CS 344 **Problem Set 4**
Fall 2022 **Due:** Dec 13, 2022, 11:59 p.m.

Name: (Dong Shu) **NetID**: (ds1657)

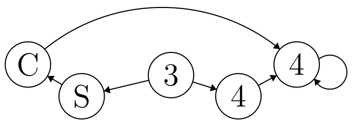
Honor Code: Students may discuss and work on homework problems in groups, which is encouraged. However, each student must write down their solutions independently to show they understand the solution well enough to reconstruct it by themselves. Students should clearly mention the names of the other students who offered discussions. We check all submissions for plagiarism. We take the honor code seriously and expect students to do the same.

Instruction for Submission: This homework has a total of 50 points + 20 bonus points, it will be rescaled to 10 points + 4 bonus points as the eventual score. We encourage you to use LaTex to write your answer, because it's particularly suitable to type equations and it's frequently used for writing academic papers. We have provided the homework3.tex file for you, you can write your answer to each question in this .tex file directly after the corresponding question, and then compile the .tex file into a PDF file for submission. As a quick introduction to LaTex, please see this *Learn LaTeX in 30 minutes* [1]. Compiling a .tex file into a PDF file needs installing the LaTex software on your laptop. It's free and open source. But if you don't want to install the software, you can just use this website `https://www.overleaf.com/`, which is also free of charge. You can just create an empty project in this website and upload the homework1.zip, and then the website will compile the PDF for you. You can also directly edit your answers on the website and instantly compile your file. You can also use Microsoft Word or other software if you don't want to use LaTex. If so, please clearly number your answers so that we know which answer corresponds to which question. You also need to save your answers as a PDF file for submission. Finally, please submit your PDF file only. You should name your PDF file as "Firstname-Lastname-NetID.pdf".

Late Policy: The homework is due on 12/13 (Tuesday) at 11:59pm. We will release the solutions of the homework on Canvas on 12/15 (Thursday) 11:59pm. If your homework is submitted to Canvas before 12/13 11:59pm, there will no late penalty. If you submit to Canvas after 12/13 11:59pm and before 12/15 11:59pm (i.e., before we release the solution), your score will be penalized by 0.9^k, where k is the number of days of late submission. For example, if you submitted on 12/15, and your original score is 80, then your final score will be $80 * 0.9^2 = 64.8$ for $15 - 13 = 2$ days of late submission. If you submit to Canvas after 12/15 11:59pm (i.e., after we release the solution), then you will earn no score for the homework.

Drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code:

[1] `https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes`



You can also draw by hand and insert a picture.

Make best use of picture in Latex: If you think some part of your answer is too difficult to type using Latex, you may write that part on paper and scan it as a picture, and then insert that part into Latex as a picture, and finally Latex will compile the picture into the final PDF output. This will make things easier. For instructions of how to insert pictures in Latex, you may refer to the Latex file of our homework 1, which includes several examples of inserting pictures in Latex.

Discussion Group (People with whom you discussed ideas used in your answers if any):

TianLe Chen

I acknowledge and accept the Honor Code. Please type your initials below:
Signed: (DS)

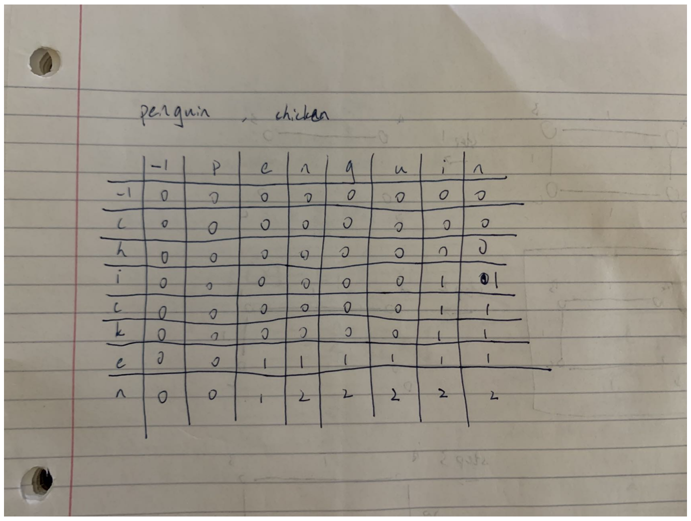
---

1. **Longest Common Subsequence** (20 points)

   Consider the two sequences `penguin` and `chicken`, using the LCS algorithm we learnt in class, find the longest common subsequence of these two words.

   (a) (10 points) Work out the 2-dimensional dynamic programming table $T(i, j)$ and provide the length of the LCS.

   [You are expected to work out every element of the dynamic programming table.]

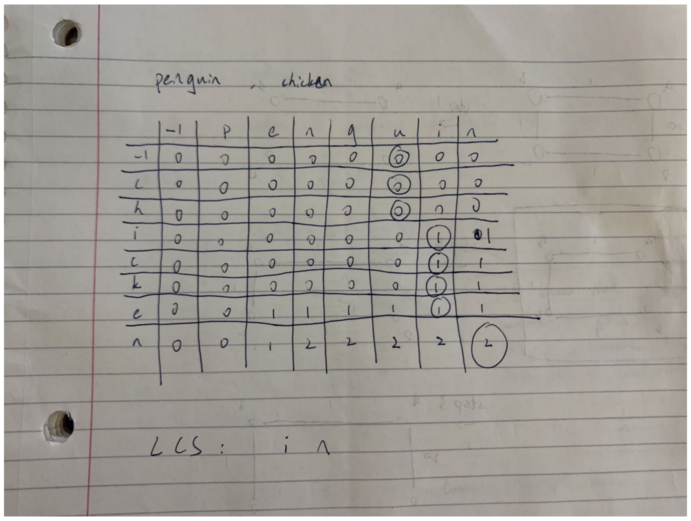


penguin , chicken

	-1	p	e	n	g	u	i	n
-1	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	~~0~~1
c	0	0	0	0	0	0	1	1
k	0	0	0	0	0	0	1	1
e	0	0	1	1	1	1	1	1
n	0	0	1	2	2	2	2	2

(b) (10 points) Traceback on the table to find out the exact longest common subsequence(s) of the two words.

[You are required to draw the traceback path(s) on the table and find out all the LCSs.]



penguin , chicken

	-1	p	e	n	g	u	i	n
-1	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	~~0~~1
c	0	0	0	0	0	0	1	1
k	0	0	0	0	0	0	1	1
e	0	0	1	1	1	1	1	1
n	0	0	1	2	2	2	2	2

LCS : i n

2. **Longest Increasing Subsequence** (30 points)

Let A be an array of length n containing real numbers. A *longest increasing subsequence* (LIS) of A is a sequence $0 \leq i_1 < i_2 < \dots i_\ell < n$ so that $A[i_1] < A[i_2] < \cdots < A[i_\ell]$, so that ℓ is as long as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then an LIS is $i_0 = 2, i_1 = 3, i_2 = 4, i_3 = 6$ corresponding to the subsequence $2, 5, 6, 8$. (Notice that a longest increasing subsequence does not need to be unique). In the following parts, we will walk through the recipe that we saw in class for coming up with DP algorithms to develop an $O(n^2)$-time algorithm for finding an LIS.

(a) (10 points) **(Identify optimal sub-structure and a recursive relationship).** We will come up with the sub-problems and recursive relationship for you, although you will have to justify it. Let $D[i]$ be the length of the longest increasing subsequence of $[A[0], \dots, A[i]]$ that ends on $A[i]$. Expain why

$$D[i] = \max_k \{D[k] + 1 \ : \ 0 \leq k < i, A[k] < A[i]\}.$$

[Provide a short informal explanation. It is good practice to write a formal proof, but this is not required for credit.]

answer:
We know that D[i] means the length of the subsequence. In this case, D[k] will be the maximum length of the subsequence before we discover A[i].
First, we know that if the array is not empty, then D[i] will be greater or equal to 1. If D[i] = 1, then that means there are no smaller A[k] that exist before A[i]. So the D[i] will just be i itself. Moreover, based on the relation, we can express the D[i] in this way. Starting at position 0 of this array all the way to position i, where

$0 \leq k < i$. Standing at position i, we need to find the max$_k$ that $0 \leq k < i$. If we find a max A[k] that is smaller than A[i], then we will add 1 to the D[k], which is the maximum length of the LIS before we move to A[i].
In this way, we know that our LIS will be valid, because we only add 1 to the D[k] when we have the max$_k$ that is before and smaller than A[i] and $0 \leq k < i$. Also, we know that our LIS will be the longest because we keep finding the max$_k$, and using the maximum length of the LIS before position i.

(b) (10 points) **(Develop a DP algorithm to find the value of the optimal solution)** Use the relationship above to design a dynamic programming algorithm that returns the *length* of the longest increasing subsequence. Your algorithm should run in time $O(n^2)$ and should fill in the array D defined above.

[**Provide the pseudo-code, and a brief justification of why the complexity is $O(n^2)$, no formal proof is required.**]

answer:
Since, $0 \leq k < i$, and we can directly tell D[i] if array length is 0 or 1. So in the for loop, i will start at position 1, and k will start at position i-1 all the way to 0. The LIS algorithm will have the complexity of $O(n^2)$, because as we can see we have a nested for loop. Eventually, both outer and inner for loop will traverse the whole array A. Therefore, we will have time complexity of $O(n^2)$

```
LIS(A):
    if(A.length() == 0){
        return EmptyArray
    }
    if(A.length() == 1){
        D[i] == 1
        return D[i]
    }
    for(i = 1, i < A.length(), i++){
        for(k = i-1, k > 0, k--){
            D[i] = max_k{ D[k] + 1: 0 <= k < i, A[k] < A[i] }
            \\We want to store D[i] to i if D[i] have changed,
            \\because D[i] will become D[k] after the next iteration
            if D[i] changed:
                Store D[i] to i
        }
    }
    return D[i]
```

(c) (10 points) **(Adapt the DP algorithm to return the optimal solution)** Adapt your algorithm above by tracking some useful information during the DP procedure, so that it returns the actual LIS, not just its length.

[**Pseudocode and a short explanation.**] answer:
To return the actual LIS, we need to create a parent list for every element. Initially,

the list only has the element itself. Every time when we find a longer D[i], we will put all of the parents of k, which is P[k] into P[i]. In this case, P[i] will have all the LIS elements before i. In the end, we can just return the P[i]

```
LIS'(A):
    Create a parent list for every element.
    if(A.length() == 0){
        return EmptyArray
    }
    if(A.length() == 1){
        D[i] == 1
        return D[i], A[i]
    }
    for(i = 1, i < A.length(), i++){
        for(k = i-1, k > 0, k--){
            D[i] = max_k{ D[k] + 1: 0 <= k < i, A[k] < A[i] }
            \\We want to store D[i] to i if D[i] have changed,
            \\because D[i] will become D[k] after the next iteration
            if D[i] changed:
                Store D[i] to i
                P[i].append_in_front(P[k])
        }
    }
    return D[i], P[i]
```

Note: Actually, there is an $O(n \log n)$-time algorithm to find an LIS, which is faster than the DP solution in this exercise.

3. **Intractable Problems** (20 bonus points)

In our class, we have discussed the P, NP and NPC problems, and we have mentioned that whether P = NP or not is an open problem. We also discussed several NPC problems, including the 3SAT problem, the Hamilton Loop problem, the TSP problem and the 0/1 knapsack problem.

(a) (10 bonus points) We mentioned that if we want to prove P $\neq$ NP, we only need to pick up any one NPC problem and prove that polynomial-time algorithm does not exist for the problem. If you want to prove P $\neq$ NP, select one NPC problem based on your preference and describe your idea of why polynomial-time algorithm does not exist for the problem. It does not have to be a formal proof, a description of your idea would be fine. answer:
We can prove P $\neq$ NP by contradiction.
Let's say P = NP. That means all NP problems can be solved in polynomial time like P problems. In this case, we can come up with a problem x that is part of NP, but not P. We know that NP-complete problems are harder than NP problems, so NP-complete problems should be harder than problem x. We know that NP-complete

problems are not in P, and problem x is not in P. Also, problem x is in NP. So NP cannot be a subset of P, because otherwise NP-complete problem and problem x should also be a subset of P. This contradicts our assumption, so P $\neq$ NP.

(b) (10 bonus points) We mentioned that if we want to prove P = NP, we only need to pick up any one NPC problem and design a polynomial-time algorithm for the problem. If you want to prove P = NP, select one NPC problem based on your preference and describe your idea of a polynomial-time algorithm that solves the problem. It does not have to be a formal algorithm or pseudo-code, a description of your idea of designing such an algorithm would be fine.

answer:

My idea is to design a randomized algorithm that sacrifices correctness for efficiency. Moreover, we can use some ideas that we just learned about Dimensionality Reduction. For example, for the 3SAT problem, the dimension for one set is 3, and the dimension for n sets is 9, which is n^2. Maybe we can find the pattern and reduce the dimension to less than n. We can also express 3SAT into the perceptron with non-linear activation function. So that we can solve it as a Logical Equations. Then, use the randomized algorithm to achieve polynomial time.