CS 5050 Final 99 Points

This is an open book exam. This means you can use the internet, books, software, and personal notes to produce the answers. Think of it like a set of problems you must solve at work. There are limits though. The work you submit must be your own, so no collaboration with any other person (including but not limited to people in the class). No copy and pasting of any internet content. By submitting the completed exam, you are affirming that you have followed these rules.

You must submit your completed exam as one pdf file. You may enter your answers by editing this .doc file or printing it, writing on it and then scanning in. Or some combination of the two. Some of the answers require writing some code. For these, the best way would be to do a screen snip that shows the new code inserted into any existing code and/or as new functions. Add a few comments so I can understand what you did. The advantage of the snip is that the formatting will be preserved so the code will be easier to read.

For this last “assignment” I will not accept late work.

Please keep your answers concise. There is no need to give me an essay as an answer. There are 9 questions in this exam and they are each worth 11 points.

Q1) Use the meta algorithms to design a dynamic programming solution to the following problem:

You have solar panels that are mounted on a pole in your front yard. They have three settings that tip the panels at different angles with respect to the horizontal. As the seasons change, the settings allow you to capture more solar energy by better aligning the tipping angle of the panels with the position of the sun in the sky. Let these settings be called 0, 1, and 2.

You are given the following:

A table that maps the day-of-year (1 to 366) and the setting (0, 1, 2) to an estimate of the energy you could collect. Let this table be **E[d, s], 0 < d <= 366**, and **0 <= s < 3**. So **E[d, s]** will return the amount of energy you could collect if the panels are in setting **s** on day-of-year **d**.

Design an algorithm that will work out the days you should change from one setting to another over the year in order to maximize your solar energy collection. Note: You can only make a maximum of four changes in settings in a year.

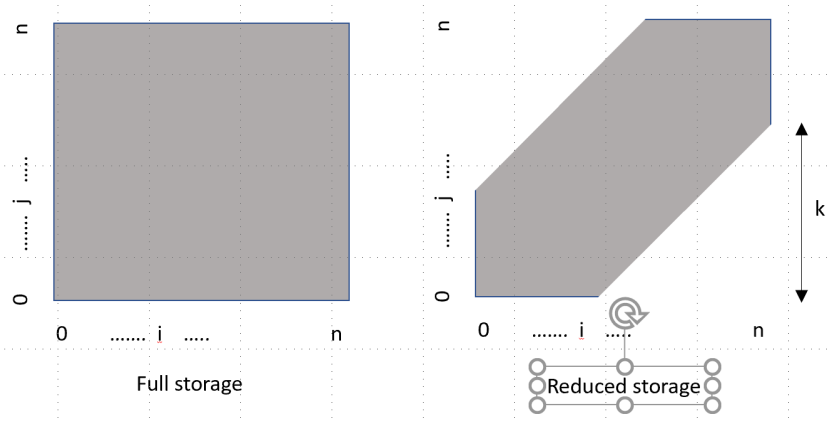
Design the recursive solution then convert it to a DP and write the traceback routine that will return the days (from 1 to 366) when an adjustment should be made and the type of adjustment ("on day 57 go to setting 2" for example)

```
import numpy as np

days = 5
E = np.reshape(np.random.random(days*3), (days, -1))
settings = [0, 1, 2]
print(E)

def solar_panels(day, curr_set, changes):
    if day >= days:
        return 0.0
    # there are no more changes
    if changes > 4:
        return sum(E[day:][curr_set])

    today_value = E[day][curr_set]
    return today_value + max(solar_panels(day + 1, setting, changes + (setting != 
curr_set)) for setting in settings)


print(solar_panels(0, 0, 0))
print(solar_panels(0, 1, 0))
print(solar_panels(0, 2, 0))
```

Q2) 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.

For the DNA alignment problem, how would this memory reduction influence the result? If you were using this result to study evolutionary relationships, how would the distance in the "tree of life" relate to an appropriate setting for **k**?

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

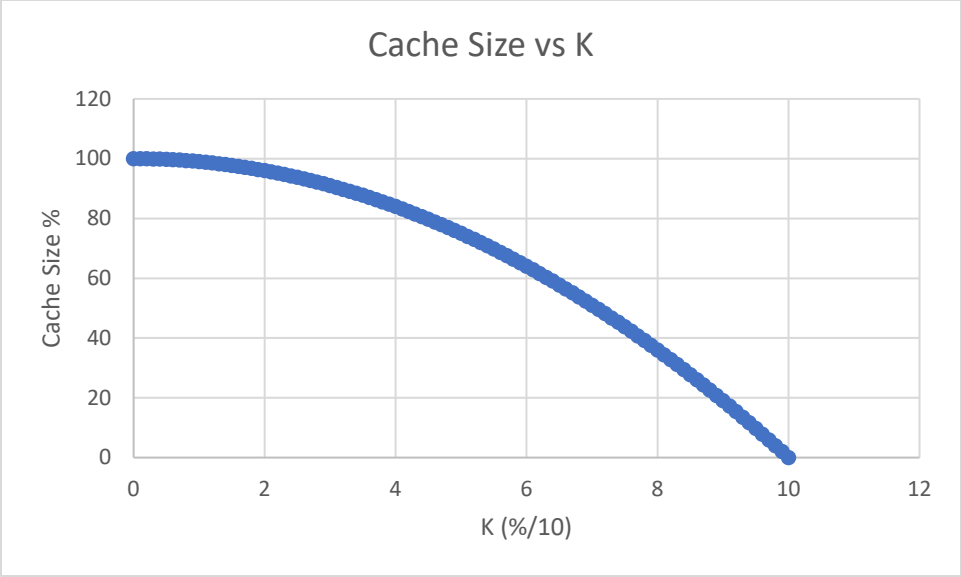
For strings of size n, you have a cache of size n^2

Cutting off the corners at the midpoint of $\frac{n}{2}$ would chop off two areas of $\frac{n}{2} \cdot \frac{n}{2} \cdot \frac{1}{2} = 0.125 \cdot n^2$. These means that if the total memory used is n^2. Then the memory saved by chopping off the ends would be $0.125 \cdot 2 \cdot n^2 = 0.25 \cdot n^2$, and the dp algorithm would use only $\frac{3}{4}$ the normal amount, if $k = \frac{n}{2}$

For a simple edit distance problem, this would be implemented by using a continue in the loop if the result of the edit was too large. The reason for this is that the corners of the cache will never be used because they represent an entire string being changed. So, this reduced cache would work most of the time, but might fail or cause strange results for comparisons that are completely different and would normally need to traverse a solution path into the corner.

By increasing the factor K, it forces the algorithm to take a more diagonal path. In term of the tree of life, increasing K would cause the solution to have more substitutions and less insertions. This might be worse and be erroneous.

As k approaches n, the memory needed goes to 0.


Cache Size vs K
120
100
80
60
40
20
0
Cache Size %
0
2
4
6
8
10
12
K (%/10)

Q3) 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”, “embodiment”, “ i”}

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

Write the recursive and DP algorithms to solve this problem.

```
def concat_set(m, goal, D):
    # goal is met
    if goal == "":
        return 1
    # there's no more words in the list
    if not D:
        return 0

    sum = 0
    for i in range(len(D)):
        item = D[i]
        if goal.startswith(item):
            sum += concat_set(m - 1, goal[len(item):], D[0: i] + D[i+1:])

    return sum


D = ["cat", "dog", "jim", "fred", "jimmy", "my", "ed", "ment", "em", "body", "embodiment", "i", "happy", 'h', 'app', 'y']
print(concat_set(len(D), "happy", D))
```

Q4) Write the simple minimum edit distance algorithm where a deletion, insertion or substitution is scored as 1 and the goal is to find the minimum number of edit steps. Answer the following questions:

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”

Modify your recursive algorithm to include this kind of edit step and count it as 1.

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

```python
# recursion is slow
def med_recur(i, j):
    if i <= 0:
        return j
    if j <= 0:
        return i
    # del from i, del from j, substitution, swap
    deli = med_recur(i - 1, j) + 1
    delj = med_recur(i, j - 1) + 1
    sub = med_recur(i - 1, j - 1) + (A[i] != B[j])
    swap = 1000000
    if i > 1 and j > 1:
        swap = med_recur(i - 2, j - 2) + 2 - (A[i-1] == B[i] and A[i] == B[i-1])

    return min(deli, delj, sub, swap)
```

```
print(med('abcdefgh', 'badcfehg'))
```

```
recursive 4
```

```
print("dp", med('abc', 'bad'))
```

```
recursive 2
dp 2
```

```
print("dp", med('the', 'hte'))
```

```
recursive 1
dp 1
```

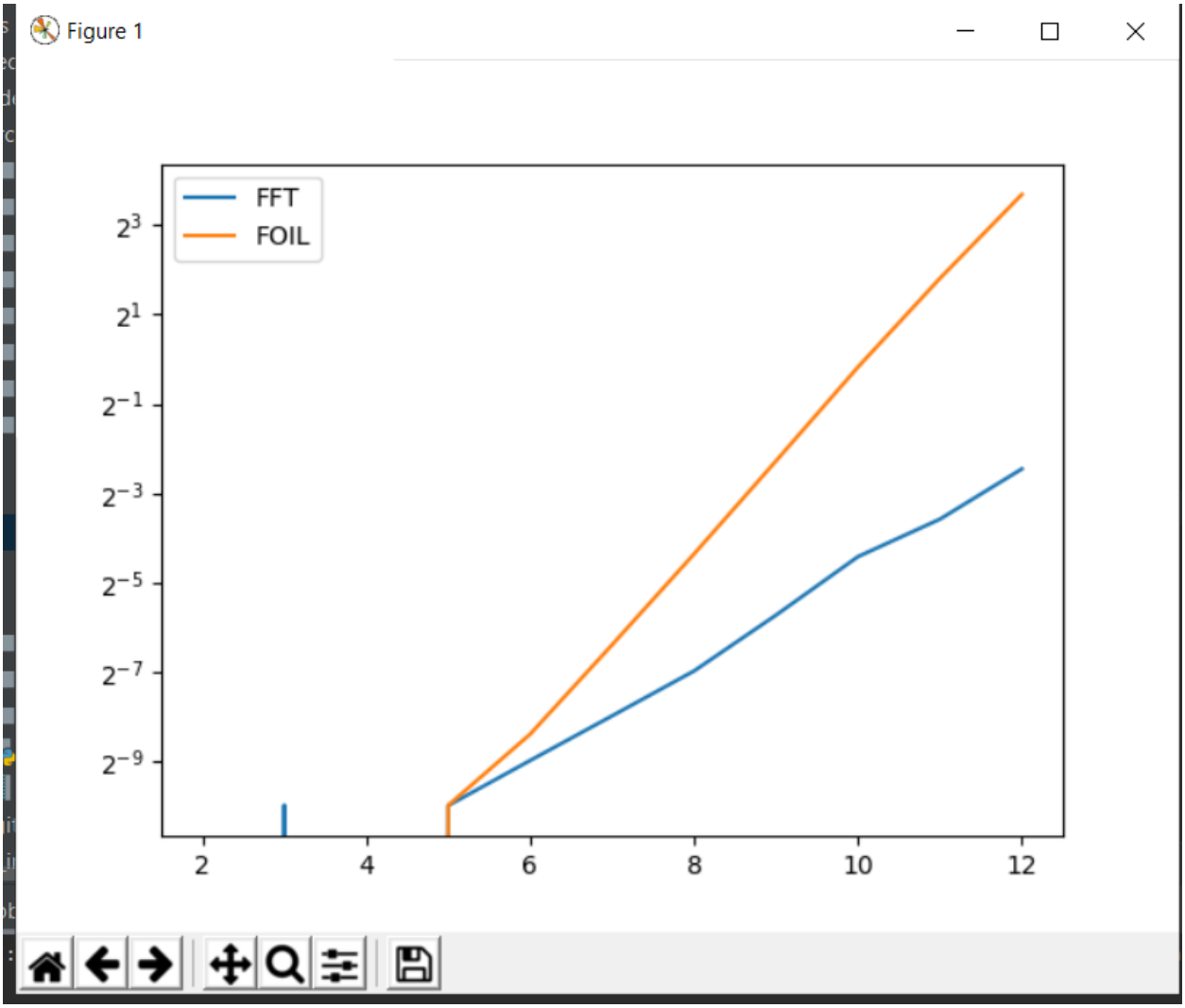
```python
# dynamic program is fast
def med_dp(i, j):
    # initialize cache with base case along axis
    c = np.zeros([i + 1, j + 1], dtype=int)
    for x in range(i + 1):
        c[x, 0] = x
    for y in range(j + 1):
        c[0, y] = y

    swap = 10000000
    # fill in cache
    for x in range(1, i + 1):
        for y in range(1, j + 1):
            insert = c[x - 1, y] + 1
            deletion = c[x, y - 1] + 1
            substitution = c[x - 1, y - 1] + (A[x] != B[y])
            if x > 1 and y > 1:
                swap = c[x - 2, y - 2] + 2 - (A[x-1] == B[y] and A[x] == B[y-1])
            c[x, y] = min(insert, deletion, substitution, swap)

    # return answer
    return c[i, j]
```

Q5) We have seen a few examples where we can identify two algorithms that solve the same problem: A simple or naïve algorithm that solve small problems quickly, but becomes impractical as the problems get large; and a clever algorithm that takes more time on smaller problems, but has a much slower growth rate in run time as problems get large. Answer the following questions:

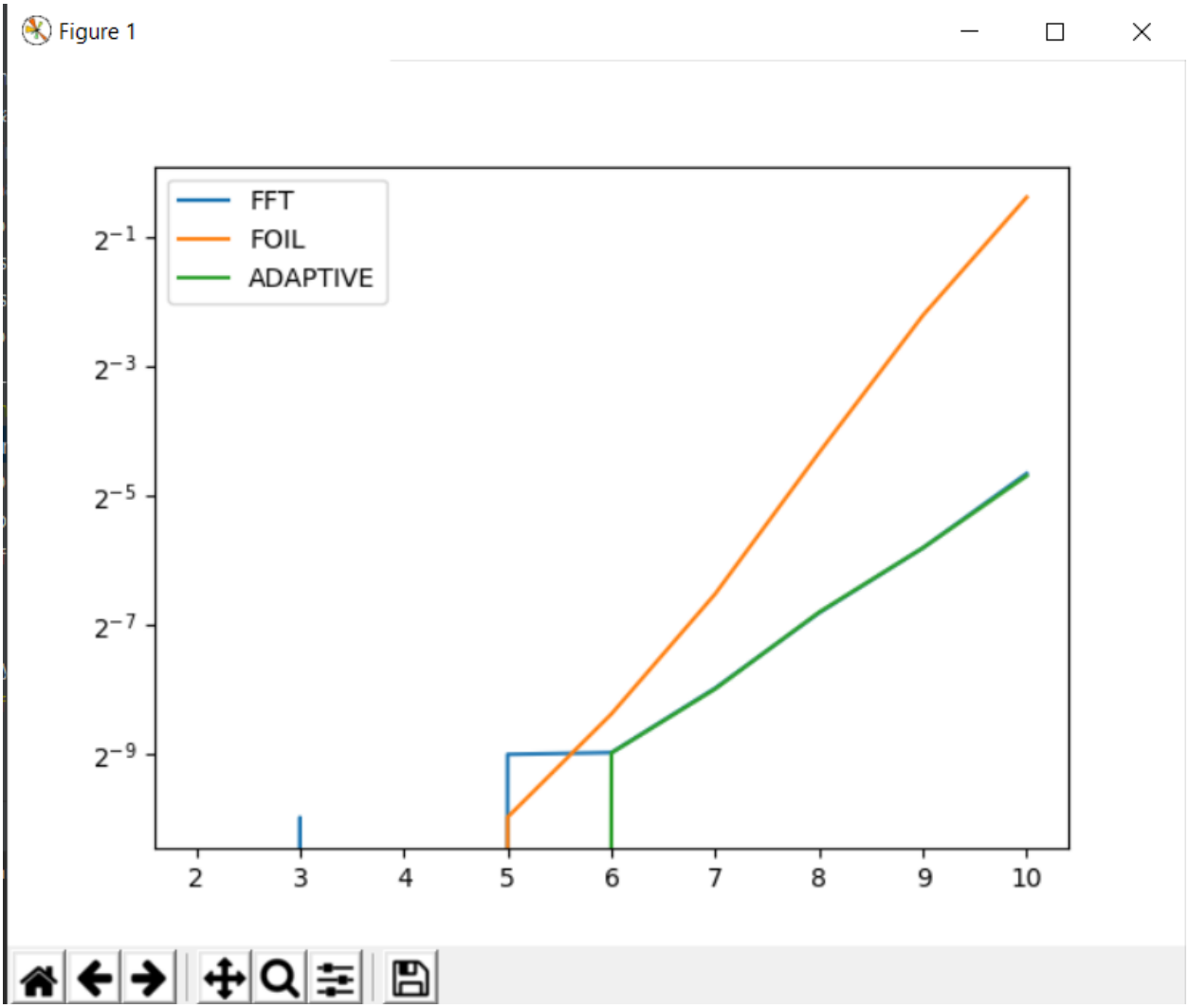
Draw or show a graph that illustrates this pattern for a problem we solved in class. Make sure it shows the crossing point of run times of the algorithms.

Write some simple code that switches between the two algorithms based on these empirical results in the graph. What would be the run time of this algorithm? Mark it on the graph.

Considering the polynomial multiplication problem and two algorithms that solve this problem: "school" and the "three subproblem", design an algorithm that uses each, but is faster than the simple switch algorithm. Show the algorithm below. No need to code and run this, you can just type up or write the algorithm.



Here's a polynomial multiply problem where FFT multiply is better than the simple FOIL multiply where n > 2^5



This is using an adaptive algorithm that switches between the foil and fft algorithms. As shown it swap at 2^6 and that is where the green line deviates from the blue line

```
def smart_multiply(arr1, arr2):
    if len(arr1) < 2**6:
        return foil_multiply(arr1, arr2)
    else:
        fft_multiply(arr1, arr2)
```

An algorithm that would be faster using the “3 sub” algorithm and the high school algorithm would be to initially use the 3 sub algorithm, which recursively breaks down the problem. In the recursion stack, if the size of the sub problem is less than 2^6 (where foiling is faster), the it hands off that sub problem to foil to finish it off. Then is reconstructs the answer on the way up the recursion stack.

Q6) Consider algorithm evaluation methods, both analytical and empirical.

a. For a recursive algorithm that generates 7 sub-problems of ¼ size with an overhead of *O(n)* each call, write the recurrence relation.

$$T(n) = T\left(\frac{n}{4}\right) \cdot 7$$

b. Solve the recurrence relation using the master "cook book" method
A=7
B=4
K=1

$$4^1 = 4 < 7$$
$$O(n^1)$$

$$T(n) = 7T\left(\frac{n}{4}\right) + O(n^1)$$

c. Briefly describe an empirical study you would perform to evaluate this algorithm and draw a graph of expected performance. Describe how you would solve for the actual function mapping problem size to run time given some timing results.

The empirical study would have run with various sizes of n and keep track of the run time of the algorithm with various sizes of n. I would expect this algorithm's runtime to linear compared to the sizes of n. The cookbook method ignores any kind of constant that may be multiplying as well. Once I did this study, I would solve for the slope of the line in order to find the algorithms runtime constant. This way I could verify that the runtime is linear and find the true runtime of any size N.

d. For a recursive algorithm that generates 3 sub-problems of *n-1, n-2* and *n-3* problem sizes, write the recurrence relation

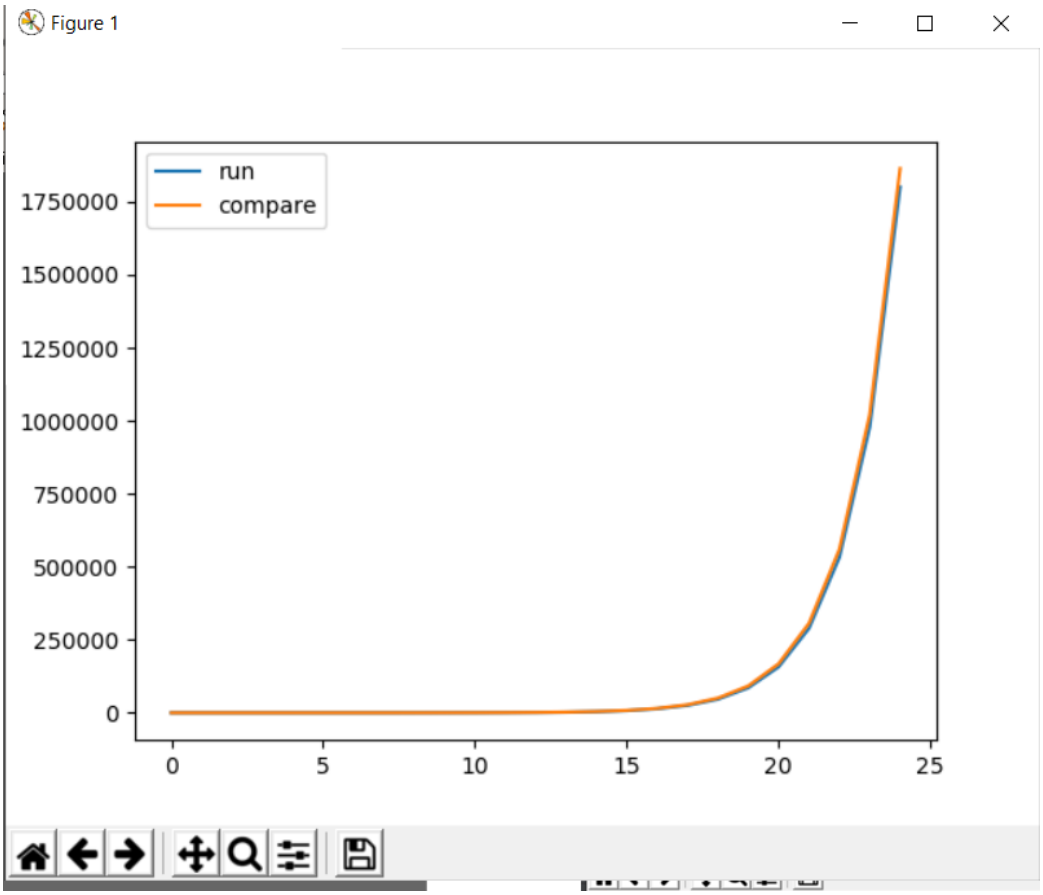
$$T(n) = T(n-1) + T(n-2) + T(n-3)$$

e. Estimate the solution to the recurrence relation in the "big O" notation.

Since decreasing the problem size n by 1,2, or 3 is relatively nothing compared to large n's, we can consider that for each n, it will run again three time at n -1. This decreases by one every time, so the runtime for this algorithm would be $= T(n) \approx 3 \cdot T(n-1)$

Therefore $O(3^n)$

f. Briefly describe an empirical study you would perform to evaluate this algorithm and draw a graph of expected performance. Describe how you would solve for the actual function mapping problem size to run time.




```python
def s(n):
    if n <= 1:
        return 1
    else:
        return s(n-1) + s(n-2) + s(n-3)

size = []
run = []
compare = []
for i in range(0, 25):
    size.append(i)
    run.append(s(i))
    compare.append(1.825**i)

plt.plot(size, run, label='run')
plt.plot(size, compare, label='compare')
plt.legend()
plt.show()
```

I would model the run time of the algorithm with S and compare that with an exponential growth. As shown in my empirical study, I found this algorithm to have a runtime of $O(1.825)^n$

Q7) Read up on the traveling salesperson problem. Some simple code that solves the problem is included as part of this assignment (see canvas). The code enumerates all possible tours and keeps track of the best so far. When it returns, the global variable bestSolution will contain the shortest tour. Answer the following questions:

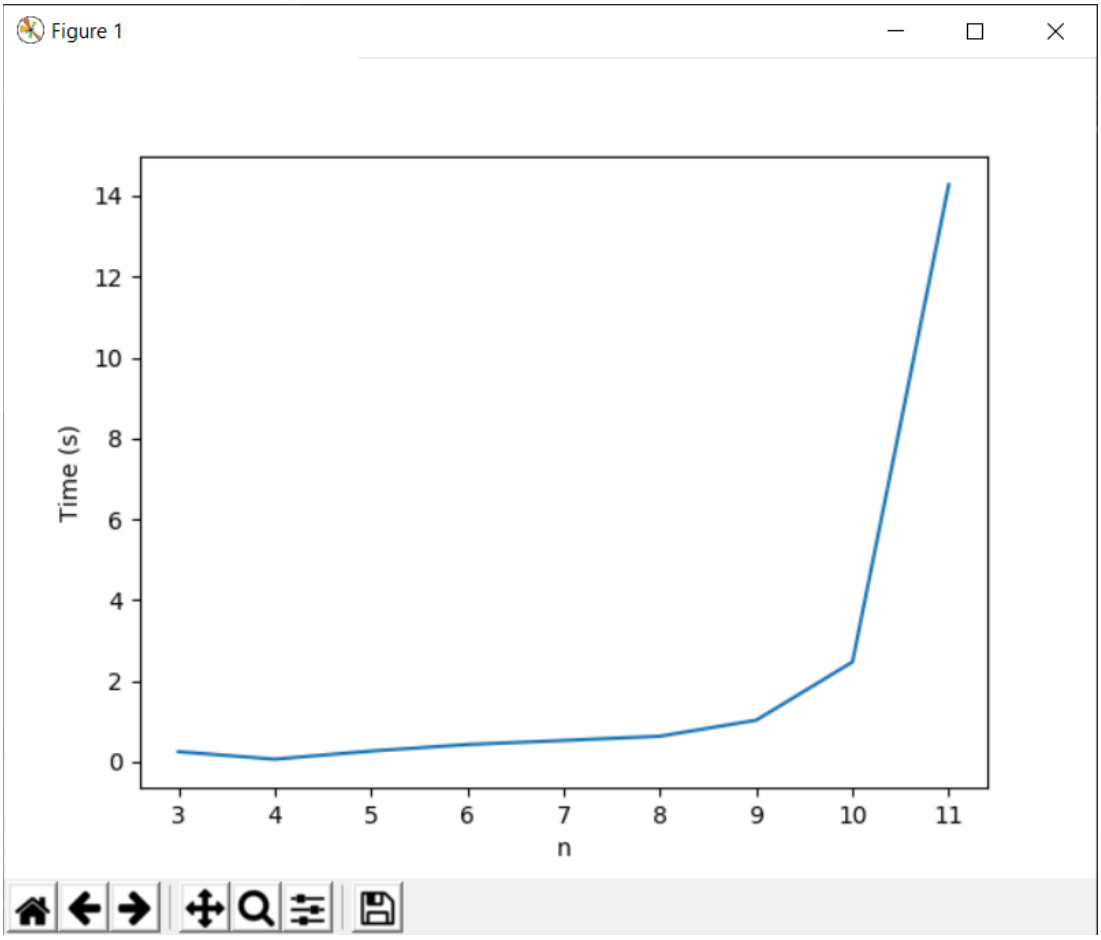
Write the recurrence relation for this algorithm that counts the number of base cases.

Write the solution to this recurrence relation.

$$T(n) = T($$

Run the code for some simple problems going from 3 to 11 (don't go too much higher) and time the computation.

Determine the appropriate graph for illustrating run time as a function of problem size and plot it.



Consider the following observations:

a) The recursive algorithm keeps track of the length of the tour so far
b) The distances between cities are always positive so tours can only get longer as they are extended

c) The algorithm has a value for the best so far solution at every function call

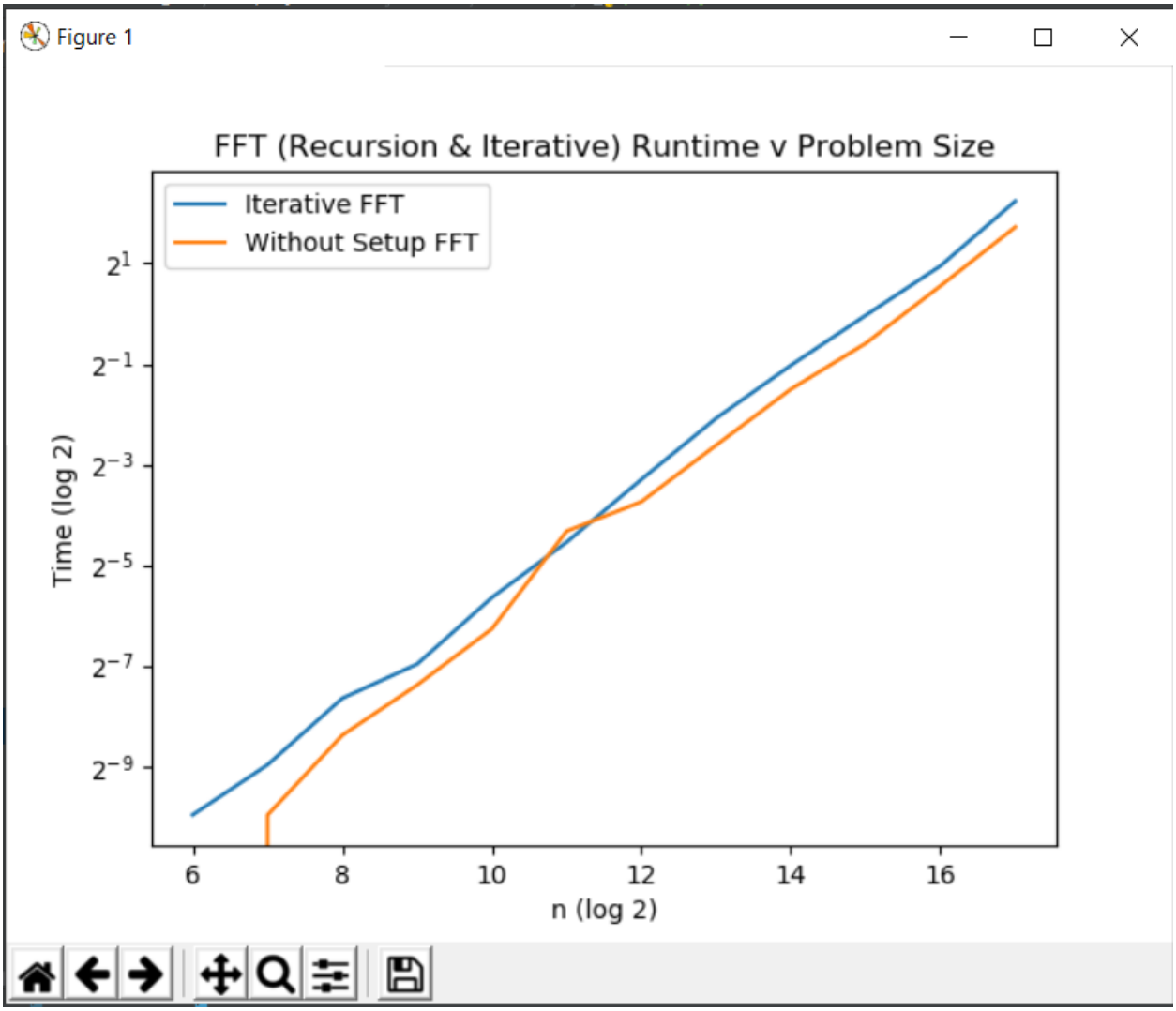
Add an additional termination condition to the recursive algorithm that will exploit these observations to avoid unnecessary computation. Repeat the calculation above and plot the graph of this new (potentially quicker algorithm).

Q8) Many students were surprised to find that the iterative version of the FFT was not clearly faster than the recursive FFT. One reason is that the following steps are repeated each time: a) allocating the solution array, computing the omega tables, and computing the reverse-bit-shuffle to fill in the base cases.

In general, these steps may be done once only and used every time from there on. This is called amortization, where the time taken is spread out over all future runs of the algorithm. Do the following:

Rerun your iterative code, not counting these three setup steps and compare the results. Show a graph of the two versions of the code, one counting the setup steps, the other without. What is the speedup?

It sped up by a small amount every time. On the graph, it appears to be constant and the line runs parallel to the no-setup fft, but because of the scaling, the speed up is also exponential. It's looks to be around $2^{0.5}$.




```
import sys
import matplotlib.pyplot as plt
```

```
import time
import numpy as np


def timer(p, function, print_line=False):
    start = time.time()
    if print_line:
        print(function(p))
    else:
        function(p)
    return time.time() - start


def setup(p, n):
    logn = np.int(np.log2(n))
    cache = np.full((logn + 1, n), np.complex)

    # reverse bit shuffle
    for i in range(n):
        cache[0, i] = np.complex(p[rbs(i, logn)])
    return cache


def fft_no_setup(p,w,n, cache):
    logn = np.int(np.log2(n))
    # fill in cache from base case row up
    for i in range(1, logn + 1):

        # squares the imaginary roots enough to mimic recursion
        omegas = w
        for s in range(logn - i):
            omegas = np.power(omegas, 2)

        # fills in cache side to side
        size = 1 << i
        for j in range(0, n, size):
            idx = 0
            for k in range(size // 2):
                cache[i][j + k] = cache[i - 1][j + k] + omegas[idx] * cache[i - 1][j
+ k + size // 2]
                cache[i][j + size // 2 + k] = cache[i - 1][j + k] - omegas[idx] *
cache[i - 1][j + k + size // 2]
                idx += 1

    return cache[logn]

def fft_inter(p, w, n):
    logn = np.int(np.log2(n))
    cache = np.full((logn + 1, n), np.complex)

    # reverse bit shuffle
    for i in range(n):
        cache[0, i] = np.complex(p[rbs(i, logn)])

    # fill in cache from base case row up
```

```
    for i in range(1, logn + 1):

        # squares the imaginary roots enough to mimic recursion
        omegas = w
        for s in range(logn - i):
            omegas = np.power(omegas, 2)

        # fills in cache side to side
        size = 1 << i
        for j in range(0, n, size):
            idx = 0
            for k in range(size // 2):
                cache[i][j + k] =             cache[i - 1][j + k] + omegas[idx] *
cache[i - 1][j + k + size // 2]
                cache[i][j + size // 2 + k] = cache[i - 1][j + k] - omegas[idx] *
cache[i - 1][j + k + size // 2]
                idx += 1

    return cache[logn]


# generates the omegas for the fft algorithm
def gen_omegas(n, sign=-1):
    return np.array([complex(np.cos(2*np.pi*i/n), sign * np.sin(2*np.pi*i/n)) for i
in range(0, n)])


# reverse bit shuffle
def rbs(i: int, logn: int):
    N = 1 << logn
    iter = i
    for j in range(1, logn):
        i >>= 1
        iter <<= 1
        iter |= (i & 1)
    iter &= N-1
    return iter

# adds buffer so that length of arr is a factor of 2. i.e. len(arr) % 2 == 0
def add_buffer(arr):
    i = 0
    n = 0
    while n != len(arr):
        n = 2**i
        # requires a buffer to have len of power of 2
        if n > len(arr):
            return np.concatenate((arr, np.zeros([n - len(arr), ])))

        # N is still smaller than arr length
        else:
            i += 1

    return arr

# helper function
```

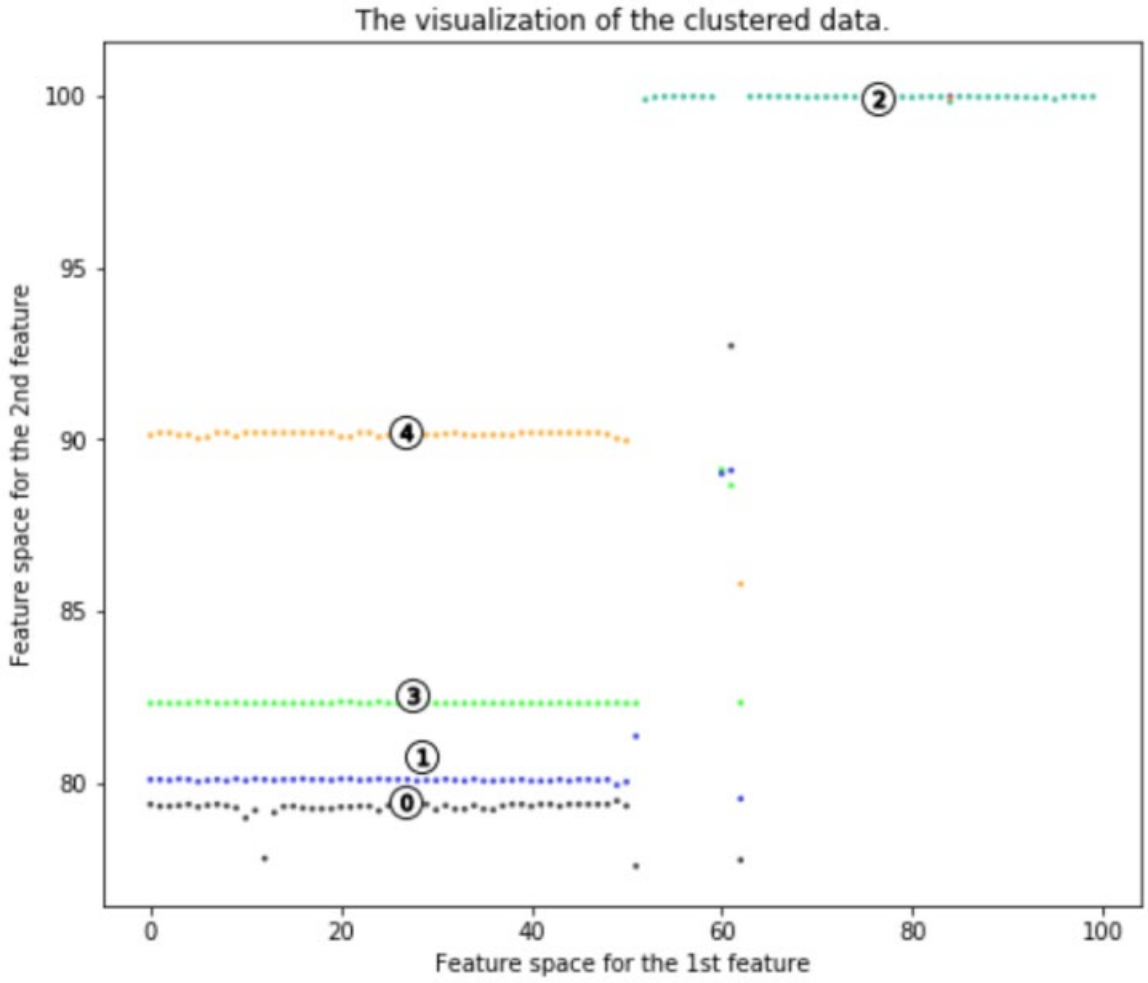
```python
# p -> polynomial
def fft(p):
    l = len(p)
    return fft_inter(add_buffer(p), gen_omegas(l), l)


if __name__ == "__main__":
    sizes = []
    iter_time = []
    without_setup_time = []
    end = 18
    for i in range(6, end):
        arr = np.random.randint(0, 1000000, 2 ** i)
        sizes.append(i)
        iter_time.append(timer(arr, fft))
        cache = setup(arr, len(arr))
        omegas = gen_omegas(len(arr))
        start = time.time()
        fft_no_setup(arr, omegas, len(arr), cache)
        without_setup_time.append(time.time() - start)

    plt.plot(sizes, iter_time, label="Iterative FFT")
    plt.plot(sizes, without_setup_time, label="Without Setup FFT")
    plt.yscale('log', basey=2)
    plt.title("FFT (Recursion & Iterative) Runtime v Problem Size")
    plt.xlabel("n (log 2)")
    plt.ylabel("Time (log 2)")
    plt.legend()
    plt.savefig(f"Recur_Iter_FFT_n={end}.png")
    plt.show()
```

Q9) COVID-19 is a new virus that has mutated from previously observed viruses like SARS. COVID-19 continues to mutate as it spreads through the population. One of our assignments was to determine the mutation “distance” between two DNA sequences and identify the most likely mutations that occurred to change one sequence to the other. Do some research and identify and briefly describe three uses of the DNA alignment algorithm that is helping scientists understand COVID-19 and help find treatments to save lives.

One thing that the algorithm is doing is identifying how far the current strains of COVID-19 have deviated from the original strands in bats and the first infected people in Wuhan, China.

Another thing that the DNA alignment algorithm is doing is showing how fast Corona -Type viruses can mutate and change. By analyzing the various changes, scientists can view how each host influences the change of the virus.

The third thing that the DNA alignment algorithm does is it quickly shows the most likely mutation. This allowed one lab in San Diego to sequence and develop a vaccine in 3 hours. I suspect that most of this time was used for the algorithm to run.



	query acc.ver	subject acc.ver	% identity	alignment length	mismatches	gap opens	q. start	q. end	s. start	29882.2	evalue	bit score
0	MN997409.1	MT020881.1	99.990	29882	3	0	1	29882	1	29882	0.0	55166
1	MN997409.1	MT020880.1	99.990	29882	3	0	1	29882	1	29882	0.0	55166
2	MN997409.1	MN985325.1	99.990	29882	3	0	1	29882	1	29882	0.0	55166
3	MN997409.1	MN975262.1	99.990	29882	3	0	1	29882	1	29882	0.0	55166
4	MN997409.1	LC522974.1	99.993	29878	2	0	4	29881	1	29878	0.0	55164
...	...	...	...	...	...	...	...	...	...	...	...	...
257	MN997409.1	AY283796.1	79.325	1925	357	35	19	1923	3	1906	0.0	1312
258	MN997409.1	AY282752.2	82.304	17716	2948	169	3956	21577	3868	21490	0.0	15175
259	MN997409.1	AY282752.2	80.063	5417	988	68	22539	27910	22414	27783	0.0	3936
260	MN997409.1	AY282752.2	90.189	1641	142	12	28257	29882	28088	29724	0.0	2121
261	MN997409.1	AY282752.2	79.305	1928	358	35	16	1923	1	1907	0.0	1312

262 rows × 12 columns

‘"We have an algorithm which we designed, and we put the DNA sequence into our algorithm and came up with the vaccine in that short amount of time," said Dr. Smith.’ -

https://cbs8.com/article/news/health/coronavirus/coronavirus-vaccine-san-diego/509-e18e37f6-347c-4b08-ad33-910968abb04f

https://towardsdatascience.com/machine-learning-for-biology-how-will-covid-19-mutate-next-4df93cfaf544