CS 170 Homework 8

Due **10/25/2023, 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 Egg Drop Revisited

Recall the Egg Drop problem from Homework 7:

You are given m identical eggs and an n story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor ℓ or lower, and always breaks if dropped from floor $\ell + 1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more. However, if an egg does not break, you can reuse it.

Let $f(n, m)$ be the minimum number of egg drops that are needed to find ℓ (regardless of the value of ℓ).

Instead of solving for $f(n, m)$ directly, we define a new subproblem $M(x, m)$ to be the maximum number of floors for which we can always find ℓ in at most x drops using m eggs.

For example, $M(2, 2) = 3$ because a 3-story building is the tallest building such that we can always find ℓ in at most 2 egg drops using 2 eggs.

(a) Find a recurrence relation for $M(x, m)$ that can be computed in constant time given the previous subproblems. Briefly justify your recurrence.

Hint: As a starting point, what is the highest floor that we can drop the first egg from and still be guaranteed to solve the problem with the remaining $x - 1$ drops and $m - 1$ eggs if the egg breaks?

(b) Give an algorithm to compute $M(x, m)$ given x and m and analyze its runtime.

(c) Modify your algorithm from (b) to compute $f(n, m)$ given n and m.

Hint: If we can find ℓ when there are more than n floors, we can also find ℓ when there are n floors.

(d) Show that the runtime of the algorithm from part (c) is $O(nm)$. Compare this to the runtime you found in last week's homework.

(e) Show that we can implement the algorithm from part (c) to use only $O(m)$ space.

(f) Suppose that we are given a special machine that is able to "revive" an egg after the first time it breaks so that it is reusable again. In other words, each egg has 2 lives. Based on this modification to the problem, write a new recurrence relation for M.

Hint: you may need to add an additional parameter to your subproblem.

Solution:

(a)

$$M(x, m) = M(x-1, m-1) + M(x-1, m) + 1$$

We will first deduce i, the optimal floor from which we will drop the first egg given we have x drops and m eggs. If the egg breaks after being dropped from floor i, we have reduced the problem to floors 0 through $i-1$, and the strategy can solve this subproblem using $x-1$ drops and $m-1$ eggs. The optimal strategy for $x-1$ drops and $m-1$ eggs can distinguish between $M(x-1, m-1)+1$ floors, so we should choose $i = M(x-1, m-1)+1$.

On the other hand, if the egg doesn't break, we reduce the problem to floors i to n with $x-1$ drops and m eggs. The maximum number of floors we can distinguish using this many drops and eggs is $M(x-1, m)+1$, so we can solve this subproblem for n as large as $i + M(x-1, m) = M(x-1, m-1) + M(x-1, m) + 1$.

(b) For base cases, we'll take $M(0, m) = 0$ for any m and $M(x, 0) = 0$ for any x. Starting with $x = 1$, we compute $M(x, m)$ for all, $1 \leq x \leq m$, and do so again for increasing values of x, up until we compute $M(x, m)$ for all $1 \leq x \leq m$. We return $M(x, m)$.

We compute xm subproblems, each of which takes constant time, so the overall runtime if $\Theta(xm)$.

(c) Again, starting with $x = 1$, we compute $M(x, m)$ for all, $1 \leq x \leq m$, and do so for increasing values of x. This time, we stop the first time we find that $M(x, m) \geq n$, and return this value of x.

(d) Because there are only n floors, the optimal number of drops, x will always be at most n. From part (b), we know the runtime is $\Theta(xm)$, so if $x \leq n$, we know the runtime must be $O(nm)$.

(This is a very loose bound, but it is still much better than the naive $O(n^2 m)$-time algorithm we'd get from using the recurrence on $f(n, m)$ directly.)

(e) While we're computing $M(x, m)$ for all $1 \leq x \leq m$, we only need to store $M(x-1, m)$ and $M(x, m)$ for all x, i.e. we only ever need to store $O(m)$ values. In particular, after computing $M(x, m)$ for all x, we can delete our stored values of $M(x-1, m)$.

(f) Let $M_1(x, m)$ be $M(x, m)$ when the mth egg has 1 life left, and $M_2(x, m)$ be $M(x, m)$ when the mth egg has 2 lives left. Then, our recurrence relation looks as follows:

$$M_1(x, m) = M_2(x-1, m-1) + M_1(x-1, m) + 1$$
$$M_2(x, m) = M_1(x-1, m) + M_2(x-1, m) + 1$$

Alternate Solution: note that this modification effectively doubles the number of eggs we have. Thus, we don't actually have to modify the recurrence relation at all, and simply return $M(x, 2m)$ as the answer.

3 Knightmare

Give a dynamic programming algorithm to find the number of ways you can place knights on an L by H ($L < H$) 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 $O(2^{3L}LH)$, and return your answer mod 1337.

Hint: if a knight is on row i, what rows on the chessboard can it affect?

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 L 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 $(h-1) \times L$ chessboard and use it to solve the $h \times L$ case. Note that as we iteratively incrementing h, a knight in the h-th rows can only affect configurations of rows $h+1$ and $h+2$. So we can denote $K(h, u, v)$ as the number of possible configurations of the first h rows with u being the $(h-1)$-th row and v being the h-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 $K(h, v, w)$ we have:

$$K(h, v, w) = \sum_{u : u,v,w \text{ are valid}} K(h-1, u, v) \bmod 1337$$

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 $h > 2$, the above recurrence is correct because for any valid h-row configuration ending in v, w, the first $h-1$ 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 h-row and $(h-1)$-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 $O(L)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3L}L)$ which gives us an overall runtime of $O(2^{3L}LH)$ operations. Although there are a potentially exponential number of possible configurations (at most 2^{HL}), we are only interested in the answer mod 1337, 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 $O(2^{3L}LH)$.

4-part solution:

(a) **Subproblems:** $f(h, u, v)$ is the number of possible valid configurations mod 1337 of the first h rows with u being the configuration of the $(h-1)$-th row and v being the configuration of the h-th row.

(b) **Recurrence and Base Cases:** For $h > 2$, $f(h, v, w) = \sum_{u:u,v,w \text{ are valid}} f(h-1, u, v)$ mod 1337. 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 $O(2^{3L}LH)$. For space complexity, note that there are only $O(H2^{2L})$ subproblems (H rows, 2^L settings to the $(H-1)$th row, and 2^L settings to the Hth row). Each subproblem is a number of possible configurations mod 1337, bounded above by 1337, so our total space is $O(2^{2L}H)$. Note that since each subproblem only depends on subproblems of the previous row, we could reduce this by a factor of H to $O(2^{2L})$ by recycling space from earlier rows.

4 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 $A[v]$. 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.

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

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 to convert it into a binary tree T_b.

(c) Using your responses to parts (a) and (b), 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.

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:

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

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

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 $f_0[\ell][j] = 0, f_1[\ell][j] = A[l]$ 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.

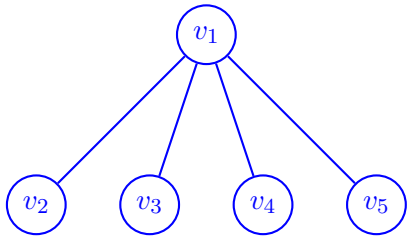
Runtime Analysis: for each $b \in \{0, 1\}$, there are nk subproblems because v can taken on n values and j can take on k values. Each subproblem takes $O(k)$ time because we have to take the max over all $j \in [i]$ or $j \in [i-1]$. Thus, the overall runtime is

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

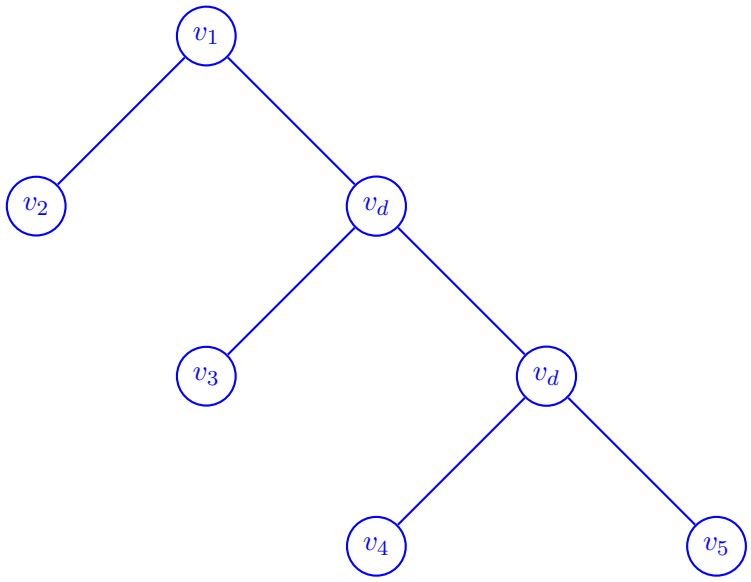
as desired.

(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 $A[v_i] = i$:



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 not 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:

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

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

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

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,

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

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

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

5 Coding Questions

For this week's coding questions, we'll implement dynamic programming algorithms to solve two classic problems: **Weighted Independent Set in a Tree** and **TSP**. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   https://github.com/Berkeley-CS170/cs170-fa23-coding

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

2. **On Datahub**: Click here and navigate to the `hw08` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 8 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Edstem Instructions:* Conceptual questions are always welcome on the public thread. If you need debugging help first try asking on the public 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 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.

1 Dynamic Programming (Part 2)

In this notebook, we'll implemement dynamic programming algorithms for some more problems: Weighted Independent Set in a Tree and TSP.

If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

If you're running locally:

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

In [19]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [1]:
```
import otter
assert (otter.__version__ >= "4.4.1"), "Please reinstall the requirements and restart your kern

grader = otter.Notebook("dp2.ipynb")
import pickle
import numpy.random as random
import random
import tqdm
import time
from autograder_utils import is_independent_set, validate_tour, handle_timeout

with open('public_data.pkl', 'rb') as f:
    test_data = pickle.load(f)

rng_seed = 42
```

1.1 Q1. Weighted Independent Set in a Tree

In class, we saw an algorithm to solve the maximum independent set on trees. For a quick refresher, you can consult https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=21.

In this problem, we'll add some twists: now, suppose that each vertex i has some weight $w[i]$, and we want to find the independent set with the maximum total weight. **Also, you must implement your algorithm with a top-down approach.**

As before, the algorithm discussed in lecture and the textbook only returns the size of the maximum independent set. However, here we want you to return the list of vertices in the acual independent set. To find the actual independent set, it may be useful to maintain an array seperate from the DP array which can help us backtrack to reconstruct the actual set (simply storing the entire set so far at each entry in the DP table is too expensive and will cause the autograder to fail).

Hint: To compute a node's children and grandchildren, feel free to re-use your code from previous homeworks.

```
In [28]: def max_independent_set(adjacency_list, weights):
             """
             Return a list containing the vertices in the maximum weighted independent set.

             args:
                 adjacency_list:ListList[[int]] = the
                 weights:List[int] = a list of vertex weights. weights[i] is the weight of vertex i.

             return:
                 List[int] Containing the labels of the vertices in the maximum weighted independent se
             """
             # BEGIN SOLUTION
             n = len(weights)
             if n == 0:
                 return []

             # DFS post number computation (modified from HW4)
             def get_post(adj_list):
                 time, post, visited = 1, {}, set()
                 unvisited = lambda x: x not in visited
                 def explore(u):
                     nonlocal time
                     visited.add(u)
                     time += 1
                     for v in filter(unvisited, adj_list[u]):
                         explore(v)
                     post[u] = time
                     time += 1
                 for i in filter(unvisited, range(len(adj_list))):
                     explore(i)
                 return post

             post = get_post(adjacency_list)

             # helper function to get the children of node u based on DFS post numbers
             def get_children(u):
                 yield from filter(lambda v: post[v] < post[u], adjacency_list[u])

             def get_grandchildren(u):
```

```
    for v in get_children(u):
        yield from get_children(v)

# initialize the DP array
dp = [None] * n

def independent_set_helper(u):
    if dp[u] is None:
        # dp[u] includes the weight of the subtree as seen in class, as well as a flag
        # indicating whether u is included in the independent set. This will be useful
        # for reconstructing the set as it will save us from recomputing this quantity
        # on the fly.
        choices = (
            (weights[u] + sum(independent_set_helper(v)[0] for v in get_grandchildren(u)),
            (sum(independent_set_helper(v)[0] for v in get_children(u)), False),
        )
        dp[u] = max(choices)
    return dp[u]

# compute the DP values
# root doesn't really matter, we choose 0 for ease
independent_set_helper(0)

# reconstruct the sequence starting from the root, checking whether each node is included,
# and adding it to the independent set if it is, and recursing on its children/grandchildr
independent_set = []
to_check = [0]

while to_check:
    u = to_check.pop()
    if dp[u][1]:
        # u is included in the independent set, so its children are not.
        # add u to the independent set and check its grandchildren
        independent_set.append(u)
        to_check.extend(get_grandchildren(u))
    else:
        # u is not included in the independent set, so check its children.
        to_check.extend(get_children(u))

return independent_set
# END SOLUTION
```

1.1.1 Debugging

A simplified verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

Note: your solution should not take inordinate amounts of time to run. Our implementation runs on all local test cases in <1s, and your implementation should run in similar time.

Points: 3

```
In [33]: for adj_list, weights, expected_weight in tqdm.tqdm(test_data['q1.1']):
             try:
                 result = max_independent_set(adj_list, weights)
             except Exception as e:
                 raise Exception(f"Your function threw an error (see above)") from e
             assert is_independent_set(adj_list, result), f"Your output is not an independent set"
             assert sum(weights[i] for i in result) == expected_weight, f"Your output is not a maximum
```

```
100%|      | 100/100 [00:00<00:00, 4684.28it/s]
```

```
In [ ]: grader.check("independent set")
```

1.2 Traveling Salesperson DP

Now, we'll implement the dynamic programming algorithm for the traveling salesperson problem (TSP). A brute force solution will be hopelessly slow even for moderate-sized test cases, but we can use dynamic programming to get a solution in slightly more reasonable (but still exponential) time. For a refresher on the TSP algorithm, see Lecture 14 or https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=20.

As with previous problems, we want you to return the actual tour, not the cost of the tour. We can once again apply the same procedure of backtracking through our subproblem array to reconstruct this tour.

A note on implementation: Since our subproblem definition takes in a set as one of its parameters, we can't just use a 2D array to store our subproblems. Instead, we recommend storing subproblems in a dictionary, where the keys are tuples of the form `(S, i)`, where `i` represents the last city visited before returning home and `S` is the set of cities visited so far.

To ensure that the keys are hashable, we recommend using Python's built-in `frozenset` class for `S`. `frozenset` is built-in to Python so you can use it without any additional imports, and works just like a normal set, except that it is immutable and hashable. You can read more about `frozenset` here: https://docs.python.org/3/library/stdtypes.html#frozenset.

An alternative approach would be to use a 2D array as usual, but use a bitmask to represent the set of visited cities. In this approach, `S` would be represented as an n-bit unsigned integer, and the i-th bit of `S` would be set to 1 if and only if the i-th city is part of the set of visited cities. Then, since `S` is an integer, we can use it to index into our 2D array.

As with before, storing the entire tour at each step is too memory-intensive and will cause the autograder to fail. Instead, consider maintaining a separate dictionary or array which stores a smaller amount of information but can still help you reconstruct the tour.

In [2]:
```python
def tsp_dp(dist_arr):
    """Compute the exact solution to the TSP using dynamic programming and returns the optimal

    Args:
        dist_arr (ndarray[int]]): An n x n matrix of distances between cities. dist_arr[i][j] i

    Returns:
        List[int]: A list of city indices representing the optimal path.
    """
    # BEGIN SOLUTION
    n = len(dist_arr)
    dp = {}
    prev = {}

    def tsp_helper(S, i):
        if (S, i) in dp:
            return dp[(S, i)]

        if S == frozenset():
            return dist_arr[0][i]  # Return to the starting city

        min_cost = float('inf')
        prev_city = None
        for city in S:
            cost = dist_arr[i][city] + tsp_helper(S.difference({city}), city)
            min_cost, prev_city = min((min_cost, prev_city) , (cost, city))

        dp[(S, i)] = min_cost
        prev[(S, i)] = prev_city
        return min_cost

    best_distance = tsp_helper(frozenset(range(1, n)), 0)

    # Backtracking to reconstruct tour
    S = frozenset(range(1, n))
    city = 0  # start at the origin
    tour = [0]

    # print(prev)

    while S:
        city = prev[(S, city)]
        tour.append(city)
        S = S.difference({city})

    return tour
    # END SOLUTION
```

1.2.1 Debugging

A simplified verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

Points: 6

```
In [63]: # tests on very small cases
         for dist_arr, expected_distance in tqdm.tqdm(test_data['q2.1']):
             try:
                 result = tsp_dp(dist_arr)
             except Exception as e:
                 raise Exception(f"Your function threw an error (see above)") from e
             assert set(result) == set(range(len(dist_arr))), f"Your output does not visit all cities"
             student_length = validate_tour(result, dist_arr)
             assert student_length >= 0, f"Your output is not a valid tour"
             assert student_length == expected_distance, f"Your output is not a minimum distance tour"
```

```
100%|      | 20/20 [00:10<00:00,  1.94it/s]
```

```
In [ ]: # tests on slightly larger cases (these might take a while to run)
        for dist_arr, expected_distance in tqdm.tqdm(test_data['q2.2']):
            try:
                result = tsp_dp(dist_arr)
            except Exception as e:
                raise Exception(f"Your function threw an error (see above)") from e
            assert set(result) == set(range(len(dist_arr))), f"Your output does not visit all cities"
            student_length = validate_tour(result, dist_arr)
            assert student_length >= 0, f"Your output is not a valid tour"
            assert student_length == expected_distance, f"Your output is not a minimum distance tour"
```

```
 96%|      | 48/50 [01:03<00:02,  1.33s/it]
```

```
In [ ]: grader.check("TSP")
```

1.3 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

In [ ]:
```
grader.export(pdf=False, force_save=True)
```