CS 580 Algorithm
Instructor: Prof. Yi Wu **Wei-Yen Day**
Problem Set 4 **wday@purdue.edu**

Collaborators

Thanks Weining Yang for a great discussion on problem 2(i).

Problem 1: Minimum Spanning Tree

i) Prim's algorithm with adjacency matrix

To implement Prim's algorithm with adjacency matrix, we only need to maintain a distance vector, *dist*, and pick up the next vertex with smallest distance based on *dist* vector. The algorithm looks like the following:

```
1: procedure Prim's Algorithm on adjacency matrix(G, V, E)
2:     s ← start
3:     V' ← {s}
4:     for each vertex i do
5:         dist[i] ← ∞
6:     end for
7:     while |V'| < |V| do
8:         min ← ∞
9:         for j in V − T do
10:            if G[s][j] < dist[j] then
11:                dist[j] ← G[s][j]
12:            end if
13:            if dist[j] < min then
14:                min ← dist[j]
15:                e ← (s, j)
16:                next ← j
17:            end if
18:        end for
19:        dist[next] ← ∞
20:        s ← next
21:        V' ← V' ∪ {s}
22:        E' ← E' ∪ {e}
23:    end while
24:    output T(V', E')
25: end procedure
```

This algorithm iterates until the tree size is $|V|$, and in each iteration it checks all the vertices in the set of remaining vertices, which involves at most $|V|$ vertices. So the running time of this algorithm should be $O(|V|^2)$.

ii) Minimum Spanning Tree algorithm

Similar to Prim's algorithm, we can prove this algorithm's correctness by proving a statement: in any iteration, each connected component is a minimum spanning tree of it's node set. Suppose the optimum tree for a connected component is T_{min}, which contains $e_1, e_2, \cdots, e_{n-1}, n = |V|$. Assume some i that $e_1, e_2, \cdots, e_{i-1}$ are in T_{min}. Consider a $e_i \notin T_{min}$, there is a circle C in it. So we replace e' with e_i and get a better spanning tree. Since we choose the edge with smallest weight from each connected component, for the same connected component it is just like we apply Prim's algorithm on this connected component. The union of these minimum spanning trees are also a minimum spanning tree. Thus, we prove the correctness of this algorithm.

Problem 2:

i)

We can first sort these n numbers (which takes $O(n \log n)$), and scan this new sequence s of numbers in $O(n^2)$ times to find if there's any $d_i + d_j + d_k = 0$ exists. Specifically, we first assign d_i to be the current iterated number of the new sequence, s_i, and let d_j and d_k be the smallest number of the remaining number (which are larger than d_i) s_{i+1} and the largest number s_n. While the summation over d_i, d_j, d_k is greater zero, we move d_k to be its previous number (which is smaller) and compute it again; while the summation is less than zero, we move d_j to be its next number (which is larger) and evaluate the summation. In this way, we only need to scan n^2 times, so the running time of this algorithm is $O(n \log n) + O(n^2) = O(n^2)$.

The algorithm is as following:

```
1: procedure ComputeTripleSum(n different integer number d_1, d_2, ..., d_n)
2:     sort d_1, d_2, ..., d_n, and get s = s_1, s_2, ..., s_n where s_1 < s_2 < ... < s_n
3:     for i from 1 to n do
4:         j ← i + 1
5:         k ← n
6:         d_i ← s_i, d_j ← s_j, d_k ← s_k
7:         while d_i + d_j + d_k ≠ 0 and j < k do
8:             if d_i + d_j + d_k < 0 then
9:                 j ← j + 1
10:                d_j ← s_j
11:            end if
12:            if d_i + d_j + d_k > 0 then
13:                k ← k − 1
14:                d_k ← s_k
15:            end if
16:            if d_i + d_j + d_k = 0 then
17:                return found
18:            end if
19:        end while
20:    end for
21:    return not found
22: end procedure
```

ii)

We can relate this problem to fast polynomial multiplication problem. First of all, let us set a polynomial function $A(x)$:

$$A(x) = x^{d_0} + x^{d_1} + x^{d_2} + \cdots + x^{d_n}$$

and we try to compute $C(x) = A(x) \cdot A(x) \cdot A(x)$. To check if any $d_i + d_j + d_k = 0$ exists, we only need to check if the coefficient of x_0 is 0, that is, if the constant term exists. Since we have a condition $-10n \leq d_i \leq 10n$, we can assume that fast polynomial multiplication can be done in $O(m \log m)$ times, where $m = 2(20n+1) - 1 = 40n+1$ for $A(x) \cdot A(x)$, and $m = 20n+1+40n+1-1 = 60n+1$ for $A(x) \cdot A(x) \cdot A(x)$. So the algorithm looks like the follows:

```
Compute B(x) = A(x)*A(x)
Compute C(x) = B(x)*A(x)
check if the coefficient of x^0 in C(x) is greater than 0
```

Since FFT can be done in $O(n \log n)$ times, we can conclude this algorithm is $O(n \log n)$.

Problem 3: More Computing Hamming Distance

i)

Suppose we have an algorithm, ComputeHammingDistance, that computes the hamming distance for alphabet size 2 with $O(n \log n)$ times, and the output is the hamming distance like the example given in the problem. Then for alphabet size k, we can run the algorithm k times given the input alphabet set as $\{c_k, \phi\}$ with each alphabet c_k and an empty character ϕ which doesn't match any other characters. For example, if $s = 12342321$ and $p = 1234$, we can run the algorithm 4 times. First time we give alphabet set $= \{1, \phi\}$ to the algorithm, and get the result as $d_1 = (3, 4, 4, 4, 4)$. Then for the second time we give $\{2, \phi\}$ (and later so on) and get $d_2 = (3, 4, 4, 3, 4)$. Third result is $d_3 = (3, 4, 4, 3, 4)$ and fourth result is $d_4 = (3, 4, 4, 4, 4)$. Then, the final result would be the sum of the complimentary on each result, that is,

$$d = \sum_{i=1}^{4} (4 - d_i) = (0, 4, 4, 2, 4)$$

The algorithm looks like the following algorithm. The running time would be $O(k \cdot n \log n)$.

```
1: procedure ComputeKHammingDistance(s, p, Σ)
2:     k ← |Σ|
3:     d ← (0, 0, ··· , 0)                                  ▷ initialize
4:     for j from 1 to k do
5:         Σ_j ← {Σ(j), φ}
6:         d_j ← ComputeHammingDistance(s, p, Σ_j)
7:         d ← d + k − d_j
8:     end for
9:     output d
10: end procedure
```

ii)

It is roughly the same idea as problem 1 in homework 3. Basically we only need to modify the part of building index from string p as a list for each character (so there are at most d items on each list), say l_c for character c's list, and check $|l_c|$ times for each round. For simplicity, we can use queue to implement the list for each character:

```
1: procedure ComputeHammingDistance(s, p)
2:     n ← s.length
3:     m ← p.length
4:     for i from 1 to m do                          ▷ Build inverted index on p
5:         pindex(p[i]).enqueue(i)
6:     end for
7:     for i from 1 to n do          ▷ Look over string s and count to corresponding position
8:         while pindex(s[i]) is not empty do
9:             pind ← pindex(s[i]).dequeue()
10:            pos ← i-pind+1
11:            if pos > 0 and pos ≤ n − m + 1 then
12:                match(pos) ← match(pos) + 1
13:            end if
14:        end while
15:    end for
16:    distance ← m - match                  ▷ Distance is complementary with match
17: end procedure
```

Since the length of each list is at most d, this algorithm checks at most d times in each iteration. As a result, the running time of this algorithm is $O(d \cdot n)$.

iii)

For this question, let us consider the case that $|\Sigma| \leq m$. We can combine the above two algorithms to get a better algorithm than $\Theta(mn)$. For simplicity, we name the algorithm in i) as Alg 1, and call the algorithm in ii) as Alg 2. From the above questions, we know that Alg 1 and Alg 2 give us a good running time on different scenarios:

- Alg 1: $O(k \cdot n \log n)$ when $|\Sigma| = k$
- Alg 2: $O(d \cdot n)$ when every character appears at most d times in p, given $d < m$

Observe that when each character appears at most d times, there are at most $\frac{m}{d} + 1$ distinct characters in pattern p. Given this observation, we can further choose a smaller set of alphabet, Σ', which contains the characters appeared in p plus 1 null character which doesn't appear in p. In other words, the size of Σ', $|\Sigma'|$, is $\frac{m}{d} + 2$. So when the size of Σ' is small enough, that is, d is large enough, we tend to choose Alg 1 as our algorithm. On the other hand, when d is small enough, $|\Sigma'|$ becomes large and we tend to choose Alg 2 to get the result. To be specifically, in the following condition, we choose Alg 1:

$$(\frac{m}{d} + 2) \cdot n \log n < d \cdot n \Rightarrow \log n < \frac{d^2}{m + 2d}$$

So this concludes:

```
if log(n) < d^2/(m+2d)
    choose Alg 1
else
    choose Alg 2
```

To prove the running time is better than $O(mn)$, we only need to prove this two algorithms are better than this complexity. For Alg 2, it's easy to say the running time is smaller than $O(d \cdot n) < O(mn)$ since we already know $d < m$. For Alg 1, we can do a simple proof on it:

$$\begin{aligned}
(\frac{m}{d} + 2) \cdot n \log n &< (\frac{m}{d} + 2) \cdot n \frac{d^2}{m + 2d} \text{ (the condition for Alg 1)} \\
&= \frac{mnd + 2nd^2}{m + 2d} \\
&= \frac{mn + 2nd}{\frac{m}{d} + 2} \\
&< \frac{mn + 2mn}{\frac{m}{m} + 2} \text{ (since } d < m\text{)} \\
&= \frac{3mn}{3} \\
&= mn
\end{aligned}$$

So we conclude that combining these two algorithms can get a better algorithm than $O(mn)$.