Problem 1

> Refs:
>
> [1] https://www.cs.duke.edu/courses/cps102/spring09/Lectures/L-18.pdf

1.

Probability of no collision q = (# of arrangements with no collision) / (# of all possible arrangements)

$$p' = \frac{P_n^{n^2}}{(n^2)^n} = \frac{(n^2)!}{n^{2n} \cdot (n^2 - n)!}$$

Then , the probability of any collision p :

$$p = 1 - p' = 1 - \frac{(n^2)!}{n^{2n} \cdot (n^2 - n)!}$$

2.

> Using P instead of $|P|$ for convenience

Let ϵ = expected # of queries needed and $F(n)$ = # of collisions after n queries.
Then ϵ should satisfy:

$$\epsilon - E(F(\epsilon)) = \frac{P}{4}$$

To calculate $E(F)$, we need $E(G)$, where $G(n)$ = # of empty slots after n queries. By empty slot I mean a hashed value that hasn't occurred yet.
The probability of a slot being empty after n queries is $(1 - \frac{1}{P})^n$, and since we are using a uniform hash, we have:

$$E(G) = P \cdot (1 - \frac{1}{P})^n$$

And # of queries without collision $=$ # of not-empty slot $= P - E(G)$.
Then # of collisions $= n - (P - E(G))$

$$
\begin{aligned}
E(F) &= n - P + E(G) \\
&= n - P + P(1 - \frac{1}{P})^n \\
&= n - P(1 - (1 - \frac{1}{P})^n)
\end{aligned}
$$

Plugin this back to the equation:

$$
\begin{aligned}
\epsilon - (\epsilon - P(1 - (1 - \frac{1}{P})^\epsilon)) &= \frac{P}{4} \\
P(1 - (1 - \frac{1}{P})^\epsilon) &= \frac{P}{4} \\
(1 - \frac{1}{P})^\epsilon &= \frac{3}{4} \\
\epsilon &= \frac{\ln \frac{3}{4}}{\ln (1 - \frac{1}{P})}
\end{aligned}
$$

3.

- Open addressing with linear probing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34		34						18			
9		34						18		9	
37		34			37			18		9	
40		34			37			18	40	9	
32		34			37			18	40	9	32
89		34	89		37			18	40	9	32

- Open addressing with double hashing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34		34						18			
9		34						18		9	
37		34			37			18		9	
40		34			37			18	40	9	
32		34			37			18	40	9	32
89	89	34			37			18	40	9	32

4.

- Table T_1, using $h_1(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							6
31				31			6
2			2	31			6
41			2	31			41
30			30	31			6
45			30	45			6
44			44	31			6

- Table T_2, using $h_2(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							
31							
2							
41	6						
30	2					41	
45	2				31	41	
44	2				30	41	45

Problem 2

> Refs:
>
> [1] http://web.ntnu.edu.tw/~algo/Substring.html#6
> [2] https://www.youtube.com/watch?v=CpZh4eF8QBw

1.

```
function query(l1, l2, n)
    if subs_cmp(l1, l2) ≥ n
        return True
    else
        return False

cmp_history = N*N array filled with -1
function subs_cmp(l1, l2)
    if l2 < l1
        swap(l1, l2)
    if l2 > N
        return 0
    if cmp_history[l1][l2] == -1 /* array has initial value -1 */
        if S[l1] ≠ S[l2]
            cmp_history[l1][l2] = 0
        else
            cmp_history[l1][l2] = 1 + subs_cmp(l1+1, l2+1)
    return cmp_history[l1][l2]
```

Explanation

The function `subs_cmp(l1, l2)` returns `m` such that `m` is the largest number that satisfies `S[l1..l1+m-1] == S[l2..l2+m-1]`. The workflow of it is direct comparison of the two strings with caching.
`query(l1, l2, n)` simply check if `subs_cmp(l1, l2)` is larger than `n`.

Space complexity

`cmp_history` is a `N*N` array storing single values in each slot, therefore the space complexity is $O(N^2)$

Time complexity

Each query takes $O(N)$ time because it's directly comparing two strings. Therefore time complexity for Q queries should be $O(QN)$. Caching should help reducing constants but I am not sure if time complexity could be tighter.

2.

`X = {8, 0, 0, 0, 3, 0, 0, 0}`

3.

```
function generateX(S)
    N = S.len
    X = array(N)
    l = 1 /* left bound of furthest interval */
    r = 1 /* right bound of furthest interval */
    for i=2 to N
        if i > r /* not in current interval */
            l = i
            r = i
            while r ≤ N and S[r] == S[r-l+1] /* project r to its position at prefix
*/
                r += 1
            r -= 1
            X[i] = r-l+1
        else
            i_prime = i-l+1
            if i+X[i_prime]-1 < r /* doesn't extend over current interval */
                X[i] = X[i_prime]
            else
                l = i
                while r ≤ N and S[r] == S[r-l+1]
                    r += 1
                r -= 1
                X[i] = r-l+1
    X[1] = N
    return X
```

Explanation

> **prefix-substring at `i`**: Longest substring starts from `S[i]` such that it is also the prefix of `S`
>
> **interval**: The the prefix-substring furthest to the right we have found so far. Bounded by `l` and `r`

The workflow is:

- Skipping `X[1]` because it's definitely `N`
- Iterate `i` from `2` to `N`. For each `i`:
    - Check if `i` is in the interval:
        - If not, set both bounds to `i`. Then compare the substring to prefix character by character and increase `r` accordingly. Substract `1` from `r`. Set `X[i] = r-l+1`.

- If it is in the interval, let `i_prime` be that corresponding position of `i` in prefix. Check if `X[i_prime]` makes the new prefix-substring touches the right bound of interval:
  - If not, then the prefix-substring at `i` must be the same as that at `i_prime`. The next `X[i_prime]+1` characters are the same thus the `X[i] == X[i_prime]`.
  - If it does, move the left bound to `i`. Then compare the substring to prefix character by character starting from `r` and increase `r` accordingly. Substract `1` from `r`. Set `X[i] = r-l+1`.

- Set `X[1]=N`.
- Return `X`

Space Complexity

Extra variables used are `N`, `l`, `r`, `i`, `i_prime` and are all single values. Therefore extra space complexity is $O(1)$.

Time Complexity

After the `for` loop ends, the `while` loop in it would take $O(N)$ time. Because each time `while` is executed `r` would increase by `1`, but `r` has upper bound `N` and `r -= 1` would be executed at most `N-2` times (in every iteration of `for`).
Not considering the `while` inside, the `for` loop clearly takes $O(N)$ time to complete. Therefore total time complexity is $O(N)$ (`while`) $+O(N)$ (`for` without `while) $+O(1) = O(N)$.

4.

```
function pattern_count(p, t)
    c = "$"
    S = p+c+t
    X = generateX(S) /* from previous subproblem */
    p_l = p.len
    cnt = 0
    for i=1 to S.len
        if X[i] == p_l
            cnt += 1
    return cnt
```

> Assuming `p` and `t` only contains uppercase alphabets.
> m : Length of `p`.
> n : Length of `t`.

Explanation

The workflow is:

1. Concatenate `p`, `c`, and `t`. `c` can be any character not in the character set of `p` and `t`. Here I choose `c = "$"`. The concatenated string is called `S`.
2. Build the `X` array of `S`.

3. Traverse `X`, check if `X[i] == p.len`. If true, the prefix-substring of `S` at `i` contains `p`, therefore add `1` to `cnt`.
4. Return `cnt`

By making the pattern the prefix of a string, we can utilize `X` to match pattern. And since `S[p.len+1]` is a character not in either `p` or `t`, `X[i]` is guaranteed to be less than `p.len`, thus we can use `X[i] == p_l`.

Space Complexity

`S` and `X` both take $O(m) + O(1) + O(n) = O(m + n)$ space. Other extra variables just store single value therefore take $O(1)$ space. Total extra space complexity is $O(m + n)$.

Time Complexity

`generateX(S)` and the `for` loop both take $O(m + n)$ time. Other operations take $O(1)$ time combined. Therefore total time complexity is $O(m + n)$.

Problem 3

> Refs:
>
> [1] https://stackoverflow.com/a/53256925/14977283
> [2] https://cs.stackexchange.com/a/108793
> [3] Tarjan, R.E., & van Leeuwen, J. (1984). Worst-Case Analysis of Set Union Algorithm. *Journal of the ACM, (31).*
> [4] Disjoint set lecture slide
> [5] https://stackoverflow.com/a/12690210/14977283
> [6] Kaplan, H., & Shafrir, N., & Tarjan, R.E. (2002). Union-find with Deletions.

1.

```
bipartite = True
first_neighbor = NIL
function INIT(N)
    for i=0 to N-1
        MAKE_SET(i)
    first_neighbor = array[N] filled with -1

function ADD_EDGE(x, y)
    if not bipartite
        return
    if FIND_SET(x) == FIND_SET(y)
        bipartite = False
        return
    if first_neighbor[x] == -1
        first_neighbor[x] = y
    else
        UNION(y, first_neighbor[x])
    if first_neighbor[y] == -1
        first_neighbor[y] = x
    else
        UNION(x, first_neighbor[y])
    return

function IS_BIPARTITE()
    return bipartite
```

Explanation

Main Idea

Maintain this property: If the graph is still bipartite, nodes in the same set have the same "color".

`INIT()`

Do `MAKE_SET()` for each vertex and initialize an array `first_neighbor[N]` filled with `-1`.

`ADD_EDGE()`

- If `bipartite` is `false`, adding any edge won't make it bipartite, therefore do nothing and return.
- Check if `x` and `y` are in the same set.
  - If true, it means that the edge have both vertices with the same color, which isn't bipartite. Therefore set `bipartite = False` and return.
  - If false, do the following
    - Set `first_neighbor[x]` if it's not set yet. Else, `UNION(y, first_neighbor[x])`.
    - Set `first_neighbor[y]` if it's not set yet. Else, `UNION(x, first_neighbor[y])`.
    - This works because `FIND_SET[x] ≠ FIND_SET[y]` happens only in these two cases
      1. `x` and `y` are not connected.
         Therefore we associate the two subgraph's color by the operations above.
      2. `x` and `y` are connected.
         Then `x` and `y` must be different color, and the operations above changes nothing because the two vertices being unioned are already in the same set.

`IS_BIPARTITE()`

Return `bipartite`.

Time Complexity

`INIT()`

`for` loop runs `N` times, and initializing `first_neighbor` takes $O(n)$ time. Total time complexity is $O(N)$.

`ADD_EDGE()`

Total time complexity = $O(1)$ + 2 · `FIND_SET()` + 2 · `UNION()`

`IS_BIPARTITE()`

Returning a stored value is $O(1)$.

`INIT()` + (`ADD_EDGE()` and `IS_BIPARTITE()` for M times combined)

For linked list + union by size implementation, `FIND_SET()` takes $O(1)$ time and `UNION()` takes $O(\log N)$ time on average. Therefore total time complexity is
$O(N) + M(O(1) + 2 \cdot O(1) + 2 \cdot O(\log N)) = O(N + M \log N)$

2.

```
contradict = False
W = NIL
L = NIL
function INIT(N)
    for i=0 to N-1
        MAKE_SET(i)
    W = array[N] filled with -1
    L = array[N] filled with -1

function WIN(a, b)
    if contradict
        return
    if FIND_SET(a) == FIND_SET(L[b])
        return
    if FIND_SET(a) == FIND_SET(b) or FIND_SET(W[b])
        contradict = True
        return
    if W[a] == -1:  W[a] = b
    else:           UNION(b, W[a])
    if L[b] == -1:  L[b] = a
    else:           UNION(a, L[b])
    if L[a] == -1
        if W[b] == -1:  return
        else:           L[a] = W[b]
    else
        if W[b] == -1:  W[b] = L[a]
        else:           UNION(L[a], W[b])
    return

function TIE(a, b)
    if contradict
        return
    if FIND_SET(a) == FIND_SET(b)
        return
    if FIND_SET(a) == FIND_SET(W[b]) or FIND_SET(L[b])
        contradict = True
        return
    UNION(a, b)
    if W[a] == -1
        if W[b] == -1:  pass
        else:           W[a] = W[b]
    else
        if W[b] == -1:  W[b] = W[a]
        else:           UNION(W[a], W[b])
    if L[a] == -1
        if L[b] == -1:  pass
        else:           L[a] = L[b]
    else
```

```
        if L[b] == -1:  L[b] = L[a]
        else:           UNION(L[a], L[b])

    function IS_CONTRADICT()
        return contradict
```

Explanation

Main Idea

We can think of people as vertices of a graph, and `a` winning `b` as a directed edge from `a` to `b`. Then a non-contradicting result should yield a graph such that:

1. Vertices can be separated into three disjoint sets.
2. Every edge into set `X` should come from set `Y` and every edge from set `X` should go into set `Z`.

In short, it's kinda like a directed tripartite graph. So operations are similar to the previous subproblem.

`INIT()`

`MAKE_SET()` for each player, then initialize array `W` and `L` with value `-1`. `W[a]` and `L[a]` stores a player that player `a` wins / loses to.

`WIN()`

- If `contradict == True`, newer game results won't make it valid. Therefore we return.
- If `FIND_SET(a) == FIND_SET(L[b])`, then nothing needs be done. Therefore we return.
- If `FIND_SET(a) == FIND_SET(b) or FIND_SET(W[b])`, previous results suggests `a` should tie or lose to `b`, meaning it's contradicting. Therefore set `contradict = True` and return.
- The only case left is `a` and `b` are not connected. Therefore associate those two subgraphs in the following steps.
- Set `W[a] = b` if it's not set yet. Else `UNION(b, W[a])`.
- Set `L[b] = a` if it's not set yet. Else `UNION(a, L[b])`.
- Check `L[a]` and `W[b]`:
  - If both are not set yet, do nothing.
  - If one is set and the other isn't, assign the set value to the not set one.
  - If both are set, `UNION(L[a], W[b])`.
- Return.

`TIE()`

- If `contradict == True`, newer game results won't make it valid. Therefore we return.
- If `FIND_SET(a) == FIND_SET(b)`, then nothing needs be done. Therefore we return.
- If `FIND_SET(a) == FIND_SET(W[b]) or FIND_SET(L[b])`, then previous results suggests `a` won't tie with `b`, meaning it's contradicting. Therefore set `contradict = True` and return.
- The only case left is `a` and `b` are not connected. Therefore associate those two subgraphs in the following steps.
- `UNION(a, b)`
- Check `W[a]` and `W[b]`:

   - If both are not set yet, do nothing.
   - If one is set and the other isn't, assign the set value to the not set one.
   - If both are set, `UNION(W[a], W[b])`.
- Check `L[a]` and `L[b]`:
   - If both are not set yet, do nothing.
   - If one is set and the other isn't, assign the set value to the not set one.
   - If both are set, `UNION(L[a], L[b])`.
- Return.

`IS_CONTRADICT()`

Return `contradict`.

Time Complexity

`FIND_SET()` : $O(1)$

`UNION()` : $O(\log N)$

`INIT()`
`for` loops runs `N` times and initializing `W` and `L` takes $O(N)$ time. Total time complexity = $O(N)$.

`WIN()`
Total time complexity = $O(1)$ + 4 `FIND_SET()` + 3 `UNION()` = $O(\log N)$ at worst case.

`TIE()`
Total time complexity = $O(1)$ + 4 `FIND_SET()` + 3 `UNION()` = $O(\log N)$ at worst case.

`IS_CONTRADICT()`
Total time complexity = $O(1)$.

`INIT()` + (`WIN()` , `TIE()` , `IS_CONTRADICT()` for M times combined)
Total time complexity = $O(N) + M \cdot O(\log N) = O(N + M \log N)$

3.

`init()`

This function simply calls `djs_init()`, so we can just look at `djs_init()`.
The `for` loop runs `n` times, and all other operations take only $O(1)$ time, therefore total time complexity is $O(N)$.

`show_cc()`

This function only prints a single stored value, so it takes $O(1)$ time.

`add_edge()`

This function just calls `djs_save()` and `djs_union()`. We will look at these two functions.

`djs_save()`

Only doing a `stack_push()`, so it's $O(1)$ time.

`djs_union()`

This function calls two `djs_find()` and two `djs_assign()`. `djs_assign()` takes only $O(1)$ time. `djs_find()` however is more complicated and we will dive into that later, let's just say it's some $f(N)$. Therefore `djs_union()` should take $O(f(N))$.

Then the total time complexity of `add_edge()` is $O(f(N))$.

`undo()`

Since this is reversing the change done by `add_edge()`, `undo()` at worst case should have the exact same time complexity $O(f(N))$.

`init()` + (`add_edge()`, `undo()`, `show_cc()` for M times combined)

Previous analysis yields total complexity $= O(N) + M \cdot O(f(N))$
According to the work by Tarjan and van Leeuwen (Ref [3]), on p.259-260

> LEMMA 7. Suppose $m \geq n$. In any sequence of set operations implemented using any form of compaction and naive linking, the total number of nodes on find paths is at most $(4m+n)\lceil \log_{1+\lfloor m/n \rfloor} n \rceil$. With halving and naive linking, the total number of nodes on find paths is at most $(8m+2n)\lceil \log_{1+\lfloor m/n \rfloor} n \rceil$.
>
> LEMMA 9. Suppose m < n. In any sequence of set operations implemented using compression and naive linking, the total number of nodes on find paths is at most $n + 2m\lceil \log n \rceil + m$.

Where m is the number of `FIND_SET()` and n is the number of `MAKE_SET()`.
These two lemmas combined tells us that $M \cdot f(n) = O(M \log N)$.

Using this result, the total time complexity $O(N) + M \cdot O(f(N)) = O(N + M \log N)$.

4.

`init()`, `add_edge()`, `undo()`, `show_cc()` is basically the same from previous subproblem, except that a new variable `sz` is maintained for each set, but the analysis remains valid. So I will continue from the last part.

`init()` + (`add_edge()`, `undo()`, `show_cc()` for M times combined)

Total time complexity we now have is $O(N) + M \cdot O(f(N))$, where $f(N)$ is the time complexity of `FIND_SET()`.

From p18 of the lecture slide (Ref [4]), since this implementation matches 方法二: tree法+Weighted Union, $f(N) = \log N$ amortized.

Therefore total time complexity $O(N) + M \cdot O(f(N)) = O(N + M \log N)$

5.

> N: total number of `MAKE_SET()`

```
nodes = array()
sets = array()

function MAKE_SET(x)
    ptr = allocate_node()
    ptr.p = ptr
    ptr.delete = False
    ptr.rank = 0
    ptr.size = 0
    ptr.empty = 0
    nodes[x] = ptr
    sets[x] = new_linked_list(ptr)

function FIND_SET(x)
    if x.p == x
        return x
    x.p = FIND_SET(x.p)
    return x.p

function SAME_SET(x, y)
    if FIND_SET(x) == FIND_SET(y)
        return True
    else
        return False

function UNION(x, y)
    x = FIND_SET(nodes[x])
    y = FIND_SET(nodes[y])
    if x == y
        return
    if x.rank < y.rank
        swap(x, y)
    if x.rank == y.rank
        x.rank += 1
    x.size += y.size
    x.empty += y.empty
    y.p = x
    sets[x].connect_tail(sets[y])

function REBUILD(root)
    live_nodes = array()
    for i in sets[root]
```

```
            if i.deleted == False
                live_nodes.append(i)
        new_root = live_nodes[1]
        new_root.p = new_root
        new_root.delete = False
        new_root.rank = 0
        new_root.size = 1
        new_root.empty = 0
        sets[new_root] = new_linked_list(new_root)
        for i in live_nodes[2..]
            i.p = new_root
            new_root.rank = 1
            new_root.size += 1
            sets[new_root].append(i)

    function DELETE(x)
        root = FIND_SET(nodes[x])
        nodes[x].delete = True
        root.empty += 1
        if root.empty ≥ floor(root.size/2)
            REBUILD(root)

    function ISOLATE(x)
        DELETE(x)
        MAKE_SET(x)
```

For a node `a`

- `a.p` is a pointer to its parent. If this points to `a` itself, it means `a` is the root.
- `a.delete` is a flag to track if it's deleted.
- `a.rank` tracks the rank of `a`. It's only maintained for roots.
- `a.size` tracks how many nodes are there.
- `a.empty` tracks the number of children that is deleted but the node is still there.

`MAKE_SET()`, `FIND_SET()`, and `UNION()` are basically the same as the tree implementation with union-by-rank and path compression. Except that three more attributes (`a.delete`, `a.size`, and `a.empty`) are maintained (all in $O(1)$ time) and an array of linked lists (`sets`) is maintained. Therefore time complexity of these three functions are the same as those on the slide p.18 (Ref [4]).

`SAME_SET(x, y)` is just calling two `FIND_SET()`s and compare, therefore has the same time complexity as `FIND_SET()`.

`DELETE(x)` is done by

- `FIND_SET(nodes[x])` to get the root `root`.
- Mark `nodes[x].deleted`.
- Increment `root.empty`.
- Check if over half of the nodes are empty nodes. If true, do `REBUILD(root)`.

`REBUILD(root)` is done by

- Traverse `sets[root]`, which stores a linked list containing all nodes in this tree.
    - If this node is not marked as deleted, append it to an array `live_nodes`.
- Make `live_nodes[1]` be the new root of live nodes. Change its attribute accordingly.
- Traverse `live_nodes[2..]`
    - Change `a.p` to `live_nodes[1]` for each node.
    - Increment `live_nodes[1].size`.

One `REBUILD()` clearly takes $O(n)$, where n is the number of nodes in the set. But since it's only executed when over half of the nodes are deleted. Thus, we can amortize this $O(n)$ costs to $\frac{n}{2}$ `DELETE()`s, making it $O(1)$ for each `DELETE()`. Then, the time complexity for `DELETE()` becomes `FIND_SET()` + $O(1)$.

`ISOLATE(k)` is done with a `DELETE(k)` and `MAKE_SET(k)`, therefore the time complexity for it is also the same as `FIND_SET()` + `MAKE_SET()`.

In short, time complexity of these functions can be mapped to the three elementary functions using this table:

New function	Time complexity
`SAME_SET()`	`FIND_SET()` + $O(1)$
`REBUILD()`	$O(1)$ amortized
`DELETE()`	`FIND_SET()` + $O(1)$
`ISOLATE()`	`FIND_SET()` + `MAKE_SET()` + $O(1)$

Finally, (`MAKE_SET()`, `UNION()`, `SAME_SET()`, `ISOLATE()` for M times combined) will have the same time complexity as (`MAKE_SET()`, `UNION()`, `FIND_SET()` for M times combined), which is $O(M\alpha(N)) = O(M\alpha(M))$.