Backtracking and Memoization

Lecture 12
Wednesday, March 4, 2020
Reduction:
Reduce one problem to another

Recursion:
A special case of reduction
1. reduce problem to a smaller instance of itself
2. self-reduction

1. Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.
Recursion in Algorithm Design

1. **Tail Recursion**: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.

2. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems that are solved separately. Conquer step puts together solution for bigger problem. Examples: Closest pair, deterministic median selection, quick sort.

3. **Backtracking**: Refinement of brute force search. Build solution incrementally by invoking recursion to try all possibilities for the decision in each step.

4. **Dynamic Programming**: problem reduced to multiple (typically) *dependent or overlapping* sub-problems. Use **memoization** to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.
Part I

Brute Force Search, Recursion and Backtracking
**Maximum Independent Set in a Graph**

**Definition**

Given undirected graph \( G = (V, E) \) a subset of nodes \( S \subseteq V \) is an independent set (also called a stable set) if for there are no edges between nodes in \( S \). That is, if \( u, v \in S \) then \((u, v) \notin E\).

Some independent sets in graph above: \( \{D\}, \{A, C\}, \{B, E, F\} \)
Maximum Independent Set Problem

Input  Graph  $G = (V, E)$

Goal   Find maximum sized independent set in  $G$
Maximum Weight Independent Set Problem

**Input**  Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

**Goal**  Find maximum weight independent set in $G$
No one knows an *efficient* (polynomial time) algorithm for this problem.

Problem is **NP-Complete** and it is *believed* that there is no polynomial time algorithm.

**Brute-force algorithm:**

Try all subsets of vertices.
Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

\[
\text{MaxIndSet}(G = (V, E)):
\]
\[
\begin{align*}
\text{max} &= 0 \\
\text{for each subset } S \subseteq V \text{ do} \\
&\quad \text{check if } S \text{ is an independent set} \\
&\quad \text{if } S \text{ is an independent set and } w(S) > \text{max} \text{ then} \\
&\quad \quad \text{max} = w(S)
\end{align*}
\]

Output \text{max}
Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

\[
\text{MaxIndSet}(G = (V, E)):
\]

\[
\begin{align*}
\text{max} & = 0 \\
\text{for each subset } S \subseteq V \text{ do} \\
\text{check if } S \text{ is an independent set} & \\
\text{if } S \text{ is an independent set and } w(S) > \text{max} \text{ then} & \\
\text{max} & = w(S)
\end{align*}
\]

Output \( \text{max} \)

Running time: suppose \( G \) has \( n \) vertices and \( m \) edges

1. \( 2^n \) subsets of \( V \)
2. checking each subset \( S \) takes \( O(m) \) time
3. total time is \( O(m2^n) \)
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$. For a vertex $u$ let $N(u)$ be its neighbors.
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$. For a vertex $u$ let $N(u)$ be its neighbors.

Observation

$v_1$: vertex in the graph.

One of the following two cases is true

Case 1 $v_1$ is in some maximum independent set.

Case 2 $v_1$ is in no maximum independent set.

We can try both cases to “reduce” the size of the problem.
A Recursive Algorithm

Let \( V = \{v_1, v_2, \ldots, v_n\} \).
For a vertex \( u \) let \( N(u) \) be its neighbors.

Observation

\( v_1 \): vertex in the graph.

One of the following two cases is true

- Case 1 \( v_1 \) is in some maximum independent set.
- Case 2 \( v_1 \) is in no maximum independent set.

We can try both cases to “reduce” the size of the problem

\[
G_1 = G - v_1 \text{ obtained by removing } v_1 \text{ and incident edges from } G
\]
\[
G_2 = G - v_1 - N(v_1) \text{ obtained by removing } N(v_1) \cup v_1 \text{ from } G
\]

\[
MIS(G) = \max\{MIS(G_1), MIS(G_2) + w(v_1)\}
\]
A Recursive Algorithm

\textbf{RecursiveMIS}(G):
    \textbf{if } G \text{ is empty} \textbf{ then } Output 0
    \textbf{a} = \text{RecursiveMIS}(G - v_1)
    \textbf{b} = w(v_1) + \text{RecursiveMIS}(G - v_1 - N(v_n))
    Output $\max(a, b)$
Running time:

\[ T(n) = T(n-1) + T\left(n - 1 - \text{deg}(v_1)\right) + O(1 + \text{deg}(v_1)) \]

where \( \text{deg}(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.
Recursive Algorithms
..for Maximum Independent Set

Running time:

\[ T(n) = T(n - 1) + T\left( n - 1 - \deg(v_1) \right) + O(1 + \deg(v_1)) \]

where \( \deg(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \deg(v_1) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n - 1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).
Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem)

Simple recursive algorithm computes/explores the whole tree blindly in some order.

Backtrack search is a way to explore the tree intelligently to prune the search space

Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method

Memoization to avoid recomputing same problem

Stop the recursion at a subproblem if it is clear that there is no need to explore further.

Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.
**Sequences**

**Definition**

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

### 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.
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
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
Naïve Enumeration

Assume \(a_1, a_2, \ldots, a_n\) is contained in an array \(A\)

\[
\text{algLISNaive}(A[1..n]):
\]
\[
\begin{align*}
\text{max} & = 0 \\
\text{for each subsequence } B \text{ of } A & \text{ do} \\
& \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then} \\
& \quad \text{max} = |B|
\end{align*}
\]

Output \(\text{max}\)
Naïve Enumeration

Assume \( a_1, a_2, \ldots, a_n \) is contained in an array \( A \)

\[
\text{algLISNaive}(A[1..n]):
\]
\[
\text{max} = 0
\]
\[
\text{for each subsequence } B \text{ of } A \text{ do}
\]
\[
\quad \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then}
\]
\[
\qquad \text{max} = |B|
\]

Output \( \text{max} \)

Running time:

\( O(n^2) \)
Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence $B$ of $A$ do
        if $B$ is increasing and $|B| > max$ then
            max = |B|
    Output max
```

Running time: $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Recursive Approach: Take 1

**LIS**: Longest increasing subsequence

Can we find a recursive algorithm for **LIS**?

\[ \text{LIS}(A[1..n]): \]
Can we find a recursive algorithm for 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
Recursive Approach: Take 1

**LIS**: Longest increasing subsequence

Can we find a recursive algorithm for **LIS**?

**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.
Recursive Approach: Take 1

**LIS:** Longest increasing subsequence

Can we find a recursive algorithm for **LIS**?

**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 \(\text{LIS}_{\text{smaller}}(A[1..n], x)\) which gives the longest increasing subsequence in \(A\) where each number in the sequence is less than \(x\).
Recursive Approach

\texttt{LIS\_smaller(A[1..n], x)}: length of longest increasing subsequence in \texttt{A[1..n]} with all numbers in subsequence less than \texttt{x}

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

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

Sequence: \[ A[1..7] = 6, 3, 5, 2, 7, 8, 1 \]
Part II

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

2. \[ \lim_{n \to \infty} \frac{F(n + 1)}{F(n)} = \phi \]
How many bits?

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

\[
\text{Fib}(n): \\
\quad \text{if} \ (n = 0) \ \\
\quad \quad \text{return} \ 0 \\
\quad \text{else if} \ (n = 1) \ \\
\quad \quad \text{return} \ 1 \\
\quad \text{else} \\
\quad \quad \text{return} \ \text{Fib}(n - 1) + \ \text{Fib}(n - 2)
\]
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).
Recursive Algorithm for Fibonacci Numbers

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

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

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 \( F(n) \).

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

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

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

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
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]
An iterative algorithm for Fibonacci numbers

\begin{verbatim}
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]
\end{verbatim}

What is the running time of the algorithm?
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.
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.
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**.
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.
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Fib(n):

- if \( n = 0 \) return 0
- if \( n = 1 \) return 1
- if \( \text{Fib}(n) \) 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 Memoization

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

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n - 1) + Fib(n - 2)
```
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?
Automatic Memoization

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

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

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

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

$$\text{Fib}(n):$$

if ($n = 0$)
  return 0
if ($n = 1$)
  return 1
if ($n$ is already in $D$)
  return value stored with $n$ in $D$
$$\text{val} \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2)$$
Store $(n, \text{val})$ in $D$
return $\text{val}$

Use hash-table or a map to remember which values were already computed.
Initialize table/array $M$ of size $n$: $M[i] = -1$ for $i = 0, \ldots, n$. 

Resulting code:

```python
Fib(n):
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    if (M[n] != -1) // M[n]: stored value of Fib(n)
        return M[n]
    M[n] ← Fib(n-1) + Fib(n-2)
    return M[n]
```

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:

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

   \texttt{Fib(n)}:
   
   \begin{verbatim}
   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]
   \end{verbatim}

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 Fib...
Automatic Memoization

Recursive version:

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

\[
\text{if } f \text{ already computed for } (x_1, x_2, \ldots, x_d) \text{ then return value already computed}
\]

NEW CODE:

Replaces any "return \(\alpha\)" with "Remember \(f(x_1, x_2, \ldots, x_d) = \alpha\); return \(\alpha\)."
Automatic Memoization

1 Recursive version:

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

CODE

2 Recursive version with memoization:

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

\[
\text{if } f \text{ already computed for } (x_1, x_2, \ldots, x_d) \text{ then} \\
\text{return value already computed} \\
\text{NEW_CODE}
\]
Automatic Memoization

1. Recursive version:

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

2. Recursive version with memoization:

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

\[
\text{if } f \text{ already computed for } (x_1, x_2, \ldots, x_d) \text{ then }
\]

\[
\text{return value already computed}
\]

NEW_CODE

3. NEW_CODE:

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

1. Explicit memoization (iterative algorithm) preferred:
   - analyze problem ahead of time

2. Implicit (automatic) memoization:
   - problem structure or algorithm is not well understood.
   - Need to pay overhead of data-structure.
   - 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.
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.
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.
How many distinct calls?

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

if \( t = 0 \) then return 0
if \( b = t \) or \( b = 0 \) then return 1
return \( \text{\texttt{binom}}(t - 1, b - 1) + \text{\texttt{binom}}(t - 1, b) \).

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

```plaintext
binom(t, b)  // computes \( \binom{t}{b} \)
if \( t = 0 \) then return 0
if \( b = t \) or \( b = 0 \) then return 1
return binom(t - 1, b - 1) + binom(t - 1, b).
```

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?

\[ D: \text{ Initially an empty dictionary.} \]
\[
\text{binomM}(t, b) \quad \text{// computes } \binom{t}{b}
\]
\[
\text{if } b = t \text{ then return } 1
\]
\[
\text{if } b = 0 \text{ then return } 0
\]
\[
\text{if } D[t, b] \text{ is defined then return } D[t, b]
\]
\[
D[t, b] \leftarrow \text{binomM}(t - 1, b - 1) + \text{binomM}(t - 1, b).
\]
\[
\text{return } D[t, b]
\]

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 numbers sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
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?
Saving space. Do we need an array of $n$ numbers? Not really.

```python
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
```