Data Structure and Algorithm
Homework #2 Solution

TA email: dsa1@csie.ntu.edu.tw

Problem 1. Pancake (30%)

1.1. (9%)

(a) (3%) FLIP 4 times: FLIP(2) FLIP(5) FLIP(2) FLIP(5)

(b) (3%) FLIP 2 times: FLIP(5) FLIP(3)

(c) (3%) FLIP 3 times: FLIP(4) FLIP(3) FLIP(4)

1.2. (7%) There are many valid answers to this question, and we give one as follows:

10, 2, 8, 6, 4, 3, 5, 7, 9, 1

1.3. (14%)

(a) (7%) The pseudo code provided in problem 1.2 can sort any sequence of N pancakes in $2N - 2$ times. The outer for loop iterates $N - 1$ times, and calls FLIP at most twice (in fact 0 or 2 time(s)) in each iteration, so the algorithm calls FLIP $(N - 1) \times 2 = 2N - 2$ times at most.

Brief proof of correctness: In the k-th iteration, the algorithm locates the k-th largest number in the sequence. If it is not at the k-th position where it should be, we first FLIP it to the top (left) and then FLIP the whole sequence, so that it is now at the bottom (right). In other words, each time we FLIP the currently largest *unsorted* pancake to the top and then FLIP it to the bottom. After at most $N - 1$ such operations, all the pancakes will be in order.

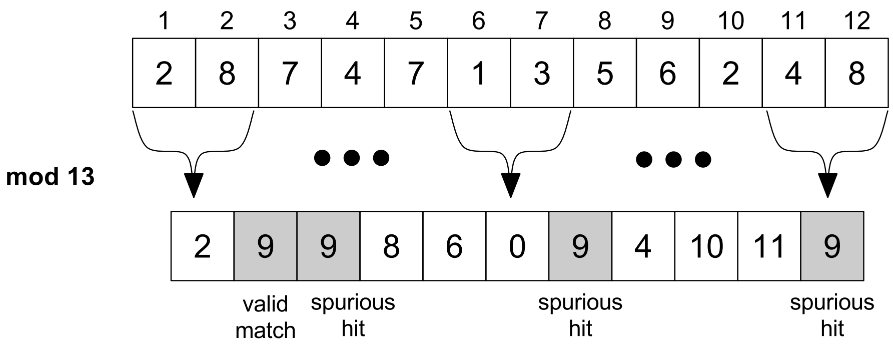
Note that there is no need to sort the smallest pancake as the last step, since there is only one possible position for it after all the other $N - 1$ elements have been moved to their respective positions.

(b) (7%) We define the *adjacency* $adj(S)$ of sequence S to be the total number of adjacent pancakes in the sequence such that no pancake has size between them. For example, $adj([1, 2, 3, 4]) = 3$ and $adj([1, 3, 2, 4] = 1$, where the adjacent pancake pair is (3, 2).

Each FLIP can increase the adjacency of a sequence by 1 at most, and it's clear that a sorted sequence of length N is of adjacency $N - 1$. Therefore, if an initial sequence S satisfies $adj(S) = 0$, it must be FLIPPED at least $N - 1$ times to increase its adjacency to $N-1$, i.e. the pancake number of S is at least $N-1$. For $n = 2$, [2, 1] needs one FLIP. For $n = 3$, [1, 3, 2] needs two FLIPs. For $n \geq 4$, $[2\lceil \frac{n}{2} \rceil - 1, 2\lceil \frac{n}{2} \rceil - 3, \ldots, 1, 2\lfloor \frac{n}{2} \rfloor, 2\lfloor \frac{n}{2} \rfloor - 2, \ldots, 2]$ has adjacency 0.

***Problem* 2.** String Matching (20%)

2.1. (4%)



$87 \equiv 9 \pmod{13}$

valid match: 1

spurious hit: 3

2.2. (3%)

```
Algorithm(T,T0)
    if len(T)!=len(T0):
        return False
    target_str = T+T
    if KMP(target_str,T0) finds strings matching:
        return True
    else:
        return False
```

We know if the length of T is different from the length of T_0, the two strings are not cyclic rotations of each other. Next, let string TT be the text to be search, and T_0 be the pattern. It's obvious that if T_0 occurs in TT, then the two given strings are cyclic rotations of each other. Because the time complexity of *Knuth Morris Pratt* algorithm is linear time, and the complexity of string concatenation is also linear time, the whole algorithm is linear time.

2.3. (5%)

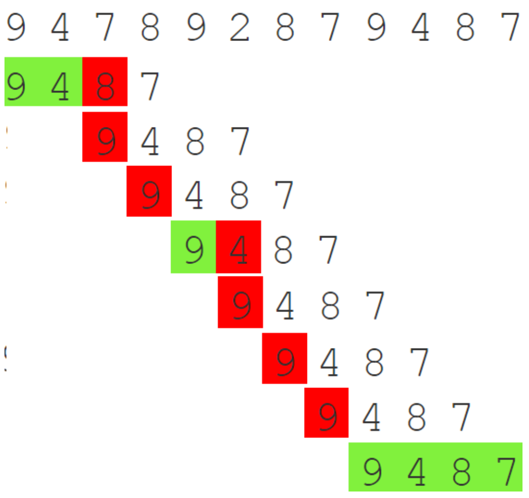
```
repetition(X, m):
    pi = Compute_Prefix_Function(X)
    k = m - pi[m]
    if m mod k == 0:
        r = m / k
        return r
    else:
        return 1
```

The running time is $\Theta(m)$ for computing prefix π.

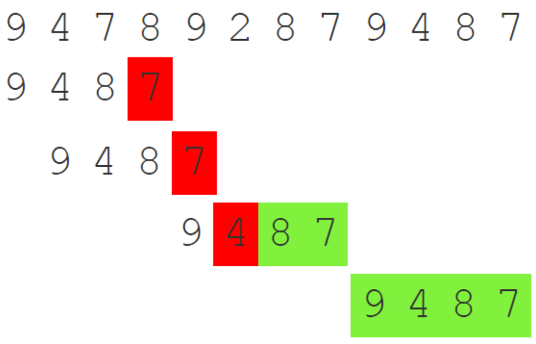
Brief explanation: Let $k = m - \pi[m]$. Suppose k not dividing m, we can only have $r = 1$ that $Y = X$. If $m \mod k = 0$, we know that k is the length of Z such that $X = Z^n$. Since $\pi[m] = m - k = |Z^n| - |Z|$, $\pi[m-k]$ would be $(m-k) - k = |Z^{n-1}| - |Z|$. By keeping this step, we get $\pi[m-(n-2)k] = \pi[2k] = (m-(n-2)k) - k = k$. Which implies that Z is the shortest string satisfying $X = Z^n$. And $Z = Y$, the value n equals to r.

2.4. (8%)

- *KMP* (8 rounds)



- *Boyer-Moore* (4 rounds)



Step 1: The Bad Character Rule (because 8 appears in 9487, shift 1 character)

Step 2: The Bad Character Rule (because 9 appears in 9487, shift 3 characters)

Step 3: The Good Suffix Rule (if you use The Bad Character Rule, you can only shift 2 characters. But if you use The Good Suffix Rule, you can shift 4 characters)

Step 4: match

We can find that the key feature of the *Boyer-Moore* algorithm is to match on the tail of the pattern rather than the head, and try to skip along the text in jumps of multiple characters. In this example, because there is no use by using prefix function, and KMP algorithm tries to match on the head of the pattern, the simulation is just about native string matching.
In KMP algorithm, we need to precompute the prefix function. In *Boyer-Moore* algorithm, we need to precalculate two tables to find the maximum of the shifts calculated by the two rules.

***Problem* 3.** Interesting String Pair (Programming problem) (25%)

First, we should find several interesting values among given pair of substring. From the definition of interesting value, $V(S,T) = max(occ(S,T), occ(T,S))$, we can actually calculate $occ(S,T)$ only. If we can solve $occ(S,T)$ for Q pairs in $O(f)$, we can swap S, T and those corresponding indices of substring and calculate $occ(S,T)$ again which also costs $O(f)$.

Following pseudo-code for clarifying:

```
Solve(S,T,Q,i1,j1,i2,j2):
  ans = [0] * Q
  for idx in 0..(Q-1):
    ans[idx]=max(ans[idx],
                 occ(S[i1[idx]:j1[idx]], T[i2[idx]:j2[idx]]))
  swap(S,T)
  swap(i1,i2)
  swap(j1,j2)
  for idx in 0..(Q-1):
    ans[idx]=max(ans[idx],
                 occ(S[i1[idx]:j1[idx]], T[i2[idx]:j2[idx]]))
  return ans
```

Thus, we only focus on calculating $occ(S,T)$ in the following discussion.

1. For 20% points, we can directly compare each pair of substring naively which costs $O(Q|S||T|)$. Pseudo-code given as follow:

```
occ(S,T):
  oc = 0
  for st in 0..(|T|-|S|):
    if S == T[st:st+|S|-1]
      oc += 1
  return oc
```

2. For 80% points, we should calculate occurences of each pair in linear time with KMP or other similar algorithm. KMP has been taught during class. Only pseudo-code given as follow:

```
occ(S,T):
  fl = [-1] * |S|
  ptr = -1
  for idx in 1..(|S|-1):
    while ptr >= 0 and not (S[ptr+1] == S[idx]):
      ptr = fl[ptr]
    if S[ptr+1] == S[idx]:
      ptr += 1
    fl[idx] = ptr
  oc = 0
  ptr = -1
  for idx in 0..(|T|-1):
    while ptr >= 0 and not (S[ptr+1] == T[idx]):
      ptr = fl[ptr]
    if S[ptr+1] == T[idx]:
      ptr += 1
    if ptr == |S|-1:
      oc += 1
      ptr = fl[ptr]
  return oc
```

3. For 100% points, a constraint is very important: **either** $i_1 = 0, j_1 = |S| - 1$ **or** $i_2 = 0, j_2 = |T| - 1$. (Remember that we only focus on $occ(S, T)$).

   - If $i_1 = 0, j_1 = |S| - 1$, we are wondering how many times does the whole string S occur in the substring $T[i_2 : j_2]$. Thus, we can precompute the occurrences of S in T. Then, we can find out the number of occurrence within a range with prefix sum. Pseudo-code given as follow:

```
precompute(S,T):
  oc = [0] * |T|
  fl = [-1] * |S|
  ptr = -1
  for idx in 1..(|S|-1):
    while ptr >= 0 and not (S[ptr+1] == S[idx]):
      ptr = fl[ptr]
    if S[ptr+1] == S[idx]:
      ptr += 1
    fl[idx] = ptr
  ptr = -1
  for idx in 0..(|T|-1):
    while ptr >= 0 and not (S[ptr+1] == T[idx]):
      ptr = fl[ptr]
    if S[ptr+1] == T[idx]:
      ptr += 1
    if ptr == |S|-1:
      oc[idx]=1
      # occurrence marked at the last matched character
      ptr = fl[ptr]
    if idx > 0:
      oc[idx] += oc[idx-1] # prefix sum
  return oc
Solve(S,T,Q,i1,j1,i2,j2):
  ...
  oc = precompute(S,T)
  for idx in 0..(Q-1):
    if i1[idx] == 0 and j1[idx] == |S|-1:
      if i2[idx]+|S|-1 <= j2[idx]:
        res = oc[j2[idx]] - oc[i2[idx]+|S|-2]
        ans[idx] = max(ans[idx], res)
  ...
```

- If $i_2 = 0, j_2 = |T| - 1$, we are wondering how many times does a substring of S occur in the whole string T. In this case, we should take advantage of the given hints. First, we concatenate string S and T as $S\$T$ where \$ is a character not being any of the lowercase English letters. The reason to insert a character \$ will be explained. Then, if we can sort suffixes of $S\$T$, those suffixes with common prefix will be neighbors (as we know, prefix of suffix is actually a substring!). The main concept is to binary search on the sorted suffixes and find the range where all of those suffixes with a prefix equal to our target string.

  For example, if $S = aba$, $T = ababa$, we are interested in $occ(S, T)$. Then, we can concataenate them as $aba\$ababa$. Then, sort its suffixes as:

start index	suffix
8	a
6	aba
4	$ababa$
0	$aba\$ababa$
2	$a\$ababa$
7	ba
5	$baba$
1	$ba\$ababa$
3	$\$ababa$

  Then, considering the original S which is now starting at 0 in the concatenated string. We can now binary search on sorted array and find start from 2nd sorted suffix to 4th sorted suffix are all matched aba as their prefix. Thus, we can use prefix sum to mark which suffix is original belong to T and find the exact occurrence out.

  If we didn't insert \$ between S and T, the suffix of S and prefix of T will directly be concatenated and may match some substring we are interested and cause extra occurrence leading to Wrong Answer.

  Let's now focus on how to implement above concept:

  (a) Sort the concatenated string $S\$T$

  Claim: Suppose we can sort suffixes for first 2^k characters, we can also sort them for first $2^{(k+1)}$ characters.

  To compare two string with $2^{(k+1)}$ characters, if their first 2^k characters are different, we can compare them by the order of first 2^k characters. Otherwise, we can compare them by the order of last 2^k characters.

Let's consider sort suffixes of *ababa* for more explanation.

To sort suffixes of *ababa* for first 1 character, we can directly sort them by the order of alphabet:

start index	suffix	order
0	*ababa*	1
2	*aba*	1
4	*a*	1
1	*baba*	2
3	*ba*	2

Thus, we already sorted some suffixes except first three and last two suffixes. Then, we already know the order of each suffix for first 2^0 character. To sort each suffix for first 2^1 characters. We first list out the order of first 2^0 character and last 2^0 character and sort each suffix by these order:

start index	suffix	first 2^0 order	last 2^0 order
0	*ababa*	1	2
2	*aba*	1	2
4	*a*	1	0
1	*baba*	2	1
3	*ba*	2	1

$\Rightarrow$

start index	suffix	first 2^0 order	last 2^0 order	order
4	*a*	1	0	1
0	*ababa*	1	2	2
2	*aba*	1	2	2
1	*baba*	2	1	3
3	*ba*	2	1	3

Now, we already sorted suffixes for first 2^1 characters. Then, let's sort them for first 2^2 characters:

start index	suffix	first 2^1 order	last 2^1 order
4	*a*	1	0
0	*ababa*	2	2
2	*aba*	2	1
1	*baba*	3	3
3	*ba*	3	0

$\Rightarrow$

start index	suffix	first 2^1 order	last 2^1 order	order
4	a	1	0	1
2	aba	2	1	2
0	$ababa$	2	2	3
3	ba	3	0	4
1	$baba$	3	3	5

Now, we already sorted suffixed of *ababa*. With this method, we can sort suffixes of given string in $O(|S| \lg |S|)$ (If sort with radix sort and counting sort) or $O(|S| \lg^2 |S|)$(If sort with merge sort or quick sort).

(b) Binary search out the matched range

Suppose we are interested in the substring of S from i_1 to j_1, we can first find out the index in sorted suffixes of S starting from i_1. Then, we need to match its prefix for at least $j_1 - i_1 + 1$ long. With the property of sorted suffixes, we know that farther from this suffix, the matched prefix is shorter. Thus, we can binary search on each side (before this one and after this one). The remaining problem is how to know the length of matched prefix. Actually, this can be calculated out with the help of order which we calculated during sorting suffixes. For i-th suffix and j-th suffix we can know the length of matched prefix with a binary serach method as follow:

```
order(i,k):
  # order of suffix starting from i-th character
  # for 2^k characters
  return order[i][k] # calculated during sorting
common_prefix(i,j):
  len = 0
  for k in (lg_|S|)..0:
    if order(i,k) == order(j,k):
      len += 2^k
      i += 2^k
      j += 2^k
  return len
```

Then, we can find the range of suffixes whose prefix matched as least $j_1 - i_1 + 1$. Then, the answer can be found with pre-computed prefix sum marked which suffix belongs to T in sorted array.

The whole algorithm implemented as above can be accepted within time complexity $O((Q + |S| + |T|) \lg(|S| + |T|))$ and may be accepted with careful implementation in time complexity $O((Q + |S| + |T|) \lg^2(|S| + |T|))$.

Actually, even without this constraint: ***either*** *$i_1 = 0, j_1 = |S| - 1$* ***or*** *$i_2 = 0, j_2 = |T| - 1$, this problem can be solved in $O((Q + |S| + |T|) \lg(|S| + |T|))$ but with a complicated data structure!*

Problem 4. Secret Code (Programming problem) (25%)

Overview

For this problem, we can simulate the cyphering procedure directly. The procedure consists of 3 steps: (1) Count the occurrences for each of the lower-case English letters. (2) Sort the letters using the method described. (3) Replacing letters to generate cypher.

Step 1

Input: S, a given string we'd like to process.

Output: *occurrences*, an array of 26 elements, each stores the number of occurrences of a lower-case English letter.

```
function counting(S):
    # initialize occurrences
    for i = 0 to 25:
        occurrences[i].letter = i
        occurrences[i].count  = 0
    for ch in S:
        if 0 <= (ch - 'a') <= 25:
            occurrences[ch - 'a'].count += 1
    return occurrences
```

Time Complexity: O($|S|$).

Step 2

Input: *occurrences*, the output of the previous step.

Output: *sorted_occurrences*, sorted with the method in homework description.

```
function sorting(occurrences):
    qsort(occurrences, compare)

function compare_occurrences(o1, o2):
    if o1.count != o2.count:
        return o2.count - o1.count
    return o1.letter - o2.letter
```

Time Complexity: O(1). (since we are sorting 26 elements, not an array of variable length.)

Step 3

Input: S and *sorted_occurrences*.

Output: *cypher*, the ultimate cyphered text we want.

```
function replacing(S, sorted_occurrences):
    # find the tail of non-0 occurrences
    for i = 0 to 25:
        if sorted_occurrences[i].count == 0:
            break
    tail = i
    # replace
    for i = 0 to (S.length - 1):
        for j = 0 to 25:
            if sorted_occurrences[j].letter == S[i] - 'a':
                break
        cypher[i] = sorted_occurrences[tail - j - 1]
    return cypher
```

Time Complexity: O($|S|$).

Summary

The above algorithm runs in O($|S|$), and can be implemented in C language effortlessly. However, this reference answer is not the only solution to this problem.