1 Fast polynomial multiplication

1.1 Problem definition

Let $A(x) = \sum_{j=0}^{n} a_j \cdot x^j$ and $B(x) = \sum_{j=0}^{n} b_j \cdot x^j$ be two polynomials over the complex field $\mathbb{C}$. We are interested in finding a sequence of coefficients $(c_0, c_1, \ldots, c_{2n})$ such that $A(x) \cdot B(x) = \sum_{j=0}^{2n} c_j \cdot x^j$. It holds that:

$$A(x) \cdot B(x) = \left( \sum_{j=0}^{n} a_j \cdot x^j \right) \cdot \left( \sum_{k=0}^{n} b_k \cdot x^k \right) = \sum_{i,j=0}^{n} (a_j \cdot b_k) \cdot x^{j+k}$$

Formally, given the sequences $(a_0, a_1, \ldots, a_n), (b_0, b_1, \ldots, b_n)$ we are interested in finding a sequence $(c_0, c_1, \ldots, c_{2n})$ such that $\forall j : c_j = \sum_{k=0}^{\min(j,n)} a_k \cdot b_{j-k}$. We call the sequence $\vec{c}$ the convolution of $\vec{a}$ and $\vec{b}$ and use the notation $\vec{c} = \vec{a} \ast \vec{b}$.

1.2 Naive algorithm

Listing 1: Naive Polynomial multiplication

```plaintext
//Input : a[0..n], b[0..n]
//output : c[0..2n] the convolution of a[0..n] and b[0..n]
//Initialization
for i:=0 to 2n do
begin
    c[i] := 0
end;
//Computation
for i:=0 to n do
begin
    for j:= 0 to n do
    begin
        c[i+j] += a[i] * c[j];
    end;
end;
```

The complexity of the naive method is obviously $\Omega(n^2)$. 

1.3 Fast multiplication algorithm

We now show an algorithm for fast polynomial multiplication with time complexity $O(n \log(n))$. We notice the polynomials $A(x), B(x)$ are uniquely defined by their values over any set of $n+1$ elements from $\mathbb{C}$. Similarly, the polynomial $C(x) = \sum_{j=0}^{2n} c_j \cdot x^j$ is uniquely defined by it’s values over any set of $an+1$ elements from $\mathbb{C}$. We use the fact that for any $\alpha \in \mathbb{C}$ it must hold $C(\alpha) = A(\alpha) \cdot B(\alpha)$ to implement fast multiplication of polynomials using the FFT algorithm.

For simplicity we assume $n$ is of the form $n = 2^m - 1$, thus $2n < 2^{m+1} - 1$, and all the polynomials $A(x), B(x), C(x)$ are uniquely defined by their values over the group of unity root of order $2^{m+1}$. We assume all out FFT/IFFT operations (in the highest level) are done over this group (of course the recursive implementation of those algorithms uses smaller subgroups).

Listing 2: Fast polynomial multiplication

```plaintext
//Input : a[0..n], b[0..n]
//output : c[0..2n] the convolution of a[0..n] and b[0..n]
m := log(n+1);
k := exp(2,m+1);
y_a[0..k-1] := FFT(a[0..n],k);
y_b[0..k-1] := FFT(b[0..n],k);
for j:=0 to k-1 do
    y_c[j] = y_a[j] * y_b[j];
end;
c[0..k-1] := IFFT(y_c[0..k-1],k);
```

Complexity analysis  The complexity of each FFT/IFFT execution is $O(k \cdot \log(k))$ where $k \leq 4n$, thus the complexity is $O(k \cdot \log(k))$. The complexity of the loop is $O(k) = O(n)$. We conclude the complexity of this algorithm is $O(n \cdot \log(n))$. 
2 String matching

2.1 Problem definition

Given two strings \( A = (a_0, a_1, \ldots, a_n) \) and \( B = (b_0, b_1, \ldots, b_m) \) over the alphabet \( 1, -1 \) we are interested to check whether \( B \) is a substring of \( A \).

2.2 Efficient solution

The complexity of the naive algorithm is \( \omega(n \cdot m) \). We will show how this problem can be solved using fast convolution (aka fast polynomial multiplication) with complexity \( O(n \cdot \log n) \). For the sake of simplicity we assume \( n \geq m \).

Listing 3: Solution

```plaintext
//Input : a[0..n], b[0..m]
//output : 'True' if b[0..m] is a subsequence of a[0..n], and 'False' otherwise
//Initialization
for i:=0 to n do
begin
  a'[i] := a[n-i];
end;
//Computation
c[0..2(n+1)] = PolynomialMultiply(a'[0..n],b[0..m]);
for i:=0 to 2(n+1) do
begin
  if c[i] == m+1 return 'True'
end;
return 'False'
```

The complexity of the solution is obviously \( O(n \cdot \log n) \) whenever we use the fast polynomials multiplication algorithm. We will show this algorithm is correct.

Claim: The algorithm returns 'True' if and only if \( (b_0, b_1, \ldots, b_m) \) is a subsequence of \( (a_0, a_1, \ldots, a_n) \).

Proof. By definition of convolution (or polynomial multiplication) we have

\[
    c_j = \sum_{k=0}^{\min(j,m)} b_k \cdot a'_j-k = \sum_{k=0}^{\min(j,m)} b_k \cdot a_{n-j+k} = \sum_{k=0}^{\min(j,m)} \begin{cases} 
    0 & \text{if } (n-j+k) < 0 \\
    1 & \text{if } a_{n-j+k} = b_k \\
    -1 & \text{if } a_{n-j+k} \neq b_k 
\end{cases}
\]

We notice the \( c_j \leq m+1 \) for any value of \( j \) as it has at most \( m+1 \) summand, each equals at most to 1. We conclude that if \( A[(n-l), \ldots, (n-l)+m] = B[0 \ldots m] \) then

\[
    c_l = \sum_{k=0}^{\min(l,m)} b_k \cdot a_{(n-l)+k} = m+1
\]

and for any \( l \) not satisfying the equivalence above \( c_l < m+1 \), as at least one of it’s summands is 0 or -1, and in particular strictly smaller then 1.

\[
\square
\]