STMC coding team Training

Lesson 6: Recaps and problem solving

Tsai Yun Chen

April 12, 2024

Goal today

Today we will have a brief recap on what is taught in previous lesson, then we will demonstrate how to solve some real life problem using all the thing taught so far.

- Recap
- Hanoi Tower problem
- Knapsack problem
- 0-1 Knapsack problem

A brief recap

- So far we have been go through quite a few things in Python
- input, output, variable, if-then-else, looping, array, function, recursion...
- these are the basic building block of all kinds of programming languages
- you can easily tranfer from one language to another once you mastered programming

Input/Output

- The first thing is input and output
- so that we can observe what is the computer doing
- i.e.

```
1  print("Hello world")
2  input("Tell me something about you: ")
```

Variable

- The second thing is variable
- so that we can store what are we doing
- i.e.

```
a=int(input("give an integer to me: "))
b=a*a
print("square of your number is:",b)
```

If-then-else

- The third thing is conditional statement
- so that we can let the computer decide the thing by itself
- i.e.

```
a=int(input("give an even integer to me: "))
if a%2==0:
  print("Thank you")
else:
  print("come on, I need an even number,",a,"is not an even
  number")
```

Looping

- The Fouth thing is looping
- so that we can do the thing repeatly
- i.e.

```
1  a=int(input("give an integer to me: "))
2  print("all the even number between 1 to",a,"are:")
3  for j in range(1,a+1):
4    if j%2==0:
5      print(j)
```

Array

- The fouth thing is array
- so that we can store more (and arbitrary number of) thing
- i.e.

```
n=int(input("How many number you have: "))
arr=[]
sum=0
for i in range(n):
  temp=int(input("give me a number:"))
  arr.append(temp)
  sum=sum+temp
mean=sum/n
max=abs(arr[0]-mean)
for i in range(1,n-1):
  if abs(arr[i]-mean)>max:
    max=abs(arr[i]-mean)
print("The largest deviation from the mean is:",max)
```

Function

- The last thing is function
- so that we can reuse the code and make use of recursion
- i.e.

```
1  def Fib(n):
2  if n == 1 or n == 2:
3    return 1
4  else:
5    return Fib(n-1) + Fib(n-2)
```

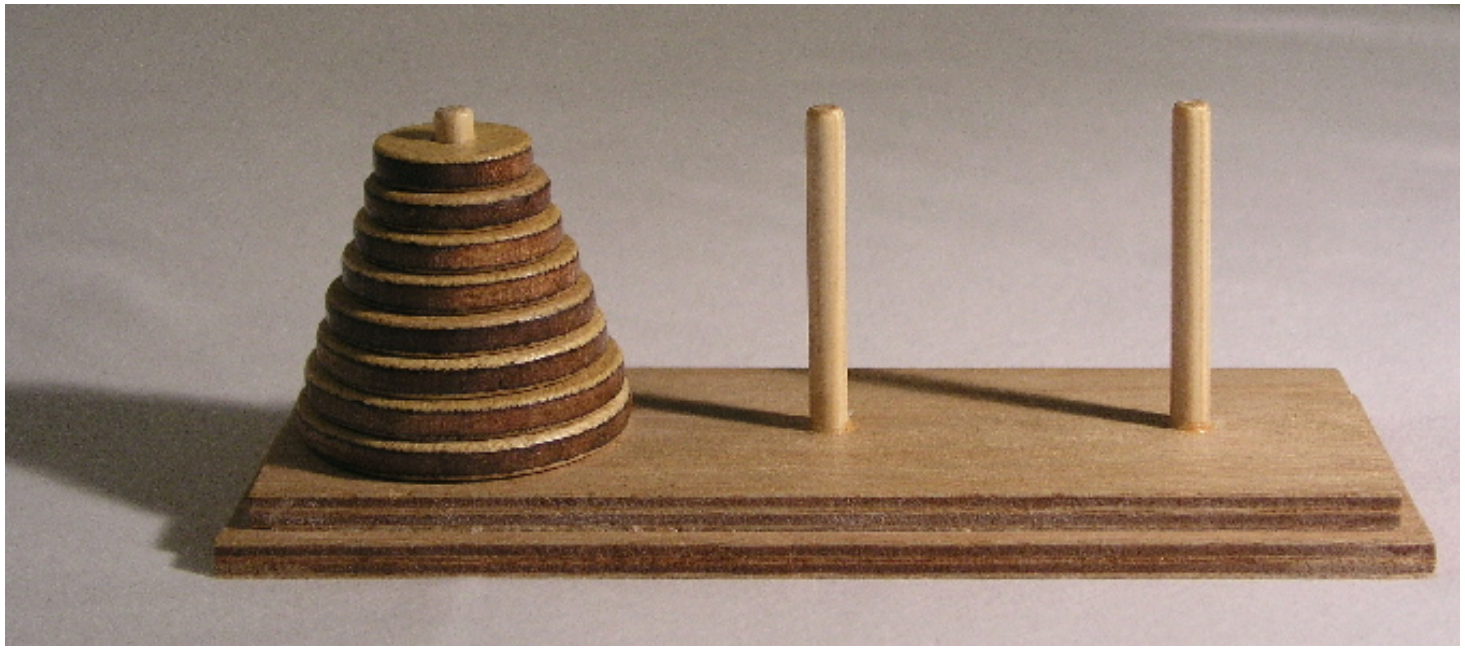
Problem Solving

- So we know how to let Python do the things we want
- But...
- We are not solving problem (yet)!
- so far what we are learning is just **coding**, which everyone can easily learn it like a normal language
- What important and hard to learn is **programming**, the skills of solving problem from coding

First Problem: Tower of Hanoi

- The Tower of Hanoi is a mathematical puzzle first introduced by Édouard Lucas in 1883
- It begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape.
- The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:
  1. Only one disk may be moved at a time.
  2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  3. No disk may be placed on top of a disk that is smaller than it.

Exercise: Tower of Hanoi



Exercise: Tower of Hanoi

- We would like to program an algorithm that would *solves* the tower of hanoi
- Given a stack of *n* disk to begin with, derive a program that will teach you how should you move the disks to solve the puzzle
- Let's try to play around with the game to gain some insights and familiarity
- Here is a link to the game: https://www.mathsisfun.com/games/towerofhanoi.html

Solution: Tower of Hanoi

Observations:

Suppose there are n disks and suppose we know how to move $n-1$ disks from rod 1 to rod 2. Then to move n disks, we just need to:

1. Move all $n-1$ disk above to the rod in the middle rod
2. Place the bottom disk to the final rod
3. Move the $n-1$ disks to the final rod

Solution: Tower of Hanoi

- In other words, once we know how to move $n-1$ disks, moving n disks is easy
- But how do we know how to move $n-1$ disks?
- Well, to move $n-1$ disk, we just need to know how to move $n-2$ disks!
- Wait how about $n-2$? We don't know how to do that right?
- Well, we just have to figure out how to move $n-3$ disks!
- . . .
- But how to move 2 disks?
- Well, to move 2 disks, we just need to know how to move 1 disk
- **But moving 1 disk is trivial!**

Coding Time

- Now it's your time to code
- To help you with the starting point, consider you need to implement the function below

```
1  def Hanoi(n,tar,emt):
2    #n is the number of pile at the rod you looking at
3    #tar is the number (or name) of the rod you wish to move the n
       pile to
4    #emt is the number (or name) of the rod that is currently
     empty
5    ###your code start below###
6
7  #Then we ask the user to provide the number of pile to be solved
     , and call the function we just implemented.
8  n=int(input("Enter the number of pile: "))
9  Hanoi(n,3,2)
```

Knapsack problem

- Now we look at another problem, assuming you are a jeweller (a person who sell jewelry) and you got a chance to sell on a large market, you only get one knapsack (backpack) to carry the jewelry to the market.
- Certainly you want to get as many jewelries as possible so you earn more, however the size of the backpack is limited so you cannot always put all the jewelries into the backpack.
- Therefore you want to plan ahead to decide which jewelry and how much you should bring. However, the calculation is so annoying you want to design a program to help you so that you don't have to do it again in the future.

Knapsack problem, formally

- Let's try to state the problem formally, assume there are K kind of jewelries, each with total value as v_i and number w_i, then the problem statement is

Input:	W : Backpack capacity
	$\{v_1, v_2, ..., v_K\}$: value of each jewelry
	$\{w_1, w_2, ..., w_K\}$: number of each jewelry
Output:	$\{n_1, n_2, ..., n_K\}$: number of each type of jewelry taken
Subject to:	$w_1 n_1 + w_2 n_2 + ... + w_K n_K \leq W$
	$v_1 n_1 + v_1 n_2 + ... + v_K n_K$ should be as large as possible

First Attempt

- Let’s start with a simple solution
- Take whatever that is the most expensive!
- Then we only need to sort by the price and take from the tail until we cannot fit any of them.

Does it works?

- Consider the following case, your backpack capacity is 100 kg
- You got three types of Jewelry to choose from
  1. Diamond, Total price: $\$90$, weight: 90 kg
  2. Gold, Total price: $\$70$, weight: 20 kg
  3. Emerald, Total price: $\$50$, weight: 20 kg
- Our solution: Take 90 kg Diamond then take 10 kg Gold
- Total price: $100 + 70/2 = \$135$
- What if: Take 20 kg Gold, 20 kg Emerald, 40 kg Diamond?
- Total Price: $70 + 50 + 90 * 4/9 = \$160 > \$135$
- What's wrong with our solution?

Second Attempt

- The price is total instead of unit, the diamond does not worth so much here
- Modified solution: Take whatever that is the most expensive **in unit price**
- New approach:
  1. For each Jewelry, compute $u_i = v_i / w_i$, which is their unit price
  2. Sort the list by u_i
  3. Take the most out of the largest whenever possible

Implementation

```
1  def Knapsack(W,v,w):
2    #Treat the first two lines below as some magic, it generates a
     list u, each element in u is a pair with the the index and the
     unit price, then u is sorted in descending order
3    u=[(idx,v[idx]/w[idx]) for idx in range(len(v))]
4    u.sort(key=(lambda pair:pair[1]),reverse=True)
5    knapsack=[]
6    for jew in u:
7      if w[jew[0]]<=W:
8        knapsack.append((jew[0],w[jew[0]]))
9        print("Take",w[jew[0]],"kg of Jewelry",jew[0])
10       W=W-w[jew[0]]
11     else:
12       knapsack.append((jew[0],W))
13       print("Take",W,"kg of Jewelry",jew[0])
14       break
15   return knapsack
```

Let's Try it

- Let's try to run the code
- We use the same example demonstrated above, we do the following

```
W=100
v=[90,70,50]
w=[90,20,20]
Knapsack(W,v,w)
```

- Can you come up with other example?
- Is there any situation where this solution can fail to find the best combination?

Variation: 0-1 Knapsack problem

- Now consider on top of the knapsack problem setting, we have one more rule
- `All or Nothing`, you either take all of one item or none of them
- Does our previous solution still work?
- No! Our method may take a portion of them item in the last step.
- What if we fix that? Changing the loop to

```
for jew in u:
    if w[jew[0]]<=W:
      knapsack.append((jew[0],w[jew[0]]))
      print("Take",w[jew[0]],"kg of Jewelry",jew[0])
      W=W-w[jew[0]]
```

Counter Example

- Consider the same case but with one more jewelry, with backpack capacity 100 kg
  1. Diamond, Total price: $\$90$, weight: 90 kg –> unit price: $\$1/kg$
  2. Gold, Total price: $\$70$, weight: 20 kg –> unit price: $\$3.5/kg$
  3. Emerald, Total price: $\$50$, weight: 20 kg –> unit price: $\$2.5/kg$
  4. Ruby, Total price: $\$80$, weight: 80 kg –> unit price: $\$1/kg$
- Our solution: Take 20 kg Gold, 20 kg Emerald
- Total Price: $70 + 50 = \$120$
- What if: Take 20 kg Gold, 80 kg Ruby
- Total Price: $70 + 80 = \$150 > \$120$

Greedy Algorithm and its downside

- What we have been using is called the Greedy algorithm
- We also take the best choice we have at each step
- Benefits:
  - Easy to implement
  - Intuitive to reason
  - Efficient, the most time taken in sorting
- Problem: May not always be optimal

Dynamic Programming to the rescue!

- As in recursion, we will try to break the problem into smaller problem
- Consider you have the solution to n items with any weight limit W, denote it as $OPT(n, W)$
- Now we add one more item into the list, what is the new solution?
- Observation: You either take the new item, with value v and weight w, or you don't.
- Sounds obvious right? But that what we want, so now we know

$$OPT(n+1, W) = \max(OPT(n, W), OPT(n, W-w) + v) \tag{1}$$

- But wait, aren't we trying to make the problem smaller?
- We can start from $n = 1$ and gradually increase it.

Solution

1. Build up a table DP with $n + 1$ rows and W column
2. Set $DP[0][w]$ to be 0 if $w_0 < w$ and v_0 otherwise
3. Loop from $k = 1$ to n and $w = 0$ to W, fill in the table $DP[k][w]$ according to equation (1)
4. return $DP[n][W]$

Implementation

```
1  def DPKnapsack(W,v,w):
2    DP=[[0]*(W+1)]
3    DP[0][w[0]:]=[v[0]]*(W-w[0]+1)
4    for i in range(1,len(v)):
5      DP.append([0]*(W+1))
6      for j in range(W+1):
7        if j<w[i]:
8          DP[i][j] = DP[i-1][j]
9        else:
10         DP[i][j] = max(DP[i-1][j],DP[i-1][j-w[i]]+v[i])
11   return DP[len(v)-1][W]
```

Exercise

- Let’s try to test the code against our previous counter example and see what is the result
- Challenge: Modify the above code so that we also know which jewelries are taken