Dynamic Programming

Lecture 13
Thursday, October 12, 2017
Part I

Recursion and Memoization
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1. \]

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly*

1. \[ F(n) = \left( \phi^n - (1 - \phi)^n \right) / \sqrt{5} \text{ where } \phi \text{ is the golden ratio} \]
   \[ (1 + \sqrt{5})/2 \approx 1.618. \]

2. \[ \lim_{n \to \infty} \frac{F(n + 1)}{F(n)} = \phi \]
Consider the $n$th Fibonacci number $F(n)$. Writing the number $F(n)$ in base 2 requires

(A) $\Theta(n^2)$ bits.
(B) $\Theta(n)$ bits.
(C) $\Theta(\log n)$ bits.
(D) $\Theta(\log \log n)$ bits.
Recursive Algorithm for Fibonacci Numbers

**Question:** Given $n$, compute $F(n)$.

$$
\text{Fib}(n) :=
\begin{cases}
  \text{if } (n = 0) & \text{return } 0 \\
  \text{else if } (n = 1) & \text{return } 1 \\
  \text{else} & \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{cases}
$$

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$
T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0
$$
Recursive Algorithm for Fibonacci Numbers

**Question:** Given \( n \), compute \( F(n) \).

\[
\text{Fib}(n) :
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{else if } (n = 1) & \quad \text{return } 1 \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

Running time? Let \( T(n) \) be the number of additions in \( \text{Fib}(n) \).

\[
T(n) = T(n - 1) + T(n - 2) + 1 \quad \text{and} \quad T(0) = T(1) = 0
\]
Question: Given $n$, compute $F(n)$.

\[
\text{Fib}(n):
\begin{align*}
\text{if} \ (n = 0) & \quad \text{return} \ 0 \\
\text{else if} \ (n = 1) & \quad \text{return} \ 1 \\
\text{else} & \quad \text{return} \ \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

Running time? Let $T(n)$ be the number of additions in Fib(n).

\[T(n) = T(n - 1) + T(n - 2) + 1 \quad \text{and} \quad T(0) = T(1) = 0\]
Recursive Algorithm for Fibonacci Numbers

Question: Given $n$, compute $F(n)$.

\[
\text{Fib}(n):
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{else if } (n = 1) & \quad \text{return } 1 \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

Running time? Let $T(n)$ be the number of additions in Fib(n).

\[
T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0
\]

Roughly same as $F(n)$

\[
T(n) = \Theta(\phi^n)
\]

The number of additions is exponential in $n$. Can we do better?
Recursion tree for the Recursive Fibonacci

0  1
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci

Sariel Har-Peled (UIUC)
Recursion tree for the Recursive Fibonacci
Recursion tree for the Recursive Fibonacci
An iterative algorithm for Fibonacci numbers

```
FibIter(n):
    if (n == 0) then
        return 0
    if (n == 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

What is the running time of the algorithm? $O(n)$ additions.
An iterative algorithm for Fibonacci numbers

\[
\text{FibIter}(n) : \\
\text{if } (n = 0) \text{ then} \\
\quad \text{return } 0 \\
\text{if } (n = 1) \text{ then} \\
\quad \text{return } 1 \\
F[0] = 0 \\
F[1] = 1 \\
\text{for } i = 2 \text{ to } n \text{ do} \\
\quad F[i] = F[i - 1] + F[i - 2] \\
\text{return } F[n]
\]

What is the running time of the algorithm? \( O(n) \) additions.
An iterative algorithm for Fibonacci numbers

\[
\text{FibIter}(n): \\
\text{if } (n = 0) \text{ then return } 0 \\
\text{if } (n = 1) \text{ then return } 1 \\
F[0] = 0 \\
F[1] = 1 \\
\text{for } i = 2 \text{ to } n \text{ do} \\
\quad F[i] = F[i - 1] + F[i - 2] \\
\text{return } F[n]
\]

What is the running time of the algorithm? \(O(n)\) additions.
What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:
Finding a recursion that can be effectively/efficiently memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
What is the difference?

1. Recursive algorithm is computing the same numbers again and again.

2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:

Finding a recursion that can be effectively/efficiently memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
What is the difference?

1. Recursive algorithm is computing the same numbers again and again.

2. Iterative algorithm is storing computed values and building bottom up the final value. **Memoization**.

**Dynamic Programming:**

Finding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n):
\begin{align*}
\text{if} \ (n = 0) & \quad \text{return} \ 0 \\
\text{if} \ (n = 1) & \quad \text{return} \ 1 \\
\text{if} \ (\text{Fib}(n) \text{ was previously computed}) & \quad \text{return} \ \text{stored value of Fib}(n) \\
\text{else} & \quad \text{return} \ \text{Fib}(n - 1) + \ \text{Fib}(n - 2)
\end{align*}
\]

How do we keep track of previously computed values? Two methods: explicitly and implicitly (via data structure)
Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n): \\
\quad \text{if } (n = 0) \quad \text{return } 0 \\
\quad \text{if } (n = 1) \quad \text{return } 1 \\
\quad \text{if } (\text{Fib}(n) \text{ was previously computed}) \quad \text{return stored value of Fib}(n) \\
\quad \text{else} \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\]

How do we keep track of previously computed values? Two methods: explicitly and implicitly (via data structure)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
Fib(n) :
\begin{align*}
& \text{if } (n = 0) \\
& \quad \text{return } 0 \\
& \text{if } (n = 1) \\
& \quad \text{return } 1 \\
& \text{if } (\text{Fib}(n) \text{ was previously computed}) \\
& \quad \text{return stored value of Fib}(n) \\
& \text{else} \\
& \quad \text{return Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n):
\]

- if \( n = 0 \) 
  - return 0
- if \( n = 1 \) 
  - return 1
- if (\text{Fib}(n) \text{ was previously computed}) 
  - return stored value of \text{Fib}(n)
- else 
  - return \text{Fib}(n - 1) + \text{Fib}(n - 2)

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)
Automatic implicit memoization

Initialize a (dynamic) dictionary data structure \( D \) to empty

\[
\text{Fib}(n) : \\
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{if } (n = 1) & \quad \text{return } 1 \\
\text{if } (n \text{ is already in } D) & \quad \text{return value stored with } n \text{ in } D \\
\text{val} & \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \\
\text{Store } (n, \text{val}) \text{ in } D \\
\text{return } \text{val}
\end{align*}
\]

Use hash-table or a map to remember which values were already computed.
Automatic explicit memoization

1. Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$.

2. Resulting code:

   ```
   Fib(n):
   
   if ($n = 0$)
       return 0
   if ($n = 1$)
       return 1
   if ($M[n] \neq -1$) // $M[n]$: stored value of Fib(n)
       return $M[n]
   
   $M[n] \leftarrow$ Fib($n - 1$) + Fib($n - 2$)
   return $M[n]
   ```

3. Need to know upfront the number of subproblems to allocate memory.
Automatic explicit memoization

1. Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$.

2. Resulting code:

   $Fib(n)$:
   
   ```
   if ($n = 0$)
       return 0
   if ($n = 1$)
       return 1
   if ($M[n] \neq -1$) // $M[n]$: stored value of $Fib(n)$
       return $M[n]$
   $M[n] \leftarrow Fib(n - 1) + Fib(n - 2)$
   return $M[n]$
   ```

3. Need to know upfront the number of subproblems to allocate memory.
Automatic explicit memoization

1. Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$.

2. Resulting code:

   \[
   \text{Fib}(n) : \\
   \begin{align*}
   \text{if } (n = 0) & \quad \text{return } 0 \\
   \text{if } (n = 1) & \quad \text{return } 1 \\
   \text{if } (M[n] \neq -1) \quad & \quad \text{// } M[n]: \text{ stored value of Fib}(n) \\
   \text{return } M[n] \\
   M[n] \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \\
   \text{return } M[n]
   \end{align*}
   \]

3. Need to know upfront the number of subproblems to allocate memory.
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fib...
Recursion tree for the memoized Fibonacci sequence.
Automatic Memoization

1. Recursive version:

\[
f(x_1, x_2, \ldots, x_d) :
\]

CODE

2. Recursive version with memoization:

\[
g(x_1, x_2, \ldots, x_d) :
\]

if \( f \) already computed for \((x_1, x_2, \ldots, x_d)\) then

    return value already computed

NEW_CODE

3. NEW_CODE:

   1. Replaces any “return \( \alpha \)” with
   2. Remember “\( f(x_1, \ldots, x_d) = \alpha \)” ; return \( \alpha \).
Automatic Memoization

1. Recursive version:

```plaintext
f(x_1, x_2, \ldots, x_d):
```

2. Recursive version with memoization:

```plaintext
g(x_1, x_2, \ldots, x_d):
    if f already computed for (x_1, x_2, \ldots, x_d) then
        return value already computed
```

3. NEW_CODE:

   1. Replaces any “return α” with
   2. Remember “f(x_1, \ldots, x_d) = α”; return α.
Automatic Memoization

1. Recursive version:

   \[ f(x_1, x_2, \ldots, x_d) : \]

   CODE

2. Recursive version with memoization:

   \[ g(x_1, x_2, \ldots, x_d) : \]
   
   if \( f \) already computed for \((x_1, x_2, \ldots, x_d)\) then
   
   return value already computed

   NEW_CODE

3. NEW_CODE:

   1. Replaces any "return \( \alpha \)" with
   2. Remember "\( f(x_1, \ldots, x_d) = \alpha \); return \( \alpha \)."
Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.

2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.

2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

1 Explicit memoization (iterative algorithm) preferred:
   1 analyze problem ahead of time
   2 Allows for efficient memory allocation and access.

2 Implicit (automatic) memoization:
   1 Problem structure or algorithm is not well understood.
   2 Need to pay overhead of data-structure.
   3 Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.

2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.

2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.
Automatic explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $i = 0, \ldots, n$.

\[
\text{Fib}(n) : \\
\quad \text{if } (n = 0) \\
\qquad \text{return 0} \\
\quad \text{if } (n = 1) \\
\qquad \text{return 1} \\
\quad \text{if } (M[n] \neq -1) (* M[n] \text{ has stored value of Fib}(n) *) \\
\qquad \text{return } M[n] \\
\quad M[n] \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \\
\quad \text{return } M[n]
\]

To allocate memory need to know upfront the number of subproblems for a given input size $n$. 
Automatic explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $i = 0, \ldots, n$.

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (M[n] \neq -1) (* M[n] has stored value of Fib(n) *)
        return M[n]
    M[n] \leftarrow Fib(n - 1) + Fib(n - 2)
    return M[n]
```

To allocate memory need to know upfront the number of subproblems for a given input size $n$. 

Automatic implicit memoization

Initialize a (dynamic) dictionary data structure $D$ to empty

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (n is already in D)
        return value stored with n in D
    val ← Fib(n − 1) + Fib(n − 2)
    Store (n, val) in D
    return val
```
Explicit vs Implicit Memoization

1. Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.

2. Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system.
   1. Need to pay overhead of data-structure.
   2. Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.
How many distinct calls does \( \text{binom}(n, \lfloor n/2 \rfloor) \) makes during its recursive execution?

(A) \( \Theta(1) \).

(B) \( \Theta(n) \).

(C) \( \Theta(n \log n) \).

(D) \( \Theta(n^2) \).

(E) \( \Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right) \).

That is, if the algorithm calls recursively \( \text{binom}(17, 5) \) about 5000 times during the computation, we count this is a single distinct call.
Running time of memoized binom?

\[ D: \text{ Initially an empty dictionary.} \]

\[ \text{binomM}(t, b) \quad // \text{computes } \binom{t}{b} \]

\[ \begin{align*}
&\text{if } b = t \text{ then return } 1 \\
&\text{if } b = 0 \text{ then return } 0 \\
&\text{if } D[t, b] \text{ is defined then return } D[t, b] \\
&D[t, b] \leftarrow \text{binomM}(t - 1, b - 1) + \text{binomM}(t - 1, b). \\
&\text{return } D[t, b]
\end{align*} \]

Assuming that every arithmetic operation takes \( O(1) \) time, what is the running time of \( \text{binomM}(n, \lfloor n/2 \rfloor) \)?

(A) \( \Theta(1) \).

(B) \( \Theta(n) \).

(C) \( \Theta(n^2) \).

(D) \( \Theta(n^3) \).

(E) \( \Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right) \).
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

1. input is $n$ and hence input size is $\Theta(\log n)$
2. output is $F(n)$ and output size is $\Theta(n)$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
Is the iterative algorithm a \textit{polynomial} time algorithm? Does it take $O(n)$ time?

1. input is $n$ and hence input size is $\Theta(\log n)$
2. output is $F(n)$ and output size is $\Theta(n)$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
Saving space. Do we need an array of $n$ numbers? Not really.

```plaintext
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp
    return prev1
```
Part II

Dynamic programming
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $foo(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $foo(x)$ generates is at most $A(n)$
- $foo(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we memoize the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Q: What is an upper bound on the running time of memoized version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$. 

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program \( \text{foo}(x) \) that takes an input \( x \).

- On input of size \( n \) the number of distinct sub-problems that \( \text{foo}(x) \) generates is at most \( A(n) \)
- \( \text{foo}(x) \) spends at most \( B(n) \) time not counting the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes \( O(1) \) time.

**Q:** What is an upper bound on the running time of memoized version of \( \text{foo}(x) \) if \( |x| = n \)? \( O(A(n)B(n)) \).
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

Suppose we *memoize* the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Q:** What is an upper bound on the running time of memoized version of $\text{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$. 
Dynamic Programming

Dynamic Programming is **smart recursion plus memoization**

**Question:** Suppose we have a recursive program \( \text{foo}(x) \) that takes an input \( x \).

- On input of size \( n \) the number of *distinct* sub-problems that \( \text{foo}(x) \) generates is at most \( A(n) \)
- \( \text{foo}(x) \) spends at most \( B(n) \) time *not counting* the time for its recursive calls.

Suppose we *memoize* the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes \( O(1) \) time.

**Q:** What is an upper bound on the running time of *memoized* version of \( \text{foo}(x) \) if \( |x| = n \)? \( O(A(n)B(n)) \).
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we memoize the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Q: What is an upper bound on the running time of memoized version of $\text{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$. 
Part III

Checking if a string is in $L^*$
Problem

Input A string \( w \in \Sigma^* \) and access to a language \( L \subseteq \Sigma^* \) via function \texttt{IsStrInL(string x)} that decides whether \( x \) is in \( L \)

Goal Decide if \( w \in L^* \) using \texttt{IsStrInL(string x)} as a black box sub-routine
Problem

**Input**  A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStrInL}(\text{string } x)$ that decides whether $x$ is in $L$.

**Goal**  Decide if $w \in L$ using $\text{IsStrInL}(\text{string } x)$ as a black box sub-routine.
Problem

**Input** A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStrInL}(\text{string } x)$ that decides whether $x$ is in $L$.

**Goal** Decide if $w \in L$ using $\text{IsStrInL}(\text{string } x)$ as a black box sub-routine.
Problem

Input  A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStrInL}(\text{string } x)$ that decides whether $x$ is in $L$

Goal  Decide if $w \in L$ using $\text{IsStrInL}(\text{string } x)$ as a black box sub-routine
Problem

Input A string \( w \in \Sigma^* \) and access to a language \( L \subseteq \Sigma^* \) via function \( \text{IsStrInL}(\text{string } x) \) that decides whether \( x \) is in \( L \)
Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStrInL} (string \ x)$ that decides whether $x$ is in $L$

Goal Decide if $w \in L^*$ using $\text{IsStrInL} (string \ x)$ as a black box sub-routine
Problem

Input A string \( w \in \Sigma^* \) and access to a language \( L \subseteq \Sigma^* \) via function \( \text{IsStrInL}(\text{string } x) \) that decides whether \( x \) is in \( L \).

Goal Decide if using \( \text{IsStrInL}(\text{string } x) \) as a black box sub-routine.

Example

Suppose \( L \) is \textit{English} and we have a procedure to check whether a string/word is in the \textit{English} dictionary.

- Is the string “isthisanenglishsentence” in \textit{English}*?  
- Is “stampstamp” in \textit{English}*?  
- Is “zibzzzad” in \textit{English}*?
Recursive Solution

When is \( w \in L^* \)?

A \( w \in L^* \) if \( w \in L \) or if \( w = uv \) where \( u \in L \) and \( v \in L^* \), \(|u| \geq 1\)

Assume \( w \) is stored in array \( A[1..n] \)

```
IsStringInLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n - 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES
        End
    End
    Output NO
```
Recursive Solution

When is \( w \in L^* \)?

A \( w \in L^* \) if \( w \in L \) or if \( w = uv \) where \( u \in L \) and \( v \in L^* \), \(|u| \geq 1\)

Assume \( w \) is stored in array \( A[1..n] \)

```plaintext
IsStringInLstar(A[1..n])

  If \( n = 0 \) Output YES
  If (IsStrInL(A[1..n]))
    Output YES
  Else
    For \( i = 1 \) to \( n - 1 \) do
      If (IsStrInL(A[1..i]) and IsStrInLstar(A[i+1..n]))
        Output YES
  Output NO
```
Recursive Solution

When is $w \in L^*$?

A $w \in L^*$ if $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$, $|u| \geq 1$

Assume $w$ is stored in array $A[1..n]$

```python
def IsStringinLstar(A[1..n]):
    If ($n = 0$) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For ($i = 1$ to $n - 1$) do
            If (IsStrInL(A[1..i]) and IsStringinLstar(A[i + 1..n]))
                Output YES
        Output NO
```
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]) :
\]
If \( n = 0 \) Output YES
If \((\text{IsStrInL}(A[1..n]))\)
  Output YES
Else
  For \( i = 1 \) to \( n - 1 \) do
    If \((\text{IsStrInL}(A[1..i]) \text{ and } \text{IsStrInLstar}(A[i + 1..n]))\)
      Output YES
  Output NO

Question: How many distinct sub-problems does \( \text{IsStrInLstar}(A[1..n]) \) generate? \( O(n) \)
Recursive Solution

Assume $w$ is stored in array $A[1..n]

\begin{align*}
\text{IsStringInLstar}(A[1..n]) : \\
&\text{If (} n = 0 \text{) Output YES} \\
&\text{If (IsStrInL}(A[1..n]) \text{) Output YES} \\
&\text{Else} \\
&\quad \text{For (} i = 1 \text{ to } n - 1 \text{) do} \\
&\quad \quad \text{If (IsStrInL}(A[1..i]) \text{ and IsStrInLstar}(A[i + 1..n]) \text{) Output YES} \\
&\text{Output NO}
\end{align*}

**Question:** How many distinct sub-problems does $\text{IsStrInLstar}(A[1..n])$ generate? $O(n)$
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]):
\]

If \( (n = 0) \) Output YES

If \( (\text{IsStrInL}(A[1..n])) \)

Output YES

Else

For \( (i = 1 \text{ to } n - 1) \) do

If \( (\text{IsStrInL}(A[1..i]) \text{ and } \text{IsStrInLstar}(A[i + 1..n])) \)

Output YES

Output NO

**Question**: How many distinct sub-problems does \( \text{IsStrInLstar}(A[1..n]) \) generate? \( O(n) \)
Consider string \textit{samiam}
Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

**ISL**(i): a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

**Base case:** ISL($n + 1$) = 1 interpreting $A[n + 1..n]$ as $\epsilon$

**Recursive relation:**

- ISL(i) = 1 if $
  \exists i < j \leq n + 1 \text{ s.t } \text{ISL}(j) \text{ and IsStrInL}(A[i..(j − 1)])$
- ISL(i) = 0 otherwise

Output: ISL(1)
Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

$\text{ISL}(i)$: a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

Base case: $\text{ISL}(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$

Recursive relation:

- $\text{ISL}(i) = 1$ if $\exists i < j \leq n + 1$ s.t $\text{ISL}(j)$ and $\text{IsStrInL}(A[i..(j - 1)])$
- $\text{ISL}(i) = 0$ otherwise

Output: $\text{ISL}(1)$
Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

**ISL($i$):** a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

**Base case:** $\text{ISL}(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$

**Recursive relation:**

- $\text{ISL}(i) = 1$ if \( \exists i < j \leq n + 1 \) s.t $\text{ISL}(j)$ and $\text{IsStrInL}(A[i..(j - 1)])$
- $\text{ISL}(i) = 0$ otherwise

**Output:** $\text{ISL}(1)$
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

**Why?** Mainly for further optimization of running time and space.

**How?**
- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

**Caveat:** Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Iterative Algorithm

**Iterative Algorithm**

```java
IsStringInLstar-Iterative(A[1..n]):
  boolean ISL[1..(n + 1)]
  ISL[n + 1] = TRUE
  for (i = n down to 1)
    ISL[i] = FALSE
    for (j = i + 1 to n + 1)
      If (ISL[j] and IsStrInL(A[i..j − 1]))
        ISL[i] = TRUE
        Break
  If (ISL[1] = 1) Output YES
  Else Output NO
```

- **Running time:** $O(n^2)$ (assuming call to `IsStrInL` is $O(1)$ time)
- **Space:** $O(n)$
Iterative Algorithm

\textbf{IsStringinLstar-Iterative}(A[1..n]):

\begin{itemize}
  \item boolean ISL[1..(n + 1)]
  \item ISL[n + 1] = TRUE
  \item for (i = n down to 1)
    \begin{itemize}
      \item ISL[i] = FALSE
      \item for (j = i + 1 to n + 1)
        \begin{itemize}
          \item If (ISL[j] and IsStrInL(A[i..j − 1]))
            \begin{itemize}
              \item ISL[i] = TRUE
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}

If (ISL[1] = 1) Output YES
Else Output NO

- \textbf{Running time}: $O(n^2)$ (assuming call to IsStrInL is $O(1)$ time)
- \textbf{Space}: $O(n)$
Iterative Algorithm

\begin{verbatim}
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n + 1)]
    ISL[n + 1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i + 1 to n + 1)
            if (ISL[j] and IsStrInL(A[i..j - 1]))
                ISL[i] = TRUE
                Break
    if (ISL[1] = 1) Output YES
    else Output NO
\end{verbatim}

- **Running time:** $O(n^2)$ (assuming call to \texttt{IsStrInL} is $O(1)$ time)
- **Space:** $O(n)$
Iterative Algorithm

\[
\text{IsStringInLstar-Iterative}(A[1..n]):
\]

boolean ISL[1..(n + 1)]

\[
\text{ISL}[n + 1] = TRUE
\]

for (\(i = n\) down to \(1\))

\[
\text{ISL}[i] = FALSE
\]

for (\(j = i + 1\) to \(n + 1\))

If (\(\text{ISL}[j]\) and \(\text{IsStrInL}(A[i..j - 1])\))

\[
\text{ISL}[i] = TRUE
\]

Break

If (\(\text{ISL}[1] = 1\)) Output YES
Else Output NO

- **Running time:** \(O(n^2)\) (assuming call to \(\text{IsStrInL}\) is \(O(1)\) time)
- **Space:** \(O(n)\)
Iterative Algorithm

\textbf{IsStringInLstar-Iterative}(A[1..n]):

\begin{align*}
\text{boolean } & \text{ ISL[1..(n + 1)]} \\
\text{ISL}[n + 1] & = \text{ TRUE} \\
\text{for } & (i = n \text { down to } 1) \\
\text{ \quad ISL[i] } & = \text{ FALSE} \\
\text{ \quad for } (j = i + 1 \text { to } n + 1) \\
\text{ \quad \quad If } (\text{ISL}[j] \text{ and IsStrInL}(A[i..j - 1])) \\
\text{ \quad \quad \quad ISL[i] } & = \text{ TRUE} \\
\text{ \quad \quad \quad Break} \\
\text{If } (\text{ISL}[1] = 1) \text{ Output YES} \\
\text{Else Output NO}
\end{align*}

- **Running time:** $O(n^2)$ (assuming call to \text{IsStrInL} is $O(1)$ time)
- **Space:** $O(n)$
Consider string \textit{samiam}
Part IV

Longest Increasing Subsequence
Sequences

**Definition**

**Sequence**: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

**Definition**

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

**Definition**

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \ldots \leq a_n$. Similarly **decreasing** and **non-increasing**.
Sequences

Example...

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9
2. Subsequence of above sequence: 5, 2, 1
3. Increasing sequence: 3, 5, 9, 17, 54
4. Decreasing sequence: 34, 21, 7, 5, 1
5. Increasing subsequence of the first sequence: 2, 7, 9.
Longest Increasing Subsequence Problem

**Input**  A sequence of numbers $a_1, a_2, \ldots, a_n$

**Goal**  Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

**Example**

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8
Longest Increasing Subsequence Problem

Input  A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal  Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

### Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8
Can we find a recursive algorithm for \( \text{LIS} \)?

\[
\text{LIS}(A[1..n]):
\]

1. Case 1: Does not contain \( A[n] \) in which case \\
   \( \text{LIS}(A[1..n]) = \text{LIS}(A[1..(n - 1)]) \)

2. Case 2: contains \( A[n] \) in which case \( \text{LIS}(A[1..n]) \) is not so clear.

**Observation**

For second case we want to find a subsequence in \( A[1..(n - 1)] \) that is restricted to numbers less than \( A[n] \). This suggests that a more general problem is \( \text{LIS\_smaller}(A[1..n], x) \) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).
Can we find a recursive algorithm for \textbf{LIS}?

\textbf{LIS}(A[1..n]):

1. Case 1: Does not contain \(A[n]\) in which case \(\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])\)

2. Case 2: contains \(A[n]\) in which case \(\text{LIS}(A[1..n])\) is not so clear.

\textbf{Observation}

For second case we want to find a subsequence in \(A[1..(n-1)]\) that is restricted to numbers less than \(A[n]\). This suggests that a more general problem is \textbf{LIS\_smaller}(A[1..n], x) which gives the longest increasing subsequence in \(A\) where each number in the sequence is less than \(x\).
Recursive Approach

$LIS(A[1..n])$: the length of longest increasing subsequence in $A$

$LIS_{\text{smaller}}(A[1..n], x)$: length of longest increasing subsequence in $A[1..n]$ with all numbers in subsequence less than $x$

\[
\text{LIS}_{\text{smaller}}(A[1..n], x) : \\
\text{if } (n = 0) \text{ then return 0} \\
m = \text{LIS}_{\text{smaller}}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
m = \max(m, 1 + \text{LIS}_{\text{smaller}}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\]

$LIS(A[1..n])$:
\[
\text{return } \text{LIS}_{\text{smaller}}(A[1..n], \infty)
\]
Example

Sequence: \( A[1..7] = 6, 3, 5, 2, 7, 8, 1 \)
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x):
\begin{align*}
\text{if } (n = 0) & \text{ then return } 0 \\
m & = \text{LIS\_smaller}(A[1..(n-1)], x) \\
\text{if } (A[n] < x) & \text{ then } \\
 & m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n-1)], A[n])) \\
\text{Output } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]): \\
\text{return } \text{LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x) :
\begin{align*}
&\text{if } (n = 0) \text{ then return 0} \\
&m = \text{LIS\_smaller}(A[1..(n - 1)], x) \\
&\text{if } (A[n] < x) \text{ then} \\
&m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]) :
\begin{align*}
&\text{return } \text{LIS\_smaller}(A[1..n], \infty)
\end{align*}
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Approach

LIS\_smaller(A[1..n], x):
  if (n = 0) then return 0
  m = LIS\_smaller(A[1..(n - 1)], x)
  if (A[n] < x) then
    m = max(m, 1 + LIS\_smaller(A[1..(n - 1)], A[n]))
  Output m

LIS(A[1..n]):
  return LIS\_smaller(A[1..n], \infty)

- How many distinct sub-problems will LIS\_smaller(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Approach

\[
\text{LIS}_\text{smaller}(A[1..n], x): \\
\text{if } (n = 0) \text{ then return 0} \\
m = \text{LIS}_\text{smaller}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
\quad m = \max(m, 1 + \text{LIS}_\text{smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\]

\[
\text{LIS}(A[1..n]): \\
\text{return } \text{LIS}_\text{smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \( \text{LIS}_\text{smaller}(A[1..n], \infty) \) generate? \( O(n^2) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(1) \) time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization? \( O(n^2) \)
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x) :
\]

\[
\begin{align*}
\text{if } (n = 0) & \text{ then return } 0 \\
 m &= \text{LIS\_smaller}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) & \text{ then} \\
 m &= \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]) :
\]

\[
\text{return LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x) : \\
\text{if } (n = 0) \text{ then return } 0 \\
m = \text{LIS\_smaller}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n])) \\
\text{Output } m
\]

\[
\text{LIS}(A[1..n]) : \\
\text{return } \text{LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n + 1$)

$LIS(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Base case: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

- $LIS(i, j) = LIS(i - 1, j)$ if $A[i] > A[j]$
- $LIS(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] \leq A[j]$

Output: $LIS(n, n + 1)$
After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n + 1$).

$LIS(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Base case: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

- $LIS(i, j) = LIS(i - 1, j)$ if $A[i] > A[j]$
- $LIS(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] \leq A[j]$

Output: $LIS(n, n + 1)$
Iterative algorithm

**LIS-Iterative**(*A[1..n]*):

\[ A[n + 1] = \infty \]

int \[ LIS[0..n, 1..n + 1] \]

for \((j = 1 \text{ to } n + 1)\) do

\[ LIS[0, j] = 0 \]

for \((i = 1 \text{ to } n)\) do

for \((j = i + 1 \text{ to } n)\)


Else \[ LIS[i, j] = \max\{LIS[i - 1, j], 1 + LIS[i - 1, i]\} \]

Return \[ LIS[n, n + 1] \]

Running time: \(O(n^2)\)

Space: \(O(n^2)\)
How to order bottom up computation?

Base case: \( \text{LIS}(0, j) = 0 \) for \( 1 \leq j \leq n + 1 \)

Recursive relation:

- \( \text{LIS}(i, j) = \text{LIS}(i - 1, j) \) if \( A[i] > A[j] \)
- \( \text{LIS}(i, j) = \max\{\text{LIS}(i - 1, j), 1 + \text{LIS}(i - 1, i)\} \) if \( A[i] \leq A[j] \)
How to order bottom up computation?

Sequence: \( A[1..7] = 6, 3, 5, 2, 7, 8, 1 \)
**Two comments**

**Question:** Can we compute an optimum solution and not just its value?
Yes! See notes.

**Question:** Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
Question: Can we compute an optimum solution and not just its value?
Yes! See notes.

Question: Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Yes! See notes.

**Question:** Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
1. Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.

2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.

3. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.

4. Optimize the resulting algorithm further
Part V

Some experiments with memoization
For the input \( n \), two random strings of length \( n \) were generated, and their distance computed using edit distance. Note, that edit-distance is simple enough to that DP gets very good performance. For more complicated problems, the advantage of DP would probably be much smaller.

The asymptotic running time here is \( \Theta(n^2) \).
The implementation was done in C++, using -O9 in compilation.

DP = Dynamic Programming = iterative implementation using arrays.

Partial memoization = Still uses recursive code, but remembers the results in tables that are managed directly by the code.

Implicit memoization = implemented using the standard unordered_map.
Conclusions

1. If you are in interview setup, you should probably solve the problem using DP. That what you would be expected to do.

2. Otherwise, I would probably implement partial memoization — it still has the simplicity of the recursive solution, while having a decent performance. If I really care about performance I would implement the DP.

3. Using implicit memoization probably makes sense only if running time is not really an issue.