Chapter 4 Algorithm

Su

北京邮电大学

2023 年 *10* 月 *10* 日

Contents

4.1 Introduction

4.1 Introduction

What is an algorithm?

4.1 Introduction

What is an algorithm?

- ***An algorithm is a* step-by-step method *of* solving some problem.**

4.1 Introduction

What is an algorithm?

- *An algorithm is a* **step-by-step method** *of* **solving some problem.**

This is a rather vague definition. We will get to know a more precise and mathematically useful definition in Chapter 12.

Algorithms typically have the following characteristics:

Algorithms typically have the following characteristics:

- **Input(输入)** *The algorithm receives input.*
- **Output(输出)** *The algorithm produces output.*
- **Precision(准确性)** *The steps are precisely stated.*
- **Determinism(决定性)** *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness(有限性)** *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness(正确性)** *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality(一般性)** *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input**(输入) *The algorithm receives input.*
- **Output**(输出) *The algorithm produces output.*
- **Precision**(准确性) *The steps are precisely stated.*
- **Determinism**(决定性) *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness**(有限性) *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness**(正确性) *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality**(一般性) *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input(输入)** *The algorithm receives input.*
- **Output(输出)** *The algorithm produces output.*
- **Precision(准确性)** *The steps are precisely stated.*
- **Determinism(决定性)** *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness(有限性)** *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness(正确性)** *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality(一般性)** *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input**(输入) *The algorithm receives input.*
- **Output**(输出) *The algorithm produces output.*
- **Precision**(准确性) *The steps are precisely stated.*
- **Determinism**(决定性) *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness**(有限性) *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness**(正确性) *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality**(一般性) *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input(输入)** *The algorithm receives input.*
- **Output(输出)** *The algorithm produces output.*
- **Precision(准确性)** *The steps are precisely stated.*
- **Determinism(决定性)** *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness(有限性)** *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness(正确性)** *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality(一般性)** *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input(输入)** *The algorithm receives input.*
- **Output(输出)** *The algorithm produces output.*
- **Precision(准确性)** *The steps are precisely stated.*
- **Determinism(决定性)** *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness(有限性)** *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness(正确性)** *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality(一般性)** *The algorithm applies to a set of inputs.*

Algorithms typically have the following characteristics:

- **Input**(输入) *The algorithm receives input.*
- **Output**(输出) *The algorithm produces output.*
- **Precision**(准确性) *The steps are precisely stated.*
- **Determinism**(决定性) *The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
- **Finiteness**(有限性) *The algorithm terminates; that is, it stops after finitely many instructions have been executed.*
- **Correctness**(正确性) *The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.*
- **Generality**(一般性) *The algorithm applies to a set of inputs.*

Example: *a simple algorithm*

This algorithm finds the largest of the numbers a, b, and c.

Example: *a simple algorithm*

This algorithm finds the largest of the numbers a, b, and c.

Input: a, b, c
Output: large (the largest of a, b, and c)

Example: *a simple algorithm*

This algorithm finds the largest of the numbers a, b, and c.

Input: a, b, c

Output: large (the largest of a, b, and c)

```
max3(a, b, c) {
large = a
if (b > large)
large = b
if (c > large)
large = c
return large
}
```

Example: *a simple algorithm*

This algorithm finds the largest of the numbers a, b, and c.

Input	*The algorithm receives input.*
Output	*The algorithm produces output.*
Precision	*The steps are precisely stated.*
Determinism	*The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.*
Finiteness	*The algorithm terminates in finite steps.*
Correctness	*The algorithm correctly solves the problem.*
Generality	*The algorithm applies to a set of inputs.*

Input: a, b, c
Output: large (the largest of a, b, and c)

```
1. max3(a, b, c) {
2. large = a
3. if (b > large)
4. large = b
5. if (c > large)
6. large = c
7. return large
8. }
```

Example: Goldbach's conjecture *states that every even number greater than 2 is the sum of two prime numbers.*

Example: Goldbach's conjecture *states that every even number greater than 2 is the sum of two prime numbers.*

Proposed algorithm: *checks whether Goldbach's conjecture is true*

4.1 Introduction

Example: Goldbach's conjecture *states that every even number greater than 2 is the sum of two prime numbers.*

Proposed algorithm: *checks whether Goldbach's conjecture is true*

1. Let n = 4.

2. If n is not the sum of two primes,
output "no" and stop.

3. Else increase n by 2
and continue with step 2.

4. Output "yes" and stop.

4.1 Introduction

Example: Goldbach's conjecture *states that every even number greater than 2 is the sum of two prime numbers.*

Proposed algorithm: *checks whether Goldbach's conjecture is true*

1. Let n = 4.
2. If n is not the sum of two primes, output "no" and stop.
3. Else increase n by 2 and continue with step 2.
4. Output "yes" and stop.

- *Which properties of an algorithm—input, output, precision, determinism, finiteness, correctness, generality—does this proposed algorithm have?*

4.1 Introduction

Example: Goldbach's conjecture *states that every even number greater than 2 is the sum of two prime numbers.*

Proposed algorithm: *checks whether Goldbach's conjecture is true*

1. Let n = 4.
2. If n is not the sum of two primes, output "no" and stop.
3. Else increase n by 2 and continue with step 2.
4. Output "yes" and stop.

- *Which properties of an algorithm—input, output, precision, determinism, finiteness, correctness, generality—does this proposed algorithm have?*
- *Do any of them depend on the truth of Goldbach's conjecture?*

4.1 Introduction

- *When the algorithm executes for some specific values. Such a simulation is called a* **trace**.

4.1 Introduction

- *When the algorithm executes for some specific values. Such a simulation is called a* **trace**.
- *In computer science,* **pseudocode** *is a plain language description of the steps in an algorithm or another system.*

An algorithm (written in pseudocodes) always consist of

- *a title,*
- *a brief description of the algorithm,*
- *the input to and output from the algorithm,*
- *the functions containing the instructions of the algorithm.*

4.2 Examples of Algorithm

4.2 Examples of Algorithm

In this section, we give examples of several useful algorithms.

Searching

Searching

- *A large amount of computer time is devoted to searching.*

Searching

- *A large amount of computer time is devoted to searching.*
- *Looking for specified text in a document when running a word processor is an example of a searching problem.*

Searching

- *A large amount of computer time is devoted to searching.*
- *Looking for specified text in a document when running a word processor is an example of a searching problem.*
- *We discuss an algorithm to solve the text-searching problem.*

Example 1. *Text Search*

Example 1. *Text Search*

- *This algorithm searches for an occurrence of the pattern p in text t.*

Example 1. *Text Search*

- *This algorithm searches for an occurrence of the pattern p in text t.*
- *It returns the smallest index i such that p occurs in t starting at index i.*

Example 1. *Text Search*

- *This algorithm searches for an occurrence of the pattern p in text t.*
- *It returns the smallest index i such that p occurs in t starting at index i.*
- *If p does not occur in t, it returns* 0.

Example 1. *Text Search*

Example 1. *Text Search*

- *Suppose that we are given text t (e.g. of length n).*

Example 1. *Text Search*

- *Suppose that we are given text t (e.g. of length n).*
- *We want to find the first occurrence of pattern p (of length m) in t or determine that p does not occur in t.*

Example 1. *Text Search*

- *Suppose that we are given text t (e.g. of length n).*
- *We want to find the first occurrence of pattern p (of length m) in t or determine that p does not occur in t.*
- **Idea**: *compare $t_i \ldots t_{i+m-1}$ and $p_1 \ldots p_m$, if equal then return i, if not, then $i = i + 1$.*

Example 1. *Text Search (pseudo code)*

Example 1. *Text Search (pseudo code)*

Input: p *(indexed from* 1 *to* m*)*, m, t *(indexed from* 1 *to* n*)*, n;

Output: i *or* 0

Example 1. *Text Search (pseudo code)*

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

Example 1. *Text Search (pseudo code)*

Input: p *(indexed from* 1 *to* m*)*, m, t *(indexed from* 1 *to* n*)*, n; **Output**: i *or* 0

$$\text{text search}(p, m, t, n)\ \{$$

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

$$\text{text search}(p, m, t, n)\ \{$$

$$\text{for}\ (i = 1;\ i \leq n - m + 1;\ i = i + 1)\{$$

$$j = 1$$

$$\text{while}\ (t_{i+j-1} == p_j)\ \{$$

$$j = j + 1$$

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

text search(p, m, t, n) {

for $(i = 1;\ i \le n - m + 1;\ i = i + 1)\{$

$j = 1$

while $(t_{i+j-1} == p_j)$ {

$j = j + 1$

if $(j > m)$

return i } }

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

```
text search(p, m, t, n) {
for (i = 1; i ≤ n − m + 1; i = i + 1){
j = 1
while (t_{i+j−1} == p_j) {
j = j + 1
if (j > m)
return i } }
return 0 }
```

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

The following figures show a trace of the text search algorithm where we are searching for the pattern "001" in the text "010001".

$j = 1$
$\downarrow$
```
  001
  010001
```
$\uparrow$
$i = 1$

(1)

$j = 2$
$\downarrow (\times)$
```
  001
  010001
```
$\uparrow$
$i = 1$

(2)

$j = 1$
$\downarrow (\times)$
```
   001
  010001
```
$\uparrow$
$i = 2$

(3)

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

The following figures show a trace of the text search algorithm where we are searching for the pattern "001" in the text "010001".

$$\begin{array}{ccc}
j = 1 & j = 2 & j = 3 \\
\downarrow & \downarrow & \downarrow (\times) \\
\texttt{\ \ 001} & \texttt{\ 001} & \texttt{001} \\
\texttt{010001} & \texttt{010001} & \texttt{010001} \\
\uparrow & \uparrow & \uparrow \\
i = 3 & i = 3 & i = 3 \\
(4) & (5) & (6)
\end{array}$$

Example 1. *Text Search (pseudo code)*

Input: *p (indexed from* 1 *to m), m, t (indexed from* 1 *to n), n;* **Output**: *i or* 0

The following figures show a trace of the text search algorithm where we are searching for the pattern "001" in the text "010001".

```
  j = 1
    ↓
   001
010001
   ↑
 i = 4
```

(7)

```
    j = 2
    ↓
   001
010001
   ↑
 i = 4
```

(8)

```
     j = 3
     ↓
   001
010001
   ↑
 i = 4
```

(9)

Remark.

Remark.

text search(p, m, t, n) {
for $(i = 1;\ i \leq nm + 1;\ i = i + 1)\{$
$j = 1$
while $(t_{i+j1} == p_j)$ {
$j = j + 1$
if $(j > m)$
return i } }
return 0 }

Remark.

text search(p, m, t, n) {
for $(i = 1;\ i \leq nm + 1;\ i = i + 1)\{$
$j = 1$
while $(t_{i+j1} == p_j)$ {
$j = j + 1$
if $(j > m)$
return i } }
return 0 }

- *i is the index in t of the first character of the substring*

Remark.

text search(p, m, t, n) {
for $(i = 1; i \leq nm + 1; i = i + 1)\{$
$j = 1$
while $(t_{i+j1} == p_j)$ {
$j = j + 1$
if $(j > m)$
return i } }
return 0 }

- *i is the index in t of the first character of the substring*
- *to compare with p, and j is the index in p*

Remark.

text search(p, m, t, n) {
for $(i = 1;\ i \leq nm + 1;\ i = i + 1)\{$
$j = 1$
while $(t_{i+j1} == p_j)$ {
$j = j + 1$
if $(j > m)$
return i } }
return 0 }

- *i is the index in t of the first character of the substring*
- *to compare with p, and j is the index in p*
- *the while loop compares $t_i \ldots t_{i+m1}$ and $p_1 \ldots p_m$*

Sorting

Sorting

- *To sort a sequence is to put it in some specified order.*

Sorting

- *To sort a sequence is to put it in some specified order.*
- *Many sorting algorithms have been devised.*

Sorting

- *To sort a sequence is to put it in some specified order.*
- *Many sorting algorithms have been devised.*
- *We discuss bubble sort (冒泡排序).*

Example 2. *Bubble sort*

Example 2. *Bubble sort*

- *Bubble sort is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed.*

Example 2. *Bubble sort*

- *Bubble sort is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed.*
- *This simple algorithm performs poorly in real world use and is used primarily as an educational tool.*

Example 2. *Bubble sort*

- *Bubble sort is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed.*
- *This simple algorithm performs poorly in real world use and is used primarily as an educational tool.*
- *The algorithm is named for the way the larger elements "bubble" up to the top of the list.*

Example 2. *Bubble sort (pseudo code)*

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list);* **Output**: *l, t*

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list);* **Output**: *l, t*

bubble sort(l,n){ int i, j, temp, $t = 0$; bool swapped;

Example 2. *Bubble sort (pseudo code)*

Input: l *(a list of numbers)*, n *(length of the list)*; **Output**: l, t

bubble sort(l,n){ int i, j, temp, $t = 0$; bool swapped;

for $(i = 0;\ i < n - 1;\ i = i + 1)$ {

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list);* **Output**: *l, t*

bubble sort(l,n){ int i, j, temp, $t = 0$; bool swapped;

for $(i = 0;\ i < n - 1;\ i = i + 1)$ {

swapped $=$ false;

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list)*; **Output**: *l, t*

bubble sort(l,n){ int i, j, temp, $t = 0$; bool swapped;

for $(i = 0;\ i < n - 1;\ i = i + 1)$ {

swapped $=$ false;

for $(j = 0;\ j < n - i - 1;\ j = j + 1)$ {

if $(l[j] > l[j + 1])$ {

temp $= l[j]$; $l[j] = l[j + 1]$;

$l[j + 1] =$ temp; $t{=}t + 1$; swapped $=$ true;} }

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list)*; **Output**: *l, t*

bubble sort(l,n){ int i, j, temp, $t = 0$; bool swapped;

for ($i = 0$; $i < n - 1$; $i = i + 1$) {

swapped = false;

for ($j = 0$; $j < n - i - 1$; $j = j + 1$) {

if ($l[j] > l[j + 1]$) {

temp = $l[j]$; $l[j] = l[j + 1]$;

$l[j + 1]$ = temp; t=$t + 1$; swapped = true;} }

if (swapped == false)

Example 2. *Bubble sort (pseudo code)*

Input: *l (a list of numbers), n (length of the list)*; **Output**: *l, t*

bubble sort$(l,n)\{$ int i, j, temp, $t = 0$; bool swapped;
for $(i = 0;\ i < n - 1;\ i = i + 1)$ {
swapped = false;
for $(j = 0;\ j < n - i - 1;\ j = j + 1)$ {
if $(l[j] > l[j + 1])$ {
temp $= l[j]$; $l[j] = l[j + 1]$;
$l[j + 1]$ = temp; $t{=}t + 1$; swapped = true;} }
if (swapped == false)
break; }
return l, t }

Randomized Algorithms

It is occasionally necessary to **relax the requirements** *of an algorithm.*

Randomized Algorithms

It is occasionally necessary to **relax the requirements** *of an algorithm.*

- *Many algorithms currently in use are not general, deterministic, or even finite.*

Randomized Algorithms

It is occasionally necessary to **relax the requirements** *of an algorithm.*

- *Many algorithms currently in use are not general, deterministic, or even finite.*
- *An operating system (e.g., Windows), for example, is better thought of as a program that never terminates rather than as a finite program with input and output.*

Randomized Algorithms

It is occasionally necessary to **relax the requirements** *of an algorithm.*

- *Many algorithms currently in use are not general, deterministic, or even finite.*
- *An operating system (e.g., Windows), for example, is better thought of as a program that never terminates rather than as a finite program with input and output.*
- *As an illustration, we present an example — a* **randomized algorithm**.

Randomized Algorithms

Randomized Algorithms

- *A* **randomized algorithm** *does not require that the intermediate steps be* **uniquely defined** *and depend only on the inputs and the middle results.*

Randomized Algorithms

- *A* **randomized algorithm** *does not require that the intermediate steps be* **uniquely defined** *and depend only on the inputs and the middle results.*
- **Description.** *A randomized algorithm executes, at some points it makes random choices.*

Randomized Algorithms

A useful function: $\mathrm{rand}(i, j)$

- *It returns a random integer between the integers i and j, inclusive.*

Randomized Algorithms

Randomized Algorithms

Example 3. *Shuffle*

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: *a (shuffled)*

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: a *(shuffled)*

- *The algorithm first swaps (i.e., interchanges the values of)* a_1 *and* $a_{\mathrm{rand}(1,n)}$.

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: a *(shuffled)*

- *The algorithm first swaps (i.e., interchanges the values of)* a_1 *and* $a_{\mathrm{rand}(1,n)}$.
- *Next, the algorithm swaps* a_2 *and* $a_{\mathrm{rand}(2,n)}$.

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: *a (shuffled)*

- *The algorithm first swaps (i.e., interchanges the values of)* a_1 *and* $a_{\text{rand}(1,n)}$.
- *Next, the algorithm swaps* a_2 *and* $a_{\text{rand}(2,n)}$.
- *The algorithm continues in this manner until it swaps* a_{n-1} *and* $a_{\text{rand}(n-1,n)}$.

Randomized Algorithms

Example 3. *Shuffle*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: a *(shuffled)*

- *The algorithm first swaps (i.e., interchanges the values of)* a_1 *and* $a_{\mathrm{rand}(1,n)}$.
- *Next, the algorithm swaps* a_2 *and* $a_{\mathrm{rand}(2,n)}$.
- *The algorithm continues in this manner until it swaps* a_{n-1} *and* $a_{\mathrm{rand}(n-1,n)}$.
- *Output* a.

Randomized Algorithms

Example 3. *Shuffle (pseudo code)*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$.

Randomized Algorithms

Example 3. *Shuffle (pseudo code)*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$.

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: *a (shuffled)*

Randomized Algorithms

Example 3. *Shuffle (pseudo code)*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$.

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers)*, n *(length of the list)*; **Output**: a *(shuffled)*

shuffle (a,n){
for $(i = 0;\ i < n - 1;\ i = i + 1)$ {
swapp$(a_i, a_{\text{rand}(i,n)})$
}

Randomized Algorithms

Example 3. *Shuffle (pseudo code)*

This algorithm shuffles the values in the sequence $(a_1, \ldots, a_n)$.

Input: $a = (a_1, \ldots, a_n)$ *(a list of numbers), n (length of the list);* **Output**: *a (shuffled)*

```
shuffle (a,n){
for (i = 0; i < n − 1; i = i + 1) {
swapp(a_i, a_rand(i,n))
}
```

Notice that the output (i.e., the rearranged sequence) depends on the random choices made by the random number generator.

4.3 Analysis of an algorithm

4.3 Analysis of an algorithm

- *In this section , we estimate the time and space (complexity) needed to execute the algorithm.*

4.3 Analysis of an algorithm

- *In this section , we estimate the time and space (complexity) needed to execute the algorithm.*

The space estimations depend so much on the hardware, so we just estimate the time in this section.

- *The time needed to execute an algorithm is a function of the input.*

- *The time needed to execute an algorithm is a function of the input.*
- *Instead of dealing directly with the input, we use the size of the input.*

Estimating the time: *we count some fundamental, dominating steps of the algorithm and obtain a useful measure of the time.*

Estimating the time: *we count some fundamental, dominating steps of the algorithm and obtain a useful measure of the time.*

- Example. if the principal activity of an algorithm is making **comparisons**,
  we might **count the number of comparisons**.

Estimating the time: *we count some fundamental, dominating steps of the algorithm and obtain a useful measure of the time.*

- Example. if the principal activity of an algorithm is making **comparisons**,
  we might **count the number of comparisons**.
- Example. if an algorithm consists of a single loop whose body executes in at most C steps, for some constant C,
  we might **count the number of iterations of the loop**.

Estimating the time: *we count some fundamental, dominating steps of the algorithm and obtain a useful measure of the time.*

- Example. if the principal activity of an algorithm is making **comparisons**,
  we might **count the number of comparisons**.
- Example. if an algorithm consists of a single loop whose body executes in at most C steps, for some constant C,
  we might **count the number of iterations of the loop**.

So the function is not so precise, but the growing order is.

If the size of the input is n.

If the size of the input is n.
We can ask for the maximum/minimum/average time function to execute the algorithm.

4.3 Analysis of Algorithms

If the size of the input is n.
We can ask for the maximum/minimum/average time function to execute the algorithm.

We have

If the size of the input is n.
We can ask for the maximum/minimum/average time function to execute the algorithm.

We have

Best-case time = minimum time *needed to execute the algorithm for inputs of size n.*

If the size of the input is n.
We can ask for the maximum/minimum/average time function to execute the algorithm.

We have

> **Best-case time = minimum time** *needed to execute the algorithm for inputs of size n.*
> **Worst-case time = maximum time** *needed to execute the algorithm for inputs of size n.*

If the size of the input is n.
We can ask for the maximum/minimum/average time function to execute the algorithm.

We have

Best-case time = minimum time *needed to execute the algorithm for inputs of size n.*
Worst-case time = maximum time *needed to execute the algorithm for inputs of size n.*
Average-case time = time averaged over all possible inputs of size *n.*

Example. *Time Complexity Analysis of Bubble Sort*

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

- **Best-case time**: $comparisons = n - 1$, $swaps = 0$;

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

- **Best-case time**: $comparisons = n - 1$, $swaps = 0$;
- **Worst-case time**: $comparisons = \frac{n(n-1)}{2}$, $swaps = \frac{n(n-1)}{2}$;

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

- **Best-case time**: *comparisons* $= n - 1$, *swaps* $= 0$;
- **Worst-case time**: *comparisons* $= \frac{n(n-1)}{2}$, *swaps* $= \frac{n(n-1)}{2}$;
- **Average-case time**: *Very hard to compute,*
  comparisons $= \frac{1}{2}\left(n^2 - n \cdot \ln n - (\gamma + \ln(2) - 1) \cdot n\right) + \mathcal{O}(\sqrt{n})$,
  swaps $= \frac{1}{4}(n^2 - n)$.

A better idea to consider the time functions is to introduce the following ***Asymptotic Notations***.

A better idea to consider the time functions is to introduce the following **Asymptotic Notations**.

Definition. *($O($_$)$)*

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$. *We write*

$$f(n) = O(g(n))$$

and say that $f(n)$ *is* **of order at most** $g(n)$ *or* $f(n)$ *is* **big oh** *of* $g(n)$ *if there exists a positive constant* C_1 *such that*

$$|f(n)| \leq C_1 |g(n)|$$

for all but finitely many positive integers n.

A better idea to consider the time functions is to introduce the following **Asymptotic Notations**.

Definition. $(\Omega(\_))$

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$. *We write*

$$f(n) = \Omega(g(n))$$

and say that $f(n)$ *is of* **order at least** $g(n)$ *or* $f(n)$ *is* **omega** *of* $g(n)$ *if there exists a positive constant* C_2 *such that*

$$|f(n)| \geq C_2|g(n)|$$

for all but finitely many positive integers n.

A better idea to consider the time functions is to introduce the following **Asymptotic Notations**.

Definition ($\Theta(\_)$)

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$. *We write*

$$f(n) = \Theta(g(n))$$

and say that $f(n)$ *is* **of order** $g(n)$ *or* $f(n)$ *is* **theta** *of* $g(n)$ *if* $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$

Asymptotic Notations.

Definition. $(\boldsymbol{O}(\_), \Omega(\_), \Theta(\_))$

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.

- $f(n) = O(g(n))$: *exists* $C_1 > 0$ *s.t.* $|f(n)| \leq C_1|g(n)|$ *for n sufficiently large.*
- $f(n) = \Omega(g(n))$: *exists* $C_2 > 0$ *s.t.* $|f(n)| \geq C_2|g(n)|$ *for n sufficiently large.*
- $f(n) = \Theta(g(n))$: $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

Asymptotic Notations.

Definition. *($O(\_), \Omega(\_), \Theta(\_)$)*

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.

- $f(n) = O(g(n))$: *exists* $C_1 > 0$ *s.t.* $|f(n)| \leq C_1|g(n)|$ *for* n *sufficiently large.*
- $f(n) = \Omega(g(n))$: *exists* $C_2 > 0$ *s.t.* $|f(n)| \geq C_2|g(n)|$ *for* n *sufficiently large.*
- $f(n) = \Theta(g(n))$: $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

When we analyze the growth of complexity functions, $f(n)$ *and* $g(n)$ *are always positive.*

Asymptotic Notations.

Definition. $(O(\_),\Omega(\_),\Theta(\_))$

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.

- $f(n) = O(g(n))$: *exists* $C_1 > 0$ *s.t.* $|f(n)| \leq C_1|g(n)|$ *for n sufficiently large.*
- $f(n) = \Omega(g(n))$: *exists* $C_2 > 0$ *s.t.* $|f(n)| \geq C_2|g(n)|$ *for n sufficiently large.*
- $f(n) = \Theta(g(n))$: $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

When we analyze the growth of complexity functions, $f(n)$ *and* $g(n)$ *are always positive. Therefore, we can simplify the big-O requirement to* $f(n) \leq C_1 g(n)$ *for n sufficiently large.*

If we use the big-O notations, the complexity of the bubble sort can be written as follows.

If we use the big-O notations, the complexity of the bubble sort can be written as follows.

Example. *Time Complexity Analysis of Bubble Sort*

If we use the big-O notations, the complexity of the bubble sort can be written as follows.

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

If we use the big-O notations, the complexity of the bubble sort can be written as follows.

Example. *Time Complexity Analysis of Bubble Sort*

Complexity=number of comparisons + number of swaps.
Size of input =n (length of series).

- **Best-case time**: $comparisons = O(n)n$, $swaps = O(1)$;
- **Worst-case time**: $comparisons = O(n^2)$, $swaps = O(n^2)$;
- **Average-case time**: *Very hard to compute,* $comparisons = O(n^2)$, $swaps = O(n^2)$.

We have the following elementary properties of the asymptotic notations.

We have the following elementary properties of the asymptotic notations.

Proposition.

Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ *be a polynomial in* n *of degree* k*, where each* a_i *is nonnegative. Then*

$$p(n) = \Theta\left(n^k\right).$$

We have the following elementary properties of the asymptotic notations.

Proposition.

Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ *be a polynomial in* n *of degree* k, *where each* a_i *is nonnegative. Then*

$$p(n) = \Theta\left(n^k\right).$$

Proof. *When* n *is sufficiently large, we have* $\frac{|a_k|}{2} n^k \leq p(n) \leq 2|a_k| n^k$.

We have the following elementary properties of the asymptotic notations.

We have the following elementary properties of the asymptotic notations.

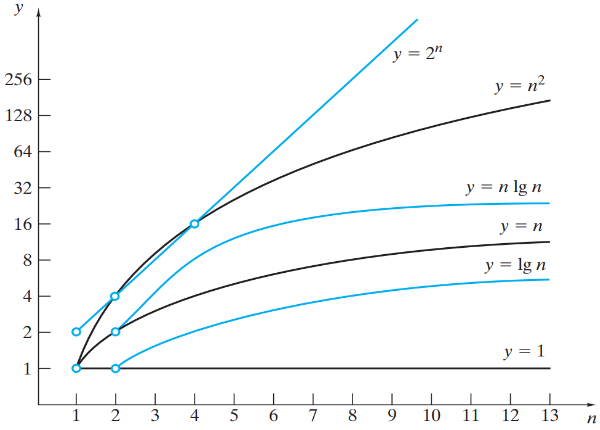


Figure 4.3.1 Growth of some common functions.

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

Certain growth functions occur so often that they are given special names, as shown in Table 4.3.3.

TABLE 4.3.3 ■ **Common Growth Functions**

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

TABLE 4.3.3 ■ Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$6n^6 + n + 4 \qquad\qquad =$$

4.3 Analysis of Algorithms

TABLE 4.3.3 ■ Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$6n^6 + n + 4 \qquad = \qquad O(n^6)$$

TABLE 4.3.3 ■ Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$6n^6 + n + 4 = O(n^6)$$
$$2\ln n + 4n + 3n\ln n =$$

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$
\begin{aligned}
6n^6 + n + 4 &= O(n^6) \\
2\ln n + 4n + 3n\ln n &= O(n\ln n)
\end{aligned}
$$

4.3 Analysis of Algorithms

TABLE 4.3.3 ■ Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$
\begin{aligned}
6n^6 + n + 4 &= O(n^6) \\
2\ln n + 4n + 3n\ln n &= O(n \ln n) \\
\frac{(n+1)(n+3)}{n+2} &=
\end{aligned}
$$

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k)$, $k \geq 1$	Polynomial
$\Theta(c^n)$, $c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$\begin{aligned} 6n^6 + n + 4 &= O(n^6) \\ 2\ln n + 4n + 3n\ln n &= O(n \ln n) \\ \frac{(n+1)(n+3)}{n+2} &= O(n) \end{aligned}$$

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$
\begin{aligned}
6n^6 + n + 4 &= O(n^6) \\
2\ln n + 4n + 3n\ln n &= O(n\ln n) \\
\frac{(n+1)(n+3)}{n+2} &= O(n) \\
2 + 4 + 8 + 16 + \cdots + 2^n &=
\end{aligned}
$$

4.3 Analysis of Algorithms

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$6n^6 + n + 4 = O(n^6)$$

$$2 \ln n + 4n + 3n \ln n = O(n \ln n)$$

$$\frac{(n+1)(n+3)}{n+2} = O(n)$$

$$2 + 4 + 8 + 16 + \cdots + 2^n = O(2^{n+1})$$

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg\lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n\lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k\geq 1$	Polynomial
$\Theta(c^n),\ c>1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$\begin{aligned}
6n^6+n+4 &= O(n^6)\\
2\ln n+4n+3n\ln n &= O(n\ln n)\\
\frac{(n+1)(n+3)}{n+2} &= O(n)\\
2+4+8+16+\cdots+2^n &= O(2^{n+1})\\
\ln(n!) &=
\end{aligned}$$

4.3 Analysis of Algorithms

TABLE 4.3.3 Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k),\ k \geq 1$	Polynomial
$\Theta(c^n),\ c > 1$	Exponential
$\Theta(n!)$	Factorial

Exercise. *Select a theta notation from Table 4.3.3 for the following functions.*

$$\begin{aligned}
6n^6 + n + 4 &= O(n^6) \\
2\ln n + 4n + 3n\ln n &= O(n\ln n) \\
\frac{(n+1)(n+3)}{n+2} &= O(n) \\
2 + 4 + 8 + 16 + \cdots + 2^n &= O(2^{n+1}) \\
\ln(n!) &= O(n\ln n)
\end{aligned}$$

By Stirling's approximation,

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

$$\ln(n!) = n\ln n - n + O(\ln n).$$

- *A problem that has a worst-case* **polynomial-time** *algorithm is considered to have a "good" algorithm; the interpretation is that such a problem has an efficient solution.*

- *A problem that has a worst-case* **polynomial-time** *algorithm is considered to have a "good" algorithm; the interpretation is that such a problem has an efficient solution.*
- *Such problems are called* **feasible** *or* **tractable**.

- *A problem that has a worst-case* **polynomial-time** *algorithm is considered to have a "good" algorithm; the interpretation is that such a problem has an efficient solution.*
- *Such problems are called* **feasible** *or* **tractable.**
- *A problem that does not have a worst-case polynomial-time algorithm is said to be* **intractable.**

- *A problem for which there is no algorithm is said to be* **unsolvable**.

- *A problem for which there is no algorithm is said to be* **unsolvable**.
- *A large number of problems are known to be unsolvable.*

- *A problem for which there is no algorithm is said to be* **unsolvable**.
- *A large number of problems are known to be unsolvable.*
- *One of the earliest problems to be proved unsolvable is the* **halting problem**(*停机问题*)*:*
  Given an arbitrary program and a set of inputs, will the program eventually halt?

4.3 Analysis of Algorithms

- *A problem for which there is no algorithm is said to be* **unsolvable**.
- *A large number of problems are known to be unsolvable.*
- *One of the earliest problems to be proved unsolvable is the* **halting problem**(停机问题):
  Given an arbitrary program and a set of inputs, will the program eventually halt?
- *A large number of solvable problems have undetermined status. We don't know whether they are intractable.*

4.3 Problem-Solving Tips

Apply the techniques learned in the calculus courses.

4.4 Recursive Algorithms

Recursive function *(pseudocode) is a function that invokes itself.*
A **recursive algorithm** *is an algorithm that contains a recursive function.*

Example. *Fibonacci sequence*

Example. *Fibonacci sequence*

Definition

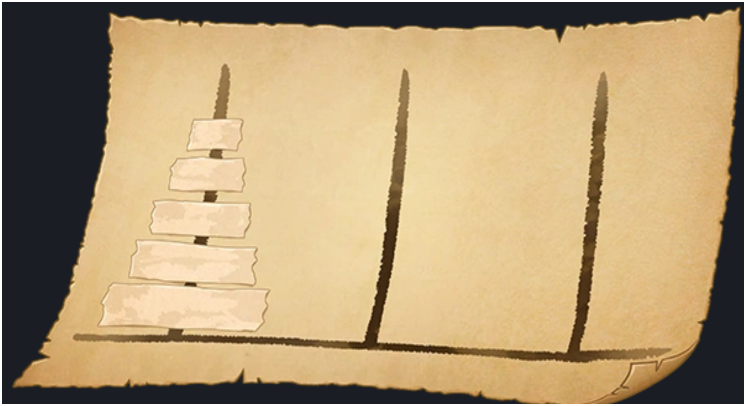
The Fibonacci sequence $\{f_n\}$ *is defined by the recursive formula*

$$f_1 = 1, \quad f_2 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad n \geq 3.$$

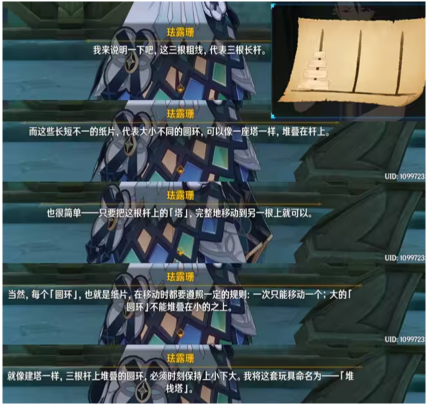
Example.


The Solution and Foundation of "Pagoda Stack"
Faruzan's Hangout Event

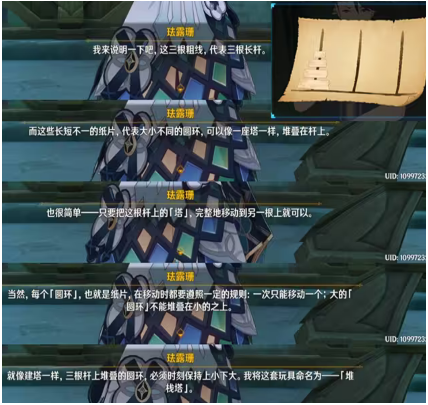
4.4 Recursive Algorithms



4.4 Recursive Algorithms

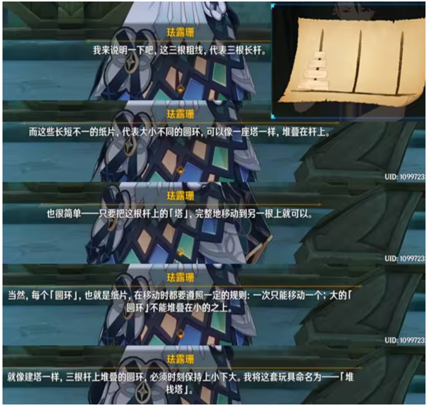


4.4 Recursive Algorithms

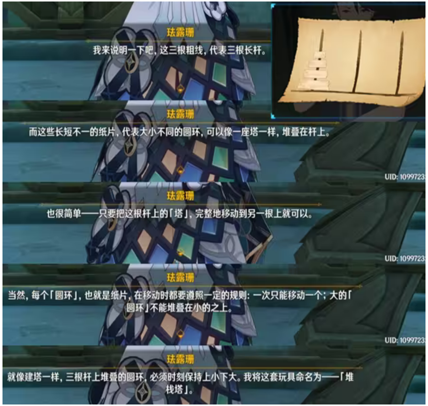


What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?

What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?

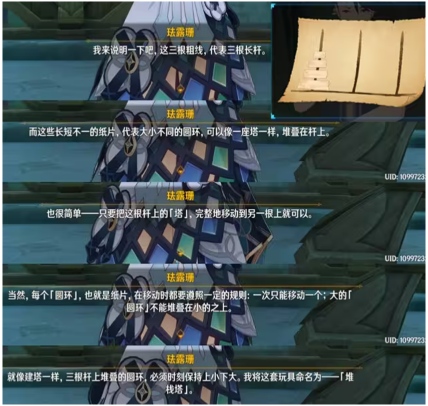


What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?



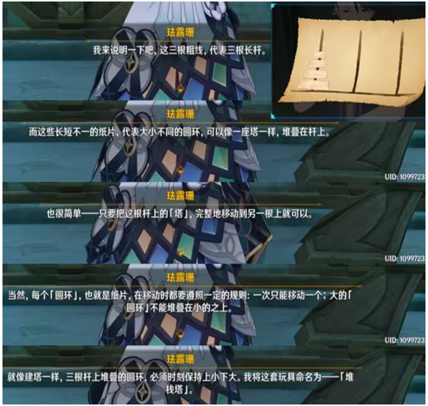
- p_n: *the lowest steps to solve n-layer "Pagoda Stack".*

What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?



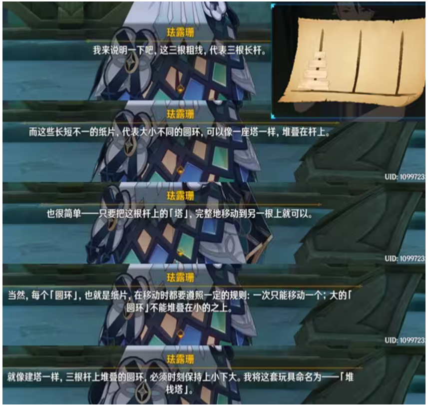
- p_n: *the lowest steps to solve n-layer "Pagoda Stack".*
- *Recursive formula:* $p_n = 2p_{n-1} + 1, p_1 = 1;$

What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?



- p_n: *the lowest steps to solve* n*-layer "Pagoda Stack".*
- *Recursive formula:* $p_n = 2p_{n-1} + 1, p_1 = 1;$
- *High school method:* $p_n + 1 = 2(p_{n-1} + 1)$, $p_n = 2^n - 1.$

What is the lowest number of steps needed to solve a 7-layer "Pagoda Stack"?



- p_n: *the lowest steps to solve n-layer "Pagoda Stack".*
- *Recursive formula:* $p_n = 2p_{n-1} + 1, p_1 = 1;$
- *High school method:* $p_n + 1 = 2(p_{n-1} + 1),$
  $p_n = 2^n - 1.$
- *In particular,* $p_7 = 127.$

4.4 Recursive Algorithms

The End