Recursion and Recurrences

Subhabrata Samajder

IIIT, Delhi

Summer Semester,
2nd May, 2022

Recursion: A Recap

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Example:

```
#include <stdio.h>

int main(void) {
    printf(" The universe is never ending! ");
    main();
    return 0; }
```

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Example:

```
int sum(int n) {
   if (n <= 1)
      return n;
   else
      return (n + sum(n - 1)); }
```

Example: sum(4)

Example: sum(4)

Function call	Value returned		
sum(1)	1		
sum(2)	$2 + \text{sum}(1)$	or	$2 + 1$
sum(3)	$3 + \text{sum}(2)$	or	$3 + 2 + 1$
sum(4)	$4 + \text{sum}(3)$	or	$4 + 3 + 2 + 1$

- The base case is considered,

Example: sum(4)

Function call	Value returned		
sum(1)	1		
sum(2)	$2 + \text{sum}(1)$	or	$2 + 1$
sum(3)	$3 + \text{sum}(2)$	or	$3 + 2 + 1$
sum(4)	$4 + \text{sum}(3)$	or	$4 + 3 + 2 + 1$

- The base case is considered,
- then working out from the base case, the other cases are considered.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

- $\mathrm{sum(n)} = n + (n-1) + \cdots + 1 = n + \mathrm{sum(n-1)}$.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

- $\mathrm{sum(n)} = n + (n-1) + \cdots + 1 = n + \mathrm{sum(n-1)}$.
- The variable n is reduced by 1 each time until
- the base case with $n = 1$ is reached.

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

For example: $5\,! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120$.

Examples: Factorial

$$0\ ! = 1, \quad n\ ! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\ ! = 1, \quad n\ ! = n\cdot((n-1)\ !) \quad \text{for } n > 0$$

For example: $5\ ! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120.$

- **Base Case:** $0! = 1$ and $1! = 1$.

Examples: Factorial

$$0\ ! = 1, \quad n\ ! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\ ! = 1, \quad n\ ! = n\cdot((n-1)\ !) \quad \text{for } n > 0$$

For example: $5\ ! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120$.

- **Base Case:** $0! = 1$ and $1! = 1$.
- **Recursive Case:** $n! = n\cdot(n-1)!$.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12$!!).

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12$!!).
- For $n > 12$, incorrect values are returned.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12$!!).
- For $n > 12$, incorrect values are returned.
- This type of programming error is common!!

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Take Away: Functions that are logically correct can return incorrect values if the logical operations in the body of the function are beyond the integer precision available to the system!!

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

"What to do?"

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

```
int IterFactorial (int n) {        /* iterative version */
    int product = 1;

    for ( ; n > 1; - -n)
        product *= n;
    return product; }
```

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

```
int IterFactorial (int n) {        /* iterative version */
    int product = 1;

    for ( ; n > 1; - -n)
        product *= n;
    return product; }
```

IterFactorial(n): Takes only 1 function call.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

- Recursion is more elegant.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

"Then why bother?"

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.
- Takes care of its bookkeeping by stacking arguments and variables for each invocation.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

"Then why bother?"

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.
- Takes care of its bookkeeping by stacking arguments and variables for each invocation.
- *This stacking of arguments, while invisible to the user, is still costly in time and space.*

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

"Then why bother?"

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.
- Takes care of its bookkeeping by stacking arguments and variables for each invocation.
- *This stacking of arguments, while invisible to the user, is still costly in time and space.*
- On some machines a simple recursive call with one integer argument can require eight 32-bit words on the stack.

Fibonacci Sequence

Fibonacci sequence is defined recursively as

$$f_1 = 1, \quad f_2 = 1, \quad f_{i+1} = f_i + f_{i-1} \quad \text{for } i = 1, 2, \ldots$$

Every element ($i \geq 3$) is the sum of it's previous two elements.

The sequence begins as $1, 1, 2, 3, 5, \ldots$

Fibonacci Sequence

Consider the following sequence:

$$
\begin{aligned}
2/1 &= 2.0 \quad \text{(bigger)}\\
3/2 &= 1.5 \quad \text{(smaller)}\\
5/3 &= 1.67 \quad \text{(bigger)}\\
8/5 &= 1.6 \quad \text{(smaller)}\\
13/8 &= 1.625 \quad \text{(bigger)}\\
21/13 &= 1.615 \quad \text{(smaller)}\\
34/21 &= 1.619 \quad \text{(bigger)}\\
55/34 &= 1.618 \quad \text{(smaller)}\\
89/55 &= 1.618
\end{aligned}
$$

Fibonacci Sequence

Consider the following sequence:

$$
\begin{aligned}
2/1 &= 2.0 \quad \text{(bigger)} \\
3/2 &= 1.5 \quad \text{(smaller)} \\
5/3 &= 1.67 \quad \text{(bigger)} \\
8/5 &= 1.6 \quad \text{(smaller)} \\
13/8 &= 1.625 \quad \text{(bigger)} \\
21/13 &= 1.615 \quad \text{(smaller)} \\
34/21 &= 1.619 \quad \text{(bigger)} \\
55/34 &= 1.618 \quad \text{(smaller)} \\
89/55 &= 1.618
\end{aligned}
$$

Note:

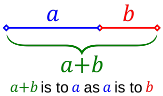
- This sequence seem to be converging!
- It converges to the *golden ratio*.

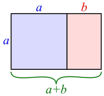
Golden Ratio

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887498948482$$

- It is a special number.
- Couple of ways to visually understand it are with

a *line segment* *Golden rectangles*





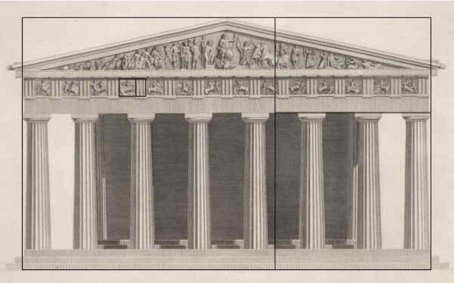
- It is an *irrational number* that is a root of the quadratic equation

$$x^2 - x - 1 = 0$$

Golden Ratio

- Reciprocal of φ or φ^{-1}:
  - $f_n/f_{n+1} \to 0.618$ as $n \to \infty$.
  - This is the reciprocal of φ: $1/1.618 = 0.618$.
  - It is highly unusual for the decimal representation of the fractional part of a number and its reciprocal to be exactly the same.
  - This only adds to the mystique of the Golden Ratio and leads us to ask: What makes it so special?

- **Some examples:**



The ancient temple in Greece fits almost precisely into a golden rectangle.

- **Some examples:**

1 : 1.618

Butterflies.

Recursive Fibonacci Sequence: Function Calls

```
int RecFibonacci (int n) {
    if (n <= 1)
        return n;
    else
        return (RecFibonacci(n - 1) + RecFibonacci(n - 2)); }
```

Recursive Fibonacci Sequence: Function Calls

Value of n	Value of RecFibonacci(n)	Number of function calls required to recursively compute RecFibonacci(n)
0	0	1
1	1	1
2	1	3
......		
23	28657	92735
24	46368	150049
......		
42	267914296	866988873
43	433494437	1402817465

Requires a large number of function calls even for moderate values of n.

Recursion: Summary

- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.

Recursion: Summary

- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.
- It is sometimes necessary to recode to an equivalent iterative method.

Recursion: Summary

- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.
- It is sometimes necessary to recode to an equivalent iterative method.
- Some programmers feel that because the use of recursion is inefficient, it should not be used.

Recursion: Summary

- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.
- It is sometimes necessary to recode to an equivalent iterative method.
- Some programmers feel that because the use of recursion is inefficient, it should not be used.
- The inefficiencies, however, are often of little consequence - as in the case of the quicksort algorithm.

Recursion: Summary

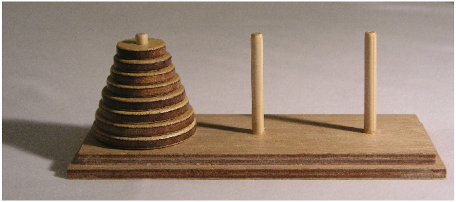
- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.
- It is sometimes necessary to recode to an equivalent iterative method.
- Some programmers feel that because the use of recursion is inefficient, it should not be used.
- The inefficiencies, however, are often of little consequence - as in the case of the quicksort algorithm.
- For many applications, recursive code is easier to write, understand, maintain.

Recursion: Summary

- It is seductive to use recursion.
- But one must be careful about run-time limitations and inefficiencies.
- It is sometimes necessary to recode to an equivalent iterative method.
- Some programmers feel that because the use of recursion is inefficient, it should not be used.
- The inefficiencies, however, are often of little consequence - as in the case of the quicksort algorithm.
- For many applications, recursive code is easier to write, understand, maintain.
- These reasons often prescribe its use.

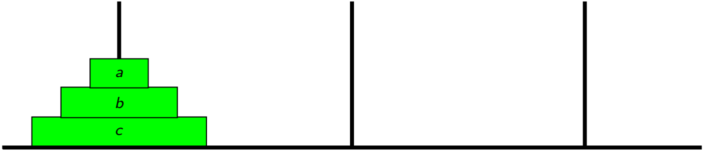
Towers of Hanoi

Towers of Hanoi: Problem Statement



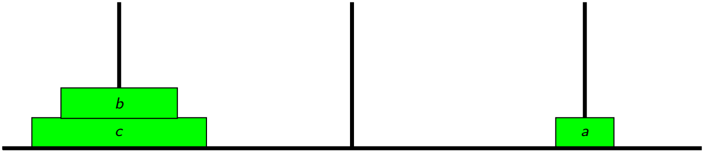
- There are three towers.
- n disks of decreasing radius are placed on the 1^{st} tower.
- Move all of the disks from the 1^{st} tower to the 3^{rd} tower.
- **Condition:** At no moment of time can a larger disk be placed on top of smaller disks.
- The remaining tower can be used to temporarily hold disks.

Towers of Hanoi: Solution for $n = 3$



Towers of Hanoi: Solution for $n = 3$

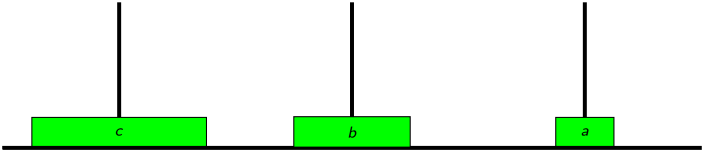
Step 1: Move disks a to tower 3.



Towers of Hanoi: Solution for $n = 3$

Step 1: Move disks a to tower 3.

Step 2: Move disks b to tower 2.

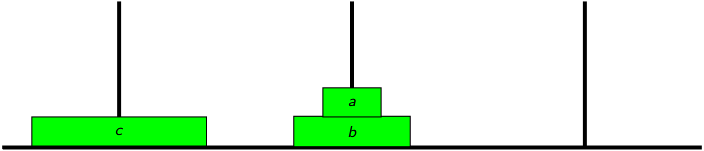


Towers of Hanoi: Solution for $n = 3$

Step 1: Move disks a to tower 3.

Step 2: Move disks b to tower 2.

Step 3: Move disks a to tower 2.



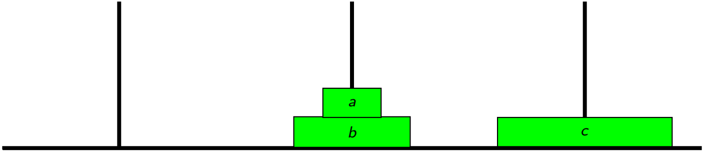
Towers of Hanoi: Solution for $n = 3$

Step 1: Move disks a to tower 3.

Step 2: Move disks b to tower 2.

Step 3: Move disks a to tower 2.

Step 4: Move disks c to tower 3.



Towers of Hanoi: Solution for $n = 3$

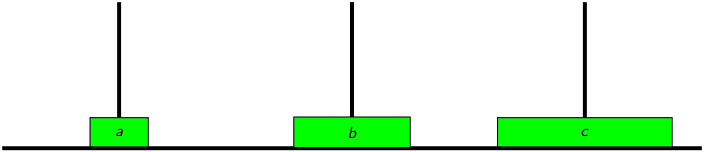
Step 1: Move disks a to tower 3.

Step 2: Move disks b to tower 2.

Step 3: Move disks a to tower 2.

Step 4: Move disks c to tower 3.

Step 5: Move disks a to tower 1.



Towers of Hanoi: Solution for $n = 3$

Step 1: Move disks a to tower 3.

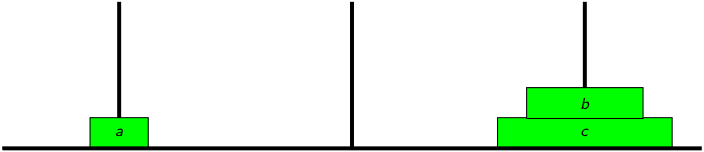
Step 2: Move disks b to tower 2.

Step 3: Move disks a to tower 2.

Step 4: Move disks c to tower 3.

Step 5: Move disks a to tower 1.

Step 6: Move disks b to tower 3.



Towers of Hanoi: Solution for $n = 3$

Step 1: Move disks a to tower 3.

Step 2: Move disks b to tower 2.

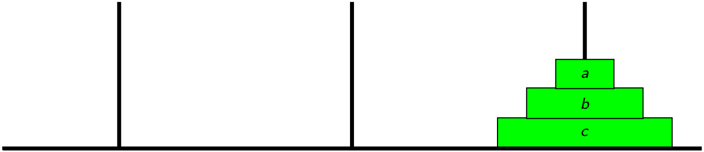
Step 3: Move disks a to tower 2.

Step 4: Move disks c to tower 3.

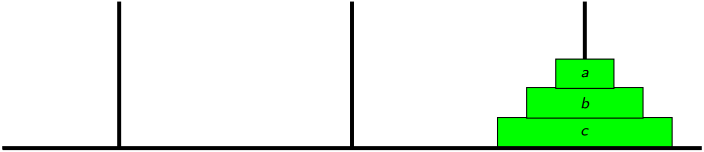
Step 5: Move disks a to tower 1.

Step 6: Move disks b to tower 3.

Step 7: Move disks a to tower 3.



Towers of Hanoi: Solution for $n = 3$



Homework:

- Write a recursive algorithm that solves the Towers of Hanoi problem for n disks.
- Implement your algorithm in C.

Recurrences

Recurrence

Definition

A **recurrence relation** is an equation that expresses each element of a sequence $\{a_n\}_{n=0}^{\infty}$ as a function of the preceding ones, i.e,

$$a_n = \psi(a_0, a_1, \ldots, a_{n-1}).$$

Why Recurrences?

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.
- $\therefore$ a sequence.

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.
- $\therefore$ a sequence.
- **Recursive algorithms:** It is not easy to compute the number of instructions by looking at code.

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.
- $\therefore$ a sequence.
- **Recursive algorithms:** It is not easy to compute the number of instructions by looking at code.
    - The number of instructions in one instance of function call depends on the number of instructions executed when recursive calls are made.

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.
- $\therefore$ a sequence.
- **Recursive algorithms:** It is not easy to compute the number of instructions by looking at code.
  - The number of instructions in one instance of function call depends on the number of instructions executed when recursive calls are made.
  - In such cases it is easier for us to express it as some *recurrence relation* of the times/space complexity.

Why Recurrences?

- Computing time/space complexity $\equiv$ a counting problem.
- $\therefore$ a sequence.
- **Recursive algorithms:** It is not easy to compute the number of instructions by looking at code.
  - The number of instructions in one instance of function call depends on the number of instructions executed when recursive calls are made.
  - In such cases it is easier for us to express it as some *recurrence relation* of the times/space complexity.
- Appears frequently in the analysis of algorithms.

Questions?

1. How to form recurrence relation for a counting problem?

Questions?

1. How to form recurrence relation for a counting problem? Easy!!

Questions?

1. How to form recurrence relation for a counting problem? Easy!!

2. How to solve such relations?

Questions?

1. How to form recurrence relation for a counting problem? Easy!!

2. How to solve such relations?

Outline

- We briefly discuss few useful technique for solving recurrences.

- Present general solutions of two classes of recurrences that are among the most common recurrences involved in analyzing algorithms.

Intelligent Guesses

- Guessing a solution may seem like a nonscientific method!
- But, keeping our pride aside, it works very well for a wide class of recurrence relations.
- It works even better when we are *not* trying to find the exact solution, but only an upper bound.
- **Why guess?** Proving a certain bound is valid is easier than deriving that bound.

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- Can compute the value of the function for every n.

  Example:

  $$F(3) = F(2) + F(1) = 2,\ F(4) = F(3) + F(2) = 3, \ldots.$$

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- Can compute the value of the function for every n.

  Example:

  $$F(3) = F(2) + F(1) = 2,\ F(4) = F(3) + F(2) = 3, \ldots.$$

- **Note:** By definition we need $n-2$ steps to compute $F(n)$.

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- Can compute the value of the function for every n.

  Example:

  $$F(3) = F(2) + F(1) = 2,\ F(4) = F(3) + F(2) = 3, \ldots.$$

- **Note:** By definition we need $n-2$ steps to compute $F(n)$.
- Would be more convenient to have an explicit (or closed-form) expression for $F(n)$.
  - It would enable us to compute $F(n)$ quickly.
  - We can also compare $F(n)$ with other known functions.

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- $F(n)$ is the sum of two previous values.

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- $F(n)$ is the sum of two previous values.
- **Possible guess:** $F(n)$ is doubled every time, i.e., $F(n) \approx 2^n$.

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- $F(n)$ is the sum of two previous values.
- **Possible guess:** $F(n)$ is doubled every time, i.e., $F(n) \approx 2^n$.
- Let $F(n) = ca^n$, then we get

$$ca^n = ca^{n-1} + ca^{n-2} \;\Rightarrow\; a^2 = a + 1 \quad \text{(Characteristic equation)}.$$

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- $F(n)$ is the sum of two previous values.
- **Possible guess:** $F(n)$ is doubled every time, i.e., $F(n) \approx 2^n$.
- Let $F(n) = ca^n$, then we get

  $ca^n = ca^{n-1} + ca^{n-2} \Rightarrow a^2 = a + 1$ (Characteristic equation).

- **Solving:** $a_1 = (1+\sqrt{5})/2$ (> 0) and $a_2 = (1-\sqrt{5})/2$ (< 0).

Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, F(2) = 1.$$

- $F(n)$ is the sum of two previous values.
- **Possible guess:** $F(n)$ is doubled every time, i.e., $F(n) \approx 2^n$.
- Let $F(n) = ca^n$, then we get

  $ca^n = ca^{n-1} + ca^{n-2} \;\Rightarrow\; a^2 = a+1$ (Characteristic equation).

- **Solving:** $a_1 = (1+\sqrt{5})/2$ (> 0) and $a_2 = (1-\sqrt{5})/2$ (< 0).
- $\therefore$ $F(n) = \mathcal{O}((a_1)^n)$.
  - Find a constant c such that $c(a_1)^n \geq F(1)$ and $F(2)$.

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.
- **General Solution:** $c_1(a_1)^n + c_2(a_2)^n$.

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.
- **General Solution:** $c_1(a_1)^n + c_2(a_2)^n$.
- Find c_1 and c_2, s. t., $F(1) = 1$ and $F(2) = 1$.

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.
- **General Solution:** $c_1(a_1)^n + c_2(a_2)^n$.
- Find c_1 and c_2, s. t., $F(1) = 1$ and $F(2) = 1$.
- **Solving:** $c_1 = 1/\sqrt{5}$, and $c_2 = -1/\sqrt{5}$.

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.
- **General Solution:** $c_1(a_1)^n + c_2(a_2)^n$.
- Find c_1 and c_2, s. t., $F(1) = 1$ and $F(2) = 1$.
- **Solving:** $c_1 = 1/\sqrt{5}$, and $c_2 = -1/\sqrt{5}$.
- **Exact solution:**

$$F(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - \frac{1}{\sqrt{5}}\left[\frac{1-\sqrt{5}}{2}\right]^n.$$

Fibonacci Sequence: Exact Solution

- Need to consider the initial values more carefully.
- **General Solution:** $c_1(a_1)^n + c_2(a_2)^n$.
- Find c_1 and c_2, s. t., $F(1) = 1$ and $F(2) = 1$.
- **Solving:** $c_1 = 1/\sqrt{5}$, and $c_2 = -1/\sqrt{5}$.
- **Exact solution:**

$$F(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - \frac{1}{\sqrt{5}}\left[\frac{1-\sqrt{5}}{2}\right]^n.$$

Note: This idea, can be used to solve recurrences of the form

$F(n) = b_1F(n-1) + b_2F(n-2) + \cdots + b_kF(n-k) \quad$ (k constant).

Books Consulted

1. *Introduction to Algorithms: A Creative Approach* by Udi Manber.

2. *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!