THE UNIVERSITY OF MELBOURNE

COMP90038 Algorithms and Complexity

Tutorial 10 Dynamic Programming

Tutor: Chuang(Frank) Wang

1. Review

Dynamic Programming

What does it mean?

- **Very good def** 📖 — Dynamic programming is an algorithm design technique which sometimes is applicable when we want to solve a recurrence relation and the recursion involves overlapping instances.
- **Dynamic** 💭 — the multistage, time-varying processes
- **Programming** 🤔 — planning or scheduling, typically by filling in a table (not refer to writing code~)

Fibonacci

1. **Dummy** 😑 — Naive recursion (Lecture 6)
2. **A bit clever** 😐 — Fibonacci numbers with tabulation - Remember Everything! (Lecture 16)
3. **Super clever** 😄 ⚡ — **Dynamic Programming - Fill Deliberately!**

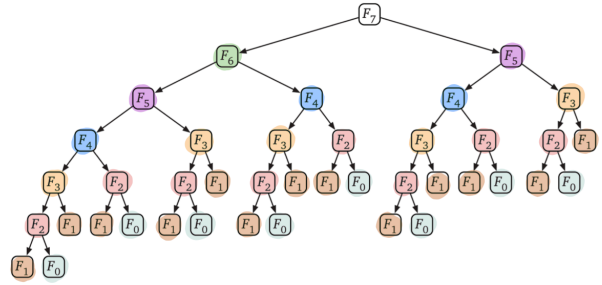


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.
The graph is referenced from *Algorithms by Jeff Erickson - Chapter 3*

To calculate $F(7)$, the recursive function $F(i)$ is called

$1 + 2 + 3 + 5 + 8 + 13 + 8 = 40$ times

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i−1] + F[i−2]
  return F[n]
```

The Coin-Row Problem

Super Clever 😄 ⚡

Dynamic Programming

```
function COINROW(C[·], n)
    S[0] ← 0
    S[1] ← C[1]
    for i ← 2 to n do
        S[i] ← max(S[i − 1], S[i − 2] + C[i])
    return S[n]
```

The Knapsack Problem

1. **Dummy** 😑 — Brute force - find all permutation of a set (Lecture 5)
2. **Clever** 😄 ⚡ — **Dynamic Programming**

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

2. Tutorial Questions

Question 74

74. Use the dynamic-programming algorithm developed in Lecture 18 to solve this instance of the coin-row problem: 20, 50, 20, 5, 10, 20, 5.

Given a row of coins, pick up the largest possible sum, subject to this constraint: ***No two adjacent coins can be picked.***

- Let the values of the coins be v1 , v2 , . . . vn.
- Let $S(i)$ be the sum that can be gotten by picking optimally from the first i coins.
- ***Pick i:*** the best we can achieve is $S(i - 2) + v_i$.
- ***Don't pick i:*** the best we can achieve is $S(i - 1)$

Goal: find

$$S(n) = \max(S(n - 1), S(n - 2) + v_n)$$

(This holds for n > 1)

Two base cases: $S(0) = 0$ and $S(1) = v_1$.

```
function CoinRow(C[·], n)
    S[0] ← 0
    S[1] ← C[1]
    for i ← 2 to n do
        S[i] ← max(S[i − 1], S[i − 2] + C[i])
    return S[n]
```

Discard V_n Pick V_n

$$S(n) = \max(S(n-1), S(n-2) + V_n)$$

Store this array

index	0	1	2	3	4	5	6	7
coin	–	20	50	20	5	10	20	5
S(i)	0	20	50	50	55	60	75	75

Dynamic Programming

$S(0) = 0$

$S(1) = 20$

$S(2) = \max(S(1), S(0) + 50) = \max(20, 50) = 50$

$S(3) = \max(S(2), S(1) + 20) = \max(50, 40) = 50$

$S(4) = \max(S(3), S(2) + 5) = \max(50, 55) = 55$

$S(5) = \max(S(4), S(3) + 10) = \max(55, 60) = 60$

$S(6) = \max(S(5), S(4) + 20) = \max(60, 75) = 75$

$S(7) = \max(S(6), S(5) + 5) = \max(75, 65) = 75$

Question 75

75. In Week 12 we will meet the concept of *problem reduction*. This question prepares you for that. First, when we talk about the length of a path in an un-weighted directed acyclic graph (dag), we mean the number of edges in the path. (You could also consider the un-weighted graph weighted, will all edges having weight 1.)

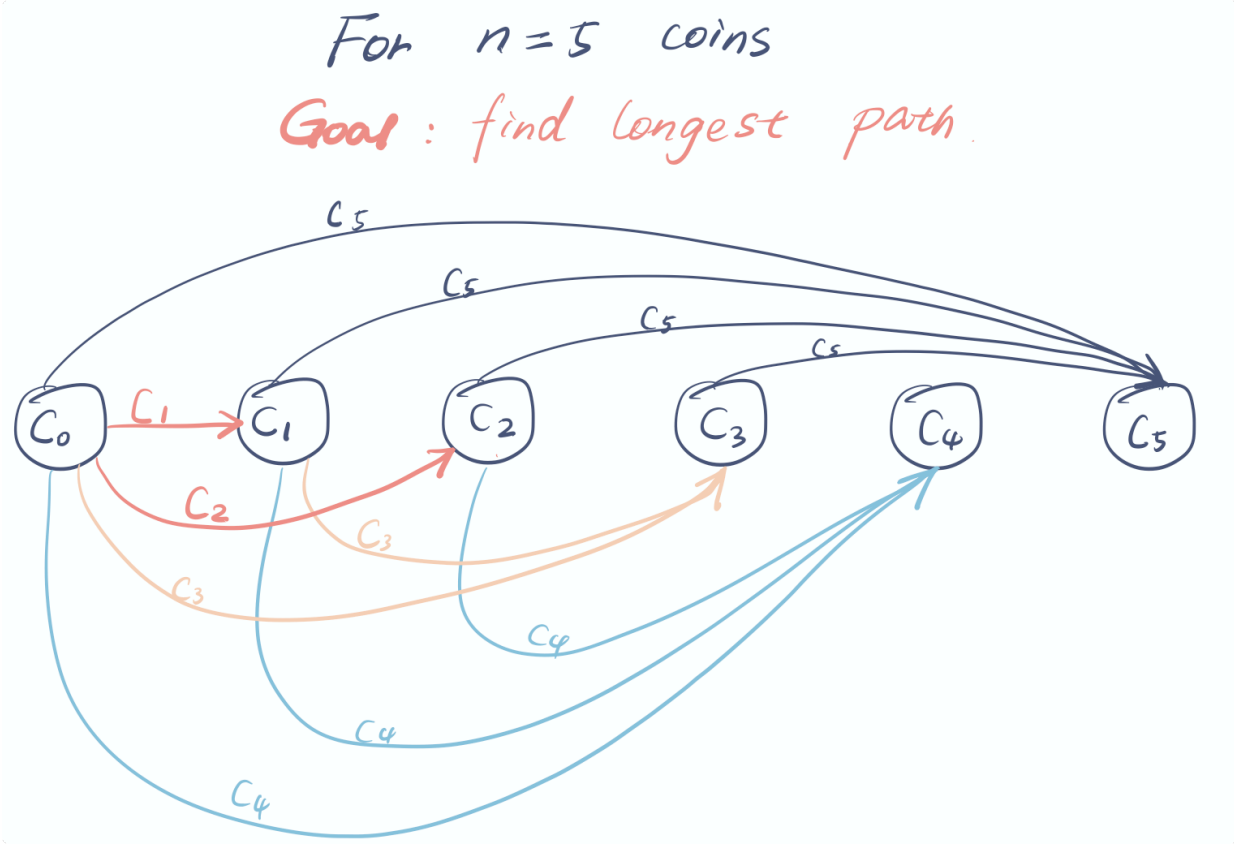
    Show how to reduce the coin-row problem to the problem of finding a longest path in a dag. That is, give an algorithm that transforms any coin-row instance into a longest-path-in-dag instance in such as way that a solution to the latter provides a solution to the former.

    Hint: If there are n coins, use $n+1$ nodes; let an edge with weight i correspond to picking a coin with value i.

Answer: Assume we have n coins $c_1, \ldots, c_n$. We generate a weighted dag with $n+1$ nodes $C_0, C_1, \ldots, C_n$. The dag has edges as follows:

- $n-1$ edges $(C_0, C_n), (C_1, C_n), \ldots, (C_{n-2}, C_n)$, each with weight c_n.
- $n-2$ edges $(C_0, C_{n-1}), (C_1, C_{n-1}), \ldots, (C_{n-3}, C_{n-1})$, each with weight c_{n-1}.
- and so on, down to two edges (C_0, C_3) and (C_1, C_3), each with weight c_3.
- one edge (C_0, C_2) with weight c_2, and
- one edge (C_0, C_1) with weight c_1.

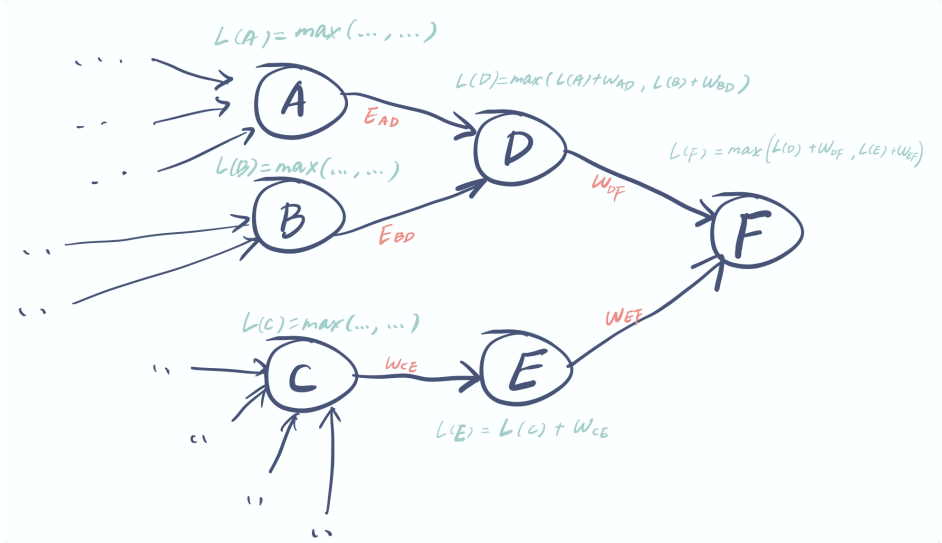
Any path in the generated dag corresponds to a legal selection of coins, and the sum of the weights along a given path is exactly the sum of the coins chosen.



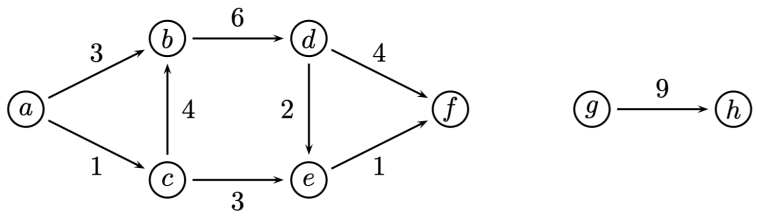
Question 76

great DP problem!

- **Step 1:**
  - ✓ work backwards
  - ✓ think of each node as the last node and ask what is longest path from the source to you
- **Step 2:**
  - ✓ now, let's do it -> calculate the longest path from source to each node & store them in the list
- **Step 3:**
  - ✓ find the largest value in the list



76. Consider the problem of finding the length of a "longest" path in a *weighted*, not necessarily connected, dag. We assume that all weights are positive, and that a "longest" path is a path whose edge weights add up to the maximal possible value. For example, for the following graph, the longest path is of length 15:



Use a dynamic programming approach to the problem of finding longest path in a weighted dag.

Answer: This is easy if we process the nodes in topologically sorted order. For each node t we want to find its longest distance from a source, and to store these distances in an array L. That is, for each t we want to calculate

$$max(\{0\} \cup \{L[u] + weight[u,t] \mid (u,t) \in E\})$$

So:

$T \leftarrow$ TopSort$(\langle V, E\rangle)$ — List of nodes sorted topologically
for each $t \in T$ (in topological order) **do**
  $L[t] \leftarrow 0$
  for each $u \in V$ **do**
    if $(u,t) \in E$ **then**
      if $L[u] + weight[u,t] > L[t]$ **then**
        $L[t] \leftarrow L[u] + weight[u,t]$
$max \leftarrow 0$
for each $u \in V$ **do**
  if $L[u] > max$ **then**
    $max \leftarrow L[u]$
return max

For the sample graph, DFS-based topsort yields the sequence g, h, a, c, b, d, e, f. The "longest path" table L gets filled as follows:

t:	a	b	c	d	e	f	g	h
$L[t]$:							0	
								9
	0							
		1						
			5					
				11				
					13			
						15		

Question 77

77. Design a dynamic programming algorithm for the version of the knapsack problem in which there are unlimited numbers of copies of each item. That is, we are given items $I_1, \ldots, I_n$ that have values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$ as usual, but each item I_i can be selected several times. Hint: This actually makes the knapsack problem a bit easier, as there is only one parameter (namely the remaining capacity w) in the recurrence relation.

```python
weights = [4, 5, 3]
values = [11, 12, 7]

def knapsack(capacity, weights, values, n):
    """given knapsack of `capacity`, n items with weights and values,
    find the most valuable selection of items that will fit in the knapsack.

    Parameters
    ----------
    capacity : int
        capacity of the bag
    weights : list of ints
        weights of each item
    values : list of ints
        values of each item
    n : int
        total number of items

    Returns
    -------
    int
        the largest value that can be put in a bag of `capacity`
    """
    V = {0: 0}
    for w in range(1, capacity + 1):
        V[w] = 0
        for i in range(0, n):
            if weights[i] <= w and (V[w - weights[i]] + values[i]) > V[w]:
                V[w] = V[w - weights[i]] + values[i]

    return V[capacity]

```

NOTE: not good naming convention/parameter passing/defensive checking etc., but just for your understanding

Answer: Assume the items $I_1, \ldots, I_n$ have values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$. Let $V(w)$ denote the optimal value we can achieve given capacity w. With capacity w we are in a position to select any item I_i which weighs no more than w. And if we pick item I_i then the best value we can achieve is $v_i + V(w - w_i)$. As we want to maximise the value for capacity w, we have the recurrence

$$V(w) = max\{v_i + V(w - w_i) \mid 1 \leq i \leq n \wedge w_i \leq w\}$$

That leads to this table-filling approach:

for $w \leftarrow 1$ to W **do**
  $V[w] \leftarrow max(\{0\} \cup \{v_i + V(w - w_i) \mid 1 \leq i \leq n \wedge w_i \leq w\})$
return $V[W]$

As an example, consider the case $W = 10$, and three items I_1, I_2, and I_3, with weights 4, 5 and 3, respectively, and values 11, 12, and 7, respectively. The table V is filled from left to right, as follows:

w:	1	2	3	4	5	6	7	8	9	10
$V[w]$:	0	0	7	11	12	14	18	22	23	25

Hence the optimal bag is $[I_1, I_3, I_3]$ for a total value of 25.

THE UNIVERSITY OF MELBOURNE

Next Week:

- *Greedy Algorithm*

Thank you !