CS 170 Homework 8

Due **Friday 10/25/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 Knightmare

Give a dynamic programming algorithm to find the number of ways you can place knights on an X by Y ($X > Y$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Knights can move in a 2×1 shape pattern in any direction.

Provide a 4-part solution. Your algorithm's runtime should be $O(2^{3Y}XY)$, and return your answer mod 1789.

Hint: consider an approach similar to Dis6 Q4 (Counting Strings).

Solution: The first part of this solution is the old 3-part format (with full explanation). The 4-part format is below it.

3-part solution:

Algorithm: We use length Y bit strings to represent the configuration of rows of the chessboard (1 means there is a knight in the corresponding square and otherwise 0).

The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(i-1) \times Y$ chessboard and use it to solve the $i \times Y$ case. Note that as we iteratively incrementing h, a knight in the h-th rows can only affect configurations of rows $i+1$ and $i+2$. So we can denote $K(i, u, v)$ as the number of possible configurations of the first i rows with u being the $(i-1)$-th row and v being the i-th row, and then use dynamic programming to solve this problem.

Let a list of bitstrings be valid if placing the knights in the first row according to the first bitstring, in the second row according to the second bitstring, etc. doesn't cause two knights to attack each other. Then we have $K(2, u, v) = 1$ if u, v are valid and 0 otherwise for all u, v pairs.

For $K(i, v, w)$ we have:

$$K(i, v, w) = \sum_{u : u,v,w \text{ are valid}} K(i-1, u, v) \bmod 1789$$

Proof of Correctness: The only 2-row configuration of knights ending in row configurations u, v is the configuration u, v itself. So $K(2, v, w) = 1$ if u, v are valid. Otherwise, $K(2, v, w) = 0$.

For $i > 2$, the above recurrence is correct because for any valid i-row configuration ending in v, w, the first $i - 1$ rows must be a valid configuration ending in u, v for some u, and for this same u, the last three rows u, v, w must be also valid configuration. Moreover, this correspondence between i-row and $(i-1)$-row configurations is bijective.

Runtime Analysis: To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(Y)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3Y}Y)$ which gives us an overall runtime of $O(2^{3Y}XY)$ operations. Although there are a potentially exponential number of possible configurations (at most 2^{XY}), we are only interested in the answer mod 1789, which means that all of our numbers are constant size, and so arithmetic on these numbers is constant time. This gives us a final runtime of $O(2^{3Y}XY)$.

4-part solution:

(a) **Subproblems:** $f(i, u, v)$ is the number of possible valid configurations mod 1789 of the first i rows with u being the configuration of the $(i - 1)$-th row and v being the configuration of the i-th row.

(b) **Recurrence and Base Cases:** For $i > 2$, $f(i, v, w) = \sum_{u:u,v,w \text{ are valid}} f(i-1, u, v)$ mod 1789. For the base case, $f(2, u, v) = 1$ if u, v are valid and 0 otherwise for all u, v pairs. No other base cases are needed.

(c) **Proof of Correctness:** See proof of correctness in 3-part solution above.

(d) **Runtime and Space Complexity:** See runtime above to yield $O(2^{3Y}XY)$. For space complexity, note that there are only $O(X2^{2Y})$ subproblems (X rows, 2^Y settings to the $(X-1)$th row, and 2^Y settings to the Xth row). Each subproblem is a number of possible configurations mod 1789, bounded above by 1789, so our total space is $O(X2^{2Y})$. Note that since each subproblem only depends on subproblems of the previous row, we could reduce this by a factor of X to $O(2^{2Y})$ by recycling space from earlier rows.

3 Max Independent Set Again

You are given a connected tree T with n nodes and a designated root r, where every vertex v has a weight $W[v]$. A set of nodes S is a k-independent set of T if $|S| = k$ and no two nodes in S have an edge between them in T. The weight of such a set is given by adding up the weights of all the nodes in S, i.e.

$$W(S) = \sum_{v \in S} W[v].$$

Given an integer $k \leq n$, your task is to find the maximum possible weight of any k-independent set of T. We will first tackle the problem in the special case that T is a binary tree, and then generalize our solution to a general tree T.

(a) Assume that T is a binary tree, i.e. every node has at most 2 children. Describe an $O(nk^2)$ algorithm that solves this special case, and analyze its runtime. Proof of correctness and space complexity analysis are not required.

Hint: your subproblem should be similar to the one used for the max independent set in a tree problem, but you need to also account for the "current" independent set size (why do we need to do this?).

(b) Now, consider any arbitrary tree T, with no restrictions on the number of children per node. Describe how we can add up to $O(n)$ "dummy" nodes (i.e. nodes with weight 0) to T, as well as some edges, so that the resulting tree is a binary tree T_b.

(c) Describe an $O(nk^2)$ algorithm to solve the general case (i.e. when T is any arbitrary tree), and analyze its runtime. Proof of correctness and space complexity analysis are not required.

Hint: there exists two ways (known to us) to solve this. One way is to combine parts (a) and (b), and then modify the recurrence to account for the dummy nodes. The other way involves 3D dynamic programming, in which you directly extend your recurrence from part (a) to iterate across vertices' children. We recommend the first way as it may be easier to conceptualize, but in the end it is up to you!

Solution:

(a) **Algorithm Description:** Let $f_0[v][i]$ be the max weight independent set of size i for the subtree rooted at v, with the constraint that v is not included in the set. Also, let $f_1[v][i]$ be the overall max weight independent set of size i for v's subtree. Our final answer would be $f_1[r][k]$.

For each v, i, we compute $f_b[v][i]$ for $b \in \{0, 1\}$ with the following recurrence:

$$f_0[v][i] = \max_{0 \leq j \leq i} f_1[\text{left}(v)][j] + f_1[\text{right}(v)][i-j] \tag{1}$$

$$f_1[v][i] = \max \begin{cases} f_0[v][i], \\ \max_{0 \leq j \leq i-1} W[v] + f_0[\text{left}(v)][j] + f_0[\text{right}(v)][i-j-1] \end{cases} \tag{2}$$

where left(v) and right(v) are the left and right children of v, respectively. The idea here is that we need to "distribute" the k nodes in the independent set to both children, while also keeping track of whether v is in the independent set to preserve the property of vertex independence.

For the base cases, we set $f_b[v][0] = 0$ for $b \in \{0, 1\}$ and $f_0[\ell][j] = 0, f_1[\ell][j] = W[\ell]$ for every leaf node ℓ, $j \in [k]$.

The order in which we solve the subproblems is from the leaves to the root, i.e. post-order.

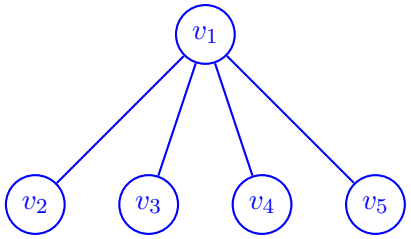
Runtime Analysis: for each $b \in \{0, 1\}$, there are nk subproblems because v can taken on n values and j can take on k values. Each subproblem takes $O(k)$ time because we have to take the max over all $j \in [i]$ or $j \in [i-1]$. Thus, the overall runtime is

$$2 \cdot nk \cdot O(k) = O(nk^2)$$

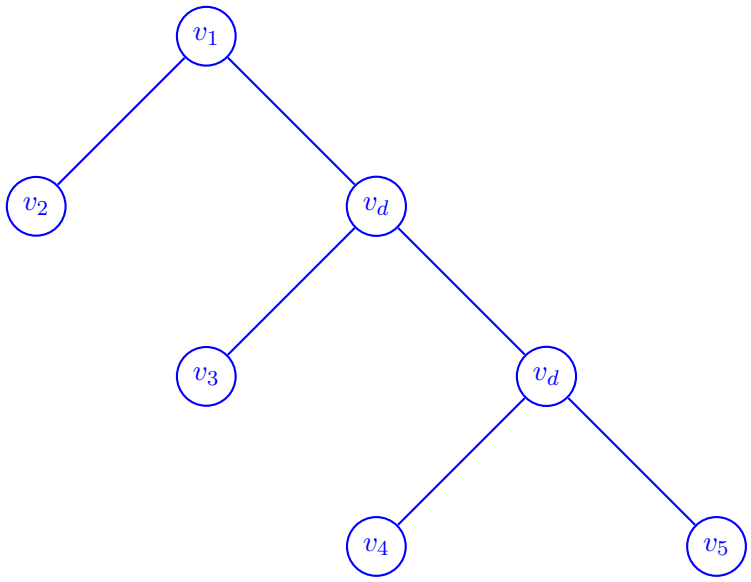
as desired.

(b) For every node v in T with more than two children, add a dummy node of weight 0 as the right child of v and connect all but the leftmost of v's children to the dummy node instead. Recursively performing this modification will result in a binary tree, while adding at most $O(N)$ dummy nodes.

For example, if T were the following tree where $W[v_i] = i$:



After following the described procedure, we would end up with the following binary tree T_b:



where $A[v_d] = 0$ for all the dummy nodes v_d.

(c) To solve the problem for a general T we want to use part (b) to convert it to a binary tree T_b and run the algorithm from part (a) on T_b. However, we actually have to adjust the recurrence from part (a) to account for the newly added dummy nodes!

To do this, we want to somehow define the subproblem for our dummy nodes such that they propagate the "state" of what their parent assigns them (i.e. whether they're in the k-independent set or not). For example, in the tree example from part (b), we want so that if v_1 is included in the k-independent set, then our algorithm will propagate non-inclusivity through the v_d's to ensure that none of v_3, v_4, v_5 are included.

Algorithm Description: We redefine the subproblems as follows:

$f_0[v][i]$ = max weight independent set of size i for the subtree rooted at v, where v is not included
$f_1[v][i]$ = max weight independent set of size i for the subtree rooted at v, where v is included
$f[v][i]$ = overall max weight independent set of size i for the subtree rooted at v's

and the final answer is $f[r][k]$ on T_b, which is obtained by running the procedure from part (b) on T. Then, we modify the recurrence as follows:

$$f_0[v][i] = \begin{cases} \max_{0\le j\le i} f[\text{left}(v)][j] + f[\text{right}(v)][i-j] & \text{if } W[v] \neq 0 \\ \max_{0\le j\le i} f_0[\text{left}(v)][j] + f_0[\text{right}(v)][i-j] & \text{if } W[v] = 0 \end{cases}$$

$$f_1[v][i] = \begin{cases} \max_{0\le j\le i-1} W[v] + f_0[\text{left}(v)[j] + f_0[\text{right}(v)][i-j-1] & \text{if } W[v] \neq 0 \\ \max_{0\le j\le i} f[\text{left}(v)[j] + f[\text{right}(v)][i-j] & \text{if } W[v] = 0 \end{cases}$$

$$f[v][i] = \max(f_0[v][i], f_1[v][i])$$

The base cases and the order of solving subproblems remain the same as in part (a).

Runtime Analysis: Converting T to T_b takes $O(|V| + |E|) = O(n)$ time. Note that T_b has only $O(n)$ more nodes (and thus edges) than T, so running part (a) on T_b will still take $O(nk^2)$ time. Thus, the overall runtime is

$$O(n) + O(nk^2) = O(nk^2).$$

Alternate Solution: there's actually a way to solve this problem for general T directly, i.e. without having to convert it to a binary tree and solving for the special case. Let f_0 and f_1 be defined in the same way as back in part (a).

Note that to calculate $f_b[v][i]$ for $b \in \{0, 1\}$, we will need to consider the $f_{\bar{b}}$ values for every child of v. However, if v has more than one child, we also need to consider all possible ways of distributing i among its children's subtrees. One way to do this by calculating additional DP subproblems.

Let c_α be the α-th child of v, for some ordering of $v's$ children. Then define $g_1[v][\alpha][i]$ to be the max weight k-independent set in the union of subtrees of $c_0, \ldots, c_\alpha$. Define $g_0[v][\alpha][i]$ similarly with the constraint that none of $c_0, \ldots, c_\alpha$ are included in the set. Then, we have the following recurrence relations:

$$g_b[v][\alpha][i] = \begin{cases} f_b[c_0][i] & \text{if } \alpha = 0 \\ \max_{0 \le j \le i} g_b[v][\alpha - 1][j] + f_b[c_\alpha][i - j] & \text{if } \alpha > 0 \end{cases}$$

for $b \in \{0, 1\}$. Suppose v has $d(v)$ children. Then, after first calculating g as above for v, we can calculate $f[v][i]$ for $i > 1$ as follows,

$$f_0[v][i] = g_1[v][d(v)][i]$$
$$f_1[v][i] = W[v] + \max\left(f_0[v][i], g_0[v][d(v)][i - 1]\right)$$

Runtime Analysis: Observe that even though the g_b DP looks like a 3-D array, there are only $d(v)$ possible values of α for every v. Since $\sum_v d(v) = n - 1$, we only need to calculate $O(nk)$ values of g_b. Thus, including the g_b's, there are a total of $O(nk) + O(nk) = O(nk)$ subproblems, with each subproblem taking $O(k)$ time. This yields an overall runtime of

$$O(nk) \cdot O(k) = O(nk^2).$$

4 Canonical Form LP

Recall that any linear program can be reduced to a more constrained *canonical form* where all variables are non–negative, the constraints are given by $\leq$ inequalities, and the objective is the maximization of a cost function.

More formally, our variables are x_i. Our objective is $\max c^\top x = \max \sum_i c_i x_i$ for some constants c_i. The jth constraint is $\sum_i a_{ij} x_i \leq b_j$ for some constants a_{ij}, b_j. Finally, we also have the constraints $x_i \geq 0$.

An example canonical form LP:

$$\text{maximize } 5x_1 + 3x_2$$

$$\text{subject to } \begin{cases} x_1 + x_2 - x_3 \leq 1 \\ -(x_1 + x_2 - x_3) \leq -1 \\ -x_1 + 2x_2 + x_4 \leq 0 \\ -(-x_1 + 2x_2 + x_4) \leq 5 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

For each of the subparts below, describe how we should modify it to so that it satisfies canonical form. If it is impossible to do so, justify your reasoning.

Note that the subparts are independent of one another. Also, you may assume that variables are non-negative unless otherwise specified.

(a) Min Objective: $\min \sum_i c_i x_i$

(b) Lower Bound on Variable: $x_1 \geq b_1$

(c) Bounded Variable: $b_1 \leq x_1 \leq b_2$

(d) Equality Constraint: $x_2 = b_2$

(e) More Equality Constraint: $x_1 + x_2 + x_3 = b_3$

(f) Absolute Value Constraint: $|x_1 + x_2| \leq b_2$ where $x_1, x_2 \in \mathbb{R}$

(g) Another Absolute Value Constraint: $|x_1 + x_2| \geq b_2$ where $x_1, x_2 \in \mathbb{R}$

(h) Min Max Objective: $\min \max(x_1, x_2, x_3, x_4)$

Hint: use a dummy variable!

Solution:

(a) $\max - \sum_i c_i x_i$

(b) $-x_1 \leq -b_1$

(c) $-x_1 \leq -b_1$ and $x_1 \leq b_2$

(d) $x_2 \leq b_2$ and $-x_2 \leq -b_2$.

(e) $x_1 + x_2 + x_3 \leq b_3$ and $-x_1 - x_2 - x_3 \leq -b_3$

(f) First, we represent $|x_1 + x_2| \leq b_2$ using linear constraints as follows:

$$x_1 + x_2 \leq b_2, -x_1 - x_2 \leq b_2$$

Then, we enforce the non-negativity of variables by substituting $x_1 = x_1^+ - x_1^-$ and $x_2 = x_2^+ - x_2^-$:

$$(x_1^+ - x_1^-) + (x_2^+ - x_2^-) \leq b_2$$
$$(x_1^+ - x_1^-) + (x_2^+ - x_2^-) \geq -b_2$$

(g) In order to enforce $|x_1 + x_2| \geq b_2$, we must use

$$(x_1 + x_2 \geq b_2) \cup (x_1 + x_2 \leq -b_2),$$

but an LP can only represent an intersection (not union) of constraints. In other words, it is impossible because we cannot have both $x_1 + x_2 \geq b_2$ and $x_1 + x_2 \leq -b_2$ hold at the same time (unless $b_2 = 0$).

(h) $\max -t\,, \quad x_1 \leq t\,, \quad x_2 \leq t\,, \quad x_3 \leq t\,, \quad x_4 \leq t$

5 Baker

You are a baker who sells batches of brownies and cookies (unfortunately no brookies... for now). Each brownie batch takes 4 kilograms of chocolate and 2 eggs to make; each cookie batch takes 1 kilogram of chocolate and 3 eggs to make. You have 80 kilograms of chocolate and 90 eggs. You make a profit of 60 dollars per brownie batch you sell and 30 dollars per cookie batch you sell, and want to figure out how many batches of brownies and cookies to produce to maximize your profits.

(a) Formulate this problem as a linear programming problem; in other words, write a linear program (in canonical form) whose solution gives you the answer to this problem. Draw the feasible region, and find the solution using Simplex.

(b) Suppose instead that the profit per brownie batch is P dollars and the profit per cookie batch remains at 30 dollars. For each vertex you listed in the previous part, give the range of P values for which that vertex is the optimal solution.

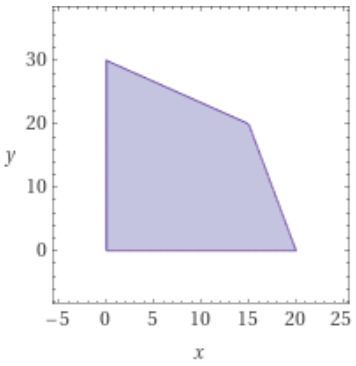
Solution:

(a) x = number of brownie batches
y = number of cookie batches

Maximize: $60x + 30y$

Linear Constraints:
$4x + y \leq 80$
$2x + 3y \leq 90$
$x \geq 0$
$y \geq 0$

The feasible region:



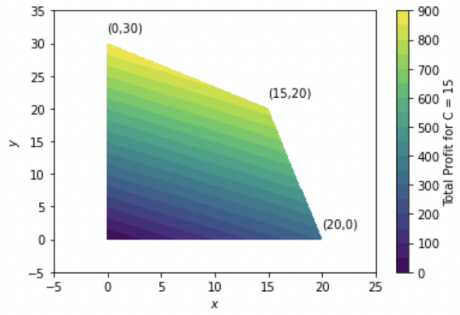
The vertices are $(x = 20, y = 0), (x = 15, y = 20), (x = 0, y = 30)$, and the objective is maximized at $(x = 15, y = 20)$, where $60x + 30y = 1500$. In other words, we can bake 15 batches of brownies and 20 batches of cookies to yield a maximum profit of 1500 dollars.

(b) There are lots of ways to solve this part. The most straightforward is to write and solve a system of inequalities checking when the objective of one vertex is at least as large as the objective of the other vertices. For example, for $(x = 15, y = 20)$ the system of inequalities would be $P \cdot 15 + 30 \cdot 20 \geq P \cdot 20$ and $P \cdot 15 + 30 \cdot 20 \geq 30 \cdot 30$. Doing this for each vertex gives the following solution:

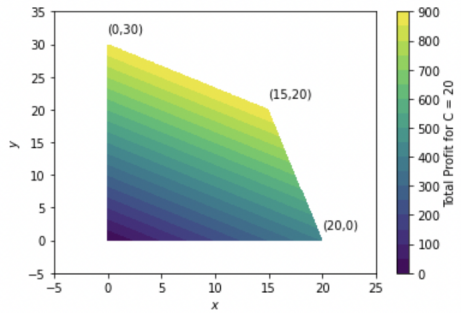
$(x = 0, y = 30) : P \leq 20$
$(x = 15, y = 20) : 20 \leq P \leq 120$
$(x = 20, y = 0) : 120 \leq P$

One should note that there is a nice geometric interpretation for this solution: Looking at the graph of the feasible region, as P increases, the vector $(P, 30)$ starts pointing closer to the x-axis. The objective says to find the point furthest in the direction of this vector, so the optimal solution also moves closer to the x-axis as P increases. When $P = 20$ or $P = 120$, the vector $(P, 30)$ is perpendicular to one of the constraints, and there are multiple optimal solutions all lying on that constraint, which are all equally far in the direction $(P, 30)$.

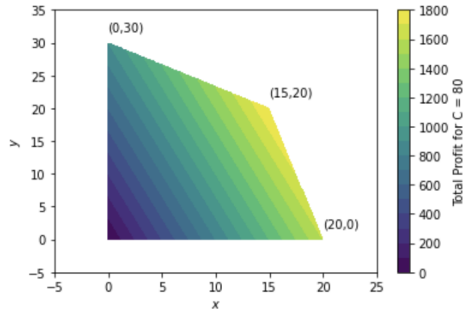
Below are some graphics to help build up your intuition about this problem.
P = 15 : (x = 0, y = 30) optimal



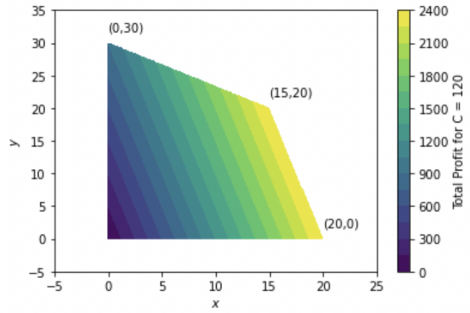
P = 20 : (x = 0, y = 30) and (x = 15, y = 20) optimal



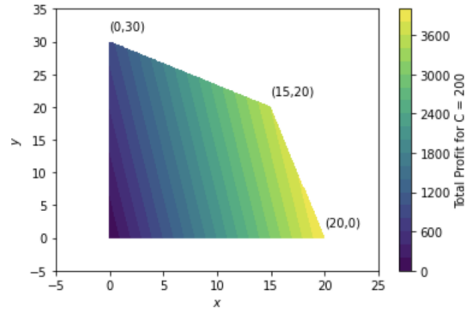
P = 80 : (x = 0, y = 30) optimal



P = 120: (x = 15, y = 20) and (x = 20, y = 0) optimal



P = 200 : (x = 20, y = 0) optimal



6 Simply Simplex

Consider the following linear program.

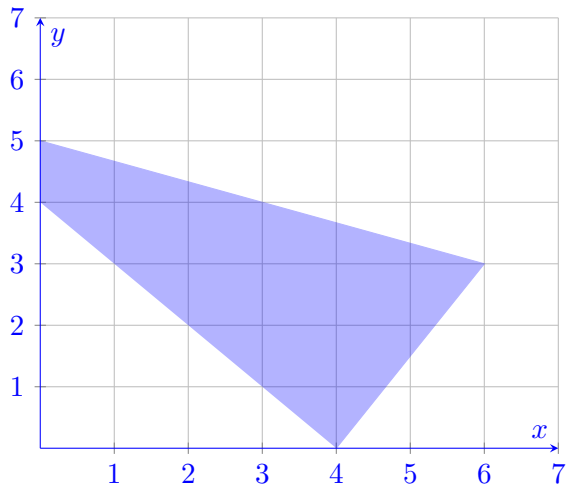
$$\max\ 5x + 4y$$
$$\text{subject to } \begin{cases} x + 3y \leq 15 \\ 3x - 2y \leq 12 \\ 4x + 4y \geq 16 \\ x \geq 0, y \geq 0 \end{cases}$$

(a) Sketch the feasible region. Make sure to clearly label your axes and vertices.

(b) Run the Simplex algorithm on this LP starting at $(0, 4)$. What are the vertices visited? **Please show your work.**

Solution:

(a) We sketch the feasible region below:



(b) The vertices of the feasible region are $(0,4), (0,5), (6,3), (4,0)$. Depending on the starting vertex, Simplex will visit the vertices in a greedy fashion that minimizes the objective $5x+4y$. For instance, suppose we started at $(0,4)$. Here are the steps that Simplex would take:

(a) We note that the objective value at $(0,4)$ is $5(0)+4(4)=16$. Its neighbors are $(0,5)$ and $(4,0)$, with objective values $5(0)+4(5)=20$ and $5(4)+4(0)=20$. Since we're trying to maximize the objective function, we know that we should go to one of the neighbors because their objective values are higher. Generally we go to the neighbor with the higher objective value, but in this it doesn't really matter since they have the same value. Let's say we go to $(0,5)$.

(b) From before, we know that the objective value of $(0,5)$ is 20. Now, we compute its neighbors' objective values: we already know that the objective value of $(0,4)$ is 16, so now just need to find the objective value of $(6,3)$, which is $5(6)+4(3)=42$. We see that $42>20$, and so we move to $(6,3)$.

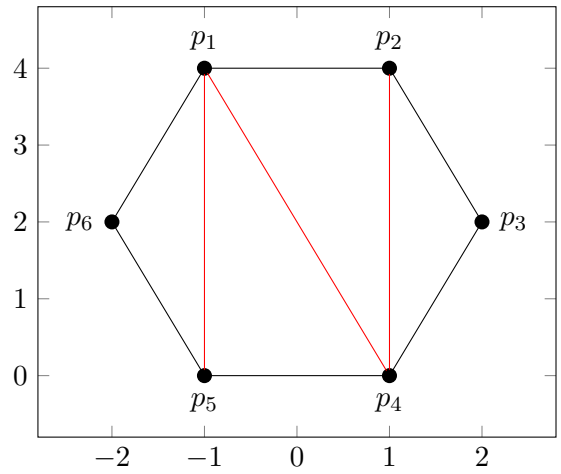
(c) The objective value of $(6,3)$ is 42, which is higher than the objective values of any of its neighbors (the value of $(0,5)$ is 20 and the value of $(4,0)$ is also 20). Thus, we have found the optimum point in the LP!

Throughout this run of Simplex, we visited $(0,4)$, $(0,5)$, and $(6,3)$ in that order.

7 (OPTIONAL) Triangulating a Polygon

You are given a convex polygon P that has n vertices $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$.. We define a triangulation of P to be a collection T of $n-3$ diagonals from P such that no two diagonals intersect inside the polygon. A triangulation splits the polygon's interior into $n-2$ disjoint triangles.

For example, suppose we have a hexagon $P = [p_1 = (-1, 4), p_2 = (1, 4), p_3 = (2, 2), p_4 = (1, 0), p_5 = (-1, 0), p_6 = (-2, 2)]$. One possible triangulation T is $\{(p_1, p_4), (p_1, p_5), (p_2, p_4)\}$, denoted in red in the depiction below:



Now, we define the cost of a triangulation to be the sum of the square lengths of the diagonals forming the triangulation:

$$\text{cost}(T) := \sum_{(p_i, p_j) \in T} d(p_i, p_j)^2$$

where the length of a diagonal connecting $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ is $d(p_i, p_j) = \sqrt{|x_i - x_j|^2 + |y_i - y_j|^2}$, the Euclidean distance between the two points.

Your task is to devise a dynamic programming algorithm to compute the minimum cost of any triangulation of a given polygon P, i.e. $\min_T\{\text{cost}(T)\}$. **Please provide a 3-part solution.**

Hint: A greedy approach will not work here. If you are stuck, reading about Chain Matrix Multiplication in DPV section 6.5 may help guide your thinking.

Solution: We will denote the points as $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ and assume that the points are arranged in a clock-wise manner. Furthermore, we will denote the distance between p_i and p_j by $d(p_i, p_j)$.

Main Idea:

We will index our sub-problems, $A(i, j)$, with pairs of indices i, j such that $1 \leq i < j \leq n$. The subproblem, $A(i, j)$, will denote the minimum cost triangulation of the polygon formed by the vertices $(x_i, y_i) \ldots (x_j, y_j)$. Our recursive call will be as follows:

$$A(i,j) = \begin{cases} \min(\min_{k\in\{i+2,\ldots,j-2\}}(A(i,k)+A(k,j)+d(p_i,p_k)^2+d(p_k,p_j)^2), \\ \qquad d(p_j,p_{i+1})^2+A(i+1,j), d(p_i,p_{j-1})^2+A(i,j-1)), & \text{if } j-i>2 \\ 0, & \text{otherwise} \end{cases}$$

At the end of the algorithm, we will output the value $A(1,n)$.

Runtime Analysis:

Notice, that the cost to compute each sub-problem is $O(n)$ and each sub-problem needs to be computed exactly once. This means we can bound the total runtime by $O(n^3)$ as there are $\binom{n}{2}$ choices for i and j.

Proof of Correctness:

We will prove the correctness of our algorithm by induction on the value of $j-i$.

Base Case: $j-i \leq 2$ In the base case, we have three or fewer vertices in our polygon. This implies our polygon is already triangulated.

Inductive Step: Suppose now that we wish to find a triangulation of the polygon, P_{ij}, formed by the vertices $p_i,\ldots,p_j$. Now consider the edge (j,i). In any triangulation of P_{ij}, the edge (j,i) has to be part of some triangle. The third vertex of the triangle is one of the vertices, $p_{i+1},\ldots,p_{j-1}$. The cases where the third vertex of the triangle is one of the vertices p_{i+1} or p_{j-1} is covered by the last two terms in the recursive formula as in both cases we add either the edge $(j,i+1)$ or $(i,j-1)$. For all the other possibilities (where $k\in\{i+2,\ldots,j-2\}$), we add both edges (i,k) and (j,k) which is covered by the first term of the recursion. In particular, for the optimal triangulation of P_{ij}, the edge (j,i) is part of some triangle, say i,j,k^*. We see that this triangulation consists of a triangulation of P_{ik^*} and P_{k^*j} along with the edges (i,k^*) and (k^*,j). We see that the cost computed by our algorithm is at most the cost of this triangulation. The correctness of our algorithm is evident from the fact that we may construct a triangulation from the choice of k minimizing the recursive formula.

