DSA Homework 2

B00401062 羅文斌

Problem 1.1

a. [2 1 3 5 4] > FLIP(2) > [1 2 3 5 4] > FLIP(5) > [4 5 3 2 1] > FLIP(2) > [5 4 3 2 1] > FLIP(5) > [1 2 3 4 5]. The pancake number = 4.
b. [5 4 1 2 3] > FLIP(5) > [3 2 1 4 5] > FLIP(3) > [1 2 3 4 5]. The pancake number = 2.
c. [1 4 3 2 5] > FLIP(4) > [2 3 4 1 5] > FLIP(3) > [4 3 2 1 5] > FLIP(4) > [1 2 3 4 5]. The pancake number = 3.

Problem 1.2

One thing to clarify is that the sequence with the maximum number of FLIPs under this algorithm does not necessarily mean that it must have the largest pancake number.
The maximun number of FLIPs this algorithm can yield is 2·(# for loop) = 2·(N-1). This occurs when the maximum is always not in place in each loop, i.e. i ≠ p. To achieve the maximum number of FLIPs, we can re-flip the sorted sequence with the following pseudocode:

```
for (let i = 2; i <= N; i++) {
    FLIP(A, i);
    FLIP(A, p);
}
```

If we arbitrarily choose p to be 1, given the original array A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] with N = 10, the result of A at each step is as follows:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] > FLIP(2) > FLIP(1) > [2, 1, 3, 4, 5, 6, 7, 8, 9, 10] > FLIP(3) > FLIP(1) >
[3, 1, 2, 4, 5, 6, 7, 8, 9, 10] > FLIP(4) > FLIP(1) > [4, 2, 1, 3, 5, 6, 7, 8, 9, 10] > FLIP(5) > FLIP(1) >
[5, 3, 1, 2, 4, 6, 7, 8, 9, 10] > FLIP(6) > FLIP(1) > [6, 4, 2, 1, 3, 5, 7, 8, 9, 10] > FLIP(7) > FLIP(1) >
[7, 5, 3, 1, 2, 4, 6, 8, 9, 10] > FLIP(8) > FLIP(1) > [8, 6, 4, 2, 1, 3, 5, 7, 9, 10] > FLIP(9) > FLIP(1) >
[9, 7, 5, 3, 1, 2, 4, 6, 8, 10] > FLIP(10) > FLIP(1) > [10, 8, 6, 4, 2, 1, 3, 5, 7, 9]

Problem 1.3

Let $A_k = [a_1, a_2, \ldots, a_{k-1}, a_k]$ whose pancake number is denoted as P_k.
The pseudocode in Problem 1.2 can be simplified as follows:

```
function Pancake_Sort(A, N){
    for (let i = N; i >= 2; i--) {
        if (is_sorted(A)) break;
        let p = max_index(A[1 ... i]); // find the index of max
        if (p == i) continue;
        FLIP(A, p);
        FLIP(A, i);
    }
}
```

a. The pseudocode given above flips the largest number in the sequence to the last index in the unsorted region each for loop. Intuitively, this algorithm will lead us to the sorted sequence. Suppose A_N is a permutation of $[a_1, a_2, \ldots, a_{N-1}, a_N]$, where $a_1 < a_2 < \ldots a_{N-1} < a_N$. Our hypothesis is that, at the k-th iteration, i.e. i = N - k + 1, every number bigger than a_{N-k} is sorted in the back of the sequence, i.e. $[\ldots, a_{N-k+1}, \ldots, a_{N+1}, a_N]$. This can be proved by induction.
   - At the 1st iteration, the maximum of A_N, i.e. a_N, will be chosen to be placed at the end of the sequence. Then, a_N is itself sorted. Therefore, the hypothesis holds for the 1st iteration.
   - Suppose at the (k-1)-th iteration, the hypothesis also holds, such that $[\ldots, a_{N-k+2}, \ldots, a_{N+1}, a_N]$, where every number bigger than a_{N-k+1} is sorted. i.e. suppose the hypothesis holds for the (k-1)-th iteration.
   - At the k-th iteration, the maximum of $[a_1, a_2, \ldots, a_{N-k+1}]$, i.e. a_{N-k+1} will be chosen to be place at the end of the unsorted sequence and yields the sequence $[\ldots, a_{N-k+1}, a_{N-k+2}, \ldots, a_{N+1}, a_N]$. As we can see, every number bigger than a_{N-k} is sorted. Therefore, the hypothesis also holds for the k-th iteration.

   Conclusion: The hypothesis is that every number bigger than a_{N-k} is sorted in the back of the sequence at the k-th iteration. It turns out that the hypothesis holds for every $k \geq 1$. After all of the iterations, a sorted sequence $[a_1, a_2, \ldots, a_{N-1}, a_N]$ is guranteed. This is a valid algorithm (pseudocode).
   After proving the algorithm is correct, we are going to examine the maximum number of FLIPs get executed, which is 2·(# for loop) = 2·(N-1). Every sequence is guaranted to be sorted after 2·(N-1) FLIPs, even in the worst case. Therefore, we can say the minimum number of FLIPs required to sort the sequence must be ≤ 2·(N-1), i.e. $P_N \leq 2 \cdot (N-1)$.
b. Our hypothesis is that there exists at least one permutation of 1 to N whose pancake number is at least $N-1$. This can be proved by induction.
   - When $k = 1$, $A_1 = [1]$ is itself sorted. The pancake number $P_1 = 0 \geq 1 - 1$. Therefore, the hypothesis holds for $k = 1$.
   - Suppose when $k = N - 1$, let $A_{N-1} = [a_1, a_2, \ldots, a_{N-1}]$ be a permutation of $[1, 2, \ldots, N-1]$, such that the pancake number $P_{N-1} \geq N - 2$, i.e. suppose the hypothesis holds for $k = N - 1$.
   - When $k = N$, we can always find a permutation of $[1, 2, \ldots, N]$, which is $A_N = [N, a_{N-1}, \ldots, a_2, a_1]$. Since N is at the first index, it always costs at least one step to flip it to the last. The minimum number of FLIPs to sort A_N can be achieved through the following steps: $[N, a_{N-1}, \ldots, a_2, a_1]$ > FLIP(N) > $[a_1, a_2, \ldots, a_{N-1}, N]$. After N is in place, we can sort the first $N-1$ numbers, i.e. A_{N-1} in P_{N-1} FLIPs based on the case $k = N - 1$. Therefore, $P_N = P_{N-1} + 1 \geq N - 1$, i.e. the hypothesis also holds for $k = N$.

   Conclusion: The hypothesis that there exists at least one permutation of 1 to N whose pancake number is at least $N-1$ always holds for $N \geq 1$.

Problem 2.1

p = P % 13 = 87 % 13 = 9.
T = 287471356248. A window of length 2 (the length of p = 87) is chosen. t = [28, 87, 74, 47, 71, 13, 35, 56, 62, 24, 48] % 13 = [2, 9, 9, 8, 6, 0, 9, 4, 10, 11, 9].
The first hit (blue) is a valid match, i.e. 87 = 87. The second, third and fourth hit (red) are spurious hits, since 74 ≠ 87, 56 ≠ 87, 48 ≠ 87. The number of valid matches and spurious hits are 1 and 3, respectively.

The pseudocode of the KMP prefix function is as follows. It will be used in Problem 2.2 and 2.3. Notice the indexing system in all of the following pseudocodes is the same as `C` language, i.e. starts from 0.

```
1  function Prefix_Function(P) {
2      let pi = new Array(P.length);
3      pi[0] = 0;
4      q = 0;
5      for (let i = 2; i <= P.length; i++) {
6          while (q > 0 && P[q + 1] != P[i]) q = pi[q];
7          if (P[q + 1] == P[i]) q = q + 1;
8          pi[i] = q;
9      }
10     return pi;
11 }
```

Problem 2.2

```
1  function Cyclic_Matcher(T, T0) { // based on KMP algorithm
2      let TT = T + T; // concatenate 2 Ts together
3      let pi= Prefix_Function(T0);
4      let q = 0;
5      for (let i = 1; i <= TT.length; i++) {
6          while (q > 0 && T0[q + 1] != TT[i]) q = pi[q];
7          if (T0[q + 1] == TT[i]) q = q + 1;
8          if (q == T0.length) return true;
9      }
10     return false;
11 }
```

The algorithm provided above tweaks KMP algorithm a little bit to match cyclic rotations. First, we concatenate two `T`s together to `TT`, as shown in line 2. If `T` is a cyclic rotation of `T0`, we can find a match of `T0` in `TT`. Based on our knowledge that KMP is a linear-time algorithm: $\Theta(|\mathbf{text}| + |\mathbf{pattern}|)$, the pseudocode given above is also linear, more precisely, $\Theta(2|T| + |T_0|) = \Theta(3|T|) = \Theta(|T|)$, where $|T| = |T_0|$.

Problem 2.3

```
1  function Max_Repetition(X) {
2      let pi = Prefix_Function(X);
3      let m = X.length;
4      let r = m - pi[m];
5      return r;
6  }
```

`pi[m]` is the last value of the prefix table, and it will roll us back to the last occurence of the smallest repetitive unit. By substracting `pi[m]` from `m`, we can get the length of the smallest repetitive unit. The pseudocode given above is also a linear-time algorithm. All operations other than `Prefix_Function()` take constant time. And `Prefix_Function()` takes $\Theta(|\mathbf{pattern}|) = O(|X|)$, which is linear.

Problem 2.4

P = 9487, T = 947892879487. In the graph column, blue denotes a match, red denotes a mismatch, and gray denotes processed (ignored) characters.

- **KMP algorithm:**
  The prefix table of the 9487 is [0, 0, 0, 0]. So we know whenever a mismatch is incurred, the pointer pointing to P will roll back to the front.

Step	Graph	Description
1	947892879487 9487	A match. Both pointers move to the next position.
2	947892879487 9487	A match. Both pointers move to the next position.
3	947892879487 9487	A mismatch. The pointer pointing to T moves to the next position, and the pointer pointing to P rolls back to the front.
4	947892879487 9487	A mismatch. The pointer pointing to T moves to the next position, and the pointer pointing to P stays at the front.
5	947892879487 9487	A match. Both pointers move to the next position.
6	947892879487 9487	A mismatch. The pointer pointing to T moves to the next position, and the pointer pointing to P rolls back to the front.
7	947892879487 9487	A mismatch. The pointer pointing to T moves to the next position, and the pointer pointing to P stays at the front.
8	947892879487 9487	A mismatch. The pointer pointing to T moves to the next position, and the pointer pointing to P stays at the front.
9	947892879487 9487	A match. Both pointers move to the next position.
10	947892879487 9487	A match. Both pointers move to the next position.
11	947892879487 9487	A match. Both pointers move to the next position.
12	947892879487 9487	A complete match. Matching finishes.

- **Boyer-Moore (BM) algorithm:**
  BM matches the pattern P and the text T from the back. Whenever a mismatch is met, BM has to decide whether to use the **good suffix rule** or the **bad character rule** based on which rule yields greater shift of P to the right.
  The **good suffix rule** tells us upon a mismatch, we have to try to keep the good suffix matched. Given a matched suffix s, here are the steps to go through.
  1. We first search for any matches of s in P that satisfies the following 2 rules (1) the rightmost occurrence, but (2) not a suffix of P, and then shift P to match them.
  2. If no such substring exists in P, we then try to look for the longeset prefix in P that is also a suffix of T, and then shift P to match them.
  3. If the above two steps do not give us any match, we then shift P to the right past the end of s.

  The **bad character rule** tells us that upon a mismatched character c, we look for if there is any occurrence of c left to the mismatch in P, and them shift P to match them. If no such c exists in P, we then shift P to the right past the end of c.

Step	Graph	Description
1	947892879487 9487	A mismatch. The good suffix rule is not applicable since there is no matching suffix. The bad character rule willl shift P to the right by 1 position to match 8.
2	947892879487 9487	A mismatch. The good suffix rule is not applicable since there is no matching suffix. The bad character rule willl shift P to the right by 3 positions to match 9.
3	947892879487 9487	A match. Both pointers move to the left by 1 position.
4	947892879487 9487	A match. Both pointers move to the left by 1 position.
5	947892879487 9487	A mismatch. The good suffix rule wlll shift P to the right by 4 position past the end of the good suffix 87. The bad character rule will only shift P to the right by 2 positions past the end of the bad character 2. 4 > 2, so the good suffix rule is applied.
6	947892879487 9487	A match. Both pointers move to the left by 1 position.
7	947892879487 9487	A match. Both pointers move to the left by 1 position.
8	947892879487 9487	A match. Both pointers move to the left by 1 position.
9	947892879487 9487	A complete match. Matching finishes.