DSA-Hw3

Problem 0 - Proper References

- Problem 1: https://alrightchiu.github.io/SecondRound/comparison-sort-merge-sorthe-bing-pai-xu-fa.html
- Problem 2: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching-in-matrix/

Problem 1 - Cracking The Interview With How-How!

1.

Ans: 7

2.

`human algorithm`:
藉由 merge-sort 的概念，將問題轉為重複比較 sorted left array 與 sorted right array 然後將兩 sub-array 合併成一 sorted array。當 left-sub-array[leftIdx] < right-sub-array[rightIdx]，表示 left-sub-array[leftIdx] < right-sub-array[rightIdx] < ... < right-sub-array[rightSub.length] ，便能累計一次 Merge 中的 feasible pairs

`pseudo code`:

```
divide-and-conquer(array A, int pairs):
    length = A.length
    if length >= 2:
        m = floor((1+n)/2)
        pairs = divide-and-conquer(A[1...m],  pairs)
        pairs = divide-and-conquer(A[m+1...length],  pairs)

        leftSub = A[1...m]
        leftSub.insert_to_end(Inf)  // suppose O(1)
        rightSub = A[m+1...length]
        rightSub.insert_to_end(Inf)  // suppose O(1)
        leftIdx = 1, rightIdx = 1
        leftLength = m, rightLength = length - m

        for i = 1 to i = length:
            if (leftSub[leftIdx] < rightSub[rightIdx])
                A[i] = leftSub[leftIdx]
                pairs += (rightLength - rightIdx + 1)
                leftIdx += 1
            else
                A[i] = rightSub[rightIdx]
                rightIdx += 1
    return pairs
```

```
find-pairs(array A):
    pairs = 0
    pairs = divide-and-conquer(A, pairs)
    return pairs
```

`time complexity`:
共分割了 n 次，分割的 time complexity 為 $O(n)$，共合併了 $log(n)$次，每次合併平均的 time complexity 為 $O(n)$，合併的 time complexity 為 $nlog(n)$，整體 time complexity 為 $O(n)+O(nlog(n))$

3-a.

Ans: 5,2,4,3

3-b.

在 while(!Ordered(P)) loop 中，function Ordered () 的 time complexity 為 $O(n)$，執行 while (i+1 < len(P)) loop 的 time complexity 為 $O(n)$，也就是說 while(!Ordered(P)) loop 內的 time complexity 為 $O(n)+O(n)=O(n)$ 。
最糟糕的情況為： $k=n-1$ ，且在 while(!Ordered(P)) loop 內每次只移除 2 個 element，因此執行了 $floor(\frac{n-1}{2})$次的 while(!Ordered(P)) loop，因此最糟糕的情況下的 time complexity 為 $floor(\frac{n-1}{2})\cdot O(n)=O(n^2)$

4.

若每次觸發 `if (P[i]. difficulty > P[i +1]. difficulty ) -> remove P[i] and P[i+1]` 時，P[i], P[i+1]只有其中一者為 misplaced problem，演算法結束時最多會觸發 k 次。 因此移除的 problems 數最多為 2k 個。

5.

`human algorithm`:
上述 pseudo code 若觸發了`if (P[i]. difficulty > P[i +1]. difficulty ) -> remove P[i] and P[i+1]` 後 p[i-1] > p[i] 需要待下一次觸發 `while(!Ordered(P))` 才會被檢查到，因此改成在同個 `while (i+1 < len (P))` loop 中可以向前檢查就可以在 $O(n)$ 內達到同樣的效果。

`pseudo code`:

```
Remove-out-of-order-pairs(Problem P []):
    i = 0
    while (i < len (P)):
        if (i+1 < len(P) and P[i].difficulty > P[i+1].difficulty ):
            remove (P[i])
            remove(P[i])
```

```
        else if (i-1 >= 0 and P[i-1].difficulty > P[i].difficulty):
            i = i-1
            remove (P[i])
            remove (P[i])
        else :
            i += 1
```

6.

`human algorithm`:
將 P array 中的 unordered problems 與 ordered problems 分別存在兩個 array 中， 再用 Merge sort 將存 unordered problems 的 array 進行排序，最後將排序好的 unordered problems array 與原本就已經排序好的 ordered problems array merge 起來。

`pseudo code`:

```
(index start from 1)
Merge(array A1, array A2):
    leftLength = A1.length
    rightLength = A2.length
    length = leftLength + rightLength
    array A[length] = {0}
    A1.insert_to_end(Inf)  // suppose O(1)
    A2.insert_to_end(Inf)  // suppose O(1)
    leftIdx = 1, rightIdx = 1

    for i = 1 to i = length:
        if (leftSub[leftIdx] < rightSub[rightIdx])
            A[i] = leftSub[leftIdx]
            leftIdx += 1
        else
            A[i] = rightSub[rightIdx]
            rightIdx += 1


Merge-Sort(array A, front, end):
    length = A.length
    if (length >= 2):
        mid = floor((front+end)/2)
        A1 = Merge-Sort(A[1...m])
        A2 = Merge-Sort(A[m+1...length])
        A = Merge(A1, A2)
    return A


Sort-P(Problem P []):
    unordered_array = empty array
    ordered_array = empty array
    unordered_array_idx = 1
```

```
    ordered_array_idx = 1

    for (i = 1 to P.length):
        ordered_array[ordered_array_idx] = P[i]
        if (ordered_array.length >= 2 and ordered_array[ordered_array_idx]
< ordered_array[ordered_array_idx-1]):
            unordered_array[unordered_array_idx] =
ordered_array[ordered_array_idx]
            unordered_array_idx += 1
            unordered_array[unordered_array_idx] =
ordered_array[ordered_array_idx-1]
            unordered_array_idx += 1
            ordered_array_idx -= 2
        ordered_array_idx ++

    sorted_unordered_array = Merge-Sort(unordered_array, 1,
unordered_array.length)
    sorted_P = Merge(ordered_array, sorted_unordered_array)
    return Sort-P
```

`time complexity`:

- 將unordered problems 與 ordered problems 分別存在兩個 array 中的 time complexity 為 $O(n)$
- 由於 unordered problems 的數目最多為 2k，因此排序 unordered problems array 的 time complexity 為 $O(klogk)$
- Merge 排序好的 unordered problems array 與原本就排序好的ordered problems array 的 time complexity 為 $O(n)$
- 整體 time complexity 為 $O(n) + O(klogk) + O(n)$ = $O(n + klogk)$

Problem 2 - DSA Chief Scientist How-How

1.

`human algorithm`:
用 rolling hash (Rabin-Karp) Algorithm ，一長度為 h 的字串可以表示成一個 h 位數的 26 進位數對一質數 prime 取 mod，假設 prime 取的值夠大使得 collision不會發生。
step1: 首先計算 pattern 的 hash value。
step2: 任何一個 map 的任一個 row，都要先計算開頭長度為 h 的字串的 hash value，接著橫向更新長度為 h 的字串的 hash value 並且比對。
step3: 當一個 row 比對結束便換到下一個 row 重複 step2，當比對的過程中有與 pattern 相同的 hash value 就可以提前終止比對，否則持續比對到所有 row 都比對完才終止。

`pseudo code`:

```
h = length of pattern
radix = 26
prime is a large prime value
```

```
calculate-hash-value(array a):
    hash_value = 0
    for i = 1 to i = h:
        hash_value = hash_value + ord(a[i]) * radix^(h - i)
        hash_value = hash_value % prime
    return hash_value

check-map(2D-array Map):
    N = number of rows
    M = number of columns
    // (step 1)
    pattern_hash_value = calculate-hash-value(pattern)
    // check every row (step 3)
    for i = 1 to i = N:
        // calculate fisrt hash value of this row
        map_hash_value = calculate-hash-value(Map[i][1...h])
        // check every sub-string in this row (step 2)
        for s = 1 to s = M:
            if (map_hash_value == pattern_hash_value):
                return True
            else:
                add_idx = (s + h) if (s + h) <= M else (s + h - M)
                map_hash_value = (radix * (map_hash_value - Map[i][s] *
radix^(h-1)) + Map[i][add_idx]) % prime
    return False

// run check-map(Map) K times to check each map
```

2.

由於此題假設 collision不會發生，因此最壞的情況就是 K 張 Map 都沒有出現 desired pattern。
計算 pattern_hash_value 的時間複雜度為 $O(h)$。
計算 Map 中任一 row 的第一個 map_hash_value 的時間複雜度同樣也是 $O(h)$，在同一個 row 中檢查所有 sub-string 的複雜度為 $O(M)$，最糟糕的情況就是 K 張 Map 的每個 row 都完整檢查過一次，時間複雜度為 $O(K*N*(h+M)) = O(K*N*M)因此最糟糕的情況下，時間複雜度為O(h) + O(K*N*M)$

3.

`human algorithm`:
用 rolling hash (Rabin-Karp) Algorithm ，一長度為 h 的字串可以表示成一個 h 位數的 26 進位數對一質數 prime 取 mod，假設 prime 取的值夠大使得 collision不會發生。
step1: 由於 pattern 縱向不只一個 row，要計算 pattern 的 hash value 要先計算 pattern 橫向每個 column 個別的 hash value， 存於 pattern_col_hash[h]。
step2: 接著以 pattern_col_hash[h] 橫向計算 pattern 的 hash value。
step3: 任一 Map 要先計算前 g rows 每個 column 個別的 hash value， 存於 map_col_hash[M]。
step4: 對於每個 g rows 都要先以 map_col_hash[M] 計算開頭長度為 h 的 hash value，接著橫向更新長度為 h

的 hash value 並且比對。前 1~g rows 比對完後若沒有檢查出 pattern 則下移一 row，更新 map_col_hash[M] 比對 2~g+1 rows。當比對的過程中有與 pattern 相同的 hash value 就可以提前終止比對，否則持續比對到 N~g-1 rows 比對完才終止。

`pseudo code`:

```
h = horizontal length of pattern
g = vertical length of pattern
radix = 26
prime is a large prime value


build-col-hash-table(2D-array a, row, col):
    col_hash_table = empty 1D-array
    for i = 1 to col:
        row_hash = 0
        for j = 1 to j=row:
            row_hash = row_hash + ord(a[j][i]) * radix^(row - i)
            row_hash = row_hash % radix
        col_hash_table.append(row_hash)
    return col_hash_table

calculate-hash-value(array a):
    hash_value = 0
    for i = 1 to i = h:
        hash_value = hash_value + ord(a[i]) * radix^(h - i)
        hash_value = hash_value % prime
    return hash_value

check-map(2D-array Map):
    N = number of rows
    M = number of columns
    // step1
    pattern_col_hash = build-col-hash-table(pattern, g, h)
    // step2
    pattern_hash_value = calculate-hash-value(pattern_col_hash)
    // step3
    map_col_hash = build-col-hash-table(Map, g, M)
    // step4
    for i = 1 to i = N:
        map_hash_value = calculate-hash-value(map_col_hash[1...h])
        for s = 1 to s = M:
            if (map_hash_value == pattern_hash_value):
                return True
            else:
                add_idx = (s + h) if (s + h) <= M else (s + h - M)
                map_hash_value = (radix * (map_hash_value - map_col_hash[s]
* radix^(h-1)) + map_col_hash[add_idx]) % prime
        // update map_col_hash
        for j = 1 to j = M:
            add_idx = (i + g) if (i + g) <= N else (i + g - N)
            map_col_hash[j] = (radix * (map_col_hash[j] - Map[i]
```

```
[j]*radix^(g-1))) + Map[add_idx][j])% prime
    return False

// run check-map(Map) K times to check each map
```

4.

由於此題假設 collision不會發生，因此最糟糕的情況就是 K 張 Map 都沒有出現 desired pattern。
建立 pattern_col_hash 的時間複雜度為 \$O(g*h*)。計算$pattern_{h}ash_{v}alue$的時間複雜度為*O(h)*
。建立$map_{c}ol_{h}ash$的時間複雜度為*O(g*M)。$updatemap_{c}ol_{h}ash$的時間複雜度為O(M)
。以$map_{c}ol_{h}ash[M]$計算開頭長度為h的$hashvalue$的時間複雜度為O(h)
。對於每$grows$橫向更新長度為h的$hashvalue$並且與$pattern_{h}ash_{v}alue$比對的時間複雜度為O(M)\$。

因此最糟糕的情況下，K 張 Map 所有 row 都檢查過一次，時間複雜度為 \$O(g*h*+*h*) + *O*(*K*(g*M*+*N*(M+h+M))) = O(g*h* + *K*M*N)\$

5.

`human algorithm`:
用 rolling hash (Rabin-Karp) Algorithm ，一長度為 h 的字串可以表示成一個 h 位數的 26 進位數對一質數 prime 取 mod，假設 prime 取的值夠大使得 collision不會發生。
step1: 由於 pattern 縱向不只一個 row，要計算 pattern 的 hash value 要先計算 pattern 橫向每個 column 個別的 hash value， 存於 pattern_col_hash[M]。
step2: 接著以 pattern_col_hash[M] 橫向計算 pattern 的 hash value。
step3: 任一 Map 要先計算 N rows 每個 column 個別的 hash value， 存於 map_col_hash[M]。
step4: 先以 map_col_hash[M] 計算開頭長度為 M 的 hash value，接著橫向更新長度為 M 的 hash value 並且比對。當比對的過程中有與 pattern 相同的 hash value 就可以提前終止比對，否則持續比對到橫向所有 hash value 都比對完才終止。

`pseudo code`:

```
M = horizontal length of pattern
N = vertical length of pattern
radix = 26
prime is a large prime value


build-col-hash-table(2D-array a, row, col):
    col_hash_table = empty 1D-array
    for i = 1 to col:
        row_hash = 0
        for j = 1 to j=row:
            row_hash = row_hash + ord(a[j][i]) * radix^(row - i)
            row_hash = row_hash % radix
        col_hash_table.append(row_hash)
    return col_hash_table
```

```
calculate-hash-value(array a):
    hash_value = 0
    for i = 1 to i = h:
        hash_value = hash_value + ord(a[i]) * radix^(h - i)
        hash_value = hash_value % prime
    return hash_value

check-map(2D-array Map):
    // step1
    pattern_col_hash = build-col-hash-table(pattern, N, M)
    // step2
    pattern_hash_value = calculate-hash-value(pattern_col_hash)
    // step3
    map_col_hash = build-col-hash-table(Map, N, M)
    // step4
    map_hash_value = calculate-hash-value(map_col_hash[1...M])
    for s = 1 to s = M:
        if (map_hash_value == pattern_hash_value):
            return True
        else:
            add_idx = (s + h) if (s + h) <= M else (s + h - M)
            map_hash_value = (radix * (map_hash_value - map_col_hash[s] *
radix^(h-1)) + map_col_hash[add_idx]) % prime

    return False

// run check-map(Map) K times to check each map
```

6.

由於此題假設 collision不會發生，因此最糟糕的情況就是 K 張 Map 都沒有出現 desired pattern。
建立 pattern_col_hash 的時間複雜度為 $O(N*M*)。計算$pattern_h ash_v alue$的時間複雜度為*O(M)*
。建立$map_c ol_h ash$的時間複雜度為*O(N*M)。計算$map_h ash_v alue$的時間複雜度為O(M)
。橫向更新長度為M的$hashvalue$並且與$pattern_h ash_v alue$比對的時間複雜度為O(M)$。

因此最糟糕的情況下，K 張 Map 橫向都要更新與比對 M 次，時間複雜度為 $O(N*M+M) + O(K*(N*M+(M+M))) = O(K*M*N)$

Problem 3 - DSA Founder How-How

1.

依照 **Least Significant Digit(LSD) first sorting**， 依序從個位數到百位數用 counting sort 排序。

- 依個位數排序： (73, 4, 184, 504, 76, 47, 299, 9)
- 依十位數排序： (4, 9, 504, 47, 73, 76, 184, 299)
- 依百位數排序： (4, 9, 47, 73, 76, 184, 299, 504)

最終排序好的 sequence 為 (4, 9, 47, 73, 76, 184, 299, 504)

2.

`human algorithm`:
使用 **counting sort** 的 time complexity 為 $O(n)$，space complexity 為 $O(n)$

`pseudo code`:

```
N = number of elements in input array
K = total number of possible label value

CountingSort(array input, array output, K, N):
    C[K] is an empty array
    // initialize C[K]
    for i = 1 to i = K:
        C[K] = 0
    // build C[K] from input array
    for i = 1 to i = N:
        C[input[i]] = C[input[i]] + 1
    // accumulate C[k]
    for i = 2 to i = K:
        C[i] = C[i] + C[i-1]
    // insert vlaue to output[N] step by step
    for i = N to i = 1:
        output[C[input[i]]] = input[i]
        C[input[i]] --
    return output
```

3.

`human algorithm`:
使用一個 stack，當 stack 為空時，push s[i] 進入 stack，當 stack 不為空時，比較 s[i] 與 stack.top，若 s[i] > stack.top，依照本題性質， s[i+1] ~ s[N]必不小於 stack.top，表示 stack.top 為當前最小值，便把 stack.top pop 出來排進 array 裡；若 s[i] <= stack.top，則 push s[i] 進入 stack。 最終處理完 s[1] ~ s[N] 後，將 stack 中的元素依序 pop 出來排進 array。

`pseudo code`:

```


stack is an empty stack
N is length of sequence

Sort(array s):
    k = 1
    for i = i to i = N:
```

```
        if stack is empty:
            stack.push(s[i])
        else:
            while(stack is not empty and s[i] > stack.top):
                s[k] = stack.pop
                k += 1
            stack.push(s[i])
    // insert remain element to array s
    while(stack is not empty):
        s[k] = stack.pop
        k += 1
    return s
```

`complexity`:
對於每個 element 都會經歷一次 stack.push 與一次 stack.pop，因此 time complexity 為 $O(N)$， stack 最多會用到 N 個額外的空間存 elements，因此 space complexity 為 $O(N)$

4.

`human algorithm`:
使用兩個 stacks， 並且維護這兩個 stacks 的關係為: 當兩 stacks 都不為空時必須維持 stack_1.top < stack_2.top。首先 push s[1] 進 stack_1，接下來若 s[i] <= stack_1.top，則 push s[i] 進 stack1；若 s[i] > stack_1.top，依照此題性質，s[i+1] ~ s[N] 都不會比 s[i] 大，將 push s[i] 進 stack_2，如此處理完s[1]~s[N] 後兩 stacks 都會呈遞增排列，再將兩 stack 中的 elements 依大小由小到大排入 array。

`pseudo code`:

```
stack_1 and stack_2 are empty stacks
N is length of sequence

Sort(array s):
    stack_1.push(s[1])
    for i = 2 to i = N:
        if s[i] > stack_1.top:
            stack_2.push(s[i])
        else:
            stack_1.push(s[i])
    k = 1
    while (k<=N):
        if stack_1 is empty:
            s[k] = stack_2.pop
        else if stack_2 is empty:
            s[k] = stack_1.pop
        else if stack_1.top < stack_2.top
            s[k] = stack_1.pop
        else
            s[k] = stack_2.pop
        k += 1
    return s
```

```
```

complexity:
對於每個 element 都會經歷一次 stack.push 與一次 stack.pop，因此 time complexity 為 $O(N)$， 2 個 stacks 最多會用到 N 個額外的空間存所有 elements，因此 space complexity 為 $O(N)$。

5.

human algorithm: 使用 K-1 個 stacks， 並且維護這 K-1 個 stacks 的關係為: 當這 K-1 個 stacks 都不為空時必須維持 stack_1.top < stack_2.top < ... < stack_K-1.top 。首先 push s[1] 進 stack_1，接下來從 j = 1 到 j = K-1 依序搜尋，若 s[i] <= stack_j.top 或 stack_j為空，則 push s[i] 進 stack_j，如此處理完s[1]~s[N] 後這 K-1 個 stacks 都會呈遞增排列，再將這 K-1 個 stacks 中的 elements 依大小由小到大排入 array。

```
stack_1, stack_2, ... stack_K-1 are empty stacks
N is length of sequence

Sort(array s):
    stack_1.push(s[1])
    for i = 2 to i = N:
        for j = 1 to j = K-1:
            if stack_j is empty or s[i] <= stack_j.top
                stack_j.push(s[i])
                break
    k = 1
    while (k<=N):
        min = inf, minIdx = 0
        for j = 1 to j = K-1:
            if stack_j is not empty and stack_j.top < min:
                min = stack_j.top
                min_Idx = j
        s[k] = stack_minIdx.pop
        k += 1
    return s
```

complexity:
對於每個 element 經歷一次 stack.push 與一次 stack.pop 最多都會經過 K-1次的比較，因 K 為一常數，因此 time complexity 為 $O(KN) = O(N)$， K-1 個 stacks 最多會用到 N 個額外的空間存所有 elements，因此 space complexity 為 $O(N)$。