1. Unlimited Knapsack problem

* Algorithm Description

Build a table where each row represents a size to consider in A and each column is the weight to consider. A particular cell in the table will hold true or false. True for if the sizes considered so far with the current weight can fit exactly and false otherwise. The table will have trues where any size is being considered with no weight and false where we are comparing no size with any weight. This will form the base cases for the dynamic programming algorithm. The table will be filled by asking if the current size we are considering can fit the weight we are considering. If it can't then the value from the row above will be used (as if we didn't use the item). Otherwise the current cell will be populated with a true if the cell on the below row is true or if a cell in the below row at the weight mod the current size is true. This is because the modulus operator gives us the remainder after division so if the remainder is 0 that means the current size is a factor of the current weight and therefore perfectly fits. If at any time we find that the column that represents the m value passed in has true in it then we know that we can be done calculating because we found some combination of sizes that perfectly fit m.

Subproblems:

The problem is broken down to worrying about filling out one row at a time.

Dependency Relation:

R[i][w] = | R[i-1][w] if size > w

| R[i-1][w] or R[i-1][w % size] otherwise

* Pseudocode

def unlimited\_knapsack(m, A):

R = [[True if j == 0 else False for j in range(m + 1)] for \_ in range(len(A) + 1)]

for i in range(1, len(A)+1):

size = A[i - 1]

for w in range(1, m+1):

if size > w:

R[i][w] = R[i-1][w]

else:

R[i][w] = R[i-1][w] or R[i-1][w % size]

if R[i][m]:

return True

return R[len(A)][m]

* Correctness

Every cell represents if the size or the size combined with previous sizes can add up to the current weight so if at anypoint the last column gets a true then we know that there is a combination of sizes that add up to m.

* Time Analysis

Filling out the table would take O(nm) time.

1. Maximized subset Knapsack problem

* Algorithm Description

Given a value m and an array of items where each item has item.weight and item.value. The algorithm finds the best value given the max amount of weight m specifies. A table is created where the first row and column are given the value of 0. This is because we are considering 0 sizes or 0 amount of weight. A particular cell represents the maxim value that can be had with the items seen so far with the current weight. When the table is filled the question of what the best sum of values for the given m value will be answered.

Subproblems:

The problem is broken down to worrying about filling out one row at a time.

Dependency Relation:

R[i][w] = | R[i-1][w] if item.weight > w

| max(R[i-1][w], item.value + R[i-1][w-item.weight]) otherwise

* Pseudocode

def knapsack(m, A):

R = [[0] \* (m + 1) for \_ in range(len(A) + 1)]

for i in range(1, len(A)+1):

item = A[i - 1]

for w in range(1, m+1):

if item.weight > w:

R[i][w] = R[i-1][w]

else:

R[i][w] = max(R[i-1][w], item.value + R[i-1][w-item.weight])

return R[len(A)][m]

* Correctness

First the initial row and column of 0s is given. Then the matrix is filled out row by row where each row represents the item being considered. Each column represents the weight that is being considered. After the matrix is filled out the best value for the max weight should be known.

* Time Analysis

Filling out the table would take O(nm) time.

1. Maximum-sum common subsequence

* Algorithm Description

Build a table where each entry will represent the largest sum that can be made with the items considered with the current weight. The first row and column of the table will have 0s because if no item or no weight is being considered then a 0 will go into the cell.

Subproblems:

The problem is broken down to worrying about filling out one row at a time.

Dependency Relation:

R[i][j] = | max(R[i-1][j-1] + A[i-1], R[i-1][j-1]) if A[i-1] == B[j-1]

| max(R[i-1][j], R[i][j-1]) otherwise

* Pseudocode

def maximum\_sum\_common\_subsequence(A, B):

R = [[0] \* (len(B)+1) for \_ in range(len(A)+1)]

for i in range(1, len(A)+1):

for j in range(1, len(B)+1):

if A[i-1] == B[j-1]:

R[i][j] = max(R[i-1][j-1] + A[i-1], R[i-1][j-1])

else:

R[i][j] = max(R[i-1][j], R[i][j-1])

return R[len(A)][len(B)]

* Correctness

At the end the last entry in the table will be the largest sum of a common subsequence.

* Time Analysis

Filling out the table would take O(nm) time.

1. Longest monotonically increasing subsequence

* Algorithm Description

R is a list that will keep track of how many items left of a particular item are less than it. After R is filled out we can backtrack to find out what items make up the longest increasing subsequence.

Subproblems:

The problem is broken down to working on how many numbers are smaller than a particular entry in A.

Dependency Relation:

R[i] = | R[j] + 1 if A[j] < A[i] and R[j] + 1 > R[i]

| 1 otherwise

* Pseudocode

def longest\_monotonically\_increasing\_subsequence(A):

R = [1] \* len(A)

for i in range(1, len(A)):

for j in range(0, i):

if A[j] < A[i] and R[j] + 1 > R[i]:

R[i] = R[j] + 1

S = []

m = max(R)

while m > 0:

for i in range(len(R)-1, -1, -1):

if m == 0:

break

if R[i] == m:

S.append(A[i])

m -= 1

m -= 1

S.reverse()

return S

* Correctness

The problem has been baked down to counting how many numbers are smaller than a particular entry. R keeps track of the number of items less than a particular entry. Thus after R is filled out the longest increasing subsequence can be found by finding the entry with the maximum number of items less than itself. Then finding the next largest and the next all the way until we have found all the items.

* Time Analysis

Every entry has to be checked with every other entry and there are two for loops going over A thus the time complexity is O(n^2)