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".

Recursive Computation Example: Factorial

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

  $2! = \quad 2 \times 1$

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

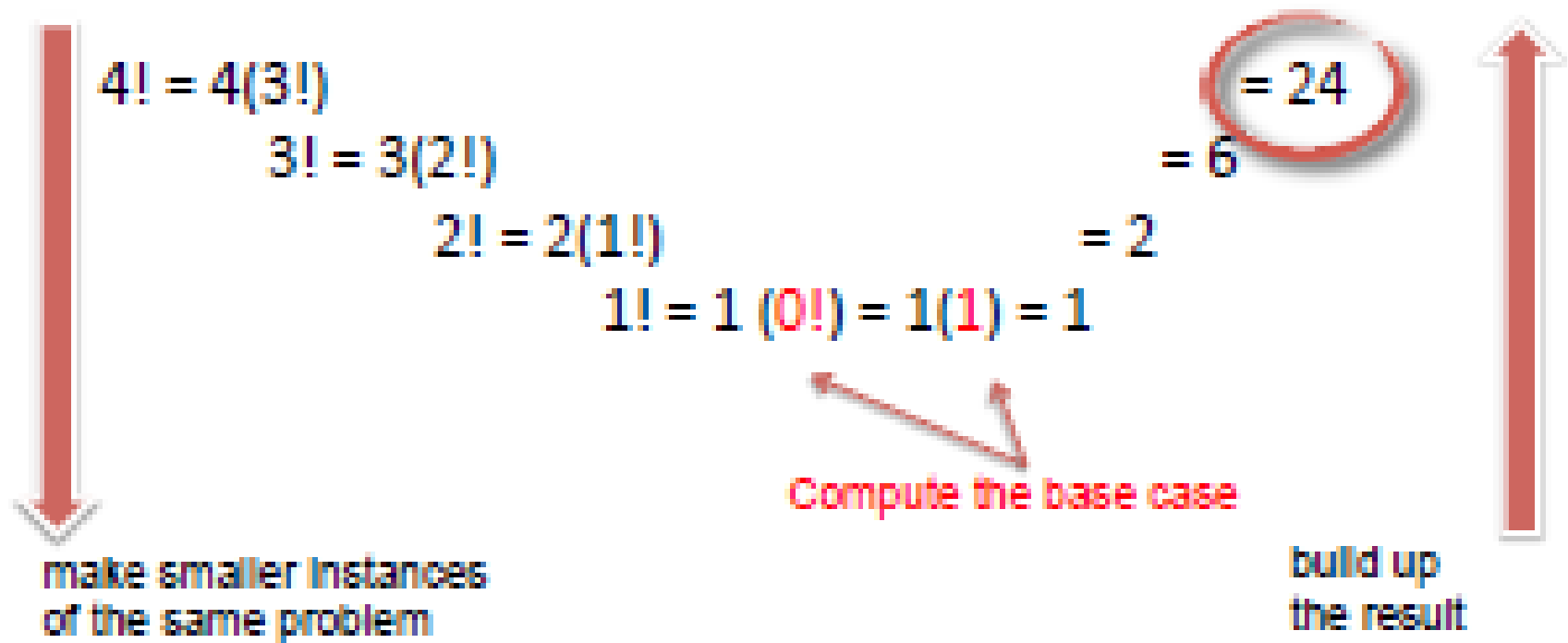 $4! = \quad 4 \times 3 \times 2 \times 1$

- 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

$4! = 4(3!)$ $= 24$

    $3! = 3(2!)$ $= 6$

        $2! = 2(1!)$ $= 2$

           $1! = 1 (0!) = 1(1) = 1$

Compute the base case

make smaller instances
of the same problem

build up
the result

6

Recursive Factorial Function in Python

```python
# 0! = 1 (Base case)
# n! = n x (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

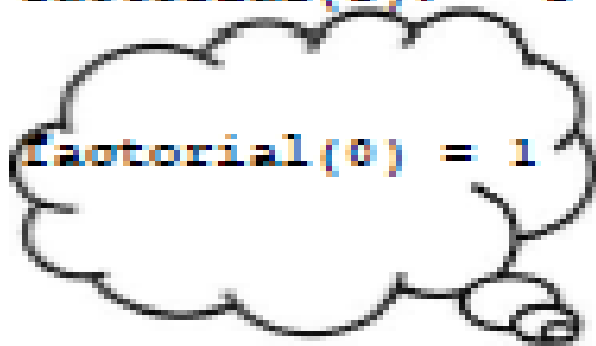| STACK		
S	n=4	factorial(4)? = 4 * factorial(3)
T	n=3	factorial(3)? = 3 * factorial(2)
A	n=2	factorial(2)? = 2 * factorial(1)
C	n=1	factorial(1)? = 1 * factorial(0)
K	n=0	factorial(0) = 1

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)

```python
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)

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

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:]                                    the "tail" of list a
[11, 111, 1111, 11111, 111111]
>>> a[2:]
[111, 1111, 11111, 111111]
>>> a[3:]
[1111, 11111, 111111]
>>> a[3:5]
[1111, 11111]
>>>
```

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

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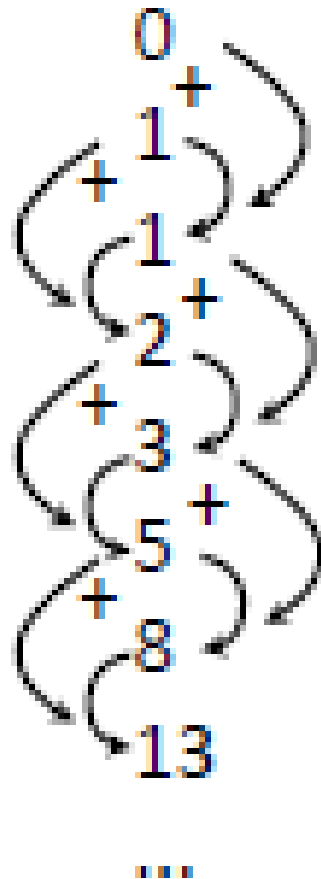
After reaching the base case, the final result is built up by the computer by adding 0+7+5+2.

Multiple Recursive Calls: Fibonacci Numbers

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

- A sequence of numbers:

0
1
1
2
3
5
8
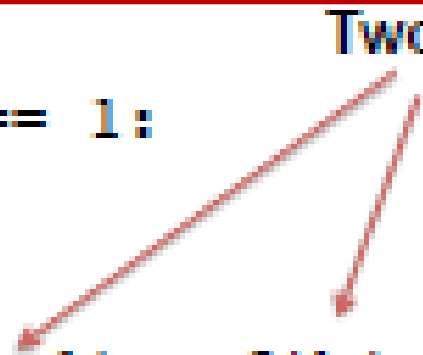13
...

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

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

Two recursive calls!

Recursive Call Tree of Fibonacci Number

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1
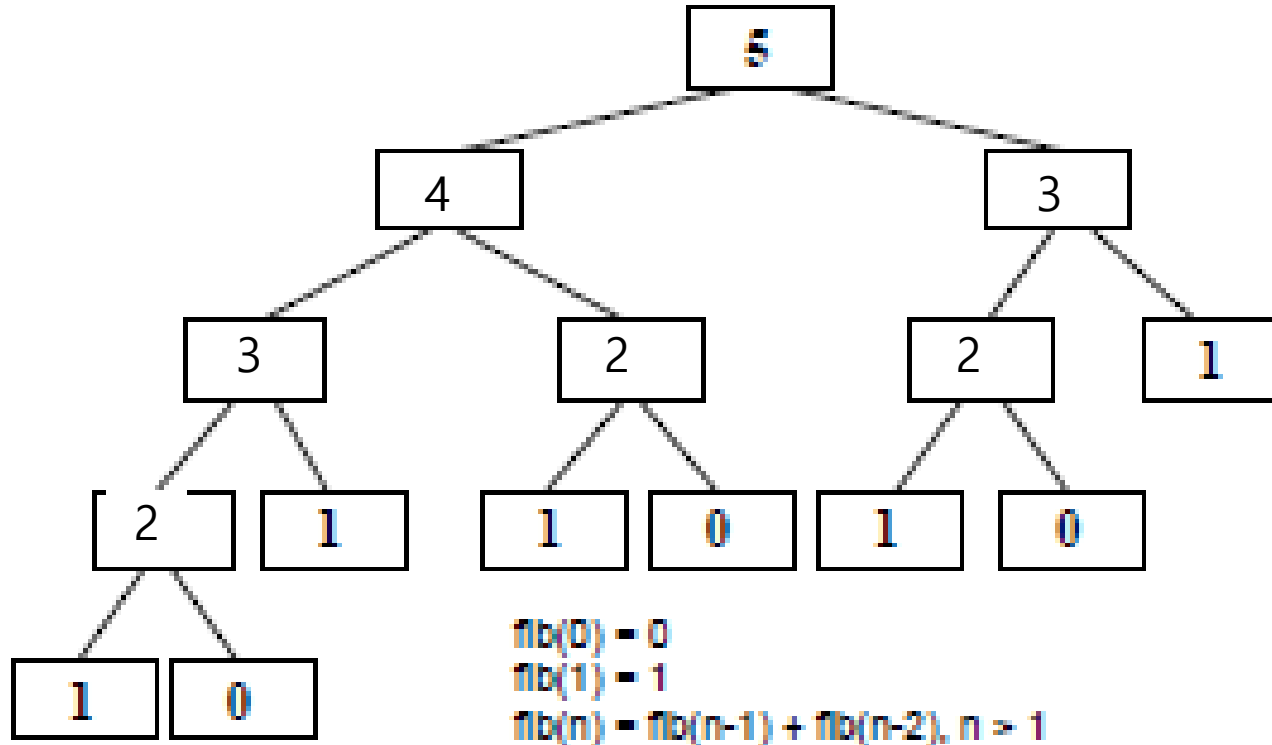
Iterative Fibonacci Python Function

```python
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!

18

Recursion on String: String Reversal [2]

```python
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 with Stack Trace: Factorial Function

Iterative Solution

```
def factorial(n):
    factorial = 1
    for i in range(2,n+1):
        factorial *= i
    return factorial
```

print(factorial(5))

Recursive Solution

```
def factorial(n):
    if (n < 2):
        return 1
    else:
        return n*factorial(n-1)
```

print(factorial(5))

Recursive Solution with Stack Trace

```
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))

Recursion with Stack Trace: Reverse Function

Iterative Solution

```
>>> def reverse(s):
        reverse = ""
        for ch in s:
                reverse = ch + reverse
        return reverse

>>>
>>> print(reverse("abcd"))
dcba
```

Recursive Solution

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

>>>
>>> print(reverse("abcd"))
```

Recursive Solution with Stack Trace

```
>>> def reverse(s, depth = 0):
        print(" "*depth, "reverse(", s, ")")
        if (s == ""):
                return ""
        else:
                return reverse(s[1:], depth + 1) + s[0]
        print(" "*depth, "-->", result)
        return result

>>>
>>> print(reverse("abcd"))
 reverse( abcd )
  reverse( bcd )
   reverse( cd )
    reverse( d )
     reverse(   )
dcba
```

Recursion with Stack Trace:
Greatest Comon Denominator (GCD) Function

Iterative Solution

```
def gcd(x,y):
    while (y > 0):
        oldX = x
        x = y
        y = oldX % y
    return x
```
print(gcd(500,420))   # 20

Recursive Solution

```
def gcd(x,y):
    if (y == 0):
        return x
    else:
        return gcd(y,x%y)
```
print(gcd(500,420))   # 20

Recursive Solution with Stack Trace

```
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
```

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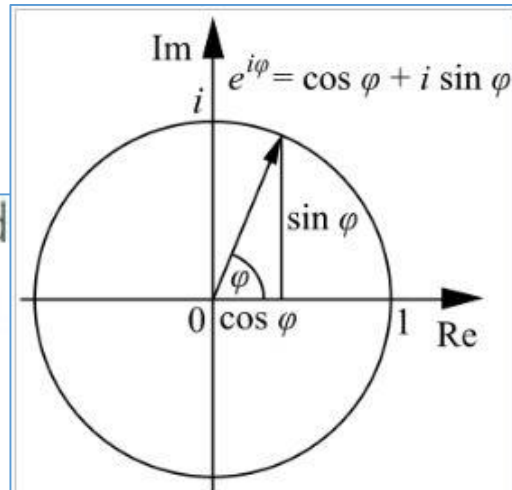
$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \lim_{n \to \infty} \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} \right) = \frac{\pi^2}{6}$$

· 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

```python
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)
```

Recursive Version of π Computation

```python
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)
```

```python
def test_pi_approx() :
    assert(pi_approx_iter(10) == 3.04936163598207)
    assert(pi_approx_iter(100) == 3.1320765318091053)
    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 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

Divide and Conquer

- In computation:
  - **Divide** the problem into "simpler" versions of itself.
  - **Conquer** each problem using the same process (usually <u>recursively</u>).
  - **Combine** the results of the "simpler" versions to form your final solution.

- Problems that fit well with Divide & Conquer recursion style
  - Tower of Hanoi
  - Fractals
  - Binary Search
  - Merge Sort
  - Quicksort
  - 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

Large instance
(Problem)

Divide

Small instance   Small instance

Conquer

Smaller   smaller   smaller   smaller
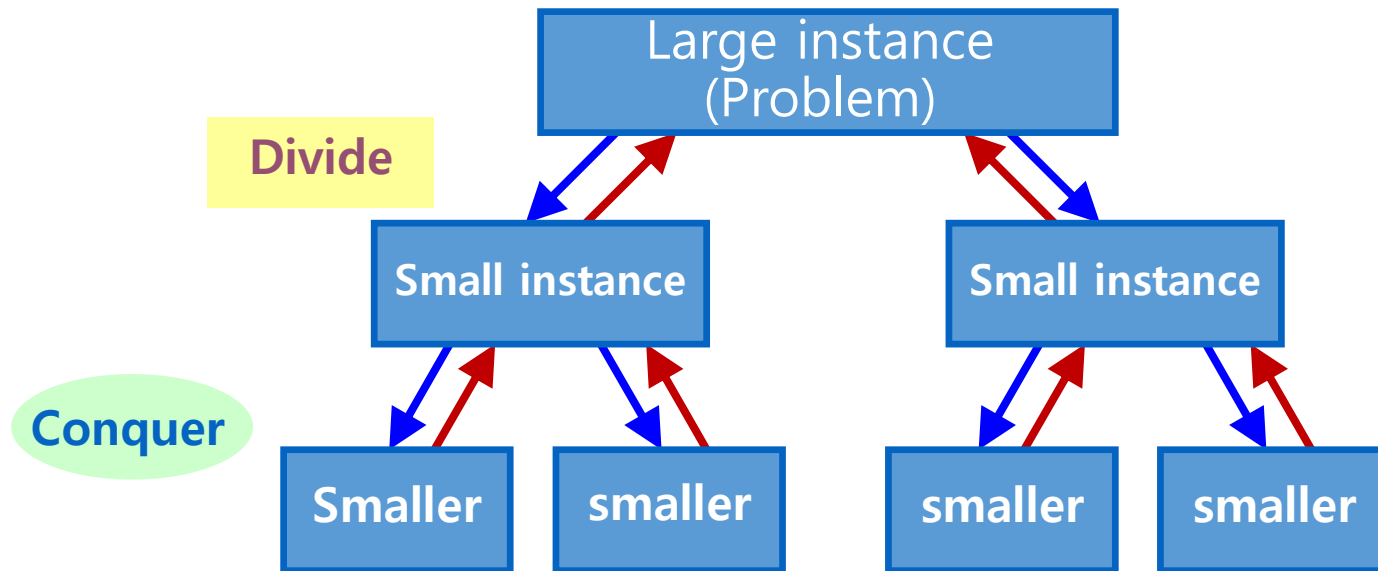
Recursion Example: Fast Exponentiation [1]

- 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]

```python
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
```

- The 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 0th 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.
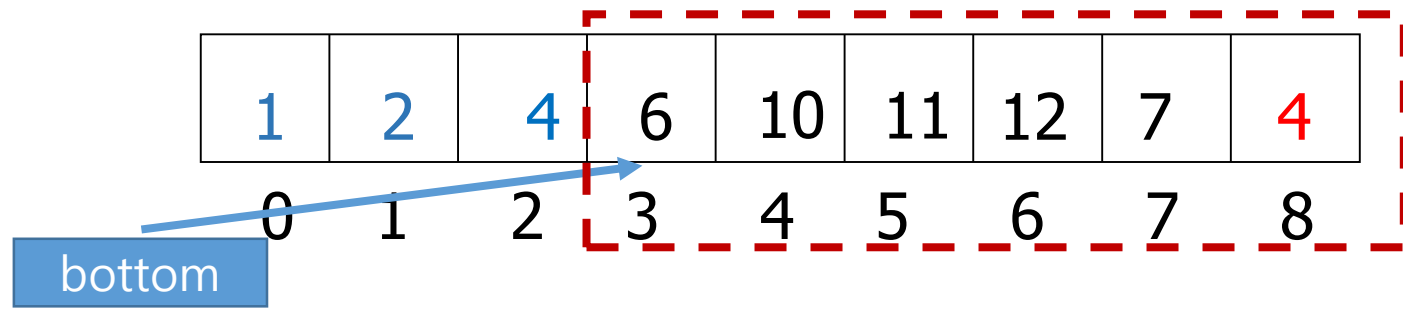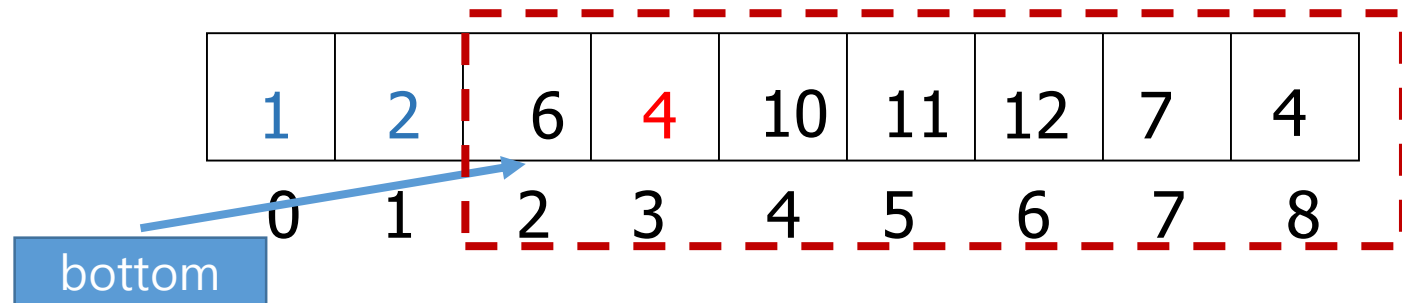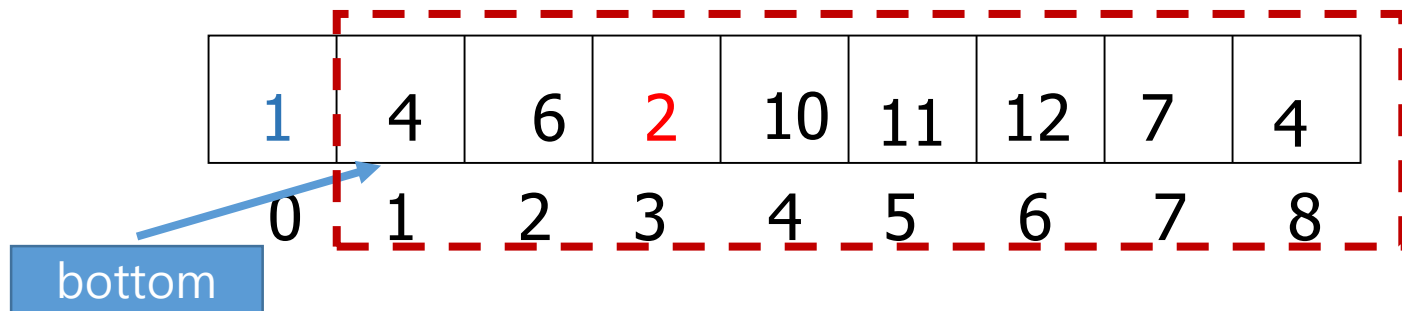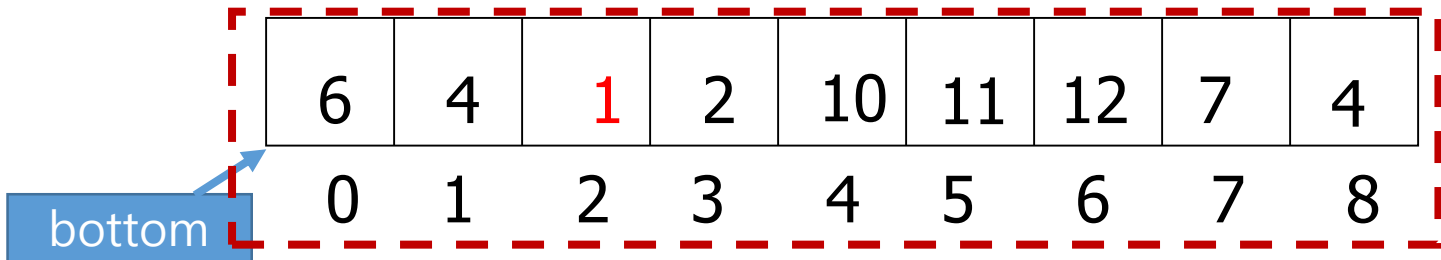 - …

Naive Sorting: Selection Sort   [1/2]

```python
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 the smallest item to the bottom
    nums[bottom], nums[mp] = nums[mp], nums[bottom]
```
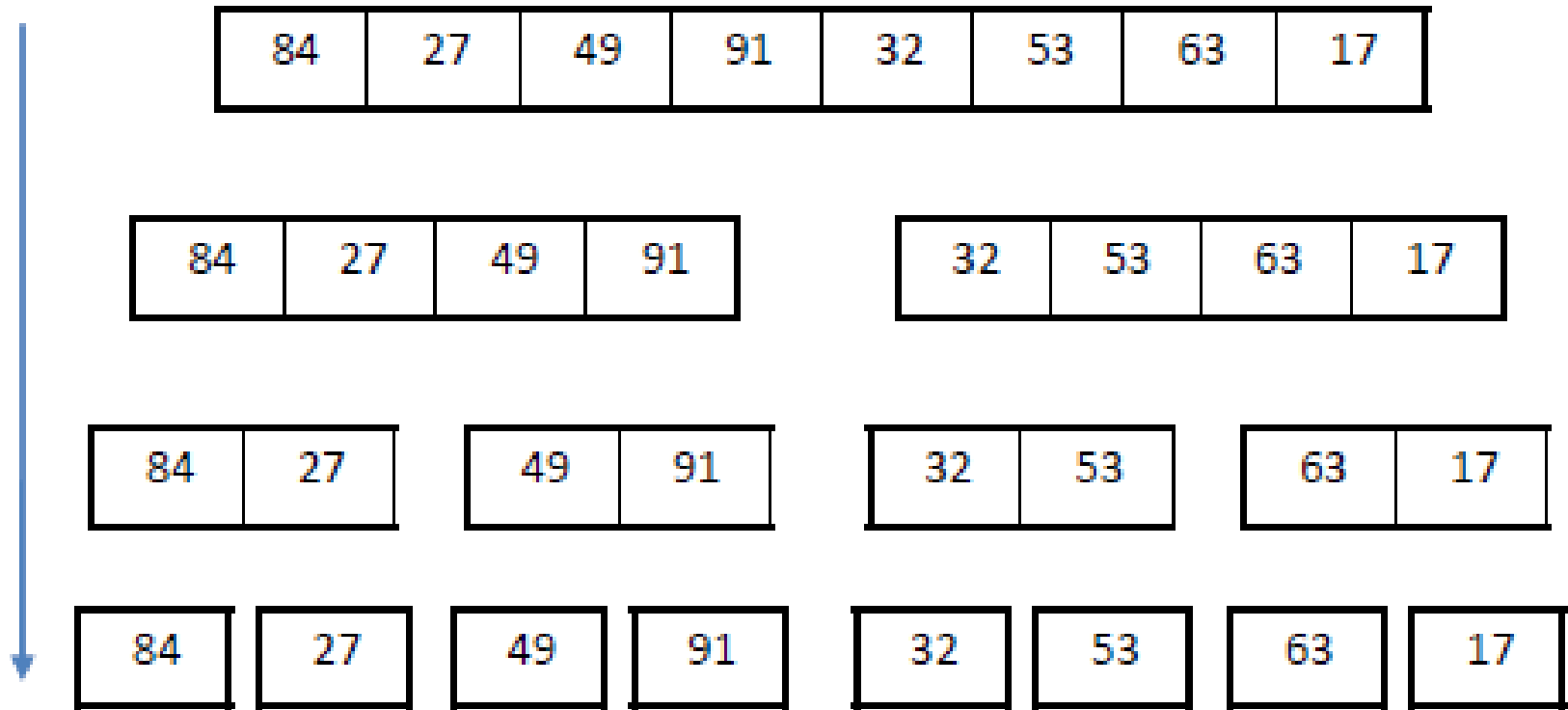
Naive Sorting: Selection Sort   [2/2]

6	4	1	2	10	11	12	7	4
0	1	2	3	4	5	6	7	8

bottom

1	4	6	2	10	11	12	7	4
0	1	2	3	4	5	6	7	8

bottom

1	2	6	4	10	11	12	7	4
0	1	2	3	4	5	6	7	8

bottom

1	2	4	6	10	11	12	7	4
0	1	2	3	4	5	6	7	8

bottom

Merge-Sort: Divide Step (Split)

Unsorted Data

| 84 | 27 | 49 | 91 | 32 | 53 | 63 | 17 |

| 84 | 27 | 49 | 91 | | 32 | 53 | 63 | 17 |

| 84 | 27 | | 49 | 91 | | 32 | 53 | | 63 | 17 |

| 84 | | 27 | | 49 | | 91 | | 32 | | 53 | | 63 | | 17 |

Merge-Sort: Conquer Step (Merge)

Final Sorted Data

17	27	32	49	53	63	84	91

27	49	84	91

17	32	53	63

27	84

49	91

32	53

17	63

84	27	49	91	32	53	63	17
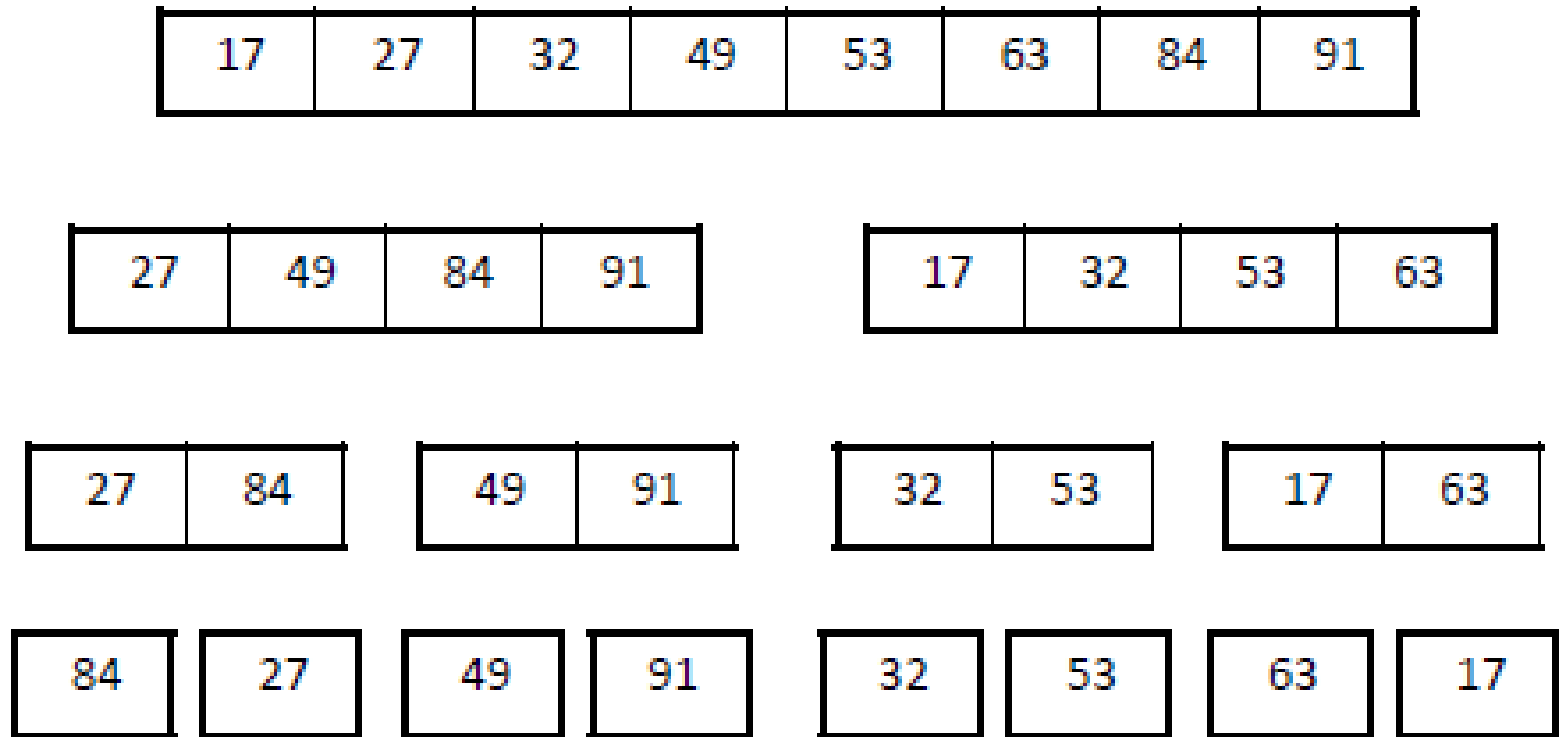
Divide and Conquer Example
msort() - Merge Sort

- Input: List a of n elements.

- Output: Returns **a new list** containing the same elements in sorted order.

- Algorithm:

  1. If less than two elements, return a **copy** of the list **(base case!)**

  2. Sort the first half using merge sort. **(recursive!)**

  3. Sort the second half using merge sort. **(recursive!)**

  4. **Merge** the two sorted halves to obtain the final sorted array.

Outline of merge(): Merging Two Sorted 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

Python code for msort()

```python
def msort(list):
    if len(list) == 0 or len(list) == 1: # base case
        return list[:len(list)] # copy the input
    # recursive case
    halfway = len(list) // 2
    list1 = list[0:halfway]
    list2 = list[halfway:len(list)]
    newlist1 = msort(list1) # recursively sort left half
    newlist2 = msort(list2) # recursively sort right half
    newlist = merge(newlist1, newlist2)
    return newlist
```
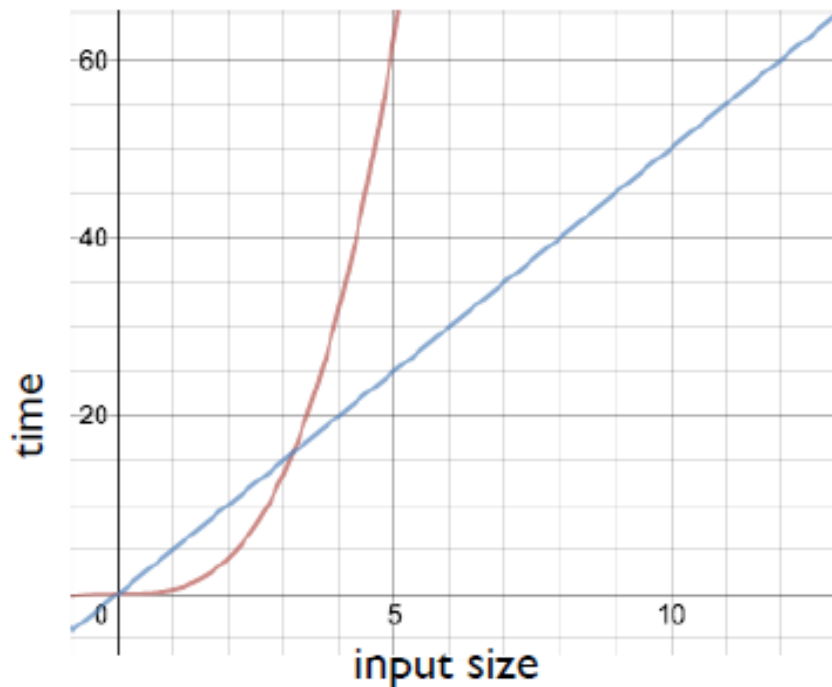
Python code for merge()

```python
def merge(a, b):
    index_a = 0 # the current index in list a
    index_b = 0 # the current index in list b
    c = []
    while index_a < len(a) and index_b < len(b):
        if a[index_a] <= b[index_b]:
            c.append(a[index_a])
            index_a = index_a + 1
        else:
            c.append(b[index_b])
            index_b = index_b + 1
    # when we exit the loop
    # we are at the end of at least one of the lists
    c.extend(a[index_a:])
    c.extend(b[index_b:])
    return c
```

Measuring Algorithm Efficiency
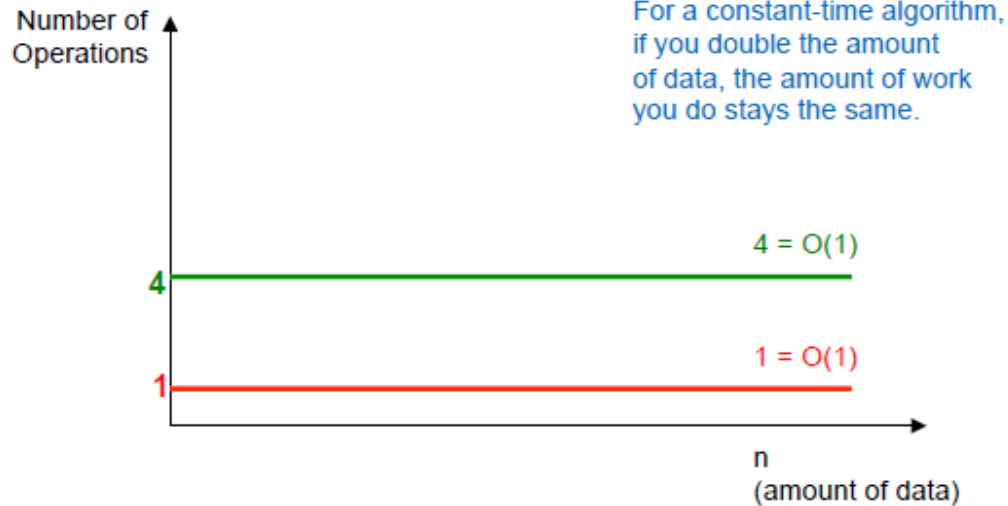
Run time as function of input size

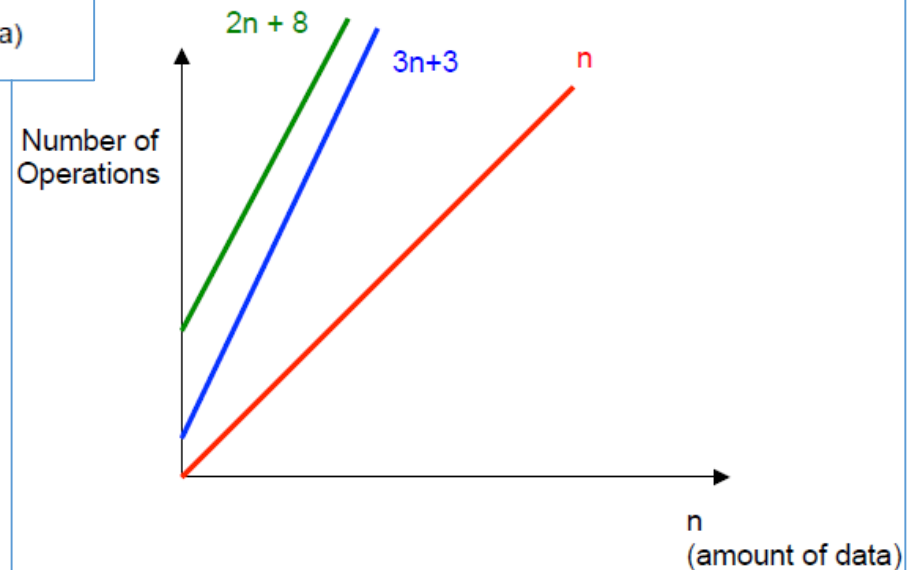- Graphs of the time complexity of a more efficient and a less efficient algorithm
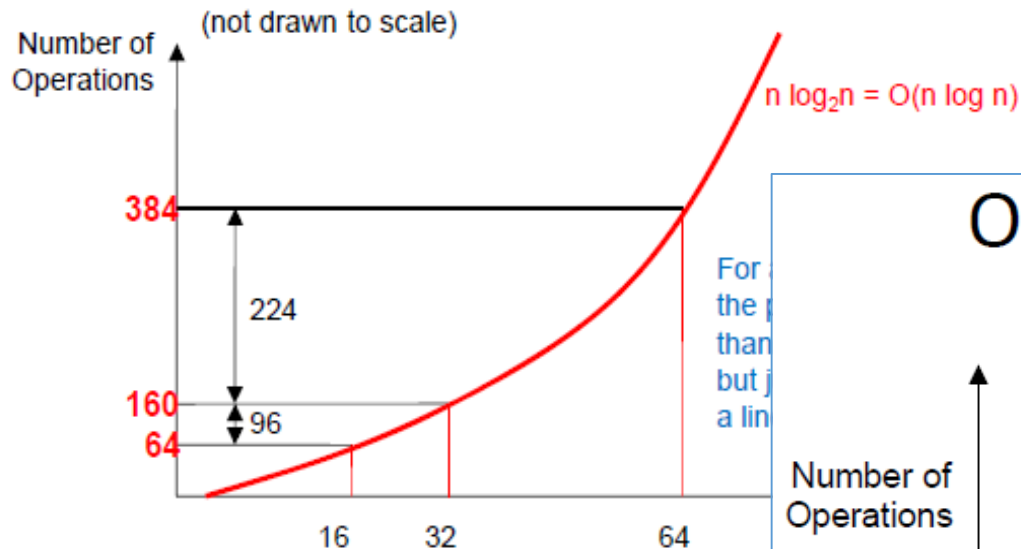
Big-Oh Notation [1/2]

O(1) ("Constant time")

Number of Operations
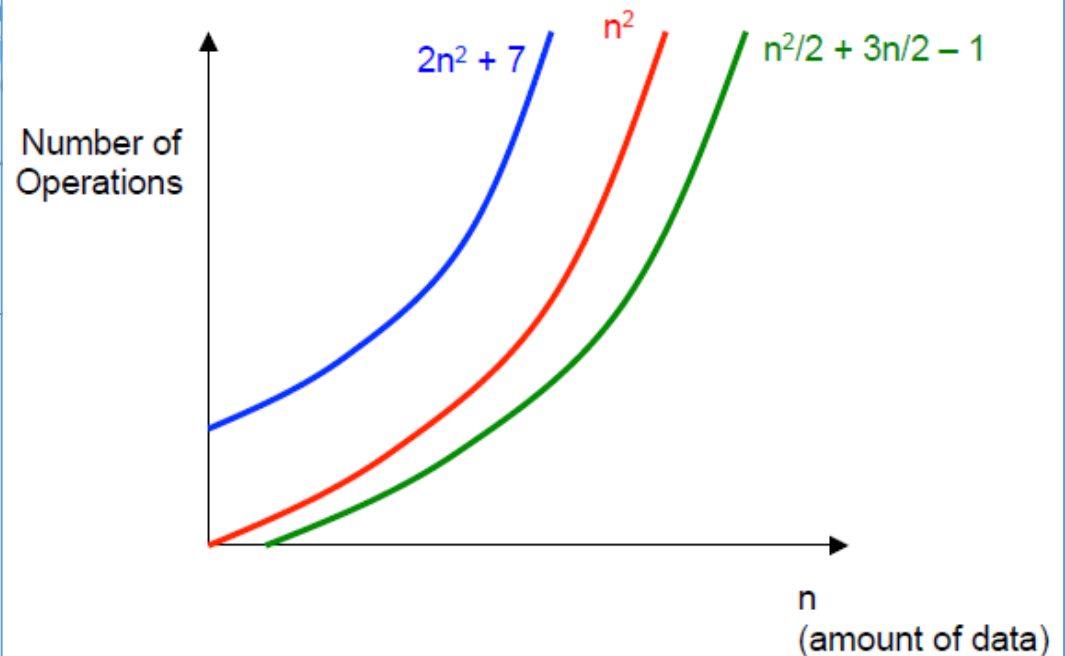
For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.

4 = O(1)

4

1 = O(1)

1

n (amount of data)

O(n) ("Linear time")

2n + 8

3n+3

n

Number of Operations

n (amount of data)

O(N log N)

(not drawn to scale)

Number of Operations

$n \log_2 n = O(n \log n)$

384

224

160

96

64

For
the
than
but j
a lin

16   32   64

O(n²) ("Quadratic time")

$2n^2 + 7$ n^2 $n^2/2 + 3n/2 - 1$

Number of Operations

n
(amount of data)

Comparing Algorithms

Comparing Big O Functions

Number of Operations

$O(2^n)$ $O(n^2)$ $O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$

n
(amount of data)
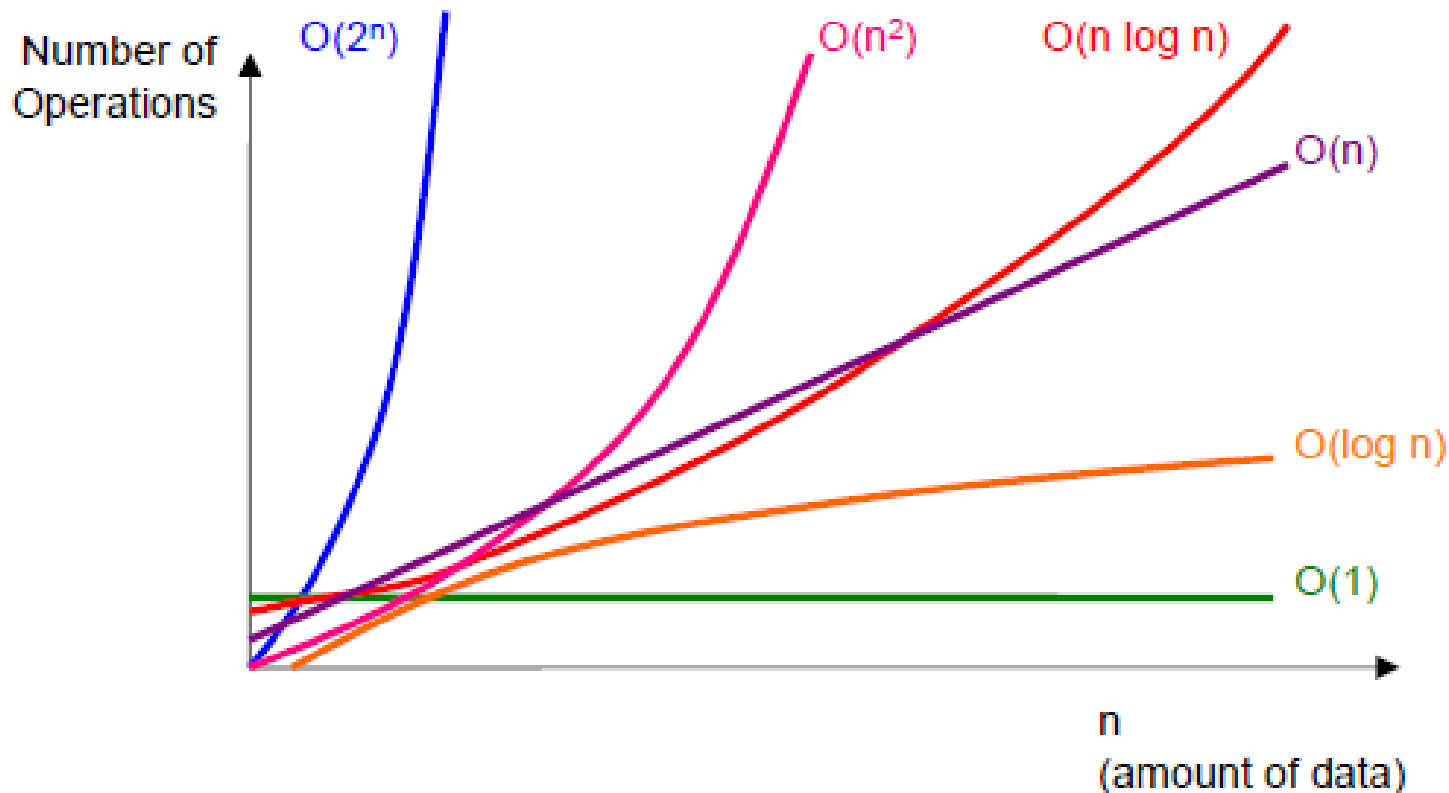
See you in Data Structure class, Algorithms class….!!!

Comparing Sorts using Time Complexity

- Selection Sort ($\rightarrow$ $O(n^2)$ 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:

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

- 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** ($\rightarrow$ $O(n*\log(n))$ 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
  - M개 원소의 2개 list를 merge: best case $\rightarrow$ (M/2)번 비교, worse case $\rightarrow$ (M-1)번 비교
  - 각 level에서 merge를 할때에 가장 worst한 경우에는 거의 n개의 비교를 해야할수 있다

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