**Problem 1)** My name is Joseph Nihill, and I am a Finance and Computer Science dual degree candidate. I am looking to explore Big Data in the future as algorithmic trading interests me the most. My preferred languages of coding are python and R. I hope to gain a better understanding of algorithms from this class, and I think that it will help me in my future by being in a position of making the outline for an algorithm (which we will be doing for this class) instead of implementing the code on a very fast system. Otherwise, than that I enjoy outdoor lead climbing, and spend most of my weekends in Rumney, NH (when it isn't cold).

**Problem 2)**
1)in a set of n numbers there are n! permutations; example is if there are three numbers say 123 there are 3 * 2 * 1 = 6 permutations… 123, 132, 213, 231, 312, 321.
2) nC3, order does not matter in subsets {1,2,3} = {3,2,1}
3) 2^n

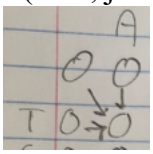
**Problem 3)**
1) a) $A = O(B)$, B dominates
b) $A = \Theta(B)$, A = B for c > 1
c) $A = O(B)$ B dominates
d) $A = \Omega(B)$, A dominates

2) e) $2^{sqrt(lg(n))}$
f) n
d) lg(n!)
a) $n^e$
c) $n^{\pi}$
b) $n^{lg(n)}$

**Problem 4)**

**1)**
You can compare each element and subset in the vegetables DNA to every element and subset using a counter to keep track of similarity to the other elements to find the most similar set of vegetables to the DNA. This would most likely take exponential runtime which is extremely slow and will not work for hundreds of items (or millions for DNA).

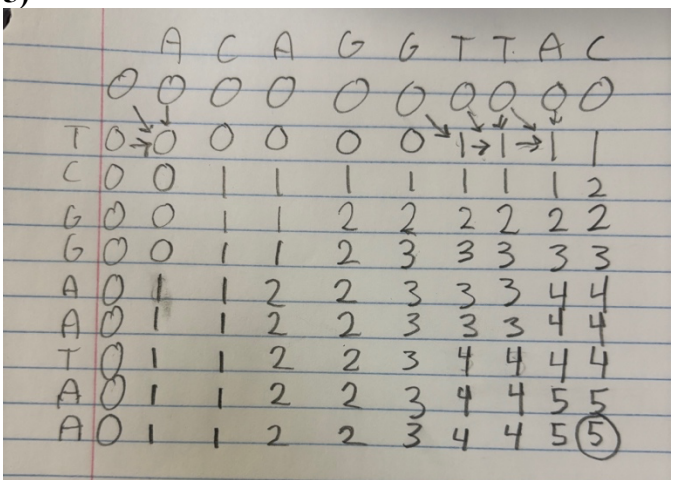
**2) Three subproblems:**
E(i -1, j)
E(i, j – 1)
E(i – 1, j – 1) + Match(i, j)



**3)** The size of the table is M * N entries

**4)** Each table entry represents
E(i, j) = max{E(i -1, j), E(i, j – 1), ( E(i – 1, j – 1) + Match(i, j) ) where Match(i, j) is 0 if the DNA sequence do not match, and 1 if the respective DNA sequences match.

**5)**

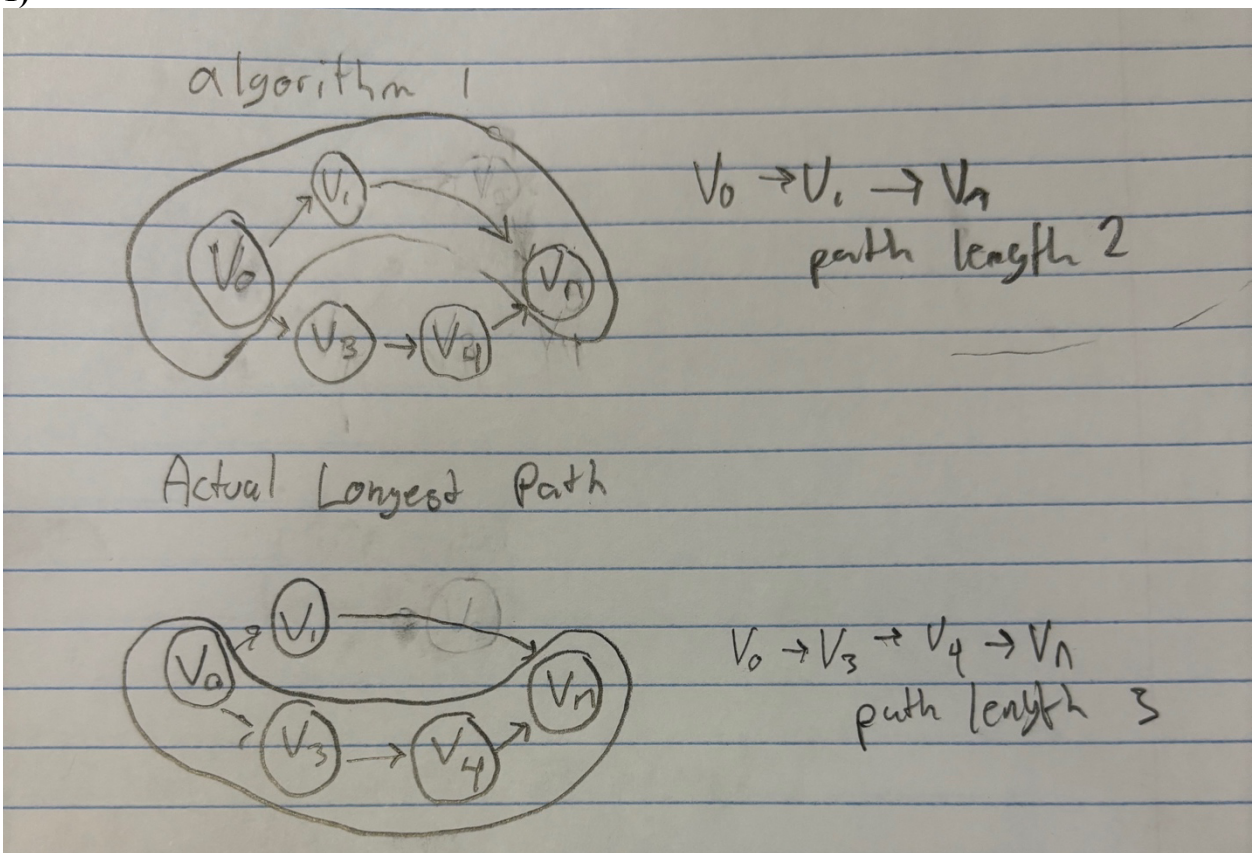



**6)**
**a)**
The algorithm is computing the sequence similarity and the most efficient way to match Human DNA Sequence with Vegetable DNA Sequence. The table specifically is comparing Derek to asparagus and the best resulting pattern is 5 which can be obtained with matching Derek: [A,C,A,G,G,T,T,A,C] to the subset of Asparagus [C,G,G,T,A]. The table uses three values and computes the max of the three of the respective formula E(i, j) = max{E(i -1, j), E(i, j – 1), ( E(i – 1, j – 1) + Match(i, j) ). The bottom right cell will demonstrate the optimal sequence length and then backtracking will allow you to recreate the sequence.

**b & c)** The induction hypothesis holds true for e(i',j') where i' < i and j' < j correctly computes e(i,j). For example: (4,3) in the graph where [A,C,A,**G**,G,T,T,A,C] aligns with [T,C,**G**,G,A,A,T,A,A]. For E(i, j) = max{E(i', j), E(i, j'), ( E(i', j') + Match(i, j) ) where i' = i – 1 and j' = j – 1. E(i', j) = 1, E(i, j') = 1, and E(i', j') + Match(i, j) = 2, the max function results in 2 and then the value is used horizontally for the subsequent sequence for calculating (i + 1, j) or (5,3) as the E(i', j) or (4,3) part of the max function results in 2. This is also used vertically for E(i, j + 1) or (4,4) where 2 is now E(i, j') or (4,3) and the resulting value for E(I,j') in the max function is 2. This recurrence interval can be computed horizontally or vertically but relies on assuming $i_0$ and $j_0$ are zeros as shown on the table and then will start computing values at (i = 1, j = 1) where [**A**,C,A,G,G,T,T,A,C] aligns with [**T**,C,G,G,A,A,T,A,A] resulting in a zero value as shown in the table. The recurrence interval and induction hypothesis hold true but must be computed horizontal or vertically from the top left cell to the bottom right cell. Thus, the induction hypothesis holds and is used throughout the table. The last resulting entry in the table ($i_n$,$j_n$) is the optimal length of the best fitting vegetable DNA sequence to match the human DNA. For real world application where the DNA are much greater in length the induction hypothesis will still hold true and will take M * N runtime using this algorithm instead of an exponential runtime which would never be able to compute in a lifetime.

**Problem 5)**

**1)**




**2)**

```
Def longest_path (v1):
        longest = []
        visited = []
        while path: #while there are still patws that exist
                path = search(v1, visited) #loop finds every path possible
                If (len(longest) < len(visited[-1])): #compare current path to previous paths
                        longest = visited[-1] #stores longest path
        return longest
```