

Scientific Programming

Lecture A06 – Recursion

Andrea Passerini

Università degli Studi di Trento

2025/05/28

Acknowledgments: Alberto Montresor

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



Table of contents

- 1 Introduction
- 2 Hanoi's Tower
- 3 Binary search
- 4 A non-classical problem

Recursion

“Of all ideas I have introduced to children, recursion stands out as the one idea that is particularly able to evoke an excited response.”

— Seymour Papert, Mindstorms

Goal of this lecture

- To learn how to formulate programs **recursively**
- To understand and apply the **three laws of recursion**
- To understand how recursion is handled by a computer system
- To understand that complex problems that may otherwise be difficult to solve, may be solved by splitting them in sub-problems
- To understand that sometimes, a recursive approach may lead to more efficient algorithms

Recursion

Definition

Recursion is the process a function goes through when one of the steps of the function involves invoking the function itself, on a smaller input. A function that goes through recursion is said to be **recursive**.

Recursion involves a function that calls itself

- Several mathematical functions **are defined** recursively
- Several problems **can be defined** recursively
- Some problems **may be solved more efficiently** with a recursive approach

Example: Factorial numbers

$$n! = \begin{cases} 1 & n = 1 \\ n \cdot (n - 1)! & n > 1 \end{cases}$$

```
def fact(n):  
    if n <= 1:  
        res = 1  
    else:  
        res = n * fact(n-1)  
    return res
```

Example: Factorial numbers

```
n = 5  
res = ?
```

Example: Factorial numbers

n =	4
res =	?

n =	5
res =	?

Example: Factorial numbers

n =	3
res =	?
n =	4
res =	?
n =	5
res =	?

Example: Factorial numbers

n =	2
res =	?
n =	3
res =	?
n =	4
res =	?
n =	5
res =	?

Example: Factorial numbers

n =	1
res =	1

n =	2
res =	?

n =	3
res =	?

n =	4
res =	?

n =	5
res =	?

Example: Factorial numbers

n =	2
res =	2

n =	3
res =	?

n =	4
res =	?

n =	5
res =	?

Example: Factorial numbers

n =	3
res =	6

n =	4
res =	?

n =	5
res =	?

Example: Factorial numbers

n = 4
res = 24

n = 5
res = ?

Example: Factorial numbers

```
n = 5  
res = 120
```

Example: Factorial numbers

```
def fact(n):  
    print("Start", n)  
    if n <= 1:  
        res = 1  
    else:  
        res = n*fact(n-1)  
    print("End", n, res)  
    return res  
  
print(fact(5))
```

Example: Factorial numbers

<code>def fact(n):</code>	Start 5
<code>print("Start", n)</code>	Start 4
<code>if n <= 1:</code>	Start 3
<code>res = 1</code>	Start 2
<code>else:</code>	Start 1
<code>res = n*fact(n-1)</code>	End 1 1
<code>print("End", n, res)</code>	End 2 2
<code>return res</code>	End 3 6
	End 4 24
<code>print(fact(5))</code>	End 5 120

The three laws of recursion

All recursive algorithms must obey three important laws

- A recursive algorithm must have a base case
- A recursive algorithm must call itself, recursively
- A recursive algorithm must move toward the base case

What happens with this code?

```
def fact(n):  
    return n*fact(n-1)
```

Divide-et-impera

Three phases

- **Divide:** Break the problem in smaller and independent sub-problems
- **Impera:** Solve the sub-problems recursively
- **Combine:** "merge" the solutions of subproblems

There is not a unique recipe for divide-et-impera

- A creative effort is required

Minimum – Recursive version 1

```
def minrec(A, i):  
    if i==0:  
        res = A[0]  
    else:  
        res = min(minrec(A, i-1), A[i])  
    return res
```

```
def mymin(A):  
    return minrec(A, len(A)-1)
```

- `minrec()` returns the minimum of the elements between 0 and i , both included.
- `mymin()` is a **wrapper** to hide the recursion from the caller

Minimum – Recursive version 2

```
def minrec(A, i, j):  
    if i==j:  
        res = A[i]  
    else:  
        m = (i+j) // 2  
        res = min(minrec(A, i, m), minrec(A,m+1,j))  
    return res
```

```
def mymin(A):  
    return minrec(A, 0, len(A)-1)
```

- `minrec()` returns the minimum of the elements between i and j , both included.
- `mymin()` is a **wrapper** to hide the recursion from the caller

Minimum – Recursive version 2 - Debug

```

def minrec(A, i, j):
    print("Start", i, j)
    if i==j:
        res = A[i]
    else:
        m = (i+j) // 2
        res = min(minrec(A, i, m),
                 minrec(A, m+1, j))
        print("COMPARE")
    print("End", i, j)
    return res

def mymin(A):
    return minrec(A, 0, len(A)-1)

L = [2, 4, 1, 3, 6, 8, 9, 12]
m = mymin(L)

```

Start 0 7

Start 0 3

Start 0 1

Start 0 0

End 0 0

Start 1 1

End 1 1

COMPARE

End 0 1

Start 2 3

Start 2 2

End 2 2

Start 3 3

End 3 3

COMPARE

End 2 3

COMPARE

End 0 3

Start 4 7

Start 4 5

Start 4 4

End 4 4

Start 5 5

End 5 5

COMPARE

End 4 5

Start 6 7

Start 6 6

End 6 6

Start 7 7

End 7 7

COMPARE

End 6 7

COMPARE

End 4 7

COMPARE

End 0 7

So far, you probably don't see why recursion is good...

Factorial can be defined recursively, but also "iteratively":

$$n! = \prod_{i=1}^n i$$

```
def fact(n):  
    res = 1  
    for k in range(1, n + 1):  
        res = res * k  
    return res
```

So far, you probably don't see why recursion is good...

Minimum can be defined recursively, but also "iteratively"

```
def mymin(A):  
    res = A[0]  
    for x in A:  
        res = min(res,x)  
    return res
```

So far, you probably don't see why recursion is good...

Sometimes there is no advantage in performance for the recursive versions

- Both versions of `fact` require $n - 1$ products,
- Both versions of `mymin` require $n - 1$ comparisons, where n is the number of items in the input
- Note: Executing the recursive invocations is more costly than executing the iterations.

Version	Time (ms)
Recursive V2	645.60
Iterative	195.03
<code>min()</code> Python	17.88

Cost of `min`(10^6 integers)

So far, you probably don't see why recursion is good...

Recursion may even be dangerous

```
def minrec(L):  
    if len(L) == 1:  
        return L[0]  
    else:  
        return min(L[0], minrec(L[1:]))
```

RecursionError: maximum recursion depth exceeded in comparison

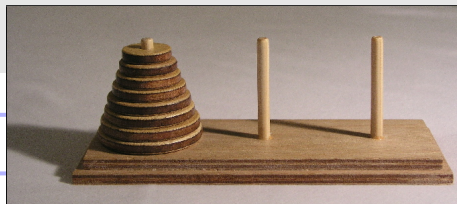
Why recursion?

- Recursion may help in solving problems that are very difficult to attack otherwise
- Recursion may lead to much more efficient algorithms, at least for very large input size

Table of contents

- 1 Introduction
- 2 Hanoi's Tower**
- 3 Binary search
- 4 A non-classical problem

Hanoi's tower



Mathematical game

- Three pins
- n disks with different sizes
- Initially, all the disks are stacked in decreasing size order (from bottom to top) on the left pin

Goal of the game

- Stack all the disks on the right pin in decreasing size order (from bottom to top)
- Never put a larger disk on top of a smaller disk
- You can move one disk at each step
- You can use the middle pin to as support

Hanoi's tower

```
def hanoi(n, src, dst, mid):  
    if n == 1:  
        print(src, "-->", dst)  
    else:  
        hanoi(n-1, src, mid, dst)  
        print(src, "-->", dst)  
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



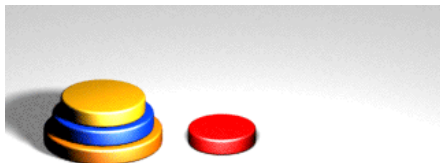
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



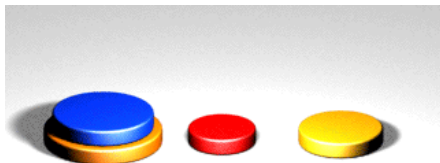
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



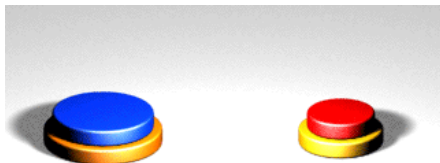
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



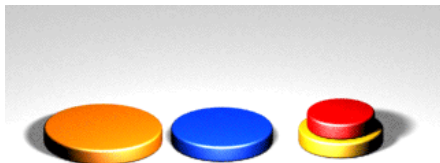
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



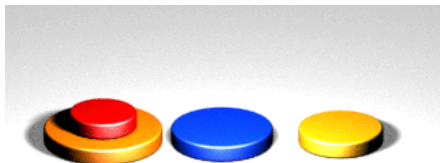
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



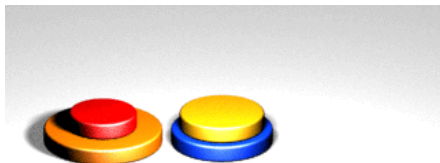
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



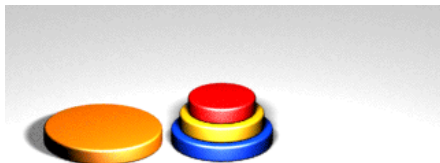
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



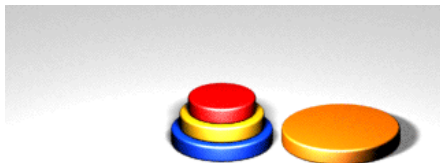
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



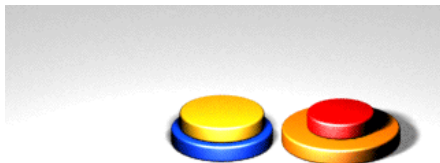
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):  
    if n == 1:  
        print(src, "-->", dst)  
    else:  
        hanoi(n-1, src, mid, dst)  
        print(src, "-->", dst)  
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



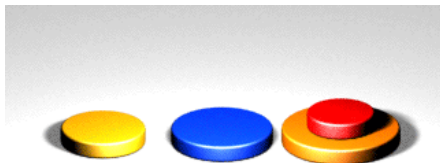
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



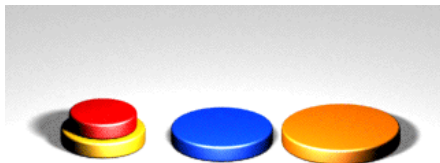
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



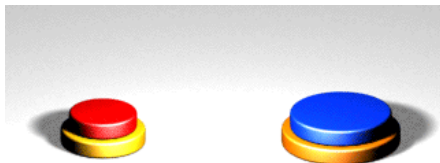
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



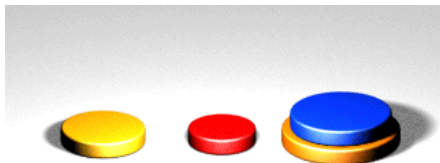
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



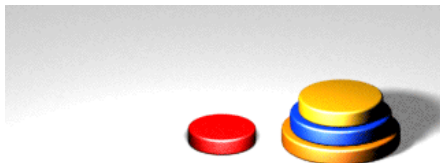
https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):
    if n == 1:
        print(src, "-->", dst)
    else:
        hanoi(n-1, src, mid, dst)
        print(src, "-->", dst)
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower

```
def hanoi(n, src, dst, mid):  
    if n == 1:  
        print(src, "-->", dst)  
    else:  
        hanoi(n-1, src, mid, dst)  
        print(src, "-->", dst)  
        hanoi(n-1, mid, dst, src)
```

Divide-et-impera

- $n - 1$ disks from *src* to *middle*
- 1 disks from *src* to *dest*
- $n - 1$ disks from *middle* to *dest*



https://it.wikipedia.org/wiki/Torre_di_Hanoi#/media/File:Tower_of_Hanoi_4.gif

Hanoi's tower - Bonus version

```

def hanoi(n, src, mid, dst):
    if n == 1:
        dst.append(src.pop())
        print(rsrc, rmid, rdst)
    else:
        hanoi(n-1, src, dst, mid)
        dst.append(src.pop())
        print(rsrc, rmid, rdst)
        hanoi(n-1, mid, src, dst)

rsrc = [4,3,2,1]
rmid = []
rdst = []
print(rsrc, rmid, rdst)
hanoi(len(rsrc), rsrc, rmid, rdst)

```

```

[4, 3, 2, 1] [] []
[4, 3, 2] [1] []
[4, 3] [1] [2]
[4, 3] [] [2, 1]
[4] [3] [2, 1]
[4, 1] [3] [2]
[4, 1] [3, 2] []
[4] [3, 2, 1] []
[] [3, 2, 1] [4]
[] [3, 2] [4, 1]
[2] [3] [4, 1]
[2, 1] [3] [4]
[2, 1] [] [4, 3]
[2] [1] [4, 3]
[] [1] [4, 3, 2]
[] [] [4, 3, 2, 1]

```

Hanoi's tower - Comments

- The number of moves that are performed by this algorithm is equal to $2^n - 1$
- This number is **optimal**: you cannot solve this problem in a smaller number of moves
- While there exist iterative (non-recursive) solutions, none of them is as **clear** as the one presented here.

Table of contents

- 1 Introduction
- 2 Hanoi's Tower
- 3 Binary search**
- 4 A non-classical problem

Search: problem definition

Search over a sorted list

Let $S = s_0, s_1, \dots, s_{n-1}$ be a list of distinct, sorted numbers, i.e. $s_0 < s_1 < \dots < s_{n-1}$.

Searching the position of value v in S corresponds to returning the index i such that $0 \leq i < n$, if v is contained at position i , -1 otherwise.

$$\text{index}(S, v) = \begin{cases} i & \exists i \in \{0, \dots, n-1\} : S_i = v \\ -1 & \text{otherwise} \end{cases}$$

First version – Iterative

```
def index(L, v):  
    for i in range(len(L)):  
        if L[i] == v:  
            return i  
    return -1
```

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"

1	5	12	15	20	23	32
---	---	----	----	----	----	----

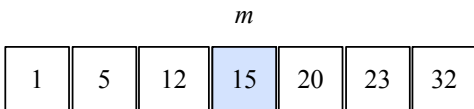
21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"



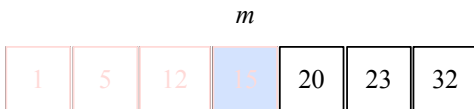
21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"



21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"

1	5	12	15	20	23	32
---	---	----	----	----	----	----

21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"



21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] > v$, look in the "right part"



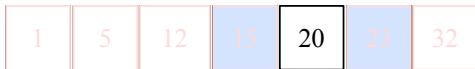
21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"



21?

A better solution - Binary (dichotomic) search

A more efficient solution

Let's consider the median m element of the list

- If $L[m] = v$, the looked-up element has been found
- If $v < L[m]$, look in the "left part"
- If $L[m] < v$, look in the "right part"



21?

Second version – Recursive

```

def index_rec(L, i, j, v):
    print("L[" + str(i) + ":", str(j) + "]", sep="")
    if (j < i):
        return -1
    else:
        m = (i+j) // 2
        if L[m] == v:
            return m
        elif L[m] < v:
            return index_rec(L, m+1, j, v)
        else:
            return index_rec(L, i, m-1, v)

L = list(range(1000))
print(index_rec(L, 0, len(L)-1, 1000))

```

L[0:999]
 L[500:999]
 L[750:999]
 L[875:999]
 L[938:999]
 L[969:999]
 L[985:999]
 L[993:999]
 L[997:999]
 L[999:999]
 L[1000:999]
 -1

Performance evaluation

Cost of execution of (i) `index()` method of Python lists, (ii) iterative version, (iii) recursive version over list of increasing size n . Note the different units of measures (ms versus μ s).

The number of comparisons is proportional to $\log_2 n$.

n	<code>list.index</code> (ms)	Iterative (ms)	Recursive (μ s)
10^3	0.01	0.04	2.14
10^4	0.10	0.37	2.90
10^5	0.98	3.67	3.51
10^6	9.17	36.95	4.22
10^7	91.82	364.61	5.07
10^8	920.67	3633.57	5.70

Table of contents

- 1 Introduction
- 2 Hanoi's Tower
- 3 Binary search
- 4 A non-classical problem

Gap: Problem definition

Gap

In a list L containing $n \geq 2$ integers, a **gap** is an index i , $0 < i < n$, such that $L[i - 1] < L[i]$.

- Prove that if $n \geq 2$ and $L[0] < L[n - 1]$, L contains at least one gap
- Design an algorithm that, given a list L containing $n \geq 2$ integers such that $L[0] < L[n - 1]$, finds a gap in the list.

Gap – Proof by contradiction

By contradiction:

- Suppose there is no gap in the list.
- Then $L[0] \geq L[1] \geq L[2] \geq \dots L[n-1]$, which contradicts the fact that $L[0] < L[n-1]$.

First version – Iterative

```
def gap(L):  
    for i in range(1,len(L)):  
        if L[i-1]<L[i]:  
            return i  
    return -1    # Never reached under the assumptions
```

```
L = list(range(100,0,-1))  
L.append(101)  
print(L)  
print(gap(L))
```

First version – Iterative

```
def gap(L):  
    for i in range(1,len(L)):  
        if L[i-1]<L[i]:  
            return i  
    return -1    # Never reached under the assumptions
```

```
L = list(range(100,0,-1))
```

```
L.append(101)
```

```
print(L)
```

```
print(gap(L))
```

```
[100, 99, 98, 97, ..., 3, 2, 1, 101]
```

```
100
```

Gap – Proof by induction

Let's reformulate the property in this way:

- Let L be a list of size n
- Let i, j be two indexes, such that $0 \leq i < j < n$ and $L[i] < L[j]$

In other words, there are at least two elements in the slice $L[i : j + 1]$ and the first element $L[i]$ is smaller than the last $L[j]$.

Gap – Proof by induction

We want to prove by induction on the size n of the slice that the slice contains a gap.

- **Base case:** $n = j - i + 1 = 2$, i.e. $j = i + 1$:
 $L[i] < L[j]$ implies that $L[i] < L[i + 1]$, which is a gap.
- **Inductive ipohthesis:** there is a gap in any slice of L smaller than n , where the first element is smaller than the last element.
- **Inductive step:** let's consider any element m such that $i < m < j$. There are two cases:
 - If $L[m] \leq L[i] < L[j]$, then there is a gap between m and j , as the slice $L[m : j + 1]$ is smaller than n .
 - If $L[i] < L[m]$, then there is a gap between i and m , as the slice $L[i : m + 1]$ is smaller than n .

Second version – Recursive

```
def gaprec(L, i, j):  
    if j == i+1:  
        return j  
    else:  
        m = (i+j) // 2  
        if L[m] < L[j]:  
            return gaprec(L,m,j)  
        else:  
            return gaprec(L,i,m)  
  
def gap(L):  
    return gaprec(L,0,len(L)-1)
```

Performance evaluation

Cost of execution of the iterative and recursive version of `gap()` over list of increasing size n . Note the different units of measures (ms versus μs).

n	Iterative (ms)	Recursive (μs)
10^3	0.06	2.05
10^4	0.61	2.78
10^5	6.11	3.36
10^6	62.44	4.01
10^7	621.69	4.87
10^8	6205.72	5.47