

Chapter 1

OBJ: main.py

What worked?

A standard for loop that quickly calculates the factorial of a whole number. Python doesn't have number size limitations so even large numbers like !99999 can be calculated within seconds.

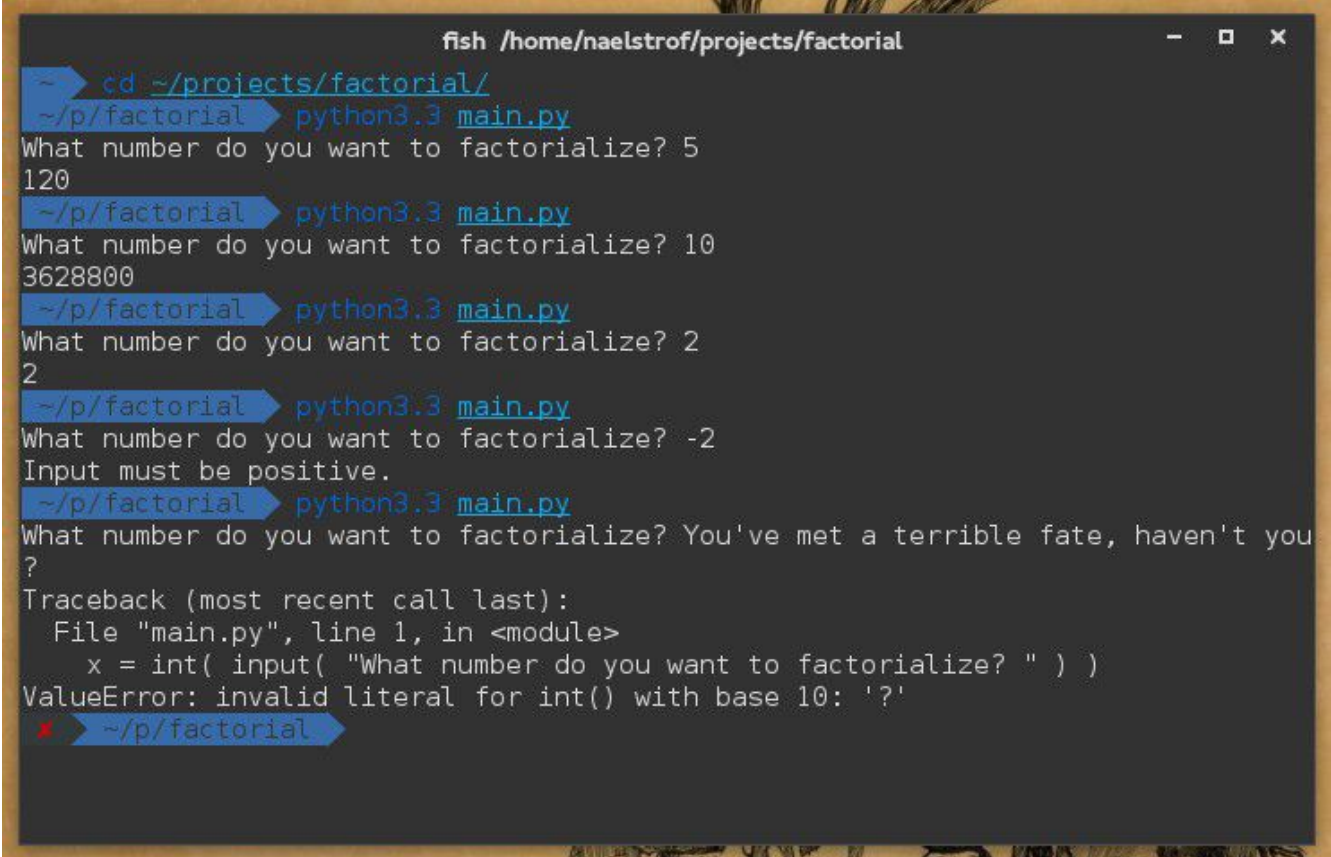
What didn't work?

I can't put letters or non-whole numbers into the input else it exits with an exception. Negative values are simply ignored.

CODE (Python)

```
x = int( input( "What number do you want to factorialize? " ) )
if x < 0:
    print( "Input must be positive" )
    exit()
f = x
for i in range( 1, x ):
    f *= i
print( f )
```

SCREENCAP



```
fish /home/naelstrof/projects/factorial
~ > cd ~/projects/factorial/
~/p/factorial > python3.3 main.py
What number do you want to factorialize? 5
120
~/p/factorial > python3.3 main.py
What number do you want to factorialize? 10
3628800
~/p/factorial > python3.3 main.py
What number do you want to factorialize? 2
2
~/p/factorial > python3.3 main.py
What number do you want to factorialize? -2
Input must be positive.
~/p/factorial > python3.3 main.py
What number do you want to factorialize? You've met a terrible fate, haven't you
?
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    x = int( input( "What number do you want to factorialize? " ) )
ValueError: invalid literal for int() with base 10: '?'
* > ~/p/factorial
```

Chapter 2

Project: Bacteria Growth

Was an interesting project because it was harder than it appeared. I started out with just finding the nearest power of two and using that for the time, but I realized the bacteria don't just suddenly double at the 23 minute mark. So I had to look up growth rates and Pert or $F = Pe^{(rt)}$. Using known values F, P, and t I could solve for r. Knowing R I could solve for t. Using the death value 7.3 million as F, I could easily find the amount of time it would take for bacteria to reach that number. After a bunch of confusing modifications I think I got a valid answer to a program that can calculate bacteria growths, even if they don't simply double.

Code: (python)

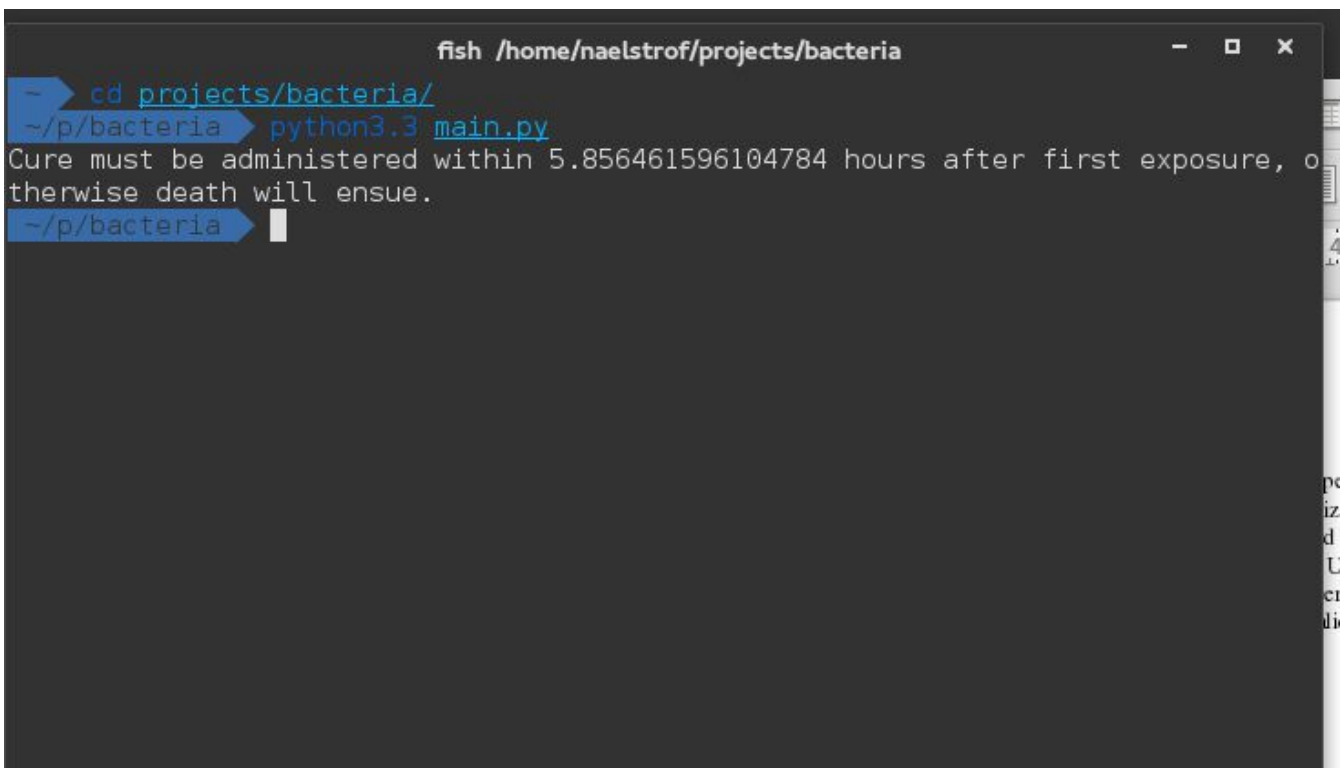
```
# Variables -----
killCap = 7.3 # in millions of bacteria
cureTime = 2.5 # in hours
doubleTime = 23 # in minutes
startingBacteria = 2
bacteriaAfterOneHour = 4

# Work -----
killCap = int( killCap * 1000000 )
growthRate = math.log( bacteriaAfterOneHour / startingBacteria, 10 ) / doubleTime
cureTime = cureTime * 60
killTime = math.log( killCap / startingBacteria, 10 ) / growthRate
print( killTime )

lastMinuteDiagnoseTime = killTime - cureTime

print( "Cure must be administered within", lastMinuteDiagnoseTime/60, "hours after first exposure, otherwise death will ensue." )
```

Screenshot (next page):



```
fish /home/naelstrof/projects/bacteria
~/p/bacteria ➤ cd projects/bacteria/
~/p/bacteria ➤ python3.3 main.py
Cure must be administered within 5.856461596104784 hours after first exposure, otherwise death will ensue.
~/p/bacteria ➤
```

Chapter 3

Bayes Theorem

What worked?

I used object oriented programming to create a tree node structure, with functions that can calculate probabilities from leaves of the tree. It supports as many events as needed and can calculate conditional probabilities using bayes theorem. It identifies events by string name. Everything works really elegantly besides actually setting up the data structure. I couldn't think of a better way to set it up at the time and now I'm too lazy to change it.

Code

```
# calculates the probability of an event happening.
def calcProb( event ):
    e = event
    n = e.probability
    while e.parent != None and e.parent.name != None:
        e = e.parent
        n = n * e.probability
    return n

# an event node tree that supports many events with independant probability values.
class event:
    name = None
    events = None
    probability = None
    parent = None
    def __init__( self, na, p ):
        self.name = na
        self.probability = p
        self.events = []
    def addEvent( self, na, p ):
        e = event( na, p )
        e.setParent( self )
        self.events.append( e )
    def getEvent( self, na ):
        for i in self.events:
            if i.name == na:
                return i
    def findEvents( self, na ):
        L = []
        for i in self.events:
            if i.name == na:
                L.append( i )
                break
            if len( i.events ) > 0:
                L = L + i.findEvents( na )
        return L
    def setParent( self, p ):
        self.parent = p
# calculates the probability of an event happening with a condition that must happen, using bayes therom.
def calcConProb( self, eventname, condition ):
    # find the first event that we are calculating the probability of.
```

```

e = self.findEvents( eventname )[0]
# then retrieve the conditional part of the event
# the numenator part of bayes therom
n = calcProb( e.findEvents( condition )[0] )
# now the denominator part of bayes therom
L = self.findEvents( condition )
d = 0
for i in L:
    d = d + calcProb( i )
# then finally return the probability
return n/d

```

there's probably a much better way to set up this data structure, but I can't imagine assignments being
much more complicated than this, and I'm too lazy to change it.

```

e = event( None, None )
e.addEvent( "Econ Grows", 0.7 )
e.addEvent( "Econ Slows", 0.3 )
e.getEvent( "Econ Grows" ).addEvent( "Stock Up", 0.8 )
e.getEvent( "Econ Grows" ).addEvent( "Stock Down", 0.2 )
e.getEvent( "Econ Slows" ).addEvent( "Stock Up", 0.3 )
e.getEvent( "Econ Slows" ).addEvent( "Stock Down", 0.7 )
print( "Given that the stock went up, the probability that the economy grew is", e.calcConProb( "Econ Grows",
"Stock Up" ) )

```

```

e = event( None, None )
e.addEvent( "France", 2/9 )
e.addEvent( "UK", 3/9 )
e.addEvent( "Canada", 4/9 )
e.getEvent( "France" ).addEvent( "Boy", 1/2 )
e.getEvent( "France" ).addEvent( "Girl", 1/2 )
e.getEvent( "UK" ).addEvent( "Boy", 1/3 )
e.getEvent( "UK" ).addEvent( "Girl", 2/3 )
e.getEvent( "Canada" ).addEvent( "Boy", 1/4 )
e.getEvent( "Canada" ).addEvent( "Girl", 3/4 )

```

```

print( "Given that the student is a boy, the probability of that boy being french is", e.calcConProb( "France",
"Boy" ) )

```

```

e = event( None, None )
e.addEvent( "Have disease", 1/100 )
e.addEvent( "No disease", 99/100 )
e.getEvent( "Have disease" ).addEvent( "Test positive", 90/100 )
e.getEvent( "Have disease" ).addEvent( "Test negative", 10/100 )
e.getEvent( "No disease" ).addEvent( "Test positive", 15/100 )
e.getEvent( "No disease" ).addEvent( "Test negative", 85/100 )

```

```

print( "Given that the patient tested positive, the probability of the patient actually being diseased is",
e.calcConProb( "Have disease", "Test positive" ) )

```

Screenshot:

```
Terminal - fish /home/naelstrof/projects/bayes
~> cd projects/bayes
~/p/bayes> python3.3 main.py
Given that the stock went up, the probability that the economy grew is 0.8615384615384616
Given that the student is a boy, the probability of that boy being french is 0.3333333333333333
Given that the patient tested positive, the probability of the patient actually being diseased is 0.05714285714285715
~/p/bayes> |
```

Chapter 4

Caesar Cipher

OBJ: main.py

What worked?

I used the built-in function "eval" to allow user input of functions. Its easy and fast, but there was no way of easily getting the inverse of the given algorithm. So basically my application assumes you know what you're doing and just blindly applies the algorithm to all the characters given to it. To decrypt something you just have to get the inverse of whatever algorithm you used and input it.

Code:

```
def encrypt( L, a ):
    L2 = ""
    for i in L:
        if i == ' ':
            L2 = L2 + ' '
            continue
        x = ord( i )
        foo = round( eval( a ) )
        L2 = L2 + chr( foo )
    return L2

print( "Enter string to modify (hello world): " )
L = input()
print( "Enter algorithm to use (x + 3): " )
a = input()
print( encrypt( L, a ) )
```

Screenshot:

fish /home/naelstrof/projects/encrypt

```
~ > cd projects/encrypt/
~/p/encrypt > python3.3 main.py
Enter string to modify (hello world):
hello world
Enter algorithm to use (x + 3):
2*x+3
ÓÍÛÚá ñácÛË
~/p/encrypt > python3.3 main.py
Enter string to modify (hello world):
ÓÍÛÚá ñácÛË
Enter algorithm to use (x + 3):
(x-3)/2
hello world
~/p/encrypt >
```

Chapter 5

Project: Trip Through Germany

I found this project incredibly unenjoyable due to various reasons. First of all some of the city names were misspelled which caused lots of problems when I was trying to get their coordinates or when I was creating their neighboring traversable cities. I probably spent over 6 hours just getting the database set up enough so I can actually create an algorithm based on it. I had to work far too much manually adding information into the database.

After all this strife and toil when I finally actually got to the programming part I didn't even have time to get it working the way I wanted. Giving me less than optimal results, but results non-the-less.

Really wish the project involved less database creation and more programming.

Code: python, main.py

```
import sqlite3 as sql
```

```
import sys
```

```
import json
```

```
import string
```

```
import math
```

```
cities = []
```

```
class City:
```

```
    def __init__( self, id, name, neighbors, latitude, longitude, extrafees, extratime, extratravel ):
```

```
        self.id = id
```

```
        self.name = name
```

```
        self.latitude = latitude
```

```
        if neighbors != None:
```

```
            self.neighbors = json.loads( neighbors )
```

```
        else:
```

```
            self.neighbors = None
```

```
        self.costs = None
```

```
        self.distances = None
```

```
        self.times = None
```

```
        self.money = None
```

```
        self.longitude = longitude
```

```
        self.extrafees = extrafees
```



```
self.extratime = extratime
self.extratransport = extratransport
```

```
# Uses haversine formula to approximate distance
# from latitude and longitudes
```

```
def distance( lonA, latA, lonB, latB ):
```

```
    r = 6371 # radius of the earth in km
```

```
    a = math.radians( latA )
```

```
    b = math.radians( latB )
```

```
    da = math.radians( latB-latA )
```

```
    db = math.radians( lonB-lonA )
```

```
    e = math.sin( da/2 ) * math.sin( da/2 ) + math.cos( a ) * math.cos( b ) *
```

```
math.sin( db/2 ) * math.sin( db/2 )
```

```
    f = 2 * math.atan2( math.sqrt( e ), math.sqrt( 1-a ) )
```

```
    return r * f # returns in km
```

```
def getDistance( cityA, cityB ):
```

```
    return distance( cityA.longitude, cityA.latitude, cityB.longitude, cityB.latitude ) +
cityB.extratransport
```

```
def getCost( cityA, cityB ):
```

```
    dis = distance( cityA.longitude, cityA.latitude, cityB.longitude, cityB.latitude ) +
cityB.extratransport
```

```
    # assume a taxi is the only available form of travel
```

```
    # and that it costs 2 Euros per km
```

```
    money = dis * 2 + cityB.extrafees
```

```
    # time in hours, assume taxi's move at a solid 130km/h
```

```
    time = dis / 130 + cityB.extratime
```

```
    return dis + money + time
```

```
def getMoney( cityA, cityB ):
```

```
    dis = distance( cityA.longitude, cityA.latitude, cityB.longitude, cityB.latitude ) +
cityB.extratransport
```

```
    # assume a taxi is the only available form of travel
```

```
    # and that it costs 2 Euros per km
```

```
    money = dis * 2 + cityB.extrafees
```

```
    # time in hours, assume taxi's move at a solid 130km/h
```

```
    time = dis / 130 + cityB.extratime
```

```
return money
```

```
def getTime( cityA, cityB ):
```

```
    dis = distance( cityA.longitude, cityA.latitude, cityB.longitude, cityB.latitude ) +  
    cityB.extratransit
```

```
    # assume a taxi is the only available form of travel
```

```
    # and that it costs 2 Euros per km
```

```
    money = dis * 2 + cityB.extrafees
```

```
    # time in hours, assume taxi's move at a solid 130km/h
```

```
    time = dis / 130 + cityB.extratime
```

```
    return time
```

```
def traverse( city, cities, visited, money, time, dist ):
```

```
    # Get a neighbor that has a low cost and hasn't been visited
```

```
    lowest = None
```

```
    lowestcity = None
```

```
    for i in range( 0, len( city.neighbors ) ):
```

```
        if lowest is None or ( city.costs[i] < lowest and not city.neighbors[i] in visited ):
```

```
            lowest = city.costs[i]
```

```
            lowestcity = city.neighbors[i]
```

```
    # If we couldn't find a city to continue traversing, we just use
```

```
    # the lowest cost traversal we can find within the remaining set
```

```
    if lowestcity in visited:
```

```
        lowest = None
```

```
        lowestcity = None
```

```
        for i in range( 0, len( cities ) ):
```

```
            if lowest is None or ( city != cities[i] and getCost( city, cities[i] ) <= lowest and
```

```
not cities[i] in visited ) or lowestcity in visited:
```

```
                lowest = getCost( city, cities[i] )
```

```
                lowestcity = cities[i]
```

```
    if lowestcity in visited:
```

```
        return [money, time, dist]
```

```
    money += getMoney( city, lowestcity )
```

```
    time += getTime( city, lowestcity )
```

```
    dist += getDistance( city, lowestcity )
```

```
    visited.append( city )
```

```
    print( city.name, "-> ", end="" )
```

```
    if len( cities ) == len( visited ):
```

```
    return [money, time, dist]
return traverse( lowestcity, cities, visited, money, time, dist )
```

```
# Meant to replace names with references to the actual cities
# this makes it easier to manipulate them
```

```
def solveNeighbors( cities ):
    for i in cities:
        for o in range( 0, len( i.neighbors ) ):
            foundMatch = False
            for p in cities:
                if p.name == i.neighbors[o]:
                    i.neighbors[o] = p
                    foundMatch = True
            if not foundMatch:
                print( "Error: City not found: ", i.neighbors[o] )
                sys.exit( 1 )
```

```
# Assumes solveNeighbors was ran
```

```
def generateCosts( cities ):
    # Cost is distance + time + money
    for i in cities:
        i.costs = []
        i.times = []
        i.money = []
        i.distances = []
        for o in i.neighbors:
            dis = distance( i.longitude, i.latitude, o.longitude, o.latitude ) + o.extratavel
            # assume a taxi is the only available form of travel
            # and that it costs 2 Euros per km
            money = dis * 2 + o.extrafees
            # time in hours, assume taxi's move at a solid 130km/h
            time = dis / 130 + o.extratime
            i.costs.append( dis + time + money )
            i.distances.append( dis )
            i.times.append( time )
            i.money.append( money )
```

```

db = None
try:
    db = sql.connect( 'cities.db' )
    cur = db.cursor()
    cur.execute( 'SELECT * FROM `cities`' )
    data = cur.fetchone()
    while data is not None:
        cities.append( City( data[0], data[1], data[2], data[3], data[4], data[5], data[6],
data[7] ) )
        data = cur.fetchone()
except sql.Error as e:
    print( "SQLException: ", e.args[ 0 ] )
    sys.exit( 1 )
finally:
    db.close()

```

```

# This section was used to populate the table for neighbors
# completely useless now that its populated
#for i in cities:
    #print( "Who is " + i.name + "'s neighbors?:" )
    #neighborstring = input()
    #neighbors = neighborstring.split( " " )
    #print( "Using ", neighbors, "\n from ", i.neighbors )
    #i.neighbors = neighbors
    #db = sql.connect( 'cities.db' )
    #db.execute( 'UPDATE `cities` SET neighbors=? WHERE id=?;'
[ json.dumps( neighbors ), str( i.id ) ] )
    #db.commit()

```

```

# Generate distances for all the city connections
solveNeighbors( cities )
generateCosts( cities )
print( "A list of the cities, their neighbors, and their costs to visit the neighbors..." )
for i in cities:
    print( i.name )
    for o in range( 0, len( i.neighbors ) ):
        print( "\t", i.neighbors[o].name, i.costs[o] )
startingcity = None

```

```
for i in cities:
    if i.name == "Frankfurt":
        startingcity = i
        break
print( "Calculating greedy optimal path starting from Frankfurt..." )
cost = traverse( startingcity, cities, [], 0, 0, 0 )
print("")
print( "Using this path you would use", cost[0], "Euros, spend", cost[1], "hours
(traveling, resting ), and travel", cost[2], "km." )
```

Screenshot:

Nuremberg 516.0373946344242

Basel 5836.414024563227

Nuremberg

Dresden 1142.051984841326

Munich 513.3316508347838

Stuttgart 750.3637567507396

Leipzig 839.9625717968265

Kassel 902.7628909308453

Frankfurt 924.9658631913526

Mannheim 973.2801851469123

Karlsruhe 1011.482456956546

Dresden

Leipzig 529.7404395415844

Nuremberg 1176.2547604285746

Berlin 569.1023645240416

Leipzig

Dresden 521.7191024654821

Nuremberg 852.0283419197845

Kassel 1088.7552424904309

Hanover 1797.2800854670877

Berlin 576.634408394186

Berlin

Leipzig 583.1621699538334

Dresden 566.8301727357423

Hanover 2120.973570078954

Rostock 753.9967084238027

Hamburg 1364.440067256444

Lubeck 1147.1069520645813

Basel

Munich 1437.54267774654

Baden-Baden 507.6344849688602

Calculating greedy optimal path starting from Frankfurt...

Frankfurt -> Wiesbaden -> Mannheim -> Karlsruhe -> Baden-Baden -> Stuttgart

-> Nuremberg -> Munich -> Leipzig -> Dresden -> Berlin -> Rostock -> Lubec

k -> Hamburg -> Bremen -> Bremen -> Kassel -> Sankt Augustin -> Bonn -> Col

ogne -> Dusseldorf -> Hanover ->

Using this path you would use 7802.140826641246 Euros, spend 48.46977241015

863 hours (traveling, resting), and travel 3181.070413320623 km.

~/p/germany

Chapter 6

Modular Exponentiation

OBJ: main.py

What worked?

Reading and interpreting the pseudo-code from the book it was easy to implement functions for Algorithm 1 and 5 in python. I easily created a base converter function and used it within my modular exponentiation function using the concepts I learned from the book. Using the Modular Exponentiation I could easily solve problems 25-28 on page 255.

CODE(Python3.3)

```
import math

def base( n, b ):
    a = ""
    while n != 0:
        a = str( n % b ) + a
        n = math.floor( n / b )
    return a

def modExpo( b, n, m ):
    x = 1
    power = b % m
    for i in reversed( base( n, 2 ) ):
        if i == "1":
            x = ( x * power ) % m
            power = ( power * power ) % m
    # x = b ^ n % m
    return x

print( "4.2.25: (7 ^ 644)%645 =", modExpo( 7, 644, 645 ) )
print( "4.2.26: (11 ^ 644)%645 =", modExpo( 11, 644, 645 ) )
print( "4.2.27: (3 ^ 2003)%99 =", modExpo( 3, 2003, 99 ) )
print( "4.2.28: (123 ^ 1001)%101 =", modExpo( 123, 1001, 101 ) )
```

SCREENSHOT

```
Terminal - fish /home/naelstrof/projects/modularexponentiation - □ ×
~> cd projects/modularexponentiation/
~/p/modularexponentiation > python3.3 main.py
4.2.25: (7^644)%645 = 436
4.2.26: (11^644)%645 = 1
4.2.27: (3^2003)%99 = 27
4.2.28: (123^1001)%101 = 22
~/p/modularexponentiation > |
```


Chapter 7

Project: n-queens

Since my student ID is odd, I had to find all solutions to a specified board size to the n-queen problem. It proved quite difficult due to error in translation from my brain logic to code. What I mean by that is that even without looking up what backtracking was, I already had the idea of backtracking in my head. I just didn't know how to translate it into clean functioning code at first. The first iteration of my program simply solving ONE solution of the puzzle was a nasty nested while loop with confusing backtracking. It did work but it was really messy and didn't find ALL solutions. From here I looked up how other people coded it, I noticed they did it recursively instead and it made waaay more sense to me how to make it cleaner.

From there I had a function that could simply solve the puzzle once, but after much thought, trial, and error (hours of it) I realized the same function couldn't generate all the solutions at once. So I split up the tasks into solve(), and permutate(). The solve() function simply takes any partial or uncompleted puzzles and attempts to complete them. While the permutate() function takes completed puzzles and recursively permutes them by row, and then uses solve() to attempt to solve the new permutation. Since the permutations all happen linearly-- there's no chance for repeating solutions to end up in the final list of solutions. Thus with a simple permutate(solve()) my program generates all solutions.

However the implementation of the program is kinda bad it definitely isn't just a simple permutate(solve()). In other words the functions are not modular and are completely state dependant, and they require somewhat of a kick-start macroing to work properly. Which is a pretty big deal to me, but I'm running low on time and I really don't want to restructure/re-comment everything again.

Code:

```
solutions = []
class queen:
    x = 0
    y = 0
    def __init__( self, x, y ):
        self.x = x
        self.y = y
    def hasCollision( self, q ):
        # can't collide with itself
        if q == self:
            return False
        dx = abs( self.x - q.x )
        dy = abs( self.y - q.y )
        if dx == dy: # diagonals is the same
            return True
        if dx == 0: # row is the same
            return True
        if dy == 0: # column is the same
            return True
        return False

def findCollisions( queens, q ):
```

```

hc = False
for nq in queens:
    if q.hasCollision( nq ):
        hc = True
        break
return hc

```

takes an existing board of queens and attempts to recursively add more queens under the condition that they cannot be attacked

if it fails to add a queen it returns an empty table, otherwise it will return the completed puzzle

```

def solve( row, size, queens ):

```

just setting up some variables to use

```

    global solutions

```

if we have a completed solution

```

    if row >= size:

```

add it to the list of solutions

```

        solutions.append( queens )

```

```

        return queens

```

```

    for column in range( 0, size ):

```

we create a plain old queen in the specified area only to test it against the queens on the board already.

```

        q = queen( column, row )

```

```

        if findCollisions( queens, q ) == False:

```

if our position is safe, add the queen to the board!

```

            queens.append( q )

```

if we couldn't find a position for the next row, backtrack to a previous queen.

```

            if solve( row+1, size, queens ) == []:

```

```

                queens.pop()

```

```

            else:

```

since the above solve(row+1, size, queens) alters the queens table, its actually already a completed puzzle now!

return it since its complete.

```

                return queens

```

```

    return []

```

start the first iteration of the recursive permutation for us.

this is basically a macro to get the permutations running recursively correctly.

```

def permutateAll( queens ):

```

```

    permutate( 0, len( queens ), queens )

```

```

    for i in range( 1, len( queens ) ):

```

```

        permutate( i, len( queens ), queens )

```

recursively permutes a completed queen set starting by moving the queen at the specified row over by one safe-spot,

then using solve() to complete the new unique puzzle fragment

then recurses onto all the new unique generated rows

```

def permutate( row, size, queens ):

```

if we're asking to permutate a row that doesn't exist, exit.

```

    if row >= size:

```

return

copy the queen table to the specified row,

permutatedQueens = []

for i **in** range(0, row):

permutatedQueens.append(queens[i])

q = queens[row]

instead of copying everything directly, we mutate the specified row by attempting to move it along the board

permutated = False

for column **in** range(q.x+1, size):

q = queen(column, q.y)

if findCollisions(permutatedQueens, q) == False:

permutatedQueens.append(q)

permutated = True

if we couldn't move it any further along, due to collisions/running out of board space, we exit.

if not permutated:

return

now with our new permutated partial board, we attempt to solve the rest of the puzzle

test = solve(row+1, size, permutatedQueens)

if we couldn't solve it, we continue to permute the current row

if test == []:

permute(row, size, permutatedQueens)

return

if we succeeded, we not only do we continue to permute the current row

permute(row, size, permutatedQueens)

but we also recursively permute all the new rows that were generated

for i **in** range(row+1, size):

permute(i, size, test)

return

some silly code to generate a grid of blocks and Qs to represent the board and where the queens are.

might not be an accurate chess board

def printTable(queens):

size = len(queens)

toggle = True

table = []

for x **in** range(0, size):

row = []

for y **in** range(0, size):

if toggle:

row.append("■")

else:

row.append(" ")

toggle = **not** toggle

table.append(row)

if size % 2 == 0:

toggle = **not** toggle

```

for i in queens:
    table[i.y][i.x] = "Q"
for x in table:
    for y in x:
        print( y, end="" )
    print( "" )

# size of the board
size = 8

# generates a starting solution of the n-queens problem.
startingboard = solve( 0, size, [] )
# recursively permutes the solution to find all other solutions.
permutateAll( startingboard )

# print all the found solutions and number them
count = 0
for i in solutions:
    print( "Unique Solution #", count, "-----" )
    printTable( i )
    count += 1

# print some information
print( "Found", len( solutions ), "unique solutions for a " + str( size ) + "x" + str( size ) + " board." )
print( "This application can find all the solutions to different sized boards, just change the size = #
variable inside the code" )

```

Screenshot:

```
Q
 Q
  Q
   Q
    Q
   Q
  Q
 Q
```

Unique Solution # 89 -----

```
Q
 Q
  Q
   Q
    Q
   Q
  Q
 Q
```

Unique Solution # 90 -----

```
Q
 Q
  Q
   Q
    Q
   Q
  Q
 Q
```

Unique Solution # 91 -----

```
Q
 Q
  Q
   Q
    Q
   Q
  Q
 Q
```

Found 92 unique solutions for a 8x8 board.

This application can find all the solutions to different sized boards, just change the size = # variable inside the code

~/p/queens

Chapter 9

Recursive vs Iterative

OBJ: main.py

What worked?

I used python's built-in timing systems called "timeit", which let me easily and accurately time the different algorithms. What was interesting is that recursive vs iterative functions is situational.

Iterative fibonacci functions are faster than recursive ones, while recursive mod expo functions are far faster than my particular iterative mod expo. It could be due to the array allocations however.

CODE(Python)

```
# calcul eus
# calcul# h#
t
phrtlobai eyy#fi trv
ttt#fi tgg t. vl hudli t.
ttth#fi tgg t( vl hudli t(
ttth#fi hudli lobai eyy#fi t t( trt=lobai eyy#fi t: twtn
tt
phrtlobai eyy#fi trv
ttt! tg t.
ttt!tg t(
tttralt##tlei nhft( *i trv
tttttt_tg t! t=tN
tttttt! tg tN
tttttt!tg t_
ttt! hudli tN

phrtbe) hfti *btrv
ttt!tg t,
ttt[ s#n#i t]g t v
tttttttg t) ul fi tE tbt=te
tttttt! tg t eusflaalfti tkbtrv
ttt! hudli te

phrt ap>!caftb*i *t trv
ttt! tg t(
tttc a[ hl tg tbtEt
tttralt##tlhOhl) hpfbe) hfti *twtrv
tttttt##t#g g t( ,v
tttttttt! tg tft! t+tc a[ hl trtEt
tttttttc a[ hl tg tftc a[ hl t+tc a[ hl trtEt
ttt! Ct! tg tb/ i E
ttt! hudli t!

phrtl ap>!caftb*i *t trv
ttt##bt' gt. talt t' twalt' t' t v
tttttt! hudli t(
ttt#fi tgg t v
tttttt! hudli t(
ttth#fi tE twtg t. v
```

tttttt!tg tl ap>!caftb* eusflaalfti kwtr* tn
tttttt!hudli tft! t+t! trtEt
ttttth) hv
tttttt!tg tl ap>!caftb* eusflaalfti kwtr* tn
tttttt!hudli tftft! t+t! trtEt t+tbEt trtEt

cl#uft, l hydL) #htobai eyy#artw(tu# hv *thi pg, t, tn
ltg tu# h#fu# h#ftzobai eyy#tw(trzt) hudcg, rla t" e#" t# calutobai eyy#*i d bhlg(. . tn
cl#uft! #la)hyai p), tn

cl#uft, Ghleu#i tobai eyy#artw(tu# hv *thi pg, t, tn
#g tu# h#fu# h#ftzobai eyy#tw(trzt) hudcg, rla t" e#" t# calutobai eyy#*i d bhlg(. . tn
cl#uft# #la)hyai p), tn

#t# tl v
ttttcl#uft, Ghleu#htOhL) #i t[#)], tn
h) hv
ttttcl#uft, l hydL) #htOhL) #i t[#)], tn

cl#uft, l hydL) #htartf77777/ 77777rE(. (v *thi pg, t, tn
ltg tu# h#fu# h#ftz ap>!caft77777*77777*(. (trzt) hudcg, rla t" e#" t# calut ap>!ca, *i d bhlg(. . tn
cl#uft! #la)hyai p), tn

cl#uft, Ghleu#i tartf77777/ 77777rE(. (v *thi pg, t, tn
#g tu# h#fu# h#ftz ap>!caft77777*77777*(. (trzt) hudcg, rla t" e#" t# calut ap>!ca, *i d bhlg(. . tn
cl#uft# #la)hyai p), tn

#t# tl v
ttttcl#uft, Ghleu#htOhL) #i t[#)], tn
h) hv
ttttcl#uft, l hydL) #htOhL) #i t[#)], tn

S3I >>U38D

Terminal - fish /home/naelstrof/projects/recreation

- □ ×

~ cd projects/recreation/

~/p/recreation python3.3 main.py

Recursive fibonacci of 21 time: 0.6997285280012875 microseconds

Iteration fibonacci of 21 time: 0.00015808099851710722 microseconds

Iterative version wins!

Recursive of (99999^99999)%101: 0.0013390149997576373 microseconds

Iteration of (99999^99999)%101: 0.0018111279987351736 microseconds

Recursive version wins!

~/p/recreation

power = b % m

Chapter 10

Project: RSA encryption

What worked? My previous modular exponentiation algorithm worked perfectly to encrypt and decrypt the sentence "The Queen Can't Roll When Sand is in the Jar" in a reasonable timeframe.

I did have problems understanding the extended Euclidean algorithm due to wikipedia expecting too much background knowledge, however I did understand what modular multiplicative inverse is and was able to implement my own simple algorithm.

Code:

```
import math

def base( n, b ):
    a = ""
    while n != 0:
        a = str( n % b ) + a
        n = math.floor( n / b )
    return a

def modExpo( b, n, m ):
    x = 1
    power = b % m
    for i in reversed( base( n, 2 ) ):
        if i == "1":
            x = ( x * power ) % m
        power = ( power * power ) % m
    # x = b^n%m
    return x

# Should be using the Extended Euclidean algorithm, but here's my own easier to understand one :).
def invmod( a, m ):
    for i in range( 1, m ):
        if i*a % m == 1:
            return i
    raise ValueError( "a and m should be prime or something" )

# Not even sure what a "totient" is, but wikipedia said this is the function for it.
def totient( p, q ):
    return (p-1)*(q-1)

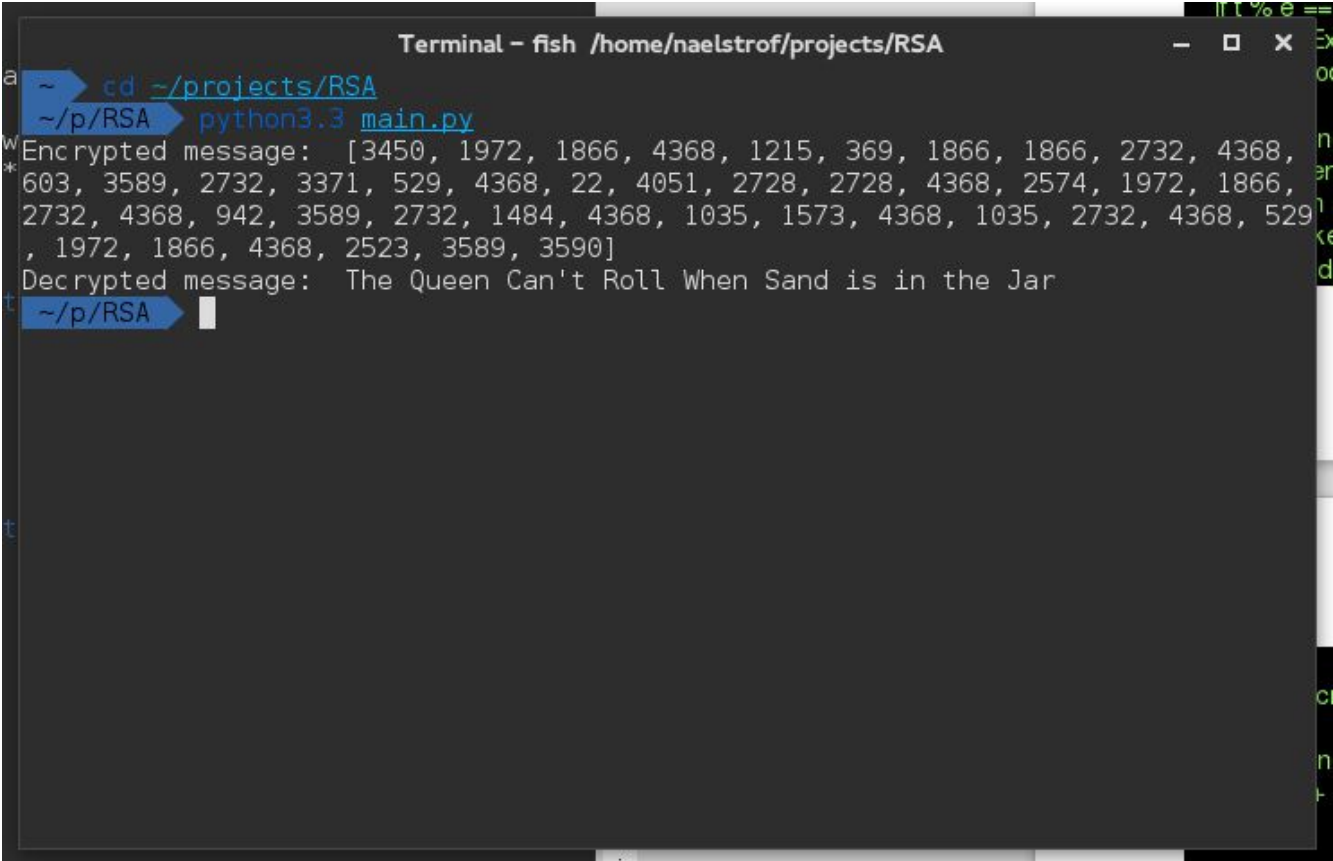
def RSAEncrypt( p, q, e, message ):
    n = p*q
    t = totient( p, q )
    if t % e == 0:
        raise Exception( "e can't be a divisor of the totient of p and q, has to be co-prime" )
    d = invmod( e, t )
    L = []
    for i in range( 0, len( message ) ):
        L.append( modExpo( ord( message[ i ] ), e, n ) )
    # d and n are the private key
    # public key is n and e
    return L, d, n
```

```
def RSADecrypt( n, d, message ):
    L = ""
    for i in range( 0, len( message ) ):
        L = L + chr( modExpo( message[ i ], d, n ) )
    return L

# d and n are the private key to decrypt
emessage, d, n = RSAEncrypt( 71, 67, 17, "The Queen Can't Roll When Sand is in the Jar" )

print( "Encrypted message: ", emessage )
print( "Decrypted message: ", RSADecrypt( n, d, emessage ) )
```

Screenshot:



The screenshot shows a terminal window titled "Terminal - fish /home/naelstrof/projects/RSA". The user navigates to the directory ~/projects/RSA and runs the command python3.3 main.py. The output shows the encrypted message as a list of integers: [3450, 1972, 1866, 4368, 1215, 369, 1866, 1866, 2732, 4368, 603, 3589, 2732, 3371, 529, 4368, 22, 4051, 2728, 2728, 4368, 2574, 1972, 1866, 2732, 4368, 942, 3589, 2732, 1484, 4368, 1035, 1573, 4368, 1035, 2732, 4368, 529, 1972, 1866, 4368, 2523, 3589, 3590]. The decrypted message is "The Queen Can't Roll When Sand is in the Jar".

```
Terminal - fish /home/naelstrof/projects/RSA
~ ➔ cd ~/projects/RSA
~/p/RSA ➔ python3.3 main.py
Encrypted message: [3450, 1972, 1866, 4368, 1215, 369, 1866, 1866, 2732, 4368,
603, 3589, 2732, 3371, 529, 4368, 22, 4051, 2728, 2728, 4368, 2574, 1972, 1866,
2732, 4368, 942, 3589, 2732, 1484, 4368, 1035, 1573, 4368, 1035, 2732, 4368, 529
, 1972, 1866, 4368, 2523, 3589, 3590]
Decrypted message: The Queen Can't Roll When Sand is in the Jar
~/p/RSA ➔
```

Sorting algorithms

OBJ: main.py

What worked?

I used python's built-in timing systems called "timeit", which let me easily and accurately time the different sorting algorithms. Quicksort, of course, outperformed pretty much any other algorithm. For the "roll-your-own" I tried to do a shell-like sort, but it was outperformed by a normal bubble sort anyway :(Understanding the quicksort algorithm wasn't very hard because of videos, but implementing it was hard because I don't have any kind of proper debugging tools yet, so I ended up having to understand a recursive function via printing to the console. I ended up getting it working anyway.

CODE(Python)

```
import random
import timeit

def init():
    # ten thousand integers.
    return [ random.randint( 0, 10000 ) for r in range( 1000 ) ]

def printList( List ):
    for i in List:
        print( str( i ) + ", " , end="" )
    print("")

def bubbleSort( List ):
    Sorted = False
    while Sorted == False:
        Sorted = True
        for i in range( 0, len( List ) - 1 ):
            #Detect if a swap is required
            if List[ i ] > List[ i + 1 ]:
                #Swap
                foo = List[ i + 1 ]
                List[ i + 1 ] = List[ i ]
                List[ i ] = foo
            #Make sure we know if we had to switch something
            Sorted = False

def split( List, pivotIndex, rightIndex ):
    leftCursor = pivotIndex + 1
    rightCursor = rightIndex
    finished = False
    swapPoint = -1
    while not finished:
        #for i in range( pivotIndex, rightIndex + 1 ):
            #print( i, ":", List[ i ], ", " , end="" )
        #print("")
        foundLeftSwap = False
        foundRightSwap = False
```

```

#Find a swap location with the left cursor
while not foundLeftSwap and leftCursor <= rightCursor:
    if List[ leftCursor ] > List[ pivotIndex ]:
        foundLeftSwap = True
        break
    if leftCursor > rightIndex:
        break
    leftCursor += 1

if not foundLeftSwap and leftCursor >= rightIndex:
    #Special case where we picked the highest number for the pivot point.
    #print( "Swapping Special", pivotIndex, "and", rightIndex )
    foo = List[ rightIndex ]
    List[ rightIndex ] = List[ pivotIndex ]
    List[ pivotIndex ] = foo
    finished = True
    swapPoint = rightIndex
    break

while not foundRightSwap and rightCursor > leftCursor:
    if List[ rightCursor ] <= List[ pivotIndex ]:
        foundRightSwap = True
        break
    if rightCursor < pivotIndex:
        break
    rightCursor -= 1

#If we found two valid swaps, swap and continue
if foundRightSwap and foundLeftSwap:
    #print( "Swapping Normal", leftCursor, "and", rightCursor )
    foo = List[ rightCursor ]
    List[ rightCursor ] = List[ leftCursor ]
    List[ leftCursor ] = foo
else:
    #Otherwise we have run out of things to do, and must put the pivot in the middle
    swapPoint = leftCursor - 1
    #print( "Sorted, now swapping", pivotIndex, "and", swapPoint )
    foo = List[ swapPoint ]
    List[ swapPoint ] = List[ pivotIndex ]
    List[ pivotIndex ] = foo
    finished = True
    break

#Now we should have a split list, to completely sort it we must recursively split it.
if swapPoint - pivotIndex > 1:
    #print( "left recursion at", swapPoint )
    split( List, pivotIndex, swapPoint - 1 )
if rightIndex - swapPoint > 1:
    #print( "right recursion at", swapPoint )
    split( List, swapPoint + 1, rightIndex )
return 0

```

```

def quickSort( List ):
    #Recursively split the list until it's sorted.
    split( List, 0, len( List ) - 1 )

def bogoSort( List ):
    Sorted = False
    while not Sorted:
        Sorted = True
        for i in range( 0, len( List ) - 1 ):
            if List[ i ] > List[ i + 1 ]:
                Sorted = False
                break
        if not Sorted:
            # :)
            random.shuffle( List )

def rollSort( List ):
    Sorted = False
    i = 0
    j = 100
    while not Sorted or j > 1:
        Sorted = True
        if i >= len( List ) - j:
            i -= len( List ) - j
            if j > 1:
                j -= 1
        while i < len( List ) - j:
            if List[ i ] > List[ i + j ]:
                foo = List[ i + j ]
                List[ i + j ] = List[ i ]
                List[ i ] = foo
                Sorted = False
            i += j

print( "Bubble Sort Time:", end=" " )
print( timeit.timeit( 'bubbleSort( init() )', setup="from __main__ import bubbleSort; from __main__ import init",
number=10 ), "seconds" )

print( "Quick Sort Time:", end=" " )
print( timeit.timeit( 'quickSort( init() )', setup="from __main__ import quickSort; from __main__ import init",
number=10 ), "seconds" )

print( "Roll-My-Own Sort Time:", end=" " )
print( timeit.timeit( 'rollSort( init() )', setup="from __main__ import rollSort; from __main__ import init",
number=10 ), "seconds" )

```

SCREENCAP

```
Terminal - fish /home/naelstrof/projects/sorting
~ > cd ~/projects/sorting
~/p/sorting > python3.3 main.py
Bubble Sort Time: 1.888590476999525 seconds
Quick Sort Time: 0.06817122899974493 seconds
Roll-My-Own Sort Time: 3.2109713790005117 seconds
~/p/sorting > |
```