Fast Fourier Transforms

Alexander Bergholm
Department of Computer Science
University of British Columbia
alexander.bergholm@alumni.ubc.ca

Abstract

Fast Fourier Transform (FFT) is an algorithm that reduces the complexity of multiplication from O(N^2) to O($N \log N$). This improvement in time complexity makes it beneficial in numerous domains, predominantly, in digital processing of waveforms (audio, radar, sonar), brain signal conversions in EEGs as well as integer and polynomial multiplication.

1 Introduction

1.1 Fourier Transform

The Fourier Transform (FT) takes an input waveform as a function of time and and decomposes it into its original frequencies. It is a mathematical function that distinguishes separate waves from one another and is therefore capable of separating them. A continuous Fourier Transform is defined as follows.

$$\text{Fourier Transform (FT) } X(f) = \int_{-\infty}^{\infty} x(t)e^{-2\pi ift}dt$$
$$\text{Inverse Fourier Transform (IFT) } x(t) = \int_{-\infty}^{\infty} X(f)e^{2\pi ift}df$$

Where t is the time elapsed, f represents the frequency and $i = \sqrt{-1}$
The FT X(f) is defined as the spectrum of x, which describes the quantity of a specific frequency present within the waveform during the time measured. An IFT can be used to return to the time domain. In practice, data is collected on a finite time frame. It is therefore imperative to use Discrete Fourier Transforms.

1.2 Discrete Fourier Transform

The Discrete Fourier Transform is equivalent to performing the Fourier Transform on a finite set of sampled data.

$$\text{Discrete Fourier Transform (DFT) } X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi\frac{ikn}{N}}dt$$
$$\text{Euler's formula } e^{ix} = \cos x + i \sin x$$
$$\text{DFT w/ Euler's formula } X_k = \sum_{n=0}^{N-1} x_n \cdot [\cos 2\pi\frac{kn}{N} - i \cdot \sin 2\pi\frac{kn}{N}]$$
$$\text{Inverse Discrete Fourier Transform (IDFT) } x_n = \frac{1}{N}\sum_{n=0}^{N-1} X_k \cdot e^{2\pi\frac{ikn}{N}}dt$$

From the first DFT equation, we note that x(n) can be a real or complex sequence and $e^{-2\pi\frac{ikn}{N}}$ is a complex function. For each k we have N complex multiplications and N-1 additions from the summation. We also note that k=0,1,...,N-1. Therefore in calculating the DFT X_k we have N^2 complex multiplications and $N(N-1)$ complex additions. For large N, DFT computation can be incredibly slow.

1.3 Fast Fourier Transform

The Fast Fourier Transform is a Divide-and-Conquer algorithm that was originally used to compute the DFT in O(Nlog(N)) time. The author of the FFT is still debated, as there are multiple authors with unpublished work revolving around the FFT. The first officially recognized method is the Cooley-Tukey FFT Algorithm. It was derived from the DFT in the following manner.
We split the DFT into even and odd subsequences.
$X_k = \sum_{r=0}^{N/2-1} x_{2r} \cdot e^{-2\pi \frac{ik2r}{N}} + \sum_{r=0}^{N/2-1} x_{2r+1} \cdot e^{-2\pi \frac{ik(2r+1)}{N}} dt$
$X_k = \sum_{r=0}^{N/2-1} x_{2r} \cdot e^{-2\pi \frac{ik2r}{N}} + e^{-2\pi \frac{ik}{N}} \sum_{r=0}^{N/2-1} x_{2r+1} \cdot e^{-2\pi \frac{ik(2r)}{N}} dt$
Note: The exp on both is the same, and therefore significantly lowers the cost of computation. By performing a butterfly on the 2 DFT's of size N/2 we now have a computational cost of 2 DFTs * $(N/2)^2 + N$ = $O(N^2/2)$. But since we can continue to split the DFT into halves until N=1, a time complexity of O(NlogN) is achieved.
In a competitive programming aspect, the FFT is often used for polynomial multiplication at O(Nlog(N)) time. This can be calculated through 3 FFT's by
$A \cdot B = InverseDFT(DFT(A) \cdot DFT(B))$

2 Problem

2.1 An Aside

While the Introduction of the problem heavily emphasizes the background from which FFTs arise, this report will now take a small detour towards it's use in a competitive programming environment. While the background is essential to understanding the algorithm, it will now focus further on it's use as a time complexity improvement.

2.2 CodeForces 528D - Fuzzy Search

Reference: https://codeforces.com/problemset/problem/528/D
Description
We are given a string S and a string T. We wish to find, given some error threshold k, the amount of times T occurs within the string S. The error threshold k is defined by the amount of positions away from its original position in the desired string T. Meaning, if the desired string T = "ACGT", then for k=1 and the letter "A", the letter could be placed one position prior or further away. This would match with a string "AGCGT" or "GCAGT". For further information, refer to the reference above.
Input
The first line contains three integers $|S|$, $|T|$, k $(1 \leq |T| \leq |S| \leq 200000, 0 \leq k \leq 200000)$ — the lengths of strings S and T and the error threshold.

The second line contains string S.

The third line contains string T.

Both strings consist only of uppercase letters 'A', 'T', 'G' and 'C'.

Output
Print a single number — the number of occurrences of T in S with the error threshold k by the given definition.

Example Input
10 4 1
AGCAATTCAT
ACAT
Example Output
3

3 Solution

The following tasks need to be completed to successfully tackle this problem. Given a position i on the string S, we want to know whether each letter ('A', 'T', 'G' and 'C') matches at this position.

Further, we want to also consider this letter 'matched' if it exists within k positions of i. This is relatively easy to code with a bitmask! We create a bitmask for each letter above on S. When the letter is found in S at position i, we store $S_{bitmask}[i-k]=1$ and $S_{bitmask}[i+k+1]=-1$. We just explained that if a letter can be found in S at position i, then we only care that at positions i-k to i+k this letter can also be found. Once we have iterated through the entirety of S, we iterate through it again summing each $S_{bitmask}[i]+=S_{bitmask}[i-1]$. This sets all the regions where the letter can be found $\pm$ threshold k to greater than 1, while setting the unfound regions to 0. We then iterate once more to set these to 1 rather than greater than 1.
So we've found the bitmask of S. We still need to calculate the bitmask of each letter with respect to T. But fortunately, the letters of T do not move, as this is what we are wanting to match. So for each letter the bitmask of T will just be $T_{bitmask}=$ letter (1 for true, 0 for false). FFT's are then used to calculate the polynomial multiplication between bitmasks.

4 Implementation

Note: Due to a lack of time and poor decision making on what portion of this written report to prioritize, I was unable to code functions reverse, fft and multiply on my own. These are directly from https://cp-algorithms.com/algebra/fft.html.

```
// Attempt at solving https://codeforces.com/contest/528/problem/D
#include <bits/stdc++.h>
using namespace std;
int S; int T; int k;

using cd = complex<double>;
const double PI = acos(-1);

int reverse(int num, int lg_n) {
    int res = 0;
    for (int i = 0; i < lg_n; i++) {
        if (num & (1 << i))
            res |= 1 << (lg_n - 1 - i);
    }
    return res;
}

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n)
        lg_n++;

    for (int i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i, lg_n)]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
```

```
            x /= n;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}

vector<int> bitmask(char s[], char a) {
  // C Bitmask
  vector<int> c(S,0);
  for (int i=0; i<S; ++i) {
    if (s[i] == a) {
      c[max(i-k, 0)] += 1;
      if (i+k+1 < S) {
        c[i+k+1] -= 1;
      }
    }
  }
  for (int i=1; i<S; i++) {
    c[i] += c[i-1];
  }
  for (int i=1; i<S; i++) {
    c[i] = c[i] > 0 ? 1 : 0; // todo: check correctness
  }
  return c;
}
vector<int> Tbitmask(char t[], char a) {
  // T's Bitmask
  vector<int> c(T,0);
  for (int i=0; i<T; ++i) {
    if (t[i] == a) {
      c[i] = 1;
    }
  }
  reverse(c.begin(), c.end());
  return c;
}

int main(){
  // Input
  // |S|, |T|, k
  scanf("%d%d%d",&S,&T,&k);
  vector<int> matches(S,0);
  char s[S];
  char t[T];
  scanf("%s%s",&s,&t); // String S, T containing only characters 'A', 'T', 'G', 'C'
  // For each character (A,T,G,C) create a bitvector spanning left and right from each letter
  // A Bitmasks
```

```
  vector<int> a = bitmask(s, 'A'); // 1 where s[i] = A +/- k
  vector<int> Ta = Tbitmask(t, 'A'); // 1 where t[i] = A
  vector<int> res1 = multiply(a, Ta);
  // T Bitmasks
  vector<int> tarr = bitmask(s, 'T');
  vector<int> Tt = Tbitmask(t, 'T');
  vector<int> res2 = multiply(tarr, Tt);
  // G Bitmasks
  vector<int> g = bitmask(s, 'G');
  vector<int> Tg = Tbitmask(t, 'G');
  vector<int> res3 = multiply(g, Tg);
  // C Bitmasks
  vector<int> c = bitmask(s, 'C');
  vector<int> Tc = Tbitmask(t, 'C');
  vector<int> res4 = multiply(c, Tc);
  // Count #scenarios in which result == |T|
  int count = 0;
  for(int i=0; i<S; i++) {
    if (res1[i]+res2[i]+res3[i]+res4[i] == T) {
      count++;
    }
  }
  printf("%d",count);
    return 0;
}
```

5 Evaluation of Complexity

We have two types of bitmasks.
For $S_{bitmask}$ the function contains 3 separate for loops traversing S. Each component within the for loops takes O(1) time to complete. We calculate a bitmask for each letter ('A', 'T', 'G', 'C'). We therefore have $4 * O(3S)$ = $12 * O(S)$.

For each $T_{bitmask}$ the function iterates through T once. Within the for loop, all operations take O(1) time to complete. We then reverse the bitmask, which takes linear O(T) time. We calculate 4 of these masks, so the time complexity is $4 * O(2T)$ = $8 * O(T)$.

We have 4 polynomial multiplications that need to be performed.
A single multiply first copies the input vectors ($O(S) + O(T)$), then performs fft twice, then multiplies element-wise (which is upper-bounded by $(O(S) + O(T))$, then performs a third fft and then computes the result (which is also upper-bounded by $(O(S) + O(T))$).
We define n as the size of the vector input to the fft function. The fft function takes O(log(n)) to calculate the value of log(n), O(nlog(n)) to calculate the swapping; O(n) for passing through the array and log(n) within the reverse function. We now reach the main loop of the fft function.
The top-most loop has a size of O(log(n)). The first inner-loop has an initial size of n/2 and diminishes on every iteration of the top-most loop. The final loop has an initial size of 1 and grows to a maximum size of n/2. This loop therefore has a size of O(nlog(n)).
The fft function therefore takes a total of $O(2nlog(n) + log(n)) = O(nlog(n))$ Note: the inverse fft has one more traversal of size O(n) with O(1) operations.

Returning to the multiply function, we therefore have O(SlogS) + O(TlogT) + O(SlogS) time complexity for the three separate fft's.
The time complexity of multiply is therefore O(2SlogS + TlogT).
Since the multiply is the clear bottleneck here, the final time complexity of this solution is 4*O(2SlogS + TlogT).

References

Works Cited

Bekele, Amente. "Cooley-Tukey FFT Algorithms." *Carleton*, 2016, people.scs.carleton.ca/~maheshwa/courses/5703COMP/16Fall/FFT_Report.pdf.

"Bluestein's FFT Algorithm: Mathematics of the DFT." *DSP*, www.dsprelated.com/freebooks/mdft/Bluestein_s_FFT_Algorithm.html.

"But What Is the Fourier Transform? A Visual Introduction." *Youtube.com*, 2018, www.youtube.com/watch?v=spUNpyF58BY.

"Cooley–Tukey FFT Algorithm." *Wikipedia*, Wikimedia Foundation, 3 Mar. 2020, en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm.

"Discrete Fourier Transform." *Wikipedia*, Wikimedia Foundation, 30 Jan. 2020, en.wikipedia.org/wiki/Discrete_Fourier_transform.

"Fast Fourier Transform." *Fast Fourier Transform - Competitive Programming Algorithms*, 2020, cp-algorithms.com/algebra/fft.html.

"Fast Fourier Transform." *Wikipedia*, Wikimedia Foundation, 24 Mar. 2020, en.wikipedia.org/wiki/Fast_Fourier_transform.

"An Intuitive Discrete Fourier Transform Tutorial." *Practical Cryptography*, 2018, practicalcryptography.com/miscellaneous/machine-learning/intuitive-guide-discrete-fourier-transform/.

"Problem - D Fuzzy Search." *Codeforces*, 2020, codeforces.com/contest/528/problem/D.

"Rader's FFT Algorithm." *Wikipedia*, Wikimedia Foundation, 4 Oct. 2019, en.wikipedia.org/wiki/Rader%27s_FFT_algorithm.

VanderPlas, Jake. “Understanding the FFT Algorithm.” *Understanding the FFT Algorithm | Pythonic Perambulations*, 2013, jakevdp.github.io/blog/2013/08/28/understanding-the-fft/.

Xu, Simon, director. *Discrete Fourier Transform - Simple Step by Step*. *Youtube.com*, 2015, www.youtube.com/watch?v=mkGsMWi_j4Q.