Notes on FFT Problems

md.mottakin.chowdhury

July 2018

1 Problems

1.1 Problem 1

You are given two DNA sequences consisting of equal length of upto 10^5 characters consisting of alphabets A, T, G, C only. Find the i^{th} cyclic shift of one string for which maximum characters are matched between the two strings.

Solution: Let the text is s and pattern is t. For each letter in the alphabet, convert the strings into bit sequences, each position is either 1 or 0, meaning that the corresponding letter occurs in that position or not. Append s to itself and reverse t. Get two polynomials, $P(s+s)$ and $P(reversed(t))$ were the bits correspond to the coefficients. The $i+m-1$ coefficient of $P(s+s) \times P(reversed(t))$ gives the number of matches of the i^{th} cyclic shift of s and t. We do this for each letter and add up the result.

1.2 Problem 2

Similar to previous problem. You are given two DNA sequences S and T of length n and m, $m \leq n$. For each substring of lenght m of S, find the hamming distance of this substring with T. Print the minimum such distance.

Solution: For each letter in the alphabet, similarly build two polynomials with bits as coefficients, $P(S)$ and $P(reversed(T))$ and find $P(S) \times P(reversed(T))$. Now we iterate from $m-1$ to $n-1$ in the resultant polynomial, add the corresponding values to some auxiliary array aux. The answer is $m - max(aux)$.

1.3 Problem 3

We are given two arrays a and b. We have to find all possible sums $a[i]+b[j]$, and for each sum count how often it appears.

Solution: We construct for the arrays a and b two polynomials A and B. The numbers of the array will act as the exponents in the polynomial ($a[i] \Rightarrow x^{a[i]}$); and the coefficients of this term will by how often the number appears in the array.

Then, by multiplying these two polynomials in $O(nlogn)$ time, we get a polynomial C, where the exponents will tell us which sums can be obtained, and the coefficients tell us how often.

1.4 Problem 4

A thief has a knapsack. The knapsack can contain k objects. There are n kinds of products in the shop and an infinite number of products of each kind. The cost of one product of kind i is $a[i]$. The thief is greedy, so he will take exactly k products (it's possible for some kinds to take several products of that kind). Find all the possible total costs of products the thief can take into his knapsack.

Solution: Like the previous problem, we will build a polynomial and take its k^{th} power.

```
const int N=(1<<20);
VL find(VL a, int k)
{
        VL ret(1<<10);
        ret[0]=1;

        for(; k>0; k>>=1)
        {
                if(k&1) Multiply(ret,a,ret);

                // total cost of k objects won't be more than 1e6
                // we ensure power of 2.
                while(ret.size()>N) ret.pop_back();
                // of course we don't need occurrence and overflow issue
                FOR(i,0,ret.size()) if(ret[i]) ret[i]=1;

                Multiply(a,a,a);
                while(a.size()>N) a.pop_back();
                FOR(i,0,a.size()) if(a[i]) a[i]=1;
        }

        return ret;
}
```

1.5 Problem 5

You are given two grids s of size $(n \times m)$ and t of size $(r \times c)$. Now, find the upper-left most position of the grid s on which if you place the $(0, 0)$ of grid t, maximum amount of characters match. Output such position and number of characters match for each letter in the alphabet. Grid t should not cross boundary of s.

Solution: Flatten both the grids, using dummy character for flattening t.

```
void solve()
{
        string sflat="", tflat="";
        FOR(i,0,n) FOR(j,0,m) sflat+=s[i][j];

        FOR(i,0,n)
        {
                FOR(j,0,m)
                {
                        if(i>=r || j>=c) tflat+='x'; // Important
                        else tflat+=t[i][j];
                }
        }
        REVERSE(sflat);
        int sz=n*m;

        vi ga(sz), la(sz), gb(sz), lb(sz);

        convert(sflat,ga,'G');
        convert(sflat,la,'L');
        convert(tflat,gb,'G');
        convert(tflat,lb,'L');

        vi gout=mult(ga,gb);
        vi lout=mult(la,lb);

        int j=0, k=0, mxval=0, sx=0, sy=0;
        int grain=0, lives=0;

        FORr(i,sz-1,0) // Important too
        {
                if(j+r-1<n && k+c-1<m)
                {
                        if(gout[i]+lout[i]>mxval)
                        {
                                mxval=gout[i]+lout[i];
                                sx=j, sy=k;
                                grain=gout[i];
                                lives=lout[i];
                        }
                }

                k++;
                if(k==m)
                {
                        j++;
                        k=0;
                }
        }
        printf("Case #%d: %d %d %d %d\n", cases++, sx+1, sy+1, grain, lives);
}
```

1.6 Problem 6

Given array A, find the XOR sum of every continuous subsequence of A and determine the frequency at which each number occurs. Then print the number and its respective frequency as two space-separated values on a single line.

Solution: We take cumulative xor-sum of the given array in C. Now we use each value of C as an exponent of a polynomial and count of its occurrence as coefficients. We then apply Fast Walsh-Hadamard transform.

```
cin>>n;
int last=0, x;
in[0]=1;
FOR(i,1,n+1)
{
      cin>>x;
      x^=last;
      in[x]++;
      last=x;
}
fwt.self_convolution(in,N);
in[0]-=n;
ll mxval=*max_element(in,in+N);
FOR(i,0,N)
{
      if(mxval==in[i])
      {
            cout<<i<<" "<<mxval/2<<endl;
            break;
      }
}
```

1.7 Problem 7

You are given two bitstrings and queries. In each query, you are given two starting position and a length. You have to output the hamming distance between the two strings starting from the given two positions with given length.

Solution: Suppose we are given two strings S and T. We want to find all hamming distances between S and T if we place T in all positions of S without crossing the boundary. This can be solved easily by FFT.

```
vi hamming(string &s, string &t)
{
        vi a, b;
        int n=s.size(), m=t.size();
        FOR(i,0,n)
        {
                if(s[i]=='1') a.pb(1);
                else a.pb(-1);
        }
        FORr(i,m-1,0)
        {
                if(t[i]=='1') b.pb(1);
                else b.pb(-1);
        }

        vi c=mult(a,b), ret;
        for(int i=0; (i+m)<=n; i++)
                ret.pb(m-(c[i+m-1]+m)/2);

        return ret;
}
```

Now for the original problem, we do square-root decomposition. We take blocks from string T and multiply each block with S as mentioned above. Now while answering queries, we take each block of S that falls inside the current query and add the value of the corresponding position from the block's multiplication values. For the rest of the positions that do not fall inside this box, we can do linear checking. But depending on the constraints, we may need to apply more block-based operation to do this linear checking more efficiently. Here is the solution:

```
void solve()
{
        int m=t.size(), last=0;
        int blockcnt=(m-1)/blocksz;
        for(int i=0; i<m; i+=blocksz)
        {
            // find the hamming distance for each block with string s
                blocks[i/blocksz]=hamming(i,min(m-1,i+blocksz-1));
                last=min(i+blocksz-1,m-1);
                // prnt(last);
        }

        if(last!=m-1) blocks[blockcnt]=hamming(last,m-1);
        // Taken chunks of BLOCK from each position to answer
        // the linear check efficiently
        for(int i=0; i+BLOCK-1<s.size(); i++)
        {
                for(int j=i; j<=i+BLOCK-1; j++)
                {
                        if(s[j]=='1') chunks[i].set(BLOCK-1-j+i); // bitsets
                }
        }

        for(int i=0; i+BLOCK-1<t.size(); i++)
        {
                for(int j=i; j<=i+BLOCK-1; j++)
                {
                        if(t[j]=='1') chunkt[i].set(BLOCK-1-j+i);
                }
        }
}

// (sx,sy) - string s position, (tx,ty) - string t positions
// (sy-sx+1)==(ty-tx+1)
int findAns(int sx, int sy, int tx, int ty)
{
        int ret=0;
        int l=tx/blocksz+1;
        int r=ty/blocksz-1;
        if(l+1<r)
        {
                int pos1=sx+l*blocksz-tx;
                int pos2=sy-(ty-(r+1)*blocksz);
                // handle the left part out of the blocks
                ret=handle(sx,pos1-1,tx,l*blocksz-1);
                // handle the right part
                ret+=handle(pos2,sy,(r+1)*blocksz,ty);
                // handle the blocks
                ret+=handleBlocks(sx,sy,tx,l,r);
        }
        else
        {
                ret=handle(sx,sy,tx,ty);
        }
        return ret;
}

int handle(int sx, int sy, int tx, int ty)
{
        int ret=0, i, j;
        for(i=sx, j=tx; ; i+=BLOCK, j+=BLOCK)
        {
                if(i+BLOCK-1>sy) break;
                bitset<BLOCK> temp=chunks[i]^chunkt[j];
                ret+=temp.count();
        }
        if(i<=sy)
        {
                for(int k=i, l=j; k<=sy, l<=ty; k++, l++)
                        ret+=(s[k]!=t[l]);
        }

        return ret;
}
```

```
int handleBlocks(int sx, int sy, int tx, int l, int r)
{
        int ret=0;
        int pos=sx+l*blocksz-tx;
        for(int i=l; i<=r; i++)
        {
                if(pos+blocksz-1>sy) break;
                ret+=blocks[i][pos];
                pos+=blocksz;
        }
        return ret;
}
```

1.8 Problem 8

You're given a sequence s of N distinct integers. Consider all the possible sums of three integers from the sequence at three different indicies. For each obtainable sum output the number of different triples of indicies that generate it.

Solution: The code is self explanatory.

```
        scanf("%d", &n);
        vector<ll> a(N), b(N), c(N);
        FOR(i,0,n)
        {
                scanf("%d", &in[i]);
                in[i]+=offset; // make all integers non-negative
                a[in[i]]++; b[in[i]*2]++; c[in[i]*3]++;
        }
    // From all possible ways, subtract ways where 2 integers are on same index
    // and one is on different. This can be done in 3C2 ways because we are multiplying
    // three polynomials. While counting this subtraction, we subtracted all three indices
    // same case thrice, but we have to subtract once. So we add 2*c[i] where c[i] is the
    // case where all three are same. Finally, divide by 6 because each different combination // three in
        vector<ll> cube=mult(a,a);
        cube=mult(cube,a);
        b=mult(b,a);

        FOR(i,0,N) cube[i]=cube[i]-3LL*b[i];
        FOR(i,0,N)
        {
                cube[i]=(cube[i]+2*c[i])/6;
                if(cube[i]) printf("%d : %lld\n", i-offset*3, cube[i]);
        }
```

1.9 Problem 9

Given at most 10^5 sticks. Pick 3 sticks from there. What is the probability that it forms a triangle? The length of the sticks can be same, but indices will be different.

Solution: First, find the number of occurrences of different summations of pair of numbers taken from the input. In this case, we will only consider different indices and distinct pairs, i.e (1,2) and (2,1) will be counted twice.

```
        scanf("%d", &n);
        int sz = 0;
        FOR(i, 0, n)
```

```
{
        scanf("%d", &a[i]);
        p[a[i]]++;
        sz = max(sz, a[i]);
}
sz++;
int len = 0;
mulpoly(p, p, out, sz, sz, len);

FOR(i, 0, n)
{
        out[a[i] + a[i]]--;
}
FOR(i, 0, len) out[i] /= 2;
FOR(i, 1, len) sum[i] = sum[i - 1] + out[i];
sort(a, a + n);
ll ans = 0;
FOR(i, 0, n)
{
        // let a[i] is the largest side of the triangle so add
        // the number of ways we can take other two sides where their summation
        // is greater than a[i].
        ans += sum[len - 1] - sum[a[i]];
        // subtract cases where one side was less a[i], other was greater, as a[i] largest
        ans -= (ll)(n - i - 1) * i;
        // subtract cases where two sides are a[i], and other is some other
        ans -= (n - 1);
        // one is a[i], other two are larger than a[i]
        ans -= ((ll)(n - i - 1) * (n - i - 2)) / 2;
}
```

1.10 Problem 10

You want to generate an array of size N where each element is a random integer between 0 and K (inclusive). Count the number of possible arrays where the xor sum of the elements is strictly greater than 0. Output its value mod 30011.

Solution: FWHT problem. We build an array $a = 1, 1, 1..., 1$ of size $K + 1$. Now we raise it to power N using the transform. We can use bigmod-like calculation on the whole array to get the result in $O(logn)$ steps. Or we can first do forward trasnform on the input, then raise each value to power N, and do inverse transform.

```
void self_convolution(T a[], int n, int p)
{
    // n is input limit, power of 2
    // p is the power we need to raise to
    fwt(a, n); // forward transform
    for (int i = 0; i < n; i++)
            a[i] = bigmod(a[i],(ll)p,mod);
    ufwt(a, n);
}
```