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

Prim's and Set Cover Cheatsheet

Prim's Algorithm

General Idea:

```
def prims(G=(V, E), s):
    T = []
    added = set()

    while added != V:
        (u, v) = "lightest edge across the cut formed by added and (V / added)"
        T.append((u, v))
        added.add(v)

    return T
```

Optimized Implementation

```
def prims(G=(V, E), s):
    h = heap()
    for each v in V:
        h.insert(v, inf) # each vertex v has a priority value, initialize at inf
        prev(v) = v

    h.decrease_key(s, 0)
    while h is not empty:
        v = h.delete_min()
        for each edge (v, u) in E:
            if h.get_priority(u) > w(v, u): # minimize edge weight to u
                prev(u) = v
                h.decrease_key(u, w(v, u))

    T = []
    visited = set()
    for each v in V:
        curr = v
        while curr != prev(curr):
            if visited(curr):
                break
            T.append((curr, prev(curr))
            curr = prev(curr)

    return T
```

↪ **Runtime of Prim's:**

- $O((|E| + |V|) \log |V|)$ using a binary min-heap
- $O(|E| + |V| \log |V|)$ using a fibonacci min-heap

Set Cover

Input:

- A set of elements $U = \{1, 2, \ldots, n\}$ (called the *universe*)
- A collection $\mathcal{S}$ of m subsets $\mathcal{S} = S_1, S_2, \ldots, S_m \subseteq U$ such that $\bigcup_{i=1}^{m} S_i = U$.

Output: a collection of subsets $\mathcal{C} \subseteq \mathcal{S}$ of minimal size such that $\bigcup_{C \in \mathcal{C}} C = U$.

Greedy Algorithm

```python
def greedy_set_cover(U, S):
    C = set()
    covered = set()

    while covered != U: # while we haven't covered all elements
        curr_best_set = "set in S with largest number of uncovered elements"

        C.add(curr_best_set)
        covered = covered.union(curr_best_set)

    return C
```

Greedy Approximate Optimality

For any instance of the Set Cover problem, if the optimal solution uses k sets, the greedy algorithm uses at most $k \log n$ sets.

↪ Proof:

We'll use an inductive-style proof. Let n_t be the number of elements not covered after t steps of the greedy algorithm. In lecture, we showed the following relation

$$n_{t+1} \leq n_t(1 - 1/k), \quad \forall t \geq 0.$$

Why is this true? Suppose you've already chosen t sets, and you're now choosing your $t+1$th set. We know that the optimal solution covers the n_t uncovered elements with $\leq k$ sets, meaning that there exists a set in in the optimal solution that covers at least n_t/k elements. Since the greedy algorithm always chooses the set which covers the most uncovered elements, the $t + 1$th set it chooses must cover at least n_t/k of the n_t uncovered elements. Then, the inequality directly follows.

Thus, we want to find the minimum value of t such that

$$n_t \leq n_{t-1}(1 - 1/k) \leq n_{t-2}(1 - 1/k)^2 \leq \cdots \leq n_0(1 - 1/k)^t = n(1 - 1/k)^t < 1$$

Since $1 + x < e^x$ for any $x \neq 0$, we have that

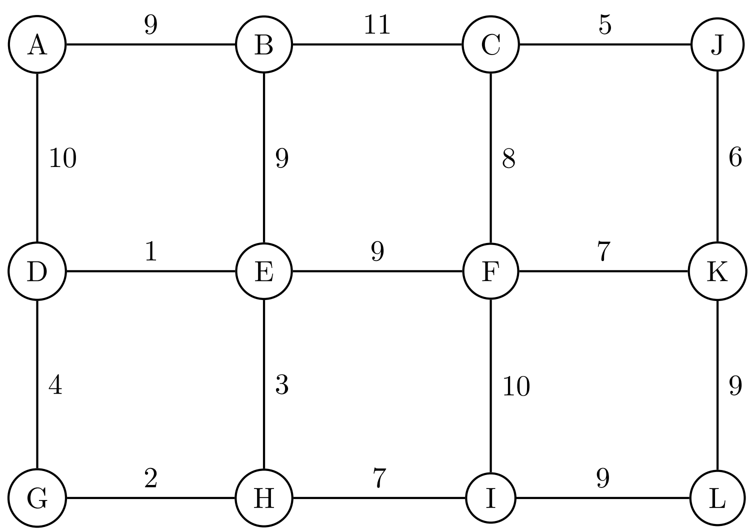
$$n(1 - 1/k)^t < ne^{-t/k} \leq 1$$

$$\implies t = -k \log(1/n) = k \log n$$

Thus, we know that the greedy algorithm outputs $O(k \log n)$ sets.

Fun fact: if you analyze the approximation a bit better, you can argue that the greedy algorithm outputs $O(k(1 + \log(n/k)))$ sets, which is a slightly tighter bound.

1 Prim's Tutorial



1. List the first **six** edges added by Prim's algorithm in the order in which they are added. Assume that Prim's algorithm starts at vertex A and breaks ties lexicographically.

2. Prim's algorithm is very similar to Dijkstra's in that a vertex is processed at each step which minimizes some cost function. These algorithms also produce similar outputs, as Dijkstra's essentially outputs a shortest paths tree. However, the trees they produce aren't optimizing for the same thing.

   To see this, give an example of a graph for which different trees are produced by running Prim's algorithm and Dijkstra's algorithm. In other words, give a graph where there is a shortest path from a start vertex A using at least one edge that doesn't appear in any MST.

2 Worst-case Instances for Greedy Set-Cover

In lecture, we proved that the greedy strategy for finding the minimum set cover over-estimates the optimal number of sets by a factor of at most $O(\log n)$, where $n = |U|$. In this problem we will prove that this bound is tight.

Show that for any integer n that is a power of 2, there is an instance of the set cover problem (i.e. a collection of sets $S_1, \ldots, S_m$) with the following properties:

i. There are n elements in the universe U.

ii. The optimal cover uses just two sets.

iii. The greedy algorithm picks at least $\Omega(\log n)$ sets.

3 Independent Set Approximation

In the Max Independent Set problem, we are given a graph $G = (V, E)$ and asked to find the largest set $V' \subseteq V$ such that no two vertices in V' share an edge in E.

Given an undirected graph $G = (V, E)$ in which each node has degree $\leq d$, give an efficient algorithm that finds an independent set whose size is at least $1/(d+1)$ times that of the largest independent set. Describe your algorithm and prove that the it finds an independent set of size is at least $1/(d+1)$ times the largest possible solution. Your algorithm should run in time $O(|V| \cdot |E|)$ (or less).

4 Dynamic Programming Introduction: Fibonacci Numbers

The Fibonacci sequence is defined by the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

with base cases $F_0 = 0$ and $F_1 = 1$. Back in CS 61A, we learned how to write a program to find the nth fibonacci number, which would look something like:

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

However, this program is actually super slow! In the box below, show that calling `fibo(n)` takes $2^{\Theta(n)}$ time.

Challenge: show that the runtime is $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

If you didn't above, in the box below draw out the recurrence tree produced when calling `fibo(5)`. Do you notice any repeated computations (i.e. nodes)?

In the recurrence tree, we notice that we end up recomputing many of the same values many times. For instance, we end up computing F_1 5 times! To reduce the number of recomputing we have to do, we can **store each fibonacci number in an array after computing it**. This way, we can simply index into that array when we need that value, rather than recomputing it every time we recurse. To implement this, fill out the blank lines in the code below:

```
def optimized_fibo(n):
    stored_fibos = [-1 for _ in range(n+1)]

    def fibo(n):
        # base case

        if __________:

            return __________

        # if we've already computed fibo(n) before, we can reuse it via
            stored_fibos!

        if ____________________:

            return ____________________

        # if we haven't already computed fibo(n), then we need to recurse as before:
        # make sure to store it in stored_fibos so that we can use it in the future!

        ______________________________

        return ____________________
```

What is the runtime of this new algorithm?

Congratulations, you've just implemented your first **dynamic programming (DP)** algorithm! This is essentially all that DP is: recursion plus storing stuff (memoization), so that we don't have to fully solve any subproblems more than once.

Now, there are actually two ways to implement DP algorithms. The implementation that you've completed above uses a **top-down** approach, i.e. you start from the largest subproblem (top) and repeatedly recurse on smaller subproblems (going down). The other implementation method uses a **bottom-up** approach, which starts from the smallest subproblems (i.e. the base cases), and builds up larger subproblems in an iterative manner.

Referencing your previous top-down approach, fill in the blank lines in the code below to complete the bottom-up implementation of the Fibonacci DP algorithm:

```
def fibo_dp_bottom_up(n):
    stored_fibos = [-1 for _ in range(n+1)]

    # define base cases at the "bottom"
    stored_fibos[0] = 0
    stored_fibos[1] = 1

    # build your subproblems "up" from your smaller subproblems
    for i in range(2, n+1):

        ________________________________________

    # what element in stored_fibos represents the nth fibonacci number?

    answer = ____________________
    return answer
```

Yay! You've now learned how to implement both types of DP algorithms.

(Challenge) What is the space complexity of your algorithm? Can you modify it to only use O(1) extra space?

```
def fibo_dp_bottom_up(n):
    # define base cases here

    if __________:

        ____________________

    ____________________

    ____________________

    # build your subproblems "up" from your smaller subproblems

    for i in range(2, n+1):

       ______________________________________

    # where are you storing the nth fibonacci number?

    answer = ____________________
    return answer
```