Z3
Solver

What is Z3 ?

- Z3 is a powerful SMT (Satisfiability Modulo Theories) solver developed by Microsoft
- Open source
- It has official bindings for various common languages like C, C++, Python, Java, etc.
- It has command line interface that takes in a standardized format called SMTLIB.

What is Z3 ?

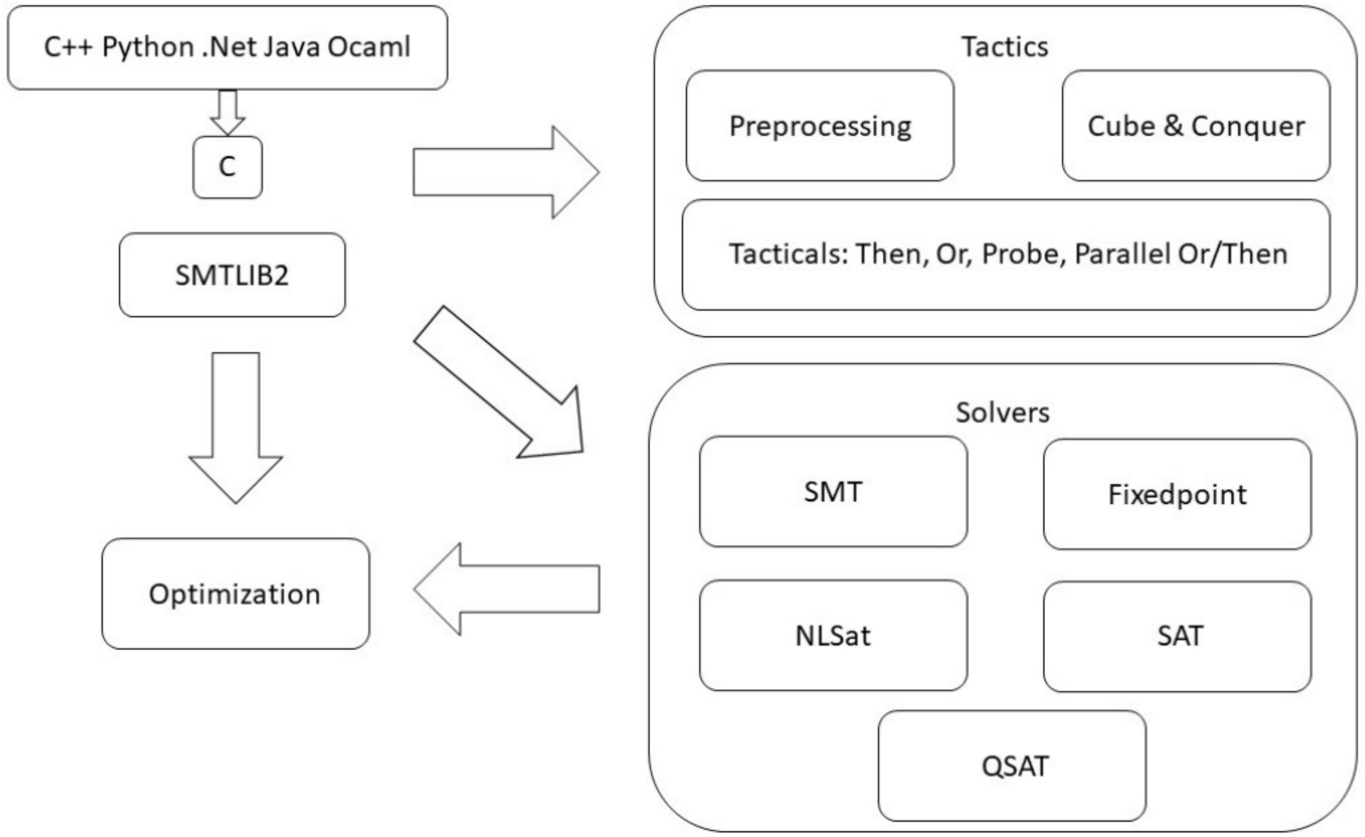


Figure 1. Overall system architecture of Z3

Things you can do with Z3

- Simplification capability
  - Simplifying complex logical or mathematical expressions
  - Reducing logical statements to a more concise form
  - Pre-processing constraints before solving, making the solver more efficient.

```
from z3 import *

x = Bool('x')
y = Bool('y')

expr = Or(x, Not(x), y)  # This should always be True
simplified = simplify(expr)
print(simplified)

# Output: True
```

- Constrain Satisfaction
  - Solving mathematical equations and inequalities
  - Program verification
  - Finding solutions to problems (e.g., Sudoku, scheduling)

```
x = Int('x')
y = Int('y')

solver = Solver()
solver.add(x + y == 10)
solver.add(x - y == 2)

if solver.check() == sat:
    print(solver.model())

# Output: x = 6, y = 4
```

Things you can do with Z3

- Manipulation & Expressing Statements
  - Z3 allows you to express logical, arithmetic, and set-based constraints in a formal way

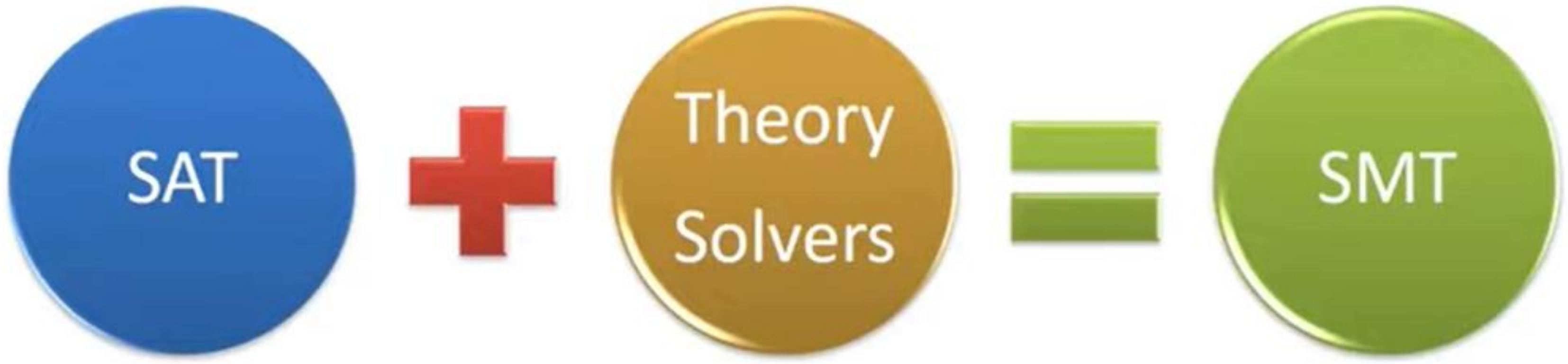
  - Boolean logic expression (AND, OR, NOT, IMPLIES, …)
  - Arithmetic constrains (=, <, >, <=, …)
  - Bit vector operations
  - Array and List
  - Quantifiers

    ⋮

Overall these capabilities enable Z3 to effectively and automatically find solution to a wide range of problems or prove that solution exits or not.

How Z3 Works

- A basis of SMT Solver



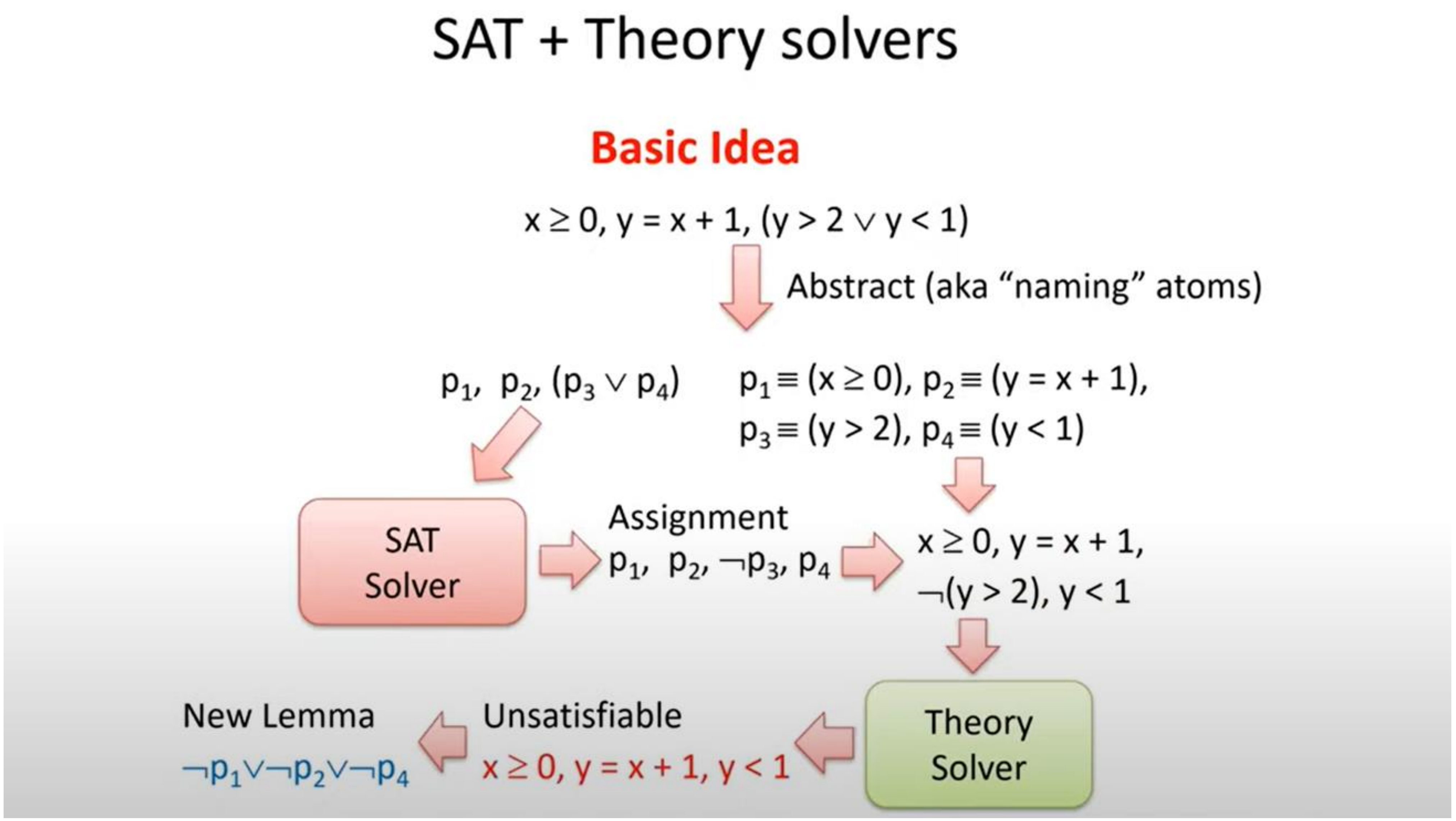
Propositional logics

$$(p \vee q) \wedge (q \vee \neg w) \wedge (w \vee r \vee p) \wedge (r \vee \neg w)$$

Complex domain specific facts

$$x + y \geq 7$$

How Z3 Works



SAT + Theory solvers
Basic Idea
$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$
Abstract (aka “naming” atoms)
$p_1, p_2, (p_3 \vee p_4)$
$p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$
SAT Solver
Assignment
$p_1, p_2, \neg p_3, p_4$
$x \geq 0, y = x + 1,$
$\neg(y > 2), y < 1$
Theory Solver
Unsatisfiable
$x \geq 0, y = x + 1, y < 1$
New Lemma
$\neg p_1 \vee \neg p_2 \vee \neg p_4$

How Z3 Works

- Z3 has built in procedures and understanding of :

  - propositional logic via SAT solving
  - Linear equations 3x+4y==7
  - Linear inequalities 3x+4y<=7
  - Bitvectors
  - Arrays
  - Algebraic Data Types
  - Uninterpeted Functions
  - polynomial equalities

How Z3 Works

➢ **What Z3 can not do:**

- You can express far more things to Z3 than it can solve
- Nonlinear equations in general are going to be tough. Not impossible, but tough.
- Z3 gives up immediately if you try to find the solution to a seemingly solvable problem involving an exponential.
- Z3 also can't really understand sines, cosines, and logarithms.
- The main exception is polynomial equality constraints, for which z3 has an intrinsic understanding, however these routines can be computationally expensive.

Input

```
x,y = Ints("x y")
pubkey = 3  * 7
solve(x * y == pubkey, x > 1, y > 1)
```

Output

```
[x = 3, y = 7]
```

Input

```
x,y = Ints("x y")
pubkey = 1000000993 * 1000001011
solve(x * y == pubkey, x > 1, y > 1)
```

Output

```
failed to solve
```

Installation

- Z3 has bindings for various programming languages: Python, C, C++, Java
- You can install the Python wrapper for Z3 using command:

pip install z3-solver

- For C++ bindings:
  - **Steps:**

```
- git clone https://github.com/Z3Prover/z3.git
- python scripts/mk_make.py
- cd build
- make
- sudo make install
- make examples
```

Z3
API
in
Python

Basics

A simple example

```
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

```
[y = 0, x = 7]
```

Z3 formula/expression simplifier

```
x = Int('x')
y = Int('y')
print (simplify(x + y + 2*x + 3))
print (simplify(x < y + x + 2))
print (simplify(And(x + 1 >= 3, x**2 + x**2
+ y**2 + 2 >= 5)))
```

```
3 + 3*x + y
Not(y <= -2)
And(x >= 2, 2*x**2 + y**2
>= 3)
```

Z3 provides functions for traversing expressions

```
x = Int('x')
y = Int('y')
n = x + y >= 3
print ("num args: ", n.num_args())
print ("children: ", n.children())
print ("1st child:", n.arg(0))
print ("2nd child:", n.arg(1))
print ("operator: ", n.decl())
print ("op name:  ", n.decl().name())
```

```
num args:
2 children: [x + y, 3]
1st child: x + y 2nd
child: 3
operator: >=
op name: >=
```

Z3 can solve nonlinear *polynomial* constraints

```
x = Real('x')
y = Real('y')
solve(x**2 + y**2 > 3, x**3 + y < 5)
```

```
[y = 2, x = 0]
```

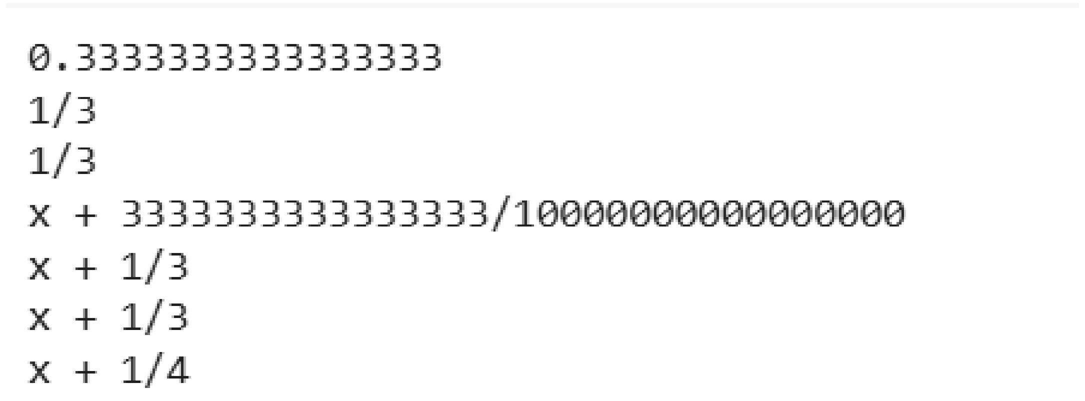
set_option is used to configure the Z3 environment

```
x = Real('x')
y = Real('y')
solve(x**2 + y**2 == 3, x**3 == 2)

set_option(precision=30)
print ("Solving, and displaying result with
30 decimal places")
solve(x**2 + y**2 == 3, x**3 == 2)
```

```
[y = -1.1885280594, x = 1.2599210498]

Solving, and displaying result with 30
decimal places

[
y = -1.188528059421316533710369365015,
x = 1.259921049894873164767210607278]
```

Different ways to create rational numbers

```
print (1/3)
print (RealVal(1)/3)
print (Q(1,3))

x = Real('x')
print (x + 1/3)
print (x + Q(1,3))
print (x + "1/3")
print (x + 0.25)
```



```
0.3333333333333333
1/3
1/3
x + 3333333333333333/10000000000000000
x + 1/3
x + 1/3
x + 1/4
```

A system of constraints may not have a solution.

```
x = Real('x')
solve(x > 4, x < 0)
```

```
no solution
```

Boolean Logic

Z3 supports Boolean operators: And, Or, Not, Implies (implication), If (if-then-else). Bi-implications are represented using equality ==

```
p = Bool('p')
q = Bool('q')
r = Bool('r')
solve(Implies(p, q), r == Not(q),
Or(Not(p), r))
```

```
[q = True, p = False, r = False]
```

The Python Boolean constants `True` and `False` can be used to build Z3 Boolean expressions.

```
p = Bool('p')
q = Bool('q')
print (And(p, q, True))
print (simplify(And(p, q, True)))
print (simplify(And(p, False)))
```

```
And(p, q, True)
And(p, q)
False
```

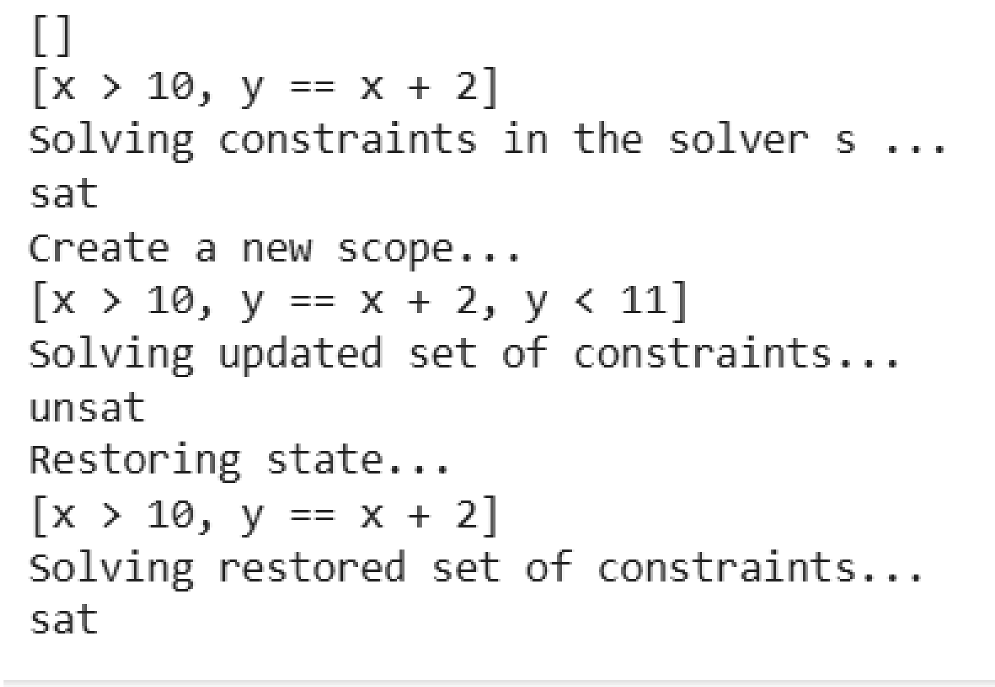
Combination of polynomial and Boolean constraints.

```
p = Bool('p')
x = Real('x')
solve(Or(x < 5, x > 10), Or(p, x**2 == 2),
Not(p))
```

```
[x = -1.414213562373095048801688724209,
p = False]
```

Solvers

```python
x = Int('x')
y = Int('y')

s = Solver()
print (s)

s.add(x > 10, y == x + 2)
print (s)
print ("Solving constraints in the solver s ...")
print (s.check())

print ("Create a new scope...")
s.push()
s.add(y < 11)
print (s)
print ("Solving updated set of constraints...")
print (s.check())

print ("Restoring state...")
s.pop()
print (s)
print ("Solving restored set of constraints...")
print (s.check())
```



```
[]
[x > 10, y == x + 2]
Solving constraints in the solver s ...
sat
Create a new scope...
[x > 10, y == x + 2, y < 11]
Solving updated set of constraints...
unsat
Restoring state...
[x > 10, y == x + 2]
Solving restored set of constraints...
sat
```

An example that Z3 cannot solve

```
x = Real('x')
s = Solver()
s.add(2**x == 3)
print (s.check())
```

```
unknown
```

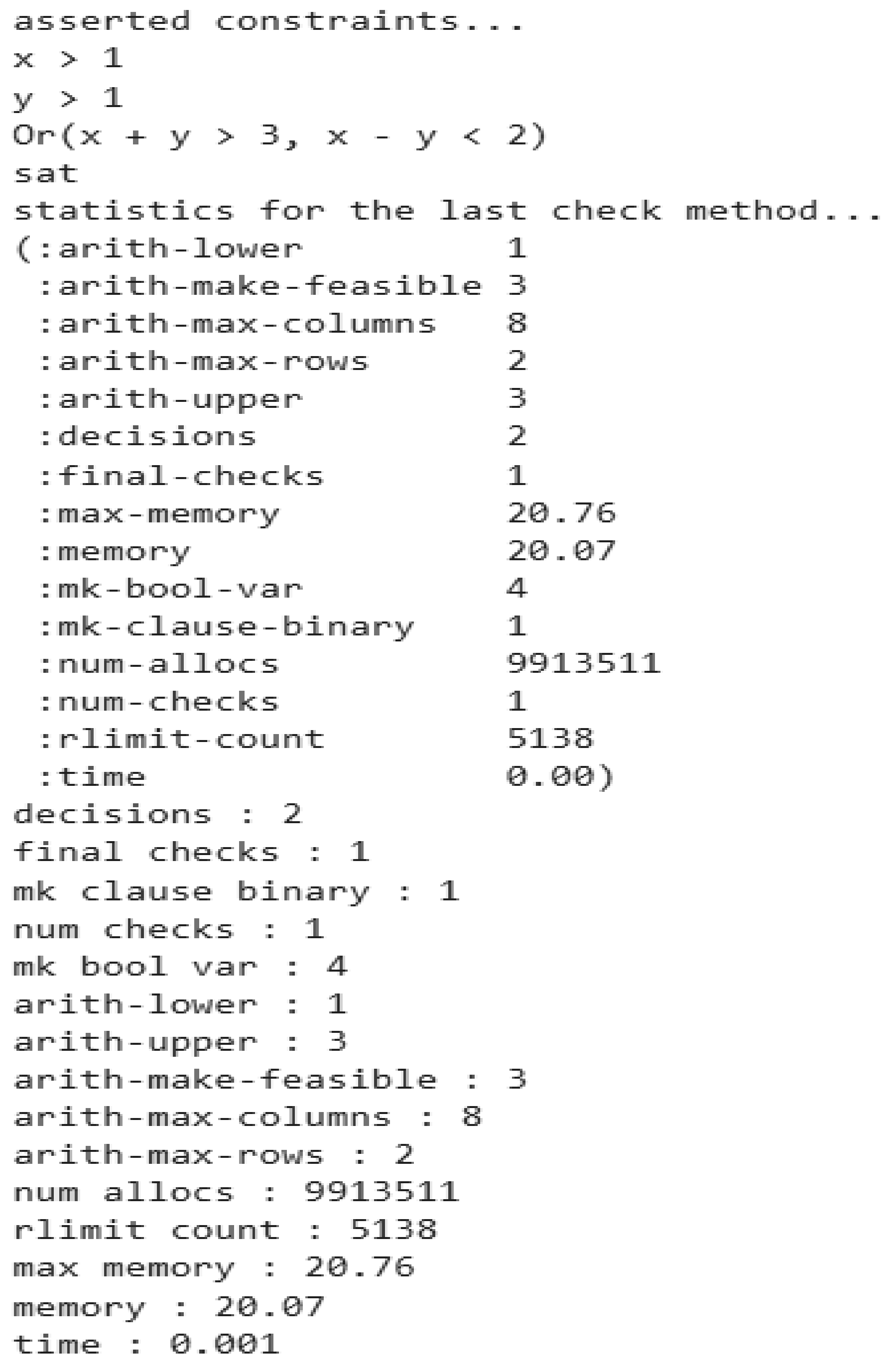
Basic methods for inspecting models

```
x, y, z = Reals('x y z')
s = Solver()
s.add(x > 1, y > 1, x + y > 3, z - x < 10)
print (s.check())

m = s.model()
print ("x = %s" % m[x])

print ("traversing model...")
for d in m.decls():
    print ("%s = %s" % (d.name(), m[d]))
```

```
sat
x = 1.5
traversing model...
y = 2
x = 1.5
z = 0
```

How to traverse the constraints asserted into a solver

```
x = Real('x')
y = Real('y')
s = Solver()
s.add(x > 1, y > 1, Or(x + y > 3, x - y <
2))
print ("asserted constraints...")
for c in s.assertions():
    print (c)

print (s.check())
print ("statistics for the last check
method...")
print (s.statistics())
# Traversing statistics
for k, v in s.statistics():
    print ("%s : %s" % (k, v))
```



```
asserted constraints...
x > 1
y > 1
Or(x + y > 3, x - y < 2)
sat
statistics for the last check method...
(:arith-lower          1
 :arith-make-feasible 3
 :arith-max-columns    8
 :arith-max-rows       2
 :arith-upper          3
 :decisions            2
 :final-checks         1
 :max-memory           20.76
 :memory               20.07
 :mk-bool-var          4
 :mk-clause-binary     1
 :num-allocs           9913511
 :num-checks           1
 :rlimit-count         5138
 :time                 0.00)
decisions : 2
final checks : 1
mk clause binary : 1
num checks : 1
mk bool var : 4
arith-lower : 1
arith-upper : 3
arith-make-feasible : 3
arith-max-columns : 8
arith-max-rows : 2
num allocs : 9913511
rlimit count : 5138
max memory : 20.76
memory : 20.07
time : 0.001
```

Arithmetic

Basic arithmetic operations

```
a, b, c = Ints('a b c')
d, e = Reals('d e')
solve(a > b + 2,
      a == 2*c + 10,
      c + b <= 1000,
      d >= e)
```

```
[d = 0, c = 0, b = 0, e = 0, a = 10]
```

Simple transformations on Z3 expressions.

```
x, y = Reals('x y')
# Put expression in sum-of-monomials form
t = simplify((x + y)**3, som=True)
print (t)
# Use power operator
t = simplify(t, mul_to_power=True)
print (t)
```

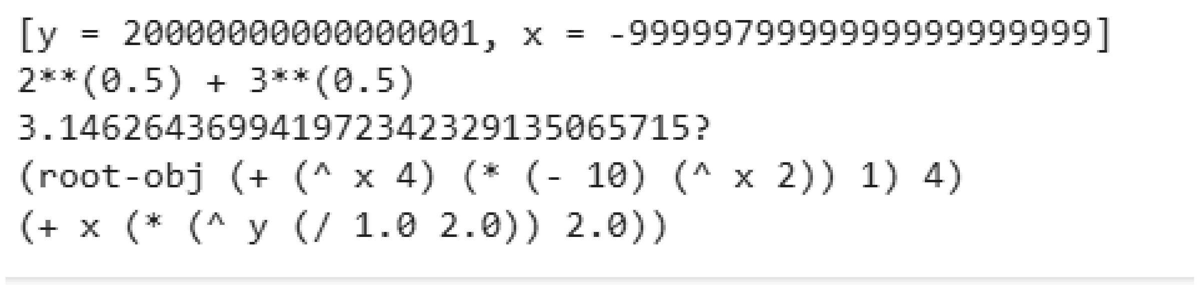
```
x*x*x + 3*x*x*y + 3*x*y*y + y*y*y
x**3 + 3*x**2*y + 3*x*y**2 + y**3
```

```
x, y = Reals('x y')
# Using Z3 native option names
print (simplify(x == y + 2, ':arith-lhs', True))
# Using Z3Py option names
print (simplify(x == y + 2, arith_lhs=True))

print ("\nAll available options:")
help_simplify()
```

```
x + -1*y == 2
x + -1*y == 2

All available options:
algebraic_number_evaluator (bool) simplify/evaluate expressions containing (algebraic) irrational numbers. (default: true)

arith_ineq_lhs (bool) rewrite inequalities so that right-hand-side is a constant. (default: false)
```

Z3Py supports arbitrarily large numbers.

```
x, y = Reals('x y')
solve(x + 10000000000000000000000 == y, y >
20000000000000000)

print (Sqrt(2) + Sqrt(3))
print (simplify(Sqrt(2) + Sqrt(3)))
print (simplify(Sqrt(2) + Sqrt(3)).sexpr())
# The sexpr() method is available for any
Z3 expression
print ((x + Sqrt(y) * 2).sexpr())
```

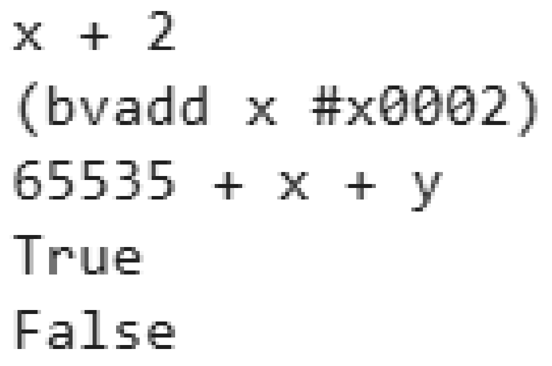


```
[y = 20000000000000001, x = -9999979999999999999999]
2**(0.5) + 3**(0.5)
3.1462643699419723423291350657155?
(root-obj (+ (^ x 4) (* (- 10) (^ x 2)) 1) 4)
(+ x (* (^ y (/ 1.0 2.0)) 2.0))
```

Machine Arithmetic

How to create bit-vector variables and constants

```
x = BitVec('x', 16)
y = BitVec('y', 16)
print (x + 2)
# Internal representation
print ((x + 2).sexpr())

# -1 is equal to 65535 for 16-bit integers
print (simplify(x + y - 1))

# Creating bit-vector constants
a = BitVecVal(-1, 16)
b = BitVecVal(65535, 16)
print (simplify(a == b))

a = BitVecVal(-1, 32)
b = BitVecVal(65535, 32)
# -1 is not equal to 65535 for 32-bit
integers
print (simplify(a == b))
```

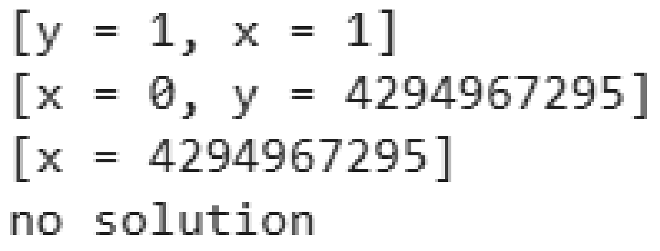


```
x + 2
(bvadd x #x0002)
65535 + x + y
True
False
```

- Z3 provides special signed versions of arithmetical operations where it makes a difference whether the bit-vector is treated as signed or unsigned.

- In Z3Py, the operators `<`, `<=`, `>`, `>=`, `/`, `%` and `>>` correspond to the signed versions. The corresponding unsigned operators are `ULT`, `ULE`, `UGT`, `UGE`, `UDiv`, `URem` and `LShR`.

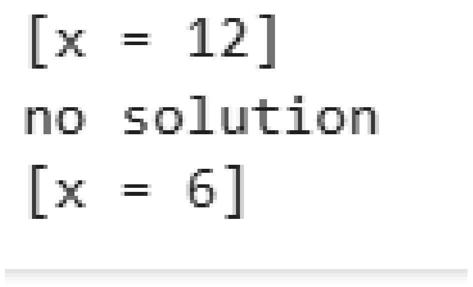
```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)

solve(x + y == 2, x > 0, y > 0)

# Bit-wise operators
# & bit-wise and
# | bit-wise or
# ~ bit-wise not
solve(x & y == ~y)

solve(x < 0)

# using unsigned version of <
solve(ULT(x, 0))
```



```
[y = 1, x = 1]
[x = 0, y = 4294967295]
[x = 4294967295]
no solution
```

The operator `>>` is the arithmetic shift right, and `<<` is the shift left. The logical shift right is the operator `LShR`.

```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)

solve(x >> 2 == 3)

solve(x << 2 == 3)

solve(x << 2 == 24)
```



```
[x = 12]
no solution
[x = 6]
```

Functions

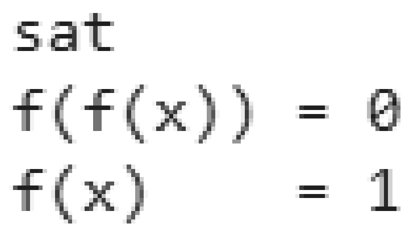
Z3 is based on first-order logic. Function and constant symbols in pure first-order logic are *uninterpreted* or *free*, which means that no a priori interpretation is attached.

```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
solve(f(f(x)) == x, f(x) == y, x != y)
```

```
[x = 0, y = 1, f = [1 -> 0, else -> 1]]
```

We can also evaluate expressions in the model for a system of constraints.

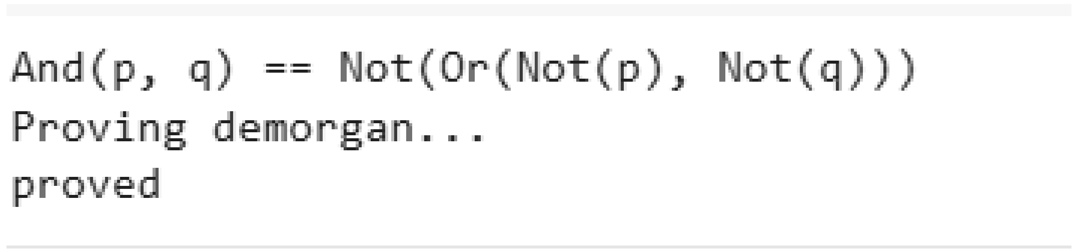
```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
s = Solver()
s.add(f(f(x)) == x, f(x) == y, x != y)
print (s.check())
m = s.model()
print ("f(f(x)) =", m.evaluate(f(f(x))))
print ("f(x)    =", m.evaluate(f(x)))
```



```
sat
f(f(x)) = 0
f(x)    = 1
```

Satisfiability and Validity

```
p, q = Bools('p q')
demorgan = And(p, q) == Not(Or(Not(p),
Not(q)))
print (demorgan)

def prove(f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        print ("proved")
    else:
        print ("failed to prove")

print ("Proving demorgan...")
prove(demorgan)
```



```
And(p, q) == Not(Or(Not(p), Not(q)))
Proving demorgan...
proved
```

List Comprehension

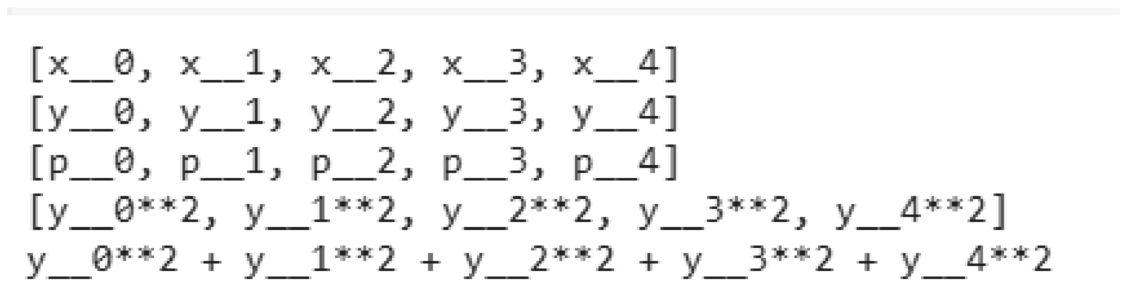
List comprehensions provide a concise way to create lists.

```
# Create list [1, ..., 5]
print ([ x + 1 for x in range(5) ])

# Create two lists containing 5 integer variables
X = [ Int('x%s' % i) for i in range(5) ]
Y = [ Int('y%s' % i) for i in range(5) ]
print (X)

# Create a list containing X[i]+Y[i]
X_plus_Y = [ X[i] + Y[i] for i in range(5) ]
print (X_plus_Y)

# Create a list containing X[i] > Y[i]
X_gt_Y = [ X[i] > Y[i] for i in range(5) ]
print (X_gt_Y)

print (And(X_gt_Y))

# Create a 3x3 "matrix" (list of lists) of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(3) ]
      for i in range(3) ]
pp(X)
```

```
[1, 2, 3, 4, 5]
[x0, x1, x2, x3, x4]
[x0 + y0, x1 + y1, x2 + y2, x3 + y3, x4 + y4]
[x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4]
And(x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4)
[[x_1_1, x_1_2, x_1_3],
 [x_2_1, x_2_2, x_2_3],
 [x_3_1, x_3_2, x_3_3]]
```

Z3Py also provides functions for creating vectors of Boolean, Integer and Real variables

```python
X = IntVector('x', 5)
Y = RealVector('y', 5)
P = BoolVector('p', 5)
print (X)
print (Y)
print (P)
print ([ y**2 for y in Y ])
print (Sum([ y**2 for y in Y ]))
```



```
[x__0, x__1, x__2, x__3, x__4]
[y__0, y__1, y__2, y__3, y__4]
[p__0, p__1, p__2, p__3, p__4]
[y__0**2, y__1**2, y__2**2, y__3**2, y__4**2]
y__0**2 + y__1**2 + y__2**2 + y__3**2 + y__4**2
```

Puzzles

Consider the following puzzle. Spend exactly 100 dollars and buy exactly 100 animals. Dogs cost 15 dollars, cats cost 1 dollar, and mice cost 25 cents each. You have to buy at least one of each. How many of each should you buy?

```
# Create 3 integer variables
dog, cat, mouse = Ints('dog cat mouse')
solve(dog >= 1,   # at least one dog
      cat >= 1,   # at least one cat
      mouse >= 1, # at least one mouse
      # we want to buy 100 animals
      dog + cat + mouse == 100,
      # We have 100 dollars (10000 cents):
      #   dogs cost 15 dollars (1500
cents),
      #   cats cost 1 dollar (100 cents),
and
      #   mice cost 25 cents
      1500 * dog + 100 * cat + 25 * mouse
== 10000)
```

```
[mouse = 56, cat = 41, dog = 3]
```

Sudoku

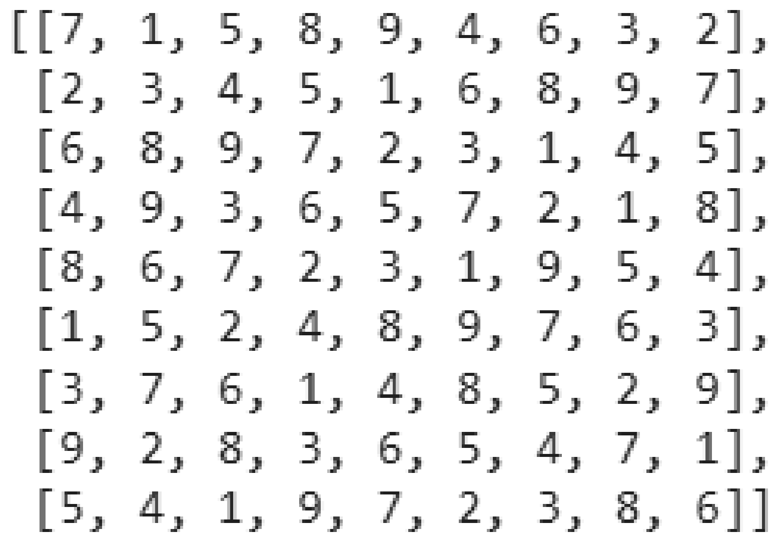
The goal is to insert the numbers in the boxes to satisfy only one condition: each row, column and 3x3 box must contain the digits 1 through 9 exactly once.

```
# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]

# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]

# each row contains a digit at most once
rows_c   = [ Distinct(X[i]) for i in range(9) ]

# each column contains a digit at most once
cols_c   = [ Distinct([ X[i][j] for i in range(9) ])
             for j in range(9) ]

# each 3x3 square contains a digit at most once
sq_c     = [ Distinct([ X[3*i0 + i][3*j0 + j]
                        for i in range(3) for j in range(3) ])
             for i0 in range(3) for j0 in range(3) ]
```

```
sudoku_c = cells_c + rows_c + cols_c + sq_c

# sudoku instance, we use '0' for empty cells
instance = ((0,0,0,0,9,4,0,3,0),
            (0,0,0,5,1,0,0,0,7),
            (0,8,9,0,0,0,0,4,0),
            (0,0,0,0,0,0,2,0,8),
            (0,6,0,2,0,1,0,5,0),
            (1,0,2,0,0,0,0,0,0),
            (0,7,0,0,0,0,5,2,0),
            (9,0,0,0,6,5,0,0,0),
            (0,4,0,9,7,0,0,0,0))
```

```
instance_c = [ If(instance[i][j] == 0,
                  True,
                  X[i][j] == instance[i][j])
               for i in range(9) for j in range(9) ]
```

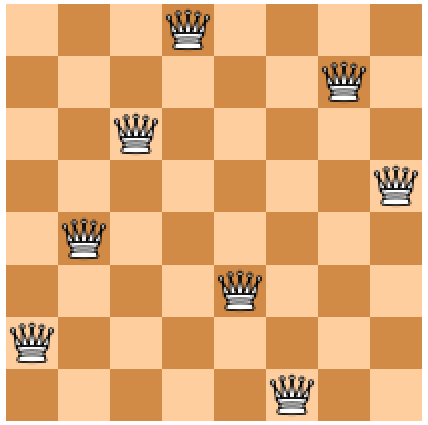
```
s = Solver()
s.add(sudoku_c + instance_c)
if s.check() == sat:
    m = s.model()
    r = [ [ m.evaluate(X[i][j]) for j in range(9) ]
          for i in range(9) ]
    print_matrix(r)
else:
    print ("failed to solve")
```



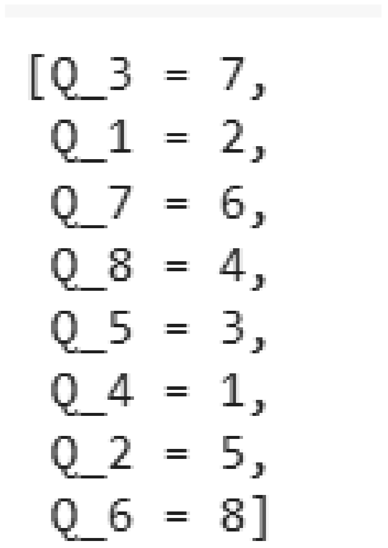
```
[[7, 1, 5, 8, 9, 4, 6, 3, 2],
 [2, 3, 4, 5, 1, 6, 8, 9, 7],
 [6, 8, 9, 7, 2, 3, 1, 4, 5],
 [4, 9, 3, 6, 5, 7, 2, 1, 8],
 [8, 6, 7, 2, 3, 1, 9, 5, 4],
 [1, 5, 2, 4, 8, 9, 7, 6, 3],
 [3, 7, 6, 1, 4, 8, 5, 2, 9],
 [9, 2, 8, 3, 6, 5, 4, 7, 1],
 [5, 4, 1, 9, 7, 2, 3, 8, 6]]
```

Eight Queens

The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.



```
# We know each queen must be in a different row.
# So, we represent each queen by a single integer: the column
position
Q = [ Int('Q_%i' % (i + 1)) for i in range(8) ]

# Each queen is in a column {1, ... 8 }
val_c = [ And(1 <= Q[i], Q[i] <= 8) for i in range(8) ]

# At most one queen per column
col_c = [ Distinct(Q) ]

# Diagonal constraint
diag_c = [ If(i == j,
              True,
              And(Q[i] - Q[j] != i - j, Q[i] - Q[j] != j - i))
           for i in range(8) for j in range(i) ]

solve(val_c + col_c + diag_c)
```



```
[Q_3 = 7,
 Q_1 = 2,
 Q_7 = 6,
 Q_8 = 4,
 Q_5 = 3,
 Q_4 = 1,
 Q_2 = 5,
 Q_6 = 8]
```

Knapsack Problem

```
#(Weight, Profit)[(7,1), (5,9), (6,10),
(7,9), (7,1)]
#sack capacity is 21
from z3 import *

s = Optimize()

i1, i2, i3, i4, i5 = Bools("i1 i2 i3 i4
i5")
total_weight = Int("total_weight")
total_profit = Int("total_profit")

s.add(total_weight == i1*7 + i2*5 + i3*6 +
i4*7 + i5*7)
s.add(total_profit == i1*1 + i2*9 + i3*10 +
i4*9 + i5*1)
s.add(total_weight<=21)

s.maximize(total_profit)

print(s.check())
print(s.model())
```

```
Sat

[i3 = True,
i5 = False,
i1 = False,
i4 = True,
i2 = True,
total_weight = 18,
total_profit = 28]
```

Question?

Assignment - 4

Consider a group of M students decided to create an exercise solution manual for the Artificial Intelligence: A Modern Approach book. Assume that the book contains total N number of problems {p_1, p_2, …, p_N} and all are independent of each other. To solve problem p_i a student has to spend t_i amount of time. All students in the group are capable of solving any problems from the book. If a student starts solving a problem then he will be busy till the problem is solved and no one else will solve that problem. What is the earliest time the group can finish solving all problems?

•Use Z3 to model this problem

•Write a random test case generator that takes ranges for - group size, number of problem and time required to solve for each food item, K and their prices - and generate 100 test cases. Run your program for each of them.

Input format:

% group size
G 10
% Total number of problems
N 100
% problem-id time-required
P 1 34
P 2 29

Submission:

•Submit ‘assg04.cpp’ or ‘assg04.py’ by 20th February.

•Submit test case generator as ‘genTestcase.cpp’ or ‘genTestcase.py’