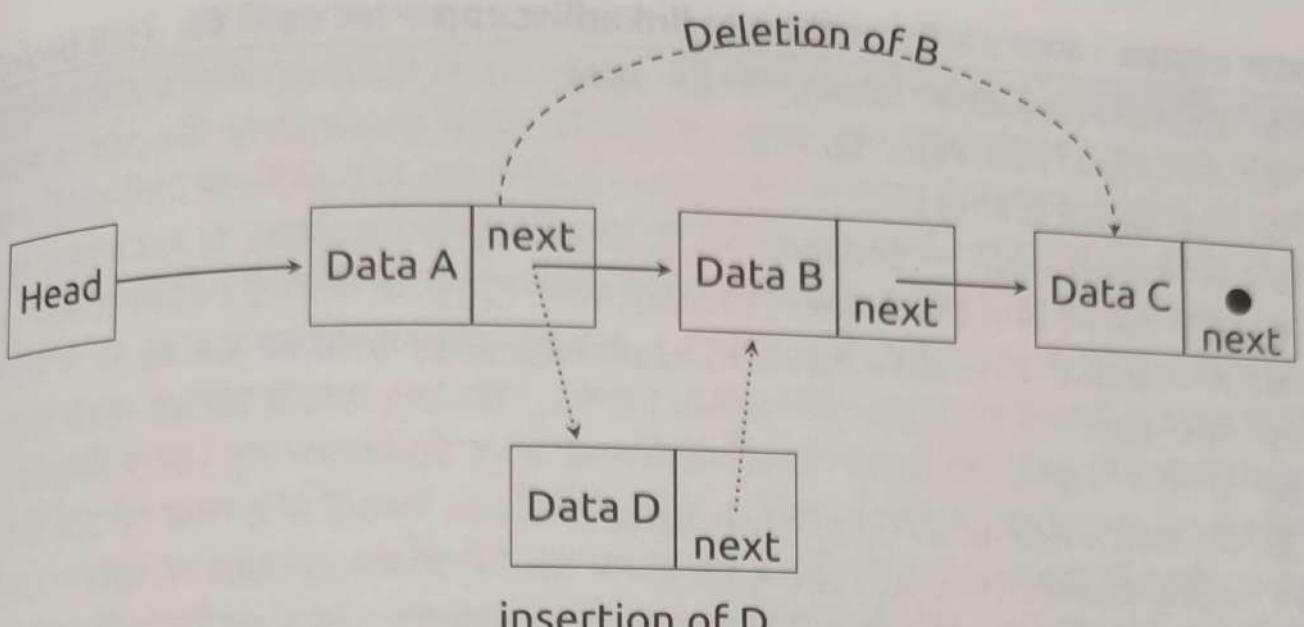
সার্চ (Search)

আমরা লিস্টের প্রথম থেকে শুরু করব। যতক্ষণ না টার্মিনাল মান পাচ্ছি ততক্ষণ পরবর্তী নোডের পয়েন্টার ধরে ধরে একে একে প্রতিটি নোডের ডেটা তে আমারা যেই মান খুঁজছি সেটি আছে কিনা তা দেখব।

ইনসার্ট (Insert)

লিঙ্কড লিস্টে ইনসার্ট করার জন্য আমরা একটি নতুন নোড নিব। এর ডেটা অংশে আমরা ডেটা রাখব আর next অংশে বর্তমানে হেড যাকে নির্দেশ করে আছে তাকে নির্দেশ করব। সেই সঙ্গে আমাদের হেডকে এই নতুন নোডে নির্দেশ করব। তাহলে আমরা আমাদের নতুন নোডকে লিঙ্কড লিস্টের শুরুতে ইনসার্ট করিয়ে দিতে পারছি।

যদি লিস্ট আগে থেকে ফাঁকা না থাকে এবং কোনো একটি নোড (যেমন চিত্র ৫.৩ এ A ও B এর মধ্যে D) এর পরে বসাতে চাই, তাহলে যা করতে হবে তা হলো, D এর next নির্দেশ করবে A এর next কে এবং A এর next নির্দেশ করবে D কে। তুমি যদি শুরুতেই A এর next কে D তে নির্দেশ করিয়ে ফেল তাহলে কিন্তু হবে না। কারণ A কাকে নির্দেশ করছিল তা কিন্তু তুমি হারিয়ে ফেলেছ। সেজন্য প্রথমে D এর next কে নির্দেশ করাতে হবে, এর পর A এর next কে পরিবর্তন করতে হবে।



নকশা ৫.৩: লিঙ্কড লিস্ট

ডিলিট (Delete)

যদি আমাদের হেডের পরের নোডকেই ডিলিট করতে হয় তাহলে হেড ওই নোডের next যাকে নির্দেশ করে আছে তাকে নির্দেশ করিয়ে দিলেই হবে। আর যদি অন্য কোনো নোডকে ডিলিট করতে হয় (যেমন চিত্র ৫.৩ এর B) তাহলে আমরা A এর next কে B এর next এ নির্দেশ করিয়ে দেব।

এখন কথা হলো এই জিনিস কোড করবা কীভাবে। আমরা ইচ্ছা করলে স্ট্রাকচার দিয়ে করতে পারি বা data ও next এর জন্য দুটি আলাদা অ্যারে রেখেও করতে পারি। আমরা কোড ৫.১ এ একটি গ্রাফের অ্যাডজাসেন্সি লিস্টের কোড দিলাম। যদি x আর y এর মধ্যে edge থাকে তাহলে আমরা $insert(x, y)$ কল করব, তাহলে y কে x এর অ্যাডজাসেন্সি লিস্টের শুরুতে ইনসার্ট করা হবে। এজন্য প্রথমে একটি নতুন নোড নেওয়া হচ্ছে (id). এর data তে y রাখা হচ্ছে আর এর next এ বর্তমান হেডের মান রাখা হচ্ছে। সেই সঙ্গে হেডে বর্তমান নোডকে নির্দেশ করা হচ্ছে। ডিলিট করার কোডে লিঙ্কড লিস্টের প্রথম নোডকে আমরা ডিলিট করছি। সার্চের কোডে আমরা x এর লিস্টে y কে কীভাবে খুঁজতে হবে সেটা দেখানো হলো। মূলত এই তিনটি জিনিসই প্রয়োজন হয়। আশা করি অন্য কোনো ফাংশন দরকার হলে তোমরা নিজেরা লিখে নিতে পারবে। যেমন কোনো একটি নোড খুঁজে ডিলিট বা কোনো নোডের পরে ইনসার্ট - এসব করার জন্য সার্চ ফাংশনকে পরিবর্তন করলেই হবে। তবে একটি জিনিস মনে রাখতে হবে যে একদম শুরুতে হেড অ্যারের প্রতিটি জিনিস যেন - 1 (বা অন্য কোনো টার্মিনাল মান) দ্বারা ইনিশিয়ালাইজেশন (initialization) করা থাকে।

কোড ৫.১: linkedlist.cpp

```

1 // total node = 10000, 0 to 9999.
2 // Initialize them to -1
3 int head[10000];
4 // total edge = 100000
5 int data[100000], next[100000];
6 // Global variable has initial value of 0.
7 int id;
8
9 // add node y in the list of x
10 void insert(int x, int y)
11 {
12     data[id] = y;
13     next[id] = head[x];
14     head[x] = id;
15     id++;
16 }
17
18 //erase first node from head of x
19 void erase(int x)
20 {
21     // if you are not sure if the linked
22     // list is empty, check head[x] == -1?
23     head[x] = next[head[x]];
24 }
25
26 //search node y in list of x
27 int search(int x, int y)
28 {
29     for(int p = head[x]; p != -1; p = next[p])
30         if(data[p] == y)
31             return 1; //found
32     return 0; //not found
33 }

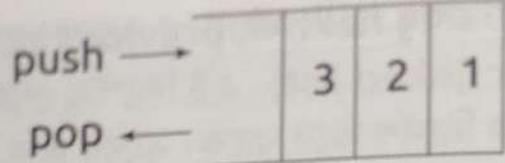
```

লিঙ্কড লিস্টের কিছু ভ্যারিয়েশন (variation) আছে। যেমন ডাবলি লিঙ্কড লিস্ট (doubly

linked list). এই লিঙ্কড লিস্টের শুধু next না, previous পয়েন্টারও থাকে। এছাড়াও আছে, সার্কুলার লিঙ্কড লিস্ট (circular linked list). এই লিস্টের শেষ next এ কোনো টার্মিনাল মান থাকে না বরং এটি আবার শুরুকে নির্দেশ করে থাকে। সুখের কথা হচ্ছে আমরা এখন আর আলাদা করে লিঙ্কড লিস্ট বানাই না, STL এর ভেস্টের (vector) কে এই কাজে ব্যবহার করে থাকি, তবে এটা মনে রাখতে হবে যে ভেস্টের মধ্যখানে ইনসার্ট বা ডিলিট আমাদের লিঙ্কড লিস্টের মতো $O(1)$ না। তবে ডায়নামিক মেমোরী অ্যালোকেশনের কাজ ভেস্টের হয়ে থাকে। মানে ধর তুমি আগে থেকে জানো না যে তোমার অ্যারের আকার কত হবে, সেক্ষেত্রে তুমি ভেস্টের পুশ ব্যাক (push back) করতে থাকो তোমার যতবার করা লাগে, কোনো সমস্যা নেই। শুধু মাঝে ইনসার্ট বা মাঝে ডিলিট না করলেই হলো। খুব কম সময়ই মাঝে ইনসার্ট বা ডিলিট করতে হয়। কিছু দিন আগে আমার একটি লিস্টের মধ্যে নোড ইনসার্ট এবং ডিলিট করার প্রয়োজন পরেছিল। আমি তখন `STL` এর লিস্ট (list) ব্যবহার করেছিলাম। এই লিস্ট নিয়ে আমার খুব একটি ভালো ধারণা নেই কিন্তু আমার কাজ ঠিক মতোই হয়েছিল। সুতরাং তোমাদের যদি কখনও লিঙ্কড লিস্টের মধ্যে ইনসার্ট বা ডিলিটের প্রয়োজন হয় তোমরা `STL` এর লিস্ট (list) ব্যবহার করার চিন্তা করতে পার। তবে লিঙ্কড লিস্টে কিন্তু বিক্ষিপ্তভাবে অ্যারেস (random access) করতে পারবে না। মানে তুমি চাইলে আমি লিস্টের 5 তম স্থানের নোডে যেতে চাই। না যেতে পারবে না। যেতে হলে হেড হতে একটি একটি নোড পার করে যেতে হবে। Random access বা বিক্ষিপ্তভাবে অ্যারেস করার মানে হলো $O(1)$ এ তুমি যেকোনো অবস্থানের নোডে যেতে পারবে। বাইনারি সার্চের ক্ষেত্রে অবশ্যই বিক্ষিপ্তভাবে অ্যারেস করার সুযোগ থাকতে হবে। নাহলে তুমি কীভাবে ফট করে মধ্যে গিয়ে দেখতে পারবে ওখানে কত আছে?

৫.২ স্ট্যাক (Stack)

আমি ঠিক জানি না স্ট্যাক (Stack) কে বাংলায় কী বলে, তবে হয়তো একে স্তুপ বলা যায়। আমরা বলে থাকি যে কাগজের স্তুপ জমে গেছে বা থালার স্তুপ জমে গেছে। এখানে স্তুপ আসলে কেমন জিনিস? আমরা যখন একটির পর একটি থালা রাখি তখন কীভাবে রাখি? নতুন যা থালা আসে তা সব থালার উপরে রাখি, আর যদি একটি থালা নিই তাহলে উপর থেকে নেই। এটিই স্ট্যাক। স্ট্যাক এ কোনো জিনিস প্রবেশ করানোকে পুশ (push) এবং কোনো জিনিস তুলে নেওয়াকে পপ (pop) বলে। স্ট্যাককে আমরা LIFO বা লাস্ট-ইন ফাস্ট-আউট (Last In First Out) বলি কারণ সবার পরে যে ঢুকেছে সেই আগে বের হবে। একে তোমরা চিত্র ৫.৪ এর মতো করে কল্পনা করতে পারো। এখানে সবার প্রথমে ঢুকেছে 1, এর পর 2, এবং সবশেষে 3. কিন্তু বের হবার সময় সবার আগে বের হবে 3, এর পর 2, এর পর 1. আমরা চাইলে একটি অ্যারে এবং হেডকে নির্দেশ করার জন্য একটি ভ্যারিয়েবল রেখে খুব সহজেই স্ট্যাক বানিয়ে ফেলতে পারি। তবে এতে সমস্যা হলো আমাদের আগে থেকেই অনেক বড় অ্যারে ডিক্লেয়ার করে রাখতে হবে। অন্য একটি উপায় হলো লিঙ্কড লিস্টের মাধ্যমে করা। তবে এখন আমাদের জন্য `STL` এ `stack` দেওয়াই আছে। স্ট্যাক এর বিভিন্ন ইম্প্লিমেন্টেশনের কোড ৫.২ এ দেয়া হলো।



নকশা ৫.৪: স্ট্যাক (Stack)

কোড ৫.২: 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  //initialization
9  head = -1, sz = 0;
10 //push
11 node[sz] = data;
12 next[sz] = head;
13 head = sz++;
14 //pop & return
15 ret = node[head];
16 head = next[head];
17 return ret;
18 //check
19 if(head != -1)
20
21 /* STL */
22 #include<stack>
23 using namespace std;
24
25 //declare, replace int by the type you want
26 stack<int> s;
27 //initialization after using
28 while(!s.empty()) s.pop();
29 s.push(5); //push

```

```

30 s.top();      //return top element, but doesn't pop
31 s.pop();      //pop but doesn't return
32 s.size();     //size of the stack
33 s.empty();    //returns true if empty

```

এখন এত সাধারণ জিনিস ব্যবহার করে কীভাবে কঠিন সমস্যা সমাধান করা সম্ভব? একটি উদাহরণ দেওয়া যাক।

৫.২.১ ০ – ১ ম্যাট্রিক্সে সব ১ ওয়ালা সবচেয়ে বড় আয়তক্ষেত্র

সমস্যা: ধরা যাক একটি $n \times m$ ডাইমেনশনের ০ – ১ ম্যাট্রিক্স দেয়া আছে। আমাদের এমন সব আয়তক্ষেত্র বের করতে হবে যার সবগুলো সংখ্যা ১। এদের মধ্যে সবচেয়ে বড় ক্ষেত্রফলটি প্রিন্ট করতে হবে।

সমাধান: প্রথমে আমরা সারির উপর লুপ চালাব। আমরা ধরে নেব যে এই সারিই হলো আমাদের আয়তক্ষেত্রের নিচের বাহু। সুতরাং আমরা এখন এমন একটি আয়তক্ষেত্র বের করতে চাই যার নিচের বাহু এই সারিতে থাকবে, যার ভেতরে সবগুলো সংখ্যা ১ এবং এর ক্ষেত্রফল সবচেয়ে বেশি। আমরা যা করব তা হলো প্রতিটি কলামে গিয়ে ওই সারি থেকে তার উপর দিকে যতগুলো পর পর ১ আছে তার সংখ্যা গুনে বের করব। এর ফলে আমরা আসলে m টি সংখ্যা পাব (প্রতিটি কলামের জন্য একটি সংখ্যা)। যেমন চিত্র ৫.৫ এ আমরা একটি ০ – ১ ম্যাট্রিক্সের চতুর্থ সারিকে ভিত্তি ধরেছি। এর ফলে ওই সারি হতে উপর দিকে কয়টি ১ আছে তার সংখ্যা নিয়ে ডান দিকের অ্যারেটি বানিয়েছি। এখন আমাদের এই অ্যারের এমন একটি সাব-রেঞ্জ (sub-range) বের করতে হবে যার দৈর্ঘ্যের সঙ্গে এই সাব-রেঞ্জে থাকা সংখ্যাগুলোর মধ্যে সবচেয়ে কমটি গুণ করলে যেই গুণফল হয় তা যেন সর্বোচ্চ হয়।

1	1	0	0	1	1	1	0
0	1	0	1	0	0	0	1
1	1	0	1	1	1	1	0
1	1	1	1	1	0	1	1
1	1	1	1	0	0	1	0

2	4	1	3	2	0	2	1
---	---	---	---	---	---	---	---

নকশা ৫.৫: স্ট্যাক (Stack)

আমরা যা করব তাহলো প্রথমে একটি ফাঁকা স্ট্যাক নিব। এর পর অ্যারের প্রথম স্থান থেকে শুরু করে একটি করে সংখ্যা নেব। যদি আমাদের স্ট্যাক ফাঁকা থাকে তাহলে ওই সংখ্যা এবং তার অবস্থান বা ইনডেক্স (index), স্ট্যাকে পুশ করব। আর যদি ইতোমধ্যে কিছু সংখ্যা স্ট্যাকে থাকে

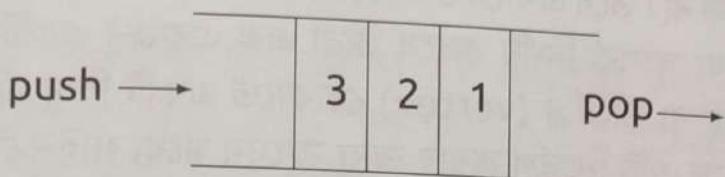
তাহলে আমরা স্ট্যাকের উপরের সংখ্যার সঙ্গে তুলনা করে দেখব যে এই সংখ্যা আমাদের বর্তমান সংখ্যার থেকে ছোট না বড়। যদি আমাদের নতুন সংখ্যা বড় হয় তাহলে আগের মতোই এই সংখ্যা এবং স্থান (index) স্ট্যাকে রাখব। আর যদি ছোট বা সমান হয়, তাহলে স্ট্যাক থেকে একটি একটি করে সংখ্যা তুলতে থাকব যতক্ষণ না স্ট্যাকের হেডের সংখ্যাটি আমাদের সংখ্যা থেকে ছোট হয়। প্রতিবার আমাদের যা করতে হবে তাহলো, স্ট্যাক থেকে পাওয়া ইনডেক্স ও আমাদের বর্তমান ইনডেক্স এর মাঝের দূরত্বকে ওই সংখ্যা দিয়ে গুণ করতে হবে। এখন এই গুণফলগুলোর মধ্যে সবচেয়ে বড়টিই আমাদের উত্তর। যখন আমরা আমাদের বর্তমান সংখ্যার থেকে ছোট একটি সংখ্যা পাব তখন আমরা সর্বশেষ যেই ইনডেক্স স্ট্যাক থেকে পপ (pop) করেছিলাম সেই ইনডেক্স ও আমাদের বর্তমান সংখ্যা পুশ করব। যেমন চিত্র ৫.৫ এর 1st অ্যারেতে প্রথমে আছে 2, একে আমরা একটি স্ট্যাকে পুশ করি। এর পর 4 কে পুশ করি, কারণ এটি স্ট্যাকের হেডে থাকা 2 এর থেকে বড়। এখন এর পরের সংখ্যা 1 যখন নিবে তখন দেখো স্ট্যাকের হেডে আছে 4 এবং আমরা আর এই 4 উচ্চতার আয়তক্ষেত্রকে প্রস্ত্রে বাড়াতে পারছি না। সুতরাং 4 উচ্চতার আয়তক্ষেত্র আমরা মাত্র 1 ঘর ব্যাপী পেয়েছি। তাই এর ক্ষেত্রফল 4. এবার 4 কে তুলে ফেলে দেই স্ট্যাক থেকে। এবার দেখি উপরের উপাদান হলো 2. এখন 2 এর ইনডেক্স থেকে এই পর্যন্ত মোট 2 টি কলাম। অর্থাৎ 2 উচ্চতার আয়তক্ষেত্র মোট $2 \times 2 = 4$ ঘর জুড়ে আছে। এবার এটিও তুলে ফেলে দাও। যেহেতু এখন স্ট্যাক ফাঁকা হয়ে গেছে (বা যদি উপরের উপাদান আমাদের বর্তমান সংখ্যা 1 এর থেকে ছোট হতো) তাই আমরা আমাদের বর্তমান সংখ্যাকে স্ট্যাকে পুশ করি। এভাবে চলতে থাকবে।

এখন কেন এই জিনিস কাজ করবে? একটু চিন্তা করলে দেখবে যে, আমরা আসলে স্ট্যাকে যা রাখছি তা হলো, আমাদের বর্তমান স্থান থেকে কতদূর পর্যন্ত কত উচ্চতার 1ওয়ালা আয়তক্ষেত্র পাওয়া যায় তা। আমরা যখন স্ট্যাক থেকে একটি উচ্চতা তুলে ফেলছি তার কারণ হলো আমাদের নতুন যেই উচ্চতা সেটি তুলে ফেলা উচ্চতা হতে ছোট, সুতরাং আমাদের পক্ষে ওই বড় উচ্চতার সমান উঁচু আয়তক্ষেত্র আসলে আর বানানো সম্ভব না। তাই ওই উপাদানকে তুলে ফেলা হচ্ছে। একটি উদাহরণ দেওয়া যাক, মনে কর স্ট্যাকের সংখ্যাগুলো হলো 5, 8, 9. এখন তোমার কাছে এল 6. খেয়াল কর তুমি কিন্তু 8 উচ্চতার বা 9 উচ্চতার আয়তক্ষেত্র কিন্তু বাড়াতে পারবে না। যেহেতু আর বাড়াতে পারবে না তাহলে 8 দিয়ে কত বড় আয়তক্ষেত্র বানাতে পারবে আর 9 দিয়েই বা কত বড় আয়তক্ষেত্র বানাতে পারবে? এটি আসলে 8 আর 9 এর সঙ্গে থাকা ইনডেক্স এবং বর্তমান ইনডেক্সের এর পার্থক্যকে ওই সংখ্যা দিয়ে গুণ করলেই পাবে। এর পর তোমার কাজ হলো 6, আর 8 এর সঙ্গে থাকা ইনডেক্স পুশ করা। কারণ যেই জায়গা থেকে আগে 8 এর আয়তক্ষেত্র ছিল এখন সেই জায়গা থেকে 6 এর আয়তক্ষেত্র শুরু হবে।

৫.৩ কিউ (Queue)

আমরা কিন্তু কিউ (queue) শব্দটি আমাদের দৈনন্দিন কথা বার্তায় প্রায়ই ব্যবহার করে থাকি। যেমন বাসের কিউ (queue), ব্যাংকে বিল জমা দেওয়ার কিউ। এখানে আসলে কী হয়? নতুন যে আসবে সে কিউয়ের একদম শেষে দাঁড়াবে আর যদি আমরা একজনকে বের করতে চাই তাহলে

শুরু থেকে বের করব। এ জন্য একে FIFO বা ফাস্ট-ইন ফাস্ট-আউট (First In First Out) বলে অর্থাৎ যে সবার প্রথমে ঢুকেছিল সেই সবার আগে বের হবে। যেমন বাসের কিউতে যে সবার আগে এসেছিল সেই সবার সামনে থাকবে এবং বাস এলে প্রথমেই সে বাসে ঢুকবে আবার নতুন কেউ এলে সে সবার শেষেই দাঁড়াবে কাউকে ডিঙিয়ে সামনে যাবে না। চিত্র ৫.৬ এর মতো করে একে কল্পনা করতে পারো। এখানে প্রথমে 1 ঢুকেছে (পুশ), এর পর 2, এর পর 3. আবার যখন বের হবে (pop) তখন প্রথমে 1 বের হবে, এর পর 2, এর পর 3. স্ট্যাকের মতো কিউয়ের ইমপ্লিমেন্টেশনের জন্য আমরা অ্যারে ব্যবহার করতে পারি, বা লিঙ্কড লিস্টও ব্যবহার করতে পারি। তবে STL এ queue নামে ডেটা স্ট্রাকচার দেওয়াই আছে। কিউয়ের বিভিন্ন ইমপ্লিমেন্টেশনের কোড ৫.৩ এ দেওয়া হলো।



নকশা ৫.৬: কিউ (Queue)

কোড ৫.৩: queue.cpp

```

1  /* array implementation */
2  head = tail = 0; //initialization
3  q[tail++] = data; //push
4  return s[head++]; //pop and return
5  //check whether there is something in stack
6  if(head == tail)
7
8  /* STL */
9  #include<queue>
10 using namespace std;
11
12 //declare, replace int by the type you want
13 queue<int> Q;
14 //initialization after using
15 while(!Q.empty()) Q.pop();
16 Q.push(5); //push
17 Q.front(); //return front element, but doesnt pop
18 Q.pop(); //pop but doesnt return
19 Q.size(); //size of the queue

```

৫.৮ গ্রাফ (graph) এর উপস্থাপন

গ্রাফ (graph) হলো সহজ অর্থে সম্পর্ক। অনেক ধরনের জিনিসের মধ্যে সম্পর্ক বোঝাতে আমরা গ্রাফ ব্যবহার করে থাকি। যেমন কিছু আগেই আমরা দেখে এসেছি যে ফেসবুকে কতগুলো মানুষের মধ্যে বন্ধুত্বের সম্পর্ক বোঝানোর জন্য আমরা গ্রাফ ব্যবহার করতে পারি। এই গ্রাফ কিন্তু আমাদের লেখচিত্রের গ্রাফ না। তবে এখানেও আঁকার জিনিস আছে তবে ছক কাগজের দরকার নেই! তুমি যেসব মানুষের মধ্যে সম্পর্ক নির্ণয় করবে তারা এক একজন একটি করে নোড বা ভার্টেক্স (vertex)। আমরা নোড বা ভার্টেক্স (vertex) বোঝাতে একটি বিন্দু এঁকে থাকি। এখন দুজন মানুষের মধ্যে বন্ধুত্ব আছে এটি নির্দেশ করার জন্য তাদের মধ্যে লাইন টেনে থাকি একে বাহু বা edge বলা হয়। তোমরা লক্ষ্য করলে দেখবে একটি মানচিত্রে বিভিন্ন শহরের মধ্যে রাস্তা, রেললাইন এসব জিনিস দাগ কেটে দেখানো থাকে। এসব ক্ষেত্রে আমরা শহরগুলোকে ভার্টেক্স ও রাস্তাগুলোকে বাহু বা edge হিসেবে কল্পনা করতে পারি আর তাহলে আমাদের এই বিশাল মানচিত্র একটি গ্রাফ হয়ে যায় যা বিভিন্ন শহরের মধ্যে রাস্তার সম্পর্ক দেখায়।

এখন গ্রাফের সমস্যা সমাধানের সময় আমাদের এই গ্রাফকে মেমোরীতে রাখতে হবে। আমরা তো আর কম্পিউটারে ছবি এঁকে রাখতে পারি না, আমাদেরকে এই ভার্টেক্সগুলোর একটি করে ত্রুটি সাংখ্যমান দিতে হয় আর কততম ভার্টেক্সের সঙ্গে কততম ভার্টেক্সের সম্পর্ক আছে তা অ্যাডজাসেন্সি লিস্ট বা অ্যাডজাসেন্সি ম্যাট্রিক্সের সাহায্যে রাখতে হয়। আমরা কিন্তু ইতোমধ্যেই এই দুটির নাম আর কীভাবে করতে হয় তা জেনে ফেলেছি!

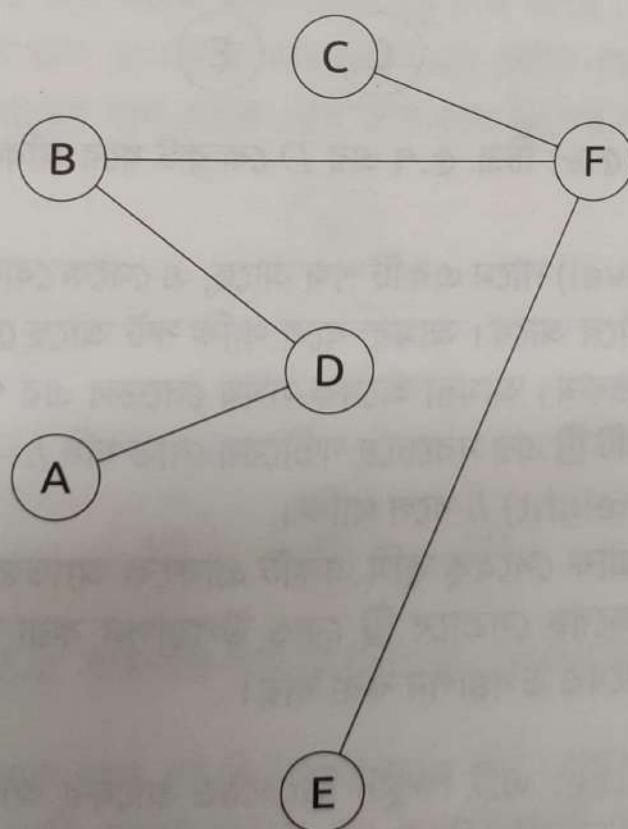
আমাদের এই ফেসবুকের উদাহরণে বন্ধু কিন্তু দ্বিমুখী বা বাইডিরেকশনাল (bidirectional) অর্থাৎ A যদি B এর বন্ধু হয় তাহলে B ও A এর বন্ধু হবে। কিন্তু অনেক সময় এই সম্পর্ক বাইডিরেকশনাল (bidirectional) না হয়ে একমুখী বা ডিরেকশনাল (directional) হয়ে থাকে। যেমন ফেসবুকের ফলোয়ার। A যদি B কে ফলো (follow) করে এর মানে এই না যে B ও A কে ফলো (follow) করছে। অর্থাৎ এখানে সম্পর্ক একতরফা। আমরা আগের গ্রাফকে বলে থাকি আনডিরেক্টেড গ্রাফ (undirected graph) আর ফলোয়ার (follower) এর গ্রাফ হলো ডিরেক্টেড গ্রাফ (directed graph). প্রথম ক্ষেত্রে A ও B এর মধ্যে যদি আনডিরেক্টেড বাহু (undirected edge) থাকে তাহলে A এর লিস্টে B কে এবং B এর লিস্টে A কে রাখতে হয়। আর যদি ডিরেক্টেড বাহু (directed edge) হয় তাহলে শুধু A এর লিস্টে B কে রাখলেই হবে যদি A এর থেকে B এর দিকে বাহু (edge) থাকে।

কখনও কখনও বাহুগুলোর সঙ্গে একটি সংখ্যা থাকে। যেমন আমাদের মানচিত্রের গ্রাফে হয়তো দুটি শহরের মধ্যে যেই রাস্তা দেখানো আছে তার পাশে সেই রাস্তার দৈর্ঘ্য দেয়া আছে। এটা হলো ওই বাহুর ওজন (weight) বা মূল্য (cost)। এধরনের গ্রাফকে বলা হয় ওয়েইটেড গ্রাফ (weighted graph) আর আগের গ্রাফকে বলা হয় আনওয়েইটেড গ্রাফ (unweighted graph)। আগে যেমন কে কার সঙ্গে সংযুক্ত এই তথ্য অ্যাডজাসেন্সি ম্যাট্রিক্স বা অ্যাডজাসেন্সি

লিস্টের মাধ্যমে উপস্থাপন করেছিলাম, এখন ঠিক তেমনি সেই তথ্যের সাথে সাথে তার ওজন (weight) টাও সংরক্ষণ করতে হবে। যদি আমরা অ্যাডজাসেন্সি ম্যাট্রিক্স ব্যবহার করি তাহলে বাহুর ওজনের জন্য আরেকটি ম্যাট্রিক্স আমরা রাখতে পারি বা অ্যাডজাসেন্সি ম্যাট্রিক্সে $0 - 1$ না রেখে $0 - weight$ রাখতে পারি। যদি আমরা অ্যাডজাসেন্সি লিস্ট ব্যবহার করি তাহলে কে কার প্রতিবেশী (neighbor) এই তথ্যের সাথে সাথে ওজনও রাখতে হবে (পারত পক্ষে আমি struct ব্যবহার করি)।

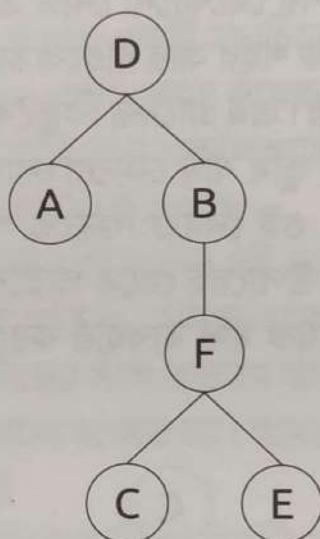
৫.৫ ট্রি (Tree)

ট্রি (Tree) একটি বিশেষ ধরনের গ্রাফ যেখানে n টি ভাট্টেক্সের জন্য $n - 1$ টি বাহু থাকে এবং পুরো গ্রাফ কানেক্টেড (connected) থাকে। কানেক্টেড (Connected) থাকার অর্থ হলো ওই গ্রাফের যেকোনো এক নোড থেকে অপর যেকোনো নোড এ তুমি এক বা একাধিক বাহু ব্যবহার করে যেতে পারবে। যদি আমরা নোড কে শহর আর বাহুকে রাস্তা মনে করি তাহলে বলা যায়, প্রতিটি শহর থেকেই অন্য সব শহরে যাওয়া যায়। এই গ্রাফের কিছু বৈশিষ্ট্য আছে। যেমন এই গ্রাফে কোনো চক্র বা সাইকেল (cycle) নেই, অর্থাৎ তুমি যদি কোনো নোড থেকে শুরু কর তাহলে কোনো বাহু দুবার ব্যবহার না করে কোনো মতেই ওই নোডে ফিরতে পারবে না। তুমি কোনো একটি নোড থেকে অপর নোডে সবসময় একটিমাত্র উপায়েই যেতে পারবে মানে তোমার যাওয়ার রাস্তা একটিই থাকবে (অবশ্যই তুমি এক রাস্তা একাধিক বার ব্যবহার করবে না)। যেমন চিত্র ৫.৭ এ একটি ৬ নোডের ট্রি দেখানো হলো।



নকশা ৫.৭: আনরুটেড ট্রি (Unrooted tree)

অনেক সময় ট্রিতে একটি বিশেষ নোড দেওয়া থাকে যাকে বলা হয় মূল বা রুট (root). এক্ষেত্রে ট্রি এর বাহ্যগুলোকে ডাইরেক্টেড (directed) ভাবে কল্পনা করা যায়। বাহ্য দিক হবে রুট থেকে কোনো নোডে যেতে হলে ওই বাহ্য দিয়ে তুমি যেদিকে যাবে সেদিক। তোমরা চাইলে রুট কে ধরে ওই ট্রি কে ঝুলিয়ে দিতে পার। তাহলে উপর থেকে নিচের দিকে হবে ওই বাহ্যগুলো। কোনো বাহ্য উপরের নোডকে অভিভাবক বা প্যারেন্ট (parent) বলা হয়, আর নিচের নোডকে ওই প্যারেন্টের সন্তান বা চাইল্ড (child) বলা হয়। আমরা সাধারণত বাংলা পরিভাষার পরিবর্তে ইংরেজী নামগুলোই ব্যবহার করবো। কোনো নোডের প্যারেন্টের অন্যান্য চাইল্ডকে এই নোডের সিবলিং (sibling) বলে। যদি কোনো নোড থেকে তুমি উপরে রুটের দিকে যেতে থাক তাহলে পথে যেসব নোড পাবে তাদের পূর্বসূরী বা অ্যানসেস্টর (ancestor) বলে। কোনো নোড থেকে উপরে না উঠে শুধু নিচে নামতে থাকলে যেসব নোড পাওয়া যায় তাদের উত্তরসূরি বা ডিসেন্ডেন্ট (descendant) বলে। চিত্র ?? তে একটি রুটেড ট্রি (rooted tree) দেখানো হলো। এটি আসলে চিত্র ৫.৭ এর D নোডকে রুট ধরে আঁকা হয়েছে।



নকশা ৫.৮: চিত্র ৫.৭ এর D কে রুট ধরে আঁকা ট্রি

ট্রি এর ক্ষেত্রে লেভেল (level) নামে একটি শব্দ আছে, এ থেকে বোঝা যায় কোনো একটি নোড আমাদের রুট থেকে কত গভীরে আছে। আমরা বলে থাকি রুট আছে লেভেল ০ তে, এর এক ধাপ নিচেরগুলো লেভেল ১ এ, এরকম। আমরা অনেক সময় লেভেল এর পরিবর্তে গভীরতা বা ডেপ্থ (depth) ও বলে থাকি। একটি ট্রি এর সবচেয়ে গভীরের নোড যদি $h - 1$ গভীরতায় থাকে তাহলে আমরা সেই ট্রি এর উচ্চতা (height) h বলে থাকি।

যেহেতু ট্রি এক ধরনের গ্রাফ সেহেতু তুমি একটি গ্রাফকে অ্যাডজাসেন্সি ম্যাট্রিক্স বা লিস্টের মাধ্যমে যেভাবে উপস্থাপন করেছ সেভাবে ট্রি কেও উপস্থাপন করা যায়। কিন্তু ট্রি এর আলাদা বৈশিষ্ট্যের জন্য একে অন্যভাবেও উপস্থাপন করা যায়।

- চাইল্ড লিস্ট (Child List): এটি কিছুটা ডাইরেক্টেড গ্রাফের অ্যাডজাসেন্সি লিস্টের মতো। আমরা প্রতিটি নোডের চাইল্ডের লিস্ট রাখব। এ ক্ষেত্রে আমরা চাইলে যেকোনো নোড থেকে শুরু করে শুধু নিচের দিকে যেতে পারি। এই উপস্থাপনের ক্ষেত্রে বেশির ভাগ সময় আমরা রুট

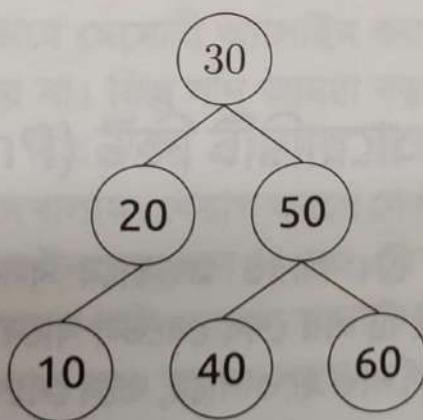
থেকে শুরু করে নিচের দিকে যেতে থাকি। একে আমরা টপডাউন উপস্থাপন (top down representation) বলতে পারি।

প্যারেন্ট লিঙ্ক (Parent Link): এক্ষেত্রে আমরা প্রতিটি নোডের প্যারেন্ট রেখে থাকি। এই উপস্থাপনের ক্ষেত্রে আমরা উপর থেকে নিচে যেতে পারি না। কিন্তু কোনো একটি নোড থেকে উঠতে পারি। একে আমরা বটমআপ উপস্থাপন (bottom up representation) বলতে পারি।

অনেক সময় আমাদের চাইল্ড লিস্ট ও প্যারেন্ট লিঙ্ক দুটিই একই সঙ্গে দরকার হয়ে থাকে। যদি প্রতিটি নোডের খুব জোর দুইটি চাইল্ড থাকে তাহলে সেই ট্রি কে বাইনারি ট্রি (binary tree) বলা হয়। আমরা ছবি আঁকার সময় যে চাইল্ডকে বাম দিকে রাখি তাকে লেফট চাইল্ড (left child) ও অপরটিকে রাইট চাইল্ড (right child) বলি। এছাড়াও ট্রি সম্পর্কিত আরও অনেক সংজ্ঞা আছে আমরা ধীরে ধীরে সেসব জানব।

৫.৬ বাইনারি সার্চ ট্রি (Binary Search Tree - BST)

এই বাইনারি ট্রি এর প্রতিটি নোডে একটি করে মান থাকে। এখন মানগুলো এমনভাবে থাকে যেন এর লেফট সাবট্রি (left subtree) এর সকল মান 1 এই নোডে থাকা মান থেকে ছোট হয় আর রাইট সাবট্রি (right subtree) এর সব মানের থেকে বড় হয়। যেমন চিত্র ৫.৯ এ একটি বাইনারি সার্চ ট্রি দেখানো হল। খেয়াল কর রঞ্জ এ ৩০ আছে, আর এর বামে এর থেকে ছোট মান আছে (১০, ২০)। আবার এর ডান দিকে এর থেকে বড় মান আছে (৪০, ৫০, ৬০)। একইভাবে ৫০ এরও বামে এর থেকে ছোট মান ৪০ আছে আর ডানে এর থেকে বড় ৬০। অর্থাৎ কোনো নোডের বাম দিকের সব মান ওই নোডের মান থেকে ছোট আর ডান দিকের সব মান বড়।



নকশা ৫.৯: বাইনারি সার্চ ট্রি (Binary Search Tree)

আমরা লিঙ্কড লিস্টের মতো করে এই জিনিস বানাতে পারি। সেক্ষেত্রে প্রতিটি নোডে আমাদের দুটি লিঙ্কের দরকার হবে, একটি লেফট চাইল্ড এর জন্য অপরটি রাইট চাইল্ডের জন্য। অনেক

¹সাবট্রি (subtree) হলো মূল ট্রি এর একটি অংশ যা নিজেও একটি ট্রি।

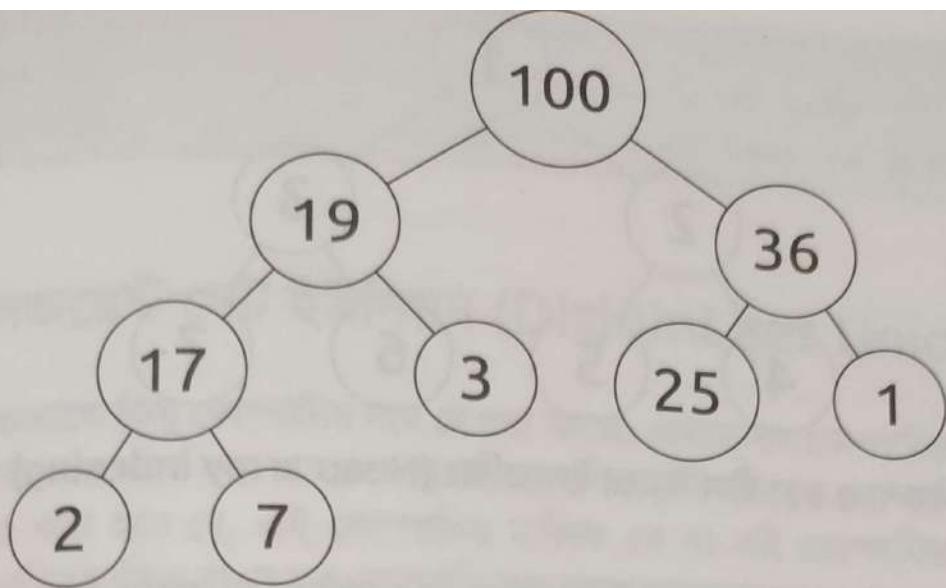
সময় প্যারেন্ট এর জন্যও আলাদা লিঙ্ক রাখা হয়। একাত বাইনার সাচ ট্রি তে কোনো একটি সংখ্যা আছে কিনা তা খুঁজে বের করা বেশ সহজ। তুমি কোনো একটি নোডে গিয়ে দেখবে যে তুমি মেই সংখ্যা খুঁজছ সেটা এখানে থাকা সংখ্যার থেকে ছোট না বড়। যদি সমান হয় তাহলে তো পেয়েই গেলে, আর যদি ছোট হয় তাহলে বাম দিকে যাবে আর বড় হলে ডান দিকে যাবা। এরকম করে তুমি ইনসার্টও করতে পারবে। ডিলিট করা একটু কঠিন। অনেক সময় প্রোগ্রামিং প্রতিযোগিতায় আমরা সত্যিকারভাবে ডিলিট না করে প্রতিটি নোডে একটি করে চিহ্ন বা ফ্ল্যাগ (Flag) রাখি। ডিলিট করলে সেই চিহ্ন বা ফ্ল্যাগ (flag) কে বন্ধ করে দিলেই হয়।

এখন কথা হলো, একটি অ্যারেতে সংখ্যা না রেখে আমরা এরকম ট্রি আকারে সংখ্যা রাখলে দাঢ় কী? খেয়াল কর, একটি সংখ্যা খোঁজার সময় আমরা যখন একটি নোডে থাকা সংখ্যার সঙ্গে আমরা সংখ্যাকে তুলনা করি তখন সেই তুলনার ভিত্তিতে আমরা একদিক বাদ দিয়ে আরেক দিকে মেঝে পারি। এখন এই ভাগাভাগি যদি ঠিক অর্ধেক হয় তাহলে আমরা প্রতিবার অর্ধেক সংখ্যা বাদ দিতে পারি। ঠিক আমাদের শেখা বাইনারি সার্চের মতো। তাহলে আমরা যদি ঠিক অর্ধেক অর্ধেক করে রাখতে পারি তাহলে আমরা $O(\log n)$ এই সার্চ করতে পারব। তাহলে আমাদের বাইনারি সার্চের সঙ্গে এর পার্থক্য কোথায়? খেয়াল কর, বাইনারি সার্চে আমরা কিন্তু কোনো একটি সংখ্যাকে ইনসার্ট বা ডিলিট করতে পারি না। কিন্তু আমরা আমাদের এই BST তে কোনো সংখ্যা ইনসার্ট বা ডিলিট করতে পারি। একটু ভাবলে বুঝবে যে সাধারণভাবে সংখ্যাগুলোকে 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 ব্যালেন্স (balance) করে নিতে হয় যেন ট্রি এর উচ্চতা বেশি বড় না হয়। এরকম ধরনের কিছু ডেটা স্ট্রাকচার আছে যেমন এভিএল ট্রি (AVL tree), রেড ব্ল্যাক ট্রি (Red Black tree), ট্রিপ (Treap), স্প্লে ট্রি (Splay tree) ইত্যাদি। তোমরা চাইলে এসব জিনিস ইন্টারনেটে দেখতে পার। তবে বেশির ভাগ সময় আমরা STL এর ম্যাপ (Map) বা সেট (Set) ব্যবহার করে BST সংক্রান্ত অনেক কাজ করে ফেলতে পারি।

৫.৭ হীপ (Heap) বা প্রায়োরিটি কিউ (Priority Queue)

এটিও এক ধরনের বাইনারি ট্রি। আরও শুধুভাবে বলতে গেলে কমপ্লিট বাইনারি ট্রি (complete binary tree)। এই ট্রি এর শেষ লেভেল বাদে প্রতিটি লেভেলে সর্বোচ্চ সংখ্যক নোড থাকবে। শুধু শেষ লেভেলটি পূর্ণ না ও হতে পারে, তবে সেক্ষেত্রেও বাম থেকে ডান দিকে নোড গুলো সাজানো থাকে। চিত্র ৫.১০ এ তোমাদের জন্য একটি হীপ দেওয়া আছে।

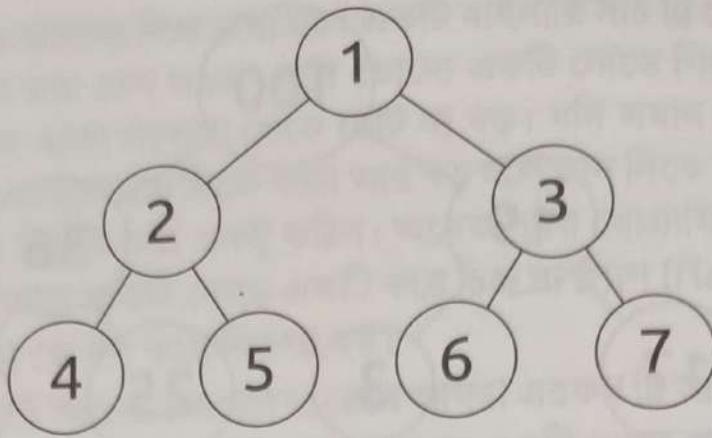
হীপ দুই রকম হতে পারে, ম্যাক্স হীপ (Max Heap), মিন হীপ (Min Heap)। ম্যাক্স হীপের বৈশিষ্ট্য হচ্ছে কোনো নোডে থাকা মান তার যেকোনো ডিসেন্ডেন্ট এর থেকে বড় হবে। অর্থাৎ কুটো এই হীপের সবচেয়ে বড় মান থাকবে, তার লেফট চাইল্ডে থাকবে লেফট সাবট্রি এর মাঝের সবচেয়ে বড় মান। এরকম করে পুরো হীপ বানানো হয়। আশা করি বুঝতেই পারছ মিন হীপ কী রকম হয়। যদি কোনো হীপে n সংখ্যক নোড থাকে তাহলে সেই হীপের উচ্চতা $\log(n)$ হয়। আমরা এই



নকশা ৫.১০: হৈপ (Heap)

ডেটা স্ট্রাকচারটি তখন ব্যবহার করে থাকি যখন আমাদের অনেকগুলো মান একে একে আসতে থাকে এবং আমাদের মাঝে মধ্যে সবচেয়ে বড় মানটি দরকার হয়। চাইলে আমাদের বড় মানটি সরিয়ে ফেলতে হয়। ফলে পরে যখন আবারো সবচেয়ে বড় মানের দরকার হয় তখন এর পরবর্তী বড় মানটি দিতে হয়। যেমন চিত্র ৫.১০ এ আমাদের সবচেয়ে বড় মান চাইলে 100 দিতে হবে, এর পরে আবারো বড় মান চাইলে 36 দিতে হবে এরকম। আবার যদি এই দ্বিতীয় চাওয়ার আগে যদি মনে কর 50 ইনসার্ট হয় তাহলে আমাদের কিন্তু 50 দিতে হবে, 36 না। একটু মনে মনে ভাব তো তোমাদের যদি একটি হৈপ বানাতে বলি কীভাবে কোড করবে? যদি ভেবে থাক লিঙ্কড লিস্টের মতো করে পয়েন্টার রেখে- তাহলে তোমরা ঠিক ভেবেছ। কিন্তু এর থেকেও সহজ উপায় আছে (এবং এর অসুবিধাও আছে!)। তোমরা যদি আগের মতো লেফট চাইল্ড লিঙ্ক ও রাইট চাইল্ড লিঙ্ক রেখে রেখে কর এবং ডায়নামিকভাবে মেমোরী অ্যাসাইন কর তাহলে আগে থেকে আমাদের বড় অ্যারে ডিক্লেয়ার করার দরকার হয় না। কিন্তু যদি আমরা বড় অ্যারে ডিক্লেয়ার করে করতে চাই তাহলে একটি সহজ উপায় আছে। চিত্র ৫.১১ এ আমরা হৈপের জন্য অ্যারে কীভাবে ইনডেক্সিং (indexing) করলে সহজ হয় তা দেখালাম। খেয়াল করলে দেখবে, কোনো একটি নোডের ইনডেক্স যদি i হয় তাহলে এর লেফট চাইল্ডের ইনডেক্স হবে $2i$ এবং এর রাইট চাইল্ডের ইনডেক্স হবে $2i+1$ । তাহলে দেখ সুন্দর করে পর পর লেভেল-বাই-লেভেল আমাদের ইনডেক্সিং হয়ে যাবে। যদি কোনো নোড i থেকে তার প্যারেন্টে যেতে চাও তাও অনেক সহজ। $i/2$ করেই যেতে পারবে। মনে রাখবে এখানে integer division হচ্ছে।

হৈপে ইনসার্ট করা খুব সহজ। যদি হৈপে ইতোমধ্যে n টি সংখ্যা থাকে তাহলে নতুন সংখ্যা তোমরা অ্যারের $n+1$ অবস্থানে বসাও। এর পর তুমি প্যারেন্ট ব্যবহার করে রুট পর্যন্ত যেতে থাকবে, যদি দেখ প্যারেন্ট তোমার থেকে ছোট তাহলে বদলাবদলি (swapping) করবে, এরকম যতক্ষণ না তোমার প্যারেন্ট তোমার থেকে বড় না হয় ততক্ষণ এই কাজ করলেই হবে। যেহেতু আমাদের উচ্চতা $\log(n)$ সুতরাং আমাদের ইনসার্ট করতে সময় লাগবে $O(\log n)$. যদি ম্যান্স হৈপ হতে



নকশা ৫.১১: হীপ অ্যারে ইনডেক্সিং (Heap array indexing)

এর রুট অর্থাৎ সবচেয়ে বড় সংখ্যা ডিলিট করতে চাও তাহলে যা করতে হবে তা হলো অ্যারের শেষ সংখ্যাকে এনে রুটে বসাতে হবে। এর পর দেখতে হবে তোমার লেফট চাইলে বড় না রাইট চাইল। ওদের যেটি বড় তা যদি আবার তোমার থেকেও বড় হয় তাহলে তার সঙ্গে বদলাবদলি কর এবং এভাবে নিচে নামতে থাক। এভাবে $O(\log n)$ এ আমরা সবচেয়ে বড় সংখ্যাকে ডিলিট করতে পারবে। কিন্তু একটি জিনিস খেয়াল রাখবে, এখানে কিন্তু কোনো একটি সংখ্যা খুব দ্রুত খুঁজে পাওয়া সম্ভব না। তোমাকে কোনো সংখ্যা খুঁজতে হলে সবগুলো নোডে তোমাকে যাচাই করতে হতে পারে worst কেইসে।

হীপকে আমরা কখনো কখনো প্রায়োরিটি কিউ (priority queue) বলে থাকি। আমরা কিউ এর উদাহরণ দিতে বাসের লাইনের কথা বলেছিলাম, এখন মনে কর, বাসের লাইন ওরকম আগে এলে আগে যাবেন এরকম না হয়ে কে কত গণ্যমান্য ব্যক্তি তার উপর ভিত্তি করে হবে। তখন এটা হয়ে যাবে প্রায়োরিটি কিউ। যত জন মানুষ আছে তাদের মধ্যে সেই যাবে যার প্রায়োরিটি সবচেয়ে বেশি। এই জিনিসই কিন্তু আমাদের হীপ. STL এ প্রায়োরিটি কিউ বানিয়ে দেওয়া আছে। কোড ৫.৪ এ তোমাদের এই STL এর ব্যবহার দেখানো হলো। তোমরা চাইলে শুধু int না, যেকোনো স্ট্রাকচারেরও প্রায়োরিটি কিউ বানাতে পার তবে সেক্ষেত্রে তোমাদের অপারেটর ওভারলোডিং (operator overloading) করতে হবে। আমরা আগের অধ্যায়েই কীভাবে অপারেটর ওভারলোডিং করতে হয় তা দেখে এসেছি।

কোড ৫.৪: priority queue.cpp

```

1 #include<priority_queue>
2 using namespace std;
3
4 priority_queue<int> PQ; //declare a max heap
5 PQ.push(4);           //insert
6 PQ.top();             //maximum element

```

```

    7 PQ.pop();
    8 PQ.size();
    9 PQ.empty();
    //pop max
    //returns size of heap
    //returns 1 if heap is empty

```

৫.৮ ডিসজয়েন্ট সেট ইউনিয়ন (Disjoint set Union)

মনে কর তোমাকে কিছু কোম্পানির নাম দেওয়া আছে। প্রথমে সব কোম্পানির মালিক আলাদা আলাদা। এর পর একে একে বলা হবে যে অনুক কোম্পানির মালিক অনুক কোম্পানি কিনে নিয়েছে। মাঝে মধ্যে প্রশ্ন করা হবে যে, এই কোম্পানির মালিক কে বা এই কোম্পানির মালিক আসলে কতগুলো কোম্পানির মালিক বা সে যত কোম্পানি ক্রয় করেছে তাদের মধ্যে সবচেয়ে বেশি লোকজন কাজ করে কোন কোম্পানিতে? এরকম নানা প্রশ্ন করা হতে পারে। এই সব ক্ষেত্রে ডিসজয়েন্ট সেট ইউনিয়ন (Disjoint Set Union) ডেটা স্ট্রাকচার ব্যবহার করে সমাধান করা সম্ভব। একে অনেকে ইউনিয়ন ফাইন্ড (Union Find) ও বলে থাকে।

এই ডেটা স্ট্রাকচারের জন্য আমাদের একটিমাত্র অ্যারে দরকার, ধরা যাক তা হলো p . $p[i]$ এর মানে হলো i কোম্পানির মালিক হলো $p[i]$ কোম্পানির মালিক। প্রথমে সব i এর জন্য $p[i] = i$. এর পরে কখনো 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 হয়ে যদি সেটা b হয়, তাহলে $p[b]$ দেখবে, এরকম করে চলতে থাকবে। এখন কথা হলো এতে তো অনেক সময় লাগার কথা, যদি আমাদের p অ্যারেটি এমন থাকে যে, $p[1] = 2, p[2] = 3, \dots, p[n-1] = n$ তাহলে যদি $a = 1$ হয় তাহলে প্রতিবার $O(n)$ সময় লাগবে। এখন খেয়াল কর তুমি যদি একবার 1 এর জন্য বুঝে যাও যে n হলো আসল মালিক তাহলে কি তুমি $p[1] = n$ লিখতে পার না? একইভাবে, তুমি 1 এর মালিক খোঁজার সময় $2, 3, \dots, n-1$ এর ভেতর দিয়ে গিয়েছ এবং সব শেষে তুমি জেনেছ যে এদের সবার মালিক হলো n । সুতরাং এখন তুমি চাইলে সবার মালিক পরিবর্তন করে n করে দিতে পার, অর্থাৎ $p[1] = p[2] = \dots = n$ । এতে করে কোনো ক্ষতি নেই, বরং তুমি এতক্ষণ যে অনেক বড় ধারা পার করে যে আসল উত্তর বের করেছ, পরে আর কখনই এই বড় ধারা ডিঙ্গাতে হবে না। একে পাথ কমপ্রেশন (path compression) বলে। আমরা ফাইন্ড (Find) এর সাহায্যে এই জিনিস বের করে থাকি। Find এর কোড ৫.৫ এ দেখতে পার।

কোড ৫.৫: 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
7     p[x] = Find(p[x]);
8     return p[x];
9 }

```

```

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 আসা যাক, a এর মালিক যদি b কোম্পানিকে কিনে নেয় তাহলে কী করবে? যদি ভেসে দেখ যে, $p[b] = a$ করবে তাহলে ভুল। কারণ b কিন্তু b কোম্পানির মালিক নাও হতে পারে। b কোম্পানির মালিক কে? এই যে কিছুক্ষণ আগে বের করা হলো: $Find(b)$. সূতরাং আমরা যা করব তা হলো, $p[Find(b)] = a$ এর মানে হলো b এর মালিক এখন a দিয়ে নিয়ন্ত্রিত। তোমরা চাইলে $p[Find(b)] = Find(a)$ ও করতে পার। একেই ইউনিয়ন (Union) বলে। এর কোডও ৫.৫ এ আছে। অনেক সময় আমাদের জানার দরকার হতে পারে যে কোন মালিকের অধীনে কতগুলো কোম্পানি আছে বা যেসব কোম্পানি আছে তাদের মধ্যে কোনটিতে সবচেয়ে বেশি মানুষ কাজ করে। এসব ক্ষেত্রে আমাদের যা করতে হবে তাহলো p ছাড়াও আমাদের $total$ বা max এর তথ্য রাখতে হবে এবং প্রতিটি ইউনিয়ন বা ফাইন্ড অপারেশনের সময় এই তথ্যগুলো আমাদের যথাযথভাবে আপডেট করতে হবে।

তোমরা ভাবতে পার এই অ্যালগরিদমের নামে সেট আছে অথচ এখানে সেটের কিছুই নেই? না আছে। আমি এখানে বুঝানোর সময় প্যারেন্ট বা মালিক দিয়ে বুঝিয়েছি কিন্তু আসলে আমরা এখানে দুইটা সেটের ইউনিয়ন করছি বা কে কোন সেটে আছে তা বের করছি। শুরুতে আমরা মনে করতে পারি যে যার সেটে আছে। এরপর দুটি সেট ইউনিয়ন করা মানে তো বুঝছোই? তাদেরকে একত্র করে ফেলা। আর ফাইন্ড অপারেশন দিয়ে মূলত যা করা হয় তা হলো এটা দেখা যে দুজন একই সেটে আছে কি না। এসবের জন্য আমরা ভাবতে পারি যে প্রতিটি সেটের একটি মালিক আছে। যখন আমরা দুটি সেট ইউনিয়ন করছি তখন আসলে এক সেটের মালিক আরেক সেটকে কিনে নিচ্ছে। আবার যখন আমরা দেখতে চাইছি দুজন একই সেটে কি না, এর মানে হলো এটা দেখা যে দুজনের মালিক একই কি না। আশা করি এখন সেটের ব্যাপারটা পরিষ্কার হয়েছে।

৫.৯ Square Root segmentation

একটি ছোট সমস্যা দিয়ে শুরু করি। মনে কর 0 হতে $n - 1$ পর্যন্ত n টি দান বাস্তু আছে। প্রথমে প্রতিটিতে 0 টাকা করে আছে। একজন করে আসে আর সে i তম বাস্তুর t টাকা দান করে চলে যায়। মাঝে মধ্যে তোমাকে জিজ্ঞাসা করা হবে যে i হতে j পর্যন্ত বাস্তুগুলোতে মোট কত টাকা আছে। তুমি কত কার্যকরভাবে এই সমস্যা সমাধান করতে পারবে? এখন খুব সাধারণ একটি সমাধান হলো: বাস্তু টাকা রাখতে হলে ওই বাস্তুর টাকার পরিমাণ বাড়িয়ে দেব: $amount[i] += t$ আর টাকার পরিমাণ জিজ্ঞাসা করলে i হতে j পর্যন্ত $amount$ যোগ করব। কিন্তু এখানে আপডেট অপারেশন

মাত্র $O(1)$ সময় নিলেও কুয়েরি (query) অপারেশন worst কেইসে $O(n)$ সময় নিবে। তাহলে আমাদের এক্ষেত্রে কুয়েরির জন্য সময় আপডেটের সময় থেকে বেশি। এখন সবগুলো সংখ্যা না যোগ করে কীভাবে আমরা অনেক সংখ্যার যোগফল বের করতে পারি? যদি আমরা 0 হতে x বাঞ্ছতে থাকা টাকার পরিমাণ $total[x]$ এ রাখি তাহলে খুব সহজেই $total[j] - total[i-1]$ করে i হতে j বাঞ্ছে থাকা মোট টাকার পরিমাণ পেয়ে যেতে পারি ($i = 0$ এর ক্ষেত্রে একটু সাবধানতা অবলম্বন করতে হবে), এক্ষেত্রে আমাদের কুয়েরি হয়ে যায় $O(1)$. কিন্তু এই যে $total[x]$ এটি নির্ণয়ের জন্য আমাদের i এ t টাকা আপডেটের সময় i হতে n পর্যন্ত $total$ এর পরিমাণ t করে বৃদ্ধি করতে হবে। অর্থাৎ এক্ষেত্রে আমাদের আপডেট হয়ে যাবে $O(n)$. আমাদের আসলে এর মাঝামাঝি কোনো একটি পদ্ধতি অবলম্বন করতে হবে, যেন কোনোটিই খুব বড় না হয়ে যায়। প্রথম পদ্ধতিতে আমাদের আপডেটে অনেক কম সময় লেগেছে কারণ আমরা খুব ছোট একটি জায়গায় পরিবর্তন করেছি, আবার দ্বিতীয় পদ্ধতিতে আমাদের কুয়েরি করতে কম সময় লেগেছে কারণ, অনেকগুলো সংখ্যার যোগফল আমরা এক জায়গায় রেখেছিলাম। আমরা যেটা করতে পারি তা হলো, 0 হতে x পর্যন্ত সকল সংখ্যার যোগফল একত্র করে না রেখে কিছু কিছু করে সংখ্যার যোগফল একত্র করে রাখতে পারি। ধরা যাক এই কিছুর পরিমাণ হলো k . অর্থাৎ, প্রথম k টি সংখ্যা (0 হতে $k-1$ বাঞ্ছের টাকার পরিমাণ) একত্রে $sum[0]$ এ থাকবে, দ্বিতীয় k টি সংখ্যার যোগফল (k হতে $2k-1$ বাঞ্ছের টাকার পরিমাণ) একত্রে $sum[1]$ এ থাকবে এরকম করে প্রতি k টি করে সংখ্যার যোগফল একত্রে থাকবে। তুমি যদি একটু ভালো করে চিন্তা কর তাহলে দেখবে i তম স্থানের সংখ্যা আসলে $sum[i/k]$ এ থাকে। সুতরাং আপডেট অপারেশনের সময় তোমাকে $amount[i]$ বৃদ্ধির সঙ্গে সঙ্গে $sum[i/k]$ কেও বাড়াতে হবে। অতএব আমাদের আপডেট হয় $O(1)$ সময়ে। কুয়েরি এর সময় আমরা আলাদা আলাদা করে যোগ না করে বেশির ভাগ স্থান গুচ্ছ গুচ্ছ করে যোগ করব। ধরা যাক আমাদের বলা হলো 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 লুপ চালাব, যদি তারা আলাদা হয় তাহলে, i হতে x সীমার শেষ পর্যন্ত যোগ করব, y সীমার শুরু হতে j পর্যন্ত যোগ করব আর $x+1$ হতে $y-1$ এর sum গুলো যোগ করব। কোনো একটি সীমা p এর শুরুর মাথার সূত্র হলো kp এবং শেষ মাথার সূত্র হলো $k(p+1)-1$. এই সুত্রদুটি ব্যবহার করে আমরা আমাদের যোগফলের প্রথম দুই অংশ লুপ চালিয়ে বের করে ফেলব। এই কাজ করতে আমাদের খুব জোর $2k$ অপারেশন লাগবে, আর মাঝের sum গুলো যোগ করতে আমাদের n/k টি যোগ করতে হতে পারে, কারণ যেহেতু প্রতিটি সীমার আকার k সুতরাং আমাদের মোট সীমার সংখ্যা n/k . তাহলে আমাদের কুয়েরির জন্য সময় লাগবে $O(k + n/k)$. তোমরা যদি ক্যালকুলাস জেনে থাক বা অসমতা (inequality) নিয়ে একটু ঘাঁটাঘাঁটি করে থাক তাহলে মনে হয় জানো এই মানটি সর্ব নিম্ন হবে যদি $k = \sqrt{n}$ হয়। এবং এক্ষেত্রে কুয়েরি 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 আপডেট অপারেশনে শুধু i কে t পরিমাণ না বাড়িয়ে যদি বলা হয় i হতে j পর্যন্ত সবাইকে t পরিমাণ বাড়াতে হবে তাহলে কীভাবে সমাধান করতে? এই সমস্যার সমাধানও আগের সমস্যার মতোই তবে প্রতি সীমার জন্য এক্ষেত্রে আলাদা আরেকটি

ভ্যারিয়েবল রাখতে হবে যা নির্দেশ করবে এই সীমার সব *amount* এর সঙ্গে অতিরিক্ত কত যোগ করলে তুমি আসল টাকার পরিমাণ পাবে। আশা করি এতক্ষণে বুঝতে পারছ যে আপডেটের সময় সীমাগুলোর ভেতরে গিয়ে প্রতিটিকে না বাড়িয়ে তুমি ওই নতুন ভ্যারিয়েবলের মান শুধু বাড়িয়ে দিলেই পুরো সীমার প্রতিটি সংখ্যার মান বেড়ে যাবে! এক্ষেত্রে কুয়েরি এর পাশাপাশি আপডেটের জন্যও $O(\sqrt{n})$ সময় লাগবে।

৫.১০ স্ট্যাটিক (Static) ডেটাতে কুয়েরি

এর আগে যা আলোচনা করলাম তাতে আমরা কুয়েরি করেছি, আপডেটও করেছি। কিন্তু যদি কোনো আপডেট না করা লাগে? অর্থাৎ প্রথমেই সব সংখ্যা বা তথ্য দিয়ে দেওয়া হবে তোমাকে, এর পরে কুয়েরি এর উন্নত দিতে হবে। আগের ডেটাতে কোনো রকম পরিবর্তন হবে না। এটি যদি যোগফলের জন্য কুয়েরি হয় তাহলে তো খুবই সোজা, সব i এর জন্য তোমরা 1 হতে i পর্যন্ত যোগফল বের করে রাখবে (1-indexing মনে করি), 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 উন্নত দিতে পারবে প্রতিটি কুয়েরির। এর জন্য প্রিপ্রেসিং (preprocessing) এর সময় লাগবে $O(n)$.

কিন্তু আমাদের কুয়েরি যদি যোগফল না হয়ে সর্বোচ্চ বা সর্বনিম্ন হয়? একটি উপায় হলো আগের মতো Square Root Segmentation ব্যবহার করা। সেক্ষেত্রে আমাদের প্রিপ্রেসিং সময় লাগবে $O(n)$ আর কুয়েরির জন্য সময় লাগবে $O(\sqrt{n})$. Tarjan এর একটি বিখ্যাত গবেষণা আছে এই ব্যাপারে। সেই পদ্ধতিতে প্রিপ্রেসিং $O(n)$ সময়ে এবং কুয়েরি $O(1)$ সময়ে করা সম্ভব। তবে সেই পদ্ধতিটি বেশ জটিল। তোমরা চাইলে পড়ে দেখতে পার এই ব্যাপারে ইন্টারনেটে। আমরা এখন যেই পদ্ধতিটি দেখব তাতে আমাদের প্রিপ্রেসিং সময় লাগবে $O(n \log n)$ আর কুয়েরি সময় লাগবে $O(1)$. যেহেতু সর্বোচ্চ এবং সর্বনিম্ন বের করার পদ্ধতি প্রায় একই আমরা এখানে সর্বোচ্চ বের করব। সংক্ষেপে এই পদ্ধতিটি হবে এরকমঃ

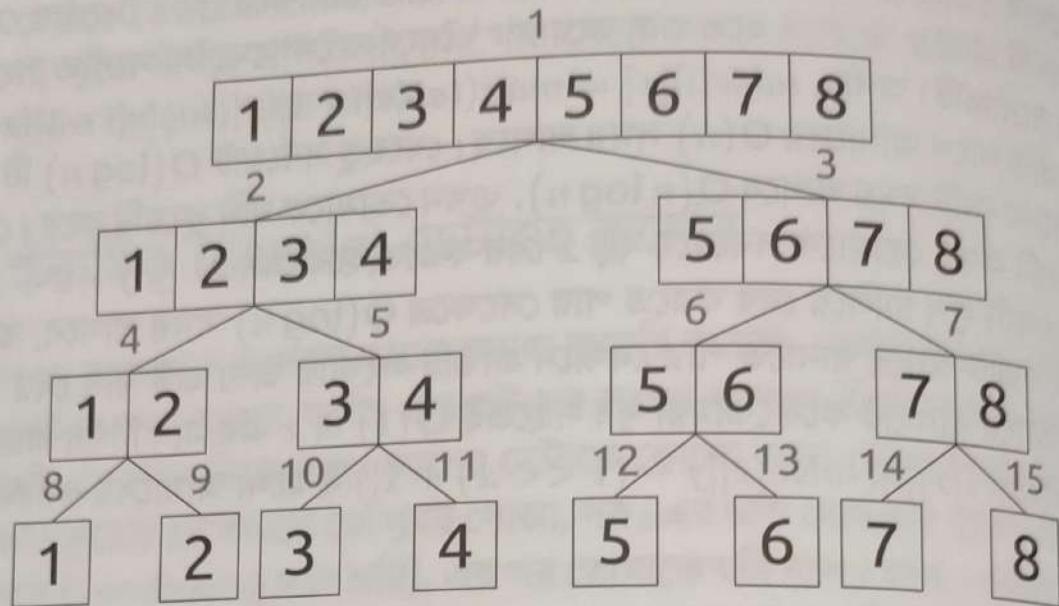
১. প্রথমে আমরা প্রতি 1 দৈর্ঘ্যের সেগমেন্ট (segment) এর সর্বোচ্চ সংখ্যাটি বের করঃ $[1, 1], [2, 2], [3, 3], \dots$
২. এর পর আমরা প্রতি 2 দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি বের করঃ $[1, 2], [2, 3], [3, 4], [4, 5] \dots$
৩. এর পর আমরা প্রতি 4 দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি বের করঃ $[1, 4], [2, 5], [3, 6], [4, 7] \dots$
৪. এর পর আমরা প্রতি 8 দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি বের করঃ $[1, 8], [2, 9], [3, 10], [4, 11] \dots$
৫. এভাবে i তম ধাপে আমরা 2^i দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি বের করঃ $[1, 2^i], [2, 2^i + 1] \dots$

এভাবে চলতে থাকবে যতক্ষণ $2^i \leq n$ হয় অর্থাৎ $i \leq \log(n)$. এই প্রিপ্রসেসিংয়ের সময় আমরা x হতে শুরু করে 2^i দৈর্ঘ্যের সর্বোচ্চ সংখ্যাটি কীভাবে বের করব? খুব সহজ, x হতে শুরু করে 2^i দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি হবে $x + 2^{i-1}$ হতে শুরু করে 2^{i-1} দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি এবং $x + 2^{i-1}$ হতে শুরু করে 2^{i-1} দৈর্ঘ্যের সেগমেন্টের সর্বোচ্চ সংখ্যাটি- এই দুটি সংখ্যার সর্বোচ্চটি। অর্থাৎ, $\text{table}[i][x] = \max(\text{table}[i-1][x], \text{table}[i-1][x+(1 << (i-1))])$. 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 কুয়েরি করে i থেকে j এর সর্বোচ্চ সংখ্যাটি কত? তোমাকে সবচেয়ে বড় x বের করতে হবে যেন $2^x \leq j - i + 1$ হয়। এটি তুমি চাইলে একটি লুপ চালিয়ে বের করতে পার সেক্ষেত্রে $O(\log n)$ সময় লাগবে, তবে তোমরা প্রথমেই যদি একটি অ্যারে বানাতে পার যেখানে প্রতিটি সংখ্যার জন্য এই মান বের করা থাকে, তাহলে সেই অ্যারে ব্যবহার করে তোমরা খুব সহজেই $O(1)$ এ x এর মান নির্ণয় করতে পারবে। তাহলে $\max(\text{table}[x][i], \text{table}[x][j - (1 << x) + 1])$ ই হলো আমাদের কাঞ্জিত মান।

৫.১১ সেগমেন্ট ট্রি (Segment Tree)

বাইনারি সার্চ ট্রি কোড করা বেশ কষ্টকর ব্যাপার। সে তুলনায় এরই জাত ভাই সেগমেন্ট ট্রি (Segment Tree) অনেক ভদ্র। জাত ভাই এই অর্থে যে এখানেও BST এর মতোই ইনসার্ট, ডিলিট, আপডেট ইত্যাদি অপারেশন করা যায় তবে এক্ষেত্রে আমাদের সংখ্যাগুলো 1 হতে n এর মধ্যে সীমাবদ্ধ থাকে। শুধু সংখ্যা ইনসার্ট বা ডিলিট না, কোনো একটি ইনডেক্সে চাইলে আমি কিছু সংখ্যা replace করতে পারি বা একটি সীমায় আমরা কুয়েরি করতে পারি। তবে হট করে দুটি ইনডেক্সের মধ্যে নতুন একটি ইনডেক্স বসিয়ে দিতে পারব না। অর্থাৎ তুমি যদি চাও যে 1 আর 2 এর মধ্যে নতুন একটি জিনিস বসাবে তা হবে না। তাহলে এর সাহায্যে কী কী করা যায়? একটি ছোটখাটো তালিকা বানানো যাক: কোনো একটি সীমায় কুয়েরি যেমন: সংখ্যাগুলোর যোগফল, সবচেয়ে বড় সংখ্যা, জোড় সংখ্যাগুলোর যোগফল ইত্যাদি; কোনো একটি সংখ্যাকে পরিবর্তন করা; কোনো একটি সীমার প্রতিটি সংখ্যাকে আপডেট করা যেমন: নির্দিষ্ট সংখ্যা যোগ করা ইত্যাদি। এটি কোড করা তুলনামূলকভাবে অনেক সহজ। সাধারণত সেগমেন্ট ট্রি ব্যবহার করে আমরা যেসব সমস্যা সমাধান করতে পারি Square Root Segmentation ব্যবহার করেও করতে পারি। তবে Square Root Segmentation এর ক্ষেত্রে আমাদের আপডেট বা কুয়েরির কমপ্লেক্সিটি হয় $O(\sqrt{n})$ আর সেগমেন্ট ট্রি এর ক্ষেত্রে হয় $O(\log n)$ (সাধারণত)। Square Root Segmentation এর ক্ষেত্রে আমরা একটি সমস্যা নিয়ে আলোচনা করছিলাম: আপডেট হলো অ্যারের একটি স্থানে একটি সংখ্যা যোগ করা আর কুয়েরি হলো অ্যারের কোনো একটি সীমার যোগফল প্রিন্ট করা। আমরা এই সেকশনে দেখব কীভাবে এই সমস্যায় আপডেট ও কুয়েরি দুটি সেগমেন্ট ট্রি ব্যবহার করে $O(\log n)$ সময়ে করা যায়।

৫.১১.১ সেগমেন্ট ট্রি তৈরী করা



নকশা ৫.১২: সেগমেন্ট ট্রি তৈরী (Segment Tree Build)

$n = 8$ এর জন্য সেগমেন্ট ট্রি চিত্র ৫.১২ এ দেখানো হল। আপাতত ধূসর সংখ্যাগুলোকে বলা দাও। আমাদের কাছে মোট ৮টি জায়গা আছে $[1, 8]$ । আমরা যা করব তা হলো এই জায়গাগুলোর সমান দুই ভাগে ভাগ করব: $[1, 4]$ এবং $[5, 8]$. যদি আমাদের কাছে $[L, R]$ এরকম একটি সীমাখণ্ডক 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 সব সীমাখণ্ডক ভাগে ভাগ করতে থাকব যতক্ষণ না আমাদের সেগমেন্টে একটিমাত্র সংখ্যা থাকে, অর্থাৎ: $L = R$ মনে করো না যে আমাদের দেখানো সেগমেন্ট ট্রি তে n একটি 2 এর ঘাত বলে এটা সম্ভব হয়েছে যদি $n = 3$ হয় তাহলে আমাদের প্রথম সেগমেন্ট $[1, 3]$ কে ভাঙলে আমরা পাব $[1, 2]$ ও $[3, 3]$ এবং $[1, 2]$ কে ভাঙলে $[1, 1]$ ও $[2, 2]$ । এখন কথা হলো, আমরা তো খুব সুন্দর করে কাগজে-কলমে ছবি এঁকে ফেললাম কিন্তু এটা কোডে করব কীভাবে? এবার ধূসর সংখ্যাগুলো খেয়াল কর। আমরা প্রতিটি সেগমেন্টের একটি করে নম্বর দিয়েছি। ছবির মতো করে নম্বর দেওয়ার একটি বিশেষ আছে। খেয়াল করলে দেখবে, কোনো নম্বর x এর বামে নিচে (লেফট চাইল্ড) সবসময় $2x$ এর ডানে নিচে সবসময় $2x + 1$ থাকে। আবার তার উপরে (প্যারেন্ট) $x/2$ হয়। এই বুদ্ধি খাটিয়ে আমরা খুব সহজেই একটি সেগমেন্ট ট্রি বানাতে পারি। কোড ৫.৬ এ কীভাবে একটি সেগমেন্ট ট্রি বানাবে যায় তা দেখানো হলো। এখানে আমাদের প্রয়োজন মতো কোড পরিবর্তন করতে হবে। যেমন যদি বলা থাকে যে আমাদের পুরো সেগমেন্ট প্রথমে ফাঁকা তাহলে আমরা 0 দিয়ে ইনিশিয়ালাইজেশন করব। আবার অনেক সময় আমাদের বলা থাকে কোন ঘরে কত সংখ্যা আছে, সেক্ষেত্রে আমরা একদম শেষ সেগমেন্টে গিয়ে সঠিক সংখ্যা বসাব বা ফিরে এসে সঠিক যোগফল রাখব ($sum[at] = sum[at * 2] + sum[at * 2 + 1]$)। খেয়াল করলে দেখবে আমাদের ট্রি এর প্রথম লেভেলে আছে মাত্র 1 টি নোড, দ্বিতীয় লেভেলে আছে মাত্র 2 টি, এর পরে 4 টি, এরকম করে 8, 16... এর মতো মাত্র নোড আছে।

নোড, যা যোগ করলে দাঁড়ায় $2n$ (গুণোত্তর ধারা)। সুতরাং আমাদের তৈরী বা বিল্ড (build) এর টাইম কমপ্লেক্সিটি $O(n)$.

কোড ৫.৬: segmentTreeBuild.cpp

```

1 // call with build(1, 1, n)
2 void build(int at, int L, int R)
3 {
4     sum[at] = 0
5     if(L == R)
6     {
7         // might need to do
8         // something like: sum[at] = num[L]
9         return;
10    }
11    int mid = (L + R)/2;
12    build(at * 2, L, mid);
13    build(at * 2 + 1, mid + 1, R);
14    // do initialization like, if there was num:
15    // sum[at] = sum[at * 2] + sum[at * 2 + 1]
16 }
```

সেগমেন্ট ট্রি এর ক্ষেত্রে টাইম কমপ্লেক্সিটি এর থেকেও গুরুত্বপূর্ণ হলো মেমোরী কমপ্লেক্সিটি। কারণ অনেকেই ভুল দৈর্ঘ্যের অ্যারে নেওয়ার জন্য Run Time Error পেয়ে থাকে। সবসময় মনে রাখবে, তোমার n যত, ঠিক তার 4 গুণ বা তার বেশি দৈর্ঘ্যের অ্যারে ডিক্লেয়ার করতে হবে। এর কারণ হলো n কিন্তু সবসময় এরকম 2 এর ঘাতে থাকবে না। 2 এর ঘাত এ থাকলে এটি দুই গুণ। কিন্তু 2 এর ঘাতে না থাকলে আসলে এই মেমোরীর পরিমাণ সুনির্দিষ্টভাবে বের করা একটু কঠিন হয়। আমরা জানি x এবং $2x$ এর মধ্যে অবশ্যই একটি 2 এর ঘাত আছে। আর আমরা জানি 2 এর ঘাত এর ক্ষেত্রে দ্বিগুণ লাগে, সুতরাং আমরা 4 গুণ দৈর্ঘ্য ডিক্লেয়ার করে থাকি। অনেকে n এর পরের 2 এর ঘাতের দ্বিগুণ ডিক্লেয়ার করে। তাহলেও হবে, কিন্তু সেক্ষেত্রে একটু হিসাব-নিকাশ করতে হবে। তাই আমি এত কিছু না ভেবে চোখ বন্ধ করে চারগুণ দৈর্ঘ্যের অ্যারে নিয়ে থাকি।

৫.১১.২ সেগমেন্ট ট্রি আপডেট করা

প্রথমে আপডেট দিয়ে শুরু করা যাক। আমরা কোনো একটি সংখ্যাকে বাড়াতে চাই। আমরা রুট থেকে শুরু করব। আমাদের রুট হলো $[1, 8]$ এবং ধরা যাক আমরা 3 কে আপডেট করতে চাই। আমরা যা করব তাহলো আমাদের সংখ্যাটি যেদিকে আছে সেদিকে যাব অর্থাৎ, রুট হতে

[1, 4] এ যাব, এর পর [3, 4] এবং সবশেষে [3, 3]। এবং ফেরার পথে আমরা তৈরীর বা বিল্ড এর সময় যেভাবে যোগফল বা sum এর অ্যারেকে তৈরী করেছিলাম ঠিক সেভাবে আমরা sum এর অ্যারেকে আপডেট করব। অর্থাৎ আমাদের আপডেট ফাংশন দেখতে ৫.৭ এর মতো হবে (এখানে আমরা pos কে u পরিমাণ বাড়াতে চাই। আর সেজন্য আমরা সেগমেন্ট ট্রি এর at নোডে আছি যার সীমা হলো L হতে R.)। যেহেতু আমাদের ট্রি এর উচ্চতা $\log(n)$ সুতরাং আমাদের আপডেট এর টাইম কমপ্লেক্সিটি হবে $O(\log n)$ ।

কোড ৫.৭: segmentTreeUpdate.cpp

```

1 // call with update(1, 1, n, pos, u)
2 void update(int at, int L, int R, int pos, int u)
3 {
4     // sometimes instead of using if-else
5     // in line 14 and 15 you can use:
6     // if(pos < L || R < pos) return;
7     if(L == R)
8     {
9         sum[at] += u;
10    return;
11 }
12
13    int mid = (L + R)/2;
14    if(pos <= mid) update(at * 2, L, mid, pos, u);
15    else update(at * 2 + 1, mid + 1, R, pos, u);
16
17    sum[at] = sum[at * 2] + sum[at * 2 + 1];
18 }
```

৫.১১.৩ সেগমেন্ট ট্রি তে কুয়েরি করা

এখন আসা যাক কুয়েরিতে। আমরা জানতে চাই $[l, r]$ এই সীমায় থাকা সংখ্যাগুলোর যোগফল কত। আমরা আগের মতো ট্রি এর রুট হতে শুরু করে ধীরে ধীরে নিচের দিকে যেতে থাকব। যদি কখনো দেখি আমরা এখন যেই নোডে আছি তার সীমা আমাদের কুয়েরি সীমার বাইরে তাহলে তা আর এখান থেকে নিচে যাওয়ার দরকার নেই, তাই না? সুতরাং আমরা এখান থেকেই বলব যে এই সীমার জন্য উত্তর 0। যদি আমরা এমন একটি নোডে থাকি যা পুরোপুরি আমাদের কুয়েরি সীমার ভেতরে তাহলেও কিন্তু নিচে যাওয়ার দরকার নেই, সেক্ষেত্রে আমরা আমাদের এই নোডের sum

এর মান রিটার্ন করব। যদি এই দুই কেইসের কোনোটিই না হয় এর মানে দাঁড়ায় যে আমাদের কুয়েরি সীমা আসলে এই নোডের দুই চাইল্ডেই কিছু কিছু করে আছে। সুতরাং আমরা দুদিকেই যাব এবং দুদিক থেকে আসা sum কে যোগ করে রিটার্ন করব। এর কোড তোমরা কোড ৫.৮ তে দেখতে পাবে।

কোড ৫.৮: segmentTreeQuery.cpp

```

1 // call with: query(1, 1, n, l, r)
2 int query(int at, int L, int R, int l, int r)
3 {
4     if(r < L || R < l) return 0;
5     if(l <= L && R <= r) return sum[at];
6
7     int mid = (L + R)/2;
8     int x = query(at * 2, L, mid, l, r);
9     int y = query(at * 2 + 1, mid + 1, R, l, r);
10
11    return x + y;
12 }
```

এখন কথা হলো এর টাইম কমপ্লেক্সিটি কত! আমাদের মনে হতে পারে এর টাইম কমপ্লেক্সিটি অনেক বেশি! কেউ কেউ ভাবতে পারে যেহেতু আমাদের ট্রি তে $O(n)$ সংখ্যক নোড আছে তাই এর টাইম কমপ্লেক্সিটিও $O(n)$. না! খেয়াল কর যদি কখনো [1, 1] ও [2, 2] আমাদের কুয়েরি সীমার মধ্যে থাকে তার মানে দাঁড়ায় আমরা আসলে [1, 2] থেকেই ফিরে যাব। অর্থাৎ তুমি যদি আসলে অনেক বেশি সীমাকে কভার করতে চাও তাহলে একটি বড় সীমা থেকেই তুমি ফেরত যাবে। তা নাহয় বোৰা গেল কিন্তু কমপ্লেক্সিটি আসলে কত? একটু চিন্তা করে দেখ, আমরা কখন কাজ করছি? যখন নিচে নামছি। কখন নিচে নামছি? যখন আমাদের নোডের সীমা আমাদের কুয়েরি সীমার সঙ্গে আংশিকভাবে overlap করে। খেয়াল কর, আমাদের ট্রি এর কোনো লেভেলে কিন্তু দুটির বেশি আংশিকভাবে overlap করা নোড থাকবে না, তাই না? বাকিগুলো হয় বাইরে না হয় একদম ভেতরে হবে। আমরা তখনই নিচে নামি যখন আংশিকভাবে overlap হয়। যেহেতু প্রতি লেভেলে আংশিকভাবে overlap এর সংখ্যা 2 আর আমাদের লেভেল আছে $\log n$ টি সুতরাং আমাদের টাইম কমপ্লেক্সিটি হবে $O(\log n)$. আমরা হয়তো এই জিনিস অন্যভাবে প্রমাণ করতে পারতাম, কিন্তু আমি এখানে এভাবে দেখালাম। কারণ এখানে টাইম কমপ্লেক্সিটি যে আসলে অনেক বেশি না সেটা আমরা ট্রি এর স্ট্রাকচার দেখে প্রমাণ করলাম। এরকম প্রমাণ আরও পাবে। তোমরা যদি কখনো লিঙ্ক কাট ট্রি (Link Cut Tree) নিয়ে পড়ার সুযোগ পাও তখন সেখানে এরকম প্রমাণ দেখতে পাবে। এবং সেই প্রমাণ আমার কাছে অনেক অসাধারণ লেগেছিল!

৫.১১.৮ Lazy without Propagation

মনে কর তোমাদেরকে বলা হলো যে n টি বাল্ব পর পর আছে এবং শুরুতে তারা সবাই বন্ধ। এখন একটি অপারেশনে; হতে j পর্যন্ত সব বাল্ব toggle করতে বলা হতে পারে। ১ আবার তোমাকে কখনো কখনো জিজ্ঞাসা করা হতে পারে যে q তম বাল্বটি চালু আছে, নাকি বন্ধ? তুমি এই সমস্যার সমাধান সেগমেন্ট ট্রি ব্যবহার করে $O(\log n)$ এ করতে পারবে। এক্ষেত্রে ধারনাটি হলো যখন তুমি i হতে j পর্যন্ত বাল্বকে toggle করবে তখন কিন্তু তুমি এই সীমার মধ্যে প্রতিটি বাল্বকে আপডেট করতে পারবে না, তোমাকে গুচ্ছ ধরে আপডেট করতে হবে। ধর তোমার কাছে $n = 8$ টি বাল্ব আছে আর তোমাকে 1 হতে 4 পর্যন্ত বাল্ব আপডেট করতে বলা হলো। তুমি যা করবে তাহলো সেগমেন্ট ট্রি এর শুধু [1, 4] এর সেগমেন্টে লিখে রাখবে যে এই সীমার প্রতিটি বাল্ব toggle করা হয়েছে। চিত্র ৫.১২ এর সঙ্গে তুলনা করতে পার। যদি তোমাকে বলে 1 থেকে 3 পর্যন্ত toggle করতে হবে, তাহলে তুমি [1, 2] এবং [3, 3] এই সীমা দুটি আপডেট করবে। আপডেট এর সময় শুধু তুমি লিখে রাখবে যে এই সীমাটি করবার toggle হয়েছে। লাভ কী? ধর তোমাকে জিজ্ঞাসা করল 3 এর অবস্থা কী? তুমি যা করবে, রুট থেকে [3, 3] পর্যন্ত যাবে এবং গুনে দেখবে এটি মেই পেরে যাবে। তাহলে কুয়েরি যে মাত্র $O(\log n)$ এ হচ্ছে তা তো খুব সহজেই বোৰা যায়, কিন্তু আপডেট? আমরা কিন্তু ইতোমধ্যেই সাবসেকশন ৫.১১.৩ তে এরকম কিছু প্রমাণ করে এসেছি। সুতরাং আমাদের আপডেটও $O(\log n)$. কোড ৫.৯ এ কুয়েরি ও আপডেটের কোড দেওয়া হলো। আমাদের এই সমাধানে আমরা যে একদম নিচ পর্যন্ত না গিয়ে উপরেই কিছু লিখে রেখে শেষ করে ফেলেছি আপডেটের কাজ, একেই lazy বলা হয়। আমরা এখানে lazy কে কিন্তু ভেঙে নিচে নামাইনি, সে জন্য একে without propagation বলে। ভেঙে নিচে নামানোর মানে কীত পরবর্তী সাবসেকশনেই পরিষ্কার হয়ে যাবে।

কোড ৫.৯: lazyWithoutPropagation.cpp

```

1 void update(int at, int L, int R, int l, int r)
2 {
3     if(r < L || R < l) return;
4     if(l <= L && R <= r) {toggle[at] ^= 1; return;}
5
6     int mid = (L + R)/2;
7     update(at * 2, L, mid, l, r);
8     update(at * 2 + 1, mid + 1, R, l, r);
9 }
10

```

¹toggle অর্থ হলো চালু থাকলে বন্ধ করা বা বন্ধ থাকলে চালু করা।

```

11 //returns 1 if ON, 0 if OFF
12 int query(int at, int L, int R, int pos)
13 {
14     if(pos < L || R < pos) return 0;
15     if(L <= pos && pos <= R) return toggle[at];
16
17     int mid = (L + R)/2;
18     if(pos <= mid)
19         return query(at * 2, L, mid, pos) ^ toggle[at];
20     else return
21         query(at * 2 + 1, mid + 1, R, pos) ^ toggle[at];
22 }

```

৫.১১.৫ Lazy With Propagation

মনে করা যাক উপরের সমস্যায় আমাদের কোনো একটি বাল্ব সম্পর্কে না জিজ্ঞাসা করে জিজ্ঞাসা করা হবে যে / হতে r এর মধ্যে কতগুলো বাল্ব চালু আছে! বলে রাখা ভালো যে এই সমস্যাও একটু চিন্তা করলে without propagation এ সমাধান করা সম্ভব। কিন্তু আমরা এখানে দেখাব কীভাবে এই সমস্যা with propagation এ সমাধান করা যায়। সমাধানে যাওয়ার আগে আমাদের একটু চিন্তা করা দরকার আমাদের এই সমস্যার সমাধানের জন্য ট্রি এর প্রতি নোডে কী কী জিনিস দরকার! প্রথমত Lazy দরকার, অর্থাৎ এই সীমার প্রতিটি বাল্ব কি toggle করা হয়েছে কি হয়নি এবং আরও দরকার এই সীমার কতগুলো বাল্ব এখন চালু আছে। বন্ধ বাল্বের সংখ্যা কিন্তু দরকার নেই, কারণ তুমি যদি চালু বাল্বের সংখ্যা জানো তাহলে বন্ধ বাল্বের সংখ্যা এমনিতেই বেরিয়ে আসবে। সুতরাং বিল্ড পর্যায়ে আমাদের প্রতি নোডে লিখতে হবে $toggle = 0$ এবং $on = 0$. এখন আসা যাক আমরা কীভাবে আপডেট করব। আগের মতোই আমরা দেখব যদি আমাদের বর্তমান নোডের সীমা কুয়েরি সীমার সম্পূর্ণ বাইরে হয় তাহলে কিছু করব না, যদি আংশিকভাবে ভেতরে হয় তাহলে সেভাবেই আমরা ডানে বা বামে যাব (বা উভয় দিকে)। এখন আসা যাক যদি সম্পূর্ণভাবে ভেতরে হয় তাহলে কী করব। খুব সহজ, $toggle = toggle \wedge 1$ করব, এবং $on = R - L + 1 - on$ করব। আশা করি বোৰা যাচ্ছে এই দুই লাইনে আসলে কী করা হচ্ছে। কিন্তু শুধু এটুকু করলে কিন্তু হবে না। কেন? একটু দূরের চিন্তা কর। মনে কর তুমি [1, 4] কে এভাবে আপডেট করলে। এর পর যদি তোমাকে বলে [1, 2] কে আপডেট করতে হবে। তুমি কী করবে? ওই নোডে গিয়ে একইভাবে আপডেট করে আসবে তাই না? কিন্তু যদি এর পরে তোমাকে কুয়েরি করে [1, 4] এ কতগুলো বাল্ব চালু আছে, তখন তুমি কীভাবে উত্তর দিবে? তুমি কিন্তু নিচে [1, 2] তে পরিবর্তন করে এসেছ, সুতরাং তুমি [1, 4] থেকেই উত্তর দিতে পারবে না এখন, কারণ [1, 2] এর পরিবর্তন [1, 4] এ কিন্তু নেই।^১ তাহলে উপায় কী?

¹একটু চিন্তা করলে তোমরা without propagation এ তাহলে কী করতে হবে তা বের করে ফেলতে পারবে

আগে দেখ আমাদের সমস্যাটা কী! আমরা যে [1, 4] এর আপডেটের সময় সেখান থেকেই কিন্তু গিয়েছি সেটি সমস্যা। আমরা যদি তা না করে একদম নিচ পর্যন্ত নামতাম এবং নোডগুলোর ওঁ ঠিক মতো আপডেট করতাম তাহলেই হয়ে যেত। কিন্তু তা করলে আমাদের টাইম কমপ্লেক্সিটি বেড়ে যাবে। তাহলে আমরা কী করব? উপায় হলো, তুমি এখানেই lazy রেখে যাবে, কিন্তু যদি কখনো এর থেকে নিচে নামতে হয় তাহলে তখন তুমি এই lazy কে এক ধাপ নামিয়ে দিবে। অর্থাৎ, আমরা যতক্ষণ না দরকার পরবে ততক্ষণ আমরা lazy নামাব না। এই যে lazy কে দরকারের সময় নিচে নামানো- একেই বলে propagation. আমরা [1, 4] এর আপডেটের পর যখন [1, 2] কে আপডেট করব তার আগেই অর্থাৎ যখন আমরা [1, 4] থেকে নিচে নামতে চাইব তখন আমরা দেখব [1, 4] এ কোনো 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। একইভাবে কুয়েরি এর সময়ও আমরা যদি কোনো নোড দিয়ে নিচে নামতে চাই, নামার আগেই আমাদের দেখে নিতে হবে এখানে কোনো lazy আছে কিনা এবং সেই অনুসারে তাকে দরকার হলে নিচে নামাতে হবে।

আরও কিছু বলার আগে কয়েকটা জিনিস পরিষ্কার হওয়া দরকার। প্রথমত lazy কী? আমরা আপডেট করার সময় প্রতিটি ইনডেক্সকে আপডেট করা অনেক সময় সম্ভব হয় না, কারণ সব ইনডেক্সে গিয়ে গিয়ে আপডেট করতে অনেক সময় লাগবে। সেজন্য আমরা Square root segmentation এর মতো একটি সীমায় কোনো একটি ভ্যারিয়েবলে আমরা লিখে রাখি যে এই সীমার সব ইনডেক্সে আমরা এতো যোগ করেছি, বা এই সীমার সব বাল্কে আমরা toggle করেছি ইত্যাদি। এইয়ে পুরো সীমার জন্য এই তথ্যটা আমরা লিখে রাখছি এটাই lazy. তৃতীয় প্রশ্ন, আমাদের lazy কেন দরকার? কারণ আমরা যদি lazy না রাখতাম তাহলে সব নোডকে আপডেট করতে অনেক সময় লাগত। তৃতীয় প্রশ্ন, lazy রাখার সমস্যা কী? মনে কর উপরের কোনো নোডে আমরা lazy রাখলাম। এখন আমরা এর নিচের কোনো সীমার জন্য কুয়েরি করলাম। তাহলে ওই কুয়েরির উত্তর এই উপরের নোডের lazy কে হিসাবে ধরবে না। অথবা যদি আমরা নিচে কোনো সীমায় আপডেট করি তাহলে উপরের এই lazy এর প্রভাবে ওই সীমায় যেই পরিবর্তন হয় সেটা আমরা প্রয়োগ করতে পারছি না। তাহলে এটা সমাধানের উপায় কী? সমাধান হলো, lazy propagation. যখনই আমরা কুয়েরি বা আপডেট করতে একটি নোডের নিচে যাব তখন আমরা দেখবো সেই নোডে কোনো lazy আছে কি না। যদি থাকে তাহলে সেই lazy কে propagate করতে হবে। Propagate মানে হলো, আগে হয়তো প্যারেন্টে লিখা ছিল এর পুরো সীমায় কি পরিমাণ toggle করা হয়েছে এখন সেটা চাইল্ডে লিখা থাকবে। এখন আমরা আরও বিস্তারিত দেখি কীভাবে এর lazy কে আমরা নামাতে পারি? তার আগে চিন্তা করে দেখ, একটি lazy কে নামালে কে কে পরিবর্তন হতে পারে? আমার নোডের toggle ও on, আমার লেফট ও রাইট চাইল্ডের toggle ও on. Lazy থাকা মানে হলো $toggle = 1$. একে নিচে নামানো মানে, আমার লেফট ও রাইট চাইল্ডের $toggle = toggle \wedge 1$ হবে এবং একই সঙ্গে on ও পরিবর্তন হবে। আর আমার বর্তমান নোডের $toggle = 0$ হবে, কিন্তু on পরিবর্তন হবে না (কারণ আমরা যখন toggle করেছিলাম তখনই আসলে on পরিবর্তন করেছিলাম)। একটু চিন্তা করলে দেখবে যে, যদি পর পর দুবার তোমাকে [1, 4] এ toggle করতে বলে তাহলে কিন্তু তোমাকে এর মধ্যে

propagation করার দরকার নেই। কারণ তুমি নিচে নামছনা, শুধু দুবার toggle = toggle ^ 1 এবং on = R - L + 1 - on করতে হবে। এবং এর ফলে প্রথমবারে যেই lazy জমা হতো সেটা পরের আপডেটের কারণে কাটাকাটি হয়ে যাচ্ছে। অর্থাৎ আমাদের প্রয়োগে আসলে lazy কাটাকাটি ও হতে পারে। আসলে কাটাকাটি হলো কি হলো 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 ৫.১০ এ এই কোড দেওয়া হলো।

কোড ৫.১০: 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 {
17     if(r < L || R < l) return;
18     if(l <= L && R <= r)
19     {
20         toggle[at] ^= 1;
21         on[at] = R - L + 1 - on;
22         return;
23     }
24
25     if(toggle[at]) Propagate(at, L, R);
26
27     int mid = (L + R) / 2;

```

```

২৮     update(at == 2, L, mid, l, r);
২৯     update(at == 2 + 1, mid + 1, R, l, r);
৩০
৩১     on[at] = on[at == 2] + on[at == 2 + 1];
৩২ }
৩৩
৩৪ int qurery(int at, int L, int R, int l, int r)
৩৫ {
৩৬     if(r < L || R < l) return;
৩৭     if(l <= L && R <= r) return on[at];
৩৮
৩৯     if(toggle[at]) Propagate(at, L, R);
৪০
৪১     int mid = (L + R)/2;
৪২     int x = query(at == 2, L, mid, l, r);
৪৩     int y = query(at == 2 + 1, mid + 1, l, r);
৪৪
৪৫     return x + y;
৪৬ }

```

৫.১১.৬ একটি উদাহরণ

নিঃসন্দেহে lazy এর ধারণা এই অধ্যায়ের সবচেয়ে কঠিন একটি ব্যপার। একে বুঝিয়ে গুছিয়ে বলাও একটু কঠিন। সেজন্য আরও একটি উদাহরণ দিয়ে এটা বোঝানোর চেষ্টা করি। আমাদের সমস্যা হলো, একটি সংখ্যার অ্যারে আছে। কুয়েরি খুব সহজ, i হতে j ইনডেক্সের মাঝের সর্বনিম্ন সংখ্যা বের করতে হবে। আর আপডেট হলো i হতে j ইনডেক্সের মাঝের প্রতিটি সংখ্যাকে। পরিমাণ বাড়াতে হবে। কীভাবে করা যায়? প্রথমত আমাদের সেগমেন্ট ট্রি তে প্রতিটি নোডে একটি ভ্যারিয়েবল রাখতে হবে যাতে থাকবে ওই সীমার সর্বনিম্ন। তাহলে আমরা যখন কুয়েরি করব তখন স্বাভাবিকভাবেই কুয়েরি করতে পারব। এখন কথা হলো আপডেট করব কীভাবে। আগের মতোই সীমাঙ্গলোতে যাব, সেখানে একটি lazy এর মতো থাকবে যা বলবে এই সীমার প্রতিটি সংখ্যাকে আরও কত যোগ করা হয়েছে। মনে কর A নোডে আমরা লিখলাম যে এই পুরো সীমায় S পরিমাণ যোগ করা হয়েছে। এরপরে যখন আমরা এই A নোড হতে নামব তখন এই S কে তার চাইলের মধ্যে দিয়ে দেব। অর্থাৎ বলব A তে কোনো অতিরিক্ত কিছু নেই, বরং এর চাইল্ডগুলোতে S করে অতিরিক্ত পরিমাণ যোগ করা হলো। এর ফলে আমাদের দুদিকের সর্বনিম্ন মানও কিন্তু S করে বেড়ে যাবে। এবার আমরা যেই আপডেট করতে এসেছি সেটা নিচের দিকে যাবে। আপডেট করা শেষে

যখন ফিরে আসব তখন আমার নোডের সর্বনিম্ন মানকে আপডেট করেত হবে। স্বাভাবিকভাবেই আপডেট করব অর্থাৎ, আমার সর্বনিম্ন মান হবে দুদিকের সর্বনিম্ন মানের মধ্যে যে কম। কারণ আমি যেই আপডেট করতে নিচে নেমেছি সেটা হয়তো কোনো একদিকে অনেক কম সংখ্যা তৈরি করে দিয়ে এসেছে। এভাবে lazy with propagation এর মাধ্যমে এই সমস্যা সমাধান করা যায়।

৫.১২ বাইনারি ইনডেক্সড ট্রি (Binary Indexed Tree)

সংক্ষেপে একে BIT বলা হয়। এটি সেগমেন্ট ট্রি এর মতোই একটি ডেটা স্ট্রাকচার তবে এটি একটু জটিল, কিন্তু মজার ব্যাপার হলো এর কোড খুবই ছোট। তুমি পুরো ডেটা স্ট্রাকচার না বুঝেও ব্যবহার করতে পারবে। সত্যি কথা বলতে আমি নিজেও এই ডেটা স্ট্রাকচার খুব ভালো মতো বুঝি না। কিন্তু এটি ব্যবহার করতে আমার বেশি কষ্টও হয় না। এটি ঠিক যে, BIT দিয়ে তুমি যা যা করতে পারবে সেগমেন্ট ট্রি দিয়েও প্রায় সবই তুমি করতে পারবে, তবে সেগমেন্ট ট্রি দিয়ে এমন কিছু করা যায় যা আসলে তুমি BIT দিয়ে করতে পারবে না। কিন্তু BIT এর সুবিধা হলো, এটি অনেক ছোট কোড, এর জন্য n আকারের মেমোরী লাগে এবং এটি অনেক দ্রুত।^১

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

১. কোনো একটি স্থান idx কে v পরিমাণ বৃদ্ধি $update(idx, v)$

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

আরও বেশ কিছু অপারেশন করা যায় যা আসলে বহুল ব্যবহৃত না। তোমরা topcoder এর চিউটেরিয়ালে দেখতে পার।

কোড ৫.১১ এ $read$ এবং $update$ দেখানো হল। এখানে $MaxVal$ হলো n এর মান। অর্থাৎ তোমার অ্যারে যত বড় আর কী! আর BIT এ তোমাকে 1 – indexing ব্যবহার করতে হবে।

কোড ৫.১১: bit.cpp

```

1 int read(int idx)
2 {
3     int sum = 0;
4
5     while (idx > 0)
6     {
7         sum += tree[idx];
8     }
9 }
```

^১তোমরা এর সম্পর্কে আরও বিস্তারিত জানতে চাইলে টপকোডারের এই <https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/> আর্টিকেলটি পড়তে পার।

```

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 }

```

৫.১৩ প্রোগ্রামিং সমস্যা

৫.১৩.১ অনুশীলনী

সহজ

ঃ UvaLive 6986* ঃ UvaLive 7025

সামান্য কঠিন

ঃ UvaLive 6830* ঃ UvaLive 6838* ঃ UvaLive 6857* ঃ UvaLive 6860**
ঃ UvaLive 6898 ঃ UvaLive 6900 ঃ UvaLive 6936* ঃ UvaLive 7057*

কঠিন

ঃ UvaLive 6946***

অধ্যায় ৬

গ্রীডি টেকনিক (Greedy Technique)

গ্রীডি (Greedy) মানে তো সবাই বোঝো? এর মানে হলো লোভী। যেমন ধর তোমাকে একটি বুফে (buffet) তে নিয়ে গিয়ে ছেড়ে দিলে কী করবে? তুমি হাপুস হপুস করে খাওয়া শুরু করে দেবে তাই না? যদি একটু বুদ্ধিমান হও তাহলে হয়তো সবচেয়ে দামি খাবার বেশি বেশি করে খাবে! কারণ যার দাম কম তা হয়তো তুমি পরে কিনে খেতেই পারবে! যেমন যদি বুফেতে গলদা চিংড়ি থাকে আর জিলাপি থাকে, তাহলে নিশ্চয়ই জিলাপি খেয়ে পেট ভরানোর থেকে চিংড়ি খেয়ে পেট ভরানো বুদ্ধিমানের মতো কাজ হবে? গ্রীডি মানে যে সবসময় বেশি বেশি করে নেওয়া তা কিন্তু না। অনেক সময় কম কম নেওয়াও লাভজনক। যেমন তোমাকে বলা হলো একটি গাড়ি কিনতে। এখন গাড়িগুলো একেকটা একেক পরিমাণ তেল খায়! নিশ্চয়ই যেই গাড়ি সবচেয়ে কম তেল খায় সেটা কেনাই বুদ্ধিমানের মতো কাজ তাই না? যদিও বাস্তব জীবনে আরও অনেক হিসাব-কিতাব আছে! যাই হোক, তো গ্রীডি মানে বলতে পার অন্য কিছু না দেখে যার মান কম বা বেশি তাকে সবসময় বাছাই করা।

৬.১ Fractional Knapsack

গ্রীডি অ্যালগরিদমের জন্য এটি খুবই পরিচিত সমস্যা। মনে কর একটি চোর একটি মুদি দোকানে চুকেছে চুরি করতে। সেখানে চাল আছে, ডাল আছে, চিনি, লবণ এরকম নানা জিনিস আছে। এখন সে সব জিনিস চুরি করতে পারবে না। কারণ তার কাছে যেই থলে আছে তার ধারণক্ষমতা ধরা যাক 20 কেজি। তাহলে সে কীভাবে চুরি করলে সবচেয়ে বেশি লাভবান হবে? খুবই সহজ। যেই জিনিসটির দাম সবচেয়ে বেশি সে সেই জিনিস আগে নেওয়া শুরু করবে। যদি দেখে ওই জিনিস শেষ এবং এখনো থলেতে কিছু জায়গা বাকি আছে তাহলে সে পরবর্তী দামি জিনিস নেওয়া শুরু করবে। এরকম করে যতক্ষণ না তার থলের ধারণক্ষমতা শেষ হচ্ছে সে নিতে থাকবে। এখানে খেয়াল কর, দাম বেশি মানে কিন্তু প্রতি কেজিতে দাম। ধর চাল আছে 1 কেজি আর দাম 100 টাকা, আর ডাল আছে 500g কিন্তু এর দাম 60 টাকা তাহলে কিন্তু ডাল নেওয়া লাভজনক হবে

কারণ, ডালের দাম প্রতি কেজিতে 120 টাকা!

এই সমাধান ঠিক থাকবে যদি তুমি কোনো জিনিসের যেকোনো পরিমাণ নিতে পার। সমস্যাটা যদি চাল ডালের দোকানে না হয়ে ইলেক্ট্রনিক্সের দোকানে হয় তাহলে তুমি আর এভাবে সমাধান করতে পারবে না। তুমি তো আর একটি টেলিভিশন ভেঙে এর অর্ধেক চুরি করবে না তাই না? টিভি হোক, ল্যাপটপ হোক আর মোবাইলই হোক তুমি যাই নিতে চাও না কেন পুরোটাই নিতে হবে। এক্ষেত্রে কিন্তু আমাদের গ্রীড়ি পদ্ধতি কাজ করবে না। উদাহরণ দেওয়া যাক, মনে কর ১ টি টিভির দাম 15000 টাকা এবং ওজন 15kg, দুটি মনিটর আছে যাদের ওজন 10kg করে এবং প্রতিটির দাম 9000 টাকা। তোমার কাছে 20 kg জিনিস নেওয়ার থলে আছে। তুমি কী করবে? টেলিভিশন নেওয়া কিন্তু বোকামি হবে যদিও এর প্রতি কেজিতে দাম বেশি তাও তোমার দুটি মনিটর নিলে লাভ হবে সবচেয়ে বেশি। সুতরাং এটি মনে করার কিছু নেই যে গ্রীড়ি পদ্ধতি সবসময় কাজ করবে। যদি তুমি জিনিসের চাইলে "কিছু" অংশ নিতে পার তাহলে সেই সমস্যাকে বলা হয় Fractional Knapsack আর যদি তোমাকে পুরোপুরি নিতে হয় তাহলে সেই সমস্যাকে বলা হয় 0–1 knapsack (এটি পরবর্তী অধ্যায়ে আমরা দেখব কীভাবে সমাধান করতে হয়)। Fractional knapsack এর কোড ৬.১ এ দেখানো হলো। দেখানোর মূল উদ্দেশ্য অবশ্য STL এর আরও কিছু ব্যবহার দেখানো!

কোড ৬.১: fractional knapsack.cpp

```
১ #include <algorithm>
২ #include <utility>
৩ #include <vector>
৪ using namespace std;
৫
৬ // from now on, we can use PII
৭ // in the place of pair<....>
৮ // we will put weight at the first place
৯ // and price at the second place
১০ typedef pair<int, int> PII;
১১
১২ vector<PII> v;
১৩
১৪ bool cmp(PII A, PII B) {
১৫     // we want to order on decreasing order of
১৬     // price/weight.
১৭     // So: return A.price / A.weight >
১৮     //      B.price / B.weight
```

```

19 // But it is better if we can avoid division
20 // because of possible precision loss.
21 // So we can rewrite it as:
22 // return A.price * B.weight > B.price * A.weight
23 // we use first for weight, and second for price.
24 return A.second * B.first > A.first * B.second;
25 }
26
27 void fractional_knapsack() {
28     int n, W;
29     scanf("%d", &n);
30     for (int i = 0; i < n; i++) {
31         int weight, price;
32         scanf("%d %d", &weight, &price);
33         V.push_back(PII(weight, price));
34     }
35     sort(V.begin(), V.end(), cmp);
36     scanf("%d", &W);
37     int ans = 0;
38     for (int i = 0; i < n; i++) {
39         // W is remaining capacity.
40         // V[i].first is weight of the i'th element.
41         // so steal minimum of them.
42         int z = min(W, V[i].first);
43         W -= z;
44         ans += z * V[i].second;
45     }
46     printf("Maximum cost: %d\n", ans);
47 }

```

৬.২ মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree)

এই সেকশনের নাম দেখে ভয় পাওয়ার কিছু নেই। খুবই সহজ জিনিস। আমরা আগেই জেনে এসেছি ট্রি কাকে বলে, এখন দেখে নেওয়া যাক মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree) কী জিনিস। মনে কর আমাদের একটি ওয়েইটেড গ্রাফ (weighted graph) দেওয়া আছে

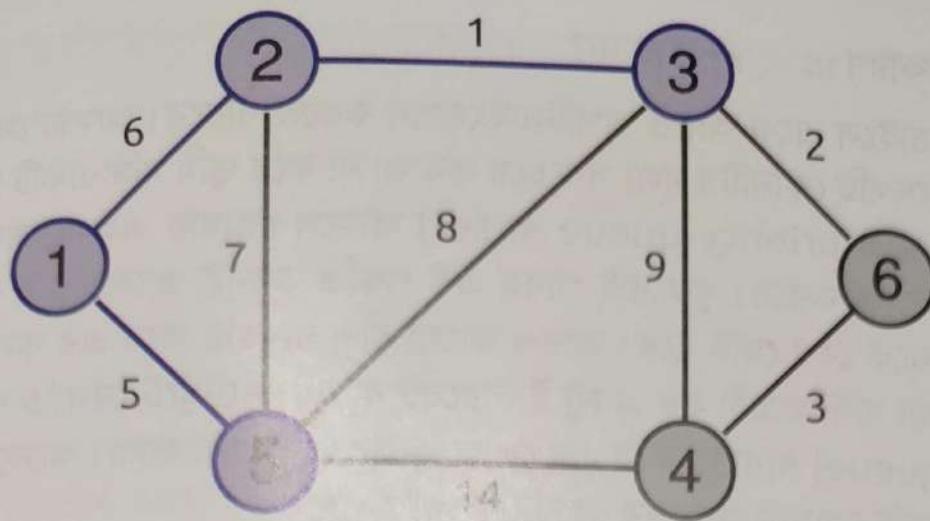
(ওজন বা weight গুলো ধনাত্মক) অর্থাৎ কিছু ভার্টেক্স (vertex), কিছু বাহু (edge) এবং সেই বাহুগুলোর মান বা ওজন (weight). তোমাকে এখন এদের মধ্য থেকে কিছু বাহু বাছাই করতে হবে যেন তাদের ওজনের যোগফল সর্বনিম্ন হয় এবং সব ভার্টেক্স যেন কানেক্টেড (connected) হয়। এটি নিচয়ই বুঝতে পারছ যে তোমার যদি n টি ভার্টেক্স থাকে তাদের কানেক্টেড করতে আসলে তোমার কমপক্ষে $n - 1$ টি বাহু লাগবেই। আর তুমি যদি তোমার বাছাই করা বাহুগুলোর ওজনের যোগফল সর্বনিম্ন করতে চাও তাহলে অবশ্যই $n - 1$ টির বেশি বাহু নিবে না। n সংখ্যক ভার্টেক্স এবং $n - 1$ সংখ্যক বাহুগুলো একটি কানেক্টেড গ্রাফ হলো ট্রি। অর্থাৎ আমাদের সব ভার্টেক্সকে কানেক্টেড করতে আমরা যেসব বাহু নির্বাচন করব তাদের ওজনের যোগফল সর্বনিম্ন করতে চাইলে যেই গ্রাফটি দাঁড়ায় সেটিই হলো মিনিমাম স্প্যানিং ট্রি (সংক্ষেপে MST). এখানে মিনিমাম (minimum) আর ট্রি শব্দ দুটি তো বুঝেছই? স্প্যানিং (spanning) অর্থ কানেক্টেড মনে করতে পার। এখানে আমরা MST বের করার জন্য দুটি অ্যালগরিদমের কথা বলব। তোমরা কেউ কেউ মনে করতে পার যে হয়তো এই সেকশনটি গ্রাফের অধ্যায়ে থাকলে ভালো হতো। কিন্তু আমরা যেই দুটি অ্যালগরিদম আলোচনা করব তারা আসলে গ্রীডি ধরনের বলা যায়। আর তোমরা কীভাবে একটি গ্রাফকে উপস্থাপন করা যায় তা তো শিখেই ফেলেছ! সুতরাং চিন্তা কী! আমাদের পরবর্তী দুটি সেকশনের জন্য ধরে নিই যে আমাদের ইনপুট গ্রাফে n টি ভার্টেক্স ও m টি বাহু আছে।

৬.২.১ প্রিম এর অ্যালগরিদম (Prim's Algorithm)

যেহেতু আমাদের সবগুলো ভার্টেক্সকে কানেক্টেড করতে হবে সুতরাং আমরা যেকোনো ভার্টেক্স থেকে শুরু করতে পারি। এখন আমরা দেখব, এই ভার্টেক্সের সঙ্গে যেই যেই বাহু আছে তাদের মধ্যে কার ওজন সবচেয়ে কম। যার সবচেয়ে কম সেই বাহু আমরা নিব এবং তাহলে আমাদের এখন দুইটা ভার্টেক্সও একটি বাহু হয়ে গেল। এখন দেখব, এই দুইটি ভার্টেক্স থেকে যেসব বাহু বের হয়েছে তাদের মধ্যে কার ওজন সবচেয়ে কম তাকে নিব। এভাবে নিতে থাকব যতক্ষণ না আমাদের সব ভার্টেক্স নেওয়া হয়ে যায়। এটা আশা করি বুঝেছ যে যখন এই সবচেয়ে কম ওজনের বাহু নিচ্ছ তখন সেই বাহুর এক মাথা তোমার ইতমধ্যে বানানো ট্রি এর ভেতরে যেন থাকে এবং অপর মাথায় যেন আমরা এখনো নির্বাচন করিনি এরকম ভার্টেক্স থাকে। দুই মাথাই যদি আমাদের ট্রি এর মধ্যে থাকে তাহলে কিন্তু লাভ নেই! কারণ তারা তো ইতোমধ্যেই কানেক্টেড, শুধু শুধু এই বাহু নিয়ে ওজনের যোগফল বাড়ানোর কি কোনো মানে আছে?

একটি উদাহরণ দেওয়া যাক। মনে কর চিত্র ৬.১ এ আমরা ১ নোড হতে প্রিম এর অ্যালগরিদম শুরু করেছি। তাহলে প্রথমে আমরা ১ – 5 বাহু নির্বাচন করব, এরপর ১ – 2, এরপর ২ – 3. কীভাবে আমরা এইসব বাহু নির্বাচন করছি? চিত্রের এই অবস্থা থেকে আমরা তা দেখি। চিত্রে নীল নোডগুলো হলো ইতোমধ্যেই নির্বাচিত এবং কালোগুলো এখনো নির্বাচন করা হয়নি। এখন সেসব বাহু দেখ যাদের এক মাথা নীল নোডে এবং অপর মাথা কালো নোডে। এরকম বাহুগুলো হলো ৩ – 6, ৩ – 4 এবং ৫ – 4. এদের মধ্যে সবচেয়ে কম ওজনের বাহু হলো ৩ – 6. সুতরাং আমাদের পরের নির্বাচিত বাহু হবে ৩ – 6 আর নির্বাচিত নোড হবে 6.

এখন কথা হলো এই অ্যালগরিদমের টাইম কমপ্লেক্সিটি কত! প্রথমত তুমি n সংখ্যক বাহু এই



নকশা ৬.১: প্রিমে এর অ্যালগরিদম (Prim's algorithm)

নতুন ভাট্টেক্স নির্বাচন করার কাজ করছ এবং প্রতিবার হয়তো তুমি সব বাহু দেখছ। সুতরাং তোমার কমপ্লেক্সিটি দাঁড়ায় $O(nm)$. একে তুমি খুব সহজেই $O(n^2)$ করতে পার। মনে কর তুমি প্রথমে a নামক ভাট্টেক্স নিয়েছিলে এবং আমাদের বর্তমান পদ্ধতিতে প্রতিবার a এর সঙ্গে লাগানো সব বাহু প্রতিবার যাচাই করছ। কিন্তু প্রতিবার যাচাই করার কি দরকার আছে? তুমি যখন একটি নতুন ভাট্টেক্স আমাদের ট্রি তে অন্তর্ভুক্ত করবে তখন এর সঙ্গে লাগানো সব বাহু দেখবে, দেখে সেই বাহুর অপর প্রান্ত কত কম খরচে আমাদের ট্রি তে নেওয়া যায় সেটি আপডেট করবে। অর্থাৎ আমাদের প্রতিটি নোডে একটি করে মান থাকবে যা নির্দেশ করবে কত কম খরচে সেই নোড আমাদের ট্রি তে অন্তর্ভুক্ত করা যায়। চিত্র ৬.১ এ খেয়াল কর, এই অবস্থায় নোড 4 এ থাকা মান হবে 9 এবং নোড 6 এ থাকা মান হবে 2. সুতরাং আমরা নির্বাচন করব নোড 6. এই নোড নির্বাচন করার পর আমরা এর সঙ্গে লাগানো সব বাহু দেখব এবং অপর মাথা প্রয়োজনে আপডেট করব। নোড 6 এর সঙ্গে লাগানো একটি বাহু হলো 6 – 4 এবং এর ওজন হলো 3. সুতরাং 3 – 6 বাহু নেওয়ার পর আমরা নোড 4 এর আগের মূল্য বা cost 9 আপডেট করে 3 করে দেব। একটু অন্যভাবে বলি। মনে কর, আমরা নোড 3 যখন নিয়েছি তখন দেখেছি যে নোড 4 কে আমরা 9 মূল্যে অন্তর্ভুক্ত করতে পারি, কিন্তু নোড 6 অন্তর্ভুক্ত হয়ে যাওয়ার পর দেখলাম যে নোড 4 কে আরও কম খরচে তুমি অন্তর্ভুক্ত করতে পারবে! তখন তুমি নোড 4 এর মূল্যকে আপডেট করবে। এটি গেল ভেতরের কাজ, আমাদের কিন্তু বাইরে একটি লুপ n বার চলছে যেটি প্রতিবার একটি একটি করে নতুন নোড নির্বাচন করছে। এই নোড কীভাবে নির্বাচন করা হচ্ছে তা মোটামুটি বলেছি তাও আরেকবার বলি, প্রতিবার আমাদের দেখতে হবে যে কোন কোন ভাট্টেক্স এখনো নির্বাচন করা হয়নি এবং তাদের মধ্য হতে সবচেয়ে কম খরচের ভাট্টেক্সকে তোমাকে নির্বাচন করতে হবে। তাহলে কী দাঁড়াল? তুমি মোট n বার কাজ করবে, প্রতিবার সব অনির্বাচিত ভাট্টেক্স যাচাই করে দেখবে যে কোনটি সবচেয়ে কম, তাকে নিবে। এর পর এর সঙ্গে লাগানো সব বাহু বরাবর গিয়ে দেখবে যে তার অপর প্রান্তে কোনো অনির্বাচিত ভাট্টেক্স আছে কিনা, থাকলে তার মূল্যকে আপডেট করবে। তাহলে আমাদের কমপ্লেক্সিটি কত? n সংখ্যকবার কাজ করছি আর প্রতিবার n টি ভাট্টেক্স আমরা যাচাই করে দেখছি আর আমরা সর্বমোট খুব জোর $2m$ বার বাহু যাচাই করছি, সুতরাং আমাদের কমপ্লেক্সিটি দাঁড়ায় $O(n^2 + m)$ যা আসলে

$O(n^2)$ বলা যায় কারণ $m < n$ তবে ...

তোমরা কিন্তু চাইলে একে আরও অপটিমাইজেশন করতে পারবে। আমরা যে প্রতিবার n টি ভার্টেক্স ঘুরে ঘুরে দেখছি কোনটির মূল্য সবচেয়ে কম তা না করে তুমি যদি একটি মিন হৈপ (min heap) রাখ (C++ এ priority queue বা set) তাহলে তোমার এই অ্যালগরিদম আসলে $O(m \log n)$ এ কাজ করবে। বুবাতেই পারছ এই পদ্ধতি তখনই ভালো কাজ করবে যখন n তোমার n^2 এর থেকে বেশ ছোট হবে। আরও ভালো হৈপ ব্যবহার করে এর কমপ্লেক্সিটি আরও কমান যায়। তোমরা যদি আগ্রহী হও একটু ইন্টারনেট বা বই দেখে দেখে দেখতে পার। থ্রোরিটিকিউ (Priority queue) ব্যবহার করে এর কোড ৬.২ এ দেখানো হলো। অনেকে আরও সুন্দর করে কোড করে, তাই তোমরা অন্যদের কোডখ দেখতে পার।

কোড ৬.২: prim.cpp

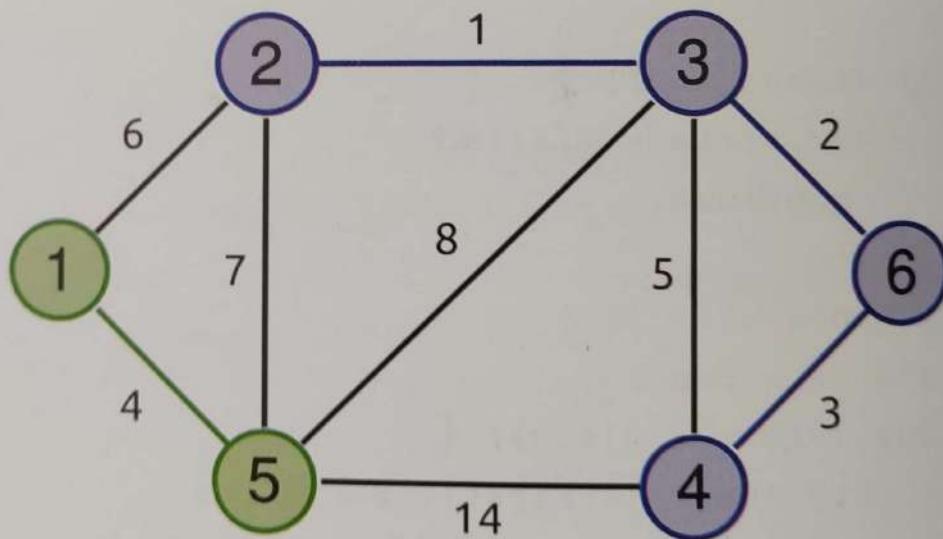
```
1  typedef pair<int, int> PII;
2  // weighted adjacency list.
3  // first element neighboring node
4  // second element weight of the edge
5  vector<PII> V[100];
6
7  struct Node {
8      int u, cost;
9      Node() {} // default constructor
10     Node(int _u, int _cost) {
11         u = _u;
12         cost = _cost;
13     }
14 };
15
16 bool operator<(Node A, Node B) {
17     // priority-queue is a max heap.
18     // so we need to compare the cost
19     // in reverse order to get minimum
20     // cost vertex at top.
21     return A.cost > B.cost;
22 }
23
24 priority_queue<Node> PQ;
```

```

20 // minimum cost so far for a node.
21 int cost[100];
22 // if the node is already taken.
23 int taken[100];
24
25 int prim() {
26     // n = number of nodes.
27     // INF = infinite, something like  $10^9$ .
28     for (int i = 0; i < n; i++)
29         cost[i] = INF, taken[i] = 0;
30     // s is the vertex you want to start prim from.
31     cost[s] = 0;
32     PQ.push(Node(s, 0));
33     int ans = 0;
34     while (!PQ.empty()) {
35         Node x = PQ.top();
36         PQ.pop();
37
38         if (taken[x.u]) {
39             // already visited.
40             continue;
41         }
42         taken[x.u] = 1;
43         ans += x.cost;
44         for (PII v : V[x.u]) {
45             if (taken[v.first]) continue;
46             if (v.second < cost[v.first]) {
47                 // cost of current edge to v.first
48                 // is less than whatever we saw
49                 // so far.
50                 PQ.push(Node(v.first, v.second));
51                 cost[v.first] = v.second;
52             }
53         }
54     }
55 }
```

৬.২.২ ক্রুসকাল এর অ্যালগরিদম (Kruskal's Algorithm)

এটিও বেশ সহজ অ্যালগরিদম। কোনো কারণে যখনই MST আমাকে কোড করতে হ্যাঁ আছি ক্রুসকাল এর অ্যালগরিদম (Kruskal's algorithm) ই করে থাকি। হয়তো আমার কাছে এটি সহজ লাগে সেজন্য! এই অ্যালগরিদম বোঝানো খুবই সহজ, কোড করাও অনেক সহজ কিন্তু যেভাবে কোড করতে হবে সেটি বোঝানো একটু কষ্টকর। এই অ্যালগরিদমে তুমি যা করবে তাহলে সবচেয়ে কম ওজনের বাহু নিবে, দেখবে এর দুই মাথার ভাট্টের দুটি ইতোমধ্যেই একই ট্রি বা component এ আছে কিনা, থাকলে এই বাহু নিবে না। না থাকলে নিবে। এভাবে মূলের উর্ধ্বক্রমে সব বাহুর ইউনিয়ন (union) নিয়ে এই কাজ করতে হবে। শেষ! এখন প্রশ্ন হচ্ছে কীভাবে বুঝবে যে দুটি ভাট্টের একই ট্রি তে আছে কিনা! উত্তর: ডিসজয়েন্ট সেট (Disjoint Set). অদ্যমে সব ভাট্টেরকে আলাদা আলাদা সেট আকারে কল্পনা কর। আমরা যখনই একটি বাহু নিচ্ছি তখন দুটি সেটকে জোড়া লাগানোর চেষ্টা করছি এবং সেজন্য যাচাই করছি যে, এই দুটি ভাট্টের একই সেটে আছে কিনা!



নকশা ৬.২: ক্রুসকাল এর অ্যালগরিদম (Kruskal's algorithm)

একটি উদাহরণ দেখা যাক। চিত্র ৬.২ এ আমরা প্রথমে 2 – 3 কে জোড়া দিয়েছি। এরপর 3 – 6, 6 – 4, 1 – 5। ওজনের উর্ধ্বক্রমে আসলে আমাদের পরের বাহু হবে 3 – 4 কিন্তু এই দুটি নোড একই ট্রি তে আছে সুতরাং আমরা আর এই বাহু জোড়া লাগাব না। এর পরের বাহু হলো 1 – 2 এবং এটি দুটি আলাদা ট্রি কে জোড়া লাগায় সুতরাং আমরা এই বাহু নিব এবং ট্রি দুটিকে জোড়া লাগাব। অর্থাৎ যদি ডিসজয়েন্ট সেট ইউনিয়নের ভাষায় বলতে হয়, তাহলে প্রথমে আমরা বাহুগুলোকে মূল্য অনুযায়ী ছোট হতে বড় আকারে সাজাবো। এরপর তাদের একে একে নিবো আর

দেখবো এই বাহুর দুই মাথা একই সেটে আছে কি না। থাকলে এই বাহু নিবো না। আর না থাকলে সেই বাহু নিবো আর তাদের দুই মাথার সেটগুলো ইউনিয়ন করে দেব।

এখন প্রশ্ন হলো এই অ্যালগরিদমের টাইম কমপ্লেক্সিটি কত? সহজ, বাহুগুলোকে ওজন অনুযায়ী সর্ট করতে $O(m \log m)$ এবং প্রতি বাহুর জন্য আমরা ফাইন্ড (find) করছি বা দুটি সেটকে ইউনিয়ন করছি যাদের কমপ্লেক্সিটি আমরা $O(1)$ ধরে নিতে পারি। সুতরাং $O(m \log m + m) = O(m \log m)$. এর কোড ৬.৩ এ দেওয়া হলো।

কোড ৬.৩: kruskal.cpp

```
1 struct Edge {
2     // there is an edge between u and v
3     // of weight w.
4     int u, v, w;
5 };
6
7 bool operator<(Edge A, Edge B) {
8     // sort edges according to weights.
9     return A.w < B.w;
10 }
11 // edge list is enough, no need for adjacency list.
12 vector<Edge> E;
13 // array to keep track of parents for union find.
14 int p[100];
15
16 int find(int x) {
17     if (p[x] == x) return x;
18     return p[x] = find(p[x]);
19 }
20
21 int kruskal() {
22     sort(E.begin(), E.end());
23
24     // once I read somewhere that, someone
25     // got TLE because he was invoking size
26     // again and again in the loop.
27     // So I usually store size in a variable
```

```

28      // before the loop.
29      // with C++11 now I use range loop.
30      int sz = E.size();
31      int ans = 0;
32      for (int i = 0; i < sz; i++) {
33          if (find(E[i].u) != find(E[i].v)) {
34              // union.
35              p[p[E[i].u]] = p[E[i].v];
36              ans += E[i].w;
37          }
38      }
39      printf("Cost of MST = %d\n", ans);
40  }

```

খেয়াল কর আমরা কিন্তু এই দুই অ্যালগরিদমেই গ্রীডি উপায়ে সবচেয়ে কম খরচের ভার্টেন্স বা বাহু নির্বাচন করে পুরো সমস্যা সমাধান করে ফেলেছি। তোমাদের যদি এই অ্যালগরিদম দুটির কোনোটিতে সন্দেহ হয় তা হলে প্রমাণ করে দেখতে পার কেন এই গ্রীডিভাবে বাহু নির্বাচন করলে সঠিক উত্তর দিবে। চাইলে কোনো ভালো অ্যালগরিদম বইয়েও প্রমাণ দেখে নিতে পার।

৬.৩ ওয়াশিং মেশিন ও ড্রায়ার

মনে কর তুমি একটি কাপড় কাচার কোম্পানি চালাও। তোমার কাছে কিছু সেট কাপড় আছে এবং সেই সঙ্গে একটি ওয়াশিং মেশিন ও একটি ড্রায়ার আছে। তুমি প্রতিটি সেটকে প্রথমে ওয়াশিং মেশিনে দিবে এবং এর পর ড্রায়ারে দিবে। তুমি কিন্তু প্রথমে ড্রায়ার পরে ওয়াশিং মেশিনে দিতে পারবে না। এখন প্রতি সেটের জন্য তোমার জানা আছে যে সেটি ওয়াশিং মেশিনে কত সময় নিবে এবং ড্রায়ারে কত সময় নিবে। অবশ্যই একই সঙ্গে কয়েক সেট কাপড় তুমি একই মেশিনে দিতে পারবে না কিন্তু একই সঙ্গে দুই সেট কাপড় আলাদাভাবে দুই মেশিনে দিতে পারবে। তুমি কত কম সময়ে সব কাপড় পরিষ্কার করে ফেলতে পারবে?

সমস্যাটি যত না সুন্দর এর সমাধান তার থেকে বেশি সুন্দর। মনে কর i তম সেটের জন্য a_i হলো ওয়াশিং মেশিনে দরকারি সময় আর b_i হলো ড্রায়ারে দরকারি সময়। এখন মনে কর অপটিমাল ক্রমে (optimal order) দুটি পাশাপাশি সেট হলো i আর j . অর্থাৎ তুমি চাচ্ছ যে ঠিক i কাজের পর j কাজ করবে তাহলে এই দুটি 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 হলো 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 সেটের পর করা ভালো। এই দুটি সমীকরণ যদি সত্য হয় তাহলে প্রমাণ করা যায় যে $a_i + \max(b_i, a_k) + b_k \leq a_k + \max(b_k, a_i) + b_i$.

$b_k \leq a_k + \max(b_k, a_i) + b_i$ অর্থাৎ i কাজের পর k কাজ করা ভালো (এটি তোমরা প্রফ বাই কন্ট্রাডিকশন বা proof by contradiction এর মাধ্যমে একটু খেটেখুটে করতে পার)। তাহলে কী দাঁড়াল? আমরা কাজগুলোকে আসলে একটি ক্রমে সাজাতে পারি। তবে কোন কাজের আগে কোন কাজ আসবে সেটি নির্ণয় করার জন্য আমাদের উপরের সমীকরণের সাহায্য নিয়ে দেখতে হবে যে কোন সেট আগে করলে কম সময় নেয়। এভাবে কাজগুলোকে সাজালে আমরা অপটিমাল ক্রম পাব। অর্থাৎ আমাদেরকে কাপড়ের সেটকে সর্ট করতে হবে। যখন দুটি সেটের মধ্যে তুলনা করবে তখন উপরের সমীকরণ ব্যবহার করে বলবে কে ছোট আর কে বড়। শেষ!

এই সমাধান বের করা মোটেও সহজ নয়, সুতরাং তোমরা যদি একবার পড়ে এই সমাধান না বোঝ তাহলে আরও কয়েকবার পড়ে দেখ। দুএকটি উদাহরণ হাতে হাতে করে দেখ।

৬.৪ হাফম্যান কোডিং (Huffman Coding)

হাফম্যান কোডিং (Huffman coding) জানার আগে তোমাদের জানতে হবে প্রিফিক্সবিহীন কোডিং (prefix free coding) কী জিনিস। কোডিং (coding) হলো বর্ণমালার বিভিন্ন বর্ণ বা ক্যারেক্টার (character) কে অন্য কিছু দিয়ে প্রকাশ করা। আমাদের এই সমস্যায় আমরা ইংরেজী বর্ণমালার কিছু ক্যারেক্টারকে 0 আর 1 ব্যবহার করে এক একটি সংখ্যা দিয়ে প্রকাশ করব। যেমন আমরা হয়তো a কে প্রকাশ করব 001 দিয়ে, b কে প্রকাশ করব 110 দিয়ে ইত্যাদি। প্রিফিক্সবিহীন কোডিং হলো কোনো কোডই অপর কোডের প্রিফিক্স (prefix) হতে পারবে না। প্রিফিক্স মানে হলো শুরুর অংশ। যেমন 01 হলো 0110 এর একটি প্রিফিক্স। সুতরাং এই দুটি একই সঙ্গে কোড হতে পারবে না। মজার ব্যাপার হলো এরকম 0 আর 1 দিয়ে প্রকাশিত যেকোনো কোডিং তুমি চাইলে একটি বাইনারি ট্রি দিয়ে প্রকাশ করতে পার। মনে কর কোনো নোড থেকে বাম দিকে যেই বাহু যায় তার লেভেল হলো 0 আর ডান দিকে যেই বাহু যায় তার লেভেল হলো 1. তাহলে রুট থেকে শুরু করে 0 আর 1 অনুসারে ডানে বামে যাও এবং কোডের শেষ মাথায় এলে সেই নোডকে মার্ক করে ফেল। এটিই হলো আমাদের কোডিং এর বাইনারি ট্রি তে উপস্থাপন। তাহলে একটু চিন্তা করে দেখ তো প্রিফিক্সবিহীন কোডিংয়ের উপস্থাপন কেমন হবে? আগের মতোই হবে শুধুমাত্র একটি অতিরিক্ত বৈশিষ্ট্য থাকবে আর তাহলো মার্ক করা নোডগুলো কেউ কারও অ্যানসেস্টর (ancestor) বা প্রিডিসেসর (predecessor) হবে না। কোনো কোডিং যদি প্রিফিক্সবিহীন হয় তাহলে কী সুবিধা? সুবিধা হলো তুমি কোনো space ছাড়াই তাদের ডিকোডিং (decoding) করতে পারবে। একটি উদাহরণ দেওয়া যাক, মনে কর $a = 01, b = 0, c = 1$. এখানে কোডিং কিন্তু প্রিফিক্সবিহীন না। এখন যদি তোমাদের 01 দেয় তাহলে ডিকোডিং করলে কী হবে? bc নাকি a ? দুটিই হতে পারে। কিন্তু যদি প্রিফিক্সবিহীন হয় তাহলে কিন্তু কোনোই মাথা ব্যাথা নেই, একে একভাবেই ডিকোডিং করা যায় আর ডিকোডিং করাও খুব সহজ, তুমি ট্রি এর রুট থেকে traverse করতে থাকবে যতক্ষণ না কোনো মার্ক করা নোডে না পৌছাও। এরকম কোনো নোড পেলে তুমি সেই ক্যারেক্টার লিখে রেখে আবার রুট থেকে traverse করা শুরু করবে। এভাবেই তুমি ডিকোডিং করতে পারবে।

যাই হোক এখন মূল সমস্যায় আসা যাক। তোমাকে কিছু ক্যারেক্টার দেওয়া থাকবে এবং

সেই ক্যারেন্টারগুলো কত বার করে একটি লেখাতে আসবে তা বলা আছে। তোমাকে এই ক্যারেন্টারগুলোর এমন একটি প্রিফিক্সবিহীন কোডিং বের করতে হবে যেন পুরো লেখাটির এর এনকোডিং (encoding) এর পরের দৈর্ঘ্য সবচেয়ে কম হয়। একটি উদাহরণ দেওয়া যাক, মনে কর তোমাকে ৩ টি ক্যারেন্টারের ফ্রিকোয়েন্সি (frequency), অর্থাৎ কোন ক্যারেন্টার কতবার করে এলো, দেওয়া আছে: $(a, 10), (b, 4), (c, 8)$, ধরা যাক আমরা এদের কোডিং করলাম এভাবে: $(a, 01), (b, 1), (c, 00)$ তাহলে এনকোডিং এর পরে লেখাটির দৈর্ঘ্য হবে $10 \times 2 + 4 \times 1 + 8 \times 2 = 40$. এর থেকেও যে ভালো করা সম্ভব সেটি তোমরা বুঝতেই পারছ তাই না? কারণ b মাত্র ৪ বার আছে কিন্তু এর দৈর্ঘ্য ১, অন্য দিকে a আছে ১০ বার কিন্তু এর দৈর্ঘ্য ২! যেই ক্যারেন্টার বেশি সংখ্যকবার এসেছে তাকে যদি ছোট দৈর্ঘ্যের কোড দেই তাহলে কিন্তু আমাদের মোট দৈর্ঘ্য কমে যাবে। তিনটি ক্যারেন্টারের ক্ষেত্রে নাহয় এরকম নানান যুক্তি তর্ক দিয়ে সমাধান করা যায় কিন্তু অনেকগুলো ক্যারেন্টারের জন্য সমাধান কীভাবে করবে? খুব একটা কঠিন না। কিছুক্ষণ আগেই বলেছি যাদের ফ্রিকোয়েন্সি কম তাদের বেশি দৈর্ঘ্যের কোড দেয়া উচিত। অর্থাৎ তাদের নোড প্রিফিক্সবিহীন কোডিংয়ের ট্রি এর একদম নিচে থাকবে। আমরা একটি নোড নিয়ে সবচেয়ে কম ফ্রিকোয়েন্সি ওয়ালা দুটি ক্যারেন্টারকে তার চাইল্ড বানিয়ে দেই আর এই নতুন নোডের ফ্রিকোয়েন্সি দেই ওই দুইটি ক্যারেন্টারের ফ্রিকোয়েন্সির যোগফল। কেন? উদাহরণ দিয়ে বোঝা যাক ব্যাপারটা। উপরের সহজ উদাহরণে আমাদের সবচেয়ে কম ফ্রিকোয়েন্সি ওয়ালা দুটি ক্যারেন্টার হলো b আর c . এখন আমরা এদেরকে একত্র করে একটি নোড বানাবো যার ফ্রিকোয়েন্সি হবে $4 + 8 = 12$. কেন? আমরা এভাবে চিন্তা করতে পারি, এদের ফ্রিকোয়েন্সি যেহেতু অনেক কম তাই এদের সবচেয়ে বড় কোড দিব (তা নাহলে আমরা কিছুক্ষণ আগের মতো কোডকে অদল বদল করতে পারি)। অর্থাৎ এরা ট্রি এর একদম নিচে। অর্থাৎ শেষ ক্যারেন্টার বাদে বাকিটুকুর কোড একই। বা আমরা বলতে পারি বাকিটুকুর ফ্রিকোয়েন্সি তাদের দুজনের ফ্রিকোয়েন্সির যোগফলের সমান। বোঝা যাচ্ছে? আমরা n টি ক্যারেন্টার থেকে এভাবে $n - 1$ টি ক্যারেন্টারে আমাদের সমস্যাকে নামিয়ে ফেললাম। এভাবে একে একে বাকি ক্যারেন্টারগুলোর সবচেয়ে ছোট দুটি ফ্রিকোয়েন্সিকে যোগ করে একটি একটি করে ক্যারেন্টার কমিয়ে ফেলতে পারি। এই অ্যালগরিদম একটি হীপ বা প্রায়োরিটি কিউ বা মাল্টিসেট (multiset) ব্যবহার করে খুব সহজেই $O(n \log n)$ এ করে ফেলা যায়। STL এর priority queue ব্যবহার করে কীভাবে এটা করা যায় তা কোড ৬.৪ এ দেখানো হলো।

কোড ৬.৪: huffman coding.cpp

```

1 #include <vector>
2 #include <queue>
3 #include <functional>
4 using namespace std;
5
6 // there are n characters.
7 // their frequencies are in freq array.

```

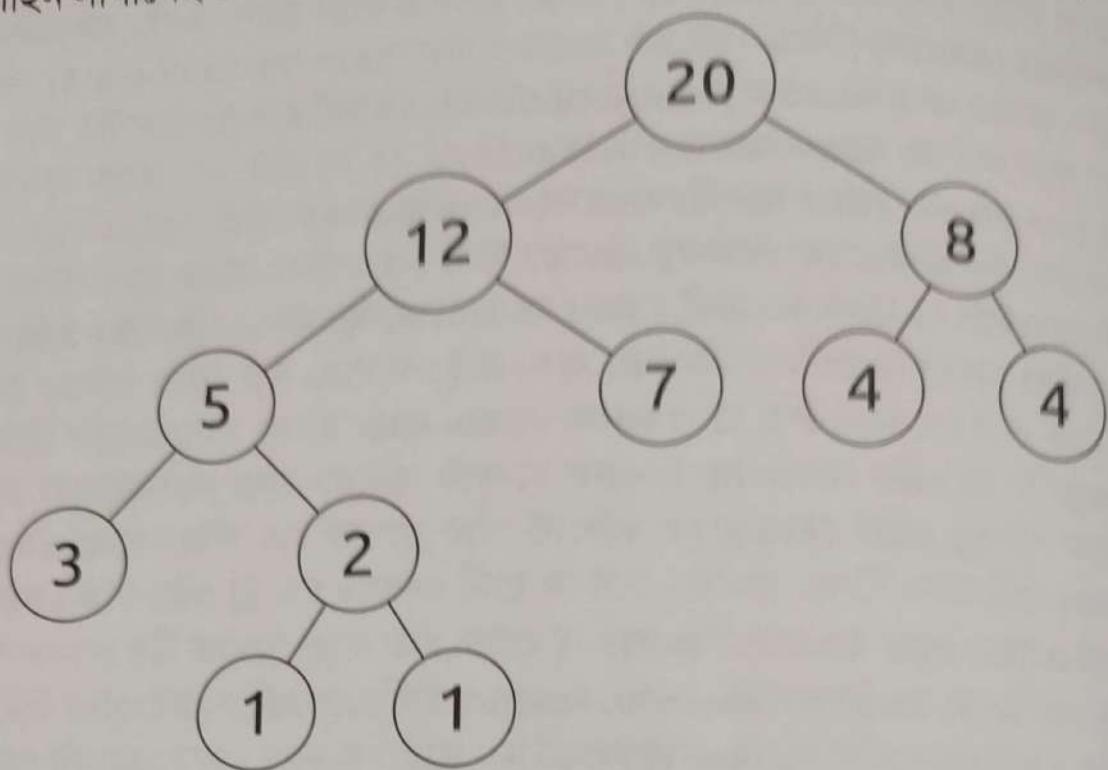
```

8 int n, freq[100];
9
10 int huffman() {
11     // we are using priority-queue.
12     // it is max-heap by default.
13     // so we pass greater<int> parameter
14     // to make it min-heap.
15     priority_queue<int, vector<int>, greater<int>> PQ;
16     for (int i = 0; i < n; i++) {
17         // insert all the frequencies
18         // in the min heap.
19         PQ.push(freq[i]);
20     }
21     while (PQ.size() != 1) {
22         // find minimum element
23         int a = PQ.top();
24         PQ.pop();
25         // find second minimum
26         int b = PQ.top();
27         PQ.pop();
28         // insert their sum.
29         PQ.push(a + b);
30     }
31     return PQ.top();
32 }

```

তোমাদের সুবিধার জন্য আরও একটি বড় উদাহরণ দেখা যাক। মনে কর 6 টি ক্যারেন্টারের ফ্রিকোয়েন্সি দেয়া আছে $(a, 1), (b, 1), (c, 3), (d, 4), (e, 4), (f, 7)$. প্রথমে আমরা সবচেয়ে ছোট দুইটি সংখ্যা নিয়ে তাদের ফ্রিকোয়েন্সি একত্র করব অর্থাৎ 1 আর 1 একত্র হয়ে 2 হবে $(2, 3, 4, 4, 7)$. এবার 2 আর 3 একত্র হয়ে হবে $5(4, 4, 5, 7)$. এর পর হবে $(5, 7, 8)$. এর পর $(8, 12)$ এবং সবশেষে 20. অর্থাৎ আমাদের এই 6 টি ক্যারেন্টারকে প্রিফিক্সবিহীন কোডিং দিয়ে লিখতে সর্বনিম্ন 20 খরচ হবে। তোমরা যেখানে যেখানে নোড একত্র করেছ সেটা আসলে একটি ট্রি আকারে কল্পনা করতে পারো। তাহলে তুমি চিত্র ৬.৩ এর মতো ছবি পাবে। এখন আমরা যদি বাম দিকের বাহুগুলোকে 0 আর ডান দিকের বাহু গুলোকে 1 নাম দেই তাহলে ক্যারেন্টারগুলোর কোড হবে $(a, 0010), (b, 0011), (c, 000), (d, 10), (e, 11), f(01)$. তোমাদের কাজ হলো চিন্তা করা যে কোডের মাধ্যমে এই প্রিফিক্সবিহীন কোডিং বের করা। আশা করি কোড ৬.৪ এর সঙ্গে আরও

৫ – ৬ লাইন লাগালেই তোমরা এই প্রিফিক্স বিহীন কোডিং বের করে ফেলতে পারবে।



নকশা ৬.৩: হাফম্যান ট্ৰি (Huffman tree)

৬.৫ প্ৰোগ্ৰামিং সমস্যা

৬.৫.১ অনুশীলনী

সহজ

- UvaLive 6789 □ UvaLive 6828 □ UvaLive 6829 □ UvaLive 6855 □ UvaLive 6940 □ UvaLive 6957 □ UvaLive 7027 □ UvaLive 7028

সামান্য কঠিন

- UvaLive 6979

অধ্যায় ৭

ডায়নামিক প্রোগ্রামিং (Dynamic Programming)

ডায়নামিক প্রোগ্রামিং (Dynamic Programming) একটি অত্যন্ত গুরুত্বপূর্ণ বিষয় এবং বলা যায় সবচেয়ে কঠিন বিষয় প্রোগ্রামিং প্রতিযোগীতায়। এটিকে কঠিন বলার কারণ হলো এতে ভালো করার এক মাত্র উপায় হলো অনুশীলন করা এবং বেশি বেশি করে এই জাতীয় সমস্যা দেখা। এখানে আসলে শেখানোর তেমন কিছু নেই। এই পদ্ধতির প্রধান বিষয় হলো বড় সমস্যার সমাধান ছোট সমস্যার সমাধান থেকে আসবে!

৭.১ আবারও ফিবোনাচি

মনে কর তোমাকে বলা হলো, এমন কয়টি অ্যারে আছে যার সংখ্যাগুলো 1 বা 2 এবং তাদের যোগফল n হয়। যেমন যদি $n = 4$ হয় তাহলে তুমি মোট 5 ভাবে অ্যারে বানাতে পারবে: $\{1, 1, 1, 1\}$, $\{1, 1, 2\}$, $\{1, 2, 1\}$, $\{2, 1, 1\}$ এবং $\{2, 2\}$. এখন কথা হলো এই সমস্যা কীভাবে সমাধান করব! খেয়াল কর, আমাদের অ্যারের প্রথম সংখ্যা হয় 1 হবে না হলে 2. যদি 1 হয় বাকি অংশটুকু $n - 1$ সংখ্যার ক্ষেত্রে যতভাবে অ্যারে পাওয়া যায় ঠিক ততভাবে 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 সুত্র কাজ করবে যদি $n > 1$ হয়। সেক্ষেত্রে $way(2) = way(1) + way(0)$. $way(1)$ বা $way(0)$ এর মান কিন্তু আমরা এই সুত্র ব্যবহার করে বের করতে পারব না। কারণ আমরা আগেই বলেছি এই সুত্র কাজ করবে যদি $n > 1$ হয়। আমরা এই দুটি মান হাতে হাতে বের করব। $way(1)$ মানে হলো 1 কে আমরা কতভাবে সাজাতে পারব। খুব সহজ, একভাবে আর সেটি হলো: $\{1\}$. অর্থাৎ $way(1) = 1$.

এখন আসা 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 আয়ের যোগফল 0 তাহলে কি খুব একটা ভুল হবে? আচ্ছা তোমাদের অন্যভাবে বোঝানোর চেষ্টা করি। $way(2)$ এর মান কত? 2 তাই না? কারণ: {2} এবং {1, 1} এই দুটি হলো $n = 2$ এর জন্য উভর। 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 হলো? দেখ, তুমি $n = 2$ এর জন্য যদি প্রথমে 1 নাও তাহলে বাকি $2 - 1 = 1$ তুমি একভাবে সাজাতে পারবে কারণ $way(1) = 1$ বা {1}. এখন তুমি যদি সামনে 2 নাও তাহলে কিন্তু বাকি আর কিছু নিতে পারবে না, অর্থাৎ কিছু না নিতে পারা হলো একভাবে নেওয়া!!! অর্থাৎ $way(0) = 1$ বা {}. আমরা আগেই বলে এসেছি (যখন আমরা রিকার্সিভ ফাংশন শিখেছি) যখন আমাদের এরকম সূত্র কাজ করবে না সেটাকে বলা হয় base কেইস। আর $way(n) = way(n - 1) + way(n - 2)$ কে বলা হয় রিকারেন্স (recurrence).

এখন চল এটাকে কোড করি। প্রায় সব DP¹ সমস্যার কোড দুইভাবে করা যায়। ইটারেটিভ উপায়ে (Iteratively) এবং রিকার্সিভ উপায়ে (Recursively). ইটারেটিভ উপায়ে সমাধান করলে কোড দেখতে কিছুটা কোড ৭.১ এর মতো হবে আর রিকার্সিভ উপায়ে করলে কোড ৭.২ এর মতো হবে।

কোড ৭.১: fibIterative.cpp

```

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

```

কোড ৭.২: fibRecursive.cpp

```

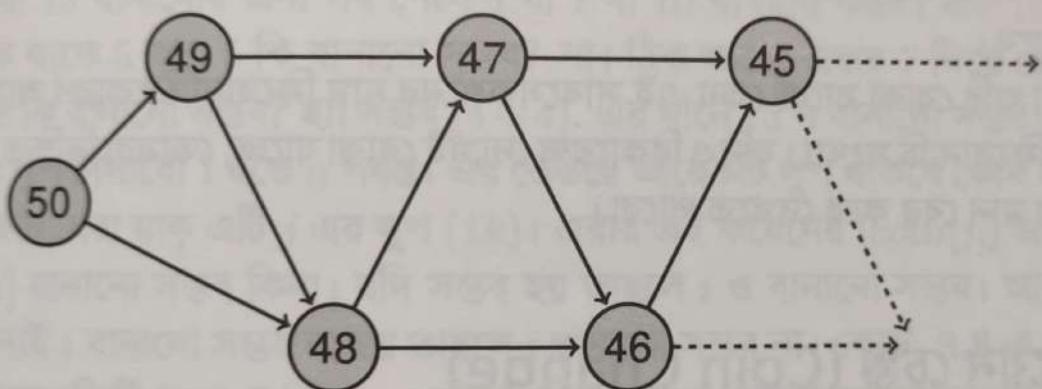
1 int way(int n)
2 {
3     if(n == 0 || n == 1) return 1;
4     return way(n - 1) + way(n - 2);
5 }

```

এবার তোমরা এই দুটি কোড n এর বিভিন্ন মানের জন্য চালিয়ে দেখতো কী হয়! দেখবে ইটারেটিভ সমাধানটি n এর বড় বড় মানের জন্যও ভালো মতোই কাজ করছে কিন্তু রিকার্সিভ ফাংশনের কোড $n = 50$ এর জন্যই অনেক সময় নিয়ে ফেলবে। কেন? একটু চিন্তা করলে দেখবে যে, ইটারেটিভ সমাধানটিতে $way(0)$ হতে $way(n)$ প্রতিটিরই কেবলমাত্র একবার করে

¹ডায়নামিক প্রোগ্রামিং (Dynamic Programming) কে সংক্ষেপে আমরা DP বলে থাকি

মান বের করা হয়। কিন্তু রিকার্সিভ সমাধানটিতে বহুবার করে। একটি উদাহরণ দিয়ে বোঝানো যাক। ধর, আমরা বের করতে চাইছি $way(50)$ তাহলে আমাদের রিকার্সিভ কল হবে $way(49)$ এবং $way(48)$ । আবার $way(49)$ বের করতে $way(48)$ এবং $way(47)$ কল হবে। অর্থাৎ $way(48)$ কিন্তু ইতোমধ্যেই দুবার কল করা হয়ে গেছে। চিত্র ৭.১ দেখলে বুবাবে একেকটি $way(i)$ বহুবার করে কল হয়। যেমন 50 হতে 45 সর্বমোট 8 ভাবে যাওয়া যায় এর মানে $way(45)$ মোট 5 বার কল হবে। যেহেতু কোনো একটি i এর জন্য $way(i)$ বহুবার কল হয় তাই সর্বমোট রানটাইমও অনেক বেশি।^১ এ থেকে বাঁচার উপায় হলো মেমোয়াইজেশন (memoization)। এই পদ্ধতিতে যা করতে হবে তা হলো, কোনো একটি i এর জন্য $way(i)$ বের করতে বললে আগে দেখতে হবে আগেই এই মান বের করা হয়েছে কিনা। যদি হয় তাহলে আগের মানই রিটার্ন করতে হবে। না হলে পুরো মান আমরা বের করব। এটি করার জন্য আমরা একটি অ্যারে নিব। সেই অ্যারেকে আমরা শুরুতেই -1 দিয়ে ইনিশিয়ালাইজেশন (initialization) করে ফেলব। এর পর যখন আমাদের $way(i)$ এর জন্য কল আসবে তখন আমরা অ্যারে এর i তম উপাদান দেখব যে সেখানে -1 আছে কিনা। যদি না থাকে, তাহলে সেই মান রিটার্ন করব। অন্যথায়, আমরা $way(i)$ এর মান বের করব এবং সেই মান অ্যারেতে রেখে দিব পরবর্তীতে ব্যবহার করার জন্য। এই কোডটি ৭.৩ তে দেওয়া হলো।



নকশা ৭.১: ফিবোনাচি রিকার্সিভ কল ট্রি (Fibonacci Recursive Call Tree)

কোড ৭.৩: fibDp.cpp

```

1 int dp[1000]; // initialize to -1.
2
3 int way(int n)
4 {
5     if(n == 0 || n == 1) return 1;
6     if(dp[n] != -1) return dp[n];
7     return dp[n] = way(n - 1) + way(n - 2);
  
```

^১ 50 কল হয় 1 বার, 49 ও 1 বার, 47 হবে 2 বার, 46 হয় 3 বার, 45 হয় 5 বার। 1, 1, 2, 3, 5... পরিচিত লাগে?

আমাদের এই সমস্যায় না, কিন্তু অনেক সময় -1 ও উত্তর হতে পারে। সেক্ষেত্রে অ্যারেকে -1 দিয়ে ইনিশিয়ালাইজেশন করা যাবে না। হয়তো তোমার ফাংশনের রিটার্ন এর মান যেকোনো সংখ্যা হতে পারে সুতরাং তুমি $0, -1, -2$ বা এরকম কোনো সংখ্যা দিয়ে ইনিশিয়ালাইজেশন করতে পারবে না। এরকম অবস্থা হলে তোমরা আরেকটি অ্যারে নিতে পার ধরা যাক সেটির নাম *visited* এবং এখানে 0 ও 1 ব্যবহার করে আমরা কোনো মানের জন্য উত্তর আগেই বের করে নেওয়েছি কিনা তা যাচাই করে দেখতে পারি। সুতরাং এক্ষেত্রে আমাদের এই *visited* এর অ্যারেকে 0 দিয়ে ইনিশিয়ালাইজেশন করতে হবে। আর এই মানের জন্য ফাংশন কল করা হয়েছে বুবাতে সেখানে 1 রাখব। এর থেকে ভালো উপায় হলো একটি অ্যারে *mark* এবং একটি ভ্যারিয়েবল *marker* নেওয়া। প্রতিবার নতুনভাবে DP কল করার আগে *marker* এর মান এক বাড়াবে এবং দেখবে যে *mark* এ *marker* এর সমান মান আছে কিনা। এর উপর ভিত্তি করে তুমি আগের মান রিটার্ন করবে না হয় নতুন করে মান বের করবে। আশা করি এটি বুবেছ যে প্রতিবার DP কল করার আগে মানে প্রতিবার ফাংশন কল করার আগে না, প্রতি টেস্ট কেইসে আর কী। হয়তো তোমার DP ফাংশন n , এর উপর নির্ভর করে যেটা ইনপুটে দেওয়া আছে। তাহলে আমাদের আগের পদ্ধতিতে অ্যারেতে -1 রাখতে হতো বা *visited* কে 0 করতে হতো। সেটি না করে *marker* এর মান এক বাড়িয়ে দিলেই হয়ে যাবে।

আর আশা করি বোৰা যাচ্ছে কেন এই সাবসেকশনের নাম ফিবোনাচি! কারণ আমাদের *way* হলো আসলে ফিবোনাচি সংখ্যা। যদিও রিকারেন্স দেখেই বোৰা যাচ্ছে, তোমরা নিশ্চিত হতে চাইলে কিছু *way* এর মান বের করে দেখতে পারো।

৭.২ কয়েন চেঞ্জ (Coin Change)

এই ধরনের সমস্যার মূল জিনিস হলো, তোমার কাছে কিছু কয়েন আছে ধর 1 টাকা, 2 টাকা, 8 টাকার। তোমাকে একটা পরিমাণ বলা হবে ধরা যাক 50 টাকা। প্রশ্ন হলো তুমি তোমার কাছে থাকা কয়েনগুলো ব্যবহার করে এই টাকা বানাতে পারবে কিনা? পারলে কতভাবে পারবে? আবার যেই কয়েনগুলো দেওয়া আছে সেগুলো কখনো কখনো বলা থাকে যে সেগুলো একবারের বেশি ব্যবহার করতে পারবে না। কখনও কখনও বলা থাকে যে যত খুশি ব্যবহার করতে পারবে আবার কখনো কখনো একটা সীমা বলা থাকে। এই ধরনের সমস্যাগুলোকে আমরা কয়েন চেঞ্জ (coin change) DP বলে থাকি। চল কিছু কয়েন চেঞ্জ DP এর ভ্যারিয়েশন (variation) দেখা যাক।

৭.২.১ Variant 1

তোমাদের কিছু কয়েন দেওয়া আছে এবং প্রতিটি কয়েন তুমি যত বার খুশি ব্যবহার করতে পারবে। মনে কর এই কয়েনগুলো হল: $coin[1 \dots k]$ (মানে $coin[1], coin[2]$ এরকম করে k টি কয়েন আছে)। এখন প্রশ্ন হলো তুমি n বানাতে পারবে কিনা?

ধরা যাক $possible[i]$ হলো i পরিমাণ বানাতে পারব কিনা। যদি পারি তাহলে এর মান হবে 1 আর না পারলে 0. আমাদের বের করতে হবে $possible[n]$. তুমি স্বাভাবিকভাবে চিন্তা কর তুমি যদি হাতে হাতে বের করতে চাইতে যে n বানানো সম্ভব কিনা কীভাবে চিন্তা করলে ভালো হত? যেটা করা যায় তা হলো, $n - coin[1]$ বা $n - coin[2]$ বা ... $n - coin[k]$ এর কোনো একটি যদি বানানো সম্ভব হয় তাহলেই n বানানো সম্ভব না হলে না। অর্থাৎ আমরা বড় একটি মানের জন্য উত্তর বের করতে ছোট মানের সমাধান ব্যবহার করছি। এটিই DP! সূতরাং $1 \dots n$ প্রতিটি মানে গিয়ে তুমি k টি কয়েন একে একে ব্যবহার করে দেখবে যে ছোট মানটি বানানো যায় কিনা। যদি তাদের কোনো একটি বানানো যায় তাহলে এই বড় মানও বানানো যাবে। আর যেহেতু আমরা কোনো মান বানাতে গিয়ে ছোট মান বানানো হয়েছে কি না তা জানতে চাই, সেহেতু আমাদের প্রথমে ছোটগুলোর উত্তর বের করতে হবে এরপর বড়গুলোর। অর্থাৎ কার জন্য উত্তর বের করবা সেই লুপ 1 হতে n পর্যন্ত চালাতে হবে। একটা ছোট উদাহরণ দেয়া যাক। মনে কর তোমার কাছে থাকা কয়েনগুলো হলো $4, 7, 10$. আর মনে কর তুমি বের করতে চাও যে 15 বানানো সম্ভব কিনা। আমি এখন যেই কাজ 15 এর জন্য করব, সেটা 15 এর জন্য করার আগে 1 হতে 14 এর জন্য করে আসতে হবে। যখন 1 হতে 14 পর্যন্ত করা শেষ হবে তখন তুমি 1 হতে 14 প্রতিটি মান এর জন্য বলতে পারবে যে সেই মান বানানো সম্ভব কিনা। এখন 1 হতে 14 এর উত্তর জানলে আমরা 15 এর জন্য উত্তর দিয়ে দিতে পারি। আমরা 15 বানানোর জন্য সব শেষে 4 বা 7 বা 10 ব্যবহার করব। যদি 10 ব্যবহার করি তাহলে বাকি থাকে 5 আর 5 কি বানানো সম্ভব? না। ঠিক আছে, এবার 7 দিয়ে চেষ্টা করা যাক। $15 - 7 = 8$ কি বানানো সম্ভব? হ্যাঁ সম্ভব ($4 + 4$). এর মানে 15 ও বানানো সম্ভব। অর্থাৎ আমরা একটি i এর লুপ চালাবো 1 হতে n পর্যন্ত। এর ভেতরে আরেকটি লুপ থাকবে কোন কয়েন ব্যবহার করব তার জন্য, ধরা যাক এটি j এর লুপ ($1k$)। এবার এই কয়েনের ($coin[j]$) জন্য দেখবো যে $i - coin[j]$ বানানো সম্ভব কিনা। যদি সম্ভব হয় তাহলে i ও বানানো সম্ভব। আর যদি কোনো কয়েনের জন্যই i বানানো সম্ভব না হয় তাহলে i বানানো সম্ভব না। কোড ৭.৪ এ দেওয়া হলো। এর টাইম কমপ্লেক্সিটি হলো $O(nk)$. একটা জিনিস কোডে খেয়াল করলে দেখবে যে এর base কেইস হলো $n = 0$ এবং আমরা বলেছি যে এটি বানানো সম্ভব। কেন? প্রথমত যদি আমরা কোনো কয়েন ব্যবহার না করি তাহলেই 0 বানানো সম্ভব। দ্বিতীয়ত 0 যদি সম্ভব হয় তাহলে দেখো উপরের উদাহরণে 4 এ গিয়ে আমরা চেষ্টা করব যে $4 - 4 = 0$ বানানো সম্ভব কিনা। যদি 0 বানানো সম্ভব না হতো তাহলে আমরা বলতাম যে 4 বানানো সম্ভব না। কিন্তু আমরা তো জানি 4 বানানো সম্ভব। সূতরাং আমাদের বলতে হবে যে 0 ও বানানো সম্ভব।

কোড ৭.৪: variant1.cpp

```

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

```

আশা করি বুঝতে পারছ কিভাবে base কেইস এবং সেক্ষেত্রে উভয় বের করতে হবে? প্রথমত দেখবে যে তোমার সুত্র কোন মানের জন্য কাজ করবে না বা কাজ করানো যায় না। সেসব ক্ষেত্রে তোমাকে চিন্তা করতে হবে base কেইসের উভয় কী হলে বাকিগুলোর মান ঠিক মতো আসবে। এভাবেই তুমি base কেইস আর তার মান বের করতে পারবে।

৭.২.২ Variant 2

তোমাদের কিছু কয়েন দেওয়া আছে এবং প্রতিটি কয়েন তুমি যত বার খুশি ব্যবহার করতে পারবে। বলতে হবে n পরিমাণ তোমরা কতভাবে বানাতে পারবে। এখানে কয়েনের ক্রমে যায় আসে। অর্থাৎ $1 + 3$ আর $3 + 1$ কে আমরা আলাদা বিবেচনা করব। তাহলে তোমাকে যদি 1 আর 2 টাকার কয়েন দেয়া হয় তাহলে তুমি 4 টাকা মোট 5 ভাবে বানাতে পারবে: $1 + 1 + 1 + 1$, $1 + 1 + 2$, $1 + 2 + 1$, $2 + 1 + 1$ এবং $2 + 2$.

ধরা যাক $way[n]$ হলো কতভাবে n বানানো যায়। এখন n বানানোর জন্য তুমি প্রথমে $coin[1]$ ব্যবহার করতে পার বা $coin[2]$ বা প্রদত্ত k টা কয়েনের যেকোনোটি। যদি $coin[1]$ ব্যবহার কর তাহলে বাকি থাকে $n - coin[1]$ পরিমাণ যা তুমি $way[n - coin[1]]$ ভাবে বানাতে পারবে। অর্থাৎ আগের মতোই কিছুটা! আমরা n তৈরি করার জন্য প্রতিটি কয়েন দিয়ে শুরু করব। এর পর দেখব বাকি টুকু কতভাবে বানানো যায়। এই সবগুলো যোগ করলেই তুমি n কতভাবে বানাতে পারবে তা বের করে ফেলতে পারবে। উদাহরণ দেয়া যাক। মনে কর তোমার কাছে কয়েন আছে 2 আর 3 . আমরা জানি 1 বানানো যায় না; $2, 3, 4(2 + 2)$ একভাবে বানানো যায়। প্রশ্ন হলো: কতভাবে বানানো যায়? আমরা চাইলে 2 দিয়ে শুরু করতে পারি, তাহলে বাকি থাকে 3 আর আমরা জানি 3 একভাবে বানানো যায়। কিন্তু যদি আমরা 3 দিয়ে শুরু করতাম তাহলে বাকি থাকত 2 আর আমরা জানি 2 একভাবেই বানানো যায়। তাহলে মোট 2 ভাবে আমরা 5 বানাতে পারি ($2 + 3, 3 + 2$)। আর আগের মতই 0 কতভাবে বানাতে পারি? এই জিনিসটা একটু চিন্তা করলে বুঝবে যে আমরা 1 ভাবে 0 বানাতে পারি। কোড ৭.৫ এ দেওয়া হলো। এখানে তোমার টাইম কমপ্লেক্সিটি দাঁড়াবে $O(nk)$.

কোড ৭.৫: variant2.cpp

```

১ way[0] = 1
২ for(i = 1; i <= n; i++)
  ৩   for(j = 1; j <= k; j++)
    ৪     if(i >= coin[j])
      ৫       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 তাহলো 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, $n - \text{coin}[i]$ পরিমাণ $i - 1$ পর্যন্ত কয়েন ব্যবহার করে বানানো যায় কিনা। অর্থাৎ আগে আমরা যাচাই করতাম যে $dp[n - \text{coin}[i]]$ সত্য কিনা, এখন আমাদের দেখতে হবে $dp[i - 1][n - \text{coin}[i]]$ সত্য কিনা। আমরা বানাতে পারব কিনা সেটি এখন আর শুধু পরিমাণের উপর নির্ভর করছে না, কত পরিমাণ এবং কোন কয়েন পর্যন্ত ব্যবহার করা হয়েছে এই দুটি জিনিসের উপর নির্ভর করে। আমরা যদি দেখতে চাই যে, n পরিমাণ i পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা তাহলে আমাদের দুটি জিনিস দেখতে হবে তাহলো n পরিমাণ $i - 1$ পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা। আর $n - \text{coin}[i]$ পরিমাণ $i - 1$ পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা। অর্থাৎ আমাদের DP তে এখন দুটি প্যারামিটার (parameter). সুতরাং আমাদের 2D অ্যারে লাগবে এই সমস্যা সমাধান করতে। এই সমাধানে আমাদের টাইম ও মেমোরী উভয় কমপ্লেক্সিটি ই $O(nk)$.

আমরা চাইলে মেমোরী কমপ্লেক্সিটি কমিয়ে $O(n)$ করতে পারি। এজন্য খেয়াল কর, আমরা প্রথম i টি কয়েন ব্যবহার করে কোন কোন পরিমাণ বানাতে পারি সেটি জানার জন্য শুধু আমাদের জানতে হয় প্রথম $i - 1$ টি কয়েন ব্যবহার করে কোন কোন পরিমাণ বানানো যায়। সুতরাং প্রতিবার আমাদের শুধু দুটি সারি (row) লাগে। প্রথম থেকে কয়টি কয়েন ব্যবহার করা হচ্ছে সেটি সারি আর কোন পরিমাণ বানাতে হবে সেটিকে কলাম (column) হিসেবে বিবেচনা করে দেখ। তাহলে বুঝবে যে $dp[j][\dots]$ বানাতে আমাদের শুধু $dp[j - 1][\dots]$ জানলেই হয়। আরও মজার ব্যাপার হলো এই আপডেটের সময় যদি তুমি পরিমাণের উর্ধ্বক্রমে না গিয়ে নিম্নক্রমেও যাও তাহলে কিন্তু দুটি সারি এর দরকার হয় না, একটি হলেই হয়ে যায়। প্রথমত আমরা যখন j তম কয়েনের জন্য i এ এসেছি তখন $dp[i]$ এ $j - 1$ পর্যন্ত কয়েনের জন্য উভর লিখা আছে। সুতরাং আমাদের আগের $dp[j - 1][i]$ আসলে এখন $dp[i]$ তে লিখা। এখন কথা হলো $dp[j - 1][i - \text{coin}[j]]$ কোথায় পাব? চিন্তা করে দেখো $dp[i - \text{coin}[j]]$ ব্যবহার করলে সমস্যা কই? সমস্যা হল, আমরা যদি i এর লুপ 1 হতে n পর্যন্ত চালাই, তাহলে j তম কয়েনের জন্য i এ আসার আগে আমরা $i - \text{coin}[j]$ পার করে এসেছি, এবং হয়তো আগের মানকে আপডেটও করে এসেছি। কিন্তু আপডেট করলে তো $dp[j - 1][i - \text{coin}[j]]$ এর মান $dp[i - \text{coin}[j]]$ তে থাকবে না। উপায় হলো, i এর লুপ 1 হতে n না চালিয়ে n হতে 1 চালাও। তাহলে তুমি যখন i এ আছো তখনও $i - \text{coin}[j]$ এর মান পরিবর্তন হয় নাই। এটাই হলো চালাকি। তোমাদের জন্য এই দুইটিরই কোড ৭.৬ তে দেওয়া হলো।

কোড ৭.৬: variant3.cpp

১ // $O(n^2)$ memory.

```

2   dp[0][0] = 1;
3   for (int j = 1; j <= k; j++) {
4       for (int i = 1; i <= n; i++) {
5           if (dp[j - 1][i] ||
6               (i >= coin[j] && dp[j - 1][i - coin[j]])) {
7                   dp[j][i] = 1;
8               }
9           }
10      }
11
12 // O(n) memory.
13 dp[0] = 1;
14 for (int j = 1; j <= k; j++) {
15     for (int i = n; i >= 1; i--) {
16         if (i >= coin[j] && dp[i - coin[j]]) {
17             dp[i] = 1;
18         }
19     }
20 }

```

৭.২.৪ Variant 4

বুরতেই পারছ আমরা variant 3 এর জন্য জানতে চাইব কতভাবে বানানো সম্ভব! আমরা variant 1 কে পরিবর্তন করে variant 2 যেভাবে সমাধান করেছিলাম, variant 3 কে একইভাবে পরিবর্তন করে variant 4 সমাধান করা সম্ভব।

৭.২.৫ Variant 5

আমরা variant 2 তে $1 + 2 + 1$ এবং $2 + 1 + 1$ কে আলাদা ভেবেছিলাম এবং সেজন 1 আর 2 কয়েন দিয়ে 4 বানানো সম্ভব ছিল 5 ভাবে ($1 + 1 + 1 + 1, 1 + 1 + 2, 1 + 2 + 1, 2 + 1 + 1, 2 + 2$). কিন্তু যদি আলাদা না হয়? অর্থাৎ যদি 4 এর ক্ষেত্রে উত্তর হয় 3 ভাবে ($1 + 1 + 1 + 1, 1 + 1 + 2, 2 + 2$)? ধরা যাক $way[n][i]$ হলো প্রথম i টি কয়েন ব্যবহার করে "কে কতভাবে বানানো যায়। এখন মনে কর আমরা জানি যে $i - 1$ পর্যন্ত কয়েন দিয়ে প্রতিটি সংখ্যা কতভাবে বানানো যায়। আমরা জানতে চাই যে যদি i তম কয়েনও ব্যবহার করি তাহলে প্রতিটি সংখ্যা কতভাবে বানানো যায়। একটি উপায় হলো প্রতিটি $way[n][i]$ এ যাওয়া এবং একটি লুপ

চালানো যে আমরা i তম কয়েন কত বার ব্যবহার করব। ধরা যাক t সংখ্যক বার। তাহলে i তম কয়েনকে t সংখ্যক বার ব্যবহার করে (আর বাকিটুকু $i - 1$ কয়েন দিয়ে) n কতভাবে বানানো যায়? $way[n - coin[i] * t][i - 1]$. অর্থাৎ আমরা প্রতি $way[n][i]$ এ গিয়ে t এর একটি লুপ চালিয়ে বের করে ফেলতে পারি এর মান কত। কিন্তু এতে আমাদের টাইম কমপ্লেক্সিটি থার্য $O(n^2k)$ এর মত হয়। একে কমানোর জন্য আমরা কি করতে পারি?

প্রথমত দেখো, আমরা variant 2 এ প্রতি সংখ্যায় গিয়েছি এবং এর পর বিভিন্ন কয়েন ব্যবহার করেছি। অর্থাৎ 1 এ গিয়ে বিভিন্ন কয়েন, 2 এ গিয়ে বিভিন্ন কয়েন, 3 এ গিয়ে বিভিন্ন কয়েন এরকম। কিন্তু এটা করা যাবে না এবার। কারণ ধর 10 এ গিয়ে তুমি $coin[C]$ ব্যবহার করেছ হয়তো 20 এ এসে $coin[C + 1]$ আবার 30 এ এসে $coin[C]$, অর্থাৎ $C, C + 1, C$ আর $C, C, C + 1$ এসব আলাদা আলাদা করে গণনা করা হবে। কয়েন ব্যবহারের মাঝে এখানে কোনো নীতি মেনে চলা হচ্ছে না। এটা দূর করার উপায় হলো আমরা একটা কয়েন নিবো, এর পর প্রতিটি সংখ্যায় গিয়ে তাকে বানানোর চেষ্টা করব। অর্থাৎ কিছুটা variant 3 এর মতো। Variant 3 এ আমরা একটি কয়েন একাধিক বার ব্যবহার করতে পারতাম না, কিন্তু এখানে পারব তবে পর পর ব্যবহার করতে হবে, এটাই পার্থক্য। এখন খেয়াল কর, আমরা i এর লুপ সেখানে পেছন থেকে চালিয়েছিলাম যাতে একটা কয়েন একবারের বেশি ব্যবহার না করতে হয়। একটু চিন্তা করে দেখতো সামনে থেকেই লুপ চালালে কী হতো? তাহলেই কিন্তু আমাদের variant 5 সমাধান হয়ে যায়। তাহলে variant 1 আর variant 5 এর মাঝে পার্থক্য কোথায়? পার্থক্য হলো i এর লুপ আগে নাকি j এর লুপ আগে। এর উপর নির্ভর করছে তুমি $1 + 2 + 1$ আর $1 + 1 + 2$ কে একই ধরছ নাকি আলাদা ধরছ। আমাদের টাইম কমপ্লেক্সিটি হলো $O(nk)$ আর মেমোরী কমপ্লেক্সিটি $O(n)$.

৭.৩ ট্রাভেলিং সেলসম্যান সমস্যা (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 দূরত্ব তুমি জানো। তুমি প্রথমে গিয়ে তোমার সবচেয়ে ভালো বন্ধু 1 এর বাসায় যাবে, এর পর একে একে সবার বাসা ঘুরে আবারও 1 এর বাসায় ফেরত আসবে। সবচেয়ে কম মোট কত দূরত্ব অতিক্রম করে তুমি সবার বাসা ঘুরতে পারবে? এটি হলো ট্রাভেলিং সেলসম্যান সমস্যা (Travelling Salesman Problem). আমরা এতক্ষণ একটি সমস্যাকে DP উপায়ে সমাধান করার জন্য যা করেছি তাহলো বড় একটি সমস্যাকে ছেট সমস্যা দিয়ে সমাধান করেছি। আরেকটি উপায় হলো একই রকম জিনিস খুঁজে বের করা। যেমন আমাদের এই সমস্যার ক্ষেত্রে খেয়াল কর, তুমি মনে কর $1 - 2 - 3 - 4$ এভাবে চার জন বন্ধুর বাসা ঘুরেছ বাকি আছে $5 \dots n$ বন্ধুরা। এই বাকি বন্ধুদের বাসা ঘুরতে তোমার যেই সবচেয়ে কম খরচ সেটি $1 - 3 - 2 - 4$ ঘোরার পর বাকি বন্ধুদের বাসা ঘুরে ফেলার জন্য সবচেয়ে কম খরচের সমান। অর্থাৎ, কোনো এক সময় তোমাকে শুধু জানতে হবে তুমি কোন কোন বন্ধুর বাসা ঘুরে ফেলেছ এবং এখন তুমি কোথায় আছ। বিভিন্নভাবে আমরা একই দশা বা স্টেট (state) এ

আসতে পারি যেমন উপরের উদাহরণে আমরা প্রথম চার জন বন্ধুর বাসা দুভাবে ঘুরে এখন 4 এর বাসায় আছি। অর্থাৎ তোমার স্টেট হলো তুমি কার কার বাসা ঘুরে ফেলেছ (1, 2, 3, 4) আর এখন কোথায় আছি সেটি শুধু একটি সংখ্যা, কিন্তু তুমি কোথায় কোথায় ঘুরে ফেলেছ এই জিনিস অনেকগুলো সংখ্যার সেট। আমরা DP এর সময় স্টেটকে অ্যারের প্যারামিটার হিসেবে লিখি। এক্ষেত্রে আমরা একটি সেটকে কীভাবে সংখ্যা আকারে লিখতে পারি? খেয়াল কর, আমাদের মোট n জন বন্ধু আছে, কার কার বাসায় গিয়েছি তাদের 1 আর কার কার বাসায় এখনও যাওয়া হয়নি তাদের 0 দিয়ে লিখতে পারি। তাহলে n টি 0 – 1 দিয়ে আমরা কার কার বাসায় গিয়েছি সেটি বানিয়ে ফেলতে পারি। কিন্তু তুমি যদি অ্যারের মাত্রা বা ডাইমেনশন (dimension) n নিতে চাও তাহলে নিচয়ই কোড করা খুব একটা সুখকর হবে না? এখানে একটি মজার চালাকি আছে তাহলো তুমি এই 0 – 1 সংখ্যাকে বাইনারি ফর্মে ভাবতে পার। যেমন: তোমার যদি 1, 2, 4 নম্বর বন্ধুর বাসা ঘোরা হয়ে থাকে তাহলে তোমার নম্বর হবে: 0000...1011 = 7. এখন এই সংখ্যা কত বড় হতে পারে? 2^n কারণ একটি বন্ধু থাকতে পারে নাও পারে। যেহেতু আমাদের মোট n জন বন্ধু তাই এই সংখ্যা 2^n রকম হতে পারে। তাহলে আমাদের পুরো স্টেট কত বড়? $n \times 2^n$ এবং প্রতি স্টেটে গিয়ে তুমি অন্যান্য সবার বাসায় যাওয়ার চেষ্টা করবে (n ভাবে)। সুতরাং আমাদের টাইম কমপ্লেক্সিটি হবে $O(n^2 2^n)$. কোড ৭.৭ এ দেওয়া হলো।

কোড ৭.৭: tsp.cpp

```

1 // Assuming that there are 20 friends
2 int dp[1<<20][20];
3 // mask = friends I already visited
4 // at = last visited friend
5 int DP(int mask, int at)
6 {
7     // I used reference. It helps me not
8     // writing dp[mask][at] again and again.
9     int& ret = dp[mask][at];
10    // Assume that we initialized dp with -1
11    if (ret != -1) return ret;
12
13    // initialize ret with infinity
14    ret = 1000000000;
15    // n = number of friend
16    // dist contains distance between every two nodes
17    for (int i = 0; i < n; i++)
18        if (!(mask & (1<<i)))

```

```

19     // mask | (1<<i) turns on i'th bit in mask.
20     ret = MIN(ret,
21         DP(mask | (1<<i), i) + dist[at][i]));
22
23     return ret;
24 }

```

৭.৮ দীর্ঘতম ক্রমবর্ধমান সাবসিকোয়েন্স (Longest Increasing Subsequence)

সংখ্যার একটি ধারা বা সিকোয়েন্স (sequence) আছে। এই সিকোয়েন্স থেকে কিছু সংখ্যা (হয়তো একটিও না) মুছে ফেলতে হবে যেন বাকি সংখ্যাগুলো উর্ধ্বক্রম (increasing order) এ থাকে। আমাদের লক্ষ্য হলো সবচেয়ে দীর্ঘ ক্রমবর্ধমান সাবসিকোয়েন্স (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 আমাদেরকে n টি সংখ্যা দেওয়া আছে। এর LIS (Longest Increasing Subsequence) বের করতে হবে। আমরা যদি প্রথম $n - 1$ টির LIS জানি তাহলে কি কোনো লাভ আছে? চিন্তা করে দেখা যাক। আমরা n তম সংখ্যা কে কার পেছনে বসাব? এমন একটি সংখ্যার পেছনে যা n তম সংখ্যার থেকে ছোট, ধরা যাক $a[i]$ (a হলো মূল সিকোয়েন্স এবং $i < n$). এখন এরকম তো অনেক $a[i]$ আছে যেন $a[i] < a[n]$ কিন্তু কোনটির পেছনে? যে সবচেয়ে ছোট তার পেছনে? না, কারণ: 1, 2, 3, 4 এদের মধ্যে কার পেছনে আমরা 5 কে বসাতে চাইব? নিশ্চয় 1 না। আমরা 4 এর পেছনে বসাতে চাইব কারণ প্রথমত 5 এর থেকে 4 ছোট আর দ্বিতীয়ত এই 4 পর্যন্তই সবচেয়ে বড় LIS আছে। সুতরাং n তম সংখ্যাকে নিয়ে প্রথম n টি সংখ্যার LIS হল, $LIS[i] + 1$ এর মধ্যে সবচেয়ে বড় মান যেখানে $a[i] < a[n]$ । এই পদ্ধতির টাইম কমপ্লেক্সিটি $O(n^2)$. আমরা খুব সহজেই সেগমেন্ট ট্রি (segment tree) ব্যবহার করে এটিকে $O(n \log n)$ করতে পারি। আমরা n এ এসে $1 \dots a[n] - 1$ পর্যন্ত কুয়েরি (query) করব আর শেষে $a[n]$ এ আপডেট করব।

তবে সাধারণত আমরা অন্য আরেকভাবে $O(n \log n)$ এ LIS বের করে থাকি। মনে করা যাক আমাদের ইনপুটের অ্যারে হলো a আর আমাদের কাছে একটি অ্যারে আছে তা ধরা যাক b . প্রথমে b ফাঁকা থাকবে। এখন আমরা যা করব তাহলো a এর শুরু থেকে শেষ পর্যন্ত যাব আর এই সংখ্যাগুলো নিয়ে কিছু একটা কাজ করব। কাজটি সংক্ষেপে বললে বলতে হয় যে b তে আমাদের বর্তমান সংখ্যা $a[i]$ কে এমন জায়গায় বসাব যেন b সর্টেড (sorted) থাকে। যদি আমরা $a[i]$ নিয়ে দেখি আমাদের

^১একটি সিকোয়েন্স থেকে কিছু সংখ্যা মুছে ফেললে যা বাকি থাকে তাই সাবসিকোয়েন্স (subsequence). এখানে কিন্তু বাকি সংখ্যাগুলোর ক্রম একই থাকতে হবে। যেমন 1, 2 হলো 1, 2, 3 এর একটি সাবসিকোয়েন্স কিন্তু 2, 1 কিন্তু না।

b ফাঁকা (এটি কেবলমাত্র প্রথম সংখ্যার ক্ষেত্রে ঘটতে পারে) তাহলে তো এই $a[i]$ কে b তে ঢুকিয়ে দিব। আর যদি তা না হয় তাহলে আমরা b এর ভেতরে সবচেয়ে ছোট এমন একটি সংখ্যা খুঁজে বের করব যেন সেই সংখ্যাটি আমাদের সংখ্যার সমান বা বড় হয়। সেই সংখ্যার জায়গায় আমরা আমাদের সংখ্যাকে বসিয়ে দেব। আর যদি সেরকম কোনো সংখ্যা না পাওয়া যায় তাহলে b এর শেষে আমাদের সংখ্যাকে ঢুকিয়ে দেব। কিছুটা ইনসার্শন সর্ট (insertion sort) এর মতো চিন্তা করতে পার তবে মূল পার্থক্য হলো এই নতুন $a[i]$ কিন্তু ইনসার্ট হবে না, বরং প্রতিস্থাপিত হবে। তবে যদি b তে থাকা সব সংখ্যা আমাদের বর্তমান সংখ্যা হতে ছোট হয় তাহলে আমাদের সংখ্যাকে b এর শেষে ইনসার্ট করব। কিছু উদাহরণ দেওয়া যাক।

যদি $a[i] = 10$ হয় এবং $b = 2, 4, 8, 12, 15$ হয় তাহলে 10 কে বসাতে হবে 12 তে, কারণ এটি সবচেয়ে ছোট সংখ্যা যা 10 এর থেকে বড়। সুতরাং আমাদের b পরিবর্তন হয়ে দাঁড়াবে 2, 4, 8, 10, 15। এখন মনে করো $a[i] = 18$. তাহলে কই বসাবে? উন্নত সহজ 15 এর পরে অর্থাৎ b হয়ে যাবে 2, 4, 8, 10, 15, 18. এখন যদি $a[i] = 1$ হয় তাহলে b পরিবর্তন হয়ে দাঁড়াবে 1, 4, 8, 10, 15, 18. এভাবে আমাদের b এর অ্যারে পরিবর্তন হতে থাকবে। প্রথম হতে শেষ সব a নিয়ে কাজ হয়ে গেলে b এর দৈর্ঘ্যই হবে আমাদের LIS এর দৈর্ঘ্য। কিন্তু এটা ভেবে বসো না যে b হবে LIS. যেমন $a = 2, 3, 1$ হলে সবশেষে b হবে 1, 3. কিন্তু 1, 3 কিন্তু LIS না, তাহলে আমরা পুরো সিকোয়েন্স বের করতে চাইলে কী করতে হবে? খুব সহজ, যখন কোনো একটি সংখ্যা বসাবে তখন তার ঠিক আগের সংখ্যা কত তা লিখে রাখবে। আসলে সেই সংখ্যাটি না লিখতে হবে সেই সংখ্যা a তে যেখানে আছে সেই ইনডেক্স (index). অনেক সময় প্রবলেম ভেদে খেয়াল রাখতে হয় যে স্ট্রিটলি ইনক্রিজিং (strictly increasing) চেয়েছে নাকি নন-ডিক্রিজিং (non-decreasing) চেয়েছে। স্ট্রিটলি ইনক্রিজিং হলো 1, 5, 6, 7 এরকম আর নন-ডিক্রিজিং হলো 1, 2, 2, 3, 3, 3, 4, 5, 5 এরকম। এক্ষেত্রেও সমাধানের মূল আইডিয়া একই থাকবে।

এখন কথা হলো এই অ্যালগরিদমের টাইম কমপ্লেক্সিটি কত? প্রথমত আমাদের n বার কাজ করতে হচ্ছে। যদি আমরা লুপ চালাই b এর অ্যারেতে তাহলে প্রতিবার n সময় লাগবে। কিন্তু আমরা যদি লুপ না চালিয়ে বাইনারি সার্চ করি তাহলেই কিন্তু আমরা এই কাজ $O(\log n)$ সময়ে করতে পারি। এবং এভাবে আমাদের রানটাইম হবে $O(n \log n)$. এখন একটি প্রশ্ন করি, আমরা ইনসার্শন সর্টে তাহলে বাইনারি সার্চ চালিয়ে রানটাইম কমালাম না কেন? কারণ হলো, ইনসার্শন সর্টে এ আমাদের শুধু সঠিক জায়গা খুঁজে বের করলেই চলবে না তাকে ইনসার্ট করতে হবে। প্রতিস্থাপন বা replace করতে কিন্তু $O(1)$ কাজের প্রয়োজন হয় কিন্তু ইনসার্ট করতে worst কেইসে $O(n)$ পরিমাণ কাজ করতে হতে পারে। আবার তোমরা যদি মনে কর লিঙ্কড লিস্ট ব্যবহার করে ইনসার্ট তুমি $O(1)$ সময়ে করতে পার, তাহলে সমস্যা হলো লিঙ্কড লিস্টে তুমি বাইনারি সার্চ করতে পারবে না। কারণ বাইনারি সার্চ করতে হলে একটি নির্দিষ্ট ইনডেক্সে যাওয়ার দরকার হয় যা লিঙ্কড লিস্টে সন্তুষ্ট না। একটু চিন্তা করলে অবশ্য নিজে থেকে কষ্ট করে বাইনারি সার্চ করতে হবে না, set আর upper bound ব্যবহার করলেই হবে। কোড ৭.৮ এ এর কোড দেওয়া হলো।

```

1 #include <stdio.h>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main() {
7     vector<int> v;
8     int n, num;
9     scanf("%d", &n);
10    for (int i = 0; i < n; i++) {
11        scanf("%d", &num);
12        // use upper_bound if you want
13        // longest non-decreasing
14        // subsequence.
15        vector<int>::iterator iv =
16            lower_bound(v.begin(), v.end(), num);
17        if (v.end() == iv) v.push_back(num);
18        else v[iv - v.begin()] = num;
19    }
20    printf("Lis = %d\n", v.size());
21 }

```

আমরা মনে হয় আগে কোথাও upper bound আর lower bound নিয়ে দেখি নাই। এই সুযোগে এদের নিয়ে কথা বলা যাক। এই দুটিই বাইনারি সার্চের মতো। মনে কর V একটি স্টেড ভেস্ট্র (vector) বা সেট (set)। তাহলে আমরা লিখতে পারি $lower_bound(V.begin(), V.end(), x) - V.begin()$ বা V যদি একটি স্টেড অ্যারে হয় তাহলে $lower_bound(V, V + n, x) - V$ যেখানে x হলো আমরা যা খুঁজছি। এই দুটি মান আসলে কী দেয়? দেয় আমরা যা খুঁজছি তার ইনডেক্স। এখন কথা হলো lower bound আর upper bound এর মাঝে পার্থক্য কী? প্রথমত upper bound হলো x এর থেকে বড় সবচেয়ে ছোট সংখ্যার ইনডেক্স। যেমন V যদি হয় $\{1, 2, 4, 4, 6\}$ আর $x = 5$ তাহলে upper bound আমাদের 6 এর ইনডেক্স দিবে। যদি $x = -10$ হতো তাহলে আমাদের 1 এর ইনডেক্স দিত। lower bound একটু আলাদা। এটি x এর সমান বা বড় প্রথম সংখ্যার ইনডেক্স দিবে। উপরের $x = 5$ এর ক্ষেত্রে lower bound আমাদের 6 এর ইনডেক্স দেবে তবে $x = 4$ হলে আমাদের প্রথম 4 এর ইনডেক্স দেবে। যদিও বোঝার সুবিধার জন্য বললাম যে lower bound বা upper bound উভয়েই ইনডেক্স দেয় কিন্তু কথাটা পুরোপুরি ঠিক নয়। অ্যারের ক্ষেত্রে এরা ওই স্থানের

পয়েন্টার (pointer) আর ভেষ্টের বা সেটের ক্ষেত্রে ওই স্থানের ইটারেটর (iterator) দেবে। তোমরা যদি ইনডেক্স চাও উপরে দেখানো উপায়ে V বা $V.begin()$ বিয়োগ করতে হবে। তবে হ্যাঁ, সেটের ক্ষেত্রে এই বিয়োগ কাজ করবে না, এই বিয়োগ কেবল মাত্র অ্যারে আর ভেষ্টের ক্ষেত্রে কাজ করবে। আরও একটি জিনিস, উপরের উদাহরণে যদি $x = 7$ হতো? অর্থাৎ যদি V তে উভর না থাকতো তাহলে? সেক্ষেত্রে অ্যারের ক্ষেত্রে এটি n ইনডেক্স দেবে (অ্যারের দৈর্ঘ্য) আর ভেষ্টের বা সেটের ক্ষেত্রে $V.end()$.

৭.৫ দীর্ঘতম সাধারণ সাবসিকোয়েন্স (Longest Common Subsequence)

দুটি স্ট্রিং S এবং T দেওয়া থাকবে, আমাদের এমন একটি স্ট্রিং বের করতে হবে যা S এবং T উভয়েরই সাবসিকোয়েন্স হয় এবং দীর্ঘতম হয়। এই সমস্যার ক্ষেত্রে আমাদের স্টেট হবে: যদি আমাদের S এবং T সম্পূর্ণভাবে না দিয়ে S এর প্রথম s সংখ্যক এবং T এর প্রথম t সংখ্যক ক্যারেক্টার দেওয়া হয় তাহলে Longest Common Subsequence (LCS) কত? যদি $S[s] = T[t]$ হয় (১ ইনডেক্সিং ধরে) তাহলে কিন্তু আমাদের উভর হলো S এর প্রথম $s - 1$ এবং T এর $t - 1$ এর যত উভর তার থেকে এক বেশি। আর যদি $S[s] \neq T[t]$ হয় তাহলে S এর প্রথম $s - 1$ ও T এর প্রথম t ক্যারেক্টারের ক্ষেত্রে উভরের মধ্যে যেটি বড় সেটি। এখন যদি আমাদের শুধু উভয়ের দৈর্ঘ্য নয় সেরকম একটি স্ট্রিংও প্রিন্ট করতে বলে তাহলে আমরা প্রথমে DP টেবিল অর্থাৎ উভয়ের ছক বানিয়ে নেব। এর পর দুটি স্ট্রিংয়েরই শেষ থেকে আসতে হবে। যদি শেষ ক্যারেক্টার দুটি একই হয় তাহলে আমরা সেটি নেবই। আর না হলে আমরা DP টেবিল থেকে দেখব যে কোন স্ট্রিং থেকে শেষ ক্যারেক্টার বাদ দেওয়া উচিত। খেয়াল করে দেখ, এভাবে করলে সমস্যা হলো আমরা স্ট্রিংটি উল্টো দিক থেকে তৈরি করছি। যেহেতু আমাদের স্ট্রিংকে সামনের দিক থেকে প্রিন্ট করতে হবে সেহেতু হয় আমাদের কোনো একটি স্ট্রিংয়ে ক্যারেক্টারগুলো নিয়ে পরে উল্টো করে বা reverse করে প্রিন্ট করতে হবে অথবা এই পুরো কাজটি রিকার্সিভ উপায়ে করতে হবে। আরেকটি বুদ্ধি হলো আমরা যদি সামনের দিক থেকে DP না করে পেছন থেকে DP করি তাহলে আর স্ট্রিং উল্টো করার বামেলা থাকে না। সাধারণ একটি while লুপ দিয়েই আমরা পুরো স্ট্রিং প্রিন্ট করে ফেলতে পারি। খেয়াল কর, আমি এখানে অনেক উপায়ে স্ট্রিং প্রিন্ট^১ করার পদ্ধতি বললাম, একেক সমস্যার ক্ষেত্রে একেক উপায়ে পাথ প্রিন্ট করা সহজ হয়।

^১যেকোনো DP সমস্যায় শুধু মান না, মানটা কীভাবে হয় সেটও চেয়ে থাকে, একে আমরা পাথ প্রিন্টিং (path printing) বলে থাকি।

৭.৬ ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন (Matrix Chain Multiplication)

যেকোনো দুটি সংখ্যা আমরা চাইলেই গুণ করতে পারি। কিন্তু দুটি ম্যাট্রিক্স কিন্তু চাইলেই গুণ করা যায় না। আমরা $A(p \times q)$ এবং $B(r \times s)$ মাত্রা বা ডাইমেনশন (dimension) এর দুটি ম্যাট্রিক্স গুণ করতে পারব যদি $q = r$ হয়। অর্থাৎ A এর কলাম সংখ্যা যদি B এর সারিসংখ্যার সমান হয় তাহলেই আমরা $A \times B$ করতে পারব ($B \times A$ আর $A \times B$ কিন্তু ম্যাট্রিক্সের ক্ষেত্রে আলাদা কথা)। এখন মনে কর আমাদের অনেকগুলো ম্যাট্রিক্স পর পর আছে: A_1, A_2, \dots, A_n . এদের পর পর গুণ করা যাবে যদি এদের ডাইমেনশন এরকম হয়: $A_1(p_1 \times p_2), A_2(p_2 \times p_3), A_3(p_3 \times p_4) \dots A_n(p_n \times p_{n+1})$. দুটি ম্যাট্রিক্স $A(p \times q)$ এবং $B(q \times r)$ গুণ করার মূল্য বা cost হলো $p \times q \times r$. কেন? কারণ আমাদের এই দুটি ম্যাট্রিক্স গুণ করতে চাইলে তিনটি লুপ p, q এবং r পর্যন্ত চলাতে হবে। এখন একটি জিনিস খেয়াল কর $A_1 \times (A_2 \times A_3)$ এর মূল্য আর $(A_1 \times A_2) \times A_3$ এর মূল্য কিন্তু আলাদা হতে পারে। একটা উদাহরণ দেখা যাক। মনে কর $A_1 = 2 \times 3, A_2 = 3 \times 5$ এবং $A_3 = 5 \times 4$. তাহলে $A_1 \times (A_2 \times A_3)$ এর মূল্য হবে $3 \times 5 \times 4 + 2 \times 3 \times 4 = 60 + 24 = 84$ আর $(A_1 \times A_2) \times A_3$ এর মূল্য হবে $2 \times 3 \times 5 + 2 \times 5 \times 4 = 30 + 40 = 70$. যদি n টি ম্যাট্রিক্স থাকে তাহলে তাদের অনেকভাবে গুণ করা যায় (আরও নির্দিষ্টভাবে বললে এটি কাটালান সংখ্যা (catalan number) এর সমান)। সুতরাং n এর বড় মান, ধরা যাক 100, এর জন্য তুমি সবভাবে ঢেঙ্গ করতে পারবে না। কারণ তাতে অনেক সময় লেগে যাবে। তাহলে উপায় কী? প্রথমত খেয়াল কর তুমি কিন্তু লাফ দিয়ে A_1 এর সঙ্গে A_5 এর গুণ দিতে পারবে না। তোমাকে সবসময় পরপর গুণ করতে হবে। অপটিমাল গুণ কেমন হবে তুমি একটু কল্পনা কর। তুমি পাশাপাশি দুটি দুটি করে গুণ করছ যতক্ষণ না তোমার কাছে একটি ম্যাট্রিক্স বাকি থাকে। তুমি শেষ গুণটা খেয়াল কর। শেষ গুণের সময় ম্যাট্রিক্স দুটি হবে কিছুটা এরকম: $(A_1 \times A_2 \times \dots A_i) \times (A_{i+1} \times \dots A_n)$. অর্থাৎ শুরুর দিকের কিছু ম্যাট্রিক্স একত্রে "কোনভাবে" গুণ হবে, শেষের দিকের বাকি ম্যাট্রিক্স কোনভাবে গুণ হবে এবং এরা দুইজন আবার গুণ হবে। এদের দুজনের গুণের খরচ কিন্তু তুমি জানো কারণ তোমার প্রথম গুণফলের ম্যাট্রিক্সের ডাইমেনশন হবে $p_1 \times p_{i+1}$ এবং দ্বিতীয় গুণফলের ম্যাট্রিক্স এর ডাইমেনশন হবে $p_{i+1} \times p_{n+1}$. সুতরাং এদের গুণের মূল্য হবে $p_1 \times p_{i+1} \times p_{n+1}$. এখন যেটি জানি না তা হলো এই দুটি অংশের মূল্য। এটিই কিন্তু DP. আমরা 1 হতে n পর্যন্ত গুণ করার মূল্য বের করার জন্য ছোট দুটি সীমার মূল্য জানতে চাচ্ছি। সুতরাং আমাদের একটি রিকার্সিভ ফাংশন থাকবে যাকে বলব আমাকে A_i হতে A_j পর্যন্ত সব ম্যাট্রিক্সের গুণফলের মূল্য বল। সে তেতরে যা করবে যাকে বলব আমাকে A_i হতে A_k এবং A_{k+1} হতে A_j পর্যন্ত গুণ করার মূল্যকে রিকার্সিভ উপায়ে বের করবে। এর সঙ্গে ওই দুই অংশের গুণফলের ম্যাট্রিক্স গুণ করার মূল্য যোগ করবে। এভাবে প্রতি $i \leq k < j$ এর জন্য আমাদের মূল্য বের করতে হবে। এই সব মূল্যের মধ্যে যেটি সবচেয়ে কম সেটিই A_i হতে A_j পর্যন্ত অপটিমালভাবে গুণ করার মূল্য। প্রশ্ন হলো base কেইস কী? দুভাবে

^১গুণ করা যে যাবে সেটি নিয়ে কোনো সন্দেহ নেই, কারণ তুমি A_i হতে A_j পর্যন্ত যেভাবেই গুণ কর না কেন এর ডাইমেনশন হবে $p_i \times p_{j+1}$.

চিন্তা করতে পার। এক, তুমি ভেবে দেখ কত ছোট কাজ তুমি এমনিই করে ফেলতে পারবে। সহজ, তোমাকে যদি দুটি ম্যাট্রিক্স দেয় তাহলে তুমি জানো যে তুমি একভাবেই গুণ করতে পারবে আর গুণের মূল্য এত। দুই, তুমি এভাবেও চিন্তা করতে পার যে কত পর্যন্ত ভাগ করা যায়। তোমাকে যদি একটি ম্যাট্রিক্স দেয় অর্থাৎ $i = j$ যদি হয় তাহলে কিন্তু $i \leq k < j$ এই সমীকরণ অনুসারে কোনো k পাবে না অর্থাৎ ভাঙা যাবে না। এখন তোমাকে একটি ম্যাট্রিক্স দিয়ে যদি বলে এদের গুণ করার মূল্য কত? কী উত্তর হবে? 0, তাই না? কারণ এখানে গুণ করার কিছু নেই। এ তো গেল base কেইস। এখন চিন্তা কর এর টাইম কমপ্লেক্সিটি কত? এজন্য দেখ তোমার DP এর প্যারামিটার কয়টি? দুটি, i এবং j অর্থাৎ $O(n^2)$ এর মতো। i, j প্যারামিটার এর জন্য তোমাকে $i \leq k < j$ এর মধ্যে একটি k এর লুপ চালাতে হবে। অর্থাৎ মোট $O(n^3)$. যেহেতু সরাসরি আমরা n পর্যন্ত লুপ চালাচ্ছি না তাই তোমরা ভাবতে পার এটা তো n^3 নাও হতে পারে। যারা বিশ্বাস করছ না তারা একটু হিসাব করলেই দেখতে পারবে যে এটা আসলেই $O(n^3)$. আর হ্যাঁ আশা করি মেমোয়াইজেশনের কথা ভুলোনি। মেমোয়াইজেশন না করলে কিন্তু তোমাদের অ্যালগরিদম আর $O(n^3)$ হবে না বরং এটি হয়ে যায় ব্যাকট্র্যাকিং (backtracking). অর্থাৎ অন্যভাবে বলা যায় ব্যাকট্র্যাকিংয়ে স্টেটকে মেমোয়াইজেশন করলেই DP হয়ে যায়। কিন্তু সমস্যা হলো অনেক সময় এই স্টেট এত বড় হয়ে যায় যে মেমোয়াইজেশন করা অসম্ভব হয়ে যায়।

যাই হোক, এতক্ষণ আমি রিকার্সিভ উপায়ে উত্তর বের করার কথা বললাম। ইটারেটিভ উপায়েও কিন্তু এই সমাধান করা যাবে। তবে এতে i, j এর লুপ না চালিয়ে প্রথমে দৈর্ঘ্যের লুপ। চালাতে হবে এর পর শুরুর মাথার লুপ i . তাহলে শেষ মাথা $j = i + l - 1$. এখন তুমি k এর লুপ চালাবে। কেন আমরা এভাবে করলাম? খেয়াল কর, তুমি যদি প্রথমে i, j এর লুপ চালাতে এবং এর ভেতরে k তাহলে তুমি $dp[i][k]$ এবং $dp[k+1][j]$ এর মান জানতে চাইবে। প্রথমটির মান ইতোমধ্যেই বের করে ফেলেছ কিন্তু i এখনো k পর্যন্ত যায়নি! সুতরাং $dp[k+1][j]$ আমরা এখনো জানি না। তাই এভাবে করলে হবে না। মূল ধারনা হলো ছোট থেকে আসা। তুমি যদি ছোটটির উত্তর জানে তাহলেই বড়টির উত্তর বের করতে পারবে। এখানে ছোট বা বড় কিসের সাপেক্ষে? দৈর্ঘ্য। এ জনাই আমরা আগে দৈর্ঘ্যের লুপ চালিয়েছি।

৭.৭ অপটিমাল বাইনারি সার্চ ট্রি (Optimal Binary Search Tree)

বাইনারি সার্চ ট্রি কী জিনিস তা তো তোমাদের ইতোমধ্যেই বলেছি। এটি এমন একটি ট্রি যার প্রতি নোডে একটি করে সংখ্যা থাকে। তোমাকে কোনো কুয়েরি দিলে সেই নম্বর খুঁজে বের করতে হয়। আমরা সবসময় রঞ্ট থেকে শুরু করি এবং যদি দেখি আমাদের বর্তমানের নোড আমরা যেই সংখ্যা খুঁজছি তার সমান তাহলে তো হয়েই গেল, আর যদি তা না হয় তাহলে দেখব এই নোডের সংখ্যার থেকে আমাদের সংখ্যা ছোট নাকি বড়, ছোট হলে বামে যাব আর বড় হলে ডানে। এভাবে যতক্ষণ না খুঁজে পাচ্ছি ততক্ষণ এই কাজ চলতেই থাকবে। আমাদের এই সমস্যায় মনে করব সবসময় উত্তর খুঁজে পাওয়া যাবে মানে ট্রিতে যেসব সংখ্যা আছে শুধু তাদেরকেই কুয়েরি করা হবে।

যাই হোক, এখন যদি আমাদের যেকোনো কুয়েরি আসার সম্ভাবনা সমান (equi-probable) হয় তাহলে আমাদের ব্যালেন্সড বাইনারি সার্চ ট্রি (balanced binary search tree) বানাতে হবে। কিন্তু যদি কুয়েরিগুলো সমসম্ভাব্য না হয়? যদি আমাদের বলা থাকে কোন কুয়েরি আসার সম্ভাবনা কত তাহলে? এখানে কিন্তু হাফম্যান ট্রি (huffman tree) এর টেকনিক খাটবে না কারণ আমাদের সংখ্যাগুলো অবশ্যই স্টেড আকারে থাকতে হবে ট্রি তে^১। এখন মনে কর তোমার কাছে n টি সংখ্যা আছে 1 হতে n পর্যন্ত আর; তম সংখ্যা কুয়েরি হওয়ার সম্ভাবনা p_i . তাহলে কোনো একটি ট্রির মোট মূল্য বা প্রত্যাশিত মূল্য হবে কোন সংখ্যার কুয়েরি হওয়ার সম্ভাবনা আর সেই সংখ্যার ট্রিতে গভীরতার গুণফলসমূহের যোগফল। আসলে তুমি যদি অন্য অ্যালগরিদমের বই দেখ বা ইন্টারনেটে দেখ তাহলে দেখবে যে অপটিমাল বাইনারি সার্চ ট্রি (optimal binary search tree) সমস্যা এটি না। ওটি আরেকটু জটিল তবে মূল ধারনা একই এবং সমাধানের ধরনও একই। যাই হোক, এখন তুমি চিন্তা কর এই n টি সংখ্যাকে নিয়ে কীভাবে ট্রি বানাবে? অবশ্যই যদি একটি নোড বাকি থাকে তাহলে তাকে নিয়ে ট্রি বানানোর খরচ 0. কিন্তু যদি একাধিক থাকে তাহলে? তাহলে যেসব সংখ্যা বাকি আছে তাদের একটি হবে রুট। ধরা যাক i হতে j পর্যন্ত সংখ্যা বাকি আছে আর আমরা k কে রুট হিসেবে নির্বাচন করলাম যেখানে $i \leq k \leq j$. তাহলে এ জন্য আমাদের মূল্য কত হবে? প্রথমত k এর জন্য মূল্য 0. বাকি $[i, k - 1]$ যাবে বামে আর $[k + 1, j]$ যাবে ডানে। এই configuration এ মূল্য হবে $dp[i, k - 1] + dp[k + 1, j] + (p_i + p_{i+1} + \dots + p_{k-1}) + (p_{k+1} + \dots + p_j)$. কারণ প্রথম দুটি হলো রুট এর দুই পাশের ট্রি অপটিমালভাবে বানানোর মূল্য আর বাকিটুকু হলো এত সম্ভাব্যতায় আমাদের নিচে নামতে হতে পারে বা এত সম্ভাব্যতায় আমাদের আরও 1 মূল্য দিতে হতে পারে। এই সমাধান আমাদের ম্যাট্রিক্স চেইন মাল্টিপ্লিকেশন (matrix chain multiplication) এর মতো। এর টাইম কমপ্লেক্সিটি যে $O(n^3)$ তা নিয়ে কোনো সন্দেহ নেই। তবে খুব ছোট একটি অপটিমাইজেশন করে আমরা একে $O(n^2)$ করে ফেলতে পারি। প্রথমেই বলে নেই এই রকম অপটিমাইজেশন কিন্তু সব প্রবলেমের ক্ষেত্রে খাটবে না। কিছু বিশেষ বৈশিষ্ট্য থাকলেই কেবল হবে। তবে সেই বিশেষ বৈশিষ্ট্যটি কী সেটি আমি নিজেও ভালো মতো বুঝি না। শুধু জানি অন্তত এই সমস্যায় এই অপটিমাইজেশন কাজ করবে। অপটিমাইজেশন টি এখন বলি। মনে কর $[i, j]$ এর জন্য অপটিমাল k হলো $P[i, j]$. তাহলে আমরা লিখতে পারি $P[i, j-1] \leq P[i, j] \leq P[i+1, j]$. অর্থাৎ তুমি যখন $[i, j]$ তে আসবে তখন k এর লুপ i হতে j নাচালিয়ে $P[i, j-1]$ হতে $P[i+1, j]$ চালাবে। এই কাজ করলেই আমাদের রানটাইম $O(n^2)$ এ নেমে আসবে। কেন, কীভাবে- এসবের উত্তর আমি নিজেও খুব ভালো মতো জানি না। সুতরাং যাদের আগ্রহ আছে তারা ইন্টারনেটে ঘেটেযুটে জেনে নিও। এই অপটিমাইজেশন এর একটি নাম আছে আর তা হলো নুথ অপটিমাইজেশন (knuth optimization). ইচ্ছা আছে এই অপটিমাইজেশন এবং এছাড়াও ডায়নামিক প্রোগ্রামিং সম্পর্কিত যেসব অপটিমাইজেশন টেকনিক আছে সেসব নিয়ে প্রবর্তীতে লিখার।

^১স্টেড বলতে কী বোঝাচ্ছি আশা করি বোঝা যাচ্ছে? কঠিনভাবে বলতে গেলে বলতে হয় in-order traversal করলে স্টেড সংখ্যা পাবে।

৭.৮ প্রোগ্রামিং সমস্যা

৭.৮.১ অনুশীলনী

সহজ

- UvaLive 6782 □ UvaLive 6783 □ UvaLive 6787 □ UvaLive 6801 □ UvaLive 6834 □ UvaLive 6848 □ UvaLive 6853 □ UvaLive 6861 □ UvaLive 6892 □ UvaLive 6896 □ UvaLive 6904 □ UvaLive 6908* □ UvaLive 6952** □ UvaLive 6972 □ UvaLive 6980 □ UvaLive 6991 □ UvaLive 6993 □ UvaLive 6997 □ UvaLive 7023 □ UvaLive 7029 □ UvaLive 7030 □ UvaLive 7050 □ UvaLive 7065 □ UvaLive 7143* □ UvaLive 7165

সামান্য কঠিন

- UvaLive 6822 □ UvaLive 6824** □ UvaLive 6840** □ UvaLive 6905 □ UvaLive 6914 □ UvaLive 6923* □ UvaLive 6927 □ UvaLive 6931* □ UvaLive 6932* □ UvaLive 6937 □ UvaLive 6938** □ UvaLive 6961 □ UvaLive 7003 □ UvaLive 7005 □ UvaLive 7055** □ UvaLive 7061 □ UvaLive 7063** □ UvaLive 7078** □ UvaLive 7172*

কঠিন

- UvaLive 7046* □ UvaLive 7167

অধ্যায় ৮

গ্রাফ

আগেই বলে রাখি যেকোনো গ্রাফের অ্যালগরিদম কোড করার সময় STL ব্যবহার করলে কোড অনেক ছোট ও সহজ হয়। সুতরাং আমরা এই অধ্যায়ে যেসব কোড দেখাব তাদের বেশির ভাগেই STL ব্যবহার করা। কোডগুলো বুঝতে STL খুব ভালোভাবে বুঝতে হবে তা না। তুমি শুধু মাথায় রেখ যে আমরা কী কোড করছি আর কোন লাইনে কী করতে চাওয়া হচ্ছে তাহলেই তোমরা STL এর কোডগুলো বুঝতে পারবে। ব্যবহার দেখতে দেখতে কোনো জিনিস শিখলে সে জিনিস মনে থাকে অনেক দিন।

৮.১ ব্রেডথ ফার্স্ট সার্চ (Breadth First Search - BFS)

এটি একটি গ্রাফে ঘুরে বেড়ানোর বা ট্রাভার্স (traverse) করার একটি টেকনিক। মনে কর গ্রাফের s নোড হতে তুমি ঘোরা শুরু করবে। তুমি যা করতে পারো তা হলো, শুরুর নোডের 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 গিয়েছ সেখানে তো আবার যাওয়ার কোনো মানে নেই, তাই কোনো বাহু (edge) দিয়ে অন্য প্রান্তে যাওয়ার সময় দেখবে যে- অন্য প্রান্তে ইতোমধ্যেই গিয়েছ কিনা আর যদি গিয়ে থাকো তাহলে সেখানে যাওয়ার কোনো মানে নেই।

যদি তোমার মোট ভার্টেক্স (vertex) থাকে V টি এবং বাহু থাকে E টি তাহলে আমাদের এই ঘুরে বেড়াতে সময় লাগবে $O(V + E)$. কারণ তোমরা প্রতি নোডে একবারের বেশি যাচ্ছ না আর প্রতি বাহুর ক্ষেত্রে দুই মাথা থেকে তুমি দুবার অন্য মাথায় যাওয়ার চেষ্টা করবে। এর কোড ৮.১ এ দেওয়া হলো।

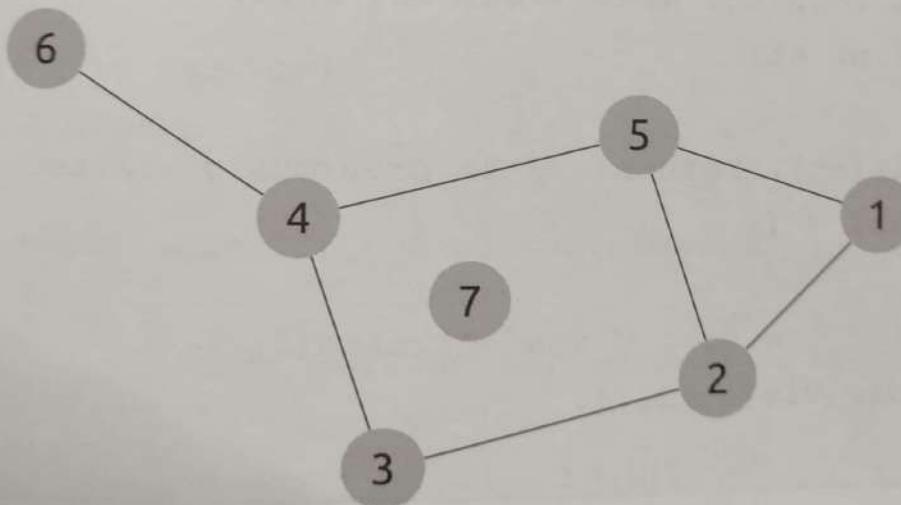
```

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

```

একটি উদাহরণ দেওয়া যাক। চিত্র ৮.১ দেখো। মনে কর আমরা 6 থেকে আমাদের BFS
শুরু করব ($s = 6$)। তাহলে আমরা একটি কিউ (queue) তে 6 কে ঢুকাব (লাইন 15) আর
6 কে *visited* করে দেব (লাইন 16)। এবার আমরা কিউ যতক্ষণ না ফাঁকা হয় (লাইন 18)

ততক্ষণ কিউ হতে একে একে নোড তুলব (লাইন 20, 21)। প্রথমে আমাদের উঠবে 6. এখন 6 এর অ্যাডজাসেন্সি লিস্ট (adjacency list) দেখবো (লাইন 23)। চাইলে তোমরা অ্যাডজাসেন্সি ম্যাট্রিক্স (adjacency matrix) ও ব্যবহার করতে পারো, তবে সেক্ষেত্রে তোমার টাইম কমপ্লেক্সিটি আর $O(V + E)$ থাকবে না $O(V^2)$ হয়ে যাবে। যাই হোক, 6 এর অ্যাডজাসেন্সি লিস্টে কেবল 4 আছে আর তা এখনও visited হয় নাই। তাই আমরা 4 কে visited করে (লাইন 27) একে কিউতে পুশ (push) করে দেব (লাইন 28)। আবার কিউ থেকে একটি নোড তুলব, এবার বের হবে 4, 4 এর অ্যাডজাসেন্সি লিস্টে আছে 5, 3 আর 6. 5 আর 3 যেহেতু এখনও visited হয় নি তারা কিউতে পুশ হবে আর 6 যেহেতু ইতোমধ্যেই visited হয়ে গেছে তাই একে কিছু করা হবে না। এবার কিউ থেকে উঠবে 5. এখানে একটি জিনিস বলে রাখা প্রয়োজন আর তা হলো অ্যাডজাসেন্সি লিস্টে আমাদের নোডগুলো কি অর্ডার (order) বা ক্রমে আছে তার উপর ভিত্তি করে কিন্তু কিউ থেকে কে উঠবে সেটা ভিন্ন হতে পারে। যাই হোক, 5 এর অ্যাডজাসেন্ট (adjacent) বা প্রতিবেশী (neighbor) হল 1, 2 আর 4. 1 আর 2 কিউতে পুশ হবে আর যেহেতু 4 ইতোমধ্যেই visited হয়ে গেছে তাই আর কিছু করা লাগবে না। এবার কিউ থেকে উঠবে 3 যার অ্যাডজাসেন্ট 2 ও 4, কিন্তু নতুনই visited হয়ে গেছে তাই নতুন করে কিউতে আর কেউ ঢুকবে না। এভাবে একে একে 1 আর 2 উঠবে। কিন্তু নতুন কেউ আর কিউতে ঢুকবে না। খেয়াল কর এখানে কিন্তু 7 visited হয় নাই। কারণ 6 এর সঙ্গে কোনোভাবেই 7 সংযুক্ত বা কানেক্টেড (connected) না। BFS দিয়ে কি কি করা যেতে পারে তার একটি উদাহরণ দিলাম মাত্র। কিছু পরে আরও বিস্তারিতভাবে জানতে পারবে এর সাহায্যে আর কী কী করা সম্ভব।



নকশা ৮.১: ব্রেডথ ফাস্ট সার্চ (Breadth First Search - BFS)

৮.২ ডেপথ ফাস্ট সার্চ (Depth First Search - DFS)

এটি গ্রাফে ঘুরে বেড়ানোর আরেকটি উপায়। এই পদ্ধতিতে যা করা হয় তাহলো শুরুর নোড s এ শুরু করে তুমি যেতেই থাকবে, যতক্ষণ না তুমি এমন নোডে পৌছাও যেখানে এর আগে এসেছিলে। সেরকম কোনো নোডে পৌছালে তোমাকে পিছিয়ে যেতে হবে। পিছিয়ে গিয়ে তুমি অন্য বাহু দিয়ে

যাওয়ার চেষ্টা করবে। খেয়াল কর, এই কাজটি কিন্তু কিছুটা রিকার্সিভ (recursive) গোছের। তুমি একটি নোডে আছ, তোমার কাজ হলো এর কোনো একটি বাল্ড দিয়ে বের হওয়া। যদি গিয়ে দেখ যে সেখানে আগেই এসেছিলে তাহলে ফিরে এসো আর না হলে রিকার্সিভ উপায়ে একই কাজ কর। এর কোড ৮.২ এ দেওয়া হলো। একটি জিনিস বলে রাখি, তুমি চাইলে DFS ফাংশন কল করার আগেই কিন্তু যাচাই করে দেখতে পারো যে যেখানে তুমি এখন যেতে চাচ্ছ সেখানে আগেই গিয়েছিলে কিনা। অনেক সময় দেখা যায় যে DFS ফাংশন কল করার মূল্য (cost) অনেক বেশি (অনেক সময় বড় বড় স্টেট (state) পরিবর্তন করতে হয়) সেক্ষেত্রে আগে থেকে যাচাই করে যাওয়া বুদ্ধিমানের মতো কাজ।

কোড ৮.২: 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 কমপ্লেক্সিটি কিন্তু $O(V + E)$ কিন্তু এর একটি সমস্যা হলো এতে সর্বোচ্চ V টি পর পর রিকার্সিভ কল হতে পারে। সুতরাং আমাদের কম্পাইলারের ডিফল্ট স্ট্যাক সাইজ (default stack size) যদি কম হয় তাহলে এই কোডে স্ট্যাক ওভারফ্লো (stack overflow) হতে পারে। বেশির ভাগ সময়ই অনলাইন জাজগুলোতে এই সমস্যা হয় না। কিন্তু যদি সমস্যা হয় তাহলে আমাদের পুরোপুরি নিজে নিজে এই রিকার্সিভ এর কাজ স্ট্যাক এর মাধ্যমে করতে হবে। স্ট্যাক ব্যবহার করে DFS এর কোড ৮.৩ এ দেওয়া হলো।

```

1 #include<vector>
2 #include<stack>
3 using namespace std;
4
5 vector<int> adj[100];
6 int edge_id[100];
7 int vis[100];
8
9 // s is starting vertex
10 // n is number of vertices
11 void dfs(int s, int n)
12 {
13     for(int i = 0; i < n; i++) edge_id[i] = vis[i] = 0;
14
15     stack<int> S;
16     S.push(s);
17     while(!S.empty())
18     {
19         int u = S.top();
20         S.pop();
21
22         while(edge_id[u] < adj[u].size())
23         {
24             // start looking into edges, from
25             // the index we left
26             int v = adj[u][edge_id[u]];
27             // update edge pointer to check
28             // next time
29             edge_id[u]++;
30             // if the vertex is not yet visited
31             if(vis[v] == 0)
32             {
33                 vis[v] = 1;
34                 // order of push important for dfs
35                 S.push(u);
36             }
37         }
38     }
39 }

```

```

36           // first we will check v, then we
37           // will come back to u
38           s.push(v);
39           // note, stack is last in first
40           // out. So v will be popped before u
41           break;
42       }
43   }
44 }
45 }

```

আগের চিত্র ৮.১ এ কীভাবে DFS করা যায় তা বলি। আমরা এখানে লাইন বলতে কোড ৮.২ এর লাইন বোঝাচ্ছি। মনে কর �DFS করা শুরু করেছ ৫ থেকে। প্রথমে আমরা দেখি *5visited* কিনা (লাইন 11)। যেহেতু না তাই আমরা একে *visited* করে দেই (লাইন 12)। এবারের অ্যাডজাসেন্সি লিস্টের ভেতর দিয়ে যাই। মনে করি এর অ্যাডজাসেন্সি লিস্ট হলো 2, 1 ও 4। প্রথমে 2 এ DFS করি। যেহেতু এটি এখনও *visited* হয় নাই তাই একে *visited* করে এর অ্যাডজাসেন্সি লিস্ট (5, 1, 3) দেখি। 5 এ গিয়ে দেখবো যে এটি ইতোমধ্যেই *visited* হয়ে গেছে তাই আমরা লাইন 11 হতে ফেরত আসব। এর পর দেখবো 1 আর এটি এখনও *visited* হয় নাই। একে *visited* করে এর অ্যাডজাসেন্সি লিস্ট (5, 2) দেখবো। দুজনই ইতোমধ্যেই *visited* হয়ে গেছে তাই দুজনের ক্ষেত্রেই আমরা DFS এ ঢুকেই বের হয়ে যাব। আমরা 1 এর অ্যাডজাসেন্সি লিস্টের সবার জন্য DFS করা শেষ করেছি তাই আমরা ফেরত যাব। কোথায় ফেরত যাব? যেখান থেকে 1 এর DFS কল হয়েছিল। কোথায় থেকে হয়েছিল? 2 থেকে। আমরা এবার 2 এর অন্যান্য অ্যাডজাসেন্ট নোডগুলো দেখবো। 5 আর 1 এর জন্য কিন্তু ইতোমধ্যেই DFS কল করে ফেলেছি। তাই এবার আমরা 3 এর জন্য কল করব। এভাবে চলতে থাকবে। আমরা চাইলে *visited* হয়েছে কিনা সেটা DFS এ ঢুকে যাচাই না করে, DFS কল করার আগেই অর্থাৎ লাইন 15 এর আগে যাচাই করতে পারতাম।

৮.৩ DFS ও BFS এর কিছু সমস্যা

DFS ও BFS কিছুটা একই ধরনের অ্যালগরিদম। এর মূল কাজ হলো গ্রাফ ট্রাভার্স করা। যদিও কিছু জটিল সমস্যা আছে যা কেবল মাত্র BFS দিয়ে বা কেবল মাত্র DFS দিয়ে সমাধান করা যায় কিন্তু অপরটি দিয়ে না, কিন্তু সাধারণ সমস্যাগুলো বেশির ভাগ সময় BFS বা DFS দুটি দিয়েই সমাধান করা যায়। কোনটা দিয়ে করবে তা আসলে তোমার উপর নির্ভর করবে। যাই হোক, কিছু সমস্যা দেখা যাক।

৮.৩.১ কম্পোনেন্ট (Component) বের করা

এই সমস্যায় আমাদের বের করতে হবে কোন কোন নোডগুলো কানেক্টেড। ২ টি নোডকে কানেক্টেড বলা হয় যদি বাহ্যিকভাবে এক নোড হতে অন্য নোডটায় যাওয়া সম্ভব হয়। আমরা এই সমস্যাটি BFS ও DFS দুটি অ্যালগরিদম দিয়েই সমাধান করতে পারি। আমরা আমাদের BFS ও DFS এর বাহিরে একটি লুপ চালাবো। প্রতি নোডের জন্য দেখবো যে এটি *visited* হয়েছে কিনা। নাহলে ওই নোড হতে BFS বা DFS করব। এভাবে যতবার আমাদের BFS বা DFS করতে হবে সেটিই উভয়। যেমন চিত্র ৮.১ তে আমরা ১ থেকে BFS বা DFS করলে ১ হতে ৬ সবাই *visited* হয়ে যাবে। সুতরাং আমাদের ২, ৩, ..., ৬ পর্যন্ত কাউকে দিয়ে আবারো BFS বা DFS করার দরকার নেই। কিন্তু এখনও আমাদের ৭ *visited* হয় নাই। সুতরাং ৭ এর জন্য আমরা আবার BFS বা DFS করব। তাহলে আমাদের সব নোড *visited* হয়ে যাবে। মোট দুবার বাহিরের লুপ হতে আমাদের BFS বা DFS কে কল করতে হয়েছে। সুতরাং এখানে আমাদের কম্পোনেন্ট আছে ২ টি।

৮.৩.২ দুটি নোডের দূরত্ব

মনে কর একটি গ্রাফ আর দুটি নোড দিয়ে বলা হলো যে তাদের দূরত্ব বের কর। দূরত্ব বলতে সর্বনিম্ন কয়টি বাহু পার করে যেতে হয় সেটি বোঝানো হচ্ছে এখানে। কীভাবে করবে? এটি কিন্তু BFS দিয়ে খুব সহজেই সমাধান করা যায়। খেয়াল করে দেখ, BFS এর ক্ষেত্রে কিন্তু শুরুর নোড থেকে যেই পথে অন্য একটি নোডে যাওয়া হয় তা কিন্তু সবচেয়ে সংক্ষিপ্ত পথ বা শর্টেস্ট পাথ (shortest path)। সুতরাং যখন আমরা কোনো নোড থেকে আরেকটি নতুন নোডে যাব তখন আমরা বলতে পারি নতুন নোডের দূরত্ব যেখান থেকে আসা হচ্ছে তার থেকে এক বেশি। সুতরাং আমাদের যে দুটি নোড দেওয়া আছে তাদের একটি থেকে BFS শুরু করলে আমরা অন্য নোডে যাওয়ার দূরত্ব পেয়ে যাব।

খেয়াল কর আমরা কিন্তু এই সমস্যা DFS দিয়ে সমাধান করতে পারব না। কারণ DFS দিয়ে সবসময় শর্টেস্ট পাথ (shortest path) এ যাওয়া হয় না। মনে কর A, B ও C তিনটি নোড। প্রত্যেকটি থেকে অন্য দুটি নোডে যাওয়া যায়। এখন A হতে DFS শুরু করলে ধরা যাক আমরা প্রথমে 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 দুটি নোডের মধ্যে শর্টেস্ট পাথটি প্রিন্ট করতে বলে? পাথ প্রিন্টিং (path printing) টেকনিক মোটামুটি সব ক্ষেত্রেই একই রকম। তুমি কোনো নোডে যাওয়ার সময় লিখে রাখবে এখানে কীভাবে এসেছ। যেমন BFS এর ক্ষেত্রে তুমি কোনো নতুন নোডে আসার সময় লিখে রাখবে কোন নোড থেকে এখানে এসেছ। তাহলে BFS শেষে, তুমি দেখবে শেষ নোডে কোথায় থেকে এসেছ, এর পর দেখবে সেই নোডে কোথায় থেকে এসেছ এভাবে এক সময় দেখবে তুমি শুরুর নোডে চলে এসেছ। তাহলে এটিই তোমার পাথ। যদিও জিনিসটি খুব একটি কঠিন না তাও কোড ৮.৪ এ আমরা s হতে t তে যাওয়ার শর্টেস্ট পাথকে প্রিন্ট করে দেখালাম।

```

1 vector<int> adj[100];
2 int visited[100];
3 int dist[100]; // distance from starting vertex
4 int p[100]; // we came to i from p[i]
5
6 // s is the starting vertex
7 // t is the destination vertex
8 // n is the number of vertices (0 ... n - 1)
9 void bfs(int s, int n)
10 {
11     for(int i = 0; i < n; i++) vis[i] = 0;
12
13     queue<int> Q;
14     Q.push(s);
15     visited[s] = 1;
16     dist[s] = 0;
17     p[s] = s;
18
19     while(!Q.empty())
20     {
21         int u = Q.front();
22         Q.pop();
23
24         for(int i = 0; i < adj[u].size(); i++)
25             if(visited[adj[u][i]] == 0)
26             {
27                 int v = adj[u][i];
28                 visited[v] = 1;
29                 dist[v] = dist[u] + 1;
30                 p[v] = u;
31                 Q.push(v);
32             }
33     }
34     if (vis[t] == 0) {

```

```

// no path from s to t.

06     return;
07 }
08
09     vector<int> path;
10     path.push_back(t);
11     int now = t;
12     while (now != s) {
13         now = p[now];
14         path.push_back(now);
15     }
16
17     // In vector path, the entire path is in
18     // reverse order.
19
20     // So we can run a loop from path.size() - 1 to 0.
21     // Or use reverse(path.begin(), path.end())
22     // to reverse the entire vector;
23 }

```

৮.৩.৩ তিনটি গ্লাস ও পানি

খুব প্রচলিত একটি ধাঁধাঁ হলো, তোমাকে 3 ও 5 লিটারের দুটি ফাঁকা গ্লাস আর 8 লিটারের একটি পানি ভর্তি গ্লাস দেওয়া 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 ঢালাচালি করে? তাহলে তোমাকে BFS করতে হবে। এটা তো বললাম যে এটা BFS দিয়ে সমাধান করা যাবে কিন্তু এখানে নোডই বা কই আর বাহুই বা কই? আসলে BFS করতে যে ভার্টেক্স বা বাহু লাগবে এই ধারণা ঠিক না। আমাদের যা জানতে হবে তা হলো আমরা কোথায় আছি আর এখন থেকে আমরা কোথায় যেতে পারি। কিছুটা DP এর মতো চিন্তা করতে পার। আমরা যেখানে আছি সেটাকে একটি স্টেট (state) আকারে উপস্থাপন করতে হবে- এটাই মূল জিনিস। যেমন আমাদের এই সমস্যার স্টেট হতে পারে তিনটি পাত্রে কতখানি করে পানি আছে। সুতরাং আমরা 1D অ্যারেতে *visited* না রেখে একটি 3D অ্যারেতে *visited* রাখতে পারি আর কিউতে একটি নম্বর না রেখে তিনটি নম্বর একত্রে স্ট্রাকচার করে রাখতে পারি (স্ট্রাকচার এর কিউ)। আর বাহু? এই স্টেট এর উপর কী কী অপারেশন (operation) করতে পারবে এবং সেই অপারেশনের ফলে তুমি কোন কোন স্টেট এ যাবে সেটিই হলো বাহু। যেমন তুমি প্রথম গ্লাস হতে দ্বিতীয় গ্লাসে পানি ঢালবে বা ইত্যাদি এসব অপারেশনের ফলে কোন স্টেট এ যেতে পারবে সেটিই হলো বাহু।

এভাবে BFS করলেই এই সমস্যা সমাধান হয়ে যাবে। তোমরা চাইলে কিন্তু এই স্টেটকে দুটি সংখ্যা দিয়েও প্রকাশ করতে পার: প্রথম দুই পাত্রে কতখানি করে পানি আছে। কারণ 8 থেকে ওই পরিমাণ বাদ দিলে তুমি তৃতীয় পাত্রের পানির পরিমাণ পেয়ে যাবে। এই অপটিমাইজেশনের ফলে তোমার রানটাইমে কোনো পরিবর্তন হবে না তবে কোনো একটি স্টেট কী *visited* হয়েছে কি না তা জানার জন্য যেই *visited* এর অ্যারে লাগবে তার জায়গা কম লাগবে। তোমরা চাইলে কিউতে তিনটি নম্বরই রাখতে পারো এতে করে কষ্ট করে 8 থেকে বিয়োগ করার কাজ করতে হবে না। আর সেই সঙ্গে আমরা *visited* রাখার সময় প্রথম দুটি সংখ্যা ব্যবহার করব যাতে আমাদের মেমোরী কম লাগে। এই চালাকি প্রায়ই কাজে লাগে। যাতে বেশি গাণিতিক অপারেশনের দরকার না হয় সেজন্য আমরা কিউতে সব জিনিসই রেখে দেব কিন্তু *visited* বা মেমোয়াইজেশন (memoization) এর জন্য যাতে কম জায়গা লাগে সেজন্য আমরা ছোট *visited* ম্যাট্রিক্স ব্যবহার করব।

৮.৩.৮ UVa 10653

তোমাকে একটি গ্রিড দেওয়া থাকবে। সেই সঙ্গে তোমার শুরুর জায়গা আর শেষ গন্তব্য দেওয়া থাকবে। তোমাকে সবচেয়ে কম কত সময়ে গন্তব্যে পৌঁছানো যায় তা বলতে হবে। এটি খুব ভালো মতোই বুঝা যাচ্ছে যে গ্রিডের একেকটি ঘর হলো একেকটি নোড। তোমরা যারা প্রথম প্রথম প্রোগ্রামিং করছ তারা হয়তো প্রতি ঘরকে $1, 2, \dots, RC$ এভাবে নম্বর দিবে কিন্তু এর থেকে সুবিধা হবে তুমি যদি নোডকে (r, c) ভাবে প্রকাশ কর। আর কোড ৮.৫ এর মতো দুটি ম্যাট্রিক্স রাখ।

কোড ৮.৫: cellbfs.cpp

```

১ int dr[] = {-1, 0, 1, 0};
২ int dc[] = {0, 1, 0, -1};
৩
৪ int valid(int r, int c)
৫ {
৬     return r >= 0 && r < R && c >= 0 && c < C;
৭     // also may be check if (r, c) is empty.
৮     // you may also check if the cell is visited.
৯ }
```

তাহলে (r, c) থেকে নতুন যেসব ঘরে যেতে পারবে সেসব হলো $(r + dr[i], c + dc[i])$ (i এর একটি লুপ 0 হতে 3 পর্যন্ত চালাও) আর এই নতুন ঘরটি আদো *valid* কিনা তা জানার জন্য ৮.৫ কোডের *valid* ফাংশনকে কল করে দেখ। এভাবে অনেক সহজে গ্রিড এ BFS এর কোড হয়ে যায়।

এটি তুমি BFS বা DFS যেকোনোটি দিয়েই সমাধান করতে পারবে। কারণ এখানে সর্বনিম্ন কয়টি pebble থাকবে অর্থাৎ কোন কোন গেইম configuration এ যাওয়া যাবে সেটিই মূল জিনিস। এই সমস্যার স্টেট কীভাবে প্রকাশ করবে? যেহেতু মাত্র 12 টি ঘর আর প্রতিটি ঘরে pebble আছে কিনা এটা জানলেই হয় তাই 12 টি 0 – 1 দিয়ে আমরা স্টেট বানাতে পারি। এজন্য আমরা বিটমাস্ক (bitmask) অর্থাৎ 12 টি বিট ব্যবহার করে স্টেট বানালে 2^{12} আকারের একটি আরেতেই visited রাখতে পারি।

৮.৩.৬ ০ ও 1 মূল্য (cost) এর গ্রাফ

ধরা যাক তোমাকে একটি ওয়েইটেড গ্রাফ (weighted graph) দেওয়া হলো যার বাহুর মূল্য (edge cost) হয় 0 না হয় 1। এক্ষেত্রে এক নোড থেকে আরেক নোডে যাওয়ার শর্টেস্ট পাথ বের করার একটি সহজ উপায় হলো BFS এর মতো কাজ করা। তুমি যখনই 0 দিয়ে যেতে চাইবে তখন কিউ এর শুরুতে ইনসার্ট (insert) করবে, আর 1 দিয়ে যেতে চাইলে কিউ এর শেষে। আর নেওয়ার সময় সবসময় সামনে থেকে নেবে। আসলে দুই দিকে ইনসার্ট করতে পারলে সেটা আর কিউ থাকে না, এর আরেক নাম হলো ডিকিউ (dequeue). STL এ dequeue বলে বিল্ট-ইন ডেটা স্ট্রাকচার আছে। যাই হোক এই সমাধানের সময় ডিকিউয়ের দৈর্ঘ্য প্রায় $2n$ এর সমান হয়ে যেতে পারে। তোমাদের একটি $dist$ এর অ্যারে নিয়ে তাতে দূরত্ব রাখতে হবে এবং তখনই তুমি ডিকিউয়ে ইনসার্ট করবে যখন তোমার এখনকার মূল্য, $dist$ অ্যারেতে থাকা মূল্যের থেকে কম হয়। আর আগেই বলেছি যদি তুমি 0 মূল্যের বাহু ব্যবহার করে আসো তাহলে তো ডিকিউয়ের সামনে রাখবে আর যদি 1 মূল্যের বাহু হয় তাহলে পেছনে।

৮.৪ সিঙ্গল সোর্স শর্টেস্ট পাথ (Single Source Shortest Path)

শর্টেস্ট পাথ সমস্যা হলো, কোনো একটি ওয়েইটেড গ্রাফে এক নোড থেকে আরেক নোডে যাওয়ার সর্বনিম্ন মূল্য বের করার সমস্যা। সাধারণত আমরা দুই ধরনের শর্টেস্ট পাথ সমস্যা দেখে থাকি। সিঙ্গল সোর্স শর্টেস্ট পাথ (Single source shortest path) এবং অল পেয়ার শর্টেস্ট পাথ (All pair shortest path)। সিঙ্গল সোর্স শর্টেস্ট পাথ সমস্যায় আমরা এক নোড থেকে অন্য সব নোডে যাওয়ার সর্বনিম্ন মূল্য বের করে থাকি আর অল পেয়ার শর্টেস্ট পাথ সমস্যায় আমাদের প্রতিটি নোড থেকে অন্য সব নোডে যাওয়ার মূল্য বের করতে হয়। তোমরা হয়তো ভাবতে পার যে তাহলে Single Source Single Destination Shortest Path বলে আরও একটি কিছু বলছি না কেন? কারণ হলো, সিঙ্গল সোর্স শর্টেস্ট পাথের অ্যালগরিদমের মাধ্যমে এই Single Destination এর ভ্যারিয়েশন (variation) টি সল্ভ করা যায় আর তাছাড়া মূল

কারণ হলো, Single Destination এর ভ্যারিয়েশন সমাধান করার জন্য আসলে "সহজতর" কোনো অ্যালগরিদম নেই। খেয়াল কর আমি কিন্তু "সহজতর" বলেছি। আমি এখানে সিঙ্গল সোর্স শটেস্ট পাথের অ্যালগরিদমের সঙ্গে তুলনা করছি। অর্থাৎ, আমরা সিঙ্গল সোর্স শটেস্ট পাথের জন্য যেসব অ্যালগরিদম দেখব, Single Destination এর জন্য তার থেকেও কার্যকর বা ইফিসিয়েন্ট (efficient) অ্যালগরিদম আসলে আমার জানা নেই। তবে হ্যাঁ, হয়তো খুবই নামান্য অপটিমাইজেশন করতে পারবে কিন্তু আসলে worst কেইসে একই টাইম কমপ্লেক্সিটি পাবে। এসব কথা এখন বুঝতে না পারলেও সমস্যা নেই, নিচের অ্যালগরিদমগুলো বুঝে এসে এই কথাগুলো পড়লে আশা করি বুঝতে পারবে আমি কোন অপটিমাইজেশনের কথা বলছি, বা কেন বলছি যে Single Destination এর ভ্যারিয়েশনে আমরা সিঙ্গল সোর্স শটেস্ট পাথের অ্যালগরিদমই ব্যবহার করব।

সিঙ্গল সোর্স শটেস্ট পাথের জন্য দুটি অ্যালগরিদম খুব বেশি ব্যবহার করা হয়। একটি হলো ডায়াকস্ট্রা'র অ্যালগরিদম (Dijkstra's Algorithm)¹ আর আরেকটি হলো বেলম্যান ফোর্ড অ্যালগরিদম (Bellman Ford Algorithm).

৮.৪.১ ডায়াকস্ট্রা'র অ্যালগরিদম (Dijkstra's Algorithm)

সাধারণত এই অ্যালগরিদম ব্যবহার করা হয় যদি সব বাহুর মূল্য অঞ্চলাত্মক (non-negative) হয়। একে বিভিন্নভাবে ইমপ্লিমেন্ট (implement) করলে বিভিন্ন টাইম কমপ্লেক্সিটি পাওয়া সম্ভব। আমরা $O(n^2)$ দিয়ে শুরু করি।

- ঃ প্রথমে একটি n দৈর্ঘ্যের অ্যারে নেই, ধরা যাক এর নাম *dist* (distance এর সংক্ষিপ্ত রূপ) এবং এর প্রতিটি উপাদানকে অসীম (infinity) মূল্য দেই। অনেকে অসীম হিসেবে 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 কোডটি সেই অনুসারে লিখলেই হবে।
- ঃ যেহেতু আমরা সিঙ্গল সোর্স শটেস্ট পাথ সমস্যা সমাধান করছি, সুতরাং আমাদের কাছে একটি উৎস বা সোর্স (source) বা যেখানে থেকে আমাদের যাত্রা শুরু সেই নোড আছে। ধরে নেই সেটা s । তাহলে আমরা উপরের অ্যারেতে s এর অবস্থানে 0 বসাব। এর মানে হলো, s এ পৌঁছানোর মূল্য হলো 0.
- ঃ আরও একটি n দৈর্ঘ্যের অ্যারে নেই যার নাম ধরা যাক *visited* এবং এর প্রতিটি স্থান 0 দিয়ে ইনিশিয়ালাইজেশন (initialization) করি।

¹আমি আসলে জানি না আসল উচ্চারণ কী! অনেকে অনেকভাবে উচ্চারণ করে। আমি ও বিভিন্ন বয়সে বিভিন্ন উচ্চারণ করতাম, ছোটবেলায় ডিজিকস্ট্রা বড় হয়ে ডায়াকস্ট্রা। মাঝে মনে হয় আরও অনেক কিছুই বলতাম। তবে মোটামুটি সবাই ডায়াকস্ট্রা বলে অভ্যন্ত।

এখন আমাদের এই ধাপটি বার বার করতে হবে। এই ধাপে আমরা দেখব কোন কোন নোডের *visited* এ ০ আছে, তাদের মধ্য থেকে যেই নোডে যাওয়ার দূরত্ব সবচেয়ে কম অর্থাৎ $dist$ অ্যারে সবচেয়ে কম মূল্যের নোডকে নির্বাচন করি। ধরা যাক সেই নোডটি হলো u . এখন *visited* অ্যারেতে u এর অবস্থানে ১ বসিয়ে দেই। এবার আমরা দেখব u থেকে কোথায় কোথায় যাওয়া যায়? ধরা যাক, u থেকে v তে যাওয়ার জন্য একটি বাহু আছে যার মূল্য হলো c . আমরা দেখব কোনটি ছোট $dist[v]$ নাকি $dist[u] + c$? অর্থাৎ আমরা দেখতে চাচ্ছি যে v তে এখন পর্যন্ত যেই কম খরচে যাওয়ার রাস্তা বের হয়েছে সেটা ভালো নাকি আমরা যদি প্রথমে u তে এসে এর পর $u - v$ বাহু ব্যবহার করে যাই তাহলে সেটা ভালো হবে। যদি $dist[u] + c$ কম হয় তাহলে $dist[v]$ কে এই মান দিয়ে আপডেট (update) করি।^১ এভাবে আমরা একে একে u এর সঙ্গে লাগানো সব বাহু নিয়ে নিয়ে এভাবে আপডেট করার চেষ্টা করব।^২ এভাবে যতক্ষণ না আমাদের সব নোড *visited* হয়ে যায় (অর্থাৎ *visited* অ্যারেতে সবাই ১ হওয়া পর্যন্ত) ততক্ষণ আমরা এই পদ্ধতি চালাতে থাকব।

এখন তুমি $dist$ অ্যারেতে s থেকে সব নোডের সর্বনিম্ন দূরত্ব পেয়ে যাবে।

এখানে আমাদের চতুর্থ ধাপ কিন্তু চলবে n সংখ্যকবার। প্রতিবার আমরা সব নোড পর্যবেক্ষণ করে বের করছি আমাদের ওই ধাপের u কে হবে। সুতরাং আমরা n বার n সমান কাজ করছি: $O(n^2)$. এর পরে প্রতি u এর জন্য আমরা এর সঙ্গে লাগানো বাহুগুলো যাচাই করছি, এর মানে হলো প্রতিটি বাহু আসলে খুব জোর ২ বার যাচাই হবে। সুতরাং আমাদের কমপ্লেক্সিটি হবে $O(n^2 + m)$ বা $O(n^2)$, কারণ $m \leq n^2$. তা না হওয়া মানে হলো আমাদের গ্রাফে মাল্টিএজ (multi-edge) আছে, আর মাল্টিএজ থাকলে আমাদের সবচেয়ে কম খরচের বাহু রেখে দিলেই হয়, সব সমান্তরাল বা প্যারালাল (parallel) বাহু না রাখলেও চলে।

এখন আমরা এই কমপ্লেক্সিটিকে চাইলেই কমিয়ে $O(m \log m)$ করতে পারি। খেয়াল করে দেখ আমরা একটি ধাপে লুপ চালিয়ে কোন *unvisited* নোডে যাওয়ার খরচ সবচেয়ে কম তা বের করেছিলাম। তা না করে আমরা চাইলে STL এর প্রায়োরিটি কিউ (priority queue) ব্যবহার করতে পারি। এজন্য যা করতে হবে তা হলো, একটি স্ট্রাকচারের প্রায়োরিটি কিউ বানাতে হবে। এর পর যখনই কোনো নোডের মূল্য আপডেট করা হবে তখনই সেই নোডকে মূল্যসহ প্রায়োরিটি কিউতে পুশ করে দিতে হবে। আর প্রতিবার সর্বনিম্ন মূল্যের নোড নির্বাচন করার সময় প্রায়োরিটি কিউ থেকে সবচেয়ে কম মূল্যের স্ট্রাকচার নিয়ে দেখতে হবে সেখানে যে নোড আছে সেটা *visited* কিনা। যদি *visited* হয়ে থাকে তাহলে এটা নিয়ে আর কাজ করার দরকার নেই। খেয়াল কর আমরা এই পদ্ধতিতে কিন্তু একটি নোড একাধিকবার পুশ করছি। আমরা ধরে নিতে পারি প্রতি বাহুর জন্য খুব জোর একটি নোড পুশ হয়। সুতরাং সর্বোচ্চ m বার পুশ হয় m আকারের হীপ (heap) বা প্রায়োরিটি কিউতে, সুতরাং আমাদের কমপ্লেক্সিটি $O(m \log m)$.

^১আপডেট (update) করা মানে হলো পরিবর্তন করা, বা ভালো মান দিয়ে পরিবর্তন করা।

^২খেয়াল কর, আমাদের মূল লক্ষ্য হলো u থেকে যেই যেই বাহু দিয়ে যাওয়া যায় তাদের আপডেট করা। সুতরাং আমাদের গ্রাফ ডিরেক্টেড (directed) বা আনডিরেক্টেড (undirected) যাই হোক না কেন কোনো সমস্যা নেই।

যারা মনোযোগ দিয়ে পড়েছে আশা করি বুঝতে পারছ যে আমাদের আরও অপটিমাইজেশনের সুযোগ আছে। আমরা যদি প্রায়োরিটি কিউতে বার বার পুশ না করে আগের পুশ করা নোডের মূল্য আপডেট করতে পারতাম তাহলে কিন্তু কমপ্লেক্সিটি কমে যেত। সুতরাং তোমাদের যদি প্রোগ্রামকে আরও ইফিসিয়েন্ট করার প্রয়োজন হয় তাহলে নিজেরা হীপ বানিয়ে করতে পার কিন্তু এতে আরও অনেক কোড করতে হয় বলে আমরা সহজে নিজে থেকে হীপ বানাই না।

যারা STL এর সেট (set) সম্পর্কে জানো তারা হয়তো মনে করতে পার প্রায়োরিটি কিউ ব্যবহার না করে সেট ব্যবহার করলে তো কমপ্লেক্সিটি আরও কমতে পারে! কারণ সেট এ চাইলে ডিলিট (remove) করা যায়, সুতরাং আমরা আমাদের কমপ্লেক্সিটি $O(m \log n)$ তে নামিয়ে ফেলতে পারি। কিন্তু সমস্যা হলো, সেটের আভ্যন্তরীণ অ্যালগরিদম অনেক জটিল আরও পরিষ্কার করে বলতে গেলে বলতে হয়, প্রায়োরিটি কিউ আসলে একটি হীপ আর সেট আসলে একটি রেড ব্ল্যাক ট্রি (red black tree). রেড ব্ল্যাক ট্রি এর আভ্যন্তরীণ গঠন অনেক জটিল বিধায় এদের দুজনের মোটামুটি সব অপারেশনের কমপ্লেক্সিটি $O(\log n)$ হলেও সেটের "constant factor" আমার জানা মতে অনেক বেশি। সুতরাং বেশির ভাগ সময়েই দেখা যায়, প্রায়োরিটি কিউ, সেটের থেকে ডায়াকস্ট্র্যাক্সে'র অ্যালগরিদম এ ভালোভাবে কাজ করছে। তবে যদি কখনও তোমরা খুব ঘন সঞ্চিবিষ্ট (dense) গ্রাফের সমূখ্যীন হও অর্থাৎ $m \approx n^2$ তাহলে প্রায়োরিটি কিউ না ব্যবহার করে সেট ব্যবহার করলে ভালো ফলাফল পাবে।

এখন আশা করি নিজেরাই বুঝতে পারছ কেন এই অ্যালগরিদম ঋণাত্মক বাহুর মূল্য (negative edge cost) ক্ষেত্রে কাজ করবে না। ঋণাত্মক বাহুর মূল্য থাকলে বড় সমস্যা হলো এই অ্যালগরিদম অনুসারে গ্রাফে প্রসেসিং করতে থাকলে এক সময় দেখা যাবে visited নোডের মূল্য কমবে কিন্তু আমরা এই অ্যালগরিদমে visited নোডকে দুবার প্রসেসিং করি না। যদি আমরা বার বার প্রসেসিং করতাম আর গ্রাফে ঋণাত্মক চক্র বা সাইকেল (negative cycle) না থাকত তাহলে এই অ্যালগরিদমই ঋণাত্মক বাহুর মূল্যেও কাজ করত তবে সেক্ষেত্রে আমাদের কমপ্লেক্সিটি আসলে খুব একটা ভালো হবে না, আমি নিজেও নিশ্চিত না কমপ্লেক্সিটি কত হবে, মনে হয় এক্সপোনেন্সিয়াল (exponential) এর মতো কিছু হবে। তবে এভাবে যে ঋণাত্মক বাহুর মূল্যের গ্রাফে শেট্টেস্ট পাথ বের করা সম্ভব তা জেনে রাখা ভালো। যদি আমার দুর্বল সূতিশক্তি আমার সঙ্গে দুষ্টুমি না করে তাহলে আমার মনে হয় আমাকে এভাবেও কিছু সমস্যা সমাধান করতে হয়েছিল। তোমাদের সুবিধার জন্য এর কোড ৮.৬ তে প্রায়োরিটি কিউ ব্যবহার করে দেওয়া হলো। এখানে আমি visited অ্যারে ব্যবহার করলাম না। যেহেতু গ্রাফের বাহসমূহের মূল্য অঋণাত্মক সেহেতু আমরা visited এর কাজ লাইন 36 আর লাইন 41 এর if দিয়ে করে ফেলেছি। একটু চিন্তা করে দেখো।

কোড ৮.৬: dijkstra.cpp

```

1 struct Node {
2     int at, cost;
3     Node(int _at, int _cost) {

```

```

at = _at;
cost = _cost;
}

};

bool operator<(Node A, Node B) {
    // Priority queue returns the greatest value.
    // So we need to write the comparator in a way
    // so that cheapest value becomes greatest value.
    return A.cost > B.cost;
}

struct Edge {
    int v, w;
};

vector<Edge> adj[100]; // adjacency list of weighted edges.
priority_queue<Node> PQ;
int dist[100];
int n;

void dijkstra(int s) {
    for (int i = 1; i <= n; i++) {
        dist[i] = 1000000000;
    }
    dist[s] = 0;
    PQ.push(Node(s, 0));

    while (!PQ.empty()) {
        Node u = PQ.top();
        PQ.pop();

        if (u.cost != dist[u.at]) {
            continue;
        }
    }
}

```

```

80         for (Edge e : adj[u.at]) {
81             if (dist[e.v] > u.cost + e.w) {
82                 dist[e.v] = u.cost + e.w;
83                 PQ.push(Node(e.v, dist[e.v]));
84             }
85         }
86     }
87 }

```

৮.৪.২ বেলম্যান ফোর্ড অ্যালগরিদম (Bellman Ford Algorithm)

এটিও সিঙ্গল সোর্স শট্টেস্ট পাথ বের করার একটি অ্যালগরিদম এবং এটি ডায়াকস্ট্র্যার অ্যালগরিদমের তুলনায় অনেক সহজ এবং ঝণাত্মক বাহুর মূল্যে এটি কাজ করে। তাহলে আমরা ডায়াকস্ট্র্যার অ্যালগরিদম শিখলাম কেন? কারণ এর কমপ্লেক্সিটি $O(mn)$ যা ডায়াকস্ট্র্যার অ্যালগরিদমের তুলনায় অনেক বেশি। যদি গ্রাফে ঝণাত্মক সাইকেলও থাকে তাহলে এই অ্যালগরিদম তা বুঝতে পারে। ঝণাত্মক সাইকেল হলো গ্রাফের এমন একটি সাইকেল যেখানে বাহুর মূল্যের যোগফল ঝণাত্মক হয়। অর্থাৎ তুমি একটি নোড থেকে শুরু করে বিভিন্ন বাহু হয়ে আবার শুরুর নোডে ফিরে আসবে আর দেখতে পাবে যে তোমার বাহুর মূল্যের যোগফল ঝণাত্মক হয়ে গেছে। এটি কেন সমস্যার কারণ বুঝতে পারছ তো? কারণ হলো, ঝণাত্মক সাইকেলে আছে এমন একটি নোডে তুমি যদি যেতে পারো তাহলে সেই নোডে পৌঁছানোর খরচ তুমি কিন্তু ঝণাত্মক সাইকেল ব্যবহার করে কমাতেই থাকতে পার। সুতরাং ওই সব নোডের সর্বনিম্ন মূল্য আসলে অসংজ্ঞায়িত বা ঝণাত্মক অসীম (negative infinity) বা এরকম অনেক কিছুই বলতে পার। অনেক সমস্যাই আছে যেখানে তোমাকে বলতে বলবে কোন কোন নোড এরকম ঝণাত্মক সাইকেলে আছে বা শুরুর নোড থেকে কোন কোন নোডে যাওয়া যায় যারা ঝণাত্মক সাইকেলে আছে। এসব ক্ষেত্রে আমরা বেলম্যান ফোর্ড অ্যালগরিদম ব্যবহার করতে পারি। খেয়াল কর, ঝণাত্মক সাইকেলে থাকা মানেই শুরুর নোড থেকে ঝণাত্মক অসীম মূল্যে পৌঁছানো নাবে!!

এই অ্যালগরিদমকে দুটি অংশে ভাগ করা যায়।

প্রথম অংশে আমরা শট্টেস্ট পাথ বের করব। প্রথমত আমাদের একটি n দৈর্ঘ্যের $dist$ এর অ্যারে নিতে হবে যার সব উপাদান হবে অসীম কেবল সোর্স হবে 0। এখন আমাদের একটি কাজ n সংখ্যকবার করতে হবে। কাজটি হলো, সব বাহু একে একে নিতে হবে, ধরা যাক একটি বাহু হলো a থেকে b তে এবং তার মূল্য হলো c (যদি গ্রাফটি বাইডিরেকশনাল (bidirectional) হয় তাহলে অন্য দিকের বাহুটিও আলাদাভাবে বিবেচনা করতে হবে)। এখন তোমাকে দেখতে হবে, $dist[b]$ বড় নাকি $dist[a] + c$ বড়। সেই অনুসারে আপডেট করতে হবে। এই ধাপটি n সংখ্যকবার চলালেই তোমাদের শট্টেস্ট পাথ বের হয়ে যাবে। খেয়াল কর, আমরা বাইরের লুপ চালাচ্ছি n সংখ্যকবার আর ভেতরেও বাহু এর লুপ চলছে m সংখ্যকবার সুতরাং আমাদের কমপ্লেক্সিটি হবে $O(mn)$. তোমরা

চাইলে এখনে একটি ভালো অপটিমাইজেশন করতে পার এবং এটি প্রায়ই কাজে লাগে বিশেষ করে যখন মিনকস্ট ম্যাক্সফ্লো (mincost maxflow) সমস্যাতে আমরা বেলম্যান ফোর্ড অ্যালগরিদম ব্যবহার করে থাকি। অপটিমাইজেশনটি হলো, আমরা যখনি দেখব n এর লুপের ভেতরের বাহর লুপে কোনো বাহতে কোনো আপডেট ঘটেনি, তখনি n এর লুপকে break করে ফেলব। কারণ পরের অন্য কোনো লুপে আর কোনো আপডেট হবে না। আর আরেকটি কাজও করতে পার, সেটি হলো, বেলম্যান ফোর্ড অ্যালগরিদম চালানোর আগে বাহ্যগুলোর ক্রম তুমি এলোমেলো করে ফেলতে পার। এতে সুবিধা হলো কেউ যদি বেলম্যান ফোর্ড অ্যালগরিদমের জন্য বাজে কেইস বানায়ও, তুমি বাহর ক্রম পরিবর্তন করে ফেলায় সেটি আর বাজে থাকবে না।

এখন দ্বিতীয় অংশে আসা যাক। দ্বিতীয় অংশে আমরা বের করব গ্রাফে ঝাগাত্মক সাইকেল আছে কিনা। এটি করার জন্য যা করতে হবে তা হলো, আমাদের আগের n এর লুপের ভেতরের অংশ আরেকবার চালাতে হবে। যদি দেখ এই $n + 1$ তম বারে আবারও কোনো বাহু দিয়ে নোডের মূল্য আপডেট করা যায় তাহলেই বুঝবে যে তোমার গ্রাফে ঝাগাত্মক সাইকেল আছে।

তোমাদের সুবিধার জন্য খুব সাধারণ একটি বেলম্যান ফোর্ড অ্যালগরিদমের কোড ৮.৭ এ দেওয়া হলো।

কোড ৮.৭: bellmanford.cpp

```

1 struct Edge {
2     int u, v, w;
3 };
4
5 vector<Edge> E; // weighted edge list
6 int dist[100];
7 int n;
8
9 void bellman_ford(int s) {
10     for (int i = 1; i <= n; i++) {
11         dist[i] = 1000000000;
12     }
13     dist[s] = 0;
14
15     for (int i = 1; i < n; i++) {
16         for (Edge e : E) {
17             if (dist[e.v] > dist[e.u] + e.w) {

```

যাক আমার দুর্বল স্মৃতিশক্তি আমার সঙ্গে দুষ্টুমি করেনি! আমি মিনকস্ট ম্যাক্সফ্লো সমস্যাতে ঝাগাত্মক মূল্যের বাহর জন্য ডায়াকস্ট্র্যাক্টের অ্যালগরিদম ব্যবহার করেছি বহুবার।

```

18         dist[e.v] = dist[e.u] + e.w;
19     }
20 }
21 }
22 }

```

৮.৫ অল পেয়ার শট্টেস্ট পাথ (All pair shortest path) বা ফ্লয়েড ওয়ার্শাল অ্যালগরিদম (Floyd Warshall Algorithm)

আমাদের ডায়াকস্ট্রা'র অ্যালগরিদমের কমপ্লেক্সিটি ছিল $O(m \log n)$ এর মতো। আমরা যদি সব নোড থেকেই ডায়াকস্ট্রা'র অ্যালগরিদম চালাতাম তাহলে অল পেয়ার শট্টেস্ট পাথ বের করতে সময় লাগত প্রায় $O(nm \log n)$ এর মতো। যদি গ্রাফটি খুব একটা ঘন সন্নিবিষ্ট ($m \approx n^2$) না হয় তাহলে n সংখ্যক বার ডায়াকস্ট্রা'র অ্যালগরিদম চালানোই ভালো। সত্যি কথা বলতে যেখানে ফ্লয়েড ওয়ার্শাল অ্যালগরিদম চালাতে হবে সেখানে বার বার ডায়াকস্ট্রা'র অ্যালগরিদম চালালেই হয়ে যাওয়ার কথা। তাহলে আমরা ফ্লয়েড ওয়ার্শাল অ্যালগরিদম শিখব কেন? এর একমাত্র কারণ হলো এর কোড খুবই ছোট ও সহজ। মাত্র পাঁচ লাইন আর সেই পাঁচ লাইনের মধ্যে তিনটি for লুপ। এটি মনে রাখাও খুব সহজ। প্রথমেই আমরা অ্যালগরিদমটি দেখে নেই কোড ৮.৮ এ:

কোড ৮.৮: apsp.cpp

```

1 for(k = 1; k <= n; k++)
2     for(i = 1; i <= n; i++)
3         for(j = 1; j <= n; j++)
4             if(w[i][j] > w[i][k] + w[k][j])
5                 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 কী করা হচ্ছে তা তো বুঝতে পারছ? দেখা হচ্ছে যে, i থেকে j তে যাওয়ার মূল্য কী k হয়ে যাওয়ার মূল্য থেকে ভালো না খারাপ। এর পর আমরা ভালো মূল্য দিয়ে আপডেট করে দেব। তোমরা যারা এখনো ভাবছ যে w এর অ্যারেতে কী আছে তাদের জন্য বলছি, এই অ্যারের প্রাথমিক মান হবে অসীম। যদি তোমার গ্রাফে i ও j এর মধ্যে কোনো বাহু থাকে তাহলে তার মূল্যকে $w[i][j]$ এ রাখতে হবে। যদি দুটি নোডের মাঝে একাধিক বাহু থাকে তাহলে সর্বনিম্নটি নিতে হবে। যদি গ্রাফটি আনডি঱েন্ড হয় তাহলে একই সঙ্গে $w[j][i]$ তেও সেই মান দিয়ে দিবে।

এখন প্রশ্ন হলো এর কমপ্লেক্সিটি কত? খুবই সহজ $O(n^3)$.

আরও একটি প্রশ্ন হলো, খণ্ডাত্মক বাহুর মূল্যে ফ্লয়েড ওয়ার্শাল অ্যালগরিদম কী কাজ করবে? হাঁ করবে। শুধু তাই না, গ্রাফে কোন কোন নোড দিয়ে খণ্ডাত্মক সাইকেল যায় তাও বের করা যাবে। তুমি শুরুতে সব $w[i][i]$ এ ০ নিবে। এর পর ফ্লয়েড ওয়ার্শাল অ্যালগরিদম চালানোর পর যদি দেখ যে কোনো একটি $w[i][i]$ এ খণ্ডাত্মক মান, তার মানে হলো ওই নোড দিয়ে একটি খণ্ডাত্মক সাইকেল গিয়েছে।

৪.৬ ডায়াকস্ট্র্যান্ড, বেলম্যান ফোর্ড, ফ্লয়েড ওয়ার্শাল অ্যালগরিদম কেন সঠিক?

ডায়াকস্ট্র্যান্ডের অ্যালগরিদম কেন সঠিক এটা আসলে ব্যাখ্যা করার কিছু নেই। তুমি প্রতিবার সবচেয়ে কম মূল্যে যাওয়া যায় এমন ভার্টেক্সকে *visited* করছ আর যেহেতু তোমার বাহুর মূল্য অখণ্ডাত্মক সেহেতু আগের *visited* কোনো নোডে আসলে আরও কম খরচে তুমি যেতে পারবে না।

বেলম্যান ফোর্ড অ্যালগরিদমে ভেতরের লুপে কী করছি তা তো বোঝা যায়, যেটা বোঝা যায় না সেটা হলো কেন সেই কাজ n বার করলেই আমরা শর্টেস্ট পাথ পাব! খেয়াল কর, তুমি যদি s হতে শর্টেস্ট পাথ বের করতে চাও সব জায়গার তাহলে প্রতিটি জায়গায় তুমি n এর থেকে কম বাহু দিয়ে পৌঁছাতে পারবে। এখন ভেতরের লুপ দিয়ে কিন্তু আমরা এই কাজটিই করছি। যদি মনে করে থাকো একবার চালালেই তো সব বের হয়ে যাওয়া উচিত। না, এখানে কিন্তু বাহুর ক্রমটি খুব গুরুত্বপূর্ণ। যদি কেউ চায় তাহলে সে বাহুর ক্রম এমনভাবে দিতে পারে যে একবার লুপ ঘুরলেই সব জায়গার শর্টেস্ট পাথ বের হয়ে যাবে, আবার কেউ যদি চায় তাহলে n সংখ্যকবারই ঘোরাতে পারবে।¹

ফ্লয়েড ওয়ার্শাল অ্যালগরিদম কেন সঠিক এটা বোঝা একটু ঝামেলা। এর জন্য যেটা বুঝতে হবে সেটা হলো k এর লুপটা বাইরে কেন?² এখানের k এর লুপকে বেলম্যান ফোর্ড অ্যালগরিদমের বাইরের লুপের মতো ভাবলে হবে না। ভেতরের দুটি লুপ n সংখ্যকবার ঘুরানো এর উদ্দেশ্য না, এর উদ্দেশ্য হলো i হতে j তে যাওয়ার সময় যদি k দিয়ে যাওয়া হয় তাহলে সেটা ভালো হয় কিনা এটা বোঝা। আরও ভালো করে বলতে, যদি বাইরের লুপ k বার ঘুরে এর মানে হবে, i হতে j পর্যন্ত শুধু $1, 2, \dots, k$ দিয়ে গেলে সবচেয়ে কম যত মূল্যে যাওয়া যায় তা $w[i][j]$ তে থাকবে। সুতরাং আমরা যদি n পর্যন্ত লুপ চালাই তাহলে আসলে শর্টেস্ট পাথ পেয়ে যাব।

এখন অনেকে ভাবতে পারে যে- বুঝলাম $k - i - j$ কেন সঠিক কিন্তু $i - k - j$ বা $i - j - k$ কেন সঠিক না? এই দুটি কেন সঠিক না সেটার জন্য আমি কেইস দিচ্ছি। তোমরা নিজেরা চিন্তা করে দেখবে কেন এই দুটি উদাহরণে $i - k - j$ বা $i - j - k$ সঠিকভাবে কাজ করে না। ধরা যাক

¹আসলে $n - 1$ সংখ্যকবার ঘুরালেই হয়। কোনো এক ঐতিহাসিক কারণে আমরা সবসময় n বার বলে থাকি।

²আগে আমি বেশ কয়েকবার এই ভুল করতাম, প্রায় সময় k এর লুপ ভেতরে দিয়ে থাকতাম ভাবতাম একই তো কথা! কিন্তু এক কথা না।

আমাদের গ্রাফে দুটি শেটেস্ট পাথ হলো: 1 – 5 – 3 – 2 এবং 5 – 4 – 3 – 2, তাহলে এই দুই
কেইসে আমাদের পরিবর্তিত সমাধান কাজ করবে না।

৮.৭ আর্টিকুলেশন ভার্টেক্স (Articulation vertex) বা আর্টিকুলেশন বাহু (Articulation edge)

একটি আনডিরেক্টেড গ্রাফ (undirected graph) এ যদি কোনো নোডকে মুছে ফেললে
গ্রাফটি ডিসকানেক্টেড (disconnected) হয়ে যায় তাহলে তাকে আর্টিকুলেশন ভার্টেক্স
(articulation vertex) বলে। একইভাবে যদি কোনো বাহুকে মুছে ফেললে গ্রাফটি
ডিসকানেক্টেড হয়ে যায় তাহলে তাকে আর্টিকুলেশন বাহু (articulation edge) বা আর্টিকুলেশন
ব্রিজ (articulation bridge) বলে। কখনও কখনও এদের কাটনোড (cut node) বা কাটএজ
(cut edge) ও বলে। DFS ব্যবহার করে খুব সহজেই আর্টিকুলেশন ভার্টেক্স বা বাহু বের করে
ফেলা যায়। DFS এর একটি শক্তিশালী প্রয়োগ হলো এটি। এটা করার জন্য আমাদের কয়েকটি
জিনিসের সঙ্গে পরিচিত হতে হবে: `dfsStartTime`, `dfsEndTime` এবং `low`. আর্টিকুলেশন
ভার্টেক্স বা বাহু বের করতে এদের সবগুলোই যে দরকার তা নয়, কিন্তু এদের ব্যবহার করে আমরা
বেশ জটিল জটিল সমস্যা সমাধান করে ফেলতে পারি। বিশেষ করে ইনফরমেটিক্স অলিম্পিয়াডে
এধরনের অনেক সমস্যা দেখা যায়। যতদূর মনে পরে 2006 সালের IOI এ এরকম একটি সমস্যা
ছিল। যাই হোক, `dfsStartTime` ও `dfsEndTime` খুবই সহজ জিনিস। তুমি `dfsTime`
বলে একটি ভ্যারিয়েশন রাখবে যার প্রাথমিক মান হবে 0. এর পর তোমরা DFS করার সময়
যখনই কোনো একটি নতুন *unvisited* নোডে আসবে তখনই `dfsTime` কে এক বাড়াবে আর
`dfsStartTime` এ `dfsTime` এর বর্তমান সময় নোট করবে আর কোনো নোডের সব চাইল্ড
(child) এর *visit* শেষ হয়ে গেলে `dfsTime` এর বর্তমান মান `dfsEndTime` এ মার্ক করে
রাখবে। এটা তো গেল `dfsStartTime` আর `dfsEndTime`. `low` একটু জটিল জিনিস।
আমরা তো জানি DFS পুরো গ্রাফে একটি ট্রি এর মতো করে আগায়। মনে কোনো নোডের যদি
unvisited অ্যাডজাসেন্ট ভার্টেক্স থাকে তাহলে আমরা সেটা *visit* করি (নিচে নামি) আর যদি
কোনো *unvisited* অ্যাডজাসেন্ট ভার্টেক্স না থাকে তাহলে ফেরত যাই (প্যারেন্ট (parent) এ
ফেরত যাই)। যদি সেই নোড থেকে কোনো *visited* নোডে যাওয়া যায় তা অবশ্যই এর অ্যানসেস্টর
(ancestor) হবে অর্থাৎ ওই নোড থেকে রুটের পাথের মধ্যেই থাকবে অথবা তার নিজের সাবট্রি
(subtree) তে থাকবে। কেন? চলো চিন্তা করে দেখা যাক। মনে কর DFS করার সময় আমরা A
হতে ইতোমধ্যেই *visited* হওয়া একটি নোড B তে বাহু পেয়েছি। ধরা যাক B উপরে বলা দুরন্তের
মাঝে কোনোটিতেই পরে না। তাহলে B কী ধরনের নোড? এটি হবে A এর কোনো অ্যানসেস্টর কোনো
(ধরা যাক C) এর যেই সাবট্রি তে A আছে সেটি বাদে অন্য যেসব সাবট্রি আছে তাদের কোনো
একটির অংশ। যদি তাই হয় তাহলে C থেকে যখন B কে *visit* করা হয়েছে তখন সেখান থেকে
আমরা A তেও আসতাম। অর্থাৎ A আর B দুজনই C এর একই সাবট্রিতে থাকার কথা ছিল।
যেহেতু তানা, তাই আমরা বলতে পারি B হয় A এর অ্যানসেস্টর অথবা A এর সাবট্রি এর অংশ।

$low[u]$ হলো // নোড বা u এর সাবট্রিতে থাকা নোডগুলো থেকে সবচেয়ে উপরে (রুটের কাছের) যেই নোডে যাওয়া যায় তার $dfsStartTime$.

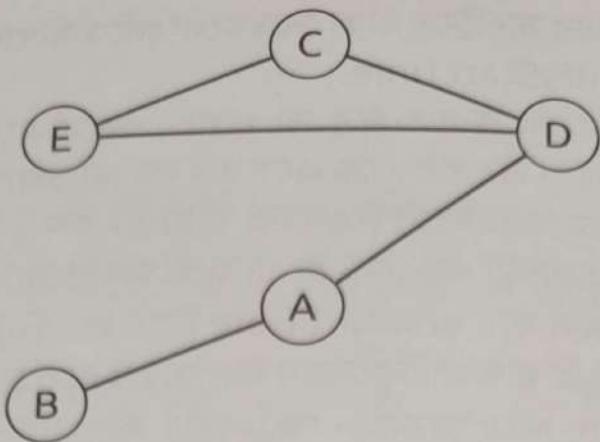
$low[u]$ বের করার জন্য যা করতে হবে তা হলো: প্রথমে একে $dfsStartTime$ দিয়ে ইনিশিয়ালাইজেশন করতে হবে। এর পর একে একে এর সব অ্যাডজাসেন্ট নোডগুলোকে দেখতে হবে। ধরা যাক v হলো u এর অ্যাডজাসেন্ট কোনো ভাট্টের। যদি v ইতোমধ্যেই *visited* হয়ে যায় তাহলে হয় v হবে u এর সাবট্রি এর কেউ বা প্যারেন্ট অথবা অ্যানসেন্ট। যদি অ্যানসেন্ট রয়ে কিন্তু প্যারেন্ট না হয় তাহলে তার $dfsStartTime$ দিয়ে $low[u]$ কে আপডেট করতে হবে। আর যদি প্যারেন্ট হয় এবং তুমি যদি আর্টিকুলেশন বাহু বের করতে চাও তাহলে একটু সতর্ক হতে হবে। প্যারেন্ট থেকে যেই বাহু দিয়ে আমরা u তে এসেছি যদি সেই বাহু হয় এটি তাহলে আমরা কোনো কিছু করব না, আর যদি এটি ভিন্ন বাহু হয় তাহলে আগের মতো আপডেট করতে হবে। যদি আমাদের গ্রাফ মাল্টিএজ¹ গ্রাফ না হয় তাহলে আমাদের এটা নিয়ে ভাবার কিছু নেই। এখন যদি আমাদের v নোডটি আগে থেকে *visited* না হয় তাহলে তার DFS করতে হবে রিকার্সিভ উপায়ে এবং $low[v]$ দিয়ে $low[u]$ কে আপডেট করতে হবে। এখানে আপডেট করা মানে হলো সর্বনিম্ন মানটি বের করা। এই $low[v]$ ভ্যালু কিন্তু কোনো একটি নোডের $dfsStartTime$, আমাদের এই মান যত কম হবে ততই সেই নোড রুটের কাছাকাছি হবে।

এখন আমাদের সব দরকারি মান বের করা হয়ে গেছে। এই মানগুলো দেখে আমরা বলতে পারব কোনটা আর্টিকুলেশন ভাট্টের আর কোনটা আর্টিকুলেশন বাহু। আর্টিকুলেশন বাহু বের করা তুলনামূলক সহজ। প্রতিটি বাহু নাও, মনে করা যাক $u - v$. যদি দেখো $low[u] < low[v]$ তাহলে এটি আর্টিকুলেশন বাহু। কারণ v এর সাবট্রি হতে কোনো back edge নেই যা u বা u এর উপরের সঙ্গে যুক্ত। অর্থাৎ $u - v$ বাহু কেটে দিলে তারা ডিসকানেক্টেড হয়ে যাবে।

আর্টিকুলেশন নোড বের করা একটু কঠিন। প্রথমত চিন্তা করে দেখো u যদি রুট হয় অর্থাৎ u যদি DFS এর শুরুর নোড হয় তাহলে কী হবে? দেখতে হবে যে DFS ট্রি তে এর একের বেশি সাবট্রি বা চাইল্ড আছে কিনা। থাকলে u একটি আর্টিকুলেশন নোড। এখন u যদি রুট নোড না হয় সেইক্ষেত্রে চিন্তা করা যাক। মনে কর $u - v$ একটি বাহু। এটি যদি back edge হয় তাহলে u আর v কে আলাদা করা যাবে না। ধরা যাক এটি back edge না, অর্থাৎ DFS ট্রি এর বাহু। যদি দেখো $dfsStartTime[u] \leq low[v]$ তার মানে হলো আমরা যদি u কে ডিলিট করে দেই তাহলে v ডিসকানেক্টেড হয়ে যাবে (খেয়াল রাখতে হবে v যেন রুট না হয়, চিন্তা করে দেখতে পারো কেন রুটের ক্ষেত্রে এটি কাজ করবে না)। এভাবে আমরা আর্টিকুলেশন ভাট্টের বের করে ফেলতে পারি।

একটি উদাহরণ দেওয়া যাক। চিত্র ৮.২ এ আমরা A হতে DFS শুরু করব আর আমাদের অ্যাডজাসেন্ট লিস্ট হবে নোডগুলোর নামের ক্রমে। মানে D এর অ্যাডজাসেন্ট লিস্ট হবে A, C, E . তাহলে প্রতিটি নোডের $dfsStartTime$ কত হবে? প্রথমে আমরা আছি A তে, সুতরাং এর $dfsStartTime$ আর low হবে ১. এর পর আমরা যাব B তে এর $dfsStartTime$ ও low হবে ২. এর পর আমরা যাব D তে, এখানে $dfsStartTime$ ও low হবে ৩. এর পর যাওয়া হবে C তে যাব $dfsStartTime$ ও low হবে ৪। এর পর E তে এসে এর $dfsStartTime$ ও low হবে ৫. তবে পরম্পরাগত যখন আমরা $E - D$ back edge আবিষ্কার করব তখন E এর low কমে D

¹মাল্টিএজ (multi edge) গ্রাফ মানে দুটি নোডের মাঝে এখানিক বাহু থাকতে পারে।



নকশা ৮.২: আর্টিকুলেশন নোড ও বাহু (Articulation node and edge)

এর dfsStartTime অর্থাৎ 3 হয়ে যাবে। C তে ফিরে যাওয়ার পর আমরা C এর low কে E এর low দিয়ে আপডেট করে 3 করে ফেলব। D বা A এর low এর কোনো আপডেট হবে না। তাহলে নোডগুলোর dfsStartTime ও low হলো $A(1, 1), B(2, 2), C(4, 3), D(3, 3), E(5, 3)$ । এখন যদি $A-B$ আর $A-D$ বাহুগুলো দেখো তাহলে বুঝবে এরা আর্টিকুলেশন বাহু। কেন? $A-B$ ক্ষেত্রে $\text{dfsStart}[A] < \text{low}[B]$ একইভাবে $A - D$ এর ক্ষেত্রেও তবে, অন্য বাহুগুলোর নোডগুলো বের করা যাক। A যেহেতু রুট আর এর একাধিক সাবট্রি আছে তাই এটি চোখ বন্দ করে আর্টিকুলেশন নোড। তোমরা যদি $D-C$ বাহু দেখো তাহলে $\text{dfsStart}[D] \leq \text{low}[C]$ পারে তাই D একটি আর্টিকুলেশন নোড। অন্য কোনো বাহুর ক্ষেত্রে এই শর্ত সত্য না, তাই আর কোনো আর্টিকুলেশন নোড নেই।

৮.৮ অয়লার পাথ (Euler path) এবং অয়লার সাইকেল (euler cycle)

আমরা প্রথমে শুধু আনডিরেস্টেড গ্রাফ নিয়ে ভাবব। অয়লার পাথ (Euler¹ path) হলো কোনো একটি গ্রাফে যদি একটি ভাট্টেক্স থেকে যাত্রা শুরু করে, প্রতিটি বাহু ঠিক একবার করে ঘুরে কোনো একটি ভাট্টেক্সে যাত্রা শেষ করা যায় তাহলে তাকে অয়লার পাথ বলে। আর যদি শুরু ও শেষের ভাট্টেক্স একই হয় তাহলে তাকে অয়লার সাইকেল (euler cycle) বা অয়লার সার্কিট (euler circuit) বলে। কোনো একটি গ্রাফে অয়লার পাথ বা সাইকেল আছে কিনা তা বের করা খুবই সহজ। প্রথম শর্ত হলো গ্রাফকে কানেক্টেড হতে হবে। এখন যদি সবগুলো নোডের ডিগ্রী (degree) জোড় হয় তাহলে গ্রাফে অয়লার সাইকেল আছে (অয়লার সাইকেল থাকা মানে কিন্তু অয়লার পাথও থাকা, কিন্তু উল্টোটি সত্য নয়)। আর যদি এই গ্রাফের শুধুমাত্র দুটি নোড বেজোড় ডিগ্রী (odd

¹Euler এর উচ্চারণ অয়লার।

degree) ওয়ালা হয় তাহলেও গ্রাফটিতে অয়লার পাথ থাকবে তবে সেক্ষেত্রে আমাদের অবশ্যই ওই দুটি নোডের কোনো একটি থেকে যাত্রা শুরু করতে হবে। এরকম হওয়ার কারণ হলো, শুরু আর শেষের নোড বাদে বাকি সব নোডের ক্ষেত্রে আমরা কিন্তু একবার চুকলে বের হতে হয় সুতরাং আমাদের বাহুগুলো জোড়ায় জোড়ায় থাকে বা বলতে পারি মাঝের সব ভাট্টের গুলোর ডিগ্রী হবে জোড়। এখন যদি সাইকেল হয় তাহলে যেখান থেকে শুরু করেছি সেখানেই শেষ করেছি সুতরাং সেই নোডের ডিগ্রীও জোড় হবে। কিন্তু যদি এটি সাইকেল না হয়ে পাথ হয় তাহলে দেখ শুরু আর শেষ ভাট্টের আলাদা এবং তাদের ডিগ্রী হবে বিজোড়। এটি তো আমরা প্রমাণ করলাম যে অয়লার পাথ বা সাইকেল হলে এরকম বৈশিষ্ট্য থাকবে। কিন্তু এরকম বৈশিষ্ট্য থাকলেই যে অয়লার পাথ বা সাইকেল হবে তা কিন্তু প্রমাণ করিনি। সেটি প্রমাণ করাও কিন্তু খুব একটা কঠিন না। তোমরা গাণিতিক আরোহ বা ইনডাকশন (induction) ব্যবহার করে খুব সহজেই প্রমাণ করতে পারবে।

এখন কোড করে কীভাবে আমরা অয়লার পাথ বা সাইকেল বের করতে পারব? এটিও খুব সহজ, DFS এর মতো তুমি কোনো একটি ভাট্টের থেকে যাত্রা শুরু কর। তবে এখানে আমাদের ভাট্টের জন্য কোনো *visited* থাকবে না, থাকবে বাহুর জন্য। খেয়াল রেখো কোনো একটি বাহু কিন্তু দুই দিকের কোনো একদিক থেকেই *visit* করা যায়। এখন কোনো একটি ভাট্টের আমরা দাঁড়িয়ে দেখব যে এর থেকে বের হওয়া কোন কোন বাহু এখনো *visited* হয়নি, যদি এমন কোনো বাহু বাকি থাকে তাহলে সেটি দিয়ে বের হয়ে যাব এবং আগের মতোই বুরতে থাকব। আর যদি দেখি এই ভাট্টের সঙ্গে লাগানো সব বাহুই *visited* হয়ে গেছে তাহলে এই ভাট্টেরকে প্রিন্ট করে দেব। খেয়াল কর এই পদ্ধতিতে কিন্তু একটি ভাট্টের কিন্তু অনেকবার *visit* হতে পারে।

এবার দেখা যাক ডিরেক্টেড গ্রাফ (directed graph) এ কীভাবে আমরা অয়লার পাথ বা সাইকেল বের করতে পারি। আসলে আমি এই অংশটি লেখার আগে এই জিনিসের সম্মুখীন হইনি বা হলেও মনে নেই। সুতরাং আমি একটু ইন্টারনেট ঘেঁটেযুটে যা দেখলাম তা হলো ডিরেক্টেড গ্রাফের অয়লার পাথ বা সাইকেল বের করা প্রায় পুরোপুরি আনডিরেক্টেড গ্রাফের মতো। আমরা কোনো একটি নোড থেকে শুরু করব, এর বহিগামী বা আউটগোয়িং (outgoing) কোনো বাহু দিয়ে বের হব যতক্ষণ কোনো না কোনো আউটগোয়িং বাহু (outgoing edge) বাকি থাকে। যখনই শেষ হয়ে যাবে আমরা প্রিন্ট করে দিব এবং আগের নোডে ফিরে যাব, ঠিক আগের DFS এর মতো। আশা করি বুরতে পারছ যে আমাদের অয়লার পাথ বা সাইকেল এর ঠিক উল্টো ক্রমে প্রিন্ট হয়েছে! আরেকটি জিনিস, তা হলো পাথ প্রিন্ট করার আগে প্রতিটি নোডের আউটডিগ্রী (outdegree) আর ইনডিগ্রী (indegree)।^১ একটু দেখে নিতে হবে। প্রতিটি নোডের ইনডিগ্রী আর আউটডিগ্রী সমান হতে হবে, অথবা কেবল একটি নোডের ইনডিগ্রী, আউটডিগ্রী হতে এক বেশি হতে পারবে এবং একটি নোডের আউটডিগ্রী, ইনডিগ্রী এর থেকে এক বেশি হতে পারবে। তাহলে তারা যথাক্রমে পাথের শেষ ও শুরু হবে। কিন্তু সবার যদি ইনডিগ্রী আর আউটডিগ্রী সমান হয় তাহলে যেকোনো নোড থেকে শুরু করে সেখানে ফেরত আসা যাবে। তবে আগের মতোই কানেক্টেড ব্যাপারটি একবার

^১ সাধারণ ডিগ্রী মানে তো জানোই? একটি আনডিরেক্টেড গ্রাফে একটি নোডের সঙ্গে কতগুলো বাহু লাগানো আছে। ইনডিগ্রী (indegree) ও আউটডিগ্রী (outdegree) হলো ডিরেক্টেড গ্রাফের পদ। ইনডিগ্রী মানে হলো কোনো একটি নোডে কতগুলো ডিরেক্টেড বাহু (directed edge) চুকেছে, আর আউটডিগ্রী মানে হলো কোনো একটি নোড হতে কতগুলো ডিরেক্টেড বাহু বের হয়েছে।

দেখে নিতে হবে। খেয়াল রাখতে হবে যেন, শুরুর নোড থেকে যেন সব জায়গায় যাওয়া যায় আর শেষের নোডে যেন সব জায়গা থেকে আসা যায়।

৮.৯ টপোলজিক্যাল সর্ট (Topological sort)

একটি ডিরেক্টেড গ্রাফে নোডগুলোকে অমন্ত্রমে সাজাতে হবে যেন, যদি u থেকে v তে কোনো ডিরেক্টেড বাহু থাকে তাহলে সর্টেড অ্যারেতে u , v এর আগে থাকে। এটি তখনই সন্তুষ্ট যখন গুই গ্রাফে কোনো ডিরেক্টেড সাইকেল (directed cycle) থাকবে না। সাইকেল থাকলে তো ওই সাইকেল এর নোডগুলোকে তুমি কোনোভাবেই অমন্ত্রমে সাজাতে পারবে না তাই না? আমরা এই অ্যালগরিদম ব্যবহার করে কোনো ডিরেক্টেড গ্রাফে সাইকেল আছে কিনা তাও বের করে ফেলতে পারব।

আমরা দুভাবে টপোলজিক্যাল সর্ট (topological sort) করতে পারি। একটি হলো BFS দিয়ে আরেকটি DFS দিয়ে। আমার কাছে BFS দিয়ে তুলনামূলক সহজ মনে হয়, এছাড়াও এখানে প্ট্যাক গুভারফ্লো নিয়ে মাথা ঘামাতে হয় না। কিন্তু DFS একটু তুলনামূলকভাবে ছোট হয়ে থাকে। প্রথমে দেখা যাক আমরা BFS দিয়ে কীভাবে সমাধান করতে পারি।

প্রথমে আমাদের একটি ইনডিগ্রী এর অ্যারে লাগবে যেখানে প্রতিটি নোডের ইনডিগ্রী লেখা থাকবে। এখন একটি কিউতে সেসব নোড রাখতে হবে যাদের ইনডিগ্রী শূন্য। এবার কিউ থেকে একে একে নোড তোলার পালা। একটি করে নোড তুলব আর তার থেকে যেসব বাহু বের হয়ে গেছে তাদের দেখে দেখে অন্য প্রান্তের নোডের ইনডিগ্রী কমিয়ে দিব। যদি অন্য প্রান্তের ইনডিগ্রী শূন্য হয়ে যায় তাহলে তাকে কিউতে ঢুকিয়ে দিব। এভাবে যতক্ষণ না কিউ ফাঁকা হয়ে যায় ততক্ষণ চলতে থাকবে। কিউতে নোড যেই ক্রমে ঢুকেছে সেটিই কিন্তু টপোলজিক্যাল সর্টের ক্রম। তবে একটি জিনিস, যদি কিউ ফাঁকা হয়ে যাওয়ার পরও দেখ যে কোনো নোডের ইনডিগ্রী এখনো শূন্য হয়নি তার মানে গ্রাফটিতে সাইকেল আছে। এটি কিন্তু খুবই স্বজ্ঞালক্ষ একটি অ্যালগরিদম। কেন কাজ করছে তা খুব সহজেই বোবা যায়। যেমন শুরুতে আমরা ইনডিগ্রী শূন্যওয়ালা নোডগুলোকে কিউতে ঢুকিয়েছি কারণ এসব নোড কারো উপর নির্ভর করে না। এখন এসব নোডকে যখন আসলেই নিয়ে ফেলব এর মানে এর উপর যারা নির্ভর করে তাদের নির্ভরতা এক করে করে যাবে। যদি তাদের নির্ভরতা শূন্য হয়ে যায় তার মানে আমরা চাইলে এই নোডকে এখন নিতে পারব, সেজন্য আমরা একে কিউতে ঢুকাই। সহজ না?

এবার আসা যাক DFS দিয়ে কীভাবে এটা সমাধান করা যায়। ধরা যাক T হলো আমাদের রেজাল্টের একটি অ্যারে, আর *visited* আরেকটি অ্যারে যার প্রাথমিক মান 0। এখন আমরা একে একে প্রতিটি নোড দেখব আর যদি সেই নোডের *visited* এর মান এখনো 2 না হয়ে থাকে তাহলে তার DFS কল করব। DFS এর প্রথমেই আমরা যা করব তা হলো এর *visited* এর মান 1 করে দেব। এর পর এখান থেকে যেই যেই ডিরেক্টেড বাহু বের হয়েছে তাদের দেখব। যদি অন্য প্রান্তের নোড *visited* এর মান 1 হয়ে থাকে এর মানে হলো আমরা একটি সাইকেল পেয়ে গেছি সুতরাং নোডগুলোর কোনো টপোলজিক্যাল ক্রম (topological order) নেই। যদি *visited* এর মান 2

য় তাহলে আমাদের করার কিছু নেই। আর যদি 0 হয় তাহলে আমরা ওই নোডের জন্য DFS কল করব। এভাবে অ্যাডজাসেন্ট প্রতিটি নোডের জন্য প্রসেসিং শেষ হলে আমরা সেই নোডের *visited* কে 2 করে দেব এবং তাকে T তে ঢুকিয়ে দেব এবং DFS থেকে রিটার্ন করব। এভাবে সব নোডের জন্য DFS শেষ হয়ে গেলে আমরা T তে টপোলজিক্যাল সর্ট উল্টো অর্ডারে পাবো। কেন? খেয়াল কর আমরা T তে কোনো নোড ঢুকানোর আগে এর সব ডিসেন্ডেন্ট (descendent) দের জন্য DFS কল করি। অর্থাৎ তাদেরকে আগে T তে ঢুকাই আর এর পর নিজেকে। সুতরাং আমাদের গ্রাফে যদি কোনো সাইকেল না থাকে তাহলে এই পদ্ধতি নিঃসন্দেহে কাজ করবে। এখন কথা হলো সাইকেল থাকলে আমরা কীভাবে বুঝতে পারব? এটাও সহজ DFS করতে করতে যদি আমরা আগের *visited* নোডে এসে পড়ি তাহলেই সাইকেল আছে। কথাটা কিন্তু পুরোপুরি সত্য না। কখন সত্য না সেটা চিন্তা করা যাক। ধর আমাদের গ্রাফে একটি বাহু আছে A হতে B . এখন প্রথমে যদি B এর DFS কর, এরপর A এর তাহলে কিন্তু তুমি $A - B$ বাহু দিয়ে একটি *visited* নোডে এসে পড়ো। অথচ এখানে কোনো সাইকেল নেই। আরও একটি উদাহরণ দেখা যাক, তিনটি বাহুর গ্রাফ $A - B, A - C, C - B$. এখানে মনে কর প্রথমে A এর DFS কল করা হয়, এরপর এখান থেকে আমরা B তে যাই, একে *visit* করে ফিরে এসে যখন C তে এসে আবারো B কে *visit* করতে যাই তখন কিন্তু আমরা একটি *visited* নোড পেয়ে যাব, অথচ এখানে কোনো সাইকেল নেই। তাহলে উপায় হলো, বর্তমানে DFS হচ্ছে, এখনও ফিরে আসো নাই এরকম কোনো নোডে যদি যেতে পারো তাহলেই সাইকেল আছে। যেমন আগের দুটি উদাহরণে আমরা DFS করে ইতোমধ্যেই B হতে ফিরে এসেছি। উপরে এই ফিরে আসাকেই 2 দিয়ে *visited* অ্যারেতে লিখে রাখা হচ্ছে। আর 1 মানে হচ্ছে এখনও DFS চলছে। যেমন মনে কর আমাদের গ্রাফের বাহু গুলো হলো $A - B, B - C, C - A$. এখন A হতে DFS শুরু করে আমরা B , এর পর C , এর পর আবারো A তে আসবো। এখানে দেখবো যে A এর এখনও DFS শেষ হয় নাই অথচ আমরা A তে এসেছি! ব্যাস তুম বুঝে যাবা এখানে সাইকেল আছে।

৮.১০ স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Strongly Connected Component - SCC)

একটি ডিরেক্টেড গ্রাফে যখন একটি নোড u থেকে v তে যাওয়া যায় এবং একই সঙ্গে v থেকে u থেকে যাওয়া যায় তখন আমরা বলব ওই দুটি 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 কারণ তুমি w থেকে v তে যেতে পার আর v থেকে u তে যেতে পার, একইভাবে u থেকেও v হয়ে তুমি w তে যেতে পার। সুতরাং u আর w একই SCC তে। একটু চিন্তা করলে বুঝবে যে এর মানে দাঁড়ায় তুমি পুরো গ্রাফকে আসলে অনেকগুলো SCC তে ভাগ্তে পারবে যেন কোনো একটি নোড কোনো একটি এবং কেবলমাত্র একটি SCC এর অংশ। যেমন ধর, যদি আমাদের বাহুগুলো হয়: $(u, v), (v, w), (w, u)$ তাহলে এখানে কেবলমাত্র একটি SCC: $\{u, v, w\}$. আবার ধরা যাক আমাদের বাহুগুলো হলো: $(u, v), (w, v)$ তাহলে কিন্তু তিনটি SCC: $\{u, v\}, \{w\}$.

$\{u\}, \{v\}, \{w\}$. আবার $(u, v), (v, u), (w, u), (w, v)$ হলে দুটি SCC: $\{u, v\}, \{w\}$.

এখন আমরা SCC বের করার অ্যালগরিদম শিখব। এর জন্য দুটি বহুল প্রচলিত অ্যালগরিদম আছে। একটি হলো কোসারাজু অ্যালগরিদম (Kosaraju's algorithm) বা 2-DFS অ্যালগরিদম, আরেকটি হলো টারজানের অ্যালগরিদম (Tarjan's algorithm)। আমি আসলে শুধু প্রথমটিই জানি। এটি করা বা বলা সহজ, কিন্তু আমার কাছে মনে রাখা বেশ কষ্টকর মনে হয় এবং বোঝাও একটু কষ্টকর মনে হয়। প্রথমে একটি *visited* অ্যারে নাও এবং সব নোডকে *unvisited* করে দাও। এখন একে একে নোডগুলো যাচাই কর, যদি কোনো *unvisited* নোড পাও তাহলে তার জন্য DFS কল কর। DFS এর ভেতরে যা করবে তা হলো, ওই নোডকে *visited* করে দিবে আর ওই নোড থেকে যদি কোনো *unvisited* নোডে যাওয়া যায় তাহলে তার DFS কল করবে। অ্যাডজাসেন্ট সব নোড *visited* হয়ে গেলে DFS থেকে রিটার্ন করার আগে একটি লিস্টের শেষে (ধরা যাক তার নাম L) ওই নোডকে পুশ করে যাবে। তাহলে সব নোড *visited* হয়ে গেলে কিন্তু আমাদের ওই L লিস্টে সব নোড থাকবে। এবার যেটি করতে হবে তা হলো ওই গ্রাফের সব বাহকে উল্টো করে দিতে হবে (আসলে তুমি শুরুতেই দুটি অ্যাডজাসেন্সি লিস্ট বানিয়ে নিবে একটি ঠিক দিকে আরেকটি উল্টো দিকে)। তুমি যেই নোডের লিস্ট L বানিয়ে ছিলে ওটাকেও উল্টো করতে হবে। এবার আমরা আরও একটি DFS করব। আমাদের আবার সব নোডকে *unvisited* করে দিতে হবে। এখন আমরা L এর সামনের দিক থেকে একটি একটি করে নোড নিব এবং সে যদি *visited* না হয়ে থাকে তার জন্য DFS কল করব। খেয়াল রেখ, এবার DFS এ কিন্তু আমরা উল্টো গ্রাফ ব্যবহার করছি। এই DFS এর সময় যেই যেই নোড *visited* হবে তারা একটি SCC¹। এভাবে আমরা সব SCC পেয়ে যাব। বেশ জটিল তাই না? একটি ছোট উদাহরণ দেখা যাক। মনে কর তিনি নোডের গ্রাফ, বাহগুলো হলো $A - B, B - C, C - B$. বুঝাই যাচ্ছে আমাদের SCC গুলো হলো $\{A\}, \{B, C\}$. ধরা যাক আমরা A হতে প্রথম DFS করছি। এখান থেকে আমরা যাব B তে, এবং সবশেষে C তে। যখন আমরা DFS থেকে ফিরে আসবো তখন এদেরকে একে একে L এ ঢুকাই, তাহলে L এ নোডগুলোর অর্ডার হলো $\{C, B, A\}$. এবার গ্রাফ উল্টানোর পালা। আমাদের নতুন গ্রাফের বাহগুলো হবে $B - A, C - B, B - C$. আর L ও উল্টিয়ে হয়ে যাবে $\{A, B, C\}$. এখন আমরা L এর শুরু থেকে নোড নিবো। প্রথমে A , আমরা এর DFS কল করলে আর কোনো নোড *visit* হবে না (এখন কিন্তু $A - B$ বলে আর কোনো বাহু নেই)। সুতরাং এই বারে যেহেতু শুধু A ই *visit* হয় তাই এটি একটি SCC. এবার আমরা B হতে DFS শুরু করব, এবং এবারে B এবং *Cvisit* হয়। তাই এরা দুজনে একটি SCC.

এই বই লিখতে গিয়ে আমি টারজানের SCC অ্যালগরিদম দেখলাম। খুব একটা কঠিন না। এই অ্যালগরিদমটি কিছুটা আর্টিকুলেশন ব্রিজ বা ভার্টেক্স বের করার মতো। তবে মনে রাখতে হবে এবার আমরা ডিরেক্টেড গ্রাফ নিয়ে কাজ করছি। সবসময়ের মতো 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 দিয়ে আপডেট করব।

¹তোমরা চাইলে DFS এর ফাংশনকে একটি সংখ্যামান দিয়ে দিতে পার যে এটি হলো SCC এর নম্ব, এটি দিয়ে *visited* কে মার্ক করতে, বা চাইলে একটি লিস্টও প্যারামিটারে দিয়ে দিতে পার যেন *visited* নোডগুলো ওই লিস্টে রাখা হয়।

এরপর time এর মান এক বাড়িয়ে দেব। সেই সঙ্গে এই নোডকে একটি স্ট্যাকে পুশ করব। এবার এই নোডের অ্যাডজাসেন্ট যত নোড আছে তাদের একে একে দেখতে হবে। আমরা যদি এমন একটি অ্যাডজাসেন্ট নোড পেয়ে যাই যা আগে থেকেই visited (ঠিক visited না, DFS চলছে এমন। টপোলজিক্যাল স্টের সময় আমরা যেভাবে $0 - 1 - 2$ দিয়ে মার্ক করেছিলাম সেরকম), তাহলে আমরা বর্তমান নোডের low কে উপর নোডের startTime দিয়ে আপডেট করব (সর্বনিম্ন নিব) আর যদি unvisited হয় তাহলে তার জন্য DFS কল করব এবং DFS শেষে low এর মান আপডেট করব। এভাবে সব প্রতিবেশী নোডের জন্য প্রসেসিং শেষ হলে আমাদের দেখতে হবে যে তার low এর মান startTime এর মানের সমান কিনা, অর্থাৎ আমরা তাকে দিয়ে বা তার প্রতিবেশী দিয়ে তার থেকেও আগের কোনো নোডে 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 স্ট্যাক থেকে নোড তুলতেই থাকব যতক্ষণ না আমাদের বর্তমান নোড পাই। এই সব নোডগুলো হলো একটি SCC.

৪.১১ 2-satisfiability (2-sat)

$(a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c)$ এই সমীকরনের একটি সমাধান হতে পারে: $a = 1, b = 0, c = 0$. কিন্তু অনেক সময় কোনো সমাধান নাও থাকতে পারে, যেমন: $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$. শুধু ভাবে বলতে গেলে a, b, c এগুলোকে ভ্যারিয়েবল বলা হয় আর দুটি করে ভ্যারিয়েবল বা তাদের \neg ('not' অপারেশন) নিয়ে \vee ('or' অপারেশন) করে যে এক একটি জোড়া বানানো হয় তাদের ক্লজ (clause) বলে। অনেকগুলো ক্লজের \wedge ('and' অপারেশন) করে বড় সমীকরণ বা স্টেটমেন্ট (statement) তৈরি করা হয়। আমাদের লক্ষ্য হলো, ভ্যারিয়েবলগুলোতে এমন মান বসানো যায় কিনা যেন আমাদের স্টেটমেন্টটি সঠিক হয়। যেহেতু প্রত্যেকটি ক্লজ দুটি করে পদ থাকে সেজন্য একে 2-sat সমস্যা বলা হয়। তোমাদের জানার জন্য বলে রাখি 3-sat সমস্যা হলো NP আর দুনিয়ার মোটামুটি অনেক সমস্যা এদিক ওদিক করে 3-sat এ কল্পনাট করা যায়।

যদি আমাদের স্টেটমেন্টে n সংখ্যক ভ্যারিয়েবল থাকে তাহলে আমাদেরকে একটি $2n$ নোডের ডিরেক্টেড গ্রাফ বানাতে হবে। x দিয়ে যদি একটি ভ্যারিয়েবল থাকে তাহলে x এর জন্য একটি নোড আরেকটি $\neg x$ এর জন্য। এখন ধরা যাক $(\neg x \vee y)$ হলো একটি ক্লজ, তাহলে একে আমরা দুটি ইম্পলিকেশন চিহ্ন (implication sign) দিয়ে লিখতে পারি: $x \rightarrow y$ আর $\neg y \rightarrow \neg x$.^১ অথবা চাইলে $(\neg x \vee y)$ কে এভাবে তুমি চিন্তা করতে পার যে- যদি $\neg x = \text{false}$ হয় তাহলে $y = \text{true}$ হবে (যদি $x = \text{true}$ হয় তাহলে $y = \text{true}$ হতে হবে), অথবা যদি $y = \text{false}$ হয় তাহলে $\neg x = \text{true}$ হবে (যদি $\neg y = \text{true}$ হয় তাহলে $\neg x = \text{true}$ হতে হবে)। যেভাবেই তুমি চিন্তা

^১যারা ইম্পলিকেশন চিহ্ন (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 পার: $a = \text{true}$ হলে $b = \text{true}$ হবে, $a = \text{false}$ হলে b যা খুশি তাই হোক যায় আসে না।

কর না কেন আমরা একে গ্রাফে ডিরেস্টেড বাহু দিয়ে প্রকাশ করব। ($\neg x \vee y$) এর ক্ষেত্রে আমাদের বাহু হবে x থেকে y এর দিকে আর $\neg y$ থেকে $\neg x$ এর দিকে (ইমপ্লিকেশন চিহ্নের দিক অনুসারে)। অন্যভাবে বলতে: যদি $x = \text{true}$ হয় তাহলে $y = \text{true}$ হবে, আর যদি $\neg y = \text{true}$ হয় তাহলে $\neg x = \text{true}$ হবে এ জিনিসটা তুমি ইমপ্লিকেশন চিহ্ন দিয়ে যেভাবে প্রকাশ করেছো সেভাবে ডিরেস্টেড বাহু দেওয়া হবে। এখন আমরা এভাবে প্রত্যেকটি ক্লজের জন্য দুটি করে বাহু দিব। সব বাহু আঁকা শেষে আমাদের দেখতে হবে যে, x আর $\neg x$ (এখানে x হলো যেকোনো ভ্যারিয়েবল, অর্থাৎ তোমাকে n টি ভ্যারিয়েবল এর জন্য যাচাই করে দেখতে হবে) এর জন্য x থেকে $\neg x$ এ আর $\neg x$ থেকে x এ দুটিকেই পাথ আছে কিনা। যদি থাকে তাহলে আমাদের 2-sat সমাধান করা যাবে না। কারণ x হতে $\neg x$ এ পাথ থাকা মানে $x = \text{true}$ হলে $\neg x = \text{true}$ হবে। সুতরাং এক্ষেত্রে আমরা বলতে পারি $x = \text{true}$ হতে পারবে না, $x = \text{false}$ হবে। কিন্তু যদি অন্যদিকেও পাথ থাকে মানে $\neg x$ হতে x এ তাহলে তো আবার একইভাবে আমরা বলব $\neg x = \text{false}$ হতেই হবে। কিন্তু x আর $\neg x$ তো একই সঙ্গে false হতে পারে না তাই না? এজন্য যদি x আর $\neg x$ এর যদি একটি থেকে উপরটিতে যাওয়া যায় এর মানে হবে সমাধান নেই। আর যদি এমন কোনো ভ্যারিয়েবল খুঁজে পাওয়া না যায় তাহলে সমাধান করা যাবে। আমরা দুটি নোড থেকে একে অপরের দিকে যাওয়া যায় কিনা কীভাবে সহজে বের করতে পারি? SCC! যদি আমাদের গ্রাফকে SCC তে ভাঙ্গার পরে দেখি x ও $\neg x$ একই কম্পোনেন্টে তাহলে বুঝব একে উপরের দিকে যাওয়া যায়, আর না হলে যাবে না।

এতক্ষণ আমরা বের করলাম কীভাবে 2-sat সমাধানযোগ্য কিনা তা বের করা যায়। কিন্তু এর সমাধান কীভাবে বের করা যায় (অর্থাৎ কোন ভ্যারিয়েবল false আর কোন ভ্যারিয়েবল true)? এর উপায় হলো, তুমি SCC এর প্রত্যেক কম্পোনেন্টকে কন্ট্র্যাক্ট (contract) বা সংকুচিত করে একটি নোড বানাও। এই পরিবর্তিত গ্রাফ অবশ্যই একটি DAG হবে (DAG = Directed Acyclic Graph) অর্থাৎ এই ডিরেস্টেড গ্রাফে কোনো সাইকেল নেই। যেহেতু কোনো সাইকেল নেই তাই এর টপোলজিক্যাল ক্রম আছে। আমাদের যা করতে হবে, এই অর্ডারের শেষ থেকে আসতে হবে আর প্রত্যেক কম্পোনেন্টকে true দেওয়ার চেষ্টা করতে হবে (কোনো কম্পোনেন্টকে true দেবার মানে হলো এতে থাকা সব নোড true)। যদি দেখ তোমার এই কম্পোনেন্টে এমন একটি নোড আছে যার মান আগে থেকেই বসানো করা হয়ে গেছে আর তোমার এই এখন true করতে চাওয়া মানের সঙ্গে মিলছে না তাহলে false দিবে। তুমি চাইলে চিন্তা করে দেখতে পার বা প্রমাণও করতে পার কেন এই গ্রীডি (greedy) পদ্ধতিটি ঠিকভাবে মান বসাচ্ছে।

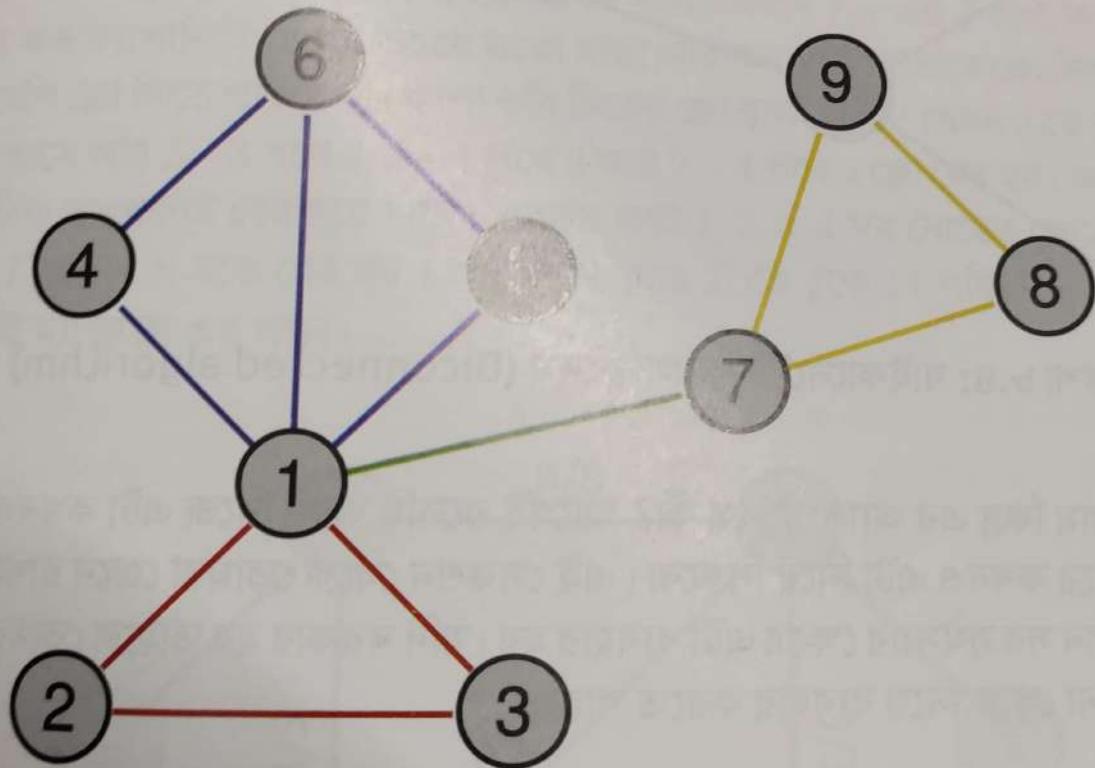
৮.১২ বাইকানেস্টেড কম্পোনেন্ট

কম্পোনেন্ট

(Biconnected
component)

সত্যি কথা বলতে আমি এই অ্যালগরিদম নিজে থেকে কখনও ইমপ্লিমেন্ট করিনি। আমার কাছে বেশ কঠিন লাগে বা বলতে পার সময়সাপেক্ষ লাগে। বাইকানেস্টেড গ্রাফ (Biconnected graph) মানে হলো সেই গ্রাফের কোনো ভাট্টেকে যদি আমরা মুছে ফেলি তাহলে সেই গ্রাফ ডিসকানেস্টেড হবে না। যেমন ধরা যাক আমাদের তিন নোডের একটি গ্রাফ আছে এবং এদের

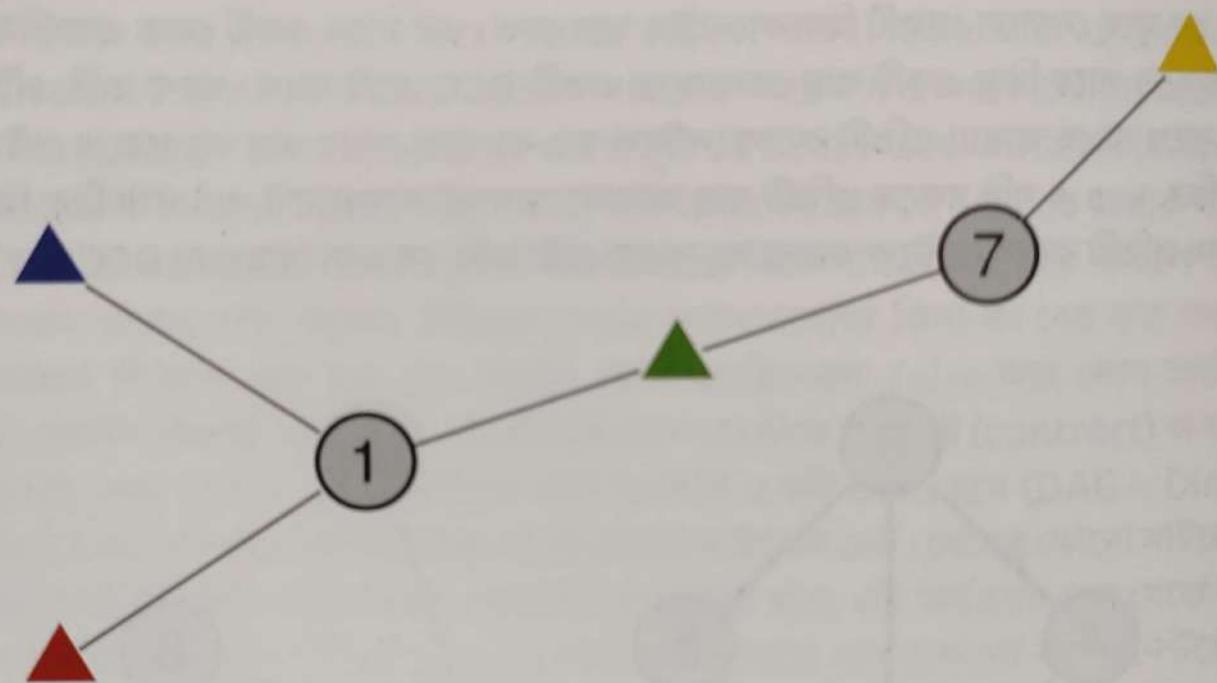
মাঝের বাহু গুলো হলো 1 – 2, 2 – 3. এটি কিন্তু বাইকানেন্টেড গ্রাফ না কারণ এই গ্রাফ থেকে আমরা যদি 2 মুছে ফেলি তাহলে বাকি নোড দুটি ডিসকানেন্টেড হয়ে যায়। কিন্তু এই গ্রাফটিতে যদি আরও একটি বাহু 1 – 3 থাকত তাহলে কিন্তু এটি বাইকানেন্টেড গ্রাফ হতো। আমরা যেকোনো গ্রাফকে (biconnected component) বা বাইকানেন্টেড কম্পোনেন্ট খেয়াল কর, একটি বাহু একাই কিন্তু একটি BCC হতে পারে কারণ এর এক মাথার নোড মুছে ফেললে কিন্তু কেউ ডিসকানেন্টেড হয় না। আমরা চিত্র ৮.৩ তে একটি গ্রাফকে BCC তে ভাঙিয়ে দেখালাম। প্রতিটি রঙ একেকটি কম্পোনেন্ট। এখানে খেয়াল কর একটি নোড কিন্তু একাধিক কম্পোনেন্টের অংশ হতে পারে। কেন? কারণ দেখ নীল রঙের কম্পোনেন্টে তুমি যদি 1 নোডটি মুছে ফেল তাহলে বাকি নোডগুলো কানেন্টেড থাকে। আবার লাল অংশেও একই কথা সত্য। তবে তুমি যদি লাল আর নীল এই দুই অংশকে একত্র করে যদি বলতে এটি একটি BCC তাহলে তা কিন্তু সত্য হতো না কারণ এর অংশ হতে পারে কিন্তু একটি বাহু কেবলমাত্র একটি BCC এরই অংশ। আশা করি এটি বলার অপেক্ষা রাখে না যে আমরা প্রতিটি কম্পোনেন্টকে যত বড় করা সম্ভব তত বড় করতে চেষ্টা করি। তোমরা চিত্র ৮.৩ এ যদি বলতে প্রতিটি বাহু আলাদা আলাদা কম্পোনেন্ট, হ্যাঁ কথা ঠিক কিন্তু এই যে বললাম প্রতিটি কম্পোনেন্টকে আমরা বড় করার চেষ্টা করি, সে জন্য আমাদের BCC হবে চিত্রের মতো।



চিত্র ৮.৩: বাইকানেন্টেড অ্যালগরিদম (Biconnected algorithm)

অনেক সময় সমস্যা ভেদে তোমাকে হয়তো গ্রাফকে কম্পোনেন্টে ভাগ করতে হয় যেন একই কম্পোনেন্টের কোনো বাহু মুছে ফেললে গ্রাফটি ডিসকানেন্টেড না হয়ে যায়। সেক্ষেত্রে আমাদের একটি কষ্ট করতে হবে না, শুধু আর্টিকুলেশন ব্রিজ বা বাহু বের করে একটি BFS বা DFS চালিয়ে

লিঙ্গেই আমাদের সব কম্পোনেন্ট বের হয়ে যাবে। আমরা প্রতিটি *unvisited* মোডের জন্য BFS বা DFS চালাব এবং আর্টিকুলেশন বিজ ব্যতীত অন্য সব বাছ দিয়ে আমরা ট্রাভার্স করব। BCC এর সঙ্গে সম্পর্কিত আরেকটি জিনিস আছে আর তা হলো ব্লক কার ভার্টেক্স ট্রি (Block cut vertex tree)। প্রতিটি আনডিরেক্টেড গ্রাফকে যেমন আমরা BCC তে ভাগ করতে পারি ঠিক তেমনই সেই BCC কে আমরা একটি ট্রি আকারে সাজাতে পারি যেখানে ট্রি এর মোড়গুলো হলো BCC এর কাট ভার্টেক্স (আর্টিকুলেশন নোড) সমূহ এবং ব্লক (block) বা কম্পোনেন্টগুলো। যদি কোনো একটি কাট ভার্টেক্স একটি ব্লকের অংশ হয় তাহলে তাদের মধ্যে বাছ থাকবে। তাহলে যেকোনো কানেক্টেড আনডিরেক্টেড গ্রাফ (connected undirected graph) এর জন্য আমরা একটি ট্রি পাব। যেমন চিত্র ৮.৩ এর ব্লক কাট ভার্টেক্স ট্রি হবে চিত্র ৮.৪ এর মতো। বেশ কিছু সমস্যায় আমরা দেখব যে এই ট্রি বেশ কাজে লাগে।



নকশা ৮.৪: বাইকানেক্টেড অ্যালগরিদম (Biconnected algorithm)

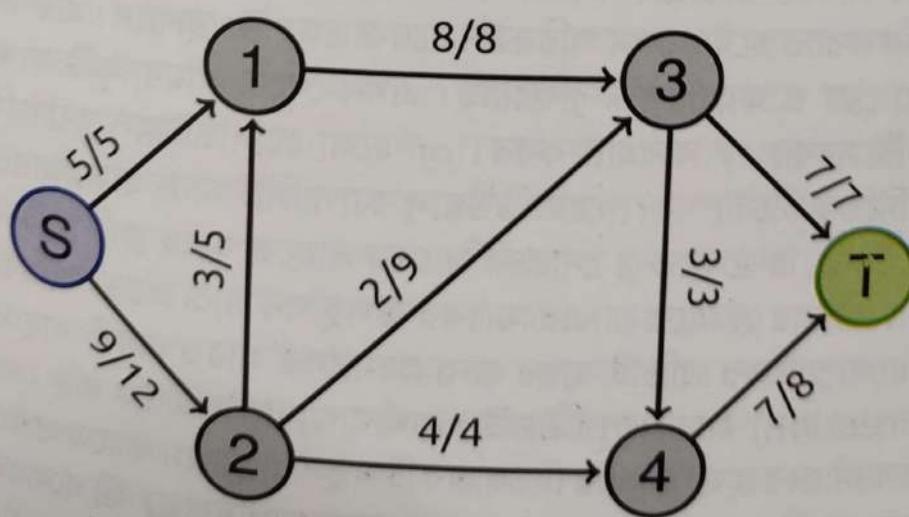
সবই বুকলাম কিন্তু এর অ্যালগরিদম কী? আগেই বলেছি আমি নিজে এটা কথনও কোড করি নাই। হয়তো পরে কথনও এটা নিয়ে লিখবো। এই সেকশন থেকে তোমরা জেনে রাখলে BCC কী জিনিস এবং কোন সব সমস্যার ক্ষেত্রে এটা ব্যবহার হয়। যদি দরকার হয় তাহলে তোমরা ইন্টারনেট থেকে এর কোনো কোড নিয়ে ব্যবহার করতে পারো।

৮.১৩ ফ্লো (Flow) সম্পর্কিত অ্যালগরিদম

নিঃসন্দেহে ফ্লো (flow) একটি কঠিন টপিক। এর কোড বেশ সহজ কিন্তু এটি ঠিক মতো বোঝা বা একে রঙ করা খুবই কঠিন। দেখা যাক তোমাদের এর মূল ধারনাটি বোঝাতে পারি কিনা।

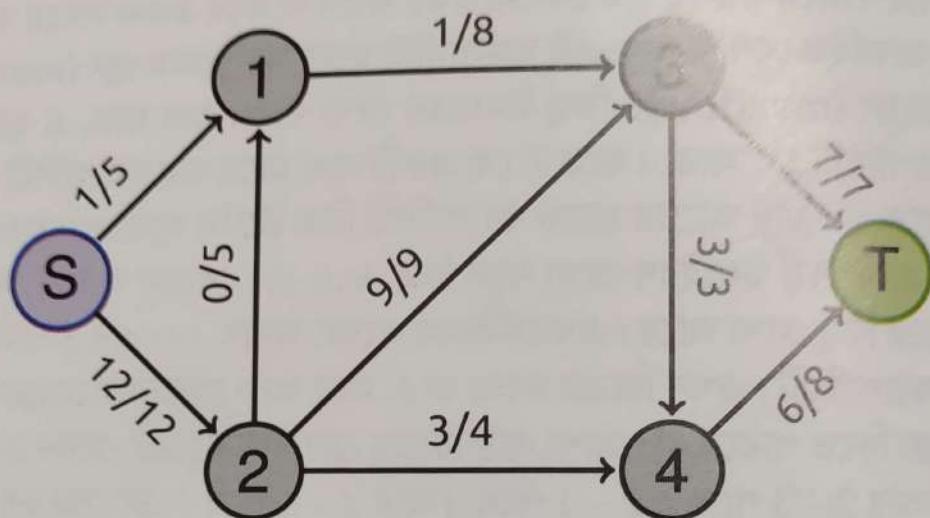
৮.১৩.১ ম্যাক্সিমাম ফ্লো (Maximum flow)

এই সমস্যায় কিছু নোড এবং কিছু ডিরেক্টেড বাহু থাকবে। নোডসমূহের মাঝে দুটি বিশেষ নোড থাকবে। একটি হলো উৎস বা সোর্স (source) যা সাধারণত আমরা S দিয়ে প্রকাশ করি আর আরেকটি হলো গন্তব্য বা সিঙ্ক (sink) যা আমরা সাধারণত T দিয়ে প্রকাশ করি। ডিরেক্টেড বাহসমূহ এর সঙ্গে একটি সংখ্যা থাকবে, একে আমরা ওজন (weight) বলব না, একে আমরা বলব সিঙ্কে যেকোনো সময়ে অপরিসীম তরল পদার্থ আছে, আর তাদের একেকটি পাইপ মনে কর যাদের ক্যাপাসিটি (capacity). এখন মনে কর সোর্সে অপরিসীম তরল পদার্থ আছে, আর সিঙ্কে যেকোনো সময়ে অপরিসীম তরল প্রবেশ করতে পারে। অন্যান্য যেসব বাহুর কথা বললাম সর্বোচ্চ কত তরল ওই পাইপ দিয়ে প্রবাহিত হতে পারে তা দেওয়া আছে, আর কোনো একক সময়ে কী পরিমাণ তরল সোর্স হতে সিঙ্কে প্রবাহিত হতে পারে? তোমরা ধরে নিতে পার যে, কোনো পাইপ দিয়ে তরল যেতে কোনো সময় লাগে না তবে একক সময়ে তার ক্যাপাসিটির বাসংক্ষেপে ম্যাক্সফ্লো (maxflow). কিছু উদাহরণ দেখা যাক। ধরা যাক, S হতে A তে একটি বাহ আছে যার ক্যাপাসিটি 10, আর A হতে T তে একটি বাহ আছে যার ক্যাপাসিটি 20. তাহলে এই গ্রাফে ম্যাক্সফ্লো হবে 10. যদি আগের গ্রাফে ক্যাপাসিটি ঠিক উল্টো হতো তাহলেও কিন্তু ম্যাক্সফ্লো হতো 10. এবার একটু বড় উদাহরণ দেখা যাক চিত্র ৮.৫ এ। খেয়াল করলে দেখবে যে এখানে বাহতে f/c আকারে কিছু লেখা আছে। এখানে প্রথম সংখ্যা f/c এর f হলো কত ফ্লো হচ্ছে, আর c হলো কত ক্যাপাসিটি। এখন চিত্রের মতো ছাড়া আর অন্য কোনোভাবে ফ্লো দিলেও তুমি 14 পথে ফ্লো আসে আর 2 – 3 পথে 2, 2 – 1 পথে 3 আর 2 – 4 পথে 4 ফ্লো বের হয়। অর্থাৎ যতখানি ঝুঁকবে ঠিক ততখানিই বের হয়ে যাবে। এরকম কথা 1, 2, 3, 4 সব নোডের ক্ষেত্রেই প্রযোজ্য। শুধু S আর T ছাড়া। S হতে বের হয় 14 পরিমাণ, আর T তে চুকে 14 পরিমাণ। এই পরিমাণই সর্বোচ্চ করাই ম্যাক্সফ্লো এর লক্ষ্য।



নকশা ৮.৫: ম্যাক্সিমাম ফ্লো (Maximum flow)

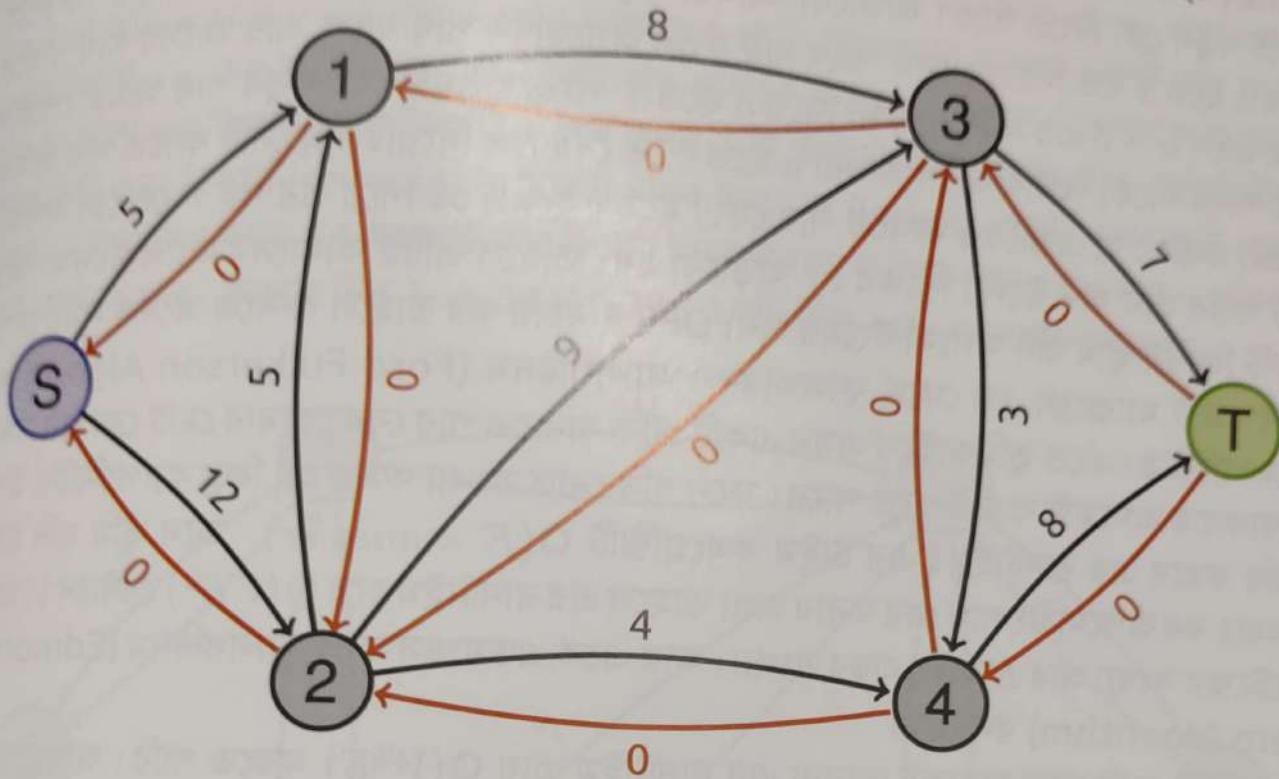
তাহলে আশা করি ম্যাত্রিফ্লো কী জিনিস তা বোঝা গেছে? এখন আসো কঠিন অংশে। কীভাবে ম্যাত্রিফ্লো সমস্যা সমাধান করা যায়? সহজ হতে শুরু করা যাক। একটি খুব সাধারণ উপায় হলো যতক্ষণ আমরা S হতে T তে এমন একটি পাথ পাব যা দিয়ে কিছু পরিমাণ ফ্লো দেওয়া যাব সেই পাথে ফ্লো দেওয়া। যখন আর এমন কোনো পাথ পাব না তাকে ম্যাত্রিফ্লো বলবো। কিন্তু এটা কাজ করবে না। চিত্র ৮.৫ এর গ্রাফে আমরা যদি অন্যভাবে ফ্লো দেই তাহলেই দেখতে পারবে যে মোট ফ্লো 14 না অর্থাৎ আর কোনো ফ্লো দিতে পারছ না। মনে কর, $S - 2 - 3 - T$ দিয়ে 7, $S - 2 - 3 - 4 - T$ দিয়ে 1 এবং $S - 2 - 4 - T$ দিয়ে 3 ফ্লো যদি দাও তাহলে মোট ফ্লো হয় $7 + 2 + 1 + 3 = 13$ এবং আমাদের গ্রাফ দেখতে চিত্র ৮.৬ এর মত হবে। এখানে দেখো আর কোনো পাথ পাবে না যা দিয়ে তুমি আরও ফ্লো দিতে পারো কিন্তু এর ফ্লো হলো 13। আমরা চিত্র ৮.৫ এ দেখে এসেছি যে অন্তত 14 পাওয়া যাবে। সুতরাং আমাদের এ পদ্ধতি কাজ করে না।



নকশা ৮.৬: ম্যাত্রিমাম ফ্লো (Maximum flow): লোকাল ম্যাত্রিমা (local maxima) কিন্তু ম্যাত্রিমাম (maximum) নয়

তাহলে কীভাবে আমরা ম্যাত্রিফ্লো সমস্যা সমাধান করব? এই সমাধান বুঝতে হলে আমাদের গ্রাফে কিছু পরিবর্তন করতে হবে। প্রথম পরিবর্তন হবে গ্রাফের উপস্থাপনে। এতক্ষণ কোনো বাহুতে f/c বলতে আমরা ফ্লো ও ক্যাপাসিটি বুঝিয়েছি। এখন থেকে আমরা দুটি সংখ্যা f/c ব্যবহার না করে মাত্র একটি সংখ্যা cf ব্যবহার করব। cf হলো রেসিডিউয়াল ক্যাপাসিটি (residual capacity) বা গাণিতিকভাবে $c - f$ । রেসিডিউয়াল ক্যাপাসিটি মানে হলো আরও কত ফ্লো যেতে পারে। যেমন চিত্র ৮.৫ এ 2 হতে 3 নোডের মাঝের বাহুতে আছে $2/9$ এটি নতুন উপস্থাপনে হবে $9 - 2 = 7$ আবার S হতে 1 এর মাঝের উপস্থাপন $5/5$ হতে পরিবর্তন হয়ে 0 হবে। দ্বিতীয় পরিবর্তন হলো গ্রাফের প্রতিটি বাহুর জন্য আমাদের আরও একটি বাহু থাকবে যার দিক বাহুরেকশন (direction) হবে সম্পূর্ণ উল্লেখ। অর্থাৎ আমাদের গ্রাফে যদি 1 হতে 3 এ কোনো বাহু থাকে তাহলে আমাদের 3 হতে 1 এর দিকে একটি নতুন বাহু যোগ করতে হবে। এখন প্রশ্ন হলো এই বাহুর প্রাথমিক cf কী হবে? তার আগে সব মূল বাহুর প্রাথমিক cf কী হবে তা পরিষ্কার করি। এটি হবে c , কারণ প্রথমে কোনো ফ্লো থাকবে না তাই $f = 0$ এবং $cf = c - f = c - 0 = c$.

এখন এর বিপরীত বাহুর কথা ভাবা যাক, এর প্রাথমিক লেবেল হবে 0. কারণ আমাদের এর ডেতর দিয়েও কোনো প্রাথমিক ফ্লো যাবে না এবং এর ক্যাপাসিটি 0. তাহলে এখন আমরা চির 8.7 এ আমাদের গ্রাফের প্রাথমিক রূপ (মানে যখন কোনো ফ্লো যায় নাই সেই অবস্থা) পরিবর্তিত উপস্থাপন (cf এর মাধ্যমে) এ দেখি। আশা করি বুবাতে পারছ যে লাল বাহুগুলো হলো উল্টো বাহু।



নকশা ৮.৭: ম্যাক্সিমাম ফ্লো (Maximum flow): পরিবর্তিত উপস্থাপনে প্রাথমিক রূপ

এখন যা করতে হবে তা হলো এমন একটি পাথ খুঁজে বের করতে হবে যার প্রতিটি বাহুতে কিছু পরিমাণ রেসিডিউয়াল ক্যাপাসিটি থাকে। অর্থাৎ আমরা S হতে শুরু করবে এবং একটি BFS বা DFS করে T পর্যন্ত যাওয়ার চেষ্টা করবে সেসব বাহু ব্যবহার করে যাদের $cf > 0$. এই পাথকে বলা হয় অগমেন্টিং পাথ (augmenting path). এখন যা করতে হবে তা হলো এই পাথের সব বাহু এর cf এর মধ্যে সর্বনিম্ন cf বের করতে হবে। আমরা এই পুরো পাথ দিয়ে এই পরিমাণ ফ্লো করাব। মনে কর এই সর্বনিম্ন cf হলো x . তাহলে যা করতে হবে তা হলো এই পাথের সব বাহু cf কে x পরিমাণ করাতে হবে। কারণ cf বলে কী পরিমাণ আরও ফ্লো করানো যাবে আর সেহেতু আমরা x পরিমাণ ফ্লো করছি সেহেতু রেসিডিউয়াল ক্যাপাসিটি x পরিমাণ করাতে হবে। এটুকু তো বোৰা গেল কিন্তু এর পর যা বলব সেটিই হলো মূল ঝামেলা। সেটি হলো, আমরা যেই যেই বাহু cf কে x কমিয়েছি তার উল্টো বাহুর cf কে x বাঢ়াতে হবে। অর্থাৎ যদি অগমেন্টিং পাথ কোনো বাহু দিয়ে যায় তাহলে তার উল্টো বাহুর cf বাঢ়াতে হবে। এই পুরো প্রসেসকে অগমেন্ট (augment) করা বলে। এখন কথা হলো আমরা কেন উল্টো দিকের বাহুগুলোর cf বাঢ়াব? এটি আসলে বলে অতটা ভালো করে বোঝানো সম্ভব না, এটি কিছুটা অনুভবের বিষয়। তবুও বলি, যদি আমরা a হতে b এর দিকে ফ্লো দেই এর মানে হলো S হতে কোনোভাবে আমরা a তে এসেছি এর পর $a - b$ বাহু দিয়ে আমরা b তে এসেছি, এর পর কোনোভাবে b হতে T তে গিয়েছি। ধরা যাক

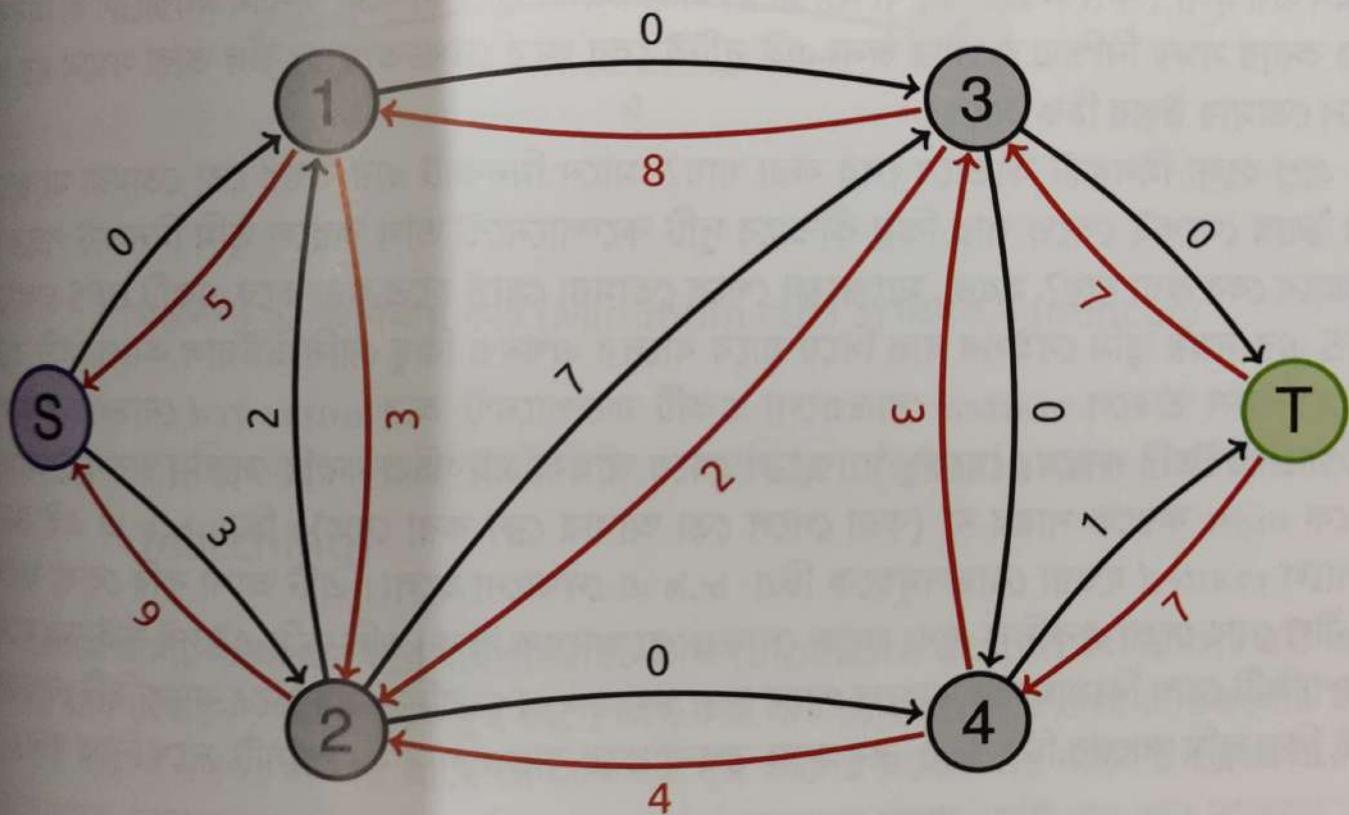
এই পাথের নাম P . এখন কথা হলো আমরা তো চাইলে $a - b$ না গিয়ে $a - c$ ও যেতে পারতাম তাই না? এবং হয়তো ওইভাবে গেলে আমরা অপটিমাল সমাধান পেতাম। কিন্তু আমরা তো $a - b$ দিয়ে চলে এসেছি। কী করা যায়? আচ্ছা মনে কর পরের ধাপে আমরা S হতে কোনোভাবে b তে এসেছি। এখন যদি আমরা a হতে T পর্যন্ত কোনো পাথ পাই তাহলে এই নতুন পাথ আর P মিলে পাথে b হতে T তে গিয়েছে সেই পাথে এই নতুন অগমেন্টিং পাথ যাবে, আর আগের পাথ S হতে a তে এসে $a - b$ এর মধ্য দিয়ে না গিয়ে a হতে T পর্যন্ত যাওয়ার যেই নতুন পাথ আমরা পেয়েছি বাতিল করছি। বা এভাবেও ভাবতে পার যে S হতে b হয়ে a তে গিয়ে এর পর T তে ফ্লো দিলাম। এটি সন্তুষ্ট হবে যদি উল্টো দিকের cf বাড়ানো হয়। তাহলে এটিই আমাদের অ্যালগরিদম। তবে একটি জিনিস তুমি যদি অগমেন্টিংয়ের জন্য DFS ব্যবহার কর তাহলে আসলে অনেক সময় লেগে যাবে। এটি ম্যাক্সফ্লো এর ফোর্ড ফালকারসন অ্যালগরিদম (Ford Fulkerson Algorithm) নামে পরিচিত। এতে তুমি চাইলে এমন একটি গ্রাফ বানাতে পার যেন তোমার মোট ফ্লো এর সমান সংখ্যক বার অগমেন্ট করতে হতে পারে। ফলে যদি নোড সংখ্যা কমও হয় কিন্তু ক্যাপাসিটির উপর নির্ভর করবে এর রানটাইম। এর টাইম কমপ্লেক্সিটি $O(E \times \text{answer})$. তবে তুমি যদি BFS ব্যবহার কর অগমেন্টিং পাথ বের করার জন্য তাহলে এর রানটাইম হবে $O(V E^2)$ যেখানে V হলো ভার্টেক্সের সংখ্যা আর E হলো বাহুর সংখ্যা। আর একে এডমন্ডস কার্প অ্যালগরিদম (Edmonds Karp Algorithm) বলা হয়।

একটি ছোট কাজ করলেই আমরা এর রানটাইম প্রায় $O(V^2E)$ করতে পারি। আমাদের যা করতে হবে তা হলো, প্রথমে রেসিডিউয়াল ক্যাপাসিটির উপর একটি পূর্ণ BFS চালাতে হবে। পূর্ণ BFS চালানোর সময় BFS এর ডেপথ বা গভীরতার তথ্য রাখতে হবে। অর্থাৎ BFS কে তো একটি ট্রি হিসেবে কল্পনা করা যায়। কোন নোড কত গভীরতায় আছে সেই তথ্য রাখতে হবে। এর পর একটি ব্যাকট্র্যাক (backtrack) বা DFS এর মতো কাজ করতে হবে। আমরা S হতে শুরু করব এবং প্রতিবার পরের গভীরতার কোনো নোডে যাওয়ার চেষ্টা করব (যদি রেসিডিউয়াল ক্যাপাসিটি ধনাত্মক হয়)। যদি আমরা T তে পৌঁছাই তাহলে এই পাথটি অগমেন্ট করব। এভাবে চলতে থাকবে। একে ডিনিকের অ্যালগরিদম (Dinic's algorithm) বা ডিনিকের ব্লকিং অ্যালগরিদম (Dinic's blocking algorithm) বলা হয়। এর থেকেও ভালো অ্যালগরিদম সাধারণত দরকার হয় না।

আরও কিছু বলার আগে ছোটখাটো দুই একটি কথা জেনে রাখা ভালো। প্রথমত আমি এখানে বোঝানোর সুবিধার জন্য ইচ্ছা করে ডিরেক্টেড গ্রাফ নিয়েছিলাম। কিন্তু তুমি যদি বই পড় বা সমস্যা দেখ তাহলে দেখবে বেশির ভাগ সময় আনডিরেক্টেড বাহুতে ক্যাপাসিটি দেওয়া থাকে। এসময় যা করতে হয় তা হলো দুই দিকের জন্য দুটি ডিরেক্টেড বাহু তোমাকে লাগাতে হবে। তুমি চাইলে এই দুটি দিয়েই কাজ সারতে পার বা এই দুটির উল্টো দিকে আরও দুটি বাহু লাগাতে পার। আরেকটি বিষয় হলো তুমি কীভাবে এই গ্রাফের বাহু বা cf রাখবে? তুমি চাইলে অ্যাডজাসেন্সি ম্যাট্রিক্স রাখতে পার। তাতে সমস্যা হলো BFS এর সময় আর তোমার টাইম কমপ্লেক্সিটি $O((\cdot)E)$ থাকবেনা।

¹পূর্ণ বলার কারণ হলো এর আগেরবার আমরা S হতে BFS শুরু করতাম আর T তে পৌঁছে গেলেই আমরা শেষ করে দিতে পারতাম চাইলে। কিন্তু এবার যতক্ষণ না সব ভার্টেক্সই *visited* হচ্ছে ততক্ষণ আমাদের BFS চালাতে হবে।

$O(V^2)$ হয়ে যাবে। আবার তুমি যদি অ্যাডজাসেন্সি লিস্ট কর তাহলে উল্টো দিকের cf পরিবর্তন করা আবার আরেক ঝামেলা। তুমি চাইলে একই সঙ্গে অ্যাডজাসেন্সি লিস্ট ও ম্যাট্রিক্স রাখতে পার। তবে আমি ভেক্টর (vector) দিয়ে অ্যাডজাসেন্সি লিস্ট বানিয়ে এই কোড করে থাকি। মনে কর তোমাকে x হতে y এর দিকে একটি বাহু দেওয়া হলো। প্রথমে x আর y এর অ্যাডজাসেন্সি লিস্টে কতগুলো উপাদান আছে তা দেখে নাও। ধরা যাক এই দুটি সংখ্যা যথাক্রমে $size_x$ এবং $size_y$ । এর মানে তুমি x হতে y এই বাহুটি যখন আমাদের ডেটা স্ট্রাকচারে রাখবে একটি বাহু থাকবে x এর লিস্টে sz_x তম স্থানে আর তার উল্টো বাহু থাকবে y এর sz_y স্থানে। তোমাকে যা করতে হবে তা হলো একটি বাহু যখন ইনসার্ট করবে তখন 3 টি তথ্য রাখবে: অপর প্রান্ত, রেসিডিউয়াল ক্যাপাসিটি এবং রিভার্স (reverse) বা উল্টো বাহুর ইনডেক্স (index)। শেষ। এখন তুমি কোনো বাহু cf কমানোর সময় খুব সহজেই তার উল্টো দিকের বাহুতে গিয়ে তার cf এর মান পরিবর্তন করে ফেলতে পার। তাহলে চিত্র 8.7 এর ম্যাট্রিক্সে রূপটি চিত্র 8.8 এ দেখানো হলো।



চিত্র 8.8: ম্যাট্রিমাম ফ্লো (Maximum flow): ম্যাট্রিক্সে (maxflow) তে গ্রাফের ছবি

৮.১৩.২ মিনিমাম কাট (Minimum cut)

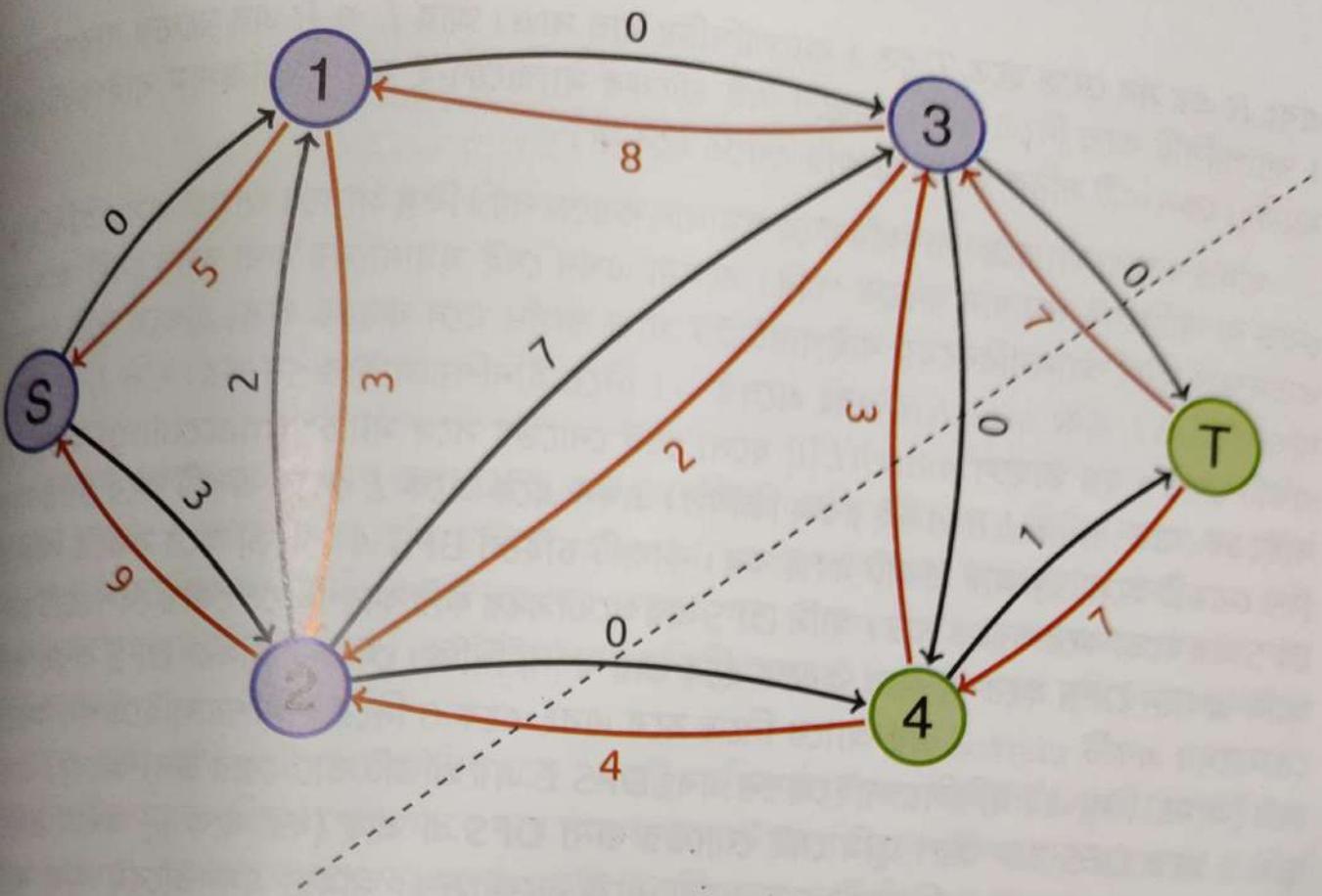
যেকোনো অপটিমাইজেশন সমস্যার একটি দ্বিতীয় বা dual সমস্যা থাকে। অর্থাৎ যদি একটি সমস্যা থাকে যেখানে আমাদের কোনো কিছুর মান বাঢ়াতে বা ম্যাট্রিমাইজেশন (maximization) করতে হবে তাহলে আমরা সেই সমস্যাকে অন্যভাবে দেখতে পারি যেখানে কোনো কিছুর মানকে কমাতে বা মিনিমাইজেশন (minimization) করতে হয়। প্রধান সমস্যা বা primal এর যা উত্তর কি dual এরও একই উত্তর হয়। আমরা কিছুক্ষণ আগে ম্যাট্রিক্সে সমস্যা দেখলাম। সেখানে

আমরা ফ্লো কে ম্যাক্সিমাইজেশন করেছিলাম। এখন কথা হলো এর dual সমস্যা কী? এটিই হলো মিনিমাম কাট (minimum cut) বা সংক্ষেপে মিনকাট (mincut). মিনকাট সমস্যাটি হলো, আগের মতোই সোর্স বা সিঙ্ক থাকবে এবং বাহ্যসমূহের ক্যাপাসিটি থাকবে। তোমাকে কিছু বাহ ডিলিট করে সোর্স এবং সিঙ্ককে ডিসকানেক্টেড করতে হবে যেন সোর্সের কম্পোনেন্ট থেকে সিঙ্কের কম্পোনেন্টে যাওয়া বাহ্যসমূহের ক্যাপাসিটির যোগফল বিয়োগ সিঙ্কের কম্পোনেন্ট থেকে সোর্সের কম্পোনেন্টে যাওয়া বাহ্যসমূহের ক্যাপাসিটির যোগফল সর্বনিম্ন হয়। খেয়াল কর আমি (S, 1, 2) কে একটি কম্পোনেন্ট আর (3, 4, T) কে আরেকটি কম্পোনেন্ট ধরি তাহলে এই কাট (cut) এর মূল্য হবে $8 + 4 + 9 = 21$. যদি (S, 1, 2, 3, 4) এক কম্পোনেন্ট আর বাকি (T) এক কম্পোনেন্ট হয় তাহলে মূল্য হবে $7 + 8 = 15$. যদি (S, 1) একটি আর (2, 3, 4, T) আরেকটি হয় তাহলে মূল্য হয় $8 - 5 + 12 = 15$. অপটিমাল কাট হবে (S, 1, 2, 3) এবং (4, T) এর ক্ষেত্রে। কারণ এর মূল্য $7 + 3 + 4 = 14$ যা ম্যাক্সিমুম এর সমান। তুমি কাগজে কলমে ম্যাক্সিমুম বা মিনকাট বের করার সময় নিশ্চিত হওয়ার জন্য এই দুটিই বের করে দেখতে পার, যদি তারা সমান হয় তার মানে তোমার উত্তর ঠিক আছে।

প্রশ্ন হলো মিনকাট কীভাবে বের করা যায়? মানে মিনকাট এর উত্তর তো তোমরা ম্যাক্সিমাইজ এর উত্তর থেকেই পেতে পার কিন্তু কীভাবে দুটি কম্পোনেন্টে ভাগ করলে তুমি মিনকাট পাবে তা কীভাবে বের করা যায়? সহজ, ম্যাক্সিমাইজে শেষে তোমরা সোর্স হতে শুরু করে একটি BFS চালাবে, BFS এর সময় তুমি সেইসব বাহু দিয়ে যাবে যাদের এখনও কিছু রেসিডিউয়াল ক্যাপাসিটি বাকি আছে। ব্যাস তাহলে *visited* নোডগুলো একটি কম্পোনেন্ট আর *unvisited* নোডগুলো অপর কম্পোনেন্ট তৈরি করবে। যেহেতু ম্যাক্সিমাইজে আমরা এই কাজ করছি সেজন্য অবশ্যই আমরা T কে *visit* করতে পারব না (করা গেলে তো আবার ফ্লো করা যেত)। চিত্র ৮.৮ এ এই BFS চালালে *visited* হওয়া নোডসমূহকে চিত্র ৮.৯ এ দেখানো হলো। এটি আশা করি বোৰা যাচ্ছে যে নীল নোডগুলো একদিক এবং সবুজ নোডগুলো আরেক দিক। যদি তুমি এই দুই সাইডের মধ্যে ক্যাপাসিটি যোগ বিয়োগ কর তাহলে পাবে $7 + 3 + 4 = 14$. যদিও এই চিত্রে ক্যাপাসিটি দেখানো নেই কিন্তু তুমি আগের চিত্র ৮.৫ এর সঙ্গে তুলনা করে বাহুগুলোর ক্যাপাসিটি দেখে নিতে পার।

৪.১৩.৩ মিনিমাম কস্ট ম্যাক্সিমাম ফ্লো (Minimum cost maximum flow)

অনেক সময় বাহসমূহের মূল্য বা কস্ট (cost) ও দেওয়া থাকে। অর্থাৎ এই বাহ দিয়ে যদি তুমি 1 ফ্লো চালাও তাহলে তোমাকে এতো মূল্য দিতে হবে। আর আমাদের কাছে জানতে চাওয়া হয় যে সবচেয়ে কম খরচে সবচেয়ে বেশি সংখ্যক ফ্লো কীভাবে দেওয়া যায়। এর সমাধান ম্যাক্সফ্লো এর মতোই। শুধু পার্থক্য হলো এখানে আমরা S হতে T তে যাওয়ার পাথ, BFS বা DFS করে বের করব না। বরং বেলম্যান ফোর্ড অ্যালগরিদম বা repeated ডায়াকস্ট্র্যাক্সেপ্ট অ্যালগরিদম করে বের করব। আর প্রতিবার ফ্লো এর মূল্যগুলো যোগ করব। তাহলেই মিনিমাম কস্ট ম্যাক্সিমাম ফ্লো (Minimum cost maximum flow) সমাধান হয়ে যাবে।



নকশা ৮.৯: মিনিমাম কাট (Minimum cut) বা মিনকাট (mincut)

৮.১৩.৮ ম্যাস্ক্রিমাম বাইপারটাইট ম্যাচিং (Maximum Bipartite Matching)

প্রথমে আমাদের জানতে হবে বাইপারটাইট গ্রাফ (bipartite graph) কী জিনিস। এটি এমন একটি গ্রাফ যেখানে নোডগুলোকে দুই ভাগে ভাগ করা যায় যেন প্রতিটি ভাগের নোডগুলোর মধ্যে কোনো বাহু না থাকে। যা বাহু আছে তা হবে এই দুই ভাগের মধ্যে। এর বাস্তব উদাহরণ এরকম হতে পারে, মনে কর তোমার কাছে কিছু বল আর কিছু বাল্ল আছে। যদি বল আর বাল্লকে তুমি নোড দিয়ে প্রকাশ কর তাহলে তুমি সেই সব বল আর বাল্লের মধ্যে বাহু দিবে যেন সেই বল সেই বাল্লের মধ্যে আঁটে। এই গ্রাফটি অবশ্যই একটি বাইপারটাইট গ্রাফ কারণ এখানে দুটি বল বা দুটি বাল্লের মধ্যে কোনো বাহু নেই। এখন যদি তোমাকে জিজ্ঞাসা করা হয় তুমি সর্বোচ্চ কতগুলো বল কোনো বাল্লতে ভরতে পারবে? মানে অবশ্যই তুমি একটি বলকে দুটি বাল্লে বা কোনো বাল্লে দুটি বল রাখতে পারবে না। এই সমস্যাটিই হলো ম্যাস্ক্রিমাম বাইপারটাইট ম্যাচিং (maximum bipartite matching). অর্থাৎ তুমি সর্বোচ্চ কতগুলো বাহু নির্বাচন করতে পারবে যেন কোনো সাড়ে একাধিক নির্বাচিত বাহু না থাকে।

ম্যাস্ক্রিমে ব্যবহার করে এই সমস্যা খুব সহজেই সমাধান করা যায়। মনে কর নোডগুলো যেই দুই ভাগে বিভক্ত তাদের নাম L ও R (left, right এর সংক্ষিপ্ত রূপ)। এখন তুমি একটি সোর্স S নাও এবং T হতে L এর সব নোডের 1 ক্যাপাসিটির বাহু দাও। একইভাবে একটি সিঙ্কে T নাও

এবং R এর সব নোড হতে T তে 1 ক্যাপাসিটির বাহু দাও। আর L ও R এর মধ্যের বাহুগুলোর 1 ক্যাপাসিটি করে দিতে হবে। তাহলে এই গ্রাফের ম্যাত্রিক্ষেত্রে-ই হলো ম্যাত্রিমাম বাইপারটাইট ম্যাচিং। কেন এটি সঠিক তা আশা করি বলতে হবে না।

যদিও আমরা ম্যাত্রিক্ষেত্রে ব্যবহার করে সমাধান করতে পারি কিন্তু আসলে আমরা এসব অতিরিক্ত নোড না লাগিয়েই সমাধান করতে পারি। আমরা এখন যেই সমাধানের কথা বলব সেটি আসলে এডমন্ডস কার্প অ্যালগরিদমের বাইপারটাইট গ্রাফ ভার্সন মনে করতে পার। প্রথমে দুটি অ্যারে নাও $matchL$ এবং $matchR$ এবং এদের -1 দিয়ে ইনিশিয়ালাইজেশন কর। যদি L এ থাকা একটি নোড i হয় তাহলে $matchL[i]$ হলো সেই নোডের সঙ্গে ম্যাচিং (matching) হওয়া R সাইডের নোড। $matchR$ একই রকম জিনিস। এখন একে একে L থেকে একটি করে নোড নাও DFS এর মতো করে করতে পার। আমি DFS এর মতো করে করি কারণ এতে কোড বেশ ছেট হয়। আমি এখানে DFS করে কীভাবে করতে হবে তার বর্ণনা দিচ্ছি। যেহেতু আমরা DFS করব তাই তোমাদের একটি *visited* এর অ্যারে নিতে হবে এবং একে 0 দিয়ে ইনিশিয়ালাইজেশন করতে হবে (আমরা কিন্তু এই ইনিশিয়ালাইজেশন এবং DFS L এর প্রতিটি ভাট্টেক্সের জন্য করব)। এখন তুমি x হতে DFS শুরু কর। তুমি যেই নোডের জন্য DFS এ আছ (ধরা যাক y , অর্থাৎ প্রথমে y এর মান হবে x) তার প্রতিবেশী নোডগুলো একে একে দেখ। আমরা এমনভাবেই কাজ করব যেন y সবসময় L এর হয়ে থাকে। সুতরাং এর প্রতিবেশী হবে R এর, ধরা যাক এরকম একটি প্রতিবেশী হলো z . এখন তোমাকে দেখতে হবে $matchR[z]$ কি -1 কিনা। যদি -1 না হয় তাহলে $matchR[z]$ এর DFS কল করতে হবে, তবে কল করার আগে দেখে নিও যে এই নোডটি আগেই *visited* কিনা। যদি *visited* হয় তাহলে আর কল করার দরকার নেই। আর যদি -1 হয় এর মানে একটি অগমেন্টিং পাথ পেয়েছে। তখন তোমাকে $matchL[y] = z$ এবং $matchR[z] = y$ করতে হবে এবং 1 রিটার্ন করতে হবে। কিছুক্ষণ আগে কিন্তু তুমি $matchR[z] = 1$ না হলে DFS কল করেছিলে। যদি এই DFS তোমাকে 1 রিটার্ন করে তার মানে তুমি অগমেন্টিং পাথ পেয়েছে এবং আগের মতোই $matchL[y] = z$ এবং $matchR[z] = y$ করবে এবং 1 রিটার্ন করে বুঝাতে হবে যে তুমি অগমেন্টিং পাথ পেয়ে গেছো। আর যদি দেখ y এর জন্য সব z শেষ কিন্তু তাও তুমি অগমেন্টিং পাথ পাওনি তাহলে 0 রিটার্ন কর। এই DFS এর কোডটি দেখতে কোড ৮.৯ এর মতো হবে।

কোড ৮.৯: bpm dfs.cpp

```

1 int dfs(int y) {
2     visited[y] = 1;
3     for (int i = 0; i < v[y].size(); i++) {
4         int z = v[y][i];
5         if (matchR[z] == -1 ||
6             (!visited[matchR[z]] && dfs(matchR[z]))) {

```

```

9     matchL[y] = z;
10    matchR[z] = y;
11    return 1;
12  }
13 }

```

একদম শুরুতে তো x এর জন্য DFS কল করেছিলে। যদি এই কল 1 রিটার্ন করে এর মানে তোমার ম্যাচিং 1 বেড়েছে বা তুমি চাইলে $matchL$ এ দেখতে পার কতগুলোর জন্য -1 নেই এভাবে তুমি ম্যাঞ্চিমাম ম্যাচিং কয়টি তা বের করতে পার আর $matchL$ দেখে এও বলতে পার যে কার সঙ্গে কে ম্যাচিং করেছে। এর টাইম কমপ্লেক্সিটি হলো $O(VE)$ কারণ তুমি V সংখ্যক বার DFS চালাচ্ছ।

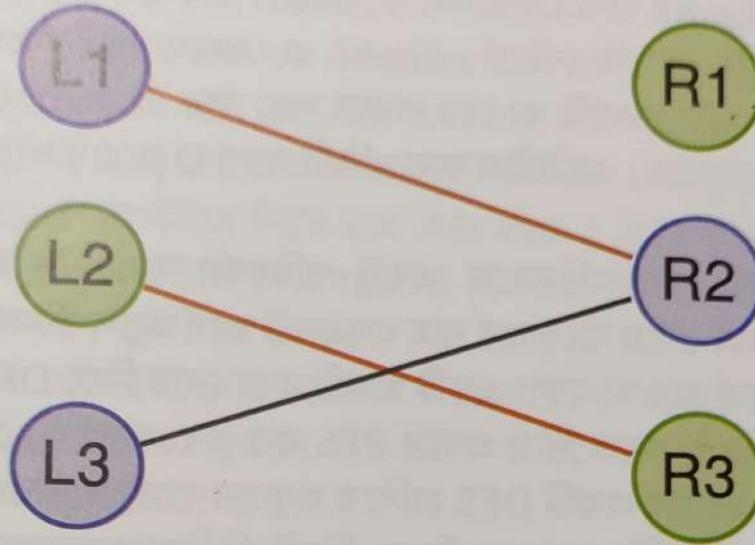
তবে তুমি যদি এই অ্যালগরিদমকে একটু পরিবর্তন কর তাহলে $O(E\sqrt{V})$ টাইম কমপ্লেক্সিটির অ্যালগরিদম পেয়ে যাবে যার নাম হপক্রফট কার্প অ্যালগরিদম (Hopcroft Karp Algorithm)। যা করতে হবে তা হলো একটি একটি করে নোড নিয়ে DFS না করে, L এর সব আনম্যাচড (unmatched) নোড নিয়ে প্রথমে BFS কর এবং BFS ট্রি তে L এর সব নোডের গভীরতা লিখে রাখো। এর পর একটি DFS চালিয়ে শুধুমাত্র পরের গভীরতার নোডে যেতে চেষ্টা করো এবং দেখ একটি অগমেন্টিং পাথ পাও কিনা। কিছুটা ডিনিকের অ্যালগরিদমের মতো। এটিই তোমাকে $O(E\sqrt{V})$ টাইম কমপ্লেক্সিটি দিবে। আমি মূল ধারনাটি সহজভাবে বললাম। ইন্টারনেটে আরও বিস্তারিত পড়তে পার।

৮.১৩.৫ ভার্টেক্স কাভার (Vertex cover) ও ইনডিপেন্ডেন্ট সেট (Independent set)

ম্যাঞ্চিমাম বাইপারটাইট ম্যাচিং সমস্যার সঙ্গে জড়িত দুটি সমস্যা হলো ভার্টেক্স কাভার (vertex cover) এবং ইনডিপেন্ডেন্ট সেট (independent set)। ভার্টেক্স কাভার বলে এমন কিছু নোড নির্বাচন করতে যেন সব বাহুর কোনো একটি মাথা যেন অবশ্যই নির্বাচিত ভার্টেক্সগুলোর মধ্যে একটি হয়। যেহেতু আমরা চাইলেই সব নোড নির্বাচন করতে পারি তাই আমাদের লক্ষ্য হলো সবচেয়ে কম সংখ্যক নোড নির্বাচন করা। এ জন্য এই সমস্যাকে বলা হয় মিনিমাম ভার্টেক্স কাভার (minimum vertex cover)। ইনডিপেন্ডেন্ট সেট ঠিক এর উল্লেখ সমস্যা। এই সমস্যায় বলে এমন কিছু নোড নির্বাচন করতে যেন তাদের মধ্যে কোনো বাহু না থাকে। যেহেতু আমরা চাইলেই মাত্র একটি নোড নির্বাচন করতে পারি সেহেতু আমাদের লক্ষ্য হলো সবচেয়ে বেশি সংখ্যক নির্বাচন করা, সেজন্য একে বলা হয় ম্যাঞ্চিমাম ইনডিপেন্ডেন্ট সেট (maximum independent set)। কথা হলো সাধারণ থাকে এই দুটি সমস্যা NP অর্থাৎ আমরা জানি না এদের ভালো কোনো সমাধান আছে কিনা। তবে বাইপারটাইট থাকে এদের খুব সুন্দর সমাধান আছে। সেজন্য আমরা এখানে

বাইপারটাইট গ্রাফে এই দুটি সমস্যা দেখব।

প্রথমত বলে নেই যে ম্যাঞ্জিমাম বাইপারটাইট ম্যাচিং এবং মিনিমাম ভার্টেক্স কাভার সমস্যা হলো dual. অর্থাৎ দুটির উভয় সমান হবে। এখন কথা হলো আমরা এরকম একটি ভার্টেক্স সেট কীভাবে বের করব? খুব একটা কঠিন না। আমার কাছে সহজ লাগে এক্ষেত্রে সমস্যাটিকে ম্যাঞ্জিমেন্ট কর। অবশ্যই L এর কিছু অংশ হবে S এর অংশে এবং বাকি অংশ T এর অংশে। মনে করি এরা $L_T \cup R_S$. অন্তত লাগলেও এটি সত্য। একটি উদাহরণ দেখা যাক।



নকশা ৮.১০: বাইপারটাইট ম্যাচিং (Bipartite matching), ভার্টেক্স কাভার (Vertex cover) ও ইনডিপেন্ডেন্ট সেট (Independent set)

চিত্র ৮.১০ এ লাল বাহু দিয়ে ম্যাচিং বাহু বোঝানো হয়েছে। এখন তুমি মিনিমাম এর অ্যালগরিদম চালালে নীল নোডগুলো S অংশ আকারে চিহ্নিত হবে এবং সবুজ নোডগুলো T অংশ হিসেবে। সুতরাং আমাদের ভার্টেক্স কাভার হবে $L_T \cup R_S = \{L2, R2\}$. তোমরা হয়তো জিজ্ঞাসা করতে পার এই S ও T অংশ বের করার কী কোনো সহজ উপায় নেই? মিনিমাম করতে হবে? না অবশ্যই সহজ উপায় আছে। কল্পনা কর এটি যদি ফ্লো এর গ্রাফ হতো তাহলে সোর্স S হতে BFS করলে কোন কোন নোডে যেতে? L এর সেসব নোডে যাদের ম্যাচিং নেই। সুতরাং BFS এর জন্য একটি কিউতে এসব নোড ঢুকিয়ে ফেল। এখন চিন্তা কর মিনিমাম এর BFS এর সময় তুমি কখন L হতে R এ যেতে? যখন এই বাহুটি ম্যাচিংয়ে থাকবে না তখনই তার $cf = 1$ হবে এবং তুমি সেই বাহু ব্যবহার করতে। সুতরাং আমাদের এখনকার BFS এর সময় L এর কোনো নোডে থাকলে তার নন-ম্যাচিং (non-matching) সব প্রতিবেশী নোডে যাও। এখন আবার দেখ মিনিমাম এ R এর কোনো নোডে থাকলে কী করতে? এটি L সাইডে যার সঙ্গে ম্যাচিং করা সেখানে যেতে, সুতরাং এখনকার BFS এও তুমি তাই করবে। এভাবে BFS করলেই হবে।

এখন একটি মজার জিনিস খেয়াল কর, ভার্টেক্স কাভার হলো এমন কিছু নোডের সেট যেখানে সব বাহুর অন্তত একটি মাথা আছেই। তাহলে এমন কোনো বাহু নেই যাদের দুটি মাথাই ভার্টেক্স কাভারের বাইরে তাই না? অর্থাৎ ভার্টেক্স কাভারের ঠিক কমপ্লিমেন্ট

(complement) হলো ইনডিপেন্ডেন্ট সেট। যদি আমরা ভার্টুয়েল কাভারকে মিনিমাইজেশন করি তাহলেই আমরা ইনডিপেন্ডেন্ট সেটকে ম্যাঞ্চিমাইজেশন করতে পারব। অর্থাৎ আমাদের উপরের উদাহরণে $\{L_2, R_2\}$ ছিল মিনিমাম ভার্টুয়েল কাভার, তাহলে $\{L_1, L_3, R_1, R_3\}$ হবে ম্যাঞ্চিমাম ইনডিপেন্ডেন্ট সেট। শেষ!

৮.১৩.৬ ওয়েইটেড ম্যাঞ্চিমাম বাইপারটাইট ম্যাচিং (Weighted maximum bipartite matching)

ম্যাঞ্চেন্সে এর যেমন ওয়েইটেড ভার্সন ছিল (মিনকস্ট ম্যাঞ্চেন্সে) ঠিক তেমনই ম্যাঞ্চিমাম বাইপারটাইট ম্যাচিং এরও ওয়েইটেড ভার্সন আছে। এটিই ওয়েইটেড ম্যাঞ্চিমাম বাইপারটাইট ম্যাচিং (weighted maximum bipartite matching)। আমাদেরকে ম্যাচিং বাহ্যসমূহের মূল্যকে ম্যাঞ্চিমাইজেশন বা মিনিমাইজেশন করতে বলে। দুটিই একই কথা। যদি আমরা ওজন (weight) গুলোকে -1 দিয়ে গুণ করি বা একটি বড় সংখ্যা ধর M থেকে সব বাহ্য মূল্য বিয়োগ করি তাহলেই ম্যাঞ্চিমাইজেশন সমস্যা মিনিমাইজেশন সমস্যায় পরিবর্তিত হয়ে যায়। এখন আমরা এই মিনিমাম ওয়েইটেড ম্যাঞ্চিমাম বাইপারটাইট ম্যাচিং (minimum weighted maximum bipartite matching) সমস্যা কীভাবে সমাধান করতে পারি? একটি সহজ উপায় হলো একে মিনকস্ট ম্যাঞ্চেন্সে দিয়ে সমাধান করা। আরেকটি উপায় হলো হাঙ্গেরিয়ান অ্যালগরিদম (hungarian algorithm)। এটি বেশ কঠিন মনে হয় আমার কাছে, আমি নিজেই এটি পারি না, তবে এর উপর topcoder এ আর্টিকেল আছে। তোমরা চাইলে সেই আর্টিকেল¹ পড়ে দেখতে পার। এর রান টাইম $O(V^3)$.

৮.১৪ প্রোগ্রামিং সমস্যা

৮.১৪.১ অনুশীলনী

সহজ

- UvaLive 6788 • UvaLive 6790 • UvaLive 6800 • UvaLive 6807 • UvaLive 6811 • UvaLive 6827 • UvaLive 6837 • UvaLive 6851 • UvaLive 6885 • UvaLive 6887 • UvaLive 6897 • UvaLive 6930 • UvaLive 6992 • UvaLive 6995 • UvaLive 6996 • UvaLive 7001 • UvaLive 7008 • UvaLive 7015 • UvaLive 7026 • UvaLive 7043 • UvaLive 7079 • UvaLive 7081

¹<https://www.topcoder.com/community/data-science/data-science-tutorials/assignment-problem-and-hungarian-algorithm/>

সামান্য কঠিন

ঃ UvaLive 6888 ঃ UvaLive 6891 ঃ UvaLive 6894 ঃ UvaLive 6910 ঃ UvaLive
6920 ঃ UvaLive 6941 ঃ UvaLive 6953 ঃ UvaLive 6970 ঃ UvaLive 6981
ঃ UvaLive 7016 ঃ UvaLive 7018 ঃ UvaLive 7021 ঃ UvaLive 7037** ঃ UvaLive
7042 ঃ UvaLive 7052 ঃ UvaLive 7168

কঠিন

ঃ UvaLive 6894 ঃ UvaLive 6920

অধ্যায় ৯

কিছু অ্যাডহক পদ্ধতি (Adhoc Technique)

অ্যাডহক (Adhoc) সমস্যা বলতে আমরা বুঝি আমাদের জানা কোনো নির্দিষ্ট ক্যাটাগরিতে না পড়া সমস্যা। একইভাবে অ্যাডহক পদ্ধতি হলো এমন কিছু পদ্ধতি যা নির্দিষ্টভাবে কোনো ভাগে ফেলা যায় না বা ফেলা কঠিন হয়। প্রোগ্রামিং প্রতিযোগিতা করতে গিয়ে প্রায়ই ব্যবহার করতে হয় এমন কিছু অ্যাডহক পদ্ধতি নিয়েই আমাদের এই অধ্যায়। অনেকে হয়তো এখানে আলোচিত সমস্যাগুলোকে বিভিন্ন ক্যাটাগরিতে(dp, greedy, math) ভাগ করতে চাইবে, কিন্তু আমি তা না করে অ্যাডহক সেকশনে রাখলাম।

৯.১ কিউমিউলেটিভ যোগফল পদ্ধতি (Cumulative sum technique)

কিউমিউলেটিভ যোগফল (Cumulative sum) মানে হলো পর পর অনেক জিনিসের যোগফল। ধরা যাক তোমার কাছে একটি n দৈর্ঘ্যের অ্যারে আছে। আমরা আমাদের সুবিধার জন্য অ্যারেকে $1 - index$ ধরব। আসলে ঠিক $1 - index$ না, কারণ আমরা মনে করব যে $0 - index$ বলে একটি জায়গা আছে যা আপাতত অব্যবহৃত। অর্থাৎ গাণিতিকভাবে লিখতে গেলে আমাদের অ্যারের সংখ্যাগুলো হলো $A[1 \dots n]$ এবং $A[0]$ আপাতত অব্যবহৃত আছে। আমাদের এই A অ্যারের জন্য কিউমিউলেটিভ সামের অ্যারে হবে S , যেখানে $S[i]$ হলো A অ্যারের $1 - index$ হতে $i - index$ পর্যন্ত সংখ্যাগুলোর যোগফল বা সাম (sum). যেহেতু আমাদের A অ্যারের দৈর্ঘ্য n সেহেতু স্বাভাবিকভাবেই S অ্যারের দৈর্ঘ্যও n . এখন কথা হলো আমরা কীভাবে বা কত দ্রুত এই কিউমিউলেটিভ যোগফলের অ্যারে বের করতে পারব? সবচেয়ে সহজ উপায় মনে হয়, আমরা S এর প্রত্যেকটি ইনডেক্স (index) এ যাব (n সংখ্যকবার) এবং i তম ইনডেক্সের জন্য $A[1 \dots i]$ যোগ করে কিউমিউলেটিভ যোগফল বের করব। এভাবে করতে গেলে আমাদের টাইম কমপ্লেক্সিটি (time complexity) দাঁড়াবে $O(n^2)$. একটু চিন্তা করলে দেখবে যে আমরা কিছু কাজ বার বার করছি। যেমন ধরা যাক, আমরা $S[i - 1]$ জানি অর্থাৎ আমরা $A[1 \dots i - 1]$ এর যোগফল জানি।

এখন এর পরে যখন $S[i]$ বের করতে যাব তখন কিন্তু আমাদের $A[1 \dots i]$ এর যোগফল পুরোটি নতুন করে বের করার দরকার নেই। বরং আগের $A[1 \dots i - 1]$ এর যোগফল অর্থাৎ $S[i - 1]$ এর সঙ্গে শুধু $A[i]$ যোগ করলেই কিন্তু $S[i]$ পেয়ে যাবে। এভাবে আমরা মাত্র $O(n)$ সময়েই পুরো কিউমিউলেটিভ যোগফলের অ্যারে বের করে ফেলতে পারব। অর্থাৎ আমরা $i = 1$ হতে $i = n$ পর্যন্ত লুপ চালাব এবং $S[i] = S[i - 1] + A[i]$ করব। একটু খেয়াল করলে দেখবে যে $S[1]$ বের করার সময় আমরা $S[0]$ ব্যবহার করছি, সুতরাং আমাদেরকে $S[0] = 0$ বসাতে হবে সবার শুরুতে। আমরা চাইলে লুপ 1 থেকে শুরু না করে 2 থেকে শুরু করতে পারতাম এবং $S[1] = A[1]$ বসাতে পারতাম। তবে মনে হয় $S[0] = 0$ বসানো অনেক বেশি সহজবোধ্য। এই অ্যারে ব্যবহার করে কিন্তু আমরা খুব সহজেই অ্যারের a হতে b পর্যন্ত যোগফল বের করে ফেলতে পারি। আমাদের সুত্র হবে: $S[b] - S[a - 1]$.

এই তো গেল এক মাত্রার বা $1 - dimensional$ ($1D$) অ্যারের কিউমিউলেটিভ যোগফল। আমরা চাইলে ত্রিমাত্রিক বা $2 - dimensional$ ($2D$) অ্যারেতেও এই পদ্ধতি ব্যবহার করতে পারি। $2D$ তে আমাদের সমস্যাটা দাঁড়াবে কিছুটা এরকম- আমাদের কাছে একটি $2D$ ম্যাট্রিক্স থাকবে। আমাদেরকে (a, b) হতে (c, d) পর্যন্ত বিস্তৃত আয়তক্ষেত্রের ভেতরের সংখ্যাসমূহের যোগফল বের করতে হবে। আগের মতোই আমরা ম্যাট্রিক্সের প্রত্যেক স্থান (i, j) এর জন্য $(1, 1)$ হতে (i, j) পর্যন্ত আয়তক্ষেত্রের ভেতরের সংখ্যার যোগফল বের করব। যদি আমাদের এই যোগফল রাখার অ্যারে হয় S এবং আমাদের মূল অ্যারে হয় A তাহলে আমরা লিখতে পারি: $S[i][j] = S[i - 1][j] + S[i][j - 1] - S[i - 1][j - 1] + A[i][j]$. তুমি একটি ছবি আঁকলেই বুঝতে পারবে কেন এই সুত্র সঠিক। এই সুত্র ব্যবহার করে আমরা সব প্রিফিক্স (prefix) যোগফল (আসলে প্রিফিক্স যোগফল বলা ঠিক হচ্ছে না কিন্তু তোমরা আশা করি বুঝতে পারছ আমি কী বোঝাতে চাচ্ছি) মাত্র $O(n^2)$ সময়েই বের করে ফেলতে পারব (ধরে নেই আমাদের মূল ম্যাট্রিক্সটি $n \times n$ আকারের)। এখন প্রশ্ন হলো আমরা (a, b) হতে (c, d) পর্যন্ত বিস্তৃত আয়তক্ষেত্রের ভেতরের সংখ্যাসমূহের যোগফল কীভাবে বের করব? খুব সহজ: $S[c][d] - S[a - 1][d] - S[c][b - 1] + S[a - 1][b - 1]$. আশা করি বুঝতে পারছ যে, ত্রিমাত্রিক বা $3 - dimensional$ ($3D$) অ্যারের ক্ষেত্রে সুত্রগুলো দেখতে কেমন হবে!

১.২ সর্বোচ্চ যোগফল পদ্ধতি (Maximum sum technique)

১.২.১ একমাত্রিক সর্বোচ্চ যোগফল সমস্যা (One dimensional Maximum sum problem)

মনে কর তোমাদের কাছে একটি সংখ্যার অ্যারে আছে। তোমাকে এই অ্যারে থেকে সর্বোচ্চ যোগফল (maximum sum) বিশিষ্ট সাব-অ্যারে (sub-array) বের করতে বলা হলো। অর্থাৎ তোমাকে এমন একটি সাব-অ্যারে বের করতে হবে যাতে করে সেই সাব-অ্যারেতে থাকা সংখ্যাগুলোর যোগফল অন্যান্য যেকোনো সাব-অ্যারেতে থাকা সংখ্যাগুলোর যোগফল থেকে বেশি

হয়। মনে করা যাক আমাদের অ্যারে হলো: $5, -10, 6, -2, 4, -10, 1$. তাহলে আমাদের উভয় হবে 8 যা $6, -2, 4$ এই সাব-অ্যারে নিলে পাওয়া যাবে।¹ খুবই সহজ পদ্ধতি হতে পারে আমরা সব সাব-অ্যারের জন্য একটি লুপ চালিয়ে তার যোগফল বের করব। কিন্তু এই পদ্ধতিতে আমাদের সময় লাগবে $O(n^3)$. কারণ আমাদের মোট সাব-অ্যারে আছে $O(n^2)$ টি এবং প্রতিটির যোগফল বের করতে সময় লাগবে $O(n)$. আমরা চাইলেই এই ভেতরের যোগফল বের করার লুপ বাদ দিয়ে দিতে পারি। আমরা তো একটি লুপ দিয়ে সাব-অ্যারের শুরুর মাথা নির্বাচন করছি। আরেক লুপ দিয়ে অপর মাথা। এখন মনে কর প্রথম লুপের মধ্যে একটি ভ্যারিয়েবল আছে sum . যখন দ্বিতীয় লুপ দিয়ে আমরা শেষ মাথা বাড়াচ্ছি তখন আমরা এই sum এর পরিমাণকে ঐ সংখ্যার পরিমাণ বাড়াতে পারি, তাহলে এই sum এর ভিতরে সবসময় সাব-অ্যারের যোগফল থাকবে। আমাদের আর অতিরিক্ত আরেকটা লুপ চালিয়ে যোগফল বের করার দরকার হবে না। সুতরাং আমাদের সময় লাগবে $O(n^2)$. আমরা কী কোনোভাবে কনজিকিউটিভ যোগফল (consecutive sum) এই পদ্ধতিটি ব্যবহার করতে পারি? খেয়াল কর, আমাদের কনজিকিউটিভ যোগফলের অ্যারে তৈরি করতে সময় লাগবে $O(n)$ আর প্রতিটি সাব-অ্যারের উপর লুপ চালিয়ে তার যোগফল আমাদের কনজিকিউটিভ যোগফলের অ্যারে থেকে বের করতে সময় লাগবে $O(1)$. যেহেতু আমাদের সাব-অ্যারে আছে প্রায় n^2 টি, সুতরাং আমাদের টাইম কমপ্লেক্সিটি $O(n^2)$. আমাদের খুব একটা ভাল হচ্ছে না।

একটু চিন্তা করে দেখ আমরা কনজিকিউটিভ যোগফলের অ্যারে ব্যবহার করে কীভাবে কোনো একটি সাব-অ্যারের মান বের করতে পারি? আমাদের i হতে j পর্যন্ত যোগফল হবে $S[j] - S[i-1]$. কোডের কথা চিন্তা করলে জিনিসটি এমন যে, বাইরে j এর লুপ চলবে, ভেতরে 1 হতে j পর্যন্ত i এর লুপ চলবে আর এই দুই লুপের মধ্যে আমরা $S[j] - S[i-1]$ এর মান বের করব আর তাদের সবার মধ্যে সর্বোচ্চটি বের করব। এখন বাইরের লুপ j এর জন্য ভেতরের লুপ i এর $[1, j-1]$ এর মধ্যে কেন মানের জন্য আমরা $S[j] - S[i-1]$ এর সর্বোচ্চ মানটা পাব? আমাদের এই চিন্তাধারাটা একটু ভালো করে দেখ। আমরা এখানে j কে নির্দিষ্ট করে ফেলছি এবং জানতে চাচ্ছি যে i এর কোনো মানের জন্য আমাদের উদ্দেশ্য সফল হবে। এখন তোমরাই বলো যদি $S[j] - S[i-1]$ এ $S[j]$ কে নির্দিষ্ট করা হয় তাহলে $S[j] - S[i-1]$ কে সর্বোচ্চ করার জন্য তোমরা কোন $S[i-1]$ কে নিবে? অবশ্যই সর্বনিম্ন $S[i-1]$ কে বা $S[0 \dots j-1]$ এর মধ্যের সর্বনিম্ন মানকে। আমরা কিন্তু চাইলেই বাইরের লুপ j এর মধ্যেই $1 \dots j-1$ এর সর্বনিম্ন মান বের করে ফেলতে পারি, আলাদা করে ভেতরের লুপের দরকার নেই। আসলে খেয়াল করলে দেখবে তুমি যদি $S[1 \dots j]$ এর মধ্যের সর্বনিম্নটি বের কর তাহলে ক্ষতি নেই। সুতরাং আমরা এভাবে খুব সহজেই $O(n)$ এ আমাদের সর্বোচ্চ যোগফল বিশিষ্ট সাব-অ্যারে বের করে ফেলতে পারছি।

উপরে দেখানো পদ্ধতিতে সর্বোচ্চ যোগফল বিশিষ্ট সাব-অ্যারে বের করা অনেক বেশি সহজবোধ্য মনে হয় আমার কাছে। তবে তোমরা অনেকেই হয়তো ক্লাসে বা অন্যান্য অ্যালগরিদমের শেই এ অন্য একটি পদ্ধতি দেখেছ। সেই পদ্ধতি বলা অনেক সহজ কিন্তু সেটি কেন সঠিক তা বোঝা উচিত। সেটা সহজ মনে হয় না, সত্যি কথা বলতে সেটি কেন সঠিক তা আমি নিজেকে কীভাবে সন্তুষ্ট করব-এটি চিন্তা করতে বেশ খানিকটা সময় ব্যয় করে ফেলেছি। যাই হোক আগে পদ্ধতিটি বলে নেই।

¹অনেকে সাব-অ্যারে কে contiguous subsequence নামে চেনে।

তোমরা একটি ভ্যারিয়েবল রাখবে ধরা যাক তার নাম *sum*. এখন অ্যারের শুরু থেকে শেষ পর্যন্ত যাও। প্রত্যেক জায়গায় সেখানের মান *sum* এ যোগ কর। যদি দেখ *sum* এর মান ঝণাত্বক হয়ে গেছে তাহলে *sum* কে 0 করে দাও। এভাবে প্রত্যেক জায়গায় *sum* এর যেই মান পাছ তাদের মধ্যে সর্বোচ্চটাই আমাদের উত্তর। আমরা এই সর্বোচ্চ বের করার জন্য *max* নামের একটি ভ্যারিয়েবল ব্যবহার করব এবং সেখানে এ পর্যন্ত পাওয়া সর্বোচ্চ *sum* রাখব। এখন কথা হলো এটি কেন সঠিক? প্রথমত প্রতিটি স্থানে পৌঁছে *sum* এর মান হবে ওই পর্যন্ত শেষ হওয়া সব সাব-অ্যারের সর্বোচ্চ যোগফল। গাণিতিক আরোহ বা ইনডাকশন (induction) এর মতো করে চিন্তা কর। এই কথাটি () তে থাকা অবস্থায় সঠিক ছিল কারণ *sum* কে আমরা 0 দ্বারা ইনিশিয়ালাইজেশন (initialization) করেছি (এই কথা আগে বলি নাই কিন্তু এতক্ষণে তোমাদের এটা বুঝে যাবার কথা)। এখন মনে কর $i - 1$ অবস্থানে $i - 1$ এ শেষ হওয়া সাব-অ্যারের মধ্যে সর্বোচ্চ যোগফল *sum* এ আছে। তাহলে আমরা যখন i এ যাব তখন আমাদের পদ্ধতিতে *sum* এ i পর্যন্ত শেষ হওয়া সাব-অ্যারের মধ্যে সর্বোচ্চ যোগফল কি পাব? অবশ্যই, কারণ i এ শেষ হতে গেলে আমাদের i তম উপাদানকে নিতে হবে এবং $i - 1$ এ শেষ হওয়া সর্বোচ্চ যোগফল বিশিষ্ট সাব-অ্যারেকে নিতে হবে। তবে এই সর্বোচ্চটি যদি আবার ঝণাত্বক হয় তাহলে কিন্তু এটি না নেওয়াই ভালো অর্থাৎ i পর্যন্ত শেষ হওয়া ফাঁকা অ্যারে আমাদের সর্বোচ্চ মান দিবে এরকম চিন্তা করতে পার। একটি কথা বলে রাখা যেতে পারে যে, আমাদের এই পদ্ধতিতে উত্তর কিন্তু কমপক্ষে 0 হবে। এর যুক্তি হলো, আমরা যদি ফাঁকা সাব-অ্যারে নেই তাহলেই 0 পাওয়া সম্ভব। তবে যদি এটি বলা থাকে যে সাব-অ্যারের দৈর্ঘ্য কমপক্ষে 1 হতে হবে তাহলে তোমাদের এই পদ্ধতিকে সামান্য পরিবর্তন করতে হবে। কী পরিবর্তন করতে হবে এটি তোমরা নিজেরা বের করে নিও।

১.২.২ দ্বিমাত্রিক সর্বোচ্চ যোগফল সমস্যা (Two dimensional Maximum sum problem)

তোমাকে দ্বিমাত্রিক বা $2D$ একটি অ্যারে দেওয়া আছে, তোমাদের এমন একটি আয়তক্ষেত্র বের করতে হবে যার মধ্যে থাকা সব সংখ্যার যোগফল সর্বোচ্চ হয়। আশা করি তোমরা এই প্রবলেমের সরল সমাধান (naive solution) বের করে ফেলেছ যার কমপ্লেক্সিটি $O(n^6)$. এই সমাধানে আমরা আয়তক্ষেত্রের দুই কোনা চারটি লুপ চালিয়ে বের করব এবং আরও দুটি লুপ দিয়ে এই আয়তক্ষেত্রের ভেতরের সংখ্যাগুলোকে যোগ করব। আমরা চাইলে প্রতি কলাম (column) বা সারি (row) তে কনজিকিউটিভ যোগফল পদ্ধতি ব্যবহার করে কমপ্লেক্সিটি কিছুটা কমিয়ে ফেলতে পারি। ধরা যাক আমরা প্রতিটি কলামে কনজিকিউটিভ যোগফলের অ্যারে রেখেছি। এখন ধরি আমরা আয়তক্ষেত্রের দুই কোনা চারটি লুপ চালিয়ে নির্ধারণ করেছি এবং তারা হলো: (a, b) এবং (c, d) যেখানে $a \leq c, b \leq d$ এবং a ও c ধরা যাক সারিসংখ্যা এবং b ও d কলামসংখ্যা। তাহলে আমরা এই চারটি লুপের ভেতরে আরও একটি লুপ চালাব $[b, d]$ সীমায় এবং প্রতি কলামে আমরা সেই কলামের কনজিকিউটিভ যোগফলের অ্যারে ব্যবহার করে a হতে c পর্যন্ত সংখ্যাগুলোর যোগফল বের করে ফেলতে পারি। এই পদ্ধতিতে আমাদের কমপ্লেক্সিটি হবে $O(n^5)$. আমরা চাইলে $2D$ অ্যারেতে কনজিকিউটিভ যোগফল পদ্ধতি ব্যবহার করে $O(n^4)$ এ সমাধান করতে পারি। দুই কোনা নির্ধারণ

করার পর তো আমাদের আগের সেকশনের $S[c][d] - S[a-1][d] - S[c][b-1] + S[a-1][b-1]$
সুত্র ব্যবহার করে ভেতরের সব সংখ্যার যোগফল বের করে ফেলতে পারি।

আমার জানা এই সমস্যার সবচেয়ে ভালো সমাধান হলো $O(n^3)$. এই সমাধান খুব একটা
কঠিন না। আমাদেরকে প্রত্যেক কলামের জন্য কনজিকিউটিভ যোগফলের অ্যারে তৈরি করে রাখতে
হবে। এবার দুটি লুপ চালিয়ে আয়তক্ষেত্রের উপরের আর নিচের দুই সারিকে নির্দিষ্ট করে ফেলব।
এখন প্রত্যেক কলামের জন্য নির্দিষ্ট করা দুই সারির মধ্যের অংশের যোগফল বের করব। তাহলে
আমরা আসলে একটি $1D$ অ্যারে পাব। একটু চিন্তা করে দেখ তো আমরা এখন যদি $1D$ তে
সর্বোচ্চ যোগফল সমস্যা সমাধান করি তাহলেই $2D$ এর সমস্যাটি সমাধান হয়ে যায় কিনা! এখানে
আমরা দুটি সারি নির্বাচন করেছি $O(n^2)$ এ। এর পর প্রতি কলামে গিয়ে গিয়ে ঐ দুই সারির
মধ্যের সংখ্যাগুলোর যোগফল বের করেছি। কলামের উপর দিয়ে লুপ চালাতে লাগে $O(n)$ আর
কিউমিউলেটিভ যোগফলের অ্যারে হতে যোগফল বের করতে লাগে $O(1)$. এর পর আমরা একটি
 $1D$ তে সর্বোচ্চ যোগফল সমস্যার সমাধান $O(n)$ এ করি। আমাদের এখানে তিনটি নেস্টেড
(nested) লুপ লাগছে (শেষ দুটি লুপ কিন্তু একটি আরেকটির ভেতরে না বরং সমান্তরাল, চাইলে
তোমরা দুটি লুপ না করে কিউমিউলেটিভ যোগফল বের করা আর $1D$ এ সর্বোচ্চ যোগফল বের
করার কাজ একই লুপে করতে পারো)। তাই মোট কমপ্লেক্সিটি $O(n^3)$.

৯.৩ প্যাটার্ন (Pattern) খোঁজা

৯.৩.১ LightOJ 1008

আমাদের টেবিল ৯.১ এর মতো একটি ছক থাকবে। বলতে হবে n সংখ্যাটি কোন সারি এবং
কোন কলামে আছে। n এর মান সর্বোচ্চ 10^{15} হতে পারে।

টেবিল ৯.১: LightOJ 1008 সমস্যার টেবিল

10	11	12	13
9	8	7	14
2	3	6	15
1	4	5	16

n এর এত বড় মান দেখে বোঝাই যাচ্ছে যে আমরা কোনোভাবেই 1 হতে n প্রতিটি মান বসিয়ে
বসিয়ে এত বড় টেবিল তৈরি করতে পারব না। এসব সমস্যার ক্ষেত্রে প্রধানত যা করতে হয় তা
হলো এই টেবিলকে ভালো মতো পর্যবেক্ষণ করতে হয়। খেয়াল করলে দেখবে যে নিচের বাম কোনা
হতে x আকারের বর্গে 1 হতে x^2 পর্যন্ত সব সংখ্যা থাকে। এই সাবসেকশনে আমরা এখন থেকে বর্গ
বলতে নিচের বাম কোনা থেকে শুরু হওয়া বর্গ বুঝব। সুতরাং আমরা যদি কোনো একটি সংখ্যা, ধরা

যাক 85 নেই তাহলে এটি 10 আকারের বর্গের ভেতরে থাকবে কারণ $9^2 = 81$ আর $10^2 = 100$.
 তাহলে কোনো একটি সংখ্যা n দেওয়া থাকলে সে কত আকারের বর্গে থাকবে তা আমরা কীভাবে
 বের করতে পারি? বর্গমূল বের করলে তো ভগ্নাংশযুক্ত সংখ্যা আসতে পারে।
 তাহলে? উপায় হলো আমাদের হয় $floor$ নিতে হবে অথবা $ceiling$ নিতে হবে। কিন্তু কোনটা?
 এক্ষেত্রে আমাদের উদাহরণ নিয়ে কাজ করতে হবে। ধরা যাক, $n = 3$, তাহলে $\sqrt{3} = 1.732 \dots$
 কিন্তু আমরা দেখতে পাচ্ছি এটি 2 আকারের বর্গে আছে, সুতরাং আমাদের $ceiling$ নিতে হবে। বা
 অন্যভাবে বলতে এমন একটি সর্বনিম্ন মান x খুঁজে বের করতে হবে যেন $x^2 \geq n$ হয়। তোমরা
 চাইলে বাইনারি সার্চ (binary search) করে এই x বের করতে পার বা `math.h` এর `sqrt`
 ফাংশন ব্যবহার করতে পার। তবে `sqrt` ফাংশন ব্যবহার করতে চাইলে পূর্ণ বর্গ সংখ্যা এর বর্গমূল
 বের করার সময় একটু খেয়াল রাখতে হবে। এসব ক্ষেত্রে সাবধানতার জন্য আমি যা করি তা হলো
 নিজে একটি `sqrt` ফাংশন লিখি যেখানে `math.h` এর `sqrt` ফাংশন কল করি এবং তার $ceiling$
 নেই বা `int` টাইপে টাইপকাস্টিং (type-casting) করি। ধরা যাক এই মান হলো y . এখন আমি
 $y - 1$ থেকে $y + 1$ পর্যন্ত একটি লুপ চালাই এবং এই লুপ চালিয়ে সিদ্ধান্ত নেই যে কোন মানটা
 আসলে সঠিক। আরেকটি জিনিস, তা হলো এই যে আমরা বের করলাম যে আমাদের $ceiling$ নিতে
 হবে বা $floor$ নিতে হবে এসব সুত্র বের করার সময় বাউন্ডারি কেইস (boundary case) দিয়ে
 সুত্র ভালো করে যাচাই করে দেখতে হবে। যেমন আমাদের এই প্রবলেমে বাউন্ডারি কেইস হবে
 $1, 2, 4, 5, 9, 10, 16 \dots$ অনেক সময় দেখা যায় তুমি যেই সুত্র বের করেছ তা বাউন্ডারি কেইসে
 কাজ করে না। যাই হোক, আমরা তাহলে পেয়ে গেলাম কোন বর্গে আমাদের সংখ্যা আছে। যদি $x = \lceil \sqrt{n} \rceil$ হয় তাহলে n হয় x -তম সারিতে না হয় x -তম কলামে আছে (ধরে নিলাম আমাদের টেবিলটি
 নিচ হতে উপরে এবং বাম হতে ডান দিকে $1 - indexed$)। কিন্তু কোনটি সত্য? যদি টেবিলের দিকে
 তাকাও তাহলে দেখবে যে x যদি জোড় হয় তাহলে আমাদের বর্গের শেষ ফিতাটি বাম হতে নিচে
 আসে, আর যদি বিজোড় হয় তাহলে নিচ থেকে বামে যায়। আমাদের এই ফিতাটির এক পাশে
 আকার x . সুতরাং আমরা যদি জানি যে n এই শেষ ফিতাটিতে কত তম সংখ্যা তাহলেই আমাদের
 সমাধান হয়ে যায়। খেয়াল কর আমরা যদি x তম ফিতায় থাকি তাহলে এর আগের ফিতাটির শেষ
 সংখ্যা হলো $(x - 1)^2$ সুতরাং আমাদের সংখ্যাটি শেষ ফিতাটির $n - (x - 1)^2$ তম সংখ্যা। এখন
 দেখতে হবে x জোড় না বিজোড়। ধরা যাক বিজোড়। তাহলে দেখতে হবে $y = n - (x - 1)^2$ কি
 x এর থেকে বড় নাকি না। যদি বড় না হয় তাহলে এর $row = y$ এবং $column = x$ আর যদি
 বড় হয় তাহলে $row = x$ এবং $column = 1 + x^2 - n$. আশা করি কেন কলামের সুত্র এরকম
 হলো তা বোঝাতে হবে না। একইভাবে x জোড় হলে সারি বা কলামের সুত্র কী হবে তা বের করে
 নিতে পারবে।

৯.৩.২ জোসেফাস সমস্যা (Josephus Problem)

এই সমস্যার একটি গল্প আছে। গল্পটি এরকম এক দিন জোসেফাস তার 40 জন সৈন্যসহ^১
 একটি গুহায় আটকা পড়ে। গুহার মুখে রোমান সৈন্যরা ছিল। সুতরাং তারা সিদ্ধান্ত নেয় যে তারা একে
 একে আত্মহত্যা করবে। এজন্য তারা বৃত্তাকারভাবে দাঁড়ায়। বৃত্তের এক স্থান থেকে শুরু হয়। প্রথম

জন আত্মহত্যা করে এরপর পরের জনকে বাদ দিয়ে এর পরের জন আত্মহত্যা করে, এরপর এক জনকে বাদ দিয়ে তার পরের জন এভাবে চলতে থাকে। এভাবে চলতে চলতে একসময় মাত্র দুইজন থাকি থাকে, তাদের একজন ছিল জোসেফাস। তারা দুইজন আর আত্মহত্যা না করে রোমানদের হাতে আত্মসমর্পণ করে।

যাই হোক, আমরা এই দুঃখের গল্প মাথা থেকে সরিয়ে ফেলি এবং চিন্তা করি n জন মানুষ থাকলে একজন কোথায় দাঁড়ালে সে সর্বশেষ জীবিত মানুষ (last man standing) হবে। যেহেতু এই সেকশনটি প্যাটার্ন খোঁজার সুতরাং আমরা প্যাটার্নের মাধ্যমে এই সমস্যা সমাধানের চেষ্টা করব। আমরা বিভিন্ন n এর মানের জন্য সর্বশেষ জীবিত মানুষের অবস্থান বা ইনডেক্স বের করি (ধরে নেই 1-indexed). টেবিল ৯.২ এ $n = 1$ হতে 15 এর জন্য সর্বশেষ জীবিত মানুষের ইনডেক্স দেওয়া হলো।

টেবিল ৯.২: জোসেফাস (Josephus) সমস্যায় n এর বিভিন্ন মানের জন্য সর্বশেষ জীবিত মানুষের ইনডেক্স

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15

আশা করি তোমরা প্যাটার্নটি বুঝতে পারছ? হয়তো কেউ কেউ বুঝতে পারছ যে প্যাটার্ন কিছু একটি আছে কিন্তু সেটিকে হয়তো নির্দিষ্ট করে বুঝতে পারছ না। যাই হোক, তাহলে খেয়াল কর, থেকে 2 এর ঘাতে আমাদের উত্তর হয় 1 আর এর পর পরবর্তী 2 এর ঘাত পর্যন্ত উত্তর 2 করে বাঢ়তে থাকে। অর্থাৎ n দিলে আমাদের প্রথম কাজ হলো এমন একটি সবচেয়ে বড় x বের করা যেন $2^x \leq n$ হয়। তাহলে $2(n - 2^x) + 1$ হবে উত্তর।

তোমরা চাইলে একে রিকার্শন (recursion) এর মাধ্যমেও সমাধান করতে পার। মনে কর তুমি জানতে চাইছ যে n জনের জন্য উত্তর কত। তাহলে আগে $n - 1$ এর জন্য উত্তর বের করে নাও। এখন তুমি n জনের প্রথম জনকে কাটো এবং 3 নম্বর জনের কাছে যাও। তুমি কিন্তু জানো একে যদি 1 ধর তাহলে কোন জন বেঁচে যায়, তাহলে এ যদি 3 নম্বর হয় তাহলে কততম জন বেঁচে যাবে সেটি আশা করি বের করতে পারবে? এই পদ্ধতি n এর ছোট মানে কাজ করবে। এছাড়াও যদি "এক জনকে বাদ দিয়ে পরের" জন না বলে " k জনকে বাদ দিয়ে পরের জন" বলত তাহলেও কাজ দিবে। তোমরা চাইলে ডোনাল্ড নুথ (Donald Knuth) এর Concrete Mathematics গইয়ে এ নিয়ে আরও বিস্তর পড়াশোনা করতে পার।

৯.৪ একটি নির্দিষ্ট সীমায় সর্বোচ্চ উপাদান

৯.৪.১ একমাত্রিক (One Dimensional বা 1D)

মনে কর তোমাকে একটি 1D আয়ারে দেওয়া আছে। এখন তোমাকে যেকোনো h আকারের একটি সাব-অ্যারে দিয়ে জিজ্ঞাসা করা হবে এই সীমার মধ্যে সর্বোচ্চ সংখ্যা কত? সব সময়ই তোমাকে h আকারের সাব-অ্যারেই জিজ্ঞাসা করা হবে তবে সেই সাব-অ্যারে বিভিন্ন জায়গায় শুরু হতে পারে। তুমি কীভাবে খুব দ্রুত এই সমস্যা সমাধান করতে পারবে? যদি অ্যারের আকার n হয় তাহলে তো আমরা $O(nh)$ এ সমাধান করতে পারি। আমরা $O(nh)$ এ সব স্থানের জন্য উত্তর বের করে একটি অ্যারেতে রেখে দিব এবং এর পর তুমি যেই সাব-অ্যারের কথাই জিজ্ঞাসা কর না কেন তুমি সেই উত্তরের অ্যারে দেখে বলে ফেলতে পারবে। কিন্তু $O(nh)$ খুব বেশি হয়ে যায়। তোমরা চাইলে square root segmentation বা সেগমেন্ট ট্রি (segment tree) ব্যবহার করে একে $O(n\sqrt{n})$ বা $O(n \log n)$ এও সমাধান করতে পার। তবে মজার ব্যাপার হলো আমরা একে $O(n)$ এ সমাধান করতে পারি। তোমরা যারা RMQ এর টারজান (Tarjan) এর $O(n)$ সমাধান জানো তাদের বলে রাখি যে, সেই সমাধানের constant factor অনেক বেশি। সুতরাং আমরা সেই সমাধান বলব না। বরং আমাদের যে বলা আছে যে আমাদের কুয়েরি (query) সাব-অ্যারে সবসময় h আকারের হবে সেই তথ্য কাজে লাগাব।

প্রথমে অ্যারের প্রতিটি ইনডেক্সকে $\text{mod } h$ ভাবে কল্পনা কর। এখন আমাদের দুটি অ্যারের দরকার হবে। ধরা যাক একটি হলো A এবং অপরটি B . $A[i]$ তে থাকবে i থেকে শুরু করে i বা i এর পরের $h - 1 \text{ mod }$ ওয়ালা ইনডেক্স পর্যন্ত সাব-অ্যারের সর্বোচ্চ সংখ্যা। আর $B[i]$ এ থাকবে i বা এর আগের 0 mod ওয়ালা ইনডেক্স পর্যন্ত সাব-অ্যারের সর্বোচ্চ সংখ্যা। যেমন $h = 4$ এর ক্ষেত্রে চিত্র ৯.৩ এ আমরা দুটি সারিতে A এবং B এর সীমা দেখালাম। A আর B এর অ্যারে বের করতে কিন্তু আমাদের $O(n)$ সময় লাগবে। B বের করার জন্য তুমি মূল অ্যারের সামনে থেকে পেছনে লুপ চালাবে। যদি দেখ তুমি 0 mod ওয়ালা ইনডেক্সে আছ তাহলে তো মূল অ্যারের সংখ্যাই তোমার সর্বোচ্চ সংখ্যা, আর যদি তা না হয় তাহলে মূল অ্যারের এই ইনডেক্সের সংখ্যা আর B অ্যারের আগের ইনডেক্সের সংখ্যার সর্বোচ্চ সংখ্যাই B এর এই ইনডেক্সের সংখ্যা হবে। একইভাবে তুমি মূল অ্যারের পেছন থেকে আসলে A কে $O(n)$ এ বের করতে পারবে। এখন কথা হলো তোমাকে (a, b) সীমার জন্য কুয়েরি করা হলো ($b - a + 1 = h$) তাহলে কীভাবে এই সীমার সর্বোচ্চ সংখ্যা বের করবে? খুব সহজ, $\max(A[a], B[b])$ হলো উত্তর। কেন? কারণ a হতে তুমি পরের $h - 1 \text{ mod }$ ওয়ালা ইনডেক্সের সর্বোচ্চ সংখ্যা পাছ A হতে আর b হতে এর আগের 0 mod ওয়ালা ইনডেক্সের সর্বোচ্চ সংখ্যা পাছ B থেকে। এটি দিয়েই তোমার পুরো সীমা ঘোরা হয়ে যাচ্ছে। যেহেতু তোমার কুয়েরি সাব-অ্যারের আকার সবসময় h সুতরাং এই সাব-অ্যারের মধ্যে সবসময় একটি (এবং কেবল মাত্র একটি) 0 mod ওয়ালা ইনডেক্স পাওয়া যাবেই। তাই এই পদ্ধতি সবসময় কাজ করবেই। সুতরাং আমরা এভাবে $O(n)$ প্রিপ্রেসিং (preprocessing) এ $O(1)$ সময়ে কুয়েরির উত্তর দিতে পারি।

টেবিল ৯.৩: একটি নির্দিষ্ট রেঞ্জে সর্বোচ্চ উপাদান বের করার জন্য $h = 4$ এ A ও B এর অ্যারে

index	0	1	2	3	4	5	6	7
index mod 4	0	1	2	3	0	1	2	3
A	[0, 3]	[1, 3]	[2, 3]	[3, 3]	[4, 7]	[5, 7]	[6, 7]	[7, 7]
B	[0, 0]	[0, 1]	[0, 2]	[0, 3]	[4, 4]	[4, 5]	[4, 6]	[4, 7]

৯.৪.২ দ্বিমাত্রিক (Two Dimensional বা 2D)

এবার 2D তে একই সমস্যা কীভাবে সমাধান করা যায় দেখা যাক। মনে কর $n \times m$ আকারের একটি অ্যারে আছে, তোমাকে প্রতিবার কোনো একটি $p \times q$ আকারের সাব-অ্যারের জন্য সর্বনিম্ন সংখ্যা বা সর্বোচ্চ সংখ্যা জিজ্ঞাসা করা হবে। কীভাবে সমাধান করবে? খুব একটা কঠিন না। তুমি আগে প্রত্যেক সারির জন্য আগের উপায়ে q আকারের সাব-অ্যারের সর্বনিম্ন সংখ্যা বা সর্বোচ্চ সংখ্যা বের করে ফেল। তাহলে তুমি একটি $n \times (m - q + 1)$ আকারের একটি সাব-অ্যারে পাবে। এবার প্রত্যেক কলামের জন্য p আকারের সাব-অ্যারের সর্বনিম্ন সংখ্যা বা সর্বোচ্চ সংখ্যা বের কর। তাহলে পাবে $(n - p + 1) \times (m - q + 1)$ আকারের সাব-অ্যারে। এটিই তোমাকে শেষ উত্তর দিচ্ছে। অর্থাৎ এই পরিবর্তিত অ্যারের কোনো একটি অবস্থান (a, b) তোমাকে $(a, b) - (a + p - 1, b + q - 1)$ সাব-অ্যারের সর্বোচ্চ সংখ্যা বা সর্বনিম্ন সংখ্যা দিবে।

৯.৫ লীস্ট কমন অ্যানসেস্টর (Least Common Ancestor)

মনে কর একটি ট্রি দেওয়া আছে। এখন দুটি নোড দিয়ে জিজ্ঞাসা করা হবে তাদের লীস্ট কমন অ্যানসেস্টর (least common ancestor) কে? লীস্ট কমন অ্যানসেস্টর হলো সবচেয়ে নিচের এমন একটি নোড যেটি কুয়েরির দুই নোডেরই অ্যানসেস্টর (ancestor). আমাদের বর্ণনার সুবিধার জন্য ধরে নেই এই কুয়েরির দুইটি নোড হলো x এবং y . $O(n)$ এ সমাধান করা তে খুব সহজ কিন্তু আমরা চাই আরও দ্রুত এই কুয়েরির উত্তর দিতে। আমরা এখানে কীভাবে $O(n \log(n))$ এ প্রিপ্রেসিং করে $O(\log(n))$ এ এই কুয়েরির উত্তর দেওয়া যায় তা দেখব। ধরে নেই $\lceil \log(n) \rceil = 17$ তাহলে আমরা $parent[17 + 1][n]$ আকারের একটি অ্যারে নিব। এখন প্রতিটি নোড i এর জন্য আমরা $parent[0][i]$ এ i এর প্যারেন্ট (parent) রাখব। রুট ($root$) এর ক্ষেত্রে আমরা চাইলে রুটকেই রাখতে পারি বা কোনো একটি sentinel যেমন -1 ব্যবহার করতে পারি। তবে আমার মতে রুটকে রাখাই ভালো, অর্থাৎ 0 যদি রুট হয় তাহলে $parent[0][root] = 0(root)$.

সবই তো বোৰা গেলো কিন্তু $parent[j][i]$ এর মানে কি? $parent[j][i]$ এর মানে হলো এর 2^j তম প্যারেন্ট। যেমন $j = 0$ তে আছে এর $2^0 = 1$ তম প্যারেন্ট। $j = 1$ এ থাকবে এর প্যারেন্টের প্যারেন্ট অর্থাৎ বলতে পারো দাদা বা গ্র্যান্ডপ্যারেন্ট (grandparent)।

$j = 2$ তে থাকবে তার দাদার দাদা (গ্যান্ডপ্যারেন্টের গ্যান্ডপ্যারেন্ট)। এরকম করে। এই জিনিস তৈরী করার উপায় হলো তুমি DFS কর। DFS এর সময় যেই নোডে আসবে সেই নোডের $parent[0 \dots 17][at]$ পূরণ করতে হবে। এটি পূরণ করার উপায় হলো $parent[j][at] = parent[j - 1][parent[j - 1][at]]$ । মানে at এর 2^{j-1} তম প্যারেন্টের 2^{j-1} তম প্যারেন্ট। এভাবে তুমি সব নোডের জন্য $parent$ এর অ্যারে $O(n \log(n))$ সময়ে পূরণ করে ফেলতে পারবে। যেহেতু আমাদের ট্রি তে n টি নোড আছে, আর আমরা 2 এর ঘাতে প্যারেন্ট দেখছি তাই কোনো নোড হতে রঞ্টে যাবার জন্য $\log n$ এর বেশি ধাপের দরকার নেই। উদাহরণ দেওয়া যাক, মনে কর $n = 10$, তাহলে $\lceil \log(10) \rceil = 4$ (আশা করি বোৰা যাচ্ছে যে আমরা 2 ভিত্তিক লগারিদম (logarithm) নিয়েছি)। তাহলে যেকোনো নোড হতে $j = 4$ তম প্যারেন্ট নেওয়া মানে $2^4 = 16$ তম প্যারেন্টে যাওয়া। যেহেতু আমাদের নোডই আছে মাত্র 10 টি তাই $parent[4+1][i]$ আকারের অ্যারেই যথেষ্ট।

এবার আসা যাক কুয়েরির সময় কী করতে হবে সেই বিষয়ে। প্রথম কাজ হলো x এবং y এর মধ্যে যেটি বেশি গভীরতা বা ডেপ্থ (depth) এ আছে তাকে x এ নাও। এখন x কে y এর গভীরতার অ্যানসেস্ট্রে আনো। আনার উপায় সহজ, তুমি প্রথমে $parent[17][x]$ দেখ, যদি এটি y এর গভীরতার থেকে কম গভীরতায় হয় তাহলে এর পরের প্যারেন্ট অর্থাৎ $parent[16][x]$ চেষ্টা কর, না হলে $x = parent[17][x]$ কর। এভাবে 17 থেকে 0 পর্যন্ত লুপ চালিয়ে এই কাজ করলে x এবং y একই গভীরতায় চলে আসবে এবং এ জন্য তোমার সময় লাগবে $O(\log(n))$. যদি x ও y একই হয়ে যায় তাহলে তো এটিই লীস্ট কমন অ্যানসেস্ট্র। আর যদি নাহয় তাহলে আবার 17 হতে 0 পর্যন্ত লুপ চালাও। ধরা যাক এটি i এর লুপ। এবার দেখ $parent[i][x]$ আর $parent[i][y]$ একই কিনা। যদি একই হয় তাহলে i এর লুপ *continue* কর আর যদি না হয় তাহলে $x = parent[i][x]$ এবং $y = parent[i][y]$ কর। এভাবে 0 পর্যন্ত লুপ চালানোর পর, উত্তর হবে x বা y এর সরাসরি প্যারেন্ট বা $parent[0][x]$. কেন? কারণ $2^0 + 2^1 + \dots + 2^{n-1} = 2^n - 1$. উত্তরটা একটু আবছা হয়ে গেলো কিন্তু আশা করি তোমরা চিন্তা করে বের করে ফেলতে পারবে।

৯.৬ প্রোগ্রামিং সমস্যা

৯.৬.১ অনুশীলনী

খুব সহজ

- UvaLive 6821 ◦ UvaLive 6899 ◦ UvaLive 6934 ◦ UvaLive 6959 ◦ UvaLive 6960* ◦ UvaLive 7012 ◦ UvaLive 7013 ◦ UvaLive 7054

সহজ

- UvaLive 6786 ◦ UvaLive 6832 ◦ UvaLive 6833 ◦ UvaLive 6852 ◦ UvaLive 6854*** ◦ UvaLive 6863 ◦ UvaLive 6895 ◦ UvaLive 6901 ◦ UvaLive 6903 ◦ UvaLive 6907 ◦ UvaLive 6911 ◦ UvaLive 6915 ◦ UvaLive 6917 ◦ UvaLive

6924 :: UvaLive 6935* :: UvaLive 6939 :: UvaLive 6944 :: UvaLive 6947*
:: UvaLive 6963 :: UvaLive 6965 :: UvaLive 6968 :: UvaLive 7049 :: UvaLive
7068** :: UvaLive 7073 :: UvaLive 7077 :: UvaLive 7084 :: UvaLive 7139
:: UvaLive 7146 :: UvaLive 7164 :: UvaLive 7170 :: UvaLive 7171

সামান্য কঠিন

:: UvaLive 6804* :: UvaLive 6810*** :: UvaLive 6849* :: UvaLive 6864
:: UvaLive 6889* :: UvaLive 6902** :: UvaLive 6922** :: UvaLive 6926
:: UvaLive 6943** :: UvaLive 6945 :: UvaLive 6958 :: UvaLive 7019 :: UvaLive
7059 :: UvaLive 7064

কঠিন

:: UvaLive 6842*** :: UvaLive 6928**

অধ্যায় ১০

জ্যামিতি (Geometry) এবং কম্পিউটেশনাল জ্যামিতি (Computational Geometry)

Geometry এর মানে হলো জ্যামিতি। কম্পিউটেশনাল জ্যামিতি (Computational Geometry) হলো জ্যামিতি বিষয়ক অ্যালগরিদমের পাঠ। আমরা এই চ্যাপ্টারে এই দুই বিষয় নিয়ে দেখব।

১০.১ মৌলিক জ্যামিতি ও ত্রিকোণমিতি

শুরু সাধারণ জ্যামিতির জ্ঞান আমাদের প্রায়ই সমস্যা সমাধান করতে গিয়ে দরকার হয়। যেমন ত্রিভুজের ক্ষেত্রফলের সূত্র, বর্গক্ষেত্রের ক্ষেত্রফল, আয়তক্ষেত্রের ক্ষেত্রফল, গোলকের আয়তন ইত্যাদি নানা সূত্র আমাদের প্রায়ই কাজে লাগে। যদি এসব সূত্র তোমাদের মনে না থাকে তাহলে এখনই ছোট ক্লাসের বই দেখে নাও। এছাড়াও উচ্চ মাধ্যমিকে পড়ে আসা ত্রিকোণমিতির সূত্র আমাদের প্রায়ই কাজে লাগে। আমরা এখানে এদের মধ্যে গুটিকয়েক সূত্র দেখব।

মনে কর আমাদের কাছে একটি ত্রিভুজের তিন বাহু দেওয়া আছে এবং তারা হলো: a, b, c . তাহলে সেই ত্রিভুজের ক্ষেত্রফল কত? এর সূত্র হলো: $\sqrt{s(s - a)(s - b)(s - c)}$ যেখানে $s = (a + b + c)/2$. একে হিরণ্যের সূত্র বলা হয়ে থাকে। অনেক সময় আমাদের বলে যে "যদি কোনো উভয় সম্ভব না হয় তাহলে impossible প্রিন্ট কর"। জ্যামিতির সমস্যার ক্ষেত্রে কোথাও এরকম কথা বললে তোমরা যা করবে তা হলো সুত্রের দিকে তাকাবে আর চিন্তা করবে এই সূত্র কখন অসম্ভব হবে। যেমন উপরের সুত্রটি অসম্ভব হবে যদি sqrt এর ভেতরের মান ঋণাত্মক হয়। যেমন $a = 10, b = 1, c = 1$ হলে উপরের সুত্রে sqrt এর ভেতরের সংখ্যা ঋণাত্মক হবে। এর মানে হলো এই তিনটি মান দিয়ে কোনো ত্রিভুজ বানানো সম্ভব না। এভাবে অন্যান্য সমস্যার ক্ষেত্রেও এই

কৌশলটি তোমাদের কাজে লাগতে পারে। একইভাবে যদি উপরের সূত্র শূন্য মান দেয় এর মানে হবে ত্রিভুজের তিনটি বিন্দুই একই রেখায় আছে।

এখন ধরা যাক আমাদের দরকার হলো যে a এর বিপরীত দিকের কোণ A বের করতে হবে। কীভাবে? তোমাদের অনেকের হয়তো ত্রিকোণমিতিতে পড়া ত্রিভুজের ক্ষেত্রফলের সূত্র মনে আছে: $\Delta = \frac{1}{2}bc \sin A$ যেখানে Δ হলো ত্রিভুজের ক্ষেত্রফল। আমরা যদি উপরের হি঱ণের সূত্র ব্যবহার করে Δ এর মান বের করি তাহলে \sin^{-1} করে A এর মান খুব সহজেই বের করে ফেলতে পারব। আসলেই কি পারব? খেয়াল কর, $\sin X = \sin(180^\circ - X)$ সূতরাং আমরা যদি \sin^{-1} করি আমরা বুঝব না যে এটা কি X নাকি $180^\circ - X$. হ্যাঁ আমরা জানি $\sin X = \sin(360^\circ + X)$, কিন্তু যেহেতু আমাদের ত্রিভুজের কোনো কোণ 180° অপেক্ষা বড় নয় সেহেতু সেটি নিয়ে মাথা ব্যাথাও করা যাবে না (হয়তো যাবে কিন্তু এর সঙ্গে আরও কিছু যাচাই করতে হবে সেক্ষেত্রে)। আমরা যদি \cos^{-1} ব্যবহার করতে পারি তাহলে কিন্তু এই সমস্যা হবে না, কারণ 0° হতে 180° প্রত্যেক ডিগ্রীর জন্য \cos এর মান আলাদা। সূতরাং আমাদের এমন একটি ত্রিকোণমিতির সূত্র ব্যবহার করতে হবে যেন তাতে \cos থাকে। এবং সেই সূত্র হলো: $a^2 = b^2 + c^2 - 2bc \cos A$.

তবে এই যে sqrt বা \sin^{-1} বা \cos^{-1} , এসব ফাংশন কল করার আগে একটু সাবধান হতে হবে। আগেই বলেছি `double` বা `float` ডেটা টাইপ কিন্তু তোমার পুরো মান রাখতে পারে না, অনেক সময় দেখা যায় 2 এর জায়গায় 1.99999... আছে আবার দেখা যায় 2.0001 আছে। এসবের জন্য sqrt বা \sin^{-1} বা \cos^{-1} ফাংশন কল করার আগে একটু সাবধান হতে হয়। মনে কর sqrt এর ভেতরের মান আসার কথা 0 কিন্তু এল -0.00001 বা \sin^{-1} এর ভেতরে মান আসার কথা 1 এল 1.00001 এসব ক্ষেত্রে তুমি যদি ওই ফাংশন ডেকে বসো তাহলে কিন্তু `runtime error` হয়ে যাবে। সেজন্য আমি যেটা করি তা হলো নিজের sqrt বা \sin^{-1} বা \cos^{-1} ফাংশন লিখি। যেখানে দেখা যায় $\text{sqrt}(\text{abs}())$ কে কল করি বা \sin^{-1} বা \cos^{-1} এর ভেতরে যাচাই করে দেখি যে যেই মান পেলাম তা 1 এর থেকে বড় হলে কত রিটার্ন করব আর -1 এর থেকে ছোট হলে কত রিটার্ন করব।

আর দুটি ফ্রোটিং পয়েন্ট সংখ্যা (floating point number) সমান কিনা এটি যাচাই করার জন্য $a == b$ করলে কিন্তু হয় না। যা করতে হবে তা হলো: $\text{abs}(a - b) <= \text{eps}$ কিনা যেখানে $\text{eps} = 10^{-7}$ এর মতো কোনো ছোট সংখ্যা। সমস্যা ভেদে দেখা যায় eps এর মান পরিবর্তন করতে হবে। একইভাবে তুমি যদি যাচাই করতে চাও যে $a \leq b$ কিনা তাহলে যাচাই করবে: $a <= b + \text{eps}$.

আমরা মাত্র দুটি জিনিস দেখলাম: 1- কীভাবে ত্রিভুজের তিন বাহু দেওয়া থাকলে তার ক্ষেত্রফল বের করতে হয়, 2- কীভাবে ত্রিভুজের কোনো কোণ বের করতে হয়। সেই সঙ্গে জ্যামিতি সমস্যার ক্ষেত্রে দরকারি কিছু কৌশল। প্রতিযোগিতায় জ্যামিতির অনেক ছোট ছোট সমস্যা আসে যা তোমরা আসলে তোমাদের সাধারণ জ্ঞান প্রয়োগ করলেই সমাধান করতে পারবে অথবা খুব জোড় তোমাদের মাধ্যমিক বা উচ্চ মাধ্যমিকের বই খুলে দেখতে হবে। আর ইন্টারনেট তো আছেই। এরকম কিছু সমস্যা আমি এখানে লিস্ট আকারে দিচ্ছি:

: ত্রিভুজের তিন বাহু দেওয়া আছে, ত্রিভুজের ...

- পরিবৃত্তের ব্যাসার্ধ বের কর (radius of circumcircle)
- অন্তঃবৃত্তের ব্যাসার্ধ বের কর (radius of inner circle)
- তিনটি মধ্যমা (median) এর দৈর্ঘ্য বের কর
- তিনটি উচ্চতা (height) বের কর
- তিনটি কোণ সরাসরি বের না করে তুমি কি বলতে পারবে কোনো ত্রিভুজ সূক্ষ্মকোণী (acute), সমকোণী (right) বা স্ফূলকোণী (obtuse) কিনা? এটি প্রায়ই দরকার হয়। কারণ আমরা যতটা সম্ভব ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশ কর করার চেষ্টা করি। যদি ত্রিভুজের তিন বাহু পূর্ণ সংখ্যায় দেওয়া থাকে তাহলে আমরা ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশ না করে বলে দিতে পারি ত্রিভুজটি সূক্ষ্মকোণী/সমকোণী/স্ফূলকোণী কিনা। বা আসলে কোনো একটি কোণ সূক্ষ্মকোণ/সমকোণ/স্ফূলকোণ কিনা। উপায় হলো, আমরা জানি পিথাগোরাসের সূত্র হলো $a^2 = b^2 + c^2$ যেখানে a হলো সমকোণের বিপরীত বাহু বা অতিভুজ। এখন তুমি যদি = এর পরিবর্তে < বা > বসাও তাহলেই তুমি বলতে পারবে যে তারা সূক্ষ্মকোণ/স্ফূলকোণ কিনা।
- ধর একটি r ব্যাসার্ধের বৃত্ত আছে। এখন এর কেন্দ্র হতে d দূরত্ব দূরে একটি সরলরেখা টেনে বৃত্তের একটি অংশ কেটে ফেলে দেওয়া হলো। বলতে হবে বাকি অংশের ক্ষেত্রফল কত? বাকি অংশের পরিধিই বা কত? এটি যদি বৃত্ত না হয়ে একটি গোলক (sphere) হতো তাহলে সুগুলো কেমন হতো?

১০.২ স্থানাঙ্কভিত্তিক জ্যামিতি (Coordinate Geometry) এবং ভেক্টর (Vector)

আমরা আগের সেকশনে জ্যামিতির খুবই মৌলিক কিছু হিসাবনিকাশ দেখলাম। এখন আমরা কিছু স্থানাঙ্কভিত্তিক জ্যামিতি (coordinate geometry) আর ভেক্টর (vector) দেখব। আমরা হয়তো এক ফাঁকে জটিল সংখ্যা (complex number) ব্যবহার করে কীভাবে ভেক্টর এবং স্থানাঙ্কভিত্তিক জ্যামিতির কিছু কিছু হিসাবনিকাশ আরও সহজে করা যায় তাও দেখব। তোমরা যদি স্থানাঙ্কভিত্তিক জ্যামিতি আর ভেক্টরের সঙ্গে একদমই পরিচিত না হও তাহলে ভালো হবে যদি তোমরা উচ্চ মাধ্যমিকের বই দেখে স্থানাঙ্কভিত্তিক জ্যামিতি আর ভেক্টর নিয়ে একটু জ্ঞান নিয়ে ফেল। আমরা এখানে প্রোগ্রামিং প্রতিযোগিতার দিক থেকে এই দুটি জিনিস দেখবো।

দ্বিমাত্রিক বা $2 - \text{dimensional}$ ($2D$) স্থানাঙ্কভিত্তিক জ্যামিতিতে আমরা একটি বিন্দুকে (x, y) দিয়ে প্রকাশ করি। অর্থাৎ আমরা একটি স্ট্রাকচার (structure) ব্যবহার করে খুব সহজেই বিন্দু (point) বানিয়ে ফেলতে পারি। একইভাবে সেই একই স্ট্রাকচার ব্যবহার করে আমরা $2D$ ভেক্টর এর কাজও করে ফেলতে পারি। সুতরাং দেখা যায় যে, point structure এর জন্য যোগ

(addition), বিয়োগ (subtraction), ক্ষেলার / ডট গুণন (scalar/dot product), ক্রস গুণন (cross product) ইত্যাদি ফাংশন বা অপারেটর ওভারলোডিং (operator overloading) ক্ষেত্রে প্রয়োজন হয়।

রেখা বা Line আবার বিভিন্ন রকম হতে পারে। আমরা অনেক ছোটবেলাতেই পঢ়েছি: (সরল) রেখা, রেখাংশ আর রশ্মি। রেখা হলো যার দুই দিকেই কোনো সীমা নেই, রেখাংশ হলো যার দুই দিকেই সীমা আছে আর রশ্মি হলো যার এক দিকে সীমা। একটি রেখাকে আমরা বিভিন্নভাবে উপস্থাপন করতে পারি। এদের একটি হলো $y = mx + c$ । অর্থাৎ শুধু m আর c দিয়ে একটি রেখা উপস্থাপন করা হবে যেখানে রেখার সূত্র হবে $y = mx + c$. এই উপস্থাপনের সমস্যা হলো $y - axis$ বা Y -অক্ষের সমান্তরাল কোনো রেখাকে আমরা উপস্থাপন করতে পারিব না। কারণ Y -অক্ষের সমান্তরাল রেখার সূত্র হবে $x = 5$ এরকম ধরনের। কোনো m বা c এর মানের জন্যই কিন্তু তুমি $y = mx + c$ কে $x = 5$ এরকম রূপে আনতে পারবে না। এছাড়াও এভাবে উপস্থাপনের জন্য বেশির ভাগ সময়ই আমাদের ফ্লোটিং পয়েন্ট সংখ্যার দরকার হয়। কারণ দুটি বিন্দু (x_1, y_1) এবং (x_2, y_2) দেওয়া থাকলে এদের ভেতর দিয়ে যাওয়া রেখার $m = \frac{y_1 - y_2}{x_1 - x_2}$. c বের করার জন্য আরও একটু জটিল হিসাবনিকাশ করতে হয় (যেহেতু এই রেখাটি x_1, y_1 দিয়ে যায় তাই তুমি m এর মান আর x_1, y_1 এই সূত্রে বসিয়ে দাও তাহলেই c এর মান পেয়ে যাবে)। এর থেকে ভালো উপায় আমার মনে হয়: $ax + by = c$. কারণ এভাবে কোনো ধরনের রেখাকে উপস্থাপন করা সমস্যা না এবং আগের মতো ফ্লোটিং পয়েন্ট সংখ্যা ব্যবহার না করলেও চলে। এসব সমীকরণের সুবিধা হলো তুমি তৃতীয় কোনো বিন্দু নিয়ে খুব সহজেই যাচাই করে দেখতে পারবে যে সেটি তোমাদের রেখার উপর আছে কিনা বা রেখার কোন দিকে আছে। মনে কর তোমাকে দুটি বিন্দু আর একটি রেখার সূত্র দিয়ে বলা হলো যে এই দুটি বিন্দু কি রেখার একই দিকে আছে কিনা। সেক্ষেত্রে তুমি এই দুটি বিন্দুর x, y রেখার সূত্রে বসিয়ে দেখো যে দুক্ষেত্রেই ধনাত্মক আসে কিনা বা দুক্ষেত্রেই ঋণাত্মক আসে কিনা। যদি কারো ক্ষেত্রে 0 আসে এর মানে তো জানোই, সে ঐ রেখার উপরে আছে।

রেখাকে উপস্থাপন করার আরেকটি উপায় হলো প্যারামেট্রিক রূপ (parametric form) এবং এটি বেশ কাজের। ধর তোমাকে দুটি বিন্দু $A(x_a, y_a)$ এবং $B(x_b, y_b)$ দেওয়া আছে। তুমি এদের ভেতর দিয়ে যায় এরকম একটি সরলরেখার প্যারামেট্রিক রূপে উপস্থাপন (parametric representation) চাও। এটি হবে: $A + t(B - A)$. এখানে $B - A$ কে একটি ভেষ্টরের মতো তাত্ত্বিক হতে পার। একটি উদাহরণ দেওয়া যাক, মনে কর $A(1, 1)$ আর $B(4, 6)$ এর ভেতর দিয়ে যায় এরকম একটি রেখার প্যারামেট্রিক রূপ বের করতে চাই। এটি হবে $(1, 1) + t[(4, 6) - (1, 1)]$ বা $(1, 1) + t(3, 5)$ বা $(1 + 3t, 1 + 5t)$. একটু অঙ্গুত্ব তাই নাই এই অনুচ্ছেদের মাঝে অংশটুকু পড় তাহলেই বুঝবে এর কাহিনী কী। এখন এই উপস্থাপনের কিছু সুবিধা দেখা যাব। এখানে তোমার ফ্লোটিং পয়েন্ট সংখ্যার দরকার হয় না- যদি A এবং B দুটি পূর্ণ সংখ্যার দ্বন্দ্বওয়ালা বিন্দু হয়। আবার খেয়াল কর t এর মান যদি $[0, 1]$ এ সীমাবদ্ধ হয় তাহলে এটি একটি রেখাংশ উপস্থাপন করে। যদি $[0, \infty]$ এ সীমাবদ্ধ হয় তাহলে এটি হবে রশ্মি, আর যদি পুরো B এর দিকে তিলে তিলে যাবে। $t = 0$ এর মানে তুমি A তেই থাকবে (সূত্রে $t = 0$ বসিয়েই দেখো না!), $t = 1$ এর মানে তুমি B তে, এর মধ্যের মান মানে তুমি A আর B এর মধ্যে আছ। এমনকি

তুমি $t = 1/2$ বা $t = 1/3$ বসিয়ে ঠিক মধ্যের বা ঠক একত্ত্বাংশ দূরের বিন্দুও বের করে ফেলতে পারবে। আবার যদি $t < 0$ হয় এর মানে হবে তুমি উল্টো দিকে যাচ্ছ। উপরের উদাহরণে আমরা এখন একে একে মান বসাই, $t = 0$ বসালে হয় (1, 1) যা কিনা A. $t = 1$ বসালে হয় (4, 6) যা কিনা B. যদি $t = 0.5$ বসাও তাহলে (2.5, 3.5) যা কিনা A আর B এর মধ্যের বিন্দু। একই ভাবে $t = 2, -1$ ইত্যাদি বসিয়ে দেখো যে আমার উপরের কথাটা সঠিক কিনা। এবার মনে কর কোনো একটি বিন্দু $C(x_c, y_c)$ দিয়ে যদি বলা হয় এটি এই রেখার উপরে আছে কিনা তাহলেও এটি বের করা বেশ সহজ। তোমাকে $A + t(B - A) = C$ এই সমীকরণ সমাধান করতে হবে। এখানে x এর জন্য একটি এবং y এর জন্য আরেকটি সমীকরণ পাবে। এর এখানে ভ্যারিয়েবল মাত্র একটি t . তোমাকে দেখতে হবে এই দুই সমীকরণেই t এর সমাধান একই কিনা। এটা তো তুমি কোনো ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশ না করেই বলে ফেলতে পারবে তাই না? তবে কোনো দুটি বিন্দু দিয়ে যদি বলে এটি AB রেখার একই দিকে আছে কিনা তাহলে মনে হয় এভাবে সন্দেব না। সেক্ষেত্রে আমি যা করি তা হলো স্থানাঙ্কভিত্তিক জ্যামিতির ত্রিভুজের ক্ষেত্রফল বের করার সুত্র ব্যবহার করি:

$$\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix}$$

এই নির্ণায়ক বা ডিটারমিন্যান্ট (determinant) এর সুত্র ব্যবহার করে আমরা যেমন ABC ত্রিভুজের ক্ষেত্রফল বের করতে পারি ($\frac{1}{2}$ দিয়ে গুণ আর পরম মান অর্থাৎ absolute value নিতে ভুলে যেও না) ঠিক তেমনি A, B, C একই সরলরেখায় কিনা (নির্ণায়কের মান শূন্য হলে) বা ABC কি ঘড়ির কাটার দিকে বা clockwise(cw) আছে নাকি ঘড়ির কাটার বিপরীত দিকে বা anti-clockwise বা counter-clockwise(ccw) আছে তাও বের করা যায়। A, B, C যদি ccw এ থাকে তাহলে নির্ণায়কের মান ধনাত্মক আর cw এ থাকলে ঋণাত্মক হয়। তোমার যদি মনে না থাকে cw হতে গেলে ধনাত্মক না ঋণাত্মক হতে হয় তাহলে তুমি (0, 0), (0, 1) আর (1, 0) বসিয়ে নির্ণায়কের মান বের করে দেখ। যেহেতু বেশির ভাগ মানই শূন্য সেহেতু তোমাকে খুব কম হিসাব করতে হবে।

এখন চিন্তা করে দেখ তোমাকে যদি একটি বৃত্ত (কেন্দ্র ও ব্যাসার্ধ) আর সরলরেখা(দুটি বিন্দু) দিয়ে যদি বলে তাদের ছেদ বিন্দু বের কর কীভাবে করবে? সহজ উপায় হলো প্রথমে তুমি পুরো স্থানাঙ্ককে বৃত্তের কেন্দ্রের দৃষ্টিকোণ থেকে চিন্তা কর অর্থাৎ স্থানাঙ্ক স্থানান্তর বা স্থানাঙ্ক ট্রান্সলেশন (coordinate translation)¹ করে নাও। তাহলে বৃত্তের সমীকরণ আসবে $x^2 + y^2 = r^2$ (কারণ বৃত্তের কেন্দ্র এখন (0, 0)). এবার সরলরেখার প্যারামেট্রিক সমীকরণ (parametric equation) বের কর আর তা বৃত্তের সমীকরণে বসিয়ে t এর মান সমাধান কর। তুমি t দিয়ে একটি দ্বিঘাত সমীকরণ (quadratic equation) পাবে। যদি দেখ এই সমীকরণের সমাধান নেই অর্থাৎ $b^2 - 4ac < 0$ এর মানে সরলরেখা বৃত্তকে ছেদ করে না। যদি $b^2 - 4ac = 0$ হয় অর্থাৎ t এর একটি

¹স্থানাঙ্ক ট্রান্সলেশন (Coordinate translation) করার মানে হলো তুমি তোমার মূলবিন্দু বা অরিজিন (origin) কে স্থানান্তর করে (x_1, y_1) বিন্দুতে কল্পনা করবে। অর্থাৎ সব স্থানাঙ্ক থেকে (x_1, y_1) বিয়োগ করবে। এরপর যা হিসাবনিকাশ করার তা করা শেষে আবার (x_1, y_1) যোগ করে স্থানাঙ্কগুলোকে আগের মতো করতে হবে।

মাত্র সমাধান থাকে তাহলে তারা পরস্পরকে স্পর্শ করে (tangent) আর যদি দুটি সমাধান পাওয়া যায় এর মানে তারা দুটি জায়গায় ছেদ করে। যদি এটি সরলরেখা না হয়ে রেখাংশ হতো তাহলেও কিন্তু তুমি। এর মান দেখে বলে দিতে পারতে যে ছেদ বিন্দুটি রেখাংশের উপর আছে না বাইরে। যদি তোমার ছেদ বিন্দুই লাগে তাহলে স্থানাঙ্ক আবারও উল্টো ট্রান্সলেশন করে নিতে ভুলে যেও না। একটু চিন্তা করলে দেখবে এটি শুধু $2D$ তে না $3D$ তেও সমানভাবে কাজ করে। তোমাকে যদি $3D$ তে দুটি বিন্দু দিয়ে যদি বলে এই দুটি বিন্দু দিয়ে যায় এরকম একটি সরলরেখার সঙ্গে একটি গোলক (sphere) এর ছেদ বিন্দু বের করতে তাহলেও কিন্তু তুমি একই পদ্ধতি অনুসরণ করতে পারতে।

বৃত্তে যখন চলেই আসলাম তখন বৃত্ত ও বৃত্তের ছেদবিন্দু কীভাবে বের করে এটি ও দেখা যাক। বৃত্তের সাধারণ সমীকরণ হলো $(x-a)^2 + (y-b)^2 = r^2$ ধরনের। অর্থাৎ বৃত্তের উপরের যেকোনো (x, y) বিন্দু যদি তুমি এই সমীকরণে বসাও তাহলে দুই পক্ষ সমান হবে। এখন তুমি যদি বৃত্তের দুটি সমীকরণকে একটি থেকে আরেকটি বিয়োগ দাও তাহলে একটি সরলরেখার সমীকরণ পাবে কারণ x^2 ও y^2 এর টার্ম কাটাকাটি যায়। এভাবে আমরা যেই সরলরেখার সমীকরণ পেলাম সেটি কিসের? ধর E_1 আর E_2 হলো দুটি বৃত্তের সমীকরণ আর $E_1 - E_2$ যদি আরেকটি সমীকরণ হয় তাহলে যেসব সমাধান E_1 ও E_2 দুটিকেই সমাধান করে তারা $E_1 - E_2$ কেও সমাধান করবে। অর্থাৎ বৃত্তের ছেদবিন্দু দিয়ে যায় এরকম একটি সরলরেখার সমীকরণ হলো আমাদের বিয়োগ করে পাওয়া সমীকরণ। যেহেতু বৃত্তের সমীকরণ জান আর ছেদবিন্দু দিয়ে যাওয়া সরলরেখার সমীকরণও জান তাহলে এখন তুমি কিছু সমীকরণ সমাধান করলেই ছেদবিন্দুগুলো পেয়ে যাবে। তবে আগের পদ্ধতিতে আর করতে পারবে না কারণে এখানে সরলরেখার সমীকরণ হলো $ax + by = c$ ধরনের। এই পদ্ধতি অবলম্বন করার আগে তোমরা দেখে নিবে যে বৃত্ত দুটি আসলেই ছেদ করে কিনা, না হলে এই হিসাবনিকাশ করার সময় বিপদে পড়তে পার। বিপদ মানে, শেষ পর্যায়ে এসে তোমাকে যখন দিয়া সমীকরণ সমাধান করতে হবে তখন তোমাকে চিন্তা করতে হবে এই সমীকরণের আদৌ সমাধান আছে কিনা ইত্যাদি। এসব বাড়তি চিন্তা থেকে মুক্ত হওয়ার জন্যই আগেই দেখে নিতে হবে বৃত্তের ছেদ করে কিনা। ছেদবিন্দু বের করার চেয়ে বৃত্ত দুটি ছেদ করে কিনা সেটা বের করা সহজ। যদি দূরত্বের দূরত্ব d হয় তাহলে দুটি বৃত্ত ছেদ করে যদি $\text{abs}(r_1 - r_2) \leq d \leq r_1 + r_2$ হয়। আরও একটি কথা, এখানে খেয়াল কর d এর মান বের করতে আমাদের একটি বর্গমূল বের করতে হয়। কিন্তু আমরা চাইলে সবগুলো মানকে বর্গ করে বর্গমূল ব্যবহার না করেও বৃত্ত দুটি ছেদ করে কিনা অবলে ফেলতে পারি। এটা খুব দরকারি কারণ আমাদের উচিত যতটুকু পারা যায় double ডেটা টাইপে হিসাবনিকাশ কে এড়িয়ে চলা।

এতক্ষণ মূলত স্থানাঙ্কভিত্তিক জ্যামিতি আর প্যারামেট্রিক সমীকরণ নিয়ে কথা বললাম। প্যারামেট্রিক সমীকরণের কথা বলতে গিয়ে একটু ভেষ্টরের কথাও বলেছি। এবার ভেষ্টরকে আরও একটু ভালোমতো দেখা যাক। মনে কর একটি রেখা আর একটি বিন্দু দিয়ে বলা হলো এই বিন্দু দিয়ে ওই সরলরেখার উপর লম্ব টানতে হবে। লম্ব দূরত্ব কিন্তু বের করা বেশ সহজ কিন্তু লম্বের পাদবিন্দু বের করা একটু কঠিন। একটি উপায় হলো রেখার উপর একটি বিন্দু A নাও এর পর রেখার প্যারামেট্রিক সমীকরণ বের করো। ধরে নাও t তে পাদবিন্দু। তাহলে A , পাদবিন্দু আর দিওয়া বিন্দু এদের নিয়ে একটি পিথাগোরাসের সূত্র লিখে ফেল তাহলেই t এর মান সমাধান করে

তুমি পাদবিন্দু B বের করে ফেলতে পারবে। তবে আরেকটি উপায় হলো ধরে নাও পাদবিন্দু B , তাহলে B হতে A এবং প্রদত্ত বিন্দুর ভেক্টরদের ডট গুণন নিলে তা শূন্য হওয়ার কথা। এই সমীকরণকে সমাধান করলেই তুমি B এর স্থানাঙ্ক পেয়ে যাবে। একইভাবে তোমাকে যদি বলে একটি রেখার একটি বিন্দুতে লম্বের সমীকরণ বের করতে হবে আশা করি তুমি তা বের করতে পারবে।

এখন মনে কর তোমাকে দুটি বিন্দু দিয়ে বলা হলো এই দুটি বিন্দু যদি কোনো সমবাহ্য ত্রিভুজের শীর্ষবিন্দু বা ভার্টেক্স (vertex) হয় তাহলে অপর বিন্দু বের কর। তাহলে কীভাবে করবে? তুমি চাইলে ধরে নিতে পার যে অপর বিন্দু (x, y) এবং এর পর এটি হতে উপর দুটি বিন্দুর দূরত্ব "এত" হবে এই বলে দুটি সমীকরণ দাঁড় করিয়ে তাদের সমাধান করতে পার। তবে এটি আমার কাছে একটু পেঁচানো লাগে। যারা স্থানাঙ্ককে রোটেশন (rotation) করাতে পার না তারা এভাবেই করতে পার শেষ অবলম্বন হিসেবে। কিন্তু যারা রোটেশন করাতে পার তারা হয়তো আরও সহজে করে ফেলতে পারবে। মূল বিন্দুর সাপেক্ষে স্থানাঙ্ক θ কোণে ঘোরানোর উপায় হলো $(x, y) \mapsto (x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)$. এই কথা বলার পর তোমাদের উচিত আমাকে বকাবকি করা। এই হড়বড়ে সুত্র কীভাবে মনে রাখা সন্তুষ্ট! অনেকে একে ম্যাট্রিক্স রূপে মনে রাখে:

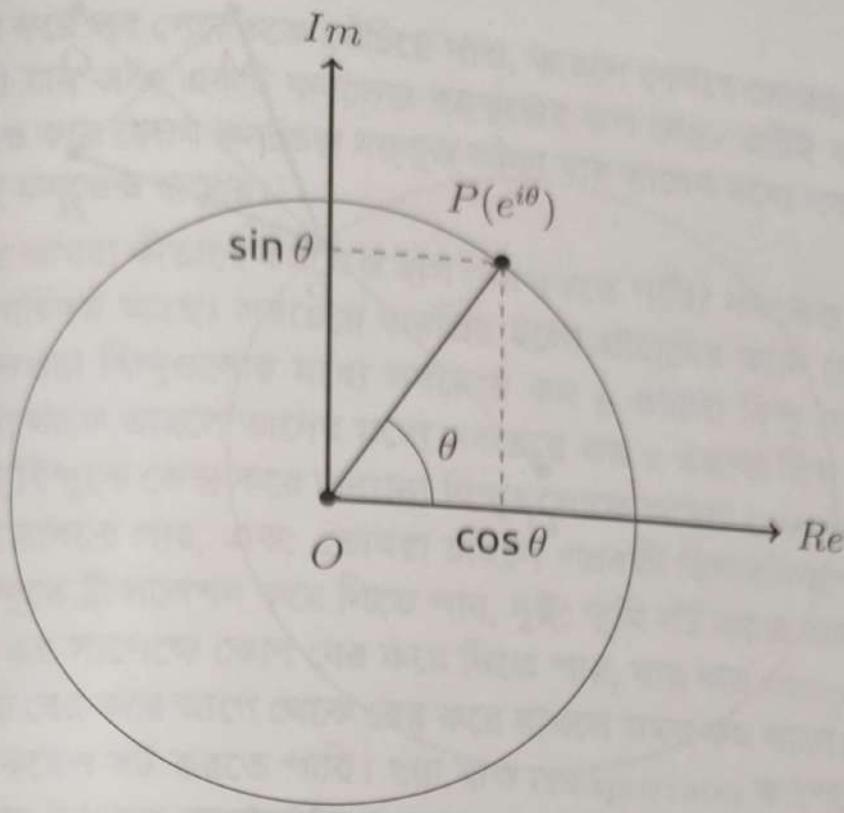
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \quad (10.1)$$

এটি বলার অপেক্ষা রাখে না যে, এটিও কোনো অংশে কম কঠিন নয়! তাহলে কি কোনো সহজ উপায় নেই? আছে, তবে এজন্য তোমাদের জটিল সংখ্যা সম্পর্কে মৌলিক জ্ঞান দরকার। আমি জানি না এখন উচ্চ মাধ্যমিকয়ের সিলেবাসে জটিল সংখ্যা আছে কিনা বা থাকলেও অয়লারের সুন্দর সুত্রটি দেখানো হয় কিনা। মনে হয় না আমাদের সময়ও অয়লারের সুত্র দেখানো হতো। অয়লারের সুত্রটি হলো $e^{i\theta} = \cos \theta + i \sin \theta$. অনেকে মনে করে $\theta = \pi$ বসালে পৃথিবীর সবচেয়ে সুন্দর সুত্র $e^{i\pi} + 1 = 0$ পাওয়া যায় যেখানে বিশ্বের সব থেকে গুরুত্বপূর্ণ ত্রুটি গুলো আছে: $0, 1, i, \pi, e$. যাই হোক, চিত্র 10.1 এ 1 একক ব্যাসার্ধের একটি বৃত্ত নেওয়া হয়েছে এবং এই বৃত্তের উপর X -অক্ষ হতে θ কোণ দূরত্বে একটি বিন্দু নেওয়া হয়েছে ধরা যাক এর নাম P . এই বিন্দুটি অয়লারের উপস্থাপন অনুসারে $e^{i\theta}$. যদি বৃত্তের ব্যাসার্ধ r হতো তাহলে এটি হতো $re^{i\theta}$. খেয়াল করলে দেখবে P বিন্দুর স্থানাঙ্ক হলো $(\cos \theta, \sin \theta)$ অর্থাৎ জটিল সংখ্যায় $a + ib$ রূপে যদি আমরা লিখি তাহলে দাঁড়াবে $\cos \theta + i \sin \theta$.

এত কিছু বলার কারণ হলো, তোমরা যদি কোনো একটি ভেক্টরকে $e^{i\theta}$ দিয়ে গুণ কর তাহলে সেই ভেক্টর মূলবিন্দু সাপেক্ষে CCW দিকে θ কোণে ঘুরে যাবে। ধরা যাক আমাদের ভেক্টরটি হলো OA যেখানে A এর স্থানাঙ্ক হলো (x, y) . একে জটিল সংখ্যায় লিখলে পাব $x + iy$. এখন একে আমরা যদি $e^{i\theta}$ দিয়ে গুণ করি তাহলে আমরা পাব:

$$\begin{aligned} e^{i\theta} \times (x + iy) &= (\cos \theta + i \sin \theta)(x + iy) \\ &= (x \cos \theta - y \sin \theta) + i(x \sin \theta + y \cos \theta) \end{aligned}$$

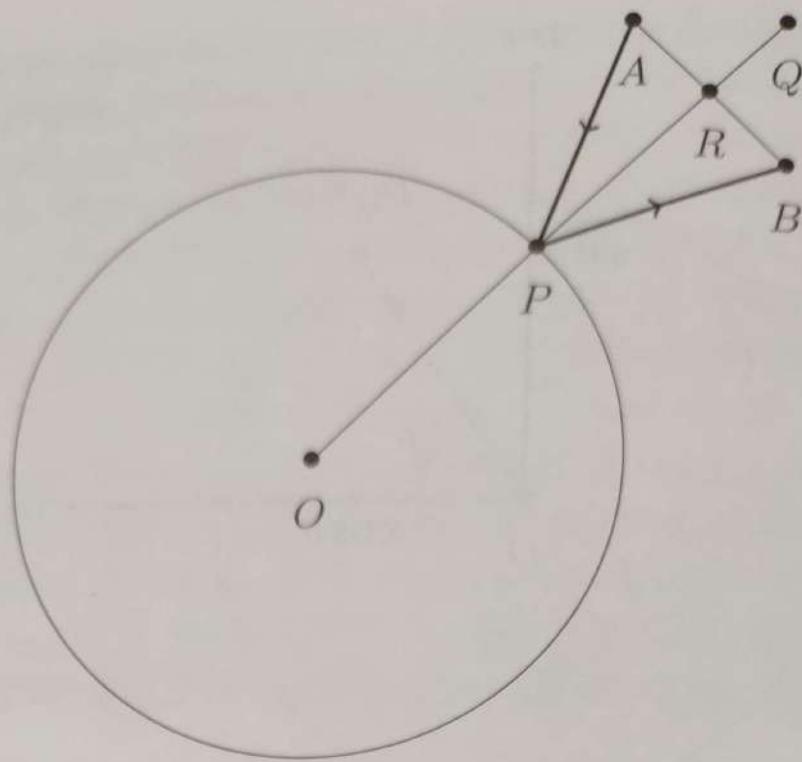
অর্থাৎ গুণ করার পর স্থানাঙ্ক দাঁড়ায় $(x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)$.



নকশা ১০.১: জটিল সংখ্যার অয়লারের উপস্থাপন

শেষ করব ভেষ্টির কত শক্তিশালী হতে পারে তার একটি উদাহরণ দিয়ে। ইচ্ছা করেই আমি এই সমস্যার 2D ভ্যারিয়েশন (variantion) টি উদাহরণ হিসেবে না নিয়ে 3D ভ্যারিয়েশনটি নেবো, উদ্দেশ্য তোমাদের দেখানো যে মাত্রা বা ডাইমেনশন (dimension) বাড়লেও ভেষ্টিরের হিসাবনিকাশের জটিলতা ততটা পরিবর্তন হয় না। আমাদের সমস্যা হল, O কেন্দ্রবিশিষ্ট, ব্যাসার্ধের একটি গোলক আছে। গোলকের উপর কোনো রশ্মি পড়লে তা প্রতিফলিত হয়। A হতে AB নির্গত ঘৰা ধৰা যাক 1 সেকেন্ডে 1 একক দূৰত্ব যায়। বলতে হবে t সময় পৰে A হতে AB এর দিকে নির্গত রশ্মি কোথায় গিয়ে পৌঁছায়। যদি AB রশ্মি গোলককে ছেদ না করে (প্যারামেট্রিক সমীকৰণ ব্যবহার করে ছেদ করে কিনা তা তো বের করতে পারবেই) তাহলে তো এটি বের করা ব্যাপারই না! কথা হলো ছেদ করলে কী হবে। আমরা প্রথমে প্যারামেট্রিক সমীকৰণের সাহায্যে ছেদ বিন্দু বের কৰি। ধৰা যাক ছেদ বিন্দু হলো P . এখন আমাদের বের করতে হবে AP রশ্মি প্রতিফলিত হয়ে কেন দিকে যাবে সেই দিক। এখন OP ভেষ্টির আঁকি, এটি আলোর প্রতিফলনের জন্য লম্ব হিসেবে দাজ করবে। আমরা OP কে Q পর্যন্ত বাড়িয়ে দেই। যদি আমাদের প্রতিফলিত রশ্মি PB হয় (ধৰি $|PB| = |AP|$) তাহলে আমরা বলতে পারি $\angle APQ = \angle BPQ$. তোমাদের সুবিধার জন্য চিত্ৰ ১০.২ এ সব কিছু আঁকা আছে।

এখন A হতে PQ এর উপর AR লম্ব টানি (BR ও তাহলে QR এর উপর লম্ব হবে)। আমরা লিখতে পারি: $\vec{AP} = \vec{AR} + \vec{RP}$. একইভাবে $\vec{BP} = \vec{BR} + \vec{RP}$. তাই আমরা লিখতে পারি $\vec{AP} - \vec{AR} = \vec{BP} - \vec{BR}$. কিন্তু $\vec{AR} = -\vec{BR}$. সুতরাং $\vec{BP} = \vec{AP} - 2\vec{AR}$. আমরা \vec{AP} জানি কারণ A ও P উভয়ই আমাদের জানা। কিন্তু AR জানি না। আমরা চাইলে A হতে PQ এর উপর লম্ব AR এর ভেষ্টির আগের বলে দেওয়া নিয়মে বের করতে পারি কিন্তু তা না করে আমরা আরও একটু



নকশা ১০.২: গোলকে প্রতিফলন

ভেট্টরীয়ভাবে সমাধান করব। আমরা যদি কোনোভাবে \vec{RP} বের করতে পারি তাহলেও কিন্তু হবে। এখন \vec{RP} হলো \vec{AP} এর \vec{QP} বরাবর উপাংশ বা কম্পোনেন্ট (component), যেটি আমরা \vec{QP} বরাবর একক ভেট্টরের সঙ্গে ডট গুণ করলেই পেতে পারি। কিন্তু আমরা কিন্তু Q একটি যেকোনো (arbitrary) বিন্দু ধরেছিলাম তবে \vec{QP} এর দিকে \vec{OP} এর দিকের সম্পূর্ণ বিপরীত দিক আর আমরা O আর P দুটিরই স্থানাঙ্ক জানি। অর্থাৎ আমরা \vec{OP} বরাবর একক ভেট্টর বের করে তাকে negate করব (অর্থাৎ, ঝণাত্মক করে দেবো) তাহলে QR বরাবর একক ভেট্টর বের হয়ে যাবে। এটি দিয়ে \vec{AP} এর ডট গুণ নিলেই \vec{RP} বের হয়ে যাবে। তাহলে তো বাকি \vec{BP} ও পেয়ে যাবা। আমার মনে হয় বাকিটুকু তোমরাই করে ফেলতে পারবে।

১০.৩ কিছু কম্পিউটেশনাল জ্যামিতির অ্যালগরিদম

১০.৩.১ কনভেক্স হাল (Convex Hull)

তোমাকে $2D$ স্থানাঙ্ক ব্যবস্থায় n টি বিন্দু দেওয়া আছে, তোমাকে এর কনভেক্স হাল (convex hull) বের করতে হবে। কনভেক্স হাল কী? প্রদত্ত বিন্দুগুলোকে ঘেরাও করে সবচেয়ে ছোট যে বহুভুজ (convex polygon) বানানো যায় তাই এই সব বিন্দুর কনভেক্স হাল। কনভেক্স কোনো তিনটি বিন্দুকে একই রেখার উপর না দেখতে চাও তাহলে সবসময় 180° এর ছোট নিতে পার। একে এভাবেও চিন্তা করতে পার- প্রদত্ত n বিন্দুতে তুমি পেরেক পুঁতো, এর পর একটি রাবার

যান্তকে প্রসারিত করে সব পেরেককে পেঁচিয়ে দাও, তাহলে দেখবে তোমার রাবার খুব দৃঢ়ভাবে পেরেকগুলো দিয়ে যায় এবং একটি কনভেক্স বহুজের রূপ নেয়। এটিই কনভেক্স হাল। প্রদত্ত বিন্দুসমূহকে ঘেরাও করে যেসব কনভেক্স বহুজ আঁকা যায় তাদের মধ্যে সবচেয়ে ছোট ফ্রেকল এবং পরিসীমা এই কনভেক্স হালের।

এখন প্রশ্ন হলো আমরা কীভাবে কনভেক্স হাল বের করতে পারি? কনভেক্স হাল বের করার জন্য অনেকগুলো অ্যালগরিদম আছে। সবচেয়ে জনপ্রিয় হলো গ্রাহামের স্ক্যান (Graham's scan)। প্রথমে আমাদের দেওয়া বিন্দুগুলোর মধ্যে সবচেয়ে কম y ওয়ালা বিন্দু বের করতে হবে, যদি একম অনেকগুলো থাকে তাহলে তাদের মধ্যে সবচেয়ে কম x ওয়ালা বিন্দু নেব। ধরা যাক এটি হলো O . এখন এই বিন্দুকে কেন্দ্র করে অন্যান্য বিন্দুগুলোকে আমরা CCW এ সর্ট করব। একে দুটি জিনিস খেয়াল রাখতে পার, এক: তোমরা চাইলে পরবর্তী হিসাবনিকাশগুলো সহজে করার জন্য O কে মূল বিন্দুতে ট্রান্সলেশন করে নিতে পার, দুই: তুমি সর্ট করার আগেই atan2 ব্যবহার করে সব বিন্দুর O এর সাপেক্ষে কোণ বের করে নিতে পার, বার বার comparison ফাংশনের ভেতরে এই কোণ না বের করে আগে থেকে বের করে রাখলে সময় কম লাগে। তবে আমরা চাইলে atan2 ব্যবহার না করেও সর্ট করতে পারি। ধরা যাক comparison ফাংশনে A ও B দুটি বিন্দু দিয়ে বলা হলো কোনটি আগে হবে? যদি OAB CCW হয় তাহলে A আগে হবে না হলে পরে। আরেকটা কথা, যদি দুটি বিন্দু O এর সাপেক্ষে একই কোণ তৈরি করে তাহলে যার দূরত্ব O হতে কম সে আগে থাকবে। এখন এই সর্টেড বিন্দুগুলোকে একে একে নিতে হবে আর শর্তসাপেক্ষে একটি স্ট্যাকে রাখতে হবে। যদি স্ট্যাক ফাঁকা হয় তাহলে সরাসরি স্ট্যাকে ঢুকিয়ে দাও আর যদি তানা হয় তবে O , স্ট্যাকের সবচেয়ে উপরে থাকা বিন্দু আর বর্তমান বিন্দু এদের যাচাই করে দেখতে হবে যে এরা কি CW নাকি CCW. যদি CW হয় তাহলে স্ট্যাকের উপরের বিন্দুকে ধরে ফেলে দিতে হবে এবং এবারের স্ট্যাকের উপরের বিন্দুকে নিয়ে আবার একইভাবে যাচাই করতে হবে। এখন এক সময় CCW হয়ে যাবে তখন বর্তমান বিন্দুকে স্ট্যাকে ঢুকিয়ে দিতে হবে। এভাবে একে একে সব বিন্দু যাচাই করা শেষ হয়ে গেলে স্ট্যাকে কনভেক্স হাল পাওয়া যাবে।

যদিও গ্রাহামের স্ক্যান অ্যালগরিদম বেশ সহজ এবং জনপ্রিয় কিন্তু এটি সংখ্যাগত হিসাবনিকাশের দিক থেকে অতটা সুস্থিত না। অর্থাৎ তোমার স্থানাঙ্ক যদি ফ্লোটিং পয়েন্ট সংখ্যায় থাকে তাহলে মাঝে মধ্যে ঝামেলা পাকাতে পারে ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশের অস্থিতিশীলতার জন্য। সেজন্য যেই অ্যালগরিদম ব্যবহার করা উচিত তা হলো মনোটোন চেইন কনভেক্স হাল অ্যালগরিদম (Monotone chain convex hull algorithm)। এটিও বেশ সহজ। প্রথমে আমাদের বিন্দুগুলোকে স্থানাঙ্ক অনুসারে সর্ট করতে হবে অর্থাৎ প্রথমে x অনুযায়ী এবং তাদের x সমান হলে y অনুযায়ী। এবার আগের মতো সর্ট করা বিন্দুগুলোকে একে একে নিব। স্ট্যাকের উপরের 2 টি বিন্দু এবং বর্তমান বিন্দু নিয়ে যাচাই করে যদি দেখি CCW তাহলে স্ট্যাকের উপরের বিন্দুকে ফেলে দিব (এখন কিন্তু আমরা বাম দিক থেকে ডান দিকে যাচ্ছি এবং আমরা শুধু উপরের বিন্দুকে ফেলে দিব এবং আমরা বাম দিক থেকে ডান দিকে শুধু দিকে যাব এবং আগের মতো যদি স্ট্যাকের উপরের দুটি বিন্দুর সঙ্গে বর্তমান বিন্দু CCW এ থাকে তাহলে স্ট্যাকের উপরের বিন্দুটি ফেলে দিব। এভাবে একবার বাম দিক থেকে ডানে এবং আরেকবার ডান থেকে বাম গেলে আমরা উপরের হাল এবং নিচের হাল তৈরি করে ফেলতে পারব। এই অ্যালগরিদম

আগের থেকে স্থিতিশীল কারণ এখানে কোণ অনুসারে সং করার কোনো ব্যাপার নেই। তোমরা চাইলে উইকিপিডিয়াতে দেওয়া ইমপ্লিমেন্টেশন (implementation)^১ টি দেখে নিতে পার।

১০.৩.২ নিকটতম বিন্দুজোড় (Closest pair of points)

২D স্থানাঙ্ক ব্যবস্থায় n টি বিন্দুর স্থানাঙ্ক দেওয়া আছে। তোমাকে এদের মধ্যের দূরত্বের দিক থেকে নিকটতম বিন্দুজোড় বের করতে হবে অর্থাৎ যে দুটি বিন্দুর মধ্যের দূরত্ব সবচেয়ে কম সেই দুটি বিন্দু বের করতে হবে বা সেই দূরত্ব বের করতে হবে। এখানে দূরত্ব মাপতে আমরা ইউক্লিডীয় দূরত্ব (euclidean distance) ব্যবহার করব। দুটি বিন্দুর স্থানাঙ্ক যদি (x_1, y_1) এবং (x_2, y_2) হয় তাহলে তাদের মধ্যের ইউক্লিডীয় দূরত্ব হবে $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. একে সরলরেখিক দূরত্ব (straight line distance) ও বলা যায়। এই সমস্যা সমাধানের জন্য আমাদের প্রথমে যা করতে হবে তা হলো বিন্দুগুলোকে x অনুযায়ী সর্ট করতে হবে (ছোট থেকে বড়)। এরপর আমরা এই বিন্দুগুলোকে দুই ভাগে ভাগ করব, এক ভাগে (বাম ভাগে) থাকবে ছোট x আরেক ভাগে বড়গুলো। মোটামুটি সমান দুই ভাগে ভাগ করতে হবে। অর্থাৎ বিজোড়ের ক্ষেত্রে একদিকে একটি বেশি থাকতে পারে আর কি! এখন তোমাদের রিকার্সিভ (recursive) উপায়ে দুই দিকের জন্য নিকটতম জোড়া বের করার ফাংশন কল করতে হবে। *closestpair* ফাংশন দুটি জিনিস দেবে- এক: তার পাওয়া বিন্দুগুলোর মধ্যে নিকটতম বিন্দুজোড়ের দূরত্ব এবং দুই: তার পাওয়া বিন্দুগুলোর y অনুযায়ী সর্টেড লিস্ট (বড় থেকে ছোট)। যদি তোমার ফাংশন মাত্র একটি বিন্দু পায় (base কেইস) তাহলে ধরা যাক সে ০০ বলবে নিকটতম বিন্দুজোড়ের দূরত্ব হিসেবে আর একটি বিন্দুর জন্য তো সর্ট করার কিছু নেই। এখন যদি একের বেশি বিন্দু হয় তাহলে তো আমরা দুই ভাগে ভাগ করে রিকার্সিভ কল করেছিলাম। ধরা যাক আমরা দুই দিক থেকে নিকটতম বিন্দুজোড়ের দূরত্ব পেয়েছি d_1 এবং d_2 । আমরা $d = \min(d_1, d_2)$ নিয়েই শুধু আগ্রহী। আরও মনে করা যাক, দুই দিকের y অনুসারে সর্টেড লিস্ট হলো P_1 এবং P_2 । এখন আশা করি বুঝতে পারছ কীভাবে এদের মিলিত y অনুসারে সর্ট করা লিস্ট পাওয়া যাবে? ঠিক মার্জ সর্ট (merge sort) এর মতো। দুই লিস্টের মাথা দেখবে এবং যার y বেশি তাকে নিবে এভাবে চলতে থাকবে (আমরা কিন্তু বড় থেকে ছোট সর্ট করছি, যদিও যেকোনো এক ভাবে করলেই হলো)। এখন মনে কর বামের ভাগের সবচেয়ে বড় x হলো x_{divider} । এটি রিকার্সিভ কল করার আগে x অনুযায়ী সর্টেড লিস্ট হতে বের করে একটি লোকাল ভ্যারিয়েবল (local variable) এ রাখতে পারো। খেয়াল কর, রিকার্সিভ কল করার পর কিন্তু সেই লিস্ট y অনুযায়ী সর্টেড হয়ে যাবে। সুতরাং আমাদের আগেই এই x_{divider} বের করে রাখতে হবে (অথবা আরও অনেক কৌশল খাটানো যায় যা আমরা পরে সংক্ষেপে বলব)। রিকার্সিভ কল শেষে পাওয়া y অনুযায়ী সর্ট করা বিন্দুগুলোকে ব্যবহার করে এদের মধ্যের নিকটতম বিন্দুজোড়ের দূরত্ব বের করতে পারি। প্রথমে খেয়াল কর, কোনো বিন্দুর x যদি $x_{\text{divider}} - d$ এর থেকে ছোট হয় বা $x_{\text{divider}} + d$ এর থেকে বড় হয় তাহলে সেই বিন্দুকে আমরা বিবেচনা না করলেও পারি (তবে সর্টেড লিস্ট পাওয়ার জন্য আমাদের সব বিন্দুই বিবেচনা করতে হবে)। কারণ আমরা এখন শুধু মাত্র এমন জোড়ার প্রতি

^১https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain

আগ্রহী যাদের একটি বিন্দু বাম ভাগে আরেক বিন্দু ডান ভাগে থাকে। বাম ভাগেই যদি দুটি বিন্দু থাকে তাহলে সেটা তো আমরা রিকার্সিভ উপায়ে বের করেই ফেলেছি। এখন মনে কর P_1 থেকে আমরা যাদের বিবেচনা করব তারা হলো P'_1 এবং একইভাবে P_2 হতে P'_2 এবং এরা y অনুযায়ী সর্টেড। এখন আমাদের দুটি পয়েন্টার (pointer) লাগবে যা দুই লিস্টের দুটি বিন্দুকে নির্দেশ বা পয়েন্ট (point) করবে। শুরুতে এরা লিস্টের শুরুর উপাদানগুলোকে পয়েন্ট করবে। এখন দেখ যদি P'_1 লিস্টের পয়েন্টকৃত বিন্দুর y যদি P'_1 এর পয়েন্টকৃত বিন্দুর y থেকে বেশি হয় তাহলে আমরা P'_2 এর পয়েন্টকৃত বিন্দুকে প্রসেসিং করব। প্রসেসিং করা শেষে এই পয়েন্টারকে বাড়িয়ে দেব। কথা হলো কীভাবে প্রসেসিং করব? ধরা যাক P'_2 এর পয়েন্টকৃত বিন্দুর y হলো y_1 . তাহলে আমাদের P'_1 এর সেসব বিন্দুর সঙ্গে তুলনা করতে হবে যাদের y $[y_1 - d, y_1 + d]$ এর মধ্যে থাকবে। প্রমাণ করা যায় যে এরকম বিন্দুর সংখ্যা আসলে 6 টির বেশি হতে পারে না। সুতরাং এভাবে $O(n)$ সময়ে তোমরা P_1 আর P_2 এর মধ্যের নিকটতম বিন্দুজোড় বের করে ফেলতে পারবে। সুতরাং আমাদের পুরো অ্যালগরিদমের কমপ্লেক্সিটি হবে $O(n \log n)$ ।

কিছুক্ষণ আগে কিছু কৌশল বলব বলেছিলাম। কৌশলটা হলো তুমি মূল অ্যারে সর্ট না করে একটি ইনডেক্স (index) এর অ্যারেকে সর্ট করতে পার। আবার তুমি চাইলে একটি সাহায্যকারী অ্যারে নিয়ে তাতে সর্ট করতে পার (মানে আরেকটা লিস্ট আর কি)।

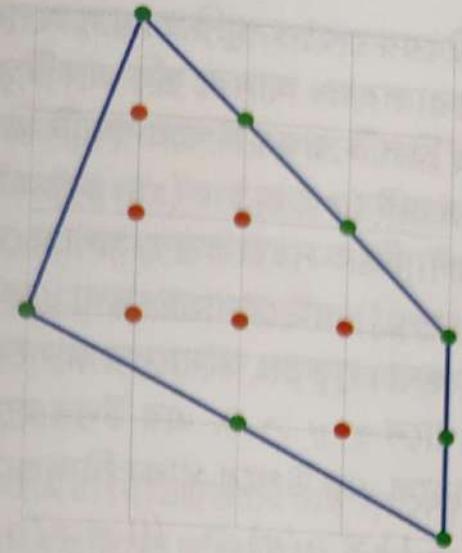
তোমরা চাইলে STL এর সাহায্যে আরও সহজে এই সমস্যা সমাধান করতে পার। এজন্য দুটি সেট (set) এর প্রয়োজন, একটি সেটে বিন্দুগুলো x অনুযায়ী সাজানো থাকবে অপরটি y অনুযায়ী। আমরা তাদের নাম দেই যথাক্রমে X এবং Y . আর এখন পর্যন্ত প্রাপ্ত নিকটতম বিন্দুজোড়ের দূরত্ব ধরা যাক d তে থাকবে যাতে শুরুতে ∞ মান থাকবে। এখন প্রথমে আমাদের দুটি সেটই ফাঁকা এবং আমাদের কাছে x অনুযায়ী সর্ট করা বিন্দুদের একটি লিস্ট আছে। এখন আমরা এই লিস্ট থেকে একে একে বিন্দুগুলোকে নিব। ধরা যাক এই বিন্দুটি হলো (x, y) , তাহলে আমাদের প্রথমে X এ দেখতে হবে $x - d$ এর থেকেও ছোট x ওয়ালা কোনো বিন্দু আছে কিনা থাকলে তাকে X এবং Y উভয় থেকেই মুছতে হবে। এরকম সব বিন্দু মোছা হয়ে গেলে আমাদের Y নিয়ে কাজ করতে হবে। আমরা Y এ lower bound ব্যবহার করে $y - d$ থেকে $y + d$ এর ভেতরে y এর মানওয়ালা যত বিন্দু Y এ আছে তাদের সঙ্গে বর্তমান বিন্দুর দূরত্ব বের করে নিকটতম বিন্দুজোড়ের দূরত্ব d কে আপডেট করব এবং সেই সঙ্গে X ও Y এ বর্তমান বিন্দু চুকিয়ে দেব। সত্যি কথা বলতে আমাদের X সেটের দরকার নেই, তোমরা x অনুযায়ী সর্টেড অ্যারেতে দুইটি হাত রেখেই এই কাজ করতে পার।

১০.৩.৩ পরস্পরচেদী রেখাংশ (Line segment intersection)

ধরা যাক তোমাদের অনেকগুলো রেখাংশ (line segment) দেওয়া আছে, বলতে হবে এদের মধ্যে কতগুলো ছেদ বিন্দু আছে, অর্থাৎ কতগুলো রেখাংশজোড় পরস্পরকে ছেদ করে। যাতে ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশ ঝামেলা না পাকাতে পারে সেজন্য অনেক সময় বলা হয় দুইয়ের বেশি রেখাংশ পরস্পরকে একই বিন্দুতে ছেদ করে না। এই সমস্যা সমাধানের জন্য আমাদের একটি ব্যালেন্সড বাইনারি সার্চ ট্রি (balanced binary search tree) এবং একটি প্রায়োরিটি কিউ

(priority queue) বা হৈপ (heap) দরকার। হৈপে অনেকগুলো ইভেন্ট (event) থাকবে এবং সবচেয়ে কম x এর ইভেন্টটি top এ থাকবে। প্রথমে সব রেখাংশের শুরু এবং শেষ প্রান্ত হৈপে প্রবেশ করাই। এবার আমরা হৈপ থেকে সবচেয়ে কম x ওয়ালা ইভেন্ট তুলব। যদি এটি কোনো রেখাংশের শুরুর প্রান্ত হয় তাহলে আমরা এই রেখাংশকে বাইনারি সার্চ ট্রি (binary search tree - BST) তে প্রবেশ করাব আর যদি এটি শেষ প্রান্ত হয় তাহলে সেই রেখাংশটি আমরা BST থেকে সরিয়ে নিব। BST টি হবে রেখাংশগুলোর উপর হতে নিচে এই ক্রমে। তবে এই ক্রম আসলে কোন x এর জন্য y দেখা হবে তার উপর নির্ভর করে। মনে কর দুটি রেখাংশ পরস্পরকে ছেদ করে তাহলে, ছেদ করার আগে একটি উপরে থাকে আর ছেদ করার পরে আরেকটি। সুতরাং রেখাংশদের নিজেদের মধ্যে কোনো ক্রম নেই। আমরা একটি x বলে দেবো, সেই x এ রেখাংশগুলোর y এর ক্রমে BST তে রেখাংশগুলো থাকবে। খেয়াল কর, যখন তুমি কোনো একটি নতুন রেখাংশ প্রবেশ করাবে তখন এর ঠিক উপরে এবং ঠিক নিচে একটি রেখাংশ থাকবে (BST যেহেতু y এর অর্ডারে থাকে সেহেতু বলা যায় রেখাংশের ঠিক আগের ও ঠিক পরের দুটি রেখাংশ)। তোমাকে এই দুটি রেখাংশের সঙ্গে এই নতুন রেখাংশের ছেদবিন্দু গুলো বের করে তা ইভেন্ট আকারে হৈপে ঢোকাতে হবে এবং ওই দুই পাশের দুই রেখাংশের মধ্যের ছেদবিন্দুর ইভেন্টটি হৈপ হতে সরিয়ে ফেলতে হবে। অর্থাৎ আমাদের তিন ধরনের ইভেন্ট আছে, একটি হলো রেখাংশের শুরু, একটি শেষ আরেকটি হলো ছেদবিন্দু। শুরু আর শেষে কী করতে হবে তা জানি, কিন্তু ছেদবিন্দুতে কী করব? ছেদবিন্দুতে আমাদের ওই দুটি রেখাংশের ক্রম পালিয়ে ফেলতে হবে, অর্থাৎ আগের উপরের রেখাংশ এবার নিচে চলে যাবে আর নিচেরটি উপরে চলে যাবে। এই হলো পুরো সমাধান কিন্তু এই জিনিসটি BST দিয়ে ইমপ্লিমেন্টেশন করা আসলেই অনেক কষ্টকর। তবে আমরা খুব সহজেই STL এর *set* ব্যবহার করে এই সমস্যা সমাধান করে ফেলতে পারি।

প্রথমে যা করতে হবে তা হলো রেখাংশের একটি সেট। যেহেতু আমরা সেট ব্যবহার করছি সেহেতু আমাদের একটি *comparison* ফাংশন লিখতে হবে যা দুটি রেখাংশের মধ্যে তুলনা করবে। তবে এই তুলনা করার জন্য আমাদের একটি x দরকার। কোন x এর সাপেক্ষে আমরা এই তুলনা করব? ধরা যাক এই x একটি গ্লোবাল ভ্যারিয়েবল (global variable) এ থাকবে এবং আমাদের *comparison* ফাংশনকে যখন দুটি রেখাংশ দিয়ে জিজ্ঞাসা করা হবে কোনটি ছোট আর কোনটি বড়? তখন আমরা এই x এ রেখাংশ দুটির y বের করে বলে দেব কোনটি বড় আর কোনটি ছোট। সুতরাং সেটে পরিবর্তনের আগে আমাদের খেয়াল রাখতে হবে x এর মান যেন সঠিক হয়। যেমন, তুমি কোনো একটি রেখাংশ শুরুর ইভেন্ট পেয়েছ, এখন তুমি সেটে এই রেখাংশ ঢোকাতে চাও। তাহলে তোমাকে x কে সেই ইভেন্টের x এর সমান করে নিতে হবে ইনসার্ট (insert) এর আগে, এরপর ইনসার্ট করতে হবে। একইভাবে যখন রেখাংশ শেষের ইভেন্ট পাবে তখন রেখাংশকে সরিয়ে ফেলার আগে x কে সেই ইভেন্টের x এর সমান করে নিতে হবে, এরপর সরাও। এখন যদি কোনো ছেদবিন্দু পাও তাহলে তো আমাদের পাল্টাপাল্টি (swapping) করতে হবে তাই না? এটি কীভাবে করা যায়? খুব সহজ, মনে কর ছেদবিন্দুর ইভেন্টের x হলো x' . তাহলে প্রথমে $x = x' - \epsilon$ বসাও এবং সেই দুটি রেখাংশকে সেট হতে সরাও, এর পর $x = x' + \epsilon$ বসিয়ে তাদের আবার ইনসার্ট কর, তাহলেই তারা পাল্টাপাল্টি হয়ে যাবে, এখানে ϵ হলো খুব ছোট মান। ছেদবিন্দু বা সেট থেকে সরানোর পর কোনো একটি রেখাংশ নিয়ে তার lower bound বা upper bound করে



নকশা ১০.৩: পিকের থেওরেম (Pick's theorem)

আমরা খুব সহজেই তার ঠিক আগে এবং পরের রেখাংশগুলো বের করে ফেলতে পারি। এভাবেই আমরা $O(I \log n)$ এ এই সমস্যার সমাধান করতে পারি যেখানে I হলো রেখাংশগুলোর মধ্যের ছেদবিন্দুর সংখ্যা।

১০.৩.৮ পিকের থেওরেম (Pick's theorem)

পিকের থেওরেম (Pick's theorem) বলে $2D$ স্থানাঙ্ক ব্যবস্থায় যদি আমরা কোনো সাধারণ বহুজ বা simple polygon আঁকি অর্থাৎ এমন একটি বহুজ যেখানে কোনো বাহু অপর বাহুর সঙ্গে ছেদ করে না বা স্পর্শও করে না (পাশাপাশি দুটি বাহু তাদের ভাট্টেরে স্পর্শ করতে পারবে) তাহলে $A = i + \frac{b}{2} - 1$ হবে (বহুজের সব শীর্ষ বিন্দুকে অবশ্যই পূর্ণ সংখ্যাবিশিষ্ট স্থানাঙ্কে থাকতে হবে)। এখানে A হলো বহুজের ক্ষেত্রফল, i হলো বহুজের অভ্যন্তরে (internal) কতগুলো পূর্ণ সংখ্যাবিশিষ্ট স্থানাঙ্ক আছে এবং b হলো বহুজের পরিসীমার উপরে কতগুলো পূর্ণ সংখ্যাবিশিষ্ট স্থানাঙ্ক আছে। যেমন চিত্র ১০.৩ এ লাল বিন্দুগুলো হলো অভ্যন্তরের বিন্দু $i = 7$, সবুজ বিন্দুগুলো পরিসীমার উপরের বিন্দু $b = 8$ এবং পুরো বহুজের ক্ষেত্রফল $A = 10$. এখানে বলে রাখা যায় যে, চিত্রের বহুজের জন্য ক্ষেত্রফল বের করা কিন্তু খুব একটা কঠিন না, তুমি কল্পনা কর $y = 2$ এই রেখার উপরে একটি ত্রিভুজ এবং নিচে একটি ত্রিভুজ। আর ত্রিভুজের ক্ষেত্রফল তো বের করা খুই সোজা। যাই হোক, সুতরাং আমরা এখন A , i এবং b এর মান জানি, পিকের সূত্রে বসিয়ে দেখি এটি কাজ করে কিনা! $10 = 7 + \frac{8}{2} - 1$ একদম সঠিক!

এখন একটি সমস্যা দেখা যাক। মনে কর তোমাদের $2D$ স্থানাঙ্ক ব্যবস্থায় একটি ত্রিভুজ দেওয়া আছে। এই ত্রিভুজের উপর (বাহুর উপর) এবং এই ত্রিভুজের অভ্যন্তরে কতগুলো বিন্দু (পূর্ণসাংখ্যিক বিন্দু বা পূর্ণ সংখ্যাবিশিষ্ট স্থানাঙ্ক বা ল্যাটিস বিন্দু (lattice point)) আছে তা বের করতে হবে। ত্রিভুজের ক্ষেত্রফলের গন্ত আছে। সুত্রের দিকে তাকালে আমরা বুঝতে পারব যে শুধু মাত্র এই ত্রিভুজের ক্ষেত্রফল আমাদের জানা। আমরা যদি কোনোভাবে ভেতরে কয়টি বিন্দু আছে সেটি বা

বাহুর উপরে কয়টি বিন্দু আছে সেটি বের করতে পারি তাহলে অপরটিও বের হয়ে যাবে। বাহুগুলোর উপরে কয়টি বিন্দু আছে তা বের করা সহজ। আমরা যদি একটি বাহুর উপরে কয়টি বিন্দু আছে তা বের করতে পারি তাহলে তিনটি বাহুর উপরে কয়টি বিন্দু আছে তাও বের করে ফেলতে পারব। ধরা যাক আমরা বের করতে চাই $(x_1, y_1) - (x_2, y_2)$ এই বাহুর উপরে কয়টি বিন্দু আছে। আমরা এখন একটু একটু করে সমস্যাটিকে সহজ করব, প্রথমে দেখ $(x_1, y_1) - (x_2, y_2)$ রেখাংশ দেখা আর $(0, 0) - (x_1 - x_2, y_1 - y_2)$ এই রেখাংশ দেখা একই কথা। আবার $(0, 0) - (|x_1 - x_2|, |y_1 - y_2|)$ দেখাও কিন্তু একই কথা। সুতরাং আমাদের আসলে বের করতে হবে $(0, 0) - (x, y)$ বাহুর উপরে কয়টি বিন্দু আছে যেখানে $x, y \geq 0$. এর উত্তর হলো $\gcd(x, y) + 1$. কেন? ধর g হলো তাদের গ.স.গ. (gcd), তাহলে এর উপরে থাকা বিন্দুগুলো হলো: $(x = g \times x/g, y = g \times y/g), ((g - 1) \times x/g, (g - 1) \times y/g), \dots (0 \times x/g, 0 \times y/g)$. এভাবে আমরা b এর মান বের করে ফেলতে পারি। A আর b জানলে তো i বের করাই যায়।

১০.৩.৫ বহুভুজ সম্পর্কিত টুকিটাকি

একটি বহুভুজ দিয়ে যদি বলে এই বহুভুজের ক্ষেত্রফল বের করতে হবে কীভাবে করবে? আলোচনার সুবিধার জন্য ধরে নিলাম আমাদের বহুভুজ হলো $P_0P_1P_2\dots P_{n-1}$. অনেকে মনে করতে পার যে $\triangle P_0P_1P_2, \triangle P_0P_2P_3, \dots \triangle P_0P_{n-2}P_{n-1}$ এই ত্রিভুজগুলোর ক্ষেত্রফল যোগ করলেই তো পুরো বহুভুজের ক্ষেত্রফল বের হয়ে যাবে। না তা ঠিক না। এটি ঠিক হবে সব কনভেক্স বহুভুজের জন্য। কনকেইভ বহুভুজ (Concave polygon) এর জন্য এটি সত্য নাও হতে পারে। একটি বহুভুজের কোনো কোণই যদি 180° এর চেয়ে বড় নাহয় তাহলে তাকে কনভেক্স বহুভুজ বলে। আর যদি কোনো একটি 180° এর থেকে বড় হয় তাহলে তাকে কনকেইভ বহুভুজ বলে। তাহলে আমরা কীভাবে বের করতে পারি? উত্তর হলো চিহ্নযুক্ত ক্ষেত্রফল (signed area) ব্যবহার করে। চিহ্নযুক্ত ক্ষেত্রফল (signed area) কী? আমরা এই অধ্যায়ের শুরুর দিকে ত্রিভুজের ক্ষেত্রফল বের করার জন্য নির্ণয়ক ব্যবহার করেছিলাম। সেভাবে বের করার জন্য বলেছিলাম যে পরম মান নিতে। কিন্তু তুমি যদি পরম মান না নাও এবং উপরের মতো $\triangle P_0P_1P_2, \triangle P_0P_2P_3, \dots \triangle P_0P_{n-2}P_{n-1}$ এর এই চিহ্নযুক্ত মানগুলো যোগ কর তাহলেই তোমরা পুরো বহুভুজের চিহ্নযুক্ত ক্ষেত্রফল পেয়ে যাবে। অর্থাৎ এই চিহ্নযুক্ত মানগুলো যোগ করে যদি তুমি পরম মান নাও তাহলে তুমি ক্ষেত্রফল পাবে। তুমি চাইলে এই কাজ P_0 কে কেন্দ্র করে না করে মূলবিন্দুকে কেন্দ্র করেও করতে পার। আমি এভাবেই করি, এক্ষেত্রে সুত্র খুব ছোট হয়ে যায়। $\sum (x_i y_{i+1} - y_i x_{i+1})$ (i এর মান $n - 1$ হলে $i + 1 = 0$ ধরতে হবে কিন্তু!)। এটি আমার জন্য মনে রাখা অনেক সহজ। অন্যান্য অনেক কাজেই এই চিহ্নযুক্ত মান ব্যবহার করে অনেক সহজে সমস্যা সমাধান করা যায়।

মনে কর তোমাদের একটি বহুভুজ আর একটি বিন্দু দিয়ে বলল যে এই বিন্দুটি বহুভুজের ভেতরে আছে না বাইরে? কীভাবে করবে? যদি এটি কনভেক্স বহুভুজের ক্ষেত্রে হয় তাহলে কিন্তু ব্যাপারটি বেশ সহজ। মনে কর তোমাদের বিন্দুটি হলো (x', y') তাহলে $y = y'$ এ বহুভুজের দুটি ছেদবিন্দু বের করতে হবে। যদি তোমার x' এই দুই ছেদবিন্দুর x এর মধ্যে থাকে তাহলে বলা যাবে যে

আমাদের বিন্দু বহুজের ভেতরে আছে। বহুজের সঙ্গে কোনো একটি $y = y'$ রেখার ছেদবিন্দু বের করা কিন্তু কঠিন কিছু না। তুমি সব বাহু দেখবে এবং তাদের জন্য সমাধান করবে, যদি কোনো শীর্ষ বিন্দুতে ছেদ করে তাহলে একটু সাবধান হতে হবে আর কি! যাই হোক, আমাদের এই পদ্ধতি কিন্তু কলকেইভ বহুজে কাজ করবে না। কনভেক্স বহুজের ক্ষেত্রে আরও একটি সহজ উপায় আছে। সেটি হলো মনে কর তোমরা জানতে চাইছ P বিন্দু কি বহুজের ভেতরে কি না। তোমরা যা করবে তা হলো PP_0P_1, PP_1P_2 এরকম করে সব বিভিন্ন ক্ষেত্রফল বের করে যোগ করবে (চিহ্নিত না)। এখন দেখবে এই যোগফল কি বহুজের ক্ষেত্রফলের সমান কি না। সমান হলে ভেতরে, আর সমান না হলে বাইরে।

কলকেইভ বহুজের ক্ষেত্রে আমরা বিভিন্নভাবে এই সমস্যা সমাধান করতে পারি। একটি উপায় হলো প্রদত্ত বিন্দু থেকে যেকোনো একদিকে রশ্মি টান। খেয়াল রাখতে হবে যেন এই রশ্মি বহুজের কোনো শীর্ষ দিয়ে যেন না যায়। তুমি এই কাজ খুব সহজে একটি দৈব (random) দিক নির্বাচন করে করতে পার। যদি সেই দিকে কোনো শীর্ষ বিন্দু থাকে তাহলে আরেকটি দৈব দিক নিবে, এভাবে চলতে থাকবে। আসলে দুই একবারের বেশি লাগার কথা না। এখন তোমাকে দেখতে হবে যে এই রশ্মি তোমার বহুজকে কয় জায়গায় ছেদ করে। যদি এই সংখ্যা বিজোড় হয় তার মানে তুমি বহুজের ভেতরে আছ আর যদি জোড় সংখ্যক হয় তার মানে তুমি বাইরে আছ। এর থেকেও সহজ উপায় আছে সমাধান করার। সেটি হলো ওয়াইন্ডিং সংখ্যা (winding number). তোমার বিন্দুর চারিদিকে বহুজ কয় পাক মারে সেটিই হলো ওয়াইন্ডিং সংখ্যা (winding number). তোমার বিন্দু সাপেক্ষে সবগুলো বাহুর জন্য চিহ্নিত কোণের যোগফল বের করলেই তুমি সমাধান করে ফেলতে পারবে। কিন্তু এটি একটু ঝামেলা মনে হয় আমার কাছে, আর তাছাড়া দরকার না হলে আমি ফ্লোটিং পয়েন্ট সংখ্যার হিসাবনিকাশ হতে বিরত থাকি। সেজন্য আমি আসলে উপরে বলা রে অটিং অ্যালগরিদম (ray shooting algorithm) এর মতো করে সমাধান করে থাকি। আমরা প্রদত্ত বিন্দু হতে ঠিক ডান দিকে একটি রশ্মি টানি। যেহেতু তোমার কুয়েরি (query) বিন্দুর সমান y ওয়ালা একটি বিন্দু বহুজের শীর্ষ বিন্দু হতে পারে সেহেতু আমরা আমাদের প্রদত্ত বিন্দুটির y স্থানাঙ্ককে $y - \epsilon$ হিসেবে ভাবতে পারি, অর্থাৎ (x, y) যদি ভেতরে থাকে তাহলে $(x, y - \epsilon)$ ও ভেতরে থাকবে, সুবিধা হলো $y - \epsilon$ এ আঁকা x অক্ষের সমান্তরাল রেখা কোনো শীর্ষ বিন্দু দিয়ে যায় না (এক্ষেত্রে অবশ্য তোমাকে দেখতে হবে এই বিন্দুটি তোমার বহুজের উপরে আছে কি না, থাকলে এই অল্প পরিমাণ সরানোর ফলে কিন্তু উভয় আলাদা হয়ে যেতে পারে)। এখন প্রতিটি বাহু নিতে হবে আর দেখতে হবে এটি এই রশ্মিকে ছেদ করে কিনা। যদি করে তাহলে সেই বাহুটি নিচ থেকে উপরে যাচ্ছিল নাকি উপর থেকে নিচে (আমরা আসলে বাহুগুলোকে একটি ক্রমে নিবো P_0P_1, P_1P_2 এরকম করে পর পর)? ধরা যাক এই দুটি ক্ষেত্রে যথাক্রমে আমরা 1 যোগ ও বিয়োগ করি (আসলে এক অর্থে এটি জোড় বিজোড় বের করা)। তাহলে সব বাহু বিবেচনা করার পর যদি দেখি আমাদের যোগফল শূন্য তাহলে আমরা বুঝব যে আমাদের বিন্দু বাইরে আছে, আর যদি শূন্য না হয় এর মানে এই বিন্দু ভেতরে আছে।

মনে কর তোমাদের কিছু বহুজ দেওয়া আছে, বলা হলো এদের ইউনিয়ন (union) এর ক্ষেত্রফল বের করতে। ইউনিয়ন (union) বলতে আমরা বুঝি যদি কোনো জিনিস একাধিকবার নেওয়া হয়ে থাকে তবুও সেই জিনিসকে আমরা একবারই বিবেচনা করব। যেমন ধর দুটি

বর্গক্ষেত্র নেওয়া হলো যেন একটি আরেকটিকে পুরোপুরিভাবে ঢেকে ফেলে। তাহলে তাদের ইউনিয়নের ক্ষেত্রফল হবে বড়টির ক্ষেত্রফলের সমান। এখন এই সমস্যার সমাধান কী? প্রথমে সব বহুভুজগুলোকে একদিকে ঘুরিয়ে নেওয়া মানে হলো হয় সবাই CW অথবা CCW. কনকেইভ বহুভুজের CW বা CCW বলে কিন্তু কোনো কথা নেই বলতে পার। কিন্তু তুমি চাইলে চিহ্নযুক্ত ক্ষেত্রফল বের করে কোনো বহুভুজ CW না CCW তা বলতে পারো। সূতরাং এভাবে তোমাকে সব বহুভুজকে একই দিকে ঘুরিয়ে নিতে হবে (অর্থাৎ উপরের চিহ্নযুক্ত ক্ষেত্রফল বের করার সূত্রে সবাই হয় ধনাত্মক হবে নাহয় ঋণাত্মক হবে)। এরপর আমাদের যা করতে হবে তা হলো প্রতিটি বহুভুজের প্রতিটি বাহু নিতে হবে এবং আমাদের অন্যান্য সব বাহু দিয়ে একে ছেদ করার চেষ্টা করতে হবে, এভাবে আমাদের বিবেচিত বাহুর উপর অন্যান্য সব বাহুর ছেদ বিন্দু বের করি। যদি আমরা প্যারামেট্রিক সমীকরণ ব্যবহার করি তাহলে খুব সহজেই এই ছেদ বিন্দুগুলোকে সর্ট করে আমরা কতিপয় রেখাংশ পাব। আমাদের বের করতে হবে এই রেখাংশের মধ্যবিন্দু নিয়ে অন্যান্য বহুভুজের ভেতরে আছে। এটি বের করার সহজ উপায় হলো এই রেখাংশের মধ্যবিন্দু নিয়ে অন্যান্য বহুভুজের ভেতরে আছে কিনা তা যাচাই করা। যেহেতু আগেই আমরা বাহুকে রেখাংশে ভাগ করে ফেলেছি সূতরাং এমন হবে না যে কোনো রেখাংশ আংশিকভাবে কোনো বহুভুজের ভেতরে আছে। যদি কোনো রেখাংশ অন্য কোনো বহুভুজের ভেতরে থেকে থাকে তাহলে তাকে ধরে ফেলে দাও! এখন বেঁচে থাকা রেখাংশগুলো নিয়ে যেকোনো বিন্দু (ধরা যাক মূলবিন্দু) এর সাপেক্ষে চিহ্নযুক্ত ক্ষেত্রফল বের কর। খেয়াল কর তুমি কিন্তু প্রথমেই বহুভুজকে একই দিকে ঘুরিয়ে নিয়েছিলে, সূতরাং প্রতিটি রেখাংশের একটি দিক আছে, সেই দিক ব্যবহার করে তোমাদের চিহ্নযুক্ত ক্ষেত্রফল বের করতে হবে। এভাবে সব চিহ্নযুক্ত ক্ষেত্রফল যোগ করলে তুমি ক্ষেত্রফলের ইউনিয়ন পেয়ে যাবে। এই সমাধানটি খুবই চমৎকার একটি সমাধান যা তোমাদের জেনে রাখা উচিত।

তোমরা চাইলে অন্যভাবেও ইউনিয়নের ক্ষেত্রফল বের করতে পারো। প্রথমে আগের মতো সব ছেদবিন্দু বের কর। এর পর সব ভাট্টের আর ছেদবিন্দু বিন্দুগুলোর x নিয়ে সর্ট কর। এখন পাশাপাশি দুটি x এর মধ্যে কিন্তু কোনো বাহু ছেদ করবে না। এটিকে বলা হয় স্লাইসিং (slicing). অর্থাৎ তুমি বিভিন্ন x এ রেখা টেনে পুরো চিত্রকে ছোট ছোট টুকরো বা স্লাইস (slice) এ ভাগ করেছো। এখন প্রতিটি স্লাইসে গিয়ে দেখো এই স্লাইসের ভেতরে কোন কোন বাহুর অংশ আছে। এসব বাহুকে উপর হতে নিচ পর্যন্ত সর্ট কর। আর আমরা তো আগেই বহুভুজের বাহুগুলোকে একই দিকে ঘুরিয়ে নিয়েছি (CW বা CCW). মনে কর যদি CW হয় তাহলে opening আর CCW মানে closing. তাহলে কি দাঁড়াল? প্রতিটি স্লাইসের মধ্যে অনেক বাহু আছে, আমরা উপর হতে নিচে যাব, opening পেলে স্ট্যাকে ঢুকাব, closing পেলে বের করব। যখন কোনো একটি open এর জন্য দেখবো স্ট্যাক ফাঁকা তখন সেই বাহুকে মার্ক করে রাখব, আর কোনো একটি closing এর জন্য স্ট্যাক থেকে বাহু তোলার পর যদি দেখি স্ট্যাক ফাঁকা তখন সেই বাহুকেও মার্ক করে রাখব। এবার এই মার্ক করে রাখা বাহুগুলো নিয়ে ট্রাপিজিয়াম (trapezium) বানিয়ে (স্লাইসিংয়ের বাহুগুলো তো সমান্তরাল) তাদের ক্ষেত্রফল যোগ করতে হবে। এভাবে সব স্লাইসের জন্য ক্ষেত্রফলগুলো যোগ করলে আমরা মোট ক্ষেত্রফল পেয়ে যাব।

১০.৩.৬ লাইন সুইপ (Line sweep) এবং রোটেটিং ক্যালিপার্স (Rotating Calipers)

লাইন সুইপ (Line sweep) বা রোটেটিং ক্যালিপার্স (Rotating calipers) কোনো আলগরিদম না, বরং একটি সমাধানের পদ্ধতি। যেমন আমরা নিকটতম বিন্দুজোড় সমস্যায় যেই চিঠীয় সমাধান দেখেছিলাম সেটিকে লাইন সুইপ বলা যায় কারণ আমরা 2D স্থানান্তর ব্যবস্থায় একদিক হতে আরেকদিকে গিয়েছিলাম। রোটেটিং ক্যালিপার্স হলো কিছুটা লাইন সুইপের মতো, তবে প্রধান পার্থক্য হচ্ছে লাইন সুইপে আমরা একটি সরলরেখিক দিকে যেতে থাকি আর রোটেটিং ক্যালিপার্সে আমরা কৌণিক সুইপ (angular sweep) করে থাকি। অনেক সময় শুধু সুইপিং না, একটি window রেখে সুইপিং করা হয়ে থাকে। আমরা এই সংক্রান্ত কিছু সমস্যা এখন দেখব।

ধরা যাক একটি কনভেক্স বহুভুজ দিয়ে বলা হলো এর সব খেকে বড় কর্ণ (diagonal) বের করতে হবে। যেহেতু বহুভুজটি কনভেক্স সূতরাং এটি বলাই যায় যে এর যেকোনো কর্ণ সম্পূর্ণভাবে বহুভুজের ভেতরে থাকবে। এখন আমরা এই সমস্যাকে একটু ভেঙে ভেঙে দেখি। মনে কর আমরা যদি প্রতিটি শীর্ষবিন্দু থেকে বের হওয়া কর্ণের মধ্যে সবচেয়ে বড়টি বের করতে পারি তাহলে তাদের মধ্যে সবচেয়ে বড়টিই আমাদের উত্তর। ধরা যাক, i তম শীর্ষের জন্য $f(i)$ হলো উপর শীর্ষ, তাহলে একটু খেয়াল করলে বুঝবে যে $i + 1$ জন্য উপর শীর্ষটা আসলে $f(i)$ বা এর কাছাকাছি কোনো একটি। অন্যভাবে বলা যায়, i থেকে CW দিকে গিয়ে যদি তুমি $i + 1$ পাও তাহলে $f(i)$ বা $f(i)$ এর থেকে কিছুটা CW গেলে তুমি $f(i + 1)$ পাবে। সূতরাং আমাদের যা করতে হবে তা হলো প্রথমে $i = 0$ নিতে হবে এবং এর জন্য আমাদের $j = f(0)$ বের করতে হবে। বের করার জন্য যা করবে তাহলে, $j = i = 0$ ধরবে এবং দেখবে যে (i, j) কর্ণ বড় নাকি $(i, j + 1)$ কর্ণ বড়। যদি পরেরটি বড় হয় তাহলে j কে এক বাড়িয়ে দিবে এবং পুনরায় একইভাবে যাচাই করবে। এভাবে ক্রতে করতে দেখবে এক সময় আর j কে বাড়ানো যাবে না কারণ (i, j) কর্ণ $(i, j + 1)$ থেকে বড়। তোমাদের এই মানই হবে $f(0)$. এবার তুমি i কে এক বাড়াও, এবং কিছুক্ষণ আগে যেই কাজ করছ সেই কাজ আবার কর। j এর মান যা ছিল সেখান থেকেই শুরু হবে, 0 করা বা i এর সমান করার দরকার নেই। এভাবে দরকার মতো j বাড়িয়ে $f(i + 1)$ বের করে ফেলবে এবং এই কাজ $i = 0$ হতে $n - 1$ পর্যন্ত করবে (n হলো বহুভুজের শীর্ষ সংখ্যা) অর্থাৎ $f(0) \dots f(n - 1)$ এর মানগুলো বের করবে। আসলে এভাবে সব $f(i)$ এর মান দরকার নেই, তুমি প্রতিবার যদি সর্বোচ্চ কর্ণের মান আপডেট কর তাহলেই হবে। এখন কথা হলো এই সমাধানের কমপ্লেক্সিটি কত? মাত্র $O(n)$. কারণ একটু চিন্তা করলে দেখবে যে j কে $2n$ বার এর বেশি বাড়ানোর দরকার হবে না। এখানে একটি জিনিস বলে নেই তা হলো, যদি সমস্যায় বলা থাকে যে বহুভুজের যেকোনো বাহুও একটি কর্ণ তাহলে এই সমাধান ঠিক আছে। তবে যদি বলে যে কর্ণটি কোনো বাহু হতে পারবে না, তাহলে j এর মান নির্দিষ্ট করার আগে একটু ভালো মতো দেখতে হবে। যেমন i কে বাড়ানোর পর প্রই দেখতে হবে যে j কি $i + 1$ এর সমান বা এর থেকে ছোট? তাহলে j কে $i + 2$ করে দিতে হবে। আবার j কে বাড়ানোর পর দেখতে হবে যে এটি কি $i - 1$ এর সমান হয়ে গেছে কিনা তাহলে আবার হবে না, $i - 2$ তে পরিবর্তন করে দিতে হবে। তোমরা আশা করি বুঝতে পারছ যে $j = n - 1$ ক্ষে যদি এক বাড়াতে হয় তাহলে কী করবে? তোমরা চাইলে if-else লাগাতে পার অথবা mod

অপারেশন ব্যবহার করতে পার। তবে n এর মান খুব ছোট হলে কিছুক্ষণ আগে যে বললাম যে "বাড়ানো কমানোর সময় খেয়াল রাখতে হবে" এই জিনিসটি হয়তো তোমাদের চিন্তা করতে একটু কষ্ট হবে, সেজন্য যেটি সহজ বুদ্ধি তা হলো $n < 10$ হলে সাধারণ $O(n^2)$ পদ্ধতি ব্যবহার করা। তাহলে আরও এই বাড়ানো কমানো নিয়ে আলাদা চিন্তা করতে হবে না।

এখন উপরের সমস্যাকে একটু পরিবর্তন করা যাক। উপরের সমস্যায় বড় কর্ণ বের করতে না দিয়ে যদি বলত কনভেক্স বহুজটির ভেতরে থাকে এরকম সবচেয়ে বড় রেখাংশ বের করতে হবে, তাহলে কী করতে? একটু চিন্তা করে দেখো, এই বড় রেখাংশকে অবশ্যই একটি কর্ণ হতে হবে। যদি তা না হয় তাহলে একটু কল্পনা কর, তোমার বের করা রেখাংশের দুই মাথাকে অবশ্যই বহুজের উপরে হতে হবে কারণ তা না হলে তুমি ওই রেখাংশকে বড় করতে পারবে। এখন এই রেখাংশের এমন একটি মাথা নাও যেটি বহুজের শীর্ষে নেই। যেহেতু এটি একটি বাহুর উপরে আছে সুতরাং তুমি সেই বাহু দিয়ে উপর প্রান্তকে দুইদিকের যেকোনো একদিকে ঠেলে দিলে অবশ্যই বড় রেখাংশ পাবে। কেন? কারণ খুব সহজ, তোমাকে যদি জিজ্ঞাসা করা হয় একটি বিন্দু আর একটি রেখা দেওয়া আছে তোমাকে ওই বিন্দু হতে ওই রেখার উপর সবচেয়ে ছোট রেখাংশ আঁকতে হবে তুমি কী করবে? লম্ব টানবে। এই লম্ব থেকে যেদিকেই যাও সেদিকেই বাড়বে। অর্থাৎ আমাদের ক্ষেত্রে আমরা বাহুর উপরের বিন্দুকে আমরা যদি লম্ব বিপরীত দিকে ঠেলে সরাতে থাকি তাহলে আমাদের রেখাংশের দৈর্ঘ্য বাড়তে থাকবে।

এখন আরও একটি সমস্যা দেখা যাক। মনে কর এবার তোমাদের বলা হলো একটি কনভেক্স বহুজের ভেতরে সবচেয়ে বড় ক্ষেত্রফলের ত্রিভুজ বের করতে হবে। যদি আমি বলি যে ত্রিভুজটির তিনটি শীর্ষই বহুজের কোনো না কোনো শীর্ষে থাকবে তাহলে আশা করি খুব একটা অবাক হবে না। এর কারণটা ও বেশ সহজ। ত্রিভুজের এমন একটি শীর্ষ নাও যেটি বহুজের শীর্ষে নেই। উপর দুই শীর্ষকে যথা�স্থানে রাখ। এখন যেই দুই শীর্ষ যথাস্থানে আছে তাদের মধ্যের বাহুকে ত্রিভুজের ভূমি মনে কর, তাহলে ত্রিভুজের ক্ষেত্রফলের সূত্র অনুসারে ত্রিভুজটির উচ্চতা যদি বাড়ে ক্ষেত্রফলও তাহলে বাড়বে। এখন তুমি এই ভূমির সমান্তরাল কিছু রেখা টান। সবচেয়ে দূরের যেই সমান্তরাল রেখা বহুজ কে ছেদ করে সেই ছেদবিন্দুতেই তুমি সবচেয়ে বড় ক্ষেত্রফলের ত্রিভুজ পাবে। আর এটি বলার অপেক্ষা রাখে না যে সেই ছেদবিন্দুগুলোর একটি অবশ্যই বহুজের শীর্ষ হবে। তাহলে কীভাবে সমাধান হবে আশা করি বুঝতে পারছ? প্রথমে $i = 0, j = 1, k = 2$ নাও। এবার k কে বাড়াতে থাক যতক্ষণ $i - j - k$ ত্রিভুজের ক্ষেত্রফল বাড়তে থাকে। এখন j কে বাড়াও দেখ k কে বাড়ালে ক্ষেত্রফল বাড়ে কিনা, যদি বাড়ে তাহলে আবার j কে বাড়ানোর চেষ্টা কর, এভাবে কিছুক্ষণ j আর k কে বাড়াতে থাকলে দেখবে আর বাড়ানো যাচ্ছে না। তখন তুমি i কে বাড়াবে এবং একইভাবে আবার $j - k$ কে বাড়ানোর চেষ্টা করবে। এই সমাধানও $O(n)$. তবে এটি কেন সঠিক তা প্রমাণ করা একটু কঠিন। তোমরা চাইলে নিজেরা প্রমাণ করে দেখতে পার অথবা ইন্টারনেটে খুঁজে দেখতে পার। সত্যি বলতে আমার নিজেরও প্রমাণটি জানা নেই তবে এটুকু জানি যে এটি সঠিক সমাধান! প্রতিযোগিতায় আমরা প্রায়ই intuition এর উপর নির্ভর করে সমাধান করে থাকি।

এতক্ষণ বহুজের ভেতরের জিনিস নিয়ে সমস্যা দেখেছি এবার একটু বাইরের জিনিস নিয়ে দেখা যাক। মনে কর অনেকগুলো বিন্দু দেওয়া আছে, তোমাদের সবচেয়ে ছোট ক্ষেত্রফলের আয়তক্ষেত্র বের করতে হবে যেন সব বিন্দু এই আয়তক্ষেত্রের ভেতরে থাকে। তুমি যদি ইন্টারনেটে

গার্চ কর তাহলে ডহাকাপাড়িয়াতে তে দেখবে মিনিমাম এনক্লোজিং আয়তক্ষেত্র (minimum enclosing rectangle) বা এরকম কোনো নামে একটি আর্টিকেল আছে এবং এতে বলা আছে যে এই আয়তক্ষেত্রটির একটি বাহু প্রদত্ত বিন্দুগুলো দিয়ে তৈরি কনভেক্স হালের কোনো একটি বাহু দিয়ে যায়। মনে কর আমরা কনভেক্স হাল বের করে ফেলেছি এবং এর একটি বাহু ($i, i+1$) দিয়ে আয়তক্ষেত্রটি যায়। তাহলে অপটিমাল (optimal) আয়তক্ষেত্রের ক্ষেত্রফল কত (যেন এই বাহু দিয়ে যায়)? আমরা কিন্তু খুব সহজেই আয়তক্ষেত্রের উচ্চতা বের করতে পারি কিছুক্ষণ আগে শেখা উপায়ে। কিন্তু প্রস্তুত বা দৈর্ঘ্য কীভাবে বের করতে পারি? এটিও খুব একটা কঠিন না, উচ্চতা বের করার মতো করে আমরা বাম দিকের সীমানা আর ডান দিকের সীমানা বের করতে পারি। কোনো একটি বিন্দু থেকে আমরা ওই বাহুর উপর লম্ব টানব। এই লম্ব ডানে আর বামে যত দূরে নেওয়া যায় নিব (লুপ চালিয়ে যেভাবে উচ্চতা বাড়তে থাকা পর্যন্ত আমরা ইনডেক্স বাড়িয়েছি সেভাবে)। তাহলেই আমরা আয়তক্ষেত্রের প্রস্তুত পেয়ে যাব। সুতরাং ($i, i+1$) দিয়ে যাওয়া অপটিমাল আয়তক্ষেত্রের ক্ষেত্রফল আমরা জেনে গেলাম। একইভাবে আমরা i কে বাড়াব এবং উচ্চতা, ডান আর বাম দিকের ভাট্টের পয়েন্টারকে আমরা আপডেট করতে থাকব। এভাবে আমরা $O(n \log n)$ এ কনভেক্স হাল বের করার পর $O(n)$ এ অপটিমাল আয়তক্ষেত্র বের করতে পারি।

এবার লাইন সুইপের কিছু সমস্যা দেখার আগে এর কিছু মূল জিনিস দেখে নেওয়া যাক। $2D$ স্থানাঙ্ক ব্যবস্থায় লাইন সুইপের সময় আমরা মূলত যা করি তা হলো একটি অক্ষ বরাবর এক দিক থেকে আরেকদিকে ধীরে ধীরে যাই (sweeping) এবং উপর অক্ষ বরাবর একটি ডেটা স্ট্রাকচার রেখে তাকে ধীরে ধীরে আপডেট করি। প্রথম অক্ষ বরাবর যে ধীরে ধীরে যাই এর মানে হলো আমরা কেবল মাত্র আমাদের জরুরি স্থানগুলোতেই থামব, যেমন নিকটতম বিন্দুজোড় সমস্যায় কিন্তু আমরা শুধু মাত্র সেসব x এই থেমেছিলাম যেখানে কোনো বিন্দু ছিল। কোন জায়গা জরুরি বা পরবর্তী কোন জায়গা জরুরি তা বের করার জন্য আমাদের আরেকটি ডেটা স্ট্রাকচার ব্যবহার করতে হয় সাধারণত। বিভিন্ন সমস্যায় বিভিন্ন ডেটা স্ট্রাকচার ব্যবহার করতে হয়, তবে মূল উদ্দেশ্য একই হয়ে থাকে। এখন কিছু সমস্যা দেখা যাক।

প্রথম সমস্যা হলো আয়তক্ষেত্রের ইউনিয়ন (union of rectangles). মনে কর তোমাকে অনেকগুলো আয়তক্ষেত্র দেওয়া হলো আর বলা হলো এদের ইউনিয়নের ক্ষেত্রফল বের করতে। কিছুক্ষণ আগেই আমরা দেখেছি কীভাবে অনেকগুলো কনভেক্স বহুভুজের ইউনিয়নের ক্ষেত্রফল বের করা যায়। কিন্তু সেই সমাধানের টাইম কমপ্লেক্সিটি অনেক ছিল। যেহেতু এই সমস্যায় আয়তক্ষেত্র দেওয়া আছে (যা কনভেক্স বহুভুজের তুলনায় অনেক বেশি সহজ স্ট্রাকচার) সেহেতু আমরা আশা করতে পারি যে এর আগের সমাধানের তুলনায় এই সমস্যার একটি সহজ সমাধান থাকবে। ইনপুট দিসেবে আয়তক্ষেত্রের উপরের ও নিচের y দেওয়া আছে আর ডান ও বামের x দেওয়া আছে। স্ল আমরা সমাধানটি দেখে নেই। আমরা x অক্ষের বাম থেকে ডান দিকে সুইপিং করব এবং y অক্ষ বরাবর একটি সেগমেন্ট ট্রি (segment tree) রাখব। আরও বেশি দূর যাওয়ার আগে আমাদের আরেকটি প্রচলিত পদ্ধতি জানতে হবে। সেটি হলো স্থানাঙ্ক সংকোচন (coordinate compression)। জ্যামিতিক সমস্যা সমাধানের সময় এই পদ্ধতি প্রায়ই ব্যবহার হয়ে থাকে। এই পদ্ধতির মূল কথা হলো সব স্থানাঙ্ক সবসময় দরকার হয় না, যেসব স্থানাঙ্ক লাগবে শুধু তাদের নিয়েই কাজ করা হলো স্থানাঙ্ক সংকোচন (coordinate compression)। এই সমস্যায় আমাদের যা

করতে হবে তা হলো আয়তক্ষেত্রগুলোর ডান মাথা আর বাম মাথা (দুটি করে x স্থানাঙ্ক) একটি লিস্টে নিয়ে তাদের স্ট করতে হবে। এর পর একটি লুপ চালিয়ে শুধুমাত্র স্বতন্ত্র (distinct) x স্থানাঙ্কগুলো বের করতে হবে। আমরা চাইলে এই স্বতন্ত্র স্থানাঙ্কগুলো ঐ একই লিস্টে রাখতে পারি বা আলাদা লিস্টে নিতে পারি। তোমরা চাইলে এই কাজ সেট ব্যবহার করেও করতে পারো। একইভাবে y এর স্বতন্ত্র স্থানাঙ্কসমূহও বের করতে হবে। ধরা যাক x এর জন্য যে লিস্ট পাওয়া গেছে তা হলো: $y[1], y[2] \dots y[ny]$. এখন আমাদের প্রতিটি আয়তক্ষেত্রের স্থানাঙ্কসমূহ আপডেট করতে হবে। যদি কোনো আয়তক্ষেত্রের উপরের বা নিচের y যদি হয় লিস্টের i তম উপাদান অর্থাৎ, $y[i]$ তাহলে তাকে i করে দাও। একইভাবে x স্থানাঙ্কগুলোকে পরিবর্তন করে দাও। এবার একটি ভেষ্টরের অ্যারে নাও যাব আকার হবে nx . এবার আয়তক্ষেত্রের উপর লুপ চালাও এবং প্রতিটির বামের x এর জন্য তার ভেষ্টরে লিখে রাখ যে এই x থেকে অমুক আয়তক্ষেত্রের সীমানা শুরু হয়েছে, একইভাবে অমুক x এ গিয়ে অমুক আয়তক্ষেত্র শেষ হয়েছে। তোমরা চাইলে এই কাজ ভেষ্টরে রেখে না করে একটি হৈপ বা প্রার্যোরিটি কিউ (priority queue) রেখেও করতে পারতে। অথবা একটি ভেষ্টরেও ইভেন্টগুলো রেখে স্ট করতে পারতে। যাই হোক, অন্যদিকে y অক্ষের জন্য আমাদের একটি সেগমেন্ট ট্রি বানাতে হবে যাব আকার হবে $ny - 1$. সেগমেন্ট ট্রি এর নোডগুলো হবে: $(1, 2), (2, 3) \dots (ny - 1, ny)$. আরও ভালো মতো বলতে গেলে এটি হবে: $(y[1] \sim y[2]), (y[2] \sim y[3]) \dots (y[ny - 1] \sim y[ny])$. এখন আমাদের x অক্ষ বরাবর ধীরে ধীরে এগাতে হবে। প্রতিটি x এ যাব আর তার ভেষ্টরে থাকা সব ইভেন্টকে আমাদের execute করতে হবে। বোঝাই যাচ্ছে যে ইভেন্ট দুই রকম হতে পারে, এক আয়তক্ষেত্রের শুরু আরেকটি হলো আয়তক্ষেত্রের শেষ। আয়তক্ষেত্র শুরুর সময় আমাদের যা করতে হবে তা হলো ওই আয়তক্ষেত্রের উপরের আর নিচের y যদি হয় যথাক্রমে $y[hi]$ এবং $y[lo]$ তাহলে $(y[lo] \sim y[lo + 1]), (y[lo + 1] \sim y[lo + 2]) \dots (y[hi - 1] \sim y[hi])$ এই সেগমেন্টের প্রতিটি স্থানে 1 যোগ করতে হবে বা সংক্ষেপে $[lo, hi - 1]$ এই সেগমেন্টের প্রতিটি স্থানে 1 যোগ করতে হবে। তাহলে আশা করি বোঝা যাচ্ছে যে আয়তক্ষেত্রের শেষ মাথার ইভেন্টে তোমাকে 1 করে বিয়োগ করতে হবে। এখন একটি জিনিস খেয়াল করো, তুমি যদি $x[i]$ এ যেসব ইভেন্ট আছে সেসব প্রসেসিং করা শেষে যদি সেগমেন্ট ট্রি তে যেসব জায়গায় অশূন্য (non-zero) মান আছে সেসব জায়গার মান (s এ অশূন্য থাকলে $y[s] - y[s - 1]$ যোগ হবে) যোগ কর এবং তাকে $(x[i + 1] - x[i])$ দিয়ে গুণ কর তাহলে $x[i]$ থেকে $x[i + 1]$ এর ভেতরে আয়তক্ষেত্রের ইউনিয়ন পাবে। এভাবে প্রত্যেক x যদি প্রসেসিং কর এবং যোগ কর তাহলেই তুমি তোমার কাজিত ক্ষেত্রফলের ইউনিয়ন পেয়ে যাবে। সেগমেন্ট ট্রি তে কুয়েরি কীভাবে করবে তা তোমাদের জন্য রেখে দেওয়া হল।

এবার একটি সহজ সমস্যা দেখা যাক। মনে কর তোমাকে $2D$ স্থানাঙ্ক ব্যবস্থায় অনেকগুলো অক্ষের সমান্তরাল রেখাংশ (axis parallel segment) দেওয়া আছে। তোমাকে বলতে হবে তাদের মধ্যে কতগুলো ছেবিন্দু আছে। মনে কর রেখাংশের সংখ্যা প্রায় $n \leq 100,000$ টি এবং স্থানাঙ্কগুলো সর্বোচ্চ 10^9 হতে পারে। আশা করি বোঝা যাচ্ছে যে প্রথমে তোমাদের স্থানাঙ্ক সংকোচন করে ফেলতে হবে। এর পরে x অক্ষ বরাবর একটি সেগমেন্ট ট্রি নাও এবং y অক্ষ বরাবর সুইপিং কর। মনে করা যাক উপর হতে নিচে সুইপিং করা হচ্ছে। সুইপিং করার মানে হচ্ছে ইভেন্ট প্রসেসিং

ক্রা। যদি y অক্ষের সমান্তরাল একটি রেখাংশের শুরুর মাথা পাও তাহলে সেগমেন্ট ট্রিতে ওই জয়গায় 1 বাড়াও। যদি শেষ মাথা পাও তাহলে 1 কমাও। আর যদি x অক্ষের সমান্তরাল একটি রেখা পাও যার x এর বিস্তার $[x_1, x_2]$ তাহলে সেগমেন্ট ট্রিতে এই সীমার জন্য একটি কুয়েরি করে গ্রীমার যোগফল বের করতে হবে। এই যোগফল তোমার উভরের সঙ্গে যোগ করতে থাক। তাহলেই তুমি মোট উভর পেয়ে যাবে। সহজ না?

এবার একটি IOI এর সমস্যা দেখা যাক। মনে কর n টি অক্ষের সমান্তরাল আয়তক্ষেত্র (axis parallel rectangle) দেওয়া আছে। তোমাকে এদের boundary এর দৈর্ঘ্য বের করতে হবে। সমস্যাটি খুব একটা কঠিন না, একটু বুদ্ধি খাটালে সমস্যাটি সহজ হয়ে যাবে। এখানে চার দরনের boundary হতে পারে। উপরে, নিচে, ডানে আর বামে। আবার উপরের boundary কিন্তু নিচের সমান। একইভাবে ডানেরটা বামের সমান। সুতরাং আমাদের দুটা বের করলেই চলে। আবার যদি আমরা উপরেরটি বের করতে পারি তাহলে x আর y পাল্টাপাল্টি করে একইভাবে ডানেরটি বের করতে পারি। সুতরাং আমরা এখন শুধু উপরেরটি কীভাবে বের করে তা দেখব। এখন আগের মতোই y অক্ষ বরাবর সেগমেন্ট ট্রি আর y অক্ষ বরাবর সুইপিং করব আমরা। আর উপরের boundary বের করার জন্য কিন্তু শুধু x অক্ষের সমান্তরাল রেখাংশই লাগবে y অক্ষের সমান্তরাল গুলোর কোনো প্রয়োজন নেই। তবে অবশ্যই রেখাংশগুলোর সঙ্গে একটি তথ্য লাগবে তা হলো এই সমান্তরালটি আয়তক্ষেত্রের উপরের মাথা নাকি নিচের মাথা। এখন সুইপিং করার সময় তুমি যদি উপরের মাথা অর্থাৎ শুরুর মাথা পাও তাহলে ট্রিতে কুয়েরি কর যে ওই সীমায় কতগুলো শূন্য আছে অর্থাৎ এই রেখাংশের কত খানি অংশ ঢেকে নেই। সেটি উভরের সঙ্গে যোগ কর। এই কুয়েরি শেষে তোমাকে এই পুরো সীমার সব সংখ্যায় 1 যোগ করে দিতে হবে। আর নিচের মাথা পেলে কী করতে হবে তা বলার দরকার দেখি না। যদি স্থানাঙ্ক খুব বড় হয় তাহলে স্থানাঙ্ক সংকোচন করে নিবে- এই কথাও এক্ষণে ডাল ভাত হয়ে যাওয়ার কথা।

এবার মনে কর তোমাদের $2D$ স্থানাঙ্ক ব্যবস্থায় কিছু বিন্দু দেওয়া আছে এবং কিছু অক্ষের সমান্তরাল আয়তক্ষেত্র দেওয়া আছে। তোমাদের বলতে হবে প্রতিটি আয়তক্ষেত্রের ভেতরে কতগুলো বিন্দু আছে অর্থাৎ প্রতিটি আয়তক্ষেত্রের জন্য আলাদা আলাদাভাবে তোমাকে উভর দিতে হবে। যদি তোমরা ইতোমধ্যেই $2D$ সেগমেন্ট ট্রি এর নাম শুনে থাকো আর ভাব যে ওই কঠিন ডেটা স্ট্রাকচার ব্যবহার করা লাগবে তাহলে ভুল ভেবে থাকবে। এটি আসলে আমাদের একক শেখা পদ্ধতিতেই এই সমাধান করা যাবে। মনে কর আমরা x অক্ষ বরাবর সুইপিং করছি আর y অক্ষ বরাবর সেগমেন্ট ট্রি আছে। যখন আমরা কোনো একটি বিন্দু পাব তখন আমাদের সেগমেন্ট ট্রি তে সেই y এ 1 যোগ করতে হবে। যদি আয়তক্ষেত্রের বামের বাহু পাও তাহলে ওই সীমায় কুয়েরি করে সেই সংখ্যা মনে রাখ। একইভাবে ডান মাথা পেলেও একইভাবে কুয়েরি করে মনে রাখতে হবে। এখন প্রতিটি আয়তক্ষেত্রের জন্য ডানের মাথার জন্য পাওয়া মান থেকে বামের মাথার জন্য পাওয়া মান বিয়োগ করলে আমরা ওই আয়তক্ষেত্রের ভেতরে কতগুলো বিন্দু আছে তা পেয়ে যাব।

মনে কর তোমাদের কিছু বিন্দু দেওয়া হলো আর বলা হলো তুমি $w \times h$ আকারের একটি আয়তক্ষেত্র কে এমনভাবে বসাও যেন সবচেয়ে বেশি সংখ্যক বিন্দু এর ভেতরে থাকে। কীভাবে করবে? সত্যি কথা বলতে এই সমস্যাটা শুনে একটু কঠিন কঠিনই লাগে। কিন্তু আমি যদি বলতাম, তোমাকে $w \times h$ আকারের বেশ কিছু আয়তক্ষেত্র দেওয়া আছে, তোমাকে এমন একটি বিন্দু বের

করতে হবে যা সবচেয়ে বেশি সংখ্যক আয়তক্ষেত্রের ভেতরে থাকে। এই সমস্যা কিন্তু তুলনামূলক সোজা লাগার কথা। কারণ এক্ষেত্রে তুমি সুইপিং করবে এবং সুইপিং এর সময় কোনো আয়তক্ষেত্রের এক মাথা সেগমেন্ট দ্বিতীয় চুকানোর পর দেখবে সেই সীমায় থাকা সব সংখ্যার মধ্যে সর্বোচ্চটি কত। ব্যাস শেষ! এখন কথা হলো মূল সমস্যাকে এই সমস্যায় পরিণত করা যায় কীভাবে? এটাও বেশ সহজ, মনে কর তোমাকে y দিয়ে (x, y) বিন্দু দেয় তাহলে তুমি মনে কর তোমাকে $(x-w, y-h) \sim (x, y)$ আয়তক্ষেত্র দিয়েছে। শেষ! কেন এমন করলাম? কারণ চিন্তা করে দেখ এই আয়তক্ষেত্রের মধ্যে যদি আমরা $w \times h$ আকারের একটি আয়তক্ষেত্রের নিচের বাম কোণা বসাই তাহলে তা ওই বিন্দুকে ঢেকে দেয় বা কাভার করে। বা অন্যভাবে বলা যায় আমরা সেই বিন্দুটি বের করছি যেখানে $w \times h$ আয়তক্ষেত্রের নিচের বামের বিন্দু বসালে সবচেয়ে বেশি বিন্দু পাওয়া যাবে।

কিন্তু উপরের সমস্যায় যদি দুটি নিশ্চেদ (disjoint) আয়তক্ষেত্র নির্বাচন করতে বলত যাতে দুটি দিয়ে ঢেকে দেওয়া বিন্দুর সংখ্যা সবচেয়ে বেশি হয় তাহলে কী করতে? একটি জিনিস খেয়াল কর দুটি আয়তক্ষেত্র যেভাবেই বসাও না কেন তুমি তাদের হয় অনুভূমিকভাবে (horizontally) না হয় উল্লম্বভাবে (vertically) একটি রেখা টেনে আলাদা করতে পারবে। অর্থাৎ তুমি আগের মতো সুইপিং করবে এবং প্রতি x এ লিখে রাখবে যে এই x এ যদি তুমি তোমার আয়তক্ষেত্রের বাম বাহু বসাতে তাহলে তুমি কতগুলো বিন্দু ঢাকতে পারবে। একই ভাবে কোনো x এ যদি তুমি তোমার আয়তক্ষেত্রের ডান বাহু বসাতে তাহলে তুমি কতগুলো বিন্দু ঢাকতে পারবে। এবার এই পাওয়া মানগুলো ব্যবহার করে সহজেই উত্তর বের করে ফেলতে পারবে। যেমন তুমি প্রতি x এ গিয়ে দেখবে এর ডানে বসালে কতগুলো বিন্দু ঢাকা যায় আর বামে বসালে কতগুলো বিন্দু ঢাকা যায়। তুমি চাইলে শুধু "বাম বাহু বসানোতে প্রাপ্ত মান" গুলো ব্যবহার করেই এই উত্তর বের করতে পারতে। কারণ তুমি জানো তোমার আয়তক্ষেত্রের প্রস্থ (width) হলো w . এটা গোল উল্লম্বভাবে ভাগ করার কেইস, কিন্তু যদি তোমাকে অনুভূমিকভাবে ভাগ করতে হয় অপটিমাল উত্তর পাবার জন্য? সহজ, সব বিন্দুর x ও y স্থানাঙ্ক পাল্টাপাল্টি করে দিয়ে উপরের কাজ আবার কর এবং দুটি উত্তর থেকে যেটি বেশি ভালো সেটি হবে তোমার আসল উত্তর। চাইলে বাম-ডানের কাজ তোমরা উপর-নিচে করতে পারতে।

১০.৩.৭ কিছু স্থানাঙ্ক সম্পর্কিত গণনা

মনে কর একটি $n \times n$ গ্রিড (অর্থাৎ প্রতি পাশে n টি করে ল্যাটিস বিন্দু থাকবে) দিয়ে বলা হলো এতে কতগুলো বর্গক্ষেত্র আঁকা যায় যেন এর সব শীর্ষ একটি ল্যাটিস বিন্দু হয়। খেয়াল কর, এখানে কিন্তু বলা হয়নি বর্গগুলো অক্ষের সমান্তরাল হবে। সুতরাং $(1, 0), (0, 1), (-1, 0), (0, -1)$ ও একটি বৈধ বর্গ। এই ধরনের সমস্যার কৌশল হলো তোমাকে একটি সীমানির্দেশকারী বাস্তু (bounding box) ধরতে হবে এবং এর ভেতরে ঠিক ঠিক আঁটে করে এরকম কতগুলো বর্গ পাওয়া সম্ভব তা বের করতে হবে। এর পর এই আকারের সীমানির্দেশকারী বাস্তু পুরো গ্রিডে কতগুলো থাকতে পারে তা বের করে হিসাবনিকাশ করলেই আমাদের উত্তর বের হয়ে আসবে। ধরা যাক আমরা বের করতে চাই $m \times m$ গ্রিডে আঁটে করে এরকম কতগুলো বর্গ আছে? উত্তর সহজ, $m-1$ টি। যদি তুমি $(i, 0)$ কে একটি শীর্ষ ধর তাহলে $(m-1, i), (m-1-i, m-1), (0, m-1-i)$

হবে উপর তিন শীর্ষ। এভাবে $i = 0 \dots m - 2$ এর জন্য তুমি একটি করে বর্গ পাবে যা $m \times m$ গ্রিডে আঁটে। একটু চিন্তা করে দেখতে পার কোনো $p \times q$ গ্রিডে কোনো বর্গ আঁটে না, সুতরাং তোমাকে $m = 1 \dots n$ এর জন্য $m \times m$ গ্রিড বিবেচনা করতে হবে এবং প্রতিটিতে কতগুলো করে বর্গ সম্ভব তা বের করতে হবে। এখন প্রশ্ন হলো $n \times n$ গ্রিডে কতগুলো $m \times m$ সাব-গ্রিড আছে? সহজ $(n - m + 1) \times (n - m + 1)$ টি। বাকিটুকু কিছু সাধারণ গণিত। তুমি চাইলে এই পুরো হিসাবনিকাশ $O(1)$ এ করতে পার।

একই রকম আরও একটি সমস্যা দেখা যাক। আগের মতোই তোমাদের $n \times m$ আকারের একটি গ্রিড দেওয়া আছে ($n, m \leq 1000$)। তোমাদের বের করতে হবে এর ভেতরে কতগুলো ত্রিভুজ আছে? এবার সমস্যাটি একটু কঠিন লাগার কথা। যেহেতু সর্বমোট $n \times m$ টি বিন্দু আছে সুতরাং আমরা $\binom{nm}{3}$ ভাবে তিনটি বিন্দু নির্বাচন করতে পারি। সমস্যা হলো যদি তিনটি বিন্দু একই রেখায় থাকে তাহলে সেটি বৈধ ত্রিভুজ হবে না। এখন কতভাবে তিনটি বিন্দু নির্বাচন করা যায় বেন যেকোনো রেখার একটি সীমানির্দেশকারী বাস্তু থাকবে। সুতরাং আমাদের সীমানির্দেশকারী বাস্তুর তারা সরলরেখায় থাকে এই সংখ্যা বের করতে পারলে আমাদের সমাধান হয়ে যাবে। প্রথমতো আকারের উপর একটি লুপ চালাতে হবে। এখন যদি বাস্তুর কোনো একটি মাত্রা যদি 1 হয় (মানে সংখ্যাটি খুব সহজেই বের করে ফেলতে পারব (সংখ্যাটি হলো 1 তাই না?))। যদি মাত্রা 1 না হয় তাহলে অবশ্যই রেখাটি কোনো একটি কর্ণ বরাবর থাকবে। যেহেতু আয়তক্ষেত্রে দুটি কর্ণ আছে সুতরাং একটি কর্ণের জন্য উত্তর বের করে দুই দিয়ে গুণ করে দিলেই হয়ে যাবে। আর আমরা কিছুক্ষণ আগেই দেখেছি একটি কর্ণের জন্য উত্তর আমরা $G.S.A.G.$ ব্যবহার করে বের করে ফেলতে পারি।

যদি সামান্তরিক (parallelogram) বলত? তাও সহজ। দুই ধরনের কেইস হতে পারে। চার কোণা সীমানির্দেশকারী বাস্তুর চার বাহুর উপরে, অথবা দুই কোণা সীমানির্দেশকারী বাস্তুর দুই কোণার উপর। যদি সামান্তরিকের চার কোণা বাস্তুর চার বাহুতে থাকে তাহলে একটি প্যাটার্ন (pattern) দেখতে পারবে। সেটি হলো- মনে কর সীমানির্দেশকারী বাস্তুর উপরের বাহুতে সামান্তরিকের যেই শীর্ষ আছে তা বাহুকে $a : b$ অনুপাতে বিভক্ত করে তাহলে নিচের বাহুকে নিচের শীর্ষ $b : a$ অনুপাতে বিভক্ত করবে। একই কথা ডান ও বামের বাহুর উপরেও খাটবে। কিন্তু ডান-বামের অনুপাত উপর-নিচের অনুপাতের উপর নির্ভর করে না। সুতরাং আমরা এই প্যাটার্নে একটি (w, h) আকারে আয়তক্ষেত্র থেকে প্রায় $(h - 1) \times (w - 1)$ টি সামান্তরিক পাব, আমি প্রায় শব্দ বললাম কারণ কিছু corner কেইস থাকতে পারে, মনে কর সামান্তরিকের চার কোণা সীমানির্দেশকারী বাস্তুর চার কোণায় পরল ইত্যাদি। তোমাদের মূল ধারনা দেখানোই আমার মূল উদ্দেশ্য। এরপর বাকিটুকু তোমরা কাগজ কলম নিয়ে বসে করে ফেলতে পারবে। যাই হোক, এখন যদি সামান্তরিকের দুই শীর্ষ যদি সীমানির্দেশকারী বাস্তুর বিপরীত দুই শীর্ষে থাকে তাহলে? তাহলে ওই দুই কোণা দিয়ে যেই কর্ণ যায় সেটি বাদে যেকোনো বিন্দুকে একটি শীর্ষ হিসেবে পছন্দ করলে বাকি শীর্ষ এমনিই পাওয়া যাবে। অর্থাৎ এখানে আমাদের $G.S.A.G.$ এর সুত্র খাটাতে হবে (দুইবার গণনা বা double counting যেন না হয় তা খেয়াল রাখবে)। একটু সাবধানে এই দুই কেইস সামলালেই তোমরা পুরো উত্তর পেয়ে যাবে যা $O(n^2 \log n)$ এ বের হয়ে যাবে।

১০.৮ প্রোগ্রামিং সমস্যা

১০.৮.১ অনুশীলনী

সহজ

:: UvaLive 6784 :: UvaLive 6835 :: UvaLive 6839 :: UvaLive 6859 :: UvaLive
6890 :: UvaLive 6925 :: UvaLive 6942 :: UvaLive 6955** :: UvaLive 6967
:: UvaLive 6989 :: UvaLive 7004 :: UvaLive 7020 :: UvaLive 7066* :: UvaLive
7166

সামান্য কঠিন

:: UvaLive 6826 :: UvaLive 6845*** :: UvaLive 6846 :: UvaLive 6951*
:: UvaLive 6956 :: UvaLive 7069 :: UvaLive 7070 :: UvaLive 7072

কঠিন

:: UvaLive 6785 :: UvaLive 6841*** :: UvaLive 6950 :: UvaLive 7022**

অধ্যায় ১১

স্ট্রিং (String) সম্পর্কিত ডেটা স্ট্রাকচার ও অ্যালগরিদম

১১.১ হ্যাশিং (Hashing)

যদিও হ্যাশিং (hashing) ঠিক স্ট্রিং সম্পর্কিত টেকনিক না কিন্তু হ্যাশিং মনে হয় স্ট্রিংয়ের সমস্যাতে বেশি ব্যবহার হয়ে থাকে। হ্যাশিংয়ের মূল ধারনা হলো তোমাকে একটি জিনিস দিবে, তোমার কাজ হলো এই জিনিসকে একটি সংখ্যাতে পরিবর্তন করা। তবে এই পরিবর্তনের পদ্ধতি এমন হতে হবে যেন এই জিনিসটি যত বারই দিক না কেন, আমাদের সংখ্যা বা হ্যাশ মান (hash value) যেন একই হয়। এই যে সংখ্যায় পরিবর্তন করার যেই পদ্ধতি একে হ্যাশিং বলে। এখন কীভাবে কোনো জিনিসকে একটি সংখ্যায় রূপান্তর করবে তার কোনো নির্দিষ্ট উপায় নেই। তুমি যেভাবে খুশি পরিবর্তন করতে পারো। তবে এই পরিবর্তনের উপর তুমি কী পেতে চাচ্ছ তা অনেক পরিমাণ নির্ভর করে। সাধারণত আমাদের প্রবলেমগুলো এরকম হয়ে থাকে, তোমাকে কিছু জিনিস দেওয়া হতে থাকবে এবং তোমাকে মাঝে মধ্যে জিজ্ঞাসা করা হবে অমুক জিনিস তোমার কাছে আছে কিনা। এজন্য তোমাকে কিছু লিস্ট নিতে হবে, এদেরকে বাকেট (bucket) বলা হয়। মনে কর তোমার কাছে n টি বাকেট আছে। এখন তুমি যা করবে তা হলো যখন তোমাকে সংরক্ষণের জন্য কোনো জিনিস দিবে তুমি তাকে হ্যাশিং করে সেই হ্যাশ মানকে n দিয়ে mod কর এবং সেই বাকেটে গিয়ে এই জিনিস রেখে দাও। এবার তোমাকে যখন কোনো কুয়েরি (query) দেওয়া হবে তখন সেই কুয়েরির হ্যাশ মান বের করে সেই বাকেটে যাও এবং সেই বাকেটের জিনিসগুলো একে একে যাচাই করে দেখ যে তাদের কোনোটি তোমার জিনিস কিনা। যদি তুমি n কে অনেক বড় নাও এবং তোমার হ্যাশিং ফাংশন যদি ভালো হয় তাহলে খুব কম খুঁজেই তুমি তোমার কুয়েরির উপর দিতে পারবে। ধরা যাক আমি বললাম যে আমি তোমাকে কিছু 1000 অক্ষের সংখ্যা দিব আর মাঝে মধ্যে জিজ্ঞাসা করব আমি তোমাকে অমুক সংখ্যা দিয়েছিলাম কিনা। তুমি মনে করলে আচ্ছা আমার হ্যাশিং ফাংশন (hashing function) হবে প্রদত্ত সংখ্যার অঙ্কগুলোর যোগফল। তাহলে

কিন্তু এটি খুব একটি ভালো হ্যাশিং ফাংশন হবে না। প্রথমত এক্ষেত্রে হ্যাশিং ফাংশনের সর্বোচ্চ মান হবে $1000 \times 9 = 9000$ কিন্তু মনে কর তোমার বাকেট আছে 100000 টি। সুতরাং বুঝতেই পারছ যে এই হ্যাশিং ফাংশনের ক্ষেত্রে অনেক বাকেট অব্যবহৃত থাকবে। তাহলে তোমার ব্যবহৃত বাকেটগুলোতে গড়ে বেশি বেশি সংখ্যা থাকবে আর এতে তোমার কুয়েরি সময়ও বাঢ়বে। তাহলে কী করা যায়? গুণফল? হ্যাঁ গুণফল অনেক বড় সংখ্যা হবে, একে mod করলে আমরা 100000 টি বাকেটই ব্যবহার করতে পারব। কিন্তু খেয়াল কর যদি আমাদের সংখ্যার কোনো একটি অঙ্ক 0 হয় তাহলে সেটি সবসময় 0 তম বাকেটে পড়বে। এখন তোমাদের ইনপুটে যেসব সংখ্যা দেওয়া থাকবে মনে কর তাদের সবারই 0 অঙ্ক আছে। তাহলে তোমার কুয়েরি করতে অনেক বেশি সময় লাগবে তাই না? সুতরাং এই হ্যাশিং ফাংশনও খুব একটি ভালো না।

তাহলে ভালো ফাংশন কেমন হয়? আগেই বলেছি হ্যাশিং ফাংশন তোমার ইচ্ছা মতো করতে পারো তবে উপরের দুটি জিনিস খেয়াল রাখতে হবে, অনেক সংখ্যা যেন একটি বাকেটে গিয়ে জড়ে না হয় আর সব বাকেট যেন সমানভাবে ব্যবহার হয়। এই দুটি দিক খেয়াল করে একটি বহুল ব্যবহৃত হ্যাশিং ফাংশন হলো পলিনোমিয়াল হ্যাশিং ফাংশন (polynomial hashing function)। একটি পলিনোমিয়াল (polynomial) দেখতে কেমন হয় তা তো জানো? $a_0 + a_1x^1 + a_2x^2 + \dots$ মনে কর $a_0, a_1 \dots$ এগুলো হলো তোমাকে দেওয়া সংখ্যার অঙ্ক বা তোমাকে দেওয়া জিনিসের ক্ষুদ্র ক্ষুদ্র অংশ। যেমন একটি স্ট্রিংয়ের ক্ষেত্রে এর প্রতিটি ক্যারেক্টর (character) এর আসকি (ascii) মান। আর x হিসেবে একটি মৌলিক সংখ্যা নেওয়া ভালো। এখন তুমি এই হ্যাশের মান বের কর। আর n টি বাকেটে ছড়িয়ে দেওয়ার জন্য এই মানকে n দিয়ে mod কর। সাধারণত এই n কেও অন্য কোনো মৌলিক সংখ্যা নেওয়া হয়ে থাকে। এটি একটি ভালো হ্যাশিং ফাংশন বলা চলে। তাহলে তুমি হ্যাশ মান বের করার পর সেই বাকেটে গিয়ে তোমার জিনিস সংরক্ষণ করবে এবং কোনো জিনিস খুঁজতে বললে তুমি তার হ্যাশ মানের বাকেটে গিয়ে প্রতিটির সঙ্গে তুলনা করে দেখবে তোমার সংখ্যা এখানে আছে কিনা। যদি অনেকগুলো বাকেট নাও তাহলে এই খোঁজার পরিমাণ অনেক অনেক কমে যাবে। এটিই হলো হ্যাশিং।

এখন অনেক সময় সংখ্যা না দিয়ে একটি সেট দিয়ে বলে যে এই সেটটি আগে এসেছিল কিনা। অর্থাৎ $\{1, 2\}$ আর $\{2, 1\}$ কে একই জিনিস হিসেবে বিবেচনা করতে হবে। এক্ষেত্রে যেটি করলে ভালো হয় তা হলো তুমি তোমার সেটের উপাদানগুলোকে সংজীবিত করে নাও। এর পর একে একে হ্যাশিং কর। বা তুমি সেটের উপাদানগুলোকে আলাদা আলাদাভাবে হ্যাশিং করে এর পর তাদের যোগ কর বা xor কর। কারণ এতে ক্রম বা অর্ডার (order) কোনো ব্যাপার হয় না। এভাবে প্রবলেম ভেদে টুকটাক কৌশল খাটিয়ে হ্যাশিং করলে ভালো ফল পাবে।

অনেক সময় দুটি বড় স্ট্রিং দিয়ে বলা হয় একটি আরেকটির ভেতরে সাব-স্ট্রিং (substring) আকারে আছে কিনা (সাব-সিকোয়েন্স (subsequence) না কিন্তু)। ধরা যাক আমাদেরকে বলা হয়েছে S এর ভেতরে T খুঁজতে। স্বাভাবিক ধারনা হলো তুমি S এর প্রতিটি জায়গায় গিয়ে $|T|$ পরিমাণ সাব-স্ট্রিং নিয়ে তাকে হ্যাশিং করে T এর হ্যাশের সঙ্গে তুলনা করা। কিন্তু এতে $|S| \times |T|$ সময় লেগে যাবে। এর থেকে ভালো উপায় আছে। খেয়াল কর S এর 0 থেকে $|T|$ সাইজের সাব-স্ট্রিংয়ের পলিনোমিয়াল হ্যাশ কেমন? $H_0 = s_0 + s_1P^1 + \dots s_{n-1}P^{n-1}$ তাই না? আবার 1 থেকে? $H_1 = s_1 + s_2P^1 + \dots s_nP^{n-1}$. এই দুটি সংখ্যার পার্থক্য কিন্তু খুব একটি বেশি না।

আমরা কিন্তু লিখতে পারি $H_0 = H_1 \times P + s_0 - s_n P^n$. অর্থাৎ আমরা যদি H_1 জানি তাহলে খুব সহজে H_0 বের করে ফেলতে পারি। এর মানে আমরা পেছন থেকে হ্যাশিং ফাংশন এর ভ্যালু বের করতে থাকলে খুব কম সময়েই সব জায়গার হ্যাশ মান বের করতে পারি। অথবা তুমি চাইলে পলিনোমিয়ালকে উল্টিয়ে দিতে পারো অর্থাৎ $a_0 P^{n-1} + a_1 P^{n-2} + \dots$ তাহলে তুমি সামনে থেকেই যেতে পারবে। তাহলে এভাবে তোমার S এর সব জায়গার জন্য হ্যাশ মান বের করতে সময় লাগবে মাত্র $|S|$ সময়। আবার একটি জিনিস থেয়াল কর এখানে T কিন্তু নির্দিষ্ট, তাই একে কিন্তু কোনো একটি বাকেটে ফেলা জরুরি না। তুমি চাইলে mod না করেই এই কাজ করতে পারো, মানে তুমি long বা long long ডেটা টাইপে যা হিসাব করার করবে। ওভারফ্লো (overflow) হলে হবে, এসব নিয়ে মাথা ব্যাথা করতে হবে না। কারণ একই জিনিসকে যদি তুমি একইভাবে হ্যাশিং কর তাহলে ওভারফ্লো হয়ে একই সংখ্যা হবে।

১১.২ নুথ-মরিস-প্র্যাট (Knuth-Morris-Pratt) বা KMP অ্যালগরিদম

আমরা কিছুক্ষণ আগে একটি স্ট্রিংয়ের ভেতর আরেকটি স্ট্রিং, সাব-স্ট্রিং আকারে আছে কিনা তা বের করলাম। তবে সমস্যা হলো আমরা জানি না আগের পদ্ধতিতে কত সময় লাগবে। মানে আমরা আশা করতে পারি যে এটি লিনিয়ার (linear) বা $O(n)$ সময় নিবে তবে এটি যে সবসময় $O(n)$ সময় নিবে তার কোনো ঠিক নেই। হয়তো তোমার হ্যাশ পদ্ধতির উপর ভিত্তি করে এমন একটি ইনপুট দেওয়া সম্ভব যেখানে অনেক সময় নিবে। কিন্তু আমরা এই সমস্যাকে হ্যাশিং ছাড়া $O(n)$ সময়ে সমাধান করতে পারি। এ জন্য বহুল প্রচলিত অ্যালগরিদম হলো KMP বা নুথ-মরিস-প্র্যাট (Knuth-Morris-Pratt) অ্যালগরিদম। এটি একটু কঠিন অ্যালগরিদম। অ্যালগরিদমটি খুব ছোট কিন্তু এটি বোৰা বিশেষ করে এটি কেন $O(n)$ সময়ে কাজ করে তা বোৰা একটু কষ্টকর।

মনে কর আমরা কোনো একটি স্ট্রিং T (Text) এর মধ্যে একটি স্ট্রিং P (Pattern) আছে কিনা তা বের করতে চাই। এজন্য আমাদের প্রথমে P এর প্রিফিক্স ফাংশন (prefix function) বের করতে হবে। আশা করি তোমাদের মনে আছে যে প্রিফিক্স (prefix) কী বা সাফিক্স (suffix) কী। প্রিফিক্স হলো কোনো স্ট্রিং এর শুরুর অংশ আর সাফিক্স হলো শেষের অংশ। যেমন $xiox$ একটি স্ট্রিং হলে কোনো স্ট্রিং এর শুরুর অংশ আর সাফিক্স হলো $\{xiox, iox, ox, x\}$. প্রোপার প্রিফিক্স (Proper prefix) হলো ওই স্ট্রিং বাদে ওই স্ট্রিংয়ের অন্যান্য প্রিফিক্স। যেমন এই স্ট্রিংের জন্য প্রোপার প্রিফিক্স হলো $\{x, xi, xio, xiox\}$ আর সাফিক্স হবে $\{xiox, iox, ox, x\}$. প্রোপার প্রিফিক্স (Proper prefix) হলো ওই স্ট্রিং বাদে ওই স্ট্রিংয়ের অন্যান্য প্রিফিক্স। যেমন এই প্রিফিক্সের "দৈর্ঘ্য" যেন তা তার সাফিক্সও হয়। যেমন যদি $P[0\dots i]$ হয় $aabcaaab$ তাহলে 3 দৈর্ঘ্যের প্রোপার প্রিফিক্স পাওয়া যাবে যা সাফিক্সও এবং এটি হলো aab । এখানে প্রোপার প্রিফিক্স প্রিফিক্সের "দৈর্ঘ্য" যেন তা তার সাফিক্সও হয়। যেমন যদি $P[0\dots i]$ হয় $aabcaaab$ তাহলে 3 দৈর্ঘ্যের প্রোপার প্রিফিক্স পাওয়া যাবে যা সাফিক্সও এবং এটি সাফিক্সও প্রিফিক্সও। বলার কারণ হলো আমি তো চাইলে পুরো স্ট্রিং নিয়ে বলতে পারতাম যে এটি সাফিক্সও প্রিফিক্সও। কিন্তু এটি আমরা চাই না, এজন্য আমি বলেছি প্রোপার প্রিফিক্স। তাহলে আমরা একটি স্ট্রিংয়ের সব হানের জন্য প্রিফিক্স ফাংশনের মান বের করি।

index	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
π	-1	-1	0	1	2	-1	0

কিছুক্ষণ আগে বলা প্রিফিক্স ফাংশনের সংজ্ঞার সঙ্গে দেখবে টেবিল ১১.১ এর π এর মানের মিল নেই। কিন্তু খুব বেশি যে অমিল তাও কিন্তু না। এটি সংজ্ঞা মোতাবেক মানের থেকে এক কম। আসলে আমরা এখানে স্ট্রিংটিকে $0 - index$ হিসেবে বিবেচনা করেছি এবং $\pi(i)$ এর মান এমন হবে যেন $P[0 \dots \pi(i)] = P[i - \pi(i) + 1 \dots i]$. অর্থাৎ π কে আমরা ঠিক দৈর্ঘ্য দিয়ে না বরং ইনডেক্স (index) দিয়ে সংজ্ঞায়িত করব। এই টেবিল এ $\pi(0) = \pi(1) = -1$ কেন তা একটু বলা দরকার। কারণ হলো আমরা মনে করতে পারি যে $P[0 \dots -1]$ হলো একটি ফাঁকা স্ট্রিং এবং যেহেতু a বা ab এর এমন কোনো প্রোপার প্রিফিক্স নেই যা সাফিক্সও তাই আমরা ধরে নেই যে ফাঁকা স্ট্রিং হবে এই প্রিফিক্স বা সাফিক্স। আসলে সত্য বলতে এটি বলার জন্য বলা, -1 না দিলে পরবর্তী অংশ কোড করতে ঝামেলা হবে বা যদি আমরা $0 - index$ না মনে করে যদি $1 - index$ নিতাম তাহলে পুরো জিনিস অনেক সহজ হতো এবং এখানে -1 না হয়ে 0 হতো। যাই হোক, তুমি যখন পুরো অ্যালগরিদমটি বুঝে যাবে তখন এসব কেন কী করছি তাও বুঝতে পারবে তাই এসব নিয়ে এখন অত বেশি চিন্তা করার কিছু নেই। এখন তাহলে টেবিল ১১.১ এর π এর মানে একবার চোখ বুলিয়ে নাও। দেখ বাদ বাকি সব মান ঠিক আছে কিনা। এখন আমাদের প্রশ্ন হলো এই π এর সব মান কীভাবে আমরা লিনিয়ার সময়ে বের করতে পারি।

প্রথমত $\pi(0) = -1$. এখন তুমি সর্বশেষ π এর মানকে একটি ভ্যারিয়েবলে নাও। ধরা যাক ভ্যারিয়েবলটি *now*. এখন 1 হতে $|P| - 1$ পর্যন্ত i এর একটি লুপ চালাও। আমরা $\pi(i)$ বের করতে চাই। এজন্য আমাদের যা করতে হবে তা হলো $P[now + 1]$ এবং $P[i]$ সমান কিনা তা দেখতে হবে। যদি না হয় তাহলে $now = \pi(now)$ করব। আর যদি সমান হয় বা $now = -1$ হয় তাহলে এসব না করে এই লুপ থেকে বের হতে হবে। তবে আমাদের আবারও $P[now + 1]$ এবং $P[i]$ তুলনা করব এবং যদি সমান হয় তাহলে now কে এক বাড়াব এবং $\pi(i)$ এ এই মান রাখব। আর যদি সমান না হয় তাহলে $now = \pi(i) = -1$ করব। এতক্ষণ যা বললাম তা এক রকম অ্যালগরিদমের বর্ণনা। কিন্তু আমাদের বুঝতে হবে কেন আমরা এরকম করছি।

আমরা প্রথমে terminal কেইস $now = -1$ নিয়ে চিন্তা না করে একটি সাধারণ কেইস (general case) নিয়ে চিন্তা করে দেখি। মনে কর কোনো এক i এর জন্য $\pi(i)$ জানি, এখন আমরা বের করতে চাই $\pi(i + 1)$ এর মান। $\pi(i)$ মানে কী? এর মানে হলো $P[0 \dots \pi(i)] = P[i - \pi(i) + 1 \dots i]$ এবং এরকম সবগুলোর মধ্যে $\pi(i)$ সর্বোচ্চ (অবশ্যই i এর থেকে ছোট)। এখন এরকম আমরা সবচেয়ে বড় $\pi(i + 1)$ বের করতে চাই। খেয়াল কর যদি $P[\pi(i) + 1] = P[i + 1]$ হতো তাহলে আমরা খুব সহজেই বলতে পারতাম যে $\pi(i + 1) = \pi(i) + 1$ কারণ এর থেকে বড় কিন্তু হওয়া সম্ভব না, হলে $\pi(i)$ আরও বড় হওয়া সম্ভব হতো তাই না? বা অন্যভাবে

বলতে হলে বলতে হয় $\pi(i+1) - 1$ কিন্তু $\pi(i)$ এর একটি candidate. সুতরাং আমরা যদি দেখি $P[\pi(i)+1] = P[i+1]$ তাহলে $\pi(i+1) = \pi(i) + 1$. এখন প্রশ্ন হলো যদি না হয়? আমাদের লক্ষ্য $P[0 \dots i+1]$ এর একটি সাফিক্স বের করা যা প্রিফিক্সও বা অন্যভাবে বলতে হলে বলা যায় $P[0 \dots i]$ এর একটি সাফিক্স বের করা যা প্রিফিক্সও এবং সেই প্রিফিক্সের পরের ক্যারেটার $P[i+1]$ এর সমান। আমরা একটি প্রিফিক্স ইতোমধ্যেই চেষ্টা করেছি আর তা হলো $P[0 \dots \pi(i)]$. এর পরের বড় candidate সাফিক্স কোনটি হবে? সেটি কিন্তু ইতোমধ্যেই বের করে রেখেছ আর তা হলো $\pi(\pi(i))$. কেন? খেয়াল কর আমরা চাই $\pi(i)$ এর থেকে ছোট সাফিক্স যা প্রিফিক্সও। ধরা যাক এরকম কোনো একটি সাফিক্স বা প্রিফিক্স হলো Z . এই Z এর বৈশিষ্ট্য হলো এটি একই সঙ্গে $P[0 \dots \pi(i)]$ এর প্রিফিক্স এবং সাফিক্স। আসলে এদের মধ্যে সবচেয়ে বড়টি আমাদের দরকার। আর সেটিই কিন্তু $\pi(\pi(i))$ তাই না? উদাহরণ দেওয়া যাক। মনে কর আমরা $ababa$ এর জন্য $\pi(4)$ জানি আর তা হলো 2. এখন মনে কর আমাদের পরের ক্যারেটার হলো c যা $P[3]$ এর সমান না। তাই আমাদের aba এর থেকে ছোট এমন একটি স্ট্রিং দরকার যা $ababa$ এর একই সঙ্গে সাফিক্স ও প্রিফিক্স। এবং সেটি কিন্তু আবার aba এরও প্রিফিক্স ও সাফিক্স। তাই আমরা যদি $\pi(2)$ দেখি তাহলে a পাব যা $ababa$ এর প্রিফিক্স ও সাফিক্স। তোমরা যদি এটুকু বুঝে থাকো তাহলে আশা করি কোড ১১.১ ও বুঝতে পারবে। একটি জিনিস বলা হয়নি তা হলো আমরা এতক্ষণ সাধারণ কেইস নিয়ে চিন্তা করেছি। Terminal কেইস নিয়ে বলা হয়নি। খেয়াল কর কিছুক্ষণ আগের উদাহরণে যখন আমরা দেখে যে $P[1]$ ও c না তখন আমরা আবার $\pi(0)$ করব আর এক্ষেত্রে আমরা পাব -1. এখন প্রথম কথা হলো এই লুপ আজীবন চলতে পারে না, এক সময় আমাদের শেষ করতে হবে আর এছাড়াও তোমরা $\pi(-1)$ কল করতে পারবে না কারণ এটি বলে কিছু নেই, তাই যখন $now = -1$ হয়ে গেছে তখন আমরা লুপ থেকে বের হয়ে গেছি। তবে এই লুপ থেকে বের হওয়ার দুটি মানে আছে এক $now + 1$ এর সঙ্গে মিলে গেছে দুই মিলে নি মানে -1 হবে। এটি যাচাই করার জন্যই আমাদের এই লুপের শেষে একটি if-else আছে।

কোড ১১.১: prefix function.cpp

```

1 int pi[100];
2 char P[100];
3
4
5 int prefixFunction() {
6     int now;
7     pi[0] = now = -1;
8     int len = strlen(P);
9     for (int i = 1; i < len; i++) {
10        while (now != -1 && P[now + 1] != P[i])
11            now = pi[now];
12        if (P[now + 1] == P[i]) pi[i] = ++now;
13    }
14}

```

```

12         else pi[i] = now = -1;
13     }
14 }

```

তবে এখনো আমাদের ম্যাচিং (matching) সমস্যার সমাধান হয়নি। আমরা কেবলমাত্র আমাদের প্যাটার্ন (pattern) অর্থাৎ P এর প্রিফিক্স ফাংশন বের করলাম। এখন আমাদের কাজ হলো P কি T এর ভেতর আছে কিনা তা বের করা। এজন্য আমরা কিছুটা আগের মতোই কাজ করব। প্রথমে $now = -1$ নাও এবং T এর উপর দিয়ে 0 হতে $|T| - 1$ পর্যন্ত একটি লুপ চালাও। i এর লুপে যখন ঢুকবে তখন now নির্দেশ করবে $T[0 \dots i - 1]$ এর দীর্ঘতম সাফিক্স যা P এর প্রিফিক্স। এখন আমাদের বিবেচনা করতে হবে $T[i]$. আগের মতো প্রথমে দেখো যে $P[now + 1]$ কি $T[i]$ এর সমান কিনা। হলে তো now কে এক বাড়িয়ে দিবে। আর যদি না হয় তাহলে $now = \pi(now)$ করবে এবং আবারও একই জিনিস যাচাই করবে। আর যদি $now = -1$ হয়ে যায় তাহলে কী করতে হবে তা তো বুঝতেই পারছ। কোড ১১.২ এ আমরা এটি কোড করে দেখালাম।

কোড ১১.২: kmp.cpp

```

1 int pi[100];
2 char P[100], T[100];
3
4 int kmp() {
5     int now;
6     now = -1;
7     int n = strlen(T);
8     int m = strlen(P);
9     for (int i = 0; i < n; i++) {
10        while (now != -1 && P[now + 1] != T[i])
11            now = pi[now];
12        if (P[now + 1] == T[i]) ++now;
13        else now = -1;
14        if (now == m) return 1;
15    }
16    return 0;
17 }

```

এখন কথা হলো এর টাইম কমপ্লেক্সিটি কত? দুটি লুপ দেখে যদি ভাব এটি $O(n^2)$ তাহলে ভুল ভাববে। খেয়াল কর `for` লুপের একটি ইটারেশন (iteration) এ now এর মান খুব জোড়

এক বারে। আবার while লুপে now এর মান কখনও বাড়ে না, শুধুই কমে। তাই while লুপ আসলে সর্বমোট লিনিয়ার সময় চলে। অর্থাৎ আমাদের প্রিফিক্স ফাংশন বের করার কোড সময় নেয় $O(|P|)$ আর ম্যাচিংয়ের কোড সময় নেয় $O(|T|)$.

১১.২.১ KMP সম্পর্কিত কিছু সমস্যা

ধরা যাক P কি T এর মধ্যে আছে কিনা শুধু এটাই জিজ্ঞাসা করেনি বরং কত বার আছে তাও জানতে চেয়েছে। তুমি কী করবে? একটি সহজ বুদ্ধি হলো $P \# T$ এর প্রিফিক্স ফাংশন (একে ফেইলিউর ফাংশন বা ইংরেজীতে failure function ও বলে) বের করা। এখানে # হলো এমন একটি ক্যারেক্টার যা P বা T কারও ভেতরে নেই। তাহলে প্রিফিক্স ফাংশনে যতবার $|P|$ আসবে সেটিই তোমার উত্তর। এছাড়াও আরেকটি উপায় হলো উপরে আমরা যখন P কে T এর ভেতরে খুঁজেছিলাম তখন যে now = $|P|$ হলেই 1 রিটার্ন করেছিলাম তা না করে আমরা তখন একটি counter এর মান বাড়াবো এবং now = $\pi(\text{now})$ কল করব।

আরেকটি সমস্যা এরকম হতে পারে যে আমাদের শুধু P কতবার পাওয়া গেছে তাই জানতে চায়নি বরং P এর সব প্রিফিক্স কতবার T তে আছে তা জানতে চাওয়া হয়েছে। কীভাবে করবে? যা করতে হবে তা হলো T তে P খুঁজার while লুপের শেষে প্রতিবার একটি অ্যারেতে now তম স্থানের মান এক বাড়াতে হবে। অর্থাৎ আমরা now পর্যন্ত প্রিফিক্স পেয়েছি এটি বোঝাতে। কিন্তু শুধু এটি করলে কিন্তু হবে না। উদাহরণস্বরূপ $P = aa$ মনে কর আর $T = aaaaa$. এখন এটি তো বুবেছই যে প্রায় সবসময় আমরা now = 1 এর মান বাড়াব কিন্তু now = 0 এর মান কিন্তু তেমন বাড়বে না যদিও প্রিফিক্স a বহুবার T তে দেখা যায়। সুতরাং আমাদের যা করতে হবে তা হলো T তে P খুঁজার কাজ শেষ হলে, P এর শেষ থেকে শুরুতে লুপ চালাতে হবে। এবং প্রতি i এর জন্য $\text{count}[\pi(i)]$ এর মান $\text{count}[i]$ পরিমাণ বাড়াতে হবে। কেন? কারণ হলো i এ শেষ হওয়া স্বচেয়ে বড় সাফিক্স যা প্রিফিক্সও তাতে কিন্তু তুমি আরও $\text{count}[i]$ বার যেতে পারতে যা আগে হিসাব করনি।

ধর একটি স্ট্রিং দিয়ে বলা হলো এতে কয়টি স্বতন্ত্র সাব-স্ট্রিং (distinct substring) আছে। কীভাবে করা যায়? মনে কর স্ট্রিংটি হলো S এবং এর প্রিফিক্স ফাংশন আমরা জানি। এ থেকে আমরা স্বচেয়ে বড় মান k বের করলাম। এর মানে কী? এর মানে হলো এই স্ট্রিংয়ের $k + 1$ হতে $|S|$ দৈর্ঘ্যের যে প্রিফিক্স তা এই স্ট্রিংয়ে আর কথাও আসেনি। কিন্তু $S[1]$ থেকে যেসব অনন্য (unique) সাব-স্ট্রিং আছে সেসব কীভাবে বের করব? সহজ, $S[1 \dots]$ এর জন্য আবার প্রিফিক্স ফাংশন বের কর। এভাবে S এর প্রতিটি সাফিক্সের জন্য অনন্য (unique) প্রিফিক্সের সংখ্যা যোগ করলে আমরা মোট স্বতন্ত্র সাব-স্ট্রিং এর সংখ্যা পেয়ে যাব। এজন্য আমাদের সময় লাগছে $O(n^2)$.

একটি স্ট্রিং S দেওয়া আছে বলতে হবে এমন কোনো ছোট স্ট্রিং P আছে কিনা যাকে বার বার পুনরাবৃত্তি করলে S পাওয়া যায়। যেমন $S = ababab$ এখানে $P = ab$ কে তিনবার পর পর বসালে S পাওয়া যায়। এর সমাধান হলো দেখতে হবে যে $n - \pi(n - 1) + 1$ কি n কে ভাগ করে কিনা। করলে সেই দৈর্ঘ্যের প্রিফিক্সই হবে উত্তর। এর প্রমাণ একটু কঠিন। তোমরা একটু চিন্তাভাবনা করে proof by contradiction পদ্ধতিতে এই জিনিস প্রমাণ করতে পার।

১১.৩ Z অ্যালগরিদম

KMP তে যেমন প্রিফিক্স ফাংশন ছিল এখানে আছে 'Z' ফাংশন। $z(i)$ এর মানে হলো স্ট্রিংয়ের i তম ইনডেক্স হতে শুরু করে কত বড় সাব-স্ট্রিং পাওয়া যায় যা মূল স্ট্রিংয়ের প্রিফিক্স। যেমন $ababc$ এই স্ট্রিংয়ের জন্য 'Z' ফাংশনের মানগুলো হবে $\{0, 0, 2, 0, 0\}$. এখানে $z(0)$ এর মান 0 করা হয়েছে। কারণ কিছুটা KMP এর $\pi(0)$ এর মতো। আর তাছাড়া এটি আমাদের কোড সহজ করবে।

এখন আসা যাক এটি কীভাবে বের করা যায় সেই প্রশ্নে। অবশ্যই $O(n^2)$ সময়ে বের করা কোনো ব্যাপার না। আমরা চাই লিনিয়ার সময়ে একটি স্ট্রিং S এর 'Z' ফাংশন বের করতে। আমাদের এজন্য দুটি ভ্যারিয়েবলের দরকার একটি হলো $left$ এবং আরেকটি হলো $right$. প্রথমে আমরা $z(0) = left = right = 0$ সেট করব। এখন আমরা $i = 1$ হতে $|S| - 1$ পর্যন্ত একটি লুপ চালাব। লুপের ভেতর কী করব তা জানার আগে $left$ আর $right$ এর মানে জানলে ভালো হয়। লুপের i তম অবস্থানে এই দুটি ভ্যারিয়েবল প্রকাশ করে যে $[0, i - 1]$ এই সীমার কোন মান x এর জন্য $x + z(x)$ এর মান সর্বোচ্চ হয়। অর্থাৎ $S[left \dots right]$ সাব-স্ট্রিংটি S এর একটি প্রিফিক্স হবে এবং $0 < left < i$ হবে আর এরকম সব **candidate** এর মধ্যে $right$ সর্বোচ্চ হয়। অর্থাৎ প্রতিবার লুপের ভেতরে $z(i)$ এর মান বের হয়ে গেলে আমাদের দেখতে হবে যে $i + z(i) - 1$ কি $right$ এর থেকে বেশি কিনা। বেশি হলে $left = i, right = i + z(i) - 1$ করতে হবে।

এখন কথা হলো এই $z(i)$ এর মান কীভাবে বের করা যায়। প্রথমে আমরা দেখব যে $i \leq right$ কিনা। হলো আমরা $z(i)$ এর মানকে এক লাফে অনেক দূর বাড়াতে পারব। কী রকম? খেয়াল কর $S[left \dots right]$ হলো S এর একটি প্রিফিক্স বা $S[0 \dots (right - left)]$ এর সমান। যেহেতু $left < i$ এবং $i \leq right$? তাহলে এটি বলা যায় যে $S[i \dots right]$ হলো $S[(i - left) \dots (right - left)]$ এর সমান। এবং যেহেতু $i - left < i$ তাই আমরা কিন্তু $z(i - left)$ এর মান আগে থেকেই জানি। এবং আমরা বলতে পারি যে $z(i)$ এর মান কিছুটা $z(i - left)$ এর মতো হবে। কারণ ওই যে বললাম $S[i \dots right]$ হলো $S[(i - left) \dots (right - left)]$ এর সমান। এখন খেয়াল কর আমরা সরাসরি $z(i) = z(i - left)$ করতে পারব না। কেন? কারণ আর যাই হোক $i + z(i) - 1 > right$ হতে পারবে না কারণ আমরা $S[right]$ এর পরের তথ্য জানি না। তাই যা করতে হবে তা হলো $z(i) = \min(z(i - left), right - i + 1)$ করতে হবে। এটি $z(i)$ এর একটি **lower limit**. অর্থাৎ $z(i)$ কমপক্ষে এত হবেই। এর থেকে বড় হবে কিনা তা নিশ্চিত না। তাই যা করতে হবে তা হলো একটি লুপ চালিয়ে আরও বড় করা যায় কিনা তা দেখতে হবে। এভাবে $z(i)$ এর মান বের করতে হবে। এর কোড ১১.৩ এ দেওয়া হলো।

কোড ১১.৩: zfunction.cpp

```
1 char S[100];
2 int z[100];
3
```

```

8 void zfunction() {
9     int left, right;
10    z[0] = left = right = 0;
11    int len = strlen(S);
12    for (int i = 1; i < len; i++) {
13        if (i <= right)
14            z[i] = min(z[i - left], z[right - i + 1]);
15        while (i + z[i] < len
16                && S[i + z[i]] == S[z[i]])
17            z[i]++;
18        if (i + z[i] - 1 > right)
19            left = i, right = i + z[i] - 1;
20    }
21 }

```

উদাহরণ দেখা যাক, মনে কর $ababab$ এর 'z' ফাংশন বের করছি, আমরা $z(2) = 4$ বের করে ফেলার পর $z(4)$ এর মান কিন্তু সেই $z(2)$ এর তথ্য থেকে 2 পেরে যাব। কীভাবে? দেখ 4 হলো $left = 2$ আর $right = 2 + 4 - 1 = 5$ এর ভেতরে। সুতরাং আমরা $z(4) = \min(z(2) = 4, 5 - 4 + 1 = 2) = 2$ করব। এর পর আর পরের while লুপ চলবে না কারণ $4 + z(4) < len$ না। এর মানে $z(4)$ বের করতে আমাদের কোনো কাজই করতে হচ্ছে না।

এখন কথা হলো এটি কেন লিনিয়ার? খেয়াল কর for লুপের ভেতরে যেই if আছে সেখানে যদি $z(i) = right - left + 1$ দিয়ে bound হয় অর্থাৎ i হতে শুরু করা স্ট্রিংটা $right$ এ গিয়ে আটকে যায়, তাহলে আমরা while লুপ দিয়ে $right$ এর মান বাড়ানোর চেষ্টা করি। আর যদি $right$ এর আগেই bound হয়ে যায় তাহলে কিন্তু এই while লুপের সমতার শর্ত সত্য হবে না। তাই কোনো কাজ না করেই এই লুপ শেষ হয়ে যাবে। অর্থাৎ এই while লুপ কিন্তু সবসময় $right$ এর মান বাড়াবে। আর কতই বা বাড়াবে? $len = |S|$ এর থেকে তো আর বড় না তাই না? তাই এটি লিনিয়ার।

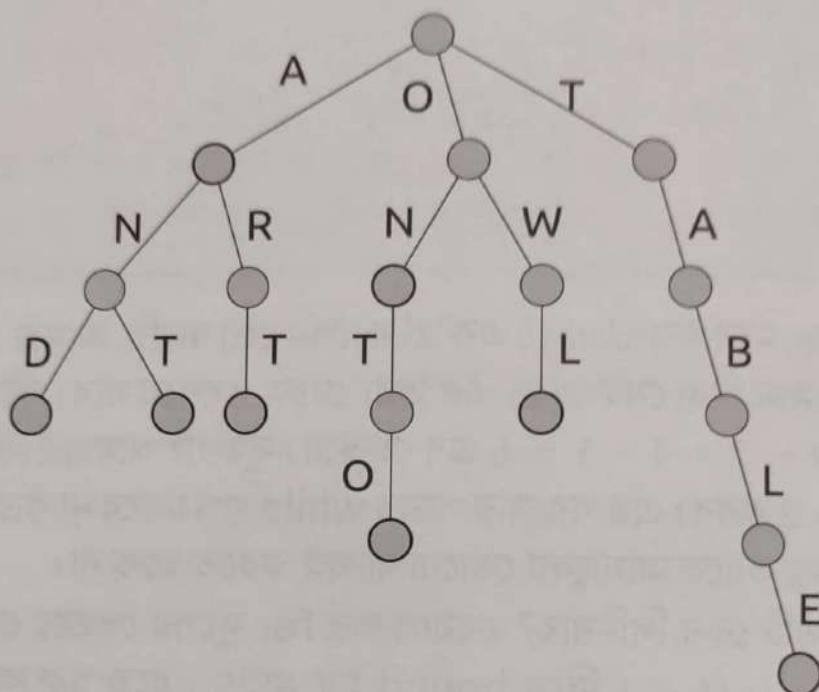
11.3.1 Z অ্যালগরিদম সম্পর্কিত কিছু সমস্যা

এখন এর মাধ্যমে কী কী সমস্যা সমাধান করা যায়? KMP দিয়ে সমাধান করা যায় এরকম বেশির ভাগ সমস্যাই সমাধান করা সম্ভব। যেমন P কি T এর ভেতর আছে কিনা? এটি সমাধানের জন্য $S = P + \# + T$ করে দেখ যে কতগুলো $z(i)$ এর মান $|P|$ তাহলেই হয়ে গেল। একইভাবে একটি স্ট্রিংয়ে কতগুলো স্বতন্ত্র সাব-স্ট্রিং আছে তাও বের করা সহজ। একটি স্ট্রিং নিয়ে তার 'z' ফাংশনের মান বের কর। ধরা যাক সর্বোচ্চ মান x . এর মানে $S[1\dots]$ এ আমরা কখনো এক সময় $S[0\dots x-1]$ এই সাব-স্ট্রিং পাব। সুতরাং এই সাব-স্ট্রিং নিয়ে এখন না চিন্তা করে

$S[0 \dots (x \dots |S| - 1)]$ এই সাব-স্ট্রিংগুলো নিয়ে চিন্তা করব। অর্থাৎ এখন আমাদের প্রত্যন্ত সাব-স্ট্রিংয়ের *count* ($|S| - 1$) - $x + 1$ বাড়াব। আর এর পরে $S[1 \dots]$ নিয়ে চিন্তা করব। এভাবে $|S|$ বার S এর $|S|$ টি সাফিক্স নিয়ে কাজ করব।

১১.৪ ট্রাই (Trie)

এটি কথা বলে বোঝানোর থেকে ছবিতে বোঝানো সহজ। চিত্র ১১.১ এ {a, and, ant, art, on, onto, owl, table} শব্দসমূহের জন্য ট্রাই এঁকে দেখানো হলো।



নকশা ১১.১: {a, and, ant, art, on, onto, owl, table} শব্দসমূহের জন্য ট্রাই

চিত্র থেকে খুব সহজেই বোঝা যাওয়ার কথা ট্রাই আসলে কী। এটি ট্রি (tree) আকারে তোমাকে দেওয়া সব শব্দকে উপস্থাপন করে। সাধারণ প্রিফিক্সের জন্য একটিই মাত্র নোড থাকে। যেমন উপরের উদাহরণ এ *ant* আর *art* দুটির জন্যই *a* নোড দুটি একই। আমরা বেশি কথা না বলে সরাসরি কোডে চলে যাব এবং আসলে এই কোড ব্যাখ্যা করারও কিছু নেই। আশা করি তোমরা কোড ১১.৪ দেখে বুঝতে পারবে আমরা কী করছি এবং কেন করছি। আর তোমাদেরকে যদি বলা হয়- আচ্ছা অনুক শব্দ এই ট্রাইয়ে আছে কিনা- তাও আশা করি বের করতে পারবে! তাও বলি রুট থেকে ট্রাভার্স (traverse) শুরু করবে আর দেখবে বর্তমান নোডে বর্তমান ক্যারেক্টার দিয়ে লেবেল করা বাহু (edge) আছে কিনা না থাকলে সেই শব্দ নেই। আর থাকলে এভাবে এগোতে থাকতে হবে। শেষে গিয়ে দেখতে হবে শেষের নোডে *isWord* এ মার্ক করা আছে কিনা।

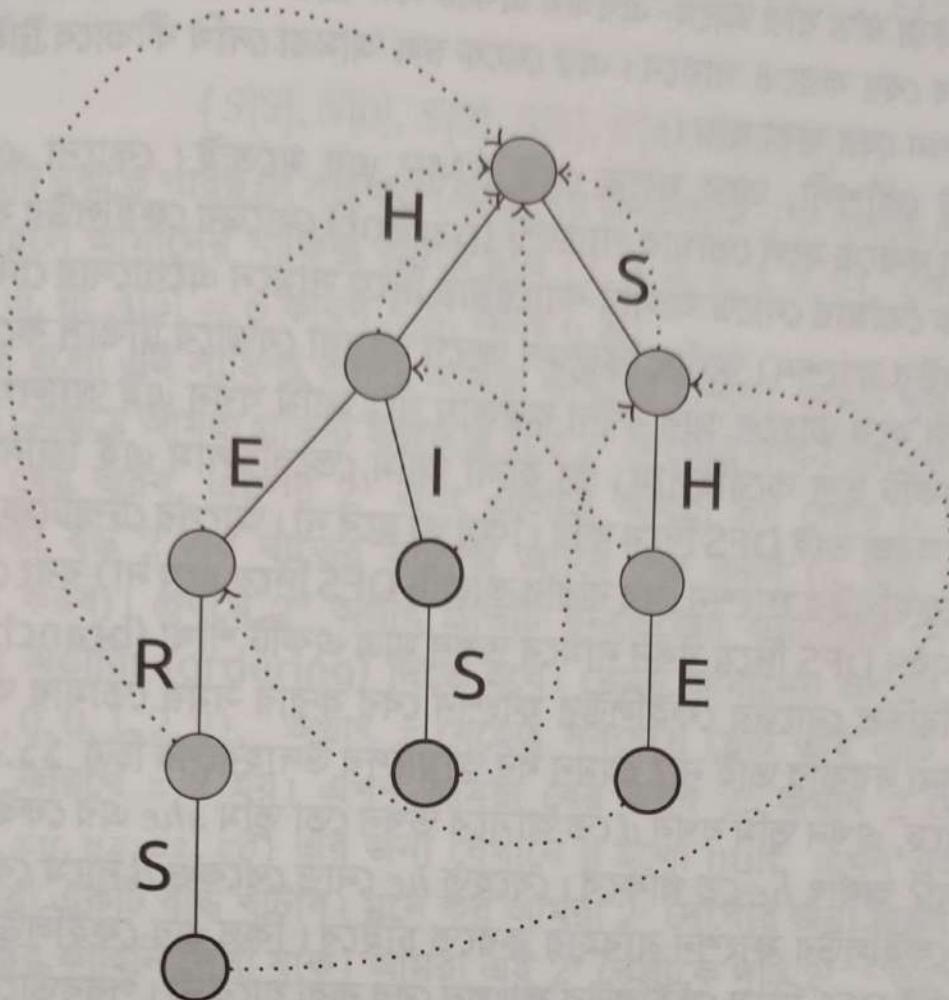
কোড ১১.৮: trie.cpp

```

1 #define MAX_NODE 100000
2 #define MAX_LEN 100
3
4 char S[MAX_LEN];
5 // assuming words only have small letters ['a', 'z']
6 int node[MAX_NODE][26];
7 int root, nnnode;
8 int isWord[MAX_NODE];
9
10 // call before inserting any words into trie.
11 void initialize() {
12     root = 0;
13     nnnode = 0;
14     for (int i = 0; i < 26; i++)
15         // -1 means no edge for ('a' + i)th
16         // character
17         node[root][i] = -1;
18 }
19
20 void insert() {
21     scanf("%s", S);
22     int len = strlen(S);
23     int now = root;
24     for (int i = 0; i < len; i++) {
25         if (node[now][S[i] - 'a'] == -1) {
26             node[now][S[i] - 'a'] = ++nnnode;
27             for (int j = 0; j < 26; j++)
28                 node[nnnode][j] = -1;
29         }
30         now = node[now][S[i] - 'a'];
31     }
32     // mark that a word ended at this node.
33     isWord[now] = 1;
34 }
```

আমরা KMP বা 'z' ফাংশন দিয়ে একটি প্যাটার্ন এবং একটি টেক্সট (text) এর জন্য লিনিয়ার সময়ে সমাধান করেছিলাম। যদি আমাদের একটি প্যাটার্ন আর অনেকগুলো টেক্সট থাকত তাহলে প্যাটার্নের জন্য একবার ফেইলিউর ফাংশন বা 'z' ফাংশন বের করে আমরা প্রতিটি টেক্সটকে আলাদা আলাদা করে সমাধান করতে পারতাম। খেয়াল কর যেহেতু প্যাটার্ন নির্দিষ্টই থাকছে সেহেতু ফেইলিউর ফাংশন (বা প্রিফিক্স ফাংশন) কিন্তু বার বার বের করার দরকার নেই। একইভাবে 'z' ফাংশনের ক্ষেত্রেও একই কথা প্রযোজ্য। সুতরাং এক্ষেত্রেও আমরা লিনিয়ার সময়ে সমাধান করছি। তবে যদি প্যাটার্ন একাধিক হয় আর টেক্সট একটি (আসলে একাধিক টেক্সটের জন্যও কাজ করবে) তাহলে লিনিয়ার সময়ে সমাধান করতে আমাদের আহো-কোরাসিক অ্যালগরিদম (Aho-Corasick Algorithm) লাগবে। মনে কর আমাদের প্যাটার্নের স্ট্রিংগুলো হলো *hers, hi, his, she.* তাহলে আমাদের চিত্র ১১.২ এর মতো একটি ডেটা স্ট্রাকচার বানাতে হবে। এই ছবির ডটেড লাইনগুলো বাদ দিলে এটি আসলে একটি ট্রাই ম্যাট্রিক্স। এখন বুঝতে হবে ডটেড লাইনের মাহাত্ম্য কী! এগুলো হলো প্রতিটি নোডের ফেইলিউর ফাংশন। অর্থাৎ রুট হতে কোনো নোড পর্যন্ত যেই স্ট্রিংটি হয় তার সর্বোচ্চ দৈর্ঘ্যের কোন সাফিক্সটি (অবশ্যই প্রোপার সাফিক্স বা ইংরেজীতে proper suffix) ট্রাইয়ে আছে? ফেইলিউর ফাংশনটি সেই নোডকে পয়েন্ট করবে। একটি উদাহরণ দেখা যাক। মনে কর *she.* এর প্রোপার সাফিক্সগুলো হলো *he* এবং *e.* এদের মধ্যে *e* শব্দটি ট্রাইয়ে নেই। মানে তুমি যদি রুট থেকে ট্রাভার্স শুরু কর তাহলে এই শব্দ পাবে না। এখানে শব্দ পাবে না মানে ট্রাভার্স করে খোঁজার সময়ে শেষে গিয়ে *isWord* দেখার প্রয়োজন নেই, নোড থাকলেই হবে। যাই হোক, তাহলে *e* বলে কিছু নেই এই ট্রাইয়ে। এখন *he* দেখা যাক, এবং হ্যাঁ *he* আছে। আর এটিই আসলে সর্বোচ্চ দৈর্ঘ্যের প্রোপার সাফিক্স যার জন্য ট্রাইয়ে নোড আছে। সুতরাং *she* নোডের জন্য ফেইলিউর ফাংশন *he* কে পয়েন্ট করবে। এভাবে প্রতিটি নোডের জন্য ফেইলিউর ফাংশন বের করতে হবে। আর হ্যাঁ, যদি কোনো সাফিক্স না পাও এর মানে মনে করবে ফাঁকা স্ট্রিং (empty string) ও একটি সাফিক্স, আর তাই ফেইলিউর ফাংশনকে ফাঁকা স্ট্রিং বা রুটে পয়েন্ট করাবে। যেমন *h* এর নোড থেকে আমরা রুটে পয়েন্ট করেছি।

এখন একটি টেক্সট দিয়ে যদি বলে কোন কোন প্যাটার্ন এই টেক্সটের মধ্যে আছে বা কতগুলো প্যাটার্ন আছে তাহলে কীভাবে সমাধান করবে? খুব সহজ, একদম KMP এর মতো। রুট হতে শুরু কর। টেক্সটের ক্যারেক্টার দেখ আর সেই অনুসারে সামনে এগোতে থাক। যদি দেখ তুমি এখন যেই নোডে আছো সেখানে সেই ক্যারেক্টারের কোনো বাহু নেই তাহলে ফেইলিউর ফাংশনের বাহু দিয়ে পিছিয়ে গিয়ে সেখানে দেখবে। সেখানে না থাকলে আবার ফেইলিউর ফাংশনের বাহু দিয়ে পিছিয়ে গিয়ে দেখবে। এভাবে করতে থাকবে যতক্ষণ না রুটে যাও। যদি রুটে আসো তাহলে কী করতে হবে তা তো আমরা KMP তেই দেখেছি তাই না? এটি আসলে KMP এরই একটি বর্ধিত রূপ। এটি বুঝতে হলে তোমাদের KMP ভালোমতো বুঝতে হবে। যাই হোক, এখন কথা হলো কীভাবে বুঝবে যে কোন কোন প্যাটার্ন এই টেক্সটে আছে? তোমরা ভাবতে পার যে এত এত গুরুত্বপূর্ণ



নকশা ১১.২: {hers, hi, his, she} শব্দসমূহের জন্য আহো-কোরাসিক ট্রাই (Aho-Corasick trie)

জিনিস না বলে বাদ দিয়ে গেলাম আর বললাম KMP থেকে দেখে নিও আর এই সহজ জিনিস নিয়ে আমি গুঁতাগুঁতি করছি! "এটি তো অনেক সহজ, টেক্সট দিয়ে ট্রাইয়ে ট্রাভার্স করার সময়ে যেসব শব্দ নোড দিয়ে ট্রাভার্স করব সেসব প্যাটার্ন এই টেক্সটে আছে!" না এটি ঠিক না। মনে কর আমাদের উদাহরণে s ও একটি প্যাটার্ন। আর তোমাকে দেওয়া টেক্সট হলো *his*. তাহলে ট্রাভার্স করার সময় কেবলমাত্র *h*, *hi* আর *his* এই তিনটি নোড ট্রাভার্স করা হবে। অর্থাৎ তুমি মাত্র *his* প্যাটার্নটি পেয়েছ। কিন্তু s শব্দের নোড কিন্তু ট্রাভার্স করনি। তাহলে কী করতে হবে? যা করতে হবে তা হলো, পুরো টেক্সটটি ট্রাভার্স করা হয়ে গেলে তোমাকে দেখতে হবে কোন কোন প্যাটার্নের নোড *visited* হয়েছে। তাদেরকে একটি BFS এর জন্য কিউ (queue) তে রাখতে হবে। এবার একে একে নোড এই কিউ থেকে তোল আর এর ফেইলিউর ফাংশনের নোডকে কিউতে ঢেকাও যদি না সেই নোডকে আগেই কিউতে ঢেকানো হয়, অর্থাৎ যদি ওই নোড আগেই *visited* না হয় আরকি। আমরা আসলে একরকমের BFS করছি যেখানে বাহু হলো ফেইলিউর ফাংশনের বাহু। আর সোর্স (source) নোড হলো যেসব প্যাটার্নকে আমরা টেক্সটের ট্রাভার্স করার সময় *visit* করেছি সেসব। এই BFS শেষে

দেখতে হবে কোন কোন প্যাটার্নের নোড আমরা *visit* করছি। সেসবই হলো আমাদের উত্তর। কেন? কারণটি বেশ সহজ আসলে, আমরা KMP তেও এরকম একটি জিনিস দেখে এসেছিলাম যখন আমরা কোন প্রিফিক্স কত বার আছে- এরকম একটি সমস্যার কথা বলেছিলাম। সুতরাং এটুকু তোমরা নিজেরাই ভেবে বের করতে পারবে। এর থেকে চল আমরা দেখি কীভাবে ট্রাইয়ে লিনিয়ার সময়ে ফেইলিউর ফাংশন বের করা যায়।

এই অংশটি একটু কৌশলী, তবে মূলত সেই KMP এর মতোই। কোনো একটি নোডের ফেইলিউর ফাংশন বের করতে হলে তোমার প্যারেন্ট (parent) নোডের ফেইলিউর ফাংশনে যেতে হবে এবং সেখান থেকে তোমার নোডে আসার ক্যারেন্টার দিয়ে সামনে এগোনোর চেষ্টা করতে হবে না হলে আবার ফেইলিউর ফাংশন। অর্থাৎ কিছুক্ষণ আগে আমরা যেভাবে ট্রাভার্স করেছি ঠিক সেই রকম। এত সহজই যদি হবে তাহলে আর বর্ণনা করতাম না। আমি যখন এই অ্যালগরিদমটি শিখি তখন এই জায়গায় একটি ভুল করেছিলাম। তা হলো আমি ভেবেছিলাম এই জিনিস DFS দিয়ে কোড করা তো অনেক সহজ তাই DFS দিয়ে করি। কিন্তু তা হবে না। আগের টেক্সটকে ট্রাভার্স করাটা DFS দিয়ে হবে কিন্তু ফেইলিউর ফাংশন বের করার কাজটি DFS দিয়ে হবে না। বরং তোমাকে BFS করতে হবে। কেন? কারণ DFS দিয়ে যখন নামবে তখন মাত্র একটি শাখা (branch) ধরে নামতে থাকবে। এই শাখার বিভিন্ন নোডের ফেইলিউর ফাংশন বের করার সময় তোমার অন্য নোডেরও ফেইলিউর ফাংশন জানা দরকার তাই না? যেমন ধর আমাদের উদাহরণের চিত্র ১১.২ এ ধর *shed* নামে একটি শব্দও আছে, এখন তুমি যখন d তে আসবে তখন তো তুমি *she* এর ফেইলিউর ফাংশন ব্যবহার করবে তাই না? অর্থাৎ *he* তে থাকবে। যেহেতু *he* নোড থেকেও d নামে কোনো বাহু বের হয়নি তাই তুমি এর ফেইলিউর ফাংশন ব্যবহার করতে চাইবে। কিন্তু এর ফেইলিউর ফাংশন তো এখনো বের করনি। তাই DFS দিয়ে ফেইলিউর ফাংশন বের করা যাবে না। পরবর্তীতে নাসা ভাইয়া আমাকে এই রকম একটি কেইস দিলে আমি বুঝতে পারি যে আমাদের আসলে BFS করতে হবে। BFS করলে হবে কারণ ফেইলিউর ফাংশন সবসময় উপরের নোড অর্থাৎ কম গভীরতার নোডে পয়েন্ট করে, আর BFS নিশ্চিত করে যে বেশি গভীরতার নোডের ফেইলিউর ফাংশন বের করার আগে কম গভীরতার নোডগুলোর ফেইলিউর ফাংশন বের হয়ে যাবে। এভাবে আমরা সম্পূর্ণ কাজটি লিনিয়ার সময়ে করতে পারি।

১১.৬ সাফিক্স অ্যারে (Suffix Array)

একটি স্ট্রিং S দেওয়া থাকবে, তোমাকে এর সাফিক্স অ্যারে (suffix array) A বের করতে হবে। $A[i]$ মানে কী? এর মানে হলো S এর সব সাফিক্সকে সর্ট করলে $S[i \dots |S| - 1]$ এই সাফিক্সটি তাদের মধ্যে $A[i]$ তম সাফিক্স হয়। খেয়াল কর এখানে সর্ট বলতে ডিকশনারি ক্রম (dictionary order) বা লেখিকোগ্রাফিকাল ক্রম (lexicographical order) বোঝানো হচ্ছে আর যেহেতু সব সাফিক্স আলাদা তাই সর্ট করতে কষ্ট হওয়ার কথা না। একটি উদাহরণ দেখা যাক। ধরা যাক $S = xyxyyzz$. তাহলে এর সাফিক্সগুলো হচ্ছে

$$\{z, zz, yzz, xyzz, xxyz, yxxyzz, xyxyyzz\}$$

এবং এদের সর্ট করলে দাঁড়ায়

$$\{xxyz, xyxyzz, xyz, yxyz, yzz, z, zz\}$$

বা

$$\{S[2], S[0], S[3], S[1], S[4], S[6], S[5]\}$$

আশা করি এটি বুঝতে পারছ যে সংক্ষেপে প্রকাশের জন্য আমি $S[i]$ বলতে $S[i\dots]$ এই সাফিল্কে বুঝিয়েছি। তাহলে আমাদের সাফিল্ক অ্যারে হবে $\{1, 3, 0, 2, 4, 6, 5\}$. কেন? $A[0] = 1$ কারণ $S[0]$ আছে 1 এ, বা $A[5] = 6$ কারণ $S[5]$ আছে 6 এ।

এখন কথা হলো এই সাফিল্ক অ্যারে আমরা কীভাবে বানাব? একটি খুব সহজ $O(n \log^2 n)$ সমাধান আছে। প্রথমে আমরা প্রতিটি ইনডেক্স হতে 2^0 দৈর্ঘ্যের সাব-স্ট্রিংের জন্য সাফিল্ক অ্যারের মতো জিনিস বের করব, এরপর 2^1 এবং এভাবে আমরা সব শেষে n এর সমান বা n এর থেকে নিকটবর্তী বড় 2 এর ঘাতের সাফিল্ক অ্যারে বের করব (n যদি 6 হয় তাহলে আমরা 2^3 পর্যন্ত বের করব)। প্রথমে 2^0 অর্থাৎ প্রতিটি ইনডেক্সের ক্যারেন্টার দেখে তাদের সর্টিংয়ের একটি ক্রম বা অর্ডারিং (ordering) দিতে হবে। যেমন আমাদের আগের উদাহরণ $xyxyzz$ টি হবে $\{0, 1, 0, 0, 1, 2, 2\}$. অর্থাৎ x যেহেতু সবচেয়ে ছোট তাই এটি 0, y এর পরে তাই এটি 1 একই কারণে z 2 হয়। এখন আমরা বের করব 2^1 অর্থাৎ 2 দৈর্ঘ্যের জন্য। অর্থাৎ $\{xy, yx, xx, xy, yz, zz, z0\}$ এর জন্য যেখানে 0 হলো null। এখন এই জিনিস সরাসরি না বের করে আমরা একটি বুদ্ধি খাটাব। মনে কর আমরা 2^j দৈর্ঘ্যের জন্য অর্ডারিং বের করব। এখন i তম ইনডেক্সের কাহিনি দেখা যাক। আমরা এই 2^j দৈর্ঘ্যকে দুটি 2^{j-1} ভাগে ভাগ করতে পারি। একটি হলো $[i, i + 2^{j-1} - 1]$ আর আরেকটি হলো $[i + 2^{j-1}, i + 2^j - 1]$. এখন এই দুটি অংশের 2^{j-1} দৈর্ঘ্যের অর্ডারিং কিন্তু আমরা জানি। তাহলে আমরা চাইলে একটি 2^j দৈর্ঘ্যকে স্ট্রিং দিয়ে প্রকাশ না করে দুটি সংখ্যামান দিয়ে প্রকাশ করতে পারি (a, b) যেখানে a হলো $[i, i + 2^{j-1} - 1]$ এর অর্ডারিং আর b হলো $[i + 2^{j-1}, i + 2^j - 1]$ এর অর্ডারিং। এভাবে আমরা প্রতিটি ইনডেক্সের জন্য দুটি সংখ্যামান পাব। তোমরা হয়তো ভাবতে পার যদি $i + 2^{j-1} \geq |S|$ হয় তাহলে ওই সংখ্যামান কোথায় পাব? সেক্ষেত্রে তুমি -1 ধরে নিতে পার, কারণ কোনো জায়গায় কোনো ক্যারেন্টার থাকার থেকে আমরা না থাকাকে বেশি গুরুত্ব দেই। তাহলে আমরা সব জায়গার জন্য আমরা জোড়া জোড়া সংখ্যামান পেয়েছি। এখন এদের সর্ট কর এবং এদের ছোট থেকে বড় ক্রমে এদের ইনডেক্সকে সংখ্যামান দিবে। সমান সংখ্যা জোড়াকে অবশ্যই একই সংখ্যামান দিবে। যেমন আমাদের উদাহরণে 2^1 এর ক্ষেত্রে আমাদের সাব-স্ট্রিংগুলো হলো $\{xy, yx, xx, xy, yz, zz, z0\}$ বা $\{(0, 1), (1, 0), (0, 0), (0, 1), (1, 2), (2, 2), (2, -1)\}$. এখন তোমাকে এদের সর্ট করতে হবে এবং সবচেয়ে ছোট জোড়া এবং তার সমান জোড়াকে তুমি সংখ্যামান দিবে 0, এর থেকে বড়কে 1 এরকম। এই সংখ্যামান কিন্তু তাদের ইনডেক্সকে দেবে। অর্থাৎ ধর আমাদের উপরের অ্যারেকে সর্ট করলে সবার আগে $(0, 0)$ আসবে। এর মানে 2 ইনডেক্স 0 পাবে। এর পর আসবে $(0, 1)$ যা 0 ও 3 অবস্থানে আছে। তাই এই দুই অবস্থানে 1 বসবে এভাবে তুমি 2^1 এর জন্য অর্ডারিং পাবে। তাহলে এই অর্ডারিং অ্যারে দেখতে এরকম হবে: $\{1, 2, 0, 1, 3, 5, 4\}$. এরকম করে তোমরা আশা করি 2^2 আর 2^3 এর জন্য অর্ডারিং পেয়ে যাবে। যেহেতু $2^3 \geq |S|$ সুতরাং 2^3 এর জন্য পাওয়া অর্ডারিংই

হলো সাফিক্স অ্যারে। বাকি দুটি ধাপ নিজেরা করে দেখো দেখি {1, 3, 0, 2, 4, 6, 5} পাও কিনা। যেহেতু আমরা $\log n$ বার স্টিং করছি তাই আমাদের কমপ্লেক্সিটি হলো $O(n \log^2 n)$.

এখন যেহেতু কখনো আমাদের অর্ডারিংয়ের সংখ্যামান n কে অতিক্রম করে না, তাই আমরা চাইলে এখানে কাউন্টিং সর্ট (counting sort) করতে পারি। জোড়া সংখ্যামানের ক্ষেত্রে কাউন্টিং সর্ট একটু কৌশলী এবং কীভাবে করে আমরা সেটি দেখব না। যাদের ইচ্ছা আছে তোমরা নিজেরা ভেবে বের করতে পার। খুব একটা কঠিন হওয়া উচিত না। সাধারণত আমি নিজেও এটি হাতে হাতে কোড করি না। হাতে হাতে কোড করা লাগলে আগের মতো করি। আর আমার লাইব্রেরীতে এই কাউন্টিং সর্ট দিয়ে $O(n \log n)$ এর কোড তোলা আছে। তাই দরকারের সময় সেটি ব্যবহার করি আমি।

তোমাদের যদি ইচ্ছা থাকে তাহলে সাফিক্স অ্যারে বের করার একটি লিনিয়ার অ্যালগরিদম আছে। সেটিও শিখে ফেলতে পার। যদিও আমি নিজে কখনো ব্যবহার করি নাই, তাই ঠিক বলতে পারছি না সেখানে কীভাবে করেছে। তবে এটুকু জানি ওখানে divide and conquer টেকনিক ব্যবহার করে করা হয়েছে, কিছুটা লিনিয়ার সময়ে সিলেকশন (selection) অ্যালগরিদম যেভাবে করা হয় সেরকম।

১১.৬.১ সাফিক্স অ্যারে সম্পর্কিত কিছু সমস্যা

সাফিক্স অ্যারে এর উপর একটি সুন্দর ডকুমেন্ট^১ আছে যেটি যতদূর সম্ভব দুইজন রোমানীয় তৈরি করেছে। আমি সেখান এর সমস্যাগুলোই একে একে তুলে ধরার চেষ্টা করব। তোমরা যদি পার তাহলে এই ডকুমেন্টটি একটু পড়ে দেখো। হয়তো কিছু নতুন জিনিস শিখবে।

আমাদের আলোচনার সুবিধার জন্য আমরা ধরে নেই $\lceil \log n \rceil = k$. আমাদের স্ট্রিংটি হলো S আর তার সাফিক্স অ্যারে থাকবে A তে। অর্থাৎ $A[i]$ বলে $S[i \dots]$ কত তম সাফিক্স। এছাড়াও আমরা আরও একটি অ্যারে বিবেচনা করব $rank$. $rank[A[i]] = i$ হবে অর্থাৎ $S[rank[i] \dots]$ হলো সর্টেড সাফিক্স লিস্টে i তম সাফিক্স।

আমাদের প্রথম সমস্যা হলো একটি স্ট্রিং S দেওয়া আছে। তোমাকে দুটি ইনডেক্স i এবং j দিয়ে বলা হলো এই দুটি অবস্থান থেকে শুরু করে যেই দুটি স্ট্রিং পাওয়া যায় তাদের দীর্ঘতম সাধারণ প্রিফিক্স বা longest common prefix (LCP) কত? অর্থাৎ যেমন $xxyyxyyz$ এই উদাহরণে যদি 0 আর 4 এই দুটি অবস্থান দিয়ে জিজ্ঞাসা করত তাহলে আমাদের উত্তর হবে 3. বুঝতেই পারছ আমরা কুয়েরি খুব দ্রুত করতে চাচ্ছি। প্রথমে আমাদের সাফিক্স অ্যারে বের করতে হবে। আমরা এখানে দুটি পদ্ধতির কথা বলব। প্রথম পদ্ধতির জন্য আমাদের 0 হতে k সব ধাপের জন্য অর্ডারিংয়ের অ্যারে লাগবে। এই পদ্ধতিটি হলো কিছুটা ট্রিতে দুটি নোডের LCA বের করার মতো। আমরা এই দুটি অবস্থানের $k - 1$ তম অ্যারেতে অর্ডারিং দেখব। যদি সমান হয় তাহলে আমরা আমাদের উত্তরের সঙ্গে 2^{k-1} যোগ করে নিব আর ইনডেক্স দুটিকে আমরা এই পরিমাণ বাড়িয়ে নিব। এর পর আমরা $k - 2$ নিয়ে যাচাই করব এবং একই কাজ করব। এভাবে আমরা $k - 1$ হতে 0 পর্যন্ত কাজ করলেই প্রতি কুয়েরিতে $O(\log n)$ সময়ে আমরা LCP বের করে ফেলতে পারব।

¹<http://web.stanford.edu/class/cs97si/suffix-array.pdf>

আরেকটি পজিশনে $Z[0] = lcp(rank[0], rank[1])$, $Z[1] = (rank[1], rank[2])$ এরকম স্টেড সাফিক্সগুলোর পাশাপাশি স্ট্রিংগুলোর LCP বের করা। এখন তোমাদের i আর j নিয়ে যদি কুয়েরি করে তাহলে, $Z[min(A[i], A[j]) \dots max(A[i], A[j]) - 1]$ এই সীমার সর্বনিম্ন বের করলেই তোমরা $S[i\dots]$ আর $S[j\dots]$ এর LCP পেয়ে যাবে। এখন পাশাপাশি এরকম জোড়া থাকবে $|S| - 1$ টি। অর্থাৎ i তম সাফিক্সটি কোথায় আছে সেটা দেখতে হবে ($A[i]$), j তম সাফিক্সটি কোথায় আছে সেটা দেখতে হবে ($A[j]$)। এবার আমাদের এদের মধ্যে সব Z এর সর্বনিম্ন মানই হবে i ও j তম সাফিক্সের LCP। আমরা চাইলে $O(n \log n)$ সময়ে এই পাশাপাশি সব সংখ্যার LCP বের করে রাখতে পারি। আর পরে দুটি অবস্থানের মধ্যের সর্বনিম্নের কুয়েরি তো আমরা $O(\log n)$ সময়ে বা $O(1)$ সময়ে করতেই পারি (আমরা কিন্তু ডেটা স্ট্রাকচার অধ্যায়ে দেখে এসেছি)। তবে মজার ব্যাপার হলো সাফিক্স অ্যারেতে পাশাপাশি সাফিক্সগুলোর LCP (Z) আসলে লিনিয়ার সময়ে বের করা সম্ভব। এর ধারনার সঙ্গে 'z' ফাংশনের ধারনার বেশ মিল আছে। এই সমাধানের মৌলিক লক্ষণীয় হলো মনে কর আমরা $lcp(i, j) = 10$ পেয়েছি^১ অর্থাৎ $S[i\dots]$ আর $S[j\dots]$ এর LCP হলো 10। তাহলে $S[i+1\dots]$ আর $S[j+1\dots]$ অবশ্যই 9 হবে তাই না? কিন্তু কাহিনি হলো $A[i+1]+1 \neq A[j+1]$ হতেই পারে, অর্থাৎ সাফিক্স অ্যারেতে $S[i\dots]$ আর $S[j\dots]$ পাশাপাশি দুটি সাফিক্স মানে $S[i+1\dots]$ আর $S[j+1\dots]$ ও যে পাশাপাশি দুটি সাফিক্স হবে তা তো আর না। তবে এটি বলাই যায় যে তাদের মধ্যে অন্তত 9 টি ম্যাটিং থাকবে। তাদের মানে হলো $i+1$ এ যেই সাফিক্স আছে তার আর তার বন্ধুর মধ্যে অর্থাৎ $rank[A[i+1]+1]$ এর মধ্যে। গাণিতিকভাষায় বললে বলা যায় $lcp(i+1, rank[A[i+1]+1]) \geq lcp(i, rank[A[i]+1]) - 1$. সুতরাং $O(n)$ এ পাশাপাশি সব LCP বের করার কোড হবে কোড ১১.৫ এর মতো।

কোড ১১.৫: saLcp.cpp

```

1 char S[100];
2 // lcp[] is the Z[] of the description.
3 int lcp[100], A[100], rank[100];
4
5 void LCP()
6 {
7     int n=strlen(S);
8     int now = 0;
9
10    for(int i = 0; i < n; i++) rank[A[i]]=i;
11

```

^১আমরা লিখতে পারি $j = rank[A[i]+1]$ কারণ আমরা তো সবসময় পাশাপাশি দুটি সাফিক্সের LCP বের করছিলাম Z এ। একটি যদি i হয় তাহলে পরেরটি হবে $rank[A[i]+1]$. কারণ i তম সাফিক্স আমাদের সাফিক্স অ্যারের $A[i]$ স্থানে আছে, এর পরের স্থান হলো $A[i]+1$ যেখানে মূল স্ট্রিংের $rank[A[i]+1]$ তম সাফিক্স আছে।

```

12     for(int i = 0; i < n; i++)
13     {
14         now = max(now - 1, 0); // lcp will never be -1.
15         if(rank[i] == n - 1) {
16             // there is nothing as lcp(n - 1, n).
17             now = 0;
18             continue;
19         }
20         // A[i] is the position of S[i ...] in
21         // suffix array.
22         // A[i] + 1, is the next one in
23         // sufix array.
24         // rank[A[i] + 1] is the index of the
25         // "next suffix in suffix array"
26         // in "the main string".
27         int j = rank[A[i] + 1];
28         while(i + now < n
29             && j + now < n
30             && s[i + now] == s[j + now])
31             now++;
32         lcp[A[i]] = now;
33     }
34 }

```

LCP বের করা তো দেখা হলো। এবার সাফিল্ল অ্যারে সম্পর্কিত আরও কিছু সমস্যা দেখা যাক। একটি স্ট্রিং দেওয়া আছে। এর মিনিমাম লেক্সিকোগ্রাফিক রোটেশন (minimum lexicographic rotation) বের করতে হবে। ধরা যাক একটি স্ট্রিং হলো *abac* তাহলে এর রোটেশন (rotation) গুলো হলো *abac*, *baca*, *acab* আর *caba*. এদের মধ্যে লেক্সিকোগ্রাফিকাল হিসেবে ছোট স্ট্রিং বের করতে হবে। সাফিল্ল অ্যারে দিয়ে এটি সমাধান করা খুবই সহজ। মনে কর স্ট্রিংটি *S* তাহলে এই স্ট্রিংটি পরপর দুইবার লিখ *SS*. এখন দেখো প্রথম $|S|$ ঘরের মধ্যে কোন অবস্থানের জন্য *A[]* এর মান সবচেয়ে ছোট। শেষ!

মনে কর তোমাদের একটি স্ট্রিং *S* দেওয়া আছে যার দৈর্ঘ্য n . এখন *S* কে *S* এর সঙ্গে জোড়া লাগিয়ে $2n$ দৈর্ঘ্যের একটি নতুন স্ট্রিং *T* পেলাম। *T* এর প্রতিটি ইনডেক্স i এর জন্য i এ শেষ হয় এবং দৈর্ঘ্য n এর বেশি না এরকম লেক্সিকোগ্রাফিকাল হিসেবে সবচেয়ে ছোট স্ট্রিং ধরা যাক $x(i)$. তোমাকে সেই ইনডেক্সটি বের করতে হবে যার জন্য $x(i)$ সবচেয়ে বড়। সমস্যার বর্ণনাটি অনেক বড় হলেও এর সমাধান খুব ছোট। একটু চিন্তা করে দেখ তো এই সমস্যা আর উপরের সমস্যা

একই কিনা? অর্থাৎ এর সমাধান মিনিমাম লেখিকোগ্রাফিক রোটেশন। তুমি চাইলে proof by contradiction করতে পারো।

এবাবের সমস্যায় তোমাদের একটি স্ট্রিং S এর জন্য সবচেয়ে বড় স্ট্রিং T এর দৈর্ঘ্য বের করতে হবে যেন S এর ভেতরে T অন্তত পক্ষে k বার থাকে। যেমন $ababa$ এর ভেতরে aba মোট ২ বার আছে। এর সমাধানও বেশ সহজ। কোনো একটি স্ট্রিং k বার থাকা মানে সাফিক্স অ্যারেতে পরপর k টি সাফিক্স তুমি পাবেই যাদের প্রিফিক্স হবে সেই k বার থাকা স্ট্রিং। তাহলে তোমাকে যা করতে হবে তা হলো সাফিক্স অ্যারের প্রত্যেক পরপর k টি সাফিক্সের LCP বের করতে হবে। আসলে $[i, i+k-1]$ এই সীমার সাফিক্সগুলোর LCP বের করার জন্য আসলে i আর $i+k-1$ তম সাফিক্সের LCP বের করলেই চলে। এভাবে প্রতিটি i এর জন্য LCP বের করে তাদের সর্বোচ্চটিই উত্তর।

একটু চিন্তা করে দেখ তো সাফিক্স অ্যারের সাহায্যে কোনো স্ট্রিংয়ের স্বতন্ত্র সাব-স্ট্রিংের count বের করতে পার কিনা। খুব একটা কঠিন না। মনে কর তুমি সাফিক্স অ্যারেতে ছোট থেকে বড়তে যাচ্ছ। i তম সাফিক্সে এসে দেখবে এর উপরের সঙ্গে তোমার কত LCP. ঠিক তত মনে করবে আগেই নিয়ে ফেলেছি, বাকি দৈর্ঘ্যটুকু তোমার উত্তরের সঙ্গে যোগ করবে। শোধ!

ধর তোমাদের একটি বড় স্ট্রিং S দেওয়া আছে আর অনেকগুলো (M টি) ছোট ছোট স্ট্রিং দেওয়া আছে যাদের দৈর্ঘ্য ধরা যাক 64 এর বেশি হবে না। এখন তোমাকে প্রতিটি ছোট স্ট্রিংয়ের জন্য বলতে হবে সেটি S এর ভেতরে কয়বার আছে। যেহেতু ছোট স্ট্রিংগুলো আসলে $64 = 2^6$ এর বেশি দৈর্ঘ্য না তাই মাত্র 7 বার সাফিক্স অ্যারে বের করার কাজ (আশা করি তোমাদের মনে আছে আমরা মোট $\log n$ বার স্ট্রিংয়ের কাজ করে সাফিক্স অ্যারে বানিয়েছিলাম?) করলেই হবে। এর পর তুমি প্রতিটি ছোট স্ট্রিংয়ের জন্য দুটি বাইনারি সার্চ (binary search) করবে। একবাবে তুমি সাফিক্স অ্যারের প্রথম সাফিক্সটি বের করবে যার সঙ্গে তোমার স্ট্রিংয়ের LCP বের করে দেখ তা ছোট স্ট্রিংয়ের সমান কিনা, আর আরেকবাবে একইভাবে বাইনারি সার্চ কর তবে তুমি সেই রকম সাফিক্সদের সবশেষেরটি বের করবে। তাহলে মোট কমপ্লেক্সিটি দাঁড়ায় $O(n \log 64 + 64M \log n)$. তুমি চাইলে কিছু বুদ্ধি খাচিয়ে আরও কমাতে পারো।

তিনটি স্ট্রিং দেওয়া আছে। তোমাকে সবচেয়ে বড় স্ট্রিংয়ের দৈর্ঘ্য বের করতে হবে যেন তা তিনটি স্ট্রিংয়েরই সাব-স্ট্রিং হয়। তোমাকে যা করতে হবে তা হলো এই তিনটি স্ট্রিংকে অব্যবহৃত ক্যারেন্টার ব্যবহার করে জোড়া লাগাতে হবে। ধরা যাক এরকম দুটি ক্যারেন্টার হলো # আর ?. তাহলে আমাদের জোড়া লাগা স্ট্রিংটি দেখতে হবে $S_1 \# S_2 ? S_3$ এর মতো। এখন তোমাকে এর সাফিক্স অ্যারে বের করতে হবে। এখন কিছুটা লাইন সুইপ (line sweep) বা স্লাইডিং উইন্ডো (sliding window) এর মতো কাজ করতে হবে। প্রথমে $i = 0$ সেট কর আর j কে 0 হতে বাঢ়াতে থাক যেন $[i, j]$ সীমার মধ্যে ওই তিনটি স্ট্রিংয়েরই সাফিক্স থাকে। এখন তুমি i আর j এর LCP বের করে তোমার উত্তরকে আপডেট কর। এবাব i কে বাঢ়াও, এবং এর জন্য j কে "দরকারে" বাঢ়াতে থাক যতক্ষণ না আবারও এই সীমার মধ্যে তিনটি স্ট্রিংয়েরই সাব-স্ট্রিং থাকে। তখন আবাব LCP বের করে উত্তরকে আপডেট করবে। তিনটি স্ট্রিংয়েরই সাব-স্ট্রিং আছে কিনা তা আসলে সাফিক্সগুলো জোড়া লাগা স্ট্রিংয়ে কোথায় আছে তা দেখেই বলতে পারবে (rank ব্যবহার করে)।

তোমাকে একটি স্ট্রিং দিয়ে বলা হলো এতে থাকা সবচেয়ে বড় প্যালিনড্রম (palindrome) এর দৈর্ঘ্য বের কর। প্রথমেই বলে নেই, যখনই প্যালিনড্রমের সমস্যা দেখবে তখনই জোড় আর বিজোড় প্যালিনড্রমকে আলাদা আলাদা করে চিন্তা করে দেখার কথা ভাববে। তাতে সমস্যা কিছুটা সহজ হয়ে যায়। আমরা আপাতত বিজোড় দৈর্ঘ্য এর কথা ভাবি। আমরা যদি কোনোভাবে $S[i\dots]$ আর $S[i\dots 0]$ এর LCP বের করতে পারতাম তাহলেই হয়ে যেত। খেয়াল কর $S[i\dots]$ হলো i তম সাফিক্স আর $S[i\dots 0]$ কে ভাবতে পার S এর উল্টো স্ট্রিংয়ের সাফিক্স। আমরা যদি S আর উল্টো S কে জোড়া লাগিয়ে তার সাফিক্স অ্যারে বের করি তাহলেই কিন্তু আমাদের সমস্যা সমাধান হয়ে যায়। আমরা প্রতি i আর $2|S| - i - 1$ এর LCP বের করব তাহলেই হবে। জোড়ের ক্ষেত্রেও একইভাবে করলেই হবে।

এখন কিছু কঠিন সমস্যা দেখা যাক। মনে কর একটি স্ট্রিং S দেওয়া আছে। এমন একটি সবচেয়ে ছোট স্ট্রিং T বের করতে হবে যেন অনেকগুলো T পরপর জোড়া লাগিয়ে S বানানো যায়। জোড়া লাগানোর সময় T গুলো পরস্পরের মধ্যে অধিক্রমন বা overlap করতে পারে। যেমন

ababbababbabababbababbababbaba

ababbaba

ababbaba

ababbaba

ababbaba

ababbaba

এর দুই রকম সমাধান বলছি। দুই সমাধানেই আমাদের সাফিক্স অ্যারে লাগবে। প্রথম সমাধান কিছুটা এরকম- খেয়াল কর T কে অবশ্যই S এর প্রিফিক্স হতে হবে। আমরা যা করব তা হলো এক এক করে প্রিফিক্সের দৈর্ঘ্য এক হতে বাড়াতে থাকব আর যাচাই করব যে এই প্রিফিক্সটি আমাদের কাঞ্জিত T হিসেবে কাজ করবে কিনা। আমরা কিছুটা স্লাইডিং উইন্ডো জাতীয় পদ্ধতি ব্যবহার করব। প্রথমে এক দৈর্ঘ্যের জন্য আমরা L এবং R দিয়ে সাফিক্স অ্যারেতে সেসব সাফিক্সকে bound করব যাদের প্রথম ক্যারেটার S এর প্রথম ক্যারেটারের সমান হয়। এর পর যখন আরও এক দৈর্ঘ্য বাড়িয়ে এদের মধ্যের সবার সঙ্গে 2 দৈর্ঘ্য মিল থাকে। এরকম করে প্রত্যেক দৈর্ঘ্যে আমাদের এই কাজ করে এই উইন্ডো ঠিক করতে হবে। এখন বলব প্রতিবার আর কী কী কাজ করতে হবে। একদম প্রথমবার আর পরে যখন L, R কমিয়ে সীমাঠিক করছিলাম তখন আমাদের সেট বা BST থেকে এই ইনডেক্সে বের করতে হবে। এখন এই যে ঢোকানো বা বের করা এই সময় পাশাপাশি দুটি ইনডেক্সের মধ্যের দূরত্ব আমাদেরকে একটি হীপ (heap) বা প্রায়োরিটি কিউ (priority queue) বা আরেকটি সেটে রাখতে হবে। এই কাজটি এর আগেও আমরা করেছি। তাও বলি, যখন তুমি কোনো একটি ইনডেক্স ঢোকাবে তখন এর দুপাশের দুটি ইনডেক্স দেখ আর তাদের মধ্যের পার্থক্যকে হীপ থেকে বাদ দিয়ে নতুন ইনডেক্সের সঙ্গে দুই পাশের দুটি ইনডেক্সের মধ্যের দূরত্ব হীপে রাখতে হবে। আর কোনো

একটি ইনডেক্স বাদ দেওয়ার সময় তার সঙ্গে তার পাশের দুটি ইনডেক্সের পার্থক্যকে হীপ থেকে মুছে ফেলে তাদের মধ্যের পার্থক্যকে হীপে রাখতে হবে। এবার যেই জিনিসটি দেখতে হবে তা হলো হীপের সর্বোচ্চ সংখ্যা কত। যদি সেটি সেই বারের প্রিফিক্সের দৈর্ঘ্যের সমান বা ছোট হয় এর মানে হলো এটিই আমাদের উত্তর। এটুকু যদি বুঝে থাক তাহলে কেন এটি সঠিক তাও একটু চিন্তা করলে বুঝতে পারবে। এখন আসা যাক দ্বিতীয় সমাধানে। দ্বিতীয় সমাধানের জন্য আমাদের S এর প্রতিটি 'z' ফাংশন লাগবে। আর এই সমস্যার ক্ষেত্রে আমরা ধরে নিব $S[0\dots]$ এই সাফিক্সের জন্য S এর সঙ্গে LCP হলো $|S|$. যাই হোক, এখন তুমি এই LCP অনুসারে সাফিক্সের ইনডেক্সকে সর্ট কর বড় হতে ছোট করে। এখন এদের একে একে একটি BST বা সেটে যোগ কর। আগের মতো একটি হীপ নিয়ে তাতে BST বা সেটে ঢোকানো ইনডেক্সগুলোর মধ্যের দূরত্ব ঢোকাতে থাক। মনে কর তুমি এখন যেই LCP এর দৈর্ঘ্যকে নিয়েছ তার দৈর্ঘ্য k আর হীপে থাকা ইনডেক্সের মধ্যের সবচেয়ে বড় পার্থক্য হলো h . এখন যদি $h \leq k$ হয় তাহলে h হবে আমাদের একটি candidate উত্তর। এভাবে সব candidate দের মধ্যে থেকে সবচেয়ে কমটি হবে আমাদের উত্তর।

একটি স্ট্রিং S দেওয়া আছে তোমাকে এর প্রতিটি প্রিফিক্সের জন্য সর্বোচ্চ পর্যায়কাল (period) বের করতে হবে। কোনো একটি প্রিফিক্সের জন্য পর্যায়কাল হবে k যদি এমন একটি স্ট্রিং A থাকে যাকে পরপর k বার জোড়া লাগালে সেই প্রিফিক্স পাওয়া যায়। আমার মনে হয় তোমরা এই সমস্যার সমাধান KMP বা 'z' ফাংশন ব্যবহার করে কীভাবে করতে হবে তা একটু চিন্তা করলে করে ফেলতে পারবে। সাফিক্স অ্যারের সমাধানও একই রকম। তোমাকে প্রতিটি সাফিক্সের জন্য মূল স্ট্রিংয়ের সঙ্গে LCP বের করতে হবে। ধর i ইনডেক্সের জন্য LCP হলো k . তাহলে বলা যায় k দৈর্ঘ্য পর্যন্ত i পর্যন্ত প্রিফিক্সকে পরপর বসানো যায়। উদাহরণ দেওয়া যাক, মনে কর $abababac$ এই স্ট্রিংয়ে $i = 2$ এর জন্য k হবে 5 কারণ i থেকে শুরু হওয়া সাফিক্স হলো $ababac$ যার সঙ্গে মূল স্ট্রিংয়ের LCP হলো 5. এর মানে প্রিফিক্স ab বার বার বসাতে থাকো যতক্ষণ না 5 দৈর্ঘ্য পাচ্ছ অর্থাৎ $(ab)(ab)(a)$. অর্থাৎ ab প্রিফিক্সটি মোট 3 বার পুনরাবৃত্তি হয়। এভাবে তুমি প্রতিটি প্রিফিক্সের জন্য তা কতবার পুনরাবৃত্তি হয় তা বের করে ফেলতে পারবে। এখন i এর প্রতিটি গুনিতকে গিয়ে আপডেট করে দিয়ে আসতে পার তার পর্যায়কাল যাতে সব শেষে তুমি প্রিফিক্সগুলোর সর্বোচ্চ পর্যায়কাল পাও। প্রতিটি গুনিতকে গিয়ে আপডেট করা কিন্তু ততটা ব্যয়বহুল না। তুমি 1 দৈর্ঘ্যের জন্য আপডেট করবে খুব জোড় $n/1$ বার, 2 এর জন্য $n/2$ বার এরকম করে n এর জন্য n/n বার। আর আশা করি তোমরা জানো যে $n/1 + n/2 + \dots n/n \approx n \log n$.

তোমাকে একটি স্ট্রিং S দেওয়া হবে। তোমাকে এর ভেতরে থেকে একটি সাব-স্ট্রিং বের করতে হবে যার পর্যায়কাল সর্বোচ্চ হয়। কোনো একটি স্ট্রিংয়ের পর্যায়কাল k মানে একটি স্ট্রিং A আছে যাকে k বার পর পর জোড়া লাগালে মূল স্ট্রিংটি পাওয়া যায়। আমরা এই A এর দৈর্ঘ্যের উপর লুপ চালাব। ধরা যাক এই দৈর্ঘ্য হলো L এবং আমরা 1 হতে $n = |S|$ পর্যন্ত L এর লুপ চালাব। প্রতি L এর জন্য আমরা যা করব তা হলো মূল স্ট্রিং S কে L ভাগে ভাগে ভাগ করব। যেমন $babbabaabaabab$ কে আমরা যদি $L = 3$ এর জন্য ভাগ করতে চাই তাহলে হবে $(bab)(bab)(aab)(aab)(aab)(ab)$. শেষ ভাগে L এর থেকে কম থাকলে সমস্যা নেই। এখন ধর আমরা প্রতিটি ভাগকে সংখ্যামান দিচ্ছি: 0 তম ভাগ, 1 তম ভাগ এরকম। আমাদের যা করতে হবে

তা হলো পাশাপাশি দুটি ভাগের LCP বের করতে হবে। যেমন উপরের ভাগের জন্য $lcp(0, 1) = 3$, $lcp(1, 2) = 0 \dots lcp(4, 5) = 1$. একইভাবে আমাদের LCS বের করতে হবে মানে দীর্ঘতম সাধারণ সাফিক্স বা longest common suffix যেমন $lcs(1, 2) = 2$ কারণ *bab* আর *aab* এর LCS হলো 2. LCS আসলে তোমরা LCP এর মতো বের করতে পার, শুধু স্ট্রিংটিকে উল্টো করে নিতে হবে। এখন তোমাকে এই LCP দেখে দেখে বুঝতে হবে কোন কোন ভাগ সমান। এটি তো বোঝা খুবই সোজা, যদি LCP L এর সমান বা বড় হয় এর মানে সমান। এখন উপরের ভাগের দিকে দেখ। তুমি কি বলবে যে পাশাপাশি *aab* তিনটি আছে তাই আসলে 3 দৈর্ঘ্যের কোনো একটি A কে 3 বারের বেশি পুনরাবৃত্তি করা যাবে না? না খেয়াল করলে দেখবে উপরের উদাহরণে *aba* কে 4 বার পুনরাবৃত্তি করা যায়। তাহলে তা কীভাবে বের করা যায়? প্রথমত আমরা সমান সেগমেন্টগুলো বের করি, যেমন উপরের উদাহরণে 2, 3, 4 সমান সেগমেন্ট। এবার দেখতে হবে এই সেগমেন্টগুলোকে ডানে ও বামে কত দূর বাড়ানো যায়। যেমন $lcs(1, 2) = 2$ আর $lcp(4, 5) = 2$. এর মানে এই 3 দৈর্ঘ্য করে সেগমেন্ট নেওয়ার কাজ বাম দিকে আরও 2 ঘর সরানো সন্তুষ্ট। তাহলে মোট $2 + 3 \times 3 + 2 = 13$ ঘর জুড়ে 3 দৈর্ঘ্যের স্ট্রিংয়ের পুনরাবৃত্তি করার কাজ করা সন্তুষ্ট। আর আমরা জানি $\lfloor 13/3 \rfloor = 4$ তার মানে আমরা 3 দৈর্ঘ্যের একটি স্ট্রিং পেয়েছি যা 4 বার পুনরাবৃত্তি হয়। শেষ! এই সমাধানের কমপ্লেক্সিটি $O(n \log n)$. এখানে আমি ধরে নিয়েছি LCP বা LCS তুমি $O(1)$ এ বের করবে। সাফিক্স অ্যারেতে পাশাপাশি দুটি সাফিক্সের মধ্যে কীভাবে LCP বের করা যায়? সহজ, আমরা বলেছি সাফিক্স অ্যারেতে; তম আর j তম সাফিক্সের মধ্যে LCP হলো পাশাপাশি LCP গুলোর মধ্যে সর্বনিম্ন। অর্থাৎ এটি রেঞ্জ মিনিমান কুয়েরি (range minimum query). আর $O(n \log n)$ এ প্রসেসিংও $O(1)$ এ কুয়েরি কীভাবে করা যায় তা তো আমরা জানিই।

১১.৭ প্রোগ্রামিং সমস্যা

১১.৭.১ অনুশীলনী

সহজ

- UvaLive 6805 • UvaLive 6976 • UvaLive 7017 • UvaLive 7033

সামান্য কঠিন

- UvaLive 6806 • UvaLive 6808 • UvaLive 6856* • UvaLive 6893**
- UvaLive 7041 • UvaLive 7071

কঠিন

- UvaLive 6933*

লেখক পরিচিতি



মো: মাহবুবুল হাসান (শান্ত)-এর জন্ম ১৯৮৬ সালে। তিনি রাজশাহীর অগ্রণী বিদ্যালয় থেকে মাধ্যমিক ও নিউ গভঃ ডিগ্রী কলেজ থেকে উচ্চ মাধ্যমিক সম্পন্ন করেন। ২০০৩ সালের প্রথম জাতীয় গণিত অলিম্পিয়াডে একশো পেয়ে সেকেন্ডারি ক্যাটাগরিতে চ্যাম্পিয়ন হন তিনি। ২০০৫ সালের বাংলাদেশ হতে আন্তর্জাতিক গণিত অলিম্পিয়াডগামী প্রথম দলের সদস্য ছিলেন। এছাড়াও ২০০৫ সালের আন্তর্জাতিক ইনফরমেটিক্স অলিম্পিয়াডগামী প্রথম দলের সদস্যও ছিলেন, যদিও ভিসা জটিলতার কারণে সেবার বাংলাদেশের অংশগ্রহণ করা হয়ে ওঠে না। কলেজ পড়ুয়াদের আইওআইগামী সেই দলটি ২০০৫ সালের ঢাকা সাইটের আইসিপিসিতে বাংলাদেশের সকল বিশ্ববিদ্যালয়ের দলকে হারিয়ে দ্বিতীয় স্থান অর্জন করেন, প্রথম হয় চীনের ফুদান বিশ্ববিদ্যালয়। বুয়েটের কম্পিউটার সায়েন্স ও ইঞ্জিনিয়ারিং বিভাগে পড়াকালীন সময়ে হাতে গোনা তিন চারটি কনটেন্ট বাদে বাকি প্রায় ত্রিশটির মতো কনটেন্টে তাদের দল চ্যাম্পিয়ন হয়। ২০০৮ ও ২০০৯ সালের এসিএম আইসিপিসি ওয়ার্ল্ড ফাইনালস-এ তাদের দল অংশগ্রহণ করে যথাক্রমে ৩১তম এবং ৩৪তম স্থান অর্জন করে। এছাড়াও প্রথম বাংলাদেশি হিসেবে তিনি টপকোডার এবং কোডফোর্সেস উভয়ের রেড কোডার হন। বুয়েটের পড়াশুনার পাট চুকিয়ে আড়াই বছর বুয়েটে শিক্ষকতা করেন। শিক্ষকতার পাশাপাশি এসময়ে তিনি বুয়েট এবং আইওআই এর ছেলেমেয়েদের প্রোগ্রামিং এর প্রশিক্ষণ দেন। ২০১১ ও ২০১৩ তে আইওআই দলের দলনেতা হিসাবে ছিলেন তিনি। বর্তমানে তিনি গুগলের সুইজারল্যান্ড অফিসে কর্মরত আছেন।



www.dimik.pub

www.dimikprokashoni.com

