05 - Recursive Algorithm

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 28 Feb - 4 March 2022
Table of contents

- The principal of recursive algorithm
- Some examples of recursive algorithms
  1. Computing factorial
  2. Proving correctness of FACTORIAL by induction
  3. Finding Maximum Element of an Array
  4. Computing sum of elements in an array
  5. Computing max recursively
- Tower of Hanoi Problem
- Binary search algorithm
- Recursive powering
- Redundancy in recursive algorithm
- Fibonacci sequence
- Advantages and drawbacks of recursive algorithm
What is recursion or recursive algorithm?
1. The principal of recursive algorithm

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

Characteristics of recursive algorithm:

1. It calls itself recursively
2. It has a base case
3. It must change its state and move towards the base case

A base case is the condition that allows the algorithm to stop recursing: a base case is typically a problem that is small enough to solve directly.

A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way.
**Iteration**: A function repeats a defined process until a condition fails. This is usually done through a loop, such as a for or while loop with a counter and comparative statement making up the condition that will fail. An infinite loop for iteration occurs when the condition never fails.

**Recursion**: Instead of executing a specific process within the function, the function calls itself repeatedly until a certain condition is met (this condition being the base case). The base case is explicitly stated to return a specific value when a certain condition is met. An infinite recursive loop occurs when the function does not reduce its input in a way that will converge on the base case.
Simple examples of recursive algorithms
2.1 - Computing factorial (1): Problem statement

\[ n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1 \]

The formula can be expressed recursively:

\[ n! = \begin{cases} 
  n \times (n - 1)!, & \text{if } n > 1 \\
  1, & \text{if } n = 1 
\end{cases} \]
Algorithm 1 Factorial of a number

1: procedure FACTORIAL(n)
2:       if \( n = 1 \) then
3:               return 1
4:       else
5:               temp = FACTORIAL(n - 1)
6:               return \( n \times \text{temp} \)
7:       end if
8: end procedure

- What is the base case?
Algorithm 2 Factorial of a number

1: procedure FACTORIAL(n)
2: if $n = 1$ then
3: return 1
4: else
5: temp = FACTORIAL($n - 1$)
6: return $n \times$ temp
7: end if
8: end procedure

- What is the base case? $n = 1$
- What is the change of states?
Algorithm 3 Factorial of a number

1: **procedure** FACTORIAL($n$)
2: 
3: if $n = 1$ then
4:  
5:       return 1
6: else
7:  
8:      temp = FACTORIAL($n - 1$)
9:  
10:     return $n \times$ temp
11: end if
12: end procedure

- What is the base case? $n = 1$
- What is the change of states? $n$ decreases
- What is the complexity? $O(n)$
Algorithm 4 Factorial of a number

1: procedure FACTORIAL($n$)
2:    if $n = 1$ then
3:        return 1
4:    else
5:        temp = FACTORIAL($n - 1$)
6:        return $n \times$ temp
7:    end if
8: end procedure

- What is the base case? $n = 1$
- What is the change of states? $n$ decreases
- What is the complexity? $\mathcal{O}(n)$
2.1 - Computing factorial (3): Diagram

Figure: Illustration of recursive algorithm FACTORIAL with $n = 3$
2.1 - Computing factorial (4): Proving correctness by induction

- **Induction base:** from line 1, we see that the function works correctly for \( n = 1 \).

- **Hypothesis:** suppose the function works correctly when it is called with \( n = m \), for some \( m \geq 1 \).

- **Induction step:** We prove that it also works when it is called with \( n = m + 1 \). By the hypothesis, we know the recursive call works correctly for \( n = m \) and computes \( m! \). Subsequently, it is multiplied by \( n = m + 1 \), thus computes \((m + 1)!\). And this is the value correctly returned by the program.
To compute the max of $n$ elements for $n > 1$ recursively:

- Compute the max of $n - 1$ elements
- Compare with the last element to find the entire max

Algorithm 5

Finding maximum of an array

1: procedure Max (A[0..n−1], int n)
2: if n = 1 then return A[0]
3: else
4: $T = \text{Max}(A, n - 1)$
5: if $T < A[n - 1]$ then
6: return $A[n - 1]$
7: else
8: return $T$
9: end if
10: end if
11: end procedure
To compute the max of $n$ elements for $n > 1$ recursively:
- Compute the max of $n - 1$ elements
- Compare with the last element to find the entire max

**Algorithm 6** Finding maximum of an array

1. procedure $\text{MAX}(A[0..n-1], \text{int } n)$
2. 
   if $n = 1$ then return $A[0]$ 
3. 
   else
4. 
   
   \[ T = \text{MAX}(A, n - 1) \]
5. 
   if $T < A[n - 1]$ then 
6. 
   
   return $A[n - 1]$ 
7. 
   else
8. 
   
   return $T$
9. 
   end if
10. end if
11. end procedure
Task: Compute the following

- Complexity?
- Correctness?
Problem: Given an array of $n$ elements $A[0..n-1]$. We want to compute the sum: $S = \sum_{i=0}^{n-1} A[i]$

**Algorithm 7 Sum of an array**

1. procedure $\text{SUM}(A[0..n], \text{int } n)$
2. \hspace{1em} if $n = 1$ then return $A[0]$
3. \hspace{1em} else
4. \hspace{2em} $S = \text{SUM}(A, n - 1)$
5. \hspace{2em} $S = S + A[n - 1]$
6. \hspace{2em} if $T < A[n - 1]$ then
7. \hspace{3em} return $S$
8. \hspace{2em} end if
9. \hspace{1em} end if
10. end procedure
Task: Compute the following

- Complexity?
- Correctness?
Problem: Given an array $A$ of $n$ elements, we aim to find an element of maximum value of the array.

Approach:

1. Divide the array into two halves sub-array, namely Left sub-array and Right sub-array.
2. Find the max of each sub-array.
3. Compare the max value of the Left array and the Right array.
4. Return the maximum of the two values.
2.6. Recursive **MAX**, 2nd approach (2)

**Algorithm 8** Finding max of an array

1. **procedure** `FINDMAX(A[i..j], n)` ▷ `i, j` are respectively the index of start, end of `A`
2. `if n = 1 then return A[S]`
3. `end if`
4. `m = ⌊i+j / 2⌋`
5. `T_1 = FINDMAX(A[i..m], ⌊n / 2⌋)` ▷ Recursive call the left sub-array
6. `T_2 = FINDMAX(A[(m+1)..j], n − ⌊n / 2⌋)` ▷ Rec. call right sub-array
7. `if T_1 ≥ T_2 then return T_1` ▷ Compare the two max elements
8. `else return T_2`
9. `end if`
10. **end procedure**

**Remark.** ⌊x⌋ means the largest integer that is ≤ x;
example: ⌊3.5⌋ = 3
2.6. Recursive Max, 2nd approach (3)

Complexity analysis: Special case when \( n = 2^k \)

Let \( f(n) \): the number of key-comparisons to find the max of an \( n \)-array, with \( n = 2^k \) for some positive integer \( k \). Hence:

\[
f(n) = \begin{cases} 
0, & n = 1 \\
1 + 2f(n/2), & n \geq 2
\end{cases}
\]

By repeated substitution:

\[
f(n) = 1 + 2f(n/2) \\
= 1 + 2[1 + 2f(n/4)] = 1 + 2 + 2f(n/4) \\
= 1 + 2 + 4 + 8f(n/4) \\
\vdots \\
= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k f(n/2^k) \\
= 1 + 2 + 4 + \cdots + 2^{k-1} \\
= 2^k - 1/(2 - 1) = 2^k - 1 \\
= n - 1
\]
2.6. Recursive Max, 2nd approach (4)

\( f(n) \): the number of key-comparisons to find the max of an \( n \)-array, with \( n = 2^k \) for some \( k \in \mathbb{Z}^+ \).

**Complexity analysis:** For general \( n \)

\[
f(n) = \begin{cases} 
0, & n = 1 \\
f(\lfloor n/2 \rfloor) + f(n - \lfloor n/2 \rfloor) + 1, & n \geq 2 
\end{cases}
\]

Prove that:

By induction, we obtain \( f(n) = n - 1 \). How?
Tower of Hanoi

**Problem:** there are three towers $A$, $B$, and $C$. Initially, there are $n$ disks of varying sizes stacked on tower $A$, ordered by their size, with the largest disk on the bottom and the smallest one on the top. The objective is to move all discs to the 2nd tower by keeping their order.

- Only one disk may be moved at a time in a restricted manner, from the top of one tower to the top of another tower.
- A larger disk must never be placed on top of a smaller disk.

Check [https://www.mathsisfun.com/games/towerofhanoi.html](https://www.mathsisfun.com/games/towerofhanoi.html) for an illustration of the problem
Figure: Initial configuration with 4 disks on Tower A
3. Tower of Hanoi problem (2): Illustration

![Diagram showing the Tower of Hanoi problem after recursively moving the top 3 disks from Tower A to Tower C.]

**Figure:** After recursively moving the top 3 disks from Tower A to Tower C
Figure: After moving the bottom disk from Tower A to Tower B
Figure: After recursively moving back 3 disks from Tower C to Tower B.
Task: Write the pseudocode of the Tower of Hanoi problem!
Task: Write the pseudocode of the Tower of Hanoi problem!

Algorithm 10 Tower of Hanoi

1: procedure TOWERS(A, B, C, n)
2: if n = 1 then
3:   MOVEONE(A, B)
4:   return
5: end if
6: TOWERS(A, C, B, n − 1)
7: MOVEONE(A, B)
8: TOWERS(C, B, A, n − 1)
9: end procedure

- TOWERS(A, B, C, n): move n disks from A to B, using A, B, C
- MOVEONE(A, B): move one disk from A to B
Correctness: by induction

- **Base case:** For $n = 1$, a single move is made from A to B. So the algorithm works correctly for $n = 1$.
- For any $n \geq 2$, suppose the algorithm works correctly for $n - 1$.
  - Then, by the hypothesis, the recursive call of line 6 works correctly and moves the top $n - 1$ disks to C, leaving the bottom disk on tower A.
  - The next step, line 7, moves the bottom disk to B.
  - Finally, the recursive call of line 8 works correctly by the hypothesis and moves back $n - 1$ disks from C to B.
- Thus, the entire algorithm works correctly for $n$. 
Recurrence equation to analyze time complexity

Let $f(n)$: the number of single moves to solve the problem for $n$ disks

Hence we have the following relation:

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + 2f(n - 1), & n \geq 2 \end{cases}$$

Remark. The above formula is known as recursive formula; read this page or watch this video to get an overview.

To obtain the explicit formula of $f(n)$, we have to solve the recurrence equation for $f(n)$. 
Method 1: Repeated substitution

\[ f(n) = 1 + 2 \cdot f(n-1) \]
\[ = 1 + 2 + 4 \cdot f(n-2) \]
\[ = 1 + 2 + 4 + 8 \cdot f(n-3) \]
\[ = \ldots \]
\[ = 1 + 2 + 2^2 + \ldots + 2^{n-1} \cdot f(1) \]

Substituting the base case \( f(1) = 1 \) and by the geometric sum formula (click here to check the formula), we obtain:

\[ f(n) = \frac{2^n - 1}{2 - 1} = 2^n - 1 \]
3. Tower of Hanoi problem (7): Time complexity analysis

Method 2: Guess the solution and prove by induction
Suppose our guess is “$f(n)$ is exponential”

Guess: $f(n) = a \cdot 2^n + b$

Inductive proof:
- **Induction base:** $n = 1$
  - $f(1) = 1$ (from the recurrence)
  - $f(1) = 2a + b$ (from the solution form)
  So we have $2a + b = 1$
- **Induction:** Suppose that the solution is correct for some $n \geq 1$:
  \[
  f(n) = a \cdot 2^n + b
  \]
  Then the solution must hold for $n + 1$, that is:
  \[
  f(n + 1) = a \cdot 2^{n+1} + b
  \]
3. Tower of Hanoi problem (8): Time complexity analysis

- From the recurrence relation, we obtain:

\[ f(n + 1) = 2f(n) + 1 \]
\[ = 2(a \cdot 2^n + b) + 1 \]
\[ = a \cdot 2^{n+1} + (2b + 1) \]

- From the two equations, we obtain:

\[
a \cdot 2^{n+1} + b = a \cdot 2^{n+1} + (2b + 1) \iff 2b + 1 = b \iff b = -1
\]

Hence, \( 2a + b = 1 \iff a = 1 \). So, \( b = -1 \).

- Hence, \( f(n) = a \cdot 2^n + b = 2^n - 1 \).
Binary Search
4. Binary search algorithm (1): Principal

**Problem:** Given a *sorted* array \(A[0..n−1]\) and a search key \(KEY\). The algorithm does the following:

- If \(KEY = A[m]\), then return \(m\)
- If \(KEY < A[m]\), then recursively search the left half of the array
- If \(KEY > A[m]\), then recursively search the right half of the array

In each step, the size of the search *is reduced by half*.
4. Binary search algorithm (2): Diagram

The Idea of Binary Search

source: https://www.enjoyalgorithms.com/blog/binary-search-algorithm
### Binary search algorithm (3): Pseudocode

#### Algorithm 11 Binary search algorithm

1: **procedure** `BINSEARCH(A, i, j, KEY)`

2:     if `i > j` then

3:         return $-1$  ▶ *Base case is reached but KEY is not found*

4:     end if

5:     $m = \left\lfloor \frac{i+j}{2} \right\rfloor$  ▶ *Choose the pivot*

6:     if `KEY = A[m]` then

7:         return $m$  ▶ *KEY is found at index m*

8:     else

9:         if `KEY < A[m]` then  ▶ *The KEY is located on the Left sub-array*

10:            return `BINSEARCH(A, i, m - 1, KEY)`  ▶ *Rec-call left part*

11:        else

12:            return `BINSEARCH(A, m + 1, j, KEY)`  ▶ *Rec-call right part*

13:        end if

14:     end if

15: end procedure
Let $f(n)$ be the number of comparisons.

**Complexity analysis**: special case when $n = 2^k$

\[
\begin{align*}
    f(n) &= \begin{cases} 
    1, & n = 1 \\
    1 + f(n/2), & n \geq 2 
    \end{cases} \\
\end{align*}
\]

By repeated substitution:

\[
\begin{align*}
    f(n) &= 1 + f(n/2) \\
    &= 1 + 1 + f(n/4) \\
    &= 1 + 1 + 1 + f(n/8) \\
    &\vdots \\
    &= k + f(n/2^k) \\
    &= k + f(1) \\
    &= k + 1 \\
    &= \log n + 1
\end{align*}
\]
Complexity analysis: general case $n$

$$f(n) = \begin{cases} 1, & n = 1 \\ 1 + f(\lfloor \frac{n}{2} \rfloor), & n \geq 2 \end{cases}$$

By induction, we obtain $f(n) = \lfloor \log n \rfloor + 1$

Exercise: show it!
4. Binary search algorithm (6): Inductive proof for TC analysis

- **Induction base:** \( n = 1 \):
  From the recurrence, \( f(1) = 1 \), and the claimed solution \( f(1) = \lfloor \log 1 \rfloor + 1 = 1 \). Correct.

- **Inductive proof:** Suppose that the formula is correct for all smaller values.
  \[
  f(m) = \lfloor \log m \rfloor, \quad \forall m < n
  \]
  Every integer \( n \) can be expressed as (for some integer \( k \)):
  \[
  2^{k-1} \leq \lfloor n/2 \rfloor < 2^k
  \]
  So, \( \lfloor \log \lfloor n/2 \rfloor \rfloor = k - 1 \).

  By the recursive function:
  \[
  f(\lfloor n/2 \rfloor) = \lfloor \log \lfloor n/2 \rfloor \rfloor + 1 = (k - 1) + 1 = k = \lfloor \log n \rfloor
  \]
  Then:
  \[
  f(n) = f(\lfloor n/2 \rfloor) + 1 = k + 1 = \lfloor \log n \rfloor + 1
  \]
A more advanced example: Recursive powering
Problem: Given $X$ and an integer $n$. We want to compute $X^n$.

Algorithm 12 Recursive powering (brute force)

1: procedure Power1($X$, $n$)
2: \hspace{1em} $T = X$
3: \hspace{1em} for $i = 2$ to $n$ do
4: \hspace{2em} $T = T \times X$
5: \hspace{1em} end for
6: end procedure

Complexity: $O(n)$. Why?
5. Recursive powering (2): Approach

Idea: $X^{16} = (((((X^2)^2)^2)^2)^2)$

Given $n = 2^k$, we can do repeated squaring.

**Algorithm 13** Recursive powering (*special case*)

1: procedure $\text{POWER2}(X, n = 2^k)$
2: $\quad T = X$
3: $\quad$ for $i = 2$ to $k$ do
4: $\quad$ $T = T \times T$
5: $\quad$ end for
6: end procedure

**Complexity:** $\mathcal{O}(\log n)$. Why?
5. Recursive powering (3): Approach

Generalize the case for $n$: Computing $X^n$ for $n \in \mathbb{Z}^+$

- Compute $X^2 = X \ast X$
- Compute $X^3 = X^2 \ast X$
- Compute $X^6 = X^3 \ast X^3$
- Compute $X^{12} = X^6 \ast X^6$
- Compute $X^{13} = X^{12} \ast X$
5. Recursive powering (4): Approach

**Basic idea:** Divide \( n \) by 2, \( n = n/2 + n/2 \). So

\[
X^n = X^{(n/2+n/2)} = X^{n/2} \cdot X^{n/2}
\]

The problem is \( n/2 \) is not always an integer. So we have to apply a little modification:

- For \( n = 0 \), then \( X^n = 1 \)
- For \( n > 0 \), then:
  - If \( n \) is *even*, then \( X^n = X^{n/2} \cdot X^{n/2} \)
  - If \( n \) is *odd*, then \( X^n = X^{\lfloor n/2 \rfloor} \cdot X^{\lfloor n/2 \rfloor} \cdot X \)
Algorithm 14 Recursive powering

1: procedure \textsc{Power3}(X, n)
2: \hspace{1em} if $n = 1$ then
3: \hspace{2em} return $X$
4: \hspace{1em} end if
5: \hspace{1em} $T = \textsc{Power}(X, \lfloor \frac{n}{2} \rfloor)$
6: \hspace{1em} $T = T \times T$
7: \hspace{1em} if $n \mod 2 = 1$ then
8: \hspace{2em} $T = T \times X$
9: \hspace{2em} return $T$
10: \hspace{1em} end if
11: end procedure

Complexity: ?
Example: Computing $3^{16}$

\[ 3^{16} = 3^8 \cdot 3^8 = (3^8)^2 \]
\[ = ((3^4)^2)^2 \]
\[ = (((3^2)^2)^2)^2 \]
\[ = ((((3^1)^2)^2)^2)^2 \]
\[ = ((((3^0) \cdot 3)^2)^2)^2)^2 \]
\[ = (((1 \cdot 3)^2)^2)^2)^2 \]
\[ = (((3)^2)^2)^2 \]
\[ = (((9)^2)^2)^2 \]
\[ = ((81)^2)^2 \]
\[ = (6561)^2 \]
\[ = 43,046,721 \]
Let \( n = 2m + r \), where \( r \in \{0, 1\} \).

- The algorithm first makes a recursive call to compute \( T = X^m \).
- Then it squares \( T \) to get \( T = X^{2m} \). If \( r = 0 \), this is returned.
- Otherwise, when \( r = 1 \), the algorithm multiplies \( T \) by \( X \), to result in \( T = X^{2m+1} \).
Let $f(n)$: the worst-case number of multiplication steps to compute $X^n$.

- The recursive call takes $f(\lfloor \frac{n}{2} \rfloor)$ multiplications.
- Then it is followed by one more multiplication. In the worst case, when $n$ is odd, one additional multiplication is performed.

Hence,

$$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2, \text{ } n \text{ odd} \\ f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2, \text{ } n \text{ even} \end{cases}$$

Show that $f(n) = 2\lfloor \log n \rfloor$.
5. Recursive powering (8): Time complexity analysis

\[ f(n) = \begin{cases} 
0, & \text{if } n = 1 \\
 f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2, \text{ } n \text{ odd} \\
 f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2, \text{ } n \text{ even} 
\end{cases} \]

The last two cases have small difference. So we can approximate the function above with the following function to simplify the computation:

\[ f(n) = \begin{cases} 
0, & \text{if } n = 1 \\
 f(\lfloor \frac{n}{2} \rfloor) + 2, & \text{if } n \geq 2 
\end{cases} \]
Show that \( f(n) = 2\lfloor \log n \rfloor \).

- **Induction base** \((n = 1)\): From the recurrence, \( f(1) = 0 \), and from the formula, \( f(1) = 2\lfloor \log 1 \rfloor = 0 \). Correct.

- **Inductive proof**: Suppose that the formula is correct for all smaller values.

  \[
  f(m) = 2\lfloor \log m \rfloor, \quad \forall m < n
  \]

  Every integer \( n \) can be expressed as (for some integer \( k \)):

  \[
  2^k \leq n < 2^{k+1}
  \]

  So, \( \lfloor \log n \rfloor = k \), and \( \lfloor \frac{\log n}{2} \rfloor = k - 1 \). By the recursive function:

  \[
  f(n) = f(\lfloor \frac{n}{2} \rfloor) + 2 = 2(k - 1) + 2 = 2k = 2\lfloor \log n \rfloor
  \]

  **Remark.** This gives a better complexity than the brute force approach \((\mathcal{O}(n))\).
Redundancy in recursive algorithm
Algorithm 14 Power by multiplications

1: procedure POWER3(X, n)
2:     if n = 1 then
3:         return X
4:     end if
5:     T = POWER(X, ⌊n/2⌋)
6:     T = T * T
7:     if n mod 2 = 1 then
8:         T = T * X
9:     return T
10: end if
11: end procedure

Is it necessary to store POWER(X, ⌊n/2⌋) in some variable T?
Algorithm 15 Recursive powering

1: procedure POWER4(X, n)
2: if n = 1 then
3: return X
4: end if
5: return POWER(X, ⌊n/2⌋) * POWER(X, ⌊n/2⌋)
6: end procedure

Is the algorithm correct? What is the complexity?
The algorithm is correct.

The number of recursive calls:

\[ f(n) = \begin{cases} 
0, & \text{if } n = 1 \\
 f(\lfloor \frac{n}{2} \rfloor) + f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2 
\end{cases} \]

By induction, we can prove that \( f(n) = n - 1 \) (asymptotically worse than the previous algorithm).

What can you conclude?
The algorithm is correct.

The number of recursive calls:

\[
f(n) = \begin{cases} 
0, & \text{if } n = 1 \\
 f(\lfloor \frac{n}{2} \rfloor) + f(\lfloor \frac{n}{2} \rfloor) + 1, & \text{if } n \geq 2 
\end{cases}
\]

By induction, we can prove that \( f(n) = n - 1 \) (asymptotically worse than the previous algorithm).

What can you conclude?

\textsc{Power4} is also not efficient, because we make two recursive calls for the same function \( f(\lfloor \frac{n}{2} \rfloor) \).
The Fibonacci sequence is defined as follows.

\[ F(n) = \begin{cases} 
1, & n = 1 \\
1, & n = 2 \\
F_{n-1} + F_{n-2}, & n \geq 3 
\end{cases} \]

Fibonacci sequence: 1, 2, 3, 5, 8, 13, 21, …
The Fibonacci sequence is defined as follows.

\[
F(n) = \begin{cases} 
1, & n = 1 \\
1, & n = 2 \\
F_{n-1} + F_{n-2}, & n \geq 3 
\end{cases}
\]

Fibonacci sequence: 1, 2, 3, 5, 8, 13, 21, …

Build an algorithm to compute the Fibonacci sequence!

- With naive algorithm (brute force), we can reach complexity \( O(n) \). How?
Redundancy (4): Fibonacci sequence

The Fibonacci sequence is defined as follows.

\[ F(n) = \begin{cases} 
1, & n = 1 \\
1, & n = 2 \\
F_{n-1} + F_{n-2}, & n \geq 3 
\end{cases} \]

Fibonacci sequence: 1, 2, 3, 5, 8, 13, 21, …

Build an algorithm to compute the Fibonacci sequence!

- With naive algorithm (brute force), we can reach complexity \( \mathcal{O}(n) \). How?

  By looping (iterative method); we add the number one by one.
Redundancy (4): Fibonacci sequence

The Fibonacci sequence is defined as follows.

\[
F(n) = \begin{cases} 
1, & n = 1 \\
1, & n = 2 \\
F_{n-1} + F_{n-2}, & n \geq 3 
\end{cases}
\]

Fibonacci sequence: 1, 2, 3, 5, 8, 13, 21, …

Build an algorithm to compute the Fibonacci sequence!

- With naive algorithm (brute force), we can reach complexity \( O(n) \). How?

  By looping (iterative method); we add the number one by one.

- Create a recursive algorithm!
Algorithm 16 Fibonacci sequence

1: procedure \texttt{FIB}(n)
2: \hspace{1em} if \( n \leq 2 \) then return 1
3: \hspace{1em} end if
4: \hspace{1em} return \((\texttt{FIB}(n - 1) + \texttt{FIB}(n - 2))\)
5: end procedure

This program makes recursive calls with a great deal of overlapping computations, causing a huge inefficiency.

Complexity: \( T(n) = \begin{cases} 0, & n = 1 \\ 0, & n = 2 \\ T(n - 1) + T(n - 2) + 1, & n \geq 3 \end{cases} \)

Prove that: the explicit function: \( T(n) \geq \left(1.618\right)^{n-2} \)
Algorithm 17 Fibonacci sequence

1: procedure \texttt{FIB}(n)
2: \hspace{1em} if $n \leq 2$ then return 1
3: \hspace{1em} end if
4: \hspace{1em} return \((\texttt{FIB}(n - 1) + \texttt{FIB}(n - 2))\)
5: end procedure

This program makes recursive calls with a great deal of overlapping computations, causing a huge inefficiency.
Algorithm 18 Fibonacci sequence

1: procedure $\text{FIB}(n)$
2:     if $n \leq 2$ then return 1
3:     end if
4:     return ($\text{FIB}(n - 1) + \text{FIB}(n - 2)$)
5: end procedure

This program makes recursive calls with a great deal of overlapping computations, causing a huge inefficiency.

Complexity:

$$T(n) = \begin{cases} 
0, & n = 1 \\
0, & n = 2 \\
T(n - 1) + T(n - 2) + 1, & n \geq 3 
\end{cases}$$

Prove that: the explicit function: $T(n) \geq (1.618)^{n-2}$. 
Advantages

- Recursion adds clarity and reduces the time needed to write and debug code (since it reduces the length of code).
- To solve such problems which are naturally recursive such as the Tower of Hanoi.
- Recursion can reduce time complexity (sometimes counter-intuitive).
- Reduce unnecessary calling of function.
- Extremely useful when applying the same solution.
Drawbacks

- Recursive functions are generally slower than non-recursive function.
- It may require a lot of memory space to hold intermediate results on the system stacks.
- Hard to analyze or understand the code.
- It is not more efficient in terms of space and time complexity (can be slow).
- The computer may run out of memory if the recursive calls are not properly checked.
Sum up...
What have we learned today?

1. Reviewing brute force approach
2. Understanding the principal of recursive approach
3. Some examples of recursive algorithms
4. Recurrence equation to analyze time complexity
5. Redundancy in recursion: be careful when writing the pseudocode
6. Binary search algorithm