COMP108
Data Structures and Algorithms

Dynamic Programming (Part I Fibonacci Numbers)

Professor Prudence Wong

pwong@liverpool.ac.uk

2022-23

Outline

Dynamic Programming Algorithms

- What is dynamic programming algorithm?
- See some examples

Learning outcomes:

- Understand what dynamic programming algorithm is
- Able to apply dynamic programming algorithm on computing Fibonacci Numbers
- Able to apply dynamic programming algorithm on the Assembly Line Scheduling Problem

Dynamic Programming

An efficient way to implement some divide and conquer algorithms

Fibonacci Numbers

Fibonacci number $F(n)$

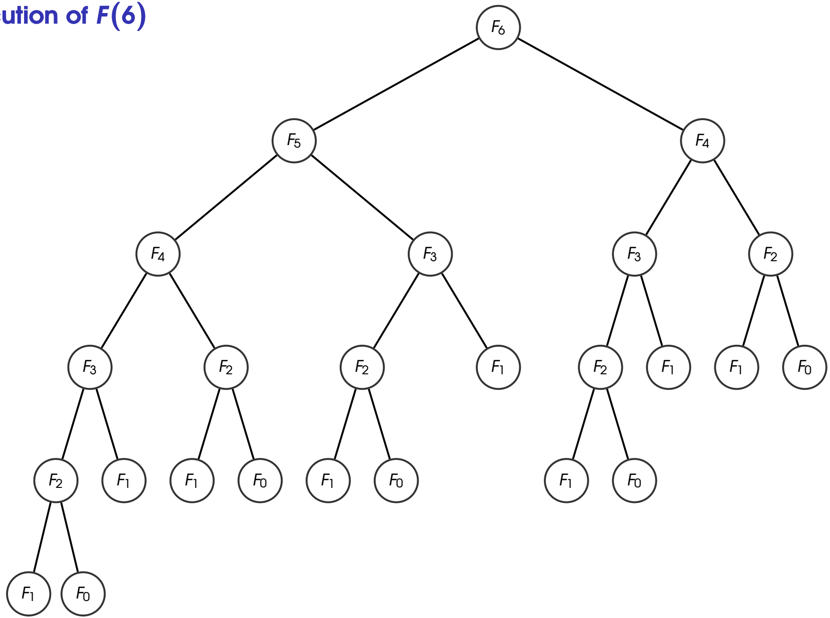
$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	1	1	2	3	5	8	13	21	34	55	89

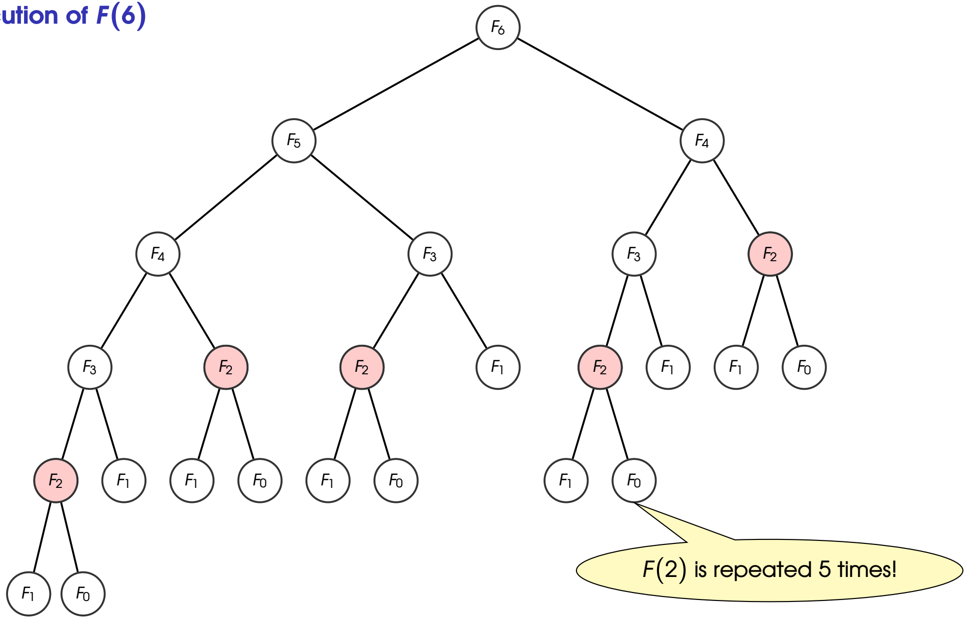
Pseudo code for the recursive algorithm:

```
Algorithm F(n)
    if n == 0 OR n == 1 then
        return 1
    else
    return F(n − 1) + F(n − 2)
```

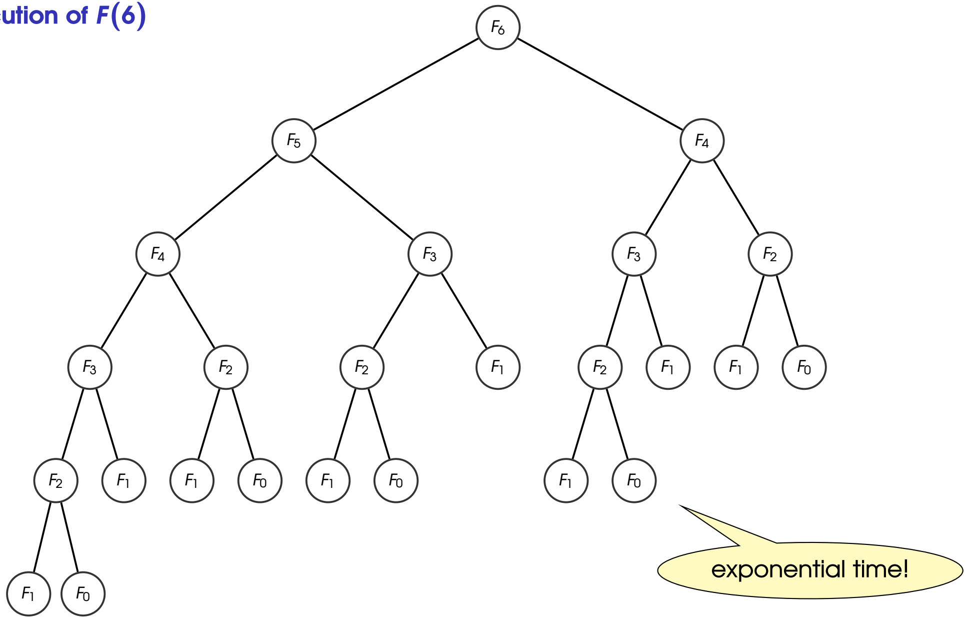
Execution of $F(6)$



Execution of $F(6)$



Execution of $F(6)$



Idea for Improvements

Memorisation:

- Store $F(i)$ somewhere after we have computed its value

Idea for Improvements

Memorisation:

- Store $F(i)$ somewhere after we have computed its value

Main Algorithm
set $v[0] \leftarrow 1$, $v[1] \leftarrow 1$
for $i \leftarrow 2$ to n do
    $v[i] \leftarrow -1$
output $F(n)$

Idea for Improvements

Memorisation:

- Store $F(i)$ somewhere after we have computed its value
- Afterward, we don't need to re-compute $F(i)$; we can retrieve its value from the memory

```
Main Algorithm
   set v[0] ← 1, v[1] ← 1
   for i ← 2 to n do
      v[i] ← −1
   output F(n)
```

Idea for Improvements

Memorisation:

- Store $F(i)$ somewhere after we have computed its value
- Afterward, we don't need to re-compute $F(i)$; we can retrieve its value from the memory

```
Main Algorithm
   set v[0] ← 1, v[1] ← 1
   for i ← 2 to n do
      v[i] ← −1
   output F(n)
```

```
Algorithm F(n)



   return v[n]
```

Idea for Improvements

Memorisation:

- Store $F(i)$ somewhere after we have computed its value
- Afterward, we don't need to re-compute $F(i)$; we can retrieve its value from the memory

```
Main Algorithm
   set v[0] ← 1, v[1] ← 1
   for i ← 2 to n do
      v[i] ← −1
   output F(n)
```

```
Algorithm F(n)
   if v[n] < 0 then

   return v[n]
```

Idea for Improvements

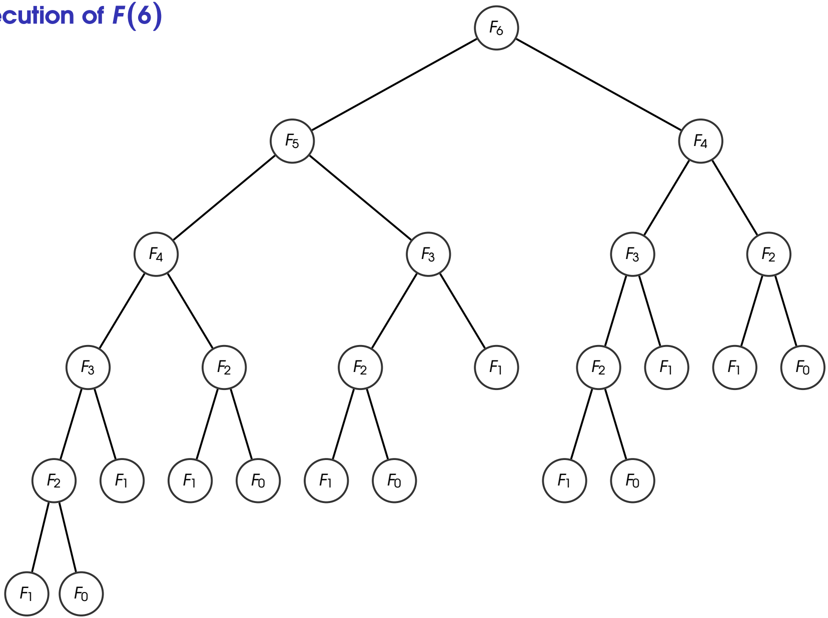
Memorisation:

- Store $F(i)$ somewhere after we have computed its value
- Afterward, we don't need to re-compute $F(i)$; we can retrieve its value from the memory

```
Main Algorithm
  set v[0] ← 1, v[1] ← 1
  for i ← 2 to n do
    v[i] ← −1
  output F(n)
```

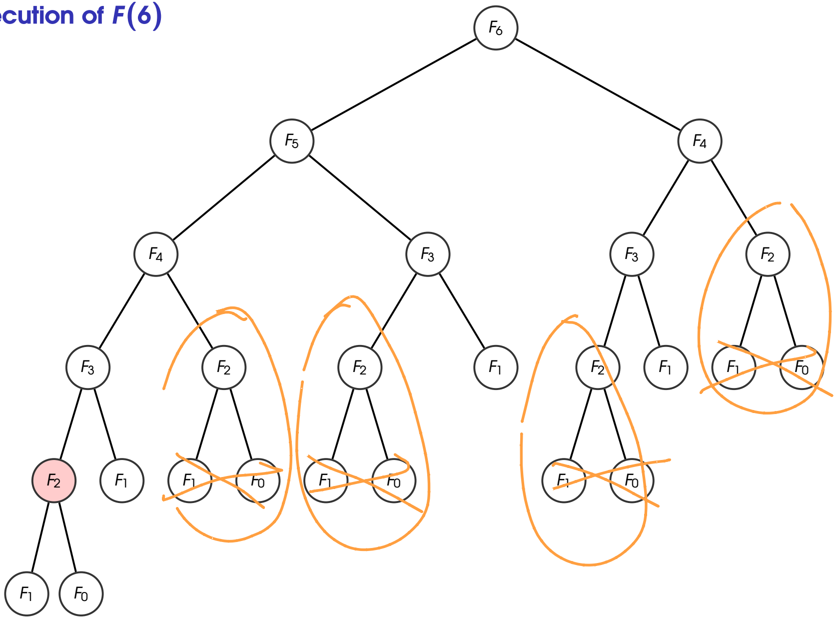
```
Algorithm F(n)
  if v[n] < 0 then
    v[n] ← F(n − 1) + F(n − 2)
  return v[n]
```

Execution of $F(6)$



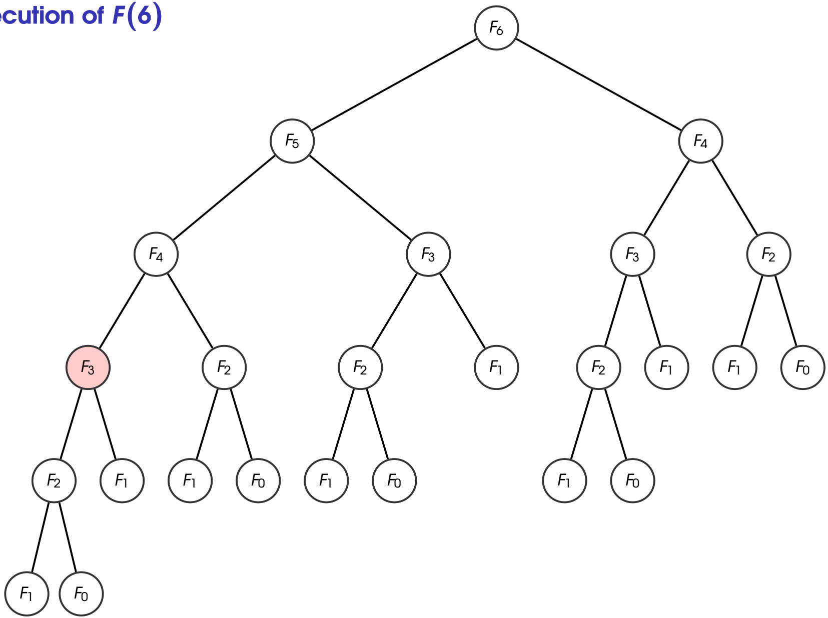
$v[0]$	1
$v[1]$	1
$v[2]$	-1
$v[3]$	-1
$v[4]$	-1
$v[5]$	-1
$v[6]$	-1

Execution of $F(6)$



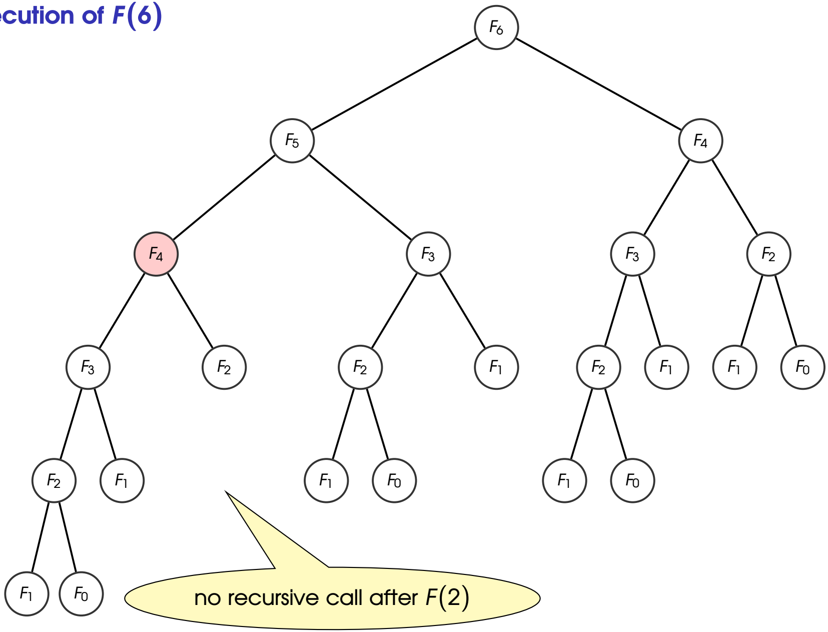
$v[0]$	1
$v[1]$	1
$v[2]$	2
$v[3]$	-1
$v[4]$	-1
$v[5]$	-1
$v[6]$	-1

Execution of $F(6)$



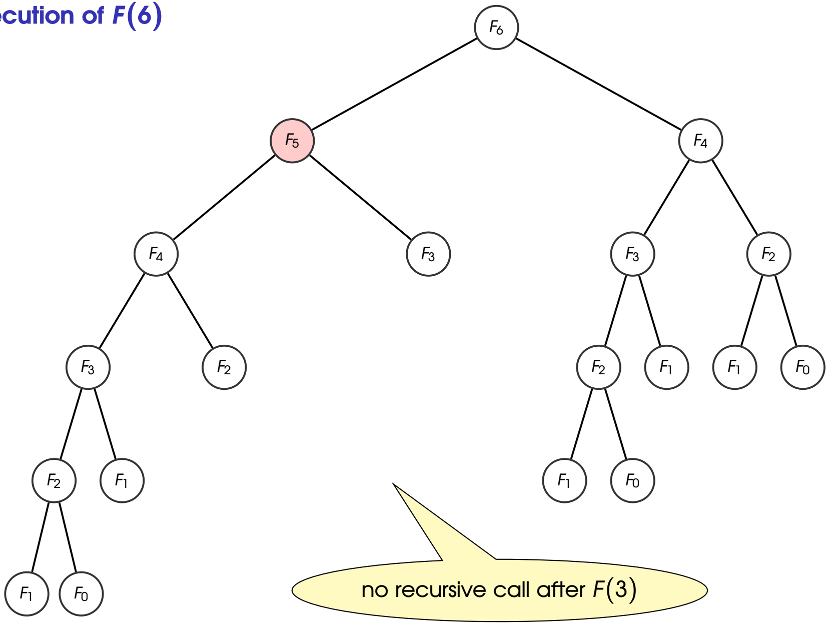
$v[0]$	1
$v[1]$	1
$v[2]$	2
$v[3]$	3
$v[4]$	-1
$v[5]$	-1
$v[6]$	-1

Execution of $F(6)$



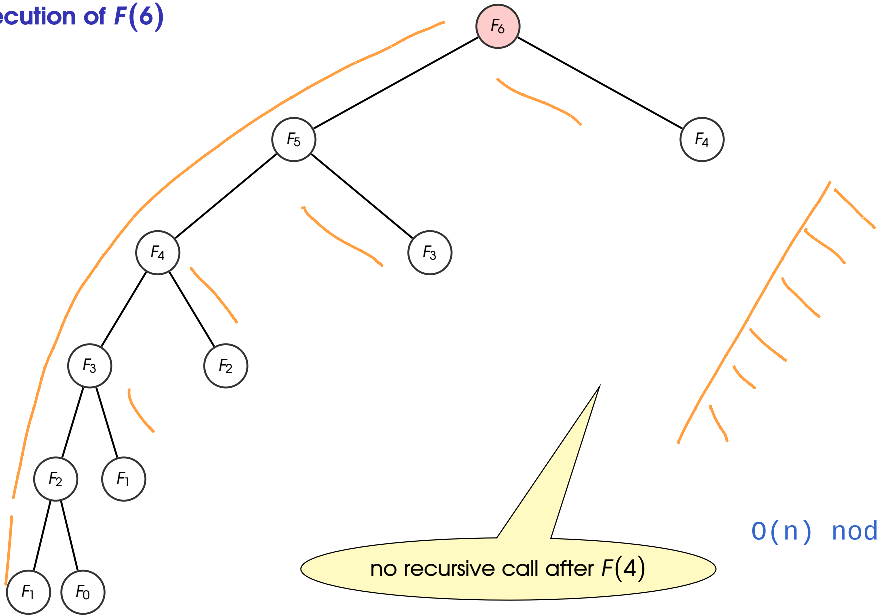
$v[0]$	1
$v[1]$	1
$v[2]$	2
$v[3]$	3
$v[4]$	5
$v[5]$	-1
$v[6]$	-1

Execution of $F(6)$



$v[0]$	1
$v[1]$	1
$v[2]$	2
$v[3]$	3
$v[4]$	5
$v[5]$	8
$v[6]$	-1

Execution of $F(6)$



$v[0]$	1
$v[1]$	1
$v[2]$	2
$v[3]$	3
$v[4]$	5
$v[5]$	8
$v[6]$	13

O(n) nodes

Can we do even better?

Can we do even better?

Observation

- The 2nd version still makes many function calls,
- each wastes time in parameters passing, dynamic linking, $\cdots$
- In general, to compute $F(i)$, we need $F(i-1)$ & $F(i-2)$ only

Can we do even better?

Observation

- The 2nd version still makes many function calls,
- each wastes time in parameters passing, dynamic linking, $\cdots$
- In general, to compute $F(i)$, we need $F(i-1)$ & $F(i-2)$ only

Idea to further improve

- Compute the values in bottom-up fashion.
- That is, compute $F(2)$ (we already know $F(0)$ and $F(1)$ are both 1),
- Then $F(3)$, then $F(4)\cdots$

Can we do even better?

Observation

- The 2nd version still makes many function calls,
- each wastes time in parameters passing, dynamic linking, $\cdots$
- In general, to compute $F(i)$, we need $F(i-1)$ & $F(i-2)$ only

Idea to further improve

- Compute the values in bottom-up fashion.
- That is, compute $F(2)$ (we already know $F(0)$ and $F(1)$ are both 1),
- Then $F(3)$, then $F(4) \cdots$

```
Algorithm F(n)
    set v[0] ← 1, v[1] ← 1


    return v[n]
```

Can we do even better?

Observation

- The 2nd version still makes many function calls,
- each wastes time in parameters passing, dynamic linking, $\cdots$
- In general, to compute $F(i)$, we need $F(i-1)$ & $F(i-2)$ only

Idea to further improve

- Compute the values in bottom-up fashion.
- That is, compute $F(2)$ (we already know $F(0)$ and $F(1)$ are both 1),
- Then $F(3)$, then $F(4) \cdots$

Algorithm $F(n)$
  set $v[0] \leftarrow 1$, $v[1] \leftarrow 1$
  for $i \leftarrow 2$ to n do
    $v[i] \leftarrow v[i-1] + v[i-2]$
  return $v[n]$

Can we do even better?

Observation

- The 2nd version still makes many function calls,
- each wastes time in parameters passing, dynamic linking, $\cdots$
- In general, to compute $F(i)$, we need $F(i-1)$ & $F(i-2)$ only

Idea to further improve

- Compute the values in bottom-up fashion.
- That is, compute $F(2)$ (we already know $F(0)$ and $F(1)$ are both 1),
- Then $F(3)$, then $F(4) \cdots$

saves lots of overhead

Algorithm $F(n)$
  set $v[0] \leftarrow 1$, $v[1] \leftarrow 1$
  for $i \leftarrow 2$ to n do
    $v[i] \leftarrow v[i-1] + v[i-2]$
  return $v[n]$

Recursive vs DP approach

Recursive algorithm:

no loop

Algorithm $F(n)$
  if $n == 0$ OR $n == 1$ then
    return 1
  else
    return $F(n-1) + F(n-2)$

function call

Dynamic programming algorithm:

Algorithm $F(n)$
  set $v[0] \leftarrow 1$, $v[1] \leftarrow 1$
  for $i \leftarrow 2$ to n do
    $v[i] \leftarrow v[i-1] + v[i-2]$
  return $v[n]$

loop

array

Recursive vs DP approach

Recursive algorithm:

Algorithm $F(n)$
    if $n == 0$ OR $n == 1$ then
        return 1
    else
        return $F(n-1) + F(n-2)$

SLOW! exponential time

Dynamic programming algorithm:

Algorithm $F(n)$
    set $v[0] \leftarrow 1$, $v[1] \leftarrow 1$
    for $i \leftarrow 2$ to n do
        $v[i] \leftarrow v[i-1] + v[i-2]$
    return $v[n]$

FAST! $O(n)$-time

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be stored and retrieved easily in a table (i.e., array)

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be stored and retrieved easily in a table (i.e., array)
- Fill the table in some bottom-up manner; start filling the solution of the smallest problem.

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be stored and retrieved easily in a table (i.e., array)
- Fill the table in some bottom-up manner; start filling the solution of the smallest problem.
  - ensures that when we solve a particular sub-problem, solutions of all smaller sub-problems on which it depends are available

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be stored and retrieved easily in a table (i.e., array)
- Fill the table in some bottom-up manner; start filling the solution of the smallest problem.
  - ensures that when we solve a particular sub-problem, solutions of all smaller sub-problems on which it depends are available

For historical reasons, we call such methodology
Dynamic Programming.
In the late 40's (when computers were rare),
programming refers to the ``tabular method''.

Summary

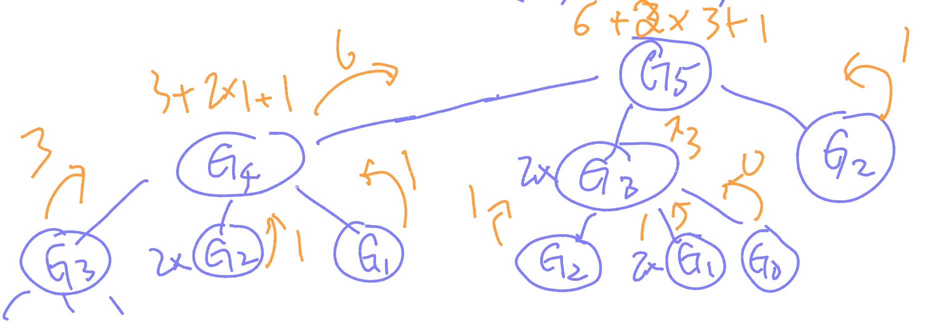
Summary: Dynamic Programming for Fibonacci Numbers

Next: Assembly Line Scheduling

For note taking

$$G(n) = \begin{cases} 0 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \text{ or } 2 \\ G(n-1) + 2\times G(n-2) + G(n-3) & \text{if } n>2 \end{cases}$$

$G(6)$? $G(5) + 2\times G(4) + G(3)$



```
recursiveG(n)
  if n==0  then
    return 0
  else if n== 1 or n==2 then
       return 1
    else
       return G(n-1) + 2 * G(n-2) + G(n-3)
```

```
nonRecursiveG(n)
  v[0] <- 0
  v[1] <- 1
  v[2] <- 1
  for i <- 3 to n
     v[i] <- v[i-1] + 2*v[i-2] + v[i-3]
  return v[n]
```