# Ch 5 recursion

One way to describe repetition within to achieve repetition is through a process known as recursion.

# 5.1 Illustrative examples

***Recursion***: technique by which a method makes one or + calls to itself during execution, or by which a data structure relies upon smaller instances of the very same type of structure in its representation.

In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks.

Four illustrative examples of the use of recursion: factorial fct, English ruler, Binary search, file system.

## 5.1.1 Factorial Function

The factorial of a positive integer n, denoted n!, is defined as the product of the integers from 1 to n. If n = 0, then n! is defined as 1 by convention.

For any integer n ≥ 0,

n! = 1 if n = 0

n ·(n−1) ·(n−2) ···3 ·2 ·1 if n ≥ 1

The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items.

Recursive def can be formalized as

n! = 1 if n = 0

n ·(n−1)! if n ≥ 1

First, *we have one or more base cases, which refer to fixed values of the function*. The above def has one base case stating that n! = 1 for n = 0.

Second, *we have one or more recursive cases, which define the function in terms of itself*. In the above definition, there is one recursive case, which indicates that n!=n·(n−1)! for n≥1.

### Recursive implementation of the factorial fct

We can use recursion to design a Java implementation of the factorial function.

Repetition is achieved through repeated recursive invocations of the method.

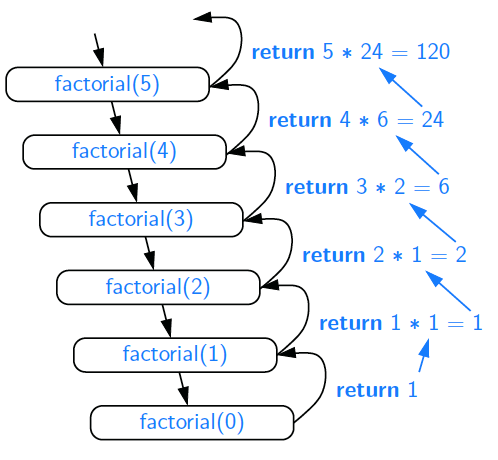
The process is finite because each time method is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made.

We illustrate execution of recursive method using a recursion trace.

Each entry of the trace corresponds to a recursive call.

Each new recursive method call is indicated by a downward arrow to a new invocation.

When the method returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow.



A recursion trace closely mirrors a programming language’s execution of the recursion.

In Java, each time a method (recursive or otherwise) is called, a structure known as an ***activation record*** or ***activation frame*** is created to store information about the progress of that invocation of the method.

This frame stores the parameters and local variables specific to a given call of the method, and information about which command in the body of the method is currently executing.

When execution of a method leads to a nested method call, execution of the former call is suspended and its frame stores the place in the source code at which the flow of control should continue upon return of the nested call. A new frame is then created for the nested method call.

This process is used both in the standard case of one method calling a different method, or in the recursive case where a method invokes itself. The *key point* is to have a separate frame for each active call.

## 5.1.2 Drawing an English Ruler

We denote the length of the tick designating a whole inch as the major tick length. Between the marks for whole inches, the ruler contains a series of minor ticks, placed at intervals of 1/2 inch, 1/4 inch, and so on.

### Recursive Approach to ruler drawing

The English ruler pattern is a simple example of a fractal, that is, a shape that has a self-recursive structure at various levels of magnification.

In general, an interval with a central tick length L ≥ 1 is composed of:

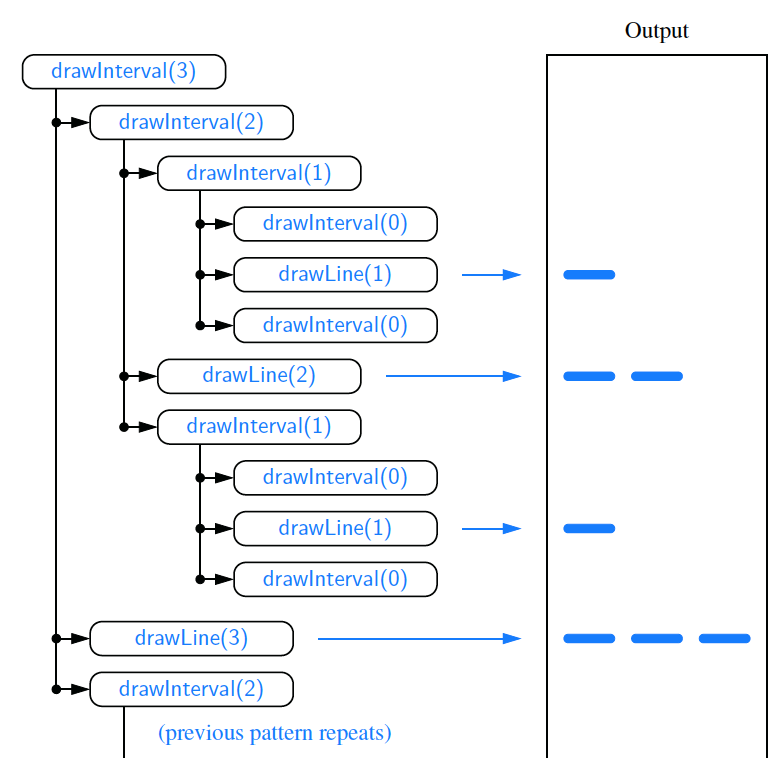
• An interval with a central tick length L−1

• A single tick of length L

• An interval with a central tick length L−1

### Illustrating Ruler Drawing Using a Recursion Trace

The execution of the recursive drawInterval method can be visualized using a recursion trace.



## 5.1.3 Binary Search

Binary search used to efficiently locate a target value within a sorted sequence of n elements stored in an array. When the sequence is ***unsorted***, the standard approach to search for a target value is to use a loop to examine every element. *Algorithm is known as linear search, or sequentialsearch, and it runs in O(n) time*.

When the sequence is sorted and ***indexable***, there is a more efficient algorithm. We call an element of the sequence a ***candidate*** if, at the current stage of the search, we cannot rule out that this item matches the target. We then compare the target value to the median candidate, that is, the element with index:

**mid = ⌊(low+high)/2⌋**

We consider three cases:

• If target == median candidate, then we have found the item we are looking for, and the search terminates successfully.

• If target < median candidate, then we recur on the first half of the sequence, that is, on the interval of indices from low to mid−1.

• If target > median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high.

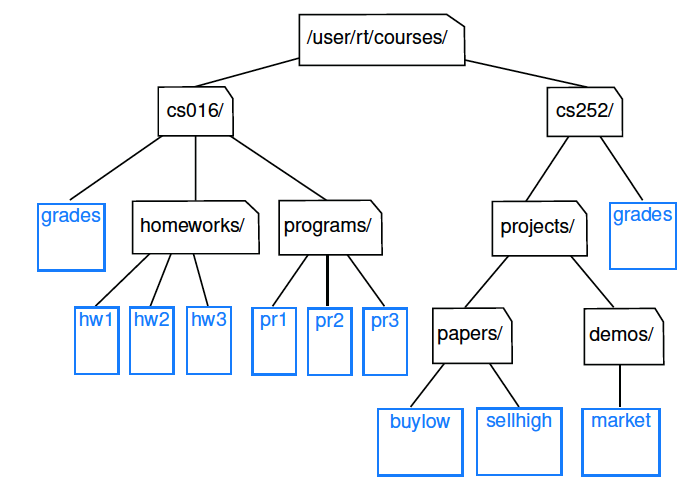
An ***unsuccessful search*** occurs if low > high, as the interval [low,high] is empty. This algorithm is known as ***binary search***. Whereas sequential search runs in O(n) time, the more efficient binary search runs in O(logn) time.

## 5.1.4 File Systems

Modern operating systems define ***file-system directories*** (also called “folders”) in a recursive way.

Namely, a file system consists of a ***top-level directory***, and the contents of this directory consists of files and other directories, which in turn can contain files and other directories, and so on.

The *operating system allows directories to be nested arbitrarily deeply*.



Recursive nature of the file-system representation, it should not come as a surprise that many common behaviours of an operating system, such as copying a directory or deleting a directory, are implemented with recursive algorithms.

We differentiate between the ***immediate*** disk space used by each entry and the ***cumulative*** disk space used by that entry and all nested features.

The cumulative disk space for an entry can be computed with a simple recursive algorithm. It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry.

### The java.io.File Class

To implement recursive algorithm for computing disk usage in Java, *we rely on the java.io.File class*.

*An instance of this class represents an abstract pathname in the operating system and allows for properties of that operating system entry to be queried*. We will rely on the following methods of the class:

• **new File(pathString) or new File(parentFile, childString**)

A new File instance can be constructed either by providing the full path as a string, or by providing an existing File instance that represents a directory and a string that designates the name of a child entry within that directory.

• **file.length()**

Returns the immediate disk usage (measured in bytes) for the operating system entry represented by the File instance (e.g., /user/rt/courses).

**• file.isDirectory( )**

Returns true if the File instance represents a directory; false otherwise.

**• file.list( )**

Returns an array of strings designating the names of all entries within the given directory.

### Recursion Trace

To produce a different form of a recursion trace, we have included an extraneous print statement within our Java implementation. It reports the amount of disk space used by a directory and all contents nested within, and can produce a verbose report.

# 5.2 Analyzing Recursive Algorithms

*We use notations such as big-Oh to summarize the relationship between the number of operations and the input size for a problem*.

With a recursive algorithm, we will account for each operation that is performed based upon the particular activation of the method that manages the flow of control at the time it is executed.

To demonstrate this style of analysis, we revisit the four recursive algorithms : factorial computation, drawing an English ruler, binary search, and computation of the cumulative size of a file system.

### Computing Factorials

It is relatively easy to analyze the efficiency of our method for computing factorials. ***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.

Each individual activation of factorial executes a constant number of operations. Conclude that overall nb of ops for computing factorial(n) is O(n), as there are n+1 activations, each of which accounts for O(1) operations.

### Drawing an English Ruler

Consider fundamental question of how many total lines of output are generated by an initial call to drawInterval(c), where c denotes the center length.

This is a reasonable benchmark for the overall efficiency of the algorithm as each line of output is based upon a call to the drawLine utility, and each recursive call to drawInterval with nonzero parameter makes exactly one direct call to drawLine.

This proof is indicative of a more mathematically rigorous tool, known as a ***recurrence equation***, that can be used to analyze the running time of a recursive algorithm.

### Performing a Binary Search

When considering running time of the binary search algorithm, we observe that a constant number of primitive operations are executed during each recursive call of the binary search method.

***Running time is proportional to number of recursive calls performed***.

We will show that at most ⌊logn⌋+1 recursive calls are made during a binary search of a sequence having n elements, leading to the following claim.

### Computing Disk Space Usage

Our final recursive algorithm is that for computing the overall disk space usage in a specified portion of a file system. To characterize the “problem size” for our analysis, we let n denote the number of file-system entries in the portion of the file system that is considered.

To characterize the cumulative time spent for an initial call to diskUsage, we must analyze the total number of recursive invocations that are made, as well as the number of operations that are executed within those invocations.

# 5.3 Further examples of recursion

maximum number of recursive calls that may be started from within the body of a single activation:

• If a recursive call *starts at most one other*, we call this a *linear recursion*.

• If a recursive call may *start two others*, we call this a *binary recursion*.

• If a recursive call may *start three or more others*, this is *multiple recursion*

## 5.3.1 Linear Recursion

If a recursive method is designed so that each invocation of the body makes at most one new recursive call, this is know as ***linear recursion***.

The code for binary search includes a case analysis, with two branches that lead to a further recursive call, but only one branch is followed during a particular execution of the body.

A consequence of def of linear recursion is that any recursion trace will appear as a single sequence of calls, as we originally portrayed for the factorial method.

Note that the linear recursion terminology *reflects the structure of the recursion trace*, not the asymptotic analysis of the running time.

### Summing the Elements of an Array Recursively

Linear recursion can be a useful tool for processing a sequence. A recursive algorithm for computing the sum of an array of integers based on this intuition is implemented.

For an input of size n, the linearSum algorithm makes n+1 method calls. It 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 algorithm is also O(n).

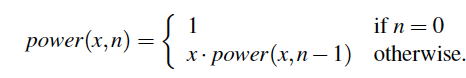
### Reversing a Sequence with Recursion

Next, let us 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. The entire process runs in O(n) time.

### Recursive Algorithms for Computing Powers

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) = xn.

We will consider two different recursive formulations for the problem that lead to algorithms with very different performance. A trivial recursive definition follows from the fact that xn = x ·xn−1 for n > 0.



A recursive call to this version of power(x,n) runs in O(n) time.

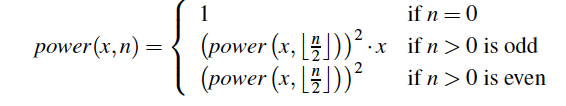
Much faster way to compute the power function using an alternative definition that employs a squaring technique.

Let k=[n/k] denote the floor of the integer division. Consider expression (xk)2.

***When n is even***, [n/k] = n/2 and therefore (xk)2= (xn/2)2. = xn.

***When n is odd***, [n/2]= (n−1)/2 and (xk)2= xn−1, and therefore xn =(xk)2·x.

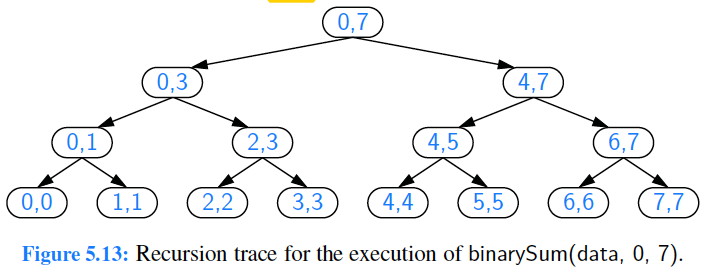
This analysis leads to the following recursive definition:



Total number of operations for computing power(x,n) is O(logn). Because the recursive depth of the improved version is O(log n), its memory usage is O(logn) as well.

## 5.3.2 Binary Recursion

When a method makes two recursive calls, we say that it uses binary recursion. Running time of binarySum is O(n), as there are 2n−1 method calls, each requiring constant time.



## 5.3.3 Multiple Recursion

Define multiple recursion as a process in which a method may make more than two recursive calls.

*Ex*: the following are all instances of what are known as summation puzzles:

pot + pan = bib

dog + cat = pig

boy+ girl = baby

To solve such a puzzle, we need to assign a unique digit (that is, 0,1, . . . ,9) to each letter in the equation, in order to make the equation true.

If nb of possible configurations is not too large can use a computer to simply enumerate all the possibilities and test each one. way.

To keep description general enough to be used with other puzzles, we consider an algorithm that enumerates and tests all k-length sequences, without repetitions, chosen from a given universe U.

Building the sequence of k elements with the following steps:

1. Recursively generating the sequences of k−1 elements

2. Appending to each such sequence an element not already contained in it.

Throughout the execution of the algorithm, we use a set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U.

# 5.4 Designing Recursive Algorithms

An algorithm that uses recursion has following form:

• ***Test for base cases***.

We begin by *testing* for a set of base cases (there should be at least one). These base cases *should be defined so that every possible chain of recursive calls will eventually reach a base case*, and the handling of each base case *should not use recursion*.

• ***Recur***.

If *not a base case*, we *perform one or more recursive calls*. This recursive step may involve a test that decides which of several possible recursive calls to make. We should define each possible recursive call so that it makes progress towards a base case.

### Parameterizing a Recursion

To design recursive algo, it is useful to think of the different ways we might define subproblems that have the same general structure as the original problem.

A *successful recursive design* requires that we redefine original problem to facilitate similar-looking subproblems. Often, this involved reparameterizing signature of the method.

Ex: when performing a binary search in an array, a natural method signature for a caller would appear as binarySearch(data, target).

BUT in 5.1.3 we defined our method with calling signature binarySearch(data, target, low, high), using additional parameters to demarcate subarrays as the recursion proceeds.

This change in parameterization is critical for binary search.

If we wish to provide a cleaner public interface to an algorithm without exposing the user to recursive parameterization, a standard technique is to make recursive version private, and to introduce a cleaner public method.

# 5.5 Recursion Run Amok

In this section:

* examine several cases in which *poorly implemented recursion causes drastic inefficiency*
* discuss strategies for recognizing and avoid such pitfalls.

First, revisiting element uniqueness problem. Can use following recursive formulation to determine if all n elements of sequence are unique.

As base case, when n = 1, elements are trivially unique.

For n ≥ 2, elements are unique if and only if the first n−1 elements are unique,

the last n−1 items are unique, and the first and last elements are different.

Nonrecursive part of each call uses O(1) time, so overall running time will be proportional to total nb of recursive invocations. If n = 1, then the running time of unique3 is O(1), since there are no recursive calls for this case. Those two calls with size n−1 could in turn result in four calls (two each) with a range of size n−2, and thus eight calls with size n−3 and so on.

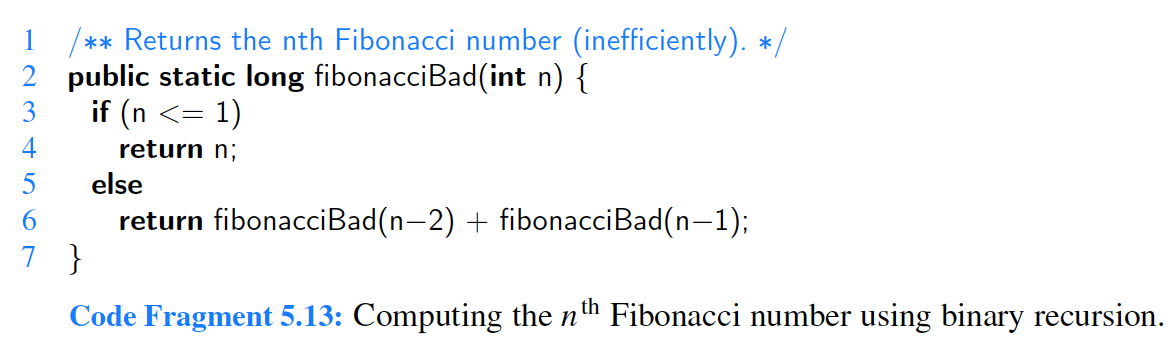
### An Inefficient Recursion for Computing Fibonacci Numbers

introduced process for generating progression of Fibonacci nb, which can be defined recursively:

F0 = 0

F1 = 1

Fn = Fn−2+Fn−1 for n > 1.

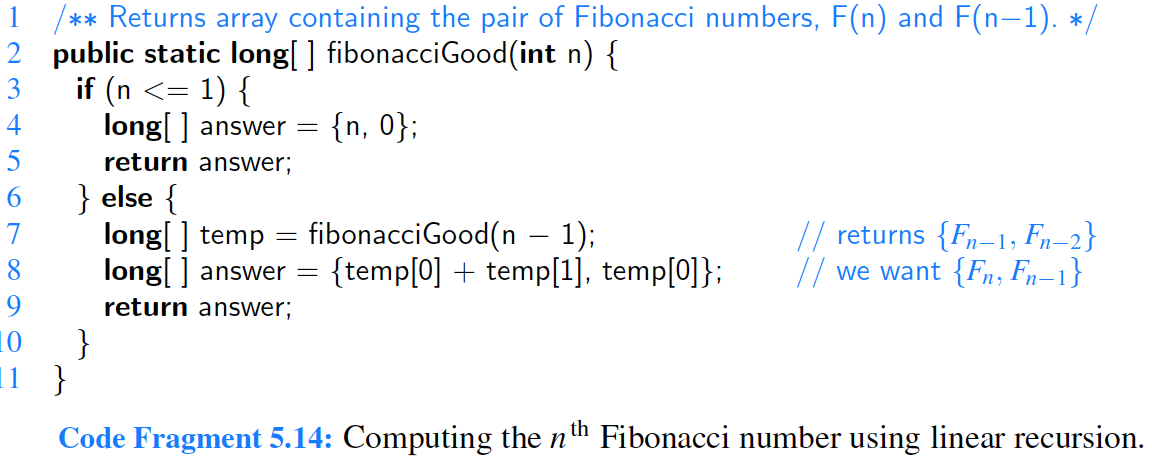


Computing nth Fibonacci nb in this way requires an exponential number of calls to method.

### An Efficient Recursion for Computing Fibonacci Numbers

Rather than having method return a single value, which is nth Fibonacci nb, we define recursive method that returns array with two consecutive Fibonacci nb {Fn,Fn−1}, using the convention F−1 = 0.

Although it seems to be a greater burden to report two consecutive Fibonacci nb instead of one, passing this extra information from one level of recursion to the next makes it much easier to continue the process.



In terms of efficiency, difference between the bad and good recursions for this problem is like night and day. The *fibonacciBad method uses exponential time*. We claim that the execution of method *fibonacciGood(n) runs in O(n) time*.

## 5.5.1 Maximum recursive Depth in Java

Another danger of recursion is ***infinite recursion***.

If each recursive call makes another recursive call, without ever reaching a base case, then we have an infinite series of such calls.

This is a fatal error. An infinite recursion can quickly swamp computing resources.

In particular, when searching a range of two elements, it becomes possible to make a recursive call on the identical range.

To combat against infinite recursions, designers of Java made an intentional decision to limit overall space used to store activation frames for simultaneously active method calls. If this limit is reached, JVM throws a StackOverflowError

# 5.6 Eliminating Tail Recursion

The main benefit of recursive approach to algorithm design is that it allows us to succinctly take advantage of repetitive structure present in many problems.

By making our algorithm description exploit repetitive structure in recursive way, we can often avoid complex case analyses and nested loops. Usefulness of recursion comes at modest cost. In particular, JVM must maintain frames that keep track of the state of each nested call.

Some forms of recursion can be eliminated without any use of auxiliary memory. One such form is known as ***tail recursion***. A recursion is a tail recursion *if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call* (if any) immediately returned by the enclosing recursion. By necessity, a *tail recursion must be a linear recursion*.

*Ex*: return n \* factorial(n−1);

This is not a tail recursion because an additional multiplication is performed after recursive call is completed, and result returned is not the same.

*Tail recursions are special*, as they can be automatically reimplemented nonrecursively by enclosing body in loop for repetition, and replacing recursive call with new parameters by reassignment of existing parameters to those values.