CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

February 8, 2019
1. Course Information
2. Introduction
3. Divide-and-Conquer Algorithms
4. Greedy Algorithms
5. Dynamic Programming Algorithms
6. Graph Algorithms
7. Intractability and Undecidability
Table of Contents

5 Dynamic Programming Algorithms
- Fibonacci Numbers
- Design Strategy
- 0-1 Knapsack
- Coin Changing
- Longest Common Subsequence
- Minimum Length Triangulation
Computing Fibonacci Numbers Inefficiently

Algorithm: $BadFib(n)$

if $n = 0$ then $f \leftarrow 0$
else if $n = 1$ then $f \leftarrow 1$
else
    \[
    \begin{cases} 
    f_1 \leftarrow BadFib(n - 1) \\
    f_2 \leftarrow BadFib(n - 2) \\
    f \leftarrow f_1 + f_2
    \end{cases}
    \]

return $(f)$;
The Recursion Tree to Evaluate $f_5$:
Complexity of the Algorithm

The recurrence tree has $f_n$ leaf nodes with the value 1 and $f_{n-1}$ leaf nodes with the value 0. So there are a total of $f_{n+1}$ leaf nodes.

The number of interior nodes is $f_{n+1} - 1$.

In the unit cost model, the complexity of computing $f_n$ is $\Theta(f_{n+1})$.

How quickly does $f_n$ grow? Let $\phi = (1 + \sqrt{5})/2$; then

$$f_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor.$$

Therefore $f_n \in \Theta(\phi^n)$ and hence we also have $f_{n+1} \in \Theta(\phi^n)$.

The value $\phi \approx 1.6$ is the golden ratio.

The time to compute $f_n$ is exponential in $n$. 
Computing Fibonacci Numbers More Efficiently

Algorithm: BetterFib(n)

\[
\begin{align*}
    f[0] & \leftarrow 0 \\
    f[1] & \leftarrow 1 \\
    \text{for } i & \leftarrow 2 \text{ to } n \\
    \quad \text{do } f[i] & \leftarrow f[i - 1] + f[i - 2] \\
    \text{return } (f[n])
\end{align*}
\]
Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure
Examine the structure of an optimal solution to a problem instance \( I \), and determine if an optimal solution for \( I \) can be expressed in terms of optimal solutions to certain subproblems of \( I \).

Define Subproblems
Define a set of subproblems \( S(I) \) of the instance \( I \), the solution of which enables the optimal solution of \( I \) to be computed. \( I \) will be the last or largest instance in the set \( S(I) \).
Recurrence Relation

Derive a \textit{recurrence relation} on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. Compute these solutions using the recurrence relation in a \textit{bottom-up} fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to $I$. 

Problem 5.1

0-1 Knapsack

Instance: Profits $P = [p_1, \ldots, p_n]$; weights $W = [w_1, \ldots, w_n]$; and a capacity, $M$. These are all positive integers.

Feasible solution: An $n$-tuple $X = [x_1, \ldots, x_n]$, where $x_i \in \{0, 1\}$ for $1 \leq i \leq n$, and $\sum_{i=1}^{n} w_i x_i \leq M$.

Find: A feasible solution $X$ that maximizes $\sum_{i=1}^{n} p_i x_i$. 
Developing a Dynamic Programming Algorithm for 0-1 Knapsack

Optimal Structure:
- Suppose $X = [x_1, \ldots, x_n]$ is an optimal solution to an instance $I$.
- If $x_n = 0$, then $X' = [x_1, \ldots, x_{n-1}]$ is the optimal solution to the instance (subproblem) with profits $p_1, \ldots, p_{n-1}$, weights $w_1, \ldots, w_{n-1}$ and capacity $M$.
- If $x_n = 1$, then $X'$ is the optimal solution to the instance (subproblem) with profits $p_1, \ldots, p_{n-1}$, weights $w_1, \ldots, w_{n-1}$ and capacity $M - w_n$.

Subproblems:
- If we apply the above analysis recursively, we consider subproblems consisting of the first $i$ objects (having profits $p_1, \ldots, p_i$ and weights $w_1, \ldots, w_i$) and capacity $m$, for all $1 \leq i \leq n$ and all $0 \leq m \leq M$.
- Let $P[i, m]$ denote the optimal profit for this subproblem. Then $P[n, M]$ is the final answer we are looking for.
Developing a Dynamic Programming Algorithm (cont.)

Recurrence Relation:

\[
P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
P[i-1, m] & \text{if } i \geq 2, \ m < w_i \\
p_1 & \text{if } i = 1, \ m \geq w_1 \\
0 & \text{if } i = 1, \ m < w_1.
\end{cases}
\]

Compute Optimal Solutions:

- We fill in the rows one at a time, beginning with row 1.
- We fill in each row of the table from left to right.
- For the last row, we only need to compute the last value \(P[n, M]\).
A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: **0-1Knapsack**($p_1, \ldots, p_n, w_1, \ldots, w_n, M$)

for $m \leftarrow 0$ to $M$

\begin{align*}
\text{if } m \geq w_1 \\
\text{do } & \{ \\
\text{then } P[1, m] \leftarrow p_1 \\
\text{else } P[1, m] \leftarrow 0
\}
\end{align*}

for $i \leftarrow 2$ to $n$

\begin{align*}
\text{for } m \leftarrow 0 \text{ to } M
\end{align*}

\begin{align*}
\text{do } \{ \\
\text{if } m < w_i \\
\text{do } & \{ \\
\text{then } P[i, m] \leftarrow P[i - 1, m] \\
\text{else } P[i, m] \leftarrow \max\{P[i - 1, m - w_i] + p_i, P[i - 1, m]\}
\}
\}
\end{align*}

return ($P[n, M]$);
Example

Suppose we have profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30.

The following table is computed:

```
  | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
---|-----------------------------------------------
 1 | 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 2 | 0 0 1 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 | 0 0 1 2 3 3 4 5 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
 4 | 0 0 1 2 3 3 4 5 5 6 7 7 8 8 9 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11 11
 5 | 0 0 1 2 3 3 4 5 5 6 7 7 8 8 9 10 10 11 11 11 11 11 11 12 12 13 14 14 15 15 16 17 17
 6 | - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 18
```

For example,

\[
P[3, 16] = \max\{P[2, 16], P[2, 11] + 3\} = \max\{3, 3 + 3\} = 6.
\]
Computing the Optimal Knapsack $X$

The optimal solution is computed by tracing back through the table.

For the previous example, consisting of profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity $30$, the optimal solution is $[1, 1, 0, 1, 0, 1]$. 

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 4 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 7 | 7 | 7 | 7 | 7 | 8 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 5 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 6 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
Computing the Optimal Knapsack $X$

**Algorithm:** \textit{ComputeOptimalKnapsack}($p_1, \ldots, p_n, w_1, \ldots, w_n, M, P$)

\[
\begin{aligned}
m & \leftarrow M \\
p & \leftarrow P[n, M] \\
& \text{for } i \leftarrow n \text{ downto } 2 \\
& \quad \left\{ \\
& \quad \quad \textbf{if } p = P[i - 1, m] \\
& \quad \quad \quad \textbf{then } x_i \leftarrow 0 \\
& \quad \quad \textbf{do } \\
& \quad \quad \quad x_i \leftarrow 1 \\
& \quad \quad \textbf{else } \\
& \quad \quad \quad p \leftarrow p - p_i \\
& \quad \quad \quad m \leftarrow m - w_i \\
& \quad \textbf{if } p = 0 \\
& \quad \quad \textbf{then } x_1 \leftarrow 0 \\
& \quad \textbf{else } x_1 \leftarrow 1 \\
& \text{return } (X);
\end{aligned}
\]
Complexity of the Algorithm

Suppose we assume the unit cost model, so additions / subtractions take time $O(1)$.

The complexity to construct the table is $\Theta(nM)$

Is this a polynomial-time algorithm, as a function of the size of the problem instance?

We have

$$\text{size}(I) = \log_2 M + \sum_{i=1}^{n} \log_2 w_i + \sum_{i=1}^{n} \log_2 p_i.$$ 

Note in particular that $M$ is exponentially large compared to $\log_2 M$. So constructing the table is not a polynomial-time algorithm, even in the unit cost model.

What would the complexity of a recursive algorithm be?
Coin Changing

Problem 5.2

Coin Changing

Instance: A list of coin denominations, \(1 = d_1, d_2, \ldots, d_n\), and a positive integer \(T\), which is called the target sum.

Find: An \(n\)-tuple of non-negative integers, say \(A = [a_1, \ldots, a_n]\), such that \(T = \sum_{i=1}^{n} a_i d_i\) and such that \(N = \sum_{i=1}^{n} a_i\) is minimized.

What subproblems should be considered?

What table of values should we fill in?

What is the complexity of the algorithm?

How do we compute the optimal set of coins (in addition to the number of coins).pause
A Dynamic Programming Algorithm for Coin Changing

Algorithm: *Coin Changing* \((d_1, \ldots, d_n, T)\)

comment: \(d_1 = 1\)

for \(t \leftarrow 0\) to \(T\)

\[
\begin{align*}
N[1, t] & \leftarrow t \\
A[1, t] & \leftarrow t
\end{align*}
\]

for \(i \leftarrow 2\) to \(n\)

for \(t \leftarrow 0\) to \(T\)

\[
\begin{align*}
N[i, t] & \leftarrow N[i - 1, t] \\
A[i, t] & \leftarrow 0
\end{align*}
\]

for \(j \leftarrow 1\) to \(\lceil t/d_i \rceil\)

\[
\begin{align*}
\text{if } j + N[i - 1, t - jd_i] & < N[i, t] \\
\text{then } N[i, t] & \leftarrow j + N[i - 1, t - jd_i] \\
A[i, t] & \leftarrow j
\end{align*}
\]

return \((N[n, T])\)
Computing the Optimal Set of Coins

We trace back through the table to compute the optimal set of coins. There are two possible approaches:

1. Recompute the relevant table entries $N[i, t]$ during the traceback
2. Store relevant extra information, while the table $N[i, t]$ is being constructed, in another table $A[i, t]$.

Suppose we follow the second approach.

The $A[i, t]$ values make it easy to determine number of coins of each denomination in the optimal solution $N[i, T]$.

This is kind of similar to 0-1 Knapsack.
Longest Common Subsequence

Problem 5.3

Longest Common Subsequence

Instance: Two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ over some finite alphabet $\Gamma$.

Find: A maximum length sequence $Z$ that is a subsequence of both $X$ and $Y$.

$Z = (z_1, \ldots, z_\ell)$ is a subsequence of $X$ if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}, 1 \leq j \leq \ell$.

Similarly, $Z$ is a subsequence of $Y$ if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}, 1 \leq j \leq \ell$. 
Computing the Length of the LCS of $X$ and $Y$

Consider $X' = (x_1, \ldots, x_{m-1})$ and $Y' = (y_1, \ldots, y_{n-1})$.

1. If $x_m = y_n$, then $\text{LCS}(X, Y) = 1 + \text{LCS}(X', Y')$ (the LCS ends with $x_m = y_n$).
2. If $x_m \neq y_n$, then $\text{LCS}(X, Y) = \max\{\text{LCS}(X, Y'), \text{LCS}(X', Y)\}$.

We consider subproblems consisting of all possible prefixes of $X$ and $Y$. Let $c[i, j]$ denote the length of the LCS of $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$. If $i = 0$ or $j = 0$, then we are considering the “empty prefix” of $X$ or $Y$ (respectively).

The optimal solution to the original problem instance is $c[m, n]$.

We have the following recurrence relation:

$$c[i, j] = \begin{cases} 
c[i - 1, j - 1] + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
\max\{c[i - 1, j], c[i, j - 1]\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j \\
0 & \text{if } i = 0 \text{ or } j = 0.
\end{cases}$$
Computing the Length of the LCS of $X$ and $Y$

**Algorithm: $LCS1(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$**

for $i \leftarrow 0$ to $m$
    do $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ to $n$
    do $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ to $m$
    for $j \leftarrow 1$ to $n$
        do $\begin{cases} 
        \text{if } x_i = y_j \\
        \text{then } c[i, j] \leftarrow c[i - 1, j - 1] + 1 \\
        \text{else } c[i, j] \leftarrow \max\{c[i, j - 1], c[i - 1, j]\} \end{cases}$

return $(c[m, n])$;
Finding the LCS of $X$ and $Y$

Algorithm: $LCS2(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ to $m$ do $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ to $n$ do $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ to $m$

for $j \leftarrow 1$ to $n$

if $x_i = y_j$

then

\[
\begin{align*}
&\{ c[i, j] \leftarrow c[i - 1, j - 1] + 1 \\
&\{ \pi[i, j] \leftarrow \text{UL}
\end{align*}
\]

else if $c[i, j - 1] > c[i - 1, j]$

then

\[
\begin{align*}
&\{ c[i, j] \leftarrow c[i, j - 1] \\
&\{ \pi[i, j] \leftarrow \text{L}
\end{align*}
\]

else

\[
\begin{align*}
&\{ c[i, j] \leftarrow c[i - 1, j] \\
&\{ \pi[i, j] \leftarrow \text{U}
\end{align*}
\]

return $(c, \pi)$;
Finding the LCS

Algorithm: \textit{FindLCS}(c, \pi, v)

\begin{align*}
\text{seq} & \leftarrow () \\
i & \leftarrow m \\
j & \leftarrow n \\
\text{while} & \min\{i, j\} > 0 \\
\begin{cases}
\text{if } \pi[i, j] = \text{UL} \\
\text{do}
\begin{cases}
\text{then} & \{\text{seq} \leftarrow x_i \parallel \text{seq} \\
i & \leftarrow i - 1 \\
j & \leftarrow j - 1
\end{cases} \\
\text{else if } \pi[i, j] = \text{L} & \text{then } j \leftarrow j - 1 \\
\text{else} & i \leftarrow i - 1
\end{cases}
\end{cases}
\end{align*}

return \( (\text{seq}) \)
LCS Example

Suppose $X = gdvegta$ and $Y = gvcekst$.

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>i = 0</th>
<th>g</th>
<th>d</th>
<th>v</th>
<th>e</th>
<th>g</th>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>= 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>0</td>
<td>↖ 1</td>
<td>← 1</td>
<td>← 1</td>
<td>← 1</td>
<td>↖ 1</td>
<td>← 1</td>
<td>← 1</td>
</tr>
<tr>
<td>v</td>
<td>2</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↖ 2</td>
<td>← 2</td>
<td>← 2</td>
<td>← 2</td>
<td>← 2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↑ 2</td>
<td>↑ 2</td>
<td>↑ 2</td>
<td>↑ 2</td>
<td>↑ 2</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↑ 2</td>
<td>↖ 3</td>
<td>← 3</td>
<td>← 3</td>
<td>← 3</td>
</tr>
<tr>
<td>k</td>
<td>5</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↑ 2</td>
<td>↑ 3</td>
<td>↑ 3</td>
<td>↑ 3</td>
<td>↑ 3</td>
</tr>
<tr>
<td>s</td>
<td>6</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↑ 2</td>
<td>↑ 3</td>
<td>↑ 3</td>
<td>↑ 3</td>
<td>↑ 3</td>
</tr>
<tr>
<td>t</td>
<td>7</td>
<td>0</td>
<td>↑ 1</td>
<td>↑ 1</td>
<td>↑ 2</td>
<td>↑ 3</td>
<td>↑ 3</td>
<td>↖ 4</td>
<td>← 4</td>
</tr>
</tbody>
</table>
Minimum Length Triangulation

Problem 5.4

Minimum Length Triangulation v1

Instance: \( n \) points \( q_1, \ldots, q_n \) in the Euclidean plane that form a convex \( n \)-gon \( P \).

Find: A triangulation of \( P \) such that the sum \( S_c \) of the lengths of the \( n - 3 \) chords is minimized.

Problem 5.5

Minimum Length Triangulation v2

Instance: \( n \) points \( q_1, \ldots, q_n \) in the Euclidean plane that form a convex \( n \)-gon \( P \).

Find: A triangulation of \( P \) such that the sum \( S_p \) of the perimeters of the \( n - 2 \) triangles is minimized.

Let \( L \) denote the perimeter of \( P \). Then we have that \( S_p = L + 2S_c \).

Hence the two versions have the same optimal solutions.
Problem Decomposition

We consider version 2 of the problem.

The edge $q_nq_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n - 1\}$.

For a given $k$, we have:

1. the triangle $q_1q_kq_n$,
2. the polygon with vertices $q_1, \ldots, q_k$,
3. the polygon with vertices $q_k, \ldots, q_n$.

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).
Recurrence Relation

For $1 \leq i < j \leq n$, let $S[i, j]$ denote the optimal solution to the subproblem consisting of the polygon having vertices $q_i, \ldots, q_j$.

Let $\Delta(q_i, q_k, q_j)$ denote the perimeter of the triangle having vertices $q_i, q_k, q_j$.

The we have the recurrence relation

$$S[i, j] = \min \{\Delta(q_i, q_k, q_j) + S[i, k] + S[k, j] : i < k < j\}.$$ 

The base cases are given by

$$S[i, i + 1] = 0$$

for all $i$.

We compute all $S[i, j]$ with $j - i = c$, for $c = 2, 3, \ldots, n - 1$. 
Memoization

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal.

Memoization is a recursive algorithm based on same recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.
Example: Computing the Fibonacci Numbers

Algorithm: \( \text{MemoFib}(n) \)

procedure \( \text{RecFib}(n) \)

\[
\begin{align*}
\text{if } n &= 0 \quad \text{then } f \leftarrow 0 \\
\text{else if } n &= 1 \quad \text{then } f \leftarrow 1 \\
\text{else if } M[n] &\neq -1 \quad \text{then } f \leftarrow M[n] \\
\text{else } \quad& \\
&\begin{cases} 
 f_1 \leftarrow \text{RecFib}(n - 1) \\
 f_2 \leftarrow \text{RecFib}(n - 2) \\
 f \leftarrow f_1 + f_2 \\
 M[n] \leftarrow f 
\end{cases} \\
\text{return } (f); \\
\end{align*}
\]

main

\[
\begin{align*}
\text{for } i &\leftarrow 2 \text{ to } n \\
&\quad \text{do } M[i] \leftarrow -1 \\
\text{return } (\text{RecFib}(n))
\end{align*}
\]
Complexity

Memoization reduces the size of the recursion tree to $\Theta(n)$. 