inzva Algorithm Study Group
Week 8

Dynamic Programming

Guide:
Ahmet Melek

Thanks to Şükrü “skr” Bezen for approaches

16.01.2021

Today's Contents

1.	**What is Dynamic Programming?**	**12.00 - 12.20**	
2.	**Greedy: Marc's Cakewalk Problem**	**12.20 - 12.40**	full solution here
3.	**Do it once: Suffix Query**	**12.40 - 13.00**	full solution here
	Break	13.00 - 13.15	
4.	**Memoization: Basic-Unbounded-Knapsack**	**13.15 - 13.35**	full solution here
5.	**Memoization: LCS**	**13.35 - 13.55**	full solution here
6.	**Memoization: LIS**	**13.55 - 14.15**	full solution here
	Break	14.15 - 14.30	

What is Dynamic Programming?

Dynamic Programming is mainly an **optimization** over plain **recursion**. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.[1]

Steps to solve a DP

1) Identify if it is a DP problem
2) Decide a state expression with least parameters
3) Formulate state relationship
4) Do tabulation (or add memoization)[1]

We will also cover Greedy Approaches this week, which is different from dynamic programming.

[1]: https://www.geeksforgeeks.org/dynamic-programming/ (this is very good material)

1- Mark’s Cakewalk

This is a greedy type of problem.

```
# Complete the marcsCakewalk function below.
def marcsCakewalk(calorie):
    calorie.sort(reverse=True)
    mysum = 0
    for i,cal in enumerate(calorie):
        mysum += 2**i * cal
    return mysum

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    n = int(input())

    calorie = list(map(int, input().rstrip().split()))

    result = marcsCakewalk(calorie)

    fptr.write(str(result) + '\n')


    fptr.close()
```

Task: Consume cakes to get minimum calories

Important: Each cake gives **2^(position) * calorie** calories

Inputs: Calorie values of cakes

Example input:
2 3 4

2- Onur and the Suffix Query

This is a do-it-once (?) type of problem. We process our input once, and note down the output, instead of for each query.

```
def count_former(recurrent_list):
    memory = []
    unique_count_list = []
    count = 0
    for item in recurrent_list[::-1]:
        if not item in memory:
            count = count + 1
            memory.append(item)
        unique_count_list.append(count)
        #print("inside ",item,count)
    return unique_count_list[::-1], count

n,m = map(int,input().split())
recurrent_list = list(map(int,input().split()))

unique_count_list, last_count = count_former(recurrent_list)
for _ in range(m):
    position = int(input())
    print(unique_count_list[position-1])
```

Task: Find number of distinct integers from position i to the end of the sequence

Inputs: The sequence, and queries consisting of beginning position i.

Example input:

1 5 3 4 1 5 3 4 100000 88888	(sequence)
1	(query for i=1)
2	(query for i=2)
9	(query for i=9)
10	(query for i=10)

3- Basic-Unbounded-Knapsack

This is a memoization problem. In a recursive process, we note down some results to cut back computation.

```
if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    t = int(input())

    for i in range(t):
        nk = input().split()

        n = int(nk[0])

        k = int(nk[1])

        arr = list(map(int, input().rstrip().split()))

        memory = {}
        result = unboundedKnapsack(k, arr)

        fptr.write(str(result) + '\n')


    fptr.close()
```

Task: We have a money limit to spend. We also have banknotes. Spend most money with the banknotes, without exceeding the limit.

Inputs: The sequence, and queries consisting of beginning position i.

Example input:

12	(money limit)
1 6 9	(banknotes)

3- Basic-Unbounded-Knapsack

Our recursive function:

```
# Complete the unboundedKnapsack function below.
sys.setrecursionlimit(10000)
def unboundedKnapsack(k, arr):
    if k in memory:
        return memory[k]

    chosen_sum = 0
    for item in arr:
        candidate_sum = 0
        if item <= k:
            candidate_sum = unboundedKnapsack(k-item, arr) + item
        else:
            continue

        if candidate_sum > chosen_sum:
            chosen_sum = candidate_sum

    memory[k] = chosen_sum

    return chosen_sum
```

4- LCS: Longest Common Subsequence

This is a memoization problem. In a recursive process, we note down some results to cut back computation.

```
n, m = map(int, input().split())
a = list(map(int, input().rstrip().split()))
b = list(map(int, input().rstrip().split()))

memory = {}
backtrack = {}
length = longestCommonSubsequence(0, 0)

printer()
```

Task: We have two sequences. Find the longest subsequence that exists in both of the sequences.

Important: A subsequence is formed by deleting elements from the original sequence

Inputs: First sequence and second sequence

Example input:

1 2 3 4 1	(first sequence)
3 4 1 2 1 3	(second sequence)

Example to explain substrings:

Original:	1 2 3 4 1
A substring of original:	1 2 3
Another substring of original:	1 1

4- LCS: Longest Common Subsequence

Our recursive function:

```
def longestCommonSubsequence(i, j):
    global a,b,n,m

    if i>=n or j>=m:
        return 0

    if (i,j) in memory:
        return memory[i,j]

    if a[i] == b[j]:
        memory[i,j] = 1 + longestCommonSubsequence(i+1,j+1)
        backtrack[i,j] = 1,1
        return memory[i,j]

    left = longestCommonSubsequence(i+1,j)
    right = longestCommonSubsequence(i, j+1)

    if left>=right:
        chosen = left
        backtrack[i,j] = 1,0
    elif right>left:
        chosen = right
        backtrack[i,j] = 0,1

    memory[i,j] = chosen

    return memory[i,j]
```

4- LCS: Longest Common Subsequence

Our print function:

```
def printer():
    i = 0
    j = 0
    while(1):
        if i+1>n or j+1>m:
            break

        if a[i] == b[j]:
            print(a[i], end = ' ')

        if (i,j) in backtrack:
            myop = backtrack[i,j]
            i = i + int(myop[0])
            j = j + int(myop[1])
        else:
            break
```

5- LIS: Longest Increasing Subsequence

This is a memoization problem. In a recursive process, we save some results to cut back computation.

```
seq_len = int(input())
sequence = list(map(int,input().split()))

dp_memory = {}

maxi = 0
current = 0

for i in range(seq_len):
    current = LIS(i)
    if current > maxi:
        maxi = current

print(maxi)
```

In this question, we do not call all of the solution-space recursively. We also use two for loops.

This is for saving memory, and for ease of implementation.

Task: We have a sequence. Find the longest subsequence. The sequence has to be also strictly increasing.

Important: A subsequence is formed by deleting elements from the original sequence

Inputs: The sequence

Example input:
15 27 14 38 26 55 46 65 85 (the sequence)

5- LIS: Longest Increasing Subsequence

Our recursive function:

```
def LIS(start_index):
    if start_index in dp_memory:
        return dp_memory[start_index]

    maxi_local = 0
    current_local = 0


    for i in range(start_index+1, seq_len):
        if(sequence[i]>sequence[start_index]):
            current_local = LIS(i)
            if current_local > maxi_local:
                maxi_local = current_local
    dp_memory[start_index] = maxi_local + 1
    return maxi_local + 1

maxi = 0

current = 0
```