Practice Problems for the Final Exam

- The following set of practice problems has been compiled from previous semesters to aid you in preparing for the final exam.
- These problems should not be taken as a strict gauge for the difficulty level of the actual exam.
- Where possible we have included the "points" value of a problem, indicating roughly how much time (in minutes) you should spend on the problem (in minutes).
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to *give an algorithm*, you may describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- If you think you have found a bug/errata, please let us know via Piazza.
- ***We encourage you to go through these problems without first looking at the solutions!***

Problem-1: T/F Questions. [3 points each] Answer *true* or *false* to each of the following. *No justification is required for your answer.*

(a) [T/F] Let n and k be integers greater than 10. Consider the following family $\mathcal{H}$ of hash functions that map elements from the set $\{1,\ldots,n\}^k$ to the set $\{1,\ldots,n\}$. $\mathcal{H}$ contains for every $1 \leq i \leq k$ a function h_i that maps $(x_1,\ldots,x_k) \in \{1,\ldots,n\}^k$ to x_i. $\mathcal{H}$ is a universal hash family.

Solution: False. Two entries from $\{1,\ldots,n\}^k$ that agree on all coordinates but one (e.g. $(1,1,1,\ldots 1)$ and $(2,1,1,\ldots,1)$) are mapped to the same element with probability $1-1/k > 1/n$.

(b) [T/F] Consider the following algorithm for approximating the weight of the minimum spanning tree. Given a weighted graph $G=(V,E)$ as input, pick the first vertex, $s \in V$, from the input, and compute the tree of shortest paths between s and all $v \in V$. Output the sum of the weights of all the edges in the tree. This algorithm is a 2-approximation for the minimum spanning tree problem.

Solution: False. Consider a graph that connects a node s to D other vertices with weight 100, and connects the D vertices between themselves in weight 0 edges. Then the MST has a weight of 100, while the weight of the tree the algorithm returns is $100D$.

(c) [T/F] Suppose that there is a polynomial time reduction from the general vertex cover problem to bipartite matching. Then, there is a polynomial time algorithm that solves SAT.

Solution: True. Vertex cover is NP-hard, so SAT can be reduced to vertex cover. Thus, if we can reduce vertex cover to bipartite matching, we can reduce SAT to bipartite matching. Since bipartite matching can be solved in polynomial time, this gives a polynomial SAT solver.

(d) [T/F] A randomized algorithm that is always correct and has an expected running time that is linear may be converted into a randomized algorithm that always runs in linear time but is correct only with probability 99%.

Solution: True. Suppose that the expected running time of the algorithm is at most cn for some constant $c \geq 0$. We create a new algorithm by running the original algorithm til time $100cn$. If the original algorithm stops by that time, we output the answer from the algorithm. Otherwise, we output some value. From Markov inequality, the original algorithm stops before time $100cn$ with probability at least 0.99, which means that the new algorithm outputs a correct answer with probability at least 99%.

(e) [T/F] For every NP-hard minimization problem there is a constant $c > 1$, such that approximating the problem to within a factor c is NP-hard.

Solution: False. Some problems, like $PARTITION$ from recitation, can be approximated within $(1 + \epsilon)$ for all $\epsilon > 0$.

(f) [T/F] Finding a maximum clique in a graph with n vertices can be formulated as a linear program whose size is $O(n^3)$.

Solution: False. Max-Clique is NP-hard, while real linear programs are easy to solve.

(g) [T/F] To find the maximum flow in any flow network G with unit capacities, we can perform the following procedure: until there are no more augmenting paths, repeatedly find an augmenting path from the source to the sink, and delete the edges in the path.

Solution: False. Consider the graph with edges $s \to 1$, $s \to 2$, $1 \to 2$, $2 \to t$, $1 \to 2$. The maximum flow is 2, but removing the path $s \to 1 \to 2 \to t$ disconnects s from t.

(h) [T/F] Given a maximization linear program LP1 and its dual LP2, the objective value of every feasible solution to LP1 is not larger than the objective value of any feasible solution to LP2.

Solution: True. By LP duality.

(i) [T/F] We consider an implementation of the ACCESS operation for self-organizing lists which, whenever it accesses an element x of the list, transposes x with its predecessor in the list (if it exists). This heuristic is 2-competitive.

Solution: False. Suppose that we have a list of length n in which the last two elements are x and y, with y coming later. Then, consider an access sequence of length n: $y, x, y, x, \ldots, y, x$. It will have cost $O(n^2)$. Meanwhile, if we consider the move-to-front heuristic, both x and y would be moved to the beginning of the list after the first two accesses, and each subsequent access would have cost $O(1)$, for a total cost of $O(n)$. Because move-to-front outperforms this heuristic by a factor of $O(n)$, this heuristic cannot be 2-competitive.

(j) [T/F] Negating all the edge weights in a weighted undirected graph G and then finding the minimum spanning tree gives us the *maximum*-weight spanning tree of the original graph G.

Solution: True.

(k) **[T/F]** Suppose we are given an array A of distinct elements, and we want to find $n/2$ elements in the array whose median is also the median of A. Any algorithm that does this must take $\Omega(n \log n)$ time.

Solution: False. It's possible to do this in linear time using SELECT: first find the median of A in $\Theta(n)$ time, and then partition A around its median. Then we can take $n/4$ elements from either side to get a total of $n/2$ elements in A whose median is also the median of A.

(l) **[T/F]** In a weighted connected graph $G = (V, E)$, each minimum weight edge belongs to some minimum spanning tree of G.

Solution: True. Consider a MST T. If it doesn't contain the minimum edge e, then by adding e to T, we get a cycle. By removing a different edge than e from the cycle, we get a spanning tree T' whose total weight is no more than the weight of T . Thus, T' is a MST that contains e.

(m) **[T/F]** Let A and B be two decision problems. If A is polynomial time reducible to B, and $B \in$ NP, then $A \in$ NP.

Solution: True. If A reduces to B, then this means that B can be used to solve A via a polynomial time reduction. Therefore, if $B \in NP$, then $A \in NP$.

(n) **[T/F]** Given a set of points $\{x_1, \ldots, x_n\}$, and a degree n bounded polynomial:

$$P(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

we can evaluate $P(x)$ at $\{x_1, \ldots, x_n\}$ in $O(n \log n)$ time using the FFT algorithm presented in class.

Solution: False. Using the FFT, we can efficiently evaluate a polynomial at the roots of unity, but not any n points.

(o) **[T/F]** The gradient descent algorithm applied to the function:

$$f(x) = (x+1)^2(x-1)(x-2) = x^4 - x^3 - 3x^2 + x + 2$$

is guaranteed to converge to a global minimum of that function.

Solution: False. To see this, note that f has at least one local maximum (a critical point between $x = -1$ and $x = 1$), and if the gradient descent algorithm starts at this local maximum then it will stay there, thus not converging to any global minimum.

(p) **[T/F]** Every function that has a single global minimum is convex.

Solution: False. A function can have multiple minima, only one of which is global. In this case it has a single global minimum but is not convex.

(q) **[T/F]** Consider the function $f(x) = \frac{x^3}{3}$. The gradient descent algorithm, initialized at some non-zero value (i.e. $x^{(0)} \neq 0$), and run with the following iteration scheme:

$$x^{(t+1)} \longleftarrow x^{(t)} - \frac{1}{2} f'(x^{(t)})$$

always converges to $-\infty$.

Solution: False. The algorithm may or may not converge to 0 depending on the value of $x^{(0)}$. For example, if $x^{(0)} = 1$, then

$$x^{(t+1)} = x^{(t)} - \frac{1}{2}(x^{(t)})^2$$

for all $t > 0$. By induction, it can be seen that $0 < x^{(t)} \leq 1$ for all $t > 0$ and hence the algorithm will not converge to $-\infty$.

Problem-2: Short Answer Questions [8 points each]

(a) Consider the *CLIQUE* problem: Given an undirected graph $G = (V, E)$ and a positive integer k, is there a subset C of V of size at least k such that every pair of vertices in C has an edge between them?

Ben Bitdiddle thinks he can solve the clique problem in polynomial time using linear programming as follows:

- Let each variable in the linear program represent whether or not each vertex is a part of our clique. Add constraints stating that each of these variables must be nonnegative and at most one.
- Go through the graph G and consider each pair of vertices. For every pair of vertices where there is not an edge in G, add a constraint stating that the sum of the variables corresponding to the endpoint vertices must be at most one. This ensures that both of them cannot be part of a clique if there is no edge between them.
- The objective function is the sum of the variables corresponding to the vertices. We wish to maximize this function.

Ben argues that the value of the optimum must be the size of the maximum-size clique in G, and we can then simply compare this value to k. *Explain the flaw in Ben's logic.*

Solution: This is an integer program, not a linear program, and therefore we don't know how to solve it in polynomial time. (Alternatively, if we don't add the integrality constraints, we can solve it in polynomial time but will likely get fractional values back; it's unclear what fractional values of the variables mean with regard to the clique.)

Problem-3: Who is missing? [15 points]

In this problem, your input is a stream of m integers, all of which are from the set $\{1, \ldots, n\}$.

(a) Suppose $m = n - 1$. This means that there is at least one element from $\{1, \ldots, n\}$ that does not appear in the stream. Suppose you are also guaranteed that there is *exactly* one element that does not appear (so all other elements appear exactly once). Give a deterministic streaming algorithm to find this element with one pass using only $O(\log n)$ space.

Solution: Keep track of the number of elements in the stream, m, as well as the sum of the elements seen so far, $\sum_i A[i]$. Then, at the end, compute the missing element, x, as:

$$x = \sum_{i=1}^{n} i - \sum_i A[i] = \frac{n(n+1)}{2} - \sum_i A[i]$$

where n is computed as $m + 1$. Since the elements in the stream are restricted to the range $[1, n]$, the running sum can be stored in $O(\log n)$ space.

(b) Suppose $m = n-2$. Then at least two elements do not appear in the stream. Again you are guaranteed that exactly two elements do not appear (so all other elements appear exactly once). Give a deterministic streaming algorithm to find both missing elements with one pass using $O(\log n)$ space.

Solution: Keep track of $\max A[i]$, $\sum_i A[i]$ and $\sum_i A[i]^2$. Then, we may retrieve the missing numbers (call them x and y) from the following system of equations:

$$\begin{aligned} n &= \max A[i] \\ x + y &= \sum_{i=1}^{n} i - \sum_i A[i] = \frac{n(n+1)}{2} - \sum_i A[i] \\ x^2 + y^2 &= \sum_{i=1}^{n} i^2 - \sum_i A[i]^2 = \frac{n(n+1)(2n+1)}{6} - \sum_i A[i]^2 \end{aligned}$$

Since we can determine the values of $x + y$ and $x^2 + y^2$, we can solve for x and y.

Problem-4: Set Cover [30 points]

Recall the SET-COVER problem: Given a set U of size n and subsets $S_1, \ldots S_m$ of U such that $\bigcup_{i=1}^{m} S_i = U$, find a set of indices $I \subseteq \{1, \ldots, m\}$ with minimum cardinality ($|I|$) such that $\{S_i\}_{i \in I}$ covers U. I.e. $\bigcup_{i \in I} S_i = U$.

Consider the following $(\ln n + 1)$-approximation algorithm for SET-COVER: if there exists a set of indices $J \subseteq \{1, \ldots, m\}$ of size $|J| = k$ such that $\{S_j\}_{j \in J}$ covers U, then the algorithm outputs a set of indices $I \subseteq \{1, \ldots, m\}$ such that $\{S_i\}_{i \in I}$ covers U and $|I| \leq k(\ln n + 1)$. The algorithm is shown below.

```
1: I = ∅
2: while U is not empty do
3:     Pick the largest subset S_i
4:     I = I ∪ {i}
5:     Remove all elements of S_i from U and from the other subsets
6: end while
7: return I
```

In this problem, you may assume a stronger assumption: that there exists a set of indices $J \subseteq \{1, \ldots, m\}$ of size $|J| = k$ such that $\{S_j\}_{j \in J}$ covers U "100 times" - i.e. for each element $u \in U$, there are at least 100 elements j's of J such that $u \in S_j$.

Given this assumption, prove that the algorithm given in class outputs a set of indices $I \subseteq \{1, \ldots, m\}$ such that $\{S_i\}_{i \in I}$ covers U and $|I| \leq \frac{k \ln n}{100} + 1$ by following those two steps:

(a) Let U_l be the value of U after the l-th iteration of the algorithm. (Note that $U_0 = U$) Prove that there exists $j' \in J$ such that $|S_{j'} \cap U_l| \geq \frac{100}{k}|U_l|$.

Solution: Let U_l be the value of U after the l-th iteration of the algorithm. We will prove that there exists $j' \in J$ such that $|S_{j'} \cap U_l| \geq \frac{100}{k}|U_l|$. To see this, consider that $\{S_j\}_{j \in J}$ covers U 100 times, which implies that $\{S_j\}_{j \in J}$ also covers U_l 100 times. Then:

$$\begin{aligned}\sum_{j \in J} |S_j \cap U_l| &= \sum_{u \in U_l} |\{j \in J \mid u \in S_j\}| \\ &\geq \sum_{u \in U_l} 100 \\ &= 100|U_l|.\end{aligned}$$

Let j' be the element of J with maximal $|S_{j'} \cap U_l|$. Then from the inequality above, we may conclude that $|S_{j'} \cap U_l| \geq \frac{100}{k}|U_l|$. □

(b) Now show that the algorithm described above outputs I of size at most $\frac{k \ln n}{100} + 1$.
Hint: the following inequality might be useful: $1 - \alpha \leq e^{-\alpha}$ for all $\alpha > 0$.

Solution: For the second part of the proof, we use the above fact to prove the claim. Consider i' we picked at the $(l+1)$-th iteration. From , there exists $j' \in J$ such that

$$|S_{j'} \cap U_l| \geq \frac{100}{k}|U_l|$$

Since we pick i' to maximize $|S_{i'} \cap U_l|$, we know that:

$$|S_{i'} \cap U_l| \geq |S_{j'} \cap U_l| \geq \frac{100}{k}|U_l|$$

Hence, we can conclude that $|U_{l+1}| \leq (1 - 100/k)|U_l|$ and therefore:

$$|U_l| \leq (1 - 100/k)^l |U_0|$$

Since $(1 - 100/k) \leq e^{-100/k}$, we have $|U_l| \leq e^{-100l/k}|U_0|$ and $|U_{k \ln n/100+1}| < 1$.

Thus, $|U_{k \ln n/100+1}| = 0$ and our algorithm ends after at most $k \ln n/100 + 1$ iterations. In other words, the output set I has at most $k \ln n/100 + 1$ elements, which concludes our proof. □

Problem-5: Fruitloaf Factory [15 points]

Food Inc has been told by the FDA that they must list more information on the nutrition label, and that customers will refuse to buy the product if it has too much fat or sugar. Some important details about FruitLoaf:

- FruitLoaf is composed entirely of n ingredients: $i_1, i_2, \ldots, i_n$.
- The *original* FruitLoaf had exactly 1 gram of each ingredient i_j.
- The *new* FruitLoaf should be exactly w grams, consisting of g_j grams of ingredient j.
- Customers will not buy a FruitLoaf unless the total amount of sugar is less or equal to S and the total amount of fat is less than or equal to F.
- Each ingredient i_j has f_j grams of fat per gram of ingredient, and s_j grams of sugar per gram of ingredient.

Customers like the old FruitLoaf flavor (without knowing what it contains), so you're hoping to stay as close to the old recipe as possible. Customers' taste of the difference is the Manhattan distance between the two recipes. In other words, the distance is: $\sum_{j=1}^{n} |1 - g_j|$.

Given the values of w, S, and F as input, *write a linear program in standard form to minimize the difference between the old FruitLoaf flavor profile and the new one subject to the constraints on the weight, fat content and sugar content.* Note that your linear program needs to be in standard form. That is, if it has variables $\{x_1, x_2, \ldots, x_n\}$, it must be written in this form:

$$
\begin{aligned}
\max \quad & c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \\
\text{subject to} \quad & a_{(1,1)} x_1 + a_{(1,2)} x_2 + \ldots + a_{(1,n)} x_n \leq b_1 \\
& a_{(2,1)} x_1 + a_{(2,2)} x_2 + \ldots + a_{(2,n)} x_n \leq b_2 \\
& \ldots \\
& a_{(m,1)} x_1 + a_{(m,2)} x_2 + \ldots + a_{(m,n)} x_n \leq b_m \\
& x_j \geq 0 \text{ for all } 1 \leq j \leq n
\end{aligned}
$$

Solution:

$$\max -\sum_{j=1}^{n} a_i \tag{1}$$

$$\text{subject to} \quad -a_j - g_j \leq -1 \qquad \forall 1 \leq j \leq n \tag{2}$$

$$-a_j + g_j \leq 1 \qquad \forall 1 \leq j \leq n \tag{3}$$

$$\sum_{j=1}^{n} g_j f_j \leq F \tag{4}$$

$$\sum_{j=1}^{n} g_j s_j \leq S \tag{5}$$

$$\sum_{j=1}^{n} g_j \leq w \tag{6}$$

$$-\sum_{j=1}^{n} g_j \leq -w \tag{7}$$

$$g_j \geq 0 \forall j \in [1, n] \tag{8}$$

$$a_j \geq 0 \forall j \in [1, n] \tag{9}$$

- (6) is the condition.
- (7) and (8) set $a_j \geq |1 - g_j|$.
- (9)(10) set the fat and sugar restrictions.
- (11)(12) set the weight to be w.
- (13)(14) set the variables used to be greater than 0.

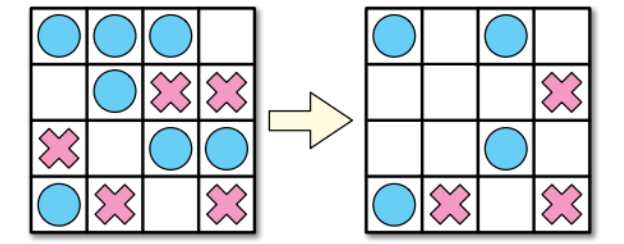
Problem-6: Board Solitaire [20 points]

Consider the following puzzle. The puzzle consists of an $m \times n$ grid of squares, where each square is either empty or occupied by a red or blue stone. The goal of the puzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

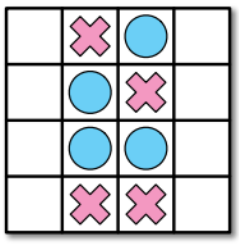
(a) Every row contains at least one stone,

(b) No column contains stones of both colors.

Note that for some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

Let $SOLITAIRE$ denote the decision problem of determining, given an initial configuration of red and blue stones, whether the puzzle can be solved. Show that $SOLITAIRE$ is NP-complete. *You may only use a reduction from* $3SAT$ *to* $SOLITAIRE$.

Solution: We give a reduction from $3SAT$ to $SOLITAIRE$. Given a instance of 3SAT, let n be the number of clauses, and m be the number of different variables. We will construct an $n \times m$ board as follows. For each variable in the 3SAT instance, mark out a distinct column on the board. Then for each clause:

- Create a row for it in the game, where each column in that row corresponds to a literal.
- In each row, a blue stone is placed if the literal is a variable, or a red stone if it is its negation.
- For the variables that are not included in the clause, we simply leave the corresponding squares to be empty.

For example, given a formula:

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

we can create an exact game setup as shown in the problem statement, where a, b, c, and d denote column 1, 2, 3 and 4 from let to right respectively.

Correctness: The conjunctive normal form of the formula ensures the first condition of the game to be satisifed, because a clause is satisfied if and only if at least one literal is

set to true, which is equivalent to at least one stone is in the corresponding row. Since contradictory literals are placed in the same column, the assignment is consistent, which means no conflicting stones will be in the same column, and satisfies the second condition of the game. Hence, the boolean formula is satisfiable if and only if the puzzle is solvable. Thus, the reduction is correct.

Complexity: We only need linear time to perform this reduction, since we can allocate a column to a new variable as we go. Hence the reduction is polynomial-time and thus $SOLITAIRE$ is NP-hard.

$SOLITAIRE$ is in NP because a valid configuration of stones can be checked in linear time by counting the number of stones in each row, and the types of stones that appear in each column. Hence, $SOLITAIRE$ is NP-complete. □

Problem-7: Forgetful Forrest [15 points]

Prof. Forrest Gump is very forgetful, so he uses automatic calendar reminders for his appointments. For each reminder he receives for an event, he has a 50% chance of actually remembering the event (decided by an independent coin flip).

(a) Suppose we send Forrest k reminders for each of n events. What is the expected number of appointments Forrest will remember? Give your answer in terms of k and n.

Solution: These are all independent events. So linearity of expectation applies. Each given event has been remembered with probability $1-2^{-k}$. So in expectation $n(1-2^{-k})$ appointments are remembered.

(b) Suppose we send Forrest k reminders for a *single* event. How should we set k with respect to n so that Forrest will remember the event with high probability, i.e., $1-1/n^{\alpha}$?

Solution: This problem is equivalent to how many times we must flip a coin to get a head with high probability. The probability of k tails in a row is $1/2^k$. Thus exactly $\alpha \lg n$ coin flips suffice.

(c) Suppose we send Forrest k reminders for each of n events. How should we set k with respect to n so that Forrest will remember *all* the events with high probability, i.e., $1 - 1/n^{\alpha}$?

Solution: We must send at least $k = \Omega(\lg n)$ reminders, because we needed this many reminders to remember one event with high probability. If we send $k = (\alpha + 1) \lg n$ reminders, then each event is remembered with probability $1 - 1/n^{\alpha+1}$. Byd union bound, we know that all events are remembered with probability $1 - 1/n^{\alpha}$. So, the number of reminders needed is $k = O(\lg n)$.

Problem-8: Piano Recital [15 points]

Prof. Chopin has a piano recital coming up, and in preparation, he wants to learn as many pieces as possible. There are m possible pieces he could learn. Each piece i takes p_i hours to learn.

Prof. Chopin has a total of T hours that he can study by himself (before getting bored). In addition, he has n piano teachers. Each teacher j will spend up to t_j hours teaching. The teachers are very strict, so they will teach Prof. Chopin only a single piece, and only if no other teacher is teaching him that piece.

Thus, to learn piece i, Prof. Chopin can either (1) learn it by himself by spending p_i of his T self-learning budget; or (2) he can choose a unique teacher j (not chosen for any other piece), learn together for $\min\{p_i, t_j\}$ hours, and if any hours remain ($p_i > t_j$), learn the rest using $p_i - t_j$ hours of his T self-learning budget. (Learning part of a piece is useless.)

(a) Assume that Prof. Chopin decides to learn exactly k pieces. Prove that he needs to consider only the k lowest p_is and the k highest t_js.

Solution: Assume there exists a selection of teachers and pieces for learning k pieces. Let the set of lowest k pieces be P_k. If there is a piece in our selection that is $\notin P_k$, then we must have a piece in P_k not in the final selection. If we swap the one with the higher cost ($\notin P_k$) with the one with lower cost ($\in P_k$), the new selection thus made will still be valid, because if the higher time cost was fulfilled in the previous selection, the lower time cost in the new selection will still be fulfilled. In this way, we can swap pieces until all of them are $\in P_k$.

Similarly, we can swap the teachers for those of higher value until they are the ones with the k highest times.

(b) Assuming part (a), give an efficient greedy algorithm to determine whether Prof. Chopin can learn exactly k pieces. Argue its correctness.

Solution: Let us sort all the teachers and pieces in increasing order beforehand. Call the sorted lists P and T. We see that if a solution exists, there is also one in which P_1 is paired with T_{n-k+1}, P_2 is paired with T_{n-k+2} and so on.

So for each $1 \leq i \leq k$, the greedy algorithm checks if $P_i \leq T_{n-k+i}$.

- If it is, then we don't need to use the shared time for this piece.
- If it is not, we need to use $T_{n-k+i} - P_i$ of the shared time.

We can add up these values. In the end, if the total shared time we need is $> T$, we return false. Otherwise, we return true. This takes $O(k)$ time, apart from the initial sorting.

(c) Using part (b) as a black box, give an efficient algorithm that finds the maximum number of pieces Prof. Chopin can learn. Analyze its running time.

Solution: Notice that if k_{max} is the maximum value of pieces we can learn, we can also learn k pieces for any $k \leq k_{max}$. This suggests that we binary search over the value of k. We try $O(\log n)$ values during the binary search, and checking each value takes $O(n)$ time. This takes $O(n \log n)$ time. The sorting also took $O(n \log n)$ time, so the algorithm takes $O(n \log n)$ time overall.

Problem-9: Startups are Hard [20 points]

For your new startup company, *Uber for Algorithms*, you are trying to assign projects to employees. You have a set P of n projects and a set E of m employees. Each employee e can only work on one project, and each project $p \in P$ has a subset $E_p \subseteq E$ of employees that must be assigned to p to complete p. The decision problem we want to solve is whether we can assign the employees to projects such that we can complete (at least) k projects.

(a) Give a straightforward algorithm that checks whether any subset of k projects can be completed to solve the decisional problem. Analyze its time complexity in terms of m, n, and k.

Solution: For each $\binom{n}{k}$ subsets of k projects, check whether any employee is required by more than one project by going through each of the k projects p and marking the employees in E_p as needed; if no employee is marked twice, then the subset of projects may be completed. Output "yes" if any subset of k project can be completed, and "no" otherwise.
Time Complexity: $\binom{n}{k} \cdot m$ because there are $\binom{n}{k}$ subsets of size k and we pay $O(m)$ time per subset (because all but one employee will be marked only once). Asymptotically, this is $(\frac{n}{k})^k m$.

(b) Is your algorithm in part (a) fixed-parameter tractable (FPT)? Explain briefly.

Solution: No; an FPT algorithm requires a time complexity of $n^{O(1)} f(k)$, whereas in our running time the exponent on n increases with k.

(c) Recall the *3D-MATCHING Problem*: You are given three sets X, Y, Z, each of size m; a set $T \subseteq X \times Y \times Z$ of triples; and an integer k. Determine whether or not there is a subset $S \subseteq T$ of (at least) k disjoint triples.

Show that the problem is NP-hard via a reduction from *3D-MATCHING* matching.

Solution: Each $(x, y, z) \in T$ becomes a project that requires employees $E_{(x,y,z)} = \{e_x, e_y, e_z\}$. Thus $n = |T|$, $E = X \cup Y \cup Z$, and $m = |X| + |Y| + |Z|$. We set k to be the same in both problems. The size of the matching is equal to the number of projects that can be completed because both problems model disjointness: if k projects can be completed, a subset S of size k can be found, and vice-versa. The reduction takes polynomial time.

Problem-10: Load Balancing [15 points]

Suppose you need to complete n jobs, and the time it takes to complete job i is t_i. You are given m identical machines $M_1, M_2, \ldots, M_m$ to run the jobs on. Each machine can run only one job at a time, and each job must be completely run on a single machine. If you assign a set $J_j \subseteq \{1, 2, \ldots, n\}$ of jobs to machine M_j, then it will need $T_j = \sum_{i \in J_j} t_i$ time. Your goal is to partition the n jobs among the m machines to minimize $\max_i T_i$.

(a) Describe a greedy approximation algorithm for this problem.

Solution: Let J_j to be the set of jobs that M_j will run and let T_j to be the total time it machine M_j is busy (i.e., $T_j = \sum_{i \in J_j} t_i$). Consider the following algorithm:

- Initialize, $J_j = \emptyset$, and $T_j = 0$ for all j.
- For $i = 1, \ldots, n$, assign job i to machine M_j such that $T_j = \min_{1 \le k \le m}(T_k)$. (i.e., $J_j = J_j \cup i$ and $T_j = T_j + t_i$).
- Output J_j's.

This runs in $O(n \lg m)$ time by keeping a min-heap of the machines based on the current total runtime of each machine.

Alternative solution: Sort jobs in non-increasing order. Without loss of generality, let the jobs in order be $t_1, \ldots, t_n$. Let $J_j = \{t_{k:k \equiv j \mod m}\}$. Variations of this algorithm also works, with different sorting orders and assignments. This takes $O(n \lg n)$ time to sort the jobs.

(b) Show that your algorithm from part (a) is a 2-approximation algorithm.

Hint: First determine an ideal bound on the optimal solution OPT and then consider the machine M_ℓ with the longest T_ℓ, and the last job i^* that was added to it.

Solution: Since the best you can do is to evenly divide the fractional jobs, and it has to run for at least as long as the longest job, a lower bound for the optimal is:

$$L = \max\left(\frac{1}{m}\sum_{1 \le i \le n} t_i, \max_i(t_i)\right)$$

Let us define:

- M_ℓ to be the machine that runs for the longest.
- i^* to be the last job that was assigned to M_ℓ using the greedy algorithm.
- T_j^* to be the total run time of all jobs of M_j immediately before assigning i^*; $T_\ell^* = \min_j T_j^*$.

Then we have:

$$m \cdot T_\ell^* \leq \sum_{1 \leq j \leq m} T_j^* = \sum_{1 \leq i \leq i^*} t_i \leq \sum_{1 \leq i \leq n} t_i \leq m \cdot L$$

Thus $T_\ell^* \leq L$. Putting it together, we have

$$T_\ell = T_\ell^* + t_{i^*} \leq L + t_{i^*} \leq 2L \leq 2OPT$$

Therefore, this is a 2-approximation algorithm. □

Alternative solution: Let L be the lower defined above. Consider the longest job t_n. Let $k \equiv n \mod m$, and let $S_k = T_k - t_n$. It must be that $S_k \leq T_j$ for all j:

- For $j > k$, we only added elements at least as large as every element of S_k.
- For $j < k$, there are $a = \lceil \frac{m}{n} \rceil$ jobs, and the last $a - 1$ jobs in J_j are greater the first $a - 1$ jobs of J_k due to the jobs being sorted, which shows that $S_k \leq T_j$.

Then:

$$\sum_{i=1}^{n} t_i = t_n + S_k + \sum_{j \neq k} T_j \geq t_n + mS_k$$

Therefore, $mS_k \leq \sum_{i=1}^{n} t_i$, implying $S_k \leq L$. Putting this together with the fact that $t_n \leq L$ gives us:

$$t_n + S_k \leq 2L \leq 2OPT$$

□

Problem-11: Testing Polynomial Products [15 points]

An instance of the *Testing Polynomial Products (TPC)* decision problem is a quadruple of univariate polynomials (A, B, C, D). We say that (A, B, C, D) is a yes-instance if $AB = CD$, and a no-instance otherwise.

Consider the following probabilistic algorithm for solving the problem. On input polynomials (A, B, C, D): Choose uniformly at random an integer y from a set of integers $\{1, \ldots, m\}$. Compute the product $L = A(y)B(y)$ and the product $R = C(y)D(y)$ If $R = L$ outputs "PASS", otherwise output "FAIL".

(a) Analyze the probability that the algorithm is correct as a function of m and n (where the degree of any of the polynomials is at most n). Namely, it passes yes-instances and fails no-instances.

Solution: The product of two polynomials with degree at most n is a polynomial with degree at most $2n$. Therefore, the probability that for a random x, $A(x)B(x) = C(x)D(x)$ is at most $\frac{2n}{m}$. This is the prbability that the algorithm returns PASS when they are not equal.

(b) How large should we make m to make the probability of correctness at least $1 - \frac{1}{\log n}$.

Solution: From part (a), we know that the probability that the algorithm returns PASS when A(x)B(x) $\neq$ C(x)D(x) is at most $\frac{2n}{m}$. Thus, if we set $m = 2nlogn$, then the probability that the algorithm returns PASS when the products are not equal will be at most 1/logn. This will help us achieve the bound that the algorithm outputs FAIL with probability at least $1 - 1/\log n$.

Problem-12: Simplex Subs [20 points]

Sandwich shop Simplex Subs sells sandwiches which are made up of one type of meat, one type of vegetable, and one type of bread. There are n different types of each ingredient: the meats are $m_1, m_2, \ldots, m_n$, the vegetables are $v_1, v_2, \ldots, v_n$, and the breads are $b_1, b_2, \ldots, b_n$. After years of market research, they have developed a compatability mapping C, where each meat is mapped to a list of compatable vegetables and each vegetable is mapped to a list of compatable breads. Simplex subs only makes sandwiches of the form (m, v, b), where $b \in C(v)$ and $v \in C(m)$.

Each ingredient has a current supply, which is a non-negative integer. One unit of supply is enough for one sandwich, so a sandwich (m, v, b) uses one unit of meat m, one unit of vegetable v, and one unit of bread b.

Here is an example for $n = 2$:

ingredient	supply	compatability
m_1	10	v_1, v_2
m_2	10	v_2
v_1	15	b_1
v_2	15	b_2
b_1	5	N/A
b_2	20	N/A

Simplex Subs has just received a large order for k sandwiches. **The order only specifies which meats the sandwiches should have** and can be represented as a list of n numbers $[x_1, x_2, \ldots, x_n]$, where $\sum x_i = k$ and each x_i is a positive integer representing the number of sandwiches that should have meat m_i.

In the above example with $n = 2$, an order of $[10, 10]$ corresponds to 10 sandwiches with meat of type m_1 and 10 with meat of type m_2, and this order can be fulfilled by making 5 sandwiches (m_1, v_1, b_1), 5 of (m_1, v_2, b_2) and 10 of (m_2, v_2, b_2).

(a) Design and analyze an algorithm which determines whether or not Simplex Subs has the required supply to fulfill a given order.

Solution: We will design a flow network whose max flow is k only if there are enough supplies to fulfill the order.

We construct the flow network as follows:

i. Create a node for each ingredient. We use the compatability mapping as our set of edges (e.g. if $v_j \in C(m_i)$, then we have the edge (m_i, v_j)). All of these edges have infinite capacity.

ii. Then, we create a source node s and for each meat m_i, we add an edge (s, m_i) with capacity x_i. We create a sink node t and for each bread b_i, we add an edge (b_i, t) with infinite capacity.

iii. Now we must make sure not to exceed the supply for each ingredient. This is analagous to representing node capacities. The trick here is to take each vertex u (excluding s and t) and then replace it with two vertices u_{in} and u_{out}. All incoming edges to u now go to u_{in} and all outgoing edges from u now start from u_{out}. We add an edge between u_{in} and u_{out} with capacity $S(u)$, the supply of ingredient u.

Finally we run Edmonds-Karp on our flow network. If the maximum flow is k, the order can be fulfilled.

Runtime Analysis: There are $O(n^2)$ edges and $2n + 2$ nodes in our graph, the runtime of the algorithm is the minimum of $O(n^3)$ and $O(kn^2)$.

(b) Now suppose you are given a function P which maps each ingredient to a non-negative cost. For each sandwich (m, v, b), the cost to Simplex Subs is $P(m)+P(v)+P(b)$. Define a linear program which finds the minimum total cost of the k sandwiches required to fulfill the order. More points are awarded to linear programs which use asymptotically fewer variables and constraints.

Solution: We will reformulate the flow network from the previous problem as a linear program. We change the objective function to be minimizing the cost of the ingredients and we require the flow be equal to k, the total number of sandwiches in the order. Let $f(u, v)$ be represent the flow from node u to v and let I be the set of all ingredients. Then our linear program is:

Minimize: $\sum_{u \in I} f(u_{in}, u_{out})P(u)$

Subject to:

$$\sum_{v|(u,v)\in E} f(u,v) - \sum_{v|(v,u)\in E} f(v,u) = 0, \forall u \in V - \{s,t\}$$

$$\sum_{v|(s,v)\in E} f(s,v) - \sum_{v|(v,t)\in E} f(v,t) = 0$$

$$\sum_{v|(s,v)\in E} f(s,v) = k$$

This linear program has $O(n^2)$ variables and $O(n)$ constraints. A more naive approach uses every possible sandwich as a variable, which needs $O(n^3)$ variables and $O(n)$ constraints.

Problem-13: Multi-operation Queue [15 points]

Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

- PUSH(x): Add item x to the end of the sequence.
- PULL(): Remove and return the item at the beginning of the sequence.

It is possible to implement a queue using a doubly-linked list and a counter, so that the worst-case time per operation is $O(1)$.

(a) Now suppose we want to support the following operation instead of PULL:

- MULTIPULL(k): Remove the first k items from the front of the queue, and return the k^{th} item removed.

Suppose we use the following algorithm to implement MULTIPULL:

```
MULTIPULL(k)
1  for i = 0 to k − 1:
2        x = PULL()
3        if i == k − 1:
4              return x
```

Use the *accouting* method to prove that in any intermixed sequence of PUSH and MULTIPULL operations, the amortized cost of each operation is $O(1)$.

Solution: Assume that each PUSH and PULL costs 1 unit. For each PUSH, we charge 2 units, 1 to pay for the cost of PUSH and 1 to store in the bank account. For MULTIPULL(k), it costs k units and pops k elements from the queue. Each of these k elements stored 1 unit to the bank when they were pushed to the queue, therefore, we can use the k units stored by the k elements to pay for the cost of the MULTIPULL. In summary, we charge 2 unites for PUSH and 0 units for MULTIPULL, so the amortized cost of PUSH and MULTIPULL is $O(1)$.

(b) Now suppose we also want to support the following operation instead of PUSH:

- MULTIPUSH(k, x): Insert $x, x+1, \ldots, x+k-1$ into the back of the queue.

Suppose we use the following algorithm to implement MULTIPUSH:

MULTIPUSH(k, x)

1 **for** $i = 0$ to $k-1$:
2 PUSH$(x+i)$

Use the *aggregate* method to prove that for any integers l and n, there is a sequence of l MULTIPUSH and MULTIPULL operations that require $\Omega(nl)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

Solution: Consider the following sequence:

MULTIPUSH(n, x), MULTIPULL(n), MULTIPUSH(n, x), MULTIPULL(n), ...

Each MULTIPUSH or MULTIPULL takes $\Omega(n)$ time, where n is also the maximum number of items in the queue at any time.

(c) Describe a data structure that supports arbitrary intermixed sequences MULTIPUSH and MULTIPULL operations in $O(1)$ amortized cost per operation. In this question, you are asked to use the *potential function* method to prove that the amotrized cost of MULTIPUSH and MULTIPUSH is $O(1)$ for your data structure.

Solution: It is possible to use a doubly-linked list to implement a front-back queue that supports the following operation.

- PUSHFRONT(x): Add item x to the begining of the sequence.
- PUSHBACK(x): Add item x to the end of the sequence.
- PULL(): Remove and return the item at the beginning of the sequence.

Our new data structure uses the above front back queue, except that each item (x, k) in the queue represents elements $(x, x+1, \ldots, x+k-1)$.

Here is the pseudocode for MULTIPUSH(k, x) and MULTIPULL(k):

```
MULTIPUSH(k, x)
1  PUSHBACK((x, k))

MULTIPULL(k)
1  while k > 0
2      (x_0, k_0) = PULL()
3      k = k − k_0
4      if k == 0
5          return x_0 + k_0 − 1
6      if k < 0
7          PUSHFRONT((x + k_0 + k, −k))
8          return x + k_0 + k − 1
```

To show that the amortized cost is $O(1)$, define potential function $\phi(D_i) = s_i$ where s_i is the number of items in the front-back queue at state D_i. Then, for MULTIPUSH, amortized cost:

$$\hat{c}_{\text{MULTIPUSH}} = c_{\text{MULTIPUSH}} + \phi(D_{new}) - \phi(D_{old}) = 1 + 1 = 2$$

For MULTIPULL(k), suppose PULL is called l times, MULTIPULL has actual cost $l + 1$ for the l PULL and the one PUSHFRONT, and has amortized cost:

$$\begin{aligned}\hat{c}_{\text{MULTIPULL}} &= c_{\text{MULTIPULL}} + \phi(D_{new}) - \phi(D_{old}) \\ &\leq l + 1 - (l - 1) \\ &= 2\end{aligned}$$

Problem-14: Largest Weight Cycle [25 points]

You are given a weighted tree on n vertices with possibly negative weights. You are allowed to add a single edge of weight 0 to this tree. Your goal is to maximize the weight of the cycle thus formed. Find an efficient way to do this.

i. Give an algorithm with running time $\mathcal{O}(n^3)$.

Solution: Add an edge between all pairs of vertices. Find the weight of the resulting cycle using e.g. BFS. Return the pair of vertices maximizing this weight.

ii. Give an algorithm with running time $\mathcal{O}(n^2)$.

Solution: Observe that maximizing the weight of the cycle formed is equivalent to finding the longest path in the tree. Because paths in a tree are unique, we can compute pathlengths between all pairs of vertices by running BFS as a single source shortest path algorithm from each vertex.

iii. Give an algorithm with running time $\mathcal{O}(n)$.

Solution: For each node u, we will compute $u.first$ and $u.second$, the two largest pathlengths associated with paths starting at u and ending at u or a descendant of u, with the restriction that these two paths may not pass through the same child of u. We can compute these values with a single (depth-first) traversal of the tree as follows:

COMPUTE-LONGEST-SUBPATHS(G, u)
  $u.first = u.second = 0$
  for each child v of u:
    # Find the length of the longest path starting at v and ending at v or a descendant of v.
    $subpath =$ COMPUTE-LONGEST-SUBPATHS(G, v)
    # Add this to the edge weight from u to v to get the length of the longest path
    # starting at u and ending at v or a descendant of v.
    $path = subpath + w(u, v)$
    # Keep track of the two largest such paths through distinct children of u.
    $u.second = \max(u.second, path)$
    Swap $u.first$ and $u.second$ if necessary to ensure that $u.first \geq u.second$.
  return $u.first$

Our final algorithm uses these values to compute the longest path in the tree:

LARGEST-WEIGHT-CYCLE(G)
  COMPUTE-LONGEST-SUBPATHS$(G, root)$
  $longest = 0$
  for $w \in V$:
    $longest = \max(longest, w.first + w.second)$
  return $longest$

Runtime: We visit each node twice, so the running time is $O(n)$.

Correctness: After running COMPUTE-LONGEST-SUBPATH$(root)$, we will have computed values for $u.first$ and $u.second$ for all $u \in V$ (and this is done correctly by the optimal substructure of the longest path problem).

Now, consider the longest path in the tree. Consider the node in this path that is closest to the root of the tree, call it w. The path must either end at w or not. If it ends at w, it is equivalently the longest path starting at w and ending at w or a descendant of w. Otherwise, because w is closest to the root, this path must pass through two children of w, and so it is the path consisting of the two longest paths starting at w and passing through distinct children of w, which is exactly what we computed. We handle both cases at once by initializing $u.first = u.second = 0$.

Problem-15: k-Center Problem [27 points]

Consider a simple undirected complete weighted graph $G = (V, E, d)$ where $d(u, v)$ represents the distance between u and v. We assume that $d(u, v)$ is symmetric and satisfies the triangle inequality. More formally for any $u, v, w \in V$ we have that $d(u, v) = d(v, u)$ and $d(u, v) \leq d(u, w) + d(w, v)$. We define the distance of a point $v \in V$ from a set of points $P \subseteq V$ as follows $d(v, P) = \min_{u \in P} d(u, v)$.

Given an integer number $k \geq 1$ our goal is to find a set $P \subseteq V$ of k points such that $\max_{v \in V} d(v, P)$ is minimized. This problem is known as the *k-center problem*. For simplicity we will assume that no two pairs of points have equal distances, i.e. $d(u, v) \neq d(w, z)$ whenever $\{u, v\} \neq \{w, z\}$. The goal of this problem is to find a 2-approximation algorithm for the k-center problem.

(a) Prove that for $k = 1$, the solution $P = \{u\}$ for any $u \in V$ is a 2-approximation to the 1-center problem.

Solution: Suppose the optimal solution is $P^* = \{x\}$. By the triangle inequality, for any vertices $u, y \in V$, we have $d(u, y) \leq d(u, x) + d(x, y) \leq 2 \cdot OPT$.

Given a solution $P = \{p_1, \ldots, p_k\}$, define its i-th cluster C_i to be the set of vertices that are closer to p_i than to any other vertex in P. More formally:

$$C_i = \{u \in V \mid d(u, p_i) < d(u, p_j) \ \ \forall j \neq i\}$$

Let OPT be the value of the optimal solution to the k-center problem and let P^* be a set of k points that achieves this optimum, namely, $OPT = \max_{v \in V} d(v, P^*)$.

(b) Consider any solution $P = \{p_1, \ldots, p_k\}$. Argue that, if p_i and p_j are contained in the same cluster of P^*, then $d(p_i, p_j) \leq 2 \cdot OPT$.

Solution: Denote by p the vertex in P^* corresponding to this cluster. Again apply the triangle inequality: $d(p_i, p_j) \leq d(p_i, p) + d(p, p_j) \leq 2 \cdot OPT$. becuase p is by definition the point in P^* closest to both p_i and p_j.

(c) Consider any solution $P = \{p_1, \ldots, p_k\}$ and assume that, for every cluster of P^*, there is a $p_i \in P$ contained in this cluster. Prove that P is a 2-approximation to the k-center problem.

Solution: Consider any point p_i in the graph. We will show that it is within a distance $2 \cdot OPT$ of some point in P, and this will complete the proof.

By assumption, there is a point $p_j \in P$ contained in the same cluster of P^* as p_i. By part (b), $d(p_i, p_j) \leq 2 \cdot OPT$.

(d) Using ideas from the previous parts, propose a greedy 2-approximation algorithm for the k-center problem. Analyze the running time of your algorithm and prove its correctness.

Solution:

GREEDY K-CENTER APPROX(G)

Initialize P to contain a single random vertex.
for $\_ = 2$ to k
  Add to P the vertex $v \in V$ currently maximizing $d(v, P)$.
return P

Runtime: Each iteration of the loop requires $O(|V||P|) = O(k|V|)$ time, so the overall runtime is $O(k^2|V|)$.

Correctness: If P satisfies the conditions of (c), then it is certainly a 2-approximation. Otherwise, by the pigeonhole principle, two points $p_i, p_j \in P$ satisfy the conditions of (b). And because at each iteration we choose the vertex v that is farthest from P, this guarantees that P is a 2-approximation.

Problem-16: Dynamic Programming. [15 points]

Prof. Child is cooking from her garden, which is arranged in grid with n rows and m columns. Each cell (i, j), with $1 \leq i \leq n$ and $1 \leq j \leq m$, has an ingredient growing in it, with ***tastiness*** given by a positive value $T_{i,j}$.

Prof. Child doesn't like cooking "by the book". To prepare dinner, she will stand at a cell (i, j) and pick one ingredient from each quadrant relative to that cell. The tastiness of her dish is the product of the tastiness of the four ingredients she chooses. Help Prof. Child find an $O(nm)$ dynamic programming algorithm to maximize the tastiness of her dish.

Here the four ***quadrants*** relative to a cell (i, j) are defined as follows:

$$\begin{aligned} \text{top-left} &= \{\text{all cells } (a,b) \mid a < i, b < j\}, \\ \text{bottom-left} &= \{\text{all cells } (a,b) \mid a > i, b < j\}, \\ \text{top-right} &= \{\text{all cells } (a,b) \mid a < i, b > j\}, \\ \text{bottom-right} &= \{\text{all cells } (a,b) \mid a > i, b > j\}. \end{aligned}$$

Because Prof. Child needs all four quadrants to be non-empty, she can only stand on cells (i, j) where $1 < i < n$ and $1 < j < m$.

(a) Define $TL_{i,j}$ to be maximum tastiness value in the top-left quadrant of cell (i, j):

$$TL_{i,j} = \max\{T_{a,b} \mid 1 \leq a \leq i, 1 \leq b \leq j\}$$

Find a dynamic programming algorithm to compute $TL_{i,j}$, for all $1 < i < n$ and $1 < j < m$, in $O(nm)$ time.

Solution: When trying to calculate $TL_{i,j}$, we see that the maximum can be at cell (i, j). If not, it must lie either in the rectangle from $(1, 1)$ to $(i, j - 1)$, or the rectangle from $(1, 1)$ to $(i - 1, j)$, or both. These three overlapping cases cover our required rectangle. We then have:

$$TL_{i,j} = \max\{T_{i,j},\, TL_{i-1,j},\, TL_{i,j-1}\}$$

For the base cases, we can just set $TL_{0,j} = TL_{i,0} = 0$ for all valid values of i and j. We can compute the DP value for each state in $O(1)$ time; since there are nm states, the algorithm runs in $O(nm)$ time.

(b) Use the idea in part (a) to obtain an $O(nm)$ algorithm to find the tastiest dish.

Solution: In part (a) we calculated range maximum for the top-left quadrant. We can similarly define range maximums for the other quadrants. Let:

$$\begin{aligned} BL_{i,j} &= \max\{T_{a,b} \mid i \leq a \leq n, 1 \leq b \leq j\} \\ TR_{i,j} &= \max\{T_{a,b} \mid 1 \leq a \leq i, j \leq b \leq m\} \\ BR_{i,j} &= \max\{T_{a,b} \mid i \leq a \leq n, j \leq b \leq m\} \end{aligned}$$

All of these can be computed in $O(nm)$ time similar to TL. To calculate the tastiest dish Prof. Child can cook when she stands at cell (i, j) for $1 < i < n$ and $1 < j < m$, compute the product:

$$TL_{i-1,j-1} \cdot BL_{i+1,j-1} \cdot TR_{i-1,j+1} \cdot BR_{i+1,j+1}$$

and pick the maximum product. This can be done in $O(nm)$ time.

Problem-17: Sushi Delivery [20 points]

After having all of their cattle repurposed as space-objects in a galactic maze, John and Bob have moved on to running a sushi resturant in a big city. The streets of this city are well planned: they are either North-South or East-West, and are evenly spaced in each direction. In other words, it can be viewed as a grid. The resturant has two delivery drivers, A and B, available to run deliveries.

One evening, n customers $1 \dots n$ called requesting sushi deliveries. These requests are given in order in which the customers called. A and B wish to split these requests so that each delivery is made by exactly one of them, and they return to the restaurant at the end. In order to be fair, if customers $i < j$ are assigned to the same driver then the delivery must be made to the customer i before customer j, even if that makes the trip longer. Note that if i and j are assigned to different drivers, deliveries can be made to them in any order.

A and B would like to keep the total distance that they travel small. However, each of them also know some of the customers better than others. Customer i will give a tip of value a_i if A delivers, or will give a tip of value b_i if B delivers. As a result, A and B would like to maximize their total happiness, which is the amount of tip that they receive minus the total distance that they travel.

Each location is given as the intersection of two streets, and is parameterized by two values, (x_i, y_i), the id of the North-South street and the East-West street that the intersection is on respectively. The resturant is located at $p_0 = (x_0, y_0)$, and customer i is located at $p_i = (x_i, y_i)$. Since A and B drive along the streets, the distance between points i and j is:

$$d(i, j) = |x_i - x_j| + |y_i - y_j|$$

(a) Describe and analyze an algorithm that takes n, the locations of customers,

$$p_0 = (x_0, y_0), p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$$

and the tip values $a_1 \dots a_n$ and $b_1 \dots b_n$ as input and outputs the maximum happiness for A and B. Prove that your algorithm is correct. Your algorithm should run in $O(n^{10})$ time or faster.

Solution: The solution is by dynamic programming. The state is specified by the people who we have made deliveries to so far, and where the other person is. We use $DP_A(i, j)$ to denote the maximum happiness when A just delivered to person i, and B's last delivery was at person $j \leq i$, and define $DP_B(i, j)$ symmetrically as the maximum happiness when B just delivered to person i, and A's last delivery was to some $j \leq i$.

Then the base case is

$$DP_A(1, 0) = a_1 - dist(0, 1)$$

and

$$DP_B(1, 0) = b_1 - dist(0, 1)$$

For the transition, we simply need to consider whether A or B made the last delivery. We have:

- If $i-1 > j$ then $DP_A(i,j) = a_i + DP_A(i-1,j) - dist(i-1,i)$.
- If $i-1 = j$ then $DP_A(i,j) = a_i + \max_{0 \le k < j} DP_B(j,k) - dist(k,i)$.

and symmetrically for $DP_B(i,j)$:

- If $i-1 > j$ then $DP_B(i,j) = b_i + DP_B(i-1,j) - dist(i-1,i)$.
- If $i-1 = j$ then $DP_B(i,j) = b_i + \max_{0 \le k < j} DP_A(j,k) - dist(k,i)$.

(b) Give an algorithm for finding the maximum happiness that uses $O(n)$ space and runs in $O(n^2)$ time or faster .

Solution: We can reduce to $2n$ states by noticing that we're giving states where A and B 'switch' deliveries priority. Let $DP_A(i)$ denote the maximum total happiness when A delivered to i, and B delivered to j, and $DP_B(i)$ similarly.
The base cases are still $DP_A(1) = a_1 - dist(0,1)$ and $DP_B(1) = b_1 - dist(0,1)$, for the transition we can enumerate over the last customer that A made a delivery to. Suppose this customer is $j-1$, then B delivered to customer j, as well as all customers from $j+1$ to $i-1$. The happiness of A delivering to $j-1$, B to j is given by $DP_B(j)$, while the happiness of B's intermediate deliveries can be calculated directly. This gives the transition

$$DP_A(i) = a_i + \max_{1 \le j < i} DP_B(j) - dist(j-1,i) + \sum_{k=j+1}^{i-1} b_k - dist(k-1,k)$$

and symmetrically:

$$DP_B(i) = b_i + \max_{1 \le j < i} DP_A(j) - dist(j-1,i) + \sum_{k=j+1}^{i-1} a_k - dist(k-1,k)$$

This gets the number of states, and therefore memory down to $O(n)$. To get a fast running time, observe that $\sum_{k=j+1,k \le i-1} b_k - dist(k-1,k)$ is the sum of a subsequence of $b_i - dist(i-1,i)$. Therefore, we can define partial sums of this array:

$$S_B(i) = \sum_{1 \le j \le i} b_j - dist(j-1,j)$$

which the transition for DP_A becomes:

$$DP_A(i) = a_i + \max_{1 \le j < i} DP_B(j) - dist(j-1,i) + S_B(i-1) - S_B(j)$$

Since each transition contains a constant number of terms, and there are a total of $O(n^2)$ transitions, this DP can be completed in $O(n^2)$ time.

(c) Give an algorithm for finding the maximum happiness that runs in $O(n \log^3 n)$ time or faster.

Solution: We perform divide-and conquer on the indices. To solve the problems for indices $[l, r]$, we find the midpoint of the interval m, and perform the following calls:

- Recurse on $[l, m]$.
- Resolve all transitions from $[l, m]$ to $[m + 1, r]$.
- Recurse on $[m + 1, r]$.

It suffices to show that all transitions from $[l, m]$ to $[m+1, r]$ can be resolved in $O((r - l) \log n)$ time, as the master theorem would then lead to an $O(n \log^2 n)$ time algorithm. By symmetry, we will only consider how to resolve the transitions to DP_A. That is, given a set of DP_B, we want to compute for each $m < i \leq r$:

$$\begin{aligned}
&a_i + \max_{l \leq j \leq m} DP_B(j) - dist(j-1, i) + S_B(i-1) - S_B(j) \\
&= a_i + S_B(i-1) + \max_{l \leq j \leq m} DP_B(j) - dist(j-1, i) - S_B(j) \\
&= a_i + S_B(i-1) + \max_{l \leq j \leq m} (DP_B(j) - S_B(j)) - |x_i - x_{j-1}| - |y_i - y_{j-1}|
\end{aligned}$$

There are two cases involving x_i and x_{j-1}: $x_i \leq x_{j-1}$ or $x_i > x_{j-1}$. The cases involving y are also similar, leading to a total of 4 cases. It suffices to show how to handle all the cases with $x_i \leq x_{j-1}$ and $y_i \leq y_{j-1}$, since we can then rotate the plane by 90 degrees 3 times, and use this procedure to handle the rest of the transitions.

For this case, the DP transition simplifies to:

$$\begin{aligned}
&\max_{l \leq j \leq m, x_{j-1} \geq x_i, y_{j-1} \geq y_i} (DP_B(j) - S_B(j)) - |x_i - x_{j-1}| - |y_i - y_{j-1}| \\
&= x_i + y_i + \max_{l \leq j \leq m, x_{j-1} \geq x_i, y_{j-1} \geq y_i} (DP_B(j) - S_B(j)) - x_{j-1} - y_{j-1}.
\end{aligned}$$

Note that the term inside is a single value that only depends on j. This means that we can associate with each j a value

$$v_j = (DP_B(j) - S_B(j)) - x_{j-1} - y_{j-1},$$

and compute the following for each i between $m + 1$ and r:

$$\max_{l \leq j \leq m, x_{j-1} \geq x_i, y_{j-1} \geq y_i} v_j$$

This can be done in $O((r - l) \log n)$ time in two ways:

i. Build an augmented binary search tree with y values as keys that supports MAX-AFTER queries. Then we sort all the x_i and x_{j-1}, and proceed through them in decreasing order. If we encounter x_{j-1} for some j, we insert the key-value pair

(y_{j-1}, v_j) into the search tree. Otherwise, if we encounter some x_i, we simply query for the maximum v value among all entries entered so far. These entries are precisely the ones satisfying $y_{j-1} \geq y_i$. Each insertion / query takes $O(\log n)$ time, giving a total cost of $O((r-l)\log n)$.

ii. Perform a divide-and-conquer on the median of the x values. Let this x value be x_{mid}, then it suffices to give a routine that for each $i \in [m+1, r]$ with $x_i \leq x_{mid}$, computes:

$$\max_{l \leq j \leq m, x_{j-1} \geq x_{mid}, y_{j-1} \geq y_i} v_j$$

Note that the set of indices i where $i \in [m+1, r]$ and $x_i \leq x_{mid}$ and j for which $l \leq j \leq m, x_{j-1} \geq x_{mid}$ are both fixed sets. Therefore we can abstract this problem as given two sets S^+ and S^-, compute for all $i \in S^-$:

$$\max_{j \in S^+, y_{j-1} \geq y_i} v_j$$

This is a prefix-min query: if we sort all y values in S^+ and S^- with y_{j-1} and y_i as key respectively, then all these queries can be answered in $O(|S^+| + |S^-|)$ time. If we use an $O(n \log n)$ time sorting algorithm, then the overall cost of resolving these transitions becomes $O((r-l)\log^2 n)$, giving a total of $O(n \log^3 n)$.
To obtain a faster algorithm, note that if we have y values in S^+ and S^- in sorted order, then we can sort all y values in $S^+ \cup S^-$ in linear time using merge sort. This means that the sorting can be done in conjunction with the divide-and-conquer, bringing the total cost back down to $O((r-l)\log n)$.

Hint: Parts (a) and (b) can be solved for any distance function $d(i, j)$.