Rational implementation using functions:

```python
def rational(n, d):
    if name == 'n':
        return n
    elif name == 'd':
        return d
    this function represents a rational number

return select
```

### Lists:

- `digits = [1, 8, 2, 8]
- `digits[3] = 8
- `pairs = [(10, 20), (30, 40)]
- `pairs[0] = 10
- `pairs[1][0] = 30

#### Executing a for statement:

```python
for x, y in pairs:
    ... same_count = same_count + 1
```

- `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`
- `range(-2, 2)`

#### Length: ending value - starting value

- `list(range(-2, 2))`
- `List constructor`
- `List(range(4))[0, 1, 2, 3]`

#### Membership:

- `digits = [1, 8, 2, 8]`
- `digits[0] = 1`
- `digits[1] = 8`
- `2 in digits = True`
- `2 not in digits = False`

Functions that aggregate iterable arguments:

- `sum(iterable, start=[]) = value`
- `max(iterable, key=None) = value`
- `min(iterable, key=None) = value`
- `all(iterable) = bool`
- `any(iterable) = bool`

#### Type dispatching:

**Exponential growth.** Recursive `fib` takes $\Theta(b^n)$ steps, where $b = \frac{1 + \sqrt{5}}{2} \approx 1.61828$.

**Linear growth.** Increasing $n$ only increases $R(n)$ by the factor $\Theta(n)$.

**Logarithmic growth.** Increasing $n$ exponentially only increases $R(n)$ by $\Theta(\log n)$.

**Constant.** The problem size doesn't matter.

### Lists:

- `[map exp for name in iter exp]` if `if filter exp`
- `[map exp for name in iter exp]` if `filter exp`

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent.
2. Create an empty result list that is the value of the expression.
3. For each element in the iterable value of `iter exp`:
   - Bind `name` to that element in the new frame step 1.
   - If `filter exp` evaluates to a true value, then add the value of `map exp` to the result list.

```python
>>> numbers = [1, 2, 3, 4]

list comprehensions:

```map exp for name in iter exp` if `if filter exp`

Short version: `[map exp for name in iter exp]`

The result of calling `repr` on a value as what Python prints in an interactive session.

**List & dictionary mutation:**

- `a = [10]`
- `b = a`
- `a = b`
- `a.append(20)`
- `b.append(20)`
- `a = [10]`
- `b = [10, 20]`
- `a = b`
- `b = [10, 20]`

- `nuns = [('I', '1'), ('V', '5'), ('X', '10')]
- `nuns[1] = 1`
- `nuns[('V', '1')] = 5`
- `nuns[(9, 4, 16, (5, 25))]`
- `nuns.values()`
- `nuns.get('A', 0)`
- `nuns.get('V', 5)`
- `(x, x * x) for x in range(3, 8)`
- `(x, x * x) for x in range(1, 5, 2)`

- `suits = ['club', 'diamond', 'spade', 'heart']`
- `suits.pop()`
- `suits.remove('heart')`
- `suits.extend(['sword', 'club'])`
- `suits.append('myriad')`
- `suits.insert(0, 'heart')`
- `suits.insert(0, 'diamond')`

**Identity:**

- `exp0 is exp1` evaluates to True if both `exp0` and `exp1` evaluate to the same object.
- `exp0 == exp1` evaluates to True if both `exp0` and `exp1` evaluate to equal values.
- Identical objects are always equal values.
- You can copy a list by calling the list constructor or slicing the list from the beginning to the end.

**Constants:**

Constant terms do not affect the order of growth of a process.

**Logarithms:**

The base of a logarithm does not affect the order of growth of a process.

**Nesting:**

When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps.

**Long-order terms:**

The fastest-growing part of the computation dominates the total.

**Status**:

- `x^2 = 2`
- `x^2` is not bound locally
- Create a new binding from name `x^2` to number 2 in the first frame of the current environment.
- `x^2 is bound locally`
- `x^2` is bound to object 2 in the first frame of the current environment.
- `x^2` is bound locally
- `x^2` is bound in the first non-local frame of the current environment in which `x^2` is bound.
- `x^2` is bound in a non-local frame
- SyntaxError: binding for `nonlocal x` found
- `x^2` is bound in a non-local frame
- SyntaxError: name `x^2` is parameter and nonlocal
- `x^2` also bound locally
Recursive description:
- A tree has a root value and a list of branches
- Each branch is a tree
- A tree with zero branches is called a leaf

Relative description:
- Each location is a node
- Each node has a value
- One node can be the parent/child of another

Recursive definition:

```python
def tree(root, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:] if is_tree(tree) else False

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    return not branches(tree)

def leaves(tree):
    """The leaf values in tree. (0, 1, 0, 1, 0, 1, 0, 1, 0, 1)"
    if tree.is_leaf():
        return [tree]
    else:
        return sum(leaves(b) for b in branches(tree)), []

# Example tree
BTree = tree(0, [5, 3, 9, None, None])
```

Python object system:

**Class**

```python
class Tree:
    def __init__(self, root, branches=[]):
        self.root = root
        self.branches = list(branches)

def __len__(self):
    return self.root + len(self.branches)

def __repr__(self):
    return Link(0, Link(1, Link(2, Link(3, Link(4)))))
```

**Function call**

```python
>>> tree(3, [tree(1), ... tree(2, [tree(1), ... tree(3)])])
[3, [1, [2, [1, [1]]]]]
```

**Method invocation**

```python
>>> Account.deposit(a, 5)
```

**Instance attributes**

```python
class Account:
    def __init__(self, account_holder, balance=0):  # constructor
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):  # method
        self.balance = self.balance + amount
        return self.balance
```

Assignment statements with a dot expression on their left-hand side affect the object of the dot expression.

- If the object is an instance, then assignment sets an instance attribute.
- If the object is a class, then assignment sets a class attribute.

```
# created an instance of Account
jim_account = Account('Jim')
```

```
# created a class of Account
Account = class
```

The `<expression>`, `<name>`

- `<expression>` can be any valid Python expression.
- `<name>` must be a simple name.

Evaluates to the value of the attribute looked up by `<name` in the object that is the value of the `<expression>`.

To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, `<name>` is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

Variations:

- If the object is an instance, then assignment sets an instance attribute.
- If the object is a class, then assignment sets a class attribute.

Python built-in sets:

```python
>>> s = {3, 2, 1, 4, 5}  # Some zero length sequence
>>> s.union((5,))  # Union
(1, 2, 3, 4, 5)
```

```
A binary search tree is a binary tree where each root is larger than all values in its left branch and smaller than all values in its right branch
```