Shopping Offers

In LeetCode Store, there are some kinds of items to sell. Each item has a price.

However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.

You are given the each item's price, a set of special offers, and the number we need to buy for each item. The job is to output the lowest price you have to pay for **exactly** certain items as given, where you could make optimal use of the special offers.

Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer.

You could use any of special offers as many times as you want.

Example 1:

```
Input: [2,5], [[3,0,5],[1,2,10]], [3,2]
Output: 14
Explanation:
There are two kinds of items, A and B. Their prices are $2 and $5 respectively.
In special offer 1, you can pay $5 for 3A and 0B
In special offer 2, you can pay $10 for 1A and 2B.
You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $
4 for 2A.
```

Example 2:

```
Input: [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]
Output: 11
Explanation:
The price of A is $2, and $3 for B, $4 for C.
You may pay $4 for 1A and 1B, and $9 for 2A ,2B and 1C.
You need to buy 1A ,2B and 1C, so you may pay $4 for 1A and 1B (special offer #1), an
d $3 for 1B, $4 for 1C.
You cannot add more items, though only $9 for 2A ,2B and 1C.
```

Note:

1. There are at most 6 kinds of items, 100 special offers.
2. For each item, you need to buy at most 6 of them.
3. You are **not** allowed to buy more items than you want, even if that would lower the overall price.

Solution 1

The basic idea is to pick each offer, and subtract the needs. And then compute the price without the offer.
Pick whichever is minimum.

Edit :) much appreciated if someone can shorten the code with Java8 :)

```java
public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Int
eger> needs) {
    int result = Integer.MAX_VALUE;
    //apply each offer to the needs, and recurse
    for(int i = 0; i < special.size(); i++) {
        List<Integer> offer = special.get(i);
        boolean invalidOffer = false;
        for(int j = 0; j < needs.size(); j++) { // subtract offer items from needs
            int remain = needs.get(j) - offer.get(j);
            needs.set(j, remain);
            if(!invalidOffer && remain < 0) invalidOffer = true; // if offer has mo
re items than needs
        }
        if(!invalidOffer) { //if valid offer, add offer price and recurse remaining
needs
            result = Math.min(result, shoppingOffers(price, special, needs) + offer
.get(needs.size()));
        }
        for(int j = 0; j < needs.size(); j++) { // reset the needs
            int remain = needs.get(j) + offer.get(j);
            needs.set(j, remain);
        }
    }
    // choose b/w offer and non offer
    int nonOfferPrice = 0;
    for(int i = 0; i < needs.size(); i++) {
        nonOfferPrice += price.get(i) * needs.get(i);
    }
    return Math.min(result, nonOfferPrice);
}
```

`UPDATE 1` **:** For the below test case, we get time limit exceeded since it's exponential. TLE due to needs=30+.
I've requested admins to add this testcase.

```
[2,5]
[[1,0,5],[1,2,10]]
[39,39]
```

~~So I made some optimization to reduce the recursive calls, by precomputing the number of times offer can be applied.~~ See **UPDATE 3**, there's an example that breaks this greedy optimization.

```
    public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List
<Integer> needs) {
        int result = Integer.MAX_VALUE;
        //apply each offer to the needs, and recurse
        for(int i = 0; i < special.size(); i++) {
            List<Integer> offer = special.get(i);
            boolean invalidOffer = false;
            int offerCount = Integer.MAX_VALUE; // number of times offer can be appl
ied
            for(int j = 0; j < needs.size(); j++) { // pre-compute number of times
offer can be called
                int remain = needs.get(j) - offer.get(j);
                if(!invalidOffer && remain < 0) invalidOffer = true; // if offer ha
s more items than needs
                if(offer.get(j) > 0)
                offerCount = Math.min(offerCount, needs.get(j)/offer.get(j));
            }
            for(int j = 0; j < needs.size(); j++) { // subtract offer items from ne
eds
                int remain = needs.get(j) - offer.get(j) * offerCount;
                needs.set(j, remain);
            }
            if(!invalidOffer) { //if valid offer, add offer price and recurse remai
ning needs
                result = Math.min(result, shoppingOffers(price, special, needs) + (
offerCount * offer.get(needs.size())));
            }

            for(int j = 0; j < needs.size(); j++) { // reset the needs
                int remain = needs.get(j) + offer.get(j) * offerCount;
                needs.set(j, remain);
            }
        }

        // choose b/w offer and non offer
        int nonOfferPrice = 0;
        for(int i = 0; i < needs.size(); i++) {
            nonOfferPrice += price.get(i) * needs.get(i);
        }
        return Math.min(result, nonOfferPrice);
    }
```

`UPDATE 2:` I think OJ is breaking with the below test case. My code handles it though. Expected output is 8000, since it has two items of 1$ each. I've requested to add the test case. Also, another assumption is that result doesn't exceed Integer.MAX_VALUE. @administrators

```
[1,1]
[[1,1,2],[1,1,3]]
[4000,4000]
```

`UPDATE 3:` From @Red_Eden 's thought, I found a test case that breaks my optimization. OJ is missing this test as well. My solution gives answer = 6, but actual is pick one offer just once = 4.

```
[500]
[[2,1],[3,2],[4,1]]
[9]
```

written by jaqenhgar original link here

Solution 2

```
    public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List
<Integer> needs) {
        Map<List<Integer>, Integer> dp = new HashMap<>();
        List<Integer> allZero = new ArrayList<>();
        for(int i=0;i<needs.size();i++) {
            allZero.add(0);
        }
        dp.put(allZero, 0);
        return dfs(needs, price, special, dp);
    }
    private int dfs(List<Integer> needs, List<Integer> price, List<List<Integer>> sp
ecial, Map<List<Integer>, Integer> dp) {
        if(dp.containsKey(needs)) return dp.get(needs);
        int res = Integer.MAX_VALUE;
        for(List<Integer> s : special) {
            List<Integer> needsCopy = new ArrayList<>(needs);
            boolean valid = true;
            for(int i=0;i<needs.size();i++) {
                needsCopy.set(i, needsCopy.get(i) - s.get(i));
                if(needsCopy.get(i) < 0) {
                    valid = false;
                    break;
                }
            }
            if(valid) {
                res = Math.min(res, s.get(needs.size()) + dfs(needsCopy, price, spec
ial, dp));
            }
        }
        //What if we do not use specials? specials can be deceiving,
        //perhaps buying using regular prices is cheaper.
        int noSpecial = 0;
            for(int i=0;i<needs.size();i++) {
                noSpecial += needs.get(i) * price.get(i);
            }
        res = Math.min(res, noSpecial);

        dp.put(needs, res);
        return res;
    }
}
```

written by chao59 original link here

Solution 3

it seems very like the dynamic programming problem. But when I solve the dp problem such like knapsack problem. I need the end of this problem, i.e. the volume of knapsack. If I know this, then the problem totally a knapsack problem. luckily, I get this from

1. There are at most 6 kinds of items, 100 special offers.
2. For each item, you need to buy at most 6 of them.
   Then I add to 6 item for every input argument.
   This code have O(special offers size) time complex. When the input is small, it's not the best time complex. And it also not very general.

```
int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>& ne
eds)
{
 int n = price.size();
 for (int i = n; i < 6; i++)
 {
  price.push_back(0);
  needs.push_back(0);
 }
 for (int i = special.size() - 1; i >= 0; i--)  // fill special to 6 items
 {
  int t = special[i][n];
  special[i][n] = 0;
  for (int j = n + 1; j < 7; j++)
   special[i].push_back(0);
  special[i][6] = t;
 }
 vector<int> temp(7);
 for (int i = 0; i < 6; i++)   //treat every item as a special offer only have itse
lf with its price
 {
  for (int j = 0; j < 6; j++)
  if (j != i)
   temp[j] = 0;
  else
   temp[j] = 1;
  temp[6] = price[i];
  special.push_back(temp);
 }
 int dp[7][7][7][7][7][7], m = special.size();
 //memset(dp, INT_MAX, 7 * 7 * 7 * 7 * 7 * 7);
       //I don't know why this memset useless, it just put every element to -1
 for (int j = 0; j < 7; j++)        // so I have initial it using loop
 {
  for (int k = 0; k < 7; k++)
  for (int p = 0; p < 7; p++)
  for (int q = 0; q < 7; q++)
  for (int r = 0; r < 7; r++)
  for (int s = 0; s < 7; s++)

   dp[j][k][p][q][r][s] = INT_MAX;
 }
 dp[0][0][0][0][0][0] = 0;
 for (int i = 0; i < m; i++)  // then it just a dynamic programming problem
 {
  for (int j = special[i][0]; j < 7; j++)
  for (int k = special[i][1]; k < 7; k++)
  for (int p = special[i][2]; p < 7; p++)
  for (int q = special[i][3]; q < 7; q++)
  for (int r = special[i][4]; r < 7; r++)
  for (int s = special[i][5]; s < 7; s++)
  {
          int tt = dp[j-special[i][0]][k-special[i][1]][p-special[i][2]]
                              [q-special[i][3]][r-special[i][4]][s-special[i][5]];
          if (tt != INT_MAX)
           dp[j][k][p][q][r][s] = min(dp[j][k][p][q][r][s], tt + special[i][6]);
  }
 }
 return dp[needs[0]][needs[1]][needs[2]][needs[3]][needs[4]][needs[5]];
}
```

written by donggua_fu original link here