Recursion in Python

- Concept of Recursion
- Recursion Practices
- Divide and Conquer

Definition of Recursion Functions

- A recursive function is one that calls itself.

```
def i_am_recursive(x) :
    maybe do some work
    if there is more work to do :
        i_am_recursive(next(x))
    return the desired result
```

- Infinite loop? Not necessarily, not if `next(x)` needs less work than `x`.

Recursive Definition [1/2]

- A description of something that refers to itself is called a *recursive* definition

$$n! = n(n-1)(n-2)...(1)$$

$$n! = n(n-1)!$$

base case

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

- A recursive definitions should have two key characteristics:
  - There are one or more base cases for which no recursion is applied
  - All chains of recursion eventually end up at one of the base cases

Recursive Definition [2/2]

- Every recursive function definition includes two parts:
  - Base case(s) (non-recursive)
    One or more simple cases that can be done right away
  - Recursive case(s)
    One or more cases that require solving "simpler" version(s) of the original problem.
    - By "simpler", we mean "smaller" or "shorter" or "closer to the base case".

4

Recursive Computation Example: Factorial

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

  $2! = \quad 2 \times 1$

  $3! = \quad 3 \times 2 \times 1$

  $4! = \quad 4 \times 3 \times 2 \times 1$

- alternatively:

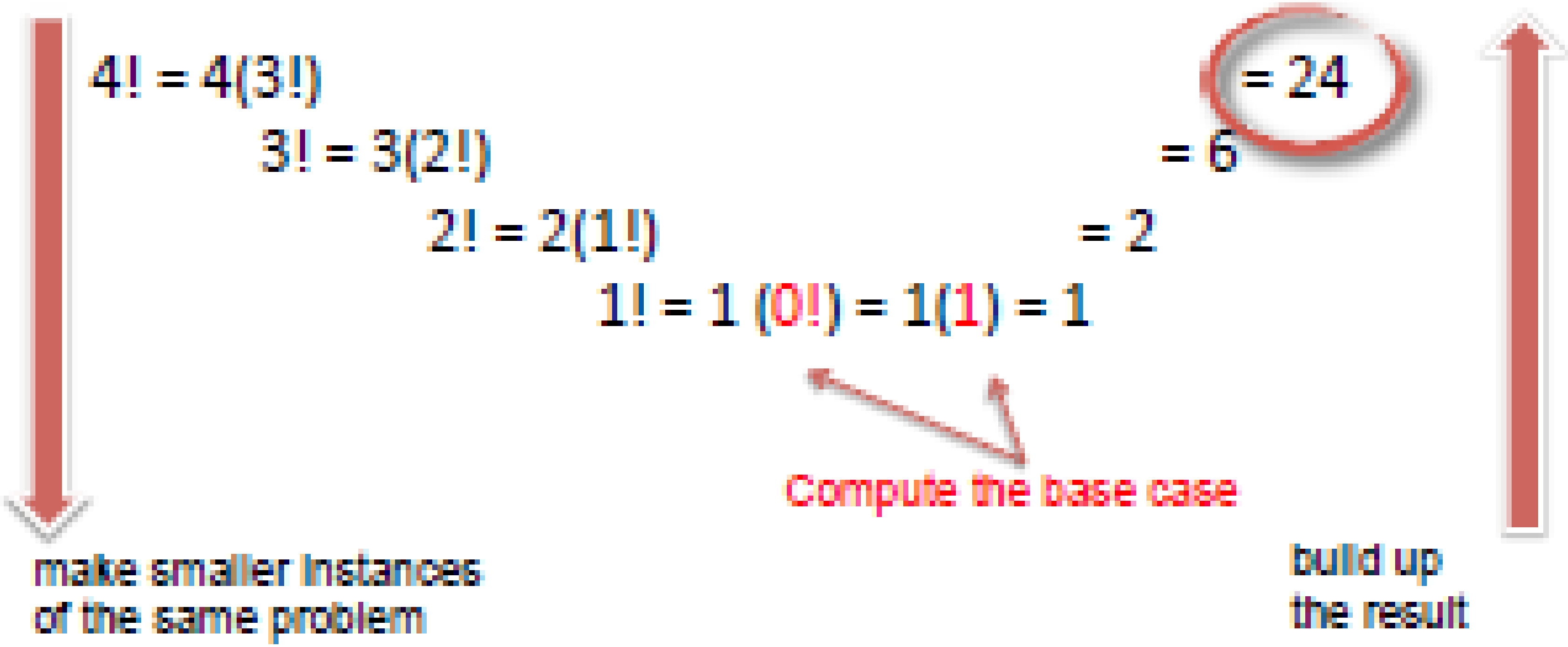
  $0! = 1$ (Base case)

  $n! = n \times (n-1)!$ (Recursive case)

  So $4! = 4 \times 3!$

  And $3! = 3 \times 2!$, $2! = 2 \times 1!$, $1! = 1 \times 0!$

Conceptual Understanding of Recursion



Recursive Factorial Function in Python

```
# 0! = 1 (Base case)
# n! = n × (n-1)! (Recursive case)
def factorial(n):
    if n == 0:        # base case
        return 1
    else:             # recursive case
        return n * factorial(n-1)
```

Inside Python Recursion Processing

S T A C K

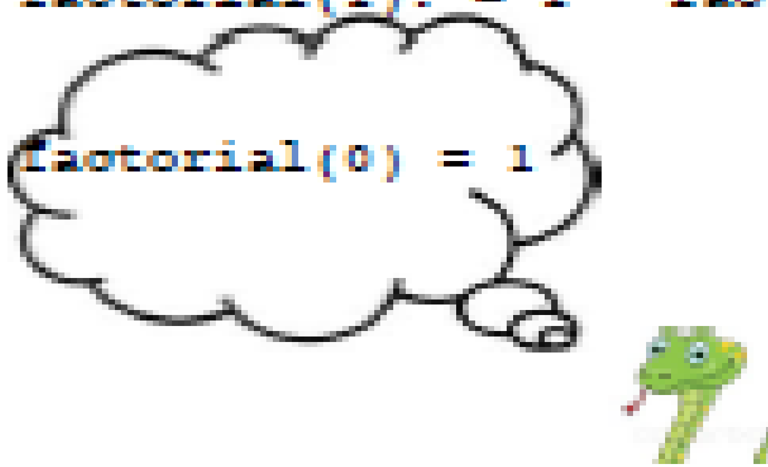
n=4 factorial(4)? = 4 * factorial(3)

n=3 factorial(3)? = 3 * factorial(2)

n=2 factorial(2)? = 2 * factorial(1)

n=1 factorial(1)? = 1 * factorial(0)

n=0



Recursive Solution vs. Iterative Solution

- For every recursive function, there is an equivalent iterative solution.
- For every iterative function, there is an equivalent recursive solution.
- But some problems are easier to solve one way than the other way.
- And be aware that most recursive programs need space for the stack, behind the scenes

Recursion in Python

- Concept of Recursion
- Recursion Practices
- Divide and Conquer

Factorial Function in Python (Iterative)

```
def factorial(n):
    result = 1    # initialize accumulator var
    for i in range(1, n+1):
        result = result * i
    return result
```

Factorial Function in Python (Recursive)

```
def factorial(n):
    if n == 0:     # base case
        return 1
    else:          # recursive case
        return n * factorial(n-1)
```

π Computation in Python [1/4]

Many Many Approximations

Bailey–Borwein–Plouffe formula

$$\pi = \sum_{i=0}^{\infty} \left[\frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right) \right].$$

Bellard's formula

$$\pi = \frac{1}{2^6} \sum_{n=0}^{\infty} \frac{(-1)^n}{2^{10n}} \left(-\frac{2^5}{4n+1} - \frac{1}{4n+3} + \frac{2^8}{10n+1} - \frac{2^6}{10n+3} - \frac{2^2}{10n+5} - \frac{2^2}{10n+7} + \frac{1}{10n+9} \right)$$

and

Chudnovsky algorithm

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}.$$

π Computation in Python [2/4]

The Algebraic Genius of Euler (1707, Switzerland)

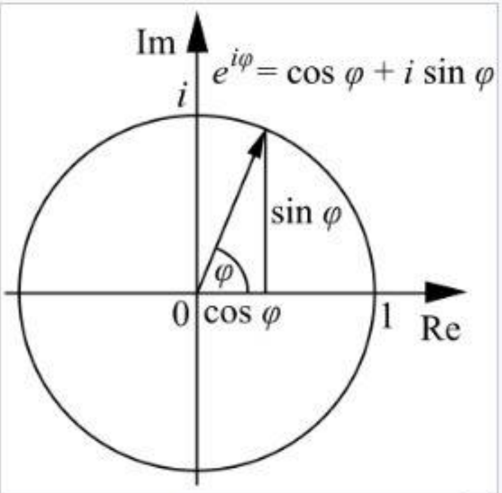
· The Basel Problem

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \lim_{n\to\infty}\left(\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\cdots+\frac{1}{n^2}\right) = \frac{\pi^2}{6}$$

· Euler's formula

He also defined the exponential function for complex numbers, and discovered its relation to the trigonometric functions. For any real number φ (taken to be radians), Euler's formula states that the complex exponential function satisfies

$$e^{i\varphi} = \cos\varphi + i\sin\varphi.$$



A geometric interpretation of Euler's formula

π Computation in Python [3/4]

Iterative Version of π Computation

- Mathematicians have proved
  $\pi^2/6 = 1 + 1/4 + 1/9 + 1/16 + \ldots$
- We can use this to approximate π
- Compute the sum, multiply by 6, take the square root

```
def pi_series_iter(n) :
    result = 0
    for i in range(1, n+1) :
        result = result + 1/(i**2)
    return result

def pi_approx_iter(n) :
    x = pi_series_iter(n)
    return (6*x)**(.5)
```

π Computation in Python [4/4]

Recursive Version of π Computation

```
def pi_series_r(i) :
    assert(i >= 0)
    # base case
    if i == 0:
        return 0
    # recursive case
    else:
        return pi_series_r(i-1) + 1 / i**2

def pi_approx_r(n) :
    x = pi_series_r(n)
    return (6*x)**(.5)
```

```
def test_pi_approx() :
    assert(pi_approx_iter(10) == 3.04936163598207)
    assert(pi_approx_iter(100) == 3.13207653180911053)
    assert(pi_approx_iter(1000) == 3.1406380562059946)
    assert(pi_approx_iter(10000) == 3.1414971639472147)
    # Python's default stack depth limit is 1000, so we can't compute pi_approx_r(1000)
    for i in range(996) :
        assert(pi_approx_r(i) == pi_approx_iter(i))
    print("Done testing pi approximations")
```

Recursion on Lists: Sum of a List [1/2]

- First we need a way of getting a smaller input from a larger one:
  - Forming a sub-list of a list:

```
>>> a = [1, 11, 111, 1111, 11111, 111111]
>>> a[1:]
[11, 111, 1111, 11111, 111111]
>>> a[2:]
[111, 1111, 11111, 111111]
>>> a[3:]
[1111, 11111, 111111]
>>> a[3:5]
[1111, 11111]
>>>
```

the "tail" of list a

Recursive Sum of a List

```
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

What if we already know the sum of the list's tail? We can just add the list's first element!

Recursion on Lists: Sum of a List [2/2]

Tracing sumlist

```
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

```
>>> sumlist([2,5,7])
sumlist([2,5,7]) = 2 + sumlist([5,7])
                       5 + sumlist([7])
                           7 + sumlist([])
                                       0
```

After reaching the base case, the final result is built up by the computer by adding 0+7+5+2.

Multiple Recursive Calls: Fibonacci Numbers

$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \qquad n > 1$

- A sequence of numbers:

0
+
1
+
1
+
2
+
3
+
5
+
8
+
13

....

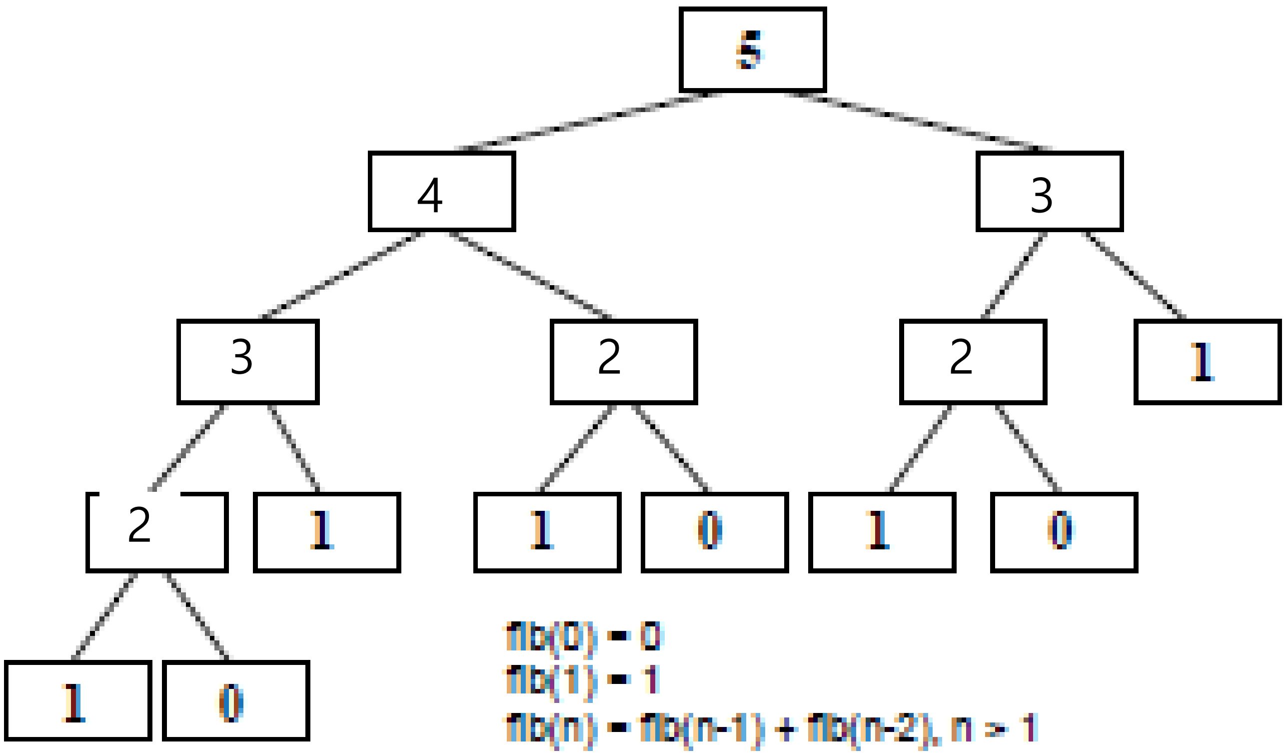
Recursive Definition of Fibonacci Numbers

- Let fib(n) = the nth Fibonacci number, n ≥ 0
  - fib(0) = 0 (base case)
  - fib(1) = 1 (base case)
  - fib(n) = fib(n-1) + fib(n-2), n > 1

Two recursive calls!

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Recursive Call Tree of Fibonacci Number



fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

Iterative Fibonacci Python Function

```
def fib(n):
    x = 0
    next_x = 1
    for i in range(1,n+1):
        x, next_x = next_x, x + next_x
    return x
```

SIMULTANEOUS ASSIGNMENT

Faster than the recursive version. Why?

Recursion on String: String Reversal [1]

- Write a function to reverse a given string
  - Divide it up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end

```
>>> def reverse(s):
        return reverse(s[1:]) + s[0]

>>> reverse("Hello")

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    reverse("Hello")
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
  …
 File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
RuntimeError: maximum recursion depth exceeded
```

- What happened? There were 1000 lines of errors!

Recursion on String: String Reversal [2]

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]
```

```
>>> reverse("Hello")
'olleH’
```

- Python stops it at 1000 calls, the default “maximum recursion depth.”
  - Each time a function is called it takes some memory.

Recursion on Greatest Common Denominator (GCD)

```python
def gcd(a, b):
    """Calculate the Greatest Common Divisor of a and b.

    Unless b==0, the result will have the same sign as b (so that when
    b is divided by it, the result comes out positive).
    """
    while b:
        a, b = b, a%b
    return a
```

```python
def gcd(x,y):
    while (y > 0):
        oldX = x
        x = y
        y = oldX % y
    return x
```

```python
print(gcd(500,420))          # 20
```

	Iterative Solution	Recursive Solution	Recursive Solution with Stack Trace

factorial	def factorial(n):
	    factorial = 1
	    for i in range(2,n+1):
	        factorial *= i
	    return factorial
	
	print(factorial(5))
		def factorial(n):
		    if (n < 2):
		        return 1
		    else:
		        return n*factorial(n-1)
		
		print(factorial(5))
			def factorial(n, depth=0):
			    print(“ “*depth,”factorial(”,n.”):”
			    if (n < 2):
			        result = 1
			    else:
			        result = n*factorial(n-1,depth+1)
			    print(“ “*depth,”→”, result)
			    return result
			
			print(factorial(5))

reverse	def reverse(s):
	    reverse = ""
	    for ch in s:
	        reverse = ch + reverse
	    return reverse
	
	print(reverse(“abcd”))
		def reverse(s):
		    if (s == ""):
		        return ""
		    else:
		        return reverse(s[1:]) + s[0]
		
		print(reverse(“abcd”))
			def reverse(s, depth=0):
			    print(“ “*depth,”reverse(”,s.”):”
			    if (s == ""):
			        result = ""
			    else:
			        result = reverse(s[1:], depth+1) + s[0]
			    print(“ “*depth,”→”, result)
			    return result
			
			print(reverse(“abcd”))

gcd	def gcd(x,y):
	    while (y > 0):
	        oldX = x
	        x = y
	        y = oldX % y
	    return x
	
	print(gcd(500,420)) # 20
		def gcd(x,y):
		    if (y == 0):
		        return x
		    else:
		        return gcd(y,x%y)
		
		print(gcd(500,420)) # 20
			def gcd(x,y,depth=0):
			    print(“ “*depth,”gcd(”,x,”,”,y,.”):”
			    if (y == 0):
			        result = x
			    else:
			        result = gcd(y,x%y,depth+1)
			    print(“ “*depth,”→”, result)
			    return result
			
			print(gcd(500,420)) # 20

Recursion in Python

- Concept of Recursion
- Recursion Practices
- Divide and Conquer

Family of Algorithms

- Greedy Methods
- Divide and Conquer
- Dynamic Programming
- Branch and Bound
- Back Tracking

전통적인 Computer Science Algorithms

- Machine Learning Algorithm
- Genetic Algorithm
- Randomized Algorithm

Approximation과 Prediction을 하는 Algorithms

- Mathematical Programming
  - Integer Programming
  - Linear Programming
  - Non-Linear Programming
  - Unconstrined Extrema
  - Constrined Extrema

Applied Mathematics or Industrial Engineering에서 하는 Algorithms

Recursion Example: Fast Exponentiation [1]

- One way to compute a^n : multiply a by itself n times.

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

- Another way to compute a^n : divide and conquer!
  - $a^n = a^{n//2}(a^{n//2})$?

$$a^n = \begin{cases} a^{n//2}(a^{n//2}) & \text{if } n \text{ is even} \\ a^{n//2}(a^{n//2})(a) & \text{if } n \text{ is odd} \end{cases}$$

  - $2^8 = 2^4(2^4)$
  - $2^9 = 2^4(2^4)2$

Recursion Example: Fast Exponentiation [2]

```
def recursivePower(a, n) :
    # raises a to the int power n
    if n ==  0:
        return 1
    else:
        factor = recursivePower(a, n//2)
        if n%2 == 0:                              # n is even
            return factor * factor
        else:                                     # n is odd
            return factor * factor * a
```

- temporary variable *factor* is used so that we don't need to calculate $a^{n//2}$ more than once

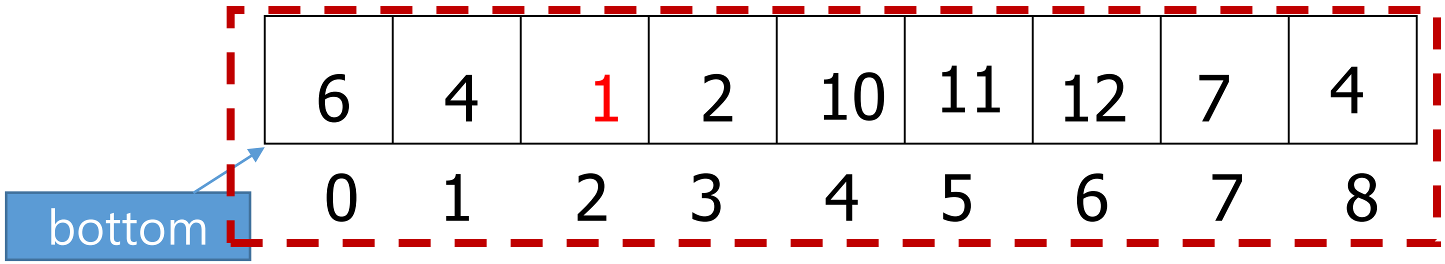
Sorting Algorithms

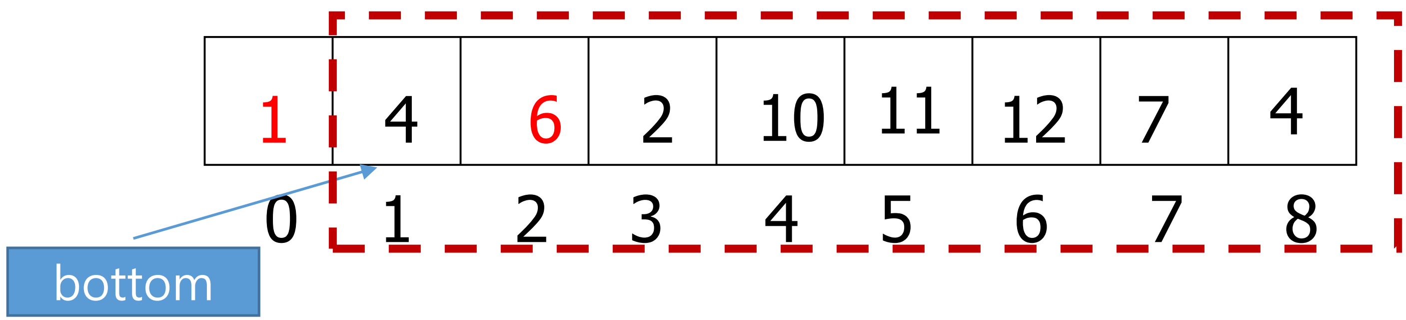
- The sorting problem
  - take a list of *n* elements
  - and rearrange it so that the values are in increasing (or decreasing) order.

- *Selection sort*
  - For *n* elements, we find the smallest value and put it in the 0^{th} position.
  - Then we find the smallest remaining value from position 1 to (*n*-1) and put it into position 1.
  - The smallest value from position 2 to (*n*-1) goes in position 2.
  - …

Naive Sorting: Selection Sort

```
def selSort(nums): # sort nums into ascending order

 n = len(nums)
 # For each position in the list (except the very last)
 for bottom in range(n-1):
    # find the smallest item in nums[bottom]...nums[n-1]

    mp = bottom                         # bottom is smallest initially
    for i in range(bottom+1, n):        # look at each position
        if nums[i] < nums[mp]:          # this one is smaller
            mp = i                      # remember its index

    # swap smallest item to the bottom
    nums[bottom], nums[mp] = nums[mp], nums[bottom]
```





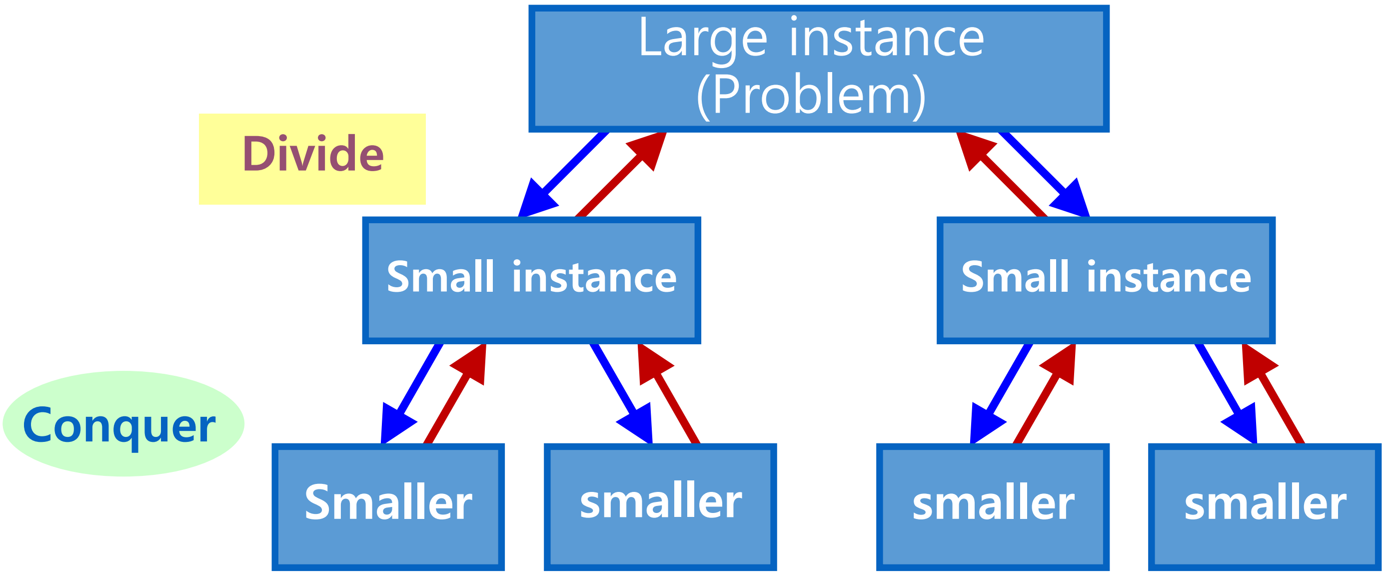
Divide and Conquer

- In computation:
  - Divide the problem into “simpler” versions of itself.
  - Conquer each problem using the same process (usually recursively).
  - Combine the results of the “simpler” versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort, Quicksort, and many, many more

Divide and Conquer style programming은 recursion이 자연스럽다!

Divide and Conquer Style Algorithm

- Distinguish between small and large instances
- Small instances solved differently from large ones
- All instances are non-overlapping

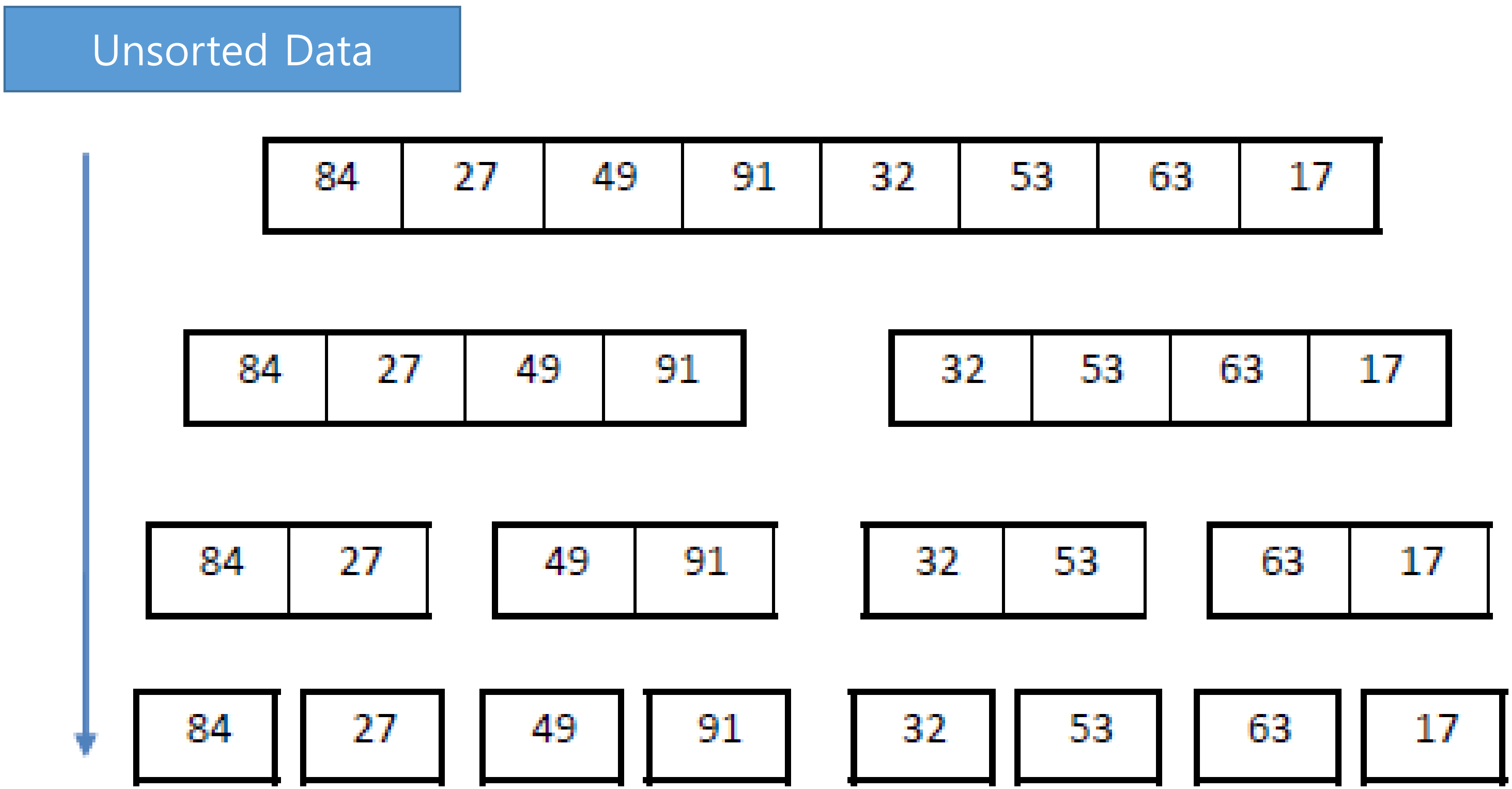


Divide and Conquer Example: Merge Sort

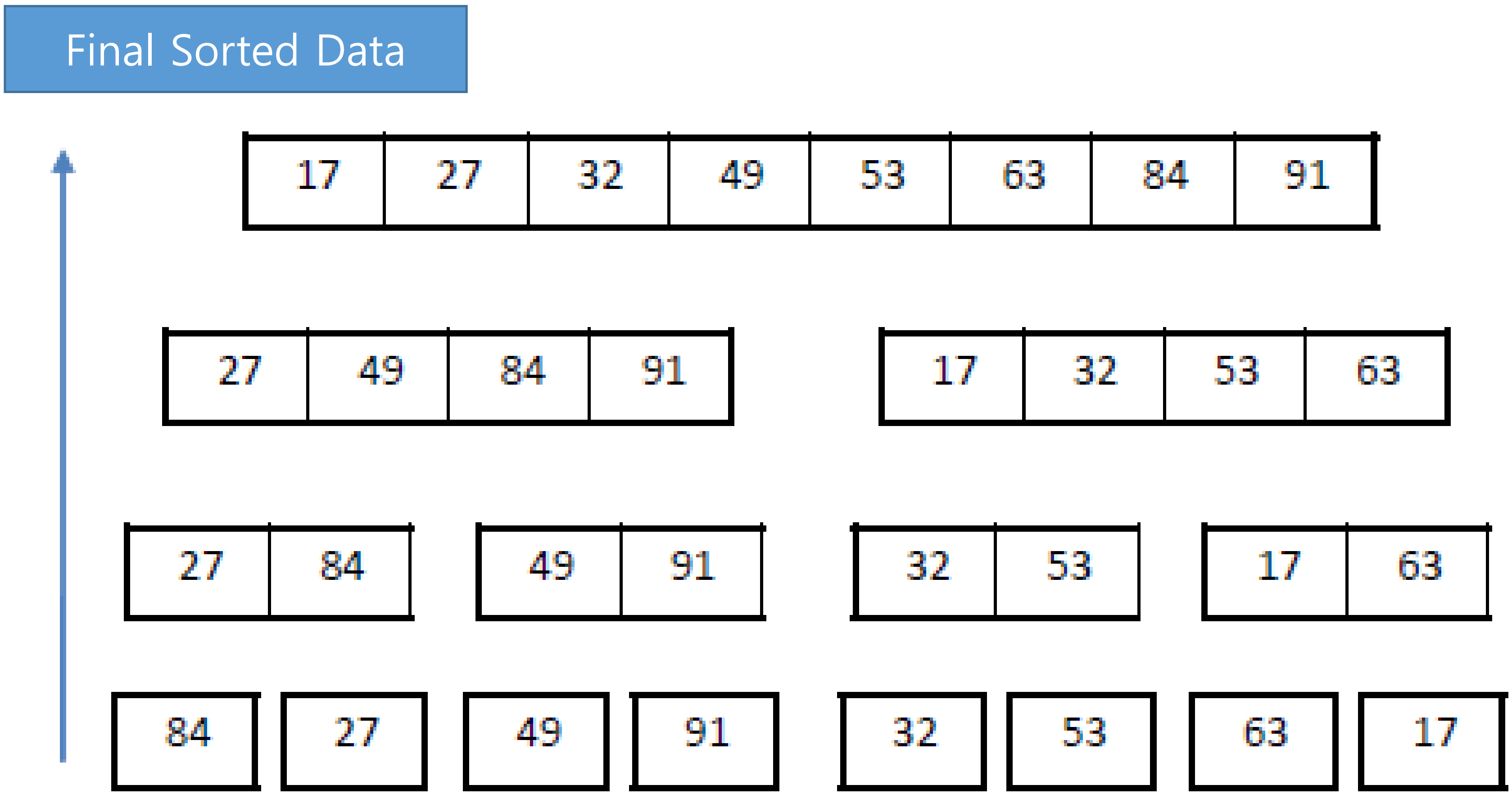
- *merge sort*
  - Merging: combining two sorted lists into a single sorted list

```
Step1: split nums into two halves

Step2: sort the first half

Step3: sort the second half

Step4: merge the two sorted halves back into nums
```

Divide (Split)



Conquer (Merge)



Outline of Merging 2 Lists

- Input: Two lists *a* and *b*, already sorted
- Output: A new list containing the elements of *a* and *b* merged together in sorted order.
- Algorithm:

1. Create an empty list *c*, set *index_a* and *index_b* to 0
2. While *index_a* < length of *a* and *index_b* < length of b
   a. Add the smaller of *a*[*index_a*] and *b*[*index_b*] to the end of *c*, and increment the index of the list with the smaller element
3. If any elements are left over in *a* or *b*, add them to the end of *c*, in order
4. Return *c*

Divide and Conquer Example: Merge Sort [1]

```
def merge(lst1, lst2, lst3):
    # merge sorted lists lst1 and lst2 into lst3
    # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0  # all start at the front
    n1, n2 = len(lst1), len(lst2)
    # Loop while both lst1 and lst2 have more items
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]: # top of lst1 is smaller
            lst3[i3] = lst1[i1] #  copy it into current spot in lst3
            i1 = i1 + 1
        else:                   # top of lst2 is smaller
            lst3[i3] = lst2[i2] #  copy itinto current spot in lst3
            i2 = i2 + 1
        i3 = i3 + 1             # item added to lst3, update position

    # Here either lst1 or lst2 is done. One of the following loops
    # will execute to finish up the merge.
    while i1 < n1: # Copy remaining items (if any) from lst1
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1
    while i2 < n2: # Copy remaining items (if any) from lst2
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
```

Divide and Conquer Example: Merge Sort [2]

```
def mergeSort(nums):
    # Put items of nums into ascending order
    n = len(nums)

    if n > 1:    # Do nothing if nums contains 0 or 1 items

        m = n/2  # split the two sublists
        nums1, nums2 = nums[:m], nums[m:]
                # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
                # merge the sorted pieces back
        merge(nums1, nums2, nums)
```

Comparing Sorts using Time Complexity

- Selection Sort (➔ n^2 algorithm)
- For a list of size n
  - To find the smallest element, the algorithm inspects all n items
  - The next time through the loop, it inspects the remaining n-1 items
- The total number of comparisons in iterations is:

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{n(n+1)}{2}$$

- contains an n^2 term: the number of steps in the algorithm is proportional to the square of the size of the list

- Merge Sort (➔ n*log(n) algorithm)
- For a list of size n
- The number of levels: $\log_2 n$
- The number of comparisons in merge step of each level: a little bit less than n

=> total work required to sort n items: $n*\log_2 n$

Comparing Algorithms