Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Analyzing Recursive Algorithms

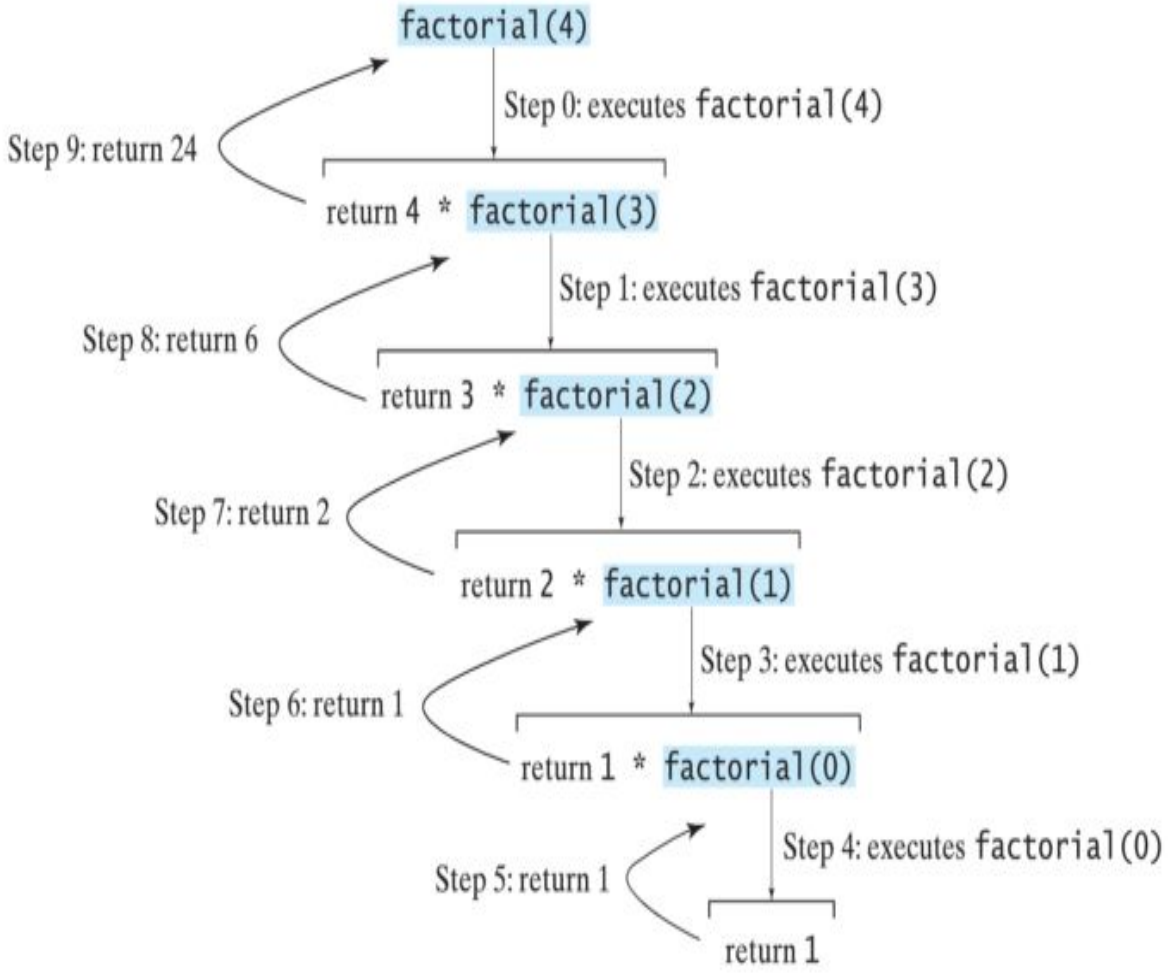
- In lecture 5 & 6, we introduced mathematical techniques for analyzing the efficiency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm
- We use notations such as big-Oh to summarize the relationship between the number of operations and the input size for a problem
- Now, we learn how to perform this type of running-time analysis to recursive algorithms

Analyzing Recursive Algorithms

- In recursive algorithms, for each invocation of the method, we only account for the number of operations that are performed within the body of that activation
- We can then account for the overall number of operations that are executed as part of the recursive algorithm by taking the sum, over all activations, of the number of operations that take place during each individual activation
- In general, we may rely on the intuition afforded by a ***recursion trace*** in recognizing how many recursive activations occur, and how the parameterization of each activation can be used to estimate the number of primitive operations that occur within the body of that activation

Analyzing Factorials

- A sample recursion trace for our factorial method is as shown in Figure
- To compute factorial(n), we see that there are a ***total of n+1 activations***, as the parameter decreases from n in the first call, to n−1 in the second call, and so on, until reaching the base case with parameter 0
- Each individual activation of factorial executes a ***constant number of operations***
- There are n+1 activations, each of which accounts for O(1) operations
- Therefore, we conclude that the overall number of operations for computing factorial(n) is O(n)



Analyzing Binary Search

- A constant number of primitive operations are executed during each recursive call of the binary search method
- The running time is proportional to the *number of recursive calls performed*
- At most $\lfloor \log n \rfloor + 1$ recursive calls are made during a binary search of a sequence having n elements

Proposition 5.2: *The binary search algorithm runs in $O(\log n)$ time for a sorted array with n elements.*

Analyzing Binary Search

- **Proposition**: The binary search algorithm runs in ***O(log n)*** time for a sorted array with ***n*** elements
- **Justification**:
  - Each recursive call the number of candidate elements still to be searched is given by the value ***high−low+1***
  - The number of remaining *candidates is reduced by at least one-half with each recursive call*

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\le\frac{\text{high}-\text{low}+1}{2}$$

or

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\le\frac{\text{high}-\text{low}+1}{2}.$$

  - Initially, the number of candidates is ***n***; after the first call in a binary search, it is at most ***n/2***; after the second call, it is at most ***n/4***; and so on
  - In general, *after the **j th** call in a binary search, the number of candidate elements remaining is at most **$n/2^j$***
  - In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate elements
  - The maximum number of recursive calls performed, is the smallest integer **r** such that **$n/2^r < 1$**. In other words, **r** is the smallest integer such that **r > log n**
  - Thus, we have **$r = \lfloor \log n \rfloor + 1$**, which implies that ***binary search runs in O(log n) time***

Examples of Recursion

- If a recursive call starts at most one other, we call this a ***linear recursion***
  - If a recursive method is designed so that *each invocation of the body makes at most one new recursive call*, this is known as linear recursion
- If a recursive call may start two others, we call this a ***binary recursion***
  - When a method makes two recursive calls, we say that it uses *binary recursion*
- If a recursive call may start three or more others, this is ***multiple recursion***
  - Generalizing from binary recursion, we define multiple recursion as a process in which a method may *make more than two recursive calls*

Linear Recursion

- If a recursive method is designed so that each invocation of the body makes at most one new recursive call, this is known as *linear recursion*
- The implementation of the ***factorial*** method is a clear example of linear recursion
- **Is the binary search algorithm also an example of linear recursion?**
- Linear recursion is that any *recursion trace will appear as a single sequence of calls*
- Note that the linear recursion terminology *reflects the structure of the recursion trace*, ***not the asymptotic analysis of the running time***
- We have seen that binary search runs in **O(log n)** time

Summing the Elements of an Array Recursively

- Linear recursion can be a useful tool for processing a sequence, such as a Java array
- How to compute the sum of an array of **n** integers using recursion?

- We can solve this summation problem using linear recursion by observing that

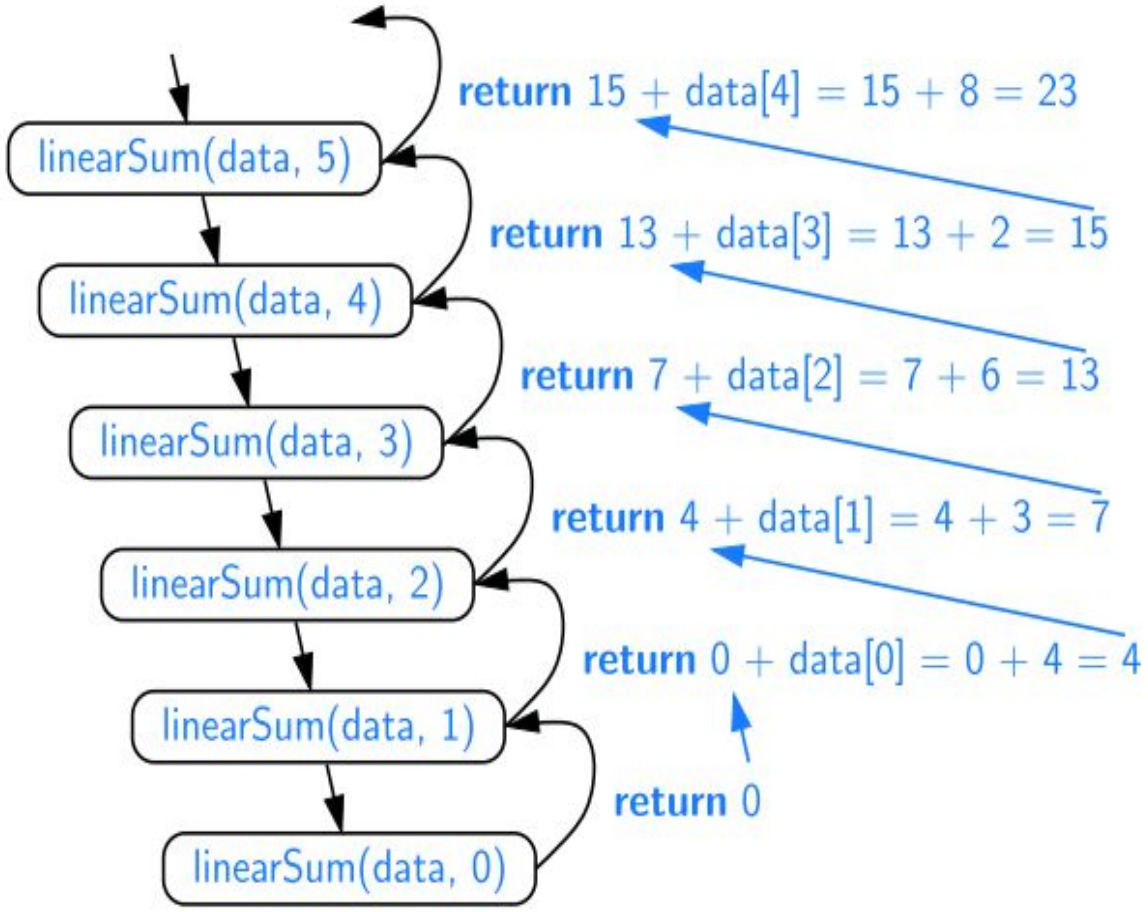
  if n = 0 sum = 0,

  if n>0 sum = the first n−1 integers in the array plus the last value in the array

Summing the Elements of an Array Recursively

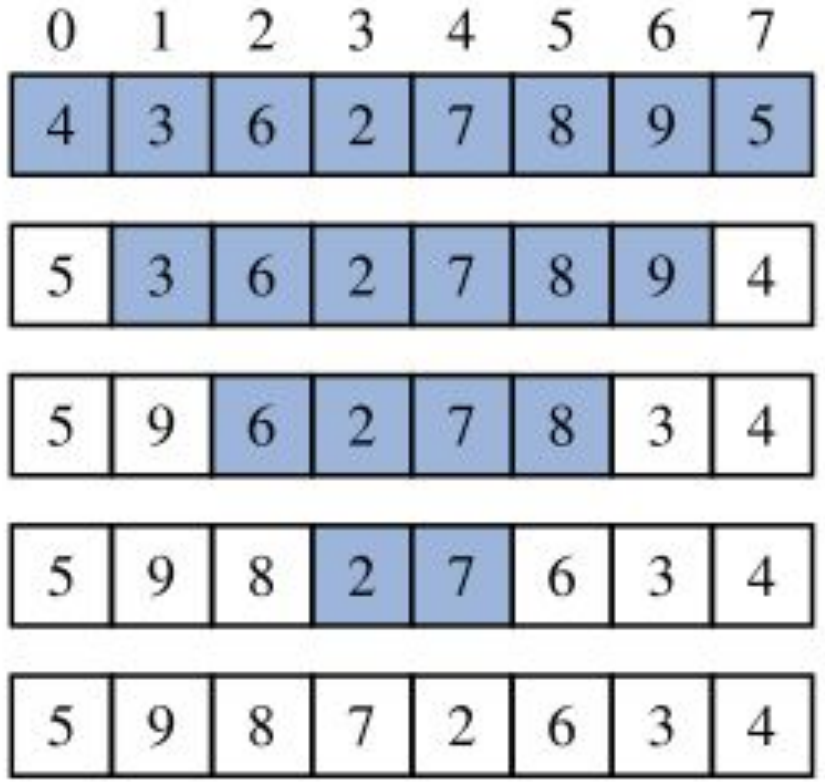
```
/** Returns the sum of the first n integers of the given array. */
public static int linearSum(int[ ] data, int n) {
  if (n == 0)
    return 0;
  else
    return linearSum(data, n−1) + data[n−1];
}
```

- The algorithm will take O(n) time, because it spends a constant amount of time performing the nonrecursive part of each call
- The memory space used by the algorithm (in addition to the array) is also O(n)



Reversing a Sequence with Recursion

- Consider the problem of reversing the **n** elements of an array, so that the first element becomes the last, the second element becomes second to the last, and so on
- How to solve this problem using linear recursion?



Reversing a Sequence with Recursion

- Whenever a recursive call is made, there will be two fewer elements in the relevant portion of the array
- Eventually a base case is reached when the condition low < high fails
- The recursive algorithm of Code Fragment is guaranteed to terminate after a total of **$1+\lfloor n/2 \rfloor$** recursive calls
- Because each call involves a constant amount of work, the entire process runs in ***O(n) time***

```
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                                  // if at least two elements in subarray
    int temp = data[low];                            // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1);           // recur on the rest
  }
}
```

Home assignment Computing Powers

- Consider the problem of raising a number x to an arbitrary nonnegative integer n. That is, we wish to compute the power function, defined as power(x,n) = x^n
- A trivial recursive definition follows from the fact that xn = x · xn−1 for n > 0

A trivial recursive definition follows from the fact that $x^n = x \cdot x^{n-1}$ for $n > 0$.

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

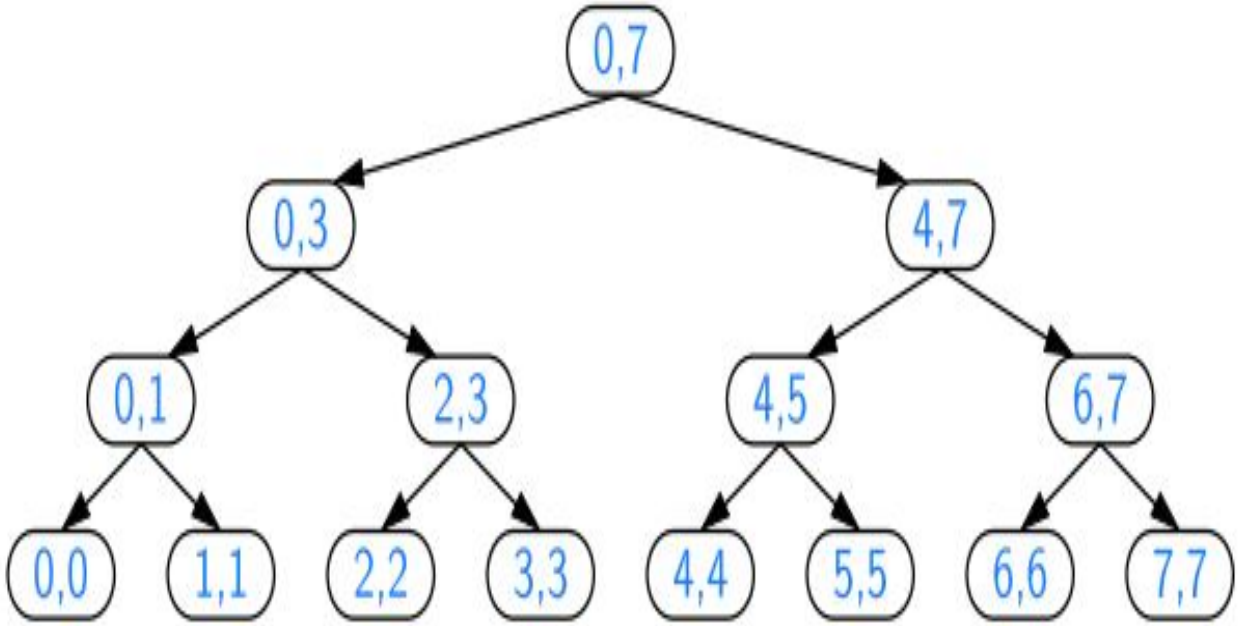
Binary Recursion - binarySum

- When a method makes two recursive calls, we say that it uses binary recursion
- Computing the sum of one or zero values is trivial. With two or more values, we can recursively compute the sum of the first half, and the sum of the second half, and add those sums together

```
/** Returns the sum of subarray data[low] through data[high] inclusive. */
public static int binarySum(int[ ] data, int low, int high) {
  if (low > high)                                    // zero elements in subarray
    return 0;
  else if (low == high)                              // one element in subarray
    return data[low];
  else {
    int mid = (low + high) / 2;
    return binarySum(data, low, mid) + binarySum(data, mid+1, high);
  }
}
```

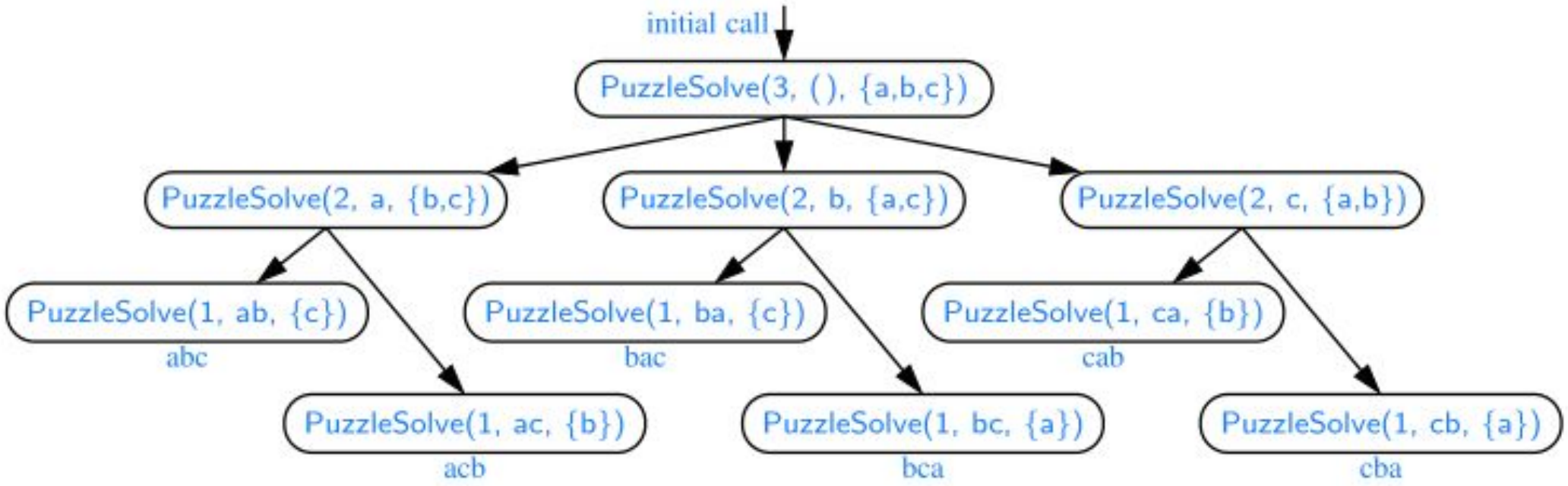
Binary Recursion - binarySum

- To analyze algorithm binarySum, we consider, for simplicity, the case where ***n*** is a power of two
- We label each box with the values of parameters low and high for that call
- The size of the range is divided in half at each recursive call, and so the depth of the recursion is $1+\log_2 n$
- The running **time** of binarySum is ***O(n)***, as there are **2n−1** method calls, each requiring constant time
- binarySum uses ***O(log n)*** amount of additional **space**, which is a big improvement over the **O(n)** space used by the linearSum method



Multiple Recursion

- Generalizing from binary recursion, we define multiple recursion as a process in which a method may make more than two recursive calls
- Disk space usage of a file system is an example of multiple recursion, because the number of recursive calls made during one invocation was equal to the number of entries within a given directory of the file system
- Other example is Generating all permutations of a given set of characters



An Inefficient Recursion for Computing Fibonacci Numbers

- Fibonacci numbers, which can be defined recursively as follows:

$$F0 = 0$$

$$F1 = 1$$

$$Fn = Fn-2 + Fn-1 \quad \text{for } n > 1.$$

- A recursive implementation based directly on this definition results in the method fibonacciBad, which computes a Fibonacci number by making two recursive calls in each non-base case

```
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n-2) + fibonacciBad(n-1);
}
```

An Inefficient Recursion for Computing Fibonacci Numbers

- Unfortunately, such a direct implementation of the Fibonacci formula results in a terribly inefficient method
- Computing the n^{th} Fibonacci number in this way *requires an exponential number of calls* to the method
- Specifically, let c_n denote the number of calls performed in the execution of fibonacciBad(n). Then, we have the following values for the c_n's:
- The *number of calls more than doubles for each two consecutive indices*
- c_4 is more than twice c_2, c_5 is more than twice c_3, c_6 is more than twice c_4, and so on
- Thus, **$c_n > 2^{n/2}$**, which means that fibonacciBad(n) makes a number of calls that is *exponential in n*

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$