Problem 1 - Hashing

1. $1 - \frac{C_n^{n^2}}{n^{2n}}$
2. $\frac{|P|}{4} + \frac{|P|}{16}$
3. a.

   1. 34 mod 11 = 1

      table = {, 34, , , , , , 18,,,}

   2. 9 mod 11 = 9

      table = {, 34, , , , , , 18,, 9,}

   3. 37 mod 11 = 4

      table = {, 34, , , 37, , , 18,, 9,}

   4. 40 mod 11 = 7(collide)

      index 7 has value 18, so index++ until find empty location which is index 8

      table = {, 34, , , 37, , , 18, 40, 9,}

   5. 32 mod 11 = 10

      table = {, 34, , , 37, , , 18, 40, 9, 32}

   6. 89 mod 11 = 1(collide)

      index 1 has value 34, so index++ until find empty location which is index 2

      table = {, 34, 89, , 37, , , 18, 40, 9, 32}

   b.

   1. 34 mod 11 = 1

      table = {, 34, , , , , , 18,,,}

2. 9 mod 11 = 9

   table = {, 34, , , , , , 18,, 9,}

3. 37 mod 11 = 4

   table = {, 34, , , 37, , , 18,, 9,}

4. 40 mod 11 = 7 (collide)

   use $(h_1(x) + ih_2(x))mod(m)$ i++ to get next empty location,

   $(h_1(x) + h_2(x))mod(m) = (7 + 0 + 1)mod(11) = 8$ (empty index)

   table = {, 34, , , 37, , , 18, 40, 9,}

5. 32 mod 11 = 10

   table = {, 34, , , 37, , , 18, 40, 9, 32}

6. 89 mod 11 = 1(collide)
   use $(h_1(x) + ih_2(x))mod(m)$ i++ to get next empty location,
   $(h_1(x) + h_2(x))mod(m) = (1 + 9 + 1)mod(11) = 0$
   (empty index)
   table = {89, 34, , , 37, , , 18, 40, 9,}

4.

1. 6 mod 7 = 6

table1 = {,,,,,, 6}

table2 = {,,,,,,}

2. 31 mod 7 = 3

table1 = {,,, 31,,, 6}

table2 = {,,,,,,}

3. 2 mod 7 = 2

table1 = {,, 2, 31,,, 6}

table2 = {,,,,,,}

4. 41 mod 7 = 6(collide), put 41 into table1[6]

use $h_2(6) = \lfloor \frac{6}{7} \rfloor mod(7) = 0$ insert 6 to table2[0]

table1 = {,, 2, 31,,, 41}

table2 = {6,,,,,,}

5. 30 mod 7 = 2(collide), put 30 into table1[2]

use $h_2(2) = \lfloor \frac{2}{7} \rfloor mod(7) = 0$ (collide), insert 2 to table2[0]

use $h_1(6) = 0$ insert 6 to table1[0]

table1 = {6,, 30, 31,,, 41}

table2 = {2,,,,,,}

6. 45 mod 7 = 3(collide), put 45 into table1[3]

use $h_2(31) = \lfloor \frac{31}{7} \rfloor mod(7) = 4$, insert 31 to table2[4]

table1 = {6,, 30, 45,,, 41}

table2 = {2,,,, 31,,}

7. 44 mod 7 = 2(collide), put 44 into table1[2]

use $h_2(30) = \lfloor \frac{30}{7} \rfloor mod(7) = 4$(collide), insert 30 to table2[4]

use $h_1(31) = 3$(collide), insert 31 to table1[3]

use $h_2(45) = \lfloor \frac{45}{7} \rfloor mod(7) = 6$, insert 30 to table2[6]

table1 = {6,, 44, 31,,, 41}

table2 = {2,,,, 30,, 45}

Problem 2 - String Matching

1. We run a for loop to check the total length N characters are the same.

Space Complexity: O(1), we don't use extra space

Time Complexity: O(NQ) we call the function Q times which every call has a for loop of time complexity O(N)

```
procedure strMatching(S, l_1, l_2, n)
    for i from 0 to n - 1
        if S[l_1 + i] != S[l_2 + i]
            return false
        end if
    end for
    return true
end procedure
```

2. x(1) = 8

   x(2) = 0

   x(3) = 0

   x(4) = 0

   x(5) = 3

   x(6) = 0

   x(7) = 0

   x(8) = 0

   X[8] = {8, 0, 0, 0, 3, 0, 0, 0}

3. According to the problem, we can use the KMP's failure function(line 2-16) to create a Failure Table which means S[1...FailureFunc[i]] == S[i... i + FailureFunc[i] - 1]. It means we can find all the possible S[1...p] == S[i... i + p -1] which p

is FailureFunc[i].

Space Complexity: O(n), FailureFun use n extra space which n is the length of string

Time Complexity: O(n) , as we know for failure function, it use O(n) to finish, and the next for loop runs n times, so it runs n + n = O(n)

```
procedure countFunc(S)
    assume FailureFunc[S.length()]
    set FailureFunc[1] to 1

    for i from 2 to S.length()
        set j to FailureFunc[i - 1]
        while j != 1 and S[i] != S[j]
            set j to FailureFunc[j - 1]
        end while
        if(S[i] == S[j])
            set FailureFunc[i] to j + 1
        else
            set FailureFunc[i] to 1
        end if
    end for

    for i from 1 to S.length()
        if i == FailureFunc[i] set x[i] to
S.length()- i
        else if FailureFunc[i] != 1
            set x[i - FailureFunc[i]] to
FailureFunc[i] - 1
        else
            set x[i] to 0
        end if
end procedure
```

4. Space Complexity: O(1) because it only use X array (not count in extra space), and two variable

   Time complexity: O(t.length() + O(s.length())) + O(t.length()) + O(t.length()) = O(t.length() + s.length()) for find the first hit and countFunc() and a for loop

```
procedure stringMatching(S, P)
    set result to 0
    set i to KMP-StringMatching(S, P) //return the first hit index
    countFunc(S[i:S.length()])
    for j from i to S.length()
        if(X[j] >= P.length()) set result to result + 1
        end if
    end for
    return result
end procedure
```

Problem 3 - Having Fun with Disjoint Set

1.
```
set isBip to true

procedure INIT(N)
    for i from 0 to N - 1
        MAKE-SET(i)
    end for
end procedure

procedure ADD-EDGE(x, y)
    if FIND-SET(x) == FIND-SET(y)
        set isBip to false
    end if
    UNION(x, y)
```

```
end procedure

procedure IS-BIPARTITE()
    return isBip
end procedure
```

2.

```
assume isContr is false

procedure INIT(N)
    for i from 0 to N + 2
        MAKE-SET(i)
    end for
end procedure

procedure WIN(a, b)
    set aSet to FIND-SET(a)
    set bSet to FIND-SET(b)
    set litSet to FIND-SET(N)
    set midSet to FIND-SET(N + 1)
    set bigSet to FIND-SET(N + 2)
    if aSet == litSet or bSet == bigSet or aSet == bSet
        set isContr to true
    else if (aSet != midSet and aSet != bigSet) or (bSet != litSet and bSet != midSet)
        if (aSet != midSet and aSet != bigSet) and (bSet != litSet and bSet != midSet)
            UNION(N + 2, a)
            UNION(N + 1, b)
        else if bSet == midSet
            UNION(N + 2, a)
        else if bSet == litSet
            UNION(N + 1, a)
        else if aSet == midSet
            UNION(N, b)
        else if aSet == bigSet
            UNION(N + 1, b)
```

```
            end if
        end if
end procedure

procedure TIE(a, b)
    UNION(a, b)
end procedure

procedure IS-CONTRADICT()
    return isContr
end procedure
```

3. To prove that it is $O(N + Mlog(N))$, we have to discuss ADD-EDGE(), SHOW-CC(), UNDO()

    - ADD-EDGE(): call dfs_save() which push a element to a stack use $O(1)$ and djs_union() which call two djs_find() with path compression use $O(logN)$ and two djs_assign() which push a element to a stack use $O(1)$, so the total time consumption is $O(1) + 2 * O(logN) + 2 * O(1) = O(logN)$
    - SHOW-CC(): $O(1)$ because it only call printf
    - UNDO(): undo() calls djs_undo() which has a while loop. In each iteration, it pop an element and a element means an ADD-EDGE() operation. Namely, it at most pop $(M - 1)$ element in only one djs_undo() in the total process(In total M operations, it calls UNDO() 1 times pop (M - 1) elements and ADD-EDGE() (M - 1) times). So it takes $O(M - 1) = O(M)$

    So the total time consumption is $O(N)$(for init) + $(M - 1)O(logN) + O(M) = O(N + MlogN)$

4. As above, we have to discuss ADD-EDGE(), SHOW-CC(), UNDO()

- ADD-EDGE(): call dfs_save() which push a element to a stack use $O(1)$ and djs_union() which calls two djs_find() with union by size use $O(logN)$ and three assign use $O(1)$ because it just do simple assignment, so an ADD-EDGE() cost $O(logN)$.
- SHOW-CC(): $O(1)$ because it only call printf
- UNDO(): undo() calls djs_undo() which has a while loop. In each iteration, it pop an element and a element means an ADD-EDGE() operation. Namely, it at most pop $(M-1)$ element in only one djs_undo() in the total process(In total M operations, it calls UNDO() 1 times pop (M - 1) elements and ADD-EDGE() (M - 1) times). So it takes $O(M-1) = O(M)$

So the total time consumption is $O(N)$(for init) + $(M-1)O(logN) + O(M) = O(N + MlogN)$

5.

```
assume SET[M] is an array
assume SIZE[M] is an array
procedure MAKE-SET(x)
    set SET[x] to x
    set SIZE[x] to 1
end procedure

procedure FIND-SET(x)
    if SET[x] != x
        set SET[x] to FIND-SET(SET[x])
    end if
    return SET[x]
end procedure

procedure UNION(x, y)
    set x to FIND-SET(x)
    set y to FIND-SET(y)
    if SIZE[x] > SIZE[y]
        set SET[y] to x
```

```
            set SIZE[x] to SIZE[x] + SIZE[y]
        else
            set SET[x] to y
            set SIZE[y] to SIZE[x] + SIZE[y]
        end if
end procedure

procedure SAME-SET(x, y)
    return FIND-SET(x) == FIND-SET(y)
end procedure

procedure ISOLATE(k)
    set x to FIND-SET(k)
    set SIZE[x] to SIZE[x] - 1
    set SET[k] to k
    set SIZE[k] to 1
end procedure
```