ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-৩ (কয়েন চেঞ্জ + রক ক্লাইম্বিং)

জুলাই ১৩, ২০১২ by *শাফায়েত*

আগের পর্বগুলো পড়ে থাকলে তুমি এখন ডাইনামিক প্রোগ্রামিং নিয়ে বেসিক ব্যাপারগুলো কিছুটা শিখে গিয়েছো,যত প্রবলেম সলভ করবে তত দক্ষতা বাড়বে। ডিপিতে আসলে কোনো নির্দিষ্ট অ্যালগোরিদম না থাকায় আমাদের চিন্তা করতে হয় অনেক বেশি,একেকটি ডিপি প্রবলেম একেক ধরণের, তবে তুমি যদি ন্যাপস্যাক,কয়েন চেঞ্জের মতো ক্লাসিক কিছু ডিপি প্রবলেমের সলিউশন জানো তাহলে তুমি বুঝতে পারবে কিভাবে তোমার চিন্তাকে এগিয়ে নিয়ে যেতে হবে,কিভাবে ডিপির স্টেট নির্ধারণ করতে হবে,তখন তুমি নতুন ধরণের ডিপি প্রবলেমও সলভ করে ফেলতে পারবে। আমি এরই মধ্যে ^nC_r নির্ণয় আর ০-১ ন্যাপস্যাকের ডিপি সলিউশন নিয়ে আলোচনা করেছি,আরো কিছু ক্লাসিক বা স্ট্যান্ডার্ড প্রবলেম নিয়ে সামনে আলোচনা করবো।

কয়েন চেঞ্জ:

এখন আমরা দেখবো কয়েন চেঞ্জ প্রবলেম। আসলে ন্যাপসাক শেখার পরে কয়েন চেঞ্জ তুমি এমনিই পারবে তারপরেও লিখছি যাতে ব্যাপারগুলো আর পরিষ্কার হয়।

তোমার কাছে কিছু কয়েন আছে যাদের মূল্য $5, 8, 11, 15, 18$ ডলার। প্রতিটা কয়েন অসীম সংখ্যকবার আছে, তুমি যেকোনো কয়েন যতবার ইচ্ছা নিতে পারো। তাহলে তোমার coin অ্যারেটা হতে পারে এরকম:

```
coin[] = {5, 8, 11, 15, 18};
```

top

এখন তোমাকে এই কয়েনগুলো নিয়ে নির্দিষ্ট কোনো ভ্যালু বানাতে হবে। ধরি সংখ্যাটি হলো $make$। $make = 18$ হলে আমরা $5+5+8$ এভাবে 18 বানাতে পারি। তোমাকে বলতে হবে কয়েনগুলো দিয়ে ভ্যালুটি বানানো যায় নাকি যায়না। প্রথমেই আমরা চিন্তা করি ডিপিতে স্টেট কি হবে। আমরা একটি একটি কয়েন নিয়ে সংখ্যাটি বানাতে চেষ্টা করতে থাকবো। তাহলে এই মুহূর্তে কোন কয়েন নিচ্ছি সেটা স্টেট রাখতে হবে, আর আগে যেসব কয়েন নিয়েছি সেগুলোর মোট ভ্যালু কত সেটা রাখতে হবে। ফাংশনটির নাম call হলে প্রোটোটাইপ হবে:

```cpp
int call(int i,int amount)
```

এরপর অনেকটা আগের মতোই কাজ। প্রথমে i নম্বর কয়েন নিতে চেষ্টা করবো:

> ```cpp
> if(amount+coin[i]<=make)
>   ret1=call(i,amount+coin[i]);
> else
>   ret1=0;
> ```

এখানে $i+1$ কল না করে আবার i কল করছি কারন এক কয়েন অনেকবার নেয়া সম্ভব। যদি এক কয়েন একাধিক বার নেয়া না যেতো তাহলে $i+1$ কে কল দিতাম। $amount + coin[i]$ যদি $make$ এর থেকে বড় হয় তাহলে কয়েনটি নেয়া সম্ভবনা। কয়েন যদি না নেই তাহলে আমরা পরবর্তী কয়েনে চলে যাবো:

```cpp
ret2=call(i+1,amount);
```

$ret1$ আর $ret2$ এর কোনো একটি true হলেও make বানানো যাবে। তাহলে সবশেষে লিখবো:

```cpp
return ret1|ret2;
```

আর বেসকেস হবে হলো, যদি সব কয়েন নিয়ে চেষ্টা করার পর $make$ বানানো যায় তাহলে return 1, অন্যথায় return 0। সম্পূর্ণ কোড:

```cpp
int coin[]={5,8,11,15,18}; //value of coins available
int make; //our target value
int dp[6][100];
int call(int i,int amount)
{
    if(i>=5) { //All coins have been taken
        if(amount==make)return 1;
        else return 0;
    }
    if(dp[i][amount]!=-1) return dp[i][amount]; //no need to calculate same state twice
    int ret1=0,ret2=0;
    if(amount+coin[i]<=make) ret1=call(i,amount+coin[i]); //try to take coin i
    ret2=call(i+1,amount); //dont take coin i
    return dp[i][amount]=ret1|ret2; //storing and returning.

}
int main()
{
    // freopen("in","r",stdin);
    while(cin>>make)
    {
        memset(dp,-1,sizeof(dp));
        cout<<call(0,0)<<endl;
    }
    return 0;
```

top

```
}
```

এখন যদি তোমাকে বলা হতো যে কোনো একটি ভ্যালু কতবার বানাতে হবে বলতে হবে তাহলে কি করতে? যেমন ১৮ বানানো যায় ২ ভাবে, এক্ষেত্রে ret1|ret2 রিটার্ন না করে ret1 + ret2 রিটার্ন করে দাও,তাহলে যত ভাবে বানানো যায় সবগুলো যোগ হয়ে যাচ্ছে। uva-674 প্রবলেমটিতে এটাই করতে বলা হয়েছে,ঝটপট সলভ করে ফেলো।

এবার মজার একটি অপটিমাইজেশন দেখো। আমরা প্রতিবার make ইনপুট নেয়ার পর ডিপি অ্যারে নতুন করে initialize বা ক্লিয়ার করেছি। যদি সেটা করতে না হতো তাহলে অনেক কম সময় লাগতো, কারণ একই মান বারবার হিসাব করা লাগবেনা। কিন্তু ক্লিয়ার করতে হচ্ছে কারণ ফাংশনটি বাইরের একটি গ্লোবাল ভ্যারিয়েবলের উপর নির্ভরশীল, "if(amount==make)return 1;" এই লাইনটাই ঝামেলা করছে, $make$ এর মান প্রতি কেসের জন্য আলাদা, তাই প্রতিবার নতুন করে সব হিসাব করতে হচ্ছে। আমরা যদি $make$ কে একটা স্টেট হিসাবে রাখি তাহলে কাজ হয় কিন্তু স্টেট বিশাল হয়ে যায়। এর থেকে আমরা সমস্যাটাকে উল্টায় ফেলি। মনে করো তোমার কাছে শুরুতে ২০ টাকা আছে, বিভিন্ন অ্যামাউন্টের কয়েন দান করে দিয়ে তোমাকে শুন্য টাকা বানাতে হবে। কোডটা এবার হবে এরকম:

```
int coin[]={5,8,11,15,18}; //value of coins available
int make=18; //we will try to make 18
int dp[6][100];
int call(int i,int amount)
{
    if(i>=5) { //All coins have been taken
        if(amount==0)return 1;
        else return 0;
    }
    if(dp[i][amount]!=-1) return dp[i][amount]; //no need to calculate same state twice
    int ret1=0,ret2=0;
    if(amount-coin[i]>=0) ret1=call(i,amount-coin[i]); //try to take coin i
    ret2=call(i+1,amount); //dont take coin i
    return dp[i][amount]=ret1|ret2; //storing and returning.

}
int main()
{
    // freopen("in","r",stdin);
    memset(dp,-1,sizeof(dp));
    while(cin>>make)
    {
        cout<<call(0,make)<<endl;
    }
    return 0;
}
```

খেয়াল করে দেখো ঠিক আগের মতোই কাজ করেছি, শুধু যোগ করার জায়গায় বিয়োগ করে $make$ থেকে শুন্য বানানোর চেষ্টা করেছি। লাভটা হলো এখন ফাংশনটি কোনো পরিবর্তনশীল গ্লোবাল ভ্যারিয়েবলের উপর নির্ভর করেনা, তাই মেইন ফাংশনে ডিপি অ্যারে লুপের মধ্যে ক্লিয়ার করা দরকার নাই। প্রতিবার কয়েন একই থাকছে বলে এই ট্রিকসটা কাজ করছে, কয়েনের মান পরিবর্তন হলে কাজ করবেনা। ডিপির প্রবলেমে অনেকসময় টেস্টকেস অনেক বেশি দেয় যাতে বার বার ক্লিয়ার করলে টাইম লিমিট পাস না করে।

top

কমপ্লেক্সিটি
ডিপি অ্যারের সাইজ হবে কয়েন সংখ্যা × সর্বোচ্চ যত ভ্যালু বানাতে হবে। তাহলে মেমরি কমপ্লেক্সিটি $O(numberofcoin \times make)$.
ডিপি অ্যারের প্রতিটা সেলই ভরাট করা লাগতে পারে, তাই টাইম কমপ্লেক্সিটিও এখানে $O(numberofcoin \times make)$। যদি ভিতরে কোনো এক্সট্রা লুপ চলতো তাহলে সেটাও টাইম কমপ্লেক্সিটির সাথে যোগ হতো।

লাইটওজেতে 1231 – Coin Change (I) প্রবলেমে কিছু কয়েন দিয়ে একটি ভ্যালু কয়ভাবে বানানো যায় সেটা বের করতে বলেছে, তবে প্রতিটি কয়েন সর্বোচ্চ কয়বার ব্যবহার করা যাবে সেটা বলা আছে, i নম্বর কয়েন C_iবারব্যবহারকরাযাবে।এইকনডিশনটাদুইভাবেতুমিহ্যান্ডেলকরতেপ
taken_iযেটাবলেদিবেনম্বরতুমিকয়বারনিয়েছো,C_{i}$ বার ব্যবহার হয়ে গেলে পরবর্তী কয়েনে চলে যাও।

```
int call(int i, int taken_i, int amount)
```

২য় উপায় হলো ফাংশনের ভিতরে C_i পর্যন্ত একটি লুপ চালিয়ে কয়েনটি যতবার নেয়া সম্ভব ততবার নিয়ে অ্যামাউন্টটি বানাতে চেষ্টা করো, এক্ষেত্রে মেমরি কম লাগবে। তাহলে আজকের ২য় কাজ হলো এই প্রবলেমটা সলভ করা।

কয়েন চেঞ্জ প্রবলেমের আরেকটি নাম হলো subset sum problem,কারণ কিছু নম্বরের সেট থেকে একটি সাবসেট আমাদের নিতে হয় যেটার যোগফল এক নির্দিষ্ট ভ্যালুর সমান।

রক-ক্লাইম্বিং প্রবলেম
তোমাকে একটি ২ডি গ্রিড দেয়া হলো:

> *-1 2 5*
>
> *4 -2 3*
>
> *1 2 10*

তুমি শুরুতে আছো (০,০) সেলে। তুমি শুধু ৩দিকে যেতে পারো:

> *(i + 1, j)*
>
> *(i + 1,j – 1)*
>
> *(i + 1, j + 1)*

প্রতিটি সেলে গেলে তোমার পয়েন্টের সাথে ওই সেলের সংখ্যাটি যোগ হয়। তুমি সর্বোচ্চ কত পয়েন্ট বানাতে পারবে?
এই প্রবলেমকে রক ক্লাইম্বিং প্রবলেমও বলা হয়।
উপরের গ্রিডে সর্বোচ্চ পয়েন্ট 7 = -1+-2+10। এই প্রবলেমের জন্য:

top

> ***স্টেট:*** *তুমি এখন কোন সেল এ আছো*
>
> ***এক থেকে অন্য স্টেটে যাওয়ার উপায়:*** *প্রতিটা সেল থেকে ৩দিকে যাবার চেষ্টা করো, যেদিকে সর্বোচ্চ পয়েন্ট পাবে সেটা রিটার্ণ করো।*
>
> ***বেসকেস:*** *যদি গ্রিডের বাইরে চলে যাও তাহলে আর কিছু নেয়া যাবেনা, শুণ্য রিটার্ণ করো।*

```
#define inf 1 << 28
int mat[][10] = {
    { -1, 2, 5 },
    { 4, -2, 3 },
    {
        1, 2, 10,
    }
};
int dp[10][10];
int r = 3, c = 3;
int call(int i, int j)
{
    if (i >= 0 && i < r and j >= 0 and j < c) //if still inside the array
    {
        if (dp[i][j] != -1)
            return dp[i][j];
        int ret = -inf;
        //try to move to 3 direction,also add current cell's point
        ret = max(ret, call(i + 1, j) + mat[i][j]);
        ret = max(ret, call(i + 1, j - 1) + mat[i][j]);
        ret = max(ret, call(i + 1, j + 1) + mat[i][j]);
        return dp[i][j] = ret;
    }
    else
        return 0; //if outside the array
}
int main()
{
    // READ("in");
    mem(dp, -1);
    printf("%d\n", call(0, 0));
    return 0;
}
```

৩দিকে মুভ করার কন্ডিশন কেনো দেয়া হয়েছে? adjacent ৪টি সেলে মুভ করতে দিলে সমস্যা কোথায় হতো? তাহলে একটি সাইকেল তৈরি হতো,একটি ফাংশনের কাজ শেষ হবার আগেই রিকার্শনে ঘুরে ফাংশনটি আবার কল হতো,এই সলিউশন কাজ করতো না। এখানে আমরা যে ৩দিকে মুভ করছি তাতে কোনো সাইকেল তৈরি হচ্ছেনা, অর্থাৎ কোনো স্টেট থেকে শুরু করে সেই স্টেটে আবার ফিরে আসতে পারছিনা। সাইকেল যদি তৈরি হতো তাহলে রিকার্সিভ ফাংশন আজীবন চলতেই থাকতো। **তাই ডিপি সলিউশন লেখার সময় অবশ্যই খেয়াল রাখতে হবে সাইকেল তৈরি হচ্ছে নাকি।**

1004 – Monkey Banana Problem এই প্রবলেমটা অনেকটা উপরের প্রবলেমের মতো,সলভ করতে কোনো সমস্যা হবেনা। তুমি যদি বিএফএস/ডিএফএস পারো তাহলে uva-11331 প্রবলেমটি সলভ করে ফেলো,(হিন্ট:বাইকালারিং+ন্যাপস্যাক)। এছাড়া এগুলো ট্রাই করো:

top

Uva 11137: Ingenuous Cubrency(Coin change)
Codeforces 118D: Caesar’s Legions(4 state)
Light oj 1047: Neighbor House
Timus 1017: Staircases

চেষ্টা করো সবগুলো প্রবলেম সলভ করতে,আটকে গেলে সমস্যা নাই,চিন্তা করতে থাকো,এছাড়া ন্যাপস্যাক আর কয়েন চেঞ্জ সম্পর্কিত যতগুলো পারো প্রবলেম সলভ করে ফেলো,যেহেতু তুমি এখন বেসিক পরবর্তী পর্বগুলোতে ডিটেইলস কমিয়ে নতুন নতুন প্রবলেম আর টেকনিক নিয়ে বেশি আলোচনা করবো।
হ্যাপি কোডিং!
সবগুলো পর্ব

ফেসবুকে মন্তব্য

0 comments

0 Comments Sort by **Newest ⇕**

Add a comment...

Facebook Comments plugin

Powered by Facebook Comments

Posted in অ্যালগোরিদম/প্রবলেম সলভিং, প্রোগ্রামিং ? Tagged ডাইনামিক প্রোগ্রামিং, ডিপি

49,522 বার পড়া হয়েছে

◂ ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি-২

গ্রাফ থিওরি: স্টেবল ম্যারেজ প্রবলেম ▸

43 thoughts on “ডাইনামিক প্রোগ্রামিং এ হাতেখড়ি- ৩ (কয়েন চেঞ্জ + রক ক্লাইম্বিং)”

top