CS 170 Homework 8

Due **3/18/2024, 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, you must explicitly write "none".

2 Faster Longest Increasing Subsequence

Recall the dynamic programming algorithm for LIS from lecture. It has the recurrence,

$$L[i] = \max_{j<i: A[j]<A[i]} L[j] + 1,$$

where $L[i]$ is the length of the longest increasing subsequence that includes and ends at $A[i]$. Using DP to compute all the $L[i]$'s takes $O(N^2)$ time, where N is the length of the array A. In this problem, we will see how to reformulate the problem so that we can use binary search to obtain a $O(N \log N)$ time algorithm.

Consider the following subproblem definition:

> $M_i[j]$ = the smallest element that ends any subsequence of length j for $A[1 \dots i]$.

where M_i is 1-indexed.

We can set $M_i[k] = \infty$ if no increasing subsequence of length k exists in $A[1 \dots i]$.

(a) Given the following array of length 10, compute the values of M_8. Recall that M_8 only considers the elements $A[1 \dots 8]$. What is the length of the LIS of $A[1 \dots 8]$, and what is the last element of the LIS?

5	3	7	4	1	2	5	7	8	3

Solution: The table M_8 looks as follows:

1	2	5	7	∞	∞	∞	∞	∞	∞

The LIS of $A[1 \dots 8]$ has length 4, ending with element 7.

Note that there is no length-5 increasing subsequence because we only consider elements in $A[:8]$.

(b) Show that M_i is a strictly increasing array, i.e. that $M_i[j] < M_i[j+1]$ for all $j = 1, \dots, N-1$.

Hint: Suppose there exists some j such that $M_i[j] \geq M_i[j+1]$, and show that this implies a contradiction.

Solution: Suppose there exists a j such that $M_i[j] \geq M_i[j+1]$. By definition of M_i, $M_i[j+1]$ is the smallest element that ends any length $j+1$ increasing subsequence. Thus, by definition of increasing subsequence, there must exist an element $A[k] < M_i[j+1]$ which ends a length j increasing subsequence. By definition of M_i, we get $M_i[j] \leq A[k] < M_i[j+1]$ which is a contradiction.

(c) Given the same array as part (a), compute M_{10}.

Solution:

1	2	3	7	8	∞	∞	∞	∞	∞

(d) Let j be the smallest index such that $M_i[j] \geq A[i+1]$. Prove that the length of the LIS ending on $A[i+1]$ is j.

Hint: Use the result from (b) to show that there exists a length j increasing subsequence ending on $A[i+1]$, and that there exist no longer increasing subsequence.

Solution: By the definition of j and since M_i is monotonically increasing, we know that $A[i+1] > M_i[j-1]$. Thus, we can form a length j increasing subsequence ending at $A[i+1]$ by appending $A[i+1]$ to the LIS ending at $M_i[j-1]$. Also by the definition of j and since M_i is monotonically increasing, all elements of $M_i[j+1:]$ are greater than $A[i+1]$, so $A[i+1]$ cannot end a subsequence longer than length j. Thus, there must exist a length j LIS ending at $A[i+1]$.

(e) Now show that only one element differs between M_i and M_{i+1}. Recall that M_i only accounts for $A[1 \ldots i]$, so we are trying to prove M_{i+1} can be computed for $A[1 \ldots i+1]$ by taking M_i and modifying one element.

Solution: From the previous subpart, we know that we should update $M_{i+1}[j] = A[i+1]$. For all other elements $k \neq j$, set $M_{i+1}[k] = M_i[k]$. For $k < j$, we have $A[i+1] > M_i[k]$ so $A[i+1]$ is not the smallest element that can end a length k subsequence. For $k > j$, we have $A[i+1] < M_i[k]$, so $A[i+1]$ cannot end those sequences.

(f) Now combining the previous subparts, write pseudocode that finds the longest increasing subsequence of an array A in $O(N \log N)$ time.

Hint: a naive implementation using the 2D subproblem $M_i[j]$ would still yield a runtime of $O(N^2)$. To achieve the $O(N \log N)$ runtime, you only need to store a single 1D array M. Then, efficiently update M by using previous subparts.

Solution:

```
M = length n array initialized to infinity
for i in range(len(A)):
    j = binary search the smallest element in M greater than A[i]
    M[j] = A[i]
```

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.

(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, to convert it into 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 $\text{left}(v)$ and $\text{right}(v)$ 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.

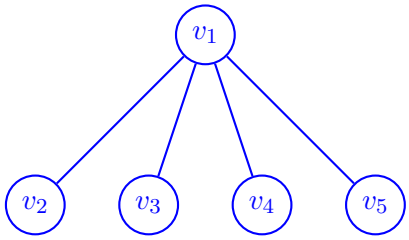
Runtime Analysis: for each $b \in \{0, 1\}$, there are nk subproblems because v can taken on n values and j can take on k values. Each subproblem takes $O(k)$ time because we have to take the max over all $j \in [i]$ or $j \in [i-1]$. Thus, the overall runtime is

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

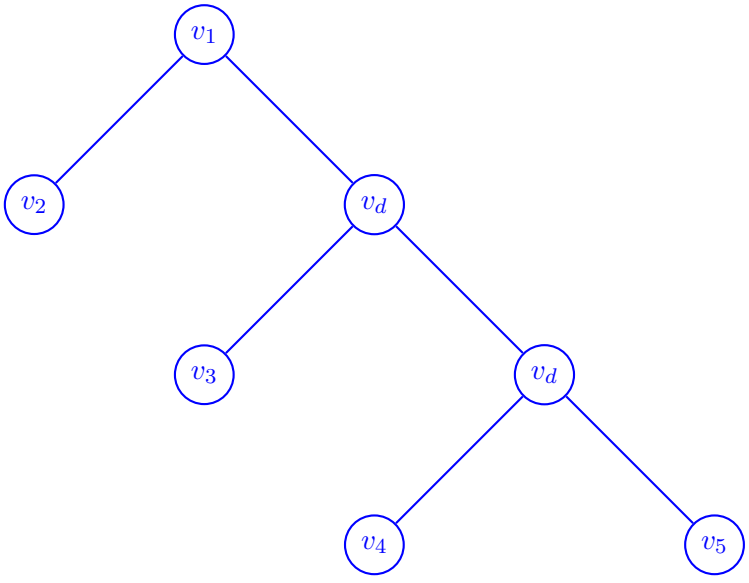
as desired.

(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:

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

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

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

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.

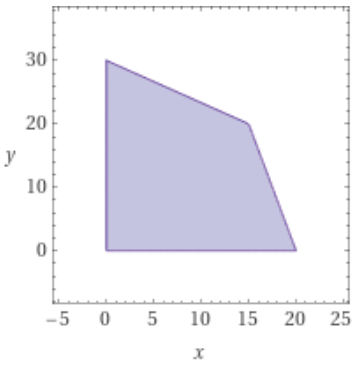
Solution:

(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:



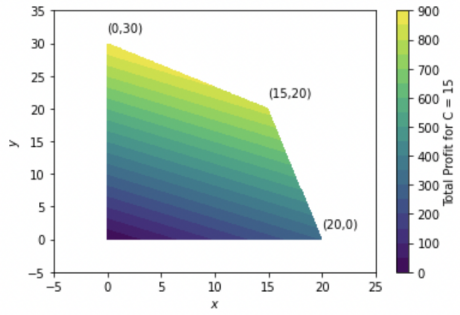
The vertices are $(x = 20, y = 0), (x = 15, y = 20), (x = 0, y = 30)$, and the objective is maximized at $(x = 15, y = 20)$, where $60x + 30y = 1500$. In other words, we can bake 15 batches of brownies and 20 batches of cookies to yield a maximum profit of 1500 dollars.

(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:

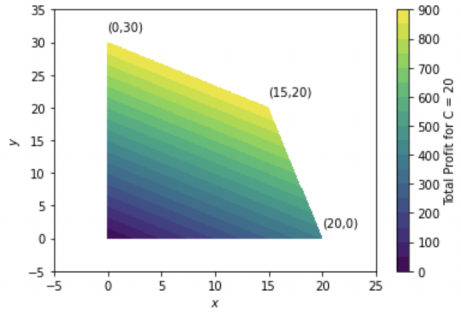
$(x = 0, y = 30) : P \leq 20$
$(x = 15, y = 20) : 20 \leq P \leq 120$
$(x = 20, y = 0) : 120 \leq P$

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 $(P, 30)$.

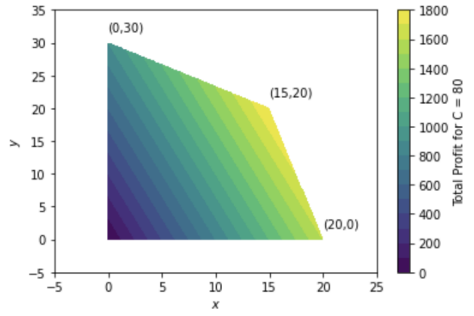
Below are some graphics to help build up your intuition about this problem.
P = 15 : (x = 0, y = 30) optimal



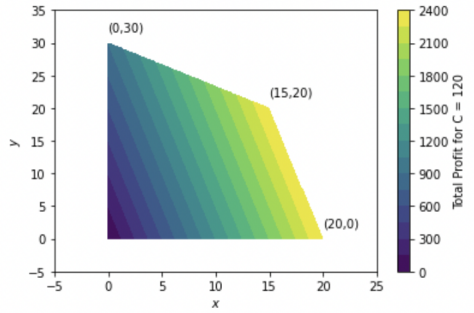
P = 20 : (x = 0, y = 30) and (x = 15, y = 20) optimal



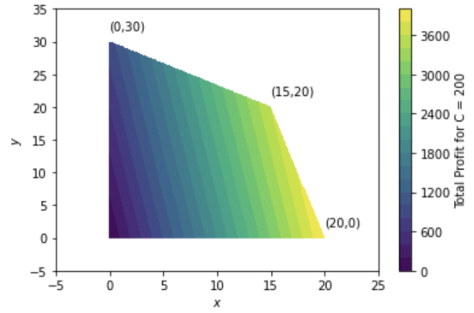
P = 80 : (x = 0, y = 30) optimal



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



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



6 [Coding] Traveling Salesperson DP

For this week's coding questions, we'll implement Dynamic Programming algorithm for the **Traveling Salesperson** problem you saw in lecture. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw08` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-sp24-coding`

   and navigate to the `hw08` folder. Refer to the `README.md` for local setup instructions.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 8 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.