Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 LP Basics

Linear Program. A *linear program* is an optimization problem that seeks the optimal assignment for a linear objective over linear constraints. Let $x \in \mathbb{R}^d$ be the set of variables and $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$. The canonical form of a linear program is

$$\begin{aligned} &\text{minimize } c^\top x \\ &\text{subject to } Ax \geq b \\ &x \geq 0 \end{aligned}$$

Any linear program can be written in canonical form.

Let's check this is the case:

(i) What if the objective is maximization?

(ii) What if you have a constraint $Ax \leq b$?

(iii) What about $Ax = b$?

(iv) What if the constraint is $x \leq 0$?

(v) What about unconstrainted variables $x \in \mathbb{R}$?

Solution:

(i) Take the negative of the objective.

(ii) Negate both sides of the inequality.

(iii) Write both $Ax \leq$ and $Ax \geq b$ into the constraint set.

(iv) Change of variable: replace every x by $-z$, and add constraint $z \geq 0$.

(v) Replace every x by $x^+ - x^-$, add constraints $x^+, x^- \geq 0$. Note that for every solution to the original LP, there is a solution to the transformed LP (with the same objective value). Similarly, if there is a feasible solution for the transformed problem, then there is a feasible solution for the original problem with the same objective value.

Dual. The dual of the canonical LP is

$$\begin{aligned} &\text{maximize } b^\top y \\ &\text{subject to } A^T y \leq c \\ &y \geq 0 \end{aligned}$$

Weak duality: The objective value of any feasible dual $\leq$ objective value of any feasible primal
Strong duality: The *optimal* objective values of these two are equal.
Both are solvable in polynomial time by the Ellipsoid or Interior Point Method.

2 Job Assignment

There are I people available to work J jobs. The value of person i working 1 day at job j is a_{ij} for $i = 1, \dots, I$ and $j = 1, \dots, J$. Each job is completed after the sum of the time of all workers spend on it add up to be 1 day, though partial completion still has value (i.e. person i working c portion of a day on job j is worth $a_{ij}c$). The problem is to find an optimal assignment of jobs for each person for one day such that the total value created by everyone working is optimized. No additional value comes from working on a job after it has been completed.

(a) What variables should we optimize over? I.e. in the canonical linear programming definition, what is x?

3.5cm

Solution: An assignment x is a choice of numbers x_{ij} where x_{ij} is the portion of person i's time spent on job j.

(b) What are the constraints we need to consider? Hint: there are three major types.

3.5cm

Solution: First, no person i can work more than 1 day's worth of time.

$$\sum_{j=1}^{J} x_{ij} \le 1 \qquad \text{for } i = 1, \dots, I.$$

Second, no job j can be worked past completion:

$$\sum_{i=1}^{I} x_{ij} \le 1 \qquad \text{for } j = 1, \dots, J.$$

Third, we require positivity.

$$x_{ij} \ge 0 \qquad \text{for } i = 1, \dots, I, j = 1, \dots, J.$$

(c) What is the maximization function we are seeking?

3.5cm

Solution: By person i working job j for x_{ij}, they contribute value $a_{ij}x_{ij}$. Therefore, the net value is

$$\sum_{i=1,j=1}^{I,J} a_{ij}x_{ij} = A \bullet x.$$

3 Linear regression

In this problem, we show that linear programming can handle linear regression. Let $A \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}^d$ be given, where n, d are not assumed to be constant. However, assume all input numbers have constant bits.

(a) Recall that the ℓ_1 norm of a vector v is given by $\|v\|_1 = \sum_{i=1}^{d} |v_i|$. The L1 regression problem asks you to find $x \in \mathbb{R}^d$ that minimizes $\|Ax - b\|_1$.

(i) Provide a linear program that finds the opimtal x, given A, b.

(ii) Argue that it can be solved in polynomial time (in n, d).

Solution: (i). Let a_i be the ith row of A. The LP is given by

$$\begin{aligned} \text{minimize } & \sum_i t_i \\ \text{subject to } & a_i \cdot x - b_i \le t_i \\ & a_i \cdot x - b_i \ge -t_i \\ & t_i \ge 0 \end{aligned}$$

(ii). The LP has $d+n$ variables and $3n$ constraints. Assuming all input numbers have constant bit complexity, the LP can be solved in polynomial time (via Ellipsoid or inteiror point method).

(b) Recall that the ℓ_∞ norm of a vector v is given by $\|v\|_\infty = \max_i |v_i|$. The L_∞ regression problem asks you to find $x \in \mathbb{R}^d$ that minimizes $\|Ax - b\|_\infty$.

(i) Provide a linear program that finds the opimtal x, given A, b.

(ii) Argue that it can be solved in polynomial time (in n, d).

Solution: (i).

$$\begin{aligned} \text{minimize } & t \\ \text{subject to } & a_i \cdot x - b_i \le t \\ & a_i \cdot x - b_i \ge -t \\ & t \ge 0 \end{aligned}$$

(ii). Similar to (a)(ii).

4 Power Outage

Suppose Alice is writing up her 170 homework when the power goes out. She continues writing her homework after the power goes out in attempts to finish on time. However, since the power is out, she continues to write out her homework while making some mistakes and finishes her homework. She doesn't remember when the power goes out and so now she has to check her work and change it to be correct. It takes her 1 second to insert a character into the homework, 2 seconds to erase a character, and 1.5 seconds to change one character to another. Compute how long it will take Alice to fix her homework if she optimally chooses her edits, given the string x with m characters is what she has written and the string y with n characters is what she intended to write.

Solution:

This problem is basically edit distance except instead of counting how many edits it takes to change x to y, we are counting how much time it takes Alice to change x to y.

So the subproblem is:

S(i, j) = the minimum amount of time it takes for Alice to edit the first i characters of x to the the first j characters of y.

From this we can write the recurrence relation:

$$S(i,j) = min \begin{cases} S(i,j-1)+1 & \text{insertion,} \\ S(i-1,j)+2 & \text{deletion,} \\ S(i-1,j-1)+1.5 & \text{if (x[i] != y[i]),} \\ S(i-1,j-1) & \text{if (x[i] == y[i])} \end{cases}$$

We know that the $S(i, j-1)$ corresponds to an insertion because we are inserting y_j to x so we want to match the first i characters of x to the first $j-1$ characters in y because the inserted character y_j will definitely match y_j. Similarly, $S(i-1, j)$ is deleting x_i and matching the first $i-1$ characters in x to the first j characters in y. The base case is $S(0,0) = 0$, for all $j \in [1,n]$ $S(0,j) = j$, and for all $i \in [1,m]$ $S(i,0) = i$.

The runtime for this problem is the same for edit distance $O(mn)$; we have $m * n$ subproblems which take O(1) time to compute.

5 Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB.

Your task is to, given a bit string s, determine how many ways one can interpret s. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time. Your algorithm should run in time at most $O(nm\ell)$ where n is the length of the input bit string s, m is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols.

Please give a 3-part solution.

Solution:

Main Idea: We define our subproblems as follows: let $A[i]$ be the number of ways of interpreting the string $s[: i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])].$$

Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula in a trivial manner.

Pseudocode:

```
procedure TRANSLATE($s$):
    Create an array $A$ of length $n + 1$ and initialize all entries with zeros.
    Let $A[0] = 1$
    for $i := 1$ to $n$ do
        for each symbol $a$ in $d$ do
            if $i \geq \text{length}(d[a])$ and $d[a] = s[i - \text{length}(d[a]) + 1 : i]$ then
                $A[i] += A[i - \text{length}(d[a])]$
    return $A[n]$
```

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, there is only one way to interpret $s[: 0]$ (the empty string). Hence, $A[0] = 1$.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[: i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol a_k. For $a_k = a$, if the suffix of $s[: i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_k$ has a one-to-one correspondence with an interpretation $a_1 \ldots a_{k-1}$ of $s[: i - \text{length}(d[a])]$. From our inductive hypothesis, there are exactly $A[i - \text{length}(d[a])]$ of the latter. On the other hand, if the suffix of $s[: i]$ differs from $d[a]$, then there is no interpretation of $s[: i]$ ending with symbol a. Summing this up over all symbols a's implies that our recurrence relation yields the right value for $A[i]$.

Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, the number of ways to interpret s.

Runtime Analysis: There are n iterations of the outer for loop and m iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nm\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n+m)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit for the problem.