CMPUT 175 Introduction to Foundations of Computing

Algorithm Analysis

Post-test

You should view the vignettes:

Objects and Classes

Fibonacci sequences

Objectives

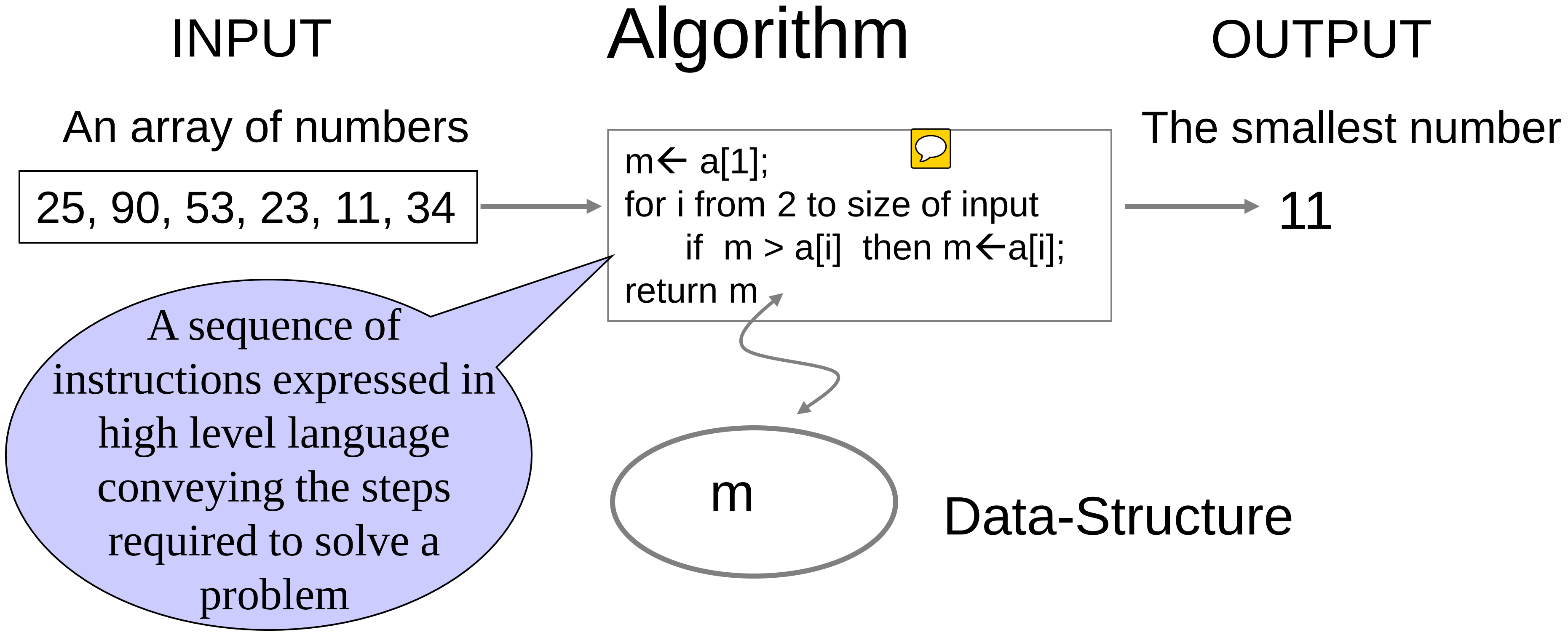
- Realizing the difference in performance between various algorithms solutions to the same problem
- Understanding algorithm analysis and realizing its importance in programming
- Understanding the notion of “Big-O” used to describe execution time for an algorithm
- Get a brief introduction to algorithm techniques
- Realizing the importance of right choices for data structures and methods with implementations with python.

What is Algorithm?

- Is a step-by-step procedure for solving a certain problem
- is any well-specified computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output after processing.
- is thus a sequence of computational steps that transform the input into the output.
- It is like a recipe that describes how to transform ingredients (input) into a dish (output)

Example: What is an Algorithm?

Problem: Input is a sequence of integers stored in a table.
Output the smallest number.



```
m← a[1];
for i from 2 to size of input
    if  m > a[i]  then m←a[i];
return m
```

INPUT
Algorithm
OUTPUT
An array of numbers
25, 90, 53, 23, 11, 34
The smallest number
11
A sequence of instructions expressed in high level language conveying the steps required to solve a problem
m
Data-Structure

What is a program ?

- A program is an implementation of an algorithm in a programming language
- There are many programming languages designed to communicate instructions to the computer
- There are many of such artificial languages, each with its own intricacies, specificities such as grammar and syntax
- So a program is a set of instructions which the computer will follow to solve a problem

Program vs. Algorithm

- Without an algorithm there can be no program
- There is an algorithm in any program that does something
- We should 1st express the algorithm in high level language before writing the program
- An algorithm can be implemented as a program in any programming language
- Example programming languages:
  - Old languages: Fortran, PL1, Cobol, Pascal, C, …
  - Common languages: Java, C^{++}, Python, perl, …

Algorithm

```
m← a[1];
for i from 2 to size of input
      if  m > a[i]  then m←a[i];
return m
```

Python

```
>>> numbers=[25,90,53,23,11,34]
>>> m=numbers[0]
>>> for i in range(1,len(numbers)):
            if m>numbers[i]:
                    m=numbers[i]

>>> print (m)
11
>>> min(numbers)
11
```

Pascal

```
var
  numbers : array[1..6] of integer;
  m, i: integer;
begin
  numbers[1]:=25;  numbers[2]:=90;  numbers[3]:= 53;
  numbers[4]:=23;  numbers[5]:=11;  numbers[6]:=34;
  m:=numbers[1];
 for i:=2 to Length(numbers) do
   begin
     if  m > numbers[i] then
     begin
        m:=numbers[i];
     end;
   end;
writeln(m);
end;
```

C

```
int main()
{
   int numbers[6]= {25,90,53,23,11,34}
   int i, m = numbers[0];

   for ( i = 1 ; i < sizeof(numbers) ; i++ )
   {
      if (m > numbers[i])
      {
        m = numbers[i];
      }
   }
   printf(m);
   return 0;
}
```

Assembler 8085

```
           LXI H,4200   ; Set pointer for array
           MOV B,M      ;  Load the Count
           INX  H        ; Set 1st element as largest data
           MOV A,M
           DCR B         ; Decremented the count
LOOP:  INX H
           CMP M         ; if A- reg < M go to AHEAD
           JC AHEAD
           MOV A,M       ; Set the new value as smalles
AHEAD:DCR B
           JNZ LOOP      ; Repeat comparisons till coun
           STA 4300      ; Store the largest value at 430
           HLT

GOAHEAD:
   ADD        SI,2
   INC        AX
   CMP        AX,6
   JL         TESTMIN
```

Perl

```
@numbers= (25,90,53,23,11,34);

$m=$numbers[0];
$m= $_<$m ? $_ : $m foreach (@numbers);
print $m;

print min(@numbers);
```

Algorithm

```
m← a[1];
for i from 2 to size of input
    if  m > a[i]  then m←a[i];
return m
```

Python

```
>>> numbers=[25,90,53,23,11,34]
>>> m=numbers[0]
>>> for i in range(1,len(numbers)):
        if m>numbers[i]:
                m=numbers[i]
>>> print (m)
11
>>> min(numbers)
11
```

Visual Basic

```
Dim numbers = New Integer() {25,90,53,23,11,34}

m=numbers(0)
For index = 1 To numbers.GetUpperBound(0)
    If m > numbers(index)  Then
        m = numbers(index)
    End If
Next

MsgBox(m)
```

PHP

```
<?php
$numbers = array(25,90,53,23,11,34);
$m=$numbers[0];
foreach ($numbers as $value) {
    if ($m > $value) {
        $m = $value;
    }
}
echo $m;
?>
```

Fortran

```
integer, dimension(6) :: numbers
integer :: i, m
numbers = (/ 25,90,53,23,11,34 /)
m= numbers(1)
do i = 2, size(numbers)
    if (m> numbers(i)) then
        m = numbers(i)
    endif
end do
print m
```

Java

```
int[] numbers= {25,90,53,23,11,34};
int m = numbers[0];
for(int i = 1;i<numbers.length;i++)
{
    if(m>numbers[i])
    {
        m = numbers[i];
    }
}
System.out.println(m);
```

Study of Algorithm

- How to analyze algorithms
  - How to measure the performance of algorithms
  - How to analyze an algorithm’s running time without coding it
- How to devise algorithms
  - Various techniques
- How to validate algorithms
- How to test programs

What do we analyze about them?

- Programs consume resources
- Algorithms require time for execution and space to store the data to be processed
- Analysis pertains to
  - Execution time: time complexity
  - Memory use: space complexity
- Look for the optimal solution: Is it possible to do better?

Comparing Algorithms

- Compare algorithms in terms of computing resources that each algorithm uses
- One algorithm is better than the other because it is more efficient in its use of resources or uses less resources
- Benchmark analysis
  - Execution time = running time
  - Memory usage

Problem: summing the first n numbers

```
def myfunction(something):
    me=0
    for alpha in range(1,something+1):
        beta=me+alpha
        me=beta
    return me
```

```
def sum1(n):
    theSum=0
    for i in range(1,n+1):
        theSum=theSum+i
    return theSum
```

- Which one is better?
- In terms of algorithms they are the same
- One program is more readable than the other, but their execution provides the same result. Also 1st one uses extra variable

Can we express it differently?

$S_n = 1 + 2 + \ldots + n = ?$

1 + 2 = 3

1 + 2 + 3 = 6

1 + 2 + 3 + 4 = 10

1 + 2 + 3 + 4 + 5 = 15

1 + 2 + 3 + 4 + 5 + 6 = 21

1 + 2 + 3 + 4 + 5 + 6 +7 = 28

1 + 2 + 3 + 4 + 5 + 6 +7 + 8 = 36

1 + 2 + 3 + 4 + 5 + 6 +7 + 8 + 9 = 45

$$S_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Rather than a loop

```
sum←0
for i from 1 to n
        sum←sum+i
```

We can have a formula

sum(n)=sum(n-1)+n

Or this formula

Sum(n)=n*(n+1)/2

Benchmarking

```
def sum1(n):
        theSum=0
        for i in range(1,n+1):
                theSum=theSum+i
        return theSum
```

```
def sum2(n):
        if n==0: return 0
        else: return  sum2(n-1) + n
```

sum(n)=sum(n-1)+n

```
def sum3(n):
        return (n*(n+1))/2
```

Sum(n)=n*(n+1)/2

```
import time
…
n=990
totalTime=0.0
for i in range(10):
        start=time.time()
        for j in range(100000): x=sum2(n)
        end=time.time()
        print("the sum is %d and it required %10.7f seconds"%(x,end-start))
        totalTime=totalTime+end-start

print("with sum2 the average time was %10.7f for n=%d"%(totalTime/10,n))
```

10 trials

Start time

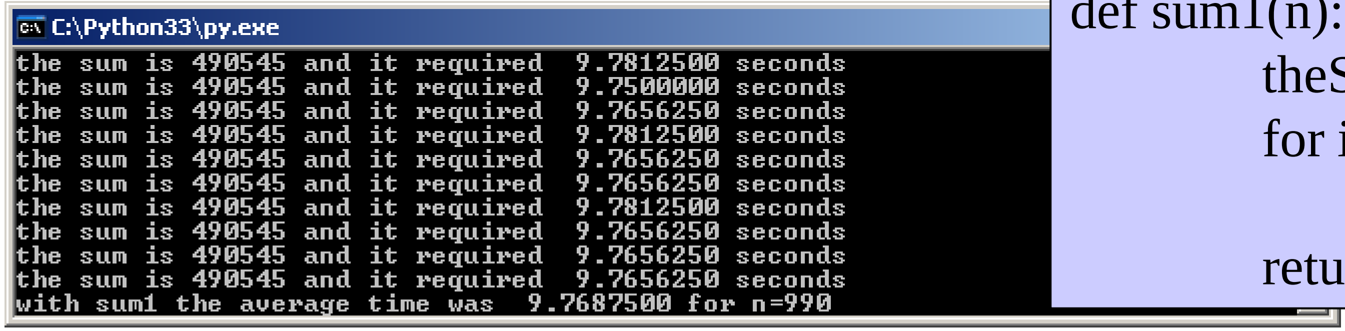
Calling 100,000 times the function

end time

Averaging over 10 trials

Benchmarking 2

Running the sum function 100,000 times averaged over 10 trials for n=990

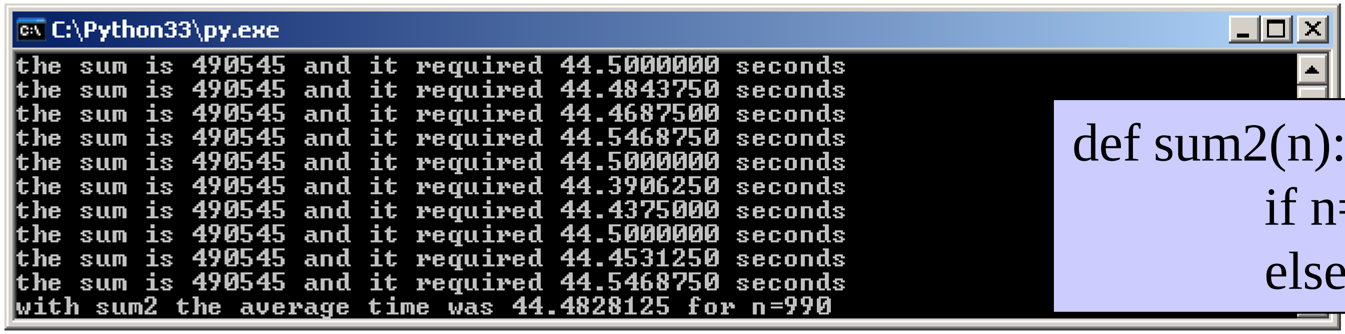


C:\Python33\py.exe

```
the sum is 490545 and it required  9.7812500 seconds
the sum is 490545 and it required  9.7500000 seconds
the sum is 490545 and it required  9.7656250 seconds
the sum is 490545 and it required  9.7812500 seconds
the sum is 490545 and it required  9.7656250 seconds
the sum is 490545 and it required  9.7656250 seconds
the sum is 490545 and it required  9.7812500 seconds
the sum is 490545 and it required  9.7656250 seconds
the sum is 490545 and it required  9.7656250 seconds
the sum is 490545 and it required  9.7656250 seconds
with sum1 the average time was  9.7687500 for n=990
```

```
def sum1(n):
        theSum=0
        for i in range(1,n+1):
                theSum=theSum+i
        return theSum
```

9.76 seconds

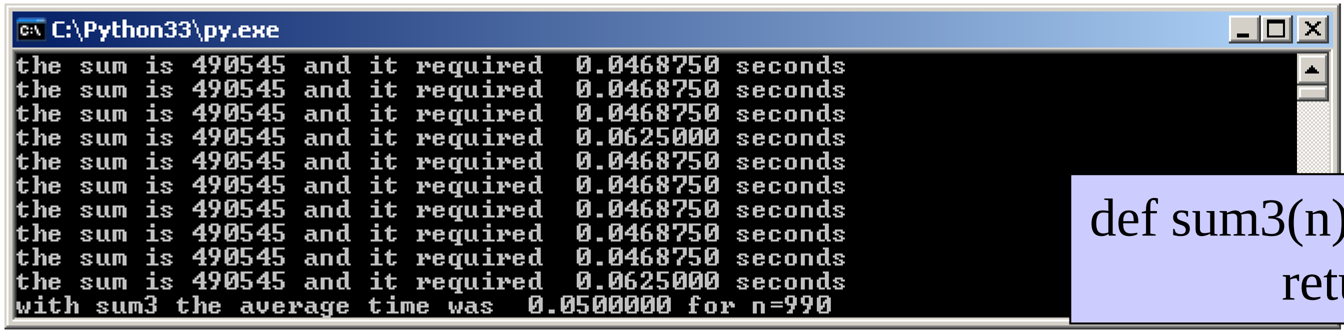


C:\Python33\py.exe

```
the sum is 490545 and it required 44.5000000 seconds
the sum is 490545 and it required 44.4843750 seconds
the sum is 490545 and it required 44.4687500 seconds
the sum is 490545 and it required 44.5468750 seconds
the sum is 490545 and it required 44.5000000 seconds
the sum is 490545 and it required 44.3906250 seconds
the sum is 490545 and it required 44.4375000 seconds
the sum is 490545 and it required 44.5000000 seconds
the sum is 490545 and it required 44.4531250 seconds
the sum is 490545 and it required 44.5468750 seconds
with sum2 the average time was 44.4828125 for n=990
```

```
def sum2(n):
        if n==0: return 0
        else: return  sum2(n-1) + n
```

44.48 seconds



C:\Python33\py.exe

```
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0625000 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0468750 seconds
the sum is 490545 and it required  0.0625000 seconds
with sum3 the average time was  0.0500000 for n=990
```

0.05 seconds

```
def sum3(n):
        return (n*(n+1))/2
```

Benchmarking 3

```
def sum1(n):
    theSum=0
    for i in range(1,n+1):
        theSum=theSum+i
    return theSum
```

```
n=100000000
totalTime=0.0
for i in range(5):
    start=time.time()
    x=sum1(n)
    end=time.time()
    print("the sum is %d and it required %10.7f seconds"%(x,end-start))
    totalTime=totalTime+end-start
print("with sum1 the average time was %10.7f for n=%d"%(totalTime/5,n))
```

Repeating **sum1** while varying n

n= 100,000

```
the sum is 5000050000 and it required  0.0156250 seconds
the sum is 5000050000 and it required  0.0156250 seconds
the sum is 5000050000 and it required  0.0000000 seconds
the sum is 5000050000 and it required  0.0156250 seconds
the sum is 5000050000 and it required  0.0156250 seconds
with sum1 the average time was  0.0125000 for n=100000
```

n= 1,000,000

```
the sum is 500000500000 and it required  0.1250000 seconds
the sum is 500000500000 and it required  0.1250000 seconds
the sum is 500000500000 and it required  0.1250000 seconds
the sum is 500000500000 and it required  0.1250000 seconds
the sum is 500000500000 and it required  0.1250000 seconds
with sum1 the average time was  0.1250000 for n=1000000
```

n= 10,000,000

```
the sum is 50000005000000 and it required  1.2343750 seconds
the sum is 50000005000000 and it required  1.2343750 seconds
the sum is 50000005000000 and it required  1.2500000 seconds
the sum is 50000005000000 and it required  1.2656250 seconds
the sum is 50000005000000 and it required  1.2187500 seconds
with sum1 the average time was  1.2406250 for n=10000000
```

n= 100,000,000

```
the sum is 5000000050000000 and it required 12.5156250 seconds
the sum is 5000000050000000 and it required 12.5312500 seconds
the sum is 5000000050000000 and it required 12.3437500 seconds
the sum is 5000000050000000 and it required 12.3125000 seconds
the sum is 5000000050000000 and it required 12.5312500 seconds
with sum1 the average time was 12.4468750 for n=100000000
```

- Each time n is multiplied by 10, the execution time is increased an order of magnitude

Benchmarking 4

```
def sum3(n):
    return (n*(n+1))/2
```

```
n=100000000
totalTime=0.0
for i in range(5):
    start=time.time()
    x=sum3(n)
    end=time.time()
    print("the sum is %d and it required %10.7f seconds"%(x,end-start))
    totalTime=totalTime+end-start
print("with sum3 the average time was %10.7f for n=%d"%(totalTime/5,n))
```

Repeating **sum3** while varying n

n= 100,000

```
the sum is 5000050000 and it required  0.0000000 seconds
the sum is 5000050000 and it required  0.0000000 seconds
the sum is 5000050000 and it required  0.0000000 seconds
the sum is 5000050000 and it required  0.0000000 seconds
the sum is 5000050000 and it required  0.0000000 seconds
with sum3 the average time was  0.0000000 for n=100000
```

n= 1,000,000

```
the sum is 500000500000 and it required  0.0000000 seconds
the sum is 500000500000 and it required  0.0000000 seconds
the sum is 500000500000 and it required  0.0000000 seconds
the sum is 500000500000 and it required  0.0000000 seconds
the sum is 500000500000 and it required  0.0000000 seconds
with sum3 the average time was  0.0000000 for n=1000000
```

n= 10,000,000

```
the sum is 50000005000000 and it required  0.0000000 seconds
the sum is 50000005000000 and it required  0.0000000 seconds
the sum is 50000005000000 and it required  0.0000000 seconds
the sum is 50000005000000 and it required  0.0000000 seconds
the sum is 50000005000000 and it required  0.0000000 seconds
with sum3 the average time was  0.0000000 for n=10000000
```

n= 100,000,000

```
the sum is 5000000050000000 and it required  0.0000000 seconds
the sum is 5000000050000000 and it required  0.0000000 seconds
the sum is 5000000050000000 and it required  0.0000000 seconds
the sum is 5000000050000000 and it required  0.0000000 seconds
the sum is 5000000050000000 and it required  0.0000000 seconds
with sum3 the average time was  0.0000000 for n=100000000
```

- Even though we increase n, the execution time doesn’t change. It is actually too fast to measure the time with time() on an AMD64 2.3Ghz machine

Benchmarking 5

Let’s slow down by calling sum3()

100000 times

```
n=100000000
totalTime=0.0
for i in range(5):
    start=time.time()
    for j in range(100000): x=sum3(n)
    end=time.time()
    print("the sum is %d and it required %10.7f seconds"%(x,end-start))
    totalTime=totalTime+end-start
print("with sum3 the average time was %10.7f for n=%d"%(totalTime/5,n))
```

```
def sum3(n):
    return (n*(n+1))/2
```

n= 100,000

```
the sum is 5000050000 and it required  0.0468750 seconds
the sum is 5000050000 and it required  0.0468750 seconds
the sum is 5000050000 and it required  0.0625000 seconds
the sum is 5000050000 and it required  0.0468750 seconds
the sum is 5000050000 and it required  0.0468750 seconds
with sum3 the average time was  0.0500000 for n=100000
```

n= 1,000,000

```
the sum is 500000500000 and it required  0.0468750 seconds
the sum is 500000500000 and it required  0.0468750 seconds
the sum is 500000500000 and it required  0.0468750 seconds
the sum is 500000500000 and it required  0.0468750 seconds
the sum is 500000500000 and it required  0.0468750 seconds
with sum3 the average time was  0.0468750 for n=1000000
```

n= 10,000,000

```
the sum is 50000005000000 and it required  0.0468750 seconds
the sum is 50000005000000 and it required  0.0468750 seconds
the sum is 50000005000000 and it required  0.0468750 seconds
the sum is 50000005000000 and it required  0.0468750 seconds
the sum is 50000005000000 and it required  0.0468750 seconds
with sum3 the average time was  0.0468750 for n=10000000
```

n= 100,000,000

```
the sum is 5000000050000000 and it required  0.0625000 seconds
the sum is 5000000050000000 and it required  0.0781250 seconds
the sum is 5000000050000000 and it required  0.0625000 seconds
the sum is 5000000050000000 and it required  0.0625000 seconds
the sum is 5000000050000000 and it required  0.0625000 seconds
with sum3 the average time was  0.0656250 for n=100000000
```

- Even though we increase n, the execution time doesn’t change. It remains relatively constant at 0.04 to 0.06 seconds for calling 100,000 times sum3(n)

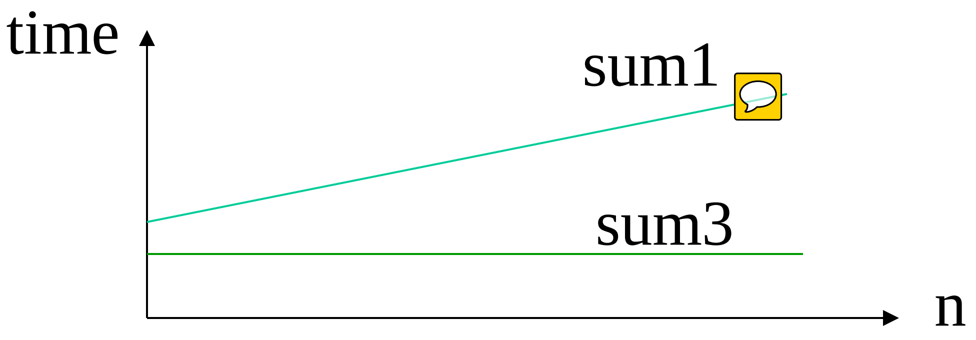
Running time complexity

- Time of **sum1(n)** increases linearly with n

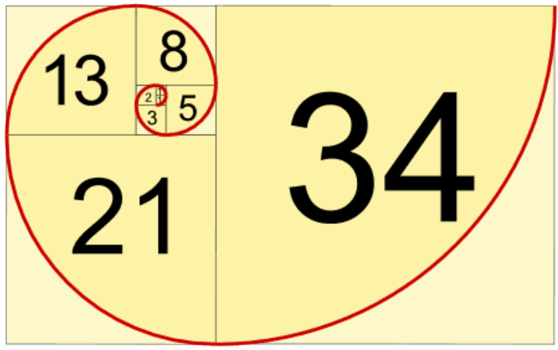
```
def sum1(n):
        theSum=0
        for i in range(1,n+1):
                theSum=theSum+i
        return theSum
```

- Time of **sum3(n)** remains constant regardless of n

```
def sum3(n):
        return (n*(n+1))/2
```



Problem: to compute Fibonacci number



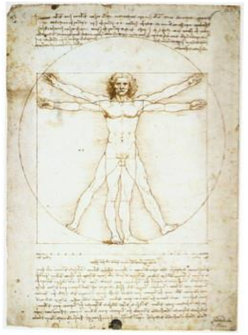
The Fibonacci numbers are the integer sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...,

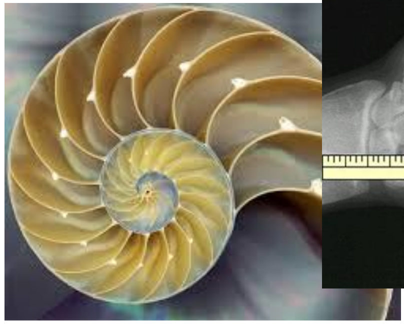
in which each number in the sequence is the sum of the previous two.

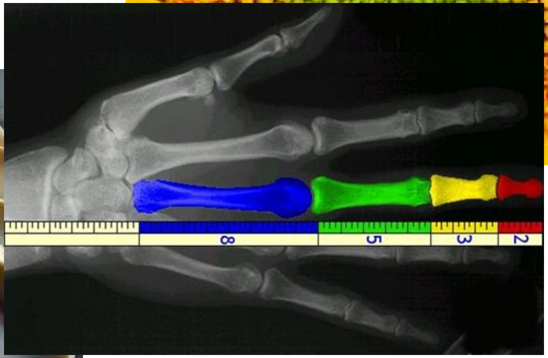
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

$F_0=0$
$F_1=1$
$F_n=F_{n-1}+F_{n-2}$



Da Vinci's Vitruvian man: Golden ratio related to Fibonacci sequence $F_n/F_{n-1} \approx 1.618$





An iteration program (bottom-up)

Algorithm

```
F(n) {
  a=0; b=1
  for i from 0 to n-1 {
        c = a + b
        a = b
        b = c
  }
  return a
}
```

```
def fibonacci(n):
    a = 0
    b = 1
    for i in range(n):
        c = a + b
        a = b
        b = c
    return a

print(fibonacci(35))
```

```
9227465
```

```
      a=0;  b=1
c=1;  a=1;  b=1
c=2;  a=1;  b=2
c=3;  a=2;  b=3
c=5;  a=3;  b=5
c=8;  a=5;  b=8
c=13; a=8;  b=13
c=21; a=13; b=21
c=34; a=21; b=34
c=55; a=34; b=55
c=89; a=55; b=89
…
```

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}.$$

A recursive program

Algorithm

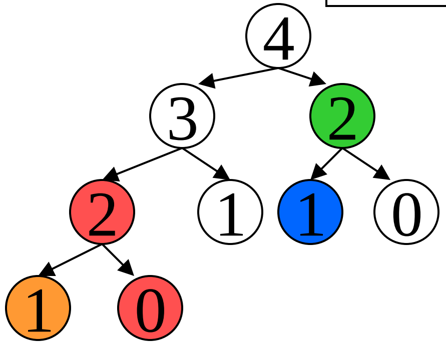
```
F(n) {
  if n==0 or n==1 return n
  else return F(n-1) + F(n-2)
}
```

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


print(fibonacci(35))
```

```
9227465
```

For example for fibonacci(4)
fibonacci(4) calls fibonacci(3) and fibonacci(2)
fibonacci(3) calls fibonacci(2) and fibonacci(1)
fibonacci(2) calls fibonacci(1) and fibonacci(0)
fibonacci(1) terminates with 1
fibonacci(0) terminates with 0
fibonacci(1) terminates with 1
fibonacci(2) calls fibonacci(1) and fibonacci(0)
fibonacci(1) terminates with 1
fibonacci(0) terminates with 0



A recursive program with cache

Algorithm

```
M[0]=0
M[1]=1
F(n) {
  if not exist M[n]
      M[n]= F(n-1) + F(n-2)
      return M[n]
}
```

```
memFibo = {0:0, 1:1}

def fib(n):
    if not n in memFibo:
        memFibo[n] = fib(n-1) + fib(n-2)
    return memFibo[n]

print(fib(35))
```

```
9227465
```

- Avoids recalculating numbers in the Fibonacci sequence that were already calculated

Another solution

Algorithm

Fibonacii(n)=F(n) = clossest integer of $\frac{\varphi^n}{\sqrt{5}}$

$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887498948$ also known as the Golden ratio

```
def fibonacci(n):
    inverseSqrt5 = 0.44721359549995793928183473374626
    phi = 1.6180339887498948482045868343656
    x=pow(phi,n)
    return int(round(x*inverseSqrt5))

print(fibonacci(35))
```

```
9227465
```

Which program is better?

- All four solutions are correct, but which one is the best? ➔ Measure the running time

fib1(): iterative fib2(): recursive fib3(): recursive with cache fib4(): Golden ratio

```
Fibbonaci(35) is 9227465. execution time with fib1() is  0.00000775187970 milliseconds
Fibbonaci(35) is 9227465. execution time with fib2() is  9.82390678195489 milliseconds
Fibbonaci(35) is 9227465. execution time with fib3() is  0.00000140225564 milliseconds
Fibbonaci(35) is 9227465. execution time with fib4() is  0.00000620676692 milliseconds
```

```
Fibbonaci(20) is 6765. execution time with fib1() is  0.00000614661654 milliseconds
Fibbonaci(20) is 6765. execution time with fib2() is  0.00727272180451 milliseconds
Fibbonaci(20) is 6765. execution time with fib3() is  0.00000149248120 milliseconds
Fibbonaci(20) is 6765. execution time with fib4() is  0.00000603759398 milliseconds
```

```
Fibbonaci(100) is 354224848179261915075. execution time with fib1() is  0.00001588345865 milliseconds
Fibbonaci(100) is 354224848179261915075. execution time with fib3() is  0.00000143609023 milliseconds
Fibbonaci(100) is 354224848179263111168. execution time with fib4() is  0.00000631954887 milliseconds
```

Error due to machine precision for floating points as the mantissa is approximated

Another Theorem for Fibonacci

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Can be proven by induction on *n*

So we take the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ to the power of n, then take the top right corner as F_n

Assuming we have a function to calculate the power of a 2x2 matrix matrixPower(). This function can be done by iterating, by diagonalization or **divide and conquer** strategy as we shall see later.

```
def fibonacci(n):
    (a,b,c,d) = matrixPower(1,1,1,0,n)
    return b

print(fibonacci(35))
```

```
9227465
```

Divide and Conquer for the Power function

X^n can be computed by multiplying X by itself n times → iteration

$$\underbrace{X * X * \ldots\ldots * X}_{\text{n times}}$$

or we can divide the problem to reduce the computation

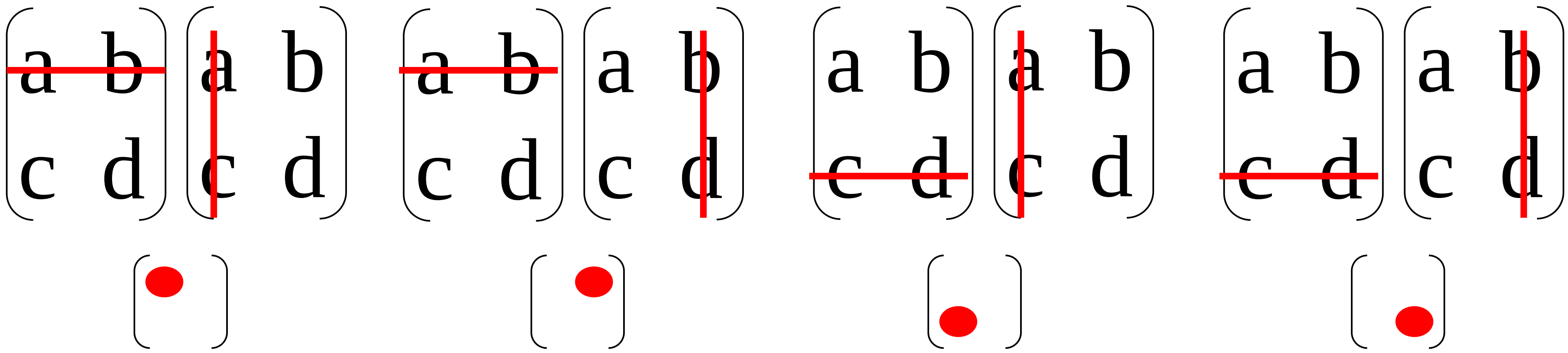
$$X^n = X^{\frac{n}{2}} \times X^{\frac{n}{2}} \qquad \text{if n is even}$$

$$X^{\frac{n-1}{2}} \times X^{\frac{n-1}{2}} \times X \qquad \text{if n is odd}$$

- When n is even we do the computation for half the sequence then multiply the result by itself. When n is odd we do almost the same and get the same savings

Reminder

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^2 = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} aa+bc & ab+bd \\ ca+dc & cb+dd \end{pmatrix}$$



Fibonacci with matrices

```
def matrixPower(a,b,c,d,n):
   if n<=1: return (1,1,1,0)
   elif n%2==0:
      # x^n = x^(n/2) * x^(n/2)
      (a1,b1,c1,d1)=matrixPower(a,b,c,d,n/2)
      a3=a1*a1+b1*c1
      b3=a1*b1+b1*d1
      c3=c1*a1+d1*c1
      d3=c1*b1+d1*d1
   else:
      # x^n = x^((n-1)/2) * x^((n-1)/2) * x
      (a1,b1,c1,d1)=matrixPower(a,b,c,d,(n-1)/2)
      a2=a1*a1+b1*c1  # x^((n-1)/2) * itself
      b2=a1*b1+b1*d1
      c2=c1*a1+d1*c1
      d2=c1*b1+d1*d1
      a3=a2*a+b2*c     # x^n = x^(n-1) * x
      b3=a2*b+b2*d
      c3=c2*a+d2*c
      d3=c2*b+d2*d

   return a3,b3,c3,d3
```

```
def fibonacii(n):
   (a,b,c,d) = matrixPower(1,1,1,0,n)
   return b
```

F(5)=?

$$\begin{pmatrix}1&1\\1&0\end{pmatrix}^5 = \underbrace{\underbrace{\underbrace{\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}1&1\\1&0\end{pmatrix}}_{\begin{pmatrix}2&1\\1&1\end{pmatrix}}\underbrace{\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}1&1\\1&0\end{pmatrix}}_{\begin{pmatrix}2&1\\1&1\end{pmatrix}}}_{\begin{pmatrix}5&3\\3&2\end{pmatrix}}\begin{pmatrix}1&1\\1&0\end{pmatrix}}_{\begin{pmatrix}8&\textcircled{5}\\5&3\end{pmatrix}}$$

F(5)=5

```
import timeit
…
t=timeit.Timer("fib5(35)","from __main__ import fib5")
f=fib5(35)
print("Fibbonaci(35) is %d. execution time with fib5() is"%(f), end=" ")
print (" %17.14f milliseconds"%(t.timeit(number=1)))
```

```
Fibbonaci(35) is 9227465. execution time with fib5() is   0.00001665037594 milliseconds
```

Performance of an Algorithm

- The performance (time) of an algorithm
  - Determined by **the number of basic operations** needed to solve the problem
  - Not by the actual running time of a program using the algorithm

  - Determined by **the size of the problem**
    - **Typically the number of data points n**

Growth Rate of an Algorithm

- The time performance is specified by

  T(n)

  - n is the size of the problem
  - T(n) is a function of n, representing the number of basic operations for the problem with size n
- The performance of an algorithm is determined by its behaviour at the large size of the problem (or as the size grows)
- The performance of an algorithm is determined by the growth rate of the number of operations with respect to the increase of the sizes of the problem

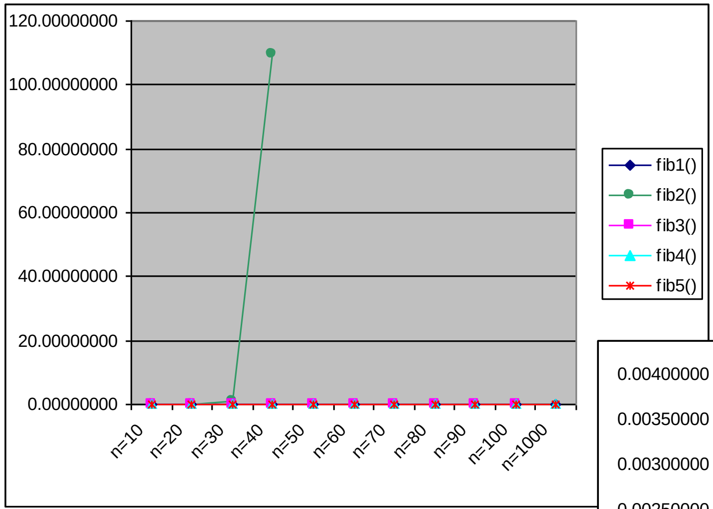
Different growth rates

- Consider the following functions
  - $T1(n) = 10$
  - $T2(n) = n$
  - $T3(n) = \log_2(n)$
  - $T4(n) = n^2$
  - $T5(n) = 2^n$
- Which one is best?
- Which one is worst?

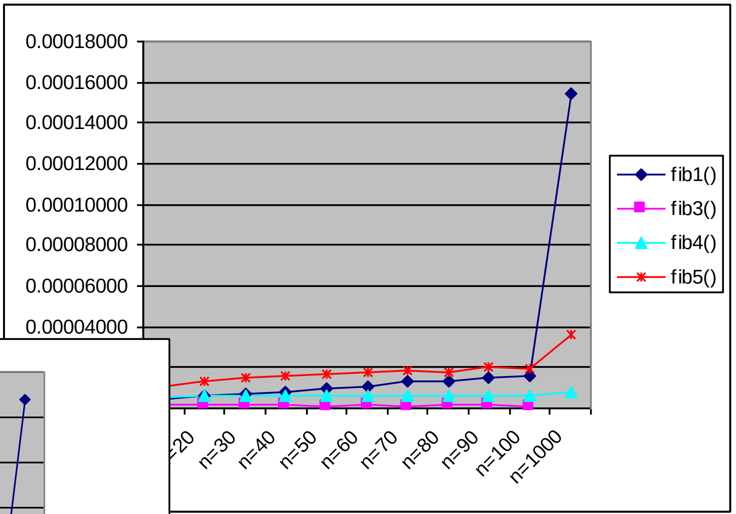
fib1(): iterative	$\rightarrow T_{fib1}(n)=n$
fib2(): recursive	$\rightarrow T_{fib2}(n)=\varphi^n$
fib3(): recursive with cache	$\rightarrow T_{fib3}(n)=n$
fib4(): Golden ratio	$\rightarrow T_{fib4}(n)=\log(n)$
fib5(): Matrix power	$\rightarrow T_{fib5}(n)=\log(n)$

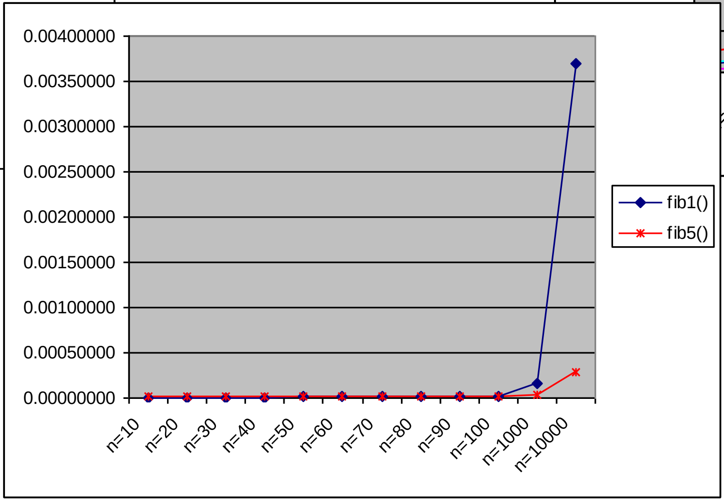
Comparative results

	n=10	n=20	n=30	n=40	n=50	n=60	n=70	n=80	n=90	n=100	n=1000	n=10000
fib1()	0.00000473	0.00000592	0.00000717	0.00000823	0.00000961	0.00001081	0.00001305	0.00001337	0.00001485	0.00001566	0.00015460	0.00368752
fib2()	0.00005934	0.00720515	0.90731057	109.52110087								
fib3()	0.00000133	0.00000138	0.00000139	0.00000139	0.00000132	0.00000136	0.00000129	0.00000137	0.00000135	0.00000132		
fib4()	0.00000567	0.00000582	0.00000612	0.00000600	0.00000595	0.00002308	0.00000605	0.00000623	0.00000610	0.00000620	0.00000765	
fib5()	0.00001074	0.00001360	0.00001494	0.00001551	0.00001684	0.00001801	0.00001875	0.00001763	0.00002058	0.00001939	0.00003592	0.00029137



fib1(): iterative
fib2(): recursive
fib3(): recursive with cache
fib4(): Golden ratio
fib5(): Matrix power





fib1(): $T_{fib1}(n)=n$
fib2(): $T_{fib2}(n)=\varphi^n$
fib3(): $T_{fib3}(n)=n$
fib4(): $T_{fib4}(n)=\log(n)$
fib5(): $T_{fib5}(n)=\log(n)$

Some could not continue
fib2(): Exponential
fib3(): recursion depth too deep
fib4(): error +
number too large

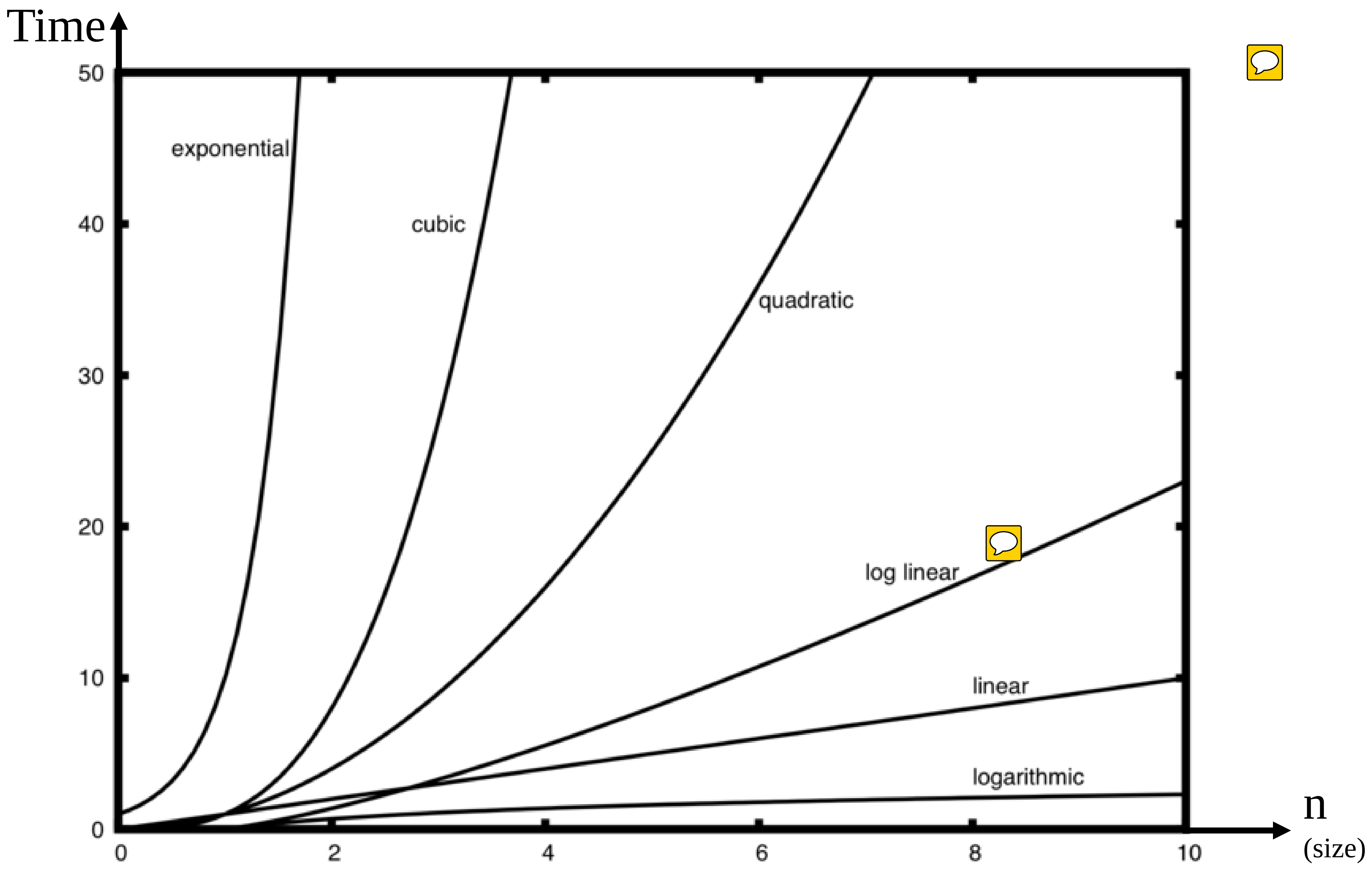
Multipliers

- Consider the following functions

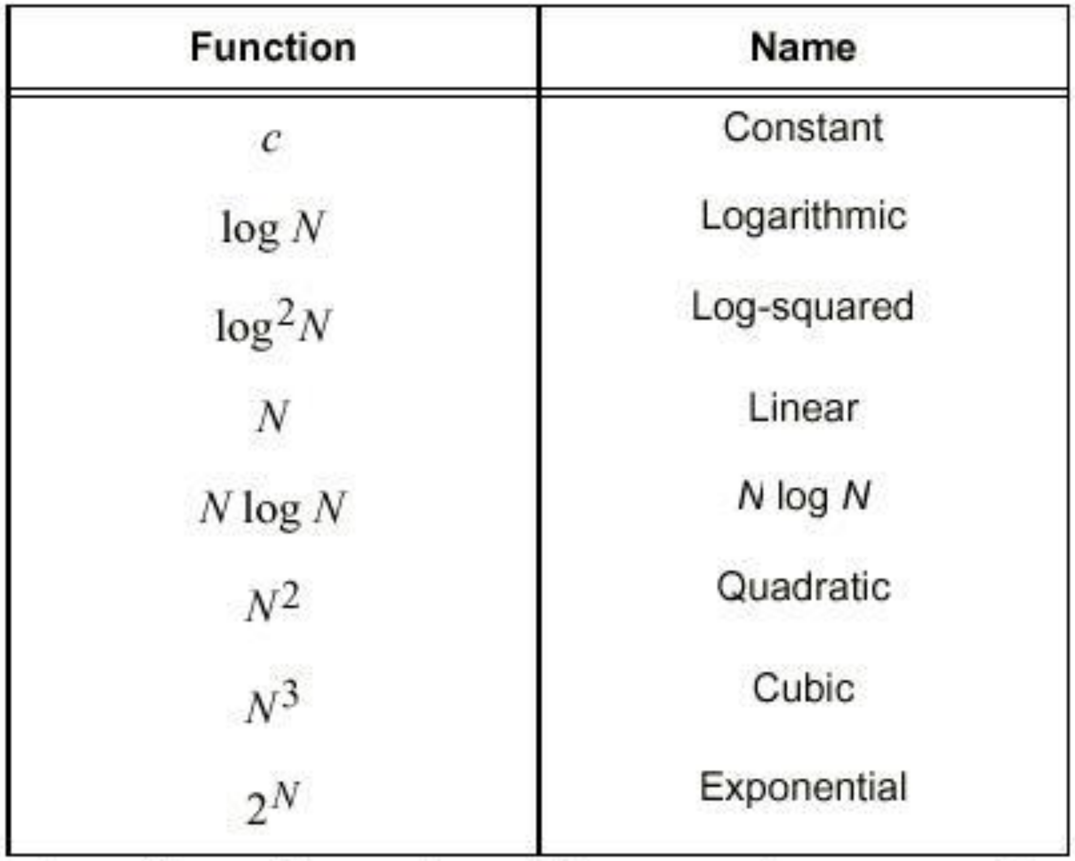
$T1(n) = 10$	$T6(n) = 40$	Constant time
$T2(n) = n$	$T7(n) = 3 * n$	linear
$T3(n) = \log_2(n)$	$T8(n) = 5 * \log_2(n)$	logarithmic
$T4(n) = n^2$	$T9(n) = 50 * n^2$	polynomial
$T5(n) = 2^n$	$T10(n) = 10 * 2^n$	exponential

- Any big difference between the left and the right columns?

Order of magnitude functions



Function of Growth rate



Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Asymptotic performance

- How does the algorithm behave as the problem size gets very large?

  $O(c)$

  $O(\log(n))$

  $O(n)$

  $O(n \log(n))$

  $O(n^2)$

  $O(n^3)$

  $O(2^n)$

Big-O notation

The time performance of algorithms

- The number of basic operations as a function of the problem size
- Represented as functions in order of increasing growth rate
- Denoted by

$$O(f(n))$$

Types of problems

- Easy problems
  - O(n)
  - Polynomial
- **Challenging problems**
  - No polynomial algorithms
  - Cannot prove that there are no polynomial algorithms
- Hard problems
  - There exist no polynomial algorithms

Algorithm Techniques

- Dynamic programming
  - Use a table to store intermediate results to avoid repeat computation
- Divided and conquer
  - Decompose a problem into two or more smaller problems and solve them separately
  - Combine solutions of smaller problems to solve the given problem
- Greedy Algorithm
  - Take the best one can get in each step
- Recursion

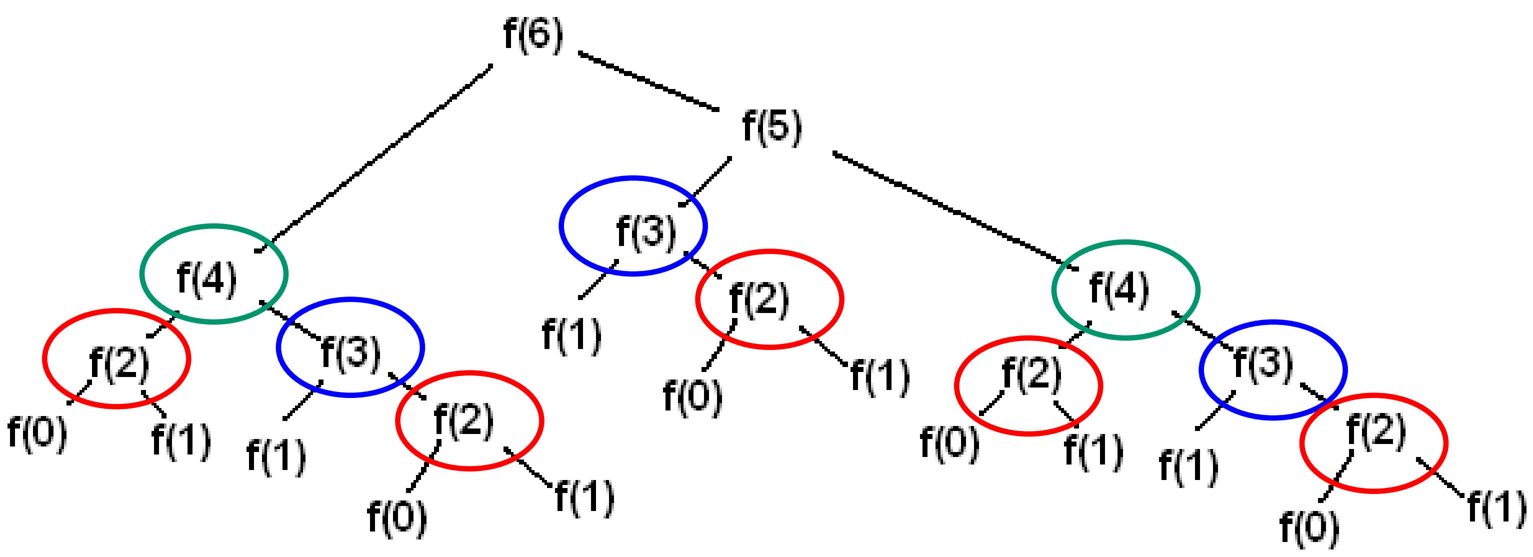
Dynamic programming

- Use a table to store intermediate results
- Avoid repeat computation

The Fibonacci numbers

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}.$$

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



The function	The number of calls
F(5)	1
F(4)	2
F(3)	3
F(2)	5
F(1)	7
F(0)	5

Using a cache

- What if we use a table to store all the intermediate results of f(m) for m < n?
- No more repeat computation
- Compromise Space for timing

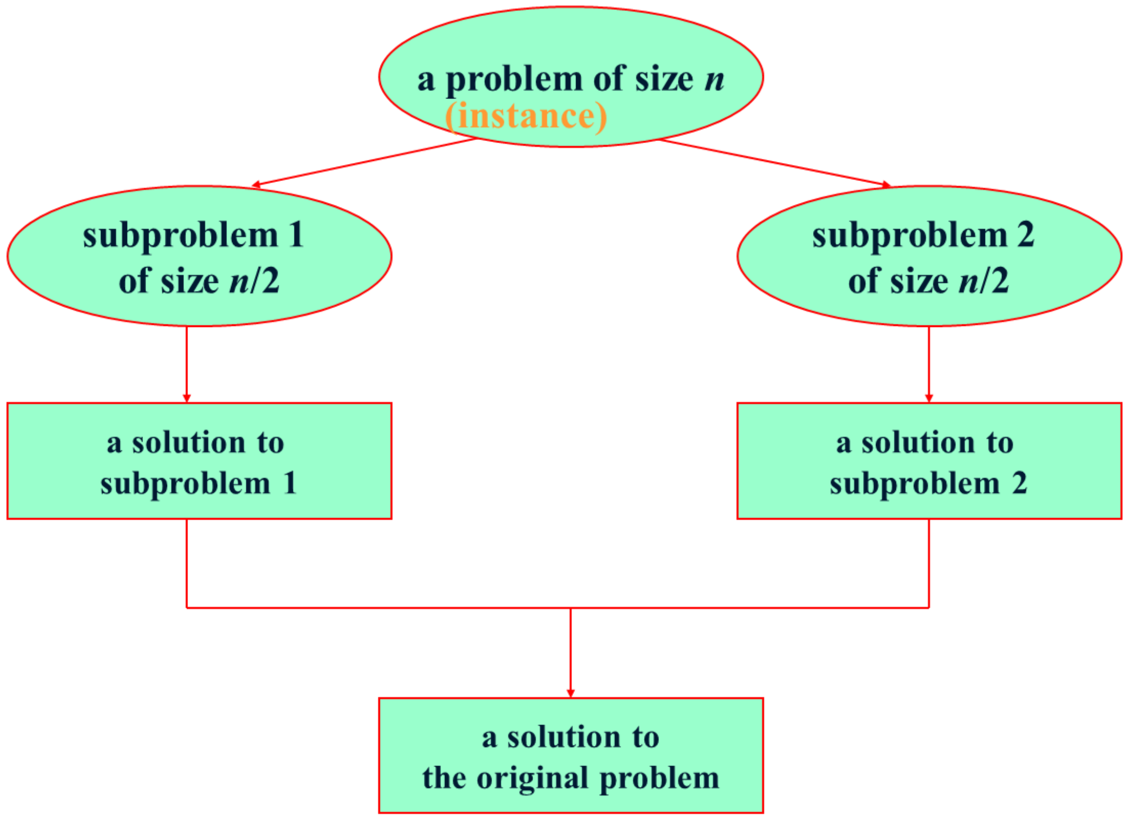
```
memFibo = {0:0, 1:1}

def fib(n):
    if not n in memFibo:
        memFibo[n] = fib(n-1) + fib(n-2)
    return memFibo[n]
```

Divide and Conquer

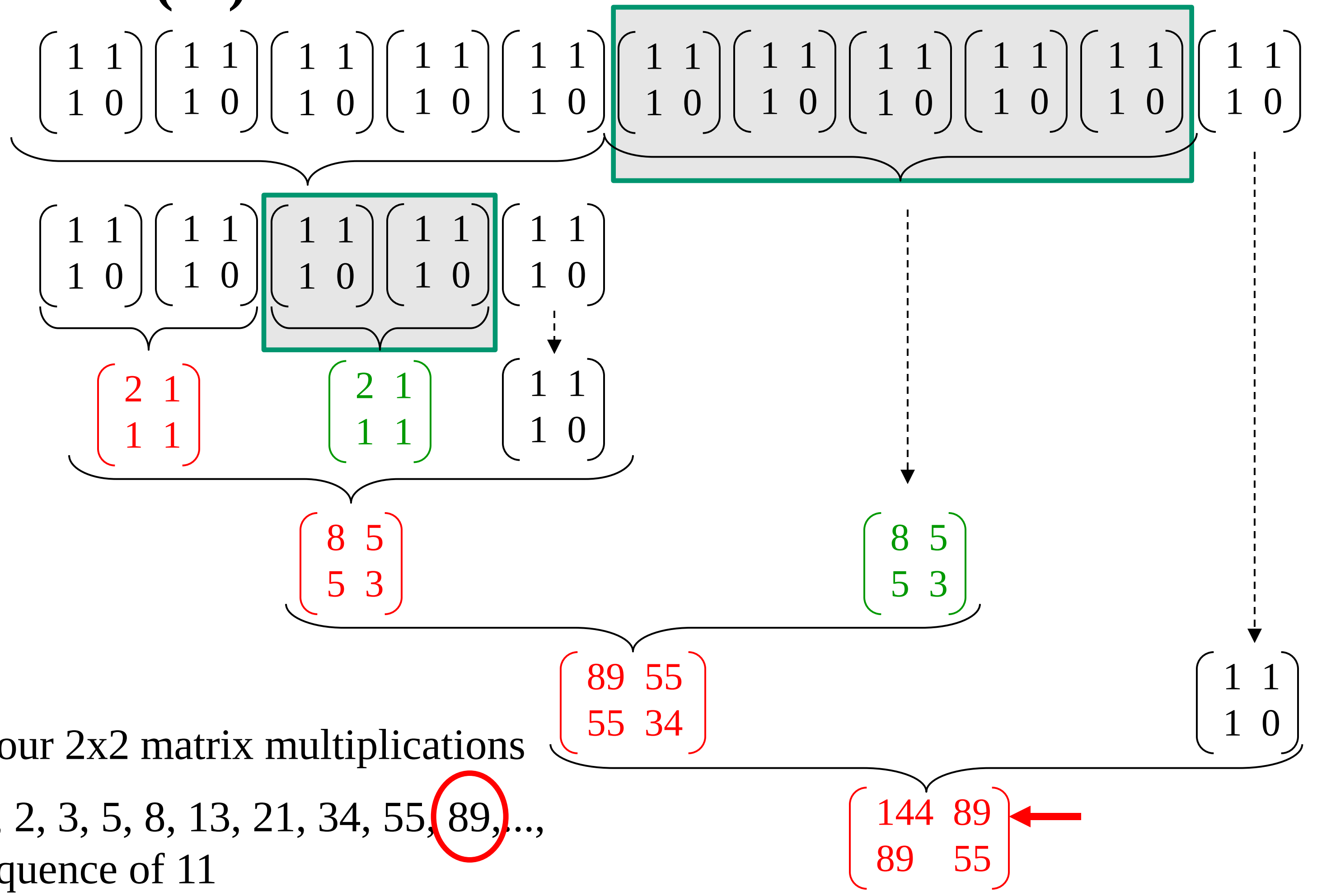
1. Divide instance of a problem into two or more smaller instances of the same type
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer (cont.)



Divide and Conquer (cont.)

Fibonacci(11)



$\begin{pmatrix}1&1\\1&0\end{pmatrix}$ ×11
$\begin{pmatrix}2&1\\1&1\end{pmatrix}$
$\begin{pmatrix}8&5\\5&3\end{pmatrix}$
$\begin{pmatrix}89&55\\55&34\end{pmatrix}$
$\begin{pmatrix}144&89\\89&55\end{pmatrix}$

Only four 2x2 matrix multiplications

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …, in a sequence of 11

Greedy Algorithm

- Optimization problems
  - An optimization problem is one in which one wants to find, not just a solution, but the best solution
- A greedy algorithm sometimes works well for optimization problems

Greedy Algorithm (cont.)

- A greedy algorithm works in steps.
- At each step
  - take the best one can get right now, without regarding the eventual optimization
  - Hope that by choosing a local optimum at each step, one will end up at a global optimum

Example: counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:

    - one $5 bill
    - one $1 coin, to make $6
    - one 25¢ coin, to make $6.25
    - one 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39

Running time and data structures

- The choice of data structure can make a difference in the running time of the implementation of an algorithm
- How the data structure is manipulated can also make a difference in the execution time
- Anything you have to iterate over → O(n)
- Accessing an indexed container → O(1)
- Sometimes it is subtile like when pop(0) at the beginning of a list requires shifting all elements ➔ O(n)

Example: ways to create a list

```
# Concatenate elements 1 by 1
def list1():
    myList= []
    for i in range(1000):
        myList = myList + [i]
```

```
# Append elements 1 by 1
def list2():
    myList = []
    for i in range(1000):
        myList.append(i)
```

```
# Comprehension
def list3():
    myList= [i for i in range(1000)]
```

```
# List range
def list4():
    myList = list(range(1000))
```

- They all do the same thing: creating a list myList with integers from 0 to 999
- Do they take the same time to initialize myList?
- Let's measure it

Benchmarking Lists

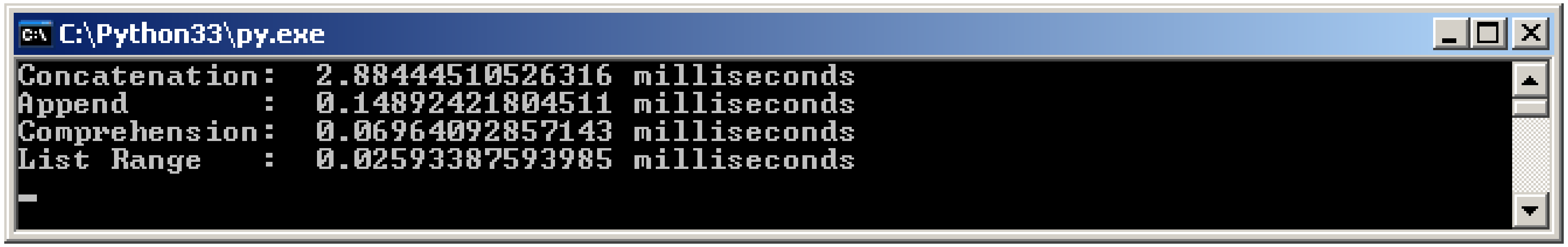
```
import timeit

…
t=timeit.Timer("list1()","from __main__ import list1")
print("Concatenation: %17.14f milliseconds"%(t.timeit(number=1000)))

t=timeit.Timer("list2()","from __main__ import list2")
print("Append       : %17.14f milliseconds"%(t.timeit(number=1000)))

t=timeit.Timer("list3()","from __main__ import list3")
print("Comprehension: %17.14f milliseconds"%(t.timeit(number=1000)))

t=timeit.Timer("list4()","from __main__ import list4")
print("List Range   : %17.14f milliseconds"%(t.timeit(number=1000)))
```



```
Concatenation:  2.88444510526316 milliseconds
Append       :  0.14892421804511 milliseconds
Comprehension:  0.06964092857143 milliseconds
List Range   :  0.02593387593985 milliseconds
_
```

Big-O Efficiency of Python List Operations

Operation	Big-O Efficiency
index []	**O(1)**
index assignment	**O(1)**
append	**O(1)**
pop()	**O(1)**
pop(i)	**O(n)**
insert(i,item)	**O(n)**
del operator	**O(n)**
iteration	**O(n)**
contains (in)	**O(n)**
get slice [x:y]	**O(k)**
del slice	**O(n)**
set slice	**O(n+k)**
reverse	**O(n)**
concatenate	**O(k)**
sort	**O(n log n)**
multiply	**O(nk)**