ন্যাপস্যাক

তাসমীম রেজা

৪ জানুয়ারী ২০১৯

ডাইনামিক প্রোগ্রামিং শুরু করার জন্য সবাই সম্ভবত ন্যাপস্যাক প্রবলেম দিয়েই শুরু করে। প্রোগ্রামিং কনটেস্টে ন্যাপস্যাক অবশ্য কখনো সরাসরি আসে না, কিছু অবজারভাশনের মাধ্যমে ন্যাপস্যাকে কনভার্ট করতে হয়। এমনিতে ন্যাপস্যাক প্রবলেম সুডো-পলিনোমিয়াল টাইমে কাজ করে (অর্থাৎ NP হার্ড প্রবলেম)। তবে ন্যাপস্যাকের অনেক ভ্যারিয়েশনের অনেক কম কমপ্লেক্সিটির অ্যালগরিদম আছে।

0/1 Knapsack

ধর তোমার কাছে n টি বস্তু আছে, i $(1 \leq i \leq n)$ তম বস্তুর ওজন w_i এবং দাম v_i। তোমার কাছে একটা ব্যাগ (ন্যাপস্যাক) আছে যা সর্বোচ্চ W ওজনের বস্তু ধারণ করতে পারে। এই ব্যাগে তুমি সর্বোচ্চ কত দামের বস্তু রাখতে পারবে?

অর্থাৎ আমাদের $S = \{1, 2, 3, \ldots n\}$ থেকে একটি সাবসেট T সিলেক্ট করতে হবে যেন $\sum_{i \in T} w_i \leq W$ হয় এবং $\sum_{i \in T} v_i$ ম্যাক্সিমাইজ হয়।

একে 0/1 Knapsack বলা হয়, কারণ এখানে প্রতিটি বস্তু সর্বোচ্চ একবারই নেওয়া যাবে। এটির জন্য আমাদের ডাইনামিক প্রোগ্রামিং এর সাহায্য নিতে হবে। ধরি $f(i, j) =$ প্রথম i টি বস্তুর মধ্যে সর্বোচ্চ কত দামের বস্তু নেওয়া যায় যাতে বস্তুগুলোর ওজনের যোগফল $\leq j$ হয়। তাহলে আমাদের রিকারেন্সটি :

$$f(i, j) = \max\{f(i-1, j), f(i-1, j-w_i) + v_i\}$$

অর্থাৎ $f(n, W)$ এর মানই হবে আমাদের অ্যান্সার। এখানে টাইম ও মেমরি কমপ্লেক্সিটি উভয়ই $O(nW)$। কোড:

```
for(int j = 0; j <= W; j++) {
  f[0][j] = 0;
}
for(int i = 1; i <= n; i++) {
  for(int j = 0; j <= W; j++) {
    if(j >= w[i]) {
      f[i][j] = max(f[i - 1][j], f[i - 1][j - w[i]] + v[i]);
    } else {
      f[i][j] = f[i - 1][j];
    }
  }
}
/* answer will be found at f[n][W] */
```

তবে যেহেতু $f(i, j)$ এর মান কেবলমাত্র $f(i-1, 0), f(i-1, 1), f(i-1, 2), \ldots, f(i-1, W)$ এর ওপর নির্ভর করে তাই $O(W)$ মেমরি দিয়েও কাজটি করা সম্ভব।

```
for(int j = 0; j <= W; j++) {
  f[j] = 0;
}
for(int i = 1; i <= n; i++) {
  for(int j = W; j >= 0; j--) {
    if(j >= w[i]) {
      f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
  }
}
/* answer will be found at f[W] */
```

লক্ষণীয় বিষয় হল দ্বিতীয় কোডটিতে j এর লুপটি 0 থেকে W পর্যন্ত না চালিয়ে W থেকে 0 পর্যন্ত চালান হয়েছে। এর কারণ কি আশা করি সবাই বুঝতে পারছ।

0-K Knapsack

ধর তোমার কাছে n টাইপের বস্তু আছে, i $(1 \leq i \leq n)$ তম টাইপের বস্তু আছে k_i টি এবং এদের প্রত্যেকটির ওজন w_i এবং দাম v_i। তোমার কাছে একটা ব্যাগ (ন্যাপস্যাক) আছে যা সর্বোচ্চ W ওজনের বস্তু ধারণ করতে পারে। এই ব্যাগে তুমি সর্বোচ্চ কত দামের বস্তু রাখতে পারবে?

আগেরটার সাথে এটার পার্থক্য হচ্ছে এখানে i তম বস্তু সর্বোচ্চ k_i সংখ্যক বার নেওয়া যাবে। এখানেও আগের মতই ডাইনামিক প্রোগ্রামিং ব্যবহার করা যায়, ধরি $f(i,j) =$ প্রথম i টি বস্তুর মধ্যে সর্বোচ্চ কত দামের বস্তু নেওয়া যায় যাতে বস্তুগুলোর ওজনের যোগফল $\leq j$ হয়। তাহলে,

$$f(i,j) = \max_{m=0}^{k_i}\{f(i-1, j-w_i m) + v_i m\}$$

অর্থাৎ i তম বস্তু কতবার নিচ্ছি সেটার সবগুলো অপশন কনসিডার করতে হবে। আগেরটার কোড বুঝে থাকলে এটার কোড নিজেরই পারার কথা। এখানে টাইম কমপ্লেক্সিটি হবে $O(W \times \sum k_i)$

কিন্তু এইখানে সমস্যা হচ্ছে $\sum k_i$ এর মান অনেক বড় হতে পারে। আশার কথা হল এই প্রবলেমের এইটাই সবচেয়ে অপটিমাল সলিউশন না। $O(W \times \sum \log k_i)$ কমপ্লেক্সিটিতেও এই প্রবলেমটি সল্ভ করা সম্ভব।

আইডিয়াটি হচ্ছে প্রত্যেক k_i এর বাইনারি রিপ্রেজেন্টেশনকে ব্যবহার করা। একটি উদাহরণ দেখা যাক, ধর কোন এক টাইপের বস্তুর $(k_i, w_i, v_i) = (27, 13, 5)$। অর্থাৎ ঐ টাইপের বস্তু আছে 27 টি এবং তার ওজন 13 ও দাম 5। এখন 27 কে এইভাবে লেখা যায়:

$$27 = 11011_2 = 1111_2 + 1100_2 = (2^4 + 2^3 + 2^2 + 2^1 + 2^0) + 12$$

অর্থাৎ আমরা যদি $(27, 13, 5)$ বস্তুটির বদলে $(1, 13 \times 2^4, 5 \times 2^4)$, $(1, 13 \times 2^3, 5 \times 2^3)$, $(1, 13 \times 2^2, 5 \times 2^2)$, $(1, 13 \times 2^1, 5 \times 2^1)$, $(1, 13 \times 2^0, 5 \times 2^0)$ এবং $(1, 13 \times 12, 5 \times 12)$ বস্তুগুলোর ওপর ন্যাপস্যাক ডিপি চালাই তাহলে উত্তর চেঞ্জ হবে না, এর কারন হচ্ছে 2^4, 2^3, 2^2, 2^1, 2^0 এবং 12 দিয়ে 0 থেকে 27 পর্যন্ত সব সংখ্যা কে লেখা যায় (শুধু তাই নয়, সব সংখ্যাকে কেবল মাত্র একভাবেই লেখা যায়)। এইভাবে প্রতিটি বস্তুকে তার বাইনারি রিপ্রেজেন্টেশন অনুযায়ী ভেঙ্গে দিতে হবে। ভেঙ্গে দেওয়ার পর কিন্তু আমাদের আর 0-K Knapsack থাকছে না, 0-1 Knapsack হয়ে যাচ্ছে। কারণ ভেঙ্গে দেওয়ার পর প্রত্যেক বস্তুকে সর্বোচ্চ একবারই নেওয়া সম্ভব $(k_i = 1)$। অর্থাৎ ভেঙ্গে দেওয়ার পর আমাদের মোট বস্তু হবে $O(\sum \log k_i)$ টি। তাই 0-1 Knapsack এর কমপ্লেক্সিটি হবে $O(W \times \sum \log k_i)$। কোড:

```
/* nv contains the modified values of the objects
   nw contains the modified weights of the objects */
```

```
vector <long long> nv;
vector <long long> nw;

for(int i = 1; i <= n; i++) {
  for(int j = 31; j >= 0; j--) { // assuming k[i] fits into 32 bit integer
    if((1 << j) <= k[i]) {
      k[i] -= 1 << j;
      nv.push_back(1LL * (1 << j) * v[i]);
      nw.push_back(1LL * (1 << j) * w[i]);
    }
  }
  nv.push_back(1LL * k[i] * v[i]);
  nw.push_back(1LL * k[i] * w[i]);
}
/* Now we will perform 0/1 Knapsack on nw, nv array */
for(int j = 0; j <= W; j++) {
  f[j] = 0;
}
for(int i = 0; i < (int) nw.size(); i++) {
  for(int j = W; j >= 0; j--) {
    if(j >= nw[i]) {
      f[j] = max(f[j], f[j - nw[i]] + nv[i]);
    }
  }
}
/* answer will be found at f[W] */
```

মজার ব্যাপার হল এই প্রবলেমের $O(W \times \sum \log k_i)$ এর চেয়েও ভাল সলিউশন আছে। $O(nW)$ কমপ্লেক্সিটিতেও 0-K Knapsack সল্ভ করা সম্ভব। রিকারেন্সটি আবার লক্ষ্য করি:

$$f(i,j) = \max_{m=0}^{k_i}\{f(i-1, j-w_i m) + v_i m\} \quad (1)$$

কোনো ফিক্সড i এর জন্য $f(i,0), f(i,1), \ldots, f(i,W)$ এর মান যদি আমরা $O(W)$ তে বের করতে পারি, তাহলেই $O(nW)$ কমপ্লেক্সিটি হয়ে যাবে। এখন লক্ষ্য করি, $f(i,j)$ এর মান $f(i-1,j), f(i-1,j-w_i), f(i-1,j-2w_i), f(i-1,3w_i), \ldots$ মানগুলোর ওপর নির্ভর করে। অন্যভাবে বলা যায় $f(i,j)$ এর মান এমন সব $f(i-1,p)$ এর মানের ওপর নির্ভর

করে যাতে $p \equiv j \mod w_i$ হয়। এটাকে কাজে লাগিয়েই $O(W)$ তে কাজটি করা সম্ভব। আমরা $f(i,j)$ এর মান $0 \le j \le W$ এর জন্য একসাথে বের না করে w_i এর প্রত্যেক মডুলো ক্লাসের জন্য আলাদা ভাবে বের করতে পারি। বুঝানোর সুবিধার্তে ধরি,

$$g_m(i,j) = f(i, m + jw_i)$$

যেখানে $0 \le m < w_i$। এখন আমরা একটা ফিক্সড m এর জন্য $g_m(i,j)$ এর সকল মান বের করব, যেখানে $(0 \le m + jw_i \le W)$। (1) নং রিকারেন্সের সাহায্যে $g_m(i,j)$ কে এইভাবে লেখা যায়:

$$\begin{aligned} g_m(i,j) &= \max_{h=j-k_i}^{j} \{g_m(i-1,h) + (j-h)v_i\} \\ &= \max_{h=j-k_i}^{j} \{g_m(i-1,h) - hv_i\} + jv_i \end{aligned}$$

এখান থেকেই বুঝা যাচ্ছে $g_m(i-1,0), g_m(i-1,1) - v_i, g_m(i-1,2) - 2v_i, \ldots$ এর প্রতিটি $k_i + 1$ দৈর্ঘ্যের সাবঅ্যারের মিনিমাম ভ্যালু বের করতে পারলেই $g_m(i,j)$ এর সকল মান আমরা সহজেই বের করতে পারব। কোনো n দৈর্ঘ্যের অ্যারের প্রতিটি m দৈর্ঘ্যের সাবঅ্যারের মিনিমাম (বা ম্যাক্সিমাম) ভ্যালু $O(n)$ এই বের করা যায় (লিঙ্ক)। অর্থাৎ প্রত্যেক মডুলো ক্লাসের জন্য আমরা লিনিয়ার টাইমেই g_m এর মান বের করতে পারব। যেহেতু প্রত্যেকটি সংখ্যাই কেবলমাত্র একটি মডুলো ক্লাসের অন্তর্ভুক্ত তাই ওভারঅল কমপ্লক্সিটি হবে $O(W)$। তাই প্রত্যেকটি i এর জন্য $f(i,j)$ এর মান বের করতে $O(nW)$ কমপ্লেক্সিটি প্রয়োজন।

সাবসেট সাম:

এই সেকশনের সব জায়গায় সেট বলতে মাল্টিসেট বুঝান হবে। অর্থাৎ সেটে একই উপাদান একাধিক বার থাকতে পারে।

ন্যাপস্যাকের সবচেয়ে গুরুত্বপূর্ণ ভ্যারিয়েশন এটি। ধর তোমার কাছে n দৈর্ঘ্যের একটা অ্যারে a এবং একটি নাম্বার m দেওয়া আছে। তোমাকে বলতে হবে a এর নাম্বার গুলো ব্যবহার করে যোগফল m বানানো যায় কিনা।

অর্থাৎ $S = \{1, 2, 3, \ldots, n\}$ হলে এমন কোন সাবসেট T পাওয়া সম্ভব কিনা যাতে $T \subseteq S$ এবং $\sum_{i \in T} a_i = m$ হয়।

ধরি,

$$f(i,j) = \begin{cases} 1, & \text{যদি প্রথম } i \text{ টি সংখ্যা হতে যোগফল } j \text{ বানানো সম্ভব হয়,} \\ 0, & \text{সম্ভব না হয়.} \end{cases}$$

তাহলে,

$$f(i,j) = f(i-1,j) \vee f(i-1,j-a_i)$$

$\vee$ এখানে or অপারেটরটাকে বুঝাচ্ছে। তাহলে এই ডিপিটা ক্যালকুলেট করতে আমাদের $O(nm)$ টাইম ও $O(m)$ মেমরি লাগছে।

```
f[0] = 1;
for(int i = 1; i <= m; i++) {
  f[i] = 0;
}
for(int i = 1; i <= n; i++) {
  for(int j = m; j >= 0; j--) {
    if(f[j] >= i) {
      f[j] |= f[j - W[i]];
    }
  }
}
/* answer will be found at f[W] */
```

তবে এই সলিউশন কে অপটিমাইজ করার জন্য আরেকটা সস্তা অপটিমাইজেশন আছে। তা হল bitset ব্যবহার করা (লিঙ্ক)। bitset ব্যবহার করলে টাইম কমপ্লেক্সিটি দাড়ায় $O(\frac{nm}{64})$ এবং মেমোরি কমপ্লেক্সিটি দাড়ায় $O(\frac{m}{64})$।

```
bitset <100000> f; // assuming m < 10^5
f[0] = 1;
for(int i = 1; i <= n; i++) {
  f |= f << w[i];
}
/* answer will be found at f[W] */
```

ডাইনামিক সাবসেট সাম:

ধর সাবসেট সাম প্রবলেমটায় তোমাকে কিছু আপডেট আর কুয়েরিও দেওয়া হল। অর্থাৎ প্রত্যেক আপডেটে তোমাকে একটি সংখ্যা p দেওয়া হবে এবং তোমাকে সংখ্যাটাকে সেটে অ্যাড করতে হবে

অথবা সেট থেকে রিমুভ করতে হবে। প্রত্যেক কুয়েরিতে তোমাকে একটি সংখ্যা r দেওয়া হবে এবং তোমাকে বলতে হবে r সংখ্যাটিকে সেটের সংখ্যাগুলোর যোগফল হিসেবে লেখা যায় কিনা।

ধরা যাক মোট আপডেট ও কুয়েরি Q টি। তাহলে যদি আমরা Q বারই সাবসেট সাম-এর ডিপি টা নতুন করে আপডেট করি তাহলে কমপ্লেক্সিটি $O(\frac{Qnr_{\max}}{64})$ হয়ে যাচ্ছে। তবে এই প্রবলেমটি $O(Qr_{\max})$ টাইমেও করা সম্ভব, যেখানে $r_{\max}$ হল r এর ম্যাক্সিমাম ভ্যালু।

এর জন্য আমাদের ডিপি টাকে একটু চেঞ্জ করতে হবে। ধরি, $f(j) =$ সেটে যেসব উপাদান আছে তাদের কোনো সাবসেট নিয়ে কতভাবে j সংখ্যাটি বানানো যায়। তাহলে প্রত্যেক কুয়েরিতে $f(r) > 0$ কিনা তা চেক করলেই হচ্ছে আমাদের। আর যদি নতুন কোন নাম্বার অ্যাড বা রিমুভ করতে হয় তাহলে নরমাল সাবসেট সাম ডিপির মতই $f(j)$ এর মান আপডেট করা যায়। এখন সমস্যা হচ্ছে $f(j)$ মান অনেক বড় হয়ে যেতে পারে, এমনকি `long long` এও আটবে না। তাই $f(r)$ কে আমরা $\mod P$ ক্যালকুলেট করব যেখানে P র‍্যানডম কোন প্রাইম নাম্বার। এখন যদি $f(r) = 0$ হয়, এবং তারপরেও r কে যোগফল হিসেবে লেখা যাবে সেটির সম্ভাবনা নেয় বললেই চলে। (কেউ চাইলে ২-৩ টি `mod` ও ব্যবহার করতে পারে)। কোড:

```
const int r_max = 100000;
const int mod = 1000000007;
int f[r_max + 1];

void add(int p) {
  for(int i = r_max; i >= 0; i--) {
    f[i] += f[i - p];
    f[i] %= mod;
  }
}
void remove(int p) {
  for(int i = 0; i <= r_max - p; i++) {
    f[i + p] -= f[i];
    f[i] %= mod;
    if(f[i] < 0) f[i] += mod;
    /* be careful when u do substraction modulo something */
  }
}
bool query(int r) {
  return (f[r] > 0);
}
```

$O\left(s \times \sqrt{\frac{s}{64}}\right)$ সাবসেট সাম:

এখানে s সেটের সবগুলো সংখ্যার যোগফল বুঝাচ্ছে। যদি কোন সংখ্যা t এর থেকে বড় হয়, তাহলে আমরা নরমালি `bitset` দিয়ে ডিপি টা আপডেট করব, এটি করতে $O\left(\frac{s}{64} \times \frac{s}{t}\right)$ কমপ্লেক্সিটি লাগে (কারন t এর থেকে বড় সংখ্যা সর্বোচ্চ $\frac{s}{t}$ বার পাওয়া যাবে)। আর যদি t এর থেকে ছোট হয় তাহলে আমরা 0-k Knapsack এর মত ডিপি টাকে আপডেট করব। অর্থাৎ t এর থেকে ছোট কোন সংখ্যা কতবার আছে সেটা বের করে তার ওপর 0-k Knapsack এর মত ডিপি টাকে আপডেট করব। এ কাজটি করতে সর্বোচ্চ $O(st)$ কমপ্লেক্সিটি লাগে। $t = \sqrt{\frac{s}{64}}$ হলে টোটাল কমপ্লেক্সিটি দাড়ায়:

$$O\left(\frac{s}{64} \times \frac{s}{t} + st\right) = O\left(s \times \sqrt{\frac{s}{64}}\right)$$