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 সূত্র কাজ করবে যদি $n > 1$ হয়। সেক্ষেত্রে $way(2) = way(1) + way(0)$. $way(1)$ বা $way(0)$ এর মান কিন্তু আমরা এই সূত্র ব্যবহার করে বের করতে পারব না। কারণ আমরা আগেই বলেছি এই সূত্র 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 অ্যারের 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 উত্তর। আর আমরা জানি, $way(2) = way(1) + way(0)$ এখন আমরা জানি $way(2) = 2$ এবং $way(1) = 1$ তাহলে তো $way(0) = 1$ হবেই তাই না? এরকম কেন 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 DP[১] 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

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

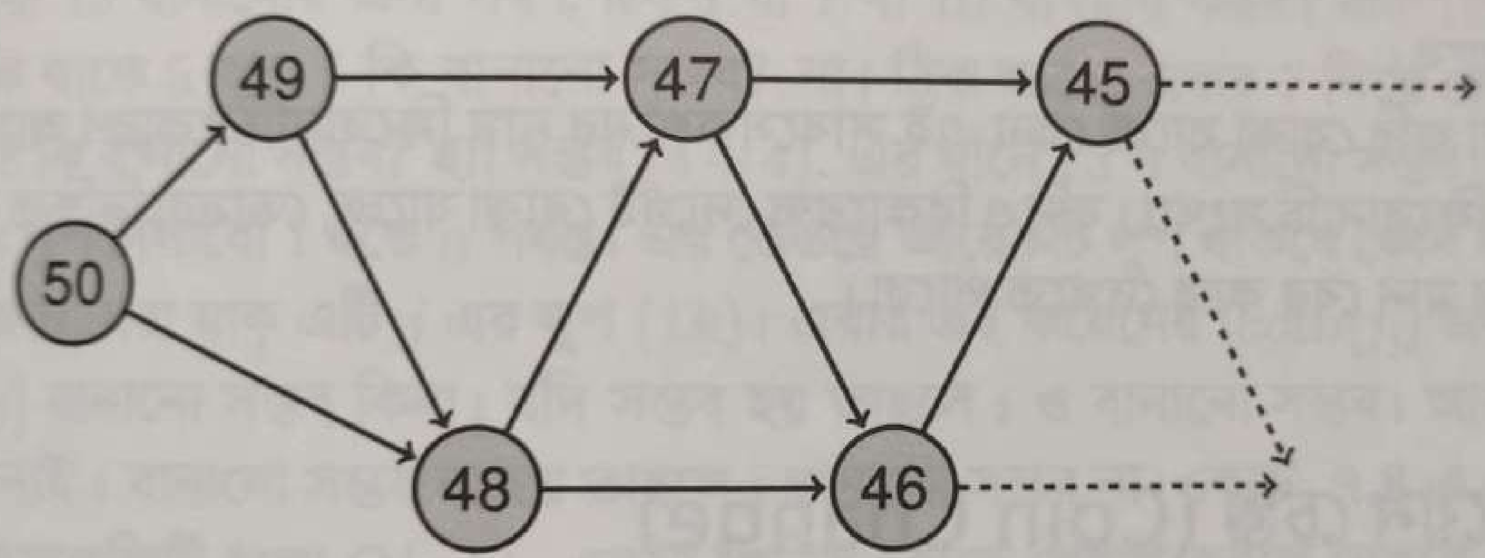
কোড ৭.২: fibRecursive.cpp

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

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

[১]ডায়নামিক 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 বুঝবে 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

```
int dp[1000]; // initialize to −1.

int way(int n)
{
    if(n == 0 || n == 1) return 1;
    if(dp[n] != −1) return dp[n];
    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?

```
৮ }
```

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 রেখেছি 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 বুঝাতে 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 বুঝেছ 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 পদ্ধতিতেতো অ্যারেতে -1 রাখতে হতো বা $visited$ কে 0 করতে হতো। সেটি না করে $marker$ এর মান এক বাড়িয়ে দিলেই হয়ে যাবে।

আর আশা করি বোঝা যাচ্ছে কেন এই সাবসেকশনের নাম ফিবোনাচি! কারণ আমাদের way হলো আসলে ফিবোনাচি সংখ্যা। যদিও রিকারেন্স দেখেই বোঝা 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: $coin[1 \ldots k]$ (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! সুতরাং $1 \ldots n$ 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। সুতরাং আমাদের বলতে হবে যে 0 ও বানানো সম্ভব।

কোড ৭.৪: variant1.cpp

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

আশা করি বুঝতে পারছ কীভাবে base কেইস এবং সেক্ষেত্রে উত্তর বের করতে হবে? প্রথমত দেখবে যে তোমার সূত্র কোন মানের জন্য কাজ করবে না বা কাজ করানো যায় না। সেসব ক্ষেত্রে তোমাকে চিন্তা করতে হবে base কেইসের উত্তর 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 $way[n - coint[1]]$ 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 5 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

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

৭.২.৩ Variant 3

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

খেয়াল কর আগের পদ্ধতিতে আমরা যা করেছি 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, $n - coin[i]$ পরিমাণ $i - 1$ পর্যন্ত কয়েন ব্যবহার করে বানানো যায় কিনা। অর্থাৎ আগে আমরা যাচাই করতাম যে $dp[n - coin[i]]$ সত্য কিনা, এখন আমাদের দেখতে হবে $dp[i-1][n - coin[i]]$ 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 দুটা জিনিস দেখতে হবে তাহলো n পরিমাণ $i - 1$ পর্যন্ত কয়েন দিয়ে বানানো যায় কিনা। আর $n - coin[i]$ 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 কমপ্লেক্সিটিই $O(nk)$.

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 $dp[j][\ldots]$ বানাতে আমাদের শুধু $dp[j-1][\ldots]$ 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 উত্তর লিখা আছে। সুতরাং আমাদের আগের $dp[j-1][i]$ আসলে এখন $dp[i]$ তে লিখা। এখন কথা হলো $dp[j-1][i - coin[j]]$ কোথায় পাব? চিন্তা করে দেখো $dp[i - coin[j]]$ 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 $i - coin[j]$ পার করে এসেছি, এবং হয়তো আগের মানকে আপডেটও করে এসেছি। কিন্তু আপডেট করলে তো $dp[j-1][i-coin[j]]$ এর মান $dp[i-coin[j]]$ তে থাকবে না। উপায় হলো, i এর লুপ 1 হতে n না চালিয়ে n হতে 1 চালাও। তাহলে তুমি যখন i এ আছো তখনও $i - coin[j]$ এর মান পরিবর্তন হয় নাই। এটাই হলো চালাকি। তোমাদের জন্য এই দুইটিরই কোড ৭.৬ তে দেওয়া হলো।

কোড ৭.৬: variant3.cpp

```
// O(n^2) memory.
```

```
dp[0][0] = 1;
for (int j = 1; j <= k; j++) {
    for (int i = 1; i <= n; i++) {
        if (dp[j - 1][i] ||
            (i >= coin[j] && dp[j - 1][i - coin[j]])) {
            dp[j][i] = 1;
        }
    }
}

// O(n) memory.
dp[0] = 1;
for (int j = 1; j <= k; j++) {
    for (int i = n; i >= 1; i--) {
        if (i >= coin[j] && dp[i - coin[j]]) {
            dp[i] = 1;
        }
    }
}
```

৭.২.৪ Variant 4

বুঝতেই 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 সেজন্য 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 n 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 সংখ্যকবার। তাহলে i তম কয়েনকে t সংখ্যকবার 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 প্রায় $O(n^2k)$ 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 করা যাবে 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 একাধিকবার 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 ছোট 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 $5 \ldots n$ 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