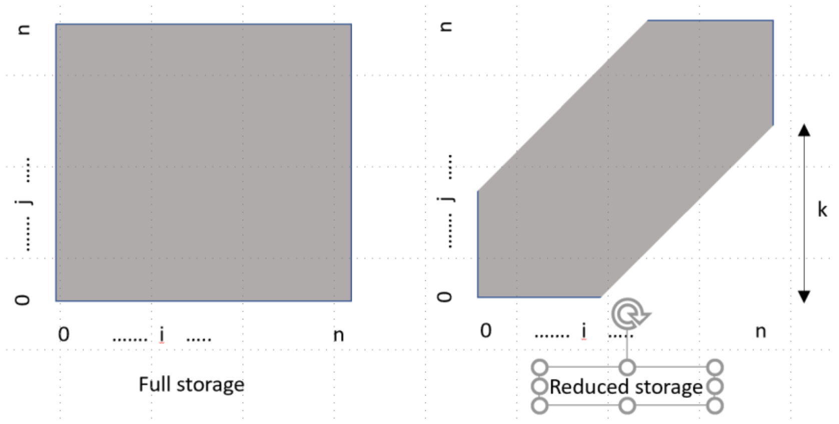
CS 5050 Midterm One 50 Points

Q1) The DNA alignment algorithm (and the minimum edit distance problem) are memory intensive. Answer the following questions.

Given two strings, both of size **n**, what is the memory required to perform an alignment?

One idea to save memory is to “cut off” the corners as illustrated in this figure, where only the grey area is allocated and used during the DP execution. How much memory would this save?

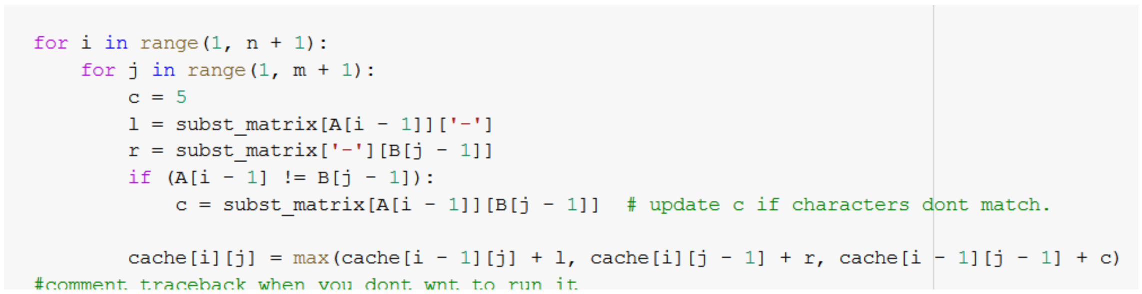


For the simple edit distance problem (where one edit step is scored as 1), how would this corner cut off change the algorithm’s behavior. Be precise.

What factor (as a function of **k**) would this change the run time?

Solution:

If we look at the dynamic programming algorithm:



```
for i in range(1, n + 1):
    for j in range(1, m + 1):
        c = 5
        l = subst_matrix[A[i - 1]]['-']
        r = subst_matrix['-'][B[j - 1]]
        if (A[i - 1] != B[j - 1]):
            c = subst_matrix[A[i - 1]][B[j - 1]]  # update c if characters dont match.

        cache[i][j] = max(cache[i - 1][j] + l, cache[i][j - 1] + r, cache[i - 1][j - 1] + c)
#comment traceback when you dont wnt to run it
```

Time: O(NM)
Space: O(NM)
If both the sequences are of length N then,
Time: O(N^2)
Space: O(N^2)

This corner cutoff is basically when the two sequences to be aligned are highly similar, it is sufficient to perform the dynamic programming algorithm in a band around the main diagonal. If the alignment path remains within the band, this algorithm achieves an $O(wN)$ time and space requirement.
w = n-k.

```
for i=1 to i = N):
  for j max(1, n - k) to min(M, n+k):
      c = 5
      l = subst_matrix[A[i - 1]]['-']
      r = subst_matrix['-'][B[j - 1]]
     if (A[i - 1] != B[j - 1]):
        c = subst_matrix[A[i - 1]][B[j - 1]]
      cache[i][j] = max(
            cache[i - 1][j] + l,          ->   (if j<i+k(M))
            cache[i][j - 1] + r,          ->   (if j>i-k(M))
            cache[i - 1][j - 1] + c)
```

We can align N and M more efficiently; in the worst case the runtime would still be $O(N^2)$.

Time: O (N ×(N-k) << $O(N^2)$ (Run time becomes somewhat faster)
Space: O (N ×(N-k) << $O(N^2)$ (Memory is saved)

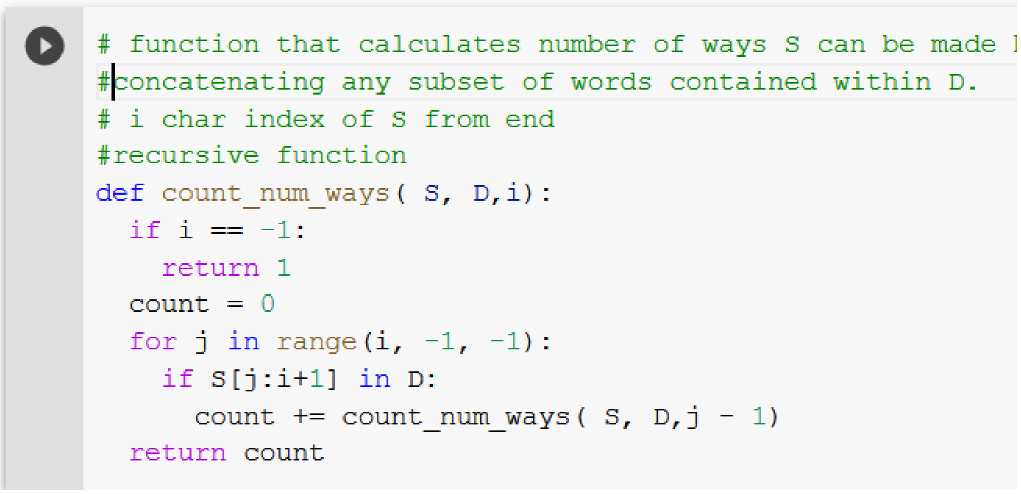
Q2) Here is a variant of one of the best known google interview questions: Given a dictionary **D** which contains a set of words (strings) and a string **S**, determine ***how many ways*** **S** can be made from concatenating any subset of words contained within **D**.

If **D** = {“cat”, “dog”, “jim”, “fred”, “jimmy”, “my”,” ed”, ”ment”, “em”, “body”, “j”, “i”}

And **S** = “jimmy”, your code would return 2.

Write the recursive, DP algorithm and trace back algorithms to solve this problem.

Recursive Algo:

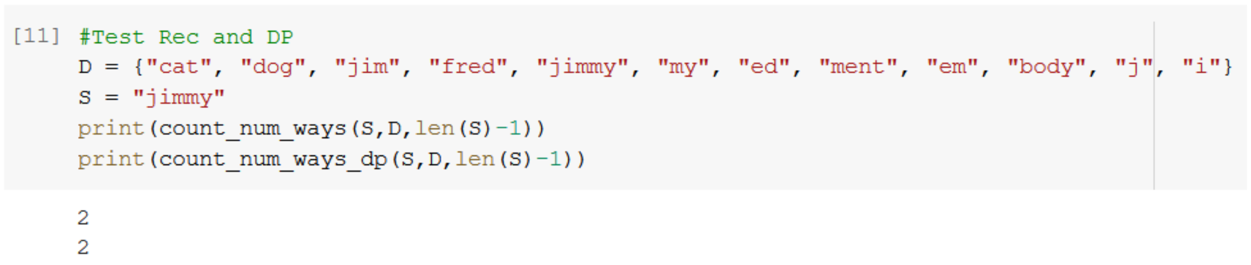


```
# function that calculates number of ways S can be made
#concatenating any subset of words contained within D.
# i char index of S from end
#recursive function
def count_num_ways( S, D,i):
  if i == -1:
    return 1
  count = 0
  for j in range(i, -1, -1):
    if S[j:i+1] in D:
      count += count_num_ways( S, D,j - 1)
  return count
```

Dynamic Programming Algorithm:

```
#DP program from recursive function
def count_num_ways_dp(S, D,i):
  N = len(S)
  cache = [0 for _ in range(0, N + 1)]
  cache[0] = 1
  for i in range(1, N + 1):
    for j in range(i, 0, -1):
      if S[j-1:i] in D:
        cache[i] += cache[j-1]
  if cache[N]: # have a solution
    return count_trace_back(S, cache)
  else:
    print("No solution")
```

Output for DP and Recursive Algorithm:



```
[11] #Test Rec and DP
     D = {"cat", "dog", "jim", "fred", "jimmy", "my", "ed", "ment", "em", "body", "j", "i"}
     S = "jimmy"
     print(count_num_ways(S,D,len(S)-1))
     print(count_num_ways_dp(S,D,len(S)-1))

     2
     2
```

Traceback:

```
def count_trace_back(s, cache):
    words=[]
    count =0
    n = len(s)
    while True:
        if n==0:
            return words
        for j in range(n, 0, -1):
            if S[j:n+1] in D and cache[n-j]
              indx= j# remember the index
              count += cache[j - 1]
              break
        words.append(s[n-indx:n])
        n= n-j
    return words;
```

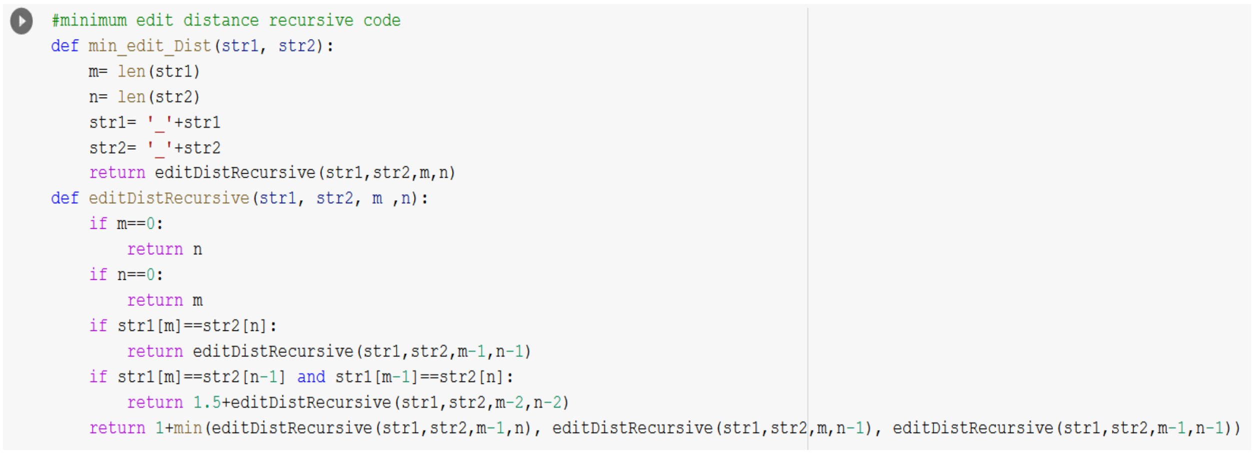
Q3) The simple minimum edit distance algorithm is where a deletion, insertion or substitution is scored as 1 and the goal is to find the minimum number of edit steps. However, many kinds of typing errors (and mutations in DNA) involve switching the order of two adjacent letters. For instance, a classic when typing is to produce "hte" rather than "the". Answer the following questions:

Develop a recursive algorithm that takes two strings and returns the minimum edit distance where a letter deletion, insertion or substitution is counted as 1. The new algorithm will include the edit step where two characters are switched. This will count at 1.5

Convert your code to a DP algorithm. Write the traceback routine and show it works on some simple problems.

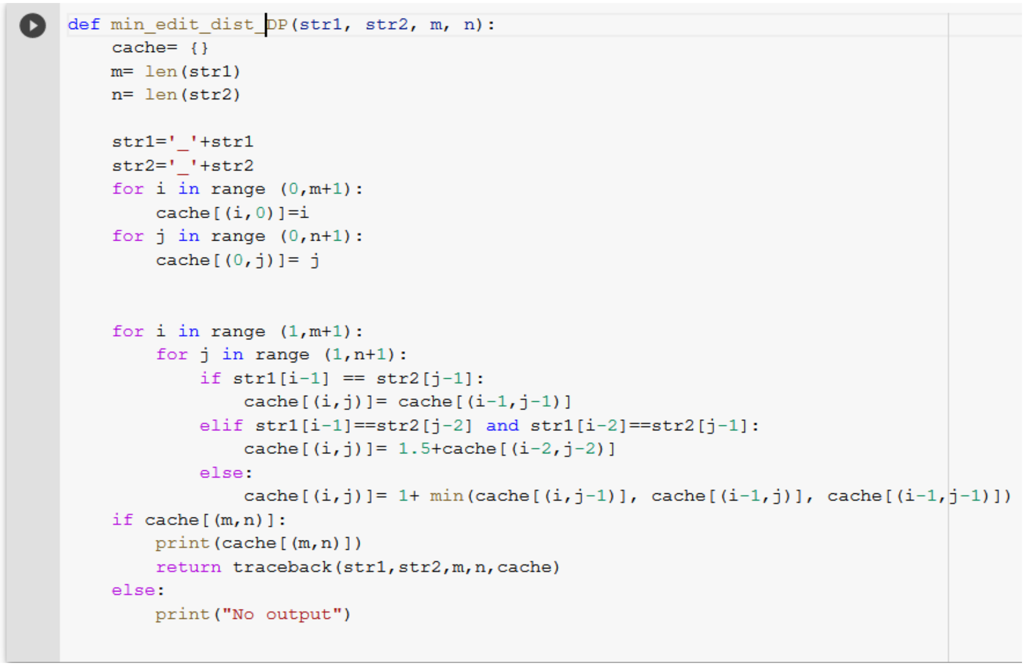
Solution:

a. **Recursive Code**



```
#minimum edit distance recursive code
def min_edit_Dist(str1, str2):
    m= len(str1)
    n= len(str2)
    str1= '_'+str1
    str2= '_'+str2
    return editDistRecursive(str1,str2,m,n)
def editDistRecursive(str1, str2, m ,n):
    if m==0:
        return n
    if n==0:
        return m
    if str1[m]==str2[n]:
        return editDistRecursive(str1,str2,m-1,n-1)
    if str1[m]==str2[n-1] and str1[m-1]==str2[n]:
        return 1.5+editDistRecursive(str1,str2,m-2,n-2)
    return 1+min(editDistRecursive(str1,str2,m-1,n), editDistRecursive(str1,str2,m,n-1), editDistRecursive(str1,str2,m-1,n-1))
```

b. **DP code:**

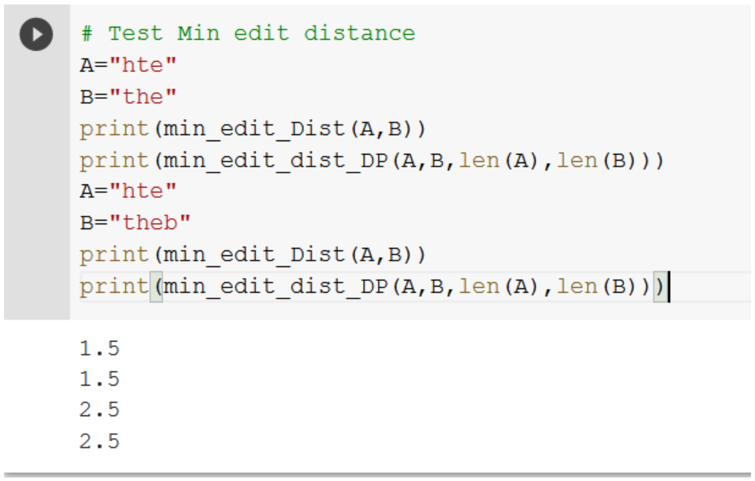


```
def min_edit_dist_DP(str1, str2, m, n):
    cache= {}
    m= len(str1)
    n= len(str2)

    str1='_'+str1
    str2='_'+str2
    for i in range (0,m+1):
        cache[(i,0)]=i
    for j in range (0,n+1):
        cache[(0,j)]= j


    for i in range (1,m+1):
        for j in range (1,n+1):
            if str1[i-1] == str2[j-1]:
                cache[(i,j)]= cache[(i-1,j-1)]
            elif str1[i-1]==str2[j-2] and str1[i-2]==str2[j-1]:
                cache[(i,j)]= 1.5+cache[(i-2,j-2)]
            else:
                cache[(i,j)]= 1+ min(cache[(i,j-1)], cache[(i-1,j)], cache[(i-1,j-1)])
    if cache[(m,n)]:
        print(cache[(m,n)])
        return traceback(str1,str2,m,n,cache)
    else:
        print("No output")
```

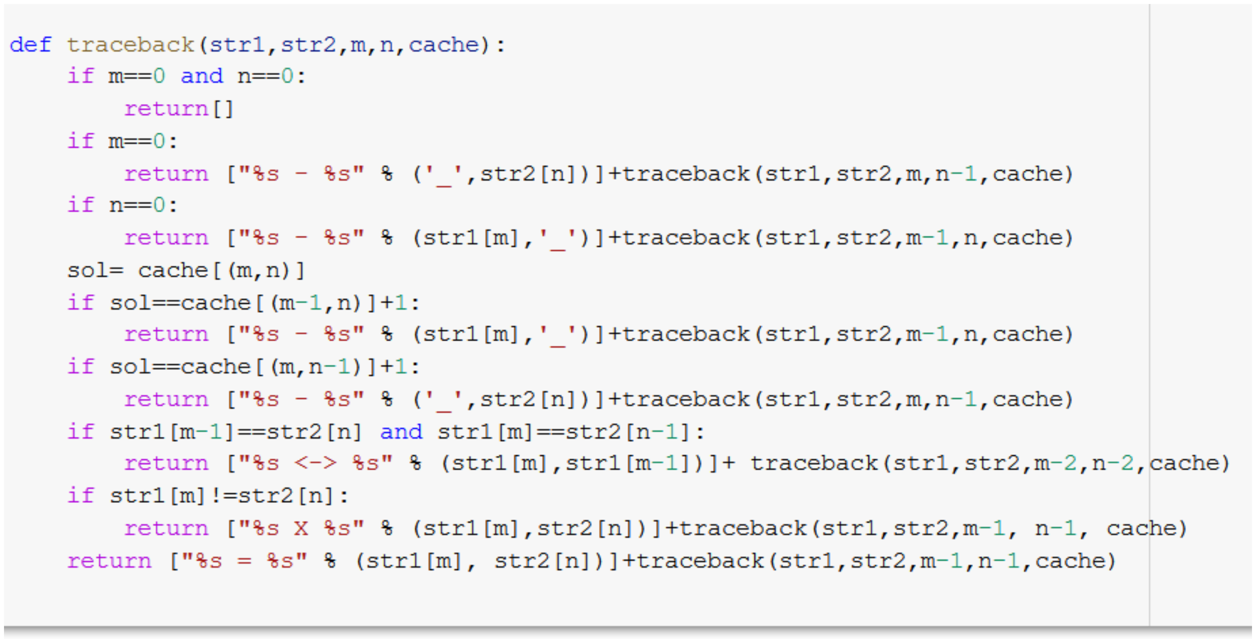
c. **Test Output**



```
# Test Min edit distance
A="hte"
B="the"
print(min_edit_Dist(A,B))
print(min_edit_dist_DP(A,B,len(A),len(B)))
A="hte"
B="theb"
print(min_edit_Dist(A,B))
print(min_edit_dist_DP(A,B,len(A),len(B)))
```

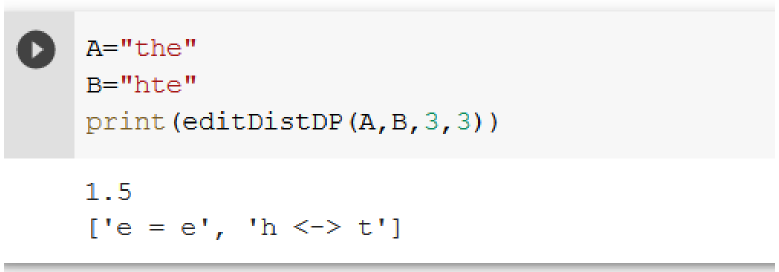
```
1.5
1.5
2.5
2.5
```

d. **DP with Traceback**



```
def traceback(str1,str2,m,n,cache):
    if m==0 and n==0:
        return[]
    if m==0:
        return ["%s - %s" % ('_',str2[n])]+traceback(str1,str2,m,n-1,cache)
    if n==0:
        return ["%s - %s" % (str1[m],'_')]+traceback(str1,str2,m-1,n,cache)
    sol= cache[(m,n)]
    if sol==cache[(m-1,n)]+1:
        return ["%s - %s" % (str1[m],'_')]+traceback(str1,str2,m-1,n,cache)
    if sol==cache[(m,n-1)]+1:
        return ["%s - %s" % ('_',str2[n])]+traceback(str1,str2,m,n-1,cache)
    if str1[m-1]==str2[n] and str1[m]==str2[n-1]:
        return ["%s <-> %s" % (str1[m],str1[m-1])]+ traceback(str1,str2,m-2,n-2,cache)
    if str1[m]!=str2[n]:
        return ["%s X %s" % (str1[m],str2[n])]+traceback(str1,str2,m-1, n-1, cache)
    return ["%s = %s" % (str1[m], str2[n])]+traceback(str1,str2,m-1,n-1,cache)
```

Traceback output:



```
A="the"
B="hte"
print(editDistDP(A,B,3,3))
```

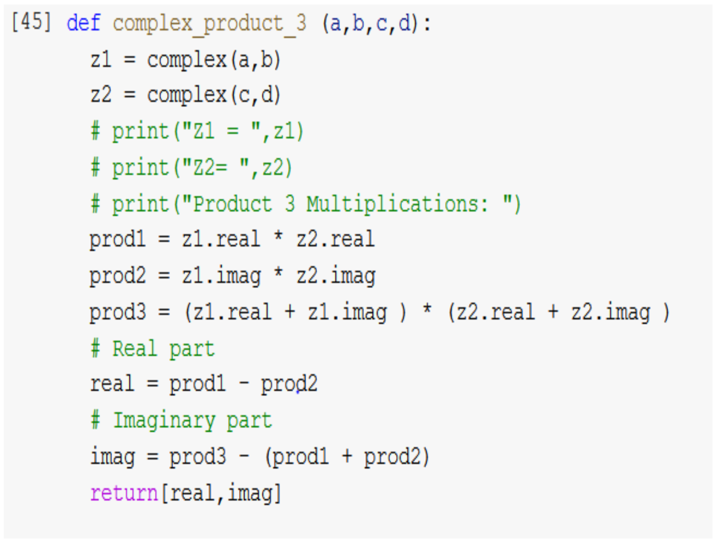
```
1.5
['e = e', 'h <-> t']
```

Q4) Multiplying two complex numbers using the traditional approach requires four multiplies. Perform the following steps:

Develop a faster algorithm that only requires three multiplies. Write two functions, one that utilizes four multiples, and one that only uses three. These functions should take four floats (real and imaginary values for the two numbers and return two numbers, the real and imaginary parts of the product). Include the code below:

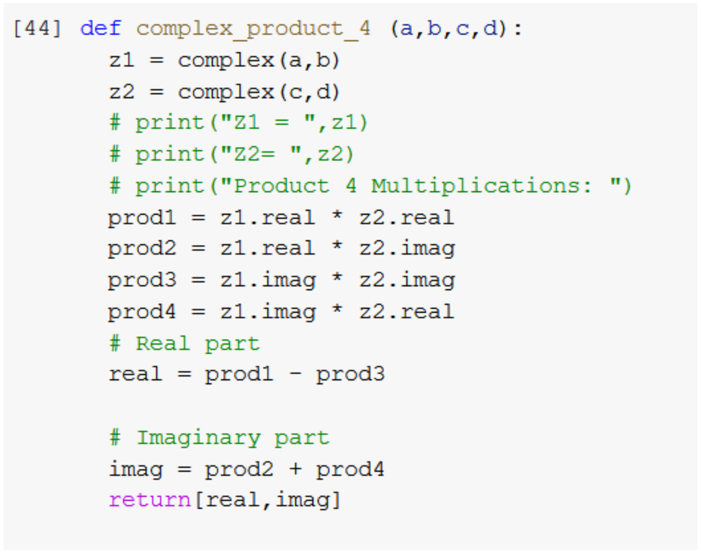
Solution:

Function takes four floats as parameters, converts them in 2 complex numbers and computes their product (3 multiplies) to give real and imaginary.



```
[45] def complex_product_3 (a,b,c,d):
        z1 = complex(a,b)
        z2 = complex(c,d)
        # print("Z1 = ",z1)
        # print("Z2= ",z2)
        # print("Product 3 Multiplications: ")
        prod1 = z1.real * z2.real
        prod2 = z1.imag * z2.imag
        prod3 = (z1.real + z1.imag ) * (z2.real + z2.imag )
        # Real part
        real = prod1 - prod2
        # Imaginary part
        imag = prod3 - (prod1 + prod2)
        return[real,imag]
```

Function takes four floats as parameters, converts them in 2 complex numbers and computes their product (4 multiplies) to give real and imaginary



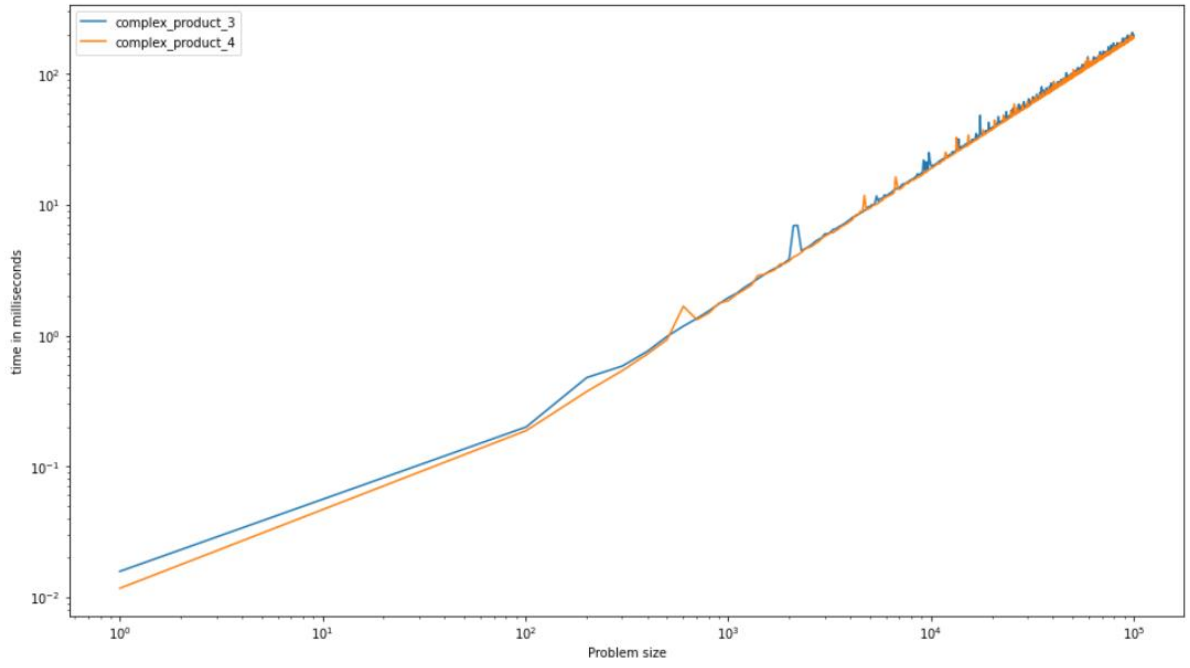
```
[44] def complex_product_4 (a,b,c,d):
        z1 = complex(a,b)
        z2 = complex(c,d)
        # print("Z1 = ",z1)
        # print("Z2= ",z2)
        # print("Product 4 Multiplications: ")
        prod1 = z1.real * z2.real
        prod2 = z1.real * z2.imag
        prod3 = z1.imag * z2.imag
        prod4 = z1.imag * z2.real
        # Real part
        real = prod1 - prod3

        # Imaginary part
        imag = prod2 + prod4
        return[real,imag]
```

Design an empirical study that will compare the runtime performance of the two algorithms with each other. Produce a graph that illustrates the run time as a function of the number of multiplies. I recommend precomputing a list of random complex numbers and use long lists so the timing is accurate. Give your graph below:

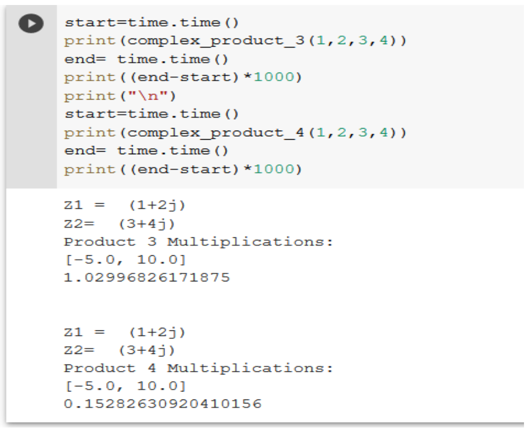
Timing study for 10^5 complex multiplies:

```
Function complex_product_3 time = 0.002510 n ^ 0.977
Function complex_product_4 time = 0.002322 n ^ 0.981
```



Fit the timing functions. Provide a brief explanation of the timing functions. How much faster is the three multiply compared to the four multiply?
Answer: If we time the functions for just multiplying 2 complex numbers, 4 multiplications algorithm works faster than 3 multiplications. They run identically for 10^5 complex multiplications. But if we see the time equations on the graph 3 multiplication algo looks slightly faster.

I used int to test if the functions work fine or not and so that I can identify the results. However for plotting the graphs I used random.uniform() functions so the functions work on 4 floats.



```
start=time.time()
print(complex_product_3(1,2,3,4))
end= time.time()
print((end-start)*1000)
print("\n")
start=time.time()
print(complex_product_4(1,2,3,4))
end= time.time()
print((end-start)*1000)
```

```
Z1 =  (1+2j)
Z2=  (3+4j)
Product 3 Multiplications:
[-5.0, 10.0]
1.02996826171875


Z1 =  (1+2j)
Z2=  (3+4j)
Product 4 Multiplications:
[-5.0, 10.0]
0.15282630920410156
```

Q5) Memoizing vs. DP: Answer the following questions:
DP is an eager algorithm while memorizing is a lazy algorithm. Define these two terms.
Both dynamic programming and memorization algorithms are applicable only when the problem can be divided into subproblems and these subproblems are not independent (subproblems share sub-subproblems).
Solution:
A dynamic programming algorithm solves every subproblem once stores them in cache and avoids recomputing the solution whenever the subproblem is encountered. Since it solves all the subproblems it is called as an eager algorithm. **An eager algorithm starts execution immediately and returns a result.**
In memoization a cache is used to store answer to the subproblems but when a subproblem is encountered for the first-time during execution, its solution is calculated and stored in the cache. i.e. Only the solutions to subproblems that are encountered is calculated so Memoization is termed as a lazy algorithm. **A lazy algorithm postpones computation until it is necessary to execute and then produces a result.**

2. Give an example problem that can be solved using DP that always computes all the sub-solutions.
If I use DP to solve the **Fibonacci problem** all the subproblems are computed.
For this problem, what would make a good cache data structure? Briefly justify your answer.
I think an array is a good data structure for this because it gives O(1) search and we always know the indexes.

Which is best, DP or memoizing to solve this problem. Briefly justify your answer.
I think DP is the best because ultimately Memoization has to solve all the subproblems. The Fibonaccci problem requires all subproblems to be solved, DP will outperform memoization by a constant factor because DP has no overhead for recursion and can use a preallocated array.

Give an example problem that can be solved using memoizing that often will not compute all the sub-solutions.
The Knapsack problem can be solved using memorizing and does not compute all the subproblem solutions.

For this problem, what would make a good cache data structure? Briefly justify your answer.
A good data structure for this problem is a 2D matrix that represents all subsets of the items with rows representing items and columns representing the bag's remaining weight capacity. Also, it is easier to access when we know the row and column indexes.

Which is best, DP or memoizing to solve this problem. Briefly justify your answer.
Only some of the subproblems need to be solved for the Knapsack problem to be solved, So memoization is preferrable since the subproblems are solved lazily, or precisely the computations that are needed are calculated.