جامعة طرابلس - كلية تقنية المعلومات

Design and Analysis Algorithms

تصميم و تحليل خوارزميات

ITGS301

Lecture 9 : المحاضرة التاسعة

Marwa Solla
m.solla@uot.edu.ly

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (“*Programming” in this context refers to a tabular method, not to writing computer code.)*

divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share sub subproblems.

In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. ***A dynamic-programming algorithm solves each sub subproblem just once and then saves its answer in a table,*** thereby avoiding the work of recomputing the answer every time it solves each sub subproblem.

Ways to implement a dynamic-programming approach.

1. Memoization (top-down method)

write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).

it returns the saved value

2. Tabulation (bottom-up method).

It depends on some natural notion of the “size” of a subproblem

sort the subproblems by size and solve them in size order, smallest first.

The Fibonacci Sequence Problem.

The Fibonacci Sequence is an infinite sequence of positive integers, starting at 0 and 1, where each succeeding element is equal to the sum of its two preceding elements. If we denote the number at position n as F_n, we can formally define the Fibonacci Sequence as:

$F_n = o$ for $n = 0$

$F_n = 1$ for $n = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n > 1$

The Fibonacci Sequence

therefore, the start of the sequence is:

0, 1, 1, 2, 3, 5, 8, 13, …

So, how can we design an algorithm that returns the n^{th} number in this sequence?

0	1	2	3	4	5	6	7	8	9	10	…
0	1	1	2	3	5	8	13	21	34	55	…

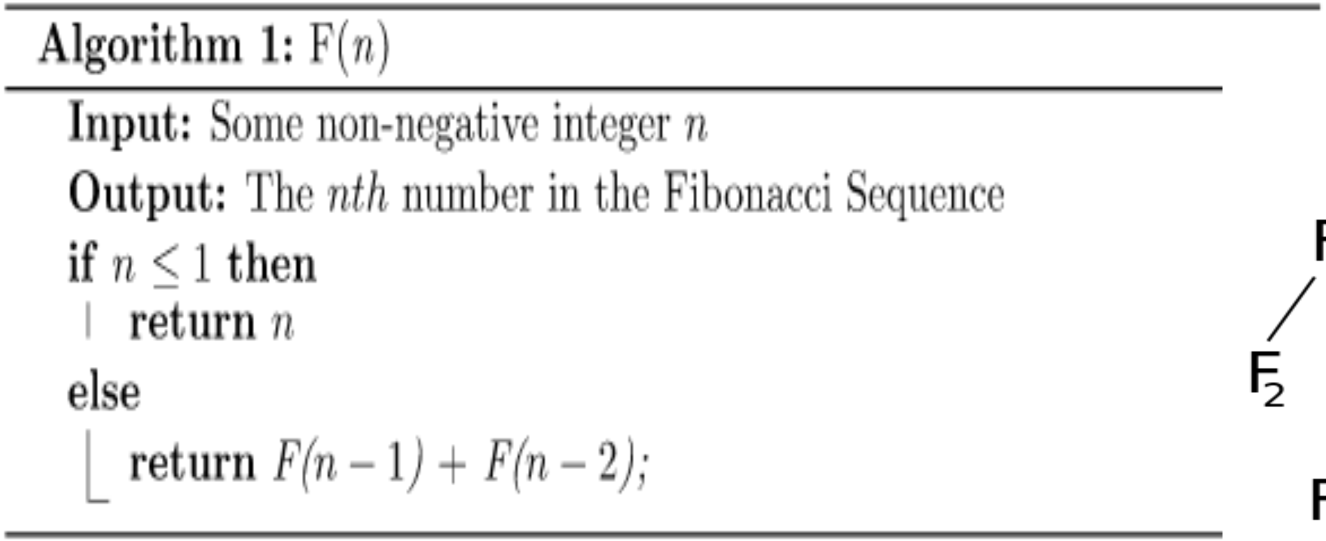
The Fibonacci sequence $f0, f1, . . .$ is recursively defined as follows: •

- base case. $f0 = 0$ and $f1 = 1$
- recursive case. for $n \geq 2$, $fn = fn-1 + fn-2$.

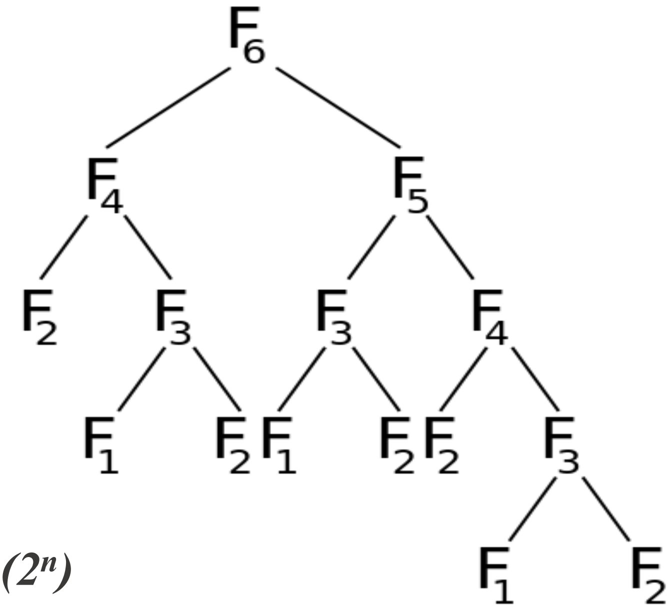
Show that the following recursive algorithm for computing the n th Fibonacci number has exponential complexity with respect to n.

EX: Algorithm 1:

Recursion algorithm



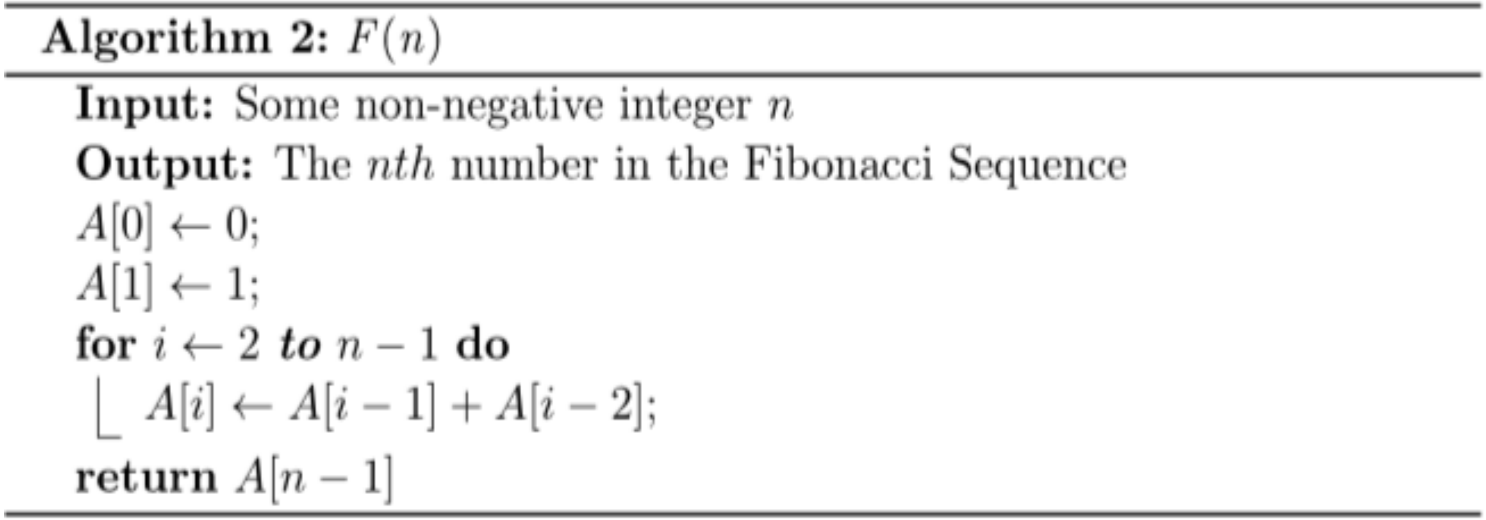
```
Algorithm 1: F(n)
Input: Some non-negative integer $n$
Output: The $nth$ number in the Fibonacci Sequence
if $n \leq 1$ then
    return $n$
else
    return $F(n-1) + F(n-2)$;
```



Complexity Time $T(n) = O(2^n)$

EX: Algorithm 2:

Tabulation (bottom-up method).



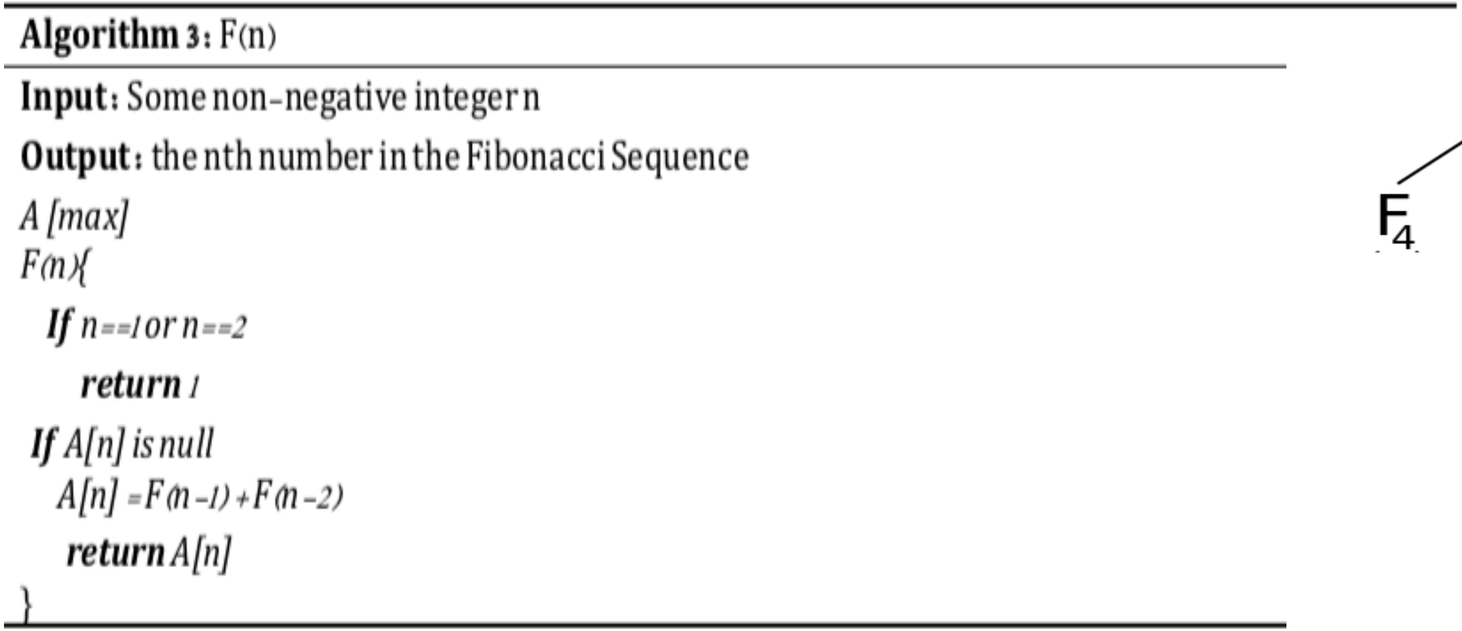
Algorithm 2: $F(n)$

```
Input: Some non-negative integer $n$
Output: The $nth$ number in the Fibonacci Sequence
$A[0] \leftarrow 0$;
$A[1] \leftarrow 1$;
for $i \leftarrow 2$ to $n-1$ do
    $A[i] \leftarrow A[i-1] + A[i-2]$;
return $A[n-1]$
```

Complexity Time $T(n) = O(n)$

EX: Algorithm 3:

Memoization (top-down method)

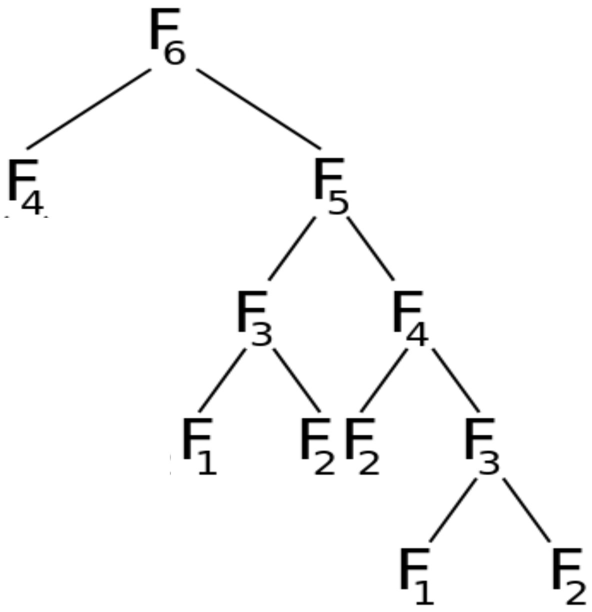


Algorithm 3: F(n)

Input: Some non-negative integer n

Output: the nth number in the Fibonacci Sequence

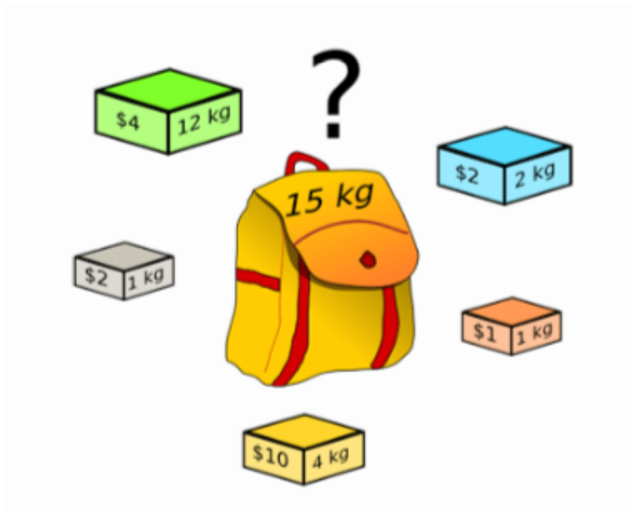
```
A [max]
F(n){
   If n==1 or n==2
      return 1
  If A[n] is null
    A[n] = F(n-1) + F(n-2)
    return A[n]
}
```



Complexity Time $T(n) = O(n)$

The Knapsack Problem

Given items $x_1, \ldots, x_n$, where item x_i has weight w_i and Value v_i (if its placed in the knapsack), determine the subset of items to place in the knapsack in order to maximize Value, assuming that the sack has capacity W.



Knapsack can be :

1. 0-1 Knapsack
2. Fractional Knapsack

0-1 Knapsack - dynamic programming

$V(i, w)$ = the maximum value that can be obtained from items *1 to i , if Knapsack has size W.*

Case 1 : takes item *i*

$V(i, w) = v_i + V(i - 1, W - w_i)$

Case 2: : does n’t takes item *i*

$V(i, w) = V(i - 1, W)$

0-1 Knapsack - dynamic programming

$$\boldsymbol{V(i\,,\,w)} = max\,\{\overbrace{v_i + V(\,i - 1,\, W - w_i\,)}^{\text{Item i was taken}}\,,\,\overbrace{V(\,i - 1,\, W\,)}^{\text{Item i was not taken}}\,\}$$

0	0	0	0	0	0	0	0
0							
0							
0							
0							

	0			W -w				W
0	0	0	0	0	0	0	0	0
	0							
i-1	0							
i	0							
	0							
n	0							

Example :

item	*weight*	*value*
1	2	12
2	1	10
3	3	20
4	2	15

Capacity W = 5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

EX1: $V(1\ ,\ 1) = V\ (0,1\ ) = 0$

EX2: $V(2\ ,\ 4) = \max\{\ 10 + V\ (2\text{-}1,4\text{-}١)\ ,\ V(2\text{-}1,\ 4)\ \}$

$V(2\ ,\ 4) = \max\{\ 10 + V\ (1,٣)\ ,\ V(1,\ 4)\ \}$

$V(2\ ,\ 4) = \max\{\ 10 + 12\ ,\ 12\ \}$

$V(2\ ,\ 4) = \max\{\ 22,\ 12\} = 22$

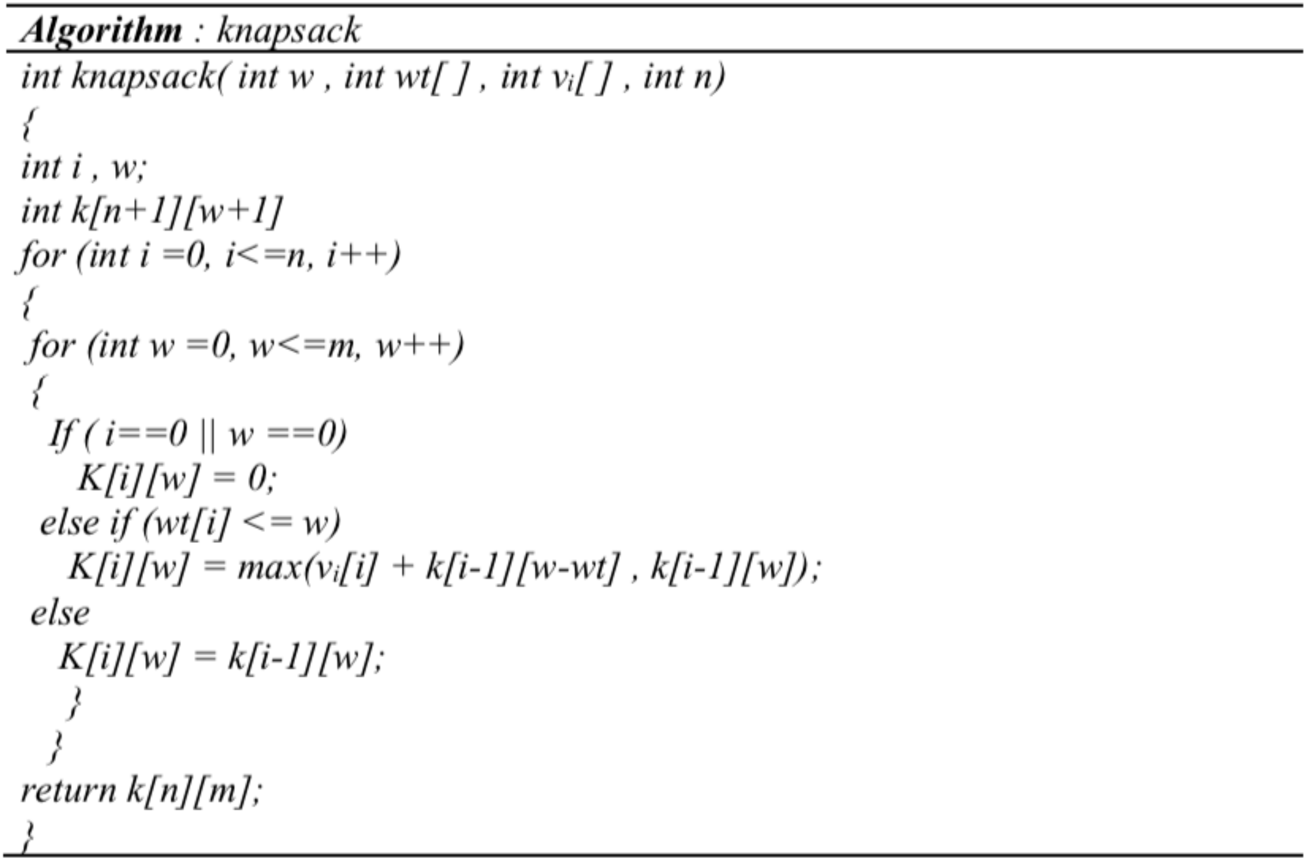
Knapsack = items (4 , 2, 1)

W = 2 + 1 + 2

W = 5 & V = 37

{ i1 , i2, i3, i4}

{ 1 , 1, 0, 1}



Algorithm : *knapsack*

```
int knapsack( int w , int wt[ ] , int v_i[ ] , int n)
{
int i , w;
int k[n+1][w+1]
for (int i =0, i<=n, i++)
{
 for (int w =0, w<=m, w++)
  {
   If ( i==0 || w ==0)
     K[i][w] = 0;
  else if (wt[i] <= w)
    K[i][w] = max(v_i[i] + k[i-1][w-wt] , k[i-1][w]);
 else
   K[i][w] = k[i-1][w];
    }
   }
return k[n][m];
}
```

The End .