مسئله کوله پشتی : یکی از مسائل معروف حوزه بهینه سازی

اگر شما با بهینه سازی سروکار داشته باشید، مثلا دانشجوی رشته کامپیوتر یا تحقیق در عملیات یا بهینه سازی باشید ، حتما یکی از مواردی که در خصوصش بحث میشود مسئله کوله پشتی یا Knapsack Problem هست. مسئله کوله پشتی یا knapsack Problem را اگر بخواهیم خیلی ساده توضیح بدهیم ، باید با یک مثال ساده این کار رو انجام بدیم.

دزدی که هوشمندانه دزدی میکند!!

یک دزد رو در نظر بگیرید که یک کیسه دست گرفته است و وارد خانه ای شده است ، حالا میخواهد کیسه خود را از اشیا و لوازم قیمتی پر کند و پا به فرار بگذارد.کیسه همراه اقای دزد، یک وزن مشخص را میتواند تحمل کند مثلا 50 کیلو، و خب وسایلی که در خانه هست هم وزن های مختلفی دارند. قیمت هر کدوم هم متفاوت هست.

دزد قصه ما دنبال این هست که وسایلی رو برداره که وزنشون کمتر و قیمتشون بیشتر باشه تا بتواند سود بیشتری از این دزدی برده باشد.

مدل سازی ریاضی مسئله کوله پشتی

خب مثال رو به صورت زیر توصیف میکنیم تا بتوانیم به یک مدل ریاضی برسیم

- اشیا را از 1 تا n شماره گذاری میکنیم. xi نشان دهنده شی i ام می باشد
- یک آرایه در نظر میگیریم و وزن اشیا (w) را در آن مینویسیم : ارایه وزنها (wi نشان دهنده وزن شی i ام است)
- یک آرایه دیگر در نظر میگیریم و ارزش یا قیمت اشیا (v) را در آن قرار میدهیم : آرایه ارزشها (vi نشان دهنده ارزش شی i ام است)

خب مسئله کوله پشتی 0 و 1 را میتوانیم بصورت زیر فرموله کنیم:

- مقدار $\sum_{i=1}^{n} v_i x_i$ را بیشینه کنید.
- به‌طوری‌که $\sum_{i=1}^{n} w_i x_i \leqslant W, \quad x_i \in \{0,1\}$

در مسئله کوله پشتی 0 و 1 ، یک شی یا داخل کیسه قرار میگیرد یا قرار نمیگیرد. و فرمول فوق به بیان ساده میگوید که دنبال بیشینه کردن ارزش اشیای انتخابی هستیم با این شرط که مجموع وزن اشیای انتخاب شده بیشتر از W (وزن قابل تحمل توسط کیسه) نباشد.

اهمیت مسئله کوله پشتی چیست؟

مسائل مختلفی در دنیای واقعی هستند که به نوعی مشابه مسئله کوله پشتی هستند و یافتن روش بهینه برای حل مسئله کوله پشتی، منجر به یافتن پاسخ بهینه برای آنها نیز میشود بعنوان مثال مسئله تخصیص منابع در شبکه ابری، انتخاب سهام در بورس، رمزنگاری در امنیت و مسائل مختلف دیگر از این دست مسائلی هستند که در کلاس مسئله کوله پشتی قرار میگیرند.

روشهای حل کوله پشتی

در طی سالهای اخیر، افراد زیادی برای حل بهینه مسئله کوله پشتی تلاش کرده اند ، و روشهای مختلفی نیز ابداع شده است ، از روش جستجوی عقبگرد گرفته ، تا روش جستجوی حریصانه و روش برنامه نویسی پویا، تا اخیرا که با معرفی الگوریتم های فراابتکاری و بهینه سازی ، استفاده از این روشها برای حل کوله پشتی همه گیر شده است . روشهایی مانند الگورتیم ژنتیک ، الگوریتم pso ، تا الگوریتم های جدید تر مانند الگوریتم ملخ، الگوریتم گرگ خاکستری ، الگوریتم شاهین هریس و غیره.

حل مساله کوله پشتی

هدف، قرار دادن این اشیا در کوله‌پشتی با ظرفیت W به صورتی است که مقدار ارزش بیشینه حاصل شود. به بیان دیگر، دو آرایه صحیح val[0..n-1] و wt[0..n-1] وجود دارند که به ترتیب نشانگر مقادیر و وزن‌های تخصیص داده شده به n عنصر هستند. همچنین، یک عدد صحیح W نیز داده شده است که ظرفیت کوله پشتی را نشان می‌دهد. هدف، پیدا کردن زیرمجموعه‌ای با مقدار بیشینه val[] است که در آن، مجموع وزن‌ها کوچک‌تر یا مساوی W باشد. امکان خرد کردن اشیا وجود ندارد و باید یک شی را به طور کامل انتخاب کرد و یا اصلا انتخاب نکرد. این گونه از مساله کوله پشتی را، «مساله کوله پشتی ۱-۰» می‌گویند.

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

یک راهکار ساده برای حل این مساله، در نظر گرفتن همه زیر مجموعه‌های ممکن از اشیا و محاسبه وزن و ارزش کلی هر یک از آن‌ها است. سپس، از میان همه زیرمجموعه‌ها مواردی انتخاب می‌شوند که وزن کلی آن‌ها کمتر یا مساوی W باشد. در نهایت، از میان زیرمجموعه‌های دارای وزن کمتر یا مساوی W، زیرمجموعه‌ای با ارزش بیشینه انتخاب می‌شود.

1. زیر ساختار بهینه

برای در نظر گرفتن همه عناصر، دو حالت برای هر عنصر وجود دارد. در حالت اول، عنصر در زیر مجموعه بهینه قرار می‌گیرد و در حالت دوم، عنصر در زیر مجموعه بهینه قرار نمی‌گیرد. بنابراین، ارزش بیشینه‌ای که می‌توان از n عنصر به دست آورد، برابر با بیشینه دو ارزش زیر است:

1. ارزش بیشینه حاصل شده از n-1 عنصر و وزن W (به استثنای عنصر nاُم)
2. ارزش عنصر nاُم به‌علاوه ارزش بیشینه به دست آمده از n-1 عنصر و وزن W منهای وزن عنصر nاُم

اگر وزن عنصر nاُم بزرگـتر از W باشد، این عنصر نمی‌تواند در کوله پشتی قرار بگیرد و تنها حالت ۱ امکان‌پذیر است.

2. زیرمساله‌های هم‌پوشان

حل مساله کوله پشتی ۰-۱ در پایتون

```
#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)
```

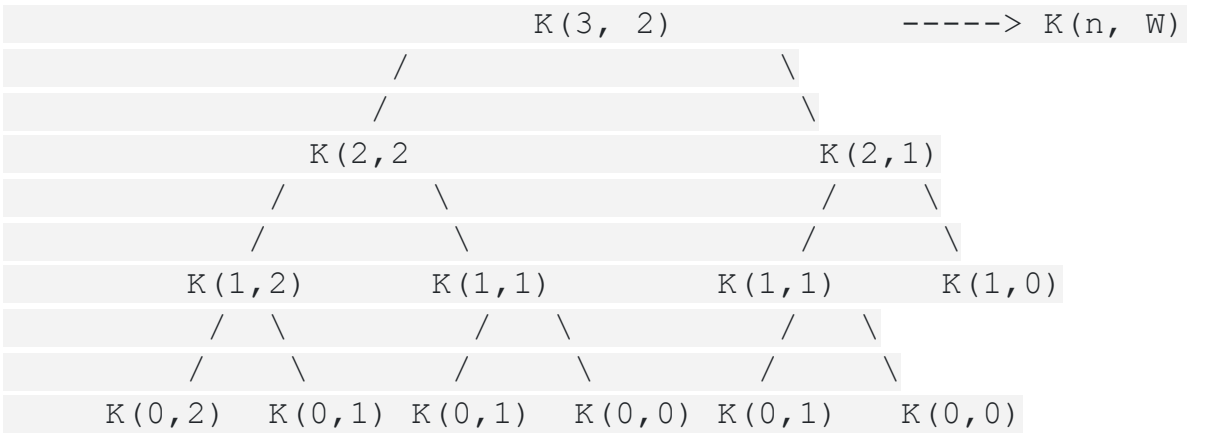
خروجی قطعه کد بالا به صورت زیر است.

```
220
```

حل مساله کوله پشتی با برنامه‌نویسی پویا

شایان ذکر است که تابع بالا، زیرمساله‌های مشابه را بارها و بارها حل می‌کند. برای مثال، زیر درخت بازگشتی k دو بار محاسبه می‌شود. پیچیدگی زمانی این روش بازگشتیِ ساده به صورت نمایی و از درجه O(2n) است.

```
In the following recursion tree, K() refers to knapSack().
The two parameters indicated in the following recursion tree are
n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}
```



با توجه به آنکه زیرمساله‌ها مجددا محاسبه می‌شوند، این مساله دارای خصوصیت زیرمساله‌های هم‌پوشان است. بنابراین، مساله کوله‌پشتی ۰-۱ دارای هر دو خصوصیت مسائل «برنامه‌نویسی پویا» (Dynamic Programming | DP) است. همچون دیگر مسائل برنامه‌نویسی پویا (DP)، می‌توان از محاسبه مجدد زیرمساله‌های مشابه با ساخت یک آرایه موقت K[][] به صورت پایین به بالا، اجتناب کرد. در ادامه، پیاده‌سازی راه حل این مساله با استفاده از برنامه‌نویسی پویا ارائه شده است.

حل مساله کوله پشتی ۰-۱ با برنامه‌نویسی پویا در پایتون

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem

# Returns the maximum value that can be put in a knapsack of capacity W

def knapSack(W, wt, val, n):

    K = [[0 for x in range(W+1)] for x in range(n+1)]

     # Build table K[][] in bottom up manner

    for i in range(n+1):

        for w in range(W+1):

            if i==0 or w==0:

                K[i][w] = 0

            elif wt[i-1] <= w:

                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])

            else:

                K[i][w] = K[i-1][w]

  return K[n][W]



# Driver program to test above function
```

```
val = [60, 100, 120]

wt = [10, 20, 30]

W = 50

n = len(val)

print(knapSack(W, wt, val, n))
```

مسئله کوله پشتی
ابو الفضل شماخی