

Programming Pearls

Sort 4M 7-digit numbers
You have 1M of RAM

Bits? Python?

```
def popcount(x):  
    x -= (x >> 1) & 0x55555555  
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333)  
    x = (x + (x >> 4)) & 0x0f0f0f0f  
    x += x >> 8  
    return x & 0x0f  
  
popcount(0) == 0  
popcount(3) == 2  
popcount(0xffffffff) == 32  
popcount(0xffefffff) == 31
```

Bits? Python?

```
def popcount(x):  
    return bin(x).count('1')  
popcount(0) == 0  
popcount(3) == 2  
popcount(0xffffffff) == 32  
popcount(0xffefffff) == 31
```

Rotate a list

Rotate a list

```
def rotate(items, cutoff):  
    items[:cutoff] = reversed(items[:cutoff])  
    items[cutoff:] = reversed(items[cutoff:])  
    items.reverse()  
    return items  
assert rotate(range(5), 2) == [2, 3, 4, 0, 1]
```

Table Oriented Programming

```
def is_tic_tac_toe_win(board):  
    for i in xrange(3):  
        if board[i][0] == board[i][1] == board[i][2] != None:  
            return True  
        if board[i][0] == board[i][1] == board[i][2] != None:  
            return True  
    if board[0][0] == board[1][1] == board[2][2] != None:  
        return True  
    if board[0][-1] == board[1][-2] == board[2][-3] != None:  
        return True  
    return False
```

Table Oriented Programming

```
WINNING_POSITIONS = [  
    ((0, 0), (0, 1), (0, 2)),  
    ((1, 0), (1, 1), (1, 2)),  
    ((2, 0), (2, 1), (2, 2)),  
    # ...  
]  
  
def is_tic_tac_toe_win(board):  
    return any(is_winning_position(board, p) for p in WINNING_POSITIONS)  
  
def is_winning_position(board, position):  
    x, y = position  
    if board[x][y] is not None:  
        return False  
    return len(set(board[px][py] for px, py in position)) == 1
```


Table Oriented Programming

```
def update(self, dt):
    pos = [self.rect.x, self.rect.y]
    self.vel.x += self.accel.x * dt
    self.vel.y += self.accel.y * dt
    pos[0] += self.vel.x * dt
    pos[1] += self.vel.y * dt

    diff_x = self.rect.x - pos[0]
    diff_y = self.rect.y - pos[1]
    for block in self.carry_block_list:
        block.rect.x -= diff_x
        block.rect.y -= diff_y

    self.rect.x = pos[0]
    self.rect.y = pos[1]
```

Table Oriented Programming

```
def update(self, dt):  
    pos = Vector2(self.pos)  
    self.vel += self.accel * dt  
  
    pos += self.vel * dt  
  
    translation = pos - self.pos  
  
    for block in self.carry_block_list:  
        block.pos += translation  
  
    self.pos = pos
```

Table Oriented Programming

```
def update(self, dt):  
    pos = Vector2(self.pos)  
    self.vel += self.accel * dt  
    pos += self.vel * dt  
  
    translation = pos - self.pos  
    for block in self.carry_block_list:  
        block.pos += translation  
  
    self.pos = pos
```

JSON-Oriented Programming

Find anagrams in a list of words

JSON-Oriented Programming

```
def word_to_key(word):
    return ''.join(sorted(word.lower()))
def anagram_list(words):
    return list(sorted(words, key=word_to_key))
def anagram_dictionary(words):
    return group_by(word_to_key, words)
def anagrams(word, anagram_dict):
    anagrams = list(anagram_dict[word_to_key(word)])
    anagrams.remove(word)
    return anagrams
d = anagram_dictionary(['hello', 'moon', 'mono'])
assert anagrams('hello', d) == []
assert anagrams('moon', d) == ['mono']
assert anagrams('mono', d) == ['moon']
```

Missing Function: histogram()

```
def histogram(items):  
    result = {}  
    for item in items:  
        if item not in result:  
            result[item] = 0  
        result[item] += 1  
    return result  
assert histogram(['a', 'b', 'a', 'a', 'c']) == {'a': 3, 'b': 1, 'c': 1}
```

Missing Function: group_by()

```
def group_by(key, items):  
    result = {}  
    for item in items:  
        k = key(item)  
        if k not in result:  
            result[k] = []  
        result[k].append(item)  
    return result  
assert group_by(lambda x: x % 3, range(5)) == {0: [0, 3], 1: [1, 4], 2: [2]}
```

Missing function: `top_n()`

Actually, it's `heapq.nlargest()`
(why `heapq`?)

Missing function: unzip()

Actually,
unzip = lambda lists: zip(*lists)

Missing function: `binary_search`

:(