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
```

$\hookrightarrow$ **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

```
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.

$\hookrightarrow$ 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

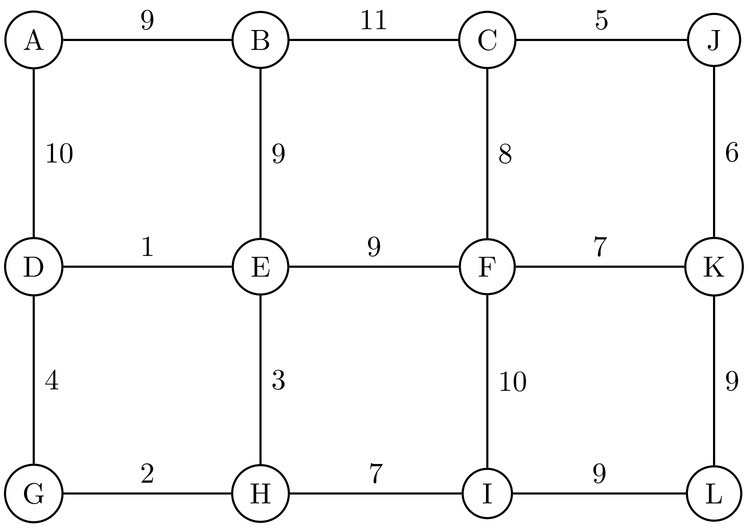
$$n(1 - 1/k)^t < ne^{-t/k} \leq 1$$

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

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

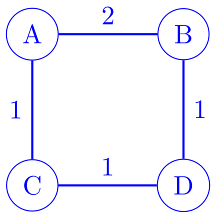


1. List the first **six** edges added by Prim's algorithm in the order in which they are added. Assume that Prim's algorithm starts at vertex A and breaks ties lexicographically.

   Solution: AB, BE, DE, EH, GH, HI

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.



Solution:

The MST is $\{(A, C), (C, D), (B, D)\}$, but the shortest path from A to B is through the weight 2 edge.

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.

Solution: Consider the base set $U = \{1, 2, 3, \ldots, 2^k\}$ for some $k \geq 2$. Let $T_1 = \{1, 3, 5, \ldots, 2^k - 1\}$ and $T_2 = \{2, 4, 6, \ldots, 2^k\}$. These two sets comprise an optimal cover. We add sets $S_1, \ldots, S_{k-1}$ by defining $l_i = 2 + \sum_{j=1}^{i} 2^{k-j}$ (take $l_0 = 0$) and letting $S_i = \{l_{i-1} + 1, \ldots, l_i\}$. We think of S_i as covering a tiny bit more than a $1/2^i$-fraction of the universe, and in particular S_1 is larger than both T_1 and T_2.

Since S_1 contains $2^{k-1}+2$ elements, it will be picked first. After the algorithm has picked $\{S_1, S_2, \ldots, S_i\}$, each of T_1 and T_2 covers $2^{k-i-1} - 1$ new elements while S_{i+1} covers 2^{k-i-1} new elements. Hence, the algorithm picks the cover $S_1, \ldots, S_{k-1}$ containing $k - 1 = \log n - 1$ sets.

An example of the above algorithm is with, say, $n = 64$. Let $T_1 = \{1, 3 \ldots, 63\}$, and $T_2 = \{2, 4, \ldots, 64\}$. Let picking T_1 and T_2 be the optimal answer. Now let $S_1 = \{1, 2, \ldots, 32, 33, 34\}$, or 2 elements more than $64/2$. Let S_2 contain the next $64/4 = 16$ elements, let S_3 contain the next $64/8 = 8$ elements, S_4 contain the next $64/16 = 4$ elements, and S_5 contain the remaining 2 elements. This works because $64 = 32 + 16 + 8 + 4 + 2 + 2$. We are just regrouping the terms as follows $64 = (32 + 2) + 16 + 8 + 4 + 2$ and creating sets out of them.

Our greedy algorithm will pick S_1, S_2, S_3, S_4, S_5 instead of T_1, T_2. Our greedy algorithm ended up picking $k - 1$ sets, since $k = 6$ for $n = 64$. The first two paragraphs of the solutions explain how this is generalized for all k.

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).

Solution: Initially, let G be the original graph and $I = \emptyset$. Repeat the process below until $G = \emptyset$:

1. Pick an arbitrary node v in G and let $I = I \cup \{v\}$.
2. Delete v and all its neighbors from the graph.
3. Let G be the new graph.

Notice that I is an independent set by construction. At each step, I grows by one vertex and we delete at most $d + 1$ vertices from the graph (since v has at most d neighbors). Hence there are at least $|V|/(d + 1)$ iterations. Let K be the size of the maximum independent set. Since $K \leq |V|$, we can use the previous argument to get:

$$|I| \geq \frac{|V|}{d+1} \geq \frac{K}{d+1}$$

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 $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Solution: The runtime recurrence is $T(n) = T(n-1) + T(n-2)$.

We can lower bound it by, $T(n) \geq 2T(n-2)$, so we know $T(n) = \Omega(2^{n/2})$. We can also upper-bound it by $T(n) \leq 2T(n-1)$, which gives $T(n) = O(2^n)$.

Hence, $T(n) = 2^{\Theta(n)}$. However, we can actually compute a more precise runtime! Since we know the runtime is exponential with respect to n, we can write the runtime in the form $T(n) = \Theta(a^n)$. Then, plugging this into the recurrence, we have

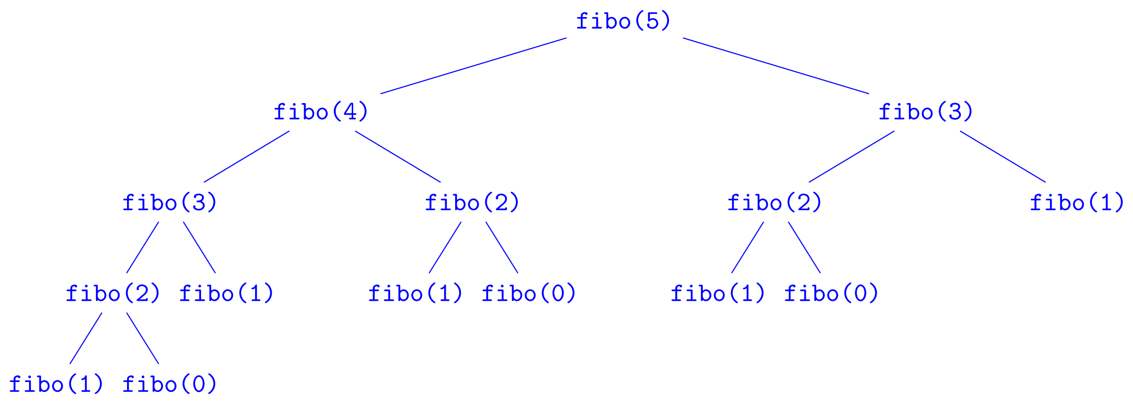
$$a^n = a^{n-1} + a^{n-2}$$
$$a^2 = a + 1$$
$$a^2 - a - 1 = 0$$

where we can divide by a^{n-2} since $a \neq 0$. By the quadratic formula, we get that $a = \frac{1\pm\sqrt{5}}{2}$. Since a must be positive, we conclude that $a = \frac{1+\sqrt{5}}{2}$ and thus

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

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)?

Solution:



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 ____________________
```

Solution:

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

    def fibo(n):
        # base case
        if n <= 1:
            return n

        # if we've already computed fibo(n) before, we can reuse it via
            stored_fibos!
        if stored_fibos[n] != -1:
            return stored_fibos[n]

        # 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!
        stored_fibos[n] = fibo(n-1) + fibo(n-2)
        return stored_fibos[n]
```

What is the runtime of this new algorithm?

Solution: Observe that for each $k \leq n$, the kth Fibonacci number will computed exactly once. All the other times `fib(k)` will return the stored value. Furthermore, `fib(k)` is called at most twice: by `fib(k+1)` and `fib(k+2)`.

Hence the runtime is $O(n)$.

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
```

Solution:

```
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):
        stored_fibos[i] = stored_fibos[i-1] + stored_fibos[i-2]

    # what element in stored_fibos represents the nth fibonacci number?

    answer = stored_fibos[n]
    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
```

Solution: The space complexity of our algorithm was $O(n)$, as we had to create a $n+1$ length array to store all the $F_0, F_1, F_2, \ldots, F_n$ fibonacci numbers.

Note that for any given fibonacci number, it only depends on the previous two fibonacci numbers. So we can actually modify our algorithm to store the last two fibonacci numbers computed at any given point of time (instead of all the fibonacci numbers seen so far). This is implemented below:

```
def fibo_dp_bottom_up(n):
    # define base cases here
    if n <= 1:
        cur_fib = n

    prev_2_fib = 0
    prev_1_fib = 1

    # build your subproblems "up" from your smaller subproblems
    for i in range(2, n+1):
        cur_fib = prev_2_fib + prev_1_fib

        # update prev_2_fib and prev_1_fib for i+1
        prev_2_fib = prev_1_fib
        prev_1_fib = cur_fib

    # where are you storing the nth fibonacci number?
    answer = cur_fib
    return answer
```