Knapsack problem hay còn gọi là bài toán sắp xếp balo, là 1 trong những bài toán kinh điển và phổ biến. Bài toán được phát như sau:

Chúng ta có một balo với sức chứa nhất định có thể chứa được “w” kg vật. Ta có một số lượng các đồ vật, mỗi đồ vật có 2 chỉ số “weight” đại diện cho trọng lượng vật và “value” đại diện cho giá trị của vật. Chúng ta cần sắp vào balo 1 số lượng các vật sao cho tổng trọng lượng không vượt quá sức tải balo và tổng giá trị phải cao nhất có thể. Biết ta không thể chia vật ra làm các phần nhỏ rồi mang theo, chỉ có 2 lựa chọn hoặc là mang theo, hoặc là bỏ lại. Cùng 1 đồ vật không thể xuất hiện 2 lần.

Với bài toán nêu trên, 1 phương pháp giải dễ dàng nhất có thể nghĩ đến đó là tiếp cận theo chiến thuật quay lui, chúng ta sẽ xét tất cả các tổ hợp có thể có và chọn ra tổ hợp đúng nhất để làm đáp án của bài toán. Tuy nhiên vì xét hết tổ hợp của vật nên chi phí để giải bài toán sẽ rất cao, không phù trong trường hợp quy mô dữ liệu bài toán lớn (có quá nhiều đồ vật cần xem xét). Sau đây là ví dụ về hướng tiếp cận theo tư tưởng quay lui. Và sau quay lui, ta sẽ đi đến xem xét 1 giải pháp khác giúp giải quyết bài toán hiệu quả hơn.

Độ phức tạp có thể lên đến O(2^n), chi phí rất cao. Nếu chú ý, có thể thấy đoạn chương trình trên sẽ có khả năng xét trùng lặp nhiều tổ hợp giống nhau. Ví dụ có [[2, 3],[3, 4], [2, 3], [2, 3]]. Với số đứng trước là weight, số đứng sau là value, cách giải trên sẽ xét tổ hợp ([2,3], [2,3]) lặp lại. Như vậy, chúng ta có thể cải thiện chi phí giải bài toán này bằng cách tránh xét lại những trường hợp đã được xét, để làm điều đó, chúng ta sẽ có một giải pháp theo hướng tiếp cận quy hoạch động (nguồn: geeksforgeeks.org)

B1: Đánh số các vật từ 1 đến n. Tạo 1 mảng 2 chiều có “n+1” dòng và “m+1” cột với m là sức tải của túi. Dòng thứ “i” đại diện cho tập hợp các vật trong khoảng “1->i” có thể mang, cột thứ “j” đại diện cho các mức chứa của túi (trong khoảng 1->m kg). Vậy giá trị nằm ở ô [i][j] đại diện cho value lớn nhất của các vật trong khoảng “1->i” mà “j” kg có thể chứa. Gọi mảng này là dp.

B2: Ở dòng thứ nhất (i=0) có nghĩa là ta không chọn vật nào để bỏ vào túi nên tất cả các cột được khởi tạo giá trị 0. Cột thứ nhất (j=0) có nghĩa là 0 kg nên không thể chứa vật nào, ta khởi tạo tất cả giá trị ở cột này là 0.

B3: Ở mỗi lần xét, ta tận dụng kết quả được xét từ trước đó để tính ra đáp án cho lần xét ở hiện tại. Nói cách khác là chia bài toán từ tìm cách bỏ các vật vào balo sao cho tối ưu nhất về mặt trị giá thành các bài toán con, ở các mức chứa khác nhau của balo thì có thể mang trị giá thế nào.

Ở lần xét hiện tại, ta đang xét vật thứ i, ta có 2 lựa chọn hoặc là mang theo i hoặc không. Vì vậy ta cần phải lấy max giữa 2 giá trị, tổng val gồm cả value của i và không gồm value của i.

Trước hết, ta kiểm xem weight của vật i có vừa balo không (<=bagCapacity), nếu nhỏ hơn thì ta khởi tạo các biến:

* MaxValWithoutCurrItem: tổng trị giá của các vật phẩm mà không có vật phẩm thứ i.
* MaxValWithCurrItem: tổng trị giá của các vật phẩm gồm cả vật phẩm i.
* CurrWeight: trọng lượng của vật phẩm i.

Ta tính trọng lượng còn lại của túi nếu bỏ vật i vào túi remainingSlot=capacity(mức chứa đang được xét)-currWeight, tiếp đến maxValWithCurrItem=(value[i] + dp[i-1][remainingSlot]).

Bước này ta đang tính tổng trị giá của vật hiện tại với tổng trị giá có thể lấp thêm vào khoảng còn lại của túi (giá trị của dòng phía trên ở cột remainingSlot tức tổng value có thể mang của các vật trong khoản từ 1 đến i-1 ở khoảng kg bằng đúng với phần còn trống của túi ở hiện tại). Vậy giá trị ở ô đang xét là max giữa maxValWithCurrItem với maxValWithoutCurrItem. (maxValWithoutCurrItem=dp[i-1][j]).

B4: Cuối cùng, kết quả nằm ở ô [n][m] ( n là số lượng vật, m là sức tải của balo ) vì đó là tổng trị giá các vật trong khoảng 1->n mà ta có thể bỏ vào balo ở mức m kg (sức tải tối đa của balo).