v

**Problem 1:**

Note: I could not get my code working for this problem. I’ll write a paragraph of what I’m trying to achieve before each subproblem.

**A.**

The recurrence relation is:

Payoff(B,R,j) = 0 if we don’t invest in j

Payoff(B,R,j) = Payoff(max(B), min(R), j) if we choose to invest j

**B.**

In this problem I tried using a hash table to store the tuple of (B,R,j) as the key and the value as (payout, True or False depending if we should invest). I iterate through a three dimensional array with the indexes B,R, and j. Through this I calculate the total payout for each investment. Whatever investment has the minimum cost with the maximum payoff and minimum risk score, receives a value of should\_invest = true. If this value is true then the program with recursively call payoff(B,R,j) with the new balance and new risk score. This keeps occurring until the balance = 0. The end result should return the maximum payoff.

def payoff(B,R,j):

# B is budget

# R is risk score

# j is investments ID in the range of 1 to j

#given values

cost = [20,10,40,50,50,70,80]

pay = [5,2,4,10,20,25,35]

risk\_score = [1,3,2,3,2,1,4]

#base case if there are no investments

if j == 0:

print("There are no available investments")

return 0

#base case 2

if B < 0 or R < 0:

payoff(B,R,j) = ('-inf')

#hash table of investment

payout = {}

should\_invest = False

for i in range(0,j-1):

for j in range(0,len(cost)-1):

for k in range(0,len(pay)-1):

if (max(k) and min(B-cost[i])):

should\_invest = True

payout[k,j,i] = (min(B-cost[i]),should\_invest)

if should\_invest == True:

return payout[k,j,i]

return payoff(B,R,j)

**C.**

To calculate the maximum profit, along with the given arguments (n investments, b budget, and r risk) we need to pass in a memoized array that stores the result of what investment to pick. When we run through the payoff function, we need to store what investment gives the maximum profit with the minimum cost and minimum risk store. This value is stored in the array and passed back into the payoff function with the updated balance and risk score. This keeps occurring until the budget is zero. The end result will return the array which has the order of investments which give maximum profit.

**D.**

In this problem I followed subproblem B and tried using a hash table to store the tuple of (B,R,j) as the key and the value as (payout, True or False depending if we should invest). I iterate through a three dimensional array with the indexes B,R, and j. Through this I calculate the total payout for each investment. Whatever investment has the minimum cost with the maximum payoff and minimum risk score, receives a value of should\_invest = true. The id, payoff, and risk score is stored in a memoized array. payoff(B,R,j) is called with the new balance and new risk score. This keeps occurring until the balance = 0. The end result should return the maximum payoff.

def payoffMemoize(B,R,j,memotable):

# B is budget

# R is risk score

# j is investments ID in the range of 1 to j

#given values

cost = [20,10,40,50,50,70,80]

pay = [5,2,4,10,20,25,35]

risk\_score = [1,3,2,3,2,1,4]

#base case if there are no investments

if j == 0:

print("There are no available investments")

return 0

#base case 2

if B < 0 or R < 0:

payoff(B,R,j) = ('-inf')

#hash table of investment

payout = {}

should\_invest = False

for j in memotable:

(B,R,j) = memoTable[j]

return (i,B)

for i in range(0,j-1):

for j in range(0,len(cost)-1):

for k in range(0,len(pay)-1):

if (max(k) and min(B-cost[i])):

should\_invest = True

payout[k,j,i] = (min(B-cost[i]),should\_invest)

if should\_invest == True:

return payout[k,j,i]

memoTable[j] = (B,R,j)

return payoffMemoize(B,R,i,memoTable)

**D.1** Max payoff with B = 120 and R = 9

-Investment 1:

-ID: 5 -Cost: 50 -R: 2

-Remaining Budget: 70

-Remaining Risk: 7

-Current Payoff: 20

-Investment 2:

-ID: 5 -Cost: 50 -R: 2

-Remaining Budget: 20

-Remaining Risk: 5

-Current Payoff: 40

-Investment 3:

-ID: 1 -Cost: 20 -R: 1

-Remaining Budget: 0

-Remaining Risk: 4

-Current Payoff: 45

The maximum payoff with a budget of 120 and risk score of 9 is 45.

**D.2** Max payoff with B = 140 and R = 12

-Investment 1:

-ID: 7 -Cost: 80 -R: 4

-Remaining Budget: 60

-Remaining Risk: 8

-Current Payoff: 35

-Investment 2:

-ID: 5 -Cost: 50 -R: 2

-Remaining Budget: 10

-Remaining Risk: 6

-Current Payoff: 55

-Investment 3:

-ID: 2 -Cost: 10 -R: 3

-Remaining Budget: 0

-Remaining Risk: 3

-Current Payoff: 57

The maximum payoff with a budget of 140 and a risk score of 12 is 57.

**Problem 2:**

**A.**

Base Cases:

-Have 0 lives before reaching position 101: NumHops(x,0)=∞ for x<101

-Hop pass position 101: NumHops(x,lives) when x>101 & lives>0

-Land on a perfect square (lose a life): NumHops(x,lives) when x=1,4,9,16,25,36,49,64,81,100 & lives>0

-Land on position 42 (gains extra life): NumHops(42,lives) = lives +1

-Otherwise (reaches 101, we win): NumHops(101,lives) when lives>0

**B.**

Case 1: You don't use hops of the largest denomination:

then the optimal number of jumps is NumHops(x-1,lives)

Case 2: You do use jumps of the largest denomination d(i). Then the optimal number of jumps is 1+ NumHops(x,lives-d(i))

Case 3: Avoid when x = 1,4,9,...,100

Case 4: Try to land on x = 42

The recurrence relation is therefore:

NumHops(x,lives) = min( NumHops(x-1,lives), 1 + NumHops(x,lives-d(i))

**C.**

After running, NumHops(0,9) took 26 turns to reach position 101 with the minimum amount of hops.

def NumHops(x, lives):

turns = 0

def min\_hops(i, num\_lives):

num\_lives = lives

x = x + i

turns = turns + 1

#base case for if out of lives

if num\_lives == 0:

print("Out of lives: you lose")

return 0

#base case if hop pass 101

elif x > 101:

print("Hopped pass 101: you lose")

return 0

#base case if land on perfect square

elif x = 1,4,9,16,25,36,49,64,81,100:

lives = lives - 1

#base case if land on 42

elif x = 42:

lives = lives + 1

#base case if land on 101

elif x = 101:

print("You made it to posisition 101: You Win!!!")

print("The amount of turns took: " + turns)

return 1

else:

return min(min\_hops(i-1, num\_lives), 1 + min\_hops(i, num\_lives-lives[i]))

return min\_hops(x, lives)