FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 10

Semester 1, 2019

Objectives of this tutorial

- To understand Dynamic Programming.

Exercise 1 *

Write down an algorithm to compute the n-th Fibonacci number using Dynamic Programming. How does this compare in terms of time complexity to the recursive and tail recursive implementations done in the class?

Solution

The standard relation for the nth Fibonacci numbers is $F(n) = F(n-1) + F(n-2)$.

If we wish to solve by Dynamic Programming we need to store the overlapping subproblems; in this case $F(n-1)$ will require $F(n-2)$ causing an algorithm following that relation directly to grow rapidly in time.

Let's instead build upwards from the base cases until we reach the desired case. A python solution is given below:

```python
def FibN(n):
        fibValues = [None]*(n+1)
        fibValues[0] = 1
        fibValues[1] = 1 #base cases

        for k in range(2,n+1):
                fibValues[k] = fibValues[k-1] + fibValues[k-2]
        return fibValues[n]
```

Exercise 2 *

You are presented with a row of 5 coins with values \$7, \$2, \$10, \$12, \$5. Pick up the largest amount of money from the row of coins, with the constraint that you cannot pick up any two adjacent coins.

How can you design an algorithm to solve the problem for a row of n coins of arbitrary values?

Solution

The largest amount of money that can be picked up is \$22 — the \$7, \$10 coin and \$5 coins.

The best solution for n coins is the maximum value out of (a) the best solution for n-1 coins, and (b) the best solution for n-2 coins, plus the value of the nth coin. (Since the nth coin is not adjacent to any coins in the solution for n-2 coins, it is always allowable to add its value to the solution for n-2 coins.) The recurrence relation is:

$$F(n) = \max\{F(n-1), F(n-2) + V_n\},$$

where $F(n)$ is the solution for n coins, and V_n is the value of the n^{th} coin.

For the example coin values given, the solutions for $n = 0 \dots 5$ would be as follows:

n	0	1	2	3	4	5
$F(n)$	0	7	7	17	19	22

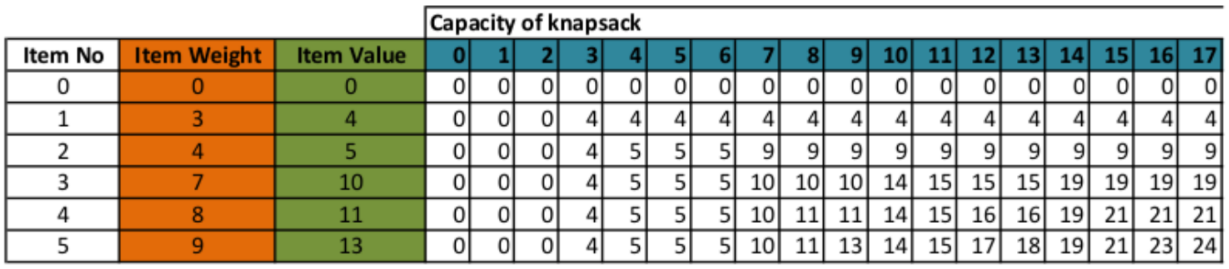
Exercise 3 *

Using dynamic programming, solve the knapsack problem given by the table below. The knapsack capacity is 17 kg.

Item	**1**	**2**	**3**	**4**	**5**
Value (\$)	4	5	10	11	13
Weight (kg)	3	4	7	8	9

Solution

The table below is generated by following the dynamic programming algorithm for knapsack, given in lectures.



			Capacity of knapsack																	
Item No	Item Weight	Item Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	0	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
2	4	5	0	0	0	4	5	5	5	9	9	9	9	9	9	9	9	9	9	9
3	7	10	0	0	0	4	5	5	5	10	10	10	14	15	15	15	19	19	19	19
4	8	11	0	0	0	4	5	5	5	10	11	11	14	15	16	16	19	21	21	21
5	9	13	0	0	0	4	5	5	5	10	11	13	14	15	17	18	19	21	23	24

Exercise 4 *

In solving the knapsack, assume that you are now allowed to take duplicates, i.e. each item can be selected multiple times, as long as the knapsack capacity is not exceeded. Discuss how you would modify the Dynamic Programming algorithm given in the lecture to solve the problem when duplicate are allowed.

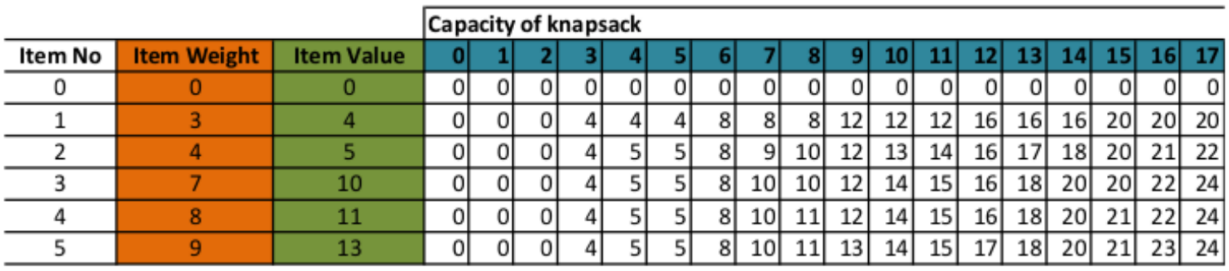
Solution

Algorithm 1 gives the modified algorithm solving the problem for when duplicate items are allowed. The only alteration required is in the initialisation of *valueIncludingI*, the best possible solution including the new item. This value now involves looking at the *current* row in the table, rather than the previous row, since the new item can be included more than once.

Algorithm 1 knapsackWithDuplicates(weights[0..N-1], values[0..N-1], capacity)

```
INPUT: List of item weights, list of item values, knapsack capacity.
OUTPUT: Maximum value of items you can carry.
ASSUMPTIONS: Duplicates allowed — i.e., there is an unlimited supply of each item.
MaxValue ← makeTable(0, N+1, capacity+1)
i ← 1
while (i ≤ N) do
   j ← 1
   while (j ≤ capacity) do
      MaxValue[i,j] ← MaxValue[i-1, j]
      if weights[i-1]≤j then
         valueIncludingI ← values[i-1] + MaxValue[i, j-weights[i-1]]
         if MaxValue[i-1, j] < valueIncludingI then
            MaxValue[i,j] ← valueIncludingI
         end if
      end if
      j ← j+1
   end while
   i ← i+1
end while
return MaxValue[N, capacity]
```

The table below shows Algorithm 1 running on the case solved in task 3.



			Capacity of knapsack																	
Item No	Item Weight	Item Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
2	4	5	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
3	7	10	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
4	8	11	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
5	9	13	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24

Exercise 5

- Write a **recursive** algorithm to compute the binomial coefficients, given by the following formula:

  $$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

  Assume $n \geq k$, with $n > 0$, $k > 0$, $n, k \in \mathbb{Z}$. Note that

  $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

- Write a **Dynamic Programming** algorithm to compute the binomial coefficients.

Compare the complexity of the two implementations.

Solution

We can solve this problem recursively by noting that the binary coefficient (n,k) is the kth number in the nth row of pascals triangle (where the top row is considered the zeroeth row). In this way, we compute binCoeff(n,k) by adding binCoeff(n-1,k) and binCoeff(n-1,k-1). The base cases are when k=0 or k=n (the edges of the trianle, where the values are all 1) and when k>n ("outside" Pascal's triangle. Another way of thinking of this base case is that one cannot choose k items from n items when k>n).

Algorithm 2 binCoeffRec(n,k)

```
1: INPUT: n, k
2: OUTPUT: $\binom{n}{k}$
3: ASSUMPTIONS: n, k are positive integers
4: if (k > n) then
5:     return 0
6: else
7:     if (k = 0 OR k = n) then
8:         return 1
9:     else
10:        return binCoeffRec(n-1,k-1) + binCoeffRec(n-1,k)
11:    end if
12: end if
```

Part 2.

The Dynamic programming version is very similar to the recursive version. The difference is that values are stored, so that the same value does not need to be computed many times.

Algorithm 3 binCoeffDyn(n,k)

```
INPUT: n, k
OUTPUT: $\binom{n}{k}$
ASSUMPTIONS: n, k are positive integers
BC ← makeTable(0,n+1,k+1)
for (i=0..n) do
   for (j=0..k) do
      if (j > i) then
         BC[i,j] ← 0
      else
         if (j = i or j = 0) then
            BC[i,j] ← 1
         else
            BC[i,j] ← BC[i-1,j-1] + BC[i-1,j]
         end if
      end if
   end for
end for
return BC[n,k]
```