Data structure and algorithm in Python

Recursion

Xiaoping Zhang

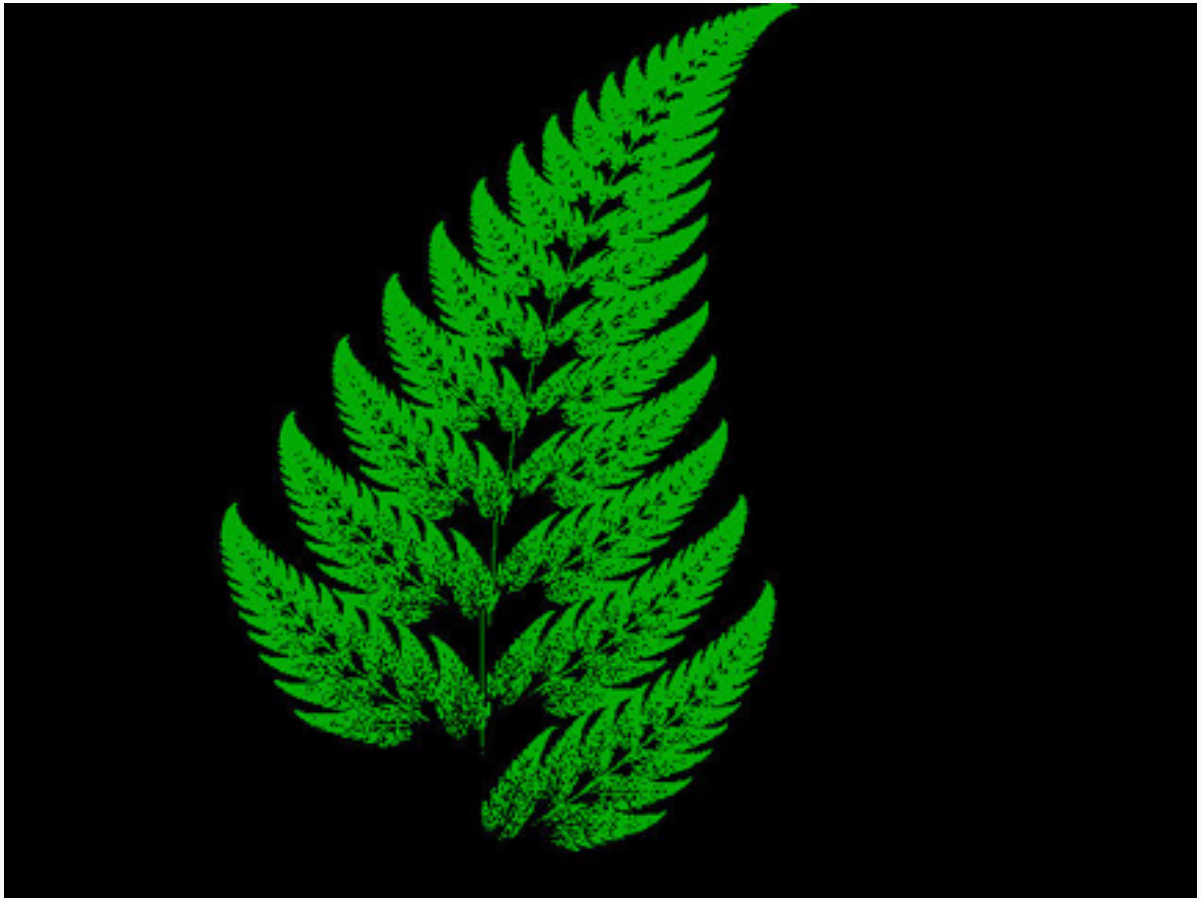
School of Mathematics and Statistics, Wuhan University

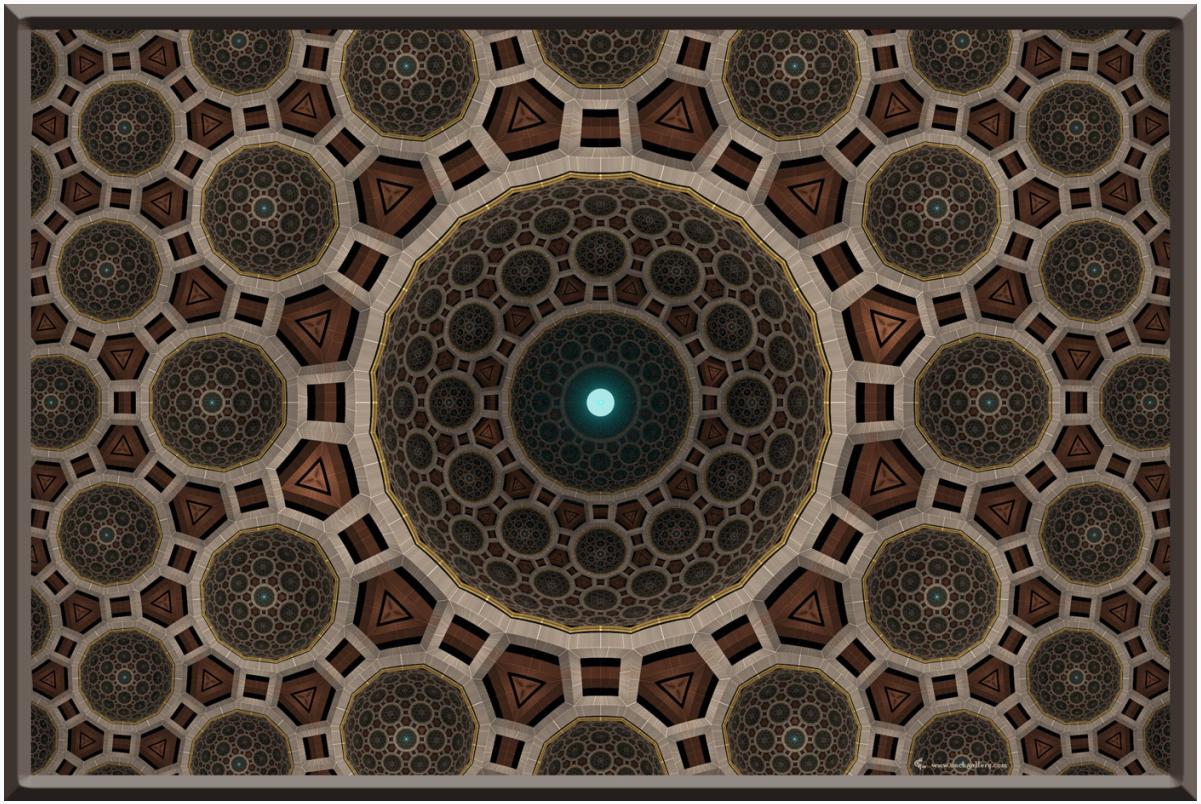
Table of contents

One way to describe repetition within a computer program is the use of loops, such as Python's while-loop and for-loop constructs. An entirely different way to achieve repetition is through a process known as recursion.

Definition : Recusion

Recursion is a technique by which a function makes one or more 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.

Most modern programming languages support functional recursion using the identical mechanism that is used to support traditional forms of function calls. When one invocation of the function make a recursive call, that invocation is suspended until the recursive call completes.

Recursion is an important technique in the study of data structures and algorithms.

Four illustrative examples of the use of recursion:

1. Factorial function
2. English ruler
3. Binary search
4. File system

Illustrative Examples

Illustrative Examples

The Factorial Function

The Factorial Function

Definition : Formal

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1) \cdots 2 \cdot 1, & n \geq 1. \end{cases}$$

Definition : Recursive

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1. \end{cases}$$

Recursive definition

- contains one or more base cases
- contains one or more recursive cases

The Factorial Function

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

The Factorial Function

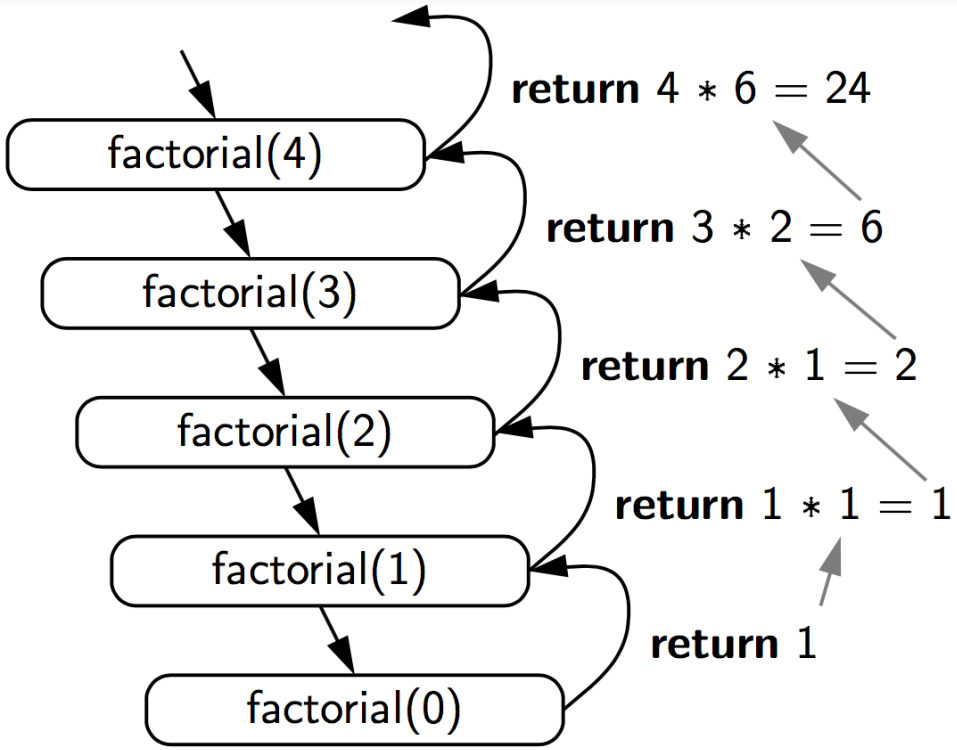


Figure 4.1: A recursion trace for the call factorial(5).

The Factorial Function

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

- In Python, each time a function is called, a structure known as an activation record or frame is created to store information about the progress of that invocation of the function.
- This activation record includes a namespace for storing the function call's parameters and local variables, and information about which command in the body of the function is currently executing.

The Factorial Function

When the execution of a function leads to a nested function call, the execution of the former call is suspended and its activation record stores the place in the source code at which the flow of control should continue upon return of the nested call. This process is used both in the standard case of one function calling a different function, or in the recursive case in which a function invokes itself. The key point is that there is a different activation record for each active call.

The Factorial Function: Algorithm Analysis

- a total of $n+1$ activations;
- each individual activation executes a constant number of operations.

The overall number of operations for computing `factorial(n)` is $O(n)$.

Illustrative Examples

English Ruler

English Ruler

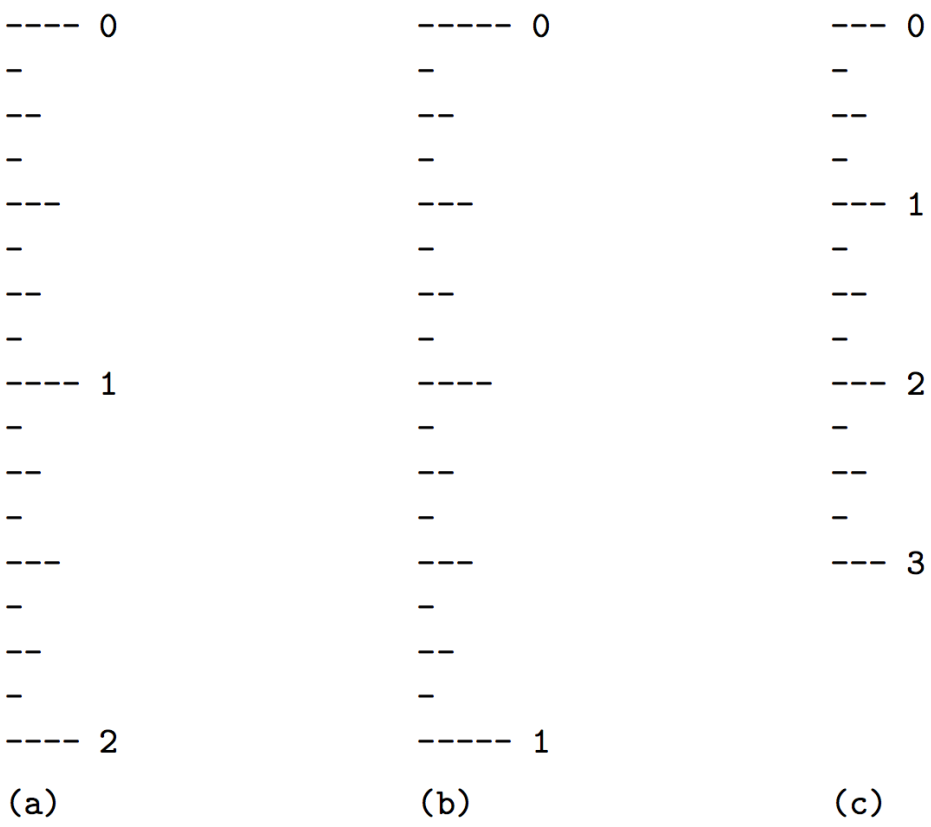


Figure 4.2: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

English Ruler

An interval with a central tick length $L \geq 1$ 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 $L-1$

English Ruler I

```
def draw_line(tick_length, tick_label=''):
  line = '-' * tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)
```

English Ruler I

```
def draw_interval(center_length):
  if center_length > 0:                    # stop
   when length drops to 0
    draw_interval(center_length - 1)       #
    recursively draw top ticks
    draw_line(center_length)               # draw
     center tick
    draw_interval(center_length - 1)       #
    recursively draw bottom ticks
```

English Ruler I

```
def draw_ruler(num_inches, major_length):
  draw_line(major_length, '0')                    # draw
    inch 0 line
  for j in range(1, 1 + num_inches):
    draw_interval(major_length - 1)               # draw
      interior ticks for inch
    draw_line(major_length, str(j))               # draw
      inch j line and label
```

English Ruler

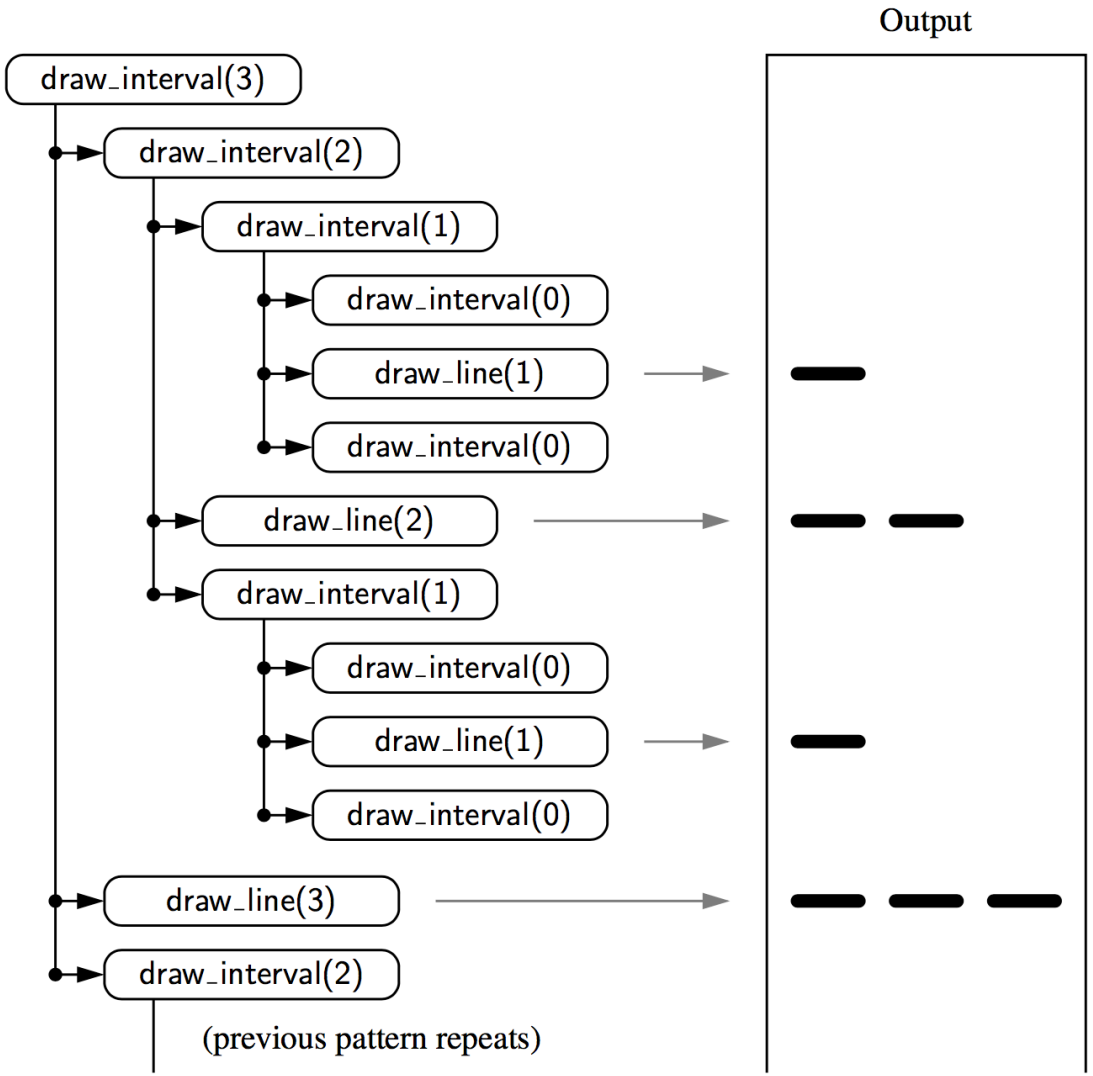


Figure 4.3: A partial recursion trace for the call draw_interval(3). The second pattern of calls for draw_interval(2) is not shown, but it is identical to the first.

English Ruler: Algorithm Analysis

Question

How many total lines of output are generated by an initial call to `draw_interval(c)`? (c denotes the center length).

A call to `draw_interval(c)` for $c > 0$ spawns two calls to `draw_interval(c1)` and a single call to `draw_line`.

Proposition

For $c \geq 0$, a call to `draw_interval(c)` results in precisely $2^c - 1$ lines of output.

Illustrative Examples

Binary Search

Binary Search

Binary Search is used to efficiently locate a target value within a sorted sequence of n elements.

This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order.

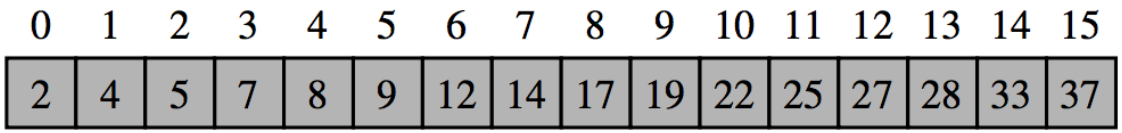


Figure 4.4: Values stored in sorted order within an indexable sequence, such as a Python list. The numbers at top are the indices.

Binary Search

When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set. This is known as the "sequential search" algorithm. This algorithm runs in $O(n)$ time (i.e., linear time) since every element is inspected in the worst case.

Binary Search

Binary Search maintains two parameters, `low` and `high`, such that all the candidate entries have index at least `low` and at most `high`.

- Initially, `low = 0` and `high = n - 1`.
- Then, compare the target value to the median candidate: `data[mid]` with $mid = \lfloor (low + high)/2 \rfloor$
- Consider three cases:
  - If `target == data[mid]`, success;
  - If `target < data[mid]`, recur on the left half of the sequence;
  - If `target < data[mid]`, recur on the right half of the sequence.

An unsuccessful search occurs if `low > high`, as the interval `[low,high]` is empty.

Binary Search I

```
def binary_search(data, target, low, high):
  # interval is empty; no match
  if low > high:
    return False
  else:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      return binary_search(data, target, low,
      mid - 1)
    else:
      return binary_search(data, target, mid +
      1, high)
```

Binary Search

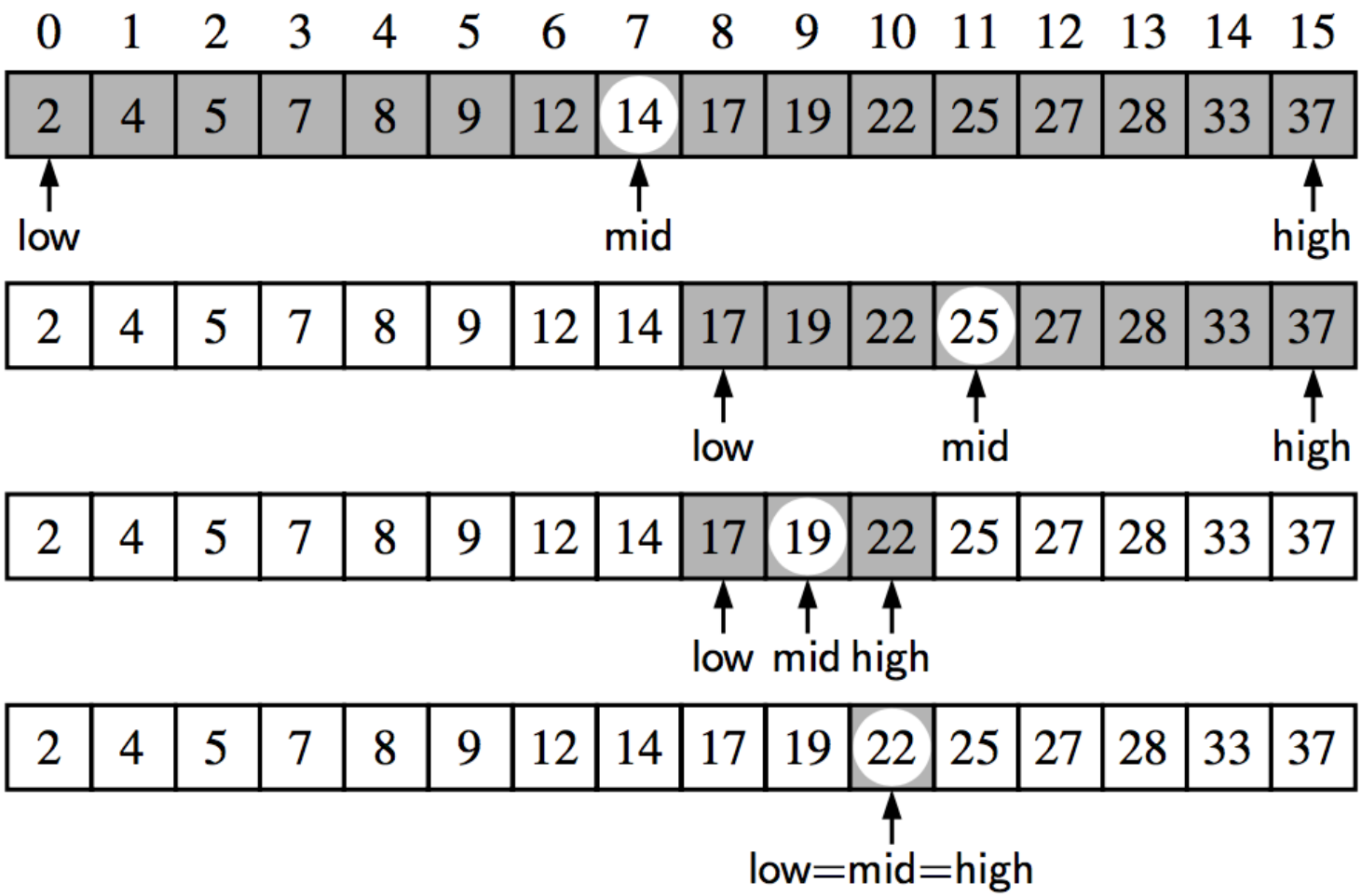


Figure 4.5: Example of a binary search for target value 22.

Binary Search: Algorithm Analysis

Proposition

The binary search algorithm runs in $O(\log n)$ time for a sorted sequence with n elements.

Proof

From the definition of mid, the number of remaining candidates is either

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

or

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

So the maximum number of recursive calls performed is the smallest integer r such that $\frac{n}{2^r} < 1$, i.e., $r > \log n$.

Illustrative Examples

File Systems

File Systems

Modern operating systems define file-system directories 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 deep (as long as there is enough space in memory), although there must necessarily be some base directories that contain only files, not further subdirectories.

File Systems

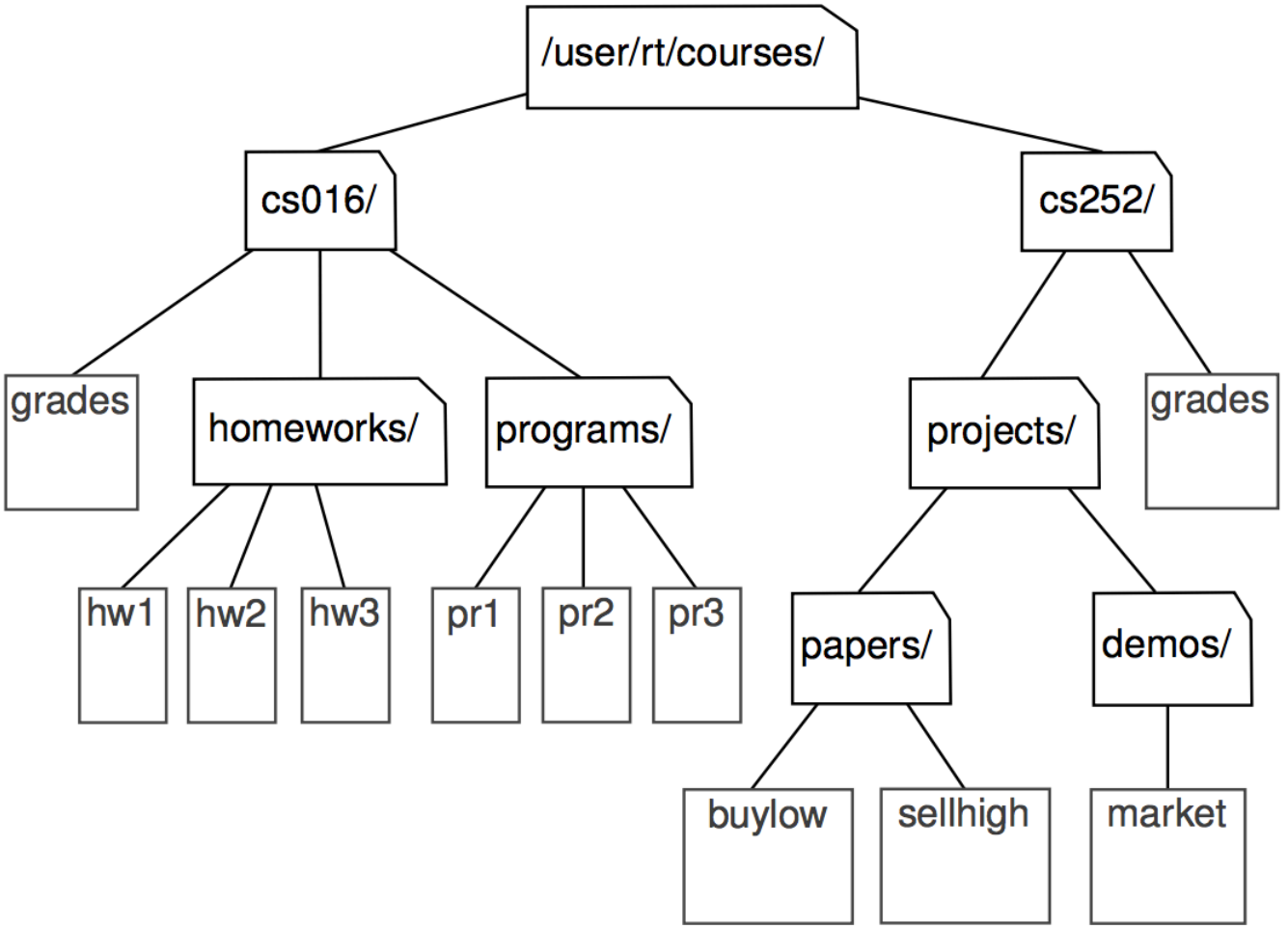


Figure 4.6: A portion of a file system demonstrating a nested organization.

File Systems

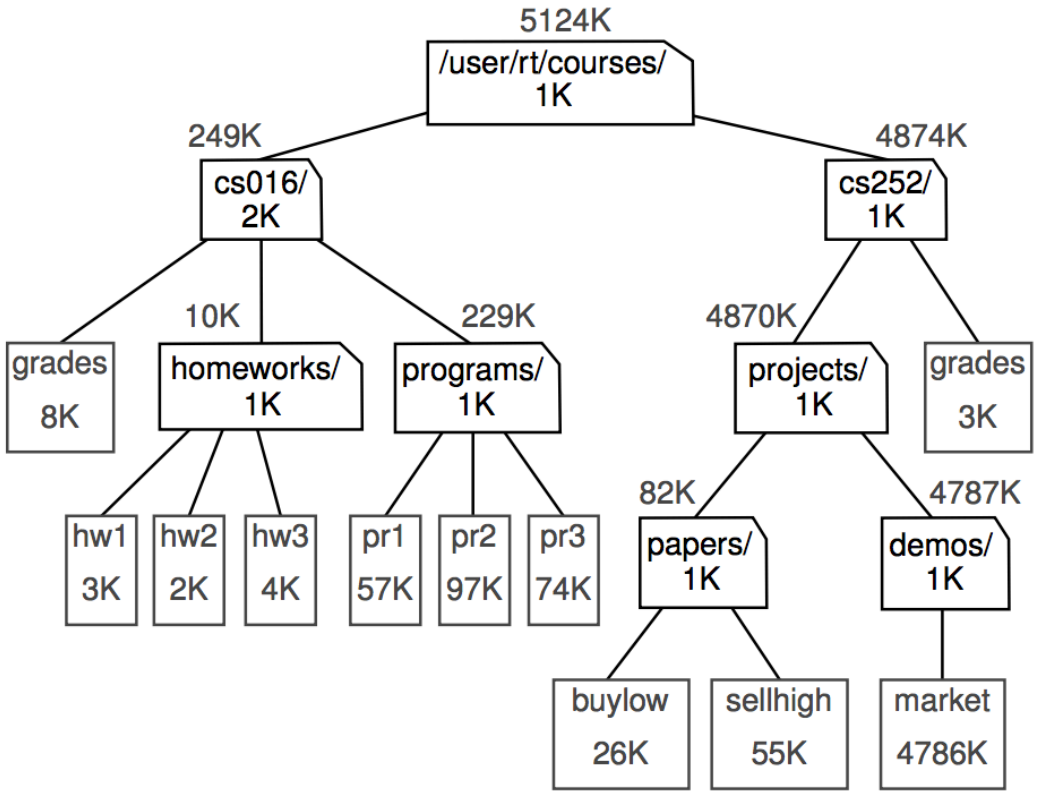


Figure 4.7: The same portion of a file system given in Figure 4.6, but with additional annotations to describe the amount of disk space that is used. Within the icon for each file or directory is the amount of space directly used by that artifact. Above the icon for each directory is an indication of the *cumulative* disk space used by that directory and all its (recursive) contents.

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. (See the cumulative disk space for cs016)

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                          {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)    {recursive call}
    return total
```

Python's os Module

Python's os module, which provides robust tools for interacting with the operating system during the execution of a program.

- `os.path.getsize(path)`
  Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path (e.g., /user/rt/courses).
- `os.path.isdir(path)`
  Return True if entry designated by string path is a directory; False otherwise.
- `os.listdir(path)`
  Return a list of strings that are the names of all entries within a directory designated by string path.
- `os.path.join(path, filename)`
  Compose the path string and filename string using an appropriate operating system separator between the two. Return the string that represents the full path to the file.

File Systems

```
import os

def disk_usage(path):
  total = os.path.getsize(path)
                    # account for direct usage
  if os.path.isdir(path):
                           # if this is a
  directory,
    for filename in os.listdir(path):
                  # then for each child:
      childpath = os.path.join(path, filename)
        # compose full path to child
      total += disk_usage(childpath)
                    # add child's usage to total
  print ('{0:<7}'.format(total), path)
              # descriptive output (optional)
  return total
```

File Systems: Algorithm Analysis

Let n be the number of file-system entries in the portion of the file system.

To characterize the cumulative time spent for an initial call to the disk usage function, we must analyze

- the total number of recursive invocations,
- the number of operations that are executed within those invocations.

Poor Implementation of Recursion

Poor Implementation of Recursion

- Although recursion is a very powerful tool, it can easily be misused in various ways.
- A poorly implemented recursion will cause drastic inefficiency.
- Should learn some strategies for recognizing and avoid such pitfalls.

Poor Implementation of Recursion

Element uniqueness problem

Poor Implementation of Recursion

Question : element uniqueness problem

Determine if all n elements of a sequence are unique.

Poor Implementation of Recursion

Question : element uniqueness problem

Determine if all n elements of a sequence are unique.

Element uniqueness problem

```
def unique1(S):
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False                    # found
        duplicate pair
  return True                           # if we reach
  this, elements were unique
```

Element uniqueness problem

```
def unique2(S):
  temp = sorted(S)                    # create a
  sorted copy of S
  for j in range(1, len(temp)):
    if S[j-1] == S[j]:
      return False                    # found
      duplicate pair
  return True                         # if we reach
  this, elements were unique
```

Poor Implementation of Recursion

- If $n = 1$, the elements are trivially unique.
- For $n \geq 2$, the 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.

Element uniqueness problem

```
def unique3(S, start, stop):
  if stop - start <= 1:
    return True    # at most one item
  elif not unique(S, start, stop-1):
    return False   # first part has duplicate
  elif not unique(S, start+1, stop):
    return False   # second part has duplicate
  else:
    return S[start] != S[stop-1]   # do first and
     last differ?
```

Element uniqueness problem

```
def unique3(S, start, stop):
  if stop - start <= 1:
    return True    # at most one item
  elif not unique(S, start, stop-1):
    return False   # first part has duplicate
  elif not unique(S, start+1, stop):
    return False   # second part has duplicate
  else:
    return S[start] != S[stop-1]   # do first and
     last differ?
```

This is a terribly insufficient use of recursion.

Element uniqueness problem

Let n denote the number of entries – i.e., `n = stop - start`.

- If $n = 1$, the running time of `unique3` is $O(1)$, since there are no recursive calls for this case.
- If $n > 1$, a single call to unique3 for a problem of size n may result in two recursive calls on problems of size $n-1$.

Thus, in the worst case, the total number of function calls is given by

$$1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1,$$

which means the running time of `unique3` is $O(2^n)$.

Poor Implementation of Recursion

Computing Fibonacci Numbers

Computing Fibonacci Numbers

Question : Fibonacci series

$$
\begin{aligned}
F_0 &= 0, \\
F_1 &= 1, \\
F_n &= F_{n-1} + F_{n-2}, \quad n > 1.
\end{aligned}
$$

Computing Fibonacci Numbers: An Inefficient Recursion

```
def bad_fibonacci(n):
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n
    -1)
```

Computing Fibonacci Numbers: An Inefficient Recursion

```
def bad_fibonacci(n):
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n
    -1)
```

Such a direct implementation of the Fibonacci formula results in a terribly inefficient function.

Computing Fibonacci Numbers: An Inefficient Recursion

Let c_n be the number of calls performed in the execution of `bad_fibonacci(n)`, then

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

So,

$$c_n > 2^{n/2},$$

which means that `bad_fibonacci(n)` makes a number of calls that is exponential in n.

Computing Fibonacci Numbers: An Efficient Recursion

In bad recursion formulation, F_n depends on F_{n-2} and F_{n-1}. But notice that after computing F_{n-2}, the call to compute F_{n-1} requires its own recursive call to compute F_{n-2}, as it does not have knowledge of the value of F_{n-2} that was computed at the earlier level of recursion.

Computing Fibonacci Numbers: An Efficient Recursion

```
def good_fibonacci(n):
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n-1)
    return (a+b, a)
```

The execuion of `good_fibonacci(n)` takes $O(n)$ time.

Poor Implementation of Recursion

Maximum Recursive Depth in Python

Maximum Recursive Depth in Python

Another danger in the misuse of recursion is known as 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, not only due to rapid use of the CPU, but because each successive call creates an activation record requiring additional memory.

Maximum Recursive Depth in Python

Example : Infinite recursion

```
def fib(n):
  return fib(n)
```

Maximum Recursive Depth in Python

Example : Infinite recursion

```
def fib(n):
  return fib(n)
```

A programmer should ensure that each recursive call is in some way progressing toward a base case.

Maximum Recursive Depth in Python

To combat against infinite recursions, the designers of Python made an intentional decision to limit the overall number of function activations that can be simultaneously active.

- The precise value of this limit depends upon the Python distribution, but a typical default value is 1000.
- If this limit is reached, the Python interpreter raises a `RuntimeError` with a message, `maximum recursion depth exceeded`.

Maximum Recursive Depth in Python

Fortunately, the Python interpreter can be dynamically reconfigured to change the default recursive limit.

```
import sys
old = sys.getrecursionlimit()
sys.setresursionlimit(1000000)
```

Further Examples of Recursion

Further Examples of Recursion

Linear Recursion

Further Examples of Recursion

Definition : Linear Recursion

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

Further Examples of Recursion

Definition : Linear Recursion

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

Example : Linear Recursion

- factorial function
- good fibnacci function
- binary search algorithm

  It includes a case analysis with two branches that lead to recursive calls, but only one of those calls can be reached during a particular execution of the body.

Linear Recursion

A consequence of the definition of linear recursion is that any recursion trace will appear as a single sequence of calls.

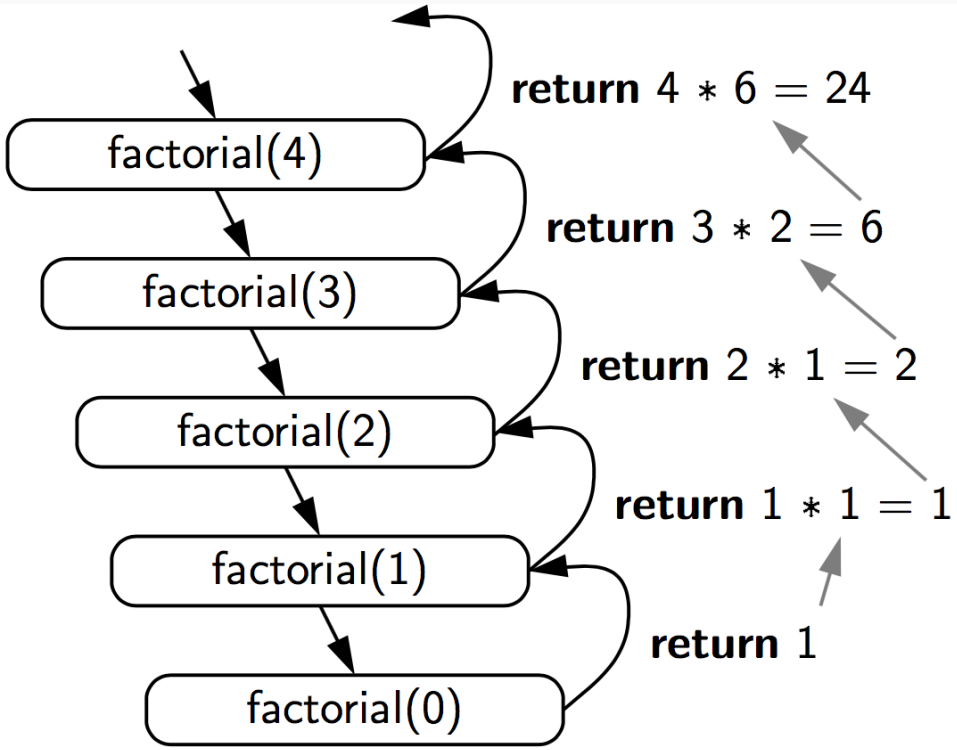


Figure 4.1: A recursion trace for the call factorial(5).

Linear Recursion: Summing the Elements of a Sequence

Example : Summing the Elements of a Sequence Recursively

Compute the sum of a sequence S of n integers.

Let s be the sum of all n integers in S, then

- if $n = 0$, $s = 0$
- if $n > 0$, s equals the sum of the first $n-1$ integers in S plus the last element in S.

Linear Recursion: Summing the Elements of a Sequence

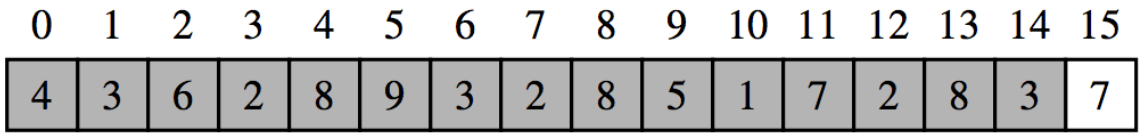


Figure 4.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n - 1$.

Linear Recursion: Summing the Elements of a Sequence

```
def linear_sum(S, n):
  if n == 0:
    return 0
  else:
    return linear_sum(S, n-1) + S[n-1]
```

Linear Recursion: Summing the Elements of a Sequence

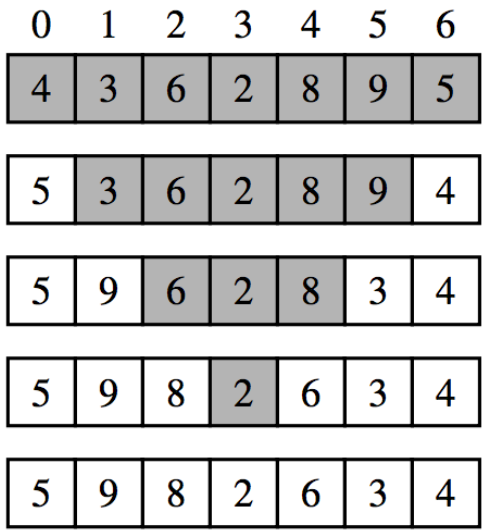


Figure 4.11: A trace of the recursion for reversing a sequence. The shaded portion has yet to be reversed.

Algorithm Analysis

- Run time
  - $n+1$ function calls
  - each call use $O(1)$ operations

  so it takes $O(n)$ time;
- Memory space
  - because it uses a constant amount of memory space for each of the $n+1$ activation records, so its memory usage is $O(n)$.

Linear Recursion: Reversing a Sequence

Example : Reversing a Sequence with Recursion

Reverse the n elements of a sequence S, so that the first element becomes the last, the second element becomes second to the last, and so on.

Linear Recursion: Reversing a Sequence

Example : Reversing a Sequence with Recursion

Reverse the n elements of a sequence S, so that the first element becomes the last, the second element becomes second to the last, and so on.

The reversal of a sequence can be achieved by swapping the first and last elements and then recursively reversing the remaining elements.

Linear Recursion: Reversing a Sequence

```
def reverse(S, start, stop):
  if start < stop - 1:                          #
    if at least 2 elements:
    S[start], S[stop-1] = S[stop-1], S[start]   #
      swap first and last
    reverse(S, start+1, stop-1)                 #
      recur on rest
```

Linear Recursion: Reversing a Sequence

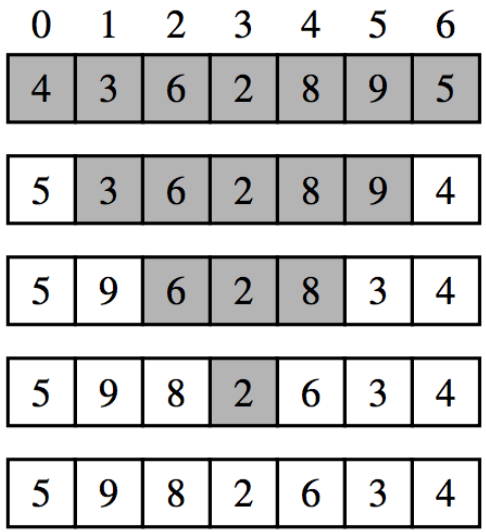


Figure 4.11: A trace of the recursion for reversing a sequence. The shaded portion has yet to be reversed.

Linear Recursion: Reversing a Sequence

- There are two implicit base case scenarios:
  - if `start == stop`, the range is empty;
  - if `start == stop-1`, the range has only one element.

  In either of these cases, there is no need for action.
- When invoking recursion, we are guaranteed to make progress towards a base case, as the difference `stop - start` decreases by two with each call.

Linear Recursion: Reversing a Sequence

- There are two implicit base case scenarios:
  - if `start == stop`, the range is empty;
  - if `start == stop-1`, the range has only one element.

  In either of these cases, there is no need for action.
- When invoking recursion, we are guaranteed to make progress towards a base case, as the difference `stop - start` decreases by two with each call.

The above argument implies that the recursive algorithm is guaranteed to terminate after a total of $1 + \lfloor \frac{n}{2} \rfloor$ recursive calls. Since each call involves a constant amount of work, the entire process runs in $O(n)$ time.

Linear Recursion: Computing Powers

Example : Recursive Algorithms for Computing Powers

Raising a number x to an arbitrary nonnegative integer n, — i.e., computing the power function $power(x, n) = x^n$.

Linear Recursion: Computing Powers

Example : Recursive Algorithms for Computing Powers

Raising a number x to an arbitrary nonnegative integer n, — i.e., computing the power function $power(x, n) = x^n$.

We will consider two different recursive formulations for the problem that lead to algorithms with very different performance.

Linear Recursion: Computing Powers

Definition

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

Linear Recursion: Computing Powers

Definition

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

```
def power(x, n):
  if n == 0:
    return 1
  else:
    return x * power(x, n-1)
```

Linear Recursion: Computing Powers

Definition

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

```
def power(x, n):
  if n == 0:
    return 1
  else:
    return x * power(x, n-1)
```

A recursive call to this version of `power(x, n)` runs in $O(n)$ time. Its recursion trace has structure very similar to that of `factorial(n)`.

Linear Recursion: Computing Powers

Definition

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \text{if } n > 0 \text{ is odd} \\ power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Linear Recursion: Computing Powers

```
def power(x, n):
  if n == 0:
    return 1
  else:
    partial = power(x, n // 2)           # rely
    on truncated division
    result = partial * partial
    if n % 2 == 1:                       # if n
    odd, include extra factor of x
      result *= x
    return result
```

Linear Recursion: Computing Powers

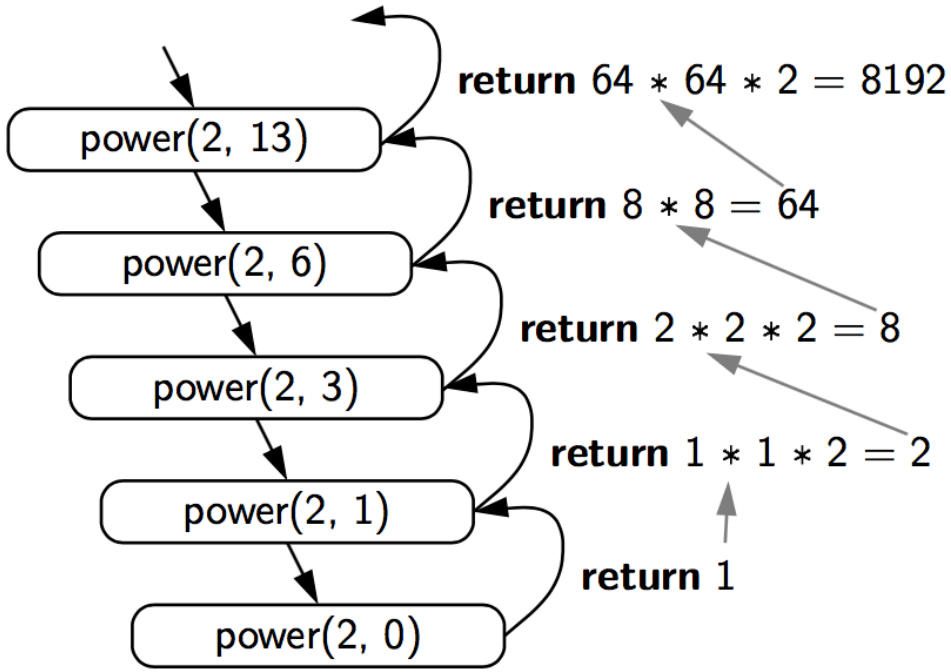


Figure 4.12: Recursion trace for an execution of power(2, 13).

Algorithm Analysis

- Run time
  - $O(\log n)$ recursive calls
  - each individual activation of the function uses $O(1)$ operations

  So the total number of operations is $O(\log n)$
- Memory usage
  - Since the recursive depth is $O(\log n)$, its memory usages is $O(\log n)$ as well.

Further Examples of Recursion

Binary Recursion

Binary Recursion

Definition : Binary Recursion

When a function makes two recursive calls, we say that it uses binary recursion.

Example : Binary Recursion

- English ruler
- Bad fibonacci function

Binary Recursion

Example : Sum of elements

Sum the n elements of a sequence S of numbers.

Binary Recursion

Example : Sum of elements

Sum the n elements of a sequence S of numbers.

- Computing the sum of one or zero elements is trivial.
- With two or more elements, we can recursively compute the sum of the first half, and the sum of the second half, and add these sums together.

Binary Recursion

```
def binary_sum(S, start, stop):
  if start >= stop:                    # zero
  elements in slice
    return 0
  elif start == stop-1:                # one
  element in slice
    return S[start]
  else:                                # two
  or more elements in slice
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) +
    binary_sum(S, mid, stop)
```

Binary Recursion

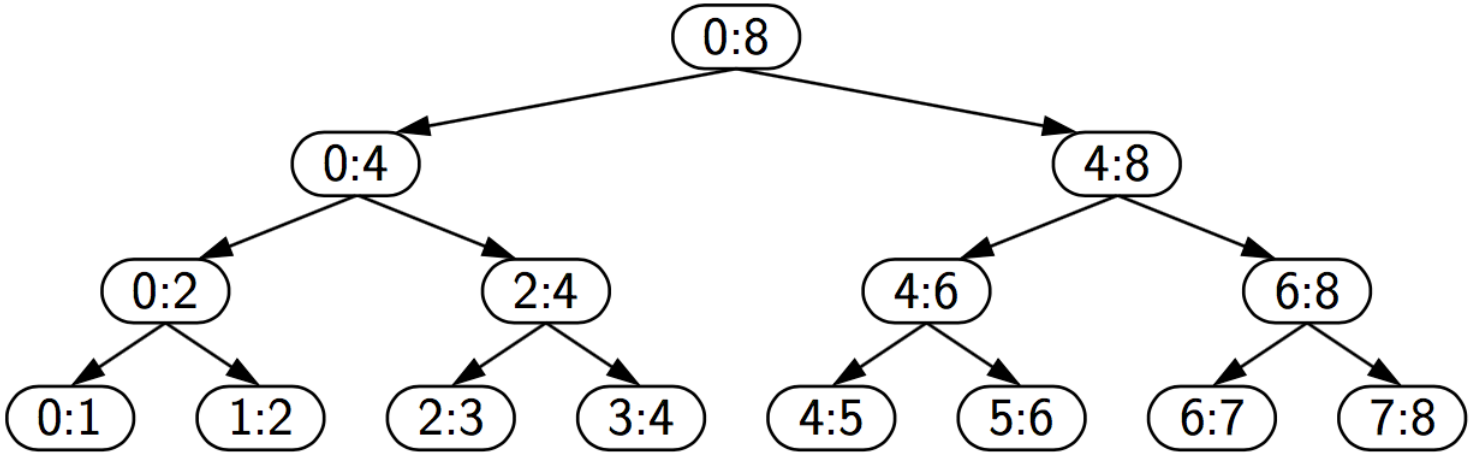


Figure 4.13: Recursion trace for the execution of binary_sum(0, 8).

Algorithm Analysis

For simplicity, consider the case where n is a power of two.

- Running time
  - $2n-1$ functions calls,
  - each call requires $O(1)$ operations,

  so the running time is $O(n)$.
- Memory usage
  - The size of the range is divided in half at each recursive call, and thus the depth of the recursion is $1+\log_2 n$,

  so its memory usage is $O(\log n)$, which is a big improvement over the $O(n)$ space used by the `linear_sum` function.

Further Examples of Recursion

Multiple Recursion

Multiple Recursion

Definition : Multiple recursion

Multiple recursion is a process in which a function may make more than two recursive calls.

Example : Multiple recursion

- Disk space usage of a file system

  the number of recursive calls made during one invocation equals the number of entries within a given directory of the file system.

Designing Recursive Algorithms

Designing Recursive Algorithms

In general, an algorithm that uses recursion typically has the 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 recur- sive 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 a recursive algorithm for a given problem,

- it is useful to think of the different ways we might define subproblems that have the same general structure as the original problem.
- If one has difficulty finding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be defined.

Parameterizing a Recursion

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

Example : Binary search

- A natural function signature for a caller would appear as `binary_search(data, target)`
- In recursive version, we use `binary_search(data, target, low, high)`, using the additional parameters to demarcate sublists.

Parameterizing a Recursion

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

Example : Binary search

- A natural function signature for a caller would appear as `binary_search(data, target)`
- In recursive version, we use `binary_search(data, target, low, high)`, using the additional parameters to demarcate sublists.

If we had insisted on `binary_search(data, target)`, the only way to invoke a search on half the list would have been to make a new list instance with only those elements to send as the first parameter. However, making a copy of half the list would already take $O(n)$ time, negating the whole benefit of the binary search algorithm.

Parameterizing a Recursion

If we wished to provide a cleaner public interface to an algorithm like binary search, without bothering a user with the extra parameters, a standard technique is to make one function for public use with the cleaner interface, such as `binary_search(data, target)`, and then having its body invoke a nonpublic utility function having the desired recursive parameters.

Parameterizing a Recursion

```
def binary_search_recur(data, target, low, high)
:
  if low > high:
    return False
  else:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      return binary_search_recur(data, target,
      low, mid - 1)
    else:
      return binary_search_recur(data, target,
      mid + 1, high)
```

Parameterizing a Recursion

```
def binary_search(data, target):
  return binary_search_recur(data, target, 0,
  len(data))
```

Eliminating Tail Recursion

Eliminating Tail Recursion

The main benefit of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems. By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.

Eliminating Tail Recursion

However, the usefulness of recursion comes at a modest cost. In particular, the Python interpreter must maintain activation records that keep track of the state of each nested call. When computer memory is at a premium, it is useful in some cases to be able to derive nonrecursive algorithms from recursive ones.

Eliminating Tail Recursion

In general, we can use the stack data structure, to convert a recursive algorithm into a nonrecursive algorithm by managing the nesting of the recursive structure ourselves, rather than relying on the interpreter to do so. Although this only shifts the memory usage from the interpreter to our stack, we may be able to reduce the memory usage by storing only the minimal information necessary.

Eliminating Tail Recursion

Some forms of recursion can be eliminated without any use of axillary memory. A notable such form is known as tail recursion.

Eliminating Tail Recursion

Some forms of recursion can be eliminated without any use of axillary memory. A notable such form is known as tail recursion.

Definition : 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.

Eliminating Tail Recursion

Some forms of recursion can be eliminated without any use of axillary memory. A notable such form is known as tail recursion.

Definition : 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.

Eliminating Tail Recursion

Example

- The functions `binary_search` and `reverse` are tail recursions.
- The functions `factorial`, `linear_sum` and `good_fibonacci` are not tail functions.

Eliminating Tail Recursion

Any tail recursion can be reimplemented nonrecursively by enclosing the body in a loop for repetition, and replacing a recursive call with new parameters by a reassignment of the existing parameters to those values.

Eliminating Tail Recursion

```
def binary_search_iterative(data, target):
  low = 0
  high = len(data)-1
  while low <= high:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      high = mid - 1
    else:
      low = mid + 1
  return False
```

Eliminating Tail Recursion

```
def reverse_iterative(S):
  start, stop = 0, len(S)
  while start < stop - 1:
    S[start], S[stop-1] = S[stop-1], S[start]   #
     swap first and last
    start, stop = start + 1, stop - 1           #
     narrow the range
```