

SCREEN SHOTS : 1. BRUTE FORCE

```

assignment_2.py x
19 # The brute force method to solve max subarray problem
20
21
22 def find_maximum_subarray_brute(A):
23     """
24
25     Return a tuple (i,j) where A[i:j] is the maximum subarray.
26
27     time complexity = O(n^2)
28
29     """
30
31     if len(A) == 0:
32         return None
33
34     start = 0
35     end = len(A)
36     maxsum = -999999 # Setting the value to infinity
37
38     for i in range(start, end, 1):
39         total = 0
40
41         for j in range(i, end, 1):
42             # add every element of the array to the variable sum
43             total = total + A[j]
44
45             if total > maxsum: # check if sum is greater than max-sum
46                 maxsum = total
47                 start_index = i # Assign the start index of the sub-array
48                 end_index = j # Assign the last index of the sub-array
49
50     return (start_index, end_index, maxsum)
51
52

```

TEST CASES:

1. Positive and Negative numbers in an array

```
STOCK_PRICE_CHANGES = [15, 20, 45, -80, 81]
```

```

assignment_2.py .
Brute Force Method
(0, 4)

```

2. Only Positive Numbers

```

STOCK_PRICE_CHANGES = [100, 113, 110, 85, 105, 102, 86, 63, 81,
                        101, 94, 106, 101, 79, 94, 90, 97]
===== test session starts =====
platform win32 -- Python 2.7.13, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: C:\Users\gu\Desktop, inifile:
collected 1 items
assignment_2.py .
Brute Force Method
(0, 16)

```

3. Only Negative numbers

```
STOCK_PRICE_CHANGES = [-13, -3, -25, -20, -3, -16, -23, -18,  
-20, -7, -12, -5, -22, -15, -4, -7]  
  
===== test session starts =====  
platform win32 -- Python 2.7.13, pytest-3.0.5, py-1.4.32, pluggy-0.4.0  
rootdir: C:\Users\gu\Desktop, inifile:  
collected 1 items  
assignment_2.py .  
Brute Force Method  
(1, 1)
```

2. MAXIMUM SUB-ARRAY (RECURSIVE)

```

assignment_2.py x
55 # The maximum crossing subarray method for solving the max subarray problem
56 def find_maximum_crossing_subarray(A, low, mid, high):
57     """
58     Find the maximum subarray that crosses mid
59     Return a tuple ((i, j), sum) where sum is the maximum subarray of A[i:j].
60     """
61     # By following the algorithm in text_book
62     # To set the values to infinity
63     # can also use decimal(-Infinity) after importing decimal
64
65     temp_sum1 = 0
66     i = mid
67     left_ptr = i
68
69     left_sum = -999999
70
71     # To find the maximum sum at the left sub-array
72     # Left array is from low to mid
73
74     while i >= low:
75         temp_sum1 = temp_sum1 + A[i]
76         if temp_sum1 > left_sum:
77             left_sum = temp_sum1
78             left_ptr = i
79         i = i - 1
80
81     # To find the maximum sum at the right sub-array
82     # Right array is from mid+1 to high
83
84     temp_sum2 = 0
85     j = mid + 1
86
87     right_sum = -999999
88
89     right_ptr = j
90     while j <= high:
91         temp_sum2 = temp_sum2 + A[j]
92         if temp_sum2 > right_sum:
93             right_sum = temp_sum2
94             right_ptr = j
95         j = j + 1
96
97     return left_sum + right_sum, left_ptr, right_ptr
98
99
100 # The recursive method to solve max subarray problem
101
102 def find_maximum_subarray_recursive_helper(A, low=0, high=-1):
103     """
104     Return a tuple ((i, j), sum) where sum is the maximum subarray of A[i:j].
105     """
106
107     # Base case -> when there is only 1 element
108     # Algorithm in textbook is implemented
109
110     if low == high:
111         return A[low], low, high
112     else:
113         mid = (low + high) // 2
114         left_sum, left_low, left_high = \
115             find_maximum_subarray_recursive_helper(A, low, mid)
116         right_sum, right_low, right_high = \
117             find_maximum_subarray_recursive_helper(A, mid + 1, high)
118         cross_sum, cross_low, cross_high = \
119             find_maximum_crossing_subarray(A, low, mid, high)
120         if left_sum >= right_sum and left_sum >= cross_sum:
121             return left_sum, left_low, left_high
122         elif right_sum >= left_sum and right_sum >= cross_sum:
123             return right_sum, right_low, right_high
124         else:
125             return cross_sum, cross_low, cross_high
126
127 # The recursive method to solve max subarray problem
128
129
130

```

```

130
131
132 def find_maximum_subarray_recursive(A):
133     """
134     Return a tuple (i,j) where A[i:j] is the maximum subarray.
135     """
136     # To check if the array is empty
137
138     if len(A) == 0:
139         return None
140
141     return find_maximum_subarray_recursive_helper(A, 0, len(A) - 1)
142
143
144 # =====
145

```

TEST CASES:

1. For mixture of positive and negative numbers

```

STOCK_PRICE_CHANGES = [13, -3, -25, 20, -3, -16, -23, 18,
                        20, -7, 12, -5, -22, 15, -4, 7]

```

```

Recursive method
((7, 10), 43)

```

2. For only positive numbers

```

STOCK_PRICE_CHANGES = [100, 113, 110, 85, 105, 102, 86, 63, 81,
                        101, 94, 106, 101, 79, 94, 90, 97]

```

```

Recursive method
((0, 16), 1607)

```

3. For only negative numbers

```

STOCK_PRICE_CHANGES = [-13, -3, -25, -20, -3, -16, -23, -18,
                        -20, -7, -12, -5, -22, -15, -4, -7]

```

```

Recursive method
((1, 1), -3)

```

3. ITERATIVE METHOD

```

145 # =====
146 # The iterative method to solve max subarray problem
147 def find_maximum_subarray_iterative(A, low=0, high=-1):
148     """
149     Return a tuple (i,j) where A[i:j] is the maximum subarray.
150     """
151     if len(A) == 0:
152         return None
153
154     low = 0
155     high = len(A)
156     totalSum = A[low]
157     tempSum = 0
158     tempLeftIndex = 0
159     leftIndex = 0
160     rightIndex = 0
161
162     for i in range(low, high, 1):
163         tempSum = max(A[i], (tempSum + A[i]))
164         if tempSum == A[i]:
165             tempLeftIndex = i
166         if tempSum > totalSum:
167             totalSum = tempSum
168             rightIndex = i
169             leftIndex = tempLeftIndex
170
171     return (leftIndex, rightIndex, totalSum)
172 # =====
173
174
175
176
177

```

TEST CASES:

1. For all negative numbers

```

STOCK_PRICE_CHANGES = [-13, -3, -25, -20, -3, -16, -23, -18,
                        -20, -7, -12, -5, -22, -15, -4, -7]

```

```

Iterative method
(1, 1)

```

2. For all positive numbers

```

STOCK_PRICE_CHANGES = [100, 113, 110, 85, 105, 102, 86, 63, 81,
                        101, 94, 106, 101, 79, 94, 90, 97]

```

```

Iterative method
(0, 16)

```

3. For both positive and negative numbers

```
STOCK_PRICE_CHANGES = [13, -3, -25, 20, -3, -16, -23, 18,
                        20, -7, 12, -5, -22, 15, -4, 7]
```

```
Iterative method
(7, 10)
```

4. Matrix multiplication

```
assignment_2.py x
175 # =====
176
177
178 def square_matrix_multiply(A, B):
179     """
180     Return the product AB of matrix multiplication.
181     """
182
183     A = asarray(A)
184     B = asarray(B)
185
186     assert A.shape == B.shape
187     assert A.shape == A.T.shape
188
189     # Since it is a square matrix, row = column = n
190
191     n = len(A)
192
193     C = [[0 for i in range(n)] for j in range(n)]
194
195     for i in range(0, n):
196         for j in range(0, n):
197             C[i][j] = 0
198             for k in range(0, n):
199                 C[i][j] = C[i][j] + A[i][k] * B[k][j]
200
201     return C
202
203
```

Output

Calling function

```
A = asarray([[1, 3], [7, 5]])
B = asarray([[6, 8], [4, 2]])
print
print("Normal square matrix multiplication")
print(square_matrix_multiply(A, B))
print
```

```
Normal square matrix multiplication
[[18, 14], [62, 66]]
```

5. Strassen's matrix multiplication

```
assignment_2.py x
204 =====
205 |
206
207 def sum_up(A, B):
208     n = len(A)
209     result = [[0 for i in range(0, n)] for j in range(0, n)]
210     for i in range(0, n):
211         for j in range(0, n):
212             result[i][j] = A[i][j] + B[i][j]
213     return result
214
215 # subtracts two matrices
216
217
218 def difference(A, B):
219     n = len(A)
220     result = [[0 for i in range(0, n)] for j in range(0, n)]
221     for i in range(0, n):
222         for j in range(0, n):
223             result[i][j] = A[i][j] - B[i][j]
224     return result
225
226
227 def square_matrix_multiply_strassens(A, B):
228     A = asarray(A)
229     B = asarray(B)
230
231     assert A.shape == B.shape
232     assert A.shape == A.T.shape
233
234     assert (len(A) & (len(A) - 1)) == 0, "A is not a power of 2"
235
236     n = len(A)
237
238     if n == 1:
239         C = [[0 for j in range(0, n)] for i in range(0, n)]
240         for i in range(0, n):
241             for j in range(0, n):
242                 C[i][j] = A[i][j] * B[i][j]
243     return C
```

```

244     else: # dividing the input matrices A and B
245         new_n = int(n / 2)
246
247         a11 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
248         a12 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
249         a21 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
250         a22 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
251
252         b11 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
253         b12 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
254         b21 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
255         b22 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
256
257         aTemp = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
258         bTemp = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
259
260         for i in range(0, new_n):
261             for j in range(0, new_n):
262                 a11[i][j] = A[i][j]
263                 a12[i][j] = A[i][j + new_n]
264                 a21[i][j] = A[i + new_n][j]
265                 a22[i][j] = A[i + new_n][j + new_n]
266
267                 b11[i][j] = B[i][j]
268                 b12[i][j] = B[i][j + new_n]
269                 b21[i][j] = B[i + new_n][j]
270                 b22[i][j] = B[i + new_n][j + new_n]
271
272         aTemp = sum_up(a11, a22)
273         bTemp = sum_up(b11, b22)
274         p1 = square_matrix_multiply_strassens(aTemp, bTemp)
275
276         aTemp = sum_up(a21, a22)
277         p2 = square_matrix_multiply_strassens(aTemp, b11)
278
279         bTemp = difference(b12, b22)
280         p3 = square_matrix_multiply_strassens(a11, bTemp)
281

```

```

281
282         bTemp = difference(b21, b11)
283         p4 = square_matrix_multiply_strassens(a22, bTemp)
284
285         aTemp = sum_up(a11, a12)
286         p5 = square_matrix_multiply_strassens(aTemp, b22)
287
288         aTemp = difference(a21, a11)
289         bTemp = sum_up(b11, b12)
290         p6 = square_matrix_multiply_strassens(aTemp, bTemp)
291
292         aTemp = difference(a12, a22)
293         bTemp = sum_up(b21, b22)
294         p7 = square_matrix_multiply_strassens(aTemp, bTemp)
295
296         aTemp = sum_up(p1, p4)
297         bTemp = sum_up(aTemp, p7)
298         c11 = difference(bTemp, p5)
299         c12 = sum_up(p3, p5)
300         c21 = sum_up(p2, p4)
301
302         aTemp = sum_up(p1, p3)
303         bTemp = sum_up(aTemp, p6)
304         c22 = difference(bTemp, p2)
305
306         C = [[0 for i in range(0, n)] for j in range(0, n)]
307         for i in range(0, new_n):
308             for j in range(0, new_n):
309                 C[i][j] = c11[i][j]
310                 C[i][j + new_n] = c12[i][j]
311                 C[i + new_n][j] = c21[i][j]
312                 C[i + new_n][j + new_n] = c22[i][j]
313         return C
314
315     pass

```


Output


```
print("Strassen's matrix multiplication")
print (square_matrix_multiply_strassens(A, B))

pass
```

```
A = asarray([[1, 3], [7, 5]])
B = asarray([[6, 8], [4, 2]])
```

Result:


```
Strassen's matrix multiplication
[[18, 14], [62, 66]]
```

FLAKE8: No warnings and complexity<10 Command Prompt

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\gu>flake8 C:\Users\gu\Desktop\assignment_2.py

C:\Users\gu>
```

 Command Prompt

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\gu>flake8 C:\Users\gu\Desktop\assignment_2.py

C:\Users\gu>flake8 --max-complexity 10 C:\Users\gu\Desktop\assignment_2.py

C:\Users\gu>
```

PYTHON CODE

```
# -*- coding: utf-8 -*-
```

```
from numpy import asarray
```

```
# TODO: Replace all TODO comments (yes, this one too!)
```

```
"""
```

```
STOCK_PRICE_CHANGES = [100, 113, 110, 85, 105, 102, 86, 63, 81,  
101, 94, 106, 101, 79, 94, 90, 97]
```

```
"""
```

```
STOCK_PRICE_CHANGES = [13, -3, -25, 20, -3, -16, -23, 18,  
20, -7, 12, -5, -22, 15, -4, 7]
```

```
# =====
```

```
# The brute force method to solve max subarray problem
```

```
def find_maximum_subarray_brute(A):
```

```
    """
```

Return a tuple (i,j) where A[i:j] is the maximum subarray.

time complexity = $O(n^2)$

```
"""
```

```
if len(A) == 0:
```

```
    return None
```

```
start = 0
```

```
end = len(A)
```

```
maxsum = -999999 # Setting the value to infinity
```

```
for i in range(start, end, 1):
```

```
    total = 0
```

```
    for j in range(i, end, 1):
```

```
        # add every element of the array to the variable sum
```

```
        total = total + A[j]
```

```
    if total > maxsum:    # check if sum is greater than max-sum
```

```
        maxsum = total
```

```
    start_index = i    # Assign the start index of the sub-array
```

```
    end_index = j    # Assign the last index of the sub-array
```

```
return (start_index, end_index, maxsum)
```

```
# =====
```

```
# The maximum crossing subarray method for solving the max subarray problem
```

```
def find_maximum_crossing_subarray(A, low, mid, high):
```

```
    """
```

```
    Find the maximum subarray that crosses mid
```

```
    Return a tuple ((i, j), sum) where sum is the maximum subarray of A[i:j].
```

```
    """
```

```
    # By following the algorithm in text_book
```

```
    # To set the values to infinity
```

```
    # can also use decimal(-Infinity) after importing decimal
```

```
    temp_sum1 = 0
```

```
    i = mid
```

```
    left_ptr = i
```

```
    left_sum = -999999
```

```
    # To find the maximum sum at the left sub-array
```

```
    # Left array is from low to mid
```

```
while i >= low:
```

```
    temp_sum1 = temp_sum1 + A[i]
```

```
    if temp_sum1 > left_sum:
```

```
        left_sum = temp_sum1
```

```
        left_ptr = i
```

```
    i = i - 1
```

```
# To find the maximum sum at the left sub-array
```

```
# Right array is from mid+1 to high
```

```
temp_sum2 = 0
```

```
j = mid + 1
```

```
right_sum = -999999
```

```
right_ptr = j
```

```
while j <= high:
```

```
    temp_sum2 = temp_sum2 + A[j]
```

```
    if temp_sum2 > right_sum:
```

```
        right_sum = temp_sum2
```

```
        right_ptr = j
```

```
    j = j + 1
```

```
return left_sum + right_sum, left_ptr, right_ptr
```

```
# The recursive method to solve max subarray problem
```

```
def find_maximum_subarray_recursive_helper(A, low=0, high=-1):
```

```
    """
```

```
    Return a tuple ((i, j), sum) where sum is the maximum subarray of A[i:j].
```

```
    """
```

```
# Base case -> when there is only 1 element
```

```
# Algorithm in textbook is implemented
```

```
if low == high:
```

```
    return A[low], low, high
```

```
else:
```

```
    mid = (low + high) // 2
```

```
    left_sum, left_low, left_high = \
```

```
        find_maximum_subarray_recursive_helper(A, low, mid)
```

```
    right_sum, right_low, right_high = \
```

```
        find_maximum_subarray_recursive_helper(A, mid + 1, high)
```

```
    cross_sum, cross_low, cross_high = \
```

```
    find_maximum_crossing_subarray(A, low, mid, high)
    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_sum, left_low, left_high
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_sum, right_low, right_high
    else:
        return cross_sum, cross_low, cross_high

# The recursive method to solve max subarray problem

def find_maximum_subarray_recursive(A):
    """
    Return a tuple (i,j) where A[i:j] is the maximum subarray.
    """
    # To check if the array is empty

    if len(A) == 0:
        return None

    return find_maximum_subarray_recursive_helper(A, 0, len(A) - 1)

# =====
```

The iterative method to solve max subarray problem

```
def find_maximum_subarray_iterative(A, low=0, high=-1):
```

```
    """
```

Return a tuple (i,j) where A[i:j] is the maximum subarray.

```
    """
```

```
    if len(A) == 0:
```

```
        return None
```

```
    low = 0
```

```
    high = len(A)
```

```
    totalSum = A[low]
```

```
    tempSum = 0
```

```
    tempLeftIndex = 0
```

```
    leftIndex = 0
```

```
    rightIndex = 0
```

```
    for i in range(low, high, 1):
```

```
        tempSum = max(A[i], (tempSum + A[i]))
```

```
        if tempSum == A[i]:
```

```
            tempLeftIndex = i
```

```
        if tempSum > totalSum:
```

```
            totalSum = tempSum
```



```
rightIndex = i
```

```
leftIndex = tempLeftIndex
```

```
return (leftIndex, rightIndex, totalSum)
```

```
# =====
```

```
def square_matrix_multiply(A, B):
```

```
    """
```

```
    Return the product AB of matrix multiplication.
```

```
    """
```

```
    A = asarray(A)
```

```
    B = asarray(B)
```

```
    assert A.shape == B.shape
```

```
    assert A.shape == A.T.shape
```

```
    # Since it is a square matrix , row = column = n
```

```
    n = len(A)
```

```
C = [[0 for i in range(n)] for j in range(n)]
```

```
for i in range(0, n):
```

```
    for j in range(0, n):
```

```
        C[i][j] = 0
```

```
        for k in range(0, n):
```

```
            C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

```
return C
```

```
# =====
```

```
# Addition
```

```
def sum_up(A, B):
```

```
    n = len(A)
```

```
    result = [[0 for i in range(0, n)] for j in range(0, n)]
```

```
    for i in range(0, n):
```

```
        for j in range(0, n):
```

```
            result[i][j] = A[i][j] + B[i][j]
```

```
    return result
```

```
# subtracts two matrices
```

```
def difference(A, B):
```

```
n = len(A)
result = [[0 for i in range(0, n)] for j in range(0, n)]
for i in range(0, n):
    for j in range(0, n):
        result[i][j] = A[i][j] - B[i][j]
return result
```

```
def square_matrix_multiply_strassens(A, B):
    A = asarray(A)
    B = asarray(B)

    assert A.shape == B.shape
    assert A.shape == A.T.shape

    assert (len(A) & (len(A) - 1)) == 0, "A is not a power of 2"

    n = len(A)

    if n == 1:
        C = [[0 for j in range(0, n)] for i in range(0, n)]
        for i in range(0, n):
            for j in range(0, n):
                C[i][j] = A[i][j] * B[i][j]
```

```
return C
```

```
else: # dividing the input matrices A and B
```

```
    new_n = int(n / 2)
```

```
    a11 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    a12 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    a21 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    a22 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    b11 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    b12 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    b21 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    b22 = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    aTemp = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    bTemp = [[0 for i in range(0, new_n)] for j in range(0, new_n)]
```

```
    for i in range(0, new_n):
```

```
        for j in range(0, new_n):
```

```
            a11[i][j] = A[i][j]
```

```
            a12[i][j] = A[i][j + new_n]
```

```
            a21[i][j] = A[i + new_n][j]
```

```
            a22[i][j] = A[i + new_n][j + new_n]
```

b11[i][j] = B[i][j]

b12[i][j] = B[i][j + new_n]

b21[i][j] = B[i + new_n][j]

b22[i][j] = B[i + new_n][j + new_n]

aTemp = sum_up(a11, a22)

bTemp = sum_up(b11, b22)

p1 = square_matrix_multiply_strassens(aTemp, bTemp)

aTemp = sum_up(a21, a22)

p2 = square_matrix_multiply_strassens(aTemp, b11)

bTemp = difference(b12, b22)

p3 = square_matrix_multiply_strassens(a11, bTemp)

bTemp = difference(b21, b11)

p4 = square_matrix_multiply_strassens(a22, bTemp)

aTemp = sum_up(a11, a12)

p5 = square_matrix_multiply_strassens(aTemp, b22)

aTemp = difference(a21, a11)

bTemp = sum_up(b11, b12)

p6 = square_matrix_multiply_strassens(aTemp, bTemp)

```
aTemp = difference(a12, a22)
bTemp = sum_up(b21, b22)
p7 = square_matrix_multiply_strassens(aTemp, bTemp)
```

```
aTemp = sum_up(p1, p4)
bTemp = sum_up(aTemp, p7)
c11 = difference(bTemp, p5)
c12 = sum_up(p3, p5)
c21 = sum_up(p2, p4)
```

```
aTemp = sum_up(p1, p3)
bTemp = sum_up(aTemp, p6)
c22 = difference(bTemp, p2)
```

```
C = [[0 for i in range(0, n)] for j in range(0, n)]
```

```
for i in range(0, new_n):
```

```
    for j in range(0, new_n):
```

```
        C[i][j] = c11[i][j]
```

```
        C[i][j + new_n] = c12[i][j]
```

```
        C[i + new_n][j] = c21[i][j]
```

```
        C[i + new_n][j + new_n] = c22[i][j]
```

```
return C
```

pass

=====

def test():

TODO: Test all of the methods and print results.

calling the brute force method of max_subarray problem

index1, index2, maxSum =
(find_maximum_subarray_brute(STOCK_PRICE_CHANGES))

print

print("Brute Force Method")

print (index1, index2)

finalSum, l_index, r_index = \

find_maximum_subarray_recursive(STOCK_PRICE_CHANGES)

print

print("Recursive method")

print ((l_index, r_index), (finalSum))

print

l, r, sum = find_maximum_subarray_iterative(STOCK_PRICE_CHANGES)

print("Iterative method")

```
print(l, r)
```

```
print
```

```
# Taking Matrix A and B from textbook
```

```
A = asarray([[1, 3], [7, 5]])
```

```
B = asarray([[6, 8], [4, 2]])
```

```
print
```

```
print("Normal square matrix multiplication")
```

```
print(square_matrix_multiply(A, B))
```

```
print
```

```
print("Strassen's matrix multiplication")
```

```
print (square_matrix_multiply_strassens(A, B))
```

```
pass
```

```
if __name__ == '__main__':
```

```
test()
```

```
# =====
```