Увод в динамичното програмиране

Основни понятия - стейт и транзишън

За да избегнем абстрактни и често неясни понятия, описанието на понятията “стейт”, “транзишън” и отговорът на въпроса “Какво решава динамичното програмиране?” ще изясним чрез пример.

Задача: По колко начина можем да получим x като сума от поредни хвърляния на зарове?

Заровете са стандартни - кубове с 6 страни и всяко едно от числата $1, 2 \cdots 6$ на по една от страните.

Въпреки че задачата ясно ни пита да изброим различни редици, съставени от числата от 1 до 6, с обща сума x, ние няма да направим това - дори за компютъра такъв алгоритъм би бил твърде бавен за по-големи стойности.

Вместо това ще ограничим задачата ни със следната идея: имаме да решим задачата за сума x, нека като допълнение си кажем, че последният хвърлен зар е ни е дал 1 точка. Изборът на точно 1 точка е прозволен - аргументът, който ще направим ще е същият за всички възможни $1 \leq a \leq 6$ избора. Броят на редиците със сума x и последно число 1 е абсолютно същият като броят на редиците със сума $x - 1$. Оказва се, че нашето “ограничение” ни превежда в напълно същата задача, но с различна сума. Това е и основната стъпка (транзишън), който ще ни послужи в решението - за да решим задачата за сума x, ще трябва първо да я решим за суми $x - 1, x - 2, \cdots x - 6$ и да се ползваме от тези решения. Изборът за точно каква сума ще решим задачата е “стейтът” на динамичното. Ползвайки напълно новия ни речник можем да кажем: За да решим задача чрез динамично програмиране, трябва да намерим какъв е стейтът му и какви са транзишъните помежду им.

За улеснение, нека с dp_x означаваме отговорът на задачата. Тогава транзишънът може да се изрази по следния начин:

$$dp_x = dp_{x-1} + dp_{x-2} + dp_{x-3} + dp_{x-4} + dp_{x-5} + dp_{x-6}$$

Важно е да забележим, че това твърдение не в сила за абсолютно всички стейтове - при $x < 6$, няма как да вземем dp_{-1} примерно. В конкретната задача $dp_1, dp_2, \cdots dp_6$ се оказват “базови стейтове” - тях ще пресметнем на ръка преди самото изпълнение на програмата. Оказва се, че $dp_x = 2^{x-1}$ за $x < 5$ (няма да се спираме на обосновка).

Имплементация на решението

Специално в тази задача доста лесно идва идеята за итеративно решение, което ползва само един for-цикъл, който обикаля по стейтовете.

```
for(int i = 7; i <= x; i ++){
    for(int a = 1; a <= 6; a ++){
        dp[i] += dp[i - a];
    }
}
```

В общия случай обаче трябва да внимаваме с такива решение - при тях е задължително преди да изчислим отговора за даден стейт, всичките стейтове, от които той зависи да бъдат изчислени предварително, а това не винаги е лесно за гарантиране.

Алтернативна имплементация на тази идея може да се възползва от рекурсия - наистина принципите “функция да вика себе си” и “задача да се решава с по-малки подобни” са много близки по идея.

```
int rec(int x){
    if(x <= 6) return pow(2, x);
    int ans = 0;
    for(int a = 1; a <= 6; a ++) ans += rec(x - a);
    return ans;
}
```

Това обаче ще удари time limit за по-големи x, защото един стейт ще се пресмята няколко пъти. Примерно:

$$rec(10) -> rec(9) -> rec(8) -> rec(7)$$
$$rec(10) -> rec(8) -> rec(7)$$
$$rec(10) -> rec(9) -> rec(7)$$
$$rec(10) -> rec(7)$$

са четири различни начина да стигнем до една и съща функцията, която във всеки случай ще дава един и същ отговор. Очевидно разхищение на ресурс. Наместо постоянно да правим еднакви сметки, ще запазим още първия път резултата в масив и ще маркираме този стейт за изчислен. Кодът би бил следния:

```
int rec(int x){
    if(x <= 6) return pow(2, x);
    if(used[x]) return dp[x];
    used[x] = true;
    int ans = 0;
    for(int a = 1; a <= 6; a ++) ans += rec(x - a);
    return dp[x] = ans;
}
```

Често срещани стейтове

От тук нататък ще изброим няколко често срещани стейта и техния резон. Това в никакъв случай не е изчерпателен списък и е напълно възможно да срещнете задачи, които не ползват кои и да е от тези стейтове.

1.3.1 Генериране на редици, изпълняващи дадени условия

Примерната задача би влязла тук - искаме да намерим броя редици, които изпълняват някакъв набор от условия, които могат да се следят чрез стейта. В горната задача условията биха били следните:

- $1 \leq a_i \leq 6$ за всяко i.
- $\sum_{i=1}^{k} a_i = x$, където k е дължината на редицата.

Две подобни задачи биха били НОИ1 2019 C3.seq и IATI B1 2017. Отново имаме да генерираме редици ограничение относно сумата на елементите си, но имаме в допълнение да следим също: 1) редиците да са нарастващи и 2) да са съставени от числа $\leq k$. В допълнение третата задача поставя ограничение на броя равни съседни елементи. Ограниченията върху входните числа са разумни и не се изискват никакви трикове - просто ще разширим стейта ни с допълнителни измерения.

Говорим за измерения на стейта, понеже като имплементация ще го запазваме в масив. Обикновено е объркващо да мислим за самата форма на масива, с който работим, защото може да се наложи да имаме четиримерно $dp[a][b][c][d]$ и тогава си е невъзможно умствено

да си представим хиперкуб.

По-"сложна" задача от този тип е ПТИ 2015 D1.robot. Тук редиците, които трябва да преброим са от точки в пространството, а не числа на числовата ос. Това не е толкова плашещо, ако осъзнаем, че просто можем да запомняме къде се намираме, ползвайки две измерения на стейта. Общо стейтът ще стане $dp[x][y][used]$, като с $used$ означаваме броя команди, които сме извършили. Транзишънът на това динамично ще е следният:

$$dp_{x,y,used} = dp_{x-1,y,used-1} + dp_{x,y-1,used-1} + dp_{x+1,y,used-1} + dp_{x,y+1,used-1}$$

игнорирайки базовите стейтове. Тук можем също да приложим трик за пестене на паметта: вместо да имаме имплементационно масив $dp[MAXX][MAXY][MAXK]$ ще отбележим, че в третото измерение винаги ползваме само "предишната стойност" - за да намерим някакъв път, ползващ 4 команди, ще се интересуваме само от пътища, ползващи 3 команди. Така можем да съкратим паметта до $dp[MAXX][MAXY][2]$, заделяйки си едното ниво на тримерния масив за стойностите, които в момента ще смятаме, а другото - за вече пресметнатите стойности.

1.3.2 Решаваме задачата за префикс от цялата задача

В тази секция ще попаднат задачи върху вече въведени редици, при които да решим същата задача за префикс от редицата улеснява цялостното решение. Да вземем НОИ2 2014 C1.seq - дадена ни е редица от цели числа и искаме да намерим максималната сума, ако ни е позволено да умножаваме елементите на редицата по 2 или по 3 с ограничението, че не трябва да има два съседни елемента, които са умножение по 3.

Вече подсказахме, че стейтът ще зависи от префикса върху който решаваме задачата. В решението, което ще разработим ще ползваме също второ измерение, което помни дали последният елемент е бил умножен по 2 или по 3. Транзишъните са естествени:

- $dp_{i,2} = \max(dp_{i-1,2}, dp_{i-1,3}) + 2 \times a_i$
- $dp_{i,3} = dp_{i-1,2} + 3 \times a_i$

Отново игнорираме базовите стейтове. В конкретния случай отново можем да оптимизираме паметта от $O(n)$ до $O(1)$, пазейки само последните няколко стойности на масива.

- **Задача:** НОИ1 2024 A3.maxseq
- **Задача:** SOA 2022 C2.expense

1.3.3 Задачи за монети, раници и суми

Най-простата и централната задача от тази секция е следната: дадена ни е редица от числа $a_1, a_2, \cdots, a_n$. Можем ли да представим x като сума на тези числа? Може да срещнете задачата с или без опцията едно число да се среща многократно - ще разгледаме случая, когато всеки елемент се взима най-много по веднъж, понеже е по-пипкав.

Тук отново имаме редица върху която работим и самата задача позволява да приложим динамично по префикса, както в горната секция. Новото нещо, което ще правим сега обаче е да отделим едно пространство за сумата - тя е централна за задачата и няма начин да си я съкратим от стейта. Така ще получим $O(nx)$ на брой стейта, което съвпада и с финалното решение на задачата. (По-бързо решение на задачата в този и вид не е открито, но съществуват такива за нейни конкретни случаи.) Финално получаваме стейт $dp_{i,s}$, където $1 \leq i \leq n$ и $0 \leq s \leq x$. Транзишъните отново излизат сравнително естествено:

$$dp_{i,s} = \max(dp_{i-1,s-a_i}, dp_{i-1,s})$$

Сумата s може да се получи с първите i числа по два начина - или сме я получили, ползвайки първите $i-1$ числа, или можем да получим сумата $s-a_i$ чрез първите $i-1$ числа и да

ползваме a_i, за да допълним сумата до s. Тук единственият нужен базов стейт е $dp_{0,0} = 1$ или с думи казано: можем да получим сума 0, още преди да сме ползвали каквито и да е елементи. Задачата се решава, когато $dp_{n,x} = 1$.

Отново можем да си спестим едно измерение на стейта, понеже префикса расте с по едни нов елемент всеки път. Тук обаче трябва и да внимаваме. Нека разгледаме една грешна имплементация на тази идея:

```
for(int i = 1; i <= n; i ++){
    for(int s = a[i]; s <= x; s ++){
        dp[s] = max(dp[s], dp[s - a[i]]);
    }
}
```

Проблемът тук е, че е възможно едно и също число да се насложи два пъти, което изрично забранихме в условието. Да кажем $dp[3] = 1$ и $a[i] = 2$ - тогава правилно ще маркираме $dp[5] = 1$, но когато проверяваме $s = 7$ отново ще маркираме тази сума за възможна, което не е задължително. Решението лесно се поправя - ще обикаляме s в намаляващ ред.

```
for(int i = 1; i <= n; i ++){
    for(int s = x; s >= a[i]; s --){
        dp[s] = max(dp[s], dp[s - a[i]]);
    }
}
```

Задача: Имаме редица от двойки (a_i, w_i) и константа W. Искаме да вземем $i_1 < i_2 < \cdots < i_k$ от тях, така че:

- $\sum_{j=1}^{k} w_{i_j} \leq W$
- да максимизираме $\sum_{j=1}^{k} a_{i_j}$

Промените спрямо миналата задача са минимални - сега вместо да знаем дали можем да получим дадена сума, ще пазим максималната печалба, която можем да получим за тази сума.

- **Задача:** НОИ2 2020 B1.double
- **Задача:** НОИ2 2017 C1.subsums
- **Задача:** НОИ2 2012 C2.sums

1.3.4 Подинтервали

Ще започнем с примерна задача ЗСИ 2015 C3.rod. Формално, задачата е следната: дадена ни е една редица $\{a_i\}_{i=1}^{n}$. Имаме правото за нея и всички следващи редици с поне 2 елемента, които ще получим да извършим следната операция - плащаме цена, равна на сумата на числата в редицата, избираме два съседни елемента и разделяме редицата на две нови, съответно префикса до левия елемент и суфикса до десния елемент. Каква е минималната цена, която можем да платим, докато приключим $n - 1$ изпълнения на горната операция?

В процеса на "чупенето" на редиците винаги ще получаваме подредици на оригиналната редица, а тях може да дефинираме ползвайки само индексът на най-левия и най-десния им елемент. В кода, динамичното ни ще изглежда по следния начин - $dp[l][r]$. Транзишънът между различните стейтове ще е "превеждане" на условието в езика на този стейт. Ако изберем два съседни елемента с индекси m и $m + 1$ между които ще чупим редицата, уравнението става:

$$dp_{l,r} = \min_{m=l}^{r-1}(dp_{l,m} + dp_{m+1,r}) + \sum_{i=l}^{r} a_i$$

Отговорът на задачата ще е $dp_{1,n}$.

Итеративната имплементация на тази идея е малко по-трудна за измисляне (рекурсивната си е директно прилагане на транзишъна и меморизация на стейтовете). Трудно ще накараме два for-цикъла, обикалящи по l и r да гарантират, че посещават (l, r) след като са посетили всички (l', r') за $l \leq l'$ и $r' \leq r$. Словом, трудно ще следим, че сме посетили всички по-къси подинтервали. За да решим този проблем ще забележим, че вместо да обикаляме двете краища, за да обходим всички подинтервали, можем да се задоволим само с единия край (l примерно) и дължината на интервала. Дължината ще обхождаме в нарастващ ред и това тривиално гарантира, че преди да посетя $(l, l + len - 1)$ съм посетил по-кратките му подинтервали.

```
for(int len = 2; len <= n; len ++){
    for(int l = 1, r = len; r <= n; l ++, r ++){
        for(int m = l; m < r; m ++){
            dp[l][r] = min(dp[l][m] + dp[m + 1][r] + sum[l, r], dp[l][r]);
        }
    }
}
```

- **Задача:** IATI 2017 B2.game
- **Задача:** НОИ2 2018 A2.palc

1.3.5 Най-дълга нарастваща подредица

Задача: Дадена е редица $a_1, a_2, \cdots a_n$. Какво е максималното k, за което съществува редица $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, за която $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$?

Ще предложим две решения на задачата – по-лесно, за $O(n^2)$ време и по-трудно за $O(n \log n)$ време.

Вече трябва да е ясна идеята за $O(n^2)$ - тя се основава на стейт за префикс от цялата редица. $dp[i]$ е максималната дължина на нарастваща подредица, която завършва на позиция i. Транзишънът си е естествен, но все пак ще го запишем.

$$dp_i = \max_{j<i, a_j<a_i} (dp_j) + 1$$

За да преминем към сложност $O(n \log n)$ имаме две опции за оптимизации - едната е директно да оптимизираме горния транзишън чрез сегментно дърво, в което на позиция x записваме дължината на най-дългата нарастваща подредица, която има последно число със стойност x. По-интересно обаче е, че можем въобще да не се занимаваме със структури за данни.

Първо ще отбележим, че можем да направим "смяна" на стейта. В момента стейтът ни може да се опише по следния начин:

$$dp[position] = max\ length$$

Обаче можем да работим също със следния стейт:

$$dp[length] = min\ value$$

За всяка дължина ще запазваме минималното a_i, което може да бъде последно число в нарастваща подредица с такава дължина. Отново можем да приложим $O(n^2)$ идея, така че само с тази смяна не сме постигнали много.

За да финализираме решението трябва да забележим, че за всяко x винаги е изпълнено

$dp[x] \leq dp[x+1]$, тоест динамичното ни дефинира монотонно растяща функция. Доказателството ще направим чрез допускане на противното: Нека има x_0, за който $dp[x_0] > dp[x_0+1]$. Нека си представим редицата с дължина x_0+1, която завършва на стойност $dp[x_0+1]$ и да махнем последното и число. Така получаваме нова нарастваща редица с дължина x_0, която завършва на стойност $a < dp[x_0]$. По дефиниция $dp[x_0] < a$. Противоречие.

Винаги когато намерим монотонна функция в задачата, която решаваме, можем да се опитаме да ползваме двоично търсене. Тук можем да го ползваме, за да намерим най-голямото l, такова че $dp[l] < a$, тоест дължината на най-дългата подредица, която можем да удължим с числото a. Ще извършим това за всяка позиция от оригиналната редица. Отговорът на задачата ще бъде най-голямото l, за което сме записали нещо в $dp[l]$.

```
for(int i = 1; i <= n; i ++){
    int l = 1, r = i, ans = 1;
    while(l <= r){
        int m = (l + r) / 2;
        if(dp[m] < a[i]){
            ans = m;
            l = m + 1;
        }
        else{
            r = m - 1;
        }
    }
    dp[ans] = min(dp[ans], a[i]);
}
```

Идеята, да поставим дължината на подредицата в стейта, може да генерализираме до "ще поставим някакво важно свойство за нарастващата редица в стейта". Примерно, ако за задачата е важно подредицата да има сума от елементите си $\leq S$, ще направим следния стейт:

$$dp[sum\ of\ elements] = min\ value$$

Забележете обаче, че можем да приложим двоично търсене само защото $dp[x]$ тук е доказуемо монотонно растяща. В примера със сумата това не е вярно – контрапример е редицата $3, 2, 2, 2, 100$. Тогава накрая ще получим $dp[3] = 3$, $dp[6] = 2$ и $dp[100] = 100$.

- **Задача:** НОИ2 2019 B1.estet
- **Задача:** SOA 2020 C3.tower

Допълнителни материали

- atcoder dp contest
- cses problemset
- Първо от три части на лекция "Динамично оптимиране" на Александър Георгиев
- cp-algo статия за динамичното

Автор: Иван Лупов